

Лекция 3

ОС

ОС: 2 взгляда

- Outside (простого пользователя)-интерфейс (для современного – графический интерфейс)
- Inside

Из Оксфордского словаря по вычислительной технике:

ОС-комплект программ, которые совместно управляют ресурсами, вычислительными системами и процессами, используют эти ресурсы при вычислениях

Основная задача ОС – управлять процессами (программами в стадии выполнения) и выделять им ресурсы

Ресурс системы – любая компонента вычислительной системы и предоставляемые ею возможности – как аппаратные, так и программные

Основные ресурсы:

- Процессорное время
- Объем оперативной (физической) памяти

Внешние устройства, ключ защиты, реентерабельные коды (код чистой процедуры-процедуры, не модифицирующей саму себя, то есть данные (переменные). Из реентерабельных кодов вынесены данные.

Данные в ОС находятся в специальных системных «таблицах» (условно). Таблица-массив структур-дополнительные накладные расходы, поэтому в ОС используют связные списки: включение, удаление элементов связано только с изменением указателей (адреса)

Дисциплины

1. Управление процессорами
2. Управление памятью (имеется в виду оперативная)
3. Взаимодействие параллельных процессов
4. Управление устройствами и файловой подсистемой-в следующем семестре
5. Классификация ядер ОС

Управление процессорами

Речь о процессах и особенностях их выполнения. Почему пишем именно процессорами? Потому что рассматриваем процессорное время и то, как оно выделяется процессам (очередь процессов)

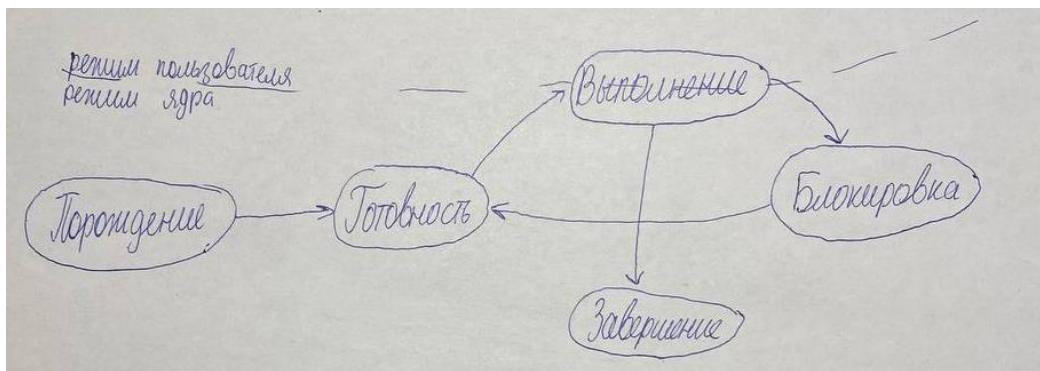
В любой ОС основной абстракцией является процесс.

С точки зрения UNIX процесс часть времени выполняется в режиме ядра (выполняется реентерабельный код ОС, а часть – в режиме пользователя (режим задачи) (выполняется собственный код). Таким образом, процесс постоянно переключается между режимами. При этом в ОС принято выделять диаграмму состояния процессов-этапы жизни процесса.

Обобщенная диаграмма

(только состояния, которые есть в любой ОС)

В общем-то, разработчики сами для себя выбирают дополнительные состояния.



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

Первый шаг системы при запуске программы – присвоение процессу идентификатора. Но надо управлять процессом (выделять ресурсы), следовательно, должна существовать структура, которая описывает процесс. Большая часть полей – указатели на другие структуры.

UNIX:

Выделяется строка в таблице процессов: а)номер=идентификатор (целое число) б)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.

3 типа событий, переводящих систему в режим ядра:

1. Системный вызов (часто называются программными прерываниями) software interrupts (trups). 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 СВЮ 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 и неисправимые. Являются синхронными событиями по отношению к коду
Неисправимые – ошибки (/0) ошибка деления, ошибки адресации
Исправимые-страничное прерывание, в результате которого отсутствующая страница будет загружена в память
3. Аппаратные прерывания – 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 иванта?

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)

[Планирование и диспетчеризация](#)

Выделение дефицитного ресурса – процессорного времени.

В многозадачных системах в состоянии готовности может быть много процессов. Необходимо организовать очередь, планирование – постановка процессов в очередь по выбранному дисц. Плану. В хорошо стр. планировщик scheduler – модуль ядра, выполняющий указанную работу.

Диспетчеризация – непосредственное выделение какого-либо ресурса.

Выделение процессу процессорного времени

Если очередь – из головы.

Дисц. Планир. Связаны с соответствующими типами ОС (к однозадачным 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.

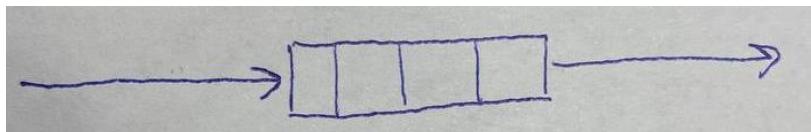
Рисунок (человек {пакет} квадрат). В системах разделения времени – наоборот: пользователь непосредственно связан – запускает, вводит данные, получает результат и реагирует – интерактив.

Классификация алгоритмов планирования

1. Без переключения
2. С переключением
3. С приоритетами
4. Без приоритетов
5. Без вытеснения (вытеснения без приоритетов не бывает)
6. С вытеснением

В системах пакетной обработки:

1. Алгоритм fifo (или fcfs – first come first server) – без переключения, без вытеснения.
Процесс, получив процессорное время, выполняется от начал и до конца



2. SJB - shortest job first (наискорейший – первым) – без вытеснения, без переключения, с приоритетом
По заявленному времени выполнения. Априорные – до опыта. Система отдавала предпочтение коротким процессам, что позволяло улучшить такой дурацкий показатель, как количество процессов в единицу времени (мера производительности)
Негативное явление – бесконечное откладывание
3. SRT – shortest remain time (наименьшее оставшееся время) – с приоритетом, с вытеснением
Выполняющийся процесс может быть прерван, если в очередь поступил процесс с меньшим процессорным временем выполнения (заявленное время – полученное=оставшееся и оно сравнивается)
Еще хуже по бесконечному откладыванию
4. HRN – highest response ratio (наибольшее относительное время)
Приоритеты пересчитываются по функции $r=(tw+ts)/ts$, 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.

1. RR – round robin

Лекция 6

Распределение памяти в разброс

Адресное пространство программы никак не зависит от ее расположения в памяти и было введено логическое адресное пространство. Его суть выражается в предложении «любая программа считает, что она начинается с нулевого адреса».

Программа может начинаться с любого физического адреса, но для доступа необходимо выполнять преобразование: физический адрес = начальный адрес раздела +смещение. И это преобразование поддерживается аппаратно

Это потребовало изменения всей системы программ.

Системно ПО делится на части

1. Операционные системы
2. Системы программирования (трансляторы, редакторы связи, загрузчики)

Появилась возможность перемещать программы в память. Когда это делать?

Существует 2 решения

1. Когда освобождается раздел
2. Когда возникает проблема загрузки – не находится раздел нужного размера

Все это относится к непрерывному выделению памяти: программа находится в памяти в непрерывной последовательности адресов.

Стремление использовать ОП (оперативную память) более эффективно привело к появлению выделения памяти вразброс. Подходы:

1. Поделить память на участки равного размера и физическую память на участки того же размера. Тогда возможно загружать программу в свободные участки физической памяти. Причем участки не обязаны располагаться один за другим – просто свободные участки в адресном пространстве физической памяти (вразброс)

Какие затраты при этом возникают?

Необходимо создать соответствующие таблицы с указанием в какие frame (страницы физической памяти) загружены страницы программы (page..).

Преобразование: если вернуться к непрерывному – там 1 начальный адрес – раздела, а здесь – каждая физическая страница с начальным адресом и нужно их менять.

2. Сегментация. (все эти решения для машин 3 поколения)

Программы стали большими, появилось логическое адресное пространство, объем памяти увеличивается.

Сегментация – логическое деление. Программа из модулей, каждый модуль – отдельный сегмент. Необходимо иметь таблицу соответствия, сегмент программы сопоставляется с выделенными в физической памяти сегментами. Кроме начального адреса необходимо указывать размер, так как сегментация – логическая.

ВИРТУАЛЬНАЯ ПАМЯТЬ

Идея виртуальной памяти

А надо ли программу целиком загружать? Все страницы/все сегменты надо загружать в память? – НЕТ. Но в памяти должны находиться те участки кода, к которым в данный момент обращается процессор. Как этого достичь? Загружать по необходимости, а необходимость возникает при обращении

В момент, когда процессор обращается к отсутствующим данным, в системе должно возникать что-то, что приведёт к загрузке страницы/сегмента в память

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!!!

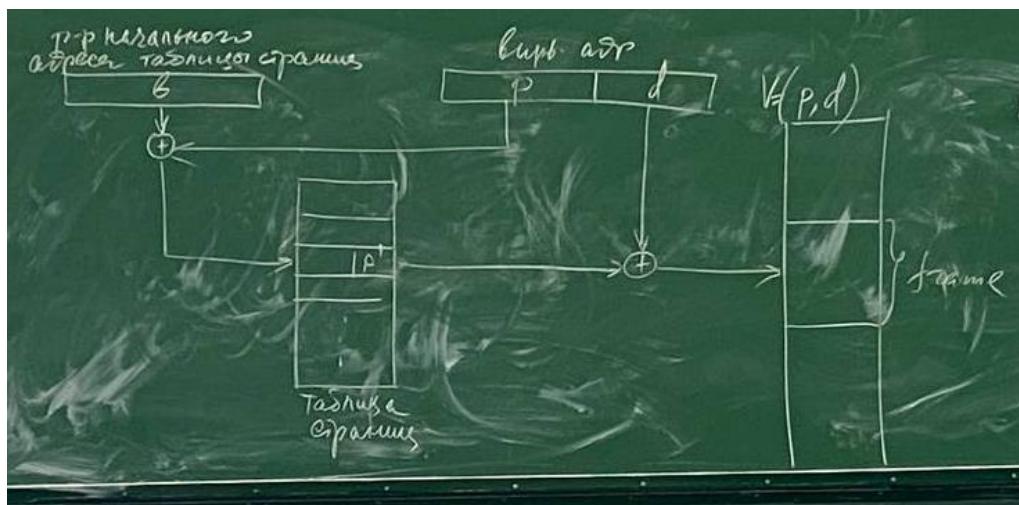
В момент порождения процесса система должна ему выделить минимально необходимый объем памяти, а в результате таких запросов загружаются необходимые участки кода.

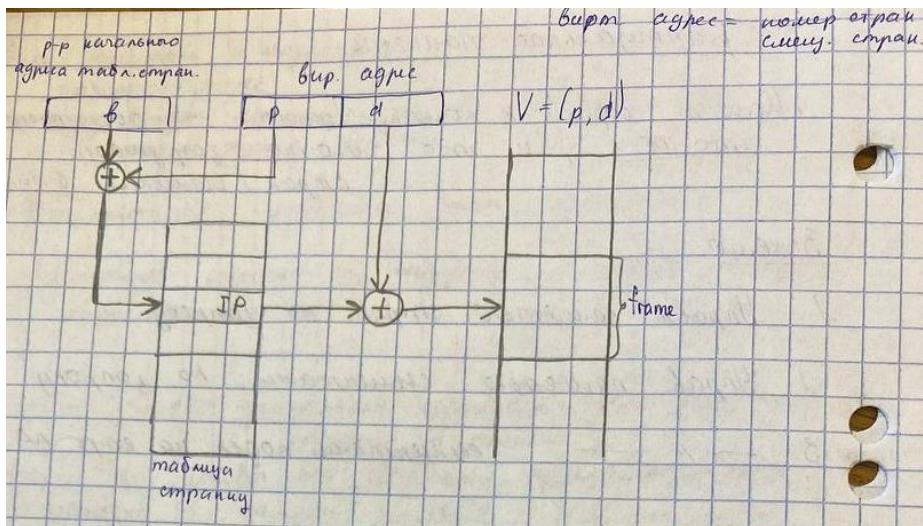
Виртуальная (кажущаяся возможной) память – память, объем которой превышает объем доступного физического адресного пространства (про диск – это понятно).

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

3 метода преобразования адресов (Теоретические основы преобразования, как в intel – посмотрим потом)

1. Прямое отображение





Виртуальный адрес делится на 2 части – номер страницы и смещение. $V=(p,d)$

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. То есть при переходе он меняется. Таблица страниц одна (и собственная) на каждый процесс. При этом это системные таблицы.

Если страница загружена в физическую память, то у этой страницы будет адрес физической памяти. 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 загружена, выполняется соответствующее преобразование, аппаратно поддерживаемое.

Таблица страниц находится в области памяти ядра системы (ОП). 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 второму

2. Ассоциативное отображение

Оно предполагает наличие в системе специального блока ассоциативной памяти (АП). АП всегда регистровая. Существуют 2 вида АП

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

АП позволяет получить доступ к информации по «ключу» за один такт. В результате схема имеет следующий вид.

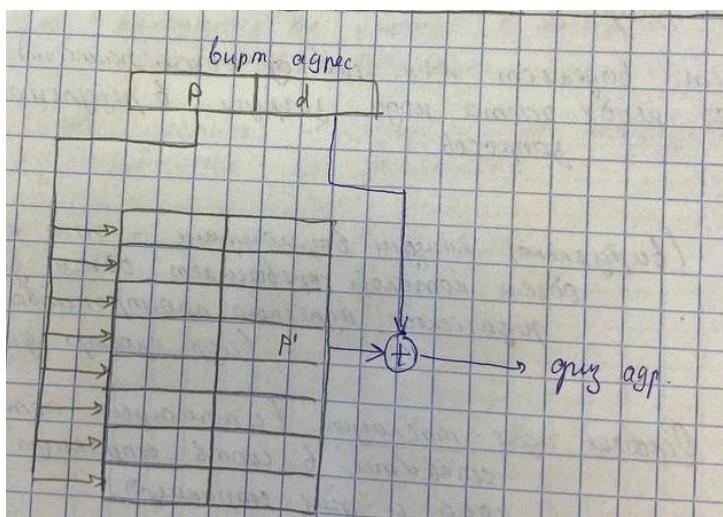
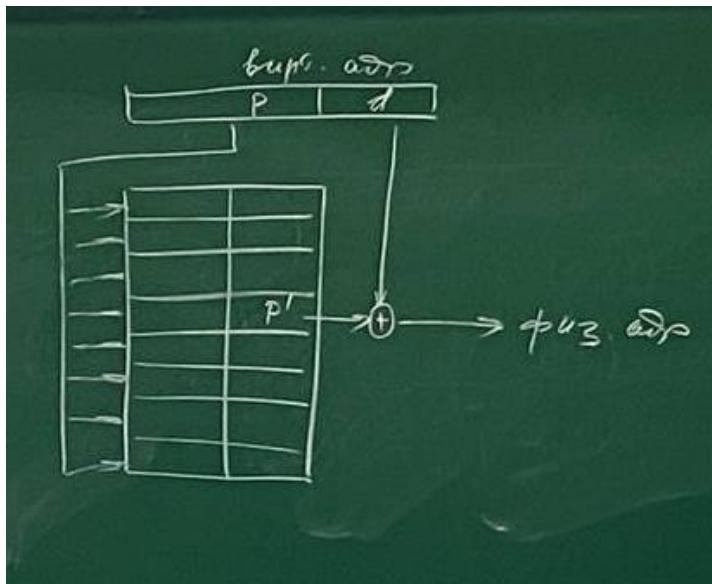
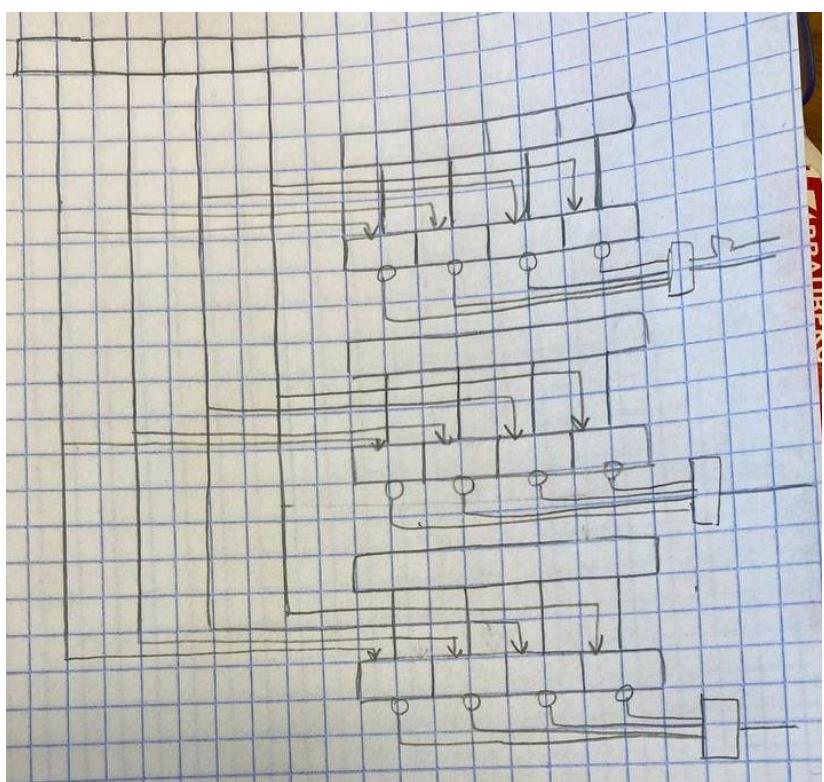
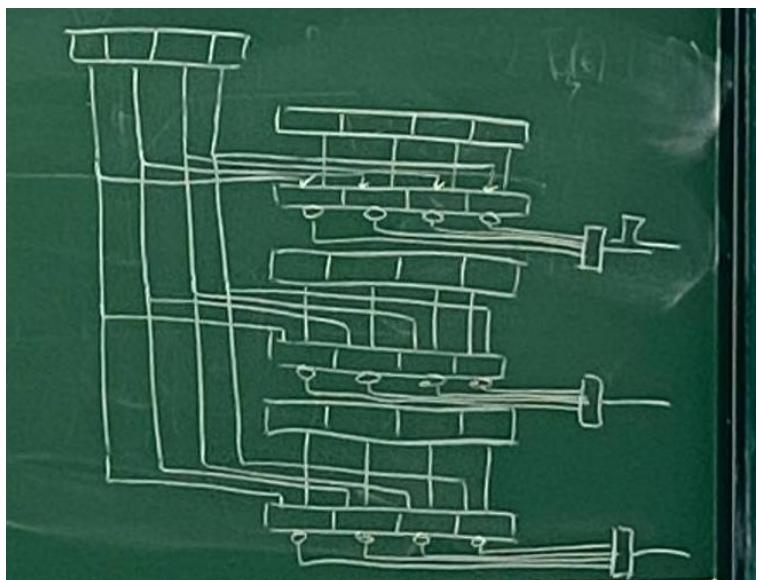


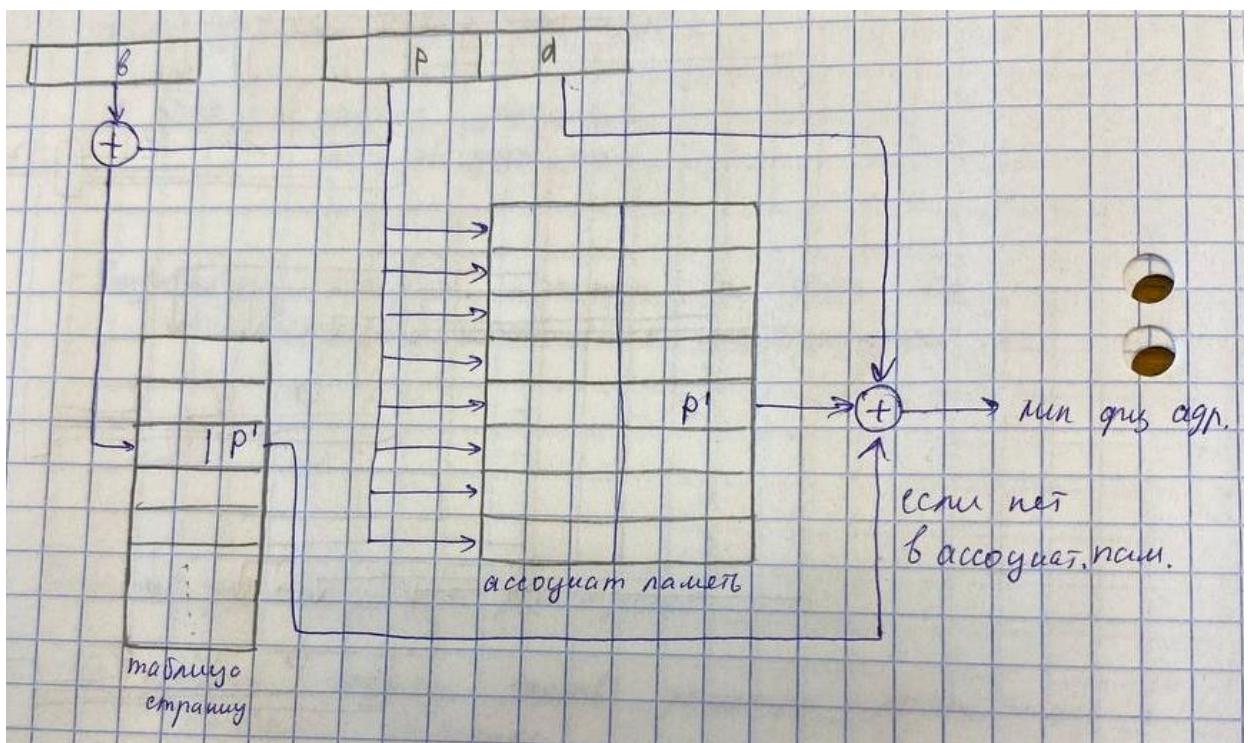
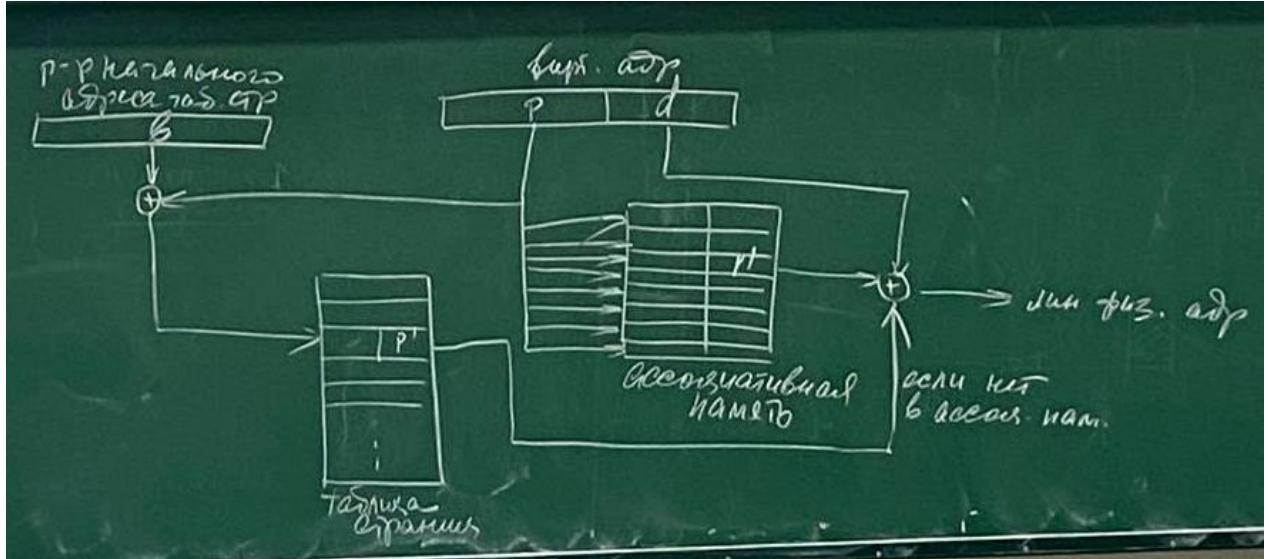
Таблица страниц загружается в специальную ассоциативную память. Номер страницы используется как ключ, сравниваются сразу все элементы всех дескрипторов и при совпадении выбирается физический адрес.

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. Сразу все разряды всех строк сравниваются с ключом, при совпадении всего – выборка информации (считывание). Дорого. Для больших таблиц страниц такое решение непригодно. Поэтому существует 3 схема

3. Ассоциативно-прямое отображение



В системе имеется небольшая по объему ассоциативная память (сейчас называется кэш). Сейчас – на 8/16 регистров, позволяла обеспечить 90% скоростных показателей полностью ассоциативной памяти

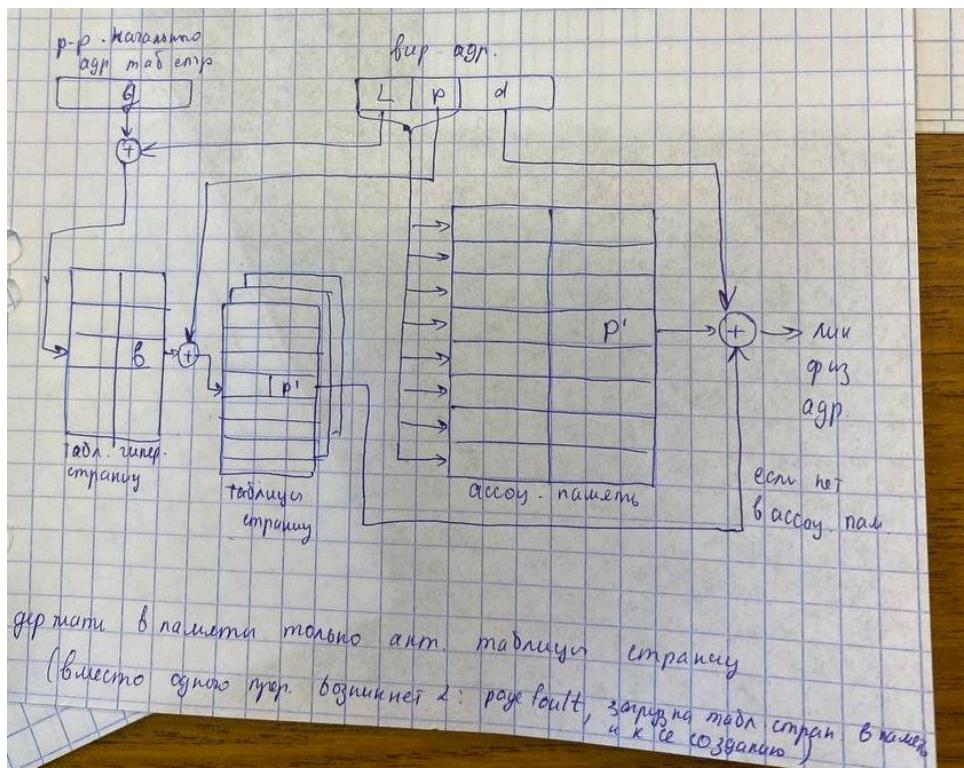
Страница сначала ищется в кэше. Если не найдена – обращение к физической памяти – таблице страниц. Если страница и там не найдена, то происходит страничное прерывание. Станичное прерывание в наших системах относится к исключениям – page fault. Это исключение, но исправимое.

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

Двухуровневая страничная организация

Была реализована в IBM360/67 и IBM370. 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.



Эти схемы вроде из дейтала (книга видимо)

У процесса 1 таблица гипер-страниц (1 на каждый процесс) и много таблиц страниц. Добавляется одно обращение к оперативной памяти. То есть время больше. Но здесь можно хранить в памяти только актуальные таблицы страниц.

Страница загружается в память только когда происходит обращение. Но вместо 1 прерывания возникнет 2: 1) page fault 2) нет таблицы страниц (загрузка таблицы страниц в память и, вероятно, ее создание, и потом загрузка страницы в память). Затраты очевидны.

Страницное преобразование пожирает процессорное время. Почему не отказались от идеи виртуальной памяти? В чем выигрыш — в увеличении мультизадачности. Можно хранить в памяти большее количество программ.

Здесь большую роль играет кеширование. 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 (опыта) 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

Страницы – единицы физического деления памяти – размер страницы установлен в системе

Вопрос – зачем тогда нужны остальные схемы (сегментами поделенными на страницы и т.д.)?

Коллективное использование в задачах данных... и что-то еще непонятное

Заинтересованность большого числа процессов в одних и тех же программах и кодах ОС. Если каждому процессу предоставлять индивидуальную копию – крайне неэффективное использование памяти. В результате чтобы коллективное использование было эффективным, программы, которые коллективно используются должны быть реентерабельными – допускать повторную входимость. Это основная проблема, которая была при управлении памятью страницами по запросу.

В результате такой потребности (особенно касается данных) (один хочет поменять что-то на странице, а другой – взять). Тогда страницу надо обозначать отдельно для каждого процесса и выдавать права доступа. На уровне страниц теряется принадлежность конкретной программе или набору данных.

Алгоритмы замещения страниц

Все физические страницы выделены. Процесс обращается к отсутствующей –страничное прерывание – загрузить страницу, предварительно выгрузив другую страницу. Page replacement.
Алгоритмы:

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 дейтоля): 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

Вот тут доступно 3, Итого 9/12 – 75% неудач

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

$9/12 = 75\%$

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
\uparrow	4	3	2	1	1	5	4	4	3	2	1	5
$M=4$		4	3	2	2	1	5	4	3	2	1	
\downarrow		4	3	3	3	2	1	5	4	3	2	
	+	+	+	+	+	+	+	+	+	+	+	+

$10/12 \approx 83\%$

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

Распределение памяти в разброс

Адресное пространство программы никак не зависит от ее расположения в памяти и было введено логическое адресное пространство. Его суть выражается в предложении «любая программа считает, что она начинается с нулевого адреса».

Программа может начинаться с любого физического адреса, но для доступа необходимо выполнять преобразование: физический адрес = начальный адрес раздела + смещение. И это преобразование поддерживается аппаратно

Это потребовало изменения всей системы программ.

Системно ПО делится на части

3. Операционные системы
4. Системы программирования (трансляторы, редакторы связи, загрузчики)

Появилась возможность перемещать программы в памяти. Когда это делать?

Существует 2 решения

3. Когда освобождается раздел
4. Когда возникает проблема загрузки – не находится раздел нужного размера

Все это относится к непрерывному выделению памяти: программа находится в памяти в непрерывной последовательности адресов.

Стремление использовать ОП (оперативную память) более эффективно привело к появлению выделения памяти вразброс. Подходы:

3. Поделить память на участки равного размера и физическую память на участки того же размера. Тогда возможно загружать программу в свободные участки физической памяти. Причем участки не обязаны располагаться один за другим – просто свободные участки в адресном пространстве физической памяти (вразброс)

Какие затраты при этом возникают?

Необходимо создать соответствующие таблицы с указанием в какие frame (страницы физической памяти) загружены страницы программы (page..).

Преобразование: если вернуться к непрерывному – там 1 начальный адрес – раздела, а здесь – каждая физическая страница с начальным адресом и нужно их менять.

4. Сегментация. (все эти решения для машин 3 поколения)

Программы стали большими, появилось логическое адресное пространство, объем памяти увеличивается.

Сегментация – логическое деление. Программа из модулей, каждый модуль – отдельный сегмент. Необходимо иметь таблицу соответствия, сегмент программы сопоставляется с выделенными в физической памяти сегментами. Кроме начального адреса необходимо указывать размер, так как сегментация – логическая.

ВИРТУАЛЬНАЯ ПАМЯТЬ

Идея виртуальной памяти

А надо ли программу целиком загружать? Все страницы/все сегменты надо загружать в память? – НЕТ. Но в памяти должны находиться те участки кода, к которым в данный момент обращается процессор. Как этого достичь? Загружать по необходимости, а необходимость возникает при обращении

В момент, когда процессор обращается к отсутствующим данным, в системе должно возникнуть что-то, что приведёт к загрузке страницы/сегмента в память

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.

ПО ЗАПРОСУ!!!

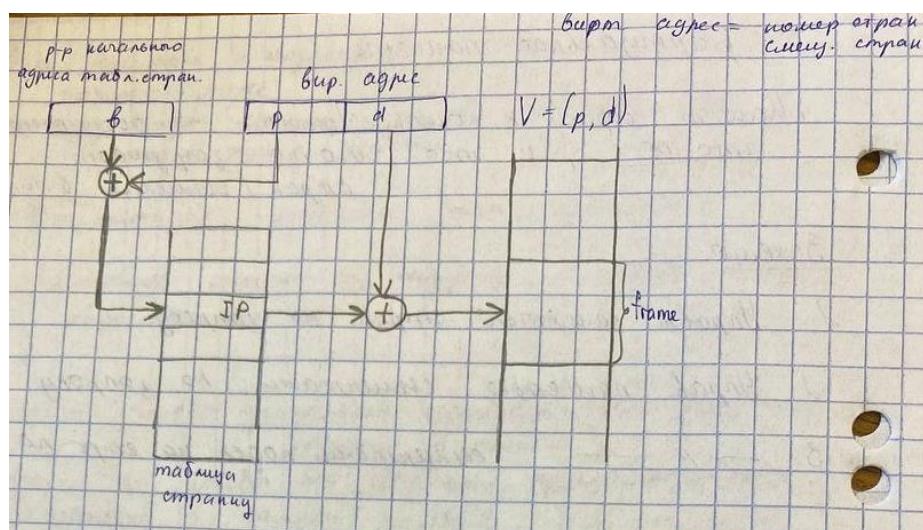
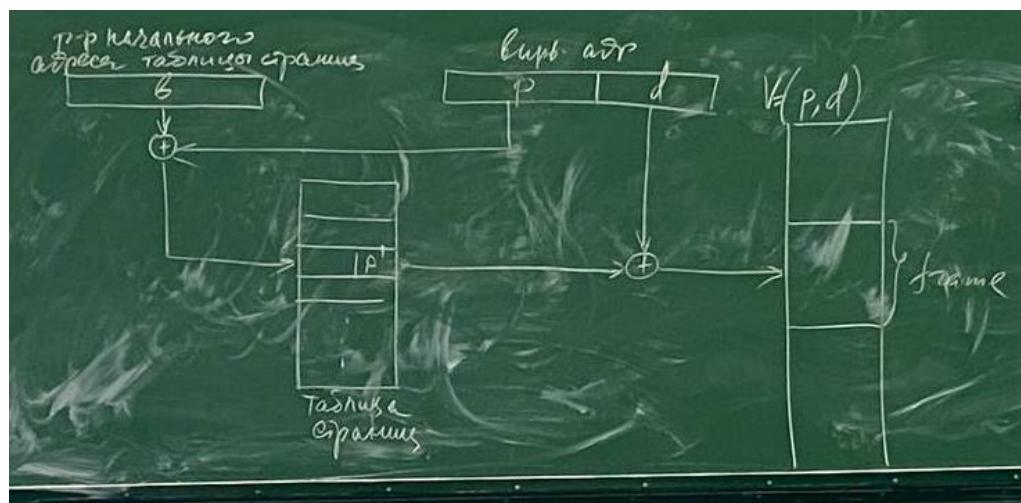
В момент порождения процесса система должна ему выделить минимально необходимый объем памяти, а в результате таких запросов загружаются необходимые участки кода.

Виртуальная (каждая возможной) память – память, объем которой превышает объем доступного физического адресного пространства (про диск – это понятно).

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

З метода преобразования адресов (Теоретические основы преобразования, как в intel – посмотрим потом)

4. Прямое отображение



Виртуальный адрес делится на 2 части – номер страницы и смещение. $V=(p,d)$

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. То есть при переходе он меняется. Таблица страниц одна (и собственная) на каждый процесс. При этом это системные таблицы.

Если страница загружена в физическую память, то у этой страницы будет адрес физической памяти. 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 загружена, выполняется соответствующее преобразование, аппаратно поддерживаемое.

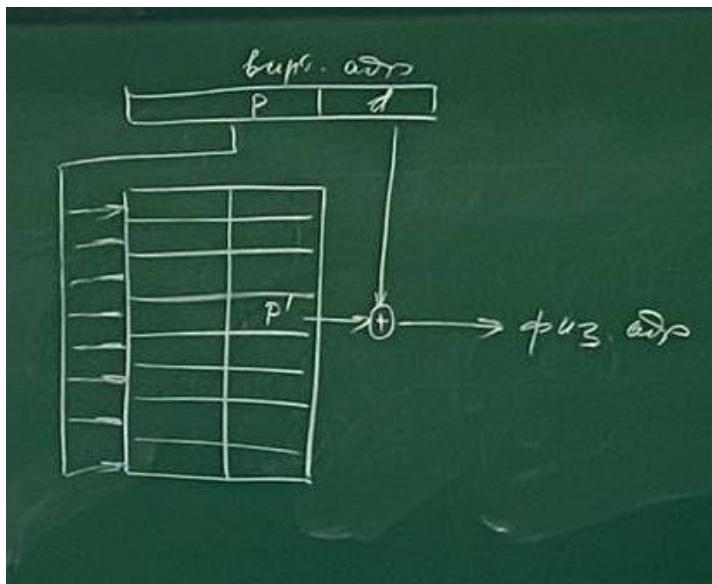
Таблица страниц находится в области памяти ядра системы (ОП). Процессов много, таблицы растут с ростом программ и что-то много места начинают занимать + Обращение к физической памяти затратное (цикл обращения к памяти – последовательность импульсов). Поэтому возник интерес ко второму

5. Ассоциативное отображение

Оно предполагает наличие в системе специального блока ассоциативной памяти (АП). АП всегда регистровая. Существуют 2 вида АП

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

АП позволяет получить доступ к информации по «ключу» за один такт. В результате схема имеет следующий вид.



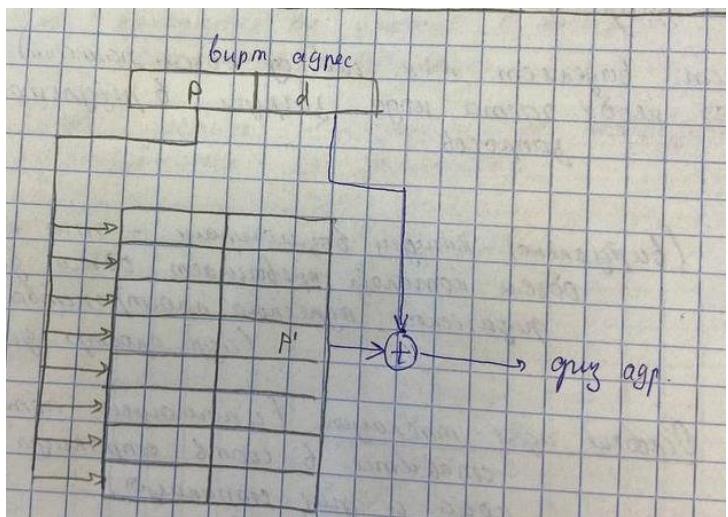
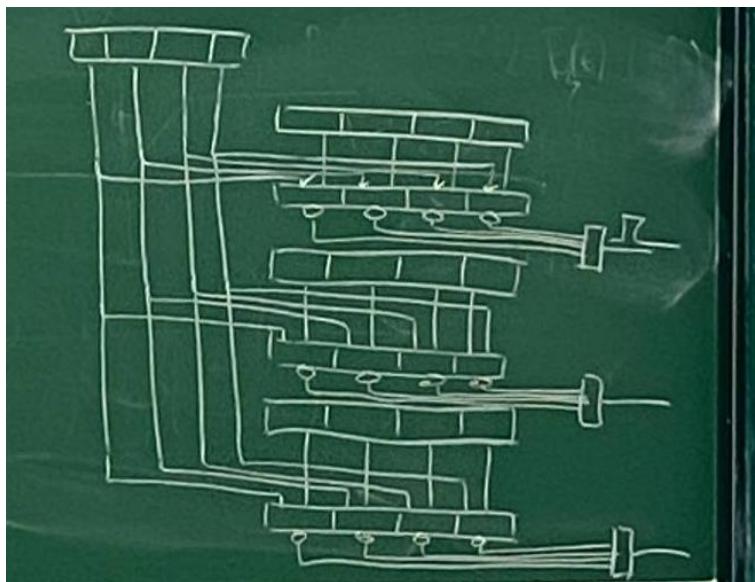
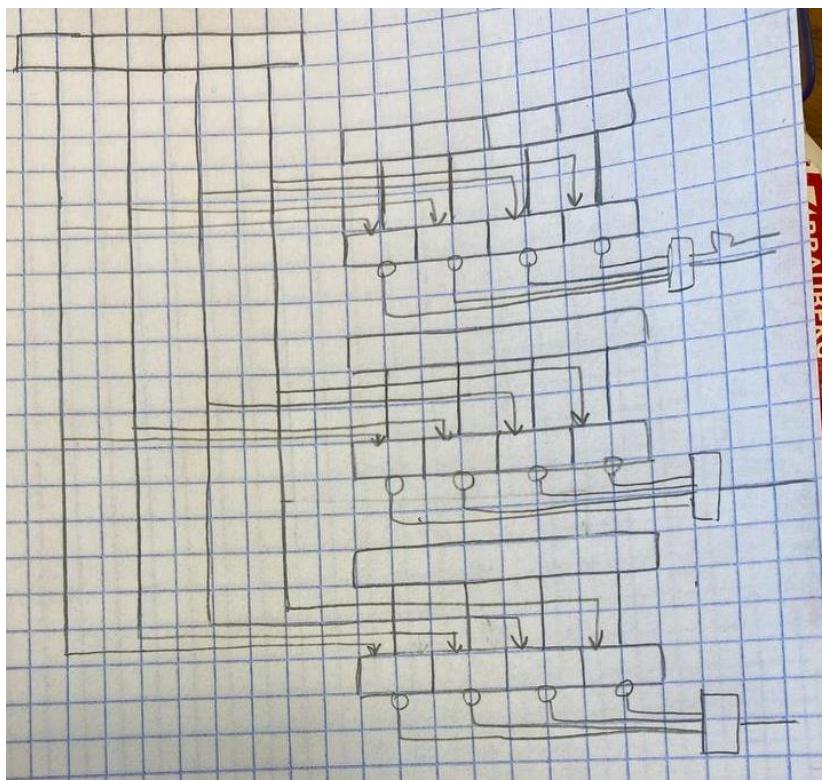


Таблица страниц загружается в специальную ассоциативную память. Номер страницы используется как ключ, сравниваются сразу все элементы всех дескрипторов и при совпадении выбирается физический адрес.

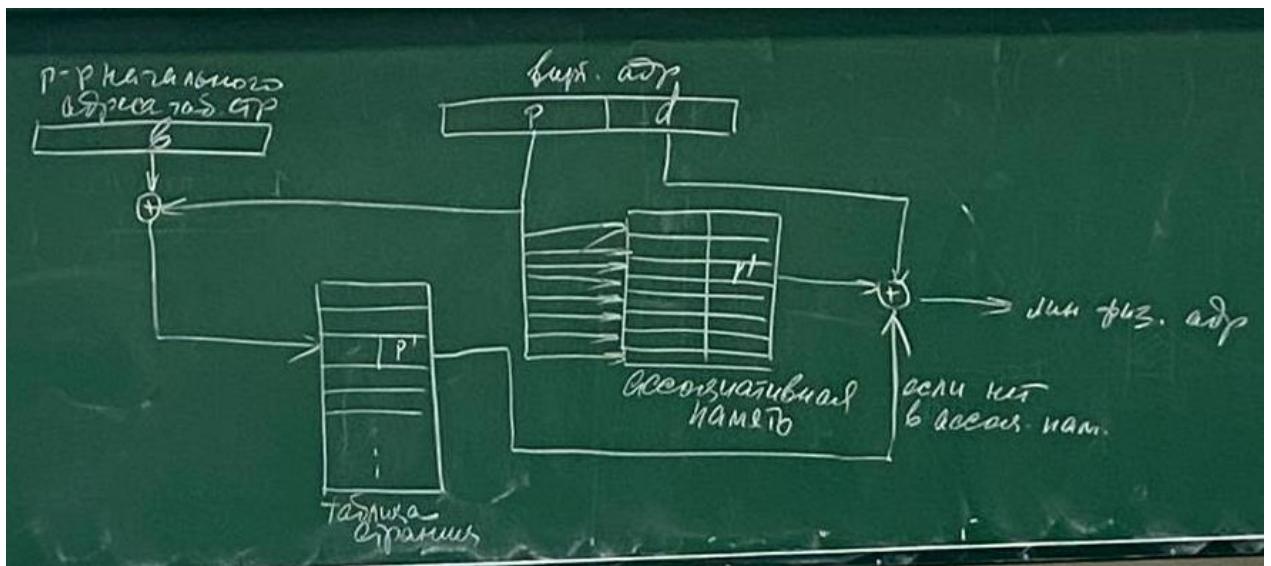
Но это дорогая память – она регистровая + чтобы осуществить такое обращение к информации необходимо на каждый разряд соответствующего регистра поставить схему совпадения. Собрать все сигналы на соответствующую схему, и если все сигналы совпали – послать сигнал разрешения считывания

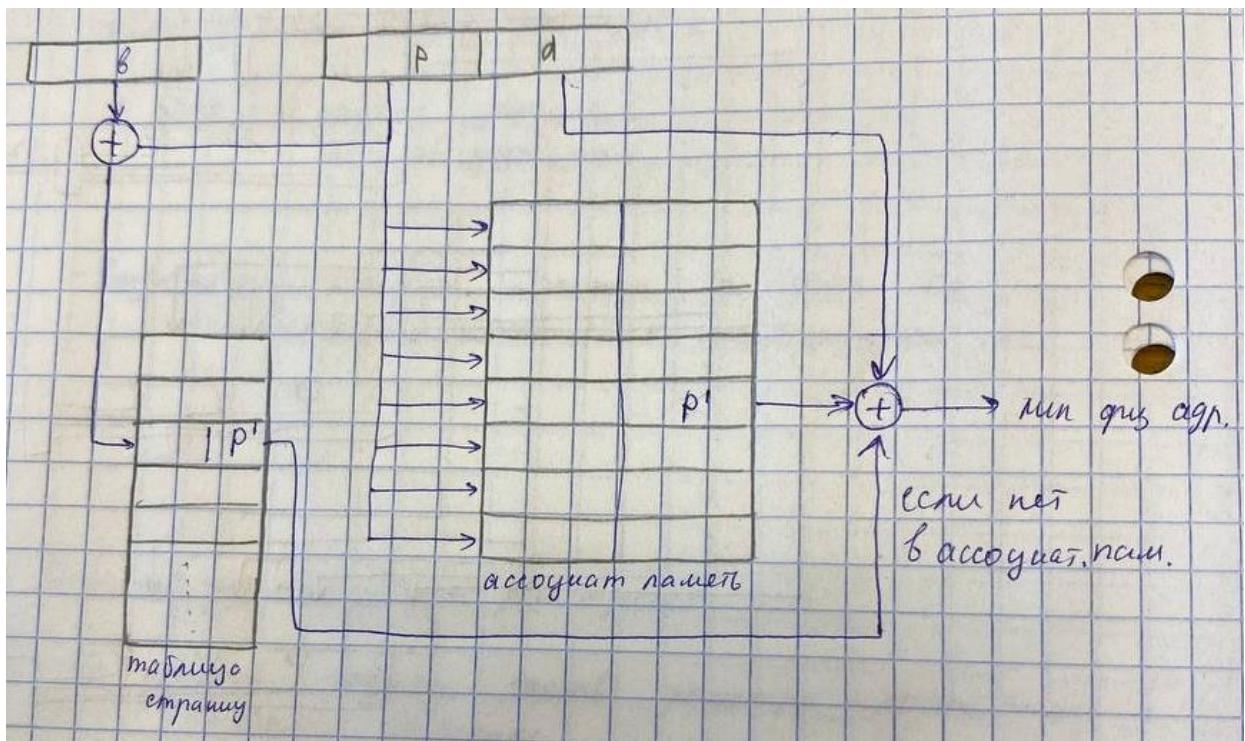




Логика удваивается + куча соединений. Сразу все разряды всех строк сравниваются с ключом, при совпадении всего – выборка информации (считывание). Дорого. Для больших таблиц страниц такое решение непригодно. Поэтому существует 3 схема

6. Ассоциативно-прямое отображение





В системе имеется небольшая по объему ассоциативная память (сейчас называется кэш). Сейчас – на 8/16 регистров, позволяла обеспечить 90% скоростных показателей полностью ассоциативной памяти

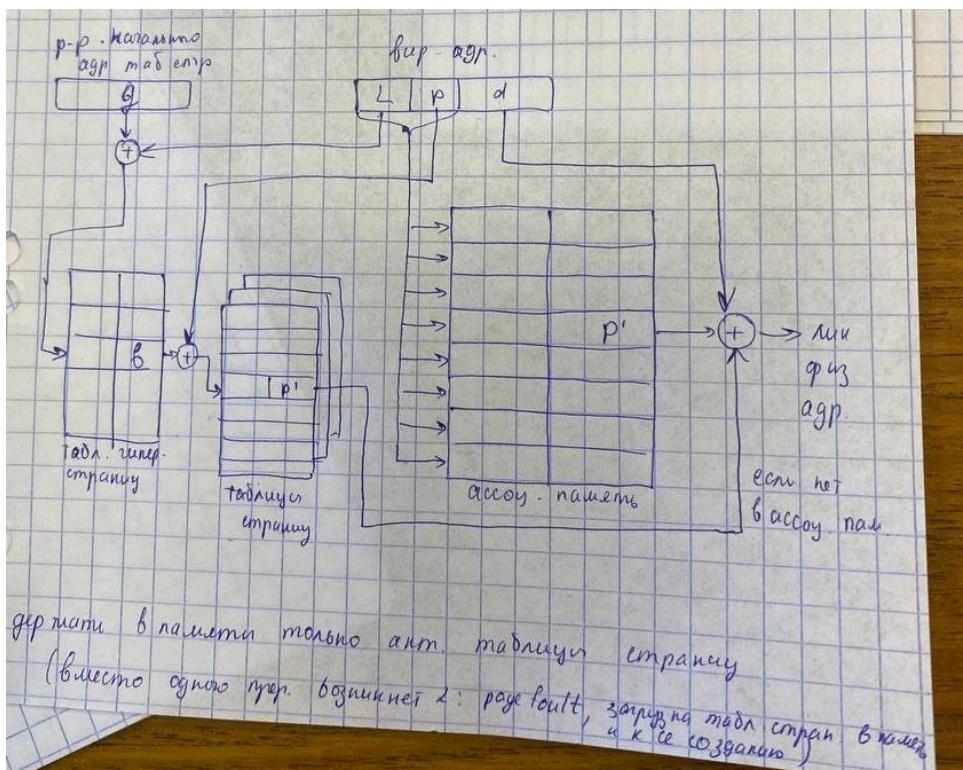
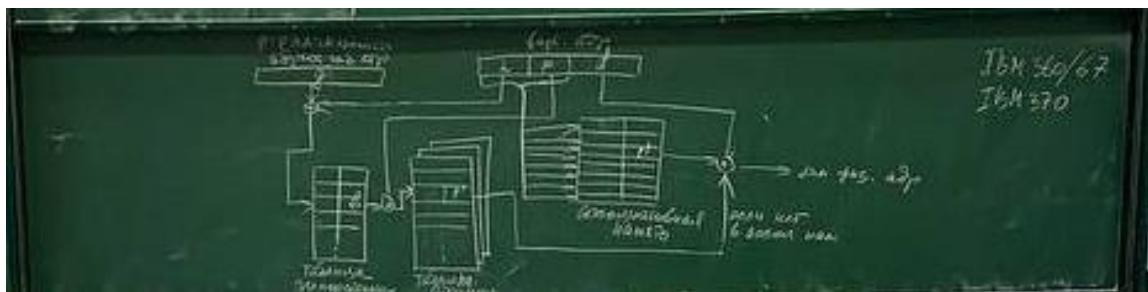
Страница сначала ищется в кэше. Если не найдена – обращение к физической памяти – таблице страниц. Если страница и там не найдена, то происходит страничное прерывание. Станичное прерывание в наших системах относится к исключениям – page fault. Это исключение, но исправимое.

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

Двухуровневая страничная организация

Была реализована в IBM360/67 и IBM370. 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.



Эти схемы вроде из дейтала (книга видимо)

У процесса 1 таблица гипер-страниц (1 на каждый процесс) и много таблиц страниц. Добавляется одно обращение к оперативной памяти. То есть время больше. Но здесь можно хранить в памяти только актуальные таблицы страниц.

Страница загружается в память только когда происходит обращение. Но вместо 1 прерывания возникнет 2: 1) page fault 2) нет таблицы страниц (загрузка таблицы страниц в память и, вероятно, ее создание, и потом загрузка страницы в память). Затраты очевидны.

Страницное преобразование пожирает процессорное время. Почему не отказались от идеи виртуальной памяти? В чем выигрыш – в увеличении мультизадачности. Можно хранить в памяти большее количество программ.

Здесь большую роль играет кеширование. 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 (опыта) 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:

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

Страницы – единицы физического деления памяти – размер страницы установлен в системе

Вопрос – зачем тогда нужны остальные схемы (сегментами поделенными на страницы и т.д.)?

Коллективное использование в задачах данных... и что-то еще непонятное

Заинтересованность большого числа процессов в одних и тех же программах и кодах ОС. Если каждому процессу предоставлять индивидуальную копию – крайне неэффективное использование памяти. В результате чтобы коллективное использование было эффективным, программы, которые коллективно используются должны быть реентерабельными – допускать повторную входимость. Это основная проблема, которая была при управлении памятью страницами по запросу.

В результате такой потребности (особенно касается данных) (один хочет поменять что-то на странице, а другой – взять). Тогда страницу надо обозначать отдельно для каждого процесса и выдавать права доступа. На уровне страниц теряется принадлежность конкретной программе или набору данных.

Алгоритмы замещения страниц

Все физические страницы выделены. Процесс обращается к отсутствующей –страничное прерывание – загрузить страницу, предварительно выгрузив другую страницу. Page replacement.
Алгоритмы:

- 4. 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.
- 5. 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

ПРИМЕР:

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

Вот тут доступно 3, Итого 9/12 – 75% неудач

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
M=3		4	3	2	1	4	3	3	3	5	2	2
		9	3	2	1	4	4	4	4	3	5	5
	+	+	+	+	+	+	+	+	+	+	+	+

$9/12 = 75\%$

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
M=4		4	3	2	2	2	1	5	4	3	2	1
		4	3	3	3	2	1	5	4	3	2	
	+	+	+	+	+	+	+	+	+	+	+	+

$10/12 \sim 83\%$

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

Лекция 7

Алгоритмы замещения страниц

Все физические страницы выделены. Процесс обращается к отсутствующей –страничное прерывание – загрузить страницу, предварительно выгрузив другую страницу. Page replacement.
Алгоритмы:

7. 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.
8. 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

ПРИМЕР:

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

Вот тут доступно 3, Итого 9/12 – 75% неудач

P	1	2	3	4	5	6	7	8	9	10	11	12
	4	1	3	2	1	1	4	1	5	1	5	1
M=3		4	3	2	1	4	3	3	3	5	2	2
		(4)	(3)	(2)	(1)	4	4	(4)	(3)	5	5	
	+	+	+	+	+	+	+	+	+	+	+	+

$9/12 = 75\%$

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	2	1	1	5	4	1	3	2	1
M=4		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)	(5)	(4)	3	
	+	+	+	+	+	+	+	+	+	+	+	+

$10/12 \approx 83\%$

9. LRU – least recently used - наименее используемый 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	2	1	4	1	3	5	1	4	3
$M=3$	4	3	2	1	4	3	5	4	3	2	1	
	4	3	2	1	4	3	5	4	3	2	1	
	+	+	+	+	+	+	+	+	+	+	+	

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	2	1	4	1	3	5	1	4	3
$M=4$	4	3	2	1	4	3	5	7	3	2	1	
	4	3	2	1	4	3	5	7	3	2	1	
	4	3	2	1	4	3	5	7	3	2	1	
	+	+	+	+	+	+	+	+	+	+	+	

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. Это априорное свойство, оно косвенно доказывается исследованиями. Короткое определение свойства локальности – если было обращение к странице, то наиболее вероятно, что следующие обращения будут к этой же

странице. Это свойство следует из следующих особенностей программ, которые мы пишем: 1) если в наших программах отбросить goto, то существует только 3 действия: следование, ветвление и повторение (в котором есть ветвление - условие по которому переходим). Если возвращаться, теорема Бема-Якопини – любой алгоритм может быть представлен структурами 3 типов-следование, ветвление, повторение. Русские – достаточно 2 – следование и повторение (но так неудобно)

Алгоритм LRU полностью соответствует свойству локальности. В программах более вероятно, что команды следуют одна за другой. Ветвления встречаются реже. Если взять хранение данных, то строки, массивы хранятся в последовательных адресах. И даже ООП не внес сильных изменений (все в виде функций – функционально законченное действие – просто if повесить – бессмысленно).

Кроме того, если посмотреть на эти 2 рисунка, заметим, что первые 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.

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).

10. LFU – least frequency 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.

11. 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.

NUR

	страница	бит обращения
0	4	0
1	5	1
2	2	1
3	1	0

указатель удаления
циклическое обращение
перед удалением

А этот - после удаления

	страница	бит обращения
0	4	0
1	5	1
2	2	0

указатель удаления
доступен
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 обращения	бит модиф
0	0
0	1
1	0
1	1

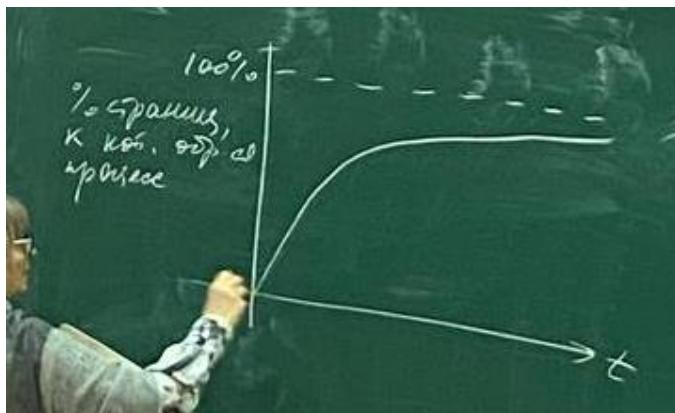
Вызывает вопрос вторая строка. Как так – обращения не было, а модификация была? Значит, эта страницы модифицировалось до сброса всех битов обращения в 0.

Страницному преобразованию (управлению памятью страницами по запросу) посвящено много статей – Медник-Денован (IBM360)->IBM скрупулёзно исследовало этот вид управления памятью). Собственность IBM. В тик-токи ваши не выкладывать!

Поведение программ и производительность

На прошлой лекции было понятно, как затратен механизм управления виртуальной памятью. Если это так очевидно затратно (преобразование адресов, обработка страницных прерываний) – так зачем это надо? Ответ/преимущество – увеличение мультизадачности (или увеличение уровня мультипрограммирования) за счет того, что снимаются ограничения, которые накладываются размером физической памяти. Но цена этому – загруженность процессора соответствующими обработками (действиями).

Рассмотрим график зависимости процента страниц, к которым обращается типичный процесс, от времени. Время здесь – от начала выполнения процесса до его завершения.



Понятно, что к концу 60-х - начале 70-х программы были размером несопоставимы с современными, но можно эти исследования анализировать. Здесь несколько искажен исходный график (Рязановой, и она об этом говорит) и асимптотически не достигает 100%, так как программы чаще всего не обращаются ко всем своим страницам, особенно современные, так как там используются динамические библиотеки и тд. Все зависит от прогона.

Но сам график демонстрирует важнейшую вещь – процесс какой-то отрезок времени выполняется, но не обращается ко всем своим страницам. Причем график нелинейный: сначала линейный (это интенсивная подгрузка), а потом нелинейный – уже не подгружает интенсивно – «знак вопроса»

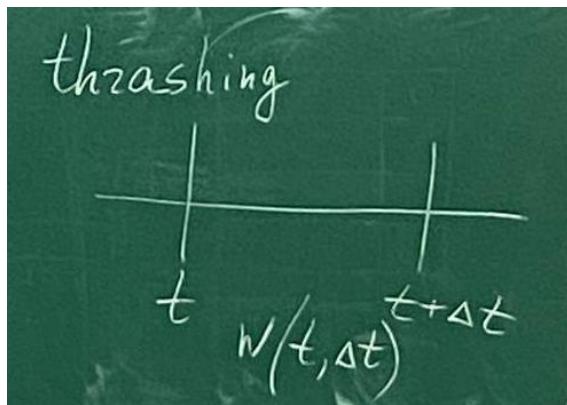
Здесь на рисунке нет прямого упоминания о страницных прерываниях, с помощью которых страница загружается

Теория рабочего множества

Зависимость страницных прерываний от объема памяти является обобщенной мерой страницного поведения программы. При этом, как видно из фотографии памяти (условно называем так картинку по рядам), поведение программы по времени выполнения не является стабильным и равномерным, то есть 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 .

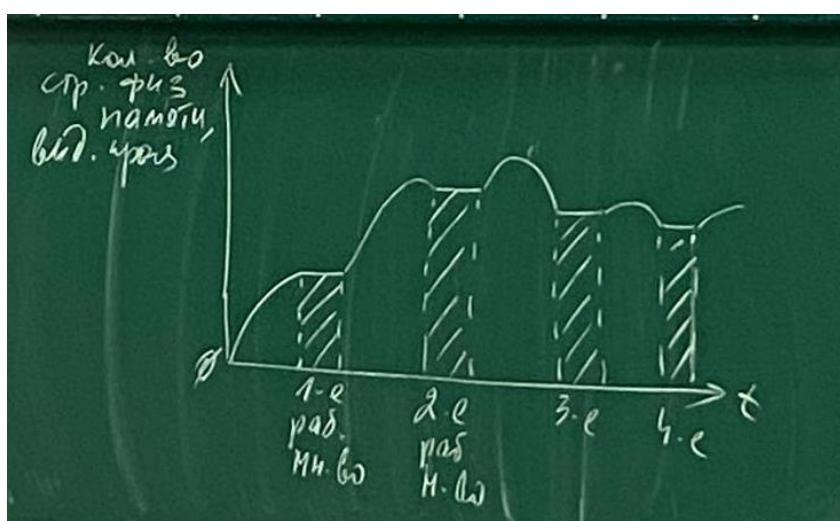
Working set – рабочее множество



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.

Можно перефразировать – чтобы процесс выполнялся эффективно, необходимо, чтобы он мог загрузить в память все свое рабочее множество. А это уже – оценка памяти которая должна быть выделена процессу для его эффективного выполнения.

При этом понятно, что предсказать – какие страницы в какой момент времени процесс затребует или будет обращаться – невозможно, это вероятность, но по факту процесс должен иметь возможность загрузить все свое рабочее множество. 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 выполнятся, при этом интенсивно подгружая страницы, пока в памяти не появится первое рабочее множество. Время предсказать

невозможно, процесс будет выполняться без прерываний (если ему удалось подгрузить все необходимое).

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)

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.

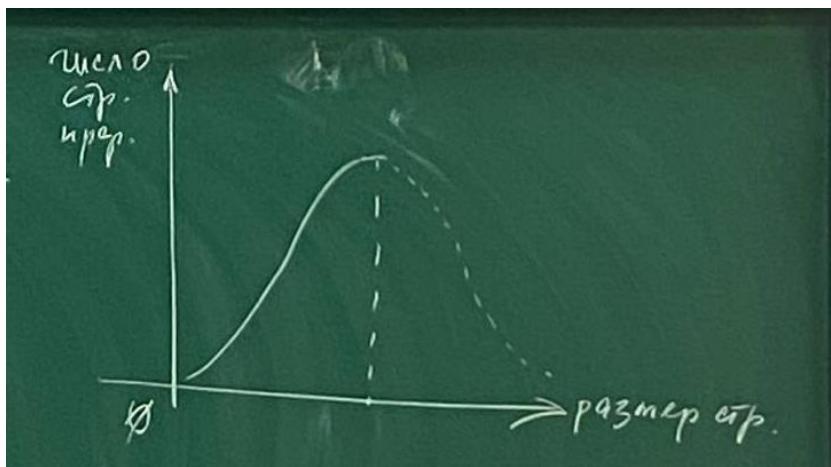
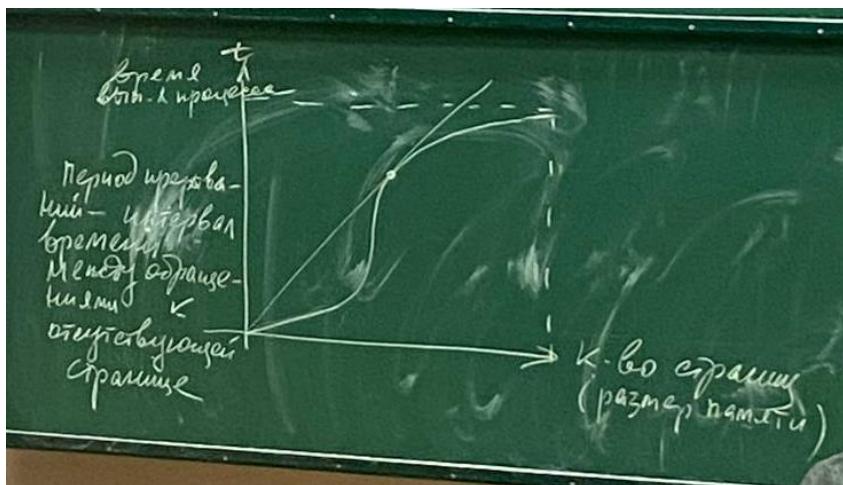


График опять же не линейный, но из него видно, что число прерываний растет с ростом размера страницы, причем интенсивно. Это связано с тем, что чем больше размер, тем меньше процент команд на ней будет выполнен и потребуется переход на другую страницу. Когда максимум достигнут, идет спад, потому что 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

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. 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.

Управление памятью сегментами по запросу

Страница – единица физического деления памяти, размер страницы определен в системе.

Сегмент – единица логического деления памяти. Поэтому дополнительно о нем должен быть известен его размер.

Схема преобразования при этом будет выглядеть следующим образом:

(рисунок) – след раз

Лекция 8

Управление памятью сегментами по запросу

Страница – единица физического деления памяти, размер страницы определен в системе.

Сегмент – единица логического деления памяти. Поэтому дополнительно о нем должен быть известен его размер.

Схема преобразования при этом будет выглядеть следующим образом:

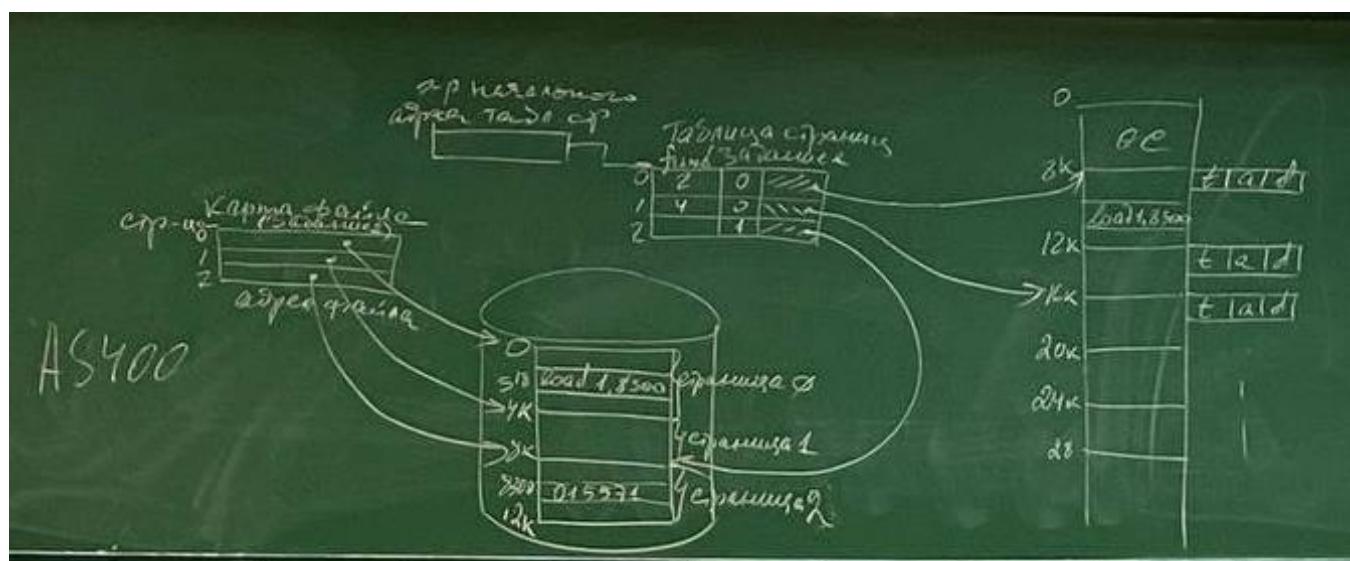
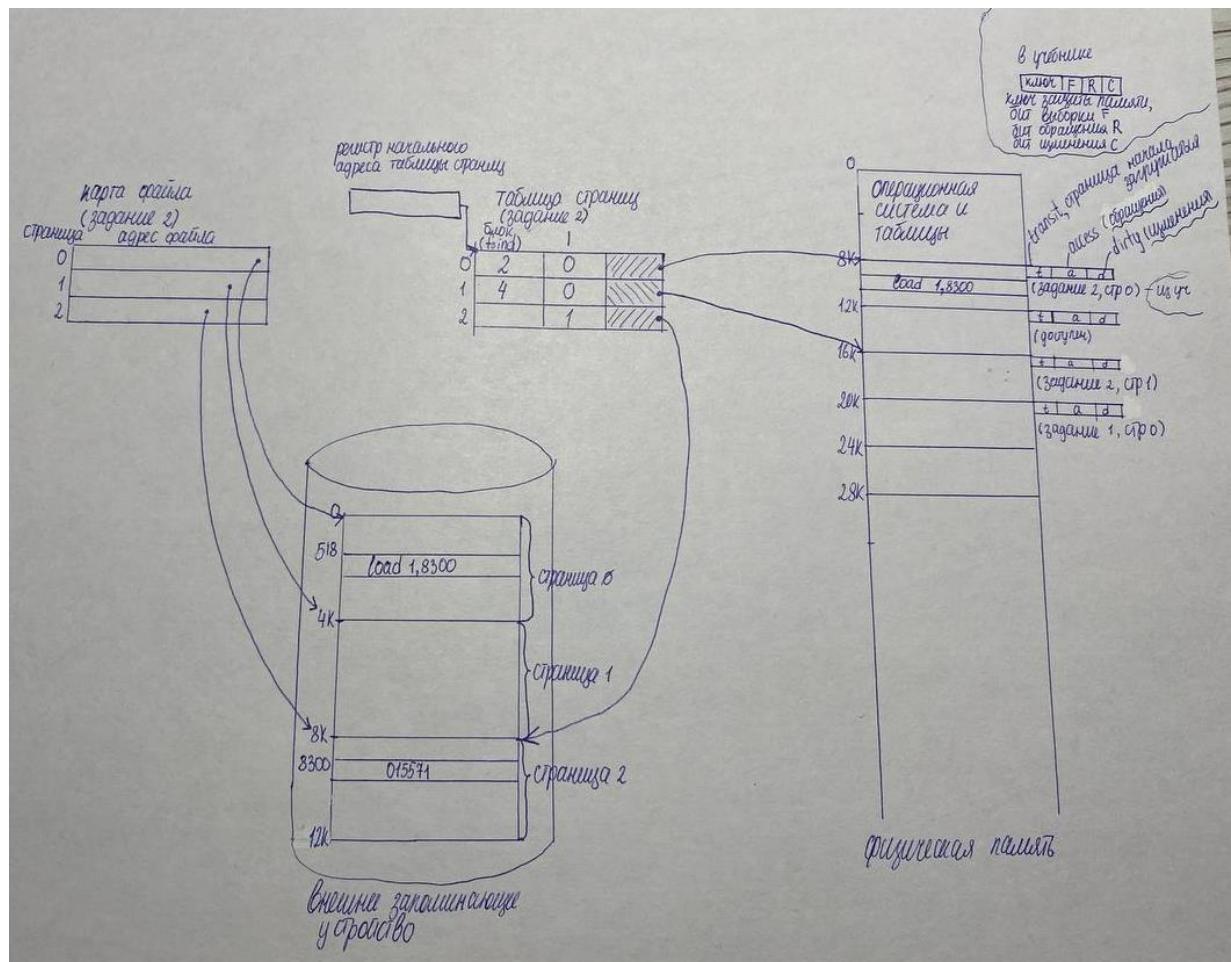
(рисунок) – след раз

Реализация управления памятью

В 64 разрядных аппаратно сегментация не поддерживается. ЛР по защищенному режиму – работа с сегментацией (GDT). Сегментация в ЗР – запутанная тема. Она была реализована в самых первых процессорах, поддерживающих виртуальную память. Но от нее уходят, переходя на страничное преобразование.

Рассмотренные ранее схемы реализованы в реальных системах, это рабочие вещи.

Начинаем со старой инфы – Медник и Денован (IBM 360). Цифры оставлены, как было в книге. Значения флагов изменены, в современных системах состояние transit не указывается, но мы оставим – значит, что страница начала загружаться, но еще не загружена (остальные – access, dirty)
(Load 1,8300 – загрузить в регистр 1 значение по адресу 8300)



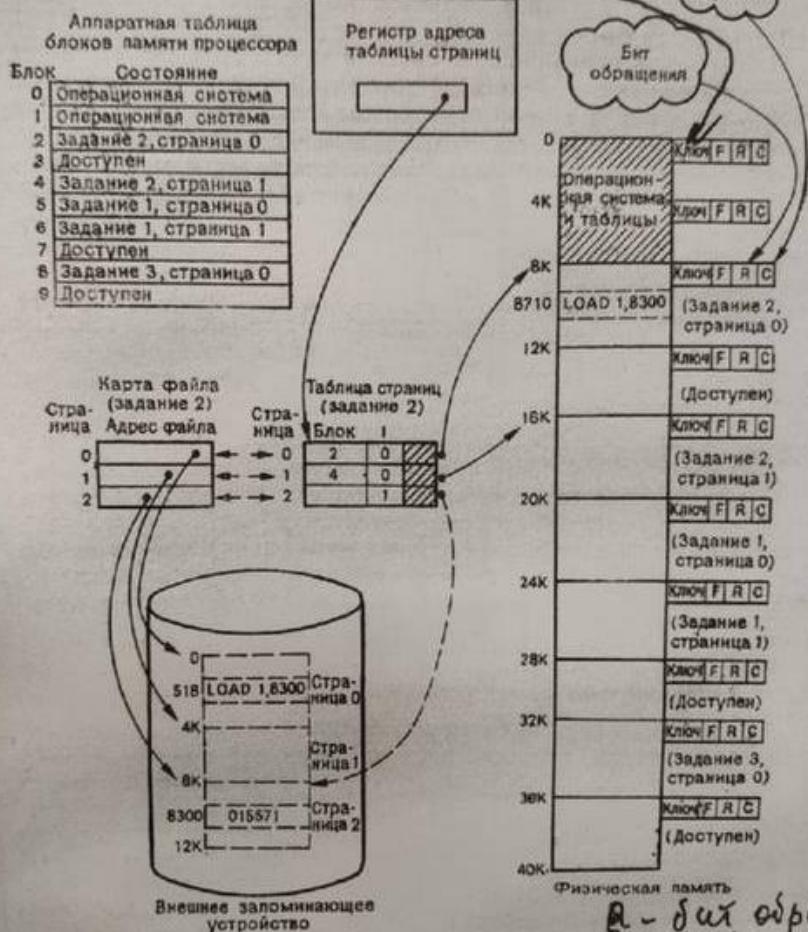


Рис. 3.22. Взаимосвязь карты файла с другими таблицами.

о состоянии) обрабатывается только программно, и поэтому формат этого поля может быть переменным. Более того, информация о файле обычно хранится в отдельной таблице, называемой картой файла, которая не используется аппаратными средствами.

<http://repo.ssau.ru/bitstream/Metodicheskie-ukazaniya/Algoritmy-upravleniya-pamyatu-Elektronnyi-resurs-metod-ukazaniya-k-lab-rabote-po-kursu-Sistem-programmirovaniye-53513/1/Куприянов%20А.В.%20Алгоритмы%20управления.pdf>

Картинка показывает, что на самом деле программа лежит во вторичной памяти. Безусловно, в системе должна существовать информация, которая позволяет загружать из вторичной в оперативную. То есть используется АП вторичной памяти. При этом АП диска делится на 2 неравные части. 1 часть – область swapping (paging), размер определялся пользователем вручную раньше, сейчас – необязательно, но можно поменять размер. Остальная часть отведена файловой подсистеме (обычные файлы – файлы, которые создаются различными приложениями – блокнот, Word, Paint, – для всех характерно, что они хранятся на диске, то есть во вторичной памяти – пользователь заинтересован в длительном хранении на энергонезависимых устройствах. Это

связано с парадигмой «В Unix все файлы». В системах есть и другие файлы- softlink, pipe, различные устройства.

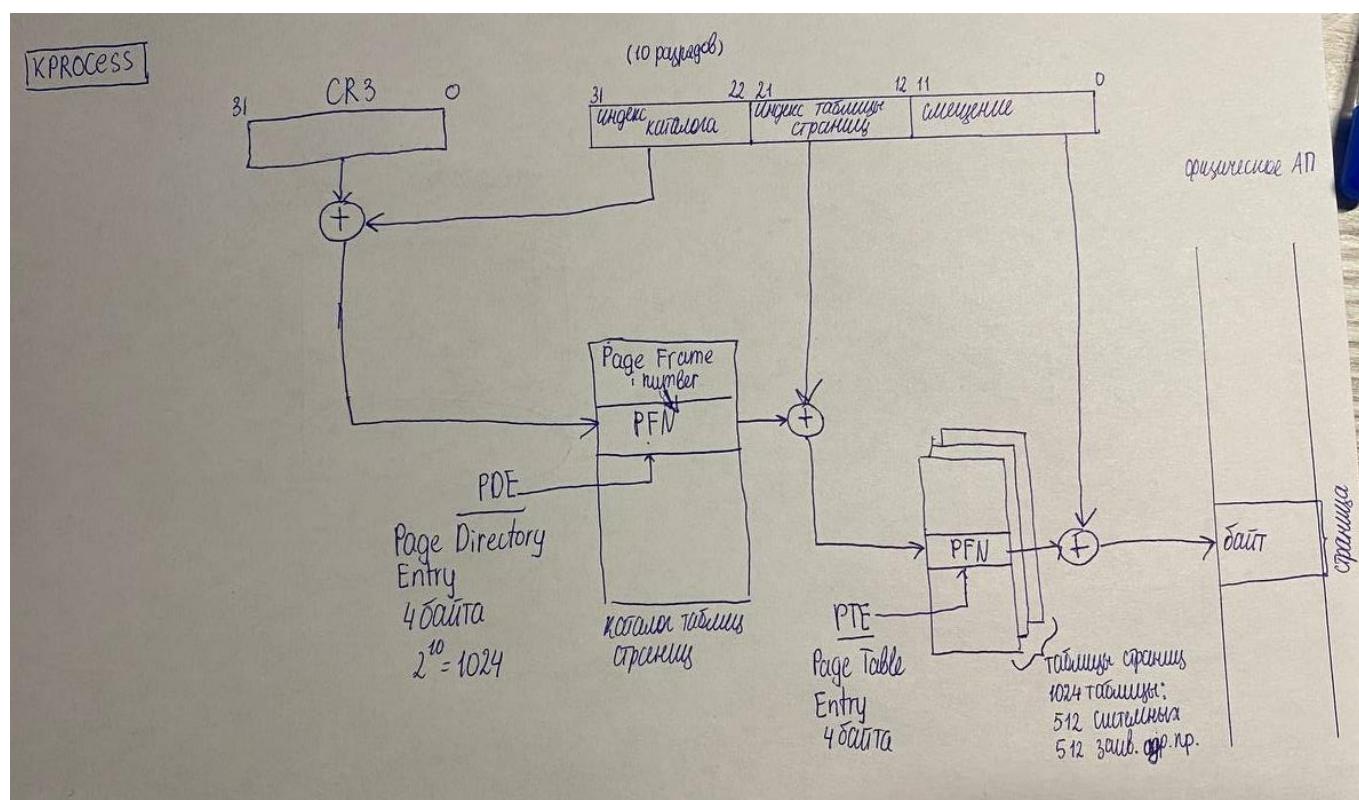
Видно, что программа во вторичной, есть таблица страниц и карта файла. Страница, которая подгружается тоже рассматривается как файл – скопировать содержимое страницы в ОП

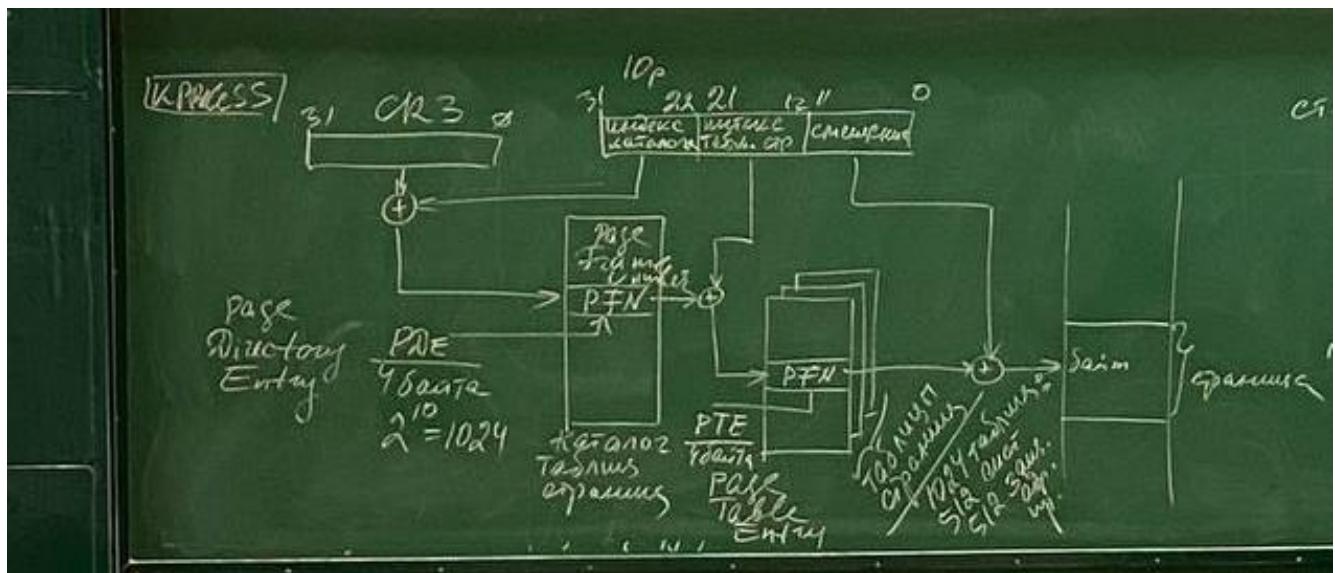
Если в системе выделена специальная область для paging, а это сейчас делается, то: сама программа (исполняемый файл) занимает АП диска, и из этой области начинается копирование страниц в ОП и вытеснение. В результате программа существует как минимум в 3 копиях. Чтобы использовать АП эффективней, используется так называемая одноуровневая память.

(AS400 – книга). Одноуровневая память: paging выполняется в адресное пространство файла, но для этого конечно надо иметь специальную таблицу для реализации отображения страниц файла в физическую память

В ЗР 2 типа страничного преобразования – стандартное и РАЕ. При стандартном 32р виртуальный адрес делится на 3 части. Эта схема коррелирует со схемой гиперстраниц

CR3 - содержит физический адрес начала каталога страниц и два флага: PCD и PWT. Этот регистр также называется PDBR (Page-Directory Base Register), он хранит 20 старших бит адреса каталога страниц (младшие подразумеваются равными 0, т.к. каталог страниц должен быть выровнен на границу страницы). Биты PCD и PWT управляют кэшированием каталога страниц во внутреннем кэше данных процессора, но не управляют TLB-кэшированием. При использовании расширения физического адреса, регистр CR3 содержит базовый адрес таблицы PDP (Page Directory Pointer).

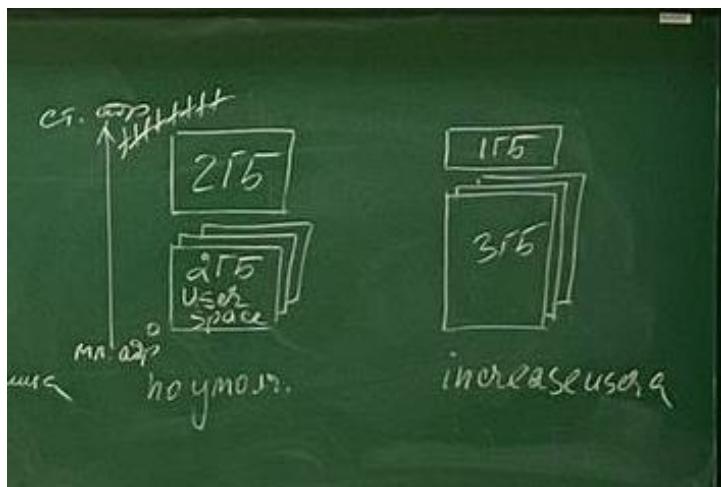




4кб страницы – не единственный поддерживаемый размер, но это обычный размер для приложений. Есть схемы с 1кб и 2кб - для баз данных. Но стандартно 4кб, тогда смещение 12 бит ($2^{12} = 4\text{кб}$).

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.

В виндах страницы делятся пополам – mapping системы и защищённое адресное пространство.



2 типичные структуры виртуального АП ОС виндовс.

2 ГБ. Это – стандарт – по умолчанию

А тут (1 гб) надо указывать параметр increase... И эта штука используется в unix.

(с сайта Microsoft) виртуальное адресное пространство для 32-разрядного Windows имеет размер 4 гигабайта (гб) и делится на две секции: одна для использования процессом, а другая зарезервирована для использования системой.

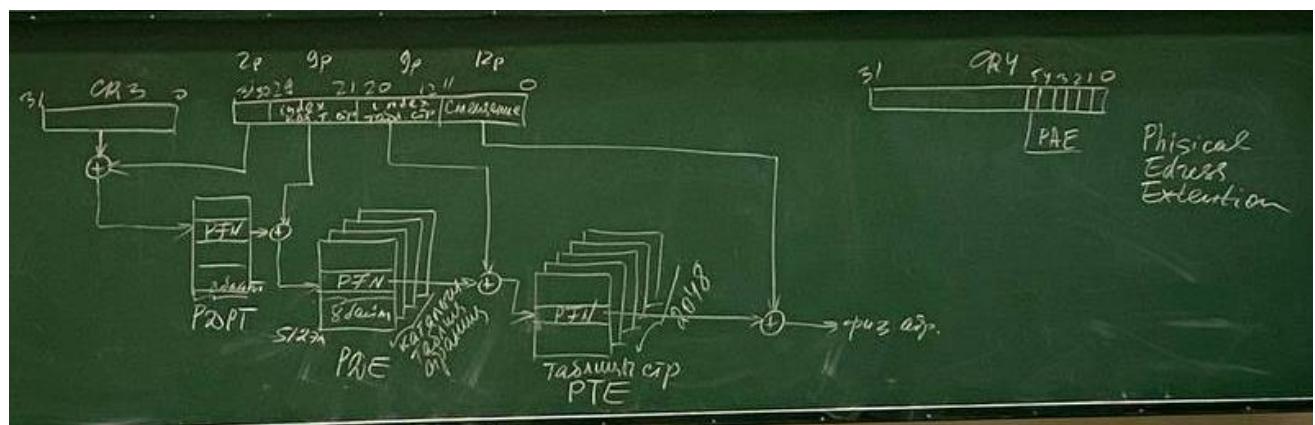
Виртуальное адресное пространство по умолчанию для 32-разрядного Windows: Низкая 2 ГБ - Используется процессом, Высокий 2 ГБ-Используется системой.

виртуальное адресное пространство для 32-разрядного Windows с 4ГТ: (Если включена Настройка 4 гигабайта (4GT)): низким 3 ГБ -Используется процессом, Высокий 1 ГБ - Используется системой.

Вообще код windows больше чем на порядок больше unix, а значит хуже (если код больше другого при одной и той же функциональности, значит больший - хуже). В unix системе всегда в 32р системах выделяется 1гб. В 64 доступно уже 128Тб, но об этом позже.

Это стандартная схема преобразования. Стремление держать в памяти только актуальные страницы.

Следующее преобразование еще лучше это демонстрирует (это видимо PAE)

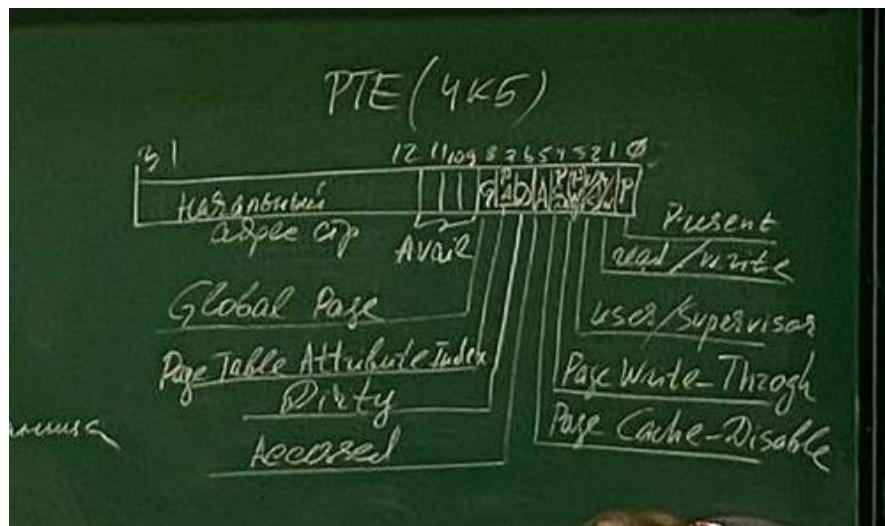


32р виртуальный адрес делится на 4 поля. Каждое такое деление связано с дополнительным обращением к ОП при преобразованиях. Каталог ТС, базовый адрес ТС, адрес страницы. Минус, но зато ТОЛЬКО актуальные таблицы.

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

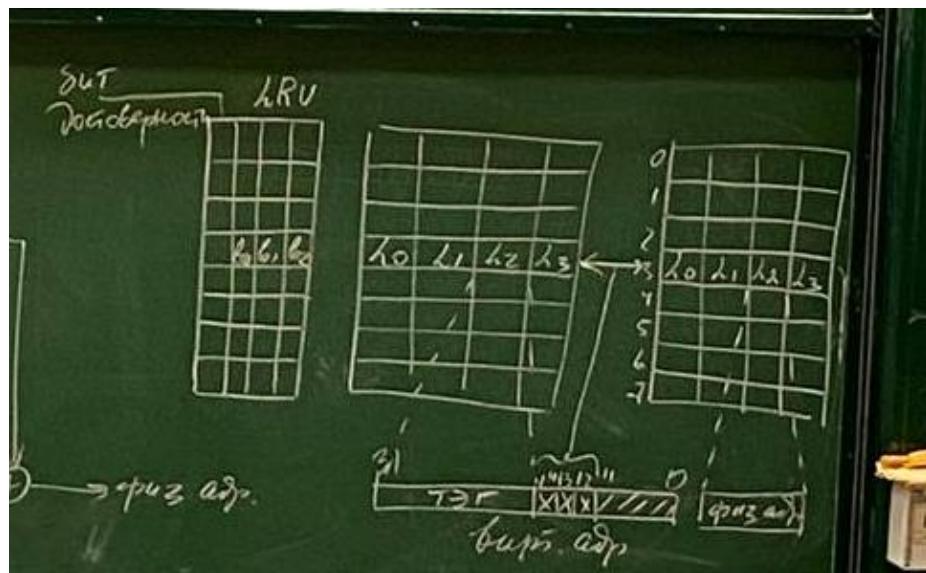
Посмотрим на соответствующие дескрипторы. Рассмотрим дескриптор таблицы страниц РТЕ для 4 Кб страницы. 11 разрядов смещения используются в виде флагов и оставшиеся разряды – начальный адрес страницы – PageBaseAddress



Access dirty для аппроксимации RLU

Еще один уровень = + 1 обращение к физической памяти.

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



TLB – четырехнаправленный ... по множеству кеш

Это только идея, внутри – секреттт.

Мы обращаемся к tLB translation ... buffer 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:

- а) если обращение в множестве было к строке I0 или I1, то бит P0 устанавливается в 1, если к I2 / I3 – сбрасывается в 0
- б) если обращение в паре к I1I0: I0 то бит p1=1, I1 p1=0
- в) I2I3: I3 – b2=0, I2 – b2=1

дополнение к рисунку слева x0

Утверждается, что в типичных системах cashtlb 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).

Лекция 9

Загрузка частей программы в память выполняется по запросу.

(объединено с chislvt)

Управление памятью сегментами по запросу

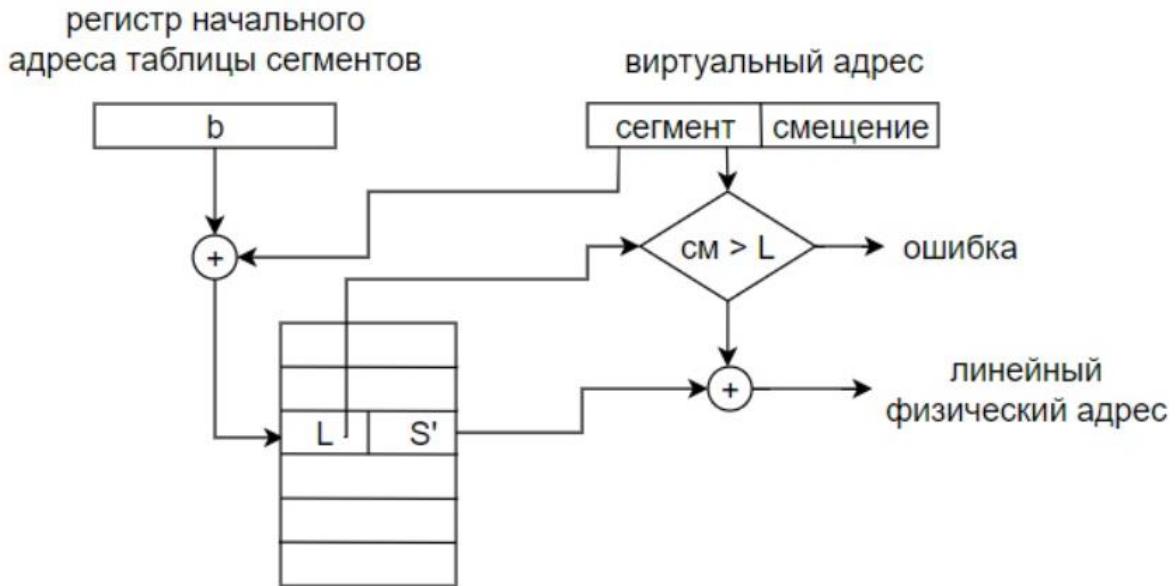
Страницы являются единицей физического деления памяти. Размер страницы определён в системе.

Сегменты являются единицей логического деления памяти. Размер сегмента определяется объёмом программного кода.

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.*

В самом общем виде схема выглядит следующим образом



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.

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. То есть система контролирует выход процесса за его адресное пространство. Это самый простой способ контроля. Это нормальное инженерное решение. Там, где начинают выдумывать непонятно что, как правило ничего не получается. Инженерные решения - это как правило простые решения. Ведь системе проще всего контролировать выход процесса за его адресное пространство. Но еще говорят по-другому. Ни один процесс не может обратиться в адресное пространство другого процесса. Но контроль осуществляется именно таким образом.

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

В частности, мы с вами видели на примере процессоров Intel, поскольку мы с вами уже выполнили lr перевода компьютера из реального режима в защищённый режим и обратно. Там как раз используется управление памятью сегментами, но мы не в полной мере насладились конечно этим процессом, мы с вами рассматривали только таблицу глобальных дескрипторов, которая описывает сегменты физической памяти, но представление можем себе создать. Соответственно, дескриптор сегмента мы с вами видели.

Можете спросить, 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.

Как уже было сказано с каждым сегментом связан дескриптор. Этот дескриптор содержит адрес сегмента, его размер, соответственно 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.

Реализация сегментированной адресации зависит от организации таблиц дескрипторов сегментов. Существует 3 подхода к организации таких таблиц.

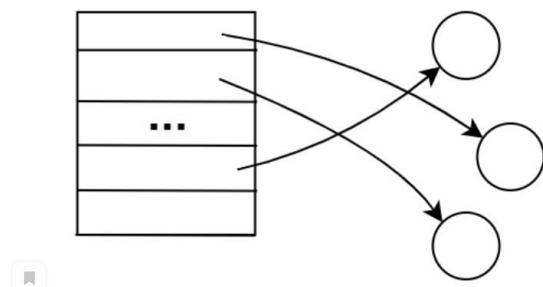
1) Единая таблица.

Все сегменты описаны в единственной таблице. То есть они имеют глобальные имена.

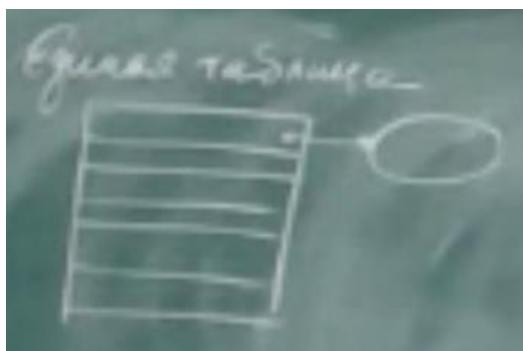
Каждый дескриптор в ней описывает сегмент физической памяти

Одна единая таблица которая содержит все дескрипторы сегментов и выполняемых программ. При этом у каждого такого сегмента в системе существует одно единственное имя, которое и является идентификатором дескриптора таблицы. В этом случае такой дескриптор должен содержать список прав доступа для всех процессов, которые могут использовать данный сегмент. Такой список прав доступа может быть очень большим. Понятно, что такая система не является гибкой, все процессы обращаются к сегменту по одному единственному его имени

Единая таблица



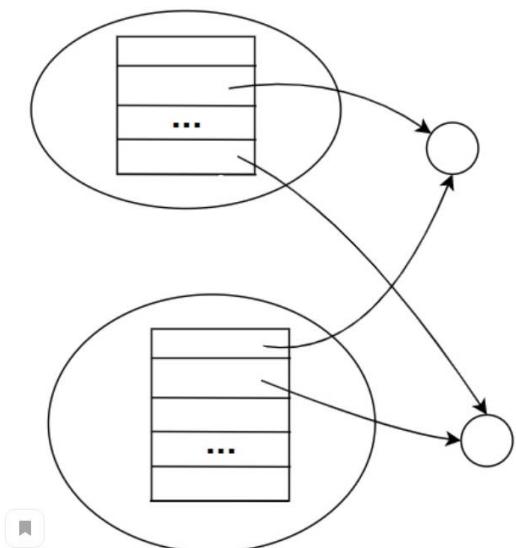
(на нашей лекции была такая картинка)



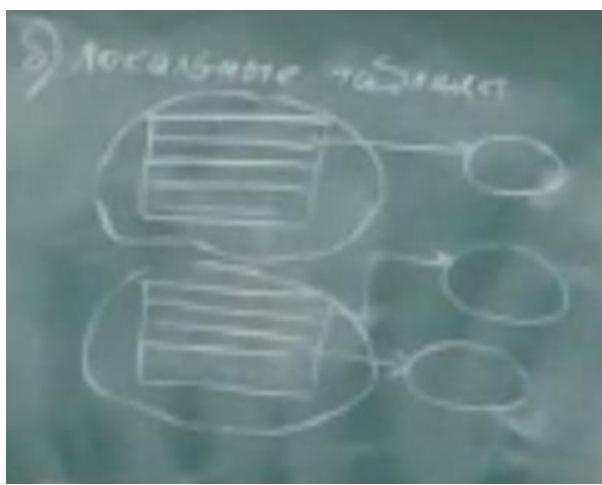
- 2) Локальные таблицы. АП каждого процесса описывается локальной таблицей дескрипторов. В этих таблицах необходимо указывать выделенные сегмент физической памяти.

Для того чтобы другие процессы могли обратиться к этому же сегменту, этот сегмент должен иметь дескриптор в соответствующих локальных таблицах. То есть для того чтобы процесс смог обращаться к сегментам других программ, эти сегменты должны быть описаны в их локальных таблицах, но при этом получается, что один сегмент в системе может иметь несколько разных имён. Это безусловно усложняет работу с сегментами в системе.

Локальные таблицы



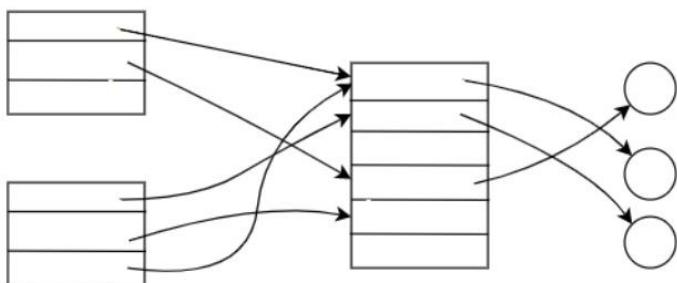
(на нашей лекции была такая картинка)



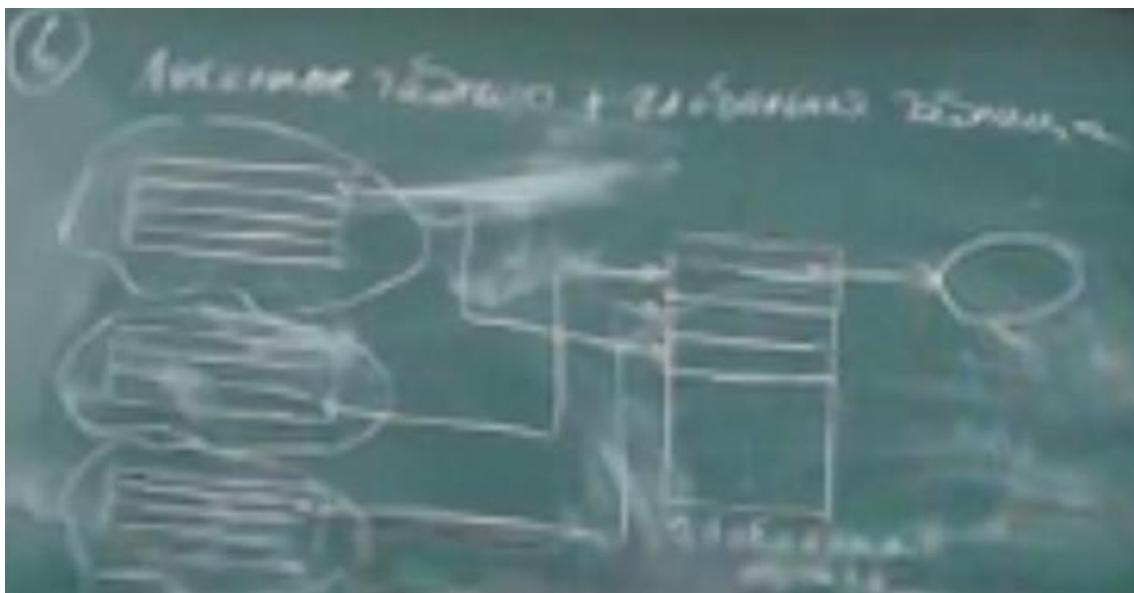
- 3) Локальные таблицы и одна глобальная таблица. Локальные таблицы описывают адресное пространство процессов, но дескрипторы ЛТ содержат ссылку на дескриптор в глобальной таблице дескрипторов. А глобальная содержит уже адреса сегментов в физической памяти

Так ли это в Intel? Мы рассматривали GDT, LDT, так ли там?

Локальные таблицы + глобальная таблица



(на нашей лекции была такая картинка)



Сегментация появилась в истории параллельно со страницным преобразованием.

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

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

Интерес к сегментации обусловлен тем, что сегмент является логической единицей деления памяти. Он определяется соответствующей программой т.е. программа занимает определённый сегмент т.е. связана с логикой программы в отличие от страниц. Но если посмотреть на подкачку сегментов, то во-первых мы уже рассматривали управление памятью разделами, перемещаемыми разделами - всё это справедливо для сегментации. 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

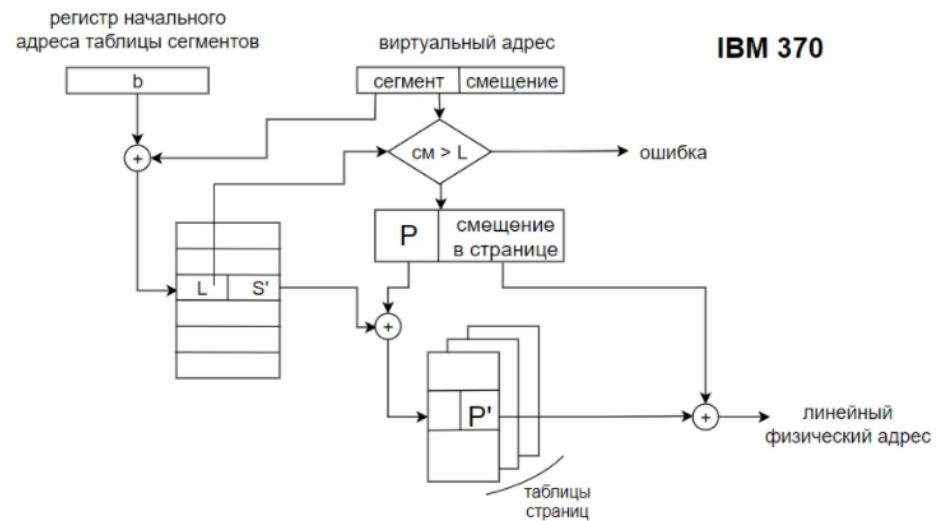
выполняется постоянно. Понятно что у нас перемещаемые сегменты, но всё равно любое перемещение приводит к соответствующим затратам.

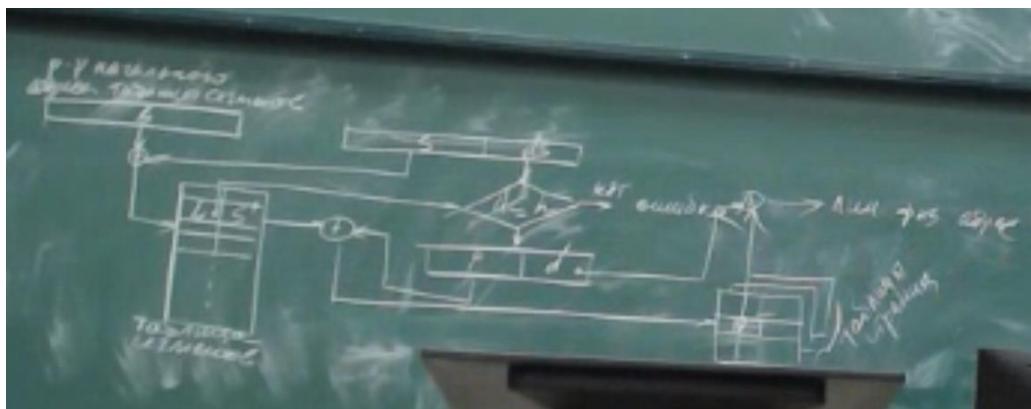
Стратегии выбора сегмента

Стратегии выбора освободившегося раздела для загрузки сегмента будут такие же как в старых схемах (предыдущих). Стратегия было 3: первый подходящий, самый тесный, самый широкий. Эти стратегии и здесь будут работать. Но при этом так как это виртуальная память, то остаются задачи вытеснения 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. Безусловно здесь так же работает тот же признак. Самым лучшим является трудно осуществимый алгоритм LRU. Для загрузки новых сегментов необходимо выбирать сегменты для замещения, но 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.

В связи с этим появлялся третий способ

Управление памятью сегментами, поделенными на страницы по запросу (коротко - сегментно-страничное)





ДААА

Сегменты делятся на страницы. Размер сегмента должен быть кратен размеру страницы. Появляется дополнительный уровень преобразований. Каждый процесс имеет 1 таблицу сегментов и столько таблиц страниц, сколько сегментов. Это похоже на двухуровневую страничную организацию. Сегмент делился на страницы и его размер определялся размером кода.

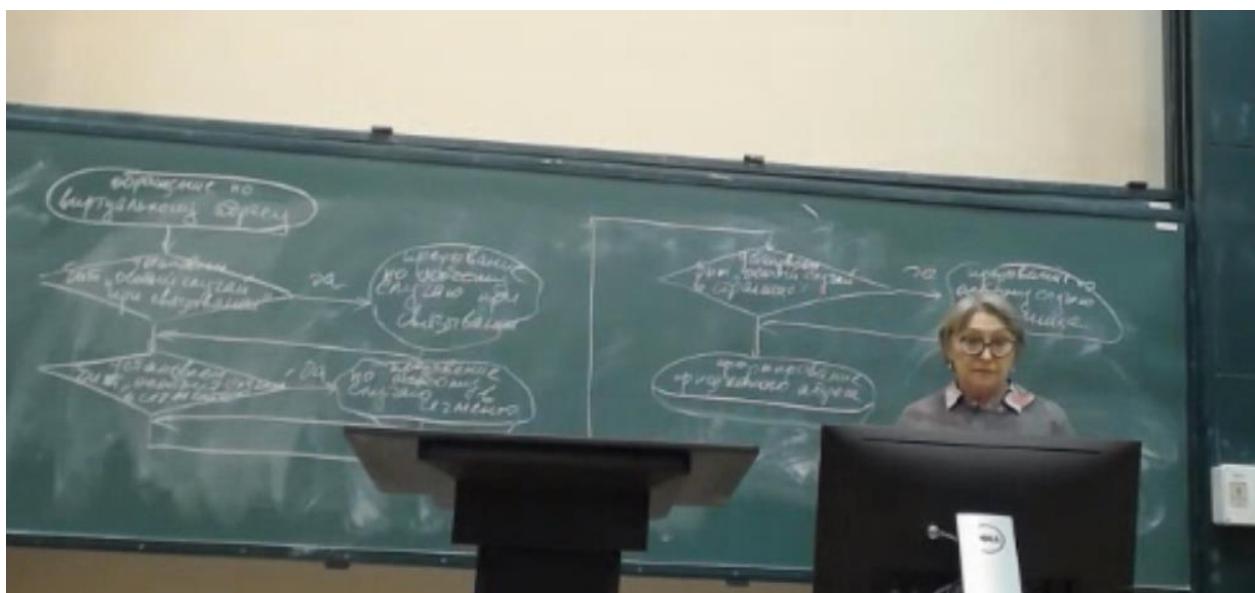
В процессоре должен быть адрес начала таблицы сегментов.

Виртуальный адрес делится на 2 поля - сегмент-смещение. Смещение проверяется на выход за границы. Если проверка прошла – обращение к дескриптору страницы в сегменте, а смещение распадается на номер страницы и смещение в странице. К начальному адресу страницы добавляется смещение и получается линейный физический адрес.

Поле смещение в дескрипторе всегда используется под флаги. Существует Present, Access (обращения), Dirty (изменения).

Вроде бы мы совместили только достоинства двух методов управления виртуальной памятью: сегментацией и страничным управлением. Сегментно-страничное преобразование было реализовано в IBM370.

Рассмотрим возникающие прерывания (особые случаи, которые возможны при таком управлении памятью) и пролонгируем на используемые в intel схемы.





Выполняется обращение по виртуальному адресу.

1 проверка – установлен бит «особый случай при связывании»? Если да, то возникает прерывание по особому случаю при связывании

2 проверка – установлен бит «особый случай» в сегменте? Если да, то возникает прерывание по особому случаю в сегменте

3 проверка - установлен бит «особый случай» в странице? Если да, то возникает прерывание по особому случаю в странице

После обработки всех прерываний формируется физический адрес.

Прерывания будут возникать в другом порядке.

Сначала выяснится, что страница не загружена в физическую память. Затем – что не существует таблицы страниц для данного сегмента (то есть необходимо ее создать). После обнаружится, что и данный сегмент не представлен в таблице сегментов (отложенное связывание, формирование дескриптора сегмента выполняется только при обращении к определенной библиотеке, а до этого он не описан в таблице сегментов. Пример – dll.) Необходимо создать строку в таблице сегментов, таблицу страниц, после этого только можно будет загрузить нужную страницу в физическую память.

Дейтал, ОС. Тема – управление памятью

Сейчас – только страницами по запросу, так как copy on write решил проблему коллективного использования.

И в стандартной, и в РАЕ: обработка страничного прерывания связана с дополнительными прерываниями в системе (не созданы еще и таблицы страниц)

Первый особый случай, который возможен(первая неудача) - особый случай при связывании. Речь идёт об отложенном связывании. Объёмы программ увеличиваются, увеличиваются задачи которые решаются с помощью соответствующего программного обеспечения и соответственно прикладное ПО отвечает, как принято говорить у политиков, на вызовы времени. В частности таким ответом являются dl - динамически подгружаемые библиотеки. То есть они появились не просто так. Они подгружаются автоматически когда в них появляется необходимость. Вот оно вам отложенное связывание. (Имеется ввиду link). Этот особый случай конечно как-то обозначается, может быть установлен специальный бит(особый случай при связывании). Тогда возникает прерывание по особому случаю при связывании. В результате обработки этого прерывания нужно сегменту с символическим именем присваивается номер строки таблицы сегментов и заполняется дескриптор сегмента.

Второй особый случай - если установлен бит особый случай в сегменте. Это приводит к прерыванию по особому случаю в сегменте. При обработке этого прерывания для сегмента создаётся таблица страниц и карта файлов. При этом адрес таблицы страниц записывается в дескриптор сегмента.

Третий особый случай - установлен бит особый случай страницы. Это прерывание по особому случаю страницы - страница отсутствует в памяти.

В итоге обработки всех этих особых случаев будет сформирован линейный физический адрес. Очевидно что и в этой схеме возможно кеширование. Мы с вами рассматривали кеширование в страничном преобразовании. В системе MULTICS такой кеш TLB (Translation Lookaside Buffer). Такой кеш должен содержать адреса физических страниц к которым были последние обращения. Такой же кеш возможен и в этой схеме.

Самым простым если бегло смотреть все три способа управления виртуальной памятью (простым в плане замещения) является страничное преобразование при котором мы заменяем страницы на страницы. Но для страниц всегда было минусом коллективное использование программ. Понятно, что в системах многие пользователи заинтересованы в использовании одних и тех же программ. Создавать для каждого отдельную копию такой программы это крайне неэффективно. Кроме того отдельные процессы могут быть заинтересованы в использовании одних и тех же таблиц, одних и тех же наборов данных. Это всё относится к сфере коллективного использования. Очевидно, что сегмент являющийся единицей логического деления программ, позволяет разделить наши программы в простейшем случае на сегмент кода, сегмент данных, сегмент стека. Мы это можем увидеть в том же адресном пространстве Unix при желании. Но для того чтобы процесс мог обращаться к каким-то таблицам, каким-то наборам данных, понятно что эти наборы данных имеют какую-то логическую целостность, принадлежат какому-то большому массиву, какой-то большой таблице (это важное заключение которое надо довести до конца, но на сегодня мы с вами закончим

Взаимодействие 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.

Очевидно, что системы предоставляют соответствующие системные вызовы, для того чтобы процессы и потоки могли взаимодействовать. *IPC* (*Inter Process Communication*) - это общее название средств взаимодействия параллельных процессов.

Рассмотрим тему «Уровни наблюдения»

Все зависит от того, на каком уровне рассматриваем выполнение программ

Два способа: строго последовательное и параллельное выполнение.

(это рисунок рязановой)

Уровни наблюдения

I Последовательное выполнение процессов

p1



II Квази параллельное выполнение

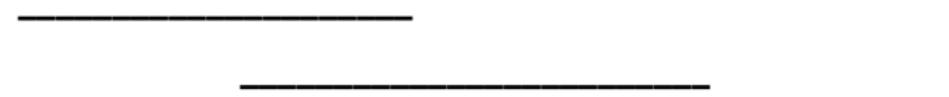
p1 p2 p1 p2 p1 p2 p1 p2 p1



III Реальная параллельность

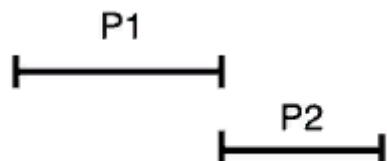
P1

P2



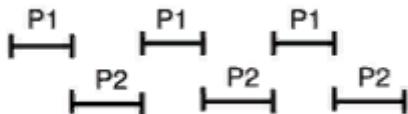
1. Последовательное выполнение.

Сначала выполняется код процесса p1, все ресурсы системы принадлежат ему. (Пример – msdos- однопрограммная ОС.) Когда p1 завершился, можно начать выполнять второй. От начала до конца выполняется процесс P1, потом от начала до конца выполняется процесс P2.

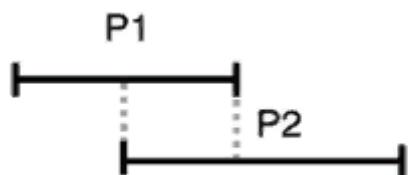


2. Квазипараллельное выполнение.

В системах разделения времени процессам в очереди выделяется только квант процессорного времени. Выполняется p1, затем время передается другому процессу. Потом этот квант получает опять p1 и тд. Для пользователя такое выполнение кажется параллельным. Квантование обеспечивает каждому пользователю иллюзию использования компьютера индивидуально. Квазипараллельное (почти параллельное). На уровне команд это последовательное выполнение. Это связано с переключением контекста



3. Реально параллельное. Выполнение процессов совпадает по времени на уровне команд. Но процессоров обычно меньше, чем процессов. Поэтому процессорное время квантуется и процессы выстраиваются в очередь за процессорным временем.



В любом случае (и 2 и 3) возникают общие проблемы, связанные с разделением ресурсов системы (использование одних и тех же данных параллельными процессами)

Бытовой пример:

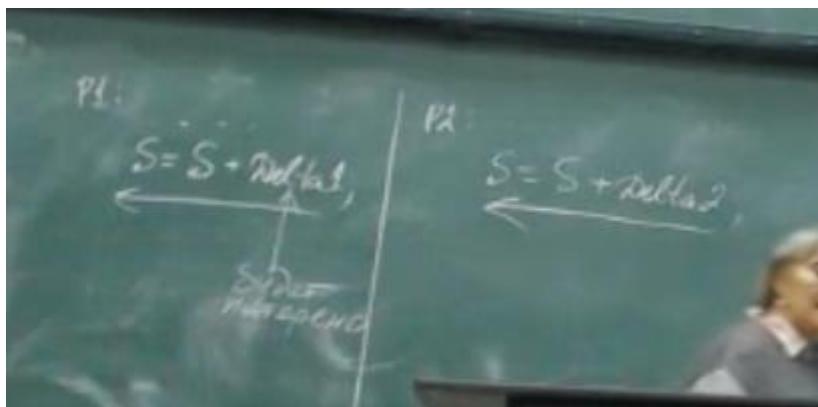
Пусть имеется 2 филиала. Переводят выручку на один и тот же расчетный счет. Это переменная s, к которой необходимо добавить выручку каждого филиала. Для параллельного выполнения пишут так: что-то делалось для первого филиала, потом суммирование суммы с дельтой. Параллельно другой филиал тоже считает и суммирует.

Команда – последовательность ассемблерных команд

Инициатива у 2 филиала. Отправляет свою выручку на РС: второй операнд, первый операнд, сложение, в аккумуляторе – сумма, но в память она не отправлена.

Теперь то же делает первый филиал, но записал

А теперь опять второй, но он уже не будет смотреть на предыдущую сумму. ТАДАМ



Оба процесса выполняют одну и ту же последовательность команд. Р1 выполняется на процессоре 1, Р2 - на процессоре 2.

Р1:

move eax, myvar

Inc eax

Mov myvar, eax

Р2 (то же):

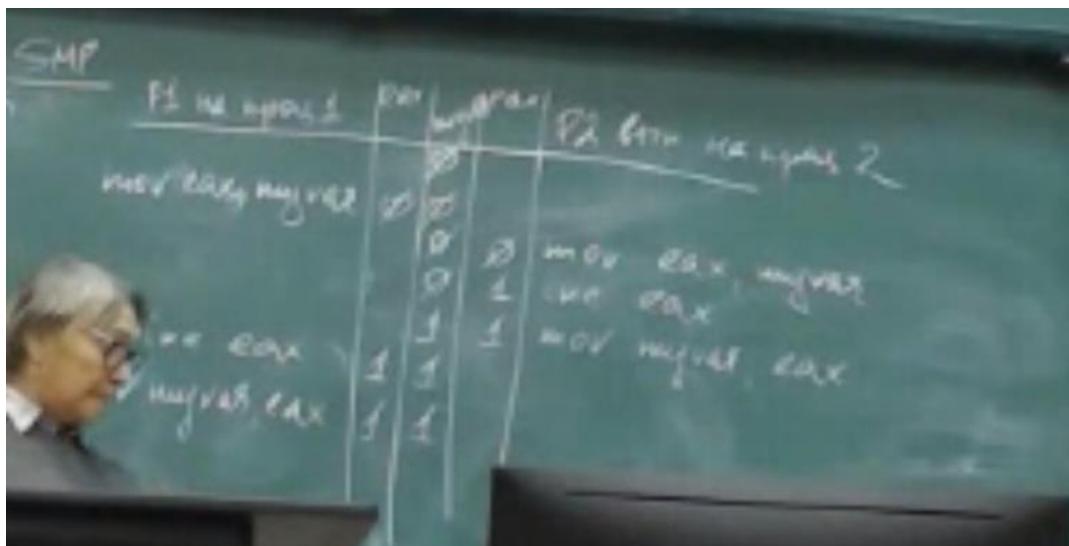
move eax, myvar

Inc 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. В СМП архитектуре память общая

В начальный момент значение переменной myvar = 0. Сначала квант получил Р1 и выполняет "mov eax, myvar", соответственно в eax попадает 0. Потом р2 – успевает сделать все. Потом р1- доделывает. А хотели в myvar 2. А получили 1. **Потерили данные**



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	-

(это рисунки рязановой)

Реальная параллельность

p1:

mov eax, myvar

inc eax

mov myvar, eax

процесс p1

p2: ...

mov eax, myvar

inc eax

mov myvar, eax

процесс p2

у каждого процессора свой eax

mov eax, myvar | 0 | 0 | | |

p1 потерял квант, а p2 квант получил

| 0 | 0 | | | mov eax, myvar

| | 0 | 1 | inc eax

| | 1 | 1 | mov myvar,eax

P1 получил квант и был восстановлен его контекст

inc eax | 1 | 1 | 1 |

mov myvar, eax | 1 | 1 | 1 |

.....

Квази параллельность

процесс p1

| eax | myvar | eax | процесс p2

eax одного процессора

mov eax, myvar

| 0 | 0 | |

p1 потерял квант и был сохранен его контекст | p2 квант получил

| | 0 | 0 | mov eax, myvar

| | 0 | 1 | inc eax

| | 1 | 1 | mov myvar, eax

P1 получил квант и был восстановлен его контекст

inc eax

| 1 | 1 | 1 |

mov myvar, eax

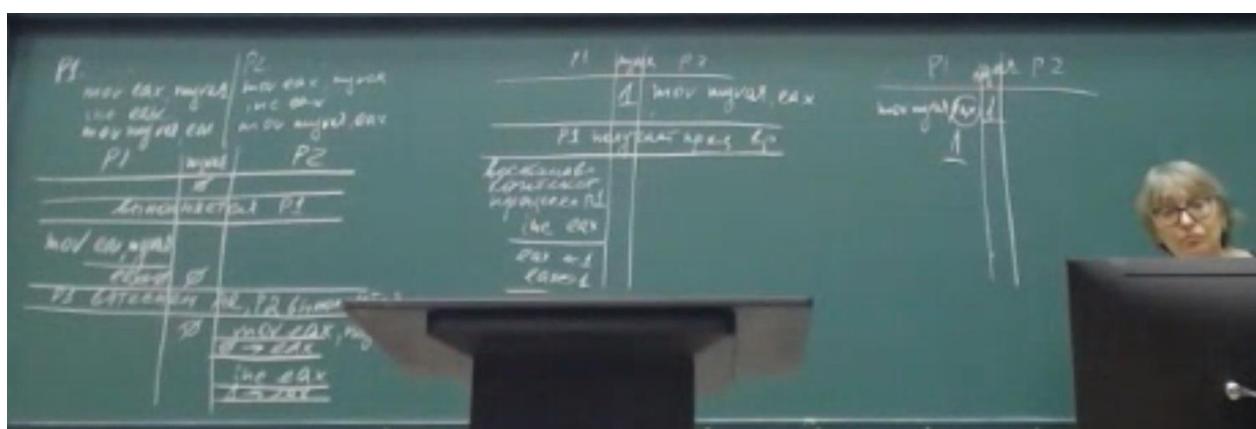
| 1 | 1 | 1 |

....

Лекция 10

Проблемы одинаковые при реальной и псевдопараллельности

Тот же пример.



Снова вывод – неважно, реальная или псевдо параллельность (в системах процессов всегда больше, чем процессоров, особенно в системах общего назначения)

Участок кода, в котором параллельные процессы обращаются к разделяемым переменным называется критическим (критическая секция). Myvar-разделяемая переменная. Проблема решается с помощью монопольного доступа процесса к разделяемым переменным: если один процесс зашел в свою критическую секцию по какой-то переменной, то другой процесс не может

зайти в свою критическую секцию по этой же переменной. Это обеспечивается методами взаимоисключения (не допустит доступ другого процесса как бы)

Методы взаимоисключения:

1. Программные
2. Аппаратный
3. С помощью семафоров (выделяется, потому что решил ряд проблем)
4. С помощью мониторов

Программный способ

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.

Example1:

Flag1, flag2: logical;

P1: while (1)

{

 While (flag2);

 Flag1 = 1;

 CR1;

 Flag1 = 0;

 PR1;

}

P2: while (1)

{

 While (flag1);

 Flag2 = 1;

 CR2;

 Flag2 = 0;

 PR2;

}

//начальные установки

```
Flag1 = 0;  
Flag2 = 0;  
(фото рязановой)
```

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.

Флаги не являются защищой критической секции, это не работает.

Предложение студента: глобальный флаг. Но это не потоки, а процессы. Общий флаг тоже разделяется.

Предложение: установить влаг до while

Example2:

Flag1, flag2: logical;

P1: while (1)

{

Flag1 = 1;

While (flag2);

CR1;

Flag1 = 0;

PR1;

}

P2: while (1)

{

Flag2 = 1;

While (flag1);

CR2;

Flag2 = 0;

PR2;

}

//начальные установки

Flag1 = 0;

Flag2 = 0;

(от рязановой)

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.

Программное решение Декера (а следом – паттерсона)

Декер – голландский математик, решил только для 2 параллельных процессов.

2 флага и 3 переменная – чья очередь

<pre> program DEKKER flag1, flag2: logical, que: int, P1: while(1) { flag1 = 1, while(flag2 == 1) { </pre>	<pre> if (que == 2) flag1 = 0, while(que == 2), flag1 = 1, } } CR1 flag1 = 0 que = 2 PR1; </pre>
--	--

<pre> P2: while(1) { flag2 = 1, while(flag1 == 1) { if (que == 1) </pre>	<pre> flag2 = 0; while(que == 1); flag1 = 1, } } CR2 flag2 = 0; que = 1; PR2; </pre>	<pre> // garanture garantiken flag1 = 0; flag2 = 0; que = 1; </pre>
--	--	---

program
 P1, P2,
 parbegin
 parend;

Program dekker,

Flag1, flag2: logical,
 Qur: int,
 P1: while(1)
 {
 Flag1 = 1,
 While (flag2 == 1)
 {
 If (qur == 2)
 {
 Flag1 = 0,
 }
 }
 }

```
    While (qur == 2);
    Flag1 = 1,
}
}//while
CR1,
Flag1 = 0;
Qur = 2;
PR1;
}

P2: while(1)
{
    Flag2 = 1,
    While (flag1 == 1)
    {
        If (qur == 1)
        {
            Flag2 = 0,
            While (qur == 1);
            Flag2 = 1,
        }
    }
}//while
CR2,
Flag2 = 0;
Qur = 1;
PR2;
}

//начальные условия
Flag1 = 0;
Flag2 = 0;
Qur = 1;
}
Parbegin
```

P1, p2,

Parend

(par или pas, que или qur)

Пусть инициатива у 2 процесса.

П2, устанавливает свой флаг, если флаг первого занят, проверяет, чья очередь. Первого – сбрасывает свой влаг 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.

Алгоритм петтерсона решает теми же средствами с теми же проблемами.

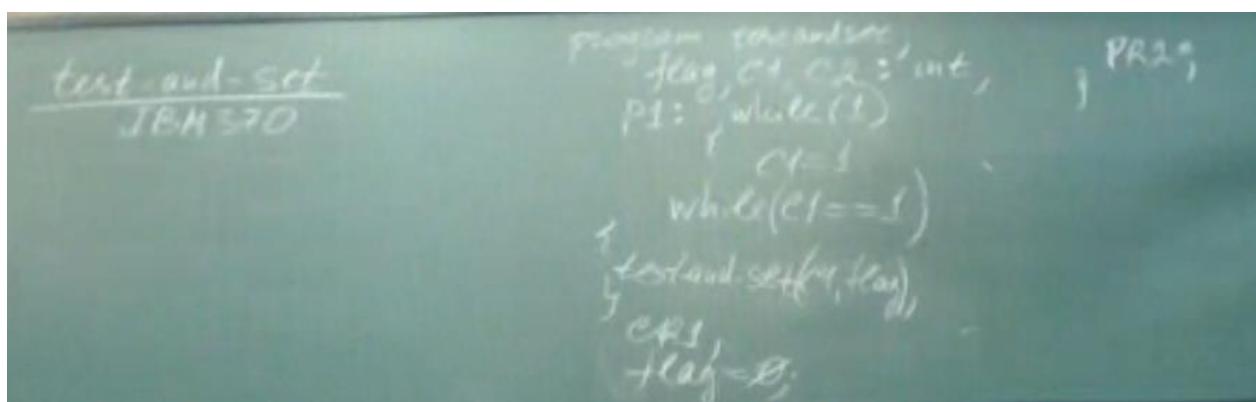
Аппаратная реализация

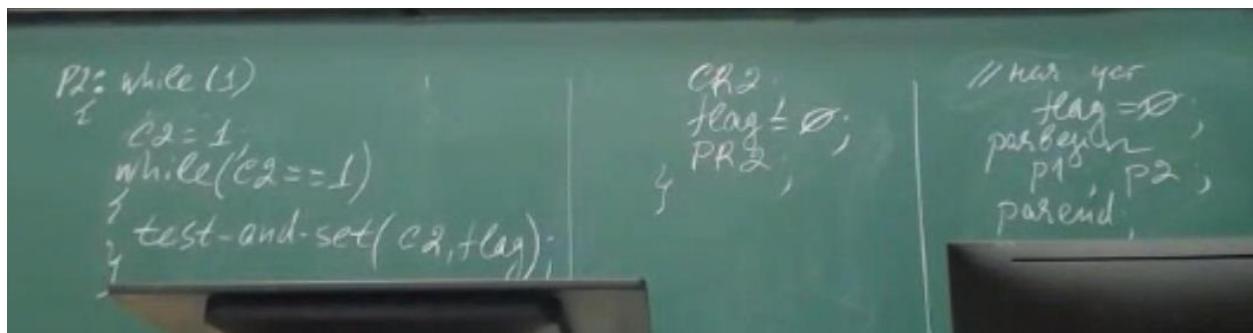
Связана с командами test-and-set – эти наборы команд есть в любой ОС

Существует чисто программный способ решения проблемы параллельных процессов, но позже

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.





Program testandset;

```
Flag, cq, c2: int; // можно и logical  
P1: while (1)  
{  
    C1 = 1;  
    While (c1 == 1)  
    {  
        Testsandset(c1, flag);  
    }  
    Cr1;  
    Flag = 0;  
    Pr1;  
}  
P2: while (1)  
{  
    C2 = 1;  
    While (c2 == 1)  
    {  
        Testsandset(c2, flag);  
    }  
    Cr2;  
    Flag = 0;  
    Pr2;  
}  
// начальные установки  
Flag = 0
```

Логическая переменная flag истинна, когда любой из процессов находится в своем критическом участке, и 0 – в противном случае.

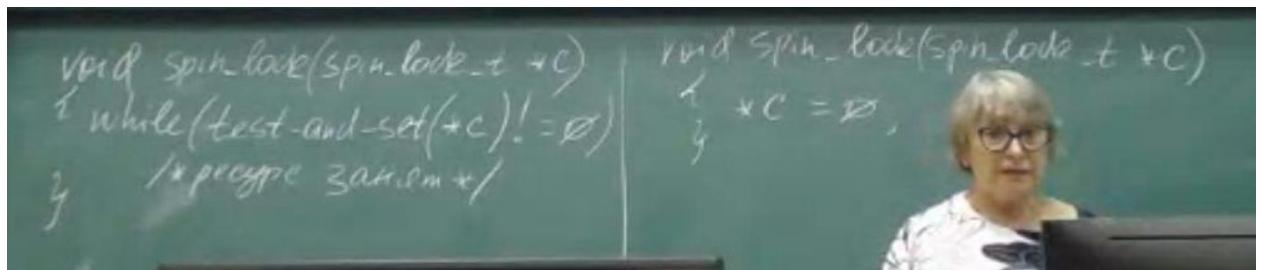
Пусть П1 хочет войти в критический участок, когда П2 в своем КУ по той же переменной.

Устанавливает 1 и входит в цикл проверки командой testandset. Поскольку П2 находится в своем КУ, флаг=1. Тест-сет обнаруживает это и устанавливает с1=1. П1 находится в цикле проверки, пока П2 не выйдет из своего КУ.

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.

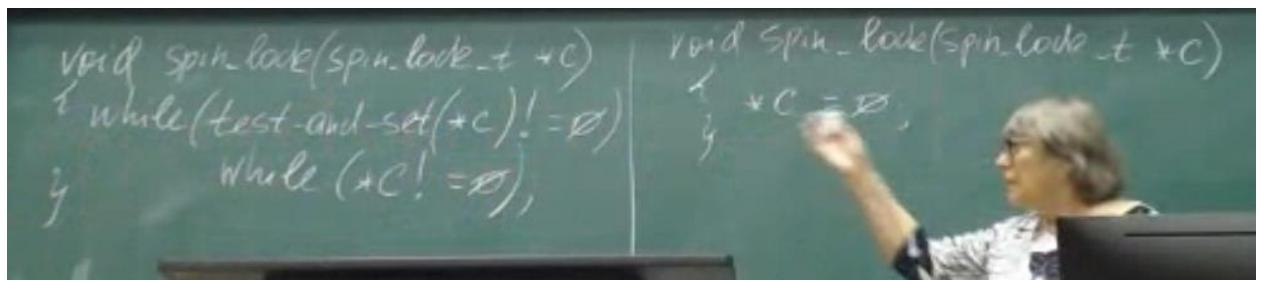


```
Void spin_lock(spin_lock_t *c)
{
    While (test_and_set(*c) != 0)
        //ресурс занят
}
```

```
Void spin_lock(spin_lock_t *c)
{
    *c = 0;
}
```

C-conditional variable-переменная типа условие

Что значит «неделимая», как это реализуется. Связана с блокировкой шины памяти – не может выполняться ни одна команда, в которой необходимо обратиться к памяти, а это крайне негативный (нежелательный) фактор. Поэтому переписывают так: - без test_and_set



```
Void spin_lock(spin_lock_t *c)
{
    While (*c != 0)
        //ресурс занят
}
```

Лекция 11 (от Р)

Взаимодействие параллельных процессов

Лекции: Рязанова Н.Ю. Часть 1.

1. Проблемы взаимодействия параллельных процессов

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. Обеспечение взаимодействия асинхронных процессов необходимыми средствами является одной из задач операционных систем.

Принципы и проблемы параллелизма

Параллелизм – может быть виртуальным или квази- - это чередование процессов во времени, чтобы создать видимость одновременного выполнения на одном процессоре. Такой параллелизм отличается от параллелизма, который предполагает подлинное одновременное выполнение на нескольких процессорах. Это особенно ясно видно, если рассмотреть не уровень наблюдения пользователя, а нижний уровень – выполнение команд программного кода.

На уровне команд:

1. Последовательное выполнение процессов:



Рис.1 Сначала выполняется процесс p1 команда за командой, затем p2

2. Квази параллельное выполнение процессов (один процессор):

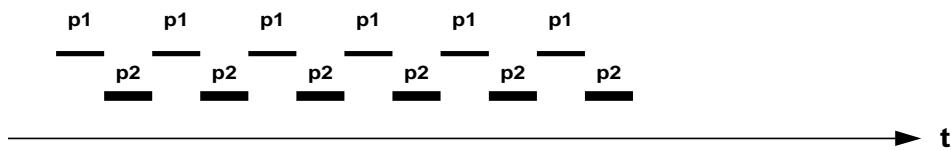


Рис.2 Определенный интервал времени выполняется процесс p1, затем выделенный интервал времени выполняется процесс p2 на одном и том же процессоре

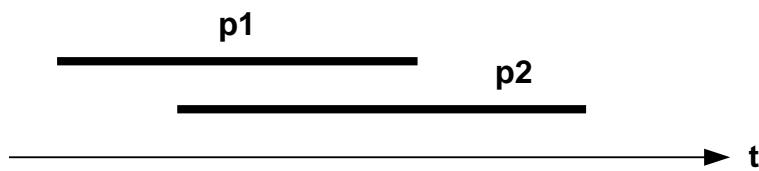


Рис.3 Реальная параллельность или перекрытие (несколько процессоров):
команды процессов p1 и p2 выполняются одновременно на разных
процессорах

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 SMP¹ 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.

Эти же проблемы возникают и в **многопоточных** приложениях.

Примеры взаимодействия параллельных асинхронных процессов

Рассмотрим следующую ситуацию. 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 вычислениях рис 4.1.

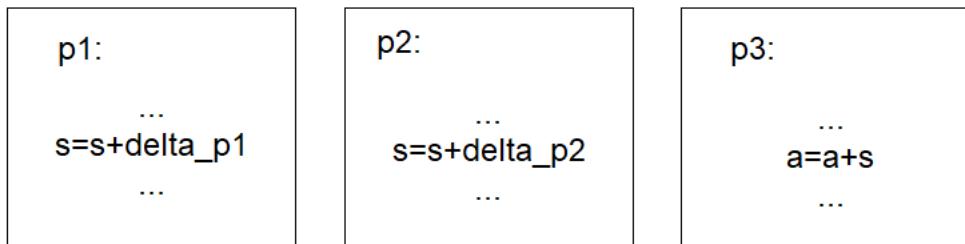


Рис.4

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 **p2**. Операция изменения значения **S** не является неделимой. Более того она требует выполнения целого ряда машинных команд. Поэтому вполне вероятно, что процесс обратившись к переменной **delta_p2** и сложив это значение с **S** будет 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 **p2** квант процессорного времени закончиться. Затем процессор получит процесс **p1**. 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

¹ Термин симметричная многопроцессорность (symmetric multiprocessing – SMP) относится как к архитектуре компьютера, так и к особенностям работы ОС и означает, что все процессоры системы используют одну и ту же память, одни и те же устройства ввода-вывода и могут выполнять одни и те же функции.

связана с асинхронностью процессов. Третий процесс может обратиться к переменной S до того, как и процесс p1 и процесс p2 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. Один процесс принимает и обрабатывает приходящие запросы, включая проверку их правильности, и передает запросы процессу-исполнителю для дальнейшей обработки. Процесс-исполнитель выполняет поиск и выборку информации, отвечающей запросу, и передает ее процессу, выполнившему запрос. Еще один процесс может отвечать за конечный результат – собственно за резервирование, связанное с редактированием базы данных мест на рейсы. Если процессы будут выполняться, как описано в предыдущем примере, то очевидным результатом будет продажа билетов на одно место на один и тот же рейс. Сценариев такого выполнения процессов множество. Предугадать поведение процессов практически невозможно, так как процессы выполняются с собственными скоростями, т.е. асинхронно.

Как было сказано ранее, безразлично выполняются ли процессы реально параллельно, т.е. одновременно на разных процессорах или квазипараллельно на одном процессоре. Важно, что, теряя процессор по любой причине (или в результате исчерпания кванта, или в результате вытеснения более приоритетным процессом), процесс продолжает свое выполнение с команды, которую не успел выполнить. Это делается путем восстановления контекста прерванного процесса (см. понятие контекста) и, в частности, аппаратного контекста (содержимого регистров процессора при выполнении процесса). В состав аппаратного контекста входит счетчик команд, содержащий адрес следующей команды, которая должна выполняться, т.е. адрес команда, которая не была выполнена.

2. Монопольный доступ и взаимоисключение

Переменные, к которым пытаются получить доступ параллельные процессы, такие как переменные S и in, 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**². Обеспечение взаимоисключения является одной из важнейших проблем взаимодействия параллельных процессов. Существует много способов решения этой проблемы как программные, так и аппаратные, как низкоуровневые, так и высокоуровневые.

Методы организации взаимоисключения делятся на:

- программные;
- аппаратные;
- с помощью семафоров;

² Взаимное исключение от англ. mutual exclusion.

- с помощью мониторов.

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**.

2.1 Программный способ реализации взаимного исключения

Простейший способ организации взаимного исключения основан на использовании переменных, играющих роль сигнальных флагов. Значение такой переменной, обычно логического типа, показывает, что процесс находится в критическом участке.

Рассмотрим взаимное исключение с помощью флагов на примере псевдкода двух параллельных процессов (листинг 1).

```
Program example1;
    flag1, flag2: logical;
P1: //первый процесс
    While (true) do
        begin
            While (flag2) do; //цикл активного ожидания
            flag1 = true;
            //критический участок первого процесса
            flag1 = false;
            //другие операторы процесса
        end;
    end; //p1
P2: //второй процесс
    While (true) do
        begin
            While (flag1) do; //цикл активного ожидания
            flag2 = true;
            //критический участок второго процесса
            flag2 = false;
            //другие операторы процесса
        end;
    end; //p2
    begin
        flag1 = 0; flag2 = 0;
    parbegin *)
        P1; P2;
    parend;
    end.
```

Листинг 1

^{*)} parbegin показывает начало, а parend – конец параллельного выполнения.

Данное решение не гарантирует монопольного доступа процессов к разделяемой переменной.

Убедимся в этом: процессы P1 и P2 выполняются параллельно и могут одновременно начать выполнять последовательность операторов в критической секции. Пусть первым в критическую секцию пытается войти процесс P1. Он входит в цикл проверки флага второго процесса и обнаруживает, что флаг не установлен. Проверив флаг, теряет квант. Процесс P2, получив квант, проверяет флаг первого процесса и обнаруживает, что флаг первого процесса не установлен. После этого он установит свой флаг и войдет в свой критический участок и, например, изменит значение переменной. Когда первый процесс снова получит квант, он продолжит свое выполнение с команды, на которой он был прерван, т.е. установит свой флаг и войдет в свой критический участок, где изменит начальное значение этой же переменной. Флаг не обеспечил монопольный доступ процессов к разделяемой переменной. Таким образом, флаги в данной реализации не гарантируют взаимоисключение.

Связано это с тем, что между моментами, когда процесс в цикле ожидания определяет, что он может продолжать выполнение, и установкой флага, сигнализирующего, что процесс находится в своем критическом участке, проходит достаточно времени. В течение этого интервала времени второй процесс может успеть обратиться к переменной флагу, обнаружить, что флаг не установлен, и войти в свой критический участок. Кроме того, данное решение не исключает бесконечного откладывания. Действительно, из-за разных относительных скоростей выполнения процессов возможна ситуация, когда один из процессов, например первый, выйдя из своего критического участка снова захватит этот же критический участок, установив свой флаг *flag1*. Таким образом, второй процесс не сможет войти в критический участок и будет находиться в состоянии бесконечного ожидания.

Рассмотрим другой вариант использования флагов. Пусть процессы устанавливают свои флаги до входа в цикл ожидания (листинг 2).

```
Program example2;
  flag1, flag2: logical;
P1: //первый процесс
  While (true) do
    begin
      flag1 = true;
      While (flag2) do; //цикл ожидания
      //критический участок первого процесса
      flag1 = false;
      //другие операторы процесса
    end;
  end;//p1
P2: //второй процесс
  While (true) do
    begin
      flag2 = true;
      While (flag1) do; //цикл ожидания
      //критический участок второго процесса
      flag2 = false;
      //другие операторы процесса
    end;
  end;//p2
begin
  flag1 = 0; flag2 = 0;
  parbegin
    P1; P2;
  parend;
end.
```

Листинг 2

Рассмотрим ситуацию, когда сначала квант получит процесс P2, установит свой флаг (flag2=1) и потеряет квант. Процесс P1 перехватывает инициативу и тоже устанавливает свой флаг (flag1=1) и входит в цикл проверки флага второго процесса. Поскольку flag2 установлен, процесс P1 будет проверять его в цикле весь выделенный ему квант. Затем второй процесс получает возможность войти в свой цикл проверки флага первого процесса (flag1) и поскольку флаг установлен будет выполнять цикла проверки флага в течение всего кванта. Другими словами, процессы входят в свои циклы ожидания. Процесс P1 ждет, когда процесс P2 сбросит свой флаг, а процесс P2 ждет сброса флага первого процесса. Оба процесса не могут продолжить выполнение, так как ждут освобождения ресурса, занятого другим процессом, и не могут продолжить свое выполнение. Это пример **тупиковой ситуации или взаимоблокировки** (dead lock).

Следующий вариант программы (листинг 3) гарантирует взаимоисключение и отсутствие тупика.

```
Program example3;
  flag1, flag2: logical;
P1: //первый процесс
  While (true) do
    begin
      flag1 = true;
      While (flag2) do
        begin
          flag1 = false;
          задержка;
          flag1 = true;
        end;
      //критический участок первого процесса
      flag1 = false;
      //другие операторы процесса
    end;
  end;//p1
P2: //второй процесс
  While (true) do
    begin
      flag2 = true;
      While (flag1) do
        begin
          flag2 = false;
          задержка;
          flag2 = true;
        end;
      //критический участок второго процесса
      flag2 = false;
      //другие операторы процесса
    end;
  end;//p2
begin
  flag1 = 0; flag2 = 0;
  parbegin
    P1,P2;
  parend;
end.
```

Листинг 3

Однако, и в этом случае также возможны негативные ситуации. Не имея никаких сведений об относительных скоростях асинхронных параллельных процессов, необходимо учитывать любые возможные ситуации при их выполнении. Например, процессы устанавливают значение своих флагов в истину, затем по порядку проверяют при входе в тело цикла значение флага другого процесса и входят в тело цикла, где сбрасывают свой флаг. Такую последовательность процессы могут выполнять снова и снова. Такой режим выполнения процессов можно назвать «зависанием» (starvation), когда процесс постоянно проверяет, устанавливает и изменяет значение одних и тех же переменных, не выполняя при этом никакой полезной работы. Очевидно, что такой режим выполнения процессов маловероятен. Однако он принципиально возможен и, следовательно, его нельзя игнорировать.

Корректное программное решение проблемы взаимоисключения предложил голландский математик **Деккер**. Данное решение исключает «зависание», бесконечное откладывание и взаимоблокировку за счет введения дополнительной переменной – очередь (queue), определяющей очередность вхождения процессов в критический участок (листинг 4).

```
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.

```

Листинг 4

В самом деле, пусть первый процесс пытается войти в свой критический участок. Он устанавливает свой флаг и проверяет флаг второго процесса. Если flag2 сброшен (ложь), то первый процесс пропускает тело цикла while и входит в свой критический участок. Иначе, если флаг второго процесса введен, то первый процесс входит в тело цикла, сбрасывает свой флаг и проверяет очередь права входления в критический участок. Если установлена очередь второго процесса, то первый процесс входит в цикл ожидания, в котором выполняет проверку переменной queue до тех пор, пока второй процесс не выйдет из своего критического участка и не установит значение queue равным 1. Выйдя из цикла ожидания, первый процесс установит свой флаг в 1 и войдет в свой критический участок. Таким образом каждый процесс, выходя из критической секции кода программы, устанавливает, что следующим должен войти другой процесс.

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.

2.2 Аппаратные средства взаимоисключения

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.

Рассмотрим пример использования команды test-and-set (листинг 5).

```

Program example-test-and-set;
active: logical;
P1: //первый процесс
flag1: logical; //локальная переменная
While (true) do
    begin
        flag1 = true;
        While (nflag1) 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. Пусть первый процесс пытается войти в свой критический участок. Он устанавливает свой флаг в 1 и в цикле выполняет команду TS. Если второй процесс находится вне своей критической секции, то переменная active имеет значение «ложь». Команда TS запишет это значение в переменную flag1 и установит для active значение «истина». При очередной проверке условия выхода из цикла будет получено значение «ложь», процесс выйдет из цикла проверки и войдет в свой критический участок. Второй процесс уже не сможет войти в свой критический участок, так как переменная active имеет значение «истина». Когда первый процесс выйдет из своего критического участка, он сбросит переменную active в 0 («ложь») и второй процесс сможет выйти из цикла ожидания, если он в нем находится, и, установив active, войдет в свой критический участок.

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.

Простая блокировка

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 mutex³).

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)

```

³ Название simple mutex может привести к путанице и смешению понятий, потому что spin lock не следует путать с мьютексом (mutex)

```

{
    *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 **c** выполняется без захвата шины. Команда `spin_lock()` базируется на команде `test-and-set` и является макрокомандой.

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.

Системные процессы или процессы, выполняемые в режиме ядра (*kernel mode*) так же нуждаются в механизмах взаимоисключения. Критическими секциями кода ядра являются участки кода, изменяющие глобальные структуры данных, например, базу данных диспетчера ядра или очереди отложенных вызовов процедуры диспетчеризации (DPC – Deferred Procedure Call). Основную проблему для обеспечения корректного доступа к данным ядра создают прерывания. Прерывание может произойти в любой момент. Например, в момент обновления супервизором базы данных ядро может произойти прерывание с целью изменения этой же базы. В Windows 2000 используется механизм маскирования прерывания путем повышения IRQL (Interrupt Request Level – уровень запроса прерывания) процессора до самого высокого уровня. Однако в многопроцессорных системах этот механизм не работает, так как повышение IRQL на одном процессоре не влияет на уровень прерываний на другом процессоре.

Для реализации взаимоисключения в многопроцессорных системах используется механизм спин-блокировки. Во многих архитектурах спин-блокировки используют неделимую команду `test-and-set`. Перед входом в критическую секцию в коде ядра устанавливается спин-блокировка. Если критическая секция занята, то ядро «крутиится» в цикле проверки пока критическая секция не освободится.

В однопроцессорных версиях системы для установки и снятия спин-блокировок в соответствующих функциях используется изменение уровня запроса прерывания (IRQL), для этого HAL (Hardware Abstraction Layer – загружаемый модуль режима ядра Hal.dll) просто повышает и понижает IRQL.

В Windows 2000 используется спин-блокировка с очередью (queued spinlock). Такая блокировка доступна только супервизору и работает следующим образом: если процессор пытается установить спин-блокировку, которая в данный момент занята, то его идентификатор ставится в очередь (FIFO) к данной спин-блокировке. Освобождая спин-блокировку процессор передает ее тому процессору, который стоит в очереди первым. Процессор, ожидающий спин-блокировку проверяет состояние не самой спин-блокировки, а флаг стоящего перед ним в очереди процессора. В системе имеется не более шести спин-блокировок, защищающих основные структуры данных ядра.

Ядро предоставляет доступ к спин-блокировкам и другим компонентам исполнительной системы, например драйверам устройств, через функции ядра системы: `KeAcquireSpinLock` и `KeReleaseSpinLock` [подробнее смотри в сол.]

Для рассмотренных алгоритмов характерно не только активное ожидание, но и **инверсия приоритетов**. Рассмотрим два процесса с разными приоритетами. Пусть процесс P1 имеет высокий приоритет H, а процесс P2 – более низкий приоритет L. Допустим, что в некоторый момент времени процесс P1 выполняет операцию ввода-вывода, а процесс P2 получил возможность войти в свою критическую секцию. Затем процесс P1 завершает операцию ввода-вывода и процессор немедленно передается в его распоряжение и P1 начинает выполнять цикл активного ожидания. Поскольку P1 имеет более высокий приоритет, процесс P2 не получит кванта процессорного времени и не сможет выйти из своего критического участка. Таким образом, P1 навсегда останется в цикле ожидания.

2.3 Семафоры

Эджеер Дейкстра (Dijkstra E.W.) в своих работах по взаимодействию параллельных процессов, первая из которых была опубликована в 1965 году, суммировал опыт предыдущих разработок и поднятые в них проблемы и предложил механизм, названный семафором, в качестве средства реализации взаимоисключения.

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.

Если семафор принимает только два значения, то он называется *бинарным* или *двоичным*. Семафор, принимающий значения от 0 до n , называется *считывающим*.

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. Блокировка процесса на семафоре и постановка его в очередь выполняется функциями ядра. 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.

Семафор можно представить как систему вращающихся дверей (рис.5). «Двери» устроены так, что повернуть их может только выходящий из второй двери. Промежуток между дверями – критическая секция. Когда процесс находится «между дверями», другой процесс попасть туда не может, так как входная дверь заблокирована. Выходя из критической секции, процесс как бы поворачивает обе двери и следующий процесс попадает в свою критическую секцию. Таким образом, процессы стоят в очереди к семафору пассивно – они заблокированы и, следовательно не занимают циклы процессорного времени. Семафоры исключают *активное ожидание на процессоре*, так как заблокированный процесс активизируется другим процессом, выполнившим операцию освобождения семафора («поворот двери»).

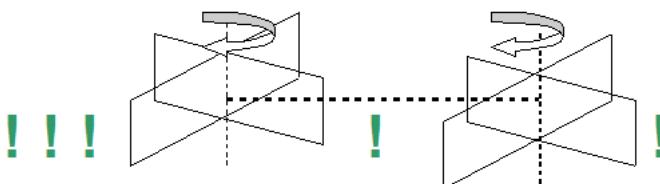


Рис.5

Рассмотрим пример использования бинарного семафора для реализации задачи взаимоисключения на псевдокоде (листинг 6).

```
Program example_sem;
S : 0..1;
P1 : //первый процесс
    While (true) do
        begin
        // другие операторы
```

^{*)} P – первая буква датского слова *passeren*, которое переводится как пропустить и V – первая буква слова *vrygeven*, переводимого как освободить.

```

P(S);
// критическая секция первого процесса
V(S);
// другие операторы
end;
end; // P1
P2 : //второй процесс
While (true) do
begin
// другие операторы
P(S);
// критическая секция второго процесса
V(S);
// другие операторы
end;
end; // P2
begin
S = 1;
parbegin
P1; P2;
parend;
end.

```

Листинг 6

Считывающие семафоры используются в тех случаях, когда имеется n единиц ресурса. Рассмотрим пример использования считающих семафоров, относящийся к задачам типа производство-потребление. Производитель помещает единицы продукции в хранилище, из которого их забирает потребитель. Емкость хранилища - n единиц. Очевидно, что если хранилище пусто, то взять из него нечего, и, если хранилище полностью заполнено, то производитель не может добавить в него ни одной единицы продукции. Для решения задачи используется два семафора с начальными значениями $S1 = n$ и $S2 = 0$. Решение, предложенное Дейкстра, выглядит следующим образом:

Производитель:

```

...
While (true) do
begin
Изготавливает единицу продукции
P(S1);
Помещает единицу в хранилище
V(S2);
end;
...

```

Потребитель:

```

...
While (true) do
begin
P(S2);
Берет единицу из хранилища
V(S1);
Использует единицу
end;

```

Рассмотрим следующую последовательность событий для двух единиц ресурса:

События	Производитель	Потребитель	S1	S2	Состояние процесса
0	2	0	
1	...	P(S2)	2	0	P2 заблокирован на
S2					
2	P(S1)	...	1	0	

3	V(S2)	...	1	1	освобождение P
P(S2)			{		
4	P(S1)	...	11	0	после выполнения P(S2)
заняты					все ячейки буфера
5	...	V(S1)	0	0	возвращение ячейки
в буфер					
6	V(S2)	...	1	1	
7	P(S1)	...	0	1	все ячейки буфера
заняты					
8	V(S2)	...	0	2	
9	P(S1)	...	0	2	P1 заблокирован на S1
10	...	P(S2)	0	1	
11	...	V(S1)	{	1	освобождение P
P(S1)				0	
...					

и так далее.

Таким образом, взаимоисключение, реализованное на считающих семафорах, работает.

Множественные семафоры

Множественные семафоры (наборы семафоров) обладают способностью проверки в одном примитиве сразу всех или нескольких семафоров набора.

Возможности множественных семафоров рассмотрим на примере задачи об обедающих философах, которая используется для демонстрации различных алгоритмов разделения ресурсов. 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 (рис.6). Таким образом вилок всего пять.

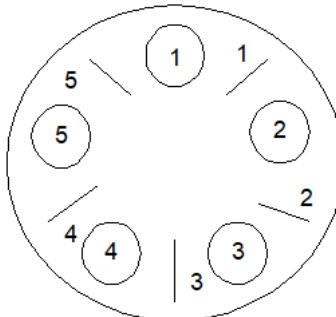


Рис.6

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).

```
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 следующими способами:

- Каждый из философов пытается взять сразу оба прибора и, если ему это удается, то он начинает есть. Поев, кладет обе вилки на стол.
- Философ берет правую вилку и, если ему это удается, то удерживая правую пытается взять левую вилку.
- Философ берет правую вилку и, если левую вилку взять не может, то кладет правую.

Данные три способа действия философов демонстрируют три негативные ситуации в системе:

«голодание» или бесконечное откладывание, тупик и захват и освобождение одних и тех же ресурсов.

Решение Э. Дейкстры задачи «Производство-потребление»

Постановка задачи: 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 (рис.7).

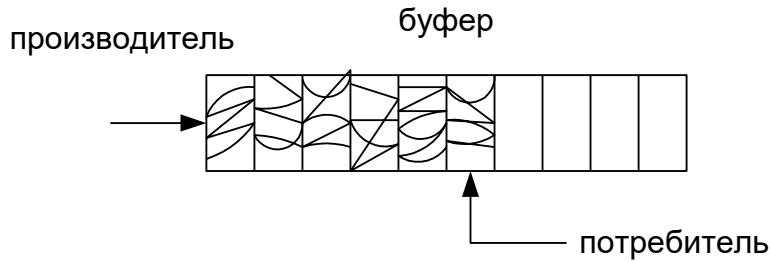


Рис.7

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 */
```

}

Поскольку потребитель удаляет элемент из буфера, значение «buffer_full» уменьшается на 1, если это возможно, иначе при нулевом значении семафора buffer_full потребитель блокируется, ожидая, когда производитель заполнит хотя бы одну ячейку буфера. Значение бинарного семафора bin_sem декрементируется, чтобы другой производитель или производитель не могли получить доступ к буферу в данный момент.

Проблемы использования семафоров

Основными проблемами при использовании семафоров являются «гонки» (race conditions) и взаимоблокировки. Использование одиночных семафоров может привести к взаимоблокировке в случае, если процесс использует несколько семафоров. Простейшим примером возможности попадания двух процессов в тупик является пример на рис.8:

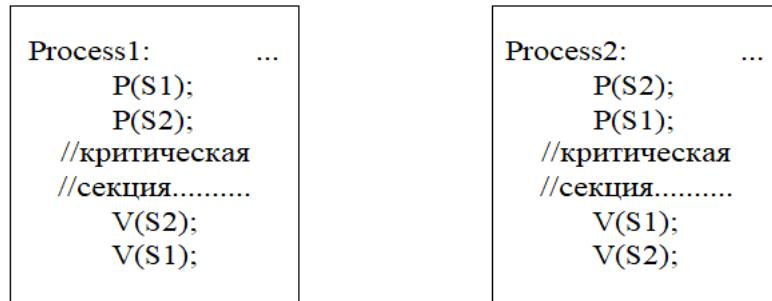


Рис.8

Первый процесс захватил семафор S1, а второй – семафор S2. Затем первому процессу для продолжения требуется семафор S2, занятый вторым процессом, а второму процессу нужен семафор S1, занятый первым процессом. Первый процесс заблокирован в ожидании освобождения S2 вторым процессом, а второй процесс заблокирован в ожидании освобождения семафора S1 первым процессом. Каждый из двух процессов заблокирован в ожидании освобождения ресурса, занятого вторым процессом. Другая проблема, возникает из-за наличия отдельных операций по изменению управляющих параметров набора семафоров и обращению к семафорам. Такой подход может привести к проблеме состязательности или «гонкам» (англ.- race condition) [2].

Лекция 11

Семафоры

Были предложены Дейкстрапа Dijkstra EW.

(Лампорт решил с помощью очереди)

Основной недостаток: активное ожидание на процессоре. Чтобы войти в CR используется цикл проверки (и процесс циклится в этом цикле) – этот неэффективное использование процессорного времени.

В 1965 – первая работа Дейкстры, связанная с семафорами.

Семафор-неотрицательная защищенная переменная, на которой определены 2 операции: P(S) и V(S) (P – пропустить, V-освободить)

Семафор (0, 1) – бинарный.

Если неотрицательное значение, то считающий. Ро – защищает переменную, так как ее можно изменить только 2 командами – Р и В (команды неделимы)

1. P(S) – декремент ($S=S-1$), если это возможно, то есть $S > 0$. Если декремент невозможен, тот процесс будет блокирован на семафоре S.
2. V(S) – инкремент $S=S+1$. S становится >0 (если был =0) и тем самым другой процесс может выполнить P(S)

Так активизируется процесс, который наход в очереди к семафору.

С точки зрения ОС P(S) и V(S) – системные вызовы, то есть их может выполнить только ядро системы.

Семафоры

Были предложены Дейкстрой. В 1965 была опубликована первая статья, где он выдвигал 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. Крайне неэффективное использование процессорного времени. И тогда Дейкстра предложил новое решение, которое и назвал семафором. В общем-то семафор не очень похож на этот семафор (который красный-жёлтый-зелёный). Скорее он похож на турникет, который толкает тот, кто выходит. То есть войти можно только если кто-то выходит. Мы будем использовать обозначения, которые предложил Дейкстра.

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 .

Заблокировать процесс и разблокировать процесс можно только на уровне ядра. То есть P(s) и V(s) – системные вызовы, которые переводят систему в режим ядра. В режиме ядра процесс будет заблокирован и когда это станет возможно V(s) переведёт систему в режим ядра и выполняя эту команду, система разблокирует первый процесс из очереди семафора. Сразу же здесь нужно сказать, что любой переход в режим ядра - это дополнительные расходы, это переключение

контекста. Мы избавились от активного ожидания, но заплатили за это переходом в режим ядра. У всех альтернативных решений в программировании всегда есть своя цена.

Понятно, что $P(s)$ и $V(s)$ это макрокоманды, это большие действия системы (там используются команды более низкого уровня), в основе которых лежит test-and-set. Пишем простейшее взаимоисключение с помощью семафоров:

```
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
```

Как мы видим, процесс разблокируется другим процессом.

Если семафор может принимать неотрицательные положительные значения, то такой семафор называется считывающим (это дословно). Рассмотрим решение Дейкстры задачи, которое получила имя "Производство-потребление". Это очень важная задача.

Задача Дейкстры "Производство-потребление"

Для этой задачи характерно наличие двух типов процессов - процессов-производителей, которые производят единицу информации и записывают её в буфер и процессов-потребителей, которые только потребляют информацию, выбирая её из буфера. Решение этой задачи с использованием трёх семафоров и было предложено Дейкстрой. Он предложил использовать два считающих семафора и один бинарный. Это задача одна из немногих в ОС, которая получила собственное название. Пишем:

se - семафор, который показывает, что буфер пуст (empty)

sf - семафор, который показывает, что буфер полон (full)

sb - бинарный семафор.

```
// consumer
while (1) {
    P(sf);
    P(sb);
    N = N - 1; // взять из буфера
    V(sb);
    V(sf);
    // вывести значение
    ...
}
```

```
// produser
while (1) {
    P(se);
    P(sb);
    N = N + 1; // добавить в буфер
    V(sb);
    V(sf);
```

...

}

```
// начальные установки  
se = N; // кол-во пустых ячеек = равно размеру буфера  
sf = 0; // кол-во заполненных  
sb = 1; // позволяет выполнять действия не блокируя...
```

Produser может положить в буфер единицу информации, если есть свободная ячейка. Если свободных нет, то producer будет блокирован на семафоре empty до тех пор пока consumer не освободит хотя бы одну ячейку. P(se) - consumer (producer же должен быть 57:20) может декрементировать семафор, если он не равен 0, то есть если есть пустые ячейки. Иначе он будет блокирован. Соответственно бинарный семафор защищает разделяемую переменную N. После того как produser смог положить в буфер единицу информации он инкрементирует семафор full.

Consumer может взять единицу информации из буфера, если семафор full не равен 0, то есть если есть заполненные ячейки. Бинарный семафор защищает переменную. Освободив ячейку буфера, потребитель инкрементирует семафор empty, то есть увеличивает кол-во пустых ячеек, тем самым он может . Это очень важное решение и на третьей ЛР по юникс будем его писать.

Множественные семафоры

По-другому - массив считающих семафоров. ОС, с которыми мы работаем поддерживают именно наборы считающих семафоров. Обладают очень важным свойством: одной неделимой операцией можно выполнить проверку всех или части семафоров набора. Это важно. "Это ж не спроста" - как говорил Винни-Пух. Потому что использование семафоров в программах, когда процессы одновременно захватывают и освобождают одни и те же семафоры, приводили к тупикам. Задание будет на экзе!!!: придумать пример тупика для двух бинарных семафоров (два процесса, два семафора). Ну а мы рассмотрим множественные семафоры на примере классической задачи, которая получила название "Обедающие философы". Версий много, но суть одна.

Обедающие философы

За круглым столом сидят 5 философов. Перед каждым из них тарелка, а справа и слева по одному прибору (для приёма пищи необходимо два прибора) (надо вставить картинку). Жизнь философа проста: он какое-то время размышляет, потом пытается поесть. Понятно, что приборы являются ресурсами. На самом деле существует три способа действия философа:

Каждый пытается одновременно взять обе вилки. Если удаётся взять приборы, философ некоторое время ест, а потом одновременно кладёт оба прибора.

Философ берёт правый прибор и удерживая его пытается взять левый.

Философ берёт правый, пытается взять левый. Если не удаётся - кладёт правый.

Это модель трёх негативных ситуаций в системе:

Модель бесконечного откладывания. Сколько философов умрёт от голода? Рязанова заявляет, что один и приводит простой пример - у одному из философов постоянно не достаётся, то правого, то левого.

Тупиковая ситуация. Если все философы взяли по правому прибору.

Захват и освобождение одних и тех же ресурсов.

Пишем код, который как раз демонстрирует св-во множественных семафоров:

```
forks: array[1..5] of semaphore;
```

```
left, right: 1...5;
```

```
//p1 - первый философ
```

```
left = (i + 1) mod 5;
```

```
right = i mod 5;
```

```
while(1) {
```

```
    /*размышляет*/
```

```
    p(forks[left], forks[right]); // неделимой операцией изменяется несколько семафоров
```

```
    /*ест*/
```

```
    v(forks[left], forks[right]);
```

```
}
```

```
...
```

```
//p5 - пятый
```

```
left = 4;
```

```
right = 5;
```

```
while (1) {
```

```
    /*размышляет*/
```

```
    p(forks[left], forks[right]);
```

```
    /*ест*/
```

```
    v(forks[left], forks[right]);
```

```
}
```

```
...
```

Понятно, что код у каждого философа аналогичный. Это первый случай - берут сразу по два прибора. Можно с mod, можно без mod.

Mutex

Семафоры привели к идее множественным семафором. Но использование семафоров всё равно черевато появлением сложного отлавливаемых тупиков.

Mutex (mutual exclusion) - это так же средство предоставляемое системой - системный вызов. О сравнении семафоров и мьютексов можно много говорить, есть такой материал "семафоры против мьютексов". Фактически мьютекс представляет из себя (ближе всего) бинарный семафор, но есть важнейшие отличия:

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.

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.

Процесс, захвативший мьютекс не может быть случайно удалён, завершён. А захвативший семафор может.

На самом деле все примитивы (в разных системах это могут быть разные название) можно назвать двумя словами: lock и unlock. Но это не касается семафоров!!! Семафоры это особенный договор. Всё касается средств близких к мьютексам. Это самое общее название. В зависимости от системы могут быть другие термины (wait-post, wait-signal).

Мониторы

Мониторы являются более высокоуровневыми средствами чем ранее перечисленные примитивы.

Лекция 12 от Р

2.4 Мониторы⁴

Монитор (monitor) – это программное средство, разработанное Хоаром и дающее возможность управляемого совместного использование ресурсов среди асинхронных процессов, включая возможность управляемого обмена параметрами между процессами. Идея монитора заключается в создании механизма, который унифицирует взаимодействие параллельных процессов по разделяемым данным и процедурам, которые обрабатывают эти данные.

Монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации динамического распределения конкретного ресурса. Синтаксически монитор

⁴ Монитор – это пример множественного значения [М.186]:

- устройство: а) видеотерминал и клавиатура как пульт управления; б) видеомонитор;
- другое название супервизора
- программное средство, разработанное Хоаром.

начинается ключевым словом monitor. Монитор защищает свои переменные. Доступ к переменным монитора можно получить, только используя процедуры монитора.

Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, вызывая процедуру монитора. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Если вызов был успешным, то процесс считается, находящимся в мониторе. Вход в монитор находится под жестким контролем — здесь осуществляется взаимоисключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессы, которые хотят войти в монитор, когда он уже занят, ставятся в очередь к монитору, причем режимом ожидания автоматически управляет сам монитор. Монитор сам является ресурсом.

При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждет. Проверка условия выполняется самим монитором, который и разблокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимоисключение процессов, отсутствуют серьезные проблемы, связанные с организацией параллельных взаимодействующих процессов. При первом обращении монитор присваивает своим переменным начальные значения. При каждом последующем обращении используются те значения переменных, которые сохранились от предыдущего обращения. Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдает команду ожидания с указанием условия ожидания. Процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится. Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса или, если уже имеются процессы, ожидающие освобождения данного ресурса, выполнить команду извещения (сигнализации), чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.

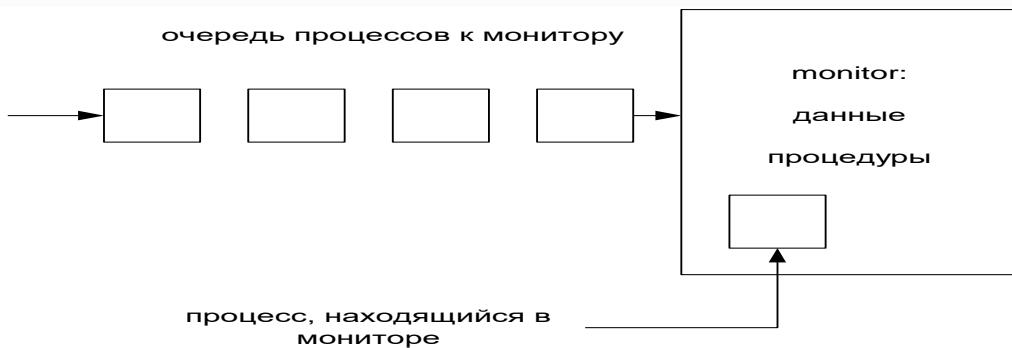


Рис.8

Монитор осуществляет доступ к разделяемым ресурсам посредством использования переменных типа условие (conditional). Будем такие переменные называть просто «условие». 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 (листинг 1).

```

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.

```

Листинг 1

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. Монитор представлен на листинге 2.

```

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.

```

Листинг 2

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).

Монитор «Читатели-писатели»

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.

```

Листинг 3

Когда число читателей равно 0, 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 start_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.

Лекция 12

Мониторы

Были предложены как альтернатива семафорам. Но семафоры – не единственные примитивы, предоставляемые пользователю операционной системой.

Монитор – набор процедур, переменных, может быть сложных структур данных, которые объединяются в единый объект ядра. Чтобы обратиться к данным монитора, надо обратиться к его функциям, что делает данные защищенными. Мониторы появились раньше классов. единая структура, содержащая поля данных и функции, которые могут обращаться к этим данным.

Простой монитор предоставляет 1 единственный ресурс многим процессам.

Кольцевой – решает задачу производства-потребления. Существует 2 типа процессов: производители – производят записи и помещают в хранилище (в вычислительных машинах - буфер), потребители – только выбирают данные из хранилища.

В этой задаче в этом мониторе есть поля данных – буфер полон, буфер пуст типа условие: позиции, с помощью которых производитель и потребитель обращаются, процедуры – произвести и потребить

Монитор читатели и писатели (редактора). 2 типа процессов: писатели – могут изменять поля структур данных, читатели – могут только читать эти поля.

Решение Хоара. Монитор Хоара читатели-писатели.

Псевдокод (пусть он и известен) подвергался различным изменениям (в книге современные ОС например). Нам дают классический, но в интерпретации Дейтала.

Напомним, что в мониторах используются 2 функции: wait - блокирует процесс в ожидании возможности обращения к разделяемым ресурсам, сигнал- сигнализирует, что возможность такая имеется, то есть освобождает процесс из блокировки, выбирая как правило первый процесс, который находится в очереди к нужному ресурсу.

Мониторы сами являются ресурсами!

Функции wait и signal определяются на переменных типа условие. Их тут 2 c_read, c_write (можно читать, писать).

Особенности процессов читателей и писателей. Писатели могут изменять значения переменных, читатели – только читать.

Наиболее популярный пример – система продажи билетов на самолеты, поезда, кино, театры – то есть системы, в которых приобретаются билеты на конкретный рейс/сеанс на конкретное место.

При этом изменять значение выбранного места на конкретное мероприятие может только процесс-писатель. Причем, когда писатель изменяет значение этого поля (очевидно, в базе данных) конкретной структуры, никакой другой писатель не может обратиться к этому же полю/записи и никакие читатели не должны получить доступ к чтению этого значения, потому что они могут прочитать неверное значение.

ТО в этой задаче писатели могут работать только имея монопольный доступ к разделяемой переменной.

Читатель, если в это время писатель не изменяет этого поля структуры, могут читать, причем параллельно с другими читателями (они друг другу не мешают)

В мониторе 4 функции – начать/закончить читать/писать.

Nr = nr + 1 – это инкремент и мы, как профессионалы, должны использовать команду инкремент, потому что она реализована с аппаратной поддержкой и будет выполнено меньше действий.
Обращаем на это внимание.

Monitor. observe;
variables
nr : int; //читатели
wrt : logical; //активный писатель
c-read, c-write: conditional;
procedure startread;
begin
 if (wrt or turn(c-write)) then
 wait(c-read);
 nr = nr + 1;
end;
begin
 Signal(c.read);
end;
procedure stopread;
begin
 nr = nr - 1;
 if (nr = 0) then
 signal(c.write);
end;
procedure startwrite;
begin
 if (nr > 0 or wrt) then
 wait(c.write);
 wrt = true;
end;
procedure stopwrite;
begin
 wrt = false;
 if (turn(c.read)) then
 signal(c.read);
 else
 signal(c.write);
end;

begin
 nr = 0;
 wrt = false;
end;

Важно, что читатели могут читать параллельно. Чтобы начать читать читатель вызывает функцию start_read. Он может начать читать, если нет активного писателя или ждущих писателей (очередь ждущих пуста). Иначе он блокируется на переменной типа условие функцией wait(c_read). Если проходит проверку, количество активных читателей

Функция start_read: читатель не сможет начать работу, если есть ждущие писатели. Когда читатель заканчивает читать, он проверяет – все ли дочитали – цепная реакция. Если нет, то сигнал can_write. Всем дается возможность прочитать.

Тут устала и забила, нормально расписано у крис.

Мы уже видели Семафоры, функция P и W, функции wait и signal. Для всех них характерно наличие 2 функций, одна из которых позволяет захватить, другая - освободить критическую секцию.

Различные ОС предоставляют разные средства. В IBM370 кроме семафоров были функции wait(x) и post(x), где переменная икс – байт блокировки (как мы рассматривали в set_and_test).

Посмотрим, какие функции предоставляет Windows.

WaitforSingleObject() и waitforMultipleObject().

Из названия следует – блокировка процесса в ожидании освобождения единственного объекта ядра или сразу нескольких. Такими объектами могут быть mutex и event.

Например, если процесс ожидает освобождения события, то

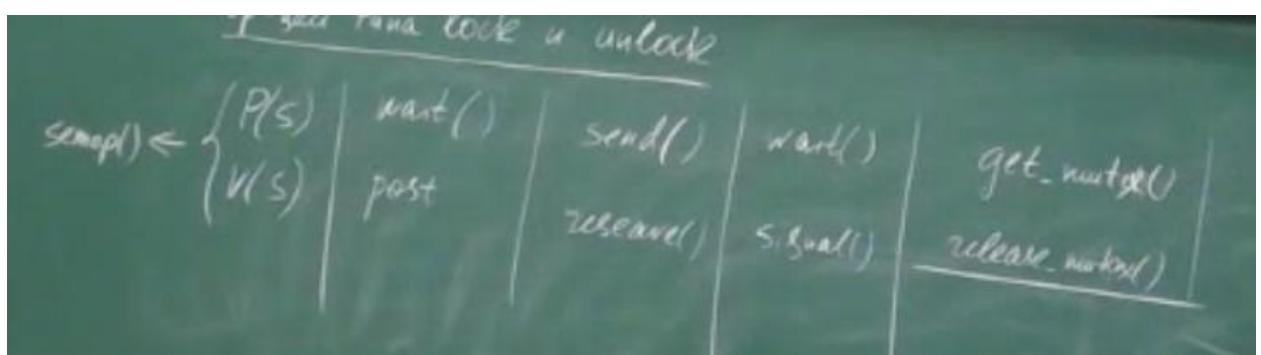
waitforSingleObject(myevent, INFINITE)

Парной к ней является сбросить



ОС предоставляет пользователю функции типа lock и unlock.

К ним относятся все блокирующие и разблокирующие. Коротко – можно представить в виде сопоставления функций P(S) и V(S) (это самые общие названия).



Наиболее интересные – send и receive. Речь идет о сообщениях – наиболее общее средство взаимодействия параллельных процессов. Начать хотя бы с параллельных сетей, где возможно взаимодействие только отправкой и приемом сообщений.

Get_mutex, release_mutex – гипотетические!

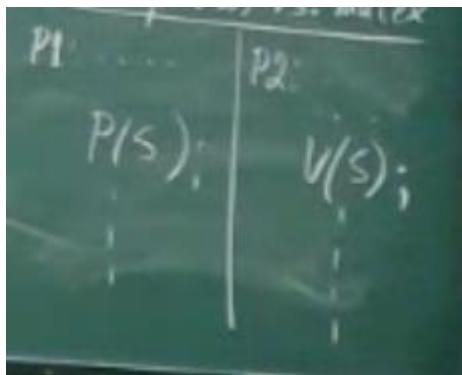
ОТЛИЧИЕ семафора от мьютекса.

Семафоры в ряду этих функций обладают свойством, выделяющим их. Это и демонстрирует сравнение. Называется semaphores vs mutex (семафоры против мьютексов).

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.

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

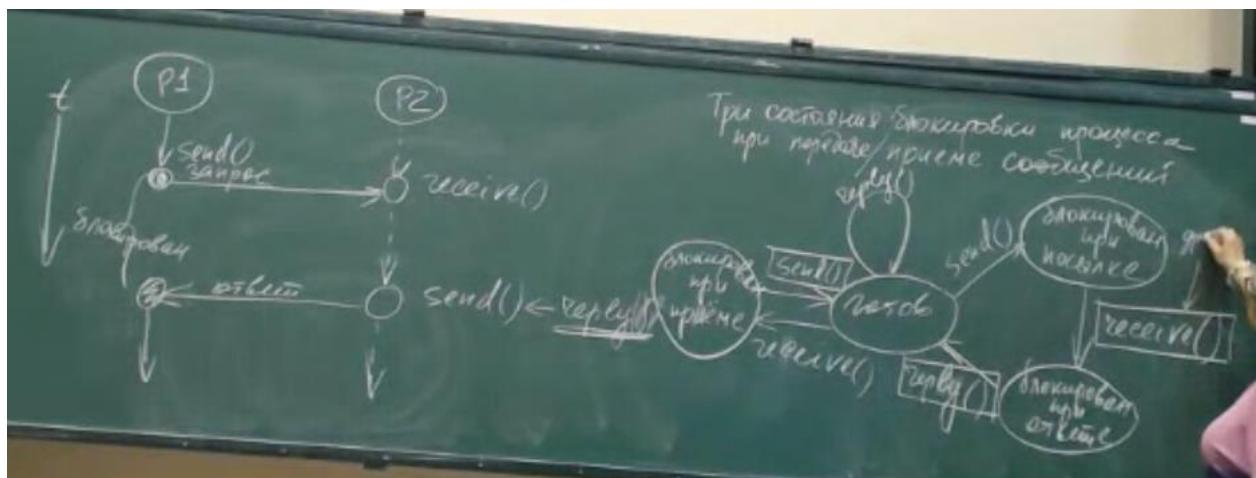
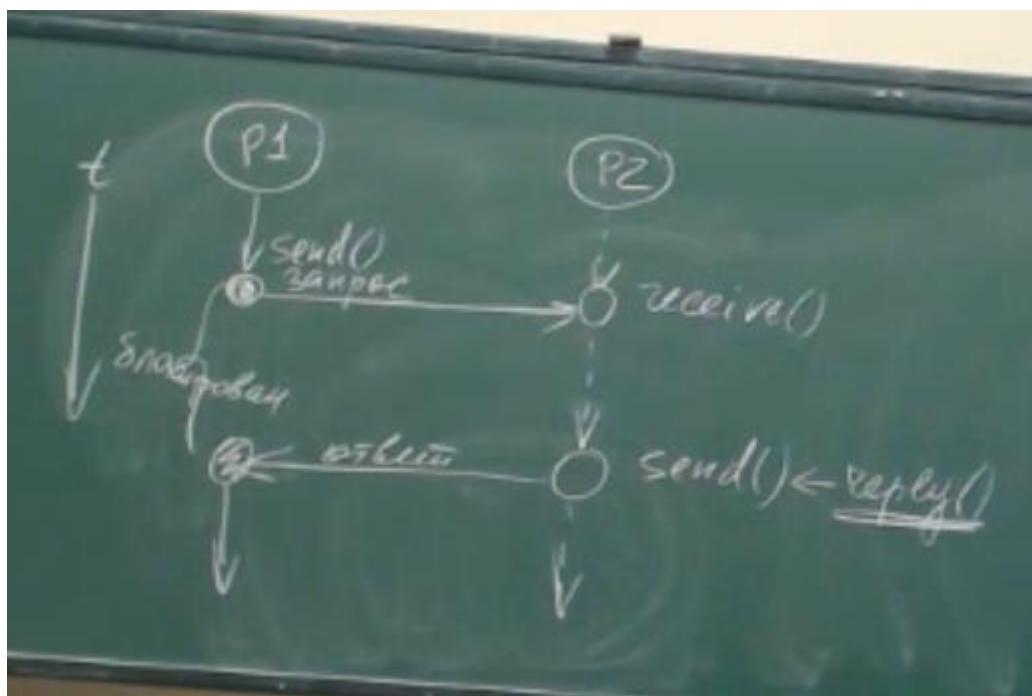
Это различие можно увидеть в производство-потребление и читатели-писатели.

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 писатель не захватили, неважно что там было до.

Синхронизация процессов

P2 должен принять это сообщение. Какое-то время P2 потратит на обработку сообщения и также вызовет функцию send, чтобы отправить ответ. Send запрос и send ответ. P1 получит и может продолжить свою работу. Это и есть синхронизация.



В рамку – функции, которые выполняются от имени другого процесса.

Алгоритм Ламфорта Булочная (Bakery algorithm)

Этот алгоритм решает задачу вхождения процесса в критическую секцию способом, который мы не рассматривали. Проблема взаимоисключений является важной, и это решение интересно. Оно может выполнено программными средствами, или с помощью системных вызовов. Это уникальное решение, которое породило большое количество решений, которые обсуждаются и в наши дни.

Алгоритм решает проблему вхождения большого числа процессов в критическую секцию в приложении. Основная идея была навеяна работой булочной. Потребители при входе берут номера и первыми обслуживаются те, у которых меньшие номера. Имеется проблема небольшая. Входят через дверь, но могут одновременно войти в дверь (процессы).

Когда процесс желает войти в свой критический участок, он должен получить номер в очереди. Могут 2 одновременно. Тогда процессы получают одинаковые номера. Как решить этот конфликт? – по номеру паспорта (на майнфрейме – дескриптор процесса – его паспорт, по идентификатору процесса как номеру паспорта).

Решение.

Речь идет о простой очереди. Критическим ресурсом оказывается номер.

Слово *shared* указывает, что массивы являются разделяемыми.

Пусть процесс P_i выбирает номер. i -й может получить номер на 1 больше максимального из выданных номеров.

var choosing : shared array [0..n-1] of boolean;
number : shared array [0..n-1] of integer;
repeat
choosing[i] = true; // Pisz do pisanie liczb
number[i] = max(number[0], number[1], ..., number[i-1]) + 1;
choosing[i] = false;

for j = 0 to n-1 do
begin
while choosing[j] do (* nothing *)
while number[j] <= 0 and (number[j], j) <

for j = 0 to n-1 do
begin
while choosing[j] do (* nothing *)
while number[j] <= 0 and (number[j], j) < (number[i], i) do
(* nothing *)

for j = 0 to n-1 do
begin
while choosing[j] do (* nothing *)
while number[j] <= 0 and (number[j], j) < (number[i], i) do
(* end. enclosed section *)
number[i] = 0; (* nothing *)

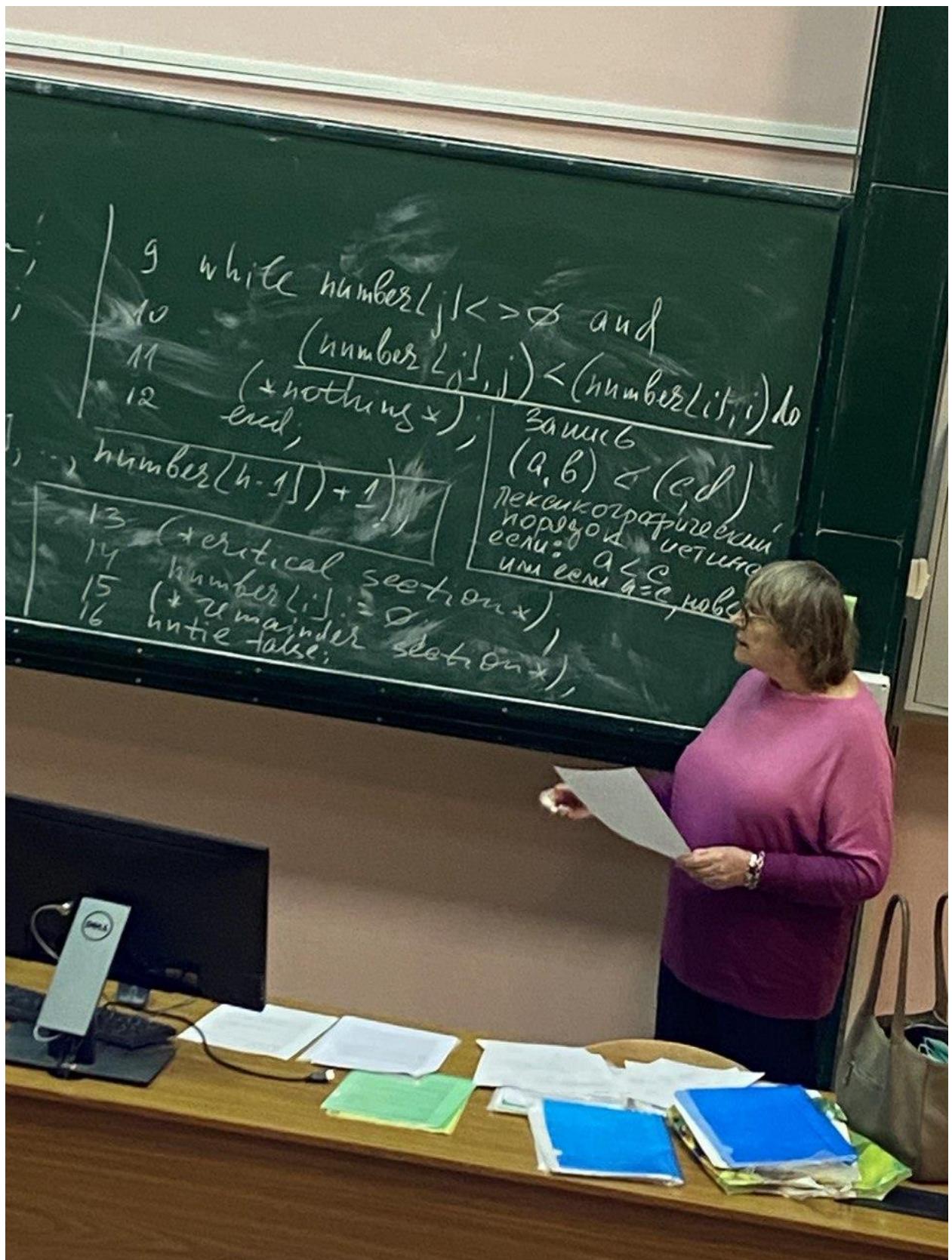
Лекция 13

(повторение булочной с прошлой лекции)



4, 5, 6- выбор номера процессом – пытается получить номер, при этом должен получить номер на 1 больше чем максимальный выданный номер

10 – лексикографическое отношение – если номера равны, то сравниваются идентификаторы из дескрипторов. Записывается следующим образом



14 – если процесс не пытается войти в критический участок, его номер равен нулю

Это называется общим названием concurrent access - параллельный доступ

Блокировки – зло, но необходимое. Блокировки приводят к тому, что другие параллельные операции не могут быть выполнены. Но универсального подхода без блокировок не существует

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 приемная 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

Ada был разработан для военных целей – язык параллельного программирования для моделирования военных действий (взаимодействие войск)

Задачи task в языке ада связываются с помощью входа

Task <имя> is

 Entry <идентификатор входа> (дискретный диапазон)(формальная часть);

 <другие объявления входов>

End <имя>;

Если одна задача выдала сообщение ко входу другой задачи, то обе задачи теряют свою независимость, так как они установили randevu и до тех пор, пока randevu действует, задачи синхронизированы. В механизме randevu отсутствует симметрия – если 1 задача запрашивает randevu со 2 задачей, 2 может принять или отказаться. Р произойдет только когда 2 задача примет этот вызов.

Поэтому в ада есть оператор accept

Accept:=<идентификатор входа>(выражение)формальная часть по последовательность операторов end;

Если принимает, то они синхронизируются – последовательность операторов выполняется от имени обеих задач и результаты они получают одновременно. (без блокировок)



Взаимодействие параллельных процессов в распределенных системах

Синхронизация часов.

Рассмотрим работу программы make в ОС Unix. Большие программы разделяются на несколько файлов. Make проверяет время последней модификации всех исходных и объектных файлов программы. Если время исходного больше соответствующего объектного, то make считает, что файл нужно перекомпилировать. На отдельно стоящей машине это не считается проблемой.

На одной машине – изменение, на другой–перекомпиляция.

(фото)

Часы на компе для редактирования запаздывают и модифицированный файл получает временную метку 1223. Метка об 1224. Модифицированный файл не будет перекомпилироваться. В результирующий файл изменения не попадут.

У каждого компа собственный генератор с собственной точностью

Нельзя синхронизироваться по локальным часам.

В мире есть служба. Усредняет время из разных стран.

Алгоритмы взаимного исключения в распределенных системах

```

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[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
10      (number[i,j],j)<=(number[i,t,i]) do
11      (+nothing+); | заменяется
12      end; | (q,b) < (c,d)
13      number[n-1]:=number[n-1]+1; | лексикографический
14      number[i]:=0; | порядок ветвления
15      (+remainder section+); | если a <= c или even a <= c, moved
16      continue false; | (+critical section+),

```

KOMN. на кот
 врем. на комманду ↓
 1223 1224 1225 1226 ← время на выполнение задачи
 заменяется
 ↓
 1222 1223 1224 1225

KOMN. на кот
 врем. на пакет

в 215 998 > 216 002



```

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[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
10	$(\text{number}[l_0, j]) < (\text{number}[l_1, j]) \text{ do}$
11	$(+\text{nothing}+)$; занес
12	end;
	$(a, b) \subset (cd)$
	пеканкогороду ногоду вершина: если $a \leq c$, $b \leq d$
13	$(+\text{critical section}+)$
14	number[i]:=0;
15	$(+\text{unmainder station}+)$,
16	until false;

KOMR. на KOS
стм. ср. комманд

cospar sum>0

KOMR. на KOS
стм. ср. комманд

отправляем

sum>0

бонус рабочий

sum>0

аварийный:

1. Алгоритм критичанс
2. Алгоритм беркли

TAI - International
Atomic Time

3. Упрощающие алгоритмы

UTC -

искусст 133
9192631777

и 215 998 до 216 002

mean solar second
leap seconds

0	1	2
0	8	10
6	16	20
12	24	30
18	32	40
24	40	50
30	48	60
36	56	66
42	64	76
48	72	80
54	80	90
60	88	100

0	1	2
0	8	10
6	14	20
12	24	30
18	32	40
24	40	50
30	48	60
36	56	66
42	64	76
48	72	80
54	80	90
60	88	100



Лекция 14

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.

Речь о репликации. Очевидно, репликация позволяет повысить надежность системы, но требует поддержание нескольких копий. При этом рассматривается

- строгая непротиворечивость,
- последовательная непротиворечивость
- причинная непротиворечивость
- непротиворечивость FIFO.

Например, для повышения производительности по выполнению запросов банк может положить копии данных по счетам в разных местах. Запросы посылаются в ближайшие копии. Очевидно, что за все надо платить. Платой за обеспечение быстрых ответов будет необходимость репликации, то есть обновление после каждой операции всех копий базы данных.

Чтобы не происходило потери информации, репликации происходили в правильном порядке и используются часы Лампорта.

Неделимые транзакции

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. Такой подход часто называют алгоритмом забияки.

Если процесс обнаруживает отсутствие К, то он инициализирует выборы координатора и посыпает 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 инициализирует 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 посыпает 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, рассыпает $n-1$ сообщение-запрос всем взаимодействующим процессам с указанием ид 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

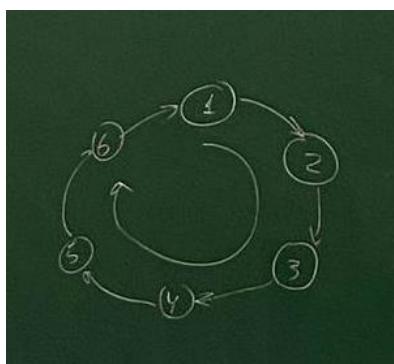
1. Процесс-получатель не находится в данной КС и не собирается в нее входить. Посыпает запросившему процессу разрешение.
2. Получатель находится в указанной КС. Тогда не посыпает сообщение о разрешении, а ставит сообщение-запрос в очередь.
3. Получатель хочет сам войти в указанную КС. Тогда сравнивает 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.

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, посыпает токен дальше.

Если какой-либо из процессов завершится аварийно, логическая цепь будет разорвана

Самой надежной системой является централизованный алгоритм

В распределенной – должны быть синхронизированы по времени

Неделимые транзакции

Все что мы рассмотрели – механизмы нижнего?? Уровня

Модель транзакций пришла из бизнеса. Когда говорят транзакция – подразумевается неделимость. Можно представить процесс переговоров между 2 фирмами. Если он подписан 2 сторонам – все, изменен не будет.

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 (атомарность, согласованность, изолированность, устойчивость) – свойства транзакции

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

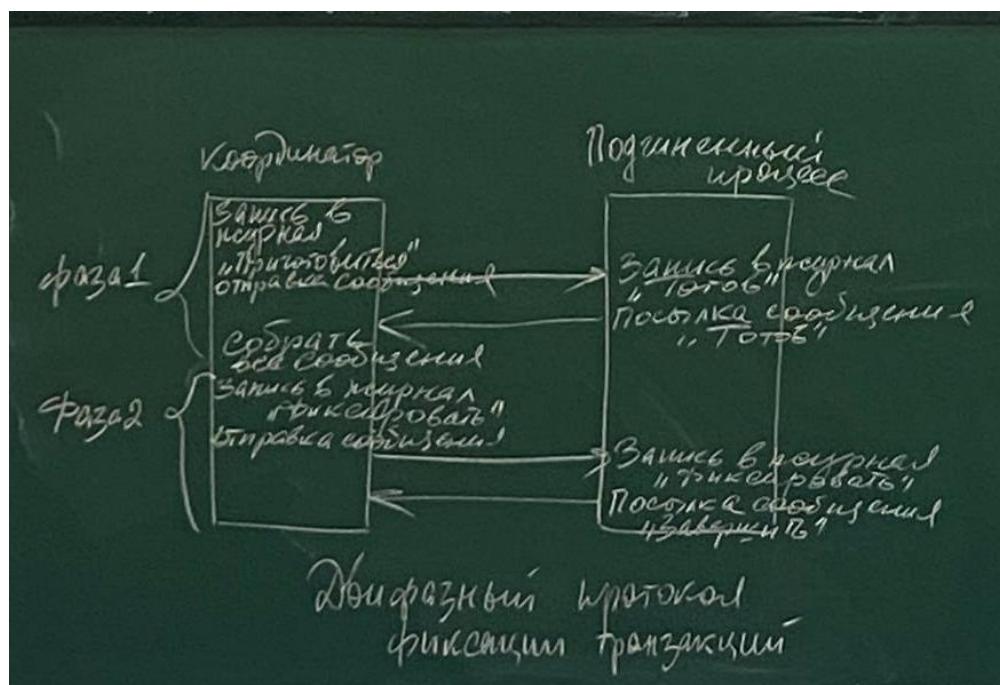
протокол двухфазной фиксации транзакции

Один процесс выполняет функции координатора. К начинает транзакцию и делает запись в свое локальном журнале регистрации. После этого он посылает 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 посылают сообщение готов к фиксации. Потом переходят в состояние ожидания соответствующего сообщения от K.

K собирает все сообщения от подчиненных процессов готов к фиксации. 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.

Если K получил сообщения от всех – посылает сообщение фиксировать и все фиксируют результат транзакции

Этот протокол гарантирует успешное завершение транзакции всеми процессами



Тупики

Книга Шоу – логическое проектирование ОС

От ресурса к процессу – получение, наоборот – запрос

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

Типы ресурсов и тупики

Ресурсы с точки зрения особенности их использования, делятся на повторно используемые и потребляемые. К повторно используемым относятся аппаратура компьютера, процессоры, память, каналы, устройства ввода-вывода и ПО (например, реинтегральные коды ОС, системные таблицы).

К потребляемым – сообщения. Взяли – и оно перестает существовать.

(атомарность, согласованность, изолированность, устойчивость) – свойства транзакции

Лекция 15

Тупики

Книга Шоу – логическое проектирование ОС

От ресурса к процессу – получение, наоборот – запрос

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

Типы ресурсов и тупики

Ресурсы с точки зрения особенности их использования, делятся на повторно используемые (ресурсы, использование которых никак не меняет их свойства) и потребляемые.

К повторно используемым относятся аппаратура компьютера, процессоры, память, каналы, устройства ввода-вывода и ПО (например, реинтегральные коды ОС, системные таблицы).

К потребляемым – сообщения. Взяли – и оно перестает существовать.

Память раньше была важным ресурсом, сейчас – проще. Процессор – как средство выполнения.

Простые очереди (то, что раньше называлось pulling теперь не используется).

Семафор – повторно используемый, так как структура никак не меняется после захвата и освобождения, функции также не меняются. Вообще все это называется объектами ядра.

Вся теория тупиков написана для повторно используемых ресурсов, потому что их количество в системе как правило известно и неизменно (не меняется резко), в отличие от сигналов (любой процесс может произвести или потребить любое количество сигналов). А как же семафоры – если есть приложение, которое использует набор семафоров, то мы определили количество семафоров в наборе, оно динамически не меняется, с этим можно работать.

Чтобы подчеркнуть, что теория тупиков в настоящее время не актуальны (так как там про аппаратное, а это сильно поменялось). Но это никак не коснулось используемых объектов ядра.

4 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
2. Ожидание – когда процессы, удерживая полученные им ресурсы, запрашивают и ждут получения дополнительных ресурсов, чтобы продолжить свое выполнение. Hold and wait
3. Не перераспределяемость – 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

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.

3 основных метода борьбы с тупиками.

1. Недопущение тупиков (исключение самой возможности возникновения тупиков).
2. Обход тупиков (предотвращение тупиков)
3. Обнаружение и восстановление

1. Недопущение 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 – незэффективное использование ресурсов. Это касается аппаратной составляющей.

Это также может привести к бесконечному откладыванию.

Раньше - Задания разбивались на части, каждая часть запрашивала.

Надо сказать, какое из условий возникновения тупика нарушается в этом подходе!!!! (вроде ожидание)

2. **Второй подход – реализуется с помощью упорядочивания ресурсов.** Também 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 онидерживают.

Тоже нарушает одно из условий.

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.

Системы реального времени в тупики попадать не могут. Это техническое понятие, РВ – выполнение обработки запроса, внешнего по отношению к процессу – за строго определенный интервал времени. Это определяемая характеристиками внешнего процесса или объекта – они определяют время выполнения.

В системах РВ процессы, которые выполняются в системе, хорошо изучены, у разработчиков есть полная информация о них, и для таких систем вполне может применяться иерархический метод.

3. Третий способ. Устраняется условие неперераспределяемости. Если процесс не может получить нужный ему ресурс, то он должен освободить занимаемый им ресурс.

Отобрать у процесса ресурс – значит откатить процесс к состоянию до получения данного ресурса. А чтобы откатить, надо запомнить точку отката, или выполнять обратные действия. Отобрать ресурс – большая проблема. Она должна реализовываться специальным образом.

Существуют такие реализации, когда для процесса пишутся обратные действия. Нелинейно Этот подход может привести к захвату и освобождению одних и тех же ресурсов.

Надо иметь в виду, что системы существенно изменились

2. Обход 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.

Менеджер ресурсов ищет последовательность процессов, которое может гарантированно завершиться.

Процессы	Текущие ресурсы	Свободные единицы ресурса	Заявка
P1	1		4
P2	3		5
P3	5	2	9

Задача менеджера – определить, является ли данное распределение безопасным относительно тупика.

Безопасным относительно тупика называется состояние, когда все процессы могут успешно завершиться (найдена такая последовательность). Если такая последовательность не найдена – небезопасное.

В данной таблице состояние системы является безопасным относительно тупика. Можно увидеть. Процесс 2 сможет завершиться, даже если он запросит 2 единицы ресурса, поскольку они имеются у системы. Больше он запросить не сможет – максимальная потребность 5. Завершившись, он вернет в пул свободных ресурсов занимаемые им единицы ресурсов и в сумме со свободными единицами они смогут удовлетворить 1 процесс. Завершившись, 1 освободит, они вернутся в пул свободных ресурсов системы и в сумме смогут удовлетворить максимальную потребность 3 процесса. Значит существует последовательность процессов, что все смогут завершиться.

Совсем необязательно, что если текущее состояние безопасно, то следующее будет также безопасно относительно тупика. (переход от 1 таблицы). Например, процессу 1 выделили

Процессы	Текущие ресурсы	Свободные единицы ресурса	Заявка
P1	2		4
P2	3		5
P3	5	1	9

Здесь нельзя найти последовательность. Небезопасное относительно тупика состояние.

Небезопасное состояние не обязательно приведет к тупику – процессы могут не запросить заявленного ими количества ресурсов.

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 позволит 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

Для выполнения требуются $n!$ проверок. То есть время выполнения алгоритма банкира очень велико. Кроме того, у алгоритма имеются очень серьезные ограничения.

В современных системах процессы выполняются по мере ..., а ресурсы выполняются по мере надобности. Все – динамически. Но нельзя исключать существование систем, в которых характеристики процессов хорошо изучены и все детерминировано.

Поскольку алгоритм дейкстры очень затратный, он имеет теоретическое значение

2. Алгоритм хаббермана

Одним из самых известных алгоритмов, который рассматривается как развитие алгоритма дейкстры – алгоритм хаббермана

Deadlocks avoidance Haberman's algorithm

Существует несколько изложений.

Чтобы узнать, каким является текущее состояние системы, работа начинается с некоторого заданного состояния. И алгоритм D, и X, имитирует выполнение каждого процесса, выделяя ему максимальное количество запрошенных ресурсов. Если все процессы при этом завершаются, то состояние безопасное.

В алгоритме X 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

- Deadlocks avoidance Kebermann's algorithm
- Max-Avail matrix $A = (a_1, a_2, \dots, a_m)$
где $a_i = |R_i|$
 - Max-Claim matrix $B = \begin{vmatrix} b_{11} & b_{1m} \\ b_{n1} & \dots & b_{nm} \\ \vdots & & \vdots \end{vmatrix} = \begin{vmatrix} B_1 \\ \vdots \\ B_n \end{vmatrix}$
 $b_{ij} = \max|R_j|$ среди P_i
 - Allocation matrix $C = \begin{vmatrix} c_{11} & c_{1m} \\ c_{n1} & \dots & c_{nm} \\ \vdots & & \vdots \end{vmatrix} = \begin{vmatrix} C_1 \\ \vdots \\ C_n \end{vmatrix}$
 $c_{ij} = \#R_j$ для P_i

- 1) Для всех k , $B_k \leq A$
- 2) $C \leq B$
- 3) $\sum_{k=1}^n C_k \leq A$
- 4) Вектор доступных ресурсов D :
 $D = (d_1, d_2, \dots, d_m) = A - \sum_{k=1}^n C_k$
- 5) Матрица запросов E

- 7) Предварительное состояние
- $$D \leftarrow D - F_i$$
- (доступно-запрос)
- $$C_i \leftarrow C_i + F_i$$
- (распределено+запрошено)
- $$E_i \leftarrow E_i - F_i$$
- (уже-запрошено)
- Алгоритм:
1. $F_i \leq E_i \leq D$
 2. $D \leftarrow D + C_i$. Результат как зад.

Need matrix $E = \begin{vmatrix} e_{11} & e_{1m} \\ e_{n1} & \dots & e_{nm} \\ \vdots & & \vdots \end{vmatrix} = B - C = \begin{vmatrix} E_1 \\ \vdots \\ E_n \end{vmatrix}$

6) Матрица запросов $F = \begin{vmatrix} f_{11} & f_{1m} \\ f_{n1} & \dots & f_{nm} \\ \vdots & & \vdots \end{vmatrix} = \begin{vmatrix} F_1 \\ \vdots \\ F_n \end{vmatrix}$

3. баланс / текущий состояния

$$D \leftarrow D + F_i$$

$$C_i \leftarrow C_i - F_i$$

$$D \leftarrow D - C_i$$

$$E_i \leftarrow E_i + F_i$$

- Max-avail matrix $A = (a_1, a_2, \dots, a_m)$; где $a_i = |R_i|$ - количество единиц i-го ресурса.

Заявки могут быть представлены только двумерным массивом.

- Max-claim matrix $B = |b_{11}, \dots, b_{1m}| \dots |b_{n1}, \dots, b_{nm}| = |B_1, \dots, B_n|$
 b_{ij} - максимальное количество ресурса R_j , заявленное процессом P_i

- Матрица распределения
- Allocation matrix C

C_{ij} - Количество единиц ресурса R_j , которые выделены ресурсу P_i

1) Для всех k , $B_k \leq A$, то есть процесс не может требовать больше единиц ресурса, чем доступно. Это отражено в векторе A .

2) $C \leq B$ – процесс не может пытаться запросить больше ресурсов, чем указано в его заявке

3) сумма ($k=1; n$) $C_k \leq A$ – никогда не выделяется больше ресурса, чем доступно

4) дополнительно для анализа вводится вектор доступных ресурсов D :

$D = (d_1, \dots, d_m) = A - \sum_{k=1}^n C_k$ – разница между имеющимися и уже выделенными единицами ресурса.

5) Матрица запросов E

Need matrix E – возможные запросы

6) Матрица запросов Request matrix F

7) Предварительное состояние. При этом предположим, что удовлетворяются все запросы. Тогда

$D \leftarrow D - F_i$ (доступно-запросы)

$C_i \leftarrow C_i + F_i$ (распределено + запрошено)

Ei<-Ei-Fi (нужно – запрошено)

Запрос процесса может быть удовлетворен только в том случае, если предыдущее состояние системы безопасно

Алгоритм проверки безопасного состояния.

1. Выбирается незавершившийся процесс P_i такой, что $Ei \leq D$. То есть доступно больше ресурсов, чем нужно процессу P_i . Если такого процесса нет, то переходим на шаг 3, иначе - 2
2. $D < D + Ci$. P_i отмечается как завершившийся и переходим на шаг 1
3. Если все процессы отмечены как завершенные, то система находится в безопасном состоянии, иначе – в опасном. Если система находится в небезопасном состоянии, то запрос блокируется и состояние системы сбрасывается следующим образом. Безоп/не безопасный

D<-D+Fi

Ci<-Ci-Fi

Ei<-Ei+Fi

Откат по пункту 7

Дополнительно процесс, помеченный как завершенный на шаге 2, сбрасывается

D<-D-Ci

И повторно P_i помечается как незавершенный

(пример с почты)

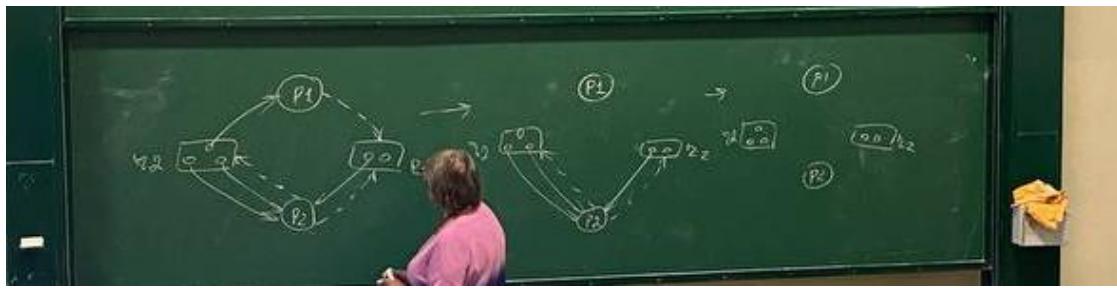
3. Обнаружение тупиковых ситуаций

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 график становится необходимым, чтобы убедиться, что система пришла в тупик и причины. Речь идет о большой системе взаимодействующих процессов.

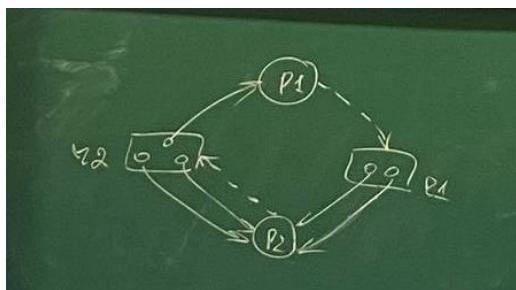
Тупиковая ситуация с использованием ГМХ обнаруживается методом редукции графа. Если запрос процесса может быть удовлетворен, то такая дуга может быть удалена. В результате того, что запрос удовлетворен, он может освободить ресурсы, которые ранее занял, и эти ресурсы могут быть использованы другими процессами. Если в результате 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.

Пример.



Пробуем редуцировать. Если будет удовлетворен запрос P_1 , то он сможет завершиться и освободить. И тогда можно будет удовлетворить и P_2 . В результате получаем изолированные вершины процессов в рассматриваемом графе \rightarrow система не находится в тупике.

Пример. Усложним ситуацию. Имеется петля (цикл) запросов



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

Очевидно, что для того, чтобы обнаружить, находится ли система в состоянии тупика, необходимо описать ее.

Можно описать 2 матрицами – матрицами запросов и матрицами выделения.

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

Обычно процессы запрашивают ресурсы по одному. Хотя если взять страницы.... Но для того, чтобы проводить анализ состояния системы кроме указанных матриц надо иметь вектор свободных ресурсов F. F_i – сколько единиц ресурса свободно.

Для каждого процесса есть строка – одномерный массив. Можем сравнивать строки с вектором свободных ресурсов. Наша задача – редуцировать граф. Останутся процессы, которые попали в тупик.

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.

Восстановление работоспособности системы.

2 глобальных подхода

1. Последовательно завершаются процессы, попавшие в тупик. У них отбираются ресурсы и другие могут продолжить
2. Завершение других процессов, которые выполняются самостоятельно, не имеют отношения к тупику

Либо kill, либо откат до возникновения запроса. Для этого надо хранить состояния.

Тупики в распределенных системах

В распределенных системах процессы могут обмениваться только сообщениями. 1 тип – известны пот, неизвестны пр, 2 – известны пр, неизвестны пот, 3 – никто неизвестен

Особенности:

- Задержка передачи данных в сетях затрудняет точную оценку состояния процессов в распределенной системе. В результате переход в тупиковое состояние мб не сразу замечен или, наоборот, мб принято решение, что система находится в тупике, хотя она продолжает оставаться работоспособной, то есть в тупике не находится

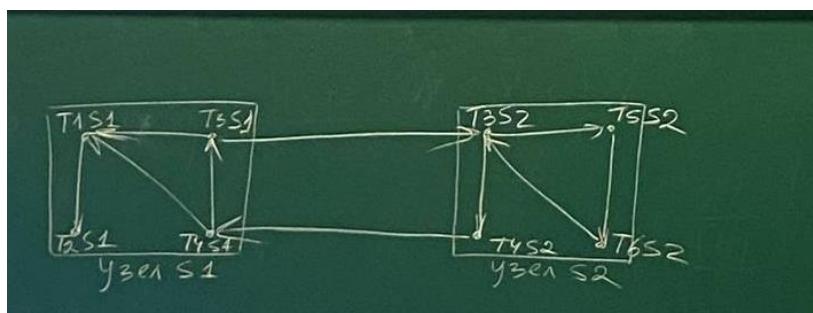
В РС задача обнаружения и недопущения тупиков сводится к известным методам, которые описаны для повторно используемых ресурсов, а именно – 3 подхода – предотвращение тупиковых ситуаций, обход, обнаружение и устранение

Особенностью рассуждения является понятие блокировка.

Тупики при транзакциях

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.

(граф ожиданий)



1. Проведем направленную дугу из точки $TiSk$ в точку $TjSk$, если транзакция $Ti \dots$ в блокирована в ожидании, когда транзакция Tj освободит ресурс, который нужен транзакции Ti

Будем говорить, что узел $TiSk$ ожидает ресурс от узла $TjSk$

2. Проведем направленную дугу из $TKSi$ в $TKSj$, если транзакция TK в узле ViJ блокирована в ожидании осо. Будем го

Необходимым и достаточным условием возникновения тупика является наличие цикла в графе.

Существуют 3 подхода.

При этом предотвращение (недопущение) связано с предварительным получением всех блокировок. То есть транзакция получает все блокировки перед началом выполнения. И сохраняет все блок на протяжении всей транзакции. Если другой транзакции требуется занятая блокировка, то она должна дождаться, пока все необходимые ей блокировки не будут доступны.

Обход предполагает, что возможность тупика анализируется до тупика, то есть анализируются блокировки (запрос транзакции). Анализируется, приводит ли ожидание запрошенного узла к тупику.

Коротко: транзакции начинают выполняться и запрашивают элементы данные, которые необходимы. В результате получения, они их блокируют.

Диспетчер блокировок проверяет, доступна ли блокировка. Если да, то диспетчер блокировок выделяет элемент данных, и транзакция получает блокировку. Однако если элемент данных заблокирован другой транзакцией в тн несовместимом режиме, диспетчер блокировок запускает алгоритм, чтобы проверить, приведет ли оставление транзакции в состоянии ожидания к deadlock.

Другими словами, определяется, может ли транзакция ждать, или одна должна быть прервана.

Существует 2 алгоритма

1) wait-die – ожидание смерть

2) wount - wait – ожидание ранение

Первый. Если $T1$ старше, чем $T2$, то $T1$ может подождать. Иначе – если $T1$ младше, то $T1$ прерывается и перезапускается потом.

Второй. Если $T1$ старше $T2$, $T2$ прерывается и перезапускается позже. Иначе – если $T1$ младше $T2$, то $T1$ может подождать

Обнаружение. 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-forgraph.

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.

Последний подход подходит в первую очередь для систем с низким уровнем Т, в которых требуется быстрый ответ на запросы блокировок.

Обработка тупиков в распределенных системах

Распределенная БД – БД, которая находится на нескольких сайтах и использует данные с разных сайтов. Это сложнее, так как объем данных распределяется неравномерно между сайтами и время обработки сильно варьируется. В результате одна и та же Т может выполняться (быть активной) на одних сайтах и неактивна для других.

Пример: на сайте 2 конфликтующие Т. Одна из них может находиться в неактивном состоянии. Такого в централизованной системе случиться не может. Эта проблема получила название – проблема местоположения транзакции

В этой модели Т несет определенные детали при перемещении с одного сайта на другой. Под деталями, например, понимается список необходимых таблиц, список требуемых сайтов, список посещенных таблиц и сайтов. Список таблиц и сайтов, которые еще нужно посетить. Список блокировок с указанием типов.

После того, как Т заканчивается или фиксацией, или прерыванием, информация должна быть отправлена на все заинтересованные сайты (репликация, согласование данных)

Решение строится на известных методах – недопущение, обход, обнаружение. Недопущение – транзакция должна запросить все блокировки. Для этого должна использоваться централизованная система, 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. Недостатки: длительное время задержек связи между сайтами. В результате время между Т увеличивается. Если Т переходит в состояние ожидания, не получив все ответы, это может долго, причем уже заблокированные все еще блокированы, не дают выполняться другим Т.

Если управляющий сайт потерпел крах, он не может связаться с другими сайтами, которые удерживают блокировки.

Обход

Лекция 16 (от И)

Тупики при транзакциях

.....
как много раз она обращала внимание наше, методы борьбы с тупиками используются те же, что и в централизованных системах

.....
предварительная проверка запросов процессов, обход тупика, чей процесс не может быть удовлетворен, может быть завершен, обнаружение тупиковых ситуаций, система работает не задумываясь, что может попасть в тупиковую ситуацию (подозрения по таймаутам), тогда в системе может быть запущена проверка, находится ли система в тупике

Мы уже рассмотрела недопущение тупиков, (предотвращение), аналогично централизованной системе, как предполагается – транзакция должна запросить все блокировки (среда обеспечивает монопольного доступа к разделяемым ресурсам – локи)

Избежание тупиков (обход – мы называем), аналогично ситуации в центре системы – избежать распределенный тупик можно за счет предвар обработки ситуации, которая может возникнуть, если запрос транзакции на блокировку будет удален.

Дополнительно в распределенных системах должны быть решены вопросы положения транзакций (локейшин) и контроль транзакции, при этом могут возникнуть именно из распределенной природы транзакций след конфликты, след конфликты (при этом надо разделять конфликты между двумя транзакциями на одной сайте и 2 транзакции на разных сайтах)

Случае конфликта одна транзакция может быть прервана или ей разрешено ждать в соответствии с распределенными алгоритмами ожидания, или можно коротко нахватать – в соответствии с распределенным ожиданием.

Предположим имеются 2 транзакции T1 и T2. T1 – эрайвз (превыбает) на сайт P и пытается заблокировать элемент данных, который уже забачен транзакцией T2 на этом же сайте. Этот сайт называют зоной P (сайт P – зона P) ((хз P англ или П русская)

И возможны 3 алгоритма

1. Distributed wound-die – распределенная смертельная рана)
первый способ – если T1 старше T2, то есть T2 пришла позже, то T1 может подождать, при этом T1 может возобновить свое выполнение после того, как сайт P получит сообщение о том, что транзакция T2 успешно завершилась или была прервана.

второй способ – если T1 младше T2, то T1 прерывается. При этом специальный менеджер на сайте P должен отправить сообщение всем сайтам, которые посещала T1, чтобы корректно ее прервать. При этом управляющий сайт должен уведомить пользователя, что T1 была прервана на всех сайтах успешно.

2. Distributed wait-wait – распределенное ожидание ожидание (да да, именно 2 раза слово ожидание). Опять же два способа
 - первый способ– Если T1 старше T2, то T2 следует прервать. Если T2 активна (то есть выполняется на сайте P), то сайт P прерывает и откатывает T2, после этого рассыпает сообщение об этом на другие сайты. Если T2 покинула сайт P, (то есть завершилась на сайте P), но выполняется на сайте (фул???), то сайт P оповещает, что T2 была прервана. Но по логике вещей сайт Q должен прервать эту T2 и отправить соответственно сообщение другим сайтам, или это может сделать сайт управляющие транзакцией (скажем Лидер).
 - второй способ?? – Если T1 младше T2, то T1 разрешено ждать (тоесть T1 блок в ожидании). При этом T1 сможет возобновить выполнения после того, как сайт P получит сообщение , что T2 завершилась.

Это формальные вещи и она это понимает. Почему нам это рассказывают?? – это современный материал проф значимый относительно тупиков, это большая проблема для распределенных приложений, который в большей части выполняются как транзакции. (T – очень интересный механизм).----- тут все про транзакции началось (я расписывалась и не записала) ВСЕ БЫЛО ИЗЛОЖЕНО!!!

И последний метод связан с распределенными транзакциями

3. Обнаружение распределенных тупиков (англ вариант я не запишу). Также как и в централизованных системах, обнаружить, что распределенная транзакция попала в тупик можно с помощью графа, но при этом создаются так называемые глобальные графы ожидания (картинка есть – она уникальная, взята из старой книги ...автор.... больше она ее не встречала нигде!!!!!!) Наличие цикла ожидания в глобальном графе является признаком тупика или взаимной блокировки (дедлоком). Но несмотря на то, что механизм известен, обнаружить такую блокировку сложно, потому что транзакция распределенная и это значит, что она ожидает ресурсов в сети, поэтому для выявления таких ситуаций используются таймеры. Каждая Т связана с таймером, установленным на определенный интервал времени (будильник). Очевидно, что этот интервал выбирается предположительно разработчиками и отражает ожидаемое время завершения транзакции. Если Т не завершает за установленный интервал времени, то формируется признак – возможный..... тут ты мне писала и я не успела....

Есть еще один инструмент для обнаружения тупиковых ситуаций, так называемый детектор тупиковых ситуаций. В централизованной системе имеется один детектор тупика. В распределенной системе может быть несколько. При этом детектор в распределенной системе может обнаруживать тупиковых ситуаций для подконтрольных ему сайтов.

Обратить внимание на очевидность подхода – параллельное обнаружение (сократить время обнаружения). Имеются 2 альтернативы для обнаружения тупиков в распредел системе.

1. Централиз детектор тупика – то есть одна локация назначается центральным детектором взаимоблокировки
 2. Иерархический детектор тупика. Детекторы сформированы в иерархию и уже дело разработчика системы , как будет построено взаимодействие системы в этой иерархии.
 3. Детектора распредел тупиков. Все сайты участвуют в обнаружении и устранении тупика.
-

три показательные теоремы (на прошлой лекции она забыла)

1. Теорема 1: граф – речь идет о бихроматическом направленном графе Холда, Граф явл полностью сокращаемым, если существует такая последовательность сокращений, которая устраниет все дуги. Если граф нельзя сократить полностью, то анализируемое состояние является тупиком (см рисунок полностью сокращаемого графа). Какие дуги можно удалить, дуги запросов..... Какая дуга запросов может быть удалена, та которая отражает запрос на ресурс, который есть у системы, или который освобожден.
 2. Теорема 2: Цикл в графе повторных ресурсов является необходимым условием тупика.
 3. Если состояние S (С большое) не является состоянием тупика и из S->T (из С система переходит в T), то состояние T(может) – явл состоянием тупика, если операция процесса Pi (п итое) явл запросом. И в результате Pi находится в T в тупике. Тупик может наступить в системе в результате запроса на доп ресурсы.
-

Дополнительно: Алгоритм обнаружения тупиков в распредел системах, который имеет название Алгоритм Chandy-Misra-Hoas (по именам авторов)

Если при очередном запросе в системе необходимый ресурс занят, то процесс запросивший этот ресурс генерирует спец сообщение и посыпает его процессам которые предположительно могли захватить данный ресурс, это система взаимодействующих процессов,.....

В этом СПЕЦИАЛЬНОМ сообщении указываются 3 числа: 1- свой идентификатор (номер в системе), 2 - также свой номер, 3 - номер процесса, которому посыпается сообщение. Процесс получивший сообщение проверяет ждет ли он сам ресурс, занятый другим процессором (не этот же самый, а просто находится ли он в состоянии блокировки). Если он тоже блокирован, (вопрос – не получает проц время и не может выполняться —ответ это какая-то спец блокировка может быть), то прописывает свой номер во второе поле полученного сообщения. В 3 поле записывает номер процесса, захватившего ресурс, освобождение которого он ожидает, то есть он редактирует полученное сообщение и посыпает его дальше. В результате такой многократной пересылке: Если процесс получает сообщение и в первом поле этого сообщения обнаруживает свой номер, и главное в 3 поле он также обнаруживает свой номер, то ничего ему не остается, как сделать вывод о том, что система в тупике (круговое ожидание). Очевидно, что можно уничтожить этот процесс.

Однако если в тупике много процессов, то такой подход довольно расточителен, очевидно, что можно по разному реализовывать выход из тупика, например можно доп, чтобы процессы в своем сообщение прописывали свои номер, но еще и отдельный список процессов – получивших это сообщение. И тогда, например, можно завершить процесс с наибольшим номером.

Обращает наше внимание на след обстоятельства, которые не указывала в методе убеждения (Ха может и не такой метов) тупика.

тут ломаный англ, дестребует девойд

Каждая Т хранит время своего создания, при этом очевидно, предполагается, что время синхронизировано, например по алгоритму Лампорта, в результате можно определить, какая Т старше, а какая моложе, на этом построены способы предубеждения. И также обращает внимание: Например, блокирование Т происходит тогда, когда Т запрашивав ресурс моложе той, которая ресурс удерживает.

Дополнительно обращает внимание.

В алгоритме, где 3 автора (выше написан). Примитивное решение – посылаются сообщения зомби.... (не уверена).

Если процесс обнаруживает, что слишком должно ждет, то может послать такое сообщение и также возможно использование сообщения зомби.

Это распространенный способ в локальных сетях (чтобы быть уверенным, что сеть свободна и передача выполняется надежно)

НОВАЯ ТЕМА

Архитектура ядер операционных систем

Существует 2 типа ядер

1. монолитное
2. микро- ядро

2 тип – микроядерной архитектуры, как только не называют, и нано и экза, но идея одна и та же

1 тип – исторически первый. Монолитное ядро – единая программа, состоящая из подпрограмм, собруутинс (ее английский)

В результате – любое изменение вносимое в такую программу, требует ее перекомпиляции.

Очевидно, что современные системы с монолитным ядром,

Юникс, линукс – с монолитными ядрами.

В обоих системах ОС , которые позволяют вносить доп функционал без его перекомпиляции.

2 тип – микро керал (ее англ)

В отличии от 1 типа, очень небольшое ядро, которое выполняет только самые низкоуровневые действия, при этом другие компоненты ОС являются самостоятельными программами, которые выполняются в разных АП, причем в режиме пользователя.

Начнем с монолитного ядра.

Юникс линукс в отличии от винды....

Все монолитное ядро построено на прерываниях, как мы говорили , в ОС существует система прерывания, в современных системах принято выделять 2 типа прерывания, 1 системные вызовы, 2 – это исключения, 3 – аппаратные прерывания, ну только аппаратные прерывания называются интераптс (мой коверканный англ)

Когда хотят подчеркнуть, что ОС представляет интерфейс пользователя., то говорят об опи, при этом Опи определены в позикс

но например функция сигнал, в позикс не входит, но входит в анси-с, и от этого она не перестает быть ОПИ.

еще в середине 1960 годов, товарищи Меник и Доновал презентовали иерархическую машину, она отображает расположения соответствующих функций ядра по отношению к аппаратной части.

сейчас будет рисунок.....

Вич машина – это программно управляемое устройство (железка, аппаратура)

на последние минут 10 меня не хватило.....

Лекция 17

Каждый следующий слой получает не реальную машину, а виртуальную. Между ними – интерфейс – набор функций, которые могут вызывать верхний слой

Интерфейсы бывают непрозрачный, прозрачный, полупрозрачный

- Непрозрачный – функции верхнего слоя могут обращаться только к следующим по порядку слоя
- Прозрачный – через слой
- Полупрозрачный – часть – строго вниз, часть – и через слой

Денован свою иерархическую машину рассматривал с точки зрения распараллеливания ОС. Если интерфейс между уровнями определен – то можно так разрабатывать. Ненаписанные функции заменяются заглушками.

Нисходящее и восходящее программирование.

Иерархическая структура unix BSD 4.4

(Фото)

1. Символьный уровень
2. Именование
3. Отображение адреса
4. Страницочное прерывание

Юникс хорошо структурирована, видим в части процессов все то же, что у медника-денована (там только не было сигналов). Но в Юниксс 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 тч) 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).

Дальше – 34.

Под отображением адреса- преобразование виртуального адреса к физическому. Выполняется, когда процессор обращается к соответствующей команде – в последний момент. Ну и страницочное прерывание.

Именование и символьный уровень. Именование – дескриптор, inode.

Понятие терминал для юникс – базовое. В книге Стивен радио.

Микро ядерная архитектура строится на этой иерархии. Все более высокоуровневые – в виде отдельных процессов.

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

Последовательность действий в системе при запросе приложения на ввод/вывод (из книги Шоу – у нас с изменениями, названия без изменений)

(Фото)

Происходит вызов read/write и система переходит в режим ядра. Ни одна система не дает прямое обращение к внешним устройствам. Потому что защита. Это эквивалентно обращению к объектам ядра. (вместо прерываний пишем системный вызов).

Первое действие – системный вызов из функции, который переводит в режим ядра, туда же соответствующие данные. Super visor call . Супервизор - ... в стадии выполнения. IH- interrupt handler. В результате обработки системного вызова будет вызван драйвер внешнего устройства (программа ввода/вывода). Драйверу будут переданы данные в его формате. Драйвер инициализирует работу внешнего устройства. На этом управление процессором работы заканчивается, он отключается, потому что работой внешнего устройства управляют контроллеры. Инициализация его работы (если канальная архитектура – то канал инициализирует работу внешнего устройства). Тем не менее начиная с 3 поколения реал распараллеливание – управление внешними устройствами передано контроллерам (шинная арх) и каналам (канал).

По завершении операции ввода/вывода будет сформировано контроллером устройства прерывание, которое в простой схеме поступит на контроллер прерывания и в результате будет определен адрес точки входа обработчика прерывания и он начнет выполняться (IO IH). Процессор перейдет на выполнение обработчика, так как аппаратные прерывания имеют наивысший приоритет. ОП входит в состав драйвера и является одной из очек входа драйвера внешнего устройства. Драйвер всегда содержит 1 обработчик прерывания.

D3.

Поскольку обработчик – точка входа драйвера, у него есть callback функция – задача вернуть запрашиваемые данные приложению. В результате драйвер через подсистему ввода/вывода должен передать данные приложению. Для этого работа приложения должна быть возобновлена. Процесс, который запросил ввода, блокируется.

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

Чтобы вернуть приложению запрошенное значение, оно должно продолжить выполняться. (7) – разбудить.

То на этой схеме показана полная последовательность действий при запросе приложения на ввод, и как это обрабатывается ядром.

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

Микроядерная архитектура

Идея мя – оставить в ядре только самые низкоуровневые функции. Поскольку остальная часть ОС реализуется в виде отдельных процессов в собственных АП, то обеспечение взаимодействия этих процессов – основная функция.

(фото)

Если оставлены только самые низкоуровневые. Остальные ф выполняются в режиме пользователя. Современные ОС предоставляют возможность внесения в монолитное ядро собств... 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). В ... - просто сервера. Взаимодействие микроядерной 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.

Сообщение сопровождается сообщением сообщь принят. Ни один запрос не может остаться без ответа. В состав таких серверов входит сервер памяти, сетевой сервер, сервер безопасности, файловый сервер, сервер процессов. При этом могут также общаться между собой.

(фото)

Такое взаимодействие должно быть надежным – посылка сообщь 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 (производительность) мя архитектуры

В монолитном ядре при обработке системного вызова будет выполнено 2 переключения – 1 из режима задачи в режим ядра, 2 – обратно. (А на предыдущем фото – как минимум 4)

(2 фото).

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

Mach – первая такая ОС

Несмотря на то, что эффективность мя архитектуры по сравнению с ... 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.

POSIX определение реального времени в ОС (стандарт 1003.1) :

РВ в ОС – это способность ОС обеспечить требуемый уровень сервиса за определенный промежуток времени.

Задачи РВ составляют одну из сложнейших и важнейших областей применения вычислительной техники. СПЕЦИАЛЬНЫЕ задачи по управлению внешними объектами или процессами по отн к выч системе. Вот для них мя открывает широкие возможности.

ОС mach

Как и любая ОС, она определяет основные абстракции, на которых базируется ее работа. 5 основных абстракций

1. Процессы
2. Threads – потоки

И понятие процесс, и понятие поток не отличается от классического определения. Поток независимо планируемый контекст (аппаратный), своего АП не имеет, выполняется в АП процесса

3. Объекты памяти
4. Порты
5. Сообщения

Отличительная особенность – понятие объект памяти. Memory object – представляет собой структуру данных, котороая мб отображена в АП процесса. Объекты памяти могут занимать одну или несколько страниц и являются основой для системы управления виртуальной памятью. Когда процесс обращается к объекту памяти, который отсутсвует физической памяти, возникает страничное прерывание. Ядро его обрабатывает, но в отличие от других систем, ядро для загрузки страницы посылает сообщение серверу режима пользователя, только после того, как запрос будет обработан, ядром будет полуен ответ от сервера, соотв страница мб загружена в ОП

Межпроц вzd в этой ОС основано на передаче сообщений. Чтобы передавать сообщения, процесс пользователя просит ядро создать защищенный почтовый ящик, который в мач называется порт. Порт создается в АП ядра и способен поддерживать очередь сообщений.

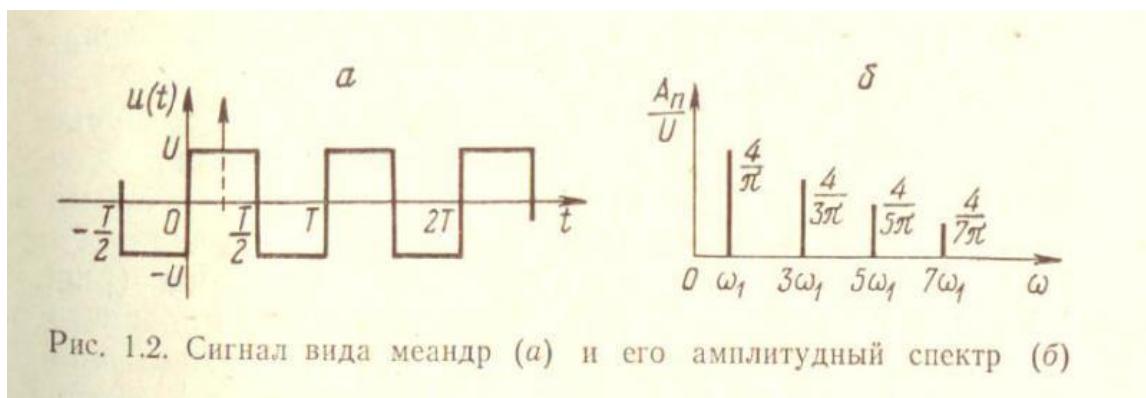
Как правило, очередь имеет опр размер и если процесс пытается послать в нее сообщ, а она уже заполнена, то такой процесс будет блокирован д того момента, когда порт не будет разгружен от этих сообщений. В мач поддерживаются разные типы портов. Такое взд сложное и дб надежным.

- Порт процесса (исп процессом для взд с ядром) – тужа посыает запросы на обслуживание (с пом сообщений (как в `remote proc call` – не надо знает все тонкости обработки запросов) – внешне выглядит как системный вызов)
- Порт загрузки. Используется при старте системы. Самый первый процесс 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 исключений.
- Зарегистрированные порты – используются для обеспечения взд процессов со стандартными системными серверами

Семинар 0

Прерывания по тику (системному таймеру)

Тактовая частота – меандр (наверху 1,5В). Но длительность подъема и спада неодинакова. Подъем скруглен.



Импульсы – тики, приблизительно 18,2 раза в секунду (точное значение - 1193180/65536 раз в секунду – то есть чуть больше).

Основа – компьютеры работают от электричества, в них передаются сигналы. Сигналы 3 типов:

- Данные (собственно данные и команды)
- Адреса
- Сигналы управления

Существует концептуально 3 соответствующие шины. Так как частота высокая, на самом деле – литая пара.

Компьютер – это процессор и оперативная память, а все остальное – внешние устройства. Они взаимодействуют с помощью прерываний.

Программируемый контроллер прерываний PIC (Programmable Interrupt Controller)

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.

По завершению 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 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

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.

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 msi сообщениями, данных, адресов.

2 адресных пространства – оперативная память и порты ввода/вывода.

отвечает за приём запросов прерываний от различных устройств, их хранение в ожидании обработки, выделение наиболее приоритетного из одновременно присутствующих запросов и выдачу его вектора в процессор, когда последний пожелает обработать прерывание. Слово «программируемый» в названии контроллера означает, что режимы его работы устанавливаются программно, а не являются жёстко «зашитыми».

(https://osdev.fandom.com/ru/wiki/Программируемый_контроллер_прерываний#.D0.9F.D1.80.D0.B8.D0.BD.D1.86.D0.B8.D0.BF.D1.8B_.D1.80.D0.B0.D0.B1.D0.BE.D1.82.D1.8B)

Зубков

Существует два контроллера прерываний. Первый контроллер, обслуживающий запросы на прерывания от IRQ0 до IRQ7, управляемся через порты 20h и 21h, а второй (IRQ8 - IRQ15) - через порты 0A0h и 0A1h.

если несколько прерываний происходят одновременно, обслуживается в первую очередь то, 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.

mov al,20h ; команда "неспецифичный конец прерывания"

out 20h,al ; посыпается в первый контроллер прерываний

Системные прерывания:

- Системные вызовы

Системные вызовы (system calls) - это интерфейс между операционной системой и пользовательской программой. (INT).

- Исключения

Исключительная ситуация (exception) - событие, возникающее в результате попытки выполнения программой команды, которая по каким-то причинам не может быть выполнена до конца.

- Аппаратные прерывания (interrupt)

Прерывание (hardware interrupt) - это событие, генерируемое внешним (по отношению к процессору) устройством. Посредством аппаратных прерываний аппаратура либо информирует центральный процессор о том, что произошло событие, требующее немедленной реакции (например, пользователь нажал клавишу), либо сообщает о завершении операции ввода вывода (например, закончено чтение данных с диска в основную память). Каждый тип аппаратных прерываний имеет собственный номер, однозначно определяющий источник прерывания. Аппаратное прерывание - это асинхронное событие, то есть оно возникает вне зависимости от того, какой код исполняется процессором в данный момент. Обработка аппаратного прерывания не должна учитывать, какой процесс или поток является текущим.

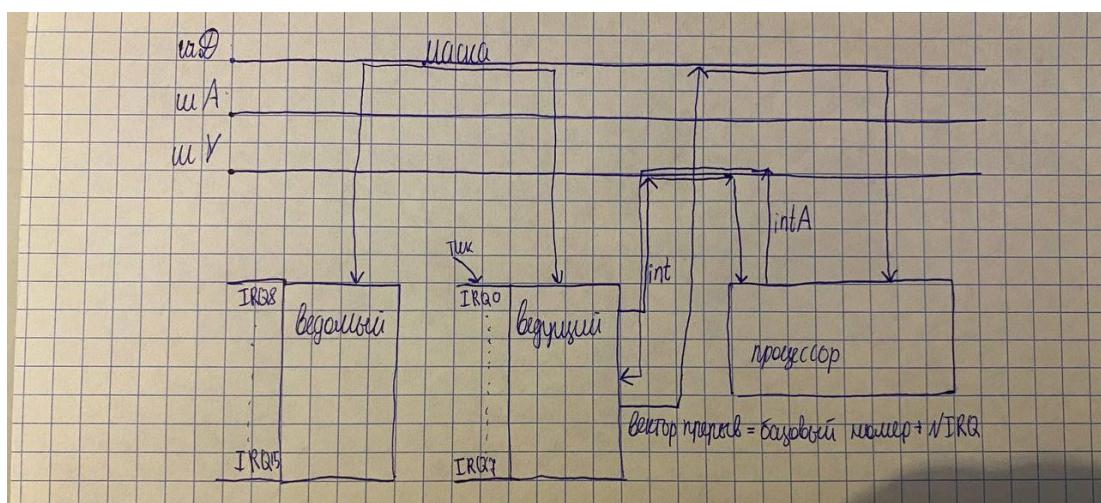
Что поступает на IRQ:

- 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 посыпает intA по шине управления в контроллер. Контроллер выставляет на шину данных вектор прерывания, который поступает в процессор и используется им для адресации обработчика прерывания.

просто прерываниями. В принципе обработка прерывания выполняется, за исключением своей начальной аппаратной стадии, так же, как и обработка исключения: при поступлении на вход INT микропроцессора сигнала от внешнего устройства процессор считывает с шины данных выставленный контроллером прерываний номер вектора, находит в таблице дескрипторов прерываний дескриптор с соответствующим номером и, сохранив предварительно в стеке адрес возврата, осуществляет переход на обработчик прерывания. Команда ret, которой заканчивается обработчик прерывания, возвращает управление в программу. Таким образом, для обработки аппаратных прерываний мы должны дополнить таблицу IDT дескрипторами обработчиков аппаратных прерываний и включить сами обработчики в текст программы.



Вектор прерывания=базовый вектор + №IRQ

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 в порт команды наброс.

Смотри про 8h вво вся теория

Семинар 1

Режимы

Компьютеры на базе процессоров Intel работают в 3 режимах:

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

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)

- Максимально возможный размер сегмента в реальном режиме - 64КБ
- Минимальная адресная единица памяти – байт.

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

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.

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

Почему «защищенный»: Windows – система разделения времени. Адресное пространство каждого процесса должно быть защищено, как и адресное пространство ОС.

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. Режима v86 нет!

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

Группы регистров

1. Регистры общего назначения (32)

РОны		
31	15	0
EAX	AH	AL
EBX	BH	BL
ECX	CH	CL
EDX	DH	DL

Аппаратно доступна младшая часть, в которой доступны младшая и старшая части (?Док-во того, что аппаратно поддерживается реальный режим)

2. Индексные и указательные регистры (32)

	31	15	0
ESI		SI	
EDI		DI	
ESP		SP	
EBP		BP	

3. Сегментные регистры (16)

В защищенном режиме 6 сегментных регистра, в реальном режиме 4.

CS, DS, SS, ES, FS, GS

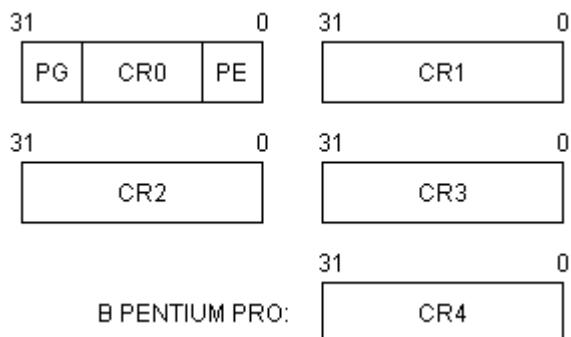
4. Регистры системных адресов (регистры управления памятью)

При работе в защищенном режиме микропроцессора адресное пространство делится на глобальное – общее для всех задач; локальное – отдельное для каждой задачи.

- 1) GDTR(32, в РФ-6байт) (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 Idtr, содержит селектор, т. е. указатель на дескриптор в таблице GDT. Для переключения задач.

Линейный адрес - физический адрес оперативной памяти

5. Управляющие регистры (32)



- CR0 – регистр слова состояния.

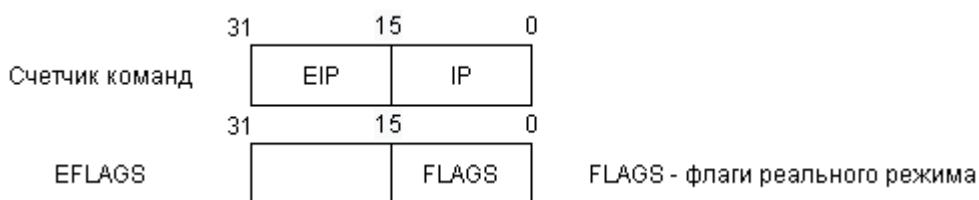
Содержит 6__(7) флагов

- 0 PE (protection enable) - определяет, в каком режиме работает комп. Если установлен, значит в защищенном, нет – в реальном

- 31 PG (paging enable). Если установлен, значит выполняется страничное преобразование. Но это имеет смысл только в защищенном режиме, т.е PG=1 имеет смысл только при PE=1
- CR1 – не исп.
- CR2 - регистр линейного адреса ошибки обращения к странице (страничная неудача) – исключение, которое возникает, когда процессор обращается к странице, отсутствующей в физической памяти. На какой команде это произошло, ее адрес будет занесен.
- CR3 - регистр начального адреса каталога таблиц страниц. ЕГО НАЛИЧИЕ – доказательство того, что управление таб. по запросу независимо от сегмента (те 2 схема, а не 3) (чисто страничное преобразование). А флаг PG не является доказательством.

В LONG только PAE Physical Address Extension (расширение физического адреса). Включается же PAE установкой пятого бита регистра CR4 в единицу. С помощью PAE 32битный указатель позволяет адресовать до шестидесяти четырех (2^{36}) гигабайт физической памяти.

6. 32-p: Тестовые, отладочные регистры, EIP, EFLAGS



19-VIF флаг виртуального прерывания. *используется совместно с флагом VIP и позволяет обеспечить нормальное выполнение старого ПО, использующего команды управления внешними маскируемыми прерываниями (векторы от 32 до 255), в современной мультипроцессорной и мультизадачной программно-аппаратной среде.*

20-VIP Флаг ожидания виртуального прерывания

17 VM – Virtual 8086 mode (386+)

В режиме long – RFLAGS, но все не используются.

Переход в защищенный режим.

Установить флаг PE не достаточно чтобы прейти в защищенный режим - нужно провести некоторые приготовления. Программам нужны некоторые ресурсы, чтобы они могли исполняться - это может быть память, процессорное время, регистры процессора, порты ввода/вывода.

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 защищенном же режиме есть поддержка виртуальной памяти, в том числе и на аппаратном уровне.

Управление памятью, сегментами в защищенном режиме.

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

Регистры системных адресов - чтобы поддерживать эти таблицы.

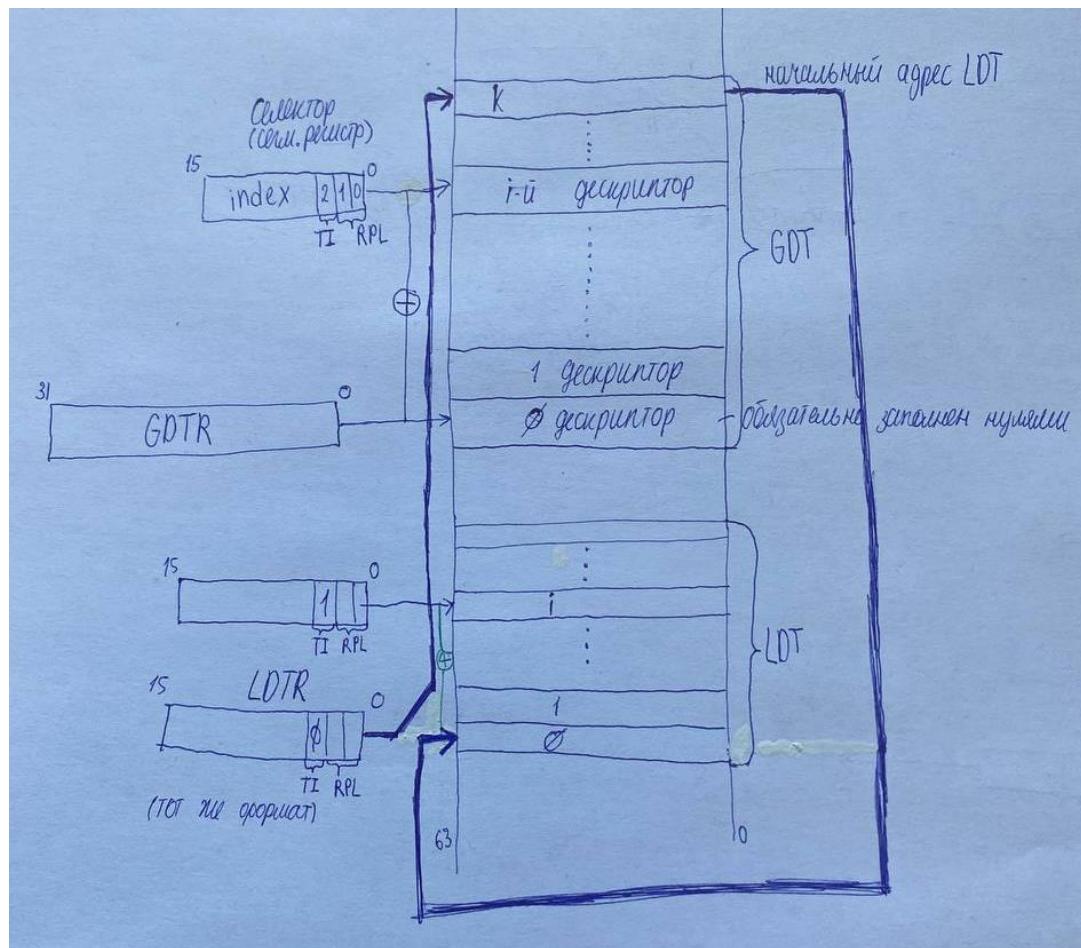
- GDTR
- IDTR
- LDTR
- TR

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

Системные таблицы

Глобальная таблица дескрипторов

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

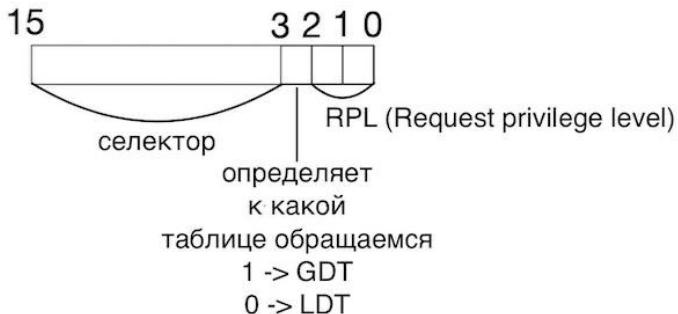


Так как одна память, то GDT в системе одна. Она находится в памяти ядра системы. На начало таблицы указывает GDTR (32).

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

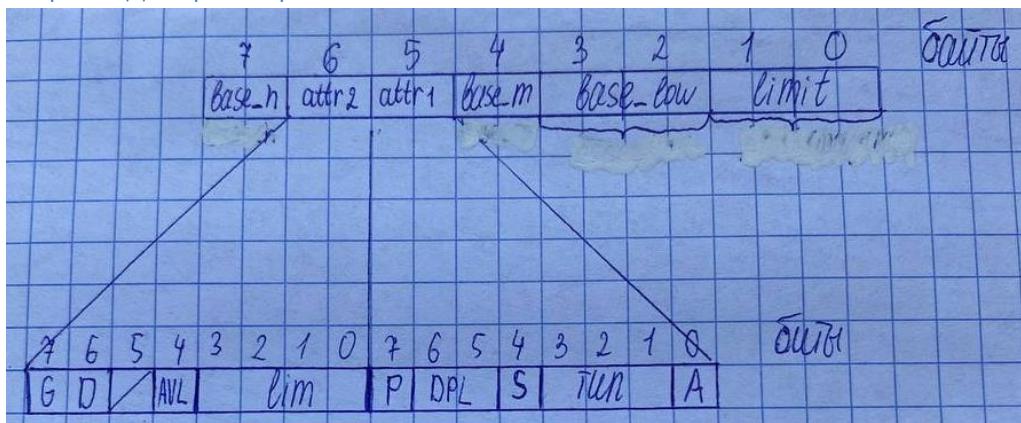
Формат селектора.

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)

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_heigh): 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.*4K+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).

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)

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.

Начинается с нулевого дескриптора.

См. рисунок выше

Формат дескрипторов абсолютно такой же.

В LDTR хранится селектор дескриптора глобальной таблицы, по этому смещению хранится дескриптор сегмента LDT. То есть у нас локальная таблица дескрипторов находится в сегменте, который уже должен быть выделен в GDT.

GDT размещается в защищенной области памяти, к которой имеет доступ только ядро операционной системы.

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

Таким образом мы берем из GDT базовый линейный адрес и получаем начало таблицы LDT.

Например, мы загрузили в DS селектор из LDT, то процессору нужно сначала найти сегмент где хранится LDT, получить ее базовый линейный адрес и уже тогда к нему прибавить смещение из селектора. Получим снова базовый линейный адрес, к которому прибавляем смещение - получаем соответствующий линейный адрес.

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

Семинар 2

GDT позволяет адресовать память. В архитектуре ФН все адресуется, команды и данные в оперативной памяти, доступ по адресу, который нужно сформировать.

В ЗР (под 9) 2 схемы управления памятью:

- сегментами по запросу
- страницами по запросу.

Они предполагают выполнение преобразований.

- Сегментированный адрес – не прямая аналогия с DOS. В DOS это связано с увеличением до 20р шины данных, и адрес вычисляется как сдвинутый сегментный регистр + смещение.
- В ЗР – полноценно: есть начальный адрес сегмента, к которому добавляется смещение (смещение берется из команды). Любая программа считает, что она начинается с 0 адреса, следовательно, смещение берется из команды. Это счетчик команд, или индексный регистр, или указательный регистр, или операнд из команды. Это прибавляется к адресу сегмента, и мы получаем линейный физический адрес.

Intel поддерживает LDT, но LDTR 16 разрядный – не может содержать линейный физический адрес. Но по аналогии с сегментными регистрами – селекторы к дескрипторам GDT, и они уже описывают сегменты памяти. LDT описывает виртуальное адресное пространство, следовательно, их столько, сколько процессов. (Но мы процессы не создаем в ЛР)

GDT и IDT – системные таблицы.

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 РР, следовательно, смещение в РР не может превышать 2^16. FFFF=2^20=1Мбайт (мое с семинара, но что-то странное)

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.

(в ЛР тк в РР смещение <=ffff, ни один сегмент не выходит за границы (видимо, про ненужную загрузку ffff во 2 ЛР))

Определение объема доступного физического адресного пространства

Max=4ГБ в ЗР. Чтобы его адресовать, надо объявить.

Потому что в 1 мб лежит ROM BIOS и мы его перезаписываем и происходит ошибка прав доступа. По-моему она говорила ROM Bios

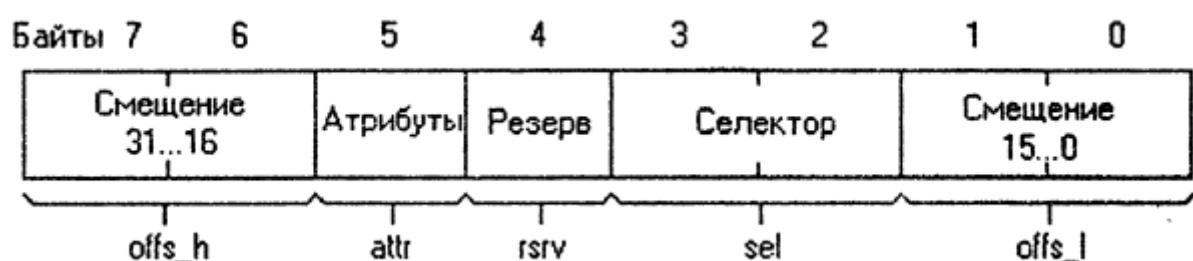
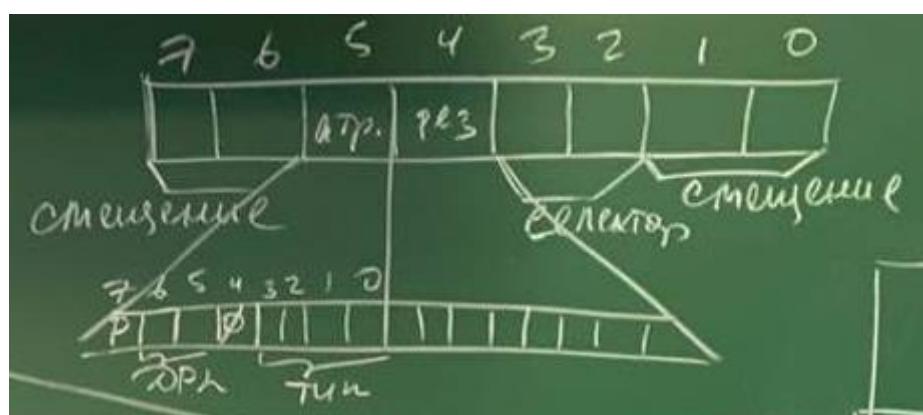
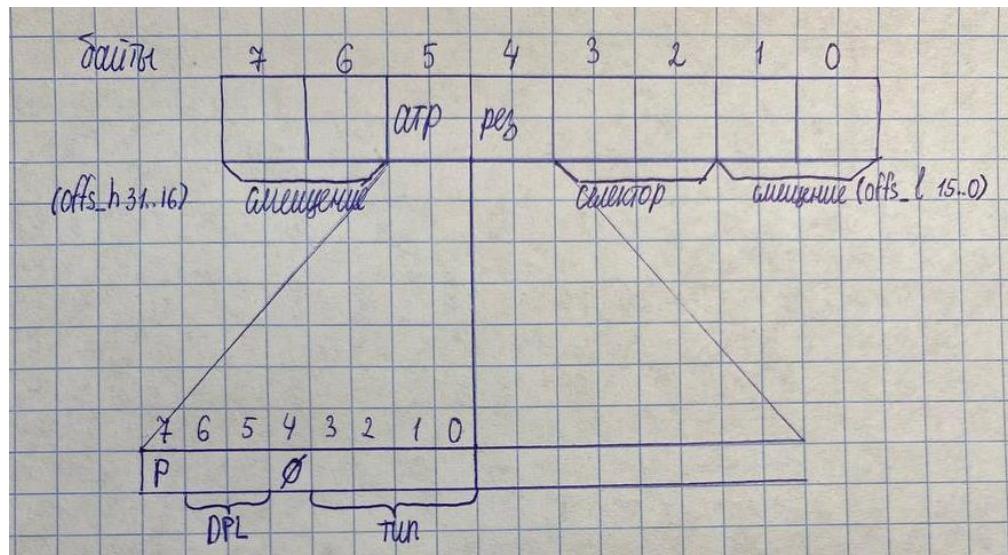
1Мб пропускаем (там наша программа или *В первом мегабайте хранится операционная система ROM BIOS* - спорно), со второго сохраняем байт/слово, записываем туда сигнаттуру, сравниваем со своей, если совпали-память и инкремент счетчика. Сегмент данных, limit=fffff, G=1, чтение и запись.

В GDT дб описаны: 16р кода (тк стартуем в 16р), 16р данных (в РР описываем там таблицы), 32р кода (тк перешли), 32р данных (для определения памяти), 32р стек (для прерываний в ЗР)(хотя при возвращении надо бы вернуться к новому). В ЗР 2 обработчика прерываний-от клавиатуры и системного таймера.

Прерывания в ЗР

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(сег. 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:

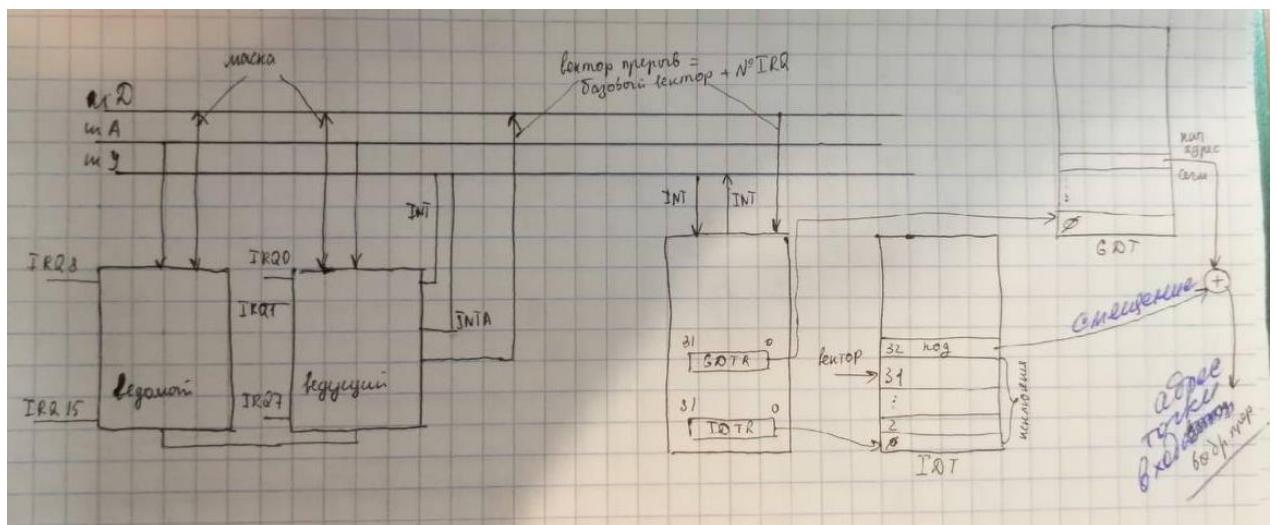
- 0-не определен
- 1-свободный сегмент состояния задачи TSS 80286
- 2-LDT
- 3-занятый сегмент состояния задачи TSS 80286
- 4-шлюз вызова Call Gate 80286
- 5-шлюз задачи Task Gate
- 6-шлюз прерываний Interrupt Gate 80286
- 7-шлюз ловушки Trap Gate 80286
- 8-не определен
- 9- свободный сегмент состояния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

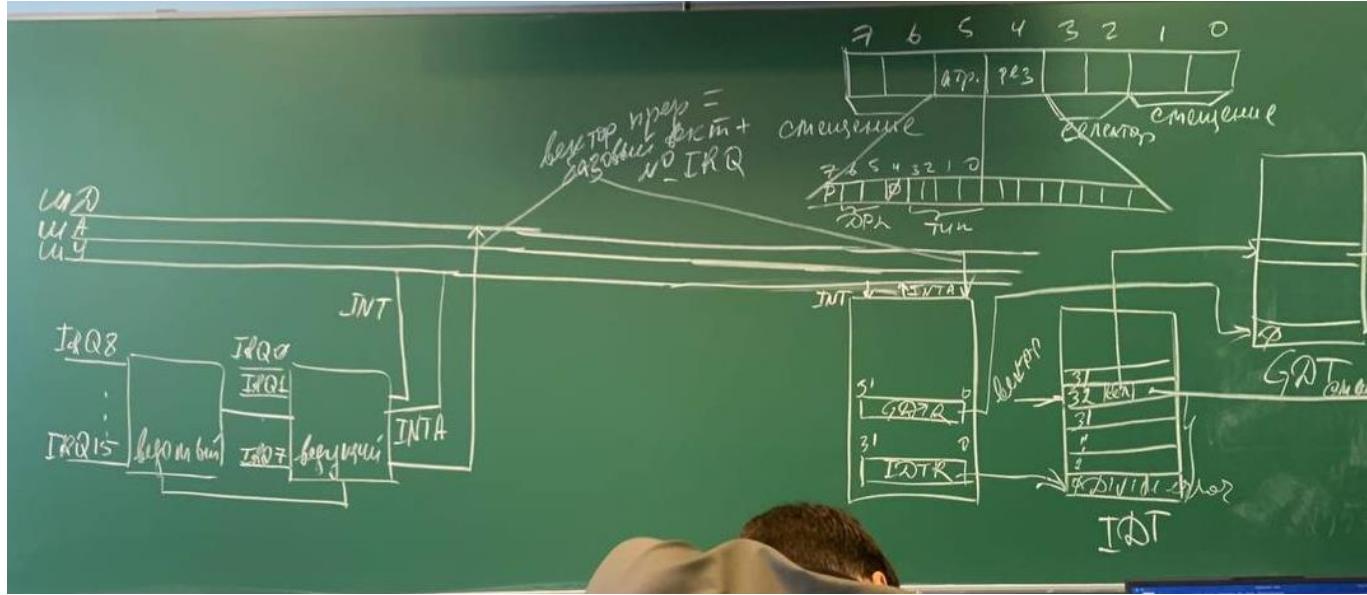
Адресация исключений, аппаратных прерываний и системных вызовов (программных прерываний в ЗР). Исключения и системные вызовы-TRAP. Когда говорят interrupt без ничего – аппаратное прерывание

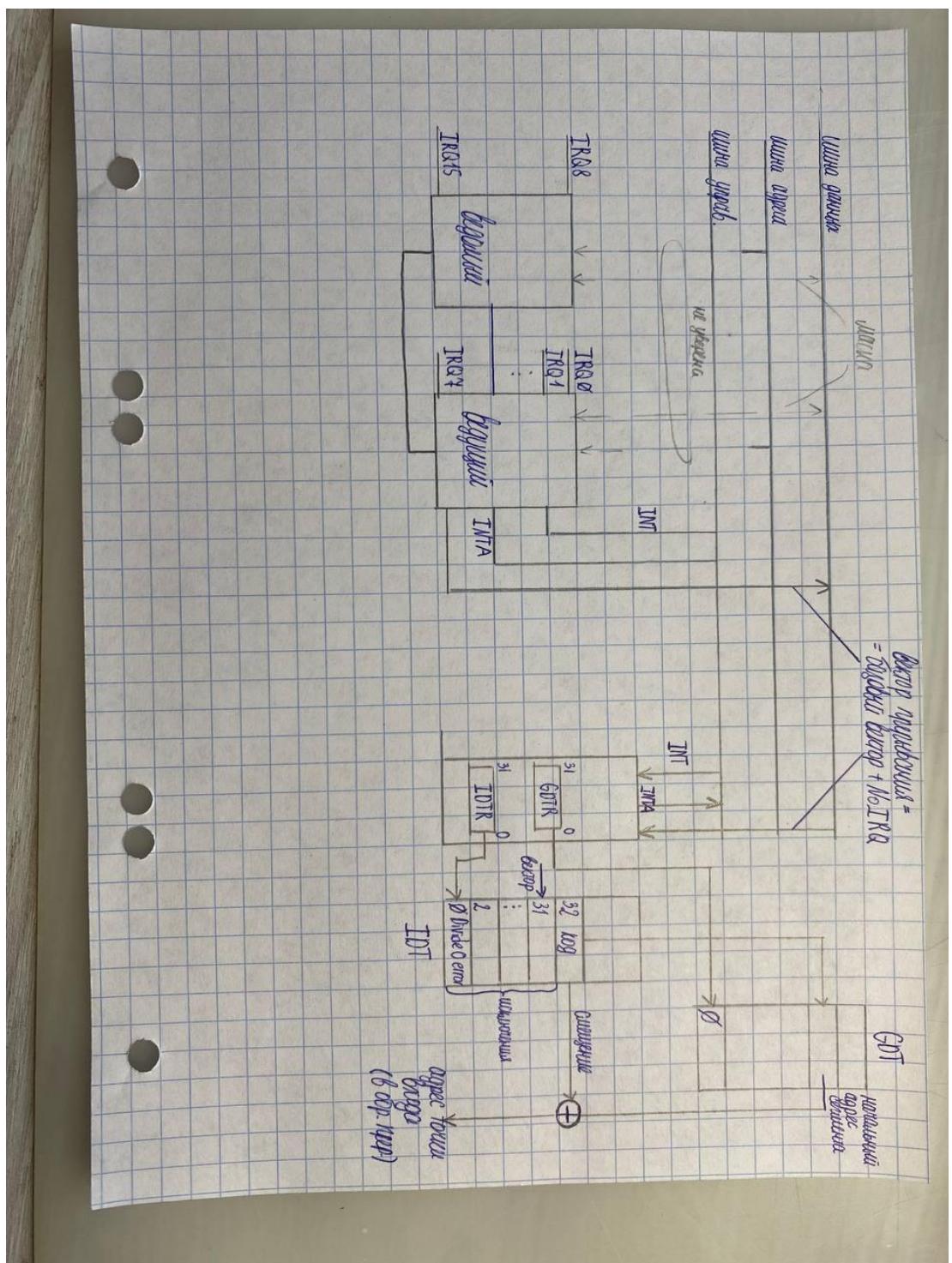
через шлюзы прерываний (interrupt) обрабатываются аппаратные прерывания, ловушек (TRAP)-программные прерывания (системные вызовы) и исключения

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) через шлюзы прерываний обрабатываются аппаратные пр., ловушек-программные пр. и искл. (это по РФ)

- 4-пустой, 5-6-DPL, 7 P (1)







(шедевральный рисунок Рязановой)

(говняный рисунок Иры с божественными подписями Рязановой)

(точка входа)

(к рисункам выше)

IDTR-32 разрядный. Содержит начальный линейный адрес IDT (все эти регистры в каждом ядре).
 Нулевой регистр не пустой, как в GDT, а деление на 0.

IDT:

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, на рисунке-для 486))

32-255 – определяются пользователем (user defined)

- 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	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	Нарушение	Нет	esc, wait
	Недоступен			
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			

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)

Схема:

(Если прерывание не замаскировано)

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 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??

Вопросы ко 2 ЛР

- какую программу вы написали

В защищенном режиме написали управляющую программу с 0 уровнем привилегий. Данная программа выполняет две функции ОС: переход в защищенный режим и выделение сегментов памяти, описанных в глобальной таблице дескрипторов и размер, которых мы подсчитываем в программе

Программа 0 уровня привилегий - фактически часть ОС или даже своя ОС.

где в двух местах говорится что у нас 0 уровня привилегии

DPL и RPL

Где находится rpl ?

В селекторе.

- что пришлось создать в этой программе раз это такая программа

Создать две системные таблицы – глобальную таблицу дескрипторов сегментов (GDT) для описания сегментов физической памяти, с которыми будет работать запущенная программа и таблицу дескрипторов прерываний (IDT), в которой заполняются дескрипторы прерываний, которые необходимы для выполнения поставленной задачи.

- где явно установлен соответствующий уровень привилегий

То, что мы используем системные таблицы.

GDT размещается в защищенной области памяти, к которой имеет доступ только ядро операционной системы. Она находится в памяти ядра системы.

МБ еще команды lidt, lgdt

привилегии (4 кольца защиты) != приоритет (приоритет процессов потоков, который назначается и потом может быть пересчитан)

- какие сегменты вы описали в глобальной таблице и почему (для чего)

1. 16-разрядный сегмент кода, граница в байтах (G=0)(для реального режима)
2. 32-разрядный (D=1) сегмент данных, размер: 4 Гб (для определения объема выделенной памяти)
3. 32-разрядный (D=1) сегмент кода, граница в байтах (G=0);(для защищенного режима)

4. 32-разрядный (D=1) сегмент данных, граница в байтах (G=0) (от нас там данные всякие, рязанова хотела 16-р)
5. 32-разрядный (D=1) сегмент стека, граница в байтах (G=0);(для прерываний в ЗР)
6. видеобуфер (0,0)

- охарактеризовать дескриптор сегмента дополнительной памяти, который описали

`gdt_data4gb descr <0FFFFh,0,0,92h,0CFh,0>`

- limit=fffff=2³²=4Гб
- (G=1 (7 бит 6 байта attr2) граница в блоках по 4 Кбайт размер: 4 Гб
- (D=1 (6 бит 6 байта attr2)32-разрядный по умолчанию
- attr_1=92h=10010010b:
 - 0 бит A -не интересует
 - 1-3 тип:
 - 3 бит=0 – сегмент данных/стека
 - 2 бит=0 сегмент данных
 - 1 бит=1 разрешены чтение и запись,
 - 4 бит S=1 является сегментом памяти
 - 5-6 DPL=0 - 0 уровень привилегий
 - 7 бит S=1 присутствует в памяти,

- почему таблица дескрипторов прерываний имеет такую структуру

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

исключение 13 - нарушение общей защиты (нарушение, код ошибки-та команда) происходит: за пределами сегмента, запрет чтения, за гр. таблицы дескр., int с отс. Номером

32-255 – определяются пользователем (user defined)

Смещение=номер исключения*8

Затем 16 векторов аппаратных прерываний (асинхронные),

- что написали для исключений
заглушки
- как адресуются аппаратные прерывания в ЗР (ударный)

Контроллер прерывания получает сигнал о прерывании и формирует вектор прерывания, который содержит селектор к IDT.

Взяв значение из регистра IDTR значение базового адреса IDT. В нём по селектору находим дескриптор который уже и содержит селектор, смещение и атрибуты

По селектору выбираем дескриптор из ТГД, берём оттуда базовый адрес сегмента и прибавляем его к смещению

Получаем линейный адрес точки входа

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.

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)

Таким образом, обработчики для аппаратных прерываний должны начинаться минимум с 32-го (в винде принято начинать обработку прерываний со смещения 50h, но мы на это забьём, чтобы не пилить ещё дофига лишних обработчиков).

- когда вызывается обработчик от клавы? таймера?

Показать обработчик клавиатуры

Что этот обработчик анализирует? Скан коды, приходящие с клавиатуры.

Когда вызывается? Вызывается при нажатии или отжатии кнопки на клавиатуре.

Что мы можем использовать в своем обработчике прерываний от клавиатуры? Только порты, так как мы работаем с голым железом

Когда вызывается наш обработчик прерывания от системного таймера?

По тику (18.2 раза в секунду)

- **какие действия необходимо выполнить для корректного возвращения в РР**
 - запретить маскируемые прерывания cli
 - сбросить влаг PE (0) в CR0-слово состояния машины
 - загрузить в используемые сегментные регистры адреса соответствующих сегментов к регистру CS недопустимо прямое программное обращение, поэтому юзаем переход
 - перепрограммировать ведущий контроллер на 8 обратно
 - восстановить маски ведущего и ведомого (из сохраненных)
 - загрузить idtr
 - закрыть линию A20
 - разрешить маскируемые прерывания sti

что пишем в теневые регистры и почему.

после перехода в защ. режим прога не должна работать, т.к. в регистре CS ещё нет селектора сегмента команд и процессор не может обращаться к этому сегменту. в действительности это не совсем так. в процессоре для каждого из сегментных регистров имеется так называемый теневой регистр дескриптор, который имеет формат дескриптора. теневые регистры недоступны программисту. они автоматически загружаются процессором из таблицы дескрипторов каждый раз, когда процессор инициализирует соответствующий сегментный регистр. таким образом в защ. режиме пр-мист 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. после перехода в защ. режим прежде всего

следует загрузить в используемые сегментные регистры селекторы соответствующих сегментов. это позволит процессору правильно заполнить все поля теневых регистров из таблиц дескрипторов. к регистру CS недопустимо прямое программное обращение, поэтому юзаем переход. при работе в реальном режиме некоторые поля теневых регистров должны быть заполнены определенным образом. граница fffffh, бит дробности 0, доступ для записи разрешен. границы всех сегментов должны быть точно равны fffffh. перед переходом в реальный режим необходимо исправить все дескрипторы всех наших сегментов. линия a20 для обращения к расширенной памяти.

- что такое теневые регистры, для чего они включены в процессор, какую информацию содержат

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. 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.

таким образом в защ. режиме пр-мист 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

Процессоры начиная с 80386 начали поддерживать защищенный режим - для этого в них появились теневые регистры, вся адресация памяти теперь проходит через них. В РЕАЛЬНОМ режиме для обеспечения обратной совместимости при загрузке в CS,DS,SS адреса сегмента происходит неявное создание дескриптора сегмента. MOV AX,DataSeg MOV DS,AX Где-то на этом участке происходит определение размера сегмента DataSeg, и создание его дескриптора (8 байт). После этого полученные 8 байт сразу же загружаются в теневой регистр (быдлоговоря, DS_shad). В ЗАЩИЩЕННОМ режиме "основной" сегментный регистр для адресации используется не целиком: его младшие биты (0,1,2) содержат в себе флаги (за описаниями к Р. и Ф.). В качестве смещения используются биты 3..7. Другими словами, если в регистре лежит значение 8d (00001000b), то смещение == 00001b. При этом смещение всегда обязательно начинается с 1 - по смещению 0 лежит пустой дескриптор - его адрес используется для определения, а где же в памяти лежит сама GDT (Global Descriptor Table).

- Немного про линию A20

В защищенном режиме определяют порт линии A20 - 21-ая адресная линия. Компьютер стартует в реальном режиме. Связано это со стремлением хранить в энергонезависимой микросхеме ...

(предполагаю что тут "как можно меньший объем информации"). ОС - программа, и пока компьютер выключен, она хранится во внешней памяти.

Когда компьютер стартует, линия A20 принудительно обнулена. При переходе в защищенный режим нужно открыть линию A20, то есть снять заземление.

- Зачем мы открываем линию A20?

Открываем, чтобы получить все адреса. Иначе у нас недоступны любые адреса, имеющие в 20 бите 1 - она обращается в ноль (линия заземлена, любой сигнал ноль)

открытие линии A20

Перед переходом в защищенный режим (или после перехода в неё) следует открыть линию A20, т.е. адресную линию, на которой устанавливается единичный уровень сигнала, если происходит обращение к мегабайтам адресною пространства с номерами 1, 3, 5 и т.д. (первый мегабайт имеет номер 0). В реальном режиме линия A20 заблокирована, и если значение адреса выходит за пределы FFFFh, выполняется его циклическое оборачивание (линейный адрес 100000h превращается в 00000h, адрес 10001h в 00001h и т.д.). Открытие (разблокирование) линии A20 выключает механизм циклического оборачивания адреса, что позволяет адресовать расширенной памяти.

Управление блокированием линии A20 осуществляется через порт 64h, куда сначала едету посыпать команду D1h управления линией A20, а затем - код открытия (DFh). За вентиль линии A20 отвечает 1-й разряд порта; остальные разряды изменять нельзя.

```
in al, 92h
```

```
or al, 2
```

```
out 92h, al
```

- Что будет, если при переходе в защищенный режим не откроем линию A20?

Нам не доступны адреса, в которых 20-ая адресная линия 1.

Если не включить линию A20, то 20ый бит всегда будет равен нулю. Если мы захотим обратиться к адресам, в которых этот бит равен единице, то мы не сможем получить к ним доступ. Мы получаем «битую память»

- Можно ли её не закрывать при переходе в реальный режим? что произойдет, если при возвращении в РР забудем закрыть линию a20

Если в реальном режиме открыть линию A20, то в реальном режиме станет доступно еще 64 Кбайта памяти - HMA, high memory area.

закрытие линии A20 (если не закроем, то сможем адресовать еще 64кб памяти

```
in al, 70h
```

```
and al, 7Fh
```

```
out 70h, al
```

Ничего страшного не произойдет, если мы так сделаем.

- 64 мб это какая память?

память кот. мы выделили под вирт. машину. (Нам недоступна вся оперативная память.)

- В сегменте кода можно выполнять read - write?

Можно,

- а в стек можно делать r-w?

да

- что делает 13 dup? Выделяет, но как? Почему не выделяем подряд 13 дескрипторов, почему так написали? Дублируем
- могли бы 32 строки не выделять под исключения? Что нужно обеспечить, если мы отказываемся от этого?

Ну и запретить аппаратные прерывания видимо

перепрограммировать контроллер. Базовый вектор меняется с8 на 32 и обратно.

- Для этого что они перепрограммируют? в результате формируемый вектор прерываний является. Чем является базовый вектор прерываний?

смещение в табл. дескр. прер.

- «Чему кратно значение селектора?»

«8»

- «Что такое индекс?»

«Смещение в таблице дескрипторов.»

- «В реальном режиме есть привилегии?»

«Нет»

- «Как вы считаете объем оперативной памяти?»

Почему мы не можем использовать первый мегабайт памяти?

В первом мегабайте хранится операционная система ROM BIOS. Это зона ROM - read only memory. Если попытаться что-то в него записать, то возникнет ошибка.

-«У нас есть сегмент, объемом в 4 гигабайта. Считываем значение из ячейки памяти. Сохраняем его. Записываем в ячейку свое значение. Считываем. Если записанное и считанное значения совпадают, то это память.

Возвращаем в ячейку первоначальное значение. Если значение не совпадают, то пустота. Мы действуем до первого прокола»

- «Какой формат дескриптора?»
- «Какие характеристики имеет сегмент «памяти»?»

- «Почему вы в реальном режиме используете привилегированную команду lgdt?»

«Мы можем это делать, т.к. используем директиву .386»

ряд вопросов по коду, такие как

«Покажите где вы входите и выходите из защищенного режима», «Покажите какие сегменты вы описали»,

«Что вы делаете до входа и после входа в защищенный режим».

- Почему вдруг ваша программа стала выполнять функции ОС?

Потому что наша программа работает в защищенном режиме с нулевым уровнем привилегий.

- Есть у вас системные вызовы?

Нет.

- Если у вас нет системных вызовов, то какие средства вы используете для того, чтобы вводить и выводить символы?

Для этого мы используем бесконечный цикл. Для того, чтобы вывести символ мы напрямую обращаемся к видеопамяти, т.к. у нас нет ДОСовой команды вывода страницы.

- Почему существует деление на простые команды и привилегированные?

Привилегированные команды могут выполняться только с нулевым уровнем привилегий.

Это нужно для того, чтобы какая-нибудь программа не смогла нарушать работу системы, выполнив привилегированную команду.

- Что использую в обработчике прерываний от клавиатуры?

Порт клавиатуры

- В чём особенность реального режима?

особенность реального режима - ограничение объёма адресуемой оперативной памяти величиной 1мб.

перевод в защ. реж =

- увеличие адресуемого пространства до 4гб.
- возможность работать в виртуальном адресном пространстве.
- организация многозадачного режима с параллельным выполнением нескольких программ(процессов)
- страничная организация памяти, повышающая уровень защиты задач друг от друга.
- в защищённом режиме процессор выполняет процедуру прерывания не так как в реальном. при поступлении сигнала прерывания процессор не обращается к таблице векторов прерываний в первом кб памяти, как в реальном режиме, а извлекает адрес программы обработки прерывания из таблицы дескрипторов прерываний, построенной схоже с ТГД.

- Привилегированные команды.

тридцати двухразрядные микропроцессоры отличаются расширенным набором команд, часть которых относится к привилегированным. Для того чтобы разрешить транслятору обрабатывать эти команды в тексте включена директива .386P

Просто кусок информации

На процессорном уровне поддерживаются уровни привилегий 0..3; ОС запускается с уровнем 0, приложения с 3. Для каждого сегмента памяти устанавливается уровень привилегий, необходимый для доступа. Существует три режима работы процессора: реальный, защищенный и виртуальный-86. Процессор всегда запускается в реальном режиме и выполняет код биоса. После этого он путём плясок с бубном и скакания на [s]майдане[/s] костылях переходит в защищённый.

Реальный режим: однозадачный, 16битные регистры, поддерживает до 1 мегабайта памяти. НЕ ПОДДЕРЖИВАЕТ разделение доступа (^ уровни 0..3) и виртуальную адресацию памяти.
Защищенный - умеет в наоборот: 32битные регистры, многозадачность, до 4 ГБ виртуальной памяти. ПОДДЕРЖИВАЕТ уровни доступа (кольца защиты) и виртуальную адресацию. В защищенном режиме добавлены несколько дополнительных регистров. Сегментные: FS, GS; управляющие: CR0, CR1, CR2, CR3. Все они, соответственно, 32-разрядные.

Управляющие регистры CR2 и CR3 используются для страничного преобразования, благодаря которому работает техномагия виртуальных адресов. В CR0 лежит несколько флагов, которые управляют поведением процессора - например, последний (31 (не забываем что нумерация у нас с нуля)) бит Protected отвечает за режим, в котором находится процессор - защищённый если ==1 и реальный если ==0. CR1 зарезервирован для будущих поколений процессоров.

Сегменты памяти описываются дескрипторами по 8 байт.

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.

Кроме того, используется также таблица дескрипторов прерываний, содержащая в себе следующие структуры: int_descr struc offs_l dw 0 ;смещение в сегменте, нижняя часть sel dw 0 ;селектор сегмента с кодом прерывания counter db 0 ;счётчик, не используется в программе attr db 0 ;атрибуты offs_h dw 0 ;смещение в сегменте, верхняя часть int_descr ends Обработчики 0..16 зарезервированы под прерывания и исключения системы; 17..31 - под "будущие поколения процессоров"; остальные могут быть использованы пользователем. Таким образом, обработчики для аппаратных прерываний должны начинаться минимум с 32-го (в винде принято начинать обработку прерываний со смещения 50h, но мы на это забьём, чтобы не пилить ещё дофига лишних обработчиков).

- Нужны ли OFFFFh? Работает ли без этого?

Они не нужны так как не нужно т к не выходим за 1МБ ffff

- С помощью какой функции выводили строку в реальном режиме

Для вывода сообщения используется 9 функция 21 прерывания. (INT 21H: сервис DOS)

- Что делали с базовым вектором при переходе в защищенный режим и обратно?

В реальном режиме наш базовый вектор 8, в защищенном 32, при переходе в другой режим мы перепрограммируем контроллер прерываний.

- Каким образом мы перепрограммируем контроллер прерываний? Почему надо перепрограммировать контроллер прерываний?

Мы смещаем базовый вектор контроллера прерываний на 32. Потому что теперь прерывания имеют базовый вектор не 8, а 32.

перепрограммирование ведущего контроллера, т.к. в ЗР первые 32 вектора зарезервированы для обработки исключений, аппаратным прерываниям нужно назначить другие векторы 32=20h
Для смены базового вектора требуется полностью выполнить процедуру инициализации контроллера, которая состоит из ряда команд инициализации СКИ

```
mov al, 11h ;СКИ1: два контроллера в компьютере, будет СКИЗ
out 20h, al
mov al, 32 ;СКИ2: базовый вектор (был 8)
out 21h, al
mov al, 4 ;СКИ3: ведомый подключен к уровню 2
out 21h, al
mov al, 1 ;СКИ4:8086, требуется EOI программно
out 21h, al
```

В чем особенность 32разрядного сегмента данных?

4 гига

для чего? Для того, чтобы посчитать память

"Основной ответ - размер 4 гб Показываешь ей 5 эфок И говоришь про бит гранулярности и дигит"

"Ну есть особенность, что там, например, offset чего-то будет не 2, а 4 байта. И с циклами будет работать не CX, а ECX".

"У него установлен максимальный лимит

- В каком режиме мы в сегментные регистры записываем селекторы и для чего?

В защищенном.

- Что происходит при загрузке селектора в сегментный регистр?

теневой регистр, связанный с сегментным регистром, обновляются значением линейного адреса сегмента

- Что мы делаем с масками прерываний?

Маски - перед переходом в защищенный режим у контроллера ведущего и ведомого есть свои маски. Мы их сохраняем, чтобы потом восстановить. Мы их сохраняем, потому что мы ставим свои - теряем информацию о том что было. Потом их надо вернуть.

На черный день советы от старости четвертой группы

Главный вопрос: Опишите переход в реальный режим

1. Устанавливаем флаг перех. в pp
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

A20 и теневые регистры, доп вопросы

Семинар 3

Про уровни привилегий

Программа в защищённом режиме имеет 0 уровень привилегий.

В каком режиме вызываются привилегированные lgdt, lidt? – в реальном. Но это не значит, что там есть 0 уровень привилегий.

В прошлом семестре, чтобы система вызывала наш обработчик, мы в таблице векторов прерываний устанавливали новый вектор, старый запоминали, чтобы потом восстановить. То есть нет никакой защиты – залезли и «нагадили».

А привилегированные команды могут вызываться только на определенном уровне привилегий.

То, что мы в реальном режиме вызываем привилегированные команды, еще раз подтверждает, что в реальном режиме нет никакой защиты и уровней привилегий.

Перепрограммирование контролера прерываний

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

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 посылаются 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
```

Линия a20

Рассмотрим 2 спецификации (в России – гости). Технические гости вообще важная составлявшая техники, регламентирует много важной ху.

XMS	EMS
extended memory specification	expanded memory specification
Дополнительная память	Растянутая (расширенная) память
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. Частично она продемонстрирована в методичке к 1 лабе. (но на семе она рисует что-то вообще другое)

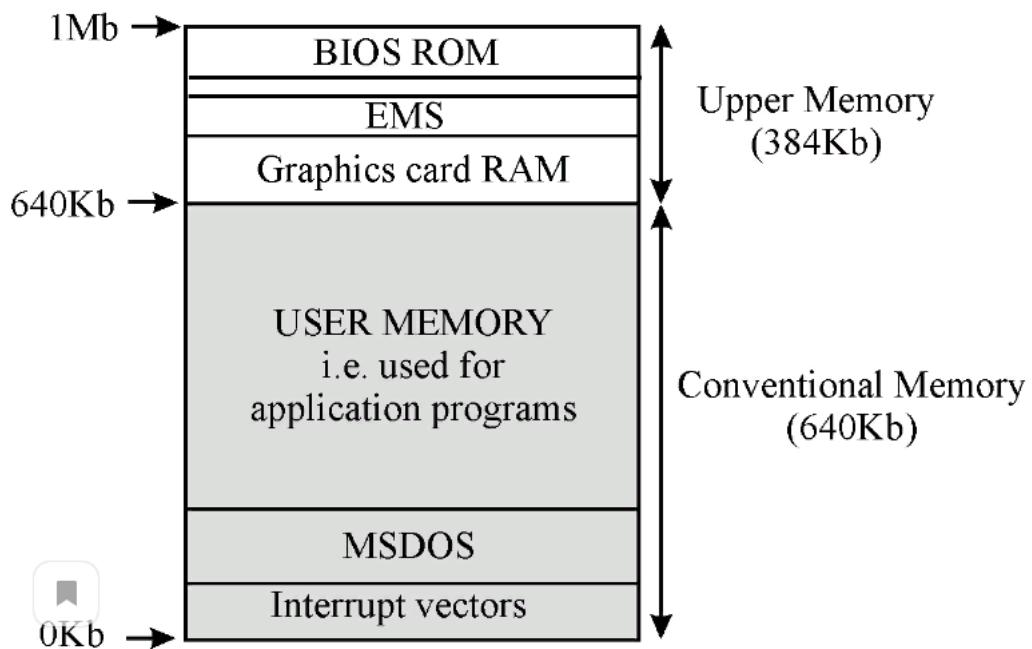
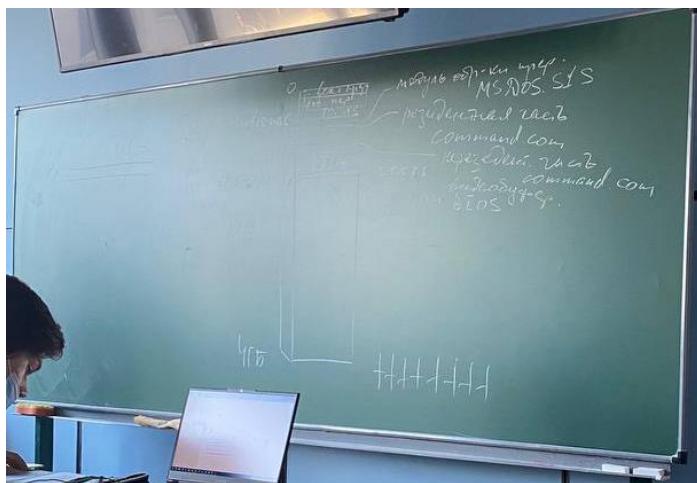
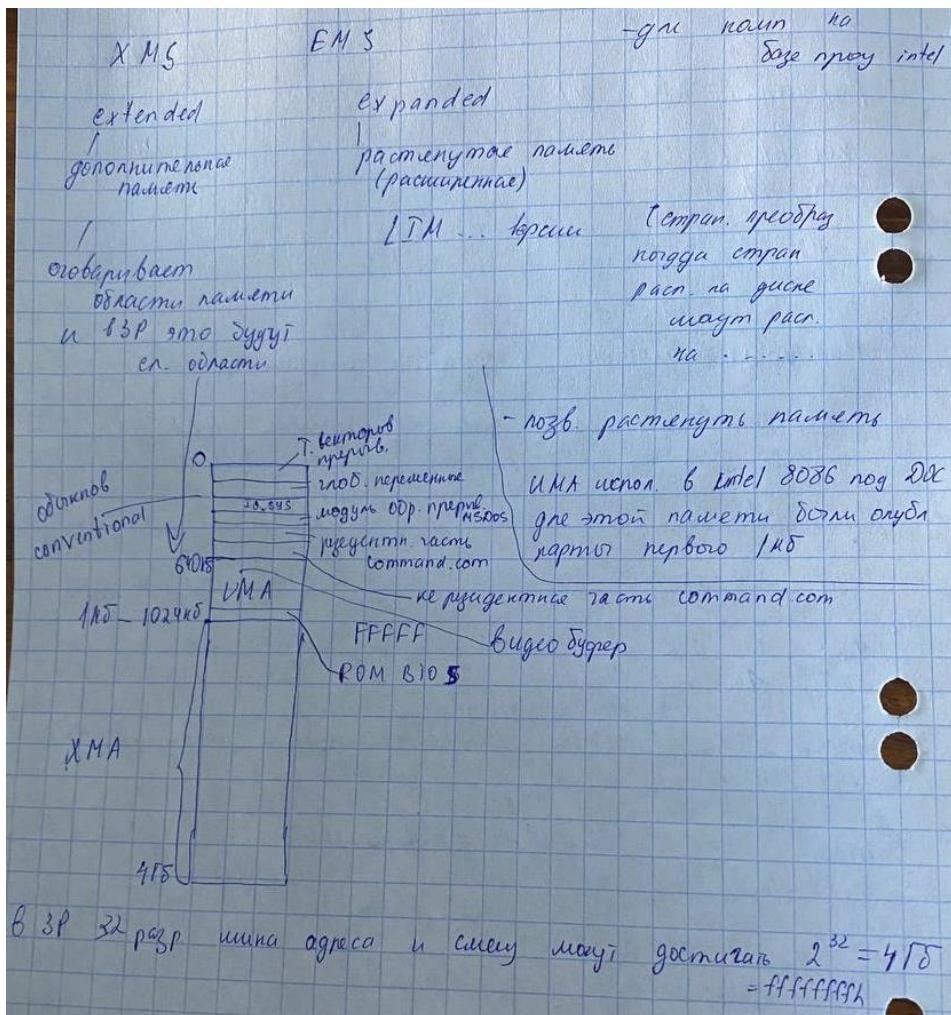


Рис. 15 базовая карта памяти системы XT под управлением MS-DOS.



MSDOS SYS – модуль обработки прерываний

Резидентная часть command.com

Когда появились, были тоже созданы карты первого мбайта. Это FFFFFF

В защищенном режиме 32 бита адреса – смещения могут достигать 4 ГБ – это 8 ф **FFFFFFF**

Память сверх одного мбайта –ХМА

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 сlijешь», 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 a20. У нас 32 адресные линии, поэтому это уже не 20 линия. В реальном режиме линия a20 закрыта. Она на 0 потенциале.

Когда переводим, ее надо открыть (с принудительного обнуления). Что будет, если забудем - то получаем битую память. Адреса с 1 в этом бите нам не будут доступны.

НЮАНС

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

Если мы в реальном режиме откроем линию a20, нам станет доступно еще 64 кбайт памяти- high memory area. Но это в реальном режиме. В защищенном режиме нам доступно 4 ГБ.

При переходе обратно, надо закрыть. Если не закроем – она нам будет доступна.

«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??

То есть если не запишем в limit ffff – криминала никакого.

Задача теневых регистров – хранить инфу о сегментах непосредственно в процессоре, чтобы исключить вечное обращение к памяти – затратное действие. Обращение к таблице глобальных дескрипторов. Из дескриптора – начальный адрес, из команды – смещение. Чтобы исключить обращение к таблице и ввели теневые регистры.

Существует 2 кода, которые используют при работе с a20:

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

Семинар 4

Linux

Linux это Unix-подобная ОС, реализует все его парадигмы. Как и в любой ОС, основной абстракцией является процесс.

В Unix новый взгляд на процесс: процесс часть времени выполняется в режиме «задача» (режим пользователя), и тогда он выполняет собственный код, а часть времени – в режиме ядра, и тогда он выполняет реентерабельный код ОС.

В режиме ядра процесс выполняет повторно входимые процедуры (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.

Unix имеет многопоточное ядро. Чистый Си, структуры. Потоки могут разделять структуры (экземпляры структур), они, естественно глобальные. Системные таблицы. Доступно в коде ядра linux. Но его ядро и его функции плохо описаны.

В Unix любой процесс создается системным вызовом fork (это база, есть другая инфа, но это база). 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 – не математическое, так просто говорят. Это отношение поддерживается соответствующими указателями. Процесс, вызвавший fork (...видимо сохраняет указатель на потомка). Потомок получает указатель на своего предка.

В unix и linux процесс описывается структурой: unix - struct proc, linux – struct task_struct. task_struct- найти в интернете и распечатать (большая).

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.

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: unix написана, когда - ? и для какой машины - PDP11) 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. 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().

Exit() – системный вызов завершения процесса.

В результате fork создается иерархия процессов в отношении предок-потомок. Эта иерархия поддерживается указателями (в unix есть связный список типа дерево). Но в unix есть двусвязные списки блокированных (или выполняемых, не успели) процессов, каких-то зомби.

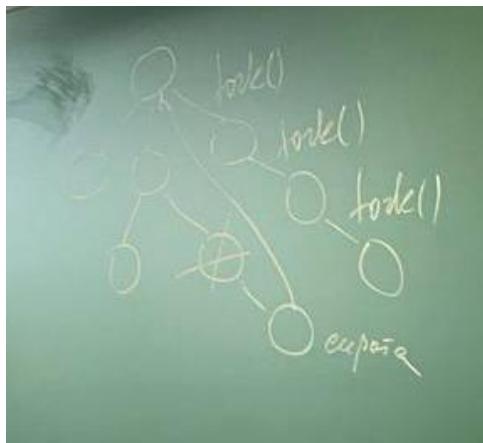
OFFTOP В системе таблицы не могут использоваться, ведь это массив структур, а это дополнительные накладные расходы при добавлении или удалении элементов: необходимо сдвигать. А тут-динамика, постоянно надо. А двусвязные – для ускорения поиска. А вообще сортировка – чтобы использовать быстрые поиски (бинарный, например). Если массив не отсортирован, придется полный перебор вообще. Поэтому двусвязный список – топ.

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 и процесс с идентификатором 1. Все идентификаторы в unix – целые положительные числа. Процесс с идентификатором 0 – процесс, запустивший систему. Процесс с идентификатором 1 – процесс, открывший терминал. Терминал – базовое понятие unix. Терминал – это монитор, который подключается....

Unix - система разделения времени sharing time.

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. 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. Если да, то начинается процесс усыновления – его усыновит процесс с идентификатором 1. При этом потомок получит указатель на нового предка, а предок – на нового потомка. То есть реально много действий.

Fork – вилка.

```
#include <stdio.h>
#include <unistd.h>/> она сказала это для маков
Int main(void)
{
    Int childpid; *
    If ((childpid=fork()) == -1) **
    {
        Perror("Can't fork");
        Exit(1)
    }
    else if (childpid==0) ***
    {
        Printf("childpid: pid=%d, ppid=%d", getpid(), getppid());
        Return 0;
    }
    Else
    {
        Printf("parent: childpid =%d, pid=%d\n", childpid, getpid());****
        Return 0;
    }
}
```

* здесь pid_t вместо int сейчас. Свой тип для всего. Зачем – это позволяет выполнять дополнительный контроль типов

** Часто fork проверяется на <0, но по соглашению в случае ошибки возвращается именно -1, так проверять круче по Р

*** else не нужен, но для полноты

**** вот тут видно, что Parent в результате fork получает идентификатор потомка Return – возврат управления, а exit – другое. Return 0 возвращает код системе.

Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родитель получает идентификатор (PID) потомка. Если это значение будет отрицательным, следовательно при порождении процесса произошла ошибка. Процесс-потомок получает в качестве кода возврата значение 0, если вызов fork() оказался успешным.

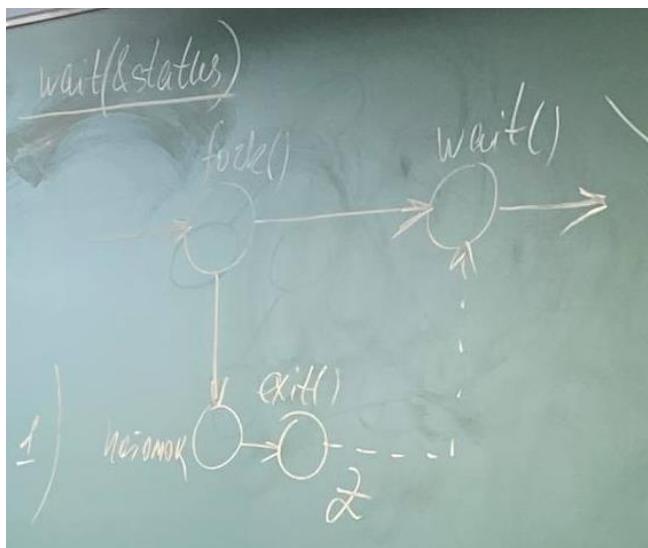
Таким образом, можно проверить, был ли создан новый процесс:

```
switch(ret=fork())  
{  
    case -1: /* при вызове fork() возникла ошибка */  
    case 0 : /* это код потомка */  
    default : /* это код родительского процесса */  
}
```

getppid (): возвращает идентификатор процесса родителя вызывающего процесса. Если вызывающий процесс был создан функцией fork (), и родительский процесс все еще существует во время вызова функции getppid, эта функция возвращает идентификатор процесса родительского процесса. В противном случае эта функция возвращает значение 1, которое является идентификатором процесса для процесса инициализации

getpid (): возвращает идентификатор процесса вызывающего процесса. Это часто используется подпрограммами, которые генерируют уникальные временные имена файлов.

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,

Системный вызов exec.

Есть в методе. Вахалия. Эти пункты надо выучить как стих.

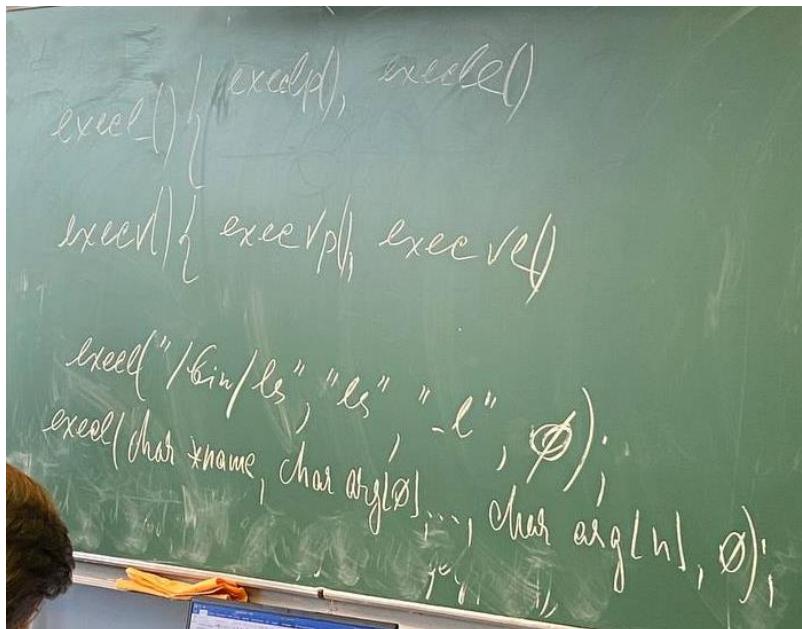
Exec() создает так называемый низкоуровневый процесс. Почему так называемый? – потому что на самом деле процесс не создается. Процесс создается через fork – с идентификатором и дескриптором. А exec для программы, которая передается exec в качестве параметра, создает адресное пространство, то есть - таблицу 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. Exec 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 в RIP (счетчик команд). То есть системный вызов exec переводит процесс на новое адресное пространство. А что это последнее предложение значит? Это надо разобрать, спросят

Система не контролирует, где exec(). По логике – именно потомок переходит на выполнение новой программы. Причем по логике fork происходит следующее: если exec() не вызван, то потомок и предок начинают параллельно выполнять один и тот же код. Fork bomb?

Иногда в этом есть смысл (массовая рассылка), но чаще нет. Поэтому потомок переходит на выполнение новой программы, в новом адресном пространстве. То есть любая программа запускается в 2 этапа – создать процесс и перейти на новое адресное пространство.



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

[С лабораторной работы 3](#)

D – директория

l-softlink

p-pike – именованный программный канал

s-socket – сокеты семейства афтишч

c-спец файл символьного устройстваи

b- спец файл блочного устройства

Следующие 9 букв – права доступа

Всего 3 – read/write/execute

По 3 на каждую группу

1) права доступа user

2) права доступа группы юзера

3) права доступа азерс – остального мира

Дальше столбик чисел – hard links- жесткие ссылки – еще одно равноправное имя файлв. То есть в unix имя файла не является его идентификатором, а идентифявляютс яструктура inode – описывает файл, у inode есть номер.

Время создания и имя. Первая строка заканчивается .-текущая лиректория, ..-родитель.

К 4

В лабе – строку printf с предком сначала вызвать до sleep, затем еще раз после sleep

Во 2 – sleep убираем, оставляем один printf + добавляется wait. Анализ кода завершения с помошь. Макросов

В 3 в потомках переходим на выполнение других программ. Потомки должны выполнять совершенно разные коды. Не who или echo.

НЕДОСТАТОК – мизерное исполнение потомков

В 4 – взаимодействие через неименованный программный канал. В предке создается один канал. Оба потомка записывают свои разные!!!! Сообщения. Не потомок 1, потомок 2. ПРИМЕР – буквы ABCD, другой – ZYX

В 5 – 4 программа со своим обработчиком сигнала. Объявить флаг – переменную и в обработчике присваивать ему значение 1. И ВЕТВИТЬСЯ ПО ФЛАГУ, Когда сигнал поступает, потомки отпраляют сообщение в канал, иначе – не отправляют

Обратить внимание на готов-выгружен, готов-загружен. Речь о памяти – в очередь готовых процессов. Если недостаточно памяти, то процесс готов, но выгружен.

Struct proc

Первые строки, указатели на очереди, дерево процессов в виде предок-потомок. Youngest living child – самый молодой живущий потомок.

ПЕРЕПИСАННЫАЯ от руки последовательность действий системы при выполнении fork и exec!!!!

Семинар 5

Linux

Linux это Unix-подобная ОС, реализует все его парадигмы. Как и в любой ОС, основной абстракцией является процесс.

В Unix новый взгляд на процесс: процесс часть времени выполняется в режиме «задача» (режим пользователя), и тогда он выполняет собственный код, а часть времени – в режиме ядра, и тогда он выполняет реентерабельный код ОС.

В режиме ядра процесс выполняет повторно входимые процедуры (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.

Unix имеет многопоточное ядро. Чистый Си, структуры. Потоки могут разделять структуры (экземпляры структур), они, естественно глобальные. Системные таблицы. Доступно в коде ядра linux. Но его ядро и его функции плохо описаны.

В Unix любой процесс создается системным вызовом fork (это база, есть другая инфа, но это база). 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 – не математическое, так просто говорят. Это

отношение поддерживается соответствующими указателями. Процесс, вызвавший fork (...видимо сохраняет указатель на потомка). Потомок получает указатель на своего предка.

В unix и linux процесс описывается структурой:

- unix - struct proc,
- linux – struct task_struct.

task_struct- найти в интернете и распечатать (большая). (см конец файла)

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.

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: unix написана, когда - ? и для какой машины - PDP11) 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)

3. 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/exes
4. 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 и данных 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().

Exit() – системный вызов завершения процесса.

В результате fork создается иерархия процессов в отношении предок-потомок. Эта иерархия поддерживается указателями (в unix есть связный список типа дерево). Но в unix есть двусвязные списки блокированных (или выполняемых, не успели) процессов, каких-то зомби.

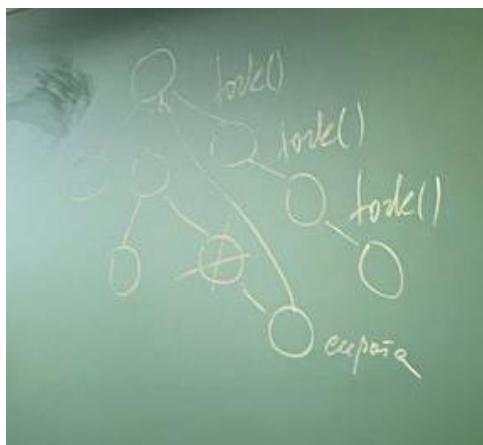
OFFTOP В системе таблицы не могут использоваться, ведь это массив структур, а это дополнительные накладные расходы при добавлении или удалении элементов: необходимо сдвигать. А тут-динамика, постоянно надо. А двусвязные – для ускорения поиска. А вообще сортировка – чтобы использовать быстрые поиски (бинарный, например). Если массив не отсортирован, придется полный перебор вообще. Поэтому двусвязный список – топ.

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 и процесс с идентификатором 1. Все идентификаторы в unix – целые положительные числа. Процесс с идентификатором 0 – процесс, запустивший систему. Процесс с идентификатором 1 – процесс, открывший терминал. Терминал – базовое понятие unix. Терминал – это монитор, который подключается....

Unix - система разделения времени sharing time.

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. 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. Если да, то начинается процесс усыновления – его усыновит процесс с идентификатором 1. При этом потомок получит указатель на нового предка, а предок – на нового потомка. То есть реально много действий.

Fork – вилка.

```
#include <stdio.h>
#include <unistd.h> // она сказала это для маков
Int main(void)
{
    Int childpid; *
    If ((childpid=fork()) == -1) **
    {
        Perror("Can't fork");
        Exit(1)
    }
    else if (childpid==0) ***
    {
        Printf("childpid: pid=%d, ppid=%d", getpid(), getppid());
        Return 0;
    }
    Else
    {
        Printf("parent: childpid =%d, pid=%d\n", childpid, getpid());****
        Return 0;
    }
}
```

* здесь pid_t вместо int сейчас. Свой тип для всего. Зачем – это позволяет выполнять дополнительный контроль типов

** Часто fork проверяется на <0, но по соглашению в случае ошибки возвращается именно -1, так проверять круче по Р

*** else не нужен, но для полноты

**** вот тут видно, что Parent в результате fork получает идентификатор потомка

Return – возврат управления, а exit – другое. Return 0 возвращает код системе.

Процесс-потомок и процесс-родитель получают разные коды возврата после вызова fork(). Процесс-родитель получает идентификатор (PID) потомка. Если это значение будет отрицательным, следовательно при порождении процесса произошла ошибка. Процесс-потомок получает в качестве кода возврата значение 0, если вызов fork() оказался успешным.

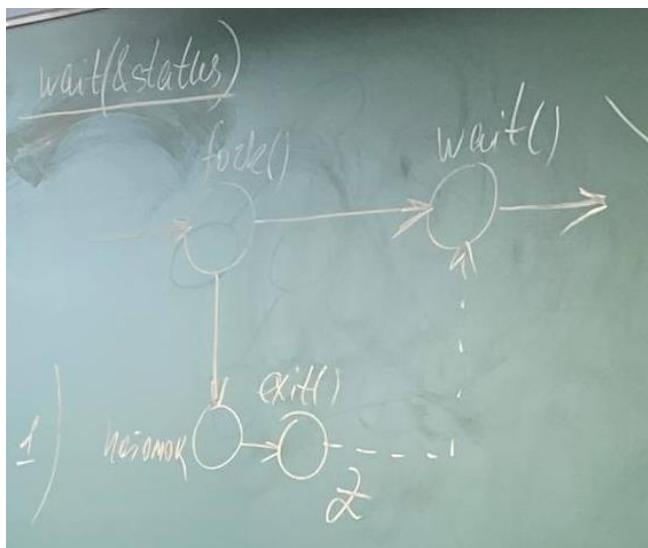
Таким образом, можно проверить, был ли создан новый процесс:

```
switch(ret=fork())  
{  
    case -1: /* при вызове fork() возникла ошибка */  
    case 0 : /* это код потомка */  
    default : /* это код родительского процесса */  
}
```

getppid (): возвращает идентификатор процесса родителя вызывающего процесса. Если вызывающий процесс был создан функцией fork (), и родительский процесс все еще существует во время вызова функции getppid, эта функция возвращает идентификатор процесса родительского процесса. В противном случае эта функция возвращает значение 1, которое является идентификатором процесса для процесса инициализации

getpid (): возвращает идентификатор процесса вызывающего процесса. Это часто используется подпрограммами, которые генерируют уникальные временные имена файлов.

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,

Системный вызов exec.

Есть в методе. Вахалия. Эти пункты надо выучить как стих.

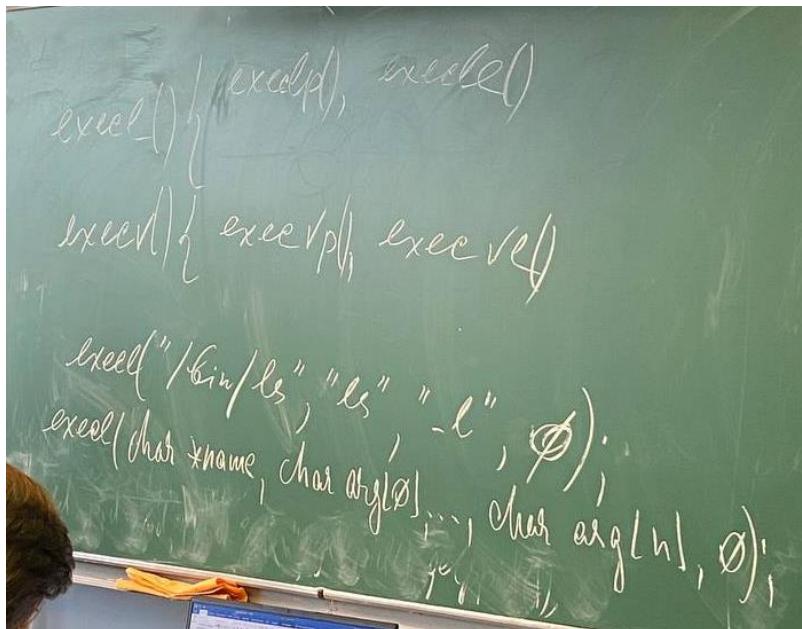
Exec() создает так называемый низкоуровневый процесс. Почему так называемый? – потому что на самом деле процесс не создается. Процесс создается через fork – с идентификатором и дескриптором. А exec для программы, которая передается exec в качестве параметра, создает адресное пространство, то есть - таблицу 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. Exec 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 в RIP (счетчик команд). То есть системный вызов exec переводит процесс на новое адресное пространство. А что это последнее предложение значит? Это надо разобрать, спросят

Система не контролирует, где exec(). По логике – именно потомок переходит на выполнение новой программы. Причем по логике fork происходит следующее: если exec() не вызван, то потомок и предок начинают параллельно выполнять один и тот же код. Fork bomb?

Иногда в этом есть смысл (массовая рассылка), но чаще нет. Поэтому потомок переходит на выполнение новой программы, в новом адресном пространстве. То есть любая программа запускается в 2 этапа – создать процесс и перейти на новое адресное пространство.



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

Обратить внимание на диаграмме состояний процессов на готов-выгружен, готов-загружен. Речь о памяти – в очередь готовых процессов. Если недостаточно памяти, то процесс готов, но выгружен.

Struct proc

Первые строки, указатели на очереди, дерево процессов в виде предок-потомок. Youngest living child – самый молодой живущий потомок.

ПЕРЕПИСАННЫЯ от руки последовательность действий системы при выполнении fork и exec!!!!

С лабораторной (от Иры)

FLAGS:

- 1 – был fork, но не было exec (важно с точки зрения флага COW)
- 4 – used super_user privileges
- 0 – был fork и exec

Super_user – может обращаться к функциям и структурам ядра

Если exec:

- Не было, то parent, child разделяют одно АП
- Был, то нормальные права доступа

State

- (обратить внимание на R)
- (система не разделяет, выполняется ли процедура или стоит в очереди готовых процедур
- D, I, R, S, T, t, W, X, Z

Hardlink: ln <file> hardlink

Softlink: ln -s <file> softlink

Изменится приоритет: renice <приор> -p <номер процесса>

Создать pipe:

```
mknnod pipe -p  
echo "abc" > pipe  
tee < pipe
```

Softlink-специальный файл, который создает путь к файлу (обозначается L)-символьная строка (путь к файлу)

PRI – системный приоритет процесса (чем он больше, тем ниже приоритет)

Не помню, к какой команде, вроде ps –ajl

Kthread – процесс запускает потомки

Kworkr /0 /1 (всего 4 процессора, то будет /0 /1 /2 /3)

Ksoftirqd – демон планир выполнения прерываний (по инету – мягкие прерывания выполняются этим демоном)

ls –al

- обычный файл, который создается приложением (чтобы хранить долгое время)

d – тип справочных или директорий

l – softlink

p-pipe

s-сокет

c-специальный файл символьного устройства

b-специальный файл блочного устройства

столбик чисел -> то hardlink-равноправное имя файла. То есть имя файла не является его идентификатором. Идентификатор-метаданные-номера inode (структуры, описывающей файл)

ls –ali – номера inode выводит

Приложение

- Про структуры

Процессы, например, в Unix BSD описываются структурой proc:

Основные поля структуры `proc` охватывают:

- ◆ идентификацию: каждый процесс обладает уникальным *идентификатором процесса* (process ID, или *PID*) и относится к определенной *группе процессов*. В современных версиях системы каждому процессу также присваивается *идентификатор сеанса* (session ID);
- ◆ расположение карты адресов ядра для области и данного процесса;
- ◆ текущее состояние процесса;
- ◆ предыдущий и следующий указатели, связывающие процесс с очередью планировщика (или очередью приостановленных процессов, если данный процесс был заблокирован);
- ◆ канал «сна» для заблокированных процессов (см. раздел 7.2.3);
- ◆ приоритеты планирования задач и связанную информацию (см. главу 5);
- ◆ информацию об обработке сигналов: маски игнорируемых, блокируемых, передаваемых и обрабатываемых сигналов (см. главу 4);
- ◆ информацию по управлению памятью;
- ◆ указатели, связывающие эту структуру со списками активных, свободных или завершенных процессов (зомби);
- ◆ различные флаги;
- ◆ указатели на расположение структуры в *очереди хэша*, основанной на *PID*;
- ◆ информация об иерархии, описывающая взаимосвязь данного процесса с другими.

Эта структура содержит всю информацию о процессе. Информацию можно посмотреть по адресу `/usr/src/sys/sys/proc.h`:

```
/*
 * Description of a process.
 *
 * This structure contains the information needed to manage a thread of
 * control, known in UN*X as a process; it has references to substructures
 * containing descriptions of things that the process uses, but may share
 * with related processes. The process structure and the substructures
 * are always addressable except for those marked "(PROC ONLY)" below,
 * which might be addressable only on a processor on which the process
 * is running.
 */
struct proc {
    struct proc *p_forw;      /* Doubly-linked run/sleep queue. */
    struct proc *p_back;
    struct proc *p_next;      /* Linked list of active procs */
    struct proc **p_prev;     /* and zombies. */

    /* substructures: */
    struct pcred *p_cred;    /* Process owner's identity. */
    struct filedesc *p_fd;   /* Ptr to open files structure. */
    struct pstats *p_stats;  /* Accounting/statistics (PROC ONLY). */
    struct plimit *p_limit;  /* Process limits. */
    struct vmspace *p_vmspace; /* Address space. */
    struct sigacts *p_sigacts; /* Signal actions, state (PROC ONLY). */

#define p_ucred    p_cred->pc_ucred
#define p_rlimit   p_limit->pl_rlimit

    int    p_flag;           /* P_* flags. */
}
```

```

char p_stat;           /* S* process status. */
char p_pad1[3];

pid_t p_pid;           /* Process identifier. */
struct proc *p_hash;  /* Hashed based on p_pid for kill+exit+... */
struct proc *p_pgrpnxt; /* Pointer to next process in process group. */
struct proc *p_pptr;   /* Pointer to process structure of parent. */
struct proc *p_osptr;  /* Pointer to older sibling processes. */

/* The following fields are all zeroed upon creation in fork. */
#define p_startzero p_ysptr
    struct proc *p_ysptr; /* Pointer to younger siblings. */
    struct proc *p_cptr;  /* Pointer to youngest living child. */
    pid_t p_oppid;       /* Save parent pid during ptrace. XXX */
    int p_dupfd;         /* Sideways return value from fdopen. XXX */

/* scheduling */
u_int p_estcpu;        /* Time averaged value of p_cpticks. */
int p_cpticks;          /* Ticks of cpu time. */
fixpt_t p_pctcpu;      /* %cpu for this process during p_swtime */
void *p_wchan;          /* Sleep address. */
char *p_wmesg;          /* Reason for sleep. */
u_int p_swtime;         /* Time swapped in or out. */
u_int p_slptime;        /* Time since last blocked. */

struct itimerval p_realtimer; /* Alarm timer. */
struct timeval p_rtime;   /* Real time. */
u_quad_t p_uticks;      /* Statclock hits in user mode. */
u_quad_t p_sticks;      /* Statclock hits in system mode. */
u_quad_t p_iticks;      /* Statclock hits processing intr. */

int p_traceflag;         /* Kernel trace points. */
struct vnode *p_tracep;  /* Trace to vnode. */

int p_siglist;           /* Signals arrived but not delivered. */

struct vnode *p_textvp;  /* Vnode of executable. */

char p_lock;             /* Process lock (prevent swap) count. */
char p_pad2[3];          /* alignment */

/* End area that is zeroed on creation. */
#define p_endzero p_startcopy

/* The following fields are all copied upon creation in fork. */
#define p_startcopy p_sigmask

sigset_t p_sigmask;      /* Current signal mask. */
sigset_t p_sigignore;    /* Signals being ignored. */
sigset_t p_sigcatch;    /* Signals being caught by user. */

u_char p_priority;        /* Process priority. */
u_char p_usrpri;         /* User-priority based on p_cpu and p_nice. */
char p_nice;              /* Process "nice" value. */
char p_comm[MAXCOMLEN+1];

struct pgrp *p_pgrp;      /* Pointer to process group. */

struct sysentvec *p_sysent; /* System call dispatch information. */

struct rt prio p_rt prio; /* Realtime priority. */

/* End area that is copied on creation. */
#define p_endcopy p_addr

```

```

struct user *p_addr; /* Kernel virtual addr of u-area (PROC ONLY). */
struct mdproc p_md; /* Any machine-dependent fields. */

u_short p_xstat; /* Exit status for wait; also stop signal. */
u_short p_acflag; /* Accounting flags. */
struct rusage *p_ru; /* Exit information. XXX */
};

struct task_struct {
    volatile long state; //The running state of the task (- 1 is not
running, 0 is running (ready), > 0 has stopped).
    void *stack; //Process Kernel Stack
    atomic_t usage; //Several processes are using this structure
    unsigned int flags; //per process flags, defined below//
information about the state of the reaction process, but not the running
state
    unsigned int ptrace; //system call

    int lock_depth; /* BKL lock depth */

#endif CONFIG_SMP
#ifndef __ARCH_WANT_UNLOCKED_CTXSW
    int oncpu; //Which CPU to run on
#endif
#endif

    int prio, static_prio, normal_prio;//Static priority, dynamic priority
    unsigned int rt_priority; //Priority of Real-time Tasks
    const struct sched_class *sched_class; //Scheduling-related functions
    struct sched_entity se; //Scheduling entity
    struct sched_rt_entity rt; //Real-time Task Scheduling Entities

#endif CONFIG_PREEMPT_NOTIFIERS
//list of struct preempt_notifier:
    struct hlist_head preempt_notifiers; //Relevant to seizure
#endif

/*
 * fpu_counter contains the number of consecutive context switches
 * that the FPU is used. If this is over a threshold, the lazy fpu
 * saving becomes unlazy to save the trap. This is an unsigned char
 * so that after 256 times the counter wraps and the behavior turns
 * lazy again; this to deal with bursty apps that only use FPU for
 * a short time
 */
    unsigned char fpu_counter;
#endif CONFIG_BLK_DEV_IO_TRACE
    unsigned int btrace_seq;
#endif

    unsigned int policy; //scheduling strategy
    cpumask_t cpus_allowed; //Bitmaps for managing CPU in multi-core
architecture

#endif CONFIG_TREE_PREEMPT_RCU //A New Lock Mechanism
    int rcu_read_lock_nesting;
    char rcu_read_unlock_special;
    struct rcu_node *rcu_blocked_node;

```

```

        struct list_head rcu_node_entry;
#endif /* #ifdef CONFIG_TREE_PREEMPT_RCU */

#if defined(CONFIG_SCHEDSTATS) || defined(CONFIG_TASK_DELAY_ACCT)
    struct sched_info sched_info; // Scheduling related information,
    such as running time on CPU / waiting time in queue, etc.
#endif

    struct list_head tasks;           //Task queue
    struct plist_node pushable_tasks;

    struct mm_struct *mm, *active_mm; //mm is the memory management
information of the process

/* task state */
    int exit_state;                //Status of a process at exit
    int exit_code, exit_signal; //Signals emitted when a process exits
    int pdeath_signal; /* The signal sent when the parent dies */
    /* ??? */
    unsigned int personality; //Because Unix has many different versions
and variations, applications also have scope of application.
    unsigned did_exec:1;          //According to POSIX programming
standards, did_exec is used to indicate whether the current process is
executing the original code or executing the new code scheduled by execve.
    unsigned in_execve:1; /* Tell the LSMs that the process is doing an
                           * execve */
    unsigned in_iowait:1;

/* Revert to default priority/policy when forking */
    unsigned sched_reset_on_fork:1;

    pid_t pid; //Process ID
    pid_t tgid; //Thread group ID

#endif /* CONFIG_CC_STACKPROTECTOR */
/* Canary value for the -fstack-protector gcc feature */
    unsigned long stack_canary;
#endif

/*
 * pointers to (original) parent process, youngest child, younger
sibling,
 * older sibling, respectively. (p->father can be replaced with
 * p->real_parent->pid)
 */
    struct task_struct *real_parent; /* real parent process */
    struct task_struct *parent; /* recipient of SIGCHLD, wait4() reports
*/
/*
 * children/sibling forms the list of my natural children
 */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */

/*
 * ptraced is the list of tasks this task is using ptrace on.
 * This includes both natural children and PTRACE_ATTACH targets.
 * p->ptrace_entry is p's link on the p->parent->ptraced list.
 */

```

```

    struct list_head ptraced;
    struct list_head ptrace_entry;

    /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;

    struct completion *vfork_done;           /* for vfork() */
    int __user *set_child_tid;             /* CLONE_CHILD_SETTID */
    int __user *clear_child_tid;          /* CLONE_CHILD_CLEARTID */

    cputime_t utime, stime, utimescaled, stimescaled; // Utme is the
    time consumed by process user state, and stime is the time spent by system
    state.

    cputime_t gtime;
    cputime_t prev_utime, prev_stime;
    unsigned long nvcsw, nivcsw; /* context switch counts */
    struct timespec start_time;           /* monotonic time */
    struct timespec real_start_time;     /* boot based time */
/* mm fault and swap info: this can arguably be seen as either mm-specific
or thread-specific */
    unsigned long min_flt, maj_flt;

    struct task_cputime cputime_expires; //How long does the process
    expire?
        struct list_head cpu_timers[3]; //??? /* process credentials */ /
        / / / / Refer to the annotations for the cred structure definition file

    const struct cred __rcu *real_cred; /* objective and real subjective task
    * credentials (COW) */
        const struct cred __rcu *cred; /* effective (overridable)
    subjective task * credentials (COW) */
            struct cred *replacement_session_keyring; /* for
    KEYCTL_SESSION_TO_PARENT */
                char comm[TASK_COMM_LEN]; /* executable name excluding path -
    access with [gs]et_task_comm (which lock it with task_lock()) -
    initialized normally by setup_new_exec */
                    /* file system info */
                    int link_count, total_link_count; //How many hard connections are
    there?
            #ifdef CONFIG_SYSVIPC/* ipc stuff //Interprocess communication
    related things
                struct sysv_sem sysvsem;
            #endif
            #ifdef CONFIG_DETECT_HUNG_TASK/* hung task detection */
                unsigned long last_switch_count;
            #endif/* CPU-specific state of this task */
    struct thread_struct thread; /*Because task_stcut is independent of
    hardware architecture, thread_struct is used to accommodate different
    architectures.*/
        /* filesystem information */
        struct fs_struct *fs;
        /* open file information */
        struct files_struct *files;
        /* namespaces //In-depth discussion on namespaces
        struct nsproxy *nsproxy; /* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;
        sigset_t blocked, real_blocked;
        sigset_t saved_sigmask; /* restored if set_restore_sigmask() was
    used */

```

```

    struct sigpending pending; //Indicates that the process has
received a signal but has not yet processed it.
    unsigned long sas_ss_sp;size_t sas_ss_size;
    /*Although signal handling takes place in the kernel, the
installed signal handlers run in usermode - otherwise,
it would be very easy to introduce malicious or faulty code into
the kernel and undermine the system security mechanisms.
    Generally, signal handlers use the user mode stack of the process
in question.
    However, POSIX mandates the option of running signal handlers on
a stack set up specifically for this purpose (using the
    sigaltstack system call). The address and size of this
additional stack (which must be explicitly allocated by the
    user application) are held in sas_ss_sp and sas_ss_size,
respectively. (Professional Linux® Kernel Architecture Page 384)*/
    int (*notifier)(void *priv);
    void *notifier_data;
    sigset_t *notifier_mask;
    struct audit_context *audit_context; //See Professional Linux <
Kernel Architecture Page 1100
    #ifdef CONFIG_AUDITSYSCALL
    uid_t loginuid;
    unsigned int sessionid;
    #endif
    seccomp_t seccomp;
    /* Thread group tracking */
    u32 parent_exec_id;
    u32 self_exec_id; /* Protection of (de-)allocation: mm, files, fs,
tty, keyrings, mems_allowed, * mempolicy */
    spinlock_t alloc_lock;
    #ifdef CONFIG_GENERIC_HARDIRQS /* IRQ handler threads */
    struct irqaction *irqaction; #endif /* Protection of the PI data
structures: */ //PI --> Priority Inheritance raw_spinlock_t pi_lock;
    #ifdef CONFIG_RT_MUTEXES //RT -> RealTime Task Real-time task
/* PI waiters blocked on a rt_mutex held by this task*/
    struct plist_head pi_waiters; /* Deadlock detection and priority
inheritance handling */
    struct rt_mutex_waiter *pi_blocked_on;
    #endif
    #ifdef CONFIG_DEBUG_MUTEXES /* mutex deadlock detection */
    struct mutex_waiter *blocked_on;
    #endif
    #ifdef CONFIG_TRACE_IRQFLAGS
    unsigned int irq_events;
    unsigned long hardirq_enable_ip;
    unsigned long hardirq_disable_ip;
    unsigned int hardirq_enable_event;
    unsigned int hardirq_disable_event;
    int hardirqs_enabled;
    int hardirq_context;
    unsigned long softirq_disable_ip;
    unsigned long softirq_enable_ip;
    unsigned int softirq_disable_event;
    unsigned int softirq_enable_event;
    int softirqs_enabled;
    int softirq_context;
    #endif
    #ifdef CONFIG_LOCKDEP
    # define MAX_LOCK_DEPTH 48UL
    u64 curr_chain_key;
    int lockdep_depth; //Lock depth

```

```

unsigned int lockdep_recursion;
struct held_lock held_locks[MAX_LOCK_DEPTH];
gfp_t lockdep_reclaim_gfp;
#endif
/* journalling filesystem info */
void *journal_info; //File system log information
/* stacked block device info */
struct bio_list *bio_list; //Block IO Device Table
#ifndef CONFIG_BLOCK
/* stack plugging */
struct blk_plug *plug;
#endif
/* VM state */
struct reclaim_state *reclaim_state;
struct backing_dev_info *backing_dev_info;
struct io_context *io_context;
unsigned long ptrace_message;
siginfo_t *last_siginfo;
/* For ptrace use. */
struct task_io_accounting ioac; //a structure which is used for
recording a single task's IO statistics.
#if defined(CONFIG_TASK_XACCT)
u64 acct_rss_mem1;
/* accumulated rss usage */
u64 acct_vm_mem1;
/* accumulated virtual memory usage */
cputime_t acct_timexpd;
/* stime + utime since last update */
#endif
#ifndef CONFIG_CPUSETS
nodemask_t mems_allowed;
/* Protected by alloc_lock */
int mems_allowed_change_disable;
int cpuset_mem_spread_rotor;
int cpuset_slab_spread_rotor;
#endif
#ifndef CONFIG_CGROUPS
/* Control Group info protected by css_set_lock */
struct css_set __rcu *cgroups;
/* cg_list protected by css_set_lock and tsk->alloc_lock */
struct list_head cg_list;
#endif
#ifndef CONFIG_FUTEX
struct robust_list_head __user *robust_list;
#endif
#ifndef CONFIG_COMPAT
struct compat_robust_list_head __user *compat_robust_list;
#endif
struct list_head pi_state_list;
struct futex_pi_state *pi_state_cache;
#endif
#ifndef CONFIG_PERF_EVENTS
struct perf_event_context *perf_event_ctxp[perf_nr_task_contexts];
struct mutex perf_event_mutex;
struct list_head perf_event_list;
#endif
#ifndef CONFIG_NUMA
struct mempolicy *mempolicy;
/* Protected by alloc_lock */
short il_next;
short pref_node_for;
#endif
atomic_t fs_excl; /* holding fs exclusive resources
*///Whether process exclusive filesystems are allowed. 0 denotes No.

```

```

        struct rcu_head rcu; /* * cache last used pipe for splice */
        struct pipe_inode_info *splice_pipe;
        #ifdef CONFIG_TASK_DELAY_ACCT
        struct task_delay_info *delays;
        #endif
        #ifdef CONFIG_FAULT_INJECTION
        int make_it_fail;
        #endif
        struct prop_local_single dirties;
        #ifdef CONFIG_LATENCYTOP
        int latency_record_count;
        struct latency_record latency_record[LT_SAVECOUNT];
        #endif
        /* * time slack values; these are used to round up poll() and *
        select() etc timeout values.
            These are in nanoseconds. */
        unsigned long timer_slack_ns;
        unsigned long default_timer_slack_ns;
        struct list_head *scm_work_list;
        #ifdef CONFIG_FUNCTION_GRAPH_TRACER
        /* Index of current stored address in ret_stack */
        int curr_ret_stack; /* Stack of return addresses for return
        function tracing */
        struct ftrace_ret_stack *ret_stack; /* time stamp for last schedule
        */
        unsigned long long ftrace_timestamp;
        /* * Number of functions that haven't been traced * because of
        depth overrun. */
        atomic_t trace_overrun;
        /* Pause for the tracing */
        atomic_t tracing_graph_pause;
        #endif
        #ifdef CONFIG_TRACING
        /* state flags for use by tracers */
        unsigned long trace; /* bitmask and counter of trace recursion */
        unsigned long trace_recursion;
        #endif /* CONFIG_TRACING */
        #ifdef CONFIG_CGROUP_MEM_RES_CTLR
        /* memcg uses this to do batch job */
        struct memcg_batch_info {int do_batch; /* incremented when batch
        uncharge started */
        struct mem_cgroup *memcg; /* target memcg of uncharge */
        unsigned long nr_pages; /* uncharged usage */
        unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
        } memcg_batch;
        #endif
        #ifdef CONFIG_HAVE_HW_BREAKPOINT
        atomic_t ptrace_bp_refcnt;
        #endif
    };
}

```

Рассмотрим более подробно, что же делается при выполнении вызова fork():

1. Резервируется пространство свопинга для данных и стека процесса-потомка;
2. Назначается идентификатор процесса PID и структура proc потомка;
3. Инициализируется структура proc потомка. Некоторые поля этой структуры копируются от процесса-родителя: идентификаторы пользователя и группы, маски

сигналов и группа процессов. Часть полей инициализируется 0. Часть полей инициализируется специфическими для потомка значениями: PID потомка и его родителя, указатель на структуру proc родителя;

4. Создаются карты трансляции адресов для процесса-потомка;
5. Выделяется область и потомка и в нее копируется область и процесса-предка;
6. Изменяются ссылки области и на новые карты адресации и пространство свопинга;
7. Потомок добавляется в набор процессов, которые разделяют область кода программы, выполняемой процессом-родителем;
8. Постранично дублируются области данных и стека родителя и модифицируются карты адресации потомка;
9. Потомок получает ссылки на разделяемые ресурсы, которые он наследует : открытые файлы (потомок наследует дескрипторы) и текущий рабочий каталог;
10. Инициализируется аппаратный контекст потомка путем копирования регистров родителя;
11. Поместить процесс-потомок в очередь готовых процессов;
12. Возвращается PID в точку возврата из системного вызова в родительском процессе и 0 - в процессе-потомке.

Системный вызов `exec` выполняет следующие действия:

1. Разбирает путь к исполняемому файлу и осуществляет доступ к нему.
2. Проверяет, имеет ли вызывающий процесс полномочия на выполнение файла.
3. Читает заголовок и проверяет, что он действительно исполняемый¹.
4. Если для файла установлены биты SUID или SGID, то эффективные идентификаторы UID и GID вызывающего процесса изменяет на UID и GID, соответствующие владельцу файла.
5. Копирует аргументы, передаваемые в `exec`, а также *переменные среды* в пространство ядра, после чего текущее пользовательское пространство готово к уничтожению.
6. Выделяет пространство свопинга для областей данных и стека.
7. Высвобождает старое адресное пространство и связанное с ним пространство свопинга. Если же процесс был создан при помощи `vfork`, производится возврат старого адресного пространства родительскому процессу.
8. Выделяет карты трансляции адресов для нового текста, данных и стека.
9. Устанавливает новое адресное пространство. Если область текста активна (какой-то другой процесс уже выполняет ту же программу), то она будет совместно использоваться с этим процессом. В других случаях пространство должно инициализироваться из выполняемого файла. Процессы в системе UNIX обычно разбиты на страницы, что означает, что каждая страница считывается в память только по мере необходимости.
10. Копирует аргументы и переменные среды обратно в новый стек приложения.
11. Сбрасывает все обработчики сигналов в действия, определенные по умолчанию, так как функции обработчиков сигналов не существуют в новой программе. Сигналы, которые были проигнорированы или заблокированы перед вызовом `exec`, остаются в тех же состояниях.
12. Инициализирует аппаратный контекст. При этом большинство регистров сбрасывается в 0, а указатель команд получает значение точки входа программы.

Семинар 6

Ввод количества элементов массива, блокируется, потом вторая также. Первая введенная 2 – для первой программы.

Процесс блокирован – значит, что он не получает процессорное время, а значит он не выполняется

Void *shmat(int shmid, const void *shmaddr, int shmflg);, где const void *shmaddr=0,SHM_RND

Семинар

Дейкер- только для 2, флаг не работает, test_and_set- широко используется в макрокомандах и даже семафорах

Средство взаимодействия процессов IPC (inner process communication, межпроц вzd, System V).

System и Unix BSD стали сильно различаться.

В 5 используются средства межпроц вzd

- 1) сигналы
- 2) семафоры
- 3) программные каналы (именованные и неименованные)
- 4) сегменты разделяемой памяти
- 5) очереди сообщений

В юникс – это сокеты (след семетр)

Linux-Unix подобная ОС, содержит все системные вызовы Ansi и Sosi и Posix. Также выжедаются файлы, отображаемый в память (mapping ..), но мы их не рассматриваем.

Семафоры в Unix

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 вzd – передают информацию

Все рассмотренные ранее средства связаны с активным ожиданием на процессе с помощь. Циклов, на проверку которых тратится процессорное время. Поэтому появление семафоров позволило устраниТЬ это ожидание, но потребовало использования системных вызовов (P(s), V(s)) -- система переходит в режим ядра, приходится переключать контекст, но больше нет 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, Unix поддерживает наборы семафоров.

В ядре системы имеется таблица семафоров. В ней отслеживаются все созданные в системе наборы семафоров. Для этого существует специальная структура struct semid_ds в библиотеке <sys/sem.h> (ds=descriptor, sem=semafor).

Таблица – таблица дескрипторов семафоров. Каждая строка описывает отдельный набор. О каждом наборе известно:

1) Имя – целое число. В Unix все ид-цел. Присваивается процессом, который создал набор семафоров. Другие процессы по этому имени могут набрать набор и по этому номеру получить дескриптор для доступа к набору.

Свойство наборов семафоров – одной неделимой операцией можно изменить все или часть набора семафоров.

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

На семафорах определены системные вызовы с формальными параметрами:

Int semget(key_t key, int num_sem, int fl), key-не совсем вписывается в классику, имеется в виду имя, num-sem-количество семафоров в наборе, fl-набор флагов(права доступа и тд, и то, что далее в примере) Возвращает файловый дескриптор. Далее по нему можно обращаться

Int semctl(int semfd, int num, int cmd, union semun arg), где semfd (filedescriptor), num-количество семафором

Int semop(int semfd, struct sembuf, *opstr, int len);

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

Struct sembuf

```
{  
Ushort sem_num;-индекс ссемафора в наборе  
Short sem_op – операция на семафоре  
Short semfl - Флаги, определенные на смеафоре  
}
```

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

Semop>0-инкремент, освобождение семафора=V(S)

Semop=0-(нет у дейкстры)-процесс, выполнивший этот св переводится в состояние ожидания до момента высвобождения ресурсов

Semop < 0 – декремент, захват семафора=P(S). Процесс блокируется на семафоре, если он захвачен, иначе...

Флаги

IPS_NOWAIT – информирует ядро о нежелании процесса переходить в состояние ожидания. Наличие этого влага 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 вот так следит

Пример

```
#include <sys/types.h>  
#include <sys/ipe.h>  
#include <sys/sem.h>  
  
Struct sembuf sbuf[2] = {{0, -1, SEM_UNDO|IPC_NOWAIT}, {1, 0, 1}}; //массив структур – одной неделимой semop например можно все изменить
```

Первый набор семафора захватывается, второй проверяется на 0

```
Int main()  
{  
Int perms = S_IRWXV|S_IRWXG|S_IRWX_0; //полные права доступа для всех категорий пользователей группы user  
  
Int fd = semget(100, 2, IPC_CREATE|perms); //создает набор из 2 семафоров с идентификатором 100, возвр файловый дескриптор
```

```

If (fd == -1)
{
    Набор создать не удалось
    perror("semop");
    exit(1);
}

//передаётся дескриптор набора, массив структур, количество семафоров, над которыми
выполняется операция

If (semop(fd, sbuf, 2) == -1)
{
    perror("semop");
}

Return 0;
}

```

Семафор был захвачен, не освобожден, процесс завершился.

Но система отменить сделанные изменения, потому что установлены флаги

Сегменты разделяемой памяти

Коммуникация связана с передачей сообщений. 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 ид 100 размером 1024 б с полными правами для всех категорий пользователей

```
Int fd = shmget(100, 1024, IPC_CREAT|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.





Семинар 7

System V, signal

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.

В классическом юникс было определено 20 сигналов (Define NSIGN 20). В современных их больше:

#define NSIG 20

#define 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

SIUSR2 17

SIGCHLD – сигнал, который получает предок при завершении потомка

Макросы

#define SIG_DFL(int(*)()) 0

#define SIG_IGN(int(*)()) 1

//(две 1 не смущают, так как располагаются в разных местах системного вызова сигнала

Signal(1, 1)//

Средством посылки и приема сигналов в ОС unix служат 2 системных вызова - signal() и 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 посылается группе процессов, =0 – всем процессам с идентификаторами группы, совпадающим с идентификатором группы процесса вызвавшего *kill* (то есть *pid* – не всегда идентификатор)

Пример

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 AMSI C и, следовательно, имеется во всех UNIX. 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.

Signal не входит в POSIX, поэтому его использование не рекомендуется, так как его поведение в system 5 отличается

от поведения в bsd.

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;
}
```

(см методичку оптимизация fork)

Системный вызов, который входит в POSIX – *sigaction*

```
Int sigaction(int sig_num, struct sigaction *action, Struct sigaction *old_action)
```

Struct `sigaction` определена в библиотеке `signal.h`

Процесс, определив собственную реакцию на сигнал, 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

PIPE

(буквальный перевод - труба)

Изначально были созданы неименованные программные каналы, потом именованные.

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 фалов. Это потоковая модель передачи данных. Симплексная связь, односторонняя. Для двусторонней – дупликсной, 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.

Именованные программные каналы создаются командой `mknod` (в командной строке писали `mknod <имя> p`)

`Mknod(<имя>, FIFO | ACCESS, 0)` – это можно сказать фактические параметры (а не формальные, они пишутся с типами)

АП защищенное. Программные каналы могут создаваться только в АП ядра системы.

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.

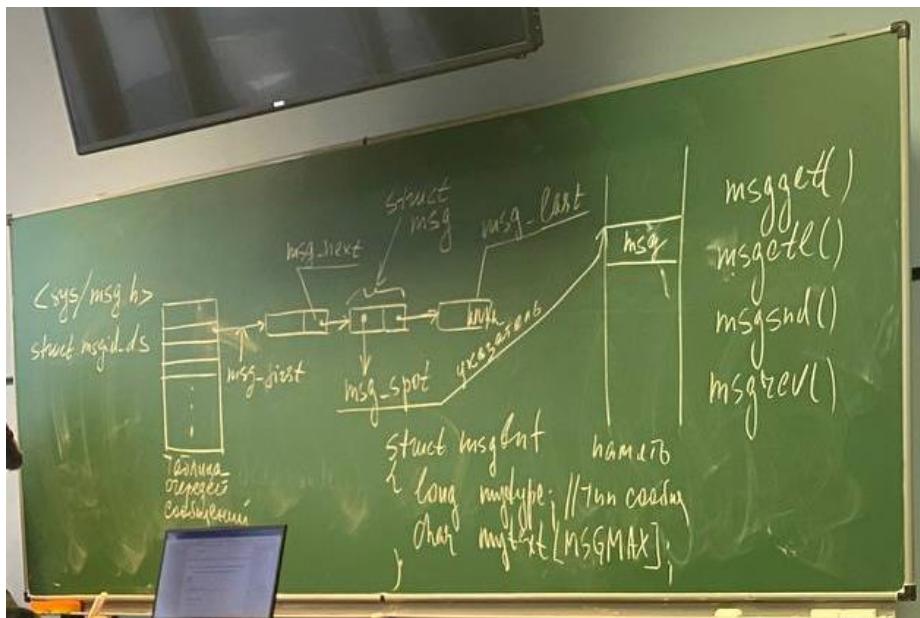
Еще раз – разница между каналами и разделяемой памятью. Канал обеспечивает потоковую модель передачи данных. При считывании сообщения оно перестает существовать. Он имеет встроенные средства взаимоисключения. Разделяемая память не имеет. Это буфер,

подключающийся к АП, нет средств взаимоисключений, поэтому их обычно используют с семафорами. Что положили – так и будет там лежать

Последнее средство взаимодействия – очереди сообщений

В ядре системы существует таблица очередей сообщений

Каждый элемент таблицы (очереди) имеет указатель на следующее сообщение, последний – NULL. Связный список. FIFO.



<sys/msg.h>

Struct msgid_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()

Msgctl()

Msgsnd() – положить сообщение

Msgrev() – прочитать сообщение

Определена структура

struct msg_buf

{

Long mytype; //тип сообщения

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 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)

Пример

```
#ifndef MSGMAX  
#define MSGMAX 1024  
  
#endif  
  
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.

На очередях сообщений определено много флагов. IPC_EXCL – прочитать самостоятельно