

## Билет 10

- 1) [Классификация структур ядер ОС. Особенности ОС с микроядром. Модель клиент-сервер. Три состояния процесса при передаче сообщений. Достоинства и недостатки микроядерной архитектуры.](#)
- 2) Прерывание реального режима Int 8h, функции. Задачи прерывания по таймеру в защищенном режиме. Префиксная команда lock()

## Билет 20

- 1) Взаимодействие процессов: монопольное использование – программная реализация взаимоисключения, примеры. Семафоры (определения, виды, примеры использования)
- 2) Планирование процессов в Win

## Билет 17

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

## Билет 9

- 1) Unix: концепция процессов – процессы «сироты», процессы «зомби», демоны; примеры.
- 2) Виртуальная память: сегментно-страничное распределение памяти по запросам. Достоинства и недостатки. Схема преобразования. Последовательность прерываний по особому случаю.
- 3) Это билет Даши, ей задали доп вопрос “Какими указателями поддерживается youngest living child в struct proc”

## Билет 8

- 1) Взаимоисключение и синхронизация процессов и потоков. Семафоры: определение, виды, примеры. «читатели и писатели», «производство-потребление» на семафорах
- 2) Аппаратные прерывания. Прерывание от системного таймера в защищенном режиме: функции

## Билет 18

- 1) Взаимодействие параллельных процессов, взаимоисключение, монопольный доступ, программная реализация взаимоисключения, алгоритм Деккера
- 2) защищенный режим, перевод компьютера в защищенный режим - последовательность действий

## Билет 1

- 1) ОС – определение, место ОС в системе программного обеспечения ЭВМ. Ресурсы вычислительной системы, потоки, их виды, процессы, диаграмма состояний, прерывания, классификация прерываний.
- 2) Три режима работы компьютера на базе процессоров Intel, адресация прерываний в защищенном режиме

## Билет 15

- 1) Unix: Процессы, разделяемая память, программные каналы
- 2) Синхронизация и взаимоисключение в распределенных системах - централизованно и ... то, что без token ring

## Билет 21

- 1) Взаимодействие параллельных процессов, монопольный доступ, методы взаимоисключения, мониторы, примеры, простой монитор, монитор "кольцевой буфер", задача «читатели и писатели»
- 2) Процессы. Создание/выполнение. fork/exec/диаграмма процессов

1. ОС – определение, место ОС в системе программного обеспечения ЭВМ. Ресурсы вычислительной системы. Режимы ядра и задачи: переключение в режим ядра. ОС с монолитным ядром. Система прерываний. Точные и неточные прерывания. Ядро ОС: многопоточное ядро; взаимоисключение в ядре – спин - блокировки.
2. Три режима работы компьютера на базе процессоров Intel. Защищенный режим: системные таблицы – GDT, IDT, теневые регистры. Уровни привилегий. Прерывания в защищенном режиме (таблица ID). Защищенный режим: перевод компьютера в защищенный режим – последовательность действий.
3. Классификация операционных систем и их особенности. Иерархическая машина. Виртуальная машина. Виртуальная память: сегментно-страничное распределение памяти по запросам. Достоинства и недостатки. Прерывания в сегментно-страничном распределении памяти. Методы управления виртуальной памятью, особенности, сравнение – достоинства и недостатки.
4. XMS, линия A20 – адресное заворачивание. Спецификация XM ( XMS ): Conventional, HMA, UMA, EMA.
5. Прерывания: классификация, аппаратные прерывания - механизм реализации. Прерывания точные и неточные. Методы организации ввода-вывода: программируемый, с прерываниями и прямой доступ к памяти
6. Понятие процесса. Процесс как единица декомпозиции системы. Процессы и потоки. Типы потоков. Диаграмма состояний процесса. Планирование и диспетчеризация.
7. Обеспечение монопольного доступа к разделяемым данным в задаче "писатели-читатели", используя Win32 API +Задача: читатели-писатели, решение Дейкстра с API ОС Unix.
8. Взаимоисключение и синхронизация процессов и потоков. Семафоры: определение, виды, примеры.
9. Защищенный режим: EMS, преобразование адреса при страничном преобразовании в процессорах Intel.
10. Управление памятью. Распределение памяти сегментами по запросам: стратегии выделения памяти, достоинства и недостатки.
11. Режимы работы компьютера IBM PC, кэши TLB и данных.
12. Классификация структур ядер ОС. Особенности ОС с микроядром. Модель клиент-сервер. Три состояния процесса при передаче сообщений. Достоинства и недостатки микроядерной архитектуры
13. Unix: команды fork(), wait(), exec(), pipe(), signal(). Unix: концепция процессов – процессы «сироты», процессы «зомби», демоны; примеры, средства взаимодействия процессов, сравнение – достоинства и недостатки.
14. Процессы: организация монопольного доступа – реализация взаимоисключения в помощь команды test-and-set, алгоритм Деккера.

15. [Unix: разделяемая память\(shmget\(\), shmat\(\)\) и семафоры \( struct sem, semget\(\), semop\(\)\). Пример использования](#)
16. [Подсистема ввода-вывода: синхронный и асинхронный ввод-вывод.](#)
17. [Средства взаимодействия процессов: мониторы – простой монитор, монитор "кольцевой буфер".](#)
- | Число процессов | Число ресурсов |
|-----------------|----------------|
| 1               | 2              |
| 2               | 2              |
| 2               | 3              |
| 3               | 3              |
| 3               | 4              |
18. [Предположим, что все ресурсы идентичны. Они могут приобретаться и освобождаться строго по одному. При этом процессам не требуется ресурсов более двух единиц ресурса. Сможет ли возникнуть тупик в каждой из следующих систем?](#)
19. [Виртуальная память: распределение памяти страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество.](#)
20. [Процессы: синхронизация процессов и алгоритмы взаимного исключения в распределенных системах.](#)
21. [Взаимодействие процессов: монопольное использование – программная реализация взаимоисключения, взаимоисключение с помощью семафоров; сравнение – достоинства и недостатки.](#)
22. [Тупики: определение тупиковой ситуации для повторно используемых ресурсов, четыре условия возникновения тупика, обход тупиков - алгоритм банкира. Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа и методы восстановление работоспособности системы.](#)
23. [Win32 API : CreateThread\(\), WaitForSingleObject\(\), WaitForMultipleObject\(\).](#)
24. [Процессы: активное ожидание на процессоре, зависание, тупиковая ситуация - анализ на примере задачи об обедающих философах. Считывающие и множественные семафоры. Мониторы: монитор кольцевой буфер.](#)
25. [Синхронизация процессов ОС Unix на примере задачи «производство-потребление».](#)
26. [Процессы: процесс как единица декомпозиции системы.. Контекст процесса. Переключение контекста. Классификация алгоритмов планирования. Ситуация - бесконечное откладывание – причины возникновения, алгоритмы адаптивного планирования.](#)
27. [Прерывание реального режима Int 8h. функции. Задачи прерывания по таймеру в защищенном режиме.](#)

[Назад](#)

**ОС – определение, место ОС в системе программного обеспечения ЭВМ. Ресурсы вычислительной системы. Режимы ядра и задачи: переключение в режим ядра. ОС с монолитным ядром. Система прерываний. Точные и неточные прерывания. Ядро ОС: многопоточное ядро; взаимоисключение в ядре – спин - блокировки.**

ОС – комплекс программ, которые совместно управляют ресурсами вычислительной системы и процессами, использующими их при вычислениях



**Ресурс вычисл. сист.** - любой из компонентов вычислительной системы и предоставляемые ею возможности (Время процессора, Объем физической памяти (ОЗУ), Устройство ввода/вывода, Каналы, Таймер, Данные, Ключи защиты, Реентерабельные коды самой системы(разработана таким образом, что одна и та же копия инструкций программы в памяти может быть совместно использована несколькими пользователями или процессами))

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

**Режим ядра** - привилегированный режим. Те части, 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 - API - ни одна ОС не позволяет напрямую обращаться к устройствам I/O. Все обращение происходит через системные вызовы, происходит переход системы в режим ядра, ядро задействует необходимый **драйвер** (специальная программа ядра, управляющая работой внешних устройств), в итоге формируется аппаратное прерывание.

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) - **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.

3) исключительные ситуации- exception(**Нарушения – fault** – это исключение, фиксируемое до выполнения команды или в процессе её выполнения. **Ловушка – Trap** – процессором обрабатывается после команды, вызвавшей это исключение. **Авария – abort** – данный тип исключения является следствием невосстановимых, неисправимых ошибок, например, деление на ноль. **Исправимые** – например, обращение

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

**Точные прерывания** - это прерывание, оставляющее машину в строго определенном состоянии (счетчик команд указывает, на команду, до которой все команды полностью выполнены. не одна команда после той, на которую указывает счетчик команд – не выполнена. состояние команды, на которую указывает счетчик команд – известно, причем в перечисленных условиях не говориться, что команды после той, на который указывает счетчик команд не могут выполняться, а утверждается , что все изменения связанные с выполнением этих команд должны быть отменены до выполнения обработки прерывания.)  
При аппаратных прерываниях счетчик команд обычно указывает на следующую команду.  
При исключениях – указывает на ту команду, которая вызвала прерывание.

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

Альтернативное решение: одни прерывания выполнять как точные, другие как неточные. Прерывания ввода вывода – точный, а исключения могут быть неточными, т.к. процесс будет завершен аварийно.

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

В одно/**многопоточных системах** система использования критического ресурса решается путем взаимоисключения, а механизм называется SpinLock. Простейшая функция механизмов взаимоисключений базируется на аппаратных реалиях: Interlocked Increment, Interlocked Decrement, Interlocked Exchange

При реализации этих функций шина блокируется на время выполнения, чтобы процесс не мог выполнить эту команду. **LOCK()** - префиксная команда, указывает, что следующая за ней команда выполняется как неделимая, то есть прервать выполнение команды нельзя (процессор во время выполнения команды, не может обратиться к памяти)

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

**VOID KeInitializeSpinLock(IN PKSPIN\_LOCK SpinLock)** Эта функция инициализирует объект ядра KSPIN\_LOCK. Память под спин-блокировку уже должна быть выделена в невыгружаемой памяти.

**VOID KeAcquireSpinLock(IN PKSPIN\_LOCK SpinLock, OUT PKIRQL OldIrql)** Эта функция захватывает

[Назад](#)

спин-блокировку. Функция не вернет управление до успеха захвата блокировки. При завершении функции уровень IRQL повышается до уровня DISPATCH\_LEVEL. Во втором параметре возвращается уровень IRQL, который был до захвата блокировки (он должен быть  $\leq$  DISPATCH\_LEVEL).

**Монолитное ядро** – программа, состоящая из подпрограмм (имеющая модульную структуру), содержащих в себе все функции ОС, включая планировщик, файловую систему, драйверы, менеджеры памяти. Поскольку это одна программа, то она имеет одно адресное пространство, следовательно, все её подпрограммы имеют доступ ко всем её внутренним структурам. Такие ОС делятся на 2 части – резидентную и нерезидентную. Любые изменения приводят к необходимости перекомпиляции всей программы. Пример: ОС UNIX, хотя она имеет минимизированное ядро. Состояние с минимизированным ядром. 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, которые непосредственно взаимодействуют с клавиатурой.

- **Монолитное ядро.** Отличительной особенностью данной схемы операционной системы является то, что все части её ядра – это составные компоненты одной программы, работающие в одном адресном пространстве. К основным преимуществам относят высокую скорость работы и простоту разработки модулей. Основным недостатком является то, что при нарушении работы одного из компонентов ядра, перестает работать вся система. При внесении изменений в аппаратное обеспечение, необходимо произвести полную перекомпиляцию всего ядра.
- **Микроядро.** Данная архитектура предоставляет минимальный набор для взаимодействия с оборудованием и основные функции для работы с процессами. К достоинствам можно отнести высокую степень модульности и возможность устойчивой работы, при возникновении ошибок или сбоев оборудования. Недостатком является то, что передача информации требует больших расходов ресурсов и времени .

[Назад](#)

[Назад](#)

**Три режима работы компьютера на базе процессоров Intel. Защищенный режим: системные таблицы – GDT, IDT, теневые регистры. Уровни привилегий. Прерывания в защищенном режиме (таблица ID). Защищенный режим: перевод компьютера в защищенный режим – последовательность действий.**

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

**Защищенный режим (protected mode)** 32-разрядный, многопоточный, многопроцессный, 4 уровня привилегий, доступно 4 Гб виртуальной памяти(для Pentium-64Гб). Параллельные вычисления могут быть защищены программно-аппаратным путем. В защищенном режиме 4 уровня привилегий. Ядро системы находится на 0-м уровне. Создан для работы нескольких независимых программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимодействие задач должно регулироваться. Программы, разработанные для реального режима, не могут функционировать в защищенном режиме. (Физический адрес формируется по другим принципам.)

**Режим виртуального 8086** В режиме V86 процессор может перейти из защищённого режима, если установить в регистре флагов EFLAGS бит виртуального режима (VM-бит). Номер бита VM в регистре EFLAGS - 17. Когда процессор i80386 находится в виртуальном режиме, для адресации памяти используется схема <сегмент:смещение>, размер сегмента составляет 64 килобайта, а размер адресуемой в этом режиме памяти - 1 мегабайт. Виртуальный режим - это не реальный режим процессора i8086, имеются существенные отличия. Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима. В виртуальном режиме используется трансляция страниц памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство, каждое размером в 1 мегабайт. Все задачи виртуального режима обычно выполняются в третьем, наименее привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор автоматически переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему, работающую в защищённом режиме.

**GDT** (global descriptor table) – таблица, которая описывает сегменты системы, общие сегменты (сегменты ОП) На начальный адрес GDT указывает GDT Register (32 разрядный). В системе только одна GDT.

**LDT** (local descriptor table) – таблица, которая описывает адресное пространство процесса. В LDT Register находится смещение до соответствующего дескриптора в GDT, описывающего сегмент, в котором находится LDT . Таблиц LDT столько, сколько процессов.

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

**GDT** Global descriptor table  
**LDT** Local descriptor table

—, inverupr descriptable

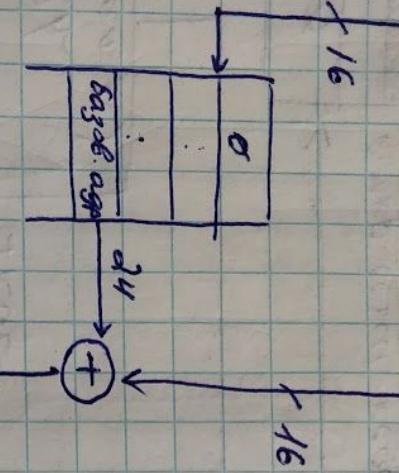
recapitulation now. age 6 yrs:

cenksoop cencessence

asymptotic

7 бус зраннійкою - 0 : 1-32р  
6 овер. пасу. операцій - 0 - 16р

sunrise.



physiognomy  
appear

output 1

7 Discrepancy set. concr. & nausse  
6 } Descriptor privilege level

7 Bus occupancy. cover. 8 na  
6 } Descriptor privilege level

7 Des occupancy. concr. & names  
 6 } Descriptor privilege level  
 5 }

7 Bus occupancy. chanc. of nanocars  
 6 } Descriptor privacy level  
 5 }  
 4 S: cars observe (0): 1.08m<sup>2</sup>. car.m.  
 3 Bus occupancy: circumlocut. ke

7 Bus spesialist. concer. & naauces  
 6 }  
 5 }  
 4 } Desriptor privelogy level  
 3 Bus nspesialis: cem gan / concer. ke  
 2 Bus nspesialis: cem gan / concer. ke

7 Subspace guest, owner, & manager  
 6 } Descriptor privilege level  
 5 }  
 4 S: user observes (0): 1. owner, owner.  
 3 Subspace guest: owner / owner, key  
 2 Key negotiator, owner, owner / owner  
 1 Nonowner, nonowner

7 Bus upcycleg. cover. & names  
6 } Descriptor privilege level  
5 }  
4 S: user objects (o): 1. obj. cover.  
3 Bus upcycleg. cover. / cover. key  
2 cognog4/cover; cover/cover  
1 dangers as cover / cognog4/cover  
+ dangers as cover / cognog4/cover

7 Bus npwcygert. conur. & naucresu  
 6 } Descriptor privilege level  
 5 }  
 4 } 5: user objects (o): 1. conur. conur.  
 3 Bus npwcygert: conur gant / conur. ko  
 2 kog nog 4 / conur; conur gant / conur es.  
 1 gampes 49 uyc em / Bus npwcygert gant  
 0 Bus npwcygert. conur. & naucresu

7 Sus upcyclets. coner. & naucsesu  
6 } Descripitor privilege level  
5 }  
4 S: cues obsecus (0): 1. obsec. corr.  
3 Sus npegnas: cem gak / coner. ke  
2 eognog 4/ obsecus; cem gak / coner. c  
1 gampes us wceim / huognupiugupgan  
0. bono/ ne bono obsecus. & coner.

7. *Sus uppgifts. concr. & nancess.*  
6 } *Descriptor privilege level*  
5 }  
4 }  
3 } *Sus nörgångar: cem/gan / concr. k  
2 } *eognog4/connex, cem/gan / concr.  
1 } *gasper ut i cem / monito/renosse  
0. } *Bono/ ne bono operans. k concr.****

7 Bus opacitetsf. cener. & nærværel  
6 } Desriptor privilege level  
5 }  
4 5: enes observer (0): 1. bønny. cener.  
3 Bus npegnar: cern gær / cener. k  
2 eognog 4/ cerner, cern gær / cener  
1 ydampes us ycen / mordno/kensie  
0. børne/ ne børne observer. k cener.

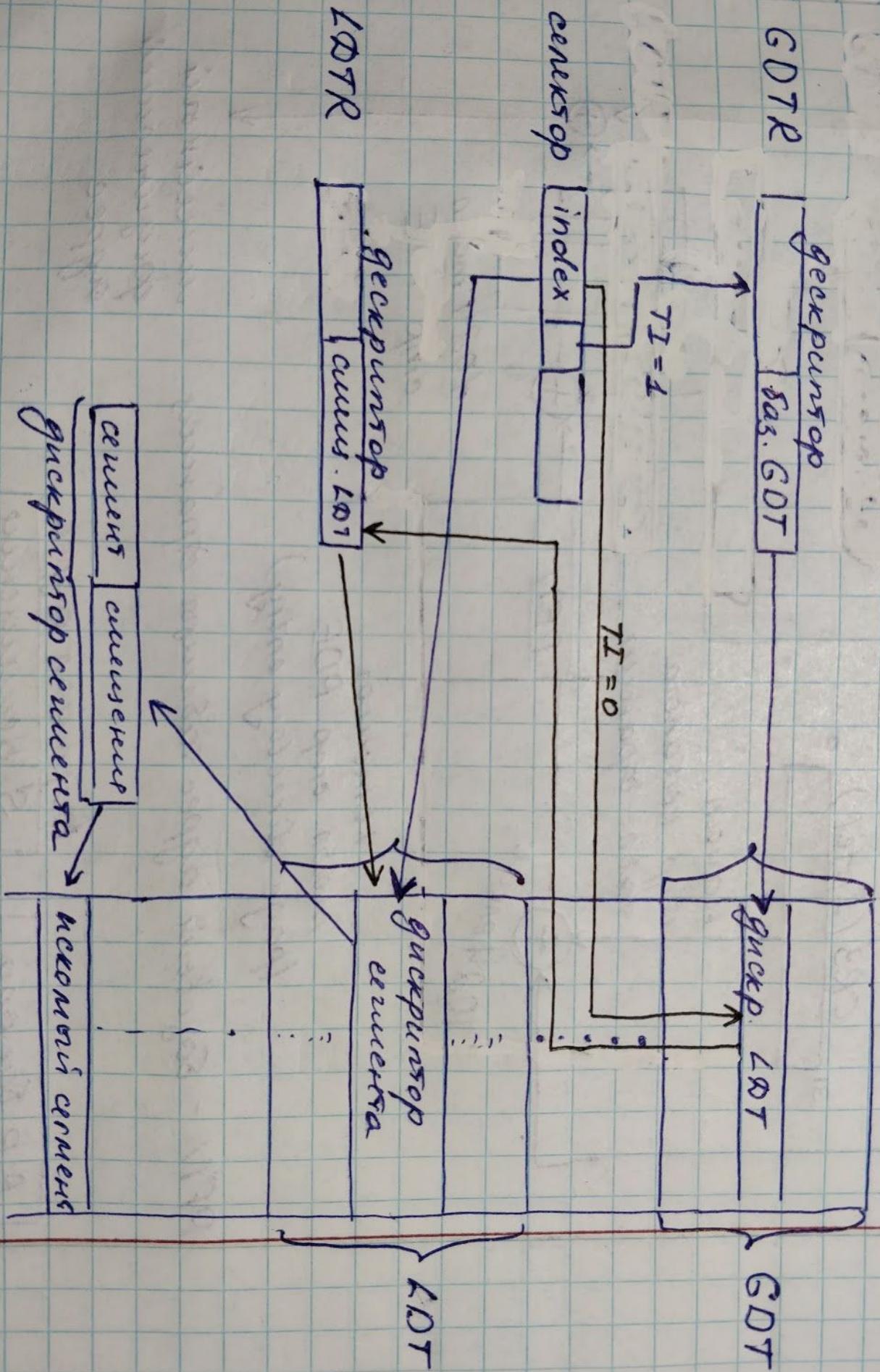
7 Bus npriyayest. cener. & naucenu  
6 } Descriptor privilege level  
5 }  
4 S: user objects (0): 1. obony. center.  
3 Bus npriyayest: centgant / centar. ke  
2 cognog4 / centner; centgant / centner  
1 yadper et yu cent / mognos / uenosce  
0. bnoa/ ne bnoa oobpus. k cener.

7 Bus upcyclegt. concer. & naaucesu  
6 } Descriptor privilege level  
5 }  
4 S: exec objects (0): 1. obby. com.  
3 Bus pregnaysr: cern gant / concer. ka  
2 eognog 4/ obbans, cern gant / concer. s  
& gasper us uenur / kroogungupriggga  
0. bono/ ne bono obbans. & concer.

7 Bus upcrgd. contr. & names  
6 } Descriptor privilege level  
5 }  
4 S: exec objects (e): 1. obj. contr.  
3 Bus upgrdmas: contr/gan/contr. kontr.  
2 obj/nog4/objekt, contr/gan/contr.cs  
1 gampes og upcrgd / Bus upgrdmas  
0. brno/ ne brno objekt. & contr.

7 Bus npwcygret. conur. & naucresu  
6 } Descriptor privilege level  
5 }  
4 5: cues obseus (o): 1. conur. conur.  
3 Bus npwgnasr: cem gan / conur. ka  
2 kog nog u / conur, 'cem gan / conur c  
1 y dmpes us wycem / nwnmno/nenasic  
0. bnow/ ne bono obseus. & ceece.

7 Bus upcyclet. cancer. & nanocesu  
6 } Descriptor privilege level  
5 }  
4 S: cues observe (o): 1. obser. corr.  
3 Bus nregnayr: cern gant / cancer. ke  
2 eognogu / corne, cern gan / can cr  
1 gasper us uceru / mognu / enosu  
0. sono/ ne sono observe. & cancer.



**Теневые регистры** В процессоре у каждого из сегментных регистров сопоставлен теневой регистр. Эти регистры доступны программисту и загружаются автоматически из таблицы дескрипторов сегментов в момент загрузки соответствующего сегментного регистра (параллельно). Это делается, чтобы как можно реже обращаться к оперативной памяти. "теневой регистр", в котором хранится дескриптор LDT текущей задачи. Это ускоряет в последующем обращение к локальной таблице дескрипторов текущей задачи. При переключении с одной задачи на другую для замены используемой LDT достаточно загрузить в регистр LDTR селектор новой LDT, а процессор уже автоматически загрузит в теневой регистр дескриптор новой LDT при первом обращении к нему.

## Уровни привилегий

Так как число программ, которые могут выполняться на более высоком **уровне привилегий**, уменьшается к уровню 0 и так как программы уровня 0 действуют как **ядро системы**, **уровни привилегий** обычно изображаются в виде четырех **колов защищты** (*Protection Rings*) (рис. 5.1).

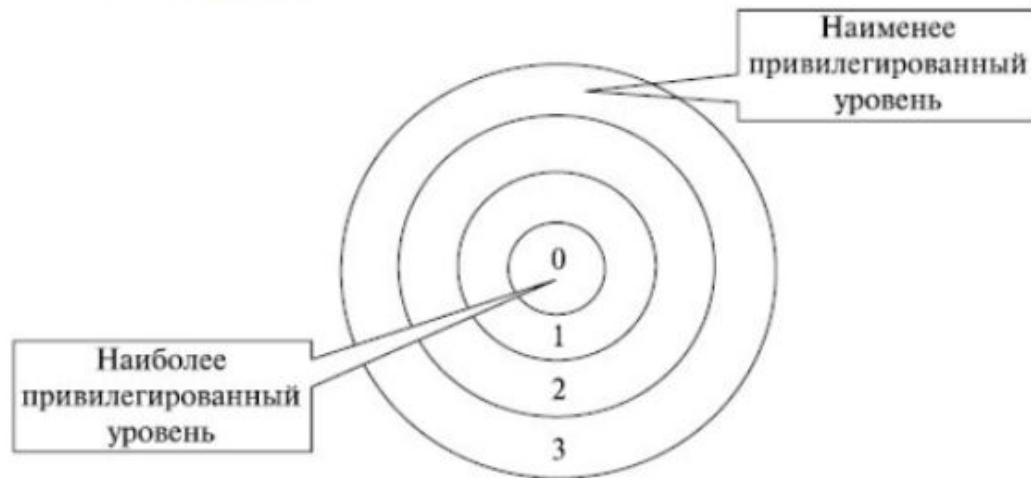


Рис. 5.1. "Кольца защиты"

Типовое распределение программ по кольцам защиты выглядит следующим образом:

- **уровень 0**: ядро ОС, обеспечивающее инициализацию работы, управление доступом к памяти, защиту и ряд других жизненно важных функций, нарушение которых полностью выводит из строя процессор;
- **уровень 1**: основная часть программ ОС (утилиты);
- **уровень 2**: служебные программы ОС (драйверы, СУБД, специализированные подсистемы программирования и т. д.);
- **уровень 3**: прикладные программы пользователя.

Аппаратные средства процессора, работающие в **зашитченном режиме**, постоянно контролируют, что текущая **программа** достаточно привилегированна для того, чтобы:

- выполнять некоторые команды, называемые привилегированными;
- выполнять операции ввода/вывода на том или ином внешнем устройстве;
- обращаться к данным других программ;
- передавать управление внешнему (по отношению к самой программе) коду командами межсегментной передачи управления.

**Привилегированные команды** - это те команды, которые влияют на **механизмы управления памятью, защиты и некоторые другие жизненно важные функции**. Это, например, команды загрузки таблиц дескрипторов *GDT*, *IDT*, *LDT*, команды обмена с **регистрами управления CRi**. Они могут выполняться только программами, имеющими наивысший (нулевой)

**уровень привилегий**. Это приводит к тому, что простую незашитченную систему можно целиком реализовать только в кольце 0, так как в других **кольцах защиты** не будут доступны все команды.

[Назад](#)

**Прерывания в защищенном режиме:** К дескрипторам GDT и LDT мы обращаемся с помощью селекторов, к дескриптору IDT мы обращаемся по смещению, которое берем из прерывания.

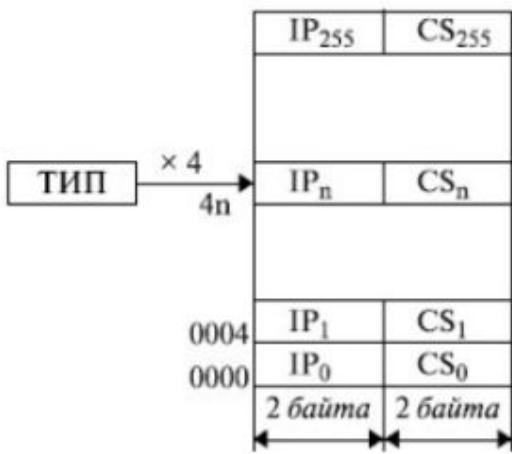
Обработчик прерывания: IDTR (указывает на начало IDT) + смещение из прерывания = дескриптор в IDT

Из дескриптора в IDT берем селектор С помошью 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.

В защищном режиме для вызова обработчика прерывания используется таблица дескрипторов прерываний *IDT*.

Элементами таблицы являются 8-байтные дескрипторы типа шлюз - специальные программные структуры, через которые происходит передача управления обработчику.



#### **Список действий для перехода в защищенный режим.**

- Необходимо проверить установлен ли бит 0 регистра CR0 в единицу. Если это так, то мы уже находимся в защищном режиме
- Сформировать таблицы GDT(LDT) и IDT
- Запретить маскируемые и немаскируемые прерывания
- Открыть линию A20 (для обеспечения адресного заворачивания)
- В сегментные регистры записать данные селекторов
- Занести базовые адреса сегментов в таблицу GDT
- С помощью привилегированных команд lgdt и lidt базовые адреса соответствующих таблиц в регистры GRDR и IDTR
- Устанавливаем бит 0 регистра CR0 в единицу.
- Jmp far protected\_entry (прыгаем на участок, который выполняется в защищном режиме).

#### **Список действие для перехода обратно:**

- Заполнение теневых регистров значениями FFh для включения адресации реального режима
- Заполняем сегментные регистры селекторами
- Возвращаем предыдущие значения регистров GDTR и IDTR
- Разрешить маскируемые и немаскируемые прерывания

[Назад](#)

[Назад](#)

**Классификация операционных систем и их особенности. Иерархическая машина. Виртуальная машина. Виртуальная память: сегментно-страничное распределение памяти по запросам. Прерывания. Достоинства и недостатки. Методы управления виртуальной памятью, особенности, сравнение – достоинства и недостатки.**

**Классификация ОС :** Однопрограммные пакетной обработки, Мультипрограммная пакетной обработки Мультипрограммная с разделением времени(В оперативной памяти находится большое число программ, процессорное время квантуется, чтобы обеспечить гарантированное время ответа системы. Время ответа системы не должно превышать 3 секунд). Системы реального времени. Серверные ОС – предоставляющие доступ к аппаратным ( принтеры ) и программным ресурсам (файлы доступа к Интернет) из сети. Многопроцессорные. Встроенные ОС (телевизоры, микроволновые печи). ОС для смарт-карт.

**Иерархическая машина:**

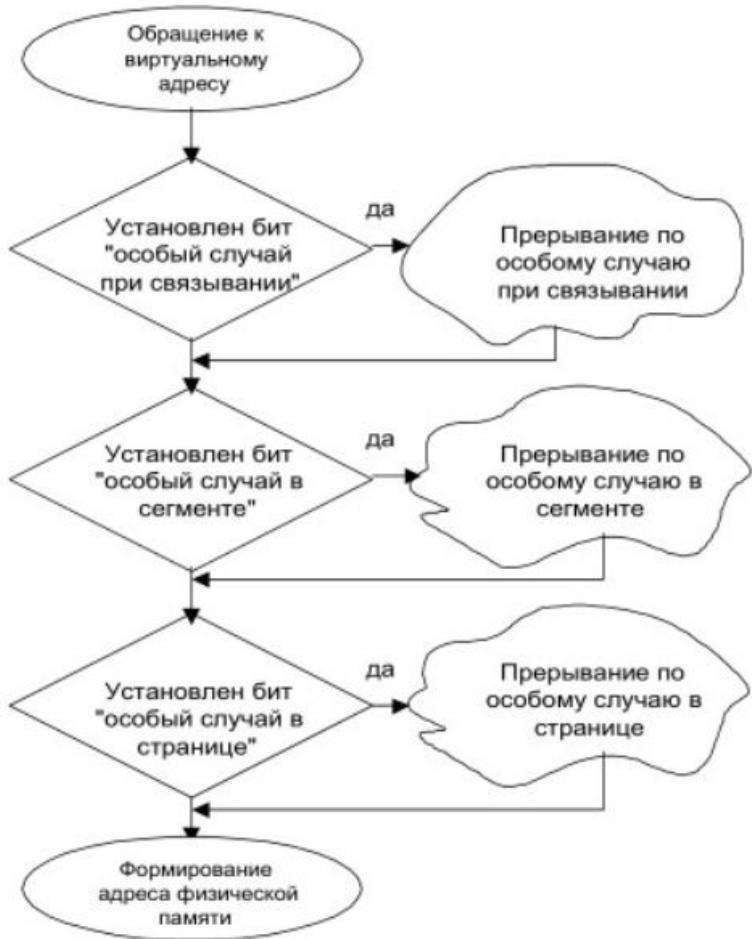
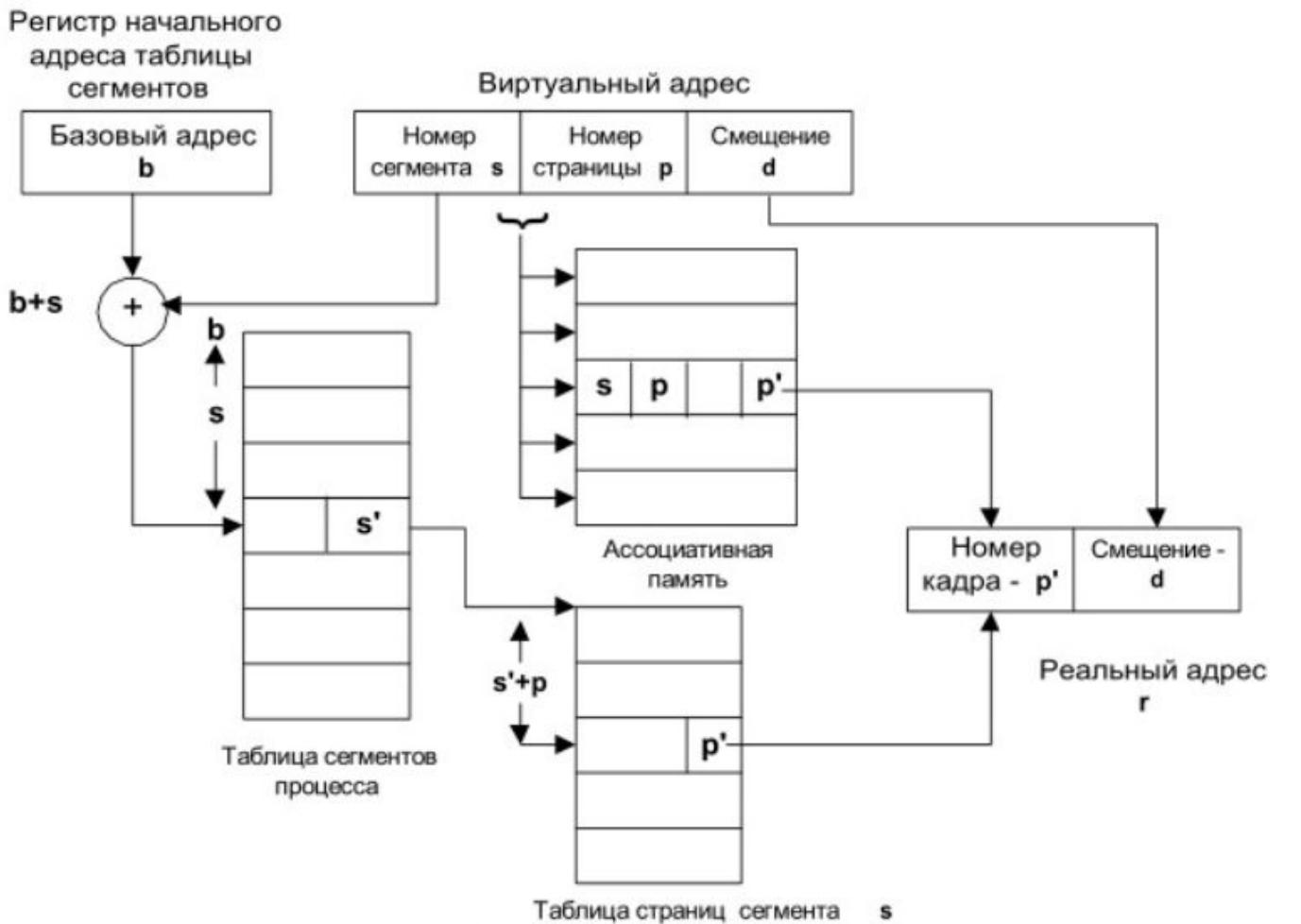
- Символьный уровень файлов подсистемы
- Управление информацией
- Управление устройствами
- Верхний уровень управления процессами
- Планирование процессов - управление оперативной памятью
- Нижний уровень управления процессами, выделение физической страницы
- Аппаратная машина

**Виртуальная машина** - кажущаяся, возможная, набор команд и функций, необходимых пользователю для получения сервиса операционной системы. **Виртуальная память** - кажущаяся, возможная. Виртуальная машина – набор команд и функций, необходимых пользователю для получения сервиса операционной системы. **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.

**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. 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. Каждая строка таблицы сегментов указывает на таблицу страниц соответствующего сегмента. Каждая строка таблицы страниц указывает либо на страничный кадр, к которому размещается данная страница, либо на адрес внешней памяти, где хранится эта страница. Если разместить все указанные таблицы в основной памяти, то они займут значительный объем. Компромиссным решением является размещение частей таблиц во внешней памяти. Коллективное использование в данной схеме реализуется указанием в таблицах сегментов адреса одной и той же таблицы страниц.



[Назад](#)

#### **страничное распределение памяти по запросам:**

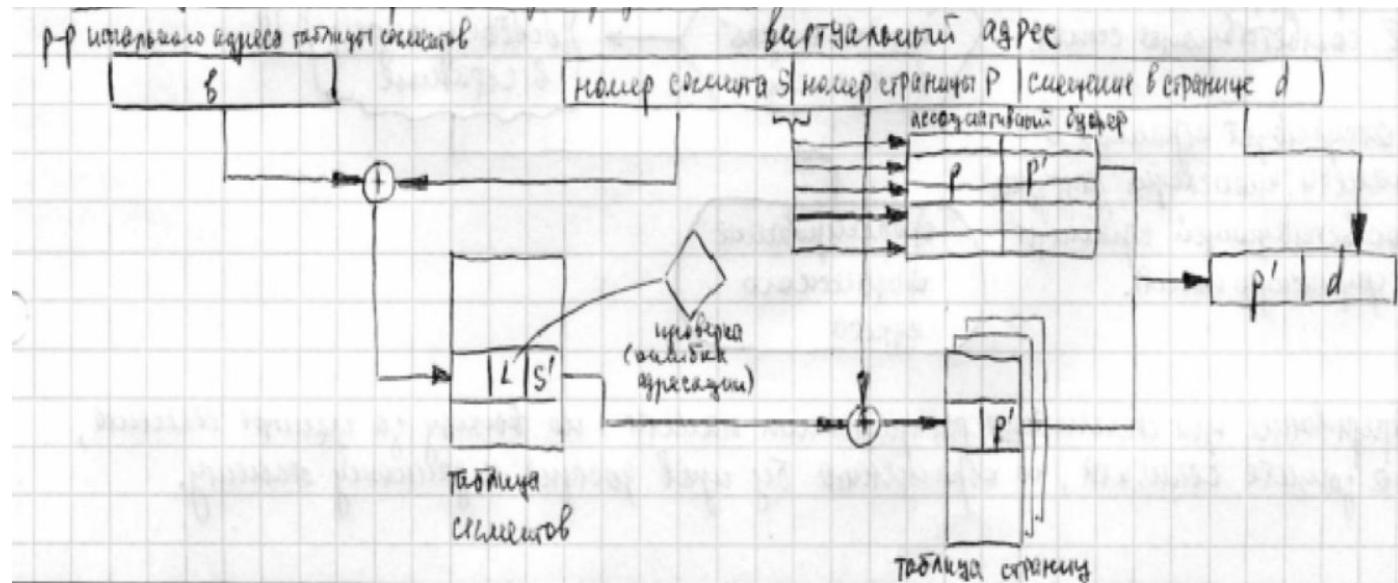
- + ) Такое распределение легко реализовать
- + ) Алгоритм LRU в этом достаточно эффективен
- ) На уровне страниц теряется их принадлежность.(Связано с преобразованием адр.) Существуют страницы, которые надо использовать одновременно нескольким процессам, которые в свою очередь могут иметь разные права доступа.

#### **сегментное распределения памяти по запросам:**

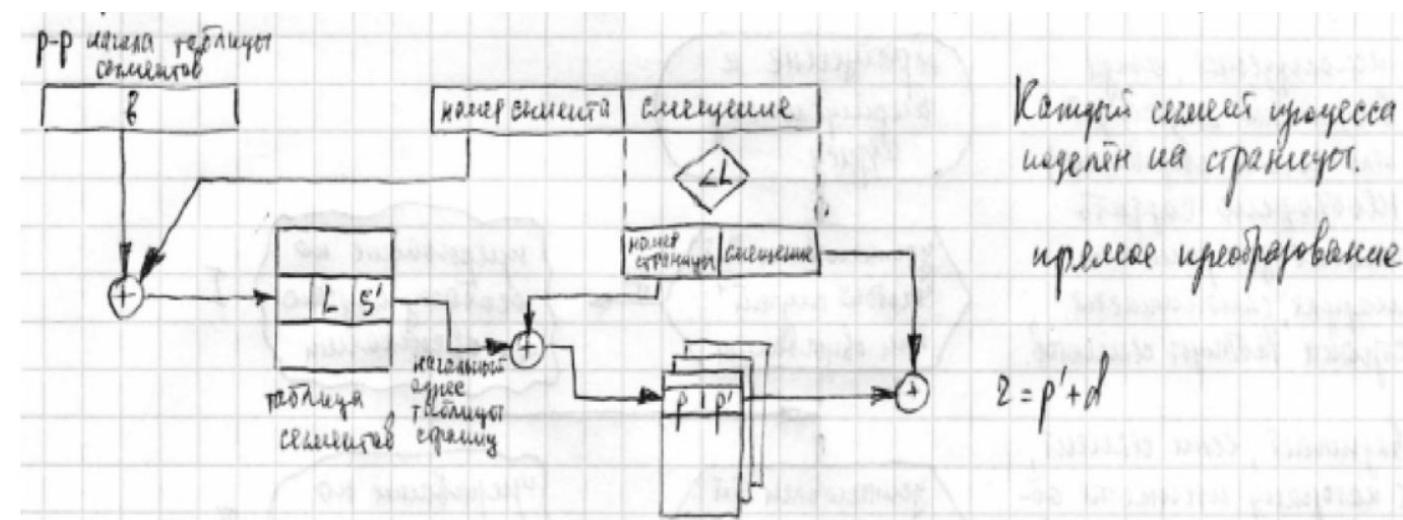
- + ) Легко реализовать коллективное использование, так как сегмент является логической единицей деления памяти
- ) Необходимость корректировки таблицы дескрипторов всех процессов при изменении размеров сегментов
- ) Загрузка новых сегментов (в памяти должно существовать пространство необходимого размера).
- ) Фрагментация

Назад

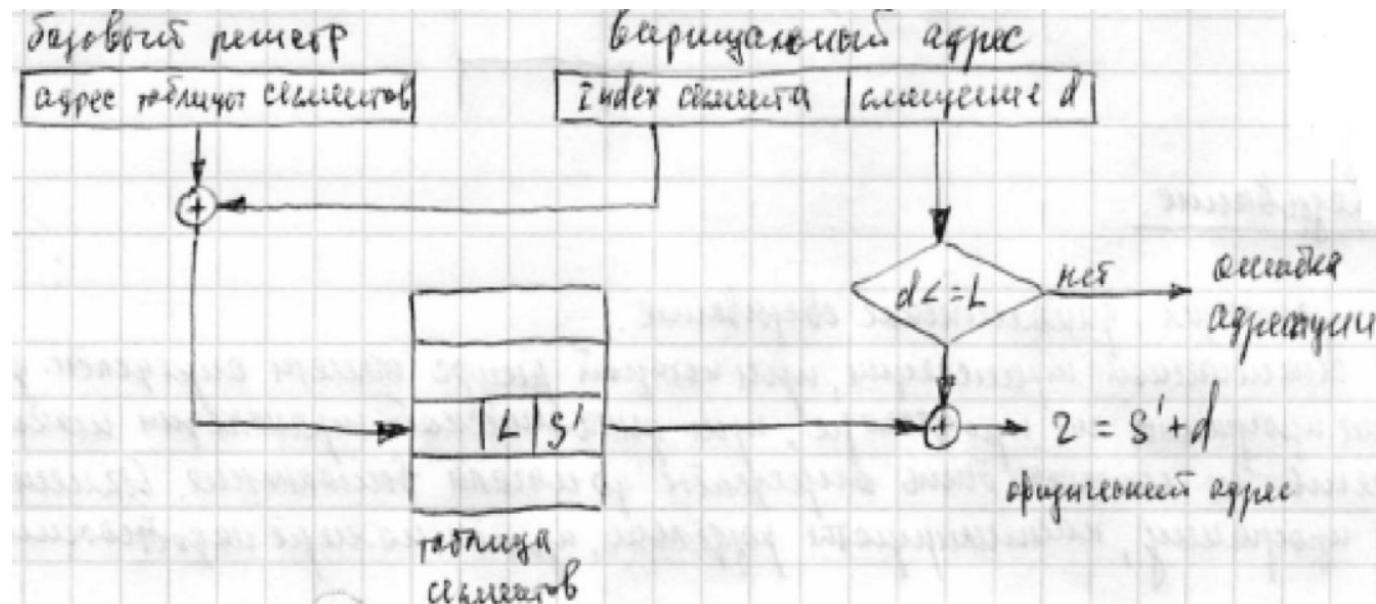
*сегментно-страничная с использованием кеша*



Без



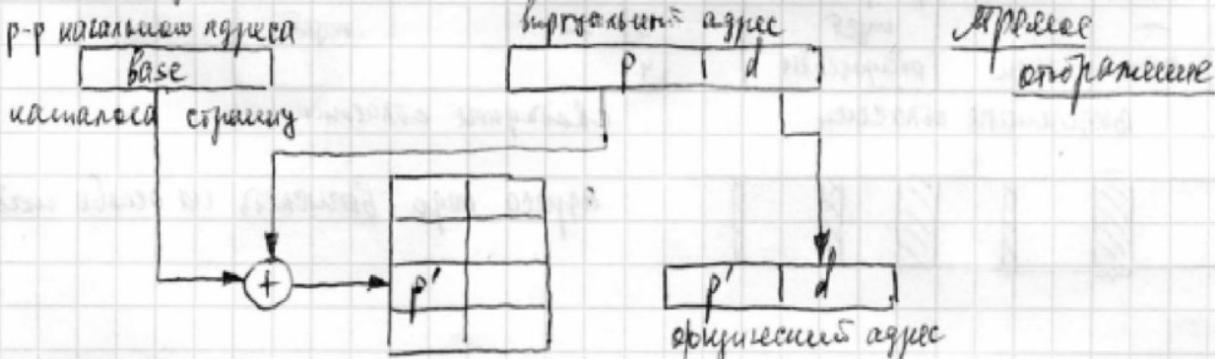
## **Управление памятью сегментами по запросам**



## **Одноуровневая страницная организация**

Существует 3 вида отображения изображения при спиралевидной распределении:

- прямое воспроизведение
  - ассоциативное
  - прямое-ассоциативное



[Назад](#)

**XMS, линия A20 – адресное заворачивание.** Спецификация XM ( XMS ): Conventional, HMA, UMA, EMA. Линия A20-21 – адресная линия, заземленная, нужно чтобы осуществлялось адресное заворачивание (для совместимости). При переходе в защ. режим необходимо открыть линию A20. В Intel 8086 только 20 адресных линий, что позволяло адресовать лишь 1 Mb памяти. При переполнении происходило обращение к таблице прерываний. Если в реальном режиме открыть линию A20, то станет доступным доп 64Kb памяти, что позволяет адресовать до 4 гигов памяти. Изначально линия A20 заземлена.

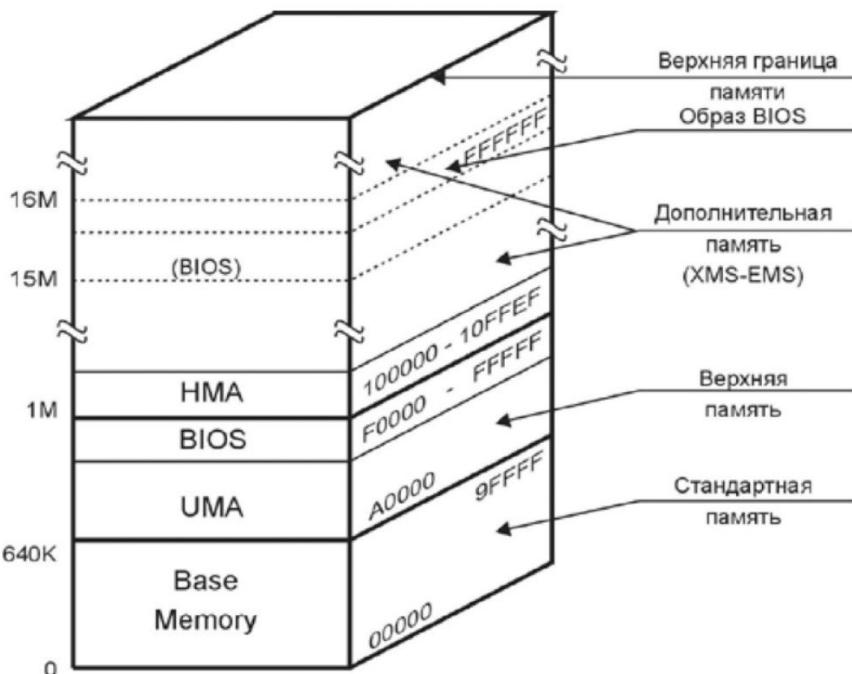
**Расширенная память XMS** (eXtended Memory Specification) - программная спецификация использования дополнительной памяти DOS-программами, разработанная компаниями Lotus, Intel, Microsoft и AST для компьютеров на процессорах 286 и выше. Эта спецификация позволяет программе получить в распоряжение одну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

также использовать область HMA.

Распределением областей ведает диспетчер расширенной

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

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



### Адресное заворачивание:

Процессор в реальном режиме поддерживает адресное пространство до 1Мбайт. Адресное пространство разбито на сегменты по 64Кбайт. 20-битный базовый адрес сегмента вычисляется сдвигом значения селектора на 4 бита влево. Данные внутри сегмента адресуются 16-битным смещением. В этом режиме формирования линейного адреса есть возможность адресовать пространство между 1Мб и 1Мб+64Кб (например, указав в качестве селектора 0FFFFh, а в качестве смещения 0FFFFh, мы получим линейный адрес 10FFEH). 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 пространства, аналогичное МП 8086.

[Назад](#)

**Прерывания: классификация, аппаратные прерывания - механизм реализации. Прерывания точные и неточные. Методы организации ввода-вывода: программируемый, с прерываниями и прямой доступ к памяти**

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

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

Исключения бывают 3 видов:

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

**Исправимые искл.** – приводят к вызову определенного менеджера системы, в результате работы которого может быть продолжена работа процесса (пример: страничная неудача с менеджером памяти)

**Неисправимые искл.** – в случае сбоя или в случае ошибки программы (пример: ошибка адресации). В этом случае процесс завершается.

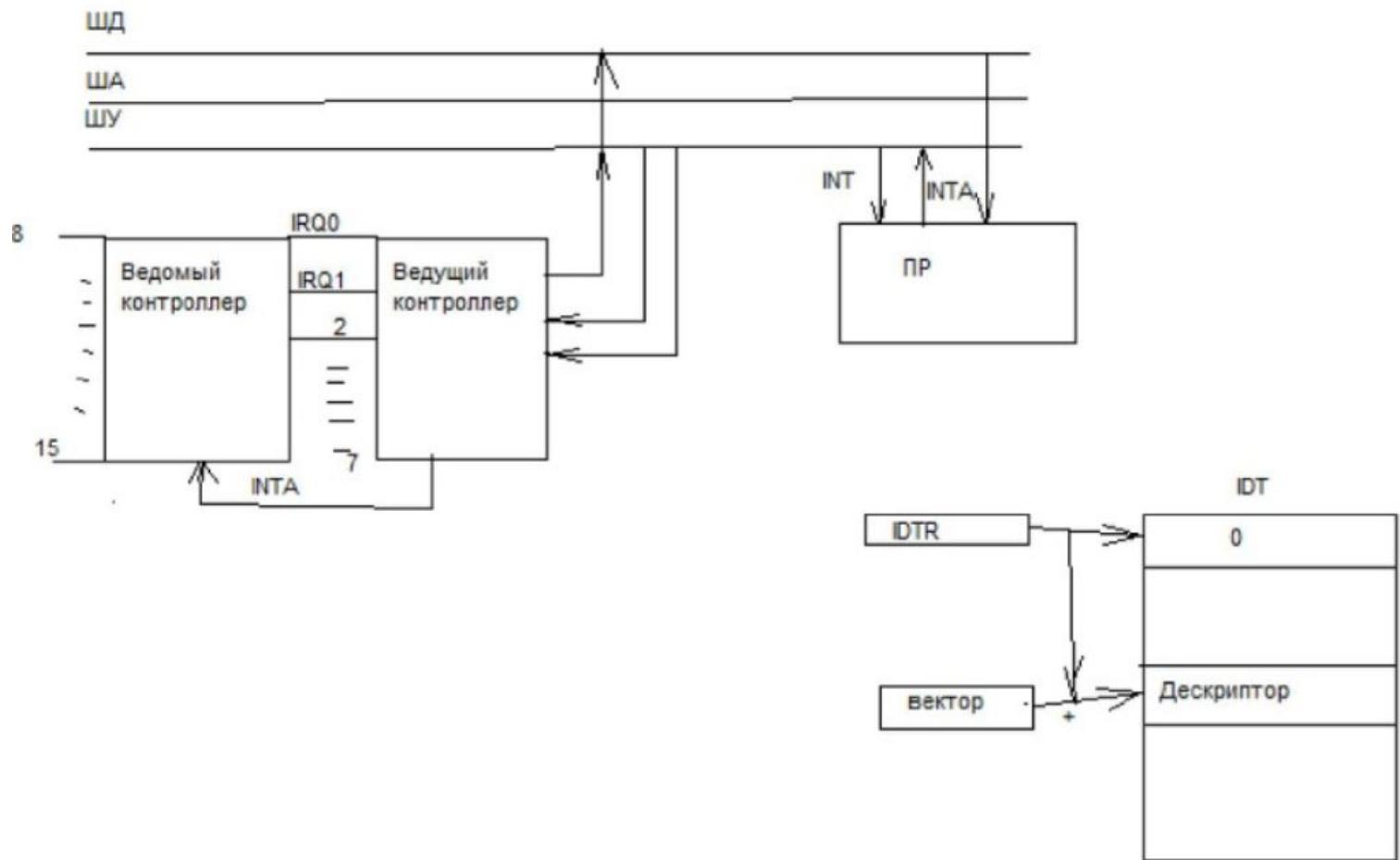
**Аппаратные прерывания** - возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура, системные часы, жесткий диск и т.д.), по времени возникновения эти прерывания асинхронны, т.е. происходят в случайные моменты времени. Различают прерывания: От таймера, От действия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

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

### **Механизм реализации аппаратных прерываний**

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

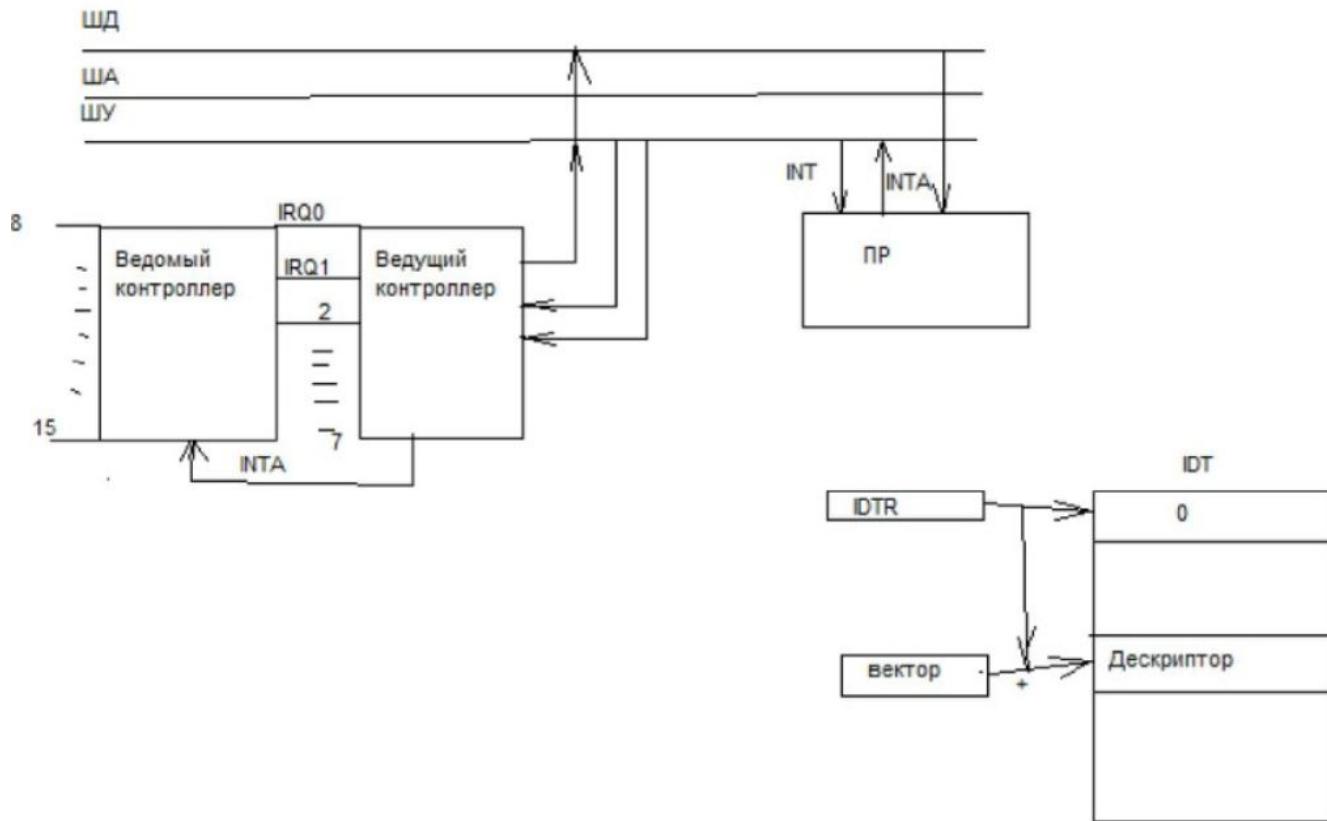
По этому вектору расположена таблица векторов и по адресу происходит переход на адрес обработчика прерывания. В защ режиме(IDT) все обработчики прерываний имеют соответствующий дескриптор в таблице дескрипторов прерываний.



**Точное прерывание** – это прерывание, оставляющее машину в строго определенном состоянии.  
(счетчик команд указывает, на команду, до которой все команды полностью выполнены. не одна команда после той, на которую указывает счетчик команд – не выполнена. состояние команды, на которую указывает счетчик команд – известно, причем в перечисленных условиях не говориться, что команды после той, на который указывает счетчик команд не могут выполняться, а утверждается , что все изменения связанные с выполнением этих команд должны быть отменены до выполнения обработки прерывания.)

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

[Назад](#)



1) **Программируемый IO.** В процессе выполнения программы встречается команда IO: периодический опрос нужных битов контроллера. Способ опроса – rolling. На ЦП возлагается непосредственное управление операцией IO: опрос, пересылка команд, передача данных. Для реализации данного способа необходимо иметь следующий набор команд: команды управления для инициализации работы внешнего устройства  
команды анализа состояния для проверки битов состояния контроллера  
команды передачи (из регистров ЦП в регистры устройства и наоборот)

2) **С использованием прерываний.** В конечном итоге будет выполняться передача из ЦП в контроллер команды соответствующей команды. После ЦП отключается от управления операцией IO и переходит на выполнение другой работы. Когда контроллер будет готов обменяться данными с ЦП, он пошлет прерывание. Процесс с точки зрения контроллера: Контроллер получает от ЦП команду (например, read). Он переходит к считыванию данных со своего устройства. Как только эти данные окажутся в регистрах контроллера, посыпается сигнал прерывания. После получения вектора центральным процессором, контроллер посыпает по шине данных данные из своих регистров. Но Каждое прерывание вызывает остановку выполнения текущих процессов и сохранение аппаратного контекста.

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

[Назад](#)

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

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



**Порождение(Рождение)** – присвоение процессу строки в таблице процессов

**Готовность** – попадание в очередь готовых процессов – получили все необходимые ресурсы.

**Блокировка(Ожидание)** – ожидание необходимого ресурса. Если процесс интерактивный, то он постоянно блокируется в ожидании ввода - вывода.

**Планирование** – определение, в какой последовательности процессы будут получать время ЦП (FIFO, приоритеты, динамические приоритеты).

После постановки в очередь происходит **диспетчеризация** – получение процессами времени ЦП.

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

Потоки на уровне пользователя (прикладные потоки)



Потоки на уровне ядра



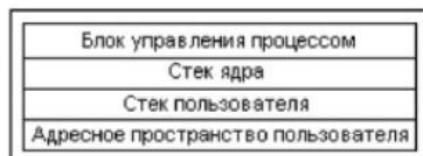
**Диспетчеризация** – выделение процессу процессорного времени. Вытеснение основано на системе с приоритетами. Если все процессы равноправны, то вытеснения нет. Процессорное время передается процессу с более высоким приоритетом. Статические назначаются в начале и с течением времени не меняются. Динамические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

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

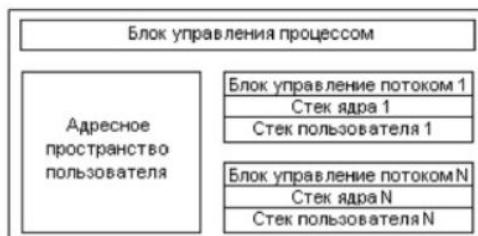
**Планировщик** – программа, которая отвечает за управление использованием совместного ресурса.

Необходимо убедиться, что процесс не будет поврежден сам и не повредит другие процессы.

Выбор между несколькими процессами, при возможности доступа любого к ресурсу, осуществляется некоторыми алгоритмами планирования. (без переключения и с переключением + без вытеснения и с вытеснением, +без приоритетов и с приоритетами (относительные или абсолютные, статические или динамические))



однопоточная модель процесса

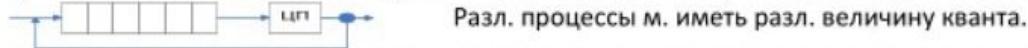


) многопоточная модель процессора

**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



- 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.

[Назад](#)

## Обеспечение монопольного доступа к разделяемым данным в задаче "писатели-читатели", используя Win32 API. Задача: читатели-писатели, решение Дейкстра с API ОС Unix

```
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);
}
```

WINDOWS

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

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

void stop_read(int semid){
    struct sembuf unlock[1]={ {READERS_COUNT,-1,0}
                           };
    semop(semid,unlock,1);
}

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

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

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

UNIX

содержимое бд. когда число читателей = 0, писатель получает возможность начать работу. Новый читатель не может начать работу, пока не завершится писатель. 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 может 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.

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 устраниют активное ожидание на процессоре.

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

```
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
```

## [Назад](#)

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

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

`P` (от датск. `passeren` - пропустить) и `V` (от датск. `vrygeven` - освободить). Защищённость семафора означает, что значение семафора может 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

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. находится только 1 процесс. Монитор является ресурсом.

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

**Разделяемый ресурс** - переменная, к которой обращаются разные процессы.

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

**Мьютекс (mutex – mutually exclusive (взаимно исключающий))** - механизм синхронизации, используемый для упорядочч. 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 безопасности
      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.

[Назад](#)

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

**Семафор** – неотрицательная защищенная переменная S, над которой определено 2 неделимые операции: P (от датск. passeren - пропустить) и V (от датск. vrugeven - освободить). Защищенность семафора означает, что значение семафора может 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 устраниют активное ожидание на процессоре. **Монопольный доступ осуществляется взаимоисключением, т.е. процесс, получивший доступ к разделяемой переменной, исключает доступ к ней др. процессов.**

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.

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

**Критическая секция** - строки кода, в которых происходит обращение к критическому ресурсу. Необходимо обеспечить монопольный доступ процесса к критическому ресурсу до тех пор пока процесс его не освободит. Т.е. чтобы не могли одновременно войти в крит. секцию. Все алгоритмы программной реализации обобщил Дейкстра, введя понятие семафора.

**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага (занятости ресурса другим процессом) Активное ожидание на процессоре является неэффективным использованием процессорного времени.

- Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции - ок
- Возможно **бесконечное откладывание** (зависание) – ситуация, когда разделенный ресурс снова захватывается тем же процессом.
- **Тупик** (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом

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

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

[Назад](#)

При проверке флага мы не переходим в режим ядра.

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

**Test-and-set** - простая неразрывная процессорная инструкция, которая копирует значение переменной в регистр, и устанавливает некое новое значение. Во время исполнения данной инструкции процессор не может прервать ее выполнение и переключиться на выполнение другого потока. Если используется многопроцессорная архитектура, то, пока один процессор выполняет эту инструкцию с ячейкой памяти, другие процессоры не могут получить доступ к этой ячейке, что может достигаться путем кратковременного блокирования шины памяти.

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

Особенность множественных семафоров – операции могут выполняться сразу над всеми семафорами данного множества. Изменение переменной S можно рассматривать как событие в системе.

Производство-потребление – считающие – буфер пуст и полон, один бинарный (монитор – кольцевой буфер) Читатели-писатели – монитор Хоара Обедающие философы – множественные семафоры

[Назад](#)

**Защищенный режим: EMS, преобразование адреса при страничном преобразовании в процессорах Intel.**

**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. Страницное преобразование может быть включено или не включено. Сегментное преобразование включено всегда. Если страницное преобразование включено, то используется пейджинг страниц, а не свопинг. 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 (upper memory area).

При страничном преобразовании адрес делится на три части, 12 бит смещение, индекс таблицы 10 бит, индекс оглавления 10бит.

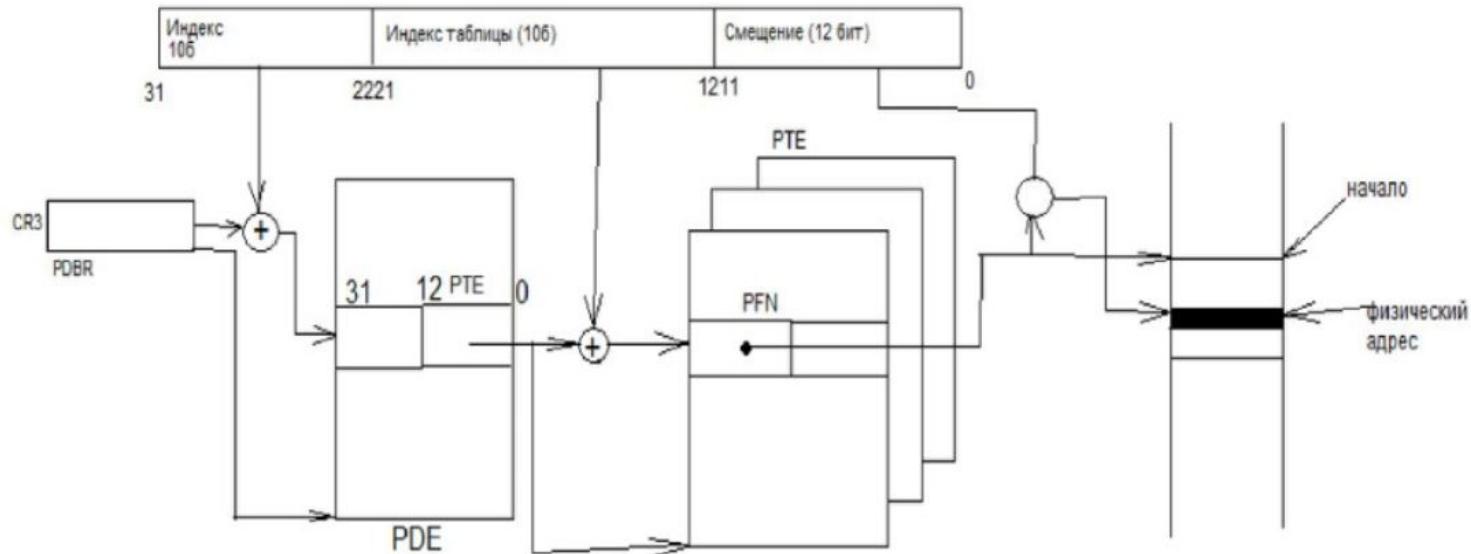
PDE – page directory entry (каталог страниц)

PFN – page frame number – указывает базовый адрес таблицы страниц

PTE – page table entry

Каталог страниц один на процесс. Может содержать 1024 дескриптора.

PDBR – содержит адрес каталога страниц для текущего процесса



PDE – page directory entry

PFN – page frame number

PTE – page table entry

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

[Назад](#)

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

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

### Управление виртуальной памятью:

Способы распределения:

1. Стр по запросам
2. Сегм по запросам
3. Сегменты подел по стр по запросам

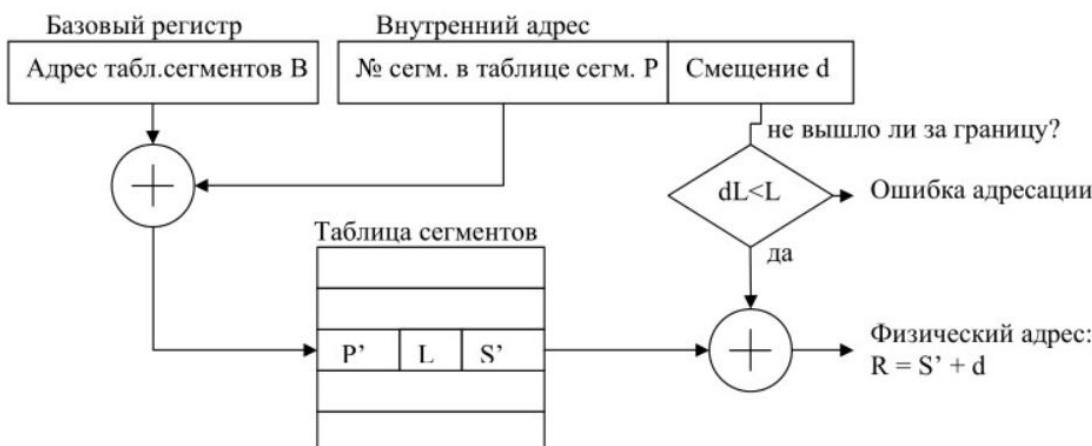
**Виртуальная память** – память размер которой превосходит размер реального физического адресного пространства. Виртуальное адресное пространство процесса делится на сегменты, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. При загрузке процесса часть сегментов помещается в оперативную память (при этом для каждого из этих сегментов операционная система

выбирает подходящий участок свободной памяти), а часть сегментов размещается в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки. **Каждый сегмент описывается дескриптором сегмента.** ОС строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в ОП или внешней памяти в дескрипторе отмечает его текущее местоположение (бит присутствия).

Дескриптор содержит поле адреса, с которого сегмент начинается и поле длины сегмента. Благодаря этому можно осуществлять контроль размещения сегментов без наложения друг на друга, обращаясь ли код выполняющейся задачи за пределы текущего сегмента. В дескрипторе содержатся также данные о правах доступа к сегменту (запрет на модификацию, можно ли его предоставлять другой задаче) защиты.

- + общий объем виртуальной памяти превосходит объем физической памяти
- + возможность размещать в памяти как можно больше задач (до определенного предела) увеличивает загрузку системы и более эффективно используются ресурсы системы
- увеличивается время на доступ к искомой ячейке памяти, т.к. должны вначале прочитать дескриптор сегмента, а потом уже, используя его данные, можно вычислить физический адрес (для уменьшения этих потерь используется кэширование - дескрипторы, с которыми работа идет в данный момент размещаются в сверхоперативной памяти - в специальных регистрах процессора);
- фрагментация;
- потери памяти на размещение дескрипторных таблиц
- потери процессорного времени на обработку дескрипторных таблиц.

[CLICK](#)



[Назад](#)

## **Режимы работы компьютера IBM PC, кэши TLB и данных.**

### Режимы работы компьютера на базе процессора INTEL

КЭШи применяются для ускорения доступа к какому-нибудь устройству. промежуточный буфер с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей вероятностью. Доступ к данным в кэше осуществляется быстрее, чем выборка исходных данных из более медленной памяти или удаленного источника, однако её объём существенно ограничен по сравнению с хранилищем исходных данных.

Кэш состоит из набора записей. Каждая запись ассоциирована с элементом данных или блоком данных (небольшой части данных), которая является копией элемента данных в основной памяти. Каждая запись имеет идентификатор, часто называемый тегом, определяющий соответствие между элементами данных в кэше и их копиями в основной памяти. Когда клиент кэша (ЦПУ, веб-браузер, операционная система) обращается к данным, прежде всего исследуется кэш. Если в кэше найдена запись с идентификатором, совпадающим с идентификатором затребованного элемента данных, то используются элементы данных в кэше. Такой случай называется *попаданием кэша*. Если в кэше не найдена запись, содержащая затребованный элемент данных, то он читается из основной памяти в кэш, и становится доступным для последующих обращений. Такой случай называется *промахом кэша*. Процент обращений к кэшу, когда в нём найден результат, называется *уровнем попаданий*, или *коэффициентом попаданий* в кэш.

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

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

**TLB** транслирует линейный адрес в физический и работает только со старшими 20 битами каждого из них; младшие 12 битов (представляющие собой смещение в странице) одинаковы как для линейного адреса, так и для физического.

Блоку элементов данных соответствует блок элементов достоверности, 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.

[Назад](#)

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

### **1. Монолитное ядро**

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

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

### **2. Микроядро**

Специальный модуль нижнего уровня, который обеспечивает работу с аппаратурой на самом низком уровне и базовые функции работы с процессами. Остальные компоненты – самостоятельные процессы, которые могут работать в разных ядерных пространствах.

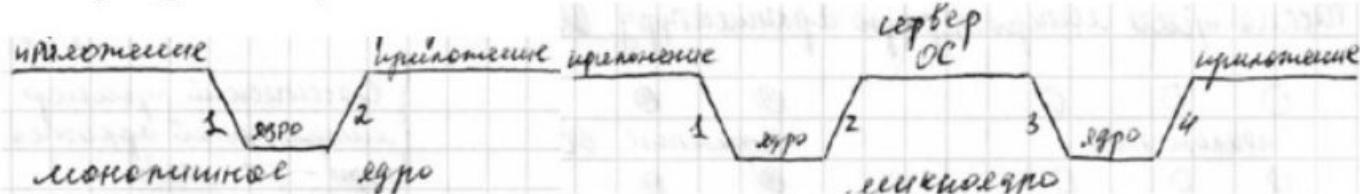
Общение между компонентами происходит с помощью сообщений через адресное пространство микроядра.

Микроядерная архитектура основана на модели клиент-сервер (например, ОС Mach, Hurd и Win2k(но не в классическом понимании))

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



В микроядре минимум 4 исключения



Модель клиент-сервер

Система рассматривается как совокупность двух групп процессов

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

Принято считать, что данная модель работает на уровне транзакций (запрос и ответ представляет неделимая операция)

[Назад](#)

3 состояния процесса при передаче сообщения(протокол обмена):

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

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

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

[Назад](#)

**Unix: команды fork(), wait(), exec(), pipe(), signal(). Unix: концепция процессов – процессы «сироты», процессы «зомби», демоны; примеры, средства взаимодействия процессов, сравнение – достоинства и недостатки.**

Базовое понятие – процесс. Процесс может находиться в двух состояниях – задача (процесс выполняет собственный код) и система (выполняет реентерабельный код ОС). Процесс может создавать любое число процессов (системный fork())

Строгая иерархия в отношении предок – потомок.

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

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

В результате вызова **fork** создается процесс-потомок, который наследует адресное пространство предка и все открытые файлы (фактически наследует код). В Unix все рассматривается как файл (файлы, директории, устройства). В состав этой ОС включен системный вызов **exec**, который заменяет адресное пространство на адресное пространство, которое вызывается в этом системном вызове. Бывает шести видов: execlp, execvp, execl, execv, execle, execve. Например, execl("/bin/ps", "ps", 0) Системный вызов **wait(&status)**. Процесс приостанавливается до тех пор, пока один из непосредственно порожденных им процессов не завершится **pipe()** Программный канал – это специальный буфер, который создается в системной области памяти.

Информация в канал записывается по принципу FIFO и не модифицируется. Предок и потомок могут обмениваться сообщениями с помощью не именованного программного канала. **signal()** Для изменения хода выполнения программы. Необходимо написать свой обработчик (в зависимости от того был получен сигнал или нет выполняются разные действия). 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 не завершившихся 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 (где 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.

[Назад](#)

```
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;  
}
```

### Средства взаимодействия процессов в Unix:

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

[Назад](#)

## Процессы: организация монопольного доступа – реализация взаимоисключения в помощью команды **test-and-set**, алгоритм Деккера. + [Семафоры](#)

Монопольный доступ осуществляется взаимоисключением, т.е. процесс, получивший доступ к разделяемой переменной, исключает доступ к ней др. процессов. Аппаратная реализация взаимоисключения (**test-and-set**). Впервые **test-and-set** была введена в OS360 для IBM 370. Эта команда является машинной и неделимой, т.е. ее нельзя прервать. Она одновременно производит проверку и установку ячейки памяти, называемой ячейкой блокировки: читает значение логической переменной B, копирует его в A, а затем устанавливает для B значение «истина» (все это делается за счет одной шины данных). Она присутствует в наборе сист вызовов Win и Unix Test-and-set(a, b) : a = b; b = true;

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

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

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

Пусть P1 хочет войти в свой критический участок, когда P2 уже там. P1 уст в единицу и входит в цикл проверки. Поскольку P2 нах в критическом участке, то у P2 – 1. P1 будет находиться в цикле активного ожидания, пока P2 не выйдет из своего критического участка. Т.к. **test-and-set** машинная неделимая команда и выполняется очень быстро, то бесконечного откладывания не возникает. Бесконечное откладывание – ситуация, когда разделённый ресурс снова захватывается тем же процессом. Ниже - алг. Деккера

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

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

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

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

Недостаток обоих методов – активное ожидание на процессоре.

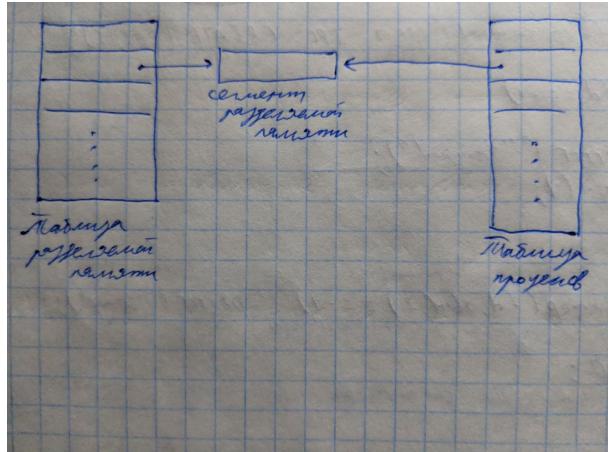
**Активное ожидание на процессоре** – ситуация, когда процесс занимает процессорное время, проверяя значение флага. Активное ожидание на процессоре является неэффективным использованием процессорного времени.

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

[Назад](#)

**Разделяемые сегменты памяти** – средство взаимодействия процессов через разделяемое адресное пространство. Ускоряет взаимодействие, т.к. отсутствует копирование из пространства пользователя в пространство ядра и наоборот.

Разделяемые сегменты подключаются к адресному пространству (далее АП) процесса. Поддерживаются системной таблицей разделяемых сегментов



Разделяемые сегменты находятся в **АП ядра**, т.к. АП каждого процесса защищено и взаимодействовать они могут только через посредника в лице другого АП (ядра)

**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\_CREAT 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\_CREAT (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 АП 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

Пример: создание сегмента с id 100, размером 1024 и полными правами доступа:

Нет механизмов управления доступом и взаимоисключения, поэтому часто используется вместе с семафорами

```
#include <sys/shm.h>
#include <string.h>
#include <sys/types.h>

int main()
{
    params = S_IRWXU | S_IRWXG | S_IROWXO;
    int fd = shmget(100, 1024, IPC_CREAT | params);
    if (fd == -1) { perror("shmget"); exit(1); }

    char *addr = (char *) shmat(fd, 0, 0);
    if (addr == (char *) -1) { perror("shmat"); exit(1); }

    strcpy(addr, "Aaaa");
    if (shmdt(addr) == -1) perror("shmdt");

    return 0;
```

**Семафор** – неотрицательная защищённая переменная, над которой определено 2 операции: P (пропустить) и V (освободить)

В linux - наборы считающих семафоров, представляются как массивы (но не совсем). Каждый семафор

описывается структурой struct sem {

short semid; // 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 }; Поддерживается все это таблицей семафоров в ядре

**int semget(key\_t key, int nsems, int semflg);** Возвращает идентификатор массива из nsem семафоров, связанного с ключом, значение которого задано аргументом key. Если массива семафоров, связанного с таким ключом, нет и в параметре semflg имеется значение IPC\_CREATE или

значение ключа задано IPC\_PRIVATE, создается новый массив семафоров. Значение ключа IPC\_PRIVATE гарантирует уникальность идентификации нового массива семафоров. semflg формируется аналогично shmflg, т.е. IPC\_CREATE | S\_IRWXU | S\_IRWXG...

**int semop(int semid, struct sembuf \*sops, unsigned nsops);** Выполняет операции над выбранными элементами массива семафоров, задаваемого идентификатором semid. Каждый из nsops элементов массива, на который указывает sops, задает одну операцию над одним семафором и содержит поля:

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

Значение поля sem\_op

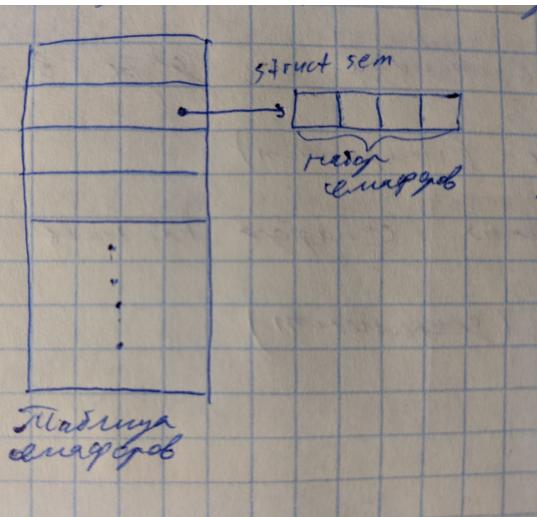
возможны следующие:

Sem\_op > 0: V  
(освобождение, инкремент семафора)

Sem\_op < 0: P (захват, декrement семафора)

Sem\_op = 0: Проверка семафора на 0 (если не равно, проверяющий процесс блокируется)

Пример: создание набора с id 100, из 2 семафоров, с полными правами доступа. Чтобы одной неделимой операцией semop изменять значения сразу нескольких семафоров набора, создаем массив структур



```
#include <sys/types.h>  
#include <sys/ipc.h>  
#include <sys/types.h>  
  
int main()  
{  
    int perms = S_IRWXU | S_IRWXG | S_IROKO;  
    struct sembuf sbuf[2] = {{0,-1,IPC_NOWAIT},  
                           {1,0,0}};  
  
    int fd = semget(100, 2, IPC_CREAT | perms);  
    if (fd == -1)  
    {  
        perror("semget");  
        exit(1);  
    }  
  
    if (semop(fd, sbuf, 2) == -1) perror("semop");  
    return 0;  
}
```

## **Подсистема ввода-вывода: синхронный и асинхронный ввод-вывод**

[Назад](#)

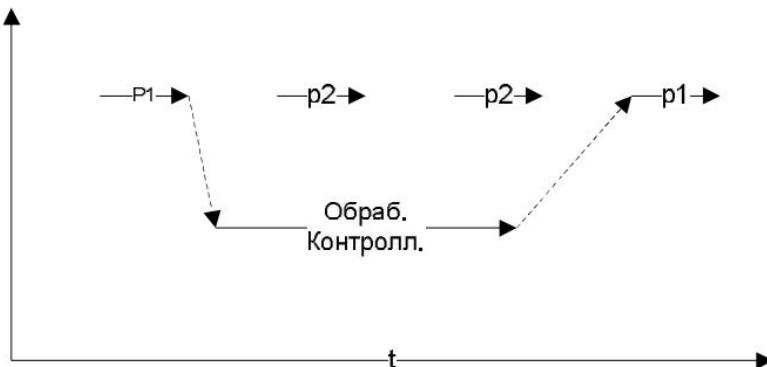
**Система вв/вывода** – часть вычислительной системы, ориентированная прежде всего на обмен сообщениями с центральным процессором.

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

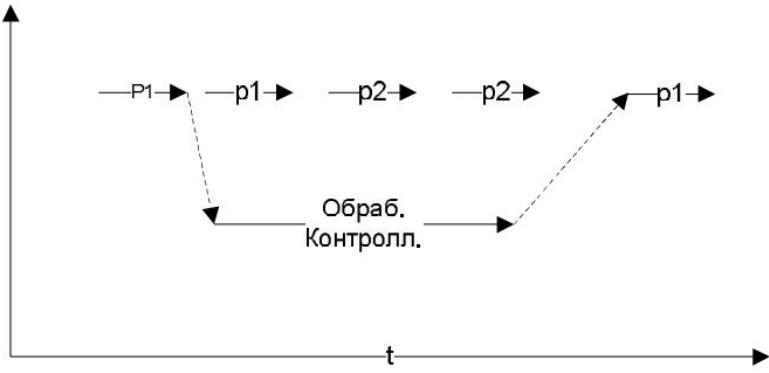
**Разработка подсистемы ввода/вывода** – наиболее трудоемкая задача при разработке ОС. Подсистема ввода-вывода решает следующие задачи:

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

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



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



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

[Назад](#)

**Монитор** - конструкция, предоставляемая языком программирования или системой, предназначенная для решения проблем взаимоисключения и взаимодействия процессов. Защищает данные (доступ возможен только функцией монитора).

**Монитор** - одно из основных средств структурирования ОС (задача ОС - динамически распределять ресурсы параллельным процессам/потокам).

простой монитор обеспечивает выделение единственного ресурса произвольному числу процессов (на фотке). Основа реализации - `wait()` и `signal()`, определенные на переменной типа “условие”

---

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

RESOURCE: MONITOR;

var

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

procedure producer(data: type)

begin

if (pos=n) then

  wait(buffer\_empty);

  bcircle[i]:=data;

  inc(pos);

  i:=(i+1) mod n;

  signal(buffer\_full);

end;

procedure consumer(var data: type)

begin

if (pos=0) then

  wait(buffer\_full);

  data:=bcircle[j];

  dec(pos);

  j:=(j+1) mod n;

  signal(buffer\_empty);

end;

begin

pos:=0;

i:=0;

j:=n;

end.



Число процессов	Число ресурсов	
1	2	
2	2	
2	3	
3	3	
3	4	

*Предположим, что все ресурсы идентичны. Они могут приобретаться и освобождаться строго по одному. При этом процессам не требуется ресурсов более двух единиц ресурса. Сможет ли возникнуть тупик в каждой из следующих систем?*

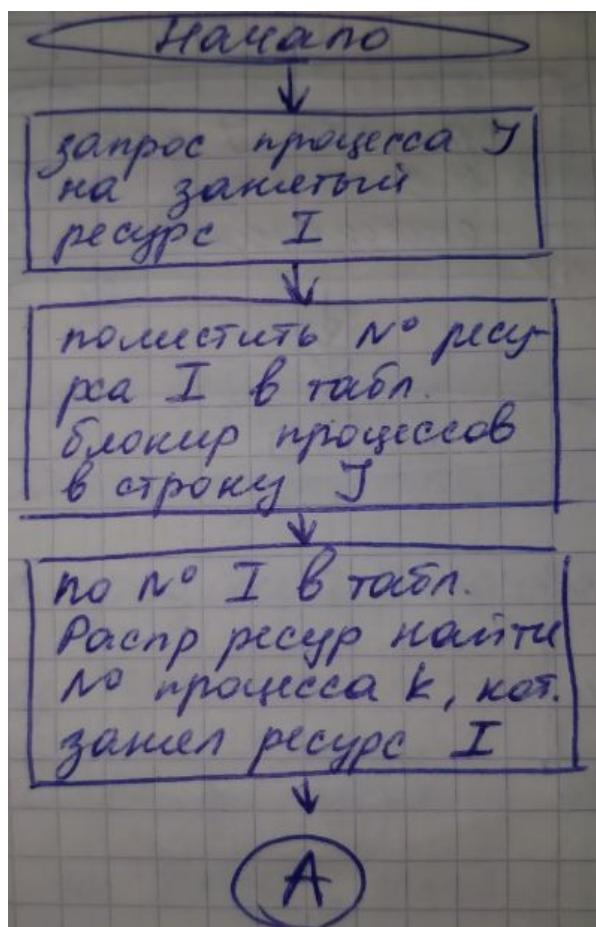
[Назад](#)

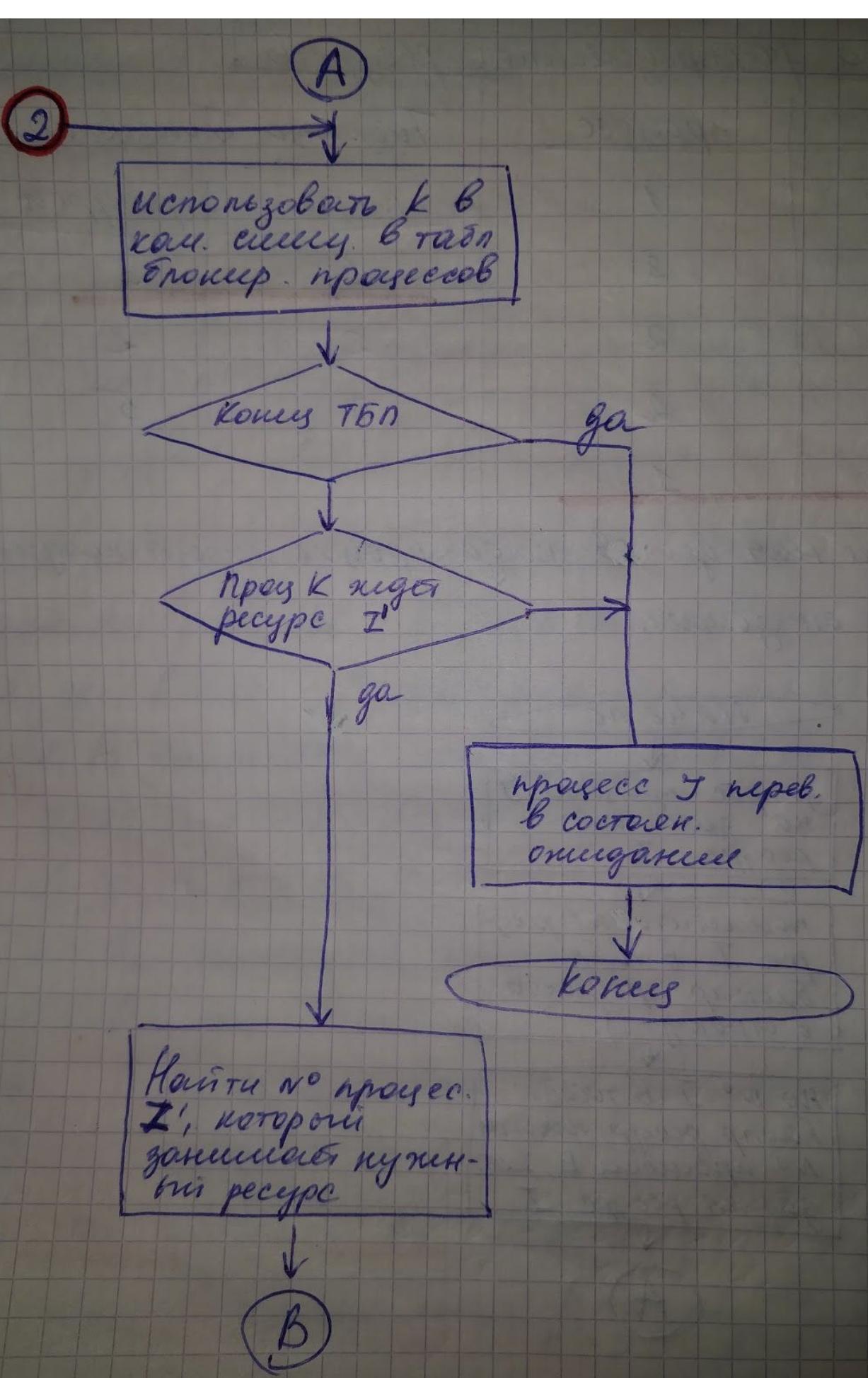
Несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет изменить свое состояние. Такая ситуация называется тупиком (deadlock). Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет.

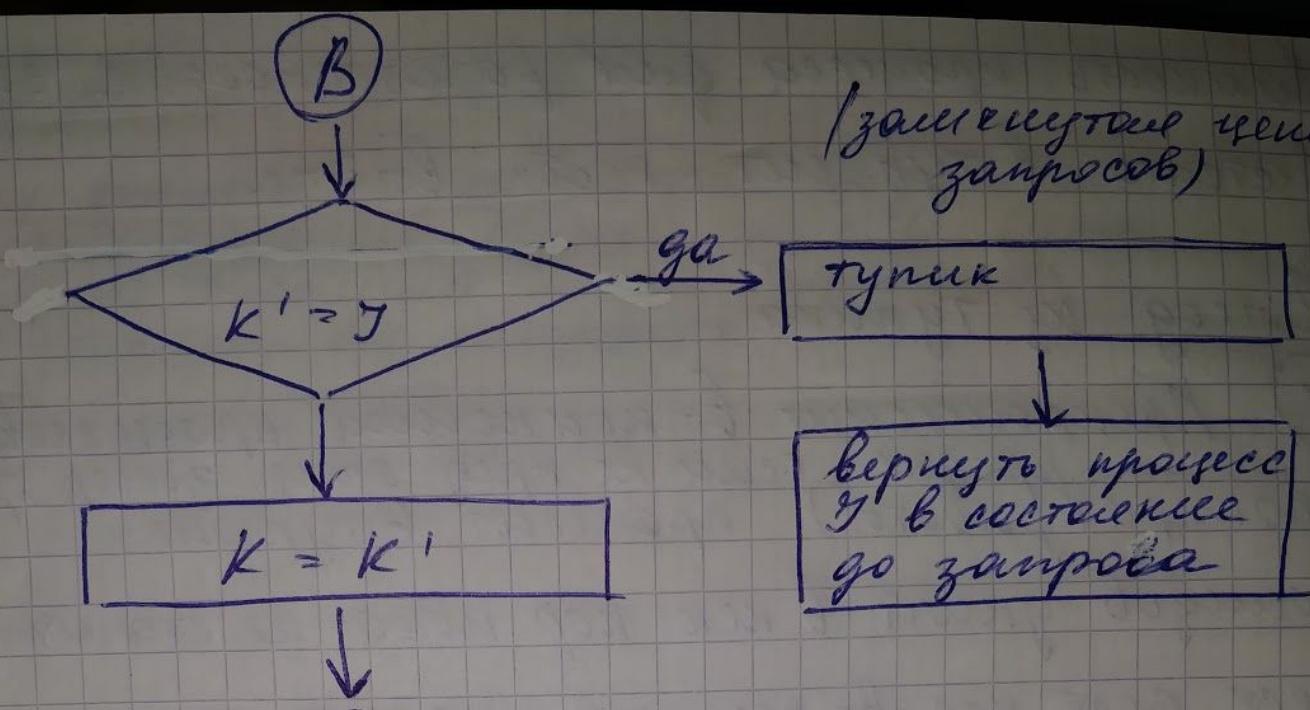
Условия возникновения тупиков были сформулированы Коффманом, Элфиком и Шошани в 1970 г.

1. Условие взаимоисключения (Mutual exclusion). Одновременно использовать ресурс может только один процесс.
2. Условие ожидания ресурсов (Hold and wait). Процессы удерживают ресурсы, уже выделенные им, и могут запрашивать другие ресурсы.
3. Условие неперераспределемости. Ресурс, выделенный ранее, не может быть принудительно забран у процесса. Освобождены они могут быть только процессом, который их удерживает.
4. Условие кругового ожидания (Circular wait). Существует кольцевая цепь процессов, в которой каждый процесс ждет доступа к ресурсу, удерживаемому другим процессом цепи.

Для образования тупика необходимым и достаточным является выполнение всех четырех условий, т.е необходимо наличие конкурирующих процессов. Для обнаружения тупиковой ситуации в системе используется анализ таблицы распределения ресурсов и таблицы блокированных процессов. Алгоритм действует тогда, когда процесс не может получить ресурс, так как ресурс занят







(запускает цепь запросов)

туник

вернуть процессу  
у в состояние  
до запроса

2

не могут получить

Процессы не могут выполн. запрос ~~требующие ресурсов~~  
Туник - замкнутая цепь запросов.

После обнаружения тумана

(предварительный анализ запрос - удобно  
для систем реального времени)

система отвечает other за первого опр. проце.  
времени - с параллелизмом, определяющим  
высочайшее процеессы. Честное real-time  
- неуде проходить интервал - патч.

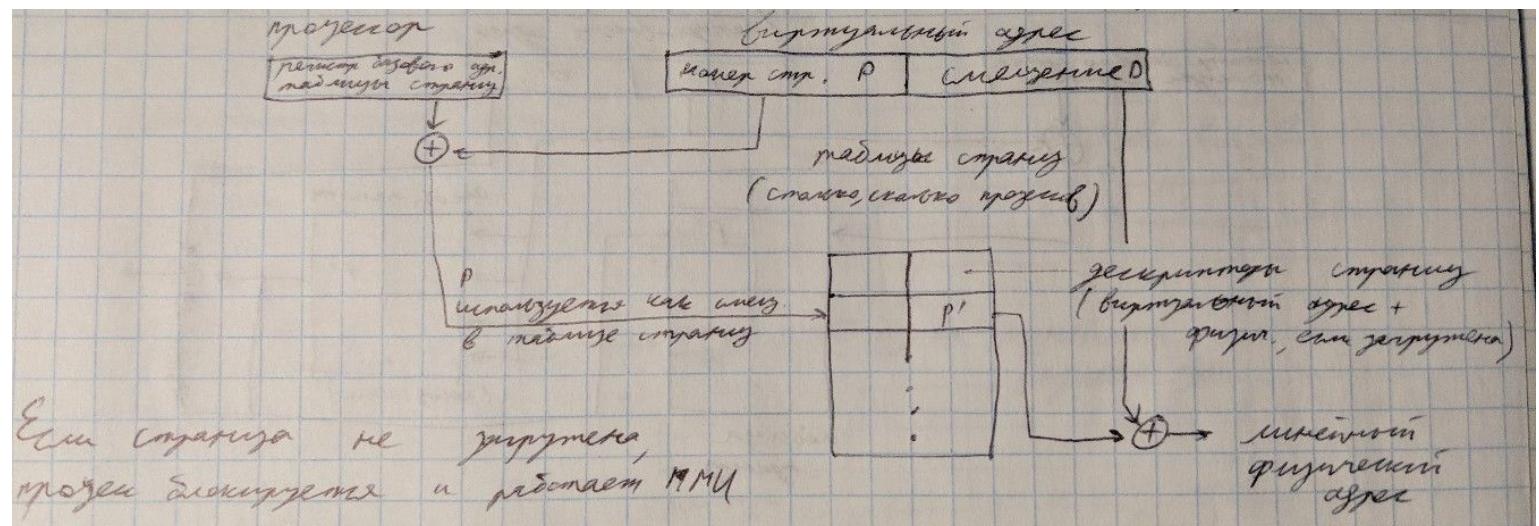
Выход из тумана - мониторинг состояния  
системы, не допускать возникновения тумана

Процесс не блок, чтобы сразу не создать  
тум. ситуации, но проан. неделе. Сигнализ.

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

[Назад](#)

Виртуальная память – память размер которой превосходит размер физического адресного пространства. 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



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

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

**Свойство локальности.** Использовании ассоциативного буфера на 8 адресов при страничном преобразовании дает нам 90% скорости полностью ассоциативной памяти благодаря свойству локальности. Локальность бывает 2х типов

1- ВременнаЯ – процесс обратившийся к одной странице наиболее вероятно в следующую единицу времени обратится к этой-же странице

2- Пространственная – процесс обратившийся к одной странице наиболее вероятно обратится к соседним страницам

## **Анализ страничного поведения процессов и рабочее множество**

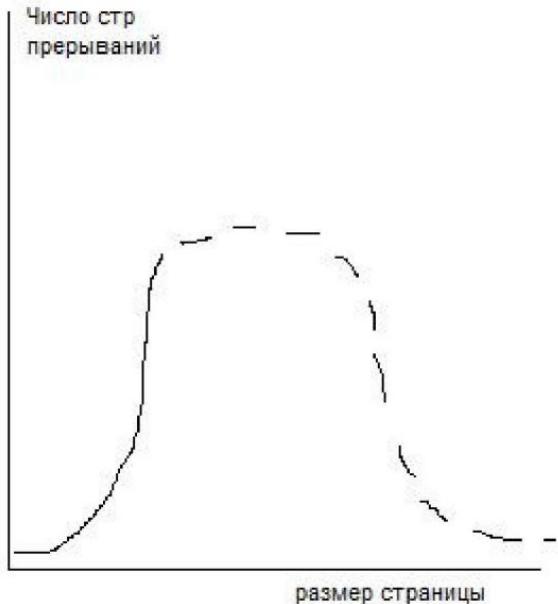
Для каждого процесса в каждый момент времени существует набор страниц которые он должен держать в памяти – рабочее множество. Если это набор не будет загружен возникнет трешинг страниц (постоянная загрузка и выгрузка страниц).

### Размер страницы

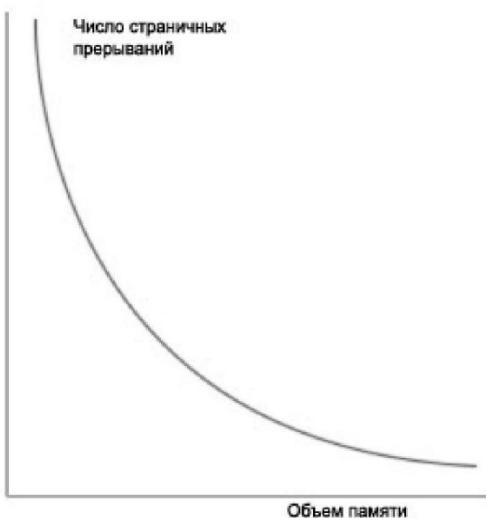
- Чем меньше размер страницы, тем меньше суммарный объем фрагментации памяти, но больше объем таблицы страниц памяти.
- Чем больше размер страницы, тем меньше команд на ней реально выполняется

### Аномалия размера страниц

- Увеличение числа страничных прерываний при увеличении числа страниц!



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





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

[Назад](#)

## Процессы: синхронизация процессов и алгоритмы взаимного исключения в распределенных системах

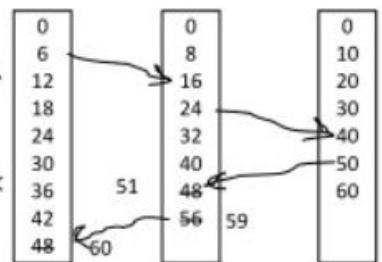
[Назад](#) **Процесс** - программа в стадии выполнения. Единица декомпозиции системы, с той точки зрения, что

### 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 входления в критические секции. Как только процесс готов войти в критич. область, он посыпает 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 посыпает 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 > критич. 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 посыпается 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 посып. 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 посыпает 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 1. Когда процесс получает Token, он смотрит, не требуется ли ему войти в критическую секцию. Если надо – входит (удерживая Token), если нет – посыпает 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 посыпает 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 посыпает др. Token, чтобы процессы могли обрабатывать свои данные

именно ему выдаются ресурсы ОС.

### 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 посыпает: «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.

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

[Назад](#)

[Алгоритм Деккера](#) + определение семафоров

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

Процесс - программа в стадии выполнения. Основная абстракция ОС.

Процесс многократно переходит из режима задачи (пользователя) в режим ядра и наоборот. В режиме задачи процесс выполняет свой код, в режиме ядра - реентерабельные коды ОС (не модифицирующие сами себя)

Каждый процесс имеет свою строку в таблице процессов - дескриптор. Ее номер = id процесса. Иерархия процессов похожа на дерево, и Unix старается любыми способами поддерживать древовидную структуру с помощью полей указателей в дескрипторе процессов. Это нужно для передачи сигналов.

Дескриптор - struct proc (Unix), struct task\_struct (Linux). Большая часть полей - указатели

Всегда есть процесс с id 0 - запустивший систему, и id 1 - терминальный процесс (мб не один такой)

Единственный способ создать процесс - fork()

Процесс, запустивший форк, становится лидером группы со своим id. Важнейшая группа процессов - терминальная. Процессы одной группы получают одни сигналы

«сирота» — возникает в том случае, если процесс предок завершился раньше своих потомков. При завершении процесса система проверяет не осталось ли у этого процесса незавершенных потомков. Если остались – система принимает действия по их усыновлению: Процесс потомок усыновляется терминальным процессом. Система переписывает идентификатор предка в дескрипторе процесса на 1, дает ему указатель на терминальный процесс, а последнему - указатель на нового потомка.

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

«демон» — процесс, предок которого - id 0 (не привязан ни к какому терминалу, сервисные функции)

**Тупики: определение тупиковой ситуации для повторно используемых ресурсов, четыре условия возникновения тупика, обход тупиков - алгоритм банкира. Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа и методы восстановление работоспособности системы.**

[Назад](#)

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. неперераспределемости (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.  $> \sum$  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.

**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  $X \neq \emptyset$  и множество рёбер  $U \neq \emptyset$ , а также инцидентор (трехместный предикат)  $P$ , причём высказывание  $P(x, u; y)$  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  $\pi = \{p_1, p_2, \dots, p_n\}$  и

подмножество вершин-ресурсов  $\rho = \{r_1, r_2, \dots, r_n\}$ .  $\pi, \rho : X = \rho \cup \pi, \rho \cap \pi = \emptyset$ .

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  $p_j$ , если эта  $p_j$  не является ни заблокированной, ни изолированной, путём удаления всех рёбер, входящих в  $p_j$  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  $P_i$  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

Как табл, так и списки д.б.

$A =$       |      |       $B =$       |      |      моноп. использ.

|      a<sub>ij</sub>      |      |      b<sub>ij</sub>      |

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:

1. на основе цены повт. запуска проц. от текущей точки
2. 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.

[Назад](#)

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

[Назад](#)

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,
    __in DWORD dwMilliseconds
```

```
// дескриптор объекта
```

```
// время ожидания
```

```
);
```

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

- Указанный объект находится в сигнальном состоянии.
- 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 сигнальное состояние
- WAIT\_OBJECT\_0 - Объект перешел в сигнальное состояние
- WAIT\_TIMEOUT - Истек срок ожидания
- WAIT\_FAILED - Произошла ошибка (например, получено неверное значение hHandle)

```
__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, чтобы одно приложение могло уведомить другое об окончании той или иной операции.

Объекты ядра мьютексы 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);
```

[Назад](#)

**Процессы: активное ожидание на процессоре, зависание, тупиковая ситуация - анализ на примере задачи об обедающих философах. Считывающие и множественные семафоры. Мониторы: монитор кольцевой буфер.**

## [Назад](#)

Процесс - программа в стадии выполнения. Единица декомпозиции ОС (именно ей 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 критические секции - ок
2. Бесконечное откладывание (зависание) – ситуация, когда разделённый ресурс снова захватывается тем же процессом.
3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и ждут. Т.е. Каждый ожидает освобождения ресурса, занятого другим процессом.

Проблема обедающих философов. Существуют только 3 схемы действия философов:

1. пытается взять обе вилки сразу. Если удаётся, 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. ugrueven - освободить). Защищённость семафора означает, что значение семафора может изменяться только операциями P и V.

- 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).
- 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 могут устанавливаться 1 операцией.

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

Пример: Производство/потребление:

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

```
                           S = 1;
                           Sf = 0;
                           P(Sf); // Sf-
                           P(S); // Занять
                           N--
                           V(S); // Освободить
                           V(Se); // Se++
```

```
Sf = 0; ...
```

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 может 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 – открытие доступа к монитору, задерживая выполнение процесса. Оно д.б. восстановлено операцией signal другим процессом. Если очередь пуста, то из очереди выбирается 1 из процессов и инициализируется его выполнение.

Монитор «кольцевой буфер» – решает задачу «производство-потребление», то есть существуют производители и потребители, а также буфер – массив заданного размера, куда производители помещают данные, а потребителичитывают оттуда данные в том порядке, в котором они помещались (FIFO).

```
resource: monitor circle_buffer;

const n = size;

var      buffer: array [0..n-1] 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
```

[Назад](#)

## 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)
```

[Назад](#)

**Процессы: процесс как единица декомпозиции системы., Контекст процесса. Переключение контекста.**

**Классификация алгоритмов планирования. Ситуация - бесконечное откладывание – причины возникновения, алгоритмы адаптивного планирования.**

## [Назад](#)

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 текущего 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. время ожидания, St - 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 ИО попадают в очередь с наивысшим приоритетом (FIFO). Квант времени для этой очереди выбирается из расчёта, чтобы максимальное количество процессов успело выдать запрос ИО 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.

[Назад](#)

## **Прерывание реального режима Int 8h, функции. Задачи прерывания по таймеру в защищенном режиме.**

### [Назад](#)

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 посыпается 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.

[Назад](#)