

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

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

Лекции по курсу «Системное программирование»

Конспект лекций

2011

Автор: КУПРИЯНОВ Александр Викторович

Конспект лекций предназначен для бакалавров направления 010400.68 “Прикладная математика и информатика”.

1. Лекция 1

Понятие о системном программном обеспечении. Задачи, решаемые с использованием ЭВМ. Классификация компьютеров. Виды программного обеспечения.

Программа

Программное обеспечение

Классы ПО

Системное программное обеспечение

Классификация системного программного обеспечения

Классификация компьютеров по области их применения

Классификация компьютеров по быстродействию.

Персональные компьютеры и рабочие станции

X-терминалы

Серверы

Мейнфреймы

Кластерные архитектуры

Структура вычислительной системы

Классификация программ

Встроенные, нерезидентные и резидентные программы

1.1. Программы и программное обеспечение

ПРОГРАММА

— это данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ) в целях реализации определенного алгоритма.

Обратите внимание: программа — это данные.

Один из основных принципов машины фон Неймана — то, что и программы, и данные хранятся в одной и той же памяти. Сохраняемая в памяти программа представляет собой некоторые коды, которые могут рассматриваться как данные. Возможно, с точки зрения программиста программа — активный компонент, она выполняет некоторые действия. Но с точки зрения процессора команды программы — это данные, которые процессор читает и интерпретирует. С другой стороны программа — это данные с точки зрения обслуживающих программ, например, с точки зрения компилятора, который на входе получает одни данные — программу на языке высокого уровня (ЯВУ), а на выходе выдает другие данные — программу в машинных кодах.

Программное обеспечение (ПО) — совокупность программ СОИ и программных документов, необходимых для их эксплуатации.

Существенно, что ПО — это программы, предназначенные для многократного использования и применения разными пользователями. В связи с этим следует обратить внимание на ряд необходимых свойств ПО.

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Программное обеспечение - это то, что оживляет компьютер, который без программного обеспечения был бы нагромождением ненужной и очень дорогой электроники - "железа".

Программы обычно являются последовательным набором команд, сообщаемых компьютеру, что ему делать в той или иной ситуации.

Все программы можно разделить на несколько классов, каждый из которых занимает определенную ступень в иерархической лестнице программного обеспечения.

КЛАССЫ ПО

Традиционно программное обеспечение подразделяют на два класса:

1. системное программное обеспечение (СПО) и
2. прикладное (пользовательское) программное обеспечение (ППО)

Выделим еще один класс (скорее группу) программ - специальное программное обеспечение информационных и управляющих систем.

Прикладные программы предназначены для решения функциональных задач, они выполняют обработку информации различных предметных областей.

Это самый многочисленный класс программных продуктов.

К специальному программному обеспечению информационных и управляющих систем относятся

1. программы (системы) управления базами данных;
2. программы управления языком интерфейса информационных систем;
3. программы сбора и предварительной обработки информации (в информационно-измерительных системах, например, бортовые системы).

ПО этого класса часто оказывается скрытым в составе драйверов оборудования или поставляется в виде библиотек функционального расширения языков программирования.

Поэтому часто такие ПО относят к системному программному обеспечению.

Система управления базами данных (СУБД) - это сложная программная система накопления и последующего манипулирования данными. Каждая СУБД предоставляет интерфейс с базой данных и может располагать средствами непосредственного доступа к последней ее пользователей.

С помощью языка описания данных создаются описания элементов и записей данных, а также взаимосвязей между ними. Для выполнения операций с базой данных (например, выборка или обновление) в прикладных программах используется язык манипулирования данными. Фактическая структура физического хранения данных известна только СУБД.

Системное программное обеспечение (System Software) - совокупность программ и программных комплексов для обеспечения работы компьютера и сетей ЭВМ.

СПО управляет ресурсами компьютерной системы и позволяет пользователям программировать в более выразительных языках, чем машинный язык компьютера. Состав СПО мало зависит от характера решаемых задач пользователя.

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

предназначено для:

1. создания операционной среды функционирования других программ (другими словами, для организации выполнения программ);
2. автоматизации разработки (создания) новых программ;
3. обеспечения надежной и эффективной работы самого компьютера и вычислительной сети;
4. проведения диагностики и профилактики аппаратуры компьютера и вычислительных сетей;
5. выполнения вспомогательных технологических процессов (копирование, архивирование, восстановление файлов программ и баз данных и т.д.).

Данный класс программных продуктов тесно связан с типом компьютера и является его неотъемлемой частью.

Программные продукты данного класса в основном ориентированы на квалифицированных пользователей - профессионалов в компьютерной области: системного программиста, администратора сети, прикладного программиста, оператора.

Однако знание базовой технологии работы с этим классом программных продуктов требуется и конечным пользователям персонального компьютера, которые самостоятельно не только работают со своими программами, но и выполняют обслуживание компьютера, программ и данных.

Программные продукты данного класса носят общий характер применения, независимо от специфики предметной области.

К системным программным продуктам предъявляются высокие требования по надежности и технологичности работы, удобству и эффективности использования.

КЛАССИФИКАЦИЯ СИСТЕМНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

СПО традиционно включают системные управляющие и системные обрабатывающие программы.

Управляющие системные программы организуют корректное функционирование всех устройств системы.

Основные системные функции управляющих программ -

управление вычислительными процессами и вычислительными комплексами и
работа с внутренними данными ОС.

Как правило, они находятся в основной памяти. Это резидентные программы, составляющие ядро ОС. Управляющие программы, которые загружаются в память непосредственно перед выполнением, называются транзитными (transitive).

В настоящее время системные управляющие программы поставляются фирмами-разработчиками и фирмами-дистрибьюторами в виде инсталляционных пакетов операционных систем и драйверов специальных устройств.

Обрабатывающие системные программы выполняются как специальные прикладные задачи, или приложения.

Эти программы поставляются чаще в виде дистрибутивных пакетов, включающих ПО

Замечание. В пакеты системных программ помимо основных программ, допускающих реконфигурацию, входят специальные настроечные программы, называемые программами инсталляции. Мы будем создавать и такие программы.

1.2. Классификация компьютеров

КЛАССИФИКАЦИЯ КОМПЬЮТЕРОВ ПО ОБЛАСТИ ИХ ПРИМЕНЕНИЯ

- A. Персональные компьютеры и рабочие станции.
- B. X-терминалы
- C. Серверы
- D. Мэйнфреймы
- E. Кластерные архитектуры

Во все времена людям нужно было считать. В туманном доисторическом прошлом они считали на пальцах или делали насечки на костях. Примерно около 4000 лет назад, на заре человеческой цивилизации, были изобретены уже довольно сложные системы счисления, позволявшие осуществлять торговые сделки, рассчитывать астрономические циклы, проводить другие вычисления. Несколько тысячелетий спустя, появились первые ручные вычислительные инструменты. А в наши дни невозможно представить решение сложных вычислительных задач и выполнение операций, казалось бы, не связанных с числами, без помощи "электронного мозга", называемого компьютером, или, по старинке, ЭВМ.

Специалисты (а таковыми являются в наш век все подростки старше десяти лет), не преминут заметить, что компьютер - это не мозг (по крайней мере пока - уточнят особенно талантливые). Это просто-напросто еще один инструмент, еще одно устройство, придуманное для того, чтобы облегчить наш труд или усилить нашу власть над природой. При всем своем кажущемся великолепии компьютер обладает, по существу, одним-единственным талантом - реагировать с молниеносной быстротой на импульсы электрического напряжения. Истинное величие заключено в человеке, его гении, который нашел способ преобразовывать разнообразную информацию, поступающую из реального мира, в последовательность нулей и единиц, переводить все многообразие нашей не подчиняющейся строгим математическим законам жизни в строгий язык математики, понятный электронным схемам компьютера. Именно поэтому я и решил, что моя будущая специальность будет связана с компьютерами: вопреки опасениям писателей-фантастов, человек не стал придатком машины, а получил возможность лучше проявить свои способности и сделать еще один шаг от чисто механического труда, к творческому. Мы вступаем в постиндустриальный век, век информатики.

КЛАССИФИКАЦИЯ КОМПЬЮТЕРОВ ПО БЫСТРОДЕЙСТВИЮ.

Первые компьютеры были созданы почти полвека назад, и, хотя они и занимали тогда целые залы, их быстродействие было сравнимо с быстродействием нынешних "персоналок"; они вполне удовлетворяли потребностям крупных министерств и корпораций. Тем не менее, миниатюризация и стремление сделать компьютеры достоянием масс привели к тому, что в двадцать первый век мы входим не под знаменами из перфокарт, на которых изображены гигантские вычислительные центры, а размахивая красочными обложками компьютерных журналов, написанных простым языком, понятным миллионам пользователей персональных компьютеров.

Почему же компьютеры стали такими популярными и превратились из религии одиночек в орудие производства миллионов? Почему они с успехом решают поставленные перед ними и удовлетворяют придирчивые требования ученых и соответствуют изысканному вкусу секретарш? Для того, чтобы ответить на эти и другие вопросы, я расскажу о компьютерах то, что считаю нужным, стараясь не углубляться в технические подробности, и, прошу прощения, не очень заботясь о логичности повествования.

Для начала - история. В пятидесятых - шестидесятых годах фирмы производящие компьютеры, которые были тогда доступны лишь крупным компаниям и учреждениям из-за своих размеров и цены, стали стремиться производить компьютеры, которые были бы меньше и дешевле, чем у их конкурентов. Это делалось в борьбе за покупателей, в борьбе за увеличение объемов продаж. (Вот они - преимущества капиталистического общества, могло ли такое быть у нас в стране? Нет, у нас делали лишь большие компьютеры - БЭСМы, и маленькие большие компьютеры - МЭСМы.) Так вот, благодаря изобретению транзисторов, памяти на магнитных сердечниках, миниатюризации внешних устройств, и, наконец, появлению интегральных схем, стало возможным появление в 1965 году мини-компьютера PDP-8 размером с холодильник и стоимостью 20 тысяч долларов. Дальше - больше (в смысле меньше). В конце 1970 года был выпущен в продажу первый микропроцессор Intel-4004 - интегральная схема, интегральная схема, аналогичная по своим функциям центральному процессору большой ЭВМ. Вслед за этим четырехбитным (скажи сейчас какому-нибудь ребенку фразу "четыре бита" - он будет долго смеяться и заявит : "SEGA Mega Drive II - 16 бит! Это - класс!"), появились восьми битные (то есть уже почти как "Денди"!) модели 8008 и 8080, которые до конца семидесятых стали стандартом компьютерной индустрии.

Вначале эти процессоры использовались только электронщиками-любителями и в различных специальных устройствах, но с 1975 года, на основе Intel-8080, стали серийно производиться первые персональные компьютеры. Объемы их продаж были невысоки, но для абсолютно нового товара коммерческий успех был ошеломляющ. Что же было причиной этого? Уже тогда, на заре, успех "железа" зависел от программного обеспечения. Так "хитами" в те годы стали языки программирования для "чайников" Basic, текстовый редактор WordStar (назначения "горячих" клавиш которого используются до сих пор) и табличный процессор VisiCalc, переросший к настоящему времени в гиганта под названием Excel. Деловой мир всего мира увидел, что покупать компьютеры весьма выгодно: с их помощью стало возможно значительно эффективнее выполнять бухгалтерские расчеты, составлять документы и так далее. В результате оказалось, что для многих организаций необходимые им расчеты можно выполнять не на больших ЭВМ, а на персональных компьютерах, что значительно дешевле.

Распространение персональных компьютеров к концу семидесятых годов привело к некоторому снижению спроса на большие и мини ЭВМ. Это стало предметом серьезного беспокойства корпорации IBM - ведущей компании по производству таковых. В 1979 году руководство фирмы решило произвести как бы мелкий эксперимент (что-то вроде одной из десятков проводившихся в фирме работ по созданию нового оборудования) - попробовать свои силы на рынке персональных компьютеров. Чтобы на этот эксперимент слишком много денег, руководство фирмы предоставило подразделению, ответственному за данный проект, невиданную в фирме свободу. В частности, ему было разрешено не конструировать персональный компьютер "с нуля", а использовать блоки, изготовленные другими фирмами. И это подразделение сполна использовало предоставленный шанс. Прежде всего в качестве основного микропроцессора компьютера был выбран новейший тогда 16-разрядный микропроцессор Intel-8088. Его использование позволило значительно увеличить потенциальные возможности компьютера, так как новый микропроцессор позволял работать с 1Мбайтом памяти, а все имевшиеся тогда компьютеры были ограничены 64Кбайтами. В компьютере были использованы и другие комплектующие различных фирм, а его программное обеспечение было поручено разработать небольшой фирме Microsoft.

В августе 1981 года новый компьютер под названием IBM PC был официально представлен публике и вскоре после этого он приобрел большую популярность у пользователей. Через один-два года компьютер IBM PC стал стандартом персонального компьютера. Сейчас такие компьютеры ("совместимые с IBM PC") составляют около 90% всех производимых в мире персональных компьютеров. В чем же причина их феноменального успеха? Если бы IBM PC был сделан так же, как другие существовавшие тогда компьютеры, он бы устарел через два-три года и мы бы все уже давно о нем забыли. Но к счастью, в IBM PC была заложена возможность усовершенствования его отдельных частей и использования новых устройств. Фирма IBM сделала компьютер не единым неразъемным устройством, а обеспечила возможность его сборки из независимо изготовленных

частей, подобно детскому конструктору. При этом методы сопряжения устройств с компьютером IBM PC не только не держались в секрете, но и были доступны всем желающим. Действительно, персональный компьютер очень напоминает обыкновенный конструктор: схемы, управляющие всеми устройствами - монитором, дисками, принтером, модемом и т.д., реализованы на отдельных платах, которые вставляются в стандартные разъемы системной платы - слоты. Весь компьютер питается от единого блока питания. Этот принцип, названный принципом открытой архитектуры, наряду с другими достоинствами обеспечил потрясающий успех персональному компьютеру IBM.

В настоящее время индустрия производства компьютеров и программного обеспечения для них является одной из наиболее важных сфер экономики развитых стран. Ежегодно в мире продаются десятки миллионов компьютеров и еще больше программ для них. Крупные производители компьютерной техники вкладывают миллиарды долларов в научно-исследовательские разработки, а бюджеты компьютерных игр превосходят бюджеты Голливудских фильмов. Отрасли, связанные с компьютерами самые быстроразвивающиеся и прибыльные. В чем же причина такого стремительного роста индустрии персональных компьютеров? Одна из причин - их невысокая стоимость (как правило, от нескольких сотен до десяти тысяч долларов) и их сравнительная выгодность для многих деловых применений по сравнению с большими ЭВМ. Но есть и другие:

" простота использования, обеспеченная диалоговым и интерактивным взаимодействием с программами, их удобным интерфейсом (меню, пиктограммы, всегда доступная подробная "помощь" и так далее);

" истинная "персональность" компьютера, то есть возможность взаимодействия с ним без посредников и ограничений;

" относительно высокие возможности по переработке информации (скорость вычислений - несколько миллионов операций в секунду, большая емкость оперативной памяти и магнитных носителей);

" высокая надежность и простота ремонта, основанные на интеграции компонентов компьютера;

" возможность расширения и адаптации к особенностям применения компьютеров - один и тот же компьютер может быть оснащен различными периферийными устройствами и разным программным обеспечением;

" наличие программного обеспечения, охватывающего почти все сферы человеческой деятельности, а также мощных систем для разработки новых программ.

Как уже было сказано, персональный компьютер представляет собой всего лишь сложный конструктор, и мне кажется, что еще долгие годы все человечество будет с упоением играть в него.

ПЕРСОНАЛЬНЫЕ КОМПЬЮТЕРЫ И РАБОЧИЕ СТАНЦИИ

Персональные компьютеры (ПК) появились в результате эволюции миникомпьютеров при переходе элементной базы машин с малой и средней степенью интеграции на большие и сверхбольшие интегральные схемы. ПК, благодаря своей низкой стоимости, очень быстро завоевали хорошие позиции на компьютерном рынке и создали предпосылки для разработки новых программных средств, ориентированных на конечного пользователя. Это прежде всего - "дружественные пользовательские интерфейсы", а также проблемно-ориентированные среды и инструментальные средства для автоматизации разработки прикладных программ.

Миникомпьютеры стали прародителями и другого направления развития современных систем - 32-разрядных машин. Создание RISC-процессоров и микросхем памяти емкостью более 1 Мбит привело к окончательному оформлению настольных систем высокой производительности, которые сегодня известны как рабочие станции. Первоначальная ориентация рабочих станций на профессиональных пользователей (в отличие от ПК, которые в начале ориентировались на самого широкого потребителя непрофессионала) привела к тому, что рабочие станции - это хорошо сбалансированные системы, в которых высокое быстродействие сочетается с большим объемом оперативной и внешней памяти, высокопроизводительными внутренними магистралями, высококачественной и быстродействующей графической подсистемой и разнообразными устройствами ввода/вывода. Это свойство выгодно отличает рабочие станции среднего и высокого класса от ПК и сегодня. Даже наиболее мощные IBM PC совместимые ПК не в состоянии удовлетворить возрастающие потребности систем обработки из-за наличия в их архитектуре ряда "узких мест".

Тем не менее, быстрый рост производительности ПК на базе новейших микропроцессоров Intel в сочетании с резким снижением цен на эти изделия и развитием технологии локальных шин (VESA и PCI), позволяющей устранить многие "узкие места" в архитектуре ПК, делают современные персональные компьютеры весьма привлекательной альтернативой рабочим станциям. В свою очередь производители рабочих станций создали изделия так называемого "начального уровня", которые по стоимостным характеристикам близки к высокопроизводительным ПК, но все еще сохраняют лидерство по производительности и возможностям наращивания. Насколько успешно удастся ПК на базе процессоров и

Pentium бороться против рабочих станций UNIX, покажет будущее, но уже в настоящее время появилось понятие "персональной рабочей станции", которое объединяет оба направления.

Современный рынок "персональных рабочих станций" не просто определить. По сути он представляет собой совокупность архитектурных платформ персональных компьютеров и рабочих станций, которые появились в настоящее время, поскольку поставщики компьютерного оборудования уделяют все большее внимание рынку продуктов для коммерции и бизнеса. Этот рынок традиционно считался вотчиной миникомпьютеров и мэйнфреймов, которые поддерживали работу настольных терминалов с ограниченным интеллектом. В прошлом персональные компьютеры не были достаточно мощными и не располагали достаточными функциональными возможностями, чтобы служить адекватной заменой подключенных к главной машине терминалов. С другой стороны, рабочие станции на платформе UNIX были очень сильны в научном, техническом и инженерном секторах и были почти также неудобны, как и ПК для того чтобы выполнять серьезные офисные приложения. С тех пор ситуация изменилась коренным образом. Персональные компьютеры в настоящее время имеют достаточную производительность, а рабочие станции на базе UNIX имеют программное обеспечение, способное выполнять большинство функций, которые стали ассоциироваться с понятием "персональной рабочей станции". Вероятно, оба этих направления могут серьезно рассматриваться в качестве сетевого ресурса для систем масштаба предприятия. В результате этих изменений практически ушли со сцены старомодные миникомпьютеры с их патентованной архитектурой и использованием присоединяемых к главной машине терминалов. По мере продолжения процесса разукрупнения (downsizing) и увеличения производительности платформы Intel наиболее мощные ПК (но все же чаще открытые системы на базе UNIX) стали использоваться в качестве серверов, постепенно заменяя миникомпьютеры.

Среди других факторов, способствующих этому процессу, следует выделить:

" Применение ПК стало более разнообразным. Помимо обычных для этого класса систем текстовых процессоров, даже средний пользователь ПК может теперь работать сразу с несколькими прикладными пакетами, включая электронные таблицы, базы данных и высококачественную графику.

" Адаптация графических пользовательских интерфейсов существенно увеличила требования пользователей ПК к соотношению производительность/стоимость. И хотя оболочка MS Windows может работать на моделях ПК 386SX с 2 Мбайтами оперативной памяти, реальные пользователи хотели бы использовать все преимущества подобных систем, включая возможность комбинирования и эффективного использования различных пакетов.

" Широкое распространение систем мультимедиа прямо зависит от возможности использования высокопроизводительных ПК и рабочих станций с адекватными аудио и графическими средствами, и объемами оперативной и внешней памяти.

" Слишком высокая стоимость мэйнфреймов и даже систем среднего класса помогла сместить многие разработки в область распределенных систем и систем клиент-сервер, которые многим представляются вполне оправданной по экономическим соображениям альтернативой. Эти системы прямо базируются на высоконадежных и мощных рабочих станциях и серверах.

В начале представлялось, что необходимость сосредоточения высокой мощности на каждом рабочем месте приведет к переходу многих потребителей ПК на UNIX-станции. Это определялось частично тем, что RISC-процессоры, использовавшиеся в рабочих станциях на базе UNIX, были намного более производительными по сравнению с CISC-процессорами, применявшимися в ПК, а частично мощностью системы UNIX по сравнению с MS-DOS и даже OS/2.

Производители рабочих станций быстро отреагировали на потребность в низко стоимостных моделях для рынка коммерческих приложений. Потребность в высокой мощности на рабочем столе, объединенная с желанием поставщиков UNIX-систем продавать как можно больше своих изделий, привела такие компании как "Sun Microsystems" и "Hewlett Packard" на рынок рабочих станций для коммерческих приложений. И хотя значительная часть систем этих фирм все еще ориентирована на технические приложения, наблюдается беспрецедентный рост продаж продукции этих компаний для работы с коммерческими приложениями, требующими все большей и большей мощности для реализации сложных, сетевых прикладных систем, включая системы мультимедиа.

Это привело к временному отступлению производителей ПК на базе микропроцессоров Intel. Острая конкуренция со стороны производителей UNIX-систем и потребности в повышении производительности огромной уже инсталлированной базы ПК, заставили компанию Intel форсировать разработку высокопроизводительных процессоров семейства и Pentium. Процессоры и Pentium, при разработке которого были использованы многие подходы, применявшиеся ранее только в RISC-процессорах, а также использование других технологических усовершенствований, таких как архитектура локальной шины, позволили снабдить ПК достаточной мощностью, чтобы составить конкуренцию рабочим станциям во многих направлениях рынка коммерческих приложений. Правда, для многих других приложений, в частности, в области сложного графического моделирования, ПК все еще сильно отстают.

Х-ТЕРМИНАЛЫ

Х-терминалы представляют собой комбинацию бездисковых рабочих станций и стандартных ASCII-терминалов. Бездисковые рабочие станции часто применялись в качестве дорогих дисплеев и в этом случае не полностью использовали локальную вычислительную мощь. Одновременно многие пользователи ASCII-терминалов хотели улучшить их характеристики, чтобы получить возможность работы в многооконной системе и графические возможности. Совсем недавно, как только стали доступными очень мощные графические рабочие станции, появилась тенденция применения "подчиненных" Х-терминалов, которые используют рабочую станцию в качестве локального сервера.

На компьютерном рынке Х-терминалы занимают промежуточное положение между персональными компьютерами и рабочими станциями. Поставщики Х-терминалов заявляют, что их изделия более эффективны в стоимостном выражении, чем рабочие станции высокого ценового класса, и предлагают увеличенный уровень производительности по сравнению с персональными компьютерами. Быстрое снижение цен, прогнозируемое иногда в секторе Х-терминалов, в настоящее время идет, очевидно, благодаря обострившейся конкуренции в этом секторе рынка. Многие компании начали активно конкурировать за распределение рынка, а быстрый рост объемных поставок создал предпосылки для создания такого рынка. В настоящее время уже достигнута цена в \$1000 для Х-терминалов начального уровня, что делает эту технологию доступной для широкой пользовательской базы.

Как правило, стоимость Х-терминалов составляет около половины стоимости сравнимой по конфигурации бездисковой машины и примерно четверть стоимости полностью оснащенной рабочей станции.

Что такое Х-терминал?

Типовой Х-терминал включает следующие элементы:

" Экран высокого разрешения - обычно размером от 14 до 21 дюйма по диагонали;

" Микропроцессор на базе Motorola 68xxx или RISC-процессор типа Intel i960, MIPS R3000 или AMD29000; " Отдельный графический сопроцессор в дополнение к основному процессору, поддерживающий двухпроцессорную архитектуру, которая обеспечивает более быстрое рисование на экране и прокручивание экрана; " Базовые системные программы, на которых работает система X-Windows и выполняются сетевые протоколы; " Программное обеспечение сервера X11; " Переменный объем локальной памяти (от 2 до 8 Мбайт) для дисплея, сетевого интерфейса, поддерживающего TCP/IP и другие сетевые протоколы. " Порты для подключения клавиатуры и мыши.

Х-терминалы отличаются от ПК и рабочих станций не только тем, что не выполняют функции обычной локальной обработки. Работа Х-терминалов зависит от главной (хост) системы, к которой они подключены посредством сети. Для того, чтобы Х-терминал мог работать, пользователи должны установить программное обеспечение многооконного сервера X11 на главном процессоре, выполняющим прикладную задачу (наиболее известная версия X11 Release 5). Х-терминалы отличаются также от стандартных алфавитно-цифровых ASCII и традиционных графических дисплейных терминалов тем, что они могут быть подключены к любой главной системе, которая поддерживает стандарт X-Windows. Более того, локальная вычислительная мощь Х-терминала обычно используется для обработки отображения, а не обработки приложений (называемых клиентами), которые выполняются удаленно на главном компьютере (сервере). Вывод такого удаленного приложения просто отображается на экране Х-терминала.

Минимальный объем требуемой для работы памяти Х-терминала составляет 1 Мбайт, но чаще 2 Мбайта. В зависимости от функциональных возможностей изделия оперативная память может расширяться до 32 Мбайт и более.

Оснащенный стандартной системой X-Windows, Х-терминал может отображать на одном и том же экране множество приложений одновременно. Каждое приложение может выполняться в своем окне, и пользователь может изменять размеры окон, их месторасположение и манипулировать ими в любом месте экрана.

X-Windows - результат совместной работы Массачусетского технологического института (MIT) и корпорации DEC. Система X-Windows (известная также под именем X) в настоящее время является открытым де-факто стандартом для доступа к множеству одновременно выполняющихся приложений с возможностями многооконного режима и графикой высокого разрешения на интеллектуальных терминалах, персональных компьютерах, рабочих станциях и Х-терминалах. Она стала стандартом для обеспечения интероперабельности (переносимости) продуктов многих поставщиков и для организации доступа к множеству приложений. В настоящее время X-Windows является стандартом для разработки пользовательского интерфейса. Более 90% поставщиков UNIX-рабочих станций и многие поставщики персональных компьютеров адаптировали систему X-Windows и применяют в качестве стандарта.

СЕРВЕРЫ

Прикладные многопользовательские коммерческие и бизнес-системы, включающие системы управления базами данных и обработки транзакций, крупные издательские системы, сетевые приложения и системы обслуживания коммуникаций, разработку программного обеспечения и обработку изображений все более настойчиво требуют перехода к модели вычислений "клиент-сервер" и распределенной обработке. В распределенной модели "клиент-сервер" часть работы выполняет сервер, а часть пользовательский компьютер (в общем случае клиентская и пользовательская части могут работать и на одном компьютере). Существует несколько типов серверов, ориентированных на разные применения: файл-сервер, сервер базы данных, принт-сервер, вычислительный сервер, сервер приложений. Таким образом, тип сервера определяется видом ресурса, которым он владеет (файловая система, база данных, принтеры, процессоры или прикладные пакеты программ).

С другой стороны существует классификация серверов, определяющаяся масштабом сети, в которой они используются: сервер рабочей группы, сервер отдела или сервер масштаба предприятия (корпоративный сервер). Эта классификация весьма условна. Например, размер группы может меняться в диапазоне от нескольких человек до нескольких сотен человек, а сервер отдела обслуживать от 20 до 150 пользователей. Очевидно, в зависимости от числа пользователей и характера решаемых ими задач, требования к составу оборудования и программного обеспечения сервера, к его надежности и производительности сильно варьируются.

Файловые серверы небольших рабочих групп (не более 20-30 человек) проще всего реализуются на платформе персональных компьютеров и программном обеспечении Novell NetWare. Файл-сервер, в данном случае, выполняет роль центрального хранилища данных. Серверы прикладных систем и высокопроизводительные машины для среды "клиент-сервер" значительно отличаются требованиями к аппаратным и программным средствам.

Типичными для небольших файл-серверов являются: процессор Pentium или более быстродействующий, 32-Мбайт ОЗУ, 9 Гбайт дискового пространства и один адаптер Ethernet 10BaseT, имеющий быстродействие 10 Мбит/с. В состав таких серверов часто включаются флоппи-дискковод и CDRом. Графика для большинства серверов несущественна, поэтому достаточно иметь обычный монохромный монитор с разрешением VGA.

Скорость процессора для серверов с интенсивным вводом/выводом не критична. Они должны быть оснащены достаточно мощными блоками питания для возможности установки дополнительных плат расширения и дисковых накопителей. Желательно применение устройства бесперебойного питания. Оперативная память обычно имеет объем не менее 32 Мбайт, что позволит операционной системе (например, NetWare) использовать большие дисковые кэши и увеличить производительность сервера. Как правило, для работы с многозадачными операционными системами такие серверы оснащаются интерфейсом SCSI (или Fast SCSI). Распределение данных по нескольким жестким дискам может значительно повысить производительность.

При наличии одного сегмента сети и 10-20 рабочих станций пиковая пропускная способность сервера ограничивается максимальной пропускной способностью сети. В этом случае замена процессоров или дисковых подсистем более мощными не увеличивают производительность, так как узким местом является сама сеть. Поэтому важно использовать хорошую плату сетевого интерфейса.

Хотя влияние более быстрого процессора явно на производительности не сказывается, оно заметно снижает коэффициент использования ЦП. Во многих серверах этого класса используются процессоры Pentium, с тактовой частотой 166 и 200 МГц, microSPARC-II и PowerPC. Аналогично процессорам влияние типа системной шины (EISA со скоростью 33 Мбит/с или PCI со скоростью 132 Мбит/с) также минимально при таком режиме использования.

Однако для файл-серверов общего доступа, с которыми одновременно могут работать несколько десятков, а то и сотен человек, простой однопроцессорной платформы и программного обеспечения Novell может оказаться недостаточно. В этом случае используются мощные многопроцессорные серверы с возможностями наращивания оперативной памяти до нескольких гигабайт, дискового пространства до сотен гигабайт, быстрыми интерфейсами дискового обмена (типа Fast SCSI-2, Fast&Wide SCSI-2 и Fiber Channel) и несколькими сетевыми интерфейсами. Эти серверы используют операционную систему UNIX, сетевые протоколы TCP/IP и NFS. На базе многопроцессорных UNIX-серверов обычно строятся также серверы баз данных крупных информационных систем, так как на них ложится основная нагрузка по обработке информационных запросов. Подобного рода серверы получили название суперсерверов.

По уровню общесистемной производительности, функциональным возможностям отдельных компонентов, отказоустойчивости, а также в поддержке многопроцессорной обработки, системного администрирования и дисковых массивов большой емкости суперсерверы вышли в настоящее время на один уровень с мейнфреймами и мощными миникомпьютерами. Современные суперсерверы характеризуются:

" наличием двух или более центральных процессоров RISC, либо Pentium , либо Pentium II;

" многоуровневой шинной архитектурой, в которой запатентованная высокоскоростная системная шина связывает между собой несколько процессоров и оперативную память, а также множество стандартных шин ввода/вывода, размещенных в том же корпусе;

" поддержкой технологии дисковых массивов RAID;

" поддержкой режима симметричной многопроцессорной обработки, которая позволяет распределять задания по нескольким центральным процессорам или режима асимметричной многопроцессорной обработки, которая допускает выделение процессоров для выполнения конкретных задач.

Как правило, суперсерверы работают под управлением операционных систем UNIX, а в последнее время и Windows NT (на Digital 2100 Server Model A500MP), которые обеспечивают многопоточную многопроцессорную и многозадачную обработку. Суперсерверы должны иметь достаточные возможности наращивания дискового пространства и вычислительной мощности, средства обеспечения надежности хранения данных и защиты от несанкционированного доступа. Кроме того, в условиях быстро растущей организации, важным условием является возможность наращивания и расширения уже существующей системы.

МЕЙНФРЕЙМЫ

Мейнфрейм - это синоним понятия "большая универсальная ЭВМ". Мейнфреймы и до сегодняшнего дня остаются наиболее мощными (не считая суперкомпьютеров) вычислительными системами общего назначения, обеспечивающими непрерывный круглосуточный режим эксплуатации. Они могут включать один или несколько процессоров, каждый из которых, в свою очередь, может оснащаться векторными сопроцессорами (ускорителями операций с суперкомпьютерной производительностью). В нашем сознании мейнфреймы все еще ассоциируются с большими по габаритам машинами, требующими специально оборудованных помещений с системами водяного охлаждения и кондиционирования. Однако это не совсем так. Прогресс в области элементно-конструкторской базы позволил существенно сократить габариты основных устройств. Наряду со сверхмощными мейнфреймами, требующими организации двухконтурной водяной системы охлаждения, имеются менее мощные модели, для охлаждения которых достаточно принудительной воздушной вентиляции, и модели, построенные по блочно-модульному принципу и не требующие специальных помещений и кондиционеров.

Основными поставщиками мейнфреймов являются известные компьютерные компании IBM, Amdahl, ICL, Siemens, Nixdorf и некоторые другие, но ведущая роль принадлежит безусловно компании IBM. Именно архитектура системы IBM/360, выпущенной в 1964 году, и ее последующие поколения стали образцом для подражания. В нашей стране в течение многих лет выпускались машины ряда ЕС ЭВМ, являвшиеся отечественным аналогом этой системы.

В архитектурном плане, мейнфреймы представляют собой многопроцессорные системы, содержащие один или несколько центральных и периферийных процессоров с общей памятью, связанных между собой высокоскоростными магистралями передачи данных. При этом основная вычислительная нагрузка ложится на центральные процессоры, а периферийные процессоры (в терминологии IBM - селекторные, блок-мультиплексные, мультиплексные каналы и процессоры телеобработки) обеспечивают работу с широкой номенклатурой периферийных устройств.

Первоначально мейнфреймы ориентировались на централизованную модель вычислений, работали под управлением патентованных операционных систем и имели ограниченные возможности для объединения в единую систему оборудования различных фирм-поставщиков. Однако повышенный интерес потребителей к открытым системам, построенным на базе международных стандартов и позволяющим достаточно эффективно использовать все преимущества такого подхода, заставил поставщиков мейнфреймов существенно расширить возможности своих операционных систем в направлении совместимости. В настоящее время они демонстрируют свою "открытость", обеспечивая соответствие со спецификациями POSIX 1003.3, возможность использования протоколов меж соединений OSI и TCP/IP или предоставляя возможность работы на своих компьютерах под управлением операционной системы UNIX собственной разработки.

Стремительный рост производительности персональных компьютеров, рабочих станций и серверов создал тенденцию перехода с мейнфреймов на компьютеры менее дорогих классов: миникомпьютеры и многопроцессорные серверы. Эта тенденция получила название "разукрупнение" (downsizing). Однако этот процесс в последнее время несколько замедлился. Основной причиной возрождения интереса к мейнфреймам эксперты считают сложность перехода к распределенной архитектуре клиент-сервер, которая оказалась выше, чем предполагалось. Кроме того, многие пользователи считают, что распределенная среда не обладает достаточной надежностью для наиболее ответственных приложений, которой обладают мейнфреймы.

Очевидно выбор центральной машины (сервера) для построения информационной системы предприятия возможен только после глубокого анализа проблем, условий и требований конкретного заказчика и долгосрочного прогнозирования развития этой системы.

Главным недостатком мейнфреймов в настоящее время остается относительно низкое соотношение производительность/стоимость. Однако фирмами-поставщиками мейнфреймов предпринимаются значительные усилия по улучшению этого показателя.

Следует также помнить, что в мире существует огромная инсталлированная база мейнфреймов, на которой работают десятки тысяч прикладных программных систем. Отказаться от годами наработанного программного обеспечения просто не разумно. Поэтому в настоящее время ожидается рост продаж мейнфреймов по крайней мере до конца этого столетия. Эти системы, с одной стороны, позволят модернизировать существующие системы, обеспечив сокращение эксплуатационных расходов, с другой стороны, создадут новую базу для наиболее ответственных приложений.

КЛАСТЕРНЫЕ АРХИТЕКТУРЫ

Двумя основными проблемами построения вычислительных систем для критически важных приложений, связанных с обработкой транзакций, управлением базами данных и обслуживанием телекоммуникаций, являются обеспечение высокой производительности и продолжительного функционирования систем. Наиболее эффективный способ достижения заданного уровня производительности - применение параллельных масштабируемых архитектур. Задача обеспечения продолжительного функционирования системы имеет три составляющих: надежность, готовность и удобство обслуживания. Все эти три составляющих предполагают, в первую очередь, борьбу с неисправностями системы, порождаемыми отказами и сбоями в ее работе. Эта борьба ведется по всем трем направлениям, которые взаимосвязаны и применяются совместно.

Повышение надежности основано на принципе предотвращения неисправностей путем снижения интенсивности отказов и сбоев за счет применения электронных схем и компонентов с высокой и сверхвысокой степенью интеграции, снижения уровня помех, облегченных режимов работы схем, обеспечение тепловых режимов их работы, а также за счет совершенствования методов сборки аппаратуры. Повышение уровня готовности предполагает подавление в определенных пределах влияния отказов и сбоев на работу системы с помощью средств контроля и коррекции ошибок, а также средств автоматического восстановления вычислительного процесса после проявления неисправности, включая аппаратную и программную избыточность, на основе которой реализуются различные варианты отказоустойчивых архитектур. Повышение готовности есть способ борьбы за снижение времени простоя системы. Основные эксплуатационные характеристики системы существенно зависят от удобства ее обслуживания, в частности от ремонтпригодности, контроле пригодности и т.д.

В последние годы в литературе по вычислительной технике все чаще употребляется термин "системы высокой готовности" (High Availability Systems). Все типы систем высокой готовности имеют общую цель - минимизацию времени простоя. Имеется два типа времени простоя компьютера: плановое и неплановое. Минимизация каждого из них требует различной стратегии и технологии. Плановое время простоя обычно включает время, принятое руководством, для проведения работ по модернизации системы и для ее обслуживания. Неплановое время простоя является результатом отказа системы или компонента. Хотя системы высокой готовности возможно больше ассоциируются с минимизацией неплановых простоев, они оказываются также полезными для уменьшения планового времени простоя.

Существует несколько типов систем высокой готовности, отличающиеся своими функциональными возможностями и стоимостью. Следует отметить, что высокая готовность не дается бесплатно. Стоимость систем высокой готовности на много превышает стоимость обычных систем. Вероятно поэтому, наибольшее распространение в мире получили кластерные системы, благодаря тому, что они обеспечивают достаточно высокий уровень готовности систем при относительно низких затратах. Термин "кластеризация" на сегодня в компьютерной промышленности имеет много различных значений. Строгое определение могло бы звучать так: "реализация объединения машин, представляющегося единым целым для операционной системы, системного программного обеспечения, прикладных программ и пользователей". Машин, кластеризованные вместе таким способом могут при отказе одного процессора очень быстро перераспределить работу на другие процессоры внутри кластера. Это, возможно, наиболее важная задача многих поставщиков систем высокой готовности.

Первой концепцию кластерной системы анонсировала компания DEC, определив ее как группу объединенных между собой вычислительных машин, представляющих собой единый узел обработки информации. По существу VAX-кластер представляет собой слабосвязанную многомашинную систему с общей внешней памятью, обеспечивающую единый механизм управления и администрирования. В настоящее время на смену VAX-кластерам приходят UNIX-кластеры. При этом VAX-кластеры предлагают проверенный набор решений, который устанавливает критерии для оценки подобных систем.

VAX-кластер обладает следующими свойствами:

Разделение ресурсов. Компьютеры VAX в кластере могут разделять доступ к общим дисковым накопителям. Все компьютеры VAX в кластере могут обращаться к отдельным файлам данных как к локальным.

Высокая готовность. Если происходит отказ одного из VAX-компьютеров, задания его пользователей автоматически могут быть перенесены на другой компьютер кластера. Если в системе имеется несколько контроллеров внешних накопителей и один из них отказывает, другие контроллеры автоматически подхватывают его работу.

Высокая пропускная способность. Ряд прикладных систем могут пользоваться возможностью параллельного выполнения заданий на нескольких компьютерах кластера.

Удобство обслуживания системы. Общие базы данных могут обслуживаться с единственного места. Прикладные программы могут устанавливаться только однажды на общих дисках кластера и разделяться между всеми компьютерами кластера.

Расширяемость. Увеличение вычислительной мощности кластера достигается подключением к нему дополнительных VAX-компьютеров. Дополнительные накопители на магнитных дисках становятся доступными для всех компьютеров, входящих в кластер.

Работа любой кластерной системы определяется двумя главными компонентами: высокоскоростным механизмом связи процессоров между собой и системным программным обеспечением, которое обеспечивает клиентам прозрачный доступ к системному сервису.

В настоящее время широкое распространение получила также технология параллельных баз данных. Эта технология позволяет множеству процессоров разделять доступ к единственной базе данных. Распределение заданий по множеству процессорных ресурсов и параллельное их выполнение позволяет достичь более высокого уровня пропускной способности транзакций, поддерживать большее число одновременно работающих пользователей и ускорить выполнение сложных запросов. Существуют три различных типа архитектуры, которые поддерживают параллельные базы данных:

" Симметричная многопроцессорная архитектура с общей памятью (Shared Memory SMP Architecture). Эта архитектура поддерживает единую базу данных, работающую на многопроцессорном сервере под управлением одной операционной системы. Увеличение производительности таких систем обеспечивается наращиванием числа процессоров, устройств оперативной и внешней памяти.

" Архитектура с общими (разделяемыми) дисками (Shared Disk Architecture). Это типичный случай построения кластерной системы. Эта архитектура поддерживает единую базу данных при работе с несколькими компьютерами, объединенными в кластер (обычно такие компьютеры называются узлами кластера), каждый из которых работает под управлением своей копии операционной системы. В таких системах все узлы разделяют доступ к общим дискам, на которых собственно и располагается единая база данных. Производительность таких систем может увеличиваться как путем наращивания числа процессоров и объемов оперативной памяти в каждом узле кластера, так и посредством увеличения количества самих узлов.

" Архитектура без разделения ресурсов (Shared Nothing Architecture). Как и в архитектуре с общими дисками, в этой архитектуре поддерживается единый образ базы данных при работе с несколькими компьютерами, работающими под управлением своих копий операционной системы. Однако в этой архитектуре каждый узел системы имеет собственную оперативную память и собственные диски, которые не разделяются между отдельными узлами системы. Практически в таких системах разделяется только общий коммуникационный канал между узлами системы. Производительность таких систем может увеличиваться путем добавления процессоров, объемов оперативной и внешней (дисковой) памяти в каждом узле, а также путем наращивания количества таких узлов.

Таким образом, среда для работы параллельной базы данных обладает двумя важными свойствами: высокой готовностью и высокой производительностью. В случае кластерной организации несколько компьютеров или узлов кластера работают с единой базой данных. В случае отказа одного из таких узлов, оставшиеся узлы могут взять на себя задания, выполнявшиеся на отказавшем узле, не останавливая общий процесс работы с базой данных. Поскольку логически в каждом узле системы имеется образ базы данных, доступ к базе данных будет обеспечиваться до тех пор, пока в системе имеется по крайней мере один исправный узел. Производительность системы легко масштабируется, т.е. добавление дополнительных процессоров, объемов оперативной и дисковой памяти, и новых узлов в систему может выполняться в любое время, когда это действительно требуется.

Параллельные базы данных находят широкое применение в системах обработки транзакций в режиме on-line, системах поддержки принятия решений и часто используются при работе с критически важными для работы предприятий и организаций приложениями, которые эксплуатируются по 24 часа в сутки.

1.3. Виды программного обеспечения

СТРУКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

Из чего состоит любая вычислительная система? В первую очередь, это то, что в англоязычных странах принято называть словом hardware, или техническое обеспечение: процессор, память, монитор, дисковые устройства и т.д., обычно объединенные магистральным соединением, которое называется шиной

Во вторую очередь это программное обеспечение. Все программное обеспечение принято делить на две части: прикладное и системное. К прикладному программному обеспечению, как правило, относятся разнообразные банковские и прочие business программы, игры, текстовые процессоры, и т.п. Под системным программным обеспечением обычно понимают программы, способствующие функционированию и разработке

прикладных программ. Надо сказать, что деление на прикладное и системное программное обеспечение является отчасти условным и зависит от того, кто осуществляет такое деление. Так, обычный пользователь, неискушённый в программировании, может считать Microsoft Word системной программой, а с точки зрения программиста это приложение. Компилятор языка Си для обычного программиста это системная программа, а для системного прикладная. Несмотря на эту нечеткую грань, эту ситуацию можно отобразить в виде последовательности слоев:



КЛАССИФИКАЦИЯ ПРОГРАММ

1. BIOS - встроенная система ввода-вывода, хранится в постоянной памяти PC, не переписывается и не сбрасывается при выключении компьютера. Flash BIOS.
2. Операционные системы.
3. Оболочки систем.
4. Прикладные программы.
5. Языки программирования.

ВСТРОЕННЫЕ, НЕРЕЗИДЕНТНЫЕ И РЕЗИДЕНТНЫЕ ПРОГРАММЫ

Программы разделяются по способу работы:

Встроенные Нерезидентные Резидентные

Встроенные программы постоянно хранятся внутри компьютера, всегда готовы к работе, их не надо загружать. Они обеспечивают связь между аппаратной частью компьютера (железом) и другими работающими программами. (BIOS)

Нерезидентные (обычные)- самая распространенная категория программ продуктов. Решают самые различные задачи, но после того, как решение закончено, они полностью выгружаются из памяти, передают управление другим программам или операционным системам.

Резидентные программы после окончания не выгружаются полностью, а оставляют в памяти либо себя целиком, либо свои небольшие кусочки и передают управление другой программе. При этом они могут просто дать окончание работы этой программы, затем взять управление на себя, либо выполняет в это время какие-либо действия: (выводит текущее время в правом верхнем углу экрана).

Лекция 2

Микропроцессоры семейства x86. Архитектура микропроцессора Pentium 4.

Микропроцессоры семейства x86 Микропроцессоры семейства Pentium Характеристика CISC и RISC архитектур Развитие архитектуры IA-32 в семействе Pentium MMX, SSE, SSE2 Список регистров микропроцессора Pentium Регистры общего назначения Сегментные регистры Регистры состояния и управления Регистр флагов Регистры управления Регистры системных адресов Регистры отладки Управляющие регистры Системные адресные регистры Регистры отладки

1. Архитектура и команды микропроцессоров

МИКРОПРОЦЕССОРЫ СЕМЕЙСТВА X86

Микропроцессоры корпорации Intel и персональные компьютеры на их базе прошли не очень длинный во времени, но значительный по существу путь развития, на протяжении которого кардинально изменялись и возможности и даже сами принципы их архитектуры. В то же время, внос в микропроцессор принципиальные изменения, разработчики были вынуждены постоянно иметь в виду необходимость обеспечения совместимости новых моделей со старыми, чтобы не отпугивать потенциального покупателя перспективой полной замены освоенного или разработанного им программного обеспечения. В результате современные микропроцессоры типа Pentium, обеспечивая такие возможности, как 32-битную адресацию почти неограниченных объемов памяти, многозадачный режим с одновременным выполнением нескольких программ, аппаратных средства защиты операционной системы и прикладных программ друг друга, богатый набор дополнительных эффективных команд и способе адресации, в то же время могут работать (и часто работают) в режиме первых микропроцессоров типа 8086, используя всего лишь 1 мегабайт оперативной памяти, 16-разрядные операнды (т. е. числа в диапазоне до $2^{16} - 1 = 65535$) и ограниченный состав команд. Поскольку программирование на языке ассемблера напрямую затрагивает аппаратные возможности микропроцессора, прежде всего следует выяснить, в какой степени программист может использовать новые возможности микропроцессоров в своих программах и какие проблемы программной несовместимости могут при этом возникнуть.

Первые персональные компьютеры корпорации IBM, появившиеся в 1981 г. и получившие название IBM PC, использовали в качестве центрального вычислительного узла 16-разрядный микропроцессор с 8-разрядной внешней шиной Intel 8088. В дальнейшем в персональных компьютерах стал использоваться и другой вариант микропроцессора, 8086, который отличался от 8088 тем, что являлся полностью 16-разрядным. С тех пор его имя стало нарицательным, и в программах, использующих только возможности процессоров 8088 или 8086, говорят, что они работают в режиме 86-го процессора.

В 1983 г. корпорацией Intel был предложен микропроцессор 80286, в котором был реализован принципиально новый режим работы, получивший название защищенного. Однако процессор 80286 мог работать и в режиме 86-го процессора, который стали называть реальным.

В дальнейшем на смену процессору 80286 пришли модели 80386, i486 и, наконец, различные варианты процессора Pentium. Все они могут работать и в реальном, и в защищенном режимах. Хотя каждая следующая модель была значительно совершеннее предыдущей (в частности, почти на два порядка возросла скорость работы процессора, начиная с модели 80386 процессор стал 32-разрядным, а в процессорах Pentium реализован даже 64-разрядный обмен данными с системной шиной), однако с точки зрения программиста все эти процессоры весьма схожи. Основным их качеством является наличие двух режимов работы - реального и защищенного. Строго говоря, в современных процессорах реализован еще и третий режим - виртуального 86-го процессора, или V86, однако в плане использования языка ассемблера этот режим не отличается от обычного режима 86-го процессора, мы его касаться не будем.

Реальный и защищенный режимы прежде всего принципиально различаются способом обращения к оперативной памяти компьютера. Метод адресации памяти, используемый в реальном режиме, позволяет адресовать память лишь в пределах 1 Мбайт; в защищенном режиме используется другой механизм (из-за чего, в частности, эти режимы и оказались полностью несовместимыми), позволяющий обращаться к памяти объемом до 4 Гбайт. Другое важное отличие защищенного режима заключается в аппаратной поддержке многозадачности с аппаратной же (т.е. реализованной в самом микропроцессоре) защитой задач друг от друга. Реальный и защищенный режимы имеют прямое отношение к работе операционной системы, установленной на компьютере.

МИКРОПРОЦЕССОРЫ СЕМЕЙСТВА PENTIUM

Архитектура чипов PPlain и PMMX имеет много особенностей, которые оптимизированы специально под некоторые наиболее часто используемые инструкции или комбинации инструкций. В этих случаях нельзя использовать общие методики оптимизации. Следовательно, правила для оптимизирования программного обеспечения весьма сложны и имеют много исключений, но возможный выигрыш в производительности значителен. У процессоров PPro, PI и PIII совершенно разный дизайн в том, как процессор берет на себя часть оптимизации, запуская инструкции в совершенно разном порядке, но усложненный дизайн этих процессоров становится причиной множества потенциальных узких мест, поэтому можно много выиграть, оптимизируя код вручную для этих процессоров. У Pentium 4 также другой дизайн, и пути оптимизации для Pentium 4 отличаются от тех, которые используются для других версий. Это руководство не рассказывает о Pentium 4 - читатель может обратиться к руководствам от Intel.

Затем вам нужно выделить критические части вашей программы. Зачастую более, чем 99% времени процессора тратится на внутренний цикл программы. В этом случае вам нужно только оптимизировать этот цикл и оставить все остальное на языке высокого уровня. Некоторые асм-программисты тратят огромное количество энергии на оптимизирование не тех частей своей программы, единственным значительным результатом чего становится то, что программу становится труднее отлаживать и распространять!

Если не совсем понятно, какие части вашей программы являются критическим, вы можете использовать профайлер, чтобы найти их. Если выясняется, что узким местом является доступ к диску, вы можете модифицировать вашу программу таким образом, чтобы сделать доступ к диску последовательным (для улучшения работы кэша диска), а не переключаться на ассемблерное программирование. Если узким местом является вывод графики, вы можете поискать путь к уменьшению вызовов графических процедур.

ХАРАКТЕРИСТИКА CISC И RISC АРХИТЕКТУР

Двумя основными архитектурами набора команд, используемыми компьютерной промышленностью на современном этапе развития вычислительной техники являются архитектуры CISC и RISC. Основоположником CISC-архитектуры можно считать компанию IBM с ее базовой архитектурой /360, ядро которой используется с 1964 года и дошло до наших дней, например, в таких современных мейнфреймах как IBM ES/9000. Лидером в разработке микропроцессоров с полным набором команд (CISC - Complete Instruction Set Computer) считается компания Intel со своей серией x86 и Pentium. Эта архитектура является практическим стандартом для рынка микрокомпьютеров. Для CISC-процессоров характерно: сравнительно небольшое число регистров общего назначения; большое количество машинных команд, некоторые из которых нагружены семантически аналогично операторам высокоуровневых языков программирования и выполняются за много тактов; большое количество методов адресации; большое количество форматов команд различной разрядности; преобладание двухадресного формата команд; наличие команд обработки типа регистр-память.

Основой архитектуры современных рабочих станций и серверов является архитектура компьютера с сокращенным набором команд (RISC - Reduced Instruction Set Computer). Зачатки этой архитектуры уходят своими корнями к компьютерам CDC6600, разработчики которых (Торнтон, Крэй и др.) осознали важность упрощения набора команд для построения быстрых вычислительных машин. Эту традицию упрощения архитектуры С. Крэй с успехом применил при создании широко известной серии суперкомпьютеров компании Cray Research. Однако окончательно понятие RISC в современном его понимании сформировалось на базе трех исследовательских проектов компьютеров: процессора 801 компании IBM, процессора RISC университета Беркли и процессора MIPS Стенфордского университета.

Среди других особенностей RISC-архитектур следует отметить наличие достаточно большого регистрового файла (в типовых RISC-процессорах реализуются 32 или большее число регистров по сравнению с 8 - 16 регистрами в CISC-архитектурах), что позволяет большему объему данных храниться в регистрах на процессорном кристалле больше время и упрощает работу компилятора по распределению регистров под переменные.

Для обработки, как правило, используются трехадресные команды, что помимо упрощения дешифрации дает возможность сохранять большее число переменных в регистрах без их последующей перезагрузки.

Развитие архитектуры RISC в значительной степени определялось прогрессом в области создания оптимизирующих компиляторов. Именно современная техника компиляции позволяет эффективно использовать преимущества большого регистрового файла, конвейерной организации и большей скорости выполнения команд. Современные компиляторы используют также преимущества другой оптимизационной техники для повышения производительности, обычно применяемой в процессорах RISC: реализацию задержанных переходов и суперскалярной обработки, позволяющей в один и тот же момент времени выдавать на выполнение несколько команд.

Следует отметить, что в последних разработках компании Intel (имеются в виду Pentium и Pentium Pro), а также ее последователей-конкурентов (AMD R5, Cyrix M1, NexGen Nx586 и др.) широко используются идеи, реализованные в RISC-микропроцессорах, так что многие различия между CISC и RISC стираются. Однако сложность архитектуры и системы команд x86 остается и является главным фактором, ограничивающим производительность процессоров на ее основе.

РАЗВИТИЕ АРХИТЕКТУРЫ IA-32 В СЕМЕЙСТВЕ PENTIUM

Общая архитектура процессора определяет комплекс средств, предоставляемых пользователю для решения различных задач. Эта архитектура задаёт базовую систему команд процессора и реализуемых способов адресации, набор программно-доступных регистров (регистровая модель), возможные режимы работы процессора и обращения к памяти и внешним устройствам (организация памяти и реализация обмена по системной шине), средства обработки прерываний и исключений.

В процессоре Pentium 4 реализуется архитектура IA-32 (Intel Architecture-32), общая для всех 32-разрядных микропроцессоров Intel, начиная с i386. В табл. 1 приведены основные модели процессоров, в которых используется эта архитектура, и некоторые их характеристики. Отметим, что модели Pentium II Xeon и Pentium III Xeon ориентированы на работу в высокопроизводительных мультимикропроцессорных системах (серверах, рабочих станциях). Для этих же приложений планируется выпуск в 2001 году модификации процессора Pentium 4 с поддержкой мультимикропроцессорного режима работы (название проекта — Foster).

Таблица 1. Некоторые характеристики процессоров архитектуры IA-32

Модель, начало выпуска	Число транзисторов	Тактовая частота, МГц	Объем внутренней кэш-памяти
i386, октябрь 1985 г.	275 тыс.	до 40	Нет
i486, апрель 1989 г.	1,2 млн.	до 100	8 Кбайт - команды 8 Кбайт - данные
Pentium, март 1993 г.	3,1 млн.	до 200	8 Кбайт - команды 8 Кбайт - данные
Pentium Pro, ноябрь 1995 г.	5,5 млн.	до 200	8 Кбайт - команды 8 Кбайт - данные
Pentium MMX, январь 1997 г.	4,5 млн.	до 233	8 Кбайт - команды 8 Кбайт - данные
Pentium II, май 1997 г. (Xeon, июнь 1998 г.)	7,5 млн.	до 450	16 Кбайт - команды 16 Кбайт - данные
Celeron, апрель 1998 г.		до 750	128 Кбайт - общий
Pentium III, февраль 1999 г. (Xeon, март 1999 г.)	8,5 млн.	до 1000 (до 700)	16 Кбайт - команды 16 Кбайт - данные
Pentium 4, ноябрь 2000 г. (Foster, 2001 г.)	42 млн.	до 1500 (2000)	256 Кбайт - общий 12 К - микрокоманды 8 Кбайт - данные

История архитектуры IA-32 насчитывает уже более 15 лет, и её основные черты достаточно полно описаны в ряде монографий. Поэтому ограничимся их кратким обзором.

В процессе развития IA-32 производилось расширение возможностей обработки данных, представленных в различных форматах (рис. 7.1). Процессоры i386 выполняли обработку только целочисленных операндов. Для обработки чисел с “плавающей точкой” использовался внешний сопроцессор i387, подключаемый к микропроцессору. В состав процессоров i486 и последующих моделей Pentium введён специальный блок FPU (Floating-Point Unit), выполняющий операции над числами с “плавающей точкой”. В процессорах Pentium MMX была впервые реализована групповая обработка нескольких целочисленных операндов разрядностью 1, 2, 4 или 8 байт с помощью одной команды. Такая обработка обеспечивается введением дополнительного блока MMX (Multi-Media Extension — Мультимедийное Расширение). Название

блока отражает его направленность на обработку видео- и аудиоданных, когда одновременное выполнение одной операции над несколькими операндами позволяет существенно повысить скорость обработки изображений и звуковых сигналов.

MMX, SSE, SSE2

MultiMedia eXtension - дополнительные возможности, ориентированные на обработку цифрового изображения и звука, анонсированные Intel в процессорах P55C. Включают в себя 57 новых команд, предназначенных для обработки звуковых и видеосигналов; команды могут использоваться в режиме SIMD (Single Instruction, Many Data - одна команда, много данных), когда одной командой одновременно обрабатывается несколько элементов данных. Процессоры с MMX имеют также удвоенный (32 кб) объем внутреннего (L1) кэша.

Расширения MMX реализованы в виде дополнительного режима, в который процессор может переключаться из обычного режима работы. В режиме MMX набор регистров сопроцессора (FPU) используется для хранения данных MMX-команд - это гарантирует совместимость с существующими операционными системами, которые не поддерживают MMX напрямую. Однако такое совмещение может снизить эффективность работы в случае попеременного использования обычных вычислений с плавающей точкой и работы в режиме MMX.

Использование MMX позволит перенести основную нагрузку по обработке изображения и звука на центральный процессор, оставив видео и звуковым адаптерам только преобразование аналог-цифра. Иначе говоря, с ростом мощности центральных процессоров становится выгоднее выполнять на них ту работу, которая несколько лет назад была отдана периферийным видео- и звуковым процессорам по причине недостаточной мощности центрального; сейчас опять происходит возврат к централизованной обработке.

Начиная с модели Pentium III, в процессоры вводится блок SSE (Streaming SIMD Extension — Потокное SIMD-расширение) для групповой обработки чисел с “плавающей точкой”.

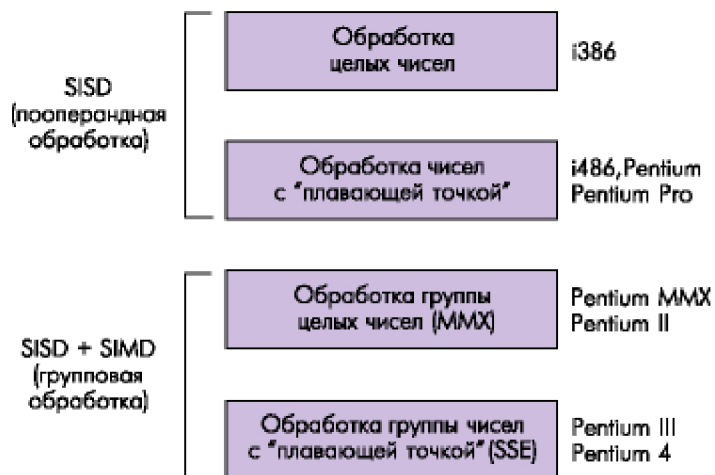


Рис. 7.1. Эволюция архитектуры IA-32

Таким образом, если первые модели процессоров Pentium выполняли только пооперандную обработку данных по принципу “Одна команда – Одни данные” (SISD — Single Instruction – Single Data), то, начиная с процессора Pentium MMX, реализуется также их групповая обработка по принципу “Одна команда – Много данных” (SIMD — Single Instruction – Multiple Data).

Соответственно, расширяется и набор регистров процессора, используемых для промежуточного хранения данных (рис. 7.2). Кроме 32-разрядных регистров для хранения целочисленных операндов, процессоры Pentium содержат 80-разрядные регистры, которые обслуживают блоки FPU и MMX. При работе FPU регистры ST0-ST7 образуют кольцевой стек, в котором хранятся числа с “плавающей точкой”, представленные в формате с расширенной точностью (80 разрядов). При реализации MMX-операций они используются как 64-разрядные регистры MM0-MM7, где могут храниться несколько операндов (8 8-разрядных, 4 16-разрядных, 2 32-разрядных или один 64-разрядный), над которыми одновременно выполняется поступившая в процессор команда (арифметическая, логическая, сдвиг и ряд других).

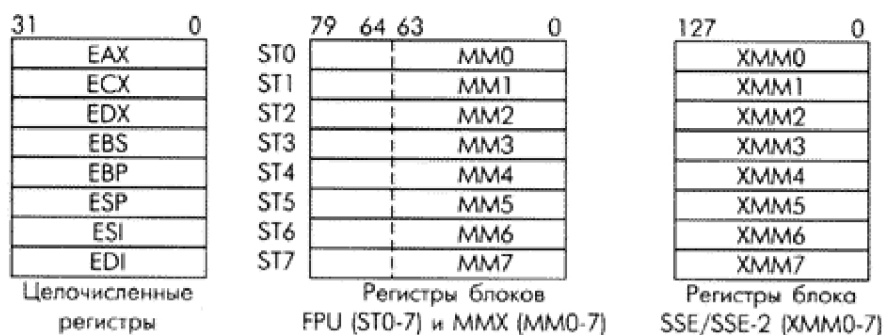


Рис. 7.2. Регистры хранения данных в процессорах Pentium

Блок SSE-2, введённый в состав процессора Pentium 4, значительно расширяет возможности обработки нескольких операндов по принципу SIMD, по сравнению с блоком SSE в модели Pentium III. Этот блок реализует 144 новые команды, обеспечивающих одновременное выполнение операций над несколькими операндами, которые располагаются в памяти и в 128-разрядных регистрах XMM0-XMM7. В регистрах могут храниться и одновременно обрабатываться 2 числа с “плавающей точкой” в формате двойной точности (64 разряда) или 4 числа в формате одинарной точности (32 разряда). Этот блок может также одновременно обрабатывать целочисленные операнды: 16 8-разрядных, 8 16-разрядных, 4 32-разрядных или 2 64-разрядных. В результате производительность процессора Pentium 4 при выполнении таких операций оказывается вдвое выше, чем Pentium III.

Операции SSE-2 позволяют существенно повысить эффективность процессора при реализации трёхмерной графики и Интернет-приложений, обеспечении сжатия и кодирования аудио- и видеоданных и в ряде других применений.

Введение большой группы команд SSE-2 является основной особенностью реализованного в Pentium 4 варианта архитектуры IA-32. Что касается базового набора команд и используемых способов адресации операндов, то они практически полностью совпадают с набором команд и способов адресации в предыдущих моделях Pentium. Процессор обеспечивает реальный и защищённый режимы работы, реализует сегментную и страничную организации памяти. Таким образом пользователь имеет дело с хорошо знакомым набором регистров и способов адресации, может работать с базовой системой команд и известными вариантами реализации прерываний и исключений, которые характерны для всех моделей семейства Pentium.

СПИСОК РЕГИСТРОВ МИКРОПРОЦЕССОРА PENTIUM

EAX = (16+AX=(AH+AL))

EBX = (16+BX=(BH+BL))

ECX = (16+CX=(CH+CL))

EDX = (16+DX=(DH+DL))

ESI = (16+SI)

EDI = (16+DI)

EBP = (16+BP)

ESP = (16+SP)

Регистры EAX,EBX,EDX,ECX называют рабочими регистрами. Регистры EDI, ESI – индексные регистры, играют особую роль в строковых операциях. Регистр EBP обычно используется для адресации в стеке параметров и локальных переменных. Регистр ESP - указатель стека, автоматически модифицируется командами PUSH, POP, RET, CALL. Явно используется реже.

РЕГИСТРЫ ОБЩЕГО НАЗНАЧЕНИЯ

Все регистры этой группы позволяют обращаться к своим “младшим” частям. Рассматривая этот рисунок, заметьте, что использовать для самостоятельной адресации можно только младшие 16 и 8-битные части этих регистров. Старшие 16 бит этих регистров как самостоятельные объекты недоступны. Это сделано, как мы отметили выше, для совместимости с младшими 16-разрядными моделями микропроцессоров фирмы Intel.

Перечислим регистры, относящиеся к группе регистров общего назначения. Так как эти регистры физически находятся в микропроцессоре внутри арифметико-логического устройства (АЛУ), то их еще называют *регистрами АЛУ*:

- **eax/ax/ah/al** (Accumulator register) — *аккумулятор*.
Применяется для хранения промежуточных данных. В некоторых командах использование этого регистра обязательно;
- **ebx/bx/bh/bl** (Base register) — *базовый* регистр.
Применяется для хранения базового адреса некоторого объекта в памяти;
- **ecx/cx/ch/cl** (Count register) — *регистр-счетчик*.
Применяется в командах, производящих некоторые повторяющиеся действия. Его использование зачастую неявно и скрыто в алгоритме работы соответствующей команды.
К примеру, команда организации цикла loop кроме передачи управления команде, находящейся по некоторому адресу, анализирует и уменьшает на единицу значение регистра *ecx/cx*;
- **edx/dx/dh/dl** (Data register) — *регистр данных*.
Так же, как и регистр *eax/ax/ah/al*, он хранит промежуточные данные. В некоторых командах его использование обязательно; для некоторых команд это происходит неявно.

Следующие два регистра используются для поддержки так называемых цепочечных операций, то есть операций, производящих последовательную обработку цепочек элементов, каждый из которых может иметь длину 32, 16 или 8 бит:

- **esi/si** (Source Index register) — *индекс источника*.
Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;
- **edi/di** (Destination Index register) — *индекс приемника* (получателя).
Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

В архитектуре микропроцессора на программно-аппаратном уровне поддерживается такая структура данных, как стек. Для работы со стеком в системе команд микропроцессора есть специальные команды, а в программной модели микропроцессора для этого существуют специальные регистры:

- **esp/sp** (Stack Pointer register) — *регистр указателя стека*.
Содержит указатель вершины стека в текущем сегменте стека.
- **ebp/bp** (Base Pointer register) — *регистр указателя базы кадра стека*.
Предназначен для организации произвольного доступа к данным внутри стека.

Не спешите пугаться столь жесткого функционального назначения регистров АЛУ. На самом деле, большинство из них могут использоваться при программировании для хранения операндов практически в любых сочетаниях. Но, как мы отметили выше, некоторые команды используют фиксированные регистры для выполнения своих действий. Это нужно обязательно учитывать.

Использование жесткого закрепления регистров для некоторых команд позволяет более компактно кодировать их машинное представление. Знание этих особенностей позволит вам при необходимости хотя бы на несколько байт сэкономить память, занимаемую кодом программы.

СЕГМЕНТНЫЕ РЕГИСТРЫ

В программной модели микропроцессора имеется шесть сегментных регистров: cs, ss, ds, es, fs. Их существование обусловлено спецификой организации и использования оперативной памяти микропроцессорами Intel. Она заключается в том, что микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых сегментами. Соответственно, такая организация памяти называется сегментной.

Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены сегментные регистры. Фактически, с небольшой поправкой, как мы увидим далее, в этих регистрах содержатся адреса памяти с которых начинаются соответствующие сегменты. Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются адреса во вполне определенных сегментных регистрах. Микропроцессор поддерживает следующие типы сегментов:

2. *Сегмент кода*. Содержит команды программы.
Для доступа к этому сегменту служит регистр cs (code segment register) — *сегментный регистр кода*. Он содержит адрес сегмента с машинными командами, к которому имеет доступ микропроцессор (то есть эти команды загружаются в конвейер микропроцессора).

3. **Сегмент данных.** Содержит обрабатываемые программой данные.
Для доступа к этому сегменту служит регистр ds (data segment register) — *сегментный регистр данных*, который хранит адрес сегмента данных текущей программы.
4. **Сегмент стека.** Этот сегмент представляет собой область памяти, называемую *стеком*.
Работу со стеком микропроцессор организует по следующему принципу: *последний записанный в эту область элемент выбирается первым*. Для доступа к этому сегменту служит регистр ss (stack segment register) — *сегментный регистр стека*, содержащий адрес сегмента стека.
5. **Дополнительный сегмент данных.**
Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds, при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

РЕГИСТРЫ СОСТОЯНИЯ И УПРАВЛЕНИЯ

В микропроцессор включены несколько регистров которые постоянно содержат информацию о состоянии как самого микропроцессора, так и программы, команды которой в данный момент загружены на конвейер. К этим регистрам относятся:

- регистр флагов **eflags/flags**;
- регистр указателя команды [cip/ip](#).

Используя эти регистры, можно получать информацию о результатах выполнения команд и влиять на состояние самого микропроцессора. Рассмотрим подробнее назначение и содержимое этих регистров:

eflags/flags (flag register) — регистр *флагов*. Разрядность **eflags/flags** — 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются *флагами*. Младшая часть этого регистра полностью аналогична регистру **flags** для i8086. На рис. 7.3 показано содержимое регистра **eflags**.

ФЛАГИ СОСТОЯНИЯ:

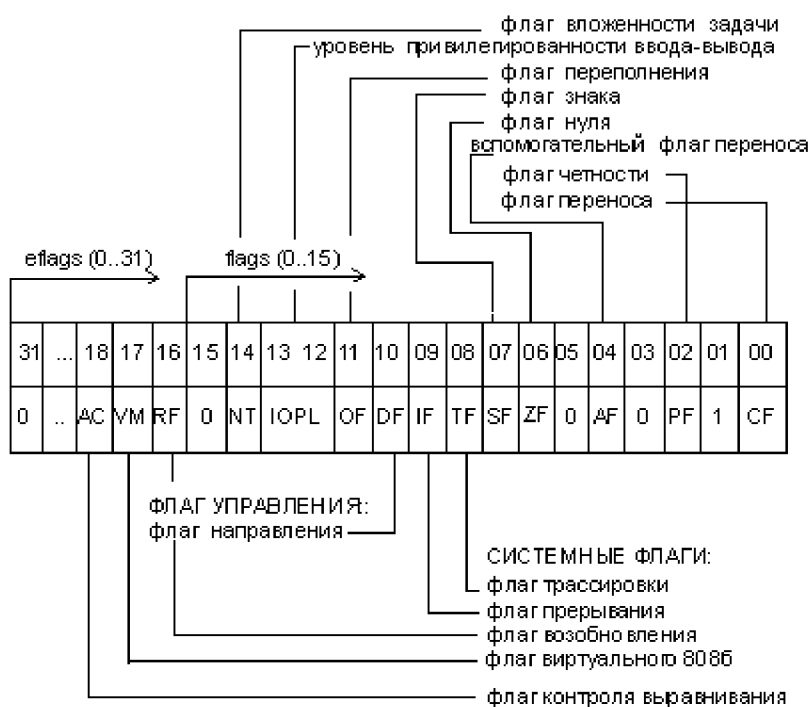


Рис 7.3 Содержимое регистра **eflags**

Исходя из особенностей использования, флаги регистра *eflags/flags* можно разделить на три группы:

- *8 флагов состояния.* Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра *eflags* отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм.
- *1 флаг управления.* Обозначается **df** (Directory Flag). Он находится в 10-м бите регистра *eflags* и используется цепочечными командами. Значение флага *df* определяет направление поэлементной обработки в этих операциях: от начала строки к концу (*df* = 0) либо наоборот, от конца строки к ее началу (*df* = 1). Для работы с флагом *df* существуют специальные команды: *cld* (снять флаг *df*) и *std* (установить флаг *df*). Применение этих команд позволяет привести флаг *df* в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;
- *5 системных флагов,* управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

РЕГИСТР ФЛАГОВ

Содержит 32 бита. Вот используемые значения битов.

0 й бит,	флаг переноса (CF), устанавливается в 1 если был перенос из старшего бита.
1 й бит	1.
2 й бит	флаг четности. Устанавливается в 1, если младший байт результата содержит четное число единиц.
3 й бит	0.
4 й бит	флаг вспомогательного переноса (AF). Устанавливается в 1, если произошел перенос из третьего бита в четвертый.
5 й бит	0.
6 й бит	флаг нуля (ZF). Устанавливается в единицу, если результат операции ноль.
7 й бит	флаг знака (SF). Равен старшему биту результата.
8 й бит	флаг ловушки (TF). Установка в единицу этого флага приводит к тому, что после каждой команды вызывается INT 3. Используется отладчиками в реальном режиме.
9 й бит	флаг прерываний (IF). Сброс этого флага в 0 приводит к тому, что микропроцессор перестает воспринимать прерывания.
10 й бит	флаг направления (DF). Данный флаг учитывается в строковых операциях. Если флаг равен 1, то в строковых операциях адрес автоматически уменьшается.
11 й бит	флаг переполнения (OF). Устанавливается в единицу, если результат операции над числом со знаком вышел за допустимые пределы.
12,13 й биты	уровень привилегий ввода вывода (IOPL).
14 бит	флаг вложенной задачи (NT).
15 й бит	0.
16 й бит	флаг возобновления (RF). Используется совместно с регистрами точек отладочного останова.
17 й бит	в защищенном режиме включает режим виртуального режима 8086 (VM).
18 й бит	флаг контроля выравнивания (AC). При равенстве этого флага 1 и при обращении к невыровненному операнду вызывает исключение 17.
19 й бит	виртуальная версия флага IF (VIF). Работает в защищенном режиме.
20 й бит	виртуальный запрос прерывания (VIP).
21 й бит	флаг доступности команды идентификации.
22-31 –й,	0.

Таблица 1. Флаги состояния

Мнемоника флага	Флаг	Номер бита в <i>eflags</i>	Содержание и назначение
cf	Флаг переноса (Carry Flag)	0	1 — арифметическая операция произвела перенос из старшего бита результата. Старшим является 7, 15 или 31-й бит в зависимости от размерности операнда; 0 — переноса не было
pf	Флаг паритета (Parity Flag)	2	1 — 8 младших разрядов (этот флаг — только для 8 младших разрядов операнда любого размера) результата содержат четное число единиц; 0 — 8 младших разрядов результата содержат нечетное число единиц
af	Вспомогательный флаг переноса (Auxiliary carry Flag)	4	Только для команд работающих с BCD-числами. Фиксирует факт заема из младшей тетрады результата: 1 — в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде; 0 — переносов и заемов в(из) 3 разряд(а) младшей тетрады результата не было
zf	Флаг нуля (Zero Flag)	6	1 — результат нулевой; 0 — результат ненулевой
sf	Флаг знака (Sign Flag)	7	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно): 1 — старший бит результата равен 1; 0 — старший бит результата равен 0
of	Флаг переполнения (Overflow Flag)	11	Флаг of используется для фиксирования факта потери значащего бита при арифметических операциях: 1 — в результате операции происходит перенос (заем) в(из) старшего, знакового бита результата (биты 7, 15 или 31 для 8, 16 или 32-разрядных операндов соответственно); 0 — в результате операции не происходит переноса (заема) в(из) старшего, знакового бита результата
iopl	Уровень Привилегий ввода-вывода (Input/Output Privilege Level)	12, 13	Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода в зависимости от привилегированности задачи
nt	флажок вложенности задачи (Nested Task)	14	Используется в защищенном режиме работы микропроцессора для фиксации того факта, что одна задача вложена в другую

Таблица 2. Системные флаги

Мнемоника флага	Флаг	Номер бита в <i>eflags</i>	Содержание и назначение
tf	Флаг трассировки (Trace Flag)	8	Предназначен для организации пошаговой работы микропроцессора. 1 — микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; 0 — обычная работа
if	Флаг прерывания (Interrupt enable Flag)	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR). 1 — аппаратные прерывания разрешены; 0 — аппаратные прерывания запрещены
rf	Флаг возобновления (Resume Flag)	16	Используется при обработке прерываний от регистров отладки.

vm	Флаг виртуального (Virtual 8086 Mode)	17	Признак работы микропроцессора в режиме виртуального 8086. 1 — процессор работает в режиме виртуального 8086; 0 — процессор работает в реальном или защищенном режиме
ac	Флаг контроля выравнивания (Alignment Check)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти. Используется совместно с битом am в системном регистре cr0. К примеру, Pentium разрешает размещать команды и данные с любого адреса. Если требуется контролировать выравнивание данных и команд по адресам кратным 2 или 4, то установка данных битов приведет к тому, что все обращения по некртным адресам будут возбуждать исключительную ситуацию

eip/ip (Instruction Pointer register) — регистр-указатель команд.

Регистр *eip/ip* имеет разрядность 32/16 бит и содержит смещение следующей подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра *eip/ip*.

РЕГИСТРЫ УПРАВЛЕНИЯ

В группу регистров управления входят 4 регистра:

cr0, cr1, cr2, cr3.

Эти регистры предназначены для общего управления системой.

Регистры управления доступны только программам с уровнем привилегий 0.

Хотя микропроцессор имеет четыре регистра управления, доступными являются только три из них — исключается *cr1*, функции которого пока не определены (он зарезервирован для будущего использования).

Регистр **cr0** содержит *системные флаги*, управляющие режимами работы микропроцессора и отражающие его состояние глобально, независимо от конкретных выполняющихся задач.

Назначение системных флагов:

- **pe** (Protect Enable), бит 0 — *разрешение защищенного режима работы*. Состояние этого флага показывает, в каком из двух режимов — *реальном* (pe=0) или *защищенном* (pe=1) — работает микропроцессор в данный момент времени.
- **mp** (Math Present), бит 1 — *наличие сопроцессора*. Всегда 1.
- **ts** (Task Switched), бит 3 — *переключение задач*. Процессор автоматически устанавливает этот бит при переключении на выполнение другой задачи.
- **am** (Aligment Mask), бит 18 — *маска выравнивания*. Этот бит разрешает (am = 1) или запрещает (am = 0) контроль выравнивания.
- **cd** (Cache Disable), бит 30, — *запрещение кэш-памяти*. С помощью этого бита можно запретить (cd = 1) или разрешить (cd = 0) использование внутренней кэш-памяти (кэш-памяти первого уровня).
- **pg** (PaGing), бит 31, — *разрешение* (pg = 1) или *запрещение* (pg = 0) *страничного преобразования*. Флаг используется при страничной модели организации памяти.

Регистр **cr2** используется при страничной организации оперативной памяти для регистрации ситуации, когда текущая команда обратилась по адресу, содержащемуся в странице памяти, отсутствующей в данный момент времени в памяти.

В такой ситуации в микропроцессоре возникает исключительная ситуация с номером 14, и линейный 32-битный адрес команды, вызвавшей это исключение, записывается в регистр *cr2*. Имея эту информацию, обработчик исключения 14 определяет нужную страницу, осуществляет ее подкачку в память и возобновляет нормальную работу программы;

Регистр **cr3** также используется при страничной организации памяти.

Это так называемый *регистр каталога страниц первого уровня*. Он содержит 20-битный физический базовый адрес каталога страниц текущей задачи. Этот каталог содержит 1024 32-битных дескриптора, каждый из которых содержит адрес таблицы страниц второго уровня. В свою очередь каждая из таблиц страниц второго уровня содержит 1024 32-битных дескриптора, адресующих страничные кадры в памяти. Размер страничного кадра — 4 Кбайт.

РЕГИСТРЫ СИСТЕМНЫХ АДРЕСОВ

Эти регистры еще называют *регистрами управления памятью*.

Они предназначены для защиты программ и данных в мультизадачном режиме работы микропроцессора.

При работе в защищенном режиме микропроцессора адресное пространство делится на:

- *глобальное* — общее для всех задач;
- *локальное* — отдельное для каждой задачи.

Этим разделением и объясняется присутствие в архитектуре микропроцессора следующих системных регистров:

- *регистра таблицы глобальных дескрипторов **gdt*** (Global Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16—47) базовый адрес глобальной дескрипторной таблицы GDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы GDT;
- *регистра таблицы локальных дескрипторов **ldtr*** (Local Descriptor Table Register) имеющего размер 16 бит и содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;
- *регистра таблицы дескрипторов прерываний **idt*** (Interrupt Descriptor Table Register) имеющего размер 48 бит и содержащего 32-битовый (биты 16—47) базовый адрес дескрипторной таблицы прерываний IDT и 16-битовое (биты 0—15) значение предела, представляющее собой размер в байтах таблицы IDT;
- *16-битового регистра задачи **tr*** (Task Register), который подобно регистру **ldtr**, содержит селектор, то есть указатель на дескриптор в таблице GDT. Этот дескриптор описывает *текущий сегмент состояния задачи* (TSS — Task Segment Status). Этот сегмент создается для каждой задачи в системе, имеет жестко регламентированную структуру и содержит контекст (текущее состояние) задачи. Основное назначение сегментов TSS — сохранять текущее состояние задачи в момент переключения на другую задачу.

РЕГИСТРЫ ОТЛАДКИ

Это очень интересная группа регистров, предназначенных для аппаратной отладки. Средства аппаратной отладки впервые появились в микропроцессоре i486. Аппаратно микропроцессор содержит восемь регистров отладки, но реально из них используются только 6.

Регистры **dr0**, **dr1**, **dr2**, **dr3** имеют разрядность 32 бит и предназначены для задания линейных адресов четырех точек прерывания. Используемый при этом механизм следующий: любой формируемый текущей программой адрес сравнивается с адресами в регистрах **dr0...dr3**, и при совпадении генерируется исключение отладки с номером 1.

Регистр **dr6** называется регистром состояния отладки. Биты этого регистра устанавливаются в соответствии с причинами, которые вызвали возникновение последнего исключения с номером 1.

Перечислим эти биты и их назначение:

- **b0** — если этот бит установлен в 1, то последнее исключение (прерывание) возникло в результате достижения контрольной точки, определенной в регистре **dr0**;
- **b1** — аналогично **b0**, но для контрольной точки в регистре **dr1**;
- **b2** — аналогично **b0**, но для контрольной точки в регистре **dr2**;
- **b3** — аналогично **b0**, но для контрольной точки в регистре **dr3**;
- **bd** (бит 13) — служит для защиты регистров отладки;
- **bs** (бит 14) — устанавливается в 1, если исключение 1 было вызвано состоянием флага **tf** = 1 в регистре **eflags**;
- **bt** (бит 15) устанавливается в 1, если исключение 1 было вызвано переключением на задачу с установленным битом ловушки в TSS **t** = 1.

Все остальные биты в этом регистре заполняются нулями. Обработчик исключения 1 по содержимому **dr6** должен определить причину, по которой произошло исключение, и выполнить необходимые действия.

Регистр **dr7** называется регистром управления отладкой. В нем для каждого из четырех регистров контрольных точек отладки имеются поля, с помощью которых можно уточнить следующие условия, при которых следует сгенерировать прерывание:

- *место регистрации контрольной точки* — только в текущей задаче или в любой задаче. Эти биты занимают младшие восемь бит регистра *dr7* (по два бита на каждую контрольную точку (фактически точку прерывания), задаваемую регистрами *dr0*, *dr1*, *dr2*, *dr3* соответственно).
Первый бит из каждой пары — это так называемое *локальное* разрешение; его установка говорит о том, что точка прерывания действует если она находится в пределах адресного пространства текущей задачи.
Второй бит в каждой паре определяет *глобальное* разрешение, которое говорит о том, что данная контрольная точка действует в пределах адресных пространств всех задач, находящихся в системе;
- *тип доступа*, по которому инициируется прерывание: только при выборке команды, при записи или при записи/чтении данных. Биты, определяющие подобную природу возникновения прерывания, локализованы в старшей части данного регистра.

Большинство из системных регистров программно доступны. Не все из них понадобятся в нашем дальнейшем изложении, но, тем не менее, я коротко рассмотрел их с тем, чтобы возбудить у читателя интерес к дальнейшему исследованию архитектуры микропроцессора.

УПРАВЛЯЮЩИЕ РЕГИСТРЫ

Регистр CR0.

- 0-й бит, разрешение защиты (PE). Переводит процессор в защищенный режим.
- 1-й бит, мониторинг сопроцессора (MP). Вызывает исключение 7 по каждой команде WAIT.
- 2-й бит, эмуляция сопроцессора (EM). Вызывает исключение 7 по каждой команде сопроцессора.
- 3-й бит, бит переключения задач (TS). Позволяет определить, относится данный контекст сопроцессора к текущей задаче или нет. Вызывает исключение 7 при выполнении следующей команды сопроцессора.
- 4-й бит, индикатор поддержки инструкций сопроцессора (ET).
- 5-й бит, разрешение стандартного механизма сообщений об ошибке сопроцессора (NE).
- 5-15-й бит, не используются.
- 16-й бит, разрешение защиты от записи на уровне привилегий супервизора (WP).
- 17-й бит, не используется.
- 18-й бит, разрешение контроля выравнивания (AM).
- 19-28-й бит, не используются.
- 29-й бит, запрет сквозной записи кэша и циклов аннулирования (NW).
- 30-й бит, запрет заполнения кэша (CD).
- 31-й бит, включение механизма страничной переадресации.

Регистр CR1 пока не используется.

Регистр CR2 хранит 32-битный линейный адрес, по которому был получен последний отказ страницы памяти.

Регистр CR3 – в старших 20 битах хранится физический базовый адрес таблицы каталога страниц.

Остальные биты.

- 3-й бит, кэширование страниц со сквозной записью (PWT).
- 4-й бит, запрет кэширование страницы (PCD).

Регистр CR4

- 0-й бит, разрешение использования виртуального флага прерываний в режиме V8086 (VME).

- 1-й бит, разрешение использования виртуального флага прерываний в защищенном режиме (PVI).
- 2-й бит, превращение инструкции RDTSC в привилегированную (TSD).
- 3-й бит, разрешение точек останова по обращению к портам ввода-вывода (DE).
- 4-й бит, включает режим адресации с 4-мегабайтными страницами (PSE).
- 5-й бит, включает 36-битное физическое адресное пространство (PAE).
- 6-й бит, разрешение исключения MC (MCE).
- 7-й бит, разрешение глобальной страницы (PGE).
- 8-й бит, разрешает выполнение команды RDPMC (PMC).
- 9-й бит, разрешает команды быстрого сохранения/восстановления состояния сопроцессора (FSR).

СИСТЕМНЫЕ АДРЕСНЫЕ РЕГИСТРЫ

GDTR – 6-байтный регистр, в котором содержится линейный адрес глобальной дескрипторной таблицы.

IDTR – 6-байтный регистр, содержащий 32-битный линейный адрес таблицы дескрипторов обработчиков прерываний.

LDTR – 10-байтный регистр, содержащий 16-битный селектор (индекс) для GDT и 8-байтный дескриптор.

TR – 10-байтный регистр, содержащий 16-битный селектор для GDT и весь 8-байтный дескриптор из GDT, описывающий TSS текущей задачи.

РЕГИСТРЫ ОТЛАДКИ

DR0...DR3 – хранят 32-битные линейные адреса точек останова.

DR6 (равносильно DR4) – отражает состояние контрольных точек.

DR7 (равносильно DR5) – управляет установкой контрольных точек.

Лекция 3

Команды микропроцессора.

Структура машинной команды Способы задания операндов команды Прямая адресация Косвенная адресация Команды пересылки данных общего назначения Команды ввода-вывода в порт Команды преобразования данных Команды работы со стеком

Структура машинной команды

Машинная команда представляет собой закодированное по определенным правилам указание микропроцессору на выполнение некоторой операции или действия. Каждая команда содержит элементы, определяющие:

- *что делать?* (ответ на этот вопрос дает элемент команды, называемый кодом операции (КОП));
- *объекты, над которыми нужно что-то делать* (эти элементы называются операндами);
- *как делать?* (эти элементы называются типами операндов — обычно задаются неявно).

Приведенный на рис. 1 формат машинной команды является самым общим. Максимальная длина машинной команды — 15 байт. Реальная команда может содержать гораздо меньшее количество полей, вплоть до одного — только КОП.

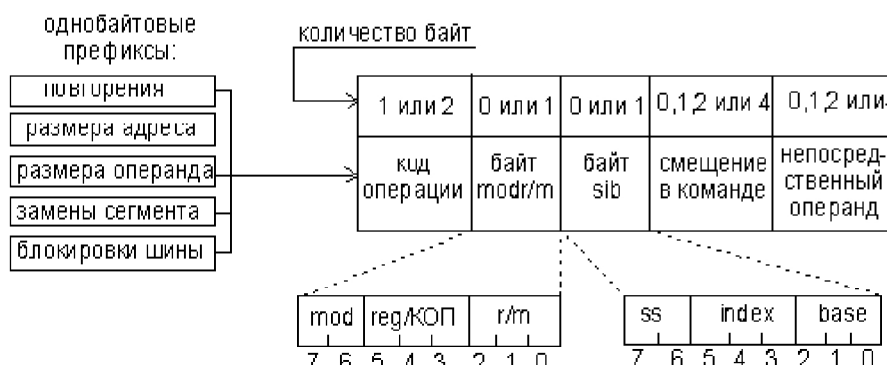


Рис. 1. Формат машинной команды

Опишем назначения полей машинной команды.

- **Префиксы.** Необязательные элементы машинной команды, каждый из которых состоит из одного байта или может отсутствовать. В памяти префиксы предшествуют команде. Назначение префиксов — модифицировать операцию, выполняемую командой. Прикладная программа может использовать следующие типы префиксов:
 - *Префикс замены сегмента.* В явной форме указывает, какой сегментный регистр используется в данной команде для адресации стека или данных. Префикс отменяет выбор сегментного регистра по умолчанию. Префиксы замены сегмента имеют следующие значения:
 - 2eh — замена сегмента cs;
 - 36h — замена сегмента ss;
 - 3eh — замена сегмента ds;
 - 26h — замена сегмента es;
 - 64h — замена сегмента fs;
 - 65h — замена сегмента gs.
 - *Префикс разрядности адреса* уточняет разрядность адреса (32 или 16-разрядный). Каждой команде, в которой используется адресный операнд, ставится в соответствие разрядность адреса этого операнда. Этот адрес может иметь разрядность 16 или 32 бит. Если разрядность адреса для данной команды 16 бит, это означает, что команда содержит 16-разрядное смещение (см. [рис. 1](#)), оно соответствует 16-разрядному смещению адресного операнда относительно начала некоторого сегмента. В контексте [рис. 2](#) это смещение называется эффективным адресом. Если разрядность адреса 32 бит, это означает, что команда содержит 32-разрядное смещение (см. [рис. 1](#)), оно соответствует 32-разрядному смещению адресного операнда относительно начала сегмента и по его значению формируется 32-битное смещение в сегменте. С помощью префикса разрядности адреса можно изменить действующее по умолчанию значение разрядности адреса. Это изменение будет касаться только той команды, которой предшествует префикс.

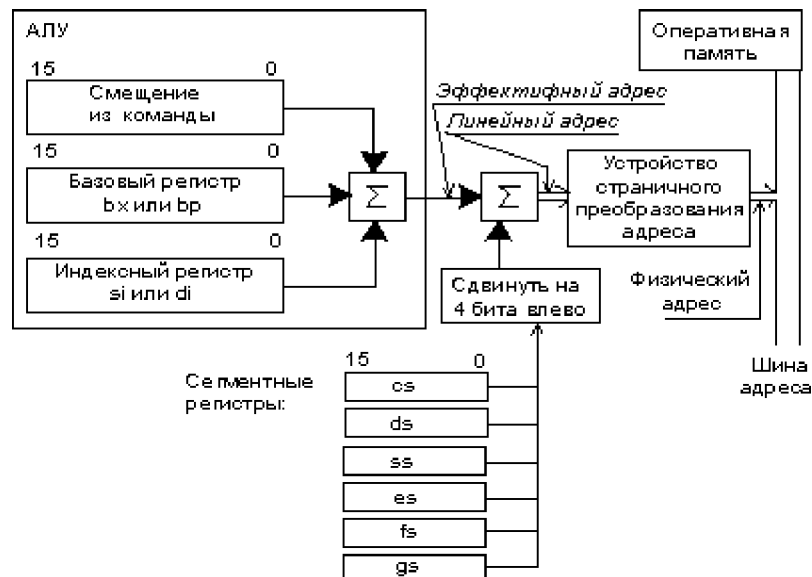


Рис. 2. Механизм формирования физического адреса в реальном режиме

- Префикс разрядности операнда аналогичен префиксу разрядности адреса, но указывает на разрядность операндов (32 или 16-разрядные), с которыми работает команда. В соответствии с какими правилами устанавливаются значения атрибутов разрядности адреса и операндов по умолчанию?
В реальном режиме и режиме виртуального i8086 значения этих атрибутов — 16 бит. В защищенном режиме значения атрибутов зависят от состояния бита *D* в дескрипторах исполняемых сегментов (см. урок 16). Если *D* = 0, то значения атрибутов, действующие по умолчанию, равны 16 бит; если *D* = 1, то 32 бит.
Значения префиксов разрядности операнда **66h** и разрядности адреса **67h**. Вы можете с помощью префикса разрядности адреса в реальном режиме использовать 32-разрядную адресацию, но при этом необходимо помнить об ограниченности размера сегмента величиной 64 Кбайт. Аналогично префиксу разрядности адреса вы можете использовать префикс разрядности операнда в реальном режиме для работы с 32-разрядными операндами (к примеру, в арифметических командах).
- Префикс повторения используется с цепочечными командами (командами обработки строк). Этот префикс “зацикливает” команду для обработки всех элементов цепочки. Система команд поддерживает два типа префиксов:
 - безусловные (rep — 0f3h), заставляющие повторяться цепочечную команду некоторое количество раз;
 - условные (repe/repz — 0f3h, repne/repnz — 0f2h), которые при зацикливании проверяют некоторые флаги, и в результате проверки возможен досрочный выход из цикла.
- **Код операции.** Обязательный элемент, описывающий операцию, выполняемую командой. Многим командам соответствует несколько кодов операций, каждый из которых определяет нюансы выполнения операции.

Последующие поля машинной команды определяют местоположение операндов, участвующих в операции, и особенности их использования. Рассмотрение этих полей связано со способами задания операндов в машинной команде и потому будет выполнено позже.

- **Байт режима адресации *modr/m*.** Значения этого байта определяет используемую форму адреса операндов. Операнды могут находиться в памяти в одном или двух регистрах. Если операнд находится в памяти, то байт *modr/m* определяет компоненты (смещение, базовый и индексный регистры), используемые для вычисления его эффективного адреса (см. [рис. 2](#)). В защищенном режиме для определения местоположения операнда в памяти может дополнительно использоваться байт *sib* (Scale-Index-Base — масштаб-индекс-база). Байт *modr/m* состоит из трех полей (см. [рис. 1](#)):
 - поле *mod* определяет количество байт, занимаемых в команде адресом операнда (см. [рис. 1](#), поле *смещение в команде*).
Поле *mod* используется совместно с полем *r/m*, которое указывает способ модификации адреса операнда смещение в команде.
К примеру, если *mod* = 00, это означает, что поле *смещение в команде* отсутствует, и адрес операнда определяется содержимым базового и (или) индексного регистра. Какие именно регистры будут использоваться для вычисления эффективного адреса, определяется значением этого байта.
Если *mod* = 01, это означает, что поле *смещение в команде* присутствует, занимает один байт и модифицируется содержимым базового и (или) индексного регистра.
Если *mod* = 10, это означает, что поле *смещение в команде* присутствует, занимает два или четыре байта (в зависимости от действующего по умолчанию или определяемого префиксом размера адреса) и модифицируется содержимым базового и (или) индексного регистра.
Если *mod* = 11, это означает, что операндов в памяти нет: они находятся в регистрах. Это же значение байта *mod* используется в случае, когда в команде применяется непосредственный операнд;
 - поле *reg/кон* определяет либо регистр, находящийся в команде на месте первого операнда, либо возможное расширение кода операции;
 - поле *r/m* используется совместно с полем *mod* и определяет либо регистр, находящийся в команде на месте первого операнда (если *mod* = 11), либо используемые для вычисления эффективного адреса (совместно с полем *смещение в команде*) базовые и индексные регистры.
- **Байт масштаб-индекс-база (байт *sib*)** используется для расширения возможностей адресации операндов.
На наличие байта *sib* в машинной команде указывает сочетание одного из значений 01 или 10 поля *mod* и значения поля *r/m* = 100. Байт *sib* состоит из трех полей:
 - поля *масштаба ss*. В этом поле размещается масштабный множитель для индексного компонента *index*, занимающего следующие три бита байта *sib*.
В поле *ss* может содержаться одно из следующих значений: 1, 2, 4, 8.
При вычислении эффективного адреса на это значение будет умножаться содержимое индексного регистра. Более подробно с практической точки зрения эта расширенная возможность индексации рассматривается на уроке 12 при обсуждении вопросов работы с массивами;
 - поля *index* — используется для хранения номера индексного регистра, который применяется для вычисления эффективного адреса операнда;
 - поля *base* — используется для хранения номера базового регистра, который также применяется для вычисления эффективного адреса операнда. Напомню, что в качестве базового и индексного регистров могут использоваться практически все регистры общего назначения.
- **Поле смещения в команде.** 8, 16 или 32-разрядное целое число со знаком, представляющее собой, полностью или частично (с учетом вышеприведенных рассуждений), значение эффективного адреса операнда.
- **Поле непосредственного операнда.** Необязательное поле, представляющее собой 8, 16 или 32-разрядный непосредственный операнд. Наличие этого поля, конечно, отражается на значении байта *modr/m*.

СПОСОБЫ ЗАДАНИЯ ОПЕРАНДОВ КОМАНДЫ

В ходе предыдущего изложения мы поневоле касались вопроса о том, где располагаются операнды, с которыми работает машинная команда, и как это отражается на содержимом ее полей.

В этой части занятия мы рассмотрим этот вопрос более систематизировано и в полном объеме. Это позволит нам уже со следующего занятия перейти непосредственно к практическим вопросам программирования на языке ассемблера.

Операнд задается неявно на микропрограммном уровне. В этом случае команда явно не содержит операндов. Алгоритм выполнения команды использует некоторые объекты по умолчанию (регистры, флаги в *eflags* и т. д.). Например, команды *cli* и *sti* неявно работают с флагом прерывания *if* в регистре *eflags*, а команда *xlat* неявно обращается к регистру *al* и строке в памяти по адресу, определяемому парой регистров *ds:bx*.

Операнд задается в самой команде (непосредственный операнд). Операнд находится в коде команды, то есть является ее частью. Для хранения такого операнда в команде выделяется поле длиной до 32 бит (см. [рис. 1](#)). Непосредственный операнд может быть только вторым операндом (источником). Операнд получатель может находиться либо в памяти, либо в регистре.

Например: `mov ax,0ffffh` пересылает в регистр `ax` шестнадцатеричную константу `ffff`. Команда `add sum,2` складывает содержимое поля по адресу `sum` с целым числом 2 и записывает результат по месту первого операнда, то есть в память.

Операнд находится в одном из регистров. Регистровые операнды указываются именами регистров. В качестве регистров могут использоваться:

- 32-разрядные регистры `EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `ESP`, `EBP`;
- 16-разрядные регистры `AX`, `BX`, `CX`, `DX`, `SI`, `DI`, `SP`, `BP`;
- 8-разрядные регистры `AH`, `AL`, `BH`, `BL`, `CH`, `CL`, `DH`, `DL`;
- сегментные регистры `CS`, `DS`, `SS`, `ES`, `FS`, `GS`.

Например, команда `add ax,bx` складывает содержимое регистров `ax` и `bx` и записывает результат в `bx`. Команда `dec si` уменьшает содержимое `si` на 1.

Операнд располагается в памяти. Это наиболее сложный и в то же время наиболее гибкий способ задания операндов. Он позволяет реализовать следующие два основных вида адресации: *прямую* и *косвенную*.

В свою очередь, *косвенная* адресация имеет следующие разновидности:

- косвенная базовая адресация; другое ее название — регистровая косвенная адресация;
- косвенная базовая адресация со смещением;
- косвенная индексная адресация со смещением;
- косвенная базовая индексная адресация;
- косвенная базовая индексная адресация со смещением.

Операндом является порт ввода/вывода. Как мы уже отмечали, помимо адресного пространства оперативной памяти микропроцессор поддерживает адресное пространство ввода-вывода, которое используется для доступа к устройствам ввода-вывода. Объем адресного пространства ввода-вывода составляет 64 Кбайт. Для любого устройства компьютера в этом пространстве выделяются адреса. Конкретное значение адреса в пределах этого пространства называется портом ввода-вывода. Физически порту ввода-вывода соответствует аппаратный регистр (не путать с регистром микропроцессора), доступ к которому осуществляется с помощью специальных команд ассемблера `in` и `out`.

Например,

`in al,60h` ;ввести байт из порта 60h

Регистры, адресуемые с помощью порта ввода-вывода, могут иметь разрядность 8, 16 или 32 бит, но для конкретного порта разрядность регистра фиксирована.

Команды [in](#) и [out](#) работают с фиксированной номенклатурой объектов. В качестве источника информации или получателя применяются так называемые регистры-аккумуляторы `eax`, `ax`, `al`. Выбор регистра определяется разрядностью порта. Номер порта может задаваться непосредственным операндом в командах `in` и `out` или значением в регистре `dx`. Последний способ позволяет динамически определить номер порта в программе.

Например:

```
mov dx,20h    ;записать номер порта 20h в регистр dx
mov al,20h    ;записать значение 20h в регистр al
out dx,al     ;вывести значение 20h в порт 20h
```

Операнд находится в стеке.

Команды могут совсем не иметь операндов, иметь один или два операнда. Большинство команд требуют двух операндов, один из которых является операндом-источником, а второй — операндом назначения. Важно то, что один операнд может располагаться в регистре или памяти, а второй операнд обязательно должен находиться в регистре или непосредственно в команде. Непосредственный операнд может быть только операндом-источником.

В двухоперандной машинной команде возможны следующие сочетания операндов:

- регистр—регистр;
- регистр—память;
- память—регистр;
- непосредственный операнд—регистр;
- непосредственный операнд—память.

У данного правила есть исключения, которые касаются:

- команд работы с цепочками, которые могут перемещать данные из памяти в память;
- команд работы со стеком, которые могут переносить данные из памяти в стек, также находящийся в памяти;
- команд типа умножения, которые кроме операнда, указанного в команде, используют еще и второй, неявный операнд.

Из перечисленных сочетаний операндов наиболее часто употребляются *регистр—память* и *память—регистр*. Ввиду их важности рассмотрим их подробнее. Обсуждение мы будем сопровождать примерами команд ассемблера, которые будут показывать, как изменяется формат команды ассемблера при применении того или иного вида адресации. В связи с этим посмотрите еще раз на [рис. 2](#), на котором показан принцип формирования физического адреса на адресной шине микропроцессора. Видно, что адрес операнда формируется как сумма двух составляющих — сдвинутого на 4 бит содержимого сегментного регистра и 16-битного эффективного адреса, который в общем случае вычисляется как сумма трех компонентов: базы, смещения и индекса.

Перечислим и затем рассмотрим особенности основных видов адресации операндов в памяти:

- [Прямая адресация](#)
- [Косвенная базовая \(регистровая\) адресация](#)
- [Косвенная базовая \(регистровая\) адресация со смещением](#)
- [Косвенная индексная адресация со смещением](#)
- [Косвенная базовая индексная адресация](#)
- [Косвенная базовая индексная адресация со смещением](#)

ПРЯМАЯ АДРЕСАЦИЯ

Это простейший вид адресации операнда в памяти, так как эффективный адрес содержится в самой команде и для его формирования не используются никаких дополнительных источников или регистров. Эффективный адрес берется непосредственно из поля смещения машинной команды (см. [рис. 1](#)), которое может иметь размер 8, 16, 32 бит. Это значение однозначно определяет байт, слово или двойное слово, расположенные в сегменте данных.

Прямая адресация может быть двух типов:

- *Относительная прямая адресация*. Используется для команд условных переходов, для указания относительного адреса перехода. Относительность такого перехода заключается в том, что в поле смещения машинной команды содержится 8, 16 или 32-битное значение, которое в результате работы команды будет складываться с содержимым регистра указателя команд `ip/eip`. В результате такого сложения получается адрес, по которому и осуществляется переход. Например:

```
jc ml      ;переход на метку ml, если флаг cf = 1
mov  al,2
...
ml:
```

- Несмотря на то, что в команде указана некоторая метка в программе, ассемблер вычисляет смещение этой метки относительно следующей команды (в нашем случае это `mov al,2`) и подставляет его в формируемую машинную команду `jc`.

- **Абсолютная прямая адресация.** В этом случае эффективный адрес является частью машинной команды, но формируется этот адрес только из значения поля смещения в команде. Для формирования физического адреса операнда в памяти микропроцессор складывает это поле со сдвинутым на 4 бит значением сегментного регистра. В команде ассемблера можно использовать несколько форм такой адресации.
Например:

```
mov     ax,dword ptr [0000]
;записать слово по адресу
;ds:0000 в регистр ax
```

Но такая адресация применяется редко — обычно используемым ячейкам в программе присваиваются символические имена. В процессе трансляции ассемблер вычисляет и подставляет значения смещений этих имен в формируемую им машинную команду в поле смещение в команде. В итоге получается так, что машинная команда прямо адресует свой операнд, имея, фактически, в одном из своих полей значение эффективного адреса. Например:

```
data     segment
perl     dw      5
...
data     ends
code     segment
mov      ax,data
mov      ds,ax
...
mov      ax,perl ;записать слово perl (его физический адрес ds:0000) в ax
```

Мы получим тот же результат, что и при использовании команды `mov ax,dword ptr [0000]`

КОСВЕННАЯ АДРЕСАЦИЯ

Остальные виды адресации относятся к *косвенным*. Слово “косвенный” в названии этих видов адресации означает то, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах, на которые указывают своим содержимым байт `modr/m` и, возможно, байт `sib`.

Косвенная базовая (регистровая) адресация

При такой адресации эффективный адрес операнда может находиться в любом из регистров общего назначения, кроме *sp/esp* и *bp/ebp* (это специфические регистры для работы с сегментом стека). Синтаксически в команде этот режим адресации выражается заключением имени регистра в квадратные скобки []. К примеру, команда **mov ax,[ecx]** помещает в регистр `ax` содержимое слова по адресу из сегмента данных со смещением, хранящимся в регистре `ecx`. Так как содержимое регистра легко изменить в ходе работы программы, данный способ адресации позволяет динамически назначить адрес операнда для некоторой машинной команды. Это свойство очень полезно, например, для организации циклических вычислений и для работы с различными структурами данных типа таблиц или массивов.

Косвенная базовая (регистровая) адресация со смещением

Этот вид адресации является дополнением предыдущего и предназначен для доступа к данным с известным смещением относительно некоторого базового адреса. Этот вид адресации удобно использовать для доступа к элементам структур данных, когда смещение элементов известно заранее, на стадии разработки программы, а базовый (начальный) адрес структуры должен вычисляться динамически, на стадии выполнения программы. Модификация содержимого базового регистра позволяет обратиться к одноименным элементам различных экземпляров однотипных структур данных.

К примеру, команда **mov ax,[edx+3h]** пересылает в регистр `ax` слова из области памяти по адресу: содержимое `edx` + `3h`. Команда **mov ax,mass[dx]** пересылает в регистр `ax` слово по адресу: содержимое `dx` плюс значение идентификатора `mass` (не забывайте, что транслятор присваивает каждому идентификатору значение, равное смещению этого идентификатора относительно начала сегмента данных).

Косвенная индексная адресация со смещением

Этот вид адресации очень похож на косвенную базовую адресацию со смещением. Здесь также для формирования эффективного адреса используется один из регистров общего назначения. Но индексная адресация обладает одной интересной особенностью, которая очень удобна для работы с массивами. Она связана с возможностью так называемого масштабирования содержимого индексного регистра. Что это такое? Посмотрите на [рис. 1](#). Нас интересует байт *sib*. При обсуждении структуры этого байта мы отмечали, что он состоит из трех полей. Одно из этих полей — поле масштаба *ss*, на значение которого умножается содержимое индексного регистра.

К примеру, в команде `mov ax,mas[si*2]` значение эффективного адреса второго операнда вычисляется выражением $mas + (si) * 2$. В связи с тем, что в ассемблере нет средств для организации индексации массивов, то программисту своими силами приходится ее организовывать.

Наличие возможности масштабирования существенно помогает в решении этой проблемы, но при условии, что размер элементов массива составляет 1, 2, 4 или 8 байт.

Косвенная базовая индексная адресация

При этом виде адресации эффективный адрес формируется как сумма содержимого двух регистров общего назначения: базового и индексного. В качестве этих регистров могут применяться любые регистры общего назначения, при этом часто используется масштабирование содержимого индексного регистра.

Например:

```
mov     eax,[esi][edx]
```

В данном примере эффективный адрес второго операнда формируется из двух компонентов $(esi) + (edx)$.

Косвенная базовая индексная адресация со смещением

Этот вид адресации является дополнением косвенной индексной адресации. Эффективный адрес формируется как сумма трех составляющих: содержимого базового регистра, содержимого индексного регистра и значения поля смещения в команде.

К примеру, команда `mov eax,[esi+5][edx]` пересылает в регистр *eax* двойное слово по адресу: $(esi) + 5 + (edx)$.

Команда `add ax,array[esi][ebx]` производит сложение содержимого регистра *ax* с содержимым слова по адресу: значение идентификатора *array* + $(esi) + (ebx)$.

○ Команды пересылки данных

Для удобства практического применения и отражения их специфики команды данной группы удобнее рассматривать в соответствии с их функциональным назначением, согласно которому их можно разбить на следующие группы команд:

- [пересылки данных общего назначения](#)
- [ввода-вывода в порт](#)
- [работы с адресами и указателями](#)
- [преобразования данных](#)
- [работы со стеком](#)

КОМАНДЫ ПЕРЕСЫЛКИ ДАННЫХ ОБЩЕГО НАЗНАЧЕНИЯ

К этой группе относятся следующие команды:

mov <операнд назначения>, <операнд-источник>

xchg <операнд1>, <операнд2>

mov - это основная команда пересылки данных. Она реализует самые разнообразные варианты пересылки. Отметим особенности применения этой команды:

- командой *mov* нельзя осуществить пересылку из одной области памяти в другую. Если такая необходимость возникает, то нужно использовать в качестве промежуточного буфера любой доступный в данный момент регистр общего назначения. К примеру, рассмотрим фрагмент программы для пересылки байта из ячейки *fls* в ячейку *fld*:

```
masm
model    small
.data
fls      db      5
fld      db      ?
.code
start:
...
        mov     al, fls
        mov     fld, al
        ...
end      start
```

- нельзя загрузить в сегментный регистр значение непосредственно из памяти. Поэтому для выполнения такой загрузки нужно использовать промежуточный объект. Это может быть регистр общего назначения или стек. Если вы посмотрите листинги 3.1 и 5.1, то увидите в начале сегмента кода две команды *mov*, выполняющие настройку сегментного регистра *ds*. При этом из-за невозможности загрузить напрямую в сегментный регистр значение адреса сегмента, содержащееся в предопределенной переменной *@data*, приходится использовать регистр общего назначения *ax*;
- нельзя переслать содержимое одного сегментного регистра в другой сегментный регистр. Это объясняется тем, что в системе команд нет соответствующего кода операции. Но необходимость в таком действии часто возникает. Выполнить такую пересылку можно, используя в качестве промежуточных все те же регистры общего назначения. Вот пример инициализации регистра *es* значением из регистра *ds*:

```
mov     ax, ds
mov     es, ax
```

- Но есть и другой, более красивый способ выполнения данной операции — использование стека и команд *push* и *pop*:

```
push    ds      ;поместить значение регистра ds в стек
pop     es      ;записать в es число из стека
```

- нельзя использовать сегментный регистр *cs* в качестве операнда назначения. Причина здесь простая. Дело в том, что в архитектуре микропроцессора пара *cs:ip* всегда содержит адрес команды, которая должна выполняться следующей. Изменение командой *mov* содержимого регистра *cs* фактически означало бы операцию перехода, а не пересылки, что недопустимо.

Для двунаправленной пересылки данных применяют команду **xchg**. Для этой операции можно, конечно, применить последовательность из нескольких команд *mov*, но из-за того, что операция обмена используется довольно часто, разработчики системы команд микропроцессора посчитали нужным ввести отдельную команду

обмена `xchg`. Естественно, что операнды должны иметь один тип. Не допускается (как и для всех команд ассемблера) обменивать между собой содержимое двух ячеек памяти. К примеру,

```
xchg    ax,bx    ;обменять содержимое регистров ax и bx
xchg    ax,word ptr [si]    ;обменять содержимое регистра ax
                        ;и слова в памяти по адресу в [si]
```

КОМАНДЫ ВВОДА-ВЫВОДА В ПОРТ

На уроке 6 при обсуждении вопроса о том, [где могут находиться операнды машинной команды](#), мы упоминали **порт ввода-вывода**.

Посмотрите на **рис. 1**. На нем показана сильно упрощенная, концептуальная схема управления оборудованием компьютера.

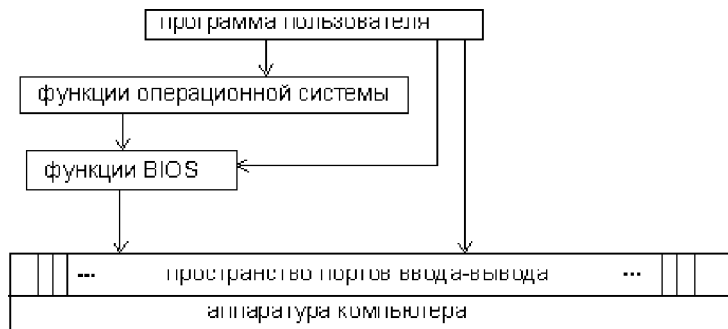


Рис. 1. Концептуальная схема управления оборудованием компьютера

Как видно из рис. 1, самым нижним уровнем является уровень BIOS, на котором работа с оборудованием ведется напрямую через порты. Тем самым реализуется концепция независимости от оборудования. При замене оборудования необходимо будет лишь подправить соответствующие функции BIOS, переориентировав их на новые адреса и логику работы портов.

Принципиально управлять устройствами напрямую через порты несложно. Сведения о номерах портов, их разрядности, формате управляющей информации приводятся в техническом описании устройства. Необходимо знать лишь конечную цель своих действий, алгоритм, в соответствии с которым работает конкретное устройство, и порядок программирования его портов. То есть, фактически, нужно знать, что и в какой последовательности нужно послать в порт (при записи в него) или считать из него (при чтении) и как следует трактовать эту информацию. Для этого достаточно всего двух команд, присутствующих в системе команд микропроцессора:

in аккумулятор,номер_порта — ввод в аккумулятор из порта с номером номер_порта;

out порт,аккумулятор — вывод содержимого аккумулятора в порт с номером номер_порта.

Команды работы с адресами и указателями памяти

При написании программ на ассемблере производится интенсивная работа с адресами операндов, находящимися в памяти. Для поддержки такого рода операций есть специальная группа команд, в которую входят следующие команды:

lea назначение,источник — загрузка эффективного адреса;

lds назначение,источник — загрузка указателя в регистр сегмента данных ds;

les назначение,источник — загрузка указателя в регистр дополнительного сегмента данных es;

lgs назначение,источник — загрузка указателя в регистр дополнительного сегмента данных gs;

lfs назначение,источник — загрузка указателя в регистр дополнительного сегмента данных fs;

lss назначение,источник — загрузка указателя в регистр сегмента стека ss.

Команда *lea* похожа на команду *mov* тем, что она также производит пересылку. Однако, обратите внимание, команда *lea* производит пересылку не данных, а эффективного адреса данных (то есть смещения данных относительно начала сегмента данных) в регистр, указанный операндом назначение.

Часто для выполнения некоторых действий в программе недостаточно знать значение одного лишь эффективного адреса данных, а необходимо иметь полный указатель на данные. Вы помните, что полный указатель на данные состоит из сегментной составляющей и смещения.

Все остальные команды этой группы позволяют получить в паре регистров такой полный указатель на операнд в памяти. При этом *имя сегментного регистра*, в который помещается сегментная составляющая адреса, определяется кодом операции. Соответственно, *смещение* помещается в регистр общего назначения, указанный операндом *назначение*.

Но не все так просто с операндом *источник*. На самом деле, в команде в качестве источника нельзя указывать непосредственно имя операнда в памяти, на который мы бы хотели получить указатель.

Предварительно необходимо получить само значение полного указателя в некоторой области памяти и указать в команде получения полного адреса имя этой области. Для выполнения этого действия необходимо вспомнить директивы резервирования и инициализации памяти.

При применении этих директив возможен частный случай, когда в поле операндов указывается имя другой директивы определения данных (фактически, имя переменной). В этом случае в памяти формируется адрес этой переменной. Какой адрес будет сформирован (эффективный или полный), зависит от применяемой директивы. Если это *dw*, то в памяти формируется только 16-битное значение эффективного адреса, если же *dd* — в память записывается полный адрес. Размещение этого адреса в памяти следующее: в младшем слове находится смещение, в старшем — 16-битная сегментная составляющая адреса.

Например, при организации работы с цепочкой символов удобно поместить ее начальный адрес в некоторый регистр и далее в цикле модифицировать это значение для последовательного доступа к элементам цепочки. В листинге 1 производится копирование строки байт *str_1* в строку байт *str_2*.

В строках 12 и 13 в регистры *si* и *di* загружаются значения эффективных адресов переменных *str_1* и *str_2*.

В строках 16 и 17 производится пересылка очередного байта из одной строки в другую. Указатели на позиции байтов в строках определяются содержимым регистров *si* и *di*. Для пересылки очередного байта необходимо увеличить на единицу регистры *si* и *di*, что и делается командами сложения *inc* (строки 18, 19). После этого программу необходимо зациклить до обработки всех символов строки.

Листинг 1. Копирование строки

```
<1>;-----Prg_7_2.asm-----
<2>    masm
<3>    model    small
<4>    .data
<5>    ...
<6>    str_1 db      'Ассемблер — базовый язык компьютера'
<7>    str_2 db      50 dup ( ' ')
<8>    full_pnt dd str_1
<9>    ...
<10>   .code
<11>   start:
<12>   ...
<13>   lea     si,str_1
<14>   lea     di,str_2
<15>   les     bx,full_pnt      ;полный указатель на str1 в паре es:bx
<16>   ml:
<17>   mov     al,[si]
<18>   mov     [di],al
<19>   inc     si
<20>   inc     di
<21> ;цикл на метку ml до пересылки всех символов
<22> ...
<23>   end     start
```

Необходимость использования команд получения полного указателя данных в памяти, то есть адреса сегмента и значения смещения внутри сегмента, возникает, в частности, при работе с цепочками.

В строке 14 листинга 1 в двойном слове *full_pnt* формируются сегментная часть адреса и смещение для переменной *str_1*. При этом 2 байта смещения занимают младшее слово *full_pnt*, а значение сегментной составляющей адреса — старшее слово *full_pnt*. В строке 14 командой *les* эти компоненты адреса помещаются в регистры *bx* и *es*.

КОМАНДЫ ПРЕОБРАЗОВАНИЯ ДАННЫХ

К этой группе можно отнести множество команд микропроцессора, но большинство из них имеют те или иные особенности, которые требуют отнести их к другим функциональным группам.

Поэтому из всей совокупности команд микропроцессора непосредственно к командам преобразования данных можно отнести только одну команду:

xlat [адрес_таблицы_перекодировки]

Это очень интересная и полезная команда. Ее действие заключается в том, что она замещает значение в регистре `al` другим байтом из таблицы в памяти, расположенной по адресу, указанному операндом `адрес_таблицы_перекодировки`.

Слово “таблица” весьма условно — по сути это просто строка байт. Адрес байта в строке, которым будет производиться замещение содержимого регистра `al`, определяется суммой $(bx) + (al)$, то есть содержимое `al` выполняет роль индекса в байтовом массиве.

При работе с командой `xlat` обратите внимание на следующий тонкий момент. Несмотря на то, что в команде указывается адрес строки байт, из которой должно быть извлечено новое значение, этот адрес должен быть предварительно загружен (например, с помощью команды `lea`) в регистр `bx`. Таким образом, операнд `адрес_таблицы_перекодировки` на самом деле не нужен (необязательность операнда показана заключением его в квадратные скобки). Что касается строки байт (таблицы перекодировки), то она представляет собой область памяти размером от 1 до 255 байт (диапазон числа без знака в 8-битном регистре).

В качестве иллюстрации работы данной команды мы рассмотрим программу, которая преобразует двузначное шестнадцатеричное число, вводимое с клавиатуры (то есть в символьном виде), в эквивалентное двоичное представление в регистре `al`. Ниже (листинг 2) приведен вариант этой программы с использованием команды `xlat`.

Листинг 2. Использование таблицы перекодировки

```
<1>;-----Prg_7_3.asm-----
<2>;Программа преобразования двузначного шестнадцатеричного числа
<3>;в двоичное представление с использованием команды xlat.
<4>;Вход: исходное шестнадцатеричное число; вводится с клавиатуры.
<5>;Выход: результат преобразования в регистре al.
<6>.data                                ;сегмент данных
<7> message      db      'Введите две шестнадцатеричные цифры,$'
<8>  tab1         db      48 dup (0),0,1,2,3,4,5,6,7,8,9, 8 dup (0),
<9>              db      0ah,0bh,0ch,0dh,0eh,0fh,27 dup (0)
<10>             db      0ah,0bh,0ch,0dh,0eh,0fh, 153 dup (0)
<11> .stack      256              ;сегмент стека
<12> .code
<13> ;начало сегмента кода
<14> proc        main            ;начало процедуры main
<15>   mov        ax,@data        ;физический адрес сегмента данных в регистр ax
<16>   mov        ds,ax          ;ax записываем в ds
<17>   lea        bx,tab1        ;загрузка адреса строки байт в регистр bx
<18>   mov        ah,9
<19>   mov        dx,offset message
<20>   int        21h            ;вывести приглашение к вводу
<21>   xor        ax,ax          ;очистить регистр ax
<22>   mov        ah,1h          ;значение 1h в регистр ah
<23>   int        21h            ;вводим первую цифру в al
<24>   xlat                     ;перекодировка первого введенного символа в al
<25>   mov        dl,al
<26>   shl        dl,4           ;сдвиг dl влево для освобождения места для младшей цифры
<27>   int        21h            ;ввод второго символа в al
<28>   xlat                     ;перекодировка второго введенного символа в al
<29>   add        al,dl          ;складываем для получения результата
<30>   mov        ax,4c00h       ;пересылка 4c00h в регистр ax
<31>   int        21h            ;завершение программы
<32> endp        main            ;конец процедуры main
<33> code        ends           ;конец сегмента кода
<34> endmain      ;конец программы с точкой входа main
```


Сама по себе программа проста; сложность вызывает обычно формирование таблицы перекодировки. Обсудим этот момент подробнее.

Прежде всего нужно определиться с значениями тех байтов, которые вы будете изменять. В нашем случае это символы шестнадцатеричных цифр. Сконструируем в сегменте данных таблицу, в которой на места байтов, соответствующих символам шестнадцатеричных цифр, помещаем их новые значения, то есть двоичные эквиваленты шестнадцатеричных цифр. Строки 8-10 листинга 2 демонстрируют, как это сделать. Байты этой таблицы, смещения которых не совпадают со значением кодов шестнадцатеричных цифр, нулевые. Таковыми являются первые 48 байт таблицы, промежуточные байты и часть в конце таблицы.

Желательно определить все 256 байт таблицы. Дело в том, что если мы ошибочно поместим в `al` код символа, отличный от символа шестнадцатеричной цифры, то после выполнения команды `xlat` получим непредсказуемый результат. В случае листинга 2 это будет ноль, что не совсем корректно, так как непонятно, что же в действительности было в `al` — код символа “0” или что-то другое.

Поэтому, наверное, есть смысл здесь поставить “защиту от дурака”, поместив в неиспользуемые байты таблицы какой-нибудь определенный символ. После каждого выполнения `xlat` нужно будет просто контролировать значение в `al` на предмет совпадения с этим символом, и если оно произошло, выдавать сообщение об ошибке.

После того как таблица составлена, с ней можно работать. В сегменте команд строка 18 инициализирует регистр `bx` значением адреса таблицы `tbl`. Далее все очень просто. Поочередно вводятся символы двух шестнадцатеричных цифр, и производится их перекодировка в соответствующие двоичные эквиваленты.

КОМАНДЫ РАБОТЫ СО СТЕКОМ

Эта группа представляет собой набор специализированных команд, ориентированных на организацию гибкой и эффективной работы со стеком.

Стек — это область памяти, специально выделяемая для временного хранения данных программы. Важность стека определяется тем, что для него в структуре программы предусмотрен отдельный сегмент. На тот случай, если программист забыл описать сегмент стека в своей программе, компоновщик `tlink` выдаст предупреждающее сообщение.

Для работы со стеком предназначены три регистра:

- **ss** — сегментный регистр стека;
- **sp/esp** — регистр указателя стека;
- **bp/ebp** — регистр указателя базы кадра стека.

Размер стека зависит от режима работы микропроцессора и ограничивается 64 Кбайт (или 4 Гбайт в защищенном режиме).

В каждый момент времени доступен только один стек, адрес сегмента которого содержится в регистре `ss`. Этот стек называется *текущим*. Для того чтобы обратиться к другому стеку (“переключить стек”), необходимо загрузить в регистр `ss` другой адрес. Регистр `ss` автоматически используется процессором для выполнения всех команд, работающих со стеком.

Перечислим еще некоторые особенности работы со стеком:

- запись и чтение данных в стеке осуществляется в соответствии с принципом LIFO (Last In First Out — “последним пришел, первым ушел”);
- по мере записи данных в стек последний растет в сторону младших адресов. Эта особенность заложена в алгоритм команд работы со стеком;
- при использовании регистров `esp/sp` и `ebp/br` для адресации памяти ассемблер автоматически считает, что содержащиеся в нем значения представляют собой смещения относительно сегментного регистра `ss`.

В общем случае стек организован так, как показано на рис. 2.

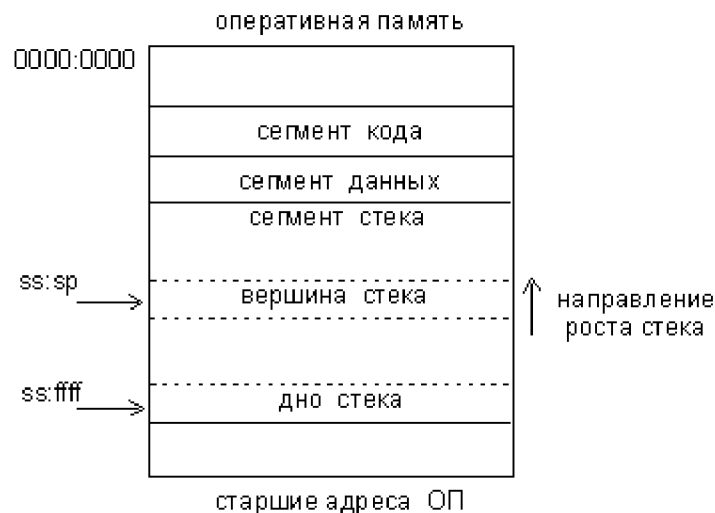


Рис. 2. Концептуальная схема организации стека

Для работы со стеком предназначены регистры *ss*, *esp/sp* и *ebp/bp*.

Эти регистры используются комплексно, и каждый из них имеет свое функциональное назначение.

Регистр *esp/sp* всегда указывает на вершину стека, то есть содержит смещение, по которому в стек был занесен последний элемент. Команды работы со стеком неявно изменяют этот регистр так, чтобы он указывал всегда на последний записанный в стек элемент. Если стек пуст, то значение *esp* равно адресу последнего байта сегмента, выделенного под стек.

При занесении элемента в стек процессор уменьшает значение регистра *esp*, а затем записывает элемент по адресу новой вершины.

При извлечении данных из стека процессор копирует элемент, расположенный по адресу вершины, а затем увеличивает значение регистра указателя стека *esp*.

Таким образом, получается, что стек растет вниз, в сторону уменьшения адресов.

Что делать, если нам необходимо получить доступ к элементам не на вершине, а внутри стека?

Для этого применяют регистр *ebp*. Регистр ***ebp*** — *регистр указателя базы кадра стека*.

Например, типичным приемом при входе в подпрограмму является передача нужных параметров путем записи их в стек. Если подпрограмма тоже активно работает со стеком, то доступ к этим параметрам становится проблематичным. Выход в том, чтобы после записи нужных данных в стек сохранить адрес вершины стека в указателе кадра (базы) стека — регистре *ebp*. Значение в *ebp* в дальнейшем можно использовать для доступа к переданным параметрам.

Начало стека расположено в старших адресах памяти. На рис. 2 этот адрес обозначен парой *ss:ffff*. Смещение *ffff* приведено здесь условно. Реально это значение определяется величиной, которую программист задает при описании сегмента стека в своей программе.

К примеру, для программы в [листинге 2](#) началу стека будет соответствовать пара *ss:0100h*. Адресная пара *ss:ffff* — это максимальное для реального режима значение адреса начала стека, так как размер сегмента в нем ограничен величиной 64 Кбайт (*0ffffh*).

Для организации работы со стеком существуют специальные команды записи и чтения.

***push* источник** — запись значения источник в вершину стека.

Интерес представляет алгоритм работы этой команды, который включает следующие действия (рис. 3):

- $(sp) = (sp) - 2$; значение *sp* уменьшается на 2;
- значение из источника записывается по адресу, указываемому парой *ss:sp*.

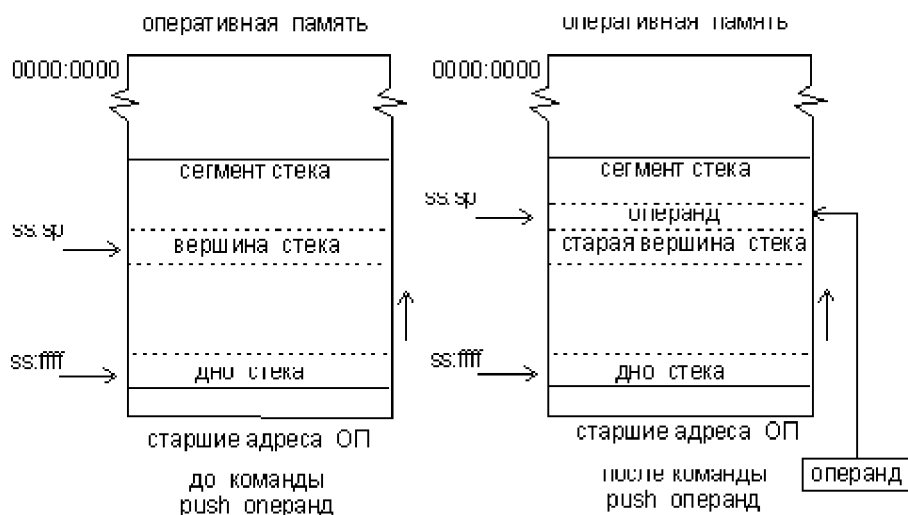


Рис. 3. Принцип работы команды push

pop назначение — запись значения из вершины стека по месту, указанному операндом назначения. Значение при этом “снимается” с вершины стека.

Алгоритм работы команды pop обратен алгоритму команды push (рис. 4):

- запись содержимого вершины стека по месту, указанному операндом назначения;
- $(sp) = (sp) + 2$; увеличение значения sp.

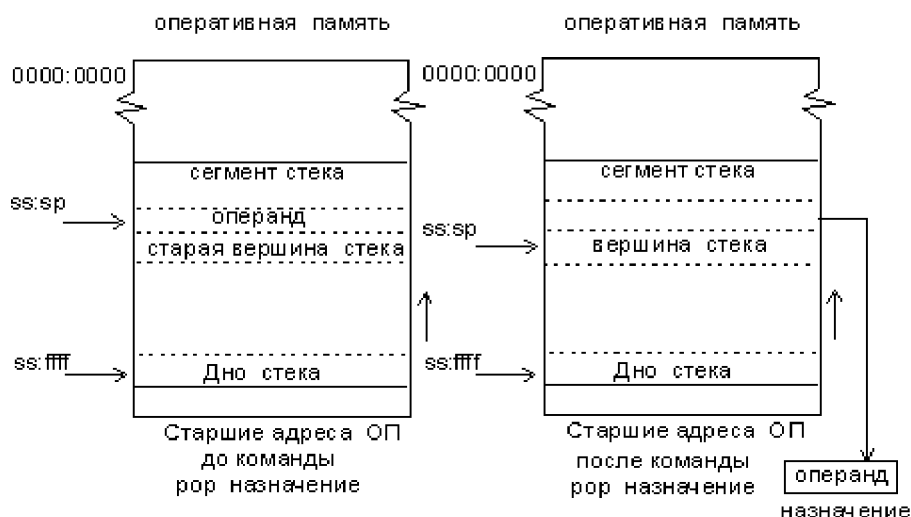


Рис. 4. Принцип работы команды pop

pusha — команда групповой записи в стек.

По этой команде в стек последовательно записываются регистры ax, cx, dx, bx, sp, bp, si, di. Заметим, что записывается оригинальное содержимое sp, то есть то, которое было до выдачи команды pusha (рис. 5).

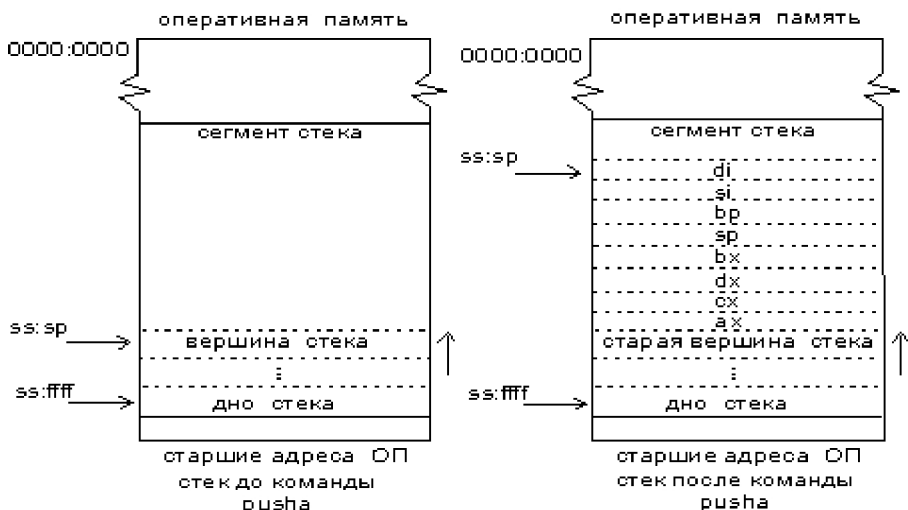


Рис. 5. Принцип работы команды pusha

pushaw — почти синоним команды pusha. В чем разница? На уроке 5 мы обсуждали один из атрибутов сегмента — атрибут разрядности. Он может принимать значение use16 или use32. Рассмотрим работу команд pusha и pushaw при каждом из этих атрибутов:

- use16 — алгоритм работы pushaw аналогичен алгоритму pusha.
- use32 — pushaw не изменяется (то есть она нечувствительна к разрядности сегмента и всегда работает с регистрами размером в слово — ax, cx, dx, bx, sp, bp, si, di). Команда pusha чувствительна к установленной разрядности сегмента и при указании 32-разрядного сегмента работает с соответствующими 32-разрядными регистрами, то есть eax, ecx, edx, ebx, esp, ebp, esi, edi.

pushad — выполняется аналогично команде pusha, но есть некоторые особенности, которые вы можете узнать из “Справочника команд”.

Следующие три команды выполняют действия, обратные вышеописанным командам:

popa;
popaw;
popad.

Группа команд, описанная ниже, позволяет сохранить в стеке регистр флагов и записать слово или двойное слово в стек. Отметим, что перечисленные ниже команды — единственные в системе команд микропроцессора, которые позволяют получить доступ (и которые нуждаются в этом доступе) ко всему содержимому регистра флагов.

pushf — сохраняет регистр флагов в стек.

Работа этой команды зависит от атрибута размера сегмента:

- use16 — в стек записывается регистр flags размером 2 байта;
- use32 — в стек записывается регистр eflags размером 4 байта.

pushfw — сохранение в стеке регистра флагов размером в слово. Всегда работает как pushf с атрибутом use16.

pushfd — сохранение в стеке регистра флагов flags или eflags в зависимости от атрибута разрядности сегмента (то есть то же, что и pushf).

Аналогично, следующие три команды выполняют действия, обратные рассмотренным выше операциям:

popf
popfw
popfd

И в заключение отметим основные виды операции, когда использование стека практически неизбежно:

- вызов подпрограмм;
- временное сохранение значений регистров;
- определение локальных переменных.

Лекция 4

Команды микропроцессора.

Структура машинной команды *Обзор группы арифметических команд и данных*
Арифметические операции над целыми двоичными числами *Вспомогательные команды для целочисленных операций* *Арифметические операции над двоично-десятичными числами* *Логические команды* *Команды сдвига*

Устройства для выполнения операций

Микропроцессор может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных блока:

- устройство для выполнения целочисленных операций;
- устройство с плавающей точкой.

Каждое из этих устройств имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.

ОБЗОР ГРУППЫ АРИФМЕТИЧЕСКИХ КОМАНД И ДАННЫХ

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд. На рис. 1 приведена классификация команд этой группы.

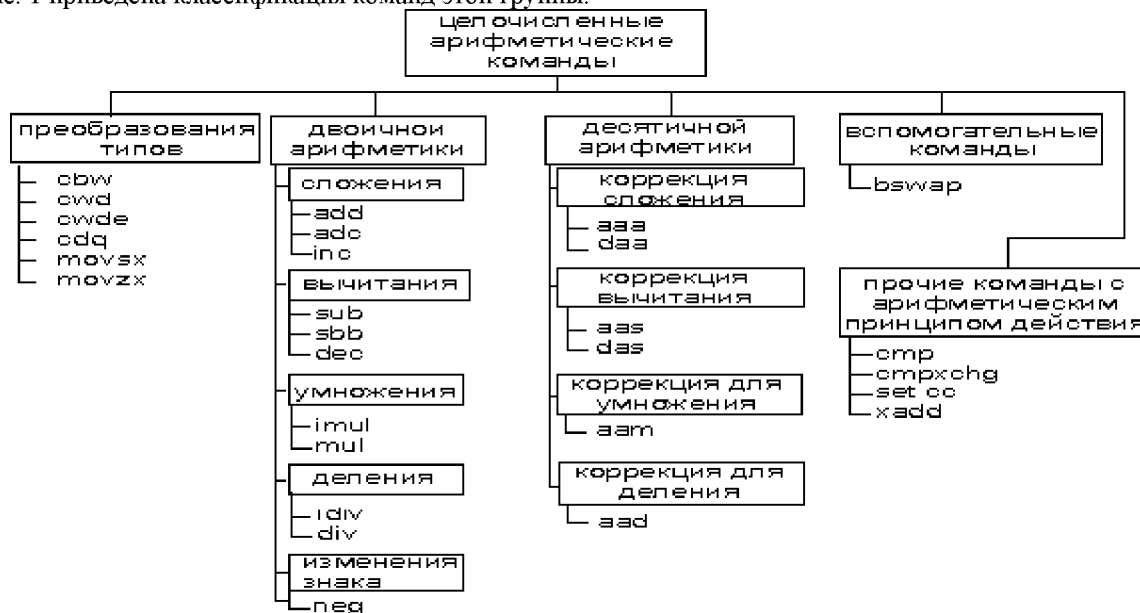


Рис. 1. Классификация арифметических команд

Группа арифметических целочисленных команд работает с двумя типами чисел:

- целыми двоичными числами. Числа могут иметь знаковый разряд или не иметь такового, то есть быть числами со знаком или без знака;
- целыми десятичными числами.

Целое двоичное число с фиксированной точкой — это число, закодированное в двоичной системе счисления.

Размерность целого двоичного числа может составлять 8, 16 или 32 бита. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа. Это 7-й, 15-й или 31-й биты для чисел соответствующей размерности (см. *Типы данных*). При этом интересно то, что среди арифметических команд есть всего две команды, которые действительно учитывают этот старший разряд как знаковый, — это команды целочисленного умножения и деления `imul` и `idiv`. В остальных случаях ответственность за действия со знаковыми числами и, соответственно, со знаковым разрядом ложится на программиста. К этому вопросу мы вернемся чуть позже. Диапазон значений двоичного числа зависит от его размера и трактовки старшего бита либо как старшего значащего бита числа, либо как бита знака числа (табл. 1).

Таблица 1. Диапазон значений двоичных чисел

Размерность поля	Целое без знака	Целое со знаком
Байт	0...255	–128...+127
Слово	0...65 535	–32 768...+32 767
двойное слово	0...4 294 967 295	–2 147 483 648...+2 147 483 647

Как описать числа с фиксированной точкой в программе?

Это делается с использованием *директив описания данных*. К примеру, последовательность описаний двоичных чисел из сегмента данных листинга 1 (помните о принципе “младший байт по младшему адресу”) будет выглядеть в памяти так, как показано на рис. 2.

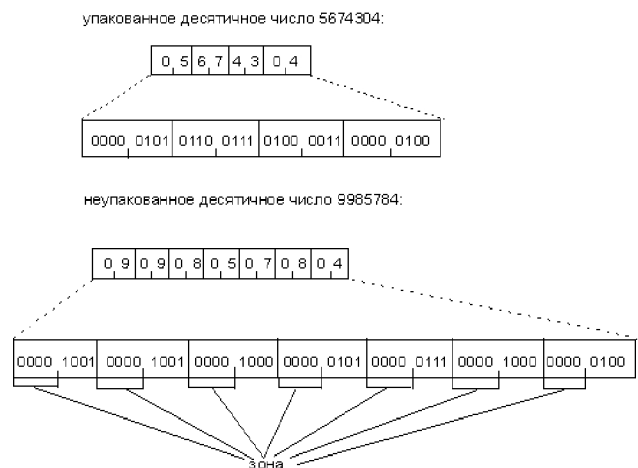
```

Листинг 1. Числа с фиксированной точкой
;prg_8_1.asm
masm
model    small
stack    256
.data    ;сегмент данных
per_1    db      23
per_2    dw      9856
per_3    dd      9875645
per_4    dw      29857
.code    ;сегмент кода
main:    ;точка входа в программу
        mov     ax,@data    ;связываем регистр dx с сегментом
        mov     ds,ax       ;данных через регистр ax
exit:    ;посмотрите в отладчике дампы сегмента данных
        mov     ax,4c00h    ;стандартный выход
        int     21h
end       main    ;конец программы

```

Десятичные числа — специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех бит. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом двоично-десятичном коде (BCD — Binary-Coded Decimal). Микропроцессор хранит BCD-числа в двух форматах (рис. 3):

- **упакованном формате** — в этом формате каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером 4 бита. При этом код старшей цифры числа занимает старшие 4 бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99;
- **неупакованном формате** — в этом формате каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие четыре бита имеют нулевое значение. Это так называемая зона. Следовательно, диапазон представления десятичного неупакованного числа в одном байте составляет от 0 до 9.



Для описания двоично-десятичных чисел в программе можно использовать только две директивы описания и инициализации данных — *db* и *dt*. Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип “младший байт по младшему адресу”, что, как мы увидим далее, очень удобно для их обработки. И вообще, при использовании такого типа данных как BCD-числа, порядок описания этих чисел в программе и алгоритм их обработки — это дело вкуса и личных пристрастий программиста. Это станет ясно после того, как мы ниже рассмотрим основы работы с BCD-числами. К примеру, приведенная в сегменте данных листинга 2 последовательность описаний BCD-чисел будет выглядеть в памяти так, как показано на рис. 4.

```

Листинг 2. BCD-числа
;prg_8_2.asm
masm
model    small
stack    256
.data    ;сегмент данных
per_1    db      2,3,4,6,8,2 ;неупакованное BCD-число 286432
per_3    dt      9875645 ;упакованное BCD-число 9875645
.code    ;сегмент кода
main:    ;точка входа в программу
        mov     ax,@data        ;связываем регистр dx с сегментом
        mov     ds,ax          ;данных через регистр ax
exit:    ;посмотрите в отладчике дампы сегмента данных
        mov     ax,4c00h        ;стандартный выход
        int     21h
end      main    ;конец программы

```

После столь подробного обсуждения объектов, с которыми работают арифметические операции, можно приступить к рассмотрению средств их обработки на уровне системы команд микропроцессора.

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ НАД ЦЕЛЫМИ ДВОИЧНЫМИ ЧИСЛАМИ

В данном разделе мы рассмотрим особенности каждого из четырех основных арифметических действий для двоичных чисел со знаком и без знака:

- [сложение двоичных чисел без знака;](#)
- [сложение двоичных чисел со знаком;](#)
- [вычитание двоичных чисел без знака;](#)
- [вычитание двоичных чисел со знаком;](#)
- [вычитание и сложение операндов большой размерности;](#)
- [умножение чисел без знака;](#)
- [умножение чисел со знаком;](#)
- [деление чисел без знака;](#)
- [деление чисел со знаком.](#)

Сложение двоичных чисел без знака

Микропроцессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда. Например, при сложении операндов размером в байт результат не должен превышать число 255. Если это происходит, то результат оказывается неверным. Рассмотрим, почему так происходит. К примеру, выполним сложение: $254 + 5 = 259$ в двоичном виде. $11111110 + 0000101 = 1\ 00000011$. Результат вышел за пределы восьми бит и правильное его значение укладывается в 9 бит, а в 8-битовом поле операнда осталось значение 3, что, конечно, неверно. В микропроцессоре этот исход сложения прогнозируется и предусмотрены специальные средства для фиксирования подобных ситуаций и их обработки. Так, для фиксирования ситуации выхода за разрядную сетку результата, как в данном случае, предназначен флаг переноса *cf*. Он располагается в бите 0 регистра флагов *eflags/flags*. Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Естественно, что программист должен предусматривать возможность такого исхода операции сложения и средства для корректировки. Это предполагает включение участков кода после операции сложения, в которых анализируется флаг *cf*. Анализ этого флага можно провести различными способами. Самый простой и доступный — использовать команду условного перехода [*jcc*](#). Эта команда в качестве операнда имеет имя метки в текущем сегменте кода. Переход на эту метку осуществляется в случае, если в результате работы предыдущей команды флаг *cf* установился в 1. Если теперь посмотреть, то видно, что в системе команд микропроцессора имеются три команды двоичного сложения:

- **inc операнд** — операция инкремента, то есть увеличения значения операнда на 1;
- **add операнд_1,операнд_2** — команда сложения с принципом действия: $\text{операнд_1} = \text{операнд_1} + \text{операнд_2}$
- **adc операнд_1,операнд_2** — команда сложения с учетом флага переноса *cf*. $\text{операнд_1} = \text{операнд_1} + \text{операнд_2} + \text{значение_cf}$

Обратите внимание на последнюю команду — это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления такой единицы мы уже рассмотрели. Таким образом, команда `adc` является средством микропроцессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые микропроцессором длины стандартных полей.

Рассмотрим пример вычисления суммы чисел (листинг 3).

Листинг 3. Вычисление суммы чисел

```
<1>      ;prg_8_3.asm
<2>      masm
<3>      model    small
<4>      stack    256
<5>      .data
<6>      a        db        254
<7>      .code    ;сегмент кода
<8>      main:
<9>      mov      ax,@data
<10>     mov      ds,ax
<11>     ...
<12>     xor      ax,ax
<13>     add      al,17
<14>     add      al,a
<15>     jnc      ml      ;если нет переноса, то перейти на ml
<16>     adc      ah,0     ;в ah сумма с учетом переноса
<17>     ml:      ...
<18>     exit:
<19>     mov      ax,4c00h      ;стандартный выход
<20>     int      21h
<21>     end      main      ;конец программы
```

В листинге 3 в строках 13–14 создана ситуация, когда результат сложения выходит за границы операнда. Эта возможность учитывается строкой 15, где команда `jnc` (хотя можно было обойтись и без нее) проверяет состояние флага `cf`. Если он установлен в 1, то это признак того, что результат операции получился больше по размеру, чем размер операнда, и для его корректировки необходимо выполнить некоторые действия. В данном случае мы просто полагаем, что границы операнда расширяются до размера `ax`, для чего учитываем перенос в старший разряд командой `adc` (строка 15).

Сложение двоичных чисел со знаком

Теперь настала пора раскрыть небольшой секрет. Дело в том, что на самом деле микропроцессор не подозревает о различии между числами со знаком и без знака. Вместо этого у него есть средства фиксирования возникновения характерных ситуаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели при обсуждении сложения чисел без знака:

- флаг переноса `cf`, установка которого в 1 говорит о том, что произошел выход за пределы разрядности операндов;
- команду `adc`, которая учитывает возможность такого выхода (перенос из младшего разряда).

Другое средство — это регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения `of` в регистре `eflags` (бит 11).

Вы, конечно, помните, как представляются числа в компьютере: положительные — в двоичном коде, отрицательные — в дополнительном коде. Рассмотрим различные варианты сложения чисел. Примеры призваны показать поведение двух старших битов операндов и правильность результата операции сложения.

Пример 1.

```
30566 = 01110111 01100110
+
00687 = 00000010 10101111
=
31253 = 01111010 00010101
```

Следим за переносами из 14-го и 15-го разрядов и правильностью результата: переносов нет, результат правильный.

Пример 2.

```

30566 = 01110111 01100110
+
30566 = 01110111 01100110
=
61132 = 11101110 11001100

```

Произошел перенос из 14-го разряда; из 15-го разряда переноса нет. Результат неправильный, так как имеется переполнение — значение числа получилось больше, чем то, которое может иметь 16-битное число со знаком (+32 767).

Пример 3.

```

-30566 = 10001000 10011010
+
-04875 = 11101100 11110101
=
-35441 = 01110101 10001111

```

Произошел перенос из 15-го разряда, из 14-го разряда нет переноса. Результат неправильный, так как вместо отрицательного числа получилось положительное (в старшем бите находится 0).

Пример 4.

```

-4875 = 11101100 11110101
+
-4875 = 11101100 11110101
=
-9750 = 11011001 11101010

```

Есть переносы из 14 и 15-го разрядов. Результат правильный.

Таким образом, мы исследовали все случаи и выяснили, что ситуация переполнения (установка флага of в 1) происходит при переносе:

- из 14-го разряда (для положительных чисел со знаком);
- из 15-го разряда (для отрицательных чисел).

И наоборот, переполнения не происходит (то есть флаг of сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах.

Итак, переполнение регистрируется с помощью флага переполнения of. Дополнительно к флагу of при переносе из старшего разряда устанавливается в 1 и флаг переноса cf. Так как микропроцессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Проанализировать флаги cf и of можно командами условного перехода `jc\jnc` и `jo\jno` соответственно.

Что же касается команд сложения чисел со знаком, то они те же, что и для чисел без знака.

Вычитание двоичных чисел без знака

Как и при анализе операции сложения, порассуждаем над сутью процессов, происходящих при выполнении операции вычитания. Если уменьшаемое больше вычитаемого, то проблем нет, — разность положительна, результат верен. Если уменьшаемое меньше вычитаемого, возникает проблема: результат меньше 0, а это уже число со знаком. В этом случае результат необходимо завернуть. Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Микропроцессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим, в разрядной сетке операнда. Поясним на примере.

Пример 5.

```

05 = 00000000 00000101
-10 = 00000000 00001010
чтобы произвести вычитание, произведем воображаемый заем из старшего разряда:
100000000 00000101
-
00000000 00001010
=
11111111 11111011

```

Тем самым по сути выполняется действие

(65 536 + 5) — 10 = 65 531,

0 здесь как бы эквивалентен числу 65 536. Результат, конечно, неверен, но микропроцессор считает, что все нормально, хотя факт заема единицы он фиксирует установкой флага переноса cf. Но посмотрите еще раз внимательно на результат операции вычитания. Это же –5 в дополнительном коде! Проведем эксперимент: представим разность в виде суммы $5 + (-10)$.

Пример 6.

```

5      = 00000000 00000101
+
(-10) = 11111111 11110110
=
      11111111 11110111

```

то есть мы получили тот же результат, что и в предыдущем примере.

Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага cf. Если он установлен в 1, то это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

Аналогично командам сложения, группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматриваем, а учет особых ситуаций должен производиться самим программистом. К командам вычитания относятся следующие:

- dec операнд — операция декремента, то есть уменьшения значения операнда на 1;
- sub операнд_1, операнд_2 — команда вычитания; ее принцип действия:
операнд_1 = операнд_1 – операнд_2
- sbb операнд_1, операнд_2 — команда вычитания с учетом заема (флага cf):
операнд_1 = операнд_1 – операнд_2 – значение_cf

Как видите, среди команд вычитания есть команда sbb, учитывающая флаг переноса cf. Эта команда подобна adc, но теперь уже флаг cf выполняет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Рассмотрим пример (листинг 4) программной обработки ситуации, разобранный в примере 6.

Листинг 4. Проверка при вычитании чисел без знака

```

<1> ;prg_8_4.asm
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> .code ;сегмент кода
<7> main: ;точка входа в программу
<8> ...
<9> xor ax,ax
<10> mov al,5
<11> sub al,10
<12> jnc ml ;нет переноса?
<13> neg al ;в al модуль результата
<14> ml: ...
<15> exit:
<16> mov ax,4c00h ;стандартный выход
<17> int 21h
<18> end main ;конец программы

```

В этом примере в строке 11 выполняется вычитание. С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальному виду (получить его модуль), применяется команда neg, с помощью которой получается дополнение операнда. В нашем случае мы получили дополнение дополнения или модуль отрицательного результата. А тот факт, что это на самом деле число отрицательное, отражен в состоянии флага cf. Дальше все зависит от алгоритма обработки. Исследуйте программу в отладчике.

Вычитание двоичных чисел со знаком

Здесь все несколько сложнее. Последний пример (листинг 4) показал то, что микропроцессору незачем иметь два устройства — сложения и вычитания. Достаточно наличия только одного — устройства сложения. Но для вычитания способом сложения чисел со знаком в дополнительном коде необходимо представлять оба операнда — и уменьшаемое, и вычитаемое. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего они связаны с тем, что старший бит операнда рассматривается как знаковый. Рассмотрим пример вычитания $45 - (-127)$.

Пример 7.

Вычитание чисел со знаком 1

```
45 = 0010 1101
-
-127 = 1000 0001
=
-44 = 1010 1100
```

Судя по знаковому разряду, результат получился отрицательный, что, в свою очередь, говорит о том, что число нужно рассматривать как дополнение, равное -44 . Правильный результат должен быть равен 172. Здесь мы, как и в случае знакового сложения, встретились с переполнением мантиссы, когда значащий разряд числа изменил знаковый разряд операнда. Отследить такую ситуацию можно по содержимому флага переполнения of. Его установка в 1 говорит о том, что результат вышел за диапазон представления знаковых чисел (то есть изменился старший бит) для операнда данного размера, и программист должен предусмотреть действия по корректировке результата.

Другой пример разности рассматривается в примере 7, но выполним мы ее способом сложения.

Пример 8.

Вычитание чисел со знаком 2

```
-45 - 45 = -45 + (-45) = -90.
-45 = 1101 0011
+
-45 = 1101 0011
=
-90 = 1010 0110
```

Здесь все нормально, флаг переполнения of сброшен в 0, а 1 в знаковом разряде говорит о том, что значение результата — число в дополнительном коде.

В завершение обсуждения команд сложения и вычитания отметим, что кроме флагов cf и of в регистре eFlags есть еще несколько флагов, которые можно использовать с двоичными арифметическими командами. Речь идет о следующих флагах:

- zf — флаг нуля, который устанавливается в 1, если результат операции равен 0, и в 0, если результат не равен 0;
- sf — флаг знака, значение которого после арифметических операций (и не только) совпадает со значением старшего бита результата, то есть с битом 7, 15 или 31. Таким образом, этот флаг можно использовать для операций над числами со знаком.

Умножение чисел без знака

Для умножения чисел без знака предназначена команда

mul сомножитель_1

Как видите, в команде указан всего лишь один операнд-сомножитель. Вторым операндом — сомножителем_2 задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и размещения второго операнда и результата приведены в табл. 2.

Таблица 2. Расположение операндов и результата при умножении

сумножитель_1	сумножитель_2	Результат
Байт	al	16 бит в ax: al — младшая часть результата; ah — старшая часть результата
Слово	ax	32 бит в паре dx:ax: ax — младшая часть результата; dx — старшая часть результата
Двойное слово	eax	64 бит в паре edx:eax: eax — младшая часть результата; edx — старшая часть результата

Из таблицы видно, что произведение состоит из двух частей и в зависимости от размера операндов размещается в двух местах — на месте сумножитель_2 (младшая часть) и в дополнительном регистре ah, dx, edx (старшая часть). Как же динамически (то есть во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре или что он превысил размерность регистра и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам по предыдущему обсуждению флаги переноса cf и переполнения of:

- если старшая часть результата нулевая, то после операции произведения флаги cf = 0 и of = 0;
- если же эти флаги ненулевые, то это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Рассмотрим следующий пример программы.

```

Листинг 5. Умножение
<1> ;prg_8_5.asm
<2> masm
<3> model small
<4> stack 256
<5> .data ;сегмент данных
<6> rez label word
<7> rez_l db 45
<8> rez_h db 0
<9> .code ;сегмент кода
<10> main: ;точка входа в программу
<11> ...
<12> xor ax,ax
<13> mov al,25
<14> mul rez_l
<15> jnc ml ;если переполнение, то на ml
<16> mov rez_h,ah ;старшую часть результата в rez_h
<17> ml:
<18> mov rez_l,al
<19> exit:
<20> mov ax,4c00h ;стандартный выход
<21> int 21h
<22> end main ;конец программы

```

В этой программе в строке 14 производится умножение значения в rez_l на число в регистре al. Согласно информации в табл. 2, результат умножения будет располагаться в регистре al (младшая часть) и регистре ah (старшая часть). Для выяснения размера результата в строке 15 командой условного перехода jnc анализируется состояние флага cf и если оно не равно 1, то результат остался в рамках регистра al. Если же cf = 1, то выполняется команда в строке 16, которая формирует в поле rez_h старшее слово результата. Команда в строке 18 формирует младшую часть результата.

Теперь обратите внимание на сегмент данных, а именно, на строку 6. В этой строке содержится директива `label`. Мы еще не раз будем сталкиваться с этой директивой. В данном случае она назначает еще одно символическое имя `rez` адресу, на который уже указывает другой идентификатор `rez_1`. Отличие заключается в типах этих идентификаторов — имя `rez` имеет тип слова, который ему назначается директивой `label` (имя типа указано в качестве операнда `label`). Введя эту директиву в программу, мы подготовились к тому, что, возможно, результат операции умножения будет занимать слово в памяти. Обратите внимание, что мы не нарушили принципа: младший байт по младшему адресу. Далее, используя имя `rez`, можно обращаться к значению в этой области как к слову.

Умножение чисел со знаком

Для умножения чисел со знаком предназначена команда

`imul операнд_1[,операнд_2,операнд_3]`

Эта команда выполняется так же, как и команда `mul`. Отличительной особенностью команды `imul` является только формирование знака.

Если результат мал и умещается в одном регистре (то есть если $cf = of = 0$), то содержимое другого регистра (старшей части) является расширением знака — все его биты равны старшему биту (знаковому разряду) младшей части результата.

В противном случае (если $cf = of = 1$) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата.

Если вы посмотрите описание команды [`imul`](#), то увидите, что она допускает более широкие возможности по заданию местоположения операндов. Это сделано для удобства использования.

Деление чисел без знака

Для деления чисел без знака предназначена команда

`div делитель`

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бит. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера операндов. Результатом команды деления являются значения частного и остатка.

Варианты местоположения и размеров операндов операции деления показаны в табл. 3.

Таблица 3. Расположение операндов и результата при делении

Делимое	Делитель	Частное	Остаток
16 бит в регистре <code>ax</code>	Байт регистр или ячейка памяти	Байт в регистре <code>al</code>	Байт в регистре <code>ah</code>
32 бит <code>dx</code> — старшая часть <code>ax</code> — младшая часть	Слово 16 бит регистр или ячейка памяти	Слово 16 бит в регистре <code>ax</code>	Слово 16 бит в регистре <code>dx</code>
64 бит <code>edx</code> — старшая часть <code>eax</code> — младшая часть	Двойное слово 32 бит регистр или ячейка памяти	Двойное слово 32 бит в регистре <code>eax</code>	Двойное слово 32 бит в регистре <code>edx</code>

После выполнения команды деления содержимое флагов неопределенно, но возможно возникновение прерывания с номером 0, называемого “деление на ноль”. Этот вид прерывания относится к так называемым исключениям. Эта разновидность прерываний возникает внутри микропроцессора из-за некоторых аномалий во время вычислительного процесса.

Прерывание 0, “деление на ноль”, при выполнении команды `div` может возникнуть по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную под него разрядную сетку, что может случиться в следующих случаях:
 - при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого в более чем 256 раз больше значения делителя;
 - при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого в более чем 65 536 раз больше значения делителя;
 - при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого в более чем 4 294 967 296 раз больше значения делителя.

К примеру, выполним деление значения в области `del` на значение в области `delt` (листинг 6).

```
Листинг 6. Деление чисел
<1> ;prg_8.6.asm
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> del_b label byte
<7> deldw 29876
<8> delt db 45
<9> .code ;сегмент кода
<10> main: ;точка входа в программу
<11> ...
<12> xor ax,ax
<13> ;последующие две команды можно заменить одной mov
    ax,del
<14> mov ah,del_b ;старший байт делимого в ah
<15> mov al,del_b+1 ;младший байт делимого в al
<16> div delt ;в al — частное, в ah — остаток
<17> ...
<18> endmain ;конец программы
```

Деление чисел со знаком

Для деления чисел со знаком предназначена команда

`idiv` делитель

Для этой команды справедливы все рассмотренные положения, касающиеся команд и чисел со знаком.

Отметим лишь особенности возникновения исключения 0, “деление на ноль”, в случае чисел со знаком. Оно возникает при выполнении команды `idiv` по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную для него разрядную сетку.

Последнее в свою очередь может произойти:

- при делении делимого величиной в слово со знаком на делитель величиной в байт со знаком, причем значение делимого в более чем 128 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от -128 до +127);
- при делении делимого величиной в двойное слово со знаком на делитель величиной в слово со знаком, причем значение делимого в более чем 32 768 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от -32 768 до +32 768);
- при делении делимого величиной в учетверенное слово со знаком на делитель величиной в двойное слово со знаком, причем значение делимого в более чем 2 147 483 648 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от -2 147 483 648 до +2 147 483 647).

ВСПОМОГАТЕЛЬНЫЕ КОМАНДЫ ДЛЯ ЦЕЛОЧИСЛЕННЫХ ОПЕРАЦИЙ

В системе команд микропроцессора есть несколько команд, которые могут облегчить программирование алгоритмов, производящих арифметические вычисления. В них могут возникать различные проблемы, для разрешения которых разработчики микропроцессора предусмотрели несколько команд. Рассмотрим их в следующем разделе.

Команды преобразования типов

Что делать, если размеры операндов, участвующих в арифметических операциях, разные? Например, предположим, что в операции сложения один операнд является словом, а другой занимает двойное слово. Выше сказано, что в операции сложения должны участвовать операнды одного формата. Если числа без знака, то выход найти просто. В этом случае можно на базе исходного операнда сформировать новый (формата двойного слова), старшие разряды которого просто заполнить нулями. Сложнее ситуация для чисел со знаком: как динамически, в ходе выполнения программы, учесть знак операнда? Для решения подобных проблем в системе команд микропроцессора есть так называемые команды преобразования типа. Эти команды расширяют байты в слова, слова — в двойные слова и двойные слова — в учетверенные слова (64-разрядные значения). Команды преобразования типа особенно полезны при преобразовании целых со знаком, так как они автоматически заполняют старшие биты вновь формируемого операнда значениями знакового бита старого объекта. Эта операция приводит к целым значениям того же знака и той же величины, что и исходная, но уже в более длинном формате. Подобное преобразование называется *операцией распространения знака*.

Существуют два вида команд преобразования типа:

- Команды без операндов — эти команды работают с фиксированными регистрами:
 - `cbw` (Convert Byte to Word) — команда преобразования байта (в регистре `al`) в слово (в регистре `ax`) путем распространения значения старшего бита `al` на все биты регистра `ah`;
 - `cwd` (Convert Word to Double) — команда преобразования слова (в регистре `ax`) в двойное слово (в регистрах `dx:ax`) путем распространения значения старшего бита `ax` на все биты регистра `dx`;
 - `cwde` (Convert Word to Double) — команда преобразования слова (в регистре `ax`) в двойное слово (в регистре `eax`) путем распространения значения старшего бита `ax` на все биты старшей половины регистра `eax`;
 - `cdq` (Convert Double Word to Quarter Word) — команда преобразования двойного слова (в регистре `eax`) в учетверенное слово (в регистрах `edx:eax`) путем распространения значения старшего бита `eax` на все биты регистра `edx`.
- Команды `movsx` и `movzx`, относящиеся к командам обработки строк (см. урок 11). Эти команды обладают полезным свойством в контексте нашей проблемы:
 - `movsx` операнд_1, операнд_2 — переслать с распространением знака. Расширяет 8 или 16-разрядное значение операнд_2, которое может быть регистром или операндом в памяти, до 16 или 32-разрядного значения в одном из регистров, используя значение знакового бита для заполнения старших позиций операнд_1. Данную команду удобно использовать для подготовки операндов со знаками к выполнению арифметических действий;
 - `movzx` операнд_1, операнд_2 — переслать с расширением нулем. Расширяет 8 или 16-разрядное значение операнд_2 до 16 или 32-разрядного с очисткой (заполнением) нулями старших позиций операнд_2. Данную команду удобно использовать для подготовки операндов без знака к выполнению арифметических действий.

К примеру, вычислим значение $y = (a + b)/c$, где a , b , c — байтовые знаковые переменные (листинг 7).

```
Листинг 7. Вычисление простого выражения
<1> ;prg_8_9.asm
<2> masm
<3> model small
<4> stack 256
<5> .data
<6> a db ?
<7> b db ?
<8> c db ?
<9> y dw 0
<10> .code
<11> main: ;точка входа в программу
<12> ...
<13> xor ax,ax
<14> mov al,a
<15> cbw
<16> movsx bx,b
<17> add ax,bx
<18> idiv c ;в al — частное, в ah — остаток
<19> exit:
<20> mov ax,4c00h ;стандартный выход
<21> int 21h
<22> end main ;конец программы
```

В этой программе делимое для команды `idiv` (строка 17) готовится заранее. Так как делитель имеет размер байта, то делимое должно быть словом. С учетом этого сложение осуществляется параллельно с преобразованием размера результата в слово (строки 13–16). Для примера расширение операндов со знаком производится двумя разными командами — `cbw` и `movsx`.

Другие полезные команды

xadd назначение,источник — обмен местами и сложение.

Команда позволяет выполнить последовательно два действия:

- обменять значения назначение и источник;
- поместить на место операнда назначение сумму:
назначение = назначение + источник.

neg операнд — отрицание с дополнением до двух.

Команда выполняет инвертирование значения операнд. Физически команда выполняет одно действие: операнд = 0 – операнд, то есть вычитает операнд из нуля.

Команду `neg операнд` можно применять:

- для смены знака;
- для выполнения вычитания из константы.

Дело в том, что команды `sub` и `sbb` не позволяют вычесть что-либо из константы, так как константа не может служить операндом-приемником в этих операциях. Поэтому данную операцию можно выполнить с помощью двух команд:

```
neg ax ;смена знака (ax)
...
add ax,340 ;фактически вычитание: (ax)=340-(ax)
```

Арифметические операции над двоично-десятичными числами

В данном разделе мы рассмотрим особенности каждого из четырех основных арифметических действий для упакованных и неупакованных двоично-десятичных чисел:

- [сложение неупакованных BCD-чисел](#);
- [вычитание неупакованных BCD-чисел](#);
- [умножение неупакованных BCD-чисел](#);
- [деление неупакованных BCD-чисел](#);
- [сложение упакованных BCD-чисел](#);
- [вычитание упакованных BCD-чисел](#).

У вас справедливо может возникнуть вопрос: а зачем нужны BCD-числа? Ответ может быть следующим: BCD-числа нужны в деловых приложениях, то есть там, где числа должны быть большими и точными. Как мы уже убедились на примере двоичных чисел, операции с такими числами довольно проблематичны для языка ассемблера. К недостаткам использования двоичных чисел можно отнести следующие:

- значения величин в формате слова и двойного слова имеют ограниченный диапазон. Если программа предназначена для работы в области финансов, то ограничение суммы в рублях величиной 65 536 (для слова) или даже 4 294 967 296 (для двойного слова) будет существенно сужать сферу ее применения (да еще в наших экономических условиях — тут уж никакая деноминация не поможет);
- наличие ошибок округления. Представьте себе программу, работающую где-нибудь в банке, которая не учитывает величину остатка при действиях с целыми двоичными числами и оперирует при этом миллиардами? Не хотелось бы быть автором такой программы. Применение чисел с плавающей точкой не спасет — там существует та же проблема округления;
- представление большого объема результатов в символьном виде (ASCII-коде). Деловые программы не просто выполняют вычисления; одной из целей их использования является оперативная выдача информации пользователю. Для этого, естественно, информация должна быть представлена в символьном виде. Перевод чисел из двоичного кода в ASCII-код, как мы уже видели, требует определенных вычислительных затрат. Число с плавающей точкой еще труднее перевести в символьный вид. А вот если посмотреть на шестнадцатеричное представление неупакованной десятичной цифры (в начале нашего занятия) и на соответствующий ей символ в таблице ASCII, то видно что они отличаются на величину 30h. Таким образом, преобразование в символьный вид и обратно получается намного проще и быстрее.

Наверняка, вы уже убедились в важности овладения хотя бы основами действий с десятичными числами. Далее рассмотрим особенности выполнения основных арифметических операций с десятичными числами. Для предупреждения возможных вопросов отметим сразу тот факт, что отдельных команд сложения, вычитания, умножения и деления BCD-чисел нет. Сделано это по вполне понятным причинам: размерность таких чисел может быть сколь угодно большой. Складывать и вычитать можно двоично-десятичные числа как в упакованном формате, так и в неупакованном, а вот делить и умножать можно только неупакованные BCD-числа. Почему это так, будет видно из дальнейшего обсуждения.

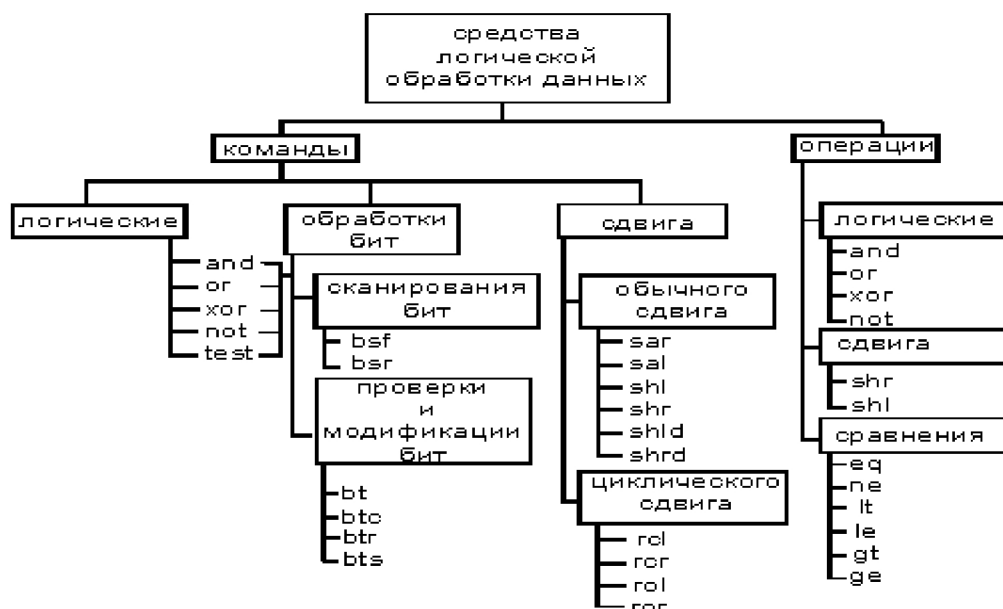
○ **Логические команды.**

ЛОГИЧЕСКИЕ КОМАНДЫ

Наряду со средствами арифметических вычислений, система команд микропроцессора имеет также средства логического преобразования данных. Под логическими понимаются такие преобразования данных, в основе которых лежат *правила формальной логики*.

Формальная логика работает на уровне утверждений **истинно** и **ложно**. Для микропроцессора это, как правило, означает **1** и **0** соответственно.

Для компьютера язык нулей и единиц является родным, но минимальной единицей данных, с которой работают машинные команды, является *байт*. Однако, на системном уровне часто необходимо иметь возможность работать на предельно низком уровне — на уровне *бит*.



К средствам логического преобразования данных относятся логические команды и [логические операции](#). В связи с последним напомним, что операнд команды ассемблера в общем случае может представлять собой выражение, которое, в свою очередь, является комбинацией операторов и операндов. Среди этих операторов могут быть и операторы, реализующие логические операции над объектами выражения.

Перед подробным рассмотрением этих средств давайте посмотрим, что же представляют собой сами *логические данные* и какие операции над ними производятся.

Логические данные

Теоретической базой для логической обработки данных является *формальная логика*.

Существует несколько систем логики. Одна из наиболее известных — это *исчисление высказываний*.

Высказывание — это любое утверждение, о котором можно сказать, что оно либо *истинно*, либо *ложно*. Исчисление высказываний представляет собой совокупность правил, используемых для определения истинности или ложности некоторой комбинации высказываний.

Исчисление высказываний очень гармонично сочетается с принципами работы компьютера и основными методами его программирования. Все аппаратные компоненты компьютера построены на логических микросхемах. Система представления информации в компьютере на самом нижнем уровне основана на понятии бита. Бит, имея всего два состояния: 0 (ложно) и 1 (истинно), — естественным образом вписывается в исчисление высказываний.

Согласно теории, над высказываниями (над битами) могут выполняться следующие логические операции:

- [отрицание \(логическое НЕ\)](#);
- [логическое сложение \(логическое включающее ИЛИ\)](#);
- [логическое умножение \(логическое И\)](#);
- [логическое исключающее сложение \(логическое исключающее ИЛИ\)](#).

- **отрицание** (логическое *НЕ*) — логическая операция над одним операндом, результатом которой является величина, обратная значению исходного операнда. Эта операция однозначно характеризуется следующей таблицей истинности (табл. 1).

Значение операнда	0	1
Результат операции	1	0

- **логическое сложение** (логическое включающее *ИЛИ*) — логическая операция над двумя операндами, результатом которой является “истина” (1), если один или оба операнда имеют значение “истина” (1), и “ложь” (0) — если оба операнда имеют значение “ложь” (0). Эта операция описывается с помощью следующей таблицы истинности (табл. 2).

Значение операнда 1	0	0	1	1
Значение операнда 2	0	1	0	1
Результат операции	0	1	1	1

- **логическое умножение** (логическое *И*) — логическая операция над двумя операндами, результатом которой является “истина” (1) только в том случае, если оба операнда имеют значение “истина” (1). Во всех остальных случаях значение операции “ложь” (0). Эта операция описывается с помощью следующей таблицы истинности (табл.3).
- **логическое исключающее сложение** (логическое исключающее *ИЛИ*) — логическая операция над двумя операндами, результатом которой является “истина” (1), если только один из двух операндов имеет значение “истина” (1), и ложь (0), если оба операнда имеют значение “ложь” (0) или “истина” (1). Эта операция описывается с помощью следующей таблицы истинности(табл. 4).

Значение операнда 1	0	0	1	1
Значение операнда 2	0	1	0	1
Результат операции	0	0	0	1

Значение операнда 1	0	0	1	1
Значение операнда 2	0	1	0	1
Результат операции	0	1	1	0

Система команд микропроцессора содержит пять команд, поддерживающих данные операции. Эти команды выполняют логические операции над битами операндов. Размерность операндов, естественно, должна быть одинакова. Например, если размерность операндов равна слову (16 бит), то логическая операция выполняется сначала над нулевыми битами операндов и ее результат записывается на место бита 0 результата. Далее команда последовательно повторяет эти действия над всеми битами с первого до пятнадцатого. Возможные варианты размерности операндов для каждой команды приведены в “Справочнике команд”.

Логические команды

В системе команд микропроцессора есть следующий набор команд, поддерживающих работу с логическими данными:

and операнд_1,операнд_2 — операция логического умножения.

Команда выполняет поразрядно логическую операцию И (конъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

or операнд_1,операнд_2 — операция логического сложения.

Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

xor операнд_1,операнд_2 — операция логического исключающего сложения.

Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов операнд_1 и операнд_2. Результат записывается на место операнд_1.

test операнд_1,операнд_2 — операция “проверить” (способом логического умножения).

Команда выполняет поразрядно логическую операцию И над битами операндов операнд_1 и операнд_2. Состояние операндов остается прежним, изменяются только флаги zf, sf, и pf, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

not операнд — операция логического отрицания.

Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

Для представления роли логических команд в системе команд микропроцессора очень важно понять области их применения и типовые приемы их использования при программировании.

С помощью логических команд возможно *выделение* отдельных битов в операнде с целью их *установки, сброса, инвертирования или просто проверки на определенное значение*.

Для организации подобной работы с битами *операнд_2* обычно играет роль **маски**. С помощью установленных в 1 битов этой маски и определяют нужные для конкретной операции биты операнд_1. Покажем, какие логические команды могут применяться для этой цели:

- Для установки определенных разрядов (бит) в 1 применяется команда **or операнд_1,операнд_2**.
В этой команде операнд_2, выполняющий роль маски, должен содержать единичные биты на месте тех разрядов, которые должны быть установлены в 1 в операнд_1.

```
or    eax,10b ;установить 1-й бит в регистре eax
```

- Для сброса определенных разрядов (бит) в 0 применяется команда **and операнд_1,операнд_2**.
В этой команде операнд_2, выполняющий роль маски, должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в операнд_1.

```
and    eax,ffffffdh ;сбросить в 0 1-й бит в регистре eax
```

- Команда **xor операнд_1,операнд_2** применяется:
 - для выяснения того, какие биты в операнд_1 и операнд_2 различаются;
 - для инвертирования состояния заданных бит в операнд_1.

```
xor    eax,10b ;инвертировать 1-й бит в регистре eax
jz     mes     ;переход, если 1-й бит в al был единичным
```

- Интересующие нас биты маски (операнд_2) при выполнении команды xor должны быть единичными, остальные — нулевыми.
- Для проверки состояния заданных бит применяется команда **test операнд_1,операнд_2** (проверить операнд_1).
Проверяемые биты операнд_1 в маске (операнд_2) должны иметь единичное значение. Алгоритм работы команды test подобен алгоритму команды and, но он не меняет значения операнд_1.
Результатом команды является установка значения флага нуля **zf**:
 - если $zf = 0$, то в результате логического умножения получился нулевой результат, то есть один единичный бит маски, который **не совпал** с соответствующим единичным битом операнд_1;
 - если $zf = 1$, то в результате логического умножения получился ненулевой результат, то есть *хотя бы один* единичный бит маски **совпал** с соответствующим единичным битом операнд_1.

```
test    eax,00000010h
jz      ml      ;переход, если 4-й бит равен 1
```

Как видно из примера, для реакции на результат команды test целесообразно использовать команду перехода [jnz](#) метка (Jump if Not Zero) — переход, если флаг нуля **zf** ненулевой, или команду с обратным действием — [jz](#) метка (Jump if Zero) — переход, если флаг нуля **zf** = 0.

Следующие две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала так и от конца операнда:

bsf операнд_1,операнд_2 (Bit Scanning Forward) - сканирование битов вперед.

Команда просматривает (сканирует) биты операнд_2 от младшего к старшему (от бита 0 до старшего бита) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд_1 заносится номер этого бита в виде целочисленного значения. Если все биты операнд_2 равны 0, то флаг нуля zf устанавливается в 1, в противном случае флаг zf сбрасывается в 0.

```
mov     al,02h
bsf     bx,al    ;bx=1
jz      ml      ;переход, если al=00h
...
```

bsr операнд_1,операнд_2 (Bit Scanning Reset) — сканирование битов в обратном порядке.

Команда просматривает (сканирует) биты операнд_2 от старшего к младшему (от старшего бита к биту 0) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд_1 заносится номер этого бита в виде целочисленного значения.

При этом важно, что позиция первого единичного бита слева отсчитывается все равно относительно бита 0. Если все биты операнд_2 равны 0, то флаг нуля zf устанавливается в 1, в противном случае флаг zf сбрасывается в 0.

Листинг 1 демонстрирует пример применения команд bsr и bsf. Введите код и исследуйте работу программы в отладчике (в частности, обратите внимание на то, как меняется содержимое регистра bx после команд bsf и bsr).

```
Листинг 1 Сканирование битов
;prg_9_1.asm
masm
model    small
stack    256
.data    ;сегмент данных
.code    ;сегмент кода
main:    ;точка входа в программу
mov      ax,@data
mov      ds,ax

;...
.486     ;это обязательно
xor      ax,ax
mov      al,02h
bsf      bx,ax    ;bx=1
jz      ml      ;переход, если al=00h
bsr      bx,ax

ml:
;...
mov      ax,4c00h    ;стандартный выход
int      21h
end      main
```

В последних моделях микропроцессоров Intel в группе логических команд появилось еще несколько команд, которые позволяют осуществить доступ к одному конкретному биту операнда. Операнд может находиться как в памяти, так и в регистре общего назначения. Положение бита задается смещением бита относительно младшего бита операнда. Значение смещения может задаваться как в виде непосредственного значения, так и содержаться в регистре общего назначения. В качестве значения смещения вы можете использовать результаты работы команд `bsr` и `bsf`. Все команды присваивают значение выбранного бита флагу `cf`.

bt операнд,смещение_бита (Bit Test) — проверка бита.

Команда переносит значение бита в флаг `cf`.

```
bt      ax,5      ;проверить значение бита 5
jnc     ml        ;переход, если бит = 0
```

bts операнд,смещение_бита (Bit Test and Set) — проверка и установка бита.

Команда переносит значение бита в флаг `cf` и затем устанавливает проверяемый бит в 1.

```
mov ax,10
bts     pole,ax ;проверить и установить 10-й бит в pole
jcm1    ;переход, если проверяемый бит был равен 1
```

btr операнд,смещение_бита (Bit Test and Reset) — проверка и сброс бита.

Команда переносит значение бита в флаг `cf` и затем устанавливает этот бит в 0.

btc операнд,смещение_бита (Bit Test and Convert) — проверка и инвертирование бита.

Команда переносит значение бита в флаг `cf` и затем инвертирует значение этого бита.

КОМАНДЫ СДВИГА

Команды этой группы также обеспечивают манипуляции над отдельными битами операндов, но иным способом, чем логические команды, рассмотренные выше.

Все команды сдвига перемещают биты в поле операнда влево или вправо в зависимости от кода операции.

Все команды сдвига имеют одинаковую структуру:

коп операнд,счетчик_сдвигов

Количество сдвигаемых разрядов — *счетчик_сдвигов* — располагается, как видите, на месте второго операнда и может задаваться двумя способами:

- *статически*, что предполагает задание фиксированного значения с помощью непосредственного операнда;
- *динамически*, что означает занесение значения счетчика сдвигов в регистр `cl` перед выполнением команды сдвига.

Исходя из размерности регистра `cl`, понятно, что значение счетчика сдвигов может лежать в диапазоне от 0 до 255. Но на самом деле это не совсем так.

В целях оптимизации микропроцессор воспринимает только значение *пяти младших битов* счетчика, то есть значение лежит в диапазоне от 0 до 31.

В последних моделях микропроцессора, в том числе и в микропроцессоре Pentium, есть дополнительные команды, позволяющие делать 64-разрядные сдвиги. Мы их рассмотрим чуть позже.

Все команды сдвига устанавливают флаг переноса `cf`.

По мере сдвига битов за пределы операнда они сначала попадают на флаг переноса, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

- [команды линейного сдвига](#);
- [команды циклического сдвига](#).

Команды линейного сдвига

К командам этого типа относятся команды, осуществляющие сдвиг по следующему алгоритму:

- очередной “выдвигаемый” бит устанавливает флаг cf;
- бит, вводимый в операнд с другого конца, имеет значение 0;
- при сдвиге очередного бита он переходит во флаг cf, при этом значение предыдущего сдвинутого бита *теряется!*

Команды линейного сдвига делятся на два подтипа:

- команды логического линейного сдвига;
- команды арифметического линейного сдвига.

К командам *логического* линейного сдвига относятся следующие:

shl операнд,счетчик_сдвигов (Shift Logical Left) - логический сдвиг влево.

Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик_сдвигов. Справа (в позицию младшего бита) вписываются нули;

shr операнд,счетчик_сдвигов (Shift Logical Right) — логический сдвиг вправо.

Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик_сдвигов. Слева (в позицию старшего, знакового бита) вписываются нули. На рис. 1 показан принцип работы этих команд.

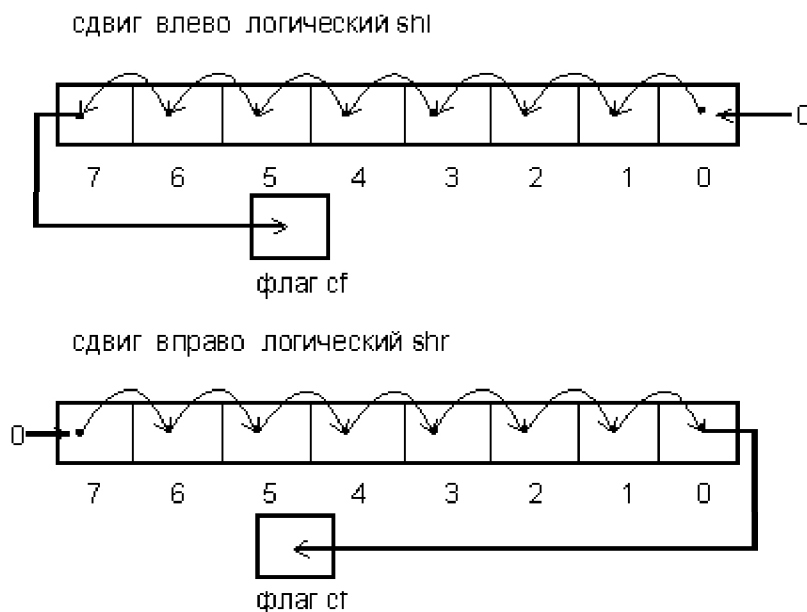


Рис. 1. Схема работы команд линейного логического сдвига

Ниже показан фрагмент программы, который выполняет преобразование двух неупакованных BCD-чисел в слове памяти bcd_dig в упакованное BCD-число в регистре al.

```
...  
bcd_dig dw      0905h    ;описание неупакованного BCD-числа 95  
...  
    mov     ax,bcd_dig    ;пересылка  
    shl     ah,4          ;сдвиг влево  
    add     al,ah         ;сложение для получения результата: al=95h
```

Команды *арифметического* линейного сдвига отличаются от команд *логического* сдвига тем, что они особым образом работают со *знаковым разрядом* операнда.

sal операнд, счетчик_сдвигов (Shift Arithmetic Left) — арифметический сдвиг влево.

Содержимое операнда сдвигается влево на количество битов, определяемое значением *счетчик_сдвигов*.

Справа (в позицию младшего бита) вписываются нули. Команда **sal** **не сохраняет знака**, но *устанавливает флаг cf в случае смены знака* очередным выдвигаемым битом. В остальном команда **sal** полностью аналогична команде **shl**;

sar операнд, счетчик_сдвигов (Shift Arithmetic Right) — арифметический сдвиг вправо.

Содержимое операнда сдвигается вправо на количество битов, определяемое значением *счетчик_сдвигов*.

Слева в операнд вписываются нули. Команда **sar** **сохраняет знак**, восстанавливая его после сдвига каждого очередного бита.

На рис. 2 показан принцип работы команд линейного арифметического сдвига.

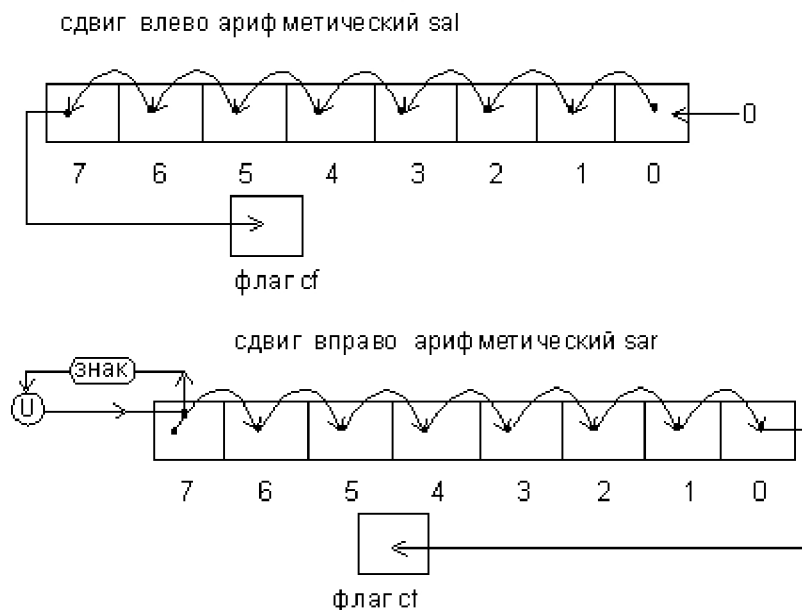


Рис. 2. Схема работы команд линейного арифметического сдвига

Команды циклического сдвига

К командам *циклического* сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига:

- команды простого циклического сдвига;
- команды циклического сдвига через флаг переноса cf.

К командам *простого циклического* сдвига относятся:

rol операнд, счетчик_сдвигов (Rotate Left) — циклический сдвиг влево.

Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов*.

Сдвигаемые влево биты записываются в тот же операнд справа.

ror операнд, счетчик_сдвигов (Rotate Right) — циклический сдвиг вправо.

Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов*.

Сдвигаемые вправо биты записываются в тот же операнд слева.

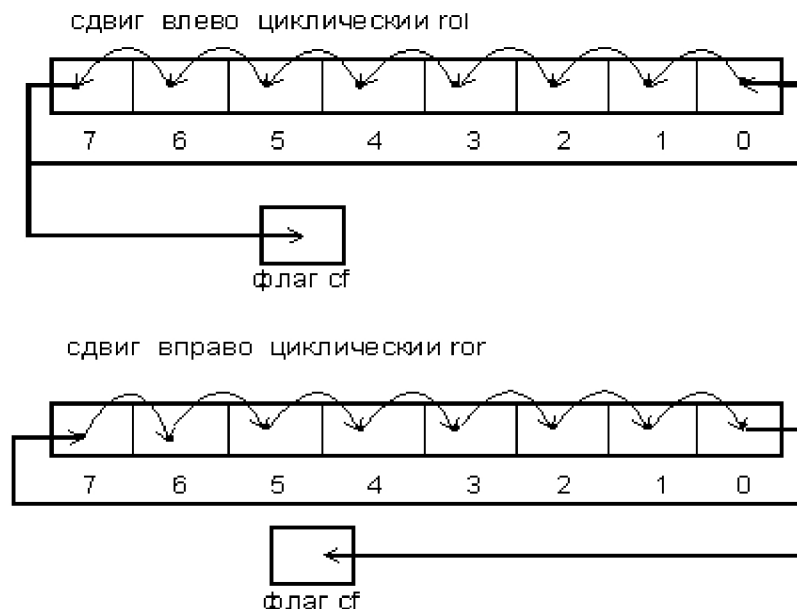


Рис. 3. Схема работы команд простого циклического сдвига

Как видно из рис. 3, команды простого циклического сдвига в процессе своей работы осуществляют одно полезное действие, а именно: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становится значением флага cf.

К примеру, для того чтобы обменять содержимое двух половинок регистра eax, достаточно выполнить следующую последовательность команд:

```
...
mov     eax,ffff0000h
mov     cl,16
rol     eax,cl
```

Команды циклического сдвига *через флаг переноса cf* отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а записывается сначала в флаг переноса cf. Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита с другого конца операнда (см. рис. 4).

К командам циклического сдвига *через флаг переноса cf* относятся следующие:

rol операнд,счетчик_сдвигов (Rotate through Carry Left) — циклический сдвиг влево через перенос. Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые биты поочередно становятся значением флага переноса cf.

ror операнд,счетчик_сдвигов (Rotate through Carry Right) — циклический сдвиг вправо через перенос. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик_сдвигов*. Сдвигаемые биты поочередно становятся значением флага переноса cf.

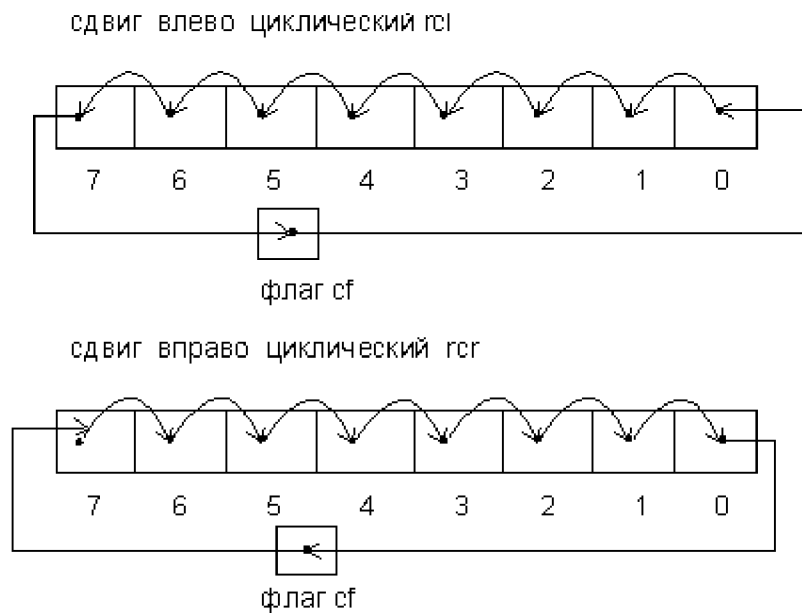


Рис. 4. Команды циклического сдвига через флаг переноса cf

Из рис. 4 видно, что при сдвиге через флаг переноса появляется промежуточный элемент, с помощью которого, в частности, можно производить подмену циклически сдвигаемых битов, в частности, *рассогласование* битовых последовательностей.

Под рассогласованием битовой последовательности здесь и далее подразумевается действие, которое позволяет некоторым образом локализовать и извлечь нужные участки этой последовательности и записать их в другое место.

Дополнительные команды сдвига

Система команд последних моделей микропроцессоров Intel, начиная с i80386, содержит дополнительные команды сдвига, расширяющие возможности, рассмотренные нами ранее.

Это — команды сдвигов *двойной точности*:

shld операнд_1, операнд_2, счетчик_сдвигов — сдвиг влево двойной точности.

Команда shld производит замену путем сдвига битов операнда операнд_1 влево, заполняя его биты справа значениями битов, вытесняемых из операнд_2 согласно схеме на рис. 5. Количество сдвигаемых бит определяется значением *счетчик_сдвигов*, которое может лежать в диапазоне **0...31**. Это значение может задаваться непосредственным операндом или содержаться в регистре cl. Значение операнд_2 не изменяется.



Рис. 5. Схема работы команды shld

shrd

shrd операнд_1, операнд_2, счетчик_сдвигов — сдвиг вправо двойной точности.

Команда производит замену путем сдвига битов операнда операнд_1 вправо, заполняя его биты слева значениями битов, вытесняемых из операнд_2 согласно схеме на рис. 6. Количество сдвигаемых бит определяется значением *счетчик_сдвигов*, которое может лежать в диапазоне **0...31**. Это значение может задаваться непосредственным операндом или содержаться в регистре cl. Значение операнд_2 не изменяется.

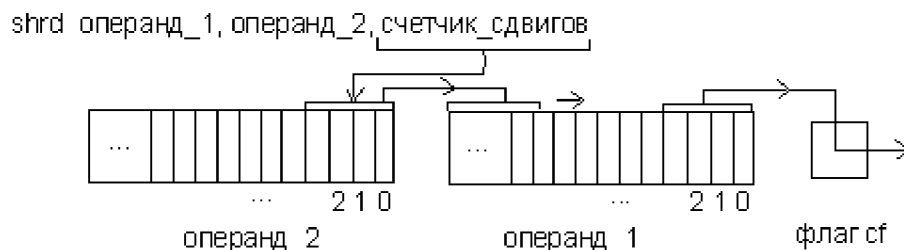


Рис. 6. Схема работы команды shrd

Как мы отметили, команды shld и shrd осуществляют сдвиги до 32 разрядов, но за счет особенностей задания операндов и алгоритма работы эти команды можно использовать для работы с полями длиной до 64 бит. Например, рассмотрим, как можно осуществить сдвиг влево на 16 бит поля из 64 бит.

```
;...
.data
pole_l dd 0b21187f5h1
pole_h dd 45ff6711h
.code
;...
.386

mov     cl,16    ;загрузка счетчика сдвига в cl
mov     eax,pole_h
shld    pole_l,eax,cl
shl     pole_h,cl    ;pole_l=87f50000h, pole_h=6711b211h
```


Лекция 5

Команды микропроцессора.

Команды передачи управления Безусловные переходы Условные переходы Команда сравнения стр Организация циклов Процедуры Использование подпрограмм Вызовы подпрограмм Система прерываний Аппаратная организация прерываний Обработка прерываний Программные прерывания.

Циклы и переходы.

КОМАНДЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ

На предыдущих уроках мы познакомились с некоторыми командами, из которых формируются *линейные* участки программы. Каждая из них в общем случае выполняет некоторые действия по преобразованию или пересылке данных, после чего микропроцессор передает управление следующей команде. Но очень мало программ работают таким последовательным образом. Обычно в программе есть точки, в которых нужно принять решение о том, какая команда будет выполняться следующей. Это решение может быть

- *безусловным* — в данной точке необходимо передать управление не той команде, которая идет следующей, а другой, которая находится на некотором удалении от текущей команды;
- *условным* — решение о том, какая команда будет выполняться следующей, принимается на основе анализа некоторых условий или данных.

Как вы помните, программа представляет собой последовательность команд и данных, занимающих определенное пространство оперативной памяти. Эта пространство памяти может быть либо непрерывным, либо состоять из нескольких фрагментов.

На уроке 5 нами были рассмотрены средства организации фрагментации кода программы и ее данных на сегменты. То, какая команда программы должна выполняться следующей, микропроцессор узнает по содержимому пары регистров **cs:(e)ip**:

- **cs** — сегментный регистр кода, в котором находится физический (базовый) адрес текущего сегмента кода;
- **еip/іp** — регистр указателя команды, в котором находится значение, представляющее собой смещение в памяти следующей команды, подлежащей выполнению, относительно начала текущего сегмента кода.

Напомним, почему мы записываем регистры **еip/іp** через косую черту. Какой конкретно регистр будет использоваться, зависит от установленного режима адресации **use16** или **use32**. Если указано **use16**, то используется **іp**, если **use32**, то используется **еip**.

Таким образом, команды передачи управления изменяют содержимое регистров **cs** и **еip/іp**, в результате чего микропроцессор выбирает для выполнения не следующую по порядку команду программы, а команду в некотором другом участке программы. Конвейер внутри микропроцессора при этом сбрасывается.

По принципу действия, команды микропроцессора, обеспечивающие организацию переходов в программе, можно разделить на четыре группы:

1. Команды безусловной передачи управления:
 - команда [безусловного перехода](#);
 - [вызова процедуры и возврата из процедуры](#);
 - вызова программных прерываний и возврата из программных прерываний.
2. Команды [условной передачи управления](#):
 - команды [перехода по результату команды сравнения **сmp**](#);
 - команды [перехода по состоянию определенного флага](#);
 - команды [перехода по содержимому регистра **есх/сх**](#).
3. Команды [управления циклом](#):
 - команда [организации цикла со счетчиком **есх/сх**](#);
 - команда организации цикла со счетчиком **есх/сх** с возможностью [досрочного выхода из цикла по дополнительному условию](#).

БЕЗУСЛОВНЫЕ ПЕРЕХОДЫ

Предыдущее обсуждение выявило некоторые детали механизма перехода. Команды перехода модифицируют регистр указателя команды **еір/ір** и, возможно, сегментный регистр кода **сs**. Что именно должно подвергнуться модификации, зависит:

- от типа операнда в команде безусловного перехода (ближний или дальний);
- от указания перед адресом перехода (в команде перехода) *модификатора*; при этом сам адрес перехода может находиться либо непосредственно в команде (прямой переход), либо в регистре или ячейке памяти (косвенный переход).

Модификатор может принимать следующие значения:

- **near ptr** — прямой переход на метку внутри текущего сегмента кода. Модифицируется только регистр **еір/ір** (в зависимости от заданного типа сегмента кода **use16** или **use32**) на основе указанного в команде адреса (метки) или выражения, использующего символ извлечения значения СчА — **\$**;
- **far ptr** — прямой переход на метку в другом сегменте кода. Адрес перехода задается в виде непосредственного операнда или адреса (метки) и состоит из 16-битного селектора и 16/32-битного смещения, которые загружаются, соответственно, в регистры **сs** и **ір/еір**;
- **word ptr** — косвенный переход на метку внутри текущего сегмента кода. Модифицируется (значением смещения из памяти по указанному в команде адресу, или из регистра) только **еір/ір**. Размер смещения 16 или 32 бит;
- **dword ptr** — косвенный переход на метку в другом сегменте кода. Модифицируются (значением из памяти — и только из памяти, из регистра нельзя) оба регистра, **сs** и **еір/ір**. Первое слово/двойное слово этого адреса представляет смещение и загружается в **ір/еір**; второе/третье слово загружается в **сs**.

Команда безусловного перехода **jmp**

Синтаксис команды безусловного перехода

jmp [модификатор] адрес_перехода - безусловный переход без сохранения информации о точке возврата.

Адрес_перехода представляет собой адрес в виде метки либо адрес области памяти, в которой находится указатель перехода.

Всего в системе команд микропроцессора есть несколько кодов машинных команд безусловного перехода **jmp**. Их различия определяются дальностью перехода и способом задания целевого адреса.

Дальность перехода определяется местоположением операнда *адрес_перехода*. Этот адрес может находиться в текущем сегменте кода или в некотором другом сегменте. В первом случае переход называется *внутрисегментным*, или *ближним*, во втором — *межсегментным*, или *дальним*.

Внутрисегментный переход предполагает, что изменяется только содержимое регистра **еір/ір**.

Можно выделить три варианта внутрисегментного использования команды **jmp**:

- прямой короткий;
- прямой;
- косвенный.

УСЛОВНЫЕ ПЕРЕХОДЫ

Микропроцессор имеет 18 команд условного перехода (см. “Описание команд”). Эти команды позволяют проверить:

- отношение между операндами со знаком (“больше — меньше”);
- отношение между операндами без знака (“выше — ниже”)²;
- состояния арифметических флагов **zf**, **sf**, **cf**, **of**, **pf** (но не **af**).

Команды условного перехода имеют одинаковый синтаксис:

јсс метка_перехода

Как видно, мнемокод всех команд начинается с “**ј**” — от слова *jump* (прыжок), **сс** — определяет конкретное условие, анализируемое командой. Что касается операнда *метка_перехода*, то эта метка может находиться только в пределах текущего сегмента кода, межсегментная передача управления в условных переходах не допускается.

Для того чтобы принять решение о том, куда будет передано управление командой условного перехода, предварительно должно быть сформировано условие, на основании которого и будет приниматься решение о

передаче управления.

Источниками такого условия могут быть:

- любая команда, изменяющая состояние арифметических флагов;
- команда сравнения **cmp**, сравнивающая значения двух операндов;
- состояние регистра **ecx/cx**.

Обсудим эти варианты, чтобы разобраться с тем, как работают команды условного перехода.

КОМАНДА СРАВНЕНИЯ **cmp**

Команда сравнения **cmp** имеет интересный принцип работы. Он абсолютно такой же, как и у команды вычитания.

sub операнд_1,операнд_2. Команда **cmp** так же, как и команда **sub**, выполняет вычитание операндов и устанавливает флаги. Единственное, чего она не делает — это запись результата вычитания на место первого операнда.

Синтаксис команды **cmp**:

cmp операнд_1,операнд_2 (compare) — сравнивает два операнда и по результатам сравнения устанавливает флаги.

Флаги, устанавливаемые командой **cmp**, можно анализировать специальными командами условного перехода. Прежде чем мы их рассмотрим, уделим немного внимания мнемонике этих команд условного перехода (табл. 1). Понимание обозначений при формировании названия команд условного перехода (элемент в названии команды **jcc**, обозначенный нами **cc**) облегчит их запоминание и дальнейшее практическое использование.

Таблица 1. Значение аббревиатур в названии команды **jcc**

Мнемоническое обозначение	Английский	Русский	Тип операндов
E e	equal	Равно	Любые
N n	not	Не	Любые
G g	greater	Больше	Числа со знаком
L l	less	Меньше	Числа со знаком
A a	above	Выше, в смысле “больше”	Числа без знака
B b	below	Ниже, в смысле “меньше”	Числа без знака

Таблица 2. Перечень команд условного перехода для команды **cmp операнд_1,операнд_2**

Типы операндов	Мнемокод команды условного перехода	Критерий условного перехода	Значения флагов для осуществления перехода
Любые	je	операнд_1 = операнд_2	zf = 1
Любые	jne	операнд_1 <> операнд_2	zf = 0
Со знаком	jl/jnge	операнд_1 < операнд_2	sf <> of
Со знаком	jle/jng	операнд_1 <= операнд_2	sf <> of or zf = 1
Со знаком	jg/jnle	операнд_1 > операнд_2	sf = of and zf = 0
Со знаком	jge/jnl	операнд_1 >= операнд_2	sf = of
Без знака	jb/jnae	операнд_1 < операнд_2	cf = 1
Без знака	jbe/jna	операнд_1 <= операнд_2	cf = 1 or zf = 1
Без знака	ja/jnbe	операнд_1 > операнд_2	cf = 0 and zf = 0
Без знака	jae/jnb	операнд_1 >= операнд_2	cf = 0

Не удивляйтесь тому обстоятельству, что одинаковым значениям флагов соответствует несколько разных мнемокодов команд условного перехода (они отделены друг от друга косой чертой в табл. 2).

Разница в названии обусловлена желанием разработчиков микропроцессора облегчить использование команд условного перехода в сочетании с определенными группами команд. Поэтому разные названия отражают скорее различную функциональную направленность. Тем не менее, то, что эти команды реагируют на одни и те же флаги делает их абсолютно эквивалентными и равноправными в программе. Поэтому в табл. 2 они сгруппированы не по названиям, а по значениям флагов (условиям), на которые они реагируют.

Команды условного перехода и флаги

Мнемоническое обозначение некоторых команд условного перехода отражает название флага, с которым они работают, и имеет следующую структуру: первым идет символ “j” (*jump*, переход), вторым — либо обозначение флага, либо символ отрицания “n”, после которого стоит название флага.

Такая структура команды отражает ее назначение.

Если символа “n” нет, то проверяется состояние флага, и если он равен 1, производится переход на метку перехода.

Если символ “n” присутствует, то проверяется состояние флага на равенство 0, и в случае успеха производится переход на метку перехода.

Мнемокоды команд, названия флагов и условия переходов приведены в табл. 3.

Эти команды можно использовать после любых команд, изменяющих указанные флаги.

Таблица 3. Команды условного перехода и флаги

Название флага	Номер бита в eflags/flag	Команда условного перехода	Значение флага для осуществления перехода
Флаг переноса cf	1	jc	cf = 1
Флаг четности pf	2	jp	pf = 1
Флаг нуля zf	6	jz	zf = 1
Флаг знака sf	7	js	sf = 1
Флаг переполнения of	11	jo	of = 1
Флаг переноса cf	1	jnc	cf = 0
Флаг четности pf	2	jnp	pf = 0
Флаг нуля zf	6	jnz	zf = 0
Флаг знака sf	7	jns	sf = 0
Флаг переполнения of	11	jno	of = 0

Если внимательно посмотреть на табл. 2 и 3, видно, что многие команды условного перехода в них являются эквивалентными, так как в основе и тех, и других лежит анализ одинаковых флагов.

Команды условного перехода и регистр есх/сх

Архитектура микропроцессора предполагает специфическое использование многих регистров.

К примеру, регистр **eax/ax/al** используется как аккумулятор, а регистры **bp**, **sp** - для работы со стеком.

Регистр **есх/сх** тоже имеет определенное функциональное назначение — он выполняет роль *счетчика* в командах управления циклами и при работе с цепочками символов. Возможно, что функционально команду условного перехода, связанную с регистром **есх/сх**, правильнее было бы отнести к этой группе команд.

Синтаксис этой команды условного перехода таков:

jcхз метка_перехода (*Jump if cx is Zero*) — переход, если **сх** ноль;

jecхз метка_перехода (*Jump Equal esx Zero*) — переход, если **есх** ноль.

Эти команды очень удобно использовать при организации цикла и при работе с цепочками символов.

Нужно отметить ограничение, свойственное команде **jcхз/jecхз**. В отличие от других команд условной передачи управления, команда **jcхз/jecхз** может адресовать только короткие переходы — на –128 байт или на +127 байт от следующей за ней команды.

ОРГАНИЗАЦИЯ ЦИКЛОВ

Цикл, как известно, представляет собой важную алгоритмическую структуру, без использования которой не обходится, наверное, ни одна программа.

Организовать циклическое выполнение некоторого участка программы можно, к примеру, используя команды условной передачи управления или команду безусловного перехода **jmp**. При такой организации цикла все операции по его организации выполняются “вручную”. Но, учитывая важность такого алгоритмического элемента, как цикл, разработчики микропроцессора ввели в систему команд группу из трех команд, облегчающую программирование циклов. Эти команды также используют регистр **есх/сх** как *счетчик цикла*. Дадим краткую характеристику этим командам:

loop метка_перехода (Loop) — повторить цикл. Команда позволяет организовать циклы, подобные циклам for в языках высокого уровня с автоматическим уменьшением счетчика цикла. Работа команды заключается в выполнении следующих действий:

- декремента регистра `ecx/cx`;
- сравнения регистра `ecx/cx` с нулем:
 - если $(ecx/cx) > 0$, то управление передается на метку перехода;
 - если $(ecx/cx) = 0$, то управление передается на следующую после `loop` команду.

loope/loopz метка_перехода (Loop till `cx <> 0` or Zero Flag = 0) — повторить цикл, пока `cx <> 0` или `zf = 0`. Команды `loope` и `loopz` — абсолютные синонимы, поэтому используйте ту команду, которая вам больше нравится. Работа команд заключается в выполнении следующих действий:

- декремента регистра `ecx/cx`;
- сравнения регистра `ecx/cx` с нулем;
- анализа состояния флага нуля `zf`:
 - если $(ecx/cx) > 0$ и `zf = 1`, управление передается на метку перехода;
 - если $(ecx/cx) = 0$ или `zf = 0`, управление передается на следующую после `loop` команду.

loopne/loopnz метка_перехода (Loop till `cx <> 0` or Not Zero flag=0) — повторить цикл пока `cx <> 0` или `zf = 1`. Команды `loopne` и `loopnz` также абсолютные синонимы. Работа команд заключается в выполнении следующих действий:

- декремента регистра `ecx/cx`;
- сравнения регистра `ecx/cx` с нулем;
- анализа состояния флага нуля `zf`:
 - если $(ecx/cx) > 0$ и `zf = 0`, управление передается на метку перехода;
 - если $(ecx/cx) = 0$ или `zf = 1`, управление передается на следующую после `loop` команду.

Команды **loope/loopz** и **loopne/loopnz** по принципу своей работы являются взаимобратными. Они расширяют действие команды **loop** тем, что дополнительно анализируют флаг `zf`, что дает возможность организовать досрочный выход из цикла, используя этот флаг в качестве индикатора.

Недостаток команд организации цикла **loop**, **loope/loopz** и **loopne/loopnz** в том, что они реализуют только короткие переходы (от -128 до +127 байт). Для работы с длинными циклами придется использовать команды условного перехода и команду `jmp`, поэтому постарайтесь освоить оба способа организации циклов.

Рассмотрим простой пример организации цикла.

Пусть в программе зарезервировано место для массива размером 10000 слов, и этот массив надо заполнить натуральным рядом чисел от 0 до 9999. Эти числа, заполняющие последовательные элементы массива, иногда называют числами-заполнителями. Соответствующий фрагмент программы будет выглядеть следующим образом:

```
;В сегменте данных
array dw 10000 dup(0)
;В программном сегменте
mov BX,offset array ; Адрес массива
mov SI,0 ;Индекс
mov AX,0 ; Начальное значение заполнителя
mov CX,10000 ; Счетчик цикла
fill: mov [BX][SI],AX ;Заполнитель пошлем в массив
inc AX ;Инкремент заполнителя
add SI,2 ; модификация индекса
loop fill ; Команда цикла
```

На этапе подготовки мы заносим в регистр `BX` относительный адрес начала массива, отождествляемый с его именем `array`, устанавливаем начальное значение индекса элемента массива в регистре `SI` (с таким же успехом можно было взять `DI`) и начальное значение числа-заполнителя. Сам цикл состоит из трех команд - единственной содержательной команды засылки числа-заполнителя в очередной элемент массива (по адресу, который вычисляется, как сумма содержимого регистров `BX` и `SI`), а также модификации числа-заполнителя и индекса очередного элемента массива. Завершающей командой `loop` управление передается на метку `fill`, и цикл повторяется столько раз, каково содержимое `CX`, в данном случае 10000 шагов.

Следует обратить внимание на команду модификации индекса - в каждом шаге к содержимому SI добавляется 2, так как массив состоит из двухбайтовых слов. Если бы нужно было заполнить байтовый массив, то в каждом шаге содержимое регистра цикла SI следовало увеличивать на 1.

Стоит отметить некоторые детали, связанные с механизмом выполнения команды loop. При реализации этой команды процессор сначала уменьшает содержимое регистра CX на 1, а затем сравнивает полученное число с нулем. Если $CX > 0$, переход на указанную метку выполняется. Если $CX = 0$, цикл разрывается и процессор переходит на команду, следующую за командой loop. Поэтому после нормального выхода из цикла содержимое CX всегда равно 0.

Другое обстоятельство связано с кодированием команды loop. В ее коде под смещение к точке перехода отводится всего 1 байт. Поскольку смещение должно являться величиной со знаком, максимальное расстояние, на которое можно передать управление командой loop, составляет от -128 до +127 байт (хотя довольно трудно представить себе цикл, в котором переход осуществляется вперед). Другими словами, тело цикла ограничивается всего 128 байтами. Если циклически повторяемый фрагмент программы имеет большую длину, цикл придется организовать другим, более сложным способом.

9.1. Процедуры.

ПРОЦЕДУРЫ

В языке ассемблера есть несколько средств, решающих проблему дублирования участков программного кода. К ним относятся:

- механизм процедур;
- [макроассемблер](#);
- механизм прерываний.

Процедура, часто называемая также *подпрограммой*, — это основная функциональная единица декомпозиции (разделения на несколько частей) некоторой задачи.

Процедура представляет собой группу команд для решения конкретной подзадачи и обладает средствами получения управления из точки вызова задачи более высокого уровня и возврата управления в эту точку.

В простейшем случае программа может состоять из одной процедуры. Другими словами, процедуру можно определить как правильным образом оформленную совокупность команд, которая, будучи однократно описана, при необходимости может быть вызвана в любом месте программы.

Для описания последовательности команд в виде процедуры в языке ассемблера используются две директивы: **PROC** и **ENDP**.

Синтаксис описания процедуры таков (рис. 1).

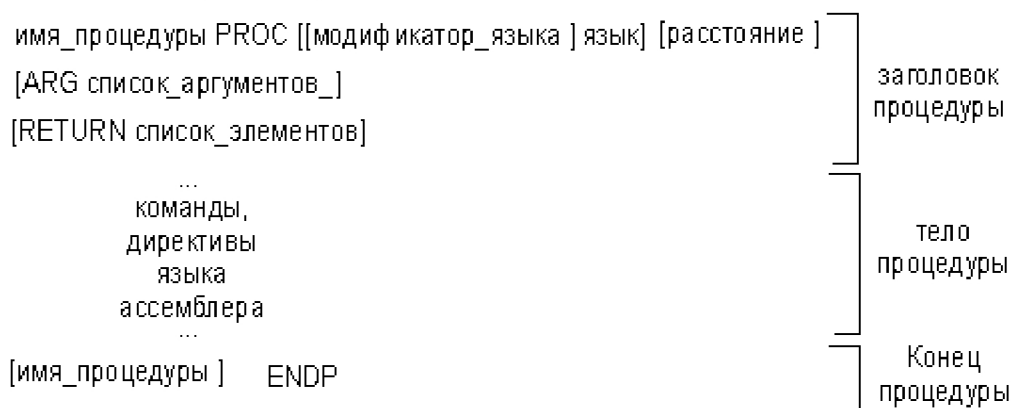


Рис. 1. Синтаксис описания процедуры в программе

Из рис. 1 видно, что в заголовке процедуры (директиве PROC) обязательным является только задание имени процедуры. Среди большого количества операндов директивы PROC следует особо выделить **[расстояние]**. Этот атрибут может принимать значения **near** или **far** и характеризует возможность обращения к процедуре из другого сегмента кода. По умолчанию атрибут **[расстояние]** принимает значение **near**.

Процедура может размещаться в любом месте программы, но так, чтобы на нее случайным образом не попало управление. Если процедуру просто вставить в общий поток команд, то микропроцессор будет воспринимать команды процедуры как часть этого потока и соответственно будет осуществлять выполнение команд процедуры.

ИСПОЛЬЗОВАНИЕ ПОДПРОГРАММ

Общая идея использования подпрограмм очевидна: если в программе требуется многократно выполнять один и тот же фрагмент, его можно оформить в виде подпрограммы и вызывать по мере необходимости. Если подпрограмма не требует для своего выполнения никаких параметров и не должна возвращать в основную программу результат своей работы, то дело ограничивается оформлением текста подпрограммы в виде процедуры, завершающейся командой `ret`, и вызовом этой процедуры с помощью команды `call`. Как уже отмечалось ранее, подпрограмма может и не образовывать процедуру, а быть просто частью основной программы. Важно только, чтобы у нее была входная метка, и чтобы она завершалась командой `ret`. В следующем примере подпрограмма `delay` используется для включения в основной текст программы программных задержек фиксированной величины.

Пример 3-8. Вызов подпрограммы без параметров

```
code segment
assume cs:code,ds:data
delay proc                                ;Процедура-подпрограмма
push CX                                  ;Сохраним CX основной программы
mov CX,2000                              ;Счетчик внешнего цикла
del1: push CX                            ;Сохраним его
mov CX,0                                  ;Счетчик внутреннего цикла
del2: loop del2                          ;Внутренний цикл (64К шагов)
pop CX                                   ;Восстановим внешний счетчик
loop del1                                ;Внешний цикл (2000 шагов)
pop CX                                   ; Восстановим CX программы
ret                                       ;Возврат в подпрограмму
delay endp
main proc
mov AX,data                             ;Настроим DS
mov DX,AX                               ;на сегмент данных
mov AH,09h                              ;Функция вывода на экран
mov DX,offset npl1                      ;Адрес первой строки
mov CX,3                                ;Будем выводить строки в цикле
cntrl1: int 21h                          ;Вызов DOS
call delay                               ;Вызов подпрограммы задержки
add DX,msg_len                           ;Прибавим к смещению длину строки
loop cntrl                               ;Цикл вызовов DOS
mov AX,4C00h                             ;Завершение программы
int 21h
main endp
code ends
data segment
msg1 db "Процесс стартовал",13,10,'$'
msg_len=$-msg1
msg2 db "Процесс идет",13,10,'$'
msg3 db "Процесс завершается",13,10,'$'
data ends
stk segment stack
dw 128 dup(')
stk ends
end main
```

В тексте программы сначала описана процедура-подпрограмма, затем основная программа. Как уже отмечалось, порядок их описания роли не играет; важно только, чтобы в завершающей директиве окончания трансляции `end` был указан в качестве точки входа адрес основной программы (`main` в нашем примере). Подпрограмма реализует задержку с помощью вложенных циклов с командой `loop`, использующей в качестве счетчика шагов регистр `CX`. В основной программе этот регистр используется для организации цикла вывода трех строк. Поэтому первое, что должна сделать подпрограмма - это сохранить содержимое регистра `CX`, для чего естественно использовать стек. Перед завершающей командой `ret` регистр `CX` должен быть восстановлен. Фрагмент, реализующий задержку, был описан ранее, в разделе 3.2.

Основная программа выводит на экран с помощью функции `09h` три строки текста. Для упрощения программы, а также чтобы продемонстрировать некоторые приемы программирования, вывод строк реализован в цикле. Строки сделаны одной длины, и модификация смещения к очередной строке выполняется прибавлением к содержимому регистра `DX` длины строки. Полезно обратить внимание на организацию цикла в основной

программе. В цикл, помимо команды вызова подпрограммы задержки и предложения, модифицирующего регистр DX, включена лишь команда `int 21h`. Регистр AH с номером функции заново не настраивается. Это и не нужно, так как DOS, выполняя затребованную операцию, первым делом сохраняет все регистры программы, а перед возвратом в программу их восстанавливает. Поэтому, вызывая функции DOS (или BIOS) можно не заботиться о сохранении регистров - их содержимое система не разрушает. Надо только иметь в виду, что многие функции DOS и BIOS после своего завершения возвращают в программу некоторую информацию (число реально введенных символов, доступный объем памяти, номер видеорежима и т.п.) Обычно эта информация возвращается в регистре AX, однако могут использоваться и другие регистры или их сочетания. Поэтому, обращаясь в программе к системным функциям, необходимо ознакомиться с их описанием и, в частности, посмотреть, какие регистры они могут использовать для возвращаемых значений. Запустив программу, можно убедиться в том, что строки текста появляются на экране через заметные промежутки времени.

В примере 3-8 подпрограмма не требовала параметров. Чаше, однако, подпрограмма должна принимать один или несколько параметров и возвращать результат. В этом случае необходимо организовать взаимодействие основной программы и подпрограммы. Никаких специальных средств языка для этого не существует; передачу параметров в подпрограмму и из нее программист организует по своему усмотрению. Для передачи параметров как в одну, так и в другую сторону можно использовать регистры общего назначения, ячейки памяти или стек.

Передача параметров в подпрограмму через регистры общего назначения или даже через сегментные регистры вполне возможна, однако на практике для передачи параметров чаще всего используют стек, хотя бы потому, что регистров немного, а в стек можно поместить любое число параметров. При этом применяется своеобразная методика работы со стеком не с помощью команд `push` и `pop`, а с помощью команд `mov` с косвенной адресацией через регистр BP, который архитектурно предназначен именно для адресации к стеку.

Преобразуем пример 3-8а так, чтобы единственный в этом примере параметр (условная величина задержки) передавался в подпрограмму не через регистр SI, а через стек. Вызов подпрограммы `delay` в этом случае должен выполняться следующим образом:

```
push 2000 ;Проталкиваем в стек значение параметра
```

```
call delay ;Вызываем подпрограмму delay
```

Текст подпрограммы подвергнется значительным изменениям:

Пример 3-8в. Передача параметра через стек

```
delay proc                ;Процедура-подпрограмма
push CX                  ;Сохраним CX основной программы
push BP                  ;Сохраним BP
mov BP,SP                ;Настроим BP на текущую вершину стека
mov CX, [BP+6]           ;Скопируем из стека параметр
del1: push CX            ;Сохраним его
mov CX,0                 ;Счетчик внутреннего цикла
del2 loop del2           ;Внутренний цикл (64К шагов)
pop CX                   ;Восстановим внешний счетчик
loop del1                ;Внешний цикл
pop BP                   ;Восстановим BP
pop CX                   ;и CX программы
ret 2                    ;Возврат и снятие со стека
                        ;ненужного уже параметра
```

Команда `call`, передавая управление подпрограмме, сохраняет в стеке адрес возврата в основную программу. Подпрограмма сохраняет в стеке еще два 16-разрядных регистра. В результате стек оказывается в состоянии, изображенном на рис. 3.9.

После сохранения в стеке исходного содержимого регистра BP (в основной программе нашего примера этот регистр не используется, однако в общем случае это может быть и не так), в регистр BP копируется содержимое указателя стека, после чего в BP оказывается смещение вершины стека. Далее командой `mov` в регистр CX заносится содержимое ячейки стека, на 6 байтов ниже текущей вершины. В этом месте стека как раз находится передаваемый в подпрограмму параметр, как это показано в левом столбце рис. 3.8. Конкретную величину смещения относительно вершины стека надо для каждой подпрограммы определять индивидуально,



Рис. 3.8. Состояние стека в подпрограмме после сохранения регистров.

исходя из того, сколько слов сохранено ею в стеке к этому моменту. Напомним, что при использовании косвенной адресации с регистром BP в качестве базового, по умолчанию адресуется стек, что в данном случае и требуется.

Параметр, полученный таким образом, используется далее в подпрограмме точно так же, как и в примере 3-8а. Выполнив возложенную на нее задачу, подпрограмма восстанавливает сохраненные ранее регистры и осуществляет возврат в основную программу с помощью команды `ret`, в качестве аргумента которой указывается число байтов, занимаемых в стеке отправленными туда перед вызовом подпрограммы параметрами. В нашем случае единственный параметр занимает 2 байт. Если здесь использовать обычную команду `ret` без аргумента, то после возврата в основную программу параметр останется в стеке, и его надо будет оттуда извлекать (между прочим, не очень понятно, куда именно, ведь все регистры у нас могут быть заняты). Команда же с аргументом, осуществив возврат в вызывающую программу, увеличивает содержимое указателя стека на значение ее аргумента, тем самым осуществляя логическое снятие параметра. Физически этот параметр, как, впрочем, и все остальные данные, помещенные в стек, остается в стеке и будет затерт при дальнейших обращениях к стеку.

Разумеется, в стек можно было поместить не один, а сколько угодно параметров. Тогда для их чтения надо было использовать несколько команд `mov` со значениями смещения BP+6, BP+8, BP+0Ah и т.д.

Рассмотренная методика может быть использована и при дальних вызовах подпрограмм, но в этом случае необходимо учитывать, что дальняя команда `call` сохраняет в стеке не одно, а два слова, что повлияет на величину рассчитываемого смещения относительно вершины стека.

Вызовы подпрограмм

Практически в любой программе, независимо от ее содержания, встречаются участки, которые требуется выполнять (возможно, с небольшими изменениями) несколько раз по ходу программы. Такие повторяющиеся участки целесообразно выделить из общей программы, оформить в виде подпрограмм и обращаться к ним каждый раз, когда в основной программе возникает необходимость их выполнения.

Подпрограмма, в зависимости от выполняемых ею функций, может требовать передачи из вызывающей программы определенных данных (называемых аргументами, или параметрами), возвращать в вызывающую программу результаты вычислений или обходиться и без того, и без другого.

Подпрограмма может быть оформлена в виде процедуры, и тогда имя этой процедуры будет служить точкой входа в подпрограмму:

```
drawline proc ;Подпрограмма-процедура
. . . ;Тело подпрограммы
ret ;Команда возврата в вызывающую программу
drawline endp
```

С таким же успехом можно обойтись без процедуры, просто пометив первую строку программы некоторой меткой:

```
drawline: ;Подпрограмма, начинающаяся с метки
. . . ;Тело подпрограммы
ret ;Команда возврата в вызывающую программу
. . . ;Продолжение основной программы или ;другие подпрограммы
```

В любом случае вызов подпрограммы осуществляется командой `call`. Подпрограмма должна завершаться командой `ret`, служащей для возврата управления в ту точку, откуда подпрограмма была вызвана. Вопросы использования подпрограмм, передачи в них параметров и возвращения результата будут рассмотрены в следующей главе. Здесь мы остановимся только на таких принципиальных архитектурных вопросах, как механизм выполнения и возможности команд `call` и `ret`. При этом надо иметь в виду, что синтаксические особенности и закономерности использования команд `call` и `jmp` во многом совпадают, и

значительная часть пояснений к командам перехода справедлива и для команд вызова.

Команда вызова подпрограммы `call` может использоваться в 4 разновидностях. Вызов может быть:

- прямым ближним (в пределах текущего сегмента команд);
- прямым дальним (в другой сегмент команд);
- косвенным ближним (в пределах текущего сегмента команд через ячейку с адресом перехода);
- косвенным дальним (в другой сегмент команд через ячейку с адресом перехода).

Рассмотрим последовательно перечисленные варианты.

Прямой ближний вызов. Как и в случае прямого ближнего перехода, в команде прямого вызова в явной форме указывается адрес (смещение) точки входа в подпрограмму; в качестве этого адреса можно использовать как имя процедуры, так и имя метки, характеризующей точку входа в подпрограмму. В код команды, кроме кода операции `E8h`, входит смещение к вызываемой подпрограмме. В приведенном ниже примере подпрограмма оформлена в виде процедуры.

```
code segment
main proc ;Основная программа
...
call sub ;Код E8 dddd
...
main endp
sub proc near ;Подпрограмма
...
ret ;Код C3
sub endp
code ends
```

Процедура-программа находится в том же сегменте команд, что и вызывающая программа. В коде команды `dddd` обозначает смещение в сегменте команд к точке входа в подпрограмму. При выполнении команды `call` процессор помещает адрес возврата (содержимое регистра `IP`) в стек выполняемой программы (рис. 2.16), после чего к текущему содержимому `IP` прибавляет `dddd`. В результате в `IP` оказывается адрес подпрограммы. Команда `ret`, которой заканчивается подпрограмма, выполняет обратную процедуру - извлекает из стека адрес возврата и заносит его в `IP`.

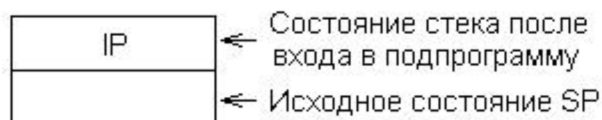


Рис. 2.16. Участие стека в механизме вызова ближней подпрограммы.

Участие стека в механизме вызова подпрограммы и возврата из нее является решающим. Поскольку в стеке хранится адрес возврата, подпрограмма, сама используя стек, например, для хранения промежуточных результатов, обязана к моменту выполнения команды `ret` вернуть стек в исходное состояние. Команда `ret`, естественно, никак не анализирует состояние или содержимое стека. Она просто снимает со стека верхнее слово, считая его адресом возврата, и загружает это слово в указатель команд `IP`. Если к моменту выполнения команды `ret` указатель стека окажется смещенным в ту или иную сторону, команда `ret` по-прежнему будет рассматривать верхнее слово стека, как адрес возврата, и передаст по нему управление, что неминуемо приведет к краху системы.

Прямой дальний вызов. Этот вызов позволяет обратиться к подпрограмме из другого сегмента. В код команды, кроме кода операции `9Ah`, входит полный адрес (сегмент плюс смещение) вызываемой подпрограммы. Обычно в исходном тексте программы с помощью описателя `far ptr` указывается, что вызов является дальним, хотя, если транслятор настроен на трансляцию в два прохода, этот описатель не обязателен. Структура программного комплекса, содержащая дальний вызов подпрограммы, может выглядеть следующим образом:

```
code1 segment
assume CS:code1
main proc ;Основная программа
call far ptr subr ; Код 9A dddd ssss
...
main endp
code1 ends
code2 segment
assume CS:code2
```

```

subr proc far ;Объявляем подпрограмму дальней
...
ret ;Код CB - дальний возврат
subr endp
code2 ends

```

Процедура-подпрограмма находится в другом сегменте команд той же программы. В коде команды dddd обозначает относительный адрес точки входа в подпрограмму в ее сегменте команд, а ssss - ее сегментный адрес. При выполнении команды call процессор помещает в стек сначала сегментный адрес вызывающей программы, а затем относительный адрес возврата (рис. 2.17). Далее в сегментный регистр CS заносится 5555 (у нас это значение code2), а в IP - dddd (у нас это значение subr). Поскольку процедура-подпрограмма атрибутом far объявлена дальней, команда ret имеет код, отличный от кода аналогичной команды ближней процедуры и выполняется по-другому: из стека извлекаются два верхних слова и переносятся в IP и CS, чем и осуществляется возврат в вызывающую программу, находящуюся в другом сегменте команд. В языке ассемблера существует и явное mnemonic обозначение команды дальнего возврата - rtf.



Рис. 2.17. Участие стека в механизме вызова дальней подпрограммы.

Косвенный ближний вызов. Адрес подпрограммы содержится либо в ячейке памяти, либо в регистре. Это позволяет, как и в случае косвенного ближнего перехода, модифицировать адрес вызова, а также осуществлять вызов не с помощью метки, а по известному абсолютному адресу. Структура программы с косвенным вызовом подпрограммы может выглядеть следующим образом:

```

code segment
main proc ;Основная программа
...
call DS:subadr ;Код FF 16 dddd
main endp

subr proc near ;Подпрограмма
...
ret ;Код C3
subr endp
code ends
data segment
...
subadr dw subr ;Ячейка с адресом подпрограммы
data ends

```

Процедура-программа с атрибутом near находится в том же сегменте, что и вызывающая программа, а ее относительный адрес в ячейке subadr в сегменте данных. В коде команды dddd обозначает относительный адрес слова subadr в сегменте данных. Второй байт кода команды (16h в данном примере) зависит от способа адресации. Косвенный вызов позволяет использовать разнообразные способы адресации подпрограммы:

```

call BX ; В BX адрес подпрограммы

call[BX] ; В BX адрес ячейки с адресом подпрограммы

call[BX][SI] ; В BX адрес таблицы адресов подпрограмм,
              ; в SI индекс в этой таблице.

tbl[SI] ;tbl - адрес таблицы адресов подпрограмм,
; в SI индекс в этой таблице

```

Косвенный дальний вызов. Отличается от косвенного ближнего вызова лишь тем, что подпрограмма находится в другом сегменте, а в ячейке памяти содержится полный адрес подпрограммы, включающий сегмент и смещение.

```

code1 segment
main proc ;Основная программа
call dword ptr subadr ;Код FF IE dddd
...
main endp
code1 ends
code2 segment
subr proc far ;Подпрограмма
...
ret ;Код CB
subr endp

code2 ends
data segment
...
subadr dd subr ;Двухсловная ячейка с адресом подпрограммы
data ends

```

Процедура-подпрограмма с атрибутом *far* находится в другом сегменте команд той же программы, а ее полный двухсловный адрес - в ячейке *subadr* в сегменте данных. Второй байт кода команды (IE в данном примере) зависит от способа адресации. Косвенный дальний вызов, как и косвенный ближний, позволяет использовать различные способы адресации.

9.2. Система прерываний.

СИСТЕМА ПРЕРЫВАНИЙ

Система прерываний любого компьютера является его важнейшей частью, позволяющей быстро реагировать на события, обработка которых должна выполняться немедленно: сигналы от машинных таймеров, нажатия клавиш клавиатуры или мыши, сбой памяти и пр. Рассмотрим в общих чертах компоненты этой системы.

Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера или подключенных к нему, поступают в процессор не непосредственно, а через два контроллера прерываний, один из которых называется ведущим, а второй - ведомым (рис. 1.11)

Прерывание представляет собой операцию, которая приостанавливает выполнение программ для специальных системных действий. Необходимость прерываний обусловлено двумя основными причинами: преднамеренный запрос таких действий, как операции ввода-вывода на различные устройства и непредвиденные программные ошибки (например, переполнение при делении).

Система BIOS (Basic Input/Output System) находится в ROM и управляет всеми прерываниями в системе. В предыдущих главах уже использовались некоторые прерывания для вывода на экран дисковых операций ввода-вывода и печати. В этой главе описаны различные BIOS- и DOS-прерывания, резидентные программы и команды IN и OUT.

В компьютерах IBM PC ROM находится по адресу FFFF0H. При включении компьютера процессор устанавливает состояние сброса, выполняет контроль четности, устанавливает в регистре CS значение FFFFH, а в регистре IP - нуль. Первая выполняемая команда поэтому находится по адресу FFFF:0 или FFFF0, что является точкой входа в BIOS. BIOS проверяет различные порты компьютера для определения и инициализации подключенных устройств. Затем BIOS создает в начале памяти (по адресу 0) таблицу прерываний, которая содержит адреса обработчиков прерываний, и выполняет две операции INT 11H (запрос списка присоединенного оборудования) и INT 12H (запрос размера физической памяти).

Следующим шагом BIOS определяет имеется ли на диске или дискете операционная система DOS. Если обнаружена системная дискета, то BIOS выполняет прерывание INT 19H для доступа к первому сектору диска, содержащему блок начальной загрузки. Этот блок представляет собой программу, которая считывает системные файлы COMMAND.COM с диска в память.

АППАРАТНАЯ ОРГАНИЗАЦИЯ ПРЕРЫВАНИЙ.

Два контроллера используются для увеличения допустимого количества внешних устройств. Дело в том, что каждый контроллер прерываний может обслуживать сигналы лишь от 8 устройств. Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них веерообразную структуру. В современных машинах устанавливают два контроллера, увеличивая тем самым возможное число входных устройств до 15 (7 у ведущего и 8 у ведомого контроллеров).



К входным выводам IRQ1...IRQ7 и IRQ8...IRQ15 (IRQ - это сокращение от Interrupt Request, запрос прерывания) подключаются выходы устройств, на которых возникают сигналы прерываний. Выход ведущего контроллера подключается к входу INT микропроцессора, а выход ведомого - к входу IRQ2 ведущего. Основная функция контроллеров - передача сигналов запросов прерываний от внешних устройств на единственный вход прерываний микропроцессора. При этом, кроме сигнала INT, контроллеры передают в микропроцессор по линиям данных номер вектора, который образуется в контроллере путем сложения базового номера, записанного в одном из его регистров, с номером входной линии, по которой поступил запрос прерывания. Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Для ведущего контроллера базовый вектор всегда равен 8, для ведомого - 70h. Таким образом, номера векторов, закрепленных за аппаратными прерываниями, лежат в диапазонах 8h...Fh и 70h...77h. Очевидно, что номера векторов аппаратных прерываний однозначно связаны с номерами линий, или уровнями IRQ, а через них - с конкретными устройствами компьютера.

На рис. 1.11 указаны некоторые из стандартных устройств компьютера, работающих в режиме прерываний. Процессор, получив сигнал прерывания, выполняет последовательность стандартных действий, обычно называемых процедурой прерывания. Подчеркнем, что здесь идет речь лишь о реакции самого процессора на сигналы прерываний, а не об алгоритмах обработки прерываний, предусматриваемых пользователем в программах обработки прерываний.

Объекты вычислительной системы, принимающие участие в процедуре прерывания, и их взаимодействие показаны на рис. 1.12.

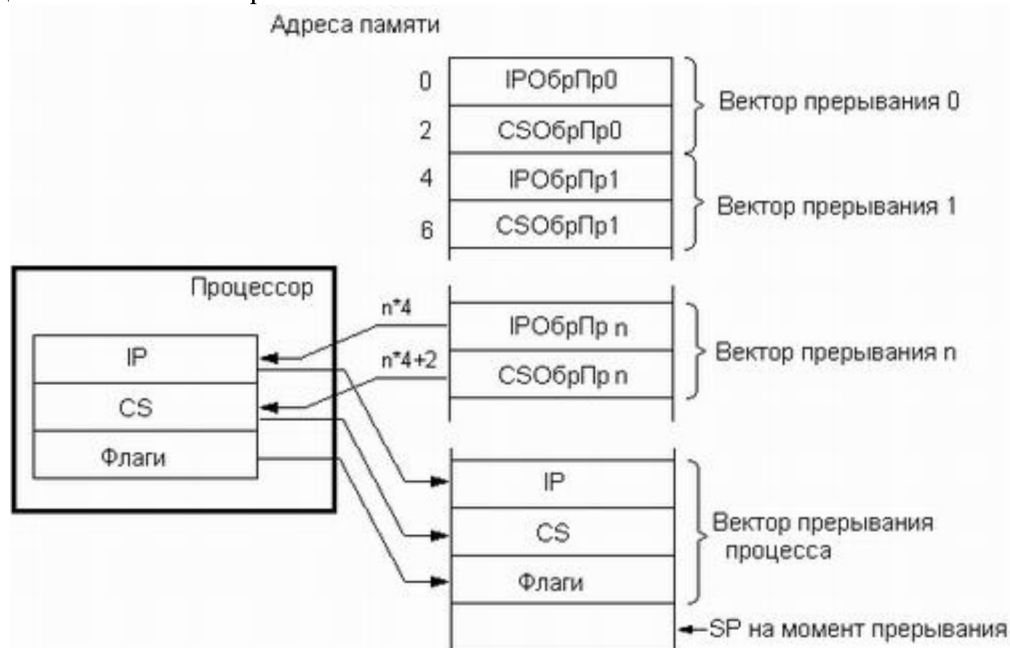


Рис. 1.12. Процедура обслуживания прерывания.

ОБРАБОТКА ПРЕРЫВАНИЙ

Самое начало оперативной памяти от адреса 0000h до 03FFh отводится под векторы прерываний - четырехбайтовые области, в которых хранятся адреса обработчиков прерываний (ОбрПр на рис. 1.12). В два старшие байта каждого вектора записывается сегментный адрес обработчика, в два младшие - смещение (относительный адрес) точки входа в обработчик. Векторы, как и соответствующие им прерывания, имеют номера, причем вектор с номером 0 располагается, начиная с адреса 0, вектор 1 - с адреса 4, вектор 2 - с адреса 8 и т.д. Вектор с номером n занимает, таким образом, байты памяти от $n*4$ до $n*4+3$. Всего в выделенной под векторы области памяти помещается 256 векторов.

Получив сигнал на выполнение процедуры прерывания с определенным номером, процессор сохраняет в стеке выполняемой программы текущее содержимое трех регистров процессора: регистра флагов, CS и IP. Два последних числа образуют полный адрес возврата в прерванную программу. Далее процессор загружает CS и IP из соответствующего вектора прерываний, осуществляя, тем самым, переход на обработчик прерывания, связанный с этим вектором.

Обработчик прерываний всегда заканчивается командой `iret` (interrupt return, возврат из прерывания), выполняющей обратные действия - извлечение из стека сохраненных там слов и помещение их назад в регистры IP и CS, а также в регистр флагов. Это приводит к возврату в основную программу в ту самую точку, где она была прервана.

В действительности запросы на обработку прерываний могут иметь различную природу. Помимо описанных выше аппаратных прерывания от периферийных устройств, называемых часто внешними, имеются еще два типа прерываний: внутренние и программные.

Внутренние прерывания возбуждаются цепями самого процессора при возникновении одной из специально оговоренных ситуаций, например, при выполнении операции деления на ноль или при попытке выполнить несуществующую команду. За каждым из таких прерываний закреплен определенный вектор, номер которого известен процессору. Например, за делением на 0 закреплен вектор 0, а за неправильной командой - вектор 6. Если процессор сталкивается с одной из таких ситуаций, он выполняет описанную выше процедуру прерывания, используя закрепленный за этой ситуацией вектор прерывания.

ПРОГРАММНЫЕ ПРЕРЫВАНИЯ.

Наконец, еще одним чрезвычайно важным типом прерываний являются программные прерывания. Они вызываются командой `int` с числовым аргументом, который рассматривается процессором, как номер вектора прерывания. Если в программе встречается, например, команда

```
int 13h
```

то процессор выполняет ту же процедуру прерывания, используя в качестве номера вектора операнд команды `int`. Программные прерывания применяются в первую очередь для вызова системных обслуживающих программ - функций DOS и BIOS. С командой `int 21h` вызова DOS мы уже сталкивались в примере 1-1 и будем встречаться еще многократно. В дальнейшем будут также приведены примеры использования команды `int` для вызова прикладных обработчиков программных прерываний.

Важно подчеркнуть, что описанные действия процессора выполняются совершенно одинаково для всех видов прерываний - внутренних, аппаратных и программных, хотя причины, возбуждающие процедуру прерывания, имеют принципиально разную природу.

Большая часть векторов прерываний зарезервирована для выполнения определенных действий; часть из них автоматически заполняется адресами системных программ при загрузке системы. Приведем краткую выдержку из таблицы векторов, позволяющую продемонстрировать разнообразие ее состава:

00h	внутреннее прерывание, деление на 0;
01h	внутреннее прерывание, пошаговое выполнение (при TF=1);
02h	немаскируемое прерывание (вывод NMI процессора);
08h	аппаратное прерывание от системного таймера;
09h	аппаратное прерывание от клавиатуры;
0Eh	аппаратное прерывание от гибкого диска;
10h	программное прерывание, программы BIOS управления видеосистемой;
13h	программное прерывание, программы BIOS управления дисками;
16h	программное прерывание, программы BIOS управления клавиатурой;
1Dh	не вектор, адрес таблицы видеопараметров, используемой BIOS;
1Eh	не вектор, адрес таблицы параметров дискеты, используемой BIOS;
21h	программное прерывание, диспетчер функций DOS;
22h	программное прерывание, адрес перехода при завершении процесса, используемый DOS;
23h	программное прерывание, обработчик прерываний по <Ctrl>/C, используемый DOS;
25h	программное прерывание, абсолютное чтение диска (функция DOS);
26h	программное прерывание, абсолютная запись на диск (функция DOS);
60h...66h	зарезервировано для программных прерываний пользователя;
68h...6Fh	программные прерывания, свободные векторы;
70h	аппаратное прерывание от часов реального времени (с питанием от аккумулятора);
76h	аппаратное прерывание от жесткого диска;

Как видно из таблицы, векторы прерываний можно условно разбить на следующие группы:

- векторы внутренних прерываний процессора (01h, 02h и др.);
- векторы аппаратных прерываний (08h...0Fh и 70h...77h);
- программы BIOS обслуживания аппаратуры компьютера (10h, 13h, 16h и др.);
- программы DOS (21h, 22h, 23h и др.);
- адреса системных таблиц BIOS (IDh, IEh и др.).

Системные программы, адреса которых хранятся в векторах прерываний, в большинстве своем являются всего лишь диспетчерами, открывающими доступ к большим группам программ, реализующих системные функции. Так, видеодрайвер BIOS (вектор 10h) включает программы смены видеорежима, управления курсором, задания цветовой палитры, загрузки шрифтов и многие другие. Особенно характерен в этом отношении вектор 21h, через который осуществляется вызов практически всех функций DOS: ввода с клавиатуры и вывода на экран, обслуживания файлов, каталогов и дисков, управления памятью и процессами, службы времени и т.д. Для вызова требуемой функции надо не только выполнить команду `int` с соответствующим номером, но и указать системе в одном из регистров (для этой цели всегда используется регистр AH) номер вызываемой функции. Иногда для "многофункциональных" функций приходится указывать еще и номер подфункции (в регистре AL).

9.3. Программные прерывания BIOS и DOS

ПРЕРЫВАНИЯ BIOS

INT 05H. Печата т ь э к р а н а. Выполняет вывод содержимого экрана на печатающее устройство. Команда INT 05H выполняет данную операцию из программы, а нажатие клавишей Ctrl/PrtSc – с клавиатуры. Операция запрещает прерывания и сохраняет позицию курсора.

INT 10H. У п р а в л е н и е д и с п л е е м. Обеспечивает экранные и клавиатурные операции, детально описанные в гл.9.

INT 11H. З а п р о с с п и с к а п р и с о е д и н е н н о г о о б о р у д о в а н и я. Определяет наличие различных устройств в системе, результирующее значение возвращает в регистре AX.

INT 12H З а п р о с р а з м е р а ф и з и ч е с к о й п а м я т и. Возвращает в регистре AX размер памяти в килобайтах

INT 13H. Д и с к о в ы е о п е р а ц и и в в о д а – в ы в о д а.

INT 16H. В в о д с к л а в и а т у р ы. Обеспечивает три типа команд ввода с клавиатуры, подробно описанные в гл.9.

INT 17H. В ы в о д н а п р и н т е р. Обеспечивает вывод данных на печатающее устройство. Подробно рассмотрено в гл.19.

ПРЕРЫВАНИЯ DOS

INT 20H. З а в е р ш е н и е п р о г р а м м ы. Запрос завершает выполнение программы и передает управление в DOS. Данный запрос обычно находится в основной процедуре.

INT 21H. З а п р о с ф у н к ц и й DOS. Основная операция DOS, вызывающая определенную функцию в соответствии с кодом в регистре AH. Назначение функций DOS описано в следующем разделе.

INT 27H.З а в е р ш е н и е п р о г р а м м ы, о с т а в л я ю щ е е е е р е з е д е н т н о й. Позволяет сохранить COM-программу в памяти. Подробно данная операция рассмотрена в последующем разделе "Резидентные программы".

ФУНКЦИИ ПРЕРЫВАНИЯ DOS INT 21H

00	Завершение программы (аналогично INT 20H).
01	Ввод символа с клавиатуры с эхом на экран.
02	Вывод символа на экран.
06	Прямой ввод с клавиатуры и вывод на экран.
07	Ввод с клавиатуры без эха и без проверки Ctrl/Break.
08	Ввод с клавиатуры без эха с проверкой Ctrl/Break.
09	Вывод строки символов на экран (гл.8).
0A	Ввод с клавиатуры с буферизацией (гл.8).
0B	Проверка наличия ввода с клавиатуры.
0C	Очистка буфера ввода с клавиатуры и запрос на ввод.

ЭКРАННЫЕ И КЛАВИАТУРНЫЕ ОПЕРАЦИИ: РАСШИРЕННАЯ ВЕРСИЯ DOS

Существуют следующие стандартные файловые номера:

- 0 Ввод (обычно с клавиатуры) CON
- 1 Вывод (обычно на экран) CON
- 2 Вывод по ошибке (на экран) CON
- 3 Ввод/вывод на внешнее устройство AUX
- 4 Вывод на печать LPT1 или PRN

Прерывание DOS для ввода/вывода - INT 21H, необходимая функция запрашивается через регистр AH: шест.3F - для ввода, шест.40 - для вывода. В регистр CX заносится число байт для ввода/вывода, а в регистр DX - адрес области ввода/вывода.

В результате успешного выполнения операции ввода/вывода очищается флаг переноса (CF) и в регистр AX устанавливается действительное число байт, участвующих в операции. При неуспешной операции устанавливается флаг CF, а код ошибки (в данном случае 6) заносится в регистр AX. Поскольку регистр AX может содержать как длину данных, так и код ошибки, то единственный способ определить наличие ошибки - проверить флаг CF, хотя ошибки чтения с клавиатуры и вывода на экран - явления крайне редкие. Аналогичным образом используются файловые номера для дисковых файлов, здесь ошибки ввода/вывода встречаются чаще. Можно использовать эти функции для перенаправления ввода-вывода на другие устройства, однако эта особенность здесь не рассматривается.

ВЫВОД НА ЭКРАН: РАСШИРЕННАЯ ВЕРСИЯ DOS

Следующие команды иллюстрируют операцию вывода на экран в расширенной версии DOS:

```
DISAREA DB 20 DUP(' ');Область данных
...
MOV AH,40H ;Запрос на вывод
MOV BX,01 ;Выводное устройство
MOV CX,20 ;Максимальное число байт
LEA DX,DISAREA ;Адрес области данных
INT 21H ;Вызов DOS
```

Команда LEA загружает в регистр DX адрес DISAREA для возможности DOS локализовать информацию, предназначенную для вывода. В результате успешной операции флаг переноса очищается (это можно проверить), а в регистре AX устанавливается число выведенных символов. Ошибка в данной операции может произойти, если установлен неправильный файловый номер. В этом случае будет установлен флаг CF и код ошибки (в данном случае 6) в регистре AX. Поскольку регистр AX может содержать или длину, или код ошибки, то единственный способ определить состояние ошибки - проверить флаг CF.

ВВОД С КЛАВИАТУРЫ: РАСШИРЕННЫЙ DOS

Ниже приведены команды, иллюстрирующие использование функции ввода с клавиатуры в расширенной версии DOS:

```
INAREA DB 20 DUP(' ');Область ввода
MOV AH,3FH ;Запрос на ввод
MOV BX,00 ;Номер для клавиатуры
MOV CX,20 ;Максимум байт для ввода
LEA DX,INAREA ;Адрес области ввода
INT 21H ;Вызов DOS
```

Команда LEA загружает относительный адрес INAREA в регистр DX. Команда INT ожидает, пока пользователь не введет символы с клавиатуры, но не проверяет превышает ли число введенных символов максимальное значение в регистре CX (20 в приведенном примере). Нажатие клавиши Return (код шест. 0D) указывает на завершение ввода. Например, после ввода текста "PC Users Group" INAREA будет содержать:

PC Users Group, шест.0D, шест.0A

После введенного текста непосредственно следует символ возврата каретки (шест.0D), который был введен, и символ конца строки (шест.0A), который не был введен. В силу данной особенности максимальное число символов и размер области ввода должны предусматривать место для двух символов. Если будет введено символов меньше максимального значения, то область памяти за введенными символами сохранит прежнее значение.

В результате успешной операции будет очищен флаг CF (что можно проверить) и в регистре AX будет установлено число байт, введенных с клавиатуры. В предыдущем примере это число будет равно 14 плюс 2 для перевода каретки и конца строки, т.е. 16. Соответствующим образом программа может определить действительное число введенных символов. Хотя данное свойство весьма тривиально для ответов типа YES или NO, оно может быть полезно для ответов с переменной длиной, таких, например, как имена.

Ошибка ввода может возникнуть, если определен неправильный номер файла. В этом случае будет установлен флаг CF и в регистр AX будет помещен код ошибки (6 в данном случае). Так как регистр AX может содержать или длину введенных данных, или код ошибки, то единственный способ определения наличия ошибки - проверка флага CF.

Если вводить текст, который превышает максимальную длину, установленную в регистре CX, то будут приниматься все символы. Рассмотрим ситуацию, когда регистр CX содержит 08, а пользователь введет символы "PC Exchange". В результате первые восемь символов "PC Excha" попадут в область ввода без кодов возврата каретки и конца строки. В регистре AX будет установлена длина 08. Следующая команда INT будет принимать данные не с клавиатуры, а из собственного буфера, поскольку там еще остались предыдущие данные. Таким образом, в область ввода будут приняты символы "nge", символ перевода каретки и символ новой строки, в регистре AX будет установлено значение 05. Обе операции ввода являются вполне нормальными и флаг CF будет очищен.

```
Первый INT: PC Excha AX = 08
Второй INT: nge,0D,0A AX = 05
```

Программа может определить факт ввода законченного текста, если а) в регистре AX получится значение меньше, чем в регистре CX или б) если содержимые AX и CX равны, но последние два символа в области ввода - 0D и 0A.

Встроенные в DOS проверки по функции 0AH для ввода с клавиатуры имеют более мощные средства. Их выбор для использования в программах является предпочтительным.

Один из способов получения более эффективного вывода на экран - использование управляющих символов возврата каретки, перевода строки и табуляции:

	Десятичные ASCII	Шестнадцатиричные
CR	13	0DH
LF	10	0AH
TAB	09	09H

Использование директивы EQU для определения кодов делает программу более понятной:

```
CR EQU 13 ;или EQU 0DH
LF EQU 10 ;или EQU 0AH
TAB EQU 09 ;или EQU 09H
MESSAGE DB TAB, 'PC Users Group Annual' DB 'Report', CR, LF
```

Лекция 6

Система прерываний микропроцессора

Система прерываний. Аппаратная организация прерываний. Обработка прерываний. Программные прерывания.

СИСТЕМА ПРЕРЫВАНИЙ

Система прерываний любого компьютера является его важнейшей частью, позволяющей быстро реагировать на события, обработка которых должна выполняться немедленно: сигналы от машинных таймеров, нажатия клавиш клавиатуры или мыши, сбоя памяти и пр. Рассмотрим в общих чертах компоненты этой системы.

Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера или подключенных к нему, поступают в процессор не непосредственно, а через два контроллера прерываний, один из которых называется ведущим, а второй - ведомым (рис. 1.11)

Прерывание представляет собой операцию, которая приостанавливает выполнение программ для специальных системных действий. Необходимость прерываний обусловлено двумя основными причинами: преднамеренный запрос таких действий, как операции ввода-вывода на различные устройства и непредвиденные программные ошибки (например, переполнение при делении).

Система BIOS (Basic Input/Output System) находится в ROM и управляет всеми прерываниями в системе. В предыдущих главах уже использовались некоторые прерывания для вывода на экран дисковых операций ввода-вывода и печати. В этой главе описаны различные BIOS- и DOS-прерывания, резидентные программы и команды IN и OUT.

В компьютерах IBM PC ROM находится по адресу FFFF0H. При включении компьютера процессор устанавливает состояние сброса, выполняет контроль четности, устанавливает в регистре CS значение FFFFH, а в регистре IP - нуль. Первая выполняемая команда поэтому находится по адресу FFFF:0 или FFFF0, что является точкой входа в BIOS. BIOS проверяет различные порты компьютера для определения и инициализации подключенных устройств. Затем BIOS создает в начале памяти (по адресу 0) таблицу прерываний, которая содержит адреса обработчиков прерываний, и выполняет две операции INT 11H (запрос списка присоединенного оборудования) и INT 12H (запрос размера физической памяти).

Следующим шагом BIOS определяет имеется ли на диске или дискете операционная система DOS. Если обнаружена системная дискета, то BIOS выполняет прерывание INT 19H для доступа к первому сектору диска, содержащему блок начальной загрузки. Этот блок представляет собой программу, которая считывает системные файлы COMMAND.COM с диска в память.

АППАРАТНАЯ ОРГАНИЗАЦИЯ ПРЕРЫВАНИЙ.



Два контроллера используются для увеличения допустимого количества внешних устройств. Дело в том, что каждый контроллер прерываний может обслуживать сигналы лишь от 8 устройств. Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них всевозможную структуру. В современных машинах устанавливают два контроллера, увеличивая тем самым возможное число входных устройств до 15 (7 у ведущего и 8 у ведомого контроллеров).

К входным выводам IRQ1...IRQ7 и IRQ8...IRQ15 (IRQ - это сокращение от Interrupt Request, запрос прерывания) подключаются выходы устройств, на которых возникают сигналы прерываний. Выход ведущего контроллера подключается к входу INT микропроцессора, а выход ведомого - к входу IRQ2 ведущего. Основная функция контроллеров - передача сигналов запросов прерываний от внешних устройств на единственный вход прерываний микропроцессора. При этом, кроме сигнала INT, контроллеры передают в микропроцессор по линиям данных номер вектора, который образуется в контроллере путем сложения базового номера, записанного в одном из его регистров, с номером входной линии, по которой поступил запрос прерывания. Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Для ведущего контроллера базовый вектор всегда равен 8, для ведомого - 70h. Таким образом, номера векторов, закрепленных за аппаратными прерываниями, лежат в диапазонах 8h...Fh и 70h...77h.

Очевидно, что номера векторов аппаратных прерываний однозначно связаны с номерами линий, или уровнями IRQ, а через них - с конкретными устройствами компьютера.

На рис. 1.11 указаны некоторые из стандартных устройств компьютера, работающих в режиме прерываний. Процессор, получив сигнал прерывания, выполняет последовательность стандартных действий, обычно называемых процедурой прерывания. Подчеркнем, что здесь идет речь лишь о реакции самого процессора на сигналы прерываний, а не об алгоритмах обработки прерываний, предусматриваемых пользователем в программах обработки прерываний.

Объекты вычислительной системы, принимающие участие в процедуре прерывания, и их взаимодействие показаны на рис. 1.12.

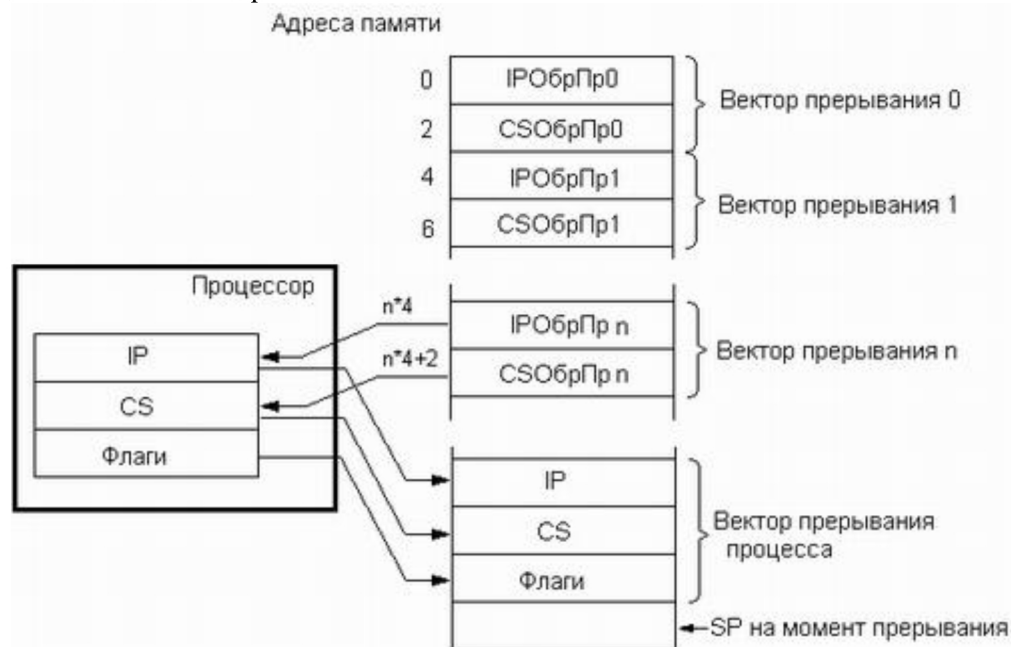


Рис. 1.12. Процедура обслуживания прерывания.

ОБРАБОТКА ПРЕРЫВАНИЙ

Самое начало оперативной памяти от адреса 0000h до 03FFh отводится под векторы прерываний - четырехбайтовые области, в которых хранятся адреса обработчиков прерываний (ОбрПр на рис. 1.12). В два старших байта каждого вектора записывается сегментный адрес обработчика, в два младших - смещение (относительный адрес) точки входа в обработчик. Векторы, как и соответствующие им прерывания, имеют номера, причем вектор с номером 0 располагается, начиная с адреса 0, вектор 1 - с адреса 4, вектор 2 - с адреса 8 и т.д. Вектор с номером n занимает, таким образом, байты памяти от $n*4$ до $n*4+3$. Всего в выделенной под векторы области памяти помещается 256 векторов.

Получив сигнал на выполнение процедуры прерывания с определенным номером, процессор сохраняет в стеке выполняемой программы текущее содержимое трех регистров процессора: регистра флагов, CS и IP. Два последних числа образуют полный адрес возврата в прерванную программу. Далее процессор загружает CS и IP из соответствующего вектора прерываний, осуществляя, тем самым, переход на обработчик прерывания, связанный с этим вектором.

Обработчик прерываний всегда заканчивается командой `iret` (interrupt return, возврат из прерывания), выполняющей обратные действия - извлечение из стека сохраненных там слов и помещение их назад в регистры IP и CS, а также в регистр флагов. Это приводит к возврату в основную программу в ту самую точку, где она была прервана.

В действительности запросы на обработку прерываний могут иметь различную природу. Помимо описанных выше аппаратных прерывания от периферийных устройств, называемых часто внешними, имеются еще два типа прерываний: внутренние и программные.

Внутренние прерывания возбуждаются цепями самого процессора при возникновении одной из специально оговоренных ситуаций, например, при выполнении операции деления на ноль или при попытке выполнить несуществующую команду. За каждым из таких прерываний закреплен определенный вектор, номер которого известен процессору. Например, за делением на 0 закреплен вектор 0, а за неправильной командой - вектор 6. Если процессор сталкивается с одной из таких ситуаций, он выполняет описанную выше процедуру прерывания, используя закрепленный за этой ситуацией вектор прерывания.

ПРОГРАММНЫЕ ПРЕРЫВАНИЯ.

Наконец, еще одним чрезвычайно важным типом прерываний являются программные прерывания. Они вызываются командой `hit` с числовым аргументом, который рассматривается процессором, как номер вектора прерывания. Если в программе встречается, например, команда

int 13h

то процессор выполняет ту же процедуру прерывания, используя в качестве номера вектора операнд команды int. Программные прерывания применяются в первую очередь для вызова системных обслуживающих программ - функций DOS и BIOS. С командой int 2In вызова DOS мы уже сталкивались в примере 1-1 и будем встречаться еще многократно. В дальнейшем будут также приведены примеры использования команды int для вызова прикладных обработчиков программных прерываний.

Важно подчеркнуть, что описанные действия процессора выполняются совершенно одинаково для всех видов прерываний - внутренних, аппаратных и программных, хотя причины, возбуждающие процедуру прерывания, имеют принципиально разную природу.

Большая часть векторов прерываний зарезервирована для выполнения определенных действий; часть из них автоматически заполняется адресами системных программ при загрузке системы. Приведем краткую выдержку из таблицы векторов, позволяющую продемонстрировать разнообразие ее состава:

00h	внутреннее прерывание, деление на 0;
01h	внутреннее прерывание, пошаговое выполнение (при TF=1);
02h	немаскируемое прерывание (вывод NMI процессора);
08h	аппаратное прерывание от системного таймера;
09h	аппаратное прерывание от клавиатуры;
0Eh	аппаратное прерывание от гибкого диска;
10h	программное прерывание, программы BIOS управления видеосистемой;
13h	программное прерывание, программы BIOS управления дисками;
16h	программное прерывание, программы BIOS управления клавиатурой;
1Dh	не вектор, адрес таблицы видеопараметров, используемой BIOS;
1Eh	не вектор, адрес таблицы параметров дискеты, используемой BIOS;
21h	программное прерывание, диспетчер функций DOS;
22h	программное прерывание, адрес перехода при завершении процесса, используемый DOS;
23h	программное прерывание, обработчик прерываний по <Ctrl>/C, используемый DOS;
25h	программное прерывание, абсолютное чтение диска (функция DOS);
26h	программное прерывание, абсолютная запись на диск (функция DOS);
60h...66h	зарезервировано для программных прерываний пользователя;
68h...6Fh	программные прерывания, свободные векторы;
70h	аппаратное прерывание от часов реального времени (с питанием от аккумулятора);
76h	аппаратное прерывание от жесткого диска;

Как видно из таблицы, векторы прерываний можно условно разбить на следующие группы:

- векторы внутренних прерываний процессора (01h, 02h и др.);
- векторы аппаратных прерываний (08h...0Fh и 70h...77h);
- программы BIOS обслуживания аппаратуры компьютера (10h, 13h, 16h и др.);
- программы DOS (21h, 22h, 23h и др.);
- адреса системных таблиц BIOS (1Dh, 1Eh и др.).

Системные программы, адреса которых хранятся в векторах прерываний, в большинстве своем являются всего лишь диспетчерами, открывающими доступ к большим группам программ, реализующих системные функции. Так, видеодрайвер BIOS (вектор 10h) включает программы смены видеорежима, управления курсором, задания цветовой палитры, загрузки шрифтов и многие другие. Особенно характерен в этом отношении вектор 21h, через который осуществляется вызов практически всех функций DOS: ввода с клавиатуры и вывода на экран, обслуживания файлов, каталогов и дисков, управления памятью и процессами, службы времени и т.д. Для вызова требуемой функции надо не только выполнить команду int с соответствующим номером, но и указать системе в одном из регистров (для этой цели всегда используется регистр AH) номер вызываемой функции. Иногда для "многофункциональных" функций приходится указывать еще и номер подфункции (в регистре AL).

ПРЕРЫВАНИЯ BIOS

INT 05H. П е ч а т ь э к р а н а. Выполняет вывод содержимого экрана на печатающее устройство. Команда INT 05H выполняет данную операцию из программы, а нажатие клавишей Ctrl/PrtSc – с клавиатуры. Операция запрещает прерывания и сохраняет позицию курсора.

INT 10H. У п р а в л е н и е д и с п л е е м. Обеспечивает экранные и клавиатурные операции, детально описанные в гл.9.

INT 11H. З а п р о с с п и с к а п р и с о е д и н е н н о г о о б о р у д о в а н и я. Определяет наличие различных устройств в системе, результирующее значение возвращает в регистре AX.

INT 12H З а п р о с р а з м е р а ф и з и ч е с к о й п а м я т и. Возвращает в регистре AX размер памяти в килобайтах

INT 13H. Д и с к о в ы е о п е р а ц и и в в о д а – в ы в о д а.

INT 16H. В в о д с к л а в и а т у р ы. Обеспечивает три типа команд ввода с клавиатуры, подробно описанные в гл.9.

INT 17H. В ы в о д н а п р и н т е р. Обеспечивает вывод данных на печатающее устройство. Подробно рассмотрено в гл.19.

ПРЕРЫВАНИЯ DOS

INT 20H. З а в е р ш е н и е п р о г р а м м ы. Запрос завершает выполнение программы и передает управление в DOS. Данный запрос обычно находится в основной процедуре.

INT 21H. З а п р о с ф у н к ц и й DOS. Основная операция DOS, вызывающая определенную функцию в соответствии с кодом в регистре AH. Назначение функций DOS описано в следующем разделе.

INT 27H.З а в е р ш е н и е п р о г р а м м ы, о с т а в л я ю щ е е е е р е з е д е н т н о й. Позволяет сохранить COM-программу в памяти. Подробно данная операция рассмотрена в последующем разделе "Резидентные программы".

ФУНКЦИИ ПРЕРЫВАНИЯ DOS INT 21H

- 00 Завершение программы (аналогично INT 20H).
- 01 Ввод символа с клавиатуры с эхом на экран.
- 02 Вывод символа на экран.
- 06 Прямой ввод с клавиатуры и вывод на экран.
- 07 Ввод с клавиатуры без эха и без проверки Ctrl/Break.
- 08 Ввод с клавиатуры без эха с проверкой Ctrl/Break.
- 09 Вывод строки символов на экран (гл.8).
- 0A Ввод с клавиатуры с буферизацией (гл.8).
- 0B Проверка наличия ввода с клавиатуры.
- 0C Очистка буфера ввода с клавиатуры и запрос на ввод.

ЭКРАННЫЕ И КЛАВИАТУРНЫЕ ОПЕРАЦИИ: РАСШИРЕННАЯ ВЕРСИЯ DOS

Существуют следующие стандартные файловые номера:

- 0 Ввод (обычно с клавиатуры) CON
- 1 Вывод (обычно на экран) CON
- 2 Вывод по ошибке (на экран) CON

- 3 Ввод/вывод на внешнее устройство AUX
- 4 Вывод на печать LPT1 или PRN

Прерывание DOS для ввода/вывода - INT 21H, необходимая функция запрашивается через регистр AH: шест.3F - для ввода, шест.40 - для вывода. В регистр CX заносится число байт для ввода/вывода, а в регистр DX - адрес области ввода/вывода.

В результате успешного выполнения операции ввода/вывода очищается флаг переноса (CF) и в регистр AX устанавливается действительное число байт, участвующих в операции. При неуспешной операции устанавливается флаг CF, а код ошибки (в данном случае 6) заносится в регистр AX. Поскольку регистр AX может содержать как длину данных, так и код ошибки, то единственный способ определить наличие ошибки - проверить флаг CF, хотя ошибки чтения с клавиатуры и вывода на экран - явления крайне редкие. Аналогичным образом используются файловые номера для дисковых файлов, здесь ошибки ввода/вывода встречаются чаще. Можно использовать эти функции для перенаправления ввода-вывода на другие устройства, однако эта особенность здесь не рассматривается.

ВЫВОД НА ЭКРАН: РАСШИРЕННАЯ ВЕРСИЯ DOS

Следующие команды иллюстрируют операцию вывода на экран в расширенной версии DOS:

```
DISAREA DB 20 DUP(' ') ;Область данных
...
MOV AH,40H ;Запрос на вывод
MOV BX,01 ;Выводное устройство
MOV CX,20 ;Максимальное число байт
LEA DX,DISAREA ;Адрес области данных
INT 21H ;Вызов DOS
```

Команда LEA загружает в регистр DX адрес DISAREA для возможности DOS локализовать информацию, предназначенную для вывода. В результате успешной операции флаг переноса очищается (это можно проверить), а в регистре AX устанавливается число выведенных символов. Ошибка в данной операции может произойти, если установлен неправильный файловый номер. В этом случае будет установлен флаг CF и код ошибки (в данном случае 6) в регистре AX. Поскольку регистр AX может содержать или длину, или код ошибки, то единственный способ определить состояние ошибки - проверить флаг CF.

ВВОД С КЛАВИАТУРЫ: РАСШИРЕННЫЙ DOS

Ниже приведены команды, иллюстрирующие использование функции ввода с клавиатуры в расширенной версии DOS:

```
INAREA DB 20 DUP(' ') ;Область ввода
MOV AH,3FH ;Запрос на ввод
MOV BX,00 ;Номер для клавиатуры
MOV CX,20 ;Максимум байт для ввода
LEA DX,INAREA ;Адрес области ввода
INT 21H ;Вызов DOS
```

Команда LEA загружает относительный адрес INAREA в регистр DX. Команда INT ожидает, пока пользователь не введет символы с клавиатуры, но не проверяет превышает ли число введенных символов максимальное значение в регистре CX (20 в приведенном примере). Нажатие клавиши Return (код шест. 0D) указывает на завершение ввода. Например, после ввода текста "PC Users Group" INAREA будет содержать:

PC Users Group, шест.0D, шест.0A

После введенного текста непосредственно следует символ возврата каретки (шест.0D), который был введен, и символ конца строки (шест.0A), который не был введен. В силу данной особенности максимальное число символов и размер области ввода должны предусматривать место для двух символов. Если будет введено символов меньше максимального значения, то область памяти за введенными символами сохранит прежнее значение.

В результате успешной операции будет очищен флаг CF (что можно проверить) и в регистре AX будет установлено число байт, введенных с клавиатуры. В предыдущем примере это число будет равно 14 плюс 2 для перевода каретки и конца строки, т.е. 16. Соответствующим образом программа может определить действительное число введенных символов. Хотя данное свойство весьма тривиально для ответов типа YES или NO, оно может быть полезно для ответов с переменной длиной, таких, например, как имена.

Ошибка ввода может возникнуть, если определен неправильный номер файла. В этом случае будет установлен флаг CF и в регистр AX будет помещен код ошибки (6 в данном случае). Так как регистр AX может содержать или длину введенных данных, или код ошибки, то единственный способ определения наличия ошибки - проверка флага CF.

Если вводить текст, который превышает максимальную длину, установленную в регистре CX, то будут приниматься все символы. Рассмотрим ситуацию, когда регистр CX содержит 08, а пользователь введет символы "PC Exchange". В результате первые восемь символов "PC Excha" попадут в область ввода без кодов возврата каретки и конца строки. В регистре AX будет установлена длина 08. Следующая команда INT будет принимать данные не с клавиатуры, а из собственного буфера, поскольку там еще остались предыдущие данные. Таким образом, в область ввода будут приняты символы "nge", символ перевода каретки и символ новой строки, в регистре AX будет установлено значение 05. Обе операции ввода являются вполне нормальными и флаг CF будет очищен.

```
Первый INT:  PC Excha    AX = 08
Второй INT:  nge,0D,0A  AX = 05
```

Программа может определить факт ввода законченного текста, если а) в регистре AX получится значение меньше, чем в регистре CX или б) если содержимые AX и CX равны, но последние два символа в области ввода - 0D и 0A.

Встроенные в DOS проверки по функции 0AH для ввода с клавиатуры имеют более мощные средства. Их выбор для использования в программах является предпочтительным.

Один из способов получения более эффективного вывода на экран - использование управляющих символов возврата каретки, перевода строки и табуляции:

Десятичные ASCII Шестнадцатиричные

```
CR      13      0DH
LF      10      0AH
TAB     09      09H
```

Использование директивы EQU для определения кодов делает программу более понятной:

```
CR      EQU     13      ;или EQU 0DH
LF      EQU     10      ;или EQU 0AH
TAB     EQU     09      ;или EQU 09H
MESSAGE DB TAB, 'PC Users Group Annual' DB 'Report', CR, LF
```

Лекция 7

Основные понятия операционной системы Концепции ОС.

Операционная система *Утилиты* *Системы программирования*. *Средства сетевого доступа*

Основные функции ОС *Системные вызовы* *Прерывания* *Исключительные ситуации* *Файлы* *Процессы, нити*

Монолитное ядро *Слоеные системы (Layered systems)* *Виртуальные машины* *Микроядерная архитектура*. *Смешанные системы*

Управление ресурсами *Многозадачность*. *Многопользовательский режим*. *Параллельная обработка* *Сетевая ОС*.

Особенности областей использования *Особенности методов построения*

Понятие процесса *Состояния процесса* *Операции над процессами* *Переключение контекста*

ОПЕРАЦИОННАЯ СИСТЕМА

– совокупность программных средств, обеспечивающая управление аппаратной частью компьютера и прикладными программами, а также их взаимодействием между собой и пользователем.

Операционная система предназначена для управления выполнением пользовательских программ, планирования и управления вычислительными ресурсами ЭВМ.

Операционная система, с одной стороны, выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами, с другой стороны, предназначена для эффективного использования ресурсов вычислительной системы и организации надежных вычислений.

Системы управления файлами предназначены для организации более удобного доступа к данным, организованным как файлы.

Вместо низкоуровневого доступа к данным с указанием конкретных физических адресов система управления файлами позволяет использовать логический доступ с указанием имени файла.

Любая система управления файлами не существует сама по себе - она разработана для работы в конкретной ОС и с конкретной файловой системой. То есть можно было бы систему управления файлами отнести к ОС.

Но в связи с тем, что

1) ряд ОС позволяет работать с несколькими файловыми системами (либо с одной из нескольких, либо сразу с несколькими одновременно); а дополнительную файловую систему можно установить (т.е. они самостоятельны)

2) простейшие ОС могут работать и без файловых систем;

системы управления файлами выделяются в отдельную группу системных программ.

Заметим, что часто в специальной литературе системы управления файлами относят все-таки к операционным системам.

СЕРВИСНОЕ программное обеспечение - программы и программные комплексы, которые расширяют возможности базового программного обеспечения и организуют более удобную среду работы пользователя.

Это набор сервисных, дополнительно устанавливаемых программ, которые можно классифицировать по функциональному признаку следующим образом:

1. драйверы специфических и специальных устройств (те, которые не поставляются в составе ОС).
2. программы диагностики работоспособности компьютера;
3. антивирусные программы, обеспечивающие защиту компьютера, обнаружение и восстановление зараженных файлов;
4. программы обслуживания дисков, обеспечивающие проверку качества поверхности магнитного диска, контроль сохранности файловой системы на логическом и физической уровнях, сжатие дисков, создание страховых копий дисков, резервирование данных на внешних носителях и др.;
5. программы архивирования данных, которые обеспечивают процесс сжатия информации в файлах с целью уменьшения объема памяти для ее хранения;
6. программы обслуживания сети.

Эти программы часто называются утилитами. (Заметим, что к антивирусным средствам этот термин обычно не применяется)

УТИЛИТЫ

- программы, служащие для выполнения вспомогательных операций обработки данных или обслуживания компьютеров (диагностики, тестирования аппаратных и программных средств, оптимизации использования дискового пространства, восстановления разрушенной на магнитном диске информации и т.п.).

Отдельно вспомним о такой группе системного ПО как

СИСТЕМЫ ПРОГРАММИРОВАНИЯ.

Это набор специализированных программных продуктов, которые являются инструментальными средствами разработчика. Программные продукты данного класса поддерживают все этапы процесса программирования, отладки и тестирования создаваемых программ.

Система программирования включает следующие программные компоненты:

- редактор текста;
- транслятор с соответствующего языка;
- компоновщик (редактор связей);
- отладчик;
- библиотеки подпрограмм.

Заметим, что любая система программирования может работать только в соответствующей ОС, под которую она и создана, однако при этом она может позволять разрабатывать программное обеспечение и под другие ОС.

Редактор текста - это программа для ввода и модификации текста.

Трансляторы предназначены для преобразования программ, написанных на языках программирования, в программы на машинном языке. Программа, подготовленная на каком-либо языке программирования, называется исходным модулем. В качестве входной информации трансляторы применяют исходные модули и формируют в результате своей работы объектные модули, являющиеся входной информацией для редактора связей. Объектный модуль содержит текст программы на машинном языке и дополнительную информацию, обеспечивающую настройку модуля по месту его загрузки и объединение этого модуля с другими независимо оттранслированными модулями в единую программу.

Трансляторы делятся на два класса: **компиляторы** и **интерпретаторы**. Компиляторы переводят весь исходный модуль на машинный язык. Интерпретатор последовательно переводит на машинный язык и выполняет операторы исходного модуля

(У интерпретаторов два основных недостатка. Первый - низкая скорость работы интерпретируемых программ.)

Преимущество интерпретатора перед компилятором состоит в том, что программа пользователя имеет одно представление - в виде текста. При компиляции одна и та же программа имеет несколько представлений - в виде текста и в виде выполняемого файла.

Компоновщик, или редактор связей - системная обрабатывающая программа, редактирующая и объединяющая объектные (ранее оттранслированные) модули в единые загрузочные, готовые к выполнению программные модули. Загрузочный модуль может быть помещен ОС в основную память и выполнен.

Отладчик позволяет управлять процессом исполнения программы, является инструментом для поиска и исправления ошибок в программе. Базовый набор функций отладчика включает:

- · пошаговое выполнение программы (режим трассировки) с отображением результатов,
- · остановка в заранее определенных точках,
- · возможность остановки в некотором месте программы при выполнении некоторого условия;
- · изображение и изменение значений переменных.

Загрузчик - системная обрабатывающая программа, объединяющая основные функции редактора связей и программы выборки в одном пункте задания. Загрузчик помещает находящиеся в его входном наборе данных объектные и загрузочные модули в оперативную память, объединяет их в единую программу, корректирует перемещаемые адресные константы с учетом фактического адреса загрузки и передает управление в точку входа созданной программы.

СРЕДСТВА СЕТЕВОГО ДОСТУПА

обеспечивают обработку, передачу и хранение данных в сети.

Заметим, что чаще говорят о сетевых операционных системах, которые предоставляют пользователям различные виды сетевых служб (управление файлами, электронная почта, процессы управления сетью и др.)

Ключом к использованию этих ресурсов является сервер, специальная программа на компьютере, подключенному к сети, которая принимает запросы (или команды) и посылает ответы автоматически.

Программы, предназначенные для подачи запросов серверу, называются программами-клиентами. Сервер предназначен для их обслуживания. Клиент посылает запросы пользователя на сервер, используя стандартизированный формат, называемый протоколом. Ответ сервера содержит информацию, представленную в виде файла, содержащего данные того или иного формата.

Постоянно ведется разработка все новых программ-клиентов, предлагающих более удобные способы взаимодействия с сервером.

Таким образом, в системном ПО мы выделили пять групп системных программ:

1. операционные системы;
2. интерфейсные оболочки для взаимодействия пользователя с ОС (операционная оболочка) и программные среды;
3. системы управления файлами;
4. системы программирования;
5. утилиты;
6. средства сетевого доступа.

Обратим внимание на то, что в ходе развития компьютерных систем наиболее используемые прикладные программы могут быть перенесены на уровень системных, что позволяет использовать их в различных приложениях. Например, средства управления диалоговым взаимодействием с пользователем в системных оболочках (типа Windows). С другой стороны, наиболее распространенные и критические по времени системные функции были частично или полностью реализованы аппаратно. Например, средства управления многопрограммным защищенным режимом и средства управления мультимедиа-устройствами в процессорах фирмы Intel.

Требования к системному программному обеспечению

Системные программы должны удовлетворять следующим требованиям:

- прозрачность работы; · гарантированная надежность выполнения в соответствии со спецификациями;

Функциональные требования называются спецификациями

- максимальная скорость выполнения; · минимальные затраты на хранение машинных кодов; · поддержка стандартных средств связи с прикладными программами.

Эффективность системных программ зависит от времени их создания и надежности исполняемого кода. Требование эффективности системных программ вызывает необходимость использования специальных языков: · машинно-ориентированных типа языка Assembler и высокого уровня типа С или С++.

К типам данных этих языков отнесены указатели на данные различных типов или адреса данных и программных объектов.

Работа с большинством пакетов для разработки системного программного обеспечения предполагает знание и использование ассемблера для создания модулей и ассемблерных вставок.

2.1. Что такое операционная система.

ПОНЯТИЕ – ОПЕРАЦИОННАЯ СИСТЕМА.

Операционная система (ОС) - это программа, которая обеспечивает возможность рационального использования оборудования компьютера удобным для пользователя образом.

Операционная система является фундаментальным компонентом системного программного обеспечения. Большинство пользователей имеет свой опыт эксплуатации операционных систем, но, тем не менее, затруднятся дать точное определение. Давайте кратко рассмотрим основные точки зрения.

Операционная система как виртуальная машина

Архитектура большинства компьютеров на уровне машинных команд очень неудобна для ее использования прикладными программами. Например, работа с диском предполагает знакомство с внутренним устройством его электронного компонента - контроллера для ввода команд вращения диска, поиска и форматирования дорожек, чтения и записи секторов и т.д. Ясно, что средний программист не в состоянии учитывать все особенности работы оборудования (в современной терминологии заниматься разработкой драйверов устройств), а должен иметь простую высокоуровневую абстракцию, скажем, представляя информационное пространство диска как набор файлов. Файл можно открывать для чтения или записи, использовать для получения или сброса информации, а потом закрывать. Это концептуально проще, чем заботиться о деталях перемещения головок дисков или организации работы мотора. Аналогичным образом, с помощью простых и ясных абстракций, скрываются от программиста все ненужные ему подробности организации прерываний, работы таймера, управления памятью и т.д. Более того, на современных вычислительных комплексах может быть создана иллюзия неограниченного размера операционной памяти и числа процессоров. Всем этим занимается операционная система. Таким образом, операционная система представляется пользователю виртуальной машиной, с которой проще иметь дело, чем непосредственно с оборудованием компьютера.

Операционная система как менеджер ресурсов

Операционная система предназначена для управления всеми частями весьма сложной архитектуры компьютера. Представим для примера, что случится, если несколько программ, работающих на одном компьютере, будут пытаться одновременно осуществлять вывод на принтер. Мы получили бы неупорядоченную смесь строчек и страниц, выведенных различными программами. Операционная система предотвращает хаос такого рода за счет буферизации информации, предназначенной для печати, на диске и организации очереди на печать. Для многопользовательских компьютеров, необходимость управления ресурсами и их защиты еще более очевидна. Следовательно, операционная система как менеджер ресурсов, осуществляет упорядоченное и контролируемое распределение процессоров, памяти и других ресурсов между различными программами, их использующими.

Операционная система как защитник пользователей и программ

Если вычислительная система допускает совместную работу нескольких пользователей, то возникает проблема организации их безопасной деятельности. Необходимо обеспечить сохранность информации на диске, чтобы никто не мог удалить или повредить чужие файлы. Нельзя разрешить программам одних пользователей произвольно вмешиваться в работу программ других пользователей. Нужно пресекать попытки несанкционированного использования вычислительной системы. Всю эту деятельность осуществляет

операционная система как организатор безопасной работы пользователей и их программ. С такой точки зрения операционная система выглядит системой безопасности в государстве, на которую возложены полицейские и контрразведывательные функции.

Операционная система как постоянно функционирующее ядро

Наконец, можно дать и такое определение: операционная система это программа, постоянно работающая на компьютере и взаимодействующая со всеми прикладными программами. Казалось бы, это абсолютно правильное определение, но, как мы увидим дальше, во многих современных операционных системах постоянно работает на компьютере лишь часть операционной системы, которую принято называть ее ядром.

Как видим, существует много точек зрения на то, что такое операционная система. Не существует ее адекватного строгого определения. Нам проще сказать, не что есть операционная система, а для чего она нужна, и что она делает. Для выяснения этого вопроса кратко рассмотрим историю развития вычислительных систем.

2.2. Краткая история эволюции вычислительных систем

Мы будем рассматривать историю развития именно вычислительных, а не операционных систем, потому что hardware и программное обеспечение эволюционировали совместно, оказывая взаимное влияние друг на друга. Появление новых технических возможностей приводило к прорыву в области создания удобных, эффективных и безопасных программ, а новые идеи в программной области стимулировали поиски новых технических решений. Именно эти критерии удобство, эффективность и безопасность играли роль факторов естественного отбора при эволюции вычислительных систем.

Первый период (1945-1955). Ламповые машины. Операционные системы отсутствовали.

Мы начнем исследование развития компьютерных комплексов с появления электронных вычислительных систем (опуская историю механических и электромеханических устройств).

Первые шаги по созданию электронных вычислительных машин были предприняты в конце второй мировой войны. В середине 40-х были созданы первые ламповые вычислительные устройства, и появился принцип программы, хранимой в памяти машины (John Von Neumann, июнь 1945г). В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не регулярное использование компьютеров в качестве инструмента решения каких-либо практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. За пультом мог находиться только один пользователь. Программа загружалась в память машины в лучшем случае с колоды перфокарт, а обычно с помощью панели переключателей. Вычислительная система выполняла одновременно только одну операцию (ввод-вывод, собственно вычисления, размышления программиста). Отладка программ велась с пульта управления с помощью изучения состояния памяти и регистров машины. В конце этого периода появляется первое системное программное обеспечение: в 1951-52 гг. возникают прообразы первых компиляторов с символических языков (Fortran и др.), а в 1954 г. Nat Rochester разрабатывает ассемблер для IBM-701. В целом первый период характеризуется крайне высокой стоимостью вычислительных систем, их малым количеством и низкой эффективностью использования.

Второй период (1955-Начало 60-х). Компьютеры на основе транзисторов. Пакетные операционные системы

С середины 50-х годов начался новый период в эволюции вычислительной техники, связанный с появлением новой технической базы - полупроводниковых элементов. Применение транзисторов вместо часто перегоравших электронных ламп привело к повышению надежности компьютеров. Теперь они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач. Снизилось потребление вычислительными машинами электроэнергии. Проще стали системы охлаждения. Размеры компьютеров уменьшились. Эксплуатация и обслуживание вычислительной техники подешевели. Началось использование ЭВМ коммерческими фирмами. Одновременно наблюдается бурное развитие алгоритмических языков (ALGOL-58, LISP, COBOL, ALGOL-60, PL-1 и т.д.). Появляются первые настоящие компиляторы, редакторы связей, библиотеки математических и служебных подпрограмм. Упрощается процесс программирования. Пропадает необходимость взваливать на одних и тех же людей весь процесс разработки и использования компьютеров. Именно в этот период происходит разделение персонала на программистов и операторов, специалистов по эксплуатации и разработчиков вычислительных машин.

Изменяется сам процесс прогона программ. Теперь пользователь приносит программу с входными данными в виде колоды перфокарт и указывает требуемые для нее ресурсы. Такая колода получает название задания. Оператор загружает задание в память машины и запускает его на исполнение. Полученные выходные

данные печатаются на принтере, и пользователь получает их обратно через некоторое (довольно большое) время.

Смена запрошенных ресурсов вызывает приостановку выполнения программ. В результате процессор часто простаивает. Для повышения эффективности использования компьютера задания с похожими требуемыми ресурсами начинают собирать вместе, создавая пакет заданий.

Появляются первые системы пакетной обработки, которые просто автоматизируют запуск одной программы из пакета за другой и, тем самым, увеличивают коэффициент загрузки процессора. При реализации систем пакетной обработки был разработан формализованный язык управления заданиями, с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом.

Третий период (Начало 60-х - 1980). Компьютеры на основе интегральных микросхем. Первые многозадачные ОС.

Следующий важный период развития вычислительных машин относится к началу 60-х - 1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Вычислительная техника становится более надежной и дешевой. Растет сложность и количество задач, решаемых компьютерами. Повышается производительность процессоров.

Повышению эффективности использования процессорного времени мешает низкая скорость механических устройств ввода-вывода (быстрый считыватель перфокарт мог обработать 1200 перфокарт в минуту, принтеры печатали до 600 строк в минуту). Вместо непосредственного чтения пакета заданий с перфокарт в память начинают использовать его предварительную запись сначала на магнитную ленту, а затем и на диск. Когда в процессе выполнения заданию требуется ввод данных, они читаются с диска. Точно так же выходная информация сначала копируется в системный буфер и записывается на ленту или диск, а реально печатается только после завершения задания. Вначале действительные операции ввода-вывода осуществлялись в режиме off-line, то есть с использованием других, более простых, отдельно стоящих компьютеров. В дальнейшем они начинают выполняться на том же компьютере, который производит вычисления, то есть в режиме on-line. Такой прием получает название spooling (сокращение от Simultaneous Peripheral Operation On Line) или подкачки-откачки данных. Введение техники подкачки-откачки в пакетные системы позволило совместить реальные операции ввода-вывода одного задания с выполнением другого задания, но потребовало появления аппарата прерываний для извещения процессора об окончании этих операций.

Магнитные ленты были устройствами последовательного доступа, то есть информация считывалась с них в том порядке, в каком была записана. Появление магнитного диска, для которого не важен порядок чтения информации, то есть устройства прямого доступа, привело к дальнейшему развитию вычислительных систем. При обработке пакета заданий на магнитной ленте очередность запуска заданий определялась порядком их ввода. При обработке пакета заданий на магнитном диске появляется возможность выбора очередного выполняемого задания. Пакетные системы начинают заниматься планированием заданий: в зависимости от наличия запрошенных ресурсов, срочности вычислений и т.д. на счет выбирается то или иное задание.

Дальнейшее повышение эффективности использования процессора было достигнуто с помощью мультипрограммирования. Идея мультипрограммирования заключается в следующем: пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при однопрограммном режиме, а выполняет другую программу. Когда операция ввода-вывода заканчивается, процессор возвращается к выполнению первой программы. Эта идея напоминает поведение преподавателя и студентов на экзамене. Пока один студент (программа) обдумывает ответ на поставленный вопрос (операция ввода-вывода), преподаватель (процессор) выслушивает ответ другого студента (вычисления). Естественно, что такая ситуация требует наличия в комнате нескольких студентов. Точно также мультипрограммирование требует наличия в памяти нескольких программ одновременно. При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом, и не должна влиять на выполнение другой программы. (Студенты сидят за отдельными столами и не подсказывают друг другу.)

Появление мультипрограммирования требует целой революции в строении вычислительной системы. Большую роль, здесь играет аппаратная поддержка, наиболее существенные особенности которой:

Реализация защитных механизмов. Программы не должны иметь самостоятельного доступа к распределению ресурсов, что приводит к появлению привилегированных и непривилегированных команд. Привилегированные команды, например команды ввода-вывода, могут исполняться только операционной системой. Говорят, что она работает в привилегированном режиме. Переход управления от прикладной программы к ОС сопровождается контролируемой сменой режима. Во-вторых, это защита памяти, позволяющая изолировать конкурирующие пользовательские программы друг от друга, а ОС от программ пользователей.

Наличие прерываний. Внешние прерывания оповещают ОС о том, что произошло асинхронное событие, например, завершилась операция ввода-вывода. Внутренние прерывания (сейчас их принято называть исключительными ситуациями) возникают, когда выполнение программы привело к ситуации, требующей вмешательства ОС, например, деление на ноль или попытка нарушения защиты.

Не менее важна в организации мультипрограммирования роль операционной системы. Наиболее существенные изменения состояли в следующем:

Интерфейс между прикладной программой и ОС был организован при помощи набора системных вызовов.

Организация очереди из заданий в памяти и выделение процессора одному из заданий потребовали планирования заданий.

Для переключения процессора с одного задания на другое возникла потребность в сохранении содержимого регистров и структур данных, необходимых для выполнения задания, иначе говоря, контекста, для обеспечения правильного продолжения вычислений.

Поскольку память является ограниченным ресурсом, оказались нужны стратегии управления памятью, то есть потребовалось упорядочить процессы размещения, замещения и выборки информации из памяти.

Так как программы могут пожелать произвести санкционированный обмен данными, стало необходимо их обеспечить средствами коммуникации. И, наконец,

Для корректного обмена данными необходимо предусмотреть координацию программами своих действий, т.е. средства синхронизации.

Мультипрограммные пакетные системы дают окружение, в котором различные системные ресурсы (например, процессор, память, периферийные устройства) используются эффективно. И все же пользователь не мог непосредственно взаимодействовать с заданием и должен был предусмотреть с помощью управляющих карт все возможные ситуации. Отладка программ по-прежнему занимала много времени и требовала изучения многостраничных распечаток содержимого памяти и регистров или использования отладочной печати.

Появление электроннолучевых дисплеев и переосмысление возможностей применения клавиатур поставили на очередь решение этой проблемы. Логическим расширением систем мультипрограммирования стали time-sharing системы или системы разделения времени **. В них процессор переключается между задачами не только на время операций ввода-вывода, но и просто по прошествии определенного интервала времени. Эти переключения происходят столь часто, что пользователи могут взаимодействовать со своими программами во время их выполнения, то есть интерактивно. В результате появляется возможность одновременной работы многих пользователей на одной компьютерной системе. У каждого пользователя для этого должна быть хотя бы одна программа в памяти. Чтобы уменьшить ограничения на количество работающих пользователей, была внедрена идея неполного нахождения исполняемой программы в оперативной памяти. Основная часть программы находится на диске и необходимый для ее дальнейшего выполнения кусок может быть легко загружен в оперативную память, а ненужный выкачан обратно на диск. Это реализуется с помощью механизма виртуальной памяти. Основным достоинством такого механизма является создание иллюзии неограниченной оперативной памяти ЭВМ.

В системах разделения времени пользователь получил возможность легко и эффективно вести отладку своей программы в интерактивном режиме, записывать информацию на диск, не используя перфокарты, а непосредственно с клавиатуры. Появление on-line файлов привело к необходимости разработки развитых файловых систем.

Параллельно внутренней эволюции вычислительных систем в этот период наблюдается и внешняя их эволюция. До начала этого периода вычислительные комплексы были, как правило, несовместимы. Каждый имел свою собственную специальную операционную систему, свою систему команд и т.д. В результате программу, успешно работающую на одном типе машин, необходимо было полностью переписать и заново отладить для другого типа компьютеров. В начале третьего периода появилась идея создания семейств программно-совместимых машин, работающих под управлением одной и той же операционной системы. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию цена/производительность. За ней последовала линия компьютеров PDP, несовместимых с линией IBM, кульминацией которой стала PDP-11.

Сила одной семьи была одновременно и ее слабостью. Широкие возможности этой концепции (наличие всех моделей: от миникомпьютеров до гигантских машин; обилие разнообразной периферии; различное окружение; различные пользователи) порождали сложную и огромную операционную систему. Миллионы строчек ассемблера, написанные тысячами программистов, содержали множество ошибок, что вызывало непрерывный поток публикаций о них и попыток их исправления. Только в операционной системе

OS/360 содержалось более 1000 известных ошибок. Тем не менее, идея стандартизации операционных систем была широко внедрена в сознание пользователей и в дальнейшем получила активное развитие.

Четвертый период (1980-настоящее время). Персональные компьютеры. Классические, сетевые и распределенные системы.

Следующий период в эволюции вычислительных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер, не отличающийся по архитектуре от PDP-11, по цене и простоте эксплуатации стал доступен отдельному человеку, а не отделу предприятия или университета. Наступила эра персональных компьютеров. Первоначально персональные компьютеры предназначались для использования одним пользователем в однопрограммном режиме, что повлекло за собой деградацию архитектуры этих ЭВМ и их операционных систем (в частности, пропала необходимость защиты файлов и памяти, планирования заданий и т.п.).

Компьютеры стали широко использоваться неспециалистами, что потребовало разработки "дружественного" программного обеспечения, это положило конец кастовости программистов.

Однако рост сложности и разнообразия задач, решаемых на персональных компьютерах, необходимость повышения надежности их работы привели к возрождению практически всех черт, характерных для архитектуры больших вычислительных систем.

В середине 80-х стали бурно развиваться сети компьютеров, в том числе персональных, работающих под управлением сетевых или распределенных операционных систем.

В сетевых операционных системах пользователи, при необходимости воспользоваться ресурсами другого сетевого компьютера, должны знать о его наличии и уметь это сделать. Каждая машина в сети работает под управлением своей локальной операционной системы, отличающейся от операционной системы автономного компьютера наличием дополнительных средств (программной поддержкой для сетевых интерфейсных устройств и доступа к удаленным ресурсам), но эти дополнения существенно не меняют структуру операционной системы.

Распределенная система, напротив, внешне выглядит как обычная автономная система. Пользователь не знает и не должен знать, где его файлы хранятся на локальной или удаленной машине, и где его программы выполняются. Он может вообще не знать, подключен ли его компьютер к сети. Внутреннее строение распределенной операционной системы имеет существенные отличия от автономных систем.

ОСНОВНЫЕ ФУНКЦИИ ОС

1. Планирование заданий и использования процессора.
2. Обеспечение программ средствами коммуникации и синхронизации.
3. Управление памятью.
4. Управление файловой системой.
5. Управление вводом-выводом.
6. Обеспечение безопасности

Каждая из приведенных функций обычно реализована в виде подсистемы, являющейся структурным компонентом ОС. В каждой конкретной операционной системе эти функции, конечно, реализовывались по-своему, в различном объеме. Они не были придуманы как составные части деятельности операционных систем изначально, а появились в процессе развития, по мере того, как вычислительные системы становились удобнее, эффективнее и безопаснее. Эволюция вычислительных систем, созданных человеком пошла по такому пути, но никто еще не доказал, что это единственно возможный путь их развития. Операционные системы существуют потому, что на настоящий момент их существование - это разумный способ использования вычислительных систем. Рассмотрение общих принципов и алгоритмов реализации их функций и будет составлять содержание большей части нашего курса.

2.3. Основные понятия ОС

В процессе эволюции возникло несколько важных концепций, которые стали неотъемлемой частью теории и практики ОС. Рассматриваемые в данном разделе понятия будут встречаться и разъясняться на протяжении всего настоящего курса. Здесь дается их краткое описание.

СИСТЕМНЫЕ ВЫЗОВЫ

В любой операционной системе поддерживается некоторый механизм, который позволяет пользовательским программам обращаться за услугами ядра ОС. В операционных системах наиболее известной советской вычислительной машины БЭСМ-6 соответствующие средства общения с ядром назывались экстракодами, в операционных системах IBM они назывались системными макрокомандами и т.д. В ОС UNIX такие средства называются системными вызовами.

Системные вызовы (system calls) интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и используют различные объекты, главные из которых процессы и файлы. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек сделать системный вызов похожим на обычный вызов подпрограммы.

Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или режим ядра (kernel mode). Поэтому системные вызовы иногда еще называют программными прерываниями в отличие от аппаратных прерываний, которые чаще называют просто прерываниями.

В этом режиме работает код ядра операционной системы, причем он исполняется в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса одной или нескольких областей памяти с параметрами вызова и адреса одной или нескольких областей памяти для результатов вызова.

В большинстве операционных систем системный вызов осуществляется командой программного прерывания (INT). Таким образом, программное прерывание это синхронное событие.

ПРЕРЫВАНИЯ

Прерывание (hardware interrupt) событие, генерируемое внешним (по отношению к процессору) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что возникло какое-либо событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении асинхронной операции ввода/вывода (например, закончено чтение данных с диска в основную память). Важный тип аппаратных прерываний прерывания таймера, которые генерируются периодически через фиксированный промежуток времени. Прерывания таймера используются операционной системой при планировании процессов. Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. Аппаратное прерывание это асинхронное событие, то есть оно возникает вне зависимости от того, какой код исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс является текущим.

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Исключительная ситуация (exсerption) событие, возникающее в результате попытки выполнения программой недопустимой команды, доступа к ресурсу при отсутствии достаточных привилегий или обращения к отсутствующей странице памяти. Исключительные ситуации так же, как и системные вызовы, являются синхронными событиями, возникающими в контексте текущей задачи. Исключительные ситуации можно разделить на исправимые и неисправимые. К исправимым относятся такие исключительные ситуации, как отсутствие нужной информации в оперативной памяти. После устранения причины исправимой исключительной ситуации программа может продолжить выполнение. Возникновение в процессе работы операционной системы исправимых исключительных ситуаций является нормальным явлением. Неисправимые исключительные ситуации обычно возникают в результате ошибок в программах. Обычно операционная система реагирует на такие ситуации завершением программы, вызвавшей исключительную ситуацию.

ФАЙЛЫ

Файлы предназначены для хранения информации на внешних носителях, то есть, принято, что информация, лежащая, например, на диске, должна находиться внутри файла. Обычно под файлом понимают часть пространства на носителе информации, имеющую имя.

Главная задача файловой системы (file system) скрыть особенности ввода-вывода и дать программисту простую абстрактную модель файлов, независимых от устройств. Для чтения, создания, удаления, записи, открытия и закрытия файлов также имеется обширная категория системных вызовов (create, delete, open, close, read, write).. Пользователям хорошо знакомы такие понятия, связанные с организацией файловой системы, как каталог, текущий каталог, корневой каталог, путь, для манипулирования которыми в операционной системе имеются системные вызовы. Файловая система ОС описана в лекции 12.

ПРОЦЕССЫ, НИТИ

Фундаментальным понятием для изучения работы операционных систем является понятие процессов, как основных динамических объектов, над которыми системы выполняют определенные действия. Данная глава посвящена описанию таких объектов, их состояний и свойств, их представлению в вычислительных системах, а также операциям, которые могут проводиться над ними.

ПОНЯТИЕ ПРОЦЕССА

Термины "программа" и "задание" предназначены для описания статических, неактивных объектов. Программа же в процессе исполнения является динамическим, активным объектом. По ходу ее работы компьютер обрабатывает различные команды и преобразует значения переменных. Для ее выполнения операционная система должна выделить определенное количество оперативной памяти, закрепить за ней определенные устройства ввода-вывода или файлы (откуда должны поступать входные данные и куда нужно доставить полученные результаты), то есть зарезервировать определенные ресурсы из общего числа ресурсов всей вычислительной системы. Их количество и конфигурация могут изменяться с течением времени. Для описания таких активных объектов внутри компьютерной системы вместо терминов "программа" и "задание" мы будем использовать новый термин "процесс".

Понятие процесса характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы. Не существует взаимно однозначного соответствия между процессами и программами, обрабатываемыми вычислительными системами. В некоторых операционных системах для работы определенных программ может организовываться более одного процесса или один и тот же процесс может исполнять последовательно несколько различных программ. Более того, даже в случае обработки только одной программы в рамках одного процесса, нельзя считать, что процесс представляет собой просто динамическое описание кода исполняемого файла, данных и выделенных для них ресурсов. Процесс находится под управлением операционной системы и поэтому в нем может выполняться часть кода ее ядра (не находящегося в исполняемом файле!), как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ими ситуациях (например, при обработке внешних прерываний).

СОСТОЯНИЯ ПРОЦЕССА

При использовании такой абстракции все, что выполняется в вычислительных системах (не только программы пользователей, но и, возможно, определенные части операционных систем), организовано как набор процессов. Понятно, что реально на однопроцессорной компьютерной системе в каждый момент времени может исполняться только один процесс. Для мультипрограммных вычислительных систем псевдопараллельная обработка нескольких процессов достигается с помощью переключения процессора с одного процесса на другой. Пока один процесс выполняется, остальные ждут своей очереди на получение процессора.

Предлагаемая модель хорошо описывает поведение процессов во время их жизни, она акцентирует внимания на появлении процесса в системе и его исчезновении из системы.



Диаграмма состояний процесса.

Для появления в вычислительной системе процесс должен пройти через состояние рождение. При рождении процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса; ему выделяются стек и системные ресурсы; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние готовности. При завершении своей деятельности процесс из состояния исполнения попадает в состояние – закончил исполнение.

В конкретных операционных системах состояния процесса могут быть еще более детализованы, могут появиться некоторые новые варианты переходов из состояния в состояние. Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы UNIX - 9. Тем не менее, в принципе, все операционные системы подчиняются изложенной выше модели.

ОПЕРАЦИИ НАД ПРОЦЕССАМИ

Процесс не может сам перейти из одного состояния в другое. Изменением состояния процессов занимается операционная система, совершая операции над ними.

ПЕРЕКЛЮЧЕНИЕ КОНТЕКСТА

Деятельность мультипрограммной операционной системы состоит из цепочек операций, выполняемых над различными процессами, и сопровождается переключением процессора с одного процесса на другой. В любой момент времени процесс полностью описывается своим контекстом, состоящим из регистровой, системной и пользовательской частей. В операционной системе процессы представляются определенной структурой данных.

Для корректного переключения процессора с одного процесса на другой необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется переключением контекста. Время, затраченное на переключение контекста, не используется вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно меняется от машины к машине и обычно находится в диапазоне от 1 до 1000 микросекунд. Существенно сократить накладные расходы в современных операционных системах позволяет расширенная модель процессов, включающая в себя понятие threads of execution (нити исполнения или просто нити).

Резюме

Процессы могут находиться в пяти основных состояниях: рождение, готовность, исполнение, ожидание, закончил исполнение. Из состояния в состояние процесс переводится операционной системой в результате выполнения над ним операций. Операционная система может выполнять над процессами следующие операции: создание процесса, завершение процесса, приостановка процесса, запуск процесса, блокирование процесса, разблокирование процесса, изменение приоритета процесса. Между выполнением операций содержимое PCB не изменяется. Деятельность мультипрограммной операционной системы состоит из цепочек перечисленных операций, выполняемых над различными процессами, и сопровождается процедурами сохранения/восстановления работоспособности процессов, т. е. переключением контекста. Переключение контекста не имеет отношения к полезной работе, выполняемой процессами, и время, затраченное на него, уменьшает полезное время работы процессора.

2.4. Архитектурные особенности ОС.

Мы с вами до сих пор говорили о взгляде на операционные системы извне, о том, что делают операционные системы. Дальнейший наш курс будет посвящен тому, как они это делают. Но мы пока ничего не сказали о том, что они представляют внутри себя, какие подходы существуют к их построению.

Монолитное ядро

По сути дела, операционная система это обычная программа, поэтому было бы логичным и организовать его так же, как устроено большинство программ, то есть составить из процедур и функций. В этом случае компоненты операционной системы являются не самостоятельными модулями, а составными частями одной большой программы. Такая структура операционной системы называется монолитным ядром (monolithic kernel). Монолитное ядро представляет собой набор процедур, каждая из которых может вызвать каждую. Все процедуры работают в привилегированном режиме. Таким образом, монолитное ядро это такая схема операционной системы, при которой все ее компоненты являются составными частями одной программы, используют общие структуры данных и взаимодействуют друг с другом путем непосредственного вызова процедур. Для монолитной операционной системы ядро совпадает со всей системой.

Во многих операционных системах с монолитным ядром сборка ядра, то есть его компиляция, осуществляется отдельно для каждого компьютера, на который устанавливается операционная система. При этом можно выбрать список оборудования и программных протоколов, поддержка которых будет включена в ядро. Так как ядро является единой программой, перекомпиляция это единственный способ добавить в него новые компоненты или исключить неиспользуемые. Следует отметить, что присутствие в ядре лишних компонентов крайне нежелательно, так как ядро всегда полностью располагается в оперативной памяти. Кроме того, исключение ненужных компонент повышает надежность операционной системы в целом.

Монолитное ядро старейший способ организации операционных систем. Примером систем с монолитным ядром является большинство Unix-систем.

Даже в монолитных системах можно выделить некоторую структуру. Как в бетонной глыбе можно различить вкрапления щебенки, так и в монолитном ядре выделяются вкрапления сервисных процедур, соответствующих системным вызовам. Сервисные процедуры выполняются в привилегированном режиме, тогда как пользовательские программы в непривилегированном режиме. Для перехода с одного уровня привилегий на другой иногда может использоваться главная сервисная программа, определяющая, какой именно системный вызов был сделан, корректность входных данных для этого вызова, и передающая управление соответствующей сервисной процедуре с переходом в привилегированный режим работы. Иногда выделяют также набор программных утилит, которые помогают выполнению сервисных процедур.

СЛОЕНЫЕ СИСТЕМЫ (LAYERED SYSTEMS)

Продолжая структуризацию, можно разбить всю вычислительную систему на ряд более мелких уровней с хорошо определенными связями между ними, так чтобы объекты уровня N могли вызывать только объекты из уровня N-1. Нижним уровнем в таких системах обычно является hardware, верхним уровнем интерфейс пользователя. Чем ниже уровень, тем более привилегированные команды и действия может выполнять модуль, находящийся на этом уровне. Впервые такой подход был применен при создании системы THE (Technische Hogeschool Eindhoven) Дейкстры и его студентами в 1968 г. Эта система имела следующие уровни:

5	Интерфейс пользователя
4	Управление вводом-выводом
3	Драйвер устройства связи оператора и консоли
2	Управление памятью
1	Планирование задач и процессов
0	Hardware

Рис. 1.2 Структура слоеной системы THE.

Слоеные системы хорошо реализуются. При использовании операций нижнего слоя не нужно знать, как они реализованы, нужно знать лишь, что они делают. Слоеные системы хорошо тестируются. Отладка начинается с нижнего слоя и проводится послойно. При возникновении ошибки мы можем быть уверены, что она находится в тестируемом слое. Слоеные системы хорошо модифицируются. При необходимости можно заменить лишь один слой, не трогая остальные. Но слоеные системы сложны для разработки: тяжело правильно определить порядок слоев, и что, к какому слою относится. Слоеные системы менее эффективны, чем

монолитные. Так, например, для выполнения операций ввода-вывода программе пользователя придется последовательно проходить все слои - от верхнего до нижнего.

ВИРТУАЛЬНЫЕ МАШИНЫ

В начале лекции мы говорили о взгляде на операционную систему как на виртуальную машину, когда пользователю нет необходимости знать детали внутреннего устройства компьютера. Он работает с файлами, а не с магнитными головками и двигателем; он работает с огромной виртуальной, а не ограниченной реальной оперативной памятью; его мало волнует, единственный он на машине пользователь или нет. Рассмотрим несколько другой подход. Пусть операционная система реализует виртуальную машину для каждого пользователя, но, не упрощая ему жизнь, а, наоборот, усложняя. Каждая такая виртуальная машина предстает перед пользователем как абсолютно голое железо копия всего hardware в вычислительной системе, включая процессор, привилегированные и непривилегированные команды, устройства ввода-вывода, прерывания и т.д. И он один на один с этим железом. При попытке обратиться к этому виртуальному железу на уровне привилегированных команд, в действительности происходит системный вызов реальной операционной системы, которая и производит все необходимые действия. Такой подход позволяет каждому пользователю загрузить свою собственную операционную систему на виртуальную машину и делать с ней все, что душа пожелает.

Программа пользователя	Программа пользователя	Программа пользователя
MS-DOS	Linux	Windows-NT
Виртуальное hardware	Виртуальное hardware	Виртуальное hardware
Реальная операционная система		
Реальное hardware		

Рис. 1.3 Вариант виртуальной машины.

Первой реальной системой такого рода была система CP/CMS или VM/370, как ее называют сейчас, для семейства машин IBM/370.

Недостатком таких операционных систем является снижение эффективности виртуальных машин по сравнению с реальной машиной, и, как правило, они очень громоздки. Преимуществом - использование на одной вычислительной системе программ, написанных для разных операционных систем.

МИКРОЯДЕРНАЯ АРХИТЕКТУРА.

Современная тенденция в разработке операционных систем это перенесение значительной части системного кода на уровень пользователя и одновременной минимизации ядра. Речь идет о подходе к построению ядра, называемом микроядерной архитектурой (microkernel architecture) операционной системы, когда большинство ее составляющих являются самостоятельными программами. В этом случае взаимодействие между ними обеспечивает специальный модуль ядра, называемый микроядром. Микроядро работает в привилегированном режиме и обеспечивает взаимодействие между программами, планирование использования процессора, первичную обработку прерываний, операции ввода-вывода и базовое управление памятью. Остальные компоненты системы взаимодействуют друг с другом путем передачи сообщений через микроядро.

Основное достоинство микроядерной архитектуры высокая степень модульности ядра операционной системы. Это существенно упрощает добавление в него новых компонент. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Существенно упрощается процесс отладки компонент ядра, так как новая версия драйвера может загружаться без перезапуска всей операционной системы. Компоненты ядра операционной системы ничем принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства. В то же время, микроядерная архитектура операционной системы вносит дополнительные накладные расходы, связанные с передачей сообщений, что существенно влияет на производительность. Для того чтобы микроядерная операционная система по скорости не уступала операционным системам на базе монолитного ядра, требуется очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними. Таким образом, основная сложность при создании микроядерных операционных систем необходимость очень аккуратного проектирования.

СМЕШАННЫЕ СИСТЕМЫ

Все рассмотренные подходы к построению операционных систем имеют свои преимущества и недостатки. В большинстве случаев современные операционные системы используют различные комбинации этих подходов. Примером смешанного подхода может служить возможность запуска операционной системы с монолитным ядром под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре

Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляется монолитным ядром. Данный подход возник в результате попыток использовать преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженный код монолитного ядра.

Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT. Хотя Windows NT часто называют микроядерной операционной системой, это не совсем так. Микроядро NT слишком сложно и велико (более 1 Мб), чтобы носить приставку микро. Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных операционных системах. В тоже время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром. Кроме того, в Windows NT существует разделение между режимом ядра и режимом пользователя еще одна черта монолитного ядра. Причина всего этого проста. По мнению Microsoft, причина проста: чисто микроядерный дизайн коммерчески непрактичен, так как слишком неэффективен. Таким образом, Windows NT можно с полным правом назвать гибридной операционной системой.

2.5. Классификация ОС

Операционные системы могут различаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами.

Ниже приведена классификация ОС по нескольким наиболее основным признакам.

УПРАВЛЕНИЕ РЕСУРСАМИ

От эффективности алгоритмов управления локальными ресурсами компьютера во многом зависит эффективность всей сетевой ОС в целом. Поэтому, характеризуя сетевую ОС, часто приводят важнейшие особенности реализации функций ОС по управлению процессорами, памятью, внешними устройствами автономного компьютера. Так, например, в зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на многозадачные и однозадачные, многопользовательские и однопользовательские, на системы, поддерживающие многопотоковую обработку и не поддерживающие ее, на многопроцессорные и однопроцессорные системы.

Многозадачность.

По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса:

однозадачные (например, MS-DOS, MSX) и

многозадачные (OS EC, OS/2, UNIX, Windows 95).

Однозадачные ОС в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Многозадачные ОС, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

Многопользовательский режим.

По числу одновременно работающих пользователей ОС делятся на:

однопользовательские (MS-DOS, Windows 3.x, ранние версии OS/2);

многопользовательские (UNIX, Windows NT).

Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

Вытесняющая и невытесняющая многозадачность. Важнейшим разделяемым ресурсом является процессорное время. Способ распределения процессорного времени между несколькими одновременно

существующими в системе процессами (или нитями) во многом определяет специфику ОС. Среди множества существующих вариантов реализации многозадачности можно выделить две группы алгоритмов:

невывесняющая многозадачность (NetWare, Windows 3.x);

вывесняющая многозадачность (Windows NT, OS/2, UNIX).

Основным различием между вывесняющим и невывесняющим вариантами многозадачности является степень централизации механизма планирования процессов. В первом случае механизм планирования процессов целиком сосредоточен в операционной системе, а во втором - распределен между системой и прикладными программами. При невывесняющей многозадачности активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению процесс. При вывесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается операционной системой, а не самим активным процессом.

ПАРАЛЛЕЛЬНАЯ ОБРАБОТКА

Поддержка многонитевости. Важным свойством операционных систем является возможность распараллеливания вычислений в рамках одной задачи. Многонитевая ОС разделяет процессорное время не между задачами, а между их отдельными ветвями (нитеями).

Многопроцессорная обработка. Другим важным свойством ОС является отсутствие или наличие в ней средств поддержки многопроцессорной обработки - мультипроцессирование. Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами.

В наши дни становится общепринятым введение в ОС функций поддержки многопроцессорной обработки данных. Такие функции имеются в операционных системах Solaris 2.x фирмы Sun, Open Server 3.x компании Santa Cruz Operations, OS/2 фирмы IBM, Windows NT фирмы Microsoft и NetWare 4.1 фирмы Novell.

Многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой: асимметричные ОС и симметричные ОС. Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

Выше были рассмотрены характеристики ОС, связанные с управлением только одним типом ресурсов - процессором. Важное влияние на облик операционной системы в целом, на возможности ее использования в той или иной области оказывают особенности и других подсистем управления локальными ресурсами - подсистем управления памятью, файлами, устройствами ввода-вывода.

Специфика ОС проявляется и в том, каким образом она реализует сетевые функции: распознавание и перенаправление в сеть запросов к удаленным ресурсам, передача сообщений по сети, выполнение удаленных запросов. При реализации сетевых функций возникает комплекс задач, связанных с распределенным характером хранения и обработки данных в сети: ведение справочной информации о всех доступных в сети ресурсах и серверах, адресация взаимодействующих процессов, обеспечение прозрачности доступа, тиражирование данных, согласование копий, поддержка безопасности данных.

СЕТЕВАЯ ОС.

На свойства операционной системы непосредственное влияние оказывают аппаратные средства, на которые она ориентирована. По типу аппаратуры различают операционные системы персональных компьютеров, мини-компьютеров, мэйнфреймов, кластеров и сетей ЭВМ. Среди перечисленных типов компьютеров могут встречаться как однопроцессорные варианты, так и многопроцессорные. В любом случае специфика аппаратных средств, как правило, отражается на специфике операционных систем.

Очевидно, что ОС большой машины является более сложной и функциональной, чем ОС персонального компьютера. Так в ОС больших машин функции по планированию потока выполняемых задач, очевидно, реализуются путем использования сложных приоритетных дисциплин и требуют большей вычислительной мощности, чем в ОС персональных компьютеров. Аналогично обстоит дело и с другими функциями.

Сетевая ОС имеет в своем составе средства передачи сообщений между компьютерами по линиям связи, которые совершенно не нужны в автономной ОС. На основе этих сообщений сетевая ОС поддерживает разделение ресурсов компьютера между удаленными пользователями, подключенными к сети. Для поддержания функций передачи сообщений сетевые ОС содержат специальные программные компоненты, реализующие популярные коммуникационные протоколы, такие как IP, IPX, Ethernet и другие.

Многопроцессорные системы требуют от операционной системы особой организации, с помощью которой сама операционная система, а также поддерживаемые ею приложения могли бы выполняться параллельно отдельными процессорами системы. Параллельная работа отдельных частей ОС создает дополнительные проблемы для разработчиков ОС, так как в этом случае гораздо сложнее обеспечить согласованный доступ отдельных процессов к общим системным таблицам, исключить эффект гонок и прочие нежелательные последствия асинхронного выполнения работ.

Другие требования предъявляются к операционным системам кластеров. Кластер - слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой. Наряду со специальной аппаратурой для функционирования кластерных систем необходима и программная поддержка со стороны операционной системы, которая сводится в основном к синхронизации доступа к разделяемым ресурсам, обнаружению отказов и динамической реконфигурации системы. Одной из первых разработок в области кластерных технологий были решения компании Digital Equipment на базе компьютеров VAX. Недавно этой компанией заключено соглашение с корпорацией Microsoft о разработке кластерной технологии, использующей Windows NT. Несколько компаний предлагают кластеры на основе UNIX-машин.

Наряду с ОС, ориентированными на совершенно определенный тип аппаратной платформы, существуют операционные системы, специально разработанные таким образом, чтобы они могли быть легко перенесены с компьютера одного типа на компьютер другого типа, так называемые мобильные ОС. Наиболее ярким примером такой ОС является популярная система UNIX. В этих системах аппаратно-зависимые места тщательно локализованы, так что при переносе системы на новую платформу переписываются только они. Средством, облегчающем перенос остальной части ОС, является написание ее на машинно-независимом языке, например, на С, который и был разработан для программирования операционных систем.

ОСОБЕННОСТИ ОБЛАСТЕЙ ИСПОЛЬЗОВАНИЯ

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

системы пакетной обработки (например, ОС ЕС),

системы разделения времени (UNIX, VMS),

системы реального времени (QNX, RT/11).

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели в системах пакетной обработки используются следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины; так, например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается "выгодное" задание. Следовательно, в таких ОС невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит только в случае, если активная задача сама отказывается от процессора, например, из-за необходимости выполнить операцию ввода-вывода. Поэтому одна задача может надолго занять процессор, что делает невозможным выполнение интерактивных задач. Таким образом, взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок снижает эффективность работы пользователя.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки - изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю системы разделения времени предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину. Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая "выгодна" системе, и, кроме того, имеются накладные расходы вычислительной мощности на более частое переключение процессора с задачи на

задачу. Критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя.

Системы реального времени применяются для управления различными техническими объектами, такими, например, как станок, спутник, научная экспериментальная установка или технологическими процессами, такими, как гальваническая линия, доменный процесс и т.п. Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа, управляющая объектом, в противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности для систем реального времени является их способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия). Это время называется временем реакции системы, а соответствующее свойство системы - реактивностью. Для этих систем мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется исходя из текущего состояния объекта или в соответствии с расписанием плановых работ.

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть задач может выполняться в режиме пакетной обработки, а часть - в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

ОСОБЕННОСТИ МЕТОДОВ ПОСТРОЕНИЯ

При описании операционной системы часто указываются особенности ее структурной организации и основные концепции, положенные в ее основу.

К таким базовым концепциям относятся:

Способы построения ядра системы - монолитное ядро или микроядерный подход. Большинство ОС использует монолитное ядро, которое компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский и наоборот. Альтернативой является построение ОС на базе микроядра, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС - серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским, зато система получается более гибкой - ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

Построение ОС на базе объектно-ориентированного подхода дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри операционной системы, а именно: аккумуляцию удачных решений в форме стандартных объектов, возможность создания новых объектов на базе имеющихся с помощью механизма наследования, хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне, структурированность системы, состоящей из набора хорошо определенных объектов.

Наличие нескольких прикладных сред дает возможность в рамках одной ОС одновременно выполнять приложения, разработанные для нескольких ОС. Многие современные операционные системы поддерживают одновременно прикладные среды MS-DOS, Windows, UNIX (POSIX), OS/2 или хотя бы некоторого подмножества из этого популярного набора. Концепция множественных прикладных сред наиболее просто реализуется в ОС на базе микроядра, над которым работают различные серверы, часть которых реализуют прикладную среду той или иной операционной системы.

Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах. В распределенной ОС реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера. Характерными признаками распределенной организации ОС являются: наличие единой справочной службы разделяемых ресурсов, единой службы времени, использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам, многократной обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети, а также наличие других распределенных служб.

Лекция 8

Управление памятью в ОС.

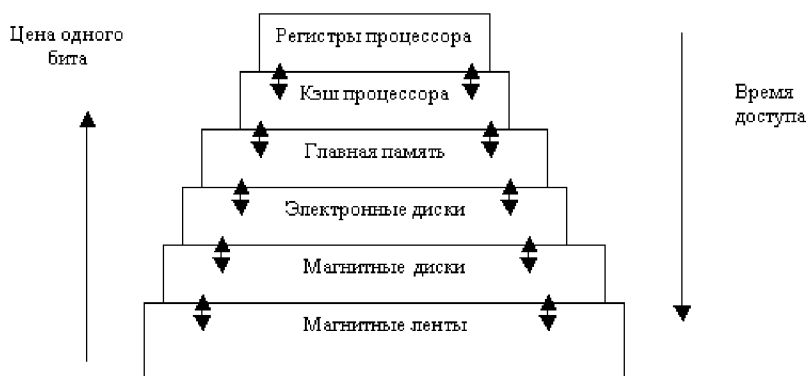
*Иерархия памяти_Менеджер памяти_Сегментация_Физическая и логическая память
Связывание адресов_Схемы управления памятью._Свопинг_Виртуальная память.
Страничная память_Сегментная и сегментно-страничная организации памяти_Таблица
страниц_Размер страницы*

• Схемы управления памятью.

Главная задача компьютерной системы - выполнять программы. Программы, в течение выполнения, вместе с данными, к которым они имеют доступ должны (по крайней мере, частично) находиться в главной (основной, оперативной) памяти. Таким образом, память (storage, memory) является важнейшим ресурсом, требующим тщательного управления. В недавнем прошлом память - самый дорогой ресурс.

ИЕРАРХИЯ ПАМЯТИ

Рассмотрим схему многоуровневой памяти. Например, как показано на рисунке, разновидности памяти могут быть организованы в иерархию по убыванию скорости доступа и возрастанию цены.



Считается, что затраты, связанные с переписью информации из одной памяти в другую окажутся меньше выигрыша в быстродействии, который получается за счет сокращения времени выборки из более быстрых слоев памяти. Информация о странице, которая находится в памяти верхнего уровня, хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную страницу на i -м уровне, он начинает искать ее на последующих уровнях. Когда нужная страница найдена, она переносится в более быстрые уровни.

При этом происходит вытеснение какой-то старой страницы, обычно той, которая дольше всего не использовалась. Идея состоит в том, чтобы те страницы, которые чаще всего нужны в настоящее время, находились в более быстрых частях памяти. Эффективность такой схемы обусловлена все тем же свойством локальности (подробнее о локальности и связанным с ним понятием рабочего множества страниц будет рассказано в следующей главе). В результате среднее время доступа для многоуровневой схемы памяти оказывается весьма близким ко времени доступа первого уровня.

МЕНЕДЖЕР ПАМЯТИ

Часть ОС, которая управляет памятью, называется менеджером памяти. В процессе эволюции в менеджерах памяти современных ОС было реализовано несколько основополагающих идей.

Во-первых, это идея сегментации.

СЕГМЕНТАЦИЯ

По-видимому, вначале сегменты памяти появились в связи с необходимостью обобществления процессами фрагментов программного кода (текстовый редактор, тригонометрические библиотеки и т.д.), без чего каждый процесс должен был хранить в своем адресном пространстве дублирующую информацию. Эти отдельные участки памяти, хранящие информацию, которую система отображает в память нескольких процессов, получили название сегментов. Память, таким образом, стала двумерной. Адрес состоит из двух компонентов: номер сегмента, смещение внутри сегмента. Далее оказалось удобным размещать в разных сегментах данные разных типов (код программы, данные, стек и т. д.). Попутно выяснилось, что можно контролировать характер работы с конкретным сегментом, приписав ему атрибуты, например, права доступа или типы операций, разрешенные с данными, хранящимися в сегменте. Большинство современных ОС поддерживают сегментную организацию памяти (см. раздел, где описана модель памяти процесса). В некоторых архитектурах (Intel, например) сегментация поддерживается оборудованием.

ФИЗИЧЕСКАЯ И ЛОГИЧЕСКАЯ ПАМЯТЬ

Вторая идея, о которой можно упомянуть, рассматривая поддержку памяти в ОС, это разделение памяти на физическую и логическую. Адреса, к которым обращается процесс, отделяются от адресов, реально существующих в оперативной памяти. Адрес, сгенерированный программой, обычно называют логическим (в системах с виртуальной памятью он обычно называется виртуальным) адресом, тогда как адрес, который видит устройство памяти (то есть нечто, загруженное в адресный регистр) обычно называется физическим адресом. Задача ОС, в какой-то момент времени осуществить связывание (или отображение) логического адресного пространства с физическим (см. раздел 8.2).

И, наконец, идея локальности. Свойство локальности присуще природе. Пространственная локальность - соседние объекты характеризуются похожими свойствами (если в данной местности хорошая погода, то вероятнее всего, что в близкой окрестности также хорошая погода). Временная локальность - если в 15:00 была хорошая погода, то, вероятно, что и в 14:30 и в 15:30 также наблюдалась хорошая погода. Свойство локальности (скорее эмпирическое) присуще и работе ОС. Фактически свойство локальности объяснимо, если учесть, как пишутся программы и организованы данные, то есть обычно в течение какого-то отрезка времени ограниченный фрагмент кода работает с ограниченным набором данных. Понимание данной особенности позволяет организовать иерархию памяти, используя быструю дорогостоящую память для хранения минимума необходимой информации, размещая оставшуюся часть данных на устройствах с более медленным доступом и подкачивая их в быструю память по мере необходимости. Типичный пример иерархии: регистры процессора, кэш процессора, главная память, внешняя память на магнитных дисках (вторичная память).

Главная память - это массив слов или байт. Каждое слово имеет свой адрес. Использование вторичной памяти (хранение данных на дисках) в качестве расширения главной дает дополнительные преимущества. Во-первых, главная память слишком мала, чтобы содержать все необходимые программы и данные постоянно. Во-вторых, главная память есть изменчивое (volatile) устройство, которое теряет свое содержимое, когда питание отключено или по другим причинам. Одно из требований к вторичной памяти - умение хранить большие объемы данных постоянно.

Функциями ОС по управлению памятью являются: отображение адресов программы на конкретную область физической памяти, распределение памяти между конкурирующими процессами и защита адресных пространств процессов, выгрузка процессов на диск, когда в оперативной памяти недостаточно места для всех процессов, учет свободной и занятой памяти.

Существует несколько схем управления памятью. Выбор той или иной схемы зависит от многих факторов. Рассматривая ту или иную схему важно учитывать:

- Механизм управления памятью или идеологию построения системы управления.
- Архитектурные особенности используемой системы.
- Структуры данных в ОС, используемые для управления памятью.
- Алгоритмы, используемые для управления памятью.

Вначале будут рассмотрены простейшие схемы, затем, описана доминирующая на сегодня схема виртуальной памяти, ее аппаратная и программная поддержка.

СВЯЗЫВАНИЕ АДРЕСОВ

Одна из функций управления памятью отображение информации в память. Отображение обычно понимается как преобразование адресных пространств. Адреса, с которыми имеет дело менеджер памяти, бывают логические (виртуальные для систем с виртуальной памятью) и физические.

Пользовательская программа не видит реальных физических адресов, а имеет дело с логическими адресами, которые являются результатом трансляции символьных имен программы. Логические адреса обычно образуются на этапе создания загрузочного модуля (линковки программы).

Набор адресов, сгенерированный программой, называют логическим (виртуальным) адресным пространством, которому соответствует физическое адресное пространство.

Максимальный размер логического адресного пространства обычно определяется разрядностью процессора (например, 2^{32}) и в современных системах значительно превышает размер физического адресного пространства.

Связывание логического адреса, порожденного оператором программы, с физическим должно быть осуществлено до начала выполнения оператора или в момент его выполнения.

Обычно программа проходит нескольких шагов:

- текст на алгоритмическом языке,
- объектный модуль,
- загрузочный модуль,
- бинарный образ в памяти.

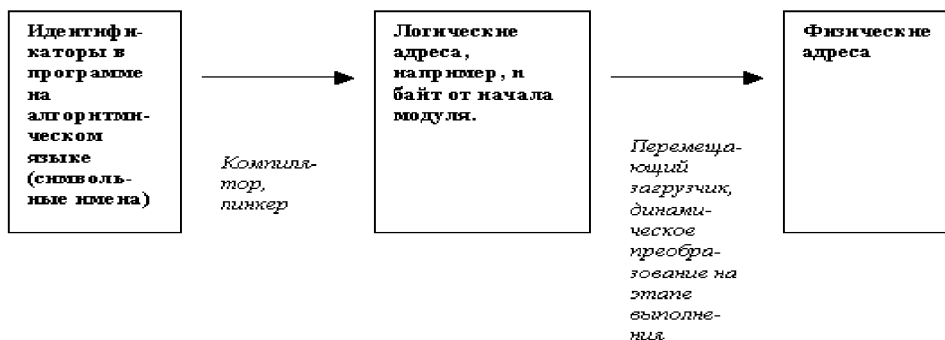


Рис. 4.1 Этапы связывания адресов.

Используемые программой адреса в каждом конкретном случае могут быть представлены различными способами. Например, адреса в исходных текстах обычно символические. Компилятор связывает эти символические адреса с перемещаемыми адресами (такими как n байт от начала модуля). Загрузчик или линкер, в свою очередь, связывают эти перемещаемые адреса с виртуальными адресами. Каждое связывание - отображение одного адресного пространства в другое.

Привязка инструкций и данных к памяти в принципе может быть, таким образом, сделана на следующих шагах:

- Этап компиляции (Compile time). Когда на стадии компиляции известно точное место размещения процесса в памяти, тогда генерируются абсолютные адреса. Если стартовый адрес программы меняется, необходимо перекомпилировать код. В качестве примера можно привести .com программы MS-DOS, которые связывают ее с физическими адресами на стадии компиляции.
- Этап загрузки (Load time). Если на стадии компиляции не известно где процесс будет размещен в памяти, компилятор генерирует перемещаемый код. В этом случае окончательное связывание откладывается до момента загрузки. Если стартовый адрес меняется, нужно всего лишь перезагрузить код с учетом измененной величины.
- Этап выполнения (Execution time). Если процесс может быть перемещен во время выполнения из одного сегмента памяти в другой, связывание откладывается до времени выполнения. Здесь желательно специализированное оборудование, например регистры перемещения. Их значение прибавляется к каждому адресу, сгенерированному процессом. Например, MS-DOS использует четыре таких (сегментных) регистра.

СХЕМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ.

ОС начали свое существование с применения очень простых методов управления памятью. Применявшаяся техника распространялась от статического распределения памяти (каждый процесс пользователя должен полностью поместиться в основной памяти, и система принимает к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещаются в основной памяти), с промежуточным решением в виде "простого свопинга" (система по-прежнему располагает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю память и заменяет его в основной памяти образом некоторого другого процесса). Схемы такого рода имеют не только историческую ценность. В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

1) Схема с фиксированными разделами.

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. По мере прибытия процесс помещается в тот или иной раздел.

Как правило, происходит условное разбиение физического адресного пространства. Связывание логических адресов процесса и физических происходит на этапе его загрузки в конкретный раздел.

Каждый раздел может иметь свою очередь или может существовать глобальная очередь для всех разделов.

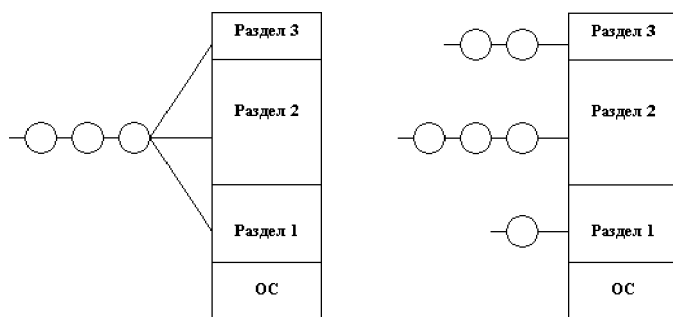


Рис. 4.2 Схема с фиксированными разделами: (а) с общей очередью процессов, (б) с отдельными очередями процессов.

Эта схема была реализована в IBM OS/360 (MFT) и в DEC RSX-11.

Подсистема управления памятью сравнивает размер программы, поступившей на выполнение, выбирает подходящий раздел, осуществляет загрузку программы и настройку адресов.

В какой раздел помещать программу? Распространены три стратегии:

1. Стратегия первого подходящего (First fit). Задание помещается в первый подходящий по размеру раздел.
2. Стратегия наиболее подходящего (Best fit). Задание помещается в тот раздел, где ему наиболее тесно.
3. Стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что с точки зрения утилизации памяти и уменьшения времени первые два способа лучше. С точки зрения утилизации первые два примерно одинаковы, но первый способ быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например, для размещения файлов на диске.

Связывание (настройка) адресов для данной схемы возможны как на этапе компиляции, так и на этапе загрузки.

Очевидный недостаток этой схемы число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внешней фрагментации потерей памяти, не используемой ни одним процессом. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или вследствие не использования некоторых разделов, которые слишком малы для выполняемых пользовательских программ.

1.1) Один процесс в памяти

Частный случай схемы с фиксированными разделами работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС - сверху, снизу или посередине. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение - расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Чтобы пользовательская программа не портила кода ОС, требуется защита ОС, которая может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

1.2) Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше чем размер выделенного ему раздела (или больше чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея - держать в памяти только те инструкции программы, которые нужны в данный момент времени.

Потребность в таком способе загрузки появляется, если логическое адресное пространство системы мало, например 1 мегабайт (MS-DOS) или даже всего 64 килобайта (PDP-11), а программа относительно велика. На современных 32-разрядных системах, где виртуальное адресное пространство измеряется гигабайтами, проблемы с нехваткой памяти решаются другими способами (см. раздел Виртуальная память).

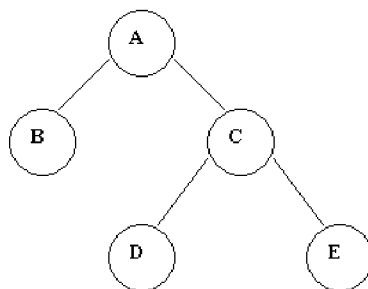
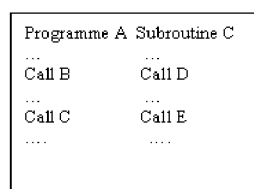


Рис 4.3 . Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы.

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для конструирования оверлеев необходимы специальные алгоритмы перемещения и связывания. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Например, для примера, приведенного на рис. 8.3 , текст этого файла может выглядеть так:

A-(B,C)

C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи не требуют специальной поддержки со стороны ОС. Они могут быть полностью реализованы на пользовательском уровне с простой файловой структурой. ОС лишь делает несколько больше операций ввода-вывода. Типовое решение порождение линкером специальных команды, которые включают загрузчик каждый раз: когда требуется обращение к одной из перекрывающихся ветвей программы.

Программист должен тщательно проектировать оверлейную структуру. Это требует полного знания структуры программы, кода, данных, языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с лимитами на память и т.д. Как мы увидим в дальнейшем проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что здесь мы впервые сталкиваемся со свойством локальности, которое дает возможность хранить в памяти только ту информацию, которая необходима в каждый конкретный момент вычислений.

2) Свопинг

Имея дело с пакетными системами можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping)

СВОПИНГ

- перемещение процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск связана с пейджингом (paging) будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. В системах со свопингом время переключения контекстов лимитируется временем загрузки/выгрузки процессов. Для эффективной утилизации процессора необходимо, чтобы величина кванта времени существенно его превышала.

Оптимизация свопинга может быть связана с выгрузкой лишь реально используемой памяти или выгрузкой процессов, реально не функционирующих. Кроме того, выгрузка обычно осуществляется в специально отведенное пространство для свопинга, то есть быстрее, чем через стандартную файловую систему (пространство выделяется большими блоками, поиск файлов и методы непосредственного выделения не используются).

Во многих версиях Unix свопинг обычно запрещен, однако он стартует, когда возрастает загрузка системы.

3) Мультипрограммирование с переменными разделами.

В принципе, система свопинга может базироваться на фиксированных разделах. На практике, однако, использование фиксированных разделов приводит к большим потерям используемой памяти, когда задача существенно меньше раздела.

Более эффективной представляется схема с переменными (динамическими) разделами. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется необходимая память. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой набор занятых и свободных участков (рис. 8.4). Смежные свободные участки могут быть объединены в один.

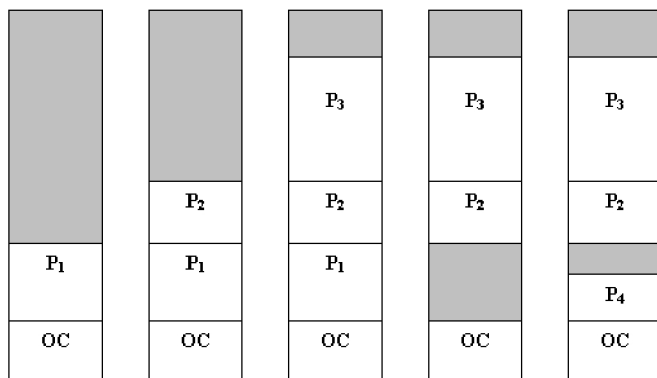


Рис. 4.4 Динамика распределения памяти между процессами. Серым цветом показана неиспользуемая память.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборке его среди имеющихся в соответствие с одной из стратегий (first fit, best fit, worst fit), загрузке процесса в выбранный раздел и последующем внесении изменений в таблицы свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может быть осуществлено на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов.

Этому методу также присуща внешняя фрагментация вследствие наличия большого числа участков свободной памяти. Проблемы фрагментации могут быть различными. В худшем случае мы можем иметь участок свободной (потерянной) памяти между двумя процессами. Если все эти куски объединить в один блок, мы смогли бы разместить больше процессов. Выбор между first-fit и best-fit слабо влияет на величину фрагментации.

В зависимости от суммарного размера памяти и среднего размера процесса эта проблема может быть большей или меньшей. Статистический анализ показывает, что при наличии n блоков пропадает $n/2$ блоков, то есть 1/3 памяти! Это известное 50% правило (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены в один).

Одно из решений проблемы внешней фрагментации - разрешить адресному пространству процесса не быть непрерывным, что разрешает выделять процессу память в любых доступных местах. Один из способов реализации такого решения - это paging, используемый во многих современных ОС (будет рассмотрен ниже).

Другим способом борьбы с внешней фрагментацией является сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память

образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать.

Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден, и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

- **Виртуальная память. Архитектурные средства поддержки виртуальной памяти**

Уже давно существует проблема размещения в памяти программ, размер которых превышает размер доступной памяти. Один из вариантов ее решения организация структур с перекрытием рассмотрен в предыдущей главе. При этом предполагалось активное участие программиста в процессе сегментации и загрузки программы. Было предложено переложить проблему на компьютер. Развитие архитектуры компьютеров привело к значительному усложнению организации памяти, соответственно, усложнились и расширились задачи операционной системы по управлению памятью. Одним из главных усовершенствований архитектуры стало появление виртуальной памяти (virtual memory). Она впервые была реализована в 1959 г. на компьютере Атлас, разработанном в Манчестерском университете, и стала популярной только спустя десятилетие.

ВИРТУАЛЬНАЯ ПАМЯТЬ.

При помощи виртуальной памяти обычно решают две задачи. Во-первых, виртуальная память позволяет адресовать пространство, гораздо большее, чем емкость физической памяти конкретной вычислительной машины. В соответствии с принципом локальности для реальных программ обычно нет необходимости в помещении их в физическую память целиком.

Возможность выполнения программы, находящейся в памяти лишь частично имеет ряд вполне очевидных преимуществ:

Программа не ограничена величиной физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.

Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.

Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге, каждая программа будет работать быстрее.

Таким образом, возможность обеспечения (при поддержке операционной системы) для программы видимости практически неограниченной (32- или 64-разрядной) адресуемой пользовательской памяти при наличии основной памяти существенно меньших размеров очень важный аспект. Но введение виртуальной памяти позволяет решать другую не менее важную задачу обеспечение контроля доступа к отдельным сегментам памяти и в частности защиту пользовательских программ друг от друга и защиту ОС от пользовательских программ.

С целью защиты виртуальная память поддерживалась и на компьютерах с 16-разрядной адресацией, в которых объем основной памяти зачастую существенно превышал 64 Кбайта (размер виртуальной памяти). Например, 16-разрядный компьютер PDP-11/70 мог иметь до 2 Мбайт оперативной памяти. Операционная система этого компьютера, тем не менее, поддерживала виртуальную память, основным смыслом которой являлось обеспечение защиты и перераспределения основной памяти между пользовательскими процессами.

Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа, - (логические адреса) - называются виртуальными, и они формируют виртуальное адресное пространство. В отсутствие механизма виртуальной памяти виртуальное адресное пространство непосредственно отображается в физическое пространство.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после получения соответствующей аппаратной поддержки. Идея аппаратной части механизма виртуальной памяти состоит в том, что адрес памяти, вырабатываемый командой, интерпретируется аппаратурой не как реальный адрес некоторого элемента основной памяти, а как некоторая структура, где адрес является лишь одним из компонентов наряду с атрибутами, характеризующими способ обращения по данному адресу.

Традиционно считается, что существует три модели виртуальной памяти: страничная, сегментная и их комбинация - сегментно-страничная модель. По-видимому, более правильно считать, что существует (и поддерживается аппаратно большинством платформ) страничная модель виртуальной памяти. Сегментно-страничная модель является синтезом страничной модели и идеи сегментации, изложенной в предыдущей

главе. Причем для тех архитектур, в которых сегменты не поддерживаются аппаратно их реализация задача архитектурно-независимой компоненты менеджера памяти. Сегментная организация в чистом виде практически не встречается.

Архитектурные средства поддержки виртуальной памяти.

Очевидно, что невозможно создать полностью машинно-независимый компонент управления виртуальной памятью. С другой стороны, имеются существенные части программного обеспечения, связанного с управлением виртуальной памятью, для которых детали аппаратной реализации совершенно не важны. Одним из достижений современных ОС является грамотное и эффективное разделение средств управления виртуальной памятью на аппаратно-независимую и аппаратно-зависимую части. Коротко рассмотрим, что и каким образом входит в аппаратно-зависимую часть подсистемы управления виртуальной памятью. Компоненты аппаратно-независимой подсистемы рассмотрены в следующей главе.

Итак, мы имеем большое (для 32-разрядных архитектур это обычно $2^{32} = 4 \text{ Гб}$) виртуальное адресное пространство и физическое пространство существенно меньшего размера. Пользовательский процесс или ОС должны иметь возможность осуществить запись по виртуальному адресу, а задача ОС сделать так, чтобы записанная информация оказалась в физической памяти (впоследствии при нехватке оперативной памяти она может быть вытеснена во внешнюю память).

Таким образом, важный компонент менеджера виртуальной памяти система или функция отображения (трансляции) адресов. Механизм преобразования виртуальных адресов в физические должен предусматривать ведение таблиц, показывающих, какие области виртуальной памяти в текущий момент находятся в физической памяти и где именно размещаются. Если бы такое отображение осуществлять побайтно, то информация об отображении была бы велика, и для ее хранения потребовалось бы больше реальной памяти, чем для процессов. Необходим способ, позволяющий существенно сократить объем информации отображения. Поэтому обычно отображаемая информация группируется в блоки (программа занимает целое количество блоков памяти).

СТРАНИЧНАЯ ПАМЯТЬ

В наиболее простом и наиболее часто используемом случае страничной виртуальной памяти виртуальная память и физическая представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами. Страницы, в отличие от сегментов, имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться.

Виртуальный адрес в страничной системе упорядоченная пара (p, d) , где p - номер страницы в виртуальной памяти, а d - смещение в рамках страницы p , где размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачена) из внешней памяти. Поступившая страница может быть размещена в любой свободный страничный кадр. Система отображения виртуальных адресов в физические сводится к системе отображения виртуальных страниц в физические и представляет собой таблицу страниц.

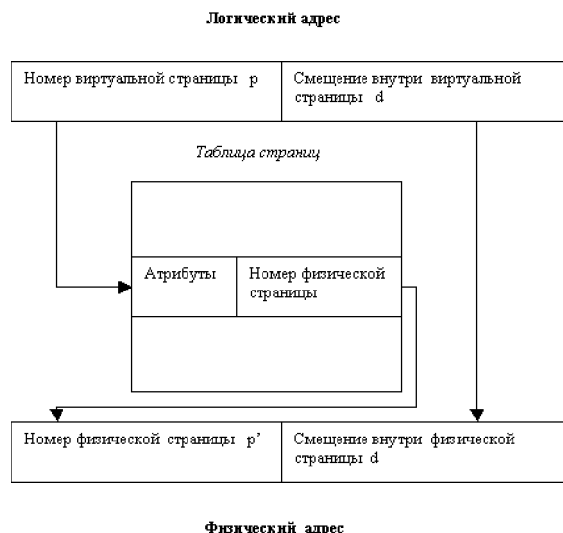


Рис. 4.5 Связь логического и физического адресов при страничной организации памяти.

Для преобразования адресного пространства каждого процесса используется одна или несколько таблиц страниц, которые обычно хранятся в оперативной памяти. Для ссылки на таблицу страниц используется специальный регистр процессора. Особенности хранения таблицы страниц описаны ниже. Интерпретация виртуального (логического) адреса показана на рис. 4.5

Динамическое преобразование адресов в системе осуществляется следующим образом. Выполняемый процесс обращается по виртуальному адресу $v = (p, d)$. Механизм отображения ищет номер страницы p в таблице отображения и определяет, что эта страница находится в страничном кадре p' , формируя реальный адрес из p' и d .

Например, в машинной инструкции `move reg, 0`, адрес 0 находится на первой виртуальной странице, которая может быть локализована, во втором страничном кадре. В реальных системах функция отображения хранит бит присутствия страницы в физической памяти. При отсутствии страницы в памяти возникает исключительная ситуация, называемая страничным нарушением (page fault). Обработка страничного нарушения приводит к тому, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и продолжается прерванный код. При отсутствии свободных страничных кадров на диск выгружается мало используемая страница.

Номер страницы, как индекс хранится в таблице страниц (page table), которая адресуется при помощи специального регистра процессора, и позволяет определить номер кадра по виртуальному адресу. Помимо этой основной задачи при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защите.

Используя схему пэйджинга, мы не имеем внешней фрагментации. Однако сталкиваемся с внутренней фрагментацией: адресное пространство процесса занимает целое число страниц, и в среднем половина страницы на процесс пропадает.

Важный аспект - различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя его память - единое непрерывное пространство, содержащее, только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя недоступна чужая память. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение должно происходить корректно даже в сложных случаях. ОС поддерживает одну или несколько (при наличии нескольких сегментов памяти) таблиц страниц для каждого процесса, для ссылки, на которую, как уже говорилось, обычно используется специальный регистр. При переключении процессов диспетчер должен найти его таблицу страниц, указатель на которую, таким образом, входит в контекст процесса.

В большинстве современных компьютеров со страничной организацией виртуальной памяти все таблицы страниц хранятся в основной памяти, а быстрота доступа к элементам таблицы текущей виртуальной памяти достигается, как будет показано ниже, за счет наличия сверхбыстродействующей буферной памяти (кэша).

Сегментная и сегментно-страничная организации памяти

Существуют две другие схемы организации виртуальной памяти: сегментная и сегментно-страничная. Идеи сегментации изложены во введении к предыдущей главе. При сегментной организации виртуальный адрес по-прежнему является двумерным и состоит из двух полей - номера сегмента и смещения внутри сегмента. Заметим, что с точки зрения ОС сегменты являются логическими сущностями и их главное назначение хранение и защита однородной информации (кода, данных и т.д.).

С точки зрения пользователя процесс представляется обычно не как линейный массив байтов, а как набор сегментов переменного размера (данные, код, стек). Сегментация - схема управления памятью, поддерживающая этот взгляд пользователя. Сегменты содержат процедуры, массивы, стек или скалярные величины, но обычно не содержат информацию смешанного типа.

Программисты, пишущие на языках низкого уровня должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство - набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). Пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением. (В отличие от схемы пэйджинга, где пользователь задает только один адрес, который разбивается hardware на номер страницы и смещение, прозрачным для программиста образом.)

Каждый сегмент - линейная последовательность адресов от 0 до максимума. Различные сегменты могут иметь различные длины, которые могут меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента (если виртуальный сегмент содержится в основной памяти) содержится длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает прерывание.

Логический адрес - упорядоченная пара $v=(s,d)$, номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам сегментов. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и др. и определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

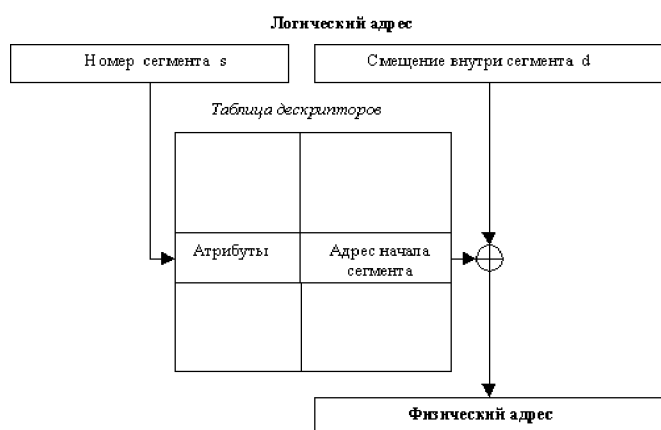


Рис. 4.6 Преобразование логического адреса при сегментной организации памяти.

Аппаратная поддержка сегментов относительно слабо распространена (главным образом на процессорах архитектуры Intel) и характеризуется довольно медленной загрузкой селекторов в сегментные регистры, выполняемая при каждом переключении контекста и при каждом переходе между разными сегментами. В системах с чисто страничной организацией памяти для описания типового адресного пространства процесса, представляющего собой набор сегментов, сегментация реализуется на уровне, независимом от аппаратуры.

Хранение в памяти сегментов большого размера может оказаться неудобным. Возникает идея их пейджинга. При сегментно-страничной организации виртуальной памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае виртуальный адрес состоит из трех полей: номера сегмента виртуальной памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения - таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

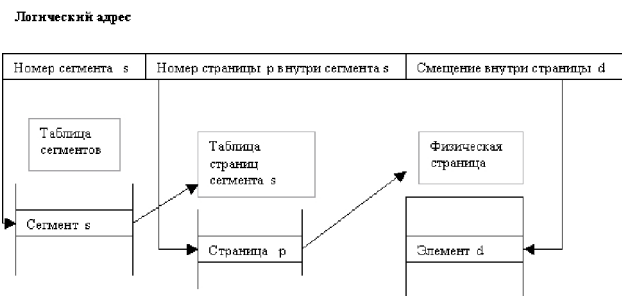


Рис. 4.7 Формирование физического адреса при сегментно-страничной организации памяти.

Сегментно-страничная организация виртуальной памяти позволяла совместно использовать одни и те же сегменты данных и программного кода в виртуальной памяти разных задач (для каждой виртуальной памяти существовала отдельная таблица сегментов, но для совместно используемых сегментов поддерживались общие таблицы страниц).

ТАБЛИЦА СТРАНИЦ

Организация таблицы страниц один из ключевых элементов механизмов страничного и сегментно-страничного преобразований. Рассмотрим структуру таблицы страниц более детально.

Итак, виртуальный адрес состоит из виртуального номера страницы (high-order bits) и смещения (low-order bits). Номер виртуальной страницы используется как индекс в таблице страниц для нахождения записи (entry) о виртуальной странице. Из этой записи в таблице страниц находится номер кадра (page frame number), затем прибавляется смещение и формируется физический адрес. Помимо этого запись в таблице страниц содержит информацию об атрибутах страницы, в частности биты защиты.

Основную проблему для эффективной реализации таблицы страниц создают большие размеры виртуальных адресных пространств современных компьютеров, которые обычно определяются разрядностью архитектуры процессора. Самыми распространенными на сегодняшний день являются 32-разрядные процессоры, позволяющие создавать виртуальные адресные пространства такого размером 4 Гб (для 64-разрядных компьютеров эта величина равна 2^{64}).

Подсчитаем примерный размер таблицы страниц. В 32-битном адресном пространстве при размере страницы 4К (Intel) получаем 1М страниц, а в 64-битном и того более. Т.е. таблица должна иметь 1М строк (entry), причем запись в строке состоит из нескольких байт. Заметим, что каждый процесс, нуждается в своей таблице страниц (а в случае сегментно-страничной схемы по одной на каждый сегмент). Итак, в этом случае таблица страниц может быть слишком большой.

Кроме того, отображение должно быть быстрым. Отображение должно быть быстрым т.к. оно делается при каждом обращении к памяти, которое происходит практически в каждой машинной инструкции. Эта проблема решается главным образом за счет реализации ассоциативной памяти (см. следующий раздел).

Для того чтобы избежать необходимости иметь огромную таблицу в памяти все время, а хранить лишь несколько ее фрагментов (это возможно опять же на основании свойства локальности!), многие компьютеры используют многоуровневую таблицу страниц.

Количество уровней в таблице страниц зависит от конкретных особенностей архитектуры. Можно привести примеры реализации одноуровневого (DEC PDP-11), двухуровневого (Intel, DEC VAX), трехуровневого (Sun SPARC, DEC Alpha) paging'a, а также paging'a с задаваемым количеством уровней (Motorola). Функционирование RISC процессора MIPS R2000 осуществляется вообще без таблицы страниц. Здесь поиск нужной страницы, если эта страница отсутствует в ассоциативной памяти, должна взять на себя ОС (так называемый zero level paging).

РАЗМЕР СТРАНИЦЫ

Дизайнеры ОС для существующих машин редко имеют возможность влиять на размер страницы. Однако для вновь создаваемых компьютеров решение относительно оптимального размера страницы является актуальным. Как и можно было ожидать нет одного наилучшего размера. Скорее есть набор факторов, влияющих на размер. Обычно размер страницы это степень двойки от 2^9 до 2^{14} байт.

Чем больше размер страницы, тем меньше будет размер структур данных, обслуживающих преобразование адресов, но тем больше будут потери, связанные с тем, что память можно выделять только постранично.

Как следует выбирать размер страницы? Во-первых, нужно учитывать размер таблицы страниц, здесь желателен большой размер страницы (страниц меньше, соответственно и таблица страниц меньше). С другой стороны память лучше утилизируется с маленьким размером страницы. В среднем половина последней страницы процесса пропадает. Необходимо также учитывать объем ввода-вывода для взаимодействия с внешней памятью и другие факторы. Проблема не имеет хорошего ответа. Историческая тенденция состоит в увеличении размера страницы. Как правило, размер страниц задается аппаратно, например, на Intel - это 4096 байт (или 4 Кбайт), на DEC PDP-11 - 8 Кбайт, на DEC VAX - 512 байт, на других архитектурах, таких как Motorola 68030, размер страниц может быть задан программно.

Некоторые аспекты организации работы с памятью предлагается на самостоятельное изучение.

- Структуры данных, используемые для описания сегментной модели
- Уровни управления виртуальной памятью
- Исключительные ситуации при работе с памятью.
- Стратегии управления страничной памятью
- Алгоритмы замещения страниц

Аппаратно-независимая модель памяти процесса.

Реализация функций операционной системы, связанных с поддержкой памяти: ведение таблиц страниц, трансляция адреса, обработка страничных ошибок, управление ассоциативной памятью и др., тесно связана со структурами данных, обеспечивающими удобное представление адресного пространства процесса. Формат этих структур сильно зависит от аппаратуры и особенностей конкретной ОС.

Чаще всего виртуальная память процесса ОС разбивается на сегменты пяти различных типов: кода программы, данных, стека, разделяемый и сегмент файлов, отображаемых в память.

Сегмент программного кода содержит только команды. Сегмент программного кода не модифицируется в ходе выполнения процесса, обычно страницы данного сегмента имеют атрибут read-only. Следствием этого является возможность использования одного экземпляра кода для разных процессов.

Сегмент данных содержащий переменные программы и сегмент стека, содержащий автоматические переменные, могут динамически менять свой размер (обычно данные в сторону увеличения адресов, а стек в сторону уменьшения) и содержимое, и должны быть доступны по чтению и по записи и являются приватными сегментами процесса.

С целью обобществления памяти между несколькими процессами создаются разделяемые сегменты, допускающие доступ по чтению и записи. Вариантом разделяемого сегмента может быть сегмент файла, отображаемого в память. Специфика таких сегментов состоит в том, что из таких сегментов откатка осуществляется не в системную область свопинга, а непосредственно в отображаемый файл.

Загрузка исполняемого файла (системный вызов `exec`) осуществляется обычно через отображение (mapping) его частей (кода, данных) в соответствующие сегменты адресного пространства процесса. После установления отображения, система начинает генерировать `page fault`'ы, в первую очередь для сегментов кода, данных и стека, подкачивая с диска необходимую информацию.

Как уже говорилось, сегмент данных динамически меняет свой размер. Пользователь, осуществляя аллокацию (библиотечный вызов `malloc`) или освобождение (`free`) динамических данных, фактически изменяет границу динамически выделенных процессу данных через системный вызов `brk`, который модифицирует значение переменной `brk` из структуры данных процесса. В результате происходит выделение физической памяти, а соответствующие строки таблиц страниц получают осмысленные значения. Ведение списков занятых и свободных областей памяти в сегменте данных пользователя осуществляется на уровне системных библиотек. На практике, освобожденная пользователем память (библиотечный вызов `free`) системе не возвращается, а резервируется для обслуживания возможных дальнейших запросов пользовательского процесса на выделение памяти.

Описанная система управления памятью является совокупностью программно-технических средств, обеспечивающих производительное функционирование современных компьютеров. Успех реализации той части ОС, которая относится к управлению виртуальной памятью, определяется близостью архитектуры аппаратуры, поддерживающей виртуальную память, к абстрактной модели виртуальной памяти ОС. Для справедливости заметим, что в подавляющем большинстве современных компьютеров аппаратура выполняет функции, существенно превышающие потребности модели ОС, так что создание аппаратно-зависимой части подсистемы управления виртуальной памятью ОС в большинстве случаев не является чрезмерно сложной задачей.

Режимы работы микропроцессора

Реальный режим

Это режим работы первых 16-битовых микропроцессоров. Наличие его обусловлено тем, что необходимо обеспечить в новых моделях микропроцессоров функционирование программ, разработанных для старых моделей.

Защищенный режим (protected mode)

Означает, что параллельные вычисления могут быть защищены программно-аппаратным путем.

Позволяет полностью использовать все возможности, предоставляемые микропроцессором. Все современные многозадачные ОС работают в этом режиме.

Создан для работы нескольких независимых программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимодействие задач должно регулироваться.

Программы, разработанные для реального режима, не могут функционировать в защищенном режиме. (Физический адрес формируется по другим принципам.)

Режим виртуального 8086

Переход в этот режим возможен, если микропроцессор уже находится в защищенном режиме. Возможна одновременная работа нескольких программ разработанных для i8086. Возможно работа программ реального режима. Физический адрес формируется по правилам реального режима.

Организация памяти

|| Физическая память, к которой микропроцессор имеет доступ по шине адреса, называется *оперативной памятью* (или оперативным запоминающим устройством - ОЗУ).

ОП организована как последовательность байтов.

|| Каждому байту соответствует уникальный адрес (его номер), который называется *физическим адресом*.

! Диапазон значений адресов зависит от разрядности шины адреса микропроцессора.

• Для i486 и Pentium он находится в диапазоне от 0 до $2^{32}-1$ --- 4 Гбайт (32-разрядная шина адреса).

! Механизм управления памятью полностью аппаратный, т.е. программа сама не может сформировать физический адрес памяти на адресной шине.

Микропроцессор аппаратно поддерживает несколько моделей использования оперативной памяти:

- сегментированную модель
- страничную модель

Понятие о сегментированной модели памяти

Память для программы делится на непрерывные области памяти, называемые *сегментами*.

Сегменты - это логические элементы программы.

Сама программа может обращаться только к данным, которые находятся в этих сегментах.

|| Сегмент представляет собой независимый, поддерживаемый на аппаратном уровне блок памяти.

Сегментация - механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния.

Замечание. Программист может либо самостоятельно разбивать программу на фрагменты (сегменты), либо автоматизировать этот процесс и возложить его на систему программирования.

Для микропроцессоров Intel принят особый подход к управлению памятью. Каждая программа в общем случае может состоять из любого количества сегментов, но непосредственный доступ она имеет только к 3 основным сегментам: кода, данных и стека и к дополнительным сегментам данных (всего 3).

Операционная система (! а не сама программа) размещает сегменты программы в ОП по определенным физическим адресам, а значения этих адресов записывает в определенные места, в зависимости от режима работы микропроцессора:

- в реальном режиме адреса помещаются непосредственно в сегментные регистры (cs, ds, ss, es, fs);
- в защищенном режиме - в специальную системную дескрипторную таблицу (Элементом дескрипторной таблицы является дескриптор сегмента. Каждый сегмент имеет дескриптор сегмента -8 байт. Существует три дескрипторные таблицы. Адрес каждой таблицы записывается в специальный системный регистр).

Для доступа к данным внутри сегмента обращение производится относительно начала сегмента линейно, т.е. начиная с 0 и заканчивая адресом, равным размеру сегмента. Этот адрес называется *смещением (offset)*.

Таким образом, для обращения к конкретному физическому адресу ОП необходимо определить адрес начала сегмента и смещение внутри сегмента.

Физический адрес принято записывать парой этих значений, разделенных двоеточием
segment : offset

Например, 0040:001Ch; 0000:041Ch; 0020:021Ch; 0041:000Ch.

Каждый сегмент описывается дескриптором сегмента.

ОС строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в ОП или внешней памяти в дескрипторе отмечает его текущее местоположение (бит присутствия).

Дескриптор содержит поле адреса, с которого сегмент начинается и поле длины сегмента. Благодаря этому можно осуществлять контроль

- 1) 1) размещения сегментов без наложения друг на друга
- 2) 2) обращается ли код исполняющейся задачи за пределы текущего сегмента.

В дескрипторе содержатся также данные о правах доступа к сегменту (запрет на модификацию, можно ли его предоставлять другой задаче) $\Rightarrow$ защита.

Достоинства:

- 1) 1) общий объем виртуальной памяти превосходит объем физической памяти
- 2) 2) возможность размещать в памяти как можно больше задач (до определенного предела) $\Rightarrow$ увеличивает загрузку системы и более эффективно используются ресурсы системы

Недостатки:

- 1) 1) увеличивается время на доступ к искомой ячейке памяти, т.к. должны вначале прочитать дескриптор сегмента, а потом уже, используя его данные, можно вычислить физический адрес (для уменьшения этих потерь используется кэширование - дескрипторы, с которыми работа идет в данный момент размещаются в сверхоперативной памяти - в специальных регистрах процессора);
- 2) 2) фрагментация;
- 3) 3) потери памяти на размещение дескрипторных таблиц
- 4) 4) потери процессорного времени на обработку дескрипторных таблиц.

! Сегментированная модель памяти поддерживается и в реальном, и в защищенном режимах работы микропроцессора.

Понятие о страничной модели памяти

Это настройка над сегментной моделью.

ОП делится на блоки фиксированного размера 4 Кб (должно быть число, кратное степени двойки, чтобы операции сложения можно было бы заменить на операции конкатенации).

Каждый такой блок называется *страницей*.

Их число $1.048.576 \Rightarrow 4$ Гб адресуемой памяти.

Основное применение этой модели связано с организацией **виртуальной памяти**.

Для того, чтобы использовать для работы программ пространство памяти большее, чем объем физической памяти используется механизм виртуальной памяти.

Суть его заключается в том, что у микропроцессора существует возможность по обмену страницами памяти с жестким диском. В случае, если программа требует памяти больше, чем объем физической памяти, редко используемые страницы памяти записываются на жесткий диск в специальный файл виртуальной памяти (файл обмена, или страничный файл, или файл подкачки, чаще swp-файлом, подчеркивая, что страницы этого файла замещают друг друга в ОП).

Замечание. В некоторых ОС выгруженные страницы располагаются не в файле, а в специальном разделе диска, например, в ОС UNIX есть специальный раздел, но могут использоваться и файлы, если не достаточно объема раздела.

В настоящее время файл подкачки может динамически изменять свой размер в зависимости от потребностей системы.

Для i486 и Pentium размер возможной виртуальной памяти может достигать 4 Тб (терабайт).

Обратим внимание на то, что программа также разбивается на фрагменты - страницы. Все фрагменты программы одинаковой длины, кроме последней страницы.

Говорят, что *память разбивается на физические страницы, а программа - на виртуальные страницы*.

Трансляция (отображение) виртуального адресного пространства задачи на физическую память осуществляется с помощью таблицы страниц.

Для каждой текущей задачи создается **таблица страниц**.

Диспетчер памяти для каждой страницы формирует соответствующий дескриптор. Дескриптор содержит так называемый бит присутствия.

Если он = 1, это означает, что данная страница сейчас размещена в ОП.

Если он = 0, то страница расположена во внешней памяти.

Защита страничной памяти основана на контроле уровня доступа к каждой странице.

Каждая страница снабжается кодом уровня доступа (только чтение; чтение и запись; только выполнение). При работе со страницей сравнивается значение кода разрешенного уровня доступа с фактически требуемым. При несовпадении работа программы прерывается.

! Страничная модель памяти поддерживается только в защищенном режиме работы микропроцессора.

• Основное достоинство страничного способа распределения памяти - *минимально возможная фрагментация (эффективное распределение памяти).*

Недостатки:

- 1) 1) потери памяти на размещение таблиц страниц
- 2) 2) потери процессорного времени на обработку таблиц страниц (диспетчер памяти).
- 3) 3) Программы разбиваются на страницы случайно, без учета логических взаимосвязей, имеющихся в коде
⇒ межстраничные переходы осуществляются чаще, чем межсегментные + трудности в организации разделения программных модулей между выполняющими процессами

Чтобы избежать недостатка №3 был предложен сегментно-страничный способ распределения памяти.

Сегментно-страничный способ распределения памяти

Программа разбивается на сегменты.

Адрес, по-прежнему, состоит из двух частей - сегмент + смещение.

Но смещение относительно начала сегмента может состоять из двух полей: виртуальной страницы и индекса.

Для доступа к памяти необходимо:

- 1) 1) вычислить адрес дескриптора сегмента и прочитать его;
- 2) 2) вычислить адрес элемента таблицы страниц этого сегмента и извлечь из памяти необходимый элемент;
- 3) 3) к номеру (адресу) физической страницы приписать номер (адрес) ячейки в странице.

⇒ Задержка в доступе к памяти (в три раза больше, чем при прямой адресации).

Чтобы избежать этого вводится кэширование (кэш строится по ассоциативному принципу). Мы будем это изучать позже.

Плоская модель памяти

Если считать, что задача состоит из одного сегмента, который, в свою очередь, разбит на страницы, то фактически мы получаем только один страничный механизм работы с виртуальной памятью.

Этот подход называется плоской памятью.

Достоинства:

- • При использовании плоской модели памяти упрощается создание и ОС, и систем программирования.
- • уменьшаются расходы памяти на поддержку системных информационных структур

В абсолютном большинстве современных 32-разрядных ОС (для микропроцессоров Intel) используется плоская модель памяти.