1

1. ОС – определение, место ОС в системе программного обеспечения ЭВМ. Ресурсы вычислительной системы. Режимы ядра и задачи: переключение в режим ядра – классификация событий. Процесс, как единица декомпозиции системы.

ОС – комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, которые используют эти ресурсы в вычислениях.

Классификация ОС:

1. Однопрограммная пакетной обработки – в оперативной памяти м.б. только 1 прикладная программа.
2. Мультипрограммная пакетной обработки – в оперативной памяти одновременно много программ. Загрузка – перфокартами.
3. Мультипрограммная с разделением времени – в оперативной памяти одновременно большое число программ, процессорное время – квантуется (чтобы обеспечить фиксированное время ответа <= 3 с).
4. Реального времени – главное – время отклика. Организует работу вычислительной системы в темпе, обеспечивающем обслуживание некоторого внешнего процесса не зависимо от вычислительной системы. Время отклика системы <= время поступления запроса на ответ. В ситемах реального времени используется система абсолютных приоритетов.
   a. Жесткая система (строгое обеспечение времени ответа).
   b. Гибкая система (цифровые аудио и мультимедийные средства).
5. Сетевые (серверные).
6. Многопроцессорные.
7. Встроенные (в ТВ, микроволновки) – Windows CE.
8. ОС для смарт-карт.

ОС ПК – многопроцессорная система с разделением времени.

Ресурс – любой из компонентов вычислительной системы и предоставляемые им возможности.
Ресурсы: время CPU, объем ОЗУ, каналы I/O, timer, данные, ключи защиты, реентерабельные коды ОС (коды, не модифицирующие сами себя (доступна повторная входимость).
Процесс – единица декомпозиции системы, именно ему выделяются ресурсы системы.
Процесс – программа в стадии выполнения.

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

2. Три режима работы компьютера на базе процессоров Intel.

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

Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес – сегмент/смещение). Минимальная адресная единица памяти – байт.

2^{20} = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).

Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

1-проц. режим под управлением MS-DOS (главное – минимизация памяти, занимаемой ОС => нет многозадачности).

0-256 б.: таблица векторов прерываний
резидентная часть DOS
место резидентных программ
сегменты программы
pool или heap (куча)
транзитная часть DOS
Резидентная часть DOS 1 Мб

2. Защищенный режим – многопроцессный режим. В памяти компьютера одновременно находится большое число программ с квантованием процессорного времени с виртуальной памятью. Управляет защищенным режимом ОС с разделением времени (Windows, Linux). В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.

Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Формирование таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

3. Специальный режим защищенного режима (V86) – как задачи выполняются ОС реального режима, в кажд. из кот. выполн. по 1 прогр. реального режима. Многозадачный режим с поддержкой виртуальной памяти.

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.

VMM (Virtual Machine Manager) – для запуска виртуальных машин реального режима.

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)

2

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

ОС – комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, которые используют эти ресурсы в вычислениях.

Классификация ОС:

1. Однопрограммная пакетной обработки – в оперативной памяти м.б. только 1 прикладная программа.
2. Мультипрограммная пакетной обработки – в оперативной памяти одновременно много программ. Загрузка – перфокартами.
3. Мультипрограммная с разделением времени – в оперативной памяти одновременно большое число программ, процессорное время – квантуется (чтобы обеспечить фиксированное время ответа <= 3 с).
4. Реального времени – главное – время отклика. Организует работу вычислительной системы в темпе, обеспечивающем обслуживание некоторого внешнего процесса не зависимо от вычислительной системы. Время отклика системы <= время поступления запроса на ответ. В системах реального времени используется система абсолютных приоритетов.
   c. Жесткая система (строгое обеспечение времени ответа).
   d. Гибкая система (цифровые аудио и мультимедийные средства).
5. Сетевые (серверные).
6. Многопроцессорные.
7. Встроенные (в ТВ, микроволновки) – Windows CE.
8. ОС для смарт-карт.

ОС ПК – многопроцессорная система с разделением времени.

Иерархическая машина – ОС разбивается на функции и определяется место этих функций по удаленности от аппаратной части.

Виртуальная машина (?) – совокупность команд машины и команд ОС, кот. м. использовать программы для получения сервиса ОС.

2. XMS, линия A20 – адресное заворачивание.

XMS (eXtended Memory Specification, спецификация расширенной памяти) – спецификация Microsoft на расширенную память (XMS 2.0), позволяющая DOS-программам с помощью диспетчера расширенной памяти (XMM) использовать расширенную память ПК на процессорах 80286 и более новых. *(добавленная, продленная)*
XMS оговаривает все вопросы, связанные с дополнительной памятью (сверх 1 Мб).
EMS (Expanded Memory Specification, спецификация отображаемой памяти) – стандарт, разработанный в 1985 г. фирмами Lotus, Intel и Microsoft для доступа из DOS к областям памяти выше 1 Мбайт в системах на базе процессоров 80386 и более поздних. *(расширенная, растягиваемая)*

- 0-640 Кб: основная память (conventional) – память, доступная DOS и программам реального режима.

Стандартная память является самой дефицитной в PC, когда речь идет о работе в среде ОС 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 используется для векторов прерываний, области переменных BIOS; области DOS; предоставляется пользователю (до 638 Кбайт).

- 640-1024 Кб: UMA (upper memory area) – обл. верхней памяти – зарезервирована для системных нужд

Размещаются обл. буферной памяти адаптеров (пр. – видеопамять) и постоянная память (BIOS с расширениями).

- С 1024 Кб –…(защ.) XMA (extended memory area) – непосредственно доступна только в защищенном режиме для компьютеров с процессорами 286 и выше.
- 1024-1088 Кб (Реал): HMA (high memory area) – верхняя область памяти (1-й сегмент размером 64 Кбайт, расположенный выше мегабайтной отметки памяти PC с операционной системой MS-DOS) Единственная область расширенной памяти, доступная 286+ в реальном режиме при открытом вентиле Gate A20.

XMS – программная спецификация использования дополнительной памяти DOS-программами для компьютеров на процессорах 286 и выше. 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 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 реальном режиме существует 2 вида адресного заворачивания.
В 8086 – 20 линий адреса, заворачивание на начало. 2^{20} = 1 Мб = FFFFF
Если в реальном режиме открыть линию A20 (21-ю линию), то в реальном режиме станет доступно дополнительно 64 Кб памяти. Линия A20 – для совсместимости (для заворачивания).
По умолчанию компьютер начинает работу в реальном режиме. Для обеспечения адресного заворачивания A20 сброшена – заземлена. Для перехода в защищенный режим надо открыть A20.

В реальном 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.

3

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

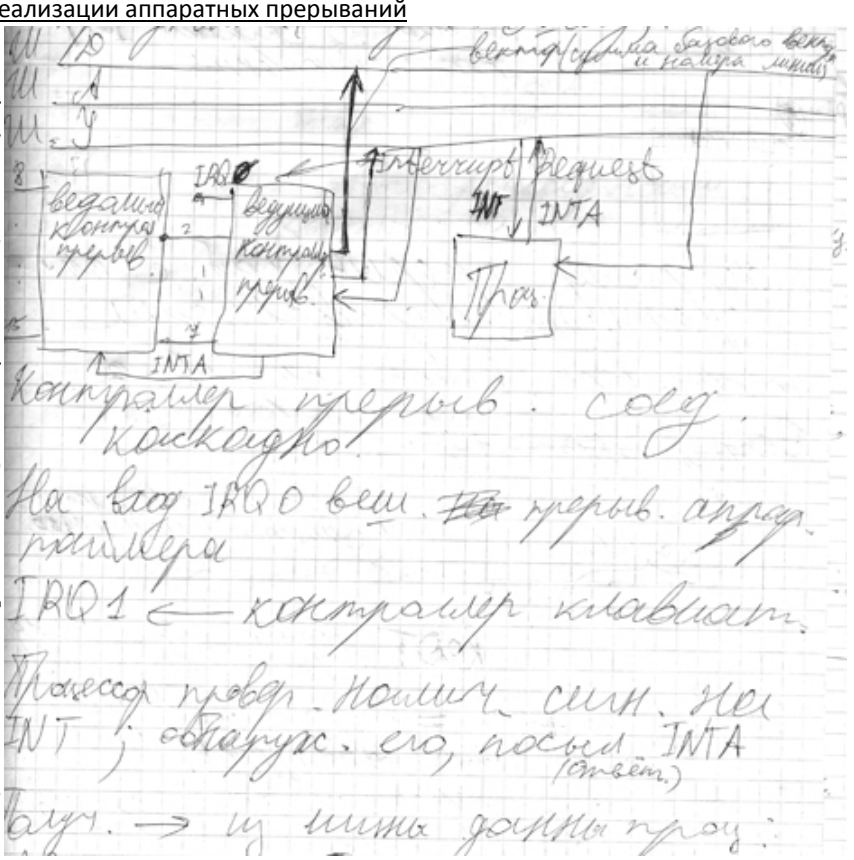
Классификация прерываний (в зависимости от источника)

- программные (системные вызовы) – вызываются искусственно с помощью соответствующей команды из программы (int), предназначены для выполнения некоторых действий ОС (фактически запрос на услуги ОС), является синхронным событием.
- аппаратные – возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, мышь, 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 (посыл. сигнал о завершении процесса вв/выв на контроллер прерываний)
- 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.
  - Исправимые – приводят к вызову определенного менеджера системы, в результате работы которого может быть продолжена работа процесса (пр.: страничная неудача с менеджером памяти)
  - Неисправимые – в случае сбоя или в случае ошибки программы (пр.: ошибка адресации). В этом случае процесс завершается.

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 реальном режиме базовый вектор = 8h, в защищенном – первые 32 строки IDT отведены под исключения => базовый вектор = 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 подтвержд. разреш. контроллеру издавать новые прерывания.

Точные прерывания – прерывание, оставляющее машину в строго определенном состоянии. Обладает св-ми:

1. Счетчик команд указывает на команду (текущая), до которой все команды выполнены.
2. Ни одна команда после текущей не выполнена.

Не говорится, что команды после текущей не могли начать выполнение, а то, что все изменения, связанные с этими командами, должны быть отменены.

3. Состояние текущей команды известно (аппаратн. – обычно не нач. вып., при искл. – привела к исключ.)

Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера или подключенных к нему, поступают в процессор не непосредственно, а через 2 контроллера прерываний, I – ведущий, а II – ведомым. 2 контроллера используются для увелич. допуст. кол-ва внешних устройств (кажд. контроллер м. обслуживать сигналы от 8 устройств). Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них веерообразную структуру. В совр. машинах устанавливают 2 контроллера, увеличивая число входных устройств до 15 (7 у ведущего и 8 у ведомого).

Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Для ведущего контроллера в реальном режиме базовый вектор равен 8, для ведомого - 70h.

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

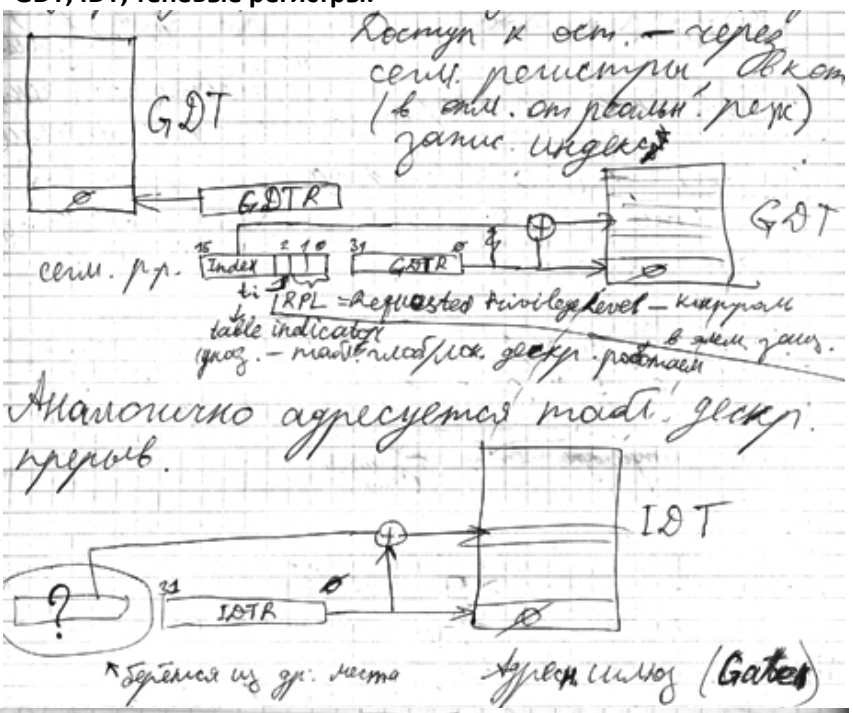
Защищенный режим – 32х-разрядный, поддерж. многопоточность и многопроцессность. В отличие от реального режима здесь доступно 4 Гб памяти (в реальном диапазон адресов памяти ограничен 1 мб). В защищ. реж. 4 уровня привилег. Ядро ОС находится на 0-м уровне.

GDT (global descriptor table) – таблица, которая описывает сегменты основной памяти ОС. В системе только 1 GDT. На начальный адрес GDT указывает GDT Register (32 разрядный).

IDT (interrupt descriptor table) – таблица, предназначенная для хранения адресов обработчиков прерываний. Базовый адрес IDT помещен IDT Register. IDT столько, сколько процессоров.

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.



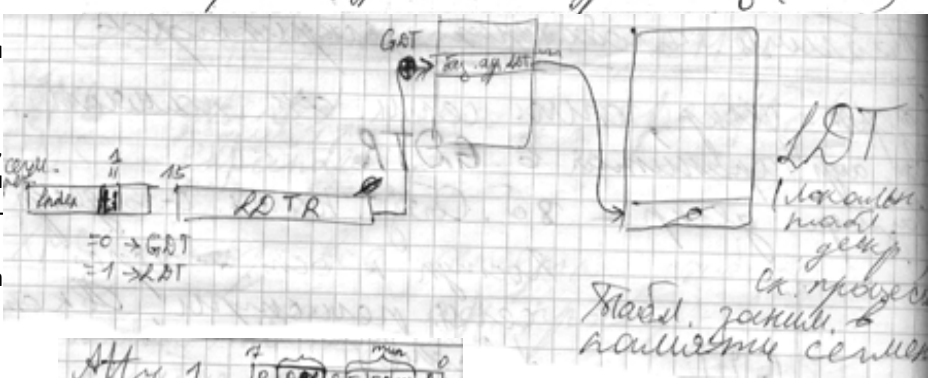
Формат селектора (явл. ID сегмента):

Индекс равен (кратен) 8 и является смещением в таблице дескрипторов.

Обязательно наличие 0-го дескриптора.

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.



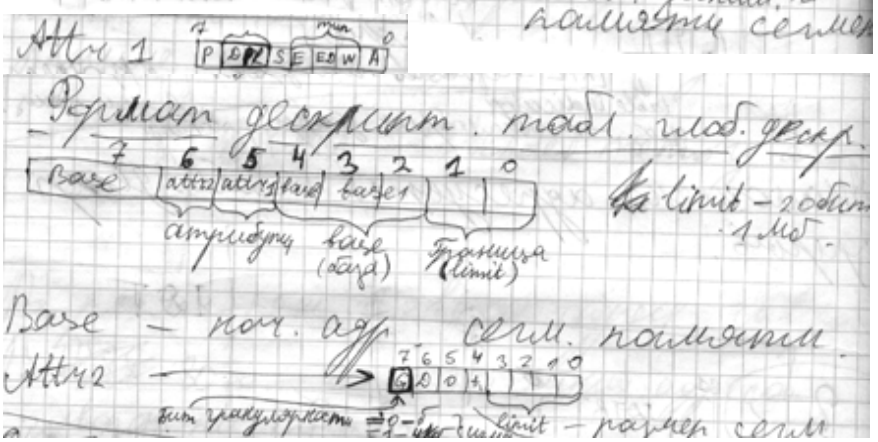
Формат дескриптора сегмента (GDT):

A – access – бит доступа к сегменту (устан. аппаратно при доступе к сегменту).

Тип – опред. поля доступа:

w: для сегмента кода:
- 0: чтение запрещено (не касается выборки команд)
- 1: чтение разрешено

для сегмента данных:
- 0: модиф. запрещ., 1: модиф. разреш.



S – опред., что описывает дескриптор;

DPL – уровень привилегий

P – бит присутствия, исп. для работы с ВП. (0-сегмента нет в ВП, 1 – есть.)

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

G – бит гранулярности (0 – размер сегмента задан в байтах, 1 – в страницах по 4 Кб).

Процесс не м. выйти за размер своего сегмента – контроль ОС (защита адресных простр-в сегментов др. от др.).

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.

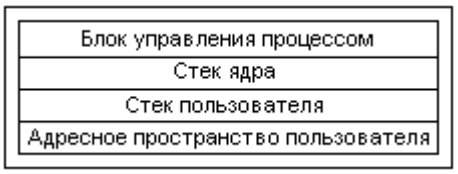
В процессоре каждому из сегментных регистров (CS, DS, SS, ES, FS, GS) сопоставлен теневой регистр (дескриптора). Он не доступен программисту и загружается (параллельно) автоматически из таблицы дескрипторов соотв. сегмента чтобы реже обращаться к ОП.

4

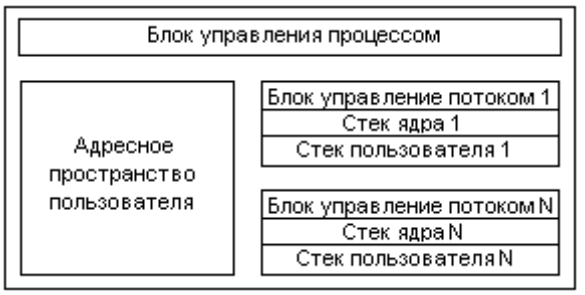
1. Понятие процесса. Процесс как единица декомпозиции системы. Процессы и потоки. Типы потоков. Диаграмма состояний процесса. Планирование и диспетчеризация.

Процесс - программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).
М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.
Поток (thread, light-weight process, нить) - часть последовательного кода процесса, которая м. выполняться параллельно с другими частями кода. Не имеет своего адресного пространства.

1) однопоточная модель процесса



2) многопоточная модель процессора



← Поток

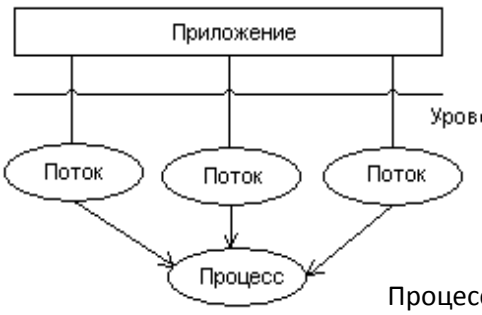
← Поток

Типы потоков

1) потоки на уровне пользователя (прикладные потоки)

\+ быстрота переключения
Каждая нить имеет свой контекст
Возм. распараллел. при асинхр. системн. вызовах
Управл. потоками заним. спец. библиотека.

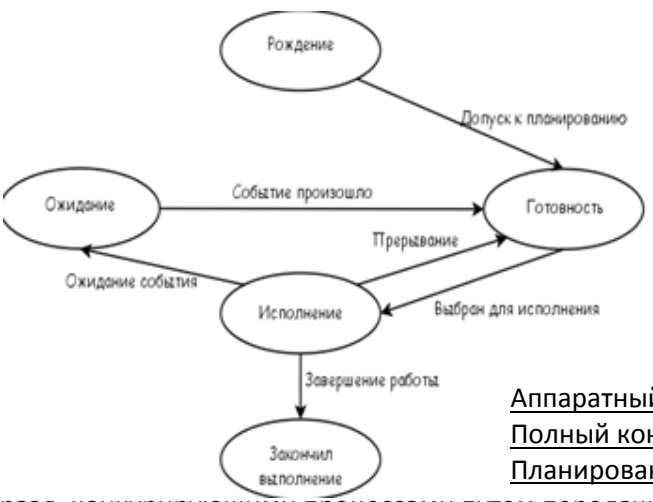
2) потоки на уровне ядра



Ядро знает о нитях все.
Заним. переключ.нитей

Независ. поток управл., выполн. в контексте некоторого процесса.
Процесс остается владельцем ресурсов, потоку управл. передается регистровый (аппар.) контекст и стеки (пользователя, ядра).

Диаграмма состояний процесса



Порождение – присв. процессу строки в таблице процессов
Готовность – попадание в очередь готовых процессов – получили все необходимые ресурсы.
Выполнение
Блокировка (Ожидание) – ожидание необходимого ресурса. Если процесс интерактивный, то он постоянно блокируется в ожидании ввода/вывода.
(схема для мультипрограммной пакетной обработки)
Прерывание = истек квант времени.

Аппаратный контекст процесса – сост. регистров.
Полный контекст процесса – сост. регистров + сост. памяти.
Планирование – управление распределением ресурсов ЦП между разл. конкурирующими процессами путем передачи им управления согласно некот. стратегии планирования.
Диспетчеризация – выделение процессу процессорного времени.

Классификация планирования (по отн. к проц. t):
Системы бывают:

- с/без переключения
- с/без вытеснения
- с/без приоритетов

Классификация приоритетов:

- относительные/абсолютные
- статические/динамические

Процесс выполняется от начала и до конца при получении процессорного времени (для систем с однопрограммной монопольной обработкой), либо процесс выполняется до тех пор, пока он не запросит IO, после чего он блокируется в ожидании завершения IO (для систем с мультипрограммной обработкой). Т.о., процесс выполняется произв. кол-во времени (зависит от характера процесса) → нет гарантир. времени отклика.
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.

2. Обеспечение моноп. доступа к разделяемым данным в задаче "писатели-читатели", используя Win32 API .

```
#include <stdlib.h>
#include <conio.h>
#include <windows.h>
volatile LONG ActReadersCount = 0, ReadersCount = 0,
        WritersCount = 0;
volatile int Value = 0;
HANDLE CanRead, CanWrite, Writing;

void StartRead()
{
        InterlockedIncrement(&ReadersCount);
        WaitForSingleObject(Writing, INFINITE);
        ReleaseMutex(Writing);
        if (WritersCount)
            WaitForSingleObject(CanRead, INFINITE);
        InterlockedDecrement(&ReadersCount);
        InterlockedIncrement(&ActReadersCount);
        if (ReadersCount)
                SetEvent(CanRead);
}
void StopRead()
{
        InterlockedDecrement(&ActReadersCount);
        if (!ReadersCount)
                SetEvent(CanWrite);
}
void StartWrite()
{
        InterlockedIncrement(&WritersCount);
        if (ActReadersCount)
            WaitForSingleObject(CanWrite, INFINITE);
        WaitForSingleObject(Writing, INFINITE);
        InterlockedDecrement(&WritersCount);
}
void StopWrite()
{
        ReleaseMutex(Writing);
        if (ReadersCount)
                SetEvent(CanRead);
        else
                SetEvent(CanWrite);
}
DWORD WINAPI Reader(PVOID pvParam)
{
        for (;;)
        {
                StartRead();
                Sleep(rand()/100.);
                printf("Reader %d: %d\n", (int)pvParam,
                        Value);
                StopRead();
                Sleep(rand()/10.);
        }
        return 0;
}
DWORD WINAPI Writer(PVOID pvParam)
{
        for (;;)
        {
                StartWrite();
                Sleep(rand()/75.);
                printf("Writer %d: %d\n", (int)pvParam,
                        ++Value);
                StopWrite();
                Sleep(rand()/7.);
        }
        return 0;
}
int main()
{
        HANDLE Writers[3], Readers[3];
       CanRead = CreateEvent(NULL, false, false, NULL);
       CanWrite = CreateEvent(NULL, false, true, NULL);
        Writing = CreateMutex(NULL, false, NULL);
        for (int i = 0; i < 3; i++)
                Writers[i] = CreateThread(NULL, NULL,
                        Writer, (LPVOID)i, NULL, NULL);
        for (int j = 0; j < 3; j++)
                Readers[j] = CreateThread(NULL, NULL,
                        Reader, (LPVOID)j, NULL, NULL);
        getch();
        return 0;
}
```

5

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

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 некорректного разделения. В каждый момент времени на процессоре выполняется 1 процесс. Каждому процессу выделяется квант процессорного времени.

Разделяемый ресурс - переменная, к которой обращаются разные процессы.

Критическая секция – область кода, из которой осуществляется доступ к разделяемому ресурсу.

Монопольное использование – если процесс получил доступ к разделяемому ресурсу, то др. процесс не м. получить доступ к этому ресурсу.

Необходимо обеспечить монопольный доступ процесса к разделяемому ресурсу до тех пор, пока процесс его не освободит. 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.

Возможные варианты развития событий:

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические 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 (ввел Дэйкстра в 1965 г.)

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции:
P (от датск. passeren - пропустить) и V (от датск. vrygeven - освободить). Защищенность семафора означает, что значение семафора м. изменяться только операциями P и V.

1. Операция P(S): S = S – 1. Декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить операцию P, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой процесс, выполняющий операцию V(S).
2. Операция V(S): S = S + 1. Инкремент S одним неделимым действием (последовательность непрерывных действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0, то V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожид. на семафоре.

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 поддерживаются ОС-ой. Семафоры устраняют активное ожидание на процессоре.

Семафоры бывают:

- бинарные (S принимает значения 0 и 1)
- считающие (S принимает значения от 0 до n)
- множественные (набор считающих семафоров). Все семафоры 1 набора м. устанавливаться 1 операцией.

Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо привилегированный процесс. В Windows после освобождения на семафоре приоритет процесса повышается.

При проверке флага мы не переходим в режим ядра. При вызове команды test-and-set (семафор) переходим в режим ядра. Изменение переменной S можно рассматривать как событие в системе.

Примеры использования:

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

Производство/потребление: Se (пустые ячейки), Sf(полные ячейки), S: int;

```
producer:   while (1) do                    consumer:   while (1) do
              P(Se);      // Se—                          P(Sf);      // Sf—
              P(S);       // Занять                       P(S);       // Занять
              N++                                         N--
              V(S);       // Освободить                   V(S);       // Освободить
              V(Sf);      // Sf++                         V(Se);      // Se++
begin: Se = N;   Sf = 0;          S = 1;          …
```

Читатели-писатели – монитор Хоара. Обедающие философы – множественные семафоры.

2. Режимы работы процессоров Intel Pentium. Прерывания в защищенном режиме (таблица ID).

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

Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес – сегмент/смещение). Минимальная адресная единица памяти – байт.

2^{20} = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).

Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

1-проц. режим под управлением MS-DOS (главное – минимизация памяти, занимаемой ОС => нет многозадачности).

0-256 б.: таблица векторов прерываний
резидентная часть DOS
место резидентных программ
сегменты программы
pool или heap (куча)
транзитная часть DOS
Резидентная часть DOS

1 Мб

2. Защищенный режим – многопроцессный режим. В памяти компьютера одновременно находится большое число программ с квантованием процессорного времени с виртуальной памятью. Управляет защищенным режимом ОС с разделением времени (Windows, Linux). В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.

Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Формирование таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

3. Специальный режим защищенного режима (V86) – 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.

Прерывания в защищенном режиме:

IDT (interrupt descriptor table) – таблица, предназначенная для хранения адресов обработчиков прерываний. Базовый адрес IDT помещен IDT Register. IDT столько, сколько процессоров.

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.

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. 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.

6

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

1) ***Программируемый***. Когда процессору встречается команда вв/выв, выполняя эту команду, процессор вызывает библиотечную функцию, кот. переводит процессор в режим ядра, осуществляет передачу команды вв/выв. Контроллер выполняет необходимые действия, затем устанавливает соответствующие биты в регистрах состояний вв/выв. Процессор д. периодически проверять состояние этих битов, т.е. опрашивать соответствующие регистры контроллера. На процессор возложено непосредственное управление вв/выв: опрос битов, пересылка команд чтения/записи, передача данных. Т.о. необходимо иметь следующий набор команд:

- команды управления (исп. для инициализации внешних устройств, указывают им, что делать);
- команды анализа состояния (проверка состояний флагов вв/выв);
- передача данных (для чт/зап команд или данных из/в внешн. устр. в/из регистры процессора);

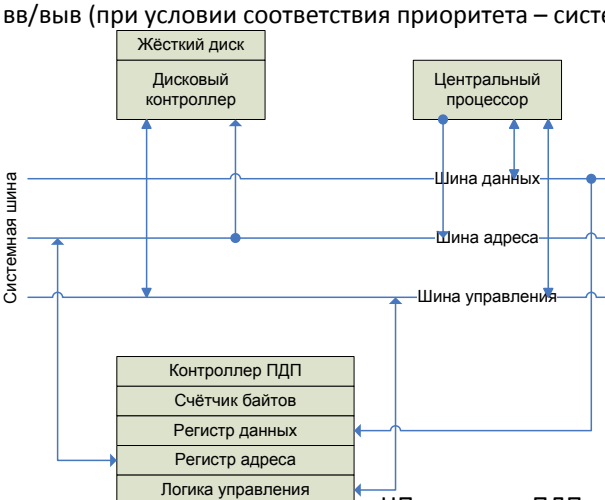
Недостаток: процессор постоянно опрашивает флаги – остается мало времени для остального.

2) ***С использованием прерываний.*** Прерывание от устройства вв/выв поступает, когда оно завершило процесс вв/выв. Процессор освобождается от проверки флагов и м. переключиться на др. работу. Метод требует включ. в состав ОС контроллера прерываний.

Контроллер прерываний посылает по шине управления сигнал. В конце выполнения каждой команды процессор проверяет входной сигнал с шины управления (если прерывания не замаскированы в ОС). Если получен сигнал – посылается ответный сигнал контроллеру прерываний, в ответ контроллер прерываний формирует и посылает вектор прерывания. Вектор передается по шине данных. Полученный вектор используется для процедуры обработки прерыв.

Процесс с точки зрения контроллера: Контроллер получает от ЦП команду (пр., read). Он переходит к считыванию данных со своего устройства. Как только эти данные окаж. в регистрах контроллера, посылается сигнал прерывания. После получения вектора ЦП-ом, контроллер посылает по шине данных данные из своих регистров.

Процесс с точки зрения процессора: Процессор генерирует команду read. ЦП выполняет вызов библиотечной функции, кот. переводит процессор в режим ядра (выполняется смена контекста: сохраняется содержим. программного счётчика команд, др. регистров), выполняющийся процесс блокируется, процессор переходит на выполнение др. работы. В конце каждого цикла команд процессор проверяет наличие прерываний. При поступлении прерывания ЦП сохраняет контекст выполненяемой задачи и переходит к выполнению п/п обработки прерывания. При этом процессор читает слова из контроллера вв/выв и пересылает их в память (через свои регистры). По завершению восстанавливает свой контекст команды, то кот. поступила команда вв/выв (при условии соответствия приоритета – система динамического изменения приоритетов).



Недостатки: Кажд. прерыв. вызывает остановку выполнен. текущих процессов и сохранение аппаратного контекста, любое слово, передающ. между памятью и устройством, д. пройти через регистры процессора. Но все равно намного эффективнее программируемого, т.к. искл. активное ожидание на процессоре.

3) ***Прямой доступ к памяти*** (ПДП, DMA – Direct Memory Access). Для управления ПДП подключ. контроллер ПДП (программно управл. устр. – спецпроцессор). Устройству вв/выв или каналу временно передается управление памятью (обычно на время, равное 1 циклу памяти), благодаря чему слово или группа слов м.б. записана в память. Процессор своей памяти не имеет, поэтому все осущ. через ОП. Передача – через регистры процессора.

ЦП передает ПДП: тип команды (read или write); адрес устройства; начальный адрес блока в ОП, используемого для чтения или записи; размер передаваемого блока.

+ Сокращение числа прерываний

– 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.

2. Защищенный режим: EMS; преобразование адреса при страничном преобразовании в процессорах Intel.

Защищенный режим – 32х-разрядный, поддерж. 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 ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.

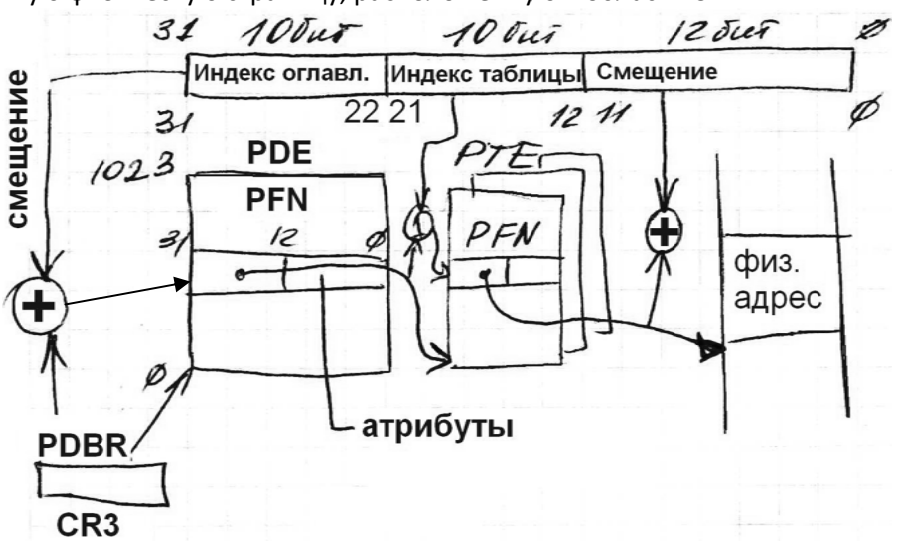
EMS (Expanded Memory Specification, спецификация отображаемой памяти) – программная спецификация, разработанный в 1985 г. фирмами Lotus, Intel и Microsoft для доступа из DOS к областям памяти выше 1 Мбайт в системах на базе процессоров 80386 и более поздних. *(расширенная, растягиваемая)*

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.



Если преобразование занимает несколько сегментов, значит должно быть столько таблиц страниц, сколько у него сегментов.

PDE – page directory entry (каталог, оглавление страниц)

PFN – page frame number

PTE – page table entry

PDBR – page directory base register (CR3)

Каталог стр. (PFN) – по 1 на процесс, м. содерж. 1024 дескр.

Табл. стр. – до 512 на –запрещ. адр. пр-ве, 512 на обычной памяти

Значение в регистре CR3 округл. до 4К – 1000h.

Значения в регистре стр. также округл. до 4К и указ. на физ. адр. байта.

Для ускорения необх. использовать ассоциативную память (кэш).

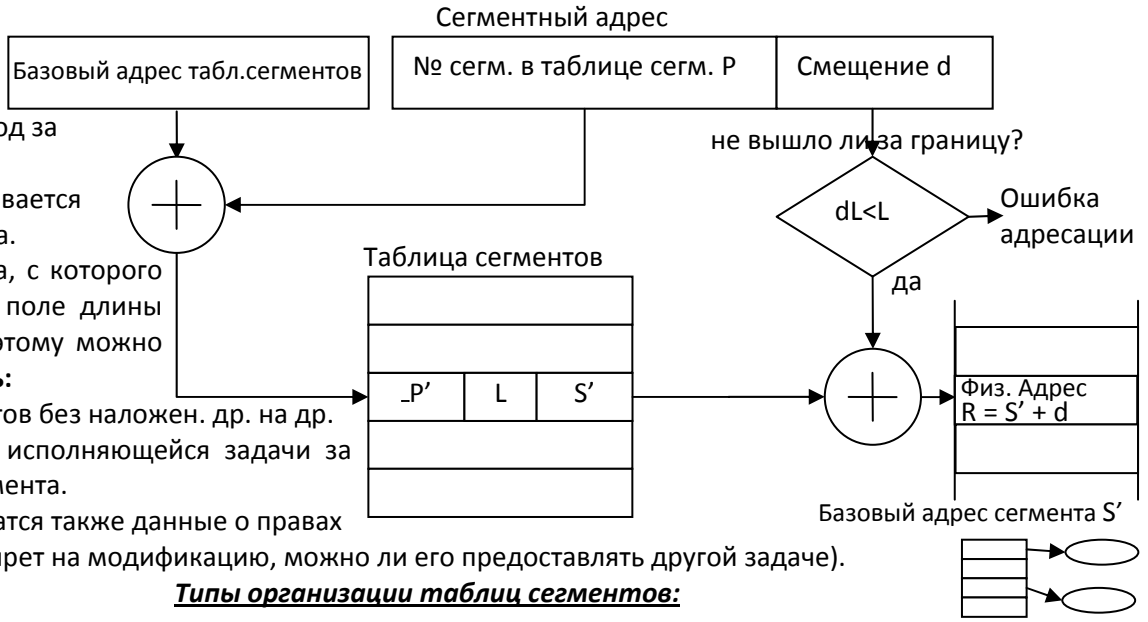
В Intel – кэш TLB (Translation Lookaside Buffer – буфер предварительной трансляции) – хранятся адреса стр., к кот. были последние обращения.

Есть также кэш данных и кэш команд.

1. Управление памятью: распределение памяти сегментами по запросам; стратегии выделения памяти; фрагментация.

При страничной организации виртуальное адресное пространство процесса делится механически на равные части. Это не позволяет дифференцировать способы доступа к разным частям программы (сегментам), а это свойство часто бывает очень полезным. Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент программы, а для сегмента данных разрешить только чтение. Кроме того, разбиение программы на "осмысленные" части делает принципиально возможным разделение одного сегмента несколькими процессами. Например, если два процесса используют одну и ту же математическую подпрограмму, то в оперативную память может быть загружена только одна копия этой подпрограммы.

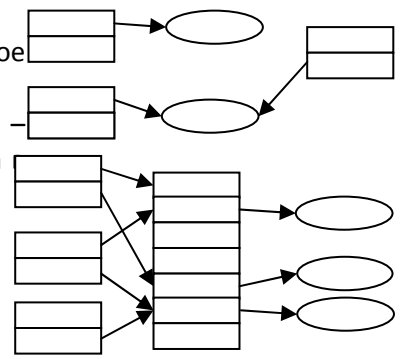
Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содерж. в них информации. Отдельный сегмент м. представлять собой подпрограмму, массив данных... При загрузке процесса часть сегментов помещ. в оперативную память (для каждого из этих сегментов ОС выбирает подходящий участок свободной памяти), а часть сегментов размещ. в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки.



Необх. контролир. размер сегмента.
Необх. контролир. выход за размер сегмента.
Каждый сегмент описывается дескриптором сегмента.
(содержит поле адреса, с которого сегмент начинается и поле длины сегмента). Благодаря этому можно осуществлять **контроль:**
1) размещения сегментов без наложен. др. на др.
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).

Типы организации таблиц сегментов:

1) Единая таблица – у сегмента есть единственное имя, при этом дескриптор содерж. список прав доступа каждого пользователя сегмента.
2) Локальные таблицы – каждая локальная таблица описывает определ. адресное пространство (опред. среду). В разн. средах 1 и тот же сегмент имеет разн. имена.
2) Локальные таблицы + глобальные таблицы – каждый сегмент имеет 2 ID – локальный и глобальный. Локальная таблица ссылается на дескриптор сегмента глобальной таблице. Физич. хар-ки описываютсяв главной таблице.



Алгоритмы, используемые при замещении сегментов:

1. Самый широкий
2. Самый узкий
3. Первый подходящий

Способы распределения памяти:

1. Связанное распределение
   a. Одиночное – программа загруж. в память целиком в последовательные адреса. Запрещ. обращ. программы в ОП ОС (регистр границы).
   b. Статическими разделами – размер разделов определяется в момент загрузки ОС.
   c. Динамическими разделами –разделы до начала выполнения задачи не устанавливаются.
2. Несвязанное выделение
   a. Несвязанные разделы опред. размера (страницы (д.б. таблицы страниц).
   b. Поделить память на логические единицы (сегменты)

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

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

Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес – сегмент/смещение). Минимальная адресная единица памяти – байт.

2^{20} = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).

Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

1-проц. режим под управлением MS-DOS (главное – минимизация памяти, занимаемой ОС => нет многозадачн.).

0-256 б.: таблица векторов прерываний

резидентная часть DOS, место резидентных программ, сегменты программы, pool или heap (куча),

транзитная часть DOS, Резидентная часть DOS → 1 Мб

2. Защищенный режим – многопроцессный режим. В памяти компьютера одновременно находится большое число программ с квантованием процессорного времени с виртуальной памятью. Управляет защищенным режимом ОС с разделением времени (Windows, Linux). В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.

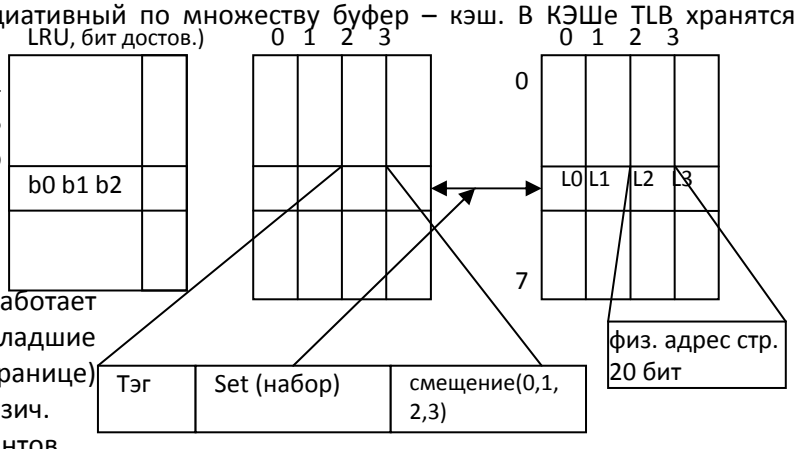
Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Формирование таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

3. Специальный режим защищенного режима (V86) – как задачи выполняются ОС реального режима, в кажд. из кот. выполн. по 1 прогр. реального режима. Многозадачный режим с поддержкой виртуальной памяти.

VMM (Virtual Machine Manager) – для запуска виртуальных машин реального режима.

TLB (translation look-aside buffer, буфер быстрого преобразования адреса, буфер предварительной трансляции) – таблица в блоке управл. памятью, отвечающая за преобразование виртуальных адресов в физические (в чипе).

TLB представляет собой четырехканальную ассоциативный по множеству буфер – кэш. В КЭШе TLB хранятся адреса страниц, к кот. были посл. обращения.



В блоке данных находится 8 наборов по 4 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 опр. 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 физич.

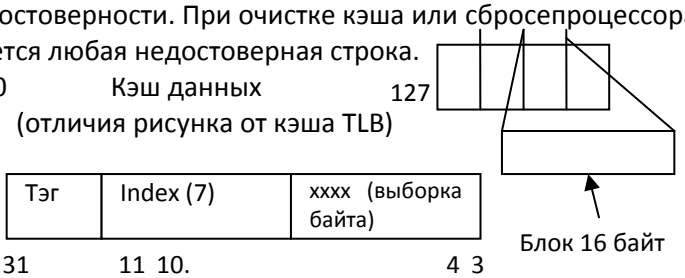
Блоку элементов данных соответствует блок элементов достоверности, атрибутов и тега (признака). Элемент тега состоит из 17 старших битов линейного адреса. При трансляции адреса процессор использует биты 12, 13 и 14 линейного адреса для выбора одного из восьми наборов, а затем проверяет четыре тега из 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. При очистке кэша или сбросепроцессора все биты достоверности сбрасываются в 0. При записи ищется любая недостоверная строка.

1. Если посл. обращ к стр. L0 или L1 → b0=1, иначе b 0 = 0
2. Если паре L0-L1 к L0 → b1=1
3. Если паре L2-L3 к L2 → b2=1

Данные – данные или команды (то, что по шине данных).

Кэш данных (отличия рисунка от кэша TLB)



8

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

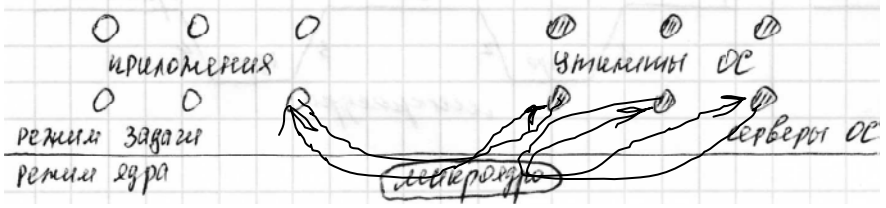
ОС разбивается на несколько уровней, причем обращение через уровень невозможно (непрозр. интерфейс).

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 – имеет блочную структуру, сост. из п/п. Модуль ОС, обеспечив. взаимод. между процессами ОС, и взаимод. пользоват. процессов между собой и с процессами ОС. Также вкл. функции самого низкого уровня – выделение аппаратных ресурсов. Драйверы имеют многоуровневую структуру. Драйвер-фильтр позвол. менять функциональность, не трогая драйвер нижнего уровня. 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)).

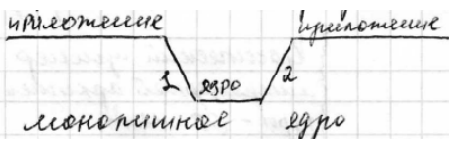
   (К серверам ОС относятся: сервер файлов, процессов, безопасности, виртуальной памяти)

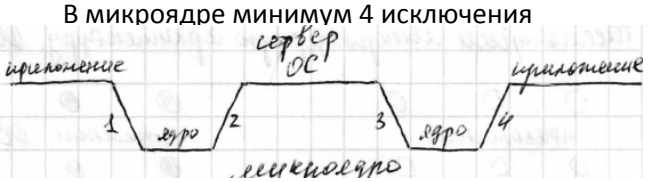


Монолитное ядро

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 – неделимая операция).

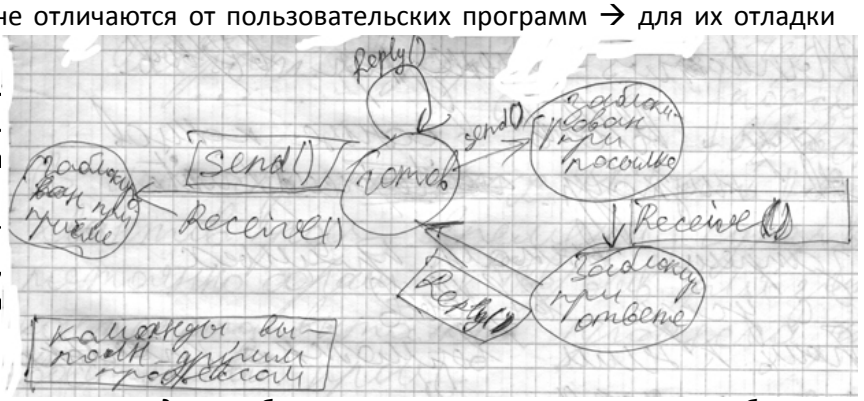
3 состояния процесса при передаче сообщения(протокол обмена):
- ✓ запрос: клиент запрашивает сервер для обработки запроса
- ✓ ответ: сервер возвращает результат операции
- ✓ подтверждение: клиент подтверждает прием пакета от сервера

Для обеспечения надежности обмена в протокол обмена могут входить следующие действия:
- ✓ сервер доступен? (запрос клиента)
- ✓ сервер доступен (ответ сервера)
- ✓ перезвоните (сервер – недоступен)
- ✓ адрес ошиб. (процесса с № нет в ОС)

\+ Высокая степень модульности ядра ОС (упрощает добавление новых компонентов). М. загружать и выгружать нов. драйверы, файловые системы и т. д., т.о. упрощается процесс отладки компонентов ядра.

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

\+ Повышается надежность системы, т.к. ошибка на уровне непривилег. программы < опасна, чем отказ на уровне режима ядра.



\- *Микроядерная архитектура ОС* вносит дополнительные накладные расходы, связанные с передачей сообщений, что сущ. влияет на производительность.

\- Для того чтобы микроядерная ОС по скорости не уступала *ОС* на базе *монолитного ядра*, треб. очень аккуратно проектировать разбиение системы на компоненты, стараясь минимизировать взаимодействие между ними.

2. Unix: команды fork(), wait(), exec(), pipe(), signal().

Unix создавалась как ОС разделения времени.

Базовое понятие Unix – процесс (единица декомпозиции ОС, программа времени выполнения). Процессрассматривается как виртуальная машина с собств. адресным пространством, выполн. пользов. прогр., предоставл. набор услуг. Процесс может находиться в двух состояниях – «задача» (процесс выполняет собственный код) и «система» или «ядро»(выполняет реентерабельный код ОС). Процессы все время переходят «пользователь» ←→ «система». Unix – ОС с динамическим управлением процессами.

Любой процесс может создавать любое число процессов с помощью системного вызова fork() – ветвление. Процессы образуют иерархию в виде дерева процессов, процессы связ. отношением потомок-предок.

В результате вызова fork создается процесс-потомок, который является копией процесса-предка (наследует адресное пространство предка и дескрипторы всех открытых файлов (фактически наследует код)). В Unix все рассматривается как файл (файлы, директории, устройства).

fork() возвращает 0 – для потомка, -1 – если ветвление невозможно, для родителя возвращ. натуральное число (ID потомка).Любой процесс имеет предка, кроме демонов.

Все процессы имеют прародителя –Init с ID = 0 (порожд. в нач. работы и существует до окончания работы ОС).

Все ост. порожд. по унифиц. схеме с помощью сист. вызова fork().

Системный вызов exec(), заменяет адресное пространство потомка на адресное пространство программы, указанной в системном вызове. exec () НЕ создает новый процесс!!!

Бывает шести видов: execlp, execvp, execl, execv, execle, execve. Например, execl(”/bin/ls”, ”ls”, “-l”, 0)

Системный вызов wait(&status) вызывается в теле предка. Предок б. ждать завершения всех своих потомков. При их завершении он получит статус завершения.

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

Неименованный программный канал создается системным вызовом pipe(). Труба (pipe) –это специальный буфер, который создается в системной области памяти. Информация в канал записывается по принципу FIFO и не модифицируется. Родственники могут обмениваться сообщениями с помощью неименованного программного канала (симплексная связь). Программные каналы описываются в соответствующей системной таблице. Канал имеет собственные средства синхронизации. Канал д.б. закрыт для чт/зап, если в/из него пишут/читают.

```
int mp[2];
pipe (mp);
…
close(mp[0]);
write(mp[1], msg, sizeof(msg));
…
```

Прогр. канал буфферизация. на 3-х уровнях: в сист. обл. памяти; при переполнении наибольш. врем. существ. пишется на диск.; если процесс пишет > 4 Кб, то канал буфер. по времени, пока все данные не б. прочитаны.

Любое важное событие в ОС сопровожд соотв. сигналом. Процессы м. порождать, принимать и обрабатывать сигналы. М.б. синхронные и асинхронные. Средство передачи сигнала – kill(), приема сигнала – signal(). Для изменения хода выполнения программы. Необходимо написать свой обработчик (в зависимости от того был получен сигнал или нет выполняются разные действия)

kill(getpid(), SIGALARM); – передать самому себе сигнал будильника.

Системный вызов signal возвращает указатель на предыдущий обработчик данного сигнала.

```
void catch(int num)
{       if(ChangedMode == 0) ChangedMode = 1;
        else ChangedMode = 0;                          }
…
signal(SIGINT, catch);    // обработать полученный сигнал
```

9

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

Virtual Memory – система при которой рабочее пространство процесса частично располагается в основной 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 пейджинга, 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.

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, по которому находится нужный элемент.

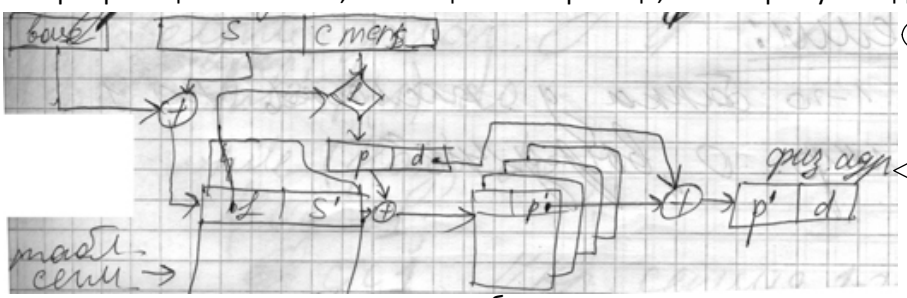
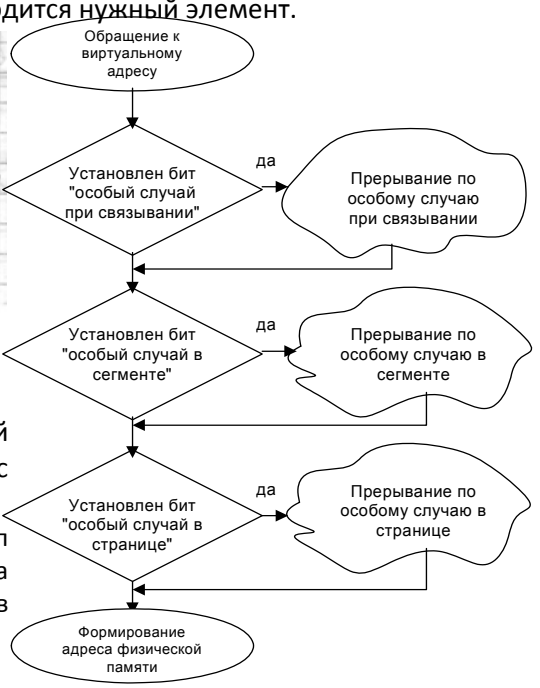


табл. страниц



Действия по управлению памятью при сегментно-страничном → распределении по запросам

Обращение к виртуальному адресу при отсутствии его в физической памяти может вызвать следующие действия связанные с соответствующими типами прерываний или особых случаев:

1. Прерывание по особому случаю при связывании. Данный тип прерывания возникает при отложенном связывании. Для данного сегмента заполняется строка таблицы сегментов и устанавливается бит особ. случая в сегменте, заполняется элемент в таблице активн. ссылок.
2. 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. Когда сегмент находится в памяти, обращение к таблице страниц может показать, что нужная страница отсутствует в памяти. Менеджер памяти возьмет, найдет нужную страницу во внешней памяти и загрузит ее в основную память. М. потреб. замещение к-л. находящейся в памяти страницы. Таблица страниц корректир.
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.

На самом верхнем уровне находится таблица процессов, в которой для каждого процесса, выполн. в ОС, выделена строка. Строка таблицы процессов указывает на таблицу сегментов этого процесса. Строка таблицы сегментов указывает на таблицу страниц соответствующего сегмента. Каждая строка таблицы страниц указывает или на страничный кадр, в кот. размещ. данная страница, или на адрес внешней памяти, где хран. эта страница.

\+ страничного: легко реализовать, алгоритм LRU в этом достаточно эффективен

– страничного: сложность коллективного использования.

\+ сегментного: легко реализовать коллективное использование, т.к. сегмент является лог. 1 деления памяти

– сегментного: необх. корректировки таблицы дескрипторов всех процессов при изменении размеров сегментов

– сегментного: сложн. при загрузке новых сегментов (в памяти д. сущ. адр пространство необходимого размера).

– сегментного: Фрагментация (Интенсивная загрузка, и выгрузка может привести к маленьким участкам, в которые загрузить ничего не удастся), хотя система может устранить её путём переноса сегментов

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

Unix создавалась как ОС разделения времени.

Базовое понятие Unix – процесс (единица декомпозиции ОС, программа времени выполнения). Процессрассматривается как виртуальная машина с собств. адресным пространством, выполн. пользов. прогр., предоставл. набор услуг. Процесс может находиться в двух состояниях – «задача» (процесс выполняет собственный код) и «система» или «ядро»(выполняет реентерабельный код ОС). Процессы все время переходят «пользователь» ←→ «система». Unix – ОС с динамическим управлением процессами.

Любой процесс может создавать любое число процессов с помощью системного вызова fork() – ветвление. Процессы образуют иерархию в виде дерева процессов, процессы связ. отношением потомок-предок.

В результате вызова fork создается процесс-потомок, который является копией процесса-предка (наследует адресное пространство предка и дескрипторы всех открытых файлов (фактически наследует код)). В Unix все рассматривается как файл (файлы, директории, устройства).

fork() возвращает 0 – для потомка, -1 – если ветвление невозможно, для родителя возвращ. натуральное число (ID потомка). Любой процесс имеет предка, кроме демонов.

Все процессы имеют прародителя –Init с ID = 0 (порожд. в нач. работы и существует до окончания работы ОС).

Все ост. порожд. по унифиц. схеме с помощью сист. вызова fork().

Системный вызов exec(), заменяет адресное пространство потомка на адресное пространство программы, указанной в системном вызове. exec () НЕ создает новый процесс!!!

Системный вызов wait(&status) вызывается в теле предка. Предок б. ждать завершения всех своих потомков. При их завершении он получит статус завершения.

«Сирота» — процесс, родитель которого завершился раньше него. При завершении процессов ОС проверяет, не осталось ли у процессов незавершившихся потомков, если остались – принимает усилия по усыновлению – редактирует строку данного проц.-потомка, заменяя строку предка на 1 (усыновл. терминальным процессом).

«Зомби». Если процесс-потомок завершился до тех пор, пока предок успел вызвать wait (ghb аварийном завершении), то для того, чтобы предок не завис в ожидании несуществующего процесса, ОС отбирает все ресурсы ку процесса (оставляет только строку в таблице процессов) и помечает процесс-потомок как «зомби». Это делается для того, чтобы предок, дойдя до wait() смог получить статусы завершения всех своих потомков.

«Демон» — процесс, который не имеет предков. Срабатывает по определенному событию. «Демон» Unix примерно соответствует «сервису» Windows.

Зомби

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

Демон

```
int daemonize(void) {
switch (fork()) {
case 0:
return setsid();
case -1:
return -1;
default:
exit(0);}}
```

10

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

ОС разбивается на несколько уровней, причем обращение через уровень невозможно (непрозр. интерфейс).

В качестве ядра выделяется самый низкий уровень распределения аппаратных ресурсов процессам самой ОС (непосредственное обращение с аппаратурой). Существует 2 типа структур ядер: монолитное ядро и микроядро.

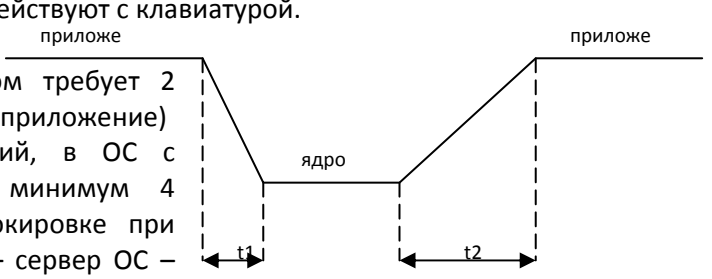
Монолитное ядро – программа, состоящая из подпрограмм (имеющая модульную структуру), содержащих в себе все функции ОС, включая планировщик, файловую систему, драйверы, менеджеры памяти. Поскольку это 1 программа, то она имеет 1 адресное пространство → все её подпрограммы имеют доступ ко всем её внутренним структурам. Такие ОС делятся на 2 части – резидентную и нерезидентную. Любые изменения приводят к необходимости перекомпилирования всей ОС. Пример: ОС UNIX, хотя она имеет минимизированное 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).



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. 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 – являются синхронным событием, возникают в процессе выполнения программы, когда (пр.):

Арифметическое переполнение, /0, Попытка выполнить некорректную команду, Ссылка на запрещенную область памяти, При образовании адреса, при обращении к физическому адресу, которого нет.

1. *Исправимые* – процесс м. продолжаться с той же команды, в которой произошло исключение.
2. *Неисправимые* - заканчиваются завершением программы

Системные вызовы

Также являются исключениями, но с точки зрения реализации - это системные ловушки. Набор м. 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, кот. передает № системного вызова в стек пользователя и вызывает спец. инструкцию системного прерывания, которое меняет режим задачи на режим ядра и передает управление обработчику системного вызова. Системные вызовы выполняются в режиме ядра, но в контексте пользователя (задачи, процесса). Имеют доступ к адресному пространству и управляющим структурам, вызвавшего их процесса, также они м. обращаться к стеку ядра этого процесса.

Точные прерывания – прерывание, оставляющее машину в строго определенном состоянии. Обладает св-ми:

1. Счетчик команд указывает на команду (текущая), до которой все команды выполнены.
2. Ни одна команда после текущей не выполнена.

Не говорится, что команды после текущей не могли начать выполнение, а то, что все изменения, связанные с этими командами, должны быть отменены.

3. Состояние текущей команды известно (аппаратн. – обычно не нач. вып., при искл. – привела к исключ.)

2. Задача: читатели-писатели, решение с использованием семафоров Дейкстра для ОС Unix.

```
#define READERS 0 ReadersQueue,
#define WRITERS 1 WritersQueue,
#define ACTREAD 2 ActiveReaders,
#define WRTPROC 3 IsWriting (in Process)
        // Semaphore operation: P, V, Wait
#define P 0
#define V 1
#define W 2
int semathores;
unsigned short initialValues[4] = {0, 0, 0, 0};

        // Semathores operations
struct sembuf readers[3] = {{READERS, -1, 0}, {READERS, 1, 0}, {READERS, 0, 0}};
struct sembuf writers[3] = {{WRITERS, -1, 0}, {WRITERS, 1, 0}, {WRITERS, 0, 0}};
struct sembuf actread[3] = {{ACTREAD, -1, 0}, {ACTREAD, 1, 0}, {ACTREAD, 0, 0}};
struct sembuf writing[2] = {{WRTPROC, -1, 0}, {WRTPROC, 1, 0}};

int *data;
int semID, shmID, procID, procID2;
void startRead()
{
        semop(semID, &readers[V], 1);
        semop(semID, &writing[W], 1);
        semop(semID, &writers[W], 1);
        semop(semID, &actread[V], 1);
        semop(semID, &readers[P], 1);
}

void stopRead()
{
        semop(semID, &actread[P], 1);
}

void Read(int N)
{
        startRead();
        printf("Reader %d: %d\n", N, *data);
        slp(0);
        stopRead();
}
void startWrite()
{
        semop(semID, &writers[V], 1);
        semop(semID, &actread[W], 1);
        semop(semID, &writing[P], 1);
}

void stopWrite()
{
        semop(semID, &writing[V], 1);
        slp(0);
        semop(semID, &writers[P], 1);
}

void Write(int N)
{
        startWrite();
        (*data)++;
        printf("Writer %d: %d\n", N, *data);
        slp(0);
        stopWrite();
}
int main()
{
        int perms = S_IRWXU | S_IRWXG | S_IRWXO;

        if ((semID = semget(IPC_PRIVATE, 4, IPC_CREAT | perms)) == -1)
                er("semget", 1);
        printf("SemaphoreID = %d\n", semID);
        semctl(semID, 0, SETALL, &initialValues);

        if ((shmID = shmget(100, sizeof(int), IPC_CREAT | perms)) == -1)
                er("shmget", 2);
        printf("SharedMemoryID = %d\n", shmID);
        data = (int*)shmat(shmID, 0, 0);
        *data = 0;
        if ((procID = fork()) == 0)
```

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 1 писатель. Нет смысла ограничивать количество процессов-читателей, так как они не измен. содержимое БД. Когда число читателей = 0, писатель получает возможность начать работу. Новый читатель не м. начать работу, пока не завершитсяписатель. Писатель может начать работу, когда c_write принимает значение «истина» (писать можно). Когда читателю необходимо провести чтение, он вызывает start_read(), когда завершить – stop_read(). Он м. читать, если нет писателя, который изменяет в данный момент данные, а также если нет писателя в очереди. Второе условие необходимо, чтобы предотвратить бесконечное откладывание писателей. start_read() заканчивается signal(c_read), который пробуждает следующего читателя, ожидающего в очереди. Следующий читатель начинает работать и т.д. Возникает цепная реакция читателей, продолжающаяся, пока в очереди есть неактивированные читатели. Так как они не мешают друг другу, то читатели могут выполняться реально параллельно. Такая цепная реакция обслуживается по принципу FIFO. В stop_read() количество читателей уменьшается на 1 и может стать равным 0. В этот момент вырабатывается сигнал «можно писать». Процессы-писатели для начала работы вызывают start_write(). Для обеспечения монопольного доступа к полю БД, если есть процесс-писатель или другой активный писатель, то писатель переводится в состояние ожидания, пока c_write не равно значению «истина». Получив возможность работать, писатель присваивает логической переменной wrt значение «истина», тем самым блокируя доступ других писателей к данному полю. Чтобы не возникло бесконечного откладывания читателей, писатель проверяет, нет ли ожидающих читателей. Если они есть, то читатель из очереди активизируется. Если нет, то подаётся сигнал о возможности работы следующего писателя.

11

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 лог. переменной B, копирует его в A, а затем устанавливает для B значение True.

test_and_set(a, b): a = b; b = true;

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

```
flag, a1, a2: boolean;
P1:   a1 = 1;                                  P2:   a2 = 1;
      while(a1 == 1)                                 while(a2 == 1)
            test_and_set(a1, flag);                        test_and_set(a2, flag);
      CR1;                                           CR2;
      flag = 0;                                      flag = 0;
      PR1;                                           PR2;
```

Flag = true, когда один из процессов – в своем критическом участке.

Теоретически не исключено бесконечное откладывание, но вероятность → 0 (т.к. test_and_set – машинная неделимая команда и выполняется очень быстро.

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

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

Программная реализация (алгоритм Деккера).

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

```
flag1, flag2: boolean;
queue: integer;
p1:   flag1 = 1;                               p2:   flag2 = 1;
      while(flag2)do                                 while(flag1)do
            if(queue == 2) then                            if(queue == 1) then
            begin                                          begin
                  flag1 = 0;                                     flag2 = 0;
                  while(queue == 2)do;                           while(queue == 1)do;
                  flag1 = 1;                                     flag2 = 1;
            end;                                           end;
      CR1;                                           CR2;
      flag1 = 0;                                     flag2 = 0;
      queue = 2;                                     queue = 1;
end P1;                                        end P2;

main: flag1 = 0;
      flag2 = 0;
      queue = 1; // Или 2
      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()). Пр. использования.

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 б. 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 - освободить).

```
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).

```
struct sembuf writing[2] = {{WRTPROC, -1, 0}, {WRTPROC, 1, 0}};
```

1. Если значение `sem_op` отрицательно, то:
   a. Если значение семафора больше или равно абсолютной величине `sem_op`, то абсолютная величина `sem_op` вычитается из значения семафора.
   b. 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:
   a. Если значение семафора = 0, то управление сразу же возвращается вызывающему процессу.
   b. Если значение семафора не <> 0, то выполнение вызывающего процесса приостанавливается до установки значения семафора в 0.

```
int *data;semathores;
int semID, shmID, procID, procID2;
int perms = S_IRWXU | S_IRWXG | S_IRWXO;

if ((semID = semget(IPC_PRIVATE, 4, IPC_CREAT | perms)) == -1)
        er("semget", 1);
printf("SemaphoreID = %d\n", semID);
semctl(semID, 0, SETALL, &initialValues);

if ((shmID = shmget(100, sizeof(int), IPC_CREAT | perms)) == -1)
        er("shmget", 2);
printf("SharedMemoryID = %d\n", shmID);
data = (int*)shmat(shmID, 0, 0);
*data = 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. 2-ой определяет, что операция д. откатываться при завершении процесса.

```
semop(semID, &writing[P], 1);
```

12

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

Процесс - программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).
М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.
Аппаратный контекст процесса – сост. регистров. Полный контекст процесса – сост. регистров + сост. памяти.
Планирование – управление распределением ресурсов ЦП между разл. конкурирующими процессами путем передачи им управления согласно некот. стратегии планирования.
Диспетчеризация – выделение процессу процессорного времени.
Программа-планировщик – программа, отвечающая за управление использованием совместного ресурса.

1. Последовательное выполнение: I----------I p1
   I----------I p2
2. Квазипараллельное выполнение: I---------I I---------I p1
   I---------I I---------I p2
3. Реальная параллельность: I----------I p1
   I----------I p2

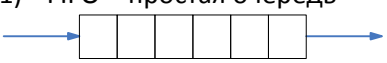
Классификация алгоритмов планирования:

- 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: абсолютные и относительные, статические и динамические}
- С вытеснением / без вытеснения

Бесконечное откладывание – ситуация, когда процесс никогда не получает необх. для выполнения ресурсов (точнее, кванта времени). Возникает, когда диспетчер всегда отдаёт квант др. процессу, т.к. его приоритет >.

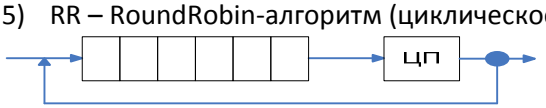
Алгоритмы планирования *(1–4 – для ОС пакетной обработки, 5– –для др. ОС):*

1) FIFO – простая очередь



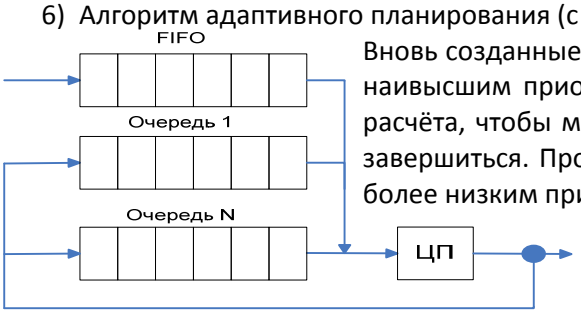
Процессор Без переключения, без приоритетов.

2) SJF – Shortest Job First – наикратчайшее задание – первое. М.б. бесконечное откладывание. Д.б. априорная инф. о времени выполнения программы, объеме памяти. Статич. приор., без переключ.
3) SRT – Shortest Remaining Time – с вытеснением. Выполняющийся процесс прерывается, если в очереди появляется процесс с меньшим временем выполнения (меньше оставшегося до заверш. времени).
4) HRR – Highest Response Ratio (наиболее высокое относительное время ответа) – с динамич. приоритетами. Используется в UNIX. $priority = \frac{t_W - t_S}{t_S}$, t_W – время ожидания, t_S - запрошенное время обслуживания. Чем больше ожидает, тем больше приоритет. Позволяет избежать бесконечного откладывания.
5) RR – RoundRobin-алгоритм (циклическое планирование) – с переключ., без вытеснения, без приоритетов



Разл. процессы м. иметь разл. величину кванта.

6) Алгоритм адаптивного планирования (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 максимальное количество процессов успело выдать запрос IO или завершиться. Процессы, не сделавшие ни того, ни другого, переходят в очередь с более низким приоритетом по алгоритму RR. В очереди N «крутится» idle-процесс (холостой) и там же оказываются все вычислит. процессы.

Также может учитываться объем памяти, необходимый процессу – адаптивно-рефлексивное планирование, т.е. выделяется очередной квант только при наличии свободной памяти в системе.

В современных системах:
Процессы создаются по мере необходимости, ресурсы выделяются по мере надобности. Априорная информация о времени выполнения процессов отсутствует.

2. Защищ. режим работы ПК с процессорами Intel (486, ..). Уровни привилегий. Сист. таблицы: GDT, LDT, IDT.

Защищенный режим – 32х-разрядный, поддерж. многопоточность и многопроцессность. В отличие от реального режима здесь доступно 4 Гб памяти (в реальном диапазон адресов памяти ограничен 1 мб). В защищ. реж. 4 уровня привилег. Ядро ОС находится на 0-м уровне.

GDT (global descriptor table) – таблица, которая описывает сегменты основной памяти ОС. В системе только 1 GDT. На начальный адрес GDT указывает GDT Register (32 разрядный).

IDT (interrupt descriptor table) – таблица, предназначенная для хранения адресов обработчиков прерываний. Базовый адрес IDT помещен IDT Register. IDT столько, сколько процессоров.

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.

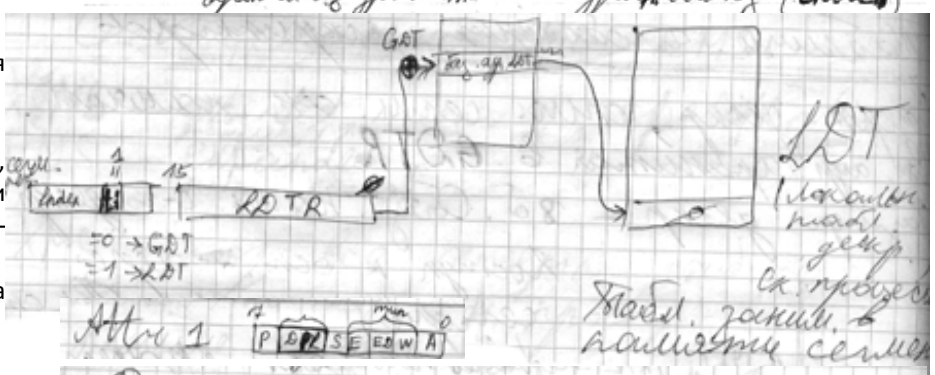
Формат селектора (явл. ID сегмента):

Индекс равен (кратен) 8 и является смещением в таблице дескрипторов.

Обязательно наличие 0-го дескриптора.

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 (GDT):

A – access – бит доступа к сегменту

Тип – опред. поля доступа:

DPL – уровень привилегий

P – бит присутствия, исп. для работы с ВП.

D – бит разрядн. операндов и адр. (0 – 16, 1 – 32)

G – бит гранулярности (0 – размер сегмента задан в байтах, 1 – в страницах по 4 Кб).

Процесс не м. выйти за размер своего сегмента – контроль ОС (защита адресных простр-в сегментов др. от др.). К дескр. GDT и LDT обращаемся с помощью селекторов, к дескр.

IDT обращаемся по смещению, кот. берем из прерывания. Обработчик прерывания: IDTR (указывает на начало IDT) + смещение из прерывания = дескриптор в IDT. Из дескр. в IDT берем селектор. С помощью селектора узнаем, с какая табл.

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 каждому из сегментных регистров (CS, DS, SS, ES, FS, GS) сопоставлен теневой регистр (дескриптора). Он не доступен программисту и загружается || автоматически из таблицы дескрипторов соотв. сегмента чтобы реже обращ. к ОП.

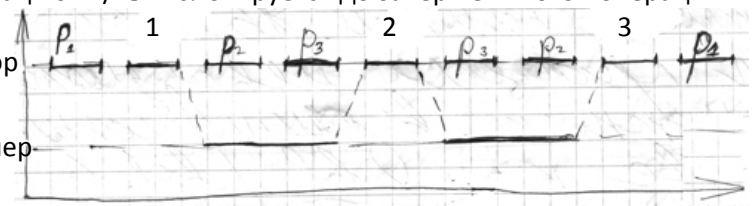
Уровни привилегий (кольца защиты): В защищенном режиме предусмотрен механизм защиты с помощью системы уровней привилегий. Существует 4 уровня привилегий: от 0 до 3. 0-ой уровень – предоставляет макс. привилегии, используется для ядра ОС. 3-ий уровень – исп. для прикладных программ. Каждому сегменту программы придется опред. уровень привилег., указываемый в поле DPL (Descriptor Privilage Level, уровень привилег. дескр.) его дескр. Уровень привилег., 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.

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

Задача системы вв/выв – обеспечить взаимодействие процессора с внешними устройствами.

Синхронный – приложение, запросившее операцию вв/выв. блокируется до завершения этой операции.

1 – обработка системного вызова
2 – процедура вв/выв
3 – завершение операции вв/выв

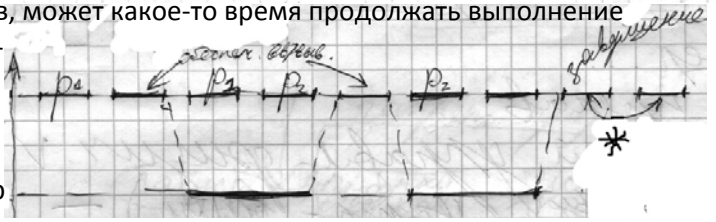


Синхронное выполнение →

Асинхронный – приложение, подавшее запрос на вв/выв, может какое-то время продолжать выполнение

* – синхронизация (мьютекс, событие, порт завершения-completion port).

Отложенная запись – lazy write.



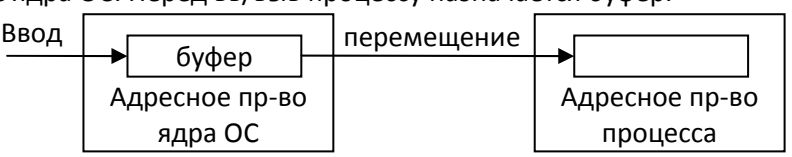
Асинхронное выполнение →

Синхронный: ОС стремится выгрузить процесс из памяти, но для данных необходимо оставить блок данных в памяти – проблема.

При вв/выв используется буферизация.

Одинарный буфер – выделяется в пространстве ядра ОС. Перед вв/выв процессу назначается буфер.

Сначала данные помещаются в буфер, затем копируются в адресное пространство процесса. Операция вв/выв выполняется быстрее, т.к. процесс не блокиров. – все данные получ. сразу).



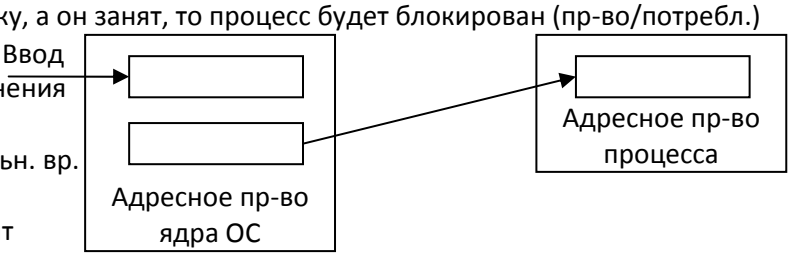
Опережающее считывание.

Исключаются проблемы, т.к. буфер находится в системной (невыгружаемой) области памяти.

Схемы передачи информации: блок-ориентированная (поточно-ориент.), побайтно-ориент. (построчно).

Если процесс пытается поместить в буфер строку, а он занят, то процесс будет блокирован (пр-во/потребл.)

Используется двойная буферизация:



1 буфер передается, др. -использ-ся для заполнения – усложняются действия ОС

Проблема буферизации остро встает в ОС реальн. вр.

2 способа перемещения:

1. Копирование – передача последов. байт в обл. памяти.
2. Мэпинг – получение указателя на адресное пространство в ядре ОС.

ОС должна сообщать о завершении вв/выв приложению.

Классич. Unix не выполняет асинхронного вв/выв, предоставл. прикл. программе чисто синхр. интерфейс.

Современный Unix позволяет программисту выбирать отложенной записью (Fire&Forget) и чисто синхр. Разработчик сам должен разрабатывать нити. Доп. нить должна сообщать основной о завершении операции вв/выв. Чаще всего исп. семафоры, сигналы, спинлоки, мьютексы. **«–»** усложнение программной логики.

В Windows 200/XP для асинхронного вв/выв исп. порты завершения или др. объекты синхронизации (реже).

Порт завершения – механизм сообщения по токам о завершении операции вв/выв. Если файл сопоставить с портом завершения, то по окончании операции вв/выв в очередь порта завершения став. пакет завершения. Приложение проверяет наличие пакета завершения в порте завершения и т.о. узнает о факте завершения.

Независимо от типа запроса операции вв/выв инициализ. драйвером действия в интересах приложения выполняется асинхронно. После выдачи запроса драйвер возвращает управление подсистеме вв/выв. Когда подсистема вв/выв вернет управление зависит от типа запроса.

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

Базовым понятием в UNIX является процесс. **Процесс** - программа в стадии выполнения, единица диспетчеризации. В UNIX принято рассматривать процесс как виртуальную машину с собственным адресным пространством. Эта виртуальная машина выполняет программу, предоставляя набор услуг.

Средства взаимодействия процессов: программные каналы (именованные и неименованные), сигналы, разделяемая память, очереди сообщений, семафоры, ввод/вывод с отображением в память

Программный канал – это специальный буфер, который создается в системной области памяти. Они описываются в соответствующей системной таблице.

Информация в канал записывается по принципу FIFO и не модифицируется. В канал нельзя писать, если его читают и наоборот. 2 типа программных каналов: именованные, неименованные.

- Именованные каналы создаются mknod name p

- pipe() или | – создает неименованный программный канал. (есть дескриптор, но нет id)

Предок и потомок могут обмениваться сообщениями с помощью неименованного программного канала.

1 процесс пишет, другой считывает из трубы – симплексная связь.

Канал является средством передачи информации, который не имеет собственных средств синхронизации.

Программный канал буферизуется на 3х уровнях.

В системной области памяти. При переполнении трубы, наиболее долго существующей, переносится на диск.

Если процесс пытается записать более 4 Кб данных, то труба буфирезуется во времени, приостанавливая «писателя», пока все данные не будут прочитаны.

Техника сигналов поддерживается ОС и отвечает стандартным требованиям системы прерываний, к которой относятся обработка системных исключений, снешних и внутренних прер-й, а также маскируемых и немаскируемых прер-й. В UNIX процессы могут порождать и принимать сигналы. Сигналы: синхронные(порождены процессом) и асинхронные(внешним действием: польз или ядром).

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

signal(SIGTERM, catch_sig | SIG_IGN | SEG_DFL);

Разделяемые сегменты – средство взаимодействия процессов через разделяемое адресное пространство. Т.к. адресное пространство защищено, процессы могут взаимодействовать только через ядро. Подключается к адресному пространству процесса (виртуальный).

+ быстрота передачи информации. Осуществляется мэппинг, но не копирование => повыш. быстродействие.

Разделяемая панять не имеет мредмтв взаимоисключения.

В ядре создается таблица разделяемых сегментов.

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.

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.

Для поддержки очереди сообщений в адресном пространстве ядра создается система сообщений.

Шаблон сообщения описывает Struct msgbuf {long mytype; char mytext[MSG_MAX];}

Ядро только выполняет размещение и выборку сообщений. Менеджер ресурсов отслеживает число очередей. При посылке сообщения производится копирование текста сообщение в адресное пространство ядра системы. При получении – обратная операция. **Недостаток**: двойное копирование.

Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает его в связный список указанной очереди. Процесс, отправивший сообщение, может завершиться.

Когда какой-либо процесс выбирает сообщение из очереди, он может выбрать :

1) самое старое сообщение независимо от его типа

2) по указанному id. Если существует несколько сообщений, то берется самое старое.

3) взять сообщение, числовое значение типа которого является наименьшим из меньших или равным значению типа, указанного процессом.

Т.о. ни отправитель, ни получатель не будут заблокированы. Msgget, msgctl, msgsnd, msgrev

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

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

Использование семафоров часто приводит к взаимоблокировке. Понятие «монитор» предложил Хоар.

Монитор – языковая конструкция, состоящая из структур данных и подпрограмм, использующих данные структуры. Монитор защищает данные. Доступ к данным монитора могут получить только п/п монитора. В каждый момент времени в мониторе м. находиться только 1 процесс. Монитор является ресурсом.

Процесс, захвативший монитор, – процесс **в** мониторе, процесс, ожидающий в очереди, – процесс **на** мониторе.

Используется переменная типа «событие» для каждой причины перевода процесса в состояние блокировки. Над переменной – 2 операции:

1. wait – открывает доступ к монитору, задержив. выполнение процесса. Оно д.б. восстановлено операцией
2. signal другого процесса.

Если очередь перем. к типу усл. <> 0, то из очереди выбирается 1 из процессов и инициализируется его выполн.

Если очередь пуста, то signal ничего не выполняет.

Чтобы гарантировать, что процесс, обратившийся к ресурсу, когда-нибудь его получит, необходимо, чтобы процесс, находящийся в ожидании имел больший приоритет, чем вновь пытающийся обратиться в монитор.

Мониторы м. входить в состав языка программирования (например, C#).

Простой монитор – предназначен для разделения 1 единственного ресурса произвольному числу процессов. Монитор, обслуживающий произвольное количество процессов, ограничен длиной своей очереди.

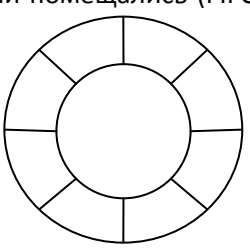
```
monitor distribution;
var   busy: logical;
      free: conditional;

      // Захват
procedure acquire;
begin
      if (busy) then
            wait(free);
      busy := true;
end;

      // Освобождение
procedure release;
begin
      busy := false;
      signal(true);
end;

begin
      busy := false;
end.
```

Монитор «кольцевой буфер» – решает задачу «производство-потребление», 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”).



```
resource: monitor circle_buffer;
const n = size;
var
      buffer: array [0..n-111]]] of <type>;
      pos, write_pos, read_pos: 0..n-1;
      full, empty: conditional;

procedure producer(p: process, data:<
type>)
begin
      if (pos = n) then
            wait(empty);
      buffer[write_pos] := data;
      Inc(pos);
      write_pos = (write_pos + 1) mod n;
      signal(full);
end;

procedure consumer(p: process, var data:
<type>)
begin
      if (pos = 0) then
            wait(full);
      data := buffer[read_pos];
      Dec(read_pos);
      read_pos := (read_pos + 1) mod n;
      signal(empty);
end;

begin
      pos := 0;
      write_pos := 0;
      read_pos := 0;
end.
```

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

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

Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес – сегмент/смещение). Минимальная адресная единица памяти – байт.

2^{20} = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).

Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров. Компьютер начинает работать в реальном режиме.

Защищенный режим – 32х-разрядный, поддерж. многопоточность и многопроцессность. В отличие от реального режима здесь доступно 4 Гб памяти (в реальном диапазон адресов памяти ограничен 1 мб). В защищ. реж. 4 уровня привилег. Ядро ОС находится на 0-м уровне.

Уровни привилегий (кольца защиты): В защищенном режиме предусмотрен механизм защиты с помощью системы уровней привилегий. Существует 4 уровня привилегий: от 0 до 3. 0-ой уровень – предоставляет макс. привилегии, используется для ядра ОС. 3-ий уровень – исп. для прикладных программ. Каждому сегменту программы придется опред. уровень привилег., указываемый в поле DPL (Descriptor Privilage Level, уровень привилег. дескр.) его дескр. Уровень привилег., 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.

Чтобы перейти в защищенный режим, достаточно установить бит PE — нулевой бит в управляющем регистре CR0, и процессор немедленно окажется в защищенном режиме. Единственное дополнительное требование, которое предъявляет Intel, чтобы в этот момент все прерывания, включая немаскируемое, были отключены.

Инициализируем GDT, IDT загружаем с помощью привилегированных команд в регистры, перенастраиваем контроллер прерываний на новый базовый вектор 20h, запрещаем прерывания устанавливаем бит PE в 1.

Переключение в защищенный режим:

1. Открыть адресную линию A20.
2. Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, она в нулевом кольце защиты).
3. Подготовить в оперативной памяти таблицу дескрипторов прерываний IDT.
4. Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по определенному адресу, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по этому адресу.
5. Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
6. Запретить все маскируемые и немаскируемые прерывания. Сохранить маски прерываний. Перепрограммировать контроллер прерываний.
7. Загрузить регистр GDTR и IDTR.
8. ***Перейти в защищенный режим*** (установить бит PE — нулевой бит в управляющем регистре CR0 в 1).
9. Загрузить новый селектор в регистр CS.
10. Загрузить сегментные регистры селекторами на соответствующие дескрипторы.
11. Разрешить прерывания.

Возвращение в реальный режим:

1. **Переключиться в реальный режим**
2. Сбросить очередь предвыборки, загрузить CS реальным сегментным адресом
3. Задать регистры для работы в реальном режиме
4. Загрузка IDTR (Interrupt Descriptor Table Register) для реального режима
5. Разрешаем немаскируемые прерывания
6. Разрешить маскируемые прерывания

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

Virtual Memory – система при которой рабочее пространство процесса частично располагается в основной 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. Используется адресное пространство диска как область свопинга или педжинга, т.е. для временного хранения областей памяти.

Виртуальная память – память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или пейджинга, 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

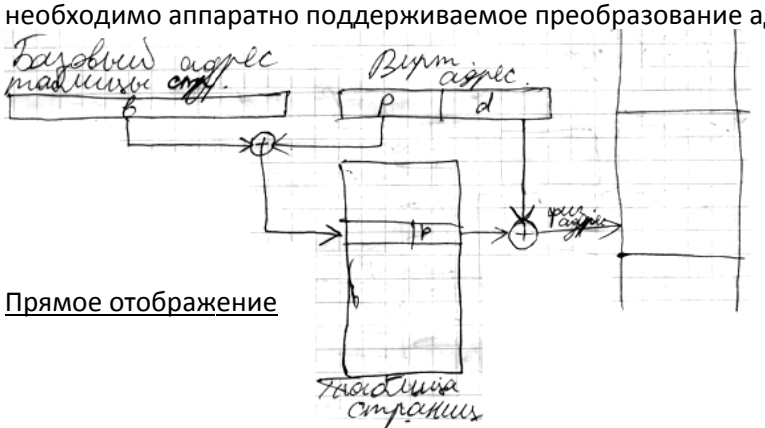
Страница - является единицей физического деления памяти. Её размер устанавливается системой.

Сегмент – является единицей логического деления памяти. Её размер определяется объемом кода.

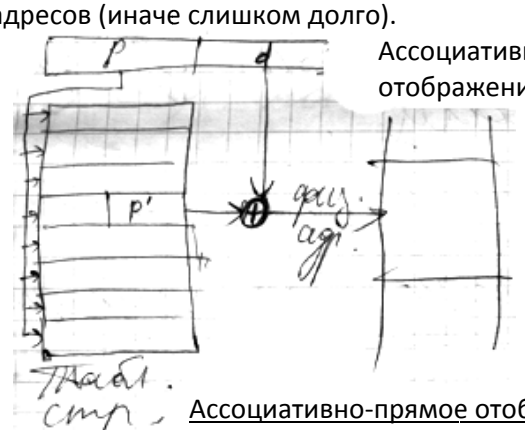
Распределение памяти страницами по запросам

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).



Прямое отображение



Ассоциативное отображение

+ легко реализовать, алгоритм LRU в этом эффект.

– сложность коллективного использования

Первый бит в таблице страниц – бит присутств.

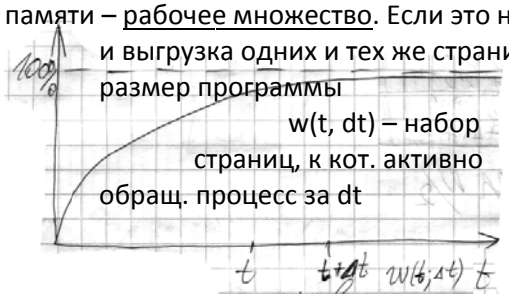
1 – страница загружена в ОП. 0 – не загружена.

Свойство локальности – использовании ассоциативного буфера на 8 адресов при страничном преобразовании дает 90% скорости полностью ассоциативной памяти благодаря свойству локальности. Бывает 2-х типов:

1. ВременнАя – процесс, обратившийся к 1 странице наиболее вероятно в след. един. времени обратится к этой же стр.
2. Пространственная – процесс, обратившийся к 1 странице наиболее вероятно обратится к соседним страницам.

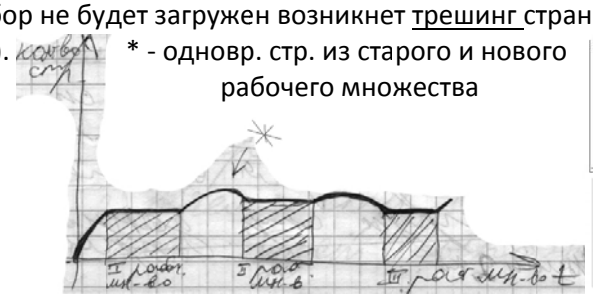
Ассоциативно-прямое отображение (снач. ищется в ассоциативной памяти, затем в физической). Около 90%-ассоц

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 одних и тех же страниц).

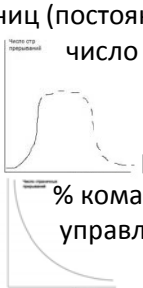


размер программы

w(t, dt) – набор страниц, к кот. активно обращ. процесс за dt



* - одновр. стр. из старого и нового рабочего множества



число стр. прерыв.

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

Unix

Прерывание таймера имеет второй приоритет (после прерывания по сбою питания).

Функции обработчика прерывания таймера:

- Инкремент счетчика таймера.
- Вызов процедуры обновления статистики использования процессора текущим процессом.
- Пробуждение в нужные моменты времени системных процессов (например, swapper и pagedaemon)
- Поддержка профилирования выполнения процессов в режимах ядра и задачи при помощи драйвера параметров.
- Вызов обработчиков отложенных вызовов.
- Декремент значения кванта процессорного времени.
- После истечения кванта процессорного времени:
    - Посылка текущему процессу сигнала SIGXCPU, если тот превысил выделенный ему квант процессорного времени.
    - Вызов функций планировщика (пересчет приоритетов).

Т.к. некоторые из задач не требуют выполнения на каждом тике, то вводится понятие **основного** тика (равен n тикам), часть задач выполняется только при основном тике.

Windows

В многопроцессорной системе каждый процессор получает прерывания системного таймера, но обновление значения системного таймера в результате обработки этого прерывания осуществляется только одним процессором. Однако все процессоры используют это прерывание для измерения кванта времени, выделенного потоку, и для вызова процедуры планирования по истечении этого кванта.

Функции обработчика прерывания таймера:

- Инкремент счетчика таймера.
- Вызов процедуры сбора статистики использования процессорного времени.
- Вызов обработчиков отложенных вызовов.
- Декремент значения кванта процессорного времени.
- После истечения кванта процессорного времени:
    - Посылка текущему потоку сигнала по превышении кванта процессорного времени.
    - Вызов функций, относящихся к работе диспетчера ядра (пересчет приоритетов).
    - Постановка DPC (Deferred Procedure Call – отложенный вызов процедуры) в очередь, чтобы инициировать диспетчеризацию потоков

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

Процесс - программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС). М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.

В распределенных системах процессы не имеют общей памяти (например, сети). Могут синхронизироваться только сообщениями. Взаимодействие по системе «клиент-сервер». Выдел. процессы-серверы, предоставляющие набор сервисов, и процессы-клиенты, пользующиеся предоставленными сервисами.

send <список значений> to <адресат>, где <список значений> – значения набора парам. в момент посылки

receive <список переменных> from <адресат> <адресат> – указывается адресат сообщения и отправитель

Считается, что модель «клиент-сервер» работает на уровне транзакций (условно неделимая последовательность действий – перерывы не приводят к изменению данных, обеспеч. целостность данных).

Простейший протокол обмена «клиент-сервер»:

1. Запрос – клиент запраш. сервер для обраб. запр.
2. Ответ – сервер возвращает результат операции.
3. Подтверждение – клиент подтверждает прием сообщения от сервера.

+ *дополнительные*:

RPC (remote procedure call) – вызов удаленной процедуры.

4. Клиент запрашивает: «Сервер доступен?».
5. Если сервер доступен, то посылает: «Я доступен».
6. Если сервер недоступен (занят), то может послать сообщение: «Перезвоните».
7. Адрес неверен. Процесс с зад. ID в сист. отсутств.

Проблема синхронизации

Процессам часто нужно взаимодействовать друг с другом, например, 1 процесс может передавать данные др. процессу, или несколько процессов могут обрабатывать данные из общего файла. В этих сл. возникает проблема синхронизации процессов. Она связана с потерей доступа к параметрам из-за их некорректного разделения.

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

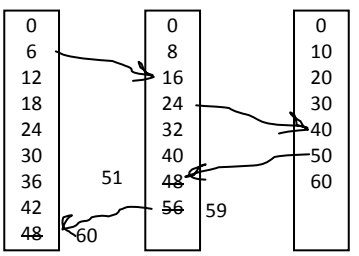
Критическая секция - строки кода, в кот. происходит обращение к критическому ресурсу.

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.

Алгоритм синхронизации логических часов (алг. Лампорта)

Для 2-х произв. событий вводятся понятия «случилось до», «случилось после». Время приема НЕ м.б. < времени посылки сообщения.

1. Каждому сообщ. припис. время отправки по локальным часам отправителя.
2. Получатель сравнивает это время со своим временем. Если собств. время < полученного, то собств. время устанавливается на > времени полученного.



Алгоритмы взаимоисключения в распределенных системах

1. Централизованный алгоритм (как на 1 машине). Процесс-координатор (м.б. выбран процесс с наиб. сетевым адресом) отвечает за возможность вхождения в критические секции. Как только процесс готов войти в критич. область, он посылает сообщение, где указывает ID крит. секции.Процесс сможет зайти в крит. секцию только после ответа координатора. Координатор проверяет, не находится ли к-л. др. процесс в крит. области (по полученным сообщениям). Если находится, то корд. не посылает ответ, а ставит запрос в очередь. После освобождения крит. области процессу б. выслан ответ-подтверждение. Если координатор аварийно завершился необходимо выбрать новый координатор. Если к-л. процесс обнаружит отсутств. корд. (время ожид ответа > кртитич. времени), то процесс инициирует выбор нового координатора. Самый эффект. алг.
2. Распределенный алгоритм. Процесс, желающий войти в критический участок, формирует запрос, содержащий ID критического участка, свой номер и время по локальным часам. Запрос посылается всем процессам в системе. Полагается, что передача сообщений является надежной. Когда др. процесс получ. сообщ., то:
   a. Если получат. не находится и не собирается входить в критический участок, то он посыл. ответ-разреш.
   b. Если получатель уже находится в критическом участке, то он не отвечает и ставит запрос в очередь.
   c. Если получатель желает войти в критическую секцию, то он сравнивает свое время и время в запросе. Если время в запросе меньше его собственного времени (собств. – позже) – (a), иначе (b).

   Процесс входит в критическую обл., если получ. n-1 разреш. (от всех, кроме себя). После выхода посылает разрешение всем процессам в очереди. Необходима посылка и получение n-1 сообщений.
3. Алгоритм Token Ring. Самый распространенный стандарт локальных сетей. Все процессы образуют логическое кольцо (направленное). Кажд. процесс знает № своей позиции и № ближайшего к нему процесса. При инициализ. Token (спец. сообщение) к 0, передается от процесса n-1 к n. Когда процесс получает Token, он смотрит, не требуется ли ему войти в критическую секцию. Если надо – входит (удерживая Token), если нет – посылает Token дальше. Если желающих нет, Token циркулирует по кольцу с большой скоростью. Кол-во сообщений – от 1 до ∞ (если ни 1 не входил).
4. Модифицированный Token Ring. Реализован IBM в 1984 году. Если процесс заинтересован в передаче данных, то при получении Tokena, Token изымается из кольца. Процесс посылает в кольцо свой Token, содержащий адрес источника и адрес получателя. Передача Tokena = копирование. При обнаружении в Tokene своего адреса, процесс копир. Token в свой буфер и вставл. в него свое подтверждение приема. Процесс, пославший Token и получивший подтверждение изымает Token и посылает др. Token, чтобы процессы могли обрабатывать свои данные

2. Спецификация XM (XMS): Conventional, HMA, UMA, EMA.

XMS (eXtended Memory Specification, спецификация расширенной памяти) – спецификация Microsoft на расширенную память (XMS 2.0), позволяющая DOS-программам с помощью диспетчера расширенной памяти (XMM) использовать расширенную память ПК на процессорах 80286 и более новых. *(добавленная, продленная)*
XMS оговаривает все вопросы, связанные с дополнительной памятью (сверх 1 Мб).
EMS (Expanded Memory Specification, спецификация отображаемой памяти) – стандарт, разработанный в 1985 г. фирмами Lotus, Intel и Microsoft для доступа из DOS к областям памяти выше 1 Мбайт в системах на базе процессоров 80386 и более поздних. *(расширенная, растягиваемая)*

- 0-640 Кб: основная память (conventional) – память, доступная DOS и программам реального режима.

Стандартная память является самой дефицитной в PC, когда речь идет о работе в среде ОС 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 используется для векторов прерываний, области переменных BIOS; области DOS; предоставляется пользователю (до 638 Кбайт).

- 640-1024 Кб: UMA (upper memory area) – обл. верхней памяти – зарезервирована для системных нужд

Размещаются обл. буферной памяти адаптеров (пр. – видеопамять) и постоянная память (BIOS с расширениями).

- С 1024 Кб –…(защ.) XMA (extended memory area) – непосредственно доступна только в защищенном режиме для компьютеров с процессорами 286 и выше.
- 1024-1088 Кб (Реал): HMA (high memory area) – верхняя область памяти (1-й сегмент размером 64 Кбайт, расположенный выше мегабайтной отметки памяти PC с операционной системой MS-DOS) Единственная область расширенной памяти, доступная 286+ в реальном режиме при открытом вентиле Gate A20.

XMS – программная спецификация использования дополнительной памяти DOS-программами для компьютеров на процессорах 286 и выше. 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 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 реальном режиме существует 2 вида адресного заворачивания.
В 8086 – 20 линий адреса, заворачивание на начало. 2^{20} = 1 Мб = FFFFF
Если в реальном режиме открыть линию A20 (21-ю линию), то в реальном режиме станет доступно дополнительно 64 Кб памяти. Линия A20 – для совсместимости (для заворачивания).
По умолчанию компьютер начинает работу в реальном режиме. Для обеспечения адресного заворачивания A20 сброшена – заземлена. Для перехода в защищенный режим надо открыть A20.

В реальном 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

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 некорректного разделения. В каждый момент времени на процессоре выполняется 1 процесс. Каждому процессу выделяется квант процесс. времени.

Разделяемый ресурс - переменная, к которой обращаются разные процессы.

Критическая секция – область кода, из которой осуществляется доступ к разделяемому ресурсу.

Монопольное использование – если процесс получил доступ к разделяемому ресурсу, то др. процесс не м. получить доступ. Необходимо обеспечить монопольный доступ процесса к разделяемому ресурсу до тех пор, пока процесс его не освободит. 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.

Программная реализация

```
flag: boolean;
p1:   while (1) do
            while(flag)do;
            flag = 1;
            CR1;
            flag = 0;
end P1;
main: flag = 0;
      parbegin
            P1;
            P2;
      parend;
```

```
p2:   while (1) do
            while(flag)do;
            flag = 1;
            CR2;
            flag = 0;
end P2;
```

Возможно бесконечное откладывание.

Возможно, выполн. while, но не измен. флага.

Программная реализация (алгоритм Деккера).

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

```
flag1, flag2: boolean;
queue: integer;
p1:   flag1 = 1;
      while(flag2)do
            if(queue == 2) then
            begin
                  flag1 = 0;
                  while(queue == 2)do;
                  flag1 = 1;
            end;
      CR1;
      flag1 = 0;
      queue = 2;
end P1;
main: flag1 = 0;
      flag2 = 0;
      queue = 1; // Или 2
      parbegin
            P1;
            P2;
      parend;
```

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

queue – очередь процесса входить в критическую секцию.

Недостаток – активное ожидание на процессоре.

Семафоры (ввел Дэйкстра в 1965 г.)

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

P (от датск. passeren - пропустить) и V (от датск. vrygeven - освободить). Защищенность семафора означает, что значение семафора м. изменяться только операциями P и V.

1. Операция P(S): S = S – 1. Декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить операцию P, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой процесс, выполняющий операцию V(S).
2. Операция V(S): S = S + 1. Инкремент S одним неделимым действием (последовательность непрерывных действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0, то V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожид. на семафоре.

```
S = 1;      producer:   P(S); // Занять
                        CR1;
                        V(S); // Освободить
```

```
consumer:   P(S); // Занять
            CR2;
            V(S); // Освободить
```

	Программно	семафоры
Активное ожидание	+	-
Бесконечное откладывание	+ (- для Деккера)	-
Возможность тупиков	+	+

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

Unix создавалась как ОС разделения времени.

Базовое понятие Unix – процесс (единица декомпозиции ОС, программа времени выполнения). Процессрассматривается как виртуальная машина с собств. адресным пространством, выполн. пользов. прогр., предоставл. набор услуг. Процесс может находиться в двух состояниях – «задача» (процесс выполняет собственный код) и «система» или «ядро»(выполняет реентерабельный код ОС). Процессы все время переходят «пользователь» ←→ «система». Unix – ОС с динамическим управлением процессами.

Любой процесс может создавать любое число процессов с помощью системного вызова fork() – ветвление. Процессы образуют иерархию в виде дерева процессов, процессы связ. отношением потомок-предок.

В результате вызова fork создается процесс-потомок, который является копией процесса-предка (наследует адресное пространство предка и дескрипторы всех открытых файлов (фактически наследует код)). В Unix все рассматривается как файл (файлы, директории, устройства).

fork() возвращает 0 – для потомка, -1 – если ветвление невозможно, для родителя возвращ. натуральное число (ID потомка). Любой процесс имеет предка, кроме демонов.

Все процессы имеют прародителя –Init с ID = 0 (порожд. в нач. работы и существует до окончания работы ОС).

Все ост. порожд. по унифиц. схеме с помощью сист. вызова fork().

Системный вызов exec(), заменяет адресное пространство потомка на адресное пространство программы, указанной в системном вызове. exec () НЕ создает новый процесс!!!

Системный вызов wait(&status) вызывается в теле предка. Предок б. ждать завершения всех своих потомков. При их завершении он получит статус завершения.

«Сирота» — процесс, родитель которого завершился раньше него. При завершении процессов ОС проверяет, не осталось ли у процессов незавершившихся потомков, если остались – принимает усилия по усыновлению – редактирует строку данного проц.-потомка, заменяя строку предка на 1 (усыновл. терминальным процессом).

«Зомби». Если процесс-потомок завершился до тех пор, пока предок успел вызвать wait (ghb аварийном завершении), то для того, чтобы предок не завис в ожидании несуществующего процесса, ОС отбирает все ресурсы ку процесса (оставляет только строку в таблице процессов) и помечает процесс-потомок как «зомби». Это делается для того, чтобы предок, дойдя до wait() смог получить статусы завершения всех своих потомков.

«Демон» — процесс, который не имеет предков. Срабатывает по определенному событию. «Демон» Unix примерно соответствует «сервису» Windows.

Зомби

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

Демон

```
int daemonize(void) {
switch (fork()) {
case 0:
return setsid();
case -1:
return -1;
default:
exit(0);}}
```

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

Повторно-используемые ресурсы – количество в системе постоянно и при использовании они не изменяются (или редко): аппаратура (ОП, ЦП), реентерабельные коды, системные таблицы (изменения в них могут вноситься только супервизором), процедуры ОС (так как они являются реентерабельными).

Потребляемые ресурсы – количество в ОС переменно и произвольно: сообщения. Процесс может создать любое количество сообщений. Процесс получения сообщения заканчивается его уничтожением.

Тупик – ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другими процессами, ожидающими освобождение ресурса, занятого 1-м процессом.

Условия возникновения тупика в системе (необходим. и достаточн.):

1) Усл. взаимоисключения (процессы требуют предоставления права монопольного использования ресурсов).
2) Усл. ожидания ресурса (процесс удерживает занимаемые им ресурсы и ожидает выделения доп. ресурсов).
3) Усл. неперераспределяемости (ресурсы нельзя отобрать у процесса, их использующего, – только вернет сам).
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).

Методы борьбы с тупиками: предотвращение (исключение), обход (недопущение), обнаружение и восстановл.

Обход или недопущение тупиков связаны с анализом запросов. Предполаг., что тупик потенциально возможен, но в ОС создаются такие условия, при которых тупик становится невозможен. Очевидно, что анализ ситуации связан с анализом запросов ресурсов процессами.

Алгоритм Банкира (метод обхода, авт. – Дейкстра)

В качестве банкира выступает менеджер ресурсов. Процессы указывают в своих заявках, макс. потребность в ресурсах данного типа. Процесс не м. затребовать больше ресурсов, чем указано в его заявке. Условия:

1) число процессов в ОС фиксировано
2) число ресурсов в ОС фиксировано
3) процесс не м. запросить > ресурсов, чем есть в ОС
4) процесс не м. запрос. > ресурсов, чем в его заявке
5) всех распределенных ресурсов данного класса не м.б. > ∑ ресурсов данного класса в ОС.

Менеджер гарантирует, что не возникнет тупик. Алгоритм:

1) Каждый запрос проверяется на отношение к количеству ресурсов в системе
2) Каждая заявка проверяется относительно суммы всех заявок на ресурсы
3) Менеджер ресурсов при получении очередной заявки в ОС ищет такую последовательность процессов, кот. м. гарантированно завершиться.

Пример

Менеджер решает: p2, p1, p3

Текущее состояние системы является надёжным (безопасным относительно тупика), так как имеется послед. процессов, которая может успешно завершиться.

Процессы	Текущее распределение	Свободные единицы ресурсов	Заявка
P1	1		4
P2	3		5
P3	5	2	9

Процессы	Текущее распределение	Свободные единицы ресурсов	Заявка
P1	2		4
P2	3		5
P3	5	1	9

Ненадежное состояние (нельзя гарантировать успешность завершения процессов). Ненадежное состояние м. не привести к тупику (процесс м. не запросить указ. в заявке кол-во ресурсов.

Состояние безопасное, если:

- Существует процесс, кот. завершится, т.к. если он запросит макс. кол-во единиц ресурса, у ОС найдется необх. кол-во свободных единиц.
- Существует процесс, которому хватит всех единиц от освободившегося 1-го + свободные единицы…
- …Существует i-й процесс, все i-1 процессы освободили все ресурсы, всех свободных ресурсов хватит при макс. запросе i-го процесса.

Недостатки: 1, 2 усл.; процесс д. подавать заявки на макс. кол-во ресурсов (процессы не знают).; много вычисл.

Т.о., всякий раз, когда процесс выполнил запрос, менеджер ресурсов должен найти последовательность процессов, которые гарантировано завершатся, и только в этом случае запрос может быть удовлетворён.

Необходимо исследовать **n!** последовательностей прежде чем признать состояние системы безопасным. Алгоритм банкира остаётся теоретической основой для разрешения ситуаций, когда необходим анализ запросов.

Пример аппроксимации алг. банкира – Алг. $O(n^2)$; S – последовательность процессов.

```
while (S <> []) do
      if not (find (процесс A из S, кот. м. завершиться)) then
            состояние – небезопасное, вывести A из S, отобрать у A ресурсы и добавить их
в пул нераспределенных ресурсов.
состояние – безопасное.
```

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

Поток – часть последовательного кода процесса, которая может выполняться || с другими частями кода.

```
HANDLE CreateThread(
      LPSECURITY_ATTRIBUTES lpThreadAttributes, // дескриптор защиты
      SIZE_T dwStackSize,                       // начальный размер стека
      LPTHREAD_START_ROUTINE lpStartAddress,    // функция потока
      LPVOID lpParameter,                       // параметр потока
      DWORD dwCreationFlags,                    // опции создания
      LPDWORD lpThreadId                        // идентификатор потока
);
```

Если функция завершается успешно, величина возвращаемого значения – дескриптор нового потока. Если функция завершается с ошибкой, величина возвращаемого значения - ПУСТО (NULL).

Поток создается с приоритетом потока THREAD_PRIORITY_NORMAL.

Когда поток заканчивает работу, объект потока приобретает сигнального состояния, удовлетворяя любые потоки, которые ждали объект. Объект потока остается в системе, до тех пор, пока не поток закончит работу, и все дескрипторы к нему не будут закрыты через вызов CloseHandle.

```
DWORD WINAPI WaitForSingleObject(
      __in  HANDLE hHandle,        // дескриптор объекта
      __in  DWORD dwMilliseconds   // время ожидания
);
```

Ожидает и возвращают значение тогда, когда происходит одно из ниже перечисленного:

- Указанный объект находится в сигнальном состояние.
- Интервал времени простоя истекает. Интервал времени простоя может быть установлен в INFINITE (БЕСКОНЕЧНО), чтобы определить, что ожидание будет непрерывным.

Возвращает событие, заставившее функцию завершиться:

- WAIT_ABANDONED - поток, владевший объектом, завершился, не переведя объект в сигнальн. состояние
- WAIT_OBJECT_0 - Объект перешел в сигнальное состояние
- WAIT_TIMEOUT - Истек срок ожидания
- WAIT_FAILED - Произошла ошибка (например, получено неверное значение hHandle)

```
DWORD WINAPI WaitForMultipleObjects(
 __in  DWORD nCount,                  // количество дескрипторов
 __in  const HANDLE *lpHandles,       // массив дескрипторов
 __in  BOOL bWaitAll,                 // ждать всех (true) или хотя бы одного (false)
 __in  DWORD dwMilliseconds           // время ожидания
);
```

Возвращаемые значения те же, за исключением одного: WAIT_OBJECT_N, где N – объект, заставивший функцию завершиться.

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.

Мьютексы, семафоры и события позволяют синхронизировать потоки, используемые в разных процессах, чтоб одно приложение могло уведомить другое об окончании той или иной операции.

Объекты ядра мьютксы гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Мьютексы ведут точно также как критические секции, однако они являются объектами пользовательского режима. Недостаток – невозможность синхронизации потоков разных процессов.

Функции API: CreateMutex() — создать мьютекс; WaitForSingleObject() — ждать освобождения мьютекса, eventa или семафора, ReleaseMutex() — освободить мьютекс.

Пример.

```
HANDLE Writing, CanRead, Writer;
CanRead = CreateEvent(NULL, false, false, NULL);      Writing = CreateMutex(NULL, false, NULL);
WaitForSingleObject(Writing, INFINITE);
ReleaseMutex(Writing);                                SetEvent(CanRead);
Writer = CreateThread(NULL, NULL, Procedure_name, Param, NULL, NULL);
```

19

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

Процесс - программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).

Асинхронные процессы – процессы, каждый из которых выполняется с собственной скоростью. Все процессы в системе являются асинхронными, то есть нельзя сказать, когда процесс придёт в какую-то точку.

1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критичские секции - ок
2. Бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захв. 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. Существуют только 3 схемы действия философов:

1. пытается взять обе вилки сразу. Если удается, он может есть. → Бесконечное откладывание
2. берет левую вилку и пытается взять правую (левую держит в руке) → Тупик
3. берет левую вилку, и если не удается взять правую, то кладет левую вилку обратно → Ok

Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции: P (от датск. passeren - пропустить) и V (от датск. vrygeven - освободить). Защищенность семафора означает, что значение семафора м. изменяться только операциями P и V.

1. Операция P(S): S = S – 1. Декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить операцию P, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой процесс, выполняющий операцию V(S).
2. Операция V(S): S = S + 1. Инкремент S одним неделимым действием (последовательность непрерывных действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0, то V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожид. на семафоре.

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 поддерживаются ОС-ой. Семафоры устраняют активное ожидание на процессоре.

Семафоры бывают: бинарные (S принимает значения 0 и 1), считающие (S принимает значения от 0 до n), множественные (набор считающих семафоров). Все семафоры 1 набора м. устанавливаться 1 операцией.

В Windows после освобождения на семафоре приоритет процесса повыш. Изменение S м. рассматривать как событие в ОС.

Пример: *Произвдство/потребление:*

```
Se (пустые ячейки), Sf(полные ячейки), S: int;
producer:   while (1) do                          consumer:   while (1) do
                  P(Se);// Se—                                      P(Sf);// Sf—
                  P(S); // Занять                                   P(S); // Занять
                  N++                                               N--
                  V(S); // Освободить                              V(S); // Освободить
                  V(Sf);// Sf++                                     V(Se);// Se++
begin: Se = N;    Sf = 0;           S = 1;        …
```

Использование семафоров часто приводит к взаимоблокировке. Понятие «монитор» предложил Хоар.

Монитор – языковая конструкция, состоящая из структур данных и подпрограмм, использующих данные структуры. Монитор защищает данные. Доступ к данным монитора могут получить только п/п монитора. В каждый момент времени в мониторе м. находиться только 1 процесс. Монитор является ресурсом.

Процесс, захвативший монитор, – процесс **в** мониторе, процесс, ожидающий в очереди, – процесс **на** мониторе.

Используется переменная типа «событие» для каждой причины перевода процесса в состояние блокировки.

wait – откр. доступ к монитору, задержив. выполнение процесса. Оно д.б. восстановлено операцией signal др. процесса.

Если очередь перем. к типу усл. <> 0, то из очереди выбирается 1 из процессов и инициализируется его выполн.

Монитор «кольцевой буфер» – решает задачу «производство-потребление», 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).

```
resource: monitor circle_buffer;
const n = size;
var   buffer: array [0..n-111]]] of <type>;
      pos, write_pos, read_pos: 0..n-1;
      full, empty: conditional;
procedure producer(p: process, data:<
type>)
begin
      if (pos = n) then
            wait(empty);
      buffer[write_pos] := data;
      Inc(pos);
      write_pos = (write_pos + 1) mod n;
      signal(full);
end;
begin       pos := 0;   write_pos := 0;   read_pos := 0;    end.
```

```
procedure consumer(p: process, var data:
<type>)
begin
      if (pos = 0) then
            wait(full);
      data := buffer[read_pos];
      Dec(read_pos);
      read_pos := (read_pos + 1) mod n;
      signal(empty);
end
```

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

```
#define Se 0
#define Sf 1
#define S 2
#define size 7

int semathores;. // Free cells count, full cells count, binary semaphore (set of semathores)
unsigned short initialValues[3] = {size, 0, 1};

.       // Semathores operations
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 *buffer;
int semID, shmID, procID;
void product()
{
.       if (semop(semID, &PP[0], 2) == -1)
.       {
.       .       perror("semop");
.       .       exit(3);
.       }
.       int ind = buffer[size];
.       buffer[ind] = ind+1;
.       printf("Production: %d\n", buffer[ind]);
.       buffer[size]++;
.       if (semop(semID, &VP[0], 2) == -1)
.       {
.       .       perror("semop");
.       .       exit(4);
.       }
}

void consumpt()
{
.       if (semop(semID, &PC[0], 2) == -1)
.       {
.       .       perror("semop");
.       .       exit(5);
.       }
.       buffer[size]--;
.       int ind = buffer[size];
.       printf("Consumption: %d\n", buffer[ind]);
.       if (semop(semID, &VC[0], 2) == -1)
.       {
.       .       perror("semop");
.       .       exit(6);
.       }
}

int main()
{
.       int perms = S_IRWXU | S_IRWXG | S_IRWXO;

.       ((semID = semget(100, 3, IPC_CREAT | perms)) == -1)    ...
.       semctl(semID, 0, SETALL, &initialValues);
.
.       ((shmID = shmget(100, size+1, IPC_CREAT | perms)) == -1)
.       buffer = (char*)shmat(shmID, 0, 0);
.       buffer[size] = 0;
.       if ((procID = fork()) == 0)
```

20

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 ОС (именно ему выделяются ресурсы ОС).
М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.

Диаграмма состояний процесса

Порождение – присв. процессу строки в таблице процессов
Готовность – попадание в очередь готовых процессов – получили все необходимые ресурсы. **Выполнение**
Блокировка (Ожидание) – ожидание необходимого ресурса. Если процесс интерактивный, то он постоянно блокируется в ожидании ввода/вывода.
(схема для мультипрограммной пакетной обработки)
Прерывание = истек квант времени.

Аппаратный контекст процесса – сост. регистров.
Полный контекст процесса – сост. регистров + сост. памяти.

Планирование – управл. распредел. ресурсов ЦП между разл. конкур. процессами путем передачи им управления согласно некот. стратегии планирования.
Диспетчеризация – выделение процессу процессорного времени.

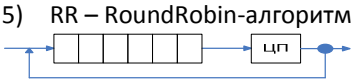
Контекст выполнения – Функции ядра могут выполняться в контексте процесса, либо в системном контексте. При выполнении в контексте процесса ядро функционирует от имени текущего процесса, имея доступ к данным тек. процесса, стеку ядра, адресному пространству процесса (всё это – user area). Ядро м. заблокировать процесс.
При выполнении в контексте ядра, оно обслуживает прерывание от внешних устройств и выполняет пересчёт приор. процессов. Всё это выполн. в сист. контексте (контексте прерываний). Переключ. – мультизадачность.
Процесс может быть разделён на параллельно выполняющиеся потоки (threads).
Программа-планировщик – программа, отвечающая за управление использованием совместного ресурса.
Последовательное выполнение – Квазипараллельное выполнение – Реальная параллельность

- 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: абсолютные и относительные, статические и динамические}
- С вытеснением / без вытеснения

Бесконечное откладывание – ситуация, когда процесс никогда не получает необх. для выполнения ресурсов (точнее, кванта времени). Возникает, когда диспетчер всегда отдаёт квант др. процессу, т.к. его приоритет >.

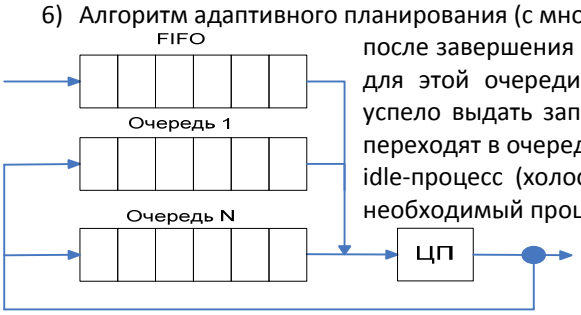
Алгоритмы планирования *(1–4 – для ОС пакетной обработки, 5– –для др. ОС):*

1) FIFO – простая очередь. Без переключения, без приоритетов.
2) SJF – Shortest Job First – наикратчайшее задание – первое. М.б. бесконечное откладывание. Д.б. априорная инф. о времени выполнения программы, объеме памяти. Статич. приор., без переключ.
3) SRT – Shortest Remaining Time – с вытеснением. Выполняющийся процесс прерывается, если в очереди появляется процесс с меньшим временем выполнения (меньше оставшегося до заверш. времени).
4) HRR – Highest Response Ratio (наиболее высокое относительное время ответа) – с динамич. приоритетами. Используется в UNIX. $priority = \frac{t_W - t_S}{t_S}, t_W$ – время ожидания, t_S - запрошенное время обслуживания. Чем больше ожидает, тем больше приоритет. Позволяет избежать бесконечного откладывания.
5) RR – RoundRobin-алгоритм (циклическое планирование) – с переключ., без вытеснения, без приоритетов



Разл. процессы м. иметь разл. величину кванта.

6) Алгоритм адаптивного планирования (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 максимальное количество процессов успело выдать запрос IO или завершиться. Процессы, не сделавшие ни того, ни другого, переходят в очередь с более низким приоритетом по алгоритму RR. В очереди N «крутится» idle-процесс (холостой) и все вычислит. процессы. Также м. учитываться объем памяти, необходимый процессу – адаптивно-рефлексивное планир., т.е. выделяется очередной квант только при наличии свободной памяти в ОС.



В современных системах: Процессы создаются по мере необходимости, ресурсы выделяются по мере надобности. Априорная информация о времени выполнения процессов отсутствует.

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.

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

```
#include <windows.h>
volatile LONG ActReadersCount = 0, ReadersCount = 0,
        WritersCount = 0;
volatile int Value = 0;
HANDLE CanRead, CanWrite, Writing;
void StartRead()
{       InterlockedIncrement(&ReadersCount);
        WaitForSingleObject(Writing, INFINITE);
        ReleaseMutex(Writing);
        if (WritersCount)
            WaitForSingleObject(CanRead, INFINITE);
        InterlockedDecrement(&ReadersCount);
        InterlockedIncrement(&ActReadersCount);
        if (ReadersCount)
                SetEvent(CanRead);                      }
void StopRead()
{       InterlockedDecrement(&ActReadersCount);
        if (!ReadersCount)
                SetEvent(CanWrite);                     }
void StartWrite()
{       InterlockedIncrement(&WritersCount);
        if (ActReadersCount)
            WaitForSingleObject(CanWrite, INFINITE);
        WaitForSingleObject(Writing, INFINITE);
        InterlockedDecrement(&WritersCount);            }
void StopWrite()
{       ReleaseMutex(Writing);
        if (ReadersCount)
                SetEvent(CanRead);
        else
                SetEvent(CanWrite);                     }
DWORD WINAPI Reader(PVOID pvParam)
{       for (;;)
        {       StartRead();
                Sleep(rand()/100.);
                printf("Reader %d: %d\n", (int)pvParam,
                        Value);
                StopRead();
                Sleep(rand()/10.);      }
        return 0;       }
DWORD WINAPI Writer(PVOID pvParam)
{       for (;;)
        {       StartWrite();
                Sleep(rand()/75.);
                printf("Writer %d: %d\n", (int)pvParam,
                        ++Value);
                StopWrite();
                Sleep(rand()/7.);       }
        return 0;       }
int main()
{       HANDLE Writers[3], Readers[3];
      CanRead = CreateEvent(NULL, false, false, NULL);
      CanWrite = CreateEvent(NULL, false, true, NULL);
        Writing = CreateMutex(NULL, false, NULL);
        for (int i = 0; i < 3; i++)
                Writers[i] = CreateThread(NULL, NULL,
                        Writer, (LPVOID)i, NULL, NULL);
        for (int j = 0; j < 3; j++)
                Readers[j] = CreateThread(NULL, NULL,
                        Reader, (LPVOID)j, NULL, NULL);
        getch();
        return 0;       }
```

Мьютекс (mutex – *mutually exclusive* (взаимно исключающий)**)** - механизм синхронизации, используемый для упорядоч. доступа к ресурсам. Этот объект обеспечивает исключительный *(exclusive)* доступ к охраняемому блоку кодов. Также мьютекс позволяет установить время блокировки ресурса. Мьютекс предоставляет доступ к объекту любому из потоков, если в данный момент объект не занят, и запоминает текущее состояние объекта. Если объект занят, то мьютекс запрещает доступ. Однако можно подождать освобождения объекта с помощью функции WaitForSingleObject, в которой роль управляющего объекта выполняет тот же мьютекс.

```
HANDLE CreateMutex
(       LPSECURITY_ATTRIBUTES lpMutexAttributes,      // атрибут безопастности
        BOOL bInitialOwner,                    // флаг начального владельца
        LPCTSTR lpName          );             // имя объекта
```

>= 2 процессов м. создать мьютекс с одним и тем же именем, вызвав метод CreateMutex . Первый процесс действительно создает мьютекс, а следующие процессы получают хэндл уже существующего объекта. Это дает возможность нескольким процессам получить хэндл одного и того же мьютекса.

```
BOOL ReleaseMutex
(       HANDLE hMutex           ); // дескриптор mutex
```

Освобождает объект. При успешном выполнении возвращает TRUE.

Событие (event) - это объект, выполняющий роль переключателя. У него есть только два состояния: вкл и выкл. Событие создается и помещается в коде соответствующего потока, где ведется наблюдение за состоянием объекта. Если объект события выключен, ждущие его потоки блокируются.

```
HANDLE CreateEvent
(       LPSECURITY_ATTRIBUTES lpEventAttributes,      // атрибут защиты
        BOOL bManualReset,                            // тип сброса TRUE - ручной
        BOOL bInitialState,                  // начальное состояние TRUE - сигнальное
        LPCTSTR lpName          );           // имя обьекта
```

SetEvent() меняет сост. на сигнальное (есть событие). **ResetEvent()** меняет сост. на невыделенное (нет события).

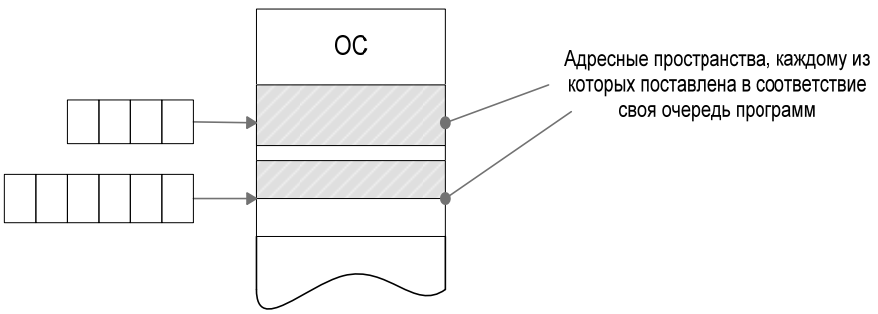
1 Управление памятью: выделение памяти разделами фиксированного размера, выделение памяти разделами переменного размера, стратегии выделения памяти, фрагментация памяти.

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.

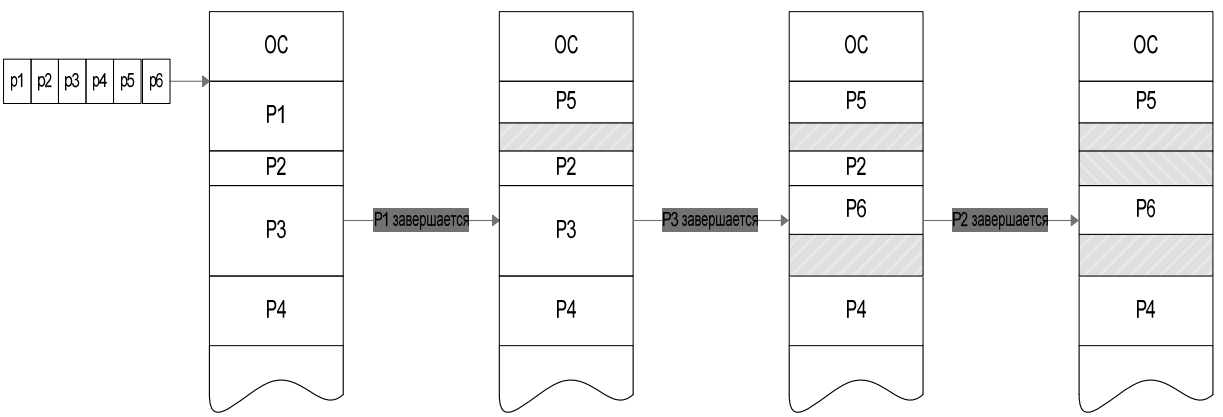
Распределение памяти разделами фиксированного размера. Существует 3 типа разделов небольшой, средний и большой. Размер определялся до работы системы и не менялся.

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 int 8h (реальный режим) - функции. Задачи прерывания по таймеру в защищенном режиме.

3 основные функции таймера в реальном режиме:

1. инкремент счётчика времени – тиков – в области данных BIOS.
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.

Unix

Прерывание таймера имеет второй приоритет (после прерывания по сбою питания).

Функции обработчика прерывания таймера:

- Инкремент счетчика таймера.
- Вызов процедуры обновления статистики использования процессора текущим процессом.
- Пробуждение в нужные моменты времени системных процессов (например, swapper и pagedaemon)
- Поддержка профилирования выполнения процессов в режимах ядра и задачи при помощи драйвера параметров.
- Вызов обработчиков отложенных вызовов.
- Декремент значения кванта процессорного времени.
- После истечения кванта процессорного времени:
    - Посылка текущему процессу сигнала SIGXCPU, если тот превысил выделенный ему квант процессорного времени.
    - Вызов функций планировщика (пересчет приоритетов).

Т.к. некоторые из задач не требуют выполнения на каждом тике, то вводится понятие **основного** тика (равен n тикам), часть задач выполняется только при основном тике.

Windows

В многопроцессорной системе каждый процессор получает прерывания системного таймера, но обновление значения системного таймера в результате обработки этого прерывания осуществляется только одним процессором. Однако все процессоры используют это прерывание для измерения кванта времени, выделенного потоку, и для вызова процедуры планирования по истечении этого кванта.

Функции обработчика прерывания таймера:

- Инкремент счетчика таймера.
- Вызов процедуры сбора статистики использования процессорного времени.
- Вызов обработчиков отложенных вызовов.
- Декремент значения кванта процессорного времени.
- После истечения кванта процессорного времени:
    - Посылка текущему потоку сигнала по превышении кванта процессорного времени.
    - Вызов функций, относящихся к работе диспетчера ядра (пересчет приоритетов).
    - Постановка DPC (Deferred Procedure Call – отложенный вызов процедуры) в очередь, чтобы инициировать диспетчеризацию потоков

Обработчики прерываний таймера в системах UNIX и WINDOWS практически идентичны. Это объясняется тем, что и UNIX и WINDOWS являются операционными системами разделения времени с динамическими приоритетами.

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

Условия возникновения тупика в системе:

1) Условие взаимоисключения (процессы треб. 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.

Представление графов:

1. Матрица
2. Связный список

В простейшем варианте двудольный (бихроматический) граф м.б. описан 2 матрицами:

1. матрица запросов (отраж. запросов проц.) – A = {p, r}
2. матрица распредел. (кол-во рес, выдел. к-л. проц.) – B = {r, p}

	запросы		распредел.	i = процесс, j = ресурс
A =		B =		Как табл, так и списки д.б. моноп. использ.
	a_{ij}		b_{ij}	

Восстановление тупика. Система д. поддерживать ср-ва приостановки возобновления. Последов. прекращ. процессов, попавших в тупик, в опред. порядке, пока пока не б. достаточно ресурсов для устранения тупика.

1. М. отбирать ресурсы у всех процессов
2. М. отбирать ресурсы только у процессов, попавших в тупик.

Минимизация м.б. осущ. на приор. проц:

- на основе цены повт. запуска проц. от текущей точки
- на основе внешней цены

Д.б. корректное освобождение ресурса – процесс д. вернуться к состоянию (точке), предшеств. запросу на данный ресурс (перед кажд. запросом необх. сохр. сост. процесса) – откат.

2. Три режима работы процессора Intel (486 ,…): защищенный режим, перевод 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.

Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес – сегмент/смещение). Минимальная адресная единица памяти – байт.

2^{20} = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).

Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

1-проц. режим под управлением MS-DOS (главное – минимизация памяти, занимаемой ОС => нет многозадачности).

0-256 б.: таблица векторов прерываний

резидентная часть DOS место резидентных программ сегменты программы

pool или heap (куча) транзитная часть DOS Резидентная часть DOS 1 Мб

2. Защищенный режим – многопроцессный режим. В памяти компьютера одновременно находится большое число программ с квантованием процессорного времени с виртуальной памятью. Управляет защищенным режимом ОС с разделением времени (Windows, Linux). В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.

Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Формирование таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

3. Специальный режим защищенного режима (V86) – 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.

VMM (Virtual Machine Manager) – для запуска виртуальных машин реального режима.

Переключение в защищенный режим:

1. Открыть адресную линию A20.
2. Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, она в нулевом кольце защиты).
3. Подготовить в оперативной памяти таблицу дескрипторов прерываний IDT.
4. Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по определенному адресу, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по этому адресу.
5. Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
6. Запретить все маскируемые и немаскируемые прерывания. Сохранить маски прерываний. Перепрограммировать контроллер прерываний.
7. Загрузить регистр GDTR и IDTR.
8. ***Перейти в защищенный режим*** (установить бит PE — нулевой бит в управляющем регистре CR0 в 1).
9. Загрузить новый селектор в регистр CS.
10. Загрузить сегментные регистры селекторами на соответствующие дескрипторы.
11. Разрешить прерывания.

Возвращение в реальный режим:

1. **Переключиться в реальный режим**
2. Сбросить очередь предвыборки, загрузить CS реальным сегментным адресом
3. Задать регистры для работы в реальном режиме
4. Загрузка IDTR (Interrupt Descriptor Table Register) для реального режима
5. Разрешаем немаскируемые прерывания
6. Разрешить маскируемые прерывания

1. Методы управления виртуальной памятью, особенности, сравнение – достоинства и недостатки.

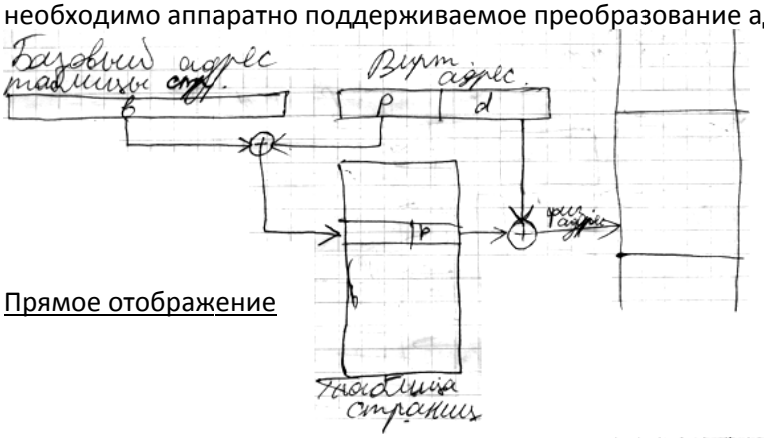
Virtual Memory – система при которой рабочее пространство процесса частично располагается в основной памяти и частично во вторичной. При обращении к какой либо памяти, система аппаратными средствами определяет, присутствует ли область физической памяти, если отсутствует, то генерируется прерывание, это 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 пейджинга, т.е. для временного хранения областей памяти.

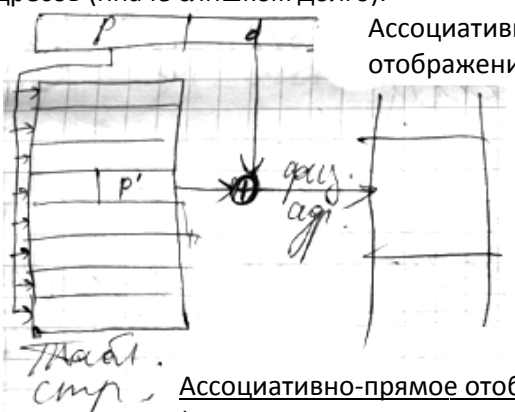
Подходы к реализации управления виртуальной памятью:

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

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 кот. 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).



Прямое отображение



Ассоциативное отображение

Ассоциативно-прямое отображение (снач. ищется в ассоциативной памяти, затем в физической). Около 90%-ассоц

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.

В системе с сегментно-страничной организацией примен. трехкомпонен. (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, по которому находится нужный элемент.

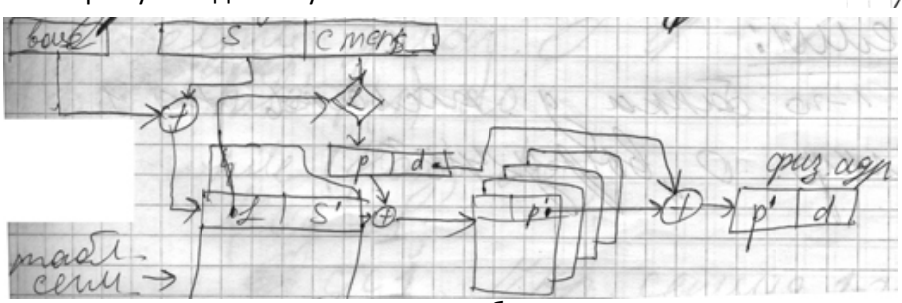


табл. страниц

+ страничного: легко реализовать, алгоритм LRU в этом достаточно эффективен

– страничного: сложность коллективного использования.

+ сегментного: легко реализовать коллективное использование, т.к. сегмент является лог. 1 деления памяти

– сегментного: необх. корректировки таблицы дескрипторов всех процессов при изменении размеров сегментов

– сегментного: сложн. при загрузке новых сегментов (в памяти д. сущ. адр пространство необходимого размера).

– сегментного: Фрагментация (Интенсивная загрузка, и выгрузка может привести к маленьким участкам, в которые загрузить ничего не удастся), хотя система может устранить её путём переноса сегментов

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

3 основные функции таймера в реальном режиме:

1. инкремент счётчика времени – тиков – в области данных BIOS.
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.

Unix Прерывание таймера имеет второй приоритет (после прерывания по сбою питания).

- Инкремент счетчика таймера.
- Вызов процедуры обновления статистики использования процессора текущим процессом.
- Пробуждение в нужные моменты времени системных процессов (например, swapper и pagedaemon)
- Поддержка профилирования выполнения процессов в режимах ядра и задачи при помощи драйвера параметров.
- Вызов обработчиков отложенных вызовов.
- Декремент значения кванта процессорного времени.
- После истечения кванта процессорного времени:
    - Посылка текущему процессу сигнала SIGXCPU, если тот превысил выделенный ему квант процессорного времени.
    - Вызов функций планировщика (пересчет приоритетов).

Т.к. некоторые из задач не требуют выполнения на каждом тике, то вводится понятие **основного** тика (равен n тикам), часть задач выполняется только при основном тике.

Windows В многопроцессорной системе каждый процессор получает прерывания системного таймера, но обновление значения системного таймера в результате обработки этого прерывания осуществляется только одним процессором. Однако все процессоры используют это прерывание для измерения кванта времени, выделенного потоку, и для вызова процедуры планирования по истечении этого кванта.

- Инкремент счетчика таймера.
- Вызов процедуры сбора статистики использования процессорного времени.
- Вызов обработчиков отложенных вызовов.
- Декремент значения кванта процессорного времени.
- После истечения кванта процессорного времени:
    - Посылка текущему потоку сигнала по превышении кванта процессорного времени.
    - Вызов функций, относящихся к работе диспетчера ядра (пересчет приоритетов).
    - Постановка DPC (Deferred Procedure Call – отложенный вызов процедуры) в очередь, чтобы инициировать диспетчеризацию потоков

Обработчики прерываний таймера в системах UNIX и WINDOWS практически идентичны. Это объясняется тем, что и UNIX и WINDOWS являются операционными системами разделения времени с динамическими приоритетами.

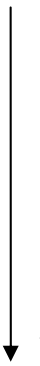
1. Ядро ОС: многопоточное ядро; взаимоисключение в ядре – спин - блокировки.

Системные процессы (выполняющиеся в режиме ядра) также нуждаются в механизме взаимоисключения. Критическими секциями ядра являются разделы кода ядра, в которых, например, модифицируются данные диспетчера ядра или DPC ((Deferred Procedure Call) отложенный вызов процедуры). Наибольшую проблему представляют прерывания, так как они могут возникнуть в любой момент, например в момент обновления БД ядра, при этом обработчик прерывания может также модифицировать ту же самую БД ядра.

DPC – управляющий объект ядра, который описывает запрос и позволяет отложить обработку запроса до повышения IRQL до уровня DPC/dispatch (диспетчеризации) (на примере Windows)

Пользоваться запретом прерываний нельзя.

В ядре выполняется большое число потоков. В частности, каждый драйвер представляет отдельный поток. Необходимо корректно решать задачу взаимоисключения.



P1	P2
Получить указатель на конец очереди. Записать данные в текущую поз.	
	Получить указатель на конец очереди. Записать данные в текущую поз.
Обновить указатель на конец очереди	
	Обновить указатель на конец очереди

Не урегулирован вопрос монопольного использования указателя на конец очереди.

В однопроцессорных и многопроцессорных системах проблема монопольного использования решается взаимоисключением. Механизм “spin lock”

В ядре возможно только постоянно в цикле проверять переменные блокировки (активное ожидание). Другие способы не возможны.

SpinLock

В основе лежит команда test_and_set. Простейшая форма механизмов в ядре базируется на аппаратной реализации. Существуют еще более простые механизмы: InterLockedIncrement, InterLockedDecriment, InterLockedExchange. При реализации этих команд блокируется многопроцессорная шина на время операции, чтобы другой процесс не мог выполнить команды. Основная проблема в ядре – прерывания. Но запрет прерываний не является хорошим выходом. В Win2000 запрещены только некоторые прерывания, обработчик которых использует этот же ресурс.

Пусть код работает на IRQL_Passive. Этот код захватил spin-блокировку. После этого код прерван кодом с уровнем IRQL_Dispatch, который пытается захватить ту же spin-блокировку. Для этого он входит в бесконечный цикл ожидания, в котором проверяет значение переменной и блокирует все действия в системе. Этот цикл таким образом никогда не закончится.

Для корректной работы, необходимо обеспечить выполнение определенных действий при захвате спин-блокировки: не позволить коду с более высоким уровнем IRQL прерывать код с более низким – повышать текущий уровень IRQL в момент захвата спин-блокировки до некоторого уровня, который связан с этой спин-блокировкой.

При освобождении уровень понижается до прежнего значения. Система предоставляет следующие команды:

1) KeInitializeSpinLock(IN_PKSPIN_LOCK SpinLock) – создает соответствующую спин-блокировку, то есть выделение памяти в невыгружаемой памяти.

2) KeAcquireSpinLock(IN_PKSPIN_LOCK SpinLock, OUT_PKIRQL OldIRQL) – уровень IRQL повышается до dispatch. Второй параметр связан с восстановлением IRQL до прежнего уровня

3) KeRealeaseSpinLock(IN_PKSPIN_LOCK SpinLock, OUT_PPIRQL NewIRQL)

4) KeAcquireLockAtGpcLevel (IN_PKSPIN_LOCK SpinLock) – автоматический переход на соответствующий уровень.

На одноплатформенной fh[/ функция ничего не делает.

5) KeRealeaseLockFromGpcLevel(IN_PKSPIN_LOCK SpinLock)

Простая spin-блокировка →

1й поток захватывает 1ю спин-блокировку

2й – вторую

Первый поток пытается захватить первую

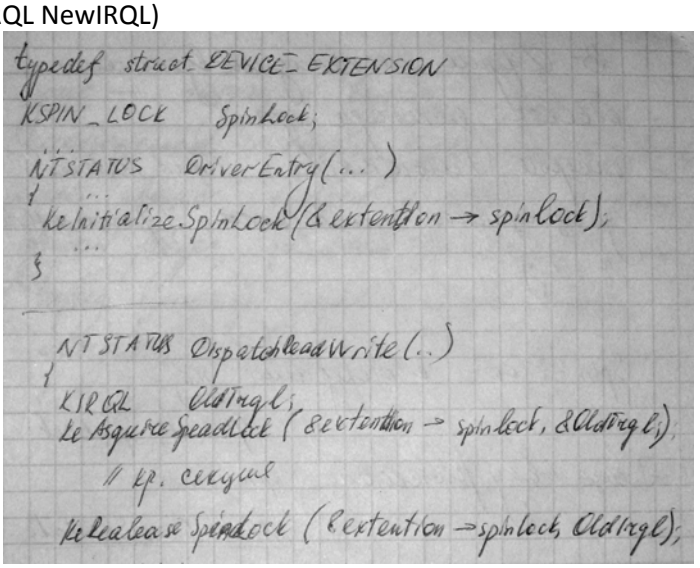
Второй – вторую

Потоки блокируют друг друга - ТУПИК!

Для разрешения проблемы используется очередь спин-блокировок (FIFO) – нумерация. QuenedSpinLock.

Существуют еще внутрестековые InStackQuenedSpinLock.

Другие механизмы взаимоисключения: объекты диспетчера ядра, быстрые мьютиксы, защищенные мьютиксы



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

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

Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес – сегмент/смещение). Минимальная адресная единица памяти – байт.

2^{20} = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).

Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

1-проц. режим под управлением MS-DOS (главное – минимизация памяти, занимаемой ОС => нет многозадачности).

0-256 б.: таблица векторов прерываний

резидентная часть DOS место резидентных программ сегменты программы

pool или heap (куча) транзитная часть DOS Резидентная часть DOS 1 Мб

2. Защищенный режим – многопроцессный режим. В памяти компьютера одновременно находится большое число программ с квантованием процессорного времени с виртуальной памятью. Управляет защищенным режимом ОС с разделением времени (Windows, Linux). В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.

Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Формирование таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

3. Специальный режим защищенного режима (V86) – 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.

VMM (Virtual Machine Manager) – для запуска виртуальных машин реального режима.

Переключение в защищенный режим:

1. Открыть адресную линию A20.
2. Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, она в нулевом кольце защиты).
3. Подготовить в оперативной памяти таблицу дескрипторов прерываний IDT.
4. Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по определенному адресу, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по этому адресу.
5. Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
6. Запретить все маскируемые и немаскируемые прерывания. Сохранить маски прерываний. Перепрограммировать контроллер прерываний.
7. Загрузить регистр GDTR и IDTR.
8. ***Перейти в защищенный режим*** (установить бит PE — нулевой бит в управляющем регистре CR0 в 1).
9. Загрузить новый селектор в регистр CS.
10. Загрузить сегментные регистры селекторами на соответствующие дескрипторы.
11. Разрешить прерывания.

Возвращение в реальный режим:

1. **Переключиться в реальный режим**
2. Сбросить очередь предвыборки, загрузить CS реальным сегментным адресом
3. Задать регистры для работы в реальном режиме
4. Загрузка IDTR (Interrupt Descriptor Table Register) для реального режима
5. Разрешаем немаскируемые прерывания

Разрешить маскируемые прерывания

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

Классификация прерываний (в зависимости от источника)

- программные (системные вызовы) – вызываются искусственно с помощью соответствующей команды из программы (int), предназначены для выполнения некоторых действий ОС (фактически запрос на услуги ОС), является синхронным событием.
- аппаратные – возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, мышь, 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 (посыл. сигнал о завершении процесса вв/выв на контроллер прерываний)
- 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.
  - Исправимые – приводят к вызову определенного менеджера системы, в результате работы которого может быть продолжена работа процесса (пр.: страничная неудача с менеджером памяти)
  - Неисправимые – в случае сбоя или в случае ошибки программы (пр.: ошибка адресации). В этом случае процесс завершается.

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 реальном режиме базовый вектор = 8h, в защищенном – первые 32 строки IDT отведены под исключения => базовый вектор = 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 подтвержд. разреш. контроллеру издавать новые прерывания.

Точные прерывания – прерывание, оставляющее машину в строго определенном состоянии. Обладает св-ми:

1. Счетчик команд указывает на команду (текущая), до которой все команды выполнены.
2. Ни одна команда после текущей не выполнена.

Не говорится, что команды после текущей не могли начать выполнение, а то, что все изменения, связанные с этими командами, должны быть отменены.

3. Состояние текущей команды известно (аппаратн. – обычно не нач. вып., при искл. – привела к исключ.)

Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера или подключенных к нему, поступают в процессор не непосредственно, а через 2 контроллера прерываний, I – ведущий, а II – ведомым. 2 контроллера используются для увелич. допуст. кол-ва внешних устройств (кажд. контроллер м. обслуживать сигналы от 8 устройств). Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них веерообразную структуру. В совр. машинах устанавливают 2 контроллера, увеличивая число входных устройств до 15 (7 у ведущего и 8 у ведомого).

Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Для ведущего контроллера в реальном режиме базовый вектор равен 8, для ведомого - 70h.

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

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

Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20-разрядный адрес – сегмент/смещение). Минимальная адресная единица памяти – байт.

2^{20} = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).

Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.

1-проц. режим под управлением MS-DOS (главное – минимизация памяти, занимаемой ОС => нет многозадачности).

0-256 б.: таблица векторов прерываний

резидентная часть DOS место резидентных программ сегменты программы

pool или heap (куча) транзитная часть DOS Резидентная часть DOS 1 Мб

2. Защищенный режим – многопроцессный режим. В памяти компьютера одновременно находится большое число программ с квантованием процессорного времени с виртуальной памятью. Управляет защищенным режимом ОС с разделением времени (Windows, Linux). В защищенном режиме 4 уровня привилегий, ядро ОС – на 0-м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться.

Разработан фирмой Digital Equipments (DEC) для 32-разрядных компьютеров VAX-11. Формирование таблиц описания памяти, которые определяют состояние её отдельных сегментов/страниц и т. п.

3. Специальный режим защищенного режима (V86) – 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.

VMM (Virtual Machine Manager) – для запуска виртуальных машин реального режима.

Переключение в защищенный режим:

12. Открыть адресную линию A20.
13. Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, она в нулевом кольце защиты).
14. Подготовить в оперативной памяти таблицу дескрипторов прерываний IDT.
15. Для обеспечения возможности возврата из защищённого режима в реальный необходимо записать адрес возврата в реальный режим в область данных BIOS по определенному адресу, а также записать в CMOS-память в ячейку 0Fh код 5. Этот код обеспечит после выполнения сброса процессора передачу управления по адресу, подготовленному нами в области данных BIOS по этому адресу.
16. Запомнить в оперативной памяти содержимое сегментных регистров, которые необходимо сохранить для возврата в реальный режим, в частности, указатель стека реального режима.
17. Запретить все маскируемые и немаскируемые прерывания. Сохранить маски прерываний. Перепрограммировать контроллер прерываний.
18. Загрузить регистр GDTR и IDTR.
19. ***Перейти в защищенный режим*** (установить бит PE — нулевой бит в управляющем регистре CR0 в 1).
20. Загрузить новый селектор в регистр CS.
21. Загрузить сегментные регистры селекторами на соответствующие дескрипторы.
22. Разрешить прерывания.

Возвращение в реальный режим:

6. **Переключиться в реальный режим**
7. Сбросить очередь предвыборки, загрузить CS реальным сегментным адресом
8. Задать регистры для работы в реальном режиме
9. Загрузка IDTR (Interrupt Descriptor Table Register) для реального режима
10. Разрешаем немаскируемые прерывания

Разрешить маскируемые прерывания

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.

- 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, клавиатура, 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

Прерывания в последовательности ввода\вывода

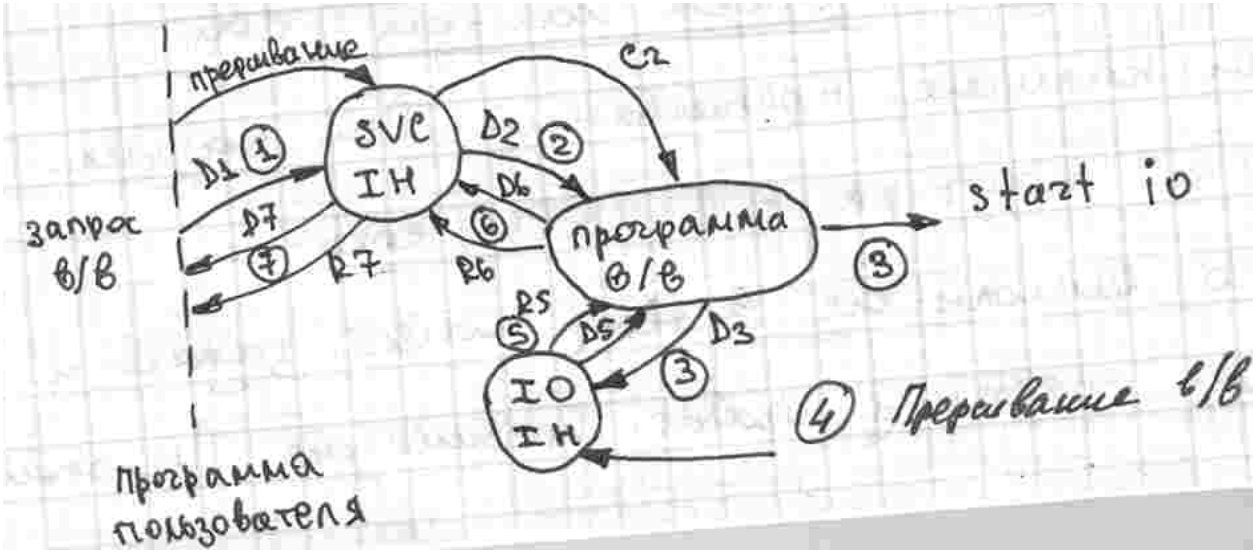
Прерывание сигнализирует о наступлении некоторого события. При программируемом вводе/выводе CPU должен ждать готовности устройства и постоянно опрашивать флаг его состояния (тратятся циклы процессорного времени).

При вводе/выводе с прерыванием CPU передает контроллеру команду ввода/вывода и переходит к выполнению другой работы. Когда контроллер будет готов обменяться данными с CPU, он пошлет CPU сигнал прерывания (сообщит, что его необходимо обслужить).

Чтение:

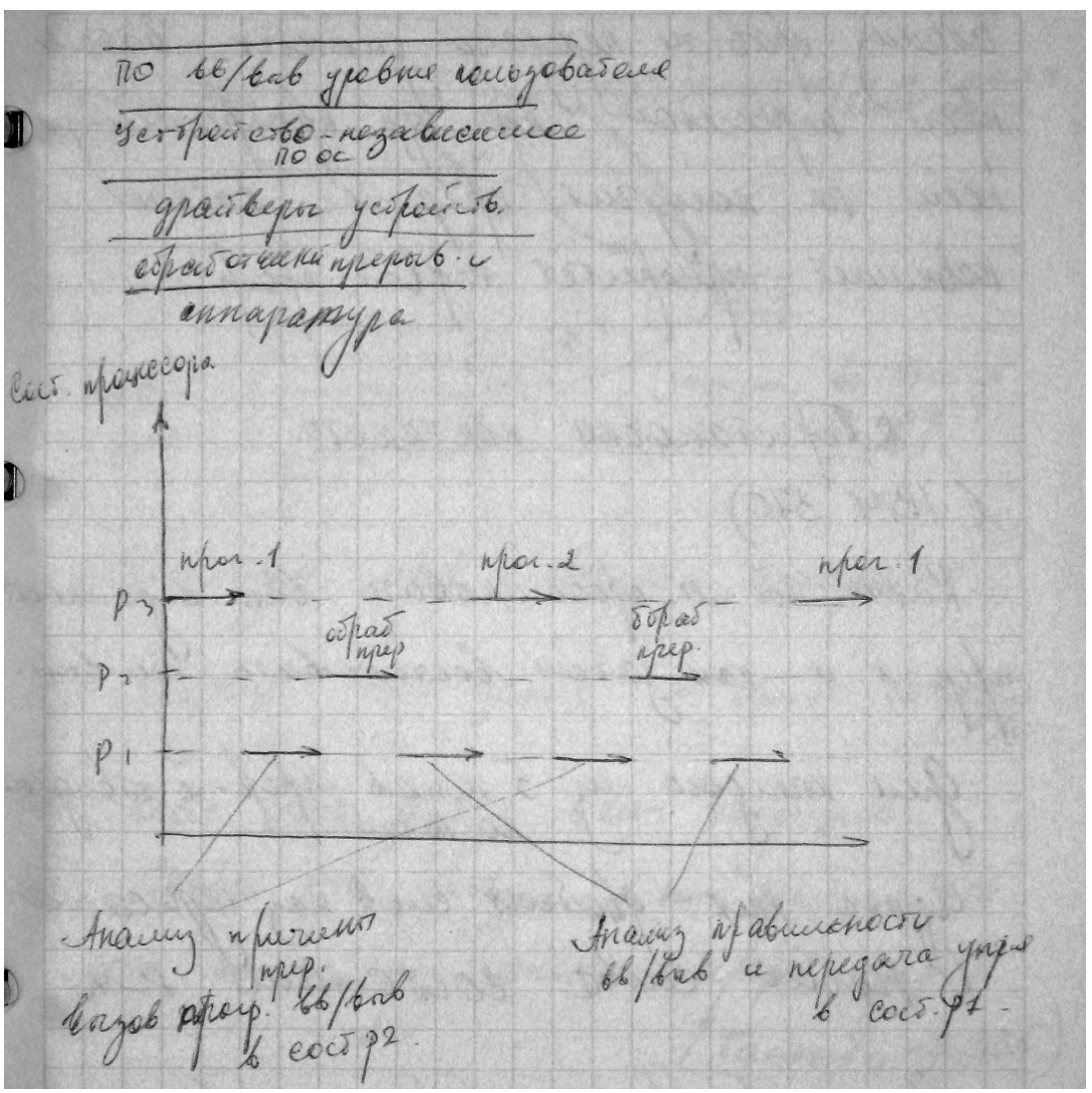
CPU генерирует команду READ, сохраняет содержимое счетчика команд и других своих регистров и переходит к выполнению других операций.

В конце каждого цикла команды CPU проверяет наличие прерываний. При поступлении прерывания от контроллера ввода/вывода, CPU сохраняет информацию о выполняемой в текущий момент задаче и выполняет программу, обрабатывающую прерывание. При этом он считывает слова из регистров контроллера ввода/вывода на шину данных и заносит их в память. По завершении обработки прерывания он восстанавливает контекст программ, от которой поступило прерывание и продолжает выполнение.



Ci – вызвать или разбудить
Ri – возвратить или разбудить
Di – данные
IH – interrupt handler
SVC – super visor call

Ввод/вывод с прерываниями более эффективен, чем программируемый, т.к. процессор освобождается от опроса флагов. Вся передача осуществляется через регистры CPU (медленная операция).



Программа 1, запросив операцию вв/выв была блокирована обработчиком прерывания, но перед этим система определила причину, вызвала обработчик => драйвер => инициализация операции вв/выв => обработчик прерывания. Затем анализ правильности передачи данных. Затем процессор переключился на другую программу пока не получит прерывание. Затем повышает приоритет процесса.