1. ОС – определение, место ОС в системе программного обеспечения ЭВМ. Ресурсы вычислительной системы. Режимы ядра и задачи: переключение в режим ядра.

ОС – комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, использующих их при вычислениях

Операционная система является фундаментальным компонентом системного программного обеспечения.

Классификация ОС

1. Однопрограммные пакетной обработки
2. Мультипрограммная пакетной обработки
3. Мультипрограммная с разделением времени

В оперативной памяти нах. большое число программ, процессорное время квантуется, чтобы обеспечить гарантированное время ответа системы. Время ответа системы не должно превышать 3 секунд.

4. Системы реального времени
5. серверные ОС – предоставляющие доступ к аппаратным (принтеры) и программным ресурсам (файлы доступа к Интернет) из сети.
6. многопроцессорные
7. встроенные ОС (телевизоры, микровольновые печи).
8. опер системы для смарт-карт (самые маленьеие операционные системы).

Ресурс – любой из компонентов вычислительной системы и предоставляемые им возможности.
Ресурсы вычислительной системы:

1. Время процессора
2. Объем физической памяти (ОЗУ)
3. Устройство ввода/вывода
4. Каналы
5. Таймер
6. Данные
7. Ключи защиты
8. Реентирабельные коды самой системы

Пользовательский режим - наименее привилегированный режим, поддерживаемый NT; он не имеет прямого доступа к оборудованию и у него ограниченный доступ к памяти.

Режим ядра - привилегированный режим. Те части NT, которые исполняются в режиме ядра, такие как драйверы устройств и подсистемы типа Диспетчера Виртуальной Памяти, имеют прямой доступ ко всей аппаратуре и памяти. Различия в работе программ пользовательского

режима и режима ядра поддерживаются аппаратными средствами компьютера (а именно - процессором).

Переключение процесса в режим ядра
Существуют 3 типа событий, которые могут перевести ОС в режим ядра:
1) системные вызовы (программные прерывания) – software interrupt – traps
2) аппаратные прерывания (прерывания, поступившие от устройств) - interrupts (от таймера, от устройств I/O, прерывания от схем контроля: уровень напряжения в сети, контроль четности памяти)
3) исключительные ситуации- exception

Процесс может находиться или в состоянии задачи или в состоянии ядра.
Вызовы системных сервисов – системные вызовы.
Аппаратные прерывания поступают от контроллера прерываний.
Прерывание таймера – аппаратное прерывание.
Вызов диспетчера, запускающий новый процесс, возложено на таймер.
Если в системе имеется переключение процессов, то процессорное время квантуется, если процесс не успел выполниться, то возвращается в очередь процессов.

Исключения
Исключения являются синхронным событием, возникают в процессе выполнения программы, возникаю в таких случаях, как:
- Арифметическое переполнение
- деление на нуль
- попытка выполнить некорректную команду,
- Ссылка на запрещенную область памяти
- При образовании адреса, при обращении к физическому адресу, которого нет

Исключения бывают 3 видов:
1. Нарушения – fault – это исключение, фиксируемое до выполнения команды или в процессе её выполнения.
2. Ловушка – Trap – процессором обрабатывается после команды, вызвавшей это исключение.
3. Авария – abort – данный тип исключения является следствием невосстановимых, неисправимых ошибок, нпример, деление на ноль.

Исключения бывают:
1. *Исправимые* – например, обращение к некорректному адресу, но он прошел проверку адреса, обращение к отсутствующему сегменту. Процесс может продолжаться с той же команды, в которой произошло исключение.
2. *Неисправимые* - заканчиваются завершение программы (деление на ноль)

Аппаратные прерывания:
Аппаратные - возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени.
Различают прерывания:
- От таймера
- От действия оператора (пример: ctrl+alt+del)
- От устройств вв/выв

Аппаратные прерывания: сигналы от внешних устройств поступают на контроллер прерывания, причем эти прерывания не зависят от выполняемого процесса, т.е процесс вполне может переключиться на выполнение какого-либо другого процесса. Аппаратные прерывания обрабатываются в системном контексте, при этом доступ в адресное пространство процесса им не нужен, т.е. им не нужен доступ к контексту процесса. Обработчик прерывания не обращается к контексту процесса. Очевидно, что прерывания interrupts не должны производить блокировку процесса.

Системные вызовы

Системный вызов – вызывается искусственно с помощью соответствующей команды из программы (int), предназначен для выполнения некоторых действий операционной системы (фактически запрос на услуги ОС), является синхронным событием.

Также являются исключениями, но с точки зрения реализации - это системные ловушки.
Набор можно рассматривать как программный интерфейс, предоставляемый ядром системы пользовательским процессам. Эти функции называются API функциями.
Supervisor call. – исполняемое ядро ОС. При системных вызовах сначала вызывается библиотечная функция, которая передает номер системного вызова в стек пользователя и вызывает специальную инструкцию системного прерывания (int)[Int 2eh – системные вызовы в Windows. Motorola 680.0 - trap], которое меняет режим выполнение на режим ядра и передает управление обработчику системного вызова. Системные вызовы выполняются в режиме ядра, но в контексте пользователя (задачи, процесса). Следовательно, они имеют доступ к адресному пространству и управляющим структурам, вызвавшего их процесса. С другой стороны, они могут обращаться к стеку ядра этого процесса. [Unix – syscall. Win2000 – KiSystem Service - диспетчер системных сервисов]. Обработчик системных вызовов загружает в stack pointer (sp) адрес стека режима ядра (kernel stack) процесса и сохраняет в этом стеке, аппаратный контекст процесса. В аппаратный контекст включается:

- instruction pointer (ip)
- stack pointer (sp)
- слово состояния процессора (processor status word) (PSW)
- регистры управления памятью
- регистры сопроцессора (floating point unit).

Затем, по номеру системного вызова, происходит обращение к соответствующей таблице [В win2000 к таблице диспетчеризации системных вызовов. (System Service dispatch table)], содержащей указатель на обработчик системных прерываний. После завершения, супервизор восстанавливает возвращенные из обработчика системных вызовов значения или код ошибки в соответствующие регистры. Затем восстанавливает аппаратный контекст процесса, вызывает библиотечную функцию, которая восстанавливает режим задачи или пользовательский режим, а указатель стека (sp) переключается на стек режима пользователя (user stack).

Эти оба стека принадлежат процессу. (user stack, kernel stack)

2. Реальный, защищенный, виртуальный 8086

Реальный режим Это режим работы первых 16-битовых микропроцессоров с 20ти разрадной шиной адреса и диапазон адресов памяти ограничен одним мегабайтом. Наличие его обусловлено тем, что необходимо обеспечить в новых моделях микропроцессоров функционирование программ, разработанных для старых моделей.

Защищенный режим (protected mode) 32-разрадный, многпоточный, многопроцессный, 4 уровня привелегий, доступно 4 Гб виртуальной памяти(для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. 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

В режим V86 процессор может перейти из защищённого режима, если установить в регистре флагов EFLAGS бит виртуального режима (VM-бит). Номер бита VM в регистре EFLAGS - 17.

Когда процессор i80386 находится в виртуальном режиме, для адресации памяти используется схема <сегмент:смещение>, размер сегмента составляет 64 килобайта, а размер адресуемой в этом режиме памяти - 1 мегабайт.

Виртуальный режим - это не реальный режим процессора i8086, имеются существенные отличия. Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима.

В виртуальном режиме используется трансляция страниц памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство, каждое размером в 1 мегабайт.

Все задачи виртуального режима обычно выполняются в третьем, наименее привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор автоматически переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему, работающую в защищённом режиме.

1. Реальный режим (или режим реальных адресов) - это название было дано прежнему способу адресации памяти после появления 286-го процессора, поддерживающего защищённый режим.
2. Защищенный режим - Режим защиты памяти. Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Основная мысль сводится к формированию таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.
3. Виртуальный режим - В процессоре i386 компания Intel учла необходимость лучшей поддержки реального режима, потому что программное обеспечение времени его появления не было готово полностью работать в защищенном режиме. Поэтому, например, в i386, возможно переключение из защищенного режима обратно в реальный (при разработке 80286 считалось, что это не потребуется, поэтому на компьютерах с процессором 80286 возврат в реальный режим осуществляется схемно - через сброс процессора).

В качестве дополнительной поддержки реального режима, i386 позволяет задаче (или нескольким задачам) защищенного работать в виртуальном режиме — режиме эмуляции режима реального адреса (таким образом в переключении в реальный режим уже нет необходимости). Виртуальный режим предназначается для одновременного выполнения программы реального режима (например, программы DOS) под операционной системой защищенного режима. Выполнение в виртуальном режиме практически идентично реальному, за несколькими исключениями, обусловленными тем, что виртуальная задача выполняется в защищенном режиме:

- виртуальная задача не может выполнять привилегированные команды, потому что имеет наинизший уровень привилегий
- все прерывания и исключения обрабатываются операционной системой защищенного режима (которая, впрочем, может инициировать обработчик прерывания виртуальной задачи)

1. Классификация операционных систем и их особенности. Иерархическая машина. Виртуальная машина.

Классификация ОС

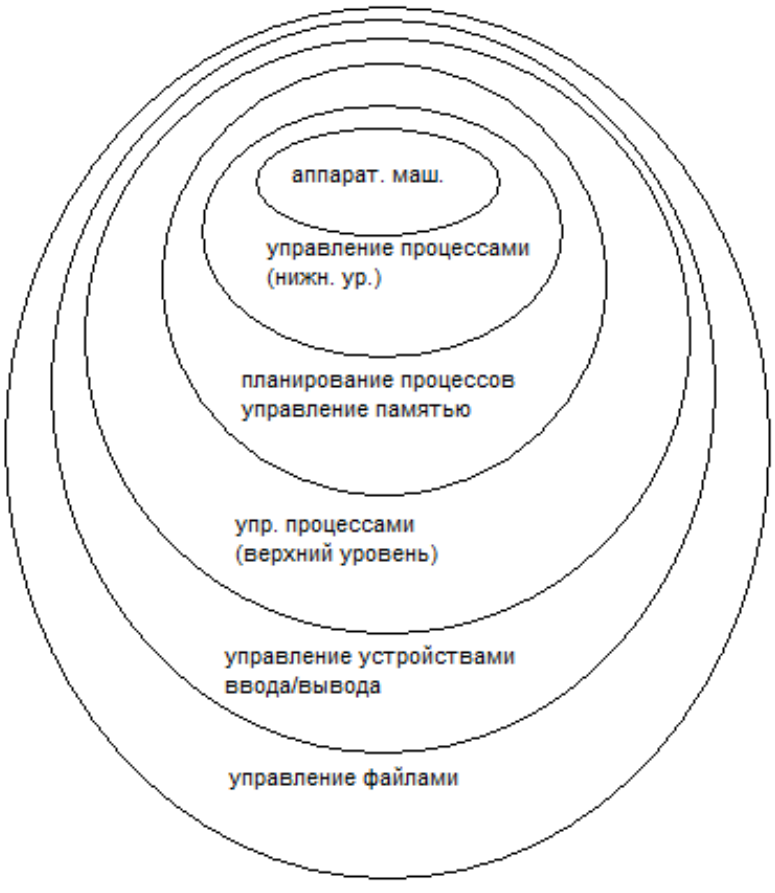
1. Однопрограммные пакетной обработки
2. Мультипрограммная пакетной обработки
3. Мультипрограммная с разделением времени

В оперативной памяти нах. большое число программ, процессорное время квантуется, чтобы обеспечить гарантированное время ответа системы. Время ответа системы не должно превышать 3 секунд.

4. Системы реального времени
5. серверные ОС – предоставляющие доступ к аппаратным (принтеры) и программным ресурсам (файлы доступа к Интернет) из сети.
6. многопроцессорные
7. встроенные ОС (телевизоры, микровольновые печи).
8. опер системы для смарт-карт (самые маленьеие операционные системы).

Пакет – набор программ, которые одновременно загруженны в память.
Виртуальная машина – кажущаяся, возможная. Виртуальная машина – набор команд и функций, необходимых пользователю для получения сервиса операционной системы

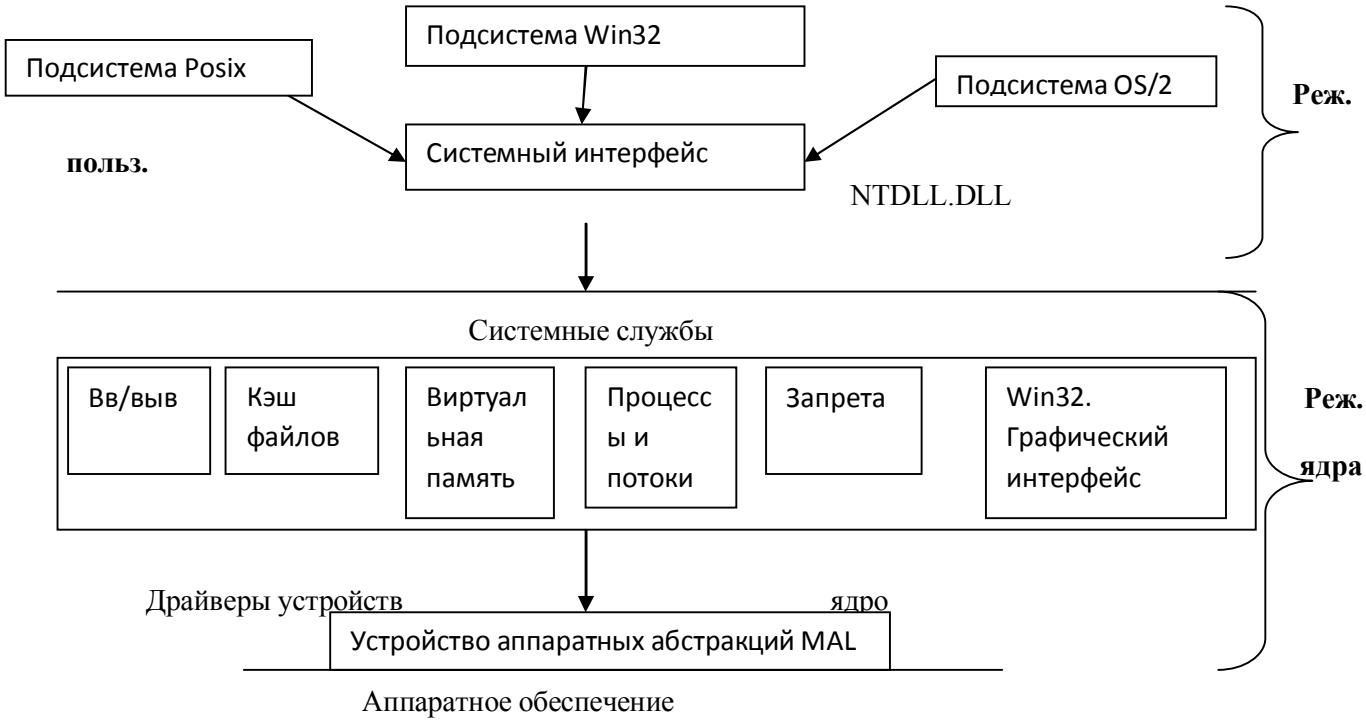
Иерархическая машина – ОС разбивается на функции, и опред место функции по отношению к аппаратной части
Иерархическая машина Данавана:



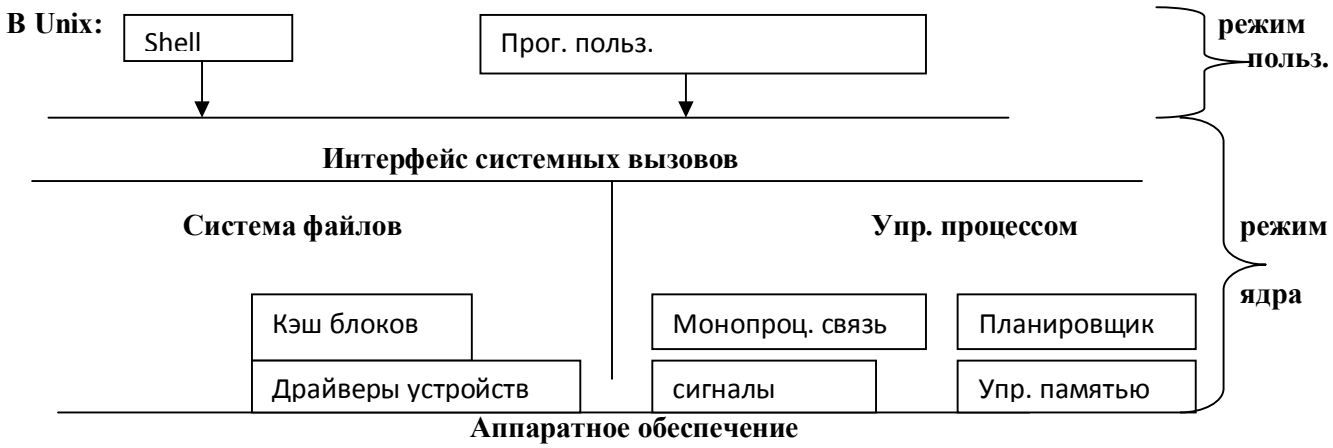
Уровень определяется близостью к аппаратной части. Между уровнями определяется интерфейс взаимодействия
Интерфейс:

1) непрозрачный – запрещено обращаться через уровни
2) полупрозрачный или прозрачный – возможно обращение через уровни

В Windows:



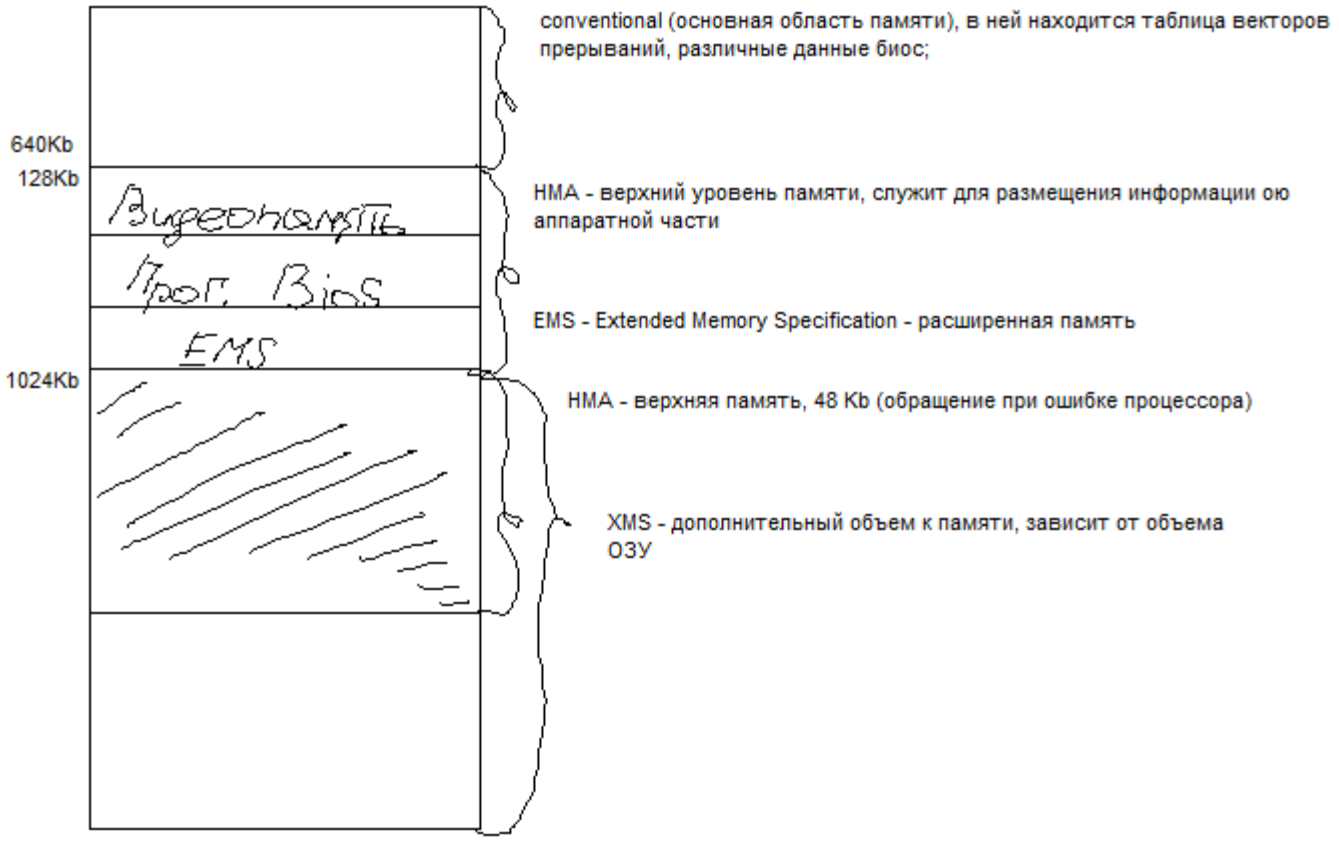
NTDLL.DLL – обращаясь к этой библиотеке процессы могут перейти в режим ядра



2. XMS, линия A20 – адресное заворачивание

При включении компьютера, он нах. в реальном режиме. Линия A20-21 – адресная линия, заземленная, нужно чтобы осуществлялось адресное заворачивание (для совместимости). При переходе в защ. режим необх. открыть линию A20.

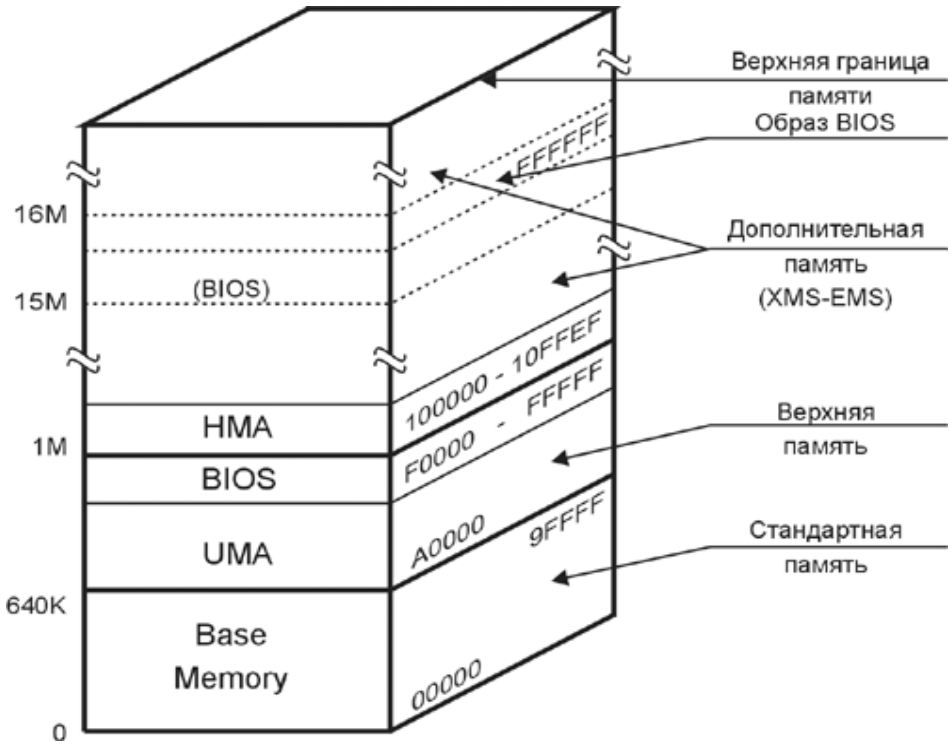
Логическая адресация памяти



В Intel 8086 только 20 адресных линий, что позволяло адресовать лишь 1 Mb памяти. При переполнении происходило обращение к таблице прерываний.

Если в реальном режиме открыть линию A20, то станет доступным доп 64Kb памяти, что позволяет адресовать до 4 гигов памяти. Изначально линия A20 заземлена.

МОЖНО ПОДРОБНЕЕ



00000h-9FFFFh - Conventional (Base) Memory, 640 Кбайт - стандартная (базовая) память, доступная DOS и программам реального режима. В некоторых системах с видеоадаптером MDA верхняя граница сдвигается к AFFFFh (704 Кбайт). Иногда верхние 128 Кбайт стандартной памяти (область 80000h-9FFFFh) называют Extended Conventional Memory. Стандартная память является самой дефицитной в PC, когда речь идет о работе в среде операционных систем типа MS-DOS. На ее небольшой объем (типовое значение 640 Кбайт) претендуют и BIOS, и ОС реального режима, а остатки отдаются прикладному ПО. Стандартная память распределяется следующим образом:

00000h-003FFh - Interrupt Vectors - векторы прерываний (256 двойных слов);

00400h-004FFh - BIOS Data Area - область переменных BIOS;

00500h-00xxxh - DOS Area - область DOS;

00xxxh-9FFFFh - User RAM - память, предоставляемая пользователю (до 638 Кбайт);

при использовании PS/2 Mouse область 9FC00h-9FFFFh используется как расширение BIOS Data Area, и размер User RAM уменьшается.

A0000h-FFFFFh - Upper Memory Area (UMA), 384 Кбайт - верхняя память, зарезервированная для системных нужд. В ней размещаются области буферной памяти адаптеров (например, видеопамять) и постоянная память (BIOS с расширениями). Эта область, обычно используемая не в полном объеме, ставит непреодолимый архитектурный барьер на пути непрерывной

(нефрагментированной) памяти, о которой мечтают программисты. Верхняя память имеет области различного назначения, которые могут быть заполнены буферной памятью адаптеров, постоянной памятью или оставаться незаполненными. Раньше эти “дыры” не использовали из-за сложности “фигурного выпиливания” адресуемого пространства. С появлением механизма страничной переадресации (у процессоров 386 и выше) их стали по возможности заполнять “островками” оперативной памяти, названными блоками верхней памяти UMB (Upper Memory Block). Эти области доступны DOS для размещения резидентных программ и драйверов через драйвер EMM386, который отображает в них доступную дополнительную память.

Память выше 100000h - Extended Memory - дополнительная (расширенная) память, непосредственно доступная только в защищенном (и в “большом реальном”) режиме для компьютеров с процессорами 286 и выше. В ней выделяется область 100000h-10FFEFh - высокая память, HMA, - единственная область расширенной памяти, доступная 286+ в реальном режиме при открытом вентиле Gate A20.

ТО ЧТО НЕПОСРЕДСТВЕННО ОТНОСИТСЯ К БИЛЕТУ - НИЖЕ

Расширенная память XMS (eXtended Memory Specification) - программная спецификация использования дополнительной памяти DOS-программами, разработанная компаниями Lotus, Intel, Microsoft и AST для компьютеров на процессорах 286 и выше. Эта спецификация позволяет программе получить в распоряжение одну или несколько областей дополнительной памяти, а также использовать область HMA. Распределением областей ведает диспетчер

расширенной памяти - драйвер HIMEM.SYS. Диспетчер позволяет захватить или освободить область HMA (65 520 байт, начиная с 100000h), а также управлять вентилем линии адреса A20. Функции собственно XMS позволяют программе:

- определить размер максимального доступного блока памяти;
- захватить или освободить блок памяти;
- копировать данные из одного блока в другой, причем участники копирования могут быть блоками как стандартной, так и дополнительной памяти в любых сочетаниях;
- запереть блок памяти (запретить копирование) и отпереть его;
- изменить размер выделенного блока.

Спецификации EMS и XMS отличаются по принципу действия: в EMS для доступа к дополнительной памяти выполняется отображение (страничная переадресация) памяти, а в XMS - копирование блоков данных.

Адресное заворачивание:

Процессор в реальном режиме поддерживает адресное пространство до 1Мбайт. Адресное пространство разбито на сегменты по 64Кбайт. 20-битный базовый адрес сегмента вычисляется сдвигом значения селектора на 4 бита влево. Данные внутри сегмента адресуются 16-битным смещением.

В этом режиме формирования линейного адреса есть возможность адресовать пространство между 1Мб и 1Мб+64Кб (например, указав в качестве селектора 0FFFFh, а в качестве смещения 0FFFFh, мы получим линейный адрес 10FFEFh). Однако МП 8086, обладая 20-разрядной шиной адреса, отбрасывает старший бит, "заворачивая" адресное пространство (в данном примере МП 8086 обратится по адресу 0FFEFh). В реальном режиме микропроцессоры IA-32 "заворачивания" не производят. Для 486+ появился новый сигнал - A20M#, который позволяет блокировать 20-й разряд шины адреса, эмулируя таким образом "заворачивание" адресного пространство, аналогичное МП 8086.

1. Прерывания: классификация; аппаратные прерывания - последовательность операций при выполнении аппаратного прерывания. Прерывания точные и неточные.

Классификация прерываний

В зависимости от источника, прерывания делятся на

- программные (по Рязановой это сист. вызов)
- аппаратные
- исключения

Системный вызов – вызывается искусственно с помощью соответствующей команды из программы (int), предназначен для выполнения некоторых действий операционной системы (фактически запрос на услуги ОС), является синхронным событием.

Исключения – являются реакцией микропроцессора на нестандартную ситуацию, возникшую внутри микропроцессора во время выполнения некоторой команды программы (деление на ноль, прерывание по флагу TF (трассировка)), являются синхронным событием.

Исключения бывают 3 видов:

1. Нарушения – fault – это исключение, фиксируемое до выполнения команды или в процессе её выполнения.
2. Ловушка – Trap – процессором обрабатывается после команды, вызвавшей это исключение.
3. Авария – abort – данный тип исключения является следствием невосстановимых, неисправимых ошибок, нпример, деление на ноль.

- Исправимые искл. – приводят к вызову определенного менеджера системы, в результате работы которого может быть продолжена работа процесса (пример: страничная неудача с менеджером памяти)
- Неисправимые искл. – в случае сбоя или в случае ошибки программы (пример: ошибка адресации). В этом случае процесс завершается.

Аппаратные - возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени.

Различают прерывания:

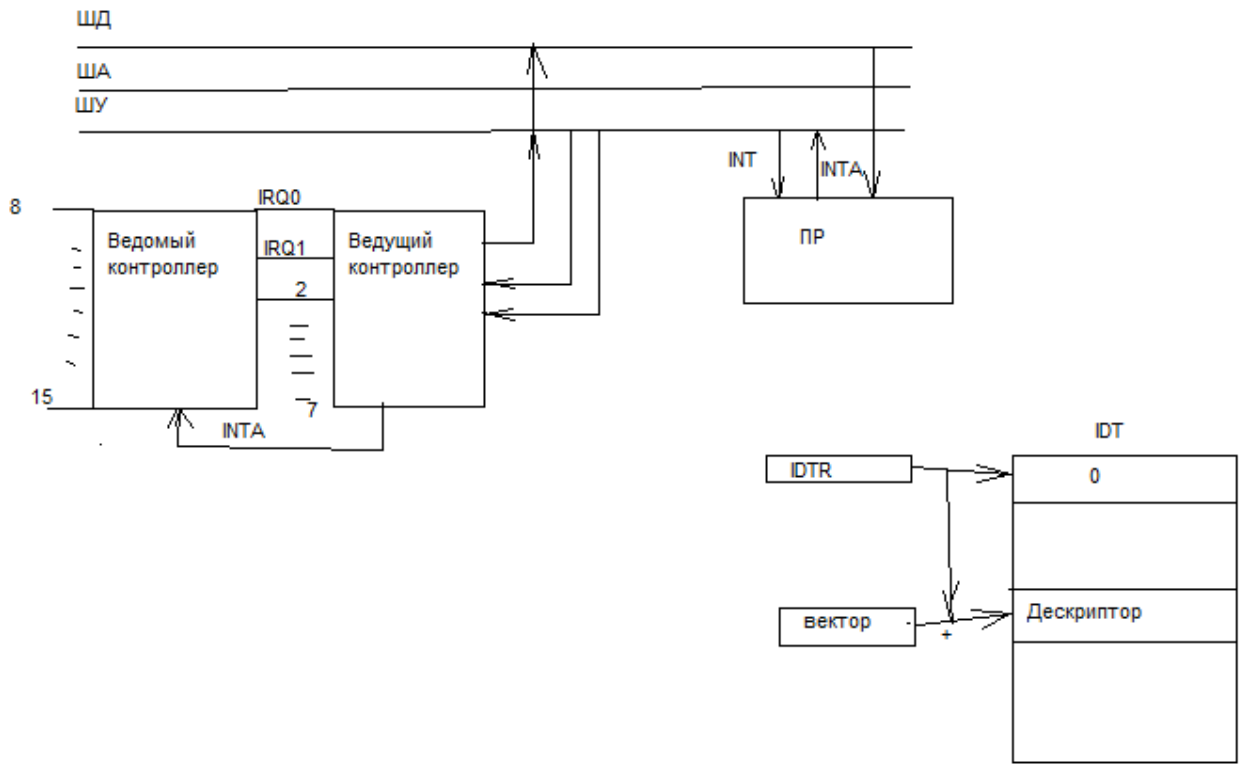
- От таймера
- От действия оператора (пример: ctrl+alt+del)
- От устройств вв/выв

В реальном и защищенном режиме работы микропроцессора обработка прерываний осуществляется принципиально разными методами.

Механизм реализации аппаратных прерываний

Когда устройство заканчивает свою работу, оно инициирует прерывание (если они разрешены ОС). Для этого устройство посылает сигнал на выделенную этому устройству специальную линию шины. Этот сигнал распознается контроллером прерываний. При отсутствии других необработанных запросов прерывания контроллер обрабатывает его сразу. Если при обработке прерывания поступает запрос от устройства с более низким приоритетом, то новый запрос игнорируется, а устройство будет удерживать сигнал прерывания на шине, пока он не обработается.

Контроллер прерываний посылает по шине вектор прерывания, который формируется как сумма базового вектора и № линии IRQ (в р.р б.в.=8h, в з.р. первые 32 строки IDT отведены под искл=> б.в.=20h). С помощью вектора прерывания дает нам смещении в IDT, из которой мы получаем точку входа в обработчик. Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, записывая определенное значение в порт контроллера прерываний. Это подтверждение разрешает контроллеру издавать новые прерывания.



Контроллер прерываний соединен каскадно. На вход IRQ0 вешается тик, на IRQ1 приходит сигнал от контроллера клавиатуры. Получив сигналконтроллер прерываний по шине управления посылает сигнал Int, сигнал приходит на соответствующую ножку процессора. Процессор постоянно проверяет наличие сигнала на ножке INT, обнаружив отсылает по щине управления сигнал INTA, контроллер получив сигнал формирует вектор прерываний (базовый вектор + номер линии) и выставляет его на шине данных. Получив этот вектор в разных режимах процессор поступает по раному. ПО этому вектору расположена таблица векторов и по адресу происходит переход на адр обработчика прерывания.

В защ режиме(IDT) все обработчики прерываний имеют соответствующий дескриптор в таблице дескрипторов прерываний.
Точное прерывание – это прерывание, оставляющее машину в строго определенном состоянии. Имеет свойства:

1. счетчик команд указывает, на команду, до которой все команды полностью выполнены.
2. не одна команда после той, на которую указывает счетчик команд – не выполнена.
3. состояние команды, на которую указывает счетчик команд – известно, причем в

перечисленных условиях не говориться, что команды после той, на который указывает счетчик команд не могут выполняться, а утверждается , что все изменения связанные с выполнением этих команд должны быть отменены до выполнения обработки прерывания.

При аппаратных прерывания счетчик команд обычно указывает на следующую команду.

При исключениях – указывает на ту команду, которая вызвала прерывание.

Неточное прерывание – прерывание, не удовлетворяющее перечисленным требованиям.

Машины с неточными прерываниями обычно выгружают в стек огромное количество данных, чтобы дать ОС возможность определить, что происходило в момент прерывания. Сохранение

больших объемов данных при каждом прерывании значительно замедляет вход процедуры обработки прерывания. Восстановление после прерывания является сложно и от сюда медленной.

2. Защищенный режим: системные таблицы – GDT, IDT, теневые регистры.

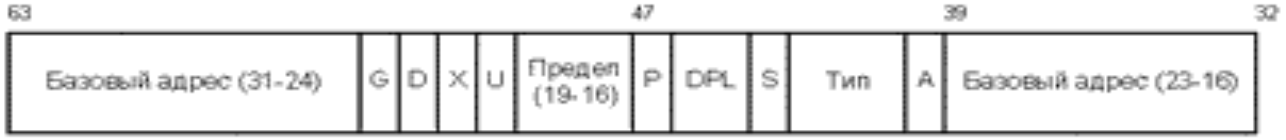
Защищенный режим (protected mode) 32-разрадный, многпоточный, многопроцессный, 4 уровня привелегий, доступно 4 Гб виртуальной памяти(для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. 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.)

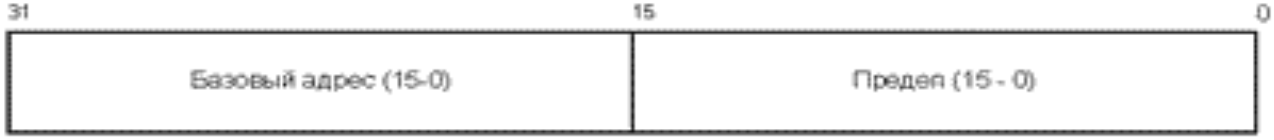
GDT (global descriptor table) – таблица, которая описывает сегменты системы, общие сегменты (сегменты ОП)
На начальный адрес GDT указывает GDT Register (32 разрядный). В системе только одна GDT.
LDT (local descriptor table) – таблица, которая описывает адресное пространство процесса. В LDT Register находится смещение до соответствующего дескриптора в GDT, описывающего сегмент, в котором находится LDT . Таблиц LDT столько, сколько процессов.
IDT (interrupt descriptor table) – таблица, предназначенная для хранения адресов обработчиков прерываний. Базовый адрес IDT помещен IDT Register.

Формат селектора (является идентификатором сегмента):

Индекс кратен 8 и является смещением в таблице дескрипторов
0 и 1 биты – Requested Privilege Level, показывает на каком уровне привилегии работаем (00-нулевой уровень)
2 бит – Table indicator, 0 – адрес в GDT, 1 – в LDT
Селектор указывает на дескриптор сегмента в таблице дескриптора.

Формат дескриптора сегмента:





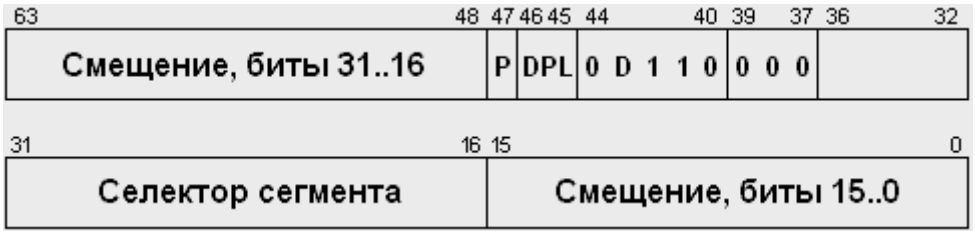
A – access – бит доступа к сегменту

Тип – 3 бита: r/w, c/cd, i. **r/w** – для символ. кода 1-чтение разрешено, 0-нет; для символ. данных 1-запись разрешена. **c/cd** – 0-для сегмента данных, 1- для сегмента стека. **i** – бит предназначения 0-сегм д. или стека, 1-кода

000b - сегмент данных, разрешено только считывание.
001b - сегмент данных, разрешено считывание и запись.
010b - сегмент стека, разрешено только считывание (не используется в практике.)
011b - сегмент стека, разрешено чтение и запись.
100b - сегмент кода, разрешено только выполнение.
101b - сегмент кода, разрешено выполнение и считывание.
110b - подчиненный сегмент кода, разрешено только выполнение.
111b - подчиненный сегмент кода, разрешено выполнение и считывание.

S – определяет, что описывает дескриптор
DPL – уровень привилегий
P – бит присутствия, используется для работы с ВП. 0 – сегмента нет в ВП, 1 – есть
D – бит разрядности операндов и адресов. 0 - 16-разрядные , 1 – 32.
G – 0 – размер сегмента задан в байтах, 1 – в страницах

Формат дескриптора прерывания:



К дескрипторам GDT и LDT мы обращаемся с помощью селекторов, к дескриптору IDT мы обращаемся по смещению, которое берем из прерывания.

Обработчик прерывания:

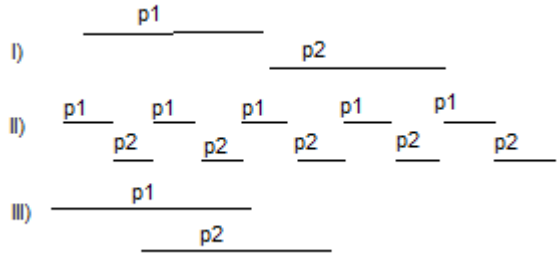
IDTR (указывает на начало IDT) + смещение из прерывания = дескриптор в IDT
Из дескриптора в IDT берем селектор
С помощью селектора узнаем с какой таблицей мы работаем.

1. Если работаем с GDT, то с помощью селектора получаем дескриптор сегмента, в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT мы получаем точку входа в обработчик прерывания.
2. Если работаем с LDT, то с помощью LDTR (в котором у нас смещение до дескриптора сегмента в GDT, в котором находится LDT) находим этот дескриптор, получаем сегмент. В этом сегменте находится нужная LDT, в ней с помощью селектора получаем дескриптор сегмента в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT получаем точку входа в обработчик прерывания.

В процессоре у каждого из сегментных регистров сопоставлен теневой регистр. Эти рег-ры доступны программисту и загружаются автоматически из таблицы дескрипторов сегментов в момент загрузки соответствующего сегментного регистра (параллельно).
Это делается, чтобы как можно реже обращаться к оперативной памяти.

1. Понятие процесса. Процесс как единица декомпозиции системы. Процессы и потоки. Типы потоков. Диаграмма состояний процесса. Планирование и диспетчеризация.

Процесс - программа в стадии выполнения. Единица декомпозиции системы, с той точки зрения, что именно ему выдаются ресурсы ОС. Может делиться на потоки, программист создает в своей программе потоки, которые выполняются квазипараллельно.



I) Последовательное выполнение программ. Все ресурсы системы выделяются этой программе.
II) Программы выполняются последовательно, но не до конца. Квазипар. процесс, система разделения времени. Однопроцессорная система.
III) Две программы могут выполняться параллельно.

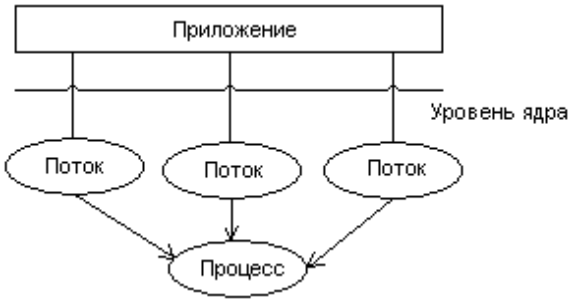
Диаграмма состояний процесса

Порождение(Рождение) – присвоение процессу строки в таблице процессов
Готовность – попадание в очередь готовых процессов – получили все необходимые ресурсы.
Блокировка(Ожидание) – ожидание необходимого ресурса. Если процесс интерактивный, то он постоянно блокируется в ожидании ввода - вывода.
(схема для мультипрограммной пакетной обработки)
Планирование – определение, в какой последовательности процессы будут получать время ЦП (FIFO, приоритеты, динамические приоритеты).
После постановки в очередь происходит диспетчеризация – получение процессами времени ЦП.

Поток – некоторая непрерывная часть кода программы, которая может выполняться параллельно с другии частями кода программы. Поток не имеет своего адресного пространства, а выполняется в адресном пространстве процесса. Потоки бывают разные: потоки на уровне ядра и на уровне пользователя (о них ОС не знает). Управлением пользовательских потоков занимается пользовательская библиотека.

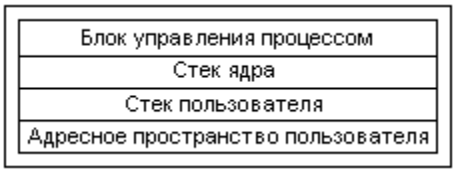
1) Потоки на уровне пользователя (прикладные потоки)

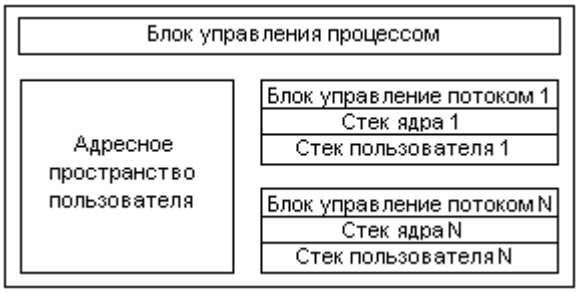
2) Потоки на уровне ядра



Планирование и деспетчерезация
Диспетчеризация – выделение процессу процессорного времени.
Вытеснение основано на системе с приоритетами. Если все процессы равноправны, то вытеснения нет. Процессорное время передается процессу с более высоким приоритетом. Статические назначаются в начале и с течением времени не меняются.
Динамические меняются в течение жизни.

Планирование процессов – управление распределением ресурсов центральным процессором между конкурирующими процессами, путем передачи упраления согласно некоторой стратегии планирования.
Планировщик – программа, которая отвечает за управление использования совместного ресурса.
Доступ к ресурсу, который используется совместно, предоставляется с учетом двух требований
1. Необходимо убедиться, что процесс не будет поврежден сам и не повредит другие процессы.
2. Выбор между несколькими процессами, при возможности доступа любого к ресурсу, осуществляется некоторыми алгоритмами планирования.
Планирование:
- без переключения и с переключением
- без вытеснения и с вытеснением
- без приоритетов и с приоритетами (относительные или абсолютные, статические или динамические)

1) однопоточная модель процесса 

2) многопоточная модель процессора 

2.Обеспечение монопольного доступа к разделяемым данным в задаче "писатели-читатели" в ОС Windows .

```
CVal::~CVal()
{
		CloseHandle(canRead);
		CloseHandle(canWrite);
		CloseHandle(writeLockMutex);
}

void CVal::startRead()
{
		InterlockedIncrement(&waitingReadersCount);

		if(WaitForSingleObject(writeLockMutex, 0) == WAIT_TIMEOUT ||
waitingWritersCount > 0)
			WaitForSingleObject(canRead, INFINITE);

		InterlockedDecrement(&waitingReadersCount);
		InterlockedIncrement(&ReadersCount);
		ResetEvent(canWrite);
		SetEvent(canRead);
}

void CVal::stopRead()
{
		InterlockedDecrement(&ReadersCount);
		if(ReadersCount == 0)
			SetEvent(canWrite);
}

void CVal::startWrite(int number)
```

```
{
		InterlockedIncrement(&waitingWritersCount);

		if(ReadersCount > 0 || WaitForSingleObject(writeLockMutex, 0) == WAIT_TIMEOUT)
			WaitForSingleObject(canWrite, INFINITE);

		ReleaseMutex(writeLockMutex);
		WaitForSingleObject(writeLockMutex, INFINITE);
		ResetEvent(canWrite);
		ResetEvent(canRead);
		InterlockedDecrement(&waitingWritersCount);
}

void CVal::stopWrite(int number)
{
		if(waitingReadersCount > 0)
			SetEvent(canRead);
		else
			SetEvent(canWrite);

		ReleaseMutex(writeLockMutex);
}

int CVal::getValue()
{
		return value;
}

void CVal::setValue(int newValue)
{
		value = newValue;
}
```

Main.cpp

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

#include "cval.h"
#include "preset.h"

CVal Values[VALUES_COUNT];

DWORD WINAPI writer(LPVOID lpParameter)
{
		int number = *(int*)lpParameter;
		int valueNumber = 0;
		int loops = 0;
		int val;
		while(1)
```

```
		{
			CVal& cvalue = Values[valueNumber];
			cvalue.startWrite(number);
			cvalue.setValue(val = (cvalue.getValue() + 1));
			printf("Writer %d: set value [%d] to %d\n", number, valueNumber, val);
			cvalue.stopWrite(number);

			valueNumber++;
			if(valueNumber >= VALUES_COUNT)
			{
				valueNumber = 0;
				loops++;
				if(loops > MAX_LOOPS)
					ExitThread(0);
			}
			Sleep(100);
		}
}

DWORD WINAPI reader(LPVOID lpParameter)
{
		int number = *(int*)lpParameter;
		int valueNumber = 0;
		int loops = 0;
		while(1)
		{
			CVal& cvalue = Values[valueNumber];
			cvalue.startRead();
			printf("Reader %d: reads value [%d] is %d\n", number, valueNumber,
cvalue.getValue());
			cvalue.stopRead();

			valueNumber++;
			if(valueNumber >= VALUES_COUNT)
			{
				valueNumber = 0;
				loops++;
				if(loops > MAX_LOOPS)
					ExitThread(0);
			}
			Sleep(100);
		}
}
```

1. Взаимоисключение и синхронизация процессов и потоков. Семафоры: определение, виды, примеры использования.

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов.Она связана с потерей доступа к параметрам из-за их некоррректного разделения.

Критический ресурс - разделенная переменная,к которой обращаются разные процессы.

Критическая секция - строки кода,в кот происходит обращение к критическому ресурсу.

Необходимо обеспечить монопольный доступ процесса к критическому ресурсу до тех пор пока процесс его не освободит. Т.е. чтобы не могли одновременно войти в крит. секцию.

Все алгоритмы программной реализации обобщил Дейкстра,введя понятие семафора.
Активное ожидание на процессоре – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом) Активное ожидание на процессоре является неэффективным использованием процессорного времени.

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критичские секции - ок

2. Возможно **бесконечное откладывание (зависание) –** ситуация, когда разделённый ресурс снова захватывается тем же процессом.

3. **Тупик (deadlock, взаимоблокировка) –** ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом

Семафоры

(устраняют активное ожидание на процессоре)

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции:

P (от датск. passeren - пропустить) и V(от датск. vrygeven - освбодить).

Операция V(S): означает увеличение значения S на 1 одним неделимым действием (последовательнось непрерывных действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0, то V(S) приведёт к V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожидающего на семафоре.

Операция P(S): означает декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить операция P, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой процесс, выполняющий операцию V(S).

S может быть изменена только операциями P(S) и V(S). Это и есть защищённость S.

P(S) и V(S) есть неделимые (атомарные) операции.

Суть: процесс пытающий выполнить операцию P(S) блокируется, становится в очередь ожидания данного семафора, освобождает его другой процесс, который выполняет V(S). Таким образом исключается активное ожидание.

Семафоры бывают:

1) бинарные (S принимает значения 0 и 1)
2) считающие (S принимает значения от 0 до n)
3) множественные (массив считающих семафоров)

Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо привелегированный процесс.

При проверке флага мы не переходим в режим ядра.

При вызове команды test-and-set (семафор) переходим

Процесс блокируется на семафоре, если он не находится в очереди готовых процессов.

Особенность множественных семафоров – операции могут выполняться сразу над всеми семафорами данного множества.

Изменение переменной S можно рассматривать как событие в системе.

Примеры использования:

Производство-потребление – считающие – буфер пуст и полон, один бинарный (монитор – кольцевой буфер)

Читатели-писатели – монитор Хоара

Обедающие философы – множественные семафоры

2. Режимы работы процессоров Intel Pentium. Прерывания в защищенном режиме.

1. **Реальный режим** (или режим реальных адресов) - это название было дано прежнему способу адресации памяти после появления 286-го процессора, поддерживающего защищённый режим.
2. **Защищенный режим** - Режим защиты памяти. Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Основная мысль сводится к формированию таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

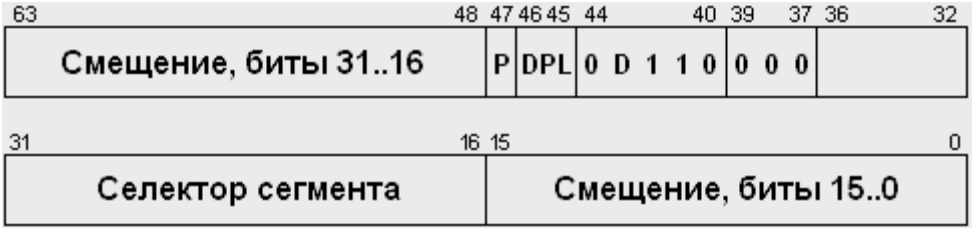
32-разрадный, многпоточный, многопроцессный, 4 уровня привелегий, доступно 4 Гб виртуальной памяти(для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. Создан для работы нескольких независимых 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.)
3. **Виртуальный режим** - В процессоре i386 компания Intel учла необходимость лучшей поддержки реального режима, потому что программное обеспечение времени его появления не было готово полностью работать в защищенном режиме. Поэтому, например, в i386, возможно переключение из защищенного режима обратно в реальный (при разработке 80286 считалось, что это не потребуется, поэтому на компьютерах с процессором 80286 возврат в реальный режим осуществляется схемно - через сброс процессора).
В качестве дополнительной поддержки реального режима, i386 позволяет задаче (или нескольким задачам) защищенного работать в виртуальном режиме — режиме эмуляции режима реального адреса (таким образом в переключении в реальный режим уже нет необходимости). Виртуальный режим предназначается для одновременного выполнения программы реального режима (например, программы DOS) под операционной системой защищенного режима.

Выполнение в виртуальном режиме практически идентично реальному, за несколькими исключениями, обусловленными тем, что виртуальная задача выполняется в защищенном режиме:

- виртуальная задача не может выполнять привилегированные команды, потому что имеет наинизший уровень привилегий
- все прерывания и исключения обрабатываются операционной системой защищенного режима (которая, впрочем, может инициировать обработчик прерывания виртуальной задачи)

Механизм реализации аппаратных прерываний
Когда устройство заканчивает свою работу, оно инициирует прерывание (если они разрешены ОС). Для этого устройство посылает сигнал на выделенную этому устройству специальную линию шины. Этот сигнал распознается контроллером прерываний. При отсутствии других необработанных запросов прерывания контроллер обрабатывает его сразу. Если при обработке прерывания поступает запрос от устройства с более низким приоритетом, то новый запрос игнорируется, а устройство будет удерживать сигнал прерывания на шине, пока он не обработается.
Контроллер прерываний посылает по шине вектор прерывания, который формируется как сумма базового вектора и № линии IRQ (в р.р б.в.=8h, в з.р. первые 32 строки IDT отведены под искл=> б.в.=20h). С помощью вектора прерывания дает нам смещении в IDT, из которой мы получаем точку входа в обработчик. Вскоре после начала своей работы процедура обработки прерываний подтверждает получение прерывания, записывая определенное значение в порт контроллера прерываний. Это подтверждение разрешает контроллеру издавать новые прерывания.
Формат дескриптора прерывания:



К дескрипторам GDT и LDT мы обращаемся с помощью селекторов, к дескриптору IDT мы обращаемся по смещению, которое берем из прерывания.

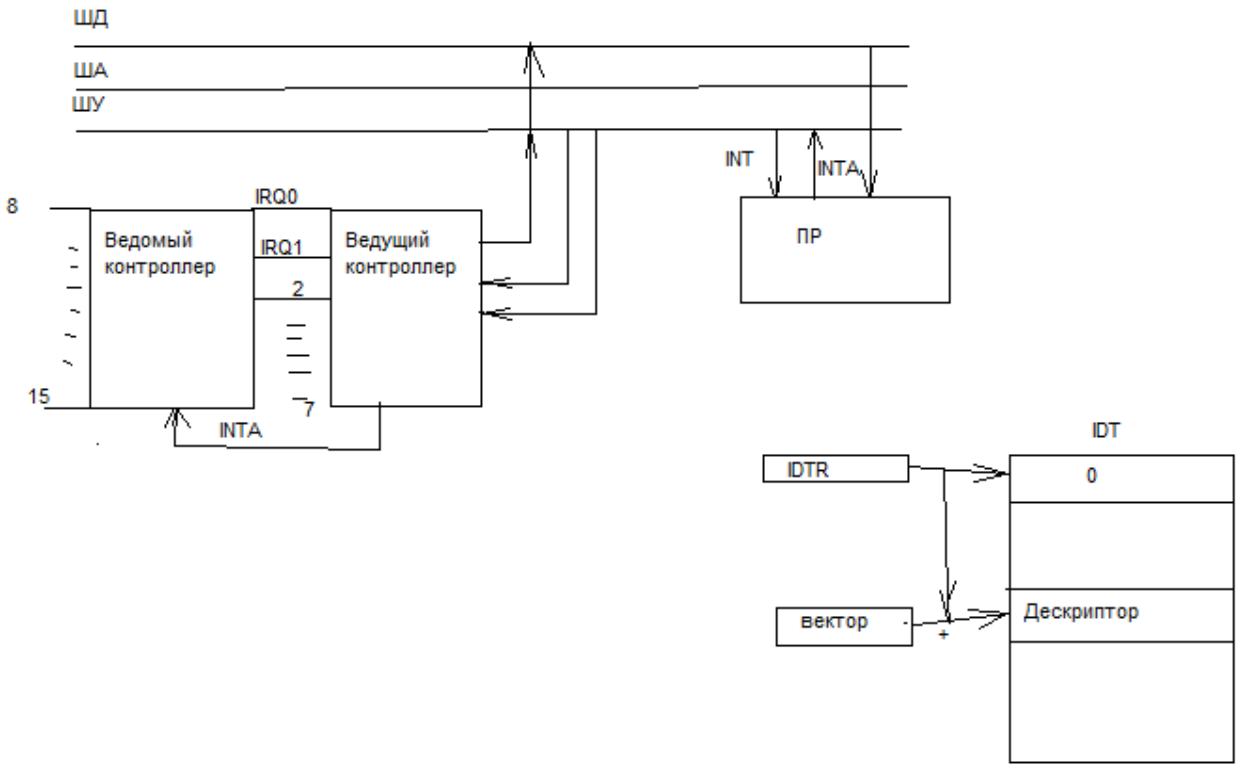
Обработчик прерывания:

IDTR (указывает на начало IDT) + смещение из прерывания = дескриптор в IDT

Из дескриптора в IDT берем селектор

С помощью селектора узнаем с какой таблицей мы работаем.

1. Если работаем с GDT, то с помощью селектора получаем дескриптор сегмента, в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT мы получаем точку входа в обработчик прерывания.
2. Если работаем с LDT, то с помощью LDTR (в котором у нас смещение до дескриптора сегмента в GDT, в котором находится LDT) находим этот дескриптор, получаем сегмент. В этом сегменте находится нужная LDT, в ней с помощью селектора получаем дескриптор сегмента в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT получаем точку входа в обработчик прерывания.



Контроллер прерываний соединен каскадно. На вход IRQ0 вешается тик, на IRQ1 приходит сигнал от контроллера клавиатуры. Получив сигналконтроллер прерываний по шине управления посылает сигнал Int, сигнал приходит на соответствующую ножку процессора. Процессор постоянно проверяет наличие сигнала на ножке INT, обнаружив отсылает по щине управления сигнал INTA, контроллер получив сигнал формирует вектор прерываний (базовый вектор + номер линии) и выставляет его на шине данных. Получив этот вектор в разных режимах процессор поступает по раному. ПО этому вектору расположена таблица векторов и по адресу происходит переход на адр обработчика прерывания.

В защ режиме(IDT) все обработчики прерываний имеют соответствующий дескриптор в таблице дескрипторов прерываний.

1. Методы организации ввода-вывода: программируемый, с прерываниями и прямой доступ к памяти.

Способы организации IO и виды IO

В современных ОС существуют 3 метода:

1. программируемый IO
2. IO с использованием прерываний
3. прямой доступ к памяти

1) Программируемый IO. В процессе выполнения программы встречается команда IO: периодический опрос нужных битов контроллера. Способ опроса – rolling. На ЦП возлагается непосредственное управление операцией IO: опрос, пересылка команд, передача данных. Для реализации данного способа необходимо иметь следующий набор команд:

- команды управления для инициализации работы внешнего устройства
- команды анализа состояния для проверки битов состояния контроллера
- команды передачи (из регистров ЦП в регистры устройства и наоборот)

2) В конечном итоге будет выполняться передача из ЦП в контроллер команды соответствующей команды. После ЦП отключается от управления операцией IO и переходит на выполнение другой работы. Когда контроллер будет готов обменяться данными с ЦП, он пошлет прерывание. Процесс с точки зрения контроллера: Контроллер получает от ЦП команду (например, read). Он переходит к считыванию данных со своего устройства. Как только эти данные окажутся в регистрах контроллера, посылается сигнал прерывания. После получения вектора центральным процессором, контроллер посылает по шине данных данные из своих регистров.

С точки зрения процессора: Процессор генерирует команду read. ЦП сохраняет содержимое программного счётчика команд, других регистров и переходит к выполнению другой работы. При поступлении сигнала прерывания ЦП сохраняет данные о выполняемой задаче и переходит к выполнению обработчика прерывания, считывает слова из регистров контроллера и заносит их в память(через свои регистры). По завершению восстанавливает свой PSW - контекст и продолжает прерванную работу.

Таким образом, каждое слово, которое передаётся между памятью и устройством, должно пройти через регистры процессора.

Минусы такого подхода:
Каждое прерывание вызывает остановку выполнения текущих процессов и сохранение аппаратного контекста.
Например, принтер: процесс запросивший печать строки на принтер, оказывается заблокированный на все время, когда принтер напечатает символ и будет готов принять он инициализирует прерывание, это вызовет приостановку текущего процесса и сохранения его контекста. Прерывание происходит при печатанье каждого символа. Очень много времени тратит на обработку часто возникающих прерываний.

3) Прямой доступ к памяти. Для передачи больших объёмов данных используется именно этот способ. Для реализации прямого доступа к памяти (далее ПДП) в состав компьютера включается контроллер ПДП, но иногда функции ПДП возлагаются на контроллер IO. Контроллер ПДП способен формировать адрем в некотором диапазоне, который определяется размером данных. То есть некоторый блок данных может быть передан без участия регистров ЦП. Контроллер ПДП берёт на себя управление шиной данных.

Если требуется считать или записать блок данных, то ЦП генерирует команду для контроллера ПДП, передавая ему:

1) тип команды (read или write)
2) адрес устройства
3) начальный адрес блока в ОП, используемого для чтения или записи
4) количество байтов или слов

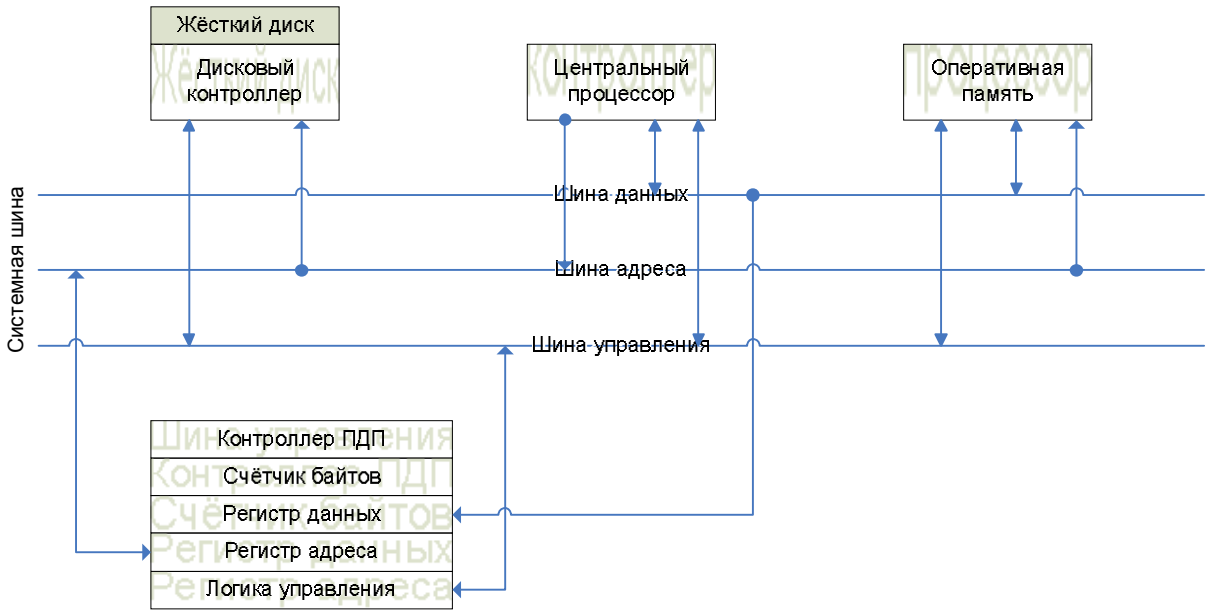
Передавая управление контроллеру ПДП, ЦП отключается от управления шиной данных, и продолжает работу, несвязанную с IO. Контроллер ПДП передаёт данные по словам или байтам и по окончании передачи посылает сигнал прерывания, сообщающий о завершении IO.

Очевидно, что ЦП не может отключиться от шины данных на такое долгое время. Он отключается на время, равно циклу шины данных, то есть если ЦП потребуется IO до завершения цикла шины данных (уже после отключения от неё), то ему придётся ждать. Таким образом, во время использования прямого доступа к памяти, замедляется выполнение центральным процессором команд, требующих передачи данных.

Плюсы ПДП:
Сокращение числа прерываний

Минусы ПДП:
Контроллер ПДП отстаёт по быстродействию от ЦП. Поэтому, если контроллер ПДП не может поддерживать текущую скорость IO, то ПДП не используется, а используются 2 предыдущих способа.

Контроллер ПДП получает доступ к системной шине независимо от ЦП.



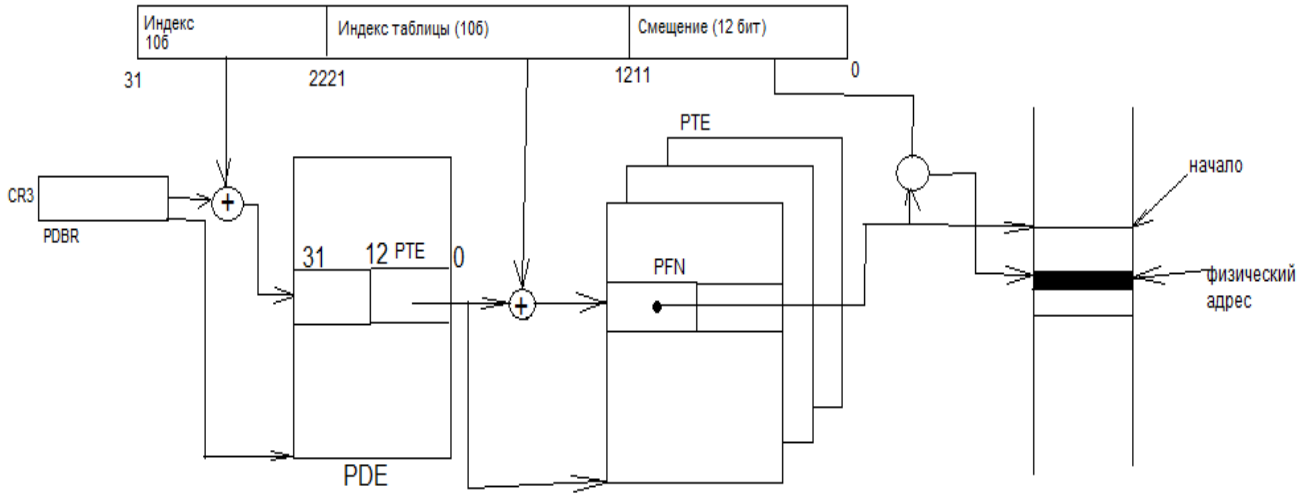
2. Защищенный режим: EMS, преобразование адреса при страничном преобравании в процессорах Intel.

Отображаемая память **EMS** (Expanded Memory Specification) - программная спецификация использования дополнительной памяти DOS-программами реального режима.

Expanded Memory Specification.(Expanded – растянутая, в смысле что вытесняется на жесткий диск). Определяет способ управления пейджингом, в какие области он выполняется и определяет схему преобразования виртуального адреса в физический.

Страничное преобразование может быть включено или не включено. Сегментное преобразование включено всегда. Если страничное преобразование включено, то используется пейджинг страниц, а не свопинг.
Оперативная память делится на кадры или фреймы. Размер сегмента кратен размеру страницы (4Кб)
Система EMS в основном предназначена для хранения данных - для исполняемого в данный момент программного кода она неудобна, поскольку требует программного переключения страниц через каждые 16 Кбайт. программа через диспетчер назначает отображение требуемой логической страницы из выделенной ей области дополнительной памяти на выбранную физическую страницу, расположенную в области UMA.

При страничном преобразовании адрес делитсяна три части, 12 бит смещение, индекс таблицы 10 бит, индекс оглавления 10бит.
PDE – page directory entry (каталог страниц)
PFN – page frame number – указывает базовый адрес таблицы страниц
PTE – page table entry
Каталог страниц один на процесс. Может содержать 1024 дескриптора.
PDBR – содержит адрес каталога страниц для текущего процесса



Если преобразование занимает несколько сегментов, значит должно быть столько таблиц страниц, сколько у него сегментов
PDE – page directory entry
PFN – page frame number
PTE – page table entry

1. Управление памятью. Распределение памяти сегментами по запросам: стратегии выделения памяти, достоинства и недостатки.

Сегмент – размер явл размером программного кода, логическая единица деления памяти

Управление памятью в современном компьютере – иерархия памяти в зависимости от удаления от процессора.
Разделяются на две задачи управления: верт.(передача информации с одного типа на другой), гориз. (управление данным типом памяти).
Управление виртуальной памятью:
Способы распределения:
1. Стр по запросам
2. Сегм по запросам
3. Сегменты подел по стр по запросам

Виртуальная паямять – память размер которой превосходит размер реального физ адресного пространства.

Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система выбирает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки.

Каждый сегмент описывается дескриптором сегмента.

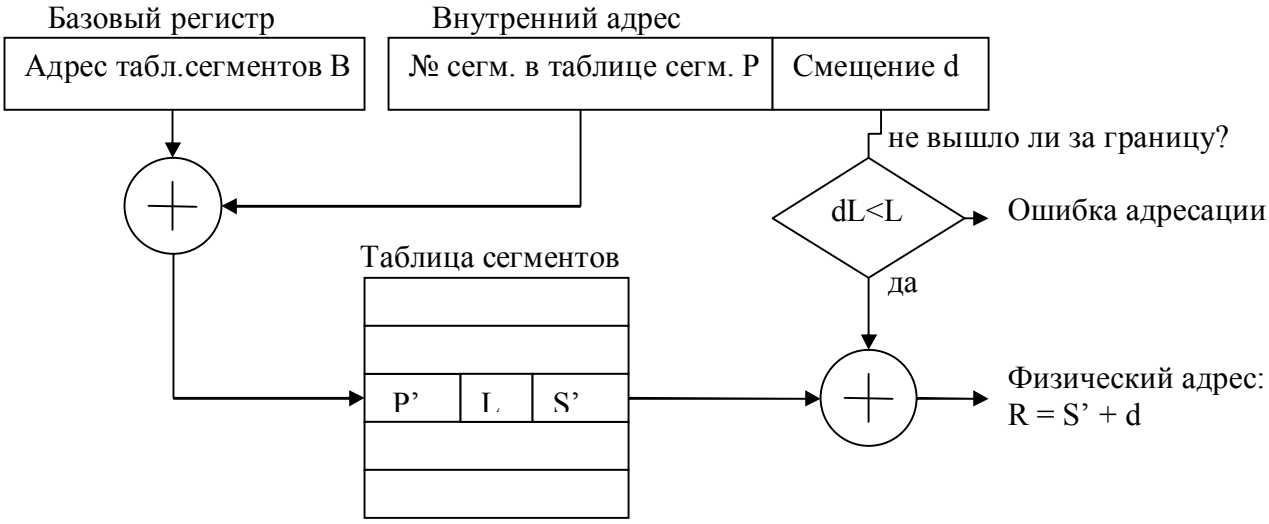
ОС строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в ОП или внешней памяти в дескрипторе отмечает его текущее местоположение (бит присутствия).

Дескриптор содержит поле адреса, с которого сегмент начинается и поле длины сегмента. Благодаря этому можно осуществлять **контроль**
1) размещения сегментов без наложения друг на друга
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) Þ защита.

Достоинства:
1) общий объем виртуальной памяти превосходит объем физической памяти
2) возможность размещать в памяти как можно больше задач (до определенного предела) Þ увеличивает загрузку системы и более эффективно используются ресурсы системы

Недостатки:
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) фрагментация;
3) потери памяти на размещение дескрипторных таблиц
4) потери процессорного времени на обработку дескрипторных таблиц.



Стратегии замещения (в лекциях страниц, но на сегменты по-моему то же самое):

1. Выгрузка случайного сегмента.
2. Алгоритм FIFO: замещать сегмент который дольше всего находится в памяти. Способы: временная метка либо ведется связный список. Этот алгоритм исключает возможность выгрузки только что загруженного сегмента, но не исключает выгрузку часто использующегося.
3. Алгоритм Last Recently Used: замещаем наименее используемый в последнее время. Приближен к оптимальному. Необходимо или изменять временную метку при каждом обращении, или при кажд. обращении помещать этот сегмент в начало списка.
4. Алгоритм NUR (Not Used Recently): Приписывание каждому сегменту бита обращения. При загрузке и при каждом обращении бит обращения 1. Периодически все биты обращений сбрасываются в 0. Когда нужно заместить, ищется сегмент с нулебым битом обращения.
5. Алгоритм LFU (Least Frequency Used): Контролируется частота обращения. При переполнении счетчик сбрасывается. НО! Наименее используемым может оказаться только что загруженный сегмент.

2.Режимы работы компьютера IBM PC, кэши TLB и данных.

Реальный режим Это режим работы первых 16-битовых микропроцессоров с 20ти разрадной шиной адреса и диапазон адресов памяти ограничен одним мегабайтом. Наличие его обусловлено тем, что необходимо обеспечить в новых моделях микропроцессоров функционирование программ, разработанных для старых моделей.

Защищенный режим (protected mode) 32-разрадный, многпоточный, многопроцессный, 4 уровня привелегий, доступно 4 Гб виртуальной памяти(для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. 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
В режим V86 процессор может перейти из защищённого режима, если установить в регистре флагов EFLAGS бит виртуального режима (VM-бит). Номер бита VM в регистре EFLAGS - 17.
Когда процессор i80386 находится в виртуальном режиме, для адресации памяти используется схема <сегмент:смещение>, размер сегмента составляет 64 килобайта, а размер адресуемой в этом режиме памяти - 1 мегабайт.

Виртуальный режим - это не реальный режим процессора i8086, имеются существенные отличия. Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима.
В виртуальном режиме используется трансляция страниц памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство, каждое размером в 1 мегабайт.
Все задачи виртуального режима обычно выполняются в третьем, наименее привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор автоматически переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему, работающую в защищённом режиме.

1. Реальный режим (или режим реальных адресов) - это название было дано прежнему способу адресации памяти после появления 286-го процессора, поддерживающего защищённый режим.
2. Защищенный режим - Режим защиты памяти. Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Основная мысль сводится к формированию таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

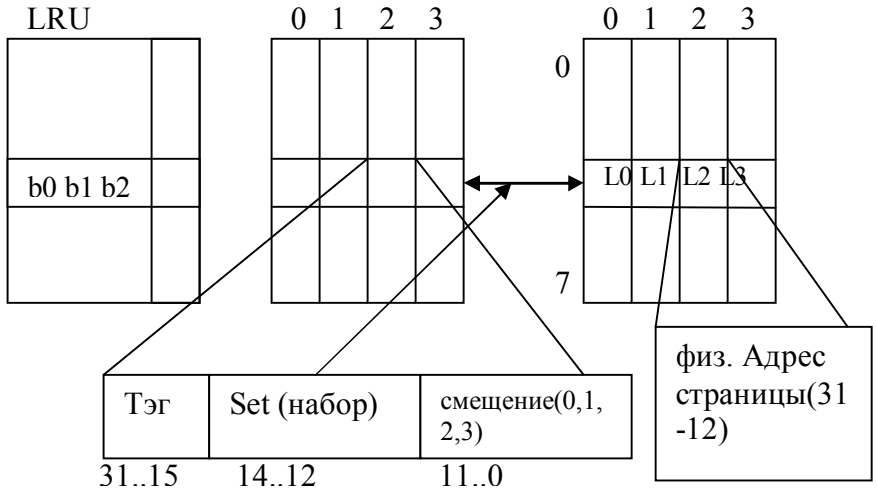
3. Виртуальный режим - В процессоре i386 компания Intel учла необходимость лучшей поддержки реального режима, потому что программное обеспечение времени его появления не было готово полностью работать в защищенном режиме. Поэтому, например, в i386, возможно переключение из защищенного режима обратно в реальный (при разработке 80286 считалось, что это не потребуется, поэтому на компьютерах с процессором 80286 возврат в реальный режим осуществляется схемно - через сброс процессора).
В качестве дополнительной поддержки реального режима, i386 позволяет задаче (или нескольким задачам) защищенного работать в виртуальном режиме — режиме эмуляции режима реального адреса (таким образом в переключении в реальный режим уже нет необходимости). Виртуальный режим предназначается для одновременного выполнения программы реального режима (например, программы DOS) под операционной системой защищенного режима.
Выполнение в виртуальном режиме практически идентично реальному, за несколькими исключениями, обусловленными тем, что виртуальная задача выполняется в защищенном режиме:

- виртуальная задача не может выполнять привилегированные команды, потому что имеет наинизший уровень привилегий
- все прерывания и исключения обрабатываются операционной системой защищенного режима (которая, впрочем, может инициировать обработчик прерывания виртуальной задачи)

TLB – является ассоциативным буфером, в котором хранится физ. адрес страницы, к кот. были последние обращения.

TLB представляет собой четырех-канальную ассоциативную память.В блоке данных находится восемь наборов по четыре элемента данных в каждом. Элемент данных в **TLB** состоит из 20 битов старшего порядка физического адреса. Эти 20 битов могут интерпретироваться как базовый адрес страницы, который по определению имеет 12 очищенных битов младшего порядка.

TLB транслирует линейный адрес в физический и работает только со старшими 20 битами каждого из них; младшие 12 битов (представляющие собой смещение в странице) одинаковы как для линейного адреса, так и для физического.
Блоку элементов данных соответствует блок элементов достоверности, аттрибутов и тега (признака). Элемент тега состоит из 17 старших битов линейного адреса. При трансляции адреса процессор использует биты 12, 13 и 14 линейного адреса для выбора одного из восьми наборов, а затем проверяет четыре тега из этого набора на соответствие старшим 17 битам линейного адреса. Если соответствие найдено среди тегов выбранного набора, а соответствующий бит достоверности равен 1, то линейный адрес транслируется заменой старших 20 битов на 20 битов соответствующего элемента данных.

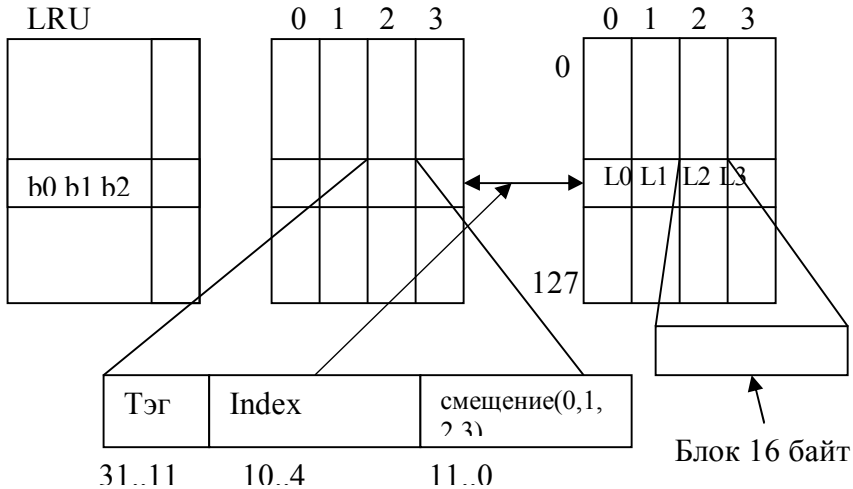


Каждому набору соответствует три бита **LRU**: они отслеживают используемость данных в наборе и проверяются при необходимости в новом элементе (а также следят за достоверностью всех элементов в наборе). b0 b1 b2 – для реализации алгоритма псевдо-LRU, послед бит – бит достоверности.

1. Если послед обращ было к L0, L1, то b0 уст в 1, если обращ было к L2, L3, то b0 уст в 0
2. Если послед обращ в паре L0-L1 было к L0, то b1- в 1, если к L1, то b1 -0
3. Прим тоже самое с b2, только в паре L3-L4.

Про кэш данных:

Имел 128 строк, в 486 процессоре к кешу данных относились команды и данные, все что относится к шине данных – кеш данных.



1. Классификация структур ядер ОС. Особенности ОС с микроядром. Модель клиент-сервер. Три состояния процесса при передаче сообщений. Достоинства и недостатки микроядерной архитектуры.

ОС разбивается на несколько уровней, причем обращение через уровень невозможно (непрозрачный интерфейс).
В качестве ядра выделяется самый низкий уровень распределения аппаратных ресурсов процессам самой ос (непосредственное обращение с аппаратурой).

Существует два типа структур ядер:

1. Монолитное ядро

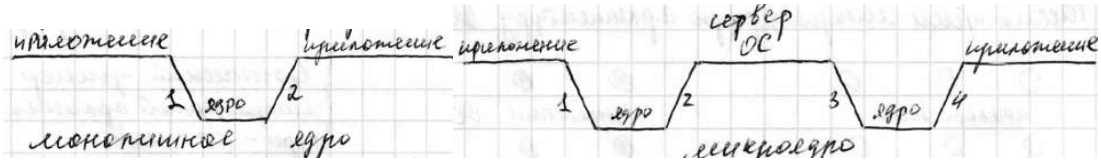
− ядро – это все, что выполняется в режиме ядра
− ядро представляет собой единую программу с модульной структурой (выделены функции, такие как планировщик, файловая система, драйверы, менеджеры памяти)
− при изменении к.-л. функции нужно перекомпилировать все ядро
− такие ОС делятся на две части – резидентную и нерезидентную

Пример – Unix, хотя она имеет минимизированное ядро(часть функций вынесены в shell).

2. Микроядро

Специальный модуль нижнего уровня, который обеспечивает работу с аппаратурой на самом низком уровне и базовые функции работы с процессами. Остальные компоненты – самостоятельные процессы, которые могут работать в разных ядерных пространствах.
Общение между компонентами происходит с помощью сообщений через адресное пространство микроядра.
Микроядерная архитектура основана на модели клиент-сервер (например, ОС Mach, Hurd и Win2k(но не в классическом понимании))
(К серверам ОС относятся: сервер файлов, процессов, безопасности, виртуальной памяти)

В микроядре минимум 4 исключения



Модель клиент-сервер
Система рассматривается как совокупность двух групп процессов

- ✓ процессы-серверы, предоставляющие набор сервисов
- ✓ процессы-клиенты, запрашивающие сервисы

Принято считать, что данная модель работает на уровне транзакций (запрос и ответ представляет неделимая операция)
3 состояния процесса при передаче сообщения(протокол обмена):

- ✓ запрос: клиент запрашивает сервер для обработки запроса
- ✓ ответ: сервер возвращает результат операции
- ✓ подтверждение: клиент подтверждает прием пакета от сервера

Для обеспечения надежности обмена в протокол обмена могут входить следующие действия:

- ✓ сервер доступен? (запрос клиента)
- ✓ сервер доступен (ответ сервера)
- ✓ перезвоните (ответ сервера о недоступности)

✓ адрес неверен (процесса с данным № нет в системе)

Достоинства и недостатки микроядерной архитектуры:

+ Высокая степень модульности ядра *ОС*, что существенно упрощает добавление в него новых компонентов. В микроядерной ОС можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д., т.о. упрощается процесс отладки компонентов ядра.

+ Компоненты ядра *ОС* принципиально не отличаются от пользовательских программ, поэтому для их отладки можно применять обычные средства.

+ Повышается надежность системы, поскольку ошибка на уровне непривилегированной программы менее опасна, чем отказ на уровне режима ядра.

− *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.

2. Unix: а) основные концепции;б) команды fork(), wait(), exec(), pipe(), signal().

Базовое понятие – процесс.

Процесс может находиться в двух состояниях – задача (процесс выполняет собственный код) и система (выполняет реентерабельный код ос)

Процесс может создавать любое число процессов (системный **fork())**

Строгая иерархия в отношении предок – потомок.

Т.к. Unix система разделения времени, то существует понятие терминала. Процесс, запустивший терминал имеет PID = 1 Все процессы, запущенные на этом терминале, являются его потомками.

Все процессы в Unix объединены в группы, процессы одной группы получают одни и те же сигналы.

В результате вызова **fork** создается процесс-потомок, который наследует адресное пространство предка и все открытые файлы (фактически наследует код)

В Unix все рассматривается как файл (файлы, директории, устройства)

В состав этой ОС включен системный вызов **exec**, который заменяет адресное пространство на адресное пространство, которое вызывается в этом системном вызове. Бывает шести видов: execlp, execvp,execl,execv,execle, execve.
Например, execl(”/bin/ps”, ”ps”, 0)

Системный вызов **wait(&status).** Процесс приостанавливается до тех пор, пока один из непосредственно порожденных им процессов не завершится

pipe() Программный канал – это специальный буфер, который создается в системной области памяти. Информация в канал записывается по принципу FIFO и не модифицируется. Предок и потомок могут обмениваться сообщениями с помощью неименованного программного канала.

signal() Для изменения хода выполнения программы. Необходимо написать свой обработчик (в зависимости от того был получен сигнал или нет выполняются разные действия)

1. Виртуальная память: сегментно-страничное распределение памяти по запросам. Достоинства и недостатки.

Virtual Memory – система при которой рабочее пространство процесса частично располагается в основоной памяти и частично во вторичной. При обращении к какой либо памяти, система аппаратными средствами определяет присутствует ли область физической памяти, если отсутствует, то генерируется прерывание, это позволяет супервизору передать необходимые данные из вторичной в основную.

Виртуальная память – память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или педжинга, т.е. для временного хранения областей памяти.

Подходы к реализации управления виртуальной памятью:

1. страничное распределение памяти по запросам.
2. сегментное распределение памяти по запросам
3. сегментно - страничное распределение памяти по запросам

Сегмент представляется в виде совокупности страниц, что позволяет устранить проблемы, связанные с перекомпоновкой и ограничением размера сегмента. Впервые такой подход был применен в системе разделения времени TSS для IBM 370 и в системе MULTICS для Honeywell 6180.
В системе с сегментно-страничной организацией применяется трехкомпонентная (трехмерная) адресация. Виртуальный адрес определяется как упорядоченная тройка **v=(s,p,d)**, где **s**- номер сегмента, **p**- номер страницы в сегменте, **d**- смещение в странице, ко которому находится нужный элемент.

На рис показана схема динамического преобразования адресов с применением ассоциативно-прямого отображения. Работа данной схемы динамического преобразования адресов аналогична работе ранее рассмотренных схем. При отсутствии сегмента в памяти происходит прерывание по отсутствию сегмента; отсутствие страницы вызовет страничное прерывание. Если виртуальный адрес выходит за границы сегмента, то произойдет прерывание по выходу за границы сегмента. Если осуществляется попытка несанкционированного доступа к сегменту, то произойдет прерывание по защите сегмента. Ассоциативная память (или аналогичная по быстродействию кэш-память), которые часто называются TLB - буфер быстрой переадресации, обеспечивает более эффективную работу механизма динамического преобразования адресов. При полностью прямом преобразовании каждое обращение к адресуемому элементу потребовало бы трех циклов обращения к памяти и реальное быстродействие системы составило бы лишь приблизительно треть от номинального значения. Интересно, что использование только восьми или шестнадцати ассоциативных регистров позволяет во многих системах получить показатели быстродействия в размере 90 и более процентов от реального быстродействия ЦП. Таким образом, благодаря TLB накладные расходы, вызываемые механизмом преобразования адресов, удается снизить до менее чем 10%.

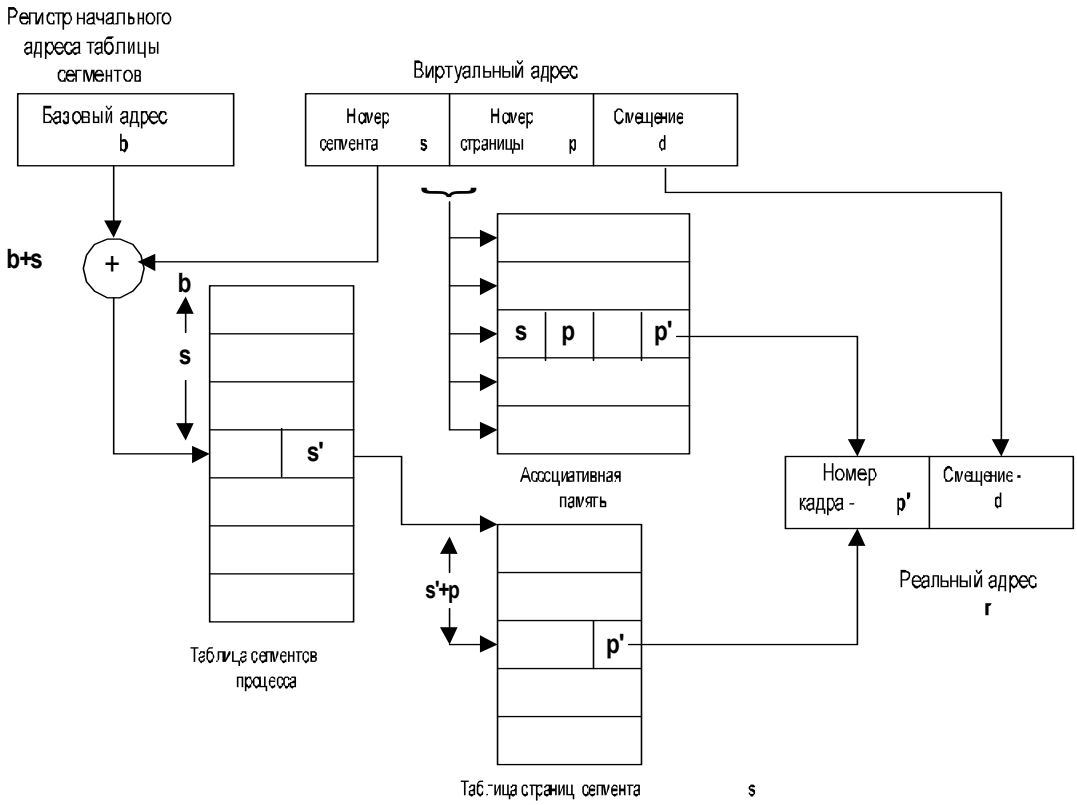


Рис.6.27

На рис.6.28 показана структура таблиц, которые ведутся и обрабатываются ОС при сегментно-страничном распределении памяти по запросам. На самом верхнем уровне находится таблица процессов, в которой для каждого процесса, выполняющегося в системе, выделена строка. Строка таблицы процессов указывает на таблицу сегментов этого процесса.
Каждая строка таблицы сегментов указывает на таблицу страниц соответствующего сегмента. Каждая строка таблицы страниц указывает либо на страничный кадр, к котором размещается данная страница, либо на адрес внешней памяти, где хранится эта страница.
Если разместить все указанные таблицы в основной памяти, то они займут значительный объем. Компромиссным решением является размещение частей таблиц во внешней памяти. Коллективное использование в данной схеме реализуется указанием в таблицах сегментов адреса одной и той же таблицы страниц.

1. *Прерывание по особому случаю при связывании*. Данный тип прерывания возникает при отложенном связывании. Для данного сегмента заполняется строка таблицы сегментов и устанавливается бит особого случая в сегменте, заполняется элемент в таблице активных ссылок.

2. *Прерывание по особому случаю в сегменте*. Просмотр таблицы сегментов может показать, что сегмент, к которому пытается обратиться процесс, отсутствует в основной памяти. Менеджер памяти найдет нужный «виртуальный» сегмент, сформирует для него таблицу страниц (во всех строках таблицы страниц устанавливается бит особого случая в странице) и карту файла, занесет адреса этих таблиц в элемент таблицы активных ссылок, адрес таблицы страниц заносится так же в соответствующий элемент таблицы сегментов.

3. *Прерывание по особому случаю в странице*. Когда сегмент находится в памяти, обращение к таблице страниц может показать, что нужная страница отсутствует в памяти. Менеджер памяти возьмет управление на себя, найдет нужную страницу во внешней памяти и загрузит ее в основную память. При этом может потребоваться замещение какой-либо находящейся в памяти страницы. Таблица страниц корректируется.

4. Как и при чисто сегментном распределении, адрес виртуальной памяти может выйти за границу сегмента. В этом случае произойдет *прерывание по выходу за границу сегмента.*

5. Если контроль по признакам доступа к сегменту показывает, что операция, запрашиваемая по указанному виртуальному адресу не разрешена, произойдет *прерывание по защите сегмента.*

В операционной системе должна быть предусмотрена обработка всех этих прерываний (прерывания 4 и 5 на рисунке не изображены).

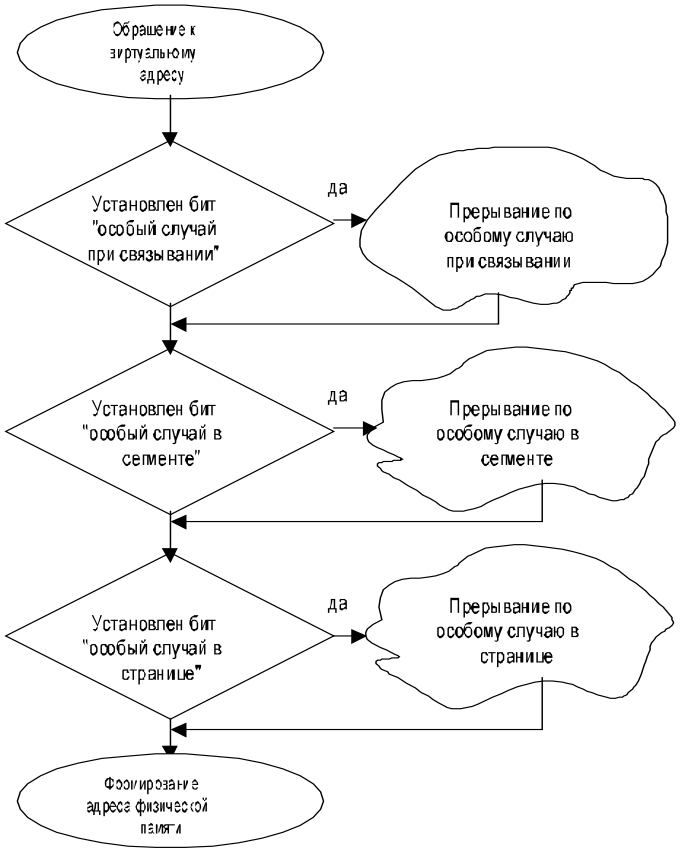


Рис.6.29

Достоинства и недостатки:

Плюсы страничного распределения памяти по запросам:

1) Такое распределение легко реализовать

2) Алгоритм LRU в этом достаточно эффективен

Минусы страничного распределения памяти по запросам:

1) Коллективное использование страниц – под вопросом. На уровне страниц теряется их принадлежность.(Связано с преобразованием адр.) Существуют страницы, которые надо использовать одновременно нескольким процессам, которые в свою очередь могут иметь разные права доступа. Где хранить информацию о том, какие процессы обращаются к странице?

Плюсы сегментного распределения памяти по запросам:

1) Легко реализовать коллективное использование, так как сегмент является логической единицей деления памяти

Минусы сегментного распределения памяти по запросам:

1) Необходимость корректировки таблицы дескрипторов всех процессов при изменении размеров сегментов

2) Сложности при загрузке новых сегментов (в памяти должно сущ. адр пространство необходимого размера). При необходимости система может перенести сегменты путём изменения базового адреса сегмента в дескрипторе (можно возложить на систему, но приводит к большим затратам).

3) Фрагментация (Интенсивная загрузка, и выгрузка может привести к маленьким участкам, в которые загрузить ничего не удастся), хотя система может устранить её путём переноса сегментов (см. выше).

2. Unix: концепция процессов – процессы «сироты», процессы «зомби», демоны; примеры.

Процесс многократно переходит из одного состояния в другое. Из режима пользователя в режим ядра и наоборот. Концепция: процесс выполняется в одной из двух стадий:

1. пользователь/задача;
2. система.

В стадии «задача» процесс выполняет собственный код, в стадии «система» процесс выполняет реентерабельный код ядра (не модифицирует сам себя).

В теле процедуры можно изменить данные. Для реентерабельной процедуры необходимо вынести данные из тела процедуры.

Несколько процессов могут находится одновременно в разных точках одной и той же процедуры. Ядро Unix полностью реентерабельно! Unix система разделения времени.

«сирота» — возникает в том случае, если процесс предок завершился раньше своих потомков. При завершении процесса система проверяет не осталось ли у этого процесса незавершенных потомков. Если остались – система принимает действия по их усыновлению. Процесс потомок усыновляется терминальным процессом. Система переписывает индефикатор такого процесса на 1.

«зомби» — если процесс потомок завершился до того как предок вызвал wait (возможно при аварийном завершении exec), то для того чтобы предок не завис в ожидании несуществующего процесса, система отбирает у него все ресурсы и помещает строку в таблице процессов, помечая такой процесс как зомби. Сделано это для того, чтобы процесс получил статус завершения всех своих потомков.

Чтобы родитель не завис система отбирает у потомка все ресурсы, но оставляет строку в таблице процессов.

«демон» — процесс, который не имеет предков (сервисные функции)

Демон

```
int daemonize(void) {

switch (fork()) {
```

```
case 0:
return setsid();
case -1:
return -1;
default:
exit(0);}}
```

Зомби

```
int zombi(void){
int ChildPid;
if ((ChildPid=fork())==-1){
                perror("Can't fork!!");
                exit(1);
} else{
        if(!ChildPid){
                printf("Child, ChildID=%d, ParentID=%d, GroupID=%d\n",
                        getpid(), getppid(), getgid());
        }else{
                printf("Parent, ChildID=%d, ParentID=%d, GroupID=%d\n",
                        ChildPid, getpid(), getgid());
                wait();
        }
return 0;}
```

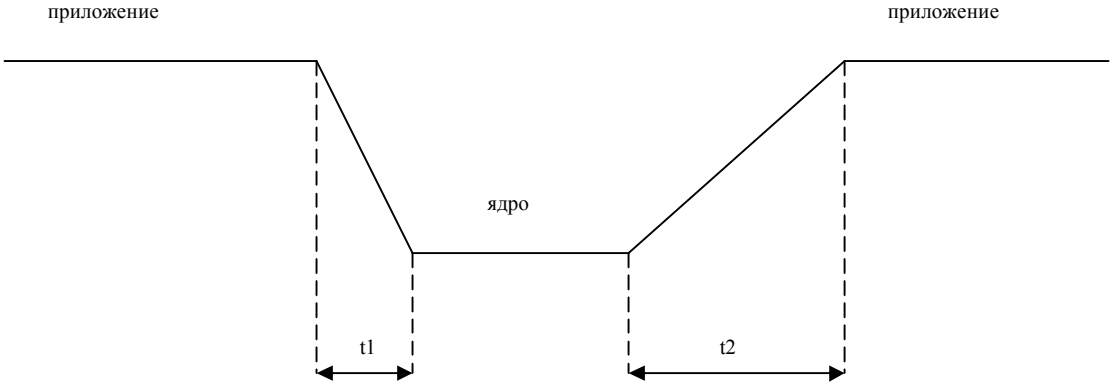
1. **ОС с монолитным ядром. Переключение в режим ядра. Система прерываний. Точные и неточные прерывания.**

В большинстве ОС ядро разбито на несколько уровней, каждый из которых ориентирован на свои функции управления системными аппаратными средствами. Ядро является самым нижним уровнем, оно предназначено для выделения аппаратных ресурсов процессам ОС и программам, выполняющимся под её управлением.
В настоящее время выделяют 2 типа ядер ОС: монолитное ядро и микро ядро.
Монолитное ядро – программа, состоящая из подпрограмм (имеющая модульную структуру), содержащих в себе все функции ОС, включая планировщик, файловую систему, драйверы, менеджеры памяти. Поскольку это одна программа, то она имеет одно адресное пространство, следовательно, все её подпрограммы имеют доступ ко всем её внутренним структурам. Такие ОС делятся на 2 части – резидентную и нерезидентную. Любые изменения приводят к необходимости перекомпилирования всей программы. Пример: ОС UNIX, хотя она имеет минимизированное ядро.
Состояние с минимизированным ядром. [Программа, имеющая модульную структуру. приводит к необходимости перекомпилировать ядро.]
Часть функций вынесены за пределы ядра. В Unix вынесены в shell. Следующие функции остаются в ядре:

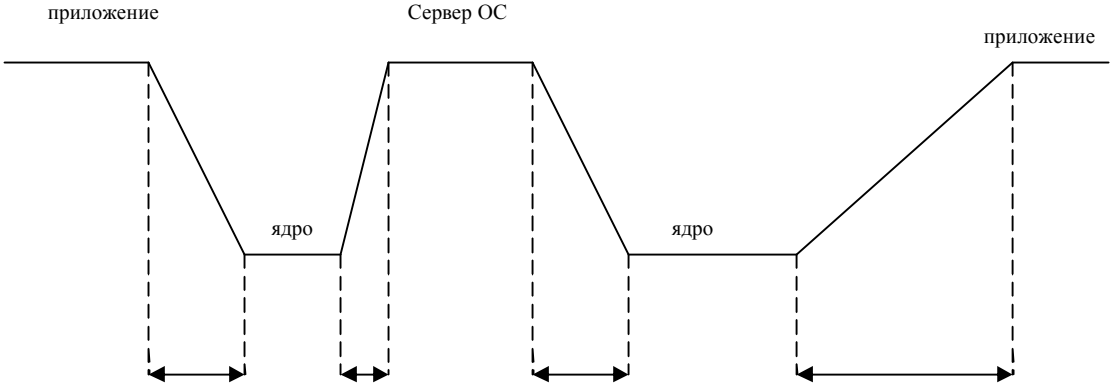
- ✓ Управление процессами нижнего уровня, (диспетчеризация),
- ✓ управление памятью, упр. физ. памятью.
- ✓ Драйверы устройств, кот непосредственно взаимодействуют с клавиатурой.

Производительность

1) Выполнение вызова в ОС с монолитным ядром требует 2 переключений в режим ядра (приложение – ядро – приложение)



2) Без учёта времени передачи самих сообщений, в ОС с микроядром системный вызов требует как минимум 4 переключений плюс время, проводимое в блокировке при передаче сообщений (приложение – микроядро – сервер ОС – микроядро – приложение).



Пр: ОС Mach.

Предлагают надстройки над примитивными сервисами м./я. Системы построены на м./я. не эффективны. (с точки зрения времени, потраченного на выполнение какой-либо функции).

3) Отсутствует четкая граница между функциями самой ОС и функциями прикладной проги. В ОС Hurd пользователь может организовать собственную. ф.с. Но платим мы за это эффективностью выполнения ОС. (Большие затраты).

Переключение процесса в режим ядра

Существуют 3 типа событий, которые могут перевести ОС в режим ядра:

1) системные вызовы (программные прерывания) – software interrupt – traps
2) аппаратные прерывания (прерывания, поступившие от устройств) - interrupts (от таймера, от устройств I/O, прерывания от схем контроля: уровень напряжения в сети, контроль четности памяти)
3) исключительные ситуации- exception

Процесс может находиться или в состоянии задачи или в состоянии ядра.
Вызовы системных сервисов – системные вызовы.
Аппаратные прерывания поступают от контроллера прерываний.
Прерывание таймера – аппаратное прерывание.
Вызов диспетчера, запускающий новый процесс, возложено на таймер.
Если в системе имеется переключение процессов, то процессорное время квантуется, если процесс не успел выполниться, то возвращается в очередь процессов.

Аппаратные прерывания:

- Прерывания от действия операторов (Ctrl+Alt+Del),а так же от схем компьютера правильности работы + контроль уровня напряжения в сети.

Аппаратные прерывания: сигналы от внешних устройств поступают на контроллер прерывания, причем эти прерывания не зависят от выполняемого процесса, т.е процесс вполне может переключиться на выполнение какого-либо другого процесса. Аппаратные прерывания обрабатываются в системном контексте, при этом доступ в адресное пространство процесса им не нужен, т.е. им не нужен доступ к контексту процесса. Обработчик прерывания не обращается к контексту процесса. Очевидно, что прерывания interrupts не должны производить блокировку процесса.

Исключения

Исключения являются синхронным событием, возникают в процессе выполнения программы, возникаю в таких случаях, как:

- Арифметическое переполнение
- деление на нуль
- попытка выполнить некорректную команду,
- Ссылка на запрещенную область памяти
- При образовании адреса, при обращении к физическому адресу, которого нет

Исключения бывают:

1. *Исправимые* – например, обращение к некорректному адресу, но он прошел проверку адреса, обращение к отсутствующему сегменту. Процесс может продолжаться с той же команды, в которой произошло исключение.
2. *Неисправимые* - заканчиваются завершение программы

Исключения бывают 3 видов:

1. Нарушения – fault – это исключение, фиксируемое до выполнения команды или в процессе её выполнения.
2. Ловушка – Trap – процессором обрабатывается после команды, вызвавшей это исключение.
3. Авария – abort – данный тип исключения является следствием невосстановимых, неисправимых ошибок, нпример, деление на ноль.

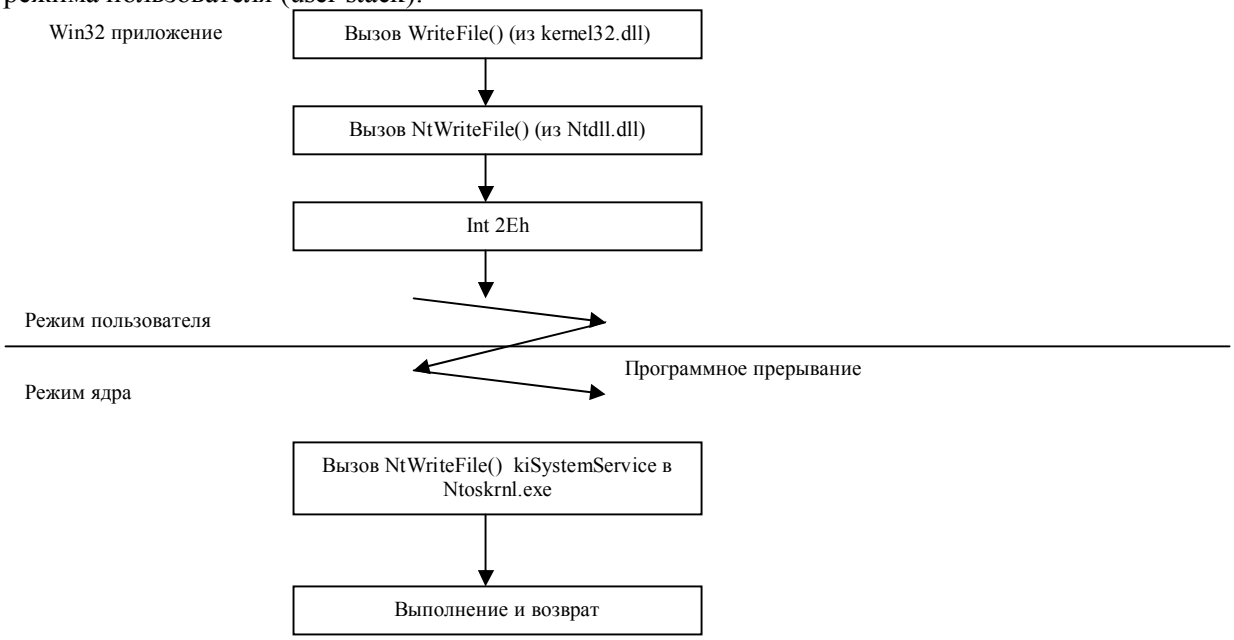
Системные вызовы

Также являются исключениями, но с точки зрения реализации - это системные ловушки.

Набор можно рассматривать как программный интерфейс, предоставляемый ядром системы пользовательским процессам. Эти функции называются API функциями.
Supervisor call. – исполняемое ядро ОС. При системных вызовах сначала вызывается библиотечная функция, которая передает номер системного вызова в стек пользователя и вызывает специальную инструкцию системного прерывания (int)[Int 2eh – системные вызовы в Windows. Motorola 680.0 - trap], которое меняет режим выполнение на режим ядра и передает управление обработчику системного вызова. Системные вызовы выполняются в режиме ядра, но в контексте пользователя (задачи, процесса). Следовательно, они имеют доступ к адресному пространству и управляющим структурам, вызвавшего их процесса. С другой стороны, они могут обращаться к стеку ядра этого процесса. [Unix – syscall. Win2000 – KiSystem Service - диспетчер системных сервисов]. Обработчик системных вызовов загружает в stack pointer (sp) адрес стека режима ядра (kernel stack) процесса и сохраняет в этом стеке, аппаратный контекст процесса. В аппаратный контекст включается:

- instruction pointer (ip)
- stack pointer (sp)
- слово состояния процессора (processor status word) (PSW)
- регистры управления памятью
- регистры сопроцессора (floating point unit).

Затем, по номеру системного вызова, происходит обращение к соответствующей таблице [В win2000 к таблице диспетчеризации системных вызовов. (System Service dispatch table)], содержащей указатель на обработчик системных прерываний. После завершения, супервизор восстанавливает возвращенные из обработчика системных вызовов значения или код ошибки в соответствующие регистры. Затем восстанавливает аппаратный контекст процесса, вызывает библиотечную функцию, которая восстанавливает режим задачи или пользовательский режим, а указатель стека (sp) переключается на стек режима пользователя (user stack).



Эти оба стека принадлежат процессу. (user stack, kernel stack)

В момент прерывания счетчик команд может не отражать истинные границы между выполненными и невыполненными командами. Скорее всего, он будет указывать на адрес команды, которую следует считать из памяти, а не той которой завершена. Следовательно, при возврате из прерывания ОС не может просто начать заполнять конвейер с адреса содержащегося в счетчике команд. Она должна определить последнюю выполненную команду и это требует серьезного анализа состояния процессора. В прерываниях суперскалярного процессора – еще хуже. Например, может возникнуть следующая ситуация команды 1,3,5,8 – выполнены; 4,6,7,9,10. Счетчик может указывать на команды 9,10,11. В связи со всем этим вводиться понятия *точного* прерывания и *неточного*.

Точное прерывание – это прерывание, оставляющее машину в строго определенном состоянии. Имеет свойства:

1. счетчик команд указывает, на команду, до которой все команды полностью выполнены.
2. не одна команда после той, на которую указывает счетчик команд – не выполнена.
3. состояние команды, на которую указывает счетчик команд – известно, причем в

перечисленных условиях не говориться, что команды после той, на который указывает счетчик команд не могут выполняться, а утверждается , что все изменения связанные с выполнением этих команд должны быть отменены до выполнения обработки прерывания.

При аппаратных прерывания счетчик команд обычно указывает на следующую команду.
При исключениях – указывает на ту команду, которая вызвала прерывание.

Неточное прерывание – прерывание, не удовлетворяющее перечисленным требованиям.
Машины с неточными прерываниями обычно выгружают в стек огромное количество данных, чтобы дать ОС возможность определить, что происходило в момент прерывания. Сохранение больших объемов данных при каждом прерывании значительно замедляет вход процедуры обработки прерывания. Восстановление после прерывания является сложно и от сюда медленной.

Сверхбыстрый суперскалярный процессор не пригоден для задач реального времени из-за страшно медленных прерываний.
Альтернативное решение: одни прерывания выполнять как точные, другие как неточные. Прерывания вводы вывода – точный, а исключения могут быть неточными, т.к. процесс будет завершен аварийно.
Пример: Pentium Pro – суперскалярный процессор, поддерживающий точные прерывания.
Цена точных прерываний сложной внутрепроцессорной логикой прерывания. Скрыто от пользователя – выполняется аппаратно. Ценой является не скорость обработки а сложность процессора.

2.Задача: читатели-писатели, решение с использованием семафоров Дейкстра для ОС Unix.
Существует набор процедур, обращающихся базе данных, чтобы получить оттуда информацию, и существуют процедуры, которые имеют право изменять содержимое базы данных. Критическим ресурсом является конкретное поле записи в базе данных. Поскольку процесс-писатель изменяет данные, он должен иметь монопольный доступ к БД. Очевидно, что монопольный доступ надо устанавливать на уровне конкретного поля структуры, а не всей БД. В каждый момент времени может работать только один процесс-писатель. Не имеет смысла ограничивать количество процессов-читателей, так как они не изменяют содержимое БД.

```
// мой код
void init_semaphors(int semid){
   union semun arg;
   arg.val=0;
   semctl(semid,0,SETVAL,arg);
   semctl(semid,1,SETVAL,arg);
   semctl(semid,2,SETVAL,arg);
}

void start_read(int semid){
        struct sembuf incReaders[1]={{READERS_COUNT,1,0}};
        struct sembuf lock[2]={
                        {WR_NLOCK,0,0},//WAIT 0 - notlock
                        {WRITERS_COUNT,0,0}//NO WRITERS
        };
        semop(semid,lock,2);
        semop(semid,incReaders,1);
}

void stop_read(int semid){
        struct sembuf unlock[1]={
```

```
                        {READERS_COUNT,-1,0}
        };
        semop(semid,unlock,1);
}

void start_write(int semid){
        struct sembuf incWriters[1]={{WRITERS_COUNT,1,0}};//WRITERS++
        struct sembuf lock[3]={
                        {WR_NLOCK,0,0},//WAIT 0 nlock
                        {READERS_COUNT,0,0}, //0 - Readers
                        {WR_NLOCK,1,0}//NLOCK++
        };
        semop(semid,incWriters,1);
        semop(semid,lock,3);

}

void stop_write(int semid){
        struct sembuf unlock[2]={
                        {WR_NLOCK,-1,0},
                        {WRITERS_COUNT,-1,0}
        };
        semop(semid,unlock,2);
}

void reader(int semid, int id,int sleep_sec){
  while(1){
        sleep(sleep_sec);
    start_read(semid);
    printf("READER%d:%d\n",id,*res);
    stop_read(semid);
    }
}

void writer(int semid,int id,int sleep_sec){
  while(1){
        srand ( time(NULL) );
    sleep(sleep_sec);
    start_write(semid);

    int send =  (rand()*rand()%100+30); //30-130
    *res=send;
    printf("Writer%d:%d\n",id,send);
    stop_write(semid);
  }
}
```

1. Процессы: организация монопольного доступа – реализация взаимоисключения в помощью команды test-and-set, алгоритм Деккера.

Монопольный доступ осуществляется взаимоисключением, т.е. процесс, получивший доступ к разделяемой переменной, исключает доступ к ней др. процессов.

Аппаратная реализация взаимоисключения (test-and-set).

Впервые test-and-set была введена в OS360 для IBM 370. Эта команда является машинной и неделимой, т.е. ее нельзя прервать. Она одновременно производит проверку и установку ячейки памяти, называемой ячейкой блокировки: читает значение логической переменной B, копирует его в A, а затем устанавливает для B значение «истина» (все это делается за счет одной шины данных). Она присутствует в наборе сист вызовов Win и Unix

Test-and-set(a, b) : a = b; b = true;

В Windows это называется спин-блокировкой. /* спин-блокировкой по-рязаной, наз. проверка флага в цикле */

```
flag, c1, c2: logical;
P1:    while(1)
  c1 = 1;
  while(c1 == 1)
        test_and_set(c1, flag);
  CR1;
  flag = 0;
  PR1;
end P1;

P2:    while(1)
  c2 = 1;
  while(c2 == 1)
        test_and_set(c2, flag);
  CR2;
  flag = 0;
  PR2;
end P2;

flag = 0;
parbegin
P1;
P2;
parend;
```

Пусть P1 хочет войти в свой критический участок, когда P2 уже там. P1 уст в единицу и входит в цикл проверки. Поскольку P2 нах в критическом участке, то у P2 – 1. P1 будет находится в цикле активного ожидания, пока P2 не выйдет из своего критичнского участка.
Т.к. test-and-set машинная неделимая команда и выполняется очень быстро, то бесконечного откладывания не возникает.

Бесконечное откладывание – ситуация, когда разделённый ресурс снова захватывается тем же процессом.

Программная реализация (алгоритм Деккера).

Деккер – голландский математик. Предложил способ свободный от бесконечного откладывания.

```
flag1, flag2: logical;
queue: int;

p1:    while(1)
  flag1 = 1;
  while(flag2)
        if(queue == 2) then
        begin
                flag1 = 0;
                while(queue == 2);
                flag1 = 1;
        end;
  CR1;
  flag1 = 0;
  queue = 2;
  PR1;
end P1;

p2:    while(1)
        flag2 = true;
  while(flag1)
        if(queue == 1) then
        begin
                flag2 = 0;
                while(queue == 1);
                flag2 = 1;
        end;
  CR2;
  flag2 = 0;
  queue = 1;
  PR2;
end P2;

flag1 = 0;
flag2 = 0;
parbegin
P1;
P2;
parend;
```

queue – очередь процесса входить в критическую секцию.

Недостаток обоих методов – активное ожидание на процессоре.

Активное ожидание – ситуация, когда процесс занимает процессорное время, проверяя значение флага. Активное ожидание на процессоре является неэффективным использованием процессорного времени.

2. Unix: разделяемая память(shmget(), shmat()) и семафоры (struct sem, semget(), semop()).

Разделяемые сегменты – средство взаимодействия процессов через разделяемое адресное пространство. Т.к. адресное пространство защищено, процессы могут взаимодействовать только через ядро.

int shmget(key_t *key*, size_t *size*, int *shmflg*);

Возвращает идентификатор общего сегмента памяти, связанного с ключом, значение которого задано аргументом key. Если сегмента, связанного с таким ключом, нет и в параметре shmflg имеется значение IPC_CREATE или значение ключа задано IPC_PRIVATE, создается новый сегмент. Значение ключа IPC_PRIVATE гарантирует уникальность идентификации нового сегмента.

Значение параметра semflg формируется как логическое ИЛИ одного из значений: IPC_CREATE (создать новый сегмент) или IPC_EXCL (получить идентификатор существующего) и 9 бит прав доступа (S_IRWXU, S_IRWXG, S_IRWXG,…)

void *shmat(int *shmid*, const void **shmaddr*, int *shmflg*);

Присоединяет разделяемый сегмент памяти, определяемый идентификатором shmid к адресному пространству процесса. Если значение аргумента shmaddr равно нулю, то сегмент присоединяется по виртуальному адресу, выбираемому системой. Если значение аргумента shmaddr ненулевое, то оно задает виртуальный адрес, по которому сегмент присоединяется.

Если в параметре shmflg указано SHM_RDONLY, то присоединенный сегмент будет доступен только для чтения.

Семафоры.(устраняют активное ожидание на процессоре)

Семафор – неотрицательная защищённая переменная, над которой определено 2 операции: P (от датск. passeren - пропустить) и V(от датск. vrygeven - освбодить).

```
struct sem {

  short   sempid;              // ID процесса, проделавшего последнюю операцию
  ushort  semval;              // Текущее значение семафора
  ushort  semncnt;        // Число процессов, ожидающих освобождения требуемых ресурсов
  ushort  semzcnt;        // Число процессов, ожидающих освобождения всех ресурсов
};
```

int semget(key_t *key*, int *nsems*, int *semflg*);

Возвращает идентификатор массива из nsem семафоров, связанного с ключом, значение которого задано аргументом key. Если массива семафоров, связанного с таким ключом, нет и в параметре semflg имеется значение IPC_CREATE или значение ключа задано IPC_PRIVATE, создается новый массив семафоров. Значение ключа IPC_PRIVATE гарантирует уникальность идентификации нового массива семафоров.

semflg формируется аналогично shmflg, т.е. IPC_CREATE | S_IRWXU | S_IRWXG…

int semop(int *semid*, struct sembuf **sops*, unsigned *nsops*);

Выполняет операции над выбранными элементами массива семафоров, задаваемого идентификатором semid. Каждый из nsops элементов массива, на который указывает sops, задает одну операцию над одним семафором и содержит поля:

```
short sem_num; /* Номер семафора */
short sem_op;  /* Операция над семафором */
short sem_flg; /* Флаги операции */
```

Значение поля sem_op возможны следующие:

1. Если значение sem_op отрицательно, то:

- Если значение семафора больше или равно абсолютной величине sem_op, то абсолютная величина sem_op вычитается из значения семафора.
- В противном случае процесс переводится в ожидание до тех пор, пока значение семафора не станет больше или равно абсолютной величине sem_op.

2. Если значение sem_op положительно, то оно добавляется к значению семафора.

3. Если значение sem_op равно нулю, то:

- Если значение семафора равно нулю, то управление сразу же возвращается вызывающему процессу.
- Если значение семафора не равно нулю, то выполнение вызывающего процесса приостанавливается до установки значения семафора в 0.

Флаг операции может принимать значения IPC_NOWAIT или/и SEM_UNDO. Первый из флагов определяет, что semop не переводит процесс в ожидание, когда этого требует выполнение семафорной операции, а заканчивается с признаком ошибки. Второй определяет, что операция должна откатываться при завершении процесса.

1.Управление процессорами: планирование и диспетчеризация, алгоритмы планирования – классификация; приоритетное планирование, планирование в современных системах.

Планирование – управление распределением ресурсов центральным процессором между конкурирующими процессами путем передачи им управления, согласно некоторой стратегии планирования.

Планировщик – программа, которая отвечает за управление, использование совместного ресурса с учетом 2х требований:

1) Условие правильности – при получении доступа к ресурсу процесс не повреждает сам себя и систему.

2) Планирование осуществляется по некоторому алгоритму.

Диспетчеризация - выделение процессу кванта времени процессора.

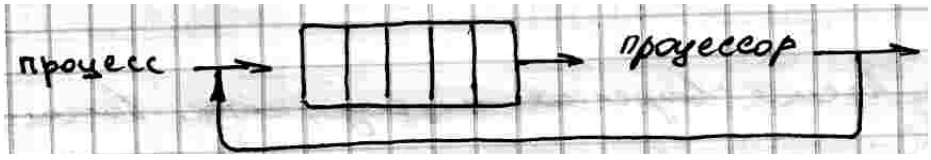
Классификация алгоритмов планирования:

- С переключением / без переключения (процессор работает от начала и до конца при получении процессорного времени т.е. процесс выполняется произвольное кол-во времени в зависимости от самого процесса, что не гарантирует время отклика)
- С приоритетами / без приоритетов {приоритеты бывают абсолютные и относительные (вычисляются относительно базового параметра), статические и динамические}
- С вытеснением / без вытеснения (процесс может вытеснить другой процесс (если есть система приоритетов))

Алгоритмы планирования:

1) FIFO (First In First Out) Очередь. Без приоритетов, без переключения. Программа выполняется от начала и до конца.

2) RR (Round Robin) . Циклический алгоритм. Без приоритетов, с переключением. Процесс, у которого истек квант, оправляется в конец очереди.

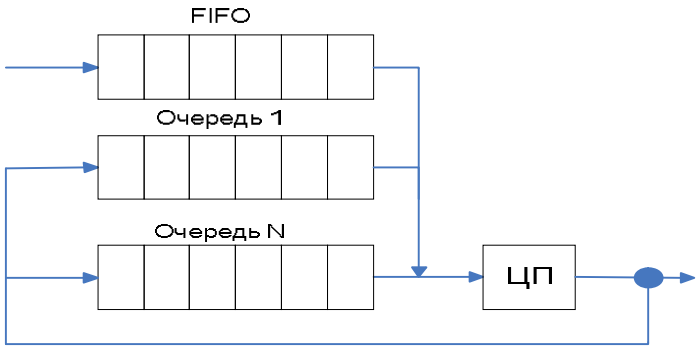


3) SJF (Shortest Job First). Первыми выполняются короткие задания. Со статическими приоритетами, без переключений. Чреват бесконечным откладыванием

4) SRT (Shortest Remaining Time) С переключением, с вытеснением. Учитывает время нахождения процесса в очереди. Выполняющийся процесс может быть прерван, если будет найден процесс с меньшим оценочным временем. Необходимо отслеживать текущее время обслуживания.

5) HRN (Highest Response Next – наибольшее относительное время ответа) С динамическими приоритетами. $priority = \frac{t_W - t_S}{t_S}, t_W$ – время ожидания в очереди, t_S - запрошенное время обслуживания, т.е. чем больше ожидает, тем больше приоритет.

Адаптивное планирование (многоуровневые очереди)



В первую очередь, которая имеет наивысший приоритет , попадают новые процессы и вернувшиеся из блокировки. Если процесс, находясь в 1-ой очереди не успел завершится или выдать запрос, то он по истечении кванта попадет в очередь с более низким приоритетом. В последней очереди(тип RR) крутится холостой(idle) процесс и неинтерактивные процессы. Также может учитываться обьем памяти, необходимы процессу – адаптивно-рефлексивное планирование, т.е. выделяется очередной квант только при наличии свободной памяти в системе.

В современных системах априорная информация о времени выполнения процессов отсутствует.

2. Защищенный режим работы персонального компьютера с процессорами Intel (486, ..). Уровни привилегий. Системные таблицы: GDT, LDT, IDT.

Защищенный режим - 32 разрядный с поддержкой многопроцессности, виртуальной памятью, адресное пространство до 4 Гб. 2 типа организации памяти: сегментная, сегментно-страничная память по запросам. 4 уровня привилегий защиты. (кольца защиты). 32-разрадный, многпоточный, многопроцессный, 4 уровня привелегий, доступно 4 Гб виртуальной памяти(для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. 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.)

Уровни привилегий:

В защищенном режиме предусмотрен механизм защиты с помощью системы уровней привтлегий. Существует 4 уровня привелегий: от 0 до 3. Нулувой уровень – предоставляет максимальные привилегии, используется для ядра ОС. Третий уровень - минимальные привилегии, используется для прикладных программ.

Каждому сегменту сегменту программы придется определенный уровень привилегий, указываемый в поле DPL(Descriptor Privilage Level, уровень привилегий дескриптора) его дескриптора. Уровень привилегий, указанный в дескрипторе, назначается всем обьектам, входящим в данный сегмент. Уровень привилегий выполняемого в данный момент сегмента команд называется текущим уровнем привилегий – CPL(Current Privilage Level). Он определяется полем RPL селектора сегмента команд, загружаемого в CS. Вся система привилегий основана на сравнении CPL выполняемой программы с уровнями привилегий DPL сегментов, к которым она обращается. В реальном режиме уровней привилегий нет.

Системные таблицы:

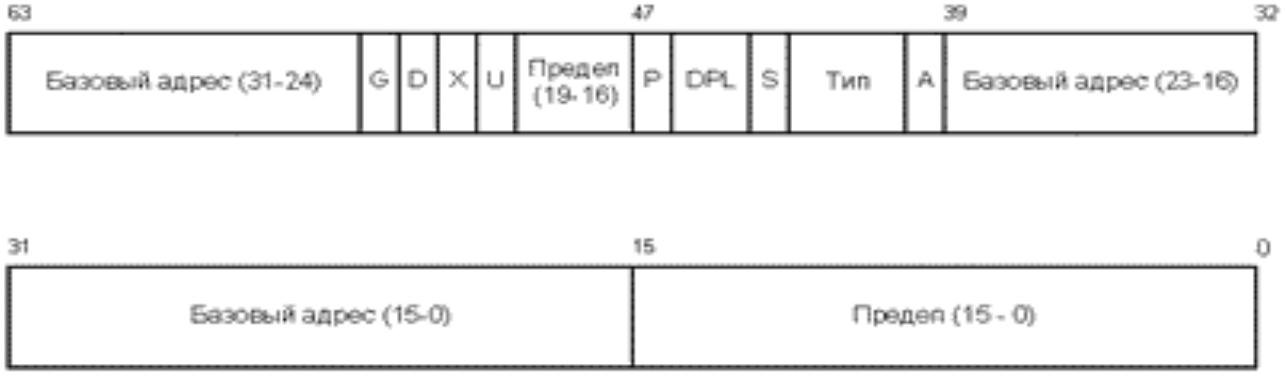
Защищенный режим – 32х-разрядный режим, поддерживает многопоточность и многопроцессность. В отличие от реального режима здесь доступно 4 Гб памяти (в реальном диапазон адресов памяти ограничен 1 мб). В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне.

GDT (global descriptor table) – таблица, которая описывает сегменты системы, общие сегменты (сегменты ОП)
На начальный адрес GDT указывает GDT Register (32 разрядный). В системе только одна GDT.
LDT (local descriptor table) – таблица, которая описывает адресное пространство процесса. В LDT Register находится смещение до соответствующего дескриптора в GDT, описывающего сегмент, в котором находится LDT . Таблиц LDT столько, сколько процессов.
IDT (interrupt descriptor table) – таблица, предназначенная для хранения адресов обработчиков прерываний. Базовый адрес IDT помещен IDT Register.

Формат селектора (является идентификатором сегмента):

Индекс кратен 8 и является смещением в таблице дескрипторов
0 и 1 биты – Requested Privilege Level, показывает на каком уровне привилегии работаем (00-нулевой уровень)
2 бит – Table indicator, 0 – адрес в GDT, 1 – в LDT
Селектор указывает на дескриптор сегмента в таблице дескриптора.

Формат дескриптора сегмента:



A – access – бит доступа к сегменту

Тип – 3 бита: r/w, c/cd, i. **r/w** – для символ. кода 1-чтение разрешено, 0-нет; для символ. данных 1-запись разрешена. **c/cd** – 0-для сегмента данных, 1- для сегмента стека. **i** – бит предназначения 0-сегм д. или стека, 1-кода

000b - сегмент данных, разрешено только считывание.
001b - сегмент данных, разрешено считывание и запись.
010b - сегмент стека, разрешено только считывание (не используется в практике.)
011b - сегмент стека, разрешено чтение и запись.
100b - сегмент кода, разрешено только выполнение.
101b - сегмент кода, разрешено выполнение и считывание.
110b - подчиненный сегмент кода, разрешено только выполнение.
111b - подчиненный сегмент кода, разрешено выполнение и считывание.

S – определяет, что описывает дескриптор

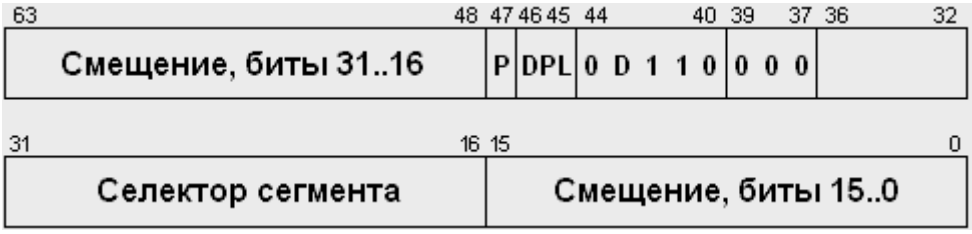
DPL – уровень привилегий

P – бит присутствия, используется для работы с ВП. 0 – сегмента нет в ВП, 1 – есть

D – бит разрядности операндов и адресов. 0 - 16-разрядные , 1 – 32.

G – 0 – размер сегмента задан в байтах, 1 – в страницах

Формат дескриптора прерывания:



К дескрипторам GDT и LDT мы обращаемся с помощью селекторов, к дескриптору IDT мы обращаемся по смещению, которое берем из прерывания.

Обработчик прерывания:

IDTR (указывает на начало IDT) + смещение из прерывания = дескриптор в IDT

Из дескриптора в IDT берем селектор

С помощью селектора узнаем с какой таблицей мы работаем.

1. Если работаем с GDT, то с помощью селектора получаем дескриптор сегмента, в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT мы получаем точку входа в обработчик прерывания.
2. Если работаем с LDT, то с помощью LDTR (в котором у нас смещение до дескриптора сегмента в GDT, в котором находится LDT) находим этот дескриптор, получаем сегмент. В этом сегменте находится нужная LDT, в ней с помощью селектора получаем дескриптор сегмента в котором находится наш обработчик, в этом сегменте с помощью смещения из дескриптора в IDT получаем точку входа в обработчик прерывания.

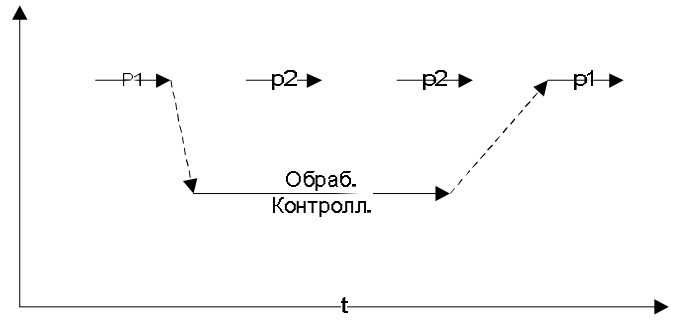
1.Подсистема ввода - вывода: синхронный и асинхронный ввод-вывод.

Система вв/вывода – часть вычислительной системы, ориентированная прежде всего на обмен сообщениями с центральным процессором.

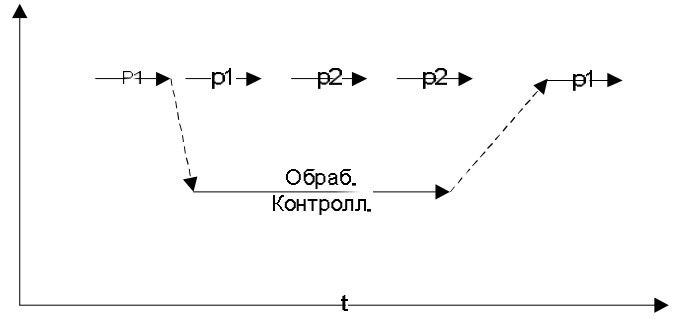
В системе ввода-вывода все операции называются read/write. Ни одна программа не может обратиться к вв/выв напрямую, это сделано чтобы защитить ос. Для этого существует система ввода/вывода (у нее нет спец.
названия, во всех ОС по разному), позволяющая однообразно обращаться к внешним устройствам. Подсистемы ввода/вывода объединяют аппаратные и программные компоненты, которые затем выполняются процессором, подключенным к системе.

Разработка подсистемы ввода/вывода – наиболее трудоемкая задача при разработке ОС.
Подсистема ввода-вывода решает следующие задачи:
1) Задача обеспечения подключения к ВС различных по типу и характеристикам устройств. Решается путем унификации способов обмена информации и способов подключения к системе.
2) Задача управления работой устройств вв/выв. Решается включением к цепь управления специальных программ, называемых драйверы. Драйвер – это прога, входящая в состав ОС, предназначенная для обслуживания конкретного периферийного устройства. Драйвер учитывает специфику работы и обслуживания
3) Задача обеспечения доступа к устройствам ВВ всему множеству || выполняемых задач, и при этом эффективное разделение устройств между этими || задачами
4) Предоставление пользователю удобного интерфейса, обеспечивающего возможность использования команд ВВ, стандартных потоков. Для этого скрываются детали аппаратного обеспечения.

Синхронный ввод-вывод – ситуация когда приложение блокируется в ожидании вв/вывода. Завершение этой задачи реализуется системой ВВ.

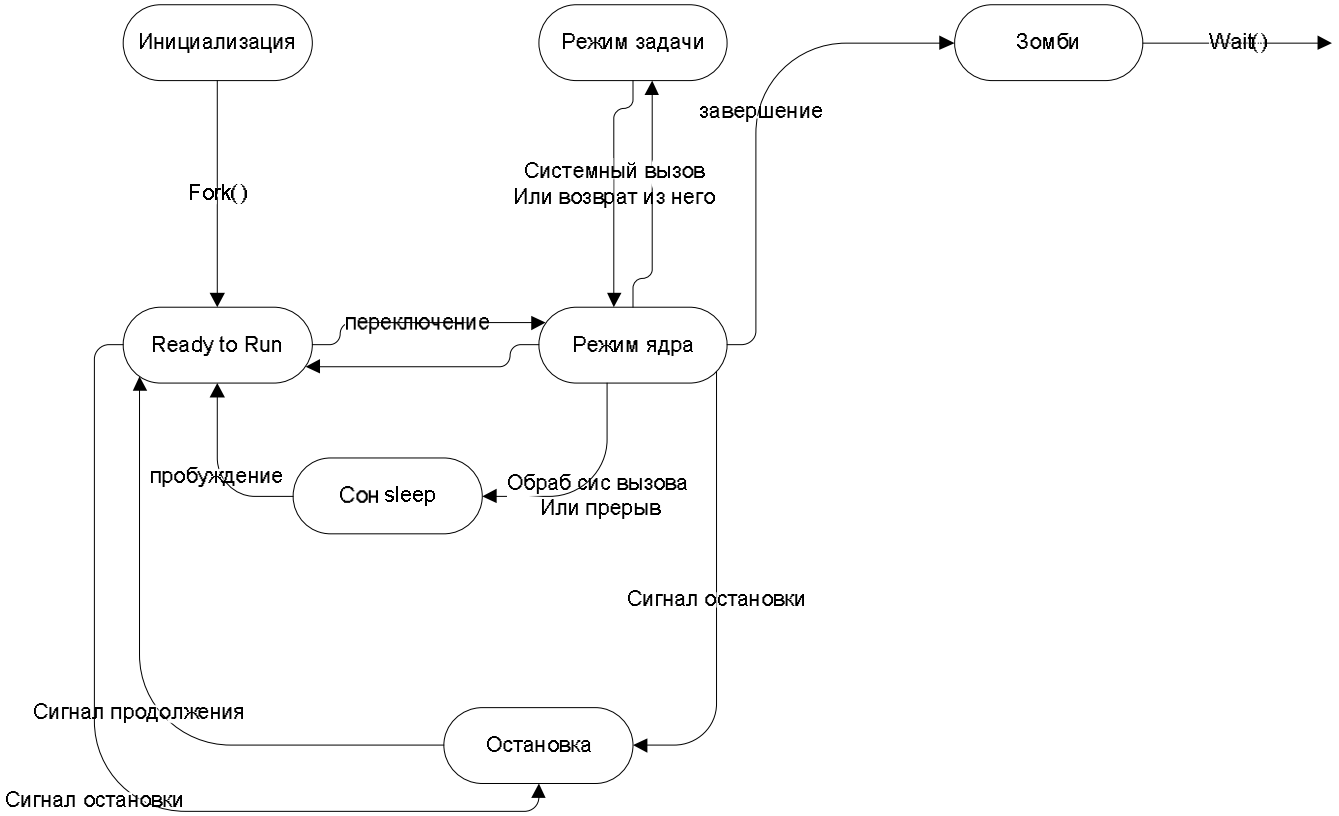


Асинхронный ввод-вывод – приложение, выдавшее запрос на ВВ, не сразу блокируется, а еще некоторое время продолжается.



2.Процессы в ОС Unix: средства взаимодействия процессов, сравнение – достоинства и недостатки.

Процесс - программа в стадии выполнения. Делиться на потоки. Программист создает в своей программе потоки, которые выполняются квазипараллельно.



Средства взаимодейтсвия процессов в Unix:

1) Сигналы. Важные сообщения в системе сопровождаются сигналами. В Unix процесс может принимать, обрабатывать, порождать сигналы. Команды:
a. Signal(int sig_num, void* catcher) – установка обработчика сигнала.
b. Kill(int pid, int sig) послать сигнал процессу.
2) Семафоры – неотицательная переменная с опредленными над ней двумя неделимыми операциями p(s) и v(s). p(s) – dec(s) и проверка, v(s) – inc(s). Семафоры в Unix создаются системным вызовом semget(key_t key, int count, int flag). Операции над семафором – semop(semid, buffer, op_count).
3) Программные каналы – труба типа FIFO. Программные каналы описываются в соответствующей таблице. При этом канал имеет собственные средства синхронизации. Для этого создается буфер [2] типа integer, 1 канал которого предназначен для чтения, второй для записи.
a. Именнованные – системный вызов MKNOD(). Любой процесс, знающий ID канала, может с ним работать.
b. Неименнованные – порождаются вызовом pipe(buffer). У неименованных нет ID, но есть дескриптор. Неименнованные каналы доступны только родственникам.
В системной области памяти при переполнении буфера, имеющиеся дольше всего, перезаписываются на диск. Если процесс записал больше 4 кб, то труба буферизируется во времени, останавливая процесс *писателя*, пока все данные не будут прочитаны
4) Очереди сообщений. При посылке сообщения сообщение копируется в адресное пространство ядра, при получении – в адресное пространство процесса => избыточное копирование. Системные вызовы:
a. Msgget
b. Msgsnd
c. Msgrsv
5) Сегменты разделяемой памяти – это сегменты, выделенные в адресном про-ве ядра, это адресное пространство подключается к адресному виртуальному пространству процесса. Получается указатель на разделяему память, т.е. mapping. Т.к. нет копирования => сегменты разделяемой памяти очень быстрые. Системные вызовы:
a. Shmget
b. ShmAt
c. ShmDt

1. Средства взаимодействия процессов: мониторы – простой монитор, монитор "кольцевой буфер".

Монитор – языковая конструкция , состоящая структуры данных и набора подпрограмм, которые используют эту структуру. При этом монитор защищает данные. Доступ к монитору имеют только подпрограммы, включенные в монитор.
Процесс, который захватил монитор, называется процессом, находящемся в мониторе, остальные процессы переводятся в очередь и стоят в ожидании на мониторе.
Для каждой отдельно взятой причины, при которой процесс может быть переведен в состояние блокировки назначается условие и своя событийная переменная.
Над переменной типа “условие” разрешено две операции: wait и signal. Оператор wait задерживает выполнение процесса, вызвавшего монитор и открывают доступ к монитору.
Оператор signal выполняется следующим образом, если очередь переменных типа signal не пуста, то из очереди берется процесс и инициализируется его выполнение. Если пуста – signal ничего не делает.
Чтобы контролировать для процесса вход в ресурс, он должен иметь приоритет выше, чем процесс, пытающийся войти в ресурс.
Мониторы используются в ОС для организации доступа к ресурсу, мониторы могут представляться системе как системные вызовы.
Мониторы могут предмтавляться пользователю как языковая программа, чтобы реализовывать взаимоисключение.
Монитор – программное средство, разработанное Хоаром и дающее возможность управления совместным использованием ресурсов для асинхронных процессов, включая возможность управляемого обмена данными между процессами.
Монитор гарантирует, что в каждый момент времени процедуры монитора может использовать только 1 процесс, находящийся в мониторе. Если какой-либо процесс находится в мониторе, то любой другой процесс ставится в очередь (переводится в состояние ожидания) вне монитора.

- ✓ простой монитор

обеспечивает выделение единственного ресурса произвольному числу процессов. Монитор, обслуживающий произвольное количество процессов, ограничен длиной своей очереди.

```
RESOURCE:MONITOR;
var
  busy:boolean;
  x:conditional;

procedure acquire
//если busy – истина -> по переменной x выполняется wait
//если busy – ложь -> монитор без задержки получает доступ к ресурсу
begin
  if (busy) then
    wait(x);
  busy := true;
end;

procedure release
//процедура выполняется если занимающий ресурс обратился к монитору для //освобождения ресурса, busy устанавливается в false
//при помощи функции signal проверяется список процессов, стоящих в
//очереди к переменной x. В зависимости от алгоритма, выбирается один
//из процессов и запускается на выполнение
begin
```

```
  busy := false;
  signal(x);
end;

begin
  busy := false;
end.
```

✓ монитор «кольцевой буфер»

решает задачу производства-потребления, то есть существуют процессы-производители и процессы-потребители, а также буфер – массив заданного размера, куда производители помещают данные, а потребители считывают оттуда данные в том порядке, в котором они помещались (FIFO – “первым вошел, первым вышел”). Монитор – средство, которое предложил Хоар.

Кольцевой буфер – это структура данных, широко применяемая в ОС для буферизации обменов между процессами-производителями и процессами-потребителями.

```
RESOURCE: MONITOR;
var
  bcircle:array[0..n-1] of byte;
  pos:0..n;                    //текущая позиция
  i:0..n-1;                    //заполненные позиции
  j:0..n-1;                    //освобождённые позиции
  buffer_full:conditional;
  buffer_empty:conditional;

procedure producer(data: type)
begin
  if (pos=n) then
    wait(buffer_empty);
  bcircle[i]:=data;
  inc(pos);
  i:=(i+1) mod n;
  signal(buffer_full);
end;

procedure consumer(var data: type)
begin
  if (pos=0) then
    wait(buffer_full);
  data:=bcircle[j];
  dec(pos);
  j:=(j+1) mod n;
  signal(buffer_empty);
end;

begin
  pos:=0;
  i:=0;
  j:=n;
end.
```

2. Защищенный режим: перевод компьютера из реального режима в защищенный и обратно.

Защищенный режим – 32х битный, многопоточный, многопроцессный режим работы работы процессора, с использованием виртуальной памяти, 4гб адресного пространства, 2 способа организации памяти: страницы по запросу, сегменто-страничная память по запросу. 4 уровня привилегий.

Список действий для перехода в защищенный режим.

1) Необходимо проверить установлен ли бит 0 регистра CR0 в единицу. Если это так, то мы уже находимся в защищенном режиме
2) Сформировать таблицы GDT(LDT) и IDT
3) Запретить маскируемы и немаскируемые прерывания
4) Открыть линию A20 (для обеспечения адресного заворачивания)
5) В сегментные регистры записать данные селекторов
6) Занести базовые адреса сегментов в таблицу GDT
7) С помощью привилегированных команд lgdt и lidt базовые адреса соответствующих таблиц в регистры GRDR и IDTR
8) Устанавливаем бит 0 регистра CR0 в единицу.
9) Jmp far protected_entry (прыгаем на участок, который выполняется в защищенном режиме).

Список действие для перехода обратно:

1) Заполнение теневых регистров значениями FFh для включения адресации реального режима
2) Заполняем сегментные регистры селекторами
3) Возвращаем предыдущие значения регистров GDTR и IDTR
4) Разрешить маскируемы и немаскируемые прерывания

1. Виртуальная память: распределение памяти страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество.

Виртуальная память – память размер которой превосходит размер физического адресного пространства. Используется адресное пространство диска как область свопинга или педжинга, т.е. для временного хранения областей памяти.

3 способа организации :
1- Страничное распределение памяти по запросам
2- Сегментное распределение памяти по запросам
3- Сегментно-страничное распределение памяти по запросам

Страница - является единицей физического деления памяти. Её размер устанавливается системой.
Сегмент – является единицей логического деления памяти. Её размер определяется объемом кода.

Распределение памяти страницами по запросам

Адресное пространство процесса и адресное пространство физической памяти делится на блоки равного размера. Блоки, на которые делится адресное пространство процесса называют страницами, а блоки на которые делится физическая память – кадрами, фреймами или блоками.

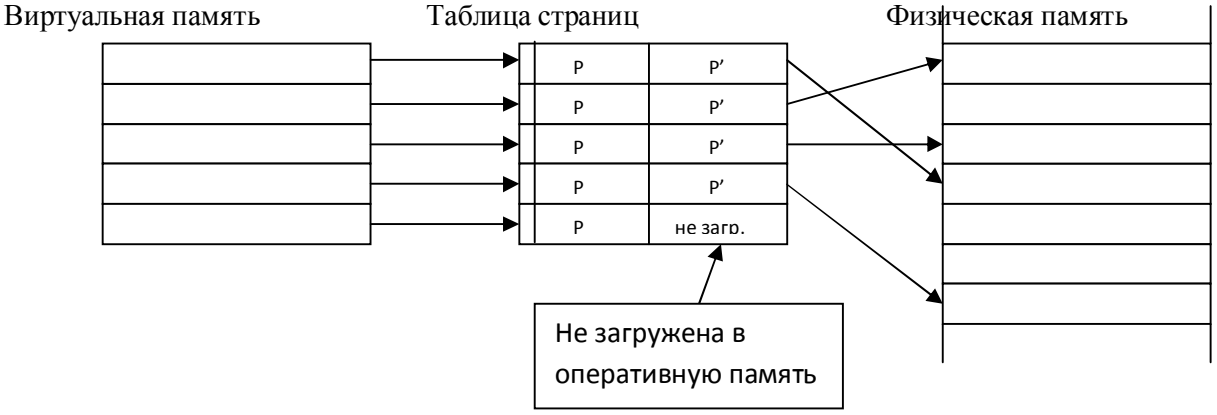
Процесс копируется в страничный файл в области свопинга, таким образом для него создается виртуальное адресное пространство. Соответственно размер виртуального адресного пространства может превышать объем физической памяти.

Для возможности отображения страниц на соответствующие блоки физической памяти необходимо аппаратно поддерживаемое преобразование адресов (иначе слишком долго)

Виртуальный адрес состоит из 2х частей – номер страницы (p) и смещение страницы (d)

P – номер страницы	D - смещение

В реализации виртуальной памяти участвует таблица страниц – она сопоставляет номеру страницы физический кадр оперативной памяти (P’), смещение остается таким-же из-за одинакового размера страниц.



Первый бит в таблице страниц – бит присутствия. Если 1 – указывает на то что страница загружена в оперативную память. Если 0 – не загружена.

При инициализации программы мы создаем таблицу виртуальной памяти и должны загрузить часть в оперативную память (как минимум точку входа). Процесс получив квант времени вскоре потребует

страницу которой нету в оперативной памяти – возникнет прерывание (страничная неудача – исправимое исключение).
Тк возникло прерывание система перейдет в режим ядра и будет работать менеджер памяти который попытается загрузить страницу в свободную память а процесс на это время будет заблокирован. По завершении работы менеджера памяти страница будет загружена и процесс будет продолжать выполнятся с той команды на которой возникло исключение. Если свободная страница в физической памяти отсутствует то менеджер памяти должен выбрать страницу для замещения.

Свойство локальности

Использовании ассоциативного буфера на 8 адресов при страничном преобразовании дает нам 90% скорости полностью ассоциативной памяти благодаря свойству локальности.

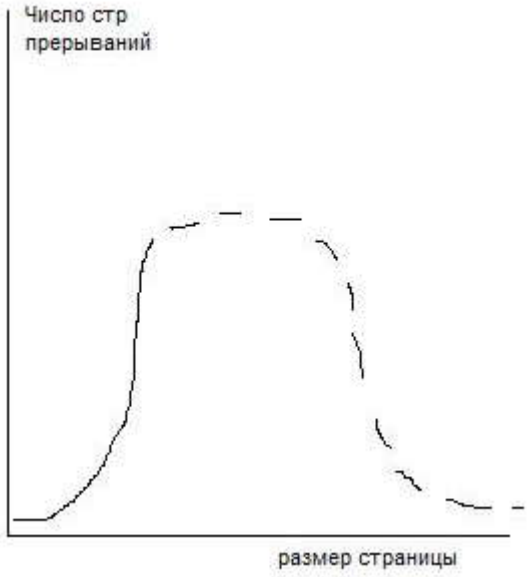
Локальность бывает 2х типов
1- ВременнАя – процесс обратившийся к одной странице наиболее вероятно в следующую единицу времени обратится к этой-же странице
2- Пространственная – процесс обратившийся к одной странице наиболее вероятно обратится к соседним страницам

Анализ страничного поведения процессов и рабочее множество

Для каждого процесса в каждый момент времени существует набор страниц которые он должен держать в памяти – рабочее множество. Если это набор не будет загружен возникнет трешинг страниц (постоянная загрузка и выгрузка страниц).

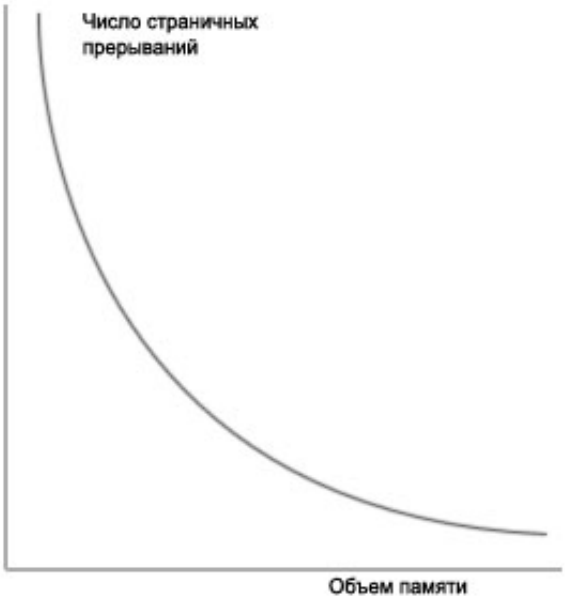
Размер страницы
- Чем меньше размер страницы, тем меньше суммарный объем фрагментации памяти, но больше объем таблицы страниц памяти.
- Чем больше размер страницы, тем меньше команд на ней реально выполняется

Аномалия размера страниц
-Увеличение числа страничных прерываний при увеличении числа страниц!

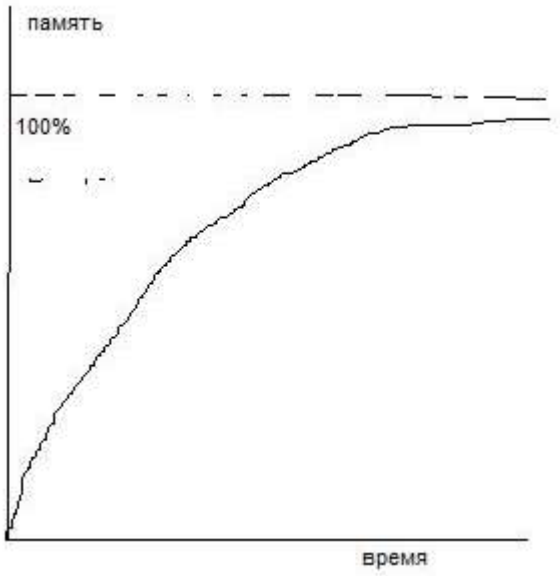


Увеличение размера страниц приводит к увеличению кол-ва страничных прерываний. Связано это с тем что при увеличении размера страницы в память попадает большое число операций и данных к которым обращение не выполняется тем самым уменьшается количество информации которое может быть загружено в память и выполнится.

Зависимость от объёма ОЗУ



Зависимость загрузки памяти от времени



Зависимость от количества кадров выделенных процессу

-Средний интервал между 2-мя страничными прерываниями называется временем жизни
-По мере загрузки рабочего множества растет интервал между страничными прерываниями
-Точка перегиба соответствует моменту когда число выделенных кадров становится равным его рабочему множеству, причем выделение дополнительных кадров не приводит к увеличению времени жизни страницы
Данный анализ доказывает наличие рабочего множества и то что его надо учитывать!

2. Прерывание от системного таймера в защищенном режиме: функции.

В операционной системе Windows

Каждый тик:
- подсчитывает тики аппаратного таймера;
- декрементирует счетчики отложенных вызовов;

По истечению кванта:
- вызывает функции, относящиеся к работе диспетчера ядра, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- по превышении выделенной квоты использования процессора посылает текущему процессу сигнал;
- обновляет статистику использования процессора текущим процессом;

Каждый главный тик
- поддерживает профилирование выполнения процессов в режимах ядра и задачи;

Прерывание в Unix - системах

Каждый тик:
- ведет счет тиков таймера;
- декрементирует счетчики отложенных вызовов;

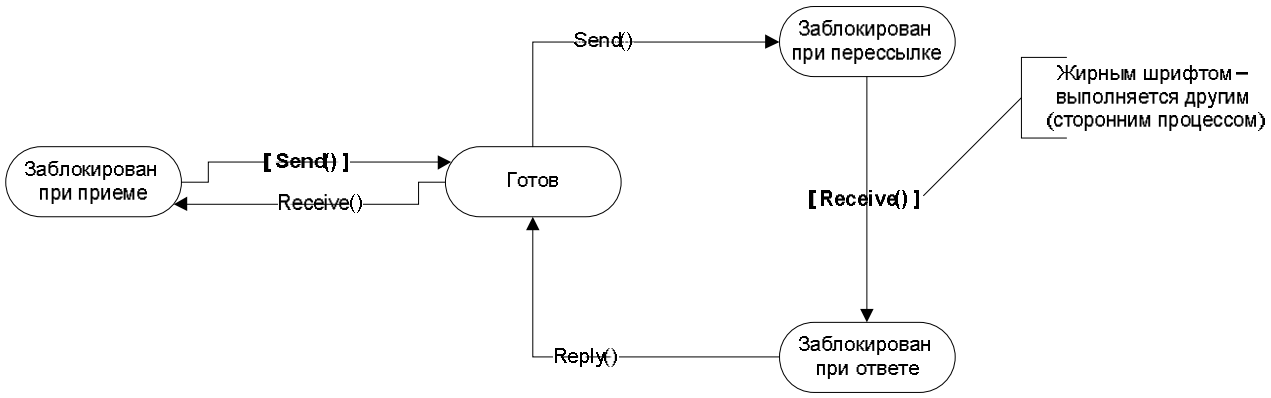
По истечению кванта
- вызывает процедуру обновления статистики использования процессора текущим процессом;
- вызывает функции, относящиеся к работе планировщика, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- посылает текущему процессу сигнал SI6XCPU, если тот превысил выделенную ему квоту использования процессора;

Каждый главный тик:
- пробуждает в нужные моменты системные процессы, такие как swapper и pagedaemon;
- поддерживает профилирование выполнения процессов в режимах ядра и задачи при помощи драйвера параметров;

1. Процессы: синхронизация процессов и алгоритмы взаимного исключения в в распределенных системах.

Процесс - программа в стадии выполнения. Единица декомпозиции системы, с той точки зрения, что именно ему выдаются ресурсы ОС. Может делиться на потоки, программист создает в своей программе потоки, которые выполняются квазипараллельно.

Диаграмма состояний процесса



Проблема синхронизации

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов.Она связана с потерей доступа к параметрам из-за их некоррректного разделения.
Критический ресурс - разделенная переменная,к которой обращаются разные процессы.
Критическая секция - строки кода,в кот происходит обращение к критическому ресурсу.
Необходимо обеспечить монопольный доступ процесса к критическому ресурсу до тех пор пока процесс его не освободит. Т.е. чтобы не могли одновременно войти в крит. секцию.

Алгоритмы взаимоисключения в распределенных системах.

В однопроцессорных системах решение задач взаимного исключения, критических областей и других проблем синхронизации осуществлялось с использованием общих методов, таких как семафоры и мониторы. Однако эти методы не совсем подходят для распределенных систем, так как все они базируются на использовании разделяемой оперативной памяти. Например, два процесса, которые взаимодействуют, используя семафор, должны иметь доступ к нему. Если оба процесса выполняются на одной и той же машине, они могут иметь совместный доступ к семафору, хранящемуся, например, в ядре, делая системные вызовы. Однако, если процессы выполняются на разных машинах, то этот метод не применим, для распределенных систем нужны новые подходы.
Системы, состоящие из нескольких процессоров, часто легче программировать, используя так называемые критические секции.
1. **Централизованный алгоритм**
Наиболее очевидный и простой путь. это попытка применения тех же методов, которые используются в однопроцессорных системах. Один из процессов выбирается в качестве координатора (например, процесс, выполняющийся на машине, имеющей наибольшее значение сетевого адреса). Когда какой-либо процесс хочет войти в критическую секцию, он посылает сообщение с запросом к координатору, оповещая его о

том, в какую критическую секцию(по какому разд. ресурсу) он хочет войти, и ждет от координатора разрешение. Если в этот момент ни один из процессов не находится в критической секции, то координатор посылает ответ с разрешением . Если же некоторый процесс уже выполняет критическую секцию, связанную с данным ресурсом, то никакой ответ не посылается; запрашивавший процесс ставится в очередь, и после освобождения критической секции ему отправляется ответ-разрешение.
Если процесс обнаружил отсутствие координатора,он инициирует его выборы. Для этого он посылает соответствующий запрос со своим номером по сети. Если процесс, получающий сообщение о начале выборов,имеет номер больше,то он посылает назад подтверждение,а сам инициирует новые выборы. В результате такой цепочки остается 1 процесс - он и станет координатором.
Этот алгоритм гарантирует взаимное исключение, но вследствие своей централизованной природы обладает низкой отказоустойчивостью.

2. **Распределенный алгоритм**

Когда процесс хочет войти в критическую секцию, он формирует сообщение(в лек - пакет с информацией), содержащее имя нужной ему критической секции, номер процесса и текущее значение времени. Затем он посылает это сообщение всем другим процессам. Предполагается, что передача сообщения надежна, то есть получение каждого сообщения сопровождается подтверждением. Когда процесс получает сообщение такого рода, его действия зависят от того, в каком состоянии по отношению к указанной в сообщении критической секции он находится. Имеют место три ситуации:
Если получатель не находится и не собирается входить в критическую секцию в данный момент, то он отсылает назад процессу-отправителю сообщение с разрешением.
Если получатель уже находится в критической секции, то он не отправляет никакого ответа, а ставит запрос в очередь.
Если получатель хочет войти в критическую секцию, но еще не сделал этого, то он сравнивает временную отметку поступившего сообщения со значением времени, которое содержится в его собственном сообщении, разосланном всем другим процессам. Если время в поступившем к нему сообщении меньше, то есть его собственный запрос возник позже, то он посылает сообщение-разрешение, в обратном случае он не посылает ничего и ставит поступившее сообщение-запрос в очередь.

Процесс может войти в критическую секцию только в том случае, если он получил ответные сообщения-разрешения от всех остальных процессов. Когда процесс покидает критическую секцию, он посылает разрешение всем процессам из своей очереди и исключает их из очереди.

3.**Алгоритм Token Ring**

Все процессы системы образуют логическое кольцо, т.е. каждый процесс знает номер своей позиции в кольце, а также номер ближайшего к нему следующего процесса. Когда кольцо инициализируется, процессу 0 передается так называемый токен(сообщение определенного формата). Токен циркулирует по кольцу. Он переходит от процесса n к процессу n+1 путем передачи сообщения по типу "точка-точка". Когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему самому войти в критическую секцию. Если да, то процесс входит в критическую секцию. После того, как процесс выйдет из критической секции, он передает токен дальше по кольцу. Если же процесс, принявший токен от своего соседа, не заинтересован во вхождении в критическую секцию, то он сразу отправляет токен в кольцо. Следовательно, если ни один из процессов не желает входить в критическую секцию, то в этом случае токен просто циркулирует по кольцу с высокой скоростью.

Сравним эти три алгоритма взаимного исключения. Централизованный алгоритм является наиболее простым и наиболее эффективным. При его использовании требуется только три сообщения для того, чтобы процесс вошел и покинул критическую секцию: запрос и сообщение-разрешение для входа и сообщение об освобождении ресурса при выходе. При использовании распределенного алгоритма для одного использования критической секции требуется послать (n-1)

сообщений-запросов (где n - число процессов) - по одному на каждый процесс и получить (n-1) сообщений-разрешений, то есть всего необходимо 2(n-1) сообщений. В алгоритме Token Ring число сообщений переменно: от 1 в случае, если каждый процесс входил в критическую секцию, до бесконечно большого числа, при циркуляции токена по кольцу, в котором ни один процесс не входил в критическую секцию.
все три алгоритма плохо защищены от отказов. В первом случае к краху приводит отказ координатора, во втором - отказ любого процесса (парадоксально, но распределенный алгоритм оказывается менее отказоустойчивым, чем централизованный), а в третьем - потеря токена или отказ процесса.

Неделимые транзакции

Все средства синхронизации, которые были рассмотрены ранее, относятся к нижнему уровню, например, семафоры. Они требуют от программиста детального знания алгоритмов взаимного исключения, управления критическими секциями, умения предотвращать клинчи (взаимные блокировки), а также владения средствами восстановления после краха. Однако существуют средства синхронизации более высокого уровня, которые освобождают программиста от необходимости вникать во все эти подробности и позволяют ему сконцентрировать свое внимание на логике алгоритмов и организации параллельных вычислений. Таким средством является неделимая транзакция.

Модель неделимой транзакции пришла из бизнеса. Представьте себе переговорный процесс двух фирм о продаже-покупке некоторого товара. В процессе переговоров условия договора могут многократно меняться, уточняться. Пока договор еще не подписан обеими сторонами, каждая из них может от него отказаться. Но после подписания контракта сделка (transaction) должна быть выполнена.

Компьютерная транзакция полностью аналогична. Один процесс объявляет, что он хочет начать транзакцию с одним или более процессами. Они могут некоторое время создавать и уничтожать разные объекты, выполнять какие-либо операции. Затем инициатор объявляет, что он хочет завершить транзакцию. Если все с ним соглашаются, то результат фиксируется. Если один или более процессов отказываются (или они потерпели крах еще до выработки согласия), тогда измененные объекты возвращается точно к тому состоянию, в котором они находились до начала выполнения транзакции. Такое свойство "все-или-ничего" облегчает работу программиста.
Примитивы:

EGIN_TRANSACTION команды, которые следуют за этим примитивом, формируют транзакцию.

END_TRANSACTION завершает транзакцию и пытается зафиксировать ее.

ABORT_TRANSACTION прерывает транзакцию, восстанавливает предыдущие значения.

READ читает данные из файла (или другого объекта)

WRITE пишет данные в файл (или другой объект).

Первые два примитива используются для определения границ транзакции. Операции между ними представляют собой тело транзакции. Либо все они должны быть выполнены, либо ни одна из них. Это может быть системный вызов, библиотечная процедура или группа операторов языка программирования, заключенная в скобки.

Транзакции обладают следующими свойствами: упорядочиваемостью, неделимостью, постоянством.

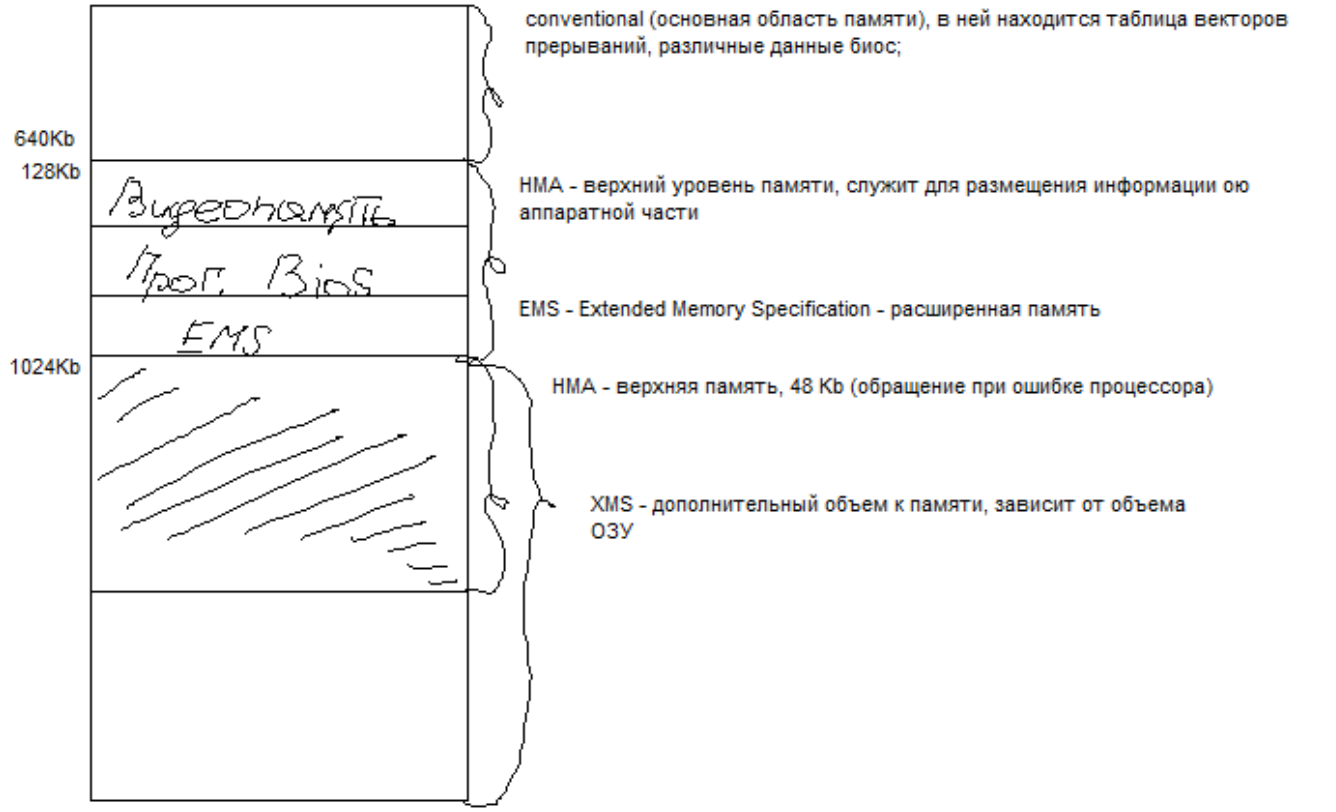
Упорядочиваемость гарантирует, что если две или более транзакции выполняются в одно и то же

время, то конечный результат выглядит так, как если бы все транзакции выполнялись последовательно в некотором (в зависимости от системы) порядке.

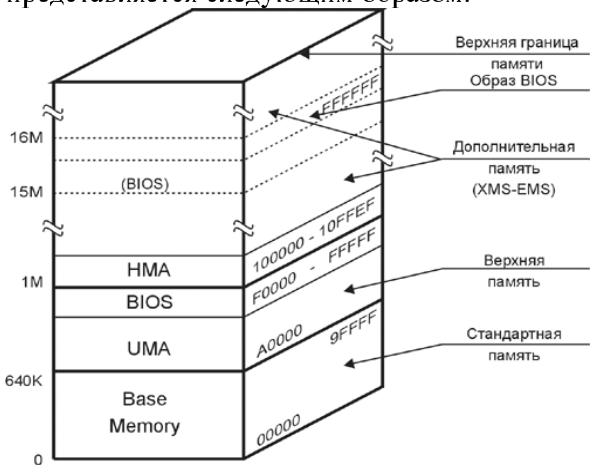
Неделимость означает, что когда транзакция находится в процессе выполнения, то никакой другой процесс не видит ее промежуточные результаты.

Постоянство означает, что после фиксации транзакции никакой сбой не может отменить результатов ее выполнения.

2. Спецификация XM (XMS):области - Conventional, HMA, UMA, EMA, адресное заворачивание.



Распределение памяти PC, непосредственно адресуемой процессором, приведено на рисунке 2 и представляется следующим образом.



00000h-9FFFFh - Conventional (Base) Memory, 640 Кбайт - стандартная (базовая) память, доступная DOS и программам реального режима. В некоторых системах с видеоадаптером MDA верхняя граница сдвигается к AFFFFh (704 Кбайт). Иногда верхние 128 Кбайт стандартной памяти (область 80000h-9FFFFh) называют Extended Conventional Memory.
Стандартная память является самой дефицитной в PC, когда речь идет о работе в среде операционных систем типа MS-DOS. На ее небольшой объем (типовое значение 640 Кбайт) претендуют и BIOS, и ОС реального режима, а остатки отдаются прикладному ПО. Стандартная память распределяется следующим образом:
00000h-003FFh - Interrupt Vectors - векторы прерываний (256 двойных слов);
00400h-004FFh - BIOS Data Area - область переменных BIOS;
00500h-00xxxh - DOS Area - область DOS;
00xxxh-9FFFFh - User RAM - память, предоставляемая пользователю (до 638 Кбайт);
при использовании PS/2 Mouse область 9FC00h-9FFFFh используется как расширение BIOS Data Area, и размер User RAM уменьшается.

A0000h-FFFFFh - Upper Memory Area (UMA), 384 Кбайт - верхняя память, зарезервированная для системных нужд. В ней размещаются области буферной памяти адаптеров (например, видеопамять) и постоянная память (BIOS с расширениями). Эта область, обычно используемая не в полном объеме, ставит непреодолимый архитектурный барьер на пути непрерывной (нефрагментированной) памяти, о которой мечтают программисты. Верхняя память имеет области различного назначения, которые могут быть заполнены буферной памятью адаптеров, постоянной памятью или оставаться незаполненными. Раньше эти “дыры” не использовали из-за сложности “фигурного выпиливания” адресуемого пространства. С появлением механизма страничной переадресации (у процессоров 386 и выше) их стали по возможности заполнять “островками” оперативной памяти, названными блоками верхней памяти UMB (Upper Memory Block). Эти области доступны DOS для размещения резидентных программ и драйверов через драйвер EMM386, который отображает в них доступную дополнительную память.

Память выше 100000h - Extended Memory - дополнительная (расширенная) память, непосредственно доступная только в защищенном (и в “большом реальном”) режиме для компьютеров с процессорами 286 и выше. В ней выделяется область 100000h-10FFEFh - высокая память, HMA, - единственная область расширенной памяти, доступная 286+ в реальном режиме при открытом вентиле Gate A20.

Расширенная память XMS (eXtended Memory Specification) - программная спецификация использования дополнительной памяти DOS-программами, разработанная компаниями Lotus, Intel, Microsoft и AST для компьютеров на процессорах 286 и выше. Эта спецификация позволяет программе получить в распоряжение одну или несколько областей дополнительной памяти, а также использовать область HMA. Распределением областей ведает диспетчер расширенной памяти - драйвер HIMEM.SYS. Диспетчер позволяет захватить или освободить область HMA (65 520 байт, начиная с 100000h), а также управлять вентилем линии адреса A20. Функции собственно XMS позволяют программе:
- определить размер максимального доступного блока памяти;
- захватить или освободить блок памяти;
- копировать данные из одного блока в другой, причем участники копирования могут быть блоками как стандартной, так и дополнительной памяти в любых сочетаниях;
- запереть блок памяти (запретить копирование) и отпереть его;
- изменить размер выделенного блока.

Спецификации EMS и XMS отличаются по принципу действия: в EMS для доступа к дополнительной памяти выполняется отображение (страничная переадресация) памяти, а в XMS - копирование блоков данных.

Адресное заворачивание:
Процессор в реальном режиме поддерживает адресное пространство до 1Мбайт. Адресное пространство

разбито на сегменты по 64Кбайт. 20-битный базовый адрес сегмента вычисляется сдвигом значения селектора на 4 бита влево. Данные внутри сегмента адресуются 16-битным смещением.

В этом режиме формирования линейного адреса есть возможность адресовать пространство между 1Мб и 1Мб+64Кб (например, указав в качестве селектора 0FFFFh, а в качестве смещения 0FFFFh, мы получим линейный адрес 10FFEFh). Однако МП 8086, обладая 20-разрядной шиной адреса, отбрасывает старший бит, "заворачивая" адресное пространство (в данном примере МП 8086 обратится по адресу 0FFEFh). В реальном режиме микропроцессоры IA-32 "заворачивания" не производят. Для 486+ появился новый сигнал - A20M#, который позволяет блокировать 20-й разряд шины адреса, эмулируя таким образом "заворачивание" адресного пространство, аналогичное МП 8086.

1. Взаимодействие процессов: монопольное использование – программная реализация взаимоисключения, взаимоисключение с помощью семафоров; сравнение – достоинства и недостатки

Монопольный доступ осуществляется взаимоисключением, т.е. процесс, получивший доступ к разделяемой переменной, исключает доступ к ней др. процессов.

Программная реализация (алгоритм Деккера).

Деккер – голландский математик. Предложил способ свободный от бесконечного откладывания.

```
flag1, flag2: logical;
queue: int;

p1: while(1)
    flag1 = 1;
    while(flag2)
            if(queue == 2) then
            begin
                    flag1 = 0;
                    while(queue == 2);
                    flag1 = 1;
            end;
    CR1;
    flag1 = 0;
    queue = 2;
    PR1;
end P1;

p2: while(1)
            flag2 = true;
    while(flag1)
            if(queue == 1) then
            begin
                    flag2 = 0;
                    while(queue == 1);
                    flag2 = 1;
            end;
    CR2;
    flag2 = 0;
    queue = 1;
    PR2;
end P2;

flag1 = 0;
flag2 = 0;
parbegin
P1;
P2;
parend;
```

queue – очередь процесса входить в критическую секцию.
Недостаток метода – активное ожидание на процессоре.
Активное ожидание – ситуация, когда процесс занимает процессорное время, проверяя значение флага. Активное ожидание на процессоре является неэффективным использованием процессорного времени.

Семафоры

(устраняют активное ожидание на процессоре)

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции: P (от датск. passeren - пропустить) и V(от датск. vrygeven - освбодить).
Операция V(S): означает увеличение значения S на 1 одним неделимым действием (последовательнось непрерывных действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0, то V(S) приведёт к V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожидающего на семафоре.
Операция P(S): означает декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить операция P, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой процесс, выполняющий операцию V(S).

S может быть изменена только операциями P(S) и V(S). Это и есть защищённость S.
P(S) и V(S) есть неделимые (атомарные) операции.
Суть: процесс пытающий выполнить операцию P(S) блокируется, становится в очередь ожидания данного семафора, освобождает его другой процесс, который выполняет V(S). Таким образом исключается активное ожидание.
Семафоры бывают:

1) бинарные (S принимает значения 0 и 1)
2) считающие (S принимает значения от 0 до n)
3) множественные (массив считающих семафоров)

Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо привелегированный процесс.
Достоинства: Изменение семафора может быть использовано как событие в системе => устраняется активное ожидание на процессоре.

2. Unix: процессы - "сироты", "зомби", "демоны" - возникновение, особенности работы ОС с каждым типом процессов.

Процесс многократно переходит из одного состояния в другое. Из режима пользователя в режим ядра и наоборот. Концепция: процесс выполняется в одной из двух стадий:

1. пользователь/задача;
2. система.

В стадии «задача» процесс выполняет собственный код, в стадии «система» процесс выполняет реентерабельный код ядра (не модифицирует сам себя).

В теле процедуры можно изменить данные. Для реентерабельной процедуры необходимо вынести данные из тела процедуры.

Несколько процессов могут находится одновременно в разных точках одной и той же процедуры. Ядро Unix полностью реентерабельно! Unix система разделения времени.

«сирота» — возникает в том случае, если процесс предок завершился раньше своих потомков. При завершении процесса система проверяет не осталось ли у этого процесса незавершенных потомков. Если остались – система принимает действия по их усыновлению. Процесс потомок усыновляется терминальным процессом. Система переписывает индефикатор такого процесса на 1.

«зомби» — если процесс потомок завершился до того как предок вызвал wait (возможно при аварийном завершении exec), то для того чтобы предок не завис в ожидании несуществующего процесса, система отбирает у него все ресурсы и помещает строку в таблице процессов, помечая такой процесс как зомби. Сделано это для того, чтобы процесс получил статус завершения всех своих потомков.
Чтобы родитель не завис система отбирает у потомка все ресурсы, но оставляет строку в таблице процессов.

«демон» — процесс, который не имеет предков (сервисные функции)

1. Тупики: определение тупиковой ситуации для повторно используемых ресурсов, четыре условия возникновения тупика, обход тупиков - алгоритм банкира.

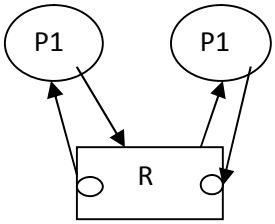
1)Бесконечное откладывание может возникнуть, если процесс не может в необходимое время получить нужный ему ресурс

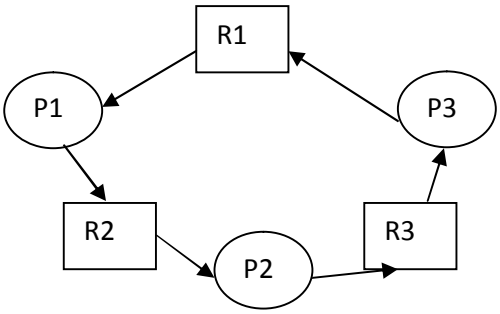
2)Тупики. Взаимная блокировка

3) Захват и освобождение одних и тех же ресурсов

В системе могут быть только эти три ситуации.

Тупиковая ситуация – тупик – система, возникающая в результате монопольного использования разделенных ресурсов, когда занятый процесс, запрашивает ресурс, занятый другим процессом или запросом через цепочку других процессов , который при этом ожидает ресурс, занятый первым процессом.





Ресурсы с точки зрения их использования можно разделить на два типа:

1 Повторно используемые

2 Потребляемые ресурсы

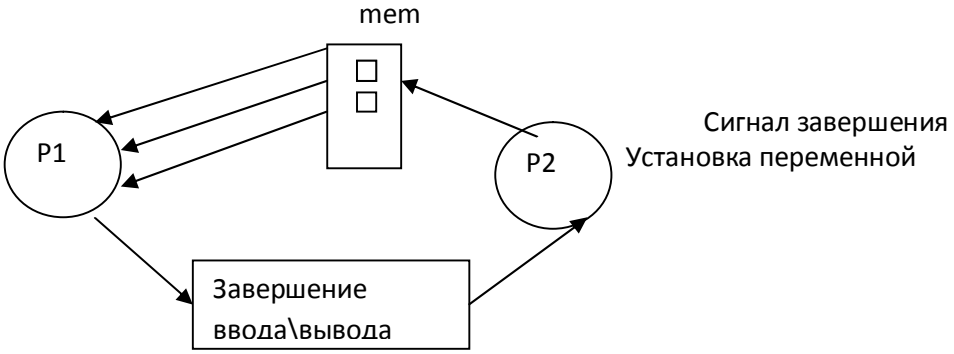
К первым относятся: аппаратура, реентирабельнве коды, т е процесс, захвативший ресурс, возвращает его таким же, каким он был до захвата. Количество этих ресурсов в системе не меняется.

Ко вторым относятся: сообщения, по особенности сообщения можно сказать: при потреблении сообщение перестает существовать. Количество сообщений в системе неограниченно. В этом сложность данного типа ресурса.

Вся теория тупиков излагается для повторно используемых ресурсов , на потребляемые ресурсы проицируется эта теория, сводя ситуации к известным.

Следует отметить, что тупик возможен в смешанной ситуации.

Пример такого тупика:



Четыре условия возникновения тупиков:

1 Монопольное использование ресурсов процессами реализованное механическим взаимоисключением

2 Ожидание – ситуация, в которой процессы удерживают занимаемые ими ресурсы, ожидая предоставления им необходимых ресурсов, занятыми другим процессом

3 Неперераспределяемость – невозможность отобрать у процесса занимаемый им ресурс до завершения процесса или до того, как процесс сам освободит ресурс.

4 Круговое ожидание – ситуация, когда возникает замкнутый процесс, т е процесс занимает ресурс необходимый для продолжения выполнения другого процесса в цепи.

Методы борьбы с тупиками:

1 Предотвращение. Создание такой ситуации, когда тупики невозможны

2 Недопущение или обход. Тупики в принципе возможны, но выполняется такая операция, которая гарантирует, что тупик не наступит.
3 Обнаружение и восстановление
Первый способ борьбы с тупиками получил название стратегии Хорендера. Он формализовал методы исключения тупиковых ситуаций. Согласно этой стратегии сущ три основных способа:

1 Опережающие требования
2 Реализация путем упорядочивания ресурсов
3 Исключение условия ожидания

Алгоритм Банкира(предложил Дейкстра)
Действие этого алгоритма связывают с действиями банкира, который владеет ресурсами (деньгами) . Он отдает эти деньги, с целью вернуть назад. Однако, чтобы вернуть заем, заемщику нужен еще один заем. И он решает кому этот заем дать, кто гарантированно вернет.
В качестве банкира в системе – менеджер процессов. Заемщики – процессы. Процессы указывают в своих заявках максимальную потребность в ресурсе, при этом процесс не может потребовать ресурсов больше, чем у него в заявке. Это позволяет менеджеру проводить предварительный анализ и принять правильное решение.
Для выполнения необходимы следущие условия:
1 Число процессов фиксированно
2 Число ресурсов фиксированно
3 Процесс не может запросить больше ресурсов, чем имеется в системе
4 Процесс не может запросить и получить больше процессов, чем указанно в заявке
5 Сумма всех распределенных ресурсов данного класса не может превосходить общего количества ресурсов плана в системе
Каждый запрос проводится по отношению к количеству ресурсов в системе, каждая заявка проверяется относительно суммы всех заявок на ресурс. Менеджер ресурсов анализирует ситуацию распределения ресурсов в системе (при получении заявки) , ищет такую последовательность процессов в системе, которая может гарантированно завершится.
Эта ситуация является безопасной по отношению к тупику, т к тут можно выделить последовательность процессов , которые могут гарантированно завершится. Таким образом менеджер ресурсов определяет эту ситуацию как безопасную и выделяет ресурсы.

2. Win32 API : CreateThread(), WaitForSingleObject(), WaitForMultipleObject().

- Функция CreateThread создает поток в адресном пространстве процесса. Вид:

```
HANDLE CreateThread(
LPSECURITY_ATTRIBUTES lpThreadAttributes, // указатель на атрибуты
// безопасности потока
   DWORD dwStackSize,     // размер инициализируемого
// стека потока в байтах
   LPTHREAD_START_ROUTINE lpStartAddress, // указатель на функцию потока
   LPVOID lpParameter,     // аргумент для нового потока
   DWORD dwCreationFlags,    // флаг создания
   LPDWORD lpThreadId      // указатель на возвращаемый
// номер потока
  );
```

Указатель на структуру SECURITY_ATTRIBUTES определяет, может ли возвращаемый поток наследоваться дочерними процессами. Если lpThreadAttributes - NULL, поток не может быть унаследован.

dwStackSize - определяет размер стека для нового потока в байтах. Если 0, то устанавливается размер по умолчанию, равный размеру стека первичного потока. Стек распределяется автоматически в пространстве памяти процесса, и освобождается по завершению потока. Размер стека возрастает при необходимости.

lpStartAddress - стартовый адрес нового потока. Обычно это адрес WINAPI функции, принимающей единственный 32-битовый указатель как аргумент и возвращающей 32-битовый выходной код.

Прототип: DWORD WINAPI MyThreadFunc(LPVOID);
lpParameter - определяет единственный 32-битный параметр, передающийся в поток.

dwCreationFlags - определяет дополнительные флаги, которые управляют созданием потока. Если установлен флаг CREATE_SUSPENDED, то поток создается в приостановленном состоянии, и не запускается пока не вызвана функция ResumeThread. Если dwCreationFlags = 0, поток запустится на выполнение немедленно после создания.

lpThreadId - Указатель на 32-битную переменную, возвращающую идентификатор потока.

Если функция выполнилась успешно, она возвращает номер (handle) нового потока, иначе NULL (описание ошибки в GetLastError).

- Функция WaitForSingleObject - ожидание завершения потока.

DWORD WaitForSingleObject(HANDLE hHandle, DWORD dwMilliseconds);

Возвращаемое значение
Если функция успешно завершает свою работу, то возвращаемое значение указывает на причину завершения работы функции. Оно может принимать одно из следующих значений:
WAIT_ABANDONED - указанный объект является мютексом, который не был освобожден потоком, которому он принадлежит, перед завершением данного потока. Принадлежность мютекса вызывающему потоку гарантируется. Поэтому этот объект остался неотмеченным;
WAIT_OBJECT_0 -указанный объект находится в отмеченном состоянии;
WAIT_TIMEOUT - истек период ожидания, а объект остался неотмеченным.
Если функция аварийно завершает свою работу, она возвращает значение WAIT_FAILED. Более подробную информацию об ошибке можно получить, вызвав функцию GetLastError.

Аргументы
hHandle - дескриптор объекта. Список типов объектов, дескрипторы которых могут использоваться в качестве данного аргумента, содержится в примечании. В Windows NT дескриптор должен иметь уровень доступа SYNCHRONIZE.
dwMilliseconds - определяет интервал времени, измеряемый в миллисекундах. По истечении этого интервала времени, если указанный объект остается неотмеченным, функция завершает свою работу. Если данный аргумент имеет нулевое значение, функция проверяет состояние объекта и немедленно прекращает свою работу. Если аргумент dwMilliseconds имеет значение INFINITE, то функция ждет отметки объекта неограниченное время.

Описание
Функция WaitForSingleObject возвращает свое значение в двух случаях: когда указанный объект устанавливается в отмеченное состояние; когда истекает время ожидания. Данная функция проверяет текущее состояние указанного объекта. Если данный объект находится в неотмеченном состоянии, выполнение потока приостанавливается. В процессе ожидания поток практически не использует процессорное время. Перед завершением своей работы функция изменяет состояние некоторых объектов синхронизации. Изменения происходят только в том случае, если изменение состояния объекта привело к выходу из функции. Например, счетчик семафора уменьшается на единицу.
Функция WaitForSingleObject может использоваться со следующими объектами:

извещениями об изменениях;
вводом с системной консоли;
объектами событий;
заданиями;
мютексами;
процессами;
семафорами;
потоками;
таймерами ожидания.
Необходимо соблюдать известную осторожность при вызове функций ожидания и программ, прямо или косвенно создающих объекты окон. Если поток создает любое окно, оно должно обрабатывать сообщения. Сообщения посылаются всем окнам системы. Поток, использующий функцию ожидания без интервала ожидания, может "подвесить" систему.

- **WaitForMultipleObject**

Часто одна задача должна дожидаться завершения сразу нескольких задач или процессов, либо одной из нескольких задач или процессов. Такое ожидание нетрудно выполнить с помощью функции WaitForMultipleObjects, прототип которой приведен ниже:

```
DWORD WaitForMultipleObjects(
DWORD cObjects, // количество идентификаторов в массиве
CONST HANDLE *lphObjects, // адрес массива идентификаторов
BOOL fWaitAll, // тип ожидания
DWORD dwTimeout); // время ожидания в миллисекундах
```

Через параметр lphObjects функции WaitForMultipleObjects нужно передать адрес массива идентификаторов. Размер этого массива передается через параметр cObjects.
Если содержимое параметра fWaitAll равно TRUE, задача переводится в состояние ожидания до тех пор, пока все задачи или процессы, идентификаторы которых хранятся в массиве lphObjects, не завершат свою работу. В том случае, когда значение параметра fWaitAll равно FALSE, ожидание прекращается, когда одна из указанных задач или процессов завершит свою работу. Для выполнения бесконечного ожидания, как и в случае функции WaitForSingleObject, через параметр dwTimeout следует передать значение INFINITE.

1. Процессы: бесконечное откладывание, зависание, тупиковая ситуация - анализ на примере задачи об обедающих философах. Считающие и множественные семафоры. Мониторы: монитор кольцевой буфер.

Критическая секция – место в программе, где осуществляется работа с разделяемыми переменными (ресурсами).

Для того, чтобы обеспечить корректную работу с разделяемыми переменными необходимо обеспечить монопольный доступ к ним, он реализуется с помощью взаимоисключения(невозможность доступа к переменной другим процессам при выполнении критической секции процесса по этой разделенной переменной)

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критичские секции - ок

2. Возможно **бесконечное откладывание (зависание) –** ситуация, когда разделённый ресурс снова захватывается тем же процессом.

3. **Тупик (deadlock, взаимоблокировка) –** ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом.

Проблема обедающих философов
В 1965 году Дейкстра сформулировал и решил проблему синхронизации:

Задача: за круглым столом сидит 5 философов и пытаются есть спагетти. философ должен взять в обе руки по вилке, т.к. с одной вилки спагетти сваливаются. С каждой стороны от тарелки лежит только одна вилка.

Существуют только 3 схемы действия философов:

1) философ пытается взять обе вилки сразу. если удается, он может есть.

2) философ берет левую вилку и пытается взять правую (левую держит в руке)

3) философ берет левую вилку, и если не удается взять правую, то кладет левую вилку обратно.

Бесконечное откладывание. Например все взяли левую вилку, видят правой нет — положили. Снова взяли и т.д. Процесс висит. **Тупик.** Все философы взяли по левой вилке и сидят ждут когда освободится правая.

Семафоры(устраняют активное ожидание на процессоре)

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции:

P (от датск. passeren - пропустить) и V(от датск. vrygeven - освбодить).

Операция V(S): означает увеличение значения S на 1 одним неделимым действием (последовательнось непрерывных действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0, то V(S) приведёт к V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожидающего на семафоре.

Операция P(S): означает декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить операция P, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой процесс, выполняющий операцию V(S).

S может быть изменена только операциями P(S) и V(S). Это и есть защищённость S.

P(S) и V(S) есть неделимые (атомарные) операции.

Суть: процесс пытающий выполнить операцию P(S) блокируется, становится в очередь ожидания данного семафора, освобождает его другой процесс, который выполняет V(S). Таким образом исключается активное ожидание.

Семафоры бывают:

1) бинарные (S принимает значения 0 и 1)
2) считающие (S принимает значения от 0 до n)
3) множественные (массив считающих семафоров)

Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо привелегированный процесс.

Изменение переменной S можно рассматривать как событие в системе.

Использование множественных семафоров для решения задачи об обедающих философах:

```
var:
  forks: array[1..5] of Semaphore;
  i:     integer;

begin
  i := 5;
  repeat
        forks[i] := 1;
        i := i – 1;
  until i = 0;
  parbegin
        1:
        begin
              left := 1;
              right := 2;
              ...
        ...
        5:
        begin
              var:
              left, right: 1..5;
              begin
                    left := 5;
                    right := 1;
                    repeat
                          <думает>
                          P(forks[left], forks[right]); //взять обе вилки
                          <ест>
                          V(forks[left], forks[right]); //положить обе вилки
                    forever;
                    end;
              end;
  parend;
end;
```

Каждый примитив P и V проводит обработку сразу двух. При выполнении процесс м б заблокирован на одном/другом/двух сразу.

Множественные семафоры обладают возможностью проверки в одном примитиве сразу нескольких семафоров и их обработку по определённым правилам. Множественные семафоры требуют более сложных действий от ОС, т.к. проверяется больше очередей, семафоров и множественные условия.

<u>Мониторы</u>

К средствам нижнего уровня относятся критические секции, сообщения, мьютексы. Мониторы же призваны обобщить средства нижнего уровня.

Проблема, связанная с семафорами состоит в том, что они не структурны. Из-за этого ввели структурные мониторы. Монитор может быть языковым средством, а может быть реализован в ОС.

Монитор - языковая конструкция, состоящая из структуры данных и набора подпрограмм, работающих с этой структурой, которой в каждый момент времени может пользоваться только один процесс.

- Мониторами удобно собрать все разделяемые процессами объекты, а именно переменные и семафоры.
- Мониторы должны обладать возможностью активизирования и блокировки процессов.
- Монитор защищает данные, объявленные в своей структуре, а именно, доступ к этим данным может быть осуществлён только с помощью процедур, включённых в тело монитора.
- Если какой-либо процесс находится в мониторе, то любой другой процесс ставится в очередь (переводится в состояние ожидания) вне монитора.

Монитор сам является ресурсом, поэтому его использование производится в режиме взаимоисключения. Сущ. несколько типов мониторов, которые появились в связи с важностью решаемых задач:

Монитор **кольцевой буфер** – это структура данных, широко применяемая в ОС для буферизации обменов между процессами-производителями и процессами-потребителями.

В частности этот механизм используется для реализации спулинга (spooling) — используется процессами, которые выводят на печать.

Спулер – процесс, записывающий данные, деспулер – выбирающий. Процессы-производители подготавливают и выдают строки, процессы-потребители печатают их. Процессы формируют строки и помещают их в кольцевой буфер. Нужно иметь иллюзию того, что память не кончается, поэтому кольцвой буфер и используется для этого. Он может частично и полностью размещаться в ОП.

2. Синхронизация процессов ОС Unix на примере задачи «производство-потребление».

Синхронизация - такая связь между процессами при которой один процесс не может выполняться начиная с определенного момента, до тех пор пока другой процесс не достигнет своей определенной точки (например, запись в буфер и чтение из буфера).

```
#define Se 0 // свободные ячейки
#define Sf 1 // занятые ячейки
#define S 2  // бинарный

struct sembuf pp[2] = {{Se, -1, SEM_UNDO}, {S, -1, SEM_UNDO}};
struct sembuf vp[2] = {{Sf, 1, SEM_UNDO}, {S, 1, SEM_UNDO}};

struct sembuf pc[2] = {{Sf, -1, SEM_UNDO}, {S, -1, SEM_UNDO}};
struct sembuf vc[2] = {{Se, 1, SEM_UNDO}, {S, 1, SEM_UNDO}};

char* addr;

int producer(int sem_id)
{
        while (1)
        {
                semop(sem_id, &pp[0], 2);
                (*addr)++;
                printf("Produce - %d\n", (*addr));
                semop(sem_id, &vp[0], 2);
        }
}

int consumer(int sem_id)
{
        while (1)
        {
                semop(sem_id, &pc[0], 2);
                printf("Consume - %d\n", (*addr));
                (*addr)--;
                semop(sem_id, &vc[0], 2);
        }
}
```

1 Процессы: процесс как единица декомпозиции системы,. Контекст процесса. Переключение контекста. Классификация алгоритмов планирования; алгоритм адаптивного планирования; ситуация - бесконечное откладывание – причины возникновения

Процесс - программа в стадии выполнения. Единица декомпозиции системы, с той точки зрения, что именно ему выдаются ресурсы ОС. Может делиться на потоки, программист создает в своей программе потоки, которые выполняются квазипараллельно.

Представление процесса в ОС

Черты процесса:

1) владение ресурсами (resource ownership). Процесс владеет:
   a. защищенным, виртуальным адресным пространством, которое содержит образ процесса с момента запуска.
   b. основной памятью, каналами, устройствами ввода/вывода, файлами, разделяемыми ресурсами (очереди сообщений, семафоры, сигналы, файлы, отображения в память).
2) планирование и выполнение
   a. выполнение – выполнение кода программы если выделен квант процессорного времени.
   b. план может быть передан задаче (потоку – thread).

Адресное пространство выделяется процессу. За ресурсы системы конкурируют потоки (в первую очередь за процессорное время).
Следовательно, поток – единица диспетчеризации.

Владение ресурсами, планирование и диспетчеризация.

Диспетчеризация – выделение процессу процессорного времени.
Вытеснение основано на системе с приоритетами. Если все процессы равноправны, то вытеснения нет. Процессорное время передается процессу с более высоким приоритетом. Статические назначаются в начале и с течением времени не меняются.
Динамические меняются в течение жизни.

Планирование процессов – управление распределением ресурсов центральным процессором между конкурирующими процессами, путем передачи упраления согласно некоторой стратегии планирования.
Планировщик – программа, которая отвечает за управление использования совместного ресурса. Доступ к ресурсу, который используется совместно, предоставляется с учетом двух требований

1. Необходимо убедиться, что процесс не будет поврежден сам и не повредит другие процессы.
2. Выбор между несколькими процессами, при возможности доступа любого к ресурсу, осуществляется некоторыми алгоритмами планирования.

В ОС UNIX процесс может находиться в 2 состояниях: состоянии ядра и состоянии задачи. В состоянии задачи выполняется код процесса, в состоянии ядра выполняется код ядра. Процесс не может обратиться напрямую к ядру, он обращается к ядру путём системных вызовов.
В ОС Windows используется 2 уровня привилегий: 0 уровень и 3 уровень. 0 уровень – уровень привилегий ядра, 3 уровень – пользовательский процесс.
Супервизорный процесс – ядро в стадии выполнения.
При выполнении системного вызова запускается система команд – переключатель режимов – переводит систему в режим ядра. При завершении обработки вызывается система команд, возвращающая систему в режим задачи.
Пр: примером системного вызова является запрос IO.

Контекст выполнения

Функции ядра могут выполняться в контексте процесса, либо в системном контексте. При выполнении в контексте процесса ядро функционирует от имени текущего процесса, имея доступ к данным текущего

процесса, стеку ядра, адресному пространству процесса (всё это называют user area). Ядро может заблокировать процесс.
При выполнении в контексте ядра, оно обслуживает прерывание от внешних устройств и выполняет пересчёт приоритетов процессов. Всё это выполняется в системном контексте (контексте прерываний).
Переключение – мультизадачность.

Классификация планирования:

I)
a. Без переключения
b. С переключением

II)
a. Без вытеснения
b. С вытеснением

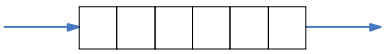
III)
a. Без приоритетов
b. Приоритетное
i. Статическое
ii. Динамическое

Приоритеты: абсолютные и относительные.
I) a. Процессорное время выделяется процессу, который выполняется от начала до конца.
III) b. Процесс с более высоким приоритетом имеет привилегированное положение (процесс с более низким приоритетом снимается с выполнения, если в очередь процессов помещен процесс с более высоким приоритетом.

Активное ожидание на процессоре – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом) Активное ожидание на процессоре является неэффективным использованием процессорного времени.
1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критичские секции - ок
2. Возможно **бесконечное откладывание (зависание) –** ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. **Тупик (deadlock, взаимоблокировка) –** ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом
Бесконечное откладывание – ситуация, когда процесс никогда не получает необходимых для выполнения ресурсов (точнее, кванта времени). Возникает когда диспетчер всегда отдаёт квант какому-то другому процессу, так как его приоритет больше.

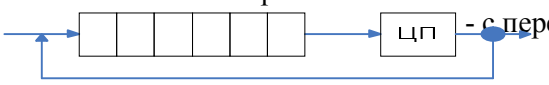
Алгоритмы планирования

1) FIFO – простая очередь



без переключения
без приоритетов
бесконечное откладывание исключено

2) RoundRobin-алгоритм

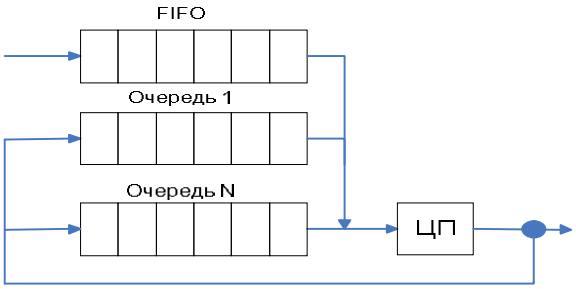


- с переключением, без приоритетов

3) Shortest Job First – со статическими приоритетами, без переключения. Чревато бесконечным откладыванием процессов, требующих много времени на выполнение.
4) Shortest Remaining Time – с вытеснением. Выполняющийся процесс прерывается, если в очереди появляется процесс с меньшим временем выполнения. Необходимо следить за текущим временем обслуживания.

5) Highest Response Ratio (наиболее высокое относительное время ответа) – С динамическими приоритетами. Используется в UNIX. $priority = \frac{t_W - t_S}{t_S}, t_W$ – время ожидания (судя по всему), t_S - требуемое для выполнения время (опять же, вроде того). Чем больше ожидает, тем больше приоритет.

Adaptive planning (алгоритм с многоуровневыми очередями)



Вновь созданные процессы и процессы после завершения IO попадают в очередь с наивысшим приоритетом (FIFO). Квант времени для этой очереди выбирается из расчёта, чтобы успело максимальное количество процессов Процессы, выдать запрос IO или завершиться. переходят в не сделавшие ни того, ни другого, алгоритму очередь с более низким приоритетом по там же RR. В очереди N «крутится» idle-процесс и процессы. оказываются все вычислительные

Адаптивно – рефлективное планирование:
- планирование с учетом требований к памяти.
- предполагает контроль над реальным использованием памяти
- каждому процессу устанавливается ограничение на использование памяти (максимальное количество страниц, которое данная программа может держать в памяти) и виртуальный квант процессорного времени.

Ограничения на память определяются оценкой текущего объема памяти, необходимой процессору и оценкой направления изменения этого объема, которая получается при анализе. Процессу выделяется очередной квант только при наличии достаточного количества свободных страниц или сегментов памяти. Виртуальный квант времени определяется объемом требуемой памяти. Величина кванта обратно пропорциональна максимальному объему памяти, затребованному процессом.
Процессам с большим рабочим множеством отводится меньший квант времени.
Алгоритм ориентирован на процессы с маленькими рабочими множествами.

2. Синхронизация потоков в ОС Windows: мьютексы, события; пояснить особенности использования на примере задачи «читатели и писатели».

В Windows реализован механизм мьютексов (mutex – mutual exception – взаимоисключение). Может использоваться параллельными процессами.
Также в Windows имеется системный вызов CRITICAL_SECTION (в Рихтере: CRITICAL_SECTION – структура данных, а вот EnterCriticalSection() и LeaveCriticalSection() и есть уже системные вызовы). Эти системные вызовы предназначены только для потоков.
Общее для ОС механизмы lock() и unlock(), захвата и освобождения. Они определяются над переменными типа «событие» (в некоторых ОС такая «событийность» перемнных подчёркивается особым типом данных).
Семафоры могут содержать lock unlock.
Пр:

```
lock w:     do while (w = 1) end;   // ожидание освобождения –
                                    // оно неактивное, т.к. при
                                    // w = 1 процесс блокируется
      w := 1;      // Захват
unlock w:   w := 0;      // Освобождение
```

Вместо lock используется команда wait, показывающая что идёт ожидание на объекте (мьютексе или критической секции).
В Windows существует 2 системных вызова: WaitForSingleObject() – ожидание освобождения одного объекта, WaitForMultipleObject() – ожидание освобождения нескольких объекта.

Также используется команда post (отмечать, регистрировать) ожидание на переменной типа событие.
Но все эти методы не применимы для многопроцессорных систем.
Языки параллельного программирования для многопроцессорных систем – Ada.

Событие – сообщение, передаваемое между процессами для синхронизации их работы (в том числе и по использованию разделяемой памяти).

```
void CVal::startWrite(int number)
{
		InterlockedIncrement(&waitingWritersCount);

		if(ReadersCount > 0 || WaitForSingleObject(writeLockMutex, 0) == WAIT_TIMEOUT)
			WaitForSingleObject(canWrite, INFINITE);

		ReleaseMutex(writeLockMutex);
		WaitForSingleObject(writeLockMutex, INFINITE);
		ResetEvent(canWrite);
		ResetEvent(canRead);
		InterlockedDecrement(&waitingWritersCount);
}

void CVal::stopWrite(int number)
{
		if(waitingReadersCount > 0)
			SetEvent(canRead);
		else
			SetEvent(canWrite);

		ReleaseMutex(writeLockMutex);
}
```

1. Управление памятью: выделение памяти разделами фиксированного размера, выделение памяти разделами переменного размера, стратегии выделения памяти, фрагментация памяти.

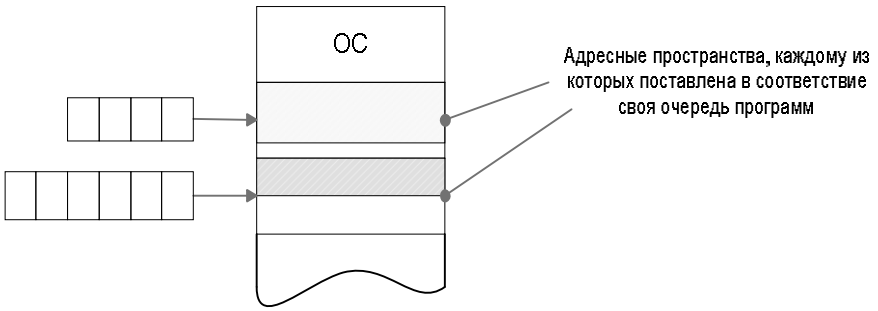
В современных системах имеется иерархия памяти. Чем ближе к процессору, тем быстрее должна быть память.

Существует вертикальное управление памятью, связанное с передачей данных с уровня на уровень. Существует горизонтальное управление, связанное с управление конкретным уровнем.

Распределение памяти разделами фиксированного размера.

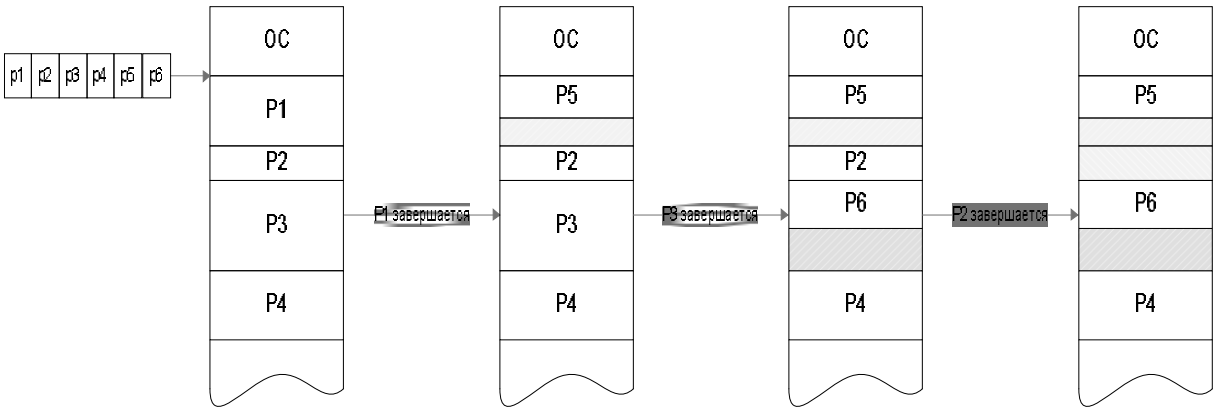
Существует 3 типа разделов небольшой, средний и большой. Размер определяется до работы системы и не меняется.

Минусы: свободные, не используемые участки памяти; в одной очереди может быть заданий много, а в другой не быть совсем. Выход: 1 очередь – но тогда возникает не эффективное использование памяти.



В процессе работы системы размеры блоков были известны и не менялись, что позволяло использовать абсолютные адреса. Если адресное пространство процесса не помещалось в блок фиксированного размера, то такой процесс откладывался. Для каждого задания ОС было известно априорно, сколько времени оно будет выполняться.

Распределение памяти разделами переменной длины



В результате получаем фрагментацию – ситуацию, когда в результате многократной загрузки-выгрузки появляется большое количества небольших свободных участков памяти, в которые ничего нельзя записать.

ОС должна обладать средствами, которые позволяли бы ей объединять свободные участки ОП.

Существует три стратегии выбора раздела для загрузки задания:

1) Выбирается первый попавшийся, подходящий по размеру

2) Выбор самого «тесного» (больше всего соответствующего по размеру, оставляющего меньше свободного места)

3) Выбор самого «широкого» (В оставшееся адресное пространство можно загрузить ещё одно задание)

2. Прерывание реального режима Int 8h, функции. Задачи прерывания по таймеру в защещенном режиме.

3 основные функции таймера в реальном режиме:

1. инкремент счётчика времени

2. вызов обработчика прерывания int 1Ch (пользовательское прерывание, а int 8h аппаратное)

3. декремент счётчика времени до отключения моторчика дисковода и посылка команды остановка на него. Таким образом реализуется отложенное отключение моторчика дисковода, по завершении операции вв/выв в счётчик времени заносится время равное ~2 сек., каждый тик значение декрементируется . Когда станет =0 посылается сигнал на выключение.

В операционной системе Windows

Каждый тик:
- подсчитывает тики аппаратного таймера;
- декрементирует счетчики отложенных вызовов;

По истечению кванта:
- вызывает функции, относящиеся к работе диспетчера ядра, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- по превышении выделенной квоты использования процессора посылает текущему процессу сигнал;
- обновляет статистику использования процессора текущим процессом;

Каждый главный тик
- поддерживает профилирование выполнения процессов в режимах ядра и задачи;

Прерывание в Unix - системах

Каждый тик:
- ведет счет тиков таймера;
- декрементирует счетчики отложенных вызовов;

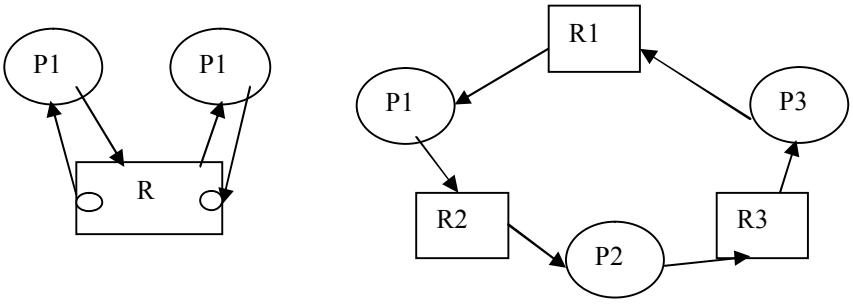
По истечению кванта
- вызывает процедуру обновления статистики использования процессора текущим процессом;
- вызывает функции, относящиеся к работе планировщика, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- посылает текущему процессу сигнал SI6XCPU, если тот превысил выделенную ему квоту использования процессора;

Каждый главный тик:
- пробуждает в нужные моменты системные процессы, такие как swapper и pagedaemon;
- поддерживает профилирование выполнения процессов в режимах ядра и задачи при помощи драйвера параметров;

1. Тупики: обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа и методы восстановление работоспособности системы.

1)Бесконечное откладывание может возникнуть, если процесс не может в необходимое время получить нужный ему ресурс
2)Тупики. Взаимная блокировка
3) Захват и освобождение одних и тех же ресурсов

В системе могут быть только эти три ситуации.
Тупиковая ситуация – тупик – система, возникающая в результате монопольного использования разделенных ресурсов, когда занятый процесс, запрашивает ресурс, занятый другим процессом или запросом через цепочку других процессов , который при этом ожидает ресурс, занятый первым процессом.



«Тупик становится проблемой, когда он возникает».Условия возникновения тупика в системе:
1) Условие взаимоисключения (процессы требуют предоставления права монопольного использования ресурсов)
2) Условие ожидания ресурса (процесс удерживает уже выделенные ресурсы и ожидает выделения дополнительных ресурсов)
3) Условие неперераспределяемости ресурсов (ресурсы нельзя отобрать у процесса, их использующего, до тех пор, пока процесс сам не вернёт их системе)
4) Условие кругового ожидания (существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, которые необходимы следующему в этой цепи процессу)

Обнаружение тупиков и восстановление работоспособности. Формализуем задачу: будем рассматривать систему как декартово произведение множества состояний, где под состоянием понимается состояние ресурса (свободен или распределён). При этом состояние может измениться процессом в результате запроса и последующего получения ресурса, а также в результате освобождения процессом занимаемого им ресурса. Если в системе процесс не может ни получить, ни вернуть ресурс, то говорят, что система находится в **тупике**, то есть не может поменять своё состояние в результате выделения или освобождения ресурса. Определить, что какое-то количество процессов находится в тупике можно при помощи графовой модели Холдта.
Граф $L = (X, U, P)$ задан, если даны множества вершин $X \neq \varnothing$ и множество рёбер $U \neq \varnothing$, а также инцидентор (трехместный предикат) *P*, причём высказвание $P(x, u; y)$ означает высказывание «Ребро *u* соединяет вершину *x* с вершиной *y*», а также удовлетворяет двум условиям:
1) предикат *P* определён на всех таких упорядоченных тройках *(x,u,y)*, для которых $x, y \in X$ и $u \in U$.
2) каждое ребро, соединяющее какую-либо упорядоченную пару вершин *x* и *y* кроме неё может соединять только обратную пару *y, x*.
Дуга – ребро, соединяющее *x* с *y*, но не *y* с *x*.
Дуги бывают двух видов: запросы и выделения. Таким образом модель Холдта представляет собой двудольный (бихроматический) граф, где *X* разбивается на подмножество вершин-процессов $\pi = \{p_1, p_2, ..., p_n\}$ и подмножество вершин-ресурсов $\rho = \{r_1, r_2, ..., r_n\}$. $\pi, \rho : X = \rho \cup \pi, \rho \cap \pi = \varnothing$.
Приобретение (выделение) – дуга (r, p), где $r \in \rho, p \in \pi$.
Запрос – дуга (p, r), где $r \in \rho, p \in \pi$.

Обнаружить процесс, попавший в тупик, можно **методом редукции (сокращения) графа**. Формализуем процедуру сокращения:
1) Граф сокращается по вершине p_i, если эта p_i не является ни заблокированной, ни изолированной, путём удаления всех рёбер, входящих в p_i и выходящих из неё.

2) Процедура сокращения соответствует действиям процессов по приобретению запрошенных ранее ресурсов и последующего освобождения всех занимаемых процессом ресурсов. В этом случае Pi становится изолированной вершиной.

Методы восстановления:
1) Прекращение работы процессов, которые попали в тупик, до некоторого момента, когда ресурсов станет достаточно для выхода из тупика
2) Отбирание ресурсов у процессов и перевод их в режим ожидания по отобранным ресурсам

2. Три режима работы процессора Intel (486 ,Pentium…), защищенный режим, переключение компьютера в защищенный режим.

1. Реальный режим (или режим реальных адресов) - это название было дано прежнему способу адресации памяти после появления 286-го процессора, поддерживающего защищённый режим.

2. Защищенный режим - Режим защиты памяти. Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Основная мысль сводится к формированию таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.) 32-разрадный, многпоточный, многопроцессный, 4 уровня привелегий, доступно 4 Гб виртуальной памяти(для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. 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.)

3. Виртуальный режим - В процессоре i386 компания Intel учла необходимость лучшей поддержки реального режима, потому что программное обеспечение времени его появления не было готово полностью работать в защищенном режиме. Поэтому, например, в i386, возможно переключение из защищенного режима обратно в реальный (при разработке 80286 считалось, что это не потребуется, поэтому на компьютерах с процессором 80286 возврат в реальный режим осуществляется схемно - через сброс процессора).
В качестве дополнительной поддержки реального режима, i386 позволяет задаче (или нескольким задачам) защищенного работать в виртуальном режиме — режиме эмуляции режима реального адреса (таким образом в переключении в реальный режим уже нет необходимости). Виртуальный режим предназначается для одновременного выполнения программы реального режима (например, программы DOS) под операционной системой защищенного режима.
Выполнение в виртуальном режиме практически идентично реальному, за несколькими исключениями, обусловленными тем, что виртуальная задача выполняется в защищенном режиме:

- виртуальная задача не может выполнять привилегированные команды, потому что имеет наинизший уровень привилегий
- все прерывания и исключения обрабатываются операционной системой защищенного режима (которая, впрочем, может инициировать обработчик прерывания виртуальной задачи)

Список действий для перехода в защищенный режим.
1) Необходимо проверить установлен ли бит 0 регистра CR0 в единицу. Если это так, то мы уже находимся в защищенном режиме
2) Сформировать таблицы GDT(LDT) и IDT
3) Запретить маскируемы и немаскируемые прерывания
4) Открыть линию A20 (для обеспечения адресного заворачивания)
5) В сегментные регистры записать данные селекторов
6) Занести базовые адреса сегментов в таблицу GDT
7) С помощью привилегированных команд lgdt и lidt базовые адреса соответствующих таблиц в регистры GRDR и IDTR
8) Устанавливаем бит 0 регистра CR0 в единицу.
9) Jmp far protected_entry (прыгаем на участок, который выполняется в защищенном режиме).

Список действие для перехода обратно:
1) Заполнение теневых регистров значениями FFh для включения адресации реального режима
2) Заполняем сегментные регистры селекторами
3) Возвращаем предыдущие значения регистров GDTR и IDTR
4) Разрешить маскируемы и немаскируемые прерывания

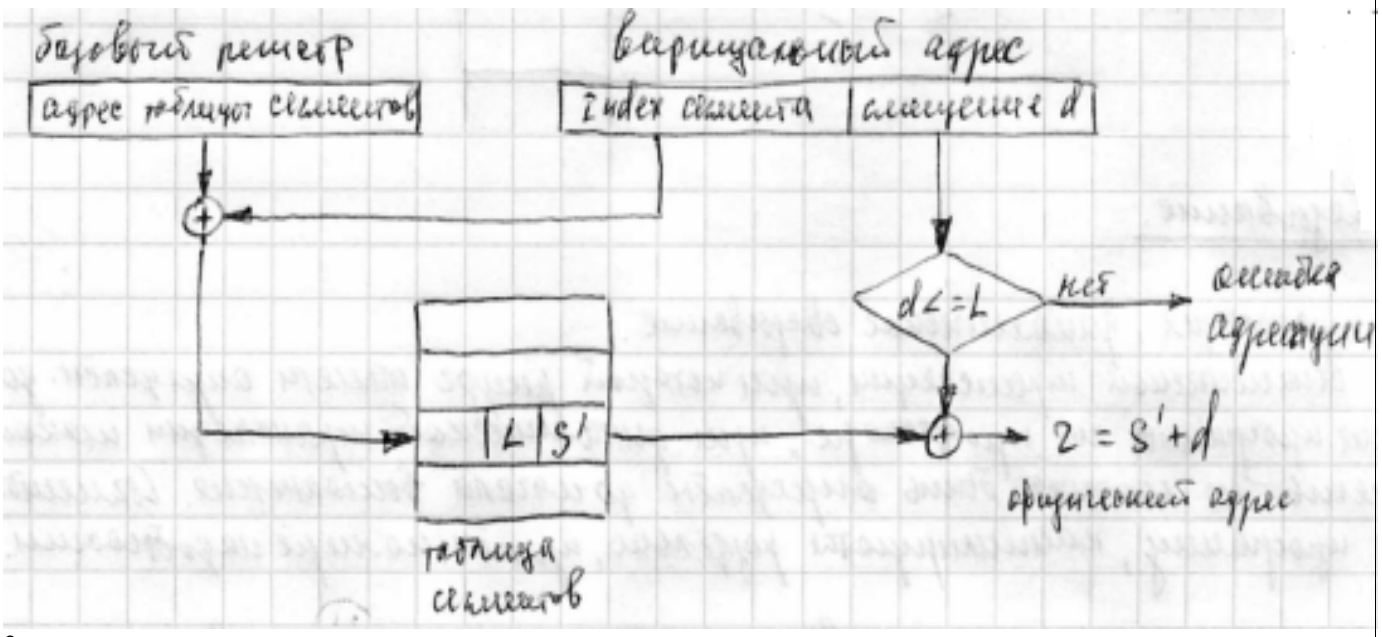
1. Методы управления виртуальной памятью: особенности, достоинства и недостатки.

Виртуальная память – память размер которой превосходит размер физического адресного пространства. Используется адресное пространство диска как область свопинга или педжинга, т.е. для временного хранения областей памяти.

3 способа организации :
1- Страничное распределение памяти по запросам
2- Сегментное распределение памяти по запросам
3- Сегментно-страничное распределение памяти по запросам

Страница - является единицей физического деления памяти. Её размер устанавливается системой.
Сегмент – является единицей логического деления памяти. Её размер определяется объемом кода.

Управление памятью сегментами по запросам



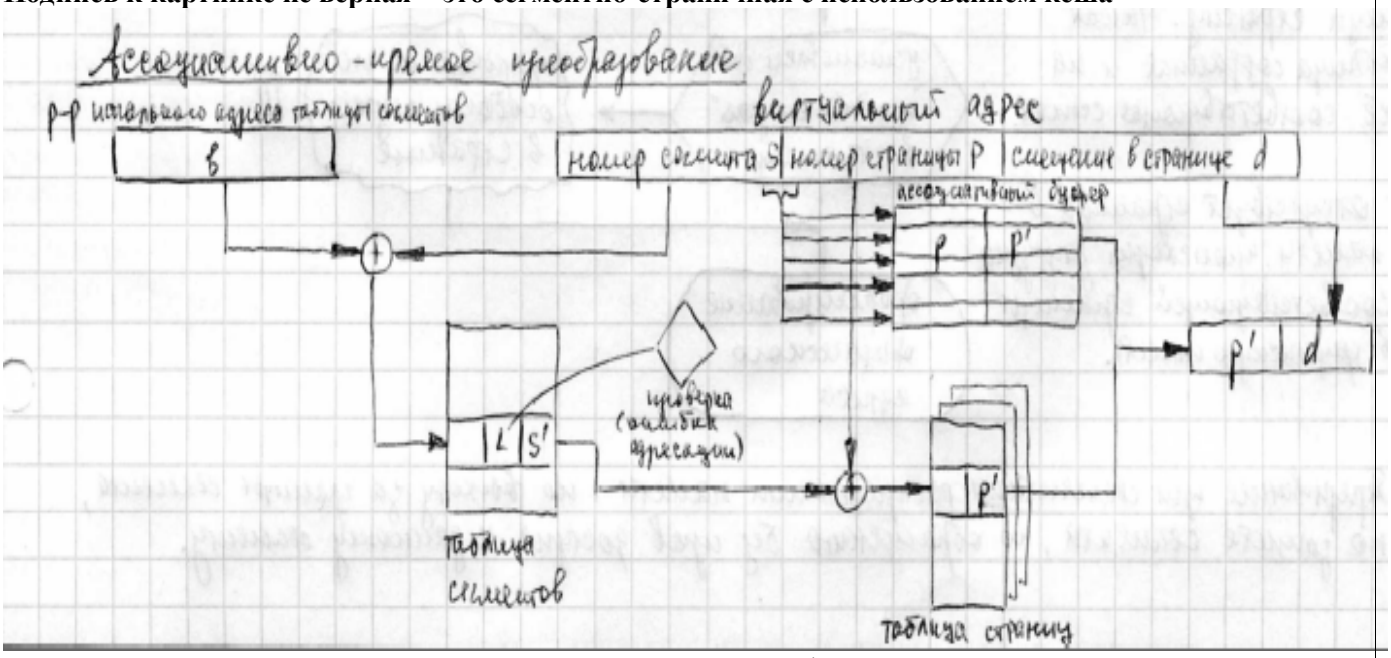
3 типа организации:
1) Локальные
2) Глобальные
3) Локальные + Глобальные

Управление памятью страницами по запросу

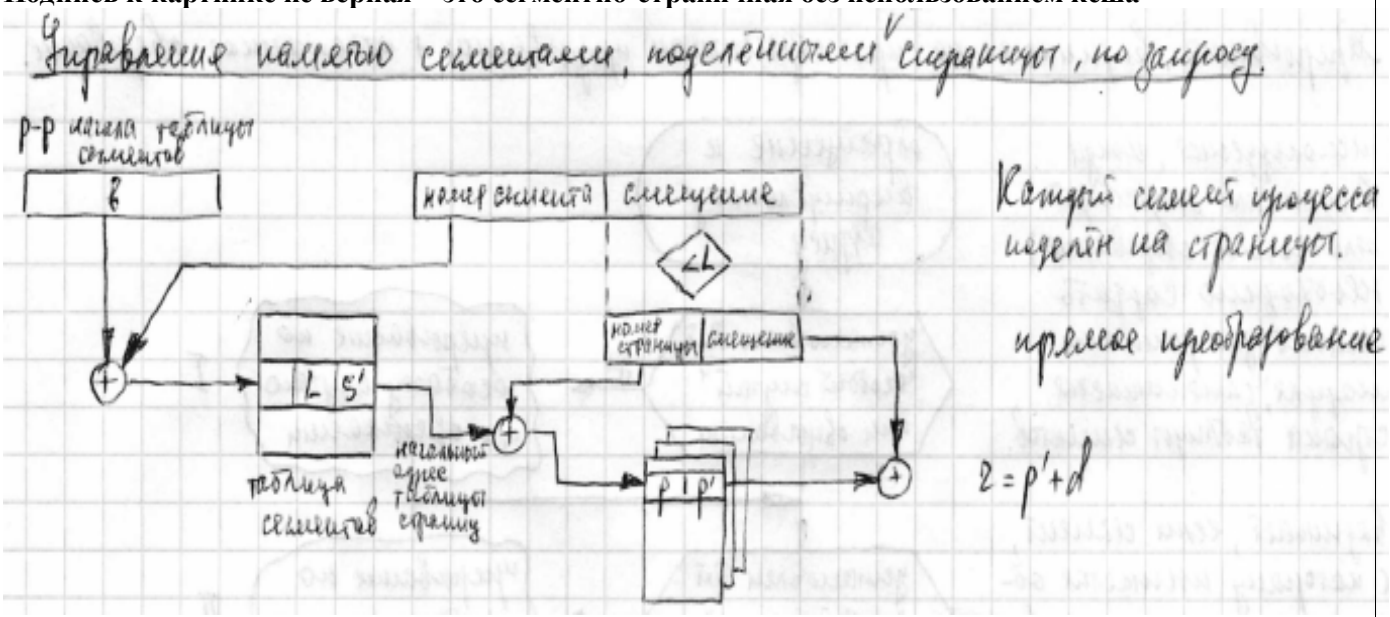
Подпись к картинке не верная – это одноуровневая страничная организация

Сегментно-страничная организация

Подпись к картинке не верная – это сегментно-страничная с использованием кеша



Подпись к картинке не верная – это сегментно-страничная без использованием кеша



Плюсы страничного распределения памяти по запросам:
1) Такое распределение легко реализовать
2) Алгоритм LRU в этом достаточно эффективен

Минусы страничного распределения памяти по запросам:
1) Коллективное использование страниц – под вопросом. На уровне страниц теряется их принадлежность.(Связано с преобразованием адр.) Существуют страницы, которые надо использовать одновременно нескольким процессам, которые в свою очередь могут иметь разные права доступа. Где хранить информацию о том, какие процессы обращаются к странице?

Плюсы сегментного распределения памяти по запросам:
1) Легко реализовать коллективное использование, так как сегмент является логической единицей деления памяти

Минусы сегментного распределения памяти по запросам:
1) Необходимость корректировки таблицы дескрипторов всех процессов при изменении размеров сегментов
2) Сложности при загрузке новых сегментов (в памяти должно сущ. адр пространство необходимого размера). При необходимости система может перенести сегменты путём изменения базового адреса сегмента в дескрипторе (можно возложить на систему, но приводит к большим затратам).
3) Фрагментация (Интенсивная загрузка, и выгрузка может привести к маленьким участкам, в которые загрузить ничего не удастся), хотя система может устранить её путём переноса сегментов (см. выше).

2. Прерывание реального режима Int 8h: функции. Задачи прерывания по таймеру в защищенном режиме.

3 основные функции таймера в реальном режиме:
a. инкремент счётчика времени
b. вызов обработчика прерывания int 1Ch (пользовательское прерывание, а int 8h аппаратное)
c. декремент счётчика времени до отключения моторчика дисковода и посылка команды остановка на него. Таким образом реализуется отложенное отключение моторчика дисковода, по завершении операции вв/выв в счётчик времени заносится время равное ~2 сек., каждый тик значение декрементируется . Когда станет =0 посылается сигнал на выключение.

В операционной системе Windows

Каждый тик:
- подсчитывает тики аппаратного таймера;
- декрементирует счетчики отложенных вызовов;

По истечению кванта:
- вызывает функции, относящиеся к работе диспетчера ядра, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;
- по превышении выделенной квоты использования процессора посылает текущему процессу сигнал;
- обновляет статистику использования процессора текущим процессом;

Каждый главный тик
- поддерживает профилирование выполнения процессов в режимах ядра и задачи;

Прерывание в Unix - системах

Каждый тик:
- ведет счет тиков таймера;
- декрементирует счетчики отложенных вызовов;

По истечению кванта
- вызывает процедуру обновления статистики использования процессора текущим процессом;
- вызывает функции, относящиеся к работе планировщика, такие как пересчет приоритетов и действия, выполняющиеся по истечении выделенного кванта времени;

- посылает текущему процессу сигнал SI6XCPU, если тот превысил выделенную ему квоту использования процессора;

Каждый главный тик:

- пробуждает в нужные моменты системные процессы, такие как swapper и pagedaemon;
- поддерживает профилирование выполнения процессов в режимах ядра и задачи при помощи драйвера параметров;

1 Ядро ОС: многопоточное ядро; взаимоисключение в ядре – спин - блокировки.

Синхронизация – процесс, в результате которого один процесс ждет когда другой процесс придет в этуу же точку. Критические секции ядра – разделы, в которых модифицируются глобальные страницы данных или очереди.

В одно/многопроцессных системах система использования критического ресурса решается путем взаимоисключения, а механизм называется SpinLock.

Простейшая функция механизмов взаимоисключений базируется на аппаратных реалиях:

- Interlocked Increment
- Interlocked Decrement
- Interlocked Exchange

При реализации этих функций шина блокируется на время выполнения, чтобы процесс не мог выполнить эту команду.

Основная проблема в ядре связана с прерываниями. Нехорошо запрещать прерывания, эту проблему пытаются решить. Эту проблему и решают спин-блокировки.

Спин-блокировка - простейший механизм синхронизации. Спин-блокировка может быть **захвачена**, и **освобождена**. Если спин-блокировка была захвачена, последующая попытка захватить спин-блокировку любым потоком приведет к бесконечному циклу с попыткой захвата спин-блокировки (состояние потока *busy-waiting*). Цикл закончится только тогда, когда прежний владелец спин-блокировки освободит ее. Использование спин-блокировок безопасно на мультипроцессорных платформах, то есть гарантируется, что, даже если ее запрашивают одновременно два потока на двух процессорах, захв атит ее только один из потоков.

Просто для понимания!!! Не совсем по НЮ, но она ведь тоже у кого-то училась:

Спин-блокировки предназначены для защиты данных, доступ к которым производится на различных, в том числе повышенных уровнях IRQL. Теперь представим такую ситуацию: код, работающий на уровне IRQL PASSIVE_ LEVEL захватил спин-блокировку для последующего безопасного изменения некоторых данных. После этого код был прерван кодом с более высоким уровнем IRQL DISPATCH_LEVEL, который попытался захватить ту же спин-блокировку, и, как следует из описания спин-блокировки, вошел в бесконечный цикл ожидания освобождения блокировки. Этот цикл никогда не закончится, так как код, который захватил спин-блокировку и должен ее освободить, имеет более низкий уровень IRQL и никогда не получит шанса выполниться!

Чтобы такая ситуация не возникла, необходим механизм, не позволяющий коду с некоторым уровнем IRQL прерывать код с более низким уровнем IRQL в тот момент когда код с более низким уровнем IRQL владеет спин-блокировкой. Таким механизмом является повышение текущего уровня IRQL в момент захвата спин-блокировки до некоторого уровня IRQL, ассоциированного со спин-блокировкой, и восстановление старого уровня IRQL в момент ее освобождения. Из сказанного следует, что код, работающий на повышенном уровне IRQL, не имеет права обращаться к ресурсу, защищенному спин-блокировкой, если уровень IRQL спин-блокировки ниже уровня IRQL производящего доступ к ресурсу кода. При попытке таким кодом захватить спин-блокировку его уровень IRQL будет понижен до уровня IRQL спин-блокировки, что приведет к непредсказуемым последствиям.

Использование обычных спин-блокировок

1. VOID KeInitializeSpinLock(IN PKSPIN_LOCK SpinLock); Эта функция инициализирует объект ядра KSPIN_LOCK. Память под спин-блокировку уже должна быть выделена в невыгружаемой памяти.
2. VOID KeAcquireSpinLock(IN PKSPIN_LOGK SpinLock, OUT PKIRQL Oldlrql); Эта функция захватывает спин-блокировку. Функция не вернет управление до успеха захвата блокировки. При завершении функции

уровень IRQL повышается до уровня DISPATCH_LEVEL. Во втором параметре возвращается уровень IRQL, который был до захвата блокировки (он должен быть <= DISPATCH_LEVEL).

3. VOID KeReleaseSpinLock(IN PKSPINJLOCK SpinLock, OUT PKIRQL Newlrql); Эта функция освобождает спин-блокировку и устанавливает уровень IRQL в значение параметра Newlrql. Это должно быть то значение, которое вернула функция KeAcquireSpinLock() в параметре Oldlrql.
4. VOID KeAcquireLockAtDpcLevel(IN PKSPIN_LOCK SpinLock); Эта оптимизированная функция захватывает спин-блокировку кодом, уже работающем на уровне IRQL DISPATCH_LEVEL. В этом случае изменение уровня IRQL не требуется. На однопроцессорной платформе эта функция вообще ничего не делает, так как синхронизация обеспечивается самой архитектурой IRQL.
5. VOID KeReleaseLockFromDpcLevel(IN PKSPIN_LOCK SpinLock); Эта функция освобождает спин-блокировку кодом, захватившим блокировку с помощью функции KeAcquireLockAtDpcLevel(). На однопроцессорной платформе эта функция ничего не делает.

2 Защищенный режим: перевод компьютера в защищенный режим – последовательность действий.

Защищенный режим – 32х битный, многопоточный, многопроцессный режим работы работы процессора, с использованием виртуальной памяти, 4гб адресного пространства, 2 способа организации памяти: страницы по запросу, сегменто-страничная память по запросу. 4 уровня привилегий.

Список действий для перехода в защищенный режим.

1) Необходимо проверить установлен ли бит 0 регистра CR0 в единицу. Если это так, то мы уже находимся в защищенном режиме
2) Сформировать таблицы GDT(LDT) и IDT
3) Запретить маскируемы и немаскируемые прерывания
4) Открыть линию A20 (для обеспечения адресного заворачивания)
5) В сегментные регистры записать данные селекторов
6) Занести базовые адреса сегментов в таблицу GDT
7) С помощью привилегированных команд lgdt и lidt базовые адреса соответствующих таблиц в регистры GRDR и IDTR
8) Устанавливаем бит 0 регистра CR0 в единицу.
9) Jmp far protected_entry (прыгаем на участок, который выполняется в защищенном режиме).

Список действие для перехода обратно:

1) Заполнение теневых регистров значениями FFh для включения адресации реального режима
2) Заполняем сегментные регистры селекторами
3) Возвращаем предыдущие значения регистров GDTR и IDTR
4) Разрешить маскируемы и немаскируемые прерывания

1.Прерывания: классификация, приоритеты прерываний , прерывания в последовательности ввода-вывода.

Классификация прерываний

В зависимости от источника, прерывания делятся на

- программные (по Рязановой это сист. вызов)
- аппаратные
- исключения

Системный вызов – вызывается искусственно с помощью соответствующей команды из программы (int), предназначен для выполнения некоторых действий операционной системы (фактически запрос на услуги ОС), является синхронным событием.

Исключения – являются реакцией микропроцессора на нестандартную ситуацию, возникшую внутри микропроцессора во время выполнения некоторой команды программы (деление на ноль, прерывание по флагу TF (трассировка)), являются синхронным событием.

Исключения бывают 3 видов:

1.Нарушения – fault – это исключение, фиксируемое до выполнения команды или в процессе её выполнения.

2.Ловушка – Trap – процессором обрабатывается после команды, вызвавшей это исключение.

3.Авария – abort – данный тип исключения является следствием невосстановимых, неисправимых ошибок, нпример, деление на ноль.

- Исправимые искл. – приводят к вызову определенного менеджера системы, в результате работы которого может быть продолжена работа процесса (пример: страничная неудача с менеджером памяти)
- Неисправимые искл. – в случае сбоя или в случае ошибки программы (пример: ошибка адресации). В этом случае процесс завершается.

Аппаратные - возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени.

Различают прерывания:

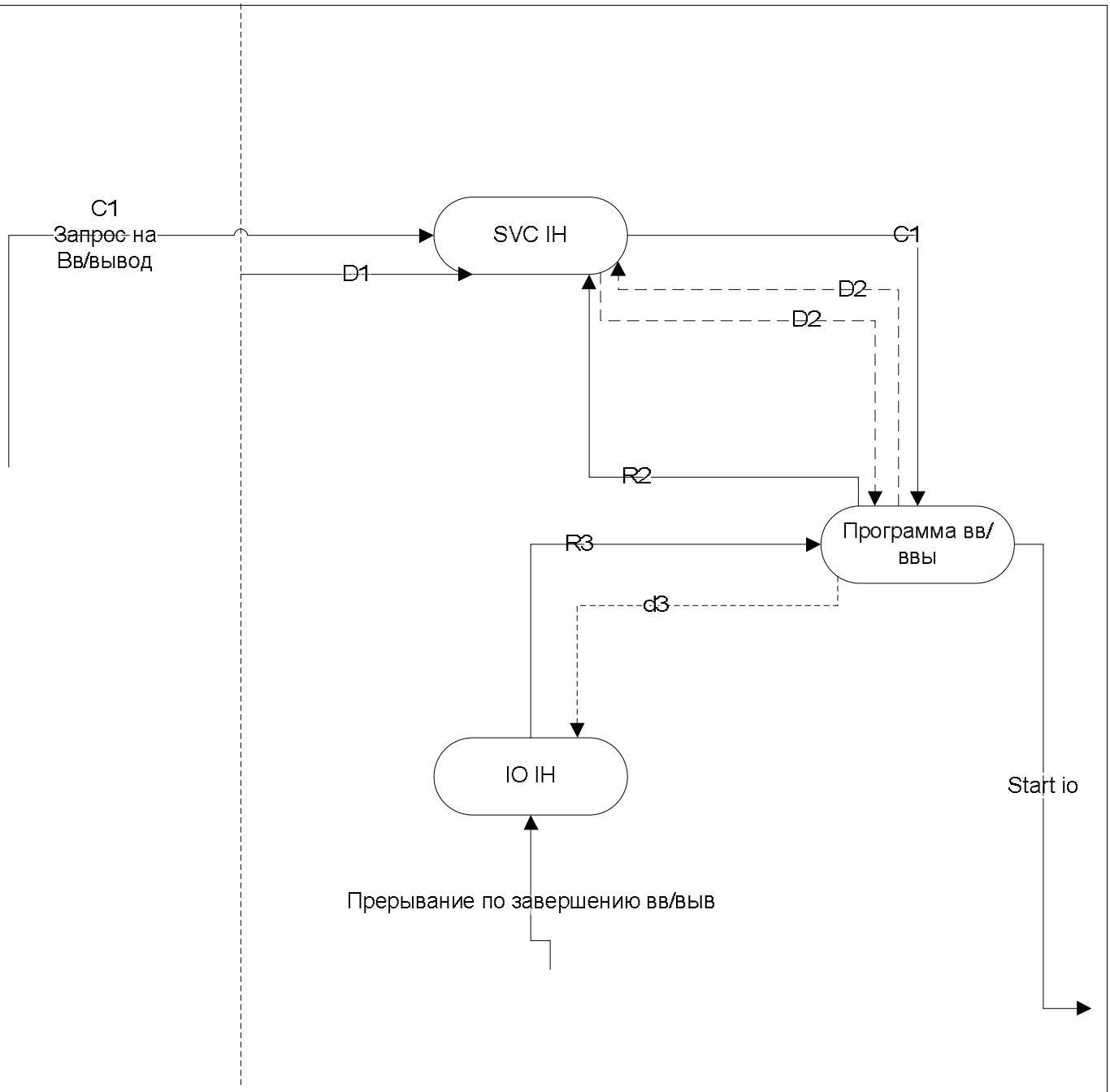
- От таймера
- От действия оператора (пример: ctrl+alt+del)
- От устройств вв/выв

В реальном и защищенном режиме работы микропроцессора обработка прерываний осуществляется принципиально разными методами.

Механизм реализации прерываний ввода/вывода

Пояснения к схеме

R – Response – Ответ
D – Data – передача прием данных
C - Command - Команда
SVC – SuperVisor Call
IH – Interupt handler



Последовательность действий:

1) Пользователь делает запрос на ввод-вывод
2) Обработчик прерывания получает запрос, получает необходимые данные и отправляет их в программу ввода/выода
3) Программа ввода/вывода, при необходимости, обменивается данными с обработчиком, посылает запрос на начало операции. И засыпает до получения ответа о конце ввода/вывода
4) Обработчик прерывания разблокирует драйвер.
5) Дальше идет череда ответов до самого пользователя

2. Защищенный режим: перевод компьютера в защищенный режим – последовательность действий.

Защищенный режим – 32х битный, многопоточный, многопроцессный режим работы работы процессора, с использованием виртуальной памяти, 4гб адресного пространства, 2 способа организации памяти: страницы по запросу, сегменто-страничная память по запросу. 4 уровня привилегий.

Список действий для перехода в защищенный режим.

1) Необходимо проверить установлен ли бит 0 регистра CR0 в единицу. Если это так, то мы уже находимся в защищенном режиме

2) Сформировать таблицы GDT(LDT) и IDT
3) Запретить маскируемы и немаскируемые прерывания
4) Открыть линию A20 (для обеспечения адресного заворачивания)
5) В сегментные регистры записать данные селекторов
6) Занести базовые адреса сегментов в таблицу GDT
7) С помощью привилегированных команд lgdt и lidt базовые адреса соответствующих таблиц в регистры GRDR и IDTR
8) Устанавливаем бит 0 регистра CR0 в единицу.
9) Jmp far protected_entry (прыгаем на участок, который выполняется в защищенном режиме).

Список действие для перехода обратно:

1) Заполнение теневых регистров значениями FFh для включения адресации реального режима
2) Заполняем сегментные регистры селекторами
3) Возвращаем предыдущие значения регистров GDTR и IDTR
4) Разрешить маскируемы и немаскируемые прерывания