

## **Билет 1**

1. ОС — определение ОС. Ресурсы вычислительной системы. Режимы ядра и задачи: переключение в режим ядра — классификация событий. Процесс, как единица декомпозиции системы, диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра. Переключение контекста. Потоки: типы потоков, особенности каждого типа потоков. (2022 также)

### **ОС**

ОС (для обычного пользователя) - это интерфейс.

ОС (для профессионала) - это менеджер ресурсов вычислительных машин.

ОС- комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами, использующими эти ресурсы при вычислениях

#### *Классификация ОС*

1. Однопрограммные пакетной обработки
2. Мультипрограммная пакетной обработки
3. Мультипрограммная с разделением времени В оперативной памяти находятся большое число программ, процессорное время квантуется, чтобы обеспечить гарантированное время ответа системы.  
*Время ответа системы не должно превышать 3 секунд.*
4. Системы реального времени
5. Серверные ОС – предоставляющие доступ к аппаратным (принтеры) и программным ресурсам (файлы доступа к Интернет) из сети.
6. многопроцессорные
7. встроенные ОС (телевизоры, микроволновые печи).
8. опер системы для смарт-карт (самые маленькие операционные системы).

### **Ресурсы вычислительной системы**

Ресурс - любой из компонентов вычислительной системы и предоставляемые ею возможности (как аппаратные, так и программные)

Основная задача ОС – управление процессами и выделение процессам ресурсов.

#### **Ресурсы вычислительной системы (первые 2 – главные)**

1. Процессорное время
2. Объем памяти (оперативной). (Объем физической памяти (ОЗУ))
3. Внешние устройства (Устройства ввода/вывода (УВВ))
4. Ключи защиты
5. Реентерабельные коды самой системы (код чистой процедуры-процедуры, не модифицирующей саму себя, то есть данные (переменные). Из реентерабельных кодов вынесены данные.)
6. Данные. Данные в ОС находятся в специальных системных «таблицах» (условно).
7. Каналы
8. Таймер
9. Семафоры
10. Разделяемая память

## **Режимы ядра и задачи:**

С точки зрения UNIX процесс часть времени выполняется в режиме ядра (выполняется реинтегрированный код ОС, а часть – в режиме пользователя (режим задачи) (выполняется собственный код). Таким образом, процесс постоянно переключается между режимами.

Режим ядра - привилегированный режим (в нем выполняется реинтегрированный код ОС). Те части NT, которые исполняются в режиме ядра, такие как драйверы устройств и подсистемы типа Диспетчера Виртуальной Памяти, имеют прямой доступ ко всей аппаратуре и памяти. Различия в работе программ пользовательского режима и режима ядра поддерживаются аппаратными средствами компьютера (а именно - процессором).

Пользовательский режим (режим задачи) - наименее привилегированный режим, он не имеет прямого доступа к оборудованию и у него ограниченный доступ к памяти.

## **Переключение в режим ядра - классификация событий**

3 события, переводящие систему в режим ядра:

### **1. Системные вызовы**

Можно вызвать из программы с помощью команды int. Часто называются программными прерываниями (software interrupts (traps)). СВ – синхронные (по отношению к выполняемой программе) события, которые происходят в процессе работы программы.

СВ-тот интерфейс, который предоставляет пользователю ОС (Application function interface API) - набор функций, определенных в системе, которые может использовать приложение, чтобы получать сервис (обслуживание) системой. Например, ввод/вывод

ОС старается минимизировать количество СВ особенно ввод/вывод, так как они связаны с обращением к внешним устройствам, а в UNIX все файл, даже устройства, чтобы со всеми устройствами система работала единообразно (read/write)

Почему обращение к внешним устройствам – системный вызов? ОС не позволяет программам напрямую обращаться, так как иначе был бы открыт доступ к структуре ядра.

**2. Исключения (исключительные ситуации).** Делятся на исправимые (страничное прерывание) и неисправимые (ошибки (/0) ошибки деления, ошибки адресации). Являются синхронными событиями по отношению к коду

**3. Аппаратные прерывания (interrupts)** - прерывания, поступившие от устройств (от таймера, от УВВ (устройства ввода вывода) и т.д.). (поступают от контроллера прерываний) асинхронные события в системе происходят вне зависимости от какой-либо работы, выполняемой процессором

Всегда отдельно рассматривается прерывание от системного таймера – особое и единственное периодическое прерывание с важнейшими системными функциями.

В системах разделения времени – декремент кванта

Прерывание от внешнего устройства по завершении операции ввода/вывода – внешние устройства информируют процессор о том, что ввод/вывод завершен и процесс может

перейти к обработке. При этом (даже вывод) происходит получение данных об успешности или неуспешности завершения операции.

Отдельно – прерывание от действий оператора (win: ctrl+alt+delete, unix :ctrl+C)

### Процесс, как единица декомпозиции системы.

Процесс - программа в стадии выполнения.

В любой ОС основной абстракцией является процесс.

2 характерные черты процесса:

1. Процесс владеет ресурсами (Resource Ownership).
2. Процесс имеет защищенное виртуальное адресное пространство.  
Адресным пространством процесс владеет все время.

Время от времени процесс владеет следующими ресурсами: физическая память, каналы, устройства ввода/вывода, файлы и т.п.

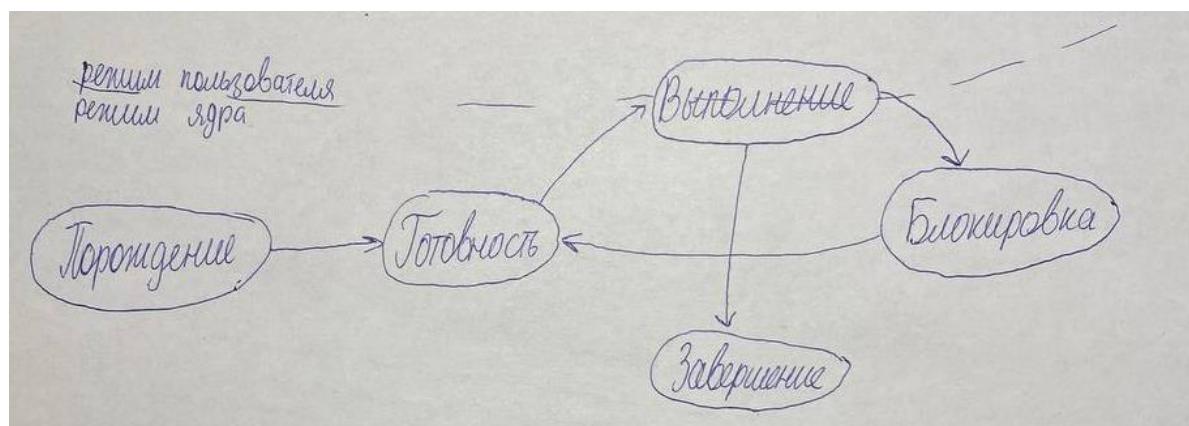
По идеологии Unix процесс часть времени выполняется в режиме задачи, и тогда он выполняет собственный код, а часть времени в режиме ядра и тогда выполняется реинтегрированный код ОС (код чистых процедур) - код, который не изменяет сам себя.

Коды ОС = ресурсы ОС. Из чистых процедур данные выносятся в соответствующие таблицы. Код может находиться в разных точках одной процедуры. Он не изменяется. Разные процессы могут исполнить этот код.

### Диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра.

Или этапы жизни процесса

(только состояния, которые есть в любой ОС, разработчики сами для себя выбирают дополнительные состояния.)



ОС – тоже программа и работает по тем же принципам. Для обработки данных нужно объявить переменную. Так и процесс должен быть определен (иметь имя)

Порождение - присвоение процессу строки в таблице процессов.

(UNIX:)

- Первый шаг системы при запуске программы – присвоение процессу идентификатора. Но надо управлять процессом (выделять ресурсы), следовательно, должна существовать структура, которая описывает процесс.
- Выделяется строка в таблице процессов: номер=идентификатор (целое число), а таблица-массив структур.
- Инициализация структуры, описывающей процесс (тех полей, которые можно)
- Выделить 1 ресурс – память.

После того, как процесс получил все необходимые ресурсы (кроме времени) – состояние готовности

В многозадачных ОС (операционных многозадачных системах пакетной обработки и системах разделения времени) в состоянии готовность одновременно может быть большое количество процессов, они организуют очередь, а в однозадачных (DOS, однозадачной пакетной обработки) – только один.

После того, как процесс в стадии готовности получил последний ресурс-время-он переходит в состояние выполнения. Из состояния выполнения он может перейти:

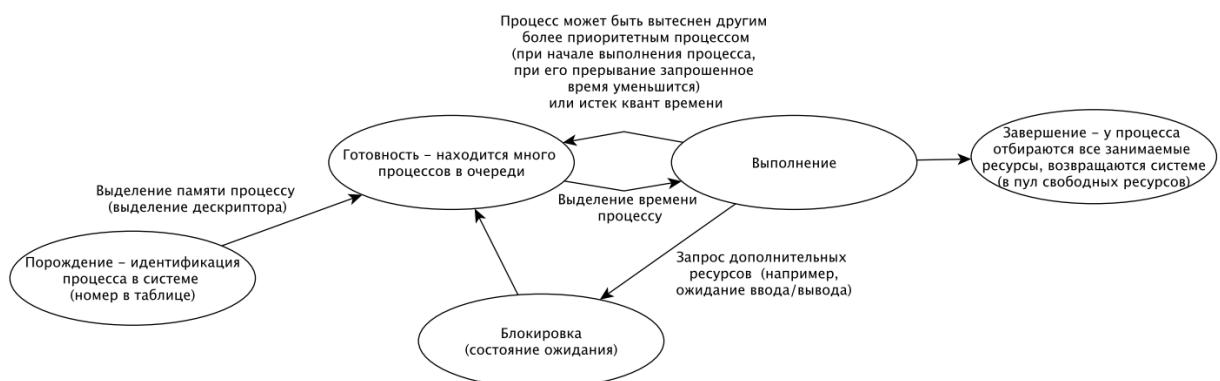
- в состояние блокировки, если он запросил ресурс, который не может быть выделен моментально, или запросил ввод/вывод. Получив необходимый ресурс, процесс перейдет в состояние готовности.
- в состояние завершения. Забираются все его ресурсы и возвращаются системе.

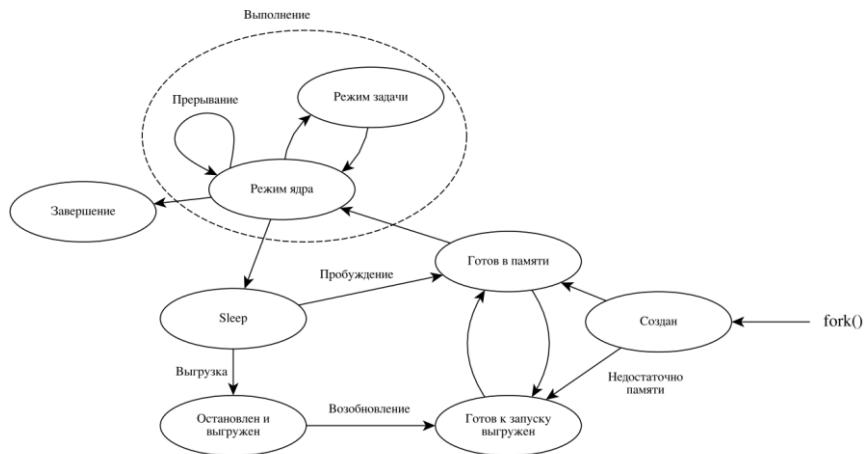
Вот именно в состоянии выполнения часть времени в режиме ядра, часть – в режиме пользователя.

При многозадачности процесс может вытесняться, если пришел более приоритетный процесс.

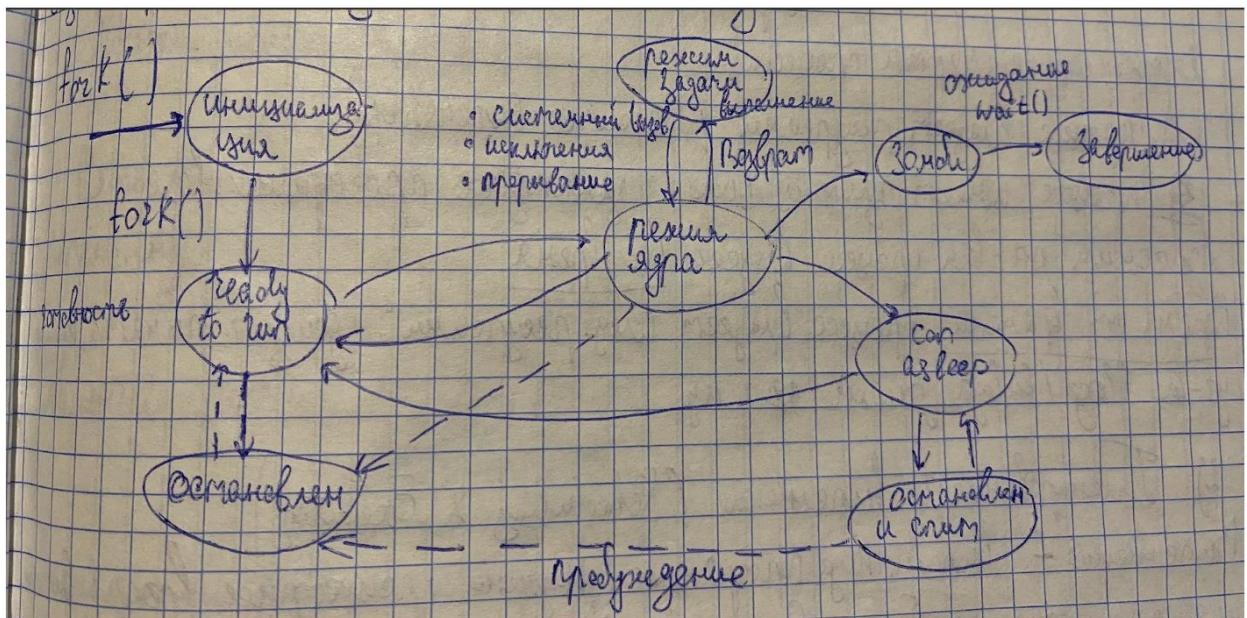
Вытеснение может произойти, если истек квант

(еще рисунки, тут можно посмотреть подписи к стрелочкам)





(если попросят еще и для юникс)



если у системы отсутствует необходимое количество памяти для нового процесса то переходит в состояние: готов, выгружен

выполняется. режиме задачи: выполняет собственной код

из состояния выполняется в режиме задачи переходит в состояние в режиме ядра при:

1. системных вызовах 2. исключениях 3. аппаратных прерываниях

если блокировка длительная, то - блокирован, выгружен (освобождает память

ядро должно переключить контекст

когда переходят из режима ядра в режим задачи, переключается только аппаратный контекст

процесс мб вытеснен другим более приоритетным процессом

зарезервирован - состояние процесса, в это состояние процесс может перевести только код ядра

### **Переключение контекста**

Переключение контекста - текущий процесс уступает процессорное время другому процессу

Переключение аппаратного контекста - сохранение содержимого регистров процессора. Такое переключение поддерживается аппаратно, для этого существует команда `pusha`. При переходе из режима ядра в режим задачи, переключается аппаратный контекст. (регистры общего назначение + IP + SP + PSW + регистры управления памятью и сопроцессора)

Переключение полного контекста - переключение аппаратного контекста и сохранение информации о выделенных процессу ресурсах (отражено в дескрипторе процесса), например, выделенная ему память, которая описывается соответствующими таблицами. Такое переключение происходит когда процесс исчерпал свой квант или происходит системный вызов, потому что системе необходимо знать информацию о тех ресурсах, которые были выделены процессу. Переключение полного контекста - затратная операция, т.к. не поддерживается аппаратно, теряется актуальность кэшей.

Аппаратный контекст `$\in\$` (является подмножеством) Полный контекст.

(Вахалия, 68 с.)

Переключение контекста - сохранение аппаратного контекста текущего процесса в области и (Блок Управления Процессом (БУП)) и загрузка аппаратного контекста другого процесса из БУП. Переключение процессора с одного процесса на другой.

В системах квантования при истечении кванта, процесс возвращается в очередь. При вытеснении происходит переключение контекста. Если процесс вытеснен или исчерпан квант, то необходимо сохранить информацию, которая поможет вернуться к продолжению выполнения.

### **Потоки**

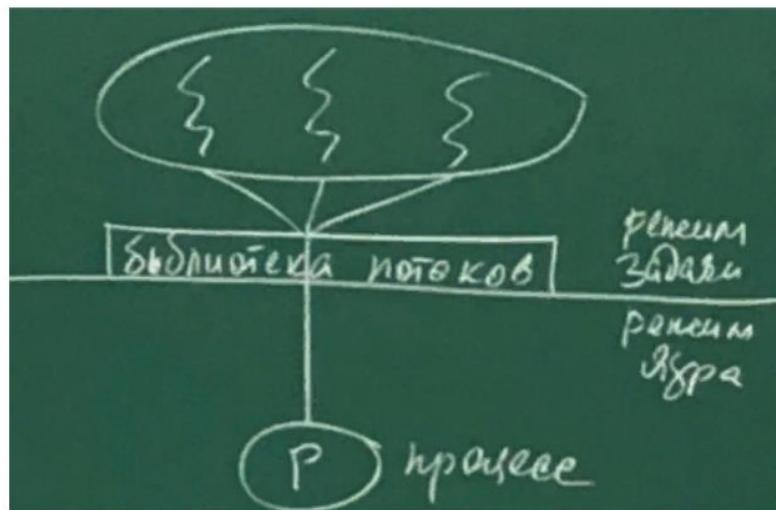
Поток - это непрерывная часть кода процесса, выполняющаяся параллельно с другими частями кода программы.

Поток не имеет своего адресного пространства, а выполняется в адресном пространстве процесса.

### Типы потоков, особенности каждого типа потоков.

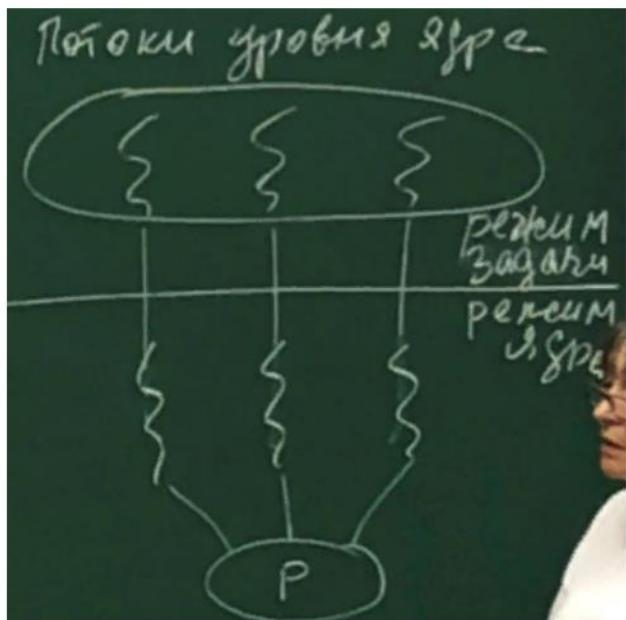
#### 1. Потоки уровня пользователя

- Поддерживаются с помощью специальной библиотеки уровня пользователя, она предоставляет функционал для работы с потоками.
- Никакой поддержки ядра нет. Пользовательские потоки (threads) - это потоки о которых ядру ничего неизвестно.
- Программа нами делится на потоки, но для ядра существует только исполняемая программа (процесс). Управление функциями библиотеки ядро не учитывает
- Библиотека должна предоставлять функции создания и удаления, планирования потоков, диспетчеризации, сохранения контекста, обеспечивать возможность взаимодействия потоков.
- Как только один поток запросил функции ядра, то весь процесс приостанавливается, пока не завершится операция ядра (например IO). Ни один другой поток не может выполняться, так как ядро ничего не знает о них



#### 2. Потоки уровня ядра

О таких потоках ядру все известно, процесс управляет потоками.



### Особенности потоков (или поток vs процесс)

(то же для потоков уровня пользователя, но тут важнее)

- Программа поделена на потоки, выполняются потоки - > процессорное время получают потоки, то есть в очередь к процессору выстраиваются потоки. Поток владеет счетчиком команд (аппаратным контекстом), поток запрашивает ресурсы
- При этом поток не имеет своего адресного пространства и выполняется в АП процесса. В результате выполнения поток запрашивает дополнительную память, но этот ресурс принадлежит процесса (Владельцем ресурсов является процесс)

Если в программе никакие потоки не создаются, то для общности создается один главный поток.

**2. Три режима компьютера на базе процессоров Intel (X86). Адресация аппаратных прерываний в защищенном режиме: таблица дескрипторов прерываний (IDT) — формат дескриптора прерываний, типы шлюзов. Пример заполнения IDT из лабораторной работы.**

### Три режима

Компьютеры на базе процессоров Intel работают в 3 режимах:

1. Реальный – 16р режим с 16-р регистрами и с 20р шиной адреса, intel 8086.  
 $2^{20}$ =позволяют адресовать до 1024Кбайт=1 Мбай

Работали под управлением DOS (disk operating system), то есть с внешней дисковой памятью. DOS – однозадачная ОС – в оперативной памяти только одна программа, которая выполняется от начала до конца.

Компьютер начинает работать в реальном режиме (чтобы выполнять меньше команд при загрузке)

В этом режиме используется сегментация памяти (адресация по типу сегмент/смещение, при этом максимальный размер сегмента - 64КБ.)

- Минимальная адресная единица памяти – байт.

2. Защищенный (protected) – 32р режим с 32р регистрами и 32р шиной адреса (4 ГБ).

- В защищенном режиме 4 уровня защиты (4 кольца привилегий). Ядро ОС находится на 0-м уровне. Пользовательские приложения находятся на 3-ем уровне
- Поддерживали 2 независимые схемы управления виртуальной (те фактически несуществующей) памятью – сегментами по запросу и страницами по запросу. Существует аппаратная схема управления памятью. (третья) сегменты, поделенные на страницы – взяли лучшее из 2 схем. Но поддерживаются только первые 2 и они независимые.
- Почему «защищенный»: Windows – система разделения времени. Адресное пространство каждого процесса должно быть защищено, как и адресное пространство ОС.

Существует специальный режим защищенного режима – v86 (virtual). Как задачи в режиме v86 запускаются ОС реального режима. Запускается виртуальная 86 машина, и в этой среде может выполняться одна программа реального режима. (таких виртуальных машин может быть несколько)

Многозадачный. Каждая запущенная виртуальная машина является v86 со всеми вытекающими (1 задача, 1Мб, 16р операнды).

*Задача исполняется с самыми низкими привилегиями в кольце 3. Когда в такой задаче возникает прерывание или исключение, процессор переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему, работающую в защищённом режиме. Прерывания обрабатываются обычными обработчиками ОС защищенного режима.*

*флаг VM регистра EFLAGS равен единице. В этом режиме происходит дизассемблирование.*

*Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима.*

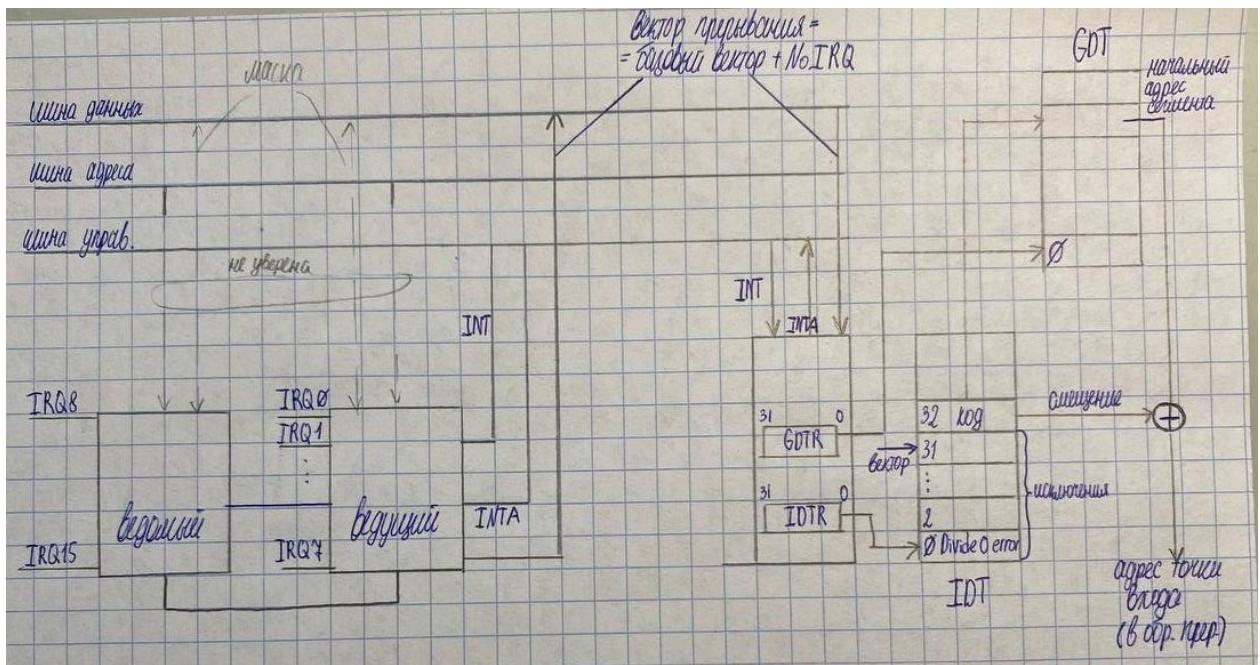
*В виртуальном режиме используется трансляция страниц памяти (в эту область могут быть отображены произвольные страницы памяти). Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме.*

3. Long (длинный) – 64р регистры, операнды. Многопроцессность. Только страничная виртуальная память. Поддерживает compatibility – режим совместимости, в котором могут выполняться 32р. Как работает обратная совместимость – рассмотреть регистры. Не поддерживает V86.

*но адреса меньше 64-х разрядов (это связано с аппаратными ограничениями).*

## Адресация аппаратных прерываний в з-р.

(если прерывание не замаскировано)



(точка входа)

IDTR-32 разрядный регистр, который содержит начальный линейный адрес IDT (все эти регистры есть в каждом ядре).

На вход контроллера приходит сигнал, контроллер формирует сигнал int, который по шине управления переходит на ножку процессора. Процессор посылает intA по шине управления в контроллер. Контроллер выставляет на шину данных вектор прерывания, который содержит селектор к IDT.

Вектор поступает в процессор. Процессор берет значение базового адреса IDT из регистра IDTR и по селектору (из вектора прерывания) находит нужный дескриптор.

Этот дескриптор содержит селектор для сегмента кода, смещение к точке входа и атрибуты. По селектору выбираем дескриптор из GDT, берём оттуда базовый адрес сегмента и прибавляем его к смещению. Получаем линейный адрес точки входа

Отдельно адресуются ведущий и ведомый контроллеры.

Зубков

Существует два контроллера прерываний.

Первый контроллер, обслуживающий запросы на прерывания от IRQ0 до IRQ7, управляет через порты 20h и 21h, а второй (IRQ8 - IRQ15) - через порты 0A0h и 0A1h. Если несколько прерываний происходят одновременно, обслуживается в первую очередь тот, у которого высший приоритет. При инициализации контроллера высший приоритет имеет IRQ0 (прерывание от системного таймера), а низший - IRQ7. Все прерывания второго контроллера (IRQ8 - IRQ15) оказываются в этой последовательности между IRQ1 и IRQ3, так как именно IRQ2 используется для каскадирования этих двух

контроллеров. В тот момент, когда выполняется обработчик аппаратного прерывания, других прерываний с низшими приоритетами нет, даже если обработчик выполнил команду *sti*. Чтобы разрешить другие прерывания, каждый обработчик обязательно должен послать команду *EOI* - конец прерывания - в соответствующий контроллер..

еще инфа

В ЗР для адресации прерывания имеется специальная таблица *IDT*. Если первые 32 исключения, и мы возьмем 8, то попадем на *double fault*. Чтобы адресовать прерывания, надо перепрограммировать контроллер на: ведущий-на базовый вектор 32, и этот номер использовать для обращения к таблице. Базового адреса нет, есть смещение и селектор. Отдельно адресуются ведущий и ведомый. От контроллера они могут получить маску??

В РР процессор использует вектор и таблицу векторов прерываний. У ведущего контроллера базовый вектор=8 ( $8+0=8h$ ) и номер используется для получения смещения к адресу в таблице векторов прерываний. В DOS адрес наз. вектор (4 байт)

В з-р для каждого сегмента программы должен быть определен дескриптор - 8-байтовое поле, в котором в определенном формате записываются базовый адрес сегмента, его длина и некоторые другие характеристики. В р-р сегменты определяются их линейными адресами. (*P-Ф*).

### Таблица дескрипторов прерываний (IDT)

*IDT* (interrupt descriptor table) - системная таблица, предназначенная для хранения адресов обработчиков прерываний. (нужна для адресации обработчиков прерываний)

Первые 32 элемента таблицы - под исключения (синхронные события в процессе работы программы) (внутренние прерывания процессора) (в 386 всего 19 исключений (0-19), остальные (20-31) зарезервированы, а в 486 – и того меньше (реально-18, остальные-зарезервированы)

32-255 – определяются пользователем (user defined)

Смещение=номер исключения\*8

Нам надо адресовать 2 обработчика – таймера и клавиатуры (аппаратные прерывания) через программирование контроллера прерываний. Сейчас прерывания как MSI (message signal interrupts)

- 0-divide error (ошибка деления на 0)
- 8-double fault (если выполнить исключение или маскируемое/немаскируемое прерывание и возникла ошибка (паника...), завершается работа компьютера)
- 11-segment not present (сегмент отсутствует – надо выполнить определенные действия, чтобы сделать сегмент доступным. Касается управления памятью (нашей программе-не очень))
- 13-general protection (общая защита, должно быть обработано специальным образом. На все исключения-заглушки (double fault-не искл.), а на 13-специальная заглушка (у РФ отражено в структуре таблицы дескрипторов прерываний-без dup))

(нарушение общей защиты (нарушение, код ошибки-та команда), происходит: за пределами сегмента, запрет чтения, за гр. таблицы дескр., int с отс. Номером)

- 14-page fault (fault переводится как исключение, но по-русски здесь прерывание) (страничное прерывание) – обращение к команде/данным, отсутствующим в программе – система должна загрузить нужную страницу. В CR2 адрес, на котором произошло прерывание)

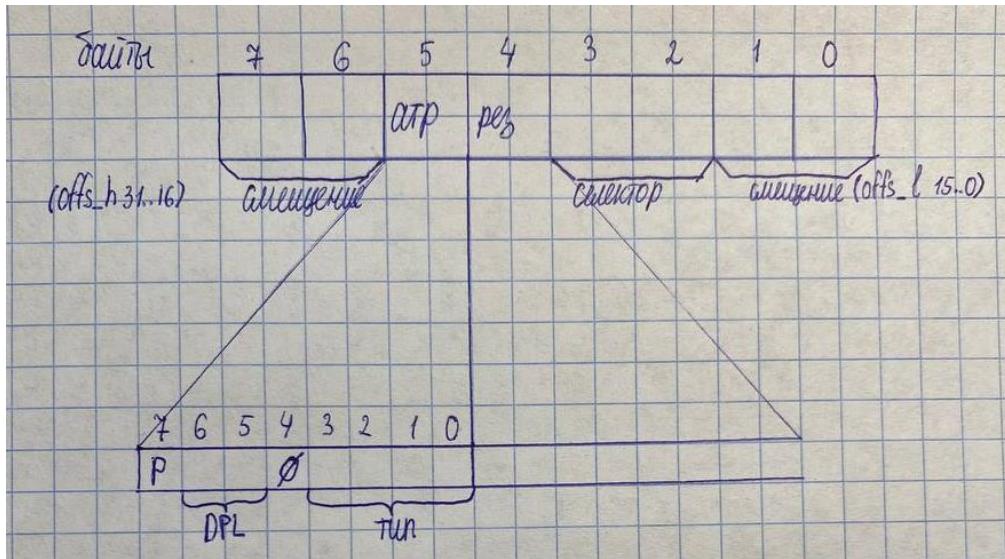
Вектор исключения	Название исключения	Класс исключения	Код ошибки	Команды, вызывающие исключение
0	Ошибка деления	Нарушение	Нет	div, idiv
1	Исключение отладки	Нарушение /ловушка	Нет	Любая команда
2	Немаскируемое прерывание			
3	int 3	Ловушка	Нет	int 3
4	Переполнение	Ловушка	Нет	into
5	Нарушение границы массива	Нарушение	Нет	bound
6	Недопустимый код команды	Нарушение	Нет	Любая команда
7	Сопроцессор недоступен	Нарушение	Нет	esc, wait
8	Двойное нарушение	Авария	Да	Любая команда
9	Выход сопроцессора из сегмента (80386)	Авария	Нет	Команда сопроцессора с обращением к памяти
10	Недопустимый сегмент состояния задачи TSS	Нарушение	Да	jmp, call, iret, прерывание
11	Отсутствие сегмента	Нарушение	Да	Команда загрузки сегментного регистра
12	Ошибка обращения к стеку	Нарушение	Да	Команда обращения к стеку
13	Общая защита	Нарушение	Да	Команда обращения к памяти
14	Страницочное нарушение	Нарушение	Да	Команда обращения к памяти
15	Зарезервировано			
16	Ошибка сопроцессора	Нарушение	Нет	esc, wait
17	Ошибка выравнивания	Нарушение	Да	Команда обращения к памяти
18...31	Зарезервированы			
32...255	Предоставлены пользователю для аппаратных прерываний и команд int			

### Типы шлюзов.

1. Ловушки (trap gate) - обработка исключений и программных прерываний (системных вызовов).
2. Прерывания (interrupt gate) - обработка аппаратных прерываний.
3. Задач (task gate)- *переключение задач в многозадачном режиме.*

### Формат дескриптора прерывания

Формат дескриптора (шлюза) для IDT (в скобках – из учебника)



- Байты 0-1 (offs\_1), 6-7 (offs\_h): 32-битное смещение обработчика
- Байты 2-3 (sel): селектор (сегмента команд) (итого полный 3-хсловный адрес обработчика селектор: смещение)
- Байт 4 зарезервирован
- Байт 5: байт атрибутов - как в дескрипторах памяти за исключением типа:

Типы: назначение:

- 0-не определен
- 1-свободный сегмент состояния задачи TSS 80286
- 2-LDT
- 3-занятый сегмент состояния задачи TSS 80286
- 4-шлюз вызова Call Gate 80286
- 5-шлюз задачи Task Gate
- 6-шлюз прерываний Interrupt Gate 80286
- 7-шлюз ловушки Trap Gate 80286
- 8-не определен
- 9- свободный сегмент состояния задачи TSS 80386+
- Ah-не определен
- Bh- занятый сегмент состояния задачи TSS 80386+
- Ch-шлюз вызова Call Gate 80386+
- Dh- не определен
- Eh-шлюз прерываний Interrupt Gate 80386+
- Fh- шлюз ловушки Trap Gate 80386+

Может принимать 16 значений, но в IDT допустимо 5: 5(задачи), 6(прерываний 286), 7(ловушки 286), Eh(прерываний 3/486), Fh(ловушки 3/486) (это по РФ)

- 4-0 (а вообще это S - system (0 - системный объект, 1 - обычный))
- 5-6-DPL - уровень привилегий (0 - уровень привилегий ядра, 3 - пользовательский/приложений, 1-2 - не используется в системах общего назначения)
- 7-1 (а вообще это P - бит присутствия (1 - если сегмент в оперативной памяти, 0 - иначе))

## Пример заполнения IDT из лабораторной работы.

```
;Структура idescr для описания дескрипторов (шлюзов) прерываний
idescr struc
    offs_l    dw 0
    sel       dw 0
    cntr     db 0
    attr      db 0
    offs_h    dw 0
idescr ends

;Таблица дескрипторов прерываний IDT
;Дескриптор: <offs_l, sel, rsrv, attr, offs_h>
;смещение позже?, селектор 32-разрядного сегмента кода
idt label byte

; Первые 32 элемента таблицы - под исключения-внутренние прерывания процессора
; (реально-18, остальные-зарезервированы)
;attr=8Fh: тип=ловушка 386/486(обр. программные пр. и искл., IF не меняется),
;системный объект, УП ядра, P=1
idescr_0_12 idescr 13 dup (<0,code32s,0,8Fh,0>)
; исключение 13 - нарушение общей защиты (нарушение, код ошибки-та команда)
; происходит: за пределами сегмента, запрет чтения, за гр. таблицы дескр., int с отс. номером
idescr_13 idescr <0,code32s,0,8Fh,0>
idescr_14_31 idescr 18 dup (<0,code32s,0,8Fh,0>

; Затем 16 векторов аппаратных прерываний,
;attr=8Eh: тип=прерывание 386/486(обр. аппаратные пр., IF сбрасывается а iret восстанавливает),
;системный объект, УП ядра, P=1
;дескриптор прерывания от таймера
int08 idescr <0,code32s,0,8Eh,0>
int09 idescr <0,code32s,0,8Eh,0>

idt_size = $-idt           ;размер
ipdescr df 0               ;псевдодескриптор
ipdescrl6 dw 3FFh, 0, 0    ;содержимое регистра IDTR в PP: с адреса 0, 256*4=1кб=2^10

; Вносим в дескрипторы прерываний (шлюзы) смещение обработчиков прерываний.
lea eax, es:except_13
mov idescr_13.offs_l, ax
shr eax, 16
mov idescr 13.offs_h, ax
```

## 2 билет

2.1 Классификация операционных систем. Особенности ОС определенных типов.  
Виртуальная машина и иерархическая машина – декомпозиция системы на уровни иерархии, иерархическая структура Unix BSD, архитектуры ядер ОС – определение, примеры.

## Классификация операционных систем. Особенности ОС определенных типов

## 1. Однопрограммная пакетной обработки

В оперативной памяти может быть только 1 прикладная программа. Режим работы ЭВМ: однопрограммный/однозадачный.

## 2. Мультипрограммная пакетной обработки

*В оперативной памяти одновременно много программ. Загрузка – перфокартами. Программа располагается в памяти целиком; процессорное время выделяется по принципу приоритетов. Планирование на принципе распараллеливания функций. Программа выполняется до тех пор, пока не запросит доп. ресурс системы или пока не придет более высокоприоритетный процесс, в случае поддержки вытеснения, может вытеснить. Режим работы ЭВМ: мультипрограммный/мультизадачный/многозадачный*

## 3. Системы разделения времени

В оперативной памяти одновременно находится большое число программ, процессорное время квантуется. Процесс выполняется до тех пор, пока не истек квант, не начался процесс ввода/вывода или не вытеснен другим высокоприоритетным процессом. Режим работы ЭВМ: однопрограммный или мультипрограммный

## 4. Системы реального времени

POSIX определение реального времени в ОС (стандарт 1003.1)

Реальное время в операционных системах – это способность операционной системы обеспечить требуемый уровень сервиса в определенный промежуток времен, величина которого определяется особенностями работы внешнего по отношению к вычислительной системе устройства или процесса. Время отклика системы  $\leq$  время поступления запроса на ответ.

В отличие от систем общего назначения ОСРВ обеспечивает ответ системы (или сервис системы) за определенный промежуток времени, то есть обслуживание запроса. Это всегда запрос или внешний по отношению к системе или внешнего процесса. 2 процесса реального времени в наших компьютерах – видео и аудио.

Жесткое реальное время – это когда интервал установлен жестко и не может быть превышен. Мягкое реальное время – возможность небольших отклонений от величины интервала.

Пакет – набор программ, которые одновременно загружены в память.

*И еще вроде*

5. серверные ОС – предоставляющие доступ к аппаратным (принтеры) и программным ресурсам (файлы доступа к Интернет) из сети.

6. многопроцессорные

7. встроенные ОС (телевизоры, микроволновые печи).

8. опер системы для смарт-карт (самые маленькие операционные системы).

## 9. Персональные операционные системы

*В многозадачных ОС (операционных многозадачных системах пакетной обработки и системах разделения времени) в состоянии готовности одновременно может быть большое количество процессов, они организуют очередь, а в однозадачных (DOS, однозадачной пакетной обработки) – только один.*

*Дисциплины планирования связаны с соответствующими типами ОС (к однозадачным не относится).*

*Два вида: мультизадачные системы пакетной обработки и системы разделения времени (мультизадачные во втором случае нельзя говорить)*

*В системах пакетной обработки пользователь отделен от процесса выполнения программы. Рисунок (человек {пакет} квадрат). В системах разделения времени – наоборот: пользователь непосредственно связан – запускает, вводит данные, получает результат и реагирует – интерактив.*

*Иерархическая структура unix BSD 4.4. На самом нижнем – диспетчеризация – выделение процессам процессорного времени (в срв или пакетной обработки)*

## **Виртуальная машина и иерархическая машина – декомпозиция системы на уровнях иерархии**

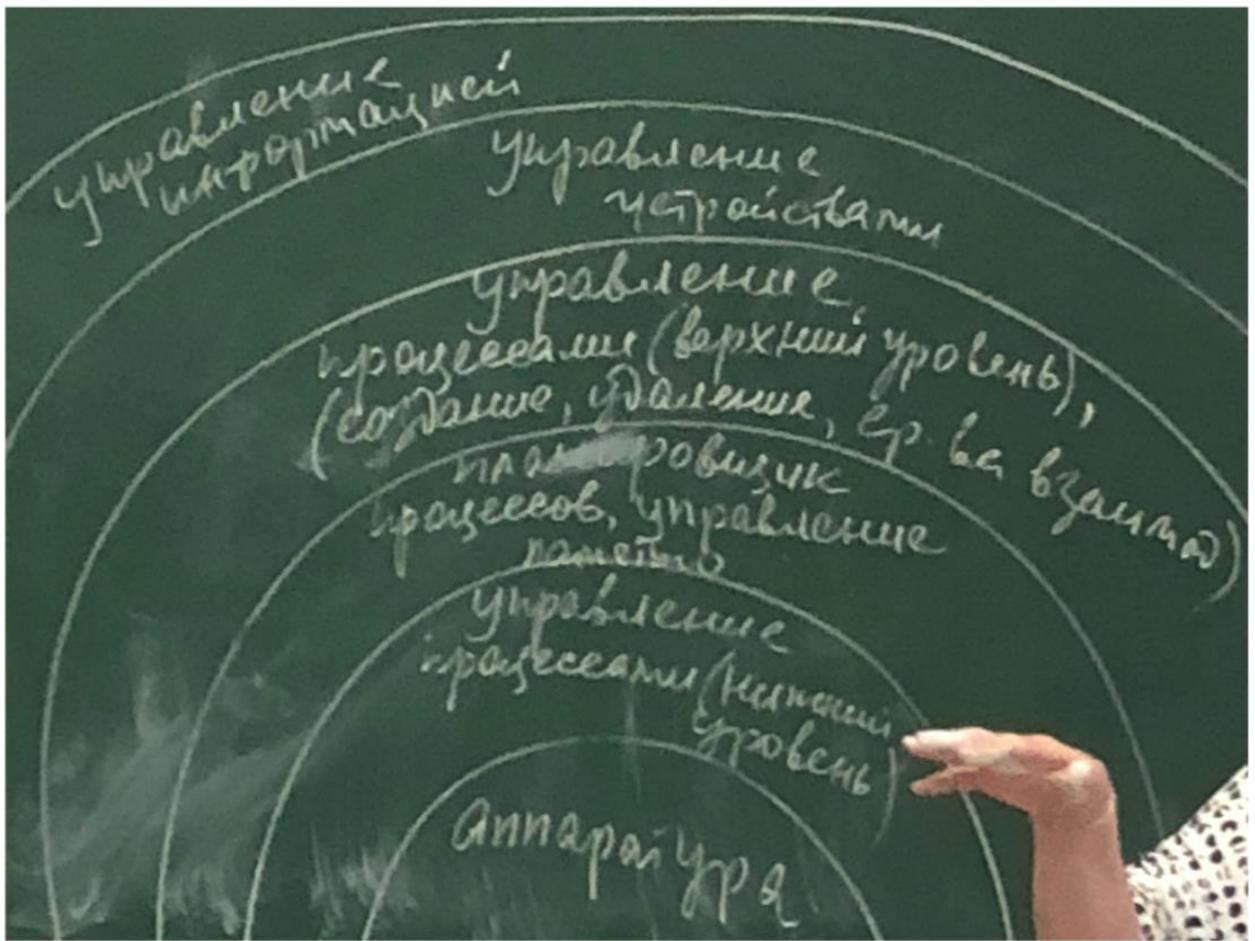
Виртуальная машина – кажущаяся возможной. Виртуальная машина – набор команд и функций, необходимых пользователю для получения сервиса операционной системы

Иерархическая машина (Медника-Донована).

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

Между уровнями определяется интерфейс взаимодействия. Интерфейс – функции, которые нижние уровни предоставляют верхним уровням.

- прозрачные - позволяет обращаться через уровни
- полупрозрачные - какие-то обращения через уровни возможны (какие-то только к низлежащим уровням)
- непрозрачные - обращения возможны только к низлежащим уровням



- Управление процессами (нижний уровень) – выделение процессу процессорного времени.
- уровень 2 - (P, V) - семафорные операции, контр. доступа к разделяемым ресурсам, планировщик процессов (операция более высокого уровня. постановка процессов в очередь к процессору или другим ресурсам), управление памятью; Процессор, оперативка, внешние устройства - аппаратные устройства
- Управление процессами (верхний уровень) – создание, уничтожение, взаимодействие при помощи сообщений. Система принимает решение об этом, более высокоуровневая операция
- уровень 4 - управление устройствами (подсистема ввода/вывода);
- уровень 5 - управление информацией (файловая система). Система предоставляет такие средства как наименование файлов, построения дерева каталогов, удобных для пользователя.

Виртуальная машина - совокупность команд машины и команд ОС, которые могут использовать программы для получения сервиса ОС.

## Иерархическая структура unix BSD 4.4

Таблица 1: Структура ядра ОС UNIX 4.4 BSD

Системные вызовы					/ / / / /	Сис. выз.	/ / / / / / / / / /	
Управление терминалом		1	2	3	4	Сокеты	Обработка сигналов	Создание и завершение процесса
Необработанный телетайп	Обработанный телетайп	Файловая система	Виртуальная память	Сетевые протоколы	Маршрутизация	Планирование процессов		
Драйверы символьных устройств	Дисциплины линий связи	Буферный КЭШ	Страницочный КЭШ	Драйверы блочных устройств	Драйверы сетевых устройств	Диспетчеризация процессов		

## 1. Символьный уровень

### 2. Именование

### 3. Отображение адреса

### 4. Страницочное прерывание

/ - аппаратные и эмулируемые прерывания

Юникс хорошо структурирована, видим в части процессов все то же, что у медника-денона (там только не было сигналов). Но в Юникс сигналы информируют процессы о событиях, которые происходят в системе – очень важно.

На самом нижнем – диспетчеризация – выделение процессам проц времени (в срв или пакетной обработки)

Более высокий уровень – планирование – определение, в какой последовательности будут выполняться процессы. Если реализуется приоритетное планирование (при этом учет времени простоя в очереди готовых процессов) – приоритеты будут динамическими и изменяться в зависимости от времени простоя (и бывает от количества полученного процессорного времени). То же делают и винды, но не по формуле, а просто сканируя.

Еще более высоко – создание и завершение процессов. Порождение – идентификация, выделение и инициализация дескриптора. Завершение – действие, в результате которого все занимаемые процессом ресурсы возвращаются системе.

Также на нижнем уровне находятся драйверы – ПО, которое напрямую взаимодействует с аппаратной, предназначено для управления внешними устройствами.

Два типа устройств – символьные (сетевые в тд) и блочные (магнитные диски, одна из функций – поддержка виртуальной памяти – совокупность всех АП процессов, которые в данный момент выполняются на компе).

Страницочный и буферный кеш (относится к файловой системе, так как все буферизуется).

## архитектуры ядер ОС – определение, примеры.

Существует два класса ядер:

1. Монолитные ядра (более ранняя)

2. Микроядра

### Монолитное ядро

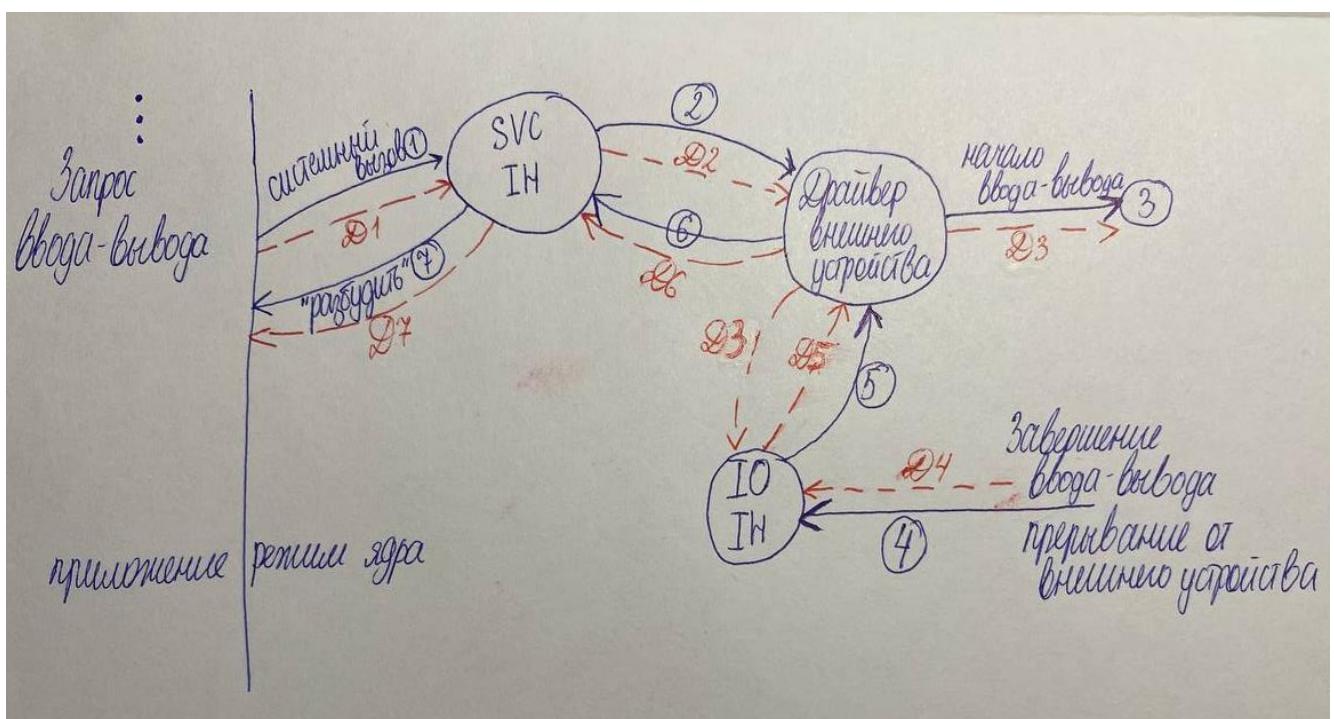
Это программа, имеющая модульную структуру, то есть состоящая из подпрограмм.

Системы Windows, Unix, Linux имеют монолитные ядра. Unix имеет минимизированное ядро (вынесен графический интерфейс).

В архитектуре с монолитным ядром все услуги для прикладного приложения выполняют отдельные части кода ядра (в адресном пространстве ядра)

Все построено на прерываниях – системные вызовы, исключения и аппаратные прерывания.

Последовательность действий в системе при запросе приложения на ввод/вывод



Происходит вызов read/write и система переходит в режим ядра (ни одна система не дает прямое обращение к внешним устройствам)

Действия (красным на схеме) (обратить внимание, что у ДЗ две стрелочки)

(на лекции все писалось сплошным текстом, но вроде если раскидать по действиям, то получится так)

1. Системный вызов из функции, который переводит в режим ядра, туда же соответствующие данные. SVC - Supervisor call. IH- interrupt handler. Супервизор - ... в стадии выполнения.
2. В результате обработки системного вызова будет вызван драйвер внешнего устройства (программа ввода/вывода). Драйверу будут переданы данные в его формате.
3. Драйвер инициализирует работу внешнего устройства. На этом управление процессором работой заканчивается, он отключается, потому что работой внешнего устройства управляют контроллеры.
4. По завершении операции ввода/вывода контроллером устройства будет сформировано прерывание, которое в простой схеме поступит на контроллер прерывания (IO IH) и в результате будет определен адрес точки входа обработчика прерывания.
5. Процессор перейдет на выполнение обработчика, тк аппаратные прерывания имеют наивысший приоритет. Обработчики прерываний входят в состав драйвера и является одной из точек входа драйвера внешнего устройства. Драйвер всегда содержит 1 обработчик прерывания.
6. Поскольку обработчик – точка входа драйвера, у драйвера есть call back функция – задача вернуть запрашиваемые данные приложению. В результате драйвер через подсистему ввода/вывода должен передать данные приложению.
7. Для этого работа приложения db возобновлена (разбудить). *Процесс, который запросил вв блокируется.*

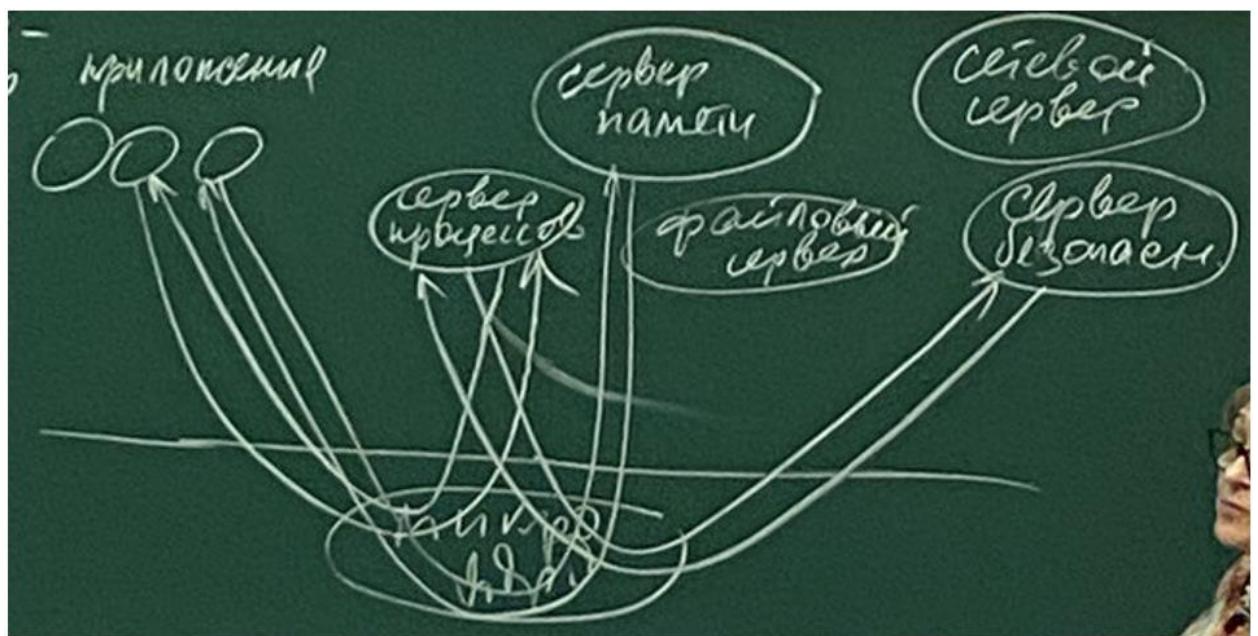
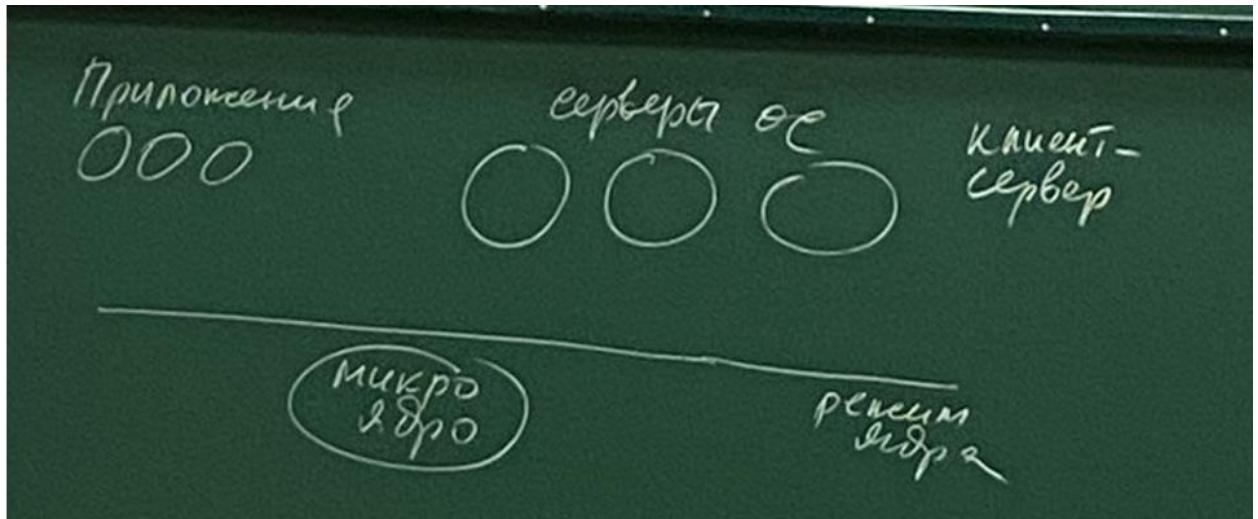
С монитором мы работаем как с памятью – mov. А input/output – использование команд ввода вывода, и это приводит к блокировкам.

Возникновение прерывания происходит асинхронно. Чтобы получить значение, процесс разблокируется. поэтому эта схема называется блокирующий синхронный ввод-вывод

## Микроядерная архитектура

Mach – первая ОС с микроядром. Классическим примером микроядерной ОС является Symbian OS.

Идея мя – оставить в ядре только самые низкоуровневые функции, остальные функции выполняются в режиме пользователя. Поскольку остальная часть ОС реализуется в виде отдельных процессов в собственных АП, то взаимодействие между компонентами ОС выполняются с помощью посылки и приема сообщений, причём этот механизм обеспечивается специальным модулем ядра, который называется микроядро.



Современные ОС предоставляют возможность внесения в монолитное ядро собственных ... без перекомпиляции. В Unix – загружаемые модули ядра. С помощью них можно сделать не все. Если надо изменить структуру ядра – надо перекомпилировать. Поэтому идея микроядра – вносить изменения в ОС, имея широкие возможности и не перекомпилировать.

Программы ОС принято называть серверами ОС. Суть – та же. Монолитное ядро предоставляет приложениям сервис (с помощью системных вызовов).

Взаимодействие микроядреной архитектуры по модели клиент-сервер.

Модель предполагает, что программы обращаются к серверу с к-либо запросами, эти запросы сервер обрабатывает, в результате формируется ответ, ожидаемый клиентом. Любая такая архитектура предполагает взаимодействие по протоколу – соглашение.

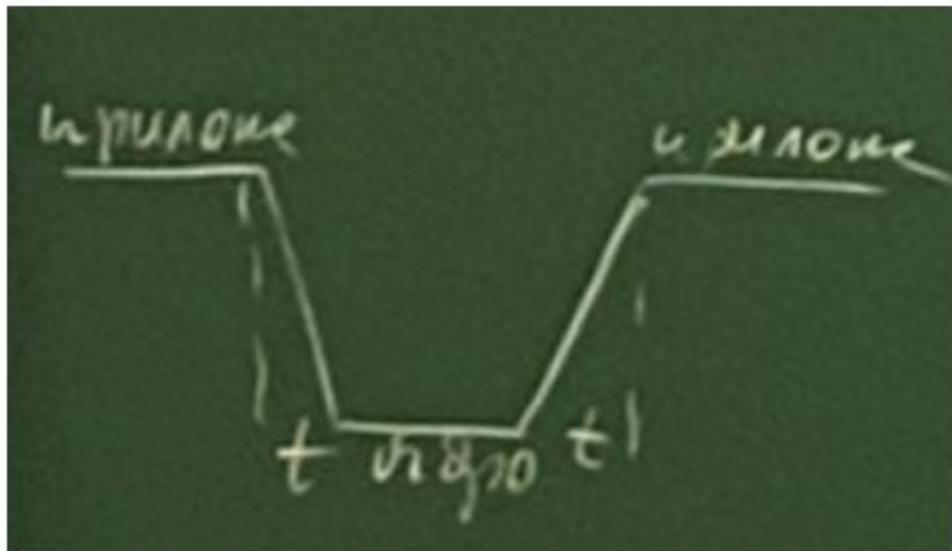
Такое взаимодействие должно быть надежным – посылка сообщения должна подтверждаться сообщением о его приеме. В результате мы рассматривали диаграмму – 3 состояния

блокировки при передаче сообщений. Все 3 состояния будут присутствовать – блокирован при посылке, блокирован при ответе, блокирован при приеме.

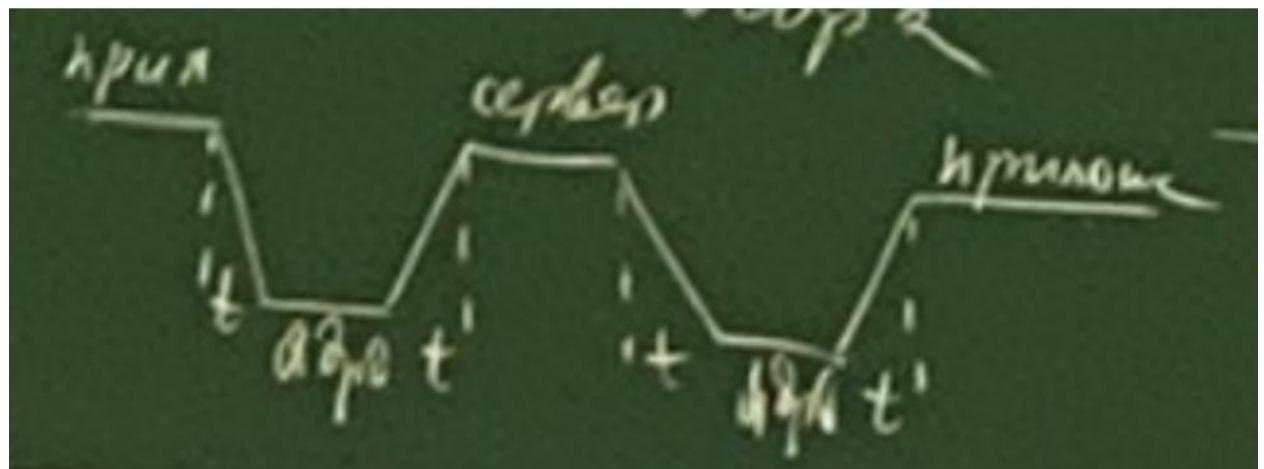
Это очень важная диаграмма состояний, напрямую связана с эффективностью микроядерной архитектуры.

### Эффективность

В монолитном ядре при обработке системного вызова будет выполнено 2 переключения – 1 из режима задачи в режим ядра, 2 – обратно.



В мя – минимум 4



Но есть вероятностные временные затраты – блокировки при приеме и ответе – которые оценить невозможно.

Несмотря на то, что эффективность мя архитектуры намного ниже, интерес не утрачивается. В чем же привлекательность.

В том, что большая часть кода ядра вынесена в режим пользователя и мб изменена без перекомпиляции ядра.

Но как пишут соломон с русиновичем, для коммерческих реализаций мя mach перестает быть уже таким микро. В частности, файловая подсистема, поддержка сетей и управление памятью в коммерческих системах mach выполняется в режиме ядра как в системах с монолитным ядром.

Причина проста: системы, построенные строго по принципу мя плохи с коммерческой тз из-за низкой эффективности. Но мя используется в системах реального времени. Например, широко известная срв QNX построена на основе мя архитектуры.

[2.2 Три режима работы вычислительной системы с архитектурой X86: особенности.](#)  
[Реальный режим: линия A20 – адресное заворачивание. Перевод компьютера в защищенный режим. Линия A20 в защищенном режиме: включение и выключение линии A20 \(код из лабораторной работы\).](#)

### Три режима

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHthsB8TTy1Xvv9SN9zM/edit#bookmark=id.6qoibo7d324a>

### Линия a20

Рассмотрим 2 спецификации (в России – гости).

XMS	EMS
extended memory specification	expanded memory specification
Дополнительная память	Растянутая (расширенная) память
	У нее много версий, начинаются словами LIM...
Оговаривает область памяти и в защищенном режиме это будут следующие области	Что-то там ... страниц Это позволяет растянуть память. Это называется виртуализация и мы получаем дополнительный что-то там...

С появлением 8086 стала доступна память 1МБ – upper memory area. Использовалась в 8086 под управлением dos, для нее были опубликованы карты первого мбайта. Память сверх одного мбайта –ХМА. В защищенном режиме 32 р шина адреса – смещения могут достигать 4 гб – это 8 ф FFFFFFFF

## **адресное заворачивание**

В РР было 20 адресных линий и 2 типа адресного заворачивания:

- Если у нас 20 единиц – 5 ф и мы прибавили еще 1, то все выльется, «со стола не сlijжешь», становятся 5 нулей и адрес начинает указывать на младшие адреса.
- Второй тип – в памяти сегменты. Максимально смещение 64Кбайт: Ffff+1=0000 – указывает на младшие адреса сегмента.

Нас интересует первое.

Для обеспечения обратной совместимости (аппаратно поддерживает обратную совместимость) создан порт линии а20. У нас 32 адресные линии, поэтому это уже не 20 линия.

В реальном режиме линия а20 закрыта. Она на 0 потенциале, заземлена. Когда переводим в зр, ее надо открыть. Что будет, если забудем - то получаем битую память. Адреса с 1 в этом бите нам не будут доступны. При переходе обратно, надо закрыть. Если не закроем – она нам будет доступна.

## **НЮАНС**

*Если мы в реальном режиме на компы 32 или 64 разрядные – можем поставить dos и комп будет работать под его управлением.*

*Если мы в реальном режиме откроем линию а20, нам станет доступно еще 64 кбайт памяти- high memory area. Но это в реальном режиме. В защищенным режиме нам доступно 4 ГБ.*

*«На хабрах ваших вонючих» пишут про теневые регистры еще. РФ пишет ffff в теневые регистры, но старшие 4 разряда не обнуляют. «Идиоты». Если остались действительно 4 гб сегменты и оставить, то будет доступно 4 гб – «жуть и чушь». Как мы будем их адресовать??*

*To есть если не запишем в limit ffff – криминала никакого.*

## **включение и выключение линии А20**

Существует 2 кода, которые используют при работе с а20:

## **Открыть**

Mov al, 0D1h ; команда управления линией a20

Out 64h, al

Mov al, 0DFh ; код открытия линии a20

Out 60h, al

**или**

In al, 0x92

Or al, 2

Out 0x92, al

Начиная с PC/2 имеется быстрый (2) вариант открытия линии a20. Он исключает опрос. Быстрый вариант считается не вполне надежным и поддерживается не всеми платформами. Невозможно убедиться в этом заранее. Поэтому пользоваться надо 1 вариантом.

**Закрыть линию a20.** Та же команда управления

Mov al, 0D1h ; команда управления линией a20

Out 64h, al

Mov al, 0DDh ; код закрытия линии a20

Out 60h, al

## **Перевод компьютера в защищенный режим**

Чтобы перейти в защищенный режим, достаточно установить бит PE - нулевой бит в управляющем регистре CR0, и процессор немедленно окажется в защищенном режиме. Единственное дополнительное требование, которое предъявляет Intel, - в этот момент все прерывания, включая немаскируемое, должны быть отключены (Зубков, стр. 488).

1. Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того, как она переключится в защищённый режим.  
Впоследствии, находясь в защищённом режиме, программа может модифицировать GDT (если, она в нулевом кольце защиты).
2. Подготовить в оперативной памяти таблицу дескрипторов прерываний IDT.
3. Записать линейные физические адреса в дескрипторы сегментов.

4. Загрузить адрес и размер GDT в GDTR.
5. Загрузить смещения обработчиков прерываний в шлюзы.
6. Сохранить маски прерываний.
7. Перепрограммировать ведущий контроллер прерываний на новый базовый вектор (32).
8. Запретить все маскируемые и немаскируемые прерывания.
9. Открыть адресную линию A20.
10. Загрузить адрес и размер IDT в IDTR.
11. Перейти в защищенный режим (установить бит PE — нулевой бит в управляемом регистре CR0 в 1).
12. Загрузить новый селектор в регистр CS.
13. Загрузить сегментные регистры селекторами на соответствующие дескрипторы.
14. Разрешить прерывания.

### З билет

3.1 Прерывания: классификация. Последовательность действий при выполнении запроса ввода-вывода. Обработчики аппаратных прерываний: виды и особенности. Функции обработчика прерываний от системного таймера в ОС семейства Windows и семейства Unix.

#### Прерывания: классификация

1. Системные вызовы (программные прерывания)
  - вызов, когда требуется сервис системы (ввод/вывод, обращение к внешнему устройству)
  - клавиатура, мышь, вторичная память (флешки, диски)
  - ни одна система не позволяет напрямую процессом обращение к устройствам ввода-вывода, если разрешить такие обращения, то систему защитить невозможно.
  - Система предоставляет соответствующие функции (примитивы, т. к. низкоуровневые (ядро) действия) (API)
  - синхронные события.
2. Исключительные ситуации

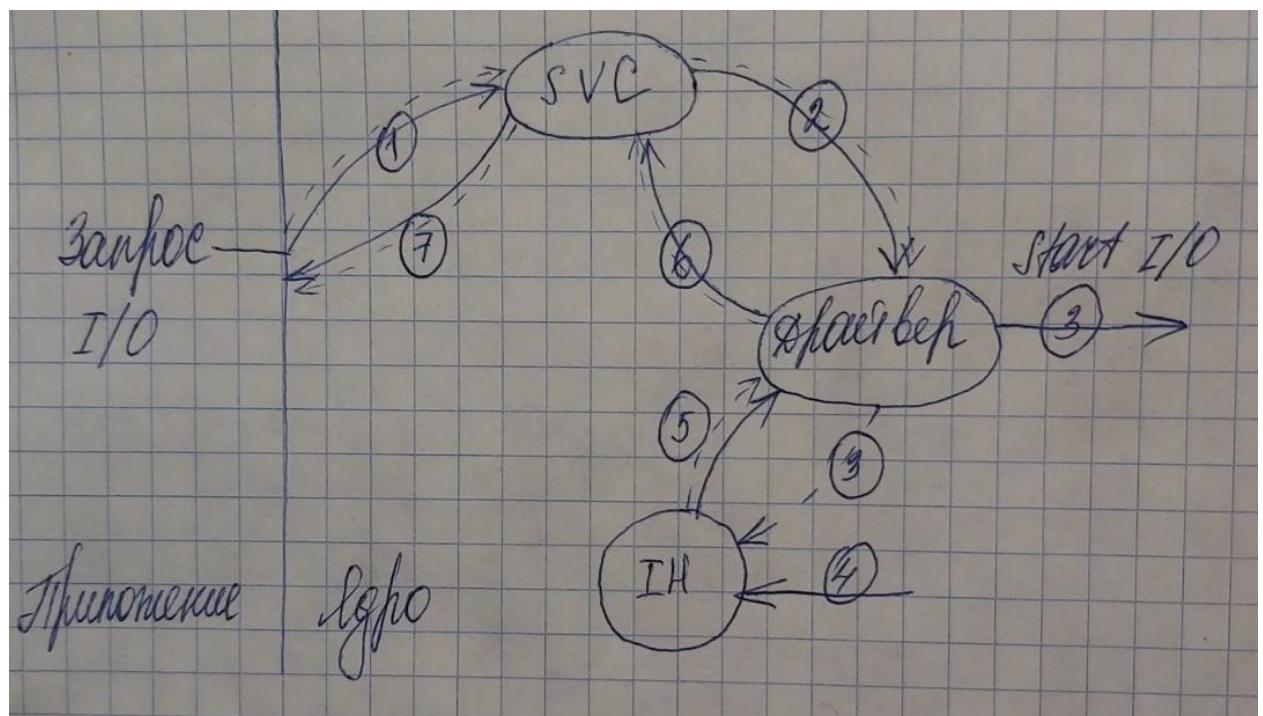
- прерывание выполняемой программы при возникновении исключения
- (исправимые – страничное прерывание; неисправимое – деление на 0)
- синхронные события.

### 3. Аппаратные прерывания

- Бывают нескольких типов:

- Прерывания системного таймера (важнейшие функции)
- Прерывание от внешних устройств (возникают по завершению операции ввода/вывода)
- Действия оператора (ctrl-alt-del)
- Асинхронные события, возникают независимо от каких-либо действий, которые выполняются в системе.

### Последовательность действий при выполнении запроса ввода-вывода

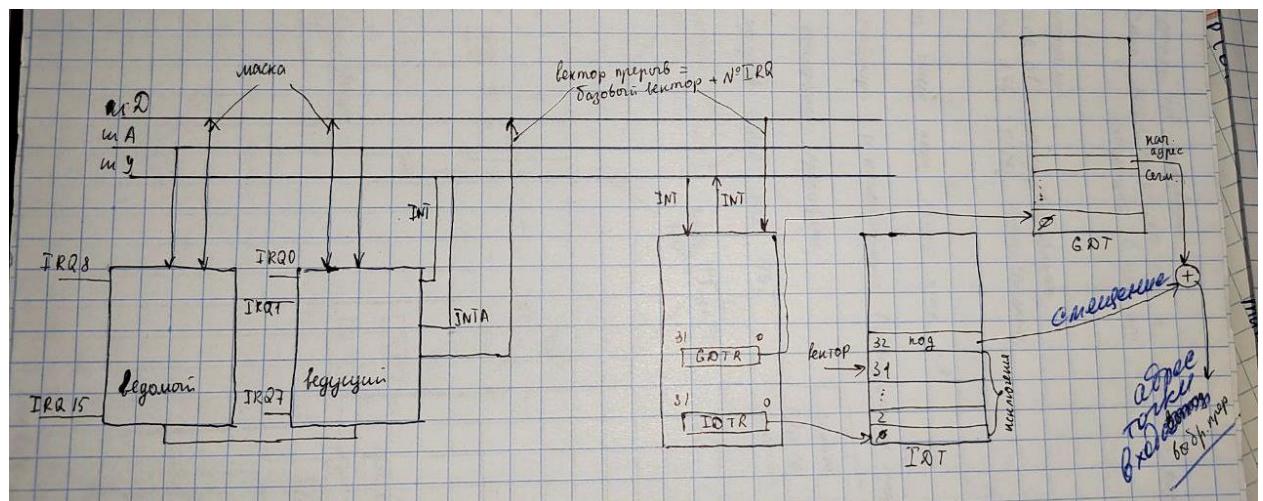


Стадии:

1. Запрос ввода-вывода
2. SVC - supervisor call, код операционной системы в стадии выполнения, переход в ядро (штриховая линия - передаются данные) (выполняется IH (interrupt handler))
3. Вызывается драйвер соответствующего устройства (в нужном формате).  
StartIO  
Прерывание
4. IH (получает данные от драйвера и в результате будут сформированы данные, которые передадутся обратно в драйвер)

5. Драйвер (будет вызвана другая часть драйвера (передачи полученных данных от внешнего устройства приложению: Read - код символа, write - информация об успехе/неуспехе вывода))
6. Чтобы приложение могло продолжить выполняться, оно должно быть выведено из состояния непрерываемого сна.
7. После выхода из сна данные записываются из буфера ядра в буфер приложения.

на всякий



- IRQ1 – сигнал от клавиатуры.
- Когда завершается операция ввода/вывода, в буфер клавиатуры записывается код, контроллер отправляет его на ножку IRQ1. Формируется запрос INT и по шине управление поступает в процессор.
- В конце цикла выполнение каждой команды процессор проверяет наличие сигнала прерывания и переходит в обработчик, прежде посыпая через INT A сигнал.
- Получив INT A контроллер формирует вектор прерывания, который по шине данных поступает в процессор. Вектор используется для адресации обработчика.
- У контроллера прерываний есть порт, значит он адресуется.

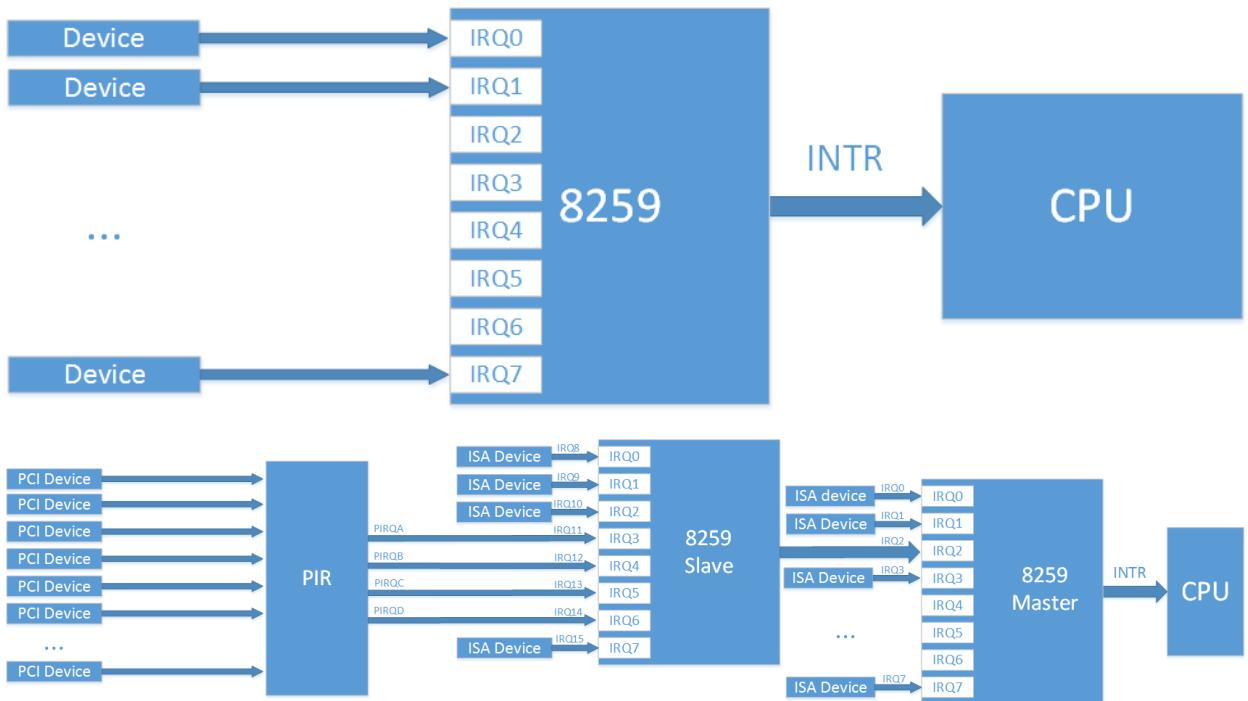
## Обработчики аппаратных прерываний: виды и особенности

### 1. PIC

Коротко:

Устройство -> вход IRQx контроллера -> ножка процессора. Используется каскадное подключение: через IRQ2 подключается ведомый контроллер, который принимает прерывания от 8 до 15 (то есть еще 8, профит: +7). Всего 15 доступных прерываний для устройств.

Этого было достаточно для систем с шиной ISA.



Контроллер прерываний PIC состоит из двух каскадно-соединенных контроллеров, называемыми master (ведущий) и slave (ведомый)

pin - входы контроллера, которые соединяются в соответствии с выходами конкретных внешних устройств

Во времена, когда основной шиной для подключения внешних устройств была шина ISA, такой системы в целом хватало. Надо было лишь следить, чтобы разные устройства не подключались на одну линию IRQ для избежания конфликтов, так как прерывания ISA не разделяемые. Обратная совместимость.

По своему устройству PIC может передавать прерывания только на один главный процессор

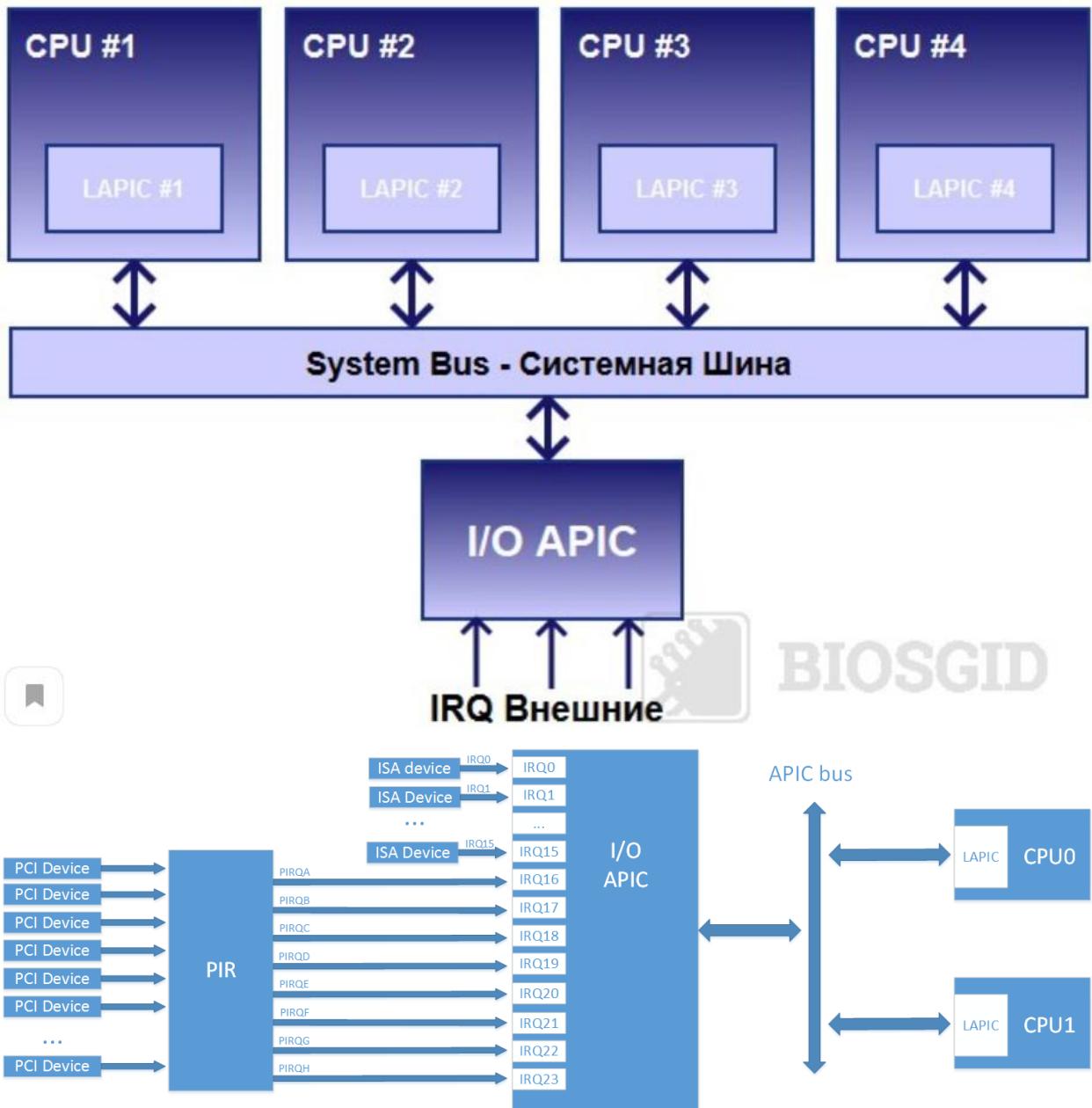
## 2. APIC

Коротко:

APIC = Advanced PIC, балансирует нагрузку. Для каждого процесса добавляется LAPIC (Local APIC) и для маршрутизации добавляется I/O APIC. Все это объединяется в общую шину - APIC.

APIC состоит из:

1. LAPIC (прямо в ядре процессора) - контроллер для каждого процессора.
2. I/O APIC (на материнской плате) - сбалансированно распределяет.



Создан для многопроцессорных систем. Для каждого процессора добавляется специальный контроллер LAPIC (Local APIC) и для маршрутизации прерываний от устройств добавляется контроллер I/O APIC. Все эти контроллеры объединяются в общую шину с названием APIC

### 3. MSI

На смену шины PCI пришёл PCI express, в котором линии прерываний было решено убрать. Чтобы сохранить совместимость, сигналы о возникновении прерываний (INTx#) эмулируются отдельными видами сообщений. В этой схеме логическое сложение линий прерываний, которое раньше производилось физическим соединением проводов, легко на плечи PCI мостов. Однако поддержка legacy INTx прерываний — это лишь поддержка обратной совместимости с шиной PCI. На деле PCI express предложил новый метод доставки сообщений о прерываниях — MSI (Message Signaled Interrupts). В этом методе

для сигнализации о прерывании устройство просто производит запись в MMIO область отведённую под LAPIC процессора.

Если раньше на одно PCI устройство (то есть на все его функции) выделялось всего 4 прерывания, то сейчас стало возможным адресовать до 32 прерываний.

В случае с MSI нет никакого разделения (sharing) для линий прерываний. Каждое прерывание соответствует своему устройству.

Прерывания MSI решают также ещё одну проблему. Допустим устройство проводит memory-write транзакцию, и хочет сообщить о её завершении через прерывание. Но write транзакция может быть задержана на шине в процессе передачи (о чём устройство никак не знает), и сигнал о прерывании придёт до процессора раньше. Таким образом CPU будет читать ещё невалидные данные. В случае если используется MSI, информация об MSI передаётся также как и данные, и раньше прийти просто не сможет.

Следует заметить, что прерывания MSI не могут работать без LAPIC, но использование MSI может заменить нам I/O APIC (упрощение дизайна).



В MSI (message signaled interrupt) у каждого процессора (они еще называются ядрами) имеется свой контроллер LAPIC (L - local), при этом прерывания на каждом процессоре обрабатываются независимо, за исключением прерывания от системного таймера, который обрабатывается как правило на CPU0.

На ввод-вывод имеется один контроллер, общий для всех.

- MSI генерируются в виде сообщений, отображаемых в память
- Передаются по PCI(сейчас уже pci-e) в виде транзакций
- Поддерживают старые прерывания для совместимости

Для поддержки MSI необходимы регистры: управления msі сообщениями, данных, адресов. 2 адресных пространства – оперативная память и порты ввода/вывода.

## **Функции обработчика прерываний от системного таймера в ОС семейства Windows и семейства Unix (ВОЗМОЖНО НАДО ПОДПРАВИТЬ, ПОТОМУ ЧТО ЭТУ ЕРЕСЬ НЕИЗВЕСТНО КАК ПИСАТЬ)**

### UNIX

По тику:

- инкрементирует счетчик тиков аппаратного таймера;
- декременирует квант текущего потока;
- обновляет статистику использования процессора текущим процессом — инкремент поля `c_scri` дескриптора текущего процесса до максимального значения 127;
- инкрементирует счетчики часов и других таймеров системы;
- декрементирует счетчик времени до отправления на выполнение отложенного вызова (если счетчик достиг нуля, то выставление флага для обработчика отложенного вызова).

По главному тику:

- регистрирует отложенные вызовы функций, относящиеся к работе планировщика, такие как пересчет приоритетов;
- пробуждает в нужные моменты системные процессы, такие как `swapper` и `pagedaemon`. Под пробуждением понимается регистрация отложенного вызова процедуры `wakeup`, которая перемещает дескрипторы процессов из списка "спящих" в очередь готовых к выполнению.
- декрементирует счётчик времени, оставшегося времени до посылки одного из следующих сигналов:
  - `SIGALRM` – сигнал, посылаемый процессу по истечении времени, предварительно заданного функцией `alarm()`;
  - `SIGPROF` – сигнал, посылаемый процессу по истечении времени заданного в таймере профилирования;
  - `SIGVTALRM` – сигнал, посылаемый процессу по истечении времени, заданного в "виртуальном" таймере.

По кванту:

- посыпает текущему процессу сигнал `SIGXCPU`, если тот превысил выделенную ему квоту использования процессора.

### Windows

По тику:

- инкрементирует счётчик системного времени;

- декрементирует квант текущего потока на величину, равную количеству тактов процессора, произошедших за тик (если количество затраченных потоком тактов процессора достигает квантовой цели, запускается обработка истечения кванта);
- декрементирует счетчики времени отложенных задач;
- если активен механизм профилирования ядра, инициализирует отложенный вызов обработчика ловушки профилирования ядра путем постановки объекта в очередь DPC: обработчик ловушки профилирования регистрирует адрес команды, выполнявшейся на момент прерывания.

По главному тику:

- освобождает объект "событие", который ожидает диспетчер настройки баланса.

По кванту:

- инициализирует диспетчеризацию потоков путем постановки соответствующего объекта в очередь DPC.

**3.2 Защищенный режим:** назначение системных таблиц – глобальной таблицы дескрипторов (GDT), таблицы дескрипторов прерываний (IDT), теневых регистров (структуры, описывающие дескрипторы GDT и IDT и заполнение дескрипторов в лабораторной работе по защищенному режиму). Адресация памяти в ЗР (**IDT нет в новом**)

3.2 (2021) Защищенный режим: назначение системных таблиц – глобальной таблицы дескрипторов (GDT), теневых регистров. Структура, описывающая дескрипторы GDT заполнение дескрипторов GDT в лабораторной работе по защищенному режиму).

Адресация памяти в ЗР

### **Защищенный режим**

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.

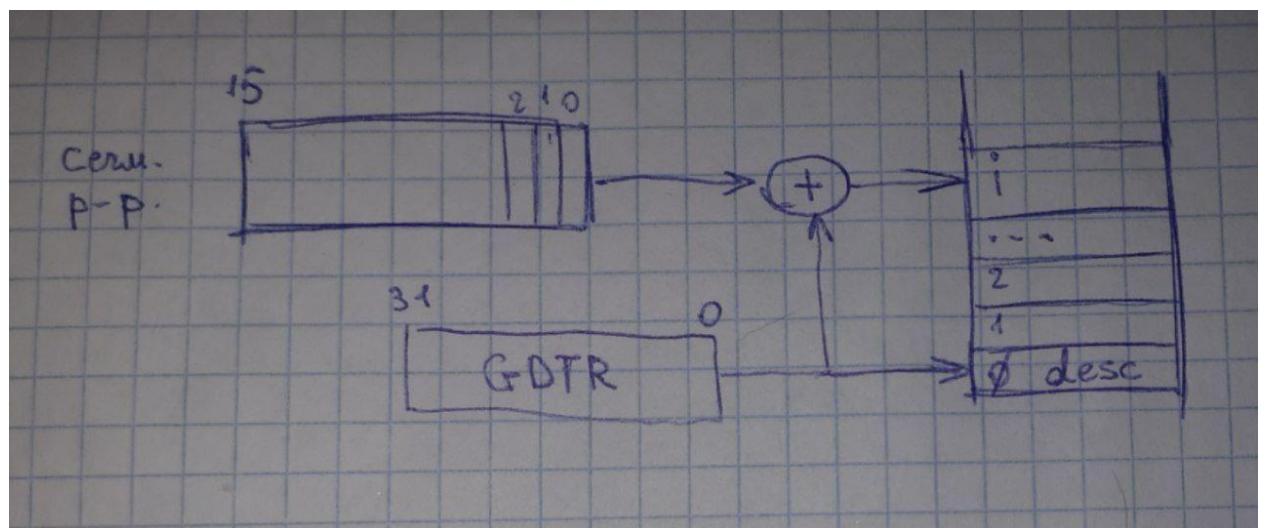
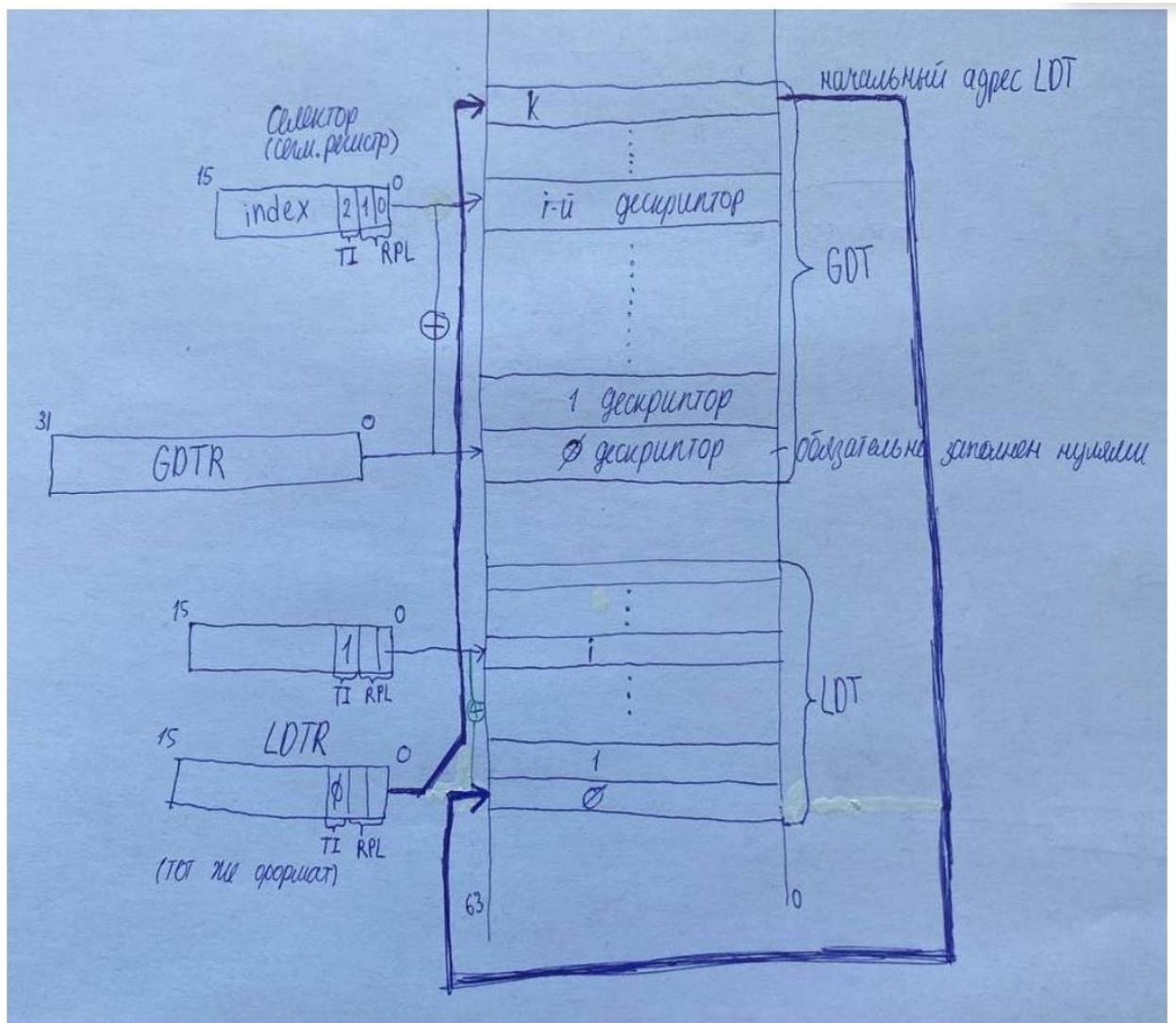
Создан для работы нескольких независимых программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимодействие задач должно регулироваться.

Программы, разработанные для реального режима, не могут функционировать в защищенном режиме. (Физический адрес формируется по другим принципам.)

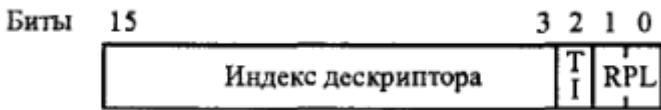
### **Таблица дескрипторов (GDT)**

Так как одна память, то и GDT в системе одна. Она находится в памяти ядра системы. На начало таблицы указывает GDTR (32).

GDT состоит из 8-байтных записей - дескрипторов. Первая запись всегда должна быть заполнена нулями и не используется. Далее следуют дескрипторы сегментов.



Для обращения к какому-либо сегменту используются селекторы: селектор помещается в сегментный регистр и имеет следующий вид:



Сегментный регистр – селектор (16) – идентификатор сегмента - внутри себя содержат номер записи в таблице(индекс), по нему получаем запись в ней - то есть дескриптор.

- 0 и 1 бит – RPL (request privilege level) отвечают за уровни привилегий - их всего 4, используются процессором при проверке возможности доступа к сегментам.
- 2 бит TI table indicator - определяет к какой таблице идет обращение (1- к локальной или 0 - к глобальной)
- 3-15 - индекс. Так как размер дескриптора 8 байт, то минимальный селектор должен тоже быть 8 - мы через первые три бита домножаем селектор на 8(добавляем три разряда в двоичной). 01 000 -> 8 (первый селектор), 10 000 -> 16 (второй селектор)

### Теневые регистры

С каждым сегментным регистром сопоставлен теневой регистр, в котором при обращении к сегментному регистру записывается информация из дескриптора. Цель – исключить обращение к GDT, которая находится в оперативной памяти. Оперативная память отстает по времени от проц., затратное действие (цикл обращения к памяти требует определенного количества тактов).

Обращение к ОП осуществляется на каждой команде, а то и несколько раз + надо сформировать (преобразовать) адрес особенно если адресация косвенная. Чтобы цикл обращения к GDT для получения физического адреса, или данных, или следующей команды, информация с дескриптора записывается в теневой регистр, и после того, как было обращение к сегментному регистру.

Теневой регистр находится непосредственно в процессоре. Это не исключает обращение для считывания команд, записи и тд

————— (не то чтобы это неверно (ниже), но писать больше)

*На каждой команде мы обращаемся к памяти, а то и несколько раз. При этом каждый раз будет выполняться преобразование адреса. Поэтому для более быстрого доступа существуют теневые регистры. В процессоре для каждого из сегментных регистров имеется так называемый теневой регистр дескриптора, который имеет формат дескриптора. Теневые регистры недоступны программисту; они автоматически загружаются процессором из таблицы дескрипторов каждый раз, когда процессор загружает соответствующий сегментный регистр. Таким образом, в защищенном режиме программист имеет дело с селекторами, т.е. номерами дескрипторов, а процессор – с самими дескрипторами, хранящимися в теневых регистрах. Именно содержимое теневого регистра (в первую очередь, линейный адрес сегмента) определяет область памяти, к которой обращается процессор при выполнении конкретной команды.*

*В реальном режиме теневые регистры заполняются не из таблицы дескрипторов, а непосредственно самим процессором. В частности, процессор заполняет поле базы каждого теневого регистра линейным базовым адресом сегмента, полученным путем*

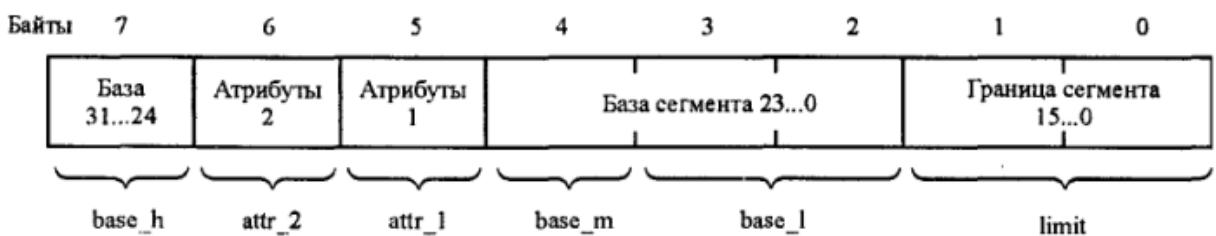
умножения на 16 содержимого сегментного регистра, как это и положено в реальном режиме. Поэтому после перехода в защищенный режим в теневых регистрах находятся правильные линейные базовые адреса, и программа будет выполняться правильно, хотя с точки зрения правил адресации защищенного режима содержимое сегментных регистров лишено смысла. Тем не менее после перехода в защищенный режим прежде всего следует загрузить в используемые сегментные регистры (и, в частности, в регистр CS) селекторы соответствующих сегментов. Это позволит процессору правильно заполнить все поля теневых регистров из таблицы дескрипторов. Пока эта операция не выполнена, некоторые поля теневых регистров (в частности, границы сегментов) могут содержать неверную информацию.

## Структура, описывающая дескриптор GDT

Структура, описывающая дескриптор GDT.

```
descr struc
    limit    dw 0
    base_l   dw 0
    base_m   db 0
    attr_1   db 0
    attr_2   db 0
    base_h   db 0
descr ends
```

Формат дескриптора В РР сегменты определяются базовыми адресами, задаваемыми в явной форме, В ЗР - дескриптором (8-байтовым полем)



Формат дескриптора для GDT:

- Байты 2-3 (base\_low), 4 (base\_middle), 7 (base\_high): база сегмента - начальный линейный адрес сегмента в адресном пространстве процессора. (имеет длину 32 бита, номер байта, может располагаться в любом месте адресного пространства 4Гбайт). Если страничная адресация выключена, он совпадает с физическим (как во 2 ЛР), включена - могут и не совпадать. База - адрес, с которого начинается данный сегмент. Повторюсь: адрес в виртуальном адресном пространстве. Вообще, все упоминаемые здесь и далее адреса упоминаются в контексте виртуальности; к физическим адресам мы доступа не имеем.
- Байты 0-1 (limit): младшие 16 бит границы сегмента - номер последнего байта сегмента).

- Байт 6 (attr\_2)
  - 0-3 (lim): оставшиеся старшие 4 бита границы сегмента (итого 20 бит). Поскольку у регистров доступны младшие части, это показывает, что старшие компьютеры поддерживают реальный режим аппаратно - основанная идея Intel - обратная совместимость. Возникает вопрос - сколько разрядов в шине адреса в реальном режиме? 20. Это максимально возможный объем который мы можем адресовать в реальном режиме - 1 МБ памяти. 0,1 - limit (только 16 разрядов), а шина 20-разрядная - нам не хватает 4 разрядов. Таким образом мы имеем базовый линейный адрес - можем адресовать начало сегмента.
  - 6 бит (D default): разрядность operandов и адресов по умолчанию
  - 0-16
  - 1-32
  - Можно изменить на противоположный префиксом замены размера 66h(операнда) и 67h (адреса). D=0 не запрещает использовать 32 регистры: компилятор сам добавит префикс
  - 7 бит G (бит дробности (гранулярности)): единицы, в которых задается граница. 0-в байтах (и тогда сегмент <=1 Мбайт), 1-в блоках по 4 Кбайт (страницах) (до 4 Гбайт) гр. сег.=гр.в.дескр.\*4К+4095 - до конца последнего 4-Кбайтного блока).
  - 5 L - флаг, который ранее был зарезервирован, теперь служит признаком 64-разрядности сегмента. Если он установлен, флаг D/B должен быть сброшен.
  - 4 AVL - неиспользуемый бит. Может использоваться по усмотрению ОС.
- Байт 5: attr\_1:
  - 0 бит (A accessed): устанавливается процессором, когда в какой-либо сегментный регистр загружается селектор данного сегмента (было обращение)
  - 1-3 биты: тип сегмента.
    - 3 бит – бит предназначения: 0 – сегмент данных/стека, 1-кода
    - 2 бит:
      - Для кода [бит подчинения: 0 – код подчинен (связан с каким – то другим сегментом), 1 – обычный]. Подчиненные, или согласованные сегменты обычно используются для хранения подпрограмм общего пользования; для них не действуют общие правила защиты программ друг от друга.
      - Для стека и данных [0-данные, 1-стек]
    - 1 бит:
      - Для кода [0–чтение из сегмента запрещено (не относится к выборке команд) - считывание из памяти и загрузка в регистры процессора, mov, 1 – разрешено]
      - Для данных [0- модификации запрещены, 1-модификации разрешены]
  - 4 бит (S system): идентификатор сегмента (0-системный сегмент, 1-сегмент памяти).
  - 5-6 биты (DPL descriptor privilege level): уровень привилегий этого дескриптора: от 0 (УП ядра системы) до 3 (УП приложений). (как в селекторе RPL request PL (программно), CPL current PL (аппаратно))

- 7 (P present): бит присутствия, представлен ли сегмент в памяти (выгружен ли из внешней в оперативную)

## Заполнение дескрипторов GDT в лабораторной работе по защищенному режиму

Начальные значение для дескрипторов GDT и IDT:

```
;GDT
gdt_null descr <>
gdt_code16 descr <code16_size-1,0,0,98h>
gdt_data4gb descr <0FFFFh,0,0,92h,0CFh>
gdt_code32 descr <code32_size-1,0,0,98h,40h>
gdt_data32 descr <data_size-1,0,0,92h,40h>
gdt_stack32 descr <stack_size-1,0,0,92h,40h>
gdt_video16 descr <3999,8000h,0Bh,92h>
```

Заполнение дескрипторов GDT (на примере дескриптора code32. Для остальных линейный адрес начала заполняется аналогично)

```
; в реальном режиме; code32 - имя сегмента,
; который содержит код для выполнения в защищенном режиме
mov ax, code32
shl eax, 4
mov word ptr gdt_code32.base_l, ax
shr eax, 16
mov byte ptr gdt_code32.base_m, al
mov byte ptr gdt_code32.base_h, ah
```

## Адресация памяти в ЗР

Нас будет интересовать память(адресация). В реальном режиме были только сегмент и смещение. Адрес = сегмент\*16 + смещение - получаем линейный физический адрес. Доступное адресное пространство - 1 МБ.

В защищенному же режиме есть поддержка виртуальной памяти, в том числе и на аппаратном уровне

– далее не понятно, что надо написать как мы подсчитываем физ адрес или же про ВП

!!!3.1 (2022) Прерывания: классификация и особенности каждого типа прерываний, примеры. Аппаратные прерывания: виды и особенности каждого вида. Последовательность действий при выполнении запроса приложения на ввод=вывод с комментариями действий. Программируемый контроллер прерываний (PCI), маскируемые и немаскируемые прерывания. Адресация прерываний в ЗР (схема)

## прерывания

### Прерывания: классификация

#### аппаратные прерывания

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHthsB8TTy1Xvv9SN9zM/edit#bookmark=id.xq2yy4r2ptdq>

#### последовательность

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHthsB8TTy1Xvv9SN9zM/edit#bookmark=id.kyv1ranwncv6>

#### программируемый контроллер PCI

Программируемый контроллер прерываний PIC (Programmable Interrupt Controller)

В шинной архитектуре (также есть канальная) внешними устройствами управляют контроллеры или адаптеры. Контроллер – программируемое устройство (имеется набор регистров и некоторая логика), находящееся в внешнем устройстве, а адаптер – на материнской плате.

По завершении операции процессор информируют специальные устройства – контроллеры. Контроллер получает от процессора команду, выполняя которую, контроллер берет на себя управление операцией ввода/вывода. По завершении операции ввода/вывода контроллер посыпает на вход контроллера прерываний сигнал.

Контроллер прерываний PIC состоит из двух каскадно-соединенных контроллеров, называемыми master (ведущий) и slave (ведомый)

pin - входы контроллера, которые соединяются в соответствии с выходами конкретных внешних устройств

Во времена, когда основнойшиной для подключения внешних устройств была шина ISA, такой системы в целом хватало. Надо было лишь следить, чтобы разные устройства не подключались на одну линию IRQ для избежание конфликтов, так как прерывания ISA не разделяемые. Обратная совместимость.

По своему устройству PIC может передавать прерывания только на один главный процессор

Что поступает на IRQ:

- IRQ 0, системный таймер;
- IRQ 1, клавиатура;
- IRQ 2, используется для запросов устройств, подключенных каскадом;

- IRQ 8, часы реального времени;
- IRQ 9, зарезервировано;
- IRQ 10, зарезервировано;
- IRQ 11, зарезервировано;
- IRQ 12, ps/2-мышь (тачпад);
- IRQ 13, сопроцессор;
- IRQ 14, контроллер «жёсткого» диска;
- IRQ 15, зарезервировано; – IRQ 3, порты COM2, COM4;
- IRQ 4, порты COM1, COM3;
- IRQ 5, порт LPT2;
- IRQ 6, контроллер дисковода;
- IRQ 7, порт LPT1, принтер.

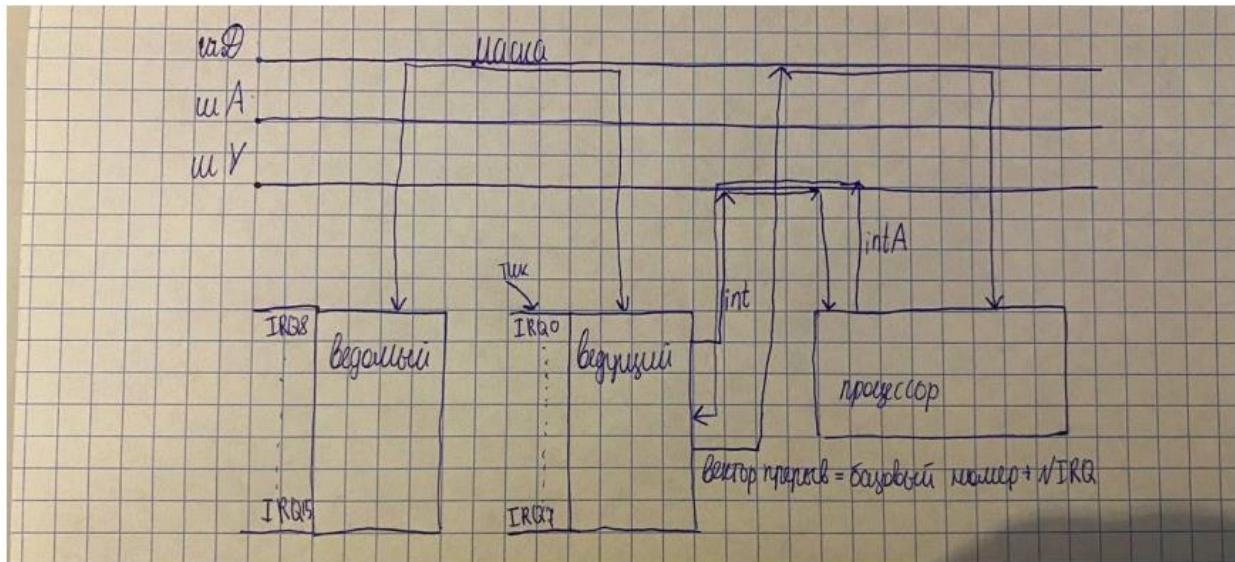
Здесь сигналы приведены в порядке убывания приоритетов.

Процесс: На вход контроллера приходит сигнал, контроллер формирует сигнал int, который по шине управления переходит на ножку процессора. Процессор посылает intA по шине управления в контроллер. Контроллер выставляет на шину данных вектор прерывания, который поступает в процессор и используется им для адресации обработчика прерывания.

Вектор прерывания=базовый вектор + №IRQ

*Пример: Int8h=ведущий контр.:8+0. В защищенном режиме прерывание от системного таймера нельзя называть int8h, так как там другой базовый вектор*

На контроллер прерываний приходят маскируемые прерывания. Маску он получает по шине данных, так как маска – это данные. Сброс контроллера прерываний – посылка впорт команды на сброс.



(на всякий)

Есть еще APIC. Создан для многопроцессорных систем. Для каждого процессора добавляется специальный контроллер LAPIC (Local APIC) и для маршрутизации

прерываний от устройств добавляется контроллер I/O APIC. Все эти контроллеры объединяются в общую шину с названием APIC.

Сейчас используется MSI - сейчас

В MSI (*message signaled interrupt*) у каждого процессора (они еще называются ядрами) имеется свой контроллер LAPIC (L - local), при этом прерывания на каждом процессоре обрабатываются независимо, за исключением прерывания от системного таймера, который обрабатывается как правило на CPU0.

На ввод-вывод имеется один контроллер, общий для всех.

- MSI генерируются в виде сообщений, отображаемых в память
- Передаются по PCI(сейчас уже pci-e) в виде транзакций
- Поддерживают старые прерывания для совместимости

Для поддержки MSI необходимы регистры: управления tsi сообщениями, данных, адресов. 2 адресных пространства – оперативная память и порты ввода/вывода.

Зубков

Существует два контроллера прерываний. Первый контроллер, обслуживающий запросы на прерывания от IRQ0 до IRQ7, управляет через порты 20h и 21h, а второй (IRQ8 - IRQ15) - через порты OA0h и OA1h. Если несколько прерываний происходят одновременно, обслуживается в первую очередь тот, у которого высший приоритет. При инициализации контроллера высший приоритет имеет IRQ0 (прерывание от системного таймера), а низший - IRQ7. Все прерывания второго контроллера (IRQ8 - IRQ15) оказываются в этой последовательности между IRQ1 и IRQ3, так как именно IRQ2 используется для каскадирования этих двух контроллеров. В тот момент, когда выполняется обработчик аппаратного прерывания, других прерываний с низшими приоритетами нет, даже если обработчик выполнил команду sti. Чтобы разрешить другие прерывания, каждый обработчик обязательно должен послать команду EOI - конец прерывания - в соответствующий контроллер.

### **маскируемые и немаскируемые прерывания**

?? так Маскируемые вызываются по маске - соответственно вызов таких прерываний можно запретить установкой соответствующих битов в *регистре маскирования прерываний*?. Не маскируемые запретить нельзя, такими могут быть ошибки различные.

### **адресация прерываний в ЗР схема**

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHth-sB8TTy1Xvv9SN9zM/edit#bookmark=id.2v9zt489i2bq>

## **4 билет**

4.1 Тупики: Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа, алгоритмы обнаружения тупиков. Пример анализа состояния системы метод редукции графа. Методы восстановления работоспособности системы.

### **Тупик, определение**

Тупик — это ситуация, которая возникает в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другим процессом, ожидающим освобождения ресурса, занятого первым процессом.

Ресурсы с точки зрения особенности их использования, делятся на:

- Повторно используемые ресурсы
- Потребляемые ресурсы

Повторное используемые ресурсы — используются многократно, использование ресурса не изменяет качества и характеристики ресурса. К повторно используемым ресурсам относится: реентерабельный код системы, системные таблицы, разделяемая память, семафоры (так как структура никак не меняется после захвата и освобождения, функции также не меняются, вообще все это называется объектами ядра), программные каналы.

Потребляемые ресурсы — количество в системе переменно и произвольно. К потребляемым ресурсам относится: сообщения (получено -> перестало существовать), память, каналы, внешние устройства, процессор

*Память раньше была важным ресурсом, сейчас – проще. Процессор – как средство выполнения. Простые очереди (то, что раньше называлось pulling теперь не используется).*

*Вся теория тупиков написана для повторно используемых ресурсов, потому что их количество в системе как правило известно и неизменно. Сравним: сигналы - любой процесс может произвести или потребить любое количество сигналов, но семафоры – если есть приложение, которое использует набор семафоров, то мы определили количество семафоров в наборе, оно динамически не меняется, с этим можно работать.*

*Теория тупиков в настоящее время не актуальна (так как там в основном про аппаратное, а это сильно поменялось). Но это никак не коснулось используемых объектов ядра.*

*Тупики в ОС в настоящее время не настолько актуальны, так как в ОС написаны с умом и принимаются меры по избеганию тупиков. Но тупики характерны для различных программ, выполняющих определенные вычисления, услуги, и особенно для распределенных систем.*

#### **4 условия возникновения тупика (в любой системе):**

- Взаимоисключение (Mutual exclusion). Возникает, когда процессы монопольно используют ресурсы.
- Ожидание (Hold and wait). Когда процессы, удерживая полученные им ресурсы, запрашивают и ждут получения дополнительных ресурсов, чтобы продолжить свое выполнение.

- Неперераспределяемость (No preemption). Когда ресурсы нельзя отобрать у процесса до их завершения, или как говорят, добровольного освобождения этих ресурсов. (надпись: ресурсы у процесса нельзя принудительно отобрать).
- Круговое ожидание (Circular wait). Когда существует замкнутая цепь процессов, в которой каждый процесс занимает необходимый другому (следующему в цепи) процессу ресурс.

### **3 основных метода борьбы с тупиками.**

- Недопущение (исключение самой возможности возникновения тупика)
- Обход (предотвращение)
- Обнаружение и восстановление

### **Обнаружение тупиковых ситуаций**

Для этого используется двудольный направленный граф (**градовая модель Холда, ГМХ**).

Имеется 2 непересекающихся подмножества вершин – подмножество вершин процессов и подмножество вершин ресурсов. При этом дуга не может соединять вершины одного подмножества.

Граф называется направленным, потому что существует 2 типа дуг – выделение, когда дуга выходит из вершины подмножества ресурсов и входит в вершину подмножества процессов, запрос – когда наоборот.

Тупик наступает только в результате запроса, причем запроса, который не может быть сразу удовлетворен.

*Граф описывает ситуацию в системе, которая может в какой-то момент возникнуть. Фактически такой граф становится необходимым, чтобы убедиться, что система пришла в тупик, и причины. Речь идет о большой системе взаимодействующих процессов.*

### **Метод редукции графа**

Тупиковая ситуация с использованием ГМХ обнаруживается методом редукции графа.

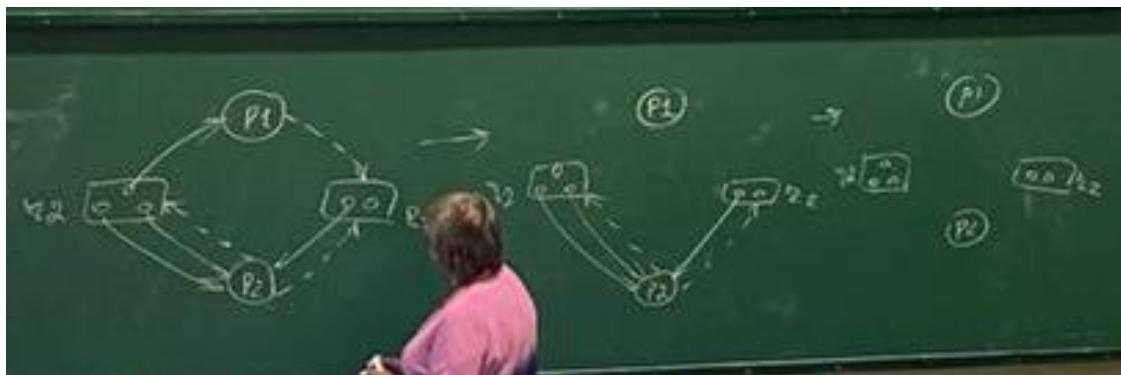
Если запрос процесса мб удовлетворен, то такая дуга мб удалена. В результате того, что запрос удовлетворен, он может освободить ресурсы, которые ранее занял, и эти ресурсы могут удовлетворить другие процессы.

Если в результате редукции все дуги удаляются и все вершины становятся изолированы, то система не находится в тупике. Если же все дуги удалить невозможно (те дуги, которые определяют петлю запросов), значит система в тупике.

## Пример анализа состояния системы методом редукции графа

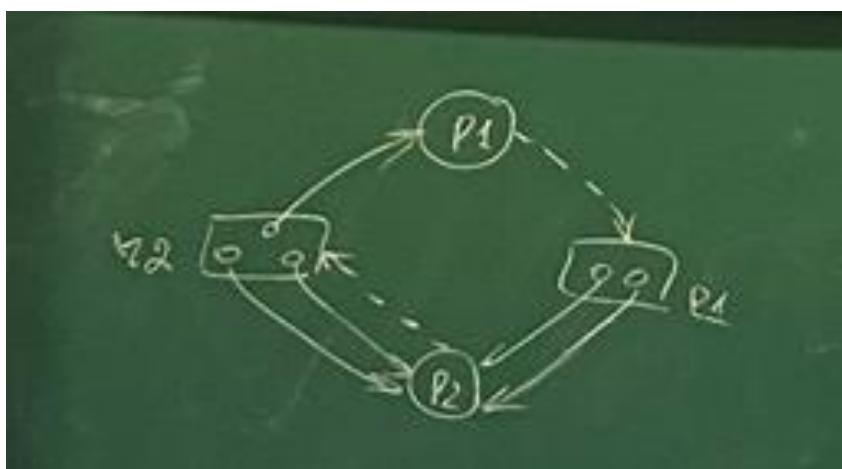
(сплошные стрелки - что уже случилось, пунктирные - новые запросы)

1.



Пробуем редуцировать. Если будет удовлетворен запрос  $\Pi_1$ , то он сможет завершиться и освободить. И тогда можно будет удовлетворить и  $\Pi_2$ . В результате получаем изолированные вершины процессов в рассматриваемом графе → система не находится в тупике.

2. Усложним ситуацию. Имеется петля (цикл) запросов



## Способы представления графа

Двудольный граф может быть описан 2 матрицами или 2 связными списками. Матричное представление более удобно, мы используем его.

## Алгоритмы обнаружения тупиков

(здесь я не уверена, что есть вот прям эти два метода. Есть предположение, что 1 и 2 надо объединить под общим заголовком Метод прямого обнаружения тупика)

1. Метод прямого обнаружения.

Последовательно рассматриваются запросы каждого процесса и определяется, может ли этот запрос быть удовлетворен. В процессе просмотра определяется возможно ли сокращение соответствующей дуги или невозможно (если возможно - сокращается). Те процессы, которые останутся в этих матрицах после всех возможных сокращений будут находиться в тупике. Очевидно, что для реализации нужно выполнить ( $m$  процессов,  $n$  ресурсов)  $(mn)^2$  проверок.

## 2. Более эффективный алгоритм.

Состояние системы можно описать двумя матрицами и вектором:

- Матрица выделения  $A$ ,  $a_{ij}$  – сколько единиц  $j$ -го ресурса получил  $i$ -й процесс
- Матрица запросов  $B$ ,  $b_{ij}$  – сколько единиц  $j$ -го ресурса запрашивает  $i$ -й процесс
- вектор свободных ресурсов  $F$ .  $f_i$  – сколько единиц  $i$ -го ресурса свободно.

В матрицах для каждого процесса есть строка. Можем сравнивать строку запросов каждого процесса с вектором свободных ресурсов: если вектор запросов процессов меньше, чем вектор свободных ресурсов, то его запросы могут быть удовлетворены.

Никаких ограничений на запросы процесса не накладываются (то есть запрашивают и получают), то чтобы обнаружить тупик, может выполняться проверка каждого запроса. Тогда меньше проверок. Или если таймаут истекает.

(!ЗДЕСЬ  $a_{ij}$ , а не  $b_{ij}$ ) В результате можно получить следующее выражение:

$$\sum_{i=1}^n b_{ij} + f_j = k_j, \text{ где } k_j$$

— общее количество единиц  $j$ -ого ресурса.

Обычно процессы запрашивает ресурсы по одному. Хотя если взять страницы....

Пример не с нашей лекции

Матрица распределения и запросов				Свободные ресурсы				
P/R	1 2 3	P/R	1 2 3	F <sub>j</sub>	1 2 3	P/R	1 2 3	P1
1	0 1 1	1	1 1 0		4 0 2	1	0 1 1	1 0 0 0
2	1 3 0	2	0 0 1		2 9 1	2	0 0 0	2 0 0 0
3	1 5 0	3	1 1 2	R <sub>j</sub>	6 9 3	3	1 5 0	3 1 5 0
2	0 1							

$\Rightarrow F = 532$

$\Rightarrow F = 693$

$\downarrow$

$F = 543$

$\downarrow$

$\begin{matrix} 1 & 1 & 2 & 3 \\ 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{matrix}$

## Методы восстановления работоспособности системы.

### 2 глобальных подхода

- Последовательно завершаются процессы, попавшие в тупик. У них отбираются ресурсы и этих ресурсов может оказаться достаточно, чтобы другие процессы могли успешно продолжить работу. Очевидный недостаток - проделанная работа завершаемого процесса будет потеряна.
- Завершение других процессов, которые выполняются самостоятельно, не имеют отношения к тупику. Их ресурсы возвращаются системе, и процессы, попавшие в тупик, мб смогут завершиться. Решается на основе приоритетов. Либо kill, либо откат до возникновения запроса. Для этого надо хранить состояния.

Критерии завершения процесса 1. Приоритет 2. Цена повторного запуска процесса 3. Внешняя цена

- (у нас в лекциях не было) Перезагрузка системы. Дорогое решение, так как будет потеряна вся проделанная завершившимся процессами работа. Для системы, работающей длительное время, такой подход невозможен.

4.2 Задача «Обедающие философы» – модели распределение ресурсов вычислительной системы. Множественные семафоры UNIX: системные вызовы, поддержка в системе, пример использования из лабораторной работы «производство-потребление».

### Множественные семафоры

Обладают очень важным свойством: одной неделимой операцией можно выполнить проверку или изменение всех или части семафоров набора. Если какая-либо операция не может быть выполнена над одним семафором, то неудачной считается операция над всеми семафорами.

Это важно. Потому что использование семафоров в программах, когда процессы одновременно захватывают и освобождают одни и те же семафоры, приводили к тупикам.

По-другому - массив считающих семафоров. ОС, с которыми мы работаем поддерживают именно наборы считающих семафоров.

Задание будет на экзе!!!: придумать пример тупика для двух бинарных семафоров (два процесса, два семафора).

Рассмотрим множественные семафоры на примере классической задачи

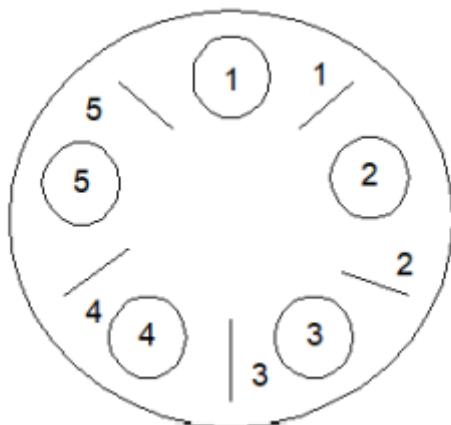
**Задача «Обедающие философы» – модели распределение ресурсов вычислительной системы.**

Эта задача используется для демонстрации различных алгоритмов разделения ресурсов и возможностей множественных семафоров.

Пять философов пытаются пообедать спагетти. Как известно, спагетти едят при помощи двух вилок: на одну вилку накручивают длинные макароны, другой вилкой подсекают. Философы сидят вокруг стола, перед каждым стоит тарелка, между тарелками лежит только по одной вилке. Таким образом вилок всего пять.

Типичный философ действует следующим образом: думает некоторое время, пытается взять две вилки и, если это удается, ест некоторое время.

**То, как философ берет вилки, – либо обе одновременно, либо сначала левую и, если это удалось, то правую и т.п. – создает различные ситуации, требующие определения.**



Рассмотрим ситуацию, когда команды P и V выполняют обработку сразу двух семафоров (листинг 7). (Пишем код, который как раз демонстрирует свойства множественных семафоров:)

```
Program example_5fl;
forks : array[1..5] of semaphore;
right, left : integer;
```

```

P1 : // первый философ
left = 5; right = 1;
While (true) do
    begin
        // размышляет
        P(forks[left], forks[right]);
        // ест некоторое время
        V(forks[left], forks[right]);
    end;
end; // P1
...
P5 : // пятый философ
left = 4; right = 5;
While (true) do
    begin
        // размышляет
        P(forks[left], forks[right]);
        // ест некоторое время
        V(forks[left], forks[right]);
    end;
end; // P5
begin
parbegin
    P1; P2; P3; P4; P5;
parend;
end.

```

Листинг 7

При выполнении операции Р процесс может быть заблокирован либо на одном семафоре, либо на другом, либо на обоих сразу. Если процессы не пересекаются по требуемым ресурсам, то они могут выполняться параллельно.

Философы могут действовать тремя способами, которые демонстрируют три негативные ситуации в системе

- Каждый из философов пытается взять сразу оба прибора и, если ему это удается, то он начинает есть. Поев, кладет обе вилки на стол.  
(«голодание» или бесконечное откладывание. Сколько философов умрёт от голода? Рязанова заявляет, что один и приводит простой пример - у одному из философов постоянно не достается, то правого, то левого)
- Философ берет правую вилку и, если ему это удается, то удерживая правую пытается взять левую вилку. (Тупиковая ситуация. Если все философы взяли по правому прибору)
- Философ берет правую вилку и, если левую вилку взять не может, то кладет правую. (захват и освобождение одних и тех же ресурсов)

## Множественные семафоры UNIX

*Все рассмотренные ранее средства связаны с активным ожиданием на процессе с помощью циклов, на проверку которых тратится процессорное время. Появление семафоров позволило устранить это ожидание, но потребовало использования системных вызовов ( $P(s)$ ,  $V(s)$ ) -- система переходит в режим ядра, приходится переключать контекст. Вместо активного ожидания – теперь блокировка.*

*Минус – если процессы используют много семафоров, то сложно следить за их изменением, сложно отлаживать. Стремление структурировать эти средства. Использование набора семафоров является некоторым решением этой проблемы.*

Семантически наборы семафоров представляются как массивы семафоров

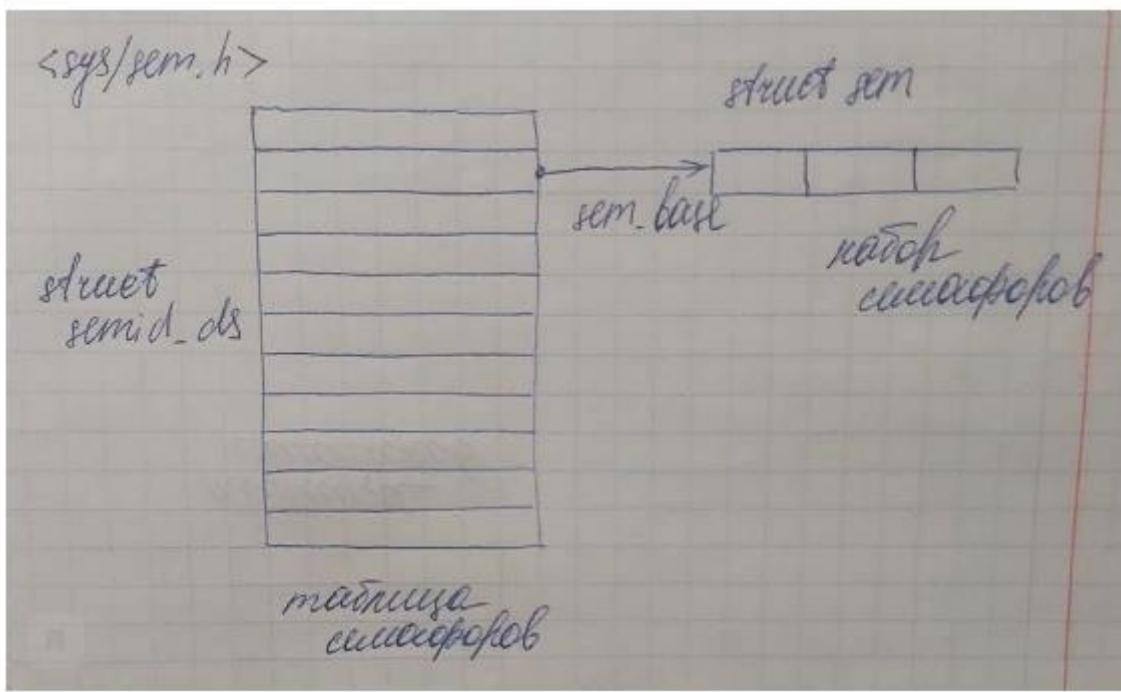
В ядре системы имеется таблица дескрипторов семафоров. В ней отслеживаются все созданные в системе наборы семафоров (Каждая строка описывает отдельный набор). Для этого существует специальная структура struct semid\_ds в библиотеке <sys/sem.h> (ds=descriptor, sem=semafor).

О каждом наборе известно:

1. Имя – целое число. В Unix все идентификаторы-целые. Присваивается процессом, который создал набор семафоров. Другие процессы по этому имени могут выбрать набор и по этому номеру получить дескриптор для доступа к набору.
2. Uid (user id) – айди создателя и его группы (groupid). Процесс, эффективный uid которого совпадает с uid создателя может удалять набор и изменять его управляющие параметры.
3. права доступа – 9 букв user, group, others – r,w,e
4. количество семафоров в наборе
5. время изменения одного или нескольких значений семафоров последним процессом (именно время). Чтобы контролировать последовательность действий – случилось до/после
6. время последнего изменения управляющих параметров наборов последним процессом (не важно, какой процесс – важно время)
7. указатель на набор или массив семафоров. Индексы начинаются с 0.

О каждом семафоре набора имеются следующие данные

1. значение семафора
2. идентификатор процесса, который оперировал семафором в последний раз
3. число процессов, заблокированных в текущий момент времени на семафоре



## СИСТЕМНЫЕ ВЫЗОВЫ, ПОДДЕРЖКА В СИСТЕМЕ

- Int semget(key\_t key, int num\_sem, int fl),

Функция `semget()` создает новый набор семафоров или открывает уже имеющийся

В случае успешного завершения функция возвращает дескриптор семафора, а в случае неудачи -1. Параметр `numb_sem` задает количество семафоров в наборе. Параметр `key` задает идентификатор семафора. Если значением `key` является макрос `IPC_PRIVATE`, то создается набор семафоров, который смогут использовать только процессы, порожденные процессом, создавшим семафор. Параметр `flag` представляет собой результат побитового сложения прав доступа к семафору и константы `IPC_CREATE`.

- Int semctl(int semfd, int num, int cmd, union semun arg),

Функция `semctl()` позволяет изменять управляющие параметры набора семафоров, где `semfd` (filedescriptor), `num`-количество семафором

- Int semop(int semfd, struct sembuf, \*op, int nop);

После получения идентификатора дескриптора семафора в системе значение семафора можно изменять с помощью системного вызова `semop()`

Параметр `op` – указатель на массив объектов типа `struct sembuf`, параметр `nop` определяет количество семафоров набора, над которыми выполняется операция. Системный вызов `semop` оперирует не с отдельным семафором, а с множеством семафоров, применяя к нему "массив операций".

Все системные вызовы в юникс если не выполнен, возвращается -1. Поэтому все должно проверяться на -1 БЕЗ ДЕФАЙН

- Структура для работы с семафорами

```
Struct sembuf
{
Ushort sem_num; -индекс семафора в наборе
Short sem_op - операция на семафоре
Short semfl - Флаги, определенные на семафоре
}
```

- Операции

В отличие от семафоров дейкстры с 2 операциями, на семафорах Юникс определено 3 операции.

1. Semop>0-инкремент, освобождение семафора=V(S). Разблокирование процесса.
2. Semop=0 (нет у дейкстры) - Проверка семафора на 0. Процесс, выполнивший этот системный вызов, переводится в состояние ожидания до момента высвобождения ресурса
3. Semop < 0 – декремент, захват семафора=P(S). Процесс блокируется на семафоре, если он захвачен, иначе...

Для выполнения операций 1 и 3 у процесса должно быть право на изменение, для выполнения 2 достаточно права на чтение.

- Флаги

На семафорах определены следующие флаги:

- IPC\_NOWAIT — информирует ядро о нежелании процесса переходить в состояние ожидания. Наличие этого флага объясняется желанием избежать блокировки всех процессов, которые находятся в очереди к семафору, в случае, если захвативший семафор процесс завершился аварийно или получил сигнал kill. В силу того, что этот сигнал нельзя перехватить, процесс не сможет освободить семафор и все процессы в очереди к данному семафору будут навсегда заблокированы.
- SEM\_UNDO — указывает ядру, что необходимо отслеживать изменения значений семафора в результате вызова sem\_op, чтобы при завершении процесса ядро могло ликвидировать сделанные процессом изменения. В результате процессы не будут заблокированы наечно, так как ядро отменит сделанные изменения. Система не отслеживает, чтобы обязательно было освобождение, но хотя бы вот так следит

ОС выполняет операции из "массива операций" по очереди, порядок не оговаривается. Если очередная операция не может быть выполнена, то эффект

предыдущих операций аннулируется. Если таковой оказалась операция с блокировкой, выполнение системного вызова приостанавливается. Если неудача потерпела операция без блокировки, системный вызов немедленно завершается, возвращая значение -1. Другой процесс не может получить доступ к промежуточному состоянию семафоров.

### пример использования из лабораторной работы «производство-потребление».

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/sem.h>
#include <wait.h>
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define N_PROD 3
#define N_CONS 3
#define N_WORKS 4
#define BIN_SEM_I 0
#define BUF_FULL_I 1
#define BUF_EMPTY_I 2

#define PROD_SLEEP_MIN 1
#define PROD_SLEEP_MAX 4
#define CONS_SLEEP_MIN 1
#define CONS_SLEEP_MAX 7

char *alphabet = "abcdefghijklmnopqrstuvwxyz";
typedef struct
{
    size_t prod_pos;
    size_t cons_pos;
    char buffer[N_WORKS];
} buf_struct;

struct sembuf PROD_LOCK[2] = {{BUF_EMPTY_I, -1, 0}, {BIN_SEM_I, -1, 0}};
struct sembuf PROD_RELEASE[2] = {{BUF_FULL_I, 1, 0}, {BIN_SEM_I, 1, 0}};
struct sembuf CONS_LOCK[2] = {{BUF_FULL_I, -1, 0}, {BIN_SEM_I, -1, 0}};
struct sembuf CONS_RELEASE[2] = {{BUF_EMPTY_I, 1, 0}, {BIN_SEM_I, 1, 0}};

int producer_run(buf_struct* const buf_t, const int sid, const int prodid)
{
    srand(time(NULL) + prodid);

    if (!buf_t)
    {
        perror("Producer buf_t error.");
        return 1;
    }

    for (size_t i = 0; i < N_WORKS; i++)
    {
        int sleep_time = rand() % PROD_SLEEP_MAX + PROD_SLEEP_MIN;
```

```

sleep(sleep_time);

if (semop(sid, PROD_LOCK, 2) == -1)
{
    perror("Producer lock error.");
    return 1;
}

const char symb = alphabet[buf_t->prod_pos % strlen(alphabet)];
buf_t->buffer[buf_t->prod_pos++] = symb;

fprintf(stdout, "Producer %d wrote: %c , sleep time=%d |\n", prodid + 1,
symb, sleep_time);

if (semop(sid, PROD_RELEASE, 2) == -1)
{
    perror("Producer release error.");
    return 1;
}
}

return 0;
}

int consumer_run(buf_struct* const buf_t, const int sid, const int consid)
{
    srand(time(NULL) + consid + N_PROD);

    if (!buf_t)
    {
        perror("Consumer buf_t error.");
        return 1;
    }

    for (int i = 0; i < N_WORKS; i++)
    {
        int sleep_time = rand() % CONS_SLEEP_MAX + CONS_SLEEP_MIN;
        sleep(sleep_time);

        if (semop(sid, CONS_LOCK, 2) == -1)
        {
            perror("Consumer lock error.");
            return 1;
        }

        char symb = buf_t->buffer[buf_t->cons_pos++];

        fprintf(stdout, " ");
        fprintf(stdout, "| Consumer %d read: %c, sleep time=%d\n", consid + 1,
symb, sleep_time);

        if (semop(sid, CONS_RELEASE, 2) == -1)
        {
            perror("Consumer release error.");
            return 1;
        }
    }
}
```

```

        return 0;
    }

int main()
{
    setbuf(stdout, NULL);
    int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

    int fd = shmget(IPC_PRIVATE, sizeof(buf_struct), perms | IPC_CREAT);
    if (fd == -1)
    {
        perror("shmget failed");
        return 1;
    }

    buf_struct* buf_t = shmat(fd, 0, 0);
    if (buf_t == (void*) - 1)
    {
        perror("shmat failed");
        return 1;
    }

    int isem_descr = semget(IPC_PRIVATE, 3, perms | IPC_CREAT);
    if (isem_descr == -1)
    {
        perror("semget failed");
        return 1;
    }

    if (semctl(isem_descr, BIN_SEM_I, SETVAL, 1) == -1 ||
        semctl(isem_descr, BUF_EMPTY_I, SETVAL, N_WORKS) == -1 ||
        semctl(isem_descr, BUF_FULL_I, SETVAL, 0) == -1)
    {
        perror("sem initialization error");
        return 1;
    }

    for (size_t i = 0; i < N_PROD; i++)
    {
        int child_pid = fork();

        if (child_pid == -1)
        {
            perror("Error: fork for producer");
            return 1;
        }
        else if (child_pid == 0)
        {
            producer_run(buf_t, isem_descr, i);
            return 0;
        }
    }

    for (size_t i = 0; i < N_CONS; i++)
    {
        int child_pid = fork();

        if (child_pid == -1)

```

```

    {
        perror("Error: fork for consumer");
        return 1;
    }
    else if (child_pid == 0)
    {
        consumer_run(buf_t, isem_descr, i);
        return 0;
    }
}

for (size_t i = 0; i < N_PROD + N_CONS; i++)
{
    int ch_status;
    int child_pid = wait(&ch_status);

    if (child_pid == -1)
    {
        perror("wait error");
        return 1;
    }

    if (!WIFEXITED(ch_status))
    {
        fprintf(stderr, "Child process %d terminated abnormally", child_pid);
    }
}

if (shmctl(fd, IPC_RMID, NULL) == -1)
{
    perror("shmctl with command IPC_RMID failed");
    return 1;
}

if (semctl(isem_descr, 0, IPC_RMID) == -1)
{
    perror("semctl with command IPC_RMID failed");
    return 1;
}

return 0;
}

```

*В отличие от мьютекса, семафор не имеет хозяина. Мьютекс может освободить только захвативший его процесс, а семафор – любой процесс, знающий его идентификатор.*

*Семафоры даже включены в средства межпроцессорное взд – передают информацию*

*Сравнение мьютексов (M) и семафоров (C).*

## 5 билет

5.1 Виртуальная память: распределение памяти страницами по запросам, схема с гиперстраницами, обоснование использования данной схемы. Управление памятью страницами по запросам в архитектурах x86 – расширенное преобразование (PAE) – схема преобразований. Анализ страничного поведения процессов: свойство локальности, рабочее множество.

### Пreamble

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

3 схемы управления виртуальной памятью

1. Управление памятью страницами по запросу
2. Управление памятью сегментами по запросу
3. Управление памятью сегментами, поделенными на страницы по запросу.

ПО ЗАПРОСУ: В момент порождения процесса система должна ему выделить минимально необходимый объем памяти, а в результате таких запросов загружаются необходимые участки кода.

Запрос – страничное прерывание, которое возникает при попытке доступа к незагруженной в память странице. При его обработке происходит пэйджинг.

Пейджинг – загрузка с диска новых и замещение старых страниц.

Страница – является единицей физического деления памяти. Её размер устанавливается системой. У страницы размер не обязательно 4Кб. Просто такой размер оптимален по количеству страничных прерываний.

Сегмент – является единицей логического деления памяти. Её размер определяется объемом кода. Любой сегмент создается такого размера, который требуется программе.

### **распределение памяти страницами по запросам**

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

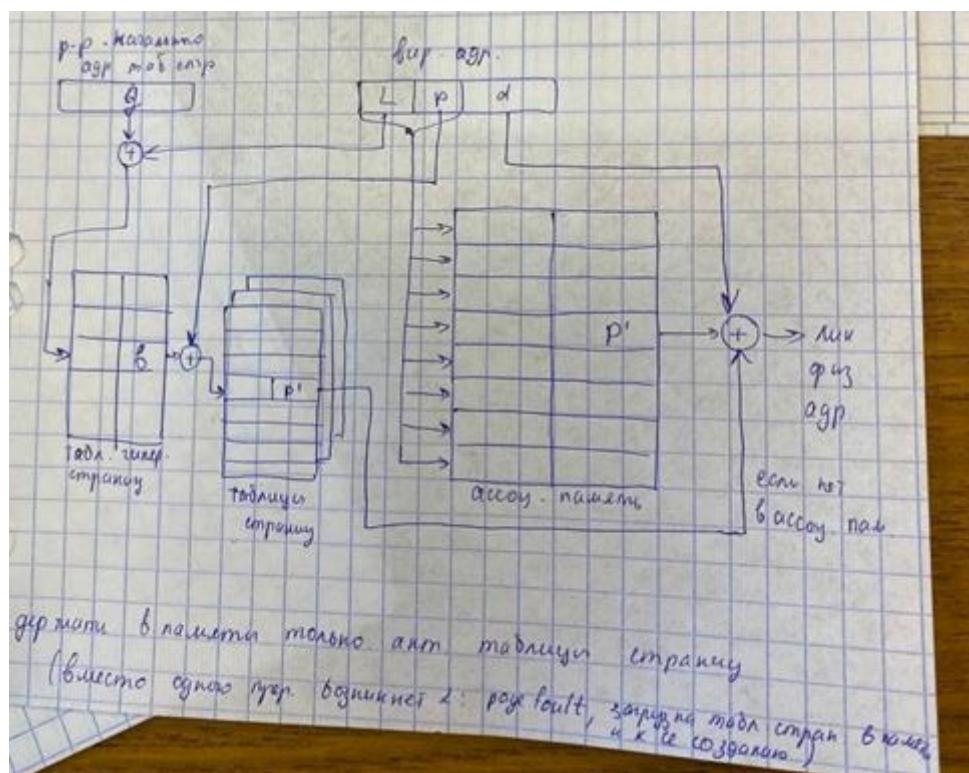
Основная идея – таблица. Необходимо иметь соответствующие таблицы, с помощью которых ставятся в соответствие страницы программного кода и физические страницы

(следует посмотреть все [3 схемы преобразования](#) виртуального адреса к физическому. Это теоретические основы, а в данном билете спрашивают практическую реализацию в интел.)

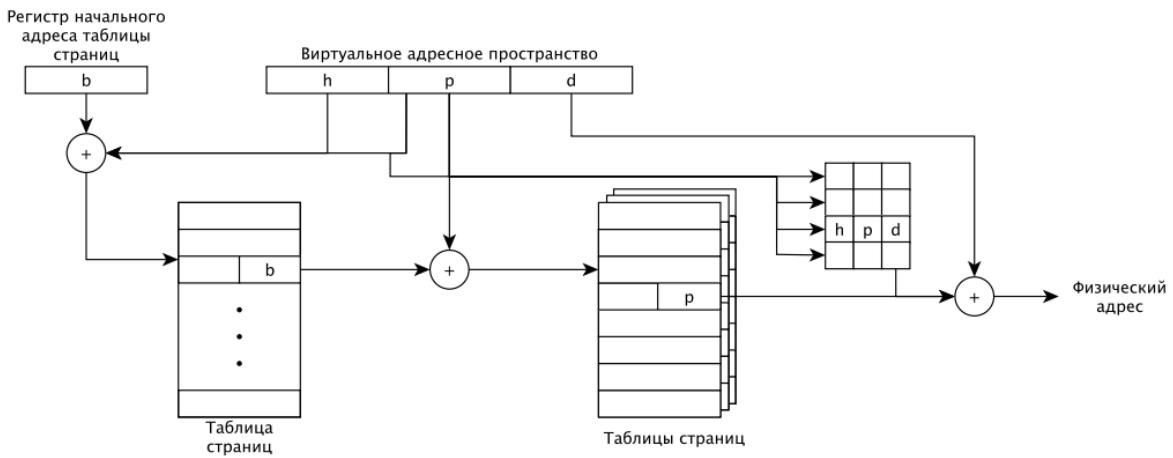
### схема с гиперстраницами, обоснование использования данной схемы

При увеличении размера программы, увеличивается виртуальное адресное пространство, все это влечет увеличение размера таблиц страниц (находятся в адресном пространстве ядра системы). Таблиц страниц столько, сколько процессов в памяти. Чтобы сократить расходы на это хранение, в IBM/360 версии 67 и IBM 370 была предложена двухуровневая страничная организация

Были введены гипер-страницы: адресное пространство процесса делится на гипер-страницы, а гипер-страницы - на страницы. То есть появляется еще один уровень таблиц страниц – таблица гипер-страниц, а виртуальный адрес делится на 3 части.



(так вроде тоже верно)



У процесса 1 таблица гипер-страниц (1 на каждый процесс) и много таблиц страниц. Из таблицы гиперстраниц получаем базовый адрес таблицы страниц. Р – смещение таблицы страниц.

“-” В итоге, сначала адрес физической страницы ищется в ассоциативном кэше, далее вместо 1 прерывания возникнет 2: 1) page fault 2) нет еще и таблицы страниц (загрузка таблицы страниц в память и, вероятно, ее создание, и потом загрузка страницы в память). Таким образом, добавляется одно обращение к оперативной памяти, что требует дополнительного времени.

“+” Но зато в таблицу гиперстраниц загружаются только актуальные таблицы страниц (то есть можно хранить в памяти только актуальные таблицы страниц). Выигрыш и в увеличении мультизадачности (больше программ может находиться в памяти)

То есть за эффективное хранение данных мы платим увеличением времени преобразования.

### **Управление памятью страницами по запросам в архитектурах x86 – расширенное преобразование (PAE) – схема преобразований.**

Начиная с Pentium Pro включен регистр CR4, где 5 бит это PAE (physical adress extension) – расширение физического адреса.

Если PAE = 1, то разрешено использование расширенной 36рр, вместо 32 pp физического адреса. При этом физический адрес остается 32 pp, все изменения касаются только работы страничного механизма.

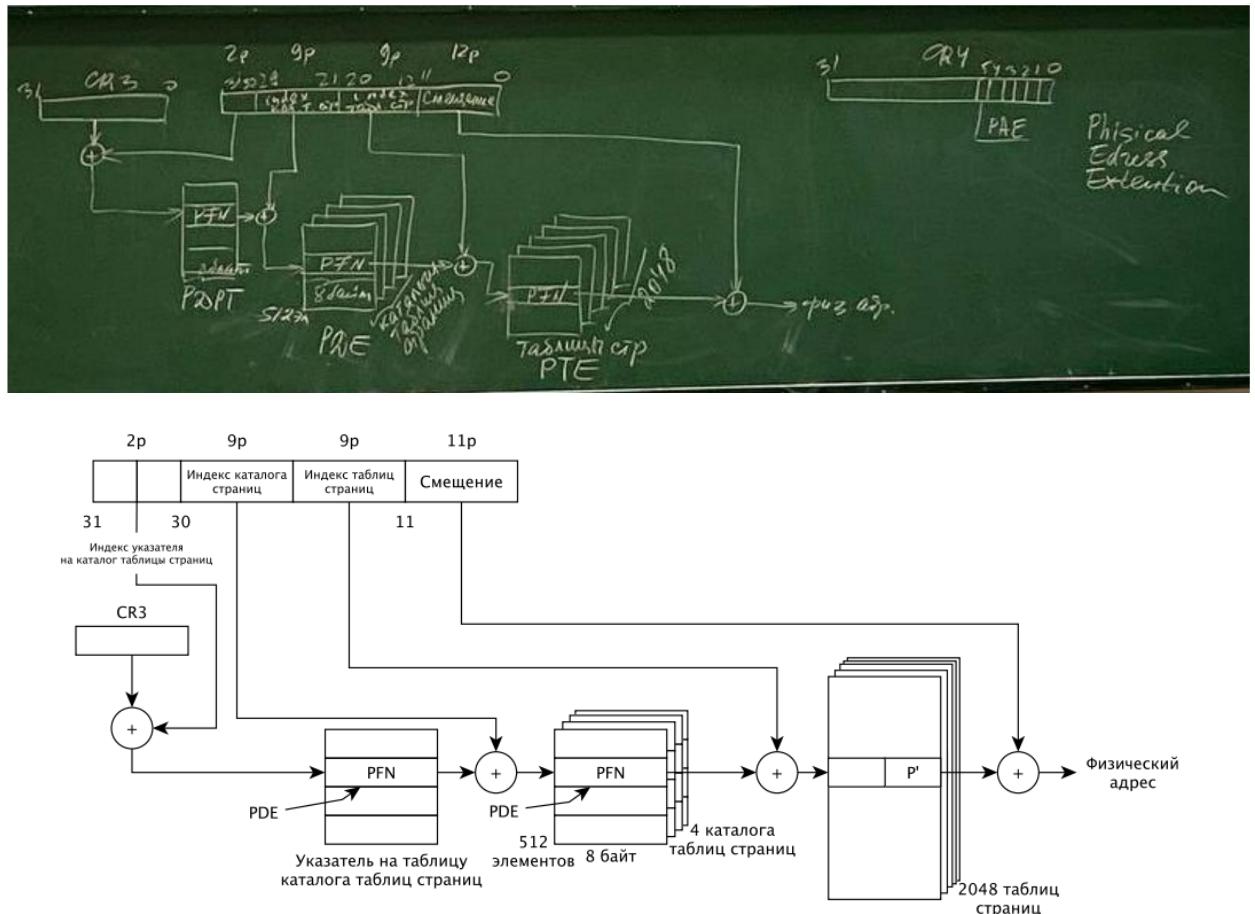
В режиме PAE 32р виртуальный адрес делится на 4 поля. Каждое такое деление связано с дополнительным обращением к ОП при преобразованиях. (Каталог ТС, базовый адрес ТС, адрес страницы). Минус, но зато ТОЛЬКО актуальные таблицы.

Появилось поле размером 2 бита – можно адресовать таблицу из 4 элементов – таблицу указателей на каталоги PDPT. В этой схеме все дескрипторы имеют размер 8 байт. Появляется возможность адресовать 4 таблицы каталогов таблиц

страниц. Все таблицы содержат дескрипторы размером 8 байт.  $2^9 = 512$  элементов в каждой таблице каталога. Итого – 2048 таблиц страниц. Эта схема в 32р введена для возможности адресации АП > 4ГБ

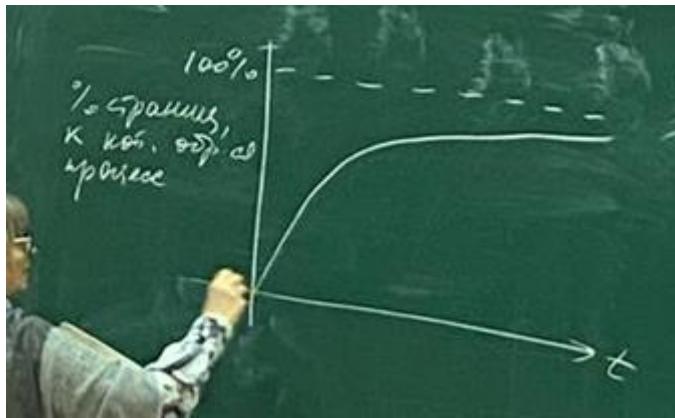
(Соломон-Руссинович). Physical Address Extension (PAE) - позволяет 32-разрядным системам осуществлять адресацию вплоть до 64 Гб физической памяти и помечать память как не содержащую исполняемый код.

Обозначения на схеме - PFN – Page Frame Number, PDE – Page Describe Entry



### Анализ страничного поведения процессов

График зависимости процента страниц, к которым обращается типичный процесс, от времени (от начала выполнения процесса до его завершения)

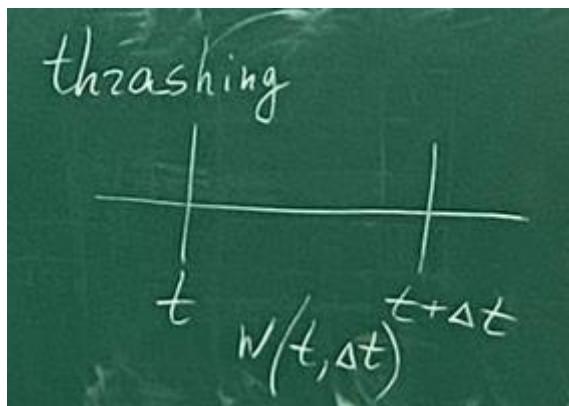


Процесс какой-то отрезок времени выполняется, но не обращается ко всем своим страницам. Причем график сначала линейный (это интенсивная подгрузка), а потом нелинейный – уже не подгружает интенсивно – «знак вопроса» (цитата)

### Теория рабочего множества

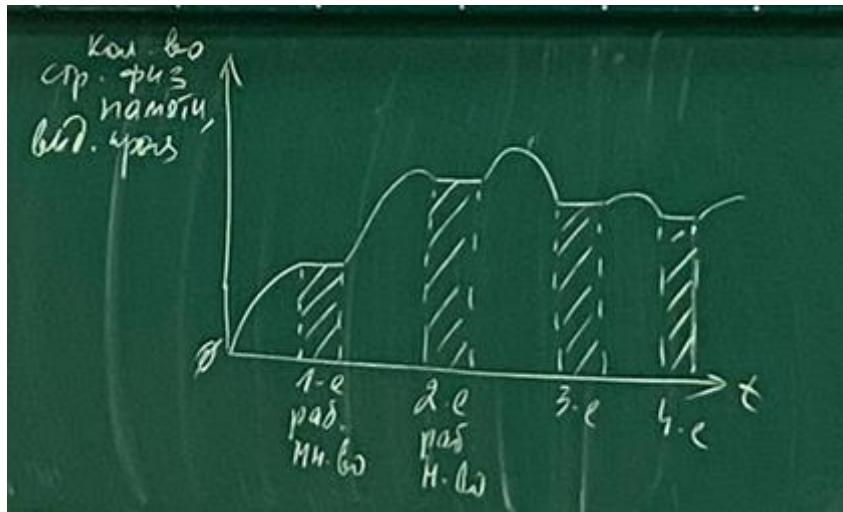
Зависимость страничных прерываний от объема памяти является обобщенной мерой страничного поведения программы. При этом за время своего выполнения программа обращается в течение времени к разному количеству страниц.

Деннинг в 1968 предложил в качестве локальной меры производительности взять число страниц, к которым программа обращается за некоторый интервал дельта  $t$  - Working set (рабочее множество)



Размер рабочего множества является монотонной функцией от дельта тэ. То есть при увеличении дельта тэ число страниц будет стремиться к некоторому пределу  $L$ , который определяет количество страниц, необходимых процессу для эффективного выполнения. Под эффективным выполнением понимают выполнение процесса без страничных прерываний.

Другими словами, если процессу удается загрузить в память свое рабочее множество, то есть все страницы, к которым он обращается в течении некоторого интервала дельта тэ, то процесс будет выполняться без страничных прерываний. Если ему это не удастся, возникнет интенсивная подкачка (пробуксовка, thrashing), то есть подкачка одних и тех же страниц.



В начальный момент (по диаграмме состояний) выделяется минимально необходимое количество страниц (скажем, 3 - код, данные, стек). Процесс начинает выполняться, при этом интенсивно подгружая страницы, пока в памяти не появится первое рабочее множество.

Затем продолжает, ему нужно новое множество, опять начинается подкачка (горбик значит, что одновременно находятся страницы и из старого, и из нового). Потом второй горизонтальный участок – второе рабочее множество, и т.д.

Working set - иногда используют название hit rate (это все из оксф. Словаря по вычислительной технике) – число страничных удач. Когда рабочее множество загружено – 100% страничных удач (только страничные удачи)

*(далее идут графики, косвенно подтверждающие факт существования рабочего множества. Думаю, стоит писать, если есть время)*

Вопрос выбора размера страницы.

Два соображения, влияющих на выбор размера страницы.

- 1) на маленькой странице выполняется больший процент команд. Чем больше страница, тем меньший процент команд на ней будет выполняться.
- 2) Чем меньше страница, тем больше таблица страниц. Каждую надо описать соответствующим дескриптором. Есть график, показывающий влияние размера страницы при фиксированном объеме памяти на число страничных прерываний.

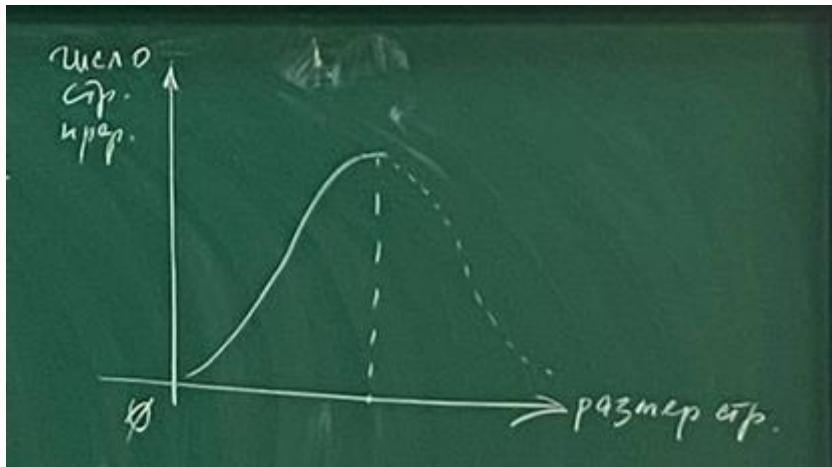
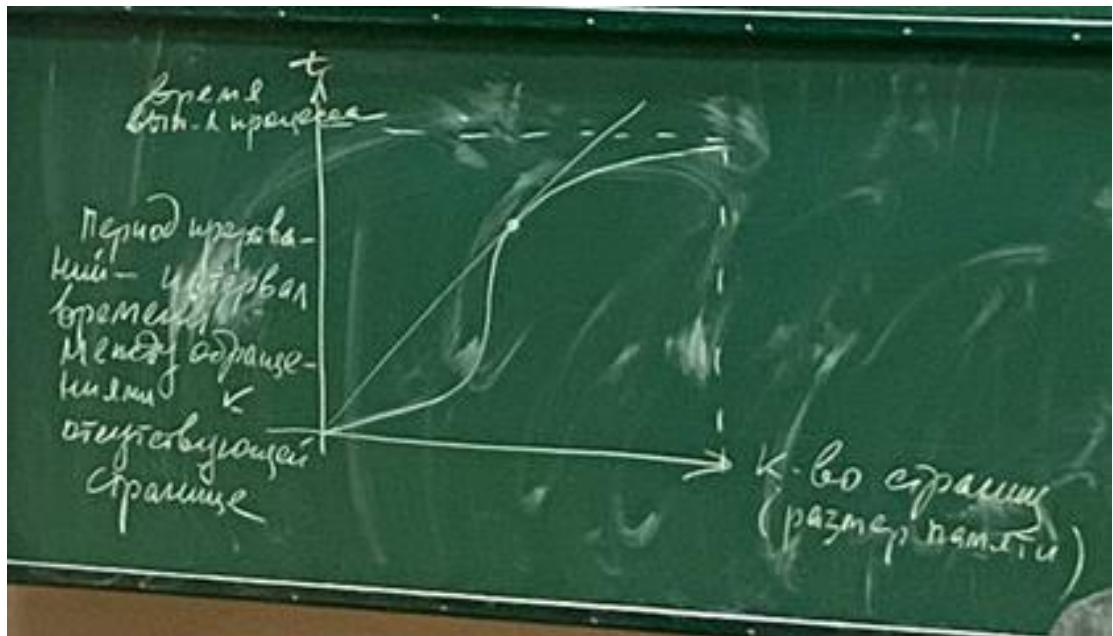


График опять же не линейный, но из него видно, что число прерываний растет с ростом размера страницы, причем интенсивно. Это связано с тем, что чем больше размер, тем меньше процент команд на ней будет выполнен и потребуется переход на другую страницу. Когда максимум достигнут, идет спад, потому что размер страницы стал сопоставим с размером программы. Очевидно, что большая страница будет содержать всю программу, а дальше пунктир, потому что никто так не делает.

Еще график – график зависимости длительности периода между прерываниями, или так называемая кривая времени жизни.



Период прерываний – это интервал времени между обращениями к отсутствующей странице.

Отмеченная точка называется точкой перегиба (или коленом). Он происходит из-за того, что в некоторый момент времени в памяти оказывается все рабочее

множество страниц процесса. Интервал между страничными прерываниями увеличивается, и именно этот интервал называется временем жизни.

Все это косвенно подтверждает факт существования рабочего множества. Контроль количества прерываний процесса является важнейшим показателем правильного выделения памяти (под правильным выделением можно понимать необходимость предоставления процессу нужного ему количества страниц)

Пример – Microsoft (открывает всему миру глаза на очевидные факты). У них есть понятие рабочего множества. Но это вероятностная величина, предсказать невозможно. Они так называют «квоту». Допустим, она устанавливается=10. Если число страничных прерываний резко увеличивается и у системы есть возможность, то квота увеличивается, и если процессу удалось загрузить свое рабочее множество, количество уменьшается. Если нет – trashing

### **Свойство локальности**

Страницное преобразование пожирает процессорное время. Почему тогда не отказались от идеи виртуальной памяти? Выигрыш – в увеличении мультизадачности (можно хранить в памяти большее количество программ)

Здесь большую роль играет кэширование. Кеш – отдельный доступ. В наших компах каждое ядро имеет 2 кэша L1, L2, а третий L3 – в кристалле и доступен всем ядрам.

В кэше хранятся физические адреса страниц, к которым были последние обращения. Это делается из эвристического (опыт) соображения, что если к странице было обращение, то вероятнее всего следующие обращения будут к этой же странице.

Это вытекает из свойства **локальности**, которым обладают наши программы. Это свойство логически объяснимо:

- 1) программы не хранятся в непрерывном адресном пространстве, но какие-то участки кода имеют последовательные адреса.
- 2) последовательность действий более вероятна, чем переход (if)
- 3) то же про данные – строки-последовательности байтов, массивы-по младшему индексу – это все хранение в последовательных адресах

Локальность бывает 2х типов

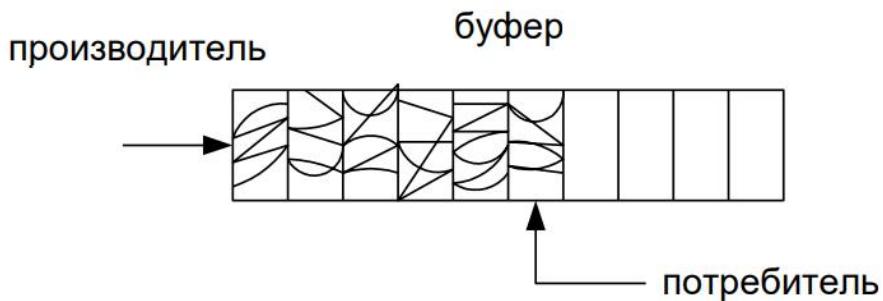
1. Временная – процесс обратившийся к одной странице наиболее вероятно в следующую единицу времени обратится к этой-же странице
2. Пространственная – процесс обратившийся к одной странице наиболее вероятно обратится к соседним страницам

5.2 Задача «Производство-потребление»: алгоритм Эд. Дейкстры, реализация на семафорах UNIX (код из лабораторной работы). Поддержка семафоров в системе UNIX.

### Задача «Производство-потребление»:

Имеется буфер фиксированного размера; процессы-производители (producer) могут только производить единичные объекты и помещать их в буфер, заполняя ячейки буфера, процессы-потребители (consumer) могут выбирать объекты из буфера по одному, освобождая ячейки буфера.

Необходимо обеспечить монопольный доступ производителей и потребителей к буферу: когда производитель помещает элемент в буфер, другой производитель или потребитель не должен иметь доступ к буферу; аналогично, когда потребитель берет элемент из буфера, то другой потребитель или производитель не могут получить доступ к буферу. В этой задаче буфер является критическим ресурсом



### алгоритм Эд. Дейкстры

Для решения этой задачи используется два считающих семафора – «БуферПолон» (buffer\_full) и «БуферПуст» (buffer\_empty) и один бинарный (bin\_sem), регулирующий доступ процессов к буферу. Семафор «БуферПолон» показывает количество элементов в буфере в любой момент времени, а семафор «БуферПуст» показывает количество пустых элементов.

```
integer N = 24; /*размер буфера*/
semaphore buffer_full, buffer_empty, bin_sem;
Инициализация семафоров:
bin_sem = 1
buffer_full = 0 /* Изначально все ячейки буфера пусты и, таким образом, количество
                  заполненных ячеек равно 0*/
buffer_empty = N /*Все ячейки буфера изначально пусты */
```

Producer:

```
{
/* производит единичный объект*/
P(buffer_empty); /*ждет, когда освободится хотя бы одна ячейка буфера*/
P(bin_sem); /*ждет, когда или другой производитель или потребитель выйдет из
              критической секции*/
/* положить в буфер */
V(bin_sem); /* освобождение критической секции*/
V(buffer_full); /* инкремент количества заполненных ячеек */
}
```

Когда производитель производит объект, значение семафора buffer\_empty уменьшается на 1, если значение buffer\_empty>0, иначе производитель блокируется в ожидании освобождения потребителем хотя бы одной ячейки буфера. Значение bin\_sem также декрементируется, чтобы обеспечить монопольный доступ к буферу. Если производитель поместил элемент в ячейку буфера, то значение семафора buffer\_full инкрементируется. Значение бинарного семафора bin\_sem устанавливается в 1, так как задача производителя выполнена и он вышел из критической секции.

Consumer:

```
{
/* выбирает из буфера единичный объект*/
P(buffer_full); /*ждет, когда будет заполнена хотя бы одна ячейка буфера*/
P(bin_sem); /*ждет, когда или потребитель, или другой производитель выйдет из
              критической секции*/
/* взять из буфера */
V(bin_sem); /* освобождение критической секции*/
V(buffer_empty); /* инкремент количества пустых ячеек */
```

Consumer:

```
{
/* выбирает из буфера единичный объект*/
P(buffer_full); /*ждет, когда будет заполнена хотя бы одна ячейка буфера*/
P(bin_sem); /*ждет, когда или потребитель, или другой производитель выйдет из
              критической секции*/
/* взять из буфера */
V(bin_sem); /* освобождение критической секции*/
V(buffer_empty); /* инкремент количества пустых ячеек */
```

реализация на семафорах UNIX (код из лабораторной работы).

Поддержка семафоров в системе UNIX.

с 2022 Сравнение программных каналов, очередей сообщений и разделяемых сегментов.

сравнение

## 6 билет

6.1 Понятие процесса. Процесс как единица декомпозиции системы. Диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра. Планирование и диспетчеризация. Классификация алгоритмов планирования. Примеры алгоритмов планирования, соотнесенные с типами ОС. Процессы и потоки. Типы потоков.

**Понятие процесса. Процесс как единица декомпозиции системы. Диаграмма состояний процесса с демонстрацией действий, выполняемых в режиме ядра.**

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHth-sB8TTy1Xvv9SN9zM/edit#bookmark=id.i52s8immrnx>

**Планирование и диспетчеризация.**

В многозадачных системах в состоянии готовности может быть много процессов, необходимо организовать очередь для выделения дефицитного ресурса – процессорного времени.

Планирование – постановка процессов в очередь по выбранному дисц. плану (в хорошо структурированных ОС есть планировщик scheduler – модуль ядра, выполняющий указанную работу)

Диспетчеризация – непосредственное выделение какого-либо ресурса.

Дисциплины планирования связаны с соответствующими типами ОС (к однозадачным вообще не относится). Два вида:

- мультизадачные системы пакетной обработки.
- системы разделения времени (мультизадачные во втором случае нельзя говорить)

В системах пакетной обработки пользователь отделен от процесса выполнения программы. Рисунок (человек {пакет} квадрат). В системах разделения времени – наоборот: пользователь непосредственно связан – запускает, вводит данные, получает результат и реагирует – интерактив.

**Классификация алгоритмов планирования.**

1. Без переключения/С переключением
  - процесс может выполняться от начала до конца, а может быть снят с выполнения (например при истечении кванта) и возвращен в очередь готовых процессов
2. С приоритетами/Без приоритетов
3. Без вытеснения/С вытеснением
  - процесс может быть вытеснен другим процессом с более высоким приоритетом, процесс снимается с выполнения и возвращается в очередь готовых процессов, а процессорное время передается процессу с более высоким приоритетом
  - (вытеснение невозможно в системе без приоритетов)!

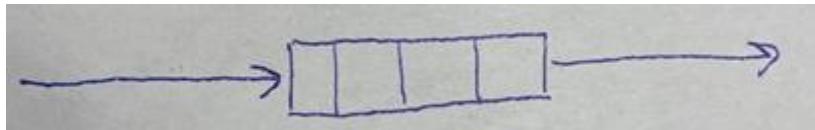
*Приоритеты:*

1. Статические - назначаются до выполнения и не меняются в процессе.
2. Динамические - меняются в процессе выполнения.

**Примеры алгоритмов планирования, соотнесенные с типами ОС.**

В системах пакетной обработки:

1. Алгоритм fifo - first in first out (или fcfs – first come first server):
  - без переключения, без вытеснения, без приоритетов.
  - Процесс, получив процессорное время, выполняется от начал и до конца. После блокировки будет поставлен в конец очереди.



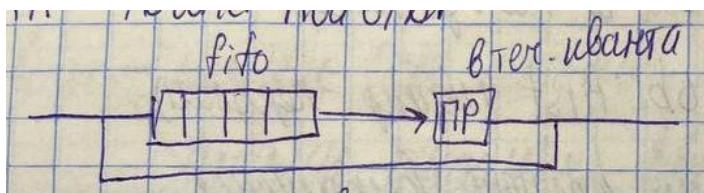
2. SJF - shortest job first (наискорейший (кратчайший) – первым)
  - без вытеснения, без переключения, с приоритетом.
  - По заявленному времени выполнения, то есть времена априорные – до опыта. Система отдавала предпочтение коротким процессам, что позволяло улучшить такой дурацкий показатель, как количество процессов в единицу времени (мера производительности)
  - Приводило к тому, что задания с большим процессорным временем все время откладывались в конец очереди — бесконечное откладывание (ситуация, когда процесс никогда не получает необходимых для выполнения ресурсов (точнее, кванта времени))
3. SRT – shortest remain time (наименьшее оставшееся время)
  - с приоритетом, с вытеснением
  - Выполняющийся процесс может быть прерван, если в очередь поступил процесс с меньшим оставшимся процессорным временем выполнения (заявленное время – полученное=оставшееся и оно сравнивается)
  - Еще хуже по бесконечному откладыванию
4. HRN – highest response ratio next (наибольшее относительное время)

- Приоритеты пересчитываются по функции  $p=(tw+ts)/ts$ , где  $tw$  – время ожидания в очереди,  $ts$ -заявленное время выполнения программы.
- То есть в зависимости от времени ожидания, следовательно, не будет бесконечного откладывания. Приоритет повышается пропорционально времени ожидания (времени простоя)

### В системах разделения времени:

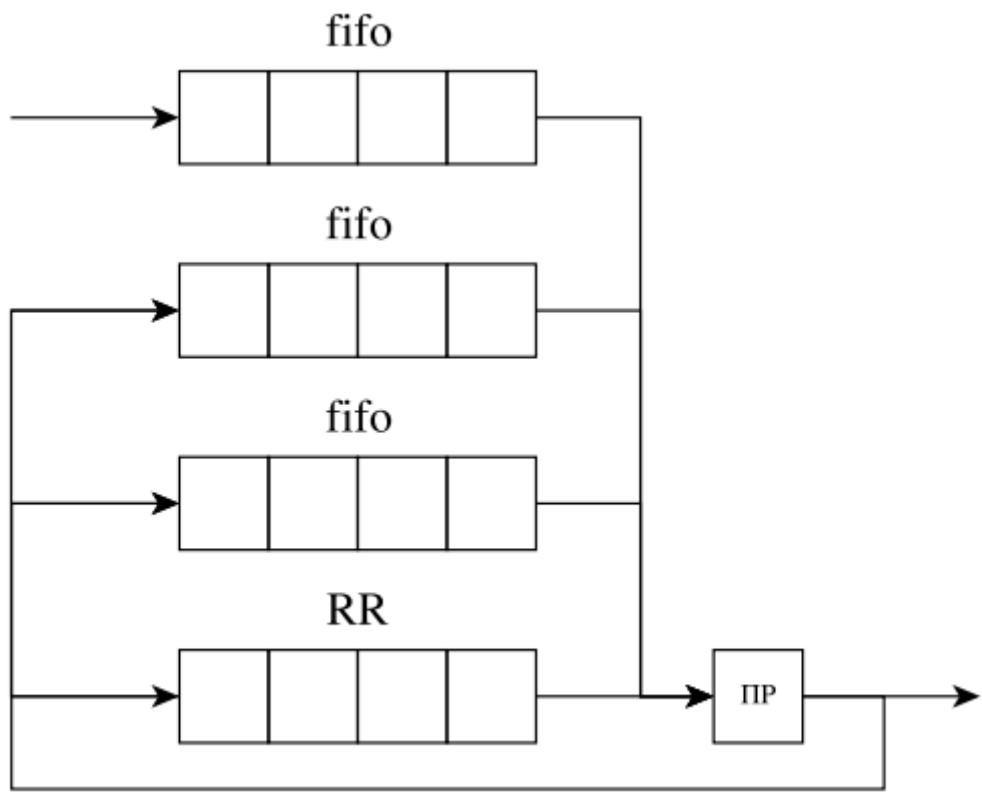
Важно гарантировать время ответа системы – пользователь не может ждать. Процессорное время стали квантовать (time slicing). Процесс выполняется до тех пор, пока не истек квант, не начался процесс ввода/вывода или не вытеснен другим высокоприоритетным процессом

1. RR – round robin - циклическое планирование
  - без приоритетов, без вытеснения, с переключением
  - процессы выстраиваются в очередь, но по истечении кванта ставятся в конец очереди.
  - основной “-” - статический квант времени
  - Простой, но есть много вариантов
  - В современных системах - путем назначения кванта времени с учетом особенностей (те динамическая величина кванта), но это требует времени
    - i. min-max RR (MMRR)
    - ii. average max RR (AMRR)
    - iii. efficiency dynamic RR (EDRR)

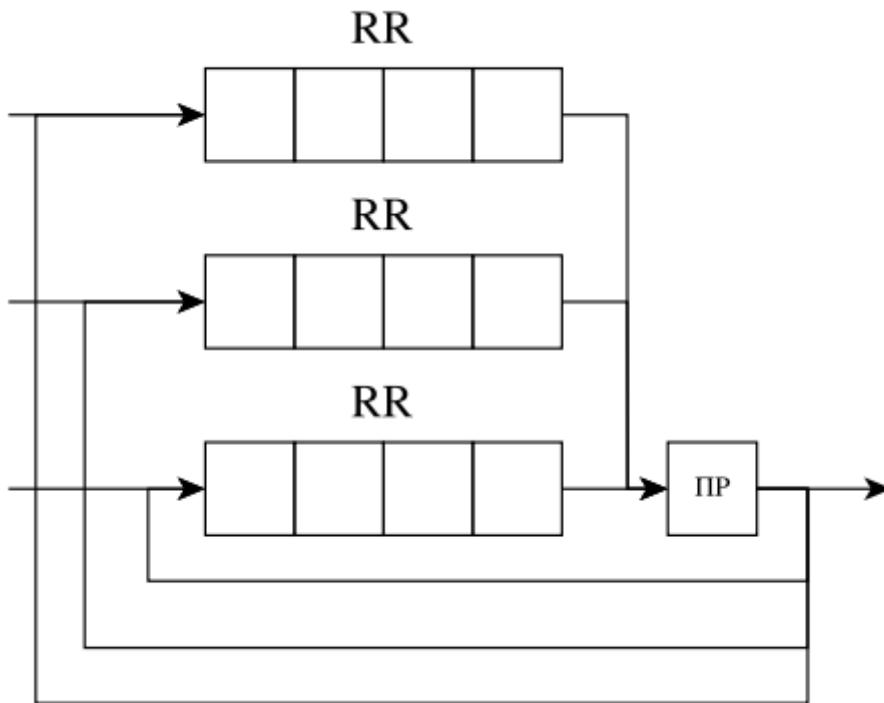


2. Алгоритм приоритетного планирования (адаптивное планирование). (Называть это многоуровневой очередью не совсем правильно (цитата))
 

(!у нас на лекции стрелочки слева к RR вроде не было)



- Много очередей с разными приоритетами (fifo), последняя - RR
  - предположительно: с наивысшим приоритетом поступают только что созданные процессы или завершившие блокировку в ожидании завершения ввода/вывода.
  - Для 1 очереди квант процессорного времени выбирается таким образом, чтобы наибольшее количество процессов успело или завершится, или выполнить запрос на ввод/вывод. Если процесс не успел завершится за выделенный квант процессорного времени, он поступает в следующую очередь с более низким приоритетом и так далее пока не окажется в очереди с самым низким приоритетом, которая будет работать по принципу RR.
  - В результате в последней очереди окажутся процессы, которым нужно много процессорного времени (процессы с большим объемом вычислений)
  - В последней очереди также крутится так называемый холостой процесс, поскольку система не может ничего не делать.
  - *Пересчитываться могут только пользовательские приоритеты (приложений). Стремится быть справедливым, чтобы у всех процессов было примерно одинаковое время, но также учитывается время простоя процесса. В старом UNIX используется формула. В windows много очередей RR с разными приоритетами*
3. (этого у нас вроде не было, а по рисунку это вообще похоже на замечание выше про windows - много очередей RR с разными приоритетами) Интерактивные – процессы, запрашивающие ввод-вывод с клавиатуры, мыши. Самые высокоприоритетные операции



### Процессы и потоки. Типы потоков.

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHthsB8TTy1Xvv9SN9zM/edit#bookmark=id.t9fv6eo06frm>

6.2 Обеспечение монопольного доступа к разделяемым данным в задаче «читатели-писатели» : реализация на базе Win32 API (пример кодов лабораторной работы «читатели-писатели» для ОС Windows).

6.2 (2021) Задача «читатели-писатели» : реализация на базе Win32 API (пример кодов лабораторной работы «читатели-писатели» для ОС Windows). Сравнение мьютексов и семафоров.

### Задача «читатели-писатели». Монитор Хоара

Задача «Читатели-писатели» является одной из известнейших в ОС. Для этой задачи характерно наличие двух типов процессов: процессов «читателей», которые могут только читать данные, и процессов «писателей», которые могут только изменять данные. Читатели могут работать параллельно, поскольку они друг другу не мешают, а писатели могут работать только в режиме монопольного доступа: только один писатель может получить доступ к разделяемой переменной, причем, когда работает писатель, то другие писатели и читатели не могут получить доступ к этой переменной. Рассмотрим монитор Хоара «Читатели-писатели», для которого характерно наличие четырех процедур: start\_read(), stop\_read(), start\_write(), stop\_write() (листинг 3).

```

RESOURCE MONITOR;
var
    active_readers : integer;
    active_writer : logical;
    can_read, can_write : conditional;
procedure star_read
begin
    if (active_writer or turn(can_write)) then
        wait(can_read);
    active_readers++; //инкремент читателей
    signal(can_read);
end;
procedure stop_read
begin
    active_readers--; //декремент читателей
    if (active_readers == 0) then signal(can_write);
end;
procedure start_write
begin
    if ((active_readers > 0) or active_writer) then wait(can_write);
    active_writer:= true;
end;
procedure stop_write

```

```

begin
    active_writer = false;
    if (turn(can_read) then
        signal(can_read)
    else signal(can_write);
end;
begin
    active_readers = 0;
    active_writer = false;
end.

```

Когда число читателей равно 0, процесс писатель получает возможность начать работу. Новый процесс читатель не сможет начать свою работу пока работает процесс писатель и не появится истинное значение условия can\_read.

Писатель может начать свою работу, когда условие can\_write станет равно истине (true).

Когда процессу читателю нужно выполнить чтение, он вызывает процедуру start\_read. Если читатель заканчивает читать, то он вызывает процедуру stop\_read. При входе в процедуру star\_read новый процесс читатель сможет начать работать, если нет процесса писателя, изменяющего данные, в которых заинтересован читатель, и нет писателей, ждущих свою очередь (turn(can\_write)), чтобы изменить эти данные. Второе условие нужно для предотвращения бесконечного откладывания процессов писателей в очереди писателей.

Процедура start\_read завершается выдачей сигнала signal(can\_read), чтобы следующий читатель в очереди читателей смог начать чтение. Каждый следующий читатель, начав чтение выдает signal(can\_read), активизирует следующего читателя в очереди читателей. В результате возникает цепная реакция активизации читателей и она будет идти до тех пор, пока не активизируются все ожидающие читатели.

«Цепная реакция» читателей является отличительной особенностью данного решения, которое эффективно «запускает» параллельное выполнение читателей. Процедура stop\_read уменьшает количество активных читателей: читателей, начавших чтение. После ее многократного выполнения количество читателей может стать равным нулю. **Если число читателей равно нулю, выполняется signal(can\_write), активизирующий писателя из очереди писателей.**

**Когда писателю необходимо выполнить запись, он вызывает процедуру start\_write.** Для обеспечения монопольного доступа писателя к разделяемым данным, если есть читающие процессы или другой активный писатель, то писателю придется подождать, когда будет установлено значение «истина» в переменной типа условие can\_write. Когда писатель получает возможность работать логической переменной can\_write присваивается значение «истина», что заблокирует доступ других процессов писателей к разделяемым данным.

**Когда писатель заканчивает работу, предпочтение отдается читателям при условии, что очередь ждущих читателей не пуста. Иначе для писателей устанавливается переменная can\_write. Таким образом исключается бесконечное откладывание читателей.**

**Реализация на базе Win32 API (пример кодов лабораторной работы «читатели-писатели» для ОС Windows).**

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

#define N_READERS 5
#define N_WRITERS 3

#define N_ITERS 4

#define MIN_READER_SLEEP 300
#define MAX_READER_SLEEP 2000
#define MIN_WRITER_SLEEP 100
#define MAX_WRITER_SLEEP 1500

HANDLE mutex;
HANDLE can_read;
HANDLE can_write;

LONG waiting_writers_amount = 0;
```

```

LONG waiting_readers_amount = 0;
LONG active_readers_amount = 0;

bool active_writer = false;

int value = 0;

void start_read()
{
    InterlockedIncrement(&waiting_readers_amount);

    if (active_writer || (WaitForSingleObject(can_write, 0) == WAIT_OBJECT_0 &&
    waiting_writers_amount))
    {
        WaitForSingleObject(can_read, INFINITE);
    }

    WaitForSingleObject(mutex, INFINITE);

    InterlockedDecrement(&waiting_readers_amount);
    InterlockedIncrement(&active_readers_amount);

    SetEvent(can_read);
    ReleaseMutex(mutex);
}

void stop_read()
{
    InterlockedDecrement(&active_readers_amount);

    if (active_readers_amount == 0)
    {
        SetEvent(can_write);
    }
}

void start_write(void)
{
    InterlockedIncrement(&waiting_writers_amount);

    if (active_writer || active_readers_amount > 0)
    {
        WaitForSingleObject(can_write, INFINITE);
    }

    InterlockedDecrement(&waiting_writers_amount);

    active_writer = true;

    ResetEvent(can_write); // 
}

```

```

void stop_write(void)
{
    active_writer = false;

    if (waiting_readers_amount)
    {
        SetEvent(can_read);
    }
    else
    {
        SetEvent(can_write);
    }
}

DWORD WINAPI reader_run(CONST LPVOID param)
{
    int reader_id = (int)param;
    srand(time(NULL) + reader_id);

    int sleep_time;

    for (size_t i = 0; i < N_ITERS; i++)
    {
        sleep_time = MIN_READER_SLEEP + rand() % (MAX_READER_SLEEP -
MIN_READER_SLEEP);
        Sleep(sleep_time);
        start_read();
        printf("Reader %d read: %3d || Sleep time: %dms\n", reader_id, value,
sleep_time);
        stop_read();
    }

    return 0;
}

DWORD WINAPI writer_run(CONST LPVOID param)
{
    int writer_id = (int)param;
    srand(time(NULL) + writer_id + N_READERS);

    int sleep_time;

    for (size_t i = 0; i < N_ITERS; i++)
    {
        sleep_time = MIN_WRITER_SLEEP + rand() % (MAX_WRITER_SLEEP -
MIN_WRITER_SLEEP);
        Sleep(sleep_time);
        start_write();
        ++value;
        printf("Writer %d wrote: %3d || Sleep time: %dms\n", writer_id, value,

```

```

sleep_time);
    stop_write();
}
return 0;
}

int main()
{
setbuf(stdout, NULL);

HANDLE readers_threads[N_READERS];
HANDLE writers_threads[N_WRITERS];

if ((mutex = CreateMutex(NULL, FALSE, NULL)) == NULL)
{
    perror("CreateMutex error");

    return -1;
}

if ((can_read = CreateEvent(NULL, FALSE, FALSE, NULL)) == NULL)
{
    perror("CreateEvent (can_read) error");
    return -1;
}
if ((can_write = CreateEvent(NULL, TRUE, FALSE, NULL)) == NULL)
{
    perror("CreateEvent (can_write) error");
    return -1;
}

for (size_t i = 0; i < N_READERS; i++)
{
    readers_threads[i] = CreateThread(NULL, 0, &reader_run, (LPVOID)i, 0,
NULL);
    if (readers_threads[i] == NULL)
    {
        perror("CreateThread (reader) error");
        return -1;
    }
}

for (size_t i = 0; i < N_WRITERS; i++)
{
    writers_threads[i] = CreateThread(NULL, 0, writer_run, (LPVOID)i, 0, NULL);
    if (writers_threads[i] == NULL)
    {
        perror("CreateThread (writer) error");
        return -1;
    }
}

```

```

}

WaitForMultipleObjects(N_READERS, readers_threads, TRUE, INFINITE);
WaitForMultipleObjects(N_WRITERS, writers_threads, TRUE, INFINITE);

CloseHandle(mutex);
CloseHandle(can_read);
CloseHandle(can_write);

for (size_t i = 0; i < N_READERS; i++)
    CloseHandle(readers_threads[i]);

for (size_t i = 0; i < N_WRITERS; i++)
    CloseHandle(writers_threads[i]);

return 0;
}

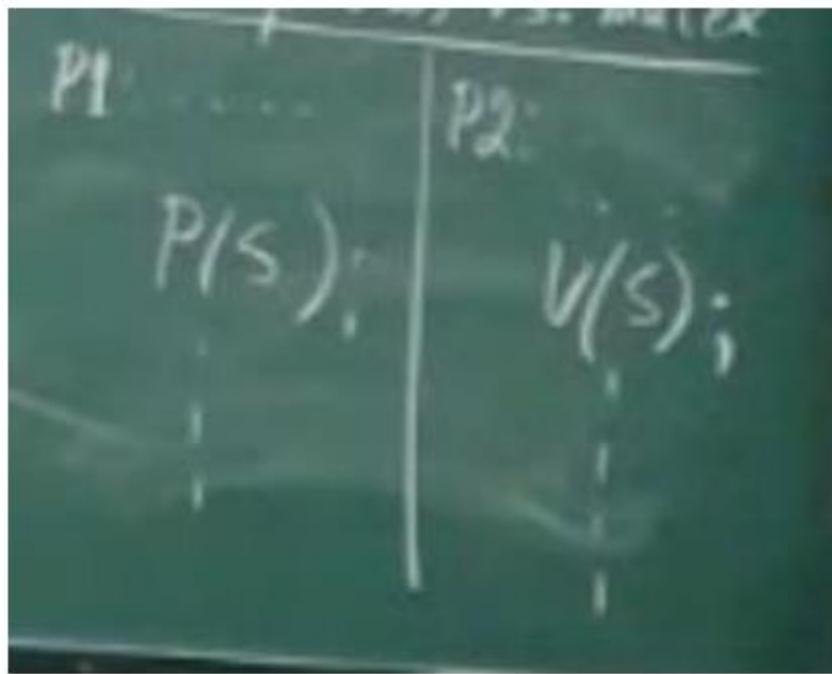
```

CanWrite должен быть с ручным сбросом (по последней информации наоборот - canread должен быть с ручным. если читатели с ручным, то второй аргумент написать true – и сделать reset в stop\_read). Такая реализация, как считает Рязанова, лучше. Слова Рязановой: Для писателей можно использовать событие с ручным сбросом. Тогда resetevent переведет событие в занятое состояние и логическая переменная вообще-то станет не нужной.

### **Сравнение мьютексов (M) и семафоров (C).**

Очень часто бинарные семафоры называют мьютексами. Это ошибка.

1. У мьютексов всегда есть владелец. Владельцем является процесс, который захватил M (get\_mutex, lock\_mutex – неважно). Только владелец M может M освободить (unlock). Для C такого понятия не существует. С может захватить один процесс, а освободить – совершенно другой. То есть для C (Для M – нет) может быть написано: ... Это основное отличие, которое делает С особенноми.



2. в отличие от С, на мютексах определена инверсия приоритетов. Это связано с тем, что никакой другой более высокоприоритетный процесс не сможет перехватить М.
3. в силу того, что М может быть освобожден только процессом захватившем м, никакого случайного удаления процесса, захватившего М это невозможно. Для С это не так. Если захвативший М процесс умер, все остальные могут быть заблокированы навсегда, поэтому unix очень аккуратно там за всем следит.

## 7 билет

7.1 Управление виртуальной памятью: распределение памяти сегментами по запросам: схема преобразования виртуального адреса, способы организации таблиц сегментов, стратегии выбора разделов памяти для загрузки сегментов, алгоритмы и особенности замещения сегментов.

### Пreamble

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

### 3 схемы управления виртуальной памятью

1. Управление памятью страницами по запросу
2. Управление памятью сегментами по запросу
3. Управление памятью сегментами, поделенными на страницы по запросу.

**ПО ЗАПРОСУ:** В момент порождения процесса система должна ему выделить минимально необходимый объем памяти, а в результате таких запросов загружаются необходимые участки кода.

**Запрос** – страничное прерывание, которое возникает при попытке доступа к незагруженной в память странице. При его обработке происходит пэйджинг.

**Пейджинг** – загрузка с диска новых и замещение старых страниц.

**Страница** – является единицей физического деления памяти. Её размер устанавливается системой. У страницы размер не обязательно 4Кб. Просто такой размер оптимален по количеству страничных прерываний.

**Сегмент** – является единицей логического деления памяти. Её размер определяется объемом кода. Любой сегмент создается такого размера, который требуется программе.

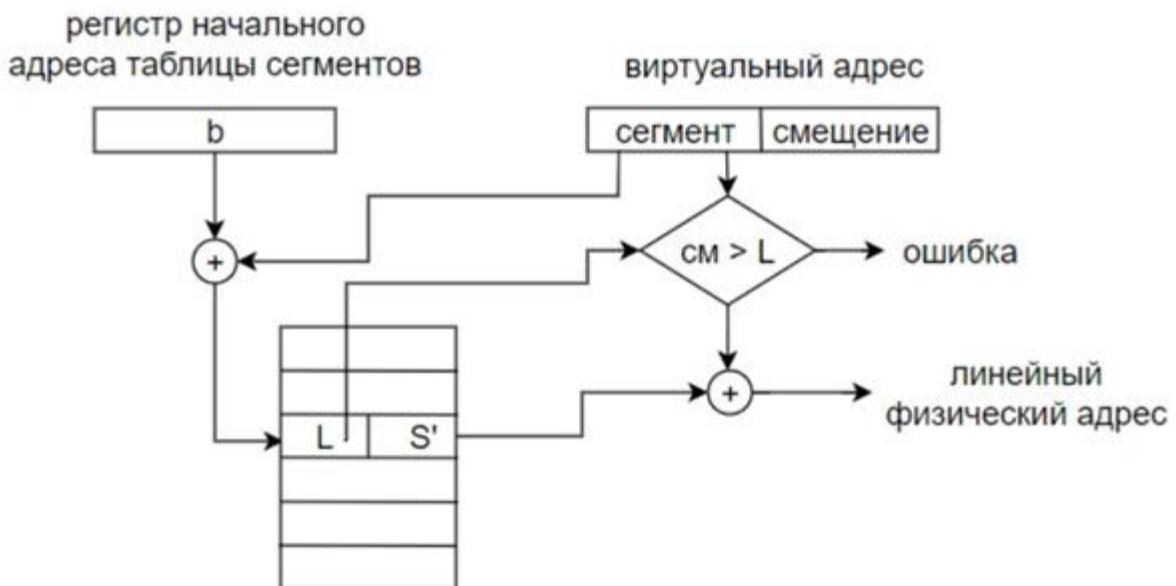
#### **распределение памяти сегментами по запросам:**

*Сегмент вроде как является более логичным для реализации в системе, поскольку действительно мы выполняем какие-то программы определённого размера и безусловно сама идея буквально *прям вот находится на поверхности*.*

Очевидно, что у сегмента не может быть безграничного размера, размер сегмента ограничен аппаратными соображениями.

#### **схема преобразования виртуального адреса**

В самом общем виде схема выглядит следующим образом



СТРЕЛКА В РОМБ ИДЕТ ОТ СМЕЩЕНИЯ

- В процессоре должен быть регистр, куда записывается начальный адрес таблицы сегментов процесса.
- Таблица сегментов содержит дескрипторы сегментов адресного пространства процесса.
- Дескрипторы сегментов содержат поле флагов, поле, определяющее размер сегмента, адрес сегмента в физической памяти. *дескриптор сегмента мы с вами видели в lr перевода компьютера из реального режима в защищенный. общее название того, что мы видели в сегментах с которыми работали это права доступа. То, что у Рудакова Финогенова записывается как можно читать, можно писать, можно выполнять это права доступа. У нас всего 3 типа (read, write, execute). Кроме того, вы видели дополнительные флаги, которые управляют своппингом. Если мы говорим о страницах, то принято говорить о пейджинге, если мы говорим о сегментах, то принято говорить о свопе.*
- Номер сегмента из виртуального адреса – смещение к дескриптору сегмента в таблице сегментов.
- Виртуальный адрес делится на 2 поля – сегмент и смещение. Смещение определяет размер сегмента. Смещение не может быть безгранично большим.
- Соответственно производится контроль обращения процесса к своему сегменту, и лимит как раз используется для такого контроля. Выполняется проверка. Если смещение выходит за размер сегмента, возникает ошибка, если все нормально, то получаем линейный физический адрес.

Таким образом и осуществляется защита адресных пространств процессов. **Ни один процесс не может обратиться в адресное пространство другого процесса.**

В дескрипторах в лабах x86 limit определяет размер сегмента. Оно лимитирует возможные размеры. В 64 архитектуре в режиме лонг только страничная организация.

*а как же осуществляется контроль при страничном преобразовании? Процесс не может обратиться к страницам, которые не описаны в его таблицах страниц. Это просто невозможно. Но здесь конечно имеется некий вариант, потому что сегмент - это единица логического деления памяти.*

### способы организации таблиц сегментов

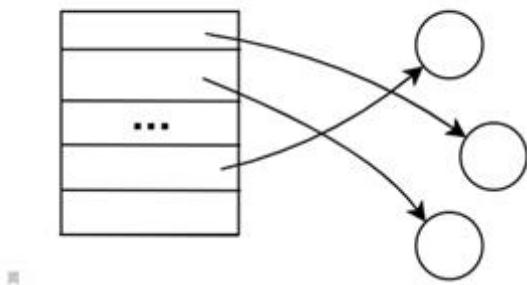
3 подхода к организации таблиц дескрипторов сегментов

*(вроде имя сегмента - его индекс в таблице)*

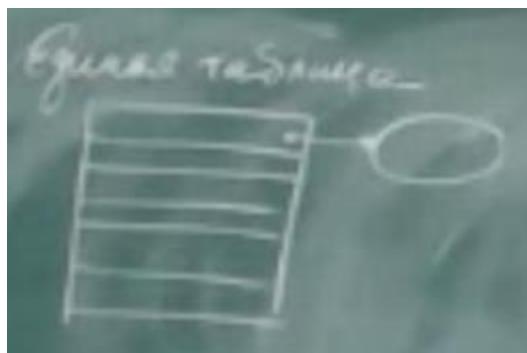
1. Единая таблица.
- Одна единая таблица которая содержит все дескрипторы сегментов выполняемых программ.
  - Каждый дескриптор в ней описывает сегмент физической памяти
  - У каждого такого сегмента в системе существует одно единственное глобальное имя, которое является идентификатором дескриптора таблицы.
  - Дескриптор должен содержать список прав доступа для всех процессов, которые могут использовать данный сегмент (список может быть очень большим)

- Такая система не является гибкой, все процессы обращаются к сегменту по одному единственному его имени

## Единая таблица



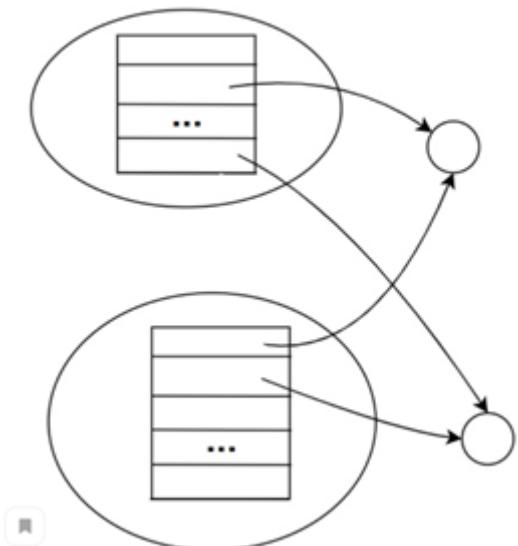
(на нашей лекции такая картинка)



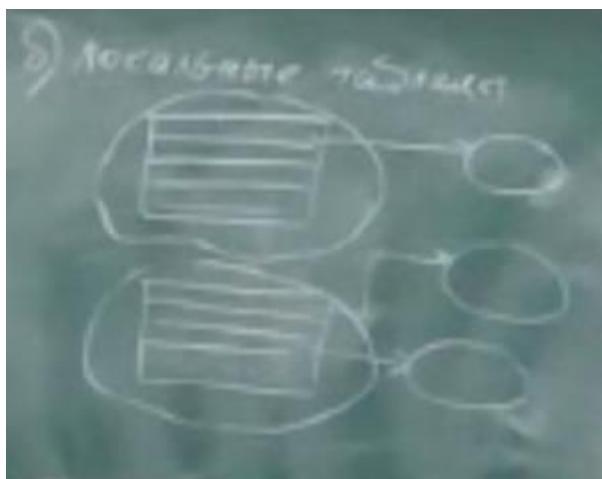
### 2. Локальные таблицы.

- Каждая локальная таблица описывает адресное пространство (определяет среду) отдельного процесса (ап процесса разбито на сегменты).
- В этих таблицах необходимо указывать выделенные сегменты физической памяти.
- чтобы процесс мог обращаться к сегментам других программ, эти сегменты должны быть описаны в его локальных таблицах. Получается, что один сегмент в системе может иметь несколько разных имён и дескрипторов. Это безусловно усложняет работу с сегментами в системе.

## Локальные таблицы



(на нашей лекции была такая картинка)

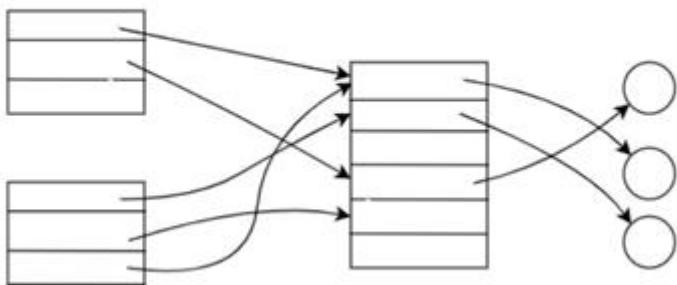


3. Локальные таблицы и одна глобальная таблица.
  - Локальная таблица описывает адресное пространство отдельного процесса
  - дескрипторы ЛТ содержат ссылку на дескриптор сегмента в глобальной таблице дескрипторов.
  - глобальная таблица содержит уже адреса сегментов в физической памяти

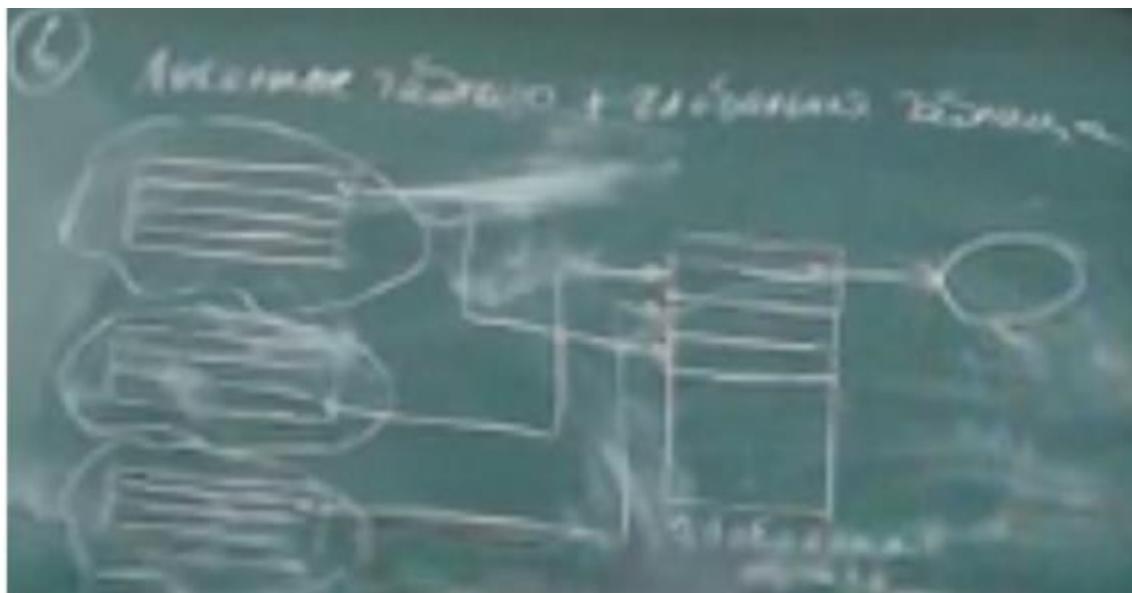
Вопрос: Так ли это в Intel? Мы рассматривали GDT, LDT, так ли там? - Мой ответ - Да

Или можно описать так: Каждый сегмент имеет главный (центральный) дескриптор, который содержит все характеристики его физического размещения (размер, начальный адрес, флаги). Кроме главного дескриптора сегмент имеет локальные дескрипторы в тех виртуальных адресных пространствах, где он доступен. Такой локальный дескриптор содержит информацию, относящуюся к данному процессу (например, права доступа) и содержит указатель на глобальный дескриптор.

## Локальные таблицы + глобальная таблица



(на нашей лекции была такая картинка)



### Страницы vs сегменты

Сегментация появилась в истории параллельно со страничным преобразованием.

#### 1. Первый + сегментами

Основной недостаток страницами – коллективное использование – на уровне страниц теряется принадлежность страницы конкретному процессу. Каждой странице надо указывать права доступа для различных процессов.

С сегментом проблемы нет, так как это логическое деление. Процессы, которые желают получить доступ к разделяемым сегментам должны просто иметь указатель на сегмент.

#### 2. Второй + сегментами

сегмент является логической единицей деления памяти. Он определяется соответствующей программой и связан с логикой программы, в отличие от страниц.

### **3. - сегментами**

Так как сегменты определяются размером программного кода, то они имеют самые разные размеры. Если мы выполняем своппинг, то выгрузив один сегмент, в освободившееся адресное пространство нужно загрузить новый сегмент. При загрузке останется адресное пространство в которое мы не сможем что-то загрузить. При этом этот своппинг выполняется постоянно. Понятно что у нас перемещаемые сегменты, но всё равно любое перемещение приводит к соответствующим затратам.

### **стратегии выбора разделов памяти для загрузки сегментов**

Стратегии выбора раздела

1. первый подходящий (по размеру отдел)
2. самый тесный (ближе всего к размеру программы)
3. самый широкий (в этом разделе останется еще место для загрузки еще одной программы).

### **алгоритмы замещения сегментов.**

Если вся память распределена и мы не смогли найти разделы нужного размера, то надо выгрузить какие-то сегменты. Выгрузка сегментов выполняется по тем же соображениям, по которым выполняется выгрузка страниц (будет рассмотрено далее).

Самым лучшим является трудно осуществимый алгоритм LRU.

### **Особенность замещения сегментов**

При этом у замещения сегментов есть особенность: может оказаться, что замещаемый сегмент имеет недостаточно большой размер для того чтобы в освободившуюся область мы могли загрузить большой сегмент. Вместо одного сегмента нам придётся выгрузить несколько. Это дополнительные накладные расходы. То есть проблема замещения сегментов значительно сложнее, чем замещения страниц, хотя все рассуждения остаются в силе.

(именно эта особенность привела к появлению третьего способа управления виртуальной памятью: сегментами, поделенными на страницы по запросу)

### **Алгоритмы замещения страниц**

1. Выталкивание случайной (первой попавшейся) страницы (во вторичную память).
  - Характеризуется малыми издержками
  - не является дискриминационным
  - Минусы: может быть вытолкнута часто используемая или только что загруженная.
2. Fifo.

- Каждой странице присваивается временная метка или организуется связный список типа очередь. То есть вновь загруженные страницы оказываются в хвосте и постепенно перемещаются в начало и та, что в начале – кандидат на выгрузку.
  - Минус: все еще возможно выталкивание часто используемой.
  - плюс: Но зато вновь загруженная уже не будет выгружена)
  - минус «аномалия fifo»: наше априорное утверждение, что при увеличении объема доступной памяти количество прерываний страниц уменьшается, может оказаться ложным ПРИМЕР:

+ - отмечены страничные неудачи (внизу), если нет - страничная удача, а наверху – что надо загрузить. В кружок – что вытесняем

а) Вот тут доступно 3, Итого  $9/12 = 75\%$  неудач

б) Увеличим память на 1 страницу. Итого 10/12 – 83%. Это и называется аномалия fifo

P	1	2	3	4	5	6	7	8	9	10	11	12
	4	1	3	1	2	1	1	4	1	5	1	15
$\uparrow$	4	1	3	2	1	4	1	5	4	1	5	1
$M=4$	4	3	2	2	2	1	5	4	3	2	1	
$\downarrow$	4	3	3	3	2	1	5	4	3	2		
	4	4	4	4	3	2	1	5	4	3	2	
	+	+	+	+	+	+	+	+	+	+	+	+

$10/12 \approx 83\%$

3. LRU – least recently used - наименее используемый в последнее время

- Когда к странице обращаются, она отправляется в конец списка (в хвост). Если используются временные метки, то временная метка обновляется.
- Пример

a) Промоделируем работу этого алгоритма на той же траектории страниц

Так, на 8 шаге выполняется обращение к 4, она уже в памяти, поэтому перемещается в конец списка (если мы считаем, что клеточки – модель связного списка), то есть страничного прерывания не происходит (эта ситуация называется страничной удачей).

P	1	2	3	4	5	6	7	8	9	10	11	12
	4	1	3	1	2	1	4	1	5	1	5	1
$\uparrow$	4	1	3	2	1	4	1	5	4	3	2	1
$M=3$	4	3	2	1	4	3	5	4	3	2	1	
$\downarrow$	4	3	2	1	4	3	2	1	5	4	3	
	4	3	2	1	4	3	2	1	5	4	3	
	+	+	+	+	+	+	+	+	+	+	+	+

б) Увеличим память на 1 страницу

Здесь также происходит страничная удача, поскольку 4 страница в памяти. Обращение к 3 приведет к тому, что она перемещается в хвост.

- + Все сработало в соответствии с нашими предположениями: увеличение памяти привело к уменьшению числа страничных прерываний. Это связано с тем, что алгоритм соответствует свойству локальности: если было обращение к странице, то наиболее вероятно, что следующие обращения будут к этой же странице
- 2 строки второго рисунка полностью соответствуют первым 2 строкам первого рисунка. Это называется свойство включения – если какая-то страница выбрана в реализации  $L(P, M, t)$  [здесь обозначения page memory time], то эта же страница будет выбрана в реализации  $L(P, M+1, t)$
- Алгоритм LRU относится к классу методов вытеснения, которые называются стековыми алгоритмами.
- крайне затратный. Для его реализации надо либо модифицировать временную метку при каждом обращении к странице, либо редактировать связный список, перемещая страницу в конец.

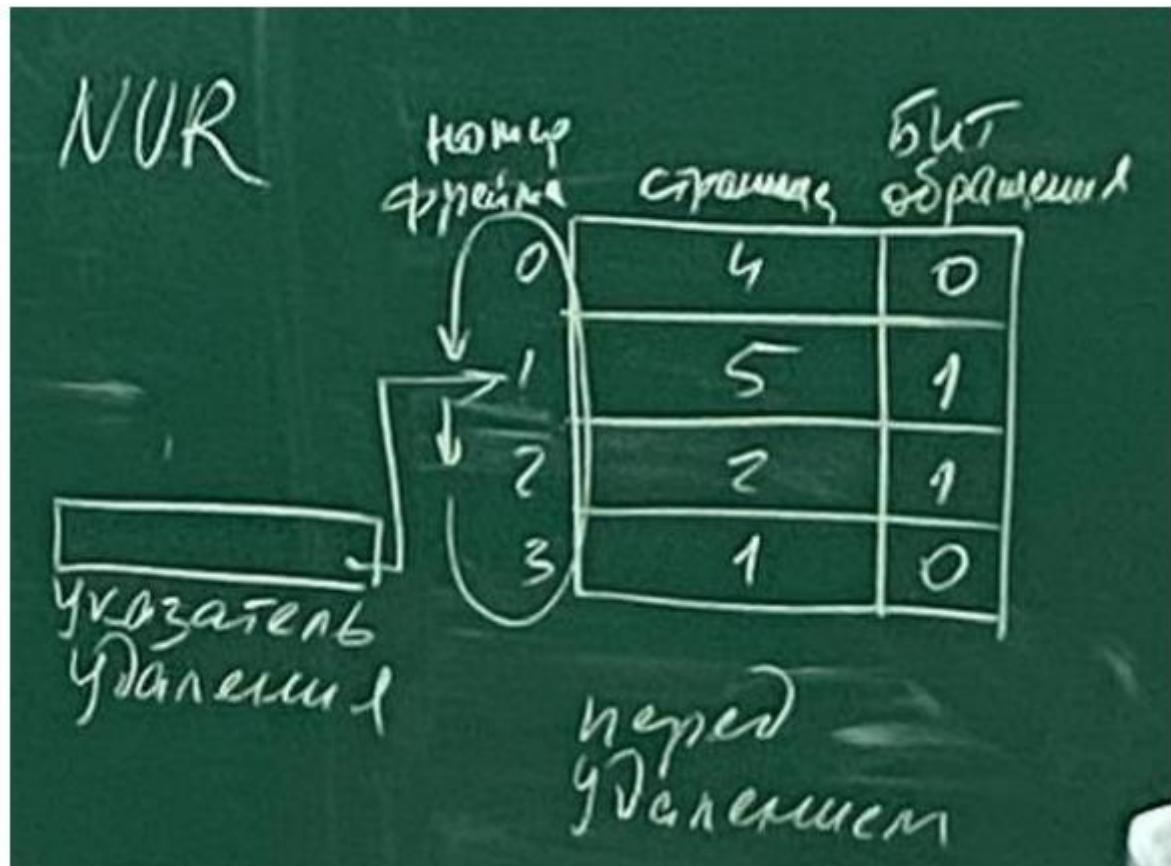
*При этом обращение к странице – на каждой команде, а то и несколько раз (именно так говорят) (обращение к команде, обращение к данным команды, а если косвенная адресация – там 2 обращения по поводу данных).*

4. LFU – least frequency used – наименее часто используемая страница
  - Здесь используется счетчик обращений к странице. Он инкрементируется при каждом обращении к странице.
  - очевидный недостаток – может быть вытеснена только что загруженная страница, так как она не набрала ещё количества обращений.

5. NUR - Not used recently page replacement – страница, не используемая в последнее время.

- Это – аппроксимация алгоритма LRU.
- вводятся 2 бита – обращения и модификации, а также указатель удаления
- бит обращения периодически сбрасывается в 0 (для всех страниц), а при обращении к странице – выставляется 1. Поэтому, если надо вытеснить какую-либо страницу, то она ищется среди тех, у которых этот бит сброшен. И это показывает, что с момента последнего сброса битов обращения, обращения к этой странице не было.
- Пример

До удаления:



после удаления

номер	страница	бит обращения	бит модификации
0	1	1	0
1	5	1	0
2	2	0	0
3	доступна	0	0

После удаления

На первом рисунке – ситуация после загрузки 5 страницы в 1 кадр (frame). Если в этот момент необходимо удалить страницу, то проверка значений битов обращения будет выполняться со 2 кадра (фрейма), поэтому и показана цикличность. Бит обращения у 2 фрейма = 1 -> ее нельзя удалить. Идем дальше, у 3 фрейма бит обращения=0, поэтому первая страница, загруженная в 3 фрейм может быть удалена, что здесь и показано.

- бит модификации ( dirty (грязный)). Делается, чтобы: какую страницу выгодно заместить – которая не модифицировалась, потому что тогда не нужно будет копирование, так как точная копия этой страницы находится во вторичной памяти.
- Значит, вводятся 2 бита – обращения и модификации. Тогда возможны 4 ситуации

бит обращения	бит модификации
0	0
1	1

Вызывает вопрос вторая строка. Как так – обращения не было, а модификация была? Значит, эта страницы модифицировалось до сброса всех битов обращения в 0.

### **Глобальное и локальное замещение страниц.**

Под глобальным замещением понимают выгрузку любой страницы любого процесса, чтобы процесс мог загрузить свою очередную страницу.

Под локальным - выгрузку только страниц данного процесса. Есть у него квота, возникло страничное прерывание – должна быть выгружена страница этого процесса.

В unix, linux есть page demon – демон страниц, в windows – swapping.

На самом деле выгружаются страницы заранее, чтобы иметь набор свободных страничных кадров. Тогда подгрузка будет быстрее и каждый конкретный процесс эффективней.

**7.2 Управление памятью сегментами по запросам в архитектуре X86. Тип организации таблиц сегментов (2021 в защищенном режиме). Формат дескриптора сегмента в таблицах дескрипторов сегментов (GDT и LDT) (код и заполнение дескрипторов GDT из лабораторной работы по защищенному режиму).**

В целом про организацию сегментами по запросам описано в вопросе 1, повторять тут не надо наверное.

### **Тип организации таблиц сегментов**

3 подхода к организации таблиц дескрипторов сегментов: Единая таблица, Локальные таблицы, Локальные таблицы и одна глобальная таблица.

В интел используется последняя:

- Локальная таблица описывает адресное пространство отдельного процесса (LDT)
- дескрипторы ЛТ содержат ссылку на дескриптор сегмента в глобальной таблице дескрипторов. (GDT)
- глобальная таблица содержит уже адреса сегментов в физической памяти

### **Управление памятью сегментами по запросам в архитектуре X86**

При работе в защищенном режиме микропроцессора адресное пространство делится на глобальное – общее для всех задач; локальное – отдельное для каждой задачи. Для того, чтобы использовать память, нам ее нужно сначала выделить и описать.

В системе есть специальные таблицы для управления памятью.

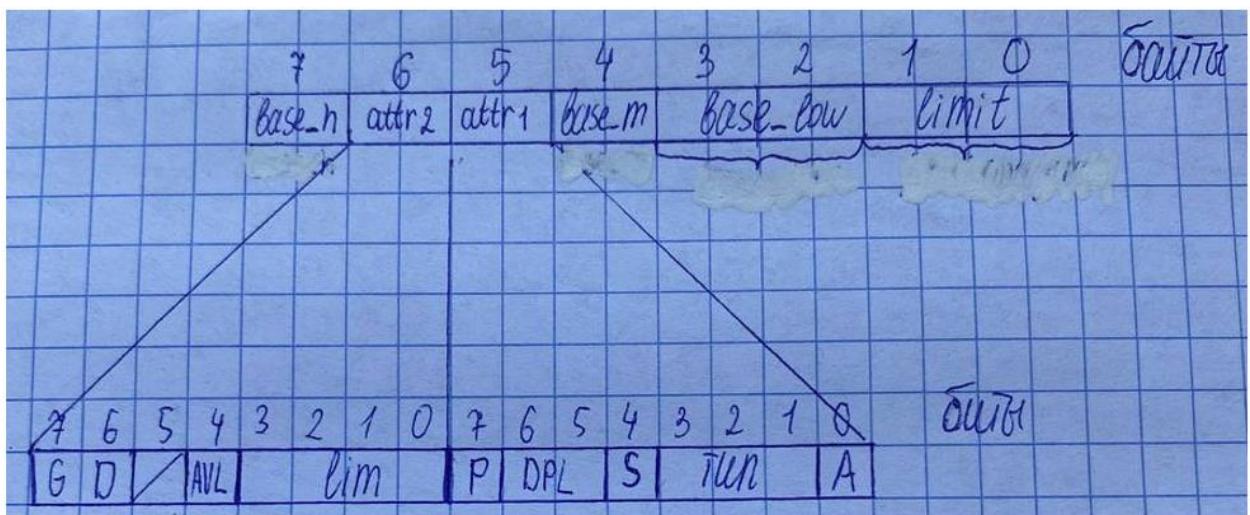
- GDT - глобальная таблица дескрипторов
- IDT - таблица дескрипторов прерываний
- LDT - локальная таблица дескрипторов

И Регистры системных адресов (регистры управления памятью) - чтобы поддерживать эти таблицы.

1. GDTR(32, в РФ-байт) (Global Descriptor Table Register) - регистр таблицы глобальных дескрипторов. Содержит линейный физический адрес начала таблицы дескрипторов – адрес байта начала таблицы глобальных дескрипторов (GDT)
2. IDTR(32) (Interrupt Descriptor Table Register) – регистр таблицы дескрипторов прерываний. Содержит линейный физический адрес начала таблицы дескрипторов прерываний. – адрес байта начала таблицы глобальных дескрипторов (IDT)
3. LDTR(16) (Local Descriptor Table Register) - регистр локальной таблицы дескрипторов дескрипторов. 16 бит -> не может содержать линейный физический адрес. содержащего так называемый селектор дескриптора локальной дескрипторной таблицы LDT. Этот селектор является указателем в таблице GDT, который и описывает сегмент, содержащий локальную дескрипторную таблицу LDT;
4. TR (16) (Task Register), который подобно регистру ldtr, содержит селектор, т. е. указатель на дескриптор в таблице GDT. Для переключения задач.

Напрямую эти регистры недоступны - есть специальные команды чтобы загружать или выгружать их, эти команды привилегированные. (Lgdt например)

#### Формат дескриптора сегмента в таблицах дескрипторов сегментов (GDT и LDT)



В РР сегменты определяются базовыми адресами, задаваемыми в явной форме, В ЗР - дескриптором (8-байтовым полем)

- Байты 2-3 (base\_low), 4 (base\_middle), 7 (base\_high): база сегмента - начальный линейный адрес сегмента в адресном пространстве процессора. (имеет длину 32 бита, номер байта, может располагаться в любом месте адресного пространства 4Гбайт). Если страничная адресация выключена, он совпадает с физическим (как во 2 ЛР), включена - могут и не совпадать.

База - адрес, с которого начинается данный сегмент. Повторюсь: адрес в виртуальном адресном пространстве. Вообще, все упоминаемые здесь и далее

адреса упоминаются в контексте виртуальности; к физическим адресам мы доступа не имеем

- Байты 0-1 (limit): младшие 16 бит границы сегмента -номер последнего байта сегмента).
  - Байт 6 (attr\_2)
    - 0-3 (lim): оставшиеся старшие 4 бита границы сегмента (итого 20 бит). *Поскольку у регистров доступны младшие части, это показывает, что старшие компьютеры поддерживают реальный режим аппаратно - основанная идея Intel - обратная совместимость. Возникает вопрос - сколько разрядов в шине адреса в реальном режиме? 20. Это максимально возможный объем который мы можем адресовать в реальном режиме - 1 МБ памяти. 0,1 - limit (только 16 разрядов), а шина 20-разрядная - нам не хватает 4 разрядов. Таким образом мы имеем базовый линейный адрес - можем адресовать начало сегмента.*
    - 6 бит (D default): разрядность operandов и адресов по умолчанию (0-16, 1-32). Можно изменить на противоположный префиксом замены размера 66h(операнда) и 67h (адреса). D=0 не запрещает использовать 32 регистры: компилятор сам добавит префикс
    - 7 бит G (бит дробности (гранулярности)): единицы, в которых задается граница. 0-в байтах (и тогда сегмент <=1 Мбайт), 1-в блоках по 4 Кбайт (страницах) (до 4 Гбайт)
- гр. сег.=гр.в.дескр.\*4К+4095 - до конца последнего 4-Кбайтного блока).
- 5 L - флаг, который ранее был зарезервирован, теперь служит признаком 64-разрядности сегмента. Если он установлен, флаг D/B должен быть сброшен.
  - 4 AVL - неиспользуемый бит. Может использоваться по усмотрению ОС.

- Байт 5: attr\_1:
  - 0 бит (A accessed): устанавливается процессором, когда в какой-либо сегментный регистр загружается селектор данного сегмента (было обращение)
  - 1-3 биты: тип сегмента.
    - 3 бит – бит предназначения: 0 – сегмент данных/стека, 1-кода
    - 2 бит:
      - Для кода [бит подчинения: 0 – код подчинен (связан с каким – то другим сегментом), 1 – обычный]. Подчиненные, или согласованные сегменты обычно используются для хранения подпрограмм общего пользования; для них не действуют общие правила защиты программ друг от друга.
      - Для стека и данных [0-данные, 1-стек]
    - 1 бит:

- Для кода [0–чтение из сегмента запрещено (не относится к выборке команд) - считывание из памяти и загрузка в регистры процессора, mov, l – разрешено]
- Для данных [0- модификации запрещены, 1-модификации разрешены]
- 4 бит (S system): идентификатор сегмента (0-системный сегмент, 1-сегмент памяти).
- 5-6 биты (DPL descriptor privilege level): уровень привилегий этого дескриптора: от 0 (УП ядра системы) до 3 (УП приложений). (как в селекторе RPL request PL (программно), CPL current PL (аппаратно))
- 7 (P present): бит присутствия, представлен ли сегмент в памяти (выгружен ли из внешней в оперативную).

Тип	Характеристики сегмента
0	Разрешено только чтение (сегмент данных)
1	Разрешены чтение и запись (сегмент данных)
2	Расширение вниз, разрешено только чтение (сегмент стека)
3	Расширение вниз, разрешены чтение и запись (сегмент стека)
4	Разрешено только исполнение (сегмент команд)
5	Разрешены исполнение и чтение (сегмент команд)
6	Разрешено только исполнение (подчиненный сегмент)
7	Разрешены исполнение и чтение (подчиненный сегмент)

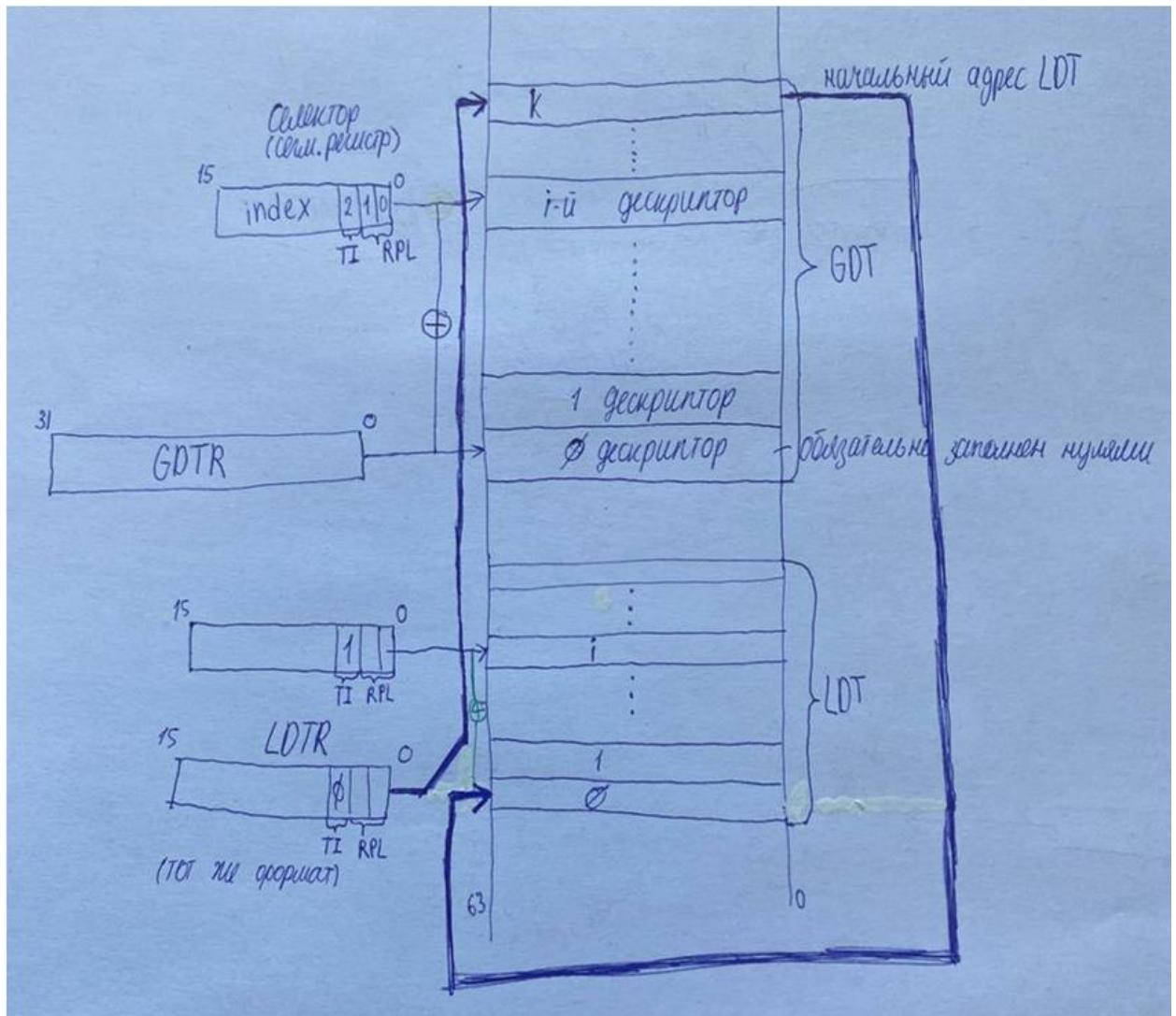
(про GDT и LDT- на всякий случай. Можно пролистнуть до заголовка [код и заполнение дескрипторов GDT из лабораторной работы по защищенному режиму](#))

### Глобальная таблица дескрипторов GDT

В SMP архитектуре (наши компы) равноправные процессоры, которые работают с общей памятью. Один – главный, обрабатывает прерывание от системного таймера, но не руководит другими процессами..

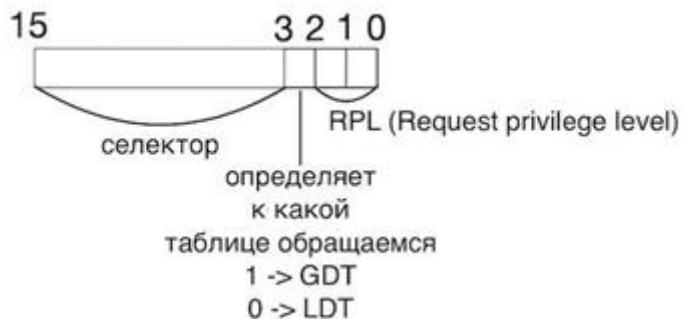
Так как одна память, то и GDT в системе одна. Она находится в памяти ядра системы. На начало таблицы указывает GDTR (32 разрядный).

GDT состоит из 8-байтных записей - дескрипторов. Первая запись всегда должна быть заполнена нулями и не используется. Далее следуют дескрипторы сегментов.



### Формат селектора.

Сегментный регистр – селектор (16) – идентификатор сегмента - внутри себя содержит номер записи в таблице(индекс), по нему получаем запись в ней - то есть дескриптор.



- 0 и 1 бит – RPL (request privilege level) отвечают за уровни привилегий - их всего 4, используются процессором при проверке возможности доступа к сегментам.

- 2 бит TI table indicator - определяет к какой таблице идет обращение (1- к локальной или 0 - к глобальной)
- 3-15 - индекс. Так как размер дескриптора 8 байт, то минимальный селектор должен тоже быть 8 - мы через первые три бита домножаем селектор на 8(добавляем три разряда в двоичной). 01 000 -> 8 (первый селектор), 10 000 -> 16 (второй селектор)

### **Локальные таблицы дескрипторов.**

LDT описывает виртуальное адресное пространство процессов-> их столько, сколько процессов выполняется системой. В памяти занимают сегмент (64 кб). Сегменты описывает GDT->LDTR-селектор к дескриптору сегмента, в котором находится таблица локальных дескрипторов LDT.

Используются процессами - они могут там хранить свои сегменты и обращаться к ним вместо того чтобы хранить это в глобальной таблице.

Формат дескрипторов абсолютно такой же.

*Регистр LDTR может загружаться при переключении между задачами и у каждой задачи может быть своя локальная таблица дескрипторов.*

*P6 есть, но нет флага, отключающего сегментное преобразование. ЭТО БАЗА. В защищенном режиме используется модель памяти flat.*

*Сегменты 16-разрядные, а ЗР 32-разрядный, следовательно, не выходят за 1 МБ. В limit для всех дескрипторов записано ffff=2^16=64Кбайт – разрядность регистров в PP, следовательно, смещение в PP не может превышать 2^16. FFFF=2^20=1Мбайт (мое с семинара, но что-то странное)*

**код и заполнение дескрипторов GDT из лабораторной работы по защищенному режиму**

```
;Структура descr для описания дескрипторов сегментов
descr struc
    limit    dw 0
    base_1   dw 0
    base_m   db 0
    attr_1   db 0
    attr_2   db 0
    base_h   db 0
descr ends
```

Начальные значение для дескрипторов GDT

```

; Таблица глобальных дескрипторов

;Дескриптор: <limit, base_l, base_m, attr_1, attr_2, base_h>
;Базы сегментов будут вычислены программно и занесены в соответствующие
;дескрипторы на этапе выполнения. Фактические значения размеров сегментов
;будут вычислены транслятором (если не указано конкретное значение в описании GDT)

;в данной программе для сегмента команд attr_1=98h=10011000b:
;    сегмент присутствует в памяти, имеет УП ядра, является сегментом памяти,
;    подчиненный сегмент кода, разрешено только исполнение

;для сегмента данных (или стека) attr_1=92h=10010010b:
;    сегмент присутствует в памяти, УП ядра, является сегментом памяти,
;    сегмент данных, разрешены чтение и запись

; Нулевой дескриптор
gdt_null descr <>

;сегмент кода, разрешено только исполнение
;G=0 (граница в байтах), D=0 (разрядность оп. и adr. по ум.16)
; (для реального режима)
gdt_code16 descr <code16_size-1,0,0,98h>

;размер: 4 Гб
;сегмент данных, разрешены чтение и запись
;G=1 (граница в блоках по 4 Кбайт), D=1 (разрядность оп. и adr. по ум.32)
; (для определения объема выделенной памяти)
gdt_data4gb descr <0FFFFh,0,0,92h,0CFh>

;сегмент кода, разрешено только исполнение
;G=0 (граница в байтах), D=1 (разрядность оп. и adr. по ум.32)
; (для защищенного режима)
gdt_code32 descr <code32_size-1,0,0,98h,40h>

;сегмент данных, разрешены чтение и запись
;G=0 (граница в байтах), D=1 (разрядность оп. и adr. по ум.32)
; (?зачем)
gdt_data32 descr <data_size-1,0,0,92h,40h>

;сегмент данных (стек), разрешены чтение и запись
;G=0 (граница в байтах), D=1 (разрядность оп. и adr. по ум.32)
; (для прерываний в ЗР)
gdt_stack32 descr <stack_size-1,0,0,92h,40h>

;видеобуфер
;размер страницы=4096 байт, базовый физический адрес=B8000h
;G=0 (граница в байтах), D=0 (разрядность оп. и adr. по ум.16)
gdt_video16 descr <4095,8000h,0Bh,92h>

gdt_size=$-gdt_null ;размер
pdescr     df 0        ;псевдодескриптор

;Селекторы
code16s=8
data4gbs=16
code32s=24
data32s=32
stack32s=40
video16s=48

```

Заполнение дескрипторов GDT (на примере дескриптора code16. Для остальных линейный адрес начала заполняется аналогично)

```

;Завершение формирования дескрипторов сегментов программы заполнением
;базовых адресов сегментов. Линейные 32-битовые адреса определяются путем
;умножения значений сегментных адресов на 16 (=побитовый сдвиг влево на 4)
;SHL и ROL - логический и циклический побитовые сдвиги op1 влево на op2 бит, соответственно
;(rol на 16 фактически обменивает местами старшую и младшую половины)
xor eax, eax

mov ax, code16
shl eax, 4
mov word ptr gdt_code16.base_l, ax
rol eax, 16
mov byte ptr gdt_code16.base_m, al

```

## 8 билет

8.1 Взаимоисключение и синхронизация процессов и потоков. Семафоры:  
определение, виды. Семафор, как средство синхронизации и передачи сообщений.  
Семафоры UNIX: примеры решения задач с помощью семафоров: «Производство-потребление» и «Читатели-писатели» в UNIX (пример реализации в лабораторной работе).

(Текст до заголовка [Монопольный доступ и взаимоисключение](#) можно писать выборочно - если будет время, лучше написать)

### **Взаимоисключение и синхронизация процессов и потоков.**

Взаимодействие параллельных процессов, выполняемых реально параллельно, когда они одновременно выполняются на разных процессорах или выполняемых квази параллельно, когда они поочередно выполняются на одном, включает в себя: взаимодействие, т.е. обработку одних и тех же данных или конкуренцию за владение одними и теми же ресурсами.

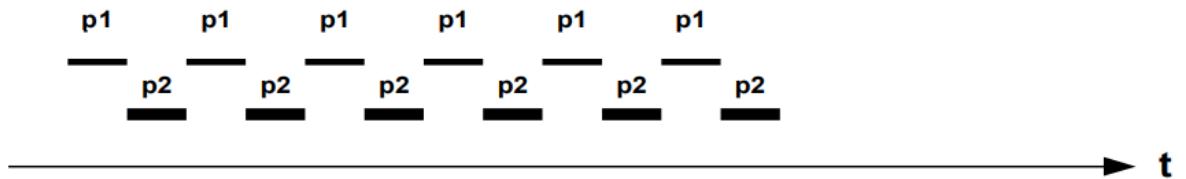
Важно отметить, что процессы являются асинхронными, т.е. они выполняются с собственной скоростью, что означает невозможность предсказания в какой момент процесс дойдет до определенной точки выполнения.

#### **Виды параллелизма:**

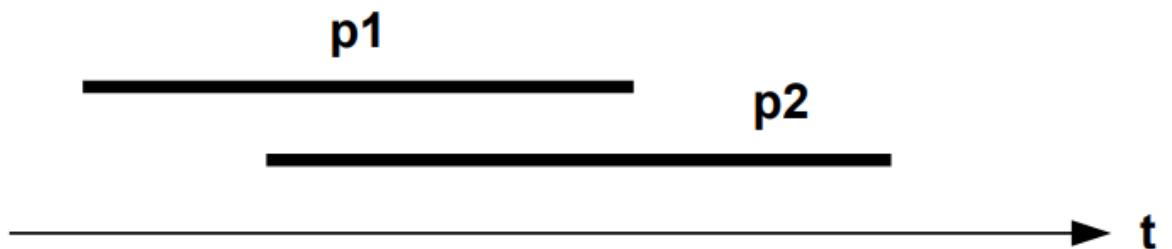
- Последовательное выполнение процессов: Сначала выполняется процесс p1 команда за командой, затем p2



- Квази параллельное выполнение процессов (один процессор). Определенный интервал времени выполняется процесс p1, затем выделенный интервал времени выполняется процесс p2 на одном и том же процессоре



3. Реальная параллельность или перекрытие (несколько процессоров): команды процессов p1 и p2 выполняются одновременно на разных процессорах



Чередование или одновременное выполнение (перекрытие) только на первый взгляд отличаются друг от друга. Потенциально перекрытие может повысить скорость выполнения процессов, но только потенциально. В обоих случаях процессы выполняются независимо друг от друга с собственными скоростями (асинхронно), что не позволяет предсказать относительную скорость выполнения асинхронных процессов.

Однако вопросы и проблемы, возникающие в связи с этими двумя видами параллелизма, в значительной степени совпадают:

- безопасное совместное использование разделяемых ресурсов;
- синхронизация выполнения процессов при их взаимодействии;
- обнаружение ошибок программирования может быть значительно затруднено, потому что контексты, в которых возникают ошибки, не всегда могут быть легко воспроизведены.

С точки зрения управления параллельными асинхронными процессами, как в однопроцессорных системах, так и в SMP возникают одни и те же проблемы:

- взаимоисключение процессов при доступе к разделяемым ресурсам;
- синхронизация параллельных асинхронных процессов;
- бесконечное откладывание и взаимоблокировка процессов. Причем третья проблема появляется как следствие первых двух.

### **Примеры взаимодействия параллельных асинхронных процессов**

Пусть есть 2 процесса: P1, P2.

Пусть у них имеются следующие участки кодов (myvar - разделяемая переменная):

P1	P2
mov eax, myvar	mov eax, myvar
inc eax	inc eax
mov myvar, eax	mov myvar, eax

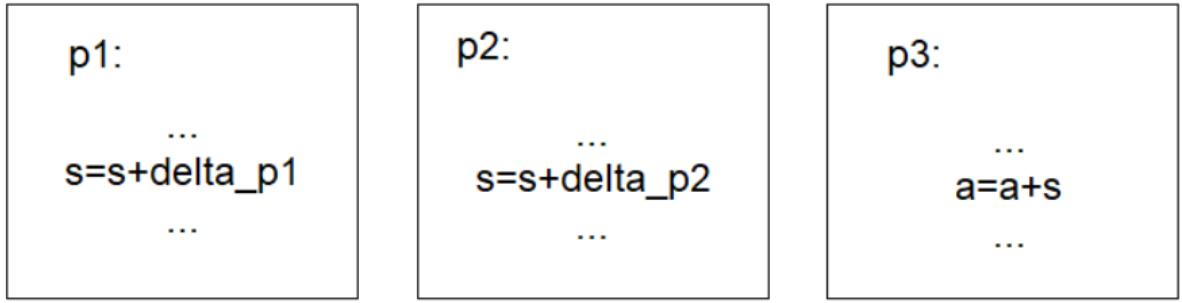
При выполнении этих 2 процессов параллельно (или квазипараллельно) возможен следующий порядок команд:

Команды P1	eax	myvar	eax	Команды P2
		0		
mov eax, myvar	0	0		
	0	0	0	mov eax, myvar
	0	0	1	inc eax
	0	1	1	mov myvar, eax
inc eax	1	1	1	
mov myvar, eax	1	1	1	

(myvar должен был стать 2, но стал 1)

или такой примерчик

Два параллельных процесса p1 и p2 ( потока ) изменяют значение одной и той же глобальной переменной S, например прибавляют к ней соответственно значения delta\_p1 и delta\_p2. Третий процесс должен получить результирующее значение этой переменной, используя ее для своих дальнейших вычислений



Здесь возникает сразу две проблемы.

Первая связана с изменением значения переменной. Допустим, что оба процесса одновременно будут пытаться изменить значение переменной  $S$ . Очевидно, что в этой «гонке» процессор получит только один процесс, допустим, что это процесс  $p_2$ . Операция изменения значения  $S$  не является неделимой. Более того она требует выполнения целого ряда машинных команд. Поэтому вполне вероятно, что процесс обратившийся к переменной  $\delta_{p2}$  и сложив это значение с  $S$  будем принудительно прерван или прерыванием по таймеру, или каким-либо прерыванием ввода-вывода и не успеет занести это значение в  $S$ . Обслуживание прерывания займет некоторое время и выделенный  $p_2$  квант процессорного времени закончится. Затем процессор получит процесс  $p_1$ . Он сможет без препятствий выполнить сложение. Затем квант процессорного времени получает третий процесс. Он использует  $S$  в своих вычислениях. Но величина  $S$  будет ошибочной, так как потеряна составляющая  $\delta_{p1}$ .

Вторая проблема связана с асинхронностью процессов. Третий процесс может обратиться к переменной  $S$  до того, как и процесс  $p_1$  и процесс  $p_2$  получат возможность выполнить сложение.

Аналогичные ситуации возможны в информационно-поисковых системах, например системах резервирования авиа или железнодорожных билетов, существует множество параллельных процессов, осуществляющих разные действия, такие как поиск информации на запрос и отправление заявки на резервирование, собственно резервирование.

### **Монопольный доступ и взаимоисключение**

Переменные, к которым пытаются получить доступ параллельные процессы, называются разделяемыми или совместно используемыми переменными. Процессы должны обрабатывать разделяемые переменные в режиме монопольного доступа. Иначе возникнут описанные выше ситуации, приводящие к потере данных.

Подобные переменные рассматриваются как критические ресурсы (critical resource). Критическим ресурсом называется ресурс, который в каждый момент времени может использоваться только одним процессом. Такое использование называется монопольным.

Часть кода процесса, в которой осуществляется обращение к монопольно используемому ресурсу, называется критической секцией или областью (critical section, critical region).

Монопольный доступ предполагает, что разделяемая переменная обрабатывается процессом монопольно, исключая доступ на это время других процессов к этой же переменной. Другими словами, если процесс находится в своей критической секции по

разделяемой переменной, то это исключает возможность доступа другого процесса к этой же критической секции. Монопольный доступ процессов к разделяемым ресурсам обеспечивается методами взаимоисключения (mutual exclusion).

Методы организации взаимоисключения делятся на

- Программный способ
- Аппаратный способ
- С помощью семафоров (выделяется, потому что решил ряд проблем)
- С использованием мониторов

.....Если захочется расписать (не советую)

### 1. Программные

*Идея от студентов, как работать с разделяемыми ресурсами: сделать флаг цикл ожидания по флагу i-го процесса*

*Если процесс 2 находится в своей критической секции, процесс 1 входит в режим ожидания по флагу 2. Когда флаг2 сброшен, процесс 1 может установить свой флаг, войти в свою критическую секцию CR1, закончить и сбросить свой флаг, перейти к другим действиям.*

#### Программный способ взаимоисключения

Пример реализации 1:

```
Var flag1, flag2: Boolean;  
  
p1: while(1)           |   p2: while(1)  
{                      |   {  
    while(flag2==1);    |       while(flag1==1);  
    flag1 = 1;          |       flag2 = 1;  
    CR1;                |       CR2;  
    flag1=0;             |       flag2 = 0;  
    PR1;                |       PR2;  
}  
...  
....  
// начальные установки  
flag1=0;flag2=0;  
  
parbegin  
  p1;p2;  
parend;
```

Пусть инициативу перехватил Р2. Ему надо попасть в свой критический участок, проверяет, флаг не установлен, теряет инициативу. Процесс 1 – также, + устанавливает свой флаг, входит в критическую секцию, изменяет переменную, сбрасывает флаг и идет дальше. Опять процесс 2, восстановил АК, продолжает со следующей команды, устанавливает, КС.

=> Предложение: установить влаг до while

Программный способ взаимоисключения

Пример реализации 2:

```
Var flag1, flag2: Boolean;
p1: while(1)           |   p2: while(1)
{
    |   {
        flag1 = 1;          |   flag2 = 1;
        while(flag2==1);   |   while(flag1==1);
        CR1;                |   CR2;
        flag1=0;             |   flag2 = 0;
        PR1;                |   PR2;
    }                   |   }
....                  ....
// начальные установки
flag1=0;flag2=0;
parbegin
p1;p2;
parend;
```

Та же последовательность. Устанавливает флаг и теряет квант. Получает первый, устанавливает и застrevает в цикле ожидания по флагу второго процесса. Все застряло. Процессы попали в deadlock – туниковая ситуация.

Еще можно написать про алгоритм Деккера (но это ТАК много)

Данное решение исключает «зависание», бесконечное откладывание и взаимоблокировку за счет введения дополнительной переменной – очередь (queue), определяющей очередность входления процессов в критический участок

```

Program example4; // алгоритм Деккера
  flag1, flag2: logical;
  queue: 1..2; //чья очередь
P1: //первый процесс
  While (true) do
    begin
      flag1 = true;
      While (flag2) do
        if (queue == 2) then
          begin
            flag1 = false;
            While (queue == 2) do; //цикл ожидания
            flag1 = true;
            end;
        //критический участок первого процесса
        queue = 2;
        flag1 = false;
        //другие операторы процесса
        end;
    end; //p1
P2: //второй процесс
  While (true) do
    begin
      flag2 = true;
      While (flag1) do
        if (queue == 1) then
          begin
            flag2 = false;
            While (queue == 1) do; //цикл ожидания
            flag2 = true;
            end;
        //критический участок второго процесса
        queue = 1;
        flag2 = false;
        //другие операторы процесса
        end;
    end; //p2
  begin

```

---

flag1 = 0; flag2 = 0;  
**parbegin**  
 P1; P2;  
**parend**;  
**end.**

Во всех рассмотренных случаях имеются циклы, в которых проверяется значение управляющих переменных. Для проверки занимаются кванты процессорного времени, что является негативным фактором, который получил специальное название – **активное ожидание (busy wait)** на процессоре.

## 2. Аппаратный

Наиболее известным аппаратным средством является неделимая команда – *test-and-set* (проверить и установить) [8]. Для реализации взаимоисключения с разделяемым ресурсом связывается так называемый байт блокировки.

Команда *test-and-set* (появилась в IBM370) Неделимая команда, которая выполняет проверку и установку содержимого ячейки памяти, которая часто называется байтом блокировки. 0 - ресурс доступен, 1-занят.

---

```

Program example-test-and-set;
active; logical;
P1: //первый процесс
    flag1: logical; //локальная переменная
    While (true) do
        begin
            flag1 = true;
            While (nfalg1) do
                TS(flag1, active); //цикл проверки переменной active
                // критическая секция первого процесса
                active = 0;
                // другие операторы первого процесса

        end;
    end; //P1
P2: //второй процесс
    flag2 : logical; //локальная переменная
    While (true) do
        begin
            flag2 = true;
            While (falg2) do
                TS(flag2, active); //цикл проверки переменной active
                // критическая секция второго процесса
                active = 0;
                // другие операторы второго процесса
        end;
    end; //P2
    begin
        active = false;
        parbegin
            P1;P2;
        parend;
    end.

```

---

*Переменная active имеет значение «истина», если какой-то процесс находится в своем критическом участке.*

### 3. С помощью семафоров

*Семафор – это неотрицательная, защищенная переменная, на которой определены две неделимые операции  $P(S)$  и  $V(S)$  :*

- *операция  $P(S)$  выполняет уменьшение значения семафора на 1, т.е.  $S = S - 1$ , если  $S > 0$ ;*
- *уменьшение невозможно, если  $S = 0$ , и процесс блокируется в очереди на  $S$  до тех пор, пока уменьшение станет возможным;*
- *операция  $V(S)$  выполняет увеличение значения семафора, т.е.  $S = S + 1$ , и тем самым разблокирует процесс, стоящий первым в очереди к данному семафору в ожидании его освобождения.*

*Основными проблемами при использовании семафоров являются «гонки» (race conditions) и взаимоблокировки.*

#### *4. С помощью мониторов*

..... Конец

Синхронизация процессов - это механизм, позволяющий обеспечить целостность какого-либо ресурса, когда он используется несколькими асинхронными процессами или потоками. Для синхронизации процессов и потоков используются семафоры, мьютексы и пр.

Организации взаимоисключения должна выполняться с учетом следующих факторов:

- два или более процессов не могут одновременно находиться в своих критических участках по одним и тем же разделяемым ресурсам, при этом другие участки процессов должны выполняться параллельно;
- при решении проблемы не делается никаких предположений о скорости асинхронных параллельных процессов;
- при аварийном завершении процесса, находящегося в критическом участке кода, операционная система должна отменить выполненные процессом действия, чтобы другие процессы получили возможность входить в свои критические участки;
- процесс, желающий войти в свой критический участок, не должен находиться в состоянии ожидания неопределенное время, т.е. необходимо устранить ситуацию бесконечного откладывания.

#### **Семафоры: определение**

Эдже́р Дейкст́ра 1965 год: семафор как средство реализации взаимоисключения.

Семафор – это неотрицательная, защищенная переменная, на которой определены две неделимые (Одной неделимой операцией осуществляется выборка переменной s, инкремент и ее запоминание.) операции P(S) и V(S) :

- операция P(S) выполняет уменьшение значения семафора на 1, т.е.  $S = S - 1$ , если  $S > 0$ ; уменьшение невозможно, если  $S = 0$ , и процесс блокируется в очереди на S до тех пор, пока уменьшение станет возможным;
- операция V(S) выполняет увеличение значения семафора, т.е.  $S = S + 1$ , и тем самым разблокирует процесс, стоящий первым в очереди к данному семафору в ожидании его освобождения. если  $\square = 0$ , то операция V(s) может активизировать некоторый процесс блокированный на семафоре

Семафоры исключают бесконечное ожидание на процессоре, но платой за это является переход в режим ядра, т.е. команды определенные на семафоре являются системными вызовами.

Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо привилегированный процесс. Осуществляется переход в режим ядра при захвате и освобождении семафора, т.е. происходит переключение контекста.

Обычно семафоры реализуются аппаратно и являются объектами ядра системы, но являются объектами высокого уровня, основанными на командах более низкого уровня таких, как test-and-set.

При этом сами семафоры являются разделяемыми ресурсами. Семафор, как разделяемая переменная, может находиться только в области данных ядра ОС, так как адресные пространства процессов являются защищенными, т.е. к ним закрыт доступ других процессов.

### Виды:

- бинарный или двоичный- принимает только два значения
- считающий - принимающий значения от 0 до n. Используются в тех случаях, когда имеется n единиц ресурса (пример - производство-потребление)
- Множественные семафоры - массив считающих семафоров. Одной неделимой операцией можно изменить все или часть семафоров набора.

### Семафор, как средство синхронизации и передачи сообщений.

Способы взаимодействий:

- Взаимоисключения, организация монопольного доступа процесса к разделяемой переменной (задача "Читатели-писатели")
- Синхронизация, когда процесс заинтересован в действиях другого процесса (задача "Производство-потребление")

### Семафоры UNIX:

Про то, что вообще с ними можно делать и как все устроено

**«Производство-потребление»**

пример использования из лабораторной работы «производство-потребление».

**«Читатели-писатели»**

Версия 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <wait.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/stat.h>
```

```

#define N_ITERS 10

#define N_READERS 5
#define N_WRITERS 3

#define ACTIVE_READERS 0
#define ACTIVE_WRITERS 1 //CAN_WRITE
#define WAITING_READERS 2 //CAN_READ
#define WAITING_WRITERS 3

#define MIN_SLEEP 1
#define MAX_SLEEP 3

struct sembuf SEM_START_READ[] =
{
    {ACTIVE_WRITERS, 0, 0},
    {WAITING_READERS, 1, 0},
    {WAITING_WRITERS, 0, 0},
    {ACTIVE_READERS, 1, 0},
    {WAITING_READERS, -1, 0},
};

struct sembuf SEM_STOP_READ[] =
{
    {ACTIVE_READERS, -1, 0},
};

struct sembuf SEM_START_WRITE[] =
{
    {WAITING_WRITERS, 1, 0},
    {ACTIVE_READERS, 0, 0},
    {ACTIVE_WRITERS, 0, 0},
    {ACTIVE_WRITERS, 1, 0},
    {WAITING_WRITERS, -1, 0},
};

struct sembuf SEM_STOP_WRITE[] =
{
    {ACTIVE_WRITERS, -1, 0},
};

int start_read(int s_id)
{
    return semop(s_id, SEM_START_READ, 5) != -1;
}

int stop_read(int s_id)
{
    return semop(s_id, SEM_STOP_READ, 1) != -1;
}

int start_write(int s_id)
{
    return semop(s_id, SEM_START_WRITE, 5) != -1;
}

int stop_write(int s_id)
{

```

```

        return semop(s_id, SEM_STOP_WRITE, 1) != -1;
    }

int reader_run(int *const shared_mem, const int s_id, const int reader_id)
{
    if (!shared_mem)
    {
        return 1;
    }

    srand(time(NULL) + reader_id);

    int sleep_time;

    for (size_t i = 0; i < N_ITERS; i++)
    {
        sleep_time = rand() % MAX_SLEEP + MIN_SLEEP;
        sleep(sleep_time);

        if (!start_read(s_id))
        {
            perror("Start reading error.");
            exit(1);
        }

        int val = *shared_mem;
        printf("Reader %d read: %2d , sleep time=%d\n", reader_id, val,
sleep_time);

        if (!stop_read(s_id))
        {
            perror("End reading error.");
            exit(1);
        }
    }

    return 0;
}

int writer_run(int *const shared_mem, const int s_id, const int writer_id)
{
    if (!shared_mem)
    {
        return 1;
    }

    srand(time(NULL) + writer_id + N_READERS);

    int sleep_time;

    for (size_t i = 0; i < N_ITERS; i++)
    {
        sleep_time = rand() % MAX_SLEEP + MIN_SLEEP;
        sleep(sleep_time);

        if (!start_write(s_id))
        {
            perror("Start writing error.");
        }
    }
}

```

```

        exit(1);
    }

    int val = ++(*shared_mem);
    printf("Writer %d wrote: %2d , sleep time=%d |\n", writer_id, val,
sleep_time);

    if (!stop_write(s_id))
    {
        perror("End writing error.");
        exit(1);
    }
}

return 0;
}

int main()
{
    setbuf(stdout, NULL);

    int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

    int fd = shmget(IPC_PRIVATE, sizeof(int), perms | IPC_CREAT);
    if (fd == -1)
    {
        perror("shmget failed");
        return 1;
    }

    int *shared_mem = shmat(fd, 0, 0);
    if (shared_mem == (void *)-1)
    {
        perror("shmat failed");
        return 1;
    }

    int s_id = semget(IPC_PRIVATE, 4, perms | IPC_CREAT);
    if (s_id == -1)
    {
        perror("semget failed");
        return 1;
    }

    if (semctl(s_id, ACTIVE_READERS, SETVAL, 0) == -1 ||
        semctl(s_id, ACTIVE_WRITERS, SETVAL, 0) == -1 ||
        semctl(s_id, WAITING_WRITERS, SETVAL, 0) == -1 ||
        semctl(s_id, WAITING_READERS, SETVAL, 0) == -1
        )
    {
        perror("sem initialization error");
        return 1;
    }

    for (size_t i = 0; i < N_READERS; i++)
    {
        int child_pid = fork();
        if (child_pid == -1)

```

```

    {
        perror("Error: fork for reader");
        return 1;
    }
    else if (child_pid == 0)
    {
        reader_run(shared_mem, s_id, i);
        return 0;
    }
}

for (size_t i = 0; i < N_WRITERS; ++i)
{
    int child_pid = fork();
    if (child_pid == -1)
    {
        perror("Error: fork for writer");
        return 1;
    }
    else if (child_pid == 0)
    {
        writer_run(shared_mem, s_id, i);
        return 0;
    }
}

for (size_t i = 0; i < N_WRITERS + N_READERS; i++)
{
    int ch_status;
    int child_pid = wait(&ch_status);

    if (child_pid == -1)
    {
        perror("wait error");
        return 1;
    }

    if (!WIFEXITED(ch_status))
    {
        fprintf(stderr, "Child process %d terminated abnormally", child_pid);
    }
}

if (shmctl(fd, IPC_RMID, NULL) == -1)
{
    perror("shmctl with command IPC_RMID failed");
    return 1;
}

if (semctl(s_id, 0, IPC_RMID) == -1)
{
    perror("semctl with command IPC_RMID failed");
    return 1;
}

```

```

    }

        return 0;
}

```

Версия 2 (на всякий, но первая вроде как правильная)

```

...
#define ACTIVE_READERS 0
#define CAN_WRITE 1
#define CAN_READ 2
#define WAITING_WRITERS 3

struct sembuf SEM_START_READ[] =
{
    {WAITING_WRITERS, 0, 0},
    {CAN_READ, 0, 0},
    {ACTIVE_READERS, 1, 0},
};

struct sembuf SEM_STOP_READ[] =
{
    {ACTIVE_READERS, -1, 0},
};

struct sembuf SEM_START_WRITE[] =
{
    {WAITING_WRITERS, 1, 0},
    {ACTIVE_READERS, 0, 0},
    {CAN_WRITE, 0, 0},
    {CAN_WRITE, 1, 0},
    {CAN_READ, 1, 0},
    {WAITING_WRITERS, -1, 0},
};

struct sembuf SEM_STOP_WRITE[] =
{
    {CAN_WRITE, -1, 0},
    {CAN_READ, -1, 0},
};

...

```

[8.2 Аппаратные прерывания: задачи обработчика прерываний от системного таймера в защищенном режиме.](#)

### Аппаратные прерывания:

адресация (на всякий)

[Адресация аппаратных прерываний в з-р.](#)

## **(по мотивам 1 лабы) задачи обработчика прерываний от системного таймера**

*Кроме часов реального времени, любой компьютер содержит устройство, называемое системным таймером. Это устройство подключено к линии запроса на прерывание IRQ0 и вырабатывает прерывание INT 8h приблизительно 18,2 раза в секунду (точное значение - 1193180/65536 раз в секунду – то есть чуть больше). При инициализации BIOS устанавливает свой обработчик для прерывания таймера.*

У обработчика восьмого прерывания всего 3 функции:

1. Инкремент счетчика реального времени

обработчик каждый раз увеличивает на единицу текущее значение 4-байтовой переменной, располагающейся в области данных BIOS по адресу 0000:046Ch - счетчик таймера. Если этот счетчик переполнится из-за того, что прошло более 24 часов с момента запуска таймера, в ячейку 0000:0470h заносится значение 1

*Есть так называемая CMOS микросхема, которая является энергонезависимой и питается от аккумуляторной батарейки – таблетки. Она продолжает работать после выключения компьютера (для времени, например).*

- Декремент счетчика времени до отключения моторчика дисковода (отложенное действие). На самом деле это делает программа, в порт посыпается команда

Другое действие, выполняемое стандартным обработчиком прерывания таймера - контроль за работой двигателей НГМД. Если после последнего обращения к НГМД прошло более 2 секунд, обработчик прерывания выключает двигатель. Ячейка с адресом 0000:0440h содержит время, оставшееся до выключения двигателя. Это время постоянно уменьшается обработчиком прерывания таймера. Когда оно становится равно 0, двигатель НГМД отключается.

Дисковод - устройство, которое разгоняет дискетку до определенной скорости, после чего можно считывать данные.

Накопитель на гибких магнитных дисках

- Вызов пользовательского прерывания 1Ch

Чтобы программист не переписывал int8h, а вешался на 1Ch, так как если написать много кода, а вызывается часто – фигня.

После инициализации системы вектор INT 1Ch указывает на команду IRET, то есть обработчик прерывания INT 1Ch ничего не делает. Программа может установить собственный обработчик этого прерывания для того чтобы выполнять какие-либо периодические действия.

Необходимо отметить, что прерывание INT 1Ch вызывается обработчиком прерывания INT 8h до сброса контроллера прерывания, поэтому во время выполнения прерывания INT

1Ch все аппаратные прерывания запрещены. В частности, запрещены прерывания от клавиатуры. Обработчик прерывания INT 1Ch должен заканчиваться командой IRET.

Если же вы подготавливаете собственный обработчик для прерывания INT 8h, перед завершением его работы необходимо **сбросить контроллер прерываний**. Это можно сделать, например, так:

```
mov al, 20h
```

```
out 20h, al
```

### **вопросы, которые были к 1 лабе**

<https://github.com/SpectralOne/bmstu-os/tree/master/sem5/lab1>

<https://github.com/chrislvt/OS/wiki/Лабораторная-01.-Обработчик-прерывания-int-8h>

от нас требуют именно про защищенный режим, поэтому я думаю, что стоит расписывать так, как описано выше (то есть рассказывать про первую часть первой лабы). Но на всякий случай приведу вторую часть первой лабы.

### **(по мотивам 2 лабы) Функции обработчика прерывания от системного таймера**

Windows

- По тику:
  - инкремент счётчика системного времени;
  - декремент остатка кванта текущего потока;
  - декремент счётчиков отложенных задач;
  - если активен механизм профилирования ядра, то инициализация отложенного вызова обработчика ловушки профилирования ядра добавлением объекта в очередь DPC (Deferred procedure call, отложенный вызов процедуры). Обработчик ловушки профилирования регистрирует адрес команды, выполнявшейся на момент прерывания.
- По главному тику:
  - инициализация диспетчера настройки баланса путем освобождения объекта «событие», на котором он ожидает.
- По кванту:
  - инициация диспетчеризации потоков добавлением соответствующего объекта в очередь DPC.

Unix

- По тику:
  - инкремент счётчика тиков аппаратного таймера;
  - инкремент часов и других таймеров системы;
  - обновление статистики использования процессора текущим процессом (инкремент поля r\_sri дескриптора текущего процесса до максимального значения, равного 127);

- декремент счетчика времени, оставшегося до отправления на выполнение отложенных вызовов, при достижении нулевого значения счетчика – выставление флага, указывающего на необходимость запуска обработчика отложенного вызова;
- декремент кванта текущего потока.
- По главному тику:
  - регистрация отложенных вызовов функций, относящихся к работе планировщика, таких как пересчет приоритетов;
  - пробуждение (то есть регистрация отложенного вызова процедуры `wakeup`, которая перемещает дескриптор процесса из списка “спящих” в очередь “готовых к выполнению”) системных процессов `swapper` и `pagedaemon`;
  - декремент счётчика времени, оставшегося до посылки одного из следующих сигналов:
    - `SIGALRM` – сигнал, посылаемый процессу по истечении времени, заданного функцией `alarm()` (будильник реального времени);
    - `SIGPROF` – сигнал, посылаемый процессу по истечении времени, заданного в таймере профилирования (будильник профиля процесса);
    - `SIGVTALRM` – сигнал, посылаемый процессу по истечении времени работы в режиме задачи (будильник виртуального времени).
- По кванту:
  - если текущий процесс превысил выделенный ему квант процессорного времени, отправка ему сигнала `SIGXCPU`.

**Регистрация** отложенного вызова функции означает отправку сигнала `SIGCONT`, обработчик которого изменяет состояние процесса с `S` (`Sleep`) на `R` (`Running`)

## 9 билет

9.1 Виртуальная память: управление памятью страницами по запросу – три схемы преобразования виртуального адреса к физическому; реализация страничного преобразования в компьютерах на базе процессоров Intel (x86): стандартное преобразование и PAE в защищенном режиме – схемы, размеры таблиц и их количество на каждом этапе преобразования.

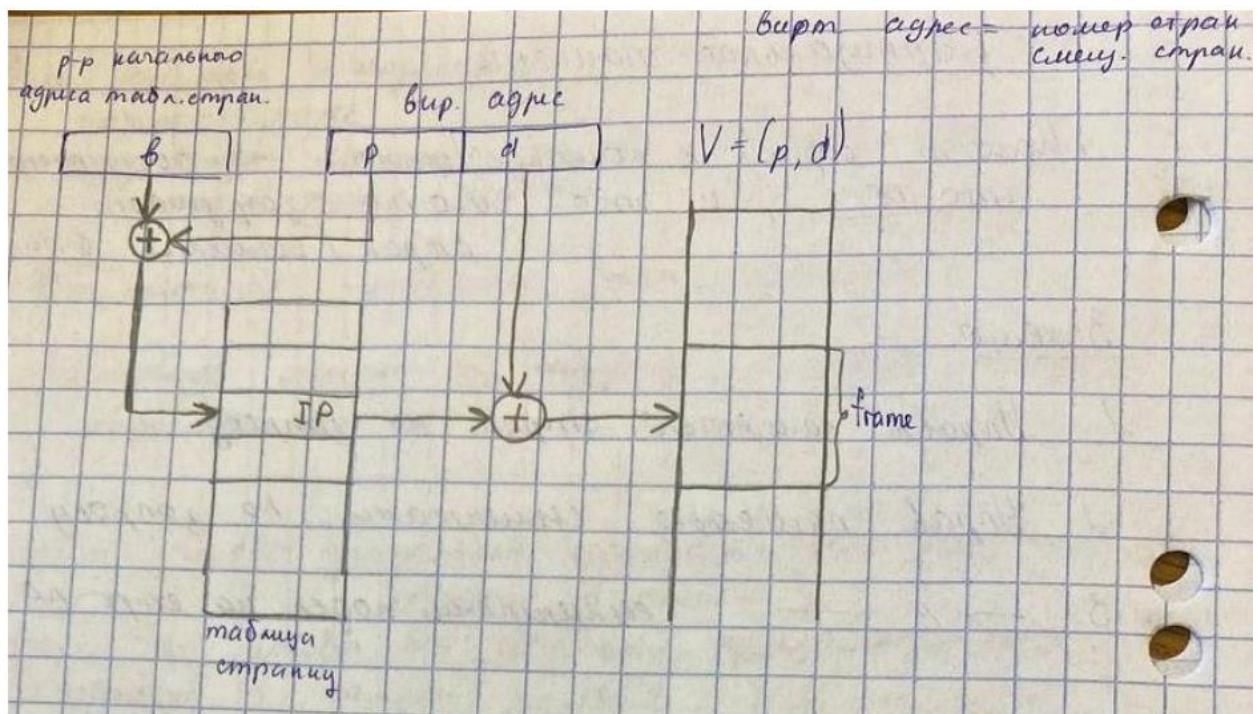
### Виртуальная память

**управление памятью страницами по запросу – три схемы преобразования виртуального адреса к физическому;**

Основная идея – таблица – необходимо иметь соответствующие таблицы, с помощью которых ставятся в соответствие страницы программного кода и физические страницы

3 метода преобразования адресов (это только теоретические основы преобразования, как в intel – посмотрим потом)

1. Прямое отображение



Виртуальный адрес делится на 2 части – номер страницы и смещение.  $V=(p,d)$ . Номер страницы используется как смещение дескриптора страницы в таблице страниц.

Нам нужен (по аналогии с тем, что было ранее) регистр начального адреса таблицы страниц выполняемой программы (процесса). В качестве названия принято `base_address` (b)

При переходе на выполнение другой программы в него будет загружен начальный адрес таблицы страниц другого процесса. Таблица страниц одна (и собственная) на каждый процесс, находится в области памяти ядра системы (ОП), это системная таблица.

Если страница загружена в физическую память, то у нее будет линейный физический адрес, который можно получить, выполнив аппаратно поддерживаемое преобразование. Можно провести аналогию с GDT. Там был бит присутствия  $P$   $present = 1$  если сегмент присутствует в памяти. (В лабе мы его ручками устанавливали). Также есть соответствующие флаги (биты – ударение на 1 и). Они в частности отражают факт присутствия.

Если страница не загружена в память, то возникнет страничное исключение

*Процессов много, таблицы растут с ростом программ и что-то много места начинают занимать + Обращение к физической памяти затратное (цикл обращения к памяти – последовательность импульсов). Поэтому возник интерес ко второй схеме*

## 2. Ассоциативное отображение

Оно предполагает наличие в системе специального блока ассоциативной памяти (АП). АП всегда регистровая. АП позволяет получить доступ к информации по «ключу» за один тик. Существуют 2 вида АП

- с параллельным
- с последовательным доступом (нас интересует 1).

Схема имеет следующий вид.

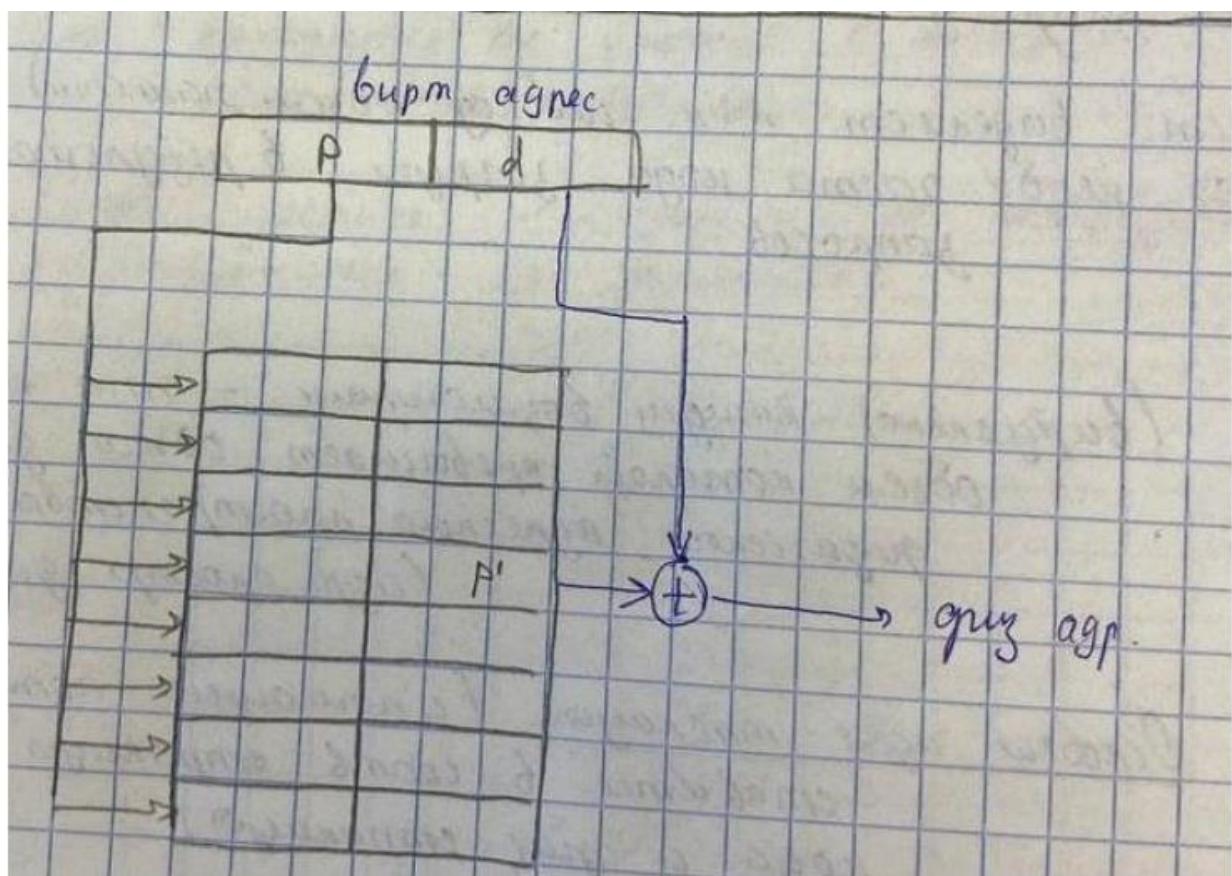
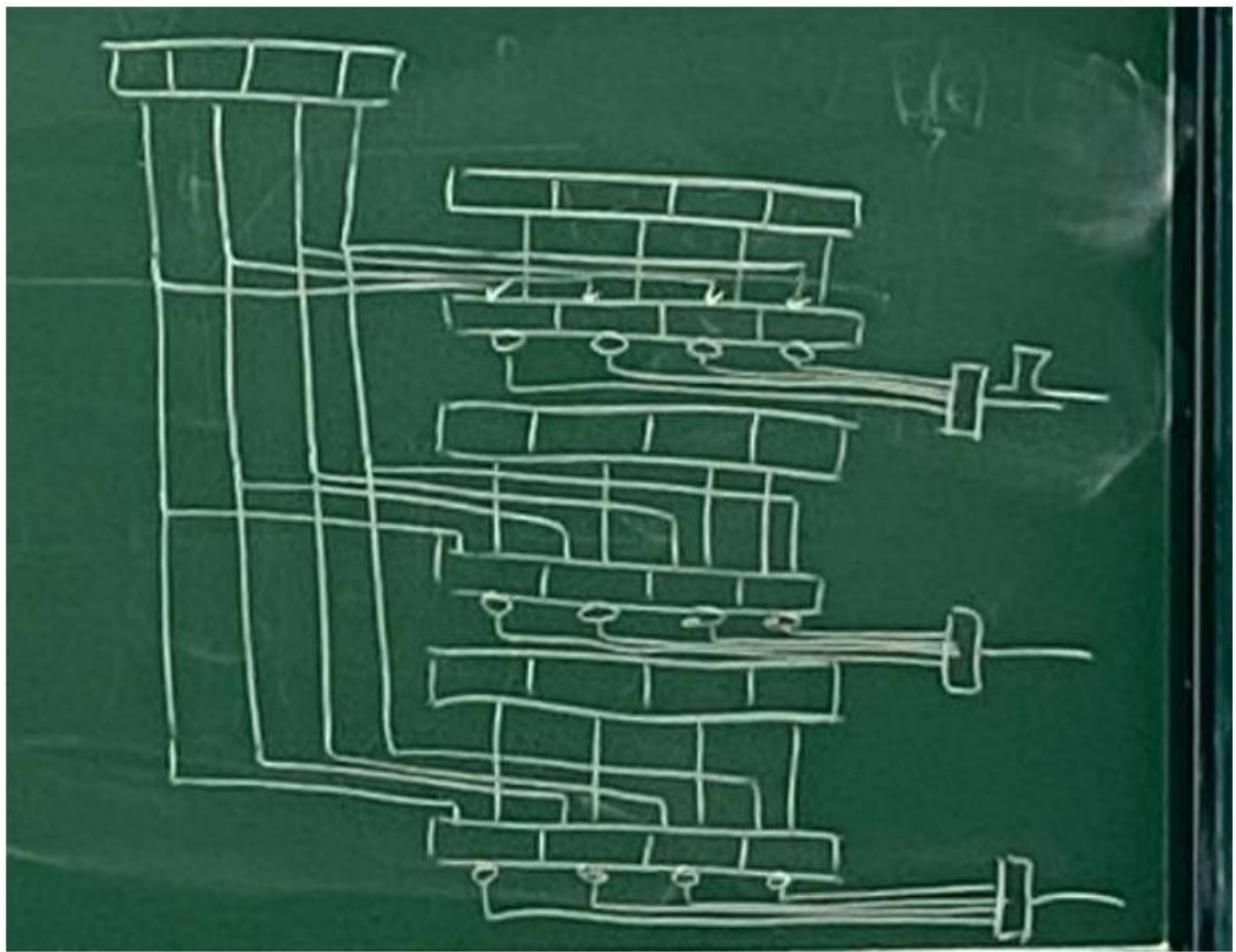


Таблица страниц загружается в специальную ассоциативную память. Номер страницы используется как ключ, сравниваются сразу все элементы всех дескрипторов (то есть все разряды всех строк) с ключом, и при совпадении происходит выборка информации/считывание/выбирается физический адрес (разные формулировки)



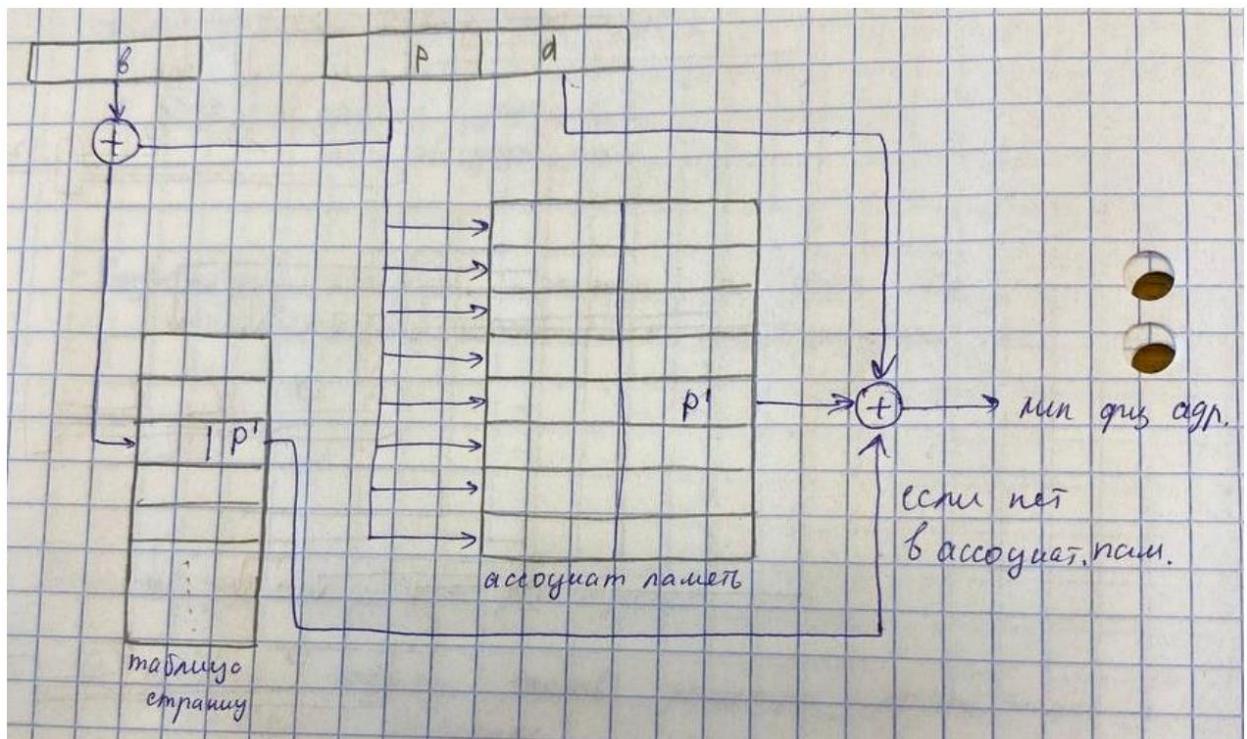
Но это дорогая память:

- она регистровая
- чтобы осуществить такое обращение к информации необходимо на каждый разряд соответствующего регистра поставить схему совпадения, собрать все сигналы на соответствующую схему, и если все сигналы совпали – послать сигнал разрешения считывания
- Логика удваивается + куча соединений.

Для больших таблиц страниц такое решение непригодно. Поэтому существует 3 схема

### 3. Ассоциативно-прямое отображение

Комбинация первых двух методов.



В системе имеется небольшая по объему ассоциативная память (сейчас называется кэш).

*Сейчас – на 8/16 регистров, позволяла обеспечить 90% скоростных показателей полностью ассоциативной памяти*

Страница сначала ищется в кэше. Если не найдена – обращение к физической памяти – таблице страниц. Если страница и там не найдена, то происходит страничное прерывание. Станичное прерывание в наших системах относится к исправимым исключениям – page fault.

После того как страница загружена в память, выполнение продолжится с той команды, на которой возникло прерывание.

*Регистр CR2 – линейного адреса ошибки обращения к странице, CR3 – регистр начального адреса каталога таблиц страниц.*

Код растет, размеры таблиц страниц увеличивается. Но это системные таблицы – они должны находиться в ядре системы. (В нашей лабе 2 таблицы находятся в той памяти, которая выделена для ядра системы) - боль

реализация страничного преобразования в компьютерах на базе процессоров Intel (x86): стандартное преобразование PAE в защищенном режиме – схемы, размеры таблиц и их количество на каждом этапе преобразования.

[9.2 Unix: концепция процессов; иерархия процессов, процессы «сироты», процессы «зомби», демоны; примеры из лабораторной работы \(5 программ\).](#)

(настоятельно предлагается писать **только** то, что не курсивом, а курсив прочитать, чтобы быть готовой к вопросам)

## **Unix: концепция процессов**

Unix - система разделения времени sharing time.

Linux это Unix-подобная ОС, которая реализует все его парадигмы. Как и в любой ОС, основной абстракцией является процесс. Процесс в Linux (как и в UNIX) это - программа, которая выполняется в отдельном защищенном виртуальном адресном пространстве (это уточнением определения, которое определяет процесс как программу в стадии выполнения)

*Когда пользователь регистрируется в системе, автоматически создается новый процесс, в котором выполняется оболочка (shell), например, /bin/bash. Если интерпретатору (shell) встречается команда, соответствующая выполняемому файлу, интерпретатор выполняет ее, начиная с точки входа (entry point). Для С-программ entry point - это функция main. Запущенная программа тоже может создать процесс, т.е. запустить какую-то программу и ее выполнение тоже начнется с функции main.*

В Unix новый взгляд на процесс: Часть времени процесс выполняется в режиме задачи (пользовательском режиме) и тогда он выполняет собственный код, часть времени процесс выполняется в режиме ядра и тогда он выполняет реентерабельный код операционной системы

*В режиме ядра процесс выполняет повторно входимые процедуры, то есть процедуры чистого кода. Повторная входимость означает, что одну и ту же процедуру могут одновременно использовать несколько процессов, причем эти процессы могут находиться в разных точках функции. Чистые функции - функции, которые не модифицируют сами себя. А что можно изменить? –variable. Поэтому из этих процедур вынесены все переменные.*

*В ОС существует специальные системные таблицы – те самые таблицы, которые какие-то функции могут редактировать, но они не находятся в коде функции. Это структуры, разделяемые процедурами ядра. То есть в самих чистых процедурах никаких переменных быть не может.*

В Linux поддерживается классическая схема мультипрограммирования (многопоточное ядро, написанное на чистом Си с использованием структур). Linux поддерживает параллельное (или квазипараллельное при наличии только одного процессора) выполнение процессов пользователя. Каждый процесс выполняется в собственном защищенном виртуальном адресном пространстве. Это значит, что ни один процесс не обратиться в адресное пространство другого процесса. Процессы защищены друг от друга и крах одного процесса никак не повлияет на другие выполняющиеся процессы и на

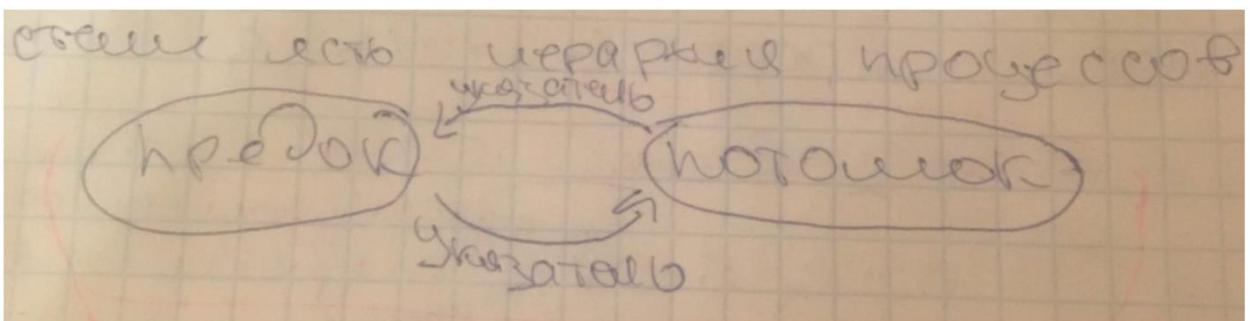
всю систему в целом. (Потоки могут разделять структуры (экземпляры структур), они, естественно глобальные. Системные таблицы. )

Ядро предоставляет системные вызовы для создания новых процессов и для управления запущенными процессами. Любая программа может начать выполняться только если другой процесс ее запустит.

В unix и linux процесс описывается структурой: unix - struct proc, linux – struct task\_struct. Эти структуры называются дескрипторами процесса. Огромные структуры, большинство полей – указатели на другие структуры. И экземпляр этой структуры, созданный для процесса, содержит всю информацию, чтобы можно было управлять этим процессом и выделять ему ресурсы. Это все – структуры ядра, они не доступны user.

### иерархия процессов

Любой процесс создается системным вызовом fork (это база, есть другая инфа, но ее рязановой не говорить). Есть в ядре функция clone. При этом создается новый процесс – процесс-потомок, который находится в отношении к процессу, вызвавшим fork, как потомок к предку. Это отношение поддерживается соответствующими указателями. Процесс, вызвавший fork сохраняет указатель на потомка, а потомок получает указатель на своего предка. Любой процесс может создать любое кол-во процессов. В результате fork создается иерархия процессов в отношении предок-потомок (parent-child).



Эта иерархия поддерживается указателями (в unix есть связный список типа дерево). Процессы в списке находятся в соответствии с их приоритетами. Приоритет может при этом повыситься. Тогда надо его передвинуть ближе к началу списка.

В системе всегда есть процесс с идентификатором 0 (запустивший систему) и процесс с идентификатором 1 (открывший терминал). Все идентификаторы в unix – целые положительные числа. Терминал – базовое понятие unix, это монитор, который подключается.... В системе может быть создано большое количество терминалов и у каждого будет Процесс с идентификатором 1. Процесс с идентификатором 1 является предком всех процессов, запущенных на данном терминале.

Почему именно двусвязный список? В системе таблицы не могут использоваться, ведь это массив структур, а это дополнительные накладные расходы при добавлении или удалении элементов: необходимо сдвигать. А тут-динамика, постоянно надо. А двусвязные – для ускорения поиска. А вообще сортировка – чтобы использовать быстрые поиски (бинарный, например). Если массив не отсортирован, придется полный перебор вообще. Поэтому двусвязный список – топ. Также в них есть двусвязные списки блокированных (или выполняемых, не успели) процессов и процессов-зомби.

В UNIX процесс, все его дочерние процессы и более отдаленные потомки образуют группу процессов.

*Когда пользователь отправляет сигнал с клавиатуры, тот достигает всех участников этой группы процессов, связанных на тот момент времени с клавиатурой (обычно это все действующие процессы, которые были созданы в текущем окне). Каждый процесс по отдельности может захватить сигнал, игнорировать его или совершить действие по умолчанию, которое должно быть уничтожено сигналом.*

### Дополнительно про fork

*Fork – создается новый процесс процесс–потомок, который является копией процесса предка в том смысле, что потомок наследует код предка, дескрипторы открытых файлов, сигнальную маску окружения.*

*В старых системах (из истории: в них написана для какой машины - PDP11) код предка копировался в адресное пространство потомка. То есть потомку создавалось собственное адресное пространство, которое описывается таблицами страниц (ранее - сегментов).*

*Очевидно, что это крайне неэффективно. В системе могут одновременно существовать какое-то количество копий одной и той же программы. Поэтому в современных системах существуют два способа оптимизации задачи создания нового процесса. (называют иногда оптимизация fork)*

1. Предложен в system5 и называется копирование при записи (*copy on write*).  
*Когда вызывается fork и создается потомок, для него создаются собственные карты трансляции адресов (в современных – таблицы страниц), и эти таблицы страниц ссылаются на страницы адресного пространства предка. При этом для страниц стека права меняются на only read и устанавливается флаг copy on write. Если предок или потомок пытаются изменить страницу, возникает исключение по правам доступа. Обрабатывая его, система обнаружит установленный флаг copy on write и создаст копию нужной страницы в адресном пространстве того процесса, который пытался ее изменить.*

*Дескриптор этой страницы должен быть добавлен в соответствующую таблицу страниц. В результате будут созданы только копии нужных страниц.*

*Решение сору on write решило проблему коллективного использования страниц и страницное преобразование (то есть управление памятью страницами по запросу) стало фактически единственным используемым*

*Ситуация, когда изменены права доступа к данным и стеку предка и установлен флаг сору on write существует до тех пор, пока процесс потомок не вызовет или системный вызов exit(), или системный вызов exec().*

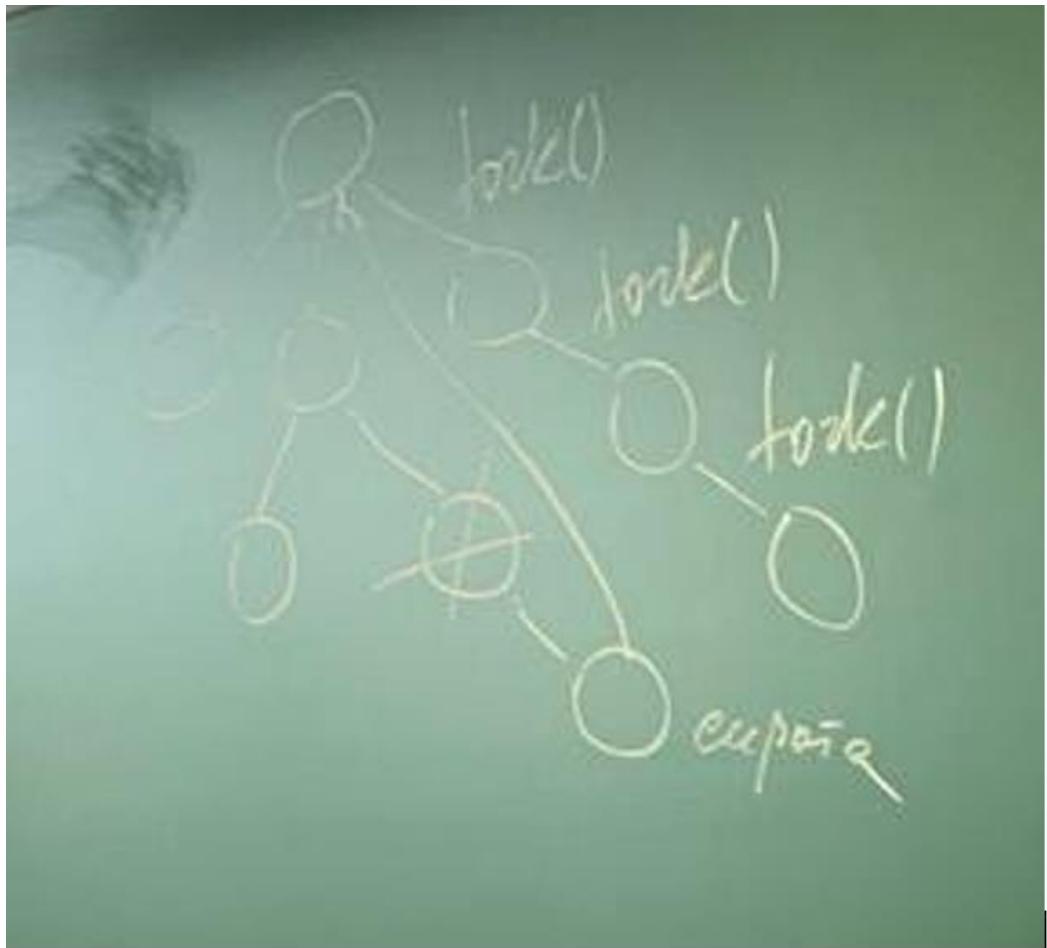
*Exit() – системный вызов завершения процесса.*

2. способ оптимизации – macos (unix bsd) реализован способ с системным вызовом vfork. В этом случае для потомка не создаются карты, а предок предоставляет потомку свои. При этом предок блокируется до того момента, пока потомок не вызовет exit/exec

*А exec для программы, которая передается exec в качестве параметра, создает адресное пространство, то есть - таблицу страниц*

**процессы «сироты»**

*В системе создается иерархия, поддерживаемая указателем. Но программы иногда аварийно завершаются. Вот завершился процесс, а потомок продолжает выполняться – иерархия разрушена, но Unix не может допустить этого.*



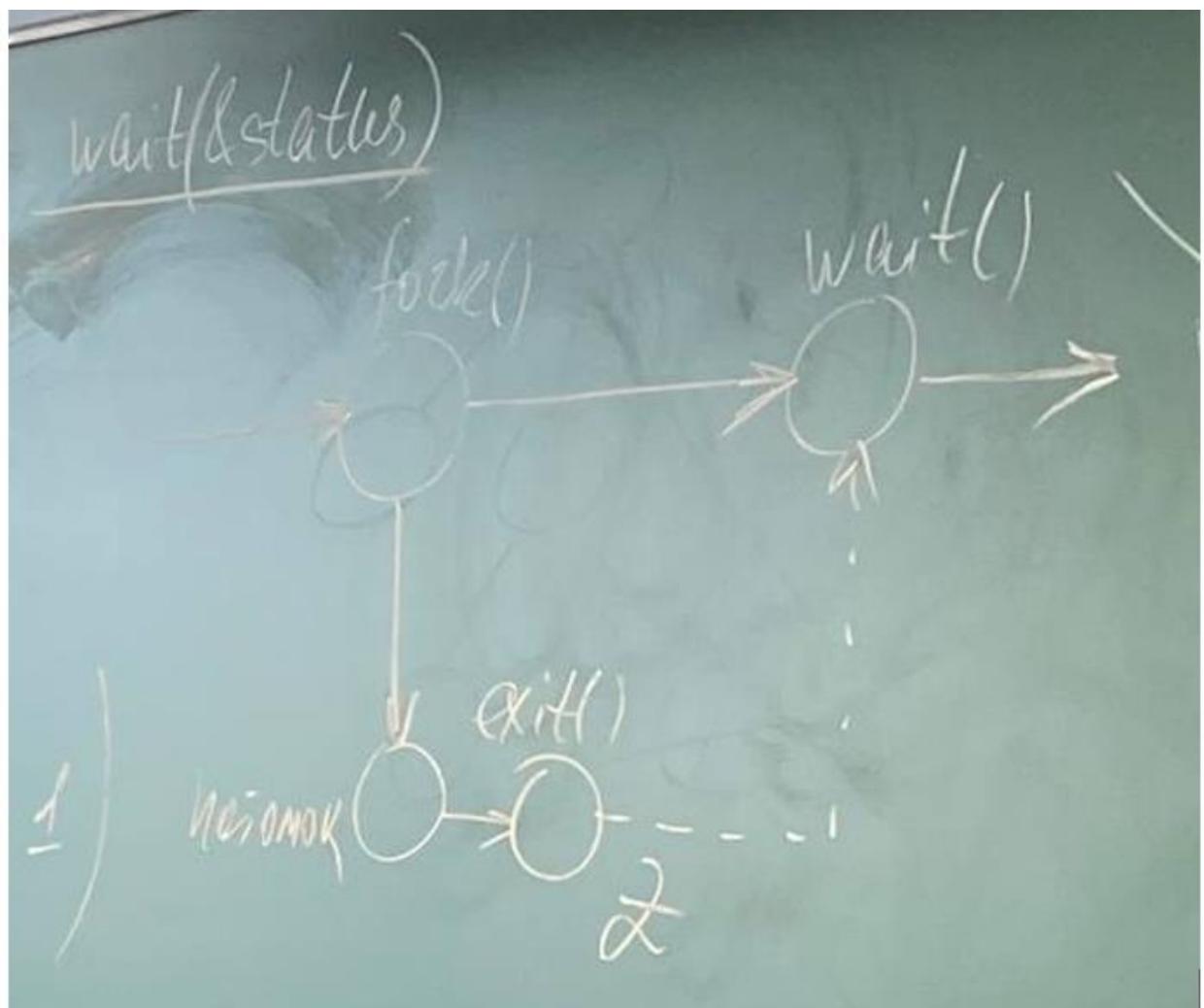
Процесс, у которого завершился предок, называется процесс-сирота.

При завершении процесса система проверяет, не остались ли незавершенные потоки (по дескриптору процесса, где есть указатели на потомков). Если да, то начинается процесс усыновления – его усыновит процесс с идентификатором 1 (открывший терминал). При этом потомок получит указатель на нового предка, а предок – на нового потомка (много действий).

Чтобы не появлялись сироты, в unix есть системный вызов `wait(&status)`. Система не контролирует, где вызван – в предке или потомке, потому что любой потомок может стать предком. Точно также она не контролирует, где вызывается `exec()`. Предок может проконтролировать код завершения потомка и ветвиться в зависимости от этого. Но с `wait` связана одна проблема в системе.

### процессы «зомби»

*Процесс выполняется и создает процесс-потомок. Тот аварийно завершился. А предок продолжал что-то делать, а только потом вызвал wait. Но предок уже не сможет получить инфу и останется навсегда заблокированным. Это решается с помощью состояния зомби.*



В unix все процессы проходят через состояние зомби – процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов (то есть дескриптор). Это сделано для того, чтобы процесс-предок, вызвавший системный вызов `wait()`, не был заблокирован навсегда и знал статус завершения своих ПОТОМКОВ.

Если дочерний процесс завершится первым, то он будет существовать как зомби, пока процесс-предок вызовет системный вызов `wait()` или завершится.

### **демоны**

Демон - процесс, который не имеет предков, существует сам и не входит ни в какие группы (сервисные функции). Выполняет какую-то фоновую задачу, не имеет управляющего терминала и, как следствие, обычно не интерактивный по отношению к пользователю.

**примеры из лабораторной работы (5 программ).**

1. **Процессы-сироты.** В программе создаются не менее двух потомков. В потомках вызывается sleep(). Чтобы предок гарантированно завершился раньше своих потомков. Продемонстрировать с помощью соответствующего вывода информацию об идентификаторах процессов и их группе. Продемонстрировать «усыновление». Для этого надо в потомках вывести идентификаторы: собственный, предка, группы до блокировки и после блокировки.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#define RET_OK 0
#define RET_ERR_FORK 1
#define FORK_OK 0
#define FORK_ERR -1
#define INTERVAL 1

int main()
{
    pid_t childpid1, childpid2;
    if ((childpid1 = fork()) == FORK_ERR)
    {
        perror("Can't fork first child process.\n");
        return RET_ERR_FORK;
    }
    else if (childpid1 == FORK_OK)
    {
        printf("First child process: pid = %d, ppid = %d, pgid =
%d\n",
               getpid(), getppid(), getpgrp());

        sleep(INTERVAL);
        printf("First child process (has become an orphan): pid =
%d, ppid = %d, pgid = %d\n",
               getpid(), getppid(), getpgrp());

        printf("First child process is dead now\n");

        exit(RET_OK);
    }

    if ((childpid2 = fork()) == FORK_ERR)
    {
        perror("Can't fork second child process.\n");
        return RET_ERR_FORK;
    }
    else if (childpid2 == FORK_OK)
    {
```

```

        printf("Second child process: pid = %d, ppid = %d, pgrp =
%d\n",
               getpid(), getppid(), getpgrp()));

        sleep(INTERVAL);
        printf("Second child process (has become an orphan): pid =
%d, ppid = %d, pgrp = %d\n",
               getpid(), getppid(), getpgrp()));

        printf("Second child process is dead now\n");
        exit(RET_OK);
    }

    printf("Parent process: pid = %d, pgrp = %d, childpid1 = %d,
childpid2 = %d\n",
           getpid(), getpgrp(), childpid1, childpid2);
    printf("Parent process is dead now\n");
    return RET_OK;
}

```

2. Предок ждет завершения своих потомком, используя системный вызов `wait()`. Вывод соответствующих сообщений на экран. В программе необходимо, чтобы предок выполнял анализ кодов завершения потомков.

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define RET_OK 0
#define RET_ERR_FORK 1

#define FORK_OK 0
#define FORK_ERR -1

#define INTERVAL 1

int main()
{
    pid_t childpid1, childpid2, childpid;
    if ((childpid1 = fork()) == FORK_ERR)
    {
        perror("Can't fork first child process.\n");

```

```

        return RET_ERR_FORK;
    }
    else if (childpid1 == FORK_OK)
    {
        printf("First child process: pid = %d, ppid = %d, pggrp =
%d\n",
               getpid(), getppid(), getpgrp());

        exit(RET_OK);
    }

    if ((childpid2 = fork()) == FORK_ERR)
    {
        perror("Can't fork second child process.\n");
        return RET_ERR_FORK;
    }
    else if (childpid2 == FORK_OK)
    {
        printf("Second child process: pid = %d, ppid = %d, pggrp =
%d\n",
               getpid(), getppid(), getpgrp());

        exit(RET_OK);
    }

    sleep(INTERVAL);
    printf("Parent process: pid = %d, pggrp = %d, childpid1 = %d,
childpid2 = %d\n",
           getpid(), getpgrp(), childpid1, childpid2);

    int ch_status;
    for (int i = 0; i < 2; i++)
    {
        childpid = wait(&ch_status);
        printf("Child with pid = %d has finished with status
%d\n", childpid, ch_status);

        if (WIFEXITED(ch_status))
            printf("Child exited normally with exit code %d\n",
WEXITSTATUS(ch_status));
        else if (WIFSIGNALED(ch_status))
            printf("Child process ended with a non-intercepted signal
number %d\n", WTERMSIG(ch_status));
        else if (WIFSTOPPED(ch_status))
            printf("Child process was stopped by a signal %d\n",
WSTOPSIG(ch_status));
    }

    printf("Parent process is dead now\n");

```

```
    return RET_OK;
}
```

3. Потомки переходят на выполнение других программ, которые передаются системному вызову exec() в качестве параметра. Потомки должны выполнять разные программы. Предок ждет завершения своих потомков с анализом кодов завершения. На экран выводятся соответствующие сообщения.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

#define RET_OK 0
#define RET_ERR_FORK 1
#define RET_CANT_EXECLP 2

#define FORK_OK 0
#define FORK_ERR -1

#define INTERVAL 1

int main()
{
    pid_t childpid1, childpid2, childpid;
    if ((childpid1 = fork()) == FORK_ERR)
    {
        perror("Can't fork first child process.\n");
        return RET_ERR_FORK;
    }
    else if (childpid1 == FORK_OK)
    {
        printf("First child process: pid = %d, ppid = %d, pgid =
%d\n",
               getpid(), getppid(), getpgrp());
        if (execl("./task3_sum", "task3_sum", "2", "3", NULL) <
0)
        {
            perror("Can't execl from first child.\n");
            exit(RET_CANT_EXECLP);
        }
        exit(RET_OK);
    }

    if ((childpid2 = fork()) == FORK_ERR)
    {
```

```

        perror("Can't fork second child process.\n");
        return RET_ERR_FORK;
    }
    else if (childpid2 == FORK_OK)
    {
        sleep(INTERVAL);
        printf("Second child process: pid = %d, ppid = %d, pgrp =
%d\n",
               getpid(), getppid(), getpgrp());
        if (execlp("./task3_write", "task3_write" ,
"file_to_write.txt", "This is test info", NULL) < 0)
        {
            perror("Can't execlp from second child.\n");
            exit(RET_CANT_EXECLP);
        }

        exit(RET_OK);
    }

    sleep(INTERVAL * 2);
    printf("Parent process: pid = %d, pgrp = %d, childpid1 = %d,
childpid2 = %d\n",
           getpid(), getpgrp(), childpid1, childpid2);

    int ch_status;
    for (int i = 0; i < 2; i++)
    {
        childpid = wait(&ch_status);
        printf("Child with pid = %d has finished with status
%d\n", childpid, ch_status);

        if (WIFEXITED(ch_status))
            printf("Child exited normally with exit code %d\n",
WEXITSTATUS(ch_status));
        else if (WIFSIGNALED(ch_status))
            printf("Child process ended with a non-intercepted
signal number %d\n", WTERMSIG(ch_status));
        else if (WIFSTOPPED(ch_status))
            printf("Child process was stopped by a signal %d\n",
WSTOPSIG(ch_status));
    }

    printf("Parent process is dead now\n");
    return RET_OK;
}

```

4. Предок и потомки обмениваются сообщениями через неименованный программный канал. Причем оба потомка пишут свои сообщения в один программный канал, а предок их считывает из канала. Потомки должны посыпать предку разные сообщения по содержанию и размеру. Предок считывает сообщения от потомков и выводит их на экран. Предок ждет завершения своих потомков и анализирует код их завершения. Вывод соответствующих сообщений на экран.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>

#define RET_OK 0
#define RET_ERR_FORK 1
#define RET_ERR_PIPE 2

#define FORK_OK 0
#define FORK_ERR -1

#define INTERVAL 1
#define N_CHILDREN 2

#define MSG1 "London is the capital of Great Britain\n"
#define LEN1 40
#define MSG2 "ABRA-kadabra\n"
#define LEN2 14
#define LENMAX 40

int main()
{
    pid_t childpid1, childpid2, childpid;
    int fd[2];

    if (pipe(fd) == -1)
    {
        perror("Can't pipe\n");
        return RET_ERR_PIPE;
    }
```

```

if ((childpid1 = fork()) == FORK_ERR)
{
    perror("Can't fork first child process.\n");
    return RET_ERR_FORK;
}
else if (childpid1 == FORK_OK)
{
    printf("First child process: pid = %d, ppid = %d, pgrp =
%d\n",
           getpid(), getppid(), getpgrp());

    close(fd[0]);
    write(fd[1], MSG1, strlen(MSG1) + 1);
    printf("Message from first child was sent\n");

    exit(RET_OK);
}

if ((childpid2 = fork()) == FORK_ERR)
{
    perror("Can't fork second child process.\n");
    return RET_ERR_FORK;
}
else if (childpid2 == FORK_OK)
{
    printf("Second child process: pid = %d, ppid = %d, pgrp =
%d\n",
           getpid(), getppid(), getpgrp());

    close(fd[0]);
    write(fd[1], MSG2, strlen(MSG2) + 1);
    printf("Message from second child was sent\n");

    exit(RET_OK);
}

sleep(INTERVAL);
printf("Parent process: pid = %d, pgrp = %d, childpid1 = %d,
childpid2 = %d\n",
      getpid(), getpgrp(), childpid1, childpid2);

```

```
int ch_status;
for (int i = 0; i < N_CHILDREN; i++)
{
    childpid = wait(&ch_status);
    printf("Child with pid = %d has finished with status
%d\n", childpid, ch_status);

    if (WIFEXITED(ch_status)) // не равно нулю, если дочерний
        процесс успешно завершился.
        printf("Child exited normally with exit code %d\n",
        WEXITSTATUS(ch_status));
    else if (WIFSIGNALED(ch_status)) // возвращает истинное
        значение, если дочерний процесс завершился из-за необработанного сигнала.
        printf("Child process ended with a non-intercepted signal
        number %d\n", WTERMSIG(ch_status));
    else if (WIFSTOPPED(ch_status)) // возвращает истинное
        значение, если дочерний процесс, из-за которого функция вернула управление,
        в настоящий момент остановлен
        printf("Child process was stopped by a signal %d\n",
        WSTOPSIG(ch_status));
}

char message[LENMAX] = "\0";

printf("Reading messages from children.\n");
close(fd[1]);

if (read(fd[0], message, LEN1) < 0)
    printf("No messages from first child.\n");
else
    printf("Message from first child:\n%s", message);

if (read(fd[0], message, LEN2) < 0)
    printf("No messages from second child.\n");
else
    printf("Message from second child:\n%s", message);

printf("Parent process is dead now\n");
return RET_OK;
}
```

5. Предок и потомки аналогично п.4 обмениваются сообщениями через неименованный программный канал. В программу включается собственный обработчик сигнала. С помощью сигнала меняется ход выполнения программы. При получении сигнала потомки записывают сообщения в канал, если сигнал не поступает, то не записывают. Предок ждет завершения своих потомков и анализирует коды их завершений. Вывод соответствующих сообщений на экран.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <string.h>
#include <signal.h>

#define RET_OK 0
#define RET_ERR_FORK 1
#define RET_ERR_PIPE 2

#define FORK_OK 0
#define FORK_ERR -1

#define INTERVAL 2
#define N_CHILDREN 2

#define MSG1 "London is the capital of Great Britain\n"
#define LEN1 40
#define MSG2 "ABRA-kadabra\n"
#define LEN2 14
#define LENMAX 40

short flag_writing_allowed = 0;

void allow_writing(int signal)
{
    flag_writing_allowed = 1;
    printf("\nSignal %d was caught. Writing is allowed now.\n",
signal);
}

int main()
```

```
{  
    pid_t childpid1, childpid2, childpid;  
    int fd[2];  
  
    // назначение обработчика сигнала  
    signal(SIGINT, allow_writing);  
    printf("Press \"CTRL+C\" to allow writing\n");  
    sleep(INTERVAL);  
  
    if (pipe(fd) == -1)  
    {  
        perror("Can't pipe\n");  
        return RET_ERR_PIPE;  
    }  
  
    if ((childpid1 = fork()) == FORK_ERR)  
    {  
        perror("Can't fork first child process.\n");  
        return RET_ERR_FORK;  
    }  
    else if (childpid1 == FORK_OK)  
    {  
        sleep(INTERVAL);  
        printf("First child process: pid = %d, ppid = %d, pgid =  
%d\n",  
            getpid(), getppid(), getpgrp());  
  
        if (flag_writing_allowed)  
        {  
            close(fd[0]);  
            write(fd[1], MSG1, strlen(MSG1) + 1);  
            printf("Message from first child was sent\n");  
        }  
        else  
        {  
            printf("Writing to pipe for first child is not  
allowed\n");  
        }  
  
        exit(RET_OK);  
    }  
}
```

```

    if ((childpid2 = fork()) == FORK_ERR)
    {
        perror("Can't fork second child process.\n");
        return RET_ERR_FORK;
    }
    else if (childpid2 == FORK_OK)
    {
        sleep(INTERVAL);
        printf("Second child process: pid = %d, ppid = %d, pgrp =
%d\n",
               getpid(), getppid(), getpgrp());

        if (flag_writing_allowed)
        {
            close(fd[0]);
            write(fd[1], MSG2, strlen(MSG2) + 1);
            printf("Message from second child was sent\n");
        }
        else
        {
            printf("Writing to pipe for second child is not
allowed\n");
        }
    }

    exit(RET_OK);
}

printf("Parent process: pid = %d, pgrp = %d, childpid1 = %d,
childpid2 = %d\n",
       getpid(), getpgrp(), childpid1, childpid2);

sleep(INTERVAL);

int ch_status;
for (int i = 0; i < N_CHILDREN; i++)
{
    childpid = wait(&ch_status);
    printf("Child with pid = %d has finished with status
%d\n", childpid, ch_status);

    if (WIFEXITED(ch_status))

```

```

        printf("Child exited normally with exit code %d\n",
WEXITSTATUS(ch_status));
        else if (WIFSIGNALED(ch_status))
            printf("Child process ended with a non-intercepted signal
number %d\n", WTERMSIG(ch_status));
        else if (WIFSTOPPED(ch_status))
            printf("Child process was stopped by a signal %d\n",
WSTOPSIG(ch_status));
    }

char message[LENMAX] = { 0 };

printf("Reading messages from children.\n");
close(fd[1]);

if (read(fd[0], message, LEN1) < 0)
    printf("No messages from first child.\n");
else
    printf("Message from first child:\n%s", message);

if (read(fd[0], message, LEN2) < 0)
    printf("No messages from second child.\n");
else
    printf("Message from second child:\n%s", message);

printf("Parent process is dead now\n");
return RET_OK;
}

```

## 10 билет

10.1 (2021) Классификация ядер операционных систем. Особенности ОС с микроядром. Три состояния блокировки процесса при передаче сообщений. Достоинства и недостатки микроядерной архитектуры, операционная система Mach: основные абстракции.  
(это доп вопрос 26.10)

### Классификация

Существует 2 типа ядер: монолитные (более ранние) и микроядра

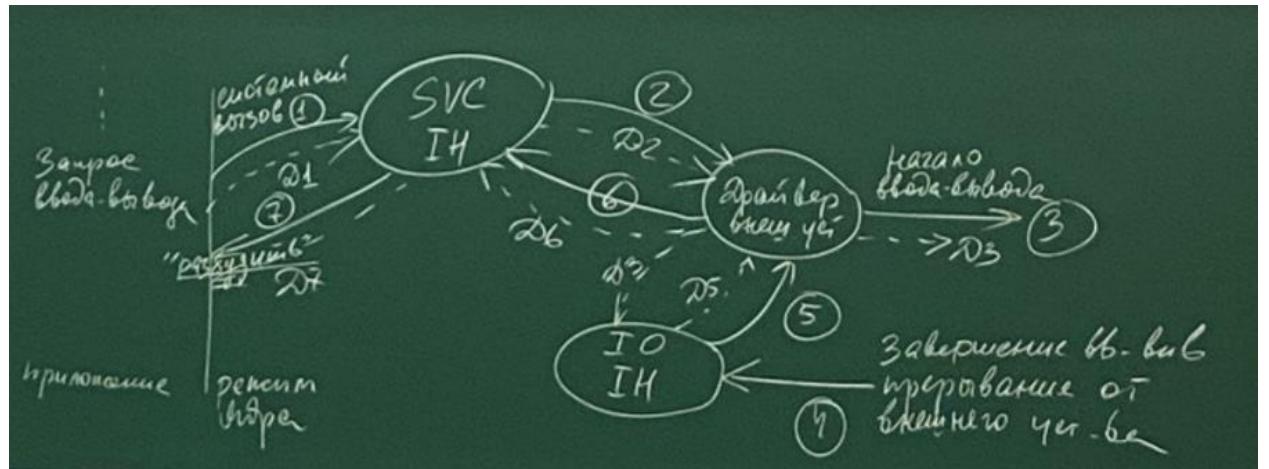
### Монолитное ядро

программа, имеющая модульную структуру, состоящая из подпрограмм. Windows, Unix, Linux – монолитные ядра. Linux – минимизированное (вынесен графический интерфейс).

В архитектуре с монолитным ядром все услуги для прикладного приложения выполняют отдельные части кода ядра (в адресном пространстве ядра)

Все построено на прерываниях – системные вызовы, исключения и аппаратные прерывания (это события, переводящие в режим ядра)

*Последовательность действий в системе при запросе приложения на ввод/вывод*



*Происходит вызов read/write и система переходит в режим ядра (ни одна система не дает прямое обращение к внешним устройствам)*

1. Системный вызов из функции, который переводит в режим ядра, туда же соответствующие данные. Super visor call . Супервизор - ... в стадии выполнения. IH - interrupt handler.
2. В результате обработки системного вызова будет вызван драйвер внешнего устройства (программа ввода/вывода). Драйверу будут переданы данные в его формате.
3. Драйвер инициализирует работу внешнего устройства. На этом управление процессором работы заканчивается, он отключается, потому что работой внешнего устройства управляют контроллеры.
4. По завершении операции ввода/вывода будет сформировано контроллером устройства прерывание, которое в прост схеме поступит на контроль прерывания и в результате будет определен адрес точки входа обработчика прерывания,
5. И обработчик начнет выполнятся (IO IH). Процессор перейдет на выполнение обработчика, тк аппаратные прерывания имеют наивысший приоритет. ОП входят в состав драйвера и является одной из точек входа драйвера внешнего устройства. Драйвер всегда содержит 1 обработчик прерывания.

6. Поскольку обработчик – точка входа драйвера, у драйвера есть *call back* функция – задача вернуть запрашиваемые данные приложению. В результате драйвер через подсистему ввода/вывода должен передать данные приложению.

7. Для этого работы приложения обновлены. Процесс, который запросил ввода, блокируется. (7) – разбудить.

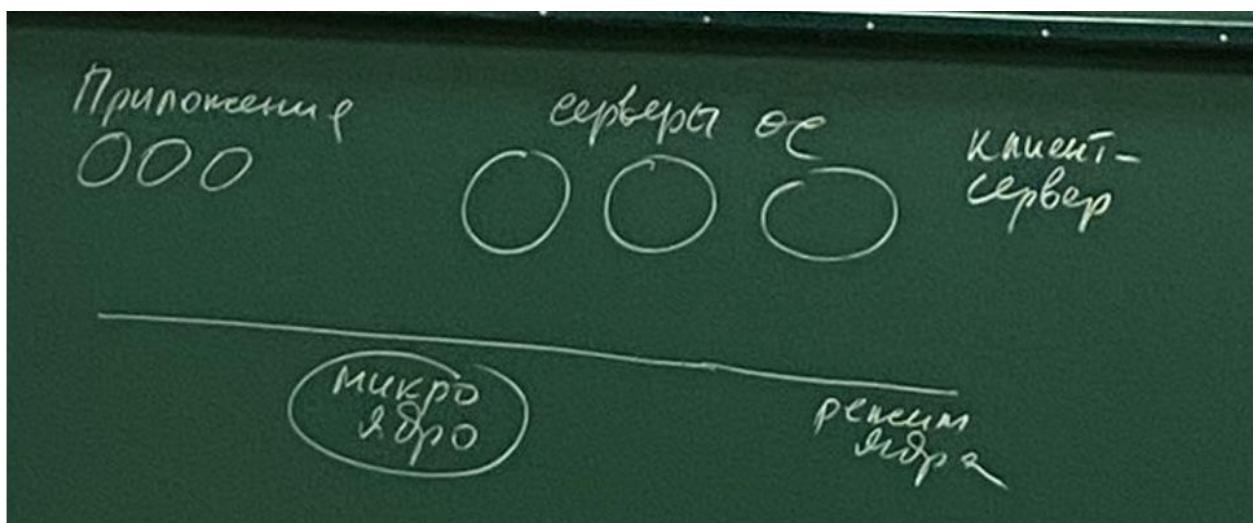
С монитором мы работаем как с памятью – то есть *input/output* – использование команд ввода вывода, и это приводит к блокировкам.

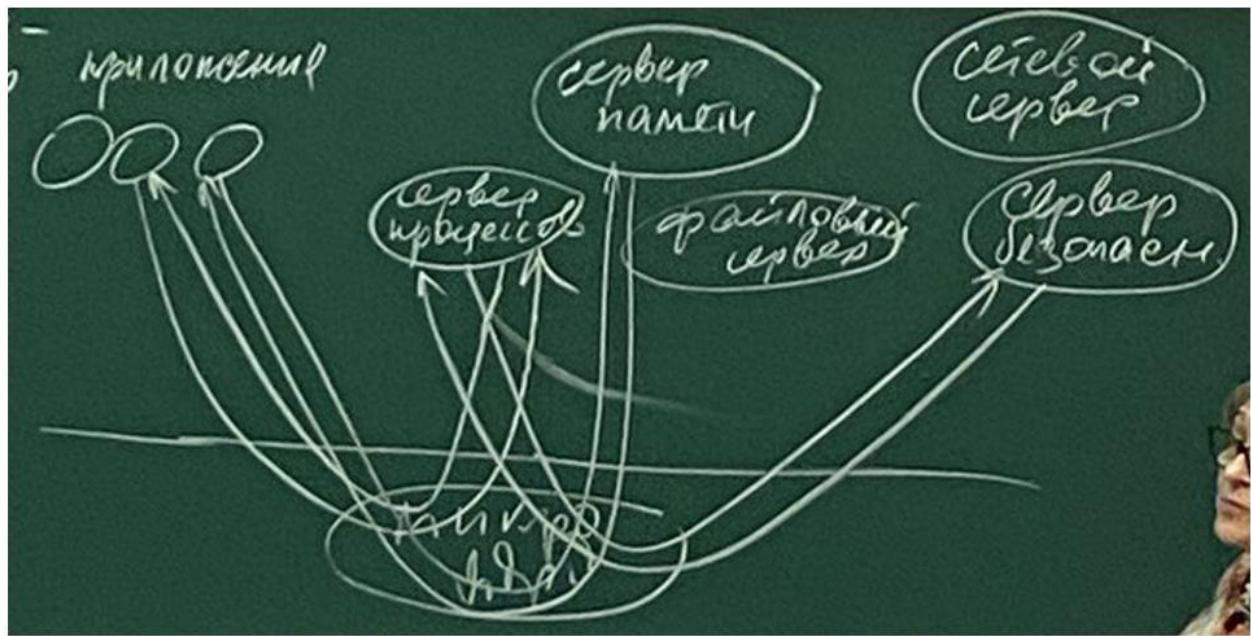
Возникновение прерывания происходит асинхронно. Чтобы получить значение, процесс разблокируется. Поэтому эта схема называется блокирующий синхронный ввод-вывод

### Микроядерная архитектура, особенности ОС с микроядром

Mach – первая ОС с микроядром. Классическим примером также является Symbian OS.

Идея – оставить в ядре только самые низкоуровневые функции, остальные функции выполняются в режиме пользователя. Остальная часть ОС реализуется в виде отдельных процессов, которые выполняются в собственных АП.





Взаимодействие между компонентами ОС выполняются с помощью посылки и приема сообщений, причем этот механизм обеспечиваются специальным модулем ядра, который называется микроядро.

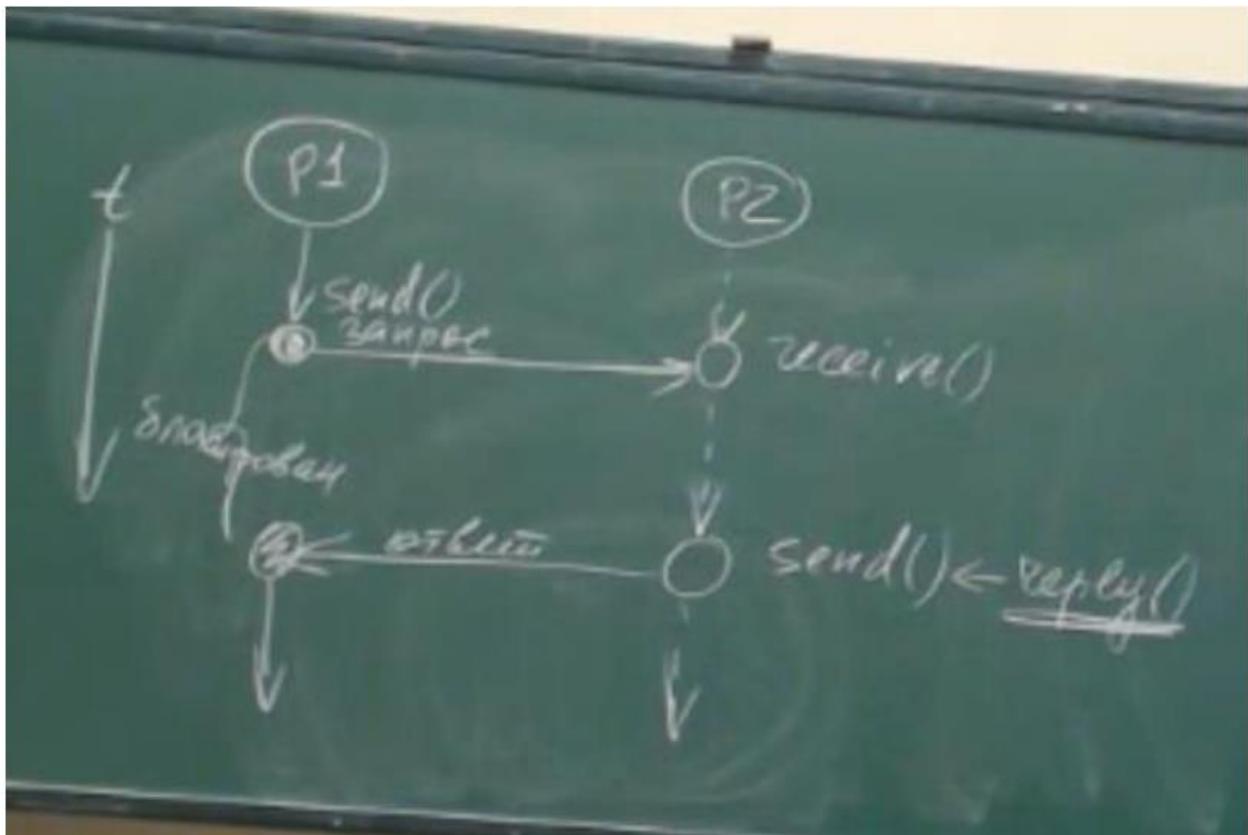
Такое взаимодействие выполняется по модели клиент – сервер. Программы ОС принято называть серверами ОС. Монолитное ядро с помощью системных вызовов предоставляет приложениям сервис. Программы обращаются к серверу с к-либо запросами, эти запросы сервер обрабатывает, в результате формируется ответ, ожидаемый клиентом. Взаимодействие осуществляется по протоколу (соглашение).

Микроядро также обеспечивает диспетчеризацию процессов, первичную обработку прерываний и низкоуровневое управление памятью. Микроядро реализует взаимодействие с аппаратным уровнем вычислительной системы (внешние устройства или ОЗУ).

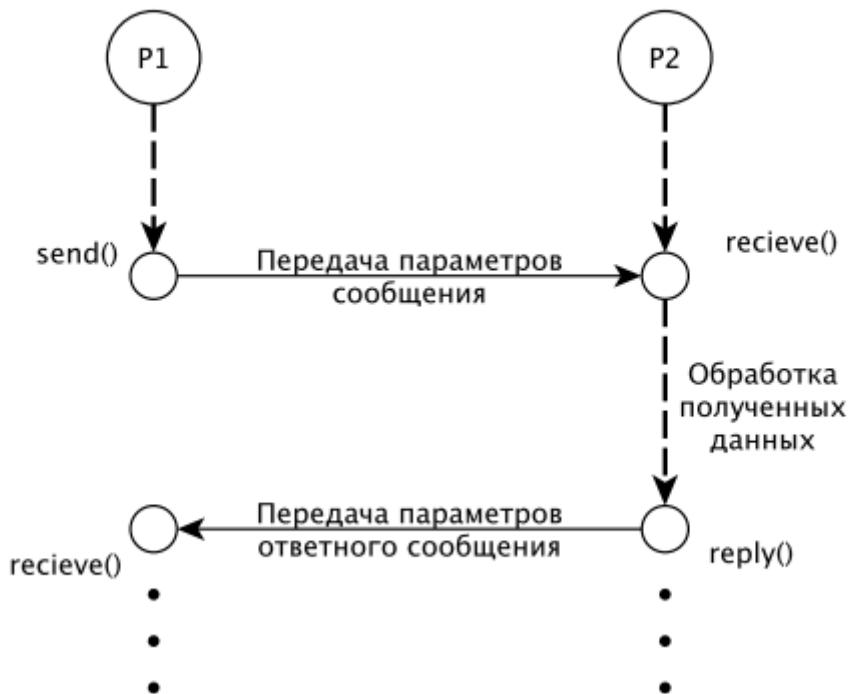
Взаимодействие клиент – сервер должно быть надежным. Посылка сообщения должна подтверждаться сообщением о его приеме.

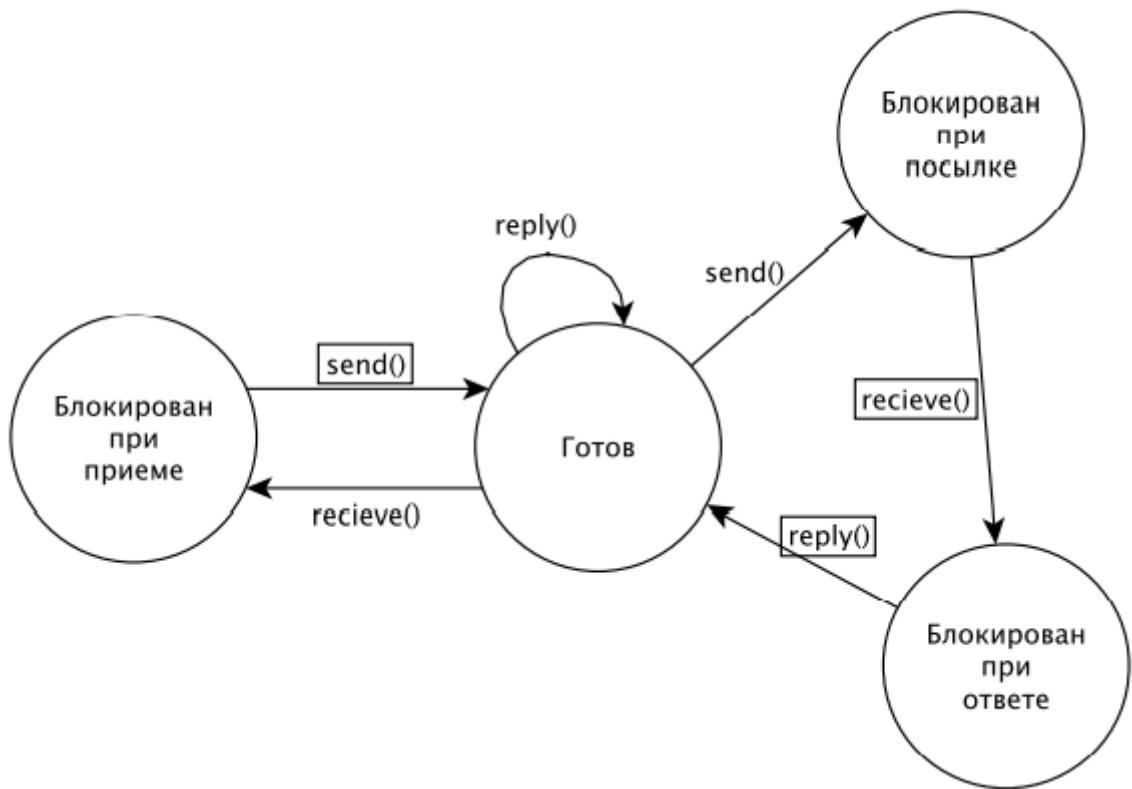
В результате мы вновь рассматриваем следующую диаграмму

### Три состояния блокировки процесса при передаче сообщений



(тут лучше видно)





В прямоугольниках команды сервера (функции, которые выполняются от имени другого процесса), без – клиента (не очень поняла про что это)

Все 3 состояния будут присутствовать – блокирован при посыпке, блокирован при ответе, блокирован при приеме. Это очень важная диаграмма состояний, напрямую связана с эффективностью микроядерной архитектуры. Время этих блокировок нельзя предсказать, это вероятностные вещи.

Очевидно, что передача сообщений связана с неконтролируемыми задержками. А так же контролируемыми временными затратами, в частности в микроядерной архитектуре больше переключений в режим ядра (как минимум в 2 раза)

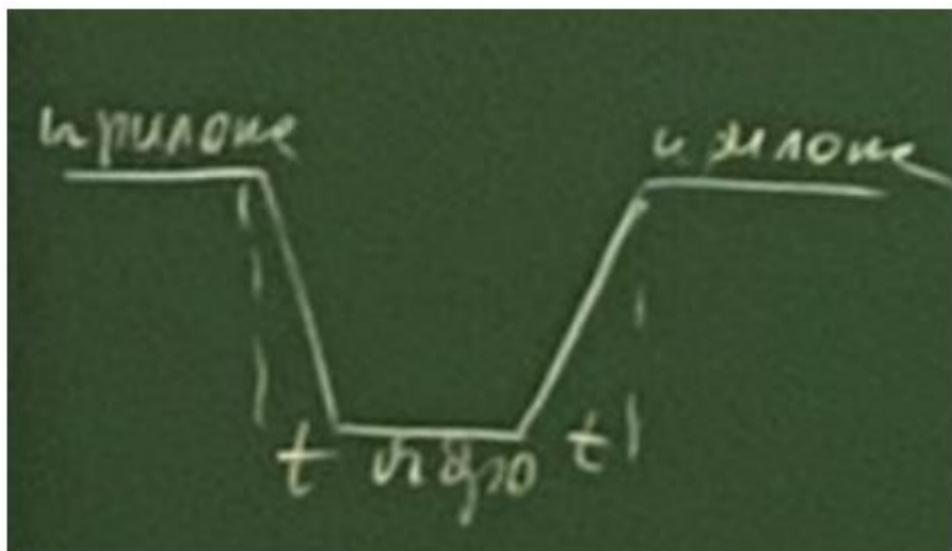
В результате микроядерная архитектура является не эффективной.

## Достоинства и недостатки микроядра

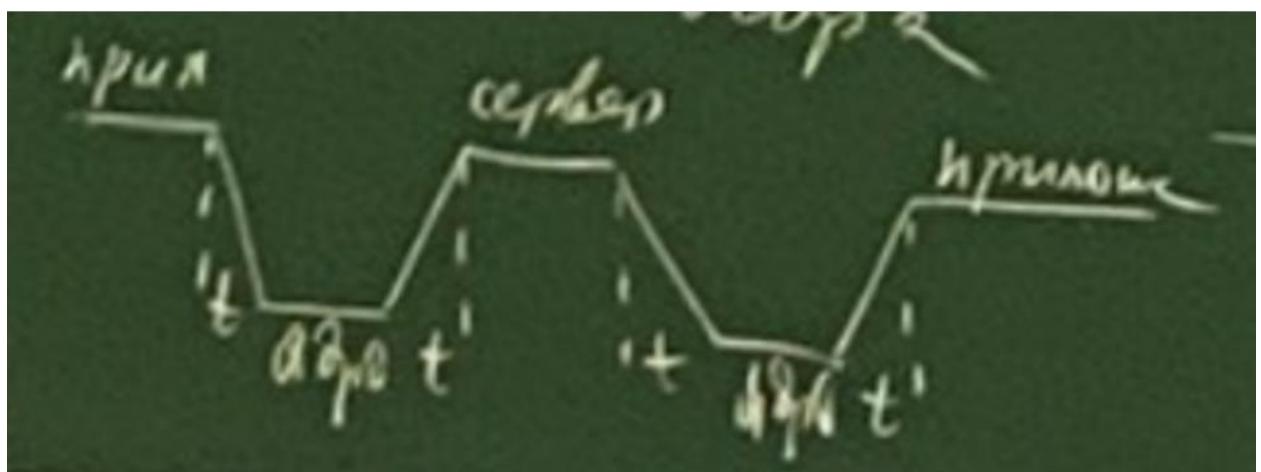
### Недостатки

- Возникают задержки при передаче сообщений моделью клиент-сервер, поскольку взаимодействие должно быть надежным.
- При обработке системного вызова происходит минимум в 2 раза больше переключений из режима ядра и обратно

В монолитном ядре при обработке системного вызова будет выполнено 2 переключения – 1 из режима задачи в режим ядра, 2 – обратно.



В мя – минимум 4



Несмотря на то, что эффективность мя архитектуры намного ниже, интерес не утрачивается. В чем же привлекательность.

Достоинства

- Выделено микроядро (низкоуровневые функции, связанные с аппаратной частью), большая часть кода (высокоуровневые функции) вынесена в режим пользователя и может быть изменена без перекомпиляции ядра.

Современные ОС предоставляют возможность внесения в монолитное ядро собственных функций без перекомпиляции. В Unix используются загружаемые модули ядра. Но с помощью них можно сделать не все. Если надо изменить структуру ядра – надо перекомпилировать. Поэтому идея микроядра – вносить изменения в ОС, имея широкие возможности и не перекомпилировать.

## Mach

Как и любая ОС, она определяет основные абстракции, на которых базируется ее работа.

### 5 основных абстракций

1. Процессы (*имеет виртуальное адресное пространство; в этом ad. pr. находится код, данные и несколько стеков; процесс – единица декомпозиции системы, так как именно процесс является владельцем ресурсов*);
2. Threads – потоки (*поток – непрерывная часть кода программы, которая может выполняться параллельно с другими частями кода; поток – независимо планируемый контекст выполнения, разделяющий единое адресное пространство процесса с другими потоками; единица планирования – поток; потоку принадлежит аппаратный контекст и счетчик команд; в процессе может быть один поток*)

И понятие процесс, и понятие поток не отличается от классического определения.

3. Объекты памяти
4. Порты (*защищенный почтовый ящик, который способен поддерживать очередь сообщений. (очередь сообщений – упорядоченный список сообщений)*)
5. Сообщения (*межпроцессное взаимодействие осуществляется с помощью сообщений; для передачи сообщений определена абстракция порт*)

Отличительная особенность – понятие объект памяти.

Memory object – представляет собой структуру данных, которая мб отображена в АП процесса. Объекты памяти могут занимать одну или несколько страниц и являются основой для системы управления виртуальной памятью (*в основе управления памяти – кластеризация (заранее загрузим следующие страницы)*). Выполняется страницами по запросам. Когда процесс обращается к объекту памяти, который отсутствует в физической памяти, возникает страничное прерывание. Ядро его обрабатывает, но в отличие от других систем, ядро для загрузки страницы посылает сообщение серверу режима пользователя (*серверу памяти*), только после того, как запрос будет обработан, ядром будет получен ответ от сервера, и тогда соответствующая страница мб загружена в ОП

Межпроцессное взаимодействие в этой ОС основано на передаче сообщений. Чтобы передавать сообщения, процесс пользователя просит ядро создать защищенный почтовый ящик, который в mach называется порт. Порт создается в АП ядра и способен поддерживать очередь сообщений.

Как правило, очередь имеет определенный размер и если процесс пытается послать в нее сообщение, а она уже заполнена, то такой процесс будет блокирован до того момента, когда порт не будет разгружен от этих сообщений.

В мач поддерживаются разные типы портов:

- Порт процесса (используется процессом для взаимодействия с ядром) – туда процесс с помощью сообщений посылает запросы на обслуживание (как в remote

proc call – не надо знать все тонкости обработки запросов). Внешне выглядит как системный вызов

- Порт загрузки. Используется при старте системы. Самый первый процесс (init) читает оттуда имена портов ядра, которые обеспечивают самые важные сервисы системы
- Порт особых ситуаций – используется системой для передачи сообщений об ошибках процессу. В монолитном ядре это называется исключения.
- Зарегистрированные порты – используются для обеспечения взаимодействия между процессами со стандартными системными сервисами. *Процесс может предоставить другому процессу посылать/получать сообщения через один из принадлежащих ему портов; такая возможность реализуется с помощью мандата, который включает указатель на порт и список прав которыми обладает другой процесс по отношению к данному порту; может выполнить команду send или только команду receive);*

Но как пишут соломон с русиновичем, для коммерческих реализаций мя mach перестает быть уже таким микро. В частности, файловая подсистема, поддержка сетей и управление памятью в коммерческих системах mach выполняется в режиме ядра как в системах с монолитным ядром.

Причина проста: системы, построенные строго по принципу мя плохи с коммерческой тз из-за низкой эффективности. Но мя используется в системах реального времени.

Например, широко известная срв QNX построена на основе мя архитектуры.

[10.2 \(2021\) Обработчик прерывания ДОС int 8h: функции; контроллер прерываний – схема, маскируемые и немаскируемые прерывания; запрет и разрешение маскируемых прерываний в обработчике int 8h, префиксная команда lock. Пример кода обработчика прерывания от системного таймера из лабораторной работы по защищенному режиму \(особенности\).](#)

## Функции

адресация (на всякий)

### **Контроллер прерываний - схема**

В шинной архитектуре (также есть канальная) внешними устройствами управляют контроллеры или адаптеры.

Контроллер – программно-управляемое устройство (имеется набор регистров и некоторая логика), находящееся в внешнем устройстве, а адаптер – на материнской плате

[Адресация аппаратных прерываний в з-р.](#)

### **Маскируемые и немаскируемые прерывания**

верного ответа вроде нет)

Маскируемые вызываются по маске - соответственно вызов таких прерываний можно запретить установкой соответствующих битов в *регистре маскирования прерываний??*. Не маскируемые запретить нельзя, такими могут быть ошибки различные.

### **запрет и разрешение маскируемых прерываний в обработчике int 8h, префиксная команда lock**

(из 1 части 1 лабы)

В самом начале обработчика прерывания int8h и после вызова прерывания 1CH: вызывается sub\_2 для запрета маскируемых прерываний

; Вызов подпрограммы sub\_2 (запрет прерываний)

020A:0746 E8 0070                           call sub\_2                           ; (07B9)

Подпрограмма sub\_2: (жирным выделены 2 способа запретить маскируемые прерывания)

```
sub_2 proc near
; Сохранение значений регистров DS, AX
020A:07B9 1E                               push ds
020A:07BA 50                               push ax
; Инициализация DS значением 0040h (адресом начала области данных BIOS)
020A:07BB B8 0040                        mov ax, 40h
020A:07BE 8E D8                            mov ds, ax
; Загрузка младшего байта FLAGS в регистр AH
020A:07C0 9F                               lahf                               ; Load ah from flags
; Проверка: поднят ли хотя бы один из флагов 10 или 13
; (2400h = 0010 0100 0000 0000b) в области BIOS по адресу 0040:0314h,
где
; находится копия флагов
; 10 - DF - Флаг направления, контролирует поведение команд обработки строк: 1 - ; в сторону уменьшения адресов, 0 - наоборот
; 12 и 13 - IOPL - Уровень приоритета ввода/вывода.
020A:07C1 F7 06 0314 2400 test word ptr ds:[314h], 2400h   ; (0040:0314=3200h)
; 1) Если поднят хотя бы один, то переход на loc_22, чтобы командой cli сбросить
; флаг разрешения прерываний IF.
; Процессор перестанет обрабатывать прерывания от внешних устройств
; (только маскируемые, так как они вызываются по маске. Немаскируемые запретить
```

```

; нельзя, например, различные ошибки)
020A:07C7  75 0C          jnz    loc_22                      ; Jump if not
zero
; 2) Если оба сброшены, то сброс IF (9 бит) командой and.
; Операция and объёмная, 2 раза обращается к памяти: считывает
значение по адресу
; 0040:0314, затем изменяет его и еще раз обращается к памяти на
запись.
; Необходимо, чтобы в промежуток, когда выполняется сама логическая
операция,
; никто не обращался к этому участку памяти, для чего используется
 префиксная
; команда lock. Будет заблокирована шина данных, и, если в системе
присутствует
; другой процессор, он не сможет обращаться к памяти, пока не
закончится
; выполнение and
020A:07C9  F0> 81 26 0314 FDFF      lock and      word ptr
ds:[314h],0FDFFh      ; (0040:0314=3200h)
020A:07D0      loc_21:
; Восстановление значений флагов SF, ZF, AF, PF и CF регистра FLAGS из
AH
020A:07D0  9E              sahf                  ; Store ah into flags
; Восстановление значений регистров AX, DS
020A:07D1  58              pop     ax
020A:07D2  1F              pop     ds
; Переход на loc_23
020A:07D3  EB 03          jmp     short loc_23          ; (07D8)
020A:07D5      loc_22:
; Сброс IF
020A:07D5  FA              cli                  ; Disable interrupts
020A:07D6  EB F8          jmp     short loc_21          ; (07D0)
; Выход из подпрограммы
020A:07D8      loc_23:
020A:07D8  C3              retn
                           sub_2      endp

```

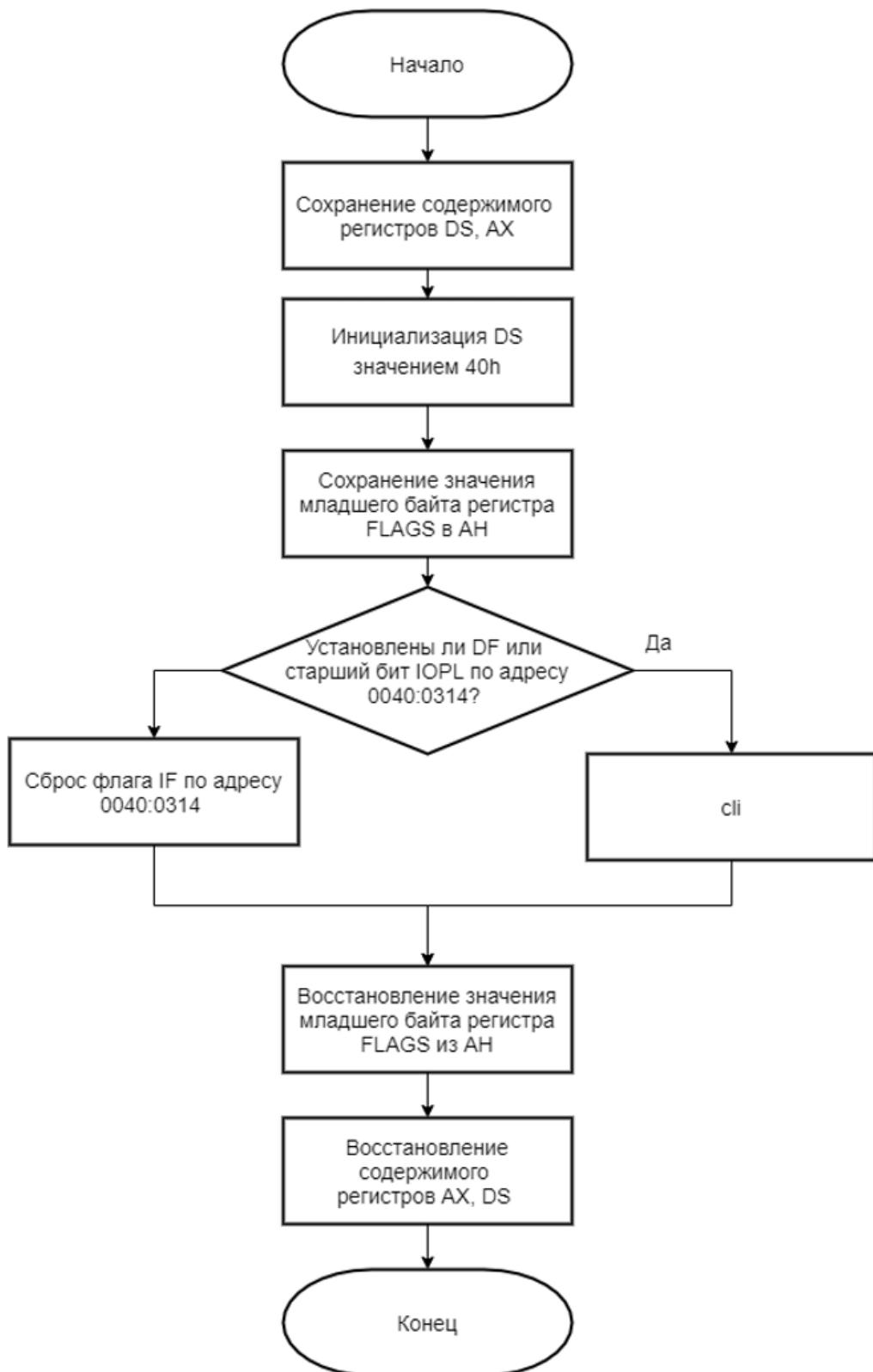
В конце обработчика int8h (уже после вызова 1Ch и повторного запрета прерываний):

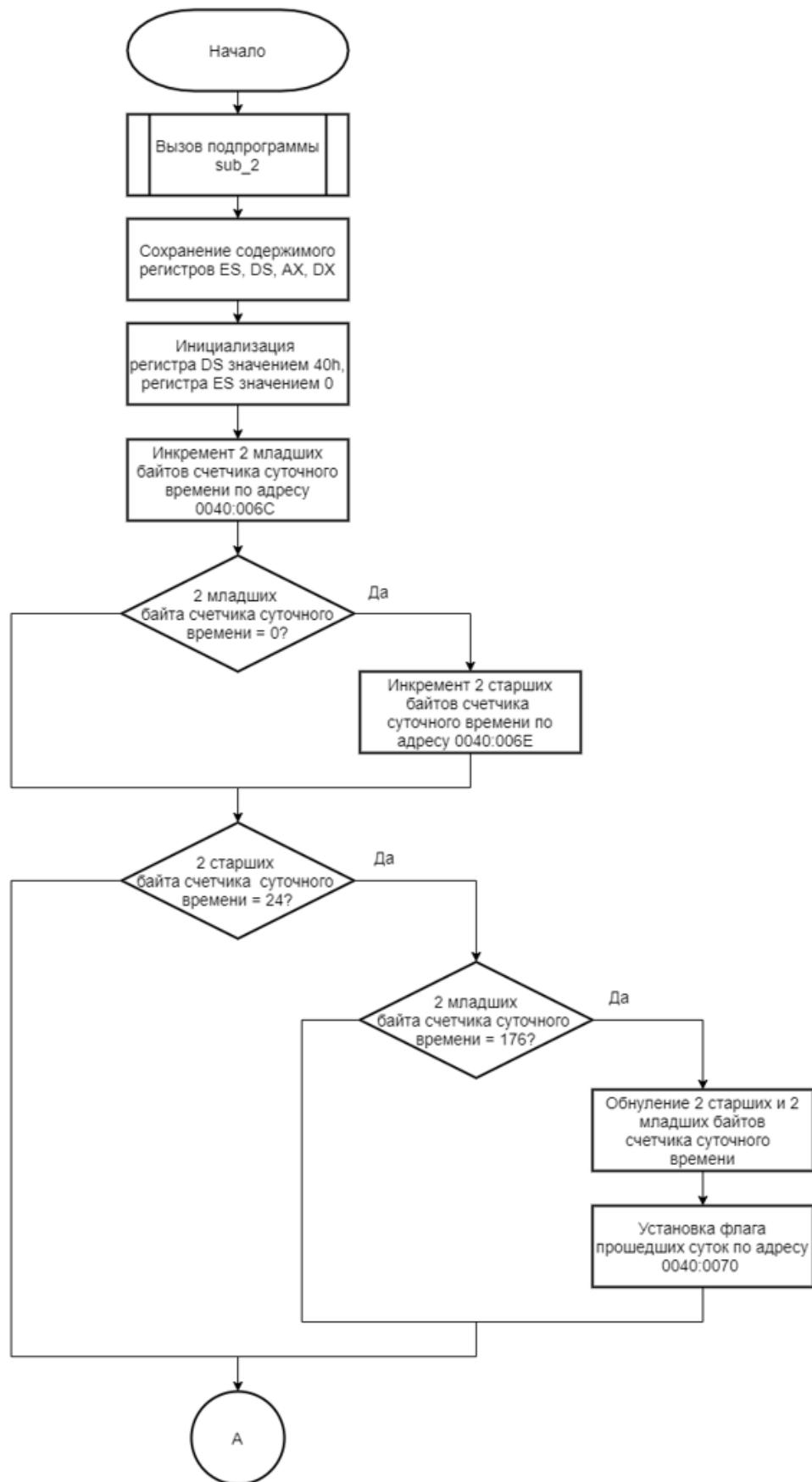
; Сброс контроллера прерываний  
; (Чтобы позволить прерываниям меньшего приоритета обрабатываться)

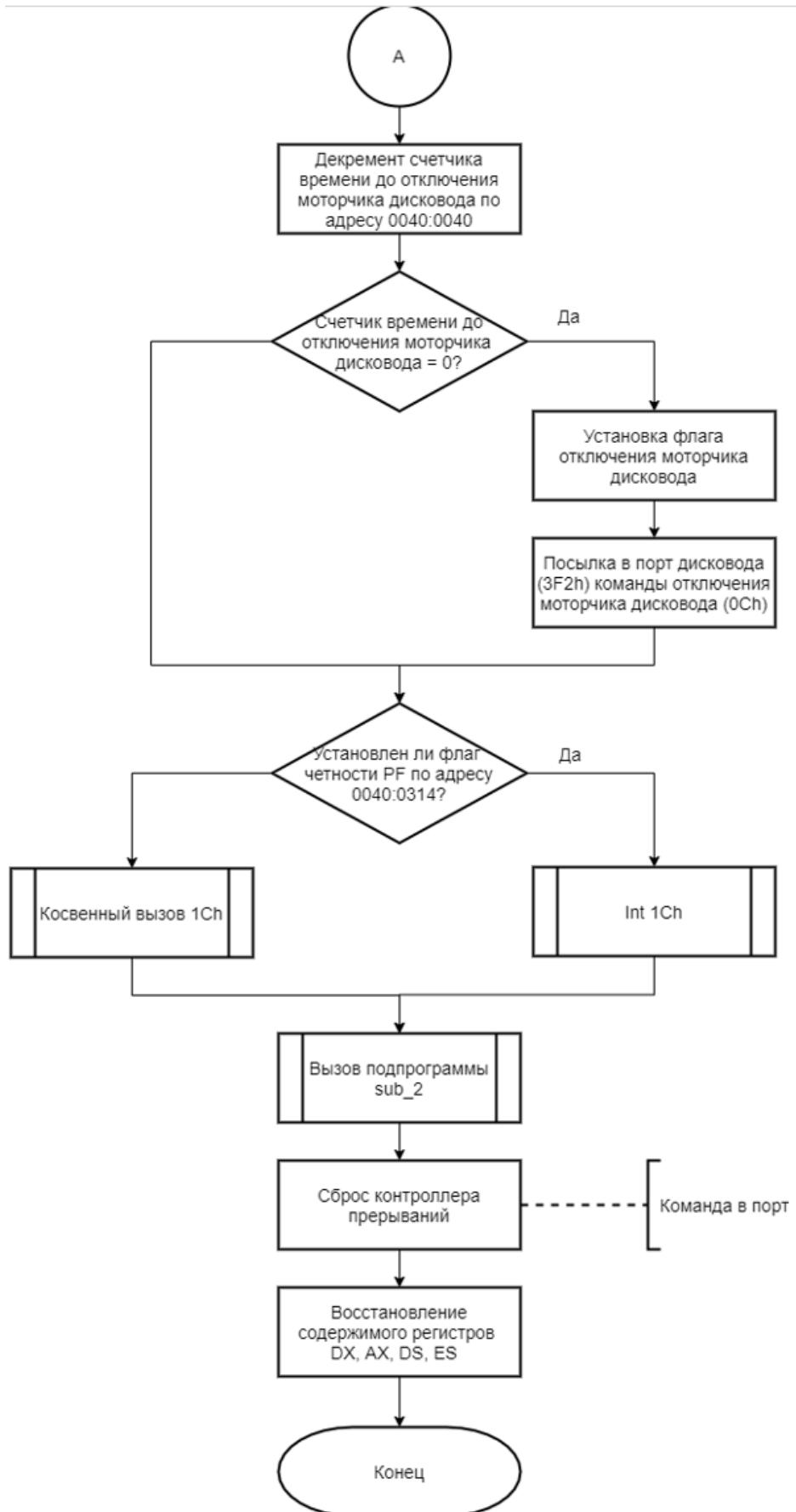
```
020A:07A8 B0 20      mov  al,20h      ; ''
020A:07AA E6 20      out  20h,al     ; port 20h, 8259-1 int command
                      ; al = 20h, end of interrupt
```

### **Схемы на всякий**

sub\_2







**Пример кода обработчика прерывания от системного таймера из лабораторной работы по защищенному режиму (особенности).**

Особенность - сброс контроллера прерываний

```
new_int08 proc
    push eax
    mov edi, cursor_pos ; поместим в edi позицию для вывода

    mov ah, param_8 ; В ah помещаем цвет текста.
    xor ah, 1         ; Сдвигаем циклически вправо параметр (он примет какое-то новое значение)
    mov param_8, ah
    mov al, cursor_symb ; Символ, который мы хотим вывести
    stosw ; al (символ) с параметром (ah) перемещается в область памяти es:di

    ; используется только в аппаратных прерываниях для корректного завершения
    ; (разрешаем обработку прерываний с меньшим приоритетом) !!
    ; необходимо сбросить контроллер прерываний
    mov al, 20h
    out 20h, al

    pop eax
    iretd ; double - 32 битный iret
new_int08 endp
```

**P.S. Также касаемо прерываний в лабе по защищенному режиму**

IDT – таблица защищенного режима и называется «Таблица дескрипторов прерываний» – это не то же самое, что таблица векторов прерываний.

Первые 32 дескриптора отведены под исключения, с 32 по 255 – дескрипторы, которые может определить пользователь. Нам надо адресовать 2 аппаратных прерывания – от клавиатуры и системного таймера. Для адресации нужен вектор прерывания=базовый вектор + номер irq. Если оставим старый базовый вектор – попадем на double fault. Поэтому мы должны перепрограммировать базовый вектор ведущего контроллера на новый вектор = 32.

При этом больше никаких прерываний не обрабатываем. На ведущий контроллер попадают только 2 импульса – клава и таймер. На ведомый – вообще не приходят, поэтому надо замаскировать все прерывания на ведомом контроллере. При этом 1 – запрет прерывания, 0 – разрешение. FF-на ведомый, на ведущий – FC.

Процессор обращается отдельно к ведущему и ведомому контроллерам через порты – будут использоваться in и out. (в таймере был 20h), контроллер ведущий 21h, ведомый A1h

*Master mask и slave mask – byte (так как 8 входов)*

**сохранение значений масок для восстановления при возвращении в реальный режим**

*mask\_master db 0*

*mask\_slave db 0*

*in al, 21h*

*mov mask\_master, al*

*in al, 0A1h*

*mov mask\_slave, al*

*запрет всех прерываний, кроме прерываний от таймера (0) и клавиатуры (1) в ведущем контроллере, запрет всех прерываний в ведомом контроллере*

*mov al, 0FCh*

*out 21h, al*

*mov al, OFFh*

*out 0A1h, al*

*восстановление масок*

*mov al, mask\_master*

*out 21h, al*

*mov al, mask\_slave*

*out 0A1h, al*

*Важнейшее условие для корректного выполнения указанных действий - запрет всех прерываний: маскируемых и немаскируемых!*

*Маскируемые – cli, немаскируемые – 70 порт, послать команду 80h*

*Cli*

*Mov al, 80h*

*Out 70h, al*

*Разрешение маскируемых и немаскируемых прерываний (NMI и MI)*

*Sti*

*Xor al, al*

*out 70h, al*

**Перепрограммирование контроллеров**

*На ведомом все замаскировано – достаточно перепрограммировать ведущий.  
Есть версии, где для ведомого устанавливают тот же, что и для ведущего.  
Есть, где вообще ничего*

*В порт контроллера посыпаются несколько команд, которые называются «слово команды исполнения»*

*перепрограммирование ведущего контроллера*

```
mov al, 11h ; СКИ1  
out 20h, al  
  
mov al, base_vec ; СКИ2  
out 21h, al  
  
mov al, 4 ; СКИ3 – IRQ2  
out 21h, al  
  
mov al, 1 ; СКИ4 – требует EOI  
out 21h, al
```

## 11 билет

11.1 Параллельные процессы: взаимодействие, обоснование необходимости монопольного доступа к разделяемым переменным, способы взаимоисключения. Мониторы: определение; примеры – простой монитор и монитор кольцевой буфер. (в 2021 просто формулировка другая, смысл и суть та же) (в 2022 также)

**Параллельные процессы: взаимодействие, обоснование необходимости монопольного доступа к разделяемым переменным, способы взаимоисключения.**

(до блока семафоры: определение)

**Мониторы: определения**

*Внимание: мониторы сами являются ресурсами! (говорилось в контексте того, что они защищают некоторую переменную-ресурс)*

Монитор (monitor) – это программное средство, разработанное Хоаром и дающее возможность управляемого совместного использование ресурсов среди асинхронных процессов, включая возможность управляемого обмена параметрами между процессами.

Монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации динамического распределения конкретного

ресурса. Монитор защищает свои переменные. Доступ к переменным монитора можно получить, только используя процедуры монитора.

Монитор осуществляет доступ к разделяемым ресурсам посредством использования переменных типа условие (conditional). Для каждой отдельно взятой причины, по которой процесс переводится в состояние ожидания (блокировки), назначается своя переменная типа условие.

На переменных «условие» определены два типа операций: `wait(c)` и `signal(c)`, которые по сути являются макрокомандами.

- Команда `wait(c)` блокирует процесс, если ресурс занят, и открывает доступ к монитору, если ресурс свободен.
- Выполнение заблокированного процесса активизируется командой `signal(c)`, которую вызывает другой процесс. Оператор `signal(c)` выполняется следующим образом: если очередь к переменной «условие» не пуста, то из очереди выбирается один из процессов и активизируется, иначе если очередь пуста, то `signal(c)` действий не выполняет

### простой монитор

Простой монитор решает задачу выделения одиночного ресурса нескольким процессам

```
RESOURCE MONITOR;
```

```
var
```

```
    busy: logical;
```

```
    X: conditional;
```

```
procedure acquire; // процедура «захватить»  
begin  
    if busy then wait(X);  
    busy = true;  
end;  
procedure release; //процедура «осуществить»  
begin  
    busy = false;  
    signal(X);  
end;  
begin  
    busy = false;  
end.
```

Монитор обслуживает произвольное число процессов, ограниченное только длиной очереди. В мониторе две процедуры: `acquire` и `release`. Если процесс нуждается в захвате ресурса, он вызывает процедуру `acquire`. Если логическая переменная `busy` – ложь (`false`), то процесс без задержки устанавливает переменную `busy` в значение истина (`true`). Если же логическая переменная `busy` – истина, то по переменной условие `X` выполняется `wait(X)` и логическая переменная `busy` не меняется. Для освобождения ресурса процесс

вызывает процедуру *release*. Переменной *busy* присваивается значение ложь (*false*) и *signal(X)* проверяет список процессов, в очереди к переменной *X*, выбирает из очереди процесс и активизирует его.

### монитор кольцевой буфер

Монитор «кольцевой буфер» решает уже рассмотренную ранее задачу «производство-потребление». Буфер – это массив заданного размера. Производитель помещает в массив данные. Потребитель считывает эти данные в порядке, в котором они были помещены в буфер. Производитель последовательно заполняет элементы массива и наступит момент, когда последний элемент буфера будет заполнен. Но массив организован как кольцевой буфер. Когда заполняется последний элемент массива, происходит переход снова на первый элемент.

```
RESOURCE MONITOR;
var
    ring_buffer: array [0..n-1] of <type>;
        pos: 0..n; //текущая позиция
            j: 0..n-1; // заполняемая позиция
            k: 0..n-1; // освобождаемая позиция
            buffer_full, buffer_empty: conditional;
procedure producer (data: type)
begin
    if pos == n then wait(buffer_empty);
    ring_buffer[j] = data;
    pos++; // инкремент
    j = (j+1) mod n; // “заворачивание” позиции заполнения
    signal(buffer_full);
end;
procedure consumer (var data: type)
begin
    if pos==0 then wait(buffer_full);
```

```
    data = ring_buffer[k];
    pos--;
    k = (k+1) mod n;
    signal(buffer_empty);
end;
begin
    pos = 0;
    j = 0;
    k = 0;
end.
```

Условие *buffer\_full* является признаком, который ждет процесс потребитель, если обнаружит, что буфер пуст. Этот признак устанавливает процесс производитель

*сигналом – `signal(buffer_full)`, когда он поместит данные в буфер. Производитель, наоборот, ждет события `buffer_empty` (буфер пуст).*

[Задача «читатели-писатели». Монитор Хоара](#)

На всякий

### [Сравнение мьютексов \(М\) и семафоров \(С\).](#)

[11.2 Средства межпроцессорного взаимодействия \(IPC\) операционной системы UNIX System V: очереди сообщений и программные каналы, сигналы – сравнение, примеры \(для программных каналов пример из лабораторной работы с сигналами\).](#)

## **Сигналы**

Механизм сигналов позволяет процессам реагировать на события, которые могут происходить внутри процесса или вне его.

Как правило, получение некоторым процессом сигнала указывает ему на необходимость завершить работу. Вместе с тем реакция процесса на сигнал зависит от того как сам процесс определяет свое поведение в случае приема к-либо определенного сигнала.

Процесс может реагировать на сигнал стандартным образом, то есть в соответствии с существующим в системе обработчиком сигнала, может игнорировать сигнал или вызвать на выполнение свой обработчик сигнала.

Начиная с классического юникс сигналы имеют числовой идентификатор, а также мнемоническую (мнемоника – форма представления, удобная для запоминания) форму записи идентификатора сигнала, которые хранятся в библиотеке `signal.h`.

В классическом юникс было определено [20 сигналов](#) (Define NSIGN 20). В современных их больше (каждая строка начинается с `#define`):

- `#define NSIG 20`
- `SIGHUP 1` – разрыв связи с терминалом
- `SIGINT 2` – классика (из юникс в dos) – сигнал завершения программы `ctrl+C`
- `SIGQUIT 3` – `ctrl+/_`
- `SIGKILL 9`
- `SIGSEGV 11` – нарушение сегментации (выход за пределы сегмента)
- `SIGPIPE 13` - запись в канал есть, чтения нет – недопустимая операция с каналом
- `SIGALARM 14` – прерывание от таймера
- `SIGTERM 15` – команда `kill`, которая выполняется в командном режиме
- `SIGUSR1 16`
- `SIGUSR2 17`
- `SIGCHLD` – сигнал, который получает предок при завершении потомка

## Макросы

- `#define SIG_DFL(int(*)()) 0`

*Используется с сигнальной функцией на месте указателя на обработчик прерывания для выбора стандартного обработчика прерывания операционной системы*

- `#define SIG_IGN(int(*)()) 1`

*Используется с сигнальной функцией на месте указателя на процедуру обработчика прерывания для игнорирования конкретного сигнала.*

*В интернетах еще есть 3 - SIG\_ERR - Возвращается сигнальной функцией на месте указателя на обработчик прерывания, для сообщения о том, что данный обработчик прерывания не может быть использован по каким-либо причинам.*

Средством посылки и приема сигналов в ОС unix служат 2 системных вызова - `signal()` и `kill()`

### 1. kill

`Int kill(int pid, int sig)`

//вызов

`Kill(pid, sig)`

Сигнал `sig` будет послан сигналу с идентификатором `pid` и всем процессам-родственникам.

В юникс помимо иерархии важны группы процессов. Процессы объединяются в группы. Например, в первом параметре указывается значение `pid <= 1` – сигнал посыпается группе процессов, `=0` – всем процессам с идентификаторами группы, совпадающим с идентификатором группы процесса вызвавшего `kill` (то есть `pid` – не всегда идентификатор)

Примеры

- `Kill(37, SIGKILL)` - Процессу с идентификатором 37 безусловно завершиться
- `Kill(getpid(), SIGALARM)` – `getpid()` получает собственный идентификатор, сам процесс, вызвавший `kill`, получит сигнал пробудки.

Системный вызов `signal` не является стандартным для POSIX 1, но он определен в ANSI C и, следовательно, имеется во всех UNIX. `.Signal` не входит в POSIX, поэтому его использование не рекомендуется, так как его поведение в system 5 отличается от поведения в bsd.

*POSIX - portable operating system interface – интерфейс переносимых ОС*

*POSIX 1 FIPS – federal information processing standard – федеральный стандарт, разработанный национальным институтом ... США. Есть POSIX 2. POSIX 1 последний был создан – 1988*

*Для создания переносимого ПО европейцы создали X/open portability gui – 2 варианта Xpg3 и Xpg4. Основан на ANSI C, POSIX 1, POSIX 2 и содержит дополнительные конструкции.*

## 2. signal

Системный вызов signal возвращает указатель на предыдущий обработчик данного сигнала и его можно использовать для восстановления предыдущего обработчика. Еще можно восстановить DFL.

```
#include <signal.h>

Int main()
{
    Void(*old_handler)(int) = signal(SIGINT, SIG_IGN);

    /*действия*/

    Signal(SIGINT, old_heandler);

    Return 0;
}
```

Еще системных вызовов

- Int sigaction(int sig\_num, struct sigaction \*action, Struct sigaction \*old\_action)  
(входит в POSIX )

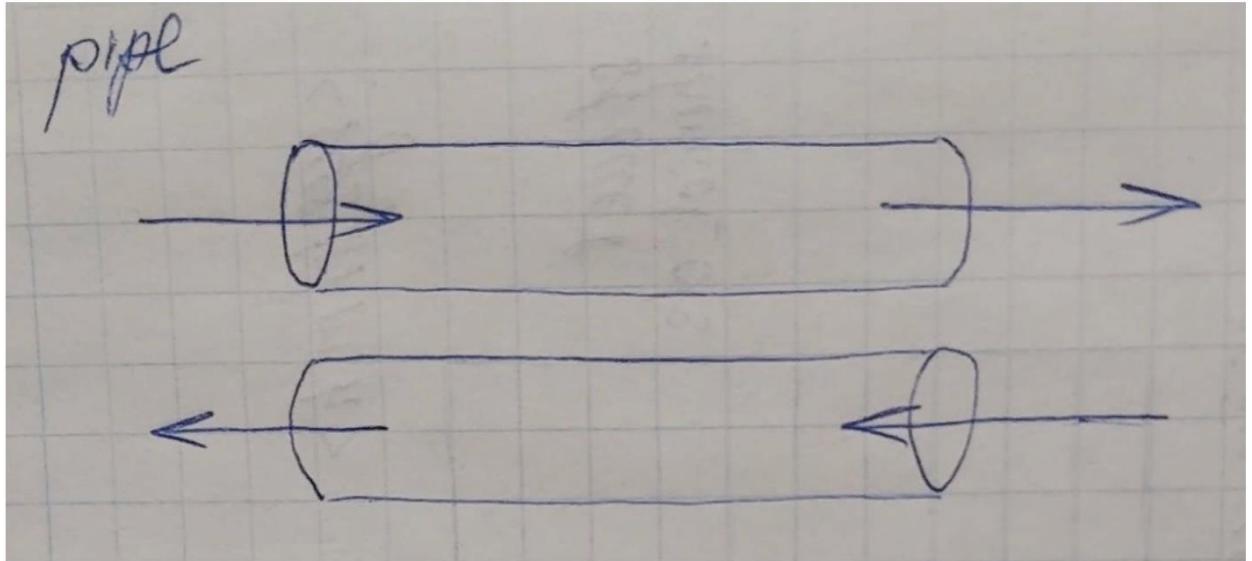
Struct sigaction определена в библиотеке signal.h

*Системный вызов sigaction используется для изменения действий процесса при получении соответствующего сигнала. Параметр signum задает номер сигнала и может быть равен любому номеру, кроме SIGKILL и SIGSTOP. Если параметр act не равен нулю, то новое действие, связанное с сигналом signum, устанавливается соответственно act. Если oldact не равен нулю, то предыдущее действие записывается в oldact.*

программисты могут использовать сигналы для изменения хода выполнения программы. В POSIX для этого определены sigsetjmp() и siglongjmp().

- Sigsetjmp – отмечает одну или несколько позиций в программе.
- Longjmp – осуществляет переход на одну из выделенных позиций

## Программные каналы (pipe, труба)



Программный канал - это специальный файл, в который можно «писать» информацию и из которого эту информацию можно «читать». Причем порядок записи информации и последующего чтения – FIFO (очередь)

Каналы - это потоковая модель передачи данных. Симплексная связь, односторонняя. Для двусторонней – дуплексной, необходимо как минимум 2 трубы.

В **отличие** от разделяемой памяти, для которой в программе есть таблица разделяемых сегментов, программные каналы поддерживаются файловой подсистемой. То есть программные каналы имеют дескриптор файла

Изначально были созданы неименованные программные каналы, потом именованные.

### Именованные

- имеют имя и inode.
- Создаются командой mknod (в командной строке писали mknod <имя> p).

```
int mknod(const char *path, mode_t mode, dev_t dev);
```

Mknod(<имя>, S\_IFIFO | ACCESS, 0) – это можно сказать фактические параметры (а не формальные, они пишутся с типами)

- Любой процесс, который знает идентификатор именованного программного канала, может работать с этим программным каналом. **Являются в системе специальными файлами и видны в файловой системе, более того - они имеют идентификатор в файловой системе.**

### Не именованные

- имеют только inode.
- создаются системным вызовом pipe, который возвращает файловый дескриптор, если удалось создать канал.

```
(int pipe(int pipefd[2]););
```

Создание канала:

```
int fd[2];
```

```
pipe(fd);
```

- Неименованный имеет только дескриптор (не имеет идентификатора), поэтому пользоваться неименованным могут только родственники, потому что процессы-потомки в результате fork наследуют от предка дескрипторы открытых файлов.

Программные каналы имеют встроенные средства взаимоисключения, то есть в канал нельзя писать, если читают, и нельзя читать, если в него пишут. Для этого определяли fd[2] – массив дескрипторов. Закрыть для записи и читать и наоборот.

АП процессов защищенное, поэтому программные каналы могут создаваться только в АП ядра системы. Программный канал – буфер в системной области памяти. (последовательность адресов)

Программный канал (pipe) буферизуется на 3 уровнях (системная область памяти, диск, время). Ограничение размера канала основано на повышении эффективности, так как при этом не используется обращение к медленной внешней памяти.

1. На 1 уровне трубы буферизуются в системной памяти (=в области данных ядра системы). Обычно не может превышать 4096 байт (1 страница).
2. При переполнении системной памяти программные каналы (буфера), имеющие наибольшее время существования, переписываются во вторичную память (диск) с помощью стандартных функций управления или работы с файлами.
3. Если процесс пытается записать в трубу больше 4096 байт, то труба буферизуется во времени, приостанавливая процесс до тех пор, пока все данные из нее не будут прочитаны. Другими словами подсистема ввода-вывод обеспечивает приостановку процесса, если канал заполнен.

Команда write будет успешно выполнена, если есть доступное место, в противном случае ожидание. Команда read процесс будет блокирован если канал пуст.

Еще раз – **разница** между каналами и разделяемой памятью. Канал обеспечивает потоковую модель передачи данных. При считывании сообщения оно перестает существовать. Он имеет встроенные средства взаимоисключения. Разделяемая

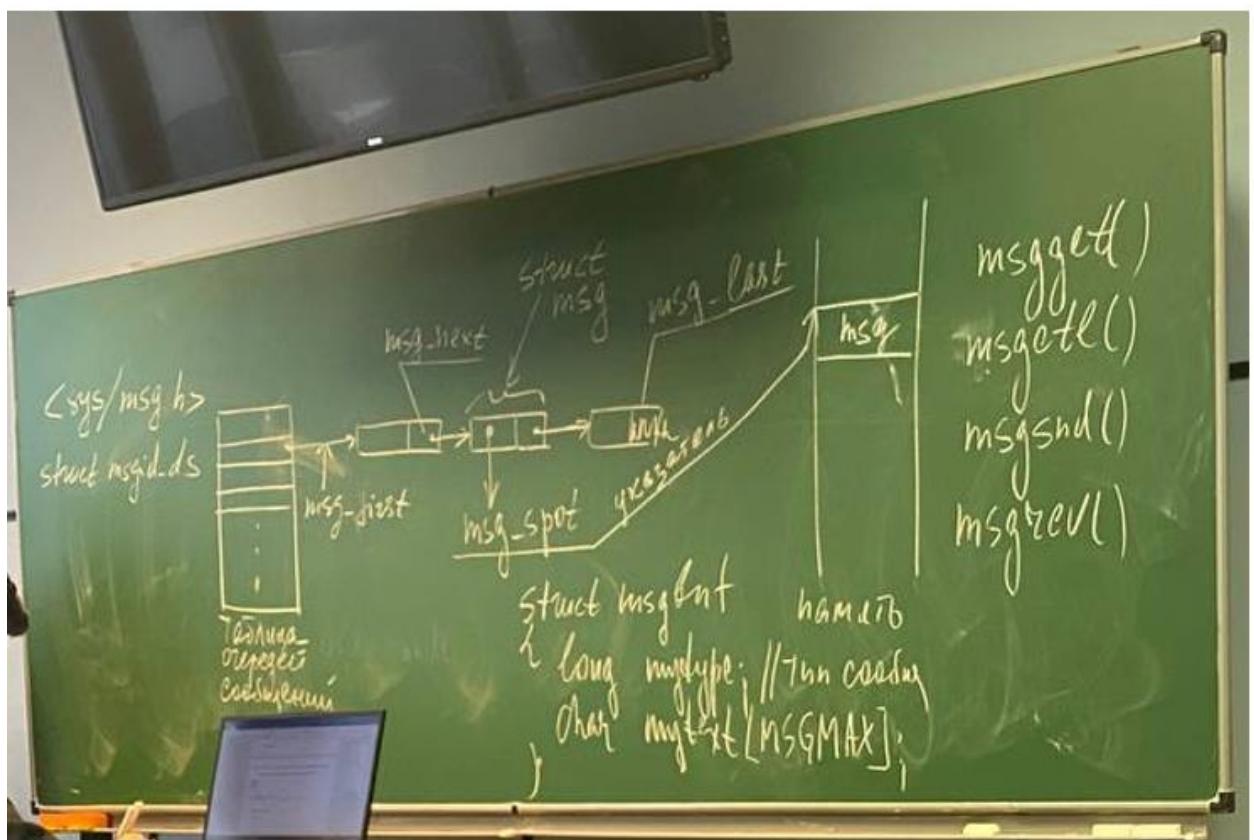
память не имеет. Это буфер, подключающийся к АП, нет средств взаимоисключений, поэтому их обычно используют с семафорами. Что положили – так и будет там лежать

### Очереди сообщений

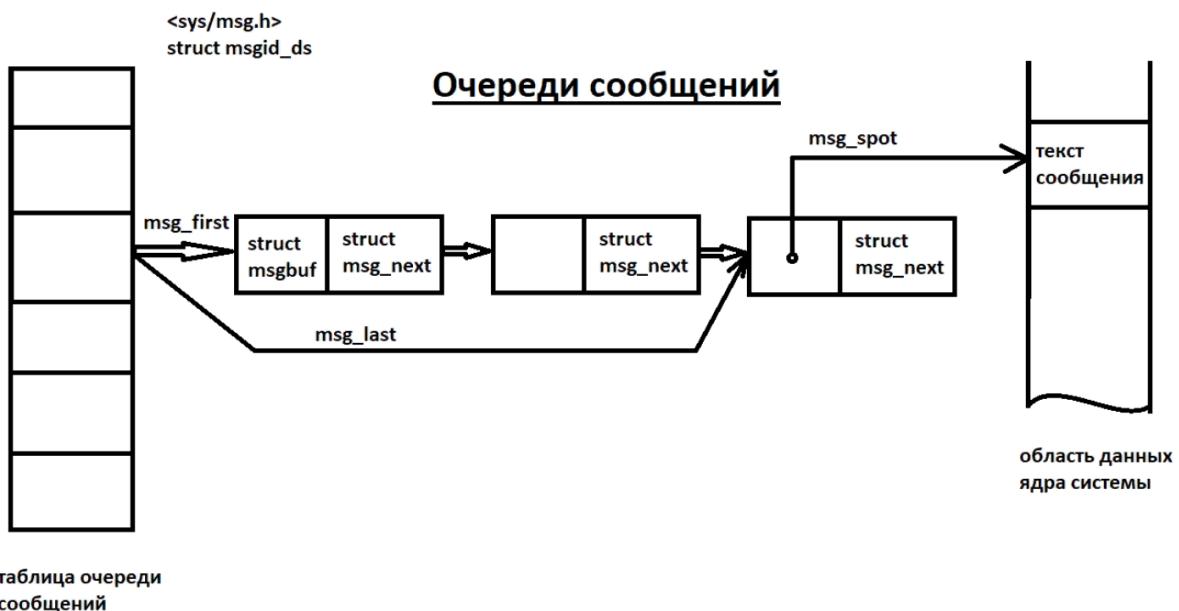
В распределенных системах вся передача данных - с помощью сообщений.

Очереди сообщений - средства взаимодействия на отдельной машине, характерны и для распределенных системах.

В ядре системы существует таблица очередей сообщений. Таблица очередей сообщений (системная таблица) содержит дескрипторы всех очередей сообщений в системе. Каждый элемент таблицы (очереди) имеет указатель на следующее сообщение, последний – NULL. Связный список. *FIFO*??



(на рисунке ниже много неточностей)



<sys/msg.h>: Struct msqid\_ds

Каждый элемент описывает struct msg. В самом элементе – msg spot – указатель на выделенную область памяти, в которой находится сообщение.

На очередях сообщений определены системные вызовы:

- Msgget() **int msgget(key\_t key, int msgflg);** – возвращает идентификатор очереди сообщений, связанный со значением параметра *key*
- Msgctl() **int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);** – Эта функция выполняет контрольную операцию, заданную в *cmd*, над очередью сообщений *msqid*.
- Mgsnd() – положить сообщение
- Msgrev() – прочитать сообщение

Определена структура

struct msg\_buf

{

    Long mytype; //тип сообщения, позволяет разделять сообщения от клиента к серверу и от сервера к клиенту.

```
    Char mytext[MSGMAX];
```

}

## ВАЖНЫЕ ОСОБЕННОСТИ

Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает ее в конец связного списка записей соответствующих сообщений указанной очереди.

В каждой такой записи указывается тип сообщения, длина сообщения в байтах и указатель на область данных ядра системы, в которую копируется сообщение и в которой оно будет фактически находиться.

Ядро копирует сообщение из пространства процесса-отправителя в область данных ядра системы, чтобы процесс-отправитель мог завершиться. При этом сообщение остается доступным для чтения другими процессами.

Когда какой-то процесс выбирает сообщение из очереди, ядро копирует это сообщение в АП этого процесса. После этого сообщение удаляется (перестает существовать). Процесс может выбрать сообщения из очереди следующими способами:

1. взять самое старое сообщение, независимо от его типа
2. взять сообщение, если идентификатор сообщения совпадает с идентификатором, который указал процесс. Если существует несколько сообщений с таким идентификатором, то взять самое старое из них
3. сообщение, числовое значение типа которого является наименьшим из меньших или равным значению типа, указанного процессом. Если таких несколько – то самое старое

При использовании очередей сообщений процессы могут не блокироваться в ожидании сообщения и при отправке, и при получении. (вспомнить диаграмму [3 состояния блокировки процесса при передаче сообщений](#))

Пример

```
#ifndef MSGMAX  
  
#define MSGMAX 1024  
  
#endig  
  
Struct mbuf  
  
{  
  
    Long mtype;  
  
    Char mtext[MSGMAX];  
  
    } mobj = {15, "Hello"};  
  
Int main()  
  
{
```

```

Int fd = msgget(100, IPC_CREATE|I{C_EXECL|0642);

If (fd == -1 || msgsnd(fd, mobj, strlen(mobj.mtext) + 1, IPC_NOWAIT))

    Perror("message");

Return 0;

}

```

Создается новая очередь сообщений с идентификатором 100, флаги: 6 – чтение и запись для владельца, 4 – только чтение для группы и 2 – только запись для остальных. Если нам удалось создать очередь, в эту очередь отправляется сообщение Hello с типом 15, при этом указывается, что вызов не блокирующий – NO\_WAIT.

**В отличие** от разделяемых сегментов, здесь копирование. При посылке – из буфера программы в область данных ядра. При чтении – из области данных ядра в буфер программы.

Если нужно прочитать сообщение, есть гибкие возможности – не будет блокирован при чтении. То есть очереди сообщений позволяют избежать лишних блокировок. Процесс, отправивший сообщение, если не заинтересован в ответе, может выполняться дальше или быть завершен.

На очередях сообщений определено много флагов. IPC\_EXCL – прочитать самостоятельно

*читаем: Вызов msgget() с IPC\_CREAT, но без IPC\_EXCL всегда выдает идентификатор (существующей с таким ключом или созданной) очереди. Использование IPC\_EXCL вместе с IPC\_CREAT либо создает новую очередь, либо, если очередь уже существует, заканчивается неудачей. Самостоятельно IPC\_EXCL бесполезен, но вместе с IPC\_CREAT он дает гарантию, что ни одна из существующих очередей не открывается для доступа.*

**примеры (для программных каналов пример из лабораторной работы с сигналами).**

(Примеры для остальных есть по ходу изложения)

### Пример с лабы

### **сравнение сигналы-ПК-ОС-РС**

(Доп сравнение - с разделяемой памятью - чтобы все в одном месте было. После сравнения - подробнее про разделяемую память)

Просто текст от рязановой

- Разница между каналами и разделяемой памятью. В отличие от разделяемой памяти, для которой в программе есть таблица разделяемых сегментов, программные каналы поддерживаются файловой подсистемой. То есть программные каналы имеют дескриптор файла

Канал обеспечивает потоковую модель передачи данных. При считывании сообщения оно перестает существовать. Он имеет встроенные средства взаимоисключения. Разделяемая память не имеет. Это буфер, подключающийся к АП, нет средств взаимоисключений, поэтому их обычно используют с семафорами. Что положили – так и будет там лежать

- Очередь: В отличие от разделяемых сегментов, здесь копирование. При посылке – из буфера программы в область данных ядра. При чтении – из области данных ядра в буфер программы.

(далее - авторское, ни за что не ручаюсь)

1. Для всех, кроме разделяемых сегментов, можно сказать правило - получил, и оно перестало существовать. Все через область памяти ядра (хотя насчет сигналов хз). Удаление, если создавший завершился - нет.
2. Как идентифицируются
  - сигналы имеют числовой идентификатор, а также мнемоническую форму записи идентификатора сигнала, которые хранятся в библиотеке signal.h.
  - Программные каналы - именованные имеют имя (=идентификатор) и inode, не именованные имеют только inode (inode = видимо дескриптор файла).
  - Очередь сообщений - каждый элемент очереди описывается struct msg, в самом элементе – msg spot – указатель на выделенную область памяти, в которой находится сообщение (ну и указатель на следующий элемент). Само сообщение содержит тип и текст: struct msg\_buf{Long mytype; Char mytext[MSGMAX];}
3. Получение/отправка
  - Сигнал. Средством посылки и приема сигналов в ОС unix служат 2 системных вызова - signal() и kill(). Как правило, получение некоторым процессом сигнала указывает ему на необходимость завершить работу. Вместе с тем реакция процесса на сигнал зависит от того как сам процесс определяет свое поведение в случае приема к-либо определенного сигнала. Можно сказать - отправляет как хочет, а реагирует сразу
  - Программные каналы. Команда write будет успешно выполнена, если есть доступное место, в противном случае ожидание. Команда read процесс будет блокирован если канал пуст. Пихается в канал (буфер в области данных ядра системы)
  - Очередь сообщений. Msgsnd() – положить сообщение, Msgrev() – прочитать сообщение. Происходит копирование - из АП отправителя в область данных ядра системы и из области данных ядра системы в АП получателя. При использовании очередей сообщений процессы могут не блокироваться в ожидании сообщения и при отправке, и при получении.

- Разделяемые сегменты. Shmget();Shmctl();Shmat() attach - подключить, Shmdt() detach – отключить. После создания разделяемого сегмента любой процесс может присоединить его к своему виртуальному АП, используя команду shmat и работать с ним как с сегментами собственного АП.
4. что надо знать/кто может
- сигнал - чтобы отправить сигнал процессу, надо знать его идентификатор
  - программный канал - именованный - достаточно знать его имя(идентификатор), не именованный - имеет только дескриптор (не имеет идентификатора), поэтому пользоваться неименованным могут только родственники, потому что процессы-потомки в результате fork наследуют от предка дескрипторы открытых файлов.
  - Разделяемые сегменты. нужна ссылка на сегмент. Shmget();Shmctl();Shmat() attach - подключить, Shmdt() detach – отключить. После создания разделяемого сегмента любой процесс может присоединить его к своему виртуальному АП, используя команду shmat и работать с ним как с сегментами собственного АП.

#### 5. “Порядок” получения

- Сигнал. Ну получил и получил
  - Программный канал - строго FIFO
  - Очередь сообщений - тремя способами, но там вроде тоже наподобие FIFO
  - как с сегментами собственного АП.
- 

6. Встроенные средства взаимоисключения - в канале есть, в разделяемых сегментах нет.  
 7. Копирование - в очереди есть, в разделяемых сегментах - нет

НУ РАЗНЫЕ ОНИ, лучше перечитать описание каждого

#### Сегменты разделяемой памяти

Разделяемая память-средство, куда один процесс может записать сообщение, а другой процесс – считать. В ядре имеется таблица сегментов разделяемой памяти (таблица разделяемых сегментов) и структура

Struct Shmid\_ds в <sys/shm.h>

Каждая строка таблицы описывает 1 разделяемый сегмент. Он создается в области памяти ядра системы, потому что адресные пространства процессов являются защищенными – ни один процесс не может обращаться в АП другого процесса, поэтому взд возможно только через АП ядра

Средство разделяемые сегменты устроено так, чтобы не выполнялось лишнего копирования, сегменты подключаются к виртуальному АП процесса (получают ссылку на разделяемый сегмент).

На разделяемых сегментах определены системные вызовы

- Shmget();
- Shmctl();
- Shmat() attach - подключить
- Shmdt() detach – отключить

После создания разделяемого сегмента любой процесс может присоединить его к своему виртуальному АП, используя команду shmat и работать с ним как с сегментами собственного АП. По завершении процесса разделяемый сегмент сохраняется. То есть РС не удаляются даже если завершаются процессы, которые их создали. У каждого РС есть назначенный владелец и удалять эту область из ядра или корректировать ее управляющие параметры могут процессы, которые имеют привилегированные права, создателя или назначенного владельца.

Пример

```
#Types, ptc, shm, string
Int main()
{
    Int perms = S_IRWXV|S_IRWXG|S_IRWXO;
    Создается разд с с ид 100 размером 1024 б с полными правами для всех
    категорий пользователей
    Int fd = shmget(100, 1024, IPC_CREATE|perms);
    If (fd == -1)
    {
        Perror("shmset"); exit(1);
    }
    Если создан, процесс пытается подключить РС к своему АП
    Char *Addr = (shar *) shmat(fd, 0, 0);
    If (addr == (char *)-1)
    {
        Perror("shmat");
        Exit(1);
    }
    И записывает hello
    Strcpy(addr, "Hello");
    Затем отсоединяется от этого сегмента
    If (shmdt(addr) == -1)
        Perror("shmdt")
    Return 0;
}
Рассмотрим формальные параметры
Voi * shmat(int shmid, const void *shmaddr, int shmflg)
```

Тип данных не важен - **void \***  
Shmid - правильней конечно fd  
Shmaddr=0 - первый полученный адрес (подходящий)

В системе на разделяемых сегментах определены следующие системные ограничения

- SHMMNI-Максимально возможное количество РС, которые могут существовать в системе одновременно
- SHMMIN-минимально возможный размер РС в байтах
- SHMMAX-максимально возможный размер РС в байтах

Если процесс пытается создать РС, а уже максимально возможное, то он будет заблокирован в ожидании, пока какой-нибудь процесс освободит сегмент. Если больше максимально возможного – такой вызов не будет выполнен.

## 12 билет

12.1 ОС с монолитным ядром. Переключение в режим ядра (и события, переводящие). Диаграмма состояний процесса и переход из одного состояния в другое – причины каждого перехода. Диаграмма состояний процесса в UNIX.

Переключение контекста и причины переключения контекста. Система прерываний (в 2021 уже не было)

Тут просят перевод в режим ядра и соответствующую диаграмму. Я не уверена, что именно хотят - в монолитном ядре (есть по ссылке 1) или в целом (по ссылке 2, (там будет сначала в целом, потом для юникс)). Привожу оба, но кажется, что хотят второе - то есть пишем в целом про монолитное ядро, а потом переходим по 2 ссылке и оттуда уже берем 2 диаграммы)

ОС с монолитным ядром. (ссылка 1)

события, переводящие в режим ядра.

Переключение в режим ядра. Диаграмма состояний процесса и переход из одного состояния в другое – причины каждого перехода. Диаграмма состояний процесса в UNIX. Переключение контекста и причины переключения контекста. Система прерываний (в 2021 уже не было) (ссылка 2)

(до блока про потоки)

**Система прерываний**

(Опять непонятно, что хотят. Если про монолитное ядро, то вот):

Важнейшей для работы систем с монолитным ядром является система прерываний, состоящая из:

- системные вызовы (часто используется термин API - те функции, которые может запросить пользователь у системы).
- исключения - аварийные ситуации в системе (исправимые - page fault, неисправимые - деление на 0).
- аппаратные прерывания - асинхронное событие, пред. 2 - синхронные. Предназначены для информирования процессора об окончании операции ввода-вывода.

1-2 - синхронные события, 3 - асинхронное событие.

Реализована идея распараллеливания функций - внешними устройствами управляет не процессор. Канальные, шинные внешние устройства - контроллеры. Команды от процессора переводят контроллер в нужный режим. В результате нажатия клавиши она поступает в буфер клавиатуры, для этого необходимо проинформировать процессор и он запустит нужный обработчик прерывания.

Приложение никогда не должно уметь обращаться напрямую к внешним устройствам, иначе оно бы смогло изменять код ОС и такую ОС невозможно было бы защитить. В операционной системе read/write главные системные вызовы.

(если в целом - было описано ранее в [события, переводящие в режим ядра.](#))

**12.2 Задача:** читатели-писатели – монитор Хоара, решение с использованием семафоров Unix и разделяемой памяти, пример реализации из лабораторной работы.

**Задача: читатели-писатели – монитор Хоара,**

**решение с использованием семафоров Unix и разделяемой памяти, пример реализации из лабораторной работы.**

## 13 билет

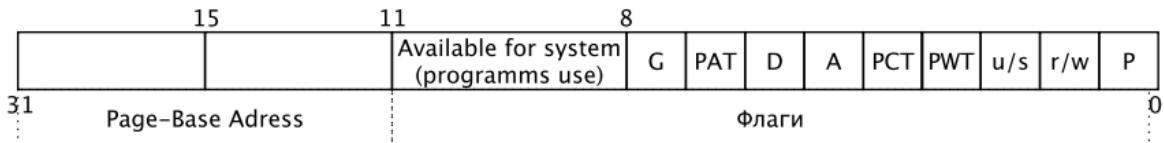
13.1 Виртуальная память: управление памятью страницами по запросу – три схемы. Алгоритмы вытеснения страниц: демонстрация особенностей на модели траектории страниц. Рабочее множество – определение, глобальное и локальное замещение. Флаги в дескрипторах страниц, предназначенные для реализации замещения страниц.

**Виртуальная память: управление памятью страницами по запросу – три схемы.**

*Пишем с преамбулы, потом будет упоминание про “следует посмотреть все 3 схемы преобразования” - переходим и пишем. Про гиперстраницы (идет после упоминания) скорее всего не надо*

## Алгоритмы вытеснения страниц: демонстрация особенностей на модели траектории страниц, глобальное и локальное замещение. Флаги в дескрипторах страниц, предназначенные для реализации замещения страниц.

И вот еще отдельно про флаги в дескрипторах страниц



### Флаги для замещения

- P [Present] - Он установлен, если страница находится в памяти.
- A [Accessed] - показывает, было ли произведено обращение к странице с момента загрузки ее в память.
- D [Dirty] - показывает была ли произведена запись в страницу.

### Остальные флаги (на всякий)

- P (Present): указывает, находится ли страница (или таблица страниц) в физической памяти.
- R/W (Read/Write): определяет привилегии чтения/записи для страницы или группы страниц (в случае, когда элемент каталога страниц указывает на таблицу страниц).
- U/S (User/Supervisor): определяет привилегии пользователя/супервизора для страницы или группы страниц.
- PWT (Page Write Throw): контролирует кэширование страницы (write-through и write-back).
- PCD (Page Cache Disable): контролирует кэширование страницы. – A (Accessed): показывает, было ли произведено обращение к странице с момента загрузки ее в память.
- D (Dirty): показывает была ли произведена запись в страницу.
- PAT (Page throw Attribyte Index)
- G (Global page): указывает, что описываемая страница является ГЛОБАЛЬНОЙ, если установлен.
- Available for system

## Рабочее множество – определение.

**13.2 Синхронизация и взаимоисключение параллельных процессов в распределенных системах: централизованный и распределенный алгоритмы, алгоритмы Token-ring; сравнение алгоритмов. Транзакции: определение, особенности, двухфазный протокол фиксации.**

### **Синхронизация и взаимоисключение параллельных процессов в распределенных системах**

Алгоритмы взаимоисключения в распределенных системах:

1. централизованные;
2. распределенные;
3. token-ring.

Распределенные системы называются так, потому что ресурсы рассредоточены, процессы не имеют общей памяти (например, сети). Процессы в них могут синхронизироваться только сообщениями. Взаимодействие по системе «клиент-сервер».

Процессам часто нужно взаимодействовать друг с другом, например, 1 процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. В 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. То есть чтобы не могли одновременно войти в критическую секцию.

*В целом: Существуют следующие способы взаимоисключения:*

1. Программный способ
2. Аппаратный способ
3. С помощью семафоров
4. С использованием мониторов

### **Централизованный алгоритм**

Используется процесс-координатор. Обычно в качестве ПК выбирается процесс, который выполняется на хосте с наибольшим значением сетевого адреса.

Если процесс желает войти в определенную секцию, он посылает ПК сообщение-запрос с указанием названия КС, в которую хочет пойти. Получив такое

сообщение, ПК проверяет, не находится ли другой процесс в данной КС, или может возникнуть ситуация, что КС пока не занята, но к ней уже имеется очередь. Тогда процесс ставится в очередь и разрешение процессу не посыпается. То есть запросивший разрешение войти в КС процесс будет блокирован до тех пор, пока не получит сообщение-разрешение. Так решается проблема взаимоисключения.

Но что если ПК аварийно завершится. Все ожидающие разрешения будут заблокированы навсегда. Решается только с помощью `timeout`-ов. Процесс не может ждать бесконечно долго. Но и здесь проблема – невозможно посчитать задержки передачи сообщений.

Чтобы такой ситуации не возникало, в централизованной системе любой процесс, обнаруживший отсутствие координатора может инициировать новые выборы координатора и восстановить работоспособность системы. Такой подход часто называют алгоритмом забияки.

Если процесс обнаруживает отсутствие координатора, то он инициализирует выборы координатора и посылает специальный запрос (например, выборы), в котором указывает свой номер процессам с большими номерами (требование про большие номера может отсутствовать – всем). Процесс, получивший сравнивает свой номер с переданным номером. Если его номер больше, то он инициирует выборы со своим номером. Если нет – то переходит в процесс ожидания (сообщения о новом координаторе). Так останется один процесс с наибольшим номером. Будет выбран новый координатор и работа системы восстановлена. Новый координатор посылает сообщение окончание выборов со своим сетевым номером.

Другого способа, кроме как вешаться на `timeout` – нет.

Каждая программа на каждой машине такой системе должна помимо функций для общих задач, содержать функции для того, чтобы стать координатором. На каждом хосте.

### **Распределенный алгоритм**

В РА процесс, желающий войти в какую-либо критическую секцию(КС), рассыпает  $n-1$  сообщение-запрос всем взаимодействующим процессам с указанием своего идентификатора, идентификатор КС, куда он хочет войти и временем, когда у него возникла такая необходимость по своим локальным часам. Процесс, получивший такое сообщение-запрос, анализирует его, и его действия зависят от того, в каком отношении к этой КС он сам находится.

Возможны 3 ситуации

1. Процесс-получатель не находится в данной КС и не собирается в нее входить. Посыпает запросившему процессу разрешение.
2. Получатель находится в указанной КС. Тогда не посыпает сообщение о разрешении, а ставит сообщение-запрос в очередь.

3. Получатель хочет сам войти в указанную КС. Тогда сравнивает время, которое пришло в запросе со временем в своем сообщении-запросе. Если время в его сообщении-запросе больше, чем время, пришедшее в сообщении, то посыпает сообщение разрешения. Иначе – ничего не посыпает и ставит сообщение – запрос в очередь

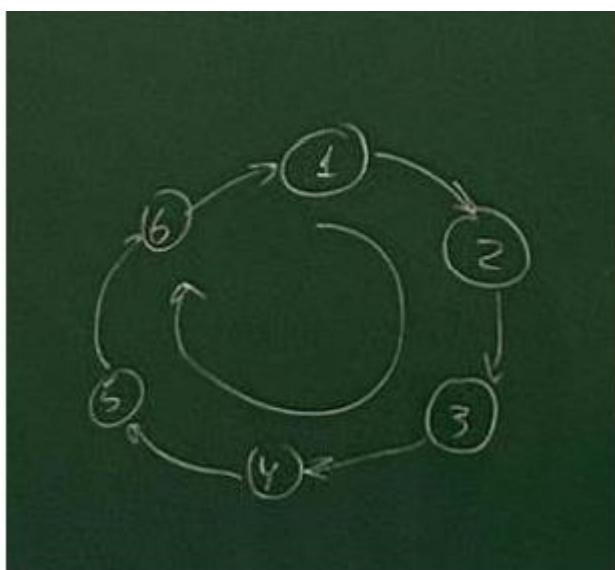
В итоге процесс сможет войти в нужную ему КС только получив  $n-1$  сообщение-разрешение.

Любой из взаимодействующих процессов может аварийно завершиться. И тогда не получит  $n-1$  сообщение. Это еще более сложная ситуация. Решается этот вопрос индивидуально.

Предполагается, что передача сообщения надёжна: получение каждого сообщения сопровождается подтверждением.

### **Токен-ринг**

Процессы выстраиваются в логическое кольцо, в котором каждый процесс знает свой номер в этом кольце и номер ближайшего к нему процесса. Кольцо логическое.



С каждой КС в такой системе связан токен – специальное сообщение, которое циркулирует по кольцу. Взаимодействие выполняется по принципу точка-к-точке (point-to-point). Когда процесс получает токен, анализирует, нужно ли ему самому войти в данную КС. Если нет – сразу посыпает токен дальше. Иначе – удерживает, входит, выходит, и посыпает токен дальше.

Если какой-либо из процессов завершится аварийно, логическая цепь будет разорвана

### **сравнение алгоритмов**

Самой надежной системой является централизованный алгоритм. За счет возможности выбора нового координатора.

В распределенной – должны быть синхронизированы по времени. Если какой-то из процессов перестал существовать, то процесс, разославший  $n - 1$  сообщение-запрос, не сможет получить  $n - 1$  сообщение-ответ: потеря работоспособности по конкретной критической секции.

В случае Токен Ринг: если какой-то процесс перестает существовать, то цепь разрывается. Чтобы ее восстановить, надо предпринять соответствующие действия. Если циркулирование токена по каким-то причинам прервано, то есть только одно средство определения: таймаут.

### **Транзакции: определение, особенности**

Все что мы рассмотрели – механизмы нижнего уровня.

Транзакция – само по себе неделимое действие. Транзакция – средство взаимодействия процессов, высокоуровневое. Транзакция – последовательность операций над одним/несколькими объектами базы данных (файлами, записями и т. д.), которые переводят систему из одного целостного состояния в другое целостное состояние. Модель транзакций пришла из бизнеса.

Чтобы транзакция поддерживалась системой, надо, чтобы предоставлялся набор функций. Обычно – begin, end, abort, read, write

Свойства:

- упорядочиваемость – свойство, которое гарантирует, что если 2 или более транзакции выполняются параллельно, то конечный результат будет выглядеть так, как если бы все транзакции выполнялись последовательно в некотором определенном порядке.
- неделимость – если транзакция находится в процессе выполнения, то никто не может увидеть ее промежуточные результаты
- постоянство – после фиксации транзакции никакой сбой не может отменить результаты ее выполнения

или так (с другой лекции) (атомарность, согласованность, изолированность, устойчивость)

Существуют 2 основных механизма реализации неделимых транзакций

#### **1. Индивидуальное рабочее пространство**

Процессы, задействованные в транзакции, имеют индивидуальные рабочие пространства, в котором копии всех файлов и объектов, которые изменяются в этой транзакции. При этом никакие изменения в исходных файлах или объектах не выполняются, пока исходная транзакция не завершена. Когда транзакция завершена, изменения копируются в исходные файлы.

Это крайне затратно – много копий одних и тех же данных – большие накладные расходы

#### **2. Список намерений**

Модифицируются сами исходные файлы или объекты. Не копии. Перед изменением любого блока любого файла или поля структуры, то есть любого изменения – в специальный журнал регистрации записывается старое значение и новое. Только после записи в журнал регистрации производится изменение в исходном файле или объекте.

- Если транзакция фиксируется, то сначала делается запись в журнал регистрации, но старые значения сохраняются.
- Если транзакция прерывается, то информация, записанная в журнал регистрации, записывается для так называемого отката. То есть для возвращения исходных файлов, объектов в исходное состояние

В распределённых системах журнал может потребовать специального протокола взаимодействия операций на разных машинах. На отдельных машинах при этом могут храниться индивидуальные наборы данных. Поэтому для достижения свойства неделимости используется специальные протоколы. Чаще – протокол двухфазной фиксации транзакции.

Протокол=соглашение, в какой последовательности какие действия выполняются  
**протокол двухфазной фиксации транзакции**

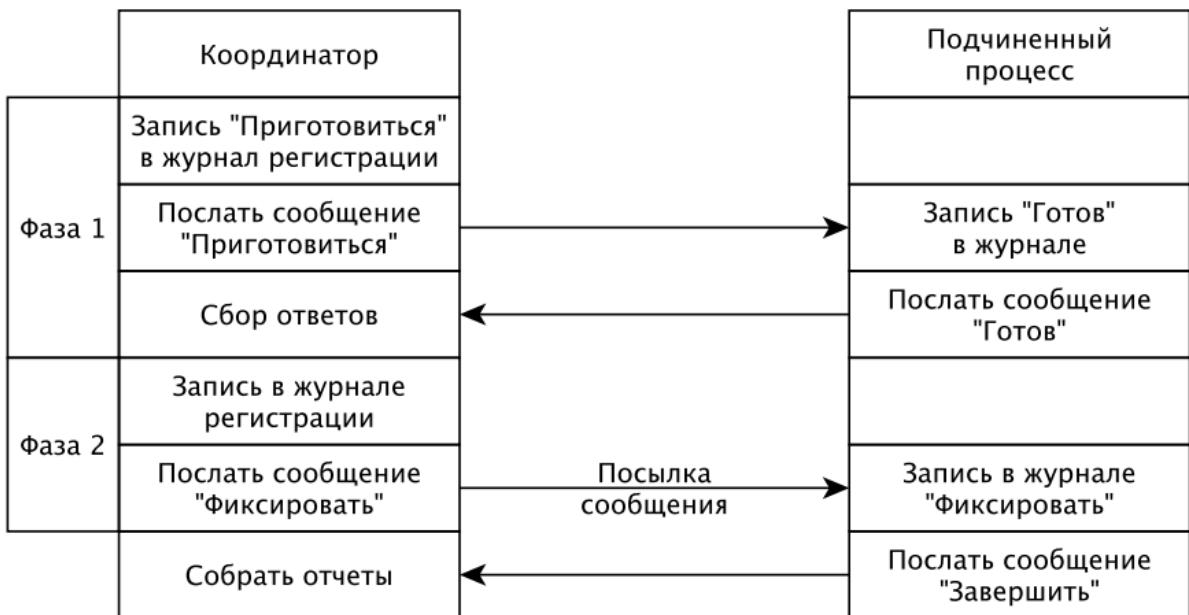
Один процесс выполняет функции координатора (К).

1. К начинает транзакцию и делает запись в свое локальном журнале регистрации. После этого он посыпает подчиненным процессам, которые выполняют эту же транзакцию сообщение подготовиться к транзакции. Получив такое сообщение, подчиненные процессы проверяют, готовы ли они к фиксации и делают запись в своих локальных журналах. Только после этого они посыпают сообщение готов к фиксации. Потом переходят в состояние ожидания соответствующего сообщения от К.

К собирает все сообщения готов к фиксации от подчиненных процессов. Если хотя бы 1 не прислал такое сообщение, то такая транзакция прерывается и должен быть сделан откат всеми системами, которые участвовали в транзакции в соответствии с теми записями, которые они делали в локальных журналах.

2. Если К получил сообщения от всех – записывает это в журнале регистрации и посыпает всем сообщение фиксировать. Получив такое сообщение, процесс проверяет готовность и делает запись в журнал Фиксировать и посыпает сообщение Завершить

Этот протокол гарантирует успешное завершение транзакции всеми процессами



## 14 билет

14.1 Особенности взаимодействия параллельных процессов в распределенных системах. Синхронизация логических часов (алгоритм Лампорта). Алгоритм взаимодействия: централизованный, распределенный алгоритмы. Token Ring. *RPC – механизм.* (с 2021 - его нет)

Особенности взаимодействия параллельных процессов в распределенных системах.  
Алгоритм взаимодействия: централизованный, распределенный алгоритмы. Token Ring.

(до транзакций)

**Синхронизация логических часов (алгоритм Лампорта).**

*Если речь идет о физическом времени, то оно в системах может устанавливаться как меньше текущего, так и больше. Время будет прибавляться или отниматься. В часах Лампорта – только увеличивается. Это используется в моделях непротиворечивости данных – самое общее название, связанное с использованием синхронизации по часам. При этом рассматривается*

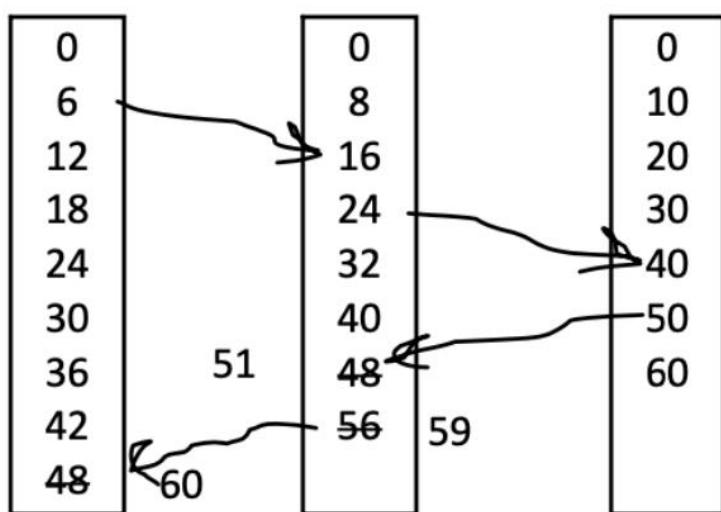
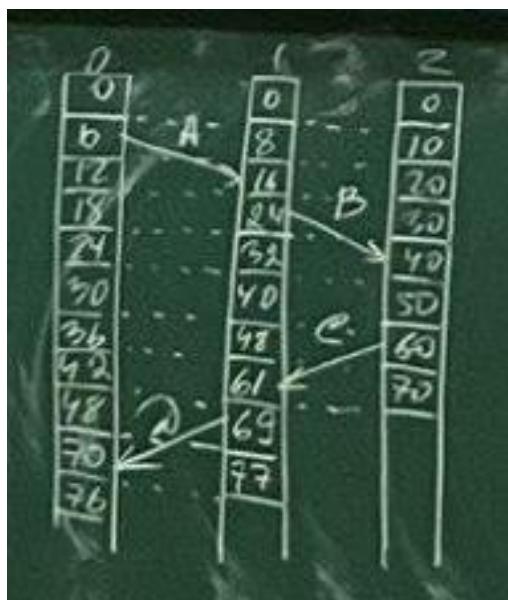
- строгая непротиворечивость,
- последовательная непротиворечивость
- причинная непротиворечивость
- непротиворечивость FIFO

Проблема: локальные часы компьютера, имеющие ограниченную точность. Сообщение получено раньше, чем было отправлено – НЕ МОЖЕТ.

Алгоритм Лампорта применяется для синхронизации временных отметок (для соблюдения отношения "случилось до / случилось после")

1. Процесс, отправивший сообщение, отправляет и время отправки по локальным часам.
2. Получивший процесс: если его время (то есть время получения) меньше, чем время отправления, то он делает: свое время=время отправки + 1

(рисунки одинаковые, просто на втором наглядно показывается, что локально время может меняться, как описано выше)



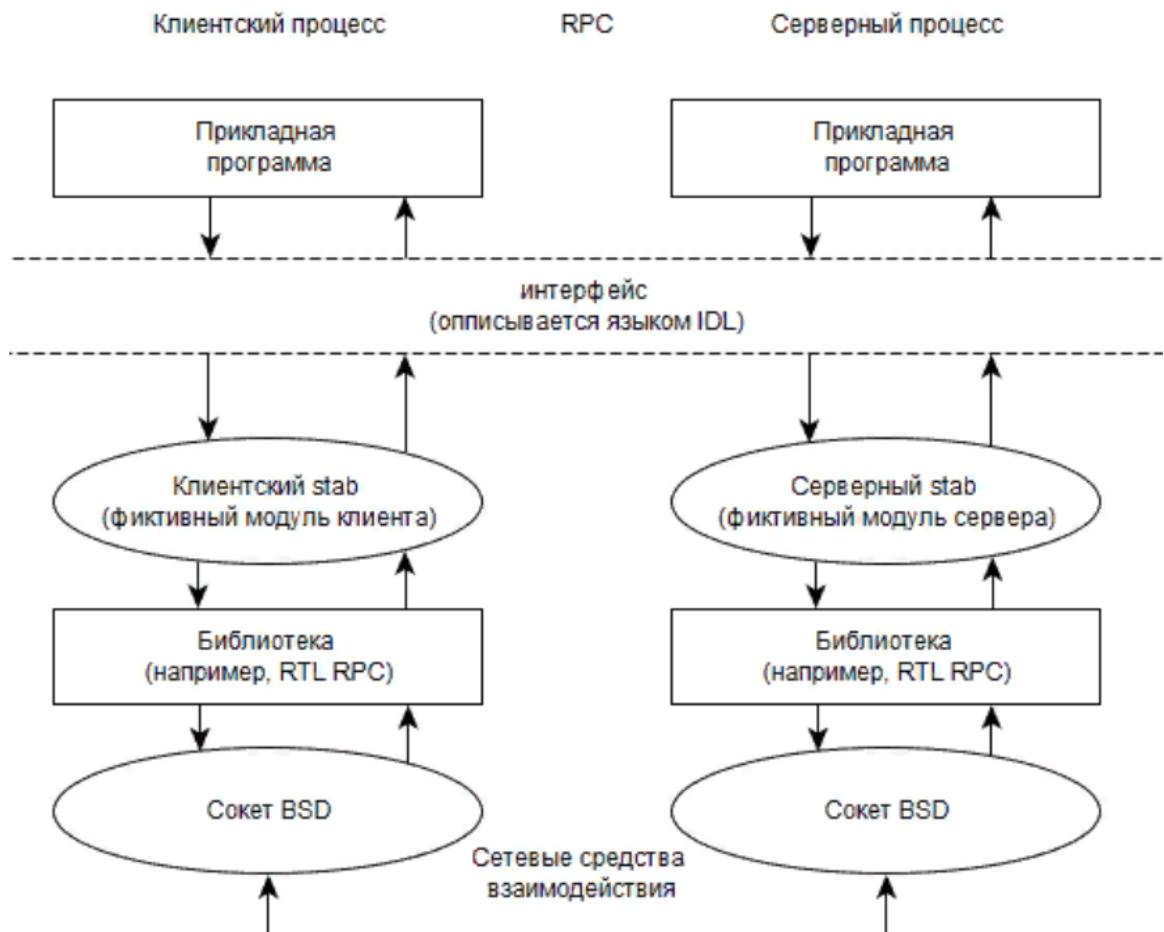
**RPC – механизм. (remote procedure call, вызов удаленных процедур).**

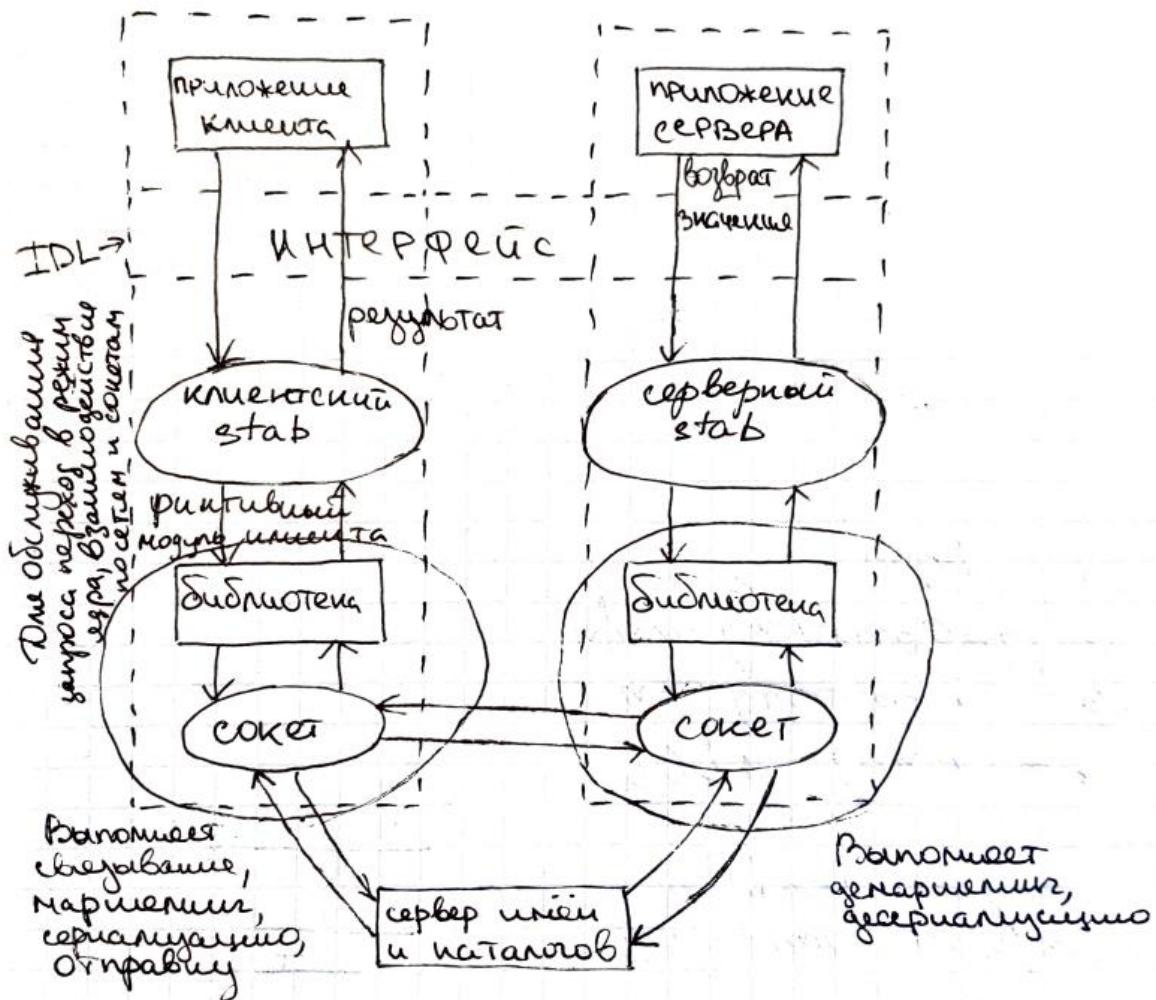
Механизм реализованный первоначально в Unix, с помощью которого один процесс активизирует другой процесс на удаленной или той же самой машине для запуска/выполнения функции от своего имени.

В Unix он напоминает вызов обычной (локальной) функции. Процесс вызывает функцию и передает ей данные (фактические параметры) и собственно ждет, когда будет возвращен результат выполнения этой функции.

Спецификация RPC – эту функцию выполняет другой процесс. Такое взаимодействие выполняется по схеме клиент-сервер. Процесс,зывающий RPC, является клиентским, а процесс, который выполняет RPC функцию – является серверным.

(не знаю, какую из 2 картинок надо - у нас не было)





RPC является низкоуровневым средством взаимодействия, от этого не становится менее интересным.

У RPC есть особенности, главное из которых является то, что механизм RPC скрывает от пользователя этот механизм удаленного доступа. Пользователь RPC может не заморачиваться на то, что он обращается к удаленной машине. RPC появились в 80-е.

### Описание работы:

Прикладная программа, выполняемая клиентским процессом, выполняет вызов удаленной процедуры. При этом для этой программы это просто вызов на локальной машине. В системе существует интерфейс, то есть вызов, выполняемый приложением, преобразуется в вызов так называемого клиентского stab'a (фиктивный модуль клиента; клиентская заглушка).

Вызов клиентского стаба, действия:

1. Подготовка буфера сообщения.
2. Упорядочивание параметров в буфере.
3. Добавление в сообщение полей заголовков.
4. Системный вызов, в результате которого происходит переход в режим ядра и выполняется переключение контекста.

Первые 4 действия выполняет стаб клиента.

5. В ядре для того, чтобы обработать сообщение, оно должно быть скопировано в буфер ядра (В режиме пользователя свой буфер, в режиме ядра - свой).
6. Определение адреса назначения.
7. Адрес помещается в заголовок сообщения.
8. Инициализируется сетевой интерфейс.
9. Включается таймер.

В результате вызова начинает выполняться stab. Происходит вызов функции стандартной библиотеки. Взаимодействие выполняется через сокеты.

В результате вызова библиотечной функции, происходит обращение к местному сокету (сокету BSD).

Все эти действия выполняются в рамках вызова клиентской программы на локальной машине, на которой выполняется клиентский процесс. В результате задействования сокетов, действуются сетевые средства (т. н. транспортный уровень). Осуществляется передача сообщения.

Итог действий на стороне клиента - формирование сетевого пакета, который передается процессу-серверу.

На серверной стороне через сокет пакет поступает на машину, на которой выполняется процесс-сервер. Вызывается библиотека и её функции. На стороне сервера имеется серверный stab.

Таким образом, клиентский стаб вызывает последовательность действий, вызывает в процессе выполнения. В процессе выполнения клиентского стаба выполняется системный вызов. Система переходит в режим ядра. В режиме ядра выполняются совершенно специфические для RPC действия, связанные с формированием сообщения. После того, как сообщение сформировано, это сообщение по сети передается другой машине. Воспринимается через сокет в режиме ядра, распаковывается в режиме ядра, передается серверному стабу, затем передается прикладной программе.

*В результате действий система создает пакет RPC, а также создает одно или несколько обращений к серверу.*

*Выполнение интерфейсов с помощью специального языка – IDL. Клиентский сервер подключается через RPC. Передаются форматированные данные (маршинг и сериализация). Маршинг – перекомпоновка и упаковка данных в связи с требованиями системы. СерIALIZАЦИЯ – преобразование сообщения в серию байтов.*

**14.2 Аппаратные прерывания: типы аппаратных прерываний и их особенности.**  
Прерывания от устройств ввода-вывода: назначение и адресация аппаратных прерываний в защищенном режиме (схема). Пример кода обработчика прерывания от системного таймера из лабораторной работы по защищенному режиму.

### **Аппаратные прерывания: типы аппаратных прерываний и их особенности.**

#### **Прерывания от устройств ввода-вывода: назначение**

ни одна ОС не дает напрямую обращаться к устройствам ввода-вывода, так как это сделает ее незащищенной (равносильно обращению к объектам ядра)

Назначение прерываний от устройств ввода-вывода - информирование процессора об асинхронных событиях, поступающих в систему.

#### **Адресация аппаратных прерываний в з-р.**

#### **Пример кода обработчика прерывания от системного таймера из лабораторной работы по защищенному режиму**

## **15 билет**

**15.1 Межпроцессорное взаимодействие в Unix System V (IPC): сигналы, программные каналы, семафоры и разделяемая память; примеры использования из лабораторных работ.**

Набор средств взаимодействия параллельных процессов в Unix System V (IPC):

- Семафоры
- Сигналы
- Программные каналы (именованные и неименованные)
- Разделяемая память

### **Сигналы**

(вопрос? что происходит с процессом, который получил сигнал?????????

*(Цитата с лекции (этого года)): Как правило, получение некоторым процессом сигнала указывает ему на необходимость завершить работу. Вместе с тем реакция процесса на сигнал зависит от того как сам процесс определяет свое поведение в случае приема к-либо определенного сигнала.*

*(с консультации) 3 возможные реакции: стандартным образом, игнорирование, вызов собственного обработчика)*

Сигнал – способ информирования процессов ядром о произшествии какого-то события. Если возникает несколько однотипных событий, процессу будет подан только один сигнал.

Механизм сигналов позволяет процессам реагировать на события, которые могут происходить внутри процесса или вне его.

Как правило, получение некоторым процессом сигнала указывает ему на необходимость завершить работу. Вместе с тем реакция процесса на сигнал зависит от того как сам сигнал определяет свое поведение в случае приема к-либо определенного сигнала.

Процесс может реагировать на сигнал стандартным образом, то есть в соответствии с существующим в системе обработчиком сигнала, может игнорировать сигнал или вызвать на выполнение свой обработчик сигнала.

Начиная с классического юникс сигналы имеют числовой идентификатор, а также мнемоническую (мнемоника – форма представления, удобная для запоминания) форму записи идентификатора сигнала, которые хранятся в библиотеке signal.h.

Описание нескольких наиболее важных сигналов.

```
#define SIGHUP 1 // Разрыв связи с терминалом
// В UNIX сигналы обозначаются большими буквами - это принятый способ обозначения
// предопределенных макросов.
// Так же есть числовой ID.
#define SIGINT 2 // ctrl + C
#define SIGKILL 9 // (я не буду комментировать)
#define SIGSEGV 11 // нарушение сегментации
// Нужно обратить внимание, почему тут есть такая ошибка, когда мы обсуждали,
// собственно выполняется странничное преобразование.
// Умеем размечать сегмент кода, данных, стека. Эта ошибка означает,
// что в системе выполняется преобразование страницами(?) по запросу
// Мы с вами говорили, что системы для защиты адрес пространств
// проверяет выход процесса за адресное пространство
#define SIGPIPE 13 // Запись в канал есть, чтения нет
#define SIGALRM 14 // Сигнал побудки (будильник) (лол, у нее неверно было)
#define SIGTERM 15 // программное прерывание
// Когда используется kill в терминале - возникает программное прерывание
//(посыпается сигнал номер 15).

// Есть 2 сигнала (SIGUSR1, SIGUSR2),
// которые предназначены непосредственно для пользователя.

#define SIGCLD 18 // завершился процесс потомок
#define SIGPWR 19 // авария питания
#define SIG_DFL(int(*)()) 0 // все установки будут использоваться по умолчанию
#define SIG_IGN(int(*)()) 1 // приписывает игнорирование сигнала.
```

## РАБОТА С СИГНАЛОМ

Int **kill**(int pid, int sig) //вызов

`kill(getpid(), SIGALARM)` // означает, что сигнал побудки будет послан самому процессу, вызвавшему `kill`

`Kill(pid, sig)`

Сигнал `sig` будет послан процессу с идентификатором `pid` и всем процессам-родственникам.

В юникс помимо иерархии важны группы процессов. Процессы объединяются в группы. Например, в первом параметре указывается значение `pid <= 1` – сигнал посылается группе процессов, `=0` – всем процессам с идентификаторами группы, совпадающим с идентификатором группы процесса вызвавшего `kill` (то есть `pid` – не всегда идентификатор)

Системный вызов **signal** не является стандартным для POSIX 1, но он определен в AMSI C и, следовательно, имеется во всех UNIX. POSIX - portable operating system interface – интерфейс переносимых ОС

*Системный вызов signal возвращает указатель на предыдущий обработчик данного сигнала и его можно использовать для восстановления предыдущего обработчика*

Системный вызов, который входит в POSIX – **sigaction**

```
int sigaction(int signal_num, struct sigaction *action, struct sigaction *old_action);
```

Структура `sigaction` определена в библиотеке `signal.h`

*В системном вызове можно определять дополнительные сигналы, которые будут, например, блокироваться при обработке signal\_num.*

*В лабе подчеркивалось, что с помощью сигналов, которые сопровождают события в системе (асинхронные по отношению к процессу) можно менять ход выполнения программы.*

*В POSIX есть две функции для этих целей:*

1. `sigsetjmp` предназначен для того чтобы отметить одну или несколько позиций в программе
2. `siglongjmp` - для перехода в одну из этих позиций

*Как мы видим, это гибкий подход к изменению поведения программы.*

*(про поэзикс POSIX данный стандарт разработан национальным институтом стандартов и технологий США (раньше - NIST). Последний вариант был опубликован в 1988 году.*

*Полное название - POSIX.1 FIPS (Federal Information Processing Standard)*

*Документ был написан для федеральных ведомств, которые покупают компьютерную технику. В любой стране основным покупателем является государство. Рыночные*

*отношения позволяют нам выбирать, игнорировать этот документ или следовать этому документу и получить в клиенты само государство.*

*Возникновение POSIX привело к появлению других стандартов. POSIX появился вследствие сильного расхождения ОС UNIX - в этих ОС стали использовать разные наборы системных вызовов - возникла ситуация, когда ПО на System5 нельзя было использовать на UnixBSD и наоборот)*

### пример из ЛР

```
#define QUITE_MODE 0  
int MODE = QUITE_MODE;  
...  
void parent_callback(int sig_number) {  
    MODE = LOUD_MODE;  
}  
...  
signal(SIGINT, parent_callback);
```

ПОЛНЫЙ

```
#include <stdio.h>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
#include <string.h>  
#include <stdbool.h>  
#include <signal.h>  
  
#define N 2  
#define TIME_SLEEP 2  
#define LEN 64  
  
#define OK 0  
#define ERR_FORK -1  
#define ERR_EXEC -1  
#define ERR_PIPE -1  
  
#define FORK_FAILURE 1  
#define EXEC_FAILURE 2  
#define PIPE_FAILURE 3  
  
_Bool flag = false;  
  
void catch_sig(int sig_num)  
{  
    flag = true;
```

```

        printf("catch_sig: %d\n", sig_num);
    }

int main()
{
    int child[N];
    int fd[N];
    char text[LEN] = { 0 };
    char *mes[N] = {"BMSTU IU7-52\n", "ABCDEFG\n"};

    if (pipe(fd) == ERR_PIPE)
    {
        perror("Can't pipe!");
        return PIPE_FAILURE;
    }

    printf("Parent process start! PID: %d, GROUP: %d\n", getpid(), getpgrp());
    signal(SIGINT, catch_sig);
    sleep(2);

    for (int i = 0; i < N; i++)
    {
        int child_pid = fork();

        if(child_pid == ERR_FORK)
        {
            perror("Can't fork()\n");
            return ERR_FORK;
        }
        else if (!child_pid)
        {
            if (flag)
            {
                close(fd[0]);
                write(fd[1], mes[i], strlen(mes[i]));
                printf("Message %d sent to parent! %s", i + 1, mes[i]);
            }

            return OK;
        }
        else
        {
            child[i] = child_pid;
        }
    }

    for (int i = 0; i < N; i++)
    {
        int status;
        int statval = 0;

```

```

pid_t child_pid = wait(&status);

printf("Child process %d finished. Status: %d\n", child_pid, status);

if (WIFEXITED(statval))
{
    printf("Child process %d finished. Code: %d\n", i + 1,
WEXITSTATUS(statval));
}
else if (WIFSIGNALED(statval))
{
    printf("Child process %d finished from signal with code: %d\n",
i + 1, WTERMSIG(statval));
}
else if (WIFSTOPPED(statval))
{
    printf("Child process %d finished stopped. Number signal:
%d\n", i + 1, WSTOPSIG(statval));
}
}

printf("\nMessage receive :\n");
close(fd[1]);
read(fd[0], text, LEN);
printf("%s\n", text);

printf("Parent process finished! Children: %d, %d! \nParent: PID: %d, GROUP:
%d\n", child[0], child[1], getpid(), getpgrp());

return OK;
}

```

## Программные каналы

В современных ОС UNIX существуют программные каналы, которые принято называть **pipe** - именованные и неименованные.

### Почему pipe (труба)?

Передача данных в одном направлении (потоковая передача данных). Если надо передавать данные в обе стороны, то нужна вторая труба, в которой движение информации будет противоположно первому.

Именованные программные каналы и неименованные - два типа программных каналов, которые поддерживаются современными ОС.

### Именованные программные каналы

**mknod** - создание именованного программного канала

При создании именованного программного канала, это будет труба типа FIFO (тип потоковой передачи данных)

Являются в системе специальными файлами и видны в файловой системе, более того - они имеют идентификатор в файловой системе. Любой процесс, который знает идентификатор именованного программного канала, может работать с этим программным каналом.

### **Неименованные программные каналы**

Не имеют идентификатора, но поддерживаются средствами файловой системы, имеют дескриптор.

В силу особенностей системного вызова fork (системный вызов fork создает новый процесс, процесс потомок, который является копией процесса предка - наследует код предка, дескрипторы открытых файлов, сигнальную маску и тд) неименованными программными каналами могут пользоваться процессы-родственники, тк процессы потомки наследуют от предка дескрипторы открытых файлов.

**Создание канала:** (В канал нельзя писать, если из него читают, из канала нельзя читать, если в него пишут.

(Канал закрывается на чтение при записи и закрывается на запись при чтении))

```
int fd[2];
pipe(fd);
```

**Важно (на самом деле не очень: не успеваешь - не пиши этот блок!)**

Труба буферизуется на трех уровнях

Программные каналы буферизуются в системной памяти (в области данных ядра системы)

При переполнении системной памяти, буфера, имеющие наибольшее время существования переписываются на диск. Используются стандартные функции работы с памятью.

Если процесс записывает в пайп больше 4096 байт, то труба будет буферизоваться по времени, приостанавливая процесс, который записывает в трубу до тех пор, пока данные не будут прочитаны.

Ограничение значений канала (4096) - повысить эффективность операции обмена. При операции обмена ОС выделяет системные буфера. Если его размер не превышает 4096, то такой канал может целиком разместиться в системном буфере. (Это важная мысль)

*Именованные имеют имя и inode, неименованные – только inode. Неименованные создаются системным вызовом pipe, который возвращает файловый дескриптор, если удалось создать канал. Неименованный имеет только дескриптор, поэтому пользоваться неименованным могут только родственники, потому что процессы-*

*потомки в результате fork наследуют от предка дескрипторы открытых файлов. Это потоковая модель передачи данных.*

*Симплексная связь, односторонняя. Для двусторонней – дуплексной, необходимо как минимум 2 трубы. Программные каналы имеют встроенные средства взаимоисключения, то есть в канал нельзя писать, если читают, и нельзя читать, если в него пишут. Для этого определяли fd[2] – массив дескрипторов. Закрыть для записи и читать и наоборот.*

пример из ЛР

Пример создания программного канала:

```
int fd[2];
int pid;
if (pipe(fd) == -1) {
    printf("Something went wrong with pipe!");
    return 1;
```

Пример чтения/записи с помощью программного канала:

```
const char* SECRET_MSGS[2] = { "secret message 1", "secret message 2" };
...
size_t written = write(fd[1], SECRET_MSGS[0], strlen(SECRET_MSGS[0])); // запись
...
char buffer[BUFFER_LEN] = { 0 };
size_t buf_len = read(fd[0], buffer, BUFFER_LEN); // чтение

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define N 2
#define TIME_SLEEP 2
#define LEN 64

#define OK 0
#define ERR_FORK -1
#define ERR_EXEC -1
#define ERR_PIPE -1

#define FORK_FAILURE 1
#define EXEC_FAILURE 2
#define PIPE_FAILURE 3

int main()
{
```

```

int child[N];
int fd[N];
char text[LEN] = { 0 };
char *mes[N] = {"BMSTU IU7-52 Kozlova\n", "ABCDEFG\n"};

if (pipe(fd) == ERR_PIPE)
{
    perror("Can't pipe!");
    return PIPE_FAILURE;
}

printf("Parent process start! PID: %d, GROUP: %d\n", getpid(), getpgrp());

for (int i = 0; i < N; i++)
{
    int child_pid = fork();

    if(child_pid == ERR_FORK)
    {
        perror("Can't fork()\n");
        return ERR_FORK;
    }
    else if (!child_pid)
    {
        close(fd[0]);
        write(fd[1], mes[i], strlen(mes[i]));
        printf("Message %d sent to parent! %s", i + 1, mes[i]);

        return OK;
    }
    else
    {
        child[i] = child_pid;
    }
}

for (int i = 0; i < N; i++)
{
    int status;
    int statval = 0;

    pid_t child_pid = wait(&status);

    printf("Child process %d finished. Status: %d\n", child_pid, status);

    if (WIFEXITED(statval))
    {
        printf("Child process %d finished. Code: %d\n", i + 1,
WEXITSTATUS(statval));
    }
    else if (WIFSIGNALED(statval))
}

```

```

        {
            printf("Child process %d finished from signal with code: %d\n",
i + 1, WTERMSIG(statval));
        }
        else if (WIFSTOPPED(statval))
        {
            printf("Child process %d finished stopped. Number signal:
%d\n", i + 1, WSTOPSIG(statval));
        }
    }

    printf("\nMessage receive :\n");
    close(fd[1]);
    read(fd[0], text, LEN);
    printf("%s\n", text);

    printf("Parent process finished! Children: %d, %d! \nParent: PID: %d, GROUP:
%d\n", child[0], child[1], getpid(), getpgrp()));

    return OK;
}

```

## Семафоры

Типы семафоров:

- бинарный
- считающий
- множественный

Наборы считающих семафоров - этот тип поддерживают все современные ОС.

Доступ к отдельному семафору из набора выполняется по индексу. Важнейшим свойством набора семафоров является то, что одной неделимой операцией можно изменить все или часть набора семафоров.

Освободить семафор может любой процесс (не только тот, который захватил семафор).

Одной неделимой операцией можно изменить весь или часть набора семафоров.

В адресном пространстве ядра имеется таблица семафоров - системная таблица. В этой системной таблице отслеживаются все создаваемые в системе наборы семафоров. В каждом элементе такой таблицы находятся следующие данные об одном наборе семафоров.

Дескриптор набора семафоров:

- имя (идентификатор) - целое число, присваивается процессом, который создал набор семафоров. Другие процессы по этому имени могут открыть набор и получить дескриптор набора для доступа
- UID - идентификатор создателя набора семафоров (процесс, эффективный UID которого совпадает с UID создателя, может удалять набор и изменять управляющие параметры набора)
- права доступа
- количество семафоров в наборе
- время изменения (одного или нескольких значений семафоров последним процессом)
- время последнего изменения управляющих параметров набора
- указатель на массив семафоров

Семантические наборы семафоров представляются как массивы, первый семафор имеет индекс 0.

Каждый отдельный семафор имеет набор параметров. Дескриптор отдельного семафора:

- Значение семафора
- Идентификатор процесса, который оперировал семафором в последний раз
- Число процессов, заблокированных на семафор

РИСУНОК (ниже)

#### **Таблица семафоров**

Каждый элемент этой таблицы описывает структуры struct semid\_ds (<sys/sem.h>)

Каждая строка таблицы описывает отдельный набор семафоров

Каждый семафор описан структурой struct sem.

На семафорах определены следующие системные вызовы:

- semget() - создаёт набор семафоров
- semctl() - позволяет изменять управляющие параметры набора семафоров
- semop() - изменяет значение семафора - отдельного, набора или части набора семафоров

на семафоре определена структура

```
struct sembuf {
```

```
    unsigned short sem_num; // - индекс семафора
```

```
    short sem_op; // - операция на семафоре
```

```
short sem_flg; // - флаги, определенные на семафоре  
};
```

В отличие от классических семафоров Дейкстры (у него две операции), на семафоре UNIX определены три операции

1.  $\text{sem\_op} > 0$  освобождение ресурса
2.  $\text{sem\_op} == 0$  - ожидание ресурса без захвата
3.  $\text{sem\_op} < 0$  - захват ресурса

Декрементировать семафор можно только если семафор имеет положительное значение.

На семафорах определены специальные флаги:

- IPC\_NOWAIT - информирует ядро системы о нежелании процесса переходить в состояние ожидания
- SEM\_UNDO - указывает ядру, что оно должно отслеживать изменение значения семафора в результате системного вызова `semop()` и при завершении процесса, вызвавшего `semop()`, ядро ликвидирует сделанные изменения для того, чтобы процессы не были заблокированы на семафоре навечно.

пример из ЛР

Пример создания и инициализации набора семафоров:

```
// IPC_PRIVATE - ключ, который показывает, что  
// Набор семафоров могут использовать только процессы,  
// Порожденные процессом, создавшим семафор.  
// Создание семафоров (3 семафора)  
sem_descr = semget(IPC_PRIVATE, 3, IPC_CREAT | perms);  
  
if (sem_descr == -1)  
{  
    perror("Ошибка при создании набора семафоров.");  
    return -1;  
}  
  
if (semctl(sem_descr, SB, SETVAL, 1) == -1)  
{  
    perror("!!! Can't set control semaphors.");  
    return -1;  
}  
  
if (semctl(sem_descr, SE, SETVAL, N) == -1)  
{  
    perror("!!! Can't set control semaphors.");  
    return -1;  
}  
  
if (semctl(sem_descr, SF, SETVAL, 0) == -1)  
{
```

```

    perror( "!!! Can't set control semaphors." );
    return -1;
}

```

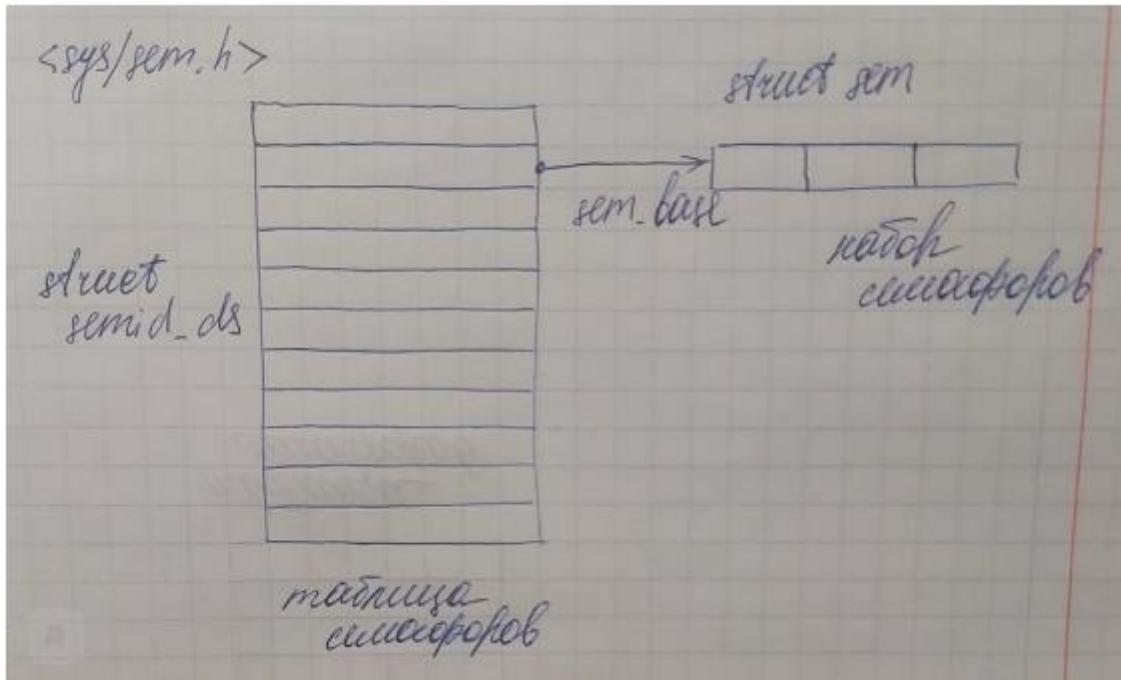
Пример захвата доступа к ресурсу с помощью семафора:

```

struct sembuf producer_begin[2] =
{
    {SE, P, 0}, // Ожидает освобождения хотя бы одной ячейки буфера.
    {SB, P, 0} // Ожидает, пока другой производитель или потребитель выйдет из
    критической зоны.
};

...
// Получаем доступ к критической зоне.
int rv = semop(sem_id, producer_begin, 2);
if (rv == -1)
{
    perror("Производитель не может изменить значение семафора.\n");
    exit(-1);
}

```



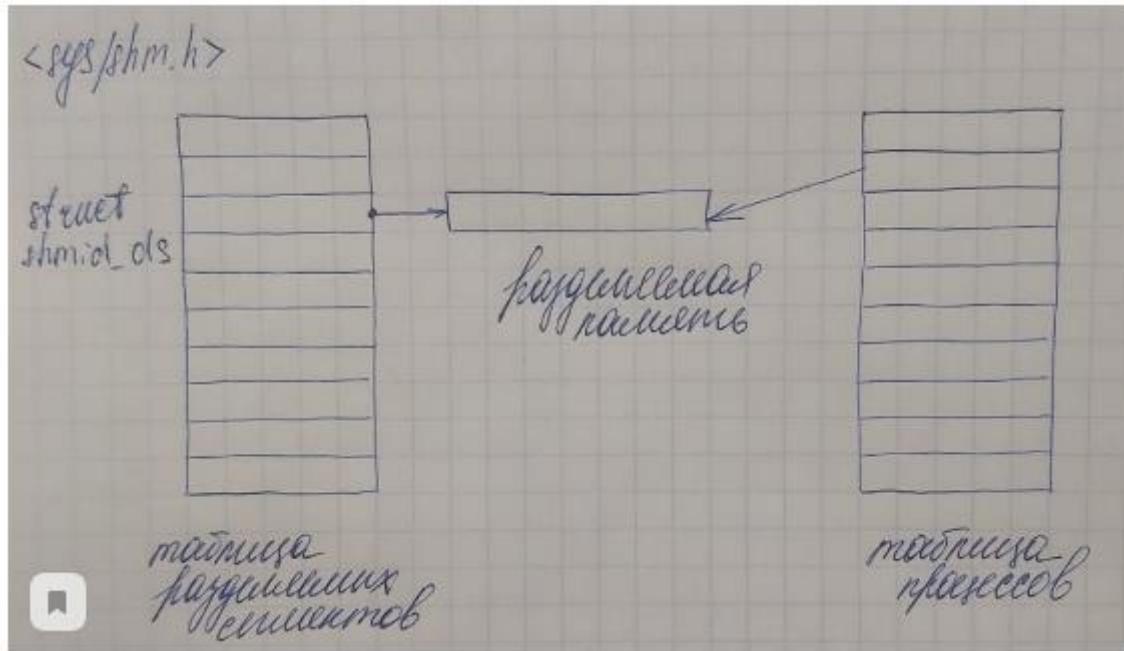
## Разделяемая память

Процессы имеют защищенное адресное пространство, поэтому другой процесс не может обратиться в такое адресное пространство. Для взаимодействия процессов существует одно единственное адресное пространство - область памяти ядра системы. В ядре создаются средства взаимодействия процессов (например, программные каналы).

Особенность разделяемых сегментов памяти: в отличие от других средств взаимодействия, разделяемые сегменты подключаются к адресному пространству

процессов. Это исключает необходимость копирования данных из адресного пространства процесса в адресное пространство ядра и наоборот. Сегменты разделяемой памяти были задуманы как средство повышения производительности при передаче сообщения от одного процесса другому.

Существует таблица разделяемых сегментов.



<sys/shm.h> - дескриптор описывается структурой struct shmid\_ds

Особенностью разделяемых сегментов является то, что процесс получает указатель на разделяемый сегмент

Для разделяемой памяти определены системные вызовы:

- shmget() - создает разделяемый сегмент
- shmctl() - изменяет упр параметры сегмента
- shmat() - (attach) получение указателя на разделяемый сегмент
- shmdt() - (detach) отделение сегмента от адресного пространства процесса

Системное ограничение	значение
-----------------------	----------

SHMMNI	максимальное число разделяемых сегментов
--------	--

SHMMIN	минимально возможный размер разд сегмента в байтах
--------	--

## SHMMAX

максимально возможный размер разд сегмента в байтах

Если процесс попытается создать новый разделяемый сегмент, а их число превысит максимальное (SHMMNI), то процесс будет заблокирован до тех пор, пока другой процесс не освободить какой-то разделяемый сегмент

Если процесс попытается создать разделяемый сегмент, размер которого превышает макс значения (SHMMAX), то системный вызов не будет выполнен - возвращена ошибка.

пример из ЛР

Пример использования разделяемой памяти:

```
// Значение IPC_PRIVATE указывает, что к разделяемой
// памяти нельзя получить доступ другим процессам.
// shmget - создает новый разделяемый сегмент.
// S_IRUSR Владелец может читать.
// S_IWUSR Владелец может писать.
// S_IROGRP Группа может читать.
// S_IROTH Остальные могут читать.

int sem_descr;
int perms = S_IRUSR | S_IWUSR | S_IROGRP | S_IROTH;
int shmid = shmget(IPC_PRIVATE, sizeof(buffer_s), IPC_CREAT | perms);
if (shmid == -1)
{
    perror("Ошибка при создании нового разделяемого сегмента.\n");
    return -1;
}

// Функция shmat() возвращает указатель на сегмент
// shmaddr (второй аргумент) равно NULL,
// то система выбирает подходящий (неиспользуемый)
// адрес для подключения сегмента.

buffer_s *buffer;
if ((buffer = (buffer_s*)shmat(shmid, 0, 0)) == (buffer_s*)-1)
{
    perror("Ошибка при попытке возврата указателя на сегмент.\n");
    return -1;
}
```

## НА ВСЯКИЙ ВОПРОС

Как в ядре организованы Каналы и очереди сообщений

## КАНАЛЫ

Труба буферизуется на трех уровнях

Программные каналы буферизуются в системной памяти (в области данных ядра системы)

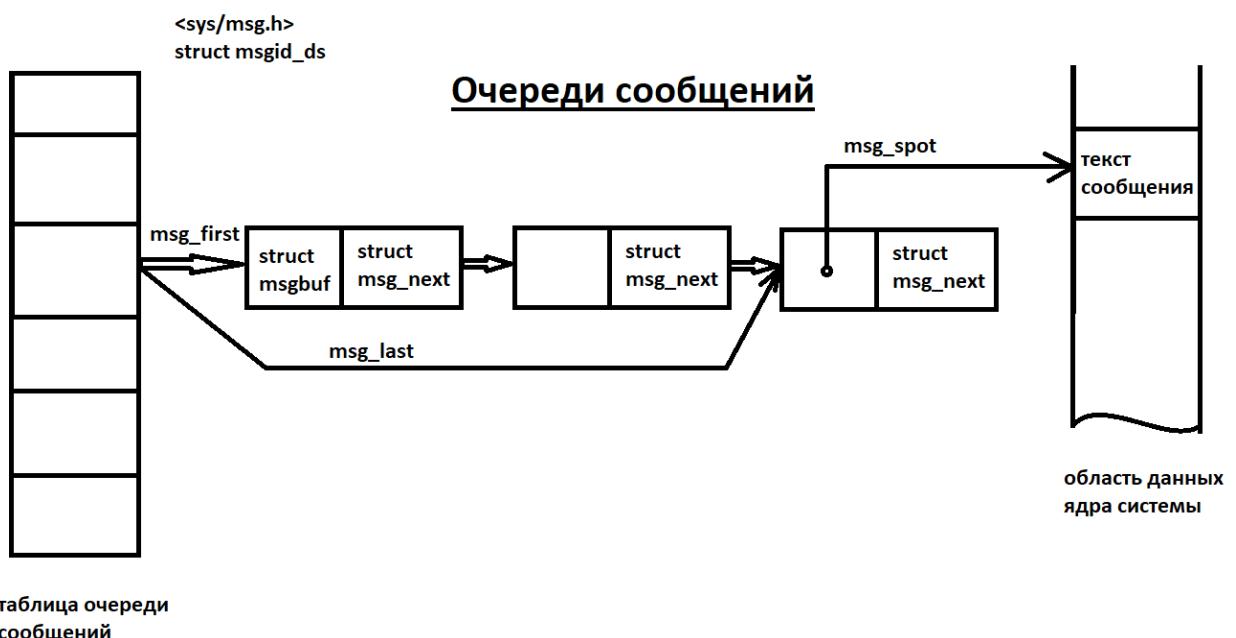
При переполнении системной памяти, буфера, имеющие наибольшее время существования переписываются на диск. Используются стандартные функции работы с памятью.

Если процесс записывает в пайп больше 4096 байт, то труба будет буферизоваться по времени, приостанавливая процесс, который записывает в дробу до тех пор, пока данные не будут прочитаны.

Ограничение значений канала (4096) - повысить эффективность операции обмена. При операции обмена ОС выделяет системные буфера. Если его размер не превышает 4096, то такой канал может целиком разместиться в системном буфере. (Это важная мысль)

Чтение из памяти одиночной переменной только на 30% быстрее, чем передача одной страницы. Это связано с тем, что в системе оптимизируется передача страниц (буфера соответствуют размеру страницы).

## ОЧЕРЕДИ СООБЩЕНИЙ



Для поддержки очередей сообщений в системе имеется системная таблица - таблица очередей сообщений. Эта таблицы содержит дескрипторы всех созданных в системе очередей сообщений. Является связным списком типа очередь, каждый элемент этой очереди имеет указатель на следующий элемент очереди. Последний элемент очереди имеет указатель NULL. Есть два указателя msg\_first и msg\_last.

Дескриптор очереди сообщений описывается структурой struct msgid\_ds (библиотека <sys/msg.h>). Элемент очереди не содержит текста сообщения, содержит ссылку на текст сообщения в области данных ядра системы.

```
struct msgid_ds
{
    long mytype,
    char mytext[MSGMAX];
}
```

Каждое сообщение имеет тип и текст.

Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает ее в конец связного списка соответствующих записей. В каждой такой записи указывается тип сообщения, число байт данных, содержащихся в сообщении, указатель на другую область данных ядра, где фактически находится сообщение.

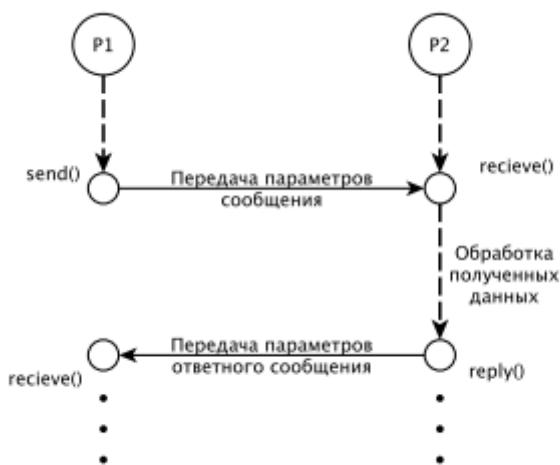
Ядро копирует сообщение из пространства процесса отправителя в область данных ядра системы, чтобы процесс отправитель мог завершиться, а его сообщение осталось доступным для чтения другим процессам.

Когда процесс выбирает сообщение из очереди, ядро копирует это сообщение в адресное пространство процесса-получателя сообщения, после этого сообщение удаляется.

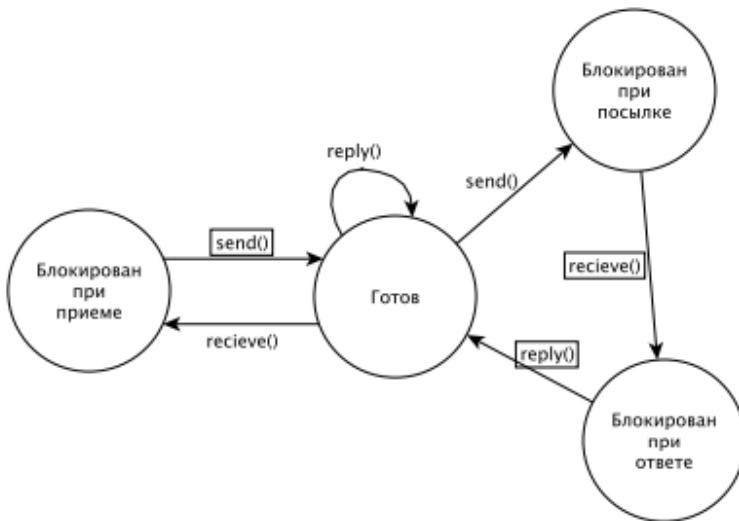
Когда какой-то процесс выбирает сообщение из очереди, ядро копирует это сообщение в АП этого процесса. После этого сообщение удаляется (перестает существовать). Процесс может выбрать сообщения из очереди следующими способами.

1. взять самое старое сообщение, независимо от его типа
2. взять сообщение, если идентификатор сообщения совпадает с идентификатором, который указал процесс. Если существует несколько процессов с таким идентификатором, то взять самое старое из них
3. сообщение, числовое значение типа которого является наименьшим из меньших или равным значению типа, указанного процессом. Если таких несколько – то самое старое

При использовании очередей сообщений процессы могут не блокироваться в ожидании сообщения и при отправке, и при получении. (вспомнить диаграмму 3 состояния блокировки процесса при передаче сообщений)



### Три состояния блокировки



## 15.2 Синхронизация и взаимоисключение параллельных процессов в распределенных системах: централизованный и распределенные алгоритмы – сравнение.

Распределенная система

Централизованный алгоритм (часто встречается как алгоритм Забияки, Под забиякой понимается процесс-координатор. Если какой-то процесс из цепи взаимодействующих процессов обнаружил отсутствие координатора, он инициирует новые выборы координатора.)

Решение, подобное решению на 1 машине. Существует процесс-координатор, например, на машине с наибольшим значением сетевого адреса. Когда какой-то процесс хочет войти, он посыпает запрос с именем участка, в который хочет зайти и ждет координатора. Координатор смотрит, не находится ли кто-то уже там или кто-то на очереди. Если да, то координатор ставит полученный запрос в очередь, иначе посыпает разрешение и помечает область занятой. Необходимо иметь алгоритм выхода из ситуации отказа координатора.

Рассмотрим ситуацию, когда каждый процесс может выполнить роль координатора. В этом случае, если какой-то процесс обнаружил, что координатор отсутствует (если прошел timeout, то он может считать, что координатор отсутствует (здесь может использоваться специальный протокол)). Тогда такой процесс может инициализировать процесс выбора нового координатора. Посыпается специальный запрос, в котором процесс указывает свой номер всем процессам. Если у принимающего процесса больший номер, то он инициирует такой запрос сам. В результате останется один процесс с наибольшим номером. Он становится координатором.

### Распределенный алгоритм

Второй тип алгоритма это распределенный алгоритм.

Алгоритм формулируется следующим образом:

1. Когда процесс хочет войти в критическую секцию, он формирует сообщение.
  - В этом сообщении он указывает:
    - a. идентификатор нужной ему критической секции;
    - b. свой номер (идентификатор);
    - c. время формирования сообщения по своим локальным часам.
  - Предполагается, что передача сообщения надёжна: получение каждого сообщения сопровождается подтверждением.
  - Протокол - соглашение о том, какие действия выполняют процессы при передаче и получении сообщения. Надёжный протокол всегда предполагает подтверждение получения сообщения.
2. Это сообщение, сформированное процессом, процесс рассыпает всем взаимодействующим процессам (то есть  $n - 1$  сообщение)
3. Получив такое сообщение, процесс проверяет, в какой ситуации он сам находится по отношению к критической секции.
  - Возможны 3 ситуации:
    - a. получивший сообщение процесс не собирается входить в данную критическую секцию (и не находится в ней). В этом случае процесс посыпает сообщение-ответ с разрешением войти в критический участок;
    - b. процесс, получивший сообщение, находится в данной критической секции. В этом случае никакое сообщение не посыпается;
    - c. Процесс, получивший сообщение, сам формировал сообщение запрос на вход в данную критическую секцию, но еще не успел это сделать.
      - (на доске: хочет войти в ту же критическую секцию). Тогда процесс проверяет временную метку создания своего сообщения, сравнивает ее с временной меткой полученного сообщения. Если его сообщение было сформировано раньше, то никакого сообщения-ответа процесс не посыпает. Если в результате таких рассылок процесс получает  $n - 1$  сообщение подтверждения входа в критический участок, то процесс может войти в критический участок. Если он не получает хотя бы одно сообщение, то в критический участок процесс войти не может.

В этом случае процесс должен послать  $n - 1$  сообщение-запрос и в ответ получить  $n - 1$  сообщение-разрешение на вход, иначе войти не может.

### **Алгоритм Токен Ринг (Token Ring)**

(еще один распределенный алгоритм)

Смысл заключается в том, что процессы создают логическое кольцо - замкнутую цепочку.

Каждый процесс в логическом кольце знает идентификатор следующего процесса в этом логическом кольце. По данному логическому кольцу передается так называемый токен. Это специальное сообщение, которое содержит идентификатор конкретной критической секции. Этот токен переходит от n-ого процесса к n + 1-ому процессу, циркулирует по логическому кольцу. Когда процесс получает токен, он анализирует, нужно ли ему войти в данную критическую секцию. Если нужно, получив токен, он входит в критическую секцию и токен удерживает. Выйдя из критической секции, процесс отправляет токен следующему в цепи процессу. Пока процесс находится в критической секции он удерживает токен. Если ни один процесс не заинтересован во входжении в критическую секцию токена, такой токен быстро циркулирует по цепочке.

## **Сравнение приведенных алгоритмов**

Самый надежный алгоритм: централизованный алгоритм. За счет возможности выбора нового координатора.

В случае распределенного алгоритма: Если какой-то из процессов перестал существовать, то процесс, разославший n - 1 сообщение-запрос, не сможет получить n - 1 сообщение-ответ: потеря работоспособности по конкретной критической секции.

В случае Токен Ринг: если какой-то процесс перестает существовать, то цепь разрывается. Чтобы ее восстановить, надо предпринять соответствующие действия. Если циркулирование токена по каким-то причинам прервано, то есть только одно средство определения: таймаут.

## **15.2 (2021) Тупики: классификация ресурсов и их особенности. Четыре условия возникновения тупика. Методы исключения тупиков.**

### **Тупики**

Определение: тупик (тупиковая ситуация) – ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другим процессом, который ожидает освобождения ресурса, занятого первым процессом. В результате процессы блокируют друг друга таким образом, что ни один из них не может продолжить выполнение и освободить занимаемый ими ресурс

### **Классификация ресурсов и их особенности**

В теории тупиков принято различать два типа ресурсов (эти типы обладают набором характерных особенностей, которые потребовали такой классификации):

- Повторно используемые ресурсы
- Потребляемые ресурсы

Повторное используемые ресурсы — используются многократно, использование ресурса не изменяет качества и характеристик ресурса. К повторно используемым относятся

аппаратура компьютера, процессоры, память, каналы, устройства ввода-вывода и ПО (например, реентерабельные коды ОС, системные таблицы).

Потребляемые ресурсы — количество в ОС переменно и произвольно. К потребляемым ресурсам относится: память, каналы, внешние устройства, процессор, сообщения (получено -> перестало существовать).

#### **четыре условия возникновения тупика**

Были сформулированы 4 условия возникновения тупика (тупиковой ситуации, deadlock – захват смерти).

Эти условия актуальны и в настоящее время

1. Взаимоисключение - Mutual exclusion. Возникает, когда процессы монопольно используют ресурсы.
2. Ожидание – когда процессы, удерживая полученные им ресурсы, запрашивают и ждут получения дополнительных ресурсов, чтобы продолжить свое выполнение. Hold and wait
3. Неперераспределаемость – когда ресурсы нельзя отобрать у процесса до его завершения или добровольного освобождения ресурса. No preemption
4. Круговое ожидание – когда существует замкнутая цепь процессов, в которой каждый процесс занимает необходимый другому (следующему в цепи) процессу ресурс. Circular wait

#### **методы исключения**

1. Недопущение тупиков (исключение самой возможности возникновения тупиков).

1. Первый подход (целый набор) – стратегия Харвендера: возникновение невозможно, если нарушено хотя бы 1 из условий возникновения тупика.

Первое решение – опережающее требование. До своего начала процесс запрашивает все необходимые ему ресурсы. Процесс начнет свое выполнение только получив все запрошенные ресурсы. Очевидные недостатки:

- 1) процесс должен знать свою максимальную потребность во всех типах ресурсов.
  - 2) процесс запрашивает и получает ресурсы задолго до их реального использования. При этом часть ресурсов он может вообще не использовать при данном проходе – неэффективное использование ресурсов. Это касается аппаратной составляющей.
2. Также подход называется иерархическим распределением. Ресурсы делятся на классы. Деление должно определяться типом ресурса, быть оправданным (в один ресурс не поместят память и семафор). Каждому классу присваивается номер и устанавливается правило, по которому процессы могут захватывать только ресурсы с большими номерами, чем ресурсы, которые они удерживают.

*Тоже нарушает одно из условий. (НЕ ТО ЧТО ЭТО НЕВЕРНО, НО И ТАК МНОГО)*

*Если процессу требуется ресурс с меньшим номером, чем ресурсы, которые он уже получил, чтобы получить ресурсы в правильном порядке, должен освободить занимаемые им ресурсы, а потом захватить в правильном порядке.*

*Очевидно, что набор ресурсов в системе широкий, в системах пытаются так сформировать структуру, чтобы отобразить наиболее часто используемую последовательность захвата ресурсов. Но это непросто*

3. Устраняется условие неперераспределемости. Если процесс не может получить нужный ему ресурс, то он должен освободить занимаемый им ресурс.

## 2. Обход тупиков (предотвращение тупиков)

### 1. Алгоритм Банкира

Классика – алгоритм банкира. Его предложил Дейкстра. Он вообще теоретический.

Существуют аппроксимации – например, алгоритм Хаббермана. Базовый алгоритм банкира.

Ограничения:

1. Число процессов известно, ограничено, то есть не меняется
2. Число ресурсов и число единиц каждого ресурса известно, не меняется

Процессы до начала выполнения должны указать в своих заявках (claim) свою максимальную потребность в каждом типе ресурса и затем при выполнении процесс не может захватывать ресурсов больше, чем указано в его заявке (естественно, нельзя заказывать больше, чем используется в системе).

Работа менеджера ресурсов выполняется по алгоритму, который гарантирует, что тупиковая ситуация не наступит. Для этого каждый запрос процесса на ресурс проверяется относительно свободных единиц ресурса данного типа в системе.

Если процесс попытается запросить больше, чем в заявке, такой запрос выполнен не будет, это невозможно.

Формально состояние системы процессов является безопасным относительно тупика, если существует последовательность процессов такая, что:

1. первый процесс в найденной последовательности гарантированно завершится, так как даже если он запросит максимально заявленное количество единиц ресурса, у системы имеется необходимое количество свободных единиц ресурса для удовлетворения его запроса.
2. Второй процесс в найденной последовательности может гарантированно завершиться, если завершился первый процесс и вернул системе все занимаемые ей единицы ресурса, что в сумме со свободными единицами

ресурса позволить удовлетворить максимальную потребность этого ресурса и тд.

3. I-й сможет завершиться, если все предыдущие  $i-1$  процессов завершились и сумма ресурсов сможет удовлетворить максимально возможную потребность в единицах ресурса.

4. В результате если все процессы могут завершиться, то система находится в безопасном состоянии

## 2. Алгоритм Хаббермана

Чтобы узнать, каким является текущее состояние системы, работа начинается с некоторого заданного состояния.

И алгоритм Д, и Х, имитирует выполнение каждого процесса, выделяя ему максимальное количество запрошенных ресурсов. Если все процессы при этом завершаются, то состояние безопасное.

В алгоритме Х также фиксированное количество процессов  $N$ , фиксированное количество типов ресурса  $M$ , о каждом типе ресурса известно количество единиц этого ресурса. То есть имеется матрица доступных ресурсов

## 3. Обнаружение и восстановление

Для этого используется **двудольный направленный граф** (градовая модель Холда). Имеется 2 непересекающихся подмножества вершин – подмножество вершин процессов и подмножество вершин ресурсов. При этом дуга не может соединять вершины одного подмножества.

*Граф называется направленным, потому что существует 2 типа дуг – выделение, когда дуга выходит из вершины подмножества ресурсов и входит в вершину подмножества процессов, запрос – когда наоборот*

Граф описывает ситуацию в системе, которая может в какой-то момент возникнуть. При этом эти графы используются для обнаружения тупиков. Фактически такой граф становится необходимым, чтобы убедиться, что система пришла в тупик и причины. Речь идет о большой системе взаимодействующих процессов.

Тупиковая ситуация с использованием ГМХ обнаруживается методом редукции графа. Если запрос процесса мб удовлетворен, то такая дуга мб удалена.

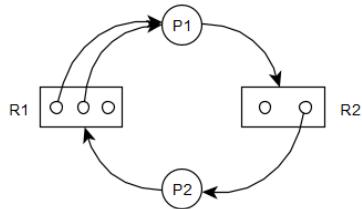
В результате того, что запрос удовлетворен, он может освободить ресурсы, которые ранее занял, и эти ресурсы могут быть другими процессами.

Если в рез редукции все дуги удалены и все вершины становятся изолированными, то система не находится в тупике. Если же все дуги удалить невозможно (те дуги, которые определяют петлю запросов), значит система в тупике.

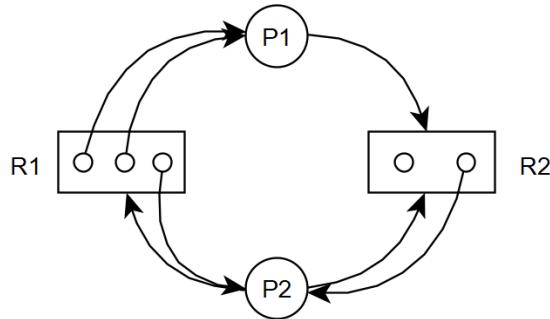
Требуется определить, является ли ситуация тупиковой или нет.

р1 запросил 2 единицы r1 и получил их. р2 запросил 1 единицу r2 и получил её. р1 дополнительно запрашивает еще 1 единицу r2, а р2 - еще 1 единицу r1. У системы есть

необходимое количество ресурсов, чтобы удовлетворить потребности и процесса p1, и процесса p2 в единицах ресурсов (ситуация на картинке ниже)

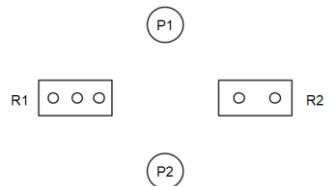


Теперь p2 получил единицу ресурса r1 и ему требуется еще 1 единица ресурса r1. Кроме единицы ресурса r1 ему требуется единица ресурса r2.

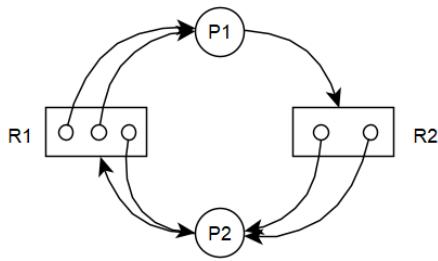


Видно, что процесс p1 может завершиться. Значит процесс p1 может стать изолированной вершиной (может освободить занимаемые им ресурсы и эти ресурсы будут возвращены системе).

Когда p1 завершится, запросы p2 могут быть удовлетворены (следовательно и он сможет завершиться) и как итог обе вершины p1, p2 станут изолированы. Такой редукцией определяется, что система не находится в тупике (вершины процессов стали изолированы).



Ситуация может быть небезопасной и тогда никакого освобождения ресурсов быть не может. Пример графа - ниже.



## Теоремы

- Граф является полностью сокращаемым, если существует такая последовательность сокращений, которая устраниет все дуги. Если граф нельзя полностью сократить, то анализируемое состояние является тупиком (доказательства есть в книге *Логическое программирование ОС*, мы не рассматриваем)  
В последнем примере, например, видно, что не хватает свободных единиц ресурса.
- Цикл в графе повторно используемых ресурсов является необходимым условием тупика (Тупик существует, если существует цикл (или как по-другому говорят: *замкнутая цепь запросов*))
- Если состояние системы  $S$  не является состоянием тупика и  $S \xrightarrow{[i]} T$  ( $i$  должно быть над стрелочкой, Марк Даун не умеет так; обозначает переход из состояния  $S$  в состояние  $T$ ), то в случае, если операция процесса  $r_i$  есть запрос и в  $T$   $r_i$  находится в тупике, состояние  $T$  является состоянием тупика (проще: *возникновение тупика в системе возможно только в результате запроса*)

## 16 билет

16.1 Взаимодействие параллельных процессов: проблемы; монопольный доступ и взаимоисключение; взаимодействие параллельных процессов в распределенных системах – особенности; централизованный алгоритм, распределенный алгоритм; синхронизация логических часов (алгоритм Лампорта).

16.1 (2021) Формулировка другая, суть та же.

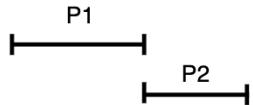
### Параллельные процессы

Взаимодействие параллельных процессов это тема которая раскрывает базовые понятия, базовые проблемы связанные действительно с взаимодействием параллельных процессов. Очевидно что такие же проблемы возникают и при взаимодействии потоков, но здесь имеются отличия. Дело в том, что **каждый процесс имеет собственное защищённое адресное пространство**, потоки своих адресных пространств не имеют и выполняются в

адресном пространстве процесса, поэтому потоки могут разделять глобальные переменные, процессы разделять глобальные переменные не могут.

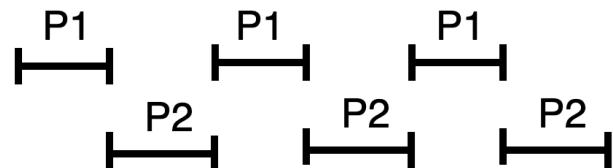
### 1. Последовательное выполнение.

От начала до конца выполняется процесс P1, потом от начала до конца выполняется процесс P2.



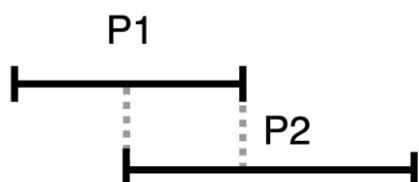
### 2. Квазипараллельное выполнение.

Квант времени выполняются команды процесса P1, потом происходит переключение и квант времени выполняются команды процесса P2. В общем то последовательное выполнение, но с точки зрения наблюдателя - параллельное. Такое выполнение называется квазипараллельным.



### 3. Реально параллельное выполнение.

Совпадение по времени выполнения команд процесса P1 и команд процесса P2.



## Проблемы

Рассмотрим два варианта (квазипараллельное и реально параллельное выполнение):

Оба процесса выполняют одну и ту же последовательность команд.

P1:		P2:
mov eax, myvar		mov eax, myvar
inc eax		inc eax
mov myvar, eax		mov myvar, eax

*Наши компьютеры имеют SMP-архитектуру (многоядерная архитектура, где каждое ядро это процессор со своими регистрами, в каждом ядре полный набор регистров)*

Понятно что каждый процессор будет иметь свой регистр EAX. Все процессоры работают с одной памятью.

В начальный момент значение переменной myvar = 0, затем инициативу перехватывает первый процесс и выполняет "mov eax, myvar", соответственно в eax попадает 0. Далее расписывать не буду по картинке всё видно.

2 ядра				
P1 на процессоре 1	eax	myvar	eax	P2 на процессоре 2
-	-	0	-	-
mov eax, myvar	0	0	-	-
-	0	0	0	mov eax, myvar
-	0	0	1	inc eax
-	0	1	1	mov myvar, eax
inc eax	1	1	1	-
mov myvar, eax	1	1	1	-

Мы потеряли данные.

Рассмотрим эту же ситуацию для однопроцессорной машины (Для двух потоков). Здесь не может быть двух регистров eax, он один.

1 ядро				
P1 на процессоре 1	eax	myvar	eax	P2 на процессоре 2
-	0	0	-	-
mov eax, myvar	0	0	-	-
Выполняется P1	0	0	-	-
P1 вытеснен, его контекст сохранён.	0	0	mov eax, myvar	-
-	0	1	inc eax	-
-	1	1	mov myvar, eax	-
P2 вытеснен, его контекст сохранён.	1	1	-	-
inc eax	1	1	-	-
mov myvar, eax	1	1	-	-

Ситуация повторилась, несмотря на то что это квазипараллельность, но мы видим что переключение связано с переключением контекста, с запоминанием содержимого регистров, соответственно мы получили тот же самый результат.

Подобные действия принято называть критическими.

Часть кода в котором выполняются такие действия принято называть критической секцией.

Myvar - разделяемая переменная.

Что же делать чтобы не терять данные? Для этого необходимо обеспечить монопольное использование разделяемых параллельными процессами ресурсов.

**Монопольный доступ и Взаимоисключение // вроде монопольный доступ еще и выше описано, так что....**

## **Взаимоисключения**

- 1. Программный.**
- 2. Аппаратный.**
- 3. С использованием семафоров.**
- 4. С использованием мониторов.**

### **1. Программный**

Рассмотрим классическое предложение - использование флага (неверное).

Пример реализации 1:

```
Var flag1, flag2: Boolean;  
p1: while(1)           |   p2: while(1)  
{                      |   {  
    while(flag2==1);    |       while(flag1==1);  
    flag1 = 1;          |       flag2 = 1;  
    CR1;               |       CR2;  
    flag1=0;            |       flag2 = 0;  
    PR1;               |       PR2;  
}                      |   }  
....                  ....  
// начальные установки  
flag1=0;flag2=0;  
parbegin  
p1;p2;  
parend;
```

*Пусть инициативу перехватил Р2. Ему надо попасть в свой критический участок, проверяет, флаг не установлен, теряет инициативу. Процесс 1 – также, + устанавливает свой флаг, входит в критическую секцию, изменяет переменную, сбрасывает флаг и идет дальше. Опять процесс 2, восстановил АК, продолжает со следующей команды, устанавливает, КС.*

**ГЛАВНОЕ СКАЗАТЬ, ЧТО ВАЙЛ НЕ ПРОВЕРЯЕТ УСЛОВИЕ ВТОРОЙ РАЗ**

Давайте поменяем последовательность действий, прежде чем проверять флаг другого процесса, сразу устанавливать свой флаг. (Спасибо конечно, но опять неправильно будет)

Пример реализации 2:

```
Var flag1, flag2: Boolean;  
p1: while(1)           |   p2: while(1)  
{                      |   {  
    flag1 = 1;          |   flag2 = 1;  
    while(flag2==1);   |   while(flag1==1);  
    CR1;               |   CR2;  
    flag1=0;            |   flag2 = 0;  
    PR1;               |   PR2;  
}  
}                      |   }  
....                  ....  
// начальные установки  
flag1=0;flag2=0;  
parbegin  
p1;p2;  
parend;
```

ТУТ ВСЕ ПРОСТО – ДЕДЛОК (то есть тупик)

### Алгоритм Декера

Декер предложил решение чисто программным способом, алгоритм исключает те негативные ситуации которые мы рассмотрели, является надёжным, исключает попадание процессов в тупик, исключает бесконечное откладывание.

Но данную задачу Декер решил только для двух параллельных процессов.

Для того чтобы решить эту проблему Декер ввёл третью переменную, которую назвал "чья очередь", мы назовём "que".

```

P1:                                | P2:                                | // Объявления и начальные установки
  while(1)                         | begin                                | flagP1, flagP2: logical;
  begin                               |   flagP2 = 1;                         | que:int;
    flagP1 = 1;                      |   while(flagP1);                     | flagP1 = 0; flagP2 = 0;
    while(flagP2);                  |   begin                                | que = 1;
    begin                               |     if (que == 1) then                | parbegin
      if (que == 2) then              |       begin                                |   P1; P2;
      begin                               |         flagP2 = 0;                   |     parend;
        flagP1 = 0;                   |         while(que == 2);             |
        while(que == 2);             |         flagP2 = 1;
        flagP1 = 1;                   |       end;
      end;                            |     end;
    end;                            |   CR2;
    CR1;                            |   flagP2 = 0;
    flagP1 = 0;                      |   que = 1;
    que = 2;                        |   PR2;
    PR1;                            | end;
  end;                            |

```

К чисто программному решению относится так же **известнейший алгоритм Лампорта (Lamport), который получил название "Булочная" ("Bakery")**.

Есть классическое изложение алгоритма Лампорта которое было предложено в статье "A new Solution of Dijkstra's concurrent programming problem". Этот алгоритм решает проблему критической секции для n прикладных процессов.

Основная идея взята из работы булочной.

Потребители берут номера (устанавливается очередь), тот кто имеет наименьший номер обслуживается следующим (вхождение в критическую секцию).

**Сточки зрения процессов:** Если два процесса входят в так называемую дверь, то они получают одинаковые номера, но затем при обслуживании будет учитываться, например, идентификатор процесса. Т.е. возникает необходимость дополнительных данных.

### **В системе 3 негативные ситуации:**

1. **Бесконечное откладывание.**
2. **Тупиковая ситуация.**
3. **Захват и освобождение одних и тех же ресурсов.**

**Примером такой ситуации является trashing (когда процесс не может загрузить в память всё своё рабочее множество и в силу этого постоянно выгружаются и загружаются снова одни и те же страницы).**

### **2.Аппаратно**

**Команда test-and-set (появилась в IBM370) Неделимая команда, которая выполняет проверку и установку содержимого ячейки памяти, которая часто называется байтом блокировки. 0 - ресурс доступен, 1-занят**

Неделимая команда test-and-set читает значение из b, копирует в переменную a, а затем для b устанавливает значение истина. Все в рамках одной неделимой операции

```
int Test_and_Set (int *target){  
    int tmp = *target;  
    *target = 1;  
    return tmp;  
}
```

**Цикл активного ожидания на процессоре** – негативный факт данного способа реализации взаимоисключения – процессорное время тратится на проверку переменной, тратятся кванты, полезная работа процессом при этом не выполняется

Данный способ не исключает бесконечное откладывание, но его вероятность мала, так как другой процесс наиболее вероятно сможет перехватить инициативу.

Testandset в цикли называется циклической блокировкой или spin lock (simple lock, простая блокировка, simple mutex (mutual exclusion)). Команда Testandset активно используется в ядре ОС.

### 3. Семафоры

Были предложены Дейкстром. В 1965 была опубликована первая статья, где он выдвинул идею семафоров. Что подвигло его на это? Стремление избежать активного ожидания.

Активное ожидание приводит к непроизводительным тратам процессорного времени.

Процесс циклится проверяя какую-то переменную, тратит на это выделенное ему процессорное время, фактически квант, соответственно пока другой процесс не выйдет из своего критического участка.

Семафор – это неотрицательная, защищённая переменная, на которой определены две неделимые операции: P(s) – пропустить ("р" - первая буква аналогичного слова в датском) и V(s) – освободить (аналогично, первая буква). Если семафор может принимать два значения 0 или 1, то такой семафор называется бинарным.

P(s) - декрементирует s:

s = s - 1, если s > 0

если s = 0, то декремент невозможен и процесс блокируется на s до тех пор, пока декремент не станет возможен.

V(s) - инкрементирует s:

s = s + 1

То есть освобождение заблокированного процесса, поскольку s становится > 0.

s: semaphore

```
// p1  
while (1) {  
    ...
```

```

P(s);
CR1:
V(S)
PR1;
...
}

// p2
while (1) {
    ...
P(s);
CR2:
V(S)
PR2;
...
}

```

// начальные установки

s = 1;

parbegin

p1; p2;

parend

#### **4.Мониторы**

monitor - программное средство разработанное Хоаром.

Задача мониторов структурировать программные средства решения задачи взаимоисключения при взаимодействии параллельных процессов. При этом надо понимать, что речь идет об асинхронных параллельных процессах.

Это процессы, которые выполняются с собственной скоростью. В итоге нельзя предсказать, в какой момент, в какую точку придет конкретный процесс.

Можно сказать, что монитор это средство организации взаимодействия и взаимоисключения параллельных асинхронных процессов.

*При этом монитор содержит как данные, так и подпрограммы, которые эти данные обрабатывают (ключевое слово - монитор). Монитор защищает свои данные благодаря тому, что доступ к данным возможен только с помощью функций монитора. Монитор может быть языковым средством, то есть может быть реализован в каком-то конкретном ЯП, а может быть реализован в системе. Особенно сложные задачи взаимодействия стоят перед системными процессами, поэтому мониторы часто являются средством структурирования операционных систем.*

В основе мониторов, как правило, лежат два системных вызова: `wait` и `signal`, которые определены на переменной типа условие (`conditional`). *По своей структуре похож на класс. Но мониторы появились раньше.*

*Собственно, в отличие от класса, ключевым словом является монитор, но вы видите, также монитор содержит данные и функции, которые могут обращаться к этим данным.*

Таким образом монитор защищает данные. В результате сам монитор является ресурсом. Для того, чтобы обратиться к данным монитора, процесс должен вызывать функцию монитора. Процесс, вызвавший функцию монитора называется процессом, находящимся в мониторе. Но при этом, к одному и тому же монитору могут обращаться большое кол-во процессов. И таким образом, к монитору будет организована очередь.

Монитор обслуживает произвольное число процессов, ограниченное только размером очереди.

В мониторе только две процедуры `acquire` (приобретать) и `release` (освобождать). Когда к монитору обращаются для захвата ресурса, то используется функция `acquire`. Если логическая переменная `busy = истина`, то по переменной `x` типа условие выполняется команда `wait()` и значение логической переменной `busy` не меняется. Ну системный вызов `wait` блокирует процесс на переменной `x`. Само слово `wait` означает блокировку. Если значение `busy = ложь`, то процесс, вызвавший `acquire`, получит доступ к ресурсу без задержки и для переменной `busy` установит значение `true`.

### **Взаимодействие параллельных процессов в распределенных системах**

Когда процессы взаимодействуют друг с другом, часто возникает проблема синхронизации процессов. Данная проблема связана с некорректным разделением критических ресурсов несколькими процессами и в следствии их утратой. Для решения этой проблемы, необходимо обеспечить процессу монопольный доступ к критическому ресурсу, до тех пор, пока процесс его не овободит.

#### **централизованный алгоритм**

- Существует процесс-координатор. Когда процесс хочет в критический участок, он отправляет запрос координатору с именем этого участка.
- Координатор анализирует, не находится какой-либо процесс в данной критической секции. В случае, если находится, то запрос ставится в очередь. В обратном случае, процесс получает разрешение на вход в критический участок.
- Если два процесса одновременно хотят войти в критический участок, то выбор происходит с помощью алгоритма Лампорта.

Минус: процесс-координатор может прекратить свое существование.

В случае, если какой-либо процесс обнаружит отсутствие координатора (таймаут), то он инициирует выборы нового. Процесс посылает сообщение с предложением выбора нового координатора всем процессам, указывая свой собственный. Если процесс получатель обнаруживает, что его номер больше, то он посылает назад подтверждение приема такого сообщения и сам инициирует новые выборы.

В результате выполнения останется один процесс с самым большим номером, который становится координатором

### **распределенный алгоритм + токен ring**

1. Когда процесс хочет войти в критическую секцию, он формирует сообщение.
  - В этом сообщении он указывает:
    - а. идентификатор нужной ему критической секции;
    - б. свой номер (идентификатор);
    - с. время формирования сообщения по своим локальным часам.
  - Предполагается, что передача сообщения надёжна: получение каждого сообщения сопровождается подтверждением.
  - Протокол - соглашение о том, какие действия выполняют процессы при передаче и получении сообщения. Надёжный протокол всегда предполагает подтверждение получения сообщения.
2. Это сообщение, сформированное процессом, процесс рассыпает всем взаимодействующим процессам (то есть  $n - 1$  сообщение)
3. Получив такое сообщение, процесс проверяет, в какой ситуации он сам находится по отношению к критической секции.
  - Возможны 3 ситуации:
    - а. получивший сообщение процесс не собирается входить в данную критическую секцию (и не находится в ней). В этом случае процесс посыпает сообщение-ответ с разрешением войти в критический участок;
    - б. процесс, получивший сообщение, находится в данной критической секции. В этом случае никакое сообщение не посыпается;
    - с. Процесс, получивший сообщение, сам формировал сообщение запрос на вход в данную критическую секцию, но еще не успел это сделать. В этом случае процесс проверяет временную метку создания своего сообщения, сравнивает ее с временной меткой полученного сообщения. Если его сообщение было сформировано раньше, то никакого сообщения-ответа процесс не посыпает. Если в результате таких рассылок процесс получает  $n - 1$  сообщение подтверждения входа в критический участок, то процесс может войти в критический участок. Если он не получает хотя бы одно сообщение, то в критический участок процесс войти не может.

В этом случае процесс должен послать  $n - 1$  сообщение-запрос и в ответ получить  $n - 1$  сообщение-разрешение на вход, иначе войти не может.

Централизованный алгоритм надежнее чем распределенный, за счет возможности выбора нового координатора.

В случае распределенного алгоритма: Если какой-то из процессов перестал существовать, то процесс, разославший  $n - 1$  сообщение-запрос, не сможет получить  $n - 1$  сообщение-ответ: потеря работоспособности по конкретной критической секции.

## ТОКЕН\_РИНГ

Смысл заключается в том, что процессы создают логическое кольцо - замкнутую цепочку.

Каждый процесс в логическом кольце знает идентификатор следующего процесса в этом логическом кольце. По данному логическому кольцу передается так называемый токен. Это специальное сообщение, которое содержит идентификатор конкретной критической секции. Этот токен переходит от  $n$ -ого процесса к  $n + 1$ -ому процессу, циркулирует по логическому кольцу. Когда процесс получает токен, он анализирует, нужно ли ему войти в данную критическую секцию. Если нужно, получив токен, он входит в критическую секцию и токен удерживает. Выйдя из критической секции, процесс отправляет токен следующему в цепи процессу. Пока процесс находится в критической секции он удерживает токен. Если ни один процесс не заинтересован во входлении в критическую секцию токена, такой токен быстро циркулирует по цепочке.

## СРАВНЕНИЕ

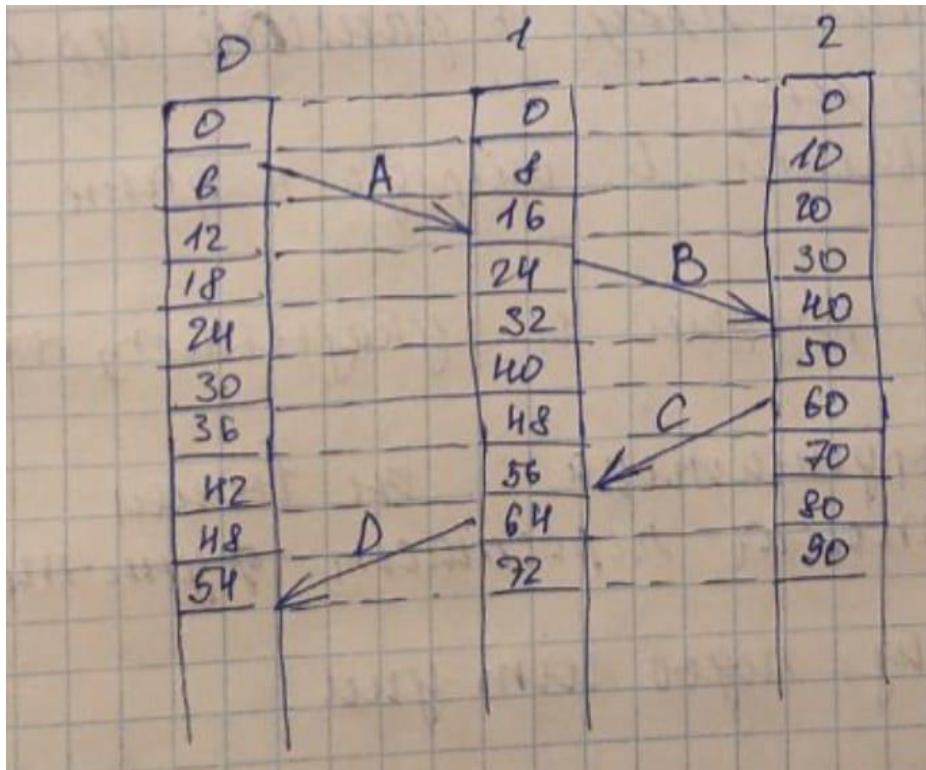
Самый надежный алгоритм: централизованный алгоритм. За счет возможности выбора нового координатора.

В случае распределенного алгоритма: Если какой-то из процессов перестал существовать, то процесс, разославший  $n - 1$  сообщение-запрос, не сможет получить  $n - 1$  сообщение-ответ: потеря работоспособности по конкретной критической секции.

В случае Токен Ринг: если какой-то процесс перестает существовать, то цепь разрывается. Чтобы ее восстановить, надо предпринять соответствующие действия. Если циркулирование токена по каким-то причинам прервано, то есть только одно средство определения: таймаут.

## синхронизация логических часов (алгоритм Лампорта)

В распределенных системах очень важно, чтобы обеспечивалось отношение случилось до / случилось после. Для того, чтобы быть уверенными, Лампортом был разработан алгоритм, который он назвал timestamps (Логические часы Лампорта)

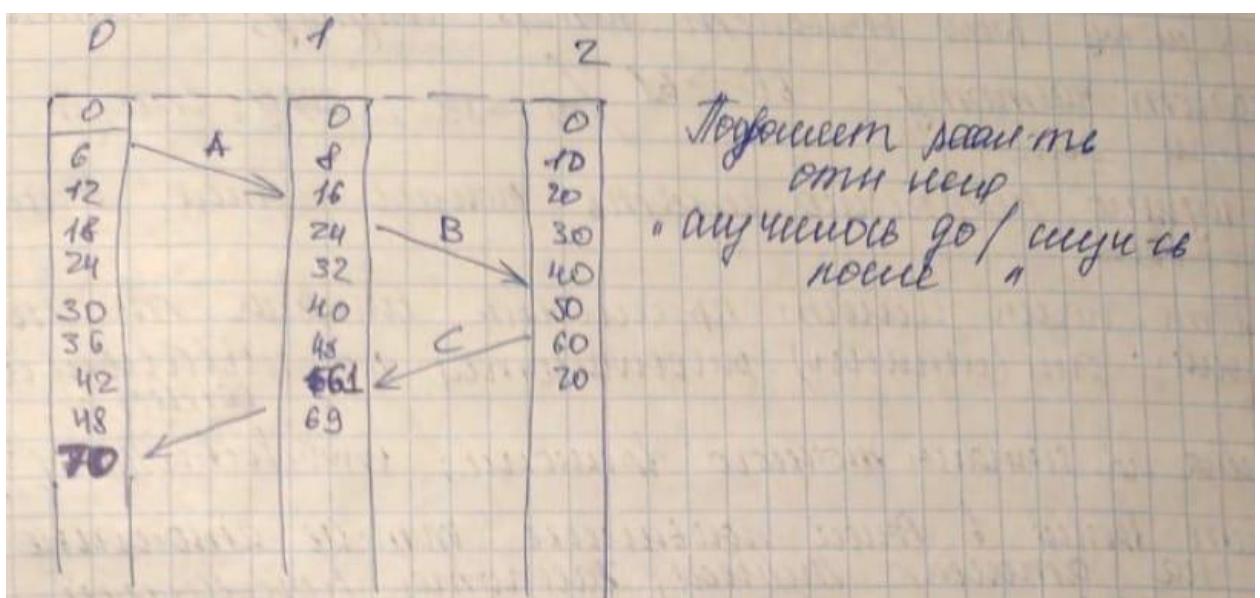


Процесс 0 в какой-то момент времени решает послать процессу 1 сообщение. Понятно, что отправка сообщения не может производиться позже его получения. То есть отправка всегда выполняется во времени раньше чем получение.

6 -> 16 - A все хорошо 24 -> 40 - B все хорошо 60 -> 56 - C ошибка 64 -> 54 - D ошибка

То есть мы не можем обеспечить отношение случилось до/случилось после.

Лампорт предложил в сообщение добавить временную отметку о локальном времени отправителя. Получатель сравнивает свое время с тем временем, что пришло с сообщением. Если его время меньше, то он устанавливает собственное время равное пришедшему + 1



6 -> 16 - А все хорошо (корректировка не требуется) 24 -> 40 - В все хорошо (корректировка не требуется) 60 -> 56 - С ошибка (время корректируется и вместо 56 становится 60 + 1) 69 -> 54 - Д ошибка (время корректируется и вместо 54 становится 69 + 1)

Это позволяет реализовать точное отношение случилось до/случилось после.

## НА ВСЯКИЙ РАЗЛИЧИЯ СЕМАФОРОВ И МЬЮТИКСОВ

*Фактически мьютекс представляет из себя (ближе всего) бинарный семафор, но есть важнейшие отличия:*

1. У мьютекса есть понятие владельца. Владельцем является процесс, захвативший мьютекс. И только владелец может освободить мьютекс. У семафора же нет владельца. Для семафора возможна следующая запись: (у меня её нет). Захватывает семафор один процесс, а освобождает совсем другой. Это самая главное различие.
2. Мьютексы предоставляют так называемую инверсию приоритетов безопасности. Мьютекс знает своего владельца и если на мьютексе блокируется другой более приоритетный процесс, то на самом деле приоритет процесса, захватившего мьютекс, повышается. То есть невозможно перехватить мьютекс.
3. Процесс, захвативший мьютекс не может быть случайно удалён, завершён. А захвативший семафор может.

## 16.2 Процессы в UNIX: системные вызовы fork(), exec(), wait(), signal() – примеры из лабораторных работ. (из 2021 + особенности выполнения)

Процесс-сирота — процесс с завершенным предком. Если предок завершен, процесс сирота усыновляется терминальным процессом (id = 1).

Процесс-зомби — процесс, у которого отобраны все ресурсы, кроме последнего — строки в таблице процессов. Это сделано для того, чтобы процесс-предок, вызвавший системный вызов wait(), не был заблокирован навсегда.

Процессы демоны — процессы которые не имеют родителей, они существуют сами по себе и не входят ни в какие группы. Демон — процесс, выполняющий какую-то фоновую задачу, не имеющий управляющего терминала и, как следствие, обычно не интерактивный по отношению к пользователю.

### fork()

Fork – создается новый процесс процесс–потомок, который является копией процесса предка в том смысле, что потомок наследует код предка, дескрипторы открытых файлов, сигнальную маску окружения.

В старых системах код предка копировался в адресное пространство потомка. То есть потомку создавалось собственное адресное пространство. Надо создать соответствующие таблицы – таблицы сегментов, а в современных системах это таблицы страниц.

Таблицы страниц описывают адресное пространство процесса.

Очевидно, что это крайне неэффективно. В системе могут одновременно существовать какое-то количество копий одной и той же программы. Поэтому в современных системах существуют два способа оптимизации задачи создания нового процесса. (называют иногда оптимизация fork)

1. 1. 2 способ оптимизации – macos (unix bsd) реализован способ с системным вызовом vfork. В этом случае для потомка не создаются карты, а предок предоставляет потомку свои. При этом предок блокируется до того момента, пока потомок не вызовет exit/exec
2. Предложен в system5 и называется копирование при записи (copy on write). Когда вызывается fork и создается потомок, для него создаются собственные карты трансляции адресов (в современных – таблицы страниц), и эти таблицы страниц ссылаются на страницы адресного пространства предка. При этом для страниц стека ... права меняются на only read и устанавливается флаг copy on write. Если предок или потомок пытаются изменить страницу, возникает исключение по правам доступа. Обрабатывая его, система обнаружит установленный флаг copy on write и создаст копию нужной страницы в адресном пространстве того процесса, который пытался ее изменить.

Дескриптор этой страницы должен быть добавлен в соответствующую таблицу страниц. ОС не может работать с неопределенными значениями. В результате будут созданы только копии нужных страниц.

Решение copy on write решило проблему коллективного использования страниц и страничное преобразование (то есть управление памятью страницами по запросу) стало фактически единственным используемым

**Ситуация, когда изменены права доступа к данным и стеку предка и установлен флаг copy on write существует до тех пор, пока процесс потомок не вызовет или системный вызов exit(), или системный вызов exec().**

ПРИМЕР ИЗ ЛР (СОЗДАНИЕ 2x ПРОЦЕССОВ)

```
int child[N];  
printf("Parent process. PID: %d, GROUP: %d\n", getpid(), getpgrp());  
  
for (int i = 0; i < 2; i++)  
{  
    int pid = fork();
```

```

if (-1 == pid)
{
    return 1;
}
else if (0 == pid) // дочерний код
{
    sleep(2);

    printf("Child process #%.d. PID: %d, PPID: %d, GROUP: %d\n", i + 1, getpid(),
getppid(), getpgrp());

    return 0;
}
else // родительский код
{
    child[i] = pid;
}
}

```

### **exec()**

Exes() создает так называемый низкоуровневый процесс. Почему так называемый? – потому что на самом деле процесс не создается.

Процесс создается через fork – с идентификатором и дескриптором. А exec для программы, которая передается exec в качестве параметра, создает адресное пространство, то есть - таблицу страниц. Но перед этим он проверяет права доступа процесса (child) к данному файлу, (child вызывает exec()), проверяется путь к файлу (существует ли файл?) и является ли данный файл исполняемым

Вообще в системе 3 вида файлов - исполняемый, объектный, исходный. Exes может быть передан только исполняемый файл.

После этого создается адресное пространство, но у child уже оно есть в результате fork – его надо уничтожить, чтобы не занимали память. После этого в дескрипторе процесса должна быть строка, содержащая адрес таблицы страниц, надо поменять адрес в этой строке на адрес новой таблицы страниц.

Чтобы программа начала выполняться, надо загрузить адрес точки входа. То есть системный вызов exec переводит процесс на новое адресное пространство.

(Системный вызов exec создает таблицу страниц для адресного пространства программы, которую передали как формальный параметр exec, после этого заменяет одну таблицу на другую путем замены адреса.) – что в скобка на 4 лабе по почте принял Рязанова (относится к вопросу, а что за “перевод на новое адресное пространство”

## ПРИМЕР ИЗ ЛР

```
int child[2];
int pid;
const char *const commands[N] = { "ls", "pwd" };

printf("Parent process. PID: %d, GROUP: %d\n", getpid(), getpgrp());

for (size_t i = 0; i < 2; i++)
{
    pid = fork();

    if (-1 == pid)
    {
        return 1;
    }
    else if (0 == pid)
    {
        sleep(2);
        int rc = execlp(commands[i], commands[i], 0);
        if (-1 == rc)
        {
            return 1;
        }

        return 0;
    }
    else
    {
        child[i] = pid;
    }
}
```

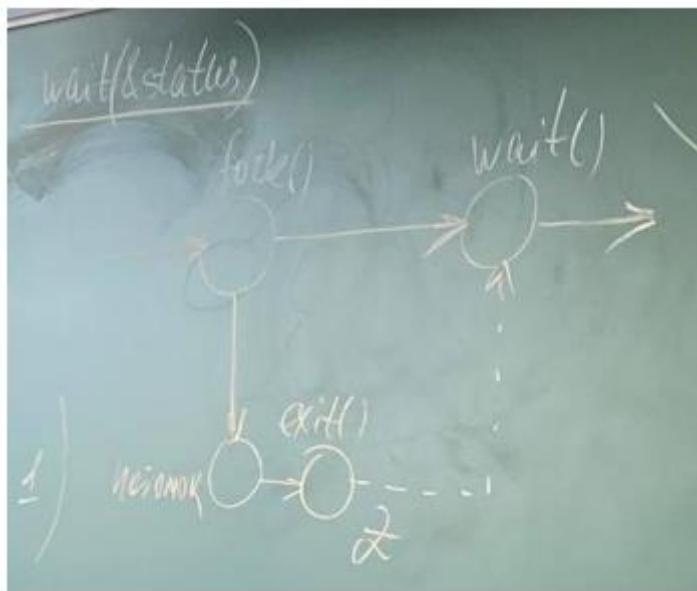
## wait()

Системный вызов `wait(&status)` блокирует родительский процесс до того момента, пока не будет завершен дочерний. При завершении, процесс получает статус завершения потомка. Интерпретировать информацию о состоянии процесса можно с помощью специальных макросов объявленных `sys/wait.h`.

Чтобы не появлялись сироты, в unix есть системный вызов `wait(&status)`.

Система не контролирует, где вызван – в предке или потомке, потому что любой потомок может стать предком. Точно также она не контролирует, где вызывается `exec()`. Это дополнительные проверки, которые системы не интересны.

Но правильная логика – предок должен ждать завершения своих потомков: в предке `wait(&status)`. Предок может проконтролировать код завершения потомка и ветвиться в зависимости от этого. Но с `wait` связана одна проблема в системе.



Процесс выполняется и создает процесс-потомок. Тот аварийно завершился. А предок продолжал что-то делать, а только потом вызвал wait. Но предок уже не сможет получить инфу и останется навсегда заблокированным.

Это решается с помощью состояния зомби. В unix все процессы проходят через состояние зомби – процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов (то есть дескриптор).

Процесс переходит в состояние зомби и когда предок вызовет wait, он может увидеть статус завершения и продолжить свое управление

#### ПРИМЕР ИЗ ЛР

```

int status, statval = 0;
pid_t childpid = wait(&status);
printf("Child process (PID %d) finished. Status: %d\n", childpid, status);

if (WIFEXITED(statval))
{
    printf("Child process #%d finished with code: %d\n", i + 1, WEXITSTATUS(statval));
}
else if (WIFSIGNALED(statval))
{
    printf("Child process #%d finished from signal with code: %d\n", i + 1, WTERMSIG(statval));
}
else if (WIFSTOPPED(statval))
{
    printf("Child process #%d finished stopped with code: %d\n", i + 1, WSTOPSIG(statval));
}

```

#### signal()

Сигнал — способ информирования процесса ядром о произшествии какого-либо события. При возникновении нескольких однотипных событий, процессу передается только один сигнал. То есть сигнал означает, что событие произошло, но не уточняется, сколько таких событий произошло.

Процесс может установить собственную реакцию на получаемых сигнал. Это делается с помощью системного вызова `signal(snum, f)`.

Системный вызов `signal` не является стандартным для POSIX 1, но он определен в AMSI C и, следовательно, имеется во всех UNIX. POSIX - portable operating system interface – интерфейс переносимых ОС

*Системный вызов signal возвращает указатель на предыдущий обработчик данного сигнала и его можно использовать для восстановления предыдущего обработчика*

`snum` - номер сигнала, `f` - адрес функции, которая должна быть выполнена при поступлении указанного сигнала. Возвращает указатель на предыдущий обработчик данного сигнала, который можно использовать для восстановления обработчика сигнала.

(вопрос? что происходит с процессом, который получил сигнал?????????

## ПРИМЕР ИЗ ЛР

```
#define QUITE_MODE 0  
int MODE = QUITE_MODE;  
...  
void parent_callback(int sig_number)  
{  
    MODE = LOUD_MODE;  
}  
...  
signal(SIGINT, parent_callback);
```

ОЧЕНЬ СТРАННО ЧТО НЕТ `exit()`

`Exit()` – системный вызов завершения процесса

## 17 билет

17.1 Виртуальная память: распределение памяти страницами по запросам, свойство локальности, рабочее множество, анализ страничного поведения процессов. Схема страничного преобразования в процессорах Intel (X86) РАЕ – размеры таблиц дескрипторов.

17.1 (2021) Виртуальная память: распределение памяти страницами по запросам, алгоритмы вытеснения, свойство локальности, рабочее множество, анализ страничного

поведения процессов. Схема страничного преобразования в процессорах Intel (X86) PAE – дескрипторы и размеры таблиц дескрипторов, обоснование использование многоуровневого преобразование, кэш TLB – структура (процесс 486). (2022 вроде такой же)

Стали делить память на страницы. Можно выполнить программу, которая находится не целиком в памяти. Для этого нужно содержать части кода с которыми в текущий момент работает процессор. Это воплотилось в понятие виртуальная память.

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

### 3 схемы управления виртуальной памятью

4. Управление памятью страницами по запросу
5. Управление памятью сегментами по запросу
6. Управление памятью сегментами, поделенными на страницы по запросу.

**Виртуальная память – память, размер которой превышает размер реального физического пространства.**

Существуют три схемы управления виртуальной памятью:

1. управление памятью страницами по запросам;
2. управление памятью сегментами по запросам;
3. управление памятью сегментами, делёнными на страницы, по запросам.

#### **Управление памятью страницами по запросам**

Загрузка частей программы в память выполняется по запросу. Т.е. соответствие части кода загружаемого по запросу, когда процессор обращается к этим частям кода. Если в памяти отсутствует страница к которой было произведено обращение, то произойдет исключение.

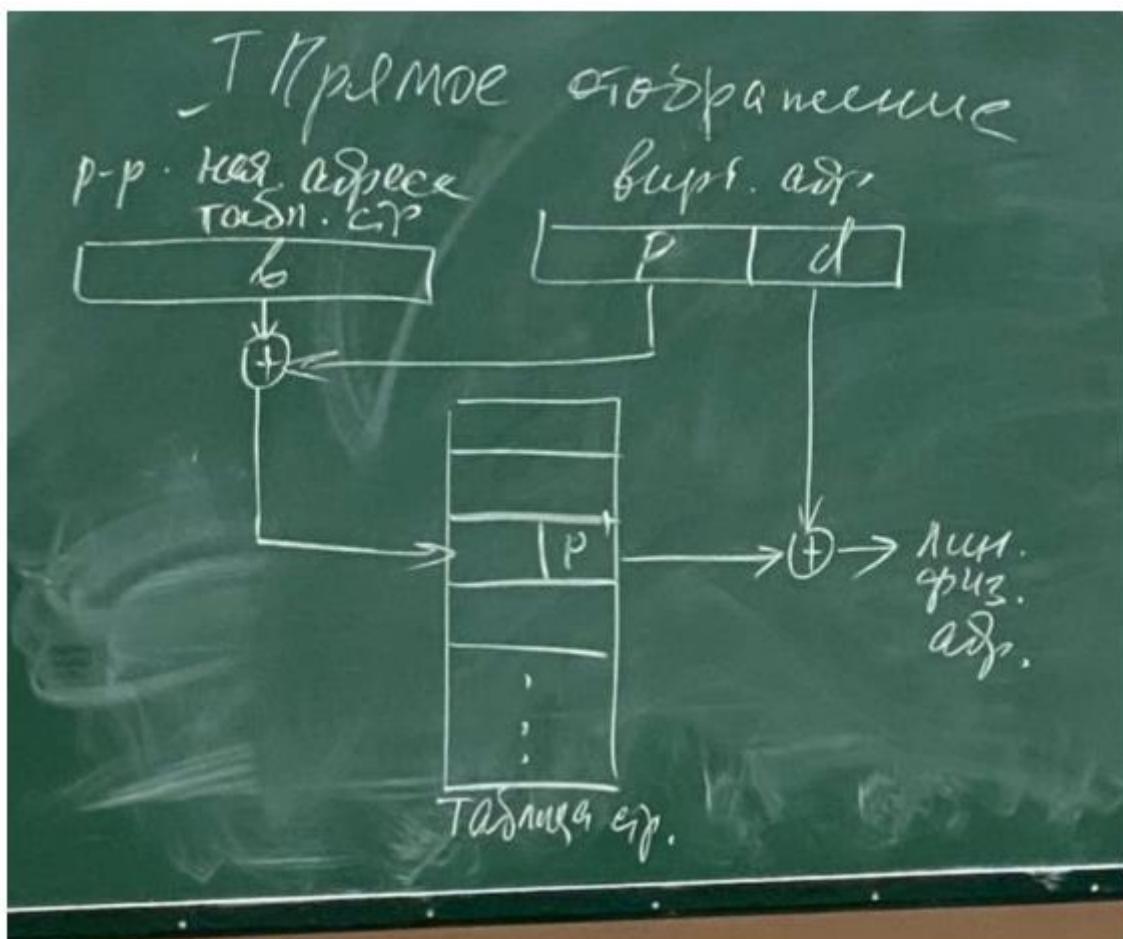
Адресное пространство процесса и адресное пространство физической памяти делится на блоки равного размера. Блоки, на которые делится адресное пространство процесса называют страницами, а блоки на которые делится физическая память – кадрами, фреймами или блоками.

Виртуальный адрес состоит из номера страницы и смещения страницы.

При выполнении программы, которая находится не целиком в памяти, процесс потребует страницу, которой нет в оперативной памяти - возникнет исключение (страничная неудача - исправимое исключение), которое будет обработано в режиме ядра. В результате менеджер памяти 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.

### 3 метода преобразования адресов (это не особо важно писать)

#### 1. Прямое отображение



Виртуальный адрес делится на 2 части – номер страницы и смещение.  $V=(p,d)$

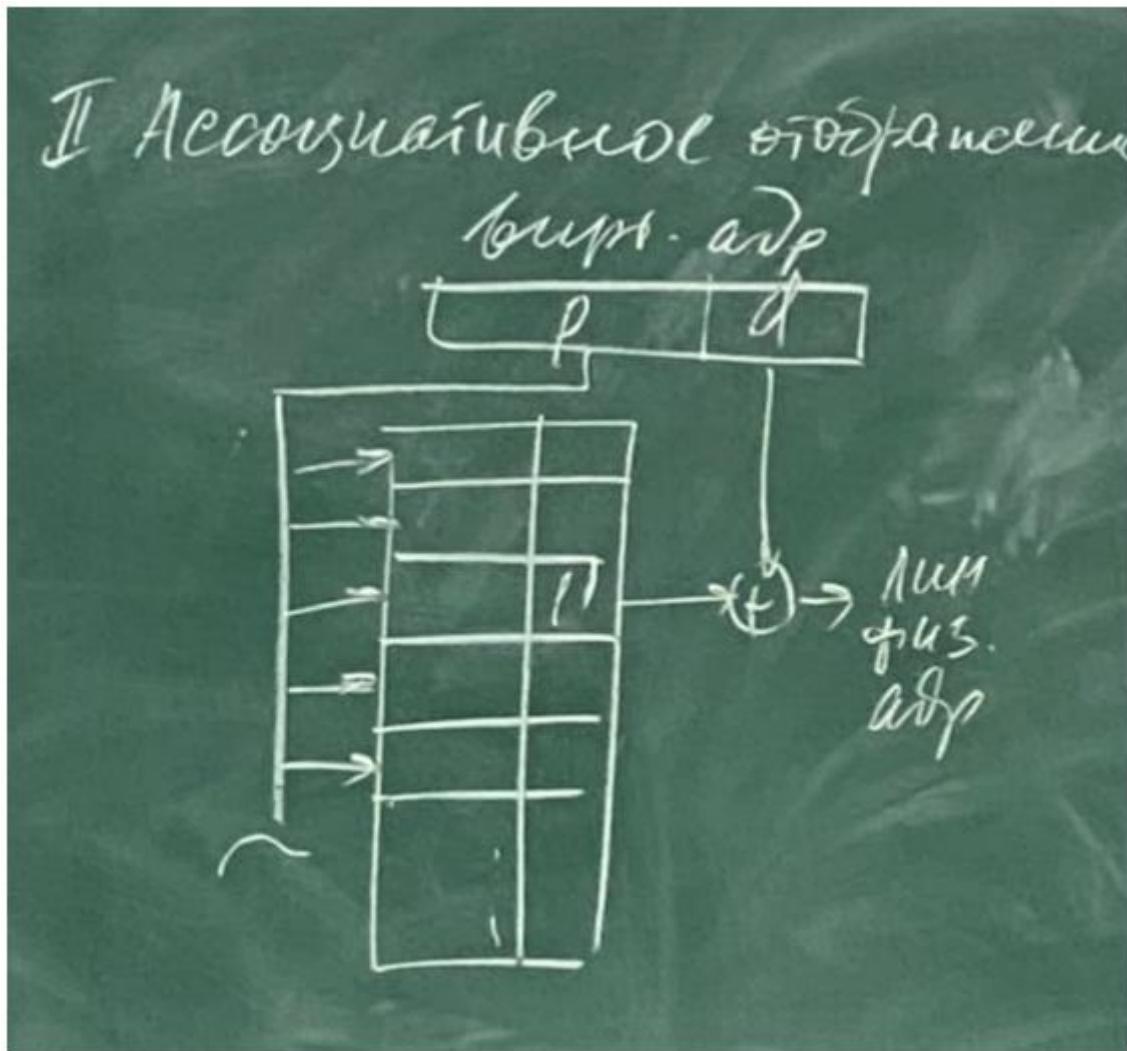
Часть виртуального адреса, которая называется номер страницы (страница) - смещение к дескриптору страницы в таблице страниц.

Если страница загружена в физическую память, то у этой страницы должен быть физический адрес. К адресу физической страницы добавляется смещение, в результате этого преобразования получаем линейный физический адрес (адрес конкретного байта физической памяти).

В дескрипторе страницы также содержится набор флагов, который позволяет работать с этой страницы. Если страницы виртуального адресного пространства и физической памяти одинаковые, то смещение в этих страницах будет одно и то же.

Таблица страниц находится в оперативной памяти, то есть для того, чтобы сформировать физический адрес команды/данных, надо обратиться к этой таблице в оперативной памяти. Обращение к памяти - крайне затратная операция. Поэтому был предложен способ ассоциативного отображения.

## 2. Ассоциативное отображение



Ассоциативное отображение предполагает использование специального вида памяти – ассоциативной памяти.

2 вида ассоциативной памяти:

- параллельная
- последовательная

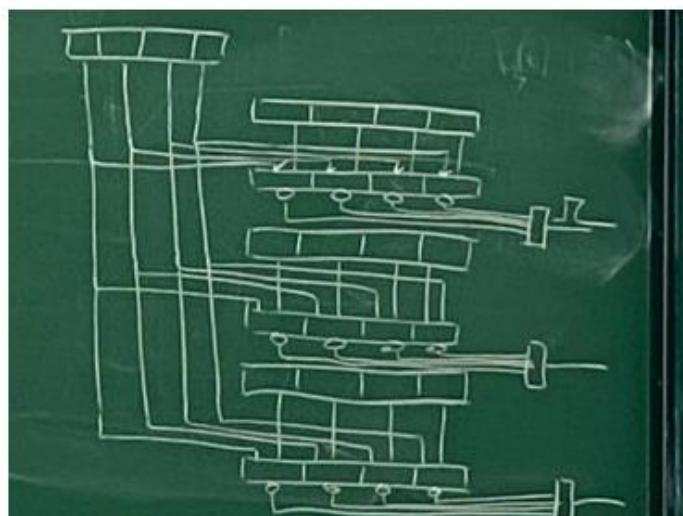
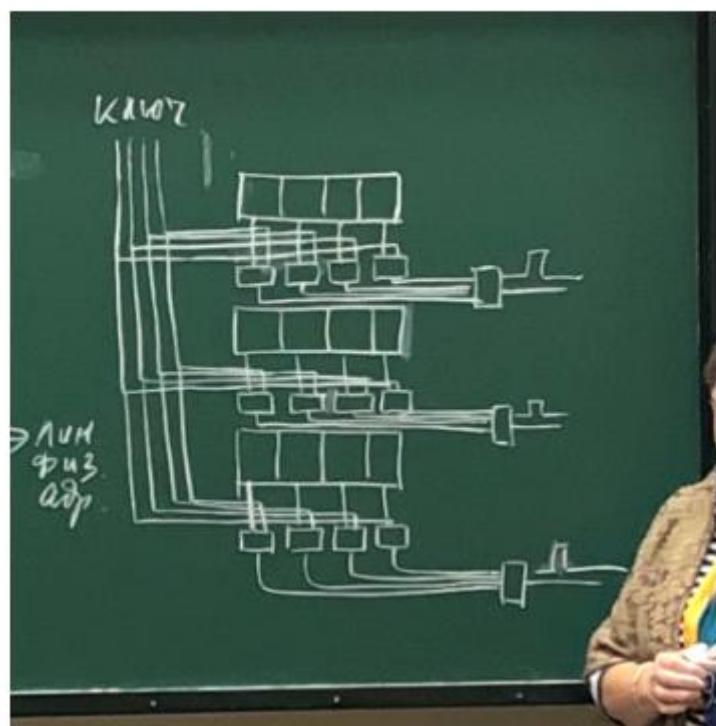
В параллельной ассоциативной памяти выборка данных осуществляется за 1 такт за счет одновременного сравнения так называемого ключа со всеми полями, позволяющими осуществлять такую выборку. В данном случае так называемый ключ - это номер страницы.

Виртуальный адрес поделен на 2 части.

Ассоциативная память - очень дорогая память. Это регистровая память. К тому же, в этой памяти число элементов удваивается из-за необходимости выполнения сравнения.

*Но это дорогая память – она регистровая + чтобы осуществить такое обращение к информации необходимо на каждый разряд соответствующего регистра поставить схему совпадения. Собрать все сигналы на соответствующую схему, и если все сигналы совпали – послать сигнал разрешения считывания*

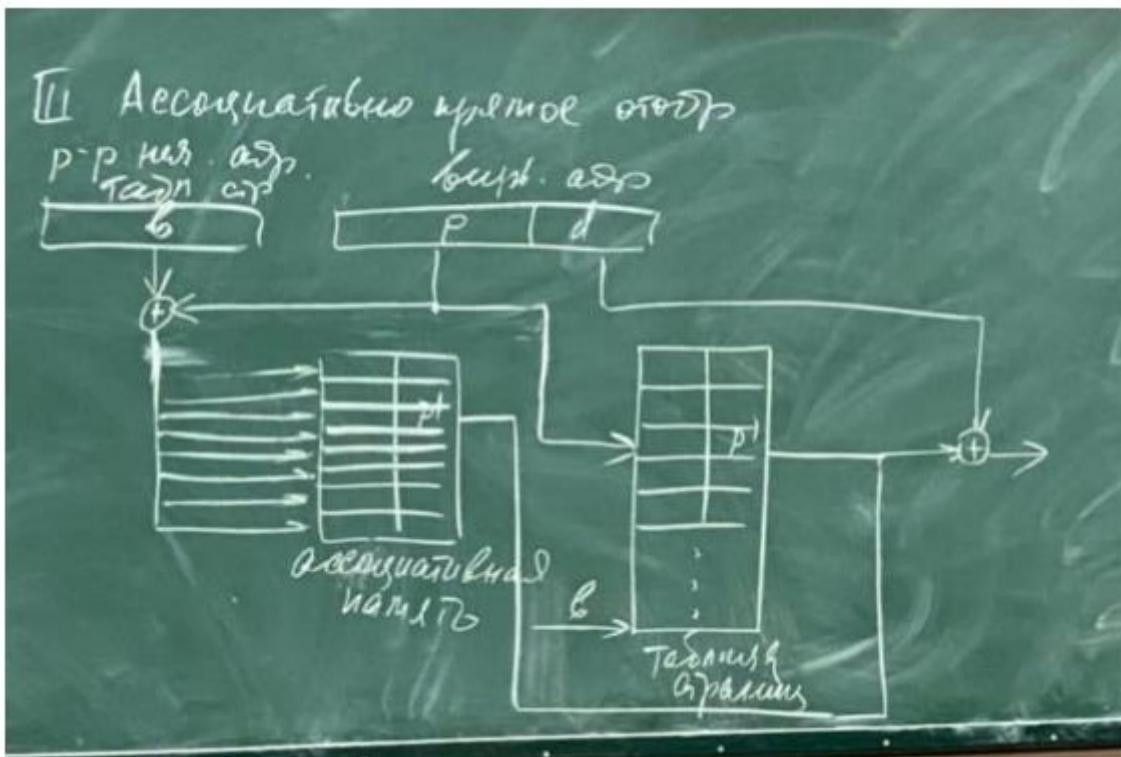
Выборка осуществляется за 1 такт за счет специальной схемы.



Если совпадение происходит по всем разрядам, то формируется разрешающий импульс и происходит считывание за один такт.

### 3. Ассоциативное-прямое отображение

Комбинация первых двух методов.



Ассоциативно-прямое отображение предполагает комбинацию первых двух методов. Существует и таблица страниц и так называемый ассоциативный кэш.

### Алгоритмы вытеснения

Если весь физический адрес занят, то нужно выгрузить страницу.

1. Выталкивание случайной (первой попавшейся) страницы (во вторичную память). Характеризуется малыми издержками, не является дискриминационным, но имеет недостатки: может быть вытолкнута часто используемая или только что загруженная.
2. Fifo. Каждой странице присваивается временная метка или организуется связный список типа очередь. То есть вновь загруженные страницы оказываются в хвосте и постепенно перемещаются в начало и та, что в начале – кандидат на выгрузку. Минусы: все еще возможно выталкивание часто используемой. (Но зато вновь загруженная уже не будет выгружена),

*а также «аномалия fifo» (чтобы посмотреть аномалию – можно у дейтала):  
наше априорное утверждение, что при увеличении объема доступной памяти  
количество прерываний страниц уменьшается, может оказаться ложным*

+ - отмечены страничные неудачи (внизу), если нет - страничная удача, а наверху – что надо загрузить. В кружок – что вытесняем

P	1	2	3	4	5	6	7	8	9	10	11	12
	4	1	3	1	2	1	1	4	1	5	1	5
$\uparrow$												
$M=3$												
$\downarrow$												
	4	3	2	1	4	3	3	3	5	2	2	
	(4)	(3)	(2)	(1)	4	4	4	(3)	5	5		
	+	+	+	+	+	+	+	+	+	+	+	

$9/12 = 75\%$

Оказывается, существуют такие траектории загрузки страниц, когда увеличение памяти приводит не к уменьшению числа страничных прерываний, а к их увеличению, как ни странно. То есть понятно, априорно можно предположить, что увеличение доступного объема физической памяти, то есть увеличение количества физических страниц, должно привести к уменьшению числа страничных прерываний.

Увеличим память на 1 страницу. Итого  $10/12 = 83\%$  Это и называется аномалия fifo

P	1	2	3	4	5	6	7	8	9	10	11	12
	4	1	3	1	2	1	1	4	1	5	1	5
$\uparrow$												
$M=4$												
$\downarrow$												
	4	3	2	2	2	1	5	4	3	2	1	
	4	3	3	3	2	1	5	4	3	2		
	4	4	(4)	(3)	(2)	(1)	(6)	(4)	3			
	+	+	+	+	+	+	+	+	+	+	+	

$10/12 \sim 83\%$

### 3. LRU - least recently used – наименее используемый в последнее время

Вернемся к FIFO и, опять же, отметим очевидные недостатки FIFO: то есть вытолкнуть только что загруженную страницу по FIFO нельзя, но какую страницу можно вытолкнуть? Часто используемую. Поэтому, естественно, стремление улучшить ситуацию привело к появлению вот этого алгоритма - least recently used - наименее используемая в последнее время.

Промоделируем работу этого алгоритма на той же траектории страниц

P	1	2	3	4	5	6	7	8	9	10	11	12
	4	1	3	1	2	1	4	(3)	5	1	4	3
$\uparrow$	4	1	3+2+	1+4+	3+5+	4	3	-2+	1+	5+		
$M=3$	4	3	2	1	4	3	5	4	3	2	1	
$\downarrow$	(4)	(3)	(2)	(1)	4	3	(5)	(4)	(3)	2		
	+	+	+	+	+	+	+	+	+	+	+	

Когда к странице обращаются, она отправляется в конец списка (в хвост). Если используются временные метки, то временная метка обновляется.

Так, на 8 шаге выполняется обращение к 4, она уже в памяти, поэтому перемещается в конец списка (если мы считаем, что клеточки – модель связного списка), то есть страничного прерывания не происходит (эта ситуация называется страничной удачей). Увеличим память на 1 страницу

P	1	2	3	4	5	6	7	8	9	10	11	12
	4	1	3	1	2	1	4	1	3	5	4	3
$\uparrow$	4	1	3+2+	1+4+	3+5+	4	3	2+	1+	5+		
$M=4$	4	3	2	1	4	3	5	7	3	2	1	
$\downarrow$	4	3	2	1	4	3	5	4	3	2		
	4	3	2	1	4	3	5	(1)	(5)	(1)	3	
	+	+	+	+	+	+	+	+	+	+	+	

Здесь также происходит страничная удача, поскольку 4 страница в памяти. Обращение к 3 приведет к тому, что она перемещается в хвост.

Как видно из рисунка, у нас все сработало в соответствии с нашими предположениями: увеличение памяти привело к уменьшению числа страничных прерываний. Это связано с тем, что алгоритм соответствует свойству локальности наших программ

Алгоритм LRU полностью соответствует свойству локальности.

Алгоритм LRU относится к классу методов вытеснения, которые называются **стековыми** алгоритмами. (был вопрос, LRU как стековый алгоритм)

Но затраты для его реализации, как я уже сказала, огромные. В чистом виде алгоритм LRU не используется. Почему он дает такие результаты? (ну кроме того, что он стековый) Он полностью соответствует свойству локальности, которое мы с вами сформулировали. Что наиболее вероятно обращение к странице, к которой мы обращались в последний момент времени. И если к странице долго не обращаться, то также очень вероятно, что к ней вообще не будет обращения.

**Никогда не приведет к росту числа страничных прерываний при увеличении памяти.**

*Несмотря на достоинства, в том виде, в котором мы его рассматриваем, этот алгоритм не используется, так как он крайне затратный. Для его реализации надо либо модифицировать временную метку при каждом обращении к странице, либо редактировать связный список, перемещая страницу в конец. При этом обращение к странице – на каждой команде, а то и несколько раз (именно так говорят) (обращение к команде, обращение к данным команды, а если косвенная адресация – там 2 обращения по поводу данных)*

4. LFU – least frequently used – наименее часто используемая страница

Часто к названию добавляют Page Replacement.

Здесь используется счетчик обращений к странице. Он инкрементируется при каждом обращении к странице. Есть очевидный недостаток – может быть вытеснена только что загруженная страница, так как она не набрала еще количества обращений.

5. NUR - Not used recently page replacement – страница, не используемая в последнее время.

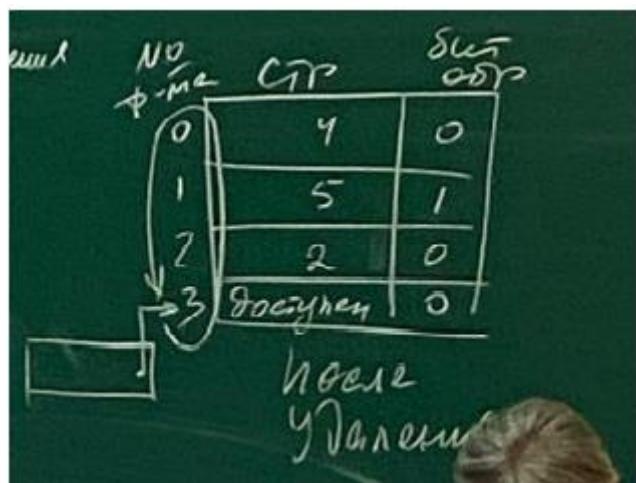
Это – аппроксимация алгоритма LRU. Для его реализации у каждой страницы вводится бит обращения к странице. Этот бит периодически сбрасывается в 0 (для всех страниц), а при обращении к странице – выставляется 1. Поэтому, если надо вытеснить какую-либо страницу, то она ищется среди тех, у которых этот бит сброшен. И это показывает, что с момента последнего сброса битов обращения, обращения к этой странице не было.

Для реализации этого алгоритма вводится указатель удаления.

Следующий рисунок показывает ситуацию перед удалением



А этот - после удаления



На первом рисунке – ситуация после загрузки 5 страницы в 1 кадр (frame). Если в этот момент необходимо удалить страницу, то проверка значений битов обращения будет выполняться со 2 кадра (фрейма), поэтому и показана цикличность. Бит обращения у 2 фрейма = 1 -> ее нельзя удалить. Идем дальше, у 3 фрейма бит обращения=0, поэтому первая страница, загруженная в 3 фрейм может быть удалена, что здесь и показано.

Кроме бита обращения вводится бит модификации. В наших системах он называется dirty (грязный). Делается, чтобы: какую страницу выгодно заместить – которая не модифицировалась, потому что тогда не нужно будет копирование, так как точная копия этой страницы находится во вторичной памяти.

Значит, вводятся 2 бита – обращения и модификации. Тогда возможны 4 ситуации

бит ообр	бит модиф
0	0
0	1
1	0

Вызывает вопрос вторая строка. Как так – обращения не было, а модификация была?  
Значит, эта страницы модифицировалось до сброса всех битов обращения в 0

### свойство локальности

#### **рабочее множество**

[Теория рабочего множества](#)

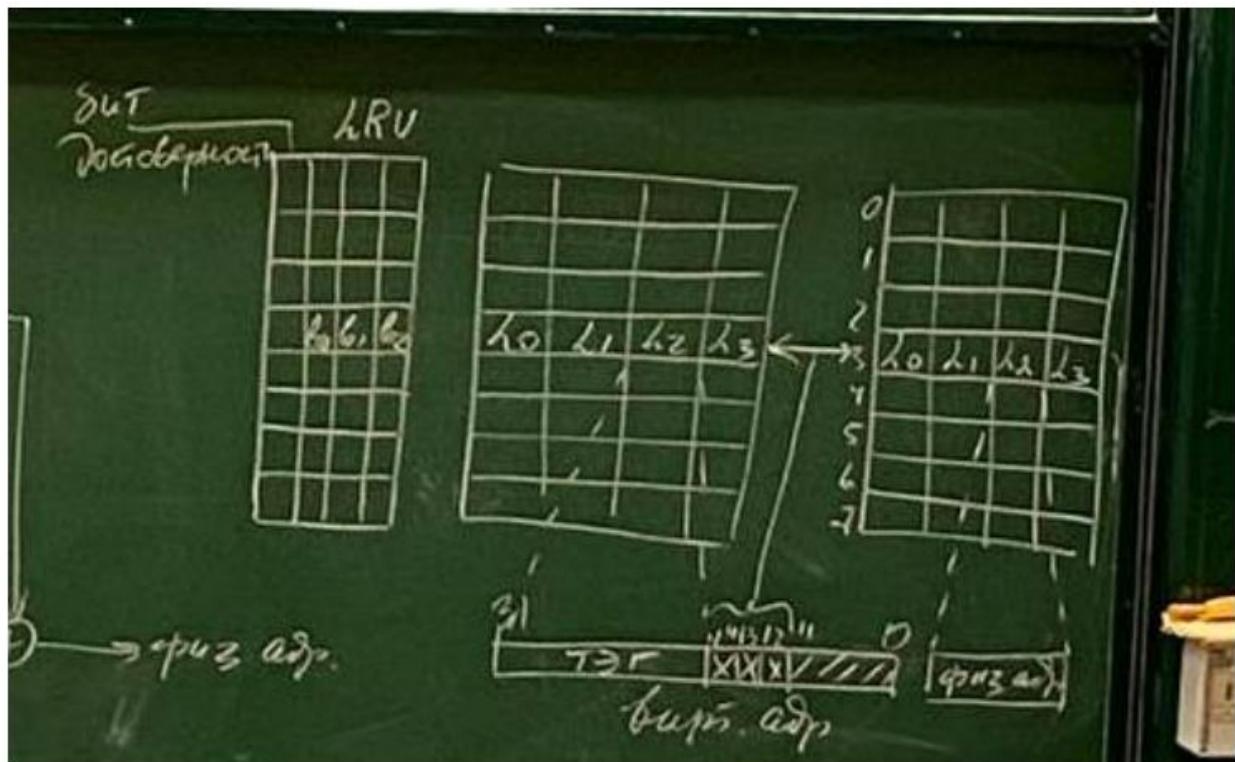
#### анализ страничного поведения

**Схема страничного преобразования в процессорах Intel (X86) PAE – размеры таблиц дескрипторов.**

[Управление памятью страницами по запросам в архитектурах x86 – расширенное преобразование \(PAE\) – схема преобразований.](#)

#### **кэш TLB – структура (процесс 486).**

Но в процессорах интел есть cash, который называется (TLB) для 486 процессора, сейчас – намного больше, там вообще 2 кеша l1, l2 и +l3 в кристалле, содержат инфу по последним обращениям, что позволяет использовать многоуровневые ТС и эффективней использовать ОП, увеличивать возможность адресации



Мы обращаемся к tLB по виртуальному адресу, и получаем физические адреса страниц, к которым были последние обращения.

Сначала страница ищется в TLB, если не найдена, то выполняется обращение к таблицам страниц в ОП и происходит замещение. 3 бита используются для определения множества, а уже в множестве – acc. выборка, то есть это частично ассоциативный кеш.

Смещение в физической и виртуальной странице одно и то же, оно просто прибавляется к начальному адресу физической страницы `base_address`, как написано выше.

Замещение – по алгоритму псевдо LRU. Для этого – блок достоверности LRU- первый достоверности и 3 бита для определения множества.,.

При очистке кеша или сбросе процессора все биты достоверности сбрасываются в 0. Когда производится заполнение строки кеша, ищется любая недостоверная строка.

Если таковых нет, то замещаемой строку выбирают по алгоритму псевдо LRU. Эти биты модифицируются при каждом попадании (страничная удача) или заполнении следующим образом:

- если обращение в множестве было к строке 10 или 11, то бит P0 устанавливается в 1, если к 12 / 13 – сбрасывается в 0
- если обращение в паре к 1110: 10 то бит p1=1, 11 p1=0
- 1213: 13 – b2=0, 12 – b2=1

Утверждается, что в типичных системах cash tlb удовлетворяет до 99% запросов на доступ к таблицам страниц. При этом (актуальность кеша – именно текущие физические адреса TLB Очищается при загрузке CR3 (при переключении на выполнение другого процесса)).

17.2 Прерывания от системного таймера в защищенном режиме: функции (по материалам лабораторной работы).

**На консультации она говорила, что UNIX/Windows работают в Long mode'е и типа это не защищенный режим, но тут написано "по материалам лабораторной работы"... Ничего не понятно....**

#### UNIX

По тику:

- инкрементирует счетчик тиков аппаратного таймера;
- декременирует квант текущего потока;
- обновляет статистику использования процессора текущим процессом — инкремент поля `c_cpri` дескриптора текущего процесса до максимального значения 127;
- инкрементирует счетчики часов и других таймеров системы;
- декрементирует счетчик времени до отправления на выполнение отложенного вызова (если счетчик достиг нуля, то выставление флага для обработчика отложенного вызова).

По главному тику:

- регистрирует отложенные вызовы функций, относящиеся к работе планировщика, такие как пересчет приоритетов;
- пробуждает в нужные моменты системные процессы, такие как `swapper` и `pagedaemon`. Под пробуждением понимается регистрация отложенного вызова процедуры `wakeup`, которая перемещает дескрипторы процессов из списка "спящих" в очередь готовых к выполнению.
- декрементирует счётчик времени, оставшегося времени до посылки одного из следующих сигналов:
  - `SIGALRM` — сигнал, посылаемый процессу по истечении времени, предварительно заданного функцией `alarm()`;
  - `SIGPROF` — сигнал, посылаемый процессу по истечении времени заданного в таймере профилирования;
  - `SIGVTALRM` — сигнал, посылаемый процессу по истечении времени, заданного в "виртуальном" таймере.

По кванту:

- посыпает текущему процессу сигнал `SIGXCPU`, если тот превысил выделенную ему квоту использования процессора.

## Windows

По тику:

- инкрементирует счётчик системного времени;
- декрементирует квант текущего потока на величину, равную количеству тактов процессора, произошедших за тик (если количество затраченных потоком тактов процессора достигает квантовой цели, запускается обработка истечения кванта);
- декрементирует счетчики времени отложенных задач;
- если активен механизм профилирования ядра, инициализирует отложенный вызов обработчика ловушки профилирования ядра путем постановки объекта в очередь DPC: обработчик ловушки профилирования регистрирует адрес команды, выполнявшейся на момент прерывания.

По главному тику:

- освобождает объект "событие", который ожидает диспетчер настройки баланса.

По кванту:

- инициализирует диспетчеризацию потоков путем постановки соответствующего объекта в очередь DPC.

17.2 (2021) Адресация прерываний от системного таймера в защищенном режиме (схема).  
(2022 вроде такой же) + формат GDT и IDT

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHth-sB8TTy1Xvv9SN9zM/edit#bookmark=id.2v9zt489i2bq> –это схема

Таблица дескрипторов (GDT)

Структура, описывающая дескриптор GDT

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHth-sB8TTy1Xvv9SN9zM/edit#bookmark=id.3bs5x4j4p98i> – это все про IDT

## 18 билет

18.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – флаги, алгоритм Деккера, алгоритм Лампорта (уточнение 2021 алгоритм Булочная) .

### Параллельные процессы

Взаимодействие параллельных процессов это тема которая раскрывает базовые понятия, базовые проблемы связанные действительно с взаимодействием параллельных процессов.

Очевидно что такие же проблемы возникают и при взаимодействии потоков, но здесь имеются отличия. Дело в том, что **каждый процесс имеет собственное защищённое адресное пространство**, потоки своих адресных пространств не имеют и выполняются в адресном пространстве процесса, поэтому потоки могут разделять глобальные переменные, процессы разделять глобальные переменные не могут.

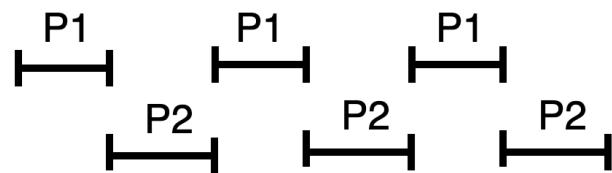
### 1. Последовательное выполнение.

От начала до конца выполняется процесс P1, потом от начала до конца выполняется процесс P2.



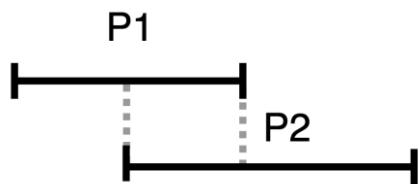
### 2. Квазипараллельное выполнение.

Квант времени выполняются команды процесса P1, потом происходит переключение и квант времени выполняются команды процесса P2. В общем то последовательное выполнение, но с точки зрения наблюдателя - параллельное. Такое выполнение называется квазипараллельным.



### 3. Реально параллельное выполнение.

Совпадение по времени выполнения команд процесса P1 и команд процесса P2.



## Проблемы

Рассмотрим два варианта (квазипараллельное и реально параллельное выполнение):

Оба процесса выполняют одну и ту же последовательность команд.

P1:		P2:
mov eax, myvar		mov eax, myvar
inc eax		inc eax
mov myvar, eax		mov myvar, eax

Наши компьютеры имеют SMP-архитектуру (многоядерная архитектура, где каждое ядро это процессор со своими регистрами, в каждом ядре полный набор регистров)

Понятно что каждый процессор будет иметь свой регистр EAX. Все процессоры работают с одной памятью.

В начальный момент значение переменной myvar = 0, затем инициативу перехватывает первый процесс и выполняет "mov eax, myvar", соответственно в eax попадает 0. Далее расписывать не буду по картинке всё видно.

2 ядра				
P1 на процессоре 1	eax	myvar	eax	P2 на процессоре 2
-	-	0	-	-
mov eax, myvar	0	0	-	-
-	0	0	0	mov eax, myvar
-	0	0	1	inc eax
-	0	1	1	mov myvar, eax
inc eax	1	1	1	-
mov myvar, eax	1	1	1	-

Мы потеряли данные.

Рассмотрим эту же ситуацию для однопроцессорной машины (Для двух потоков). Здесь не может быть двух регистров eax, он один.

1 ядро				
P1 на процессоре 1	eax	myvar	eax	P2 на процессоре 2
	eax			
P1 на процессоре 1		myvar		P2 на процессоре 2
		0		
Выполняется P1				
mov eax, myvar	0	0		
P1 вытеснен, его контекст сохранён.				
		0	0	mov eax, myvar
		0	1	inc eax
		1	1	mov myvar, eax
P2 вытеснен, его контекст сохранён.				
inc eax	1	1		
mov myvar, eax	1	1		

Ситуация повторилась, несмотря на то что это квазипараллельность, но мы видим что переключение связано с переключением контекста, с запоминанием содержимого регистров, соответственно мы получили тот же самый результат.

Подобные действия принято называть критическими.

Часть кода в котором выполняются такие действия принято называть критической секцией.

Myvar - разделяемая переменная.

Что же делать чтобы не терять данные? Для этого необходимо обеспечить монопольное использование разделяемых параллельными процессами ресурсов.

### **Взаимодействие – монопольный доступ взаимоисключение**

#### **Взаимоисключения**

1. Программный.
2. Аппаратный.
3. С использованием семафоров.
4. С использованием мониторов.

#### **Программная реализация**

##### **Флаги**

Рассмотрим классическое предложение - использование флага (неверное).

##### **Пример реализации 1:**

```
Var flag1, flag2: Boolean;  
  
p1: while(1) | p2: while(1)  
{ | {  
    while(flag2==1); | while(flag1==1);  
    flag1 = 1; | flag2 = 1;  
    CR1; | CR2;  
    flag1=0; | flag2 = 0;  
    PR1; | PR2;  
}  
} | }  
.... ....  
// начальные установки  
flag1=0;flag2=0;  
parbegin  
p1;p2;  
parend;
```

Пусть инициативу перехватил Р2. Ему надо попасть в свой критический участок, проверяет, флаг не установлен, теряет инициативу. Процесс 1 – также, +

устанавливает свой флаг, входит в критическую секцию, изменяет переменную, сбрасывает флаг и идет дальше. Опять процесс 2, восстановил АК, продолжает со следующей команды, устанавливает, КС. Флаги не являются защищой критической секции, это не работает.

Предложение: установить влаг до while

**Пример реализации 2:**

```
Var flag1, flag2: Boolean;

p1: while(1)           |   p2: while(1)
{
    flag1 = 1;          |   flag2 = 1;
    while(flag2==1);   |   while(flag1==1);
    CR1;                |   CR2;
    flag1=0;             |   flag2 = 0;
    PR1;                |   PR2;
}
}

....                   ....
// начальные установки
flag1=0;flag2=0;
parbegin
p1;p2;
parend;
```

Та же последовательность. Устанавливает флаг и теряет квант. Получает первый, устанавливает и застrevает в цикле ожидания по флагу второго процесса. Все застряло. Процессы попали в deadlock – тупиковая ситуация

**В системе 3 негативные ситуации:**

1. Бесконечное откладывание.
2. Тупиковая ситуация.
3. Захват и освобождение одних и тех же ресурсов.

Примером такой ситуации является trashing (когда процесс не может загрузить в память всё своё рабочее множество и в силу этого постоянно выгружаются и загружаются снова одни и те же страницы).

**Деккера**

Декер – голландский математик, решил только для 2 параллельных процессов.

Декер предложил решение чисто программным способом, алгоритм исключает те негативные ситуации которые мы рассмотрели, является надёжным, исключает попадание процессов в тупик, исключает бесконечное откладывание.

Но данную задачу Декер решил только для двух параллельных процессов.

Для того чтобы решить эту проблему Декер ввёл третью переменную, которую назвал "чья очередь", мы назовём "que".

```
P1:                                | P2:                                | // Объявления и начальные установки
  while(1)                         | begin                                | flagP1, flagP2: logical;
  begin                               |   begin                                | que:int;
    flagP1 = 1;                      |     flagP2 = 1;                         | flagP1 = 0; flagP2 = 0;
    while(flagP2);                  |       while(flagP1);                   | que = 1;
    begin                               |         begin                            | parbegin
      if (que == 2) then             |           if (que == 1) then            |   P1; P2;
      begin                           |             begin                        | parend;
        flagP1 = 0;                  |               flagP2 = 0;              |
        while(que == 2);             |                 while(que == 1);          |
        flagP1 = 1;                  |                   flagP2 = 1;            |
      end;                             |             end;                     |
    end;                             |           end;                     |
    CR1;                            |           CR2;                     |
    flagP1 = 0;                      |           flagP2 = 0;              |
    que = 2;                          |           que = 1;                |
    PR1;                            |           PR2;                     |
  end;                                |         end;                     |

```

Пусть инициатива у 2 процесса. П2, устанавливает свой флаг, если флаг первого занят, проверяет, чья очередь.

Первого – сбрасывает свой флаг и в цикл ожидания по переменной que. Первый процесс, выходя из своего критического участка, изменит que.

П2 установит свой флаг, войдет и выйдет из Критического участка, сбросит флаг и установит que=1. Нет тупиковой ситуации. Но решение только для 2 процессов.

Принято указывать в этих задачах, что процессы выполняются параллельно с помощью `parbegin`, `parend`

Как мы видим такое решение избавляет от deadlock-а, и поскольку процесс передаёт активность другому процессу, то исключается бесконечное откладывание.

### Лампорта (Булочная)

К чисто программному решению относится так же известнейший алгоритм Лампорта (Lamport), который получил название "**Булочная**" ("Bakery").

Есть классическое изложение алгоритма Лампорта которое было предложено в статье "A new Solution of Dijkstra's concurrent programming problem". Этот алгоритм решает проблему критической секции для  $n$  прикладных процессов.

Основная идея взята из работы булочной.

Потребители берут номера (устанавливается очередь), тот кто имеет наименьший номер обслуживается следующим (вхождение в критическую секцию).

Алгоритм прост и основан на методе, обычно используемом в булочных, где покупатель получает номер при входе, следующим обслуживается обладатель наименьшего номера, при этом может возникнуть такая ситуация, когда два покупателя одновременно входят в дверь. В этом случае они получают одинаковые номера, но первым будет обслужен покупатель у которого меньший номер паспорта.

*С точки зрения процессов:* Если два процесса входят в так называемую дверь, то они получают одинаковые номера, но затем при обслуживании будет учитываться, например, идентификатор процесса. Т.е. возникает необходимость дополнительных данных.

## ПСЕВДОКОД

```
1. var choosing: shared array[0: n - 1] of boolean;
2.     number: shared array[0: n - 1] of integer;
3. repeat
4.     choosing[i] := true;
5.     number[i] := max(number[0], ..., number[n - 1]) + 1;
6.     choosing[i] := false;
7.     for j := 0 to n - 1 do begin
8.         while choosing[j] do (*nothing*);
9.         while number[j] <> 0 and (number[j], j) < (number[i], i) do (*nothing*);
10.    end;
11.    critical section;
12.    number[i] = 0;
13.    (*remaining section*);
14.    until false;

// 1-2: Объявляются два массива.
// choosing[i] будет true если процесс Pi выбирает номер.
// Этот номер будет использоваться процессом для входа в критическую секцию.
// Если номер = 0, то процесс не пытается войти в критическую секцию.

// 4-6: Процесс выбирает номер (максимальный из выданных + 1)

// 7-12: Осуществляется выбор процесса, который входит в критический участок.
// Процесс P[i] ждёт, пока не отработают все процессы с меньшим номером. Если два
// процесса имеют один номер, используется лексиграфический порядок.
```

**18.2 Защищенный режим: перевод компьютера в защищенный режим – последовательность действий, реализация – пример кода из лабораторной работы.**

### **защищенный режим**

Защищенный режим - 32-х разрядный режим. В нём 32-х разрядные регистры и 32-х разрядная шина адреса (в основном).

Компьютер в защищенным режиме может работать или в собственно защищенном режиме или в режиме виртуального 86-ого процессора.

V86 - специальный защищенный режим, в котором запускаются виртуальные машины реального режима. В каждой такой машине может выполняться 1 программа реального режима. В защищенном режиме поддерживается виртуальная память.

### **перевод (полный, как из ЛР)**

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHthsB8TTy1Xvv9SN9zM/edit#bookmark=id.iwi16dp9jd3w> – перевод

### **реализация**

**запрет маскируемых и немаскируемых прерываний и разрешение**

[Маскируемые и немаскируемые прерывания](#)

### **перепрограммирование ведущего контроллера**

- Система команд исполнителя (СКИ)

```
; сохранение масок
in al, 21h
mov mask_master, al          ; ведущий
in al, 0A1h
mov mask_slave, al           ; ведомый
; перепрограммирование ведущего контроллера
mov al, 11h ; СКИ1 - два контроллера в компьютере -> Будет СКИЗ
out 20h, al
mov al, 32 ; СКИ2 - базовый вектор 32 (был 8)
out 21h, al
mov al, 4   ; СКИЗ - ведомый подключен в уровень2 ведущего
out 21h, al
mov al, 1   ; СКИ4 - 8086, требуется EOI требуется
out 21h, al

; маска для ведущего контроллера
```

```
    mov al, 0FCh
    out 21h, al

; маска для ведомого контроллера (запрещаем прерывания)
    mov al, 0FFh ; 1111 1111 - запрещаем все!
    out 0A1h, al
```

### **открытие и закрытие линии A20 (2 способа)**

; открытие линии A20 (если не откроем, то будут битые адреса, будет пропадать 20ый бит)а

```
    mov      AL, 0D1h
    out     64h, AL
    mov      AL, 0dfh
    out     60h, AL
; закрытие линии A20
;(если не закроем, то сможем адресовать еще 64кб памяти (HMA, см. сэм))
;A20
    mov      AL, 0D1h
    out     64h, AL
    mov      AL, 0ddh
    out     60h, AL
```

и еще один способ , но ненадежный

In al, 0x92

Or al, 2

Out 0x92, al

Начиная с PC/2 имеется быстрый (2) вариант открытия линии a20. Он исключает опрос. Быстрый вариант считается не вполне надежным и поддерживается не всеми платформами. Невозможно убедиться в этом заранее. Поэтому пользоваться надо 1 вариантом

### **НА ВСЯКИЙ схема адресации аппаратных прерываний**

<https://docs.google.com/document/d/1GZmNA96V2xYDTW69pKxmPkUHth-sB8TTy1Xvv9SN9zM/edit#bookmark=id.2v9zt489i2bq>

## 19 билет

19.1 Процессы Unix: создание процесса в ОС Unix и запуск новой программы.

Примеры программ из лабораторных работ, демонстрирующих эти действия.

Системные вызовы `wait()` и `pipe()`: назначение, примеры из лабораторных работ.

Процессы «сироты», «зомби» и «демоны».

**создание процесса**

[fork\(\)](#)

**запуск новой программы**

[exec\(\)](#)

[wait\(\)](#)

[pipe\(\)](#)

Системный вызов `pipe()` создает неименованный программный канал.

Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор. Процесс-потомок наследует все дескрипторы открытых файлов процесса-предка, в том числе и неименованных программных каналов.

Программные каналы имеют встроенные средства взаимоисключения — массив файловых дескрипторов: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают. Для этого определяется массив файловых дескрипторов.

Пример создания программного канала:

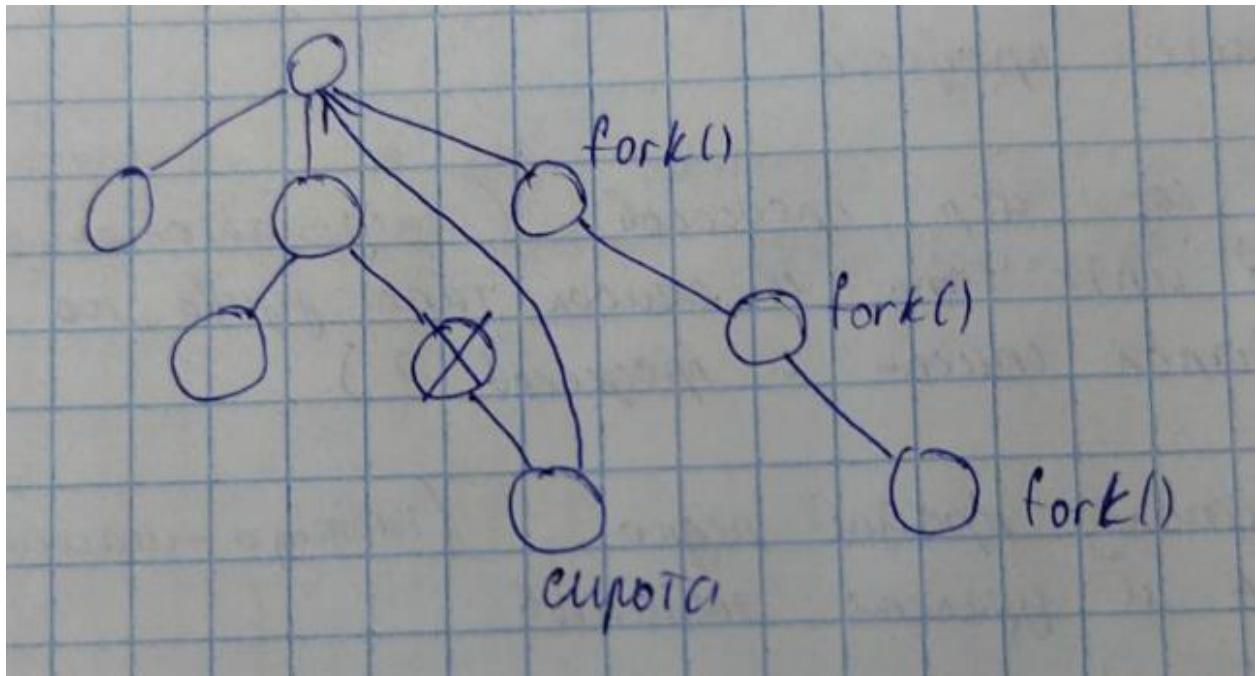
```
int fd[2];
int pid;
if (pipe(fd) == -1) {
    printf("Something went wrong with pipe!");
    return 1;
```

Пример чтения/записи с помощью программного канала:

```
const char* SECRET_MSGS[2] = { "secret message 1", "secret message 2" };
...
size_t written = write(fd[1], SECRET_MSGS[0], strlen(SECRET_MSGS[0])); // запись
...
char buffer[BUFFER_LEN] = { 0 };
size_t buf_len = read(fd[0], buffer, BUFFER_LEN); // чтение
```

## сироты

Процесс-сирота - это процесс, родитель которого завершился раньше него самого. Для сохранения иерархии предок-потомок, в случае завершения процесса-предка в системе выполняется так называемое усыновление: процесс-потомок усыновляется процессом с id=1 (процессом «открывшим» терминал и создавшим терминальную группу). (примечание как пользователя ubuntu в ubuntu усыновляет процесс, открывший терминал)



## зомби

Процесс-зомби – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов. Это сделано для того, чтобы процесс-предок, вызвавший системный вызов `wait()`, не был заблокирован навсегда. Можно увидеть переход процесса в состояние зомби, если дочерний процесс завершиться первым, то он будет существовать как зомби, пока процесс-предок или вызовет системный вызов `wait()`, или родительский процесс завершиться.

В Unix все процессы проходят через состояние зомби – процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов (то есть дескриптор)

## демоны

Процессы демоны – процессы которые не имеют родителей, они существуют сами по себе и не входят ни в какие группы. Демон – процесс, выполняющий какую-то фоновую задачу, не имеющий управляющего терминала и, как следствие, обычно не интерактивный по отношению к пользователю.

(инет)Демон - это процесс, работающий в фоновом режиме и находится вне контроля пользователя. Это значит, что демон не связан с терминалом, с помощью которого можно было бы им управлять.

[19.2 Взаимодействие параллельных процессов: мониторы – определение; монитор Хоара «читатели-писатели», реализация в для ОС Windows – пример из лабораторной работы..](#)

### Параллельные процессы

Взаимодействие параллельных процессов это тема которая раскрывает базовые понятия, базовые проблемы связанные действительно с взаимодействием параллельных процессов. Очевидно что такие же проблемы возникают и при взаимодействии потоков, но здесь имеются отличия. Дело в том, что **каждый процесс имеет собственное защищённое адресное пространство**, потоки своих адресных пространств не имеют и выполняются в адресном пространстве процесса, поэтому потоки могут разделять глобальные переменные, процессы разделять глобальные переменные не могут.

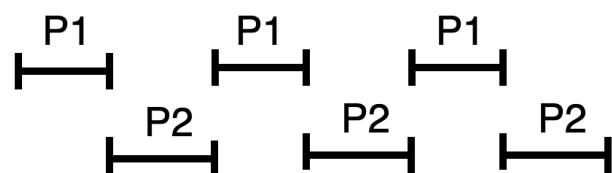
#### 1. Последовательное выполнение.

От начала до конца выполняется процесс P1, потом от начала до конца выполняется процесс P2.



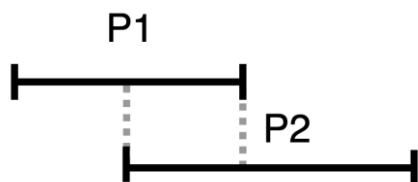
#### 2. Квазипараллельное выполнение.

Квант времени выполняются команды процесса P1, потом происходит переключение и квант времени выполняются команды процесса P2. В общем то последовательное выполнение, но с точки зрения наблюдателя - параллельное. Такое выполнение называется квазипараллельным.



#### 3. Реально параллельное выполнение.

Совпадение по времени выполнения команд процесса P1 и команд процесса P2.



## Проблемы

Рассмотрим два варианта (квазипараллельное и реально параллельное выполнение):

Оба процесса выполняют одну и ту же последовательность команд.

P1:		P2:
mov eax, myvar		mov eax, myvar
inc eax		inc eax
mov myvar, eax		mov myvar, eax

Наши компьютеры имеют SMP-архитектуру (многоядерная архитектура, где каждое ядро это процессор со своими регистрами, в каждом ядре полный набор регистров)

Понятно что каждый процессор будет иметь свой регистр EAX. Все процессоры работают с одной памятью.

В начальный момент значение переменной myvar = 0, затем инициативу перехватывает первый процесс и выполняет "mov eax, myvar", соответственно в eax попадает 0. Далее расписывать не буду по картинке всё видно.

2 ядра				
P1 на процессоре 1	eax	myvar	eax	P2 на процессоре 2
-	-	0	-	-
mov eax, myvar	0	0	-	-
-	0	0	0	mov eax, myvar
-	0	0	1	inc eax
-	0	1	1	mov myvar, eax
inc eax	1	1	1	-
mov myvar, eax	1	1	1	-

Мы потеряли данные.

Рассмотрим эту же ситуацию для однопроцессорной машины (Для двух потоков). Здесь не может быть двух регистров eax, он один.

1 ядро				
P1 на процессоре 1	eax	myvar	eax	P2 на процессоре 2
-	-	0	-	-
Выполняется P1				
mov eax, myvar	0	0	-	-
P1 вытеснен, его контекст сохранён.				
-	0	0	mov eax, myvar	-
-	0	1	inc eax	-
-	1	1	mov myvar, eax	-
P2 вытеснен, его контекст сохранён.				
inc eax	1	1	-	-
mov myvar, eax	1	1	-	-

Ситуация повторилась, несмотря на то что это квазипараллельность, но мы видим что переключение связано с переключением контекста, с запоминанием содержимого регистров, соответственно мы получили тот же самый результат.

Подобные действия принято называть критическими.

Часть кода в котором выполняются такие действия принято называть критической секцией.

Myvar - разделяемая переменная.

Что же делать чтобы не терять данные? Для этого необходимо обеспечить монопольное использование разделяемых параллельными процессами ресурсов.

**Монопольный доступ и Взаимоисключение // вроде монопольный доступ еще и выше описано, так что....**

### **Взаимоисключения**

5. Программный.
6. Аппаратный.
7. С использованием семафоров.
8. С использованием мониторов.

**Мониторы**

Мониторы определения

Задача «читатели-писатели»

## **20 билет**

20.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – примеры, семафоры – определение, виды семафоров, примеры использования множественных семафоров из лабораторных работ «производство-потребление» и «читатели-писатели».

**параллельные процессы**

Параллельные процессы

**взаимоисключения (есть выше)**

**программная реализация взаимоисключений**

Программная реализация

**семафоры, определение, виды**

Семафоры: определение

**Таблица семафоров**

Каждый элемент этой таблицы описывает структуры struct semid\_ds (<sys/sem.h>)

Каждая строка таблицы описывает отдельный набор семафоров

Каждый семафор описан структурой struct sem.

На семафорах определены следующие системные вызовы:

- semget() - создаёт набор семафоров
- semctl() - позволяет изменять управляющие параметры набора семафоров
- semop() - изменяет значение семафора - отдельного, набора или части набора семафоров

на семафоре определена структура

```
struct sembuf {  
    unsigned short sem_num; // - индекс семафора  
    short sem_op; // - операция на семафоре  
    short sem_flg; // - флаги, определенные на семафоре  
};
```

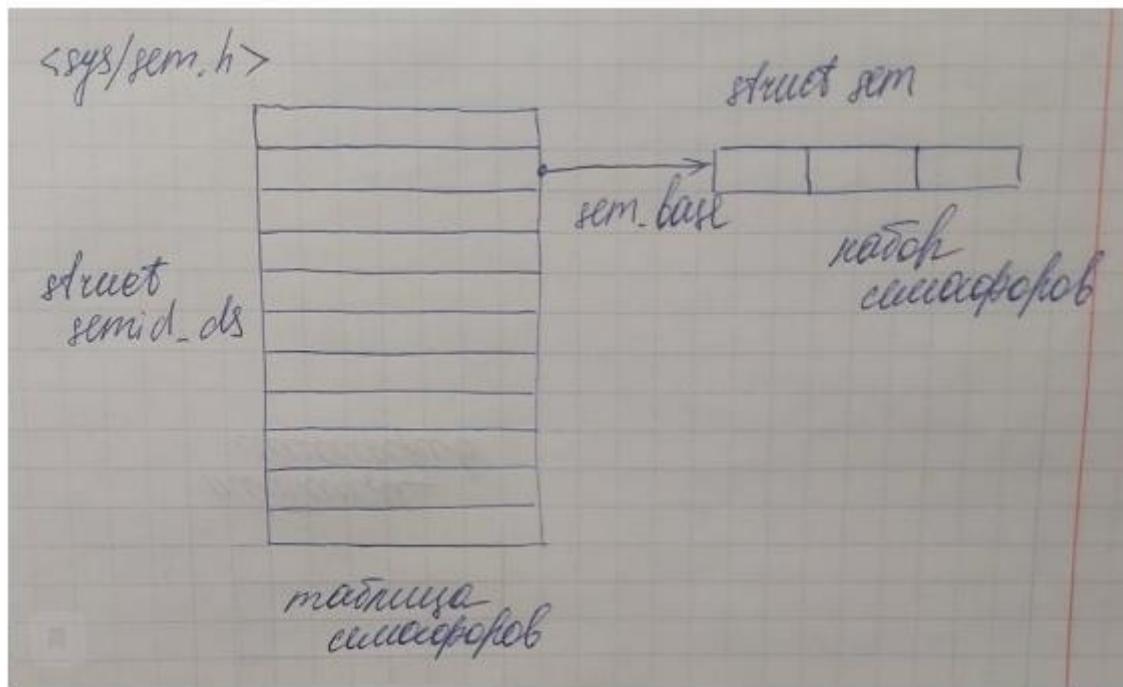
В отличие от классических семафоров Дейкстры (у него две операции), на семафоре UNIX определены три операции

4. sem\_op > 0 - освобождение ресурса
5. sem\_op == 0 - ожидание ресурса без захвата
6. sem\_op < 0 - захват ресурса

Декрементировать семафор можно только если семафор имеет положительное значение.

На семафорах определены специальные флаги:

- IPC\_NOWAIT - информирует ядро системы о нежелании процесса переходить в состояние ожидания
- SEM\_UNDO - указывает ядру, что оно должно отслеживать изменение значения семафора в результате системного вызова semop() и при завершении процесса, вызвавшего semop(), ядро ликвидирует сделанные изменения для того, чтобы процессы не были заблокированы на семафоре навечно.



**примеры использования множественных семафоров на примере читатели писатели и производство потребление**

## 20.2 Приоритетное планирование в ОС Windows (лабораторная работа).

В Windows реализуется приоритетная, вытесняющая система планирования, при которой всегда выполняется хотя бы один работоспособный (готовый) поток с самым высоким приоритетом, с той оговоркой, что конкретные, имеющие высокий приоритет и готовые к запуску потоки могут быть ограничены процессами, на которых им разрешено или предпочтительнее всего работать.

В Windows планировка потоков осуществляется на основании приоритетов готовых к выполнению потоков. Поток с более низким приоритетом вытесняется планировщиком, когда поток с более высоким приоритетом становится готовым к выполнению.

Код Windows, отвечающий за планирование, реализован в ядре. Нет единого модуля или процедуры с названием "планировщик", так как этот код рассредоточен по ядру. Совокупность процедур, выполняющих эти обязанности, называется диспетчером ядра. Диспетчеризация потоков может быть вызвана:

- • поток готов к выполнению (только что создан или вышел из состояния "ожидания")
- • поток выходит из состояния "выполняется", т.к. его квант истек, либо поток завершается, либо переходит в состояние "ожидание"
- • поменялся приоритет потока
- • изменилась привязка к процессорам => поток больше не может работать на процессоре, на котором он выполнялся.

Windows использует 32 уровня приоритета:

- от 0 до 15 – 16 изменяющихся уровней (из которых уровень 0 – зарезервирован для потока обнуления страниц) 10
- от 16 до 31 – 16 уровней реального времени

Уровни приоритета потоков назначаются Windows API и ядром операционной системы.

Windows API сортирует процессы по классам приоритета, которые были назначены при их создании:

- реального времени (real-time, 4);
- высокий (high, 3);
- выше обычного (above normal, 6);
- обычный (normal, 2);
- ниже обычного (below normal, 5);
- простой (idle, 1).

Затем назначается относительный приоритет потоков в рамках процессов:

- критичный по времени (time critical, 15);
- наивысший (highest, 2);
- выше обычного (above normal, 1);
- обычный (normal, 0);
- ниже обычного (below normal, -1);
- низший (lowest, -2);
- простой (idle, -15).

Исходный базовый приоритет потока наследуется от базового приоритета процесса.

Процесс по умолчанию наследует свой базовый приоритет у того процесса, который его создал.

#### *Соответствие между приоритетами Windows API и ядра системы*

	real-time	high	above normal	normal	below normal	idle
time critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Текущий приоритет потока в динамическом диапазоне — от 1 до 15 — может быть повышен планировщиком вследствие следующих причин:

- повышение вследствие события планировщика или диспетчера;

*При наступлении события диспетчера вызываются процедуры с целью проверить не должны ли на локальном процессоре быть намечены какие-либо потоки, которые не должны быть спланированы. При каждом наступлении такого события вызывающий*

*код может также указать, какого типа повышение должно быть применено к потоку, а также с каким приращением приоритета должно быть связано это повышение*

- повышение приоритета владельца блокировки;

*Так как блокировки ресурсов исполняющей системой и блокировки критических разделов используют основные объекты диспетчеризации, то в результате освобождения этих блокировок осуществляются повышения приоритетов, связанные с завершением ожидания. Но с другой стороны, так как высокоуровневые реализации этих объектов отслеживают владельца блокировки, то ядро может принять решение о том, какого вида повышение должно быть применено с помощью AdjustBoost.*

- Повышения приоритета, связанные с завершением ожидания

*В общем случае поток, пробуждающийся из состояния ожидания, должен иметь возможность приступить к выполнению как можно скорее.*

- повышение приоритета после завершения ввода/вывода (таблица ниже);

*Windows дает временное повышение приоритета при завершении определенных операций ввода/вывода, при этом потоки, которые ожидали ввода/вывода имеют большие шансы сразу же запуститься. Подходящее значение для увеличения зависит от драйвера устройств*

- повышение приоритета вследствие ввода из пользовательского интерфейса;

*Потоки — владельцы окон получают при пробуждении дополнительное повышение приоритета на 2 из-за активности при работе с окнами, например, при поступлении сообщений от окна. Система работы с окнами ( Win32k.sys ) применяет это повышение приоритета, когда вызывает функцию KeSetEvent для установки события, используемого для пробуждения GUI-потока. Смысл такого повышения схож со смыслом предыдущего повышения — содействие интерактивным приложениям.*

- повышение приоритета вследствие длительного ожидания ресурса исполняющей системы;

*Если поток пытается получить ресурс исполняющей системы, который уже находится в исключительном владении другого потока, то он должен войти в состояние ожидания до тех пор, пока другой поток не освободит ресурс. Для ограничения риска взаимных исключений исполняющая система выполняет это ожидание, не входя в бесконечное ожидание ресурса, а интервалами по 500 мс.*

*Если по окончании этих 500 мс ресурс все также находится во владении, то исполняющая система пытается предотвратить зависание центрального процессора путем получения блокировки диспетчера, повышения приоритета потока (потоков), владеющих ресурсом до 15 (в случае если исходный приоритет владельца был меньше, чем у ожидающего, и не был равен 15), перезапуска их квантов и выполнения еще одного ожидания.*

- повышение вследствие ожидания объекта ядра;

- повышение приоритета в случае, когда готовый к выполнению поток не был запущен в течение длительного времени;

*Диспетчер настройки баланса (механизм ослабления загруженности центрального процессора) сканирует очередь готовых потоков раз в секунду и, если обнаружены потоки, ожидающие выполнения более 4 секунд, то диспетчер настройки баланса повышает их приоритет до 15. Как только квант истекает, приоритет потока снижается до базового приоритета. Если поток не был завершен за квант времени или был вытеснен потоком с более высоким приоритетом, то после снижения приоритета поток возвращается в очередь готовых потоков. Диспетчер настройки баланса сканирует лишь 16 готовых потоков и повышает приоритет не более чем у 10 потоков (если найдет) за один проход. При следующем проходе сканирование возобновляется с того места, где оно было прервано в прошлый раз.*

- повышение приоритета проигрывания мультимедиа службой планировщика MMCSS.

Потоки, на которых выполняются различные мультимедийные приложения, должны выполняться с минимальными задержками. В Windows

такая задача решается с помощью повышения приоритетов таких потоков драйвером MMCSS (MultiMedia Class Scheduler Service).

MMCSS работает с различными определенным задачи (аудио, видео, игры)

Устройство	Приращение
Диск, CD-ROM, параллельный порт, видео	1
Сеть, почтовый ящик, именованный канал, последовательный порт	2
Клавиатура, мышь	6
Звуковая плата	8

## IRQL

Для обеспечения поддержки мультизадачности системы, когда исполняется код режима ядра, Windows использует приоритеты прерываний IRQL.

Прерывания обслуживаются в порядке их приоритета. При возникновении прерывания с высоким приоритетом процессор сохраняет информацию о состоянии прерванного потока и активизирует сопоставленный с данным прерыванием диспетчер ловушки. Последний повышает IRQL и вызывает процедуру обслуживания прерывания (ISR).

После выполнения ISR диспетчер прерывания понижает IRQL процессора до исходного уровня и загружает сохраненные ранее данные о состоянии машины. Прерванный поток возобновляется с той точки, где он был прерван. Когда ядро понижает IRQL, могут начать обрабатываться ранее замаскированные прерывания с более низким приоритетом. Тогда вышеописанный процесс повторяется ядром для обработки этих прерываний.

## 20.2 Пересчет динамических приоритетов в ОС unix и Windows (лабораторная работа). (что выше плюс ниже)

В ОС семейства UNIX и в ОС семейства Windows только приоритеты пользовательских процессов могут динамически пересчитываться

Планирование процессов в UNIX основано на приоритете процесса. Планировщик всегда выбирает процесс с наивысшим приоритетом. Приоритеты процессов изменяются с течением времени (динамически) системой в зависимости от использования вычислительных ресурсов, времени ожидания запуска и текущего состояния процесса. Если процесс готов к запуску и имеет наивысший приоритет, планировщик приостановит выполнение текущего процесса (с более низким приоритетом), даже если тот не «выработал» свой временной квант.

*Традиционное ядро UNIX является строго невытесняющим, однако в современных системах UNIX ядро является вытесняющим – то есть процесс в режиме ядра может быть вытеснен более приоритетным процессом в режиме ядра. Ядро сделано вытесняющим для того, чтобы система могла обслуживать процессы реального времени, например видео и аудио.*

Очередь процессов, готовых к выполнению, формируется согласно приоритетам и принципу вытесняющего циклического планирования, то есть сначала выполняются процессы с большим приоритетом, а процессы с одинаковым приоритетом выполняются в течении кванта времени друг за другом циклически. В случае, если процесс с более высоким приоритетом поступает в очередь процессов, готовых к выполнению, планировщик вытесняет текущий процесс и предоставляет ресурс более приоритетному процессу.

Приоритет процесса задается любым целым числом, которое лежит в диапазоне от 0 до 127 (чем меньше число, тем выше приоритет)

- 0 - 49 – зарезервированы для ядра (приоритеты ядра фиксированы)
- 50 - 127 – прикладные (приоритеты прикладных задач могут изменяться во времени)

Изменение приоритета прикладных задач зависит от следующих факторов:

- фактор ”любезности” (nice) – это целое число в диапазоне от 0 до 39 со значением 20 по умолчанию. Увеличение значения приводит к уменьшению приоритета. Пользователи могут повлиять на приоритет процесса при помощи изменения значений этого фактора, но только суперпользователь может увеличить приоритет процесса. Фоновые процессы автоматически имеют более высокие значения этого фактора.
- последней измеренной величины использования процессора

Структура proc содержит следующие поля, которые относятся к приоритетам:

- p\_pri – текущий приоритет планирования
- p\_usrpri – приоритет режима задачи
- p\_cpri – результат последнего измерения использования процессора
- p\_nice – фактор ’любезности’, который устанавливается пользователем

`p_pri` используется планировщиком для принятия решения о том, какой процесс отправить на выполнение. `p_pri` и `p_usrpri` равны, когда процесс находится в режиме задачи.

Значение `p_pri` может быть изменено (повыщено) планировщиком для того, чтобы выполнить процесс в режиме ядра. В таком случае `p_usrpri` будет использоваться для хранения приоритета, который будет назначен процессу при возврате в режим задачи.

`p_sri` инициализируется нулем при создании процесса (и на каждом тике обработчик таймера увеличивает это поле текущего процесса на 1, до максимального значения равного 127).

Каждую секунду ядро системы инициализирует отложенный вызов процедуры `schedcpu()`, которая уменьшает значение `p_pri` каждого процесса

///////////Функция `schedcpu()`:

*обновляет все приоритеты процесса. Во-первых, он обновляет статистику, которая отслеживает, как долго процессы находились в различных состояниях процесса.*

//////////

$$decay = \frac{2 \cdot load\_average}{2 \cdot load\_average + 1}$$

исходя из фактора "полураспада"

где `load_average` - это среднее количество процессов, находящихся в состоянии готовности к выполнению, за последнюю секунду. Также процедура `schedcpu()` пересчитывает приоритеты для режима задачи всех процессов по формуле

$$p\_usrpri = PUSER + \frac{p\_cpu}{2} + 2 \cdot p\_nice$$

где `PUSER` - базовый приоритет в

режиме задачи, равный 50.

Таким образом, если процесс в последний раз использовал большое количество процессорного времени, то его `p_sri` будет увеличен. Это приведет к росту значения `p_usrpri`, то есть к понижению приоритета. Чем дольше процесс простоявает в очереди на выполнение, тем больше фактор полураспада уменьшает его `p_sri`, что приводит к повышению его приоритета.

Динамический пересчет приоритетов процессов в режиме задачи позволяет избежать бесконечного откладывания.

Таким образом, приоритет процесса в режиме задачи может быть динамически пересчитан по следующим причинам:

- Вследствие изменения фактора любезности процесса системным вызовом `nice`
- В зависимости от степени загруженности процессора процессом `p_sri`
- Вследствие ожидания процесса в очереди готовых к выполнению процессов
- Приоритет может быть повышен до соответствующего приоритета сна вследствие ожидания ресурса или события

## 21 билет

21.1 Взаимодействие параллельных процессов: монопольное использование – реализация; типы реализации взаимоисключения. Мониторы – определение, примеры: простой монитор, монитор «кольцевой буфер» и монитор «читатели-писатели». Пример реализации монитора Хоара «читатели-писатели» для ОС Windows.

Необходимость монопольного доступа к разделяемым переменным при взаимодействии параллельных процессов обусловлена тем, что при немонопольном доступе возможно возникновение потерянного обновления

[тут про параллельные процессы](#)

[тут про мониторы](#)

[Задача «читатели-писатели»](#)

21.1 (2021) Взаимодействие параллельных процессов: монопольное использование – реализация; типы реализаций взаимоисключения. Алгоритм Э.Дейкстры “Алгоритм банкира” и алгоритм Хабермана с примером определения состояния системы.

21.2 Процессы Unix: создание процесса в ОС Unix и запуск новой программы.

Примеры из лабораторной работы (код).

[fork\(\)](#) – создание

[exec\(\)](#) – запуск

примеры там же

## 22 билет

22.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; аппаратная реализация взаимоисключения, спин-блокировка – реализация. (+ от 2021) семафоры Дейкстры: определение, взаимоисключения с помощью семафоров, алгоритм “Производство-потребление” – решение Дейкстры.

### **Взаимодействие параллельных процессов**

[18.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – флаги, алгоритм Деккера, алгоритм Лампорта \(уточнение 2021 алгоритм Булочная\)](#).

### **аппаратная реализация взаимоисключения**

#### ***Test-and-set***

Наиболее известным аппаратным средством является неделимая команда – test-and-set (проверить и установить) [8].

Для реализации взаимоисключения с разделяемым ресурсом связывается так называемый байт блокировки. Для каждого ресурса должен быть определен свой байт блокировки. Пусть значение байта блокировки 0 означает, что ресурс доступен, а значение 1 - ресурс занят.

Перед обращение к ресурсу процесс должен выполнить следующие шаги:

1. Проверить значение байта блокировки.
2. Установить байт блокировки в 1.
3. Если значение байта блокировки равно 1, то вернуться на шаг 1.

После завершения использования ресурса процесс должен установить байт блокировки в 0.

*В IBM370 такая команда называлась TS, которая выполняла действия 1 и 2 в рамках одной неделимой операции (a single atomic (i.e., non-interruptible) operation), т.е. ее выполнение нельзя прервать*

---

```

Program example-test-and-set;
active: logical;
P1: //первый процесс
    flag1: logical; //локальная переменная
    While (true) do
        begin
            flag1 = true;
            While (nfalg1) do
                TS(flag1, active); //цикл проверки переменной active
                // критическая секция первого процесса
                active = 0;
                // другие операторы первого процесса


---


        end;
    end; //P1
P2: //второй процесс
    flag2 : logical; //локальная переменная
    While (true) do
        begin
            flag2 = true;
            While (falg2) do
                TS(flag2, active); //цикл проверки переменной active
                // критическая секция второго процесса
                active = 0;
                // другие операторы второго процесса
            end;
    end; //P2
begin
    active = false;
parbegin
    P1;P2;
parend;
end.

```

---

### Листинг 5

Переменная active имеет значение «истина», если какой-то процесс находится в своем критическом участке.

При использовании команды типа test-and-set присутствует активное ожидание. Кроме того, не исключается потенциальная возможность бесконечного откладывания. Однако, считается, что его вероятность мала в силу того, что команда test-and-set является неделимой. Когда процесс выходит из своего критического участка, устанавливая для active значение «ложь», то команда test-and-set, вызываемая другим процессом скорее всего сможет «перехватить» ее и процесс сможет войти в свою критическую секцию

### **спин-блок – реализация**

Использование команды test-and-set в цикле проверки значения переменной называется циклической блокировкой или спин-лок (англ. - spin lock, иногда simple lock или даже simple mutex).

Чаще всего команда test-and-set возвращает предыдущее значение переменной

```
void spin_lock(spin_lock_t *c)
{
    while(test-and-set(*c) != 0)
        /* ресурс занят*/;
}
void spin_unlock(spin_lock_t *c)

{
    *c=0;
}
```

Реализация команды test-and-set во многих архитектурах связана с блокировкой локальной шины памяти. В результате длительный цикл обращения к команде test-and-set может привести к занятию шины одним потоком (нитью) и, следовательно, к существенному снижению производительности системы (снижению уровня отзывчивости).

Решить проблему можно путем использования двух вложенных циклов:

```
void spin_lock(spin_lock_t *c)
{
    while(test-and-set(*c) != 0)
        while(*c != 0);
        /* ресурс занят*/;
}
```

Если переменная занята, то выполняется вложенный цикл, в котором проверка переменной с выполняется без захвата шины. Команда spin\_lock() базируется на команде test-and-set и является макропрограммой.

Спин блокировки особенно часто используются в ядре. Это связано с тем, что часто объекты ядра не могут блокироваться, т.е. для взаимоисключения таких объектов нельзя использовать семафоры и мьютексы.

[семафоры Дейкстры](#)

[производство-потребление](#)

22.2 Процессы: бесконечное откладывание, зависание, тупиковая ситуация – анализ на примере задачи об обедающих философах и примеры аналогичных ситуаций в ОС. Множественные семафоры в Linux: системные вызовы и поддержка в ОС Linux; примеры из лабораторных работ.

Бесконечное откладывание в планировании - ситуация, когда какой-либо процесс не получает процессорного времени.

Зависание - это ситуация, в которой происходит захват и освобождение одного и того же ресурса, но полезной работы не выполняется.

Тупиковая ситуация или тупик - это ситуация, которая возникает в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другим процессом, ожидающим освобождения ресурса, занятого первым процессом.

[Задача «Обедающие философы» – модели распределение ресурсов вычислительной системы.](#)

[Множественные семафоры UNIX](#) там же производство-потребление  
[«Читатели-писатели»](#)

## 23 билет

## 24 билет

24.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; алгоритм Лампорта «Булочная» и «Логические часы» Лампорта.

отсюда все про проц и булочная

[18.1 Процессы: взаимодействие параллельных процессов – монопольный доступ и взаимоисключение; программная реализация взаимоисключения – флаги, алгоритм Деккера, алгоритм Лампорта \(уточнение 2021 алгоритм Булочная\) .](#)

[синхронизация логических часов \(алгоритм Лампорта\)](#) отсюда часы

24.2 Процессы: бесконечное откладывание, зависание, тупиковая ситуация – анализ на примере задачи об обедающих философах и примеры аналогичных ситуаций в ОС. Множественные семафоры в Linux: системные вызовы и поддержка в системе; пример из лабораторной работы «производство-потребление»..

Бесконечное откладывание в планировании - ситуация, когда какой-либо процесс не получает процессорного времени.

Зависание - это ситуация, в которой происходит захват и освобождение одного и того же ресурса, но полезной работы не выполняется.

Тупиковая ситуация или тупик - это ситуация, которая возникает в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно или через цепочку запросов другим процессом, ожидающим освобождения ресурса, занятого первым процессом.

[Задача «Обедающие философы» – модели распределение ресурсов вычислительной системы.](#)

[Множественные семафоры UNIX](#) там же производство-потребление  
[«Читатели-писатели»](#)

## 25 билет

[25.1 Алгоритмы взаимоисключения в распределенных системах и транзакции \(доп вопрос – два подхода при работе с транзакциями\). Тупики при транзакциях](#)  
по рассказам Лены этот вопрос (за исключением транзакций очень схож с 16.1) поэтому (возможно не надо так подробно писать про программный аппаратный семафоры и мониторы)

[16.1 Взаимодействие параллельных процессов: проблемы; монопольный доступ и взаимоисключение; взаимодействие параллельных процессов в распределенных системах – особенности; централизованный алгоритм, распределенный алгоритм; синхронизация логических часов \(алгоритм Лампорта\).](#)

## Транзакции: определение, особенности, двухфазный протокол фиксации.

Модель транзакций пришла из бизнеса. Когда говорят транзакция – подразумевается неделимость.

Транзакция - само по себе неделимое действие. Транзакция - высокоуровневое средство взаимодействия процессов.

Любая транзакция должна быть неделимой, это свойство транзакции.

Транзакцией называется последовательность операций над одним или несколькими объектами базы данных, такими как файлы, записи и тому подобное, которая переводит систему из одного целостного состояния в другое целостное состояние.

Свойства: (+атомарность это как определение – операция — операция, которая либо выполняется целиком, либо не выполняется вовсе)

- упорядочиваемость – свойство, которое гарантирует, что если 2 или более транзакции выполняются параллельно, то конечный результат будет выглядеть так, как если бы все транзакции выполнялись последовательно в некотором определенном порядке.
- неделимость – если транзакция находится в процессе выполнения, то никто не может увидеть ее промежуточные результаты

- постоянство – после фиксации транзакции никакой сбой не может отменить результаты ее выполнения

## Механизмы реализации транзакций

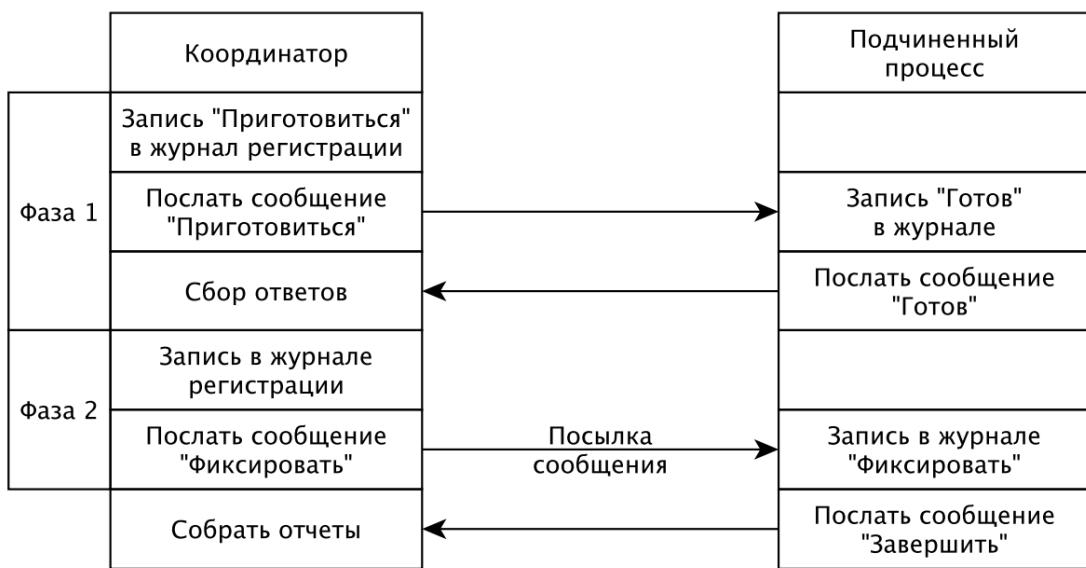
Существует два основных подхода к реализации механизма транзакций

1. Для каждого процесса, участвующего в транзакции, создается **индивидуальное рабочее пространство**. В этом пространстве находятся копии всех файлов (объектов), которые нужны процессу для выполнения транзакции. Пока транзакция не будет зафиксирована или не прервется, все изменения выполняются только над объектами в этом индивидуальном рабочем пространстве. Если транзакция успешно завершается, то все изменения копируются в исходные объекты. Если транзакция прерывается, то копии просто удаляются (исходные файлы не были изменены). Очевидный недостаток: большие накладные расходы, так как создаются дополнительные копии файлов (объектов) в системе.
2. **Список намерений**. В данном подходе модифицируются сами файлы/записи/объекты, но перед любым изменением выполняется запись в специальный файл, который называется журнал регистрации. В этом журнале отмечается, какая транзакция делает изменения, какой файл/запись изменяется и в этот журнал записывается старое и новое значения изменяемого файла записи. Только после того, как транзакция успешно выполнена, указанные изменения сохраняются в исходном файле. Если транзакция фиксируется (выполняется успешно), то об этом делается запись в журнале регистрации, но записи не удаляются. Если транзакция прерывается, то журнал регистрации используется для приведения файлов (записей) в исходное состояние - такое действие называется **откатом**.

## Протокол фиксаций транзакций

В распределённых системах фиксация транзакций может потребовать взаимодействия нескольких процессов, которые выполняются на разных машинах. В этом случае каждая такая отдельная машина хранит какие-то переменные/файлы базы данных. Для достижения неделимости транзакции в распределенных системах, используется специальный протокол - протокол двухфазной фиксации транзакций.

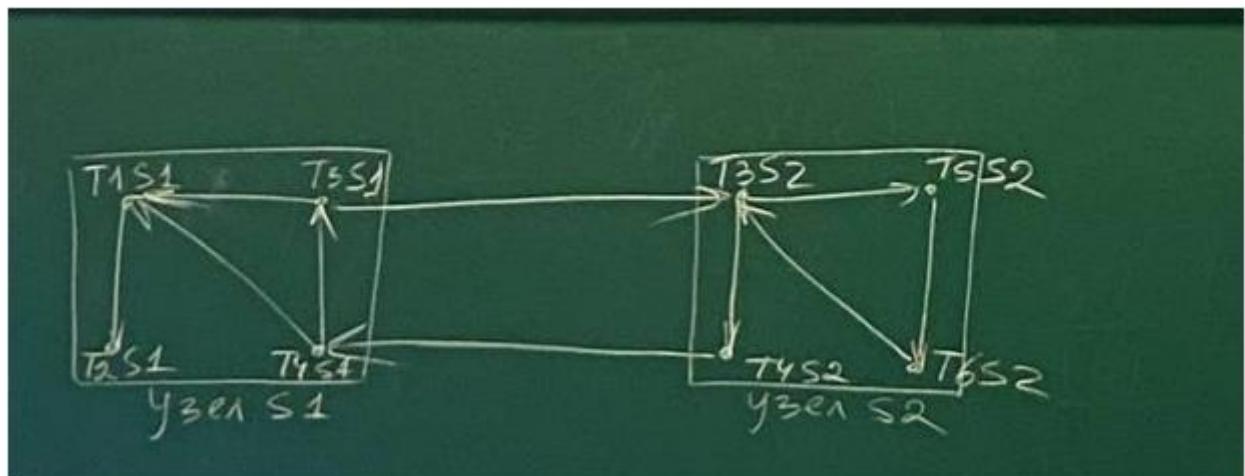
Это не единственный протокол для фиксации транзакции, но он считается наиболее надежным. Суть протокола: для обеспечения выполнения транзакции в распределенной системе один из взаимодействующих процессов выполняет функции координатора.



### тупики при транзакции

Транзакция может находиться в активном состоянии или в состоянии блокировки. Состояние блокировки – необходимый ресурс используется в другой транзакции.

Для отображения состояния всех транзакций будем использовать графовую модель, в которой каждый узел графа отображает состояние процесса, при этом каждый узел обозначим парой имен. Именем транзакции и именем узла. Имена узлов должны быть уникальными



1. если дуга из  $T_i S_k$  в  $T_j S_k$  то говорят, что  $i$  блокирована в ожидании когда  $j$  освободит ресурс
2. Если дуга  $T_k S_i$  в  $T_k S_j$  то..... СПРОСИТЬ У РЯЗАНОВОЙ

**Существуют 3 подхода к обработке тупиковых ситуаций в централизованных системах**

1. Предотвращение тупиковых ситуаций

Предварительное получение всех блокировок. Т получает все блокировки перед началом выполнения и сохраняет блокировки на протяжении всей Т. Если другой Т нужна будет блокировка, то она будет ждать. (нет взаимоблокировок, так как ни одна из ожидающих Т не удерживает блокировку)

## 2. Избежание тупиковых ситуаций

Обрабатывать тупиковые ситуации до того, как они возникнут.

Кратко: Диспетчер блокировок проверяет доступна ли блокировка, если она доступна, то диспетчер блокировок выделяет элемент данных и транзакция получает блокировку. Но если элемент заблокирован какой-либо другой Т в несовместимом режиме, диспетчер блокировок запускает алгоритм чтобы проверить, приводит ли сохранение Т в ожидании к взаимоблокировке.

Алгоритм решает, может ли Т ждать или одна из Т должна быть прервана.

2 алгоритма:

- Wait-Die если T1 старше, чем T2, T1 может подождать. В противном случае, если T1 младше T2, T1 прерывается, а затем перезапускается.
- Wound-Wait Если T1 старше T2, T2 прерывается и позже перезапускается. В противном случае, если T1 младше T2, T1 может ждать.

## 3. Обнаружение тупиковых ситуаций

Периодически запускается алгоритм обнаружения взаимоблокировок и если обнаружены, то удаляются. Когда транзакция запрашивает блокировку, диспетчер проверяет, доступна ли она. Если да, то Т разрешается заблокировать элемент данных, иначе Т разрешается ждать. То есть при выполнении запросов на блокировки нет никаких мер предосторожности, поэтому некоторые транзакции могут попасть в тупик. Для обнаружения тупиков диспетчер периодически проверяет, есть ли у графа циклы. Wait-for graph

Если обнаружен тупик, менеджер блокировок выбирает Т-жертву из каждого цикла.

Жертва прерывается и откатывается, что для Т возможно – так как есть журнал фиксации.

Методы выбора Т-жертвы.

1. Прерывают самую молодую Т
2. Выбирают Т с наименьшим количеством элементов данных
3. Выбирают Т, для которой было выполнено наименьшее количество обновлений
4. Выбирают Т с наименьшими затратами на перезапуск
5. Выбирают Т, которая является общей для 2 или более циклов.

## Обработка тупиков в распределенных системах

**Распределенная БД – БД, которая находится на нескольких сайтах и использует данные с разных сайтов.**

То есть одна и та же Т может быть активна на одном сайте и неактивна на другом. И когда на одном сайте две конфликт Т, то одна может быть неактивна. Этого никогда не возникает в централиз сис. Эта проблема называется проблемой местоположения Т.

### 1. Предотвращение (недопущение)

то есть сайт, на котором выполняется транзакция определяется как контролирующий. Он отправляет сообщения на сайт, на котором расположены элементы данных для блокировки этих элементов и ждет подтверждения. Если все сайты ответили, что заблокирован, то Т начнет выполняться. Иначе Т должна будет подождать

минус - время связи между сайтами. Если управляющий сайт потерпел крах, он не может связаться с другими сайтами, которыедерживают блокировки.

### 2. Избежание

Предположит имеются Т1 и Т2. Т1 – пребывает на сайт Р и пытается зав=блокировать элемент данных, который уже захвачен Т2 на этом же сайте. Этот сайт называют зоной Р (сайт Р – зона Р)

2 алгоритма:

#### 1. Distributed wound-die – распределенная смертельная рана)

- первый способ – если Т1 старше Т2, то есть Т2 пришла позже, то Т1 может подождать, при этом Т1 может возобновить свое выполнение после того, как сайт Р получит сообщение о том, что Т2 успешно завершилась или была прервана.
- второй способ – если Т1 младше Т2, то Т1 прерывается. При этом специальный менеджер на сайте Р должен отправить сообщение всем сайтам, которые посещала Т1, чтобы корректно ее прервать. При этом управляющий сайт должен уведомить пользователя, что Т1 была прервана на всех сайтах успешно.

#### 2. Distributed wait-wait – распределенное ожидание ожидание (да да, именно 2 раза слово ожидание).

- первый способ– Если Т1 старше Т2, то Т2 следует прервать. Если Т2 активна (то есть выполняется на сайте Р), то сайт Р прерывает и откатывает Т2, после этого рассыпает сообщение об этом на другие сайты. Если Т2 покинула сайт Р, (то ест завершилась на сайте Р), но выполняется на сайте Q, то сайт Р оповещает, что Т2 была прервана. Но по логике вещей сайт Q должен прервать эту Т2 и отправить соответственно сообщение другим сайтам, или это может сделать сайт управляющие транзакцией (скажем Лидер).
- второй способ – Если Т1 младше Т2, то Т1 разрешено ждать ( то есть Т1 блок в ожидании). При этом т1 сможет возобновить выполнения после того, как сайт Р получит сообщение , что Т2 завершилась

### 3. Обнаружение

Просто найти и удалить блокировку мы не можем, так как ожидание ресурсов у нас по сети и на графе это сложно отобразить.

В качестве альтернативы используются таймеры. Каждая Т связана с таймером, который установлен на период времени, в который ожидает завершение Т. Если Т не завершается в течение этого периода времени, таймер отключается и это указывает на возможную взаимоблокировку.

детектор тупиковых ситуаций. В централизованной системе есть один детектор тупика. Детектор может находить тупик для подконтрольных ему сайтом.

1. Централизованный детектор взаимоблокировок
2. Иерархический – ряд детекторов сгруппированы в иерархию.
3. Детектор распределенных тупиков – все сайты участвуют в обнаруж тупиков.

[25.2 Системные вызовы fork? exec? wait, \(доп концепция копи он рейт, что за такие зомби, чего помогает избежать wait\)](#)

опять таки по рассказам, это прям 16.2

[16.2 Процессы в UNIX: системные вызовы fork\(\), exec\(\), wait\(\), signal\(\) – примеры из лабораторных работ. \(из 2021 + особенности выполнения\)](#)

## 27 билет

27.1 Виртуальная память: управление памятью страницами по запросам – три схемы преобразования виртуального адреса к физическому. Алгоритмы вытеснения страниц: демонстрация особенностей на модели траектории страниц. Рабочее множество – определение, глобальное и локальное замещение. Флаги в дескрипторах страниц, предназначенные для реализации замещения страниц.

### **Виртуальная память**

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

3 схемы управления виртуальной памятью

7. Управление памятью страницами по запросу
8. Управление памятью сегментами по запросу
9. Управление памятью сегментами, поделенными на страницы по запросу.

**управление страницами по запросу – три схемы**

[Управление памятью страницами по запросам](#)

## **алгоритмы вытеснения страниц, демонстрация особенностей на модели траектории страниц**

### [Алгоритмы вытеснения](#)

#### **рабочее множество – определение**

##### [Теория рабочего множества](#)

#### **глобальное и локальное замещение**

(лекции прошлого года)

*Как же управлять системой, пейджингом, с тем чтобы процессы могли выполняться эффективно, могли загружать в память всё своё рабочее множество?* Очевидно, что это управление не выполняется вручную. Система должна выполнять это автоматически.

Существует два подхода:

1. глобальное замещение страниц;
2. локальное замещение страниц.

Глобальное замещение означает, что может быть вытеснена любая страница любого процесса для того чтобы процесс смог загрузить свою страницу в память.

Локальное замещение означает, что вытесняются только страницы данного процесса.

На самом деле можно переинчарить название рабочее множество как квота. То есть каждому процессу выделяют квоту, например 10 страниц. Любая система контролирует число страничных прерываний. Она должна это контролировать, потому что число страничных прерываний влияет на эффективность работы системы. Если у какого-то процесса резко возрастает число страничных прерываний это означает, что квота, выделенная процессу, мала. Ему надо выделить дополнительную квоту, например плюс три страницы. И опять же наблюдать за числом страничных прерываний. Потом эта квота может быть снижена через некоторое время. Таким образом в системе возможно регулирование числа страничных прерываний.

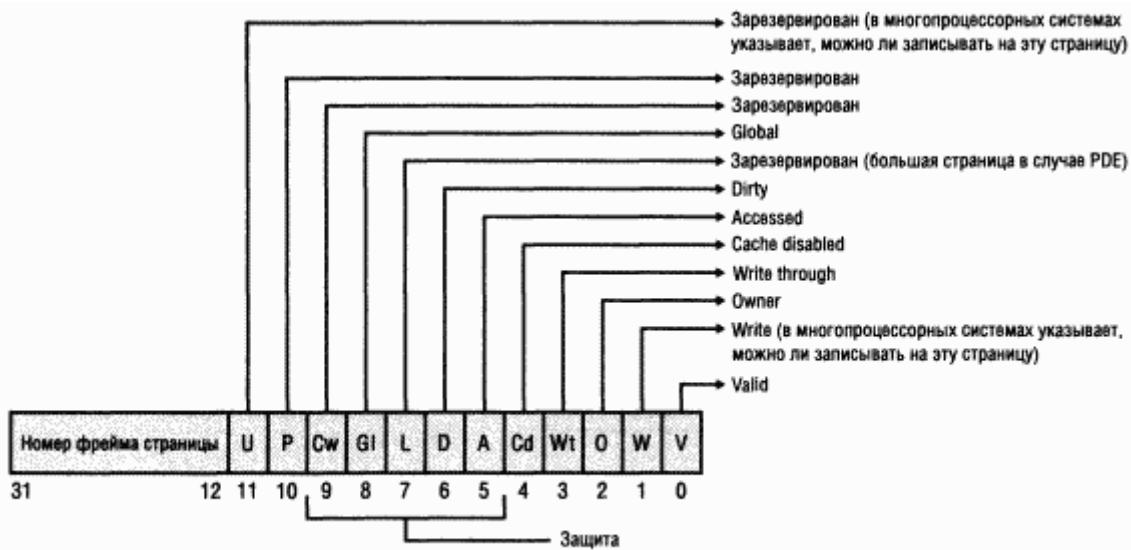
(с наших лекций)

Под глобальным замещением понимают выгрузку любой страницы любого процесса, чтобы процесс мог загрузить свою очередную страницу.

Под локальным - выгрузку только страниц данного процесса. Есть у него квота, возникло страничное прерывание – должна быть выгружена страница этого процесса.

В unix, linux есть page demon – демон страниц, в windows – swapping. Поздно пить боржоми, когда печень отвалилась. Процессу надо загрузить страницу. Но перед этим – выгрузить – а это время. На самом деле выгружаются страницы заранее, чтобы иметь набор свободных страничных кадров. Тогда подгрузка будет быстрее и каждый конкретный процесс эффективней.

**флаги в дескрипторах страниц, предназначенные для реализации замещения страниц**



**Рис. 7-19. Действительные аппаратные PTE в x86-системах**

0 - Present – Он установлен, если страница находится в памяти.

1 - read / write

2 - user / supervisor

3 - page write-Through

4 - page cache-Disable – кеширование данной страницы отключено

5 - accessed – Была ли операция чтения с данной страницы

6 - dirty – страница модифицирована

7 - Page Table Attribute index

8 - Global page –

9 - Зарезервирован

10 - Зарезервирован

11 - Зарезервирован

**27.2 Лабораторная работа по UNIX: системный вызов fork(), wait(), exec(), pipe(), signal(). Примеры.**

[16.2 Процессы в UNIX: системные вызовы fork\(\), exec\(\), wait\(\), signal\(\) – примеры из лабораторных работ. \(из 2021 + особенности выполнения\)](#)

**pipe() - (буквальный перевод - труба)**

*В отличие от разделяемой памяти, для кот в программе есть таблица разделяемых сегментов, программные каналы поддерживаются файловой подсистемой. То есть программные каналы имеют дескриптор файла*

Именованные имеют имя и inode, неименованные – только inode. Неименованные создаются системным вызовом `pipe`, который возвращает файловый дескриптор, если удалось создать канал. Неименованный имеет только дескриптор, поэтому пользоваться неименованным могут только родственники, потому что процессы-потомки в результате `fork` наследуют от предка дескрипторы открытых файлов. Это потоковая модель передачи данных.

Симплексная связь, односторонняя.

Для двусторонней – дуплексной, необходимо как минимум 2 трубы.

Программные каналы имеют встроенные средства взаимоисключения, то есть в канал нельзя писать, если читают, и нельзя читать, если в него пишут. Для этого определяли `fd[2]` – массив дескрипторов. Закрыть для записи и читать и наоборот.

## КОД ЛАБЫ С PIPE

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define N 2
#define TIME_SLEEP 2
#define LEN 64

#define OK 0
#define ERR_FORK -1
#define ERR_EXEC -1
#define ERR_PIPE -1

#define FORK_FAILURE 1
#define EXEC_FAILURE 2
#define PIPE_FAILURE 3

int main()
{
    int child[N];
    int fd[N];
    char text[LEN] = { 0 };
    char *mes[N] = {"BMSTU IU7-52 Kozlova\n", "ABCDEFG\n"};

    if (pipe(fd) == ERR_PIPE)
    {
        perror("Can't pipe!");
        return PIPE_FAILURE;
    }

    printf("Parent process start! PID: %d, GROUP: %d\n", getpid(), getpgrp());

    for (int i = 0; i < N; i++)
    {
        int child_pid = fork();

        if (child_pid == ERR_FORK)
        {
```

```

        perror("Can't fork()\n");
        return ERR_FORK;
    }
    else if (!child_pid)
    {
        close(fd[0]);
        write(fd[1], mes[i], strlen(mes[i]));
        printf("Message %d sent to parent! %s", i + 1, mes[i]);

        return OK;
    }
    else
    {
        child[i] = child_pid;
    }
}

for (int i = 0; i < N; i++)
{
    int status;
    int statval = 0;

    pid_t child_pid = wait(&status);

    printf("Child process %d finished. Status: %d\n", child_pid, status);

    if (WIFEXITED(statval)) не равно нулю, если дочерний процесс
успешно завершился.

    {
        printf("Child process %d finished. Code: %d\n", i + 1,
WEXITSTATUS(statval));
    }
    else if (WIFSIGNALED(statval)) возвращает истинное значение, если
дочерний процесс завершился из-за необработанного сигнала
    {
        printf("Child process %d finished from signal with code: %d\n",
i + 1, WTERMSIG(statval));
    }
    else if (WIFSTOPPED(statval)) возвращает истинное значение, если
дочерний процесс, из-за которого функция вернула управление,
    {
        printf("Child process %d finished stopped. Number signal:
%d\n", i + 1, WSTOPSIG(statval));
    }
}

printf("\nMessage receive :\n");
close(fd[1]);
read(fd[0], text, LEN);
printf("%s\n", text);

printf("Parent process finished! Children: %d, %d! \nParent: PID: %d, GROUP:
%d\n", child[0], child[1], getpid(), getpgrp()));

return OK;

```

```
}
```

## КОД ПРОГРАММЫ С SIGNAL()

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
#include <stdbool.h>
#include <signal.h>

#define N 2
#define TIME_SLEEP 2
#define LEN 64

#define OK 0
#define ERR_FORK -1
#define ERR_EXEC -1
#define ERR_PIPE -1

#define FORK_FAILURE 1
#define EXEC_FAILURE 2
#define PIPE_FAILURE 3

_Bool flag = false;

void catch_sig(int sig_num)
{
    flag = true;
    printf("catch_sig: %d\n", sig_num);
}

int main()
{
    int child[N];
    int fd[N];
    char text[LEN] = { 0 };
    char *mes[N] = {"BMSTU IU7-52\n", "ABCDEFG\n"};

    if (pipe(fd) == ERR_PIPE)
    {
        perror("Can't pipe!");
        return PIPE_FAILURE;
    }

    printf("Parent process start! PID: %d, GROUP: %d\n", getpid(), getpgrp());
    signal(SIGINT, catch_sig);
    sleep(2);

    for (int i = 0; i < N; i++)
    {
        int child_pid = fork();

        if(child_pid == ERR_FORK)
        {
```

```

        perror("Can't fork()\n");
        return ERR_FORK;
    }
    else if (!child_pid)
    {
        if (flag)
        {
            close(fd[0]);
            write(fd[1], mes[i], strlen(mes[i]));
            printf("Message %d sent to parent! %s", i + 1, mes[i]);
        }

        return OK;
    }
    else
    {
        child[i] = child_pid;
    }
}

for (int i = 0; i < N; i++)
{
    int status;
    int statval = 0;

    pid_t child_pid = wait(&status);

    printf("Child process %d finished. Status: %d\n", child_pid, status);

    if (WIFEXITED(statval))
    {
        printf("Child process %d finished. Code: %d\n", i + 1,
WEXITSTATUS(statval));
    }
    else if (WIFSIGNALED(statval))
    {
        printf("Child process %d finished from signal with code: %d\n",
i + 1, WTERMSIG(statval));
    }
    else if (WIFSTOPPED(statval))
    {
        printf("Child process %d finished stopped. Number signal:
%d\n", i + 1, WSTOPSIG(statval));
    }
}

printf("\nMessage receive :\n");
close(fd[1]);
read(fd[0], text, LEN);
printf("%s\n", text);

printf("Parent process finished! Children: %d, %d! \nParent: PID: %d, GROUP:
%d\n", child[0], child[1], getpid(), getpgrp());

return OK;
}

```

## 26 Дополнительные вопросы

26.1 XMS

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

26.3 Кэши TLB и данных

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

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

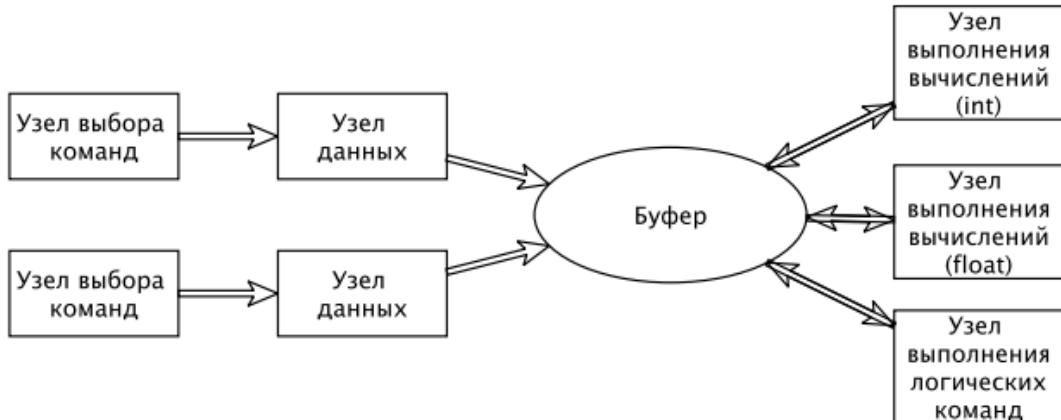
26.6 Управление памятью: выделение памяти разделами фиксированного размера, выделение памяти разделами переменного размера, стратегии выделения памяти, фрагментация памяти.

26.7 Тупики: Обнаружение тупиков для повторно используемых ресурсов методом редукции графа, способы представления графа и методы восстановления работоспособности системы

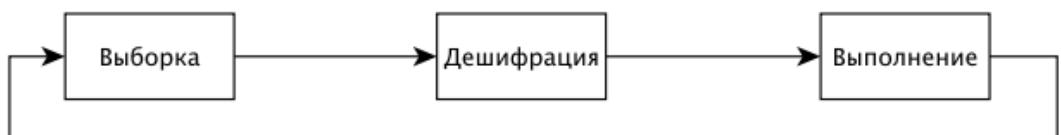
26.8 Последовательность операций при выполнении аппаратного прерывания.

Прерывания точные и неточные

Современные процессоры являются суперскалярными.



Этапы выполнения программ:



Замещение счетчика команд не отражает истинное значение, между выполненными и еще не выполненными. При возврате из прерывания невозможно просто начать выполнения с адреса, находящегося в счетчике команд.

В современных системах вводится понятие точного прерывания.

Точное прерывание – прерывание, оставляющее машину в строго определенном состоянии.

Точное прерывание условие:

- Счетчик команд – указывает на команду, до которой все команды выполнены полностью.
- Ни одна команда, на которую указывает счетчик не выполнена.
- Состояние команды, на которую указывается счетчик известно.

Следует, что все изменения связанные с этими командами должны быть отменены.

В конце цикла выполнения каждой команды процессор проверяет наличие прерывания Int на своей ножке.

При исключениях (при страничном прерывании) счетчик команд будет содержать адрес команды, на котором возникло страничное прерывание.

Для того, чтобы обработать прерывания как точное, машины с суперскалярными процессорами при каждом прерывании должны сохранять большие объемы данных, аппаратные прерывания выполняются медленно.

В итоге суперскалярные процессоры становятся непригодными для практических целей, из-за длительных прерываний.

Но не все прерывания необходимо делать точными: например, обработка деления на 0, так как процесс завершается.

Суперскалярные процессоры Pentium 0 поддерживают точные прерывания, ценой за точные прерывания является сложная внутрипроцессорная логика

## 26.9 EMS .

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

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

26.12 Прерывание int 8h (реальный режим) - функции.

## ТЕОРЕМЫ ДЛЯ ТУПИКОВ

### Теоремы

4. Граф является полностью сокращаемым, если существует такая последовательность сокращений, которая устраниет все дуги. Если граф нельзя полностью сократить, то анализируемое состояние является тупиком (доказательства есть в книге *Логическое программирование ОС*, мы не рассматриваем)  
В последнем примере, например, видно, что не хватает свободных единиц ресурса.
5. Цикл в графе повторно используемых ресурсов является необходимым условием тупика (Тупик существует, если существует цикл (или как по-другому говорят: *замкнутая цепь запросов*))
6. Если состояние системы  $S$  не является состоянием тупика и  $s -[i]-> t$  ( $i$  должно быть над стрелочкой), Марк Даун не умеет так; обозначает переход из состояния  $S$  в состояние  $T$ ), то в случае, если операция процесса  $r_i$  есть запрос и в  $T r_i$  находится в тупике, состояние  $T$  является состоянием тупика (проще: *возникновение тупика в системе возможно только в результате запроса*)

## EXEC

Execl() {execlp(), execle()}

Execv {execvp(), execvcc()}

Execl..

Execl (char \*name, char arg(0), .... , char arg(n), 0);

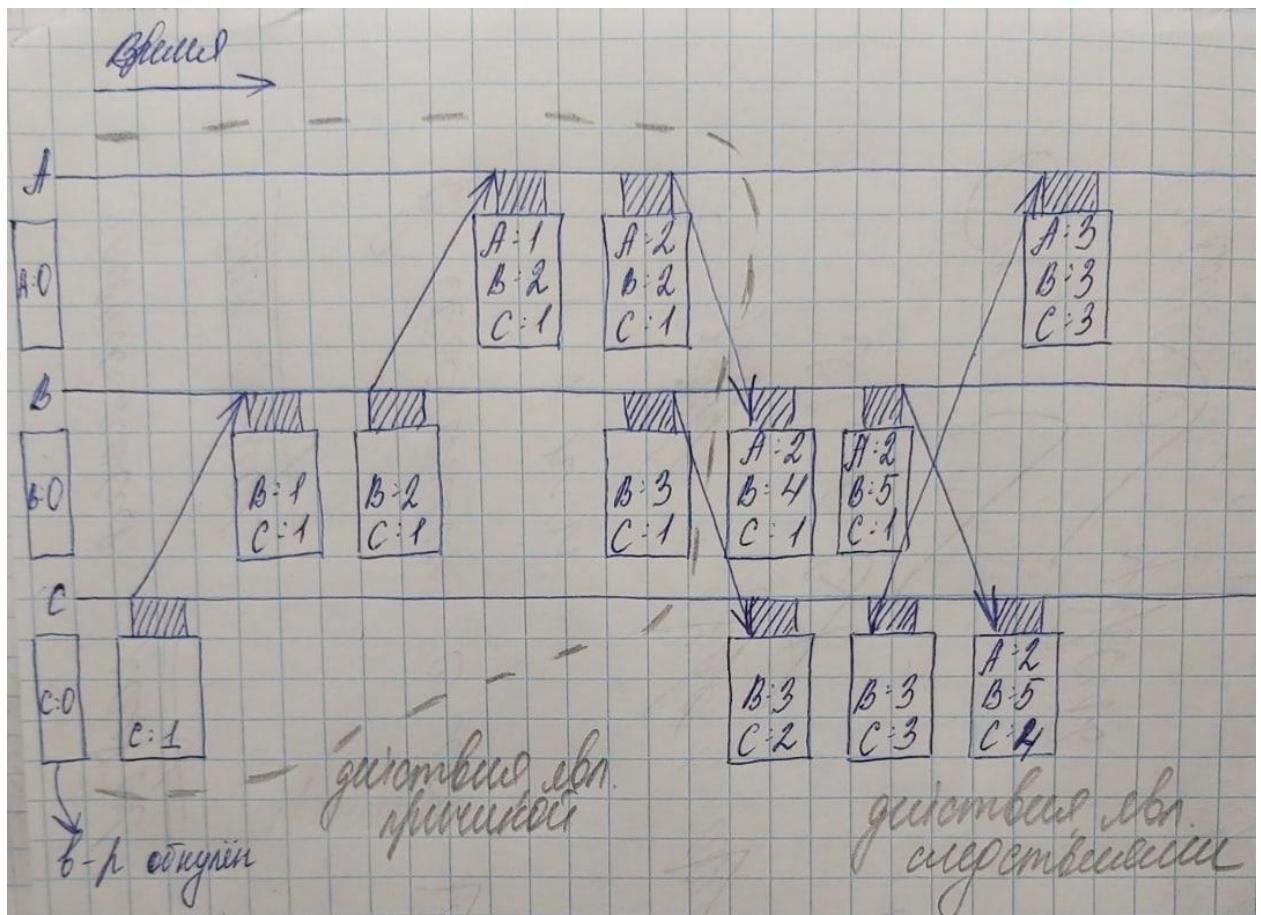
Все перечисленные системные вызовы имеют разный набор формальных параметров

## ВЕКТОРНЫЕ ЧАСЫ

Алгоритм векторные часы используется для обеспечения частично упорядоченных событий. Они широко используются в распределенных системах для обеспечения отношения причинности ("случилось до / случилось после"). В частности, векторные часы используется для обнаружения нарушения причинности.

В отличие от временных меток Лампорта, векторные часы содержат вектор отметок.

Каждый раз, когда какой-то процесс выполняет какие-то действия (когда в процессе происходит какое-то событие) в векторе отметок фиксируется отметка.



Взаимодействуют три процесса: А, В, С. В начальный момент времени вектор обнулен. В некоторый момент в каком-то процессе происходит событие: отправка или прием сообщения (собственно другие события, которые происходят на локальных машинах, нас не интересуют). Скажем, процесс в какой-то момент времени сформировал сообщение и сделал в своем векторе отметку и проставил значение 1.

Взаимодействие в распределенных системах связано с временными задержками.

Получение процессом сообщения так же является событием. Вектор, фиксированный процессом В - В:1. После этого процесс В формирует сообщение процессу А с отметкой В:2. Процесс А формирует ответное сообщение процессу В с отметкой А:2. (см ее комментарии к рисунку на записи, вроде в личный кабинет выложит (см. мыльный рисунок выше))

Важно различать причину и следствие. Пунктиром выделены действия, которые являются причиной. За пунктиром - следствия.

Каждый раз, когда процесс получает сообщение, он увеличивает собственную логическую переменную - собственные логические часы в векторе - +1. Каждое событие инкрементирует собственное значение счетчика событий процесса.

Данные вектора могут использоваться для обеспечения частичного порядка. Существуют соответствующие доказательства, что данный алгоритм - векторные часы - позволяет обеспечить свойство частичного порядка.

## RPC

RPC - *remote procedure call*

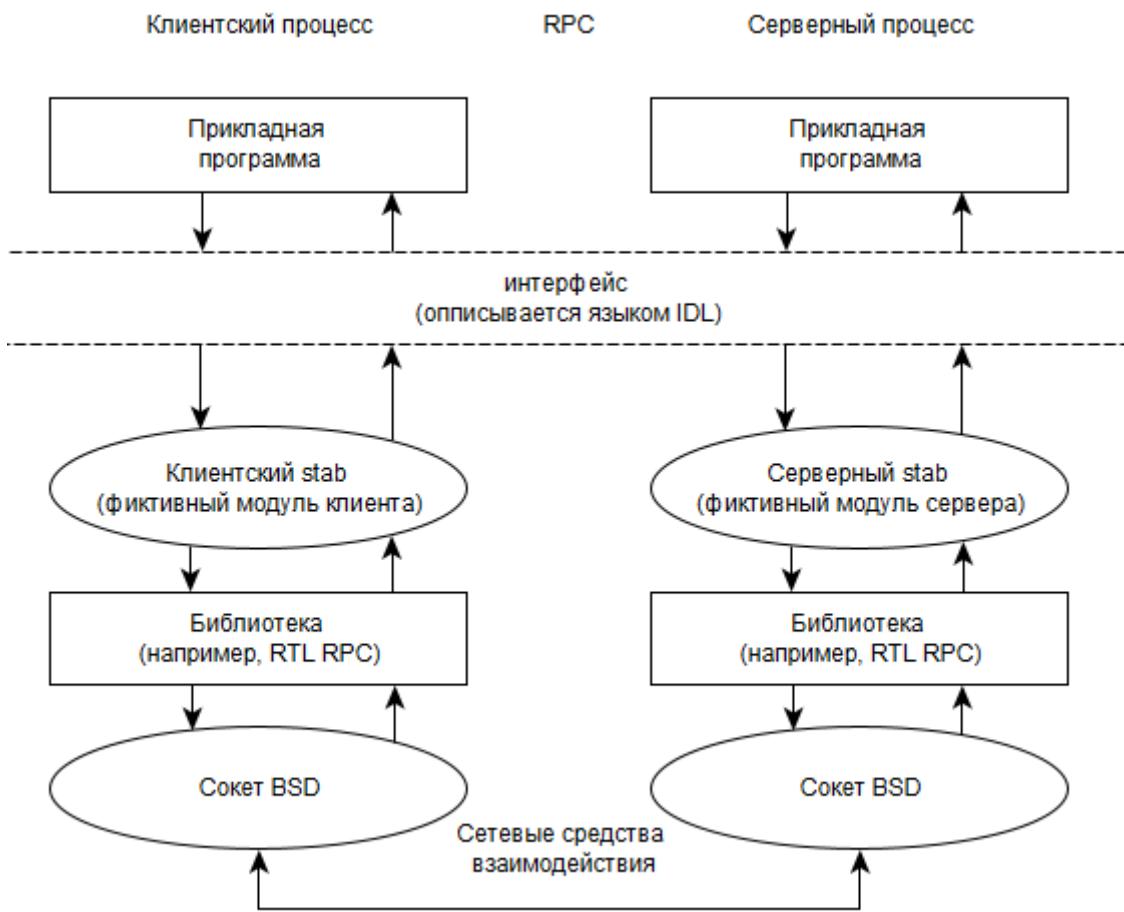
RPC является важнейшим механизмом для приложений типа клиент-сервер. Он был разработан Sun Microsystems. Он представляет собой набор библиотечных функций, например:

- RPC -> NIS (Network Information System)
- RPC -> NFS (Network File System)

## Основы

RPC - это механизм, с помощью которого один процесс активизирует другой процесс на этой же или удаленной машине для выполнения какой-то работы (функции) от своего имени. Этот механизм был реализован в Unix таким образом, что он напоминает вызов обычной (локальной) функции. Вызов функции связан с передачей этой функции каких-то данных, которые называются фактическими параметрами. Вызванная функция обрабатывает эти данные и в итоге, формирует результат, который возвращается основной функции, которая вызвала данную локальную функцию.

RPC внешне напоминает именно такие действия. Но вызываемая функция выполняется (может выполняться) на совершенно другой машине. Это взаимодействие выполняется по модели клиент-сервер. Процесс, который вызывает функцию RPC является клиентским. Процесс, который выполняет функцию на этой же или на другой машине, является серверным.



86:38

## RPC. Этапы работы.

### Клиентский процесс

- [прикладная программа]
- интерфейс (определяется с помощью языка IDL)
- (клиентский stab)
- [библиотека RTL RPC]
- (сокет BSD; клиент)

### Серверный процесс

- (сокет BSD; сервер)
- [библиотека RTL RPC]
- (серверный stab)
- интерфейс (определяется с помощью языка IDL)
- [прикладная программа сервера]

---

Формируется ответ и посыпается ответ обратно по той же схеме (снизу - вверх)

Прикладная программа заинтересована в выполнении каких-то действий на удаленной машине. Прикладная программа, выполняемая клиентским процессом, выполняет вызов удаленной процедуры. При этом для этой программы это просто вызов на локальной машине. В системе существует интерфейс, то есть вызов, выполняемый приложением, преобразуется в вызов так называемого клиентского stab'a (фиктивный модуль клиента; клиентская заглушка).

---

#### Описание работы:

Прикладная программа, выполняемая клиентским процессом, выполняет вызов удаленной процедуры. При этом для этой программы это просто вызов на локальной машине. В системе существует интерфейс, то есть вызов, выполняемый приложением, преобразуется в вызов так называемого клиентского stab'a (фиктивный модуль клиента; клиентская заглушка).

#### Вызов клиентского стаба, действия:

1. Подготовка буфера сообщения.
2. Упорядочивание параметров в буфере.
3. Добавление в сообщение полей заголовков.
4. Системный вызов, в результате которого происходит переход в режим ядра и выполняется переключение контекста.
5. В ядре для того, чтобы обработать сообщение, оно должно быть скопировано в буфер ядра (В режиме пользователя свой буфер, в режиме ядра - свой).
6. Определение адреса назначения.
7. Адрес помещается в заголовок сообщения.
8. Инициализируется сетевой интерфейс.
9. Включается таймер.

Первые 4 действия выполняет стаб клиента.

В результате вызова начинает выполняться stab. Происходит вызов функции стандартной библиотеки. Взаимодействие выполняется через сокеты.

В результате вызова библиотечной функции, происходит обращение к местному сокету (сокету BSD).

Все эти действия выполняются в рамках вызова клиентской программы на локальной машине, на которой выполняется клиентский процесс. В результате

задействования сокетов, задействуются сетевые средства (т. н. транспортный уровень). Осуществляется передача сообщения.

Итог действий на стороне клиента - формирование сетевого пакета, который передается процессу-серверу.

На серверной стороне через сокет пакет поступает на машину, на которой выполняется процесс-сервер. Вызывается библиотека и её функции. На стороне сервера имеется серверный stab.

Таким образом, клиентский stab вызывает последовательность действий, вызывает в процессе выполнения. В процессе выполнения клиентского стаба выполняется системный вызов. Система переходит в режим ядра. В режиме ядра выполняются совершенно специфические для RPC действия, связанные с формированием сообщения. После того, как сообщение сформировано, это сообщение по сети передается другой машине. Воспринимается через сокет в режиме ядра, распаковывается в режиме ядра, передается серверному стабу, затем передается прикладной программе.

## [СРАВНЕНИЕ КАНАЛЫ, СООБЩЕНИЯ, РАЗДЕЛЯЕМАЯ ПАМЯТЬ](#) [\(есть еще такая версия\)](#)

Передача сообщений.

ПС – наиболее общая форма взаимодействия процессов. То есть передача сообщений будет работать как на отдельно-стоящих машинах, так и в распределенных системах.

Важно различать 2 понятия. Обычно не делается различие между взаимоисключением и синхронизацией. Но это 2 разных понятия.

В – монопольного доступа. С – один процесс не может начать свою работу, если другой не сделал в интересах первого процесса каких-то действий. Будет ждать (сообщения, например). Мы всегда говорим об асинхронных процессах – собственные скорости, невозможно предсказать, когда процесс придет в точку

Это различие можно увидеть в производство-потребление и читатели-писатели.

Задача производство-потребление связана с синхронизацией. Потребитель не сможет работать, если в буфере нет данных, если буфер пуст. Будет ждать, пока производитель произведет единицу данных. Аналогично производитель не может начать работу, если все ячейки буфера заполнены – будет ждать, когда потребитель освободит хотя бы одну ячейку. Поэтому по 2 переменные (полон пуст) или (2 переменные типа условие)

Читатель-писатель не связаны. Читатель только ждет, пока писатель освободит критическую секцию. Так же и писатель может начать работать, если в конкретный момент времени никакой читатель или писать не захватили, неважно что там было до.

Программный канал (pipe) буферизуется на 3 уровнях (системная область памяти, диск, время).

На 1 уровне трубы буферизуются в системной памяти (=в области данных ядра системы). Обычно не может превышать 4096 байт (1 страница).

При переполнении системной памяти программные каналы (буфера), имеющие наибольшее время существования, переписываются во вторичную память (диск).

Ф использует Стандартное функции управления или работы с файлами. Если процесс пытается записать в трубу больше 4096 байт, то труба буферизуется во времени, приостанавливая процесс до тех пор, пока все данные не будут прочитаны.

Ограничение размера канала основано на повышении эффективности, так как при этом не используется обращение к медленной внешней памяти.

Еще раз – разница между каналами и разделяемой памятью.

Канал обеспечивает потоковую модель передачи данных. При считывании сообщения оно перестает существовать. Он имеет встроенные средства взаимоисключения.

Разделяемая память не имеет. Это буфер, подключающийся к АП, нет средств взаимоисключений, поэтому их обычно используют с семафорами. Что положили – так и будет там лежать

сообщение

В отличие от разделяемых сегментов, здесь копирование. При посылке – из буфера программы в область данных ядра. При чтении – из области данных ядра в буфер программы.

## Тупики Habermann

Как узнать, что состояние безопасное?

- Начинаем с заданного состояния
- Имитируем выполнение каждого процесса, выделяя ему максимальное количество запрошенных ресурсов
- Если все процессы завершатся, то состояние безопасное

Habermann's Algorithm (обобщение алгоритма Dijkstra)

$n$  процессов

$m$  типов ресурсов

- Матрица доступных ресурсов

Max-Avail matrix  $A = (a_1, a_2, \dots, a_m)$ , где  $a_i = t_i = |R_i|$

- Матрица заявок

$$b_{11} \ b_{12} \ \dots \ b_{1m} \quad B_1$$

Max-Claim matrix  $B = \begin{matrix} b_{21} & b_{22} & \dots & b_{2m} \end{matrix} = B_2$

...    ...

$$b_{n1} \ b_{n2} \ \dots \ b_{nm} \quad B_n$$

где  $b_{ij}$  – максимальное количество единиц ресурса  $R_j$  которое могло быть когда-нибудь удерживаться процессом  $P_i$

- Матрица распределения

$$c_{11} \ c_{12} \ \dots \ c_{1m} \quad C_1$$

Allocation matrix  $C = \begin{matrix} c_{21} & c_{22} & \dots & c_{2m} \end{matrix} = C_2$

...    ...

$$c_{n1} \ c_{n2} \ \dots \ c_{nm} \quad C_n$$

где  $c_{ij}$  – количество единиц ресурса  $R_j$  которое в текущий момент удерживается процессом  $P_i$

1. Все  $k$ ,  $B_k \leq A$  – процесс не может требовать больше единиц ресурса, чем доступно
2.  $C \leq B$  – процесс не может пытаться запросить больше ресурсов, чем указано в его заявке

n

3.  $\sum C_k \leq A$  – никогда не выделяется больше ресурсов, чем доступно

$k=1$

4. Матрица доступных ресурсов D:

n

$$D = (d_1, d_2, \dots, d_m) = A - \sum_{k=1}^n C_k$$

$k=1$

$d_i$  – количество единиц ресурса  $R_i$  доступное в текущем состоянии.

5. Матрица возможных запросов E

$$e_{11} e_{12} \dots e_{1m} \quad E_1$$

$$\text{Need matrix } E = \begin{matrix} e_{21} e_{22} \dots e_{2m} \end{matrix} = B - C = E2$$

... ...

$$e_{n1} e_{n2} \dots e_{nm} \quad E_n$$

6. Матрица запросов F

$$f_{11} f_{12} \dots f_{1m} \quad F_1$$

$$\text{Request matrix } C = \begin{matrix} f_{21} f_{22} \dots f_{2m} \end{matrix} = F2$$

... ...

$$f_{n1} f_{n2} \dots f_{nm} \quad F_n$$

7. Предварительное состояние:

Предположим, что удовлетворяются все запросы и посмотрим, что случится:

$D \leftarrow D - F_i$  доступно : запросы

$C_i \leftarrow C_i + F_i$  распределено : запрошено

$E_i \leftarrow E_i - F_i$  указано, что нужно : запрошено

Запрос удовлетворен, если только если предыдущее состояние является безопасным.

Алгоритм проверки безопасного состояния

1. Выбираем незавершившийся процесс  $P_i$  такой что  $E_i \leq D$  (т.е. доступно больше ресурсов чем нужно  $P_i$ ). Если нет такого процесса, переходим на шаг 3
2.  $D \leftarrow D + C_i$  Отмечаем  $P_i$  как законченный и переходим на шаг 1
3. Если все процессы отмечены как законченные, то система находится в безопасном состоянии, иначе в опасном

Если система находится в небезопасном состоянии, то запрос блокируется и состояние системы сбрасывается с помощью:

$D \leftarrow D - F_i$

Откат(все 3)       $C_i \leftarrow C_i + F_i$

$E_i \leftarrow E_i - F_i$

В добавок, процесс, помеченный как завершенный на шаге 2 сбрасывается

$D \leftarrow D - C_i$

и повторно помечается как незавершенный.

**Пример:**

$r_i$

Max-Avail A = (2 4 3)      p/r ресурсы - заявки

1 2 2

1 2 2

Max-Claim B = 1 2 1 => P<sub>i</sub> 1 2 1  
 1 1 1                          1 1 1  
 r<sub>1</sub> r<sub>2</sub> r<sub>3</sub>

1 2 0 P<sub>1</sub>

Allocation C = 0 1 1 P<sub>2</sub>

распределено 1 0 1 P<sub>3</sub>

A                           $\sum C_k$  1..3

Available D = (0 1 1) => (2 4 3) - (2 3 2) = (0 1 1)

доступно

0 0 2

Возможные запросы E = 1 1 0 => B<sub>i</sub> - C<sub>i</sub>

0 1 0

Пусть P<sub>1</sub> делает запрос F<sub>1</sub> = (0 0 1) – запрашивает единицу ресурса r<sub>3</sub>

Предварительное состояние будет:

Доступно D = (0 1 0) т.к. P<sub>1</sub> взял единицу

1 2 1

Матрица распределения C = 0 1 1

1 0 1

0 0 1

Возможные запросы E = 1 1 0

0 1 0

В этом состоянии P<sub>3</sub> может завершиться, т.к. E<sub>3</sub> ≤ D и вернет удерживаемые ресурсы (101) в пул доступных/свободных ресурсов. В результате D станет (111) ← 010 + 101.

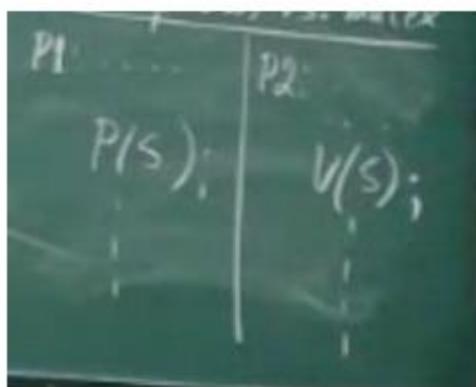
В результате можно будет удовлетворить  $P_1$  и  $P_2$  и, следовательно, предварительное состояние в результате запроса  $P_1$  является безопасным и запрос  $F_1$  может быть удовлетворен.

## СРАВНЕНИЕ МЬЮТЕКСОВ И СЕМАФОРОВ

Очень часто binC называют M. Это ошибка.

1) У мьютексов всегда есть владелец. Владельцем является процесс, который захватил M (get\_mutex, lock\_mutex – неважно). Только владелец M может M освободить (unlock).

Для C такого понятия не существует. С может захватить один процесс, а освободить – совершенно другой. То есть для C может быть написано



Для M – нет. Это основное отличие, которое делает C особенноми.

2) в отличие от C, на мьютексах определена инверсия приоритетов. Это связано с тем, что никакой другой более высокоприоритетный процесс не сможет перехватить M.

3) в силу того, что M может быть освобожден только процессом захватившем M, никакого случайного удаления процесса, захватившего M это невозможно. Для C это не так. Если бы ЗМ умер, все остальные могут быть заблокированы навсегда, поэтому unix очень аккуратно там за всем следит.

**Алгоритм на основе стека** — это тот **алгоритм**, для которого можно показать, что набор страниц в памяти для  $N$  кадров всегда является подмножеством набора страниц, который будет в памяти с  $N + 1$  кадрами.

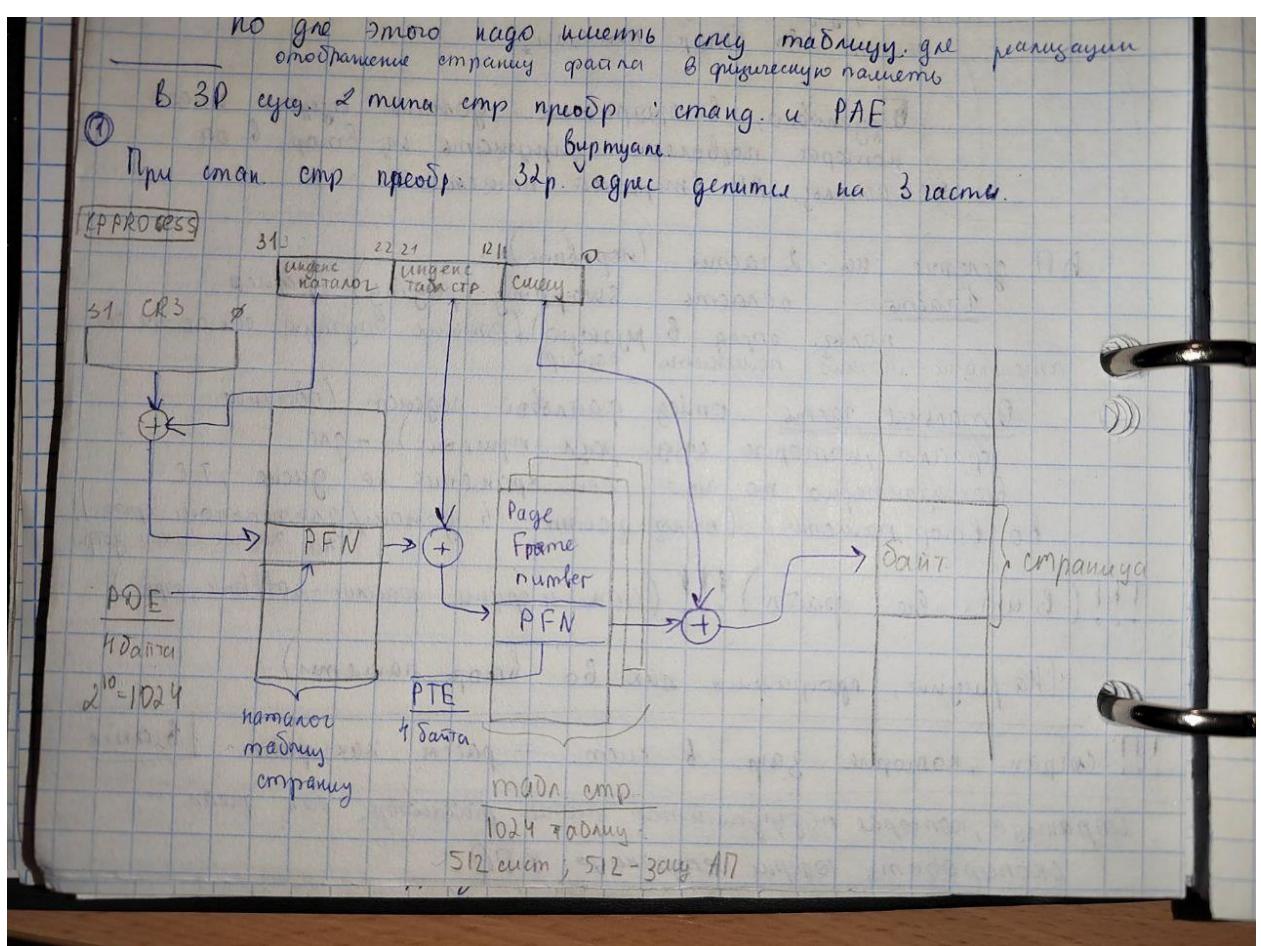
реализация страничного преобразования в компьютерах на базе процессоров Intel (x86): стандартное преобразование и PAE в защищенном режиме – схемы, размеры таблиц и их количество на каждом этапе преобразования.

### стандартное преобразование

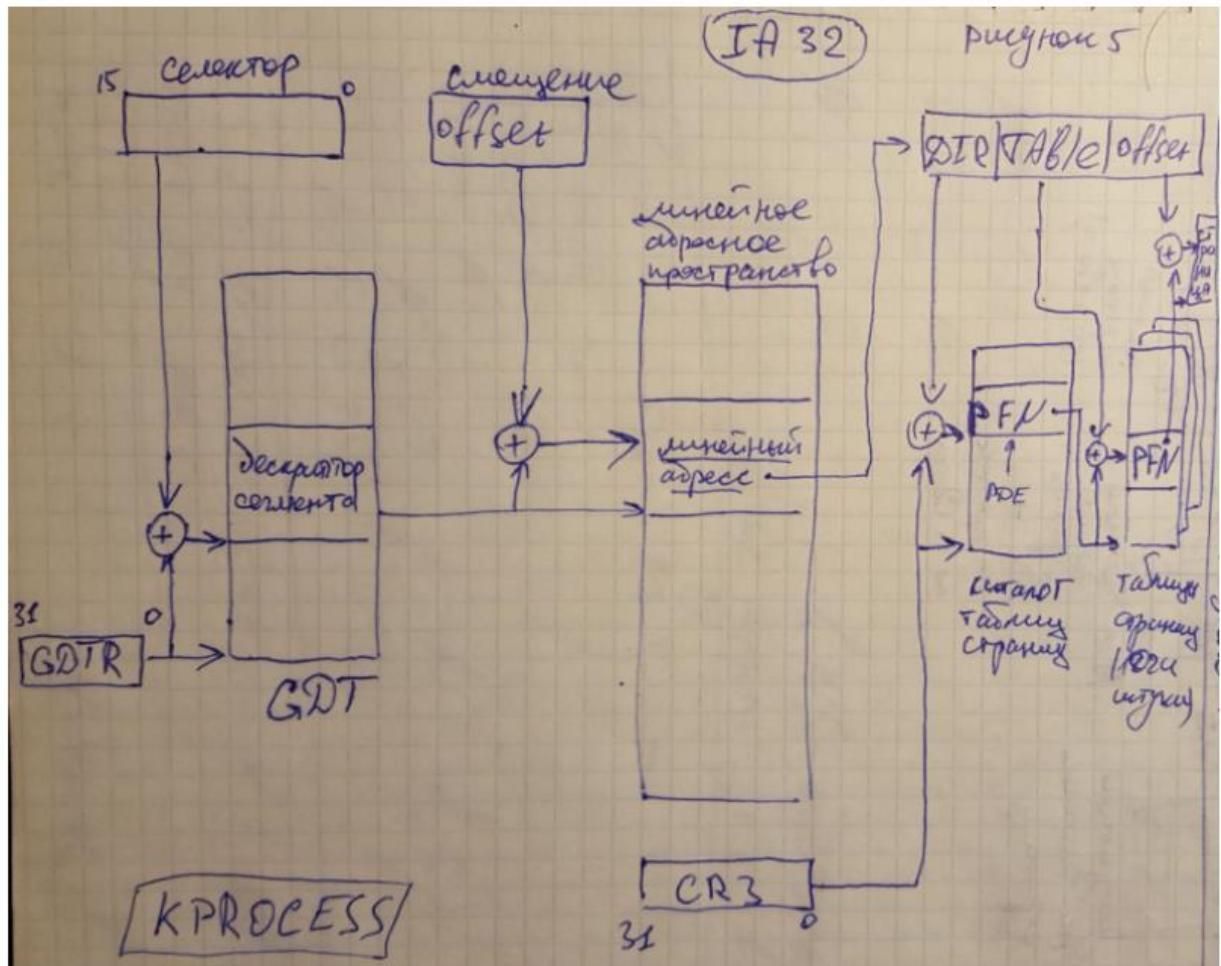
В ЗР 2 типа страничного преобразования – стандартное и PAE. При стандартном 32р виртуальный адрес делится на 3 части. Эта схема коррелирует со схемой гиперстраниц.

4кб страницы – не единственный поддерживаемый размер, но это обычный размер для приложений. Есть схемы с 1кб и 2кб - для баз данных. Но стандартно 4кб, тогда смещение 12 бит ( $2^{12} = 4\text{кб}$ ).

Приняты обозначения: PDE – page directory entry, PFN – page frame number, PTE – page table entry. CR3-загружается из Kprocesses (во всяком случае так декларируется). Начальный адрес таблицы страниц – в структуре, описывающей процесс. Может быть 1024 таблицы страниц.



КТС один на систему и может содержать 1024 строки. PDE - 4 байта. PFE (page frame number) – таблица страниц. 512 таблиц на систему, 512 – на процесс. В регистре CR3 базовый адрес каталога таблиц страниц. Адрес (в регистре cr3) берется из дескриптора процесса. На шине адреса будет всегда находиться линейный адрес, т.е. адрес байта физической памяти. К преобразованиям шина адреса не имеет никакого отношения.



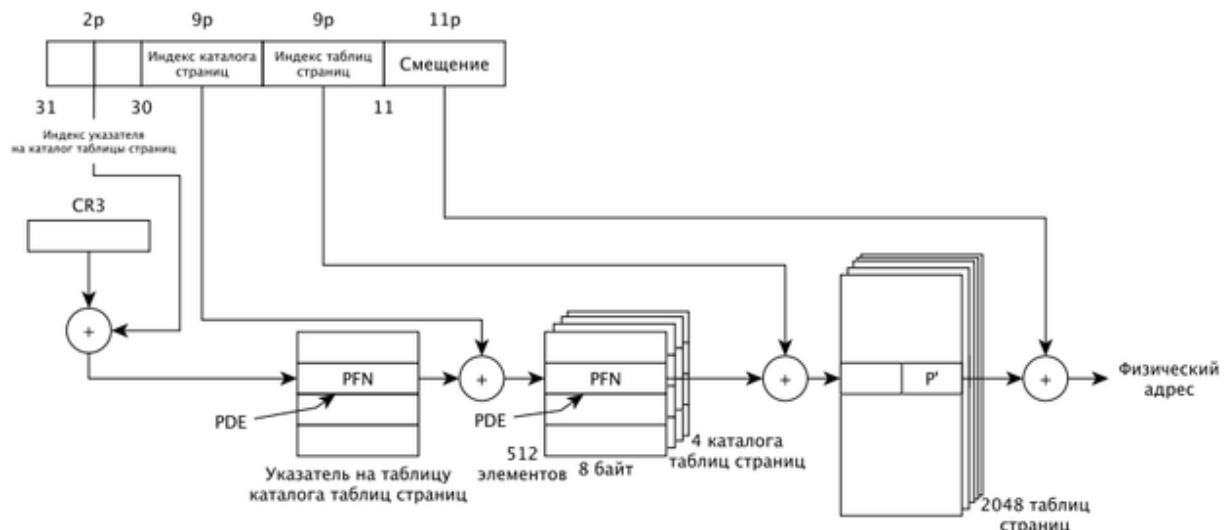
32р виртуальный адрес делится на 4 поля. Каждое такое деление связано с дополнительным обращением к ОП при преобразованиях. Каталог ТС, базовый адрес ТС, адрес страницы. Минус, но зато ТОЛЬКО актуальные таблицы.

Такая схема появилась в Pentium pro, где появился дополнительный регистр CR4. Появилось поле размером 2 бита – можно адресовать таблицу из 4 элементов – таблицу указателей на каталоги PDPT. В этой схеме все дескрипторы имеют размер 8 байт. Появляется возможность адресовать 4 таблицы каталогов таблиц страниц. Все таблицы содержат дескрипторы размером 8 байт.  $2^9 = 512$  элементов в каждой таблице каталога. Итого – 2048 таблиц страниц.

Эта схема в 32р введена для возможности адресации АП > 4ГБ

Посмотрим на соответствующие дескрипторы. Рассмотрим дескриптор таблицы страниц PTE для 4 Кб страницы. 11 разрядов смещения используются в виде флагов и оставшиеся разряды – начальный адрес страницы – PageBaseAddress

31, 30	индекс	индекс	12 бит
индекс указателя на каталог страниц.	каталога страниц	таблиц страниц	смещение offset



### Адресация памяти в ЗР

В защищённом режиме, также как и в реальном, существуют понятия логического и физического адреса. Логический адрес в защищённом режиме (иногда используется термин "виртуальный адрес") состоит из двух 16-разрядных компонент - селектора и смещения.

Селектор записывается в те же сегментные регистры, что и сегментный адрес в реальном режиме.

Однако преобразование логического адреса в физический выполняется не простым сложением со сдвигом, а при помощи специальных таблиц преобразования адресов.

На рис.4 показана сильно упрощённая схема преобразования логического адреса в физический.

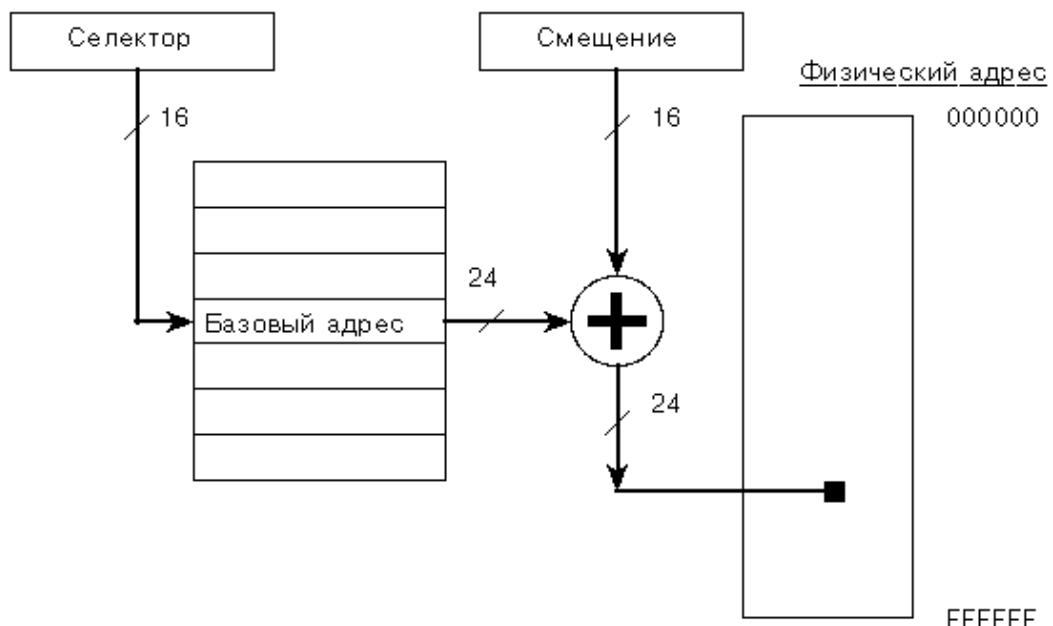


Рис. 4. Упрощённая схема преобразования логического адреса в физический в защищённом режиме.

### Формат селектора.

## АЛГОРИТМЫ ЗАМЕЩЕНИЯ СЕГМЕНТОВ

### Алгоритмы

1. Выталкивание случайного сегмента.
  - Самый простой способ реализации - первая попавшаяся
  - Недостатки:
    - может быть вытолкнут часто использующийся сегмент либо только что загруженный
2. Алгоритм FIFO: вытеснение сегмента, который дольше всего находится в памяти. Способы: временная метка (вытеснение с меньшей временной меткой) либо ведется связный список. Этот алгоритм исключает возможность выгрузки только что загруженного сегмента, но не исключает выгрузку часто использующегося.
3. Алгоритм LRU (Least Recently Used): выталкивание того, что дольше всех не использовался. Можно использовать временные метки (но надо постоянно обновлять при каждом обращении), либо можно использовать связные списки (перемещаем в начало при обращении).
 

Этот способ обладает свойством включения (*Если какой-то сегмент выбран при реализации с объемом памяти M, то этот же сегмент при той*

- же трактории будет выбран, если память  $M + 1$  страниц)*  
 LRU относится к классу стековых алгоритмов. Затраты на реализацию огромные, в чисто виде не используется.
4. Алгоритм LFU - least frequency used. "Наименее часто использующийся в последнее время". В этом алгоритме есть стратегия - контролируется частота использования страниц. Но, наименее часто использованной может оказаться только что загруженная страница. То есть она еще не успела набрать частоту обращений. Поэтому вероятность того, что она будет вытолкнута весьма.
  5. Алгоритм NUR (Not Used Recently): Апроксимация LRU  
 Для более простой реализации:
    - Когда сегмент загружается в память бит = 1
    - При обращении бит = 1
    - Периодически все сбрасывается в 0
    - Когда требуется вытеснить сегмент, вытесняется первый с битом = 0

**Логические часы, для чего надо синхронизировать время, какое соотношение должно выполняться в распределенных системах.**

**- соотношение случилось до - случилось после**

**выделение разделяемого сегмента.**

**- все функции описаны ниже + выделены в вопросе**

**любой сигнал имеет обработчик, обработчик - это код ядра, установлен по умолчанию. когда мы говорили о сигналах, указывалось 3 способа реагирования процесса на пол сигнал: (они есть у нас).**

## ПРОИЗВОДСТВО - ПОТРЕБЛЕНИЕ

про сис вызовы дл сем [системные вызовы, поддержка в системе](#)

```
#include "buffer.h"
#include <string.h>

int init_buffer(buffer_s* const buffer)
{
    if (!buffer)
        return -1;
    memset(buffer, 0, sizeof(buffer_s));
    return 0;
}

int write_buffer(buffer_s* const buffer, const char elem)
{
    if (!buffer)
        return -1;
    buffer->data[buffer->wpos++] = elem;
    buffer->wpos += 1;
}
```

```

        return 0;
    }

int read_buffer(buffer_s* const buffer, char* const dest)
{
    if (!buffer)
        return -1;

    *dest = buffer->data[buffer->rpos++];
    buffer->rpos += 1;
    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>

#include "consumer.h"
#include "buffer.h"

// Потребитель
struct sembuf ConsumerBegin[2] =
{
    {SF, P, 0}, // Ожидает, что будет заполнена хотя бы одна
    // ячейка буфера.
    {SB, P, 0} // Ожидает, пока производитель или
    // потребитель выйдет из критической зоны.
};

struct sembuf ConsumerEnd[2] =
{
    {SR, V, 0}, // Освобождает критическую зону.
    {SE, V, 0} // Увеличивает кол-во пустых ячеек.
};

void consumer_run(buffer_s* const buffer, const int sem_id, const int con_id)
{
    char ch;
    // Создаем случайные задержки
    int sleep_time = rand() % CONSUMER_DELAY_TIME + 1;
    sleep(sleep_time);

    // Получаем доступ к критической зоне
    int rv = semop(sem_id, ConsumerBegin, 2);
    if (rv == -1)
    {
        perror("Потребитель не может изменить значение семафора.\n");
        exit(-1);
    }

    // Началась критическая зона
    if (read_buffer(buffer, &ch) == -1)
    {
        perror("Something went wrong with buffer reading!");
        exit(-1);
    }
    printf("\e[1;33mConsumer #%d \tread: \t%c \tsleep: %d\e[0m\n", con_id, ch,
sleep_time);
    // Закончилась критическая зона

    rv = semop(sem_id, ConsumerEnd, 2);
    if (rv == -1)
    {
        perror("Потребитель не может изменить значение семафора.\n");
        exit(-1);
    }
}

void consumer_create(buffer_s* const buffer, const int con_id, const int sem_id)
{
    pid_t childpid;
    if ((childpid = fork()) == -1)
    {
        // Если при рождении процесса произошла ошибка
        perror("Ошибка при рождении процесса потребителя.");
        exit(-1);
    }
    else if (!childpid) // childpid == 0
    {
}

```

```

        // Это процесс потомок.
        // Каждый потребитель потребляет
        // TTFRATTNS AMOUNTSAmount.
        for (int i = 0; i < TTFRATTNS AMOUNT; i++)
            consumer_run(buffer, sem_id, con_id);
        exit(0);
    }

#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>

#include "producer.h"
#include "buffer.h"

// В структуре struct sembuf состоящей из полей:
//     short sem_num; /* semaphore number: 0 = first */
//     short sem_op; /* semaphore operation */
//     short sem_flg; /* operation flags */

struct sembuf producer_begin[2] =
{
    {SE, P, 0}, // Ожидает освобождения хотя бы одной ячейки
    // в буфере.
    {SB, P, 0} // Ожидает пока другой производитель или
    // потребитель выйдет из критической зоны.
};

struct sembuf producer_end[2] =
{
    {SB, V, 0}, // Освобождает критическую зону.
    {SF, V, 0} // Увеличивает кол-во заполненных ячеек.
};

void producer_run(buffer_s* const buffer, const int sem_id, const int pro_id)
{
    int sleep_time = rand() % PRODUCER_DELAY_TIME + 1;
    sleep(sleep_time);

    // Получаем доступ к критической зоне.
    int rv = semop(sem_id, producer_begin, 2);
    if (rv == -1)
        perror("Производитель не может изменить значение семафора.\n");
    exit(-1);

    // Началась критическая зона
    // Положить в буфер.
    //const char symb = buffer->wpos % 26 + 'a';
    const char symb = ALPHABET[buffer->wpos];
    if (write_buffer(buffer, symb) == -1)
        exit(-1);
    printf("\e[1;32mProducer #%d \twrite: \t%c \tsleep: %d\e[0m\n", pro_id,
    symb, sleep_time);
    // Закончилась критическая зона

    rv = semop(sem_id, producer_end, 2);
    if (rv == -1)
        perror("Производитель не может изменить значение семафора.\n");
    exit(-1);
}

void producer_create(buffer_s* const buffer, const int pro_id, const int sem_id)
{
    pid_t childpid;
    if ((childpid = fork()) == -1)
        // Если при порождении процесса произошла ошибка.
        perror("Ошибка при порождении процесса производителя.");
}

```

```

        exit(-1);
    }
    else if (!childpid) // childpid == 0
        // Это процесс потомок.
        // Каждый производитель производит
        // TTFRATTONS AMOUNT сообщений
        for (int i = 0; i < TTFRATTONS AMOUNT; i++)
            producer_run(buffer, sem_id, pro_id);
        exit(0);
}

#include <stdio.h>
#include <stdlib.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <time.h>
#include <wait.h>
#include <unistd.h>
#include <sys/stat.h>

#include "buffer.h"
#include "constants.h"
#include "producer.h"
#include "consumer.h"

int main(void)
{
    setbuf(stdout, NULL);
    srand(time(NULL));

    // Значение IPC_PRVATE указывает, что к разделяемой
    // памяти нельзя получить доступ другим процессам.
    // shmat - создает новый разделяемый сегмент.
    // S_ISSLR владелец может читать.
    // S_IWSSLR владелец может писать.
    // S_IRGRP группа может читать.
    // S_IROTH остальные могут читать.
    int sem_descr;
    int perms = S_ISSLR | S_IWSSLR | S_IRGRP | S_IROTH;
    int shmid = shmget(IPC_PRIVATE, sizeof(buffer_s), IPC_CREAT
| perms);
    if (shmid == -1)
        perror("Ошибка при создании нового разделяемого сегмента.\n");
    return -1;

    // Функция shmat() возвращает указатель на сегмент
    // shmaddr (второй аргумент) равно NULL,
    // то система выделяет подходящий (неиспользуемый)
    // адрес для подключения сегмента.
    buffer_s *buffer;
    if ((buffer = (buffer_s*)shmat(shmid, 0, 0)) == (buffer_s*)-
1)
    {
        perror("Ошибка при попытке возврата указателя на сегмент.\n");
        return -1;
    }
    if (init_buffer(buffer) == -1)
        perror("Ошибка при инициализации буфера.\n");
    return -1;

    // IPC_PRVATF - ключ, который показывает, что
    // набор семафоров могут использовать только процессы,
    // Поподчиненные процессом, создавшим семафор.
    // Создание семафоров (3 семафора)
    sem_descr = semget(IPC_PRIVATE, 3, IPC_CREAT | perms);
    if (sem_descr == -1)

```

```

        perror("Ошибка при создании набора семафоров.");
    }

    if (semctl(sem_descr, SB, SETVAL, 1) == -1)
    {
        perror("!!! Can't set control semaphors." );
        return -1;
    }

    if (semctl(sem_descr, SE, SETVAL, N) == -1)
    {
        perror("!!! Can't set control semaphors." );
        return -1;
    }

    if (semctl(sem_descr, SF, SETVAL, 0) == -1)
    {
        perror("!!! Can't set control semaphors." );
        return -1;
    }

    for (int i = 0; i < COUNT_PRODUCER; i++)
    {
        producer_create(buffer, i + 1, sem_descr);
    }

    for (int i = 0; i < COUNT_CONSUMER; i++)
    {
        consumer_create(buffer, i + 1, sem_descr);
    }

    for (size_t i = 0; i < COUNT_PRODUCER + COUNT_CONSUMER; i++)
    {
        int status;
        if (wait(&status) == -1)
        {
            perror("Something wrong with children waiting!");
            return -1;
        }
        if (!WTFEXTTFD(status))
            printf("One of children terminated abnormally\n");
    }

    printf("\e[1;34mOk\n");
}

// IPC_RMID используется для пометки сегмента как
удаленного.

if (shmctl(shmid, IPC_RMID, NULL))
{
    perror("Ошибка при попытке пометить сегмент как удаленный.");
    return -1;
}

```

**Функция `shmdt` отстыковывает сегмент разделяемой памяти находящийся по адресу `shmaddr` от адресного пространства вызвавшего процесса. Отстыковываемый сегмент должен быть среди пристыкованных ранее функцией `shmat`.**

```

if (shmdt((void*)buffer) == -1)
{
    perror("Ошибка при попытке отключить разделяемый сегмент от адресного
пространства процесса ");
    return -1;
}

if (semctl(sem_descr, 0, IPC_RMID, 0) == -1)
{
    perror("Ошибка при попытке удаления семафора.");
    return -1;
}

return 0;
}

```

## ЧИТАТЕЛИ - ПИСАТЕЛИ

```
#include <sys/sem.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include "reader.h"
#include "constants.h"

extern int *counter;

struct sembuf StartRead[5] =
{
    {CAN_RREAD, V, 0},
    {WATT_WRTTFRS, S, 0}, // проверка, если ли ждущие писатели
    {CAN_WRTTF, S, 0}, // проверка, что писатель не пишет
    {ACTTWF_RFADFRS, V, 0}, // инкремент активных читателей
    {CAN_READ, P, 0}
};

struct sembuf StopRead[1] =
{
    {ACTIVE_READERS, P, 0} // Декремент активных читателей
};

int start_read(int sem_id)
{
    return semop(sem_id, StartRead, 5);
}

int stop_read(int sem_id)
{
    return semop(sem_id, StopRead, 1);
}

void reader_run(const int sem_id, const int reader_id)
{
    int sleep_time = rand() % READER_SLEEP_TIME + 1;
    sleep(sleep_time);

    int rv = start_read(sem_id);
    if (rv == -1)
    {
        perror("Читатель не может изменить значение семафора.\n");
        exit(-1);
    }

    // Началась критическая зона
    printf("\e[1:33mReader #%d \tread: \t%d \tsleep: %d\0m \n",
           reader_id, *counter, sleep_time);
    // Закончилась критическая зона

    rv = stop_read(sem_id);
    if (rv == -1)
    {
        perror("Читатель не может изменить значение семафора.\n");
        exit(-1);
    }
}

void reader_create(const int sem_id, const int reader_id)
{
    pid_t childpid;
    if ((childpid = fork()) == -1)
    {
        perror("Ошибка при порождении читателя.");
        exit(-1);
    }
}
```

```

    }
    else if (childpid == 0)
    {
        // Это процесс потомок.
        //for (int i = 0; i < 4; i++)
        while (1)
            reader_run(sem_id, reader_id);
        exit(0);
    }
}

```

```

#include <sys/sem.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include "constants.h"
#include "writer.h"

extern int *counter;

struct sembuf StartWrite[6] =
{
    {WAIT_WRITERS, V, 0},      // инкремент счётчика ждущих писателей
    {ACTIVE_READERS, S, 0},     // проверка, есть ли активный читатель
    {CAN_WRITE, S, 0},         // проверка, пишет ли другой писатель
    {CAN_WRITE, V, 0},         // захват семафора активного писателя
    {CAN_READ, V, 0},          // захват семафора может ли читать (то есть запрет
    чтения)
    {WAIT_WRITERS, P, 0}        // декремент счётчика ждущих писателей
};

struct sembuf StopWrite[2] = {
    {CAN_READ, P, 0},          // Разрешает читать
    {CAN_WRITE, P, 0}           // Разрешает писать. освобождение активного
    писателя
};

// Функция производит операции над выбранными элементами из набора
// семафоров semid(1).
// Каждый из элементов nsops(3) в массиве sops(2) определяет операцию,
// производимую над семафором в структуре struct sembuf
int start_write(int sem_id)
{
    return semop(sem_id, StartWrite, 6);
}

int stop_write(int sem_id)
{
    return semop(sem_id, StopWrite, 2);
}

void writer_run(const int sem_id, const int writer_id)
{
    int sleep_time = rand() % WRITER_SLEEP_TIME + 1;
    sleep(sleep_time);
}

```

```

int rv = start_write(sem_id);
if (rv == -1)
{
    perror("Писатель не может изменить значение семафора.\n");
    exit(-1);
}

// Началась критическая зона
(*counter)++;
printf("\e[1;32mWriter #%d \twrite: %d \tsleep: %d\e[0m\n",
       writer_id, *counter, sleep_time);
// Закончилась критическая зона

rv = stop_write(sem_id);
if (rv == -1)
{
    perror("Писатель не может изменить значение семафора.\n");
    exit(-1);
}
}

void writer_create(const int sem_id, const int writer_id)
{
    pid_t childpid;
    if ((childpid = fork()) == -1)
    {
        perror("Ошибка при порождении писателя.");
        exit(-1);
    }
    else if (childpid == 0)
    {
        // Это процесс потомок.
        //for (int i = 0; i < 3; i++)
        while (1)
            writer_run(sem_id, writer_id);
        exit(0);
    }
}

```

```

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <sys/shm.h>

#include "constants.h"
#include "writer.h"
#include "reader.h"

int *counter = NULL;

int main(void)
{
    int sem_descr;
    int status;

```

```

// Функция shmget() создает новый разделяемый сегмент или,
// если сегмент уже существует, то права доступа подтверждаются.
// S_IRUSR Владелец может читать.
// S_IWUSR Владелец может писать.
// S_IRGRP Группа может читать.
// S_IROTH Остальные могут читать.
    int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    int shmid = shmget(IPC_PRIVATE, sizeof(int), perms);
    if (shmid == -1)
    {
        perror("Ошибка при создании разделяемого сегмента.\n");
        return -1;
    }

// Функция shmat() возвращает указатель на сегмент
    counter = shmat(shmid, NULL, 0);
    if (*(char *)counter == -1)
    {
        perror("Ошибка при возврата указателя на сегмент.\n");
        return -1;
    }
    *counter = 0;

// Функция semget() создает новый набор семафоров или открывает уже имеющийся.
// -1 в случае не удачи
// Если значением key является макрос IPC_PRIVATE,
// то создается набор семафоров, который смогут использовать только процессы,
// порожденные процессом, создавшим семафор.
    sem_descr = semget(IPC_PRIVATE, 4, IPC_CREAT | perms);

    if (sem_descr == -1)
    {
        perror("Ошибка при создании набора семафоров.");
        return -1;
    }

// Функция semctl() позволяет изменять управляющие параметры набора семафоров
// указанным в semid(1 арг) или над семафором с номером semnum(2 арг) из этого
// набора.
// https://www.opennet.ru/man.shtml?topic=semctl&category=2&russian=0
    if (semctl(sem_descr, CAN_WRITE, SETVAL, 0) == -1)
    {
        perror( "!!! Can't set control semaphors." );
        return -1;
    }

    if (semctl(sem_descr, CAN_READ, SETVAL, 0) == -1)
    {
        perror( "!!! Can't set control semaphors." );
        return -1;
    }

    if (semctl(sem_descr, ACTIVE_READERS, SETVAL, 0) == -1)
    {
        perror( "!!! Can't set control semaphors." );
        return -1;
    }

```

```

if (semctl(sem_descr, WAIT_WRITERS, SETVAL, 0) == -1)
{
    perror( "!!! Can't set control semaphors." );
    return -1;
}

for (int i = 0; i < READERS_COUNT; i++)
    reader_create(sem_descr, i + 1);

for (int i = 0; i < WRITERS_COUNT; i++)
    writer_create(sem_descr, i + 1);

for (int i = 0; i < READERS_COUNT + WRITERS_COUNT; i++)
    wait(&status);

// Функция shmdt() «отключает» разделяемый сегмент от адресного пространства
// процесса
if (shmdt(counter) == -1)
    perror("Ошибка при попытке отключить разделяемый сегмент от адресного
пространства процесса.");

if (semctl(sem_descr, 0, IPC_RMID, 0) == -1)
{
    perror("Ошибка при попытке удаления семафора.");
    return -1;
}

return 0;
}

```

## ЧИТАТЕЛИ - ПИСАТЕЛИ ВИНДА

Про дескрипторные типы немного рассказывалось на вводном уроке в WINAPI. Дескриптор, как говорилось ранее, — это идентификатор какого-либо объекта. Для разных типов объектов существуют разные дескрипторы. Дескриптор объекта можно описать так:

```

1      HANDLE h;

```

```
// Надо использовать неделимые операции:  
// InterLockedIncrement, InterLockedDecrement.  
// В программе должно быть 3 счетчика:  
// ждущих писателей, ждущих читателей и активных читателей.  
// Активный писатель м.б. только один и это логический тип.  
  
#include <windows.h>  
#include <stdbool.h>  
#include <stdio.h>  
#include <time.h>  
#include <stdbool.h>  
  
#define MINIMUM_READER_DELAY 100  
#define MINIMUM_WRITER_DELAY 100  
#define MAXIMUM_READER_DELAY 200  
#define MAXIMUM_WRITER_DELAY 400  
  
#define READERS_NUMBER 5  
#define WRITERS_NUMBER 3  
  
#define ITERATIONS_NUMBER 8  
  
HANDLE can_read;  
HANDLE can_write;  
HANDLE mutex;  
  
LONG waiting_writers_count = 0;  
LONG waiting_readers_count = 0;  
LONG active_readers_count = 0;  
  
bool is_writer_active = false;  
  
HANDLE readerThreads[READERS_NUMBER];  
HANDLE writerThreads[WRITERS_NUMBER];  
  
int readersID[READERS_NUMBER];  
int writersID[WRITERS_NUMBER];  
  
int readersRand[READERS_NUMBER * ITERATIONS_NUMBER];  
int writersRand[READERS_NUMBER * ITERATIONS_NUMBER];  
  
int value = 0;  
  
void StartRead()  
{  
    // ждущих читателей++  
    InterlockedIncrement(&waiting_readers_count);  
  
    if (is_writer_active || WaitForSingleObject(can_write, 0) == WAIT_OBJECT_0)  
    {  
        WaitForSingleObject(can_read, INFINITE);  
    }  
  
    WaitForSingleObject(mutex, INFINITE);
```

```

// ждущих читателей--
InterlockedDecrement(&waiting_readers_count);

// активных читателей++
InterlockedIncrement(&active_readers_count);

// Чтобы следующий читатель в очереди
// Читателей смог начать чтение
SetEvent(can_read);

// ReleaseMutex() освобождает ранее захваченный мьютекс.
ReleaseMutex(mutex);
}

void StopRead()
{
    // активных читателей--
    InterlockedDecrement(&active_readers_count);

    if (!active_readers_count)
    {
        ResetEvent(can_read);
        SetEvent(can_write);
    }
}

DWORD WINAPI Reader(CONST LPVOID param)
{
    int id = *(int *)param;
    int sleepTime;
    int begin = id * ITERATIONS_NUMBER;
    for (int i = 0; i < ITERATIONS_NUMBER; i++)
    {
        sleepTime = rand() % MAXIMUM_READER_DELAY + MINIMUM_READER_DELAY;
        Sleep(sleepTime);

        // CRITICAL START
        StartRead();
        printf("-> Reader id = %d; value = %d; sleep time = %d.\n", id, value,
sleepTime);
        StopRead();
        // CRITICAL END
    }
}

void StartWrite()
{
    // ждущих писателей++
    InterlockedIncrement(&waiting_writers_count);

    if (is_writer_active || active_readers_count > 0)
    {
        WaitForSingleObject(can_write, INFINITE);
    }

    // ждущих писателей--
    InterlockedDecrement(&waiting_writers_count);
}

```

```

        is_writer_active = true;
    }

void StopWrite()
{
    is_writer_active = false;

    if (waiting_readers_count)
    {
        SetEvent(can_read);
    }
    else
    {
        SetEvent(can_write);
    }
}

DWORD WINAPI Writer(CONST LPVOID param)
{
    int id = *(int *)param;
    int sleepTime;
    int begin = id * ITERATIONS_NUMBER;

    for (int i = 0; i < ITERATIONS_NUMBER; i++)
    {
        sleepTime = rand() % MAXIMUM_WRITER_DELAY + MINIMUM_WRITER_DELAY;
        Sleep(sleepTime);

        // CRITICAL END
        StartWrite();
        ++value;
        printf("<<< ----- Writer id = %d; value = %d; sleep time = %d.\n", id,
value, sleepTime);
        StopWrite();
        // CRITICAL END
    }
}

int init()
{
    // 2 == false значит мьютекс свободный.
    // 3 задает имя мьютекса, если нужно со
    // 4 здать именованный мьютекс. Если указывается NULL,
    // то мьютекс не именуется.
    mutex = CreateMutex(NULL, FALSE, NULL);
    if (mutex == NULL)
    {
        perror("CreateMutex\n");
        return -1;
    }

    // 2 == FALSE значит автоматический сброс.
    // 3 == FALSE значит, что объект не в сигнальном состоянии.
    // 4 == имя
    if ((can_write = CreateEvent(NULL, FALSE, FALSE, NULL)) ==
}

```

```

NULL)
{
    perror("CreateEvent (can_write)");
    return -1;
}

if ((can_read = CreateEvent(NULL, TRUE, FALSE, NULL)) ==
NULL)
{
    perror("CreateEvent (can_read)");
    return -1;
}

return 0;
}

int CreateThreads()
{
    //DWORD id = 0;
    for (short i = 0; i < WRITERS_NUMBER; i++)
    {
        writersID[i] = i;
        if ((writerThreads[i] = CreateThread(NULL, 0,
&Writer, writersID + i, 0, NULL)) == NULL)
        {
            perror("CreateThread (writer)");
            return -1;
        }
        // printf("Created writer thread id = %d\n", id);
    }

    for (short i = 0; i < READERS_NUMBER; i++)
    {
        readersID[i] = i;
        // Параметры слева направо:
        // NULL - Атрибуты защиты определены по умолчанию;
        // 0 - размер стека устанавливается по умолчанию;
        // Reader - определяет адрес функции потока, с
которой следует начать выполнение потока;
        // readersID + i - указатель на переменную, которая
передается в поток;
        // 0 - исполнение потока начинается немедленно;
        // Последний - адрес переменной типа DWORD, в которую
функция возвращает идентификатор потока.
        if ((readerThreads[i] = CreateThread(NULL, 0,
&Reader, readersID + i, 0, NULL)) == NULL)
        {
            perror("CreateThread (reader)");
            return -1;
        }
        // printf("Created reader thread id = %d\n", id);
    }

    return 0;
}

```

```

}

void Close()
{
    // Закрываем дескрипторы mutex, event и всех созданных
    // потоков.
    for (int i = 0; i < READERS_NUMBER; i++)
    {
        CloseHandle(readerThreads[i]);
    }

    for (int i = 0; i < WRITERS_NUMBER; i++)
    {
        CloseHandle(writerThreads[i]);
    }

    CloseHandle(can_read);
    CloseHandle(can_write);
    CloseHandle(mutex);
}

int main(void)
{
    setbuf(stdout, NULL);
    srand(time(NULL));

    int rc = init();
    if (rc)
    {
        return -1;
    }

    rc = CreateThreads();
    if (rc)
    {
        return -1;
    }

    // READERS_NUMBER - кол-во инитерсующих нас объектов ядра.
    // readerThreads - указатель на массив описателей объектов
    // ядра.
    // TRUE - функция не даст потоку возобновить свою работу,
    // пока не освободятся все объекты.
    // INFINITE - указывает, сколько времени поток готов ждать
    // освобождения объекта.
    WaitForMultipleObjects(WRITERS_NUMBER, writerThreads, TRUE,
    INFINITE);
    WaitForMultipleObjects(READERS_NUMBER, readerThreads, TRUE,
    INFINITE);
}

```

```
    Close();  
  
    printf("\nFINISH\n");  
    return 0;  
}
```

- [CreateThread](#)

Функция CreateThread()

Для создания дополнительных потоков, нужно вызывать из первичного потока функцию CreateThread():

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpsa;  
    DWORD cbStack;  
    LPTHREAD_START_ROUTINE lpStartAddr;  
    LPVOID lpvThreadParam;  
    DWORD fdwCreate;  
    LPDWORD lpIDThread);
```

- lpsa – указатель на структуру SECURITY\_ATTRIBUTES. Для установки атрибутов защиты, определенных по умолчанию, в параметр записывается NULL.
- cbStack – определяет, какую часть адресного пространства поток может использовать под стек. Каждый поток имеет отдельный стек. Если указывается 0, то размер стека устанавливается по умолчанию.
- lpStartAddr – определяет адрес функции потока, с которой следует начать выполнение потока.
- lpvThreadParm – указатель на переменную, которая передается в поток.
- fdwCreate – определяет флаги, управляющие созданием потока. Может иметь одно из двух значений: 0 – исполнение потока начинается немедленно, CREATE\_SUSPENDED – поток находится в состоянии ожидания.
- lpIDThread – адрес переменной типа DWORD, в которую функция возвращает идентификатор потока.

- [CreateMutex](#)

Процедура [CreateMutex\(\)](#) создает новый мьютекс. О первом аргументе сейчас знать не нужно. Если второй аргумент равен TRUE, созданный мьютекс будет сразу захвачен текущим потоком, если же второй аргумент равен FALSE, создается свободный мьютекс. Третий аргумент задает имя мьютекса, если нужно создать именованный мьютекс. Если указывается NULL, то мьютекс не именуется. Возвращаемые значения такие же, как у CreateThread.

- `ReleaseMutex`

`ReleaseMutex()` освобождает ранее захваченный мьютекс. В случае успеха процедура возвращает значение, отличное от нуля. В случае ошибки возвращается ноль, а подробности доступны через `GetLastError`. Важно, для предотвращения тупиков (deadlock) Windows позволяет потокам захватывать один и тот же мьютекс несколько раз, но и количество вызовов `ReleaseMutex()` должно быть равно количеству захватов.

- `CreateEvent`

```
function CreateEvent(
    lpEventAttributes: PSecurityAttributes; // Адрес структуры
                                            // TSecurityAttributes
    bManualReset,           // Задает, будет Event переключаемым
                            // вручную (TRUE) или автоматически (FALSE)
    bInitialState: BOOL;   // Задает начальное состояние. Если TRUE -
                            // объект в сигнальном состоянии
    lpName: PChar          // Имя или NIL, если имя не требуется
): THandle; stdcall;    // Возвращает идентификатор созданного
                        // объекта
```

- `SetEvent`

Состояние объекта события ручного сброса остается сигнализированным до тех пор, пока функция `ResetEvent` не установит его явно в состояние **без** сигнала. Любое количество ожидающих потоков или потоков, которые впоследствии начинают операции ожидания для указанного объекта события, вызывая одну из функций **ожидания**, могут быть освобождены, пока сигнализируется состояние объекта.

Состояние объекта события с автоматическим сбросом остается сигнализированным до тех пор, пока не будет освобожден один ожидающий поток, после чего система автоматически установит состояние без сигнала. Если никакие потоки не ждут, состояние объекта события остается сигнализированным.

- `InterlockedDecrement`

- `InterlockedIncrement`
- `WaitForSingleObject`

```
DWORD WaitForSingleObject(
```

```
    HANDLE hObject,
```

```
    DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, `hObject`, идентифицирует объект ядра, поддерживающий состояния «свободен-занят». (То есть любой объект, упомянутый в списке из предыдущего раздела.) Второй параметр, `dwMilliseconds`, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем `hProcess`:

```
WaitForSingleObject(hProcess, INFINITE);
```

В данном случае константа `INFINITE`, передаваемая во втором параметре, подсказывает системе, что вызывающий поток готов ждать этого события хоть целую вечность.

- `WaitForMultipleObjects`

```
DWORD WaitForMultipleObjects(
```

```
    DWORD dwCount,
```

```
    CONST HANDLE* phObjects,
```

```
    BOOL fWaitAll,
```

```
    DWORD dwMilliseconds);
```

Параметр `dwCount` определяет количество интересующих Вас объектов ядра. Его значение должно быть в пределах от 1 до `MAXIMUM_WAIT_OBJECTS` (в заголовочных файлах Windows оно определено как 64). Параметр `phObjects` — это указатель на массив описателей объектов ядра.

`WaitForMultipleObjects` приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр `fWaitAll` как раз и определяет, чего именно Вы хотите от функции. Если он равен `TRUE`, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

Event позволяет известить один или несколько ожидающих потоков о наступлении события. События используются потоками для сигнализации, что другой поток может выполнить какую-то работу.

Event бывает:

Со сбросом вручную	Будучи установленным в сигнальное состояние, остается в нем до тех пор, пока не будет переключен явным вызовом функции ResetEvent
С автосбросом	Автоматически переключается в несигнальное состояние операционной системой, когда один из ожидающих его потоков завершается

## На черный день советы от старосты четвертой группы

Главный вопрос: Опишите переход в реальный режим

1. Устанавливаем флаг перех. в реальном режиме
2. Запрещаем маскируемые прерывания
3. Переходим в реальный
4. Через команду far jmp, заданную прямо кодом, обновляем значение в теневом регистре, связанном с CS
5. Обновляем остальные теневые регистры значениями
6. Возвращаем маски контроллерам прерываний, возвращаем значение базового вектора прерывания (8, не 32)
7. Возвращаем базовый линейный адрес таблице векторов прерываний
8. Разрешаем немаскируемые
9. Разрешаем маскируемые
10. Печатаем сообщение с помощью функций Dos
11. Выходим через функцию DOS (ред.) Через что взаимодействуем с клавиатурой? — через порты (ред.) Что за память на экране? — доступная программе dosbox память
  - Почему делим? Потому что в мегабайтах (ред.)

- Зачем сегмент стека в защищенном? — для использования прерываний (ред.)
- Что происходит при загрузке селектора в сегментный регистр? — теневой регистр, связанный с сегментным регистром, обновляются значением линейного адреса сегмента

- Почему для Клавы и мыши идёт увеличение приоритета на 6
- 
- Для аудио / видео на 8

Спойлер: потому что интерактивное устройство ввода

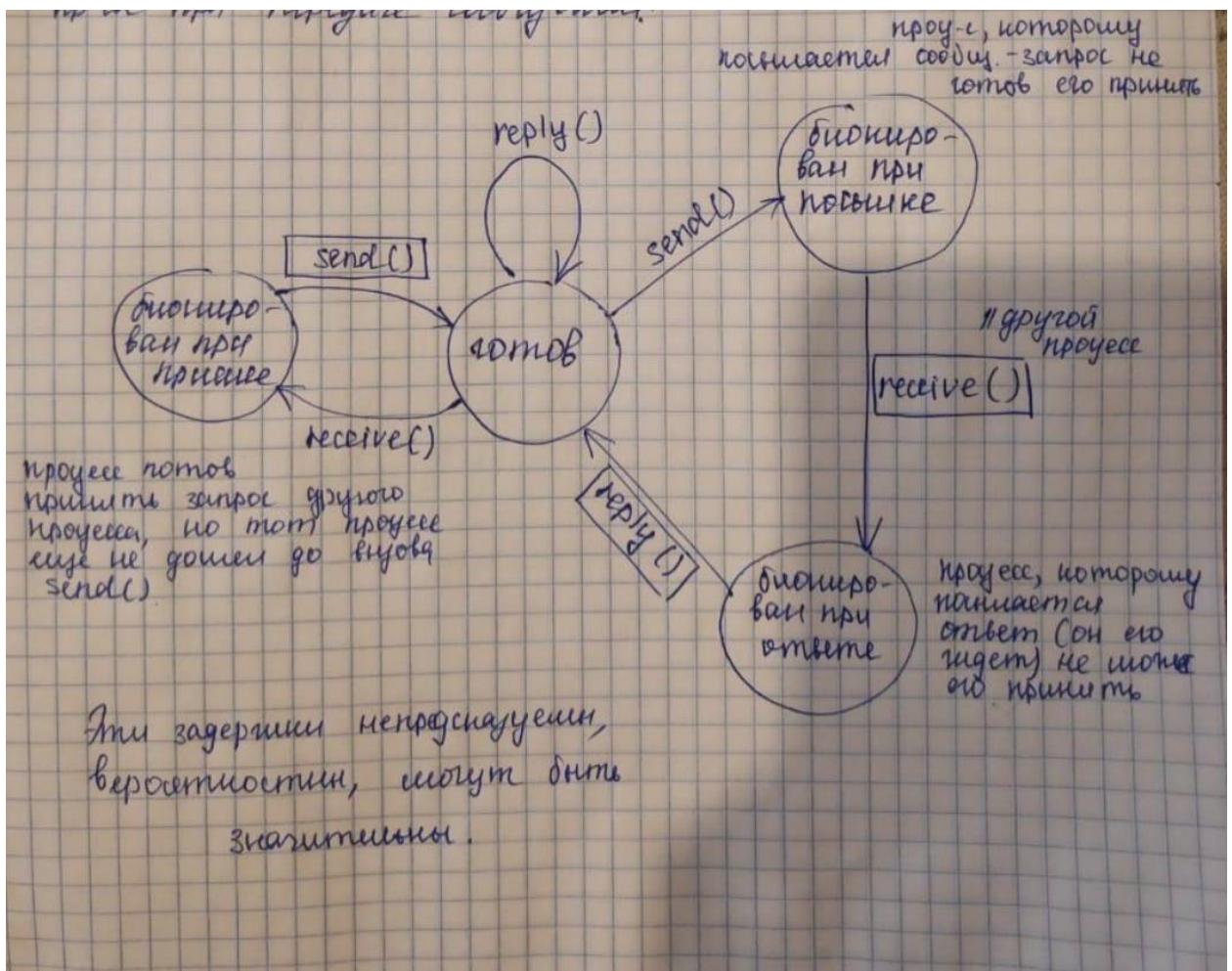
что за регистрация.....ниже

перевести соответствующие процессы из состояния sleep в состояние running

Регистрация отложенного вызова функции означает отправку сигнала SIGCONT, обработчик которого изменяет состояние процесса с S (Sleep) на R (Running)

В POSIX-системах, **SIGCONT** — сигнал, посыпаемый для возобновления выполнения процесса, ранее остановленного сигналом SIGSTOP или другим сигналом (SIGTSTP, SIGTTIN, SIGTTOU).

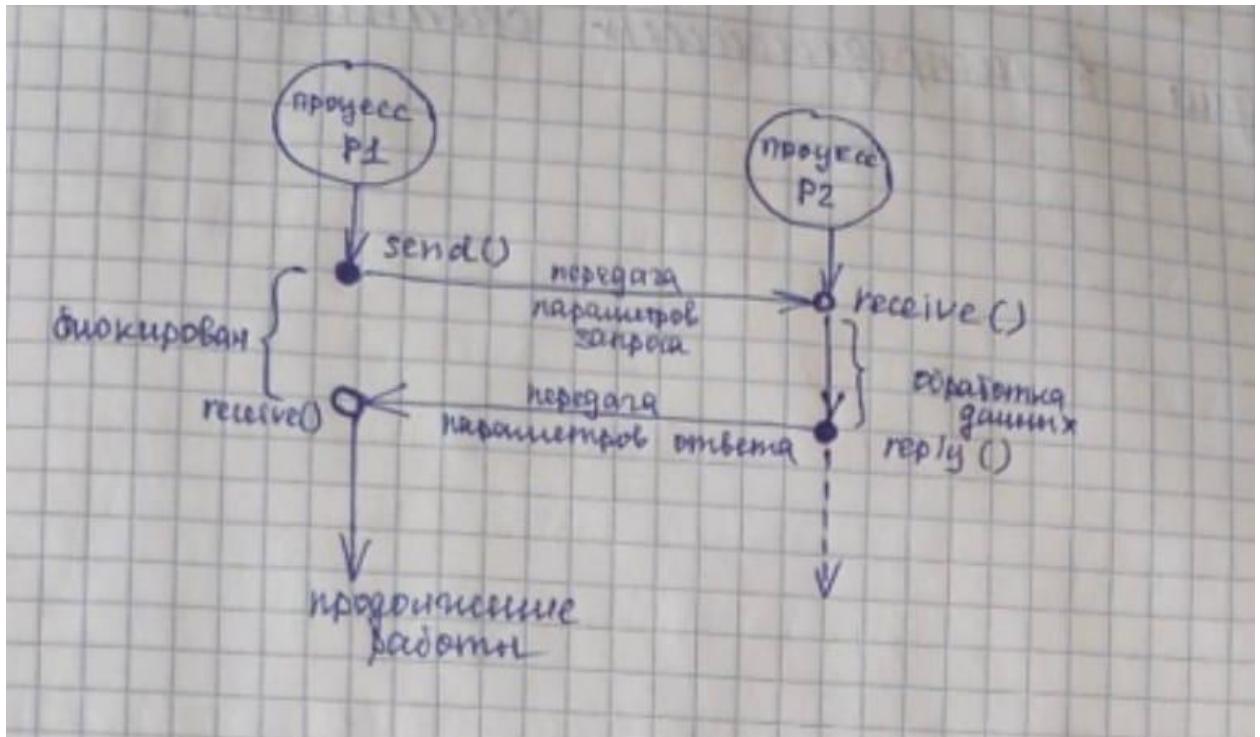
ТРИ БЛОКИРОВКИ ПРИ СООБЩЕНИЯХ



Если один процесс заинтересован в работе другого процесса, то они должны быть синхронизированы, то есть тут возможны три типа блокировок:

1. блокирован при посылке в том случае, если процесс, которому посыпается сообщение-запрос не готов его принять;
2. блокирован при ответе, если процесс сформировал ответ, но процесс который ждет этот ответ может быть не готов к приему;
3. блокирован при приеме - процесс готов принять запрос другого процесса на обслуживание, но тот процесс еще не дошел до точки где вызывает функцию `send()`.

Эти задержки выполнения процессов непредсказуемы, но они могут быть очень значительными.



Особенность взаимодействия процессов в системах с раздельной памятью состоит в том, что действуют абсолютно самостоятельные процессы которые могут выполняться на разных машинах - это процессы, которые отправляют сообщения и процессы, которые сообщения получают. Например, модель клиент-сервер. Сервер предоставляет какие-то услуги - сервисы, клиенты запрашивают эти услуги, обмен происходит с помощью сообщений. Особенности такого взаимодействия:

В какой-то момент времени, выполняя свой код, процесс p1 посылает сообщение-запрос процессу p2. (Обычно самыми общими названиями таких системных вызовов являются `send` и `receive`. Но для того, чтобы отдельно отметить, что посыпается ответ на запрос, применяется ключевое слово `reply`.) В результате системный вызов `send()` посылает какие-то параметры. Процесс p2 выполняется с собственно скоростью - это асинхронный процессы, и чтобы принять такое сообщение-запрос, процесс должен вызвать системный вызов `receive()`. Если он не готов вызвать `receive()`, то процесс p1 будет блокирован до того момента, пока другой процесс не сможет принять его сообщение. Получив сообщение-запрос, процесс p2 обрабатывает какое-то время пришедший запрос с пришедшими данными и, когда он готов, он вызывает `reply`.

## вопросы

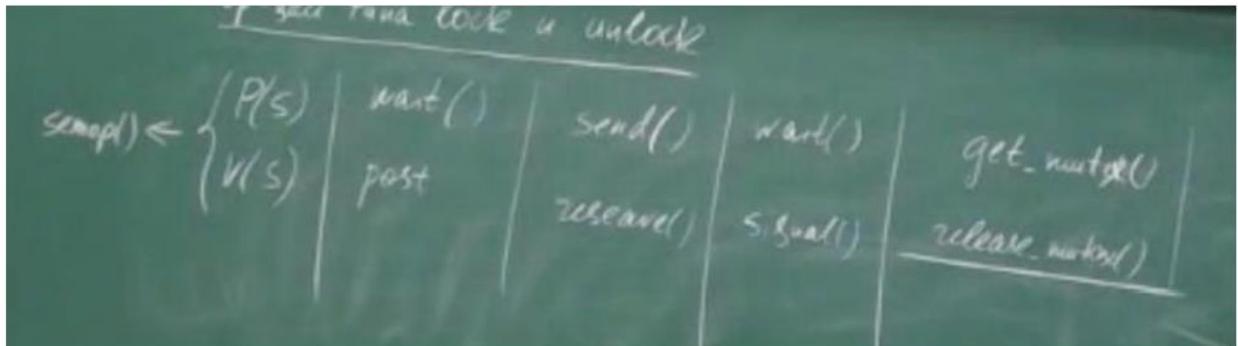
программа может реагировать на сигналы - использовать свой обработчик, стандартный обработчик или проигнорировать сигнал

для чего нужны сигналы - влиять на ход выполнения программы

- она просила сказать, какие прерывания переводят из какого в какое состояние
- 
- Например системный вызов переводит из выполнения в блокировку

## Операции над: Семафоры, мьютексы, сообщения, мониторы

Все операции можно поделить на “условные lock и unlock”



Get\_mutex, release\_mutex – гипотетические!

### Разница между взаимоисключением и синхронизацией:

Мы всегда говорим об асинхронных процессах – собственные скорости, невозможно предсказать, когда процесс придет в точку. Обычно не делается различие между взаимоисключением (B) и синхронизацией (C). Но это 2 разных понятия.

B – про монопольный доступ. C – один процесс не может начать свою работу, если другой не сделал в интересах первого процесса каких-то действий. Будет ждать (сообщения, например).

Это различие можно увидеть в производство-потребление и читатели-писатели.

- Задача производство-потребление связана с синхронизацией. Потребитель не сможет работать, если в буфере нет данных, если буфер пуст. Будет ждать, пока производитель произведет единицу данных. Аналогично производитель не может начать работу, если все ячейки буфера заполнены – будет ждать, когда потребитель освободит хотя бы одну ячейку. Поэтому по 2 переменные (полон пуст) или (2 переменные типа условие)
- Читатель-писатель не связаны. Читатель только ждет, пока писатель освободит критическую секцию. Так же и писатель может начать работать, если в конкретный момент времени никакой читатель или писатель не захватили, неважно что там было до.

## Проблема спящего парикмахера

Процессы-клиенты, ресурс-кресло.

Это классическая задача межпроцессорного взаимодействия в многозадачной ОС. Процессы асинхронные, выполняются с разными скоростями. Невозможно предсказать.

Аналогия – гипотетическая пар с одним паром. У него есть 1 кресло и приемна с несколькими стульями. Когда он отпускает одного клиента, то идет в приемную, чтобы посмотреть, есть ли ждущие клиенты. Если есть – садит одного в кресло и обслуживает. Если нет, то возвращается, садится в кресло и спит.

Каждый новый приходящий клиент смотрит, что делает парикмахер. Если спит, то будит и садится в кресло. Если работает, то идет в приемную. Если есть стул-садится, если нет – уходит.

Возможны несколько конфликтных ситуаций, общие проблемы планирования. Все они связаны с тем, что действия парикмахера и клиента асинхронные, независимые друг от друга, то есть время, которое тратится на обслуживание и проверки разное и непредсказуемое

Например, пока клиент уже идет в приемную, парикмахер может тоже начать идти в приемную, но придет раньше. Клиента нет.

Или 2 клиента пришли одновременно. А стул только 1. Оба в приемную и лезут на один стул

## Рандеву

Рассмотрим «Рандеву» как модель организации взаимодействия процессов. Модель Рандеву требует: в том случае, если процесс P1 желает передать данные процессу P2, то процесс P2 должен выразить свою готовность к установлению Рандеву (свидание/встреча) Модель Рандеву предложена Хоаром, при этом особенностью Рандеву является то, что если Рандеву принимается обеими сторонами (то есть они вступают в такое взаимодействие), то от имени обоих процессов выполняется одна и та же последовательность команд и взаимодействующие процессы получают результат выполнения этих команд одновременно. Таким образом исключается состояние ожидания при ответе. Этот способ исключает блокировки (не все конечно). Таким образом, взаимодействие выполняется быстрее и эффективнее Это очень важно, особенно для систем реального времени

О языке ADA. В языке ADA работа с параллельными процессами осуществляется при помощи средств, которые называются task. Таски - специальный вид модуля, каждый из которых может работать независимо, но таски имеют средство связи друг с другом. Таск имеет спецификацию и тело. При этом задачи связываются друг с другом при помощи входов. Если одна задача выдала обращение ко входу и это обращение принимается другой задачей, то обе задачи теряют свою независимость и устанавливают Рандеву. Пока Рандеву действует задачи синхронизированы. В механизме Рандеву отсутствует симметрия, так как задача, к которой выполнено обращение, может принять вызов, а может его не принять. Обслуживающая задача, то есть к которой направлено обращение, имеет точки входа - entry. Это можно представить себе следующим образом.

```
task <имя> is
    entry <идентификатор входа> (<дискретный диапазон>
                                         <формальная часть>);
    <другие объявления входов или фразы представления>
end <имя>;
```

.Если в спецификации задачи имеется объявление входа, то тело задачи должно содержать по крайней мере один оператор приема accept. В операторе приема accept указываются фактические условия, при которых наступает Рандеву. Формально accept записывает следующим образом:

```
accept:
    accept <идентификатор входа> (<выражение>) <формальная часть>
        do <последовательность операторов end;
```

Рандеву начинается с согласования фактических и формальных параметров, затем оно продолжается выполнением операторов, расположенных между do и end, и когда мы доходим до точки с запятой, Рандеву заканчивается. Во время Рандеву задачи могут обмениваться информацией через список параметров и обе задачи синхронизированы, а именно последовательность операторов выполняется от имени обеих задач.

Кроме указанных операторов есть оператор select, при помощи которого может осуществляться выбор.

Рандеву - ЭТО ОСОБЫЙ ВИД ВЗАИМОДЕЙСТВИЯ, ПРИ КОТОРОМ ПОСЛЕДОВАТЕЛЬНОСТЬ ОПЕРАТОРОВ ВЫПОЛНЯЕТСЯ ОТ ИМЕНИ НЕСКОЛЬКИХ ПРОЦЕССОВ. ЭТО ЗНАЧИТ, ЧТО ОНИ ОДНОВРЕМЕННО ПОЛУЧАЮТ РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ ДАННОЙ ПОСЛЕДОВАТЕЛЬНОСТИ ОПЕРАТОРОВ, ЧТО ИСКЛЮЧАЕТ ДОПОЛНИТЕЛЬНОЙ БЛОКИРОВКИ.

### Короткое обобщение методов борьбы с тупиками:

1. Недопущение (исключение самой возможности)
  - a. Опережающее требование (стратегия Харвендера)
  - b. Упорядочивание ресурсов (в основном для реального времени)
  - c. не можешь получить нужный ресурс - освободи занятые
2. Обход (предотвращение)
  - a. Алгоритм банкира (предложен Дейкстрой, теоретический)
  - b. Алгоритм Хаббермана (аппроксимация предыдущего, практический)
3. Обнаружение и восстановление
  - a. Обнаружение - по графовой модели Холда (двудольный направленный граф). Методом редукции графа
  - b. Восстановление
    - i. Перезапуск системы
    - ii. Завершение попавших в тупик
    - iii. Завершение остальных (не в тупике)

### Логические часы Лампорта

Логическими часами (logical clock) назовем такую функцию  $\varphi$ , которая отображает множество событий распределенного вычисления  $C$ , на некоторое упорядоченное множество  $(T, <)$ , где  $T$  – совокупность допустимых значений логического времени (time domain “интервал времени”).

Система логических часов (logical clock) состоит из временного интервала (time domain) и логических часов  $C$ . Элементы  $T$  образуют частично упорядоченное множество над отношением  $<$  (меньше).

Отношение  $<$  называется “меньше”, “предыдущий” или “причина-следствие”.

Причинно следственное отношение (casual precedence)

Логические часы  $C$  – это функция, которая отображает некоторое событие  $e$  в распределенной системе на элемент в интервале времени  $T$ , обозначается, как  $C(e)$  и называется временная метка  $e$  (time stamp). Это может быть обозначено следующим образом:

$$C : H \rightarrow T$$

так что для двух событий  $e_i$  и  $e_j$  выполняется следующее свойство:

$$e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$$

Это свойство монотонности называется *clock consistency condition* – условием согласованности часов.

Если  $T$  и  $C$  удовлетворяют следующему условию для двух событий  $e_i$  и  $e_j$

$$e_i \rightarrow e_j \text{ тогда и т т } C(e_i) < C(e_j)$$

Система логических часов называется сильной последовательностью (согласованной последовательностью) (*consistent* - согласован)

Тик - период времени между двумя последующими прерываниями таймера.

Основной тик - период времени равный n тикам таймера (число n зависит от конкретного варианта системы).

Квант времени (quantum, time slice) — временной интервал, в течение которого процесс может использовать процессор до вытеснения другим процессом.