МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ
БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)» (СГАУ)

ТЕХНОЛОГИИ ПРОМЫШЛЕННОГО ПРОГРАММИРОВАНИЯ

Электронный учебно-методический комплекс
по дисциплине в LMS Moodle

САМАРА
2012

УДК 681.3.06

Автор-составитель: **Баландин Александр Васильевич**

Технологии промышленного программирования [Электронный ресурс] : электрон. учеб.-метод. комплекс по дисциплине в LMS Moodle / Минобрнауки России, Самар. гос. аэрокосм. ун-т им. С. П. Королева (нац. исслед. ун-т); авт.- сост. А. В. Баландин. - Электрон. текстовые и граф. дан. - Самара, 2012. – 1 эл. опт. диск (CD-ROM).

В состав учебно-методического комплекса входят:
1. Учебное пособие.
2. Тематические планы практических (семинарских) занятий.
3. Методические указания и задания к лабораторным работам.
4. Вопросы для подготовки к экзамену.
5. Тесты для итогового контроля знаний.
6. Рабочая программа дисциплины.

УМКД «Технологии промышленного программирования» предназначен для студентов факультета информатики, обучающихся по направлению подготовки магистров 010300.62 «Фундаментальная информатика и информационные технологии» в 11 семестре.

УМКД разработан на кафедре программных систем.

Федеральное агентство по образованию

**Государственное образовательное учреждение
высшего профессионального образования
"Самарский государственный аэрокосмический
университет имени академика С.П. Королёва"**

А.В. Баландин

Средства и методы разработки программных систем реального времени

Часть 1

Язык программирования C

Учебное пособие

Самара 2012

УДК 681.3.066
ББК 32.973.26-018.2

Баландин А.В. Средства разработки программного обеспечения систем реального времени. Учебное пособие: В 4 ч. Ч.1. **Язык программирования С**. - Самар. гос. аэрокосм. ун-т. Самара, 2012. 77 с.

В пособии приведено описание алгоритмического языка С как базового средства программирования компонентов многопоточного, параллельного и распределенного программного обеспечения систем, функционирующих в режиме реального времени, в операционной системе QNX Neutrino (QNX 6).

Пособие предназначено для студентов, обучающихся по направлению 010300.68 «Фундаментальная информатика и информационные технологии», изучающих дисциплину «Технологии промышленного программирования».

ОГЛАВЛЕНИЕ

Предисловие

Использование вычислительной техники в системах промышленной автоматизации (СПА), системах автоматизации научных исследований и комплексных испытаний образцов новой техники (АСНИ), а также встраивание микропроцессоров в различные виды оборудования с целью интеллектуализации управления сложными устройствами (*встроенные системы*) послужило причиной появления и развития автоматизированных систем обработки информации и управления, функционирующих в таком режиме, когда необходимо взаимодействовать с внешними по отношению к системе процессами в темпе, согласованном с темпом протекания этих процессов. Такой режим называют *режимом реального времени*, а системы - *системами реального времени* (СРВ).

Принципиальной особенностью разработки программного обеспечения СРВ (приложений реального времени - ПСРВ) является то, что режим реального времени может быть эффективно обеспечен, если структурно ПСРВ проектируется как совокупность параллельно выполняемых и взаимодействующих задач (*процессов*). При разработке ПСРВ требуется выбирать количество параллельных процессов и способы их взаимодействия, устанавливать приоритеты и дисциплины диспетчеризации процессов для разделения ресурсов процессора, а для распределенных СРВ ещё необходимо управлять размещением процессов по узлам компьютерной или промышленной сети. При этом приходится учитывать, что выбор структуры и параметров ПСРВ существенно влияет на эффективность функционирования СРВ при тех же технических средствах. Все это делает разработку программного обеспечения СРВ сложным и трудоёмким процессом, эффективная реализация которого напрямую зависит от выбора средств программирования.

В настоящее время разработка ПСРВ осуществляется, как правило, на основе широкого спектра операционных систем реального времени (ОСРВ) с использованием их базовых или специализированных средств программирования. Во всех ОСРВ базовым языком программирования ("встроенного в ОС") является алгоритмический язык программирования С. Поэтому в качестве универсальных средств разработки ПСРВ принято рассматривать средства выбранной ОСРВ и язык программирования С. Даже если разработка осуществляется специализированными средствами (например, средствами визуального объектно-ориентированного программирования), определенные части ПСРВ могут потребовать программирования на основе универсальных средств.

Настоящее учебное пособие посвящено рассмотрению базовых средств программирования приложений реального времени операционной системы реального времени QNX, включая язык С.

Учебное пособие состоит из двух частей. В первой части пособия описывается алгоритмический язык программирования С. При этом не ставится задача обучения приемам программирования на языке С, а внимание прежде всего уделяется быстрому вхождению в синтаксис и семантику языка. При

наличии навыка программирования на языке C, первую часть пособия можно рассматривать в качестве справочника по языку C. Вторая часть посвящена средствам многопоточного, параллельного и распределенного программирования операционной системы QNX. Для тех, кто имеет навык программирования параллельных процессов в ОС UNIX, LINUX или других POSIX-ориентированных ОС, многие средства ОС QNX окажутся знакомыми.

ЧАСТЬ 1. ЯЗЫК С

ПРОГРАММА НА ЯЗЫКЕ СИ

Структура программы

Исполняемый файл программы, написанной на языке С (программный файл), является результатом компиляции и компоновки файлов проекта программы. Под проектом программы на языке С понимается совокупность файлов определенных типов, необходимых для получения программного файла. В общем случае проект программы включает в себя файлы следующих типов:

- *исходные файлы* - текстовые файлы программы (расширение - **`.c`**);
- *заголовочные файлы* - текстовые файлы программы (расширение - **`.h`**);
- *файлы объектных модулей* – результаты компиляции исходных файлов (расширение - **`.o`**);
- *файлы библиотек объектных модулей* – архивы объектных модулей (для статических библиотек расширение - **`.a`**, для динамических - **`.so`**).

Текст программы может располагаться в нескольких исходных файлах, что удобно при написании больших программ. Каждый исходный файл может независимо от других быть обработан компилятором. В процессе компиляции исходный текст программы может предварительно объединяться с содержимым *заголовочных файлов*. Результатом компиляции исходного файла является файл *объектного модуля*.

Исходный файл в общем случае может включать в себя произвольное число программных объектов: директив препроцессора, объявлений или определений типов, переменных, функций.

Объявление типа позволяет присвоить собственное имя некоторому базовому или составному типу языка С.

Объявление переменной специфицирует класс памяти, имя и тип переменной.

Определение переменной заключается в объявлении переменной и спецификации её начального значения.

Объявление функции специфицирует её класс памяти, имя, формальные параметры и тип возвращаемого значения.

Определение функции заключается в объявлении функции и спецификации тела функции (функционального блока). Функциональный блок может содержать в себе простые блоки и исполняемые операторы. Простые блоки в свою очередь могут также содержать простые блоки и операторы.

Класс памяти определяет способ выделения памяти программному объекту. Если память выделяется программному объекту при загрузке программы на

выполнение и закрепляться за ним до конца выполнения программы, то такая память относится к классу *статической памяти*. Если память не выделяется переменной сразу при загрузке программы, а выделяется по мере необходимости и на ограниченное время, то такая память относится к классу *динамической памяти*. Статическая память может быть типа **extern** или **static**. Динамическая память может быть типа **auto** или **register**. Выбор типа статической памяти позволяет управлять областью видимости имени объекта в различных файлах программы. Выбор типа динамической памяти уточняет свойства выделяемой объекту физической памяти.

Программа на языке С имеет два уровня размещения программных объектов – внешний и внутренний. Границами *внешнего уровня* являются границы файлов, проекта программы. Границами *внутреннего уровня* являются границы функциональных блоков функций программы. Если определение программного объекта находиться вне функционального блока, то это означает, что он определен и располагается на внешнем уровне (внутри границ файла его содержащего). В противном случае – программный объект считается определенным и находящимся на внутреннем уровне программы.

Если переменная или функция объявляется или определяется на внешнем уровне, то их класс памяти может быть только статический обоих типов **static** или **extern**. Переменная, определяемая на внутреннем уровне, может иметь как статическую память типа - **static**, так и динамическую память обоих типов **auto** или **register**. Кроме того, на внутреннем уровне могут находиться объявления переменных или функций, имеющих статический класс памяти типа **extern**. Объявление на внутреннем уровне функций с классом памяти **static** не допускается.

Класс памяти влияет на так называемую *область видимости* программного объекта в программе. Объект считается видимым в некоторой части программы, если в ней возможен доступ к объекту по его имени. Совокупность всех таких частей программы составляет область видимости программного объекта. Класс памяти влияет на возможность видимости программного объекта за пределами того файла, в котором он определен.

Область видимости переменной или функции, находящейся на внешнем уровне и имеющей класс памяти **static**, начинается с места объявления и простирается до конца файла. За пределами содержащего их файла имя переменной или функции не может быть видимым.

Переменная или функция, определенная в некотором файле на внешнем уровне с классом памяти **extern**, может быть объявлена и стать видимой в любом другом файле программы, в котором нет другой переменной или функции с таким же именем. Видимость переменной или функции, объявленной на внешнем уровне в некотором файле, начинается с места объявления в файле и простирается до конца файла.

Область видимости переменной или функции, объявленной на внутреннем уровне независимо от класса памяти ограничивается содержащим его блоком и простирается с места объявления в блоке и до конца блока.

Если один блок вложен в другой блок, то объявление имени программного объекта во внутреннем блоке отменяет действие конкурирующего объекта (вытесняет), объявленного с тем же именем во внешнем (охватывающем) блоке или на внешнем уровне. Этот механизм называется локальным переобъявлением программных объектов, объявленных в охватывающем блоке. Локальное переобъявление имеет силу во внутреннем блоке, а действие первоначального объявления восстанавливается, когда выполнение программы возвращается в охватывающий блок.

Область видимости создаваемых типов подчиняется тем же правилам, что и область видимости переменных.

С классом памяти связано также понятие времени жизни объекта. *Время жизни* объекта – это период времени в течение выполнения программы, когда объект обладает памятью. Если объект имеет статическую память, то он обладает *глобальным временем* жизни. Объект с динамической памятью имеет *локальное время* жизни и существует только в те периоды времени, когда выполнение программы осуществляется в программном блоке, в котором объект объявлен. При входе программы в блок объекту выделяется память и устанавливается начальное значение. После выхода программы из блока объект перестает существовать (теряет память и текущее значение).

Выполнение программы

Программа на языке С может состоять из одной или нескольких функций. Если функций более одной, то они могут находиться в одном или разных файлах проекта программы. Среди всех функций выделяется одна функция, которая называется главной. Имя главной функции не выбирается, а определено в языке С как **`main`**. Каждая программа на языке С содержит только одно определение функции с именем **`main`**. Если программа состоит из одной функции, то этой функцией является функция **`main`**. Минимальной программой на языке С является программа вида:

```
void main(void){}//Пустая программа - начинается и завершается
```

Запуск программного файла на выполнение равносилен вызову функции **`main`**. В процессе её выполнения могут вызываться другие функции, те в свою очередь также могут вызывать функции и т.д. Все функции начинают выполняться с первого оператора. Ни одна из функций не может вызвать функцию **`main`**. Если достигается завершение функции **`main`**, то завершается и вся программа. Программа, однако, может завершиться и в других функциях путем вызова стандартных функций, предназначенных для системного завершения программы (например, **`exit`**).

Препроцессор языка С

Язык С имеет специальные средства для управления исходным текстом программы перед тем, как он будет преобразован компилятором в объектный файл. Этими средствами являются директивы препроцессора языка С. Директивы препроцессора позволяют сделать текст программы параметрически настраиваемым на различные условия применения программы. Признаком директивы в тексте программы является символ <#>, который должен быть первым в строке, содержащей директиву. Директивы обрабатываются препроцессором. Препроцессор представляет собой макрогенератор, осуществляющий обработку исходного файла и преобразование исходного текста программы на предварительной фазе компиляции. Компилятор сам вызывает препроцессор, однако препроцессор может вызываться и автономно.

Рассмотрим основные директивы препроцессора.

Директива **#include**

Директива имеет две синтаксические формы:

1) **#include** "*имя_файла*"

2) **#include** <*имя_файла*>

Директива **#include** предназначена для замены её на включаемое в текст программы содержимое указанного в директиве файла. Препроцессор обрабатывает затем включаемый файл таким же образом, как если бы этот файл целиком входил в состав исходного файла в точке, где записана заменяемая директива. Как правило, включаемые файлы содержат в себе объявления и определения символических констант, типов, переменных и функций. Эти объявления и определения обычно располагаются в начале исходного текста программы. Поэтому такие файлы получили название заголовочных файлов и имеют стандартное расширением **.h** (от английского слова header).

Если используется директива в первой форме и задано полное имя файла (с путем), то файл ищется только в указанном каталоге. Если указано короткое имя файла, то поиск включаемого файла начинается в текущем каталоге, а при отсутствии файла в текущем каталоге поиск продолжается в каталогах, перечисленных в командной строке вызова компилятора, и, наконец, в каталогах, перечисленных в системной переменной окружения с именем **PATH**.

Если используется директива во второй форме, то поиск включаемого файла начинается в каталогах, перечисленных в командной строке вызова компилятора, а затем в каталогах, перечисленных в системной переменной окружения с именем **PATH**.

Примеры:

```
#include "/root/home/define.h" //Поиск только в указанном каталоге
#include "define.h" //Поиск в текущем или перечисленных каталогах
#include <stdio.h> //Поиск только в перечисленных каталогах
```

Директива **#define**

Директива имеет две синтаксические формы, называемые макроопределениями:

1) `#define` <*идентификатор*> <*текст*>

2) `#define` <*идентификатор*>(<*список_параметров*>)<*текст*>

Директива `#define` предписывает препроцессору заменить все вхождения *идентификатора* в исходном тексте, лежащем ниже директивы, на *текст*, заданный в макроопределении. Идентификатор заменяется лишь в том случае, если он представляет собой отдельную лексему, т.е. окружен символами-разделителями (например, пробелами). Если текст не умещается в строке, то необходимо завершить текущую строку символом < \ > и перейти на следующую строку. Текст в макроопределении может быть опущен, тогда соответствующие идентификатору лексемы будут удалены из исходного текста.

Если в макроопределении задан список параметров (вторая форма), то он должен содержать один или более отличающихся друг от друга идентификаторов-параметров, разделенных запятыми. Соответствующие параметрам лексемы в тексте макроопределения рассматриваются как формальные параметры текста, которые отмечают позиции, куда должны быть подставлены аргументы макровызова, который имеет вид:

<*идентификатор*>(<*список_аргументов*>)

В макровызове следом за идентификатором записываются в круглых скобках через запятую аргументы, соответствующие формальным параметрам. Подставляемый вместо макровызова текст модифицируется путем замены лексем каждого формального параметра на соответствующий фактический текстовый аргумент.

Внутрь текста макроопределения могут входить другие макроопределения. Это учитывается таким образом, что после того как выполнен соответствующий макровызов, вставленный текст вновь просматривается для поиска других макроопределений. При этом просмотр выполняется таким образом, что макроопределения подобные ниже представленному:

```
#define x x
```

не приводят к зацикливанию препроцессора.

Примеры:

```
#define ONE 1
#define WIDTH 80
#define LENGTH WIDTH+20//WIDTH будет заменен на 80
#define F(x) sin(x)
#define HUGE Если текст не умещается в строке, то необходимо \
             завершить текущую строку символом \
             и перейти на следующую строку.
```

В тексте макроопределения можно применять две специальные препроцессорные операции:

<##> - *склейка лексем*;

<#> - *заключение аргумента в двойные кавычки*.

В тексте директивы `#define` две лексемы могут быть "склеены" вместе. Для этого между ними нужно поставить знак `##` (слева и справа от `##` допустимы пробельные символы). Препроцессор объединяет такие лексемы в одну. Например, макроопределение

```
#define VAR(i,j) i ## j
```

при макровызове `VAR(x,6)` образует идентификатор `x6`.

Если необходимо, чтобы аргумент макровызова в тексте был заключен в двойные кавычки, то перед соответствующим формальным параметром в тексте макроопределения необходимо поставить символ **#**. Например, макроопределение

```
#define PUTS(STR) puts(#STR)
```

при макровызове `PUTS(line)` образует текст `puts("line")`.

Директива **#undef**

Действие директивы `#define` в исходном тексте программы можно ограничить. Для этого применяется директива `#undef`. Директива имеет следующую синтаксическую форму:

```
#undef <идентификатор>
```

Директива отменяет действие макроопределения с именем, соответствующим указанному идентификатору.

Примеры:

```
#define WIDTH 80
…
#define FUNC(x) sin(x)
…
#undef WIDTH
…
#undef FUNC
```

Не является ошибкой применение директивы `#undef` к идентификатору, который ранее не был определен или действие которого уже отменено. Это, например, можно использовать для обеспечения гарантии того, что макроопределение с таким идентификатором, которое оказалось бы во включаемых файлах, в данной части программы действовать не будет.

Директивы условной компиляции

Директивы условной компиляции позволяют управлять выборочной компиляцией частей исходного файла, связывая их компиляцию с проверкой

выполнения заданных условий. Директивы условной компиляции составляют вместе взаимосвязанную конструкцию, имеющую следующую синтаксическую форму:

```
#if <ограниченное_константное_выражение>
<текст>

#elif <ограниченное_константное_выражение>
<текст>

#elif <ограниченное_константное_выражение>
<текст>

…

#elif <ограниченное_константное_выражение>
<текст>

#else
<текст>

#endif
```

Каждой директиве `#if` в том же исходном файле должна соответствовать завершающая ее директива `#endif`. Между директивой `#if` и `#endif` допускается произвольное количество директив `#elif` (в том числе ни одной) и не более одной директивы `#else`. Если директива `#else` присутствует, то между ней и директивой `#endif` на данном уровне вложенности не должно быть других директив `#elif`. Каждая из директив (кроме `#endif`) управляет участком текста (<*текст*>), заключенного между текущей и последующей директивой. Текст может занимать более одной строки. Обычно это программный текст, однако это не обязательно, препроцессор можно использовать для обработки произвольного текста.

Препроцессор выбирает для обработки только один из всех участков текста, заключенных между директивами `#if` и `#endif`. Препроцессор выбирает текст для обработки на основе вычисления ограниченного константного выражения, следующего за каждой директивой `#if` или `#elif`. Выбирается текст, следующий за ограниченным константным выражением со значением истина (не нуль), вплоть до ближайшей директивы `#elif`, `#else`, или `#endif`, ассоциированной с данной директивой `#if`. Если ни одно ограниченное константное выражение не истинно, то препроцессор выбирает текст, следующий за директивой `#else`. Если же директива `#else` отсутствует, то никакой текст не выбирается.

Если выбранный текст содержит директивы препроцессора (в том числе и директивы условной компиляции), эти директивы выполняются.

Обработанный препроцессором текст передается на компиляцию. Участок текста, не выбранный препроцессором на стадии препроцессорной обработки, не компилируется.

Ограниченное константное выражение (см. раздел "Константные

выражения") не может содержать операцию `sizeof`, операцию приведения типа, константы перечисления и плавающие константы. Но может содержать сугубо препроцессорную операцию

`defined(<`*идентификатор*`>)`

Операция `defined` дает истинное (не нулевое) значение, если заданный <*идентификатор*> в данный момент определен; в противном случае выражение ложно (равно нулю). Следует помнить, что идентификатор, определенный без значения, тем не менее, рассматривается как определенный. Операция `defined` может использоваться в сложном выражении в директиве `#if` неоднократно.

Пример:

```
#if defined(CANAL_1)
     in(ch1);
#elif defined(CANAL_2)
     in(ch2);
#else
     in(ch0);
#endif
```

Комментарии

Средства комментирования программы не относятся к средствам препроцессора, они обрабатываются непосредственно компилятором. Но так как они участвуют в модификации текста при компиляции, то рассматриваются вместе со средствами препроцессорной обработки. Обработка комментариев заключается в том, что они просто удаляются из текста программы.

Существует два вида комментариев:

- блок комментария;
- комментарий в строке.

Блок комментария рассматривается компилятором как один пробельный символ, поэтому комментарии допускается использовать везде, где можно использовать пробельные символы. Блок комментария имеет следующий вид:

/*
<*Текст комментария*>
***/**

Комментарий может занимать более одной строки. Заметим, что внутри блока комментария недопустима комбинация символов ***/**. Это означает, что блоки комментарии не могут быть вложенными. Для того, чтобы при необходимости избежать проблемы вложенных комментариев, можно для комментирования воспользоваться конструкцией условной компиляции вида:

```
#if 0
<Текст комментария>
#endif
```

Форма комментария в строке позволяет закончить текущую строку комментарием. Комментарий в строке может начаться с любого места текущей строки и продолжается до конца строки:

<Текст>//комментарий до конца строки

Примеры:

```
1) /* это комментарий-блок*/
2) /**************************
      это комментарий-блок
   **************************/
3) main()//это основная функция (комментарий в строке)
```

Элементы языка С

Алфавит

Буквы и цифры

Алфавит языка С включает буквы латинского алфавита (A, B, C, D,.., Z, a, b, c, d,.., z) и арабские цифры (0,1,2,…, 9) , а также пробельные и специальные символы. Причем прописные и строчные буквы считаются разными.

Пробельные символы

Символ пробел, а также управляющие символы табуляция, перевод строки, возврат каретки, новая страница, вертикальная табуляция и новая строка называются пробельными, так как имеют то же назначение, что и символ пробел. Они используются в исходном тексте как разделители лексем. Любое количество подряд идущих пробельных символов рассматривается компилятором как один пробельный символ.

Символ `ctrl/z` (шестнадцатеричный код – `1A`) рассматривается в исходном тексте как конец файла. Обнаружив его, компилятор завершает обработку файла.

Специальные символы

Специальные символы предназначены для представления в тексте программы пробельных и непечатных (управляющих) символов. Специальный символ представляется в виде обратного слэша, за которым следует либо буква, либо знаки пунктуации, либо комбинация цифр. Ниже в таблице приведен список специальных символов языка С.

Специальный символ	Шестнадцатеричное значение в коде ASCII	Наименование
\n	**0A**	Новая строка
\t	**09**	Горизонтальная табуляция
\v	**0B**	Вертикальная табуляция

\b	**08**	Забой
\r	**OD**	Возврат каретки
\f	**OC**	Новая страница
\a	**07**	Звуковой сигал
\'	**2C**	Апостроф
\"	**22**	Двойная кавычка
****	**5C**	Обратный слэш
\ddd		Байтовое значение в восьмеричном представлении
\xdd		Байтовое значение в шестнадцатеричном представлении
\?	**3F**	Знак вопроса

Если обратный слэш предшествует символу, не входящему в приведенный список, то обратный слэш игнорируется, а символ воспринимается обычным образом. Например, сочетание `\h` в строковой или символьной константе воспринимается просто как символ `h`.

Конструкция `\ddd` позволяет задать произвольное байтовое значение как последовательность от одной до трех восьмеричных цифр. Конструкция `\xdd` позволяет задать произвольное байтовое значение как последовательность от одной до двух шестнадцатеричных цифр. Например, символ забой в коде ASCII может быть задан как `\010` или `\x08`.

Специальные символы позволяют посылать управляющие последовательности на внешние устройства. Например, код `\033` (символ `ESC` в коде ASCII часто используется как первый символ команд управления терминалом и принтером.

Помимо специальных символов, обратный слэш (****) используется также в качестве признака продолжения символьных строк и препроцессорных макроопределений. Если символ новой строки непосредственно следует за обратным слэшем, то комбинация ****<*символ_новой_строки*> рассматривается как продолжение предыдущей строки.

Константы

Константа – это *число*, *символ* или *строка символов*. Различают четыре типа констант: целые, с плавающей точкой, символьные константы и символьные строки.

Целые константы

Целая константа – это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целое значение.

Десятичные константы	Восьмеричные константы	Шестнадцатеричные константы
10	012	0xA или 0Xa
132	0204	0X84
32179	076663	0x7dB3 или 0X7dB3

Целые константы всегда специфицируют положительные значения. Знак минус перед константой означает формирование константного выражения и выполнение над ней унарной арифметической операции отрицания. Каждая целая константа имеет тип, определяющий её представление в памяти.

Десятичные целые константы могут иметь тип `int` или `long`. *Восьмеричные* и *шестнадцатеричные* константы в зависимости от размера могут иметь тип `int`, `unsigned int`, `long`, `unsigned long`. Ниже в таблице приведены диапазоны значений целых констант и их тип представления в памяти для `int` длиной 16 разрядов и `long` длиной 32 разряда.

Десятичные константы	Восьмеричные константы	Шестнадцатеричные константы	Тип
0 ÷ 32767	0 ÷ 77777	0x0 ÷ 0x7FFF	**int**
-	0100000 ÷ 0177777	0x8000 ÷ 0xFFFF	**unsigned int**
32767 ÷ 2147483647	02000001 ÷ 017777777777	0x10000 ÷ 0x7FFFFFFF	**long**
-	020000000000 ÷ 030000000000	0x80000000 ÷ 0xFFFFFFFF	**unsigned long**

Можно явно управлять типом представления константы в памяти, не зависимо от величины константы, используя специальные суффиксы. Для того, чтобы задать константе тип `long`, к константе приписывается суффикс `l` или `L`. Для восьмеричных и шестнадцатеричных констант можно еще использовать суффикс `u` или `U`, чтобы типизировать её как беззнаковое целое. Оба суффикса можно использовать одновременно, Например, 01234`LU` – константа типа `unsigned long`.

Константы с плавающей точкой

Константа с плавающей точкой - это действительное десятичное положительное число. Оно может включать целую, дробную часть и экспоненту. Либо целая, либо дробная часть константы может быть опущена, но не обе сразу. Либо десятичная точка с дробной частью, либо экспонента может быть опущена, но не обе сразу. Знак минус перед константой рассматривается как унарная операция. Примеры: -15.7525e3, 1.575E-7, -.123, 567., 0.00001

Все константы с плавающей точкой имеют тип `double`.

Символьные константы

Символьная константа – это буква, цифра, знак пунктуации или специальный символ, заключенный в апострофы. Значение символьной константы равно коду представляемого ею символа. Символ может быть любым, за исключением символов апостроф <'>, обратный слэш <\> и новая строка. Для их представления в качестве символьной константы необходимо вставить перед ними символ обратный слэш: `\'`, `\\`, `\n`.

Символьные константы имеют тип `int`. Младший байт хранит код символа, а старший байт – знаковое расширение младшего байта.

Примеры:

```
'X' – символьная константа со значением кода символа X.
'\'' – символьная константа со значением кода символа апостроф.
```

Символьные строки

Символьная строка – это последовательность символов, заключенная в двойные кавычки. Символьная строка рассматривается как массив символов типа `char`, который завершается двоичным нулем `\0`. Каждый символ символьной строки хранится в отдельном байте оперативной памяти. Нулевой символ автоматически добавляется в качестве последнего байта символьной строки и служит признаком её конца. При использовании в символьной строке символов двойного применения необходимо включать их в форме спецсимвола.

Примеры:

```
"это символьная строка"
"первый\\второй"
"\"C\"" - в результате будет "C"
"" - пустая строка
"длинные строки могут быть разделены \
на несколько частей”  транслятор рассмотрит это как одну строку."
```

Идентификаторы

Идентификаторы - это имена переменных, функций и меток, используемых в программе. Идентификатор вводится в объявлении переменной или функции, либо в качестве метки оператора. После этого его можно использовать в последующих операторах программы. Идентификатор - это последовательность из одной или более латинских букв, цифр и символов подчеркивания, которая начинается с буквы или символа подчеркивания. Допускается любое число символов в идентификаторе, однако только первые 32 символа рассматриваются компилятором как значащие. Если первые 32 символа у двух идентификаторов совпадают, компилятор рассматривает их как один и тот же идентификатор. Компоновщик также распознает 32 символа в именах глобальных переменных.

При использовании символов подчеркивания в качестве первых символов идентификаторов необходимо соблюдать осторожность, поскольку такие

идентификаторы могут совпасть (войти в конфликт) с именами "скрытых" библиотечных функций.

Компилятор рассматривает буквы верхнего и нижнего регистров как различные символы. Поэтому можно создавать идентификаторы, которые совпадают по буквам, но различаются регистром букв. Например, каждый из следующих идентификаторов является уникальным: `add`, `ADD`, `Add`, `aDD`.

Компилятор не допускает использования идентификаторов, совпадающих по написанию с ключевыми словами. *Ключевые слова* – это предопределенные идентификаторы, которые имеют специальное значение для компилятора. Имена объектов программы не могут совпадать с ключевыми словами. Например, идентификатор `while` недопустим, однако идентификатор `WHile` – допустим.

ОБЪЯВЛЕНИЯ

Ранее были введены понятия объявления и определения. В этом разделе описываются формат и составные части объявлений переменных, функций и типов.

Объявления в языке С имеют следующий синтаксис:

<класс_памяти> <спецификация_типа> <список_описателей>;

Класс памяти определяет время жизни и область действия переменной в программе и может специфицироваться как `auto`, `register`, `static`, `extern`. *Спецификация типа* вместе со списком описателей специфицируют имена и типы объявляемых объектов. *Список описателей* содержит перечисление через запятую описателей вида:

<описатель> = <инициализатор>

Описатель - это идентификатор простой переменной либо более сложная конструкция при объявлении переменной составного типа (массив, структура, объединение). *Инициализатор* - значение или совокупность значений, присваиваемых переменной при объявлении. Инициализатор может отсутствовать (вместе со знаком < = >). Тогда значение переменной задается по умолчанию (неявно). С позиции надежности программирования целесообразно полагать, что значение переменной в этом случае не определено и требуется явное присвоение ей значения перед первым использованием переменной в программе.

Спецификация класса памяти или спецификация типа в объявлении могут быть опущены и использованы значения по умолчанию.

Все переменные в языке С должны быть явно объявлены перед их использованием, за исключением функций, которые объявляются неявно, если их вызов следует до объявления.

В языке С определен набор базовых типов данных. Новые типы данных (абстрактные типы) можно создавать в программе посредством их объявления на основе уже определенных типов данных. Спецификация типа позволяет задавать для объекта либо базовый тип, либо абстрактный тип, либо структурный тип,

либо тип объединение.

Базовые типы данных

В языке С реализован следующий набор базовых типов данных:

Базовые типы	Название типа	Полная синтаксическая форма	Краткая синтаксическая форма	Размер памяти
Целые	*Знаковый символьный*	**signed char**	**char**	1 байт
	Знаковый целый	**signed int**	**int, signed**	зависит от платформы
	Знаковый короткий целый	**signed short int**	**short, signed short**	2 байта
	Знаковый длинный целый	**signed long int**	**long, signed long**	4 байта
	Беззнаковый символьный	**unsigned char**	-	1 байт
	Беззнаковый целый	**unsigned int**	**unsigned**	зависит от платформы
	Беззнаковый короткий целый	**unsigned short int**	**unsigned short**	2 байта
	Беззнаковый длинный целый	**unsigned long int**	**unsigned long**	4 байта
Плавающие	*Плавающий одинарной точности*	**float**	-	4 байта
	Плавающий двойной точности	**double**	-	8 байт
	Длинный плавающий двойной точности	**long double**	-	10 байт
Прочие	*Пустой (неопределенный)*	**void**	-	
	Перечислимый	**enum**	-	

Выбор для данного одного из базовых целого или плавающего типов определяется диапазоном возможных значений этого данного в процессе выполнения программы. Например, тип `char` позволяет хранить данное в диапазоне -128÷127, а `unsigned char` – в диапазоне 0÷255.

Описатели

Синтаксис описателей

В простейшем случае, когда объявляется не структурная переменная, описатель представляет собой идентификатор – имя переменной. Для объявления структурных переменных идентификатор дополняется специальными признаками *массива* – [], *функции* – (), или *указателя* - *.

Синтаксис описателей задается следующими рекурсивными правилами:

<*идентификатор*>
<*описатель*> []
<*описатель*> [<*константное_выражение*>]

*<*описатель*>
<*описатель*> ()
<*описатель*>(<*список_типов_аргументов*>)
(<*описатель*>)

Для объявления массива значений специфицированного типа, либо функции, возвращающей значение специфицированного типа, либо указателя на значение специфицированного типа, идентификатор дополняется, соответственно, квадратными скобками (справа) , круглыми скобками (справа) или звездочкой (слева). В дальнейшем будем называть квадратные скобки, круглые скобки и звездочку признаками типа массив, функция и указатель, соответственно.

Следующие примеры иллюстрируют простейшие формы описателей:

```
int Iist[20] -массив list значений целого типа;
char .*cp -указатель cp на значение типа char;
doubIe func() -функция func, возвращающая значение типа doubIe.
```

Интерпретация составных описателей

Составной описатель-это идентификатор, дополненный более чем одним признаком типа массив, указатель или функция.

С одним идентификатором можно образовать множество различных комбинаций признаков типа массив, указатель или функция. Некоторые комбинации недопустимы. Например, массив не может содержать в качестве элементов функции, а функция не может возвращать массив или функцию.

При интерпретации составных описателей сначала рассматриваются квадратные скобки и круглые скобки, расположенные справа от идентификатора. Квадратные и круглые скобки имеют одинаковый приоритет. Они интерпретируются слева направо. После них справа налево рассматриваются звездочки, расположенные слева от идентификатора. Спецификация типа рассматривается на последнем шаге после того, как описатель уже полностью проинтерпретирован.

Для изменения действующего по умолчанию порядка интерпретации описателя можно использовать внутри него круглые скобки.

Правило интерпретации составных описателей предписывает их чтение "изнутри-наружу". Начать интерпретацию нужно с идентификатора и проверить, есть ли справа от него открывающие квадратные или круглые скобки. Если они есть, то проинтерпретировать правую часть описателя. Затем следует проверить, есть ли слева от идентификатора. звездочки, и, если они есть, проинтерпретировать левую часть. Если на какой-либо стадии интерпретации справа встретится закрывающая круглая скобка (которая используется для изменения порядка интерпретации описателя), то необходимо сначала полностью провести интерпретацию внутри данной пары круглых скобок, а затем продолжить интерпретацию справа от закрывающей круглой скобки.

На последнем шаге интерпретируется спецификация типа. После этого тип

объявленного объекта полностью известен.

Следующий пример иллюстрирует применение правила интерпретации составных описателей. Последовательность шагов интерпретации пронумерована.

```
char *(*(*var)())[10];
  7   6 4 2 1   3     5
1. Идентификатор var объявлен как…
2. Указатель на…
3. Функцию, возвращающую…
4. Указатель на…
5. Массив из 10 элементов, которые являются…
6. Указателями на…
7. Значения типа char.
```

В приведенных ниже примерах обратите внимание на то, как применение круглых скобок может изменять смысл объявлений.

```
int *var[5];//var - массив указателей на значения типа int.

int (*var)[5];//var - указатель на массив значений типа int.

1ong*var(); //var - функция, возвращающая указатель на значение
              типа long.

long (*var)();//var - указатель на функцию типа long.

struct both{
           int a;
           char b;
           }(*var[5])();/* var - массив указателей на функции
                           типа структуры both.*/

void(*signal(int sig,void(*act)(int)))(int);/* signal- функция,
     двух аргументов - int и указатель на функцию с одним
     аргументом int и не возвращающую значение, возвращающая
     значение указателя на функцию, с одним аргументом int и не
     возвращающую значение */
```

Описатели с модификаторами

В составе описателей могут использоваться ключевые слова `const` и `volatile`, называемые *модификаторами*. Модификаторы позволяют при объявлении переменных уточнить их свойства и порядок использования в программе. Информация, которую несут модификаторы, учитывается компилятором в процессе обработки исходной программы и генерации программного кода.

Модификатор `const` не допускает каких либо действий в программе по изменению начального значения объявляемой переменной. Значение указателя, объявленного с модификатором `const`, не может изменяться, в отличие от значения объекта, на который он указывает.

Применение модификатора `const` заставляет компилятор выявлять в тексте программы попытки изменить начальное значение переменных, объявленных с

модификатором const, а при обнаружении – компилятор выдаёт сообщение об ошибке. Это важно, например, когда предполагается, что объекты могут находиться в постоянной памяти (ПЗУ).

Модификатор volatile противоположен по смыслу модификатору const. Он указывает на то, что значение объекта может быть асинхронно изменено внешним воздействием, например, обработчиком прерываний, либо в качестве объекта выступает порт обмена с внешним устройством. Для выражений, содержащих объекты типа volatile, компилятор не будет применять методы оптимизации, а текущие значения объектов не будут для промежуточного хранения загружаться в машинные регистры, а браться непосредственно из источника.

Модификаторы const, volatile уточняют либо непосредственно переменную (идентификатор), либо указываемый объект (звездочку), расположенную непосредственно справа от модификатора. Если уточняется идентификатор, то модифицируется тип объекта, именуемого этим идентификатором. Если уточняется признак указателя (звездочка), то модифицируется тип объекта, на который указывает указатель. Например,

```
int const var=5; // var – неизменяемый объект типа int
int const *p; // p - указатель на неизменяемый объект типа int
int const * const cp=&var; /* cp - неизменяемый указатель на
                              неизменяемый объект типа int */
```

Допускается размещение модификаторов и перед спецификацией типа:

```
const int var=5;
volatile int *p;
```

Примеры:

```
float const pi = 3.1415926;
char const max_char = 127;
char *const str1 = "Здравствуй, мир!";//Неизменяемый указатель
char const *str2 = "Здравствуй, мир!";/*Указатель на неизменяемую
                                        строку*/
```

С учетом приведенных в примерах объявлений следующие присваивания будут рассматриваться компилятором как недопустимые:

```
pi = 2.7;
str1 = "Привет!";
strcpy(str2, "Привет!")
```

Однако вызов функции

```
strcpy(str1, "Привет!")
```

допустим, т.к. в данном случае осуществляется изменение содержимого памяти, на которую указывает указатель str, а не сам указатель.

Для иллюстрации использования модификатора `volatile` рассмотрим следующий фрагмент программы:

```
volatile int ticks;//Тики аппаратного таймера
…
void wait(int timeout){//Функция задержки
     ticks = 0;
     while(ticks < timeout)/*ticks изменяется асинхронно по
                            прерыванию от аппаратного таймера*/
     ;//Пустое тело цикла
}
```

Положим, что значение переменной `ticks` формируется в программе аппаратным таймером. Если бы переменная `ticks` была объявлена без модификатора `volatile`, то компилятор в целях оптимизации мог бы вынести за пределы цикла `while` сравнение переменных `ticks` и `timeout` (чтобы выполнить его один раз), поскольку в теле цикла их значения не изменяются. Это привело бы к семантической ошибке в работе программы (зацикливанию).

ОБЪЯВЛЕНИЯ И ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННЫХ

Простая переменная

Простая переменная – это переменная базового типа. Объявление простой переменой имеет следующую синтаксическую форму:

<класс_памяти> <спецификация_базового_типа> <идентификатор>;

Идентификатор специфицирует имя простой переменной. *Класс памяти* простой переменной на внутреннем уровне программы может быть **auto** (по умолчанию) или **register,** на внешнем - **static** или **extern** (по умолчанию). Область действия простой переменной с классом памяти **static** на внешнем уровне не выходит за рамки файла, в котором она объявлена. Для определения простой переменной её необходимо проинициализировать - присвоить значение *константного выражения*, которое дополняет объявление простой переменной:

<класс_памяти>
<спецификация_базового_типа><идентификатор>=<константное_выражение>;

Объявление одной и той же простой переменной может выполняться многократно в различных частях программы, а определение - только однажды в некоторой части программы.

Примеры:

```
#define const 13.4
…
extern int x;//Объявление простой переменной x
static double order=3.5+const;//Пример константного выражения
```

```
unsigned long flag=1, reply;/*Компактная форма объявления
                              однотипных переменных*/
```

Переменная перечислимого типа

Перечислимый тип определяет список поименованных целых констант типа `int`. *Переменная перечислимого типа* отличается от простой переменной только тем, что она должна принимать значения из конечного списка значений, специфицированных перечислимым типом. Это свойство переменной перечислимого типа контролируется только на этапе компиляции программы. При его нарушении компилятор выдает сообщение об ошибке. В процессе выполнения программы это свойство не контролируется и переменной может быть присвоено значение как простой переменной типа `int`. Объявление переменой перечислимого типа имеет следующие синтаксические формы:

1) <*класс_памяти*> `enum` <*тег*> {*список_перечисления*}<*идентификатор*>;

2) <*класс_памяти*> `enum` {*список_перечисления* }<*идентификатор*>;

3) <*класс_памяти*> `enum` <*тег*> <*идентификатор*>;

Идентификатор именует переменную перечислимого типа. *Тег* – это идентификатор, который именует перечислимый тип, специфицируемый списком перечисления. Тег перечислимого типа должен отличаться от тегов других перечислимых типов, структур и объединений с той же областью действия. *Список перечисления* содержит одну или несколько отделяемых запятыми конструкций вида:

<*идентификатор*>=<*константное_выражение_типа*_`int`>;

Идентификатор именует константу типа `int` (элемент списка перечисления), значение которой равно значению *константного выражения*. Константное выражение может быть как положительным, так и отрицательным, а может быть и опущено. Тогда по умолчанию первому элементу присваивается значение 0, второму элементу – значение 1 и т.д. Если в списке встречается элемент, для которого константное выражение задано, то элемент принимает значение именно этого константного выражения. Следующий за ним элемент, если только его значение не задается собственным константным выражением, получает значение на 1 большее предыдущего элемента. Таким образом, в списке перечисления могут быть одинаковые по значению именованные константы. Однако имена констант должны быть разными, причем во всех списках перечисления перечислимых типов программы в общих областях действия.

Перечислимый тип может быть определен самостоятельно, отдельно от объявления переменной, в форме:

`enum` <*тег*> {*список_поименованных_целых_констант_типа*_`int`};

Для определения переменной перечислимого типа её необходимо проинициализировать. Форма инициализации переменной перечислимого типа эквивалентна инициализации простой переменной.

Примеры:

```
static enum day {sat,//По умолчанию sat = 0
                 sun=0;//sun задана явно равной 0
                 mon,//=1
                 tue,//=2
                 ned,//=3
                 thu,//=4
                 fri //=5
                 }weekend = fri;

enum day d1 = 1;

enum day d10 = 10;//Недопустимое значение для инициализации
```

Указатель

Указатель – это переменная, значение которой специфицирует объект некоторого типа. Значение указателя специфицирует местонахождение в памяти и размер указываемого объекта. Указатель на функцию интерпретируется как точка входа в функцию. Форма объявления указателя следующая:

<*класс_памяти*> <*спецификация_типа*> *<*описатель*>;

Описатель вместе со *спецификацией типа* специфицирует имя указателя и тип указываемого объекта. Правила спецификации *класса памяти* и связанные с ним свойства указателя эквивалентны правилам спецификации класса памяти и свойствам простой переменной.

Можно объявить указатель на объект неопределенного типа. В этом случае на месте спецификации типа объекта ставится ключевое слово `void`. Значение указателя при этом специфицирует только местонахождение объекта в памяти, а размер объекта не задан.

Определение указателя выражается в присвоении ему начального значения (инициализация указателя). Для этого указателю необходимо присвоить значение *выражения типа указатель*, которое дополняет объявление указателя:

<*класс_памяти*> <*спецификация_типа*> *<*описатель*>=<*выражение-указатель*>;

Выражение-указатель должно имеет значение указателя на объект объявленного типа. Указатель инициализируется как простая переменная значением указателя на ранее определенные объекты программы.

Примеры:

```
char *mes;/*объявляется указатель с именем mes, который будет
          указывать на объекты типа char */

int *array[10];/*массив из 10 элементов, которые являются
                указателями на объекты типа int */
```

```
struct list *next;/*объявляется указатель next на структуру типа
                    list*/.
int i;
int *p1=&i;//&i – выражение-указатель на переменную i типа int
float A[10][4];
void *p2=A;/*неопределенный указатель p2 инициализируется адресом
             массива A*/
float (*p3)[10][4]=A;/*p3 инициализируется указателем на массив
                       A[10][4]*/
```

Массив

Спецификация массива

Массив - составная переменная, содержащая как единое целое последовательность элементов одинакового типа (элементы массива). Объявление массива специфицирует имя массива, тип и количество элементов массива. Элементы массива могут иметь базовый или перечислимый тип, тип структуры или объединения, тип указателя или, в свою очередь, массива. Объявление массива имеет следующую синтаксическую форму:

<*класс_памяти*> <*спецификация_типа*><*описатель*>[<*константное_выражение*>];

Описатель вместе со *спецификацией типа* именует массив и типизирует элементы массива. Квадратные скобки, следующие за идентификатором, являются признаком массива. *Константное выражение* задает число элементов в массиве. Если в квадратных скобках константное выражение опущено, то число элементов в массиве определяется при инициализации (см. ниже).

Формально в языке С отсутствует понятие многомерного массива. Алгоритмически многомерный массив рассматривается как массив, элементами которого, в свою очередь, являются массивы. Размерность многомерного массива при этом характеризуется количеством признаков массива, подряд идущих за описателем. Содержащиеся в них константные выражения задают количество элементов в соответствующем измерении.

Отметим, что имя массива (идентификатор) интерпретируется как константа-указатель на первый элемент массива и в этом качестве может использоваться в выражениях. Спецификация типа при объявлении массива может быть опущена, по умолчанию считается `int`.

Инициализация массива

Для определения значений элементов массива используется инициализатор. Он дополняет объявление массива и имеет следующую синтаксическую форму:

<*класс_памяти*><*спецификация_типа*>
<*идентификатор*>[<*константное_выражение*>]
={<*список_инициализаторов_элементов_массива*>};

Инициализатор массива заключается в фигурные скобки. Список инициализаторов элементов массива представляет собой последовательность инициализаторов элементов, составляющих массив, разделенных запятыми. Для элементов базового типа инициализаторами в списке являются константные выражения (или константы). Если элементами массива являются массивы (т.е. массив является многомерным), то инициализаторы в списке будут иметь вид инициализаторов массива - {<*список_инициализаторов_элементов_массива*>}. Значения присваиваются элементам массива в порядке их следования в списке слева направо. Для многомерных массивов допускается использовать список инициализаторов, не имеющий вложенной структуры, аналогичной структуре элементов многомерного массива, а представляющий собой список константных выражений. В этом случае элементы списка присваиваются элементам массива в порядке следования.

В случае, когда при определении массива размерность в квадратных скобках опущена, инициализатор определяет еще и размер массива. В этом случае размер массива определяется количеством элементов в списке инициализаторов. Вариант, когда опущено константное выражение и отсутствует инициализатор, используется только как объявление массива или в качестве формального параметра функции.

Примеры:

```
int M[]={1,2,3}; //M - массив из трех элементов int.
char p[4][3]={{1,2,3},{4,5,6},{7,8,9},{10,11,12}};/*p - массив из
              четырех элементов, являющихся массивами из трех
              элементов char */
p[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};/* допустимая форма
                                        инициализации */
```

Кроме того, существует специальная форма инициализации массива с элементами типа `char` с помощью символьной строки в виде:

```
char code[]="abcd";//Это эквивалентно ={'a','b','c','d','\0'};
```

Стандарт K&R языка C при инициализации символьного массива строкой требует обязательное использование фигурных скобок:

```
char a[]={"строка"};//Инициализация массива символами строки в K&R.
```

Доступ к элементам массива

В языке C отсутствует понятие переменной с индексом. Для доступа к элементам массива используются индексные выражения, имеющие следующий синтаксис:

<*выражение1*>[<*выражение2*>]

Значение выражения1 должно иметь тип указателя на элемент массива, а значение выражения2 должно быть целого типа. Значением индексного выражения является значение элемента массива, на который указывает

указатель, вычисленный как <*выражение1*>+<*выражение2*> по правилу сложения указателя с константой. Если элементом оказывается массив или указатель на элемент базового типа, то значением индексного выражения будет соответственно массив или значение указателя. Если элементом оказывается элемент базового типа, то значением индексного выражения будет константа соответствующего типа. Простейшим значением *выражение1* является имя (идентификатор) массива.

Примеры:

```
int mas[2][3][4];
int (*pm)[3][4], (*pm1)[4], *pm2, dm;
…
pm=mas;//mas - указатель массива размерностью [3][4]
pm1=mas[0];//значение mas[0] - массив размерностью [4]
pm2=mas[1][1];//значение mas[1][1] - указатель int
dm=mas[1][2][3];//значение mas[1][2][3] - элемент int
```

Структура

Спецификация структуры

Структура - составная переменная, содержащая как единое целое последовательность элементов, которые могут иметь различные типы (*элементы структуры*). Объявление структуры специфицирует имя структуры, имена и типы элементов структуры. Элементы структуры могут иметь базовый или перечислимый тип, тип структуры или объединения, тип указателя или массива. Объявление структуры имеет следующие синтаксические формы:

1) <*класс_памяти*> `struct` <*тег*> {<*список_объявлений_элементов*>}<*идентификатор*>;
2) <*класс_памяти*> `struct` {<*список_объявлений_элементов*>}<*идентификатор*>;
3) <*класс_памяти*> `struct` <*тег*><*идентификатор*>;

Список объявлений элементов специфицирует типы и имена элементов структуры. *Тег* - это идентификатор, который именует вводимый структурный тип, специфицированный соответствующим списком элементов. *Идентификатор* специфицирует имя переменной структурного типа.
Объявление структуры в первой форме с одним и тем же значением тега может встретиться в исходном файле не более одного раза. Объявление структур во второй и третьей форме могут использоваться в произвольном количестве. При этом третья форма может использоваться, если она находится в области действия ранее специфицированного тега. Кроме первой формы тег можно специфицировать и отдельно с помощью синтаксической формы вида:

`struct` <*тег*> {<*список_объявлений_элементов*>};

Элементы структуры могут иметь базовый или перечислимый тип, либо быть массивом, указателем, объединением или структурой. Кроме того, в

качестве элементов структуры могут использоваться так называемые битовые поля (см. ниже). Объявление элемента структуры не может содержать спецификации класса памяти.

Элемент структуры не может быть структурой того же (специфицируемого) типа. Однако он может быть объявлен как указатель на структуру специфицируемого типа. Это, например, удобно использовать, когда создаются связанные списки структур.

Имена элементов специфицируемой структуры должны быть различными. Имена элементов разных структур могут совпадать.

В пределах одной области действия тег, именующий тип структуры, должен отличаться от тегов других структурных типов, тегов объединений и тегов перечислимых типов.

Битовое поле

Битовое поле используется в качестве элемента структуры преимущественно в двух целях: для экономии памяти и для организации удобного доступа к регистрам внешних устройств, в которых различные биты могут иметь самостоятельное функциональное назначение.

Битовое поле размещается только в одном машинном слове и занимает в нем некоторое количество разрядов. В одном и том же машинном слове могут разместиться несколько битовых полей, если они следуют подряд в списке объявлений элементов структуры и умещаются в нем.

Объявление битового поля имеет следующий синтаксис:

<*спецификация_целого_типа*> <*идентификатор*>:<*константное_выражение*>;

Значение *константного выражения* задает количество разрядов, выделяемых битовому полю в текущем машинном слове, распределяемом под элементы структуры. *Идентификатор* именует битовое поле. Его наличие, однако, не обязательно. В этом случае битовое поле будет неименованным. Неименованные поля предназначены для пропуска соответствующего числа битов перед размещением следующего элемента структуры. Неименованное битовое поле может иметь нулевой размер. Наличие такого поля гарантирует, что память для следующего за ним элемента структуры (и для битового поля в том числе) будет начинаться на границе машинного слова. Для битового поля *спецификация типа* должна задавать целый тип. Он может быть знаковый или беззнаковый. Следует, однако, отметить, что в различных компиляторах различные спецификации целых типов для битовых полей единообразно поддерживаются лишь синтаксически, но с точки зрения возможных значений интерпретируются по-разному. Например, значение знакового битового поля в выражениях может рассматриваться как беззнаковое. Поэтому для мобильности программы, использующей битовые поля, целесообразно объявлять битовые поля с типом `unsigned int`.

Примеры:

```
1) struct complex {float x; float y;} complex;
```

```
2) struct complex compl_1,compl_2;

3) struct {float x,y;} compl_3,compl_4;

4) struct {char name[20];
          int id;
          long class;
          } temp;

5) struct list
          {int x,y;
          struct list *plist;//указатель на структуру того же типа
          } S1,S2;

6) struct {//Битовые поля
          unsigned icon: 8;
          unsigned color: 4;
          unsigned underline: 1;
          unsigned blink: 1
          } S1,S2;

7) struct STUDENT{
          char STUDENT[25];
          int GRUP;
          int ID;
          }STUDENT;/*Имена структуры, элемента и тега могут
                     совпадать*/
```

Инициализация структуры

Для определения значений элементов структуры используется инициализатор. Он дополняет объявление структуры и имеет следующие синтаксические формы:

1) <*класс_памяти*> `struct` <*тег*> {<*список_объявлений_элементов*>} <*идентификатор*> = {<*список_инициализаторов_элементов_структуры*>};

2) <*класс_памяти*> `struct` {<*список_объявлений_элементов*>} <*идентификатор*> = {<*список инициализаторов элементов структуры*>};

3) <*класс_памяти*> `struct` <*тег*> <*идентификатор*> ={<*список_инициализаторов_элементов_структуры*>};

Формат и порядок инициализации структуры, по сути, аналогичен формату и порядку инициализации массива. Отличие от массива лишь в том, что инициализируются разнотипные элементы, для каждого из которых применяется инициализатор собственной формы.

Примеры:

```
1) struct {char a; int b[2];} st={'b',{1,2}};

2) struct {int n1,n2,n3;}nlist[2][3]={
                                       {{1,1,1},{2,2,2},{3,3,3}},
                                       {{4,4,4},{5,5,5},{6,6,6}}
                                      };
```

Доступ к элементам структуры

Для доступа к элементу структуры используется так называемое *выражение выбора элемента*, которое имеет следующие синтаксические формы:

1) <*выражение1*>**.**<*идентификатор*>

2) <*выражение2*> -> <*идентификатор*>

В первой форме *выражение1* должно иметь значение типа `struct`, а идентификатор является именем элемента структуры. Во второй форме *выражение2* должно иметь значение указателя на структуру, а идентификатор, как и в первой форме, именует элемент структуры. Если к *выражение2* применить операцию косвенной адресации, то вторую форму индексного выражения можно заменить равносильной первой формой:

(*<*выражение2*>).<*идентификатор*>

Выбор формы выражения выбора элемента зависит, таким образом, от того, имеем ли мы доступ к структуре непосредственно (форма 1) или через указатель (форма2).

Пример:

```
struct par{
           int a;
           int b;
           struct par*sp;
           } item,list[10];
…
item.sp=&item; //элементу sp присваивается адрес структуры item
(item.sp)->a = 24; // элементу a присваивается значение 24
list[6].b = 12;// элементу b 6-ого элемента массива присвоили 12
```

Объявление объединения

Спецификация объединения

Объединение - переменная, позволяющая в разные моменты времени хранить значения различного типа (*элементы объединения*). Объявление объединения специфицирует имя объединения, имена и типы элементов объединения. Элементы объединения могут иметь базовый или перечислимый тип, тип структуры или объединения, тип указателя или массива. Объявление объединения имеет следующие синтаксические формы:

1) <*класс_памяти*> `union` <*тег*> {<*список_объявлений_элементов*>}<*идентификатор*>;

2) <*класс_памяти*> `union` {<*список_объявлений_элементов*>}<*идентификатор*>;

3) <*класс_памяти*> `union` <*тег*><*идентификатор*>;

Список объявлений элементов специфицирует типы и имена элементов объединения. *Тег* - это идентификатор, который именует вводимый тип

объединения, специфицированный соответствующим списком элементов. *Идентификатор* специфицирует имя переменной типа объединения.
Объявление объединения в первой форме с одним и тем же значением тега может встретиться в исходном файле не более одного раза. Объявление объединения во второй и третьей форме могут использоваться в произвольном количестве. При этом третья форма может использоваться, если она находится в области действия ранее специфицированного тега. Кроме первой формы тег можно специфицировать и отдельно с помощью синтаксической формы вида:

`union` <*тег*> {<*список_объявлений_элементов*>};

Объявление элемента объединения не может содержать спецификации класса памяти.

Элемент объединения не может быть объединением того же (специфицируемого) типа. Однако он может быть объявлен как указатель на объединение специфицируемого типа.

Имена элементов специфицируемого объединения должны быть различными. Имена элементов разных объединений могут совпадать.

В пределах одной области действия тег, именующий тип объединения, должен отличаться от тегов других объединений, тегов структурных типов и тегов перечислимых типов.

Память, которая выделяется переменной типа объединение, определяется размером наиболее длинного из элементов объединения. Все элементы объединения размещаются в памяти с одного и того же адреса.

Примеры:

```
/*пример 1*/
union {
  char *a,b;
  float f[20];
}common;

/*пример 2*/
union sign{
  int s_var;
  unsigned u_var;
}trigger;
```

Инициализация объединения

Объединение инициализируется в формате инициализации первого элемента объединения. Для этого используется инициализатор, соответствующий первому элементу объединения. В итоге определение объединения и имеет следующую синтаксическую форму:

1) <*класс_памяти*> `union` <*тег*> {<*список_объявлений_элементов*>}
   <*идентификатор*> = {<*инициализатор_первого_элемента_объединения*>};

2) <*класс_памяти*> `union` {<*список_объявлений_элементов*>}
   <*идентификатор*> = {<*инициализатор_первого_элемента_объединения*>};

3) <*класс_памяти*> `union` <*тег*>
<*идентификатор*> = {<*инициализатор_первого_элемента_объединения*>};

Формат и порядок инициализации объединения соответствует формату и порядку инициализации первого элемента объединения.

Примеры:

```
1) union {char a; int b[2];} un1='b';
2) union {int nlist[2][3]; int n1;} un2={{1,1,1},{2,2,2}};
```

Доступ к элементам объединения

Для доступа к элементу объединения используется так называемое *выражение выбора элемента*, которое имеет следующие синтаксические формы:

1) <*выражение1*>**.**<*идентификатор*>

2) <*выражение2*> **->** <*идентификатор*>

В первой форме *выражение1* должно иметь значение типа **`union`**, а идентификатор является именем элемента объединения. Во второй форме выражение2 должно иметь значение указателя на объединение, а идентификатор, как и в первой форме, именует элемент объединения. Если к *выражение2* применить операцию косвенной адресации, то вторую форму индексного выражения можно заменить равносильной первой формой:

(*****<*выражение2*>)**.**<*идентификатор*>

Выбор формы выражения выбора элемента зависит, таким образом, от того, имеем ли мы доступ к объединению непосредственно (форма 1) или через указатель (форма2).

Пример:

```
union UNI{
      int ival;
      float fval;
      union UNI *pval;
      }val;
…
    val.ival=5;
    …
    val.pval=&val; //адрес объединения val
    …
    (val.pval)->fval=12.5;
```

Объявление типа

Существует два особых вида объявления, в которых объявляется не переменная или функция, а тип данных. Первый вид позволяет определить имя типа (тег) структуры, объединения или перечислимого типа. После такого объявления тег может быть использован в объявлениях переменных и функций

для указания типа объекта.

Второй вид объявления типа использует ключевое слово `typedef`. Это объявление позволяет присвоить осмысленные имена типам, уже существующим в языке или создаваемым пользователем.

Объявление типа имеет такую же блочную область действия, как и объявление переменной. Локальное переобъявление имени типа также возможно.

Объявление typedef

Объявление `typedef` синтаксически аналогично объявлению переменной или функции, за исключением того, что вместо спецификации класса памяти записывается ключевое слово `typedef` и отсутствует инициализатор.

Синтаксис:

`typedef` <*спецификация_типа*> <*описатель*>[,<*описатель*>...];

Объявление `typedef` интерпретируется таким же образом как объявление переменной или функции, однако идентификатор, входящий в состав описателя, специфицирует (именует) не переменную или функцию, а тип. Идентификатор становится именем объявленного типа и может употребляться в последующих объявлениях программных объектов в качестве абстрактного типа. Другими словами, создаются не новые типы, а имена для специфицированных ранее типов. С помощью `typedef` может быть объявлено имя для любого типа как базового, так и составного.

Примеры:

```
/* пример 1 */
typedef int BOOL;// BOOL - синоним типа int
/* пример 2 */
typedef struct club{char name[30];
                    int size;
                    year;
                    } GROUP;// GROUP – синоним структуры club
/* пример 3 */
typedef GROUP *PG;// PG – тип указателя на GROUP
/* пример 4 */
typedef void DRAWF(int, int};// DRAWF – тип функции
DRAWF box;//Эквивалентно void box(int, int}
```

Абстрактные имена типов

Иногда возникает необходимость специфицировать некоторый тип данных без присвоения ему имени. Такая конструкция называется *абстрактным именем типа*. Абстрактные имена типов используются в трех контекстах: в списках типов аргументов при объявлении функций, в операции приведения типа и в операции `sizeof`.

Абстрактный описатель отличается от обычного описателя программного

объекта только тем, что он не содержит идентификатора. Для интерпретации абстрактного описателя следует, прежде всего, определить в нем место подразумеваемого идентификатора. После этого интерпретация типа проводится по тем же правилам, что и для обычного описателя программного объекта с той лишь разницей, что не используется имя.

Примеры:

```
int (*)[5]     – тип указателя на массив int;
int (*)(void)  – тип указателя на функцию;
char *(*)[]    – тип указателя на массив указателей на char;
```

ФУНКЦИИ

Любая программа на языке С содержит по крайней мере одну, так называемую главную функцию, с именем `main`. Количество функций, которые могут быть определены в программе, не ограничивается. Для использования функции в программе она предварительно должна быть определена или объявлена. Функция определяется в программе только один раз, а объявляться может многократно в различных частях программы.

Определение функции

Определение функции имеет следующую синтаксическую форму:

<класс_памяти> *<спецификация_типа>*
<описатель>(*<спецификация_формальных_параметров>*)
{*<функциональный_блок>*}

Описатель в совокупности со *спецификацией типа* специфицирует имя функции и тип возвращаемого функцией значения. Имя функции так же, как и имя массива, имеет значение константы-адреса функции. Спецификация типа может быть опущена, тогда по умолчанию считается `int`. Функция может быть определена как не возвращающая значения (аналог процедуры). В этом случае в качестве спецификации типа используется тип `void`.

Класс памяти специфицирует класс памяти функции. Для функции допускается указание класса памяти `static` или `extern`. Если класс памяти в определении функции опущен, то по умолчанию считается `extern`.

Спецификация формальных параметров представляет собой список объявлений формальных переменных, разделяемых запятыми, посредством которых функция получает значения аргументов при её вызове. После последнего объявления в списке формальных параметров может следовать запятая с многоточием <,…>. Это означает, что число параметров функции не фиксировано (переменно), однако не меньше, чем следует объявлений параметров до многоточия. Функция может и вовсе не иметь формальных параметров. В этом случае список формальных параметров в круглых скобках становится пустым. Допускается (и рекомендуется) для обозначения пустого списка использовать ключевое слово `void`.

Объявления формальных параметров имеют тот же самый синтаксис, что и обычные объявления переменных. Параметры могут иметь класс памяти `auto` или `register`, по умолчанию - `auto`. Объявления формальных переменных могут содержать модификатор `const`, означающий, что соответствующий аргумент не может изменяться в теле функции. Формальные параметры могут иметь базовый тип, либо быть структурой, объединением, указателем или массивом. Массив воспринимается как указатель на тип элементов массива. Задание первой (или единственной) размерности в объявлении массива не обязательною. Таким образом, при объявлении массива как формального параметра эквивалентны, например, следующие синтаксические формы:

```
char s[];
char s[10];
char *s;
```

Функциональный блок является телом функции и представляет собой составной оператор. Он содержит операторы, которые определяют действие функции, и необходимые объявления переменных внутри тела функции (на внутреннем уровне). Таким, образом, операторам в теле функции доступны объекты, объявленные на внешнем уровне программы (типы, переменные, функции), в области действия которых находится определение функции, а также типы и переменные, объявленные внутри тела функции. Имена формальных параметров не могут совпадать с именами переменных, объявленных внутри тела функции, но возможно локальное переобъявление имен формальных параметров внутри блоков, вложенных в тело функции.

Если функция возвращает значение, то должен быть выполнен оператор `return`, содержащий выражение. Оператор `return` завершает выполнение функции. В отсутствии оператора `return` выполнение функции завершается при достижении завершающей фигурной скобки функционального блока. Если оператор `return` не выполнен, или если в операторе `return` отсутствует выражение, то возвращаемое значение не определено.

Пример:

```
static int Add(int A, int B){
if(A>0 & B>0)return A+B;
else return 0;
}
```

Область действия определения функции с классом памяти `static`, распространяется только на текущий исходный файл, в котором она определена, с места её определения и до конца файла или до очередного объявления на внешнем уровне текущего файла функции с таким же именем, но с классом памяти `extern`, а также за исключением тех блоков, в которых она локально переобъявляется функцией. То есть в каком-то блоке может быть объявлена функция с тем же именем и классом памяти `extern`, которая определена в другом файле.

Область действия определения функции с классом памяти `extern` можно распространить на любой другой исходный файл программы, в котором выполнено объявление этой функции на внешнем или внутреннем уровне. То есть, она может быть вызвана в любом другом исходном файле в любой точке этого файла, следующей за объявлением функции в этом файле, за исключением блоков, в которых она локально переобъявляется. Например, если в каком-то из файлов на внешнем уровне объявлена функция с тем же именем и классом памяти `static`, то с этого места именно она будет вызываться в этом файле.

Имя определённой функции может использоваться в выражениях в качестве константы-адреса этой функции.

Объявление функций

Объявление функции требуется для распространения действия имени некоторой определенной в программе функции в ту часть программы (файл, блок), в которой это действие в силу различных причин отсутствует, но необходим вызов этой функции. Объявление функции имеет следующую синтаксическую форму:

<класс_памяти> <спецификация_типа>
<описатель>(<спецификация_типов_формальных_параметров>);

Класс памяти, *спецификация типа* и *описатель* в объявлении функции должны соответствовать своему прототипу (определению функции). *Спецификация типов формальных параметров* в объявлении отличается от спецификации формальных параметров в определении функции тем, что в ней не требуется указывать имена формальных параметров. На месте имен формальных параметров могут использоваться любые идентификаторы (в частности, и имена формальных параметров) или не использоваться вовсе (полностью отсутствовать).

Использование функций в языке С имеет некоторые отличия от использования переменных. Во-первых, на внутреннем уровне функция может быть только объявлена, а на внешнем уровне - и объявлена, и определена. Во-вторых, для работы с переменной ее необходимо предварительно явно объявить, а для того, чтобы вызвать функцию, необязательно. При вызове функции без предварительного объявления компилятор языка С рассматривает сам вызов как *неявное объявление функции* с типом возвращаемого значения `int`, классом памяти `extern` и типами формальных параметров, соответствующими типам фактических аргументов, использованных в вызове. Если далее в файле встретится объявление или определение этой функции с другими атрибутами, компилятор сообщит об ошибке.

Область действия объявления функции распространяется только на текущий исходный файл за исключением тех областей на внешнем уровне текущего файла и блоков, в которых она локально переобъявляется. Например, в каком-то блоке может быть объявлена функция с тем же именем, но с другим классом памяти.

Примеры:

```
double (*sum(double, double))[3];/*Объявляется функция sum с двумя
                                   аргументами типа double,
                                   возвращающая указатель на массив
                                   из трех элементов типа double*/
int f1(void);/*объявляется функция f1 без аргументов, возвращающая
               значение int*/
void f2(void);/*Объявляется функция f2 без аргументов и ничего не
                возвращающая*/
```

Вызов функции.

Вызов функции передает управление и фактические аргументы (если они есть) заданной функции. Синтаксическая форма вызова функции следующая:

<выражение-функция>(<список_выражений_аргументов>)

Выражение-функция должно иметь тип функция. Оно вычисляется, и его результат интерпретируется как адрес функции.

Список выражений аргументов, в котором выражения следуют через запятую, вычисляются, над полученными результатами выполняются необходимые преобразования, обусловленные типами соответствующих формальных параметров, и их значения выступают в качестве аргументов, присваиваемых формальным параметрам при вызове функции. Между порядком следования формальных параметров в объявлении функции и аргументами в вызове функции существует взаимнооднозначное соответствие. Если функция не имеет формальных параметров, то список аргументов в вызове функции является пустым. Аргумент может быть любым значением базового типа, структурой, объединением или указателем, а также функцией. Имена массивов рассматриваются как константы-указатели первого элемента массива, а имена функций - как константы-указатели функции.

При вызове функции память распределяется формальным параметрам в стеке программы, куда и копируются значения аргументов. Таким образом, если в качестве аргумента выступает переменная программы, то её значение не может быть изменено в результате выполнения функции. То есть все аргументы передаются по значению. Если же такое изменение необходимо, то в качестве аргумента следует передавать значение указателя переменной, а формальный параметр должен быть указателем на тип, соответствующий типу этой переменной. В теле функции доступ к переменной должен обеспечиваться косвенно через указатель.

При вызове функции соответствие между формальными параметрами и аргументами устанавливается следующим образом:

1) Если вызываемая функция объявлена явно (имеется прототип), то типы в списке типов формальных параметров подвергаются преобразованию по умолчанию (см. преобразования по умолчанию). Затем вычисленное значение аргумента приводится к полученному преобразованному типу и копируется в стек.

2) Если прототип отсутствует (функция объявляется на основании вызова), то преобразования по умолчанию производятся для каждого аргумента, после чего полученные значения копируются в стек.

3) Соответствие формальных параметров их значениям в стеке, устанавливается на основании типов формальных параметров, предварительно преобразованных по умолчанию. Преобразованный тип каждого формального параметра определяет, каким образом интерпретируются значения аргументов в стеке.

Заметим, что если тип формального параметра не соответствует типу значения аргумента в стеке, то данные в стеке могут быть проинтерпретированы неверно, что может привести к плохо контролируемым ошибкам. Поэтому следует категорически избегать неявного объявления функции при вызове.

Примеры:

```
/*пример 1*/
void main(void){
void swap(int*, int*);//Объявление функции swap
int x, y;
…
swap(&x, &y);//Вызов функции swap
…
}

//Определение функции swap
void swap(int*a, int*b){
int t;
     t=*a;
     *a=*b;
     *b=t;
}

/*пример 2*/
extern int F0(int, int);
extern int F1(int, int);
extern int F2(int, int);

int (*Fmas[3])(int, int)={F0,F1,F2};/*Массив указателей на функции
                                      F0,F1,F2*/
/*Вызвы функций F0,F1,F2*/
Fmas[0](1,1);//Выражение Fmas[0] имеет значение функции F0
Fmas[1](1,1);//Выражение Fmas[1] имеет значение функции F1
Fmas[2](1,1);//Выражение Fmas[2] имеет значение функции F2
```

Функции с переменным числом аргументов

Если функция определяется как функция с переменным числом аргументов, то в функциональном блоке необходимо обеспечить получение значений аргументов, для которых нет формальных параметров. Это те аргументы, которые при вызове функции следуют за обязательными аргументами (им соответствуют формальные параметры). Все аргументы функции при вызове помещаются в стек. Количество формальных параметров, специфицированных в объявлении функции, определяет количество аргументов, которые автоматически присваиваются формальным параметрам. Доступ к остальным аргументам требует использования специальных стандартных макроопределений (макросов), определенных в системном заголовочном файле `stdarg.h`, объявления которых выглядят следующим образом:

```
1) void va_start(va_list param, previous);
```

2) `type va_arg(va_list param, type);`

3) `void va_end(va_list param);`

Все макросы используют формальный параметр `param` системного типа `va_list`, предназначенный для согласования работы макроопределений. Перед вызовом макросов необходимо объявить в программе вспомогательную переменную типа `va_list` и затем использовать её в качестве аргумента, присваиваемого формальному параметру `param` при вызовах макросов.

Макровызов `va_start` выполняется один раз и предназначен для инициализации сеанса доступа к аргументам в стеке, не связанным с формальными параметрами. Для параметра `previous` макроса `va_start` в качестве аргумента необходимо использовать *имя формального параметра*, последнего в списке формальных параметров функции с переменным числом аргументов.

Макровызов `va_arg` в результате выполнения возвращает значение очередного аргумента как значение, соответствующее типу с именем `type`, и подготавливает доступ к следующему аргументу в стеке. В качестве имени типа `type` используются имена базовых типов или любых других объявленных в программе типов. Последовательное выполнение макроса `va_arg` позволяет выбрать из стека все аргументы, переданные вызванной функции с переменным числом аргументов.

Макровызов `va_end` выполняется один раз и используется для завершения сеанса доступа к стеку для выбора аргументов.

При использовании переменного числа аргументов необходимо иметь признак окончания аргументов. Для такого признака может быть использован один из формальных параметров, посредством которого можно указывать, например, количество аргументов в вызове функции. Другой вариант – это использовать значение последнего аргумента в качестве признака конца аргументов. Можно также один из аргументов рассматривать как форматную строку, описывающую типы и количество передаваемых аргументов.

Пример:

```
long sum_int(char count,…){/*Функция с переменным числом
                              аргументов*/
  long sum=0;
  va_list curr_arg;//Вспомогательная переменная для макровызовов

va_start(curr_arg, count);/*Начало сеанса доступа к аргументам в
                            стеке*/
if(count<=0) return 0;
while(count--) sum += va_arg(curr_arg, int);/*Взять из стека
                                              аргумент типа int*/
va_end(curr_arg);//Завершение сеанса доступа к аргументам в стеке
return sum
}
/*Вызов функции с переменным числом аргументов*/
```

```
void main(void){
     long Sum;
          Sum = sum_int(6,1,2,3,4,5,6);
…
}
```

Рекурсивный вызов функции

Каждая функция может рекурсивно вызвать саму себя. При каждом рекурсивном вызове новые ячейки памяти выделяются в стеке для новых аргументов и локальных переменных класса памяти `auto` или `register`. Следовательно, их значения в предшествующих, незавершенных вызовах сохраняются. Для переменных, объявленных в теле функции с классом памяти `static` или `extern`, новые ячейки памяти при рекурсивном вызове не выделяются. Это означает, что каждый следующий рекурсивный вызов функции разделяет одни и те же статические переменные.

Формально количество рекурсивных вызовов не ограничено. Но так как каждый рекурсивный вызов требует дополнительной стековой памяти, слишком большое количество рекурсивных вызовов может привести к переполнению стека.

Пример:

```
void mirror(char *par){// par – указатель строки символов
char chr;
    chr=*par++;/*Присвоить текущий символ строки и перейти к
                 следующему*/
if(*par) mirror(par);//Рекурсивный вызов, если *par не равен '\0'
    putchar(chr); //Печать символа
}

void main(void){
    mirror("ave");//Печатает строку "ave" в обратном порядке - eva
}
```

Формальные параметры функции main

Запуск программного файла на выполнение равносилен вызову функции `main`. Функция `main` также может иметь формальные параметры, но, в отличие от остальных функций, её параметры не могут задаваться произвольно. Количество, порядок следования и семантика формальных параметров функции `main` строго определены. Кроме того, аргументы передаются формальным параметрам функции `main` при запуске программы из командной строки и из переменных окружения (системные переменные ОС). Функция `main` с формальными параметрами имеет вид:

*<спецификация_типа> main(int argc, char *argv[],char *env[])*{

…

}

Семантика формальных параметров функции **main** следующая:

`argc` – количество аргументов командной строки;

`argv` – массив указателей на аргументы командной строки;

`env` – массив указателей на переменные окружения.

Для передачи аргументов командной строки функции **main** они размещаются в командной строке вслед за именем исполняемого файла. Аргументы передаются в виде символьных строк (заключать в двойные кавычки не обязательно). Аргументы отделяются друг от друга пробелами или символами горизонтальной табуляции. Если требуется передать программе строку аргумента, содержащую внутри себя пробелы или символы горизонтальной табуляции, следует заключить её в двойные кавычки.

Массив **argv** имеет следующее содержание:

`argv[0]` – указатель строки с полным именем исполняемого файла в файловой системе;

`argv[1]` – указатель строки первого аргумента;

`argv[2]` – указатель строки второго аргумента;

…

`argv[argc-1]` – указатель строки последнего аргумента;

`argv[argc]` – нулевой указатель – `NULL`.

Значение формального параметра `argc` не может быть меньше 1. Если значение `argc` равно 1, это означает, что аргументы в командной строке отсутствуют.

Массив `env`, начиная с `env[0]`, содержит указатели на строки переменных окружения вида:

```
<идентификатор>=<значение>
```

Идентификатор является именем переменной окружения, затем следует знак равенства и оставшаяся часть строки интерпретируется как значение переменной окружения. Последний элемент массива `env` содержит нулевой указатель – `NULL`.

Задавать все параметры функции `main` не обязательно. Они используются по необходимости. Допустимы следующие объявления формальных параметров:

```
main()
main(int argc)
main(int argc, char *argv[])
main(int argc, char *argv[],char *env[])
```

Ниже приводится пример программы ARGS.C, которая выводит на экран значения аргументов, заданных в командной строке при запуске программного файла, а также текущие значения переменных среды.

Пример:

/* Программа ARGS.C */

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[]){
 int i;

 printf("Значение argc = %d \n\n", argc);
 printf("Функции main передано %d аргументов в командной \
 строке: \n\n",argc);

 for (i = 0; i < argc; i++)
  printf("argv[%d]: %s\n", i, argv[i]);

 printf("\nПереданы следующие значения переменных среды:\n\n");

 for (i = 0; env[i] != NULL; i++)
  printf(" env[%d]: %s\n", i, env[i]);

 return 0;
}
```

Предположим, что полученный в результате компиляции программный модуль имеет полное имя `/root/home/args`, находится в текущем каталоге и запущен на выполнение командой:

```
$ args   first_arg "арг с пробелами" 3 4 "предпоследний арг" stop!
```

Заметим, что можно передавать аргументы, содержащие пробелы, заключив эти аргументы в двойные кавычки, как показано выше в примере командной строки: "арг с пробелами" и "предпоследний арг".

Результат выполнения программы args (предполагая, что заданы переменные среды, имеющие представленные ниже значения) будет выглядеть на экране в виде:

```
Значение argc равно 7

Функции main передано 7 аргументов в командной строке:

argv[0]: /root/home/args
argv[1]: first_arg
argv[2]: арг с пробелами
argv[3]: 3
argv[4]: 4
```

argv[5]: предпоследний арг
argv[6]: stop!

Переданы следующие значения переменных среды:

env[0]: PROMPT=$p $g
env[1]: PATH=/SPRINT; /GCC

ВЫРАЖЕНИЯ

Выражение-это комбинация операндов и операций, задающая порядок вычисления некоторого значения. Операции определяют действия, выполняемые над операндами. Операнд в простейшем случае является константой или переменной. В общем случае каждый операнд выражения также представляет собой выражение, имеющее некоторое значение.

Операнды

Операндом выражения может быть константа, простая переменная и первичные выражения, к которым относят вызов функции, индексное выражение, выражение выбора элемента. Часть выражения, заключенная в круглые скобки, также рассматривается как операнд выражения.

Каждый операнд имеет тип. Тип операнда перед выполнением операции может быть явно или неявно преобразован к другому типу.

Константное выражение

Константное выражение-это выражение, операндами которого могут быть целые, символьные, плавающие константы, константы перечислимого типа, выражения. Использование операций в константных выражениях ограничено. В константных выражениях нельзя использовать операции присваивания, операцию последовательного вычисления. Использование операций адресации, приведения типа и плавающих констант ограничено. Только выражения, используемые для инициализации переменных, допускают применение плавающих констант и операции адресации.

Константные выражения, используемые в директивах препроцессора, имеют дополнительные ограничения, поэтому их называют *ограниченными константными выражениями*. Ограниченное константное выражение не может содержать операцию `sizeof`, операцию приведения типа, константы перечисления и плавающие константы. Но может содержать сугубо препроцессорную операцию `defined` (см. раздел препроцессор языка С).

Операции

Унарные операции

Унарный минус (-)

Операция унарного минуса выполняет арифметическое отрицание своего операнда. Операнд унарного минуса должен иметь целый или плавающий тип. Тип результата совпадает с преобразованным по умолчанию типом операнда.

Унарный плюс (+)

Операнд унарного плюса должен иметь целый или плавающий тип. Операция унарного плюса не изменяет значения своего операнда, а применяется

для того, чтобы запретить компилятору языка С реорганизовывать скобочные выражения.

Обычно компиляторы языка С осуществляют перегруппировку выражений, переупорядочивая операции, обладающие свойством коммутативности (умножение, сложение, поразрядные операции), пытаясь сгенерировать как можно более эффективный код. При этом скобки, ограничивающие операции, не принимаются в расчет. Однако компилятор не будет реорганизовывать выражения в скобках, если перед скобками записана операция унарного плюса. Это позволяет, в частности, контролировать точность вычислений с плавающей точкой. Например, если a, b, c и f имеют тип `fIoat`, выражение

```
f = a * + (b * c)
```

будет гарантированно вычисляться с учетом скобок.

Обратный код (~)

Операция обратного кода вырабатывает двоичное дополнение своего операнда, т.е. инвертирует его битовое представление. Операнд должен иметь целый тип. Результат имеет тип преобразованного операнда.

Если операнд имеет знаковый бит, то бит знака участвует в операции обратного кода (инвертируется).

Логическое отрицание (!)

Операция логического отрицания обозначается символом <**!**>. Она вырабатывает значение 0, если операнд есть ИСТИНА, и значение 1, если операнд есть ЛОЖЬ. Результат имеет тип `int`. Операнд должен иметь целый или плавающий тип либо быть указателем.

Примеры:

```
/* пример 1 * /
short x = 987;
x = -x;
/* пример 2 * /
unsigned short y = OxAAAA;

y = ~y;
/* пример 3 * /
if(!( x < y )) ...
```

В первом примере новое значение `x` равно -987. Во втором примере переменной `y` присваивается новое значение, которое является обратным кодом беззнакового значения `OxAAAA`, т. е. `Ox5555`.

В третьем примере, если `x` больше или равен `y`, то результат условного, выражения в операторе `if` равен `1` (`ИСТИНА`). Если `x` меньше `y`, то результат равен `0` (`ЛОЖЬ`).

Адресация (&)

Операция адресации обозначается символом <&>. Операндом должно быть L-выражение. В простейшем случае – это имя переменной. Результатом операции адресации является значение указателя на операнд. Тип результата – указатель на тип операнда.

Операция адресации не может применяться к битовым полям структур, а также к идентификаторам, объявленным с классом памяти `register`.

Косвенная адресация (*)

Операция косвенной адресации обеспечивает доступ к программному объекту через указатель. Ее операнд должен иметь тип указателя. В качестве операнда может также выступать имя массива (указатель на первый элемент массива).

Результатом операции является программный объект, на который указывает операнд-указатель. Типом результата является тип, ассоциированный с этим указателем. Если имеется указатель на массив, то результатом операции косвенной адресации над этим указателем будет сам массив, т.е. указатель на первый элемент этого массива.

Пример:

```
int *pa, (*pm)[29], x;
int a[29];
…
pa = &a[5];
x = *pa;//x примет значение элемента массива a[5]
pm = (int(*)[29])a;
x = *(*pm + 2);//*pm – массив a, т.е. указатель на 1-ый элемент
…
```

Операция sizeof

Операция `sizeof` определяет размер памяти, который соответствует объекту или типу. Операция `sizeof` имеет следующий вид:

`sizeof` <*выражение*>

`sizeof` (<*имя_типа*>)

Операндом является либо выражение, либо имя типа в скобках (базового или определенного пользователем). Результатом операции `sizeof` является размер памяти в байтах, соответствующий заданному объекту или типу. Тип результата операции - `unsigned int`. Если размер объекта не может быть представлен значением типа `unsigned int`, то следует использовать форму:

`(long) sizeof` <*выражение*>

Следует учитывать, что само выражение не вычисляется, а лишь оценивается тип выражения. Операция `sizeof` выполняется только на этапе компиляции программы.

Недопустимо в качестве операнда использовать тип `void`. Применение операции `sizeof` к имени функции эквивалентно определению размера указателя на функцию. Если операция `sizeof` применяется к имени массива, то результатом является размер всего массива в байтах, а не размер указателя на элемент массива. Если операция `sizeof` применяется к имени или типу структуры или объединения, то результатом является фактический размер в байтах структуры или объединения, который может включать и участки памяти, используемые для выравнивания элементов структуры или объединения на границы слов памяти. Таким образом, этот результат может превышать размер, вычисленный путем сложения размеров отдельных элементов структуры. Например, если объявлена следующая структура

```
struct {
       char m[3 ][3 ];
       } s;
```

то значение `sizeof(s.m)` будет равно 9, а значение `sizeof(s)` будет равно 10.

Операция приведения типа

Операция приведения типа предназначена для явного преобразования значения операнда к заданному типу. Она имеет следующую синтаксическую форму:

(<имя_типа>)<*выражение*>

Имя типа может быть именем базового типа или именем специфицированного абстрактного типа, введенного в программу.

Примеры:

```
int (*p)(long);//Указатель на функцию
…
(int  (*)(int))p(0);/*Преобразование значения указателя в указатель
                      на функцию с параметром типа int*/
(struct{int a; int b;}*)p->a=5;/* Преобразование значения указателя
                                  в указатель на структуру*/
```

Бинарные операции

Бинарные операции выполняются над двумя операндами. Тип первого и второго операндов могут отличаться, при этом выполняются преобразования операндов по умолчанию, приводящие значения операндов к единому типу. Типом результата является тип операндов после преобразования. В процессе выполнения операций ситуация переполнения или потери значимости не контролируется. Если результат не может быть представлен типом операндов после преобразования, то результат неопределенный.

Преобразование типов данных по умолчанию

Для выполнения большинства операций языка С требуется, чтобы

операнды имели один и тот же необходимый тип. Если типы операндов не соответствуют требованиям, то перед выполнением операции осуществляется преобразование типов операндов либо для приведения их к общему типу, либо для того, чтобы расширить значения коротких по размеру типов до требуемого размера, используемого в машинных операциях. Эти преобразования называются *преобразованиями по умолчанию*.

Преобразования по умолчанию осуществляются следующим образом:

1) Все операнды типа `float` преобразуются к типу `double`.
2) Если один операнд имеет плавающий тип, то второй операнд преобразуется к плавающему типу.
3) Все операнды типов `char` или `short` преобразуются к типу `int`.
4) Все операнды типов `unsigned char` или `unsigned short` преобразуются к типу `unsigned int`.
5) Если один операнд имеет беззнаковый тип (`unsigned …`), то второй операнд преобразуется к беззнаковому типу.
6) Если один операнд имеет более длинный по размеру тип, то второй операнд преобразуется к более длинному типу.

Знаковое целое значение преобразуется к длинному знаковому целому значению (`signed long int`) путем расширения знакового разряда влево.

Беззнаковое целое значение преобразуется к длинному беззнаковому целому значению или длинному знаковому целому значению путем дополнения нулями слева.

При преобразовании знакового целого значения к беззнаковому целому значению производится лишь преобразование к размеру беззнакового целого типа, и результат интерпретируется как беззнаковое целое значение.

Преобразование знаковых целых значений или беззнаковых целых значений к плавающим значениям происходит путем предварительного преобразования к типу `long`, а затем преобразования к плавающему типу.

Значения типа `float` преобразуются к типу `double` без потери точности.

Умножение (*)

Операция умножения выполняет умножение одного операнда на другой.

Деление (/)

Операция деления выполняет деление первого операнда на второй. Если оба операнда являются целыми значениями и не делятся нацело, то результат округляется в сторону нуля. Деление на нуль дает ошибку во время выполнения.

Остаток от деления (%)

Операндами могут быть только целые значения. Результатом операции является остаток от деления первого операнда на второй. Знак результата

совпадает со знаком делимого.

Сложение (+)

Операция сложения складывает два своих операнда. Операнды могут иметь целый или плавающий тип. Типы операндов могут различаться. Один из операндов может быть указателем; тогда другой должен быть целым значением. Когда целое значение складывается с указателем, то оно масштабируется путем умножения его на размер типа, с которым ассоциирован данный указатель. Когда преобразованное значение целого операнда складывается со значением операнда-указателя, то результатом является указатель, адресующий область памяти, отстоящую от первоначального адреса на количество объектов памяти указываемого типа, равное значению целого. Новый указатель указывает на тот же самый тип данных, что и исходный указатель.

Вычитание (-)

Операция вычитания вычитает второй операнд из первого. Операнды могут иметь целый или плавающий тип. Тип первого и второго операндов могут различаться. Допускается вычитание целого из указателя и вычитание двух указателей.

Когда целое значение вычитается из указателя, предварительно производится то же масштабирование, что и при сложении целого значения с указателем. Результатом вычитания является указатель, указывающий на область памяти, расположенную перед первоначальным адресом со смещением на количество объектов памяти указываемого типа, равное значению целого.

Один указатель может быть вычтен из другого, если они указывают на один и тот же тип данных. Разность между двумя указателями преобразуется к знаковому целому значению путем деления разности на длину типа объекта, на который указывают указатели. Результат представляет число объектов данного типа между двумя адресами.

В общем случае разность указателей может принимать большое значение и целесообразно явно приводить его к типу `long`, чтобы обеспечить гарантированное сохранение полученного результата.

Операции сдвига

Операции сдвига сдвигают свой первый операнд влево (**<<**) или вправо (**>>**) на число разрядов машинного слова, специфицированное вторым операндом. Оба операнда должны быть целыми значениями. Перед выполнением операции над операндами выполняются преобразования по умолчанию. Тип результата имеет единый тип преобразованных операндов.

При сдвиге влево правые освобождающиеся биты заполняются нулями. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от того, какой тип результата получен после преобразования первого операнда. Если тип `unsigned`, то освобождающиеся левые биты заполняются нулями. В противном случае они заполняются копией знакового бита.

Если второй операнд отрицателен, то результат операции сдвига не определен.

При выполнении операций сдвига ситуация потери значимости не контролируется. Если результат сдвига не может быть представлен типом первого операнда после преобразования, то результат неопределенный.

Пример:

```
unsigned int x, y, z;
    …
x = 0x00AA;
y = 0x5500;
z = (x << 8) + (y >> 8);/* x сдвигается влево на 8 позиций, а y
                           сдвигается вправо на 8 позиций.
                           Результаты сдвигов складываются, давая
                           значение 0xAA55, которое присваивается
                           z*/
```

Операции отношения

Операции отношения сравнивают первый операнд со вторым и вырабатывают целое значение `1` (`ИСТИНА`) или `0` (`ЛОЖЬ`). Результат имеет тип `int`. Имеются следующие операции отношения:

< - первый операнд меньше, чем второй операнд;
> - первый операнд больше, чем второй операнд;
<= - первый операнд меньше или равен второму операнду;
>= - первый операнд больше или равен второму операнду;
== - первый операнд равен второму операнду;
!= - первый операнд не равен второму операнду;

Операнды могут иметь целый, плавающий тип, либо быть указателями. Типы первого и второго операндов могут различаться. Перед выполнением операции над операндами выполняются преобразования по умолчанию.

Операндами любой операции отношения могут быть два указателя на один и тот же тип.

Сравнение между собой указателей двух произвольных объектов, вообще говоря, не имеет смысла. Однако сравнение указателей различных элементов одного и того же массива может быть полезным, поскольку элементы массива хранятся в памяти последовательно. Указатель предшествующего элемента массива всегда меньше, чем указатель последующего элемента.

Указатель можно проверять на равенство или неравенство системной символической константе `NULL` (ноль). Указатель, имеющий значение `NULL`, не указывает ни на какую область памяти. Он называется нулевым указателем.

Из-за специфики машинной арифметики не рекомендуется проверять плавающие значения на равенство, поскольку 1.0/3.0*3.0 не будет равно 1.0.

Поразрядные операции

Поразрядные операции - это "логическое И" (&), "включающее ИЛИ" (|) и "исключающее ИЛИ" (^), которые поразрядно выполняются над всеми разрядами своих операндов. Операнды поразрядных операций должны иметь целый тип, но бит знака, если он есть, также участвует в операции. Перед выполнением операции над операндами выполняются преобразования по умолчанию. Тип результата определяется типом операндов после преобразования.

Логические операции

Логические операции - это "логическое И" (&&) и "логическое ИЛИ" (||), выполняемые над логическими интерпретациями значений своих операндов. Операнды логических операций могут иметь целый, плавающий тип, либо быть указателями. Типы первого и второго операндов могут различаться. Логические операции не выполняют предварительных преобразований операндов по умолчанию.

При выполнении логической операции сначала вычисляется первый операнд, и полученный результат интерпретируется как ИСТИНА (значение не равно 0) или ЛОЖЬ (значение равно 0). Если значения первого операнда достаточно для определения результата логической операции, то второй операнд вообще не вычисляется! Если нет, то вычисляется второй операнд и полученный результат также интерпретируется как ИСТИНА или ЛОЖЬ. Итоговым результатом выполнения логической операции над вычисленными значениями является целое число типа int, имеющее значение 0 (ЛОЖЬ) или 1 (ИСТИНА).

Замечание. Указанную особенность выполнения логической операции необходимо учитывать, когда в качестве второго операнда выступает выражение, требующее обязательного вычисления.

Примеры:

```
Int x, y;
If ( x < y && y < z)  /* пример I */
printf("x меньше z\n");
If (x = = y!! x = = z) /* пример 2 */
printf("x равен У или z\n");
```

Операция последовательного вычисления

Операция последовательного вычисления последовательно вычисляет два своих операнда, сначала первый, затем второй. Оба операнда являются выражениями. Синтаксис операции:

<выражение1>, *<выражение2>*

Знак операции - запятая, разделяющая операнды. Результат операции имеет значение и тип второго операнда. Ограничения на типы операндов (т.е. типы результатов выражений) не накладываются, преобразования типов не выполняются.

Операция последовательного вычисления обычно используется для вычисления нескольких выражений в ситуациях, где. по синтаксису допускается только одно выражение.

Примеры:

```
for(i = j = 1; i + j < 20; I + = 1, j--)
func_two ((x--, y + 2), z);
```

Операции присваивания

В языке С присваивание относится не к операторам, а к операциям. Она вырабатывает значение, которое может быть использовано при вычислении выражения. Имеются следующие операции присваивания:

(**++**) *Унарный инкремент*

(**--**) *Унарный декремент*

(**=**) *Простое присваивание*

(***=**) *Умножение с присваиванием*

(**/=**) *Деление с присваиванием*

(**%=**) *Остаток от деления с присваиванием*

(**+=**) *Сложение с присваиванием*

(**-=**) *Вычитание с присваиванием*

(**<<=**) *Сдвиг влево с присваиванием*

(**>>=**) *Сдвиг вправо с присваиванием*

(**&=**) *Поразрядное логическое И с присваиванием*

(**|=**) *Поразрядное логическое включающее ИЛИ с присваиванием*

(**^=**) *Поразрядное логическое исключающее ИЛИ с присваиванием*

Левый (или единственный) операнд операции присваивания должен быть модифицируемым L-выражением. К L-выражениям относятся:

- имена переменных, объявленных без модификатора **const**;
- индексные выражения, значения которых не являются массивами;
- выражение выбора элемента, если выбираемый элемент сам является L-выражением;
- выражение косвенной адресации, если его значение не имеет тип массива или функции;
- выражение с операцией приведения типа над указателем;
- L-выражение в скобках.

При присваивании тип правого операнда преобразуется к типу левого операнда. Специфика этого преобразования зависит от обоих типов.

Преобразование типа значения при присваивании осуществляется даже тогда, когда оно влечет за собой существенное искажение значения.

Преобразования знаковых целых значений. Знаковое целое значение преобразуется к более короткому знаковому целому значению посредством усечения старших битов. Знаковое целое значение преобразуется к более длинному знаковому целому значению посредством расширения знакового разряда влево.

Преобразование знаковых целых значений к плавающим значениям происходит путем преобразования к типу `long`, а затем преобразования к плавающему типу. При этом возможна потеря точности.

При преобразовании знаковых целых значений к беззнаковому целому производится приведение к размеру беззнакового целого типа, и результат интерпретируется как беззнаковое целое значение. В итоге отрицательное значение теряется.

Преобразование беззнаковых целых значений. Беззнаковое целое значение преобразуется к короткому беззнаковому целому значению или короткому знаковому целому путем усечения старших битов. Безнаковое целое значение преобразуется к длинному беззнаковому целому значению или длинному знаковому целому путем расширения нулями влево. При преобразовании беззнакового целого значения к знаковому целому значению того же размера его битовое представление не меняется, но значение может измениться, если старший (знаковый) разряд был равен единице.

Преобразование беззнаковых или знаковых целых значений к плавающим значениям происходит путем преобразования к типу `long`, а затем преобразования к плавающему типу. При этом возможно искажение значения.

Преобразование плавающих типов. Значения типа `float` преобразуются к типу `double` без потери точности. Значение типа `double` при преобразовании к типу `float` представляется с потерей точности. Возможна даже потеря значимости, о чем сообщается во время выполнения программы.

Значения с плавающей точкой преобразуется к целым типам в два приема: сначала производится преобразование к типу `long` (дробная часть отбрасывается), а затем преобразование полученного значения к требуемому типу. Если полученное значение велико для типа `long`, то результат преобразования неопределенный.

Преобразование указателей. Указатель на значение одного типа может быть преобразован к указателю на значение другого типа. Однако компилятор выдаст предупреждающее сообщение, если только это не указатель на `void`, так как результат может оказаться семантически искаженным из-за изменения типа указываемого объекта. Указатель на `void` можно преобразовывать к указателю на любой тип и обратно.

Операции инкремента и декремента

Операции (**++**) и (**--**) инкрементируют (увеличивают на единицу) и

декрементируют (уменьшают на единицу) свой операнд. Операнд должен иметь целый, плавающий тип или быть указателем. В качестве операнда допустимо только модифицируемое L-выражение.

Операнды целого или плавающего типа увеличиваются или уменьшаются на целую единицу. Над операндом не производятся преобразования по умолчанию. Тип результата соответствует типу операнда. Операнд типа указатель инкрементируется или декрементируется на размер объекта, который он адресует. Инкрементированный указатель адресует следующий элемент данного типа, а декрементированный указатель - предыдущий.

Операции инкремента и декремента могут записываться как перед своим операндом (*префиксная* форма записи), так и после него (*постфиксная* форма записи). Для операции в префиксной форме операнд сначала инкрементируется или декрементируется, а затем его новое значение участвует в дальнейшем вычислении выражения, содержащего данную операцию. Для операции в постфиксной форме операнд инкрементируется лишь после того, как его старое значение поучаствует в вычислении выражения. Таким образом, в вычислении выражения участвует либо новое, либо старое значение операнда.

Примеры:

```
/* пример 1 */
  if(pos-- > 0) *p++ = *q++;
/* пример 2 */
  if ( line [--j] ! = '\n') retum;
```

В первом примере переменная `pos` проверяется на положительное значение, а затем декрементируется. Если значение `pos` до декремента было положительным, то выполняется следующий оператор. В нем значение, указуемое `q`, заносится по адресу, содержащемуся в `p`. После этого `p` и `q` инкрементируются.

Во втором примере переменная `j` декрементируется перед ее использованием в качестве индекса массива `line`.

Простое присваивание

При простом присваивании (=) значение правого операнда присваивается левому операнду. Левый операнд должен быть модифицируемым L-выражением. При присваивании выполняются правила преобразования типов. Операция вырабатывает результат, который может быть далее использован в выражении. Результатом операции является присвоенное значение. Тип результата - тип левого операнда.

Примеры:

```
    double x;
    int y;
…
x = y; /*Значение y преобразуется к double и присваивается x*/
```

```
    int a, b, c;
    …
    b = 2;
a = b + (c = 5);
```

Составное присваивание

Операция составного присваивания состоит из простой операции присваивания скомбинированной с какой-либо другой бинарной операцией. При составном присваивании вначале выполняется действие, специфицированное бинарной операцией, а затем результат присваивается левому операнду. Например, выражение составного присваивания со сложением имеет вид:

<*выражение1*> **+=** <*выражение2*>

По результату оно эквивалентно выражению:

<*выражение1*> **=** <*выражение1*> **+** <*выражение2*>

Значение операции вырабатывается по тем же правилам, что и для операции простого присваивания. Однако выражение составного присваивания по выполнению не эквивалентно обычной записи, поскольку в выражении составного присваивания выражение1 вычисляется только один раз, в то время как в обычной записи оно вычисляется дважды: в операции сложения и в операции присваивания. Например, выражение

```
strl.str2.ptr += 5;
```

короче и выполняется быстрее, чем

```
strl.str2.ptr = strl.str2.ptr + 5;
```

Использование составных операций присваивания может повысить эффективность программ. Каждая операция составного присваивания выполняет преобразования, которые определяются входящей в ее состав бинарной операцией, и соответственно ограничивает типы своих операндов. Результатом операции составного присваивания является значение, присвоенное левому операнду. Тип результата - тип левого операнда.

Условная операция

Условная операция является операцией с тремя операндами. Она имеет следующий синтаксис:

<*выражение1*> ? <*выражение2*> **:** <*выражение3*>

Выражение1 может иметь целый, плавающий тип, либо быть указателем. При выполнении операции сначала вычисляется выражение1 и его результат интерпретируется как ИСТИНА (значение не равно 0) или ЛОЖЬ (значение равно 0). Если выражение1 имеет значение ИСТИНА, то вычисляется выражение2 и результатом условной операции является значение выражение2. Если же выражение1 равно ЛОЖЬ, то вычисляется выражение3 и результатом является его

значение. В любом случае вычисляется только один из операндов, выражение2 или выражение3, но не оба.

Тип результата зависит от типов второго и третьего операндов (они могут различаться) следующим образом:

1) Если второй и третий операнды имеют целый или плавающий тип, то выполняются преобразования по умолчанию. Типом результата является тип операндов после преобразования.
2) Второй и третий операнды могут быть структурами, объединениями или указателями одного и того же типа. Типом результата будет тот же самый тип структуры, объединения или указателя.
3) Если либо второй, либо третий операнд имеет тип `void` (например, является вызовом функции, тип значения которой `void`), то другой операнд также должен иметь тип `void`, и результат имеет тип `void`.
4) Если либо второй, либо третий операнд является указателем на какой-либо тип, а другой является указателем на `void`, то результат имеет тип указателя на `void`.
5) Если либо второй, либо третий операнд является указателем, то другой операнд может быть константным выражением со значением 0 (`NULL`). Типом результата является указатель.

Пример:

```
j = (i < O) ? ( -i ) : ( i );/* j присваивается значение |i|*/
```

Приоритет и порядок выполнения операций

Приоритет и ассоциативность операций языка С влияют на порядок группирования операндов и вычисления операций в выражении. Приоритет операций существен только при наличии нескольких операций, имеющих различный приоритет. Выражения с более приоритетными операциями вычисляются первыми.

Ниже в таблице приведены операции в порядке убывания приоритета. Операции, расположенные в одной строке таблицы, или объединенные в одну группу, имеют одинаковый приоритет и одинаковую ассоциативность.

Из таблицы следует, что операнды, представляющие вызов функции, индексное выражение, выражение выбора элемента и выражение в скобках, имеют наибольший приоритет и ассоциативность слева направо. Приведение типа имеет тот же приоритет и порядок выполнения, что и унарные операции.

Выражение может содержать несколько операций одного приоритета. Когда несколько операций одного и того же уровня приоритета появляются в выражении, то они применяются в соответствии с их ассоциативностью - либо справа налево, либо слева направо.

Мультипликативные, аддитивные и поразрядные операции обладают свойством коммутативности. Это значит, что результат вычисления выражения,

включающего несколько коммутативных операций одного и того же приоритета, не зависит от порядка выполнения этих операций. Поэтому компилятор оставляет за собой право вычислять такие выражения в любом порядке, даже в случае, когда в выражении имеются скобки, специфицирующие порядок вычисления. Операция унарного плюса позволяет гарантировать порядок вычисления выражений в скобках.

Операция последовательного вычисления, логические операции **И** и **ИЛИ**, условная операция и операция вызова функции гарантируют определенный порядок вычисления своих операндов. Операция последовательного вычисления обеспечивает вычисление своих операндов по очереди, слева направо (запятая, разделяющая аргументы в вызове функции, не является операцией последовательного вычисления и не обеспечивает таких гарантий). Гарантируется лишь то, что к моменту вызова функции все аргументы уже вычислены. Условная операция вычисляет сначала свой первый операнд, а затем, в зависимости от его значения, либо второй, либо третий. Логические операции также обеспечивают вычисление своих операндов слева направо. Однако логические операции вычисляют минимальное число операндов, необходимое для определения результата выражения. Таким образом, второй операнд выражения может вообще не вычисляться.

Знак операции	Наименование операции	Ассоциативность
`() [] . ->`	Вызов функции, индексное выражение, выражение выбора	Слева направо
`- + ! * & ++ -- sizeof Приведение типа`	Унарные	Справа налево
`* / %`	Мультипликативные	Слева направо
`+ -`	Аддитивные	Слева направо
`<< >>`	Сдвиг	Слева направо
`== !=`	Отношение	Слева направо
`&`	Поразрядное логическое **И**	Слева направо
`^`	Поразрядное логическое исключающее **ИЛИ**	Слева направо
`|`	Поразрядное логическое включающее **ИЛИ**	Слева направо
`&&`	Логическое **И**	Слева направо
`||`	Логическое **ИЛИ**	Слева направо
`? :`	Условная операция	Справа налево
`= *= /= %= += -= <<= >>= &= |= ^=`	Простое и составное присваивание	Справа налево

,	Последовательное вычисление	Слева направо

Побочные эффекты при вычислении выражений

Побочный эффект выражается в неявном или неопределенном изменении значения переменной в процессе вычисления выражения. Побочный эффект могут вызывать все операции присваивания. Вызов функции, в которой изменяется значение какой-либо внешней переменной, либо путем явного присваивания, либо через указатель, также может иметь побочный эффект.

Приведем примеры побочных эффектов:

```
add(i+1, i=j+2);
```

Аргументы вызова функции `add` могут быть вычислены в любом порядке. Выражение `i+1` может быть вычислено перед выражением `i=j+2`, или после него, с различным результатом в каждом случае.

Унарные операции инкремента и декремента также содержат в себе присваивание и могут быть причиной побочных эффектов, как это показано в следующем примере:

```
int i, a[10];
i = 0;
a[i++] = i;
```

Неизвестно, какое значение будет присвоено элементу `a[0]` - нуль или единица, поскольку для операции присваивания порядок вычисления аргументов (выражение слева и справа) не оговаривается.

При разработке программ необходимо учитывать и избегать программных конструкций и выражений с возможными побочными эффектами.

ОПЕРАТОРЫ

Пустой оператор

Синтаксис:

```
;
```

Пустой оператор-это оператор, состоящий только из точки с запятой. Он может появиться в любом месте программы, где по правилам синтаксиса требуется оператор. Выполнение пустого оператора не меняет состояния программы. Пустой оператор применяется в том случае, когда синтаксис программной конструкции требует наличия оператора, но семантика программы никаких действий не требует (например, пустое тело оператора цикла).

Пустой оператор, подобно любому другому оператору языка С, может быть помечен меткой.

Составной оператор

Синтаксис:

```
{
 [ <объявления> ]
 ...
 [ <оператор> ]
 ...
 [ <оператор> ]
}
```

Квадратные скобки говорят о необязательности заключенных в них программных конструкций. В качестве составного оператора выступает простой блок. Действие составного оператора заключается в последовательном выполнении содержащихся в нем операторов, за исключением тех случаев, когда некий оператор явно передает управление в другое место программы.

В начале составного оператора могут содержаться объявления. Они служат для определения переменных, локальных для данного блока, либо для распространения на данный блок области действия глобальных объектов. Типично использование составного оператора в качестве тела другого оператора, например оператора `if`.

Пример:

```
if(i>0){
  line[i] = x;
  x++;
  i--;
}
```

Подобно другим операторам языка С, любой оператор внутри составного оператора может быть помечен. Передача управления по метке внутрь составного оператора возможна, однако если составной оператор содержит

объявления переменных с инициализацией, то при входе в блок по метке эта инициализация не будет выполнена и значения переменных будут непредсказуемы. Можно поставить метку и на сам составной оператор.

Оператор-выражение

Синтаксис:

<выражение>;

Выражение, за которым следует точка с запятой, приобретает статус оператора. Выполнение оператора-выражения заключается в вычислении выражения. Оператор-выражение может быть записан лишь там, где по синтаксису допустим оператор.

Примеры:

```
x = y + 3;//Выражение с операцией присваивания
func(x);//Выражение вызова функции
```

Условный оператор

Синтаксис:

```
 if(<выражение>) <оператор1>
[ else <оператор2> ]
```

Выполнение условного оператора `if` начинается с вычисления значения выражения. Если значение выражения не равно нулю (ИСТИНА), то выполняется опрератор1. Если же значение выражения отлично от нуля (ложь). то выполняется оператор2 непосредственно следующий за ключевым словом `else`. Если конструкция `else` опущена, то управление передается на оператор, следующий в программе за оператором `if`.

На ключевое слово `if` можно поставить метку.

Пример:

```
if(i > 0) y = x/i;
else{
    x = 1;
    y = f(x);
}
```

Оператор пошагового цикла

Синтаксис:

```
for( [ <начальное_выражение> ];
     [ <условное_выражение> ];
     [ <выражение-приращения> ] )
<оператор>
```

Тело цикла `for` – *оператор*, выполняется до тех пор, пока *условное выражение* не станет ложным. Если оно изначально ложно, то тело цикла не будет выполнено ни разу. *Начальное выражение* и *выражение приращения* это

обычные выражения, которые могут использоваться для инициализации и модификации параметров цикла или других значений.

Первым шагом при выполнении оператора цикла `for` является вычисление начального выражения, если оно имеется. Затем вычисляется условное выражение и производится его оценка следующим образом:

1) Если значение условного выражения есть ИСТИНА, то выполняется оператор. Затем вычисляется выражение приращения (если оно есть), и процесс повторяется.
2) Если условное выражение имеет значение ЛОЖЬ, то выполнение оператора `for` заканчивается. Оператор `for` завершиться и при выполнении операторов `break`, `goto`, `return`.
3) Если условное выражение опущено, то его значение считается ИСТИНА и процесс выполнения продолжается, как описано выше. В этом случае имеем бесконечный цикл, который может завершиться только при выполнении операторов `break`, `goto`, `return`.

Пример:

```
for (i=0, j=0; func(i,j); i++, j--) ;//Цикл с пустым оператором
```

Оператор цикла с предусловием

Синтаксис:

`while` <*выражение*>
<*оператор*>

Оператор является телом цикла. Вначале вычисляется *выражение.* Если значение выражения логически интерпретируется как ЛОЖЬ (т.е. равно нулю), то оператор не выполняется, `while` завершается и управление сразу передается на следующий за циклом оператор программы. Если значение выражения логически интерпретируется как ИСТИНА, то выполняется оператор и цикл повторяется. Перед каждым следующим выполнением оператора цикла выражение вычисляется заново. Тело оператора цикла `while` выполняется до тех пор, пока значение выражения не станет ЛОЖЬ. Этот процесс повторяется до тех пор, пока выражение не получит значение ЛОЖЬ. Оператор цикла `while` может также завершиться при выполнении в теле цикла операторов `break`, `goto`, `return`.

Пример:

```
while(i>=0){
  string1[i] = string2[i];
  i--;
}
```

Оператор цикла с постусловием

Синтаксис:

`do`

<*оператор*>
while (<*выражение*>);

Тело оператора цикла do выполняется один или несколько раз до тех пор, пока значение выражения не станет ЛОЖЬ (равным нулю). Вначале выполняется тело цикла - *оператор,* затем вычисляется условие - *выражение*. Если выражение ложно, то оператор цикла do завершается и управление передается следующему оператору программы. Если значение выражения ИСТИНА (не равно нулю), то тело цикла выполняется снова, и снова вычисляется выражение. Выполнение тела оператора цикла do повторяется до тех пор, пока выражение не станет ложным. Оператор do может также завершиться при выполнении в своем теле операторов break, goto, return.

Пример:

```
do{
   y = f(x);
   x--;
}
while(x > 0);
```

Оператор продолжения

Синтаксис:

continue;

Оператор продолжения continue может использоваться только в теле операторов цикла do, for, while. Он завершает выполнение текущей итерации цикла и передает управление на следующую итерацию в операторах цикла. В операторах цикла do и while следующая итерация начинается с вычисления условного выражения. Для оператора for следующая итерация начинается с вычисления выражения приращения, а затем происходит вычисление условного выражения.

Пример:

```
whi1e(i-- > 0){
   if((x = f(i))==1) continue;
   y += x*x;
}
```

Оператор-переключатель

Синтаксис:

switch(<*выражение*>){

[<*объявления*>]
[case <*константное_выражение*>:][<*оператор*>]
…
[case <*константное-выражение*>:][<*оператор*>]
…

```
[ default: <оператор> ]
…
}
```

Оператор-переключатель `switch` предназначен для выбора одного из нескольких альтернативных путей выполнения блока переключателя (конструкция в фигурных скобках). Для этого в блоке переключателя содержатся конструкции `case` с константными выражениями целого типа, используемыми как метки операторов, на которые передается управление при входе в блок переключателя. Значение каждого константного выражения должно быть уникальным внутри блока переключателя.

Выполнение оператора-переключателя начинается с вычисления значения *выражения* переключателя (выражения в круглых скобках) целого типа. После этого управление передается одному из *операторов* блока переключателя, перед которым стоит `case` с константным выражением, значение которого совпадает со значением выражения переключателя. Выполнение блока переключателя `switch` начинается с выбранного таким образом оператора и продолжается до конца блока или до тех пор, пока не выполнится оператор `break` или очередной оператор не передаст управление за пределы блока переключателя.

Если ни одно из константных выражений `case` не равно значению выражения переключателя, то выполнение блока переключателя `switch` начинается с оператора, следующего за ключевым словом `default`. Если же слово `default` опущено, то ни один оператор в блоке переключателя не выполняется и управление передается следующему за переключателем оператору программы.

Использование объявлений в блоке переключателя `switch` имеет существенную особенность, заключающуюся в том, что инициализаторы, включенные в объявления, не будут выполнены, поскольку управление непосредственно передается на выполняемый оператор внутри блока, обходя строки, которые содержат инициализацию.

Пример:

```
int n, z, p;
switch(i){
int r=5;//Инициализация r=5 не будет выполнена!
  case -1: n++; break;
  case  0: z++; break;
  case  1: p++; break;
  default: r+=n+z+p;
…
}
```

Оператор в блоке переключателя может быть помечен более одной меткой **`case`**, как показано в следующем примере:

```
switch (c){
  case 'a':
  case 'b':
  case 'c':
     hexcvt(c);
     …
}
```

Оператор разрыва

Синтаксис:

```
break;
```

Оператор разрыва `break` может содержаться только в теле операторов цикла `do`, `for`, `while` или оператора-переключателя `switch`. Он прерывает выполнение этих операторов, и управление передается оператору программы, следующему за прерванным.

Если оператор разрыва `break` записан внутри вложенных операторов `do`, `for`, `while`, `switch`, то он завершает только непосредственно охватывающий его оператор `for`, `while`, `switch`.

Пример:

```
for(i = 0; i < LENGTH; i++){
  for(j = 0; j < WIDTH; j++){
    if(lines[i][j] == '\0') break;
  }
  lengths[i] = j;
}
```

Оператор перехода

Синтаксис:

```
goto <метка>;
…
<метка>: <оператор>
…
```

Оператор перехода **`goto`** передает управление непосредственно на *оператор*, помеченный *меткой*. Метка представляет собой обычный идентификатор. Метка оператора имеет смысл только для оператора `goto`. Область действия метки ограничивается функцией, в которой она определена; из этого следует, во-первых, что каждая метка должна быть отлична от других меток в той же самой функции; во-вторых, что нельзя передать управление по оператору `goto` в другую функцию. Однако можно войти в блок, тело цикла, условный оператор, оператор-переключатель по метке.

Пример:

```
if (errorcode > 0)goto exit;
…
```

```
exit: return (errorcode);
```

Оператор возврата

Синтаксис:

`return[ <`*выражение*`> ];`

Оператор возврата `return` заканчивает выполнение функции, в которой он содержится, и возвращает управление в вызывающую функцию. Управление передается в точку вызывающей функции, непосредственно следующую за оператором вызова функции. Значение *выражения*, если оно задано, вычисляется, приводится к типу, объявленному для функции, содержащей оператор возврата `return`, и возвращается в качестве значения функции. Если выражение опущено, то возвращаемое функцией значение не определено.

Пример:

```
long sq(int x){
     return x*x;
}
```

Создание исполняемого модуля программы в QNX

Соглашения по работе с командной строкой

В QNX приняты традиционные для UNIX-систем соглашения об интерпретации аргументов командной строки. Аргументы подразделяются на две категории: *опции* (или *флаги*) и все остальные. Опции меняют поведение программы, а остальные аргументы содержат разного рода входные данные (например, названия файлов). Опции бывают двух видов.

– *Короткие опции* состоят из дефиса и одиночного символа (обычно это буква в нижнем или верхнем регистре). Такие опции быстрее и проще набирать.

– Длинные опции состоят из двух дефисов, после которых следует имя, содержащее буквы нижнего и верхнего регистров и дефисы. Такие опции легче запоминать и читать (например, в командных сценариях).

Обычно системные программы поддерживают обе разновидности каждой опции: первую - для краткости, вторую - для ясности. Например, большинство программ понимает опции -h и --help и трактуют их одинаково. Как правило, они указываются в командной строке сразу после имени программы. После опций могут идти другие аргументы, в частности имена файлов и другие входные данные.

Основные команды управления

Ниже приводится список наиболее ходовых команд управления операционной системой.

- pwd – показывает текущий каталог
- cd – смена каталога
- ls – выдача содержимого указанного каталога
- mkdir – создание каталога
- rm – удаление файлов или каталогов
- cp – копирование файлов
- gzip и gunzip – работа с gzip архивами
- ps – выдает список запущенных процессов и их `pid`
- kill - посылает процессу сигнал, если kill без параметров посылает процессу сигнал `SIGTERM`
- sin – в зависимости от параметров показывает информацию о системе, состояния процессов, сети, переменных окружения и т.п.

Если какая-либо утилита содержит подсказку о своей работе, то эту подсказку можно получить при помощи команды `use`. Например:

```
# use mount
```

В результате выполнения этой команды на экран выдается информация о порядке использования утилиты mount. В остальных случаях необходимо обращаться к справочной системе QNX. В GUI Photon справочная система запускается кнопкой Help.

В каталоге /usr/photon/bin находятся утилита GUI Photon, которая называется psin. Она выполняет функции инспектора процессов и выдает информацию о состоянии загруженных процессах в удобной графической форме.

Установка компилятора

В качестве стандартного компилятора в QNX входит компилятор – qcc.

Для того, чтобы инсталлировать компилятор qcc и ряд других средств программирования на языке C, необходимо после установки QNX 6 выполнить следующие действия, используя GUI Photon:

- запустить Launch -> Software -> Package Maneger
- выбрать CD-ROM Repository
- в разделе "QNX Realtime Platform -> Development выбрать "C/C++ Toolset" и "Development Core Targetting x86"
- после этого нажать Install

Примечание. Установка средств разработки приложений для версии QNX 6.2 (Momentics) производится автоматически при установке операционной системы, которая подробно описана на сайте представителей QSSL в России фирмы SWD по адресу - http://www.swd.ru/qnx/support/docs/qnx62install.html.

Компиляция исходных модулей

Процедура создания программ на языке C приведена на рис.1. Написание программ осуществляется с использованием любого текстового редактора, входящего в состав системных средств QNX. Создать текстовый файл можно, например, с использование стандартного текстового редактора Editor или с помощью встроенного редактора оболочки Midnight Commander.

В результате создаются один или более *текстовых* файлов (исходных модулей) программы. При указании имени файла исходного модуля программы предлагается использовать стандартное расширение ".c".

Первой фазой является стадия компиляции, когда файлы исходных модулей программы с включенными в них текстами файлов заголовков обрабатываются

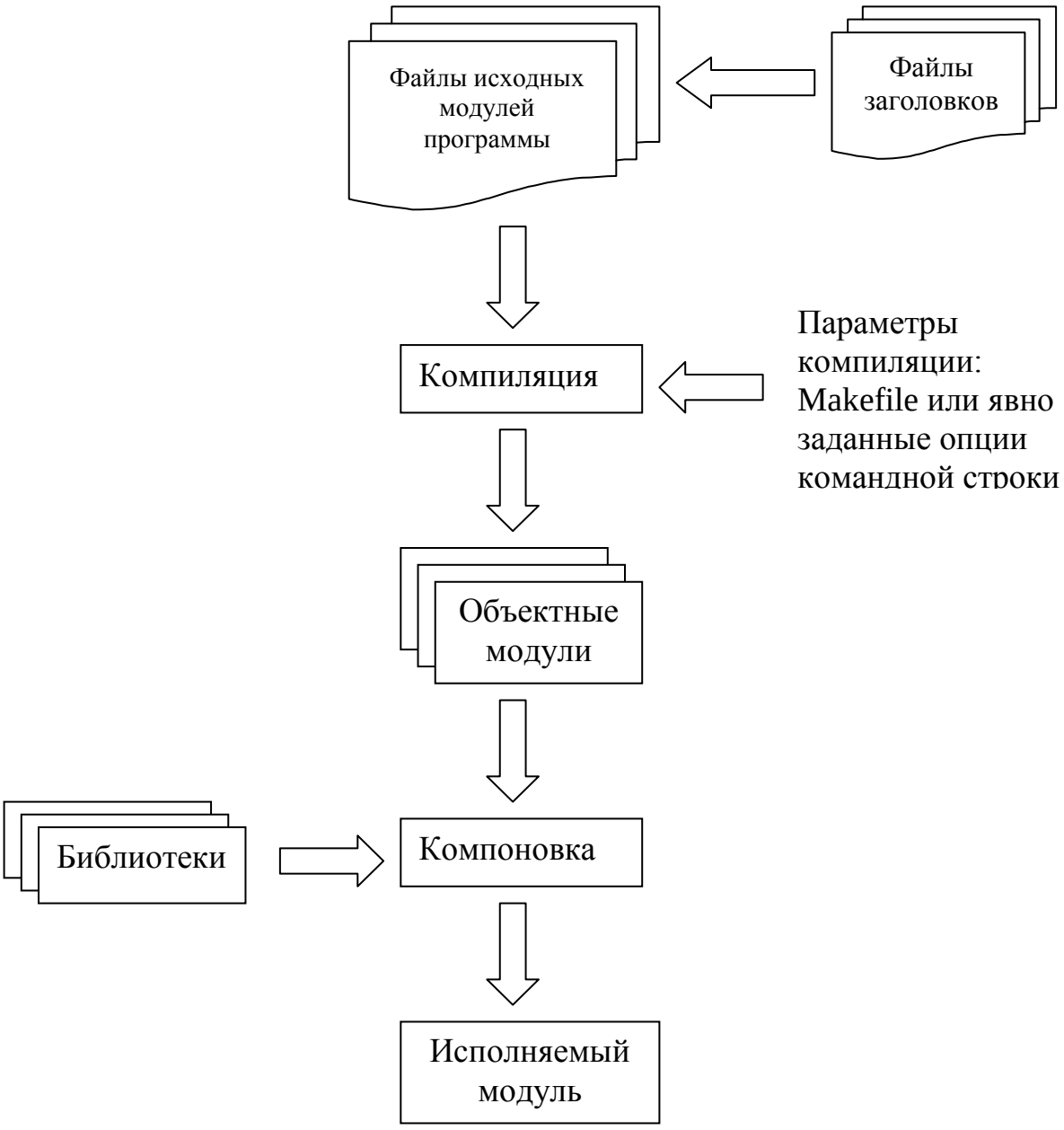


Рис. 1. Схема создания исполняемого модуля программы

компилятором qcc. Параметры компиляции задаются либо явным указанием необходимых опций в командной строке запуска компилятора, либо с помощью файла Makefile, если используется утилита make. Чтобы при компиляции исходного модуля получить объектный модуль, нужно указать опцию "-c". Эта опция говорит компилятору о необходимости получить на выходе объектный модуль. Без неё компилятор сразу попытается скомпоновать исполняемый модуль. Это возможно только в простых случаях, когда программа представляется одним исходным модулем. В общем случае для каждого

исходного модуля компилятор должен предварительно создать объектный модуль. Имена файлов объектных модулей имеют стандартное расширение ".o". Например, для компилирования исходного модуля программы example.c и получения объектного модуля example.o необходимо выполнить следующую команду:

```
# qcc -Vgcc_ntox86 –c example.c
```

Ключ -V с параметром gcc_ntox86 определяет режим компиляции программы для платформы на базе процессора Intel.

Для включения в программу файлов заголовков препроцессор компилятора по умолчанию просматривает текущий каталог, а также каталоги, где установлены файлы стандартных библиотек. Чтобы указать компилятору дополнительные каталоги для поиска файлов заголовков, необходимо использовать опцию "-I". Следующая команда дает компилятору указание дополнительно искать файлы заголовков в каталоге ../include:

```
# qcc -Vgcc_ntox86 –c -I ../include example.c
```

Если компилируемый исходный модуль использует макроконстанту, то эту макроконстанту можно определить непосредственно в командной строке, используя опцию "-D". Можно, конечно, явно добавить в исходный текст директиву `#define`, но при изменении её значения потребуется каждый раз изменять исходный текст программы. Проще сделать то же самое в командной строке:

```
# qcc -Vgcc_ntox86 –c -D LENGTH=80 example.c
```

При завершении отладки программы для получения её коммерческого варианта может оказаться полезным использование при компиляции средств оптимизации кода исполняемого модуля. Есть несколько уровней оптимизации, для большинства случаев подходит второй. Для включения средств оптимизации используется опция "-O<уровень оптимизации>". Следующая команда компилирует файл с включенным режимом оптимизации второго уровня:

```
# qcc -Vgcc_ntox86 –c -O2 example.c
```

Заметим, что включение средств оптимизации иногда приводит к проявлению скрытых ошибок, незаметных ранее.

Компоновка объектных модулей

На следующей стадии осуществляется компоновка из созданных объектных модулей и модулей, содержащихся в библиотеках (включая стандартную библиотеку по умолчанию и библиотеки, указанные пользователем к командной строке в качестве параметров), исполняемого модуля программы. Компоновка осуществляется также с использованием компилятора qcc.

Библиотеки бывают *статические* (их имена имеют расширение ".a") и *разделяемые* (имена этих библиотек имеют расширение ".so"). Указание в командной строке пользовательских библиотечных файлов осуществляется с

помощью опции "-l<имя файла>". Как и в случае с файлами заголовков, компилятор ищет библиотечные файлы в стандартных каталогах, в частности /lib и /usr/lib. Для задания дополнительных каталогов предназначена опция "-L<имя каталога>".

Для указания имени исполняемого модуля используется опция "-o <имя модуля>". Имена компонуемых объектных модулей вместе с опциями включаются в командную строку в качестве аргументов.

Следующая команда осуществляет компоновку исполняемого модуля example из двух объектных модулей example.o main.o и использует, кроме стандартных, дополнительную библиотеку mylib:

```
# qcc -Vgcc_ntox86 –o example example.o main.o -lmylib -L/usr/local
```

В отличие от препроцессора, компоновщик просматривает лишь стандартные каталоги. Поэтому, если библиотечный файл находится в текущем каталоге, об этом нужно явно сообщить с помощью опции "-L.". Например, при выполнении следующей команды компоновщик будет искать библиотеку в текущем каталоге:

```
# qcc -Vgcc_ntox86 –o example example.o main.o -L. -lmylib
```

В простых случаях все фазы автоматически выполняются вызовом команды:

```
# make example
```

или эквивалентной ей

```
# qcc -Vgcc_ntox86 –o example example.c
```

В результате в текущем каталоге создается исполняемый модуль с именем `example` (стандартно - без расширения). При просмотре каталогов в оболочке `MC` перед именем файла появится символ <*>, который указывает на то, что файл содержит исполняемый модуль.

Если имя исполняемого модуля в команде явно не задано, то по умолчанию будет создан исполняемый модуль с именем `a.out`.

Запуск исполняемого модуля на выполнение из текущего каталога осуществляется командой:

```
# ./<имя модуля> <аргументы функции main>
```

Например,

```
# ./example
```

Утилита make

Утилита make предназначена для автоматизации процесса управления выполнением команд при компиляции и компоновке исполняемого модуля для обеспечения целостности проекта, когда количество входящих в проект модулей достаточно велико. Дело в том, что при внесении изменений в некоторые модули программы необходимо следить, чтобы согласовано были переделаны и другие модули, зависящие от этих изменений. Например, если изменился некоторый

исходный модуль, то необходимо переделать и соответствующий объектный модуль, а также исполняемый модуль.

В процессе выполнения утилита make использует указания из специального текстового файла с именем Makefile. Файл Makefile как раз и содержит описание подобных зависимостей, определяющих порядок выполнения действий, необходимых для поддержания целостности проекта программы. Существуют определенные правила построения файла Makefile. Суть этих правил заключается в том, чтобы с помощью механизма меток задать *зависимости*, определяющие порядок выполнения заданных действий. В качестве меток в файле используются имена модулей, составляющих проект программы, и с ними связываются действия, которые необходимо выполнять для поддержания целостности проекта.

Метки, заданные в файле Makefile, используются в качестве аргументов при запуске утилиты make на выполнение:

```
# make <МЕТКА>
```

Первым начинается выполняться действие, соответствующее указанной метке. Последующие действия определяются установленными в файле `Makefile` между метками зависимостями. Если аргумент не задан, то выполнение `make` начинается с первой по порядку метки.

Порядок формирования содержимого файла `Makefile`, обеспечивающего компиляцию и компоновку исполняемого модуля из нескольких исходных модулей поясним на примере. Пусть проект программы включает в себя следующие файлы:

- Std.h
- Main.c
- Func1.c
- Main.o
- Func1.o
- Prog

Необходимо при работе над проектом обеспечить его целостность. Для этого воспользуемся утилитой make, для которой сформируем файл Makefile. Его содержимое будет таким:

```
CFLAGS = -Vgcc_ntox86

Prog:       Main.o Func1.o
            qcc $(CFLAGS) -o Prog Main.o Func1.o

Main.o:     Main.c Std.h
            qcc $(CFLAGS) -c Main.c

Func1.o:    Func1.c Std.h
            qcc $(CFLAGS) -c Func1.c

clean:
            rm -f *.o Prog
```

Файл Makefile включает в себя совокупность блоков с метками. Имена модулей, играющие одновременно роль меток, перечислены слева и заканчиваются двоеточием. Далее в строке перечислены имена модулей, от которых зависит целостность данного модуля и при изменении которых должна быть выполнена команда, заданная в следующей строке.

Кроме того, в него включен блок с меткой clean. Он не связан ни с какими модулями программы, и предназначен для выполнения команды удаления ненужных модулей:

```
# make clean
```

Запись $(CFLAGS) обозначает параметр настройки файла Makefile. Его можно определить либо непосредственно в файле Makefile (как в примере), либо в командной строке вызова утилиты make:

```
# make CFLAGS=-Vgcc_ntox86
```

Рассмотрим еще один пример файла Makefile. Пусть необходимо откомпилировать два исходных модуля на языке C, которые имеют имена manager.c и generator.c. Первые четыре строки файла подготавливают вызов компилятора qcc с режимом компиляции для платформы x86. Последующие строки начинаются соответственно с меток all, manager, generator и clean. Эти метки при запуске передаются в утилиту make в качестве параметров, управляющих её работой.

```
CC = qcc
LD = qcc
CFLAGS = -Vgcc_ntox86
LDFLAGS = -Vgcc_ntox86

all:    clean manager generator
manager:    manager.o
        $(LD) $(LDFLAGS) manager.o -o manager
generator:    generator.o
        $(LD) $(LDFLAGS) generator.o -o generator
clean:
        rm -f *.err *.o manager generator *.log
```

Если теперь утилиту make вызвать с параметром manager:

```
# make manager
```

то будет выполнена последовательность действий, необходимая для компиляции модуля manager.c в исполняемый файл manager с созданием объектного файла manager.o. При запуске утилиты в форме

```
# make all
```

будут последовательно выполнены предписания под метками clean, manager и generator. При запуске make без параметров будут выполнены предписания для первой метки (в нашем случае all). Вызов make с параметром clean позволяет

удалить из текущего каталога все файлы с расширениями *.err, *.o, *.log, а также исполняемые файлы manager и generator.

Создание библиотечных файлов

При создании сложных многомодульных программ целесообразно уже отлаженные объектные модули архивировать в библиотечном файле. При компоновке программы вместо длинного списка объектных модулей в команде достаточно указать содержащий их библиотечный файл (библиотеку). Для создания и работы с библиотечными файлами в QNX используется команда "ar". Использование команды рассмотрим на примере. Пусть требуется поместить в библиотеку mylib.a (одновременно и создать, если она отсутствует) объектные модули mod1.o и mod2.o. Для этого необходимо выполнить команду:

```
# ar rv mylib.a mod1.o mod2.o
```

Чтобы получить список модулей, содержащихся в библиотеке, достаточно выполнить команду:

```
# ar t mylib.a
```

Полную информацию о команде "ar" можно получить, используя команду "use".

Средства отладки программ

В качестве средства отладки программ QNX включает в себя отладчик gdb. Это программа, под управлением которой можно запустить исполняемый модуль программы. Чтобы можно было воспользоваться отладчиком, необходимо скомпилировать программу с включением в неё отладочной информации. Этой цели служит опция "-g" компилятора. В результате компилятор включает дополнительную информацию в объектные и исполняемые модули, позволяющую связать исходный текст программы с исполняемым модулем.

Чтобы воспользоваться отладчиком gdb, его необходимо запустить, указав в качестве аргумента имя исполняемого модуля:

```
# gdb <имя модуля>
```

После запуска появляется строка приглашения для ввода команд отладчика:

```
(gdb)
```

Перечислим наиболее ходовые команды отладчика:

- run - запустить исполняемый модуль.
- where - показать содержимое стека.
- up -движение вверх по стеку.
- print - показать значение переменной.
- break - установить контрольную точку (точка остановки).
- next - перейти после остановки на следующую строку.
- step - войти внутрь функции.

Например, запуск исполняемого модуля осуществляется командой:

```
(gdb)run
```

Подробную информацию о командах отладчика можно получить с помощью команды help. Например:

```
(gdb) help run
```

Создание графических пользовательских интерфейсов

GUI Photon включает в себя средство для визуального программирования прикладных графических пользовательских интерфейсов на языке C и C++ для взаимодействия с приложениями. Оно называется Photon Application Builder (PhAB). PhAB позволяет конструировать графический образ пользовательского интерфейса из графических объектов, называемых виджетами. На основе сконструированного образа интерфейса осуществляется автоматическая генерация текста программы на языке C или C++.

Запуск PhAB осуществляется из GUI Photon командой Launch/Development/Builder.

Федеральное агентство по образованию

**Государственное образовательное учреждение
высшего профессионального образования
"Самарский государственный аэрокосмический
университет имени академика С.П. Королёва"**

А.В. Баландин

Средства и методы разработки программных систем реального времени

Часть 2

Программирование параллельных процессов
в ОСРВ QNX Neutrino

Учебное пособие

**Самара
2012**

УДК 681.3.066
ББК 32.973.26-018.2

Баландин А.В. Средства и методы разработки программных систем реального времени. Учебное пособие: В 4 ч. Ч.2. **Программирование параллельных процессов в ОСРВ QNX Neutrino**. - Самар. гос. аэрокосм. ун-т. Самара, 2012. 105 с.

В пособии рассмотрены средства операционной системы ОСРВ QNX Neutrino (QNX 6) для управления файлами и программирования параллельных процессов на языке С.

Пособие предназначено для студентов, обучающихся по направлению 010300.68 «Фундаментальная информатика и информационные технологии», изучающих дисциплину «Технологии промышленного программирования».

ОГЛАВЛЕНИЕ

ЧАСТЬ 2. ОПЕРАЦИОННАЯ СИСТЕМА РЕАЛЬНОГО ВРЕМЕНИ QNX Neutrino

Операционная система реального времени QNX Neutrino имеет второе название – QNX 6. Далее для ссылки на операционную систему QNX Neutrino (QNX 6) будем использовать короткое название – QNX.

Файловое пространство QNX

Организация файлового пространства QNX

В QNX учёт файлов организован в виде древовидной структуры (дерева), называемой *файловым пространством*. Корнем этого дерева является *корневой каталог*, имеющий имя </>. Все доступное пользователям файловое пространство объединено в единое дерево каталогов[1], корнем которого является каталог с именем </>. Поэтому полное (*абсолютное*) имя любого файла начинается с </>. QNX располагает средствами, позволяющими объединять различные файловые системы, находящиеся на различных устройствах в единое файловое пространство. Поэтому файловое пространство в общем случае не однородно. Единое дерево файлового пространства, такое, каким его видит пользователь системы, может строиться из отдельных файловых систем в общем случае разных типов, которые могут располагаться на различных устройствах и иметь различные внутренние организации.

Для каждого процесса ОС ведет два атрибута, связанных с файловой системой – атрибут, указывающий на каталог, который рассматривается для процесса в качестве *корневого*, и атрибут, указывающий на каталог, который рассматривается для процесса в качестве *текущего рабочего*. В связи с этим процесс может адресоваться к файлу по имени либо в абсолютном, либо в относительном формате. Имя файла состоит из последовательности компонентов – локальных имен, разделенных символами '/'. Каталог, содержащий локальное имя, считается *родительским каталогом* этого имени. Последовательность имен каталогов, предшествующая последнему локальному имени в имени файла, считается *префиксом имени файла*. *Абсолютное* имя файла начинается с символа '/', обозначающего корневой каталог. Например, `/user/bin/sh` является абсолютным именем файла `sh`, а `/user/bin/` - его префикс. Если имя файла не начинается с символа '/', то оно обозначает путь в файловой системе от текущего каталога процесса до указанного файла. Такое имя считается *относительным*. Например, указав имя файла как `mydir/test1.c`, следует полагать, что в текущем каталоге процесса присутствует каталог с именем `mydir`.

Каждый каталог всегда содержит две скрытых системных жестких связей, ссылающихся на каталоги. Первая связь имеет локальное имя ".". Она ссылается на свой родительский каталог. Вторая ссылка имеет локальное имя "..". Она ссылается на родительский каталог своего родительского каталога. Исключение составляет корневой каталог. Для него имя "." означает то же, что и имя "..".

[1] В ОС Windows файловая система разбита по устройствам, а системная папка "Рабочий стол" играет роль общего корня.

Например, имя `../andy/mydir/test1.c` заставляет искать файл, начав с поиска локального имени `andy` в каталоге, являющемся родительским для текущего каталога.

Поиск файла по имени состоит в последовательном просмотре каталогов, указанных в префиксе, и поиске очередного локального имени.

Базовая структура файлового пространства QNX

Изначально файловое пространство QNX не является пустым (состоящим из одного корневого каталога файловой системы QNX с именем /) Файловая система QNX имеет базовую структуру системных каталогов и файлов, обеспечивающую работу ОС и её администрирование. Нарушение этой структуры может привести к неработоспособности системы или отдельных её компонентов. Типичная базовая структура файловой системы QNX включает в себя следующие каталоги.

Корневой каталог

Корневой каталог </> является основой файловой системы. Все остальные каталоги и файлы располагаются в структуре корневого каталога независимо от их физического местонахождения. Рассмотрим основные системные каталоги, к которым относятся следующие каталоги:

/bin

В каталоге `/bin` находятся наиболее часто употребляемые команды и утилиты системы, как правило, общего пользования.

/dev

Каталог `/dev` содержит файлы устройств. Каталог может содержать подкаталоги, группирующие файлы устройств одного типа.

/etc

В этом каталоге находятся системные конфигурационные файлы и многие утилиты администрирования, а также скрипты инициализации системы.

/lib

В каталоге `/lib` находятся библиотечные файлы языка С.

/root

Каталог пользователя с именем `root`. При входе в систему под этим именем этот каталог становится текущим каталогом.

/fs

Это стандартный каталог для монтирования файловых систем физических устройств к структуре корневого каталога для получения единого дерева логической файловой системы.

/home

Каталог общего употребления для размещения домашних каталогов пользователей при их регистрации в системе.

/usr

В этом каталоге находятся подкаталоги различных сервисных подсистем, исполняемые файлы утилит QNX, заголовочные файлы и т.д.

/var

Этот каталог используется для хранения временных файлов различных сервисных подсистем.

/tmp

Каталог хранения временных файлов, необходимых для работы различных подсистем QNX, а также пользователей системы.

/x86

В структуре этого каталога содержатся средства, обеспечивающие разработку программ на языке С для исполнения на базе платформы intel. В нем, в частности, находится утилита-компилятор qcc.

Монтирование файловых систем

Как правило, приложения не работают с блочными устройствами напрямую (как с физическим носителем). Полагается, что в каждом физическом разделе блочного устройства (например, жесткого диска) содержится файловая система некоторого типа. Для доступа к её содержимому она должна быть встроена в дерево корневой файловой системы QNX в виде вновь создаваемого каталога. Эта операция встраивания называется *подключением* или *монтированием* файловой системы устройства. После монтирования доступ к файловой системе устройства осуществляется в виде доступа к содержимому данного каталога.

При загрузке QNX автоматически подключает файловые системы, находящиеся в разделах НЖМД, помещая их в каталог `/fs` в виде подкаталогов с автоматически формируемыми именами. Вход в такой подкаталог приводит к попаданию в корень соответствующей файловой системы. Однако прежде чем сможет состояться работа с файлами на устройствах с мобильными носителями (CD-ROM, НГМД, флэш-память и т.п.), файловая система соответствующего устройства с установленным на нём мобильным носителем должна быть "вручную" подключена к дереву файлового пространства QNX. Процедура подключения называется монтирование файловой системы устройства.

Монтирование производится командой `<mount>`, где указывается тип файловой системы (из списка типов файловых систем, известных QNX: `dos`, `qnx4`, `cd` и т.д.), полное имя файла устройства (зарегистрированного в QNX) и полное имя формируемого каталога, ассоциируемого с монтируемой файловой системой. Например, для монтирования файловой системы типа `dos` на НГМД (ему соответствует в QNX файл устройства с именем - `/dev/fd0`) необходимо выполнить в окне терминала следующую команду shell:

```
# mount –t dos /dev/fd0 <полное имя каталога>
```

Если монтируемой файловой системе дать имя /home/A:, то команда монтирования примет вид

```
# mount –t dos /dev/fd0 /home/A:
```

В каталоге /home появляется подкаталог с именем </A:>, вход в который будет приводить к входу в файловую систему дискеты.

Если нужно монтировать дискету с файловой системой qnx4 (тип файловой системы QNX), то вместо dos нужно написать qnx4.

Файл устройства CD-ROM имеет имя - /dev/cd0. Команда для монтирования CD-ROM и создания каталога файловой системы устройства с именем cd0 в каталоге fs будет иметь вид:

```
# mount –t cd /dev/cd0 /fs/cd0
```

Если необходимость в мобильном носителе отпала, его можно демонтировать и соответствующий каталог в файловом пространстве будет аннулирован. Например, для демонтирования НГМД нужно выполнить команду

```
# umount /dev/fd0
```

или

```
# umount /home/A:
```

Типы файлов

QNX более тонко определяет понятие файла и его имени. Имя файла рассматривается как атрибут файловой системы, косвенно связанный с некоторым набором данных на диске, который не имеет имени как такового. Каждый такой набор данных (*файл*) имеет связанные с ним *метаданные* (хранящиеся в *индексных дескрипторах* - inode), содержащие все характеристики файла и позволяющие операционной системе управлять выполнением операций, заказанных прикладной задачей: открыть файл, прочитать или записать данные, создать или удалить файл. В частности, метаданные содержат указатели на дисковые блоки хранения данных файла.

Имя файла в файловой системе рассматривается как ссылка на его метаданные, в то время как метаданные не содержат сведений об имени файла.

В QNX существуют 6 типов файлов, различающихся по функциональному назначению и действиям операционной системы при выполнении тех или иных операций над файлами:

- Обычный файл
- Связь
- Каталог
- Именованный канал
- Сокет

- Файл устройства

Обычный файл

Обычный файл представляется операционной системой как файл, содержащий просто последовательность байтов. Интерпретация содержимого такого файла производится исключительно прикладной программой (приложением). К таким файлам относятся и файлы исполняемых программ.

Связь

Кроме имени файла для ссылки на его метаданные могут использоваться также так называемые связи. *Связь* в QNX является аналогом ярлыка файловой системы ОС Windows. Она позволяет поставить в соответствие одному файлу несколько различных имен (псевдонимов), размещая их затем в различных местах файловой системы.

Связь может быть организована с помощью так называемых жестких ссылок ("жесткие связи") и символических или мягких ссылок ("символические связи").

При создании для некоторого файла дополнительного имени с помощью *жесткой связи* в каталог помещается новый элемент, который ссылается на тот же файл. Для обеспечения механизма жестких связей операционная система использует специальный системный файл /.inodeas, в котором, в частности, ведется счетчик ссылок на файл. При создании очередной жесткой связи счетчик ссылок увеличивается на единицу. Жесткие связи могут быть помещены в различные каталоги, находящиеся на одном и том же физическом носителе (одном разделе жесткого диска). Создание еще одного имени файла (жесткой связи) осуществляется с помощью команды <ln>. Нельзя создавать жесткие связи для каталогов. При удалении одной из жестких сявзей реально будет удален только соответствующий элемент каталога, а счетчик ссылок на файл будет уменьшен на единицу. Как только счетчик достигнет нуля, то файл и соответствующие ему атрибуты управления будут физически уничтожены (при этом файл должен иметь статус "закрыт").

В отличие от жесткой связи *символическая связь* реализуется в виде системного текстового файла, содержащего имя указываемого файла. Отличительным свойством символической связи является то, что с её помощью можно создавать дополнительные имена для любого файла (в том числе каталога), находящегося, в общем случае, на другом физическом носителе (например, в другом разделе диска или на другом узле сети). Возможность создания символических связей для каталогов создает опасность бесконечных циклов. Поэтому число переходов по символическим связям ограничено значением системной переменной `SYMLOOP_MAX`, определенной в файле `<limits.h>`.

Каталог

Каталог - это системный файл, который отличается от обычного тем, что интерпретируется операционной системой как набор записей определенной структуры, которые называют элементами каталога. Каждый элемент каталога

связывает имя некоторого файла со служебной информацией о нем, включающей ссылку на место физического хранения данных. Любая задача, имеющая право на чтение каталога, может прочесть его содержимое, но только ОС имеет право на запись в каталог. Штатные средства просмотра содержимого файловой системы по умолчанию не показывают файлы (в том числе и каталоги), имена которых начинаются с точки ("."). Такие файлы называют "скрытыми", и обычно в них содержится системная информация.

Именованный канал

Именованный канал (FIFO) - этот тип файлов относят к средствам взаимодействия процессов, они используются для передачи данных между процессами.

Сокеты

Сокеты - этот тип файлов относят к средствам доступа к сети TCP/IP.

Устройства

Файл устройства обеспечивает доступ к физическому или виртуальному устройству, зарегистрированному в QNX. Для взаимодействия с устройствами кроме драйверов им ставятся в соответствие файлы устройств. Программы обмениваются данными с устройствами через файлы устройств. Устройства разделяют на два типа:

- *Символьные* (*байт-ориентированные*) устройства читают и записывают данные в виде последовательного потока байтов. Сюда входят последовательные и параллельные порты, накопители на магнитной ленте, терминалы и звуковые карты.
- *Блочные* (*блок-ориентированные*) устройства читают и записывают данные блоками фиксированного размера. Блочные устройства предоставляют прямой доступ к своим данным. Примером блочного устройства является накопитель на жестком или гибком диске.

Работа с файлами устройств осуществляется путем использования стандартных библиотечных функций ввода-вывода. Программы могут открывать файлы устройств, читать из них данные и осуществлять запись в них точно так же, как если бы это были обычные файлы.

Все файлы устройств, известных системе, содержатся в каталоге `/dev`. Имена этих файлов стандартизированы. Для доступа к символьному устройству достаточно открыть соответствующий файл устройства как обычный файл и осуществлять чтение/запись традиционным образом. Например, если к первому параллельному порту подключен принтер, то распечатать файл `document.txt` можно, направив его непосредственно на устройство `/dev/lp0`, используя команду копирования файлов <cat>:

```
# cat document.txt > /dev/lp0
```

Чтобы эта команда завершилась успешно, необходимо иметь право записи в файл принтера.

Послать устройству данные из программы также не сложно. В приведенном ниже фрагменте программы с помощью низкоуровневых функций ввода-вывода содержимое буфера направляется в устройство /dev/lp0:

```
int fd=open("/dev/lp0", O_WRONLY);
write(fd, buffer, buffer_length);
close(fd);
```

Виртуальные устройства

В QNX есть ряд специальных символьных устройств, которым не соответствуют никакие аппаратные компоненты. Эти устройства являются виртуальными.

Устройство /dev/null

Устройство /dev/null служит двум целям.

– Поглощает любые данные, направляемые в устройство. В тех случаях, когда выводные данные программы не нужны, в качестве выходного файла назначают устройство /dev/null, например:

```
# verbose_command > /dev/null
```

– При чтении из устройства /dev/null всегда возвращается признак конца строки (файла). Если открыть /dev/null с помощью функции open() и попытаться прочесть данные из него с помощью функции read(), то функция вернет 0 байтов. При копировании содержимого файла /dev/null будет создан пустой файл нулевой длины:

```
# cp /dev/null empty_file
# ls -l empty_file
-rw-rw----   1 ivanov  ivanov      0 Mar 12  15:27 empty_file
```

Устройство /dev/zero

Устройство /dev/zero ведет себя так, как если бы оно было файлом бесконечной длины, заполненным одними нулями. Сколько бы данных ни запрашивалось из этого файла, они всегда предоставляются в необходимом количестве.

Файл /dev/zero удобно использовать в функциях выделения памяти, которые отображают этот файл в память, чтобы инициализировать её нулями.

Устройство /dev/full

Устройство /dev/full ведет себя так, как если бы оно было файлом, в котором нет свободного места. Операция записи в этот файл завершается ошибкой, и в переменную errno помещается код, свидетельствующий о том, что устройство переполнено.

Этот файл удобен для проверки того, как программа будет вести себя в случае, если при записи в файл возникает нехватка места.

Устройства генерирования случайных чисел

Специальные устройства `/dev/random` и `/dev/urandom` предоставляют доступ к средствам генерирования случайных чисел, входящим в `QNX`. Эти устройства для получения случайных чисел используют внешний "источник хаоса". Замеряя задержки между действиями пользователя, в частности нажатиями клавиш и перемещениями мыши, устройства способны генерировать непредсказуемый поток действительно случайных чисел. Получить доступ к этому потоку можно путем чтения из устройств `/dev/random` и `/dev/urandom`.

Разница между устройствами заключается в том, что если попытаться прочесть большое количество случайных чисел из устройства `/dev/random` и при этом нем выполнять никаких пользовательских действий (не нажимать клавиши, не перемещать мышь и т.п.), то случайные числа заканчиваются и операция чтения блокируется. Только когда пользователь проявит какую-то активность, система сгенерирует новые случайные числа и разблокирует операцию чтения. В противоположность этому операция чтения из устройства `/dev/urandom` никогда не блокируется. Если в системе заканчиваются случайные числа, используется криптографический алгоритм, чтобы сгенерировать псевдослучайные числа из последней цепочки случайных чисел.

Пользователи и группы

Идентификация пользователей

QNX располагает средствами учета (идентификации) пользователей системы. Идентификация пользователя заключается в присвоении ему системного *имени* и *пароля*. Сразу после установки QNX система уже содержит имя root. Под этим именем система идентифицирует пользователя (*системного администратора*), которому предоставляются неограниченные полномочия по управлению ресурсами системы. В самом начале пароль для пользователя с именем root отсутствует (пустая строка). Вход пользователя в систему с именем root приводит к автоматической установке каталога с именем /root в качестве текущего.

Идентификация вновь добавляемых пользователей системы производится уже системным администратором с помощью команды passwd. Любой пользователь может в дальнейшем изменить свой пароль, выполнив команду passwd. Утилита запросит прежний пароль и дважды попросит ввести новый пароль. В отличие от системного администратора обычные пользователи имеют ограниченные (соответствующие им) права доступа к ресурсам системы.

Для учета пользователей система использует системные файлы /etc/passwd и /etc/shadow. Файл /etc/passwd состоит из строк следующего формата:

username:haspw:userid:group:comment:homedir:shell

- username - имя пользователя, используемое для входа в систему;
- haspw - если поле не пустое, то в файле /etc/shadow хранится пароль пользователя;
- userid - идентификатор пользователя (у root - 0);
- group - числовой идентификатор первичной группы (см. п.2.2.);
- comment - любая строка, не содержащая символа ":";
- homedir - домашний каталог пользователя, т.е. каталог, в котором пользователь может произвольно создавать и удалять файлы;
- shell - командный интерпретатор, который запускает утилита login при успешном входе пользователя в систему.

Файл /etc/shadow состоит из строк следующего формата:

username:passwd:lastch:minch:maxch:warn:inact:expire:reserved

- username - имя пользователя, используемое для входа в систему;
- passwd - зашифрованный пароль пользователя;

– lastch - время последней модификации;

– minch - минимальное количество дней для модификации;

– maxch - максимальное количество дней для модификации;

– warn - количество дней для предупреждения;

– inact - максимальное количество дней между входами в систему;

– expire - дата истечения доступа;

– reserved - зарезервировано для дальнейшего использования;

Создание и идентификация групп

Кроме идентификации и учета отдельных пользователей QNX располагает средствами учета групп пользователей (групп). Добавление и идентификация группы производится системным администратором. Добавление новых групп и их идентификация выполняется путем простого редактирования файла /etc/group, строки которого имеют формат:
groupname:reserved:group:member

– groupname - имя группы;

– reserved - зарезервировано для дальнейшего использования;

– group - числовой идентификатор группы;

– member - список имен пользователей, принадлежащих данной группе.

В список имен пользователей можно добавить имя любого пользователя (username), идентифицированного в файле /etc/passwd. Пользователь системы может быть членом нескольких групп, одна из которых назначается *первичной* (primary), остальные - *дополнительными* (supplementary). Первичной группой становится та группа, числовой идентификатор которой прописывается в поле group строки учета пользователя в файле /etc/passwd.

Как и пользователям, группам соответствуют определенные права доступа к ресурсам системы. Если пользователь является членом группы, то дополняет свои права доступа к ресурсам системы правами группы. Принцип формирования групп и включения в них пользователей определяется системным администратором.

Удаление пользователей и групп

Учетная информация о пользователях и группах хранится в файлах /etc/passwd, /etc/shadow, /etc/group. Для того чтобы удалить из системы пользователя, системный администратор должен отредактировать эти три файла, удалив или изменив строки, соответствующие удаляемому пользователю.

Для удаления группы достаточно отредактировать файл /etc/group, но необходимо обязательно убедиться, что нет пользователей, для которых эта

группа является первичной (эта группа не должна упоминаться в файле `/etc/passwd`).

Сеанс работы пользователя в системе

Для доступа пользователя к системе `QNX` выполняет процедуру аутентификации пользователя. Она заключается в том, что после включения компьютера и загрузки системы запускается утилита `login` или `phlogin`, которая запрашивает у пользователя имя и пароль. Если пользователь зарегистрирован в системе и ввел правильные имя и пароль, то `login` запускает командный интерпретатор, указанный в файле `/etc/passwd`, и пользователь входит в систему. Вход пользователя в систему специальным образом учитывается операционной системой - система создает так называемый *сеанс* работы пользователя. Сеанс логически объединяет все процессы, которые порождаются в результате входа и последующей работы пользователя в системе. Если пользователь входит в систему в режиме консоли командной строки, то на терминале появляется приглашение к вводу команд. Для системного администратора приглашение имеет вид "`#`", для обычного - "`$`". После этого пользователь может вводить командные строки.

По окончании работы пользователь завершает работу в системе. В режиме консоли командной строки работа завершается путем ввода команды "`exit`". При завершении работы пользователя в системе все процессы, принадлежащие сеансу пользователя, аннулируются.

Разграничение доступа к файлам

У каждого файла при его создании формируются атрибуты UID и GID, специфицирующие соответственно владельца-пользователя (пользователь, породивший процесс, который создал файл) и владельца-группу (группа процессов), наследуемых от процесса, создавшего файл. При открытии файла процессами эти атрибуты сравниваются с соответствующими атрибутами процессов для проверки их прав доступа к файлу. Процесс может относиться к личным владельцам файла, входить в группу, являющуюся владельцем файла, или быть отнесенным по отношению к файлу к "прочим" процессам. Для владельцев и для прочих у файла установлены соответствующие разрешения на выполнение операций с файлом (чтение, запись или исполнение). Существуют функции, позволяющие управлять владельцами файла (значением атрибутов UID и GID).

QNX регулирует возможность различных процессов выполнять операции чтения/записи с файлом. Для этого введено понятие владельца файла. Различают следующих пользователей файла.

Зарегистрированный в системе пользователь, который некоторым способом инициировал создание файла, назначается *пользователем-владельцем* этого файла. Кроме пользователя-владельца при создании файла с ним ещё ассоциируется некоторая зарегистрированная в системе группа пользователей, которая назначается как *группа-владелец*. Остальные зарегистрированные в

системе пользователи по отношению к созданному файлу рассматриваются как *прочие*. Одновременно при создании файла назначенным владельцам (пользователю, группе) и прочим устанавливаются права доступа к файлу. Любой зарегистрированный в системе пользователь может получить права доступа к файлу, владельцем которого он не является, если он становится членом группы-владельца этого файла. Включение пользователя в группу автоматически предоставляет ему по отношению файлу, которым владеет группа, установленные для группы права доступа. Наоборот - для лишения пользователя прав доступа к файлу, ассоциированных с группой, достаточно исключить его из состава этой группы.

Каждому созданному в QNX файлу устанавливаются параметры - *идентификатор владельца* (UID) и *идентификатор группы* (GID), а также атрибуты *прав доступа* для трех категорий пользователей: *владельца* (user owner), *группы* (group owner) и "*остальных*" (other). Отметим, что владелец может не являться членом группы, владеющей файлом.

Каждому запущенному в системе процессу в дескрипторе устанавливаются два параметра - так называемые *эффективный идентификатор владельца* (EUID) и *эффективный идентификатор группы* (EGID). Идентификаторы файла и процесса используются для проверки прав доступа процесса к файлу. Когда процесс пытается открыть какой-либо файл, QNX сначала проверяет, совпадает ли EUID процесса с UID файла. Если совпадает, то проверяется, имеет ли ассоциированный с процессом владелец право открыть файл с указанным режимом доступа. Если владелец имеет такие права, то файл открывается, если не имеет, то проверяется, совпадает ли EGID процесса с GID файла и право данной группы открывать файл с указанным режимом доступа. Если никакие идентификаторы не совпали, тогда данному процессу может быть разрешен режим доступа, установленный для "остальных". Для процессов с EUID=0 (т.е. для пользователя root) доступ к файлам предоставляется без процедуры проверки прав.

Права доступа к файлу

Операционная система QNX различает три базовых класса доступа к файлу:

- User access (u) - для владельца-пользователя файла.
- Group access (g) - для членов группы, являющейся владельцем файла.
- Other access (o) - для остальных пользователей (кроме суперпользователя, у которого максимальные права).

Для каждого из указанных классов доступа QNX поддерживает три типа прав доступа к файлу: на чтение (r), на запись (w) и на выполнение (x).

Права доступа могут быть изменены только владельцем файла или пользователем root посредством команды chmod.

Значение (семантика) прав доступа зависит от типа файла. Для обычного файла смысл операций вытекает из названий прав доступа. Например, если исполняемый файл является скриптом командного интерпретатора `shell`, то для его запуска понадобится право на чтение (`r`), поскольку при выполнении скрипта командный интерпретатор должен иметь возможность считывать команды из файла, а также право на выполнение (`x`). Права доступа для каталога не столь очевидны. Система трактует операции чтения и записи для каталогов отлично от остальных файлов. Право чтения каталога позволяет получить только имена файлов, находящихся в данном каталоге. Чтобы получить дополнительную информацию о файлах каталога, требуются права на "выполнение" каталога (`x`). Это же право нужно иметь для доступа ко всем каталогам на пути к указанному. Особое значение для каталога имеет право на запись. Создание и удаление файлов в каталоге требуют права на запись в этот каталог. Но при этом, чтобы удалить некоторый файл из каталога, не обязательно иметь какие-либо права доступа к этому файлу, важно лишь иметь право на запись для каталога, в котором находится этот файл.

Программный интерфейс QNX

Одной из целей, которые изначально ставились перед разработчиками QNX, являлось создание удобной среды программирования и программного интерфейса. Разработка программ не возможна без знания интерфейса системных вызовов и без понимания внутренних структур и функций, предоставляемых операционной системой. Осмысленное администрирование системы также затруднительно без представления о том, как работает QNX. Программный интерфейс QNX позволяет наглядно показать внутренние механизмы операционной системы.

Программный интерфейс выражается в виде системных вызовов и функций стандартных библиотек. Далее будут рассмотрены важнейшие, функции стандартной библиотеки ввода/вывода, системные вызовы работы с файлами и управления файловой системой, системные вызовы создания процесса и управления процессами и другие составляющие программного интерфейса.

Системные вызовы и функции стандартных библиотек

Прикладные задачи имеют возможность воспользоваться базовыми услугами, предоставляемыми QNX. Эти услуги получили название *системных вызовов*. Системный вызов инициирует функцию, выполняемую средствами операционной системы от имени процесса, выполнившего вызов, и является программным интерфейсом самого низкого уровня взаимодействия прикладных процессов с операционной системой. В среде программирования QNX они определяются как функции языка С. В QNX каждый системный вызов имеет соответствующую функцию (или семейство функций) с тем же именем, хранящуюся в стандартной библиотеке языка С (в дальнейшем эти функции будем для простоты называть системными вызовами). Фактически эти функции играют роль оболочки, которая выполняет необходимые преобразования аргументов и инициируют требуемый системный вызов ОС.

Помимо системных вызовов программный интерфейс предлагает большой набор функций общего назначения. Эти функции не являются системными вызовами, хотя в процессе выполнения многие из них выполняют системные вызовы. Эти функции также хранятся в стандартных библиотеках С и наряду с системными вызовами составляют основу среды программирования в QNX. К этим функциям относятся функции библиотеки ввода/вывода, функции распределения памяти, функции управления процессами и т.д.

Обработка ошибок

Разница между системными вызовами и библиотечными функциями проявляется еще в способе передачи процессу информации об ошибке, произошедшей во время выполнения системного вызова или функции библиотеки.

Обычно в случае возникновения ошибки системные вызовы возвращают целое значение `-1` и устанавливают значение глобальной системной переменной

errno, указывающее причину возникновения ошибки. Системный заголовочный файл <errno.h> содержит коды ошибок, значения которых может принимать переменная errno, с краткими комментариями.

Библиотечные функции, как правило, не устанавливают значение переменной errno, а код возврата различен для разных функций. Для уточнения возвращаемого значения библиотечной функции необходимо обратиться к описанию этих функций в справочной системе QNX.

Рассмотрим более подробно обработку ошибок, возникающих при выполнении системных вызовов с использованием переменной errno. Заметим, что значение errno не обнуляется следующим нормально завершившимся системным вызовом. Следовательно, значение errno имеет смысл только после вызова, который завершился с ошибкой.

Стандарт ANSI C определяет две функции, помогающие сообщить причину ошибочной ситуации:

```
#include <string.h>
char *strerror(int errnum);
```

и

```
#include <errno.h>
#include <stdio.h>
void perror(const char *s).
```

Функция strerror() принимает в качестве аргумента errnum номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации. Функция perror() выводит в стандартный поток сообщений об ошибках (обычно на экран) содержимое строки s и вслед за ним – системную информацию об ошибочной ситуации, основываясь на значении переменной errno.

Функции управления файловой системой

Создание каталога

Новый каталог можно создать с помощью вызова:

```
#include <sys/types.h>
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
```

Функция `mkdir()` создает новый пустой каталог, специфицированный в `path` с разрешениями доступа, заданными в `mode` в виде комбинации флагов разрешения, определенных в заголовочном файле `<sys/stat.h>`. ID владельца каталога устанавливается равным эффективному ID пользователя процесса. ID группы каталога устанавливается равным ID группы родительского каталога (если установлен флаг использования ID группы родительского каталога) или эффективный ID группы процесса.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается `errno`.

Пример:

```
Создается новый каталог с именем /src in /hd:

#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

int main( void )
{
    mkdir( "/hd/src",
           S_IRWXU |
           S_IRGRP | S_IXGRP |
           S_IROTH | S_IXOTH );

    return EXIT_SUCCESS;
}
```

Удаление каталога

Для удаления каталога используется вызов:

```
#include <sys/types.h>
#include <unistd.h>

int rmdir(const char* path);
```

Функция `rmdir()` удаляет каталог, специфицированный в `path`, если его счетчик связей равен 0 и он не открыт ни каким процессом. Каталог должен быть пустым.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается `errno`.

Пример:

```
    /*Удаляет каталог с именем /home/terry*/
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( void ){
    rmdir( "/home/terry" );

    return EXIT_SUCCESS;
}
```

Смена текущего каталога

Процесс может изменить текущий каталог с помощью системного вызова:

```
#include<unistd.h>
int chdir(const char *path);
```

Функция `chdir()` изменяет текущий рабочий каталог на `path`, который может быть относительным или абсолютным именем. Так как с процессом связывается только один текущий каталог, то в многонитиевых приложениях любая нить, вызвавшая `chdir()`, изменит текущий каталог для всех нитей в этом процессе.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается `errno`.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main( int argc, char* argv[] )
{
    if( argc != 2 ) {
        fprintf( stderr, "Use: cd <directory>\n" );
        return EXIT_FAILURE;
    }

    if( chdir( argv[1] ) == 0 ) {
        printf( "Directory changed to %s\n", argv[1] );
        return EXIT_SUCCESS;
    } else {
        perror( argv[1] );
        return EXIT_FAILURE;
    }
}
```

Смена корневого каталога

Процесс может изменить свой корневой каталог с помощью системного вызова:

```
#include<unistd.h>
int chroot(const char *path);
```

Функция chroot() делает каталог path корневым каталогом. После этого поиск файлов с абсолютными именами, начинающимися с '/', будет производиться, начиная с каталога, указанного аргументом path. Заметим, однако, что пользовательский текущий каталог сохраняется!

Для изменения корневого каталога значение эффективного пользовательского ID процесса (EUID) должно соответствовать системному администратору. Системная жесткая ссылка "..", входящая в корневой каталог, указывает на него самого. В связи с этим жесткая ссылка ".." не может быть использована для доступа к файлам за пределами поддерева, входящего в корневой каталог.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Создание жесткой связи

Создать связь к существующему файлу можно с помощью вызова:

```
#include <unistd.h>
int link(const char* pname, const char* new);
```

Функция link() создает новый элемент каталога с именем new (путь доступа для новой связи), являющийся жесткой ссылкой на существующий файл с именем pname (путь доступа к существующему файлу), и увеличивает счетчик связей для указанного файла на 1. При этом файл не может быть каталогом или находится на другом устройстве.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Создание символической связи

Создать символическую связь с файлом можно с помощью вызова:

```
#include <unistd.h>
int symlink(const char* pname, const char* slink);
```

Функция symlink() создает символическую связь с именем slink, которая содержит абсолютное имя файла pname (slink является именем создаваемой символической связи, pname есть абсолютное имя, содержащееся в символической связи).

Контроль прав доступа к файлу pname не выполняется и отсутствует необходимость существования файла. Символическую связь можно создать к файлу и каталогу даже на другом устройстве.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
  /* Создание символической связи для "/usr/nto/include" в
      текущем каталоге */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>

int main( void )
  {
    if( symlink( "/usr/nto/include", "slink" ) == -1) {
      perror( "slink -> /usr/nto/include" );
      exit( EXIT_FAILURE );
    }
    exit( EXIT_SUCCESS );
  }
```

Чтение символической связи

Содержимое символической связи можно прочитать и поместить в буфер с помощью вызова:

```
#include <unistd.h>
int readlink(const char* path, char* buf, size_t bufsiz);
```

Функция `readlink()` помещает содержание символической связи с именем `path` в буфер `buf`. Если `readlink()` завершается успешно, то `bufsiz` байтов содержания символической связи помещается в `buf`. Возвращаемый набор символов размером `bufsiz` не является строкой (не имеет в конце нуля).

При успешном завершении функция возвращает количество байтов, помещенных в `buf`. В случае ошибки возвращается -1 и устанавливается `errno`.

Пример:

```
/* В качестве аргумента программа принимает имя символической связи
*/
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

char buf[PATH_MAX + 1];

int main( int argc, char** argv ){
  int nread, fd;
/* Чтение содержимого символической связи */

if(( nread = readlink( argv[1], buf, PATH_MAX )) == -1) {
      perror( argv[1] );
      exit(EXIT_FAILURE);
}
buf[nread] = '\0';
printf( "Символическая связь %s -> %s\n", argv[1], buf );
exit( EXIT_SUCCESS );
}
```

Открытие файла

Для открытия или создания файла используется функция:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag [,mode_t mode]);
```

Функции `open()` позволяют процессу открыть файл (устройство), имя которого указано в `path`. Это имя может быть как абсолютным (полным, начинающимся с корневого каталога /), так и относительным (указанным относительно текущего каталога). Для открытого файла создается новый дескриптор, который не разделяется с каким-либо другим процессом в системе. *Режим доступа* к открытому файлу устанавливается согласно значению флагов, сформированных в разрядах аргумента `oflag`.

Значение `oflag` является целым значением и строится с использованием операции поразрядного "ИЛИ" над значениями флагов, символические константы для которых определены в заголовочном файле `<fcntl.h>`. Флаги позволяют определить режим открытия и доступа к существующему или вновь создаваемому файлу, а также уточнить реализацию выбранного режима доступа. Значения и семантика флагов следующая:

`O_RDONLY` - открыть существующий файл только для чтения;

`O_RDWR` - открыть для чтения и записи;

`O_WRONLY` - открыть только для записи.

Выбранный режим доступа к открываемому файлу уточняется флагами:

`O_APPEND` - если этот флаг установлен, то для режимов, допускающих запись в файл (`O_RDWR` и `O_WRONLY`), перед каждой операцией записи указатель файла будет устанавливаться в конец этого файла.

`O_DSYNC` - если этот флаг установлен, то каждый запрос `write()` будет ждать, пока все данные не будут успешно записаны. То есть все записи в файл, а также соответствующие им изменения в метаданных файла будут сохранены на внешнем носителе.

Ядро кэширует данные, считываемые или записываемые на внешний носитель, для ускорения этих операций. Обычно запись данных в файл ограничивается только записью в буферный кэш ядра операционной системы, данные из которого впоследствии записываются на внешний носитель. По умолчанию возврат из функции `write()` происходит после записи данных в буферный кэш, не дожидаясь записи на внешний носитель. Установка флага `O_DSYNC` гарантирует, что в результате завершения `write()` даже при фатальных нарушениях в системе (но исправности внешнего носителя) данные будут сохранены в файле и могут быть прочитаны при последующем открытии файла. Если носитель оборудован защитой от записи, то это интерпретируется,

как поломка носителя, данные не будут записаны, даже если write() указывает, что все успешно.

O_RSYNC - если этот флаг установлен, то каждый запрос read() завершается только тогда, когда данные успешно прочитаны.

O_SYNC - если этот флаг установлен, то каждый запрос read() или write() завершается только тогда, когда данные успешно прочитаны или записаны. Все записи в файл, а также соответствующие им изменения в метаданных файла будут сохранены на внешнем носителе.

O_CLOEXEC - дескриптор файла не будет наследоватся вновь создаваемыми процессами (будет закрыт при запуске процесса).

O_CREAT - установка этого флага указывает, что если открываемый файл с указанным именем не существует, то необходимо создать новый файл и открыть его для записи. Если файл с указанным именем уже существует, то будет ли открыт этот файл, или создан новый файл для записи, или функция open() завершится с ошибкой определяется значением флага O_EXCL.

O_EXCL - флаг используется совместно с O_CREAT. При его установке новый файл будет создан только в том случае, если файл с указанным именем не существует и это же действие с тем же именем файла в данный момент не выполняется другими процессам. В противном случае возвращается ошибка создания файла. Если флаг не установлен, то при наличии файла с указанным именем флаг O_CREAT игнорируется и открывается существующий файл. Флаг O_EXCL без O_CREAT также игнорируется.

O_LARGEFILE - разрешает, чтобы указатель файла был длиной в 64 бита (при работе с огромными файлами).

O_NOCTTY - если этот флаг установлен и указанный файл является терминальным устройством, то функция open() не делает терминальное устройство управляющим терминалом для процесса.

O_NONBLOCK - если этот флаг установлен, то функция open() завершается без ожидания, когда устройство будет готовым или доступным. Последующее поведение устройства определяется его спецификой. Если флаг очищен, то функция open() ждет, прежде чем завершиться, когда устройство будет готовым или доступным. Готовность устройства определяется его спецификой. В некоторых случаях реакция на флаг не определена.

O_REALIDS - требует использовать реальные UID и GID процесса для проверки разрешенных прав доступа к файлу.

O_TRUNC - если файл существует, является обычным файлом и он успешно открыт в режиме O_WRONLY или O_RDWR, то длина файла усекается до нуля, а права доступа и владелец оставлены неизменными. Флаг не имеет никакого эффекта для канала FIFO, файлов устройств или каталогов. Если файл открывается как O_RDONLY, флаг O_TRUNC игнорируется.

Аргумент mode имеет значение только когда создается новый файл и позволяет задать для файла *права доступа* пользователей.

Значение mode является целым значением и строится с использованием операции поразрядного "ИЛИ" над значениями флагов, специфицирующих права доступа к вновь создаваемому файлу.

Устанавливаемые права доступа и дополнительные атрибуты файла (SUID, SGID и Sticky bit) определяются следующими символическими значениями флагов (определены в заголовочном файле <sys/stat.h>):

Флаг	Значение
S_ISUID	Установить для файла бит атрибута SUID
S_ISGID	Установить для файла бит атрибута SGID или установить обязательное блокирование файла (определяется в зависимости от значений других флагов)
S_ISVTX	Установить Sticky bit (для каталогов)
S_IRWXU	Установить право на чтение, запись и выполнение для владельца
S_IRUSR	Установить право на чтение для владельца
S_IWUSR	Установить право на запись для владельца
S_IXUSR	Установить право на выполнение для владельца
S_IRWXG	Установить право на чтение, запись и выполнение для группы
S_IRGRP	Установить право на чтение для группы
S_IWGRP	Установить право на запись для группы
S_IXGRP	Установить право на выполнение для группы
S_IRWXO	Установить право на чтение, запись и выполнение для остальных
S_IROTH	Установить право на чтение для остальных
S_IWOTH	Установить право на запись для остальных
S_IXOTH	Установить право на выполнение для остальных

Атрибуты SUID и SGID имеют смысл только при создании исполняемых файлов. Назначение атрибутов заключается в следующем. При запуске дочернего процесса (на основе исполняемого файла) его идентификаторы UID, GID, EUID, EGID устанавливаются равными соответствующим идентификаторам родительского процесса. Если же для исполняемого файла установлен атрибут SUID, то идентификаторы дочернего процесса будут установлены равными идентификатору владельца исполняемого файла. Атрибут

SGID исполняемого файла делает то же с групповым идентификатором процесса. То есть, если существует, например, файл утилиты myprog, владельцем которого является пользователь с именем "user1", с правами доступа на выполнение файла для всех пользователей, то процесс, созданный при запуске утилиты myprog пользователем "user2", будет иметь UID и EUID унаследованные не от "user2", как следовало бы ожидать, а равные значениям UID и EUID владельца файла утилиты myprog - "user1". Не трудно догадаться, что установка для исполняемого файла атрибутов SUID и SGID не безобидна с точки зрения безопасности информации. Но иногда без них нельзя обойтись, например – для файла утилиты passwd, позволяющей пользователю изменить свой пароль. Изменение пароля требует изменения содержимого системных файлов /etc/passwd и /etc/shadow. Понятно, что предоставление права на запись в эти файлы всем пользователям системы является отнюдь не лучшим решением. С другой стороны необходимо, чтобы процесс, запущенный на основе исполняемого файла утилиты passwd, был согласован по правам доступа с файлами /etc/passwd и /etc/shadow. Установка атрибута SUID исполняемому файлу утилиты /usr/bin/passwd позволяет изящно разрешить это противоречие. Поскольку владельцем файла утилиты /usr/bin/passwd является пользователь root (системный администратор), то кто бы ни запустил утилиту passwd на выполнение, соответствующий процесс приобретает права системного администратора, т.е. может производить запись в системные файлы, защищенные от остальных пользователей. Понятно, что требование по безопасности для программ, подобных passwd, должны быть повышены за счет ограничения их функциональных возможностей. Они не должны выполнять никаких операций, способствующих наследованию прав доступа другими процессами (например, вызов других программ).

Значение флага S_ISGID зависит от того, установлено или нет право на выполнение для группы - S_IXGRP. В первом случае он будет означать установку SGID, а во втором - обязательное блокирование файла.

Блокирование файлов позволяет устранить возможность конфликта, когда более одного процесса одновременно работают с одним и тем же файлом. Файловая система разрешает нескольким процессам одновременный доступ к файлу для чтения и записи. Хотя операции записи и чтения, осуществляемые с помощью системных вызовов read() и write(), являются атомарными, по умолчанию отсутствует синхронизация между отдельными вызовами. Другими словами, между двумя последовательными вызовами read() одного процесса другой процесс может модифицировать данные файла. Это, в частности, может привести к несогласованным операциям с файлом, и как следствие, к нарушению целостности его данных.

Если создается новый файл, то идентификатор владельца файла устанавливается равным эффективному идентификатору владельца процесса, выполняющего функцию open() (UID=EUID), соответственно идентификатор группы устанавливается равным эффективному идентификатору группы

процесса (GID=EGID) или идентификатору группы родительского каталога (в котором создается файл), если для родительского каталога установлен флаг SGID.

Sticky bit (S_ISVTX) имеет смысл только для каталогов и влияет на возможность пользователей переименовывать или удалять файлы в таком каталоге, если каталог перезаписываемый. Для этого должны выполниться одно или большее количество следующих условий:

- EUID процесса равен UID файла;
- EUID процесса равен UID каталога, содержащего файл;
- процесс имеет право на перезапись файла;
- пользователь имеет права администратора root (UID=0).

В результате выполнения функция open() возвращает неотрицательное целое число (*дескриптор файла*), представляющее наименьшее значение неиспользуемого дескриптора файла. Для файла с прямым доступом указатель файла установлен в начало. В случае ошибки, возвращается -1 и устанавливается errno.

Замечание. При запуске программы для неё автоматически создаются три дескриптора: стандартный ввод – 0, стандартный вывод – 1, стандартный вывод сообщений об ошибках – 2.

Частным случаем функции open() является функция:

```
#include <fcntl.h>
int creat(const char *pathname, mode_t mode]);
```

Эта функция создает и открывает новый файл с именем, указанным в pathname, и правами доступа, заданными в mode. Она эквивалентна вызову:

```
open("myfile.dat",O_WRONLY|O_CREAT|O_TRUNC,mode);
```

При успешном выполнении функция creat() возвращает дескриптор файла, в случае ошибки возвращается -1 и устанавливает errno.

Пример:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main( void ) {
int fd;

/* создать и/или открыть файл для записи*/
/* существующий файл использовать снова как пустой*/
/* создать с правами чтения/записи для владельца */

fd=open("myfile.dat",O_WRONLY|O_CREAT|O_TRUNC,S_IRUSR|S_IWUSR);
```

```
…
/* открыть существующий файл для чтения */

fd = open("myfile.dat", O_RDONLY);
…
/* открыть существующий файл для дозаписи или создать новый файл
    для записи, если такой файл не существует, с правами доступа
    на чтение/запись для всех*/

fd = open("myfile.dat",
          O_WRONLY|O_CREAT|O_APPEND,
          S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
return EXIT_SUCCESS;
}
```

Дублирование дескриптора файла

Можно создать копию дескриптора открытого файла такую, что новый дескриптор, будет иметь следующие общие свойства с исходным дескриптором файла:

- тот же открытый файл или устройство;
- тот же указатель файла (это означает, что изменение одного указателя файла влечет за собой и изменение другого);
- тот же режим доступа (чтение, запись, чтение/запись).

Для получения копии используются функции:

```
#include<unistd.h>
int dup(int fd);
int dup2(int fd, int fd2);
```

При успешном выполнении `dup()` возвращает новый дескриптор файла - целое неотрицательное число. При успешном выполнении функция `dup2()` делает дескриптор `fd2` копией дескриптора `fd` и возвращает 0. Если уже имеется открытый файл с дескриптором `fd2`, то он предварительно закрывается. В случае ошибки обе функции возвращают -1 и устанавливается `errno`.

Пример:

```
#include <sys\stat.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define STDOUT  1 //Значение дескриптора стандартного вывода

int main(void)
{
    int fptr, oldstdout;
    char msg[] = "Это тест";
```

```
    /* создать файл */
    fptr = open("DUMMY.FIL", O_CREAT | O_RDWR, S_IREAD | S_IWRITE);

    if (fptr) {
      /* создать дубликат дескриптора стандартного вывода */
        oldstdout = dup(STDOUT);

     /* перенаправить стандартный вывод в DUMMY.FIL путем
дублирования дескриптора файла в дескриптор стандартного вывода */

     dup2(fptr, STDOUT);

     /* закрыть дескриптор файла DUMMY.FIL */
     close(fptr);

     /* это будет перенаправлено в DUMMY.FIL */
     write(STDOUT, msg, strlen(msg));

     /* восстановить исходное состояние дескриптора стандартного
вывода */
     dup2(oldstdout, STDOUT);

     /* закрыть дубликат дескриптора для STDOUT */
     close(oldstdout);
    }
    else
        printf("Ошибка при открытии файла!\n");
    return 0;
}
```

Переименование файла

Для переименования файла или связи используют вызов:

```
#include <stdio.h>
int rename(const char* old, const char* new);
```

Функция `rename()` меняет имя файла, специфицированного в `old`, на имя, специфицированное в `new`. Если файл (или пустой каталог) с именем `new` существует, он заменяется.

При успешном завершении функция возвращает 0. В случае ошибки возвращается значение отличное от 0 и устанавливается `errno`.

Пример:

```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    if( rename( "old.dat", "new.dat" ) ) {
        puts( "Ошибка переименования old.dat в new.dat." );

        return EXIT_FAILURE;
```

```
    }

    return EXIT_SUCCESS;
}
```

Удаление файла

Удалить файл (или связь) можно с помощью вызова:

```
#include <unistd.h>
int unlink( const char * path );
```

Функция `unlink()` удаляет файл (связь), чье имя указано в `path`. Если указана жесткая связь, то она удаляется, а счетчик связей соответствующего файла уменьшается на 1. Если указан файл или символическая связь, то реальное удаление объекта произойдет в тот момент, когда счетчик связей окажется равным 0. До этого момента реальное удаление откладывается.
Эта функция эквивалентна функции `remove()`.

Если каталог, содержащий файл, перезаписываем и у каталога установлен бит `S_ISVTX`, то процесс может удалять или переименовывать файлы внутри такого каталога только, если выполняется одно или большее количество следующих условий:

−EUID процесса равен UID файла;

−EUID процесса равен UID каталога, содержащего файл;

−процесс имеет право на перезапись файла;

−пользователь является системным администратором (UID = 0).

При успешном завершении функция возвращает 0. В случае ошибки возвращается значение отличное от 0 и устанавливается `errno`.

Пример:

```
#include <unistd.h>
#include <stdlib.h>

int main( void )
{
    if( unlink( "vm.tmp" ) ) {
        puts( "Error removing vm.tmp!" );

        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS;
}
```

Управление владельцами и правами доступа к файлам

Владение файлом определяет не только возможность доступа к файлу, но и тот набор операций (прав доступа), который пользователь может совершить с файлом: чтение, запись, запуск на выполнение (для исполняемых файлов).

Изменение прав доступа может осуществлять только владелец-пользователь, а также пользователь root.

Управление владельцами

Изменение для файла пользователя-владельца производится командой chown, а изменение для файла группы-владельца выполняется командой - chgrp. Эти команды может выполнить только пользователь root.

Одного собственника можно заменить другим с помощью функций:

```
#include <sys/types.h>
#include <unistd.h>

int chown(const char * path, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group );
```

Функция chown() или fchown() заменяет у файла, специфицированного именем, указанным в path, или дескриптором fd, текущие значения владельца-пользователя – UID, и владельца-группы – GID, на значения, содержащиеся в owner и group соответственно. Если файл является символической связью, chown() изменяет владельцев непосредственно у файла или каталога, на который осуществляется ссылка.

Для изменения соответствующих атрибутов непосредственно символической ссылки следует использовать функцию:

```
int lchown(const char * path, uid_t owner, gid_t group);
```

Заметим, что только процесс с эффективным ID пользователя (EUID), равным пользовательскому ID файла (UID), или с привилегиями системного администратора (пользователь root с UID=0) может изменить непосредственно атрибуты файла.

Замечание. В QNX при формировании прав доступа создаваемого файла может быть установлен флаг _POSIX_CHOWN_RESTRICTED (его наличие проверяется путем тестирования флага _POSIX_CHOWN_RESTRICTED с помощью функции pathconf()). Наличие этого флага означает, что только системный администратор может изменить владельцев файла. Обычным владельцам это окажется не доступным.

Если аргумент path ссылается на обычный файл, то при успешном выполнении функции атрибуты файла S_ISUID и S_ISGID очищаются.

При успешном завершении функция возвращает 0. В случае ошибки возвращается -1 и устанавливается errno.

Пример:

```
/*
 * Замена собственников файлов, заданных в списке, на
 * текущие значения собственников процесса - getuid(), getgid()
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int main( int argc, char** argv )
  {
    int i;
    int ecode = 0;

    for( i = 1; i < argc; i++ ) {
      if( chown( argv[i], getuid(), getgid() ) == -1 ) {
        perror( argv[i] );
        ecode++;
      }
    }
    exit( ecode );
  }
```

Управление правами доступа

Изменить права доступа к файлу можно с помощью функций:

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char * path, mode_t mode);

int fchmod(int fd, mode_t mode);
```

Функция `chmod()` или `fchmod()` изменяет для файла, специфицированного именем, указанным в `path`, или дескриптором файла в `fd`, флаги `S_ISUID`, `S_ISGID`, `S_ISVTX` и флаги разрешений доступа на соответствующие флаги, заданные в аргументе `mode`.

Эффективный `ID` пользователя у процесса должен соответствовать владельцу файла, или процесс, должен иметь необходимые привилегии, чтобы делать это.

Если каталог перезаписываем, и для каталога установлен бит `S_ISVTX`, то процесс может удалять или переименовывать файлы внутри такого каталога только, если выполняется одно или большее количество следующих условий:

- EUID процесса равен UID файла;
- EUID процесса равен UID каталога, содержащего файл;
- процесс имеет право на перезапись файла;
- пользователь является системным администратором (UID = 0).

Если процесс не имеет соответствующих привилегий и ID группы файла не соответствует эффективному ID группы процесса, а файл является обычным файлом, то бит `S_ISGID` в атрибутах файла очищается при успешном выполнении `chmod()`.

Если эффективный ID пользователя процесса равен ID владельца файла, или процесс имеет соответствующие привилегии (его владельцем является системный администратор), то `chmod()` устанавливает для файла биты `S_ISUID`, `S_ISGID`.

Вызов `chmod()` не имеет никакого эффекта для уже открытых файлов. В этом случае требуется вызов `fchmod()`.

При успешном завершении функции возвращают 0. В случае ошибки возвращается -1 и устанавливается `errno`.

Пример 1:

```
/*
Изменяет права доступа к файлам,
разрешая чтение/запись только личному владельцу
*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

int main( int argc, char **argv ){
  int i;
  int ecode = 0;

    for( i = 1; i < argc; i++ ) {
        if( chmod( argv[i], S_IRUSR | S_IWUSR ) == -1 ) {
            perror( argv[i] );
            ecode++;
        }
    }
    return ecode;
}
```

Пример 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>

void main( int argc, char **argv ){
  int fd;

/* Создание файла с правами r w x */
    fd = creat("my_file", S_IRUSR | S_IWUSR | S_IXUSR);

/* Добавим флаг SUID */
    fchmod(fd, S_IRWXU | S_ISUID );

/* Добавим флаг SGID */
    fchmod(fd, S_IRWXU | S_ISUID | S_ISGID );

/* Добавим флаг блокирования записей файла SGRP */
    fchmod(fd, S_IRWXU | S_ISUID | S_ISGID | S_ISGRP);
}
```

Стандартные функции работы с файлами

В среде программирования QNX существуют два основных интерфейса для работы с файлами:

1. Интерфейс системных вызовов, предлагающий системные функции низкого уровня, непосредственно взаимодействующие со средствами операционной системы.
2. Стандартная библиотека ввода/вывода, предлагающая функции буферизированного ввода/вывода.

Второй интерфейс является надстройкой над интерфейсом системных вызовов и предлагает более удобный (упрощенный) способ работы с файлами. Функции этого интерфейса определены стандартом ANSI языка С как стандартная библиотека ввода/вывода. Поэтому использование этих функций обеспечивает программе наибольшую мобильность. В то же время они не обеспечивают всех возможностей по управлению вводом/выводом, предоставляемых операционной системой QNX, которые могут потребоваться при создании приложений реального времени. Выбор между функциями интерфейса системных вызовов и стандартной библиотеки зависит от необходимой степени контроля ввода/вывода и требованием переносимости программы.

Стандартные функции ввода/вывода обеспечивают так называемый *буферизированный ввод/вывод*. Для работы с файлом в системе создается указатель на специальную структуру данных (системного типа `FILE`), называемую *потоком*. При создании потока в оперативной памяти автоматически создаются специальные буферы, участвующие в процессе ввода/вывода данных и позволяющие минимизировать число системных вызовов ввода/вывода и, следовательно, обращений к физическому устройству. Структура типа `FILE` содержит в себе:

- указатель на буфер;
- указатель позиции файла, подлежащий чтению или записи;
- число занятых байтов в буфере;
- флаги состояния потока.

Открытие файла

Для открытия файла используется функция

```
#include<stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Функция `fopen` открывает/создаёт файл с именем `filename`, связывает его с потоком и возвращает указатель на поток.
Строка `mode` задает режим доступа к файлу и может принимать одно из следующих значений:

`r` - Открыть только для чтения.

`w` - Создать новый файл для записи. Если файл с таким именем уже существует, то он будет переписан.

`a` - Открыть существующий файл для записи только в конец файла или создать файл для записи, если файл не существует.

`r+` - Открыть существующий файл для изменения (перезапись и чтение).

`w+` - Создать новый файл для записи и чтения. Если файл с таким именем уже существует, то он будет переписан.

`a+` - Открыть (или создать, если файл не существует) файл для чтения и записи только в конец файла.

Файл можно открыть в текстовом или двоичном режиме. Если содержимое файла текстовое и работа с файлом будет осуществляться исключительно как с текстом, то можно указать текстовый режим открытия файла. Для этого нужно добавить символ `t` к строке `mode` (`rt`, `w+t`, и т.д.). При открытии файла в текстовом режиме операционная система учитывает это таким образом, что может осуществлять фильтрацию некоторых управляющих символов при чтении данных из файла или включать определенные управляющие символы в файл при записи. Поэтому работа с содержимым файлов, открытых в текстовом режиме, должна осуществляться с использованием специальных функций чтения/записи символов или строк.

Если содержимое файла трактуется как набор байтов, то необходимо задать двоичный режим открытия. Для задания двоичного режима надо добавить символ `b` к строке `mode` (`wb`, `a+b`, и т.д.).

Можно помещать символы `t` и `b` между буквой и символом `+`. Например, использование `rt+` эквивалентно `r+t`.

Если `t` и `b` не указаны в строке `mode` - режим определяется значением глобальной переменной `_fmode`. Если `_fmode` установлена в `O_BINARY`, то файлы открываются в двоичном режиме. Если `_fmode` установлена в `O_TEXT` - файлы открываются в текстовом режиме. Эти константы определены в файле `fcntl.h`.

Когда файл открывается для изменения, по отношению к нему могут быть произведены как ввод, так и вывод данных. Однако следует знать, что вывод не может сразу следовать за вводом без использования функции `fseek()` или `rewind()` (см. ниже), как и ввод не может сразу следовать за выводом без вмешательства функций `fseek()` или `rewind()`, или за вводом, который встречает конец файла.

При успешном завершении функция `fopen` возвращает значение указателя на вновь открытый поток. В случае ошибки - возвращает `NULL`.

При запуске программного модуля на выполнение (процесса) автоматически становятся доступными стандартные потоки ввода/вывода, определённые следующим образом:

```
extern FILE *stdin;  //стандартный поток ввода
extern FILE *stdout; //стандартный поток вывода
```

```
extern FILE *stderr; /*стандартный поток системных сообщений об
                       ошибках*/
```

Открытие временного файла

Если файл необходим для временного хранения данных в период выполнения программы, то можно открыть временный файл без явного указания его имени. Для этого используется функция

```
#include <stdio.h>

FILE *tmpfile(void);
```

Функция `tmpfile()` создает и открывает для модификации временный двоичный файл в режиме `"w+b"`. Этот файл автоматически удаляется при его закрытии или завершении работы программы.

Функция `tmpfile()` возвращает указатель на поток созданного временного файла. Если файл не может быть создан, то `tmpfile()` возвращает `NULL`.

Пример:

```
#include <stdio.h>
#include <process.h>

int main(void)
{
   FILE *tempfp;

   tempfp = tmpfile();
   if (tempfp)
      printf("Временный файл создан\n");
   else
   {
      printf("Невозможно открыть временный файл\n");
      exit(1);
   }

   return 0;
```

Переназначение потоков

Дескриптор потока открытого файла можно переназначить (связать) с другим файлом. Это позволяет перенаправить поток ввода/вывода с одного файла на другой. Для этого используется функция:

```
#include<stdio.h>

FILE *freopen(const char *filename,
              const char *mode,
              FILE *stream);
```

Функция `freopen()` закрывает текущий файл, открывает файл, с именем, указанным в `filename` и связывает дескриптор потока `stream` с вновь открытым файлом. Эта функция закрывает исходный файл не зависимо от того, успешно ли состоялось открытие нового файла или нет. Строка `mode`,

определяет режим открытия и доступа к новому файлу так же, как и в функции fopen().

При успешном завершении freopen() возвращает значение аргумента stream. В случае ошибки - возвращает NULL.

Функцию freopen() удобно использовать для переключения стандартных потоков stdin, stdout или stderr с файлов устройств на обычные файлы.

Пример:

```
#include <stdio.h>

int main(void)
{
   /* перенаправить стандартный вывод в файл */
   if (freopen("OUTPUT.FIL", "w", stdout) == NULL)
      fprintf(stderr, "ошибка переадресации stdout\n");

   /* этот вывод будет осуществляться в файл */
   printf("Это будет выведено в файл.");

   /* закрыть стандартный поток вывода */
   fclose(stdout);

   return 0;
}
```

Доступ к файлу в текстовом режиме

Чтение символа из потока

Для чтения символа из потока используется семейство функций: fgetc(), fgetchar(), getc(), getch(), getchar(), getche(). Рассмотрим некоторые из них.

Функция

```
#include<stdio.h>
int fgetc (FILE *stream);
```

получает текущий символ из входного потока stream, перемещает указатель потока на следующий символ, возвращает код символа в виде положительного целого значения типа int или системную константу EOF, если обнаруживается конец файла или возникает ошибка ввода.

Для получения символа из стандартного потока stdin можно использовать функцию

```
int fgetchar(void);
```

Для ввода символа с клавиатуры без эхо-отображения на экране используется функция

```
#include<conio.h>
int getch(void);
```

Запись символа в поток

Для записи символа в поток используется семейство функций: `fputc()`, `fputch()`, `fputchar()`, `putch()`, `putchar()`. Рассмотрим некоторые из них.

Функция

```
#include<stdio.h>
int fputc (int ch, FILE * stream);
```

выводит символ `ch` в поток `stream`, перемещает указатель потока на следующую позицию и возвращает код этого символа в виде положительного целого значения типа `int` или системную константу `EOF`, если обнаруживается конец файла или возникает ошибка вывода.

Функция

```
int fputchar(int ch);
```

выводит символ `ch` в поток `stdout`. При успешном выполнении функция возвращает значение символа `ch`. При ошибке `fputchar()` возвращает `EOF`.

Чтение строки из потока

Для чтения строки из потока используется функция

```
#include<stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Функция `fgets()` читает символы из потока в строку `s`. Функция заканчивает чтение, когда она либо прочтет n-1 символ, либо встретит символ `\n`, который заносит в конец строки `s`. Байт `\0` добавляется к `s` для идентификации конца строки. Если функция выполняется со значением `n` равным 1, то в `s` формируется пустая строка. Функция при успешном выполнении возвращает значение указателя на строку, а при обнаружении ошибки или конца файла - `NULL`.

Запись строки в поток

Для записи строки в поток используется функция

```
#include<stdio.h>
int fputs(const char *s, FILE *stream);
```

Функция `fputs()` выводит в поток строку `s`, ограниченную байтом `\0`, который в поток не выводится. При успешном выполнении функция возвращает значение последнего записанного символа строки, либо значение 0, если строка пуста. В противном случае - значение `EOF`.

Пример:

```
#include <stdio.h>

int main(void)
{
   /* пишет строку в стандартный выходной поток */
```

```
   fputs("Здравствуй, мир\n", stdout);
   return 0;
}
```

Контроль события EOF

Ряд функций чтения/записи данных из файла могут возвращать значение EOF, которое означает либо событие конца файла, либо ошибку ввода/вывода. Для выяснения, какое из указанных событий реально возникло, служат функции:

```
#include<stdio.h>
int feof(FILE *stream);
int ferror(FILE *stream);
```

Признак конца файла сбрасывается при каждой операции ввода/вывода. Функция feof() проверяет поток на наличие признака конца файла и возвращает ненулевое значение, если обнаружен признак конца файла во время выполнения последней операции ввода/вывода в потоке, или 0 - если не был обнаружен конец файла. Как только этот признак в потоке установлен, последующие выполнения функций чтения/записи файла возвращают значение этого признака до тех пор, пока не будет вызвана функция rewind() или файл не будет закрыт.

Функция ferror() проверяет поток на наличие признака ошибки чтения/записи и возвращает ненулевое значение, если признак обнаружен во время выполнения последней операции ввода/вывода в потоке, или 0 - если признак не обнаружен. Если установлен признак ошибки потока stream, то он сохраняется до вызова функции

```
void clearerr(FILE *stream);
```

или

```
void rewind(FILE *stream);
```

или до закрытия потока. Функция clearerr() принудительно очищает признак ошибки потока. Функция rewind() перемещает указатель позиции файла в начало файла и очищает признак конца файла и признак ошибки в потоке.

Пример:

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   /* открыть файл для записи */
   stream = fopen("DUMMY.FIL", "w");

   /* попыткой чтения вызвать условие для ошибки */
   (void) getc(stream);
   if (ferror(stream))    /* проверка наличия ошибки в потоке */
```

```
    {
       /* вывести сообщение об ошибке */
       printf("Ошибка при чтении DUMMY.FIL\n");

       /* сбросить признак ошибки и конца файла EOF */
       clearerr(stream);
    }
    fclose(stream);
    return 0;
}
```

Форматный доступ к файлу в текстовом режиме

Форматные преобразования и запись данных в поток

```
#include<stdio.h>
int fprintf(FILE *stream,const char *format[,аргумент,...]);
```

Функция `fprintf`() позволяет использовать переменное число аргументов. Функция выполняет вывод данных в поток `stream`. Формат вывода (шаблон представления) для каждого аргумента задается в строке формата, определяемой указателем `format`. Число задаваемых в строке формата спецификаций форматов должно совпадать с количеством аргументов или быть меньше. Если их будет больше, то результат будет непредсказуемым (возможно, катастрофическим). Лишние аргументы (если их число больше, чем спецификаций формата) просто игнорируются.

Функция `fprintf()` принимает набор аргументов, применяет к каждому из аргументов соответствующую спецификацию формата из строки формата `format` и записывает символьное представление значения аргумента в поток `stream`. Форматная строка управляет преобразованием, форматированием и записью символьных представлений значений аргументов в поток.

Форматная строка - это строка символов, содержащая два типа объектов - простые символы и спецификации форматных преобразований. Простые символы записываются в выходной поток без изменений. Спецификации форматных преобразований применяются для форматирования значений аргументов из списка.

Спецификаторы формата имеют следующую форму:

%[флаги][ширина][.точность][модификатор_размера]type

Каждая спецификация преобразования начинается с символа процента <%>, после чего следуют признаки в приведенном ниже порядке:

флаги - необязательная последовательность символов флагов;

ширина - необязательный спецификатор ширины;

точность - необязательный спецификатор точности;

модификатор_размера - необязательный модификатор принимаемого размера;

type - форматный код преобразования.

Ниже приводятся основные правила управления форматом вывода, включая необязательные символы, спецификаторы и модификаторы форматной строки.

Флаги управляют выравниванием выводимых символов, выводом знаковых символов ('+' и '-') числа, пробелов, десятичной точкой, восьмеричными и шестнадцатеричными префиксами.

Ширина – целое число, которое определяет минимальное число выводимых символов, с учетом пробелов и нулей.

Точность – целое число, которое определяет максимальное число выводимых символов; или минимальное число выводимых цифр для целых чисел. Если не задана, то устанавливается по умолчанию.

Модификатор размера, принимающий значения h, `l` и L. Используется как префикс с форматными кодами *type*. Модификатор влияет на то, как функция интерпретирует тип соответствующего аргумента. Он заменяет тип аргумента по умолчанию для заданного кода типа на другой, соответствующий модификатору, следующим образом:

h – Используется с кодами типов d, i, o, u, x и X. Определяет, что аргумент является `short int`.

`l` - Используется с кодами типов d, i, o, u, x и X. Определяет, что аргумент является `long int`; также используется с кодами типов c, e, f, g или G, чтобы показать, что аргументы имеют тип `double`, а не `float`.

L - Используется с кодами типов e, E, f, g или G. Аргумент интерпретируется как `long double`.

Форматный код *type* специфицирует с помощью односимвольного кода тип аргумента и способ символьного представления значения аргумента в поле файла. В следующей таблице приводится список кодов, соответствующие им типы аргументов и представление значения аргумента в поле файла после преобразования. Приводимая информация основана на предположении, что в спецификацию формата не включены флаговые символы.

Таблица

Код типа	Тип аргумента	Представление значения аргумента в поле файла
d	int	Десятичное целое со знаком
i	int	Десятичное целое со знаком
o	int	Беззнаковое восьмеричное целое
u	int	Беззнаковое десятичное целое
x	int	Беззнаковое шестнадцатиричное целое с цифрами a,b,c,d,e,f
X	int	Беззнаковое шестнадцатиричное целое с цифрами A,B,C,D,E,F
f	double	Число с фиксированной точкой в форме [-]ddd.dddddd, где число цифр после десятичной точки соответствует заданной в формате точности
e	double	Число с плавающей точкой в форме [-]d.ddde[+/-]ddd, где десятичной точке предшествует одна цифра; количество цифр после десятичной точки соответствует заданной в формате точности; степень всегда содержит по крайней мере 2 разряда
E	double	Аналогично e, но с E для экспоненты ([-]d.ddddE[+/-]ddd)
g	double	Значение со знаком в форме e, E или f, в зависимости от заданного в формате значения и точности. Конечные нули и десятичная точка

		выводятся только в случае необходимости
G	double	Аналогично g, но с G для экспоненты в случае использования формата e.
c	char	Одиночный символ
s	char *	Последовательность символов до признака конца строки '\0', либо до достижения количества символов, равного явно заданному в формате значению точности
%	нет	Выводится символ %
n	int *	в ячейке, на которую указывает аргумент, сохраняется число, означающее количество символов, выведенных в выводной поток к данному моменту выполнения функции
p	Указатель (type *)	Значение аргумента выводится как значение указателя `void*`, представленное в виде 16-ричного целого

К символам-флагам относятся: минус(-), плюс(+), диез(#) и пробел ( ). Их можно указывать в любом порядке; допустимы комбинации флагов. Влияние флагов на представление значения аргумента в поле файла следующее:

(-) - Выравнивание результата по левому краю поля и заполнение правого края пробелами. Если не задан, то осуществляется выравнивание справа, и левый край заполняется пробелами или нулями.

(+) - Результаты знакового преобразования всегда начинаются со знаков плюс(+) или минус(-). флаг плюс (+) имеет приоритет над пробелом при одновременном их использовании.

пробел - Если значение неотрицательно, то вывод начинается с пробела вместо знака плюс; отрицательные значения - всегда со знака минус.

(#) - Определяет, что аргумент преобразуется с использованием "альтернативной формы".

Альтернативные формы

Флаг # влияет на представление значения аргумента для заданного кода типа следующим образом:

c,s,d,i,u - Не влияет.

o - Ненулевое значение аргумента будет начинаться с нуля.

x или X - Значение аргумента будет начинаться с 0x (или 0X).

e, E или f - Результат всегда будет содержать десятичную точку, даже если за ней не следуют цифры. Обычно десятичная точка появляется только в случае, когда после нее следуют цифры.

g или G - Аналогично e и E, но конечные нули не удаляются.

Спецификатор ширины устанавливает минимальную ширину поля для выводимого значения. Ширина определяется одним из двух способов: непосредственно с помощью строки десятичных цифр, или косвенно - с помощью знака (*). Если используется звездочка в качестве спецификатора ширины, то следующий аргумент (который должен быть целым) определяет минимальную ширину выводимого поля.

Отсутствие спецификации ширины или его недостаточное значение не вызывают усечение выводимого значения. Если результат преобразования больше, чем позволяет ширина поля, то поле просто расширяется до размеров результата.

Спецификатор точности влияет на точность вывода значения аргумента. Если точность явно не указана, то по умолчанию установлена следующая точность: 1 -

для d, i, o, u, x, X типов; 6 - для e, E, f типов; все значащие цифры – для g и G типов; вывод до первого нулевого байта '\0' - для типов s; не влияет - на тип c. Если указана точность 0, то для типов d, i, o, u, x точность устанавливается по умолчанию. Для типов e, E, f не печатается десятичная точка. Если явно задана точность N, то выводится N символов строки или N десятичных знаков числа. Если выводимое значение содержит свыше N символов, то оно может быть усечено или округлено (случится это или нет, зависит от кода типа). Если в качестве точности используется символ звездочка (*), то спецификатор точности задается как аргумент в списке аргументов, причем он предшествует форматируемому аргументу. Если явно указана нулевая точность и для вывода задан один из целочисленных форматов (т.е. d, i, o, u, x), а выводимое значение равно нулю, то ни одной цифры не будет выведено в этом поле (т.е. поле будет представлено пробелом).

Функция `fprintf()` возвращает количество выведенных байтов. В случае ошибки возвращается `EOF`.

Замечания:

Частным вариантом функции `fprintf()` для вывода в стандартный поток вывода `stdout` является функция:

```
int printf(const char *format[,аргумент,...]);
```

Кроме того, иногда требуется выполнить форматное преобразование данного в символьное представление, а результат вместо файла поместить в буфер памяти. Для этого можно использовать функцию:

```
int sprintf(const char *buf,const char *format [,аргумент,...]);
```

Пример 1

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int i = 100;
   char c = 'C';
   float f = 1.234;

   /* открыть файла для обновления */
   stream = fopen("DUMMY.FIL", "w+");

   /* записать некоторые данные в этот файл */
   fprintf(stream, "%d %c %f", i, c, f);

   /* закрыть файл */
   fclose(stream);
   return 0;
}
```

Пример 2

```
/* Программа для создания дубликата файла my_file.dat */
```

```
#include<stdio.h>

int main(void)
{
  FILE *in, *out;

  if ((in = fopen("/houme/my_file.dat", "rt")) == NULL);
  {
     fprintf(stderr, "Невозможно открыть входной файл.\n");
     return 1;
  }
  if ((out = fopen("/houme/my_file.bac", "wt")) == NULL);
  {
     fprintf(stderr, "Невозможно открыть выходной файл.\n");
     return 1;
  }
  while (!feof(in))
     fputc(fgetc(in), out);
  fclose(in);
  fclose(out);
  return 0;
}
```

Ввод и форматное преобразование данных из потока

```
#include<stdio.h>
int fscanf(FILE *stream, const char *format[,address,...]);
```

Функция `fscanf()` посимвольно сканирует набор вводимых полей, считывая их из потока `stream`. Каждое введённое из потока поле форматируется в соответствии со спецификацией формата, которая передается в виде указателя на форматную строку `format`. Преобразованные данные присваиваются переменным, адреса которых передаются в качестве аргументов функции, следующих после аргумента `format`. Количество спецификаций формата должно совпадать с количеством адресов, переданных функции.

Форматная строка управляет вводом, преобразованием и запоминанием данных из полей ввода. При этом для заданных спецификаций формата должно быть достаточное количество адресных аргументов, в противном случае результат работы функции непредсказуем и может привести к катастрофическим последствиям. Лишние адресные аргументы (которым нет соответствующих форматов) просто игнорируются. Форматная строка является символьной строкой, которая содержит три типа объектов: пробельные символы, отображаемые символы и спецификации формата.

Пробельными символами являются пробел ( ), символ табуляции (\t) и символ перехода на новую строку (\n). Если функция `fscanf()` встречает пробельный символ в форматной строке, она будет считывать, но не сохранять всю последовательность символов вплоть до следующего отображаемого символа во входном потоке.

Отображаемыми символами являются все другие символы кода ASCII, за исключением символа процента (%). Если функция `fscanf()` встречает в строке форматов отображаемый символ, то она ожидает такой же символ в потоке, прочитает соответствующий ему символ из потока, но не выполняет никаких присвоений.

Спецификации формата предписывают функции `fscanf()` осуществить чтение и преобразование символов из входного поля в значения определенного типа, затем запомнить их по адресу, указанному соответствующим адресным аргументом.

Завершающий (последний) пробельный символ не читается (включая символ перехода на новую строку), если только он не описан явно в форматной строке.

Спецификации формата функции `fscanf()` имеют следующий вид:

% [*] [widht] [type_length] <форматный_код>

Спецификация каждого формата начинается с символа процента <%>. После этого символа следуют символы спецификации в следующем порядке:

* - необязательный символ подавления назначения;

width - необязательная спецификация ширины поля;

type_length - необязательный модификатор длины типа аргумента;

<форматный код> - символ кода форматного преобразования.

Символ спецификации <*> отменяет присваивание следующего поля ввода. Спецификация ширины поля <width> задаёт максимальное число считываемых символов. Меньшее количество символов может быть считано в случае, если функция `fscanf()` встретит пробельный или непреобразуемый символ.

Форматные коды. Следующая таблица содержит список форматных кодов, типы переменных, указываемых аргументами, и вид представления вводимых значений в полях файла.

Таблица

Форматный код	Тип аргумента	Представление значения в поле файла
d	int*	Десятичное целое
o	int*	Восьмеричное целое со знаком
i	int*	Десятичное, восьмеричное или шестнадцатеричное целое со знаком
u	unsigned int*	Десятичное целое без знака
x, X	int*	Шестнадцатиричное целое
a,A,e,E, f,F,g,G	float*	Число с плавающей точкой [+/-] ddddddddd [.] dddd [E|e] [+/-] ddd
s	char*, unsigned char*	Последовательность непробельных символов до достижения количества символов, равного заданной длине поля widht, или до первого пробельного символа, если длина явно не указана. Кроме того автоматически добавляется признак конца строки - '\0'.
c	char*	Любая последовательность символов в количестве widht или

		одиночный символ, если длина не указана. Пробельные (пробельные) символы не пропускаются. Чтобы прочитать следующий отображаемый символ, следует использовать спецификацию %1s.
%	нет	Вводится символ %, преобразования не осуществляется, символ % сохраняется.
n	int *	Ввода данных не происходит, а в переменную, на которую указывает соответствующий аргумент, заносится целое значение, равное количеству ранее успешно считанных символов.
p	Указатель (type *)	Шестнадцатеричное целое, соответствующее рассмотренному выше типу x, которое трактуется как значение указателя неопределенного типа `void*`. Значение присваивается преременной, на которую указывает аргумент.

Полями ввода могут быть:

- все символы до следующего пробельного символа (не включая его);
- все символы до первого непреобразуемого по указанной спецификации формата символа (например, числа "8" и "9" по восьмеричному формату);
- последовательность символов, в количестве, определяемом значением width.

Для некоторых спецификаций формата приняты определенные соглашения, которые приведены ниже:

%c - по этой спецификации читается следующий символ, включая пробельный символ. Для пропуска пробельных символов и чтения одного отображаемого символа следует использовать спецификацию %1s.

%<width>c - преобразование предполагает, что аргумент является указателем на массив символов; массив состоит не менее чем из width элементов (`char arg[width]`).

%s – преобразование предполагает, что аргумент является указателем на массив символов (`char arg[]`). Размер массива должен быть не менее (n+1) байт, где n - длина строки s (в символах). Поле ввода завершается пробельным символом или символом перехода на новую строку <\n>. Ограничитель строки <\0> автоматически добавляется в строку после считывания и хранится в последнем элементе массива.

Поле ввода является строкой, не разделенной пробельными символами. Функция `fscanf()` считывает соответствующее поле ввода до первого символа, не являющегося элементом допустимого для текущей спецификации формата набора символов.

Символ <*> является *символом подавления присваивания* при вводе значений с помощью функции `fscanf()`. Если этот символ следует в спецификации формата за символом <%>, то следующее входное поле будет считано, но не присвоено переменной, адресуемой следующим аргументом.

Предполагается, что подавляемые входные данные специфицируются форматным кодом, который следует за символом <*>.

Спецификатор ширины <widht>, десятичное целое число, задает максимальное количество символов, которые будут считаны из текущего поля ввода. Если входное поле содержит менее, чем <widht> символов, функция считывает вначале в этом поле все символы, а затем обрабатывает следующее поле и спецификацию формата. Если пробельный или непреобразуемый символ встретился в пределах указанной ширины поля ввода, то функция считывает, преобразует и размещает по указанному адресу символы, находящиеся до пробельного или непреобразуемого символа, после чего функция обратится к следующей спецификации формата.

Непреобразуемыми считаются те символы, которые не могут быть преобразованы в соответствии с указанной спецификацией (такие как, например, символы "8" или "9" для восьмеричного формата и "J" либо "K", если указан шестнадцатиричный или десятичный формат).

Модификаторы длины типа аргументов позволяют уточнить длину типа переменных, указываемых аргументами, которым присваиваются вводимые значения. Могут использоваться следующие одно- или двухсимвольные модификаторы:

hh - для форматных кодов d, i, o, u, x, X или n означает требование преобразовать в значение типа `signed char` или `unsigned char`.

h - для форматных кодов d, i, o, u, x, X или n означает требование преобразовать в значение типа `short` или `unsigned short`.

l - для форматных кодов d, i, o, u, x, X или n означает требование преобразовать в значение типа `long` или `unsigned long`. Для форматных кодов a, A, e, E, f, F, g или G означает требование преобразовать в значение типа `double`.

L – для форматных кодов a, A, e, E, f, F, g или G означает требование преобразовать в значение типа `long double`.

z – для форматных кодов a, A, e, E, f, F, g или G означает требование преобразовать в значение типа `size_t`

Функция `fscanf`() возвращает количество аргументов, для которых было успешно выполнено присвоение введённых значений. В случае ошибки или достижения конца файла возвращается `EOF`.

Замечания:

Частным вариантом функции `fscanf()` для ввода из стандартного потока ввода `stdin` является функция:

```
int scanf(const char *format[,аргумент,...]);
```

Кроме того, иногда необходимо выполнить форматное преобразование данного, представленного в символьном виде, но находящегося не в файле , а в буфере памяти. Для этого можно использовать функцию:

```
int sprintf(const char *buf,const char *format [,аргумент,...]);
```

Пример1:

```
…
short val;
…

/* Из стандартного потока ввода (буфер клавиатуры) ввести поле
длиной не более 3-х символов и преобразовать его в целое значение
типа short int */

fscanf(stdin, "%3hd", &val);
```

Пример2:

```
…
char Arr[81];
…
/* Из стандартного потока ввода ввести строку символов до
пробельного символа, длиной не более 80 символов*/

scanf("%s", Arr);
…
```

Пример3:

```
…
int a, b, c;
…
/* Из стандартного потока ввода ввести три поля, длиной
соответственно 6, 6 и 3 символа, содержащих три целых значения*/

scanf("%6d%6d%3d", &a, &b, &c);
```

Пример4:

```
…
int group;
char Initial;
char Famil[20];
…
    scanf("%s %c%*s №гр %d%*s", Famil, &Initial, &group);

    printf("%s%c. №гр %d", Famil, &Initial, &group);

…
```

Если в четвёртом примере положить, что в буфер клавиатуры введена последовательность символов <Иванов Пётр №гр 6405 очно>, то результат печати будет:

Иванов П. №гр 6405

Доступ к файлу в двоичном режиме

Если содержимое файла не является только текстом, то его необходимо открывать в двоичном режиме. В этом случае файл рассматривается системой

как последовательность байт. Для доступа к файлу, открытому в двоичном режиме, используются функции `fwrite()` и `fread()`.

Запись в файл

```
#include<stdio.h>

size_t fwrite(const void *buf,
              size_t size,
              size_t n,
              FILE *stream);
```

Функция `fwrit()` записывает содержимое буфера `buf` из `n` элементов данных, каждый длиной в `size` байт, в поток `stream`. Общее число записанных байт равно произведению `n` на `size`.

При успешном завершении `fwrite()` возвращает количество реально записанных элементов. Если это количество меньше `n`, то возможен конец файла или ошибка. При ошибке устанавливается `errno`.

Пример:

```
#include <stdio.h>

struct mystruct{int i; char ch;};

int main(void){
   FILE *stream;
   struct mystruct s;

   /* открыть файл TEST.$$$ */
   if ((stream = fopen("TEST.$$$", "wb")) == NULL) {
      fprintf(stderr, "Невозможно открыть файл.\n");
      return 1;
   }
   s.i = 0;
   s.ch = 'A';

   /* записать структуру s в файл */
   fwrite(&s, sizeof(s), 1, stream);
   /* закрыть файл */
   fclose(stream);
   return 0;
}
```

Чтение файла

```
#include<stdio.h>

size_t fread(const void *buf,
             size_t size,
             size_t n,
             FILE *stream);
```

Функция `fread()` считывает `n` элементов данных, каждый длиной в `size` байт из входного потока `stream` в буфер, заданный указателем `buf`.

При успешном завершении `fread()` возвращает количество реально прочитанных элементов. Если это количество меньше `n`, то возможен конец файла или ошибка. При ошибке устанавливается `errno`..

Пример:

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   FILE *stream;
   char msg[] = "Это тест";
   char buf[20];

   if ((stream = fopen("DUMMY.FIL", "w+")) == NULL)
   {
      fprintf(stderr, "Невозможно открыть выходной файл.\n");
      return 1;
   }
   /* ввести некоторые данные в этот файл */
   fwrite(msg, strlen(msg)+1, 1, stream);

   /* перейти на начало файла */
   fseek(stream, SEEK_SET, 0);

   /* считать данные и вывести их на экран */
   fread(buf, strlen(msg)+1, 1, stream);
   printf("%s\n", buf);
   fclose(stream);
   return 0;
}
```

Управление внутренним указателем файла

Если файл открывается для чтения и записи с целью модификации содержимого файла, то требуется управлять положением указателя файла, устанавливая его на нужную позицию, начиная с которой содержимое файла либо читается, либо перезаписывается. Для работы с внутренним указателем файла используются функции `fseek()` и `ftell()`.

Перемещение указателя файла по потоку

```
#include<stdio.h>
int fseek(FILE *stream,long offset,int whence);
```

Функция `fseek()` устанавливает указатель файла, связанный с потоком `stream`, в новую позицию, которая смещена на `offset` байт относительно места в файле, определяемого параметром `whence`. Каждая позиция предназначена для одного байта. Нумерация позиций начинается с 0 (начало файла) и продолжается с шагом 1 до конца файла. Положительным считается смещение в сторону конца файла.

Параметр whence может иметь значения определяемые следующими системными символическими константами (определенны в stdio.h):
SEEK_SET - начало файла,
SEEK_CUR - текущее положение указателя файла,
SEEK_END - конец файла.

Функция fseek() возвращает 0, если указатель успешно перемещен, и ненулевое значение при ошибке.

Частным случаем функции fseek() является функция:

```
void rewind(FILE *stream);
```

Она устанавливает указатель текущего байта на начало файла и эквивалентна fseek(stream,0L,SEEK_SET) за исключением того, что функция rewind() очищает признаки конца файла и ошибки, а функция fseek() очищает только признак конца файла.

Определение текущего значения указателя файла в потоке

```
#include<stdio.h>
long int ftell(FILE *stream);
```

Функция ftell() возвращает текущий указатель файла потока stream(). Это значение представляет собой смещение в байтах относительно начала файла. Значение, возвращенное функцией ftell(), можно в дальнейшем использовать при вызове функции fseek().

При успешном завершении функция ftell() возвращает текущее значение указателя файла в потоке. При ошибке возвращает -1L и присваивает переменной errno положительное значение кода ошибки.

Пример:

```
#include <stdio.h>

long filesize(FILE *stream);

int main(void)
{
   FILE *stream;

   stream = fopen("MYFILE.TXT", "w+");
   fprintf(stream, "Это тест");
   printf("Размер файла MYFILE.TXT равен %ld байт\n",
                                                filesize(stream));
   fclose(stream);
   return 0;
}

long filesize(FILE *stream)
{
   long curpos, length;
```

```
    /* сохранить текущее положение в этом файле */
    curpos = ftell(stream);

    /* перейти в конец этого файла */
    fseek(stream, 0L, SEEK_END);

    /* получить текущее смещение в файле */
    length = ftell(stream);

    /* восстановить сохраненное положение курсора*/
    fseek(stream, curpos, SEEK_SET);
    return length;
}
```

Функции управления буферизацией

В зависимости от типа открываемого файла по умолчанию устанавливается определённый режим (тип) буферизации. При этом имеется возможность изменить режим буферизации. Стандартная библиотека предоставляет три типа буферизации:

1. *Полная буферизация*. В этом случае операция чтения или записи завершается после того, как данные будут получены из буфера или занесены в буфер. Фактический ввод/вывод (т.е. обращение к устройству посредством системных вызовов ввода/вывода) происходит в том случае, когда буфер пуст (при чтении) или полон (при записи) и это выполняется автоматически.
2. *Построчная буферизация*. В этом случае фактический ввод/вывод выполняется построчно при обнаружении завершения строки (символ перевода каретки). Такой тип буферизации обычно используется для потоков, ассоциированных с терминальными устройствами (как правило, стандартные потоки ввода/вывода).
3. *Отсутствие буферизации*. Буферы потокам не выделяются, и ввод/вывод фактически реализуется системными вызовами. Отсутствие буферизации характерно для вывода сообщений об ошибках на терминал.

Режим буферизации для открытых потоков может быть явно изменен с помощью функций:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Функция `setbuf()` позволяет отключить/включить буферизацию для потока `stream`. Для отключения буферизации `buf` должен быть равен `NULL`. Для включения - `buf` должен указывать на буфер системного размера `BUFSIZE` (определен в файле `stdio.h`).

Функция `setvbuf()` позволяет явно задать тип буферизации. Для этого используется аргумент `type`, который может принимать следующие значения:

`_IOFBF` – полная буферизация,

_IOLBF – построчная буферизация,

_IONBF – отсутствие буферизации.

При полной или построчной буферизации аргумент size задает размер буфера, адресованного указателем buf.

Кроме того, при использовании буферизации можно принудительно очистить буфер потока, например, для принудительного переноса данных из буфера потока вывода непосредственно в файл на внешнем носителе или для очистки буфера потока ввода от ненужных данных. Для этих целей используются функции:

```
#include<stdio.h>

int fflush(FILE *stream);
int flushall(void);
```

Функция fflush() очищает буфер указанного потока и возвращает 0 при успешном завершении, или EOF, если обнаружена ошибка. Любая операция чтения, следующая за fflush() считывает в буфер потока новые данные из входного файла.

Функция flushall() очищает буферы всех открытых потоков и возвращает целое число, которое дает общее число открытых входных и выходных потоков

Пример:

```
#include <stdio.h>

int main(void)
{
   FILE *stream;

   /* создать файл */
   stream = fopen("DUMMY.FIL", "w");

   /* сбросить все открытые потоки */
   printf("%d потоков сброшено.\n", flushall());

   /* закрыть файл */
   fclose(stream);
   return 0;
}
```

Закрытие потоков

Открытые программой потоки автоматически закрываются после завершения программы. Однако для того, чтобы вновь создаваемые файлы были прописаны в файловой системе, связанные с ними потоки должны быть закрыты явно, иначе они будут рассматриваться как временные файлы. Закрытие потоков осуществляется с помощью функций:

```
#include<stdio.h>

int fclose(FILE *stream);
```

```
int fcloseall(void);
```

Функция `fclose()` освобождает буфер потока и закрывает поток `stream`. При успешном завершении возвращает 0, если обнаружена ошибка, то `EOF` и устанавливается `errno`.

Функция `fcloseall()` освобождает буферы и закрывает все открытые программой потоки, за исключением стандартных - `stdin`, `stdout`, `stderr`. Функция возвращает общее число закрытых ею потоков или возвращает `EOF` и устанавливается `errno`, если обнаружена ошибка.

Пример:

```
#include <string.h>
#include <stdio.h>

int main(void)
{
   FILE *fp;
   char buf[11] = "0123456789";

   /* создать файл в 10 байт */
   fp = fopen("DUMMY.FIL", "w");
   if (fp) {
      fwrite(&buf, strlen(buf), 1, fp);

      /* закрыть файл */
      fclose(fp);
   }
   else
      printf("Невозможно открыть файл!\n");
   return 0;
```

Функции базового ввода/вывода

Функции стандартной библиотеки ввода/вывода, обеспечивая программе максимальную мобильность, в то же время не реализуют всех возможностей по управлению вводом/выводом, предоставляемых операционной системой QNX, которые могут потребоваться при создании приложений реального времени. В этом случае открытие файла должно быть выполнено с помощью функции `open()`, которая возвращает целочисленный дескриптор открытого файла. Для доступа к открытому файлу необходимо воспользоваться функциями, реализующими системные вызовы ввода/вывода QNX.

Доступ к файлу

Функции, реализующие системные вызовы ввода/вывода QNX, следующие:

```
#include<unistd.h>

ssize_t write(int fd, void *buf, size_t nbyte);
ssize_t read(int fd, void *buf, size_t nbyte);
off_t lseek(int fd, off_t offset, int whence);
off_t tell(int fd);
```

```
int close(int fd);
```

В представленных функциях аргумент `fd` является дескриптором открытого файла.

Функция `write()` записывает в файл `nbyte` байт из буфера `buf` и возвращает количество записанных байтов.

Функция `read()` читает из файла `nbyte` байтов и записывает их в буфер `buf`, возвращает количество реально считанных байтов.

Функция `lseek()` смещает указатель позиции файла на величину `offset` относительно указанной базы `whence`, которая может принимать значения: SEEK_SET – от начала файла, SEEK_END – от конца файла, SEEK_CUR – от текущей позиции. Возвращает новое значение позиции.

Функция `tell()` возвращает текущее значение позиции файла (указателя файла).

Функция `close()` закрывает файл и возвращает нулевое значение. Все функции в случае ошибки возвращают -1 и устанавливают `errno`.

Структура и выполнение приложений реального времени

Программы, процессы, нити

Программирование приложений реального времени в операционной системе QNX базируется на использовании таких конструктивных элементов как *программы*, *процессы*, *нити*. Программы выступают в виде исполняемых файлов, находящихся в файловой системе ОС. Операционная система QNX является многозадачной. Это значит, что в общем случае в системе независимо друг от друга могут запускаться и одновременно выполняться несколько программ. Каждый такой запуск рассматривается как запуск на выполнение некоторой задачи. Очевидно, что при запуске очередной задачи ОС должна управлять выделением необходимой для выполнения соответствующей программы доли системных ресурсов, таких как память, процессор, доступ к устройствам ввода/вывода и различным другим системным ресурсам, включая услуги ядра ОС. Выделяемые программе системные ресурсы рассматриваются операционной системой как её *среда выполнения*. Эта среда выполнения программы вместе с совокупностью данных ядра ОС, описывающих образ программы в памяти и предназначенных для управления её выполнением, называют *процессом*. Создание нового процесса осуществляется каждый раз при очередной загрузке программы на выполнение. При этом не важно разные запускаются программы или одна и та же.

Каждый процесс обеспечивает загруженной для выполнения программе собственное изолированное и уникальное адресное пространство. У программы нет возможности непосредственно обратиться в адресное пространство другого процесса. Программа может считывать и записывать информацию в собственный раздел данных и в стек, но ей недоступны данные и стеки других процессов. В то же время процесс обеспечивает программе возможность взаимодействовать с программами, исполняющимися в других процессах, используя системные средства межпроцессного взаимодействия, например, посредством передачи или приема сообщений.

Исполнение программы (в процессе) начинается с особого запуска операционной системой функции `main()`. Такой способ запуска функции называется созданием *потока управления*, для обозначения которого далее будет использоваться термин "*нить*". Внутри процесса, при необходимости, нить `main()` может запускать в качестве нитей и другие функции программы. Запуск функции в качестве нити осуществляется посредством специального запроса ОС и этим отличается от обычного вызова функции в языке С. Нить `main()` будет являться родителем запущенных ею нитей (дочерних). Нити очередного поколения могут порождать нити следующего поколения и т.д. Все нити, запущенные во всех процессах, выполняются параллельно. При этом каждая нить уже не имеет внутреннего программного параллелизма и рассматривается как процедура последовательно выполняемых инструкций.

В итоге, исполнение программы в процессе в общем случае выглядит как запуск нити `main()` и последующее порождение и параллельное выполнение

совокупности нитей, которые могут асинхронно создаваться и завершаться. Исключение составляет нить `main()`, завершение которой приводит к завершению выполнения программы и удалению процесса. Таким образом, процесс является, по сути, контейнером нитей, который содержит в начальный момент одну нить - `main()`.

Далее выполнение программы в процессе будем интерпретировать как выполнение процесса.

Свойства процессно-нитиевой структуры ПРВ

При разработке логической структуры ПРВ механизм процессов обеспечивает *структурный* параллелизм, а механизм нитей - *функциональный* параллелизм ПРВ. В частности, ПРВ может состоять из одного процесса, в котором исполняются множество нитей. Однако, представление ПРВ в виде совокупности взаимодействующих процессов придает ему следующие дополнительные свойства:

- возможность модульной организации ПРВ на макро уровне;
- гибкость и конфигурируемость ПРВ;
- повышение надежности ПРВ.

Модульность ПРВ на макроуровне обуславливает возможность разработки исполняемых процессами программ независимо друг от друга. Гибкость и конфигурируемость ПРВ вытекает из возможности динамически добавлять или модифицировать процессы непосредственно при функционировании ПРВ ("на лету"). Единственная возможность установить зависимость процессов друг от друга - наладить между ними информационную связь с помощью небольшого количества средств взаимодействия. Так как поведение процесса в рамках решаемой им задачи может быть четко специфицировано, то упрощается процедура выявления и устранения ошибок в исполняемой программе. Кроме того, процессы выполняются в независимых адресных пространствах. Две нити, работающие в разных процессах, изолированы одна от другой. Это способствует большей надежности ПРВ.

Базовая архитектура QNX

Базовая архитектура `QNX` предельно компактна и основана на двух фундаментальных понятиях: микроядро и межпроцессное взаимодействие на основе сообщений. Микроядро (его называют Neutrino) обеспечивает минимально необходимый набор системных функций:

- создание и уничтожение нитей;
- диспетчеризация нитей;
- поддержка механизмов синхронизации нитей;
- поддержка механизмов передачи сообщений;
- поддержка механизма обработки прерываний;
- поддержка часов и таймеров.

Кроме этого Neutrino ничего больше не делает. Как видно из приведенного списка функций микроядро не управляет процессами. Поддержку процессов, работающих в изолированных адресных пространствах, в QNX обеспечивает специальная системная компонента, называемая *администратор процессов*. Администратор процессов выполняет следующие функции:

- управление процессами;
- управление механизмами защиты памяти;
- поддержка механизма разделяемой памяти и механизмов межпроцессного взаимодействия;
- управление пространством имен путей.

Поскольку управление процессами и памятью являются критичными функциями операционной системы, то администратор процессов скомпонован с микроядром Neutrino в единый программный модуль procnto, который еще называют *системным процессом*. Переключение контекста между нитями в одном процессе происходит без участия администратора процессов, а - между нитями в разных процессах - с участием администратора процессов.

Вся остальная функциональность QNX обеспечивается специальными процессами, называемыми *администраторами ресурсов* и системными *прикладными процессами*. С помощью администраторов ресурсов, например, реализуется доступ к устройствам внешней памяти (администраторы различных файловых систем) или сети (администратор сети qnet, администратор TCP/IP). В качестве системного прикладного процесса выступает, например, процесс, инициируемый запуском командного интерпретатора shell.

Управление процессами

Жизненный цикл процесса включает четыре этапа.

1. *Создание*. Процесс может быть создан только другим (родительским) процессом. При этом администратор процессов создает у себя необходимые управляющие структуры данных.
2. *Загрузка кода и данных процесса в ОЗУ*.
3. *Выполнение нитей*.
4. *Завершение*. Завершение процесса проходит две стадии. На первой стадии происходит освобождение ресурсов, связанных с процессом (страницы ОЗУ, открытые файловые дескрипторы и т.п.). На второй стадии код возврата завершаемого процесса передается процессу-родителю. При этом возможны следующие варианты поведения процесса-родителя:
   - процесс-родитель заблокирован в ожидании кода завершения дочернего процесса. В этом случае код завершения сразу доставляется родителю, родитель разблокируется и дочерний процесс завершается;
   - при запуске дочернего процесса процесс-родитель установил для него флаг SPAWN_NOZOMBIE, т.е. отказался от получения кода завершения

дочернего процесса. В этом случае дочерний процесс будет немедленно завершен;

- процесс-родитель не установил флаг `SPAWN_NOZOMBIE` при запуске дочернего процесса, но и не заблокирован в ожидании кода завершения дочернего процесса. В этом случае завершающийся дочерний процесс становится DEAD-блокированным процессом или "зомби". Для такого процесса администратор процессов сохраняет минимум управляющей информации, необходимой только для того, чтобы доставить код завершения родительскому процессу, когда он выполнит вызов ожидания кода завершения дочернего процесса.

Созданный процесс имеет ряд атрибутов, определяющих свойства процесса, которые операционная система учитывает при управлении процессом. К первоочередным свойствам относятся:

- идентификатор процесса (Process ID - PID);
- идентификатор родительского процесса (Parent Process ID - PPID);
- реальные идентификаторы владельца и группы (UID и GID);
- эффективные идентификаторы владельца и группы (EUID и EGID);
- идентификаторы дополнительных групп;
- текущий каталог;
- корневой каталог;
- управляющий терминал (TTY);
- номер приоритета;
- дисциплина диспетчеризации;
- маска создания файлов (UMASK).

Идентификаторы процесса

Каждый процесс имеет уникальный идентификатор PID, позволяющий ядру системы различать процессы. Когда создается новый процесс, ядро присваивает ему очередной свободный идентификатор. Присвоение идентификаторов происходит по возрастающей, т.е. идентификатор нового процесса больше, чем идентификатор процесса, созданного перед ним. Если идентификатор достиг максимального значения, следующий процесс получит минимальный свободный `PID`, и цикл повторяется. Когда процесс завершает свою работу, ядро освобождает его идентификатор.

Нити могут определить ID своего процесса с помощью функции:

```
#include <process.h>
pid_t getpid(void);
```

Каждый процесс в качестве атрибута содержит идентификатор породившего его родительского процесса - PPID. Он используется, в частности, в качестве

адреса для доставки кода завершения дочернего процесса. Если дочерний процесс создан на том же узле локальной сети, что и родительский процесс, то нити дочернего процесса могут определить ID родительского процесса с помощью функции:

```
#include <sys/types.h>
#include <process.h>

pid_t getppid(void);
```

Если дочерний процесс создан на другом узле локальной сети, то нити дочернего процесса не смогут определить ID родительского процесса с помощью этой функции. Для этого потребуется предусмотреть некоторый способ явной передачи значения ID родительского процесса дочернему.

Важную роль для процесса играют реальный и эффективный идентификаторы владельца, реальный и эффективный идентификаторы группы. Реальным идентификатором владельца дочернего процесса, является идентификатор владельца - UID, запустившего его процесса-родителя. Эффективный идентификатор владельца - EUID, служит для управления правами доступа процесса к системным ресурсам (в первую очередь к ресурсам файловой системы). Если в файле исполняемого модуля не установлен флаг `SUID`, то при запуске на его основе процесса эффективный идентификатор владельца дочернего процесса устанавливается равным идентификатору владельца родительского процесса (EUID=UID) и процесс получает (наследует) соответствующие права доступа к ресурсам системы. Если флаг SUID в файле исполняемого модуля установлен, то эффективный идентификатор владельца дочернего процесса устанавливается равным идентификатору владельца файла исполняемого модуля, использованного для запуска процесса, и получает права доступа, соответствующие владельцу файла.

Реальным идентификатором группы дочернего процесса, является идентификатор первичной или текущей группы, к которой принадлежал родительский процесс - GID. Эффективный идентификатор группы служит для определения прав доступа процесса к системным ресурсам, когда у него отсутствуют соответствующие права доступа как у владельца. Если в файле исполняемого модуля не установлен флаг SGID, то эффективный идентификатор группы дочернего процесса делается равным идентификатору группы (EGID=GID) родительского процесса. Если флаг SGID установлен, то эффективный идентификатор группы устанавливается равным идентификатору группы исполняемого файла, использованного для запуска процесса, и получает права доступа, соответствующие группе, владеющей файлом.

При регистрации пользователя в системе утилита login запускает командный интерпретатор login shell. При этом идентификаторам UID (EUID) и GID (EGID) процесса shell присваиваются значения, полученные из записи, соответствующей пользователю в файле паролей /etc/passwd. В результате командный интерпретатор приобретает права, определенные для данного пользователя и его первичной группы. Когда далее командный интерпретатор, выполняя команду, порождает соответствующий дочерний процесс, он наследует

ему все четыре идентификатора и, следовательно, процесс получает те же права, что и shell. Поскольку в текущем сеансе работы в системе конкретного пользователя прародителем всех процессов является login shell, то их идентификаторы владельца и группы будут эквивалентными.

Для получения значений идентификаторов владельцев процесса используются следующие системные вызовы:

```
#include <sys/typs.h>
#include <unistd.h>
    uid_t getuid(void);
    uid_t geteuid(void);
    gid_t getgid(void);
    gid_t getegid(void);
```

Процесс может изменить значения идентификаторов владельцев процесса с помощью системных вызовов:

```
#include <sys/typs.h>
#include <unistd.h>
    int setuid(uid_t uid);
    int seteuid(uid_t euid);
    int setgid(gid_t gid);
    int getegid(gid_t egid);
```

Системные вызовы `setuid()` и `setgid()` устанавливают сразу реальный и эффективный идентификаторы владельцев процесса, а системные вызовы `seteuid()` и `setegid()` - только эффективные. Чтобы можно было изменить идентификатор группы необходимо, чтобы имя пользователя было в списке членов этой группы в файле /etc/group.

Команды ps и sin командного интерпретатора shell позволяют вывести список процессов, выполняющихся в системе, и их атрибуты.

Текущий и корневой каталоги

Текущий и корневой каталоги процесс наследует от родительского процесса. Они могут быть изменены с помощью функций `chdir()` и `chroot()` соответственно. Определить текущий каталог процесса можно с помощью функции:

```
#include <unistd.h>
char* getcwd(char* buffer, size_t size);
```

Функция формирует строку, заканчивающуюся признаком конца строки \0, с именем текущего каталога, которая размещается в буфере памяти, указанном в `buffer`, размером, по крайней мере, `size` байт. Максимальный размер строки с именем каталога определяется значением `PATH_MAX + 1` байт.

Функция возвращает адрес строки с именем текущей каталога, или NULL, в случае ошибки, код которой помещается в `errno`.

Приоритет и дисциплина диспетчеризации процесса

Приоритет и дисциплина диспетчеризации процесса используются ядром при определении очередности запуска нитей процесса при распределении процессорных ресурсов. В QNX имеется 64 уровня приоритетов – от 0 до 63. Самый низкий приоритет – нулевой. Значение приоритета 0 имеет специальная нить микроядра, которая всегда находится в состоянии готовности к исполнению. Системные сервисы (администраторы, менеджеры) запускаются с приоритетом 1, а драйверы устройств – с приоритетом 2. Прикладные приложения запускаются с приоритетом 3. Нити с одинаковым приоритетом используют процессорные ресурсы с учетом назначенных им дисциплин диспетчеризации.

Приоритет и дисциплина диспетчеризации процесса наследуются его главной нитью от родительского процесса.

Управляющий терминал

Управляющий терминал (терминальная линия) - это терминал или псевдотерминал, ассоциированный с процессом. Этот терминал является *управляющим терминалом* процесса. Все процессы группы имеют один и тот же управляющий терминал. С управляющим терминалом процесса связан специальный файл псевдоустройства с именем `/dev/tty`. Драйвер этого псевдоустройства по существу перенаправляет запросы на фактический терминальный драйвер, который может быть различным для различных процессов.

Типы процессов

Системные процессы

Системные процессы являются процессами, порождаемыми ядром или специальными системными программами ОС. Системные процессы, порождаемые ядром, не имеют исполняемых модулей и запускаются особым образом при инициализации ядра системы. Выполняемые инструкции и данные этих процессов находятся в ядре системы, они могут вызывать функции и обращаться к данным, недоступным для остальных процессов.

Процессы демоны

Демоны - это не интерактивные процессы, которые запускаются обычным образом - путем загрузки в память соответствующих им программных модулей (исполняемых файлов), и выполняются в фоновом режиме. Обычно демоны запускаются при инициализации системы и обеспечивают работу различных подсистем ОС: терминального доступа, печати, сетевого доступа и т.п. Демоны не связаны ни с одним пользовательским сеансом работы и не могут непосредственно управляться пользователем. Большую часть времени демоны ожидают пока тот или иной процесс запросит у ядра определенную услугу, которую ядро перенаправит соответствующему демону, например, запрос на запуск процесса или открытие файла и т.п.

Прикладные процессы

К прикладным процессам относятся все остальные процессы, выполняющиеся в системе. К ним, как правило, относят процессы, порожденные в рамках пользовательского сеанса работы. Важнейшим из таких процессов является основной командный интерпретатор (`login shell`), который обеспечивает работу в `QNX`. Он запускается после регистрации пользователя в системе, а завершение работы `login shell` приводит к отключению от системы. Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но в любом случае при выходе из системы все пользовательские процессы терминируются.

Группы и сеансы

После создания процесса ему присваивается уникальный идентификатор. Дополнительно процессу назначается *идентификатор группы процессов* (process group ID). *Группа процессов* включает один или более процессов и существует, пока в системе присутствует хоты бы один процесс этой группы. Временной интервал, начинающийся с создания группы и заканчивающийся, когда последний процесс её покинет, называется *временем жизни группы*. Последний процесс может либо завершить своё выполнение, либо перейти в другую группу.

Ряд системных вызовов могут быть применимы ко всем процессам группы. Например, системный вызов `waitpid()` позволяет родительскому процессу ожидать завершения конкретного процесса или любого процесса группы.

Каждый процесс, помимо этого, является членом *сеанса* (session), являющегося набором одной или нескольких групп процессов. Понятие сеанса введено для логического объединения групп процессов, порожденных в результате регистрации и последующей работы пользователя в системе.

Процесс имеет возможность получить значение идентификатора собственной группы процессов или группы процесса, который является членом того же сеанса. Это делается с помощью системных вызовов `getpgrp()` и `getpgid()`:

```
#include <unistd.h>
pid_t getpgrp(void);
pid_t getpgid(pid_t pid);
```

Аргумент `pid` адресует процесс, идентификатор группы которого требуется узнать. Если этот процесс не принадлежит тому же сеансу, что и процесс, сделавший системный вызов, функция возвращает ошибку.

Процесс, используя системный вызов `setgid()` может стать членом существующей группы или создать новую группу.

```
#include <unistd.h>
int setpgid(pid_t pid,pid_t pgid);
```

Функция устанавливает идентификатор группы для процесса `pid` равным `pgid`. Процесс имеет возможность установить идентификатор группы для себя и своих дочерних процессов. Однако процесс не может изменить идентификатор

группы для дочернего процесса, который преобразовался в другой процесс, выполнив системный вызов `exec()`. Если значения обоих аргументов равны, то создается новая группа с идентификатором `pgid`, а процесс становится *лидером* этой группы (group leader). Группа не удаляется при завершении её лидера, пока в неё входит хотя бы один процесс.

Сеанс также имеет свой идентификатор. Идентификатор сеанса можно узнать с помощью функции `getsid()`.

```
#include <unistd.h>
pid_t getsid(pid_t pid);
```

Идентификатор `pid` должен адресовать процесс, являющийся членом того же сеанса, что и процесс, вызвавший `getsid()`. Заметим, однако, что эти ограничения не распространяются на процессы, имеющие привилегии администратора системы.

Процесс может создать и новый сеанс с помощью функции `setsid()`.

```
#include <unistd.h>
pid_t setsid(void);
```

Новый сеанс создается лишь при условии, что процесс не является лидером какого-либо сеанса. В случае успеха процесс становится *лидером сеанса* и лидером новой группы.

Понятия группы и сеанса тесно связаны с терминалом или, точнее, с драйвером терминала. Каждый сеанс может иметь один ассоциированный терминал, который называется *управляющим терминалом* (controlling terminal), а группы, созданные в данном сеансе, наследуют этот управляющий терминал. Наличие управляющего терминала позволяет ядру контролировать стандартный ввод/вывод процессов, а также возможность направить сигнал всем процессам группы, ассоциированной с терминалом, например, при его отключении. При входе в систему терминал пользователя становится управляющим для лидера сеанса - интерпретатора `shell`, и всех процессов, порожденных лидером, которые запускает пользователь из командной строки интерпретатора. При выходе пользователя из системы `shell` завершает свою работу и отключается от управляющего терминала, что вызывает отправление сигнала SIGHUP всем незавершённым процессам текущей группы. Это гарантирует, что после завершения пользователем работы в системе в ней не останется запущенных им процессов (кроме созданных им демонов и процессов, запущенных в фоновом режиме).

Запуск процессов

Для запуска процесса в файловой системе должен присутствовать файл с необходимым исполняемым программным модулем. Процесс можно запустить "вручную", с помощью командного интерпретатора shell, или из программы, используя специальные функции. При создании нового процесса происходит обращение к администратору процессов. В общем случае администратор процессов создает среду выполнения для нового процесса. После создания среды

выполнения ядро выполняет запуск нити в этом процессе. Эта нить осуществляет некоторые подготовительные действия и вызовет функцию `main()`. Для некоторых функций порядок создания процесса имеет отличительные особенности.

Запуск процесса из shell

Для запуска процесса из командного интерпретатора достаточно в качестве команды указать имя файла исполняемого программного модуля. При этом процесс может запускаться с последующим ожиданием его завершения или без ожидания (в фоновом режиме). Например, если запускается процесс на основе программного модуля my_prog, находящегося в текущем каталоге, то команда:

```
$my_prog
```

запустит процесс и shell будет ждать его завершения, после чего только выдаст на экран приглашение для ввода следующей команды. Для запуска процесса в фоновом режиме команда должна включать опцию &:

```
$my_prog &
$
```

и приглашение к вводу следующей команды появляется немедленно. Если при запуске требуется понизить приоритет процесса, то используется команда:

```
$nice my_prog
```

Порядок запуска процессов ПРВ можно полностью возложить на shell. Для этого создаются сценарии запуска (скрипты или командные файлы). Это обычные текстовые файлы, в которых в качестве строк задаются команды shell. Командный интерпретатор рассматривает введение в качестве команды имени любого текстового файла как попытку выполнить скрипт.

Программный запуск процессов

Любая нить текущего процесса может выполнить запуск нового процесса. В QNX имеются различные способы программного запуска процессов. Для этих способов существуют соответствующие им функции запуска процесса. Они следующие:

- функция `system();`
- семейство функций `exec*();`
- семейство функций `spawn*();`
- функция `fork();`
- функция `vfork()`.

Функция `system()`

```
#include <stdlib.h>
int system(const char *command);
```

Поведение функции зависит от значения аргумента `command`. Если `command` равен `NULL`, то определяется, имеется ли в системе shell. Функция

возвращает ноль, если shell отсутствует, или отличное от нуля значение, если присутствует. Если command не равен NULL, то запускается копия shell и ему, через указатель command, передается командная строка для обработки. Если command не равен NULL, то возвращается результат выполнения shell. Если shell не смог загрузиться, то возвращается -1. При успешной загрузке shell возвращается статус завершения его выполнения, соответствующий выполненной команде.

Пример:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

int main( void ){
    int rc;

    rc = system("ls");
    if( rc == -1 ) {
      printf( "shell не может запуститься \n" );
    } else {
      printf( "Результат выполнения команды %d\n",
          WEXITSTATUS( rc ) );
    }
    return EXIT_SUCCESS;
  }
```

Функции семейства exec*()

Существует набор функций семейства exec*(), которые реализуют однотипный способ запуска процессов и отличаются только некоторыми деталями. Особенность запуска процесса этими функциями заключается в следующем. Процесс, вызвавший функцию семейства exec*(), прекращает выполнять текущий программный код и начинает выполнение инструкций нового (указанного в вызове функции) программного модуля. Важно то, что идентификатор процесса - pid, при этом остается прежним. К функциям семейства exec*() относятся функции execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe(). В качестве примера рассмотрим функции execl(), execle() и execv() семейства exec*().

Объявление функции execl() имеет вид:

```
#include <process.h>

int execl(const char *path,const char *arg0,
          const char *arg1,...,const char *argn,
          NULL);
```

Возвращает -1 в случае ошибки, иначе возвращаемый результат не определен.

Пример:

```
#include <stddef.h>
#include <process.h>
…
execl( "myprog", "myprog", "ARG1", "ARG2", NULL );
…
```

Объявление функции `execle()` имеет вид:

```
#include <process.h>
int execle(const char* path,const char* arg0,
           const char* arg1...,const char* argn,NULL,
           const char* envp[]);
```

Пример:

```
#include <stddef.h>
#include <process.h>

char* env_list[] = {"SOURCE=MYDATA",
                    "TARGET=OUTPUT",
                    "lines=65",
                    NULL
                    };
execle("myprog","myprog","ARG1", "ARG2", NULL,env_list);
```

В приведенном примере окружение для вызываемой программы состоит из трех переменных окружения: `SOURCE`, `TARGET` и `lines`. Набор переменных окружения должен завершаться нулевым указателем – `NULL`.

Объявление функции `execv()` имеет вид:

```
#include <process.h>
int execv(const char *path,char *const argv[]);
```

Возвращает -1 в случае ошибки, иначе возвращаемый результат не определен.

Пример:

```
#include <stddef.h>
#include <process.h>

char* arg_list[] = {"myprog","ARG1","ARG2",NULL};

execv("myprog",arg_list);
```

В рассмотренных примерах предполагается, что программный модуль myprog находится в текущем каталоге.

Функции семейства `spawn*()`

В отличие от функций семейства `exec*()` функции семейства `spawn*()` способны создавать новый (дочерний) процесс со своим `pid`, параллельно

выполняющийся вместе с родительским процессом. Если родительский процесс каким-то образом завершается, то это не означает, что ядро по этой причине должно сразу завершить и дочерний процесс. Имеется возможность с помощью флагов управлять поведением процесса-родителя после запуска дочернего процесса:

`P_WAIT`	Родительский процесс блокируется до тех пор, пока дочерний процесс не завершится;
`P_NOWAIT`	Родительский процесс не блокируется, выполняется параллельно с дочерним процессом и должен ожидать завершения дочернего процесса;
`P_NOWAITO`	Родительский процесс не блокируется, выполняется параллельно с дочерним процессом и не должен ожидать завершения дочернего процесса (гарантирует от перехода дочернего процесса в DEAD-блокированное состояние - "зомби", при его завершении);
`P_OVERLAY`	Родительский процесс заменяется дочерним процессом как при вызове функций семейства `exec*()`;

В качестве примера рассмотрим функции `spawnl()` и `spawnve()` семейства `spawn*()`.

Объявление функции `spawnl()` имеет вид:

```
#include <process.h>
int spawnl( int mode,
            const char * path,
            const char * arg0,
            const char * arg1...,
            const char * argn,
            NULL );
```

Аргумент `mode` предназначен для задания флага управления поведением родительского процесса. Предназначение остальных аргументов то же, что и у функции `execl()`.

Возвращаемое функцией `spawnl()` значение зависит от значения флага в аргументе `mode`:

Значение mode	Возвращаемое значение
`P_WAIT`	Статус завершения дочернего процесса.
`P_NOWAIT`	PID дочернего процесса. Чтобы получить статус завершения дочернего процесса родительский процесс должен выполнить функцию `waitpid()`, которой в качестве параметра передается полученный PID дочернего процесса.
`P_NOWAITO`	PID дочернего процесса или 0, если дочерний процесс стартовал на удаленном узле. Статус завершения такого дочернего процесса получить невозможно.

В случае ошибки возвращается -1 (устанавливается `errno`).

Пример:

```
#include <stddef.h>
#include <process.h>

int exit_val;
```

```
...
exit_val = spawnl(P_WAIT,"myprog","myprog","ARG1","ARG2",NULL);
...
```

Объявление функции `spawnve()` имеет вид:

```
#include <process.h>
int spawnve(int mode,
            const char * path,
            char * const argv[],
            char * const envp[] );
```

Аргумент argv есть указатель на вектор аргументов. Значение `argv[0]` должно указывать на имя файла исполняемого модуля. Последний член `argv` должен быть `NULL` указатель. Значение `argv` и `argv[0]` не могут быть `NULL` указателем, даже если не требуется передавать какие-либо аргументы запускаемому процессу.
Аргумент `envp` есть указатель на массив указателей на строки, определяющие переменные среды. Массив должен завершаться `NULL` указателем. Определение переменной среды задается в форме:

"имя_переменной=значение_переменной"

Если значение `envp` равно `NULL`, то дочерний процесс наследует среду родительского процесса. Дочерний процесс при этом может осуществить доступ к переменным среды, используя глобальную системную переменную `environ` (определена в `<unistd.h>`).
Возвращаемое функцией `spawnve()` значение формируется по тем же правилам, что и у функции `spawnl()`.

Пример:

```
#include <stddef.h>
#include <process.h>

char* arg_list[] = {"myprog", "ARG1", "ARG2", NULL};
char* env_list[] = {"SOURCE=MYDATA",
                    "TARGET=OUTPUT",
                    "lines=65",
                    NULL
                    };
int exit_val;
...
exit_val = spawnve( P_WAIT,"myprog",arg_list,env_list);
...
```

Функция `fork()`

Функция `fork()` создает новый (дочерний) процесс, являющийся точной копией процесса, его породившего (родительского). При этом дочерний процесс имеет уникальный PID. Идентичные дочерние процессы могут иметь разных родителей их породивших.

Отметим, что дочерний процесс имеет собственную копию дескрипторов файла родительского процесса, ссылающихся на те же самые открытые файлы родителя, а также собственные копии потоков к каталогам, открытых родительским процессом. Значения `tms_utime`, `tms_stime`, `tms_cutime`, и `tms_cstime` дочернего процесса установлены в 0. Блокировки файла, предварительно установленные родителем, дочерним процессом не наследуются. Задержанные звонки таймера очищаются для дочернего процесса. Набор задерживаемых сигналов для дочернего процесса инициализируется как пустой.

После выполнения функции `fork()` родительским процессом оба процесса продолжат выполняться с оператора, следующего за вызовом функции `fork()`.

Объявление функции имеет вид:

```
#include <sys/types.h>
#include <process.h>

pid_t fork(void);
```

В дочернем процессе функция возвращает 0, а в родительском - PID дочернего процесса. В случае ошибки `fork()` возвращает -1 родительскому процессу и устанавливает `errno`.

Пример:

```
#include <stdio.h>
#include <sys/types.h>
#include <process.h>

int main(int argc,char *argv[]){

int retval;

        printf("Это родительский процесс\n");
        fflush(stdout);
        retval = fork(void);
        printf("Кто это сказал?\n");
        return EXIT_SUCCESS;
}
```

После вызова `fork()` оба процесса выполнят второй вызов `printf()`. Данное приложение выведет на экран следующее:

```
Это родительский процесс
Кто это сказал?
Кто это сказал?
```

Чтобы различить эти два процесса необходимо проанализировать возвращаемое функцией `fork()` значение в `retval`. В дочернем процессе `retval` будет иметь нулевое значение, а в родительском - содержать PID дочернего процесса. Для пояснения рассмотрим фрагмент программы:

```
printf("PID родителя равен %d\n",getpid());
fflush(stdout);
if(retval = fork(void)){
```

```
 printf("Это родитель, PID дочернего процесса %d\n", retval);
}else{
   printf("Это дочерний процесс, PID %d\n",getpid());
}
```

Относительно функции `fork()` следует отметить, что в текущих версиях QNX она может быть успешно реализована только в процессах с одной нитью.

Функция `vfork()`

Функция `vfork()` создает новый процесс, как и функция `fork()`, но в разделяемом адресном пространстве, и блокирует родительский процесс до тех пор, пока дочерний процесс не завершится или не вызовет функцию семейства `exec*()`.

Объявление функции имеет вид:

```
#include <sys/types.h>
#include <process.h>

pid_t vfork(void);
```

Организация взаимодействия между процессами

Процесс в QNX обеспечивает нитям интерфейс взаимодействия с нитями других процессами. Ключевым механизмом этого взаимодействия является *механизм обмена сообщениями*. При передаче сообщений между процессами один процесс (нити которого принимают сообщения) считается *сервером*, а другой (нити которого посылают сообщения) - *клиентом*. Поэтому механизм обмена сообщениями в QNX называют моделью "*клиент/сервер*". Один и тот же процесс может одновременно обеспечивать, как прием, так и посылку сообщений, т.е. выполнять функции и клиента и сервера. В частности, процесс может обеспечивать взаимодействие собственных нитей между собой посредством сообщений.

Для приема сообщений некоторая нить сервера должна создать объект, называемый *каналом*. В сервере может быть создан один и более каналов. В свою очередь, чтобы нити клиента получили возможность посылать сообщения в канал сервера, некоторой нитью должен быть создан объект, называемый *соединением* (*связью*) с каналом (установить соединение с каналом). С одним каналом сервера может быть установлено произвольное количество соединений одним или несколькими клиентами. Нити процессов могут создавать и уничтожать каналы и соединения по мере необходимости.

Создание и удаление каналов

Создание канала

Создание канала может быть осуществлено любой нитью процесса (например, `main()`). Для этого используется функция:

```
#include <sys/neutrino.h>
int ChannelCreate(unsigned flags);
```

Аргумент flags представляет собой набор флагов, установка которых определяет поведение канала по отношению к процессу при возникновении особых ситуаций, контролируемых ядром QNX. Значения флагов будет рассматриваться далее по мере необходимости. Пока будем использовать значения флагов по умолчанию, для чего следует положить flags равным 0.

В случае успеха функция возвращает ID созданного канала (chid). Если возникает ошибка, то возвращается -1 и в errno помещается код ошибки.

Удаление канала

Удаление канала выполняется нитью с помощью функции

```
#include <sys/neutrino.h>
int ChannelDesstroy(int chid);
```

В качестве аргумента chid выступает ID ранее созданного канала. В случае ошибки функция возвращает -1, а в errno помещается код ошибки. Если выполнение успешное, то возвращается произвольное значение отличное от -1.

Установление и удаление соединений с каналом

Установление соединения

Установление соединения с каналом выполняется нитью с помощью функции:

```
#include <sys/neutrino.h>
int ConnectAttach(uint32_t nd,
                  pid_t pid,
                  int chid,
                  unsigned index,
                  int flags);
```

nd - ID узла в сети (nd=ND_LOCAL_NODE, если узел местный);
pid - ID процесса-сервера;
chid - ID канала сервера;
index - индекс управления значением ID соединения, возвращаемым функцией. Для формирования значения ID соединения по умолчанию - устанавливается равным 0:
flags - набор флагов, установка которых определяет поведение соединения по отношению к нитям процесса-клиента, пославших сообщение по данному соединению, при возникновении особых ситуаций, контролируемых ядром QNX.

Функция ConnectAttach() устанавливает соединение клиента с каналом chid, принадлежащим серверу pid на узле nd. Если узел местный, то nd присваивается значение системной константы ND_LOCAL_NODE. Если в качестве клиента и сервера выступает один и тот же процесс, то значение pid=0.

Если flags содержит системную константу _NTO_COF_CLOEXEC, то соединение будет удаляться, когда клиент вызывает функцию семейства exec*(), чтобы запустить новый процесс.

Функция возвращает значение системного ID соединения. Система возвращает первое доступное значение ID соединения, начинающееся со значения, установленного аргументом index.

Заметим, что при взаимодействии с системными администраторами ввода/вывода также создаются соединения. При этом ID таких соединений трактуются как дескрипторы файлов. Это может приводить к нежелательным последствиям, когда в качестве значения ID создаваемого соединения будет назначено значение, которое ассоциируется с файлом. Поэтому для гарантии создания соединения с требуемым каналом необходимо в index задавать значение _NTO_SIDE_CHANNEL. Это обеспечивает получение для соединения значения ID большего, чем значение ID любого существующего дескриптора файла. Если _NTO_SIDE_CHANNEL не используется, возможны следующие последствия:

- Если дескриптор файла со значением 0 используется, а 1 не используется, то когда вызывается ConnectAttach() с установленным в index значением 0, то возвращается ID соединения, равный 1. Так как дескриптор файла 1 в системе используется как stdout, то при выполнении процессом, например, функции printf() символьная строка будет послана каналу, с которым установлено соединение. Подобные ситуации могут случаться и тогда, когда ID соединения принимают значения 0 (stdin) и 2 (stderr).
- Дочерний процесс может наследовать дескрипторы файлов родителя. При этом соединение, созданное родителем без использования _NTO_SIDE_CHANNEL в index и _NTO_COF_CLOEXEC в аргументе flags, наследуется дочерним процессом как дескриптор файла (путем дублирования дескрипторов файлов родителя). В процессе дублирования соединения как дескриптора файла серверу посылается системное сообщение _IO_DUP (первые 2 байта этого сообщения есть 0x115), в то время как сервер такого сообщения не ожидает.

Соединения, принадлежащие клиенту, могут использоваться одновременно любой нитью клиента.

Если процесс создает параллельные соединения к тому же самому каналу, система ведет счетчик связей и разделяет ресурсы внутренних объектов ядра.

Разрыв соединения

Соединение разрывается с помощью функции

```
#include <sys/neutrino.h>
int ConnectDetach(int coid);
```

В качестве аргумента coid выступает ID соединения.

Если во время разрыва соединения какие-либо нити были блокированы в результате посылки сообщения по этому соединению, то нити разблокируются, а

посылка сообщения завершается с ошибкой. В случае ошибки функция возвращает -1, а в `errno` помещается код ошибки. Если выполнение успешное, то возвращается произвольное значение отличное от -1.

Аннулирование соединений, когда необходимость в них отпадает, необходимо обязательно выполнять, так как ресурсы ядра, связанные с поддержанием соединений, не безграничны.

Рассмотрим пример соединения с каналом, идентификатор которого равен 1, принадлежащим процессу-серверу с идентификатором равным 77 и находящемуся на одном узле сети с процессом-клиентом, и его последующий разрыв будет выглядеть так:

```
int coid;

…
coid=ConnectAttach(0,77,1,0,0);
…
ConnectDetach(coid);
```

Передача сообщений

Посылка сообщения

Посылку сообщения выполняет клиент. Предварительно вызовом `ConnectAttach()` он создает соединение `coid` с каналом сервера (предполагается, что необходимые для этого значения `nd`, `pid` и `chid` сервера ему известны). Посылка сообщения осуществляется с помощью функции:

```
#include <sys/neutrino.h>

int MsgSend(int       coid,
            const   void* smsg,
            int     sbytes,
            void*   rmsg,
            int     rbytes);
```

Посылаемые данные берутся из буфера, указанного `smsg`. Предполагается, что сервер, приняв сообщение, выполняет соответствующее действие и шлет ответное сообщение, ожидаемое клиентом. Ответное сообщение сервера размещается в буфере, указанном `rmsg`. Число посланных байтов, задается в `sbytes`, а число байтов в ответе задается в `rbytes`.

Количество переданных байтов определяется минимальным размером буферов, используемых клиентом и сервером. Это гарантирует от переполнения буферов при приеме сообщения сервером и получении ответа клиентом.

Если процесс-сервер имел нить, которая ожидала прихода сообщения (была RECEIVE-блокирова на этом канале), то перенос сообщения в адресное пространство сервера осуществляется, немедленно, а принимающая сообщение нить сервера становится готовой для выполнения. Посылающая сообщение нить клиента при этом становится REPLY-блокированной. Если нити, ожидающей приема сообщения из данного канала, в сервере нет, то пославшая сообщение нить клиента становится SEND-блокированной и ставится в очередь к каналу в порядке приоритета вместе с другими нитями, так же пославшими сообщение в

этот канал. Фактический перенос данных из адресного пространства клиента в адресное пространство сервера не происходит до тех пор, пока принимающая нить сервера не выполнит функцию получения данных из канала. После этого нить клиента, пославшая данные, становится REPLY-блокированной (ждет ответного сообщения).

В случае успешного выполнения функция возвращает значение статуса, заданного в аргументе `status` функции `MsgReply()`, которую выполняет нить сервера для посылки ответа. Если возникает ошибка, то возвращается -1 и в `errno` устанавливается код ошибки или, если нить сервера вместо `MsgReply()` использовала функцию `MsgError()`, `errno` получает значение ошибки от `MsgError()`.

Функция `MsgSend()` принадлежит семейству функций `MsgSend*()` и семантически связана с функциями `ConnectAttach()`, `TimerTimeout()` и функциями семейства `MsgReceiv*()`.

Рассмотрим пример передачи сообщения процессу с ID процесса равным `pid` в канал с ID канала равным `chid`.

```
#include <sys/neutrino.h>
#include <strino.h>
#define WIDTH 80

char *smsg="Это содержимое буфера сообщения";
char *rmsg[200]; //Это буфер ответа
int coid;
...
/* pid – ID процесса, chid – ID канала */
...
//Установить соединение
coid=ConnectAttach(0,pid,chid,_NTO_SIDE_CHANNEL,0);
if(coid==-1){fprintf(stderr,"Ошибка соединения\n")
                exit(EXIT_FAILURE);
               }
//Послать сообщение
if(MsgSend(coid,smsg,strlen(smsg)+1,
             rmsg,sizeof(rmsg))==-1){
             fprintf(stderr,"Ошибка MsgSend\n")
             exit(EXIT_FAILURE);
             }
if(strlen(rmsg)>0)printf("Сервер ответил \n%s\n",rmsg);
…
```

Прием сообщения

Процесс-сервер должен принять сообщение и послать ответ клиенту. Для приема сообщения сервером используется функция:

```
#include <sys/neutrino.h>
int MsgReceive(int chid,void *msg,int bytes,struct _msg_info *info);
```

Эта функция ожидает сообщение в канале chid. Принятое сообщение размещается в буфере, адрес которого указан в msg. Размер буфера msg задается в bytes в байтах. Число принятых байтов не может превысить размера буфера (контролируется ядром).

Если сообщение уже было в канале, когда нить сервера вызывает MsgReceive(), то оно немедленно копируется ядром в адресное пространство сервера. Если сообщения в канале нет, то принимающая сообщение нить сервера переходит в RECEIVE-блокированное состояние, ожидая пока сообщение от клиента не поступит в канал. При получении сообщения нить переходит в состояние готовности к выполнению (READY).

Если значение info отлично от NULL, то в нем сохраняется дополнительная информация относительно сообщения и нити клиента, которая послала его. Если значение info равно NULL, то эта информация, при необходимости, может быть получена, с помощью функции MsgInfo() (описание этой функции содержит описание структуры _msg_info).

Если при выполнении функции возникает ошибка, то возвращается -1 и errno присваивается код ошибки. В случае успеха возвращается идентификатор rcvid, специфицирующий нить клиента, пославшую сообщение.

Посылка ответа

Получив сообщение от клиента сервер должен послать ему ответное сообщение с помощью функции:

```
#include <sys/neutrino.h>
int MsgReply(int rcvid,int status,const void* msg,int size);
```

rcvid – ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive().

status – статус завершения, который возвращается нити клиента, пославшей сообщение, при успешном завершении функции MsgSend().

msg – указатель буфера, содержащего ответное сообщение.

size – размер сообщения в байтах.

Функция посылает ответ с сообщением нити клиента, идентифицированной rcvid. При этом нить должна находится в REPLY-блокированном состоянии. Ответ может послать любая нить сервера. Важно только, чтобы на каждое принятое сервером сообщение следовал бы ответ и только один. Выполнение функции MsgSend(), вызванной нитью клиента, которой соответствует rcvid, завершается разблокированием нити и возвратом функцией MsgSend()значения статуса, заданного сервером в аргументе status при выполнении функции MsgReply().

Ядро контролирует, чтобы число байтов ответного сообщения, принимаемых клиентом, не превышало объема буфера клиента, предназначенного для приема ответа.

Функция MsgReply() не блокирует нить сервера, передача ответа выполняется немедленно. Нет никаких особых требований к порядку ответа, но в

конечном счете серверу необходимо ответить на каждое принятое сообщение, чтобы разблокировать нить клиента, пославшую сообщение.

Заметим, что в функциях `MsgSend()` и `MsgReceive()` указывается количество посылаемых и принимаемых байт. Если они не совпадают, то выбирается минимальное из указанных значений с целью согласования размеров буферов передачи и приема. Аналогичная ситуация и с функцией `MsgReply()`. То есть, при необходимости сообщение будет "урезано" и лишние байты "отброшены".

Если возникает ошибка, то возвращается -1 и `errno` присваивается код ошибки.

Пример.

```
#include <sys/neutrino.h>
…
void server(void){
int rcvid;
int chid;
char message[512];
//Создать канал
chid=ChannelCreate(0);
//Бесконечный цикл
    while(1){
     //Получить и вывести сообщение
     rcvid=MsgReceive(chid,message,sizeof(message),NULL);
     printf("Получил сообщение, rcvid = %X\n",rcvid);
     printf("Сообщение такое: %s\n", message);

     /*Подготовить отправить ответ - используем тот же буфер, что и
     для приема сообщения*/
     strcpy(message,"Это ответ");
     MsgReply(rcvid,EOK,message,sizeof(message));
    }
}
```

Сценарии ответов

Использование `MsgReply()` достаточно прозрачно. Но за внешней простотой скрывается принципиальная возможность управления активностью клиента со стороны сервера. Во-первых, сервер совершенно не обязан посылать ответ клиенту как можно быстрее, и вообще никак не ограничен во времени с ответом клиенту. Поэтому сервер, при необходимости, может целенаправленно управлять временем нахождения клиента в REPLY-блокированном состоянии. При этом сервер может принимать и обрабатывать сообщения по тому же каналу от других клиентов (возможно, от других нитей того же процесса-клиента) и отсылать им ответы. Во-вторых, ответ сервера может быть пустым и преследовать цель только разблокировать клиента, например, `MsgReply(rcvid,EOK,NULL,0)`. Если серверу необходимо проинформировать клиента о проблемах, возникших при обработке сообщения,

полученного от клиента, то в этом случае для посылки клиенту ответа более удобно воспользоваться специальной функцией:

```
#include <sys/neutrino.h>
int MsgError(int rcvid,int error);
```

`rcvid` – ссылка на нить клиента (пославшую сообщение), возвращаемая функцией `MsgReceive()`, когда сообщение получено сервером.

`error` - код ошибки, отсылаемый клиенту.

При вызове сервером этой функции, функция `MsgSend*()` на стороне клиента завершается, возвращая -1 и присваивая `errno` значение `error`. Никаких данных клиенту не пересылается. Значение `error`, равное коду `ERESTART`, заставляет клиента немедленно повторить вызов `MsgSend*()` (этот код нельзя использовать после вызова `MsgWrite()`).

Если возникает ошибка, то возвращается -1 и `errno` присваивается код ошибки. В случае успеха - значение отличное от -1.

Сообщения типа "импульс"

Импульс - это специальное сообщение системного типа `stract _pulse`. Посылка импульса осуществляется с помощью функции:

```
#include <sys/neutrino.h>

int MsgSendPulse (int coid,
                  int priority,
                  int code,
                  int value );
```

Функция посылает короткое неблокирующее сообщение в канал процесса, по соединению coid. Импульс имеет следующую структуру:

```
stract _pulse {_uint16       type;
               _uint16       subtype;
               _int8         code;
               _uint8        zero[3];
               union sigval  value;
               _int32        scoid;
               };
```

Элементы `type` и `subtype` равны нулю (признак импульса). Содержимое элементов `code` и `value` задаются отправителем. Обычно `code` указывает причину, по которой был отправлен импульс, а `value` содержит 32 бита данных, посылаемых с импульсом (т.о. всего 40 бит). Ядро предоставляет 127 отрицательных значений `code` для использования программистами по своему усмотрению. Код может быть любым 8-битным значением меньшим ноля (-127÷-1), чтобы избежать конфликта с ядром или менеджерами QNX, генерирующими импульсы. Все безопасные системные значения кодов начинаются с `_PULSE_CODE_` и определены в `<sys/neutrino.h>`. Они

заключены в диапазоне от _PULSE_CODE_MINAVAIL до _PULSE_CODE_MAXAVAIL. Элемент value имеет тип объединения вида:

```
union sigval{int   sival_int;
             void *sival_ptr;};
```

Импульс посылается с указанием приоритета. Параметр priority должен быть в пределах диапазона правильных приоритетов в диапазоне от sched_get_priority_min() до sched_get_priority_max().

Посылка процессом-клиентом импульса и его приём процессом-сервером имеет существенные особенности. Посылка импульса не блокирует процесса-клиента. Прием импульса выполняется как прием обычного сообщения. Отличие только в том, что функция MsgReceive() возвращает ноль (признак прихода импульса) и не требуется посылать ответ, используя функцию MsgReply(). С импульсом можно передать только 40 бит полезной информации (8-битный код и 32 бита данных).

Если серверу требуется принимать только импульсы, оставляя без внимания все другие сообщения, то в этом случае необходимо использовать функцию MsgReceivePulse():

```
int MsgReceivePulse(int             chid,
                    void            *rmsg
                    int             rbytes,
                    struct msg_info *info);
```

Заметим, что параметр info всегда равен NULL.

Если по некоторому каналу принимаются и обычные сообщения, и импульсы, и при этом на нем блокированы нити, выполнившие функцию MsgReceivePulse(), и нет ни одной нити, заблокированной при выполнении функции MsgReceive(), то импульсы будут обслуживаться, а обычные сообщения обслуживаться не будут. Клиенты, пославшие обычные сообщения, будут SEND-блокированными до тех пор, пока какая-либо нить сервера не выполнит функцию MsgReceive(). Но при этом она может принять как обычное сообщение, так и импульс! Поэтому в этом случае необходимо обязательно контролировать возврат функцией MsgReceive() нулевого значения как признака приёма импульса.

Типичным программным фрагментом процесса-сервера, обрабатывающего импульсы, является:

```
#include <sys/neutrino.h>
#define MY_PULSE_TIMER …
stract _pulse *pulse;
char msg[…];
…
rcvid=MsgReceive(chid, msg, …);
if(rcvid==0){//Пришел импульс
            pulse = (stract _pulse *) msg;
```

```
                //Определить тип импульса
                switch(pulse->code){
                case MY_PULSE_TIMER://Сработал таймер
                …
                break;
                //и так далее
} else {//Обычное сообщение, обработать его
}
```

Управление сообщениями

Функции `MsgSend()`, `MsgReceive()`, `MsgReply()` должны указывать буфер фиксированной длины для приема/передачи сообщения. Однако не всегда имеется возможность заранее знать предельную длину сообщений. Сервер, например, может не знать, какие по длине сообщения ему придется принимать от клиентов, а также какие по длине ответы будут формироваться в результате обработки принятых сообщений. В таких случаях серверу может потребоваться осуществлять прием/передачу сообщений и ответов по частям, используя при приеме сообщения наряду с функцией `MsgReceive()` еще и вызовы функции `MsgRead()`, а при посылке ответа - вызовы функции `MsgWrite()`, прежде чем будет выполнена функция `MsgReply()`.

Управление приемом сообщений

Если сервер не располагает буфером, способным всегда целиком разместить поступающие сообщения, то он должен действовать осторожно, а именно - предварительно выяснять длину посланного клиентом сообщения. Такую информацию функция `MsgReceive()` предоставляет серверу посредством аргумента `info`, если при создании канала был установлен флаг `_NTO_CHF_SENDER_LEN`.

Аргумент `info` является структурой `_msg_info`:

```
struct _msg_info{
int nd;//ID принимающего узла
int srcnd; //ID передающего узла
pid_t pid; //ID клиента
int32_t chid; //ID канала
int32_t scoid; //внутренний системный ID, используемый ядром
int32_t coid; //ID соединения
int32_t msglen; //количество принятых сервером байтов сообщения
int32_t tid; //ID нити, пославшей сообщение
int16_t priority; //приоритет нити, пославшей сообщение
int16_t flags; //дополнительные информационные флаги
int32_t srcmsglen; /*длина посланного сообщения в байтах (это поле
                     актуально, если при выполнении функции
                     ChennelCreat() был установлен флаг
                     _NTO_CHF_SENDER_LEN)*/
}
```

Чтобы выяснить, целиком ли принято посланное клиентом сообщение, серверу достаточно проанализировать значения полей `msglen` и `srcmsglen`

аргумента info. Если msglen < srcmsglen, то сервер принял только часть посланного сообщения, которая уместилась в буфере сервера. Остальная часть осталась в буфере клиента. Используя функцию MsgRead() сервер имеет возможность по частям дополучить сообщение.

Функция MsgRead() имеет прототип:

```
#include <sys/neutrino.h>
int MsgRead(int rcvid,void* msg,int bytes,int offset);
```

Аргументы функции:

rcvid - ссылка на нить клиента (пославшую сообщение), возвращаемая функцией MsgReceive(),
msg - буфер сервера для приема части сообщения,
bytes - длина принимаемой сервером части сообщения,
offset - смещение относительно начала сообщения в буфере клиента.

Выполнение сервером функции MsgRead() оставляет соответствующую нить клиента в REPLY-блокированном состоянии. Поэтому сервер может многократно выполнять эту функцию для получения всего посланного нитью сообщения по частям, выделяя (например, динамически) буферы для каждой части или сразу обрабатывая принятую часть сообщения в одном и том же буфере. Когда прием и обработка сообщения полностью завершается, сервер, как и прежде, должен выполнить функцию MsgReply() для вывода нити клиента из REPLY-блокированного состояния.

Управление передачей ответа

Кроме приема по частям сообщения от клиента сервер может передавать по частям и ответ клиенту. При этом предполагается, что клиент располагает буфером, достаточным по величине для приема любого возможного ответа целиком. В противном случае ответ будет, все равно, принят только частично, заполнив буфер приема ответа, выделенный клиентом.

Для передачи клиенту ответа по частям сервер должен перед выполнением функции MsgReply() использовать функцию MsgWrite(), которая имеет прототип:

```
#include <sys/neutrino.h>
int MsgWrite(int rcvid,void* msg,int bytes,int offset);
```

Эта функция пишет данные в буфер ответа нити, ID которой указан в rcvid. Значение ID нити возвращается функцией MsgReceive() при получении сообщения сервером. Нить, по отношению к которой выполняется запись, должна быть в REPLY-блокированном состоянии. Выполнение MsgWrite() не выводит нить из REPLY-блокированного состояния.

Данные в количестве size байтов берутся из буфера, указанного в msg, и записываются в буфер ответа нити клиента, ожидающей ответ, начиная с места в буфере, отстоящего от начала буфера на величину offset байт (смещение от начала буфера).

Если размер ответа size превышает размер буфера ответа клиента, то переполнения буфера не произойдет, а превышающая размер буфера часть ответа не будет передана.

Чтобы закончить передачу ответа и вывести клиента из REPLY-блокированного состояния, серия вызовов MsgWrite() должна быть завершена вызовом функции MsgReply(). При этом ответ, отправляемый функцией MsgReply() не должен обязательно содержать какие-либо данные. Если он все-таки содержит данные, то они будут всегда записываться с нулевым смещением в буфере ответа нити назначения. Это - удобный способ записи заголовка ответа, когда он полностью передан.

Функция MsgWrite() возвращает число реально переданных байтов ответа. В случае ошибки возвращается -1 и устанавливается errno.

Передача сообщений с использованием векторов ввода/вывода

Если сообщение состоит из несвязных частей, то его передача может осуществляться с использованием так называемых векторов ввода/вывода. Под вектором ввода/вывода (IOV) понимается структура, которая содержит два поля - адрес и длину части сообщения. Для определения IOV используется системный тип iov_t, имеющий определение вида:

```
typedef struct iovec{
void *iov_base;
size_t iov_len;
} iov_t;
```

Для инициализации значения IOV удобно использовать системную макрокоманду:

```
SETIOV(_iov, addr, _len);
```

где:

_iov - имя вектора ввода/вывода;

addr - адрес части сообщения;

_len - длина части сообщения в байтах.

При передаче несвязного сообщения для каждой части такого сообщения формируется свой вектор ввода/вывода. Затем из них формируется массив векторов ввода/вывода, в котором вектора располагают в нужном порядке следования частей сообщения при передаче.

Для посылки клиентом сообщения с использованием IOV применяется функция:

```
#include <sys/neutrino.h>

int MsgSendv(int            coid,
             const iov_t*   siov,//Массив IOV сообщения
             int            sparts,//Количество IOV сообщения
             const iov_t*   riov,//Массив IOV ответа
             int            rbytes);//Количество IOV ответа
```

Для приема сервером сообщения в несвязную область памяти (буфер) с использованием IOV применяется функция:

```
#include <sys/neutrino.h>
int MsgReceivev(int chid,
                const iov_t * riov, //Массив IOV ответа
                int rparts, //Количество IOV ответа
                struct _msg_info * info ); //Доп. информация
```

Заметим, что структура буфера сообщения клиента и буфера приема сервера не обязаны совпадать. Кроме того, клиент и сервер не обязаны договариваться о способе передачи/приёма сообщения, т.е. клиент, например, может использовать для посылки сообщения функцию MsgSend(), а сервер – MsgReceivev() и наоборот.

Организация взаимодействия распределённых процессов

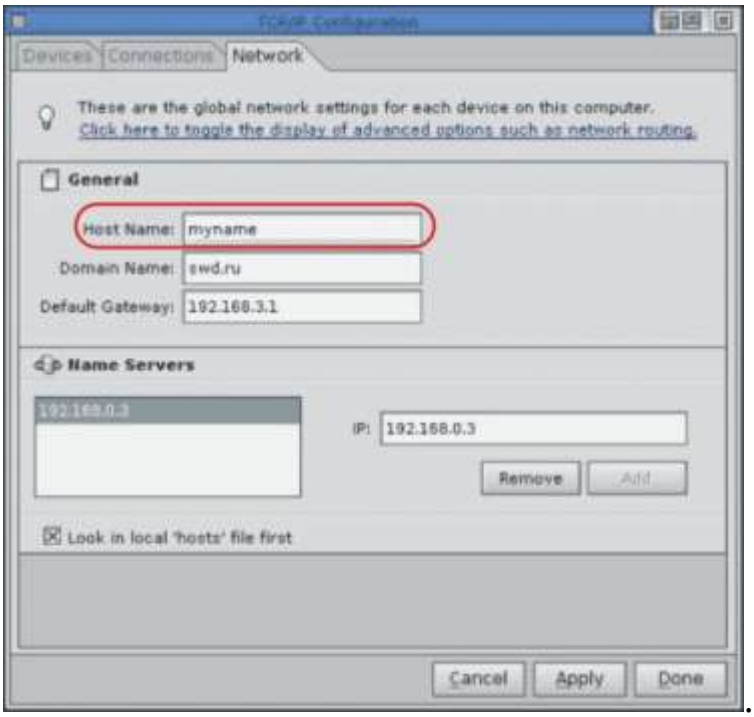
Сетевая концепция QNX

QNX изначально разрабатывалась как сетевая операционная система. Обычно локальная вычислительная сеть реализует механизм разделения файлов и внешних устройств между несколькими взаимосвязанными компьютерами. В QNX эта концепция получила дальнейшее развитие, в результате чего вся сеть стала представлять собой единый разделяемый набор ресурсов.

Любой процесс на любой машине сети может использовать любой ресурс любой другой машины. Для приложения нет никакой разницы между своим или удаленным ресурсом. Приложению не требуется иметь никаких специальных средств для обеспечения доступа к удаленному ресурсу. Пользователи имеют доступ ко всем файлам сети, могут использовать любое внешнее устройство и запускать приложения на любой машине сети (при условии, что они имеют на это соответствующее полномочие). Соответственно, все процессы могут взаимодействовать между собой по всей сети. Таким образом, механизм передачи сообщений в QNX, обеспечивает гибкую и прозрачную сетевую обработку.

Сетевая настройка QNX

Для работы в сети QNX располагает собственным сетевым протоколом - Qnet. Для настройки компьютера (узла) для работы в сети ему необходимо присвоить символическое имя (идентификатор). В среде PHOTON для этого нужно выполнить последовательно команды Launch -> Configure -> Network. В результате откроется окно с именем TCP/IP Configuration, в котором следует выбрать закладку Network:



На этой закладке в поле HostName вводим символическое имя компьютера. Введенное имя узла (в данном случае - myname), будет использоваться протоколом Qnet. Никаких других настроек для Qnet не требуется.

Для монтирования сети и запуска *администратора сети* (системного процесса npm-qnet, реализующего протокол Qnet для работы в сети) необходимо в окне терминала выполнить команду:

```
#mount –T io-net /lib/dll/npm-qnet.so
```

Для автоматического монтирования сети при запуске ОС необходимо эту команду включить в командный файл `/etc/rc.d/rc.local`. У этого файла должен быть установлен атрибут выполняемого файла (*executable*). При наличии этого файла он выполняется при загрузке ОС. Файл `/etc/rc.d/rc.local` может выглядеть следующим образом:

```
#!/bin/sh
mount –T io-net /lib/dll/npm-qnet.so
```

Если монтирование прошло успешно, на этом настройка сети закончена. В файловой системе локального узла появляется каталог `/net`, который содержит в себе каталоги с именами, соответствующими примонтированным к сети узлов. После этого доступ к ресурсам узла сети можно получить по пути `/net/<имя_узла>/<имя_ресурса>`.

Если узлу необходимо отсоединиться от сети, то следует выполнить команду:

```
umount /dev/io-net/en0
```

Организация взаимодействия удаленных процессов

Особенности обмена сообщениями в сети

Отличием обмена сообщениями через сеть является участие в этом администраторов сети qnet на узле клиента и узле сервера. На узле клиента администратор npm-qnet участвует в обслуживании запросов клиента, как на установление соединения, так и на посылку сообщений, а на узле сервера - обслуживает запросы на прием сообщения и посылку ответа. Поэтому необходимо учитывать следующие особенности обмена сообщениями в сети:

– запущенный на удаленном узле дочерний процесс *не может* получить `pid` удаленного родительского процесса с помощью функции `getppid()`, так как им окажется администратор сети npm-qnet.

– функция `ConnectAttach()` для установления соединения с удаленным процессом требует указать значение дескриптора удаленного узла отличное от нуля (0 – дескриптор локального узла);

– запрос дескриптора узла терпит неудачу, если связь нарушена, или компьютер отключен;

– возвращаемый функцией `ConnectAttach()` признак успешного соединения не гарантирует дальнейшей актуальности соединения, ошибка соединения может выявиться при попытке передать сообщение;

– в общем случае полученное процессом на местном узле значение дескриптора удаленного узла может потерять актуальность, если на местном узле необходимо "запомнить" удаленный узел, то целесообразно получить и запомнить имя удаленного узла в сети;

– если было установлено соединение в направлении некоторого удаленного узла, но при попытке послать очередное сообщение обнаружено, что соединение нарушено (нарушен кабель, отключилась машина и т.д.), функция передачи сообщения возвращает сообщение об ошибке, а соединение аннулируется, повторное установление соединения в направлении этого узла вновь потребует получения актуального значения дескриптора этого узла, так как ранее полученное значение не является постоянно закрепленным за этим узлом.

– функции `MsgReply()`, `MsgRead()`, `MsgWrite()`и им подобные являются блокирующими вызовами, т.к. при их реализации используются услуги администратора qnet как сервера. Выход из блокирующего состояния означает, что администратор qnet либо корректно завершил обработку вызовов (доставку сообщения), либо ошибку;

– при передаче клиентом сообщения по сети, сервер реально получает сообщение от местного администратора сети npm-qnet. Когда сервер выходит из `MsgReceive()`, он может не получить сообщение в полном объеме, как это было бы в локальном случае, даже при наличии у сервера достаточного объема буфера приема сообщения. Это связано с тем, что в сети, сообщение от клиента принимает в собственные буферы администратор сети npm-qnet в узле клиента, который затем передает принятое от клиента сообщение администратору сети npm-qnet на узле сервера. Так как размер буфера сервера для приема сообщения не может быть заранее известен администратору npm-qnet на стороне клиента, то он пересылает администратору npm-qnet на стороне сервера объем данных не превышающий фиксированного максимального размера (в настоящее время 8KB). Это означает, что, если клиент посылает, например, 1 Мбайт данных и сервер использует `MsgReceive()` с буфером приема в 1 Мбайт, то только администратор сети определяет число байтов, которые были бы фактически переданы. Число байтов, переданных серверу, можно определить, используя значение аргумента `info` функции `MsgReceive()`, или получить с помощью функции `MsgInfo()`. Оно содержится в поле `msglen` структуры `struct _msg_info`. Если выясняется, что сообщение получено не полностью, его остаток необходимо получить по частям, последовательно принимая части сообщения, размер которых не превышает возможностей администратора npm-qnet, пока не будет получено все сообщение. Для приема последующих частей сообщения после выполнения функции `MsgReceive()` необходимо нужное число раз использовать функцию `MsgRead()`. Например, можно в теле

цикла воспользоваться следующим кодом, гарантирующим получение от клиента целиком всего сообщения.

```
…
chid=ChannelCreate(_NTO_CHF_SENDER_LEN);
…
rcvid=MsgReceive(chid,msg,nbytes,&info);
…
/* Так как выполняется передача по сети, то проверяется все ли
сообщение было принято*/
if (rcvid>0 && info.srcmsglen>info.msglen && info.msglen<nbytes){
…
/*Тело цикла для приема остатка сообщения. */

     int n;
     if((n=MsgRead(rcvid,(char*)msg+info.msglen,
                    nbytes-info.msglen,info.msglen))<0)
        {
        MsgError(rcvid,-n);
        continue;
        }
     info.msglen+=n;
}
```

Определение дескрипторов удаленных узлов сети

Клиент и сервер в разных узлах сети практически аналогичны клиенту и серверу в локальном узле сети. Единственное, что требуется при взаимодействии через сеть - это указывать, при установлении соединения клиента с каналом сервера, отличное от нуля значения дескриптора узла сети - `nd`, используемого функцией `ConnectAttach(uint32_t nd,...)` в качестве первого параметра. Если указано нулевое значение, то используется местный узел. Заметим, что для указания дескриптора местного узла удобно использовать системную константу `ND_LOCAL_NODE` (её значение есть 0). Она определена в заголовочном файле `<sys/netmgr.h>`. Если значение не нулевое, запрос направляется процессу, находящемуся на узле, которому соответствует данный дескриптор.

В сети QNX 6 узел глобально представлен только своим именем (символьное имя, присвоенное компьютеру в сети). Значение дескриптора удаленного узла, соответствующего компьютеру с указанным именем, может быть получено процессом в локальном узле с помощью функции

```
#include <sys/netmgr.h>
int netmgr_strtond(const char *nodename, char **endstr);
```

Функция определяет местное значение дескриптора удалённого узла, соответствующее указанному имени узла. Если аргумент `endstr` не `NULL`, то в качестве его значения необходимо использовать адрес указателя на символ, следующий за именем узла в строке `nodename`.

Функция возвращает значение дескриптора узла, или -1, если произошла ошибка. В случае ошибки устанавливается значение системной переменной `errno`.

Важно отметить, что полученный в результате дескриптор узла `nd` не является уникальным для всей сети. Значение дескриптора, например `nd=5`, на одной машине может соответствовать узлу с именем "A", а такое же значение дескриптора на другой машине - узлу с именем "B". Другими словами, значения дескрипторов узлов сети уникальны только в пределах локального узла. Кроме того, значения дескрипторов узлов, в общем случае, носят временный характер. То есть, если в течение некоторого периода времени полученный дескриптор удаленного узла не используется (с использованием этого дескриптора не установлено никаких соединений), то администратор сети может переназначить его в качестве дескриптора другому удаленному узлу. Следовательно, полученный дескриптор узла необходимо немедленно использовать по назначению. Если имеются установленные соединения в направлении некоторого удаленного узла, то значение его дескриптора на локальном узле не изменяется.

Имя удаленного узла может быть получено различными путями. Если, например, известен актуальный для локального узла дескриптор некоторого удаленного узла, то имя этого удаленного узла может быть получено с помощью функции

```
int netmgr_ndtostr(unsigned flags,
                   int nd,
                   char buf,
                   size_t buflen)
```

По умолчанию (`flags` равен 0) функция помещает в буфер `buf` размером `buflen` байт строку, представляющую собой абсолютное (полное) имя узла, соответствующего указанному дескриптору узла `nd`. Эту строку можно послать любому другому узлу для использования её в функции `netmgr_strtond()` для получения локального значения дескриптора удаленного узла.

Для получения короткого имени локального узла можно воспользоваться функцией

```
#include <unistd.h>
size_t confstr(int namvar,
                   char buf,
                   size_t buflen)
```

Функция `confstr()` позволяет получить строку размером `buflen` байт, помещаемую в буфер `buf`, со значением параметра системной конфигурации, определённого в `namvar`. Для получения короткого имени узла в сети в качестве параметра системной конфигурации следует указать `_CS_HOSTNAME`.

Если на некотором узле C возникла необходимость в дескрипторе удаленного узла A, и ему известно, какое локальное значение имеет дескриптор узла A на удаленном узле B, а также собственное локальное значение

дескриптора узла B, тогда можно получить собственное локальное значение дескриптора узла A, используя функцию:

```
int netmgr_remote_nd(int local_nd_B_in_C, int local_nd_A_in_B)
```

Функция возвращает собственное локальное значение дескриптора узла A на узле C.

Если необходимо убедиться, что два дескриптора являются идентичными, можно воспользоваться макрокомандой `ND_NODE_CMP(nd1,nd2)`. Если возвращаемое значение является нулевым, дескрипторы `nd1` и `nd2` относятся к одному и тому же узлу. Если значение меньше или больше нуля, то дескрипторы разные. При необходимости это можно использовать для сортировки дескрипторов узлов.

Запуск процесса на удаленном узле

Рассмотренные ранее функции стандартной библиотеки C для запуска процессов не позволяют запустить процесс на удаленном узле. Для этого в QNX необходимо использовать системную функцию `spawn()`. Эта функция предоставляет максимальные средства управления запуском дочернего процесса. Функция имеет вид:

```
#include <spawn.h>
pid_t spawn(const char *path,
            int fd_count,
            const int fd_map[],
            const struct inheritance *inherit,
            char *const argv[],
            char *const envp[]);
```

Аргументы:

`path` - полное имя исполняемого модуля, используемого для запуска дочернего процесса.

`fd_count` - число элементов в массиве `fd_map`.

`fd_map` - массив дескрипторов файлов, которые наследует дочерний процесс. Если `fd_count` не 0, то `fd_map` должен содержать, по крайней мере, `fd_count` дескрипторов файлов, но не более величины `OPEN_MAX`. Если значение аргумента `fd_count` равно 0, то `fd_map` игнорируется, и при этом все дескрипторы файлов, кроме тех, для которых в дескрипторе установлен флаг `FD_CLOEXEC` (при создании или модифицированы функцией `fcntl()`), наследуются дочерним процессом.

`inherit` - структура типа:

```
struct inheritance {unsigned long flags;
                    pid_t pgroup;
                    sigset_t sigmask;
                    sigset_t sigdefault;
                    uint32_t nd}
```

которая используется для управления запуском и формирования свойств дочернего процесса, наследуемых от родительского процесса. Значения полей следующие:

`unsigned long flags` - аргумент для установки флагов управления запуском и наследованием свойств. Используются следующие флаги:

`SPAWN_SEARCH_PATH` – если не указано полное имя программного модуля (файла) в файловом пространстве, то установка флага предписывает осуществлять поиск программного модуля в каталогах, заданных в переменной среды `PATH`.

`SPAWN_SETGROUP` - установка флага предписывает включить дочерний процесс в группу с `GID`, заданным в поле `pgroup`. Если этот флаг не установлен, дочерний процесс становится членом текущей группы процессов.

`SPAWN_SETND` - установка флага предписывает запустить дочерний процесс на узле с дескриптором, заданным в поле `nd`.

`SPAWN_SETSIGDEF` - установка флага предписывает использовать поле `sigdefault`, чтобы определить для дочернего процесса набор сигналов с действиями по умолчанию. Если этот флаг не установлен, дочерний процесс наследует сигнальные действия родительского процесса.

`SPAWN_SETSIGMASK` - установка флага предписывает использовать поле `sigmask` для задания маски сигналов дочернего процесса.

`pid_t pgroup` - если в `inherit.flags` установлен флаг `SPAWN_SETGROUP`, то задает значение `GID` дочернего процесса. Если же `pgroup` присвоить значение `SPAWN_NEWPGROUP`, то дочерний процесс начинает новую группу с `GID` группы, равным `ID` дочернего процесса.

`sigset_t sigmask` – предназначен для задания маски сигналов дочернего процесса, если в `inherit.flags` установлен флаг `SPAWN_SETSIGMASK`.

`sigset_t sigdefault` - если в `inherit.flags` установлен флаг `SPAWN_SETSIGDEF`, определяет набор сигналов дочернего процесса с действиями по умолчанию.

`uint32_t nd` - если в `inherit.flags` установлен флаг `SPAWN_SETND`, то аогумент задаёт дескриптор удаленного узла, где запускается дочерний процесс.

`argv` -вектор аргументов. Значение argv не может быть равным `NULL`. Если аргументов нет, то `argv[0]` равен `NULL`. Если `argv[0]` не равен `NULL`, а количество передаваемых дочернему процессу аргументов равно `argc`, то `argv[0]` должно указывать на абсолютное имя файла, содержащего программный модуль. Последний элемент - `argv[argc+1]`, должен быть `NULL`.

`envp` - вектор указателей строк с определением переменных среды. Вектор заканчивается указателем `NULL`. Каждый указатель указывает на строку вида:

`<имя переменной среды> = <значение - строка символов>`,

которая определяет переменную среды. Если значение `envp` равно `NULL`, то дочерний процесс наследует окружающую среду от родителя.

Функция `spawn()` порождает и запускает новый дочерний процесс, на основе исполняемого модуля, который содержится в файле с именем `path`.

Дочерний процесс наследует следующие атрибуты от родительского процесса:

- ID группы процесса (если SPAWN_SETGROUP не установлен в inherit.flags).
- Принадлежность сеансу.
- Реальный ID пользователя и реальный ID группы процесса.
- ID дополнительной группы.
- Приоритет и дисциплину планирования.
- Текущий корневой и рабочий каталог.
- Маску создания файла.
- Маску сигналов (если SPAWN_SETSIGMASK не установлен в inherit.flags).
- Сигнальные действия, специфицированные как SIG_DFL.
- Сигнальные действия, специфицированные как SIG_IGN (за исключением измененных inherit.sigdefault, когда SPAWN_SETSIGDEF установлен в inherit.flags)

Дочерний процесс имеет некоторые отличия от родительского процесса:

– Набор сигналов, которые обрабатываются родительским процессом, установлены по умолчанию (SIG_DFL).

– Значения tms_utime, tms_stime, tms_cutime, и tms_cstime для дочернего процесса установливаются в нуль.

– Число секунд оставшихся до момента, когда сигнал SIGALRM будет сгенерирован, установлен для дочернего процесса в нуль.

- Набор отложенных сигналов для дочернего процесса пуст.
- Набор блокировок файлов, установленных родителем, не наследуется.
- Таймеры процесса, созданные родителем не наследуются.
- Блокировки и распределение памяти родителем не наследуются.

– Если дочерний процесс порожден на удаленном узле, то ID группы родительского процесса, и его принадлежность сеансу не наследуются; для дочернего процесса формируется новый сеанс и новая группа процессов.

Дочерний процесс имеет доступ к окружению родительского процесса, используя глобальную переменную окружения (находится в <unistd.h>).

Если path содержит путь, принадлежащий файловой системе, смонтированной с установленным флагом ST_NOSUID, то эффективный ID пользователя и эффективный ID группы будут соответствовать ID пользователя и ID группы родительского процесса. Иначе, если есть соответствующие установки, эффективный ID пользователя дочернего процесса, устанавливается равным ID владельца пути. Точно так же, если есть соответствующие установки,

эффективный ID группы дочернего процесса, устанавливается равным ID группы, являющейся владельцем пути.

Реальный ID пользователя, реальный ID группы и ID дополнительной группы дочернего процесса остаются такими, как у родительского процесса. Эффективный ID пользователя и эффективный ID группы дочернего процесса сохраняются такими, как ID пользователя и ID группы, заданные функцией setuid().

Важно отметить, что дочерний процесс, запущенный на удаленном узле, не наследует PID родительского процесса.

Связь между родительским процессом и дочерним не подразумевает, что дочерний процесс умирает, когда умирает родительский процесс.

Функция spawn() возвращает ID дочернего процесса, или -1 в случае ошибки. При этом в системной глобальной переменной errno, установливается код ошибки.

Локализация сервера

Для того чтобы передать сообщение от клиента к серверу, клиентская нить должна создать соединение с каналом сервера, используя функцию ConnectAttach(). Для этого ей необходимо знать идентификатор сервера (pid процесса), идентификатор узла локальной сети (nd компьютера, на котором запущен сервер), и идентификатор канала, созданного сервером (chid). Когда передающая и принимающая нити находятся в одном процессе, получение этих данных не представляет особого труда: достаточно присвоить идентификатор созданного сервером канала глобальной переменной процесса, и он будет доступен всем нитям в процессе. При этом идентификатор локального узла nd=0, а идентификатор процесса pid=getpid().

Сравнительно не сложно связать между собой родительский и дочерний процесс, находящиеся на одном узле локальной сети. В этом случае родительский процесс в начале можно рассматривать в качестве сервера по отношению к дочернему процессу – клиенту. Клиентская нить, при этом, сможет определить идентификатор сервера с помощью функции getppid(), идентификатор локального узла nd=0, а идентификатор канала, созданного сервером (chid), можно при запуске дочернего процесса передать как аргумент нити main(). В итоге многопроцессное приложение формируется от одного родительского процесса.

Процедура установления соединения значительно усложняется, если нити находятся в разных асинхронно запускаемых процессах. В этом случае процедура локализации сервера и установление с ним связи становится отдельной нетривиальной задачей. Одним из простых вариантов её решения является использование файла, который создается в сетевой файловой системе (на некотором известном всем узле) для ведения таблицы параметров запущенных процессов. Каждый запускаемый процесс получает уникальное в рамках приложения *символическое имя*, создаёт канал и регистрирует себя в таблице вместе с необходимыми для установления с ним соединения

параметрами: *символическое имя*, *имя узла*, *дескриптор процесса*, *дескриптор канала*. Это позволяет процессам находить друг друга по символическому имени, используя таблицу размещения, и получать требуемые параметры.

Наиболее радикальное решение этой задачи заключается в использовании технологии программирования процесса как администратора ресурсов ОС QNX. Особенностью процесса-администратора ресурсов, является то, что при запуске он регистрируется как объект файлового пространства (в пространстве имён путей) и ему присваивается символическое имя. В этом случае клиентской нити для создания соединения с каналом такого сервера достаточно использовать стандартную библиотечную функцию `open()`, указав в качестве имени файла имя, зарегистрированное сервером в системе. В то же время следует заметить, что разработка администратора ресурсов в полном объеме является трудоемкой задачей []. Однако, есть простая альтернатива, основанная на частичной реализации технологии написания администратора ресурсов (так называемый неполный администратор).

На практике из рассмотренных выше механизмов локализации сервера и установления с ним соединения обычно используют "механизм родительского процесса" и "механизм неполного администратора".

Механизм родительского процесса

Суть механизма в том, что стартовый процесс отвечает за устанавление связи и выступает по отношению дочернему процессу как родительский процесс-сервер. Запускаемый дочерний процесс при организации связи выступает в роли дочернего процесса-клиента. Если сервер запускает клиента на удаленном узле, то для его запуска может быть использована только функция `spawn()`. Для определённости ниже будем полагать, что имя узла, на котором стартует родительский процесс-сервер будет `CompSer`, а дочерний процесс-клиент запускается на узле с именем `CompCle`. Исполняемые модули обоих процессов находятся в файловой системе узла `CompSer` в каталоге `/root` и имеют имена соответственно `server` и `client`. Необходимо на узле `CompSer` запустить модуль `server`, а тот в свою очередь должен запустить модуль `client` на узле `CompCle`. После этого процесс `client` должен установить соединение с каналом процесса `server`.

Если решать эту задачу "в лоб", то возникает ряд проблем, связанных со спецификой запуска дочернего процесса на удаленном узле. После выполнения функции `spawn()` процесс `client` будет запущен на узле `CompCle`. Но при попытке установить соединение с каналом родительского процесса будут возникать ошибки. Это связано с уже рассмотренными выше особенностями работы процессов в сети:

1. дочерний процесс `client`, запущенный на узле `CompCle` своим местным узлом будет считать узел `CompSer`, так как наследует от родительского процесса корень его файловой системы в сети - `/net/CompSer/`;

2. при определении pid родительского процесса дочерним процессом client с помощью функции getppid(), она будет возвращать pid администратора сети на узле CompCle и, следовательно, необходим другой способ определения pid родительского процесса;
3. pid дочернего процесса client будет принадлежать уже списку процессов на узле CompCle.

Так как процесс client своим местным узлом продолжает считать узел CompSer, то при выполнении функции netmgr_strtond() для определения дескриптора родительского узла по имени CompSer будет возвращаться значение nd=0. Очевидно, что попытка использовать этот дескриптор для установления удаленного соединения с процессом server завершится ошибкой или выполнится не корректно (случайно на узле CompCle окажется процесс с такими же pid и chid, что и у процесса server на узле CompSer).

Для того чтобы процесс client считал местным узлом узел CompCle, необходимо поменять для процесса client корень сетевой файловой системы на /net/CompCle/. Для этого можно было бы, например, перед выполнением функции spawn(), запускающей удаленный дочерний процесс client на узле CompCle, поменять для процесса server корень сетевой файловой системы на /net/CompCle/, который и будет унаследован дочерним процессом client. Для этого используется функция chroot(). Однако возникает другая проблема, заключающаяся в том, что вернуть корень обратно невозможно. Процедура смены корня в сетевой файловой системе является необратимой! Т.е. вернуть корень на прежнее место родительскому процессу server невозможно! Это в общем случае нарушает дальнейшую работу процесса server.

Так как на удаленном узле процесс client не может получить pid удаленного родительского процесса с помощью функции getppid(), то родительскому процессу необходимо при запуске дочернего процесса client передавать свой pid в качестве аргумента.

Учтя выше сказанное, введем еще один процесс loader, в задачу которого будет входить только смена корня файловой системы на /net/CompCle/, запуск дочернего процесса client на удалённом узле CompCle и последующая терминация. Процесс server теперь может запустить процесс loader на локальном узле CompSer любой функцией семейства spawn*(). При этом процесс loader должен получить от родительского процесса server в качестве аргументов следующие параметры для запуска дочернего процесса client:

- путь к модулю client запускаемого дочернего процесса;
- имя узла CompCle;
- chid канала процесса server.

В итоге родительский процесс server сохраняет свой корень в сетевой файловой системе, а процесс client наследует корень на узле CompCle.

Исходный текст server может выглядеть следующим образом:

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <unistd.h>
#include <spawn.h>

#define MSG_LEN 80
#define REPLY_LEN 80
#define NODE_NAME "CompCle"
#define PATH_LOADER "/root/loader"
#define PATH_CLIENT "/root/client"
…
int  chid;
…
void main(){
char msg_to_receive[MSG_LEN];
char replybuf[REPLY_LEN];
char chid_ch[15];
int  rcvid;
…
chid=ChannelCreate(0);
…
itoa(chid,chid_ch,10);//Преобразование в строку-аргумент
spawnl(P_NOWAITO,PATH_LOADER, PATH_LOADER, PATH_CLIENT, NODE_NAME,
chid_ch,NULL);
…
rcvid=MsgReceive(chid,&msg_to_receive, sizeof(msg_to_receive),NULL);
if (rcvid!=-1){
MsgReply(rcvid,EOK,&replybuf,sizeof(replybuf));
…
}
```

При запуске процесса client процессом loader последний передает ему в качестве аргументов:

- имя узла CompSer (его loader может получить с помощью функции confstr());
- pid процесса server (его loader может получить с помощью функции getppid());
- chid созданного сервером канала (его loader может получить от server как аргумент функции main()).

Следует, однако, заметить, что, привязавшись перед выполнением функции spawn(), к корню файловой системы удаленного узла, доступ к объектам файловой системы местного узла необходимо уже осуществлять, явно указывая в пути имя местного узла - /net/<имя узла>/.

Исходный модуль loader может выглядеть следующим образом:

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>
#include <unistd.h>
#include <spawn.h>

#define BUFF_SIZE 80
…
int main(int argc,char **argv){

spawn_inheritance_type  inherit;
char                    *args[5]={NULL,NULL,NULL,NULL,NULL};
char                    *envp[1]={NULL};
char                    buff[BUFF_SIZE];
char                    path_client[80];
char                    pid_ch[15];
char                    path_root[80];


    confstr(_CS_HOSTNAME,buff,BUFF_SIZE);

    /*Формирование пути к модулю client на узле CompSer*/
    strcpy(path_client,"/net/");
    strcat(path_client,buff);
    strcat(path_client,"/");
    strcat(path_client,argv[1]);
    args[0]=path_client;

    args[1]=buff;//Имя узла родительского процесса

    itoa(getppid(),pid_ch,10);
    args[2]=pid_ch;//ID родительского процесса

    args[3]=argv[3];//ID канала родительского процесса

    /*Формирование пути для изменения корня ФС на узел клиента*/
    strcpy(path_root,"/net/");
    strcat(path_root,argv[2]);
    strcat(path_root,"/");

    inherit.nd=netmgr_strtond(argv[2],NULL);
    inherit.flags=SPAWN_SETND;

        chroot(path_root);//Изменение корня ФС
        if(spawn(path_client,0,NULL,&inherit,args,envp)!=-1)
              return(EXIT_SUCCESS);
        else return(EXIT_FAILURE);
    }
```

Модуль `client` должен принять аргументы и выполнить соединение с каналом сервера:

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>
```

```
        …
int  coid;
…
int  main(int argc, char **argv){
…
coid=ConnectAttach(netmgr_strtond(argv[1],NULL),atoi(argv[2]),
                   atoi(argv[3]),0,0);
…
}
```

Замечание. Переключение корня файловой системы на удаленный узел может выполнить и процесс `client`, используя функцию `chroot()`, сразу после его загрузки процессом `server` на удалённый узел. В этом случае процесс `loader` не нужен, но при запуске процесса `client` процесс `server` в качестве аргумента должен передать клиенту ещё и сетевое имя узла сервера.

Механизм неполного администратора

Механизм основан на использовании пространства имен для идентификации каналов процесса-сервера. Это даёт возможность клиенту устанавливать соединения с каналом сервера, используя имя созданного канала, предварительно зарегистрированное сервером в файловой системе. Для работы с именованными каналами предназначены следующие системные вызовы:

Функция	Описание
name_attach()	Создание именованного канала
name_detach()	Удаление именованного канала
name_open()	Создать соединение с именованным каналом
name_close()	Уничтожить соединение с именованным каналом

Процесс-сервер может создать канал и зарегистрировать в пространстве имён его имя с помощью вызова **name_attach():**

```
name_attach_t* name_attach(dispatch_t *dpp, // dispatch-драйвер
                           const char *name, // имя пути
                           unsigned flags); // флаги
```

В упрощенном варианте аргументу **dpp** следует установить значение **NULL**, в этом случае необходимые для взаимодействия с процессом системные ресурсы строятся автоматически. Аргумент **name** задает строку с именем пути, под которым канал будет зарегистрирован в пространстве имен путей (имя канала). Имя пути не должно начинаться ни с символа точки, ни с символа слеш. Внутри имени символ слеш использовать можно для создания иерархии имён. Аргумент **flags** устанавливает свойство видимости имени канала в локальной сети. Если **flags** равен 0, имя канала регистрируется как локальное, идентифицирующее канал процесса только в рамках текущего узла локальной сети. Если **flags** присвоить значение системной константы **NAME_FLAG_ATTACH_GLOBAL**, то

имя канала процесса будет зарегистрировано как глобальное, видимое в пределах всей локальной сети. Имена каналов процессов, зарегистрированные как локальные, помещаются в каталог **/dev/name/local**, а глобальные - **/dev/name/global**. Вызов возвращает указатель на структуру типа **name_attach_t**, которая включает в себя следующие поля:

```
typedef struct _name_attach{
                            dispatch_t* dpp; // dispatch-драйвер
                            int chid; // идентификатор канала
                            int mntid; // идентификатор монтирования
                            int zero[2]; // нулевые значения
                            } name_attach_t;
```

При выполнении данной функции канал создается посредством внутреннего вызова **ChannelCreate()**, а ID канала заносится в поле **chid** структуры **_name_attach**. Канал создается с флагами: **_NTO_CHF_UNBLOCK**, **_NTO_CHF_COID_DISCONNECT**, **_NTO_CHF_DISCONNECT**. Данный набор флагов устанавливает, что при попытке деблокировки клиентской нити или при разрыве установленных с каналом соединений ядро будет уведомлять сервер об этих событиях, отправляя ему в канал соответствующий уведомляющий импульс - системное сообщения с системной структурой **_pulse**:

```
struct _pulse {
    uint16_t            type;
    uint16_t            subtype;
    int8_t              code;
    uint8_t             zero[3];
    union sigval        value;
    int32_t             scoid;
};
```

Значения полей **code** и **value** в структуре пришедшего импульса однозначно соответствуют типу события. При попытке клиентской нити выйти из блокированного состояния (например, по таймауту или сигналу) поле **code** будет равно системной константе **_PULSE_CODE_UNBLOCK**, а поле **value** – значению, возвращенному серверу функцией **MsgReceive()** – **rcvid**. При разрыве одного из нескольких установленных с каналом соединений уведомляющий об этом событии импульс в поле **code** будет иметь значение системной константы **_PULSE_CODE_COIDDEATH**, а в поле **value** – ID соединения (**coid**). Если клиент терминируется, не разорвав предварительно его подключения к каналу, за него это сделает ядро. Если окажется, что все соединения с каналом разорваны, то ядро посылает серверу уведомляющий об этом событии импульс, поле **code** которого будет иметь значение системной константы **_PULSE_CODE_DISCONNECT**, а в поле **value** – **None**. В этом случае серверу необходимо аннулировать соединение ядра с каналом, выполнив вызов **ConnectDetach(scoid)**, где **scoid** – системный ID соединения ядра с каналом сервера, взятый из поля **scoid** принятого импульса (по умолчанию,

когда флажок **_NTO_CHF_DISCONNECT** не установлен, ядро удаляет соединение с каналом сервера автоматически).

Ядро поддерживает взаимнооднозначное соответствие между именем канала процесса и его ID, поэтому процесс с именованным каналом не может иметь копий в оперативной памяти.

Для установления клиентом соединения с именованным каналом используется функция:

```
int name_open(const char* name, // имя канала
              int flag); // флаг видимости имени
```

Аргумент **name** задает имя канала сервера, с которым клиентская нить устанавливает соединение. Аргумент **flags** определяет указанное имя канала как локальное или глобальное. Если **flags** равен 0, то имя канала процесс-клиент считает локальным. Если **flags** присваивается значение системной константы **NAME_FLAG_ATTACH_GLOBAL**, то имя канала процесс-клиент считает глобальным - уникальное имя канала в пределах локальной сети.

При удачном выполнении функция **name_open()** возвращает идентификатор соединения (**coid**), а при неудачном - значение –1.

Замечание. Важно отметить следующую особенность. При выполнении клиентом функции **name_open()** ядро с помощью вызова **MsgSend()** посылает серверу служебное сообщение системного типа **_IO_CONNECT**. Сообщение имеет следующую структуру:

```
struct _io_connect {
   uint16_t            type;
   uint16_t            subtype;
   uint32_t            file_type;
   uint16_t            reply_max;
   uint16_t            entry_max;
   uint32_t            key;
   uint32_t            handle;
   uint32_t            ioflag;
   uint32_t            mode;
   uint16_t            sflag;
   uint16_t            access;
   uint16_t            zero;
   uint16_t            path_len;
   uint8_t             eflag;
   uint8_t             extra_type;
   uint16_t            extra_len;
   char                path[1];
};
```

Служебное сообщение типа **_IO_CONNECT** используется ядром в различных целях с различным содержанием его полей. В данном случае поле **type** будет содержать значение именованной системной константы **_IO_CONNECT**, а поле **subtype** будет содержать значение именованной

системной константы **_IO_CONNECT_OPEN**. Сервер должен быть готовым принять по именованному каналу такое сообщение и вызовом **MsgReply()** послать ответ **EOK**, чтобы успешно завершить соединение.

Кроме уже рассмотренных системных сообщений ядро может послать в канал, в общем случае, и другие IO-сообщения. Значения поля **type** в заголовке таких IO-сообщений находится в диапазоне **_IO_BASE < type ≤ _IO_MAX**. При получении такого IO-сообщения по именованному каналу сервер должен рассматривать их как ошибочно отправленные ядром и послать ответ **ENOSYS**, используя вызов **MsgError()**.

Из сказанного выше следует, что сервер по созданному именованному каналу должен ожидать не только прихода обычных сообщений от клиентов, но и системных сообшений от ядра импульсов, а также системное сообщение типа **_IO_CONNECT**. Поэтому сервер вынужден контролировать тип принимаемых сообщений и должным образом на них реагировать. В частности при приёме импульса серверу нет необходимости отправлять ответ, а при приёме сообщения типа **_IO_CONNECT** сервер должен ответить **EOK**. Так как обычные сообщения от клиентов будут поступать в именованный канал вместе с системными сообщениями типа **_pulse** и **_IO_CONNECT**, то для распознавания сервером клиентских и системных сообщений процессам-клиентам следует наделять свои сообщения, посылаемые в именованный канал, четырёхбайтным заголовком, аналогичным заголовку системных сообщений типа **_pulse** и **_IO_CONNECT**:

```
uint16_t        type;
uint16_t        subtype;
```

где полям **type** и **subtype** следует присвоить нулевые значения.

Приведем пример, иллюстрирующий создание сервером именованного канала и установление клиентом с ним соединения. В одном программном модуле находятся функции сервера и клиента. Процесс запускается с аргументом **-s** в режиме сервера и с аргументом **-c** в режиме клиента.

Пример:

```
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include <sys/neutrino.h>

#define NAME_CHAN   "name_chan"


typedef struct _pulse msg_header_t; //абстрактный тип для заголовка
                                    сообщения как у импульса
typedef struct _my_data {
```

```
    msg_header_t hdr;
    int data;
} my_data_t; // абстрактный тип для сообщений клиента

/*** Регистрация сервером именованного канала ***/
int server() {
   name_attach_t *attach;
   my_data_t msg;
   int rcvid;

 /* Создание именованного канала с именем "name_chan" */

 if ((attach = name_attach(NULL, NAME_CHAN, 0)) == NULL) {
    return EXIT_FAILURE;
 }

    /* Цикл ожидания поступления сообщений в именованный канал */
 while (1){
     rcvid = MsgReceive(attach->chid,&msg,sizeof(msg),NULL);
     if (rcvid == -1) {/* Ошибка, завершение */
          break;
     }

     if (rcvid == 0) {/* Получен импульс */
         switch (msg.hdr.code) {
               case _PULSE_CODE_DISCONNECT:
               /* Клиент разорвал все ранее созданные связи (вызвал
               name_close() для каждого вызова name_open() с именем
               канала) или терминировался */
                    ConnectDetach(msg.hdr.scoid);
                    /* Уничтожить соединение ядра с каналом
                    сервера*/
                    name_detach(attach, 0);
                    /* Удалить имя процесса */
                    printf("Server receive _PULSE_CODE_DISCONNECT
                    and terminated");
                    return EXIT_SUCCESS;
               case _PULSE_CODE_UNBLOCK:
               /* Клиент хочет деблокироваться (получен сигнал или
               истек таймаут). Серверу необходимо принять решение о
               посылке ответа */
                    break;
               default:
               /* Пришел импульс от какого-то процесса или от ядра
               (PULSE_CODE_COIDDEATH или _PULSE_CODE_THREADDEATH) */
                    break;
          }
          continue;// Завершить текущую итерацию цикла
     }
     /* Полученное сообщение не импульс */
     if (msg.hdr.type == _IO_CONNECT ) {/* Получено сообщение типа
         _IO_CONNECT - клиент выполнил name_open(),нужен ответ EOK
         */
```

```
            MsgReply( rcvid, EOK, NULL, 0 );
            continue; // Завершить текущую итерацию цикла
        }


        if (msg.hdr.type > _IO_BASE && msg.hdr.type <= _IO_MAX ){
        /*Получено IO-сообщение от ядра, аннулировать его */
            MsgError( rcvid, ENOSYS );
            continue; // Завершить текущую итерацию цикла
        }

        /* Получено сообщение от клиента */
        printf("Server receive %d \n", msg.data);
        MsgReply(rcvid, EOK, 0, 0);
    }
}

/*** Функция клиента ***/
int client() {
    my_data_t msg;
    int server_coid;

    if ((server_coid = name_open(NAME_CHAN, 0)) == -1) {
        return EXIT_FAILURE;
    }

    /* Заголовок сообщения клиента */
    msg.hdr.type = 0x00;
    msg.hdr.subtype = 0x00;

    /* Послать 5-ть сообщений серверу */
    for (msg.data=0; msg.data < 5; msg.data++) {
        printf("Client sending %d \n", msg.data);
        if (MsgSend(server_coid, &msg, sizeof(msg), NULL, 0) == -1)
{
            break;
        }
    }

    /* Закрыть соединение с сервером */
    name_close(server_coid);
    return EXIT_SUCCESS;
}

int main(int argc, char **argv) {
    int ret;

    if (argc < 2) {
        printf("Usage %s -s | -c \n", argv[0]);
        ret = EXIT_FAILURE;
    }
    else
         if (strcmp(argv[1], "-c") == 0) {
```

```
                printf("Running Client ... \n");
                ret = client();   /* Запуск как клиента */
          }
          else
                if (strcmp(argv[1], "-s") == 0) {
                     printf("Running Server ... \n");
                     ret = server();   /* Запуск как сервера  */
                }
                else {
                     printf("Usage %s -s | -c \n", argv[0]);
                     ret = EXIT_FAILURE;
                }
    return ret;
}
```

Использование именованных каналов в сети

Сервис глобальных имён стал возможным, начиная с QNX версии 6.3. Этот сервис обеспечивается так называемым GNS-менеджером (утилита **gns**). Используя этот сервис, процесс-сервер может создавать глобальные именованные каналы, а клиенты, используя глобальное имя канала, могут присоединяться к ним как локально, так и через QNET-сеть. Процессы могут воспользоваться сервисом глобальных имён, если они имеют привилегии администратора системы (root).

Чтобы развернуть сервис глобальных имён на том узле, где процессы-серверы будет создавать именованные каналы, необходимо на этом узле запустить утилиту **gns** в режиме сервера (**gns**-сервер):

```
# gns -s [nodename ...]
```

GNS-менеджер, запущенный в режиме сервера, выступает в роли управляющего центральной базой данных имён созданных процессами именованных каналов. На разных узлах сети процессы-серверы могут создавать одни и те же глобальные имена, что позволяет дублировать их сервис на различных узлах. При этом содержание каталога `/dev/name/global` на всех машинах будет одинаковым.

На узле, где будет работать процесс-клиент, GNS-менеджер необходимо загрузить в режиме клиента (**gns**-клиент):

```
# gns -c [nodename ...]
```

Работа **gns**-клиента заключается во взаимодействии с **gns**-сервером (серверами) для обеспечения запросов процессов-клиентов на создание или удаление соединений с глобальными именованными каналами процессов-серверов как на локальном узле, так и в сети. Если процесс-сервер находится на том же узле, что и процесс-клиент, то именно с ним устанавливается соединение. Если местный процесс-сервер по различным причинам отказал в соединении, то

осуществляется попытка установить соединение с процессом-сервером на других узлах (порядок не определён).

Федеральное агентство по образованию

**Государственное образовательное учреждение
высшего профессионального образования
"Самарский государственный аэрокосмический
университет имени академика С.П. Королёва"**

А.В. Баландин

Средства и методы разработки программных систем реального времени

Часть 3

Программирование нитей
в ОСРВ QNX Neutrino

Учебное пособие

**Самара
2012**

УДК 681.3.066
ББК 32.973.26-018.2

Баландин А.В. Средства и методы разработки программных систем реального времени. Учебное пособие: В 4 ч. Ч.3. **Программирование нитей в ОСРВ QNX Neutrino**. - Самар. гос. аэрокосм. ун-т. Самара, 2012. 83 с.

В пособии рассмотрены средства операционной системы ОСРВ QNX Neutrino (QNX 6) для программирования на языке С многонитевых (многопоточных) программных систем, функционирующих в режиме реального времени.

Пособие предназначено для студентов, обучающихся по направлению 010300.68 «Фундаментальная информатика и информационные технологии», изучающих дисциплину «Технологии промышленного программирования».

ОГЛАВЛЕНИЕ

ЧАСТЬ 3. ПРОГРАММИРОВАНИЕ НИТЕЙ

Создание нитей

При создании процесса в нем автоматически в качестве особой главной нити запускается функция `main()`. Внутри процесса, при необходимости, нить `main()` может сама запускать нити, используя другие функции программы. Любая вновь созданная нить может создать другую нить в том же самом процессе. Нить, создавшая новую нить, считается *родительской нитью*, а созданная ею нить – *дочерней нитью*. Никаких ограничений на количество нитей (кроме объема памяти) не накладывается.

Запуск функции в качестве нити осуществляется посредством специального запроса ОС и этим отличается от обычного вызова функции в языке Си.

Формирование свойств и запуск нити

Прототип функции и атрибуты нити

Функция, используемая для запуска нитей, должна иметь следующий прототип:

```
void* thread_routine(void*);
```

В качестве аргумента функция получает указатель неопределенного типа, что позволяет передавать в функцию набор предварительно подготовленных параметров любого типа (например, в виде полей структуры). Результат выполнения функции любого типа возвращается посредством указателя неопределенного типа, который при получении можно привести к требуемому типу.

При создании нити она явно или по умолчанию наделяется определенными свойствами. В качестве таких свойств выступают:

- свойство нити быть обособленной или синхронизирующей;
- параметры стека нити;
- параметры диспетчеризации нити при использовании процессора.

Для явного определения свойств запускаемой нити формируется *атрибутная запись* нити, которая должна иметь системный тип `pthread_attr_t`. Перед формированием значений атрибутной записи она должна быть проинициализирована с помощью функции:

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```

Проинициализированная атрибутная запись `attr` используется затем в функциях, предназначенных для задания соответствующих атрибутов, определяющих свойства нити.

Обособленная или синхронизирующая нить

Свойство нити быть *обособленной* или *синхронизирующей* влияет на то, смогут ли другие нити переходить в блокированное состояние, ожидая, когда данная нить завершит своё выполнение. Если нить создаётся как синхронизирующая, то у других нитей появляется возможность ждать её завершения. Если нить создается как обособленная, то такой возможности по отношению к ней у других нитей не будет. Для определения этого свойства нити используется функция:

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t* attr,
                                int detachstate);
```

Задание свойства осуществляется с помощью аргумента `detachstate`, которому присваивается значение `PTHREAD_CREATE_JOINABLE`, если нить должна быть синхронизирующей, и `PTHREAD_CREATE_DETACHED`, если – обособленной. При успешном завершении функция возвращает `EOK`.

Параметры стека нити

При определении стека нити могут быть заданы следующие параметры:

– адрес стека;

– размер стека;

– размер "области защиты" стека.

Для задания адреса стека используется функция:

```
#include <pthread.h>
int pthread_attr_setstackaddr(pthread_attr_t* attr,
                              void* stackaddr);
```

Аргумент `stackaddr` задает адрес области памяти, которая будет использоваться в качестве стека нити. Если в качестве `stackaddr` задать `NULL`, то система по умолчанию выделяет нити стек минимального размера - `PTHREAD_STACK_MIN`, достаточный для выполнения функции, которая ничего не делает, например:

```
void nothingthread(void){
 return;
}
```

Чтобы явно задать размер стека используется функция:

```
#include <pthread.h>
int pthread_attr_setstacksize(pthread_attr_t* attr,
                              size_t stacksize);
```

Аргумент `stacksize` задает размер стека в байтах. Реальный размер стека будет кратным размеру страницы памяти, величину которой можно узнать с

помощью вызова `sysconf(_SC_PAGESIZE)`. При успешном завершении обе функции возвращают `EOK`.

Область защиты – это область памяти, предназначенная для системного контроля переполнения стека, если нить использует стек, выделяемый системой по умолчанию (атрибут `stackaddr` равен `NULL`). Она создаётся непосредственно за стеком. Запись в эту область приводит к посылке ядром сигнала `SIGEVG` данной нити.

Для задания размера области защиты используется функция:

```
#include <pthread.h>
int pthread_attr_setguardsize(pthread_attr_t* attr,
                              size_t guardsize );
```

Нити будет установлена область защиты стека размером не менее `guardsize` байт. Если задать 0, то область защиты будет отсутствовать. Размер области защиты стека по умолчанию, можно узнать с помощью вызова `sysconf(_SC_PAGESIZE)`.

Если адрес стека задан явно, то вызов функции игнорируется и область защиты не выделяется системой. При успешном завершении функция возвращает `EOK`.

Приоритет и дисциплина диспетчеризации нити

Каждая созданная нить может находиться в системе в одном из следующих *состояний*:

1. Состояние *активности* – нить в данный момент выполняется системой.

2. Состояние *готовности* – нить может выполняться и ждет, когда система предоставит ей процессорное время.

3. *Блокированное* состояние – нить не может выполняться, так как её дальнейшее выполнение зависит от определённых ожидаемых условий.

Нить в состоянии готовности постоянно конкурирует с другими готовыми и с активной нитью за переход в активное состояние. Конкуренция нитей осуществляется в соответствии с приоритетом, полученным при их создании. Приоритет нити является положительным целым числом в диапазоне от 1 до 63. Правило конкуренции заключается в том, что среди готовых нитей в состоянии активности может находиться нить, имеющая наибольший приоритет. Как только нить с более высоким, чем у текущей активной нити, приоритетом становится готовой, активная нить *вытесняется* (принудительно переводится в состояние готовности), а более приоритетная нить становится активной. Если среди готовых к выполнению нитей одинаково высокий приоритет имеют несколько нитей, то активной станет нить, которая раньше других оказалась в состоянии готовности.

Доля процессорного времени, выделяемого нити, ставшей активной, для выполнения, зависит от дисциплины диспетчеризации, назначенной ей при

создании. В ОС QNX в качестве базовых дисциплин используются следующие дисциплины диспетчеризации:

1. FIFO (*First In First Out*) – нить, ставшая активной, выполняется до тех пор, пока не завершит свою работу, не будет вытеснена более приоритетной нитью или не перейдет в блокированное состояние.
2. RR (*Round Robin*) – *карусельная* диспетчеризация, при которой продолжительность нахождения нити в активном состоянии ограничивается так называемым *квантом времени* выполнения (*time slice*), после истечения которого нить вытесняется и вновь поступает в очередь готовых к выполнению нитей, а первая готовая нить становится активной и выполняется в соответствии с её дисциплиной диспетчеризации.

Квант времени дисциплины RR является задаваемым системным параметром. Величину кванта времени - `interval`, выделяемого ядром процессу с дескриптором `pid`, можно узнать, воспользовавшись функцией:

```
#include <sched.h>
int sched_rr_get_interval(pid_t pid,
                          struct timespec* interval);
```

Особенностью дисциплины FIFO является то, что теоретически активная нить может занимать процессор "вечно", если нет готовых нитей с большим приоритетом. "Деликатная нить" может периодически пытаться добровольно уступать процессор, используя функцию:

```
#include <sched.h>
int sched_yield(void);
```

Если окажется готовой другая нить с таким же приоритетом, то она станет активной. Если такой нити не окажется, то данная нить вновь станет активной и продолжит работу.

По умолчанию созданная нить наследует от родительской нити атрибуты диспетчеризации и приоритет. При необходимости эти параметры можно явно задать в атрибутной записи, выбрав значения отличные от наследуемых. Для этого необходимо отказаться от наследования, установив в атрибутной записи атрибут отмены наследования с помощью функции:

```
#include <pthread.h>
int pthread_attr_setinheritsched(pthread_attr_t* attr,
                                 int inheritsched);
```

Для отказа от наследования, в качестве значения `inheritsched` следует задать `PTHREAD_EXPLICIT_SCHED`. По умолчанию это значение равно `PTHREAD_INHERIT_SCHED`.

Чтобы явно задать дисциплину диспетчеризации следует воспользоваться функцией:

```
#include <pthread.h>
```

```
#include <sched.h>
int pthread_attr_setschedpolicy(pthread_attr_t* attr,
                                int policy);
```

Аргумент `policy` может принимать следующие значения:

`SCHED_FIFO` – для FIFO;

`SCHED_RR` – для RR;

`SCHED_OTHER` – зависит от версии ОС и может быть `SCHED_RR`.

`SCHED_NOCHANGE` – не изменять дисциплину диспетчеризации.

Для задания приоритета необходимо предварительно определить переменную системного типа `struct sched_param`, в которой значение приоритета присваивается полю с именем `sched_priority`. Например:

```
struct sched_param param;
…
param.sched_priority = 15;//Значение приоритета
…
```

Затем следует выполнить функцию:

```
#include <pthread.h>
#include <sched.h>
int pthread_attr_setschedparam(pthread_attr_t * attr,
                               const struct sched_param* param);
```

Заметим, что можно обойтись и без функции `pthread_attr_setschedparam()` при установке приоритета. Для этого достаточно выполнить присвоение приоритета, используя непосредственно атрибутную запись, например:

```
attr.param.sched_priority = 15;//Значение приоритета
```

При успешном выполнении все рассмотренные выше функции возвращают EOK.

Создание и запуск нити

Для создания и запуска нити с подготовленными атрибутами используется функция:

```
#include <pthread.h>
int pthread_create(pthread_t* thread,
                   const pthread_attr_t* attr,
                   void* (*start_routine)(void*),
                   void* arg);
```

Функция создает и запускает как нить функцию, указанную в аргументе `start_routine`, с атрибутами, заданными в атрибутной записи, указанной в аргументе `attr`. Если предполагается использовать атрибуты нити по умолчанию, то `attr` можно положить равным NULL. При запуске нити ей в

качестве аргумента функции `start_routine` передаётся указатель `arg`. Если функция без аргументов, то аргумент `arg` делают равным NULL. Аргумент `thread` указывает переменную, в которой сохраняется дескриптор (ID) созданной нити. Если необходимости в дескрипторе нити нет, можно задать NULL.

Пример:

```
/* Создание обособленной нити с дисциплиной RR и приоритетом 15*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sched.h>

void* function(void* arg)//Определение функции для нити
{
   printf("Это нить %d\n", pthread_self());
   return(0);
}

int main(void){

   pthread_attr_t attr;//Определение атрибутной записи
   struct sched_param prio;//Для задания приоритета

   pthread_attr_init(&attr);//Инициализация атрибутной записи
   /*Задать свойство обособленности*/
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
   /*Отказаться от наследования свойств диспетчеризации*/
   pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);
   /*Задать карусельную дисциплину диспетчеризации */
   pthread_attr_setschedpolicy(&attr,SCHED_RR);

   /*Задать приоритет 15*/
   prio.sched_priority = 15;
   pthread_attr_setschedparam(&attr,prio);

   /*Запустить нить*/
   pthread_create(NULL, &attr, &function, NULL);

   sleep(60);
   return EXIT_SUCCESS;
}
```

Проблема инверсии приоритетов

Инверсия приоритетов – это фундаментальная проблема приложений реального времени. Инверсия приоритетов возникает, когда в результате взаимных блокировок управление получает нить с низким приоритетом, в то время как фактически имеются готовые к выполнению нити с более высоким приоритетом. Причины возникновения инверсии приоритетов разнообразны,

рассмотрим одну из них. Пусть в некотором приложении среди прочих имеются три процесса P1, P2, P3, в которых запущены и выполняются соответственно нити *l*(5), *h*(12), *s*(20). В скобках указаны приоритеты нитей. Положим, что процесс P3 играет роль сервера, а процессы P1 и P2 являются его клиентами. Пусть нить *s* находится в RECEIVE-блокированном состоянии, ожидая сообщения от P1 и P2. Пусть нити *l* и *h* готовы к выполнению и нить *h* активна, так как имеет больший приоритет. Допустим, что нить *h* в некоторый момент времени переходит в блокированное состояние, ожидая, например, уведомления о наступлении запланированного момента времени. Нить *l* становится активной, посылает сообщение серверу P3 и переходит в SEND-блокированное состояние. Нить s выходит из RECEIVE-блокированного состояния, принимает сообщение и начинает его обслуживать. Положим, что обработка сообщения потребовала значительного времени выполнения нити *s*, в течение которого пришло уведомление нити *h* о наступлении запланированного момента времени. Нить *h* становится готовой, но не может стать активной, так как в данный момент активной является более приоритетная нить *s*, обслуживая сообщение нити *l*, приоритет которой ниже, чем у нити *h*. Это означает, что нить *l* косвенно перехватила инициативу и задерживает выполнение готовой нити с более высоким приоритетом, воспользовавшись приоритетом обсуживающей её сообщение нити *s*. Это и называется инверсией приоритетов.

Попробуем учесть подобную ситуацию и предложить механизм восстановления приоритетного использования процессора. Таким механизмом может быть механизм *наследования сервером приоритета клиентской нити*. Если применить этот механизм к рассмотренному выше примеру, то очевидно, что в этом случае при обработке сообщения от нити *l* приоритет нити *s* окажется ниже приоритета нити *h*, нить *h* вытеснит нить *s* и станет активной. Однако, как только самой нити *h* потребуется услуга сервера P3 и она пошлет ему сообщение и перейдет в SEND-блокированное состояние, нить *s* вновь станет активной и продолжит обработку сообщения от нити *l*. Более того, готовые нити других процессов рассматриваемого приложения, имеющие приоритеты выше, чем у нити *l*, но меньше, чем у нити *h*, будут вытеснять нить *s* и тем самым способствовать задержке нити *h*. Это еще одна форма проявления инверсии приоритетов.

Чтобы компенсировать нити *h* издержки инверсии приоритетов можно предложить сделать приоритет нити *s* равным наибольшему из приоритетов всех заблокированных клиентов на сервере P3. Это ускорит завершение обработки нитью *s* сообщения нити *l* (не будут мешать другие нити) и, следовательно, ускорит переход нити *s* к обработке сообщения нити *h* и выход её из блокированного состояния.

Механизм наследования серверной нитью приоритета клиентской нити реализован в QNX по умолчанию как наиболее благоприятный. Однако заметим, что этот механизм наследования не предполагает дальнейшего последовательного наследования полученного серверной нитью приоритета. Это значит, что если сервер будет причиной блокировки некоторого высокоприоритетного клиента, а

сам в свою очередь в качестве клиента будет блокирован другим сервером, то этот сервер будет уже наследовать его собственный приоритет, а не приоритет, наследованный от собственных клиентов. Тем самым он утратит полученные от клиентов преимущества.

При необходимости можно отказаться от наследования приоритета серверной нитью, установив соответствующее свойство каналу, связывающего клиентов с сервером. Для этого следует при создании сервером канала в аргументе `flags` функции `ChannelCreate()` установить флаг `_NTO_CHF_FIXED_PRIORITY`.

Наследование сервером приоритета клиента может приводить к необходимости серверной нити устанавливать свой собственный приоритет, когда она завершила обработку всех сообщений и далее выполняет собственную работу, не связанную с обработкой сообщений клиентов (если сервер сразу переходит в RECEIVE-блокированное состояние, то его приоритет не имеет значения). Для изменения приоритета используется функция:

```
#include <pthread.h>
#include <sched.h>

int pthread_setschedparam(pthread_t* tid,
                          int policy,
                          const struct sched_param* param);
```

Эта функция устанавливает нити, на которую указывает `tid`, дисциплину диспетчеризации, заданную в `policy`, и приоритет, заданный в `param`.

Методы и функции синхронизации нитей

Под синхронизацией нитей понимается согласование хода или порядка выполнения нитей друг с другом или с реальным временем. Соответственно этому QNX предлагает ряд механизмов синхронизации, учитывающих специфику различных потребностей в согласовании поведения нитей.

Присоединение

Одним из важных аспектов синхронизации нитей является согласование момента начала продолжения работы одной нити с моментом завершения выполнения другой нити. Этот вид синхронизации нитей обеспечивается механизмом, который называется *присоединением*.

Присоединение позволяет одной нити блокироваться в состоянии ожидания завершения выполнения другой нити. Для этого нить, к которой осуществляется присоединение, должна допускать возможность присоединения (т.е. должна быть запущена с атрибутом присоединяющей нити).

Для присоединения к нити с целью ожидания её завершения присоединяющаяся нить должна выполнить функцию:

```
int pthread_join(pthread_t thread,//ID-нити присоединения
                 void **value_ptr /*возвращаемое нитью
                 значение*/
                 );
```

Если значение, возвращаемое нитью при завершении, не представляет интереса, то аргумент `value_ptr` следует положить равным `NULL`.

Пример:

```
int *thread(int);//Объявление функции, запускаемой как нить

void main(void){
    int x=5;
    void *value_ret;
    pthread_t thread_id;
    …
    pthread_create(&thread_id, NULL, (void*(*)(void*))thread,
                                                    (void*)x);
    …
    /*Нить main ждет завершения нити thread*/
    pthread_join(thread_id, &value_ret);
printf("Нить thread возвратила значение: %d\n", *(int*)value_ret);
    …
}
```

Барьеры

Барьер - метод синхронизации нитей в "пространстве", основанный на создании и управлении программным объектом - барьером. Барьер создается ядром как программный объект системного типа `pthread_barrier_t`. Метод

синхронизации барьером позволяет нити перейти в состояние ожидания "у барьера" заданного числа нитей. Если нить синхронизируется барьером (достигает барьера), а количество нитей, ранее достигших барьера, меньше заданного для барьера значения, то нить приостанавливается барьером (блокируется). После того, как заданное число нитей достигает барьера, все эти нити становятся готовыми для выполнения. Такой метод синхронизации нитей удобен, когда, например, нитям необходимо "отчитаться друг перед другом" о завершении выполнения ими необходимых действий. Этот факт выражается в их "встрече у барьера".

Создание барьера заключается в определении переменной системного типа `pthread_barrier_t` и её инициализации с помощью функции:

```
int pthread_barrier_init(pthread_barrier_t *barrier,
                         const pthread_barrierattr_t *attr,
                         int count // Количество ожидаемых нитей
                         );
```

При создании барьера указывается ожидаемое количество нитей - `count`, и атрибутная переменная - `attr`, устанавливающая его свойства.

Управляя свойствами барьера, можно определить барьер как *локальный*, доступный нитям только одного процесса, или *глобальный*, доступный нитям разных процессов. Если используются свойства по умолчанию, то в качестве `attr` задается `NULL`. По умолчанию созданный в процессе барьер является локальным и может использоваться только нитями этого процесса.

Для создания глобального барьера он должен быть определён в памяти вне адресного пространства процессов (в разделяемой памяти). Кроме того, необходимо явно определить атрибутную переменную `attr` системного типа `pthread_barrierattr_t` и проинициализировать её с помощью функции:

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
```

Для придания создаваемому барьеру свойства глобальности это свойство необходимо задать в атрибутной переменной `attr`, используя функцию:

```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr
                                   int pshared);
```

Чтобы определить барьер как глобальный, аргумент `pshared` должен получить значение `PTHREAD_PROCESS_SHARED` (по умолчанию - `PTHREAD_PROCESS_PRIVAIT`).

После того, как атрибуты барьера явно определены, создается сам барьер.

Для синхронизации нити барьером она должна выполнить функцию "ожидания у барьера":

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Когда необходимость в барьере отпадает, то для освобождения системных ресурсов ядра его можно аннулировать с помощью функции:

```
int pthread_barrier_destroy(pthread_barrier_t * barrier);
```

Пример:

```
#include <pthread.h>
#include <sync.h>
#include <sys/neutrino.h>
pthread_barrier_t barrier;//Объект типа "барьер"
/**********************/
void *thread1(void* x){
    …
    pthread_barrier_wait(&barrier);//Нить thread1 ждет у барьера
    /*Нити собрались у барьера. Продолжение работы*/
    …
}
/**********************/
void *thread2(void* x){
    …
    pthread_barrier_wait(&barrier);//Нить thread2 ждет у барьера
    /*Нити собрались у барьера. Продолжение работы*/
    …
}
/**********************/
void main(void){
    /*Создать барьер со значением счетчика равным 3*/
    pthread_barrier_init(&barrier, NULL, 3);
    /*Создать нити thread1 и thread2*/
    pthread_create(NULL, NULL, thread1, NULL);
    pthread_create(NULL, NULL, thread2, NULL);
    pthread_barrier_wait(&barrier);//Нить main ждет у барьера
    /*Нити thread1, thread2 и main собрались у барьера. Продолжение
    работы всех нитей*/
    …
}
```

Мутексы

Мутекс - метод синхронизации, обеспечивающий нитям взаимное исключение по отношению к некоторому общему ресурсу, не допускающему его одновременное использование несколькими нитями. Иными словами, в каждый момент времени ресурсом может владеть не более чем одна нить.

Метод основан на создании и управлении системным программным объектом - *мутексом* (MUTual EXclusion - взаимное исключение). Управление мутексом выражается в его захвате и освобождении нитями. Если некоторая нить пытается захватить уже захваченный другой нитью мутекс, то она блокируется до момента его освобождения предыдущей нитью.

Мутекс определяется как программный объект системного типа `pthread_mutex_t` с заданными свойствами. А именно, мутекс может быть локальным или глобальным. Локальный мутекс доступен нитям только одного процесса, которому они принадлежат. Глобальный мутекс позволяет синхронизировать нити, находящиеся в разных процессах. Кроме того, можно выбирать свойства мутекса для управления последствиями инверсии приоритетов нитей, когда некоторая нить, захватившая мутекс, является причиной блокировки более приоритетных нитей, обратившихся к этому же мутексу.

Создание мутекса

Для создания мутекса необходимо предварительно задать его свойства в атрибутной переменной системного типа `pthread_mutexattr_t` и затем переменную системного типа `pthread_mutex_t` проинициализировать с помощью функции:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutexattr_t *attr);
```

Если аргумент `attr` получает значение `NULL`, то свойства мутекса устанавливаются по умолчанию.

Свойства мутекса

Явно свойства мутекса задаются с помощью атрибутной переменной системного типа `pthread_mutexattr_t`, которую необходимо определить и затем проинициализировать с помощью функции:

```
int pthread_mutexattr_init(const pthread_mutexattr_t *attr);
```

По умолчанию мутекс является локальным. Для создания глобального мутекса он должен находиться в разделяемой процессами памяти. Свойство глобальности мутекса устанавливается в атрибутной переменной с помощью функции:

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                 int pshared);
```

Аргумент `pshared` должен получить значение `PTHREAD_PROCESS_SHARED` (значение по умолчанию - `PTHREAD_PROCESS_PRIVATE`).

По умолчанию для борьбы с инверсией приоритетов мутекс наделяется таким свойством, что ядро повышает приоритет низкоприоритетной нити, блокировавшей высокоприоритетные нити, захватив один или более мутексов с этим свойством, до уровня максимального приоритета среди приоритетов нитей, заблокированных на этих мутексах. Это способствует более быстрому

освобождению мутекса, захваченного низкоприоритетной нитью, так как она становится способной конкурировать за процессор, получив более высокий приоритет. Явно можно задать другой порядок управления приоритетом нити, захватившей мутекс. А именно, можно в качестве свойства явно задать мутексу значение приоритета, с которым будет выполняться захватившая её нить. Это свойство мутекса устанавливаются в атрибутной переменной с помощью функции:

```
int pthread_mutexattr_setprotocol(pthread_mutexattr *attr,int
                                  protocol);
```

Аргументу protocol следует установить значение `PTHREAD_PRIO_PROTECT` (по умолчанию - `PTHREAD_PRIO_INHERIT`). Тогда нить будет выполняться с приоритетом, равным наивысшему приоритету из всех приоритетов, установленных для мутексов, захваченных этой нитью и имеющих атрибут `PTHREAD_PRIO_PROTECT`, независимо от того, блокированы ли на них другие нити или нет. Значение приоритета, связываемого с мутексом, устанавливается в атрибутной записи с помощью функции:

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
```

Приоритет задаётся значением аргумента `prioceiling`.

Захват мутекса

Для синхронизации нити мутексом она должна выполнить функцию захвата мутекса:

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Функция либо блокирует выполнение нити, если мутекс ранее уже был захвачен другой нитью, либо осуществляется захват мутекса нитью.

Осторожный захват мутекса

Если блокирование нити при попытке захвата мутекса не допустимо, то следует воспользоваться функцией:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Если мутекс свободен, то он будет захвачен. В противном случае - нить продолжит свое выполнение. Для анализа результата попытки захвата мутекса нитью необходимо контролировать возвращаемое функцией значение:

`EOK` – успешный захват мутекса;

`EBUSY` – мутекс уже захвачен другой нитью;

`EINVAL` – недопустимый мутекс;

`EAGAIN` – недостаточно ресурсов системы для реализации запроса на захват мутекса.

Освобождение мутекса

Если нить, захватившая мутекс, завершает свой исключительный доступ к общему ресурсу, то она должна освободить мутекс с помощью функции:

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Уничтожение мутекса

Если необходимость в мутексе отпадает, то для освобождения ядром системных ресурсов его целесообразно уничтожить с помощью функции:

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Пример:

```
//Создание мутекса
pthread_mutex_t  mutex;//Определение переменной для управления
                                  мутексом
…
pthread_mutex_init (&mutex,NULL);//Инициализация мутекса с
                                          атрибутами по умолчанию
…
/*
Захват мутекса, если он уже захвачен, то ядро переводит нить в
состояние ожидания освобождения мутекса
*/
pthread_mutex_lock (&mutex);
/*
Исключительный доступ к ресурсу, контролируемому мутексом
…
*/
/* Завершение доступа к ресурсу, освобождение мутекса */
pthread_mutex_unlock (&mutex);
…
/* Уничтожение мутекса */
pthread_mutex_destroy (&mutex);
…
```

Создание мутекса можно отложить до его первого использования. Для этого при определении переменной управления мутексом ей необходимо присвоить начальное значение равное системной константе `PTHREAD_MUTEX_INITIALIZER`. Это указание ядру, что при первом использовании мутекса его необходимо предварительно создать.

Пример:

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
```

```
/*
Мутекс помечен как требующий создания
*/
…
/* Первый захват мутекса. Ядро предварительно его создаст */
pthread_mutex_lock (&mutex);
…
```

Создание рекурсивного мутекса

В некоторых случаях требуется определять мутекс как рекурсивный (считающий). Это необходимо, когда выполнение нити, захватившей мутекс, предполагает, например, вызов различных функций, которые в свою очередь в процессе выполнения осуществляют захват этого же мутекса. Если не допускать повторного захвата мутекса одной и той же нитью, то возникнет "мертвая" блокировка. Рекурсивный мутекс позволяет избежать этого. Повторный захват одной и той же нитью рекурсивного мутекса не приводит к блокированию нити. При этом ядро контролирует количество захватов и освобождений рекурсивного мутекса.

Чтобы мутекс был рекурсивным, переменную управления мутексом при определении необходимо явно проинициализировать значением PTHREAD_RMUTEX_INITIALIZER.

Пример:

```
pthread_mutex_t
func_mutex=PTHREAD_RMUTEX_INITIALIZER;/*Помечен как рекурсивный*/
…
high_level_func(){
    pthread_mutex_lock (&func_mutex);
    //Критическая секция
    …
    low_level_func();
    …
    //Конец критической секции
    pthread_mutex_unlock (&func_mutex);
}
low_level_func(){
    pthread_mutex_lock (&func_mutex);
    //Критическая секция
    …
    //Конец критической секции
```

```
    pthread_mutex_unlock (&func_mutex);
}
```

Блокировки чтения/записи

Блокировка чтения/записи - метод синхронизации, согласующий поведение нитей по отношению к содержимому общей области памяти, требуемой одновременно нескольким нитям для чтения или записи. При этом нити-читатели могут одновременно читать содержимое памяти, исключая при этом доступ к ней нитей-писателей, или одна нить-писатель может изменять содержимое памяти, исключая при этом доступ к ней нитей-читателей и других нитей-писателей.

Блокировка чтения/записи определяется как программный объект системного типа `pthread_rwlock_t` с заданными свойствами. А именно, блокировка чтения/записи может быть локальной или глобальной. Локальная блокировка чтения/записи доступна нитям только одного процесса, которому они принадлежат. Глобальная блокировка чтения/записи позволяет синхронизировать нити, находящиеся в разных процессах.

Создание блокировки чтения/записи

Для создания блокировка чтения/записи необходимо предварительно задать его свойства в атрибутной переменной системного типа `pthread_rwlockattr_t` и затем переменную системного типа `pthread_rwlock_t` проинициализировать с помощью функции:

```
int pthread_rwlock_init (pthread_rwlock_t *rwlock,
                         const pthread_rwlockattr_t *attr);
```

Если аргумент `attr` получает значение `NULL`, то свойства устанавливаются по умолчанию.

Свойства блокировки чтения/записи

Явно свойства блокировки чтения/записи устанавливаются с помощью атрибутной переменной системного типа `pthread_rwlockattr_t`, которую необходимо определить и затем проинициализировать с помощью функции:

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

По умолчанию блокировка чтения/записи является локальной. Для создания глобальной блокировки чтения/записи она должна находиться в разделяемой процессами памяти. Свойство глобальной блокировки чтения/записи устанавливается в атрибутной переменной с помощью функции:

```
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared);
```

Аргумент `pshared` должен получить значение `PTHREAD_PROCESS_SHARED` (значение по умолчанию - `PTHREAD_PROCESS_PRIVATE`). Заметим, однако, что эта возможность поддерживается только в том случае, если в заголовочном файле

unistd.h определена системная константа _POSIX_THREAD_PROCESS_SHARED.

Для проверки текущего свойства блокировки чтения/записи используется функция:

```
int pthread_rwlockattr_getpshared(pthread_rwlockattr_t *attr,
                                  int *valptr);
```

Значение свойства разделяемости блокировки чтения/записи помещается в переменную целого типа, на которую указывает valptr.

Аннулировать атрибутную запись attr можно с помощью функции:

```
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Захват блокировки чтения/записи

Для синхронизации нити-читателя блокировкой чтения/записи она должна выполнить функцию захвата блокировки чтения/записи для чтения:

```
int pthread_rwlock_rdlock (pthread_rwlock_t *rwlock);
```

Функция либо блокирует выполнение нити-читателя, если блокировка чтения/записи ранее уже была захвачена другой нитью-писателем, либо осуществляется захват блокировки чтения/записи нитью.

Для синхронизации нити-писателя блокировкой чтения/записи она должна выполнить функцию захвата блокировки чтения/записи для записи:

```
int pthread_rwlock_wrlock (pthread_rwlock_t *rwlock);
```

Функция либо блокирует выполнение нити-писателя, если блокировка чтения/записи ранее уже была захвачена другой нитью-писателем или нитью-читателем, либо осуществляется захват блокировки чтения/записи нитью-писателем.

Осторожный захват блокировки чтения/записи

Если блокирование нити при попытке захвата блокировки чтения/записи не допустимо, то нити-читателю следует воспользоваться функцией:

```
int pthread_rwlock_tryrdlock (pthread_rwlock_t *rwlock);
```

а нити-писателю следует воспользоваться функцией:

```
int pthread_rwlock_trywrlock (pthread_rwlock_t *rwlock);
```

При этом необходимо контролировать значения, возвращаемые функциями pthread_rwlock_trywrlock() и pthread_rwlock_tryrdlock(). Если блокировка чтения/записи свободна, то она будет захвачена нитью. В противном случае - нить получит отказ захвата и продолжит свое выполнение. В этом случае для анализа ситуации нити необходимо контролировать возвращаемое функциями значение:

EOK – успешный захват блокировки чтения/записи;

EBUSY – отказ, блокировка чтения/записи ранее уже захвачена.

Освобождение блокировки чтения/записи

Если нить, захватившая блокировку чтения/записи, завершает свой доступ к общему ресурсу, то она должна освободить блокировку чтения/записи с помощью функции:

```
int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);
```

Уничтожение блокировки чтения/записи

Если необходимость в блокировке чтения/записи отпадает, то для освобождения ядром системных ресурсов её целесообразно уничтожить с помощью функции:

```
int pthread_rwlock_destroy (pthread_rwlock_t *rwlock);
```

Все функции в случае успешного завершения возвращают EOK, а в случае ошибки - положительное значение (код ошибки):

EINVAL – недопустимая блокировка чтения/записи;

EAGAIN – недостаточно ресурсов системы для захвата блокировки чтения/записи.

Условные переменные

В ряде случаев захват ресурса не предполагает безусловную готовность ресурса к его обработке захватившей нитью. Нить предварительно должна проверить условие готовности ресурса к обработке. Если условие выполняется, то обработка осуществляется и при её завершении ресурс освобождается. Если условие не выполняется, то нить вынуждена освобождать ресурс, чтобы предоставить возможность доступа к нему другим нитям, которые, в общем случае, могут влиять на формирование результата проверяемого условия. Например, нить должна дождаться, когда ***X*** станет равным ***Y*** прежде, чем продолжит своё выполнение. Если использовать только мутексы, то получим следующее решение:

```
pthread_mutex_t condMutex;
//Создание мутекса
pthread_mutex_init (&condMutex,NULL);
…
pthread_mutex_lock(&condMutex);
while(x!=y){
    pthread_mutex_unlock(&condMutex);
//Немного подождать, чтобы позволить изменить значения x, y
    pthread_mutex_lock(&condMutex);
}
//Выполнить работу
pthread_mutex_unlock(&condMutex);
```

…

Очевидно, что в этом случае проверка условия (опрос) осуществляется явно, а хотелось бы не заниматься опросом для выяснения этого события. Применение метода условной переменной позволяет избежать опроса.

Разумно нити вновь пытаться захватывать ресурс только в том случае, если произошли какие-то события, могущие повлиять на условие готовности ресурса к обработке данной нитью. А до этого находиться в состоянии ожидания таких событий. Метод условной переменной как раз и предполагает реализацию такого механизма синхронизации (входит в стандарт POSIX). Метод основан на управлении мутексом и программным объектом системного типа `pthread_cond_t`, называемым *условной переменной*. Метод позволяет одним нитям ждать уведомления, посылаемые другими нитями, использующими в качестве посредника одну и ту же условную переменную.

Функции метода следующие:

```
/*Создание условной переменной*/
int pthread_cond_init (pthread_cond_t *condvar, pthread_condattr_t *
                       attr);
/*Ожидание уведомления по условной переменной*/
int pthread_cond_wait (pthread_cond_t * condvar, pthread_mutex_t
                       *mutex);
/* Отправка уведомления по условной переменной */
int pthread_cond_broadcast (pthread_cond_t *condvar);
int pthread_cond_signal (pthread_cond_t *condvar);
```

В отличие от безусловного использования мутекса метод условной переменной позволяет нити переходить в состояние ожидании уведомлений без необходимости явного освобождения нитью захваченного мутекса. При этом ядро неявно освобождает мутекс, когда нить переходит в состояние ожидания уведомления, и захватывает мутекс, когда выводит нить из состояния ожидания, поддерживая для нити иллюзию её владения мутексом и, следовательно, ресурсом. Из состояния ожидания нить выводится, если какая-то другая нить пошлет уведомление по условной переменной, выполнив функцию `pthread_cond_signal()` или `pthread_cond_broadcast()`. Отличие этих функций в том, что в случае отправки уведомления функцией `pthread_cond_signal()`, будет выведена из состояния ожидания только одна из возможно нескольких ждущих уведомления нитей. Это будет нить с максимальным приоритетом и максимальным временем ожидания. Остальные нити будут продолжать ждать следующего уведомления. Если же для уведомления используется функция `pthread_cond_broadcast()`, то уведомление воздействует на все нити, ожидающие его по соответствующей условной переменной, выводя их из состояния ожидания. Порядок, в котором

ядро восстанавливает для них исключительный доступ к ресурсу, определяется приоритетами нитей, а если приоритет одинаковый, то временем ожидания.

Выведенная из состояния ожидания нить должна вновь проверить возможность использования ресурса и затем выполняет работу или обратно возвращается в состояние ожидания уведомления.

Отметим, что между мутексом и условной переменной нет прямой зависимости. Одна и та же условная переменная может использоваться для получения нитью уведомления при захвате различных мутексов. Кроме того, можно анализировать состояние одного и того же ресурса (один мутекс), используя поочередно различные условные переменные.

Рассмотренный выше пример, при использовании метода условной переменной, примет следующий вид:

```
…
pthread_mutex_t condMutex;
pthread_cond_t condvar;
…
//**********************
//Нить 1
//**********************
pthread_mutex_lock(&condMutex);
while(x!=y){
//Ждать, пока нить 2 или 3 не изменят значения x или y
    pthread_cond_wait (&condvar, &condMutex);
}
//Выполнить работу
pthread_mutex_unlock(&condMutex);
…
}
//**********************
//Нить 2
//**********************
{
…
pthread_mutex_lock(&condMutex);
//Модифицировать значение x
pthread_cond_signal (&condvar);
pthread_mutex_unlock(&condMutex);
…
```

```
}
//*********************
//Нить 3
//*********************
{
…
pthread_mutex_lock(&condMutex);
//Модифицировать значение y
pthread_cond_signal (&condvar);
pthread_mutex_unlock(&condMutex);
…
}
//*********************
void main(void){
//Инициирование мутекса
pthread_mutex_init (&condMutex,NULL);
//Инициирование условной переменной
pthread_cond_init (&condvar, NULL);
//Создание нитей 1 и 2
…
}
```

Условная переменная, инициированная по умолчанию, не может разделяться нитями различных процессов. Однако с помощью явной установки атрибутов `pthread_condattr_t *attr` можно сделать условную переменную разделяемой. Параметр `attr` управляется с помощью функций:

```
int pthread_condattr_init(pthread_ condattr_t *attr);
int pthread_condattr_destroy(pthread_ condattr_t *attr);
int pthread_condattr_getpshared(pthread_ condattr_t*att,
                                int *valptr);
int pthread_condattr_setpshared(pthread_ condattr_t,
                                int value);
```

Эти функции позволяют создать и аннулировать атрибутную запись `attr`, получить ткущее значение атрибута в виде целого, на которое указывает `valptr`, и установить значение атрибута равное значению `value`. Значение `value` может быть либо `PTHREAD_PROCESS_PRIVATE`, либо `PTHREAD_PROCESS_SHARED`. Последнее значение позволяет совместное использование процессами мутекса и/или условной переменной, расположенных в разделяемой процессами области памяти.

Ждущие блокировки

Этот метод является частным случаем метода условной переменной и не относится к стандарту POSIX. Метод использует виртуальный объект `sleepon` - "*ждущая блокировка*", и основан на неявном использовании мутекса и условной переменной, в роли которой может выступить любая ячейка памяти, адрес которой доступен нитям. При этом заимствуется только адрес этой ячейки для согласования ожидающих и уведомляющих нитей, а содержимое ячейки не затрагивается.

Функции управления ждущей блокировкой следующие:

```
/*Захват ждущей блокировки*/
    int pthread_sleepon_lock (void);
/*Освобождение ждущей блокировки*/
    int pthread_sleepon_unlock (void);
/*Ожидание уведомления*/
    int pthread_sleepon_wait (const volatile void *addr);
/*Отправка уведомления*/
    int pthread_sleepon_broadcast(const volatile void *addr);
    int pthread_sleepon_signal (const volatile void *addr);
```

Для доступа к ресурсу нить должна захватить ждущую блокировку. При успешном захвате, если условие доступа к ресурсу не выполняется, то нить может перейти в состояние ожидания уведомления по указанной условной переменной, задав её адрес в вызове `pthread_sleepon_wait()`. Другие нити могут посылать уведомления о наступлении события, используя функции `pthread_sleepon_signal()` или `pthread_sleepon_broadcast()`. Отличие этих функций в том, что в случае `pthread_sleepon_signal()` будет разблокирована только одна из ждущих уведомления по указанному адресу нитей с наивысшим приоритетом. Если приоритет одинаковый, то порядок выбора активизируемой нити *не определен!* А в случае использования функции `pthread_sleepon_broadcast()` будут разблокированы все ожидающие нити. При этом ядро затем обеспечивает для них корректный порядок продолжения выполнения и исключительного доступа к ресурсу с учетом их приоритета и длительности ожидания.

Пример:

```
/* Нить - потребитель данных, поставляемых устройством */
volatile int data_ready=0; //флаг готовности данных в устройстве
Consumer(){
     while(1){
        pthread_sleepon_lock();
        while(!data_ready){
```

```
            pthread_sleepon_wait(&data_ready);
            }
            //Обработать данные
            data_ready=0;
            pthread_sleepon_unlock();
      }
/*  Нить - информирующая о событии возникновения данных в
     устройстве */
Producer(){
      while(1){
         //ждать прерывания от оборудования …
         pthread_sleepon_lock();
         data_ready=1;
         pthread_sleepon_signal(&data_ready);
         pthread_sleepon_unlock();
      }
```

Замечание. Метод обеспечивается "родными" функциями QNX/Neutrino 2, не предусмотренными стандартом POSIX. Их использование ограничивает мобильность приложений.

Семафоры

Семафоры - это метод синхронизации, основанный на создании и управлении программным объектом системного типа `sem_t`, регулирующим количество нитей, осуществляющих одновременный доступ к некоторому ресурсу. Управление семафором выражается в его захвате и освобождении. Семафор ведет счетчик захватов. Начальное значение счетчика захватов семафора устанавливается при его создании. Если нить пытается захватить семафор, счетчик которого не больше 0, то она блокируется до момента, когда значение счетчика не станет больше 0. Семафоры бывают *неименованные* и *именованные*.

Неименованный семафор

Для создания неименованного семафора необходимо предварительно определить переменную типа `sem_t` и затем выполнить функцию инициализации неименованного семафора, используя указатель этой переменной:

```
#include <semaphore.h>
int sem_init(sem_t * sem, int pshared, unsigned value);
```

Если аргумент `pshared` отличен от нуля, то семафор может разделяться нитями различных процессов, если семафор создан в разделяемой процессами памяти. С помощью аргумента `value` задается начальное значение счетчика семафора. После инициализации семафора указатель `sem` используется в функциях управления семафором.

Для аннулирования неименованного семафора используется функция

```
int sem_destroy(sem_t *sem);
```

При аннулировании семафора освобождаются соответствующие системные ресурсы ядра.

Именованные семафоры

Именованный семафор создается как файл особого типа, регистрируемый в файловой системе в каталоге `/dev/sem`. Для создания именованного семафора необходимо предварительно определить переменную-указатель именованного семафора типа `sem_t*`, и затем выполнить функцию открытия именованного семафора - `sem_open()`, используя эту переменную для получения значения указателя семафора, возвращаемого этой функцией. Функция `sem_open()` позволяет создать в файловой системе новый именованный семафор и открыть к нему доступ, если семафор с указанным именем в системе отсутствует, либо только открыть доступ к существующему семафору с указанным именем. Поэтому функция `sem_open()` является функцией с переменным числом аргументов. Если требуется открыть доступ к именованному семафору, а в случае его отсутствия создать новый семафор с указанным именем, выполняется вызов функции:

```
sem_t* sem_open(const char * sem_name, int oflags, mode_t mode,
                unsigned int value);
```

Если требуется только открыть уже существующий именованный семафор, а в случае его отсутствия создавать новый семафор не требуется, функция используется без последних двух аргументов:

```
sem_t* sem_open(const char * sem_name, int oflags);
```

Функция `sem_open()` создает и/или открывает в каталоге `/dev/sem` именованный семафор, возвращая дескриптор управления семафором. Значение аргумента `sem_name` должно начинаться с символа </>, например, `/myprog.sem1`. Имена семафоров должны быть меньше чем `(NAME_MAX - 8)` символов. Например, семафор с именем `/myprog.sem1` в файловой системе будет учитываться как `/dev/sem/myprog.sem1`. QNX позволяет использовать в именах семафоров более одного символа </>. Однако для поддержания POSIX-мобильности не рекомендуется использовать в именах семафоров более одного символа </>.

Аргумент `oflags` управляет возможностью создания нового семафора. Если при отсутствии семафора предполагается создание нового семафора, в `oflags` должен быть установлен флаг `O_CREAT`. Если нет - `(O_CREAT|O_EXCL)`.

`O_CREAT` – указывает на возможность создания нового именованного семафора. Если именованный семафор с указанным именем уже существует, то к нему будет открыт доступ. Если такого семафора нет, то он будет создан. При создании нового именованного семафора будут использованы значения

аргументов mode и value. Аргумент mode указывает режимы доступа к семафору (точно так же как при создании файла), а value задает начальное значение семафора (не должно превышать SEM_VALUE_MAX). Значение value > 0 делает семафор открытым, а значение равное 0 означает, что создаваемый семафор будет закрытым. Для мобильности в mode, необходимо установить флаги доступа для чтения, записи и выполнения, используя константы из <sys/stat.h>: S_IRWXG - доступ для группы, S_IRWXO - доступ для других, S_IRWXU - доступ для владельца.

(O_CREAT|O_EXCL) –открывается доступ только к существующему семафору с указанным именем. Если такой семафор не существует, то функция сообщает об ошибке и возвращает в системной переменой errno номер ошибки EEXIST.

Заметим, что не требуется устанавливать в oflags флаги O_RDONLY, O_RDWR, или O_WRONLY. Поведение семафора с этими флагами не определено. Функции QNX игнорируют эти флаги, но их использование может понизить мобильность программного кода.

Функция возвращает указатель на семафор или -1 в случае неудачи, а в системной переменной errno устанавливается номер ошибки.

Доступ к именованному семафору можно закрыть, используя функцию

```
int sem_close(sem_t * sem);
```

Если требуется аннулировать именованный семафор, то используется функция

```
int sem_unlink(const char * sem_name);
```

Функция sem_unlink() уничтожает открытые именованные семафоры таким же образом, как удаляются открытые файлы. То есть, процессы, которые имеют открытый семафор, могут все еще использовать его, но семафор исчезнет, как только последний процесс использует функцию sem_close(), чтобы закрыть доступ к семафору. Попытка выполнить sem_open() по отношению к уничтоженному семафору будет рассматриваться как создание нового семафора.
Семафоры сохраняются в системе не зависимо от создавших их процессов и существуют в ней пока не будут явно уничтожены или система не завершит свою работу.

Управление семафорами

Управление неименованными и именованными семафорами осуществляется с помощью одних и тех же функций:

```
#include <time.h>
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
```

```
int sem_timedwait (sem_t * sem,
                   const struct timespec * abs_timeout);
int sem_post(sem_t * sem);
int sem_getvalue(sem_t *sem, int *value);
```

Функция `sem_wait()` уменьшает значение счетчика семафора на 1. Если при этом значение семафора не большее чем ноль, то вызвавшая функцию нить блокируется до тех пор, пока не возникнет возможность уменьшить счетчик или запрос не будет завершен в связи с приходом сигнала. Предполагается, что в какой-то момент времени некоторая нить, вызовет функцию `sem_post()`, чтобы увеличить счетчик семафора.

Функция `sem_trywait()` уменьшает значение счетчика семафора на 1, если его значение большее ноля, и возвращает значение 0. В противном случае счетчик семафора не изменяется, а функция завершается, возвращая значение -1.

Функция `sem_timedwait()` захватывает семафор `sem`, как и функция `sem_wait()`. Однако, если семафор не может быть захвачен, то ожидание завершается, когда указанное время ожидания истекает. Время ожидания истекает, когда проходит момент абсолютного времени, указанный в `abs_timeout`. При этом время измеряется системными часами, на которых базируются таймауты (то есть когда значение тех часов равняется или превышает `abs_timeout`), или если абсолютное время, указанное `abs_timeout`, уже прошло во время реализации запроса. Если выбор таймеров поддерживается, то указание ожидаемого момента времени базируется на часах реального времени `CLOCK_REALTIME`. Если выбор таймеров не поддерживается, то указание ожидаемого момента времени базируется на системных часах, время которых возвращается функцией `time()`.

Функция `sem_timedwait()` уменьшает значение счетчика семафора на 1, если его значение большее ноля, и возвращает значение 0. В противном случае счетчик семафора не изменяется, а функция завершается, возвращая значение -1, а в `errno` устанавливается код ошибки `ETIMEDOUT` – прошёл указанный абсолютный момент времени.

Функция `sem_post()` увеличивает счетчик семафора `sem` на 1. Если имеются нити, которые в настоящее время блокированы, ожидая семафор, то одна из этих нитей возвратится успешно из вызова `sem_wait`. Нить, которая будет разблокирована первой, определяется в соответствии с приоритетом и временем ожидания (с наибольшим приоритетом, которая ждала дольше всех). Функция `sem_post()` может быть вызвана обработчиком сигналов.

Функция `sem_getvalue()` позволяет определить текущее значение счетчика семафора `sem`, которое заносится по адресу, заданному в `value`.

В случае успеха все функции возвращают значение 0. В противном случае возвращается -1 и в `errno` устанавливается код ошибки.

Именованные семафоры работают медленнее, чем неименованные семафоры. Но зато можно открывать доступ к именованному семафору, как к файлу, используя его символическое имя. Поэтому доступ к именованному семафору может быть открыт нитями разных и распределенных процессов.

Примеры:

```
#include <stdio.h>
#include <semaphore.h>
#include <time.h>

main(){

    struct timespec tm;
    sem_t sem;
    int i=0;

    sem_init(&sem,0,0);//Семафор не разделяем процессами, счетчик = 0

    do {
        clock_gettime(CLOCK_REALTIME, &tm);//Текущее время
        tm.tv_sec += 1;//Увеличить на 1 сек
        i++;
        printf("i=%d\n",i);
        if(i==10) sem_post(&sem);//Увеличить счетчик семафора на 1
    } while (sem_timedwait(&sem, &tm) == -1);

    printf("Семафор захвачен после %d таймаутов\n", i);
    return;
}
```

Управление памятью вне адресного пространства процессов

Нити одного процесса в QNX не могут осуществлять прямой доступ к памяти, находящейся за пределами локального адресного пространства процесса (локальной памяти). Такие области памяти контролируются только операционной системой и называются *системными областями памяти* или *системной памятью*. К системной памяти относят:

- адреса памяти, связанные с физическими устройствами;
- область оперативной памяти за пределами локальной памяти процессов (разделяемая память).

Если процессам требуется доступ к адресам физических устройств или возникает потребность в оперативной памяти за пределами локальной памяти процесса (например, разделяемой различными процессами), то операционная система предоставляет процессам такую возможность посредством *механизма отображения адресов* системных областей памяти в адресное пространство процесса. Отображение адресов системных областей памяти реализуется процессами с помощью специальных запросов к операционной системе, в результате успешного выполнения которых между адресами системной памяти и выделенными адресами адресного пространства процесса устанавливается взаимнооднозначное соответствие. В результате обращение процесса к этим адресам памяти преобразуется операционной системой в обращение к соответствующим адресам системной памяти.

Области адресов, связанные с физическими устройствами, однозначно определены архитектурой вычислительной платформы и заранее известны. Области же оперативной памяти прежде чем быть отображенными в память процесса должны быть предварительно выделены операционной системой из общего объёма системной оперативной памяти. При этом они специфицируются как *именованные области оперативной памяти* или коротко - *именованная память*. Создаваемая именованная память регистрируется в файловой системе ОС как файл устройства специального типа в каталоге `/dev/shmem`. Для доступа к именованной памяти, зарегистрированной в файловой системе, процесс должен предварительно присоединить её к себе, получив в результате дескриптор присоединенной именованной памяти.

Создание именованной памяти

Для создания и/или присоединения процессом именованной памяти используется функция:

```
#include <fcntl.h>
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

Функция `shm_open()` создает и/или присоединяет к процессу существующую именованную память с указанным именем и возвращает дескриптор именованной памяти как дескриптор файла с установленным флагом

FD_CLOEXEC. Такие дескрипторы не наследуются дочерними процессами (см. функцию fcntl()).

Аргумент name содержит символьную строку с именем именованной памяти. Имя памяти должно начинаться с символа слеш </> и содержать только один слеш. Например, "/sharemap". Длина имени не должна превышать значения NAME_MAX.

Режим доступа к содержимому именованной памяти определяется значениями флагов, указанных в аргументе oflag. Значение oflag формируется операцией поразрядного логического сложения следующих флагов, которые определены в <fcntl.h>:

O_RDONLY - открыть только для чтения.

O_RDWR - открыть для чтения и записи.

O_CREAT - создание новой и/или присоединение к существующей именованной памяти. Если указанная именованная память уже существует, то осуществляется только присоединение существующей именованной памяти с указанным именем к процессу. Иначе, именованная память создается с правами доступа владельцев, установленными в соответствии со значением аргумента mode и маской прав доступа, назначенных процессу (атрибут процесса) для создания файлов. В результате именованная память присоединяется к процессу с режимами доступа, определенными значениями флагов, указанных в аргументе oflag.

O_CREAT|O_EXCL – создание новой именованной памяти. Если именованная память с указанным именем в файловой системе не зарегистрирована, то будет создана и зарегистрирована в файловой системе новая именованная память. Если именованная память с указанным именем существует, то shm_open() возвращает ошибку. Проверка существования именованной памяти и её создание, если она не существует, является атомарной операцией по отношению к другим процессам, также выполняющим shm_open(), указывая ту же самую именованную память с набором флагов O_CREAT|O_EXCL. Это обеспечивает корректность одновременного выполнения функции shm_open() несколькими процессами.

O_TRUNC - если именованная память существует, и она успешно присоединена с флагом O_RDWR, то память урезается до нулевой длины, а права доступа владельца и владелец не изменяются. Размер памяти может затем быть установлен с помощью функции ftruncate().

Аргумент mode используется процессом для формирования прав доступа пользователей к создаваемой именованной памяти таким же образом, как при создании обычного файла, с учётом установленной процессу маски, ограничивающей его возможности по формированию прав доступа (см. описание функции umask()).

При успешном выполнении функция возвращает неотрицательное целое положительное число, которое является дескриптором именованной памяти. В

случае ошибки возвращается -1 и в errno заносится код ошибки. Если при этом произошло создание именованной памяти, то в каталоге /dev/shmem появиться файл с именем памяти.

Заметим, что вновь созданная именованная память имеет нулевую длину. Размер памяти после её создания следует установить с помощью функции ftruncate().

Если необходимость в доступе к именованной памяти у процесса отпадает, он может отсоединиться от неё, выполнив обычную функцию закрытия файла close(). Созданная именованная память, включая содержимое, хранится в файловой системе до тех пор, пока она не будет удалена функцией

```
#include <sys/mman.h>
int shm_unlink(const char *name);
```

При успешном выполнении функция возвращает значение 0, в случае ошибки - значение -1 и заносит код ошибки в системную переменную errno.

Реальное удаление будет отложено операционной системой до момента, когда именованная память не будет присоединена ни к одному процессу.

Организация доступа к именованной памяти

После того, как именованная память будет присоединена к процессу, необходимо отобразить её в адресное пространство процесса. Это осуществляется с помощью функции

```
#include <sys/mman.h>
void * mmap(void *addr,
            size_t len,
            int prot,
            int flags,
            int fd,
            off_t off);
```

В результате выполнения функции в адресное пространство процесса отображается выделенная в пределах именованной памяти с дескриптором fd область, начинающаяся со смещением off от её начала и длиною len байтов. Доступ к этой области именованной памяти в адресном пространстве процесса начинается с адреса, возвращённого функцией mmap(). Аргумент addr – указывает желательное место в адресном пространстве процесса, куда именованная память должна быть отображена. Обычно нет необходимости указывать какое-то конкретное значение addr, можно просто задать NULL. Если устанавливается не NULL , то будет ли объект памяти отображен, зависит от того, установлен ли флаг MAP_FIXED в аргументе flags. Если MAP_FIXED установлен, то область именованной памяти отображается с адреса addr, или функция завершается с ошибкой. Если MAP_FIXED не установлен, то значение addr рассматривается как желаемый адрес начала области доступа к

именованной памяти в адресном пространстве процесса, но не обязательный и заданное значение addr может быть проигнорировано.

Аргумент prot определяет порядок использования именованной памяти. Можно устанавливать следующие флаги использования (определены в <sys/mman.h>):

PROT_EXEC – память может быть использована для размещения выполняемых модулей.

PROT_NOCACHE - запрещает кэширование памяти (например может использоваться, чтобы обратиться к двухпортовой памяти)

PROT_NONE – запрещает какой-либо доступ к памяти.

PROT_READ – разрешает доступ к памяти для чтения.

PROT_WRITE – разрешает доступ к памяти для записи.

Аргумент flags определяет дополнительные свойства отображения именованной памяти, путем установки следующих флагов:

MAP_PRIVATE – отображаемая область памяти не может использоваться другими процессами.

MAP_SHARED – отображаемая область памяти может разделяться с другими процессами.

Функция возвращает начальный адрес отображения именованной памяти в локальной памяти процесса или MAP_FAILED в случае ошибки. Код ошибки помещается в errno.

Замечания:

– Новое отображение не будет перекрывать никакое из ранее выполненных отображений.

– Свойства присоединённой именованной памяти можно изменить, используя функцию shm_ctl().

– Для отображения адресов памяти доступа к устройствам следует использовать функцию mmap_device_memory() (см. ниже).

Пример.

```
/* Присоединение или создание разделяемой именованной памяти с
именем "/common" со всеми правами доступа для всех пользователей*/

fd = shm_open("/common", O_RDWR, 0777);
addr = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

Если необходимость в отображении отпала, то его можно аннулировать с помощью функции munmap().

```
#include <sys/mman.h>

int munmap( void * addr, size_t len );
```

Функция munmap() аннулирует все отображения, попадающие в диапазон адресов, начинающийся в addr и длиною len байтов, округленный в большую

сторону кратно размеру страницы памяти. Если нет никаких отображений в определенном адресном интервале, то `munmap()` не имеет никакого эффекта. Функция возвращает в случае ошибки значение -1, а код ошибки заносится в `errno`. Любое другое значение означает успешное завершение.

Пример.

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <sys/stat.h>

char *progname = "sharemem";

void main( int argc, char *argv[]){
  int fd, len, i;
  char *ptr, *name;

if(argc != 3){
     fprintf(stderr, "Ошибка параметров вызова\n");
     exit(EXIT_FAILURE);
}
name = argv[1];//Имя именованной памяти
len = atoi(argv[2]);//Длина именованной памяти

/* Присоединить именованную память для чтения и записи*/

fd = shm_open(name, O_RDWR, 0);
if (fd == -1){
fprintf(stderr, "%s: Ошибка присоединения именованной памяти'%s':
                 %s\n", progname, name, strerror(errno));
    exit(EXIT_FAILURE);
}

/* Отображение разделяемой именованной памяти для чтения и записи*/

ptr = mmap(0,len,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0);
if(ptr == MAP_FAILED){
  fprintf(stderr, "%s: Ошибка отображения: %s\n", progname,
          strerror(errno) );
  exit(EXIT_FAILURE);
}
printf( "%s: Печать содержимого именованной памяти:", progname );
for (i = 0; i < len; i++) printf("%c", ptr[i]);
printf("\n");

close(fd);
```

```
munmap(ptr, len);
}
```

Организация доступа к устройствам ввода/вывода

Для доступа к устройствам ввода/вывода ассоциированные с ними адреса физической памяти (регистров или ячеек памяти) должны быть предварительно отображены в адресное пространство процесса. Для отображения адресов ячеек памяти предназначена функция:

```
#include <sys/mman.h>

void * mmap_device_memory(void * addr,
                          size_t len,
                          int prot,
                          int flags,
                          uint64_t physical);
```

Функция `mmap_device_memory()` отображает пространство адресов физической памяти, ассоциированной с устройством, длинной `len` байтов, начиная с адреса `physical`, в адресное пространство вызвавшего её процесса и возвращает начальный адрес отображения.

Аргумент `addr` предназначен для указания адреса в адресном пространстве процесса, начиная с которого желательно выполнить отображение. Если в этом нет принципиальной необходимости, то следует просто задать `NULL`.

Аргумент `prot` предназначен для задания разрешений по использованию процессом отображаемой области памяти:

`PROT_EXEC` - область можно использовать для исполняемого кода.

`PROT_NOCACHE` – запретить кэширование содержимого памяти.

`PROT_NONE` - область памяти недоступна.

`PROT_READ` - область памяти доступна для чтения.

`PROT_WRITE` - область памяти доступна для записи.

Аргумент `flags` определяет дополнительную информацию об обработке отображенной области. Если `addr` установлен в `NULL`, то для `flags` достаточно указать значение 0.

Аргумент `physical` задаёт начальный адрес физический памяти устройства, отображаемой в адресное пространство процесса.

При успешном выполнении функция возвращает адрес отображения или `MAP_FAILED`, если ошибка (код ошибки помещается в `errno`).

Пример:

```
/*Отображение в процесс текстовой видеопамяти режима VGA, 0xb8000*/

ptr=mmap_device_memory(0,len,PROT_READ|PROT_WRITE|PROT_NOCACHE,0,
                       0xb8000);
```

```
if(ptr==MAP_FAILED) return (EXIT_FAILURE);
```

Для доступа к регистрам (портам) устройств ввода/вывода процесс должен отобразить адреса регистров в адресное пространство процесса. При этом процесс должен иметь привилегию ввода/вывода.

Для отображения используется функция:

```
#include <sys/mman.h>
uintptr_t mmap_device_io(size_t len, uint64_t io);
```

Функция `mmap_device_io()` отображает регистр устройства ввода/вывода размером `len` байтов, адрес которого указан в `io`. В результате выполнения функция возвращает объект типа `uintptr_t`, используемый в функциях семейства `in*()` и `out*()` в качестве адреса порта для доступа к регистру устройства ввода/вывода. В случае ошибки возвращает `MAP_FAILED` и устанавливает `errno`. При успешном выполнении функции `mmap_device_io()` возвращаемый результат используется в следующих функциях.

Функции чтения читают порт `port` как регистр устройства ввода/вывода и возвращают полученное значение:

```
#include <hw/inout.h>
uint8_t in8(uintptr_t port);//Чтение 8-разрядного значения
uint16_t in16(uintptr_t port); //Чтение 16-разрядного значения
uint32_t in32(uintptr_t port); //Чтение 32-разрядного значения
```

Функции записи записывают значение `val` в порт `port` как в регистр устройства ввода/вывода:

```
#include <hw/inout.h>
void out8(uintptr_t port, uint8_t val);
void out16( uintptr_t port, uint16_t val);
void out32( uintptr_t port, uint32_t val);
```

Сигналы

Механизм сигналов

Сигнал является способом уведомления процессов о некотором произошедшем событии, вызывающим изменение текущего хода выполнения процесса (нитей процесса). Воздействие сигнала на процесс схематически похоже на воздействие программно-аппаратного прерывания, однако, это воздействие полностью реализуется ядром системы. При получении сигнала управление в процессе асинхронно передается установленному для данного сигнала действию, предварительно определённому в процессе явно или по умолчанию. В качестве реакции на сигнал процесс может выбрать некоторое стандартное действие, выполняемое по отношению к процессу ядром для данного сигнала, либо может определить собственное действие в виде специальной функции, называемой *обработчиком сигнала*, которой ядром будет передано управление при получении процессом данного сигнала.

Сигнал доставляется процессу ядром в результате возникновения контролируемых ядром системных программно-аппаратных событий. В ядре ОС определен ограниченный набор стандартных сигналов, семантически связанных с возникновением соответствующих системных событий. Каждому сигналу соответствует уникальный целочисленный номер и системная символьная константа. Всего имеется 64 сигнала (см. Приложение). Среди них 8 сигналов реального времени POSIX, которые имеют значения в диапазоне `SIGRTMIN` до `SIGRTMAX` (описание сигналов приведено в документации к функции `SignalAction()`). Доступные пользователю сигналы находятся в диапазоне от 1 до (`NSIG` - 1). Например, событие, связанное с нажатием на клавиатуре клавиш `<Ctrl>/<C>`, вызывает посылку ядром процессу сигнала с номером `SIGINT`, а сигнал `SIGILL` посылается ядром, если процесс попытался выполнить недопустимую инструкцию.

Сигнал доставляется процессу ядром либо по инициативе самого *ядра* при возникновении системных событий, либо по инициативе *пользователя*, либо по инициативе некоторого *процесса*.

Пользователем сигнал инициируется с терминала при выполнении команды shell:

```
kill -<системный_номер_сигнала> <PID процесса>
```

По инициативе процесса сигнал инициируется, например, системным вызовом `kill()`:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid,int sig);
```

Аргумент `pid` адресует процесс или группу процессов, которым посылается сигнал. Если `pid>0`, то адресуется единственный процесс. Если `pid=0`, то сигнал посылается всем процессам, входящим в группу, которой принадлежит

пославший сигнал процесс. Если pid<0, то сигнал посылается каждому процессу, являющемуся членом группы процессов GID=-pid. Аргумент sig равен 0 или определяет системный номер отправляемого сигнала. Если sig равен 0, то сигнал не посылается, а проверяется возможность послать сигнал по указанному pid (наличие адресата). С помощью функции kill() процесс может послать сигнал другому процессу (в частности, самому себе) или группе процессов, если они имеют те же реальный и эффективный идентификаторы пользователя и группы, что и у посылающего сигнал процесса. Это ограничение не распространяется на процессы, обладающие привилегиями суперпользователя. Такие процессы имеют возможность отправлять сигналы любым процессам системы.

Реакция процесса на поступление сигнала, называется *действием* или *диспозицией сигнала*. Для установления процессом диспозиции сигнала используются функции signal() или sigaction(). С помощью этих функций процесс может установить одну из трех возможных реакций на поступление сигнала:

- игнорировать сигнал,
- действовать по умолчанию,
- перехватить (запустить специально объявленную функцию - обработчик сигнала).

Если сигнал игнорируется, то это означает, что при поступлении данного сигнала он будет просто аннулирован и не окажет на процесс никакого воздействия. Существуют, однако, сигналы, которые нельзя ни игнорировать, ни блокировать, ни перехватить. Например, сигнал SIGKILL и SIGSTOP. Сигнал SIGKILL, в частности, является средством принудительного завершения процесса. Если процесс вообще не устанавливает для сигнала диспозицию или явно устанавливает диспозицию по умолчанию, то будет действовать диспозиция по умолчанию. Действие по умолчанию определено в ядре для каждого сигнала. В большинстве случаев по умолчанию при получении процессом сигнала происходит завершение его выполнения, а для сигналов SIGCHLD, SIGIO, SIGURG и SIGWINCH в качестве действия по умолчанию предписано игнорировать сигнал. Например, игнорирование по умолчанию сигнала SIGCHLD, позволяет процессу не ждать завершения дочерних процессов, а они после завершения не превратятся в зомби.

Функция signal() имеет следующее определение:

```
#include <signal.h>
void(*signal(int sig,void(*act)(int)))(int);
```

Аргумент sig определяет сигнал, действие которого нужно изменить. Аргумент act определяет диспозицию сигнала. В качестве значения act может быть присвоен указатель функции обработчика сигнала или одно из следующих системных значений:

SIG_DFL – выполнить действие по умолчанию,

SIG_IGN – игнорировать сигнал.

При успешном завершении signal() возвращает предыдущую диспозицию. Это может быть функция-обработчик сигнала или системные значения - SIG_DFL, SIG_IGN. Возвращаемое значение можно использовать для восстановления диспозиции.

Пример:

```
#include <signal.h>

/*Функциия-обработчик сигнала*/
static void sig_hndlr(int signo){

  printf("Получен сигнал SIGINT = %i\n",signo);//Нажатие <ctrl/c>
}

int main(){
/*Установка диспозиций*/
    signal(SIGINT,sig_hndlr);
    signal(SIGUSR1,SIG_IGN);
    signal(SIGUSR2,SIG_DFL);
/*Бесконечный цикл*/
    while(1){
        pause();//Блокировка на неопределенное время
        puts("pause() завершена\n");
    }
}
```

В примере после запуска процесс блокируется на неопределенное время функцией pause(). При нажатии пользователем клавиш <Ctrl>/<C> ядро посылает процессу сигнал SIGINT. При поступлении сигнала в процессе запускается установленный для данного сигнала обработчик sig_hndlr(). Обработчик выдает сообщение "Получен сигнал SIGINT = 2" и завершает свое выполнение. В результате получения сигнала, который не игнорируется, функция pause() завершается, возвращая –1, выдается сообщение "pause() завершена" и вновь в цикле выполняется pause(). Если пользователь, используя команду kill, посылает процессу сигнал SIGUSR1, то этот сигнал игнорируется и не вызывает в процессе никаких реакций. Если же пользователь посылает процессу сигнал SIGUSR2, то процесс завершается (действие по умолчанию).

Использование для установки диспозиции сигнала функции signal() рассматривается как ненадежная работа с сигналами (устаревший стиль). Функция signal()не предоставляет процессу возможности замаскировать сигнал, т.е. отложить получение сигнала на период выполнения критического участка кода. Например, нельзя предотвратить вложенного вызова обработчика сигнала. Кроме того, не определено влияние сигнала на нити процесса, если их в

процессе более одной. Стандарт POSIX 1003.1 определил набор функций управления сигналами, лишенный указанного недостатка. Введенная POSIX новая семантика управления сигналами (новый стиль) обеспечивает надежную работу с ними.

Механизм надежных сигналов

Набор сигналов и маска блокирования

Механизм надежных сигналов вводит программный объект, называемый *набором сигналов*, который процесс предполагает контролировать. Процесс формирует контролируемый им набор сигналов, используя переменную типа `sigset_t`. Каждый бит такой переменной ассоциируется с соответствующим сигналом (номер бита, начиная с 1, соответствует номеру сигнала в системе). При формировании процессом контролируемого набора сигналов он устанавливает соответствующие биты, а остальные сбрасывает. Для сигналов, включенных в набор, процесс, как правило, явно задает диспозицию.

Если в старом варианте сигналы адресовались только процессу, при этом в общем случае воздействие сигнала на конкретную нить было неопределённым, то в новом варианте сигнал можно адресовать и конкретной нити. Такой сигнал на другие нити процесса не будет оказывать никакого воздействия. Кроме того, механизм надежных сигналов предоставляет нитям процесса возможность блокировать (задерживать) действие сигнала на нить. Для этих целей нити процесса могут, используя программный объект, называемый *маской блокирования* набора сигналов, выполнить запрос блокирования. Маска блокирования так же является переменной типа `sigset_t`. При формировании нитью процесса маски блокирования сигналов следует учитывать, что для блокирования сигнала соответствующий сигналу бит маски блокирования устанавливается, в противном случае – сбрасывается (сигнал деблокируется). Выполняя запрос блокирования, нить может управлять порядком воздействия ожидаемых сигналов на нить. Если сигнал блокирован нитью, то воздействие приходящих инициаций этого сигнала на эту нить задерживается до момента деблокирования сигнала нитью. Если в период блокирования сигнала происходит более одной его инициации, то процесс при установке диспозиции сигнала может выбрать способ учета приходящих инициаций сигнала. В качестве учитываемой инициации можно определить последнюю инициацию сигнала, остальные аннулируются. Или можно выбрать способ, когда все последовательно приходящие инициации одного и того же сигнала ставятся ядром в очередь и по мере возможности воздействуют на нити процесса. Если одному и тому же адресату одновременно приходят инициации разных сигналов, реакция на них осуществляется в порядке приоритета сигналов. Приоритет сигнала тем выше, чем меньше его номер.

Для формирования набора сигналов или маски блокирования сигналов процессу предоставляются следующие функции:

```
#include <signal.h>
```

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set,int signo);
int sigdelset(sigset_t *set,int signo);
int sigismember(const sigset_t *set,int signo);
```

Функция `sigemptyset()` инициализирует набор сигналов или маску блокирования, очищая все биты. Набор сигналов становится пустым, а такая маска может использоваться только для деблокирования всех системных сигналов. Функция `sigfillset()` формирует набор, включающий все системные сигналы, а соответствующая маска используется для блокирования всех сигналов (кроме сигналов, которые нельзя заблокировать). Функции `sigaddset()` и `sigdelset()` позволяют добавлять и удалять сигналы набора, а также устанавливать и сбрасывать соответствующие биты маски блокирования. Функция `sigismember()` позволяет проверить наличие указанного сигнала в наборе или установку соответствующего бита маски блокирования.

Установка диспозиции сигнала

В установке диспозиций сигналов могут участвовать все нити процесса. Однако в итоге сигналу в процессе можно назначить только одну диспозицию. Это будет та диспозиция, которую сформировала нить, последней выполнившая установку диспозиции этому сигналу. Все сформированные нитями диспозиции сигналов принадлежат процессу, а не конкретной нити.

Вместо функции `signal()` механизм надежных сигналов использует функцию `sigaction()`, позволяющую установить диспозицию сигнала, узнать текущую диспозицию или выполнить и то и другое одновременно. Функция имеет следующее определение:

```
#include <signal.h>

int sigaction(int signo,
              const struct sigaction * act,
              struct sigaction * oact);
```

Аргумент `signo` определяет номер сигнала. Диспозиция сигнала задается в структуре типа `sigaction`, которая передаётся через указатель `act`. Если текущая диспозиция не меняется, то указатель `act` равен `NULL`. Если указатель `oact` не `NULL`, то в соответствующей структуре запоминается текущая диспозиция.

Структура `sigaction` включает в себя следующие поля:

`void (*sa_handler)(int signo);` - системный тип действия или указатель обработчика сигналов старого типа.

`void (*sa_sigaction)(int signo,siginfo_t* info,void* other);` - системный тип действия или указатель обработчика сигналов нового типа.

`sigset_t sa_mask;` - маска блокирования, которая используется для блокирования сигналов, в течение времени выполнения обработчика сигнала.

`int sa_flags;` - специальные флаги. По умолчанию все флаги 0. Явно, можно установить флаги `SA_NOCLDSTOP` и `SA_SIGINFO`.

Установка флага `SA_NOCLDSTOP` предписывает ядру не генерировать родительскому процессу сигнал `SIGCHLD`, когда его дочерний процесс был остановлен сигналом `SIGSTOP`. По умолчанию ядро инициирует сигнал `SIGCHLD` родительскому процессу, чтобы проинформировать его об остановке дочернего процесса и предотвратить тем самым блокирование родительского процесса в состоянии ожидания завершения дочернего процесса на неопределенное время.

Флаг `SA_SIGINFO` управляет режимом учета поступающих сигналов. По умолчанию (`sa_flags=0`) режим учета поступления сигнала таков, что многократное инициирование процессу или нити одного и того же сигнала при условии, что сигнал заблокирован или нить находится в ожидании выделения процессорного времени, оставляет актуальной только одну последнюю инициацию сигнала. Все предыдущие инициации сигнала теряются. Если же установлен флаг `SA_SIGINFO`, то все инициации сигнала ставятся ядром в очередь и в итоге будут доставлены адресату.

Поля структуры `sigaction`, предназначенные для задания обработчика сигналов (`sa_handler` и `sa_sigaction`), используются в зависимости от выбора пользователем типа обработчика сигнала (в старом или новом стиле). Эти поля определены посредством `union` и разделяют общую память. Если флаг `SA_SIGINFO` установлен (для учёта всех инициаций сигнала), то целесообразно использовать обработчик сигналов нового типа (указатель функции обработчика сигналов заносится в `sa_sigaction`). Если используется функция обработчика сигналов старого типа, то указатель функции следует заносить в поле `sa_handler`, а режим учета инициаций сигнала целесообразно выбрать по умолчанию (следующая инициация аннулирует предыдущую) так как старый тип функции позволяет получить только номер сигнала.

Системный тип действия для сигнала задаётся символическими константами:

- `SIG_DFL` - определённое для сигнала действие по умолчанию;
- `SIG_IGN` - игнорировать сигнал.

Функция `sigaction()` возвращает 0, в случае успеха, и –1, в случае ошибки и устанавливается `errno`.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main(void){
    extern void handler(int signo);
```

```
    struct sigaction act;
    sigset_t set;

/*Формирование нкабора контролируемых процессом сигналов*/
    sigemptyset(&set);
    sigaddset(&set,SIGUSR1);
    sigaddset(&set,SIGUSR2);

    /*Определение обработчика для сигнала SIGUSR1 так, что когда он
запускается, маскируются все сигналы набора*/
    act.sa_flags = 0;
    act.sa_mask = set;
    act.sa_handler = &handler;//Используется старый тип обработчика
    sigaction(SIGUSR1,&act,NULL);//обработать сигнал SIGUSR1
/*На сигнал SIGUSR2 - действие по умолчанию (процесс завершается)*/

    kill(getpid(),SIGUSR1);

    return EXIT_SUCCESS;//До return процесс не доходит
}
//---------------------------------------------------------------
void handler(int signo){
    static int first = 1;

    printf("Обработка сигнала SIGUSR1= %d.\n", signo);
    if(first){
     first = 0;
     kill(getpid(),SIGUSR2); /*В этот момент сигнал замаскирован*/
     kill(getpid(),SIGUSR1); /*В этот момент сигнал замаскирован*/
    }
    printf( "Завершение обработчика сигнала.\n" );
}
```

В начале в `main()` инициируется сигнал `SIGUSR1`, в результате чего вызывается обработчик сигнала и маскируются сигналы `SIGUSR1` и `SIGUSR2`. В обработчике при первом входе в него последовательно инициируются сигналы `SIGUSR2` и `SIGUSR1` (т.к. `first=1`), однако, пока не завершится выполнение обработчика прерывания, нет реакции на сигналы (они задерживаются). При завершении обработчика демаскируются сигналы `SIGUSR1` и `SIGUSR2`. Задержанные сигналы `SIGUSR1` и `SIGUSR2` теперь актуализируются. Первым срабатывает сигнал `SIGUSR1`, его приоритет выше (так как номер меньше). Вновь запускается `handler()`, но сигналы в нем больше не инициируются (т.к. `first=0`). После очередного завершения обработчика сигналов демаскируется и срабатывает ожидающий сигнал `SIGUSR2`. В результате выполняется действие по умолчанию, и процесс завершается. В итоге до завершения функции `main()` (выполнения оператора `return EXIT_SUCCESS;`) процесс не доходит.

Управление сигналами

Посылка сигнала

Использование для посылки процессу сигнала функция `kill()` имеет ограниченные возможности. Во-первых, адресатом сигнала, посылаемого с помощью функции `kill()`, является процесс (процессы). Воздействие такого сигнала на нити процесса (если их более одной), не всегда очевидна. Во-вторых, если работа осуществляется в сети, функция предоставляет возможность адресовать сигнал процессам только местного узла. Поэтому механизм надежных сигналов вводит для посылки сигнала функцию `SignalKill()`, которая расширяет возможность управления посылкой сигнала. Сигнал можно послать и на удаленный узел, а также адресовать его непосредственно указанной нити. Функция `SignalKill()` имеет вид:

```
#include <sys/neutrino.h>

int SignalKill(uint32_t nd,
               pid_t pid,
               int tid,
               int signo,
               int code,
               int value);
```

Аргумент `nd` определяет дескриптор сетевого узла. Если рассматривается только местный узел, то `nd` следует присвоить значение `ND_LOCAL_NODE` (или 0). Аргумент `pid` задает значение ID процесса, которому посылается сигнал, или указывается равным 0. Аргумент `tid` задает значение ID нити, которой посылается сигнал, или указывается равным 0. Аргумент `signo` определяет системный номер посылаемого сигнала. Аргументы `code` и `value` – это ассоциируемые с сигналом некоторый код и некоторое значение, позволяющие передать вместе с сигналом дополнительную информацию.

Функция `SignalKill` позволяет послать сигнал группе процессов, процессу или нити. Если аргументу `signo` присваивается значение 0, то сигнал не посылается, но таким способом можно проверить наличие указанных процесса и нити. Сочетание значений `pid` и `tid` определяют, кому адресуется сигнал:

pid	**tid**	**Адресат**
=0	-	Группа процессов, которой принадлежит процесс, пославший сигнал
<0	-	Группа процессов (GID = -`pid`)
>0	=0	Процесс (ID процесса равен `pid`)
>0	>0	Нить в процессе (ID нити равен `tid`, ID процесса равен `pid`)

Если нить направляет сигналы процессу, который ставит их в очередь, и генерирует их быстрее, чем процесс может потребить их, то очередная посылка может потерпеть неудачу. Успешное завершение функции означает, что сигнал доставлен адресату.

Вызов не является блокирующим. При работе в сети, низкоприоритетные нити могут выполняться из-за инверсии приоритетов.

Функция `SignalKill()` в случае ошибки возвращает –1, и устанавливает значение `errno`. Любое другое значение говорит об успешном завершении.

Доставка сигнала процессу

Доставка сигнала процессу в зависимости от того, кому он был адресован, приводит в итоге к изменению текущего хода выполнения некоторой одной нити процесса и выполнению соответствующей сигналу диспозиции. Если инициация сигнала адресована процессу, то ядро доставляет эту инициацию одной из нитей, у которой деблокирован соответствующий сигнал и она активна. На какую конкретно нить окажет воздействие инициация сигнала не определено. Чтобы избежать ситуации неопределенности можно придерживаться следующих установок:

- все нити явно блокируют все сигналы своей маской за исключением одной нити, которая обрабатывает все сигналы;
- сигналы адресуются конкретным нитям.

Каждая нить имеет возможность установить собственную маску блокирования сигналов. Если нить замаскировала сигнал, его воздействие на эту нить задерживается до тех пор, пока нить не деблокирует сигнал, сбросив соответствующий бит маски блокирования.

Если сигнал адресован процессу, но все его нити установили собственную маску блокирования сигнала, то инициация сигнала будет задержана процессом. Нить, которая первой деблокирует сигнал, получает задержанную инициацию сигнала.

Если инициация сигнала адресована конкретной нити, то она будет доставлена только этой нити. Если в момент прихода сигнала нить-адресат установила маску блокирования сигнала, то сигнал будет задержан до тех пор, пока нить не демаскирует его. Сигнал никогда не перенаправляется другой нити процесса.

Если сигнал адресуется группе процессов, то сигнал доставляется описанным выше способом каждому процессу в группе.

Для управления блокировкой сигналов нитями используется функция:

```
#include <sys/neutrino.h>

int SignalProcmask(pid_t pid,
                   int tid,
                   int how,
                   const sigset_t* set,
                   sigset_t* oldset);
```

Функция изменяет или проверяет маску блокирования сигналов в направлении нити `tid` в процессе `pid`. Если `pid` равен 0, то рассматривается

текущий процесс. Если tid равен 0, то pid игнорируется, а в качестве нити-адресата выступает сама нить, выполнившая запрос.

Аргумент set используется для изменения текущей маски блокирования. Если нет необходимости изменять текущий набор масок блокирования, то в качестве аргумента следует задать NULL.

Аргумент oldset используется для сохранения предыдущего значения маски блокирования (при изменении) или для получения текущего значения маски блокирования (когда set равен NULL). Если необходимость в аргументе отсутствует, принимает значение NULL.

Аргумент how определяет способ, которым изменяется маска блокирования сигналов. Он может принимать следующие значения:

SIG_BLOCK – итоговая маска блокирования формируется как объединение текущего значения маски и значения, на которое указывает аргумент set.

SIG_UNBLOCK - итоговая маска блокирования формируется как пересечение текущего значения маски и значения, на которое указывает аргумент set.

SIG_SETMASK - итоговая маска блокирования определяется значением, на которое указывает аргумент set.

SIG_PENDING – не изменяет маски блокирования, а позволяет получить сведение о наличии задержанных сигналов, адресованных данной нити или процессу, результат сохраняется в переменной типа sigset_t, на которую указывает oldset. Значение set игнорируется.

Если сигнал при выполнении функции деблокируется, ядро проверяет наличие инициаций данного сигнала, доставленных нити и ожидающих обработки. Если нет доставленных нити инициаций сигналов, ждущих обработки, то ядро проверяет наличие инициаций сигналов, доставленных процессу. Если имеется задержанная процессом инициация сигнала, то она доставляется нити и немедленно действует. Если нет задержанной инициации сигнала, то никакое действие не выполняется.

Невозможно блокировать сигналы SIGSTOP или SIGKILL.

Функция не является блокирующей. В случае ошибки функция возвращает -1 и устанавливает errno. Любое другое значение означает успешное завершение функции.

Реакция процесса на сигнал

Реакция процесса на сигнал определяется установленной процессом диспозицией сигнала. Если в качестве диспозиции установлено игнорирование сигнала, то приход сигнала не вызывает в процессе никаких реакций (исключение составляют сигналы, которые не могут быть игнорированы). Если в качестве диспозиции установлено действие по умолчанию, то реакция процесса соответствует действию по умолчанию для данного сигнала. Если установлен

обработчик сигнала, то в результате прихода инициации сигнала вызывается установленная процессом функция обработчика сигнала (кроме сигналов, которые не допускают перехвата процессом, например, SIGKILL).

Прикладная функция, запускаемая в качестве обработчика сигнала, может объявляться в старом стиле в виде:

```
void handler(int signo)
```

или в новом стиле в виде:

```
void handler(int signo, siginfo_t* info, void* other)
```

Ядро, независимо от стиля обработчика, всегда вызывает его как обработчик нового стиля. Если был установлен обработчик сигналов старого стиля, то дополнительные аргументы все же передаются ядром, но функция их просто не использует.

При вызове ядро передает в обработчик номер сигнала signo и структуру info типа siginfo_t. Аргумент other в настоящее время не применяется и зарезервирован на будущее.

Структура siginfo_t, содержит следующие поля:

int si_signo – номер сигнала, который равен значению аргумента signo обработчика handler.

int si_code – код сигнала, который формируется инициатором сигнала и служит в качестве дополнительной информации, например, для идентификации источника и/или причины посылки сигнала.

union sigval si_value – значение, связанное с данной инициацией сигнала, которое задается инициатором сигнала.

Значение si_code является 8-разрядным целым со знаком. Значения в диапазоне -128 <= si_code <= 0 являются пользовательскими значениями. Значения 0 < signo <= 127 являются системными значениями, генерируемыми ядром, и использоваться пользователями не могут.

В качестве примера ниже приведены некоторые из системных значений si_code, определенных стандартом POSIX, которые используются ядром:

SI_USER – сигнал сгенерирован функцией kill().

SI_QUEUE – сигнал сгенерирован функцией sigqueue().

SI_TIMER – сигнал сгенерирован таймером.

SI_ASYNCIO – сигнал сгенерирован асинхронным вводом-выводом.

IOSI_MESGQ – сигнал сгенерирован очередью сообщений POSIX (не QNX).

Заметим, что сигналы с кодом SI_TIMER (системные сигналы службы реального времени) независимо от значения sa_flags никогда в очередь не ставятся.

Пока обработчик сигнала выполняется, последующие воздействия сигнала на процесс автоматически блокируются, предотвращая тем самым вложенные

вызовы обработчика как реакции на инициации того же сигнала. Кроме того, при установке диспозиции сигнала можно в `sa_mask` указать номера дополнительных сигналов, которые по "ИЛИ" добавляются к маске блокирования при вызове обработчика. Когда обработчик нормально завершает свое выполнение, предыдущее значение маски блокирования сигналов восстанавливается (изменения маски, сделанные в обработчике с использованием функции `SignalProcmask()`, теряются) и задержанные, но теперь демаскированные сигналы, начинают действовать. Однако, для возврата из обработчика сигнала можно использовать функцию `longjmp()`. Тогда маска не восстанавливается, и сигналы остаются замаскированными. При этом для восстановления исходной маски можно воспользоваться функцией `siglongjmp()`, предварительно сохранив маску с помощью функции `sigsetjmp()`.

После завершения обработки сигнала управление передается прерванной нити для её продолжения. Если нить была блокирована ядром, когда ей был доставлен сигнал, то блокирующий запрос завершается и возвращает `EINTR` (исключение составляют запросы `ChannelCreate()` и `SyncMutexLock()`).

Код и значение всегда доставляются с сигналом, несмотря на то, установлен или нет флаг `SA_SIGINFO` для сигнала `signo`. Если `SA_SIGINFO` установлен, можно использовать сигналы, чтобы без потерь доставлять небольшое количество данных. Если `signo`, `code` и `value` сигнала не изменяются, то ядро выполняет сжатие инициаций сигнала в очереди, изменяя 8-разрядный счетчик соответствующего сигнала, находящегося в очереди.

Приведём пример управления сигналами в рамках механизма надежных сигналов.

Пример:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <sys/neutrino.h>
#include <errno.h>
#include <sys/netmgr.h>

int code, value;

int main( void ){

    extern void handler1(int signo,siginfo_t *info,void *over);
    extern void handler2();

    struct sigaction act1, act2;

    sigset_t set;
```

```
    sigemptyset(&set);
    sigaddset( &set, SIGUSR1 );
    sigaddset( &set, SIGUSR2 );
    sigaddset( &set, SIGINT );//Сигнал по <ctrl/c>

    /*Определение обработчика сигнала SIGUSR1 так, что когда он
доставляется, маскируются сигналы SIGUSR1 и SIGUSR2.*/

    act1.sa_flags = SA_SIGINFO;
    act1.sa_mask = set;/Маска для SIGUSR1 и SIGUSR2
    act1.sa_sigaction = &handler1;

    act2.sa_flags = 0;
    act2.sa_mask = set;/Маска для SIGUSR1 и SIGUSR2
    act2.sa_handler = &handler2;


    sigaction( SIGUSR1, &act1, NULL );
    sigaction( SIGUSR2, &act2, NULL );

    code=-12; value=12;//Обратить внимание, что code отрицательное!

/*Посылаемый сигнал маскируется*/
    if(SignalKill(ND_LOCAL_NODE,getpid(),0,SIGUSR1,code,value)==-1)
       perror("SignalKill: ");//Печать ошибки

/*Процесс завершится после обработки сигнала SIGUSR2*/
    puts("OK");
    return EXIT_SUCCESS;//
}

void handler1(int signo,siginfo_t *info,void *over){
    static int first = 1;

    printf("Вошли в handler1 по сигналу %d.\n", signo);
    printf("code = %d.\n", info->si_code);
    printf("value = %d.\n", info->si_value);

    if(first){
      first = 0;
      SignalKill(ND_LOCAL_NODE,getpid(),0,SIGUSR1,code,value);
      /* Сигнал замаскирован */
      kill(getpid(),SIGUSR2); /* Сигнал замаскирован */
    }
    printf( "Завершение handler1.\n" );
  }


void handler2( signo ){
    printf( "Вошли в handler2 по сигналу %d.\n", signo );
    printf( "Завершение handler2.\n" );
}
```

В данном примере определяется набор из двух сигналов SIGUSR1 и SIGUSR2 и устанавливаются два обработчика сигналов handler1() и handler2() так, что при их запуске оба сигнала маскируются. При запуске программы в main() инициируется сигнал SIGUSR1, в результате чего вызывается обработчик сигнала handler1(), а сам сигнал маскируется. При первом входе в handler1() в нём последовательно инициируются сигналы SIGUSR1 (функцией SignalKill()) и SIGUSR2 (функцией kill()), однако, пока не произошло возврата в main(), сигналы остаются замаскированными и реакция на сигналы отсутствует. Возвращение в main() демаскирует сигналы SIGUSR1 и SIGUSR2 и теперь они актуализируются. Первым актуализируется сигнал SIGUSR1, так как его номер меньше и поэтому приоритет выше. Он приводит к повторному запуску handler1(), но сигналы в нем больше не инициируются (first равно 0). После завершения обработчика сигналов handler1() актуализируется ожидающий сигнал SIGUSR2, который доставляется процессу. В результате выполняется обработчик сигналов handler2(). Так как сигналы больше не поступают, то ни что не мешает процессу завершиться, и он завершается по return EXIT_SUCCESS.

Ожидание сигнала

Помимо асинхронной реакции на приход сигнала в ряде случаев нити может потребоваться явно планировать обработку сигналов, предварительно задерживая их, устанавливая маску блокирования. Для этого может использоваться функция:

```
#include <sys/neutrino.h>
int SignalWaitinfo(const sigset_t* set, siginfo_t* info);
```

При выполнении функции нить блокируется в состоянии ожидания прихода сигнала (состояние блокирования STATE_SIGWAITINFO), если в указанном наборе сигналов, на который указывает set, нет ни одного задержанного сигнала. Нить выходит из этого состояния и функция SignalWaitinfo() завершается, когда инициации одного или более сигналов из указанного набора окажутся задержанными маской блокирования или когда ей доставляется инициация сигнала, не блокированного маской. В первом случае функция SignalWaitinfo() извлекает задержанный сигнал из набора сигналов типа sigset_t, на который указывает set, возвращает номер извлечённого сигнала и сохраняет информацию, полученную с извлечённым сигналом, в структуре siginfo_t, на которую указывает аргумент info. Во втором случае реализуется диспозиция сигнала, после выполнения которой SignalWaitinfo() завершается с ошибкой, возвращает -1 и устанавливает в errno код ошибки EINTR.

Аргумент info может принимать значение NULL, если кроме номера сигнала другая информация с инициацией сигнала не важна или не передается.

Механизмы синхронизации нитей с реальным временем

Реальное время (РВ) рассматривается как особый процесс, периодически генерирующий события, называемые моментами времени. Синхронизация нитей с реальным временем преследует цель согласовать работу нитей с наступлением тех или иных моментов реального времени, при возникновении которых должно начаться выполнение нитью того или иного действия или, наоборот, то или иное выполняемое нитью действие должно завершиться.

Системное реальное время

Ядро включает в свой состав службу реального времени (РВ), которая предназначена для моделирования в системе часов реального времени и синхронизации работы нитей с реальным временем. Служба РВ ведет учет астрономического времени (абсолютное время) с точностью до секунды и позволяет отсчитывать интервалы времени (интервальное время) с точностью до тика. Величина тика определяется частотой поступления аппаратных прерываний от таймера, которые обслуживаются соответствующей подпрограммой обработки прерываний (ISR) службы РВ. В компьютерах PC аппаратный таймер строится на базе высокочастотного аппаратного генератора синхроимпульсов (1.1931816 МГц). Высокочастотный меандр этого генератора делится при помощи аппаратного счетчика Intel 82C54 на 11931, который понижает частоту импульсов до 100.00684 герц. Эта частота и является частотой аппаратных прерываний таймера, которые уже могут быть обработаны ISR. В результате величина тика (период между импульсами прерываний) соответственно равна 0.0099993160 с и округляется до 10 мс. Так как реальный тик несколько отличается от 10 мс, служба РВ ядра вводит соответствующие поправки при вычислении времени.

Разрешающая способность РВ

Разрешающая способность (точность) системных часов службы реального времени с одной стороны определяет возможность ОС быть использованной для управления физическими процессами в требуемом темпе, а с другой - определяет эффективность использования системных часов. Служба времени ядра работает в темпе поступления прерываний аппаратного таймера, который определяет величину тика и, следовательно - предельно максимальный различимый во времени темп синхронизируемых процессов. Внутри тика "течение времени" не различимо. Тик - это "протяжённость" текущего "момента времени". Если в рамках одного тика могут произойти более одного события, поступающих от контролируемого процесса, то все они будут помечены одним и тем же значением момента времени и, следовательно, во времени не различимы. Это означает, что разрешающая способность используемых системных часов не достаточна для управления таким процессом в темпе его реального существования, и точность часов надо повышать.

С разрешающей способностью системных часов связано и такое понятие, как *флуктуация отсчета времени*. Темп работы ядра ОС превышает точность

системных часов, поэтому в общем случае ядро фиксирует запросы нитей на планирование времени на протяжении всего тика. Это и приводит к флуктуации отсчета времени. Возникновение флуктуации отсчета времени объясняется ниже на рисунке:

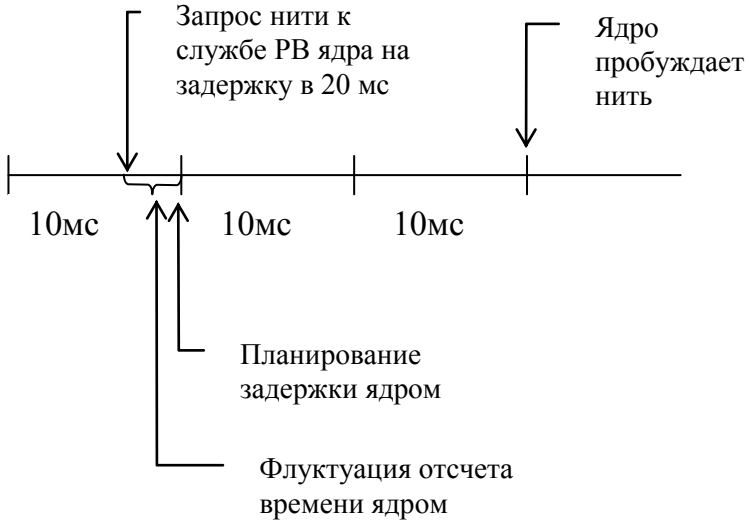


Рисунок. Флуктуации отсчета времени

Из рисунка видно, что если нить планирует задержку в 20мс, то в итоге реальный интервал задержки будет лежать в диапазоне от 20 до 30мс - в зависимости от того, насколько близко или далеко от очередного отсчета времени (прерывания аппаратного таймера) ядро получает запрос от нити. Заметим, что чем ближе темп управляемого физического процесса к темпу системных часов, тем существеннее влияние флуктуации на эффективность управления, что может потребовать увеличения точности системных часов. Однако необходимо при этом учитывать, что избыточная точность системных часов приводит к неоправданным затратам процессорного времени на обработку прерываний службы времени, что снижает эффективность ядра ОС по управлению запросами от нитей прикладных процессов.

Установка значений абсолютного и интервального времени

Рассмотрим порядок установки значений РВ. Значение абсолютного или интервального времени устанавливается с использованием одной и той же структуры данных (определена в `time.h`):

```
struct timespec{time_t   tv_sec;   //секунды
                long     tv_nsec;  //наносекунды, 1сек=10^9нс
               };
```

Видим, что структура `timespec` задает время в секундах, но с точностью до наносекунд. Задание времени с точностью до наносекунд отнюдь не означает, что ядро сможет удовлетворить эту точность. Ядро сможет учесть указанный момент времени только в рамках разрешающей способности службы реального времени. Тогда зачем задавать время с точностью до наносекунд? Ответить на этот вопрос

можно другим вопросом - а с какой точностью надо задавать время? Выбор наносекундной точности, по крайней мере, объективно превышает предельный темп изменения состояния любого процесса в природе, соответствующий скорости света.

Если время задается как абсолютное (календарное), то указанное количество секунд рассматривается как значение интервала времени, отделяющего устанавливаемый момент календарного времени от базового момента календарного времени, определенного как 00 час 00 мин 00 с, 1 января 1970 года по Гринвичу (01.01.1970 00:00:00). Заметим, что максимальное значение, которое можно присвоить переменной типа `long` равно 2 147 483 647. Это дает возможность устанавливать абсолютное время до 2038 года.

В качестве примера задания календарного времени рассмотрим значение:

```
struct timespec it_value;
…
it_value.tv_sec = 987654321;
it_value. tv_nsec = 0;
```

Данная комбинация параметров в контексте абсолютного времени соответствует моменту времени - 19.04.2001 04:25:21. В контексте интервального времени это бы соответствовало интервалу в 31.3 года, что очевидно не имеет практического значения.

В качестве примера задания интервального времени рассмотрим значение:

```
struct timespec it_value;
…
it_value.tv_sec = 5;
it_value. tv_nsec = 500 000 000;
```

Очевидно, что такое значение предполагает его интерпретацию как интервала времени в 5,5 секунды. Иначе, это был бы абсолютный момент времени 01.01.1970 00:00:05 в прошлом, что для синхронизации нитей в реальном времени не имеет практического значения.

Существует набор функций, которые помогают преобразовывать удобный для восприятия человеком формат представления календарного времени с точностью до секунд в число секунд, истекших с базового момента времени и наоборот. Это функции `time()`, `localtime()`, `mktime()`, `ctime()`, `asctime()` и др.

Для преобразования текущего момента календарного времени в количество секунд, прошедших с 00.00.00 по Гринвичу, 1 января 1970 года, используется функция:

```
#include <time.h>
time_t time(time_t * tloc);
```

Функция `time()` возвращает текущее значение абсолютного времени, выраженное в секундах, прошедших с 00:00:00 1 января 1970 года гринвичского

времени (время по Гринвичу обозначают как UTC или GMT). Если `tloc` не NULL, то текущее абсолютное время в секундах сохраняется также в объекте, на который указывает `tloc`.

Если, зная абсолютное время, выраженное в секундах, прошедших с 00:00:00 1 января 1970 года гринвичского времени, необходимо получить значение местного календарного времени, следует воспользоваться функцией:

```
#include <time.h>

struct tm *localtime(const time_t * t_sec);
```

Функция `localtime()` преобразует значение абсолютного времени в секундах по Гринвичу (UTC) в формат местного календарного времени, учитывающего сдвиг местного часового пояса, заданного в переменной среды TZ. Функция возвращает значение местного времени в виде указателя на структуру:

```
#include <time.h>

struct tm {
    int tm_sec;//            секунды [0,61];
    int tm_min;//            минуты [0,59];
    int tm_hour;//           часы [0,23];
    int tm_mday;//           число [1,31];
    int tm_mon;//            месяц с января [0,11];
    int tm_year;//           годы с 1900;
    int tm_wday;//           дни с воскресенья [0,6];
    int tm_yday;//           дни с 1 января [0,365];
    int tm_isdst;//          флажок летнего времени;
    long int tm_gmtoff;//    смещение от времени по UTC (GMT);
    const char * tm_zone;//  название часового пояса.
};
```

Если необходимо наоборот, имея значение абсолютного времени в формате местного календарного времени, представленного в структуре `struct tm`, преобразовать его в количество секунд, прошедших с 00:00:00 1 января 1970 года гринвичского времени, следует воспользоваться функцией:

```
#include <time.h>

time_t mktime(struct tm* timeptr);
```

Функция преобразовывает местное календарное время в формате структуры `struct tm` в формат `time_t`.

Если необходимо получить значение календарного времени, представленное в виде строки, то следует использовать функции:

```
#include <time.h>

char* ctime( const time_t* timer );
char* asctime( const struct tm* timeptr );
```

Функции преобразовывают значение абсолютного времени, заданное в формате `time_t` или `struct tm`, в символьную строку в виде - Tue May 7 10:40:27 2002\n\0.

Таймеры

Для синхронизации с РВ процессы создают объекты типа `timer_t`, называемые *таймерами*. Процессы могут создавать для своих нужд произвольное количество различных таймеров. Дальнейшая работа с таймером со стороны процесса заключается в том, что он планирует таймеру посылку процессу специальных уведомлений, информирующих процесс о наступлении событий реального времени, позволяющих нитям синхронизироваться как с наступлением заданных абсолютных моментов времени, так и с моментами истечения заданных интервалов реального времени.

Создание таймеров

Таймер создается как объект типа `timer_t` с помощью функции:

```
#include <time.h>
#include <sys/siginfo.h>

int
timer_create(clockid_t        clock_id,
             struct sigevent  *event,
             timer_t          *timerid);
```

Созданный таймер предоставляется для использования через указатель `timerid`, являющийся третьим аргументом функции. Первый аргумент `clock_id` сообщает функции `timer_create()` какой тип часов реального времени должен быть выбран. Стандарт POSIX утверждает, что любая платформа должна, по меньшей мере, поддерживать базовый тип `CLOCK_REALTIME`.

QNX/Neutrino поддерживает три типа часов реального времени:

`CLOCK_REALTIME` - стандартный POSIX-определенный тип часов реального времени. Таймер должен будет сработать, даже если процессор находится в режиме экономии энергопотребления.

`CLOCK_SOFTTIME` - часы реального времени, использующие тактовый генератор, который является активным, только когда процессор не находится в режиме экономии энергопотребления. Например, если процесс, использует таймер с типом `CLOCK_SOFTTIME`, чтобы запланировать период бездействия, то таймер не активизирует процесс, когда время истекло и процесс должен был бы проснуться, если в это время процессор находится в режиме экономии энергопотребления. Это позволяет воспользоваться режимом экономии энергопотребления процессора. В то время, когда процессор не находится в режиме экономии энергопотребления, `CLOCK_SOFTTIME` ведет себя так же, как `CLOCK_REALTIME`.

`CLOCK_MONOTONIC` - этот тип часов реального времени, у которого тактовый генератор всегда увеличивается с постоянной частотой и не может корректироваться.

Второй аргумент `event` представляет собой указатель на объект типа `struct sigevent`, определяющий тип уведомления нити о наступлении заданного момента времени.

Удаление таймера

Когда необходимость в таймере пропадает, для освобождения системных ресурсов его можно удалить с помощью функции:

```
#include <time.h>
int timer_delete(timer_t timerid);
```

Типы уведомления нитей

Второй аргумент функции `timer_create()` определяет структуру, которая используется для задания типа уведомления и его параметров. Этот аргумент представляет собой указатель на объект типа `struct sigevent`. Структура `sigevent` включает в себя поля: `sigev_notify`, `sigev_signo`, `sigev_coid`, `sigev_priority`, `sigev_code`, `sigev_value`. Значения этих полей соответствуют выбранному типу уведомления.

Существуют следующие типы уведомлений нитей:

- уведомление типа "послать импульс";
- уведомление типа "послать сигнал";
- уведомление типа "создать нить".

Файл `<sys/siginfo.h>` определяет макрокоманды, которые позволяют упростить инициализацию структуры `sigevent` в соответствии с выбранным типом уведомления. Все макрокоманды используют в качестве первого аргумента - `event`, указатель на `struct sigevent`. Рассмотрим далее, как формируются значения полей в зависимости от выбранного типа уведомления.

Уведомление типа "послать импульс"

Импульс - это отправляемое процессу-серверу специальное сообщение системного типа `stract _pulse`:

```
stract _pulse {_uint16       type;
               _uint16       subtype;
               _int8         code;
               _uint8        zero[3];
               union sigval  value;
               _int32        scoid;
               };
```

Элементы `type` и `subtype` равны нулю (признак импульса). Содержимое элементов `code` и `value` задаются отправителем. Обычно `code` указывает причину, по которой был отправлен импульс, а `value` содержит 32 бита данных, посылаемых с импульсом (т.о. всего 40 бит). Ядро предоставляет 127 отрицательных значений `code` для программистов для использования по своему усмотрению. Код может быть любым 8-битным значением меньшим ноля (-127÷-1), чтобы избежать конфликта с ядром или менеджерами QNX, генерирующими импульсы. Все безопасные системные значения кодов начинаются с `_PULSE_CODE_` и определены в `<sys/neutrino.h>`. Они

заключены в диапазоне от _PULSE_CODE_MINAVAIL до _PULSE_CODE_MAXAVAIL. Элемент value имеет тип объединения вида:

```
union sigval{int   sival_int;
             void *sival_ptr;};
```

Посылка процессом-клиентом импульса и его приём процессом-сервером имеет существенные особенности. Посылка импульса не блокирует процесса-клиента. Прием импульса выполняется как прием обычного сообщения. Отличие только в том, что функция MsgReceive() возвращает ноль (признак прихода импульса) и не требуется посылать ответ, используя функцию MsgReply(). С импульсом можно передать только 40 бит полезной информации (8-битный код и 32 бита данных).

Если требуется принимать только импульсы, оставляя без внимания все другие сообщения, то в этом случае необходимо использовать функцию MsgReceivePulse():

```
int MsgReceivePulse(int             chid,
                    void            *rmsg
                    int             rbytes,
                    struct msg_info *info);
```

Заметим, что параметр info всегда равен NULL.

Если по некоторому каналу принимаются и обычные сообщения, и импульсы, и при этом на нем блокированы нити, выполнившие функцию MsgReceivePulse(), и нет ни одной нити, заблокированной при выполнении функции MsgReceive(), то импульсы будут обслуживаться, а обычные сообщения обслуживаться не будут, а клиенты, пославшие обычные сообщения, будут SEND-блокированными до тех пор, пока какая-либо нить сервера не выполнит функцию MsgReceive(). Но при этом она может принять как обычное сообщение, так и импульс! Поэтому в этом случае необходимо обязательно контролировать возврат функцией MsgReceive() нулевого значения как признака приёма импульса.

Типичным фрагментом кода процесса-сервера, обрабатывающего импульсы, является:

```
#include <sys/neutrino.h>
#define MY_PULSE_TIMER …
stract _pulse *pulse;
char msg[…];
…
rcvid=MsgReceive(chid, msg, …);
if(rcvid==0){//Пришел импульс
            pulse = (stract _pulse *) msg;
            //Определить тип импульса
            switch(pulse->code){
            case MY_PULSE_TIMER://Сработал таймер
```

```
                …
                break;
                //и так далее
} else {//Обычное сообщение, обработать его
}
```

Чтобы создаваемый с помощью функции `timer_create()` таймер настроить на посылку импульсов, необходимо установить полю `sigev_notify` структуры `event` значение `SIGEV_PULSE` и задать значения ряду дополнительных полей. Для этого используется макрокоманда:

```
SIGEV_PULSE_INIT(struct sigevent   *event,
                 int               coid,
                 short             priority,
                 short             code,
                 union sigval      value)
```

где

`int coid` - ID соединения (связи) с каналом, по которому уведомляющий импульс будет посылаться (например, соединение с собственным каналом).

`short priority` – приоритет, связываемый с импульсом, который будет наследоваться нитью, принявшей импульс. Нулевое значение не допускается. Если нет необходимости в изменении приоритета принимающей импульс нити, то `priority` следует установить специальное значение `SIGEV_PULSE_PRIO_INHERIT`.

`short code` – код импульса.

`union sigval value` - 32-битное значение импульса (типа `int` или `void*`).

Рассмотрим, в каких случаях в качестве уведомления целесообразно использовать импульс. Предположим, что разрабатывается сервер, который большую часть времени проводит в RECEIVE-блокированном состоянии, ожидая прихода сообщения по каналу. При этом он планирует прием уведомлений от таймера. В этом случае логично в качестве уведомления сервера таймером использовать импульс. Заметим, что серверу придётся создать соединение с собственным каналом, ID которого будет использовано в качестве параметра при формировании уведомления импульсом.

Уведомление типа "послать сигнал"

В этом случае нить должна установить обработчик сигналов (асинхронный прием сигналов) или переходить в состояние ожидания прихода сигнала, используя функцию `SignalWaitinfo()` или `sigwait()`.Чтобы таймер настроить на посылку сигналов, необходимо установить полю `sigev_notify` структуры `event` значение `SIGEV_SIGNAL`, `SIGEV_SIGNAL_CODE` или `SIGEV_SIGNAL_THREAD`. Использование первых двух значений планирует уведомление сигналом, адресуемым процессу. Использование третьего значения позволяет запланировать посылку сигнала, адресуемого нити.

В первом случае посылается только сигнал без какой-либо дополнительной информации. Для настройки используется макрокоманда:

```
SIGEV_SIGNAL_INIT(struct sigevent *event, int signo)
```

где `int signo` - номер сигнала, который должен быть в диапазоне от `1` до `NSIG-1`.

Если при посылке сигнала необходимо передать дополнительную информацию в виде значений `sigev_code` и `sigev_value`, то надо установить полю `sigev_notify` структуры `event` значение `SIGEV_SIGNAL_CODE`. Для этого используется макрокоманда:

```
SIGEV_SIGNAL_CODE_INIT(struct sigevent  *event,
                       int              signo,
                       void             *value,
                       short            code)
```

где дополнительно:

`void *value` - 32-битное значение, предназначенное для интерпретации обработчиком сигнала.

`short code` - код, который должен быть в диапазоне от `SI_MINAVAIL` до `SI_MAXAVAIL` и предназначен для интерпретации обработчиком сигнала.

Если сигнал необходимо направлять конкретной нити, то надо установить полю `sigev_notify` структуры `event` значение `SIGEV_SIGNAL_THREAD`. Для этого используется макрокоманда:

```
SIGEV_SIGNAL_THREAD_INIT(struct sigevent  *event,
                         int              signo,
                         void             *value,
                         short            code)
```

В этом случае сигнал доставляется той нити, которая его запланировала, обратившись к ядру с помощью функции `timer_settime()`.

Уведомление типа "создать нить"

Этот тип уведомления предполагает, что в результате срабатывания таймера в процессе создается новая нить. Полю `sigev_notify` структуры `event` устанавливается значение `SIGEV_THREAD`. Для этого используется макрокоманда:

```
SIGEV_THREAD_INIT(struct sigevent   *event,
                  void              (*fn)(union sigval)
                  void              *value,
                  pthread_attr      *attributes)
```

где:

`void (*fn) (union sigval)` - указатель на функцию, которую нужно запустить как нить.

`pthread_attr *attributes` - указатель на атрибутную запись нити. Он должен быть `NULL`, или указывать на структуру, которая будет использована функцией `pthread_attr_init()`.

`void *value` - значение, которое передается функции, запускаемой как нить.

Этим типом уведомления надо пользоваться крайне осмотрительно и при отсутствии особой необходимости лучше избегать. Если таймер будет срабатывать слишком часто, и при этом будут готовы к выполнению нити с более высоким приоритетом, чем вновь создаваемые, то быстро вырастет очередь заблокированных нитей, которые исчерпают все ресурсы системы.

Типы и режимы планирования таймеров

По способу планирования событий реального времени различают таймеры относительные и абсолютные. *Абсолютный таймер* позволяет нити планировать абсолютные моменты реального времени. *Относительный таймер* позволяет нити планировать интервалы реального времени. По количеству генерируемых уведомлений таймеры делят на однократные и периодические. *Однократный таймер* - это таймер, который посылает уведомление только один раз, когда достигается нужный момент времени. *Периодический таймер* - это таймер, которому указывается интервал времени, и он посылает уведомление каждый раз по истечении этого интервала времени и планирует очередное уведомление. В связи с этим различают три типа таймеров:

- абсолютный однократный;
- относительный однократный;
- относительный периодический.

Тип таймера задается при его запуске. Запуск таймера и указание его типа (путем комбинирования значений аргументов) осуществляется с помощью функции:

```
#include <time.h>
int
timer_settime(timer_t                    timerid
              int                        flags,
              struct itimerspec         *alarm,
              struct itimerspec         *oldalarm);
```

Значение аргумента `timerid` задает запускаемый таймер, созданный функцией `Timer_Create()`. С помощью аргумента `flags` таймер определяется как абсолютный (указывается значение `TIMER_ABSTIME`) или относительный (указывается значение 0). Аргумент `alarm` используется для установки моментов срабатывания таймера как однократного или периодического. Аргумент `oldalarm` возвращает значение предыдущей

установки таймера. Если необходимости в ней нет, то oldalarm присваивают значение NULL.

Структура itimerspec имеет вид:

```
struct itimerspec{
 struct timespec it_value;/*Абсолютный момент времени или период
                            первого уведомления */
 struct timespec it_interval;//Период последующих уведомлений
};
```

Абсолютный таймер всегда однократный. Он посылает уведомление один раз, как только текущее значение реального времени окажется не меньше значения, указанного в alarm.it_value. Значение alarm.it_interval для абсолютного таймера роли не играет.

Чтобы получить относительный однократный таймер, необходимо, чтобы значение поля alarm.it_value было *отлично от нуля*, а значение поля alarm.it_interval *равно нулю*. Он посылает уведомление один раз, как только истекший интервал реального времени окажется не меньше значения, указанного в alarm.it_value.

Чтобы получить относительный периодический таймер, необходимо, чтобы значения времени в alarm.it_value и alarm.it_interval были *отличны от нуля*. Таймер первый раз пошлёт уведомление, как только истекший интервал реального времени окажется не меньше значения, указанного в alarm.it_value, и с этого момента будет периодически посылать уведомления, как только истекший с момента предыдущего уведомления интервал реального времени окажется не меньше значения, указанного в alarm.it_interval.

Если необходимо отменить действие ранее запущенного таймера, следует вновь выполнить функцию timer_settime() со значением времени в alarm.it_value равным нулю.

Пример программирования таймера

Рассмотрим пример функции, которая создает периодический таймер, уведомляющий нить импульсом каждую секунду.

```
/*
 *  timer.c
 *
 * Пример сервера, получающего периодические импульсы от
 *  таймера и сообщения от клиентов.
 *
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <signal.h>
```

```
#include <errno.h>
#include <unistd.h>
#include <sys/siginfo.h>
#include <sys/neutrino.h>

// сообщения клиентов
#define MT_DATA1    2 // сообщение1 от клиента
#define MT_DATA2    3 // сообщение2 от клиента

// импульс

#define CODE_TIMER  1 // импульс от таймера

// структура сообщения
typedef struct{
 int messageType; //содержит тип сообщения клиента
 int messageData; //данные, зависящие от типа сообщения
} ClientMessageT;

typedef union{
 ClientMessageT  msg;   // сообщение от клиента
 struct _pulse   pulse; // импульс от таймера
} MessageT;

/*
    Программа создает таймер, посылающий нити импульс с кодом
    CODE_TIMER каждую секунду
*/
int     chid;          // ID канала
char    *progname = "timer.c";

//прототипы функций
static  void gotAPulse (void);
static  void gotAMessage (int rcvid, ClientMessageT *msg);

void main(void){
    timer_t               timerid;   // ID таймера
    struct sigevent       event;     // тип уведомления
    struct itimerspec     timer;     // значение времени
    int                   coid;      // ID соединения

if ((chid = ChannelCreate (0)) == -1) {
        fprintf (stderr, "%s: не могу создать канал!\n", progname);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // установить соединение с собственным каналом
    coid = ConnectAttach (0, 0, chid, 0, 0);
    if (coid == -1) {
        fprintf (stderr, "%s:  ошибка соединения!\n", progname);
        perror (NULL);
```

```
        exit (EXIT_FAILURE);
    }
    // установить тип уведомления "послать импульс"
    SIGEV_PULSE_INIT(&event,
                        coid,
                        SIGEV_PULSE_PRIO_INHERIT,
                        CODE_TIMER,
                        0);

    // создать таймер
    if(timer_create (CLOCK_REALTIME,&event,&timerid)==-1){
        fprintf (stderr, "%s:  не могу создать таймер, errno %d\n",
                 progname, errno);
        perror (NULL);
        exit (EXIT_FAILURE);
    }

    // установить периодический таймер (1с задержки и интервал 1с)
    timer.it_value.tv_sec = 1;
    timer.it_value.tv_nsec = 0;
    timer.it_interval.tv_sec = 1;
    timer.it_interval.tv_nsec = 0;

    // запуск таймера!
    timer_settime (timerid, 0, &timer, NULL);
// прием сообщений
    for (;;) {
        rcvid = MsgReceive (chid, &msg, sizeof (msg), NULL);

        // определить, что за сообщение получено
        if (rcvid == 0) {
            // необходимо проверить поле кода
            gotAPulse ();
        } else {
            gotAMessage (rcvid, &msg.msg);
        }
    }

    //сюда никогда не попадем
    return (EXIT_SUCCESS);
}

/*********************************************
 *  Функция gotAPulse
*  Выполняется, когда получен импульс от таймера
 *********************************************
*/
void
gotAPulse (void){
time_t  now;
          time(&now);//Получаем текущий момент времени
          //Печать строки времени
```

```
          printf ("Импульс получен в %s", ctime(&now));
}
/*********************************************
 * Функция gotAMessage
 * Вызывается, когда приходит сообщение от клиента
 *********************************************/
void
gotAMessage (int rcvid, ClientMessageT *msg)
{
     // Определить тип сообщения
     switch (msg->messageType) {

     case    MT_DATA1:
                  printf ("сообщение типа MT_DATA1");
                  MsgReply (rcvid,EOK,msg,sizeof(*msg));
                  return;

     case    MT_DATA2:
     printf ("сообщение типа MT_DATA1");
     MsgReply (rcvid, EOK, msg, sizeof (*msg));
return;
}
```

Таймауты ядра

В целом ряде случаев, обращаясь к ядру с запросом, нить оказывается в заблокированном состоянии. Однако, в некоторых случаях, у нити нет возможности или принципиальной необходимости ждать реакции на свое обращение к ядру излишне долго. Для нити может оказаться более эффективным отказаться от исполнения запроса и продолжить свое выполнение. Для таких случаев ядро предоставляет нитям возможность планировать время нахождения в заблокированном состоянии, после истечения которого нить деблокируется ядром и уведомляется о досрочном деблокировании. Этот механизм называется таймаутом ядра. Ядро позволяет нитям устанавливать таймауты для всех блокированных состояний.

Для формирования таймаута нить, перед тем, как обратиться с блокирующим запросом к ядру, должна выполнить функцию `TimerTimeout()`:

```
#include <sys/neutrino.h>

int

TimerTimeout(clockid_t              clock_id,
             int                    flags,
             const struct sigevent  *event,
             const uint64_t         *timout,//в наносекундах
             uint64_t               *oldtimout); //в наносекундах
```

При успешном выполнении функция возвращает 0. Если ошибка, то -1, код ошибки помещается в `errno`. Аргумент `clock_id` задаёт выбранный тип часов реального времени (`CLOCK_REALTIME` или другой имеющийся в системе тип часов). Аргумент `flags` специфицирует блокирующее состояние,

соответствующее блокирующему запросу к ядру. В аргументе event всегда задаётся специальный тип уведомления SIGEV_UNBLOCK. Для этого удобно использовать макрос:

```
SIGEV_UNBLOCK_INIT(struct sigevent *event).
```

Но можно также просто присвоить event значение NULL. Функция рассматривает это как установку уведомления SIGEV_UNBLOCK.

Аргумент timout указывает относительное время в *наносекундах* (1 с = 1000000000 нс), спустя которое ядро должно послать нити уведомление о таймауте. Если блокирование вообще не допустимо, то timout устанавливается равным NULL. Аргумент oldtimout сохраняет предыдущее значение таймаута, если не используется - присваивается NULL.

Значение, которое необходимо задать в аргументе flags, указано для каждого состояния блокировки в справочной системе QNX в описании функции TimerTimeout().

В качестве примера рассмотрим некоторые из установок флага flags:

- _NTO_TIMOUT_JOIN – установка таймаута при использовании блокирующей функции pthread_join();
- _NTO_TIMOUT_SEND – установка таймаута для SEND-блокированного состояния при использовании блокирующей функции MsgSend();
- _NTO_TIMOUT_REPLY – установка таймаута для REPLY-блокированного состояния при использовании блокирующей функции MsgSend().

Ниже приводится определение функции pthread_join_nb(), осуществляющей присоединение без блокировки. Функция использует функцию pthread_join(), предварительно установив таймаут ядра с нулевым значением.

```
int pthread_join_nb(int tid,void **rval){
   TimerTimeout(CLOCK_REALTIME,_NTO_TIMOUT_JOIN,NULL,NULL,NULL);
   return(pthread_join(tid,rval));
}
```

Функция pthread_join_nb() проверяет, терминирована или нет нить с дескриптором tid. Если нет, то возвращается уведомление ядра о досрочном деблокировании. Если да, то возвращается EOK.

Сбрасывать таймаут после любого варианта завершения системного вызова не надо – это выполняется автоматически.

Особенность использования таймаутов ядра при посылке сообщения

При посылке сообщения, выполнив функцию MsgSend(), клиент может оказаться либо в SEND-блокированном, либо в REPLY-блокированном

состоянии. Поэтому, при планировании таймаута для выхода из состояния блокировки при передаче сообщения, необходимо предусмотреть оба блокирующих состояния, сформировав значение аргумента `flags` в виде `(_NTO_TIMOUT_SEND I _NTO_TIMOUT_REPLY)`. Это вызовет установку ядром таймаута всякий раз, когда ядро переведет клиента в SEND-блокированное, а затем и в REPLY-блокированное состояние. Если таймаут истекает в SEND-блокированном состоянии, то функция `MsgSend()` завершается, возвращая клиенту признак `ETIMEDOUT`. Так как сервер еще не выполнил функцию `MsgReceive()`, то он практически не замечает, что клиентом осуществлялась попытка послать сообщение.

Для REPLY-блокированного состояния все несколько сложнее и зависит от того, установил ли сервер при создании канала флаг `_NTO_CHF_UNBLOCK` или нет. Если флаг не установлен, то клиент будет немедленно разблокирован при истечении таймаута. При этом сервер не получит об этом никакого оповещения и будет планировать выполнение функции `MsgReply()`, хотя клиент уже не ждёт ответа. Если при создании сервером канала флаг `_NTO_CHF_UNBLOCK` был установлен, то при истечении таймаута нахождения клиента в REPLY-блокированном состоянии ядро посылает серверу уведомление в виде импульса, а клиент продолжает оставаться заблокированным до тех пор, пока сервер ему не ответит. В этом случае сервер должен принять решение и выполнить какие-то действия.

Управление прерываниями

Взаимодействие с устройствами ввода/вывода

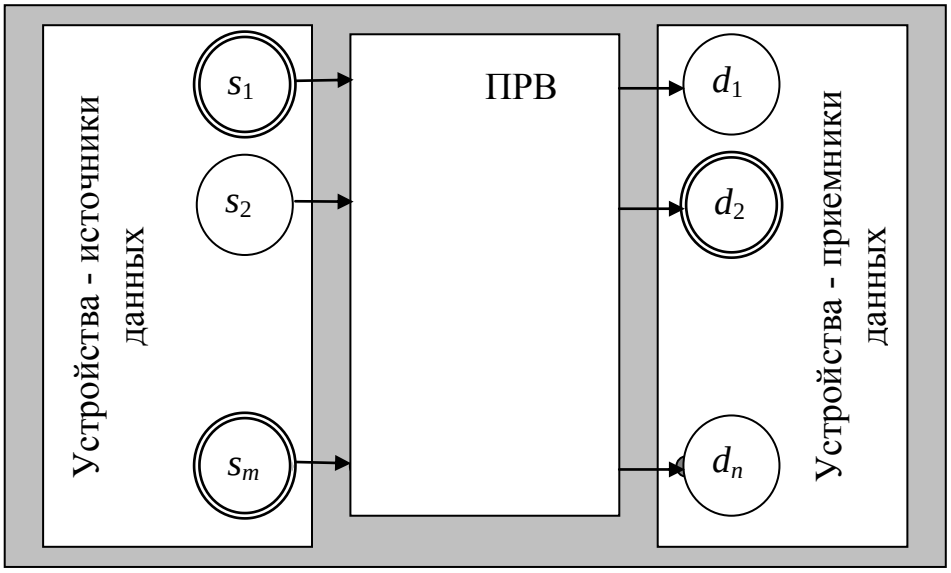


Рисунок 1. Внешнее окружение ПРВ

В процессе своего функционирования ПРВ осуществляет взаимодействие с внешними устройствами, которые по отношению к ПРВ выступают в роли источников исходных и/или приемников результатных данных (см. рисунок 1). На рисунке 1 внешние устройства схематически представлены кругами. Источники данных обозначены как s_i, (i=1,2,…,m), а приемники данных - d_j, (j=1,2,…,n). Организация взаимодействия ПРВ с внешними устройствами предполагает распределение между ними ответственности за инициацию взаимодействия. Та сторона, которая назначается ответственной за инициацию взаимодействия, считается *активной*. Противоположная сторона – *пассивной*. Если активной стороной назначается ПРВ, то ответственность за выявление готовности пассивного устройства предоставить или принять данное возлагается на ПРВ. В этом случае ПРВ вынуждено с определенной частотой опрашивать готовность устройства (статусный регистр), например, периодически планируя запуск соответствующей нити через интервал времени Δt. Если активной стороной назначается устройство, то ответственность за информирование пассивного ПРВ о готовности устройства предоставить или принять данное возлагается на устройство. Для того чтобы в момент готовности устройства заставить ПРВ начать взаимодействовать с устройством, устройство использует механизм аппаратного прерывания ПРВ. На рисунке 1 активные устройства обозначены окружностями, выполненными двойной линией.

Механизм аппаратного прерывания

Механизм прерываний обеспечивает ПРВ возможность асинхронно переключиться по сигналу, выставленному устройством на линии прерывания процессора, на взаимодействие с устройством. Появление сигнала на линии

прерывания заставляет процессор прервать выполнение текущего программного кода и переключиться на выполнение программного кода по адресу, находящемуся в векторе прерывания, соответствующего данному устройству. Обращение к этому программному коду рассматривается как вызов специальной функции, называемой *обработчиком прерывания*. Для выхода из обработчика прерывания и возврата к прерванному программному коду используется специальная процессорная команда завершения прерывания. Механизм прерывания может быть инициирован не только аппаратно внешним устройством, но и запущен программой посредством выполнения специальной процессорной команды, которая в качестве аргумента использует номер вектора прерывания.

Рассмотрим подробнее порядок инициации аппаратного прерывания внешним устройством. Инициация прерывания осуществляется контроллером внешнего устройства, подключенного к шине, путем выставления сигнала на линии прерывания процессора (запрос прерывания). При этом прерывание становится возможным только при условии, что процессор переведен в состояние готовности принимать запросы прерывания (установлен соответствующий разряд регистра словосостояния процессора – прерывания деблокированы) и завершил выполнение текущей команды. До этого момента запросы прерывания не обслуживаются. Если процессор запускает механизм прерывания, то он:

- получает от контроллера соответствующий устройству номер вектора прерывания;
- сохраняет в стеке текущее содержимое регистров процессора и блокирует прерывания (предотвращая вложенные прерывания);
- в соответствующие регистры процессора загружается новое содержимое из вектора прерывания указанного номера (в частности, устанавливается новое значение регистра словосостояния процессора, что может привести к деблокированию прерываний);
- управление передаётся по адресу, которой задан в векторе прерывания, начинает выполняться подпрограмма обработки прерывания (Interrupt Service Routine – ISR);
- при завершении выполнения и выходе из обработчика прерываний ранее сохраненные в стеке значения регистров процессора восстанавливаются, что приводит к восстановлению выполнения прерванного программного кода (с команды, непосредственно следующей за командой, после выполнения которой управление асинхронно было передано обработчику прерываний), а запросы прерывания деблокируются.

Так как линия запроса прерывания у процессора одна, а внешних устройств в общем случае больше одного, то подключение к ней устройств и упорядочение (арбитраж) их запросов прерывания осуществляется с помощью специальных программируемых контроллеров прерывания (Programmable Interrupt Controller –

PIC; в процессорах серии x86 это микросхема Intel 8259 или ей эквивалентная). Принципиальная схема PIC приведена на рисунке 2.

Контроллер прерывания имеет восемь входов IRQ0,IRQ1,…, IRQ7 (IRQ - Interrupt Request Query) для подключения к ним линий запросов прерываний контроллеров внешних устройств. Поступающие на входы IRQ запросы прерывания от контроллеров внешних устройств устанавливают в 1 соответствующие разряды регистра IRR (Interrupt Request Register). В арбитраж запросов прерывания попадают только те запросы из IRR, которым в регистре маски запросов IMR (Interrupt Mask Register) соответствует нулевое значение разрядов (на рисунке 2 это запросы IRQ3 и IRQ5). В стандартном режиме настройки PIC более приоритетным считается запрос с меньшим номером. Поэтому в регистре источников прерывания единичное значение формируется в разряде, соответствующем запросу IRQ3. В соответствии с выбранным источником прерывания формируется номер вектора прерывания, после чего подается запрос на линию прерывания процессора, который обрабатывается процессором описанным выше способом. После того, как процессор среагирует на прерывание, по линии сброса автоматически осуществляется очистка соответствующего разряда в IRR (в данном случае - IRQ3).

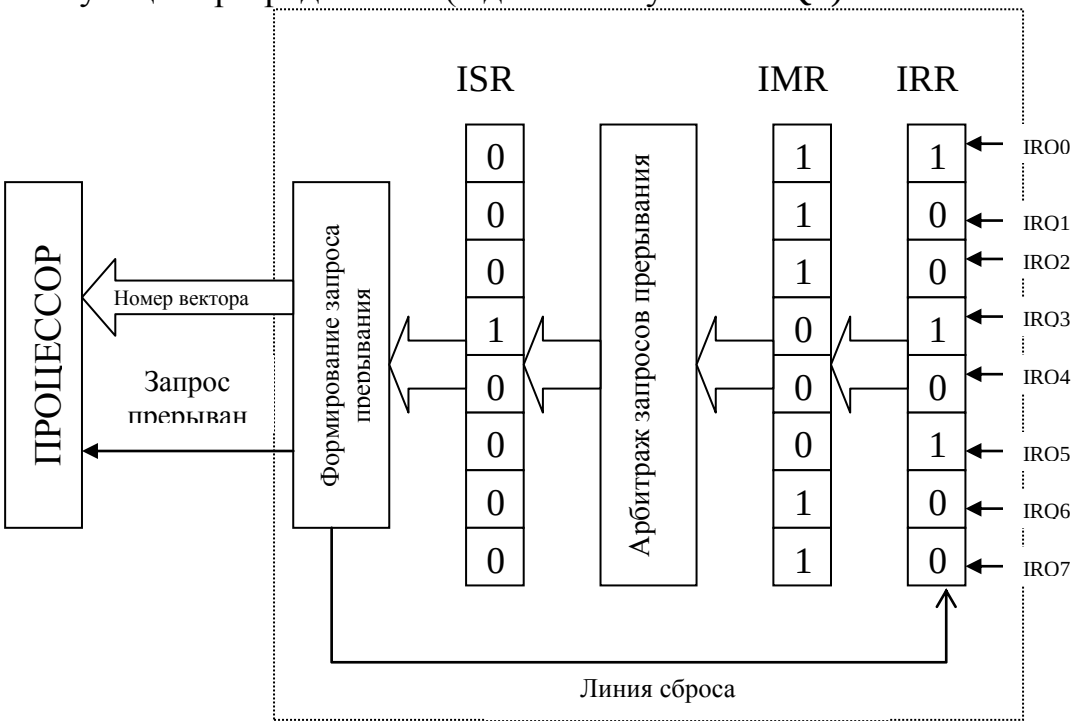


Рисунок 2. Принципиальная схема PIC

Имеется возможность программным способом изменять режимы работы PIC. Например, можно установить другой приоритет линий запроса прерывания, загрузив соответствующую информацию в статусный регистр PIC. Можно также отменить автоматический сброс по линии сброса. В этом случае сброс разрядов регистра IRR должен будет выполняться программно путем посылки в порт 20_{16} значения 20_{16} (совпадение случайное). Возможна настройка запроса прерывания по уровню, по фронту сигналов и пр.

Если количество устройств превышает 8, то используется ещё одна микросхема PIC, которая своим выходом связывается с IRQ7 предыдущего PIC, расширяя количество линий подключения запросов прерывания от устройств.

Обработка прерываний в QNX

Ядро ОС QNX полностью берет на себя заботу по фиксации и обслуживанию всех возникающих запросов прерываний, а процессам предоставляет программный интерфейс управления прерываниями, с помощью которого нити процесса могут указывать операционной системе требуемый источник прерываний и ожидать уведомления от ядра о поступлении запроса прерывания. После этого при каждом поступлении запроса прерывания от указанного источника ядро уведомляет об этом ожидающую прерывания нить, переводя её в состояние готовности. За нитью сохраняется лишь необходимость выполнения протокола взаимодействия с инициатором прерывания (например, внешним устройством для обмена данными).

В качестве источников аппаратных прерываний рассматриваются разряды регистра IRR, которые QNX идентифицирует номерами 0÷15. Типичная семантика номеров прерываний для компьютеров с архитектурой x86 следующая:

0- Прерывания тиков интервального таймера, поступающие с интервалом, устанавливаемым функцией `ClockPeriod()`.

1- Прерывания, генерируемые нажатием клавиши клавиатуры.

2- Прерывания от ведомого контроллера прерываний 8259 (используются при каскадировании аппаратных прерываний).

3- Прерывания асинхронного порта COM2.

4- Прерывания асинхронного порта COM1.

5- Прерывания сетевой (или звуковой) карты.

6- Прерывания контроллера флоппи диска.

7- Прерывания принтера (если адаптер принтера использует это прерывание).

8- Переназначаемое прерывание (для нестандартных внешних устройств).

9- Переназначаемое прерывание (для нестандартных внешних устройств).

10- Прерывания сопроцессора.

11- Прерывания сопроцессора.

12- Прерывания сопроцессора.

13- Прерывания сопроцессора.

14- Прерывания первого IDE-контроллера.

15- Прерывания второго IDE-контроллера.

Как только сигнал прерывания возникает, ядро переключается на участок кода, который выполняет необходимые подготовительные действия (настраивает окружение) для запуска на выполнение специально определенной в процессе функции, играющей роль *обработчика прерывания* - ISR. Концепция обработчика прерывания в QNX такова, что он рассматривается как посредник между ядром и соответствующей нитью процесса, ожидающей прерывание. При подключении процесса к источнику прерыванию он указывает ядру обработчик прерывания, который каждый раз будет вызываться ядром на выполнение при возникновении прерывания. Эта функция запускается с приоритетом, который выше приоритета любой нити. С учетом этого время выполнения ISR должно быть минимальным, так как в противном случае время, затрачиваемое на выполнение обработчика прерывания, может оказать серьезное воздействие на диспетчеризацию нитей. Когда обработчик прерывания завершается, он может либо сообщить ядру, что ничего больше делать не надо (полностью завершил работу, связанную с прерыванием), либо инициировать посылку от ядра процессу, подключившему ISR, уведомления, вследствие которого нить, ожидающая уведомления о прерывании, выходит из состояния ожидания.

Интервал времени с момента установки аппаратурой сигнала прерывания до выполнения первой инструкции обработчика прерываний называется *временем реакции на прерывание*. Время реакции на прерывание измеряется в микросекундах. Различные процессоры характеризуются различными временами реакции на прерывание. Это зависит от быстродействия процессора, архитектуры кэша, быстродействия памяти и, конечно, от эффективности операционной системы.

Программирование обработки прерываний в процессах

В общем случае процесс, реагирующий на прерывания, должен определить и указать ядру обработчик прерывания, а также иметь в своем составе нить, которая будет активизироваться уведомлением, посылаемым ядром для этой нити по заказу ISR.

Определение обработчика прерываний

Функция, которая используется в качестве обработчика прерывания, должна быть объявлена в виде:

```
struct sigevent* isr(void*area,int id);
```

Функция `isr()` имеет два аргумента – `area` и `id`. Аргумент `area` – адрес статической области памяти, используемой процессом для обмена данными между нитью и ISR. Это позволяет процессу подготовить в этой области памяти данные, которые будут доступны `isr()` после запуска ядром, и получить результаты работы от `isr()` после завершения. Если необходимости в обмене данными между процессом и `isr()` нет, то при установке `isr()` в качестве `area` указывается `NULL`. Второй аргумент – `id`, предназначен для получения от ядра идентификатора дескриптора источника прерываний в теле `isr()`. Это

необходимо, если один и тот же обработчик прерываний используется для подключения к разным источникам прерываний, и позволяет обработчику отличить источник текущего прерывания.

После завершения работы обработчик прерывания возвращает указатель на структуру `struct sigevent*`, специфицирующую тип уведомления о прерывании, которое ядро должно послать нити, выполнившей подключение обработчика прерывания. Уведомление о прерывании должно специфицироваться как уведомление типа `SIGEV_INTR`. Для этого удобно использовать макрос:

```
SIGEV_INTR_INIT(struct sigevent *event);
```

Если обработчик прерывания не хочет инициировать посылку нити уведомления о прерывании, он должен возвратить значение NULL.

При написании обработчика прерываний под ОС QNX необходимо учитывать следующие особенности:

1- Размер стека, отводимого для ISR, ограничен 200 байтами, поэтому следует избегать многочисленных вложенных вызовов функций и рекурсии.

2- Если прерывание разрешено, то, как только происходит прерывание, ядро асинхронно запускает ISR, вытесняя текущую активную нить.

3- Обработчик прерывания выполняется с наивысшим приоритетом (приоритетом ядра), поэтому он должен быть максимально быстрым.

4- Обработчик прерывания не может использовать любые функции программного интерфейса, а только разрешённые ОС для безопасного использования в ISR (см. описание функций), например - `InterruptMask()`, `InterruptUnmask()`, `InterruptLock()`, `InterruptUnlock()`;

5- Переменные, используемые для сохранения значений регистров и адресов памяти аппаратуры должны объявляться в процессе с модификатором `volatile` (например, `volatile int …`), чтобы запретить компилятору кэшировать значения этих переменных, поскольку они могут быть изменены в любой точке выполнения процесса.

Подключение процесса к источнику прерываний

Ядро не позволяет любой нити процесса выполнять подключение ISR к источнику прерываний и получать от ядра уведомления прерываний. Такая нить должна предварительно получить право привилегированного ввода/вывода. Получить это право могут только нити, которые выполняются под идентификатором пользователя root или которые установили свой идентификатор пользователя в root при помощи функции `setuid()`. Следовательно, реально эти права есть только у пользователя root.

Для получения нитью права привилегированного ввода/вывода используется функция:

```
#include <sys/neutrino.h>
int ThreadCtl(int cmd,void *data);
```

В качестве cmd следует использовать значение _NTO_TCTL_IO, а data – NULL. В итоге вызов функции должен иметь вид:

```
ThreadCtl(_NTO_TCTL_IO,NULL)
```

В случае ошибки функция возвращает –1 и заносит код ошибки в errno.

Процесс может подключаться к источнику прерываний с установкой собственного обработчика прерываний или без него

Чтобы подключиться к прерыванию с указанием собственного обработчика прерываний ISR, процесс должен использовать функцию:

```
#include <sys/neutrino.h>
int InterruptAttach(int intr,
                const struct sigevent* (*isr)(void*area,int id),
                const void *area,
                int size,
                unsigned flags);
```

intr – номер источника прерываний;

isr – адрес ISR;

area – адрес статической области памяти, используемой для обмена данными между процессом и ISR, который передается в isr() в качестве первого аргумента, или NULL, если память не используется;

size – размер области area или 0, если память не используется;

flags – флаги, специфицирующие порядок использования ISR.

Значение аргумента flags может быть равно 0 и тогда статус ISR устанавливается по умолчанию. Явно можно установить следующие опции:

_NTO_INTR_FLAGS_END – указывает, что данный ISR должен сработать после всех других обработчиков данного прерывания (если номер прерывания разделяется несколькими обработчиками);

_NTO_INTR_FLAGS_PROCESS – указывает на то, что ISR необходимо ассоциировать с процессом, а не с нитью, его подключившей. В результате ISR будет отключаться автоматически от прерывания только при завершении процесса, а не нити;

_NTO_INTR_FLAGS_TRK_MSK – указывает, что ядро должно отследить, сколько раз данное прерывание было маскировано. Это приводит к несколько большей загрузке ядра, но обеспечит корректное демаскирование

прерывания при завершении нити или процесса.

Функция возвращает `ID` подключенного источника прерываний. В случае ошибки возвращается `-1` и код ошибки устанавливается в `errno`.

При подключении процессом источника прерываний без установки обработчика прерываний используется функция:

```
#include <sys/neutrino.h>
int InterruptAttachEvent(int intr,
                         const struct sigevent* event,
                         unsigned flags);
```

В этом случае ядро не будет вызывать ISR, а сразу направляет соответствующей нити уведомление прерывания. Нить будет активизироваться по каждому прерыванию, хотя для разделяемых источников прерываний различными инициаторами сигналов прерывания необходимость в этом для данной нити может отсутствовать (если используется ISR, то она могла бы проверить состояние своего источника сигнала прерывания непосредственно в контроллере внешнего устройства, и не инициировать уведомления для нити, если сигнал не установлен). Такой режим обработки прерываний увеличивает затраты ядра на перепланирование нитей. Однако преимущество использования этой функции заключается в том, что отсутствие явно заданного ISR устраняет опасность разрушить систему, в случае ошибок программирования в ISR. Обработчик прерываний выполняется в пространстве ядра и ошибки ISR могут приводить к фатальным последствиям для системы реального времени. Выбор способа подключения к процессу источника прерываний в конечном счете зависит от конкретных особенностей разрабатываемого ПРВ.

Отключение процесса от прерывания

Когда у процесса отпадает необходимость в обработке некоторого подключенного ранее источника прерываний, он может отключить его, используя функцию:

```
#include <sys/neutrino.h>
int InterruptDetach(int id);
```

Функция отключает прерывание, дескриптор которого, задан в `id`. Когда нить или процесс завершаются, то подключенные ими источники прерываний автоматически отключаются. Если окажется, что это последний процесс, связанный с данным источником прерываний, то ядро автоматически маскирует соответствующий источник прерываний, чтобы запретить прерывания от этого источника.

Управление прерываниями

Все процессы могут управлять подключёнными источниками прерываний на уровне PIC и на уровне процессора. Управление прерываниями на уровне PIC

осуществляется путем маскирования/демаскирования процессом источников прерываний, устанавливая соответствующие значения разрядов регистра IMR. Это предохраняет процесс от нежелательных запросов прерывания, поступающих по замаскированному источнику прерываний. Для этих целей используются функции:

```
#include <sys/neutrino.h>
int InterruptMask(int intr,int id);
int InterruptUnmask(int intr,int id);
```

Функции позволяют маскировать/демаскировать прерывание с номером `intr`, подключённое к процессу с дескриптором `id`, который возвращается функциями `InterruptAttach()` или `InterruptAttachEvent()`. В качестве `id` можно задать –1, если необходимо, чтобы ядро автоматически отслеживало маскирование/демаскирование прерываний каждым обработчиком. Параметр `id` игнорируется, если при подключении обработчика был установлен флаг `_NTO_INTR_FLAGS_TRK_MSK`.

Если необходимо запретить прерывать процессор при выполнении, например, критических участков программного кода, то можно блокировать/деблокировать прерывания на уровне процессора. Для этого используются функции:

```
#include <sys/neutrino.h>
void InterruptLock(intrspin_t* spinlock);
void InterruptUnlock(intrspin_t* spinlock);
```

Функция `InterruptLock()` блокирует, а `InterruptUnlock()` деблокирует прерывания процессора. Для управления прерываниями на уровне процессора с использованием этих функций необходимо предварительно определить переменную системного типа `intrspin_t` и использовать указатель на неё в функциях в качестве параметра `spinlock`, разделяемого обработчиком прерывания и установившей его нитью, предназначенного для согласования их действий блокирования/деблокирования прерывания процессора. Если `spinlock` не является статическим (создан в динамически выделенной памяти), то его перед использованием необходимо проинициализировать с помощью вызова функции `memset(spinlock,0,sizeof(*spinlock))`.

Функция `InterruptLock()` пытается захватить `spinlock`, пока прерывания процессора блокированы. После этого последующее выполнение команд происходит без прерывания. Важно скорее выполнить команды и деблокировать прерывания процессора. Обычно это выглядит так:

```
InterruptLock (&spinner);

/* критическая секция */

InterruptUnlock (&spinner);
```

С помощью функций InterruptLock() и InterruptUnlock() решается общая для многих систем реального времени проблема синхронизации доступа к общим данным между обработчиком прерывания и нитью, его установившей.

Ожидание нитью уведомления о прерывании

Для ожидания уведомления о прерывании типа SIGEV_INTR нить должна вызвать функцию:

```
#include <sys/neutrino.h>
int InterruptWait(int flags,const uint64_t *timeout);
```

В текущей версии flags должен быть равен 0, а timeout должен быть равен NULL. Эта функция переводит нить в INTR-блокированное состояние до момента прихода события типа SIGEV_INTR. Чтобы ограничить время блокировки можно воспользоваться таймаутом ядра. Если уведомление приходит раньше выполнения функции, то функция сразу же успешно завершается.

В случае ошибки функция возвращает −1, при успешном завершении – любое другое значение, отличное от −1.

Общий формат процесса с обработкой прерываний

Общая структура функции main() процесса с обработкой прерываний имеет вид:

```
#include <sys/neutrino.h>
#define IRQ3 3 //Номер подключаемого источника прерывания
...
struct sigevent event;//Уведомление о прерывании
...
void *intr_thread (void *arg);//Нить обработки прерывания

const struct sigevent* isr(void *area, int id);//Функция ISR
...
/********************** main() *********************************/
main(){
/* Выполнить необходимые инициализации и т.п. */
SIGEV_INTR_INIT(&event); //Уведомление типа SIGEV_INTR
...
    /* Запустить нить, выполняющую работу, связанную с прерыванием
       */
pthread_create (NULL, NULL, intr_thread, NULL);
...
/* Продолжить выполнение необходимых действий */
...
}
```

```
/********************** intr_thread () ********************/
/* Эта нить предназначена для выполнения действий, инициируемых
прерыванием */

void * intr_thread (void *arg){
/* Разрешить привилегированный доступ к адресам и портам аппаратуры
*/
ThreadCtl(_NTO_TCTL_IO, NULL);
...
/* Инициализация оборудования и т.п. */
...
/* Подключение ISR к IRQ3 */
InterruptAttach(IRQ3,isr,NULL,0,0);

...
    /* С этого момента может быть целесообразным увеличить приоритет
       нити */
...
/* Теперь ждем уведомлений о прерываниях и выполняем необходимую
обработку */
   while(1){
   InterruptWait(NULL, NULL);
   /* Попадаем сюда, когда InterruptWait() деблокируется в результате
   прихода уведомления типа SIGEV_INTR, сформированного ядром по
   заказу ISR, что говорит о необходимости выполнить соответствующие
   действия */
   ...
   /* Выполняем работу */
   ...
   /* Если в isr() была выполнена InterruptMask(), то нить должна
   выполнить InterruptUnmask(), чтобы разрешить прерывания от
   аппаратуры */
   }
}

/************************ isr() *****************************/
// Это общая структура ISR
/* Объявление переменных, используемые для сохранения значений
регистров и адресов памяти аппаратуры. Они должны объявляться с
модификатором volatile (например, volatile int …), чтобы запретить
компилятору кэшировать значения этих переменных, поскольку они могут
быть изменены в любой точке выполнения процесса */

const struct sigevent* isr(void *area, int id){

...
/* Проанализировать установку запроса прерывания во внешнем
устройстве */
if(/* Признак отсутствует */) return(NULL)/*Не уведомлять нить*/
```

```
/* Сбросить в устройстве запрос прерывания или, по крайней мере,
замаскировать сигнал прерывания в PIC, выполнив функцию
InterruptMask() и запретив повторные прерывания ядра */

/* Возвратить указатель на структуру event, предварительно
проинициализированную в main(), которая определяет тип уведомления
SIGEV_INTR. В этом случае нить, выполнившая вызов InterruptWait(),
будет выведена из состояния ожидания. */
return (&event);
}
```

ПРИЛОЖЕНИЕ

Системные сигналы стандарта POSIX

Стандарт POSIX 1003.1a определяет 32 сигнала, включающие в себя следующие сигналы:

Название	Диспозиция по умолчанию	Событие инициации сигнала
SIGHUP	Завершить	Посылается лидеру сеанса, связанному с управляющим терминалом, когда ядро обнаруживает, что терминал отсоединился (потеря линии). Сигнал также посылается всем процессам текущей группы при завершении выполнения лидера. Сигнал удобно использовать для взаимодействия с демонами. Демон не имеет управляющего терминала и, соответственно, обычно не получает этот сигнал.
SIGINT	Завершить	Сигнал для уведомления процессов текущей группы о терминальном прерывании (пользователь может инициировать нажатием клавиш <Ctrl>+<C>).
SIGQUIT	Завершить+core	Сигнал уведомления процессов текущей группы о нажатии клавиш <Ctrl>+<\>.
SIGILL	Завершить+core	Сигнал уведомления процесса о попытке выполнить недопустимую инструкцию (после перехвата повторно не устанавливается).
SIGTRAP	Завершить	Сигнал уведомления процесса о выполнении им trap-прерывания при трассировке.
SIGIOT	Завершить	Сигнал уведомления процесса о выполнении им IOT-инструкции.
SIGABRT	Завершить+core	Сигнал уведомления процесса о выполнении им системного вызова abort().
SIGEMT	Завершить	Сигнал как реакция ядра на выполнение процессом EMT-инструкция.
SIGFPE	Завершить+core	Сигнал уведомления процесса о возникновении особой ситуации, такой как деление на 0 или переполнение операции с плавающей точкой.
SIGKILL	Завершить	Сигнал завершения процесса. При получении сигнала процесс завершается (не может быть перехвачен или игнорирован).
SIGBUS	Завершить+core	Сигнал уведомления процесса об ошибке шины. Сигнал свидетельствует о некоторой аппаратной ошибке (например, обращение к допустимому виртуальному адресу, для которого отсутствует физическая страница).
SIGSEGV	Завершить+core	Сигнал уведомления процесса о нарушении сегментации. Попытка обращения к недопустимому адресу или к области памяти, для которой у процесса недостаточно привилегий.
SIGSYS	Завершить+core	Сигнал уведомления процесса при попытке недопустимого системного вызова (об использовании ошибочного аргумента в системном вызове).
SIGPIPE	Завершить	Сигнал уведомления процесса о выполненной записи

		в устройство pipe, не открытого для чтения.
SIGALRM	Завершить	Сигнал для уведомления процесса службой часов реального времени.
SIGTERM	Завершить	Сигнал предупреждения, что процесс будет уничтожен. Позволяет процессу подготовиться к завершению.
SIGUSR1	Завершить	Сигнал, не инициируемый ядром, а только пользователем или прикладным процессом.
SIGUSR2	Завершить	Сигнал, не инициируемый ядром, а только пользователем или прикладным процессом.
SIGCHLD	Игнорировать	Сигнал для уведомления родительского процесса о завершении дочернего процесса.
SIGPWR	Игнорировать	Сигнал для уведомления процесса об угрозе потери питания. Обычно он отправляется, когда питание системы переключается на источник бесперебойного питания (UPS).
SIGWINCH	Игнорировать	Сигнал для уведомления процесса об изменении окна
SIGURG	Игнорировать	Сигнал для уведомления процесса о срочном событии на канале ввода-вывода
SIGPOLL	Завершить	Сигнал для уведомления процесса о наступлении определенного события для устройства, которое является опрашиваемым.
SIGIO	Игнорировать	Сигнал уведомления асинхронного ввода-вывода.
SIGSTOP	Остановить	Сигнал остановки, инициированный не с устройства tty (не может быть перехвачен или игнорирован).
SIGTSTP	Остановить	Сигнал остановки инициированный, с устройства tty.
SIGCONT	Продолжить	Сигнал запуска остановленного процесса для продолжения выполнения.
SIGTTIN	Остановить	Сигнал для уведомления процесса фоновой группы о попытке фонового чтения с управляющего терминала tty.
SIGTTOU	Остановить	Сигнал для уведомления процесса фоновой группы о попытке фоновой записи на управляющий терминал tty.

В дополнение к перечисленным сигналам существуют 24 сигнала, используемых службой реального времени (POSIX 1003.1b). Номер первого сигнала реального времени - SIGRTMIN, номер последнего сигнала реального времени. - SIGRTMAX.

Весь диапазон сигналов предполагает 64 сигнала. Диапазон начинается с _SIGMIN (равен 1) и заканчивается _SIGMAX (64).

Заметим, что нельзя ни игнорировать, ни перехватить сигналы SIGKILL или SIGSTOP. Для них всегда выполняется действие по умолчанию. Кроме того, при завершении процесса по умолчанию в результате прихода сигнала в ряде случаев в текущем каталоге процесса создается файл **core** (в таблице такие случаи отмечены как "Завершить+core"), в котором храниться образ памяти процесса. Этот файл может быть проанализирован программой-отладчиком для определения состояния процесса непосредственно перед завершением.

Федеральное агентство по образованию

**Государственное образовательное учреждение
высшего профессионального образования
"Самарский государственный аэрокосмический
университет имени академика С.П. Королёва"**

А.В. Баландин

Средства и методы разработки программных систем реального времени

Часть 4

СТРУКТУРНЫЙ АНАЛИЗ, МОДЕЛИРОВАНИЕ И МАКЕТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Учебное пособие

**Самара
2012**

УДК 681.3.066
ББК 32.973.26-018.2

Баландин А.В. Средства и методы разработки программных систем реального времени. Учебное пособие: В 4 ч. Ч.4. **Структурный анализ, моделирование и макетирование программных систем реального времени**. - Самар. гос. аэрокосм. ун-т. Самара, 2012. 56 с.

В пособии рассмотрен метод многоуровневого структурного анализа, графо-аналитического моделирования и макетирования распределённых программных систем обработки данных и управления в режиме реального времени. Метод ориентирован на использование базовых средств операционных систем реального времени для параллельного распределённого и многонитиевого (многопоточного) программирования.

Пособие предназначено для студентов, обучающихся по направлению 010300.68 «Фундаментальная информатика и информационные технологии», изучающих дисциплину «Технологии промышленного программирования».

ОГЛАВЛЕНИЕ

ЧАСТЬ 4. СТРУКТУРНЫЙ АНАЛИЗ, МОДЕЛИРОВАНИЕ И МАКЕТИРОВАНИЕ РАСПРЕДЕЛЕННЫХ ПРОГРАММНЫХ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Введение

Системы компьютерного мониторинга относятся к классу систем реального времени (СРВ), для которых важным условием их функционирования является способность контролировать физические параметры и состояния объекта управления в режиме реального времени. При этом эффективность функционирования СРВ одновременно зависит как от выбранной структуры и характеристик технических средств, так и от качества разработки программного обеспечения СРВ. Иными словами время реакции системы складывается из времени выполнения своих функций техническими средствами и программным обеспечением.

Этап проектирования и создания технических средств СРВ выражается в выборе и интеграции в единый комплекс оборудования, необходимого для реализации всех функций, определённых для системы в техническом задании. Выбор оборудования обычно предполагает анализ широкой номенклатуры устройств, имеющихся на рынке технических средств автоматизации, с учетом их стоимости, характеристик функционирования, надежности и т.п. Лишь в исключительных случаях, когда на рынке технических средств автоматизации не оказалось приемлемых устройств, может возникать необходимость в проектировании и разработке уникального оборудования. В итоге, когда структура технических средств СРВ сформирована (включая и сетевые средства), она становится основой для разработки программного обеспечения СРВ. При этом априори предполагается, что технические средства оставляют для программного обеспечения достаточно времени для реализации своих функций в срок, чтобы вся система функционировала в режиме реального времени.

В общем случае, ПО СРВ является сложной программной системой. В отличие от способа проектирования технических средств СРВ проектирование и разработка программной системы базируется на использовании хотя и достаточно широкого набора, но все-таки более "аморфных" средств разработки ПО СРВ, которые в меньшей степени ограничивают "фантазии" разработчиков при выборе структуры программной системы.

Как правило, на начальном этапе перед разработчиками ПО СРВ стоит проблема выбора метода (парадигмы) структуризации ПО, используя который осуществляется анализ и синтез структуры программной системы. При этом обычно полагают, что оценить априори временн'ые характеристики разрабатываемых программных компонентов и программной системы в целом нельзя, а следовательно, оценить их влияние на эффективность функционирования СРВ в режиме реального времени, либо просто не возможно, либо можно только со значительной степенью неопределённости. Одновременно считается, что у разработчика программной системы всегда есть возможность впоследствии целенаправленно влиять на временн'ые характеристики,

модифицируя структуру как отдельных программных компонентов, так и структуру программной системы в целом.

Используемые в настоящее время средства разработки ПО компьютерного мониторинга (SCADA-системы), ориентированы, прежде всего, на автоматизацию программирования и повышение производительности работ по созданию программных систем. То есть, преследуется цель получения в максимально короткие сроки надежного (без семантических ошибок) программного кода. Однако в них на этапе проектирования и разработки структуры программной системы практически не представлены специальные методы и средства оценки и контроля возможности функционирования проектируемой программной системы в рамках установленных временных ограничений. Для распределенных СРВ ситуация усугубляется ещё и тем, что крайне сложно на этапе проектирования и разработки программной системы учесть влияние на временн'ые характеристики затраты, связанные с коммуникацией данных. Поэтому обычно полагают, что оценить способность функционирования СРВ в режиме реального времени (а в его составе и ПО) можно только практически путём тестирования СРВ, некоторым образом моделируя воздействия на него внешнего окружения (объекта и субъекта управления). В результате тестирования выявляются "узкие места" функционирования СРВ, которые устраняются либо заменой соответствующих компонентов технических средств более производительными элементами, либо "доведением" (модификацией) ПО СРВ. При этом более предпочтительным является устранение узких мест путём модификации программной составляющей СРВ.

Очевидно, что эффективность работ по модификации программной составляющей СРВ зависит от наличия в выбранном методе проектирования и разработки ПО возможности функционально-эквивалентных преобразований программной системы. Функциональная эквивалентность понимается в том смысле, что очередной вариант программной системы сохраняет семантику функционирования, но при этом временн'ые характеристики разных вариантов ПО СРВ будут разными. В результате задача устранения узких мест функционирования СРВ сводится к задаче нахождения во множестве функционально-эквивалентных вариантов программной составляющей СРВ варианта, обеспечивающего режим реального времени.

При проектировании и разработке СРВ приходится рассматривать различные уровни поведения системы и решать комплекс разноплановых, но взаимоувязанных задач функционирования системы. Из этого следует, что СРВ относится к классу сложных систем. В теории сложных систем показано, что их анализ и синтез должен осуществляться на основе *иерархического принципа* организации системы [1]. Согласно этому принципу СРВ следует представлять в виде многоуровневой системы, структурное моделирование которой необходимо осуществлять на различных, взаимосвязанных уровнях абстракции. Наиболее последовательно иерархический принцип для многоуровневого моделирования *вычислительных многокомпьютерных управляющих систем*, работающих в режиме реального времени (т.е. АСРВ), был применён и описан Гертенбахом

(Gertenbach W.P.) в [2]. В этой работе, на основе обобщения опыта построения и использования вычислительных управляющих систем в области ядерных исследований, введено в рассмотрение семейство иерархических моделей СРВ, описывающих систему в различных аспектах. Предложен подход к формированию иерархической структуры СРВ, базирующийся на анализе функций, реализуемых системой, и выделении функциональных уровней (стратумов) и уровней принятия решений (слоёв). Сформированная таким образом функциональная структура СРВ используется для обоснования состава и структуры комплекса технических средств и последующего распределения выделенных функций по компонентам (модулям) технических средств вычислительного многокомпьютерного управляющего комплекса. При этом функции, возложенные на процессоры, должны быть программно реализованы и специфицируют семантику функционирования ПО СРВ и. Однако в работе на этом всё и заканчивается. Никаких принципов моделирования структуры ПО СРВ для программной реализации отнесённых к процессорам функций не предлагается. Учитывая, что программная составляющая СРВ, реализуемая процессорами, в свою очередь сама является сложной программной системой, то её проектирование и разработку необходимо также осуществлять на основе многоуровневого моделирования структуры ПО СРВ в виде взаимосвязанных моделей различных уровней абстракции и принятия решений (управлений). При этом структурная организация и способ реализации ПО АСРВ должны обеспечивать возможность эффективно осуществлять функционально-эквивалентные преобразования программной части АСРВ.

Многоуровневая модель АСРВ

Общая структура СРВ

В самом общем виде структуру системы реального времени (СРВ), в состав которой входит АСРВ, можно представить так, как показано на Рисунок 1. На рисунке представлены четыре компоненты СРВ: *объект управления* (ОУ), *субъект управления* (СУ), *АСРВ* и *хранилище данных*. Объект управления, субъект управления и хранилище данных составляют *окружение* АСРВ, с которым АСРВ взаимодействует в режиме реального времени.

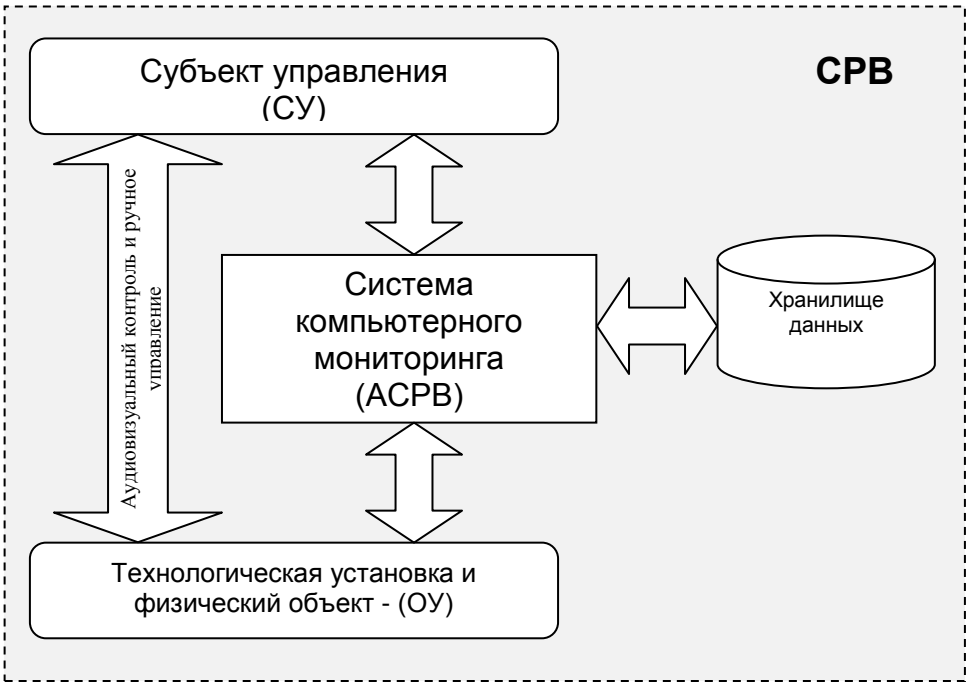


Рисунок 1 Общая структура системы реального времени

Объект управления (ОУ) – это абстракция, специфицирующая технологическую установку и физический объект как единое целое, предназначенное для реализации технологического процесса. В общем случае ОУ может иметь сложную топологию и пространственную распределённость. Объект управления имеет множество физических параметров, характеризующих его состояние во времени, и множество параметров регулирования для управления состоянием ОУ. Для снятия (измерения) значений физических параметров ОУ оснащается датчиками. Для осуществления регулирующих (управляющих) воздействий на ОУ он оснащается регуляторами (исполнительными механизмами). Кроме того, ОУ может оснащаться устройствами отображения значений параметров состояния и значений регулировок, предназначенными непосредственно человеку-оператору для организации его взаимодействия с объектом управления (аудиовизуальный контроль состояния и ручное регулирование).

Таким образом, объект управления в общем случае представляется как распределённая пространственно-топологическая структура устройств сбора,

отображения данных и устройств регулирования физических параметров на объекте управления.

Субъект управления (СУ) – это абстракция, специфицирующая человека-оператора (в общем случае - группу операторов), управляющего технологическими процессами на ОУ. Функционирование субъекта управления заключается в мониторинге отображаемых значений физических параметров и управлении состоянием ОУ непосредственно или через АСРВ путём регулирования физических параметров объекта управления посредством исполнительных механизмов. Субъект управления может так же, как и объект управления, иметь собственную распределённую пространственно-топологическую структуру, не совпадающую с пространственно-топологической структурой ОУ.

Хранилище данных – это абстракция, специфицирующая память, предназначенную для организованного накопления и хранения результатов мониторинга состояния и регулирования ОУ, а также результатов их обработки. В качестве таких результатов могут выступать тренды измеряемых параметров состояния, значения регулировок параметров технологической установки и физического объекта, режимы управления ОУ и т.п. Кроме того, хранилище данных предназначено и для хранения базы данных с системной информацией, описывающей непосредственно АСРВ как предметную область. Например, топологию объекта и субъекта управления и размещение устройств комплекса технических средств и распределённого вычислительного комплекса, топологию размещения и типы датчиков и регуляторов и т.д.

Иерархия АСРВ по уровням абстракции объекта управления

Из анализа целей и задач функционирования АСРВ, а также из опыта построения и использования вычислительных многокомпьютерных управляющих систем, работающих в режиме реального времени [2], выявлено, что АСРВ адекватно описывается иерархической моделью, специфицирующей систему на пяти уровнях абстракции (стратумах) объекта управления (Рисунок 2).

На каждом из пяти стратумов АСРВ рассматривается как абстрактная система управления, осуществляющая выполнение соответствующих стратуму набора функций, преобразующих входные данные, получаемые с ОУ, в выходные, отправляемые объекту управления. Набор входных и выходных данных, а также порядок их получения и отправки составляют интерфейс (логический или физический) взаимодействия АСРВ с абстракцией ОУ на этом стратуме.

Взаимосвязь между уровнями понимается так, что относительно некоторого выделенного для рассмотрения стратума все нижележащие уровни обеспечивают на данном уровне логику функционирования и управления объектом на этом стратуме. Для уровня находящегося непосредственно над выделенным, выделенный уровень обеспечивает интерфейс управления ОУ для выше лежащего уровня.

Контекст, в котором рассматриваются входные и выходные данные (абстракция ОУ) на каждом из выделенных уровней следующий:

- На *первом уровне* (электронная система) абстракция ОУ выражается в виде аналоговых или цифровых сигналов, являющихся результатами измерения датчиками физических параметров состояния ОУ или значений параметров регулирования технологической установки.

- На *втором уровне* (система ввода/вывода) абстракция ОУ выражается в виде адресов физической памяти (портов) устройств связи с ОУ, предназначенных для обмена данными с ОУ в виде двоичных входных или выходных слов.

- На *третьем уровне* (система сбора данных и регулирования) абстракция ОУ выражается в виде контролируемых параметров текущего состояния ОУ и параметров регулирования технологической установки и физического объекта, обеспечивающие логику управления технологическими процессами на ОУ. Параметры являются переменными абстрактного типа.

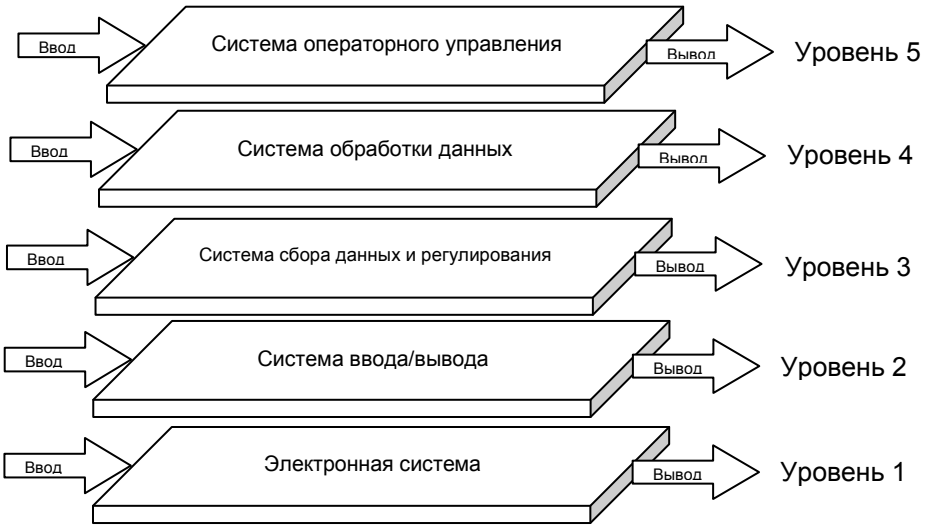


Рисунок 2 Стратумная модель АСРВ

- На *четвёртом уровне* (система обработки данных) абстракция ОУ выражается в виде логической модели организации данных, отражающих состояние ОУ на этом уровне, и структуры функций обработки данных логической модели. Результатами обработки данных являются значения абстрактных характеристик ОУ, предназначенные для передачи на следующий (пятый) уровень, или для анализа и выработки управляющих воздействий на абстрактные параметры регулирования ОУ на этом уровне.

- На *пятом уровне* (система операторного управления) абстракция ОУ рассматривается с точки зрения оператора системы - СУ. Взаимодействие СУ с ОУ выражается в аудиовизуальном восприятии оператором параметров ОУ и технологических процессов с устройств отображения и ручном вводе значений регулировок.

Иерархическая связь между уровнями выражается в том, что каждый верхний уровень структурно интегрирует и инкапсулирует данные и результаты операций нижних уровней, отображая их в структуры входных/выходных параметров и операции своего уровня. Каждый нижний уровень наоборот

осуществляет структурную декомпозицию входных/выходных данных и операций верхнего уровня.

Иерархия АСРВ по уровням управления

Стратумное моделирование АСРВ определяет на каждом из пяти стратумов абстракцию ОУ в виде пары множеств $\langle \bar{x}_l, \bar{u}_l \rangle$, $\bar{x}_l$ - множество контролируемых параметров состояния ОУ, и $\bar{u}_l$ - множество параметров регулирования ОУ, l=1,2,3,4,5. В тоже время она не специфицирует АСРВ как систему управления, т.е. аспекты, связанные непосредственно с управлением или регулированием ОУ, в ней не отражены, она не устанавливает связи между изменяющимися во времени значениями контролируемых параметров состояния ОУ и необходимыми при этом значениями регулировок ОУ (контуры управлений). Для этого, в соответствии с теорией сложных систем, АСРВ должна рассматриваться как сложная *система принятия решений* (т.е. управляющая система).

Для моделирования АСРВ как управляющей системы, её необходимо описать на каждом стратуме как систему управления ОУ (контроль/регулирование) так, чтобы общая модель управления ОУ разложилась на пять соподчиненных моделей, специфицирующих процедуры контроля/регулирования ОУ на каждом уровне стратумной модели АСРВ. Согласованное управление ОУ на каждом уровне в итоге обеспечивает для АСРВ управление ОУ в целом. Чем выше уровень абстракции ОУ, тем большую долю в управлении составляют процессы принятия решений. Чем ниже уровень абстракции ОУ, тем больше управление на этом уровне выражается в непосредственном получении значений параметров состояния ОУ и реализации регулировок (управляющих воздействий), полученных с верхних уровней.

Модель АСРВ как системы принятия решений и управления определим, как показано на Рисунок 3. Из рисунка видно, что АСРВ рассматривается как единый элемент принятия решений и управления. Задавая значения различных уставок/регулировок, оператор может вносить корректировки в режимы работы системы и следить за результатами работы системы, контролируя параметры объекта управления в самом общем виде. АСРВ полностью берёт на себя оперативное принятие решений и выдачу управляющих воздействия на объект. Относительно оператора, как субъекта управления, АСРВ играет роль системы отображения состояния объекта управления и задания оператором уставок/регулировок.

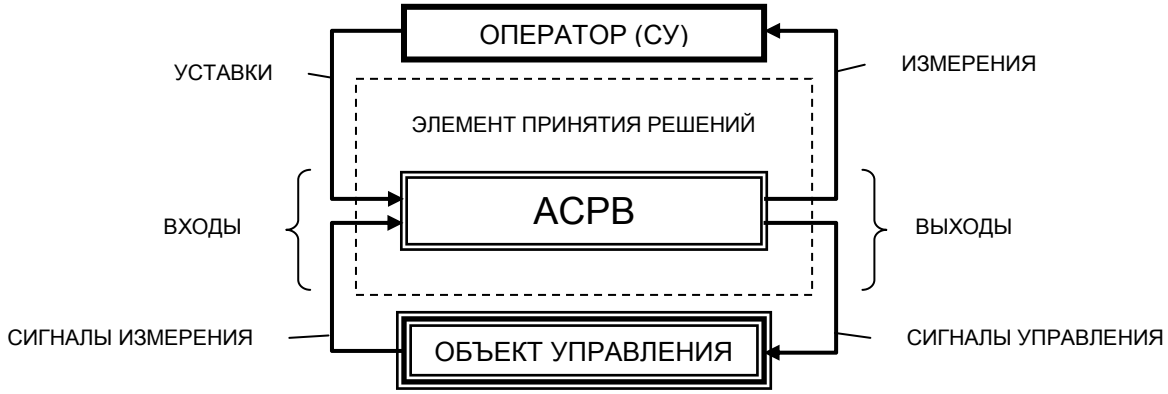


Рисунок 3 Модель АСРВ как системы принятия решений и управления

Для разбиения АСРВ по уровням принятия решений введём в рассмотрение элемент принятия решений (см. Рисунок 4):

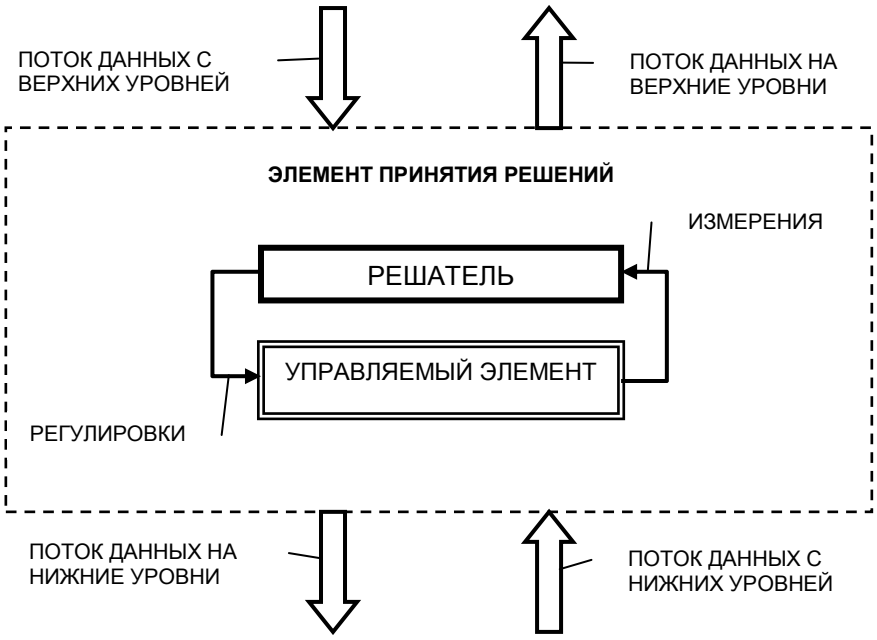


Рисунок 4 Модель элемента принятия решений

Теперь определим элементы принятия решений для каждого из уровней абстракции АСРВ, как показано на Рисунок 5.

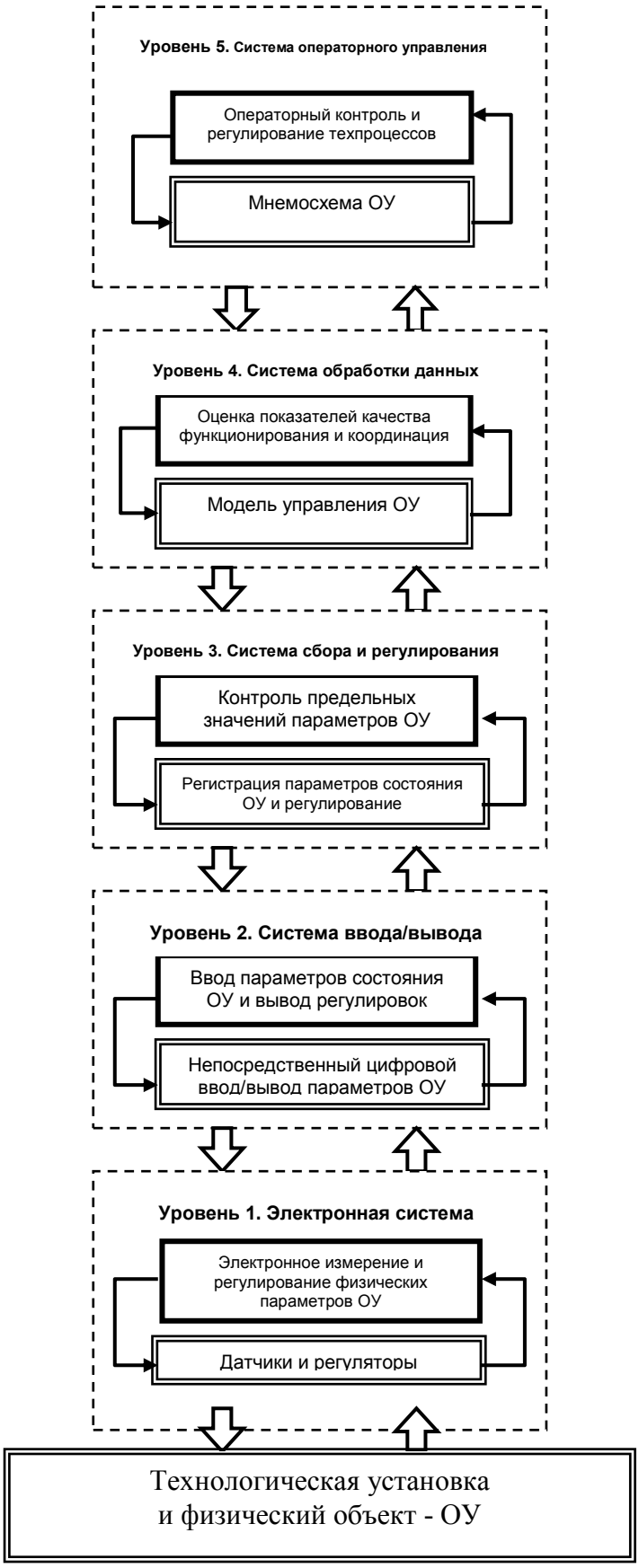


Рисунок 5 Иерархическая модель уровней управления в АСРВ

Уровень 1. На уровне электронной системы осуществляется электронное управление измерением значений физических параметров ОУ и регуляторами для формирования значений физических параметров управления ОУ.

Уровень 2. На уровне системы ввода/вывода осуществляется управление вводом значений физических параметров, а также выводом регулировок, представленных в цифровом виде.

Уровень 3. На уровне системы сбора данных и регулирования осуществляется управление сбором данных, а также выводом регулировок, обеспечивающих заданный диапазон значений параметров ОУ.

Уровень 4. На уровне системы обработки данных осуществляется оценка показателей качества функционирования системы и выработка управляющих воздействий для координации техпроцессов.

Уровень 5. На уровне системы операторного управления осуществляется операторный контроль параметров и ручное регулирование техпроцессов.

Уровни организации данных

Организация данных является сложной задачей проектирования и разработки любой информационной системы и АСРВ в том числе. Уже давно теоретически исследована и эффективно используется в качестве основного инструмента организации данных и управления данными при построении автоматизированных информационных систем (АИС) концепция баз данных и СУБД. Однако, как это на первый взгляд ни парадоксально, системы реального времени не ограничиваются использованием СУБД для организации и управления данными как системообразующим инструментом. Прежде всего, это определяется тем, что управление данными осуществляются в режиме реального времени и это накладывает на время реакции СУБД жесткие ограничения. Кроме того, с позиции АИС база данных и СУБД – это способ организации хранения, накопления и управления данными, который позволяет эффективно удовлетворять по запросу изменяющиеся информационно-аналитические потребности пользователя системы. В таком качестве концепция базы данных и СУБД, как основы разработки системы, не вполне подходит для АСРВ. Во-первых, в АСРВ порой просто нет необходимости в хранении и накоплении получаемых данных. Это связано, например, с тем, что данные, поступающие с ОУ, необходимы только для отражения его текущего состояния и выработки в реальном времени регулирующих воздействий на ОУ. После того, как полученные с ОУ данные проанализированы и заданы соответствующие регулировки, сами данные уже не нужны (отработаны) и их можно либо "выкинуть в урну", либо занести в архив для последующего использования, но уже не в режиме реального времени. Во-вторых, в отличие от АИС, в АСРВ в режиме реального времени никакого априори не запланированного анализа данных быть не может, а наоборот использование данных заранее спланировано и строго регламентировано моделью управления. Поэтому организация и управление данными в АСРВ не может быть целиком реализована путём выбора некой одной СУБД и её использования на всех уровнях рассмотрения АСРВ, а сам принцип отделения данных от программ обработки, лежащий в основе

концепции баз данных и СУБД, на разных уровнях абстракции АСРВ реализуется по-разному с использованием разных моделей организации данных и методов управления данными.

Рассмотрим особенности организации данных и управления данными на различных уровнях стратумной модели АСРВ.

Уровень электронной системы

На уровне электронной системы модель организация данных выражается перечнем и топологией размещения на ОУ датчиков и регуляторов, а также их связями с соответствующими модулями УСО системы ввода/вывода. В качестве "типов данных" выступают типы аналоговых или цифровых сигналов, формируемых датчиками, или сигналов регулировок, формируемых системой ввода/вывода и посредством УСО направляемых регуляторам (исполнительным механизмам). Таким образом, в качестве своеобразной "*сигнальной базы данных*" выступает в комплексе с датчиками и регуляторами непосредственно сам объект управления. Считывание "данных" осуществляется датчиками, а модификация – регуляторами.

Уровень системы ввода/вывода

На уровне системы ввода/вывода "сигнальная база данных" преобразуется в "*физическую цифровую базу данных*", представленную адресами физической памяти (портов) устройств связи с ОУ, предназначенными для ввода/вывода данных в виде двоичных входных или выходных слов. Логическая модель такой базы данных выражается парой $\langle P_{in}, P_{out} \rangle$, где $P_{in} = p_i^{in}{}_{i=1,2,...,n}$ - множество из n логических входных портов, предоставляющих результаты преобразования значений физических параметров ОУ в цифровой вид, и $P_{out} = p_j^{out}{}_{j=1,2,...,m}$ - множество из m логических выходных портов, принимающих значения регулировок ОУ в цифровом виде. Систему ввода/вывода можно рассматривать в качестве своеобразной СУБД, обеспечивающей чтение значений из P_{in} и запись значений в P_{out}. При этом изменение значений в P_{in} осуществляется в результате изменения физического состояния ОУ, которое в свою очередь зависит от изменения значений регулировок в P_{out}. Типы данных, используемых в $\langle P_{in}, P_{out} \rangle$, определяются свойствами соответствующих элементов памяти (регистров, ячеек) устройств связи с объектом.

Организация доступа к параметрам ОУ на уровне системы ввода/вывода выражается в виде коммуникации данных между регистрами/ячейками памяти устройств связи с объектом и буфером ввода/вывода программы (драйвера), выполнившей запрос (команду) чтения/записи. Этот уровень АСРВ является пограничным между ПО АСРВ и объектом управления.

Уровень системы сбора данных и регулирования

На уровне системы сбора данных и регулирования логическая модель ОУ как "базы данных" выражается парой $\langle \mathrm{M}_S, \mathrm{M}_D \rangle$, где $\mathrm{M}_S = \mathrm{m}_i^S{}_{i=1,2,...,k}$ - множество из k абстрактных объектов памяти m_i^S (*логических истоков*), содержимое которых

изменяется во времени, характеризуя изменение во времени состояния ОУ, а $M_D = m_j^D{}_{j=1,2,...,l}$ - множество из l абстрактных объектов памяти m_j^D (*логических стоков*), содержимое которых формируется во времени в результате обработки (например, контроля по уставкам) содержимого истоков. Часть стоков ассоциируется с абстрактными регулировками ОУ, а часть – с абстрактными параметрами ОУ, формируемыми на уровне сбора данных.

Для организации и управления данными на уровне системы сбора данных и регулирования предназначены так называемые SCADA-системы (SCADA - Supervisory Control And Data Acquisition) – это профессиональные инструменты разработки систем компьютерного мониторинга (сбора данных и регулирования). SCADA-системы позволяют методом сборки из крупных библиотечных программных модулей быстро построить надёжную систему ввода/вывода, обеспечивающую считывание оцифрованных данных. Сложность, однако, заключается в том, что различные фирмы-разработчики аппаратных средств ввода/вывода используют собственные (не стандартизованные) коммуникационные протоколы управления устройствами. Поэтому соединение в единую систему ввода/вывода разнообразных устройств приводит к необходимости использования в системе различных драйверов, поддерживающих те или иные протоколы контроллеров. Фирмы–изготовители SCADA-систем предлагают либо готовые драйверы для контроллеров различных типов, либо руководства по написанию драйверов и нескольких программных заготовок на языке С.

Кроме ввода/вывода SCADA-системы предоставляют программные средства предварительной обработки данных, средства отображения данных оператору с помощью различных устройств вывода, а также запись и долговременное хранение данных в архиве в виде трендов. Доступ к архиву предполагает возможность контекстного поиска и просмотра данных в различных формах, а также генерации отчётов.

Уровень системы обработки данных

Организация данных и управление данными на уровне системы обработки данных преследует цель обеспечить эффективное выполнение многочисленных функций обработки данных, отражающих сложную семантику анализа критериев функционирования ОУ и выработки оптимального управления. В этом случае логической моделью, адекватно описывающей изменение во времени параметров объекта управления $P(t)$ на уровне системы обработки данных, являются *тренды* - потоки датированных данных (*хронологическая модель*), которые являются дискретной моделью изменяющихся во времени значений параметров $P(t)$. Формально модели трендов будут описаны в разделе, посвящённом структурному анализу и моделированию ПО АСРВ.

Использование СУБД

Теоретически СУБД может использоваться как для управления статическими данными, так и для управления данными в реальном времени. В первом случае

СУБД используются для управления базой данных, являющейся статической информационной моделью объекта управления. Наличие такой базы данных в АСРВ прежде всего обуславливается необходимостью хранить в ней спецификации множества параметров различного типа, характеризующих объект управления и режимы управления. Это могут быть, например, типы и количества датчиков и регуляторов, данные о топологии размещения датчиков и регуляторов, перечень измеряемых параметров и их свойства, коэффициенты градуировочных характеристик измерительных каналов и т.п. В таком аспекте СУБД и база данных не используются в реальном времени, а предназначена для хранения и получения общесистемной информации, параметрах комплекса технических средств и параметрах объекта управления, режимах работы и т.п. Эта информация используется для параметрической настройки АСРВ.

Концепция базы данных и использование СУБД (обычно реляционных) для управления ОУ в реальном времени, в общем случае, не эффективно. Реляционная модель даже теоретически не приспособлена для описания трендов, что делает возможности реляционных СУБД по управлению датированными данными крайне ограниченными. Имеются, однако, попытки разработки и применения СУБД, позволяющих управлять потоками датированных данных. При этом ядро СУБД интегрируется с ОСРВ и, при этом, обеспечивает высокое быстродействие и гарантированные предельные значения выполнения запросов к базе данных.

В качестве примера такой СУБД можно назвать СУБД Empress (www.swd.ru, www.empress.ru), которая предназначена для встраиваемых систем и систем реального времени. СУБД Empress оптимизирована для высокой производительности и детерминизма (время ответа на запрос варьируется в пределах до 10 мс), имеет простой, но мощный API, обладает функциональностью для встраиваемого применения и высокой надежностью. Кроме того, обеспечивает одновременный множественный и сетевой доступ к БД. Размещение БД осуществляется как на диске, так и в резидентной памяти. Обеспечивается экспорт данных в стационарную СУБД Empress. Для СУБД Empress[1] характерно минимальное потребление ресурсов, малый размер кода, минимальное использование памяти, возможность полностью настроить ядро под нужды программной системы. Размер ядра СУБД Empress варьируется от 550 Кб до 1 Мб в зависимости от типа сборки ядра. Размер пустой базы данных - 203 Кб.

[1] СУБД Empress реализована для работы в операционных системах: AIX, Bluecat, FreeBSD, HP-UX, IRIX, Linux, Linux PPC, Lynx O/S, QNX 4, QNX 6, Red Hat, RTLinux, SCO, Solaris, SUN O/S, SUSE, Tru64 UNIX, WIN 2000, WIN NT, WIN XP.

Многоуровневая модель ПО АСРВ

Общая структура ПО АСРВ

Стратумная модель АСРВ не указывает чётких границ, отделяющих программную часть системы от программно-управляемых устройств комплекса технических средств АСРВ. Очевидно, что на каждом стратуме присутствуют как программная, так и аппаратная составляющие АСРВ. Причем аппаратная составляющая обеспечивает непосредственное или опосредованное взаимодействие программной части с объектом управления, субъектом управления и хранилищем данных. В качестве элементов программной системы, предназначенных для взаимодействия со своим *окружением* (остальной частью АСРВ) выступают абстрактные объекты памяти (порты ввода/вывода) – *истоки*/*стоки*, осуществляющие коммуникацию данных состояния, регулировок между ОУ и программной системой, между программной системой и СУ, а также между программной системой и хранилищем данных (архивом). Выделение и спецификация истоков и стоков является предметом моделирования данных программной системы на стратуме ввода/вывода и стратуме операторного управления АСРВ. В итоге спецификация истоков и стоков составит границу между программной системой и её окружением. Вычленение программной системы из АСРВ вместе с истоками и стоками представим в виде, показанном на рисунке 6.

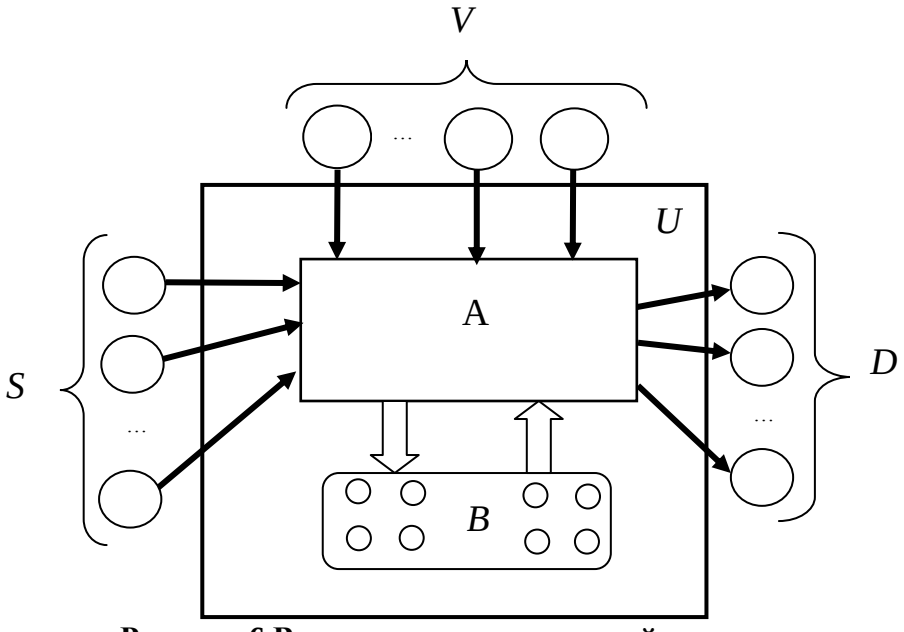


Рисунок 6 Вычленение программной системы

Формально программная система, как вычленение АСРВ, описывается кортежем U=<M,A>, где =$S \cup V \cup D \cup B$ - конечное множество абстрактных объектов памяти программной системы U=<M,A>, A - программный агрегат, получающий входные данные из истоков, и помещающий выходные данные в стоки. Подмножество ($S \cup V \cup D$)$\subset$M - *внешняя память* (окружение) программного агрегата A, $S = s_1, s_2, \ldots, s_{N_S}$ и $V = v_1, v_2, \ldots, v_{N_v}$ – множества истоков (N_S – мощность множества

S, N_v – мощность множества V), $D= d_1, d_2, \ldots, d_{N_D}$ – множество стоков (N_D – мощность множества D). Подмножество $B \subset$ М - внутренняя память программного агрегата А. Разница между истоками из S и истоками из V в том, что истоки из S ассоциируются с параметрами состояния ОУ, а истоки из V, ассоциируются с установочными параметрами программного агрегата А (служебная информация). Стоки в D ассоциируются с регулировками исполнительных механизмов ОУ или элементами отображения информации для СУ.

Программный агрегат ℵ в структуре обеспечивает функционирование АСРВ на двух стратумах - стратуме сбора данных и регулирования и стратуме обработки данных. Функционирование АСРВ на остальных стратумах реализуются аппаратными средствами. Структура программного агрегата ℵ на каждом из этих стратумов формируется относительно независимо друг от друга и отражает специфику функционирования системы на рассматриваемом стратуме.

Существуют различные инструментальные средства и методы разработки программных систем. Их выбор во многом определяет парадигму структуризации при проектировании и разработке программной системы. Если в качестве инструментальных средств программирования используется SCADA-система (специализированное средство для разработки программных систем компьютерного мониторинга), то структура программной системы на этапе проектирования формируется средствами выбранной SCADA-системы, а программный код затем в значительной части генерируются автоматически. На долю разработчика выпадает в основном конфигурирование и настройка программной системы, используя инструментальные средства SCADA-системы (реальное время специфицируется формированием кадра данных за заданное время – темп работы системы), последующая её адаптация к специфическим особенностям используемого УСО, разработка графического пользовательского интерфейса и параметрическая настройка системы в целом.

Иное дело, если в качестве программных инструментов используются более универсальные средства программирования. В этом случае разработка программного обеспечения начинается с выбора принципа структуризации, который определяет затем выбор методов структурного анализа, моделирования и макетирования программной системы.

В дальнейшем разработку программной системы компьютерного мониторинга будем рассматривать при следующих условиях:

- программная система реализуется на базе распределенного вычислительного комплекса (РВК), представляющего собой компьютерную или промышленную сеть, включающую в свой состав устройства ввода/вывода для связи с объектом управления и оператором. Одномашинный вариант рассматривается как частный случай РВК;
- функционирование программной системы осуществляться в реальном времени и в заданных временных ограничениях;
- в качестве инструментальных средств разработки программной системы используются: многопроцессная операционная система реального времени с

многонитиевой организацией вычислений внутри процессов и с клиент-серверной организацией взаимодействия процессов посредством сообщений; специализированные драйверы УСО и файлы устройств ввода/вывода; библиотека функций математической обработки; библиотека классов графических объектов для изображения мнемосхем оператора; средства архивирования данных (файловая система ОС, СУБД).

- в качестве базового средства программирования используются язык С и системные средства программирования ОСРВ (библиотека функций).

Стратумная модель ПО АСРВ

Программная система компьютерного мониторинга (ПСКМ) - U=<M,A>, является сложной системой. Поэтому согласно принципу построения сложных систем в самом общем виде структуру программного агрегата А необходимо представить в виде иерархии уровней абстрагирования (стратумов) с соответствующими каждому стратуму задачами проектирования и разработки ПСКМ. На каждом стратуме программную систему будем представлять в виде программного агрегата, извлекающего данные из истоков и отправляющего результаты работы в стоки.

Представим стратумную модель программной системы компьютерного мониторинга в виде следующих уровней иерархии (см. Рисунок 7):

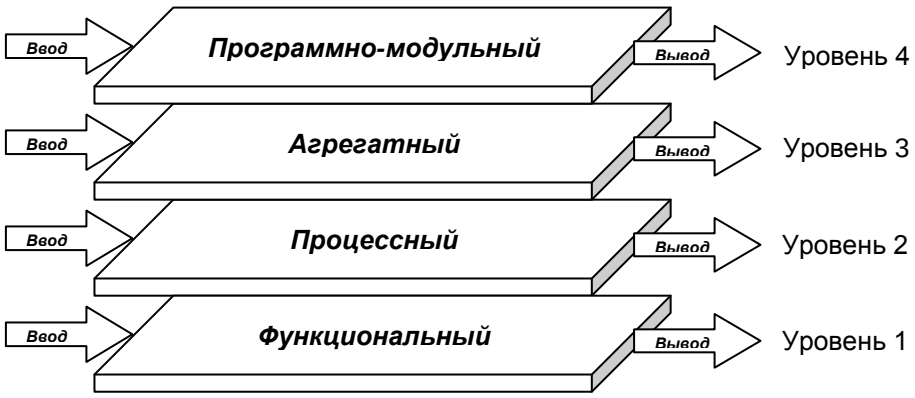


Рисунок 7 Стратумная модель ПО АСРВ

- функциональный,
- процессный,
- агрегатный,
- программно-модульный.

Функциональный стратум предназначен для моделирования функциональной структуры программной системы на рассматриваемом уровне АСРВ. Для этого осуществляется анализ требований к функционированию СРВ, изложенных в техническом задании на разработку системы (ТЗ на СРВ, ТЗ на АСРВ, ТЗ на ПСКМ).

На функциональном стратуме целевая функция стратума декомпозируется в структуру взаимосвязанных по данным неделимых функций, отражающих семантику обработки данных, принятия решений и выдачи управляющих воздействий на объект управления, а также отображения информации или занесения результатов обработки в хранилище данных (архив) на соответствующем уровне АСРВ.

Процессный стратум предназначен для моделирования структуры ПСКМ в виде совокупности взаимодействующих параллельных процессов, обеспечивающих выполнение функций, представленных на функциональном стратуме.

Агрегатный стратум предназначен для моделирования программной системы в виде структуры взаимодействующих абстрактных агрегатов, интегрирующих в себе компоненты структуры процессного стратума. Программные агрегаты рассматриваются в качестве неделимых программных элементов ПСКМ, загружаемых в процессорные компоненты РВК.

Программно-модульный стратум предназначен для представления программной системы в виде структуры каталогов с файлами, содержащими программные модули ПО АСРВ: исполняемые модули, объектные модули, библиотечные файлы и т.п. На программно-модульном стратуме решается задача размещения программных модулей ПО АСРВ на внешних носителях РВК по компонентам сетевой файловой системы.

В соответствии с введёнными уровнями абстракции процесс проектирования программной системы предполагает решение следующих задач:

1. Структурный анализ и моделирование функциональной структуры программной системы компьютерного мониторинга на рассматриваемом стратуме АСРВ.
2. Синтез процессной структуры программной системы в виде совокупности взаимодействующих параллельных процессов, реализующих функции функциональной структуры.
3. Синтез агрегатной структуры программной системы, на основе процессной структуры, и спецификация размещения программных агрегатов по процессорным компонентам РВК.
4. Верификация режима реального времени функционирования программной системы на этапах проектирования, разработки и опытной эксплуатации.
5. Модификация структуры программной системы по результатам верификации режима реального времени на этапах проектирования, разработки и опытной эксплуатации АСРВ с целью устранения выявленных нарушений ограничений реального времени.

Моделирование функциональной структуры программной системы

На рассматриваемом стратуме АСРВ структурный анализ и моделирование программной системы U=<M,A> на функциональном уровне заключается в выполнении следующих этапов:

1. Спецификация окружения программной системы U=<M,A> на рассматриваемом стратуме АСРВ в виде множества $S \cup V$ абстрактных объектов памяти, соответствующих контролируемым параметрам состояния объекта управления и программной системы, множества D абстрактных объектов памяти, соответствующих параметрам регулирования объекта управления, параметрам, отображаемым субъекту управления, и данным, помещаемым программной системой U=<M,A> в хранилище данных.
2. Спецификация типов памяти истоков $S \cup V \subset$M программной системы U=<M,A>.
3. Спецификация типов памяти стоков $D \subset$M программной системы U=<M,A>.
4. Функциональная структуризация программного агрегата A и представление его в виде структуры функций, удовлетворяющей требованиям функциональности программной системы U=<M,A> на рассматриваемом стратуме АСРВ.

Модель абстрактной памяти программной системы

Модель абстрактных объектов памяти потоковых данных

Абстрактная память M программной системы U=<M,A> - это конечное множество абстрактных *объектов памяти* (ОП), предназначенных для оперативного хранения абстрактных данных.

Для спецификации объекта памяти из M, указания его свойств, содержимого и разрешённого доступа используются следующие обозначения (отношения положения) [Флорес И. Структуры и управление данными. – М.: Финансы и статистика, 1982. – 319 с.]:

$$\mathrm{m},\ \mathrm{m}\big|_{sid}^{l},\ \not\Phi\left[\begin{smallmatrix}r\\w\end{smallmatrix}\right]\mathrm{m},\ \not\Phi\left[\begin{smallmatrix}r\\w\end{smallmatrix}\right]\mathrm{m}\big|_{sid}^{l};$$

где:

m –общее обозначение абстрактного ОП;

$\mathrm{m}\big|_{sid}^{l}$ - обозначение ОП с указанием идентификатора ОП – *sid*, и объёма памяти – *l*;

$\not\Phi\left[\begin{smallmatrix}r\\w\end{smallmatrix}\right]\mathrm{m}$ - обозначение абстрактного ОП с указанием его содержимого D – абстрактное данное, и разрешённого доступа - r, $w \in \{0,1\}$ – два флага разрешений доступа к ОП соответственно для чтения и записи;

$\not\Phi\left[\begin{smallmatrix}r\\w\end{smallmatrix}\right]\mathrm{m}\big|_{sid}^{l}$ - обозначение ОП в развёрнутом виде.

По отношению к ОП определены две операции доступа:

- `put` - поместить данное в ОП;
- `get` - получить данное из ОП.

Флаги доступа управляют выполнением операций доступа. Интерпретация значений флагов ОП следующая:

r=0 – "*запретить_получить*",

r=1 – "*разрешить_получить*";

w=0 - "*запретить_поместить*",

w=1 – "*разрешить_поместить*".

Типы объектов памяти

Тип объекта памяти определяет начальное содержимое ОП и состояние флагов доступа *r*, *w*, а также порядок изменения флагов доступа после выполнения над ОП операций `put`/`get`. Необходимость типизации ОП обуславливается двумя причинами. Во-первых, необходимостью согласования асинхронно выполняемых по отношению к ОП операций `put`/`get`. Во-вторых, необходимостью управлять датированным значением, содержащимся в ОП, когда истекает его интервал репрезентативности (данное теряет актуальность дальнейшего использования).

Типы ОП заданны следующим множеством:

Type={cell (*ячейка*), nest (*гнездо*)}.

При создании ОП абстрактной памяти M программной системы U=<M,A> им устанавливается соответствие с типами множества *Type*:

Δ_M: M→*Type*.

Тип *Ячейка* (cell)

Непосредственно после создания объекта памяти m типа *Ячейка* исходное содержимое ОП интерпретируется как пустое - $\varnothing\left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]\mathrm{m}\Big|_{sid}^{l}$, разрешена операция `put`.

При выполнении первой операции `put` в ОП помещается данное *D*, флаги доступа устанавливаются в единицу и далее не меняются - $D\left[\begin{smallmatrix}1\\4\end{smallmatrix}\right]\mathrm{m}$. После этого результатом выполнения каждой операции `get` будет копия текущего содержимого ОП. При этом содержимое ОП и флаги доступа не меняются - $D\left[\begin{smallmatrix}1\\4\end{smallmatrix}\right]\mathrm{m}$.

Тип *Гнездо* (nest)

Непосредственно после создания объекта памяти m типа *Гнездо* исходное содержимое ОП интерпретируется как пустое - $\varnothing\left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]\mathrm{m}\Big|_{sid}^{l}$, разрешена операция `put`.

При выполнении операции `put` в ОП помещается данное *D*, состояние ОП меняется на $D\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]\mathrm{m}$. ОП становится доступным для операции `get` и не доступен для операции `put`. При выполнении операции `get` содержащийся в ОП данное извлекается, состояние ОП меняется на $\varnothing\left[\begin{smallmatrix}0\\4\end{smallmatrix}\right]\mathrm{m}$, ОП доступен для операции `put` и не доступен для операции `get`.

Темпоральная модель данных

Темпоральная модель данных специфицирует содержимое абстрактных объектов памяти окружения программной системы U=<M,A> (*истоков* и *стоков*) во времени. Если значения некоторого параметра *P*(*t*) объекта управления фиксируются вместе с меткой времени, то такие значения называют *датированными значениями* параметра *P*(*t*).

Полагается, что для программной системы U=<M,A> определены системные часы реального времени с базовой шкалой времени. При этом в общем случае каждому источнику данных (параметру $P(t)$) для фиксации моментов времени устанавливаются свои локальные шкалы часов реального времени с единицей времени 1t, равной по величине значению интервала времени Δt по базовой шкале часов системного времени. Значение метки времени на локальной шкале обозначают $t\big|_{1t}$. Для получения значения метки времени $t\big|_{1t}$ по локальной шкале с единицей 1t, значение текущего момента времени t, полученное по базовой шкале системных часов реального времени, приводится к соответствующей локальной шкале времени по формуле $t\big|_{1t} = \left[\frac{t}{\Delta t}\right]$, т.е. равно целому от деления t на Δt. При этом значения меток времени всегда будут принадлежать множеству положительных целых чисел - $t\big|_{1t} \in \{0,1,2,\ldots,\omega,\ldots,\infty\}$, где $\omega = t\big|_{1t} - 1$ - последний зафиксированный момент времени ("отстаёт" от текущего момента времени); ∞ - положительное целое число, соответствующее моменту завершения функционирования программной системы U=<M,A>.

Датированное значение

Датированное значение описывается кортежем вида:

$$< d\big|_{\dot{t}}^{\dot{v}}, v\big|_{\dot{t}}^{\tau}(t) >\Big|_{1t}$$

где:

1t – значение единицы шкалы времени локальных часов;

$d\big|_{\dot{t}}^{\dot{v}}$ - абстрактное значение, порождённое в момент времени $\dot{t}$ и ассоциируемое с каждым последующим моментом времени $t \in \dot{t}, \dot{\tau}$ - *интервал репрезентативности*;

$\dot{t} \in \{0,1,2,\ldots\}$ - метка времени порождения значения $d\big|_{\dot{t}}^{\dot{v}}$ (по локальной шкале времени с единицей времени 1t) и начало интервала репрезентативности;

$\tau \in \{0,1,2,\ldots,\omega,\ldots,\infty\}$ - метка времени завершения интервала репрезентативности (по локальной шкале времени с единицей времени 1t);

$\dot{v} \in [0,1]$ – показатель, характеризующий степень соответствия значения $d\big|_{\dot{t}}^{\dot{v}}$ моменту времени $\dot{t}$ - *валидность порождения* значения $d\big|_{\dot{t}}^{\dot{v}}$;

$v\big|_{\dot{t}}^{\tau}(t) \in [0,1]$ – характеристическая функция, определяющая степень практической пригодности (*валидности*) использования датированного значения $d\big|_{\dot{t}}^{\dot{v}}$ в момент времени t как внутри интервала репрезентативности, так и за его пределами (в момент времени в прошлом или в текущий момент времени). Т.о. $v\big|_{\dot{t}}^{\tau}(t) \in [0,1]$ – *валидность использования* значения $d\big|_{\dot{t}}^{\dot{v}}$ в момент времени t.

В дальнейшем, если указание шкалы времени для обозначения датированного значения не принципиально, то для упрощения выражений будем опускать указание единицы шкалы времени и использовать запись - $< d|_{\dot{t}}^{\dot{v}}, v|_{\dot{t}}^{\tau}(t) >$.

В зависимости от значения, которое получает метка времени завершения интервала репрезентативности τ, датированные значения бывают *предопределённые* и *непредопределённые*.

Если метка времени завершения интервала репрезентативности есть положительное целое число - $\tau \in \{1,2,3,\ldots,\infty\}$, то датированное значение $< d|_{\dot{t}}^{\dot{v}}, v|_{\dot{t}}^{\tau}(t) >$ является предопределённым, конец интервала репрезентативности явно определён. Датированные значения $< d|_{\dot{t}}^{\dot{v}}, v|_{\dot{t}}^{\infty}(t) >$ соответствуют параметрам $p(t)$, значения которых во времени не изменяются (постоянные параметры).

Если $\tau=\omega$, то датированное значение $< d|_{\dot{t}}^{\dot{v}}, v|_{\dot{t}}^{\omega}(t) >$ является непредопределённым, конец интервала репрезентативности явно не определён в связи с априорной неопределённостью момента его завершения и ассоциируется с моментом времени, предшествующим текущему моменту времени. Такие датированные значения связаны с параметрами $p(t)$, для которых порождение очередного датированного значения инициируется событием, и, следовательно, момент завершения интервала репрезентативности априори не определим. В этом случае период репрезентативности датированного значения завершается в момент времени порождения следующего датированного значения. Логично полагать, что $\omega \le \infty$.

В зависимости от значения $v=\min\{\dot{v}, v|_{\dot{t}}^{\tau}(t)\}$ датированное значение $< d|_{\dot{t}}^{\dot{v}}, v|_{\dot{t}}^{\tau}(t) >$ характеризуется как:

- чёткое (*accurate*) – $v=\min\{\dot{v}, v|_{\dot{t}}^{\tau}(t)\}=1$;
- нечёткое (*fuzzy*) – $v=\min\{\dot{v}, v|_{\dot{t}}^{\tau}(t)\}\neq 1$;
- неопределённое (*nil*) - $v=\min\{\dot{v}, v|_{\dot{t}}^{\tau}(t)\}=0$.

Характеристика *accurate* говорит о том, что является абсолютно валидным в текущий момент времени.

Характеристика *fuzzy* говорит либо о не полном соответствии значения $d|_{\dot{t}}^{\dot{v}}$ моменту времени его порождения - $\dot{t}$, либо о потере датированным значением $< d|_{\dot{t}}^{\dot{v}}, v|_{\dot{t}}^{\tau}(t) >\Big|_{1t}$ репрезентативности ($t<\dot{t}$ или $t>\tau$). Если текущий момент времени t находится в пределах интервала репрезентативности, но при этом $v<1$, то это говорит о нечётком соответствии значения $d|_{\dot{t}}^{\dot{v}}$ моменту времени его порождения - $\dot{t}$. Если рассматриваемый момент времени $t \notin \dot{t},\tau$, то $v<1$ характеризует нечёткость значения $d|_{\dot{t}}^{\dot{v}}$ (степень пригодности для использования) за пределами его интервала репрезентативности. Чем ближе v к единице, тем более пригодно значение $d|_{l}^{\dot{t}}$ для использования в момент времени t.

Характеристика *nil* ($v(t)=0$) говорит о том, что значение $d\big|_t^{\dot{v}}$ полностью утратило связь с текущим временем и, следовательно, практически непригодно для его датирования моментом времени *t*.

Для определения значения валидности порождения значения $d\big|_t^{\dot{v}}$ используется формула:

$$\dot{v} = \begin{cases} 1, \textit{если } \ddot{t} = \dot{t}, \\ \dfrac{1}{\ddot{t} - \dot{t}}, \textit{если } \dot{t} < \ddot{t} < \tau. \end{cases}$$

Значение валидности $\dot{v}$ характеризует степень соответствия значения $d\big|_t^{\dot{v}}$, моменту времени его порождения – $\dot{t}$. Если датированное значение требуется получить в момент времени $\dot{t}$, а реально значение получено в момент времени $\ddot{t} > \dot{t}$, то чем больше разница между ожидаемым и реальным моментом времени порождения данного, тем менее чётко полученное значение можно ассоциировать с моментом $\dot{t}$. Заметим, что снижение валидности начинается тогда, когда расхождение между требуемым и реальным моментами времени порождения данного составляет не менее двух единиц (по шкале времени локальных часов с единицей времени 1t, в которой фиксируются моменты времени). *Валидность порождения* датированного значения есть величина постоянная, не зависящая от времени.

Для определения *валидности значения* $d\big|_t^{\dot{v}}$ в момент времени *t* применяется формула:

$$v\big|_t^{\tau}(t) = \begin{cases} \dfrac{\tau - \dot{t}}{\tau - t}, & t < \dot{t} \\ 1, & t \in [\dot{t}, \tau]; \\ \dfrac{\tau - \dot{t}}{t - \dot{t}}, & t > \tau, \end{cases}$$

Функция $v\big|_t^{\tau}(t)$ характеризует валидность (пригодность использования) датированного значения в момент времени *t*. До начала периода репрезентативности (ретроспективно) и после его окончания датированное значение может использоваться как нечёткое значение. Например, если датированное значение используется для компенсации потерянного предыдущего датированного значения. Если датированное значение является предопределенным (интервал репрезентативности априори задан и конечен), то при выходе текущего момента времени *t* за пределы интервала репрезентативности датированное значение теряет репрезентативность, значение становится нечётким и его можно только условно ассоциировать с текущим моментом времени с текущей степенью валидности. Валидность датированного значения падает с течением времени в соответствии с функцией валидности $v\big|_t^{\tau}(t)$, стремясь к нулю (неопределенному значению - *nil*). Чем больше период

репрезентативности, тем дольше нечёткое датированное значение сохраняет приемлемую валидность (например, $v|_t^{\tau}(t)$ >0.5).

Модель потоковых данных

Для моделирования потоков данных, порождаемых во времени некоторым источником данных (истоками), а также преобразователями данных или коммуникаторами, внутри агрегата A программной системы U=<M,A> используется модель *потоковых данных*, которые называют *то́кены*.

Токен обозначают в виде:

$$\mathrm{tn}=\mathrm{tn}\big|_{fid}^{\mathrm{type}} = D\ \bar{\mathrm{tn}}\big|_{fid}^{\mathrm{type}},$$

где:

tn – общее обозначение токена;

$\mathrm{tn}\big|_{fid}^{\mathrm{type}}$ - обозначение токена с указанием идентификатора потока и типа токена;

$D\ \bar{\mathrm{tn}}\big|_{fid}^{\mathrm{type}}$ - развёрнутое обозначение токена с указанием содержимого *D*;

D – содержимое токена;

fid – идентификатор потока, которому принадлежит токен;

type$\in$**{datum, mix, chain}** –тип токена.

Тип **datum** характеризует токен, содержащий датированное значение $<d|_t^{\dot{v}}, v|_t^{\tau}(t)>$. Такие токены называют *простыми*. Для обозначения простых токенов используется запись:

$$\mathrm{tn}\big|_{fid}^{\mathrm{datum}} = <d|_t^{\dot{v}}, v|_t^{\tau}(t)>\ \bar{\mathrm{tn}}\big|_{fid}^{\mathrm{datum}}.$$

Типы **mix** и **chain** специфицируют токены, содержимое которых составляют другие токены. Такие токены называют *составными*.

Тип **mix** характеризует составной токен *смешанного* потока данных. Такие токены в качестве данного могут содержать один токен произвольного типа (различных потоков), индексированный *канальным номером chnum* - $\langle \mathrm{tn} \rangle\big|_{chnum}$. Для обозначения токенов смешанного потока будем использовать запись:

$$\mathrm{tn}\big|_{fid}^{\mathrm{mix}} = \langle \mathrm{tn} \rangle\big|_{chnum}\ \bar{\mathrm{tn}}\big|_{fid}^{\mathrm{mix}}.$$

Тип **chain** характеризует составной токен, содержащий кортеж токенов - $\langle \mathrm{tn}_1, \mathrm{tn}_2, ..., \mathrm{tn}_n \rangle$, *n* – количество токенов в кортеже. Составной токен сцепленных данных коротко называют *составом*. Для обозначения состава используют запись:

$$\mathrm{tn}\big|_{fid}^{\mathrm{chain}} = \langle \mathrm{tn}_1, \mathrm{tn}_2, ..., \mathrm{tn}_n \rangle\ \bar{\mathrm{tn}}\big|_{fid}^{\mathrm{chain}}.$$

Язык схем асинхронных потоков данных

Определение АСПД-схемы

Графосимвольический язык схем асинхронных потоков данных (АСПД-схем) предназначен для формирования функциональной структуры агрегата A программной системы U=<M,A>. В самом общем виде АСПД-схема представляется графом, в качестве вершин которого выступают ОП и акторы, связанными ориентированными дугами. Дуги характеризуют доступ актора к ОП. Если актор имеет доступ к ОП, то он может выполнять по отношению к ОП операции *get* или *put*.

Объекты памяти АСПД-схемы

ОП – вершина, в которую может входить одна входная и одна выходная дуга, обозначающие соответственно доступ на помещение (*put*) и доступ на извлечение (*get*) данного. Входные дуги ОП исходят из вершин-акторов, а выходные – входят в вершины-акторы. Акторы, связанные дугами с ОП, имеют соответствующий доступ к ОП.

Истоки – объекты памяти $s \in S \cup V$, обладающие способностью порождать в себе датированные значения (неявное выполнение операции *put*).

Стоки - объекты памяти $d \in D$, обладающие способностью асинхронно поглощать находящиеся в них датированные значения (неявное выполнение операции *get*).

Потоковая память - объекты памяти $b \in B$, где B – внутренняя абстрактная память программной системы U=<M,A>, предназначенная для оперативного хранения токенов.

Множество конструктивных элементов построения АСПД-схем включает в себя: *истоки*, *стоки*, *генераторы*, *терминаторы*, *коммуникаторы*, *преобразователи* и *операторы*.

Истоки – объекты памяти $s \in S \cup P$, обладающие способностью порождать в себе датированные данные.

Стоки - объекты памяти $d \in D$, обладающие способностью асинхронно поглощать находящиеся в них датированные данные.

Потоковая память - объекты памяти $b \in B$, где B – внутренняя абстрактная память программного агрегата A, предназначенная для оперативного хранения токенов.

Генераторы предназначены для извлечения датированных значений, порождаемых истоками, и формирования трендов в заданной шкале времени.

Терминаторы предназначены для завершения трендов и занесения датированных значений в стоки.

Коммуникаторы - предназначены для управления коммуникацией токенов во внутренней абстрактной памяти B, программного агрегата A без функционального преобразования содержащихся в них датированных значений.

Преобразователи - предназначены для преобразования кортежа входных трендов в новый выходной тренд. Входные тренды терминируются, выходной тренд генерируется.

Модификаторы - предназначены для транзитной модификации токенов тренда.

Акторы

Агенты

Декомпозиция программного агрегата A программной системы U=<M,A> будет заключаться в том, чтобы представить агрегат A в виде структуры взаимодействующих абстрактных вычислителей (программных объектов), осуществляющие параллельную обработку данных, называемых программными агентами или коротко - *агентами*. Агенты имеют входы и выходы, предназначенные для связи с ОП памяти M. Объекты памяти, связанные с входами и выходами агента, соответственно рассматриваются относительно агента как его входные и выходные ОП. Все множество ОП, с которыми агент связан, называется *окружением* агента. Одни и те же ОП могут разделяться не более чем двумя агентами. Разделяемые двумя агентами ОП являются одновременно для одного агента входными, а для другого выходными. Если ОП связаны лишь с одним агентом, то они либо только входные, либо только выходные.

Поведение агента относительно своего окружения выражается в следующем:

- проверка заданных условий готовности окружения к выполнению агента;
- если некоторое из условий готовности окружения выполняется, то агент асинхронно извлекает данные из готовых входных ОП окружения, асинхронно выполняет соответствующую конечную процедуру обработки данных и асинхронно помещает результатные данные в готовые выходные ОП;
- если окружение не готово (ни одно из условий готовности окружения не выполняется), то агент бездействует.

Действие агента относительно своего окружения активизируется в результате осуществления доступа к любому из ОП его окружения другими агентами.

Для формального описания функциональной структуры и поведения программного агрегата A программной системы U=<M,A> поставим ему в соответствие три алгебры:

- *алгебру предикатов* - $<C,\sum_C>$,
- *алгебру процедур* - $<P,\sum_P>$,
- *алгебру агентов* - $<A,\sum_A>$.

Алгебра предикатов

Основой алгебры предикатов $<C,M,\sum_C>$ является множество *предикатов* C, которое содержит *предикаты* $c\Big|_{b_{out}^c}^{b_{in}^c} \in C$ - абстрактная логическая процедура над флагами доступа к объектам памяти $b \in (B_{in}^c \cup B_{out}^c) \subset M$, не приводящая к изменению

их состояния (содержимого и флагов доступа), результат выполнения которой оценивается как *истина – true*, или *ложь - false*.

Сигнатура Σ_C алгебры <C,Σ_C> включает основные операции алгебры логики: дизъюнкцию, конъюнкцию, инверсию.

Для удобства будем использовать обозначения процедур и предикатов без явного указания связанных с ними объектов памяти, если это очевидно или не важно, т.е. $c\big|_{b_{out}}^{b_{in}} \cong c$.

Алгебра процедур

Основой алгебры процедур <P,Σ_P> является множество *процедур* P, которое содержит процедуры $P\big|_{B_{out}^P}^{B_{in}^P}$ - абстрактная конечная вычислительная процедура над содержимым объектов памяти $b \in (B_{in}^P \cup B_{out}^P) \subset$M, в общем случае приводящая к изменению их состояния (содержимого и флагов доступа). По отношению к $b \in B_{in}^P$ процедура $P\big|_{B_{out}^P}^{B_{in}^P}$ может применять только операцию `get`, а по отношению к $b \in B_{out}^P$ - `put`, $B_{in}^P \cap B_{out}^P = \varnothing$. Множества B_{in}^P и B_{out}^P могут быть пустыми, но не одновременно, т.е. $P\big|_{B_{out}^P}^{\varnothing}, P\big|_{\varnothing}^{B_{in}^P} \in$M. Исключением будет только процедура $E\big|_{\varnothing}^{\varnothing} \in$P (*единичная* процедура - абстрактное бездействие) такая, что выполнение процедуры $E\big|_{\varnothing}^{\varnothing}$ не изменяет состояния объектов памяти $b \in$M.

Далее для простоты, где возможно, будем использовать обозначение процедуры без явного указания связанных с процедурой объектов памяти, т.е. $P\big|_{B_{out}^P}^{B_{in}^P} \cong P$.

Сигнатура Σ_P алгебры <P,Σ_P> включает следующие операции над процедурами:

Умножение $P \cdot Q$ – означает последовательное выполнение процедур в порядке их следования в операции умножения.

Для единичной процедуры $E\big|_{\varnothing}^{\varnothing}$ справедливы следующие соотношения:

$P \cdot E = E \cdot P = P$.

α-*Дизъюнкция* $(P \underset{\alpha}{\vee} Q)$ - означает исключительное выполнение процедур P или Q, где $\alpha \in$ C. Результат α-дизъюнкции есть конечная вычислительная процедура в P, которая выполняется следующим образом. Если α=*true*, то $(P \underset{true}{\vee} Q) = P$, в противном случае - $(P \underset{false}{\vee} Q) = Q$. Если необходимо выборочное безальтернативное выполнение процедуры P в зависимости от α, будем использовать α-дизъюнкцию в виде $(P \underset{\alpha}{\vee} E)$.

α-*Итерация* $\alpha:\{P\}$ - представляет собой новую процедуру, выполнение которой заключается в проверке условия α и циклическом выполнении процедуры P пока α=*true*. Если в результате проверки α=*false*, то P не выполняется, а процедура $\alpha:\{P\}$ заканчивается.

Алгебра агентов

Основой алгебры агентов <A,C,P,M,$\sum_A$> является *множество агентов* A, которое состоит из агентов $A\big|_{OUT_A}^{IN_A}$ - *абстрактный вычислитель* с конечным множеством входов $IN_A=\{1\big|_{in}^{A},2\big|_{in}^{A},...,n_{IN_A}\big|_{in}^{A}\}$ и конечным множеством выходов $OUT_A=\{1\big|_{out}^{A},2\big|_{out}^{A},...,n_{OUT_A}\big|_{out}^{A}\}$, $IN_A\cap OUT_A=\varnothing$; M – множество объектов памяти $b\in$M.

Сигнатура $\sum_A$ алгебры <A,M,$\sum_A$> включает следующие операции над агентами:

Подключение. Операция подключения заключается в интерпретации входов и выходов агента объектами памяти $b\in$M. Т.е.:

$$A\big|_{OUT_A}^{IN_A}\rightarrow A\big|_{B_{out}^A}^{B_{in}^A}$$

Объекты памяти, соединённые с входами, будут для агента *входными* ОП, а соединённые с выходами – *выходными* ОП. Входные и выходные ОП агента составляют его *окружение*. Агент, у которого все входы и выходы соединены с объектами памяти, назовём *подключенным*. Подключённые агенты будем обозначать $A\big|_{B_{out}^A}^{B_{in}^A}$, где $B_{in}^A,B_{out}^A\subset$M; $B_{in}^A\cap B_{out}^A=\varnothing$. Подключённые агенты начинают функционировать. Функционирование агента выражается в асинхронном извлечении данных из входных ОП и асинхронном занесении результатов своей работы в выходные ОП, приводя к изменению состояния ОП своего окружения (содержимого и флагов доступа). Таким образом, множество агентов A можно разделить на два непересекающихся подмножества: A^- - множество неподключённых агентов; A^+ - множество подключённых агентов. То есть, $A=A^-\cap A^+$, $A^-\cup A^+=\phi$.

Интерпретация. Операция интерпретации заключается в интерпретации

Агрегация $R=P\oplus Q=\left[\begin{matrix}P\big|_{B_{out}^P}^{B_{in}^P}\\ Q\big|_{B_{out}^Q}^{B_{in}^Q}\end{matrix}\right]\Big\|_{B_{out}^P\cup B_{out}^Q\setminus B_{in}^P\cup B_{in}^Q}^{B_{in}^P\cup B_{in}^Q\setminus B_{out}^P\cup B_{out}^Q}$. Результат агрегации представляется в как новый агент *R*, выполнение которого заключается в параллельном выполнении агентов *P* и *Q*. Агент *R* – результат агрегация, переходит к бездействию, когда асинхронно перешли к бездействию обе процедуры *P* и *Q*.

Выполнение актора *A* можно теперь формально выразить в виде: $A=1:\{(P_A\underset{c_A}{\vee}E)\}$. Актор бесконечно исполняется и каждый раз при c_A=1 выполняет процедуру P_A, в противном случае – *E*.

Множество конструктивных элементов построения АСПД-схем включает в себя: *истоки*, *стоки*, *генераторы*, *терминаторы*, *коммуникаторы*, *преобразователи* и *операторы*.

Истоки – объекты памяти $s\in S\cup P$, обладающие способностью порождать в себе датированные данные.

Стоки - объекты памяти $d \in D$, обладающие способностью асинхронно поглощать находящиеся в них датированные данные.

Проходная память - объекты памяти $b \in B = M \setminus (S \cup P \cup D)$, где B – внутренняя абстрактная память программного агрегата A, предназначенная для оперативного хранения токенов.

Генераторы предназначены для извлечения датированных значений, порождаемых истоками, и формирования трендов в заданной шкале времени.

Терминаторы предназначены для завершения трендов и занесения датированных значений в стоки.

Коммуникаторы - предназначены для управления коммуникацией токенов во внутренней абстрактной памяти - $B = M \setminus (S \cup P \cup D)$, программного агрегата A без функционального преобразования содержащихся в них датированных значений.

Преобразователи - предназначены для преобразования кортежа входных трендов в новый выходной тренд. Входные тренды терминируются, выходной тренд генерируется.

Модификаторы - предназначены для транзитной модификации токенов тренда.

Исток

Исток (**s**ource) $s \in S \subset M$ является объектом памяти заданного типа программной системы $\square = <\subset, \aleph>$, предназначенным для асинхронного порождения датированных значений. Истоки бывают *активные* (A-истоки) и *пассивные* (I-истоки).

A-*исток* $s \in S$, как только он переходит в состояние $s|_1^r$, асинхронно порождает датированное значение в результате выполнения операции `insert` по отношению к $s|_1^r$ со стороны окружения программной системы $\square = <\subset, \aleph>$, после чего исток переходит в состояние - $\left[< d|_l, \dot{t}, z > |_0^{v_S(t)} \right] s|_w^1$. Все атрибуты датированного значения формируются окружением программной системы. Используется базовая шкала времени.

I-*исток* $s \in (S \cup P)$, находясь в состоянии $s|_1^r$, самостоятельно не порождает датированных значений. Порождение датированного значения инициируется в истоке только в результате выполнения программным агрегатом A операции `extract`. Если при вызове на выполнение операции `extract` состояние истока - $s|_1^0$, то операция `extract` блокируется до момента перехода истока в состояние $s|_w^1$, а в истоке инициируется процесс порождения датированного данного. В результате исток порождает значение, датируемое моментом времени $\dot{t}$ - инициирования операции `extract`, и переходит в состояние - $\left[< d|_l, \dot{t}, z > |_0^{v_S(t)} \right] s|_w^1$, операция `extract` деблокируется и выполняется. Исток переходит в состояние $s|_1^r$.

Если при вызове на выполнение операции `extract` состояние истока - $s|_w^1$, то операция `extract` сразу выполняется, извлекая текущее содержимое истока,

исток переходит в состояние $s|_1^r$, а в истоке инициируется процесс порождения нового значения, датируемого моментом времени $\dot{t}$ - инициирования операции *extract*, в результате чего исток априори перейдёт в состояние - $\left[< d|_l, \dot{t}, z > |_0^{v_s(t)} \right] s|_w^1$.

Такой режим работы характерен для пассивных истоков $s \in P$, порождающих датированные значения, используемые в программном агрегате как параметры.

Сток

Сток (**d**rain) $d \in D \subset M$ является объектом памяти программной системы $\Box = <\subset, \aleph>$, который предназначен для асинхронного поглощения помещаемого в него датированного значения. Поглощение датированного значения осуществляется в результате выполнения операции *extract* по отношению к $d|_w^1$ со стороны окружения программной системы $\Box = <\subset, \aleph>$. В результате поглощения датированного значения сток переходит в состояние $d|_1^r$.

Проходная память

Проходная память (feedthrough **b**ucket) $b \in B = M \setminus (S \cup P \cup D)$ является объектом внутренней памяти агрегата A программной системы $\Box = <\subset, \aleph>$. Проходная память предназначена для оперативного хранения токенов, обрабатываемых агрегатом A. По отношению к проходной памяти в состоянии $b|_1^r$ применима операция *insert*. По отношению к проходной памяти в состоянии $b|_w^1$ применима операция *extract*.

Генераторы

Генератор предназначен для формирования трендов в заданной временн'ой шкале. Каждый генератор в АСПД-схеме имеет доступ к единственному истоку, из которого он извлекает датированные данные. Генераторы делятся на ведущие (M-*генераторы*) и ведомые (S-*генераторы*). Для доступа к I-истоку необходим M-генератор, который задаёт темп формирования тренда (интервал репрезентативности). Для доступа к A-истоку необходим S-генератор. В этом случае темп формирования тренда задается определяется интенсивностью порождения датированных данных A-истоком.

Для получения датированного данного из истока генератор инициирует по отношению к истоку операцию *get*. После выполнения операции *get* формально работа любого генератора *gn* с единицей шкалы часов PB – 1t, описывается формулой:

$$\left(\left[< d|_l, \dot{t}, \bar{z} > |_0^{v_s(t)} \right] s|_w^1 \right) \xrightarrow{generator|_{gn}^{1t}} \left[< \dot{d}|_l, \dot{t}, z > |_{1t}^{v(t)} \; tn|_{gn}^{datum} \right] b|_1^r ,$$

где для A-истока $\bar{z} = \tau$ - определенный конечный момент времени для предопределённых датированных данных, либо $\bar{z} = \omega$ - неопределённый момент времени для непредопределённых датированных данных; для I-истока - $\bar{z} = \omega$.

M-генератор извлекает сформированное датированное значение, ассоциируемое с моментом времени $\dot{t}$, из I-истока s∈S, устанавливает значение z интервала репрезентативности датированного данного $< d|_l, \dot{t}, z >|_{1t}^{v(t)}$ (временн'ые метки задаются в единицах времени – 1t), формирует потоковое данное генератора с именем *gn* и временной шкалой 1t - простой токен $<\bar{d}|_l, \dot{t}, z >|_{1t}^{v(t)}\ \bar{tn}|_{gn}^{datum}$, и помещает результат в проходную память b∈B - $tn|_{gn}^{datum}\ \bar{\bar{b}}|_1^r$.

S-генератор извлекает сформированное датированное значение, ассоциируемое с моментом времени $\dot{t}$, из A-истока s∈S, преобразует моменты времени $\dot{t}$ и $\bar{z}$ из базового РВ в шкалу – 1t, формирует простой токен $<\bar{d}|_l, \dot{t}, z >|_{1t}^{v(t)}\ |_{g\bar{n}}^{\bar{datum}}$ генератора с именем *gn* и метками времени в шкале 1t, помещает результат в проходную память b∈B. После этого S-генератор вновь инициирует по отношению к A-истоку операцию `extract`.

Формальная спецификация конструктивных элементов
АСПД-схемы.

Вводится следующий набор базовых информационных процессов: генератор, терминатор, поглотитель, синхронизатор, коммутатор, распределитель, коллектор, селектор, составитель, сортировщик, буфер, задержка, размножитель, преобразователь, склейка.

Их графическое представление показано на Рис. 1.2. На рисунках окружностями малого диаметра показаны сечения дуг. Левая часть дуг не имеет стрелки, а правая - со стрелкой. Окружностями большого диаметра показаны ОП типа сток, исток. Из истока дуга со стрелкой выходит, а в сток - входит.

Генератор (generator)

Генератор предназначен для порождения потоков данных, инициируемых связанным с ним истоком. Каждому генератору присваивается имя- *name*, тип порождаемого потока - *type*, *pr* - приоритет. Генератор использует ОП *clock* для получения значения системного времени. Спецификация генератора имеет вид

$$b = <R(\alpha) \wedge R(s) \wedge R(\beta)> \mid \downarrow(\alpha); \uparrow t_n; name \dot{\rightarrow} t_n.NAM;$$

$$type \rightarrow t_n.TYP; pr \dot{\rightarrow} t_n.PR; (clock) \rightarrow t_n.TIM;$$

$$(s) \rightarrow t_n.MSG; t_n \rightarrow \beta>$$

t_n : token, $i \in \{1,2,\ldots,n\}$,

где i - количество системных часов.

В зависимости от интерпретации типов памяти можно получить генераторы с различными режимами работы. Например, при s: cell, α : socket, β: subsocket получим генератор с активным режимом работы. Он сам будет определять темп генерации потока, но если этот темп превысит интенсивность потребления данных из порта β (темп преобразования), то появятся ошибки. Если положить s, α, β : socket, то получим асинхронный генератор, который полностью сбалансирован по темпу работы.

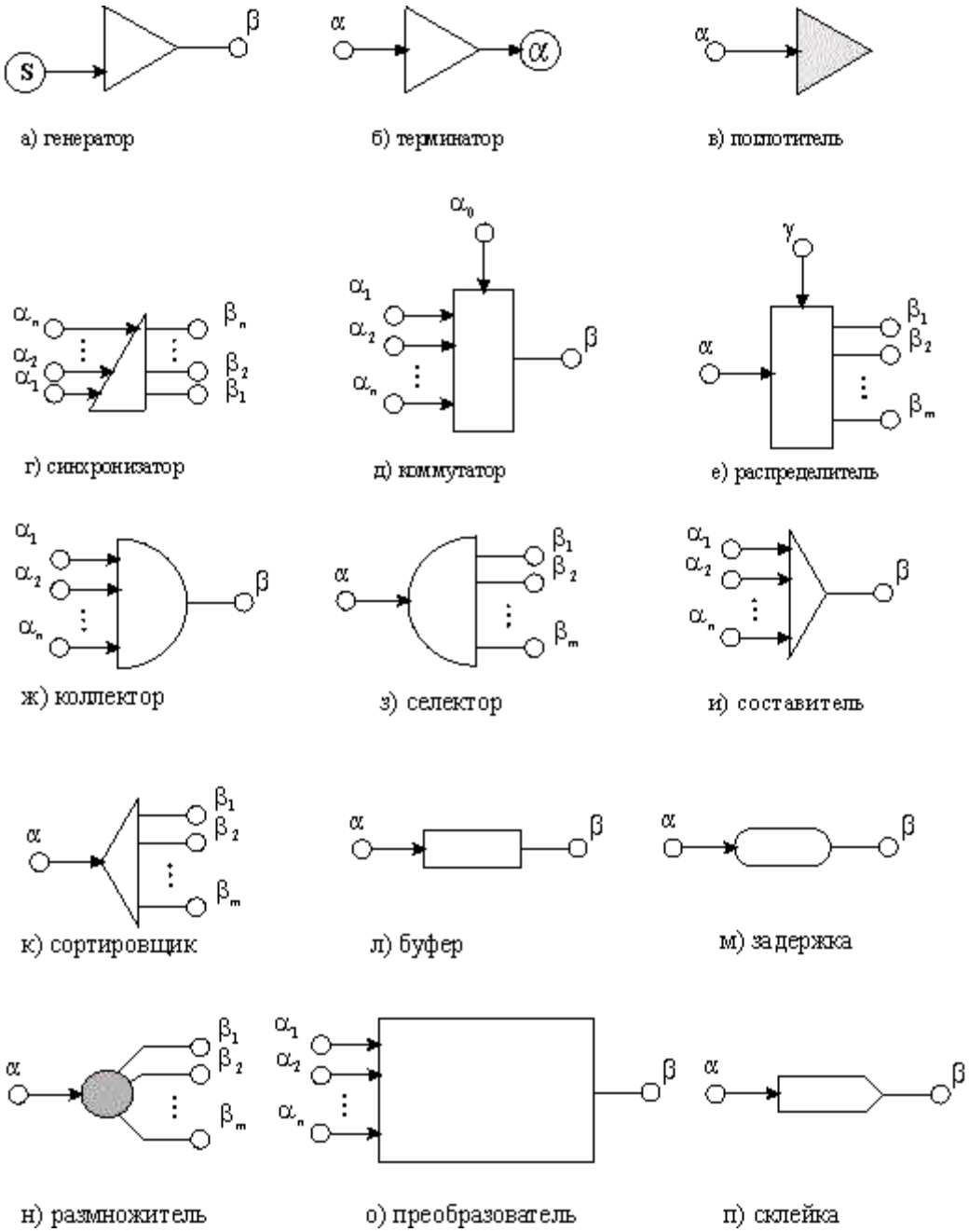


Рисунок 1.2 -Типы базовых информационных процессов АСПД-схемы.

Терминатор (terminator)

Для потоков и вывода результатов преобразования данных в стоки предназначен базовый информационный процесс терминатор. Спецификация терминатора имеет вид

$$b=<R(\alpha)\wedge W(d)/(\alpha) \rightarrow t_n;\ (t_n.MSG)\rightarrow d;\ \downarrow t_n>.$$

Как и для генератора с помощью различной интерпретации типов ОП можно получать разные режимы работы терминатора.

Поглотитель

Для уничтожения потоков данных, необходимость в выводе результатов преобразования которых отсутствует, предназначен базовый процесс - поглотитель. Его спецификация имеет вид

$$b=<R(\alpha)/\downarrow(\alpha)>$$

α : *socket, subsocket, subcell* .

Синхронизатор

Если необходимо осуществить функциональное преобразование совокупности асинхронных потоков данных или просто требуется привести их значения к единой временной метке используется информационный процесс - синхронизатор. На входы синхронизатора поступают асинхронные потоки, а на выходы - потоки, синхронизированные по временным срезам. Спецификация синхронизатора имеет вид

$b_i = <R(\alpha_i) \wedge W(\gamma_i) / (\alpha.i) \rightarrow (\gamma.i)>, \quad i = \overline{1,n}$

$b_{i+1} = < \underset{i=1}{\overset{n}{\wedge}} R(\gamma_i) \wedge \underset{i=1}{\overset{n}{\wedge}} W(\beta_j) \mid k := 1;$

$(k)>n : [((\gamma.(k)).TIM)>(time):\{((\gamma.(k)).TIM) \rightarrow time;\ k:=(k)+1; \vee k:=(k)+1;\}];$

$k:=1;\ (k)>n:[((\gamma.(k)).TIM)=(time):\{(\gamma.(k)) \rightarrow \beta.k;$

$\vee(\gamma.(k)) \rightarrow \varphi;\ (\varphi) \rightarrow \gamma.(k);\ (time) \rightarrow (\varphi).TIM;\ (\varphi) \rightarrow \beta.(k)\}] >$

$time, \varphi$*:cell,*

γ_I *: socket,* $\forall I = 1,n$

(γ_I) *: token,*

(φ) *: token.*

Процесс синхронизации заключается в следующем. Синхронизатор имеет локальные ОП - γ_I, φ, *time*. Элементы памяти γ_I предназначены для временного хранения токенов, φ - служебная память для хранения токена, а *time* предназначена для хранения метки времени. Исполнение синхронизатора начинается, если существует $W(\gamma_I)=1$ и в порт α_i поступил токен. Этот токен заносится в γ_I. Если все γ_I содержат токены, а все β_i готовы к приему токенов, то происходит срабатывание синхронизатора. Оно заключается в том, что из всех временных меток токенов, содержащихся в γ_I , выбирается наименьшая и ее значение заносится в *time*. Токены, метки которых совпадают с *time* , перемещаются из γ_I в β_i. Остальные токены перемещаются поочередно из γ_I в φ, из φ в β_i , и из φ снова в γ_I.То есть происходит размножение токена. Предварительно токену, находящемуся в φ метку времени заменяют на *time* содержимое *time*. Таким образом, выходящие из синхронизатора потоки становятся синхронизированными по времени.

Коммутатор

Информационный процесс коммутатор предназначен для управления выбором токенов из различных потоков, поступающих на его входы α_i. Для этих целей коммутатор имеет управляющий вход φ. Коммутатор имеет следующую спецификацию

$b_0 = <R(\alpha_0) \wedge W(\gamma_0) / (\alpha.0) \rightarrow \gamma.0;\ ((\gamma.0).MSG) \rightarrow l>$

$b_i = < R(\alpha_i) \wedge W(\gamma_i) / (\alpha.i) \rightarrow \gamma.i>, \quad I=1,n$

$b_{n+1} = <R(\gamma_l) \wedge W(\beta) / (\gamma.l) \rightarrow \beta>$

γ_0 *: cell,*

γ_I *: socket, subsocket, subcell;* $i=1,n$

(γ_i): token; i=0,n

((γ_0).MSG)$\in${1,2,…,n}.

Используя различные интерпретации типов ОП α_I , можно строить процессы коммутации различных видов.

Распределитель

Информационный процесс распределитель позволяет управлять маршрутом движения потока данных. Для этого он располагает управляющим входом γ и несколькими выходами β_i. Его спецификация следующая

$b_1 = <R(\gamma) \wedge W(\varphi) / (\gamma) \rightarrow \varphi; ((\varphi).MSG) \rightarrow l>$

$b_1 = <R(\alpha) \wedge R(\varphi) \wedge W(\beta_l) / (\alpha) \rightarrow \beta_l ; \downarrow(\varphi)>$

(φ) : token,

φ : socket, subsocket, subcell,

((φ).TYP) : simple,

((φ).MSG)$\in${1,2,…,m}.

Коллектор (collector)

Этот процесс предназначен для управления маршрутами движения потоков данных. Он осуществляет смешивание токенов потоков, входящих в коллектор, и ведает на выходе смешанный поток. Спецификация коллектора имеет вид

$b = <\overset{n}{\underset{i=1}{\vee}} R(\alpha_i) \wedge W(\beta) / k:=1;$

$\overline{W(\beta)} \wedge (k) > n : [R(\alpha.(k)) : \{(\alpha.(k)) \rightarrow \beta; k := (k)+1; \vee k := (k)+1\}] >$

Селектор

Процесс-селектор выполняет преобразования обратные действиям коллектора. Он разделяет смешанный поток, поступающий на его вход, и в соответствии с именем потока токены отправляются на тот или иной выход. Спецификация селектора выглядит следующим образом:

$b_1 = <R(\alpha) \wedge W(\varphi) / (\alpha) \rightarrow \varphi; f_{name}: ((\varphi).NAM) \rightarrow l; (l) \rightarrow k>$

$b_2 = <W(\beta_k) / (\varphi) \rightarrow \beta.k>$

φ : socket,

(φ): token.

Здесь φ локальный ОП, в качестве содержимого которого выступают токены.

Составитель (coupler)

Процесс составитель предназначен для получения составных токенов (составов). Из каждого входного порта выбирается по одному токену, которые становятся затем содержимым порождаемого составного токена, выдаваемого в выходной порт. Его спецификация имеет вид

$b = <\overset{n}{\underset{i=1}{\wedge}} R(\alpha_i) \wedge W(\beta) / \uparrow t_n; k:=1;$

(k)>n:[((α.(k)).PR)>(pr):{((α.(k)).PR→pr};

(α.(k))→train.(k); k:=(k)+1]

f_{name} *:(train).NAM)→t_n.NAM; complex→t_n.TYP;*

$(pr) \rightarrow t_n.PR;\ \omega \rightarrow t_n.TIM;\ (train) \rightarrow t_n.MSG; t_n \rightarrow \beta>$

$(train.i)$: token, $i=1,n$.

Из спецификации следует, что в качестве приоритета состава выбирается максимальный из приоритетов, составляющих его токенов. Метка времени для состава не определена. Имя состава является отображением вектора имен, входящих в состав токенов.

Сортировщик

Процесс - сортировщик является обратным процессу – составитель.

Приходящий на вход составной токен разбирается на составляющие его токены, которые в заданном порядке выдаются на выход. Спецификация сортировщика следующая:

$b_0 = <R(\alpha) \wedge W(train) / ((\alpha).MSG) \rightarrow train>$

$b_i = <R(train_i) \wedge W(\beta_i) / (train.i) \rightarrow \beta.i>,\ I=1,m$

$(train.i) : token.$

Буфер (канал с памятью)

Процесс-буфер предназначен для согласования темпов поступления и преобразования потока данных. Буфер имеет ограничений объем, который обозначается целым положительным числом, обозначающим количество токенов, которые могут одновременно находится в буфере. На схеме это число записывается в прямоугольник, изображающий буфер. Спецификация буфера имеет вид

$b_1 = <R(\alpha) \wedge ((v)=0 \vee ((v)<l \wedge \overline{W(\beta)})) / (\alpha) \rightarrow q; v:=(v)+1>$

$b_2 = <W(\beta) \wedge (v)>0 \ / (q) \rightarrow \beta;\ v:=(v)-1>$

$q : quit;$

$(v) \in \{0,1,2,\ldots,l\};$

где l - объем буфера.

Задержка

Информационный процесс-задержка задерживает дальнейшее продвижение токена до момента времени, значение которого является его сообщением. Задержка имеет следующую спецификацию

$b_1 = <R(\alpha) / (\alpha) \rightarrow \varphi>$

$b_2 = <(\varphi) \neq 0 \wedge ((\varphi).MSG) = (clock) \wedge ((\varphi).MSG \neq \omega / (\varphi) \rightarrow \beta>$

(φ): token,

φ : subcell,$((\varphi).TYP)$:simple.

Поступивший в задержку токен может заменить токен, ранее туда вошедший, который уничтожается. Если вошедший токен имеет неопределенное сообщение ω , то он будет задерживаться бесконечно долго. Это позволяет осуществлять планирование интеррогативных механизмов синхронизации информационных процессов.

Размножитель

Процесс - размножитель выполняет порождение копий потоков данных, поступающих на его вход. Его выполнение специфицируется следующим образом:

$$b=<R(\alpha)\wedge\overset{n}{\underset{i=1}{\wedge}}W(\beta_i)\mid k:=1;(\alpha)\to\varphi;(k)>m-1:[\uparrow t_n;^2(\varphi)\to t_n;t_n\to\beta.(k)];(\varphi)\to\beta.m>$$

φ*:cell,*
(φ*): token,*
t_n *: token.*

Преобразователь

Если рассмотренные выше информационные процессы были предназначены для управления потоками данных без изменения содержимого токенов, то преобразователь определяет наиболее сложные действия с потоками. Он преобразует входящие в него потоки данных в новый поток с новым именем, приоритетом и сообщением, порождая его наподобие генератора, а исходные потоки уничтожает как поглотитель. Спецификация преобразователя имеет следующий вид :

$$b=<W(\beta)\wedge\overset{n}{\underset{i=1}{\wedge}}R(\alpha_i)\,/\,k:=1;\ 0\to pr;(k)>n:[(\alpha.(k))\to\varphi.(k);k:=(k)+1];$$

$$\uparrow t_n;\overset{n}{\underset{i=1}{\wedge}}\overset{n}{\underset{j=1}{\wedge}}((\varphi.i).TIM)=((\varphi.j.TIM):\{\overset{n}{\underset{i=1}{\wedge}}((\varphi.i).TYP)=simple:\{$$

$$simple\to t_n.TYP;((\varphi.1).TIM)\to t_n.TIM;$$

$$f_{msg}:(\{(\varphi.i).MSG\}_{i=1,n})\to t_n.MSG\vee\omega\to t_n.TYP;((\varphi.1).TIM\to t_n.TIM;$$

$$\omega\to t_n.MSG\}\vee\omega\to t_n.TIM;\omega\to t_n.TYP;\ \omega\to t_n.MSG\};$$

$$k:=1;\ (k)>n:[((\varphi.(k)).PR)>pr:\{((\varphi.(k)).PR\to pr\}];$$

$$f_{name}:(\{(\varphi.i).NAM\}_{i=1,n})\to t_n.NAM;(pr)\to t_n.PR>$$

φ_i *: cell,*
(φ_i*): token, i=1,n.*

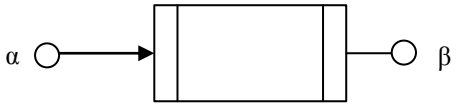
Для конкретной реализации преобразователя требуется интерпретировать отображение f_{name}, которое преобразует вектор имен входных потоков в имя выходного потока, ее отображение которое преобразует вектор сообщений входных потоков в сообщение выходного потока.

Преобразователь устойчив к ошибкам синхронизации потоков данных. В этом случае он вырабатывает токен с неопределенным типом И/ИЛИ неопределенной меткой времени, а также с неопределенным сообщением. Кроме того, если среди исходных сообщений будет неопределенное сообщение ω, то и результат будет ω (вытекает из свойств ω).

Модификатор

Преобразователь преобразует входящие в него потоки данных (в общем случае, более одного) в новый поток с новым именем, приоритетом и сообщением, порождая его наподобие генератора, а исходные потоки уничтожает как поглотитель. В отличие от преобразователя модификатор не терминирует входной поток, а осуществляет преобразование датированных данных в токенах.

Поэтому на вход модификатора могут поступать только простые потоки с токенами типа **datum**.



Склейка

Процесс-склейка предназначен для выполнения операций разреза над АСПД - схемой. Формально склейка определяется следующим образом:

$b_1 = <R(\alpha) \wedge W(d) / (\alpha) \rightarrow t_n; (t_n) \rightarrow d; \downarrow t_n>$

$b_1 = <R(d) \wedge W(s) / (d) \rightarrow s>$

$b_1 = <R(s) \wedge W(\beta) / \uparrow t_n; (s) \rightarrow t_n;\ t_n \rightarrow \beta>$

d,s : socket.

Итак, проведенным выше перечислением элементов, определено множество базовых информационных процессов - Ω_B АСПД-схемы.

Описание примера АСПД-схемы

Проиллюстрируем применение АСПД-схем на примере описания фрагмента системы автоматизации испытаний газотурбинных двигателей/1/. Описываемый фрагмент системы предназначен для сбора и преобразования данных, связанных с исследованием зависимости приведенного расхода топлива от частоты вращения ротора газотурбинного двигателя (дроссельная характеристика двигателя) в форсажном и бесфорсажном режимах работы двигателя. То есть, в результате исследований необходимо получить две зависимости:

$M_{пр} = F^{форс}_{Мпр}(N, M_{изм}(q), P_{вх}, T_{вх})$

$M_{пр} = F^{бесфорс}_{Мпр}(N, M_{изм}(\Delta t), P_{вх}, T_{вх})$:

N - частота вращения ротора;

$P_{вх}$ - давление воздуха на входе в двигатель;

$T_{вх}$ - температура воздуха на входе в двигатель;

$M_{изм}(q)$ – измеренный расход топлива при форсаже;

$M_{изм}(\Delta t)$ – измеренный расход топлива без форсажа;

$M_{пр}$ – вычисленный приведенный расход топлива.

В процессе испытаний фиксируются/измеряются:

r – режим работы двигателя;

N - частота вращения ротора;

$P_{вх}$ - давление воздуха на входе в двигатель;

$T_{вх}$ - температура воздуха на входе в двигатель;

Δt – интервала времени, за который расходуется мерный объем топлива в бесфорсажном режиме;

q – показания расходомера топлива в режиме форсажа.

В качестве результатов регистрируются:

r – режим работы двигателя;

N - частота вращения ротора;

$M_{пр}$ – вычисленный приведенный расход топлива.

На рисунке 1.3 представлена АСПД-схема, описывающая процессы получения, преобразования и регистрации потоков экспериментальных данных.

Схема включает в себя шесть генераторов ($G_1 \div G_6$), порождающих потоки данных, инициируемых связанными с генераторами истоками ($S_1 \div S_6$,). Так, генератор G_1 порождает токен, несущий двоичную информацию о режиме испытаний - r - (r =0 - бесфорсажный или r =1 - с форсажем). Этот токен используется как управляющий в коммутаторе B_1 и изменяет способ вычисления $M_{изм}$. На бесфорсажном режиме эта величина определяется путем измерения интервала времени Δt , за который расходуется мерный объем топлива (токен, содержащий значение Δt, порождается генератором G_2).

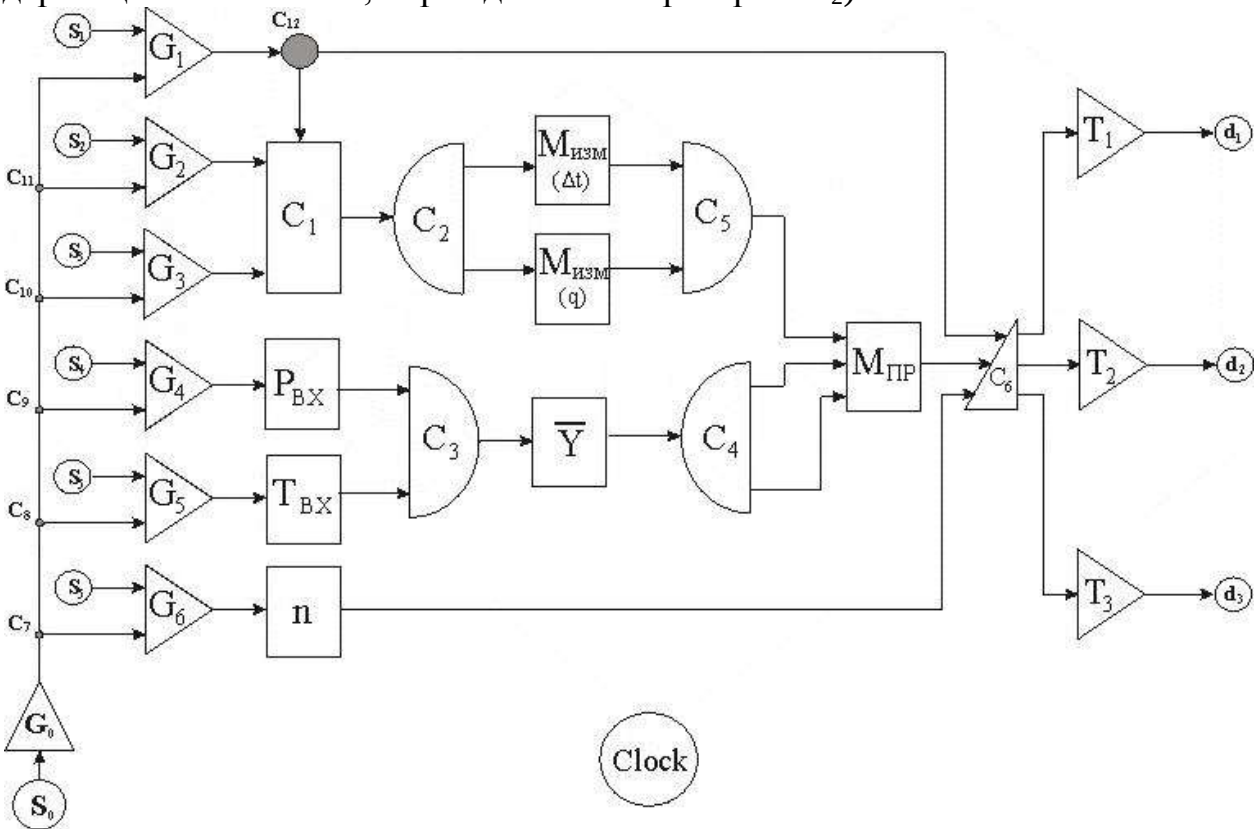


Рисунок 1.3 – Пример АСПД-схемы системы автоматизации.

На форсированных режимах расход q определяется при помощи специальных расходомеров. Поток данных по расходу q генерируется G_3. В соответствии со значением управляющего токена, коммутатор задействует либо поток токенов с Δt, либо - q, которые поступают соответственно на преобразователи $M_{изм}(\Delta t)$ или $M_{изм}(q)$, представляющие собой различные формы отображений измеренных на объекте физических величин в параметр двигателя - $M_{изм}$. Разделение смешанного потока на выходе коммутатора C_1 по преобразователям выполняется селектором C_2, который отображает имя входного потока в номер выходного порта. Коллектор C_5 осуществляет обратную операцию, объединяя полученные различным путем потоки с $M_{изм}$ в единый поток, поступающий на один из входов преобразователя $M_{пр}$.

Генераторы G_4 и G_s порождают токены, содержащие вектора двоичных измерительных кодов, характеризующих соответственно давление к температуре

на входе в двигатель. Вектор измерительных кодов является результатом многократного опроса одной и той же физической величины по сечению и по времени с целью повышения ее достоверности. В преобразователях $P_{вх}$ и $T_{вх}$ производится отбраковка результатов и отображение измерительных кодов в цифровые эквиваленты величин давления и температуры в заданных точках сечения с помощью полиномиальных преобразований и известных градировочных характеристик измерительных каналов. Аналогичным образом, только с использованием специальных устройств измерения оборотов, получается значение частоты вращения ротора (генератор G_1, преобразователь n). Коллектор C_3 смешивает потоки температур и давлений и направляет их на вход преобразователя Y, который преобразует совокупность физических значений, в заданных точках сечения на входе двигателя, в средние значения $P_{вх}$ и $T_{вх}$. При этом теги токенов проходят через преобразователь Y без изменения, а сообщения преобразуются. На выходе селектора C_4 смешанный поток средних значений разделяется, и каждый из потоков поступает на свой вход преобразователя $M_{пр}$.

Вычисление приведенного значения расхода $M_{пр}$ осуществляется только в том случае, когда на вход соответствующего преобразователя поступят все необходимые данные (токены, содержащие $M_{изм}$, $P_{вх}$, $T_{вх}$).

На выходе схемы осуществляется терминация потоков данных r, $M_{пр}$ и n в стоках d_1, d_2, d_3 . Допустим требуется, чтобы на устройствах отображения данные появлялись одновременно, а время получения этих данных разное из-за разницы маршрутов преобразования. Для удовлетворения этому требованию синхронизируем эти потоки относительно друг друга, пропустив их через синхронизатор C_6. Тогда данные на терминаторы $T_1 \div T_3$, поступят одновременно.

Рассмотрим подробнее способ управления генерацией, заданный на схеме. На вход каждого генератора поступает поток данных, выходящий из генератора G_0 и размножаемый в $C_7 \div C_{11}$. Этот поток представляет собой поток событий, порождаемых нажатием кнопки "замер" исток S_0, выполняемым инженером-испытателем при проведении испытаний. Этот же поток событий является времяобразующим потоком системного времени clock, являющегося особым истоком схемы, доступным всем объектам схемы и имеющим тип "ячейка" (связи на схеме не показываются, а подразумеваются). В данной схеме имеется только одно системное время, выражающееся в порядковых номерах замеров (исток с clock).

При нажатии кнопки "замер" на вход генераторов поступают инициирующие токены. При возникновении данных в истоках, генераторы срабатывают (возможно не одновременно). Следующее срабатывание произойдет после очередного нажатия кнопки "замер" и появления данных в истоках. С каждым нажатием кнопки происходит изменение системного времени. Если темп нажатия кнопки превысит темп преобразования данных, то будут потеряны некоторые замеры, а факт потери определится по их отсутствию в регистрируемых данных (нарушится последовательная нумерация замеров).

Моделирование ПСРВ на процессном уровне

Виртуальная потоковая машина

Моделирование ПСРВ на функциональном уровне завершается построением АСПД-схемы. Цель моделирования ПСРВ на процессном уровне – разработка структуры взаимодействующих процессов, реализующих семантику полученной АСПД-схемы, используя средства выбранной для реализации ПСРВ операционной системы реального времени. Формально выделение процессов на основе АСПД-схемы выражается в разделении АСПД-схемы на непересекающиеся фрагменты, каждый из которых инкапсулирует соответствующую часть АСПД-схемы и заменяет её программным агрегатом, интегрирующим в себе совокупность конструктивных элементов АСПД-схемы, вошедших в выделенный фрагмент. Для выделения фрагментов используется конструктивный элемент АСПД-схемы – склейка. Склейками обозначаются связи на границе фрагмента. Когда границы фрагментов отмечены склейками, осуществляется операция разреза склеек. В результате операции разреза склейка заменяется двумя конструктивными элементами: S-коннектором и R-коннектором (см. Рисунок 8).

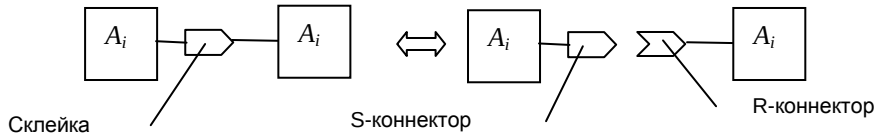


Рисунок 8. Операция разреза склейки

Программный агрегат для выделенного фрагмента АСПД-схемы представим теперь в виде процесса с внутренней нитевой структурой. С этой целью введем в рассмотрение абстрактный процесс, который назовем виртуальной потоковой машиной (ВПМ). Виртуальная потоковая машина предназначена для интерпретации работы выделенного фрагмента АСПД-схемы в среде POSIX-ориентированной операционной системы реального времени (в данном случае QNX 6).

Архитектура и функционирование ВПМ

Рассмотрим в начале структуру ВПМ (см. рис. 1) и порядок её запуска. Запуск ВПМ осуществляет *менеджер фрагмента (агрегата)* ВПМ-сети (родительский процесс), в который ВПМ входит. При запуске ВПМ ей в качестве аргумента передаётся ID канала менеджера фрагмента, предназначенного для приема сообщений от ВПМ фрагмента. Работа ВПМ начинается с нити **`main()`**, которая стартует первой при запуске ВПМ. Для учета некоторых специфических условий работы ВПМ, которые не могут быть отражены в АСПД-схеме, нить **`main()`** позволяет подключить к ней две функции (мезонины) **`b()`** и **`c()`.** Для выполнения подготовительных действий нить **`main()`** запускает мезонин **`b()`.** Для выполнения заключительных действий она запускает мезонин **`c()`**. В качестве мезонинов **`b()`** и **`c()`** выступают функции без параметров

(**int f(void)**), возвращающие целое значение 0 или 1. Значение 0 - ошибка, 1 - успешно. Для подключения этих мезонинов **main()** содержит два указателя, которые инициализируются адресами функций, соответствующих мезонинам **b()** и **c()**. Если необходимость в каком-либо мезонине отсутствует, то ставится заглушка (указатель инициализируется значением **NULL**). После выполнения подготовительных действий **b()** нить **main()** выполняет цикл запуска нитей-"подложек" (EXEC-нитей), несущих на себе мезонины **P()**, **T()** и **F()**. Завершив запуск EXEC-нитей, нить **main()** запускает нить - *менеджер ВПМ*. После этого нить **main()** переходит в состояние ожидания завершения работы менеджера ВПМ. После завершения работы менеджера ВПМ нить **main()** запускает мезонин **c(),** по завершению которого **main()** завершает свою работу, завершая тем самым работу ВПМ (процесс терминируется).

Работа менеджера ВПМ начинается с того, что он создает канал для приема сообщений (от менеджера фрагмента, в который входит ВПМ), создает служебное соединение со своим служебным каналом для последующего взаимодействия через него с OUT-нитями и устанавливает соединение с каналом менеджера фрагмента. Затем менеджер ВПМ, анализируя системную таблицу своих разъемов, последовательно создает входные каналы (разъемы) и заносит ID

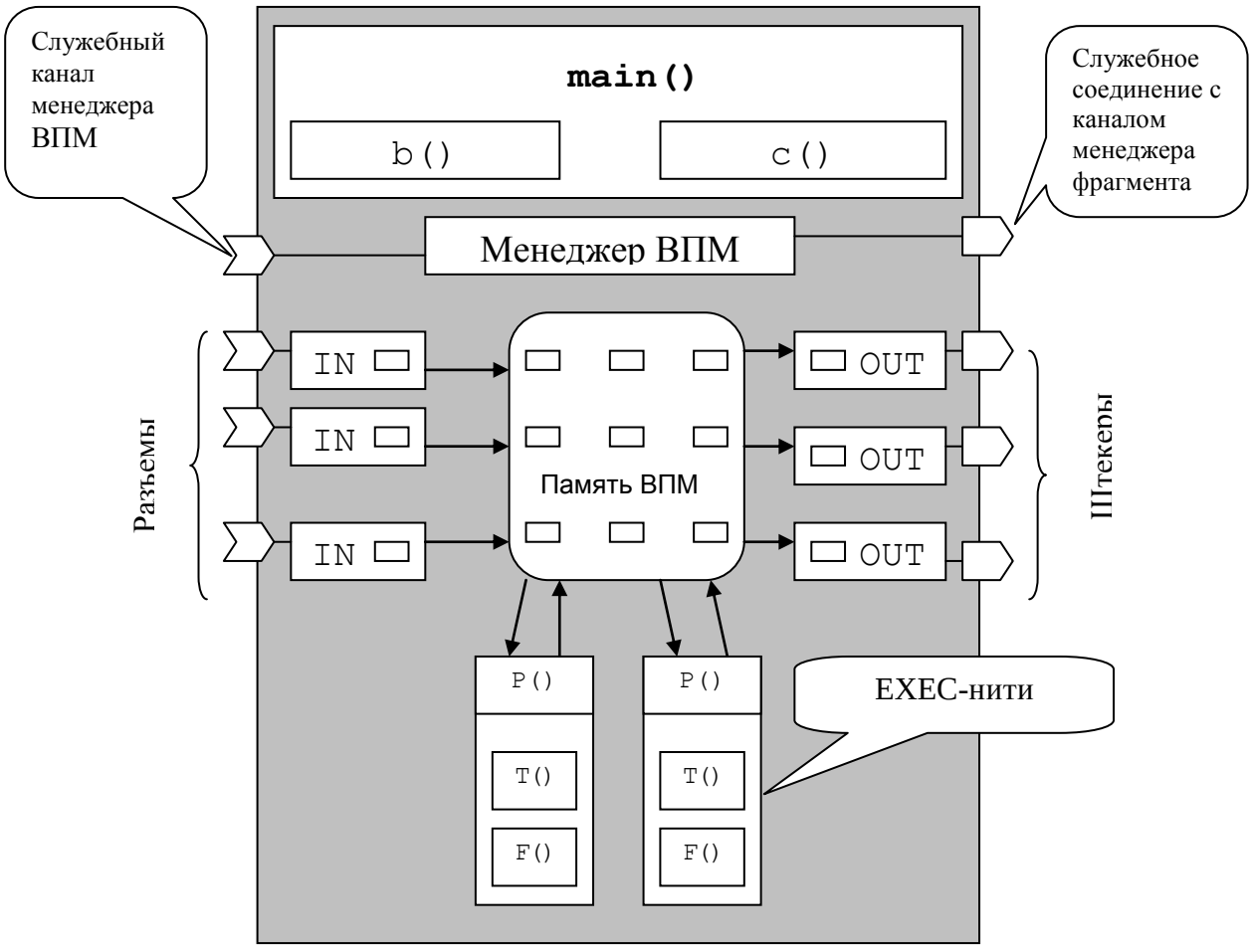


Рисунок 9. Структура ВПМ

каналов в системную таблицу разъемов. После этого менеджер запускает

обслуживающую разъем IN-нить, *соответствующую типу разъема*, передавая ей в качестве аргументов ID канала и размер внутреннего буфера, соответствующие разъему. Если разъем предназначен для приема событий (разъем событийный реального времени или прерываний), то передаются дополнительные аргументы, специфицирующие работу событийного разъема соответствующего типа. Для IN-нитей, планирующих и получающих импульсные сообщения событий реального времени (РВ), в качестве дополнительных аргументов передаются: параметры импульсного сообщения, момент времени начала генерации событий РВ, интервал времени циклических посылок событий РВ и т.п. Для IN-нитей, планирующих и получающих импульсные сообщения событий прерывания, в качестве дополнительных аргументов передаются: параметры импульсного сообщения, вектор прерывания. Затем менеджер ВПМ формирует и отсылает менеджеру фрагмента сообщение, содержащее информацию об ID каналов созданных разъемов, предназначенных для коммутации со штекерами смежных ВПМ (т.е. кроме событийных разъемов), и ID собственного служебного канала. Вслед за этим менеджер ВПМ по служебному соединению с каналом менеджера фрагмента посылает запрос на коммутацию штекеров ВПМ и переходит в свое основное состояние - ожидание прихода сообщений по своему служебному каналу. В этом состоянии он ожидает прихода сообщений от менеджера фрагмента с информацией об ID узла, ID смежной ВПМ, ID канала и номере штекера, который необходимо присоединить. По мере прихода этой информации, менеджер ВПМ создает соединение, соответствующие штекеру, и запускает нити штекера, передавая ей в качестве аргументов указатель на ID соединения штекера и ID служебного соединения для взаимодействия OUT-нитей с менеджером ВПМ при нарушении соединения. Сообщения могут прийти и от OUT-нитей, обслуживающих установленные соединения со смежными ВПМ., если возникают ошибки при выполнении ими функции посылки потоковых данных. В этом случае OUT-нить, послав менеджеру ВПМ сообщение о нарушении соединения, переходит в состояние ожидания ответа о его восстановлении.

Менеджер ВПМ завершает свою работу, когда получает терминирующее импульсное сообщение от менеджера фрагмента. После этого продолжает свое выполнение нить **`main().`** Она запускает на выполнение мезонин **`c()`** и затем завершает свою работу, терминируя тем самым процесс ВПМ.

Функционирование ВПМ

После того, как ВПМ загружена и включена менеджером фрагмента во фрагмент ВПМ-сети, она становится готовой к функционированию. Функционирование ВПМ инициируется поступлением потоков данных (токенов) на её входы. Работа ВПМ заключается в выполнении следующих действий:

- прием IN-нитями токенов в свои внутренние буферы и помещение принятых токенов в *память* ВПМ;
- обработка данных в памяти ВПМ EXEC-нитями посредством установленных на них мезонинов и формирование в памяти ВПМ результатных токенов;

- перемещение OUT-нитями результатных токенов из памяти ВПМ в собственные внутренние буферы.

Очевидно, что память ВПМ обеспечивает коммуникацию данных между нитями ВПМ и с позиции параллельного программирования является общим ресурсом для IN -нитей, OUT-нитей и EXEC-нитей. В связи с этим необходимо управлять доступом этих нитей к памяти ВПМ для обеспечения целостности потоков и корректности обработки данных, проходящих через ВПМ.

Память ВПМ

Память ВПМ в общем случае может состоять из нескольких регионов памяти. Регионы являются независимым друг от другого ресурсами памяти ВПМ. Каждая нить ВПМ использует только один регион памяти. Нити, использующие разные регионы памяти, не взаимодействуют между собой. Количество регионов памяти и их структура определяются структурой фрагмента АСПД-схемы, на основе которого сгенерирована ВПМ.

Каждый регион памяти ВПМ представляет собой множество объектов памяти (ОП) типа `dfm_mem_box_t`. Рассмотрим структуру, свойства и методы ОП. Структура ОП представлена на рис. 2.

Каждый ОП обладает следующими свойствами:

- тип;
- объем;
- состояние.

Тип и объем являются статическими свойствами ОП, которые задаются при

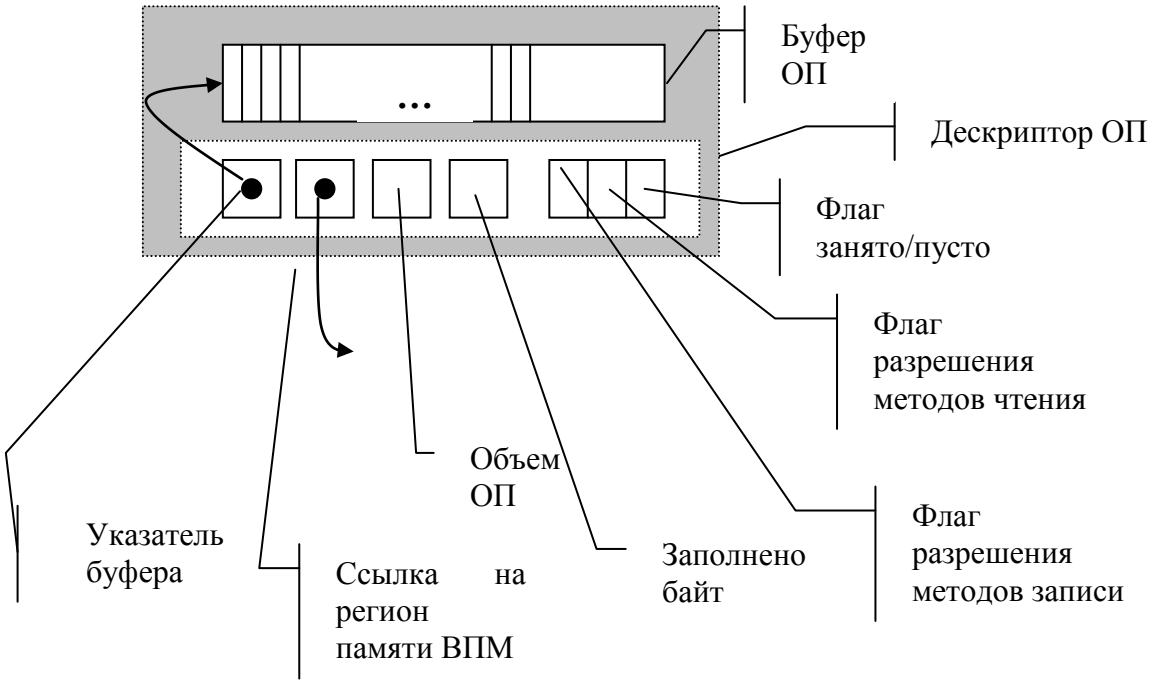


Рисунок 2. Структура ОП

создании ОП и не меняются в течение всего времени функционирования ВПМ. Объем ОП характеризует предельно возможное количество байт, которые могут

быть помещены в ОП. Величина объема ОП является результатом анализа АСПД-схемы ПСРВ. Состояние - динамическое свойство ОП, изменяющееся в процессе функционирования ВПМ. Тип ОП определяет динамику состояний ОП при применении методов ОП. Элемент памяти может находиться в одном из двух состояний:

1. ОП заполнен (`BUSY`).
2. ОП пуст (`EMPTY`).

По отношению к ОП определены методы (операции):

- открыть доступ к ОП - `dfm_mem_box_open();`
- закрыть доступ к ОП - `dfm_mem_box_close();`
- поместить блок байтов в ОП - `dfm_mem_box_insert();`
- извлечь блок байтов из ОП - `dfm_mem_box_extract();`
- переместить блок байтов из ОП1 в ОП2 - `dfm_mem_box_transfer();`
- копировать блок байтов из ОП - `dfm_mem_box_copy();`
- проверить состояние "занято/пусто" - `dfm_mem_box_state();`
- проверить состояние "готовность чтения" - `dfm_mem_box_flrd();`
- проверить состояние "готовность записи" - `dfm_mem_box_flwr();`
- проверить содержимое областей токена - `dfm_mem_box_contents();`

Метод `dfm_mem_box_open()` реализуется следующей функцией:

```
size_t // > 0 - объем ОП в байтах; < 0 - ошибка
dfm_mem_box_open(dfm_mem_box_t* mem,   //адрес            ОП
                 char   mode   //режим   использования   ОП
                 //mode     ≠     0     -     "вставить";
                 //mode       =       0       "извлечь"
                 );
```

Метод открывает ОП с целью последующего доступа к ОП или его содержимому и изменения его состояния. Изменение состояния ОП происходит при условии, что хотя бы один раз по отношению к ОП или его содержимому был применен один из методов: `dfm_mem_box_insert()`, `dfm_mem_box_extract()`, `dfm_mem_box_transfer()`. Фиксация изменения состояния ОП происходит при выполнении метода `dfm_mem_box_close()`. Если сразу за `dfm_mem_box_open()` выполнить `dfm_mem_box_close()`, то сохраняется исходное состояние ОП. Методы `dfm_mem_box_insert()`, `dfm_mem_box_extract()` и `dfm_mem_box_transfer()` выполняются только после выполнения метода `dfm_mem_box_open()`. Ошибка говорит о недопустимости открытия ОП в текущем состоянии для указанного режима использования.

Метод `dfm_mem_box_close()` реализуется следующей функцией:

```
size_t //Количество поступивших в ОП или извлеченных из ОП
       //байтов
dfm_mem_box_close(dfm_mem_box_t* mem   //адрес            ОП
             );
```

Метод закрывает доступ к ОП или его содержимому и изменяет состояние ОП, если хотя бы один раз по отношению к ОП или его содержимому был применен один из методов: `dfm_mem_box_insert()`, `dfm_mem_box_extract()`, `dfm_mem_box_transfer()`. В противном случае ОП сохраняет исходное состояние.

Метод dfm_mem_box_insert() реализуется следующей функцией:

```
size_t //Количество реально поместившихся байтов
dfm_mem_box_insert(dfm_mem_box_t* mem, //адрес           ОП
               size_t len,           //количество    байтов
               void* blockbyte,      //адрес блока   байтов
               unsigned int offset   //смещение
               );
```

Метод обеспечивает занесение копии указанного количества байтов, взятой из указанного блока байтов, в указанный ОП со смещением от начала ОП. Метод контролирует достижение предела ОП и не позволяет выйти за него. Метод возвращает реальное количество поместившихся в ОП байтов копии или `ERROR`, если осуществлена попытка применения метода без предварительного открытия ОП (`dfm_mem_box_open()`). Если метод возвращает значение меньшее `len`, то достигнут предел ОП.

Метод dfm_mem_box_extract() реализуется следующей функцией:

```
size_t //Количество реально извлеченных байтов
dfm_mem_box_extract(dfm_mem_box_t* mem,    //адрес       ОП
                size_t len,           //количество    байтов
                void* buf,            //адрес блока   байтов
                unsigned int offset   //смещение
                );
```

Метод обеспечивает извлечение указанного количества байтов из указанного ОП, начиная с заданного смещения от начала ОП, и помещает их в указанный буфер. Метод контролирует достижение предела ОП и не позволяет выйти за него. Метод возвращает реальное количество извлеченных байтов или `ERROR`, если осуществлена попытка применения метода без предварительного открытия ОП (`dfm_mem_box_open()`). Если метод возвращает значение меньшее `len`, то достигнута граница ОП. Если на место указателя `buf` поместить указатель `mem` или `NULL`, или задать `len=0`, то извлечение считается выполненным по отношению к ОП, но извлеченные данные не меняют своего положения.

Метод dfm_mem_box_transfer() реализуется следующей функцией:

```
size_t //Количество реально перенесенных байтов
dfm_mem_box_transfer(dfm_mem_box_t* memsrc,     //адрес  ОП-
                источника
                dfm_mem_box_t* memdest,    //адрес       ОП-
                приемника
                );
```

Метод обеспечивает извлечение токена из указанного ОП-источника и помещает его в указанный ОП-приемник. Метод контролирует, чтобы ОП-приемника обладал достаточным объемом. Метод возвращает -1, если недостаточно места, или ERROR, если осуществлена попытка применения метода без предварительного открытия ОП-источника или ОП-приемника (dfm_mem_box_open()).Если указаны одинаковые указатели, то извлечение считается выполненным по отношению к ОП, но извлеченные данные не меняют своего положения.

Метод dfm_mem_box_copy() реализуется следующей функцией:

```
size_t //Количество реально скопированных байтов
dfm_mem_box_copy(dfm_mem_box_t* mem, //адрес              ОП
            size_t len,              //количество    байтов
            void* buf,               //адрес блока  байтов
            unsigned int offset      //смещение
            );
```

Метод обеспечивает копирование указанного количества байтов из указанного ОП, начиная с заданного смещения от начала ОП, и помещает их в указанный буфер. Состояние ОП при этом не меняется. Метод контролирует достижение предела ОП и не позволяет выйти за его границы. Метод возвращает реальное количество скопированных байтов. Метод не требует предварительного открытия ОП функцией dfm_mem_box_open().

Метод dfm_mem_box_state() реализуется следующей функцией:

```
int //Состояние ОП: ≠0 - BUSY, 0 - EMPTY
dfm_mem_box_state(dfm_mem_box_t* mem   //адрес           ОП
             );
```

Метод обеспечивает проверку состояния указанного ОП. Состояние ОП при этом не меняется. Метод не требует предварительного открытия ОП функцией dfm_mem_box_open().

Метод dfm_mem_box_rdfl() реализуется следующей функцией:

```
int //Готовность чтения ОП: ≠0 - READY, 0 - NOREADY
dfm_mem_box_rdfl(dfm_mem_box_t* mem    //адрес           ОП
            );
```

Метод обеспечивает проверку готовности чтения указанного ОП. Состояние ОП при этом не меняется. Метод не требует предварительного открытия ОП функцией `dfm_mem_box_open()`.

Метод dfm_mem_box_wrfl() реализуется следующей функцией:

```
int //Готовность записи ОП: ≠0 - READY, 0 - NOREADY
dfm_mem_box_wrfl(dfm_mem_box_t* mem    //адрес            ОП
              );
```

Метод обеспечивает проверку готовности чтения указанного ОП. Состояние ОП при этом не меняется. Метод не требует предварительного открытия ОП функцией `dfm_mem_box_open()`.

- проверить состояние "готовность записи" - `dfm_mem_box_wrfl();`

Метод dfm_mem_box_contents() реализуется следующей функцией:

```
dfm_mem_teg_t*                    //Адрес тега токена
dfm_mem_box_contents(dfm_mem_box_t *mem,   //адрес       ОП
                 void** memdata,//указатель  адреса  данных
                 );
```

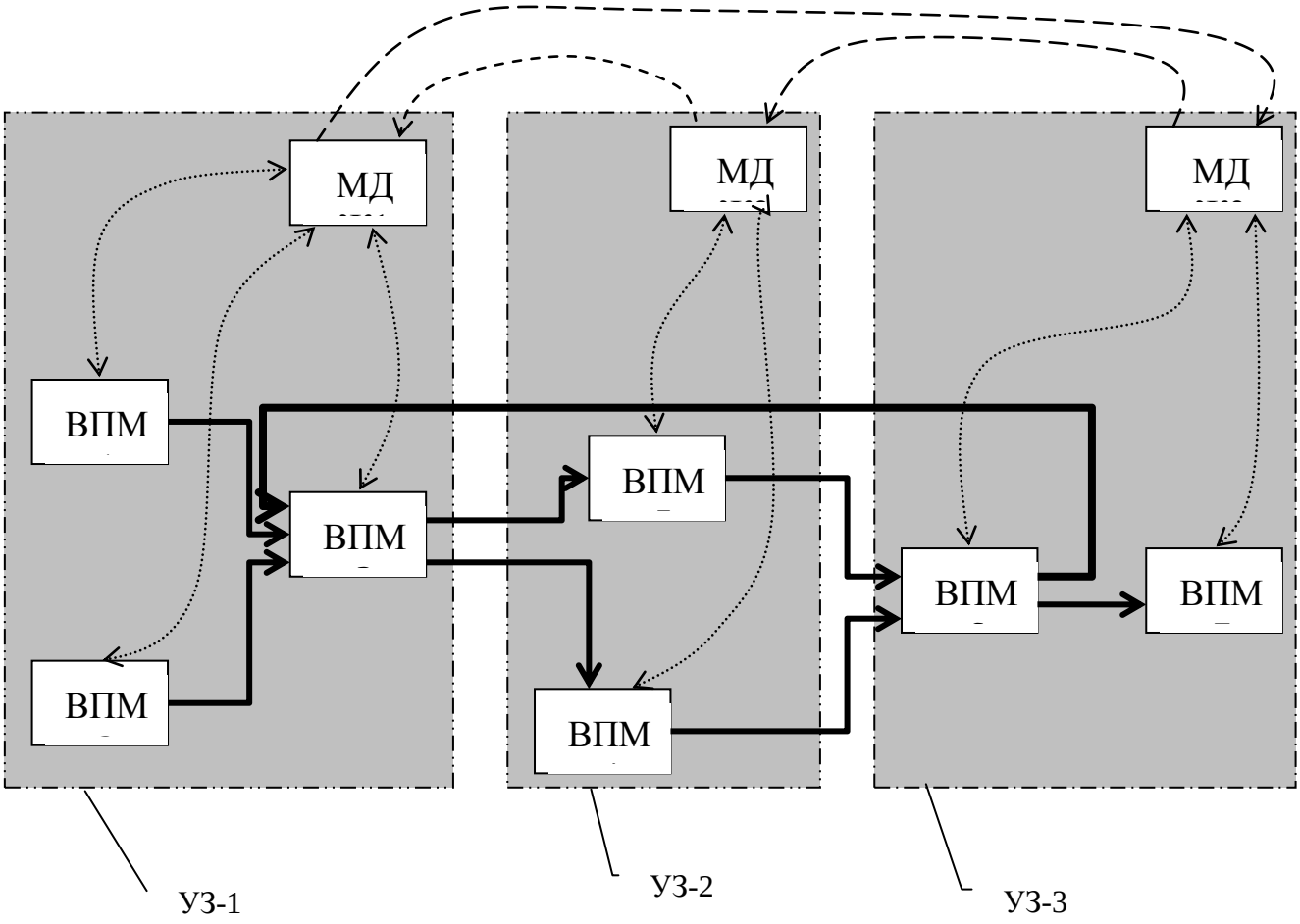
Метод обеспечивает получение ссылок на область тега и область данных токена, находящегося в указанном ОП. Состояние ОП при этом не меняется.

Моделирование ПСРВ на агрегатном уровне. А-сеть

Понятие программного агрегата

После объединения акторов и получения ВПМ-сети встает вопрос о распределении ВПМ-сети по узлам в вычислительной сети. Для этого необходимо провести следующий этап – переход от ВПМ-сети к А-сети. А-сеть – это совокупность агрегатов, представляющих собой совокупность виртуальных потоковых машин. Каждый агрегат является фрагментом ВПМ-сети (ВПМ-подсетью). Агрегат полностью помещается в узел сети. Помещение частей агрегата в разные узлы невозможно. Помещение нескольких агрегатов в один узел допускается. Совокупность агрегатов является А-сетью. Каждый агрегат представляется совокупностью ВПМ и менеджером фрагмента – служебным процессом, отвечающим за загрузку, настройку на штатный режим работы и завершения работы агрегата. А-сеть представляет собой совокупность агрегатов, загрузчик ПСРВ и журнал.

Каждый из агрегатов детально специфицируется. На основе спецификаций производится генерация исходных текстов ПО на языке Си. Далее после компилирования появляется проект ПСРВ, представляющий собой совокупность



исполняемых модулей и конфигурационных файлов в файловой системе на одном из узлов сети.

Внешние ресурсы ВПМ-сети как абстрактные устройства А-сети

Виртуальное внешнее устройство ВПМ есть логическое обобщение понятия внешнего устройства (ВУ) вычислительных систем. Номенклатура ВУ, применяемых в СРВ, очень широка – это устройства связи с объектом, устройства отображения информации, устройства связи человека с машиной, средства системного времени, каналы машинной связи. Поэтому для построения логической модели ВУ необходимо выявить общие свойства, присущие способам взаимодействия различных ВУ с вычислительными системами.

Относительно вычислительной системы ВУ можно логически разделить на источники и приемники данных (конструктивно они могут находиться в одном корпусе). Источники порождают данные, которые потребляются вычислительной системой, а приемники получают данные из системы и поглощают их (отображают, регистрируют, накапливают, передают другой системе). Анализируя различные классы внешних устройств распределенных СРВ, можно заметить, что независимо от типа все ВУ специфицируются множеством адресуемых регистров. С каждым устройством связаны два типа регистров: регистры управления и состояния (РУС), регистры данных (РД). Число и тип РУС и РД зависят от функций устройства. Регистры данных обеспечивают временное хранение данных, передаваемых в вычислительную систему или из нее. Посредством РУС обеспечиваются следующие функции взаимодействия системы с ВУ:

- контроль готовности ВУ выполнить свои функции;
- установка функции ВУ;
- проверка состояния устройства (“занято”, если ВУ в текущий момент выполняет заданную функцию, и “сделано”, если функция выполнена нормально);
- контроль ошибок взаимодействия;
- управление режимом взаимодействия.

Для приемника в состоянии “сделано” РД свободны и готовы принять новые данные, а для источника - наоборот. Взаимодействие может вестись по инициативе от ВУ (по прерыванию) и по инициативе от вычислительной системы. В последнем случае система либо ожидает состояния ВУ "сделано", либо нет. Режим взаимодействия по прерыванию и по готовности "сделано" называют асинхронным взаимодействием. В противном случае - синхронным. Таким образом, взаимодействие с ВУ заключается в установке функции и контроле ее выполнения (выполнено успешно, выполнено с ошибкой или не выполнено). В этом смысле любое ВУ является источником данных контроля функционирования. Обозначим через S_R – элементы памяти ВУ, предназначенные для порождения данных об успешном выполнении функции, а через S_E – об аварии или ошибке взаимодействия. Через S_D обозначим ЭП для порождения или поглощения информационных данных. Тогда каждому ВУ можно поставить в соответствие тройку $< S_R, S_E, S_D >$ и множество функций взаимодействия с ВУ – интерфейс взаимодействия. Такое сопоставление выполняется посредством специальных информационных процессов, называемых

драйверами устройств. Они строятся на основе механизма абстрактного типа данных и в этом смысле их можно считать виртуальными внешними устройствами.

В связи с выше изложенным, под множеством виртуальных внешних устройств ВПМ будем понимать специальные информационные процессы, предназначенные для порождения и поглощения данных и представляемые тройкой $< S_R, S_E, S_D >$, Заметим, что структура S_R, S_E, S_D специально не оговариваются, и, следовательно, могут иметь сложный вид. Тогда такой же тройкой или их совокупностью можно описывать любую систему, осуществляющую порождение или поглощение данных (например, подсистему сбора или регистрации данных в СРВ).

Таким образом, показано, что тройками $< S_R, S_E, S_D >$ можно описать любой источник или приемник данных, независимо от степени его структурной сложности.

Внешние ресурсы ВПМ-сети (стоки, истоки) не объединяются во фрагменты на этапе объединения ВПМ в отдельные агрегаты и являются внешними ресурсами и для агрегатной сети. Количество стоков и истоков в А-сети остается таким же, как и в сети виртуальных потоковых машин. При этом любой исток (сток) является входным (выходным) для того агрегата А-сети, в состав которого входит ВПМ, с которой он связан в ВПМ-сети.

Спецификация распределенного вычислительного комплекса

Распределенный вычислительный комплекс в общем случае представляет собой локальную вычислительную сеть (ЛВС), состоящую из произвольного числа компьютеров и протоколов передачи данных, с помощью которых они взаимодействуют. Каждый узел ЛВС обладает определенным набором свойств – тип и частота процессора, объем оперативной памяти, объем доступного дискового пространства, набор доступных протоколов обмена данными. Каждый агрегат и внешние ресурсы А-сети помещаются в один из узлов вычислительного комплекса. Возможно размещение на одном узле нескольких агрегатов.

В настоящее время под вычислительной сетью понимается совокупность территориально рассредоточенных ЭВМ и терминалов, объединенных сетью передачи данных. Она включает в себя сеть ЭВМ и терминальную сеть. Под сетью ЭВМ подразумевается территориально рассредоточенные ЭВМ, объединенные сетью передачи данных. Терминальная сеть – это совокупность средств и каналов связи, используемых для подключения удаленных терминалов к сети ЭВМ. Совокупность средств для передачи и распределения данных называется сетью передачи данных. Основным признаком классификации в настоящее время является признак территориальной рассредоточенности вычислительных ресурсов сетей. По этому признаку вычислительные сети делят на четыре класса:

1) географически распределенные вычислительные сети, элементы которых удалены друг от друга на значительные расстояния, измеряемые тысячами километров;

2) локально распределенные сети, элементы которых могут быть соединены сравнительно короткими линиями связи и сосредоточены в пределах одного здания или предприятия;

3) компактно распределенные сети с элементами в пределах одной лаборатории, помещения или внутри одной установки;

4) параллельные вычислительные системы с тесно связанными элементами, размещенными в одной стойке или на одной печатной плате.

В таких системах как, распределенные системы обработки данных, распределенные системы управления, вычислительные системы и им подобные, акцент переносится с территориальной рассредоточенности аппаратных компонентов сетей на децентрализованное параллельное выполнение совокупности взаимодействующих функций обработки данных. В основе таких систем лежат, так называемые, распределенные вычислительные комплексы. Под распределенными вычислительными комплексами (РВК) будем понимать комплексы вычислительных средств, включающие в свой состав средства связи с объектами физической природы, с пользователями и состояние из множества параллельно функционирующих микропроцессорных модулей, ЭВМ, объединенных в единый вычислительный комплекс каналами связи /2/. Главным признаком РВК, выделяющим их из общего множества вычислительных сетей, является использование их для реализации единой технологии обработки данных, соответствующей общим целям и задачам исследования, испытания или управления информационным объектом.

Дадим теперь более точное определение распределенных СРВ. Под распределенными СРВ будем понимать системы, построенные на базе РВК и предназначенные для выполнения процессов сбора данных, анализа информации и управления объектом, со сложной информационной и/или территориально рассредоточенной структурой, в темпе заданного системного времени /5/. С точки зрения территориальной рассредоточенности распределенные СРВ могут представлять собой локальные, компактные сети, параллельные вычислительные системы или их различные сочетания.

Для эффективной реализации распределенных СРВ, представленных АСПД-схемой, на основе РВК необходимо разработать принцип построения распределенного программного обеспечения, адекватно реализующего способ потоковой обработки данных АСПД-схемы.

Структуры сетевых программных систем, реализуемых в традиционных вычислительных сетях, основаны на концепции обобществления вычислительных ресурсов (ЭВМ, внешних устройств, программ, данных и т.п.). В соответствии с ней вычислительная сеть представляется в виде реальной или абстрактной коммуникационной среды, посредством которой активные сетевые информационные процессы осуществляют доступ к вычислительным ресурсам сети и взаимодействуют друг с другом.

ЛИТЕРАТУРА

1. Месарович М., Мако Д., Такахара И. Теория иерархических многоуровневых систем. – М.: Мир, 1973. – 344с.
2. Gertenbach W.P. A Hierarchical Organization of Real-Time Multicomputer Process Control Systems // Real-Time Data Handling & Process Control: Europ.Symp. 1-st, Berlin/West/, 1979, Brussels. – 1980. – P. 443-446.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №1
(2 часа)

ТЕМА: «Введение в технологии промышленного программирования».

1. Особенности программирования систем промышленной автоматизации (СПА). Пример задания на программирование промышленной системы с учетом особенностей функционирования в режиме реального времени.
2. Операционные системы реального времени (ОСРВ). Требования к ОСРВ. POSIX-стандарт.
3. Программные системы (приложения) реального времени (ПСРВ). Особенности программирования и отладки ПСРВ.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №2
(2 часа)

ТЕМА: «Инструментальные средства промышленного программирования».

1. Платформа разработки приложений реального времени QNX Momentics IDE 4.7.
2. Инструментальная и исполнительная системы.
3. Порядок установки инструментальной системы QNX Momentics IDE 4.7.
4. Порядок установки исполнительней системы QNX6 на базе виртуальной машины в ОС семейства Windows.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №3
(2 часа)

ТЕМА: «Файловая система ОСРВ QNX6».

1. Базовая структура файловой системы ОСРВ QNX6. Монтирование устройств.
2. Текущий и корневой каталоги. Управление каталогами.
3. Пользователи и группы.
4. Типы файлов. Разграничение прав доступа к файлам. Создание файлов и управление правами доступа с помощью системной функции open().

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №4
(2 часа)

ТЕМА: «Процессная структура ПСРВ».

1. Структура программных систем реального времени. Программные модули, процессы, нити.
2. Типы процессов, особенности их функционирования.
3. Способы запуска параллельных процессов и управление процессами.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №5
(2 часа)

ТЕМА: «Организация межпроцессного взаимодействия».

1. Программный интерфейс ОСРВ QNX для организации взаимодействия между процессами посредством сообщений.
2. Создание/Удаление канала.
3. Установление/Удаление соединений с каналом.
4. Посылка/Прием сообщения.
5. Посылка ответа. Сценарии ответов.
6. Управление приемом и передачей сообщений

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №6
(2 часа)

ТЕМА: «Нитиевая структура процессов».

1. Нитиевая структура процессов. Нить main().
2. Прототип функции и свойства нити.
3. Формирование свойств и запуск нити.
4. Приоритет и дисциплина диспетчеризации нити.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №7
(2 часа)

ТЕМА: «Управление нитями».

1. Методы и функции синхронизации нитей.
2. Присоединение.
3. Барьеры.
4. Блокировки чтения/записи.
5. Мутексы.
6. Ждущие блокировки.
7. Условные переменные.
8. Семафоры.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №8
(2 часа)

ТЕМА: «Использование процессами системной памяти.
Память внешних устройств».

1. Управление памятью вне адресного пространства процессов.
2. Именованная оперативная память.
3. Доступ к именованной памяти.
4. Доступ к памяти устройств ввода/вывода.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №9
(2 часа)

ТЕМА: «Механизм сигналов».

1. Понятие сигнала. Реакции процесса на приход сигнала. Обработчик сигнала.
2. Механизм надежных сигналов.
3. Набор сигналов и маска блокирования.
4. Посылка и доставка сигнала процессу. Блокирование сигналов.
5. Управление сигналами. Ожидание заблокированных сигналов.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №10
(2 часа)

ТЕМА: «Синхронизация нитей с реальным временем».

1. Синхронизация нитей с реальным временем. Системное реальное время. Разрешающая способность РВ.
2. Установка значений абсолютного и интервального времени.
3. Таймеры. Создание/Удаление таймера.
4. Типы уведомления нитей о временн'ых событиях.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №11
(2 часа)

ТЕМА: «Управление прерываниями».

1. Организация прерываний от устройств ввода/вывода.
2. Управление прерываниями в QNX. Функция-обработчик прерываний.
3. Механизм обработки прерываний в процессах. Установка обработчика прерываний.
4. Программный шаблон процесса с обработчиком прерываниц.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №12
(2 часа)

ТЕМА: «Проектирование ПСРВ».

1. Общая структура автоматизированной системы реального времени (АСРВ).
2. Иерархическая структура АСРВ.
3. Спецификация ПСРВ как абстрактного программного агрегата АСРВ.
4. Спецификация окружения программного агрегата: абстрактные *истоки* и *стоки* данных (абстрактные объекты памяти) как объекты абстрактной внешней памяти, процессы окружения абстрактного программного агрегата АСРВ.
5. Режимы взаимодействия абстрактного программного агрегата с истоками и стоками.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №13
(2 часа)

ТЕМА: «Абстрактная потоковая память. Темпоральная модель данных.»

1. Модель абстрактной потоковой памяти. Типы объектов потоковой памяти. Методы доступа к объектам потоковой памяти.
2. Темпоральная модель данных. Датированные значения.
3. Модель потоковых данных - *токены*. Типы токенов.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №14
(2 часа)

ТЕМА: «Моделирование функциональной структуры ПСРВ».

1. Концепции управления параллельными процессами: потоки управлений (control-flow), потоки данных (data-flow).
2. Особенности структур ПСРВ с потоками управлений.
3. Особенности структур ПСРВ с потоками данных. Схемы потоков данных.
4. Теоретическая эквивалентность потоков управлений и потоков данных.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №15
(2 часа)

ТЕМА: «Моделирование ПСРВ на функциональном уровне».

1. Концепция моделирования функциональной структуры ПСРВ в виде схемы потоков данных.
2. Построение схемы потоков данных методом иерархической декомпозиции абстрактного программного агрегата.
3. Язык схем асинхронных потоков данных (АСПД-схем). Объекты памяти АСПД-схемы: *истоки*, *стоки*, *проходная* память.
4. Акторы. Состав и формальная спецификация базовых акторов АСПД-схемы.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №16
(2 часа)

ТЕМА: «Моделирование ПСРВ на процессном уровне».

1. Понятие виртуальной потоковой машины (ВПМ)
2. Объектная структура ВПМ.
3. Функционирование ВПМ.
4. Абстрактный программный агрегат как сеть виртуальных потоковых машин (ВПМ-сеть).

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №17
(2 часа)

ТЕМА: «Моделирование ПСРВ на агрегатном уровне».

1. Абстрактная спецификация распределенного вычислительного комплекса (РВК). Задача распределения компонентов ВПМ-сети по узлам распределённого вычислительного комплекса.
2. Понятие программного агрегата (ПА). Взаимодействие программных агрегатов. Понятие сети программных агрегатов. Абстрагирование ВПМ-сети в сеть программных агрегатов (ПА-сеть).
3. Размещение ПА-сети в РВК.

ПРАКТИЧЕСКОЕ (СЕМИНАРСКОЕ)
ЗАНЯТИЕ №18
(2 часа)

ТЕМА: «Организация ПСРВ на программно-модульном уровне»

1. Понятие программно-модульной структуры ПСРВ – проект ПСРВ.
2. Файловая организация ПСРВ на программно-модульном уровне.
3. Каталоги программных модулей ПСРВ.
4. Каталоги модулей спецификации АСПД-схемы.
5. Каталоги модулей спецификации виртуальных потоковых машин.
6. Каталоги модулей спецификации программных агрегатов.
7. Каталог с модулем спецификации ПМ-сети и модулем спецификации РВК.
8. Каталог с модулем загрузки проекта ПСРВ в РВК.

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)"

Кафедра программных систем

А.В. Баландин

Лабораторная работа

Установка и настройка среды разработки приложений реального времени QNX Software Development Platform

Методические указания

Самара
2012

Содержание

Цель работы: Установка и освоение платформы разработки приложений для ОСРВ QNX - QNX Software Development Platform (QNX SDP)

Порядок выполнения работы

1. Установить и настроить QNX Momentics IDE.
2. Установить виртуальную машину VMware под QNX Neutrino.
3. Настроить сетевые подключения инструментальной и целевой машины.
4. Создать разделяемую папку с общим доступом для инструментальной и целевой машины.
5. Создать и запустить на выполнение стартовый программный проект.
6. Результаты продемонстрировать преподавателю.

Введение

Операционная система ***QNX Neutrino*** в первую очередь предназначена для реализации приложений реального времени. Особенностью таких приложений является то, что они являются программной составляющей автоматизированных систем реального времени. Как правило, работа таких приложений осуществляется в специализированных компьютерах или вычислительных устройствах, не содержащих программных средств для разработки приложений. Поэтому разработка приложений обычно осуществляется на персональном компьютере с использованием специальных программных средств для разработки приложений (инструментальный компьютер - ПК разработчика), а их исполнение и отладка – в специализированных вычислительных устройствах (целевая система). При этом ПК разработчика и целевая система связываются каналом связи для обмена данными:

Для программирования приложений, реализуемых в операционной системе реального времени ***QNX Neutrino***, компания QNX Software Systems International Corporation предлагает платформу разработки - ***QNX Software Development Platform (QNX SDP)***. Эта платформа включает в себя набор ***QNX Momentics Tool Suite***, который в свою очередь включает в себя всё необходимое для разработки приложений под ОС ***QNX Neutrino***: компилятор, компоновщик, библиотеки и другие компоненты операционной системы, скомпилированные под все архитектуры процессоров, поддерживаемые ***QNX Neutrino***. Скачать ***QNX SDP*** можно на официальном сайте ***www.qnx.com*** в разделе ***Downloads → QNX Software Development Platform 6.5.x.***

Платформу QNX SDP можно установить на инструментальный компьютер, работающий под управлением Windows XP, Windows Vista, Windows 2000, Windows 7 или Linux. Её можно установить также и на компьютер с ОСРВ QNX Neutrino для проведения резидентной разработки.

Для ОС Windows установочный файл ***QNX SDP*** (файл ***qnxsdp-6.5.0-201007091524-win32.exe***) можно скачать по адресу:

http://www.qnx.com/download/feature.html?programid=21180

Для ОС Linux установочный файл ***QNX SDP*** (файл ***qnxsdp-6.5.0-201007091524-linux.bin***) можно скачать по адресу:

http://www.qnx.com/download/feature.html?programid=21179

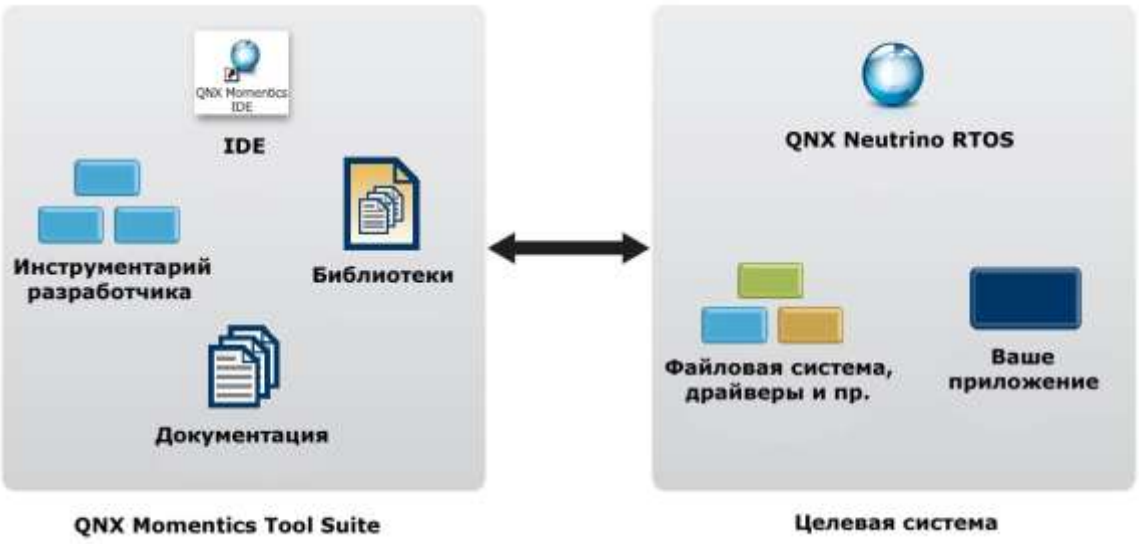
или

http://www.qnx.com/download/group.html?programid=20905

Для установки ***QNX SDP*** понадобится *лицензионный ключ*. Получить ключ для некоммерческого использования можно свободно, посетив страничку по адресу:

http://www.qnx.com/products/evaluation/non-commercial_developer.html

После того, как ***QNX SDP*** на инструментальном компьютере установлена, разработка приложения производится в среде ***QNX Momentics Tool Suite***. При этом запуск и отладка разрабатываемого приложения производится непосредственно на целевой системе и управляется по сети.



В роли целевой системы могут выступать:

- ***Персональный компьютер.*** Для этого понадобится выделить раздел на жёстком диске объёмом до 3 ГБ и установить в нём ОС ***QNX Neutrino***. Микроядро ***QNX Neutrino*** (***procnto***) требует порядка 700КБ. Добавляя к нему нужные компоненты, можно построить любую систему от лёгкой встраиваемой системы до полноценной настольной системы, которой потребуется около 300МБ свободного места.
- ***Виртуальная машина.*** Можно установить ***QNX Neutrino RTOS*** на виртуальную машину, созданную непосредственно в инструментальном компьютере. Это наиболее удобный вариант для использования в учебном процессе. В данном руководстве будет рассматриваться именно этот вариант. Образ уже настроенный и готовый для работы виртуальной машины ***VMware*** можно скачать по адресу: *http://www.qnx.com/download/feature.html?programid=21189*

Примечание. Можно установить ***QNX Neutrino RTOS*** и на системы с другими архитектурами процессора (PPC, ARM, MIPS или SH CPU). Подробнее об этом можно узнать на сайте сообщества ***QNX Foundry27***: *http://community.qnx.com/*.

Создание программного проекта и формирование программных модулей осуществляется в среде разработки ***QNX Momentics IDE*** в основной системе (ОС Windows), а последующее исполнение и отладка разработанной программы – в целевой системе - виртуальная машина под управлением ОС QNX.

Среда разработки ***QNX Momentics IDE*** построена на основе оболочки ***Eclipse***, которая широко используется в системах программирования на многих языках: Java, C/C++, Pyton и пр. Поэтому интерфейс ***QNX Momentics IDE*** оказывается интуитивно понятным тем, кто раньше работал в ***Eclipse***.

Установка и настройка QNX Momentics IDE

Для установки ***QNX Momentics IDE*** под Windows следует запустить файл ***qnxsdp-6.5.0-201007091524-win32.exe***. Рекомендуется установить ***QNX Momentics IDE*** в папку по умолчанию (**C:\QNX650**). Можно использовать и другую папку (например, в **C:\Program Files**), не содержащую в названии *кириллических символов и пробелов*. В противном случае, возможны проблемы при построении проектов. В дальнейшем для избегания проблем следует неукоснительно руководствоваться несколькими простыми правилами:

- не использовать *кириллицу* в имени пользователя;
- не использовать *кириллицу* в именах папок/файлов/проектов;
- не использовать *пробелов* в именах файлов/папок/пробелов ни при установке, ни при работе.

После установки ***QNX Momentics IDE*** на рабочем столе ОС Windows появится ярлык:

При запуске программа просит указать путь для размещения *рабочего каталога*, в котором будут храниться проекты, настройки и другие файлы. Для удобства работы целесообразно создать в ОС Windows разделяемую папку и назначить её в качестве рабочего каталога ***QNX Momentics IDE***. Эту папку можно будет затем примонтировать к файловой системе ОС ***QNX*** в целевой машине, чтобы получить доступ к программным модулям проекта для запуска процессов в целевой машине. При необходимости можно будет изменить рабочий каталог, однако всегда можно будет изменить рабочий каталог в настройках среды.

Установка VMware и настройка виртуальной машины под QNX Neutrino

Виртуальная машина — это выделенные аппаратные ресурсы инструментального компьютера, которые могут использоваться для установки на их основе любой другой операционной системы, как на отдельном компьютере (***целевой*** операционной системы). Виртуальная машина эмулируется установленной на инструментальном компьютере специальной программной системой. Виртуальная машина исполняет машинный код реального процессора. Помимо процессора, виртуальная машина может эмулировать работу как отдельных компонентов аппаратного обеспечения, так и целого реального компьютера (включая BIOS, оперативную память, жёсткий диск, сетевые адаптеры и другие периферийные устройства). В этом случае в виртуальную машину, как на реальный компьютер, можно устанавливать различные операционные системы (в нашем случае, ***QNX Neutrino***).

На одном инструментальном компьютере могут быть установлены и одновременно функционировать несколько виртуальных машин (это, например, используется для имитации локальной сети виртуальных машин, работающих под ***QNX Neutrino***).

Последнюю версию ***VMware Player*** можно скачать с официального сайта по адресу:

http://www.vmware.com/go/downloadplayer/

Ссылка для скачивания продукта будет доступна после регистрации. Регистрация является бесплатной. Последняя доступная на момент написания данного руководства версия была ***VMware Player 3.1.4*** (предоставляется студентам при выполнении цикла лабораторных работ).

Выберите (или скачайте самостоятельно) исполняемый файл с подходящей версией ***VMware Player***:

- ***VMware-player-3.1.4-385536.exe*** - 32- и 64-разрядная версия для ОС Windows,
- ***VMware-Player-3.1.4-385536.i386.bundle*** - 32-разрядная версия для ОС Linux,

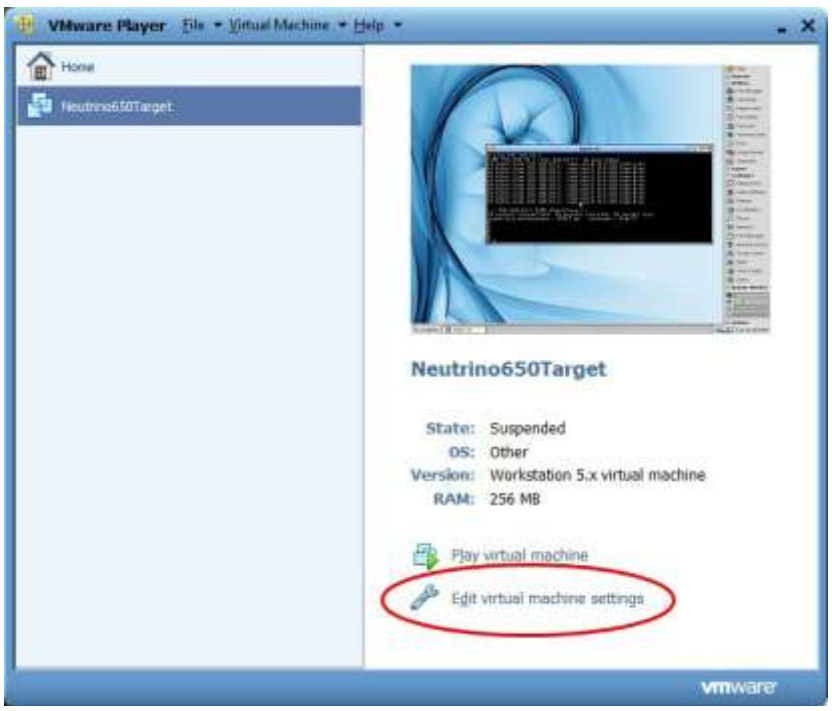
- ***VMware-Player-3.1.4-385536.x86_64.bundle*** - 64-разрядная версия для ОС Linux.

Установите программную систему ***VMware Player*** на свой компьютер. Для установки на виртуальную машину ОС ***QNX Neutrino*** распакуйте архив ***QNX_Eval_RT-201007091524.zip*** с образом ***QNX Neutrino*** для виртуальной машины в отдельную папку с именем, например, ***QNX1***. Если понадобится виртуальная локальная сеть целевых ***QNX***-машин, то для создания следующей целевой ***QNX***-машины можно создать копию папки ***QNX1***, назвав её, например, ***QNX2***. Несколько виртуальных машин могут понадобиться в том случае, если необходимо будет организовать межпроцессное взаимодействие распределённых в локальной сети параллельных процессов.

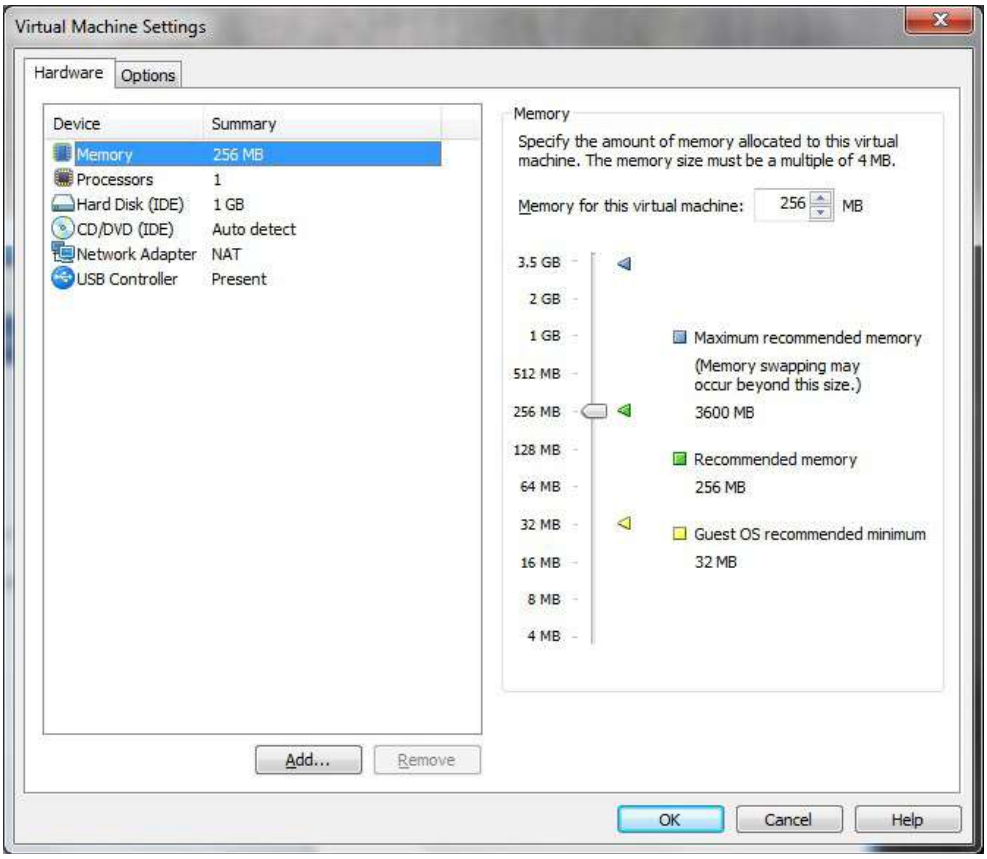
Для запуска целевой ***QNX***-машины запустите ***VMware Player***, появится окно:

Нажмите кнопку ***Open a Virtual Machine*** и укажите путь к файлу ***Neutrino650Target.vmx***, располагающемуся, например, в папке с первой виртуальной машиной ***QNX1***. Аналогично, если необходимо, то же сделайте и для второй виртуальной машины ***QNX2***.

Далее необходимо выполнить настройку параметров виртуальной машины. Для этого следует выделить нужную виртуальную машину и нажать кнопку ***Edit Virtual Machine Settings***:



Появится окно с настройками виртуальной машины:

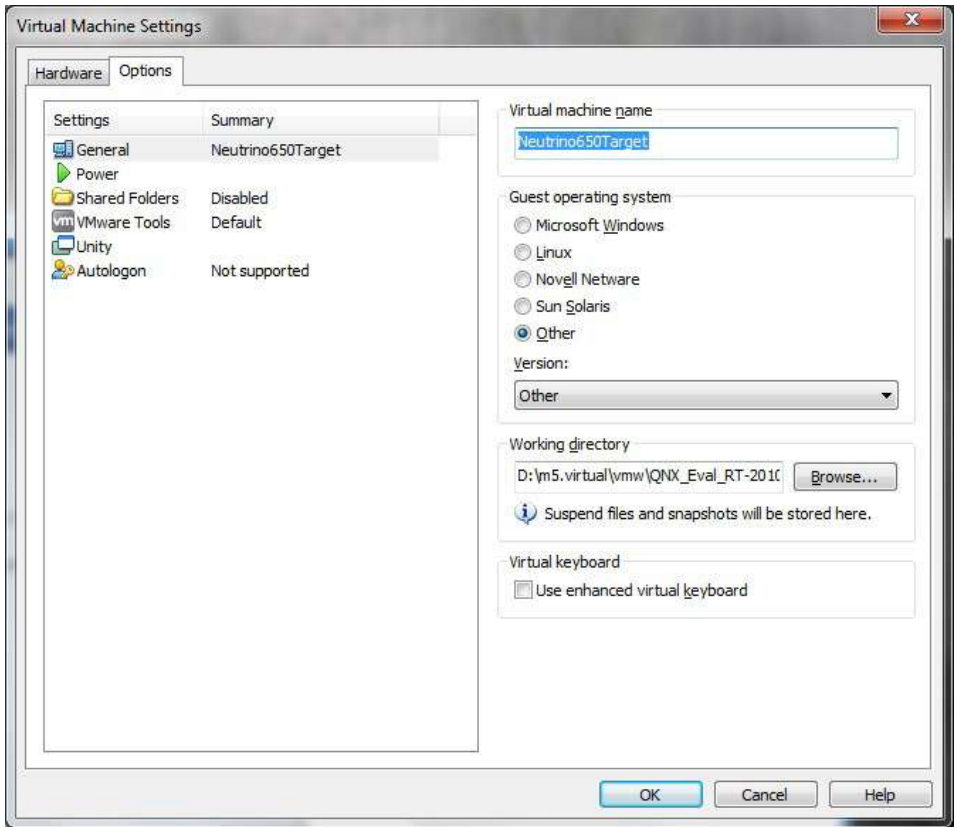


На вкладке ***Hardware*** представлены настройки аппаратной части виртуальной машины:

- ***Memory.*** Задаёт объём оперативной памяти, выделенный под виртуальную машину. При одновременном запуске нескольких виртуальных машин следите за тем, чтобы объём суммарной выделенной под них оперативной памяти был меньше свободного доступного объёма памяти в вашей основной системе.
- ***Processor.*** Задаёт количество ядер виртуального процессора. Может принимать значения от 1 до N, где N - количество логических ядер в ЦПУ компьютера.
- ***Hard Disk (IDE).*** Добавляет в виртуальную машину жёсткий диск. Здесь вы можете узнать объём свободного и занятого пространства.
- ***CD/DVD (IDE).*** Добавляет привод для компакт-дисков. Он позволяет как читать диски с вашего реального физического дисковода, так и монтировать файлы образов компакт-дисков ****.ISO***.
- ***Network Adapter.*** Добавляет сетевой адаптер. Можно выбрать один из трёх способов подключения этого сетевого адаптера к основной системе: ***NAT***, ***Bridged***, ***Host-only***.
- ***USB Controller.*** Добавляет в виртуальную машину ***USB***-контроллер, который позволяет подключать флэш-накопители, принтеры и другие USB-устройства.

При необходимости можно добавить в виртуальную машину дополнительное оборудование (кнопка ***Add***...), например, звуковую карту, параллельный порт, последовательный порт, привод гибких дисков, дополнительный жёсткий диск или дополнительный сетевой адаптер и пр. Также можно удалить не используемое оборудование (кнопка ***Remove***), например USB-контроллер.

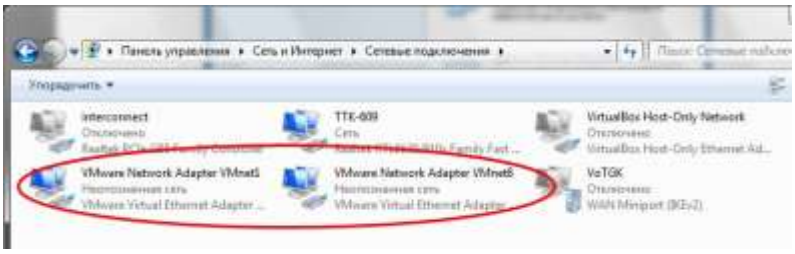
На вкладке ***Options*** представлены разные настройки, применимые, в основном, к гостевым системам семейства ОС Windows и для систем QNX, не имеющие особого значения. Выберите пункт ***General*** и задайте виртуальным машинам новые имена (***Virtual machine name***), например, ***QNX1*** и ***QNX2***.



После всех манипуляций главное окно ***VMware Player*** должно выглядеть так:

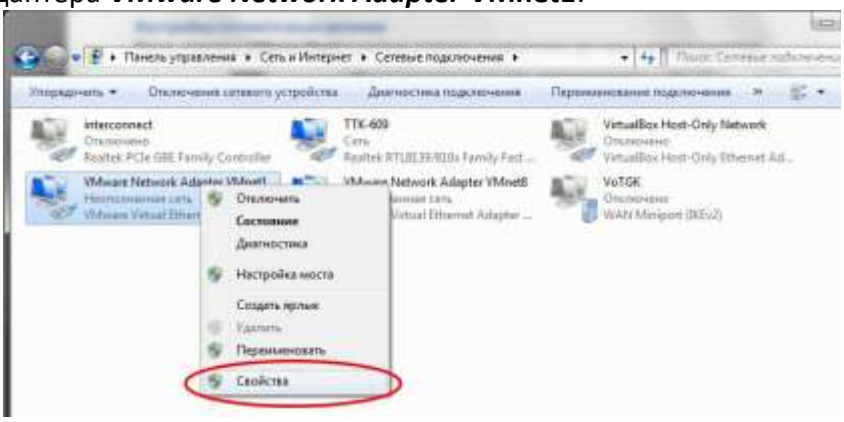
Настройка сетевого подключения

После установки ***VMware Player*** в системе появятся по умолчанию два новых сетевых адаптера:

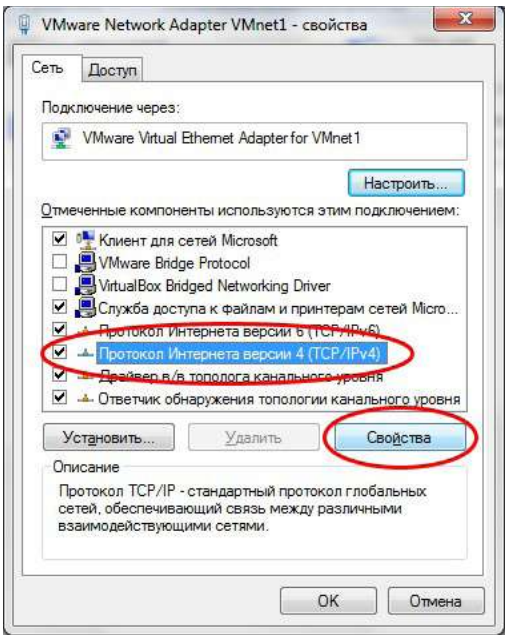


- ***VMware Network Adapter VMnet1.*** Это прямое подключение всех виртуальных адаптеров гостевых систем (***Guest OS***), работающих в режиме ***Host-Only***, к адаптеру ***VMnet1*** в основной системе (***Host OS***).
- ***VMware Network Adapter VMnet8.*** К этому адаптеру будут подключены все виртуальные адаптеры, которые работают в режиме Сетевой Трансляции Адресов (***NAT)***. При таком способе из гостевых систем можно получить доступ в глобальную сеть Интернет. (Не выбирать при этом режиме ***Host-Only****!*)
- Все виртуальные адаптеры, которые работают в режиме ***Bridged***, будут подключены непосредственно к физической сети. Фактически на одном физическом подключении будут располагаться несколько устройств со своими MAC-адресами (один адрес для физической карты, остальные - для виртуальных адаптеров). Это даёт возможность организовать реальную распределённую работу параллельных процессов.

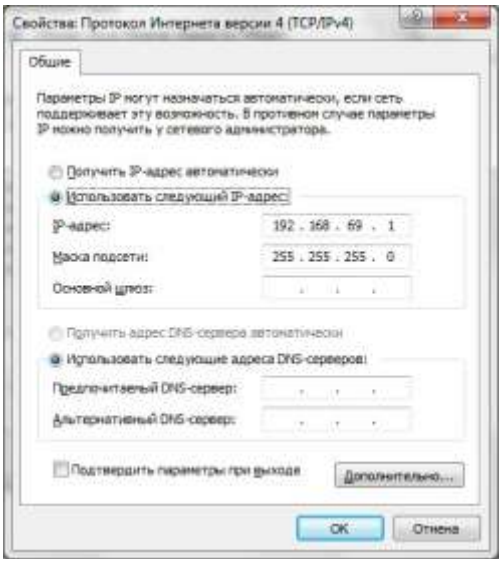
Для нашего случая вполне подойдёт вариант с адаптером ***VMware Network Adapter*** **VMnet1** (далее будет рассматриваться только он.) или ***VMware Network Adapter VMnet8***. Откройте окно свойств сетевого адаптера ***VMware Network Adapter*** **VMnet1**:



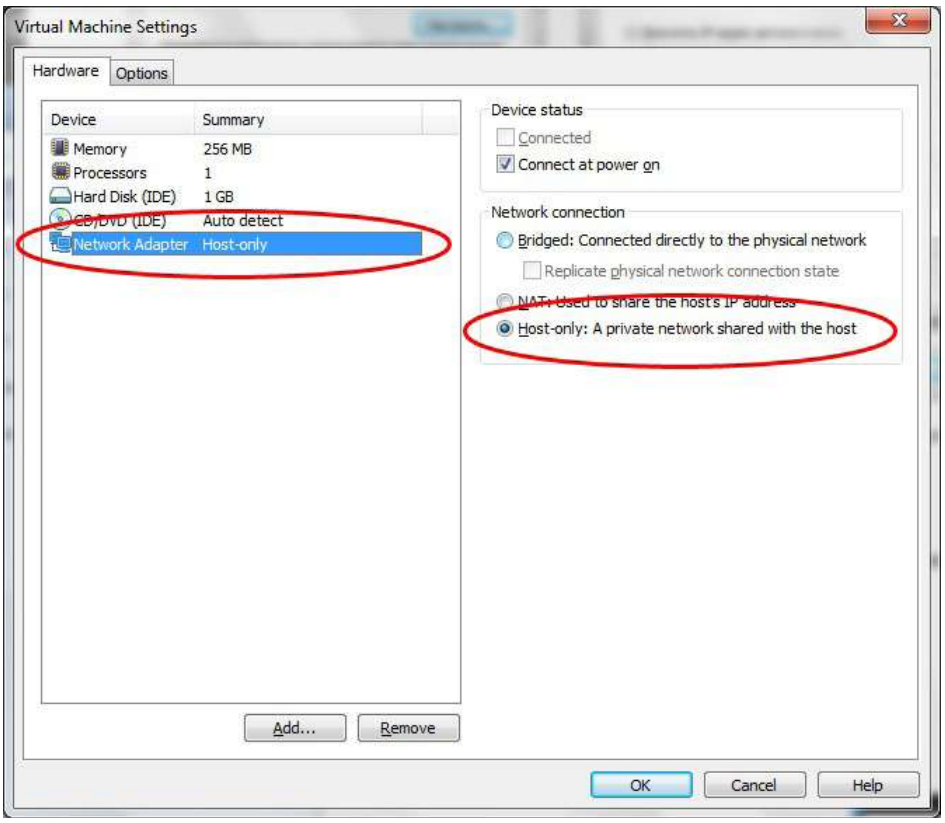
В открывшемся окне свойств



дважды кликните по пункту **Протокол Интернета версии 4 (TCP/IPv4)**. Откроется окно для установки свойств протокола:



В окне свойств следует задать IP-адрес, например, ***192.168.69.1***, и маску подсети - ***255.255.255.0***. Остальные поля свойств протокола следует оставить пустыми. Далее следует настроить сетевой адаптер виртуальной машины. Откройте окно свойств виртуальной **QNX**-машины, например, **QNX1**:

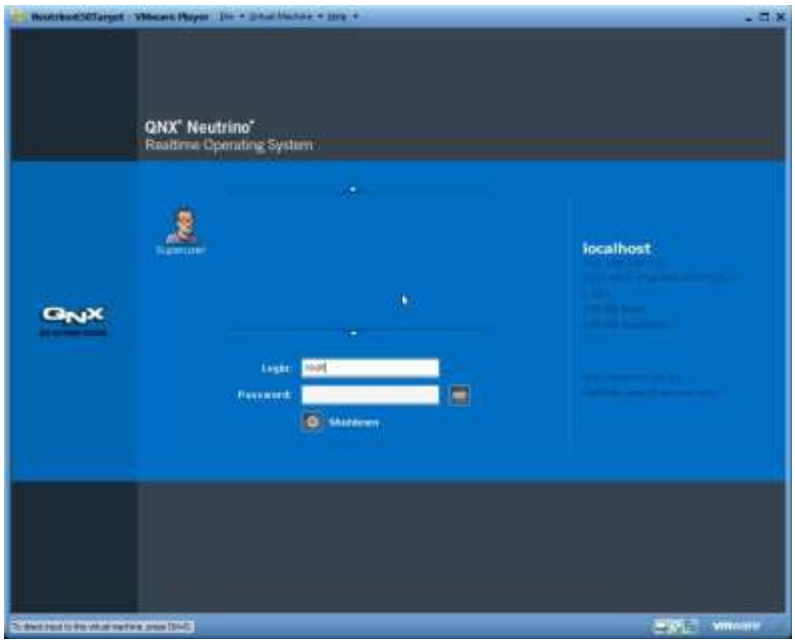


.

В открывшемся окне следует установить режим работы сетевого адаптера - ***Host-Only***. То же, при необходимости, нужно проделать и для других виртуальных машин (если они установлены). В результате, все созданные виртуальные ***QNX***-машины и основная система будут работать в одной сети ***VMnet1***.

Настройка целевой системы

Выберите одну из виртуальных ***QNX***-машин, например ***QNX1***, и запустите её, нажав ***Play virtual machine***. После того как система загрузится, появится экран приветствия графической оболочки ***QNX Photon microGUI***. В образе виртуальной ***QNX***-машины ***QNX_Eval_RT-201007091524.zip*** графическая оболочка запускается автоматически при старте системы:

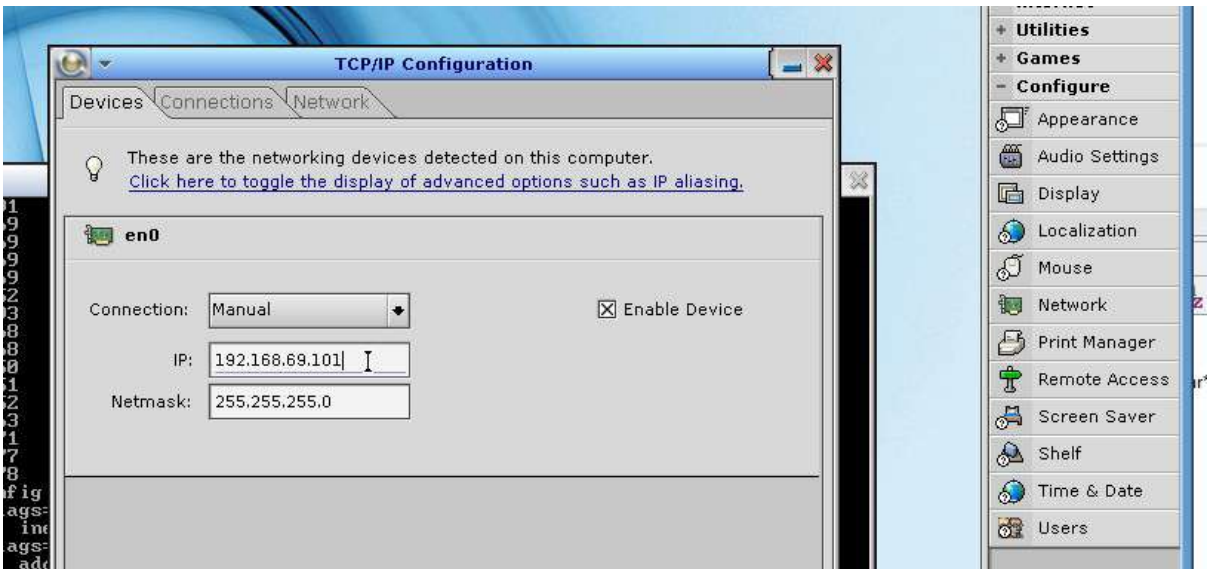


В открывшемся окне справа представлена информация о ***QNX***-машине:

- текущий IP-адрес,
- тип процессора,
- объём оперативной памяти,
- версия операционной системы.

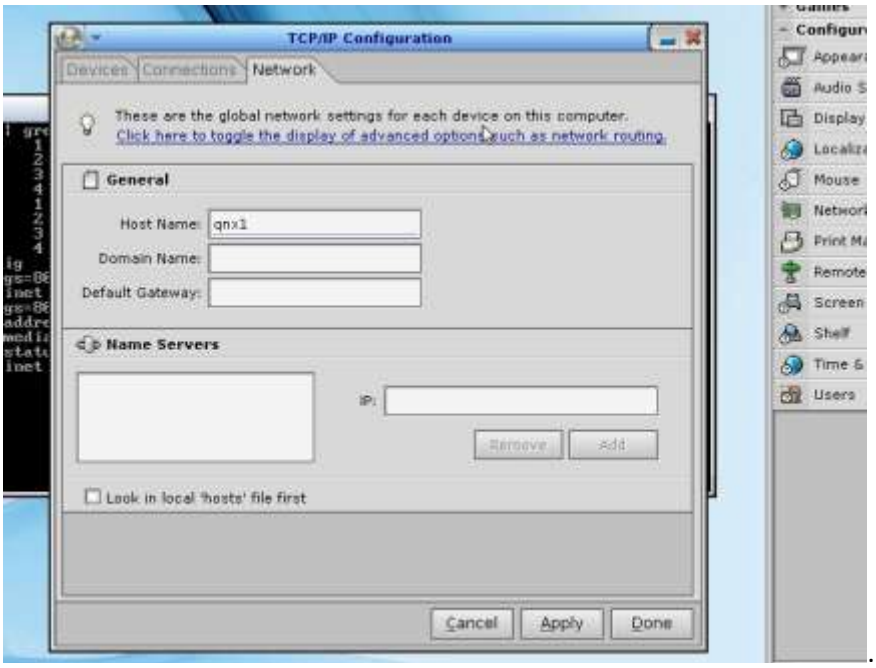
Для входа во вновь установленную систему необходимо использовать логин ***root*** и пустой пароль. В результате откроется окно графической оболочки ***Photon*** ОС ***QNX*** (аналог Рабочего стола). Для перехода из окна управления виртуальной ***QNX***-машиной в окно инструментальной машины необходимо выполнить команду ***Ctrl+Alt***. Для перехода обратно из окна управления инструментальной машиной в окно виртуальной ***QNX***-машины можно выполнить команду ***Ctrl+G*** или просто кликнуть мышью в области окна ОС ***QNX***.

Далее необходимо выполнить процедуру установления канала связи виртуальной ***QNX***-машины с инструментальной машиной, т.е. выполнить настройку сетевого адаптера в ОС ***QNX***. Для этого в открывшемся окне на панели справа следует выбрать пункты меню - ***Configure → Network***. Откроется окно настроек сетевого адаптера:



На вкладке ***Devices*** отображаются все присутствующие в системе сетевые интерфейсы (по количеству адаптеров, добавленных в настройках виртуальной машины). По умолчанию сетевой адрес интерфейсу присваивается ***DHCP*** сервером. В нашем случае необходимо задать нужный сетевой адрес вручную. Для этого необходимо в пункте ***Connection:*** явно выбрать ручной режим - ***Manual***. Так как все компьютеры (инструментальная машина и созданные ***QNX***-машины) должны

находиться в одной подсети (в нашем случае это ***192.168.69.0/24***), то в целевой системе ***QNX1*** следует в пункте **IP:** явно задать некий IP-адрес, например ***192.168.69.101***, и в пункте ***Netmask:*** маску подсети ***255.255.255.0***. После этого следует перейти на вкладку ***Network***. Откроется окно вида:



.

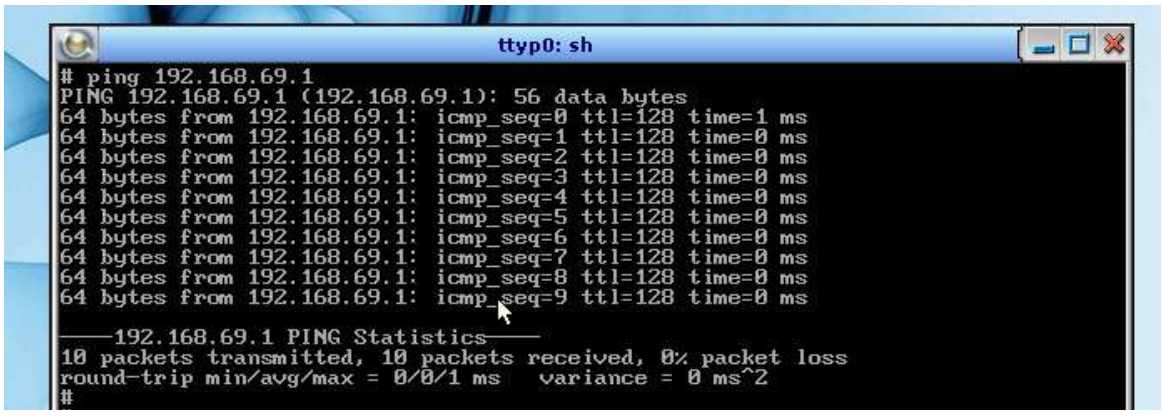
В этом окне следует ***QNX***-машине задать имя (***Host Name***), которое будет использоваться сетевым протоколом ***Qnet*** ОС ***QNX***, например, можно воспользоваться именем виртуальной ***QNX***-машины - ***qnx1***, остальные поля (***Domain Name***, ***Default Gateway***, ***Name Servers***, ***IP***) следует сделать или оставить пустыми.

То же следует проделать и с остальными целевыми машинами, если они необходимы (например, для целевой машины ***QNX2*** можно, например, указать адрес ***192.168.69.102***, и имя машины ***qnx2***.

Теперь необходимо проверить наличие связи между целевой и инструментальной машинами, используя терминальную команду **`ping`** командного интерпретатора ***shell*** для проверки связи. Для этого в ОС ***QNX*** следует открыть терминальное окно, выполнив команды меню правой панели: ***Utilities → Terminal***, или кликнуть правой кнопкой мыши в свободной области стартового окна и в контекстном меню выбрать пункт меню ***Terminal***, откроется терминальное окно. В терминальном окне введите команду:

```
# ping 192.168.69.1
```

Если всё настроено правильно, в терминальном окне будет получен отклик от инструментальной системы ОС Windows о получении пакетов передачи данных от целевой машины:



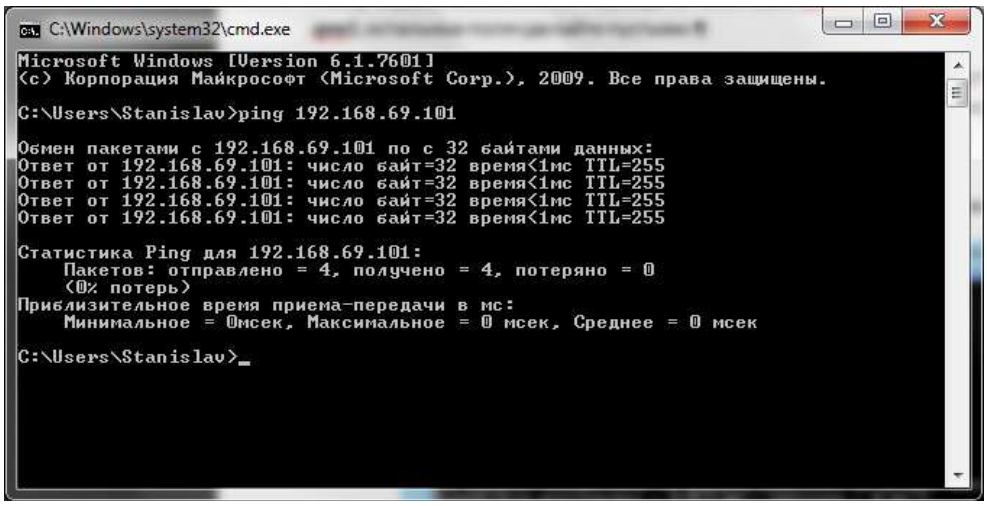
Чтобы остановить выполнение команды **`ping`** следует нажать ***Ctrl+C***.

Теперь необходимо проверить наличие связи из инструментальной системы. Для этого следует

вернуться в ОС Windows и открыть терминальное окно командного интерпретатора ОС Windows (команда ***Пуск/Выполнить*** или нажмите ***Win+R***), в открывшемся окне запуска программы введите ***cmd***. Откроется терминальное окно ОС Windows, в котором также следует выполнить команду **`ping`** с IP-адресом целевой машины:

```
# ping 192.168.69.101
```

Должен быть получен отклик от целевой машины о прохождении пакетов данных:



ВНИМАНИЕ! Если одна из систем не откликается, то проверьте настройки **сетевого экрана** (брандмауэр) ОС Windows. Рекомендуется временно отключить его при работе с целевыми машинами.

Обеспечение взаимодействия среды разработки с целевой системой

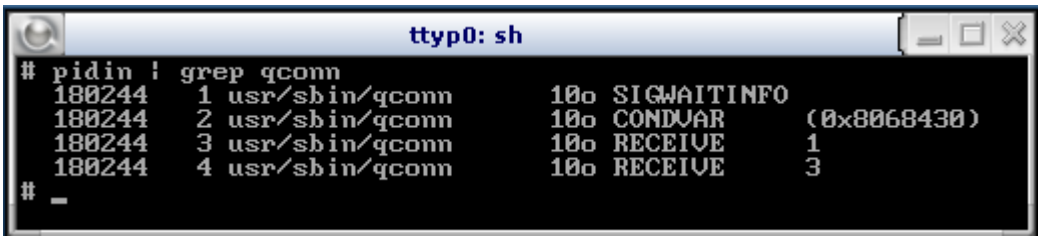
Для обеспечения взаимодействия среды разработки и целевой системы в ОС QNX должна быть запущена специальная служба ***qconn***. Для её запуска следует в терминальном окне ОС QNX целевой системы выполнить команду:

```
# qconn
```

Заметим, однако, что при работе с готовым образом целевой системы под ОС QNX для виртуальной машины этого делать не нужно – команда запуска службы ***qconn*** уже прописана в стартовом скрипте ***/etc/rc.d/rc.local*** и выполняется автоматически при запуске системы. Тем не менее, следует убедиться в наличии службы ***qconn*** с помощью команды:

```
# pidin | grep qconn
```

В результате выполнения этой команды должен быть получен отклик:



Если данный отклик не получен, следует вручную выполнить команду:

```
# /usr/sbin/qconn &
```

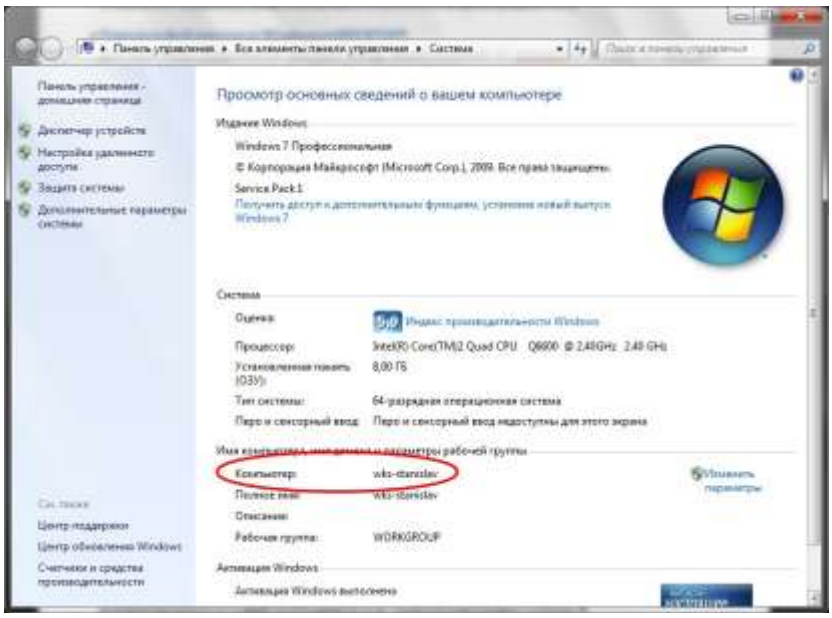
Создание разделяемой папки с общим доступом

Так как разработка программ будет производиться в инструментальной системе (ОС Windows) в ***QNX Momentics IDE***, а исполнение – в целевой (ОС QNX), то может потребоваться обмениваться файлами между основной и целевой системами. Для этого в ОС Windows следует создать папку и открыть к ней общий сетевой доступ. Затем для доступа к этой папке из целевой системы её нужно

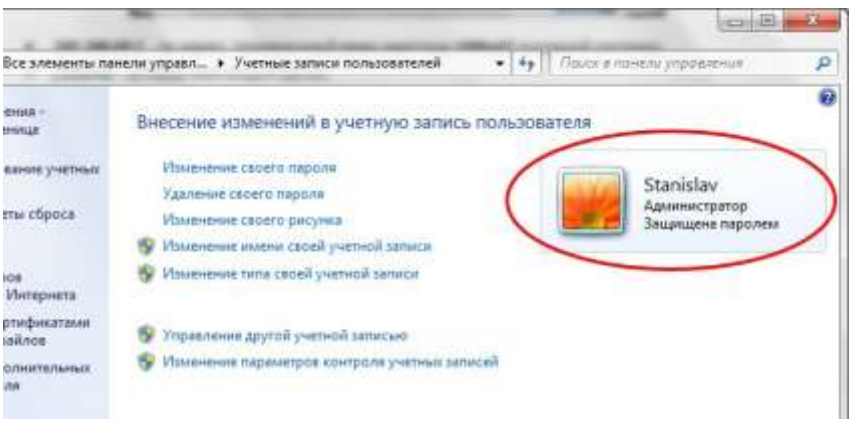
монтировать к файловой системе ОС QNX. В целевой системе (ОС QNX) следует открыть терминальное окно и выполнить команду монтирования разделяемой папки. Формат этой команды монтирования следующий:

`#fs-cifs //pcname:IP-adapter:/shared /home/host login password`

- ***pcname*** - имя компьютера в основной системе, которое можно узнать, вызвав окно свойств объекта **Мой компьютер** (или нажать ***Win+Pause***):



- **IP-adapter -** IP-адрес, присвоенный адаптеру VMnet1 в основной системе;
- **/shared -** имя разделяемой сетевой папки, созданной в основной системе;
- **/houme/host** – абсолютное (полное) имя каталога (папки) в файловой системе QNX, под которым разделяемая сетевая папка Windows будет монтирована к файловой системе ОС QNX. В результате в файловой системе QNX в системном каталоге /**houme** будет создан каталог с именем /host;
- **login** - логин пользователя в основной системе, который имеет права на чтение и запись в общую сетевую папку. Чтобы узнать имя пользователя, зайдите в **Панель управления → Учётные записи пользователей**:



- **password** - пароль. Если у пользователя пароль отсутствует (пуст) – следует указать ***none***.

Положим, например, что:

- ✓ имя инструментального компьютера – **`wks1`**;
- ✓ IP-адрес, присвоенный адаптеру VMnet1 основной системы - **`192.168.69.1`**;
- ✓ имя разделяемой сетевой папки в основной системе – **`sharQNX`**;

- ✓ имя пользователя в учетной записи основной системы (login) – **Admin**;
- ✓ пароль пользователя в учетной записи основной системы пустой (отсутствует) – **none.**

В итоге, команда монтирования разделяемой папки **sharQNX** в файловом пространстве целевой машины в виде каталога с именем **/host/home** будет иметь вид:

```
# fs-cifs //wks1:192.168.69.1:/sharQNX /home/host Admin none
```

После выполнения этой команды в целевой системе в каталоге **/houme** появится подкаталог с именем **/host,** который будет соответствовать разделяемой сетевой папке основной системы - **sharQNX**. Теперь основная и целевая системы имеют общий доступ к содержимому разделяемой сетевой папки (**sharQNX ↔ /home/host**).

ЗАМЕЧАНИЯ!

1. Если выдается сообщение об ошибке монтирования разделяемой папки, следует попробовать отключить брандмауэр и повторить попытку монтирования.

2. Если при монтировании в целевой системе разделяемой папки выдается сообщение об отсутствии доступа к папке, в этом случае следует проверить, соответствует ли имя пользователя в учетной записи, имени системной папки пользователя на диске C:. Бывают случаи, когда они не совпадают, что является причиной в отказе доступа.

*3. Возможно, что команда монтирования разделяемой папки в целевой системе с указанием пустого пароля **none** не проходит. Тогда необходимо в основной системе установить для пользователя явный пароль и использовать его в команде монтирования разделяемой папки. Например, если пароль был бы цифра 1, то рассмотренная выше команда монтирования имела бы вид:*

```
# fs-cifs //wks1:192.168.69.1:/sharQNX /home/host Admin 1
```

4. Если команда монтирования завершается с ошибкой после выполнения всех перечисленных выше действий, попробуйте закрыть целевую систему, выключить и включить компьютер заново.

Использование мобильных носителей

С целью переноса программных проектов с одного компьютера на другой (например, с домашнего компьютера в компьютерный класс для демонстрации результатов преподавателю) можно использовать мобильные носители. В общем случае виртуальная QNX-машина (целевая система) должна автоматически монтировать в каталоге **/fs** подключаемые к инструментальной системе мобильные носители: лазерные диски, флэш-носители и тп. Если монтирование флэш-носителя происходит успешно, то при загрузке QNX Momentics IDE 4.7 можно выбирать содержащуюся на нём папку с программным проектом в качестве рабочего каталога. Важно, что она же будет доступна и в виртуальной QNX-машине.

Однако на практике автоматическое монтирование флэш-памяти в целевой машине не всегда завершается успешно. Это обычно связано с системными аппаратно-программными особенностями основной машины. В этом случае на флэш-носителе можно создать разделяемую папку и монтировать её в целевой системе так же, как это уже было описано выше. В итоге, в целевой машине можно будет иметь несколько разделяемых папок.

Настройка сети Qnet

Для работы в сети QNX располагает собственным сетевым протоколом ***Qnet***. Все узлы, находящиеся в сети под управлением ОС QNX, идентифицируются по уникальному в пределах сети имени. Это имя (***Host Name***) которое было задано при настройке ***QNX***-машины. Других настроек для ***Qnet*** не требуется.

Для монтирования сети ***Qnet*** в целевой ***QNX***-машине необходимо в терминальном окне ОС QNX

выполнить команду:

```
# mount –T io-pkt /lib/dll/lsm-qnet.so
```

Если монтирование прошло успешно, в файловой системе ***QNX***-машины появляется каталог ***/net***, который содержит в себе каталоги с именами, соответствующими примонтированным к сети узлов (***QNX***-машин). Теперь через ***Qnet*** каждый узел сети QNX может получить доступ ко всем файлам, устройствам и процессам всех узлов, примонтированных к сети QNX.

Сохранение настроек

В файловой системе ОС QNX целевой машины имеется стартовый командный скрипт ***/etc/rc.d/rc.local***, который автоматически выполняется при каждом запуске системы. Обычно командный скрипт ***/etc/rc.d/rc.local*** содержит следующие команды:

```
/usr/sbin/qconn &
/usr/sbin/inetd &
```

Чтобы все ранее рассмотренные команды настройки целевой системы автоматически применялись при каждом запуске системы целесообразно добавить их в стартовый скрипт **/etc/rc.d/rc.local** ОС QNX. Для этого следует открыть и отредактировать файл стартового скрипта **/etc/rc.d/rc.local** в текстовом редакторе - **Utilities → Text Editor**:

В конец файла нужно добавить команды, например:

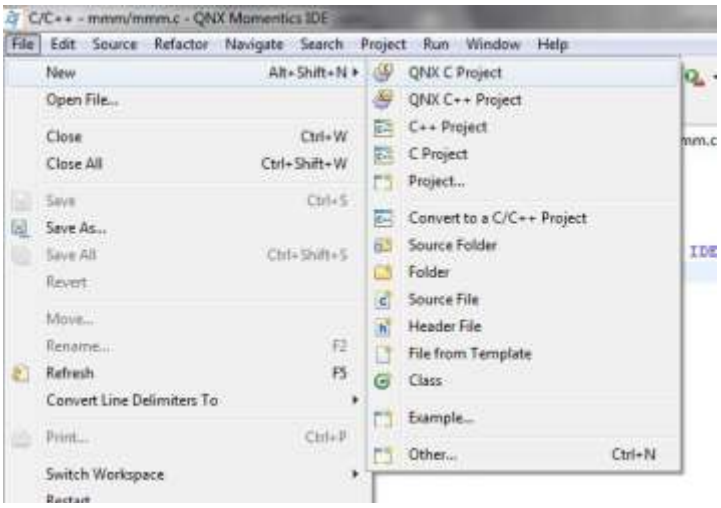
```
mount –T io-pkt /lib/dll/lsm-qnet.so
fs-cifs //wks1:192.168.69.1:/sharQNX /home/host Admin none
```

Сохраните и закройте файл скрипта. Теперь целевая система готова для запуска и взаимодействия с основной системой и со средой разработки ***QNX Momentics IDE***.

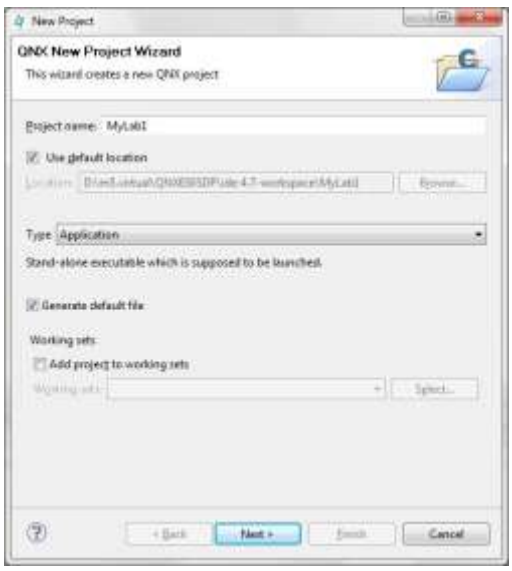
ВНИМАНИЕ! *Следует соблюсти именно указанный порядок команд, так как монтирование сети* ***Qnet*** *сбрасывает все ранее открытые сетевые подключения, поэтому оно должно выполняться раньше, чем монтирование разделяемой сетевой папки.*

Создание программного проекта

Чтобы создать новый проект на языке С следует выбрать пункты меню: ***File→New→QNX C Project***:

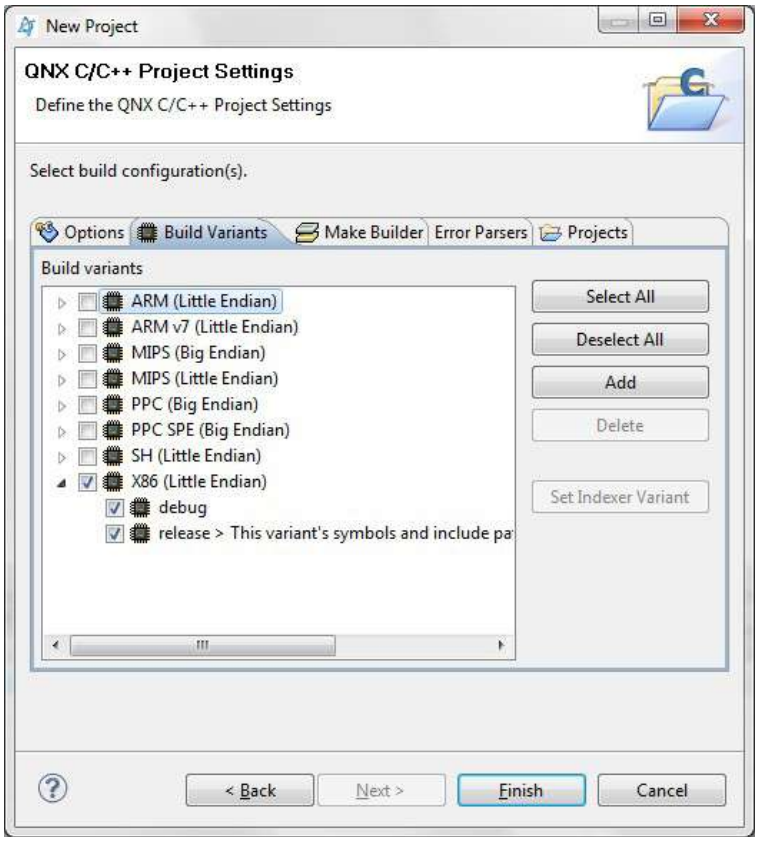


В открывшемся окне **New Project** в поле ***Project Name*** следует задать имя проекта (например, MyLab1):



Другие настройки можно оставить без изменений (по умолчанию) и нажать ***Next***.

Откроется окно, где на вкладке ***Build Variants*** необходимо указать архитектуры целевых процессоров, для которых предполагается собирать проект. В данном случае для запуска программы в целевой системе (виртуальная ***QNX***-машина с архитектурой ***x86***) следует указать архитектуру ***x86 (Little Endian)***:



Автоматически отмечаются варианты построения программы в двух вариантах:

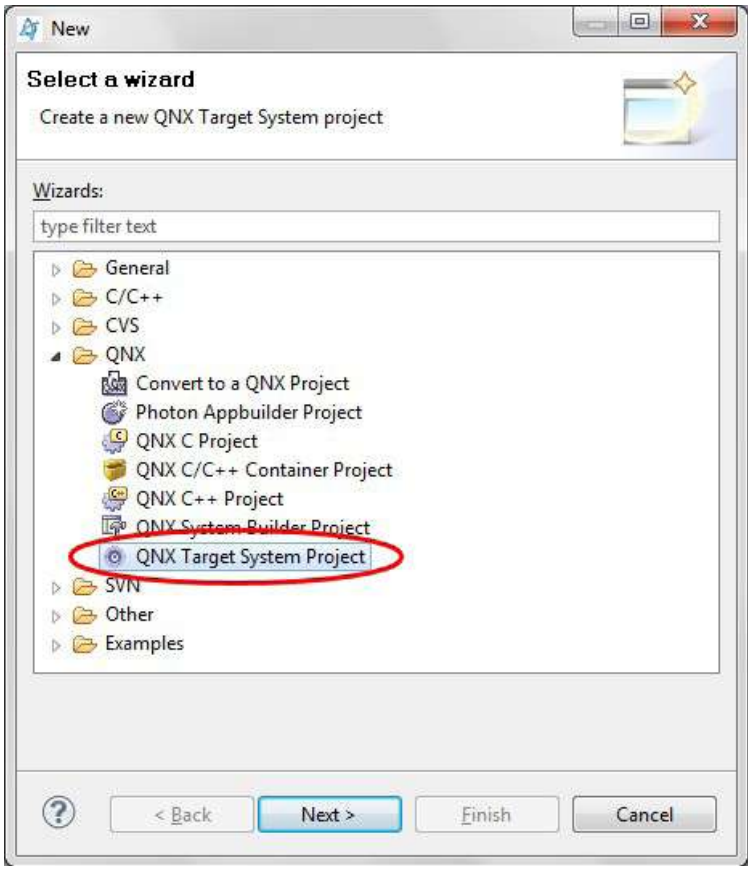
- с включёнными средствами для выполнения в режиме отладки (**debug**);
- в окончательном варианте (**release**).

Следует принять оба варианта построения программы. Это впоследствии даст возможность выбора конкретного варианта построения программы.

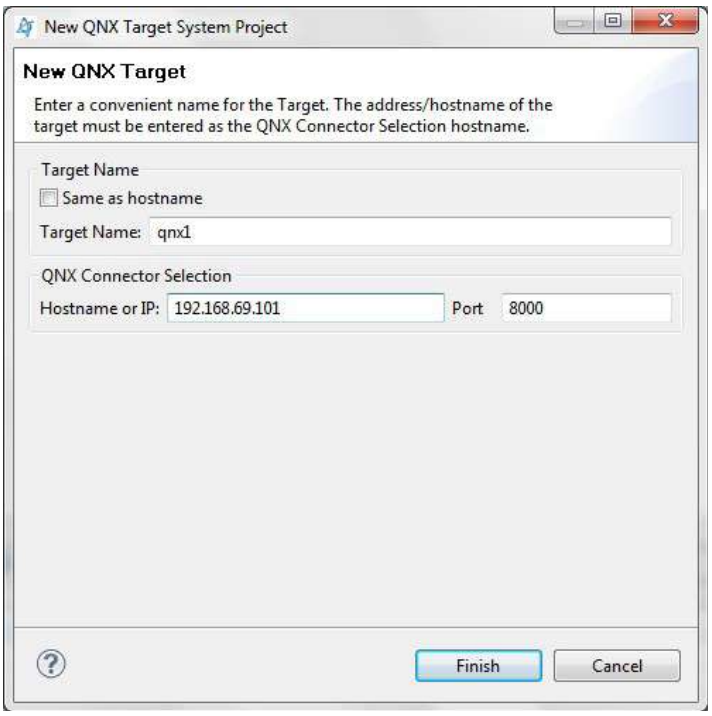
Подключение среды разработки к целевой машине

Для запуска программ из среды разработки в заданной целевой машине необходимо выполнить её подключение к проекту. Подключение выполняется следующим образом. Можно вызвать контекстное меню (кликнув правой кнопкой мыши на пустом месте панели обозревателя проекта (**Project Explorer**), откроется окно контекстного меню:

В этом окне следует выбрать пункты ***New→Other...***, откроется окно **New** для создания проекта новой целевой ***QNX***-системы:



В этом окне следует выбрать пункты ***QNX→QNX Target System Project***, Откроется окно для задания IP-адреса целевой системы и имени целевой системы, под которым она будет зарегистрирована в инструментальной системе ***QNX Momentics IDE***:



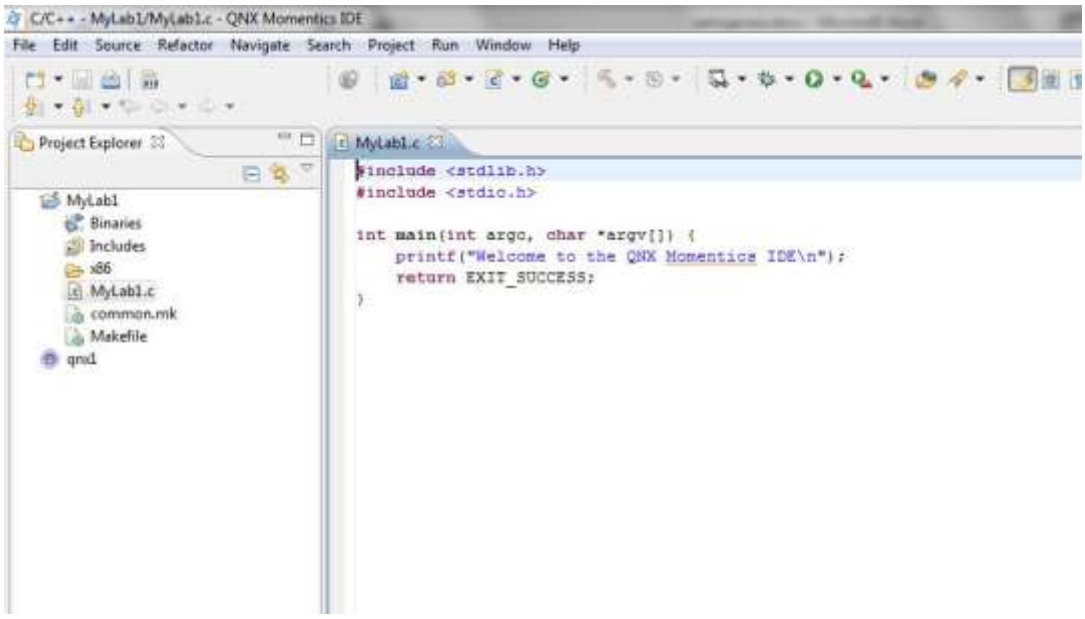
Введите имя целевой машины, например, ***qnx1*** и укажите IP-адрес сетевого адаптера, зарегистрированный в целевой системе. В нашем случае для виртуальной ***QNX***-машины ***qnx1*** IP-адрес будет ***192.168.69.101***, порт следует оставить по умолчанию - ***8000***. Подключение проекта к целевой системе завершается нажатием ***Finish***. Если подключение успешно создано, то на панели **Project Explorer** в дереве проекта появится имя целевой машины - ***qnx1***.

После подключения инструментальной системы к целевой системе она имеет доступ ко всем ресурсам ***QNX***-машины. Через среду разработки ***QNX Momentics IDE*** можно запускать и отлаживать процессы в целевой системе, отслеживать и управлять всеми запущенными в целевой

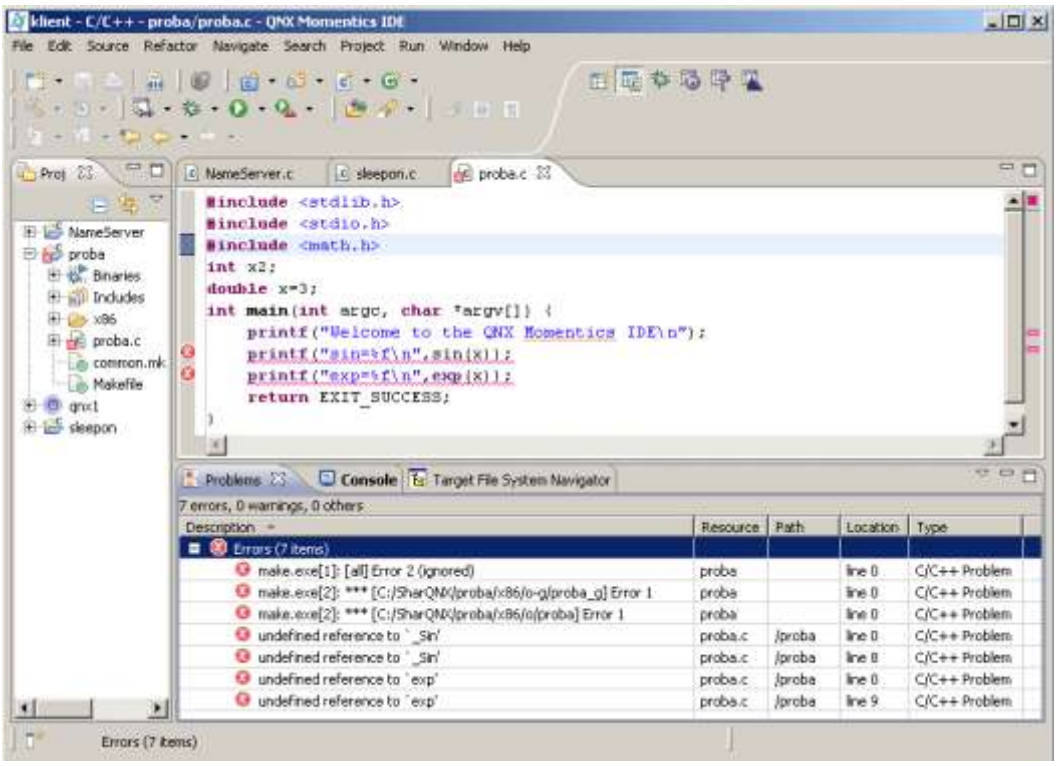
системе процессами, следить за оперативной памятью и вести историю её выделения и освобождения, и многое другое.

Компиляция и запуск проекта

Предположим, что создан проект с именем **MyLab1**, который содержит пока единственный файл ***MyLab1.c***, и главное окно выглядит так:

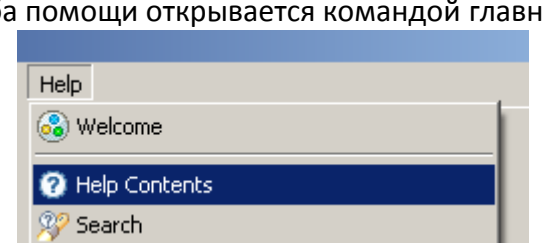


Для того чтобы скомпилировать (построить) проект, можно щёлкнуть правой кнопкой по проекту **MyLab1** в дереве **Project Explorer** и в контекстном меню выбрать пункт **Build Project**. Если компиляция прошла успешно, то в дереве проекта появляется пункт Binaries с исполняемым программным модулем проекта. Если компиляция завершилась с ошибками, то сообщения о них выдаются в окно **Problems:**



Обратите внимание, что в обозревателе проект помечается крестом на красном фоне. Если ошибки являются синтаксическими, то они выявляются на этапе компиляции исходного текста программы и устраняются при исправлении текста программы. Сложнее дело обстоит с ошибками, возникшими на этапе компоновки исполняемого модуля. Эти ошибки часто бывают вызваны тем, что при компоновке система не смогла найти библиотечные файлы, необходимые компоновщику

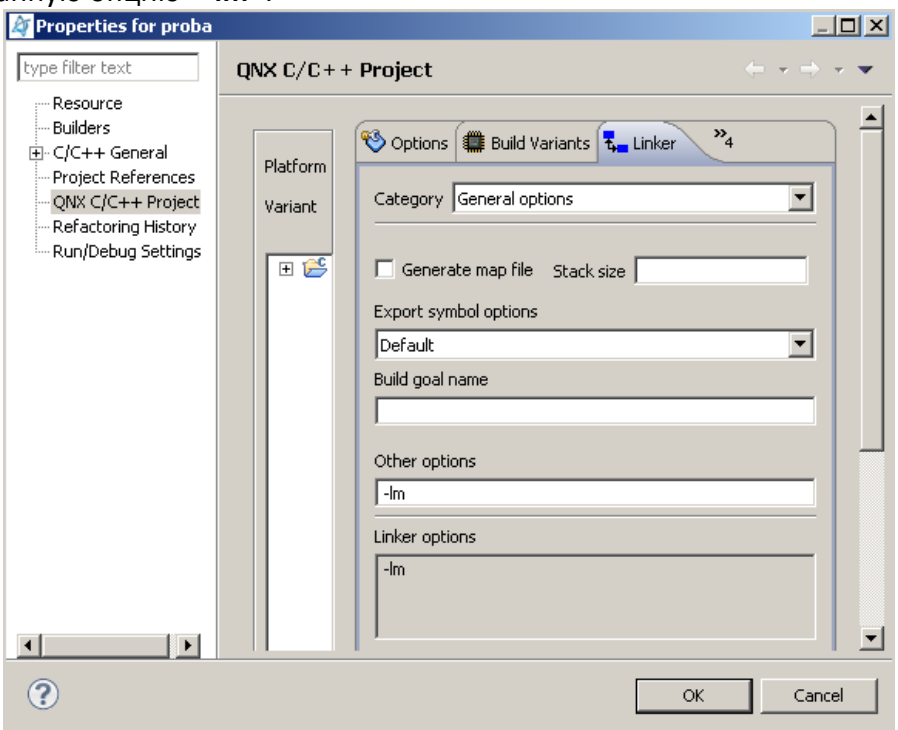
для подключения требуемых программе объектных модулей стандартных функций. Например, если в программе используются вызовы математических функций, компоновщику потребуется указать, какую библиотеку использовать. Информацию о нужной библиотеке можно получить, используя службу помощи, предоставляющей различные сведения о системе и, в частности, о стандартных функциях. Служба помощи открывается командой главного меню ***Help/Help Contents***:



Откроется окно со справочными данными относительно ОС ***QNX*** и, в частности, библиотечных функций. Следует найти описания функций, относительно которых выданы сообщения об ошибках. В описаниях функций присутствует раздел со сведениями о библиотечных файлах и опциях компилятора, которые следует указывать в свойствах проекта. Например, в описании стандартных математических функций этот раздел выглядит так:

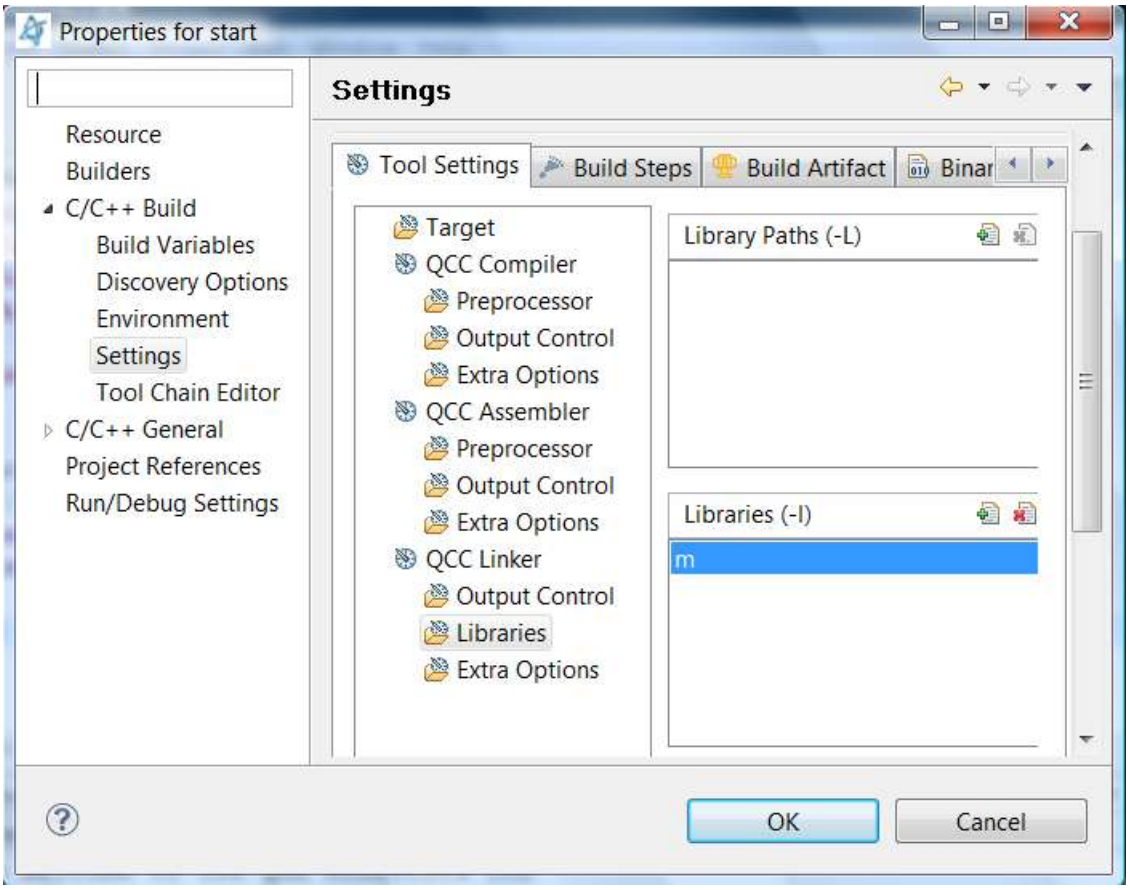
Library:
`libm`
`Use the -l m option to qcc to link against this library.`

Это означает, что проекту требуется библиотека **`libm`**, и компоновщику следует указать опцию <**-lm**>. Для этого в свойствах проекта (команда главного меню ***Project/Properties***) на вкладке ***Linker*** надо задать указанную опцию <**-lm**>:

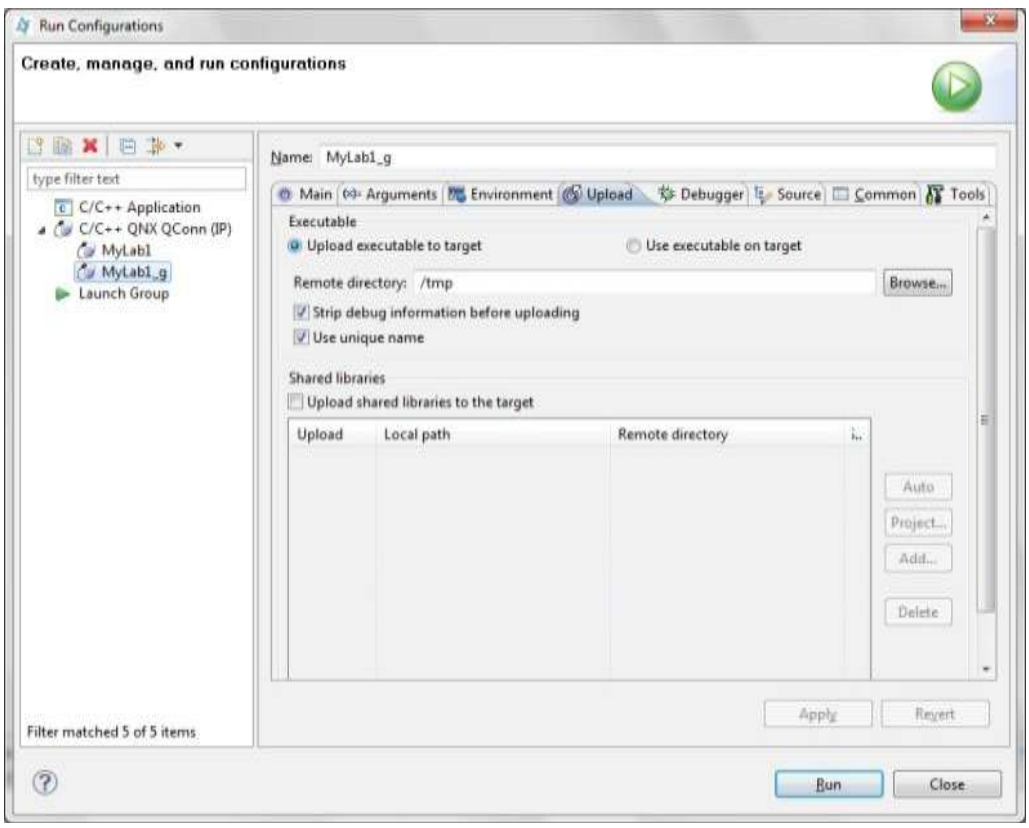


Если необходимы и другие библиотеки, то здесь же через пробел следует указать соответствующие им опции.

Заметим, что окно управления свойствами проекта может иметь и другой вид (например, под ОС Vista):



В этом случае необходимые библиотеки следует задать на вкладке **Tool Settings**.
Если после всех выше описанных действий проблема сохраняется, попробуйте, отключить в конфигурации запуска требование «использовать уникальное имя» - Use unique name (снимите галочку):



Если эта опция включена, то исполняемый файл после загрузки на целевую систему будет переименован в конкатенацию исходного названия проекта, имени пользователя ОС Windows, под которым он работает в ***QNX Momentics IDE***, и длинного случайного числа. Если имя пользователя ОС Windows содержит кириллические буквы, то возникнут проблемы при запуске через ***qconn***. С

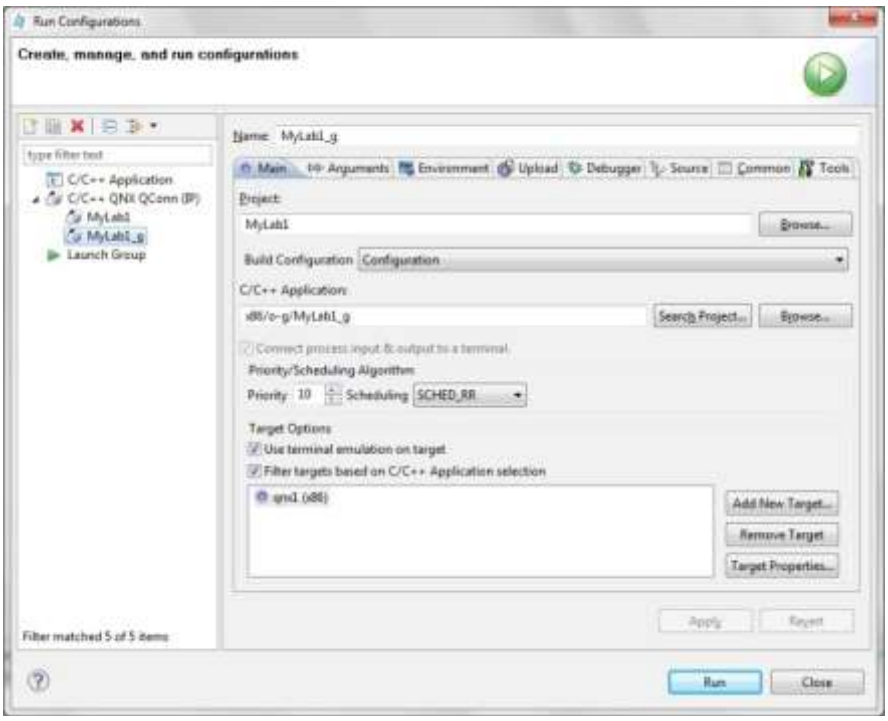
отключенной опцией ***Use unique name*** имя исполняемого файла будет совпадать с названием проекта.

Конфигурации запуска

Конфигурация запуска - это сохранённый именованный набор настроек, определяющих, где и как будет запускаться указанный проект. В эти настройки входят:

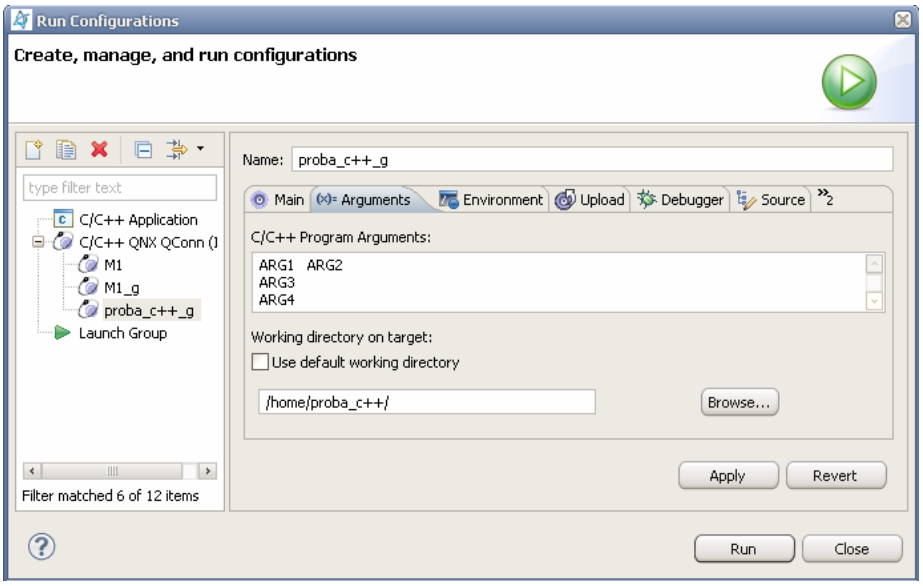
- адрес целевой системы,
- способ загрузки исполняемого файла в целевую систему,
- приоритет запускаемого процесса,
- дисциплина диспетчеризации и т.д.

Рассмотрим подробно действия по созданию и использованию *конфигурации запусков*. Чтобы войти в режим управления конфигурацией запуска проекта, можно в дереве обозревателя проектов ***Project Explorer*** щёлкнуть правой кнопкой на проекте, и с помощью контекстного меню выполнить команду ***Run As → Run Configurations***. Появится окно **Run Configuration**:



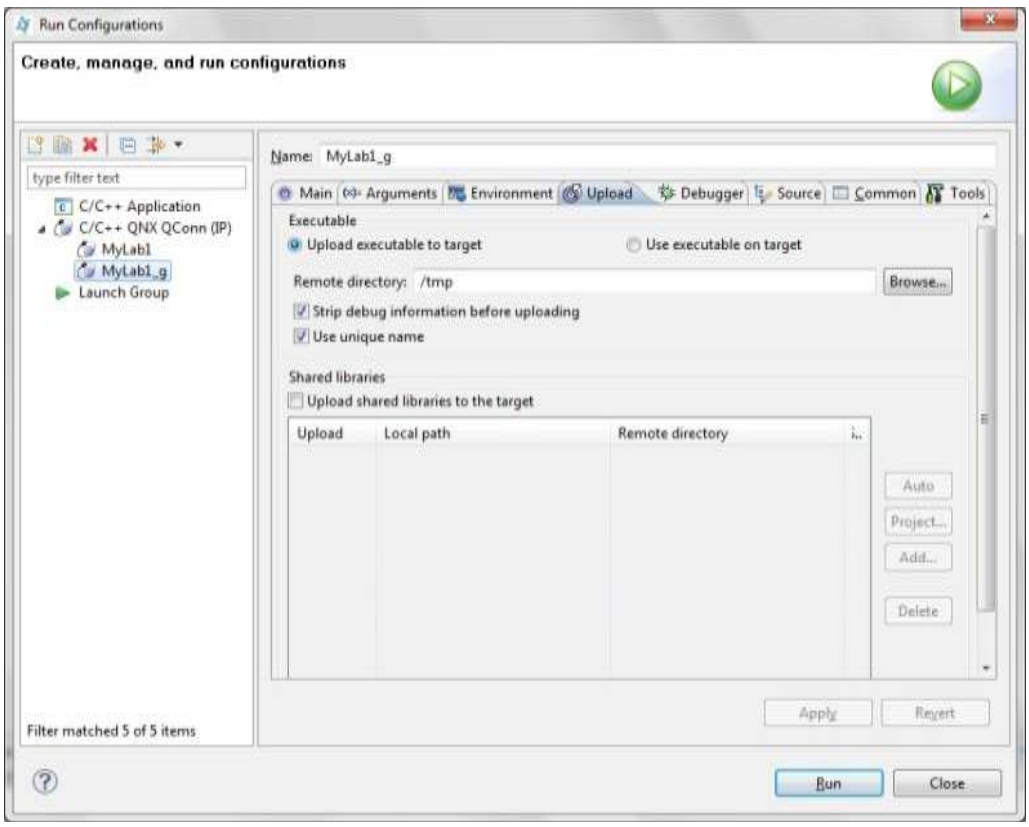
В этом окне можно создавать новые конфигурации запуска проектов и изменять существующие. Откройте вкладку ***Main***: В поле **Name** для вновь создаваемой конфигурации указывается имя конфигурации запуска. В поле **C/C++ Application** задаётся путь к исполняемому бинарному файлу. Здесь же в поле **Priority** можно указать приоритет процесса, а в поле **Scheduling** - дисциплину диспетчеризации.

На вкладке ***Arguments*** в поле ***C/C++ Program Arguments*** можно указать аргументы, с которыми будет запускаться приложение. Кроме того, можно явно установить для проекта рабочий каталог на целевой машине - ***Working directory on target*** :



Для этого необходимо снять отметку (галочку), а затем использовать кнопку ***Browse…*** для входа в файловую систему целевой ***QNX***-машины для выбора нужного каталога (целевая машина должна быть подключена).

По умолчанию создаваемый исполняемый модуль проекта в целевой машине загружаться в каталог ***/tmp***. При этом можно явно указать каталог загрузки исполняемого модуля. Для этого следует открыть вкладку **Upload** и в поле **Remote directory**, указать имя каталога:

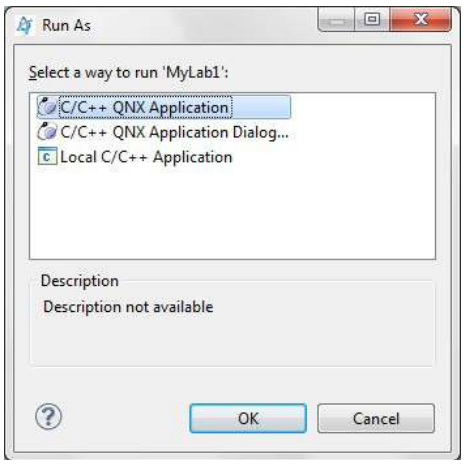


ВНИМАНИЕ! На вкладке **Upload** отключите опцию **Use unique name** (снимите галочку). Если эта опция включена, то исполняемый файл после загрузки на целевую систему будет переименован в конкатенацию исходного названия проекта, имени пользователя ОС Windows, под которым он работает в ***QNX Momentics IDE***, и длинного случайного числа. Если имя пользователя ОС Windows содержит в кириллические буквы, то возникнут проблемы при запуске через ***qconn***. С отключенной опцией ***Use unique name*** имя исполняемого файла будет совпадать с названием проекта.

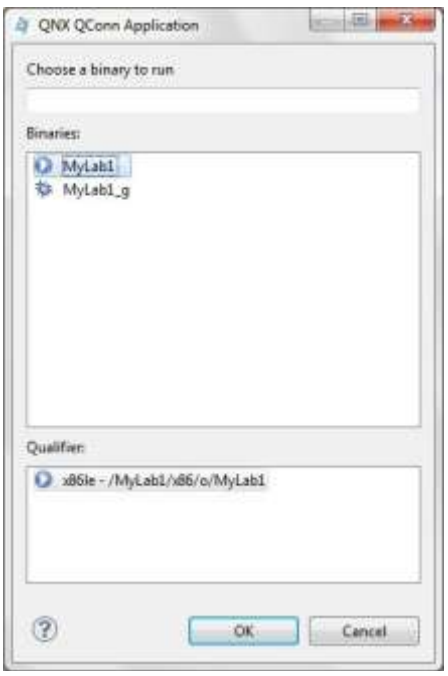
Запуск проекта

Чтобы запустить собранный проект (исполняемую программу), можно нажать кнопку ***Run*** на панели инструментов:

Заметим, однако, что для запуска проекта должна быть указана конфигурация запуска. Если пока не создано ни одной конфигурации запуска, то при выполнении запуска предварительно возникает диалог для выбора способа запуска проекта:

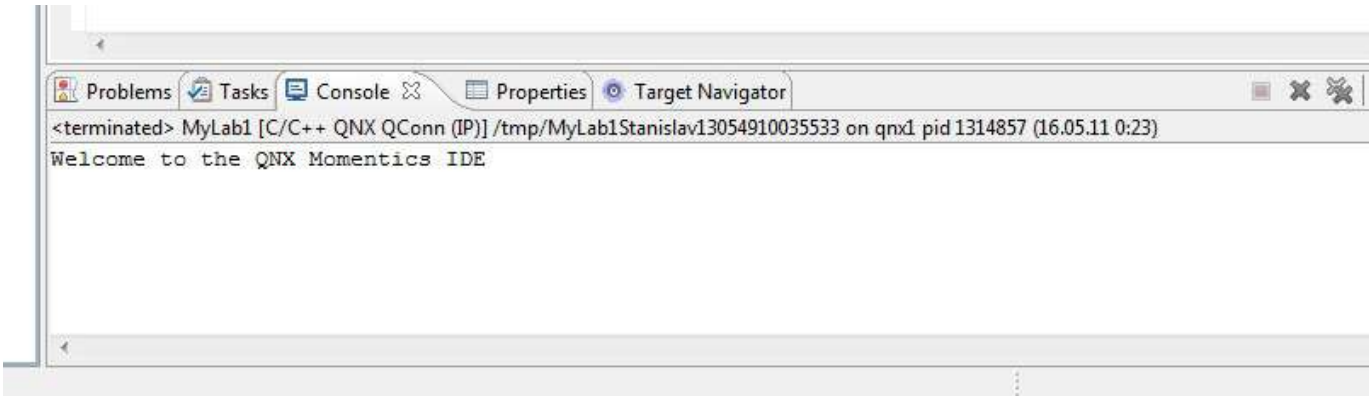


Следует выбрать пункт **C/C++ QNX Application** и нажать **OK**. Далее возникает диалог с выбором варианта формирования исполняемых файлов. Исполняемые файлы проекта могут быть собраны в двух вариантах: релизная сборка без возможности получения отладочной информации (имя исполняемого файла будет ***MyLab1***) и сборка с возможностью получения отладочной информацией (имя исполняемого файла будет ***MyLab1_g***). Для простого запуска выберете релизную сборку без возможности получения отладочной информации - ***MyLab1***:



Компиляция под нужную архитектуру процессора происходит на инструментальном компьютере разработчика в ***QNX Momentics IDE***. Затем полученные исполняемые файлы загружаются по сети с помощью сервиса ***qconn*** на целевую систему и запускаются.

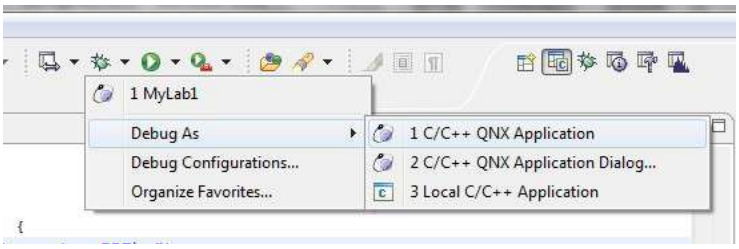
Потоки ввода и вывода перенаправляются сервисом ***qconn*** в среду разработки ***QNX Momentics IDE*** в окно **Console** виртуального терминала, расположенное обычно в нижней части главного окна ***QNX Momentics IDE***:



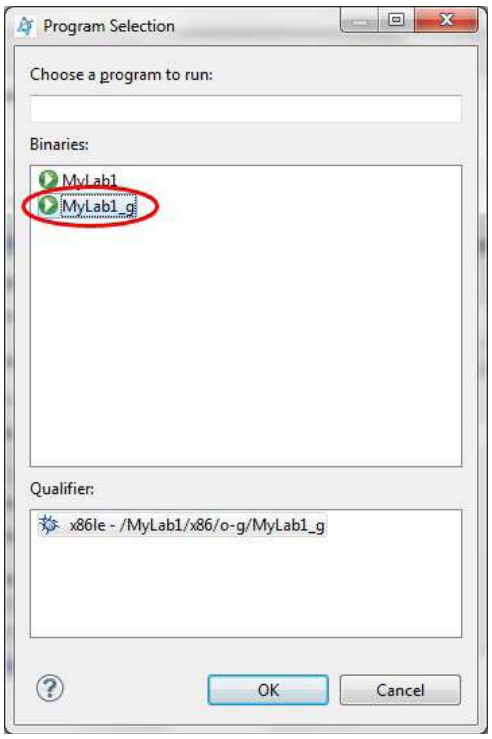
Если в программе предполагается ввод данных, то ввод данных также осуществляется в окне **Console** виртуального терминала ***QNX Momentics IDE***.

Отладка проекта

Для пошаговой отладки программы нужно создать новую конфигурацию запуска. Нажмите на стрелочку на кнопке ***Debug*** на панели инструментов. В выпадающем списке выберете ***Debug As→C/C++QNX Application:***



Теперь среди доступных бинарных файлов выберете сборку с отладочной информацией ***MyLab1_g*** и нажмите **OK**:

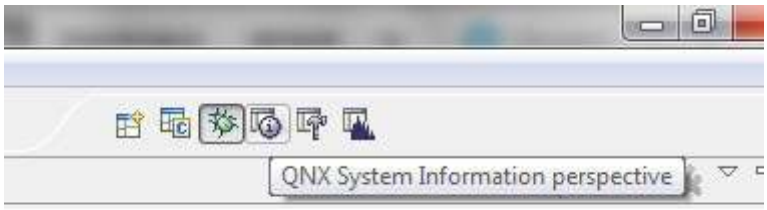


Среда разработки автоматически переключится в режим отладки ***Debug perspective***, которому соответствует другие расположение и набор окон и панелей инструментов. Программа будет остановлена на первой строке функции ***main()***. В пошаговом режиме отладки вы можете

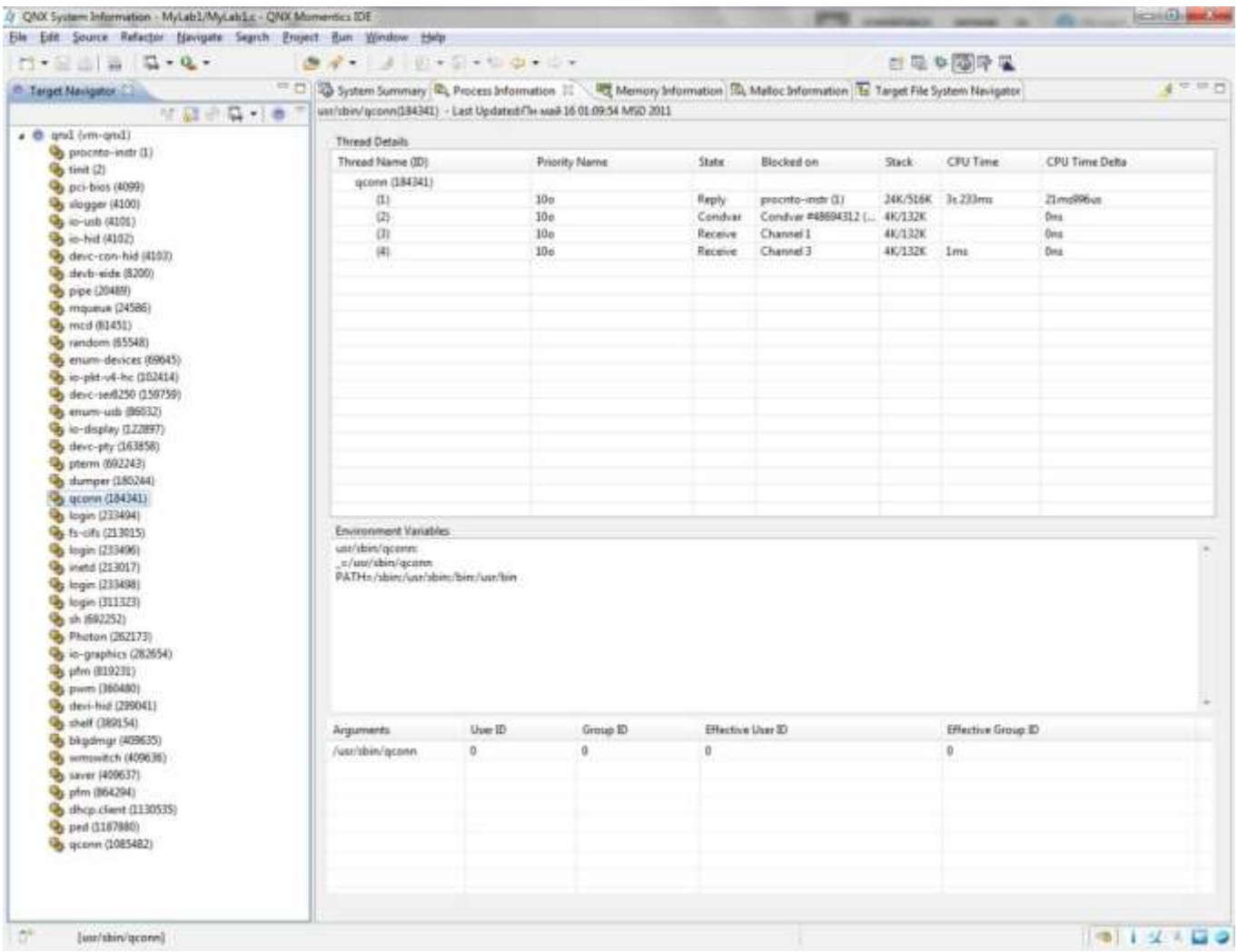
посмотреть на каждом шаге текущий стек вызова функций, содержимое локальных переменных, состояние регистров процессора, дизассемблированный код программы и многое другое, что может оказаться очень полезным при отладке приложения.

Просмотр информации о состоянии целевой системы

Информацию о состоянии целевой системы можно получить, если перейти в режим ***QNX System Information perspective***:



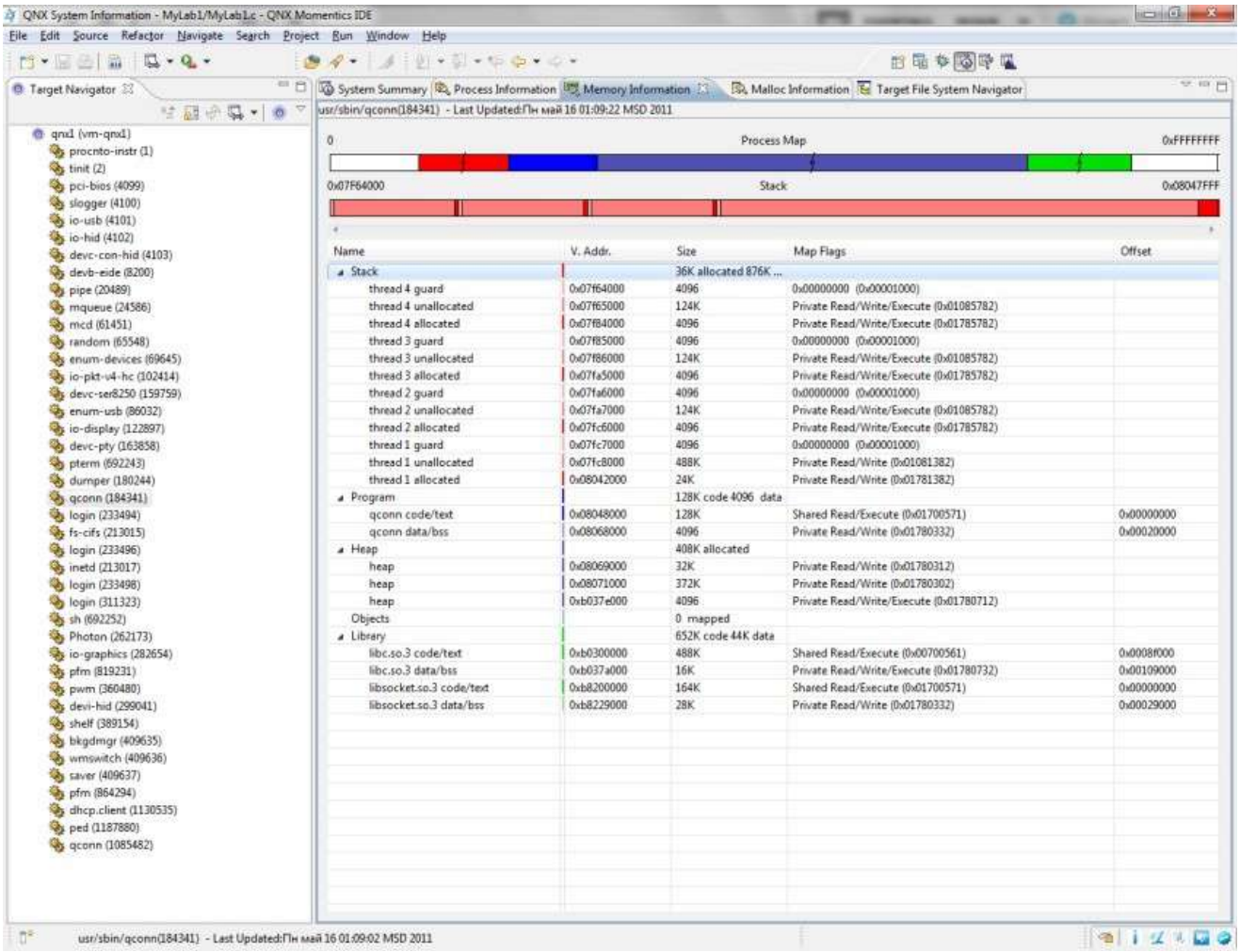
В результате откроется окно доступа к инструментам для мониторинга состояния целевой системы:



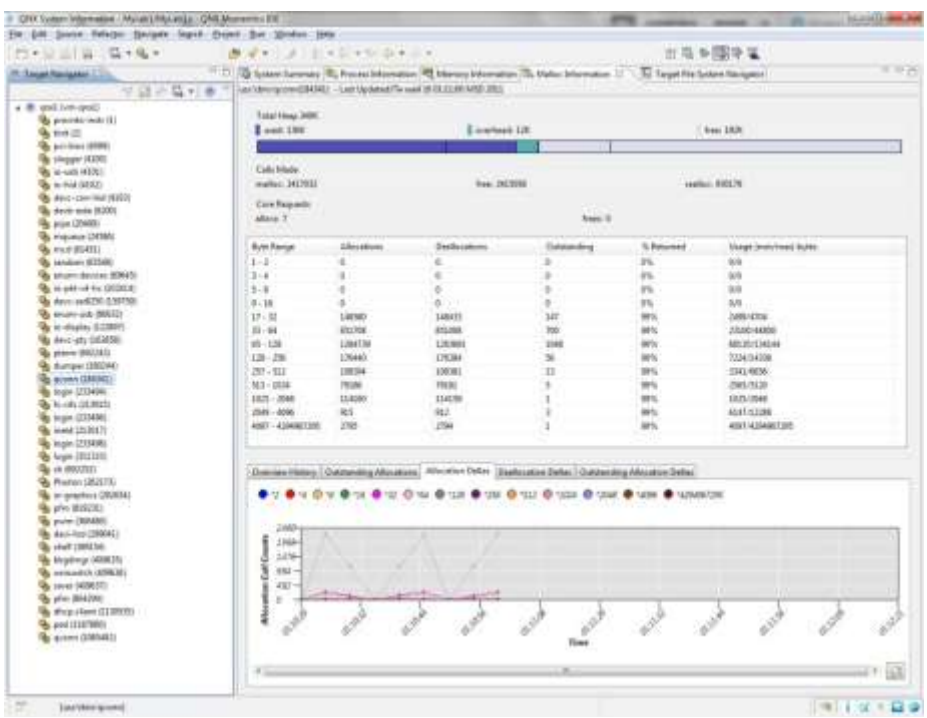
Слева в окне ***Target Navigator*** отображается дерево процессов, запущенных в данный момент в целевой системе. Для выбранного в окне ***Target Navigator*** процесса на вкладке ***Process Information*** представляется информация о процессе. Можно узнать, в каком состоянии находятся нити процесса, сколько стековой памяти занято каждой нитью, какова загрузка процессора и т.д. Всем нитям можно вручную задать приоритет, аффинити-маску и дисциплину диспетчеризации.

На вкладке ***Memory Information*** представлена информация о выделенных блоках памяти под каждую нить процесса. Доступна карта памяти всего процесса, состояние стековой памяти,

размеры блоков, выделенных в куче и т.д.:

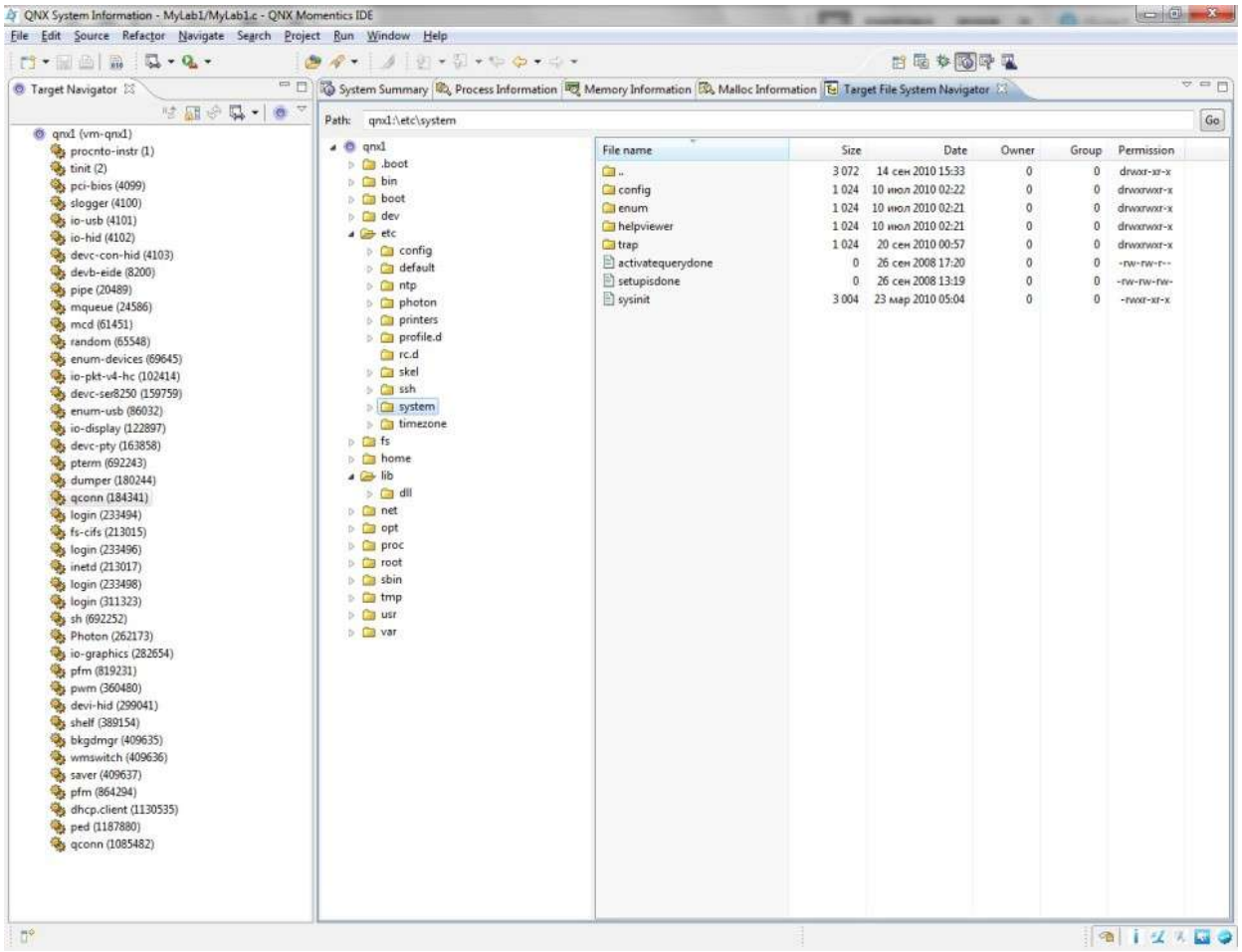


На вкладке ***Malloc Information*** представлена информация о состоянии кучи процесса, статистика по всем выделенным блокам разных размеров и история выделения блоков в куче в разные моменты времени в виде графиков:



На вкладке ***Target File System Navigation*** можно получить полный доступ к файловой системе целевой системы. Файлы можно удалять, переименовывать, копировать. Создавать новые каталоги и файлы. Менять наборы прав доступа к файлам и каталогам. Можно использовать

встроенный текстовый редактор файлов:



Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)"

Кафедра программных систем

А.В. Баландин

Лабораторная работа

Запуск и организация взаимодействия параллельных процессов

Методические указания

Самара
2012

Цель работы: изучить и освоить функции операционной системы QNX для запуска параллельных процессов и организации межпроцессного взаимодействия с помощью механизма обмена сообщениями.

Задание:

1. Перед выполнением полученного задания необходимо внимательно изучить следующие стандартные функции операционной системы QNX для запуска и организации взаимодействия параллельных процессов:

семейство функций `spawn();`

флаги запуска `P_WAIT`, `P_NOWAIT`, `P_NOWAITO`, `P_OVERLAY`;

функции управления каналом `ChannelCreate()`, `ChannelDestroy()`, `ConnectAttach()`,`ConnectDetach();`

функции передачи сообщений `MsgSend()`, `MsgReceive()`, `MsgReply()`,`MsgRead()`,`MsgWrite()`.

2. Написать программу, реализующую указанный вариант задания, максимально используя при программировании приведенные выше функции. Программа должна выводить на экран подробные комментарии к выполняемым действиям.

Форма отчетности: для отчета по лабораторной работе требуется представить следующее:

1) результат выполнения программы на компьютере в среде операционной системы QNX;

2) письменный отчет по результатам выполнения лабораторной работы, содержащий схему взаимодействия параллельных процессов в виде UML-диаграммы последовательностей, описание механизма взаимодействия процессов, программный код.

1.1 Варианты заданий

ВАРИАНТ №1

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства `spawn*()`, запускает процессы P2(M2) и P3(M3), передавая им в качестве параметра `chid` созданного канала, затем переходит в состояние ожидания сообщений по созданному каналу.

Процесс P2 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему сообщение о `chid` своего канала и, получив ответ от P1, переходит в состояние приема сообщений по созданному каналу.

Процесс P3 устанавливает соединение с каналом процесса P1 и посылает ему запрос на получение `pid` процесса P2 и `chid` его канала. Получив от P1 ответ (`pid` и `chid`), присоединяется к каналу процесса P2 и посылает ему сообщение "P3 send message to P2".

Процесс P2, приняв сообщение от процесса P3, добавляет к нему информацию "P2 send message to P1" и отправляет сформированное таким образом сообщение процессу P1.

Процесс P1, получив сообщение от P2, выдает его на экран терминала, посылает ответ "P1 OK" процессу P2 и терминируется.

Процесс P2, получив ответ от P1, выдает его на экран терминала, посылает ответ "P2 OK" процессу P3 и терминируется.

Процесс P3, получив ответ от P2, выдает его на экран терминала после чего выдает на терминал "P3 OK" и терминируется.

ВАРИАНТ №2

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства `spawn*()`, запускает процесс P2(M2), передавая ему в качестве параметра `chid` созданного канала, затем переходит в состояние приема сообщений по своему каналу.

Процесс P2 создает свой канал и, используя функцию семейства `spawn*()`, запускает процесс P3(M3), передавая ему в качестве параметра `chid` созданного канала, затем устанавливает соединение с каналом процесса P1, отправляет ему сообщение о `pid` процесса P3 и переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал, устанавливает соединение с каналом процесса P2 и посылает ему сообщение "P3 loaded". Получив ответ, посылает ему `chid` своего канала и переходит в состояние приема сообщений по своему каналу.

Процесс P2, приняв первое сообщение от процесса P3, отправляет его процессу P1, получив ответ от P1, принимает `chid` от процесса P3, посылает ему ответ и передает `chid` процессу P1. Далее выдает на терминал "P2 loaded" и переходит в состояние приема сообщений по своему каналу.

Процесс P1, получив первое сообщение от P2, выдает его на экран терминала, посылает ответ процессу P2 и принимает второе сообщение, устанавливает соединение с каналом процесса P3 и посылает по нему сообщение "stop", после ответа переходит в состояние приема сообщений по своему каналу.

Процесс P3, получив "stop" от процесса P1, отправляет его процессу P2, печатает "P3 stop" и терминируется.

Процесс P2, получив "stop", отправляет его процессу P1, печатает "P2 stop" и терминируется.

Процесс P1, получив "stop", печатает "P1 stop" и терминируется.

ВАРИАНТ №3

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства `spawn*()`, запускает процессы P2(M2) и P3(M3), передавая им в качестве параметра `chid` созданного канала, затем переходит в ожидание сообщений по своему каналу.

Процесс P2 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему сообщение о `chid` своего канала и переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему сообщение о `chid` своего канала и переходит в состояние приема сообщений по созданному каналу.

Процесс P1, приняв сообщение от процесса P2 или P3 о `chid` канала, устанавливает соединение и посылает по нему - "P1 send message to P?", принимает ответ и выдает его на терминал. После такого взаимодействия с P2 и P3 процесс P1 терминируется.

Процесс P?, получив сообщение от P1, выдает его на экран терминала, посылает ответ "P? OK" процессу P1 и терминируется.

ВАРИАНТ №4

Разработать приложение, состоящее из пяти взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 используя функцию семейства `spawn*()`, запускает процесс P2(M2) с флагом `P_NOWAIT` и P3(M2) с флагом `P_WAIT`.

Процесс P2 создает свой канал и, используя функцию семейства `spawn*()`, запускает процесс P1(M3), передавая в качестве аргумента `chid` своего канала, затем переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал и, используя функцию семейства `spawn*()`, запускает процесс P5(M3), передавая в качестве аргумента `chid` своего канала, затем переходит в состояние приема сообщений по созданному каналу.

Процесс P?(M3) устанавливает соединение с каналом родительского процесса P?(M2) и посылает сообщение "P?-OK", получив ответ , выдает его на терминал и терминируется.

Процесс P?(M2), получив сообщение от P?(M3), выдает его на экран терминала, посылает ответ "P? OK" процессу P?(M3) и терминируется.

Процесс P1, возобновив свою работу, выдает на терминал сообщение "STOP" и терминируется.

ВАРИАНТ №5

Разработать приложение, которое строится в виде цепочки процессов. Требуется написать два программных модуля – M1, M2. На базе модуля M1 из

`shell` запускается стартовый процесс P1(M1), которому передается в качестве параметра длина цепочки N (количество процессов).

Процесс P1 создает свой канал и, используя функцию семейства `spawn*()`, запускает процесс P2(M2), передавая в качестве аргумента длину цепочки N и `chid` своего канала, затем переходит в ожидание прихода сообщений по созданному каналу. После получения сообщения посылает ответ и терминируется.

Процесс P2(M2) устанавливает соединение с каналом родительского процесса и, если N не равно 0, создает свой канал и, используя функцию семейства `spawn*()`, запускает следующий процесс P?(M2), передавая ему в качестве аргумента N-1 и `chid` своего канала, затем переходит в состояние приема сообщений по своему каналу. После получения сообщения посылает ответ. Если N=0, то процесс P?(M2) посылает родительскому процессу сообщение "P? OK" и, получив ответ, выдает его на экран и терминируется. Аналогичным образом ведет себя каждый запущенный процесс цепочки.

ВАРИАНТ №6

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства `spawn*()`, запускает процесс P2(M2), передавая ему в качестве параметра `chid` созданного канала, затем переходит в состояние приема сообщений по созданному каналу.

Процесс P2 создает свой канал, и, используя функцию семейства `spawn*()`, запускает процесс P3(M3), передавая ему в качестве параметра `chid` созданного канала, затем устанавливает соединение с каналом процесса P1 и переходит в состояние приема сообщений по своему каналу.

Процесс P3 устанавливает соединение с каналом процесса P2 и посылает ему запрос на получение `pid` процесса P1 и `chid` его канала. Получив от P2 ответ (`pid` и `chid`), устанавливает соединение с каналом процесса P1 и посылает ему сообщение "P3 send message to P1". После получения ответа посылает процессу P2 сообщение "P3 stop", затем, получив ответ, выводит на экран "P3 OK" и терминируется.

Процесс P1, получив сообщение от процесса P3, выдает его на терминал, затем посылает ответ и переходит в ожидание сообщений по своему каналу.

Процесс P2, получив сообщение от процесса P3, выводит его на экран и посылает ответ. Затем посылает процессу P1 сообщение "P2 stop", получает ответ, выводит на экран "P2 OK" и терминируется.

Процесс P1, получив сообщение от процесса P2 выдает его на терминал, посылает ответ, выводит на экран "P1 OK" и терминируется.

ВАРИАНТ №7

Разработать приложение, состоящее из пяти взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает свой канал и, используя функцию семейства `spawn*()`, запускает процессы P2(M2) и P3(M2), передавая им в качестве аргумента `chid` своего канала, затем переходит в состояние приема сообщений по своему каналу.

Процесс P2 создает свой канал и, используя функцию семейства `spawn*()`, запускает процесс P1(M3), передавая в качестве аргумента `chid` своего канала, затем переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал и, используя функцию семейства `spawn*()`, запускает процесс P5(M3), передавая в качестве аргумента `chid` своего канала, затем переходит в состояние приема сообщений по созданному каналу.

Процесс P?(M3) устанавливает соединение с каналом родительского процесса P?(M2) и посылает запрос на получение `pid` процесса P1 и `chid` его канала, затем, после получения ответа (`pid` и `chid`), устанавливает соединение с каналом процесса P1 и посылает ему сообщение "P? loaded". После получения ответа выводит на экран "P? OK" и терминируется.

Процесс P?(M2) после ответа на запрос процесса P?(M3) выводит на терминал "P? OK" и терминируется.

Процесс P1, получив сообщение от процесса P1 или P5, выводит его на экран и посылает ответ. После взаимодействия с P1 и P5 процесс P1 выводит на экран "P1 OK" и терминируется.

ВАРИАНТ №8

Разработать приложение, состоящее из четырех взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства `spawn*()`, запускает процесс P2(M2), передавая ему в качестве параметра `chid` созданного канала, затем переходит в ожидание сообщений по своему каналу.

Процесс P2 создает свой канал, и, используя функцию семейства `spawn*()`, запускает процессы P3(M3) и P1(M3), передавая им в качестве параметра `chid` созданного канала, затем устанавливает соединение с каналом процесса P1 и переходит в состояние приема сообщений по своему каналу.

Процесс P?(M3) устанавливает соединение с каналом родительского процесса P2, отправляет ему сообщение "P? loaded", затем получает ответ, выводит его на экран и терминируется.

Процесс P2, получив сообщение от процесса P3 или P1, выводит его на экран, посылает ответ "P? stop". После такого взаимодействия с P3 и P1 процесс P2 посылает процессу P1 сообщение "P2 loaded", после получения ответа выводит его на экран и терминируется.

Процесс P1, приняв сообщение от процесса P2, посылает ответ "P2 stop", затем выводит на экран "P1 stop" и терминируется.

ВАРИАНТ №9

Разработать приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства `spawn*()`, запускает процесс P2(M2), передавая ему в качестве параметра `chid` созданного канала, затем переходит в состояние приема сообщений по созданному каналу.

Процесс P2 создает свой канал, и, используя функцию семейства `spawn*()`, запускает процесс P3(M3), передавая ему в качестве параметра `chid` созданного канала, затем устанавливает соединение с каналом процесса P1, передает ему `chid` своего канала, затем переходит в состояние приема сообщений по своему каналу.

Процесс P3 устанавливает соединение с каналом процесса P2 и посылает ему запрос на получение `pid` процесса P1 и `chid` его канала. Получив от P2 ответ (`pid` и chid), устанавливает соединение с каналом процесса P1 и посылает ему свой `pid` и `chid` своего канала, после чего переходит в состояние приема сообщений по своему каналу.

Процесс P1 устанавливает соединение с каналом процесса P2 и передает ему сообщение "P1 send message to P2", получив ответ, выводит его на терминал, затем устанавливает соединение с каналом процесса P3 и передает ему сообщение "P1 send message to P3", получает ответ, выводит его на экран и терминируется.

Процесс P2, получив сообщение от процесса P1, выводит его на терминал, посылает ответ "P2 OK" и терминируется.

Процесс P3, получив сообщение от процесса P1, выводит его на экран, посылает ответ "P3 OK" и терминируется.

ВАРИАНТ №10

Разработать приложение, состоящее из четырех взаимодействующих процессов. Требуется написать три программных модуля – M1, M2, M3. На базе модуля M1 из `shell` запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства `spawn*()`, запускает процессы P2(M2), P3(M2) и P1(M3), передавая им в качестве параметра `chid` созданного канала, затем переходит в ожидание сообщений по своему каналу.

Процесс P?(M2) создает свой канал, устанавливает соединение с каналом процесса P1, передает ему `chid` созданного канала и переходит в состояние приема сообщений по своему каналу.

Процесс P1 устанавливает соединение с каналом процесса P1, посылает ему запрос на получение `pid` процесса P?(M2) и `chid` его канала. Получив от P1 ответ (`pid` и `chid`), устанавливает соединение с каналом процесса P?(M2), посылает ему сообщение "P1 send message to P?" и получает ответ. После взаимодействия с процессами P2 и P3 процесс P1 выводит на экран "P1 OK" и терминируется.

Процесс P?(M2) после получения сообщения от P1 выводит его на экран и отправляет ответ, затем посылает процессу P1 сообщение "P? send message to P1", после получения ответа выводит на экран "P? OK" и терминируется.

Процесс P1, приняв сообщение от процесса P?(M2), выводит его на экран и отправляет ответ. После взаимодействия с процессами P2 и P3, процесс P1 выводит на экран "P1 OK" и терминируется.

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)"

Кафедра программных систем

А.В. Баландин

Лабораторная работа

Запуск и организация взаимодействия параллельных процессов в локальной сети

Методические указания

Самара
2012

Цель работы: Освоение механизма ОС QNX для организации межпроцессного взаимодействия распределенных параллельных процессов в локальной сети.

Общие требования

Используя задание и результаты выполнения лабораторной работы №1, внести изменения, направленные на применение механизма сигналов для передачи между процессами информации о chid созданных каналов (вместо передачи их в качестве параметров функции spawn*())

ЗАДАНИЕ 1

Разработать в локальной сети, включающей в себя узлы С1, С2, С3 с установленными именами компьютеров, распределенное приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля: М1, М2, М3. На базе модуля М1 из shell запускается стартовый процесс Р1(М1).

Процесс Р1 запускается на узле С1 локальной сети. Он создает канал и, используя функцию семейства spawn*(), запускает в соответствующих узлах локальной сети (зависит от варианта задания) процессы Р2(М2) и Р3(М3), передавая им необходимые параметры (зависит от варианта задания) для установления взаимодействия.

Процесс Р2 создает свой канал, устанавливает соединение с каналом процесса Р1, отправляет ему сообщение о chid своего канала и переходит в состояние приема сообщений по созданному каналу.

Процесс Р3 устанавливает соединение с каналом процесса Р1 и посылает ему запрос на получение pid процесса Р2, chid его канала и имени узла, на котором он запущен, для присоединения к каналу процесса Р2. Получив ответ (pid, chid, имя узла), присоединяется к каналу процесса Р2 и посылает ему сообщение - "Р2 загружен".

Процесс Р2, приняв сообщение от процесса Р3, добавляет к нему информацию "Р1 принял сообщение от Р2" и отправляет сформированное таким образом сообщение процессу Р1.

Процесс Р1, получив сообщение от Р2, выдает его на экран терминала, посылает ответ "Р1 ОК" процессу Р2 и завершает работу (терминируется).

Процесс Р2, получив ответ от Р1, выдает его на экран терминала, посылает ответ "Р2 ОК" процессу Р3 и завершает работу (терминируется).

Процесс Р3, получив ответ от Р2, выдает его на экран терминала после чего выдает на терминал "Р3 ОК" и завершает работу (терминируется).

Вариант 1: Р2 стартует на узле С2, Р3 стартует на узле С3

Вариант 2: Р2 стартует на узле С1, Р3 стартует на узле С2

Вариант 3: Р2 стартует на узле С2, Р3 стартует на узле С1

Вариант 4: Р2 стартует на узле С2, Р3 стартует на узле С2

ЗАДАНИЕ 2

Разработать в локальной сети, включающей в себя узлы С1, С2, С3 с установленными именами компьютеров, распределенное приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля: М1, М2, М3. На базе модуля М1 из shell запускается стартовый процесс Р1(М1).

Процесс Р1 запускается на узле С1 локальной сети. Он создает канал и, используя функцию семейства spawn*(), запускает в соответствующем узле локальной сети (зависит от варианта задания) процесс Р2(М2), передавая ему необходимые параметры (зависит от варианта

задания) для установления взаимодействия, и переходит в состояние приема сообщений от P2 по своему каналу.

Процесс P2 создает свой канал и, используя функцию семейства spawn*(), запускает в соответствующем узле локальной сети (зависит от варианта задания) процесс P3(M3), передавая ему необходимые параметры (зависит от варианта задания) для установления взаимодействия. Затем процесс P2 устанавливает соединение с каналом процесса P1, отправляет ему необходимые параметры процесса P3 (зависит от варианта задания) для установления взаимодействия и переходит в состояние приема сообщений от P3 по созданному каналу.

Процесс P3 создает свой канал, устанавливает соединение с каналом процесса P2 и посылает ему сообщение "P3 загружен". Получив ответ, посылает необходимые параметры (зависит от варианта задания) для установления взаимодействия и переходит в состояние приема сообщений по своему каналу.

Процесс P2, приняв первое сообщение ("P3 загружен"), отправляет его процессу P1, получив ответ от P1, принимает chid от процесса P3, посылает ему ответ и передает процессу P1 chid и имя узла процесса P3. После этого выдает на свой терминал "P2 загружен" и переходит в состояние приема сообщений по своему каналу от P3.

Процесс P1, получив первое сообщение от P2, выдает его содержание на экран своего терминала, посылает ответ процессу P2 и принимает второе сообщение. После этого P1 устанавливает соединение с каналом процесса P3 и посылает по нему сообщение "stop", после ответа переходит в ожидание сообщения по своему каналу.

Процесс P3, получив "stop", отправляет его процессу P2, печатает "stop P3" и терминируется.

Процесс P2, получив "stop", отправляет его процессу P1, печатает "stop P2" и терминируется.

Процесс P1, получив "stop", печатает "stop P1" и терминируется.

Вариант 1: P2 стартует на узле C2, P3 стартует на узле C3

Вариант 2: P2 стартует на узле C1, P3 стартует на узле C2

Вариант 3: P2 стартует на узле C2, P3 стартует на узле C1

Вариант 4: P2 стартует на узле C2, P3 стартует на узле C2

ЗАДАНИЕ 3

Разработать в локальной сети, включающей в себя узлы C1, C2, C3 с установленными именами компьютеров, распределенное приложение, состоящее из трех взаимодействующих процессов. Требуется написать три программных модуля: M1, M2, M3. На базе модуля M1 из shell запускается стартовый процесс P1(M1).

Процесс P1 создает канал и, используя функцию семейства spawn*(), запускает в соответствующих узлах локальной сети (зависит от варианта задания) процессы P2(M2) и P3(M3), передавая им необходимые параметры (зависит от варианта задания) для установления взаимодействия, и переходит в ожидание сообщений по своему каналу.

Процесс P2 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему необходимые параметры (зависит от варианта задания) для установления взаимодействия и переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал, устанавливает соединение с каналом процесса P1, отправляет ему необходимые параметры (зависит от варианта задания) для установления взаимодействия и переходит в состояние приема сообщений по созданному каналу

Процесс P1, приняв сообщение от процесса P2 или P3, устанавливает соединение и посылает по нему - "P1 принял сообщение от P?", принимает ответ и выдает его на терминал. После такого взаимодействия с P2 и P3, P1 терминируется.

Процесс P?, получив сообщение от P1, выдает его на экран терминала, посылает ответ "P? ОК" процессу P1 и завершает работу (терминируется).

Вариант 1: P2 стартует на узле C2, P3 стартует на узле C3

Вариант 2: P2 стартует на узле C1, P3 стартует на узле C2

Вариант 3: P2 стартует на узле C2, P3 стартует на узле C1

Вариант 4: P2 стартует на узле C2, P3 стартует на узле C2

ЗАДАНИЕ 4

Разработать в локальной сети, включающей в себя узлы C1, C2, C3 с установленными именами компьютеров, распределенное приложение, состоящее из пяти взаимодействующих процессов. Требуется написать три программных модуля: M1, M2, M3. На базе модуля M1 из shell запускается стартовый процесс P1(M1) на узле C1 локальной сети с заданным именем.

Процесс P1 используя функцию семейства spawn*(), запускает в соответствующих узлах локальной сети (зависит от варианта задания) процессы P2(M2) и P3(M2) и ожидает их завершения.

Процесс P2 создает свой канал и, используя функцию семейства spawn*(), запускает в соответствующем узле локальной сети (зависит от варианта задания) процесс P4(M3), передавая необходимые параметры (зависит от варианта задания) для установления взаимодействия, и переходит в состояние приема сообщений по созданному каналу.

Процесс P3 создает свой канал и, используя функцию семейства spawn*(), запускает в соответствующем узле локальной сети (зависит от варианта задания) процесс P5(M3), передавая необходимые параметры (зависит от варианта задания) для установления взаимодействия, и переходит в состояние приема сообщений по созданному каналу.

Процесс P?(M3) устанавливает соединение с каналом родительского процесса P?(M2) и посылает сообщение "P?-ОК", получив ответ , выдает его на терминал и терминируется

Процесс P?(M2), получив сообщение от P?(M3), выдает его на экран терминала, посылает ответ "P? ОК" процессу P?(M3) и завершает работу (терминируется).

Процесс P1, дождавшись завершения процессов P2(M2) и P3(M2), возобновляет свою работу, выдает на терминал сообщение "STOP" и терминируется.

Вариант 1: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 2: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C2, P5 стартует на узле C1

Вариант 3: P2 стартует на узле C2, P4 стартует на узле C1, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 4: P2 стартует на узле C2, P4 стартует на узле C2, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 5: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C2, P5 стартует на узле C2

Вариант 6: P2 стартует на узле C1, P4 стартует на узле C2, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 7: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C1, P5 стартует на узле C3

Вариант 8: P2 стартует на узле C1, P4 стартует на узле C1, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 9: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C3, P5 стартует на узле C2

Вариант 10: P2 стартует на узле C1, P4 стартует на узле C2, P3 стартует на узле C2, P5 стартует на узле C3

ЗАДАНИЕ 5

Разработать в локальной сети, включающей в себя узлы C1, C2, C3 с установленными именами компьютеров, распределенное приложение, которое строится в виде цепочки из 4 процессов, запускаемых на заданных узлах локальной сети. Требуется написать два программных модуля: M1, M2. На базе модуля M1 из shell запускается стартовый процесс P1(M1) на узле C1 локальной сети с заданным именем.

Процесс P1 создает свой канал и, используя функцию семейства spawn*(), запускает на соответствующем узле локальной сети (зависит от варианта задания) процесс P2(M2), передавая необходимые параметры (зависит от варианта задания) для установления взаимодействия, и переходит в ожидание прихода по нему сообщения от процесса P2, после чего отвечает ему и терминируется.

Процесс P2(M2) устанавливает соединение с каналом родительского процесса, создает свой канал и, используя функцию семейства spawn*(), запускает на соответствующем узле локальной сети (зависит от варианта задания) следующий процесс P3(M2), передавая необходимые параметры (зависит от варианта задания) для установления взаимодействия, и переходит в состояние приема сообщений по своему каналу.

Процесс P3(M2) устанавливает соединение с каналом родительского процесса, создает свой канал и, используя функцию семейства spawn*(), запускает на соответствующем узле локальной сети (зависит от варианта задания) следующий процесс P4(M2), передавая необходимые параметры (зависит от варианта задания) для установления взаимодействия, и переходит в состояние приема сообщений по своему каналу Процесс P4 посылает процессу P3 сообщение "P4-OK" и, получив ответ, выдает его на экран своего терминала и терминируется.

Аналогично ведут себя процессы P3 и P2 цепочки. Терминирование процесса P1 завершает работу всего приложения.

Вариант 1: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 2: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C2, P5 стартует на узле C1

Вариант 3: P2 стартует на узле C2, P4 стартует на узле C1, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 4: P2 стартует на узле C2, P4 стартует на узле C2, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 5: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C2, P5 стартует на узле C2

Вариант 6: P2 стартует на узле C1, P4 стартует на узле C2, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 7: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C1, P5 стартует на узле C3

Вариант 8: P2 стартует на узле C1, P4 стартует на узле C1, P3 стартует на узле C2, P5 стартует на узле C3

Вариант 9: P2 стартует на узле C2, P4 стартует на узле C3, P3 стартует на узле C3, P5 стартует на узле C2

Вариант 10: P2 стартует на узле C1, P4 стартует на узле C2, P3 стартует на узле C2, P5 стартует на узле C3

Порядок отчета:

1. Проверка знания средств запуска распределенных процессов и механизмов канального взаимодействия.
2. Демонстрация работы распределенного мультипроцессного приложения в локальной сети.
3. Оформление письменного отчета, по результатам выполнения лабораторной работы.

Содержание отчета:

- Название лабораторной работы, номер задания и варианта
- Фамилия И.О.
- № группы
- Тексты программных модулей процессов

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)"

Кафедра программных систем

А.В. Баландин

Лабораторная работа

Запуск и синхронизация выполнения нитей

Методические указания

Самара
2012

Содержание

Цель работы: Освоение функций ОСРВ для запуска и синхронизации нитей.

ЗАДАНИЕ 1.

Разработать приложение, состоящее из одного процесса с тремя запущенными нитями:

1. M.
2. T1.
3. T2.

В качестве нити M выступает функция main(). Нити T1(F1) и T2(F2) запускаются на базе разных функций (F1 и F2). На внешнем уровне процесса определен текстовый буфер, который используется нитями для совместного формирования в нём текста. В исходном состоянии буфер пуст. Порядок работы приложения следующий.

Нить M запускает нити T1 и T2, передавая им в качестве параметра указатель буфера, после чего первой заносит в пустой буфер текст - "Main started\n", и ожидает (выбрав механизм синхронизации из указанных для варианта задания) завершения дозаписи в буфер своих частей текста обеими нитями – T1 и T2.

После запуска нить T1 ожидает (выбрав механизм синхронизации из указанных для варианта задания) доступ к буферу для дозаписи в него (после нити M) текста - "Thread 1 written down\n". После дозаписи текста в буфер нить T1 ожидает завершения нити T2, после чего терминируется, выдав на терминал сообщение - "Thread 1 terminated\n".

После запуска нить T2, ожидает (выбрав механизм синхронизации из указанных для варианта задания) своей очереди доступа к буферу после нити T1 для дозаписи в него текста - "Thread 2 written down\n", после чего нить T2 терминируется, выдав на терминал сообщение - "Thread 2 terminated\n".

В итоге нить M должна распечатать сформированное содержимое буфера и завершается последней, выдав на терминал сообщение - "Main terminated\n ".

Варианты:

1.	Блокировки чтения/записи. Мутексы. Присоединение.
2.	Блокировки чтения/записи. Мутексы. Барьеры.
3.	Блокировки чтения/записи. Ждущие блокировки. Присоединение.
4.	Блокировки чтения/записи. Ждущие блокировки. Барьеры.
5.	Блокировки чтения/записи. Условные переменные. Присоединение.
6.	Блокировки чтения/записи. Условные переменные. Барьеры.
7.	Неименованные семафоры. Условные переменные. Присоединение.
8.	Неименованные семафоры. Условные переменные. Барьеры.
9.	Именованные семафоры. Ждущие блокировки. Присоединение.
10.	Именованные семафоры. Ждущие блокировки. Барьеры.
11.	Мутексы. Ждущие блокировки. Присоединение.
12.	Мутексы. Ждущие блокировки. Барьеры.

ЗАДАНИЕ 2.

Разработать приложение, состоящее из одного процесса с тремя запущенными нитями:

1. M.
2. T1.
3. T2.

В качестве нити M выступает функция main(). Нити T1(F1) и T2(F2) запускаются на базе разных функций (F1 и F2). На внешнем уровне процесса определен текстовый буфер, который используется нитями для совместного формирования в нём текста. В исходном состоянии буфер пуст. Порядок работы приложения следующий.

Нить M запускает нить T1(F1), передавая ей в качестве параметров указатель буфера и функцию F2, после чего первой заносит в пустой буфер текст - "Main started\n", и ожидает (выбрать механизм синхронизации из указанных для варианта задания) завершения дозаписи в буфер своих частей текста нитями – T1 и T2.

После запуска нить T1 запускает нить T2 и ожидает (выбрать механизм синхронизации из указанных для варианта задания) доступ к буферу для дозаписи в него (после нити M) текста - "Thread 1 written down\n". После дозаписи текста в буфер нить T1 ожидает завершения нити T2, после чего терминируется, выдав на терминал сообщение - "Thread 1 terminated\n".

После запуска нить T2, ожидает (выбрать механизм синхронизации из указанных для варианта задания) своей очереди дозаписи в буфер после нити T1 текста - "Thread 2 written down\n", после чего нить T2 терминируется, выдав на терминал сообщение - "Thread 2 terminated\n".

В итоге нить M должна распечатать сформированное содержимое буфера и завершается последней, выдав на терминал сообщение - "Main terminated\n ".

Варианты:

1.	Блокировки чтения/записи. Мутексы. Присоединение.
2.	Блокировки чтения/записи. Мутексы. Барьеры.
3.	Блокировки чтения/записи. Ждущие блокировки. Присоединение.
4.	Блокировки чтения/записи. Ждущие блокировки. Барьеры.
5.	Блокировки чтения/записи. Условные переменные. Присоединение.
6.	Блокировки чтения/записи. Условные переменные. Барьеры.
7.	Неименованные семафоры. Условные переменные. Присоединение.
8.	Неименованные семафоры. Условные переменные. Барьеры.
9.	Именованные семафоры. Ждущие блокировки. Присоединение.
10.	Именованные семафоры. Ждущие блокировки. Барьеры.
11.	Мутексы. Ждущие блокировки. Присоединение.
12.	Мутексы. Ждущие блокировки. Барьеры.

ЗАДАНИЕ 3.

Разработать приложение, состоящее из одного процесса с тремя запущенными нитями:

1. M.
2. T1.
3. T2.

В качестве нити M выступает функция main(). Нити T1(F) и T2(F) запускаются на базе одной функции F. На внешнем уровне процесса определен текстовый буфер, который используется нитями для совместного формирования в нём текста. В исходном состоянии буфер пуст. Порядок работы приложения следующий.

Нить M запускает нити T1(F) и T2(F), передавая им в качестве параметров указатель буфера и символьную строку с номером нити - "1" и "2" соответственно, после чего нить M первой заносит в пустой буфер текст - "Main started\n" и ожидает (выбрать механизм синхронизации из указанных для варианта задания) завершения дозаписи в буфер своих частей текста нитями – T1 и T2.

После запуска нить T1 ожидает (выбрать механизм синхронизации из указанных для варианта задания) доступ к буферу для дозаписи в него (после нити M) текста - "T1 written down\n". После дозаписи текста в буфер нить T1 ожидает завершения записи в буфер своего текста нитю T2, после чего нить T1 терминируется, выдав на терминал сообщение - "T1 terminated\n".

После запуска нить T2, ожидает (выбрать механизм синхронизации из указанных для варианта задания) своей очереди дозаписи в буфер после нити T1 текста - "T2 written down\n", после чего нить T2 терминируется вместе с нитью T1, выдав на терминал сообщение - "T2 terminated\n".

В итоге нить M должна распечатать сформированное содержимое буфера и завершается последней, выдав на терминал сообщение - "Main terminated\n ".

Варианты:

1.	Блокировки чтения/записи. Мутексы. Присоединение. Барьеры.
2.	Блокировки чтения/записи. Ждущие блокировки. Присоединение. Барьеры.
3.	Блокировки чтения/записи. Условные переменные. Присоединение. Барьеры.
4.	Неименованные семафоры. Условные переменные. Присоединение. Барьеры.
5.	Именованные семафоры. Ждущие блокировки. Присоединение. Барьеры.
6.	Мутексы. Ждущие блокировки. Присоединение. Барьеры.

ПОРЯДОК ВЫПОЛНЕНИЯ И ОТЧЁТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

При выполнении задания в соответствии с вариантом задания обязательно использовать все указанные в нем методы синхронизации в качестве основных. Другие методы синхронизации можно использовать, обосновав их необходимость, в дополнение к указанным в варианте задания.

Прием результатов лабораторной работы

1. Демонстрация работы многонитиевого процесса в соответствии с заданием.
2. Проверка теоретических знаний по средствам запуска и синхронизации нитей.
3. Представление оформленного отчета, по результатам выполнения лабораторной работы.

Отчет печатается на бумаге формата А4 через один интервал размер 12пт. По структуре отчет должен содержать стандартно оформленный титульный лист, текст задания с номером задания и номером варианта, описание порядка выполнения приложения и использования методов синхронизации, исходный текст программного модуля с подробным комментарием.

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)"

Кафедра программных систем

А.В. Баландин

Лабораторная работа

Синхронизация нитей с реальным временем

Методические указания

Самара
2012

Содержание

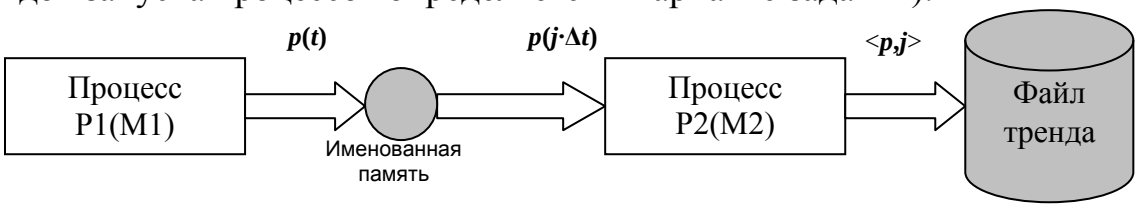
Цель работы: Освоение функций ОСРВ для работы с именованной памятью и синхронизации нитей с реальным временем.

ЗАДАНИЕ

Разработать программную систему мониторинга абстрактного физического объекта $O_{p(t)}$, $p(t)$ – изменяющийся во времени параметр. Мониторинг объекта $O_{p(t)}$ осуществляется на относительном интервале времени [0,T] с заданной единицей шкалы времени - **1t**.

Изменение параметра $p(t)$ во времени моделируется функцией $p(t)=F(t)$, где $t \in [0,T]$ - момент времени получения значения $p(t)$ на интервале времени $[0,T]$ со шкалой времени - **1t**.

Программная система должна содержать два процесса P1(M1) и P2(M2), запускаемых на базе своих программных модулей соответственно M1 и M2 (порядок запуска процессов определяется в варианте задания):



Процесс P1 создаёт ***именованную память*** и, начиная с $t=0$, периодически записывает в неё значение $p_i=F(1t \cdot i)$, где, i – порядковый номер очередного вычисленного значения параметра $p(t)$ в момент интервального времени $t=1t \cdot i \in [0,T]$, **i=0,1,2, ….**

Процесс P2, начиная с момента времени $t=0$, периодически считывает из именованной памяти значение моделируемого параметра $p(t)$ с заданным периодом Δt и формирует датированное значение в виде пары - $<p,j>$, где $p=p(j \cdot \Delta t)$, $j=0,1,2, \ldots$ – порядковый номер очередного считывания значения параметра $p(t)$ на интервале времени $[0,T]$, Результаты периодического считывания значений параметра $p(t)$, заносятся процессом P2 в текстовый файл (тренд $<p,j>$), в виде строк формата:

"<*p*><**Табуляция**><*j*>\n>"

Процессы P1 и P2 должны быть синхронизированы по моменту времени $t=0$ (должны начать действовать "одновременно", метод синхронизации выбрать самостоятельно).

При $t>T$ работа программной системы должна завершиться (все процессы терминироваться).

Для тренда параметра $p(t)$ построить график, например, загрузив содержимое файла с трендом в MS EXCEL.

Варианты задания

Номер варианта	Порядок запуска процессов P1 и P2	Вид функции **F(t)**	Единица временной шкалы 1t (сек)	Единица временной шкалы Δt (сек)	Значение T (сек)
1.	Процессы P1 и P2 запускаются из командной строки shell. Процесс P1 создает именованную память.	$F(t)=2.8e^{-1.5t}\sin(2.5\pi t)+1$	0.03, уведомление сигналом	0.1, уведомление импульсом	27
2.	Процессы P1 и P2 запускаются из командной строки shell. Процесс P1 создает именованную память.	$F(t)=2.8e^{-1.5t}\sin(2.5\pi t)+1$	0.03, уведомление импульсом	0.1, уведомление сигналом	27
3.	Процессы P1и P2 программно запускаются стартовым процессом P0, запускаемым из командной строки shell. Процесс P0 создает именованную память.	$F(t)=100e^{-t}\cos(0.1t+100)$	0.02, уведомление сигналом	0.1, уведомление импульсом	33
4.	Процессы P1и P2 программно запускаются стартовым процессом P0, запускаемым из командной строки shell. Процесс P0 создает именованную память.	$F(t)=100e^{-t}\cos(0.1t+100)$	0.02, уведомление импульсом	0.1, уведомление сигналом	33
5.	Процесс P1 запускается из командной строки shell, создает именованную память и программно запускает процесс P2.	$F(t)=100t^2e^{-0.1t}$	0.05, уведомление сигналом	0.1, уведомление импульсом	41
6.	Процесс P1 запускается из командной строки shell, создает именованную память и программно запускает процесс P2.	$F(t)=100t^2e^{-0.1t}$	0.05, уведомление импульсом	0.1, уведомление сигналом	41

7.	Процесс P2 запускается из командной строки shell, создает именованную память и программно запускает процесс P1.	$F_{P1}=100t^2e^{-0.1t}$	0.05, уведомление сигналом	0.2, уведомление импульсом	57
8.	Процесс P2 запускается из командной строки shell, создает именованную память и программно запускает процесс P1.	$F_{P1}=100t^2e^{-0.1t}$	0.05, уведомление импульсом	0.2, уведомление сигналом	57
9.	Процесс P1 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 программно запускает процесс P2	$F_{P2}=100e^{-t}\cos(0.1t+100)$	0.03, уведомление сигналом	0.2, уведомление импульсом	83
10.	Процесс P1 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 программно запускает процесс P2	$F_{P2}=100e^{-t}\cos(0.1t+100)$	0.03, уведомление импульсом	0.2, уведомление сигналом	83
11.	Процесс P2 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 программно запускает процесс P1	$F_{P3}=100e^{-t}\sin(0.1t+100)$	0.08, уведомление сигналом	0.2, уведомление импульсом	99
12.	Процесс P2 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 программно запускает процесс P1	$F_{P3}=100e^{-t}\sin(0.1t+100)$	0.08, уведомление импульсом	0.2, уведомление сигналом	99

13.	Процессы P1 и P2 запускаются из командной строки shell. Процесс P2 создает именованную память.	$F_{P1}=100e^{-t}\cos(0.1t+100)$	0.09, уведомление сигналом	0.3, уведомление импульсом	101
14.	Процессы P1 и P2 запускаются из командной строки shell. Процесс P2 создает именованную память.	$F_{P1}=100e^{-t}\cos(0.1t+100)$	0.09, уведомление импульсом	0.3, уведомление сигналом	101
15.	Процессы P1и P2 программно запускаются стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 создает именованную память.	$F_{P2}=100t^2e^{-0.1t}$	0.05, уведомление сигналом	0.3, уведомление импульсом	101
16.	Процессы P1и P2 программно запускаются стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 создает именованную память.	$F_{P2}=100t^2e^{-0.1t}$	0.05, уведомление импульсом	0.3, уведомление сигналом	107
17.	Процессы P1и P2 программно запускаются стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 создает именованную память.	$F_{P3}=100e^{-t}\sin(0.1t+100)$	0.17, уведомление сигналом	0.3, уведомление импульсом	107
18.	Процессы P1и P2 программно запускаются стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 создает именованную память.	$F_{P3}=100e^{-t}\sin(0.1t+100)$	0.17, уведомление импульсом	0.3, уведомление сигналом	107
19.	Процесс P1 запускается из командной строки shell и программно	$F_{P1}=200t^2e^{-0.2t}$	0.12, уведомление сигналом	0.4, уведомление импульсом	107

	запускает процесс P2, который создает именованную память.				
20.	Процесс P1 запускается из командной строки shell и программно запускает процесс P2, который создает именованную память.	$F_{P1}= 200t^2e^{-0.2t}$	0.12, уведомление импульсом	0.4, уведомление сигналом	107
21.	Процесс P1 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 программно запускает процесс P2. Именованную память создает процесс P0.	$F_{P2}=100t^2e^{-0.1t}$	0.09, уведомление сигналом	0.4, уведомление импульсом	107
22.	Процесс P1 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 программно запускает процесс P2. Именованную память создает процесс P0.	$F_{P2}=100t^2e^{-0.1t}$	0.09, уведомление импульсом	0.4, уведомление сигналом	107
23.	Процесс P2 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 программно запускает процесс P1. Именованную память создает процесс P0.	$F_{P3}= 200e^{-0.1t}\sin(0.5t+100)$	0.18, уведомление сигналом	0.4, уведомление импульсом	107
24.	Процесс P2 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 программно	$F_{P3}= 200e^{-0.1t}\sin(0.5t+100)$	0.18, уведомление импульсом	0.4, уведомление сигналом	107

	запускает процесс P1. Именованную память создает процесс P0.				
25.	Процесс P1 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 программно запускает процесс P2. Именованную память создает процесс P1.	$F_{P1}= 200t^2e^{-0.2t}$	0.16, уведомление сигналом	0.5, уведомление импульсом	107
26.	Процесс P1 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P1 программно запускает процесс P2. Именованную память создает процесс P1.	$F_{P1}= 200t^2e^{-0.2t}$	0.16, уведомление импульсом	0.5, уведомление сигналом	107
27.	Процесс P2 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 программно запускает процесс P1. Именованную память создает процесс P2.	$F_{P2}=100e^{-t}\cos(0.1t+100)$	0.08, уведомление сигналом	0.5, уведомление импульсом	107
28.	Процесс P2 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 программно запускает процесс P1. Именованную память создает процесс P2.	$F_{P2}=100e^{-t}\cos(0.1t+100)$	0.08, уведомление импульсом	0.5, уведомление сигналом	107
29.	Процесс P1 программно запускается стартовым процессом P0, запускаемым из	$F_{P3}=100t^2e^{-0.1t}$	0.22, уведомление сигналом	0.5, уведомление импульсом	107

	командной строки shell. Процесс P1 программно запускает процесс P2. Именованную память создает процесс P2.				
30.	Процесс P2 программно запускается стартовым процессом P0, запускаемым из командной строки shell. Процесс P2 программно запускает процесс P1. Именованную память создает процесс P1.	$F_{P1}= 250t^2e^{-0.25t}$	0.18, уведомление импульсом	0.6, уведомление сигналом	107

ПОРЯДОК ВЫПОЛНЕНИЯ И ОТЧЁТА ПО ЛАБОРАТОРНОЙ РАБОТЕ

При выполнении задания в соответствии с вариантом задания обязательно использовать указанный порядок запуска процессов (нитей), создания именованной памяти и типов уведомления. Можно использовать, в дополнение к указанным в варианте задания, вспомогательные процессы или нити, обосновав их необходимость.

Прием результатов лабораторной работы

1. Демонстрация работы программной системы в соответствии с заданием.
2. Проверка теоретических знаний по средствам запуска процессов (нитей) и синхронизации нитей с реальным временем.
3. Представление оформленного отчета, по результатам выполнения лабораторной работы, с графиком тренда.

Отчет печатается на бумаге формата А4 через один интервал размер 12пт. По структуре отчет должен содержать стандартно оформленный титульный лист, текст задания с номером задания и номером варианта, описание порядка выполнения приложения и использования методов синхронизации, исходный текст программного модуля с подробным комментарием.

Литература

1. Баландин А.В. Средства разработки программного обеспечения систем реального времени [Электронный документ формата СНМ]: учебное пособие: В 4 ч. Ч.1. Язык программирования С. - Самар. гос. аэрокосм. ун-т. Кафедра программных систем. Самара, 2012. 10 с.

2. Баландин А.В. Средства разработки программного обеспечения систем реального времени [Электронный документ формата CHM]: учебное пособие: В 4 ч. Ч.2. Программирование параллельных процессов в ОСРВ QNX Neutrino. - Самар. гос. аэрокосм. ун-т. Кафедра программных систем. Самара, 2012. 10 с.
3. Баландин А.В. Средства разработки программного обеспечения систем реального времени [Электронный документ формата CHM]: учебное пособие: В 4 ч. Ч.3. Программирование нитей в ОСРВ QNX Neutrino. - Самар. гос. аэрокосм. ун-т. Кафедра программных систем. Самара, 2012. 10 с.

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)"

Кафедра программных систем

А.В. Баландин

Лабораторная работа

Программирование прерываний в ОСРВ QNX6

Методические указания

Самара
2012

ЦЕЛЬ РАБОТЫ: Освоение средств и методов программирования прерываний в ОСРВ QNX6.

ЗАДАНИЕ N 1

Написать процесс, который следит за истечением заданного временного интервала, анализируя поток запросов прерывания, поступающих от системного таймера. По истечении временного интервала процесс выводит на экран сообщение и завершает работу. Нить main() вводит значение временного интервала (количество секунд) и переходит в состояние ожидания сигнала от нити, обрабатывающей прерывания. Нить, обрабатывающая прерывания, ведёт счетчик прерываний, который выводит на экран, а по истечении заданного интервала посылает сигнал о его завершении нити. Когда сигнал поступает, нить main() выдает на экран сообщение об истечении заданного временного интервала и завершает выполнение.

ЗАДАНИЕ N 2

Написать процесс, который считает количество нажатий любой клавиши клавиатуры кроме Ctrl и Esc (с учетом удержания клавиши и отпускания) и выдает текущее значение счетчика нажатий на экран. Процесс допускает управление путем нажатия следующих клавиш:

- сброс в счётчика в 0 (нажатие Ctrl);
- завершить работу программы (Esc).

Нить, обрабатывающая прерывания, анализирует поток запросов прерывания, поступающих от клавиатуры, печатает обновленное значение счетчика или посылает импульс с управляющим кодом. Нить main() находится в состоянии ожидания импульса, информирующего о необходимости сбросить счетчик или завершить программу, и выполняет команду.

ЗАДАНИЕ N 3

Написать процесс, который по нажатию на клавиатуре клавиши выдает на экран значение этой клавиши. Процесс ловит запросы прерывания от клавиатуры, извлекает код из буфера клавиатуры и анализирует его значение. При нажатии Esc программа завершает свою работу.

ЗАДАНИЕ N 4

Программа выдает на экран значение счетчика текущего интервала времени, прошедшего с момента ее запуска (количество сек). Программа допускает управление путем нажатия следующих клавиш:

- сброс счетчика в 0 (Ctrl/S);
- завершить работу программы (Esc).

ЗАДАНИЕ N 5

Написать процесс, который с частотой в 1 сек выдает на экран "мерцающую цифру" (цифра на экране через 1 сек заменяется пробелом, а спустя 0.5 сек вновь появляется цифра). Начальная цифра - 0. Цифра может быть изменена на другую путем нажатия соответствующей цифровой клавиши. Программа включает в себя две нити обработки прерываний. Одна из них анализирует поток запросов прерывания, поступающих от системного таймера, а другая - от клавиатуры. При нажатии Esc программа завершает свою работу.

ЗАДАНИЕ N 6

Написать процесс, который выдает на экран значение текущего интервала времени, прошедшего с момента ее запуска(мин : сек). Процесс завершает работу при нажатии клавиши Esc. Процесс анализирует поток запросов прерывания, поступающих от системного таймера и от клавиатуры, и выдает на экран значения минут и секунд или завершает свое выполнение.

ЗАДАНИЕ N 7

Написать процесс, который выдает на экран значение текущего интервала времени, прошедшего с момента ее запуска(мин : сек). Процесс допускает управление путем нажатия следующих клавиш:

- - сброс в 0 сек (Ctrl + S);
- сброс в 0 мин (Ctrl + M);
- - сброс в 0 и мин и сек (Ctrl + C);
- - завершить работу программы (Esc).

Программа включает в себя две interrupt-функции. Одна из них анализирует поток запросов прерывания, поступающих от системного таймера, а другая - от клавиатуры. Main-функция находится в состоянии ожидания(бесконечный цикл) сообщений, поступающих от interrupt-функций и выдает на экран значения минут и секунд . Кроме того она ожидает сигнал завершения программы и завершает свое выполнение при его поступлении.

ЗАДАНИЕ N 8

Повесить на прерывание от клавиатуры обработчик, который при нажатии клавиши изменяет значение переменной, играющей роль сигнала, на противоположное. Сигнал принимает только два значения - 0 и 1 .

Фоновая программа в цикле контролирует изменение значения сигнала, фиксируя тем самым нажатие клавиши. На нажатие клавиши фоновая программа реагирует следующим образом:

1. Читает скэн-код.
2. Печатает скэн-код и соответсвующий ему ASCII-код или 0 для дополнительного ASCII-кода.
3. Формирует значение для занесения в BIOS-буфер клавиатуры или изменяет соответствующие признаки статуса нажатия клавиш. Обработка ведется по схеме BIOS-обработчика.
4. Контролирует ввод в буфер не более 15 символов, отличных от Enter. Ввод в буфер ведется начиная с самой первой ячейки. При вводе 16-го символа, отличного от Enter, буфер автоматически очищается.
5. При вводе Enter программа завершает свою работу, восстанавливая прежнее значение вектора.

ЗАДАНИЕ N 9

Написать процесс, который обрабатывает прерывания от клавиатуры. Суть задачи заключается в следующем:

Задача должна анализировать коды нажатия клавиш, поступающие в BIOS-буфер клавиатуры, и, при включенном режиме Num Lock, преобразует коды цифровых клавиш в следующие символы псевдографики:

Цифра	Символ	Вид символа
1	179	│
2	180	+
3	191	┐
4	192	L
5	193	+
6	194	T
7	195	+
8	196	-
9	197	+
0	217	-

Завершение процесса выполняется нажатием клавиш <Alt/Q>.

ЗАДАНИЕ N 10

Написать процесс, который должен анализировать коды нажатия клавиш, поступающие в BIOS-буфер клавиатуры. При нажатии клавиш F1,...,F5 в режиме Num Lock задача должна очищать BIOS-буфер клавиатуры и помещать туда имена программ, которые автоматически затем вызываются на выполнение. Клавишам соответствуют следующие имена программ:

- F1 - P_F1<Enter>
- F5 - P_F5<Enter>

Завершение процесса выполняется нажатием клавиш <Alt/Q>.

ЗАДАНИЕ N 11

Написать процесс, который должен инициироваться по прерываниям от клавиатуры. Суть задачи заключается в следующем. Задача анализирует коды, поступающие в BIOS-буфер клавиатуры. В режиме Num Lock при нажатии клавиш F1,...,F5 программа заменяет их коды на последовательности символов вида:

- F1→main(){}
- F2→if(){}
- F3→while(){}
- F4→switch(){}
- F5→case :

Завершение процесса выполняется нажатием клавиш <Alt/Q>.

ЗАДАНИЕ N 12

Написать процесс, который должен инициироваться по прерываниям от клавиатуры. Суть задачи заключается в следующем. Процесс все коды нажатия клавиш, поступающие в BIOS-буфер клавиатуры, записывает в файл с именем, заданным при загрузке процесса в память, и сохраняет их в нём. Выгрузка процесса из памяти выполняется нажатием клавиш <Alt/Q>.

ЗАДАНИЕ N 13

Написать процесс, который должен инициироваться по прерываниям от таймера и клавиатуры. Суть задачи заключается в следующем. Задача с периодом T сек формирует в BIOS-буфере клавиатуры имя программы, которое интерпретируется затем command.com как вызов этой программы на выполнение. Это приводит к автоматическому запуску программы. Исходное имя программы и период T задаются при загрузке задачи в память. В процессе работы имя программы может быть изменено, если нажать одну из клавиш:

- F1→P_F1<Enter>
- F2→P_F2<Enter>
- F3→P_F3<Enter>

Выгрузка процесса из памяти выполняется нажатием клавиш <Alt/Q>.

ЗАДАНИЕ N 14

Написать процесс, который должен инициироваться по прерываниям от таймера. Суть задачи заключается в следующем. Задача с периодом T сек печатает на экране сообщения, которые считывает из файла, содержащего набор строк-сообщений. Когда файл заканчивается, то чтение повторяется с начала файла. Период T задается при загрузке задачи в память.

Выгрузка процесса из памяти выполняется нажатием клавиш <Alt/Q>.

ЗАДАНИЕ N 15

Написать процесс, который должен инициироваться по прерываниям от таймера. Суть задачи заключается в следующем. Задача в момент прерывания от таймера формирует в BIOS-буфере клавиатуры, когда он пуст, командную строку для command.com (строка заканчивается <Enter>). Этим имитируется ввод командной строки с клавиатуры. Строки считываются последовательно из файла. Когда файл заканчивается, то чтение повторяется с начала файла. Имя файла задается при загрузке задачи в память.

Завершение процесса выполняется нажатием клавиш <Alt/Q>.

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования "Самарский государственный аэрокосмический университет имени академика С.П. Королева (национальный исследовательский университет)"

Кафедра программных систем

А.В. Баландин

Лабораторная работа

Метод структурного анализа, моделирования и макетирования программной системы реального времени

Методические указания

Самара
2012

Содержание

Цель лабораторной работы

Целью выполнения лабораторной работы является освоение метода структурного анализа, моделирования и макетирования программной системы реального времени (ПСРВ) как программного агрегата системы компьютерного мониторинга (автоматизированной системы реального времени - АСРВ).

Программный агрегат реализуется на языке С. В качестве инструментальной среды разработки ПСРВ используется QNX Momentics IDE и целевая виртуальная машина под управлением ОСРВ QNX Neutrino (QNX 6).

Порядок выполнения лабораторной работы

Курсовая работа выполняется в соответствии с выданным вариантом задания. При выполнении работы необходимо реализовать этапы метода структурного анализа, моделирования и макетирования ПСРВ. В процессе выполнения проекта необходимо:

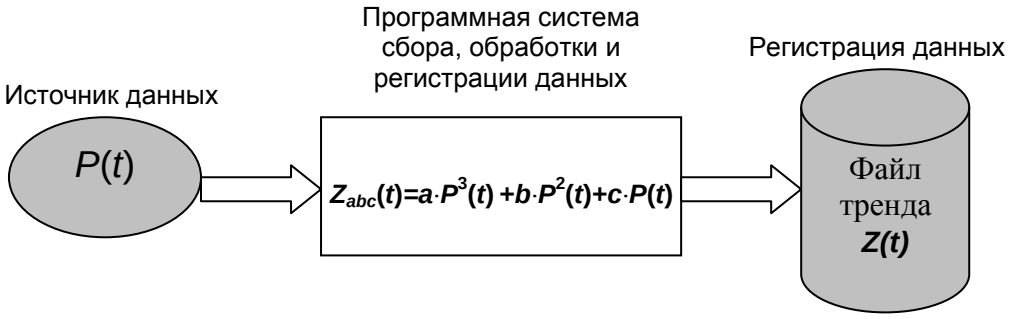
1. Дать обобщённое описание ПСРВ в виде программного агрегата, взаимодействующего с окружением, представленным абстрактными истоками и стоками данных.
2. Представить функциональную структуру ПСРВ в виде схемы асинхронных потоков данных (АСПД-схемы).
3. Представить процессную структуру ПСРВ в виде сети виртуальных потоковых машин (ВПМ-сети).
4. Представить агрегатную структуру ПСРВ в виде сети программных агрегатов (А-сети).
5. Описать программно-модульную структуру ПСРВ в файловой системе ОСРВ QNX целевой машины.

Задание
на разработку ПСРВ

Тема:

«Разработка программной системы мониторинга абстрактного физического объекта с заданными параметрами»

Необходимо разработать программную систему мониторинга абстрактного физического объекта O{$P(t)$}, $P(t)$ – изменяющийся во времени параметр источника данных:



Программная система в течение периода времени $[0,T]$ должна, начиная с t=0, периодически опрашивать источник данных $P(t)$, вычислять функцию:

$$Z_{abc}(t)=a\cdot P^3(t)+b\cdot P^2(t)+c\cdot P(t)$$

и записывать результат в файл тренда.

Изменение значений источника данных $P(t)$ моделируется функцией $\boldsymbol{F(1t\cdot i)}$, где, $\boldsymbol{i}$ – порядковый номер очередного вычисленного значения параметра $P(t_i)=\boldsymbol{F(1t\cdot i)}$ в момент интервального времени $\boldsymbol{t_i=1t\cdot i}\in[0,T]$, $\boldsymbol{i=0,1,2,\ \ldots}$; **1t** – значение единицы заданной базовой шкалы времени, выраженное в секундах.

Программная система на интервале времени $[0,T]$ периодически считывает значение источника данных $P(t)$ с заданным периодом $\boldsymbol{\Delta t=n\cdot(1t)}$ с и формирует датированное значение в виде пары:

$\langle \boldsymbol{z_j,t_j}\rangle$,

где:

$\boldsymbol{z_j}$ = $\boldsymbol{Z_{abc}(j\cdot\Delta t)}$, $\boldsymbol{t_j}$.= $\boldsymbol{j\cdot\Delta t}$, а $\boldsymbol{j=0,1,2,\ \ldots}$ – порядковый номер очередного считывания значения параметра $P(t)$ на интервале времени $[0,T]$. Результаты периодического считывания и обработки данных (датированные значения $\langle \boldsymbol{z_j,t_j}\rangle$) преобразуются в символьную строку формата:

"<z_j><\t><t_j>\n>"

и регистрируются в файле на внешнем носителе.

Источник данных и программная система должны быть синхронизированы по моменту времени $t=0$ (должны начать действовать "одновременно").

При $t=T$ работа программной системы должна завершиться (файл тренда закрывается, все процессы терминируются).

Используя файл тренда, построить график $Z_{abc}(j\cdot\Delta t)$, загрузив, например, содержимое файла в MS EXCEL, или используя другие средства построения графиков.

Варианты заданий

Номер варианта	Функция параметра $P(t)$	Величина единицы базовой временно́й шкалы, 1t (сек)	Интервал опроса параметра в количествах единиц базовой шкалы n	a	b	c	Интервал работы, T (сек
1.	$F_{P1}=2.8e^{-1.5t}\sin(2.5\pi t)+1$	0.03	5	0.1	0.2	0.3	45
2.	$F_{P2}=100e^{-t}\cos(0.1t+100)$	0.02		8.5	12.0	11.0	54
3.	$F_{P3}=100t^2e^{-0.1t}$	0.05		11.5	12.0	8.0	67
4.	$F_{P1}=100t^2e^{-0.1t}$	0.05	6	9.5	10.5	11.0	76
5.	$F_{P2}=100e^{-t}\cos(0.1t+100)$	0.03		8.0	9.0	10.0	53
6.	$F_{P3}=100e^{-t}\sin(0.1t+100)$	0.08		7.0	8.0	9.0	61
7.	$F_{P1}=100e^{-t}\cos(0.1t+100)$	0.09	3	6.0	7.0	8.0	59
8.	$F_{P2}=100t^2e^{-0.1t}$	0.05		5.0	6.0	7.0	34
9.	$F_{P3}=100e^{-t}\sin(0.1t+100)$	0.17		4.0	5.0	6.0	39
10.	$F_{P1}=200t^2e^{-0.2t}$	0.12	4	3.0	4.0	5.0	76
11.	$F_{P2}=100t^2e^{-0.1t}$	0.09		2.0	3.0	4.0	65
12.	$F_{P3}=200e^{-0.1t}\sin(0.5t+100)$	0.18		3.0	2.0	1.0	43
13.	$F_{P1}=200t^2e^{-0.2t}$	0.16	5	1.0	2.0	3.0	44
14.	$F_{P2}=100e^{-t}\cos(0.1t+100)$	0.08		0.7	0.8	0.9	47
15.	$F_{P3}=100t^2e^{-0.1t}$	0.22		0.4	0.5	0.6	51
16.	$F_{P1}=250t^2e^{-0.25t}$	0.18	3	0.1	0.2	0.3	52
17.	$F_{P2}=120e^{-t}\cos(0.2t+150)$	0.12		8.5	12.0	11.0	66
18.	$F_{P3}=130t^2e^{-0.15t}$	0.26		11.5	12.0	8.0	55
19.	$F_{P1}=120e^{-t}\cos(0.2t+150)$	0.2	2	4.0	5.0	6.0	42
20.	$F_{P2}=250t^2e^{-0.25t}$	0.12		5.0	6.0	7.0	63
21.	$F_{P3}=130t^2e^{-0.15t}$	0.33		6.0	7.0	8.0	62

22.	$F_{P1}=130t^2e^{-0.15t}$	0.23	7	9.5	10.5	11.0	58
23.	$F_{P2}=250t^2e^{-0.25t}$	0.13		8.0	9.0	10.0	33
24.	$F_{P3}=100t^2e^{-0.1t}$	0.39		8.0	9.0	10.0	36
25.	$F_{P1}=200e^{-0.1t}\sin(0.5t+100)$	0.25	4	7.0	8.0	9.0	72
26.	$F_{P2}=130t^2e^{-0.15t}$	0.15		0.1	0.2	0.3	73
27.	$F_{P3}=100e^{-t}\cos(0.1t+100)$	0.43		8.5	12.0	11.0	37
28.	$F_{P1}=100t^2e^{-0.1t}$	0.23	6	9.5	10.5	11.0	66
29.	$F_{P2}=200e^{-0.1t}\sin(0.5t+100)$	0.16		8.0	9.0	10.0	77
30.	$F_{P3}=130t^2e^{-0.15t}$	0.5		7.0	8.0	9.0	49

Оформление записки

Записка по лабораторной работе должна оформляться в соответствии со стандартом СГАУ (СТО СГАУ 02068410-004-2007 - Общие требования к учебным текстовым документам) к построению, изложению и оформлению учебных текстовых документов и должна содержать:

1. титульный лист установленного образца;
2. задание (вариант);
3. реферат;
4. содержание;
5. определения, обозначения и сокращения;
6. введение (краткая характеристика темы задания и порядка выполнения лабораторной работы);
7. Основная часть:

- представление ПСРВ в виде программного агрегата,
- описание АСПД_схемы,
- описание ВПМ-сети,
- описание А-сети,
- описание программно-модульной структуры ПСРВ,
- результаты работы ПСРВ.

8. Заключение;
9. Список использованных источников (при наличии);
10. Приложение 1. Программно-модульная структура ПСРВ
11. Приложение 2. Описание исходных текстов программных модулей.

12.Приложение 3. График тренда $Z_{abc}(t)$.

Работа ПСРВ, файл и график тренда $Z_{abc}(t)$ демонстрируются преподавателю.

Методические указания по разработке программной системы

Общая структура СРВ

В самом общем виде структуру системы реального времени (СРВ), в состав которой входит АСРВ, можно представить так, как показано на Рисунок 1. На рисунке представлены четыре компоненты СРВ: *объект управления* (ОУ), *субъект управления* (СУ), *АСРВ* и *хранилище данных*. Объект управления,

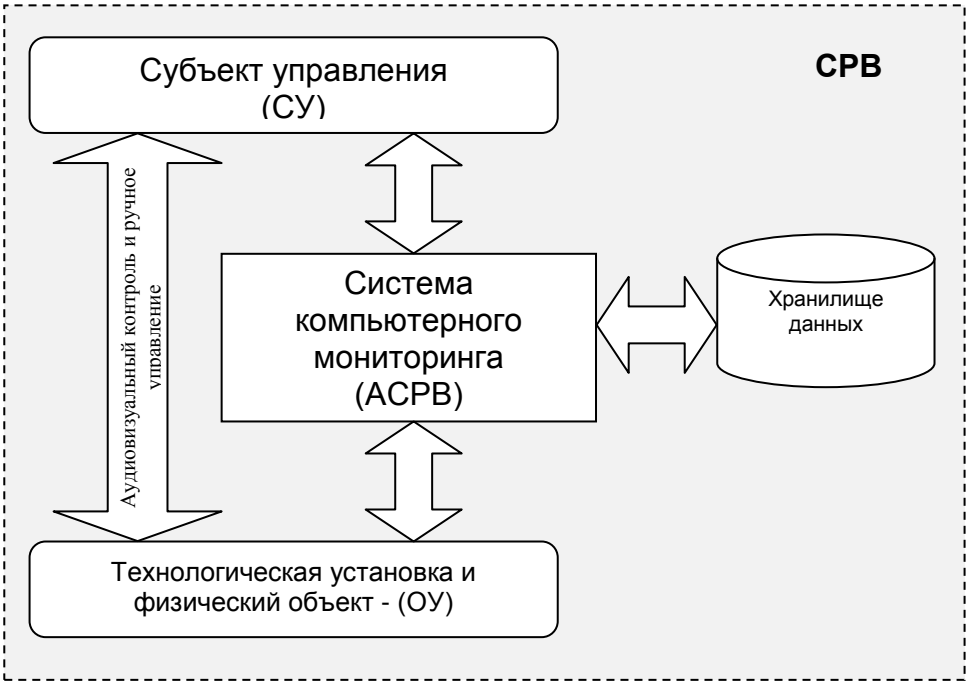


Рисунок 1 Общая структура системы реального времени

субъект управления и хранилище данных составляют *окружение* АСРВ, с которым АСРВ взаимодействует в режиме реального времени.

Объект управления (ОУ) – это абстракция, специфицирующая технологическую установку и физический объект как единое целое, предназначенное для реализации технологического процесса. В общем случае ОУ может иметь сложную топологию и пространственную распределённость. Объект управления имеет множество физических параметров, характеризующих его состояние во времени, и множество параметров регулирования для управления состоянием ОУ. Для снятия (измерения) значений физических параметров ОУ оснащается датчиками. Для осуществления регулирующих (управляющих) воздействий на ОУ он оснащается регуляторами (исполнительными

механизмами). Кроме того, ОУ может оснащаться устройствами отображения значений параметров состояния и значений регулировок, предназначенными непосредственно человеку-оператору для организации его взаимодействия с объектом управления (аудиовизуальный контроль состояния и ручное регулирование).

Таким образом, объект управления в общем случае представляется как распределённая пространственно-топологическая структура устройств сбора, отображения данных и устройств регулирования физических параметров на объекте управления.

Субъект управления (СУ) – это абстракция, специфицирующая функцию оператора (в общем случае - группы операторов), управляющего технологическим процессом на ОУ. Функционирование субъекта управления заключается в мониторинге отображаемых значений физических параметров и управлении состоянием ОУ непосредственно или через АСРВ путём регулирования физических параметров объекта управления посредством исполнительных механизмов. Субъект управления может так же, как и объект управления, иметь собственную распределённую пространственно-топологическую структуру, не совпадающую с пространственно-топологической структурой ОУ.

Хранилище данных – это абстракция, специфицирующая память, предназначенную для накопления и хранения результатов мониторинга состояния и регулирования ОУ, а также результатов их обработки. В качестве таких результатов могут выступать тренды измеряемых параметров состояния, значения регулировок параметров технологической установки и физического объекта, режимы управления ОУ и т.п. Кроме того, хранилище данных предназначено и для хранения базы данных с системной информацией, описывающей непосредственно АСРВ как предметную область. Например, топологию объекта и субъекта управления и размещение устройств комплекса технических средств и распределённого вычислительного комплекса, топологию размещения и типы датчиков и регуляторов и т.д.

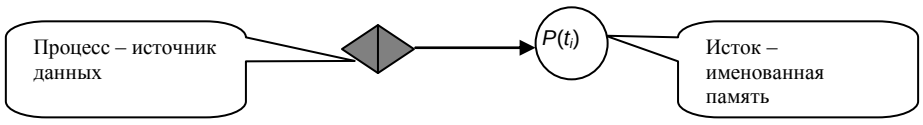
Моделирование ПСРВ в виде программного агрегата

В соответствии с методом структурного анализа, моделирования и макетирования ПСРВ описание разрабатываемой программной системы начинается с представления ПСРВ в виде абстрактного программного агрегата, взаимодействующего со своим окружением, представленного абстрактными истоками и стоками данных. Истоки и стоки данных определяют образ объекта и субъекта управления, а также хранилища данных по отношению к ПСРВ в рамках системы реального времени (программный интерфейс взаимодействия с внешним окружением). С истоками и стоками со стороны внешнего окружения связаны внешние процессы, порождающие и поглощающие данные. В лабораторной работе на этом этапе ПСРВ будет представляться в следующем виде:

На этом этапе необходимо присвоить имена истокам и стокам и специфицировать их структуры данных. Внешний процесс, порождающий данные, моделирует абстрактный источник данных внешней среды (абстрактный датчик). Внешний процесс, поглощающий данные, моделирует абстрактный поглотитель данных, которые уходят во внешнюю среду (например, отображаются на экран оператору или заносятся в файл). Таким образом, ответственность за поставку данных или их потребление в том или ином виде несут внешние процессы.

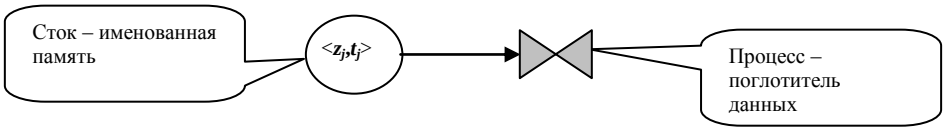
Моделирование истоков

В качестве программной модели источника данных (истока) следует использовать глобальную (именованную) память, в которую помещаются данные, поступающие из окружения программного агрегата (внешнего процесса), моделирующие изменение во времени значения параметра $P(t_i)=\boldsymbol{F(1t\cdot i)}$. В АСПД-схеме это изображается в виде:



Моделирование стоков

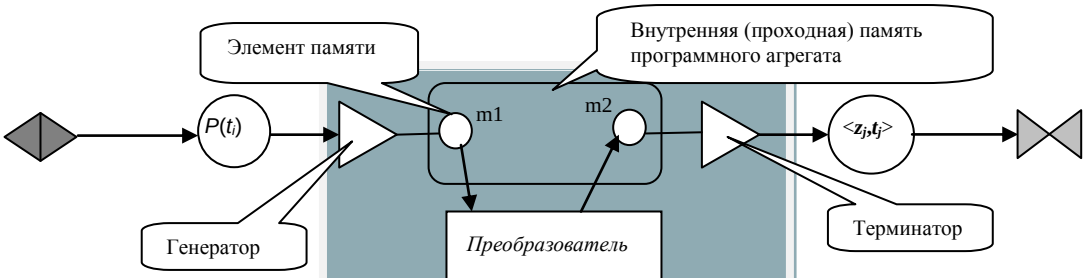
В качестве программной модели стоков следует использовать именованную память, в которую помещаются вычисляемые программным агрегатом датированные значения $<z_j,t_j>$. Внешней процесс извлекает датированные значения и заносит их в файл тренда. В АСПД-схеме это изображается в виде:



Структурный анализ программной системы на функциональном уровне

После того, как внешнее окружение программного агрегата специфицировано, осуществляется его структурный анализ и описание на функциональном уровне. Цель структурного анализа на функциональном уровне заключается в описании функциональной структуры программного агрегата и его функционирования как преобразователя асинхронных потоков данных. Функциональная структура программного агрегата выражается в виде схемы управления асинхронными потоками данных (АСПД-схемы). Для изображения АСПД-схемы используются графические конструктивные элементы схемы асинхронных потоков данных.

Доступ к истокам и стокам со стороны программного агрегата в его структуре описывается конструктивными элементами АСПД-схемы, которые называются соответственно *генераторы* и *терминаторы*. С каждым истоком связывается генератор, а со стоком – терминатор. В результате в нашем случае на функциональном уровне программный агрегат примет вид:



Кроме генератора и терминатора в структуре программного агрегата появляются элементы внутренней (проходной) памяти – m1 и m2, и *Преобразователь* потока данных.

Генератор периодически с интервалом Δt берёт данное из истока, формирует датированное данное $<d_j,t_j>$ и помещает его в элемент памяти m1, $t_j = j \cdot \Delta t$, если он является пустым. После чего m1 становится занятым. Когда m1 занят (содержит данное), а m2 свободен (пустой), срабатывает *Преобразователь*. Он извлекает $<d_j,t_j>$ из m1, вычисляет значение $<z_j,t_j>$ и помещает его в m2. Элемент памяти m1 становится пустым, а m2 – занятым. Когда m2 занят, а сток пустой, срабатывает *Терминатор*. Он извлекает $<z_j,t_j>$ из m2 и помещает его в сток, после чего внешний процесс может извлечь его из стока и отправить в файл тренда. Описанный процесс функционирования программного агрегата циклически повторяется.

Моделирование программной системы на процессном уровне

Цель моделирования ПСРВ на процессном уровне заключается в определении процессно-нитевой структуры программного агрегата.

На процессном уровне программный агрегат моделируется в виде сети виртуальных потоковых машин (ВПМ-сеть). Для этого АСПД-схема разбивается на непересекающиеся фрагменты, которые затем используются для спецификации нитевой структуры ВПМ. В нашем случае (простая АСПД-схема) нет смысла выделять более одного фрагмента. Поэтому на процессном уровне будет специфицирована всего одна ВПМ. Это означает, что программный агрегат будет состоять из одного процесса (виртуальной потоковой машины), в составе которого будет три нити:

1. Нить-генератор.
2. Нить-преобразователь.
3. Нить терминатор.

Внутренняя (проходная) память ВПМ будет состоять из двух элементов памяти m1 и m2 типа «гнездо». Тип «гнездо» означает, что когда память занята, она доступна для чтения, но недоступна для записи. Когда память свободна, она доступна для записи, но недоступна для чтения. Это означает, что доступ к памяти со стороны нитей необходимо синхронизировать.

Моделирование программной системы на агрегатном уровне

Цель моделирования программной системы на агрегатном уровне заключается в описании программной системы как совокупности взаимодействующих между собой программных агрегатов, обрабатывающих потоки данных, состоящих из виртуальных потоковых машин. Такое описание выражается в виде сети программных агрегатов (А-сети). Виртуальные потоковые машины, находящиеся в одном агрегате А-сети, должны будут грузиться как процессы в один и тот же узел компьютерной сети (локальной сети). Формально А-сеть формируется на основе ВПМ-сети путём её разбиения на непересекающиеся фрагменты, включающие в себя виртуальные потоковые машины ранее построенной ВПМ-сети. В нашем случае из-за простоты программной системы и ВПМ-сеть состоит всего из одной ВПМ. Поэтому А-сеть будет предельно простой и состоять из одного агрегата, который будет грузиться только в один узел компьютерной сети (вырожденный случай распределённой ПСРВ). Так как в ОС QNX доступ к именованной памяти одного узла не возможен из другого узла компьютерной сети, то и соответствующие внешние процессы истока и стока также должны будут грузиться в этот же узел.

Макетирование программной системы на программно-модульном уровне

Цель макетирования ПСРВ на программно-модульном уровне заключается в описании файловой структуры программной системы и описании сценария её загрузки. Иными словами, описываются все файлы с программными модулями,

на базе которых загружаются виртуальные потоковые машины, их имена и место нахождения в файловой системе компьютерной сети, файлы с данными специфицирующими структуру ВПМ-сети и А-сети, а также файл, специфицирующий распределение и загрузку агрегатов А-сети по узлам компьютерной сети. Параметрические файлы, специфицирующие параметры настройки загружаемых процессов и т.п. Кроме того, описывается порядок загрузки программной системы в компьютерную сеть как в техническую среду системы реального времени.

Литература

1. Баландин А.В. Средства разработки программного обеспечения систем реального времени [Электронный документ формата CHM]: учебное пособие: В 4 ч. Ч.1. Язык программирования С. - Самар. гос. аэрокосм. ун-т. Кафедра программных систем. Самара, 2012. 14 с.
2. Баландин А.В. Средства разработки программного обеспечения систем реального времени [Электронный документ формата CHM]: учебное пособие: В 4 ч. Ч.2. Программирование параллельных процессов в ОСРВ QNX Neutrino. - Самар. гос. аэрокосм. ун-т. Кафедра программных систем. Самара, 2012. 14 с.
3. Баландин А.В. Средства разработки программного обеспечения систем реального времени [Электронный документ формата CHM]: учебное пособие: В 4 ч. Ч.3. Программирование нитей в ОСРВ QNX Neutrino. - Самар. гос. аэрокосм. ун-т. Кафедра программных систем. Самара, 2012. 14 с.

ВОПРОСЫ
для проверки знаний по дисциплине
Технологии промышленного программирования

Работа в ОС QNX

1. Укажите имя пользователя, которое автоматически регистрируется при установке ОС QNX.

А) root

Б) admin

В) supervisor

2. Каталог с каким именем автоматически становится текущим при входе в ОС QNX под именем root?

А) fs

Б) root

В) home

3. Что должен указать пользователь при входе в ОС QNX?

А) пароль

Б) имя пользователя и пароль

В) имя группы пользователей и пароль

4. Почему в ОС QNX вместо понятия "файловая система" используется понятие "файловое пространство"?

А) Это одно и то же

Б) Собственная терминология ОС QNX

В) Файловое пространство объединяет отдельные файловые системы

5. Какую роль в файловом пространстве ОС QNX играют файлы устройств?

А) На уровне организации ввода/вывода логически ставят устройства в один ряд с обычными файлами

Б) Играют роль буфера доступа к устройству

В) Файлы, которые связывают с устройствами для приема или передачи данных

6. Какую роль в базовой структуре каталогов играет каталог /fs?

А) Содержит стандартные файловые системы

Б) Предназначен для автоматического монтирования файловых систем разделов жесткого диска

В) Текущий каталог пользователя root

7. Каких владельцев может иметь файл?

А) Владельца-пользователя

Б) Владельца-группу

В) Владельца-пользователя и владельца-группу

8. Чьи права доступа контролируются ОС QNX при открытии файла?

А) Владельца-пользователя

Б) Владельца-пользователя, владельца-группы и остальных пользователей

В) Владельца-пользователя и владельца-группы

9. Как завершится выполнение функции open("file.dat", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR), если в текущем каталоге существует файл с именем file.dat?

А) Новый файл заменит существующий (сохранив дескриптор старого)

Б) Уничтожит старый и создаст новый файл с тем же именем

В) Выдаст сообщение об ошибке

10. Какие и чьи права доступа определены при выполнении функции open("file.dat", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)?

А) Чтение и запись для пользователей, входящих в группу-владельца файла file.dat

Б) Чтение и запись для пользователей файла file.dat

В) Чтение и запись для пользователя-владельца файла file.dat

11. Что понимается под процессом в ОС QNX?

А) Функция-нить в защищенном адресном пространстве, параллельно выполняемая с другими нитями

Б) Контейнер параллельно выполняемых нитей в едином защищенном адресном пространстве

В) Один из контекстов, развивающихся на едином (как правило, реентерабельном) исполняемом модуле

12. Процессы каких типов различают в ОС QNX?

А) Системные процессы, демоны, прикладные процессы

Б) Системные процессы, прикладные процессы

В) Системные процессы, демоны

13.Может ли один процесс осуществить прямой доступ в адресное пространство другого процесса?

А) Да

Б) Нет

В) Да, если между ними установлено соединение

14.Какой процесс будет считаться родительским при запуске процесса функцией `system()`?

А) процесс, выполнивший функцию `system()`

Б) системный процесс ядра

В) командный интерпретатор `shell`

15.Какова особенность запуска дочернего процесса с помощью функций семейства exec*()?

А) Родительский процесс завершает выполнение, а дочерний процесс создается с ID родительского процесса

Б) Родительский процесс порождает дочерний процесс и ждет его завершения

В) Родительский процесс порождает дочерний процесс и выполняется параллельно с ним

16.Какова особенность запуска дочернего процесса с помощью функции spawn()?

А) Родительский процесс завершает выполнение, а дочерний процесс создается с ID родительского процесса

Б) Родительский процесс порождает дочерний процесс и выполняется параллельно с ним

В) Родительский процесс порождает дочерний процесс и ждет его завершения

17.Какова особенность запуска дочернего процесса с помощью функции fork()?

А) Родительский процесс завершает выполнение, а дочерний процесс создается с ID родительского процесса

Б) Родительский процесс порождает свою копию и ждет завершения дочернего процесса

В) Порождается процесс, являющийся копией родительского процесса, который стартует с команды, следующей непосредственно за вызовом fork(), и выполняется параллельно с родительским процессом

18.Какова особенность запуска дочернего процесса с помощью функции vfork()?

А) Родительский процесс завершает выполнение, а дочерний процесс создается с ID родительского процесса.

Б) Родительского процесс создает дочерний процесс как копию себя в своем адресном пространстве (который стартует непосредственно за вызовом vfork()) и ждет завершения дочернего процесса.

В) Порождается процесс, являющийся копией родительского процесса (который стартует непосредственно за вызовом vfork()) и выполняется параллельно с родительским процессом?

19. Какая функция должна быть выполнена процессом-клиентом для организации взаимодействия с процессом-сервером?

А) ConnectAttach()

Б) ChannalCreate()

В) MsgSend()

20. Какая функция должна быть выполнена процессом-сервером для организации взаимодействия с процессом-клиентом?

А) ConnectAttach()

Б) ChannalCreate()

В) MsgReceive()

21. В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер был в состоянии "Готов"?

А) "Готов"

Б) Reply-блокированном

В) В Send-блокированном

22. В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер был в Receive-блокированном состоянии?

А) "Готов"

Б) Reply-блокированном

В) В Send-блокированном

23. В каком состоянии окажется процесс-сервер, выполнивший функцию MsgReceive(), если процесс-клиент был в Send-блокированном состоянии?

А) "Готов"

Б) Reply-блокированном

В) В Send-блокированном

24. В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер был в Receive-блокированном состоянии?

А) "Готов"

Б) Reply-блокированном

В) В Send-блокированном

25. В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер выполнил функцию MsgError()?

А) "Готов"

Б) Reply-блокированном

В) В Send-блокированном

26. В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер выполнил функцию MsgRead()?

А) "Готов"

Б) Reply-блокированном

В) В Send-блокированном

27. В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер выполнил функцию MsgReply()?

А) В Send-блокированном

Б) Reply-блокированном

В) "Готов"

28. В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер выполнил функцию MsgWrite()?

А) "Готов"

Б) Reply-блокированном

В) В Send-блокированном

29. Что обеспечивает механизм векторов ввода/вывода при обмене сообщением между клиентом и сервером?

А) Прием/передачу сообщения, в виде совокупности отдельных частей.

Б) Возникновение программного прерывания при выполнении клиентом функции MsgSend().

В) Посылку ядром ОС сигнала серверу при выполнении клиентом функции MsgSend().

Работа в сети

1) Процесс `P1` на узле `comp1` и процесс `P2` на узле `comp2` выполняют функцию `nd1=netmgr_strtond("/net/comp3",NULL)` и `nd2=netmgr_strtond("/net/comp3",NULL)`. Как, в общем случае, соотносятся между собой значения `nd1` и `nd2`?

   a) `nd1 = nd2`, всегда.

   b) `nd1 < nd2`, если процесс `P1` выполнил функцию раньше, чем `P2`.

c) `nd1 ? nd2`, соотношение не определено.

2) Что необходимо выполнить удаленному процессу-клиенту с результатом выполнения функции `nd=netmgr_strtond("/net/comp",NULL)`, используемого для организации соединения с процессом-сервером на удаленном узле с именем `comp`?

   a) сохранить `nd` в статической памяти и использовать его каждый раз при очередной необходимости создания соединения с процессом-сервером на удаленном узле с именем `comp`.

   b) немедленно использовать `nd` в вызове `ConnectAttach()` для создания соединения с процессом-сервером на удаленном узле с именем `comp`.

   c) использовать `nd` в вызове `MsgSend()` при посылке сообщения процессу-серверу.

3) Что возвращает функция `ConnectAttach()`, если соединение устанавливается с процессом на удаленном узле?

   a) признак успешного соединения.

   b) признак успешного соединения, если `nd`, `pid`, `chid` корректны.

   c) всегда 0.

4) На каком этапе может быть выявлена некорректность соединения с сервером на удаленном узле?

   a) сразу при выполнении `ConnectAttach()`.

   b) при выполнении `ConnectAttach()` или, затем, при выполнении `MsgSend()`.

   c) только при выполнении MsgSend().

5) Клиент посылает 120 Кб данных серверу на удаленном узле. Какой предельный объем данных будет получен сервером при выполнении `MsgReceive()`?

   a) 120 Кб, если указан объем буфера приема не менее 120 Кб.

   b) объем буфера приема, если он менее 120 Кб.

   c) не более 8 Кб.

6) Являются ли вызовы `MsgReply(),MsgRead(),MsgWrite()` блокирующими для процесса-сервера?

   a) да.

   b) да, при взаимодействии с клиентом на удаленном узле.

   c) только `MsgRead(),MsgWrite()`.

7) Родительский процесс на узле с именем `compA` с помощью функции `spawn()` запустил дочерний процесс на удаленном узле с именем `compB`. Какие действия могут быть выполнены дочерним процессом для установления соединения с

родительским процессом, полагая, что родительский процесс передал дочернему процессу необходимые параметры при запуске?

a) определить `pid=getppid()`, выполнить `chroot("/net/compB/")`, определить `nd=netmgr_strtond("/net/compA",NULL)`, получить `chid` как аргумент функции `spawn()` и выполнить функцию соединения `ConnectAttach()`.

b) получить `pid` и `chid` как аргументы функции `spawn()`, определить `nd=netmgr_strtond("/net/compA",NULL)`, выполнить `chroot("/net/compB/")` и выполнить функцию соединения `ConnectAttach()`.

c) получить `pid` и `chid` как аргументы функции `spawn()`, выполнить `chroot("/net/compB/")`, определить дескриптор узла родительского процесса - `nd=netmgr_strtond("/net/compA",NULL)`, и выполнить функцию соединения `ConnectAttach()`.

Запуск нитей

8) Оказывает ли влияние изменение значений атрибутной записи нити, используемой для определения свойств нити, на изменение свойств уже запущенной на выполнение нити?
   a) нет.
   b) да.
   c) да, но только при изменении приоритета.
9) Должна ли обладать свойством реентерабельности функция, если она используется для запуска более чем одной нити?
   a) нет.
   b) да.
   c) да, если функция содержит в своем теле изменяемые переменные.
10) Какой вид имеет объявление функции, используемой для запуска нитей?
   a) `void func(void*).`
   b) `void* func(void*).`
   c) `void* func(void).`
11) Какая нить процесса называется обособленной?
   a) если для нити установлено свойство `PTHREAD_CREATE_JOINABLE`.
   b) `если для нити установлено свойство PTHREAD_CREATE_DETACHED.`
   c) нить, которая не взаимодействует с другими нитями.
12) Может ли нить `main()` быть обособленной?
   a) нет.
   b) да.
   c) да, если ей установлено свойство `PTHREAD_CREATE_DETACHED`.
13) Если текущая нить выполнила функцию передачи процессора `sched_yield()`, то какая из готовых к выполнению нитей начнет выполняться?
   a) нить с более высоким приоритетом.
   b) нить с более низким приоритетом.
   c) нить с тем же приоритетом.

14) Могут ли в приложении быть запущены нити с разными дисциплинами диспетчеризации?
    a) да.
    b) нет.
    c) да, в разных процессах.
15) Может ли процесс динамически изменять величину кванта времени диспетчеризации процессора?
    a) да.
    b) нет.
    c) да, если это суперпользовательский процесс.
16) Если завершается квант процессорного времени текущей нити с приоритетом `pr`, то ожидающей нити с каким приоритетом будет предоставлен очередной квант процессорного времени?
    a) нити с приоритетом равным pr.
    b) нити с приоритетом большим или равным `pr`.
    c) нити с приоритетом меньшим или равным `pr`.
17) Что понимается под инверсией приоритетов нитей?
    a) приобретение сервером приоритета клиента при получении его сообщения.
    b) опосредованное изменение приоритета клиента за счет приоритета сервера.
    c) изменение сервером своего приоритета в процессе выполнения.
18) Что понимается под наследованием приоритетов нитей?
    a) приобретение сервером приоритета клиента.
    b) опосредованное изменение приоритета клиента за счет приоритета сервера.
    c) результат инверсии приоритета.

Синхронизация нитей

19) Какие системные вызовы (функции) обеспечивают синхронизацию продолжения выполнения одной нити с моментом завершения существования другой нити?
    a) pthread_join().
    b) `barrier_wait()`.
    c) `pthread_join()`,`barrier_wait()`.
20) Какую синхронизацию нитей обеспечивает метод барьеров?
    a) синхронизирует продолжение выполнения заданного количества нитей с моментом завершения некоторой нити.
    b) обеспечивает одновременность продолжения выполнения для заданного количества нитей, достигших барьера.
    c) обеспечивает одновременность завершения выполнения для заданного количества нитей, достигших барьера.
21) Если для управления доступом к некоторой области памяти создан мутекс, то что выполнит ядро по отношению к нити, которая попытается осуществить доступ к этой области памяти, минуя мутекс?
    a) ничего.
    b) она будет терминирована ядром.
    c) будет заблокирована, пока имеется нить, захватившая мутекс и осуществляющая исключительный доступ к области памяти.
22) Что произойдет с нитью, если она по отношению к блокировке чтения/записи выполнит функцию `pthread_rwlock_trywrlock()` или `pthread_rwlock_tryrdlock()`, а действие недопустимо?
    a) будет блокирована.
    b) информируется о недопустимости действия и продолжает выполнение.
    c) она блокируется и ей возвращается ошибка.
23) Что произойдет с нитью, если она по отношению к блокировке чтения/записи выполнит функцию `pthread_rwlock_wrlock()` или `pthread_rwlock_rdlock()`, а действие недопустимо?
    a) будет блокирована.
    b) информируется о недопустимости действия и продолжает выполнение.
    c) она блокируется и ей возвращается ошибка.

24) Если нить стремится к исключительному доступу к некоторому ресурсу, но может его использовать только при выполнении определенных условий, то каким из приведенных методов синхронизации можно воспользоваться?

a) создать барьер, блокироваться у барьера и ждать, когда прибудет нить, отвечающая за выполнение условия доступа к ресурсу.

b) создать мутекс и условную переменную, захватить мутекс и организовать цикл ожидания событий, изменяющих условие. Проверять выполнение условия и либо вновь ждать, либо использовать ресурс и освободить мутекс.

c) создать мутекс и флаг выполнения условия. Организовать цикл ожидания готовности флага. Когда флаг готов, захватить мутекс и использовать ресурс, затем освободить мутекс и сбросить флаг.

Сигналы

25) Если процессу приходит сигнал, а обработчик сигнала не установлен, то как это отразится на процессе?
   a) процесс терминируется.
   b) сигнал будет проигнорирован.
   c) выполнится действие, предписанное сигналу по умолчанию.

26) Если сигнал генерируется функцией `kill(pid, sig)`, где `pid>0`, то кому адресуется сигнал?
   a) процессу с указанным значением `pid`.
   b) всем процессам, входящим в группу, которой принадлежит процесс, пославший сигнал.
   c) процессам группы с `GID=-pid`.

27) Если сигнал генерируется функцией `kill(pid, sig)`, где `pid=0`, то кому адресуется сигнал?
   a) процессу с указанным `pid`.
   b) всем процессам, входящим в группу, которой принадлежит процесс, пославший сигнал.
   c) всем процессам группы с `GID=-pid`.

28) Если сигнал генерируется функцией `kill(pid, sig)`, где `pid<0`, то кому адресуется сигнал?
   a) процессу с указанным `pid`.
   b) всем процессам, входящим в группу, которой принадлежит процесс, пославший сигнал.
   c) всем процессам группы с GID=-pid.

29) Какие функции позволяют установить обработчик сигналов так, чтобы приходящие сигналы ставились бы в очередь на обработку?
   a) `signal()`.
   b) sigaction().
   c) `signal()`, `sigaction()`.

30) Какие функции позволяют установить обработчик сигналов и обеспечивают управление маскированием сигналов?

a) `signal()`.

b) sigaction().

c) `signal()`, `sigaction()`.

31) Какие функции позволяют установить обработчик сигналов, который позволяет получить номер сигнала, код сигнала и значение, связанное с сигналом?

a) `signal()`.

b) sigaction().

c) `signal()`, `sigaction()`.

32) Какие функции позволяют послать сигнал адресату, находящемуся на удаленном узле локальной сети?

a) `kill()`.

b) SignalKill().

c) `kill()`, `SignalKill()`.

33) Какие функции позволяют послать сигнал, адресуемый конкретной нити?

a) `kill()`.

b) SignalKill().

c) `kill()`, `SignalKill()`.

34) Какие возможности работы с сигналами предоставляет нити системный вызов (функция) `SignalWaitinfo()`?

a) получить информацию о сигнале, поступившем процессу, содержащему эту нить.

b) перейти в состояние ожидания прихода сигналов, для которых нить установила маску блокирования.

c) получить информацию о количестве поступивших процессу сигналов, для которых нить установила маску блокирования.

Таймеры

35) Как в ОСРВ QNX6 задается абсолютное значение реального времени?

a) указывается количество секунд, прошедших с 00 час 00мин 00 сек 1 января 1970 года по Гринвичу.

b) указывается количество секунд и наносекунд, прошедших с 00 час 00мин 00 сек 1 января 1970 года по Гринвичу.

c) указывается количество наносекунд, прошедших с 00 час 00мин 00 сек 1 января 1970 года по Гринвичу.

36) Как в ОСРВ QNX6 задается значение интервала реального времени?

a) количеством секунд.

b) количеством секунд и наносекунд.

c) количеством наносекунд.

37) Какой тип уведомления таймера определяет вызов `SIGEV_PULSE_INIT()`?

a) импульс.

b) сигнал.

c) запустить нить.

38) Какой тип уведомления таймера определяет вызов `SIGEV_SIGNAL_INIT()`?

a) импульс.

b) сигнал.

c) запустить нить.

39) Какой тип уведомления таймера определяет вызов `SIGEV_SIGNAL_CODE_INIT()`?

a) импульс.

b) сигнал.

c) запустить нить.

40) Какой тип уведомления таймера определяет вызов `SIGEV_SIGNAL_THREAD_INIT()`?

a) импульс.

b) сигнал.

c) запустить нить.

41) Какой тип уведомления таймера определяет вызов `SIGEV_THREAD_INIT()`?

a) импульс.

b) сигнал.

c) запустить нить.

42) Какого типа таймер запускается, если в вызове `timer_settime()` время определяется переменной
`struct itimerspec timer;`
поля которой проинициализированы как
```
timer.it_value.tv_sec=1;
timer.it_value.tv_nsec=0;
timer.it_interval.tv_sec=1;
timer.it_interval.tv_nsec=0;
```
?

a) абсолютный однократный.

b) относительный однократный.

c) относительный периодический.

43) Какую роль играет функция `Timer_Timeout()`?

a) планирует приход процессу сигнала от таймера, спустя указанный интервал времени.

b) позволяет выйти из указанного состояния блокировки через указанный интервал времени.

c) позволяет послать импульса указанному адресату через указанный интервал времени.

44) Какова реакция на тайм-аут ядра для REPLY-блокированного состояния при передаче сообщения, если при создании канала сервером был установлен флаг `_NTO_CHF_UNBLOCK`?

a) клиент будет немедленно разблокирован, сервер не получает об этом никакого уведомления.

b) клиент будет немедленно разблокирован, сервер получает об этом уведомление в виде импульса.

c) клиент остается заблокированным, сервер получает об этом уведомление в виде импульса.

Прерывания

45) Какие функции используются для управления запросами прерывания на уровне программируемого контроллера прерываний - PIC?
   a) `InterruptLock()`, `InterruptUnlock()`.
   b) InterruptMask(), InterruptUnmask().
   c) `ThreadCtl()`, `InterruptWait()`.

46) Какие функции используются для управления запросами прерывания на уровне процессора?
   a) InterruptLock(), InterruptUnlock().
   b) `InterruptMask()`, `InterruptUnmask()`.
   c) `ThreadCtl()`, `InterruptWait()`.

47) Какая из функций подключения к прерыванию предполагает явное определение обработчика прерываний?
   a) InterruptAttach().
   b) `InterruptAttachEvent()`.
   c) `InterruptAttach()` и `InterruptAttachEvent()`.

48) Какая из функций подключения к прерыванию не предполагает явного определения обработчика прерываний?
   a) `InterruptAttach()`.
   b) InterruptAttachEvent().
   c) `InterruptAttach()` и `InterruptAttachEvent()`.

Структурный анализ, моделирование и макетирование ПСРВ

49) Чем определяется концептуальная модель распределенной ПСРВ?
   a) Тройкой <***O***,***C***,***S***> и свойствами {r_1, r_2, r_3}.
   b) Тройкой <*O*,*C*,*S*> и свойствами {r1, r2, r3, r4, r5, r6}.
   c) Двойкой <***O***,***S***> и свойствами {r_1, r_2, r_3}.

50) Из анализа информационных структур каких составляющих ПСРВ, определяемых тройкой <***O***,***C***,***S***>, вытекает свойство концептуальной модели ПСРВ: r_1 – *многообразие информационных процессов, осуществляющих импорт/экспорт данных*?
   a) Из анализа информационных структур O и S.
   b) Из анализа информационных структур ***O***,***C*** и ***S***.
   c) Из анализа информационной структуры ***C***.

51) Каким свойствам концептуальной модели распределенной ПСРВ удовлетворяют *CF*-модели программ?
   a) {r_1, r_2, r_3, r_4, r_5, r_6}.
   b) {r_1, r_2, r_3, r_4, r_5}.
   c) {r1, r2, r3}.

52) Каким свойствам концептуальной модели распределенной ПСРВ не удовлетворяют *CF*-модели программ?
   a) {r_1, r_2, r_3}.
   b) {r4, r5, r6}.
   c) {r_5, r_6}.

53) Какие значения используются для интерпретации содержимого объектов памяти в АСПД-схемах?
   a) `valid`, `void`.
   b) valid, invalid, void.
   c) `valid`, `invalid`.

54) Как интерпретируется результат операции *`извлечь`* над объектом памяти типа *`ячейка`*, если содержимое есть `void`?
   a) операция запрещена.
   b) извлекается пустое данное.
   c) извлекается данное `invalid`.

55) Как интерпретируется результат операции *`извлечь`* над объектом памяти типа *`полуячейка`*, если содержимое есть `void`?

a) операция запрещена.

b) извлекается пустое данное.

c) извлекается данное `invalid`.

56) Как интерпретируется результат операции *`извлечь`* над объектом памяти типа *`псевдоячейка`*, если текущее содержимое интерпретируется как `void`?

a) операция запрещена.

b) извлекается пустое данное.

c) извлекается данное `invalid`.

57) Как интерпретируется результат операции *`поместить`* над объектом памяти типа *`гнездо`*, если текущее содержимое интерпретируется как `valid`?

a) операция запрещена.

b) `void`.

c) `invalid`.

58) Как интерпретируется результат операции *`извлечь`* над объектом памяти типа *`гнездо`*, если текущее содержимое интерпретируется как `void`?

a) операция запрещена.

b) `void`.

c) `invalid`.

59) Как интерпретируется результат операции *`поместить`* над объектом памяти типа *`полугнездо`*, если текущее содержимое интерпретируется как `valid`?

a) операция запрещена.

b) сохраняет текущее valid.

c) `invalid`.

60) Как интерпретируется результат операции *`поместить`* над объектом памяти типа *`полуячейка`*, если текущее содержимое интерпретируется как `valid`, а помещается `invalid`?

a) операция запрещена.

b) сохраняет текущее `valid`.

c) invalid.

61) Какие типы данных могут находиться в объектах памяти АСПД-схемы?
    a) данные типа параметр, сигнал, токен.
    b) данные типа *параметр*, *токен*.
    c) данные типа *параметр*, *сигнал*.
62) Какие данные могут содержаться в качестве сообщений в *токенах*?
    a) статическое данное (STATIC), динамическое данное (DYNAMIC) , состав (TRAIN).
    b) динамическое данное (DYNAMIC), состав (TRAIN).
    c) состав (TRAIN).
63) Что включает в себя *тег токена*?
    a) Идентификатор потока, приоритет потока.
    b) Идентификатор потока, приоритет потока, тип сообщения.
    c) Идентификатор потока, приоритет потока, метку времени.
64) Сколько уровней абстракции рассмотрения программной системы включает в себя технология потоко-ориентированного проектирования и разработки распределенных ПСРВ?
    a) 5 уровней
    b) 6 уровней
    c) 7 уровней
    d) 8 уровней
    e) 9 уровней
65) На каком иерархическом уровне абстракции технология потоко-ориентированного проектирования и разработки ПСРВ специфицирует программную систему в виде *абстрактного программного агрегата*, взаимодействующего с абстрактными объектами памяти - *истоками* и *стоками* данных (внешним окружением)?
    a) на 1 уровне
    b) на 2 уровне
    c) на 3 уровне
    d) на 4 уровне
    e) на 5 уровне
    f) на 6 уровне
    g) на 7 уровне
66) На каком иерархическом уровне абстракции технология потоко-ориентированного проектирования и разработки ПСРВ специфицирует

программную систему в виде схемы асинхронных потоков данных (*АСПД-схемы*)?

a) на 1 уровне
b) на 2 уровне
c) на 3 уровне
d) на 4 уровне
e) на 5 уровне
f) на 6 уровне
g) на 7 уровне

67) На каком иерархическом уровне абстракции технология потоко-ориентированного проектирования и разработки ПСРВ специфицирует программную систему в виде *ВПМ-сети*?

a) на 1 уровне
b) на 2 уровне
c) на 3 уровне
d) на 4 уровне
e) на 5 уровне
f) на 6 уровне
g) на 7 уровне

68) На каком иерархическом уровне абстракции технология потоко-ориентированного проектирования и разработки ПСРВ специфицирует программную систему в виде *А-сети*?

a) на 1 уровне
b) на 2 уровне
c) на 3 уровне
d) на 4 уровне
e) на 5 уровне
f) на 6 уровне
g) на 7 уровне

69) На каком иерархическом уровне абстракции технология потоко-ориентированного проектирования и разработки ПСРВ специфицирует программную систему в составе *распределённой вычислительного комплекса*?

a) на 1 уровне
b) на 2 уровне
c) на 3 уровне
d) на 4 уровне
e) на 5 уровне
f) на 6 уровне
g) на 7 уровне

70) На каком иерархическом уровне абстракции технология потоко-ориентированного проектирования и разработки ПСРВ специфицирует программную систему в виде *программно-модульной* (файловой) структуры?

a) на 1 уровне
b) на 2 уровне
c) на 3 уровне
d) на 4 уровне
e) на 5 уровне
f) на 6 уровне
g) на 7 уровне

71) Какие менеджеры входят в состав ПРВ, разработанного в рамках потоко-ориентированной технологии проектирования и разработки ПО СРВ?

a) менеджер А-сети, менеджер агрегата, менеджер ВПМ.
b) менеджер АСПД-схемы, менеджер актора, менеджер ВПМ.
c) менеджер А-сети, менеджер актора, менеджер ВПМ.

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет информатики
Кафедра программных систем
Направление: 010300.68 «Фундаментальная информатика и информационные технологии» (магистр)
Дисциплина "Технологии промышленного программирования"

АТТЕСТАЦИОННЫЙ БИЛЕТ № 1

1) **Если на компьютере с именем comp выполнена команда shell:**
   #mount –T io-net /lib/dll/npm-qnet.so,
   то в файловой системе компьютера comp …
   a) **создаётся каталог /net, содержащий имена всех узлов, выполнивших монтирование локальной сети.**
   b) **в каталоге /fs появятся имена всех узлов, выполнивших монтирование локальной сети.**
   c) **в каталоге /fs/net появятся имена всех узлов, выполнивших монтирование локальной сети.**
2) **Какой процесс будет считаться родительским при запуске процесса функцией system()?**
   a) **процесс, выполнивший функцию system()**
   b) **системный процесс ядра**
   c) **командный интерпретатор shell**
3) **Какие функции позволяют запустить процесс на удалённом узле?**
   a) `execv()`, `spawnve()`
   b) `spawn()`
   c) `fork()`, `vfork()`
4) **Являются ли вызовы `MsgReply()`,`MsgRead()`,`MsgWrite()` блокирующими?**
   a) **да.**
   b) **да, при взаимодействии с клиентом на удаленном узле.**
   c) **только `MsgRead()` и `MsgWrite()`.**
5) **Какой вид имеет объявление функции, используемой для запуска нитей?**
   a) `void func(void*)`
   b) `void* func(void*)`
   c) `void* func(void)`

6) **Может ли нить `main()` быть обособленной?**
   a) **нет.**
   b) **да.**
   c) **да, если ей установлено свойство `PTHREAD_CREATE_DETACHED`.**
7) **Какие параметры стека создаваемой нити можно задать одновременно?**
   a) **Адрес стека, размер стека и размер "области защиты" стека.**
   b) **Адрес стека и размер "области защиты" стека.**
   c) **Адрес стека и размер стека.**
8) **Какие значения приоритета можно назначить нити в QNX?**
   a) **От -128 до 127.**
   b) **От 0 до 64000.**
   c) **От 0 до 63.**
9) **Активная нить выполнила функцию освобождения процессора - `sched_yield()`, какая из готовых к выполнению нитей начнет выполняться?**
   a) **нить с более высоким приоритетом.**
   b) **нить с более низким приоритетом.**
   c) **нить с тем же приоритетом.**
10) **Что понимается под *инверсией приоритетов* нитей?**
   a) **приобретение сервером приоритета клиента.**
   b) **опосредованное использование клиентом приоритета сервера.**
   c) **приоритет нити становится отрицательным.**
11) **Какие функции связывают продолжение выполнения одной нити с моментом завершения существования другой нити?**
   a) **`pthread_join()`.**
   b) **`barrier_wait()`.**
   c) **`pthread_cond_wait()`**

12) Рекурсивный мутекс позволяет нити…

a) **использовать при захвате мутекса рекурсивные функции для обработки ресурса.**

b) **осуществлять нитью многократный захват того же мутекса без его предварительного освобождения.**

c) **получить информацию о повторном захвате того же мутекса.**

13) Если нити требуется исключительный доступ к некоторому ресурсу, но использовать ресурс нить может только при выполнении определенных условий, то каким из приведенных ниже механизмов синхронизации можно воспользоваться наиболее эффективно?

a) **Создать барьер к ресурсу, который будет блокировать нити до момента прибытия к барьеру нити, отвечающей за выполнение условия доступа к ресурсу.**

b) **Создать ждущую блокировку. После захвата ждущей блокировки организовать цикл ожидания уведомлений от других нитей о возникших событиях, изменяющих условие. Для каждого уведомления проверять выполнение условия и либо вновь ждать, либо использовать ресурс и освободить ждущую блокировку.**

c) **Использовать мутекс и флаг выполнения условия. Организовать цикл ожидания готовности флага. Когда флаг готов, захватить мутекс и использовать ресурс, затем освободить мутекс и сбросить флаг.**

14) Какой объект управления синхронизацией нитей создаётся функцией:
`int sem_init(sem_t *sem,int pshared,unsigned value)` **?**

a) **Неименованный семафор.**

b) **Именованный семафор.**

c) **Семафор, разделяемый нитями разных процессов.**

15) Какая функция приводит к увеличению на 1 значения счётчика семафора?

a) `sem_t* sem_wait(sem_t *sem);.`

b) `sem_t* sem_post(sem_t *sem);.`

c) `sem_t* sem_getvalue(sem_t *sem,int value);.`

16) *Механизм отображения адресов* системных областей памяти в адресное пространство процесса используется для доступа процесса к …

a) **адресам памяти физических устройств и оперативной памяти за пределами локального адресного пространства процесса.**

b) **адресам памяти физических устройств.**

c) **адресам оперативной памяти удалённого узла локальной сети.**

17) Если процессу приходит сигнал, а обработчик сигнала не установлен, то как это отразится на процессе?

a) **процесс терминируется.**

b) **сигнал будет проигнорирован.**

c) **выполнится действие, предписанное сигналу по умолчанию.**

18) Если сигнал генерируется функцией `kill(pid, sig)`, где `pid>0`, то кому адресуется сигнал?

a) **конкретному процессу.**

b) **всем процессам, входящим в группу, которой принадлежит процесс, пославший сигнал.**

c) **процессам группы с `GID=-pid`.**

19) Какие функции позволяют послать сигнал процессу, находящемуся на удаленном узле?

a) **`kill()`.**

b) **`SignalKill()`.**

c) **`kill()`, `SignalKill()`.**

20) Какие функции позволяют установить обработчик сигналов в режиме, который предусматривает возможность образования очереди сигналов?

a) **`signal()`.**

b) **`sigaction()`.**

c) **`signal()`, `sigaction()`.**

Утверждено кафедрой ПС, протокол № ___ от _____ г.

Зав.кафедрой_____

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет информатики
Кафедра программных систем
Направление: 010300.68 «Фундаментальная информатика и информационные технологии» (магистр)
Дисциплина "Технологии промышленного программирования"

АТТЕСТАЦИОННЫЙ БИЛЕТ № 2

1) **Укажите имя пользователя, которое автоматически регистрируется при установке ОС QNX.**
   a) **root**
   b) **admin**
   c) **supervisor**
2) **Какие функции позволяют запустить процесс на удалённом узле?**
   a) **`execv()`, `spawnve()`.**
   b) **`spawn()`.**
   c) **`spawn()`, `fork()` и `vfork()`.**
3) **Клиент посылает 120 Кб данных серверу на удаленном узле. Какой предельный объем данных будет получен сервером при выполнении `MsgReceive()`?**
   a) **120 Кб, если указан объем буфера приема не менее 120 Кб.**
   b) **объем буфера приема, если он менее 120 Кб.**
   c) **не более 8 Кб.**
4) **Родительский процесс, находящийся на узле с именем `compA`, запустил дочерний процесс на удаленном узле с именем `compB` и передал ему необходимые параметры. Какая последовательность действий дочернего процесса является верной для установления соединения с родительским процессом?**
   a) **определить ID родительского процесса - `pid=getppid()`; получить ID канала `(chid)`, используя аргументы функции `main()`; выполнить `chroot("/net/compB/")`; определить ID узла родителя - `nd=netmgr_strtond("/net/compA",NULL)`, и выполнить функцию соединения `ConnectAttach()`.**
   b) **получить `pid` и `chid` родительского процесса, используя аргументы функции `main()`, определить дескриптор узла родительского процесса - `nd=netmgr_strtond("/net/compA",NULL)`, выполнить `chroot("/net/compB/")`, и выполнить функцию соединения `ConnectAttach()`.**
   c) **получить `pid` и `chid` родительского процесса, используя аргументы функции `spawn()`, выполнить `chroot("/net/compB/")`, определить дескриптор узла родительского процесса - `nd=netmgr_strtond("/net/compA",NULL)`, и выполнить функцию соединения `ConnectAttach()`.**

5) **Какова особенность запуска дочернего процесса с помощью функции vfork()?**
   a) **родительский процесс завершает выполнение, а дочерний процесс создается с ID родительского процесса**
   b) **родительский процесс порождает свою копию и ждет завершения дочернего процесса**
   c) **порождается процесс, являющийся копией родительского процесса, который стартует непосредственно за вызовом fork() и выполняется параллельно с родительским процессом**
6) **В каком состоянии окажется процесс-сервер, выполнивший функцию MsgReceive(), если процесс-клиент был в Send-блокированном состоянии?**
   a) **"Готов"**
   b) **Receive-блокированном**
   c) **Reply-блокированном**
7) **Что обеспечивает механизм векторов ввода/вывода при обмене сообщением между клиентом и сервером?**
   a) **прием/передачу сообщения, в виде упорядоченной совокупности отдельных частей.**
   b) **возникновение программного прерывания при выполнении клиентом функции MsgSend().**
   c) **посылку ядром ОС сигнала серверу при выполнении клиентом функции MsgSend().**
8) **Могут ли в приложении быть запущены нити с разными дисциплинами диспетчеризации?**
   a) **Да**
   b) **Нет**
   c) **Да, в разных процессах**
9) **Что понимается под *инверсией приоритетов* нитей?**
   a) **приобретение сервером приоритета клиента.**
   b) **опосредованное использование клиентом приоритета сервера.**
   c) **приоритет нити становится отрицательным.**
10) **В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер был в Receive-блокированном состоянии?**
   a) **"Готов"**
   b) **Reply-блокированном**
   c) **Send-блокированном**

11) Какие функции связывают продолжение выполнения одной нити с моментом завершения существования другой нити?

a) `pthread_join().`

b) `barrier_wait().`

c) `pthread_join(),barrier_wait().`

12) Если для управления доступом нитей к некоторому разделяемому программному ресурсу создан мутекс, то что предпримет ядро по отношению к нити, которая попытается осуществить доступ к этому ресурсу, не захватывая мутекс?

a) **ничего.**

b) **она будет терминирована ядром.**

c) **будет заблокирована, пока имеется нить, захватившая мутекс.**

13) Что произойдет с нитью, если она по отношению к блокировке чтения/записи выполнит функцию `pthread_rwlock_wrlock()` или `pthread_rwlock_rdlock()`, а блокировка чтения/записи уже захвачена?

a) **Нить блокируется до момента освобождения блокировки чтения/записи.**

b) **Нить блокируется и ей возвращается ошибка.**

c) **Функции завершаются, информируя о неудачной попытке захвата блокировки чтения/записи, и нить продолжает выполнение.**

14) Какой объект синхронизации нитей создаётся функцией:
`sem_t* sem_open(char* sem_name,int oflags,mode_t mode,unsigned value);` **?**

a) **Неименованный семафор.**

b) **Именованный семафор.**

c) **Разделяемый семафор.**

15) Какая функция уменьшает на 1 значение счётчика семафора?

a) `sem_t* sem_wait(sem_t *sem);.`

b) `sem_t* sem_post(sem_t *sem);.`

c) `sem_t* sem_getvalue(sem_t *sem,int value);.`

16) Для создания процессом именованной системной области памяти применяется функция…

a) `sem_t* sem_open();`

b) `void* mmap();`

c) `int shm_open();`

17) Если процессу приходит сигнал, а обработчик сигнала не установлен, то как это отразится на процессе?

a) **процесс терминируется.**

b) **сигнал будет проигнорирован.**

c) **выполнится действие, предписанное сигналу по умолчанию.**

18) Если сигнал генерируется функцией `kill(pid, sig)`, где `pid=0`, то сигнал адресуется…

a) **процессу.**

b) **всем процессам, входящим в группу, которой принадлежит процесс, пославший сигнал.**

c) **всем процессам группы с `GID=-pid`.**

19) Какие функции позволяют послать сигнал процессу, находящемуся на удаленном узле?

a) **`kill()`.**

b) **`SignalKill()`.**

c) **`kill()`, `SignalKill()`.**

20) Если нить замаскировала сигнал, то она может контролировать задержанные инициации сигнала с помощью функции …

a) **`signal()`.**

b) **`sigaction()`.**

c) **`SignalWaitinfo()`.**

Утверждено кафедрой ПС, протокол № ____ от _____ г.

Зав.кафедрой_____

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет информатики
Кафедра программных систем
Направление: 010300.68 «Фундаментальная информатика и информационные технологии» (магистр)
Дисциплина "Технологии промышленного программирования"

АТТЕСТАЦИОННЫЙ БИЛЕТ № 3

1) **Что необходимо выполнить удаленному процессу-клиенту с результатом выполнения функции `nd=netmgr_strtond("/net/comp",NULL)`, используемого для организации соединения с процессом-сервером на удаленном узле с именем `comp`?**
   a) **сохранить `nd` в статической памяти и использовать его каждый раз при очередной необходимости создания соединения с процессом-сервером на удаленном узле с именем `comp`.**
   b) **немедленно использовать `nd` в вызове `ConnectAttach()` для создания соединения с процессом-сервером на удаленном узле с именем `comp`.**
   c) **использовать `nd` в вызове `MsgSend()` при посылке сообщения процессу-серверу.**
2) **Какие и чьи права доступа будет иметь файл, созданный при выполнении функции open("file.dat", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)?**
   a) **чтение и запись для пользователей, входящих в группу-владельца файла file.dat**
   b) **чтение и запись для пользователей файла file.dat**
   c) **чтение и запись для пользователя-владельца файла file.dat**
3) **Какие функции позволяют запустить процесс на удалённом узле?**
   a) **`execv()`, `spawnve()`.**
   b) **`spawn()`.**
   c) **`fork()` и `vfork()`.**
4) **Как завершится выполнение функции open("file.dat", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR); если в текущем каталоге существует файл с именем file.dat?**
   a) **`новый файл заменит существующий (сохранив дескриптор старого)`**
   b) **`уничтожит старый и создаст новый файл с тем же именем`**
   c) **`выдаст сообщение об ошибке`**
5) **Какой вид имеет объявление функции, используемой для запуска нитей?**
   a) **`void func(void*).`**
   b) **`void* func(void*).`**
   c) **`void* func(void).`**

6) **Что означает для нити, запущенной в процессе, свойство обособленности?**
   a) **Выполнение другими нитями по отношению к ней функции `ptread_join()` не является блокирующим.**
   b) **Выполнение другими нитями по отношению к ней функции `ptread_join()` является блокирующим.**
   c) **Она не разделяет с другими нитями созданную ею именованную память.**
7) **Какие параметры стека создаваемой нити можно задать одновременно?**
   a) **Адрес стека, размер стека и размер "области защиты" стека.**
   b) **Адрес стека и размер "области защиты" стека.**
   c) **Адрес стека и размер стека.**
8) **Если завершился квант процессорного времени текущей нити с приоритетом `pr`, то ожидающей нити с каким приоритетом будет предоставлен очередной квант процессорного времени?**
   a) **нити с приоритетом равным `pr`.**
   b) **нити с приоритетом равным или большим `pr`.**
   c) **нити с приоритетом большим `pr`.**
9) **Какой процесс будет считаться родительским при запуске процесса функцией system()?**
   a) **процесс, выполнивший функцию system()**
   b) **системный процесс ядра**
   c) **командный интерпретатор shell**
10) **Что означает механизм наследования приоритетов для борьбы с инверсией приоритетов?**
   a) **Сервер при получении сообщения от клиента приобретает приоритет клиента**
   b) **Клиент при получении сервером от него сообщения приобретает приоритет сервера**
   c) **Сервер при получении сообщения от клиента приобретает приоритет клиента, если он выше приоритета сервера**
11) **Какой атрибут устанавливается процессом в атрибутной записи запускаемой нити, чтобы она была обособленной нитью?**
   a) **`PTHREAD_CREATE_JOINABLE`**
   b) **`PTHREAD_CREATE_DETACHED`**
   c) **`PTHREAD_EXPLICIT_SCHED`**

12) Если нити требуется исключительный доступ к некоторому ресурсу, но использовать ресурс нить может только при выполнении определенных условий, то каким из приведенных механизмов синхронизации можно воспользоваться наиболее эффективно?

a) **Создать барьер к ресурсу, который будет блокировать нити до момента прибытия к барьеру нити, отвечающей за выполнение условия доступа к ресурсу.**

b) **Создать мутекс и условную переменную. После захвата мутекса организовать цикл ожидания уведомлений по условной переменной от других нитей о возникших событиях, изменяющих условие. Для каждого уведомления проверять выполнение условия и либо вновь ждать, либо использовать ресурс и освободить мутекс.**

c) **Использовать мутекс и флаг выполнения условия. Организовать цикл ожидания готовности флага. Когда флаг готов, захватить мутекс и использовать ресурс, затем освободить мутекс и сбросить флаг.**

13) Что произойдет с нитью, если она по отношению к некоторой блокировке чтения/записи выполнит функцию `pthread_rwlock_trywrlock()` или `pthread_rwlock_tryrdlock()`, а блокировка чтения/записи уже захвачена?

a) **Нить блокируется до момента освобождения блокировки чтения/записи.**

b) **Нить блокируется и ей возвращается ошибка.**

c) **Функции завершаются, информируя о неудачной попытке захвата блокировки чтения/записи, и нить продолжает выполнение.**

14) Какой программный объект создаётся функцией:
`int shm_open(const char *name, int oflag, mode_t mode)?`

a) **Именованный канал**

b) **Именованный семафор**

c) **Именованная память**

15) Выполнение какой функции приводит к уменьшению на 1 значения счётчика семафора?

a) **`sem_t* sem_wait(sem_t *sem);.`**

b) **`sem_t* sem_post(sem_t *sem);.`**

c) **`sem_t* sem_getvalue(sem_t *sem,int value);.`**

16) Для отображения именованной памяти в адресное пространство процесса применяется функция…

a) **`sem_t* sem_open();`**

b) **`void* mmap();`**

c) **`int shm_open();`**

17) Если процессу приходит сигнал, а обработчик сигнала не установлен, то как это отразится на процессе?

a) **процесс терминируется.**

b) **сигнал будет проигнорирован.**

c) **выполнится действие, предписанное сигналу по умолчанию.**

18) Если сигнал генерируется функцией `kill(pid, sig)`, где `pid<0`, то кому адресуется сигнал?

a) **процессу.**

b) **всем процессам, входящим в группу, которой принадлежит процесс, пославший сигнал.**

c) **всем процессам группы с `GID=-pid`.**

19) Какие функции позволяют послать сигнал, адресуемый конкретной нити?

a) **`kill()`.**

b) **`SignalKill()`.**

c) **`kill()`, `SignalKill()`.**

20) Если сигнал адресован нити, которая замаскировала этот сигнал, то …

a) **сигнал будет проигнорирован процессом, которому принадлежит нить.**

b) **сигнал принимается процессом и направляется текущей активной нити процесса.**

c) **действие сигнала задерживается до момента сброса нитью маски сигнала.**

Утверждено кафедрой ПС, протокол № ___ от _____ г.

Зав.кафедрой_____

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ
Факультет информатики
Кафедра программных систем
Направление: 010300.68 «Фундаментальная информатика и информационные технологии» (магистр)
Дисциплина "Технологии промышленного программирования"

АТТЕСТАЦИОННЫЙ БИЛЕТ № 4

1) **Если на компьютере с именем comp выполнена команда shell:**
   #mount –T io-net /lib/dll/npm-qnet.so,
   то в файловой системе компьютера comp …
   a) **создаётся каталог /net, содержащий имена всех узлов, выполнивших монтирование локальной сети.**
   b) **в каталоге /fs появятся имена всех узлов, выполнивших монтирование локальной сети.**
   c) **в каталоге /fs/net появятся имена всех узлов, выполнивших монтирование локальной сети.**
2) **Какие функции позволяют запустить процесс на удалённом узле?**
   a) **`execv()`, `spawnve()`.**
   b) **`spawn()`.**
   c) **`spawn()`, `fork()` и `vfork()`.**
3) **Какие и чьи права доступа будет иметь файл, созданный при выполнении функции open("file.dat", O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR)?**
   a) **чтение и запись для пользователей, входящих в группу-владельца файла file.dat**
   b) **чтение и запись для пользователей файла file.dat**
   c) **чтение и запись для пользователя-владельца файла file.dat**
4) **Являются ли вызовы `MsgReply()`,`MsgRead()`,`MsgWrite()` блокирующими?**
   a) **да.**
   b) **да, при взаимодействии с клиентом на удаленном узле.**
   c) **только `MsgRead()` и `MsgWrite()`.**
5) **Может ли нить `main()` быть обособленной?**
   a) **нет.**
   b) **да.**
   c) **да, если ей установлено свойство `PTHREAD_CREATE_DETACHED`.**

6) **Какова особенность запуска дочернего процесса с помощью функции vfork()?**
   a) **родительский процесс завершает выполнение, а дочерний процесс создается с ID родительского процесса**
   b) **родительский процесс порождает свою копию и ждет завершения дочернего процесса**
   c) **порождается процесс, являющийся копией родительского процесса, который стартует непосредственно за вызовом fork() и выполняется параллельно с родительским процессом**
7) **В каком состоянии окажется процесс-клиент, выполнивший функцию MsgSend(), если процесс-сервер был в состоянии "Готов"?**
   a) **"Готов"**
   b) **Reply-блокированном**
   c) **Send-блокированном**
8) **Что обеспечивает механизм векторов ввода/вывода при обмене сообщением между клиентом и сервером?**
   a) **прием/передачу сообщения, в виде упорядоченной совокупности отдельных несвязных частей.**
   b) **возникновение программного прерывания при выполнении клиентом функции MsgSend().**
   c) **посылку ядром ОС сигнала серверу при выполнении клиентом функции MsgSend().**
9) **Если завершился квант процессорного времени текущей нити с приоритетом `pr`, то ожидающей нити с каким приоритетом будет предоставлен очередной квант процессорного времени?**
   a) **нити с приоритетом равным `pr`**
   b) **нити с приоритетом равным или большим `pr`**
   c) **нити с приоритетом большим `pr`**
10) **Активная нить выполнила функцию освобождения процессора – `sched_yield()`, какая из готовых к выполнению нитей начнет выполняться?**
   a) **нить с более высоким приоритетом.**
   b) **нить с более низким приоритетом.**
   c) **нить с тем же приоритетом.**
11) **Могут ли в приложении быть запущены нити с разными дисциплинами диспетчеризации?**
   a) **Да**
   b) **Нет**
   c) **Да, в разных процессах**

12) Какой процесс будет считаться родительским при запуске процесса функцией system()?

a) **процесс, выполнивший функцию system()**

b) **системный процесс ядра**

c) **командный интерпретатор shell**

13) Какой программный объект создаётся функцией:
`int sem_init(sem_t *sem,int pshared,unsigned value)` **?**

a) **Неименованный семафор.**

b) **Именованный семафор.**

c) **Ждущая блокировка.**

14) Если для управления доступом нитей к некоторому разделяемому программному ресурсу создан мутекс, что предпримет ядро по отношению к нити, которая попытается осуществить доступ к этому ресурсу, не захватывая мутекс?

a) **ничего.**

b) **она будет терминирована ядром.**

c) **будет заблокирована, пока имеется нить, захватившая мутекс.**

15) Что произойдет с нитью, если она по отношению к некоторой блокировке чтения/записи выполнит функцию `pthread_rwlock_trywrlock()` или `pthread_rwlock_tryrdlock()`, а блокировка чтения/записи уже захвачена?

a) **Нить блокируется до момента освобождения блокировки чтения/записи.**

b) **Нить блокируется и ей возвращается ошибка.**

c) **Функции завершаются, информируя о неудачной попытке захвата блокировки чтения/записи, и нить продолжает выполнение.**

16) Какая функция приводит к увеличению на 1 значения счётчика семафора?

a) `sem_t* sem_wait(sem_t *sem);.`

b) `sem_t* sem_post(sem_t *sem);.`

c) `sem_t* sem_getvalue(sem_t *sem,int value);.`

17) Если процессу приходит сигнал, а обработчик сигнала не установлен, то как это отразится на процессе?

a) **процесс терминируется.**

b) **сигнал будет проигнорирован.**

c) **выполнится действие, предписанное сигналу по умолчанию.**

18) Какие функции позволяют послать сигнал процессу, находящемуся на удаленном узле?

a) `kill()`.

b) `SignalKill()`.

c) `kill()`, `SignalKill()`.

19) Если сигнал генерируется функцией `kill(pid, sig)`, где `pid<0`, то кому адресуется сигнал?

a) процессу.

b) всем процессам, входящим в группу, которой принадлежит процесс, пославший сигнал.

c) всем процессам группы с `GID=-pid`.

20) Если сигнал адресован нити, которая замаскировала этот сигнал, то …

a) сигнал будет проигнорирован процессом, которому принадлежит нить.

b) сигнал принимается процессом и направляется текущей активной нити процесса.

c) действие сигнала задерживается до момента сброса нитью маски сигнала.

Утверждено кафедрой ПС, протокол № ___ от _____ г.

Зав.кафедрой_____

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный аэрокосмический
университет имени академика С.П. Королёва
(национальный исследовательский университет)»

СОГЛАСОВАНО

Управление образовательных программ

_____/ А.В. Дорошин /

"_____"_____ 20___ г.

УТВЕРЖДАЮ

Проректор по учебной работе

_____/ Ф.В. Гречников /

"____" _____ 20___ г.

РАБОЧАЯ ПРОГРАММА

Наименование модуля (дисциплины)

Технологии промышленного программирования

Цикл, в рамках которого происходит освоение модуля (дисциплины)	М2.В.2. Профессиональный цикл
Часть цикла	Вариативная
Код учебного плана	010300.68-2011-О-П-2г00м
Факультет	6
Кафедра	Программные системы

Курс	6		
Семестр	11		
Лекции (СЛ)	0		
Семинарские и практические занятия (СП)	36		
Лабораторные занятия (СЛР)	54	Экзамен	11
Контроль самастоятельной работы (КСР)	0	Зачет	-
Самостоятельная работа (СРС)	54		
Всего	144		

Наименование стандарта, на основании которого составлена рабочая программа:

010300.68 «Фундаментальная информатика и информационные технологии»

Соответствие содержания рабочей программы, условий ее реализации, материально-технической и учебно-методической обеспеченности учебного процесса по дисциплине всем требованиям государственных стандартов подтверждаем.

Составители:

Баландин Александр Васильевич, к.т.н., доцент _____ (подпись)

Заведующий кафедрой:

Коварцев Александр Николаевич, д.т.н., профессор _____ (подпись)

Рабочая программа обсуждена на заседании кафедры

Программные системы

Протокол №___ от "____" _____ 20___ г.

Наличие основной литературы в фондах научно-технической библиотеки (НТБ) подтверждаем:

Директор НТБ _____ (подпись) /_____/ (расшифровка подписи)

Согласовано:

Декан _____ (подпись) /_____/ (расшифровка подписи)

1 Цели и задачи модуля (дисциплины), требования к уровню освоения содержания

1.1 Перечень развиваемых компетенций

Общекультурные компетенции: ОК-5, ОК-7, ОК-8.
Профессиональные компетенции: ПК-1, ПК-2, ПК-3, ПК-6.

1.2 Цели и задачи изучения модуля (дисциплины)

1. Подготовка студентов в области использования современных технологий промышленного программирования.
2. Обучить студентов программным средствам современных операционных систем реального времени (ОСРВ), используемых в промышленном программировании.
3. Обучить студентов методам структурного анализа, моделирования и макетирования программного обеспечения систем компьютерного мониторинга промышленных объектов.

1.3 Требования к уровню подготовки студента, завершившего изучение данного модуля (дисциплины)

Студенты, завершившие изучение данной дисциплины, должны уметь разрабатывать программное обеспечение систем промышленной автоматизации на уровне управления промышленными объектами в режиме реального времени.

1.4 Связь с предшествующими модулями (дисциплинами)

Для успешного освоения курса "Технологии промышленного программирования" студенты должны обладать знаниями дисциплин:
1. Дискретная математика.
2. Архитектура вычислительных систем.
3. Программирование.
3. Параллельное программирование.

1.5 Связь с последующими модулями (дисциплинами)

Знания, полученные при изучении курса "Технологии промышленного программирования", могут быть использованы магистрами при выполнении научно-исследовательской работы и подготовке выпускной квалификационной работы (диссертации).

2 Содержание рабочей программы (модуля)

Семестр 1		
СЛ 0 0 часов 0 ЗЕТ	Активные 0	
	Интерактивные 0	
	Традиционные 1	

СП 0,25 36 часов 1 ЗЕТ	Активные 1	Особенности программирования систем промышленной автоматизации (СПА). Операционные системы реального времени (ОСРВ). Приложения реального времени (ПРВ).
		Платформа разработки приложений реального времени QNX Momentics IDE 4.7.
		Управление файловой системой. Разграничение доступа к файлам. Пользователи и группы.
		Структура приложений реального времени. Программные модули, процессы, нити. Типы процессов. Запуск и управление процессами.
		Программный интерфейс ОСРВ QNX для организации взаимодействия между процессами посредством сообщений. Создание/Удаление канала. Установление/Удаление соединений с каналом. Посылка/Прием сообщения. Посылка ответа. Сценарии ответов. Управление приемом и пер
		Нитиевая структура процессов. Прототип функции и свойства нити. Формирование свойств и запуск нити. Приоритет и дисциплина диспетчеризации нити. Создание и запуск нити.
		Методы и функции синхронизации нитей. Присоединение. Барьеры. Мутексы. Блокировки чтения/записи. Семафоры. Условные переменные.
		Управление памятью вне адресного пространства процессов. Именованная оперативная память. Доступа к именованной памяти. Доступа к памяти устройств ввода/вывода.
		Сигналы. Механизм надежных сигналов. Набор сигналов и маска блокирования. Посылка и доставка сигнала процессу. Реакция процесса на сигнал. Управление сигналами.
		Синхронизация нитей с реальным временем. Системное реальное время. Разрешающая способность РВ. Установка значений абсолютного и интервального времени. Таймеры. Создание/Удаление таймера. Типы уведомления нитей о временн'ых событиях.

		Прерывания от устройств ввода/вывода. Управление прерываниями в QNX. Механизм обработки прерываний в процессах.
		Проектирование ПРВ: структурный анализ, моделирование и макетирование ПРВ. Иерархическая структура ПРВ.
		Моделирование функциональной структуры ПРВ. Модель абстрактной памяти. Типы объектов памяти.
		Темпоральная модель данных. Датированные значения. Модель потоковых данных - токены.
		Моделирование ПРВ на функциональном уровне. Язык схем асинхронных потоков данных (АСПД-схем). Объекты памяти АСПД-схемы: истоки, стоки, проходная память. Акторы. Состав и формальная спецификация базовых акторов АСПД-схемы.
		Моделирование ПРВ на процессном уровне. Виртуальная потоковая машина (ВПМ). Архитектура и функционирование ВПМ. ПРВ как сеть виртуальных потоковых машин (ВПМ-сеть).
		Моделирование ПРВ на агрегатном уровне. Понятие программного агрегата (ПА). ПРВ как сеть программных агрегатов (ПА-сеть). Абстрактная спецификация распределенного вычислительного комплекса (РВК). Распределение ПА-сети в РВК.
		Организация ПРВ на программно-модульном уровне. Загрузка ПРВ в РВК.
	Интерактивные 0	
	Традиционные 0	
СЛР 0,375 54 часов 1,5 ЗЕТ	Активные 0	
	Интерактивные 1	Лабораторная работа 1. Установка платформы разработки приложений реального времени (QNX Software Development Platform - QNX SDP) на инструментальный компьютер.

		Лабораторная работа 2. Программирование в QNX SDP запуска параллельных процессов и организации взаимодействия между процессами.
		Лабораторная работа 3. Программирование в QNX SDP запуска распределенных процессов и организации взаимодействия между процессами в сети.
		Лабораторная работа 4. Программирование в QNX SDP многонитиевых приложений. Использование методов и функций синхронизации нитей.
		Лабораторная работа 5. Программирование в QNX SDP многонитиевых приложений. Использование методов и функций управления нитями в реальным времени.
		Лабораторная работа 6. Использование методов и функций ОСРВ QNX для управления прерываниями.
		Лабораторная работа 7. Освоение метода структурного анализа, моделирования и макетирования ПСРВ. Выполнение комплексного задания.
	Традиционные 0	
КСР 0 0 часов 0 ЗЕТ	Активные 1	
	Интерактивные 0	
	Традиционные 0	
СРС 0,375 54 часов 1,5 ЗЕТ	Активные 0	
	Интерактивные 1	Изучение теории, программирование, отладка и подготовка к защите отчета по результатам выполнения лабораторной работы 1.
		Изучение теории, программирование, отладка и подготовка к защите отчета по результатам выполнения лабораторной работы 2.
		Изучение теории, программирование, отладка и подготовка к защите отчета по результатам выполнения лабораторной работы 3.

		Изучение теории, программирование, отладка и подготовка к защите отчета по результатам выполнения лабораторной работы 4.
		Изучение теории, программирование, отладка и подготовка к защите отчета по результатам выполнения лабораторной работы 5.
		Изучение теории, программирование, отладка и подготовка к защите отчета по результатам выполнения лабораторной работы 6.
		Изучение теории, программирование, отладка и подготовка к защите отчета по результатам выполнения лабораторной работы 7.
	Традиционные 0	

3 Инновационные методы обучения

1. Выполнение комплексной лабораторной работы №7 с необходимостью выбора проектных решений.
2. Прием отчета по лабораторной работе №7 с элементами защиты проектных решений по выбору структуры ПСРВ.

4 Технические средства и материальное обеспечение учебного процесса

В качестве технических средств для проведения лабораторных работ используются локальная сеть ПК учебного компьютерного класса. На каждом компьютере должны быть установлены:
1) ОС Windows XP или ОС Linux.
2) Инструментальная программная платформа QNX Momentics IDE 4.7 для разработки приложений реального времени.
3) Виртуальная (VMware Player) QNX-машина.

5 Учебно-методическое обеспечение

5.1 Основная литература

1. Галатенко, Владимир Антонович. Программирование в стандарте POSIX [Текст] : курс лекций : учеб. пособие / В. А. Галатенко ; под ред. В. Б. Бетелина ; Интернет- ун-т информ. технологий. - М. : ИНТУИТ. ру, 2004. - 554 с. - 15 экз.

5.2 Дополнительная литература

1. Баррет, Стивен Ф. Встраиваемые системы. Проектирование приложений на микроконтроллерах семейства 68HC12/HCS12 с применением языка С [Текст] : переводное издание / С. Ф. Баррет, Д. Дж. Пак. - М.: ДМК-пресс, 2007. - 635 с. - 1 экз.
2. Павловская, Татьяна Александровна. С/С++. Программирование на языке

высокого уровня [Текст] : [учеб. для вузов по направлению "Информатика и вычисл. техника"] / Т. А. Павловская. - СПб. и др. : Питер : Питер Принт, 2004. - 460 с. - 20 экз.
3. Подбельский, Вадим Валериевич. Программирование на языке Си [Текст] : Учеб. пособие для вузов / В. В. Подбельский, С. С. Фомин. - 2-е доп. изд. - М. : Финансы и статистика, 2003. - 600 с. - 5 экз.
4. Подбельский, Вадим Валериевич. Практикум по программированию на языке СИ [Текст] : [учеб. пособие для вузов] / В. В. Подбельский. - М.: Финансы и статистика, 2004. - 575 с. - 2 экз.
5. Робачевский, Андрей М. Операционная система UNIX [Текст] : Учеб.пособие для вузов / Андрей Робачевский. - СПб. : БХВ-Петербург, 2002. - 514 с. - 30 экз.

5.3 Электронные источники и интернет ресурсы

Кафедра предоставляет каждому студенту в качестве раздаточного материала в электронном виде файлы в формате *.doc, содержащие учебное пособие по курсу лекций, задания и методические указания по выполнению лабораторных работ.

5.4 Методические указания и рекомендации

На практических занятиях в активной форме обсуждаются теоретические и практические аспекты проектирования приложений реального времени с использованием средств ОСРВ QNX и языка С (как средств промышленного программирования).
Текущий контроль знаний студентов в семестре осуществляется при приеме отчетов по результатам выполнения лабораторных работ, в результате которых студент получает оценку за выполнение лабораторной работы.
Получение итоговой положительной оценки по всем выполненным лабораторным работам является основанием для допуска к экзамену.
Итоговая экзаменационная оценка складывается из оценки по лабораторным работам и оценки, полученной в результате прохождения теоретического теста.
Для получения оценки отлично студенту необходимо получить не менее одной оценки отлично и второй оценки не ниже хорошо.