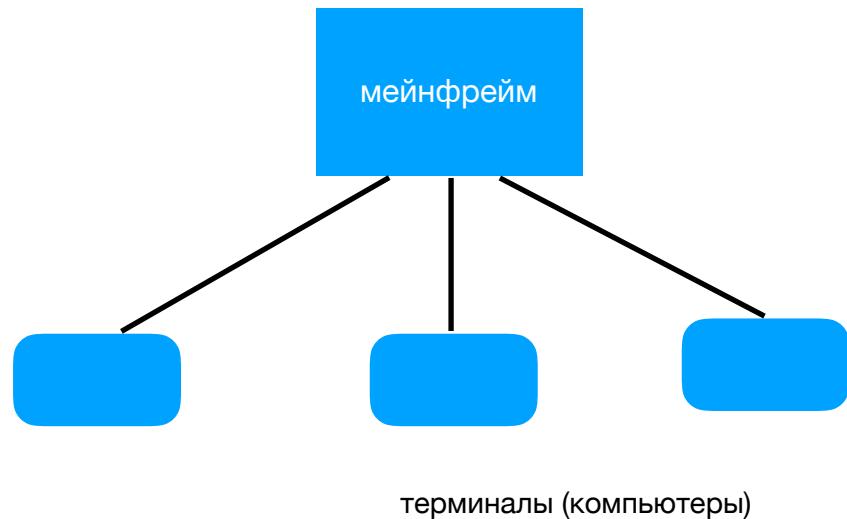


Процесс - программа в стадии выполнения  
сигналы поступают на вход контроллеру прерываний и формирует сигнал прерываний отправляемый в процессор



терминал - совокупность монитора и клавиатуры  
такая система гарантирует время ответа не более 3х секунд (должна)  
каждая программа/задание выполняется определенное кол-во времени, если не успевает,  
то переходит к другой задаче

системы разделения времени - процессоров время разделяется между выполняемыми  
программами (программами с которыми через терминалами взаимодействуют люди)  
в этих системах время квантуется

unix создавалась как система разделения времени

классификация ОС:

1. однопрограммная/однозадачная система программной обработки
2. мульпрограммная/мультизадачная/многозадачная система пакетной обработки
3. системы разделения времени
4. системы реального времени

современные ОС отличаются файловыми подсистемами

квант - процессоров время

ОС реального времени это ОС которые управляют вычислительные системы, которые в свою очередь управляют внешними по отношению к вычислительной системе устройства и такая система должна формировать ответ на полученные запросы в течении определенного интервала времени

жесткие системы реального времени

Дисциплины в этом семестре:

1. Управление процессорами
2. Управление памятью (оперативной)
3. Взаимодействие параллельных процессов

В следующем семестре (unix):

1. Управление данными
2. Вопросы управления устройствами

## **УПРАВЛЕНИЕ ПРОЦЕССОРАМИ**

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

Ресурс - любой из компонентов вычислительной системы и предоставляемыми ею возможностями

основной задачей ОС является управление процессами, в выделении этим процессам ресурсов вычислительный системы

ОС (Дейтон) - набор программ как обычных так и микро программ которые обеспечивают возможность использования аппаратуры компьютера при этом аппаратура компьютера обеспечивает сырую мощность .....

Ресурсы:

1. процессорное время
2. объем оперативной памяти
3. устройство ввода-вывода
4. таймеры
5. данные
6. ключи защиты

При разработке ОС решаются задачи

Система общего назначения - система, которая выделяет n-е количество стандартных/ общих задач

системы специального назначения (автопилот)

особенности построения ОС:

1. определение абстракций (основных понятий которыми оперирует или будет оперировать ОС) (процессы, потоки, файлы, симофоры)
2. предоставляемые системой примитивные операции (системные вызовы) (средства нижнего уровня)
3. защита (системы от пользователей, пользователей друг от друга, обеспечение совместного использования данных и ресурсов, изоляция отказов)
4. управление аппаратурой

процесс - это программа в стадии выполнения

в современных ОС процесс является основной абстракцией

именно процессу выделяются ресурсы системы

именно выполнением процессов управляет вычислительная система

ОС поддерживают выполнение параллельных процессов

факторы влияющие на сложность разработки ОС:

1. большой объем кода
2. подсистемы ОС взаимодействуют друг с другом (ОС много поточные)
3. параллелизм
4. хакеры
5. совместное использование ресурсов системы параллельными процессами
6. высокая степень универсальности
7. переносимость
8. совместимость с предыдущими версиями

определение реального времени по POSIX (стандарт 1003.1):  
реальное время в ОС - это способность операционной системы обеспечивать требуемый уровень сервиса в определенный промежуток времени

## ИЕРАРХИЧЕСКАЯ МАШИНА МЕДНИКА-ДОНОВАНА

построена с точки зрения процессов (в основном)



планирование - это определение места процесса в очереди к процессору (в очереди к процессору могут находиться только готовые процессы (готовые процессы - процессы которые получили все необходимые им ресурсы))  
очередь за какое-то время должна быть организована, этим занимается часть бедра - планировщик

управление процессами (нижний уровень):

диспетчер - задачи: непосредственное выделение процессам процессорного времени

в системах задача выделения ресурсов важная и связана не только с выделением ресурсов, а также с редактированием соответствующей системной информацией

процессору не просто выделяется процессоров время, но еще в системе регистрируется информация

верхний уровень:

если у системы нет возможности произвести процессы, то процессы откладываются

создание процессов связано с его идентификацией  
в системах unix идентификатор процесса - целое число  
процесс описывается структурой  
большинство полей являются указателями (большинство полей указывают на другие структуры)  
дискриптор - структуры которые описывают процесс

дискрипторы процессов находятся в основной таблице - таблице процессов (это мб связным списком, тк их редактировать удобнее чем массивы)

удаление процесса связано с освобождением ресурсов занимаемых процессов

планировщик процессов и управление памятью

управление устройствами:

имеет несколько уровней: верхний уровень управления устройствами

символьный уровень - самый высокий уровень в файловой подсистеме (позволяет именовать файлы)

в системе все файлы

устройства тоже файлы

интерфейс - набор функций

различаются:

1. непрозрачный интерфейс (каждый уровень может обратиться только строго к низлежащему уровню)
2. прозрачный (любой уровень может обратиться к низлежащему уровню)
3. полупрозрачный (какие-то уровни строго к низлежащему, какие-то через уровни)

## УПРАВЛЕНИЕ ПРОЦЕССОРАМИ

процесс выполняется когда получает процессоров время

для того чтобы процесс смог выполняться процессоров время ему должно быть выделено  
процессы устанавливаются в очередь процессору по каким-то критериям  
это управление процессорами

**диаграмма состояний процесса (ДСП)**



1е состояние: порождение процесса (процессу выделяется строка в таблице процессов)  
процесс идентифицируется (получает дескриптор)

получает необходимую память

переходит во 2е состояние

2е состояние: готовность (мб запущен)

в состояние готовности переходят процессы получившие все необходимые ресурсы  
организуется очередь готовых процессов по одной из выбранных в системе дисциплин  
планирования

процесс находящийся в этой очереди 1м переходит в состояние 3 когда ему выделяется  
процессоров время

3е состояние: выполнение

может перейти не только в 4е состояние (запрос ресурса), но и в состояние 5  
в состояние 2 (вытеснен или истек квант)

4е состояние: блокировка

после того как процесс получил нужный ему ресурс он переходит в состояние 2

5е состояние: завершение

определенением места процесса в очереди готовых процессов занимается планировщик

планирование обеспечивает процессам предоставление процессорам на основе принятой системы целевой функций

на целевую функцию (реализацию) влияет сив системы

планирование в системах пакетной обработки будет отличаться от планирования в системах разделения времени

особое место в задачах планирования занимают системы реального времени

в отличие от систем разделения времени которые должны обеспечивать приемлемое время ответа системы

системы пакетной обработки необходимо обеспечить производительность системы

классификация алгоритмов планирования:

- с переключением и без переключений (переключение процессорами на новое задание в системах с квантованиями)
- с приоритетами и без
- с вытеснениями и без

приоритеты бывают : абсолютные, относительные (определяются относительно другого значение), статические, динамические

алгоритмы планирования в системах пакетной обработки:

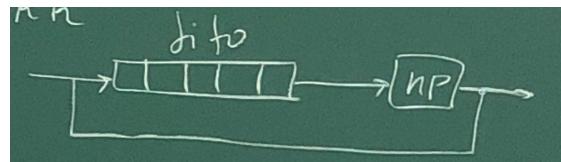
1. fifo (планирование без приоритетов без переключений) (процесс выполняется от начала и до конца, не блокируется)
2. SJF - Shortest Job First (первыми в очереди становятся процесс заявленное время выполнения которых меньше) (бесконечное откладывание, тк могут никогда не быть выполнеными) (с приоритетами без переключениями) (при вытеснении необходимо сохранять полный контекст)
3. SPT - Shortest Remaining Time (наименьшее оставшееся время) (выполняющийся процесс мб вытеснен с выполнения, если в очередь готовых процессы поступит процесс с наименьшим оценочным временем выполнения) (нужно следить за временем выполнения процессов) (процесс продолжается с того места на которым был прерван, поэтому необходимо сохранять информацию) (процесс будет возобновлен со следующей команды после команды на которой он был вытеснен, необходимо напомнить содержимое регистров процесс и счетчика команд который всегда содержит адрес счетчика команд (сохранение аппаратного контекста)) (бесконечное откладывание)
4. HRRN - Highest Response Ratio Next (наибольшее относительное время) (в этом алгоритме приоритет вычисляется по формуле:  $r = (tw + ts)/ ts$ ; где tw - время нахождения процесса в очереди готовых процессов; ts - запрошенное время обслуживания) (приоритет процесса увеличивается в зависимости от времени ожидания , т.е. исключается бесконечное откладывание) (с приоритетами без вытеснений)

## 1. RR

в системе разделения времени если процесс не завершился он возвращается в конец очереди

это называется система с переключениями

при переключении процессов переключается полный процесс (состоит из аппаратного контекста и информацией о выделенных процессу ресурсах, в частности о выделенной процессу памяти)



блокировка процессов

ни одна система не разрешает обратиться напрямую к устройству ввода-вывода

это системные вызовы которые переводят систему в режим ядра

процесс часть времени выполняется в режиме задачи (выполняет свой код), а часть в режиме ядра (выполняет код ОС)

драйвер - управляет работой внешних устройств

когда устройство формирует сигнал прерывания это приводит к обработчику прерываний

обработчик прерываний устройств входит в состав драйверов устройств

процесс должен получить какую-то информацию для его успешного продолжения

переключение в состояние блокировки и возможность перехода в состояние готовности связано с ???переключением??? контекста

переключение аппаратного контекста системой поддерживается аппаратно

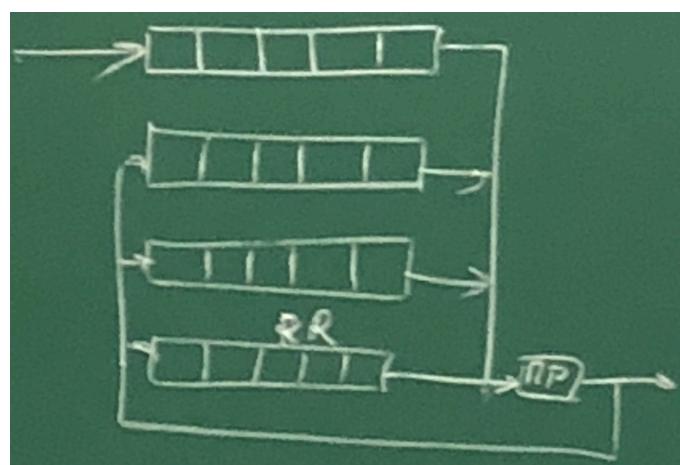
основным недостатком алгоритма RR является статический квант времени  
в любом случае процесс получает один и тот же квант времени

Очередь FCFS (первым пришел - первым обслужан)

в современных системах процессы создают по мере необходимости, ресурсы выделяются по мере надобности

АЛГОРИТМ: многоуровневые очереди (адаптированное клонирование)

алгоритм построен на ряде очередей



первую очередь в самую высоко приоритетную попадают только что созданные процесс или процесс завершившие ожидание на вводе-выводе

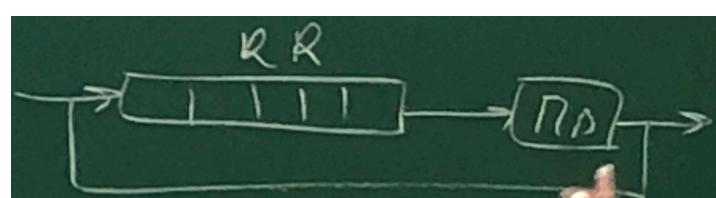
если процесс за выделенный ему квант процессорного времени не успел завершиться или ... он переходит в другую низкоприоритетную очередь, и т д, пока не окажется в самой низкоприоритетной очереди в которой работает алгоритм RR и будет там работать пока не завершится

в этой же низкоуровневой очереди работает

в самой высоко приоритетной очереди окажутся интерактивные процесс (процесс которые требуют ввод с клавиатуры или какие-либо действия с мышью)

в windows приоритет процесс изменяется в зависимости от того на каком устройстве процесс был заблокирован (на звуковой карте: приоритет повышается на 8, на мыши или клавиатуре: повышается на 6)

если обнаруживается процесс который находился в очереди больше заданного времени то приоритет такого процесса повышается на несколько единиц  
с точки зрения реализации это значит



в windows реализована вытесняющая многозадачность  
режимы:

1. режим пользователя (режим задачи)
2. режим ядра

программа выполняется в режиме пользователя

все действия связанные с обслуживанием процесса в режиме ядра

## **ПЕРЕКЛЮЧЕНИЕ В РЯЖИМ ЯДРА**

3 типа событий переводящих систему в режим ядра:

1. системные вызовы (программные прерывания SVI)
2. исключительные ситуации (exceptions)
3. аппаратные прерывания

системные вызовы - запросы процессов на обслуживание системы  
ни одна система не позволяет напрямую процессом обращение к устройствам ввода-вывода

если разрешить такие обращения, то систему защитить невозможно  
система предоставляет соответствующие функции (примитивы, тк низкоуровневые (ядро) действия) (API)

системные вызовы являются синхронными событиями

исключения: связаны с ошибками в коде, с ошибками выполнения в процессе  
в системе существуют 2 типа исключений:

1. неисправимые
2. исправимые (страничные прерывания)

исключения являются синхронными событиями по отношению к выполняемым ...

аппаратные прерывания:

1. прерывания от системного таймера
2. прерывания от устройств ввода-вывода (программируемый контроллер прерываний (получив этот сигнал процессор посыпает контроллеру этот сигнал, получив сигнал контроллер поставляет на ветром шину прерывания и т д....))
3. пребывания от действий оператора ()

аппаратные прерывания это абсолютно асинхронные события в системе  
они возникают независимо от каких-либо действий которые выполняются в системе

## **ПОТОКИ**

2 типа контекста:

1. аппаратный контекст
2. полный контекст

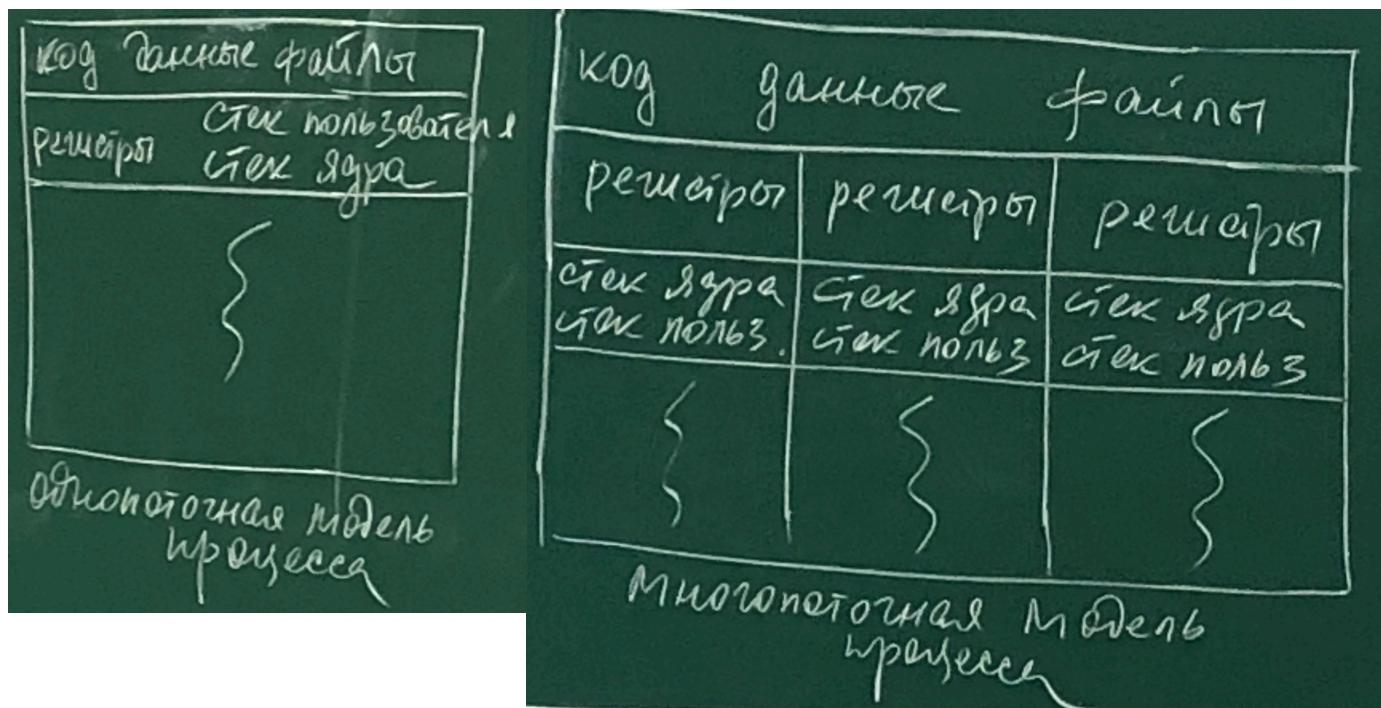
чтобы система могла управлять процессом

каждый процесс в системе управляется структурой - дескриптором процесса

переключение полного контекста является затратной операцией

поток - это часть кода процесса (непрерывная часть), которая может выполняться параллельно с другими частями кода программы, выполняя при этом какую-то задачу, в рамках общей большой задачи, которую решает программа

поток не имеет собственного адресного пространства и выполняется в адресном пространстве процесса, т.е.. они разделяют адресное пространство процесса  
процесс является владельцем ресурсов  
владельцам ресурсов является процесс

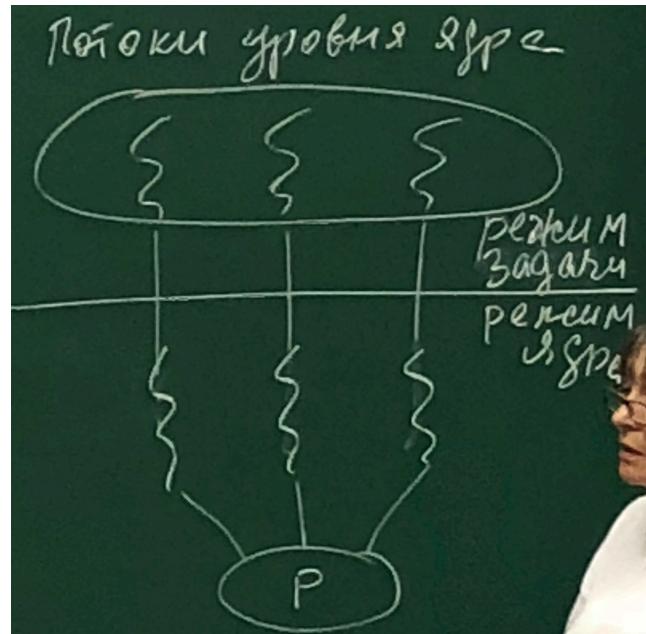


файлы принадлежат процессу  
потоки владеют регистрами  
единицей диспетчеризации стал поток  
поток получает процессоров время  
речь идет о потоках ядра  
у потока должно быть 2 стека

различаются потоки уровня ядра и потоки уровня пользователя

потоки уровня ядра

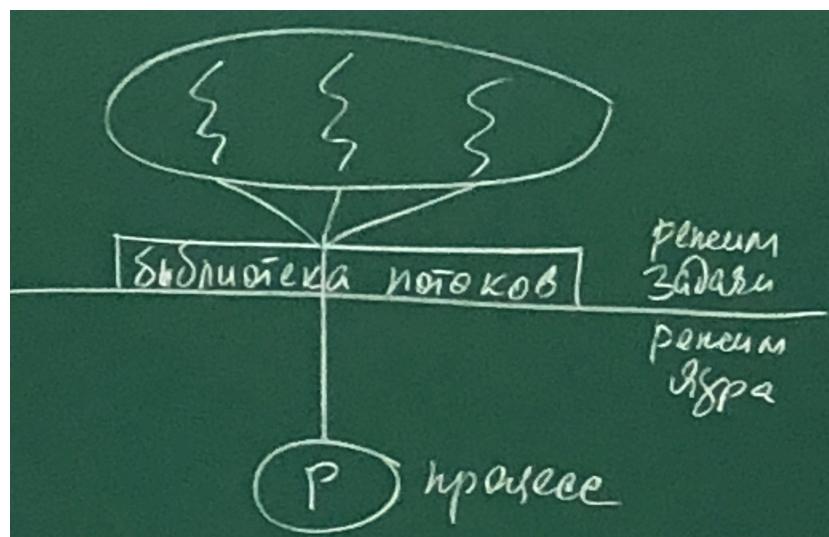
каждый поток описывается собственной структурой у которого есть ссылки



при переключении потоков переключается только аппаратный контекст  
т.е. если переключение осуществляется между потоками одного процесса, то  
переключается только аппаратный контекст

в очереди готовых потоков находятся потоки разных процессов в разной  
последовательности

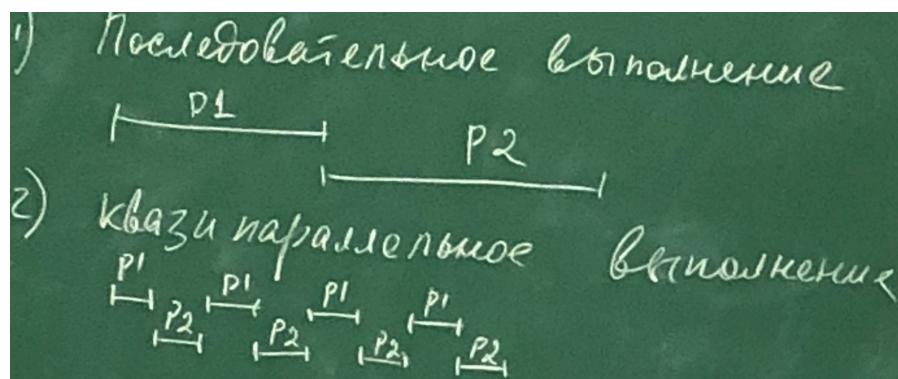
когда происходит переключение потока другого процесса будет переключен полный  
контекст



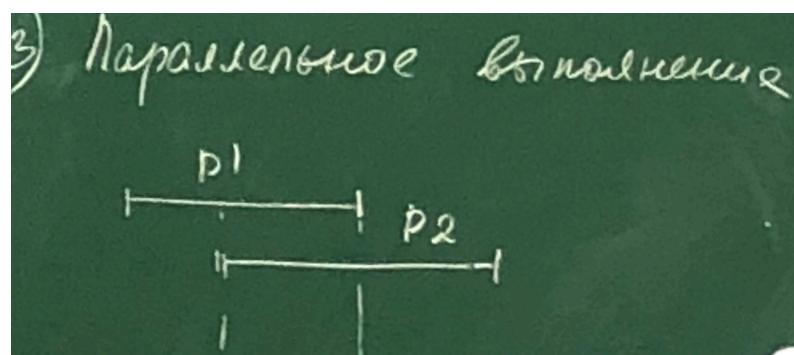
потоки уровня пользователя - это потоки о которых ядру ничего неизвестно  
выполнение этих потоков выполняет сам процесс без участия ядра  
для этого необходима специальная библиотека, которая предоставляет функции для  
работы с потоками: создание потоков, удаление потоков, планирование выполнения  
потоков, переключение контекстов, должна обеспечивать возможность взаимодействия  
потоков друг с другом

если какой-то поток запросил сервер системы, то будет заблокирован весь процесс  
все хорошо в этой схеме, пока потоки не запрашивают никаких системных ресурсов

### УРОВНИ НАБЛЮДЕНИЯ



выполнение команд программ совпадает по времени, но на разных процессорах:



проблемы взаимодействия параллельных процессов как в однопроцессорной так и в многопроцессорной системе будут одинаковыми

## ПРОЦЕССЫ UNIX

юникс принято рассматривать процесс как выполняющийся в двух режимах

процесс unix часть времени выполняется в режиме задачи (в режиме пользователя) и тогда он выполняет собственный код, а часть в ревени в режиме ядра выполняет реентерабельный код ОС

реентриабильный код - код чистых процедур

чистая процедура - процедура, которая не модифицирует саму себя

данные ОС хранятся в системных таблицах

В юникс подобной системе линукс процессы создаются единообразно .....

функция ядра - функция fork()

процесс fork() выполняет следующие действия: в результате системного вызова форк создается системный процесс, который связан с процессом вызвавшим форк отношением предок-потомок

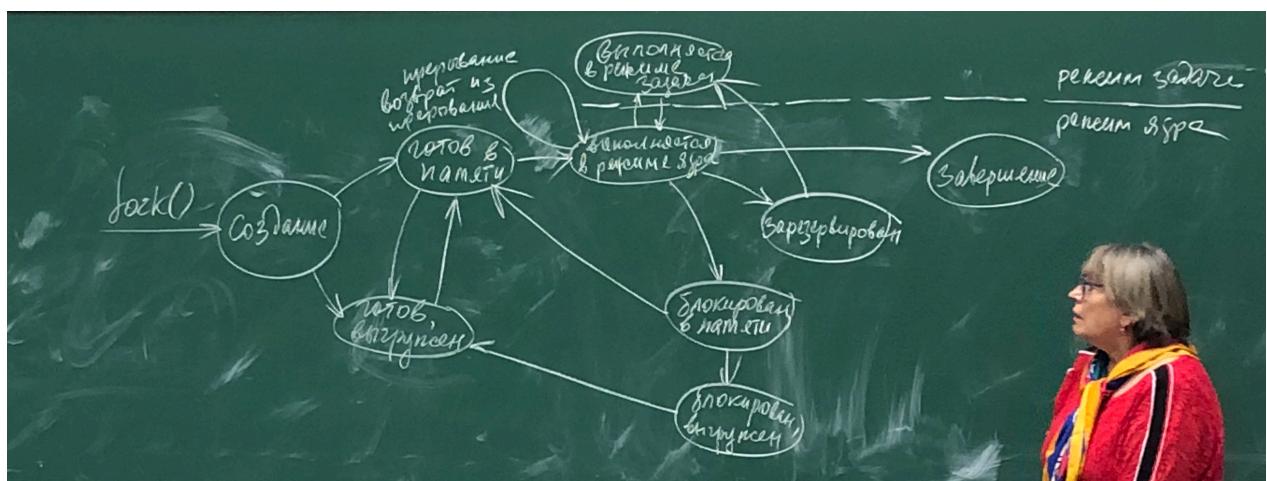
процесс потомок является копией процесса предка в том смысле, что он наследует код предка, открытые файлы, сигнальную маску и т.д.

для процесса потомка создается собственное адресное пространство, т.е. создаются собственные так называемые карты трансляции адресов

в современных системах это таблицы страниц, т.е. в современных системах виртуальное адресное пространство, для того чтобы этими пространствами оперировать создаются таблицы виртуальные (таблицы страниц)

т.е. для потомка создается виртуально адресное пространство, т.е. таблицы страниц эти таблицы страниц содержат адреса страниц адресного пространства предка таким образом выполняется наследование кода предка

диаграмма состояний процесса ДСП unix



для того, чтобы показать, что практически постоянно выполняется переключение контекста

в линукс процесс описывается структурой struct task\_struct  
в unix ДСП struct proc

эти структуры досланы содержать одинаковую информацию  
система оперирует набором двухсвязных списков  
дерево иерархическая структура которая отражает отношения процессов как предков и потомков

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

выполняется. режиме задачи: выполняет собственный код

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

1. системных вызовах
2. исключениях
3. аппаратных прерываниях

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

ядро должно переключить контекст

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

процесс мб вытеснен другим более приоритетным процессом

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

## УПРАВЛЕНИЕ ПАМЯТЬЮ (УП)

выделяются 2 уровня УП  
у системы есть иерархия памяти  
она рассматривается по отношению к процессору (насколько близко данный вид памяти находится к процессору)

1. оперативная память
2. вторичная память (внешняя память)

работой наших процессоров управляют микрокоманды (микропрограммы) которые находятся в нанопамяти

уровни иерархии вторичной памяти:  
1. уровень внешней памяти  
2. оперативная память  
3. кэши в кристалле

задачи управления памятью делится на вертикальную (передача информации с уровня на уровень) и горизонтальную (управление каждым конкретным уровнем)

задачи управления внешней памятью решаются с помощью файловых систем  
???задачи управления вторичной памятью задача хранения информации ???  
информация хранится структурированно  
файл - любая поименованная совокупность данных расположенная во вторичной памяти  
если речь идет о диске - любая поименованная совокупность данных расположенная на диске

занимается управление уровнем - файловые системы

задача файловой системы - обеспечить длительное хранение и доступ к файлам в оперативной памяти также имеется большое количество кэши которое позволяет более быстро искать информацию о файлах bla bla bla))

система должна обеспечивать решение задач управления памятью

задачи управления памятью: кому, сколько, когда, где

система должна выполнять задачу выделения памяти процессу

задачу учета выделения памяти процессу и свободной

задача освобождения памяти и возвращение ее в пул свободной памяти

задача разделения физического адресного пространства оперативной памяти между различными процессами

процессы создаются по мере необходимости

ресурсы выделяются по мере надобности

классификация алгоритмов памяти:

1. связное распределение памяти (когда программа занимает непрерывный блок адресов оперативной памяти, т.е. в память она загружается целиком, в последовательные адреса образуя блоки)
2. несвязное распределение (программа сама делится на блоки (сегменты) и эти блоки могут находиться в память не подряд, а в разброс)

одиночное непрерывное распределение

-----  
ОС | граничный регистр (переход за эту границу вызовет ошибку)  
-----

программа|



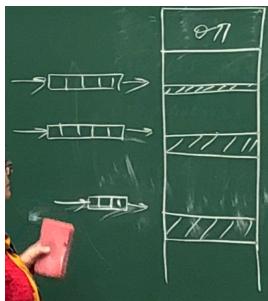
в этой системе все ресурсы предназначены только для одной программы

программа выполняется от начала до конца

в процессе выполнения она получает все необходимые ресурсы

распределение памяти фиксированными разделами

схема распределения памяти разделами фиксированного размера

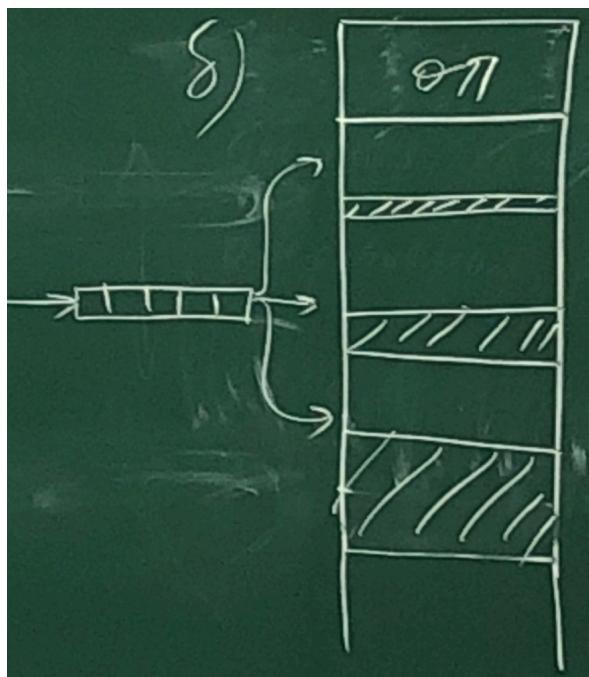


память делится на разделы

размер раздела определяется статически (соответствует наиболее часто встречающимся размерами программ)

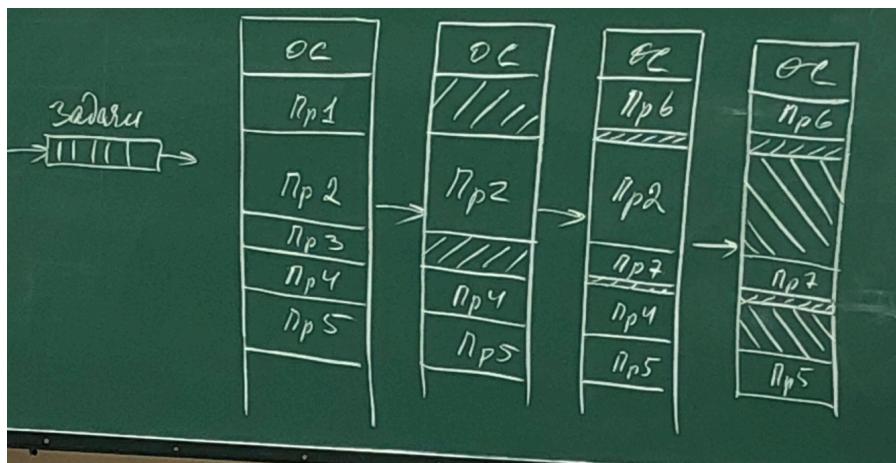
программа не может быть точно равна размеру раздела => в каждом разделе будет оставаться незанятое адресное пространство

какому-то разделу стоит длинная очередь, а какой-то раздел обслужил все программы которые стояли в очереди (неэффективное использование памяти)



решили сделать одну очередь

распределение памяти разделами размер которых определялся динамически



в освободившееся пространство мы должны загрузить новые программы  
возникнет следующая ситуация  
напрашивается решение когда надо объединять куски памяти

3 стратегии (объединения памяти)

1. первый подходящий
2. самый тесный (из всех свободных размеров выбирается раздел чтобы осталось меньше всего свободного пространства)
3. самый широкий (с расчетом на то что после загрузки останется достаточно большой раздел чтобы поместились еще одна программа, возникает задача редактирования разделов в один непрерывный раздел)

нужно иметь 2 таблицы:

1. таблицу выделенных разделов

| Таблица выделенных разделов |        |       |         |
|-----------------------------|--------|-------|---------|
| №                           | размер | адрес | своб. е |
| 1                           | 8K     | 312K  | распц   |
| 2                           | 32K    | 580K  | распц   |
| 3                           | -      | -     | пустой  |
| 4                           | 120K   | 384K  | распц   |
| 5                           | -      | -     | пустой  |

2. таблица свободных областей

| Таблица свободных областей |        |       |         |
|----------------------------|--------|-------|---------|
| №                          | размер | адрес | своб. е |
| 1                          | 32K    | 352K  | послед  |
| 2                          | 520K   | 584K  | послед  |
| 3                          | -      | -     | пустой  |
| 4                          | -      | -     | пустой  |

фрагментация памяти - ситуация когда возникают дыры в памяти в которые нельзя ничего загрузить

фрагментация могла пожирать до 30% памяти

если фрагментация достигла критического уровня раньше использовалась перезагрузка памяти (СЕЙЧАС НЕТ!!!)

теряется проделанная работа

чтобы решить задачу фрагментации нужно переместить, но чтобы переместить нужно иметь возможность это сделать а что для этого надо изменить ОНА СКАЖЕТ В СЛЕДУЮЩИЙ РАЗ!!!!!!!!!

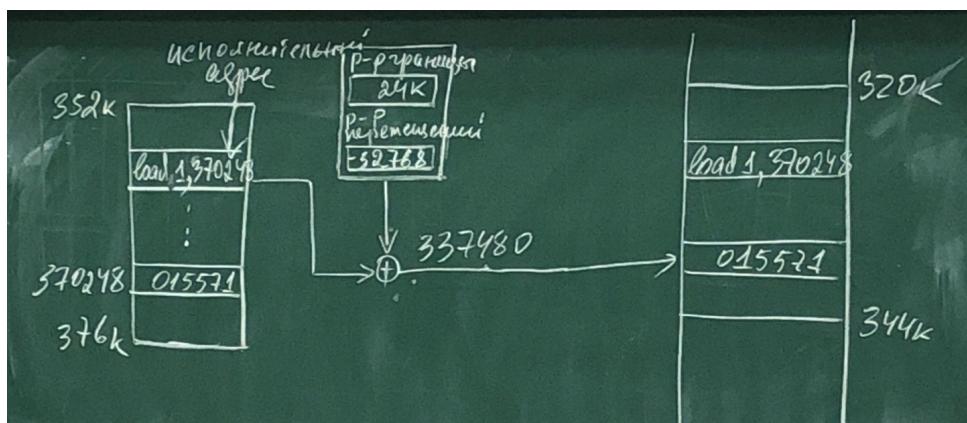
09.10.18

различается 3 типа отображения программы в память:

1. абсолютная трансляция (трансляция в абсолютных адресах) (устанавливает взаимно однозначное соответствие меток и адресов)
2. статическая трансляция (в процессе компиляции адреса кода определяются относительно некоторого фиксированного адреса памяти)
3. динамическая трансляция (выполняется таким образом, что в реальные адреса расположение программы в памяти определяются операционной системой при загрузке)

при статической трансляции код программы связан с начальным адресом и не предполагается никаких перемещений

чтобы переместить программу которая связана статически с некоторыми адресами памяти первоначально был предложен следующий подход:  
в состав процессора были введены регистры перемещения



для того чтобы переместить программу на какие-то другие адреса в регистр перемещения нужно поместить значение разницы этих адресов

возникает задача преобразования адреса

в регистр перемещения устанавливаем величину перемещения и каждый адрес программы преобразовываемая нужное перемещение

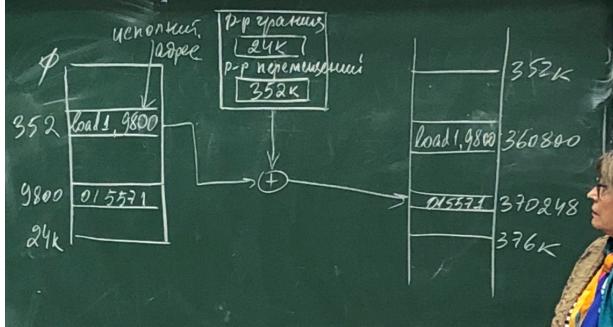
в результате преобразования мы получим новый адрес и сможем обратиться по нужному нам новому адресу

если отказаться от статической трансляции и определять относительные адреса программы, т.е. если нет смысла связывать программу с конкретным адресом в памяти, то как оформить адресацию?

каждая программа считает, что она начинается с 0го адреса, т.е. в программе адреса вычисляются смещением

связывается начальный адрес программы с каким-то адресом в памяти

=> программа начиная с этого адреса располагается подряд, но адреса вычисляются: начальный адрес + смещение



в итоге мы получаем возможность перемещать разделы памяти

когда выполнять такие перемещения:

1. сразу после освобождения какого-либо раздела
2. если не найден раздел нужного размера

в обоих случаях таблицы в которых выполняются управления памятью должны быть отредактированы

во втором случае перекомпоновка памяти будет требовать большего объема редактирования соответствующего объема от памяти

лучше не таблицы, а связанные списки

все что мы рассмотрели касается связного распределения памяти

## НЕСВЯЗНОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ

есть код, есть статические данные, обязательно должен быть раздел адресного пространства выделенный для стека

программу делили на равные участки, эти участки - страницы  
необязательно загружать страницы подряд в памяти

никогда программа не обращается одновременно ко всем своим страницам -  
неэффективное расходование памяти

можно ли выполнять программу, которая целиком не находится в памяти?

можно

если в памяти находятся части программы к которым в данный момент происходит обращение, т.е. это виртуальная память

виртуальная память - память, размер которой превышает объем физического адресного пространства

чтобы управлять такой памятью необходимы соответствующие таблицы  
виртуальное адресное пространство описывается соответствующими таблицами

существует 2 схемы управления виртуальной памятью:

1. страницами по запросам (страничное распределение)
2. сегментами по запросу (сегментное распределение)
3. сегментами поделенными на страницы по запросу (сегментно-страничное распределение)

по запросу - соответствующие страницы/сегменты загружаются в память когда было выполнено обращение к ним, при этом обращение к отсутствующему сегменту/странице приводит к возникновению исключения

в результате обработки исключения менеджер ядра выполнит соответствующие действия загрузит отсутствующую страницу/сегмент в память, в это время процесс будет блокирован (прерывание intal - 14е исключение)

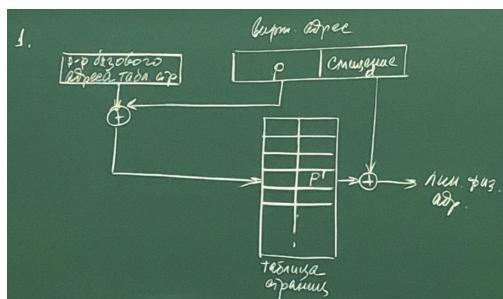
возникает новое адресное пространство (адресное пространство программы называется логическим (оно определяет смещения, т.е. смещения берутся из программы))  
виртуальное адресное пространство, его описывают соответствующие таблицы  
физическое адресное пространство

## УПРАВЛЕНИЕ ПАМЯТЬЮ СТРАНИЦАМИ ПО ЗАПРОСАМ

рассмотрим задачу преобразования адресов  
принято страницы виртуального адресного пространства называть page  
страницы физического frame  
но мы всегда называем page

существует 3 схемы преобразования виртуального адреса в физический:

1. прямое преобразование (отображение)



виртуальное адресное пространство описывается таблицей страниц  
таблица страниц должна находится в оперативной памяти  
в процессоре имеется регистр базового адреса таблицы страниц  
виртуальный адрес на 2 части: смещение d, номер страницы p  
имеется таблица страниц  
таких таблиц в системе будет столько, сколько процессов  
в такой таблице будут находиться дескрипторы страниц  
дескриптор - структура которая описывает или инфо которая необходима для работы с сегментами или инфо для работы со страницами  
в таблицы страниц находится виртуальный адрес страниц, ...  
номер страницы используется как смещение дескриптора страницы в таблице страниц  
если страница загружена в память, то выполняя преобразование мы можем получить линейный физический адрес байта памяти  
если страница не загружена в память, то возникнет страничное исключение

в системах есть жесткий диск дисковое адресное пространство делится на 2 неравные части  
1 часть больше управляет файловой системой и предназначена для хранения файлов этой частью диска управляет файловая система, этих систем мб несколько  
2 часть - область swapинга или pageинга

pageинг:

программа которую запустили на выполнения копируется в область pageинга и по необходимости страницы загружаются в физическую память  
когда в них отпадает необходимость они выгружаются на диск

шаринг: перемещение системы

в системе одновременно может находиться до 3х копий страниц/или сегментов одной программы, что неэффективно

копии:

1. наша программа продолжает находится на диске, где она файл
  2. запустили - она скопирована в область рабочего стола
  3. часть страниц загружена в физическую память
- это неэффективно

решение - одноуровневая память

идея: использовать адресное пространство файла для рабочего стола

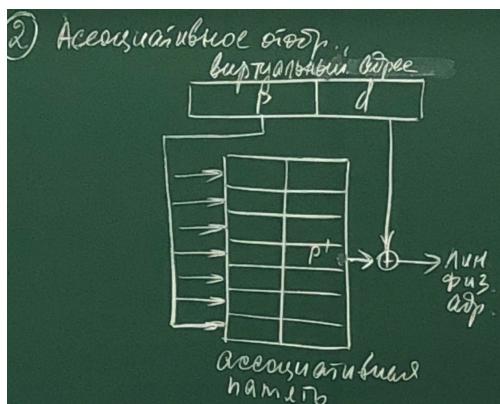
на каждой команде а то и несколько раз - обращение к памяти - затратное действие в системе

используется понятие - цикл обращения к памяти (много тактов процессора)

обращение к физической памяти - затратное действие

была предложена следующая схема

2. ассоциативное отображение  
для него используется ассоциативная память



ассоциативная память обеспечивает выборку информации по ключу  
ключ - номер страницы

выборка осуществляется за 1 такт за счет специальной схемы  
но виртуальная память очень дорогая  
она всегда регистровая

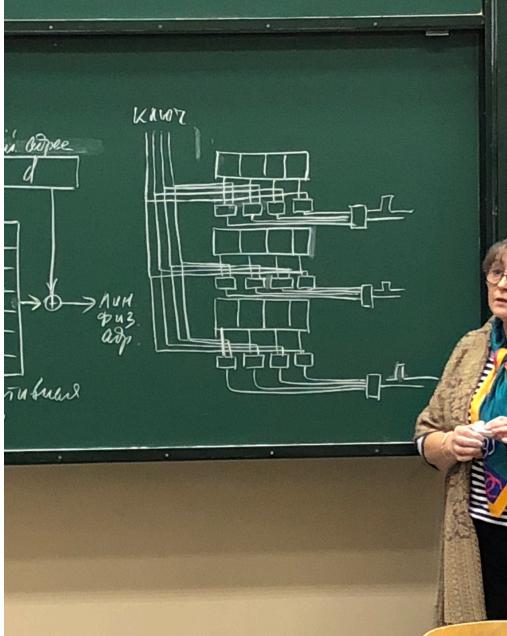


схема полностью ассоциативной памяти  
кол-во сравниваемых разрядов определяется  
разрядностью поля

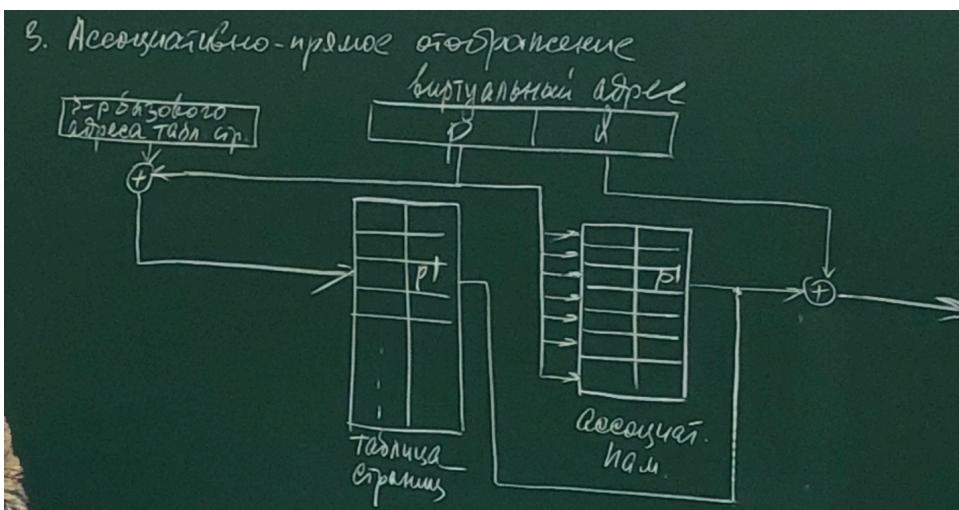
невозможно построить полностью ассоциативное схему

такая ассоциативная память должна помещать в себя все таблицы страниц

на каждый разряд каждого регистра ставится схема совпадения при совпадении информации записанной в строке с ключом схемами совпадения формируется разрешающий сигнал и происходит считывание информации за один такт

используется частично ассоциативная память - ассоциативно прямое отображение

### 3. ассоциативно-прямое отображение



есть таблица страниц которая находится в памяти

есть ассоциативный кэш, его размер 8-16 дескрипторов

размер ассоциативной памяти ограничен небольшим количеством дескрипторов

если физической странице нет в ассоциативном кэше, то она должна быть туда помещена

в кэше должны быть актуальные физические адреса страниц (физические адреса страниц, к которым были последние обращения)

небольшой объем ассоциативного кэша дает показатели скорости обращения около 90% по сравнению с полностью ассоциативной памятью (за счет подхода с последними обращениями) такой алгоритм называется LRU

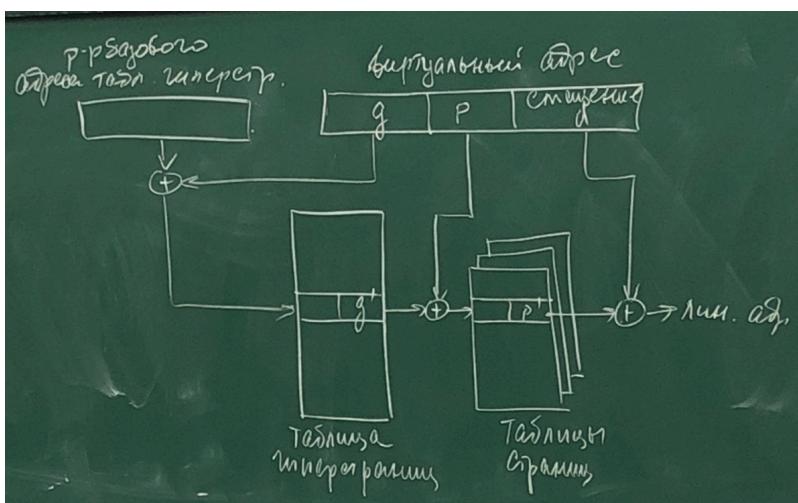
16.10.18

выигрыш это увеличение уровня мультипроцессности - в памяти одновременно может находиться большее число программ за счет ....

именно ассоциативно-прямое отображение используется в реальных системах

## ДВУХУРОВНЕВАЯ СТРАНИЧНАЯ РЕАЛИЗАЦИЯ

было предложено виртуальное адресное пространство процессов делить на гипер страницы, а гиперстраницы делятся на страницы  
номер гиперстраницы определяет смещение к дескриптору гиперстраницы в таблице кипер страниц  
а номер смещения



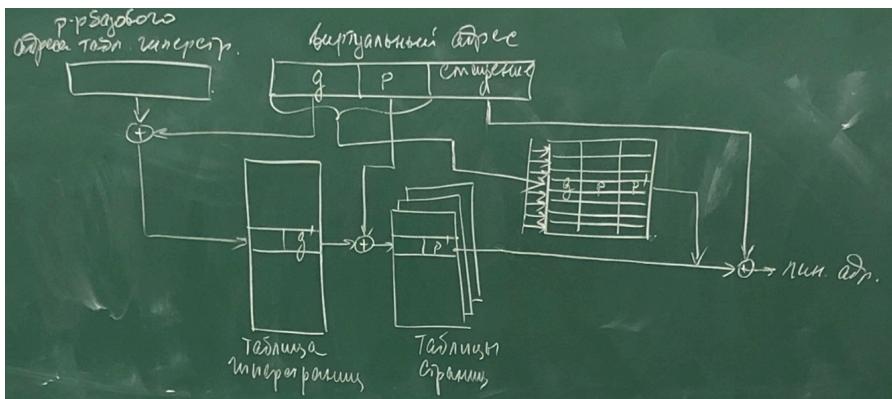
стандартная схема

таблица гиперстраниц  
называется каталогом таблиц  
страниц (в intel)

в результате виртуальный адрес делится не на 2 части, а на 3 => к одному преобразованию добавляется еще один  
общее время формирования физического адреса увеличивается  
таблицы страниц должны находиться в адресном пространстве ядра страницы  
память ядра это дефицитная область памяти и если у нас запущено огромное количество процессов то получается следующая ситуация: процесс никогда не обращается ко всем своим страницам одновременно

тк процессор не обращается ко всем адресному пространству одновременно, таблицы страниц занимают дифицитнейшую область памяти  
чтобы не держать в памяти таблицы страниц которые не используются вводятся гиперстраницы

виртуальный адрес делится на 4 поля  
чтобы сократить расходы связанные с обращением с физической памяти вводится ассоциативный кеш



лин.адр.физ.адр.

проблемы страничной организации виртуальной памяти  
 основные проблемы возникают при коллективном использовании страниц  
 это значит что многие пользователи в системах заинтересованы в выполнении одних и тех же программ при этом возникает проблема модификации страниц коллективно используемых программ  
 какой-то процесс заинтересован в модификации страницы, а другие нет  
 чтобы процесс мог обращаться к каким-то страницам программы у него должны быть ссылки на страницы  
 достоинство: то что адресное пространство делится на участки равного размера

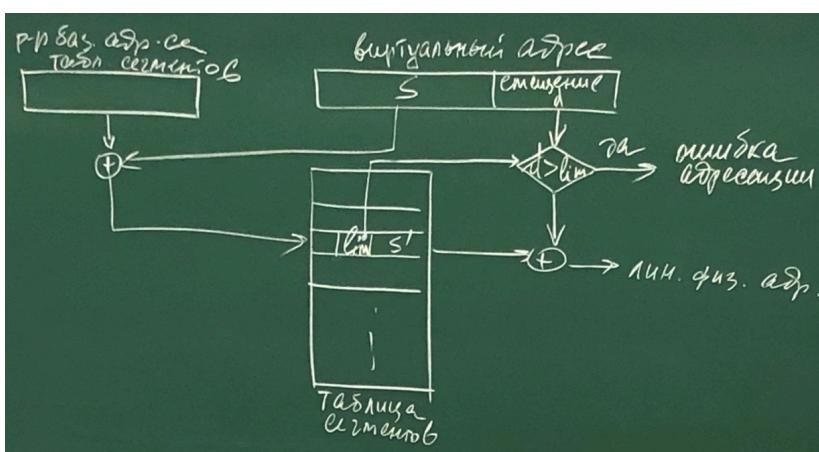
## УПРАВЛЕНИЕ ПАМЯТЬЮ СЕГМЕНТАМИ ПО ЗАПРОСУ

страница является результатом физического деления памяти  
 сегмент - единица логического деления памяти  
 размер сегмента определяется размером ... кода

в элементарной программе 3 сегмента

поле лимит ограничивает возможный размер сегмента  
 это поле используется для контроля выхода процесса за собственное адресное пространство

схема преобразований сегментированного адреса



виртуальный адрес делится на 3 части  
 в процессоре должен быть регистр базового адреса  
 таблицы сегментов  
 d - смещение

эта схема отражает для каждого процесса наличие таблицы сегментов  
 эта схема отражает: адресные пространства процессов должны быть защищены

адресное пространство ОС должно быть защищено от выполняемых программ  
адресное пространство выполняемых программ (процессов) должно быть защищено от  
самых себя, то есть:  
ни один процесс не может обратиться напрямую в адресное пространство другого  
процесса  
они защищаются: никотин процесс не может обратиться за собственное адресное  
пространство  
на схеме видим контроль такой ситуации

при страничном преобразовании также выполняется такая проверка: процесс может  
обращаться только к страницам, которые описаны в его таблицах страниц

## ОРГАНИЗАЦИЯ ТАБЛИЦ СЕГМЕНТОВ

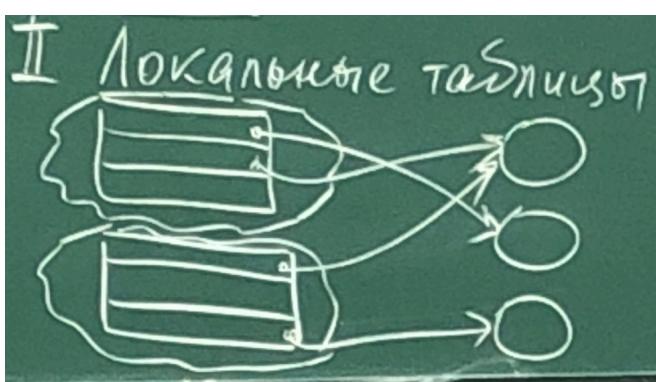
3 подхода к организации:

1. единая таблица сегментов



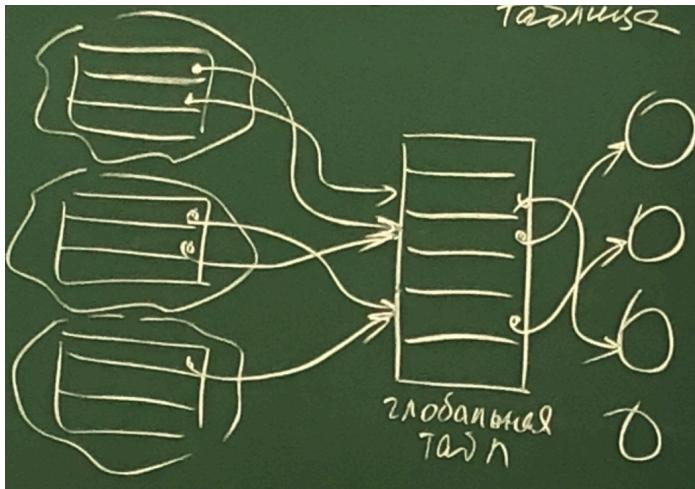
в системе имеется одна единная таблица  
сегментов => сегменты имеют только  
глобальные имена

2. локальные таблицы



адресное пространство процессов описанное локальными таблицами  
у нас есть какие-то разделы физической памяти и каждая такая локальная таблица должна  
содержать дескрипторы соответствующих сегментов физической памяти  
должна содержать ссылку непосредственно на физический сегмент

### 3. локальные таблицы и глобальная таблица



в отличие от 2й схемы локальные таблицы ссылаются на дескриптор глобальной таблицы  
должна содержать ссылку на дескриптор физического сегмента  
регистр LD...R содержит ....

сегмент является единицей логического деления памяти  
сегмент отражает какой-то код в выполнении которого мб заинтересованы многие  
процессы

остается проблема модификации  
для сегментов эта проблема решается более просто(логично), тк сегмент это какой-то  
участок кода логически завершенный, в отличие от страниц  
на уровне страниц практически теряется принадлежность к конкретному коду

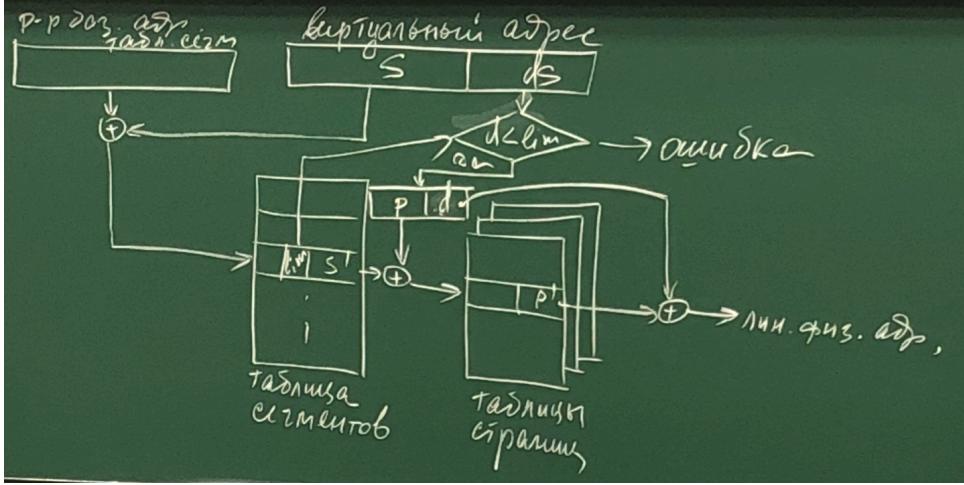
страницы управляемые по запросу. ??????????

страницы являются единицей физического деления памяти  
при этом в начальный момент когда процесс создается ему выделяются страницы  
сегмента входа, страница сегмента данных, страница сегмента схемы

у любой программы должно быть 2 стека: стек пользователя, стек ядра (используется  
когда процесс переходит в режим ядра, в этом режиме невозможно использовать  
пользовательский стек)

ситуация, когда нет свободных страниц  
если процесс не может загрузить нужную ему страницу он не сможет выполнять свое  
выполнение, чтобы смог надо вытеснить какую-то страницу: мы меняем страницу на  
страницу, тк они одного размера  
также рассуждаем для сегментов: возможна ситуация когда нет свободных сегментов  
нужно выгрузить сегмент, но сегменты разного размера, из-за этого мб такое, что  
выгружается не один сегмент

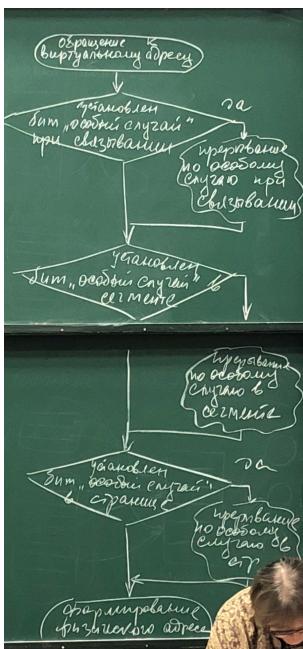
поэтому появилась 3я схема управление памятью сегментами поделенными на страницы



$p$  - указатель

у нас есть таблица сегментов, тк сегментов много у нас будет много таблиц страниц в данный схеме размер сегмента должен быть кратен размеру страницы

на примере этой схемы посмотрим сколько будет возникать прерываний



речь идет о том, что программа не обращается по всем своим адресам при конкретном выполнении она может не обращаться ко всем сегментам, к каким-то сегментом не будет вообще обращений

отложенное связывание - связывание, а именно соответствующая работа с сегментами выполняется только тогда, когда возникает нужда в конкретном сегменте

1. обработка прерывания по особому случаю при связыванию  
сегменту с конкретным именем присваивается номер строки в таблице сегментов и заполняются поля в дескрипторе сегментов, те пока к сегменту не было обращений он даже не имеет дескриптор

2. прерывание по особому случаю в сегменте  
у сегмента есть дескриптор, но обращение по конкретному адресу не мб выполнено пока по конкретному сегменту не создана таблица страниц  
когда возникает такое прерывание для сегмента создается таблица страниц и она сопоставляется с картой файлов  
сегмент находится на фиксе, необходимо сопоставить один адрес с другим  
адрес созданной таблицы страниц заносится в таблицу сегмента

дескриптор сегмента содержит базовый адрес таблицы страниц  
3. пребывание по особому случаю в странице  
возникает когда нужная страница отсутствует в физической памяти  
при обработке этого прерывания менеджер памяти должен загрузить отсутствующую  
страницу в физическую память  
при этом может выполняться замещение страниц  
после того как страница загружается в память, соответствующая таблица страниц  
редактируется, т.е. в соответствующий дескриптор заносится физический адрес страницы  
очевидно, что такое деление правомерно и необходимо

## **регистр CR3 - регистр начального адреса каталога таблиц страниц**

### **copy\_on\_write**

только вот к чему это????? 😕

### **PAGE REPLACEMENT (замещение страниц)**

при запуске программы процесс прошёл идентификацию выделении строки в таблице  
процессов  
таблица процессов - главная таблица в системе

таблица процессов - двухсвязные списки  
в начальный момент в памяти выделяется минимально необходимое количество страниц:  
страница кода, страница данных, страница стека  
когда процесс получает процессоров время, он начинает выполнять  
есть возможность, что в какой-то момент процесс обратиться к странице отсутствующий  
памяти

существует несколько алгоритмов замещения алгоритмов страниц:

1. выталкивание случайной страницы  
не требует никакой дополнительной информации  
недостатки: выброс страницы которая используется, которая недавно загружена

#### **2. FIFO**

требует информацию о времени нахождения страницы в памяти  
выталкивается страница которая дольше всего находилась в памяти  
реализация: связный список, временные метки (когда страница загружается в память, она  
получает временную метку, выгружается страница у которой временная метка самая  
маленькая) (время идет только вперед)  
недостаток: мб выгружена интенсивно используемая страница  
обладает свойством: аномалия FIFO

23.10.18

сначала рассматриваем память размером 3 страницы  
номер процесса - номер страницы которые были загружены в память

'+' - страничное прерывание (загружается в результате страничного прерывания)

надо загрузить страницу которая в памяти (страничная удача), так и остается в очереди на старом месте, очередь не редактируется

|            | 1 | 2   | 3   | 4   | 5   | 6  | 7 | 8   | 9   | 10 | 11 | 12 |
|------------|---|-----|-----|-----|-----|----|---|-----|-----|----|----|----|
|            | 4 | 13  | 12  | 11  | 4   | 13 | 5 | 14  | 13  | 12 | 11 | 5  |
| $\uparrow$ | 4 | 3   | 2   | 1   | 4   | 3  | 3 | 3   | 5   | 2  | 2  |    |
| $M=3$      |   | (4) | (3) | (2) | (1) | 4  | 4 | (4) | (3) | 5  | 5  |    |
|            | + | +   | +   | +   | +   | +  | + | +   | +   | +  | +  |    |

$9/12 = 75\%$

увеличим страничную память

|              | 1 | 2  | 3  | 4  | 5 | 6  | 7 | 8  | 9  | 10 | 11 | 12 |
|--------------|---|----|----|----|---|----|---|----|----|----|----|----|
|              | 4 | 13 | 12 | 11 | 4 | 13 | 5 | 14 | 13 | 12 | 11 | 5  |
| $\uparrow$   | 4 | 3  | 2  | 2  | 2 | 1  | 5 | 4  | 3  | 2  | 1  |    |
| $M=4$        |   | 4  | 3  | 3  | 3 | 2  | 1 | 5  | 4  | 3  | 2  |    |
| $\downarrow$ | + | +  | +  | +  | + | +  | + | +  | +  | +  | +  |    |

$10/12 \approx 83\%$

при увеличении страничной памяти - число страничных прерываний увеличилось - аномалия FIFO

### 3. LRU - Least Recently used Page Replacement

для реализации алгоритма используются временные метро или связный список, но временные метки редактируются при каждом обращении к странице, или связный список редактируется при каждом обращении к странице, т.е. при каждом обращении к странице ... перемещается в конец

увеличим объем памяти на 1 страницу

|     | 1  | 2       | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 |
|-----|----|---------|----|----|----|----|----|----|----|----|----|----|
|     | 4  | 13      | 12 | 11 | 14 | 13 | 15 | 14 | 13 | 12 | 11 | 15 |
| ↑   | 9+ | 3+2+1+4 | 3  | 5+ | 4  | 3  | 2+ | 1+ | 5+ |    |    |    |
| M=4 | 4  | 3       | 2  | 1  | 4  | 3  | 5  | 4  | 3  | 2  | 1  |    |
| ↓   | 4  | 3       | 2  | 1  | 4  | 3  | 5  | 4  | 3  | 2  |    |    |
|     | 4  | 3       | ②  | 1  | 1  | ①  | ⑤  | ④  | 3  |    |    |    |
|     | +  | +       | +  | +  |    | +  |    | +  | +  | +  |    |    |

алгоритм построен на эвристическом правиле, которое утверждает:  
если к некоторой странице было обращение, то с большей вероятностью к этой же  
странице будут следующие обращения  
обратное тоже верно

алгоритм относится к стековым алгоритмам

если сравнить любой столбец (например 7й), то мы увидим что первые 3 эл-та построят распределение с тремя страницами, т.е. 1е 3 строки 2й таблицы полностью соответствует первым 3м строкам 1й таблицы

стековый алгоритм обладает свойством включения:

если какая-то страница была выбрана при реализации  $L(P, M, T)$ , где  $P$  - траектория  $(4, 3, 2, 1, 4, \dots)$ ,  $M$  - объем памяти,  $T$  - момент времени, то эта же страница будет выбрана в реализации  $L(P, M+1, T)$

этот алгоритм крайне затратный, требуется или постоянное редактирование временных меток или редактирование связного списка - это очень затратные действия в системе поэтому в чистом виде алгоритм не используется используется аппроксимация этого алгоритма

#### 4. LFU - Least... (наименее часто используемая страница)

в этом алгоритме контролируется частота обращения к странице (количество обращений) наименее интенсивно используемой может оказаться только что загруженная страница (еще не успела набрать обращения)  
такие страницы мб вытеснены сразу после того как они загружены в память

#### 5. NUR - Not used Recently Page Replacement (страница не используемая в последнее время)

алгоритм аппроксимирует алгоритм LRU

каждой физической странице приписывается бит обращения (Accessed) периодически все биты обращения сбрасываются в 0 , при обращении к странице бит обращения устанавливается в 1

если нужно вытеснить страницу то выбирается страница у которого бит обращения = 0 (с момента последнего сброса к этой странице обращения не было)

Diagram illustrating the state of memory after loading page 5 into frame 1. The diagram shows a 4x2 grid representing memory frames. The columns are labeled 'страница' (page) and the rows are labeled 'кадр' (frame). The grid contains the following values:

|   |   |
|---|---|
| 4 | 0 |
| 5 | 1 |
| 1 | 1 |
| 2 | 0 |

Annotations in Russian:

- 'физ. стр (кадр) (frame)' points to the column header.
- 'страница' points to the row header.
- 'бит обращения' (bit of access) points to the value '1' in the cell (5, 1).
- 'указатель удаления' (deletion pointer) points to a small box containing '1'.
- A large arrow points from the text 'показывает состояние оперативной памяти после загрузки 5й страницы в 1й кадр' to the grid.

The text on the right side of the diagram states: 'показывает состояние оперативной памяти после загрузки 5й страницы в 1й кадр'

бит обращения устанавливается на 0, если нужно будет вытеснить страницу, будет вытеснена 2я страница Зго кадра

кроме бита обращения вводится бит модификации (dirty)  
модифицированная страница (dirty page)

в результате  
возможны следующие 4 ситуации

| A  |   | D |
|----|---|---|
| 0. | . | 0 |
| 0. |   | 0 |
| 1  | . | 1 |
| 1. | . | 1 |

(2 строка) страница была модифицированная до последнего сброса обращения в 0

зачем ввели флаг dirty  
потому что выгоднее выгрузить не модифицированную потому что ее копия лежит на диске

принятый размер страницы 4 кб

## РАЗМЕР СТРАНИЦ, ФАКТОРЫ ВЛИЯЮЩИЕ НА ЕГО ВЫБР

с одной стороны желательно иметь страницы меньшего размера, потому что никогда на странице не выполняются все строки (обычно около 30%)

т.е. имеется вполне определенное соображение обосновывающее уменьшение размера страниц

с другой стороны чем меньше размер страницы, тем больше размер таблиц страниц

если взять 4кб страницу и 2кб страницу, то получится что размер таблиц страниц для 2кб страниц общий будет в 2 раза больше

при одном и том же размере адресного пространства изменение размера с 4кб на 2 кб приведет к увеличению числа дескрипторов в 2 раза

т.е. увеличится количество таблиц страниц описывающих одно и тоже адресное пространство

Медником в 1973г было показано, что алгоритм LRU для определенных траекторий страниц может дать для памяти 1мб и размере страниц 4 кб, если уменьшить размер страницы в 2 раза, то это при ведет к 257 кратному увеличению страничных прерываний это показано на простой модели:

Diagram illustrating memory organization for  $M=2$  pages. The top row shows page numbers 1, 2, 3, 4, 5, 6. Below is a grid where each row represents a page and each column represents a frame. The grid is divided into two halves by a vertical line. The left half contains addresses 1+, 1+, 2+, 2+, 3+, 3+. The right half contains addresses 1, 1, 2, 2, 3, 3. A circled '1' is highlighted in the second row, second column.

|   | 1  | 2  | 3  | 4  | 5  | 6  |
|---|----|----|----|----|----|----|
| 1 | 1+ | 1+ | 2+ | 2+ | 3+ | 3+ |
| 2 | 1  | 1  | 2  | 2  | 3  | 3  |
| 3 | +  | +  | +  | +  | +  | +  |

уменьшаем размер страницы в 2 раза  
страницы делятся пополам и мы обозначим их ` и ``

Diagram illustrating memory organization for  $M=4$  pages. The top row shows page numbers 1, 2, 3, 4, 5, 6. Below is a grid where each row represents a page and each column represents a frame. The grid is divided into four quadrants by a vertical and horizontal line. The top-left quadrant contains addresses 1', 1'', 2', 2'', 3', 3''. The top-right quadrant contains addresses 1+, 1+, 2+, 2+, 3+, 3+. The bottom-left quadrant contains addresses 1'', 1'', 2'', 2'', 3'', 3''. The bottom-right quadrant contains addresses 1, 1, 2, 2, 3, 3. A circled '1'' is highlighted in the second row, second column.

|   | 1   | 2   | 3   | 4   | 5   | 6   |
|---|-----|-----|-----|-----|-----|-----|
| 1 | 1'  | 1'' | 2'  | 2'' | 3'  | 3'' |
| 2 | 1+  | 1+  | 2+  | 2+  | 3+  | 3+  |
| 3 | 1'' | 1'' | 2'' | 2'' | 3'' | 3'' |
| 4 | 1   | 1   | 2   | 2   | 3   | 3   |
| 5 | +   | +   | +   | +   | +   | +   |

количество страничных прерываний в данном примере удвоилось

выход из ситуации для алгоритма ару  
половинки страниц загружаются по отдельности но при обращении к половинке она выгружается в начало списка

## ТЕОРИЯ РАБОЧЕГО МНОЖЕСТВА

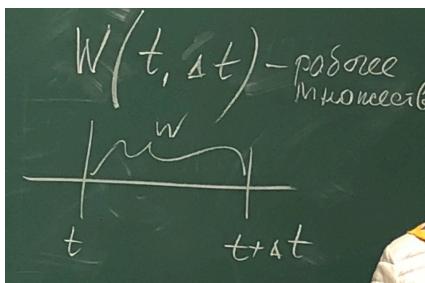
эффективное управление памяти связано с уменьшение числа страничных прерываний  
страничные прерывания связаны с блокировкой процесса (если страницы нет в памяти  
загрузка страницы потребует ряд сложных действий - такой процесс будет блокирован)

менеджер памяти , в его задачи входят определение какому процессу какую страницу  
выделить

страницы которые нужно загрузить находятся во вторичной памяти  
в процессе загрузки новых страниц будет происходить обращение к жесткому диску,  
фактически задействуется подсистема ввода-вывода  
не смотря на то что подкачка страниц всегда оптимизирована это все равно крайне  
затратное действие

возможна ситуация когда возникает слишком большое число страничных прерываний  
такие ситуации крайне отрицательно сказываются на производительности системы  
необходимо исключить возникновение большого числа страничных прерываний

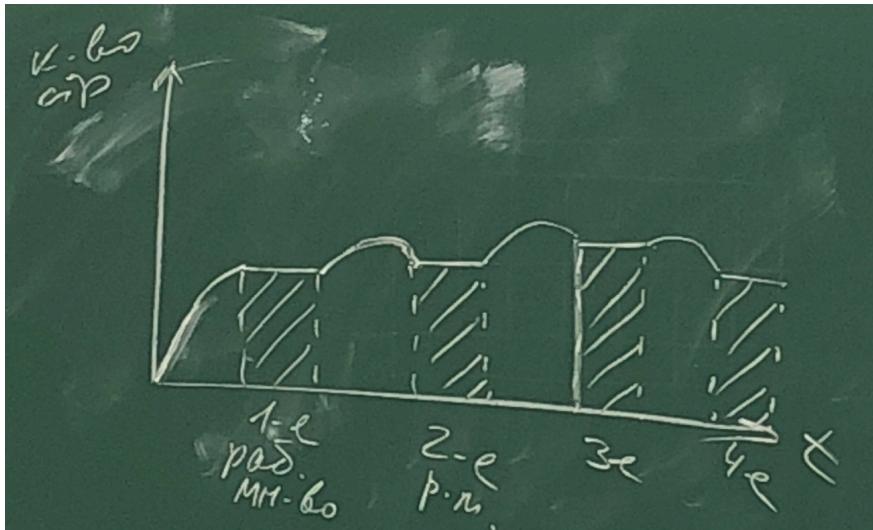
1968г деннинг предложил в качестве локальной меры производительности взять число  
страниц к которым программа обращается за интервал времени  $\delta$  дельта t



было показано, что W является монотонной функцией от дельта t  
и множество страниц к которым процессор обращается за дельта t называли рабочим  
множеством

при увеличении дельта t число страниц будет стремиться к некоторой величине L  
именно эта величина определяет количество страниц которое необходимо процессу для  
эффективного выполнения, т.е. без страничных прерываний  
т.е. если процессу удается загрузить в память свое рабочее множество то некоторое  
время процесс выполняется без страничных прерываний  
если процессу не удается загрузить в память в свое рабочее множество (у сис-мы  
недостаточно памяти), то возникнет интенсивный част числа страничных прерываний, Так  
как процесс будет подгружать одни и те же страницы  
это явление получило название трешинг (thrashing)

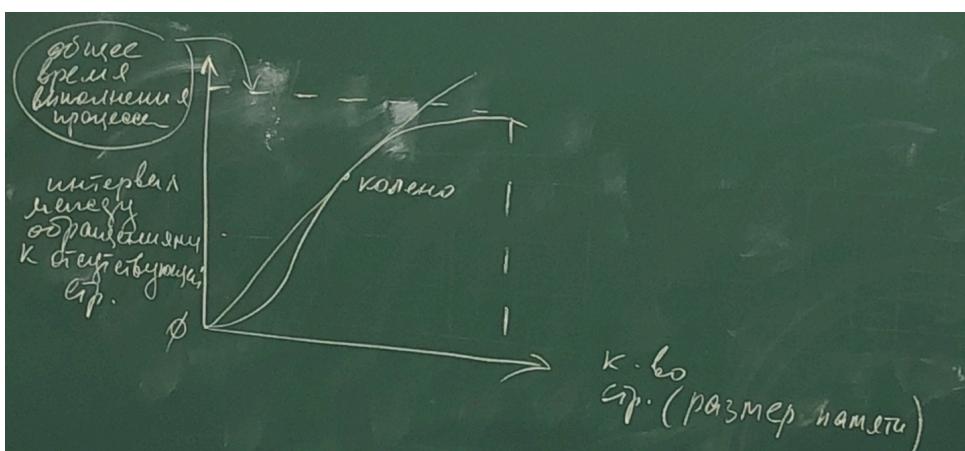
это можно представить в виде иллюстрации



в начальный момент времени, когда процесс создан ему выделяется минимально необходимое количество страниц  
после этого процесс переходит в состояние готовности и начинает интенсивно подкачивать нужные ему страницы  
выполнение процесса постоянно будет прерываться соответствующими блокировками это будет продолжаться пока процесс не загрузить свое рабочее множество, некоторое время не будет прерываний  
затем процесс перейдет к следующему множеству до тех пор пока в памяти не останется его 2е рабочее множество и т д

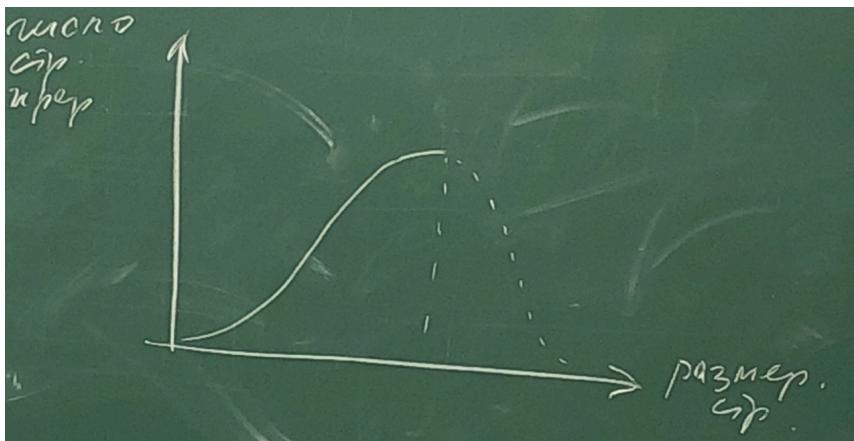
горбик связан с тем  
что при переходе с одно на другое остаются страницы старого рабочего множества и нового рабочего множества

интервал обращения к отсутствующей странице называется отсутствующей страницей график отражает зависимость длительности периода между прерываниями от размера памяти



перегиб называют коленом  
он происходит из-за того что в некоторый момент времени в оперативной памяти оказываются все страницы

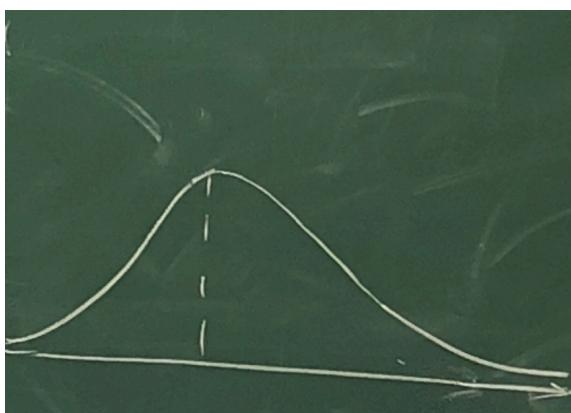
следующий график отражает влияние размера страницы при одном и том же объеме оперативной памяти на число прерываний



с ростом размера страницы число страничных прерываний возрастает  
это связано с тем, что чем больше страница, тем больше в память попадает функций, данных, к которому не будет обращений  
тем самым уменьшается доля которая занимается кодом к которому выполняются обращения поэтому число страничных прерываний растет

если размер страницы сопоставим с размером программы то страничные прерывания не возникают (пунктир)

график показывает процент команд на странице которой будет выполнен до передачи управления другой странице  
для страницы 1024 слова - 4096 байт - 4кб



на страницу 1024 слова приходится 200 команд

для сегментированной памяти все рассуждения сохраняются, но при вытеснении сегментов возникают проблемы  
мб вытеснен сегмент размер которого недостаточен чтобы подгрузить сегмент необходимого размера  
тогда потребуется вытеснить еще один сегмент

30.10.18

локальное и глобальное замещение страниц

локальное замещение подразумевает что вытеснение осуществляется только среди множества страниц которые были загружены этим процессов

эффективность выполнения действий в системе

когда процесс обратился к странице отсутствующей в памяти эту страницу нужно загрузить как можно быстрее

если будет выполняться замещение, то никакой быстроты нет

в системах вытеснение страниц выполняется заранее

в UNIX

процессы демоны - процессы которые не имеют родителей, они существуют сами и не входят ни в какие группы

процессы которые большую часть своей жизни спят и возникают только когда в них есть необходимость (ПОЧИТАТЬ ПРО ЭТО)

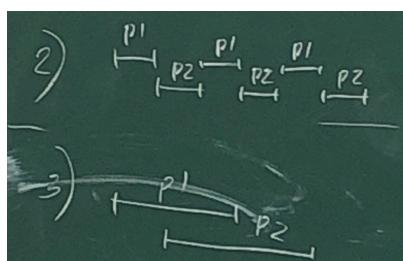
## ВЗАИМОДЕЙСТВИЕ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ

1) квази-параллельное выполнение

эти процессы существуют в системе параллельно

2) реально-параллельное выполнение

реально-параллельно выполняются программы



проблемы в этих 2х случаях одинаковы

The diagram shows two parallel processes, p1 and p2, updating a shared variable S. Process p1 is shown with the update  $S = S + \delta_1$ . Process p2 is shown with the update  $S = S + \delta_2$ . The final value of S is the sum of the updates:  $S = S + \delta_1 + \delta_2$ .

для многопроцессорной системы:

p1: ....  
mov eax, myvar  
inc eax  
mov myvar, eax

p2:.....  
mov eax, myvar  
inc eax  
mov myvar, eax

пусть в начале момент времени  $myvar=0$   
 процессоров всегда меньше чем процессов  
 всегда процессоров время квантуется

| P1 на 1-м процессоре | myvar | eax | P2 на 2-м процессоре |
|----------------------|-------|-----|----------------------|
| mov eax, myvar       | 0     | 0   |                      |
|                      | 0     | 0   | mov eax, myvar       |
|                      | 0     | 1   | inc eax              |
| inc eax              | 1     | 1   | mov myvar, eax       |
| mov myvar, eax       | 1     | 1   |                      |

потеряли значение eax в p2

для однопроцессорной системы:

| P1                             | myvar | P2                   |
|--------------------------------|-------|----------------------|
|                                | 0     |                      |
| блокируется                    |       | t>1                  |
| mov eax, myvar                 | 0     | 0                    |
| P1 потерян контекст, ван-ел P2 |       |                      |
| контекст P1 сохранен           | 0     | mov eax, myvar       |
|                                | 1     | inc eax              |
|                                | 1     | mov myvar, eax       |
| Р2 теряет контекст             | 1     | блок.ел P1           |
| inc eax                        | 1     | контекст Р2 сохранен |
| mov myvar, eax                 | 1     |                      |

Так как произошло переключение на процесс p1, восстанавливается его контекст, т.е. содержимое регистра eax  
 соответственно p1 выполняется

мы опять потеряли единицу  
 такая ситуация недопустима

часть кода в которой производится доступ параллельных процессов разделяемых переменной называется критической или критической секции  
 $myvar$  - разделяемая переменная (разделяемый ресурс)

чтобы избежать подобных ситуаций необходимо обеспечить монопольный доступ к разделяемой переменной к параллельно выполняемым процессом (потокам)

монопольный доступ обеспечивается методами взаимоисключения  
 mutual execution

доступ параллельных процессов разделяемых переменной часто называются гонками (race condition)

существует несколько основных методов реализации взаимоисключений:

1. программный
2. аппаратный
3. с помощью семафоров
4. с использованием мониторов

эти методы являются методами реализованными для отдельно стоящей машины

решение рассмотренной проблемы  
на псевдокоде

example1:

```
f1, f2:logical;
p1: while(1) do
begin
    while(f2) do;
        f1 = 1;
        cr1; // критическая секция
        f1 = 0;
        pk1;
    end;
end; // p1
```

```
p2: while(1) do
begin
    while(f1) do;
        f2=1;
        cr2;
        f2=0;
        pk2;
    end; // while
end; // p2
```

parbegin

```
f1=0;
f2=0;
parend;
```

while(f2) do; - на проверку тратится весь квант - активное ожидание на процессоре  
данное решение не надежно

example2:

```
f1, f2:logical;
p1: while(1) do
begin
    f1 = 1;
    while(f2) do;
        cr1; // критическая секция
        f1 = 0;
        pk1;
    end;
end; // p1
```

```
p2: while(1) do
begin
    f2 = 1;
    while(f1) do;
        cr2;
        f2=0;
        pk2;
    end; // while
end; // p2
```

parbegin

```
f1=0;
f2=0;
parend;
```

пример классического тупика  
каждый из процессов ждет пока другой процесс освободит флаг  
оба не могут это сделать и продолжить свое выполнение

данную задачу решил математик Декер для двухпараллельных процессов  
 алгоритм декера решает данную задачу взаимоисключений  
 вводит переменную...  
 алгоритм решает проблему бесконечного откладывания

если инициативу по входу в критический участок все время перехватывает один процесс то  
 второй процесс никак не может войти в свой критический участок - бесконечное  
 откладывание

dekker:

p1:...

```

        while(1) do
        begin
            f1=1;
            while(f2) do
                if (queue == 2) then
                begin
                    f1=0;
                    while(queue==2) do;
                        f1=1;
                end;
            cr1;
            f1=0;
            queue=1;
            pr1;
        end;
    end; // p1

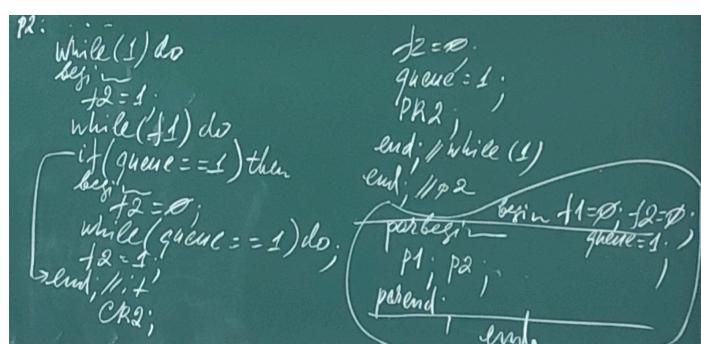
```

p2:...

```

        while(1) do
        begin
            f2=1;
            while(f1) do
                if (queue == 1) then
                begin
                    f2=0;
                    while(queue==1) do;
                        f2=1;
                end;
            cr2;
            f2=0;
            queue=1;
            pr2;
        end;
    end; // p2

```



## элегантное решение петерсона

|  |  |  |
|--|--|--|
| <p>Peterson:</p> <pre> f1, f2 : logical; queue: int.  P1: while (1) do begin     f1 = 1;     queue = 2;     while (f2 &amp;&amp; (queue == 1)) do; //недел     CR1;     f1 = 0; end; end(); </pre> | <p>P2: while (1) do begin     f2 = 1;     queue = 1;     while (f1 &amp;&amp; (queue == 1)) do; //недел     CR2;     f2 = 0;     PR2; end; /</p> | <p><u>begin</u></p> <p><u>f1=0; f2=0;</u></p> <p><u>parbegin</u> /</p> <p><u>p1; p2;</u></p> <p><u>perend</u> /</p> <p><u>end.</u></p> |
|--|--|--|

выполнение условия взаимоисключения легко показать  
после установки флага f1 процесс p1 зайти не может  
если процесс p2 уже находится в своем критическом участке то  $f2 = 1$   
и процесс ... в критический участок закрыт

возможные случаи:

1. процесс p2 не хочет входить в свой критический участок - случай невозможен, Так как при этом выполнялось бы условия  $a2=0$
2. p2 постоянно(длительное время) ожидает вход в критический участок - случай невозможен, Так как если queue =2, то p2 может войти в свой критический участок
3. p2 монополизировал вход в критический участок, переодически входя и выходя из него - случай невозможен, Так как p2 перед попыткой входа в свой критический участок должен установить queue=1 давая возможность процессу p1 войти в свой критический участок

все рассмотренные нами решения имеют один существенный недостаток: активное ожидание на процессоре, т.е. процессоров время тратится непроизводительно на проверку значений переменных и флагов  
но интерес к программному решению задачи взаимоисключения имеет место быть

новое решение проблемы параллельно программирования дейкстры  
рассмотрим алгоритм булочная

### 6.11.18

алгоритм лампорта решает проблему параллельного программирования  
каждому приходящему клиенту выдается листок с номером  
при том каждому новому клиенту выдается на единицу больший номер  
когда продавец освобождается начинает обслуживаться клиент с номером меньшим  
если возникает ситуация когда 2 клиента входят одновременно  
им выдаются одинаковые номера, но при обслуживании выбирается клиент у которого  
меньший номер паспорта (меньше идентификатор)

```
typedef cahr boolean;  
...  
shared boolean choosing[n];  
shared int num[n];  
...  
for(j = 0; i < n; j++)  
    num[j] = 0;  
...
```

Entry Protocol(для i-го процесса)  
/\* choose a number \*/  
choosing[i] = true;  
num[i] = max(num[0], ..., num[n-1]) + 1;  
choosing[i] = false;

здесь iй процесс выбирает номер  
он получает номер на 1 больше последнего выданного номера

```
/* для всех других процессов */  
for(j = 0; j < n; j++) {  
    /* ожидает, если процесс выбран */  
    while(choosing[j])  
        /* nothing */  
    /* ожидает, если процесс имеет номер и выполняется до */  
    if ((num[j] > 0) && ((num[j] < num[i]) || (num[j] == num[i] && (j < i))))  
    {  
        while (num[j] > 0) /* nothing */  
    } // if  
} // for  
/* critical section */
```

осуществляется выбор процесса который входит в критическую секцию  
процесс ждет пока имеется процесс с меньшим номером  
если 2 процесса имеют одинаковые номера  
то выбор процесса осуществляется по его идентификатору

такая запись  $(a, b) < (c, d)$  - лексикографический порядок  
выражение будет истинным, если  $a < c$  или  $a = c$ , но  $b < d$

следует отметить, что это корректное решение задачи  
но как для любого программного решения для него характерно активное ожидание на  
процессоре

Аппаратная реализация:

команда test-and-set

это неделимая (отомарная) инструкция которая копирует значение переменной в регистр и устанавливает какое-то новое значение

в многопроцессорной архитектуре

если какое-то процесс выполняет действие с соответствующей ячейкой памяти, другие процессоры не могут получить к ней доступ  
благодаря кратковременной блокировки шины памяти

например отомарная команда test - and - set читает значение логической переменное в копирует его в переменную a

example test-and-set

    fl, c1, c2 : logic;

p1:   while (1) do

    begin

        c1 = 1;

        while (c1 == 1) do

            begin

                test-and-set(c1, fl);

            end;

        CR1;

        fl = 0;

        PR1;

        end;

    end; // p1

p2:   while (1) do

    begin

        c2 = 1;

        while (c2 == 1) do

            begin

                test-and-set(c2, fl);

            end;

        CR2;

        fl = 0;

        PR2;

        end;

    end; // p2

begin

    fl = 0;

    parbegin

        p1; p2;

    parend;

end;

логическая переменная fl = 1 если любой из процессов находится в своем критическом участке, в противном случае = 0

рассмотрим ситуацию когда процесс p1 хочет зайти в свой критический участок  
процесс p2 уже находится в своем критическом участке

p1 устанавливает c1=1 и входит в цикл проверки переменной fl командой test-and-set  
тк p2 уже в своем критическом участке => fl = 1

команда test-and-set находит этот факт и устанавливает c..=1

в результате p1 находится в своем цикле активного ожидания до тех пор  
пока p2 не выйдет из своего критического участка и не установит fl = 0

данная реализация называется аппаратной

это название подразумевает, что используется машинная команда

например команда compare-and-swap(int \*reg, int oldval, int newval);  
для этого решения характерно активное ожидание на процессоре  
значение логической переменной проверяется процессом в цикле  
но считается, что данная реализация исключает бесконечное откладывание (его  
вероятность очень велика)  
когда процесс выходит из своего критического участка и сбрасывает флаг (fl = 0)  
скорее всего другой процесс сможет перехватить инициативу и установить логическую  
переменную в true

команда test-and-set - машинная команда, но в системе данная команда используется в  
спин блокировках (spin\_lock)  
спин-блокировки очень широко используются в ядре систем  
иногда их называют simple\_lock (простой блокировкой) или simple\_mutix (...)

чаще всего команда возвращает предыдущее значение объекта

```
void spin_lock(spin_lock_t *c) // c - condition - переменная типа условие
{
    while(test-and-set(c) != 0) /*ресурс занят*/
}
void spin_unlock(spin_lock_t *c) {
    c = 0;
}
```

команда test-and-set из-за связи с блокировкой шины памяти  
длительный цикл может привести к занятию шины памяти одним процессом, что  
существенно может снизить производительность системы  
блокировка шины памяти - на одном крутиться spin\_lock, а остальные процессы к памяти  
обратиться не могут

данная проблема решается с помощью 2х вложенных циклов

```
void spin_lovk(spin_lock_t *c) {
    while(test-and-set(c) != 0) // команда устроена таким образом, что шина данных будет
    //блокирована, а внутренний цикл выполняется путем проверки переменной c, без захвата
    //шины данных
        while(*c != 0);
}
```

смысл второго варианта:

если переменная занята, то выполняется внутренний цикл, в котором проверка переменной  
с выполняется без захвата шины данных

команда spin\_lock - называется макрокомандой

## СЕМАФОРЫ

Dijkstra E. W. в 1965 году опубликовал работу, в которой предложил новое решение взаимоисключения с помощью семафоров

семафор(s) - неотрицательная защищенная переменная, на которой определены 2 операции:

s      P(s) и V(s)

P - passeren(пропустить)

V - vrygeven (освободить)

ближе всего семафор к общественному туалету - кабинке  
процесс может пройти, если другой процесс вышел

если семафор может принимать только 0 и 1 - бинарный

если может принимать неотрицательные целые значения - считающий

P(s) и V(s)- неделимые - их выполнение не может быть прервано

переменная s называется защищенной, потому что доступ к ней имеют только 2 команды: P и V

операция V(s):  $s = s + 1$ , выполняется как одно неделимое действие

если  $s = 0$ , то операция V(s) может активизировать некоторый процесс блокированный на семафоре

операция P(s):  $s = s - 1$ , процесс пытающийся войти в критическую секцию пытается декрементировать семафор

если  $s = 0$ , то декремент невозможен и процесс переводится в состояние ожидания (блокировки) до тех пор, пока другого процесса не освободит ресурс (не выйдет из своего критического участка)

блокировать процесс и разблокировать процесс может только ядро системы  
как мы видим семафоры исключают бесконечное ожидание на процессоре, но платой за это является переход в режим ядра  
т.е. команды определенный на семафоре являются системными вызовами  
зато исключается активное ожидание на процессоре

запишем простейшее взаимоисключение в помощь семафоров:

в начале работы  $s = 1$ , иначе никто войти не сможет

|           |  |           |
|-----------|--|-----------|
| p1: ..... |  | p2: ..... |
| P(s);     |  | P(s);     |
| CR1;      |  | CR2;      |
| V(s);     |  | V(s);     |
| PR1;      |  | PR2;      |
| .....     |  | .....     |

ЭТО ЗНАТЬ ОБЯЗАТЕЛЬНО

рассмотрим задачу производства потребления (решение Дейкстры на 3х семафорах)  
в простейшем варианте

2 параллельных процессов: процесс, который производит единицу продукции и кладет ее в буфер

и процесс который берет единицу продукции произведенную другим процессом



se - определяет число пустых ячеек

sf - считает число заполненных ячеек буфера

sb (s) - бинарный семафор

```

example cons-prod;
    n : int
    se, sf, sb : int;
producer:   while (1) do
begin
    ;создает единицу продукции
    P(se);
    P(sb);
    h = h+1; // добавить в буфер
    V(sb);
    V(sf);
ens;

consumer:  while (1) do
begin
    ;создает единицу продукции
    P(sf);
    P(sb);
    n = n+1; // взять
    V(sb);
    V(se);
end;

begin
    se = N;
    sf = 0';
    se = 1;
    .....

```

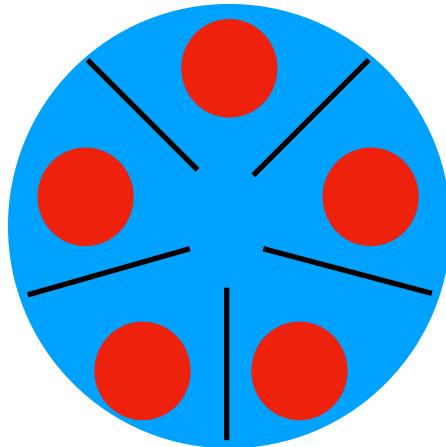
продюсер будет блокирован на семафоре se, когда se=0 (нет пустых ячеек)  
если есть пустые ячейки, то продюсер декрементирует se и проверяет sb,  
если consumer ничего из буфера не вытаскивает, то он спокойно входит в критический  
участок, создает единицу продукции и инкриминирует кол-во заполненных ячеек  
consumer будет блокировать на семафоре se, если se = 0  
если sf > 0, то consumer декрементирует кол-во заложенных ячеек  
если путь свободен, то входит в свой критический участок, перед единицей продукции,  
выходит из критического участка и инкрементирует.....

sb обеспечивает монопольный доступ к разделяемой переменной

## МНОЖЕСТВЕННЫЕ СЕМАФОРЫ

в современных системах принято использовать множественные семафоры называют массивы считающих семафоров

задача обедающие философы (задача о ресурсах(вилки это разделяемый, дефицитный ресурс(палки), круги - это философы)):



рассматривается 3 способа действия

1. каждый из философов пытается одновременно взять вилки, если ему это удается он начинает есть, через некоторое время он кладет обе вилки (в каждый момент времени мы видим, что взять обе вилки смогут только 2 философа)
2. каждый из философов сначала берет правую вилку, затем пытается взять левую, если ему не удается взять левую, то он удерживает правую
3. каждый из философов берет правую вилку, пытается взять левую, если левую взять не может, то он кладет правую

первый способ приведет к бесконечному откладыванию

второй способ - классический тупик, если они все одновременно возьмут правую вилку, ни один из них взять левую не сможет

третий способ, захват и освобождение одних и тех же ресурсов

```
var
    forks: array[1..5] of semaphore;
    i: int;
begin
    i:=5;
repeat
    forks[i]:=1;
    i:=i+1;
until i=0;

cobegin
1: begin
    var left, right : 1..5;
    left:=1, right:=2;
    ...
end;
...
5: begin
    var
        left:=5;
        right:=1;
```

```
repeat
// размышляет
P(forks[left], forks[right]);
// ест
V(forks[left], forks[right]);
forever;
end;
end;
cobegin
end.
```

у множественных семафоров есть очень важное свойство:  
одной неделимой операцией можно выполнит действие на всех или части семафоров набора  
в этом примере мы видим что одной неделимой операцией проверяется или изменяется значение сразу двух семафоров  
при этом, при выполнении P операции на семафорах, процесс может заснуть либо на одном семафоре, либо на другом семафоре, либо на обоих семафорах сразу очевидно, что механизм реализации множественных семафорах значительно сложнее, чем механизм реализации простых семафоров  
Так как в одно примитиве необходимо проверить не одну, а несколько очередей с учетом различных состояний семафоров  
если процессы не пересекаются по требуемым им ресурсам, то они могут выполняться параллельно

семафоры не единственные средства взаимоисключения в системах  
в системах встречаются самые разные примитивы, которые позволяют реализовывать взаимоисключения

## Mutex

отличие между mutex и семафорами:

иногда мьютекс рассматривается как бинарным семафором, но есть отличия  
мьютексы имеют конкретного владельца, т.е. процесс который захватил мьютекс только процесс который захватил, может освободить

в отличие от мьютексов семафор не имеет владельца, для него концепция владельца не существует  
любой процесс может разблокировать семафор (освободить) (даже если этот процесс не блокировал семафор) (он может захватить один процесс, а освободить другой)

кроме того мьютексы осуществляют надежную блокировку  
если процесс удерживает мьютекс, он не может быть удален (к семафора это не относится)

если говорить в самом общем случае, системы предоставляют примитивы взаимоисключения, например:  
наряду с мьютексами могут использоваться такие системные вызовы, как:  
wait()  
signal()  
post()  
log()  
analog()

все перечисленные семафоры их действия относятся к действиям которые называются захват и освобождения (оповещение)

все эти примитивы являются системными вызовами, т.е. при обращении к ним процесс переводится в режим ядра, в режиме ядра блокируется, и примитивы которые предназначены для разблокирования выполняются в режиме ядра

к переменным conditional в системе выстраиваются очереди ..... что-то там блокируют  
процесс захвативший семафор может быть убит (уничтожен)

## МОНИТОРЫ

мониторы могут быть реализованы на уровне языка программирования, могут предоставляться системой

товарищ Hoare

монитор это средство организации взаимодействия и взаимоисключений параллельный и асинхронных процессов, содержащее как данные, так и подпрограммы которые обрабатывают эти данные

задача мониторов структурировать программные средства предназначенные для решения проблемы взаимоисключения и взаимодействия процессов

монитор защищает свои данные, т.к. доступ к этим данным возможен только функциям монитора

особенно сложные задачи взаимоисключения стоят перед системными процессами, поэтому монитор это прежде всего средство структурирования ОС

основная задача ОС - динамическое распределение ресурсов параллельным процессам и их потокам  
распределение ресурсов связано с задачей планирования распределения этих ресурсов эти задачи в системе решаются с помощью планировщиков  
для каждого критического ресурса в системе должен быть свои планировщик, но все планировщики решают подобные задачи, а именно задачу распределения ресурсов поэтому все они могут быть построены по одной или нескольким схемам  
именно такую схему определяет монитор - общую схему взаимодействия

монитор самая является ресурсом  
в каждый момент времени только один процесс может с помощью функций монитора обращаться к его данным

в основу работы мониторов положены методы wait() и signal(), которые определены на переменной типа условие (conditional)

если процесс, которому необходим ресурс, обращается к монитору для получения этого ресурса, то функция которую процесс вызвал , если ресурс занят, выполнит системный вызов wait()  
в результате процесс будет переведен в состояние ожидания или блокировки по данному ресурсу

очевидно, что процесс не удерживает монитор  
он блокирован  
и монитор освобождает  
таким образом дает возможность другим процессом обратить к монитору

если процесс хочет освободить ресурс  
он также обращается к монитору  
и в этой функции будет выполнен сигнал (оповещение) с помощью которого процессы  
стоящие в очереди к данному ресурсу будут разблокированы (тот который стоит первым в  
очереди)

рассмотрим простой монитор  
этот монитор обеспечивает выделение единственного ресурса произвольному числу  
процессов

монитор: распределение;  
переменные

занят: логический;  
свободен: условие;

функция получать  
начало

если (занят) то  
    wait(свободен);  
    занят = истина;  
конец;

функция освободить  
начало

    занят = ложь;  
    signal(свободен);  
конец;

begin

    занят = ложь;

end.

тоже самое, но по нормальному

monitor distribution;

var

    busy: logical;

    free: conditional;

procedure receive(p: process)

    begin

        if(busy) then  
            wait(free);

    end;

procedure release;

    begin

        busy = false;  
        signal(free);

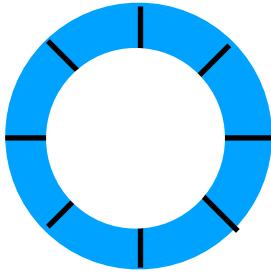
    end;

begin

    busy = false;

end.

монитор: кольцевой буфер (еще одна реализация задачи производство-потребление)



```
monitor circle_butter;
var
    bcircle: array[0..n-1] of type;
    pos: 0..n; // текущая позиция
    j: 0..n-1; // заполняемая позиция
    k: 0..n-1; // освобождаемая позиция
    butterfull, butter empty: conditional;

procedure producer(data: type)
begin
    if pos = n then
        wait(butterempty);
    bcircle[j] := data;
    pos:= pos+1;
    j := (j+1) mod n;
    signal(bufferfull);
end;
procedure consumer(var data:type)
begin
    if (pos = 0) then
        wait(bufferfull);
    data:= bcircle[k];
    pos:= pos-1;
    k:= (k+1) mod n;
    signal(bufferempty);
end;
begin
    pos:=0;
    j:=0;
    k:=0;
end.
```

механизм кольцевого буфера используется для работы с spooler (spool) (катушка)  
(simultaneous peripheral operations online)  
используется когда какой-то процесс формирует строки для вывода на принтер  
очевидно что процессы работает с разной скоростью и генерация строк происходит  
быстрее чем их печать  
и необходима буферизация данных

## 20.11.18

---

задача читателя-писателя

бытовым примером этой задачи являются системы продажи билетов на самолеты и поезда дальнего следования (существует строгое расписание и нумерация посадочных мест)

рассмотрим эту задачу

рассмотрим классический монитор Хоара с 4мя функциями (start\_read, stop\_read, start\_write, stop\_write)

им предложен монитор, в нем 2 переменные типа условия can\_write и can\_read и 4 вышеупомянутые функции

монитор имеет вид:

```
int readers = 0;
boolean write = false;
Conditional conwrite;
Conditional cornd;
monitorentry void startread()
{
    if (writelock || queue(canerite)) // 2е усл нужно для исключения бесконечного
                                      откладывания записи
    {
        wait(canread);
    }
    ++readers;
    signal(canread);
}
monitorentry void stopread()
{
    --readers;
    if (readers == 0)
    {
        signal(canwrite);
    }
}
monitorentry void startwrite()
{
    if ((readers > 0) || writelock)
    {
        wait(canwrite);
    }
    write lock = true; // захват критической секции
}
monitorentry void stopwrite()
{
    writelock = false;
    if (queue(canread)) // проверка очереди ждущих читателей
    {
        signal(canread); // если они есть, то можно читать
    }
    else
    {
        signal(canwrite); // можно писать
    }
}
```

каждая из этих функций является входом в монитор

если процесс нуждается в том, чтобы посмотреть какие-то данные, он вызывает функцию start\_read

при этом он может начать читать только если нет работающего писателя, или нет ждущих писателей(писателей находящихся в очереди к монитору)

если есть работающий писатель, то в каждый момент времени писать может только один писатель (изменять поле данных может только 1 писатель) (процесс будет блокирован на переменной условия canread)

если нет ни работающего ни ждущего, то увеличивается количество активных считателей и посыпается сигнал другим читателям, что можно читать canread

по завершении чтения, процесс вызывает функцию stopread, декрементируется кол-во активных читателей, и если оно становится равным 0, то посыпается сигнал, что можно писать

если процесс нуждается в изменении поля данных startwrite

в этой функции проверяется, есть ли активные читатели

в результате следующий читатель будет активизирован и в результате этот следующий читатель вызовет signal(canread) - реализована цепная реакция читателей, т.е. каждый рабочий читатель активизирует следующего читателя (читатели получают возможность параллельно читать интересующее их поле)

процессы получившие уведомление о том, что можно читать будут обслуживаться в в порядке очереди

когда все читатели закончились, вызывается системный сигнал, который инициализирует писателя

для обеспечения монопольного доступа писателя к разделяемой переменной, используется

когда писатель получает возможность работать писатель устанавливает writelock = true тем самым блокирует доступ читателя и писателя к разделяемой переменной

для того чтобы не возникало бесконечного откладывания процессов читателей

когда писатель завершает свою работу, он проверяет нет ли ждущего читателя

если есть то вызывается signal и активизируется читатели

если нет то ждущий писатель

## ПЕРЕДАЧА СООБЩЕНИЙ

наиболее универсальным способом взаимодействий процессов является передача сообщений  
send message и receive message

процессы которые выполняются в системе являются асинхронные процессы - каждый процесс выполняется с собственной скоростью (невозможно предсказать, когда какой процесс придет в свою конечную точку)

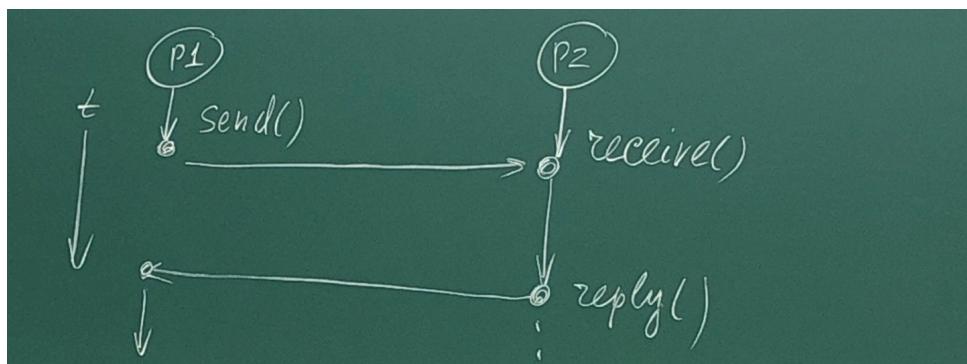
способы взаимодействий :

1. взаимоисключения (организация монопольного доступа процесса к разделяемой переменной) (чистое взаимоисключение реализуется в задаче читателя-писателя (осуществляется монопольный доступ писаля))
2. синхронизация (задача производства потребления) (если потребитель работает быстрее производителя возникнет ситуация когда буфер пуст, потребитель будет ожидать когда производитель положит что-нибудь в буфер) (когда процесс заинтересован в действиях другого процесса) (это видно при передаче сообщений, сообщения несут информацию которая интересует процесс, для того чтобы продолжить свое выполнение)

диаграмма последовательности событий:

рассмотрим процесс взаимодействия двух процессов p1 и p2 (2 асинхронных процесса - выполняются с собственной скоростью)

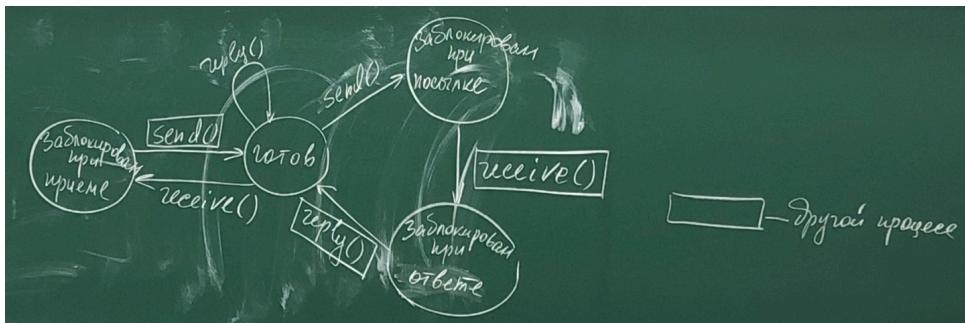
процесс p1 решает запросить что-то у процесса p2, для этого он выполняя системный вызов send(), процесс p2 должен принять сообщение системный вызовом receive() получив запрос p2 какое-то время обрабатывает полученные данные если p1 заинтересован в результате работы p2 он будет блокирован в ожидании очереди далее он может продолжать свою работу



часто такое взаимодействие выполняется по модели клиент-сервер  
задача сервера обслуживать запросы клиентов  
при таком взаимодействии возможны 3 состояния блокировки процесса при передаче  
сообщений

диаграмма 3 состояния блокировки процесса при передаче сообщения

действия выполняемые от имени другого процесса в рамочке



распределенные системы - системы не имеющие общей памяти (взаимодействие возможно только путем передачи сообщений)  
передача сообщений между различными узлами распределенной системы связанно со значительными временными затратами

### RPC вызов удаленной процедуры

это механизм, с помощью которого один процесс активизирует другой процесс, на этой же или удаленной машине, для выполнения какой-то функции от своего имени  
РПЦ напоминает вызов локальной функции, а именно процесс вызывает функцию и передает ей данные, а затем ожидает, когда она возвратит результат

специфика РПЦ заключается в том, что эту функцию выполняет другой процесс  
такое взаимодействие процессов выполняется по схеме(модели) клиент-сервер, в котором процесс активизирующий РПЦ является клиентским, а процесс выполняющий РПЦ серверным

серверный процесс обеспечивает доступ к одной или нескольким сервисным функциям, которые могут вызываться его клиентами

РПЦ используется в сетевых приложениях для подключения к сетевым ресурсам других машин  
например в распределенной БД, серверным является процесс - управление БД, обеспечивающий поиск и хранение данных в ее файлах

клиентские процессы - внешние программы БД, которые позволяют пользователям заправить и обновлять данные  
клиентские процессы преобразуют команды пользователей в РПЦ и направляют их обслуживающему процессу (серверу)  
значения которые возвращаются РПЦ функциями и являются теми данными, которые запрашивали клиенты

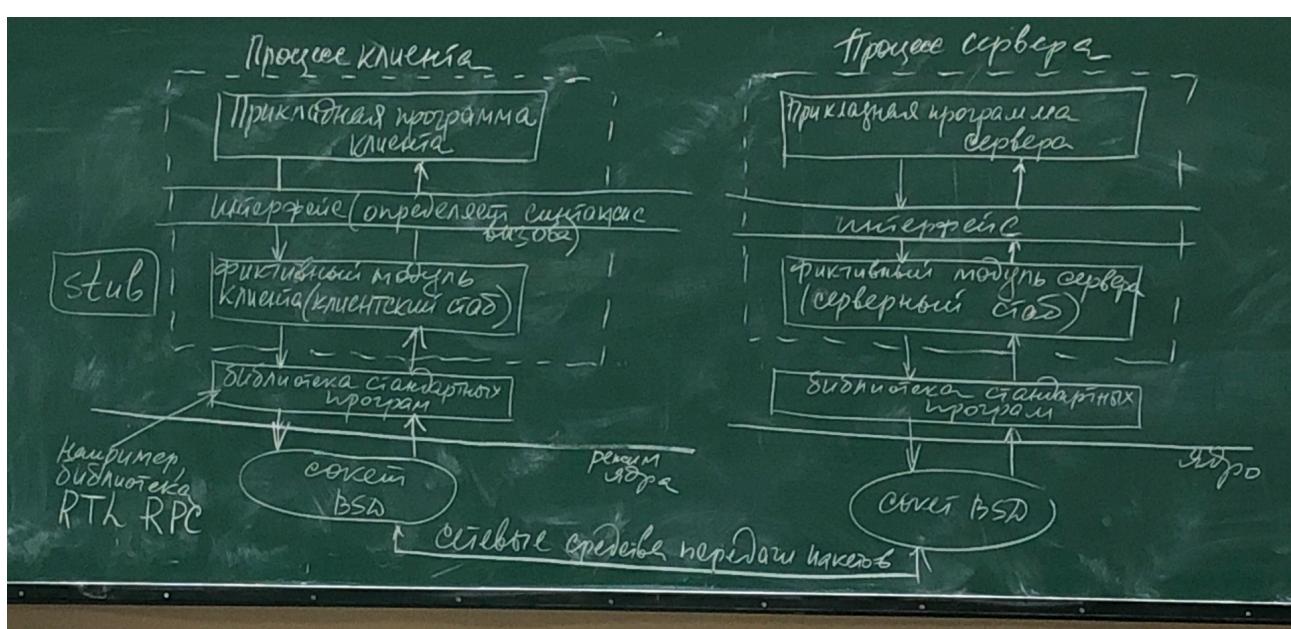
еще одним примером является спуллер печати windows (print spoller)  
 клиентская часть спуллера - winspool.drv - передает данные посредством РПЦ серверной части spoolsv.exe  
 серверная часть спуллера включает в себя: маршрутизатор печати spoolsv.dll, локальный провайдер печати localpl.dll, процессор печати, процессор разделительных страниц, монитор порта

разделительные страницы содержат: имя пользователя, имя компьютера, дату создания задания  
 локальный монитор порта управляет последовательными и параллельными портами, к которым подключены принтеры

пуллер - это программное обеспечение, которое последовательность заданий на печать хранит в буфере и отправляет каждое задание на принтер, когда принтер переходит в состояние готовности обработать такое задание

## ПРОЦЕСС КЛИЕНТА

RPC :



вызов прикладной программы приводит к вызову клиентской заглушки (стабу)  
 клиентский стаб выполняет ряд действий, затем выполняется системный вызов, который переводит в систему в режим ядра

по сети передается транспортными средствами  
 здесь пакет преобразуется средствами ядра  
 определяется какой фиктивный модуль (серверный стаб) ждет данный пакет и ему этот пакет пересыпается  
 пакет записывается в стек  
 обычным способом из стека достаются.....

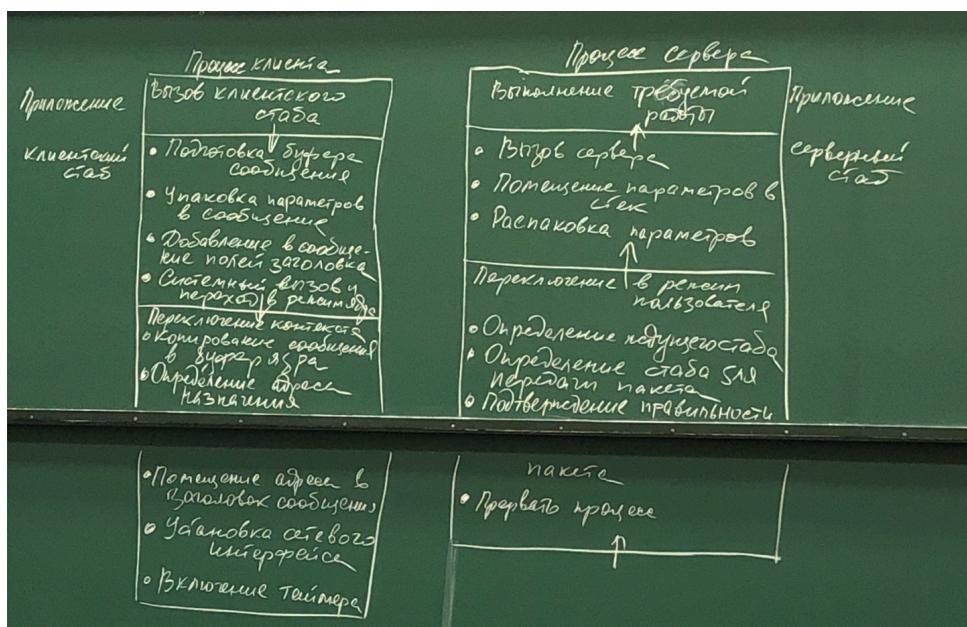
идея вызовов удаленных процедур заключается в том чтобы представить вызов удаленных процедур как вызов локальных процедур  
т.е. чтобы для пользователя такие вызовы выглядели как локальные процедуры

вызовы удаленных процедур (рпц) используют механизмы IPC гнезд фирмны беркли

сервер рпц прослушивает 1 или несколько сокетов и принимает запрос на сокет который от слушает

стабы - фиктивные модули, которые генерируются компилятором  
информация берется из вызова, который в своей программе написал пользователь

на сервере выполняется процедура, которой передаются параметры, распакованные из пакета



в других системах буфер сообщений представляет собой пуль буферов для отдельных полей сообщений

это сделано потому что часто некоторые поля таких сообщений имеют одинаковые значения для всех сообщений (чтобы не заполнять несколько раз)

после этого параметры преобразуются в соответствующий формат и записаны в буфер при этом из буфера пользовательского режима данные сообщения должны быть переписаны в буфер ядра

когда ядро получает управление, определяется адрес назначения и этот адрес помещается в заголовок сообщения

после того как сообщение отправлено, включается таймер  
это делается для того, чтобы использовать тайм аут

т.к. речь идет о распределенной системе, в ней возможны разные критические ситуации, но клиентские приложения не могут ждать вечно, поэтому ставится тайм аут (после его завершения пользователь получит сообщение об ошибке)

... записывается в ... ядра, проверяется правильность полученных данных  
для того чтобы выполнять такую работу, процессор должен на нее перейти (написано, прервать процесс и перейти к обработке полученного сообщения)  
после того как получено подтверждение правильности

.....

клиентский стаб и серверный стаб выполняются в режиме пользователя

именно стабы выполняют основную работу преобразования вызова, который написал в своей программе пользователь, в целую серию действий

## ДИНАМИЧЕСКОЕ СВЯЗЫВАНИЕ

из иллюстрации видно, что необходимо определить адрес сервера он мб задан статически

но недостаток: в случае перемещения сервера или увеличения числа серверов, или в результате изменения интерфейсов, будет необходимо перекомпилировать все программы в которых жестко был указан адрес сервера  
чтобы этого избежать, используется динамическое связывание (binding)

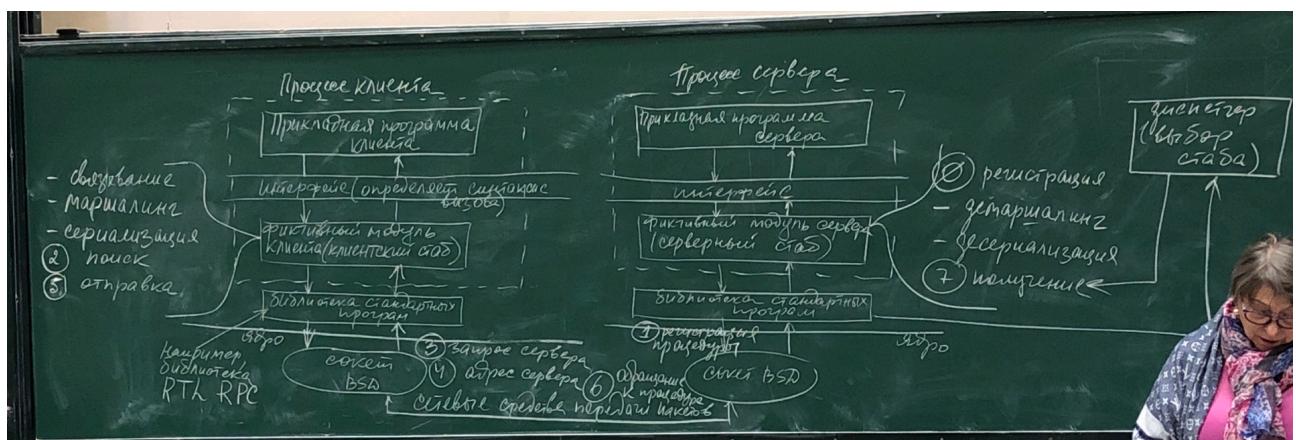
при динамическом связывании создается специализированная служба, ответственная за определение сервера

спец служба представляет из себя новый программный слой

этот слой называется сервером имен и каталогов и предназначен для поиска адресов серверов по именам вызываемых процедур

часто клиентский стаб называется клиентским переходником  
сотов серверный стаб - серверный проводник

действия в плане динамического связывания



шаги 0-1: регистрация процедуры сервера

2-3 запрос адреса сервера, реализующего требуемую процедуру

5-6-7 получение адреса от сервера имен и каталогов, вызов процедуры ...

взаимодействие в распределенных системах связана с передачей сообщений

рассмотрим работу программы make

в онинкс большие программы разделены на несколько файлов

изменения вносимые в один файл предполагают повторную компиляцию только этого файла

программа make после ее запуска проверяет время последней модификации всех исходных файлов и всех объектных файлов программы

если исходный файл имеет время последней модификации больше, чем время модификации соответствующего ему объектного файла, то мы считаем, что исходный файл был изменен и выполнит его перекомпиляцию

например

<имя>.c - 1224|

<имя>.o - 1223|

-> перекомпиляция

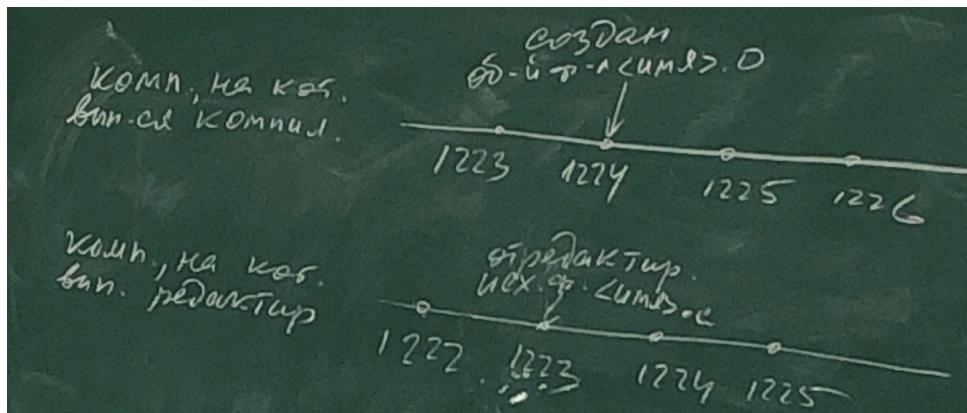
или

<имя>.c - 1224|

<имя>.o - 1225|

-> не требуется

рассмотрим эту ситуацию в распределенной системе



файл  
отредактировали,  
но он  
перекомпилирован  
не будет

в результате сделанные изменения не попадут в исполняемый файл

в распределенной системе каждый комп имеет собственный тактовый генератор,  
собственные локальные часы, использующий соответствующую локальную частоту

генераторы работают с частотой, генерируют ограниченные ...

допустимая ошибка генерации тиков в современных микросхемах кварцевых генераторах -  $10^{-5}$

если таймер генерирует 60 прерываний в секунду, то в час от генерирует 216000 тиков  
это означает, что на конкретном компьютере значение тиков может находиться в  
диапазоне: от 215998 до 216002 тиков в час

служба точного времени

призвана следить за точным временем

момент подъема солнца на максимально подъемную высоту нарыв солнечным переходом  
интервал между 2мя последовательными солнечными переходами - солнечный день

средняя солнечная секунда (mean solar second)

в 1948г были изобретены атомные часы

было определено, что секунда - время за которое атом цезия133 совершает ровно  
9192631770

международное время усредняет результаты эксперимента и выдает глобальное время TAI

солнечный день изменяется

в настоящее время солнечный день удлиняется, а tai остается неизменно

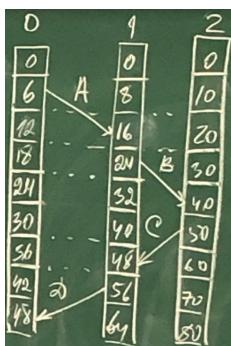
в определенное время вводится дополнительная секунда

и так.... родился високосный год

4.12.18

### ЛОГИЧЕСКИЕ ЧАСЫ ЛАМПОРТА

в какой-то локальной системе процессы обмениваются сообщениями  
но в этой локальной системе важно чтобы выполнялось отношение: случилось до или  
 случилось после  
он предложил  
на каждом компе существуют свой тактовый генератор (простой инкремент счетчика)  
возникает ситуация



модель сильно упрощена

сообщение не может быть отправлено позже, чем получат

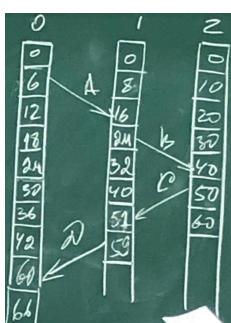
лампорт предложил локальные часы  
они увеличиваются свое значение с каждым событием на i<sup>м</sup> процессоре

Процессор pi

$$ci = ci + d \quad (d > 0 \rightarrow d = 1)$$

посылка сообщения, прием сообщения - события

в посылаемое сообщение добавлять время локаль часов  
получив это сообщение  
сравнивает свое время и время которое пришло в сообщении  
выбирает максимальное  
и свое время устанавливает на 1 больше максимального



$$ci = \max(ci, tmsg+d)$$

такие часы также называются скалярными  
и они обладают свойствами согласованности  
т.е. скалярные часы имеют свойство монотонности и соответствуют свойству  
согласованности  
 $e_i$  и  $e_j : e_i \rightarrow e_j \Rightarrow c(e_i) < c(e_j)$  (это clock)  $<$   $c(e_j)$

скалярные часы могут использоваться для упорядочения событий в распределенных системах

основная проблема: 2 или более событий в разных процессах могут иметь одинаковые временные метки

для упорядочения таких событий используется механизм разрыва связей  
связь разрывается следующий образом:

время выполнения линейно упорядочена и связь между событиями с одинаковыми временными метками нарушается, разрывается путем ... идентификаторами процессов в результате времена метка событий обозначается кортежем  $(t, i)$ , где  $t$  - время, когда произошло событие;  $i$  - идентификатор события в котором оно произошло

полное отношение порядка ( $<$ ) на 2x событиях  $e_i, e_j$

будет отмечаться кортежами  $(t, i), (t, j)$

$e(t, i) \rightarrow e(t, j)$  если  $t_i < t_j$ ,

если  $t_i = t_j$ , то  $i < j$

алгоритм позволяет упорядочить события

## ВЗАИМОИСКЛЮЧЕНИЯ В РАСПРЕДЕЛЕННЫХ СИСТЕМАХ

методы:

1. централизованный алгоритм

2. распределенный алгоритм

3. token ring ?

когда процесс хочет войти в свой критический участок

он посыпает процессу координатору сообщение запрос

в котором указывает, в какую критическую секцию он хочет войти

получив данное сообщение-сообщение-запроси процесс координатор проверяет, не находится ли какой-нибудь другой процесс в данной критической секции

или не хочет ли какой-нибудь другой процесс войти в данную критическую секцию

если таких нет, то координатор посыпает сообщение-разрешение

получая которое процесс входит в свой критический участок

если какой-то процесс уже находится в критическом участке, процесс координатор ставит запрашиваемый процесс в очередь

если другой другой процесс тоже хочет зайти в свой критический участок

здесь могут помочь временные отметки, отправляемые в сообщении

чей запрос раньше

тому послать разрешение

существует 2 подхода к решению этой(????) проблемы

1. алгоритм задиры: если какой-то процесс в результате длительного ожидания, время истекло (это для него означает, что координаты  $r$  отсутствуют), тогда такой процесс предлагает начать выборы нового координатора, посыпая врем процессам, с большими координаторами, если сообщение получает процесс с большим номером (т.е. сообщение о начале выбора получает), то он инициализирует новые выборы, а процессу с меньшим номером посыпает специальное сообщение (о том, что он существует/ выполняется)

процессы, которые инициируют выборы нового координатора, инициирует по цепочке

если какой-то процесс не получает в ответ ни одного сообщения, он является процессом с наибольшим номером и он может стать координатором

чтобы любой процесс в любой локальной системе мог стать координатором, у него должны быть прописаны такие возможности

в любой момент времени, любой процесс может получить сообщение-выборы от других процессов с меньшими номерами, тогда он посыпает сообщение, если не получил ни одного сообщения, он становится координатором и рассыпает всем процессам сообщение-координатор, т.е. процессы извещаются какому процессу посыпать запросы на вхождение в критический участок

2. круговой алгоритм - основан на использовании кольца (физического или логического) (маркер не используется) каждый процесс знает следующий процесс в круговом списке, если процесс обнаружил отсутствие координатора, он посыпает следующему за ним процессу сообщение-выборы, в котором указывает свой номер, если следующий процесс не отвечает, то сообщение посыпается следующему до тех пор, пока не будет найден работающий процесс (каждый работающий процесс добавляет в сообщение свой номер и посыпает дальше по кругу) (когда процесс получивший сообщение, обнаруживает в нем свой номер - это означает, что круг пройден, в этом случае, он меняет тип сообщения с выборы на координатора и это сообщение проходит по кругу, извещая все процессы о списке работающих процессов и координаторе, которым будет процессом с наибольшим номером в списке)

## 2) распределенный процесс

если процесс хочет войти в какой-то критический участок, он формирует сообщение в котором указан номер критической секции в которую он хочет войти и текущее время по своим локальным часам, и посыпает это сообщение n-1 процессу с которыми он взаимодействует

при получении сообщения возможно 3 ситуации:

1. получатель сообщения не находится и не собирается входить в данную критическую секцию, тогда он сразу посыпает отправителю сообщение-ответ с разрешением, что этот процесс отправитель может войти в свой критический участок
2. получатель уже находится в данном критическом участке, он никакого сообщения не отправляет, а ставит поступивший запрос в очередь
3. процесс получатель сам хочет войти в эту критическую секцию, но еще не сделал этого (запрос отправил, но еще не вошел), в этом случае получатель сравнивает временные отметки в обоих сообщениях: в своем и полученном, если его собственный запрос возник позже, то он посыпает сообщение разрешение, иначе сообщение не посыпает, а ставит поступивший запрос в очередь, в результате процесс может войти в свой критический участок, если получит n-1 сообщение-разрешение, когда процесс выходит из своего критического участка, он посыпает всем процессам с которыми взаимодействовал специальное сообщение, таким образом кол-во сообщений на вхождение в одну критическую секцию будет  $2^*(n-1)$ , где n - число процессов (очевидно, что данный алгоритм является ненадежным, т.к. если хотя бы один процесс прекратится, то остановится работа всех остальных процессов)

## 3) token ring

все взаимодействующие процессы образуют логическое кольцо, т.е. каждый процесс знает свой номер в этой цепочке процессов, а так же знает номер ближайшего к нему соседа (следующего по цепочке)

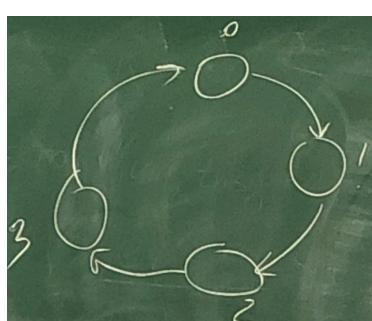
когда логическое кольцо инициализируется, процессу с идентификатором 0 передается токен циркулирует по кольцу, т.е. переходит от его процесса к i+1

когда процесс получает токен от своего соседа, он анализирует, не требуется ли ему самому войти в данную критическую секцию, если да, то он входит в критическую секцию удерживая токен

когда он выходит из критической секции, он отправляет токен дальше по кольцу

если процесс не заинтересован во вхождении, то токен передается дальше

если ни один процесс не заинтересован во вхождении, то



токен с высокой скоростью циркулирует по кольцу

для каждой критической секции свой токен  
в кольце циркулирует столько токенов, сколько критических секций

## НЕДЕЛИМЫЕ ТРАНЗАКЦИИ

транзакции или неделимые транзакции требуют от программиста минимальных знаний для осуществления взаимодействий с помощью транзакций, т.е. программист освобожден от детальных знаний

модель неделимой транзакции пришла из бизнеса  
представим переговорный процесс 2х фирм

в процессе переговором условия договора могут уточняться  
пока договор не подписан обе стороны могут отказаться от его подписания  
но после этого как договор подписан  
делка завершена и должна быть выполнена

в компе аналогично

1 процесс объявляет о желании начать транзакцию с одним или более процессов  
процессы могут некоторые время выполнять некоторые действия  
затем инициатор транзакции объявляет, что хочет завершить транзакцию  
если все взаимодействующие процессы с ним соглашаются, то результаты фиксируются  
если хотяб 1 процесс или несогласен или просто аварийно завершился, то транзакция не завершается, а все измененные объекты возвращаются в свое первоначальное состояние  
до начала транзакции  
такое свойство называется ВСЕ ИЛИ НИЧЕГО

определение транзакции

транзакцией называется последовательность операции над одним или несколькими операциями такими как файлы или записи или элементы бд, которые переводят систему из одного целостного состояния в другое целостное состояние

для того, чтобы иметь возможность программировать с помощью транзакций необходимо иметь набор функций

например (begin\_transaction, end\_transaction, abort\_transaction, read/write)

транзакции обладают следующими свойствами:

- упорядочиваемостью (гарантирует, если 2 или более транзакции выполняются в одно и тоже время, то конечный результат выглядит так, как если бы все транзакции выполнялись последовательно в некотором порядке)
- неделимостью (когда транзакция находится в процессе выполнения никакой другой процесс не может видеть ее промежуточных результатов)
- постоянством (означает что фиксация транзакции постоянна, т.е. никакой сбой не может отменить ее результатов)

механизм транзакции:

- процесс, когда он выполняет транзакцию, работает в индивидуальном рабочем пространстве, которое содержит все необходимые объекты, файлы нужные для выполнения транзакции. пока транзакция выполняется, все изменения выполняются в этом индивидуальном рабочем пространстве. если транзакция не прерывается, а фиксируется, то все изменения копируются в исходные файлы (недостатком такого расхода являются необходимость иметь большое количество копий)
- список намерений (модифицируются сами файлы (копии не создаются), но перед изменением любого блока данных, сначала выполняется запись в специальный журнал (журнал регистраций изменений), в котором отмечается какая транзакция делает изменения, в каком файле, в каком блоке файла, а так же старое и новое значение изменяемого блока, только после успешной записи в журнал делается изменения в исходном файле )(если транзакция фиксируется, то об этом делается регистрация в

журнале, но старые записи сохраняются) (если запись прерывается, то информация из журнала используется для возвращения файлов в исходное состояние (откат))

в распределенных системах фиксация транзакций может потребовать взаимодействия многих транзакций на разных машинах, каждая из которых хранит какие-либо переменные

для достижения свойства неделимости используется протокол специальный, и чаще всего используется протокол двухфазовой фиксации транзакции

10.12.18

табличка с координатором и подчиненным

разбивается на 2 фазы

если процесс тор смог собрать все сообщения от всех подчиненных процессов то транзакция будет выполнена

если хотя бы от одного не придет то все будет остановлено и откатано назад

процессы асинхронные - работающие с собственной скоростью

обычно тупики изображаются двудольным направленным графом

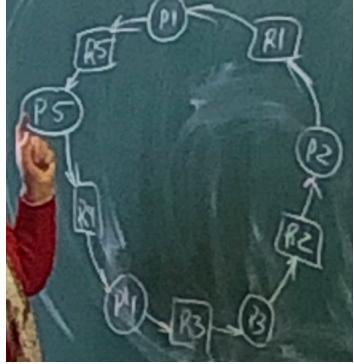


возможна ситуация: процесс п1 захватил семафор с1, а процесс п2 захватил семафор с2  
процессу п1 для продолжения необходимо получить семафор с2, но с2 занят процессом п2  
п2 не может освободить с2, тк ему для продолжения нужен с1, а он занят

классический граф тупика

на месте семафора может быть ресурс (или их еще так называют (не помню))

на примере философов  
замкнутая цепь запросов:

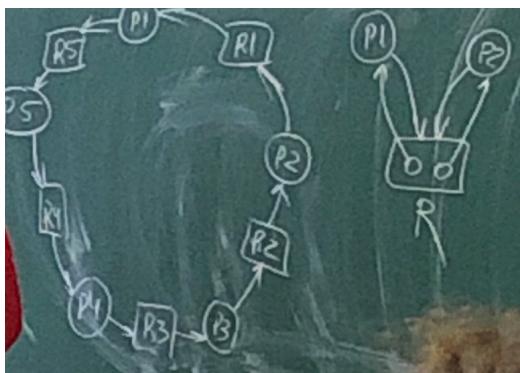


особенно неприятны тупики в распределенных системах

потому что накладывается еще время

тк в распределенных системах захват ресурсов занимает время связанное с самой сетью

тупик (тупиковая ситуация) - это ситуация, возникающая в результате монопольного использования разделяемых ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый или непосредственно или через цепочку загрузок другим процессов, ожидающим освобождения ресурса занятого первым процессом



оба процесса не могут продолжать выполняться и не могут освободить единицы ресурса

## ТИПЫ РЕСУРСОВ И ИХ ТУПИКИ

особенности их использования

ресурсы классифицируются на :

- повторно используемые
- потребляемые

все определения из Шоу

Повторно используемые - их использование никак на них не влияет  
это ресурсы которые не изменяются при их использовании: аппаратная составляющая системы и это реентерабильные коды, те повторно вводимые коды (использование этих кодов на них никак не влияет), семафоры, разделяемые ресурсы

к потребляемым (перестает существовать после потребления) относятся сообщения

повторно используемые ресурсы имеют следующие свойства:

1. число единиц ресурсов ограничено и постоянно (изменения числа повторно используемых ресурсов является исключительной ситуацией)

потребляемые ресурсы:

1. динамическое изменение числа сообщений (при получении сообщения оно перестает существовать) (известны производители сообщений, и т.д. разные вариации ситуаций когда кто-то известен а кто-то нет)

4 условия возникновения тупика:

1. взаимоисключение (когда процессы монопольно используют предоставляемые им ресурсы)
2. ожидание (процессы удерживают занимаемые ими ресурсы, ожидая предоставления им для того чтобы иметь возможность продолжить свое выполнение других ресурсов)
3. неперераспределемость (ресурсы нельзя отобрать у процессов до завершения процессов или до добровольно освобождения ресурсов самим процессом) по premtim
4. круговое ожидание (возникновение замкнутой цепи запросов в которой каждый процесс занимает ресурс который необходим следующему процессу в цепочки для продолжения его выполнения)

методы борьбы тупиками:

1. пропустила
2. обход тупика или недопущение (в системе возникают действия которые не допускают возникновения тупика)
3. обнаружение и восстановление
4. стратегия халендора ? (он показал что возникновение тупика невозможно если нарушено хотяб одно из условий возникновения тупика)

подходы к устраниению возможности возникновения тупика:

1. опережающее требование (процессы до начала своего выполнения запрашивают все необходимые им ресурсы) (с точки зрения использования ресурсов такой подход приведет к неэффективному использованию (процесс получает ресурсы за долго до их использования) (может привести к бесконечному откладыванию)) (чтобы более эффективно использовать ресурсы процесс может быть разделен на несколько задач и для каждой задачи выделяются ресурсы, более эффективно, но более сложно)
2. иерархическое распределение ресурсов (ресурсы в системе упорядочиваются, те делятся на классы каждому из которых присваивается порядковый номер, при этом процессы могут запрашивать ресурсы только из классов со строго большими номерами чем классы тех ресурсов которые онидерживают) (если ресурс порядковый номер которого меньше чем номера ресурсов которые он уже получил, то процесс должен все остановить и запросить ресурсы в правильном порядке (это обрывок фразы))
3. устранение условия неперераспределимости (если процесс в результате сделанного запроса не может получить ресурс, то он должен вернуть занимаемые им ресурсы, процесс должен быть приостановлен и через некоторое время возобновлен насчитывая на исправление в системе)

в 1м методе возникновение тупика невозможно

метод 2 :

обход тупиков:

в этом методе тупик возможен, но выполняются действия чтобы недопустимо возникновения тупика

1. алгоритм банкира (алгоритм дейкстры) (в качестве банкира выступает менеджер ресурсов который удовлетворяет заявки процессов на ресурсы) (для реализации этого алгоритма имеются ограничения: заявки отражают их максимальную потребность в каждом типе ресурсов, количество процессов ограничено (известно постоянно), количество ресурсов каждого типа в системе известно) (наличие такой информации позволяет менеджеру проводить анализ и недопекать возникновения тупика) (кроме перечисленных условий существует еще: процесс не может запросить количество единиц ресурсов больше чем указано в его заявке по каждому классу ресурсов) (менеджер ресурсов имеет возможность анализировать возникающие в системе ситуации) (каждый запрос проверяется по отношению к его заявке)

(все о чем говорили относится к повторно используемым ресурсам)

рассмотрим пример для одного типа ресурсов

| Процесс | Текущее распределение | Своб. ед. рес. | Заявку |
|---------|-----------------------|----------------|--------|
| P1      | 1                     | 4              |        |
| P2      | 3                     | 5              |        |
| P3      | 6                     | 9              |        |

процесс п2 может успешно завершиться даже если он запросит максимальное количество единиц ресурса  
завершившись процесс п2 вернет системе занимаемые им единицы ресурса

| Процессы | Текущее<br>распред. | Задачи |
|----------|---------------------|--------|
| P1       | 2                   | 4      |
| P2       | 3                   | 5      |
| P3       | 5                   | 9      |

такая ситуация является небезопасной для тупика, тк у системы нет столько единиц ресурсов

формально состояние системы является безопасным относительно тупика если существует последовательность процессов

1. 1й процесс обязательно завершится, тк даже если он запросит максимальное кол-во единиц ресурса у системы есть достаточное кол-во единиц ресурса для завершения его запрос
2. 2й процесс в последовательности может завершиться, если завершится первый процесс и вернет занимаемые им ед ресурса, что в сумме со своими позволит удовлетворить ...
3. iй может завершиться если все i-1 процессы успешно завершаться и освободят занимаемые ими ресурсы и сумма освобожденных и свободных единиц ресурсов системы сможет удовлетворить его максимальную потребность в ед ресурсов

таким образом всякий раз когда процесс делает новый запрос менеджер ресурсов должен найти успешно завершающуюся последовательность процессов и только в этом случае запрос может быть удовлетворен

система удовлетворяет только те запросы, которые оставляют ее состояние надежным относительно тупика

запросы, приводящие систему в ненадежное состояние, откладываются на некоторое время в предположении, что через некоторое время ситуация в системе изменится и запросы этих процессов смогут быть удовлетворены

Т.о. система постоянно поддерживается в нормальном состоянии и это гарантирует, что рано или поздно все запросы смогут быть удовлетворены

в результате необходимо исследовать n! последовательности прежде чем вынести решение: состояние системы надежно или нет

это крайне затратно, поэтому алгоритм банкира имеет теоретическое значение он показывает что существует путь анализа состояния системы с целью предотвращения тупиков

существует аппроксимация алгоритма банкира  $O(n^2)$

S - к-во процессов

цикл пока S > []

начало

найти процесс A в S, который может завершиться;

если нет, то сост-е небезопасное - вывести процесс A из S: отобрать у A рес-сы и вернуть их в пул своб.рес.;

конец;

состояние безопасное;

алгоритм хабермана (габермана)

менеджер ресурсов поддерживает массив  $S[0, \dots, r-1]$ , где r - число единиц ресурса

$S[i] = r-i$  для всех  $i: 0 \leq i < r$

если процесс, заявивший С ед рес и удерживающий h ед рес запрашивает еще одну ед рес, то  $S[i]$  декрем-ся для всех  $i: 0 \leq i < C-h$

если какие-то  $S[i]$  становятся  $< 0$ , то состояние - опасное относительно тупика

### 3 метод (ОБНАРУЖЕНИЕ ТУПИКОВ):

если систему рассматривать как декартово произведение множества состояний, где под состоянием понимается состояние ресурса: свободны или распределены, причем состояние может определяться процессами в результате запросов ресурсов и последующих их получений, также в результате освобождений процессами ресурсов, то система находится в тупике, если она не может поменять своего состояния, ни в результате запроса, ни в результате освобождения ресурса

вопрос: каким образом можно определить находится ли какое-то количество взаимодействующих процессов в тупике, решается с помощью грамматических моделей Хольта

модель Хольта представляет собой двудольный или бахроматический граф в котором множество вершин разбивается на 2 не пересекающихся подмножеств, а именно множество вершин процессов и множество вершин ресурсов  
вершины соединяются дугами, причем никакая дуга не может соединять вершины одного и того же подмножества

X: P, R

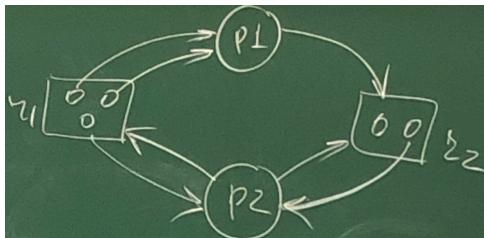
дуга (r, p) -> приоритет  
(p, r)

обнаружение тупиков выполняется в помощь редукции графа, т.е. сокращения  
если граф полностью сокращаем, то система не находится в состояние тупика  
сокращать можно вершины процессов которые могут приобретать или освобождать  
ресурсы

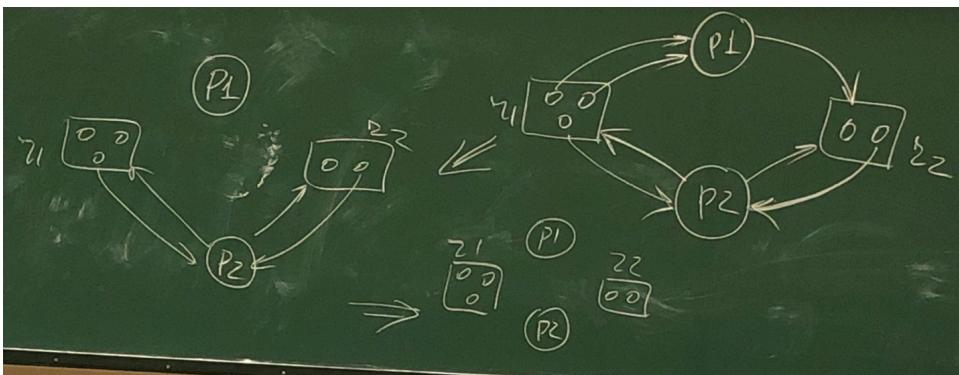
рассмотрим пример полностью сокращаемого графа

процессы p1, p2

ресурсы p1 (3 единицы), p2 (2 единицы)

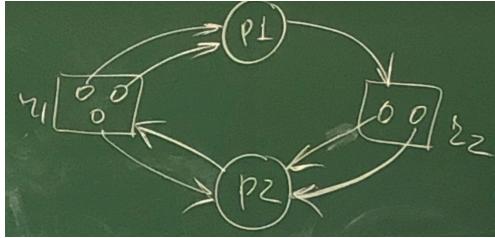


данний график может быть сокращен по вершине  $p_1$



данний график полностью сокращаем, т.е. запросы процессов могут быть последовательно удовлетворены

не сокращаемый граф:



граф сокращается процессом  $p_i$  если вершина этого процесса является не заблокированной и не изолированной  
сокращение выполняется путем всех ребер входящих и выходящих в вершины  $p_i$

Теорема 1

граф является полностью сокращаемым, если существует такая последовательность сокращения, которая устраниет все дуги, если граф нельзя полностью сократить, то анализируемое состояние является полностью тупиковым

Теорема 2

цикл в графе повторно используемых ресурсов является необходимым условием тупика

Теорема

если  $S$  не является состоянием тупика и  $S \rightarrow^i (i \text{ над стрелочкой}) T$ , где  $T$  - состояние тупика, то операция процесса  $p_i$  есть запрос, в результате которого  $p_i$  в  $T$  находится в тупике (тупик может возникнуть в системе взаимодействующих процессов только в результате запроса)

если процесс находится в тупике он не может выполнить ни приобретение, ни получение, ни освобождение ресурса

т.к. граф двудольный он может быть представлен с помощью 2х матриц или двусвязных списков

2 матрицы: 1 - матрица запросов ( $r, r'$ ), 2 - матрица распределений, выделения ресурсов ( $r, r'$ )

матрица запросов  $A$  в которой каждый элемент  $\{a_{ij}\}$  отражает количество запрашиваемых единиц ресурса  $r_j$  процесса  $p_i$  (в графе это дуга)

матрица текущего распределения  $B$   $\{b_{ij}\}$  каждый элемент отражает количество единиц ресурса  $r_i$  выделенных процессу  $p_j$  в процессе его выполнения до какого-то момента времени

для того чтобы проводить анализ необходимо иметь вектор свободных единиц каждого ресурса

$F = \{\dots, r_{fi}, \dots\}$  - свободные единицы ресурса

алгоритмы обнаружения тупиков:

1. метод прямого обнаружения (заключается в последовательном просмотре матрицы запросов, которые являются в графе дугами, такой просмотр должен выявить запросы которые были удовлетворены и соответствующие дуги могут быть сокращены) (чтобы определить можно ли сократить дугу: анализируется втор свободных единиц ресурса (если у системы есть свободная единица запрашиваемого ресурса, то запрос может быть удовлетворен, а дуга сокращена)) (если в результате всех сокращений останутся не сокращенных вершины - система находится в тупике)

более эффективно: сравнивать вектора:

вектора С и D

$C \leq D$  - это означает, что каждый элемент вектора  $C \leq$  каждому элементу вектора D

если iй процесс запрашивает jй ресурс и при этом строка процесса его процесса  $\leq$  вектору свободных единиц ресурсов, то такой запрос может быть удовлетворен, т.е. процесс выполнивший такой запрос может завершится, т.е. граф мб сокращен по вершине pi

| распределение |       |  |  |            |       |    |
|---------------|-------|--|--|------------|-------|----|
| P/р           | 1 2 3 |  |  | Запрос (I) | F     |    |
| 1             | 0 1 1 |  |  | 1 1 1 0    | 1 1 1 | II |
| 2             | 1 3 0 |  |  | 2 0 0 1    | 0 0 0 |    |
| 3             | 1 5 0 |  |  | 3 1 1 2    | 1 5 0 |    |
|               | 2 9 1 |  |  |            |       |    |
|               |       |  |  |            |       |    |
|               |       |  |  |            |       |    |

$P/R = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 0 & 1 & 1 \\ 2 & 1 & 3 & 0 \\ 3 & 1 & 5 & 0 \\ \hline 2 & 9 & 1 \end{pmatrix}$ 
 $Q/R = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 3 & 1 & 1 & 2 \\ \hline \end{pmatrix}$ 
 $F = \begin{pmatrix} 1 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ \hline 2 & 9 & 1 \end{pmatrix}$ 
 $\sum \rightarrow 6 \ 9 \ 3$

I  
 II  
 III  
 $\downarrow$   
 $\downarrow$   
 $\downarrow$   
 $F = \{5, 3, 2\}$   
 $F = \{5, 4, 3\}$   
 $F = \{6, 9, 3\}$

2я строка запросов = строке единиц ресурсов в результате строки будет обнулена, а запрос удовлетворен

17.12.2018

матрица распределений - ТР (таблица распределенных ресурсов)

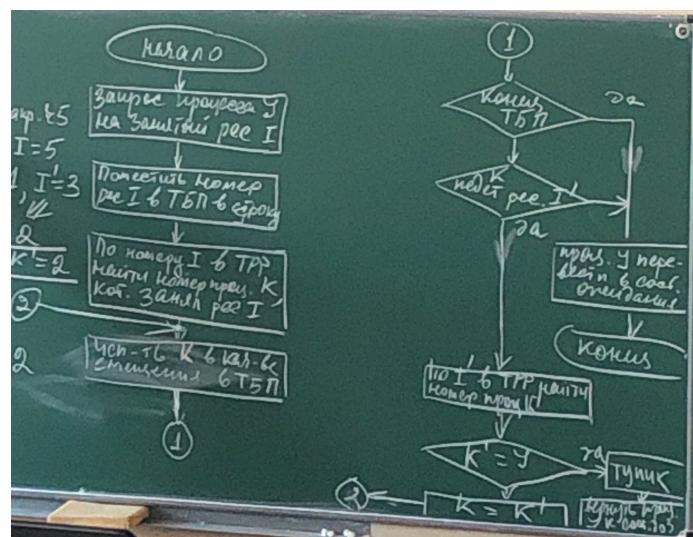
матрица ожидающих процессов - ТБВ

рассмотрим процесс единичного распределения ресурсов

если 5 различных ресурсов, каждый из которых в системе в единичном экземпляре и 3 процессы которые конкурируют

1. процесс p1 запросил и получил в свое распоряжение ресурс p1 (p1 занимает p1)
2. p2 занимает p3
3. p3 занимает p2
4. p2 занимает p4
5. p1 занимает p5
6. p1 запрашивает p3  $y=1, i=3, k=2$ , p1 - блок
7. p2 запрашивает p2  $y=2, i=2, k=3$ , p2 - блок

| TP P       |                   |
|------------|-------------------|
| рес        | прос              |
| 1          | 1                 |
| 2          | 3                 |
| 3          | 2                 |
| 4          |                   |
| 5          |                   |
| <u>ТБВ</u> | <u>К' = 8</u>     |
| ресурс     | <u>К = К' = 2</u> |
| 1   3      |                   |
| 2   2      |                   |
| 3   5      | <u>I' = 2</u>     |
|            | <u>K' = 3</u>     |
|            | <u>3 = 3!</u>     |



после того как тупик обнаружен

### Выходы из тупика

2 общих подхода:

1. прекращение выполнения процессов (процессы находящиеся в тупике последовательно уничтожаются в некотором порядке до тех пор пока не станет доступно достаточноное количество ресурсов для устранения тупика ((если не предусмотрено специальных защитных действий))
2. перехват ресурсов (процессы которые находятся в тупике не завершаются, а ресурсы забираются у процессов которые в тупике не находятся ... процессы у которых забираются ресурсы помещаются под множество блокируемых процессов . для каждого указывается количество отобранных единиц ресурсов)

## ЯРХИТЕКТУРА ЯДЕР ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

существует 2 основных типа ядер:

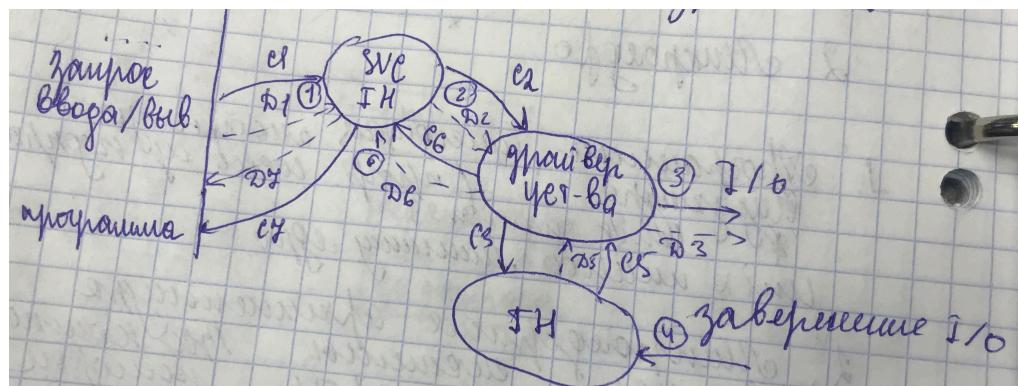
1. монолитные ядро - программа имеющая модульную структуру (состоящая из подпрограмм)
2. микро ядро (микроядерная архитектура) - в этой архитектуре все компоненты системы являются самостоятельными программами которые возможно выполняются в разных адресных пространствах (в такой системе существует модуль программы, который называется микроядро, обеспечивающее взаимодействие между программами системы)

unix имеет минимизированное ядро

рассмотрим монолитное ядро

линукс поддерживает изменение функциональности ядра без его перекомпиляции с помощью загружаемых модулей ядра

структура ядра 4.4 BSD



25.12.18

загружаемые модули ядра

функциональные драйвера - являются ... нижнего уровня

обработчики прерываний входят в состав соответствующих драйверов устройств

как правило в ОС с монолитным ядром системный вызов выполняется командой int

| ОС   |        |         |       |     |
|------|--------|---------|-------|-----|
| инт. | MS-DOS | Windows | Linux | QNX |
| int  | 21h    | 2Eh     | 80h   | 23h |

начиная с 2008 ОС перешли от использования программного прерывания int к системным вызовам, таким как sysenter, sysexit

обработчик прерываний является частью кода ядра

такой обработчик загружает в регистр стека SP адрес стека режима ядра

в стеке режима ядра сохраняются значения регистров прерванной программы

затем по соответствующей таблице находится адрес обработчика данного системного вызова и происходит передача ей управления

по завершении работы системного вызова в регистры процессора загружаются

возвращаемые значения и указатель стека переключается на адрес стека режима пользователя

в наших системах каждый процесс должен иметь минимум 2 стека (kernelstack, userstack)

состояние режима ... ядра является частью контекста процесса и должно сохраняться при переключениях контекста

один и тот же стек нельзя использовать как юзер и как кёнэл стек

это связано с тем, что программа может довольно свободно работать со стеком режима

пользователя, может организовывать несколько стеков режима пользователя, может

менять их размеры, поэтому стек пользователя не подходит на роль стека ядра

если у программы имеется несколько пользовательских стеков какой из них обрабатывает действия пользователя в ядре

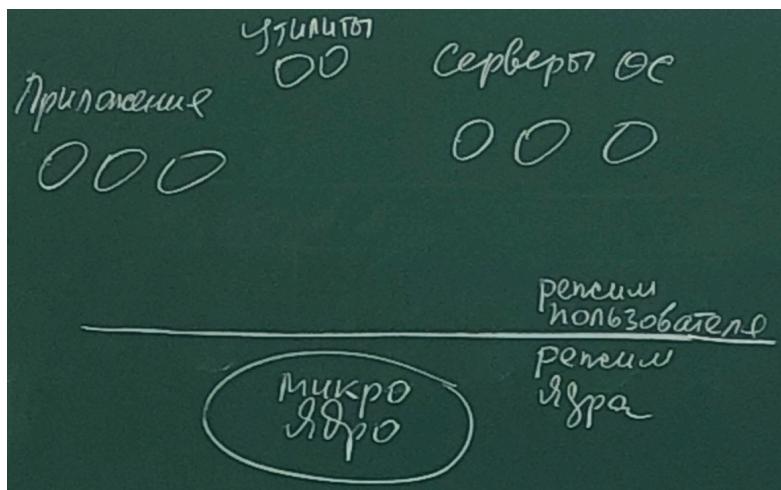
в режиме ядра могут одновременно находиться большое число процессов, поэтому каждый из этих процессов должен иметь возможность записывать нужную ему информацию и отсюда каждый процесс должен иметь свой собственный стек ядра

## МИКРОЯДЕРНАЯ АРХИТЕКТУРА

микроядро - это модуль ОС, который обеспечивает низкоуровневые операции, такие как выделение процессам физических ресурсов, а именно: процессорного времени, физических страниц, физических устройств, низкоуровневую обработку аппаратных прерываний, а также взаимодействие между процессами

микроядерная архитектура - это такая организация структуры ОС при которой все ее компоненты, кроме микроядра, являются самостоятельными процессами, работающими возможно в разных адресных пространствах и взаимодействующими друг с другом путем передачи сообщений

микроядерная архитектура реализуется в соответствии с моделью клиент-сервер

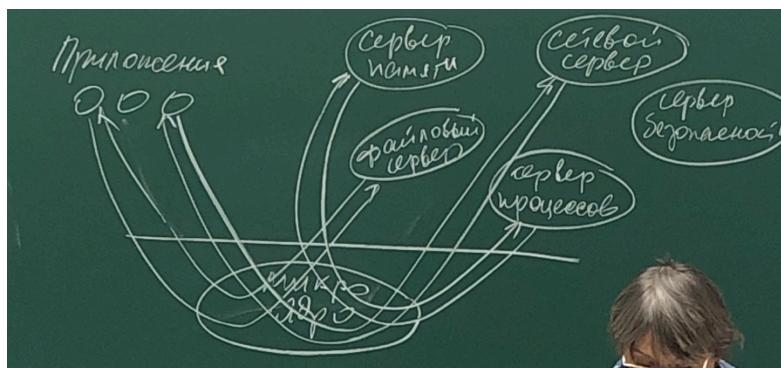


самое низкоуровневое действие - выделение процессу процессорного времени (диспетчеризация)

самое низкоуровневое действие при управлении памятью - выделение процессу конкретной физической страницы

решение какую страницу вытеснить может быть выяснено на более высоком уровне

механизмы обмена сообщениями могут быть разными, но обмен сообщениями может выполняться только через адресное пространство ядра системы, т.е. через микроядро  
микроядро должно предоставлять процессам способы, механизмы обмена сообщениями



такое взаимодействие ОС должно выполняться по строгим протоколом, т.е. запросы должны сопровождаться подтверждением их получения

3 состояния блокировки процесса при передачи сообщения

если выполнение системного вызова в ОС с монолитным ядром связано с двумя переключениями 1. из режима пользователя в режим ядра 2. наоборот то в микроядерной архитектуре таких переключений будет минимум 4

функции управления файлами каталогами и т.п. выполняются в пользовательском пространстве

МАСН

ядро управляет 5ю базовыми пространствами:

1. процессы
2. нити
3. объекты памяти
4. порты
5. сообщения

процесс mach является единицей декомпозиции в системе  
например процессу принадлежат коммуникационные каналы

процессы делятся на потоки, поток является единицей диспетчеризации (выполнения)

поток принадлежит строго одному процессу

процесс может иметь один единственный поток

объекты памяти - это структура данных которая может быть отражена в адресное пространство процесса

объекты памяти могут занимать одну или несколько страниц  
являются основой для управления виртуальной памятью

если процесс ссылается ан объект памяти отсутствующий в физическом пространстве, то возникает страничное прерывание которое перехватывается ядром

ядро mach для загрузки отсутствующей странице посылает сообщение серверу пользовательского режима, он самостоятельно выполняет нужную операцию

межпроцессное взаимодействие в ОС mach осуществляется путем передачи сообщений

для посылки и приема сообщений процесс просит ядро создать защищенный почтовый ящик - порт

порт создается в адресном пространстве ядра и способен поддерживать очередь сообщений

у процесса может быть набор портов разного назначения

это могут быть:

- порт процесса, который используется для взаимодействия с ядром. многие функции процесс совершают путем отправки сообщений в порт процесса, а не с помощью системного вызова
- порт загрузки используется при инициализации, когда система стартует
- порт особых ситуаций, используется системой для передачи сообщений об ошибках процессу (деление на 0)
- зарегистрированные порты, используются процессами для обеспечения взаимодействия с системными серверами

отклик системы - важнейший показатель чего-то у ОС

в качестве примера рассмотрим ОС qnx

qnx - это ОС поддерживает управление виртуальной памятью и взаимодействие процессов с помощью передачи сообщений

qnx обеспечивает все необходимые составляющие систем реального времени (многозадачность, диспетчеризация потоков на основе...)

особенность: быстрое переключение контекста

поддерживает вложенные прерывания

структура ядра qnx



задействованы все 4 кольца защиты

поддерживает управление виртуально памятью

в точки зрения процессов реального времени  
для чего они предназначены

под детерминированностью понимается, то что для выполнения одного сервиса ОС требуется временной интервал заведомо известной продолжительности  
теоретически это время может быть вычислено по соответствующим формулам, которые не могут содержать никаких параметров случайного характера

в отличии от ОСРВ системы общего назначения не являются детерминированными  
их сервисы могут допускать случайные задержки при работе, что приведет к увеличению времени отклика на действие пользователя

планирование в системах реального времени:

внешние события на которые должна реагировать система можно разделить на периодические и непериодически (возникающие в произвольные моменты времени)

возможно наличие как потока периодических событий в системе так и непериодических

если речь идет о периодических событиях

в зависимости от времени затрачиваемого на обработку каждого события можно определить: в состоянии ли система своевременно их обслужить

если в систему поступает  $m$  периодических событий, ie событие имеет период  $p_i$  и на его обработку тратится  $c_i$  секунд времени работы процесса, то если  $\sum_{i=1}^m c_i/p_i \leq 1$

системы которые удовлетворяют этому условию называются планируемыми

28.12.18

## СЕМИНАР

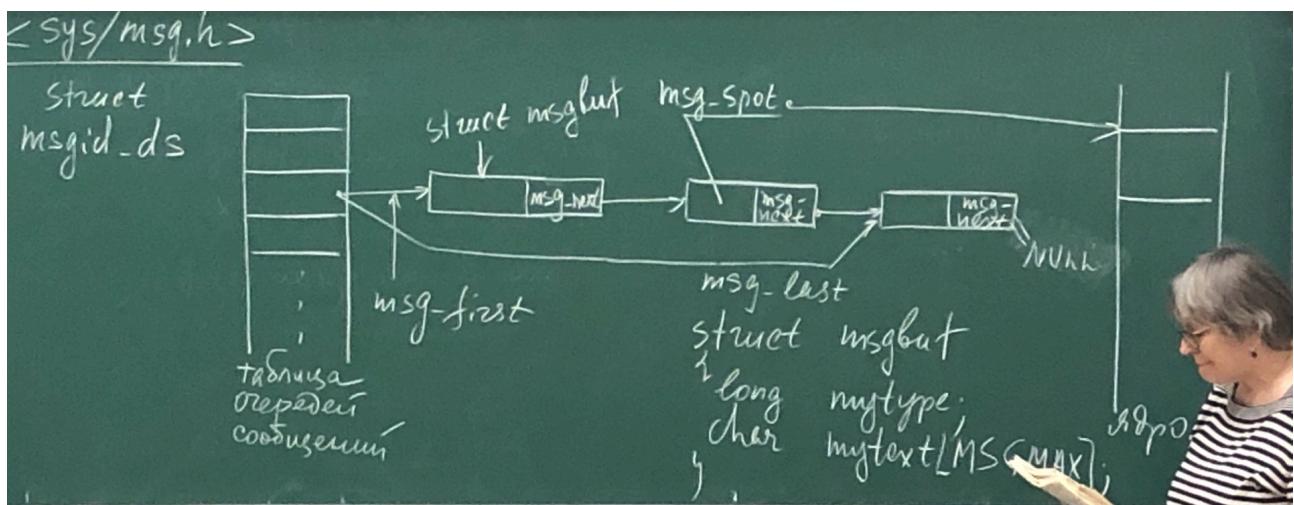
канал является средством потоков передачи данных  
если сообщение из канала считано оно перестает существовать

не существует на разделяемых сегментах никаких средств доступа управления  
разделяемых сегментов

разделяемые сегменты используются совместно с семафорами

### ОЧЕРЕДИ СООБЩЕНИЙ

в адресном пространстве ядра системы есть таблица очередей сообщений



каждая строка этой таблицы описывает одну очередь сообщений  
очередь сообщений представляет собой односвязный список  
голова - самое старое сообщение (FIFO)

последний указатель = NULL

на содержание сообщения указывает указатель `msg_spot`  
а само сообщение находится в области данных ядра системы

само сообщение имеет описанную на фото структуру

потоковая модель данных в очередях сообщений не используется  
в таком сообщении допустимы любые данные  
длина данных выбирается пользователем  
в момент создания очереди определяется максимально возможный размер сообщения

тк содержание сообщений находится в ядре системе  
система ограничивает количество сообщений которые могут быть одновременно созданы

системные вызовы:

- `msgget()`
- `msgctl()`
- `msgsnd()`
- `msgrcv()`

очереди сообщений могут исключать блокировки процессов

когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает ее в конец связного списка записей, которые соответствуют сообщениям указанной очереди

в каждой такой записи очереди указывает тип сообщения, размер сообщения в байтах и указатель на другую область памяти в которой находится сообщение  
ядро копирует сообщение из адресного пространства процесса отправителя в область данных ядра системы, при этом процесс отправитель сообщения может завершиться, а сообщение переданное в очередь сообщений останется доступным другим процессам

когда какой-либо процесс выбирает сообщение из очереди, ядро копирует его в адресное пространство процесса получателя из своего адресного пространства  
после чего сообщение удаляется и не только содержание, но и соответствующий элемент очереди (соответствующая запись)

процесс может выбрать сообщение из очереди нескольким способами:

1. процесс может взять из очереди самое старое сообщение независимо от его типа
2. процесс может взять сообщение если идентификатор этого сообщения совпадает идентификатором, который указал процесс (если существует несколько сообщений с таким идентификатором, то берется самое старое)
3. процесс может взять сообщение числовое значение типа которого, является наименьшим из меньших или равным значению типа указанного процесса (если существует несколько сообщений, то берется самое старое)

БЛОКИРОВКИ - ЗЛО, но в некоторых случаях неизбежно

---

---

## ЛЕКЦИЯ

---

особенности ОС qnx

она поддерживает несколько алгоритмов планирования потоков:

1. FIFO (дисциплина планирования без переключения потоков)( будет выполняться до тех пор пока не завершится или не будет вытеснен более приоритетным потоком, или он может добровольно уступить)
2. RR (поток выполняется в течении кванта, квант в 4 раза больше периода тактовой частоты = 1млс)
3. спорадическая планирование (sporadic scheduling)

qnx поддерживает управление виртуальной памятью

возникает страничное исключение которое обрабатывается системой  
задача системы скопировать страницу которая находится во вторичной памяти в физическую память  
на этот момент процесс блокируется

(  
в системах реального времени могут быть процессы для которых процессы не могут блокироваться  
поэтому для них не может использоваться виртуальная память  
) // возможно неправильно

в qnx потоки могут иметь 2 состояния:

1. Blocked
2. Ready

и выделяются причины блокировок, например:

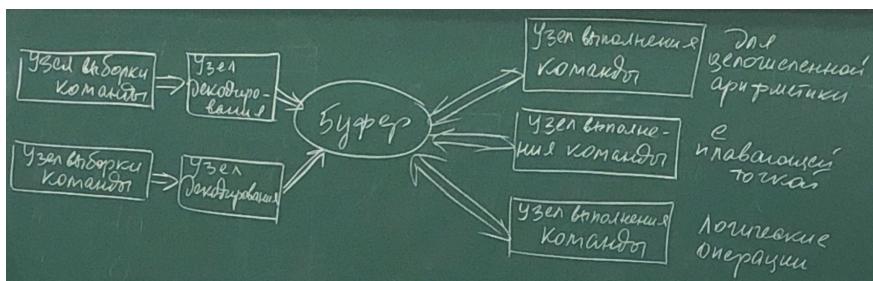
- SIGWAITINFO
- RECEIVE
- SEND
- и т д

## ТОЧНЫЕ И НЕТОЧНЫЕ ПРЕРЫВАНИЯ

диаграмма состояний цикла выполнения команды



исполнение предполагает 3 действия: выборка, дешифрация, исполнение  
отсюда возможна реализация конвейера



в результате декодирования в буфер записываются действия которые необходимо выполнить и соответствующие операции  
у суперскалярного процессора имеются несколько ... выполнения

суперскалярный процессор

за один такт считывает 2 или более команд

которые декодирующая и результат сбрасывается в буфер хранения

из этого буфера выборка осуществляется различными узлами выполнения команд  
в результате команды часто выполняются не в том порядке, в котором они следуют в программе

отсюда одна из причин необходимости барьеров

в большинстве случаев аппаратура должна гарантировать, что результат будет совпадать с тем, который получился бы при строго последовательном выполнении команд

в структуре с конвейерами различные команды находятся на разных стадиях выполнения  
в момент прерывания значение счетчика команд может не отражать истинные границы  
между выполненными и еще не выполненными командами

скорее всего он будет указывать на адрес очередной команды которую следует считать из памяти, а не той, которая выполнена

следовательно при возврате из прерывания ОС не может просто начать заполнять конвейер с адреса, который находится в счетчике команд

ОС должна определить последнюю выполненную команду, что требует выполнения серьезного анализа состояния процессора

в суперскалярных машинах все еще хуже

поскольку команды могут выполняться не в порядке их расположения в памяти  
четкой границы между уже выполненными и еще не выполненными командами нет  
например может возникнуть ситуация, когда команды 1, 2, 3, 5, 8 - выполнены  
а команды 4, 6, 9, 10 - еще нет

при этом счетчик команд может указывать на команды или 9 или 10 или 11

поэтому вводится понятие точного прерывания

точное прерывание - это прерывание, которое оставляет машину в строго определенном состоянии

имеет следующие свойства:

1. счетчик команд

2. ни одна команда после той, на которую указывает счетчик команд не выполнена

3. состояние команды на которую указывает счетчик команд известно

обратим внимание на то, что в перечислении ничего не говорится о том, что команды после той, на которую указывает счетчик команд, не могут начать выполняться

фразы в перечислении следует понимать так, что все изменения связанные с этими

командами должны быть отменены до начала обработки прерывания

прерывания не удовлетворяющие перечисленным свойствам называются неточными

вспомним что счетчик команд при аппаратном прерывании обычно указывает на

следующую команду которая должна выполняться

а при исключении на команду которая вызвала исключение

существуют машины с неточными прерываниями

обычно они выгружают в стек огромное количество данных

для того, чтобы позволить ОС определить, что реально происходило в момент прядении  
сохранение больших объемов данных при каждом прерывании сильно замедляет вход в

процедуру обработки прерывания

восстановление после прерывания является сложным и медленным

все это приводит к тому, что сверхбыстрый суперскалярный процессор непригоден к реальному времени из-за своих слишком медленных прерываний

суперскалярные процессоры «пентиум про» и все его приемники поддерживают точные прерывания

ценой за точные прерывания оказывается сложная внутрипроцессорная логика перывания, т.е. цена - это не скорость , а сложность процессора