

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
ИМЕНИ Н. Э. БАУМАНА

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»  
КАФЕДРА «ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ЭВМ  
И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ»



ЛЕКЦИИ ПО КУРСУ

---

## Операционные системы

---

СЕМЕСТР №1

Москва, 2015

# Содержание

1	История вычислительной техники . . . . .	5
1.1	Первое поколение 1944 – 1955 . . . . .	5
1.2	Второе поколение (транзисторная логика). . . . .	6
1.3	Третье поколение. 60-е . . . . .	6
1.3.1	Канальная архитектура . . . . .	6
1.4	4 поколение. 1970 – н.в. . . . .	9
1.4.0.1	Реальное время . . . . .	9
2	Разработка операционных систем . . . . .	10
2.0.0.1	Реентерабельность . . . . .	11
2.0.0.2	Разработка ОС (аспекты подходов) . . . . .	11
2.0.0.3	Особенности построения ОС . . . . .	11
2.0.0.4	Сложность проектирования ОС . . . . .	12
2.0.0.5	Концепции виртуальной машины . . . . .	12
3	Процессы . . . . .	13
3.0.1	Виды ОС . . . . .	15
3.0.2	Алгоритмы планирования для систем пакетной обработки . . . . .	15
3.0.3	Алгоритмы планирования для интерактивных систем . . . . .	15
3.1	Создание нового процесса . . . . .	16
3.1.1	События, переводящие систему в режим ядра (система прерывания) . . . . .	17
4	Потоки . . . . .	18
5	Процессы ОС Unix . . . . .	20
5.1	Системный вызов fork() . . . . .	20
5.2	Системный вызов exec() . . . . .	22
5.3	Иерархия процессов. . . . .	23
5.4	Системный вызов wait() . . . . .	25
5.5	Группы процессов (сигналы) . . . . .	25
5.5.1	Системный вызов kill() . . . . .	26
5.5.2	Системный вызов signall() . . . . .	27
6	Управление памятью . . . . .	27
6.1	Управлять памятью . . . . .	32
6.2	Отредактировать таблицы . . . . .	33
6.3	Стратегия выделения памяти . . . . .	35
6.4	Перемещаемые разделы . . . . .	35
6.5	Виртуальная память . . . . .	35
6.5.1	Управление памятью страницами по запросам . . . . .	36
6.5.1.1	Прямое отображение . . . . .	36

6.5.1.2	Ассоциативное отображение . . . . .	39
6.5.1.3	Ассоциативное – прямое отображение . . . . .	39
6.5.2	Схема преобразования сегментами по запросам . . . . .	41
6.5.3	Управление памяти сегментами, поделенными на страницы .	43
6.5.4	Алгоритмы Page replacement (замещение страниц) . . . . .	45
6.5.4.1	Выталкивание случайной страницы . . . . .	45
6.5.4.2	FIFO . . . . .	45
6.5.4.3	LRU . . . . .	45
6.5.4.4	Алгоритм NUR . . . . .	46
6.5.4.5	LFU наименее часто используемая страница . . . . .	48
6.5.4.6	Метод связанных пар . . . . .	48
6.5.5	Вытеснения . . . . .	49
6.6	Процессоры Intel . . . . .	54
7	Взаимодействие параллельных процессов . . . . .	57
7.1	Программная реализация . . . . .	57
7.1.1	Алгоритм Дейкера . . . . .	58
7.1.2	Алгоритм Петерсона . . . . .	59
7.1.3	Алгоритм Bakery (Лампорта) . . . . .	59
7.2	Аппаратная реализация . . . . .	60
7.3	Семафоры Дейкстры . . . . .	61
7.4	Мьютексы . . . . .	62
7.5	Задача обедающих философов . . . . .	62
7.6	Задача производство потребление . . . . .	64
7.7	Мониторы . . . . .	65
7.7.1	Простой монитор . . . . .	66
7.7.2	Монитор кольцевой буфер . . . . .	67
7.7.3	Монитор Хоара . . . . .	69
7.8	Проблема спящего парикмахера . . . . .	71
8	Средство межпроцессного взаимодействия Unix . . . . .	75
8.1	Сигналы . . . . .	75
8.2	Семафоры . . . . .	76
8.3	PIPE Программные каналы . . . . .	78
8.4	Файлы, отображаемые в памяти . . . . .	79
8.5	Программные каналы . . . . .	80
8.6	Сегменты разделяемой памяти . . . . .	80
8.7	Очереди сообщений . . . . .	82
9	Синхронизация в распределенных системах . . . . .	84
9.1	Алгоритмы взаимного исключения . . . . .	87
9.1.1	Централизованный алгоритм (забияки) . . . . .	87

9.1.2	Распределенный алгоритм . . . . .	88
9.1.3	Алгоритм Token ring . . . . .	88
9.2	RPC (вызов удаленных процедур) . . . . .	90
9.3	Неделимые транзакции . . . . .	91
9.4	Тупики . . . . .	92
10	Архитектура вычислительных систем с точки зрения ядра . . . . .	105
10.1	Монолитное ядро . . . . .	106
10.2	Микроядерная архитектура . . . . .	109
10.2.1	ОС Mach . . . . .	111
11	Операционные системы реального времени . . . . .	112
12	Прерывания . . . . .	115
13	ОС реального времени . . . . .	123
14	Точные и неточные прерывания . . . . .	127
	Заключение . . . . .	129
	Список использованных источников . . . . .	130

План занятий:

а) 1 семестр:

- 1) управление процессорами;
- 2) управление памятью;
- 3) взаимодействие процессов.

б) 2 семестр:

- 1) управление данными(файловые системы, доступ к данным);
- 2) управление устройствами;

## 1 История вычислительной техники

Выч. машина – программно-управляемое устройство. Без человека – автоматическое управление. С человеком – арифмометр ????. Реле подтолкнуло к двоичным вычислениям. Марк 1 – электромагнитное реле 1944 года Тьюринга. Фон Нейман – первое серийное.

### 1.1 Первое поколение 1944 – 1955

Сформулированы принципы архитектуры Фон Неймана:

- а) Основные блоки: блок управления, АЛУ, запоминающее устройство и устройство ввода – вывода;
- б) Программы и данные хранятся в одной и той же памяти, таким образом концепция хранимой программы является основной;
- в) Устройство управления и АЛУ обычно объединяются в центральный процессор и определяют действия, подлежащие выполнению путем считывания команд из оперативной памяти. Следует что программа состоит из команд, которые проверяются одна за другой;
- г) Адрес очередной ячейки памяти, из которой следует брать команду указывается счетчиком команд в устройстве управления. Следует, что данные, с которыми работает программа, могут включать в себя переменные: области памяти могут быть поименованы, так что в заполненных в них значениях можно обращаться к ним и менять по присвоенным им именам.

Следствия:

- а) ОЗУ по быстродействию должно быть сопоставимо с быстродействием процессора.
- б) Программа в памяти располагается команда за командой. Команды программы располагаются в последовательных адресах. Для обращения к очередной коман-

ды мы используем счетчик команд (убрали адрес следующей команды, появление условных переходов).

в) В машине сигналы трех типов: данные (команды и собственно данные), адреса, сигналы управления.

## 1.2 Второе поколение (транзисторная логика).

Меняются элементы памяти (магнитные сердечники). Магнитные ленты – средство хранения данных. Магнитные барабаны Серия – полная документация на машину. Что бы что-то запустить в серию должны быть созданы чертежи железа. Ассемблер – мнемоническое оформление машинных кодов.

## 1.3 Третье поколение. 60-е

Появление Интегральных микросхем. 1 шаг – загрузить программу в оперативную память. Чтобы уменьшить времяостоя процессора нужно загрузить в память несколько программ (мультипрограммный режим пакетной обработки) программы в памяти находятся целиком и выполняются от начала до конца или до ошибки. Операционные системы заменили человека оператора, который должен был загружать в память программы с соответствующими программами и библиотеками. Переключение с одной программы на другую осуществляется автоматически операционной системой. Но процессор простояивает в ожидании ввода вывода данных.

Операционная система – комплекс программ для управления процессами и выделения процессам ресурсов вычислительной машины. Процесс – программа в стадии выполнения. Программа на диске – файл.

Программы бывают:

- а) исходник (написан на языке высокого уровня);
- б) объектный (получается трансляторами, трансляторы многопроходные и оптимизирующие),
- в) экзешники (исполняемые программы получаются линковщиками, они подключают файлы библиотек). Исполняемая программа может выполняться только средствами операционной системы.

К процессору организуется очередь процессов. Прежде чем процесс сможет начать выполняться ему должно быть выделен ресурс – память.

### 1.3.1 Канальная архитектура

Семейство IBM/360: Возникла идея распараллизования функций. Появляется архитектура выч. машин. В состав ibm360 включаются ??? название канала. Такая архитектура называется канальной. Канал – программно-управляемое устрой-

ство, получающее от процессора канальную программу, под управлением которой канал берет на себя управление внешним устройством. Т.е. внешним устройством управляет не ЦП, а канал. Процессор инициирует управление внешним устройством. Непосредственное управление осуществляется каналом. Процессор может переключаться на другую программу, пока не получит управляющий сигнал, что операция ввода-вывода завершена. Появилась система прерываний.

Каналы это специальные процессоры. Процессор может переключаться на другую работу, пока канал работает с внешним устройством. Когда канал заканчивает работу – вызывается аппаратное прерывание.

Русские ЭВМ: БЭСМ большая электронная счетная машина. БЭСМ 6 – 1милл вычислений в сек. Архитектура основана на принципе распараллеливания. Всё своё. В это время в Америке интенсивное развитие. Решили перейти на единую серию и все силы брошены на копирование американцев. В результате созданы свои схемы, запущено производство интегральных схем, выход годных хуже в 100 раз.

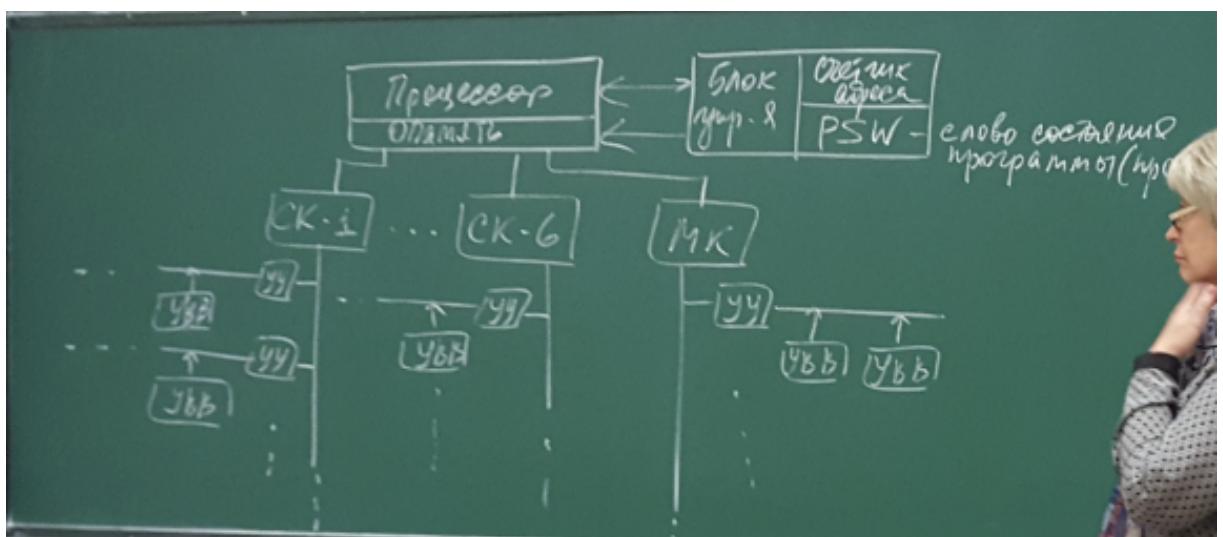


Рисунок 1.1 – Архитектура ЕС ЭВМ

1.1 Отражает канальную архитектуру, передиали с ИБМ 360 – первая линейка вычислительных машин.

Два вида каналов:

- селекторный – в каждый момент времени каждый канал работает с единственным устройством ввода – вывода. Работают с быстрыми устройствами.;
- мультиплексорный - может работать с несколькими устройствами ввода вывода. Работают с медленными устройствами.

Чтобы ускорить переключение с одного задания на другое стали загружать в оперативку большое число программ. Чтобы иметь возможность управлять набором программ был создан язык управления заданиями. В результате любая программа предварялась командами на языке управления заданиями. Указывался транслятор, максимальный необходимый объем ОЗУ и время выполнения программы. На основе этих данных система решала порядок выполнения программ в рабочей смеси. Это пакетный режим. С элементной базой и с архитектурой развивались внешние устройства.

Появились электронно-лучевых мониторы. Были на запоминающих трубках. Появились терминалы. Теперь ЦП должна обеспечивать человеку комфортную среду работы (не должен ждать неопределенное время). Система должна обеспечить гарантированное время. Называли это - системы разделения времени путем квантования процессорного времени. Каждой программе выделяется фиксированный квант процессора.

#### **1.4 4 поколение. 1970 – н.в.**

Большие и Сверх Большие интегральные схемы.

Сеймур Крей создал первый суперкомпьютер Крей 1 с первым конвейером. Создание intel 8080. Затем 16 разрядный процессор + 20 разрядная шина адреса intel 8088. Поставлена MS-DOS.

Для PDP написана Unix – система разделения времени. К PDP можно подключать большое количество терминалов. Для ускорения написания работы между операционной системой и шеллом разработчики написали СИ для нужд системного программирования. ОС Малтикс – система разделения времени для обучения. PDP 11 поставлена в научные центры Америки с открытым кодом ядра. Мультипрограммные системы пакетной обработки – в памяти большое кол-во программ.

МС-ДОС для компов intel 8088. Однопрограммная ось. Поддерживала перемещаемые сегменты.

Для ibm370 написана VM (операционная система). Позволяла как задачи запускать операционные системы пакетной обработки, разделения времени, реального времени. Одновременно выполнялись несколько операционных систем. Пользователь заказывал ось, какую он хотел.

##### **1.4.0.1 Реальное время**

речь идет о ПО. Реальное время по отношению к осям имеет специфический смысл. Любое устройство работает в реальном времени. Операционные системы реального времени предназначены для управления процессами реального времени. Это процессы, которые обслуживают внешние по отношению к системе устройства. Они называются системами РВ потому что должны обеспечивать определенные временные характеристики работы этих процессов, которые определяются характеристиками внешними устройствами. Никогда про РВ не говорят быстрые или медленные процессы, это чушь. Система должна обрабатывать полученную информацию за время меньшее чем период опроса датчиков. Т.е. управляющее воздействие системы должно быть сформировано за время, определенное характеристиками конкретной подсистемы летательного аппарата. Это РВ. Самолет и его системы работают в реальном времени.

Видео и аудио. Телевизионное изображение – изображение с регенерацией. Для человека характерно инерционность зрения. Глаз сохраняет картинку на 1/24 секунду. Частота смены кадров в телевизоре 25 кадров. Разделены на полукадры. Частота = 50Гц. Идея телевизионной развертки: изображение режется на полоски. Слух более чувствителен к задержкам. В windows ожидание аудио процесс повышает его приоритет на 8, а видео повышение приоритета незначительное.

## 2 Разработка операционных систем

**Операционная система** - комплект программ, которые управляет ресурсами вычислительной системы. ??? (Оксфорд)

**Ресурс** - любой из компонентов вычислительной системы и предоставляемые ею возможности.

**Операционная система** – набор программ, как обычных, так и микропрограмм, которые обеспечивают возможность использования аппаратуры компьютера, при этом аппаратура предоставляет сырую вычислительную мощность. А ось дает ??? и возможность удобного использования (Г. Дейтл).

Ресурсы:

- а) процессорное время;
- б) объем ОЗУ;
- в) внешние устройства;
- г) коды операционной системы (системные вызовы).

#### **2.0.0.1 Реентерабельность**

Коды операционных систем – реентерабельные. Реентерабельность – свойство повторной входимости. Программа повторно входима, если одна и та же копия программного сегмента может использоваться несколькими программами.

Повторная входимость включает в себя 2 аспекта:

- а) командный сегмент не может модифицировать сам себя;
- б) локальные данные каждого пользователя записаны в сегмент данных этого же пользователя.

Реентерабельные процедуры называются процедурами чистого кода. Данные вынесены из кода и находятся в адресном пространстве пользователя или в системных таблицах. Данные (и системные таблицы) являются ресурсом системы. Ресурс системы – железо и софт.

#### **2.0.0.2 Разработка ОС (аспекты подходов)**

Характеристики ОС определяются целями. Если система пакетной обработки, то максимальная эффективность выходной мощности. Если система разделения времени, то время ответа должно быть гарантировано. Реальное время, то управление внешними устройствами без сбоев за определенные интервалы времени.

#### **2.0.0.3 Особенности построения ОС**

- а) Определение абстракций (процессы, потоки, файл, семафоры, сообщения);
- б) Предоставляемые примитивные операции. Для управления структурами нужно предоставлять в распоряжение пользователя примитивные операции (функции нижнего уровня). Такие как создание, уничтожение, изменение. Примитивные операции в виде системных вызовов. ;
- в) Защита:
  - 1) пользователи должны быть защищены друг от друга. Для майнфреймов должны быть защищены процессы (адресное пространство).
  - 2) система должна быть защищена от пользователя

- 3) установка прав доступа файлов.
  - 4) обеспечение совместного использования данных и ресурсов системы.
  - 5) изоляция отказов.
- г) Управление аппаратурой.

#### **2.0.0.4 Сложность проектирования ОС**

- а) Большой объем кода. Современные UNIX > 1 млн. кода + 1000 на ассемблере. Windows 2000 около 29млн строк.
- б) Подсистемы ОС взаимодействуют друг с другом.
- в) Параллелизм.
- г) Совместное использование ресурсов системы и отдельных программ.
- д) Высокая степень универсальности.
- е) Хакеры
- ж) Переносимость (возможность работы ПО на разных аппаратных платформах)

#### **2.0.0.5 Концепции виртуальной машины**

ОС работающая на определенной конфигурации аппаратной части предоставляет в распоряжение пользователя некоторую виртуальную машину. Скорость работы зависит от конфигурации.

##### **Иерархическая машина [1]**

Таблица 2.1 — Ядро системы

Аппаратура
Управление процессорами (нижний уровень).
Непосредственное выделение кванта времени.
Планирование процессов и управление памятью. (P, V)
Управление процессорами верхнего уровня (создание, уничтожение, взаимодействие при помощи сообщений).
Управление устройствами.
Управление информацией (хранение, уничтожение файлов, файловая система)

**Интерфейс** – функции, которые нижние уровни предоставляют верхним уровням. **Непрозрачный интерфейс** – когда невозможно обратиться через уровни. **Прозрачный интерфейс** – можно обратиться через уровни.

### 3 Процессы

Процесс – единица декомпозиции системы. Ему выделяются ресурсы. В многопроцессной ОС одновременно существует большое количество процессов. Основной ресурс – процессорное время. За время жизни процесс переходит из одного состояния в другое. Состояние – абстракция.



Рисунок 3.1 – Самая общая диаграмма состояний процесса

Процесс в начале идентифицируется. В системе есть одна на систему таблицы процессов. Любая системная таблица – массив структур. Процесс в системе описывается структурой proc. Каждому запущенному процессу выделяется структура (дескриптор) процесса (строка в таблице процессов). Номер строки – это id процесса, целое число.

На 3.1 ребро 1 – выделяется первый необходимый ресурс – объем ОЗУ, в соответствии с концепцией хранимой программы и переходит в состояние готовности.

В состоянии готовности находится большое количество процессов. Они составляют очередь. В очереди процессы располагаются с приоритетами. Процессорное время получает самый приоритетный. Если в очередь поступил наиболее приоритетный процесс, чем тот, какой выполняется сейчас, то он вытесняется. Это реализуется в системах пакетной обработки и реального времени.

В разделении времени ???

В системах квантования при истечении кванта, процесс возвращается в очередь. При вытеснении происходит переключение контекста. Если процесс вытеснен или исчерпан квант, то необходимо сохранить информацию, которая поможет продолжению выполнения. **Полный контекст** – информация о выделенных процессу ресурсов + аппаратный контекст. ???

**Уровни наблюдения :**

- a) последовательное выполнение.
- б) квазипараллельное выполнение. (с точки зрения пользователя – параллельно)
- в) реальная параллельность.

**Планирование и диспетчеризация :**

- a) безприоритетное
- б) приоритетное

**Приоритеты :**

- a) абсолютные и относительные;
- б) статические и динамические.

**Планирование :**

- a) с переключением и без;
- б) с выталкиванием и без.

Процесс – единица декомпозиции, ему выделяются ресурсы. Он является владельцем ресурсов.

### 3.0.1 Виды ОС

- а) однопрограммные пакетной обработки
- б) мультипрограммной пакетной обработки. (планирование на принципе распараллеливания функций. Программа выполняется до тех пор, пока не запросит доп. Ресурс системы или пока не придет более высокоприоритетный процесс, в случае поддержки вытеснения, может вытеснить)
- в) системы разделения времени, система квантования. (каждому выделяется квант процессорного времени. Процесс выполняется до тех пор, пока не истек квант, не начался процесс ввода/вывода или не вытеснен другим высокоприоритетным процессом.)

### 3.0.2 Алгоритмы планирования для систем пакетной обработки

- а) FIFO. Не учитываются приоритеты, нет вытеснения.
- б) SJF. Короткие задания получают более высокий приоритет. Как результат – бесконечное откладывание.
- в) HRN – наибольшее относительное время ответа. Приоритет вычисляется по формуле  $P = (tw - ts)/ts$ , где  $ts$  – запрошенное время обслуживания.  $tw$  – время ожидания.
- г) SRT наименьшее оставшееся время. Процесс может быть вытеснен, если в очередь готовых процессов поступит процесс, с меньшим оценочным временем откладывания. Тут процессы вытесняются.

### 3.0.3 Алгоритмы планирования для интерактивных систем

- а) RR – процессы формируют очередь, но при истечении кванта они опять попадают в очередь.
- б) Многоуровневые очереди. В системе процессы организуются в несколько очередей, наивысший приоритет имеет первая очередь. В неё попадают вновь созданные процессы и процессы, завершившие ожидание ввода вывода. Если процесс не успел завершиться или попросить ввод/вывод, он поступает в следующей очереди. В конце – 4 очередь RR, в котором крутится холостой процесс.
- в) адаптивное ?? планирование - учитывает объем занимаемое процессом физической памяти. Дополнительно оценивается объем памяти занимаемое процессом, а также учитывается, может ли процесс получить необходимое для дальнейшего выполнения объем физической памяти. Только в этом случае процесс получит квант

процессорного времени. Если процесс не может получить объем физической памяти, то он может быть приостановлен до благоприятной ситуации.

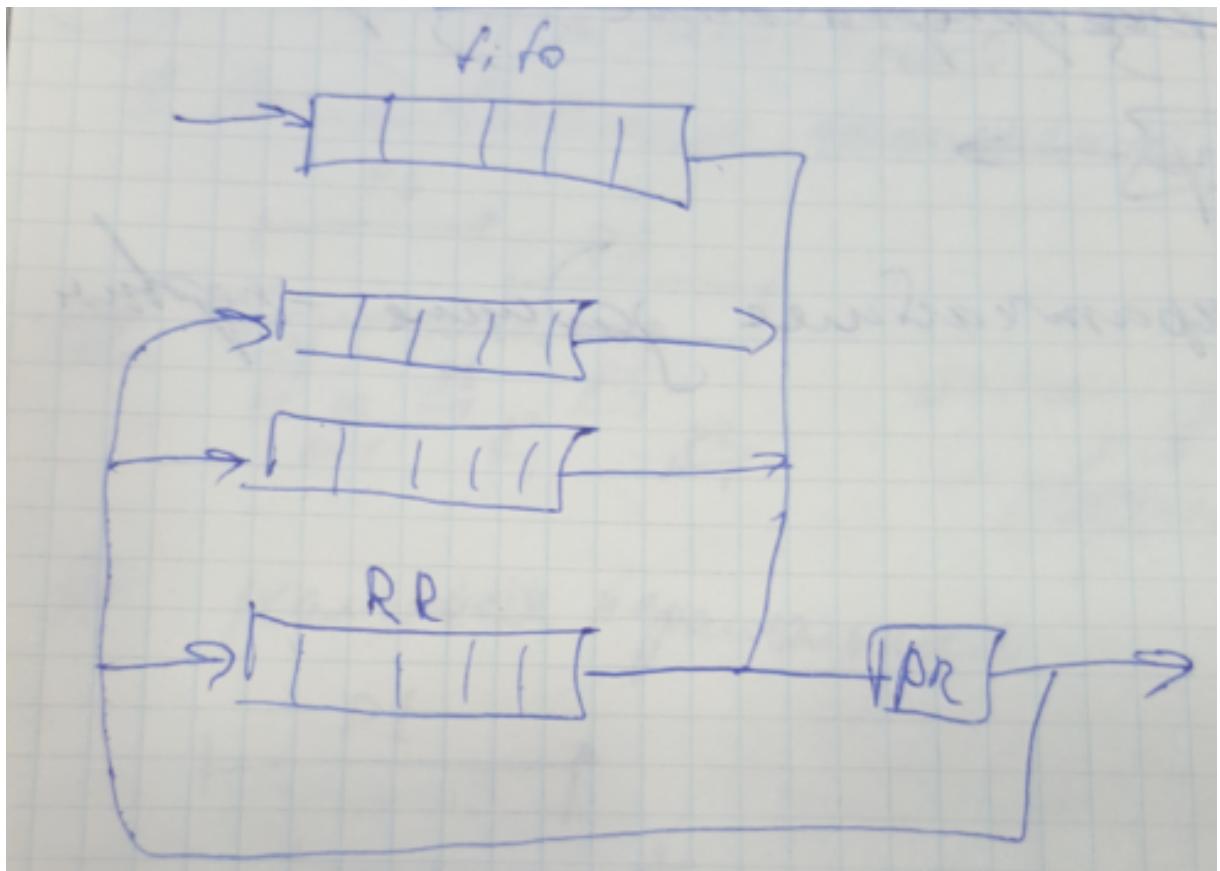


Рисунок 3.2 — Многоуровневые очереди

### 3.1 Создание нового процесса

`fork()` – создание нового процесса (порождение процесса). Процесс должен быть идентифицирован. В системе – одна главная таблица процессов. Поля – обычно указатели на другие структуры. После этого может быть выделена память. Память в системе может быть выделена той частью системы, которая называется менеджером памяти. Он должен решить сколько и где выделить памяти. Если памяти недостаточно в данный момент, то процесс будет переведен в состояние «готов к запуску, выгружен», иначе «готов к запуску, в памяти».

Процесс часть времени выполняется в режиме пользователя и выполняет свой код, а часть времени – в режиме ядра и тогда выполняет реентерабельный код ОС. В процессе выполнения процесс запрашивает дополнительные ресурсы с помощью системных вызовов. Выполнив системный вызов процесс будет переведен в режим ядра. Системный вызов – программное прерывание. В режим ядра процесс будет возвращен, когда он обратиться к коду, который отсутствует в физической. В режиме ядра будет решено, блокировать процесс или ???.



Рисунок 3.3 — pic

### 3.1.1 События, переводящие систему в режим ядра (система прерывания)

- a) Системные вызовы или программные прерывания. Выполняемая программа пользователя нуждается в ?? и выполняет системный вызов. АПИ – набор функций, которые предоставляет система. Ни одна система не позволяет процессам на прямую выполнять ввод/вывод, только через системный вызов.
- б) Исключения (устранимые, не устранимые). Системные вызовы и исключения – синхронные события по отношению к вашему процессу. Возникают в процессе выполнения программного кода. Исключения возникают в результате ошибок в программе (не устранимым, например - деление на ноль). Устранимое – страничное прерывание, необходимая страница будет загружена в память и программа продолжит выполнение.
- в) Аппаратные прерывания. Сугубо асинхронные события в системе. Также имеют различный характер в системе. Самое массовое – прерывания от внешних устройств. Второй тип – прерывание от системного таймера, которое выполняется по тику(выделенный импульс) 18,3 раз в секунду. Третий тип – прерывание от действий оператора (ctrl alt del , ctrl + C = завершение процесса в unix)

Когда выполняется запрос на ввод вывод, в режиме ядра вызывается драйвер, он формирует запрос, который посыпается ????. По завершении операции ввода/вывода контроллер посыпает сигнал прерывания в контроллер прерывания и он формирует сигнал прерывания для процессора.

## 4 Потоки

Сохранение аппаратного контекста выполняется аппаратно (pusha). Процесс обладает выделенными ему ресурсами, которые описываются таблицами. При переходе к другому процессу происходит переключение полного контекста. Это долго, поэтому появилась идея потоков.

**Виды потоков :**

- a) Потоки ядра;
- б) Легковесные процессы;
- в) Пользовательские нити. (выполняются библиотеками).

Поток – непрерывная часть кода, выполняющаяся параллельно с другими потоками кода. Поток не имеет собственного адресного пространства. Владельцем ресурсов является процесс. Поток владеет аппаратным контекстом и счетчиком команд.

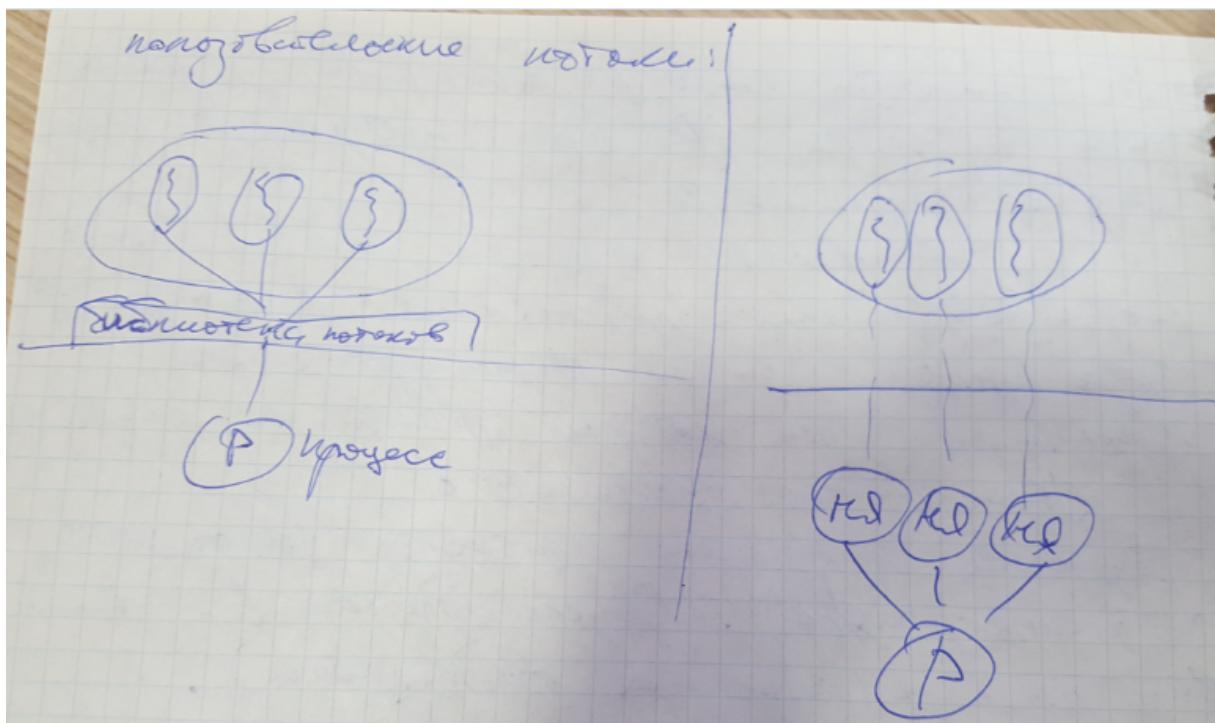


Рисунок 4.1 — Виды потоков

Переключение полного контекста – долго. Аппаратного – поддерживается на аппаратном уровне. ??? Устройству выделяются и другие ресурсы. Семафоры, ??? Вся эта информация составляет полный контекст. Она фиксируется в структурах. На них ссылается основная структура, описывающая процесс. В Unix это struct proc. Ставяясь сократить временные расходы в прошлом веке была введена концепция потоков. Поток – непрерывная часть кода, грубо говоря: кусок кода, такие части кода могут выполняться параллельно. Чтобы они могли выполняться параллельно их оформляют в виде потоков. В Windows – каждый запущенный драйвер – это поток.

### **Виды потоков :**

- а) Потоки ядра;
- б) Легковесные процессы. Потоки уровня ядра. Владельцем ресурсов является процесс, а потоку принадлежит счетчик команд и аппаратный контекст. При этом надо сказать, что выигрыш если переключаемся между потоками одного процесса, но если к потоку другого процесса, то происходит переключение полного контекста;
- в) Пользовательские нити. Для управления необходима специальная библиотека потоков. Должна предоставлять функции планирования потоков, диспетчериизации, сохранения контекста. Как только поток запросил функции ядра, то процесс приостанавливается, пока не завершиться операция ядра (например: ввода – вывода).

Поток выполняется в адресном пространстве процесса. Таблицы, описывающие память, принадлежат процессу. Адресное пространство процесса описывается соответствующими таблицами. Для взаимодействия потоков можно использовать глобальные переменные. Процессы имеют защищенные адресные пространства, в том смысле что ни один процесс не может обратиться за собственное адресное пространства.

Таблица 4.1 – Однопоточная модель процесса.

Блок управления процессом
Стек ядра
Стек пользователя
Адресное пространство пользователя

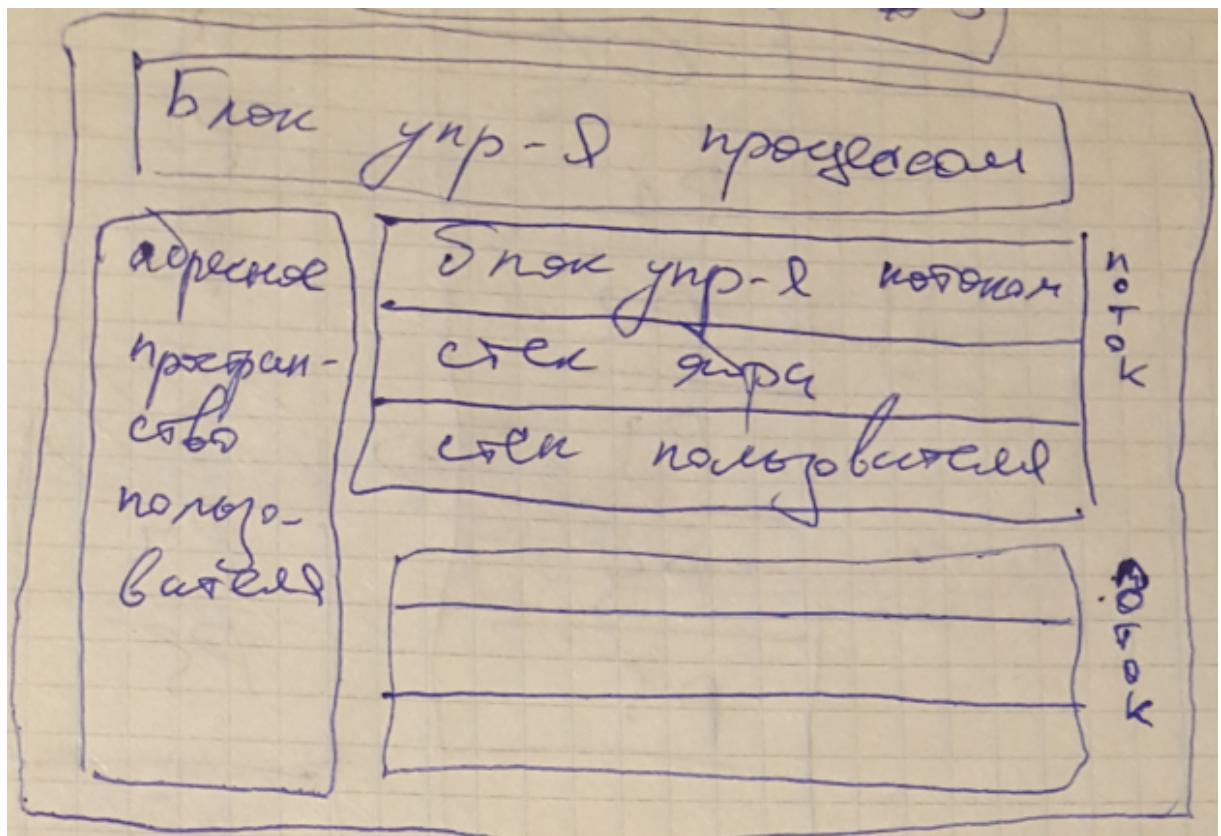


Рисунок 4.2 — Многопоточная модель процесса

Поток представляется системе дескриптором (блок управления потоком). У каждого потока есть стек ядра. Чтобы сохранять контекст (системный контекст) используется стек ядра. Нельзя использовать стек пользователя, потому что пользователь может создать несколько стеков, назначать им размеры. [2]

## 5 Процессы ОС Unix

### 5.1 Системный вызов fork()

. Создает новый процесс. В Unix все процессы создаются единообразно вызовом `fork()`. Любой процесс может создать любое кол-во процессов. При этом процессы в Unix находятся в отношении предок – потомок. В результате создается иерархия процессов. `fork()` является базовым примитивом. В ANSI C не хорошо писать большими буквами, так как много предопределенных констант написаны с больших букв.

```
1 int main {
2     int childpid, parentpid;
3     if ((childpid = fork()) == -1) {
4         perror("Can't fork");
5         exit(1);
6     } else if (childpid == 0) {
7         /* процесс потомок */
8         printf("child: childpid = %d, parentpid = %d\n", getpid(), getppid());
9         exit(0);
10    } else {
11        /*родитель*/
12        printf("parent:childpid=%d, pid=%d\n", childpid, getpid());
13        exit(0);
14    }
15 }
16 }
```

Рисунок 5.1 — listing

В результате `fork()` создается копия процесса предка, в том смысле, что процесс-потомок наследует код предка, дескрипторы открытых файлов, сигнальную маску.

Замечание 1. В Unix всё файл, внешние устройства тоже файлов.

В старых системах Unix (для pdp 11) код предка копировался в адресное пространство потомка. В результате в системе могло существовать одновременно большое количество копий одной и той же программы. Память используется неэффективно. В современных системах используется умный `fork`, который реализуется следующим образом: для процесса – потомка создаются собственные карты трансляции адресов (таблица страниц), которые указывают на страницы адресного пространства предка, т.е. в таблицы страниц потомка находятся дескрипторы страниц адресного пространства предка (адреса страниц адресного пространства предка). `Pointer` – указатель (не типизированный). `Reference` – ссылка (типованный). Процесс-потомок получает собственные карты трансляции адресов, и эти таблицы страниц указывают на страницы адресного пространства предка. Но процессы имеют особенность обращаться к собственные страницы, и писать в свои страницы сегменты данных или стека. Процесс не заинтересован в тех изменения другого процесса. Решено штукой `copy on write`. Для страниц предка (сегмента данных и стека) меняется права доступа на «только для чтения» и устанавливается флаг `copy on write`. Флаг `copy on write`. Если или предок, или потомок пытаются изменить страницу, то возникнет исключение по правам доступа. Обрабатывая это исключение, супервизор (ОС в стадии выполнения) обнаружит установленный флаг `copy on write` и создаст копию страницы в адресном пространстве того процесса, который пытался её изменить. В результате в системе создается необходимое количество копий (отдельных страниц). Это будет действовать до тех пор, пока процесс потомок не вызовет или системный вызов `exec()` или системный вызов `exit()`. Если потомок сделал эти системные вы-

зовы, то страницы предка получают обычные права (read / write) и флаг copy on write сбрасывается.

## 5.2 Системный вызов exec()

. В Unix предлагается системный вызов exec(), который переводит процесс на новое адресное пространство программы, которая указана в системном вызове `exec()`, `execl()`, `execv()`, `execle("/bin/ls", "ls", "-l", 0);`, где `/bin/ls` - исполняемая программа, которая выводит на экран список файлов

Системный вызов `exec()` создает низкоуровневый процесс. Не создают полноценный процесс, но выполняют действие, которое связано с созданием нового процесса. Создание нового адресного пространства. Системный вызов `exec()` создает карты трансляции адресов для адресного пространства программы, которая передана в системному вызову `exec()` в качестве параметра. `exec()` создает новые таблицы страниц, которая ссылается на новую программу в параметрах `exec()`. После этого `exec()` заменяет адресное пространство потомка на вновь созданное адресного пространств. Идет замена одной таблицы на другую выполняется простой замены адреса. В Intel есть регистр `CR3`, который содержит начальный адрес таблиц страниц. При переключении процессов меняются адреса в регистрах процессора. `exec()` создает новое адресное пространство, это значит его описать, создать для этого адресного пространства соответствующие таблицы. Переход на новое адресное пространство осуществляется простой заменой адреса. Мы теряем указатель на эту таблицу и она превращается в мусор.

### 5.3 Иерархия процессов.

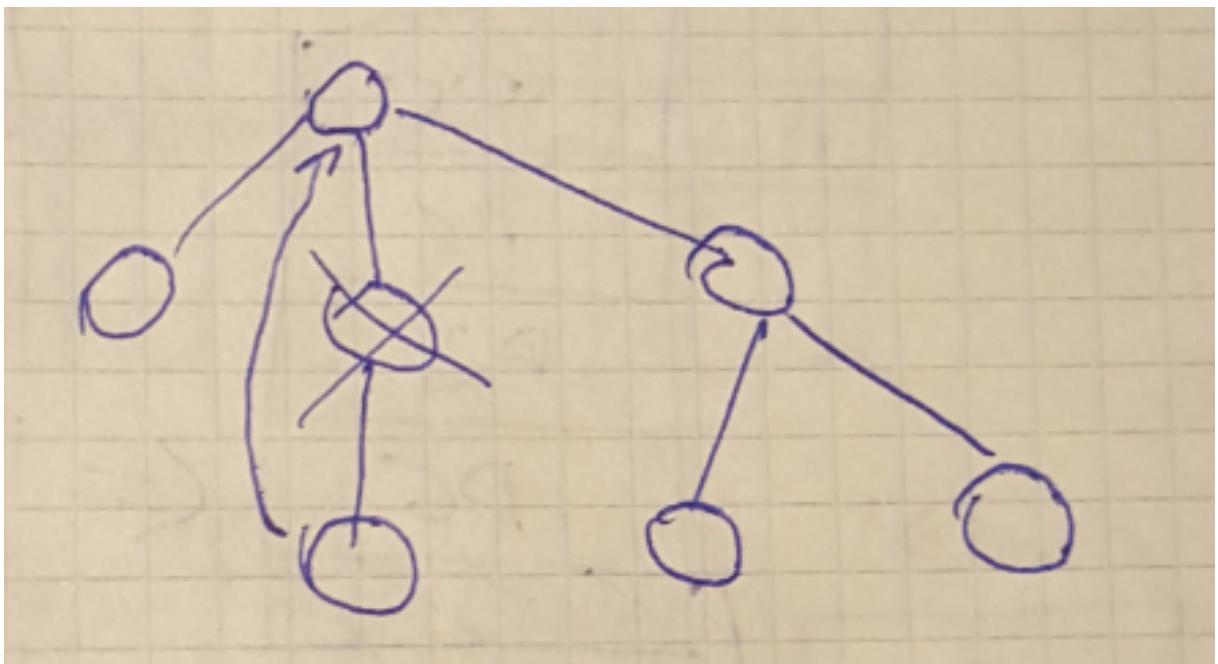


Рисунок 5.2 — Иерархия процессов

В результате `fork()` предок получает id созданных потомков. Смотрим в `struct proc`, там есть указатель на самого последнего созданного потомка. Потомков может быть создано любое кол-во, поэтому это – связный список. В результате `fork()` создается иерархия процессов, которые связаны отношением предок – потомок. Процесс предок может завершиться раньше потомка, если не предпринять специальных действий, то иерархия процессов разрушится. Процесс, у которого завершился предок, называется **сиротой**. Система предпринимает ряд действий для сохранения иерархии процессов.

Unix поддерживает понятие терминал программно. И Unix и Windows являются системами (реального ???) времени, это когда к компу подключены терминалы. Мы можем запустить любое количество терминалов. В Windows это консоль. Когда запускаем систему, то запускается процесс init с id = 0. Когда открываем терминал, то создается процесс ??. Терминальный процесс (id = 1) является предком всех процессов, запущенных в данном терминале.

Код в 5.1 чреват тем, что предок может завершиться раньше потомков. При завершении любого процесса система проверяет, не остались ли у него незавершенные потомки, если такие процессы – потомки у него остались, то выполняются действия по усыновлению осиротевших процессов терминальным процессом. Для этого модифицируются соответствующие дескрипторы. Процесс потомок в parrantid получает id терминального процесса, а терминальный процесс получает указатель на осиротевший процесс. При завершении потомков процессы-предки получают стату-

сы завершения своих потомков. Если произошло усыновление, то статус завершения получит терминальный процесс.

#### 5.4 Системный вызов wait()

Переводит процесс в состояние ожидание. Предназначен для перевода процесса-предка в состояние ожидания завершения своих потомков.

Если в 5.1 в родителя поставить wait(&status), то предок будет ждать завершение потомков. Процессы зомби – специально введённое состояние процесса. Все процессы проходят это состояние.

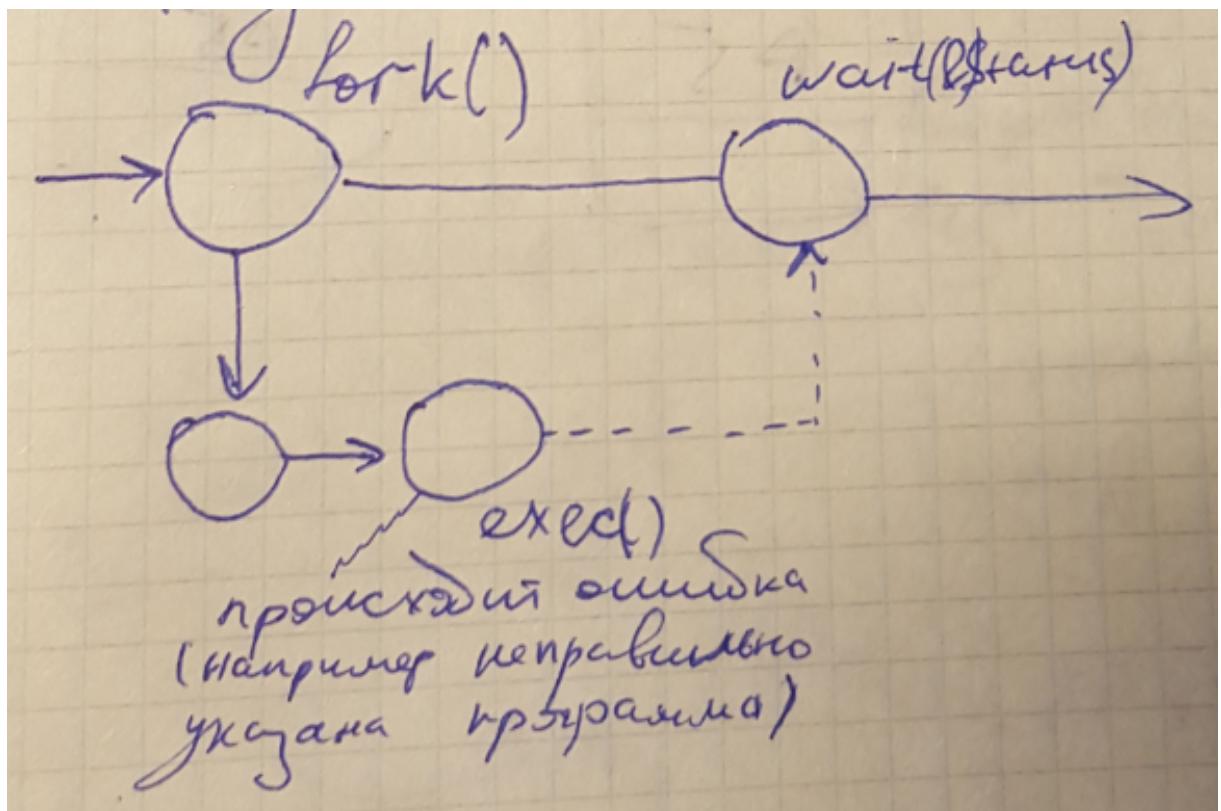


Рисунок 5.3 — Бесконечная блокировка на wait

Процесс предок бесконечно блокирован на wait, если потомок уже завершился, до вызова wait.

Процесс при завершении получает статус зомби, в системе помечается буквой Z, до момента, пока процесс не вызовет системный вызов wait. Зомби – процесс, у которого отобраны все ресурсы, кроме строки в таблице процессов (дескриптор).

#### 5.5 Группы процессов (сигналы)

В Unix процессы объединяются в группы. Главная группа – терминальная. Любой предок, вызвавший потомков, создает группу процессов. Объединение в группы для того, чтобы процессы одной группы могли получать одни и те же сигналы. В Unix – сигналы, в Windows – события. Механизм сигналов в Unix позволяет процессам

реагировать на события, которые могут произойти или внутри самого процесса или вне его. Сигнал – базовое средство информирования процессов о событиях в системе. Самым важным событием в системе является завершение процесса. Получение процессом сигнала указывает ему на необходимость завершиться. Вместе с тем, реакция процесса на принимаемый сигнал зависит от того, как сам процесс определил свою реакцию на данный сигнал. Для этого процесс должен содержать свой обработчик сигнала, если процесс желает изменить реакцию на получаемый сигнал, он должен определить эту реакцию с помощью своего обработчика сигнала. В классических системах сигналов не может быть больше 20. Для удобства каждый сигнал имеет цифровой идентификатор и буквенный. Все значения сигналов хранятся в файле `signal.h`. Сигналы относятся к IPC. Рассматриваем для того, чтобы закруглить отношение о процессах. Если затрагиваем группы процессов, то нужно затронуть сигналы.

```

1 #define NSIG 20
2 #define SIGHUP 1 //разрыв связи с терминалом
3 #define SIGINT 2 //завершение процесса по нажатию ctrl+c
4 #define SIGQUIT 3
5 #define SIGILL 4
6 #define SIGTRAP 5
7 #define SIGIOT 6
8 #define SIGEMT 7
9 #define SIGFPE 8
10 #define SIGKILL 9 //уничтожение процесса
11 #define SIGBUS 10
12 #define SIGSEGV 11 //нарушение границы сегменты, выход за пределы
сегмента
13 #define SIGSYS 12
14 #define SIGPIPE 13 //запись в канал, чтения нет
15 #define SIGALARM 14 //будильник, прерывание от таймера
16 #define SIGTERM 15 //программное прерывание, когда kill выполняется в
режиме командной строки
17 #define SIGUSR1 16 //на эти сигналы может повеситься пользователь
18 #define SIGUSR2 17 //на эти сигналы может повеситься пользователь
19 #define SIGCLD 18 //процесс потомок завершился, сигнал от потомка
20 #define SIGPWR 19 //аварийный сигнал по напряжению
21 #define SIGPOOL 22
22 //
23 #define SIG_DFL //все установки по умолчанию
24 #define SIG_IGN //игнорирование сигнала

```

Рисунок 5.4 – Сигналы

В 5.4 №3 это клавиша *QUIT* или *ctrl + /* Средством приема и посылки сигналов является системные вызовы `kill()` и `signal()`.

### 5.5.1 Системный вызов `kill()`

Предписывает уничтожение процесса. `kill(pid, sig)`. Сигнал будет послан всем процессам. Если `pid <= 1`, то сигнал будет послан группе процессов. Если `pid`

равен 0, то sig посыпается каждому процессу, который входит в группу текущего процесса. Если pid = -1, то сигнал будет послан процессам, которых user id такой же, как и у процесса, который вызвал kill.

### 5.5.2 Системный вызов signall()

. идентифицирует сигнал и воспринимает сигнал. Реакции процесса на системный вызов signal с аргументом func будет вызов функции func(). Процесс имеет возможность определить с помощью функции собственную реакцию на получаемый сигнал.

## 6 Управление памятью

Подразумевается управление ОЗУ. В системе имеется иерархия памяти, рассматривается в зависимости от близости к процессору. ПРОЦЕССОР СВОЕЙ ПАМЯТИ НЕ ИМЕЕТ. В кристале процессора три кеша: данных, команд, TLB (прим. ред. специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти.).

Таблица 6.1 — Иерархия памяти

Внешнее ЗУ
ОЗУ
КЭШ 1-го уровня
Процессор

Вертикальное – передача данных с уровня на уровень. Горизонтальное управление – управление ????. Внешним ЗУ управляет файловая система. Файловая система обеспечивает доступ к хранимой информации.

**ОС однопрограммная пакетной обработки** Всё адресное пространство ОЗУ отдано одной единственной программе. Но на самом деле 2 программы: ОС и программа. С помощью регистра границы (куда записывается адрес ОС) ОС защищается от программы.

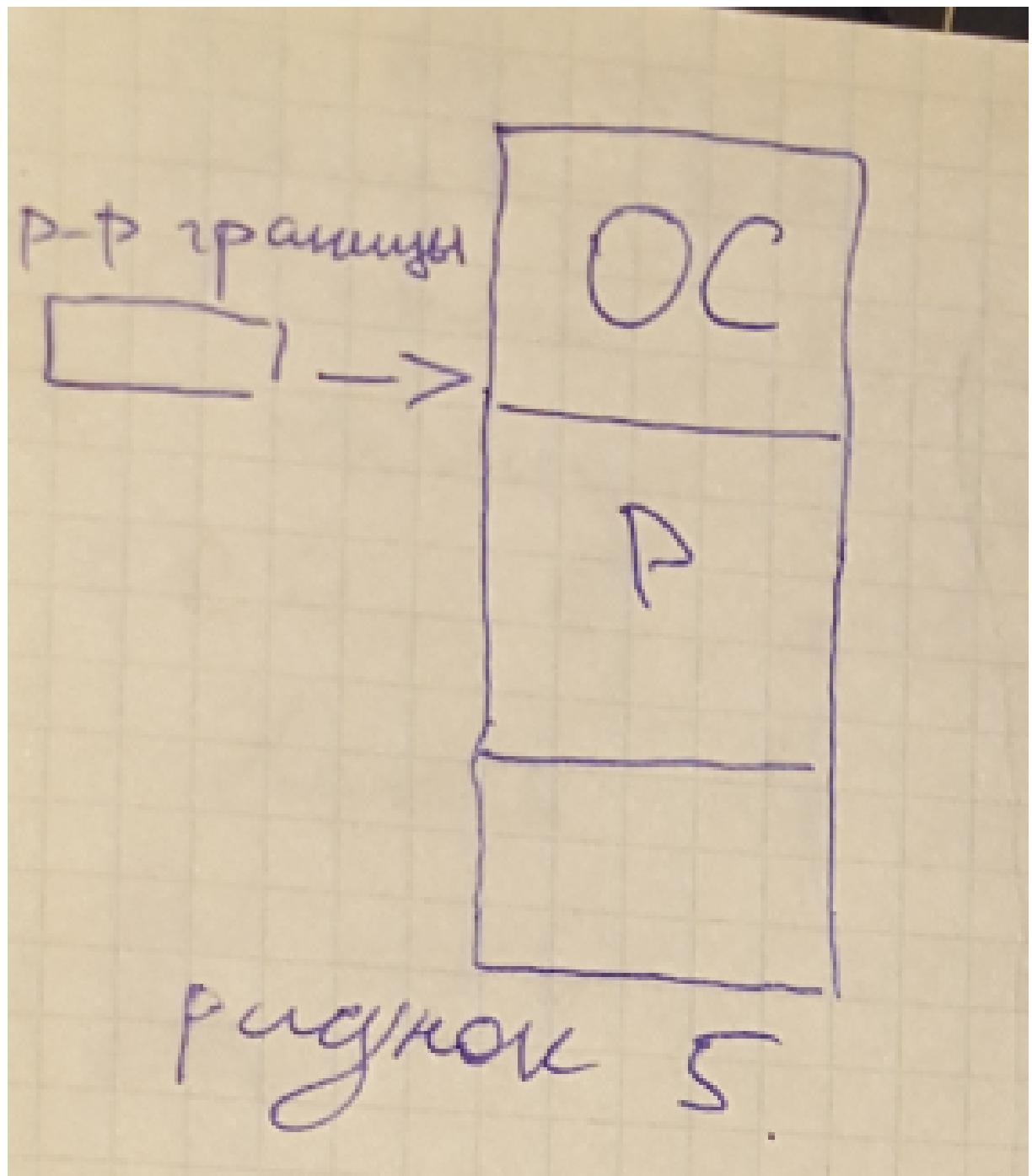


Рисунок 6.1 – ОС однопрограммная пакетной обработки

Идея ДОСа – как можно меньше места в ОЗУ.

Таблица 6.2 – Дос

0Мб	Резидентная часть (не можем выгрузить)
	Резидентные программы
	Собственные сегменты программы
	Динамическая память. Её размер – динамически меняется.
	Транзитная часть (по мере необходимости вызывается в память Сейчас называется не подгружаемая страницы.
1Мб	резидентная часть

Резидент – шпион, постоянно находящийся в чужой стране. DOS – однозадачная ОС, в памяти только одна программа. На самом деле – несколько программ: DOS, резидентные программы, программа. Память делится между ОС и приложением. Приложение не должно обращаться в некоторые важные области памяти. ДОС плохо защищенная ОС. Сегменты – перемещаемые. Одиночное непрерывное распределение памяти.

**Мультипрограммные ОС** Следующий этап развития ОС – мультипрограммные. Основная цель – сократить время переключения с одной программы на другую.

#### **Разделение памяти с фиксированным размером**

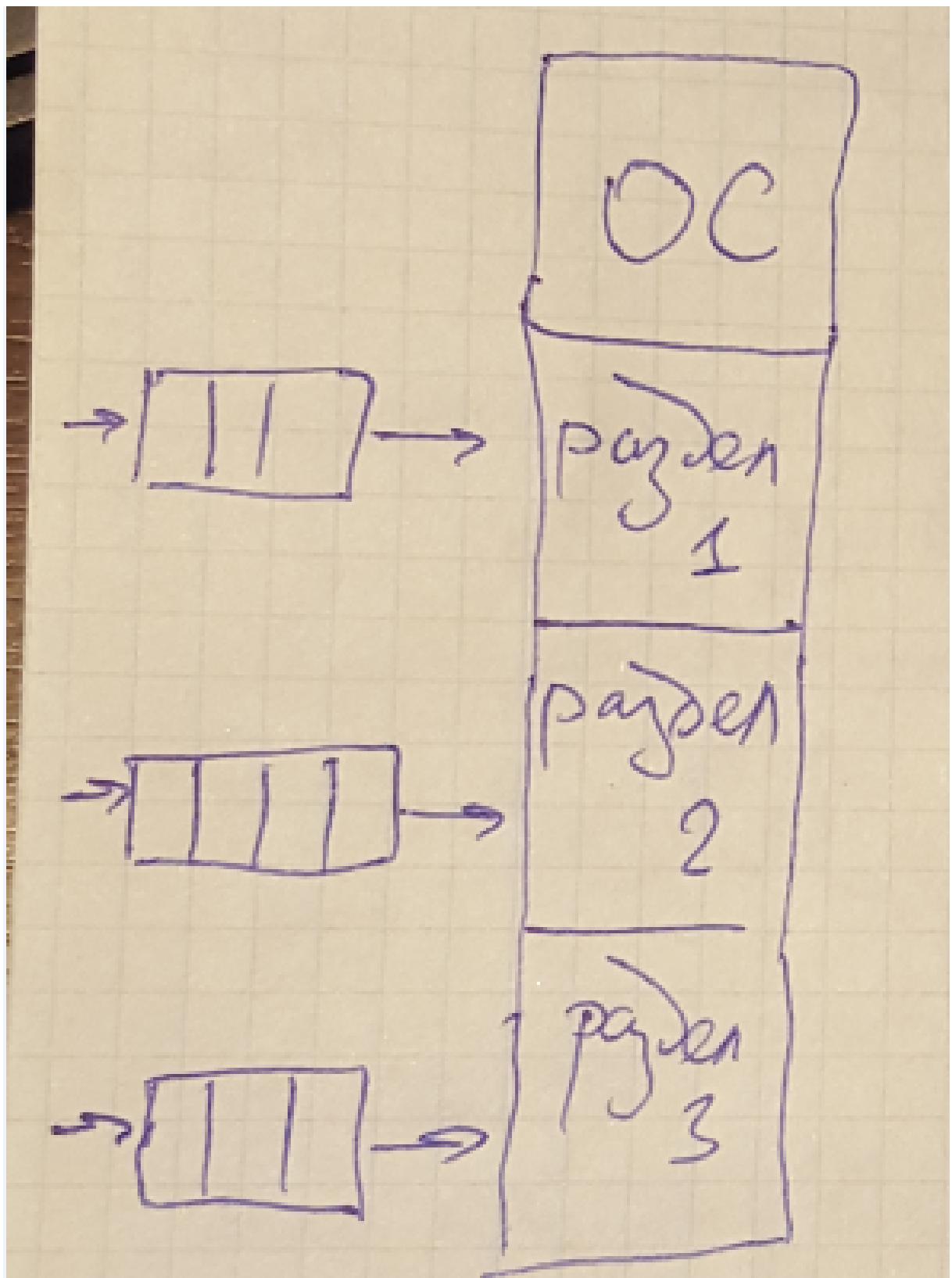


Рисунок 6.2 — разделение памяти с фиксированным размером

**Разделами переменного размера**

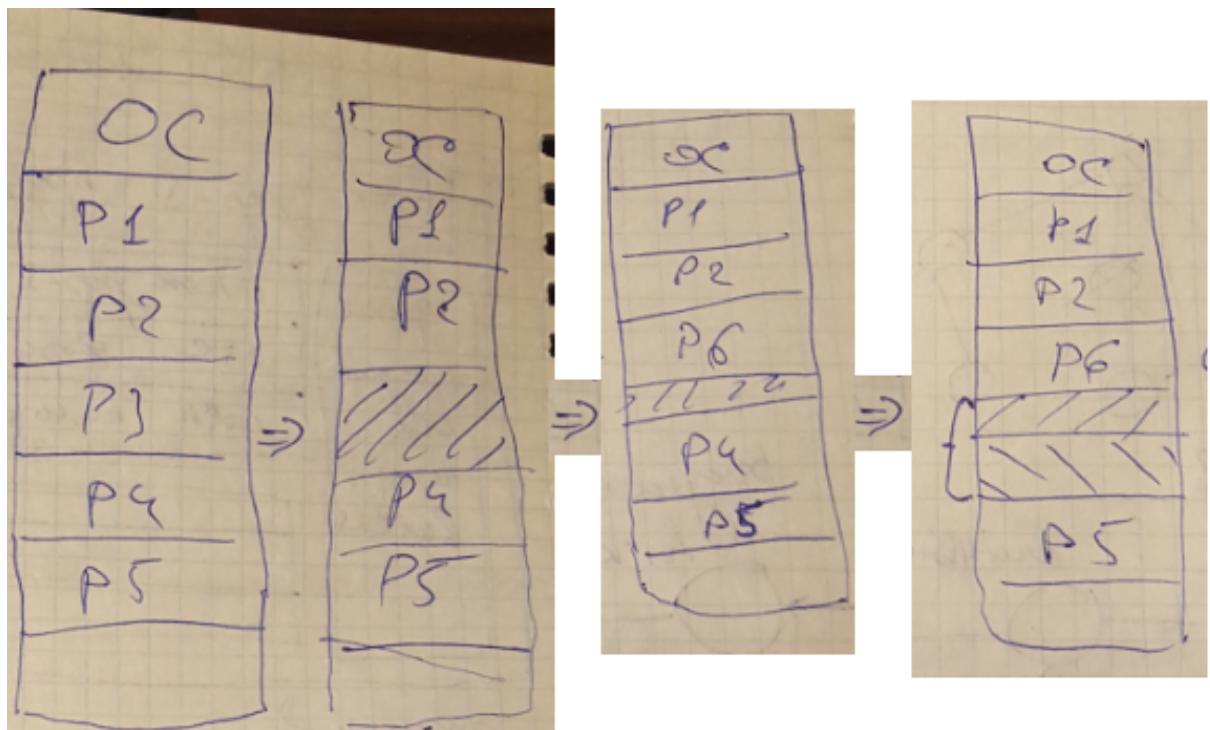


Рисунок 6.3 — разделами переменного размера

В начальный момент мы можем загрузить программы в последовательные адреса. В результате освобождения и загрузки память будет фрагментированной.

Память, как ресурс, характеризуется памятью. Для общности: регистры процессора – как сверхбыстрая память. В кристалле процессора есть кэши. Они содержат актуальную информацию, ту, к которой в последнее время обращался. Затем оперативная память. В силу технологии производства, на порядок отстает в быстродействии. Процессор с памятью связаны локальной шиной. Все делается под управлением тактовых импульсов. Обращение к памяти – медленное действие большого количества тактов. Обращение к памяти получило название – цикл обращения к памяти. Затем внешняя память. Винчестер позволяет реализовывать виртуальную память. Дисковое пространство винчестера используется для организации виртуальной памяти. Связано с процессором соответствующей шиной. Винчестер – механическое устройство, соответственно оно медленнее. Иерархия памяти рассматривается с точки зрения близости к процессору. Флэш память подключается к usb портам, там нет механики.

Вертикальное управление – передача информации с уровня на уровень. Сложная задача, реализуется через управление устройствами.

Одиночное ?? распределение характерно для древних систем. Возникает задача защиты, ОС от программы.

Современные компы – распараллеливание функций.

Мультипроцессные ОС – когда в памяти одновременно большое количество программ.

## 6.1 Управлять памятью

, чтобы выделять память. Нужно иметь инфу о занятых разделах и свободных. Это можно осуществить с помощью двух таблиц.

Таблица 6.3 – ТВР (таблица выделенных размеров)

Номер раздела	Размер	Адрес	Состояние
1	8к	312к	Распределен
2	32к	320к	Распределен
3	-	-	Пустой
4	120к	384к	Распределен

Таблица 6.4 — ТСО (таблица свободных областей)

Номер раздела	Размер	Адрес	Состояние
1	32к	352к	Доступна
2	540к	504к	Доступна
3	-	-	Пустой эл
4	-	-	Пустой эл

## 6.2 Отредактировать таблицы

Любое изменение распределения памяти отображается в соответствующих таблицах.

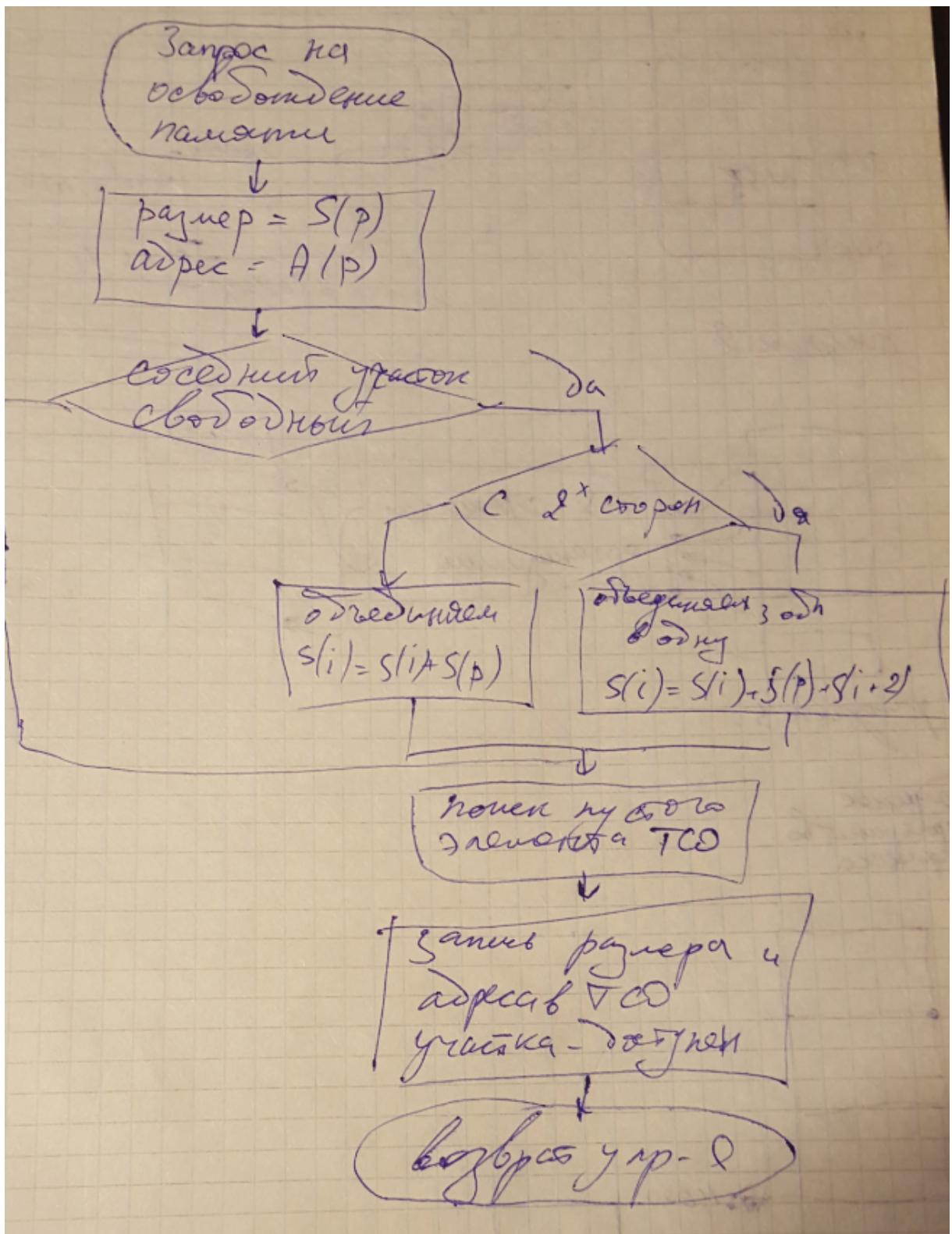


Рисунок 6.4 — Алгоритм объединения участков при освобождении памяти

Если мы постоянно загружаем и выгружаем программы, даже при наличии усилий по объединению соседних участков, в результате мы получим фрагментированную память. До 30% памяти может съесть фрагментация.

### 6.3 Стратегия выделения памяти

- a) первый подходящий по размеру
- б) самый узкий
- в) самый широкий. После загрузки программы у нас останется достаточно пространства, чтобы загрузить еще какую-то программу.

Как только отсортируем, используем продвинутые методы поиска.

### 6.4 Перемещаемые разделы

Если мы хотим перемещать программу в памяти, то нужно ввести логические адреса. Любая программа считает, что она начинается с нулевого адреса. В программе находятся смещения.

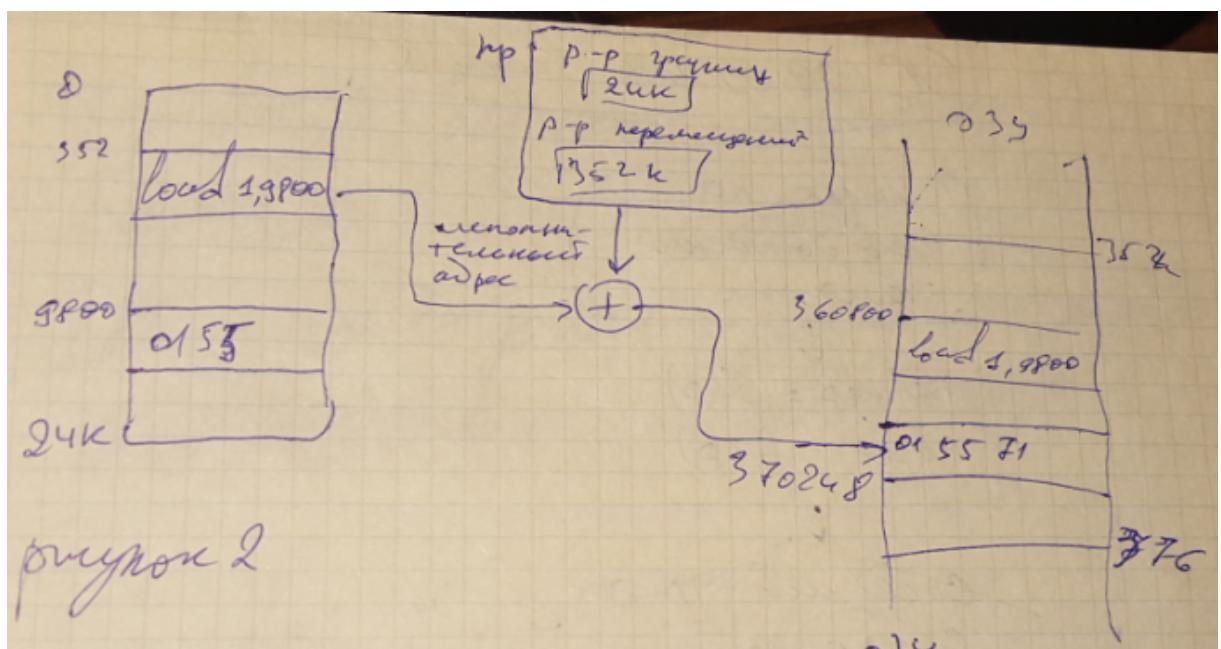


Рисунок 6.5 — ріс

Мы рассмотрели, когда память выделяется программе целиком в последовательные адреса. Такое распределение в памяти называется связанным, когда мы выделяем программе раздел последовательных адресов. Несвязное распределение памяти (либо страницами, либо сегментами). Так мы переходим к виртуальной памяти.

### 6.5 Виртуальная память

(кажущаяся, возможная)

Существует три схемы управления виртуальной памятью:

- а) управление памятью страницами по запросам;
- б) управление памятью сегментами по запросам;

в) управление памятью сегментами, деленные на страницы по запросам.

По запросам – загрузка в память осуществляется по запросу. Запрос возникает, когда программа обращается к команде или данным, отсутствующей в памяти.

Процесс начинает выполняться и в результате выполнения обращается к команде, отсутствующей в памяти, возникает страничное прерывание. Система загрузит эту страницу в память. А где же она? Дисковое адресное пространство в системах общего назначения делится на 2 части. Большая часть отведена для хранения файлов и управляется файловой системой. Меньшая часть отведена для свопинга (область свопинга (пейджинга) в зависимости от того, с чем мы работаем сегменты(страницы)). Виртуальную память можно представить, как пролонгированную на адресное пространство диска. В результате формируется виртуальное адресное пространство процесса. Каждый процесс, когда он создается, для него создается виртуальное адресное пространство. Это адресное пространство в системе описывается соответствующими таблицами. В Unix называются картами трансляции адресов (страницы сегментов или страниц).

### 6.5.1 Управление памятью страницами по запросам

Адресное пространство процесса делится на страницы.

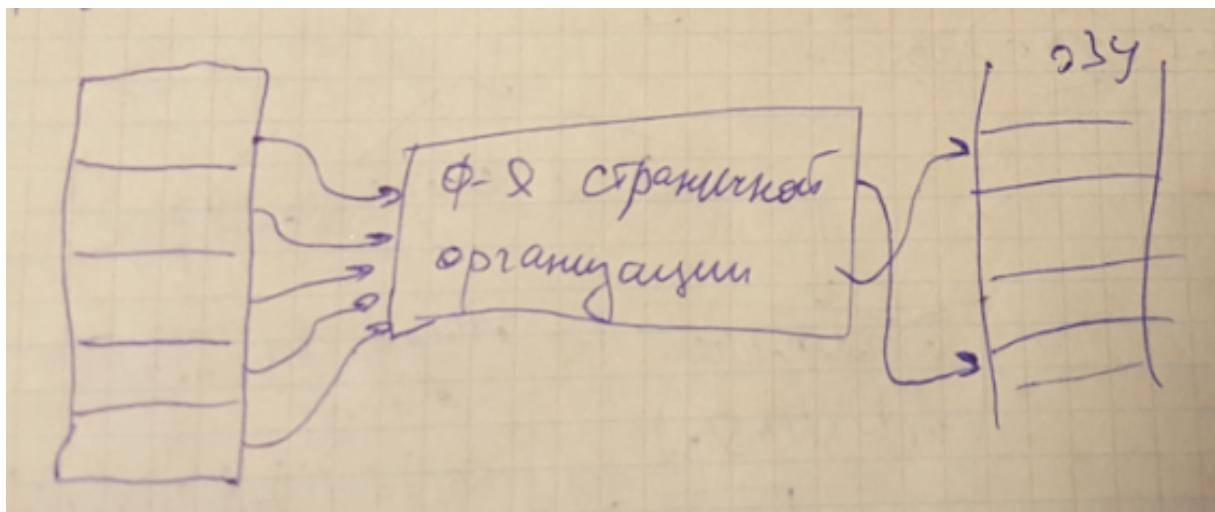


Рисунок 6.6 — pic

Страница – на которую делится адресное пространство процесса. А страницу, на которую делится физическая память называют фреймов.

#### 6.5.1.1 Прямое отображение

Отображаем страницы виртуального пространства на физическую. В процессоре имеется регистр, в который загружается начальный адрес таблицы страниц.

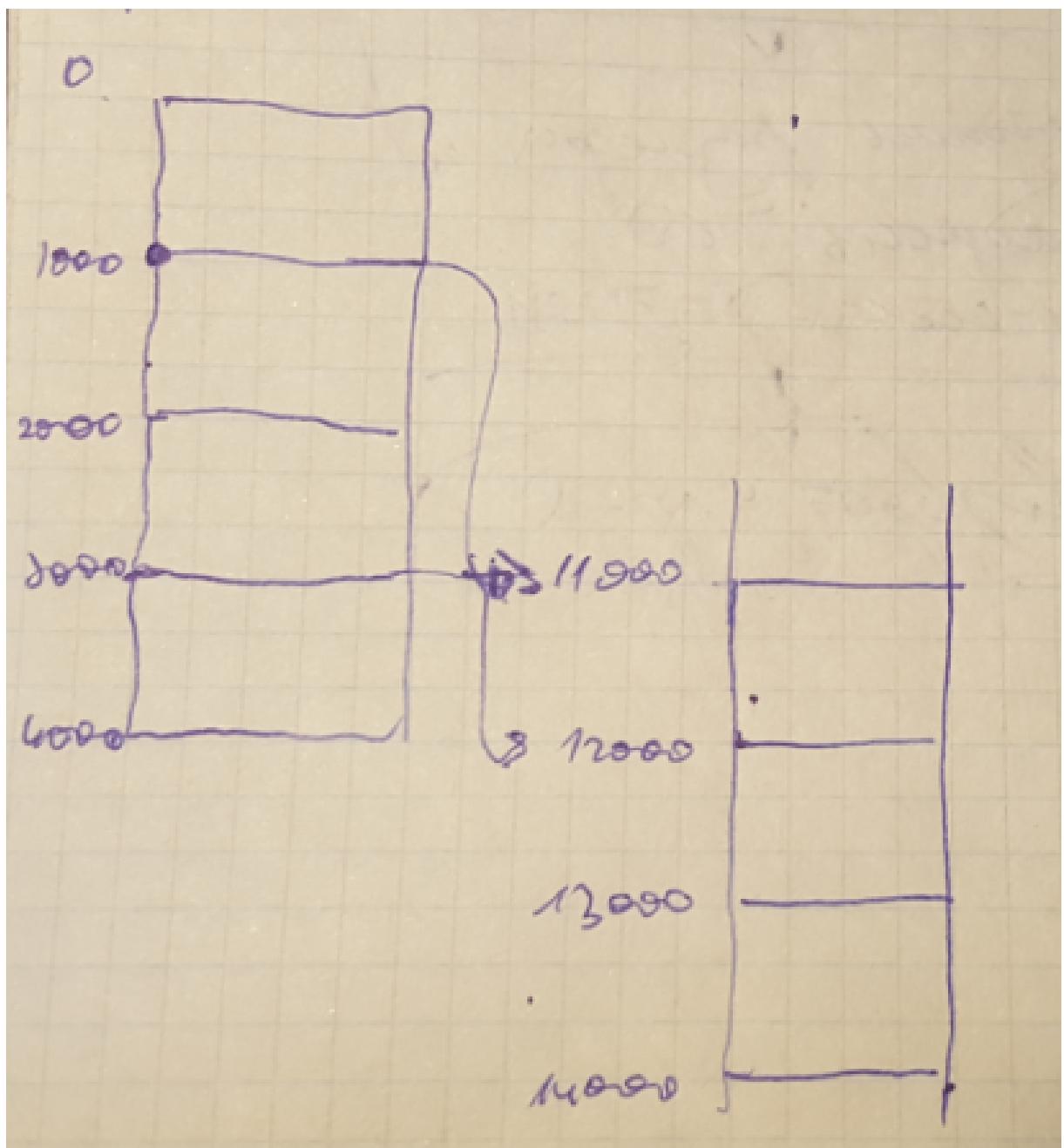


Рисунок 6.7 — pic

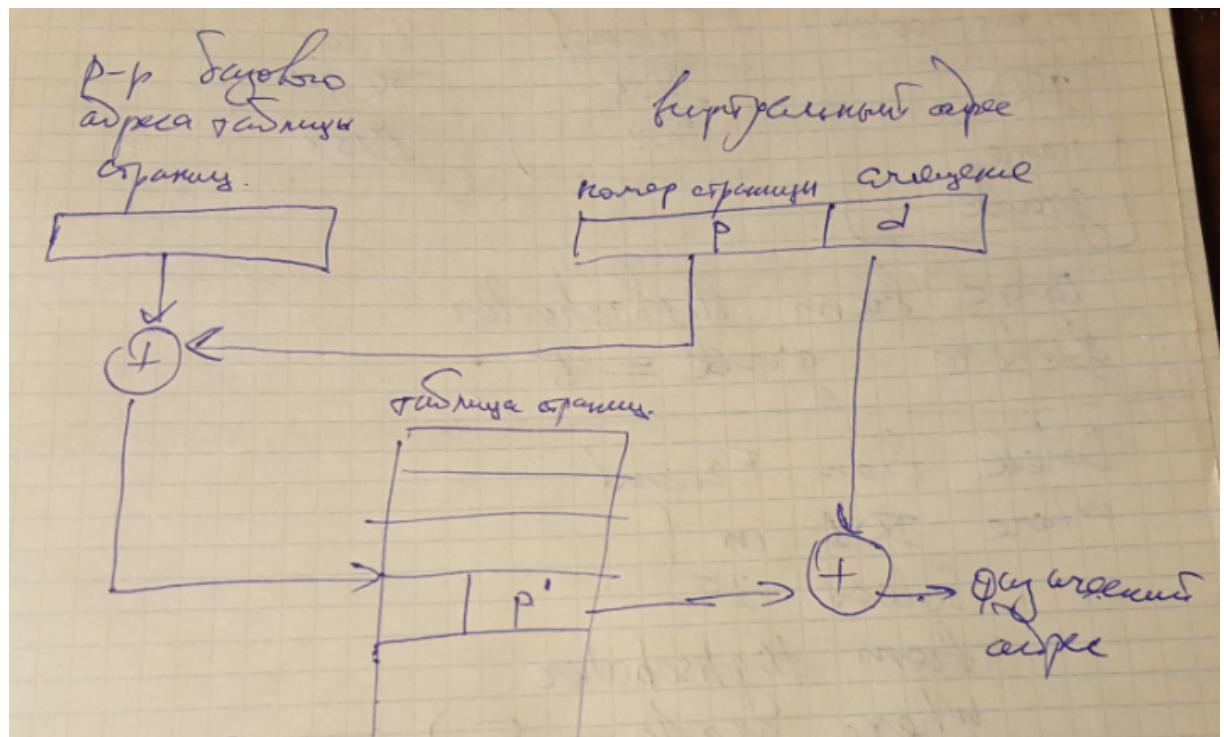


Рисунок 6.8 — pic

Таблица страниц находится в оперативной памяти. Таких страниц столько – сколько процессов. Если в системе большое количество процессов, каждый процесс имеет большое адресное пространство, таблица страниц так же занимает много памяти, если в данный момент этот процесс находится в состоянии блокировки, или sleep, то нет смысла эту таблицу держать в памяти. Каждую команду будет обращение к таблице. Увеличивается мультипроцессность.

### 6.5.1.2 Ассоциативное отображение

Предполагает наличие в системе специального вида памяти – ассоциативной. Ассоциативная память – дорогая, регистровая, катастрофически возрастает кол-во соединений. Позволяет произвести выборку информации за один такт. Параллельное сравнение ключа с другими ключами.

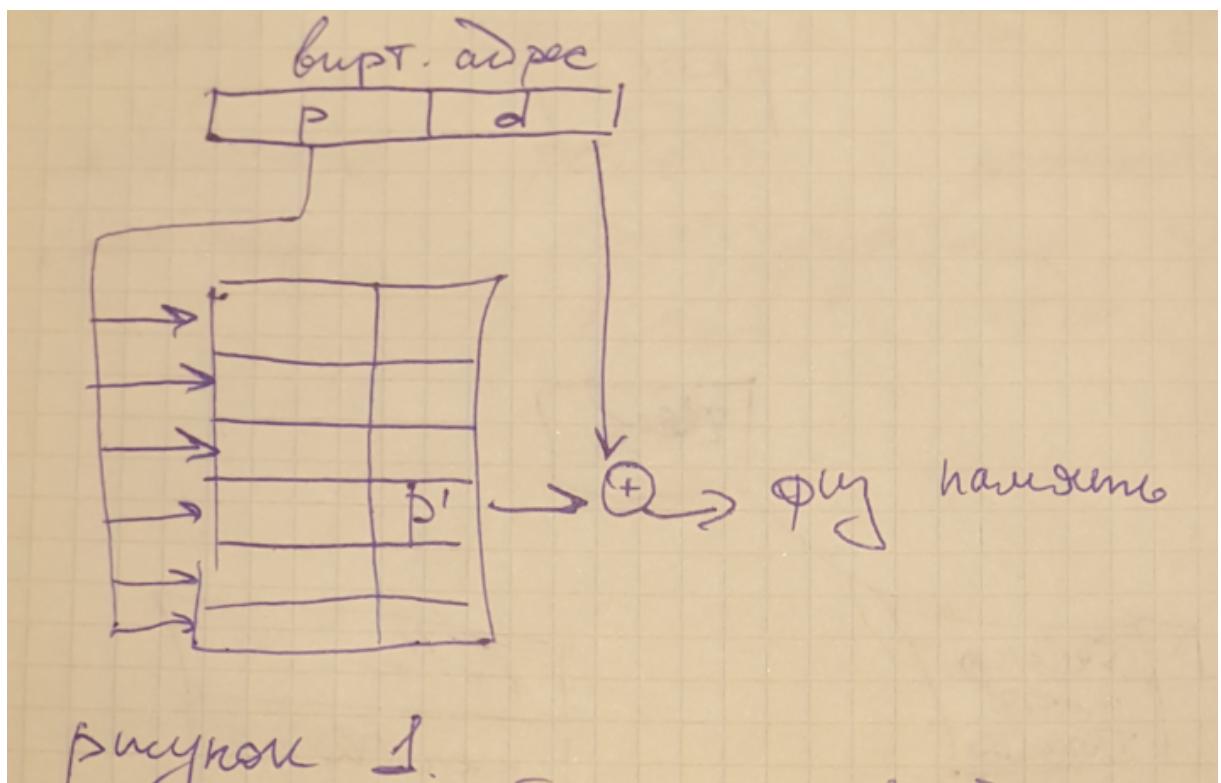


Рисунок 6.9 — pic

### 6.5.1.3 Ассоциативное – прямое отображение

В процессоре есть регистр базового адреса таблицы страниц. В нем содержится начальный адрес таблицы страниц в оперативной памяти. Страница сначала ищется в ассоциативном кэше. Если в кэше нет, то происходит обращение к физической памяти. Замещение происходит тех страниц, к которым обращение было давно.

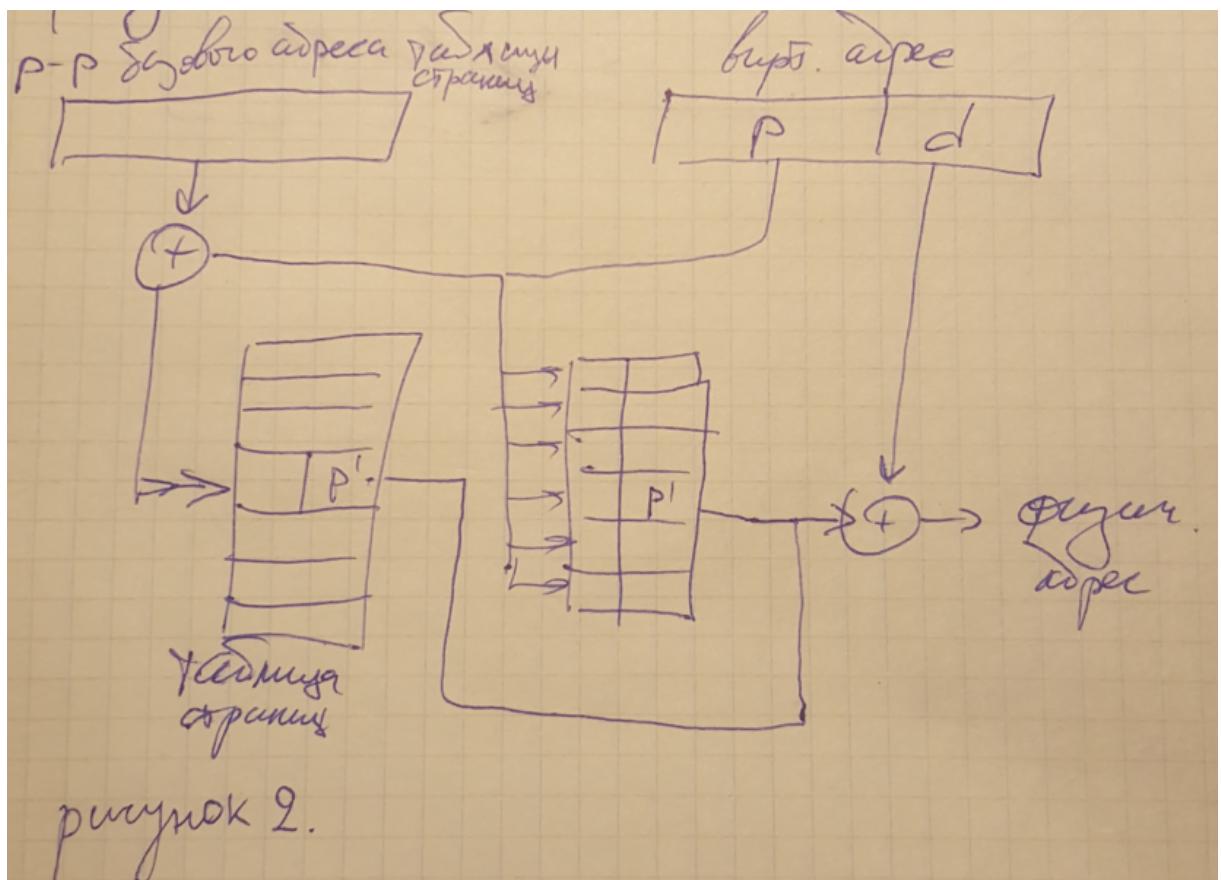


Рисунок 6.10 — pic

В процессорах Intel есть кэш TLB. Не полностью ассоциативный кэш, а частично ассоциативный. Используя алгоритм LRU, имея ассоциативный буфер 8 – 16 строк позволяет достичь порядка 90% характеристик полностью ассоциативного кэша, в который мы могли бы загрузить всю таблицу страниц. В системе столько таблиц, сколько процессов. При смене процесса актуальность кэша = 0. Таблицы страниц должны находиться в оперативной памяти. Эти таблицы пожирают ОЗУ. IBM была предложена двух уровневая организация таблиц страниц. Вводится понятие гиперстраницы. Виртуальный адрес делится на 3 части. Сохраняется стремление использовать ассоциативный буфер. Адресное пространство процесса делится на гиперстраницы, которые делятся на страницы. В результате таблиц страниц будет много, а таблица гиперстраниц – одна. Из таблицы гиперстраниц получаем базовый адрес таблицы страниц. Р – смещение таблицы страниц. Если необходимой страницы нет в кэше, то обращаемся к ??? и получаем адрес страницы физической памяти, который можем использовать для ??? линейного адреса. Мы можем выгружать те таблицы страниц, которые в данный момент не используются (к которым не было обращения).

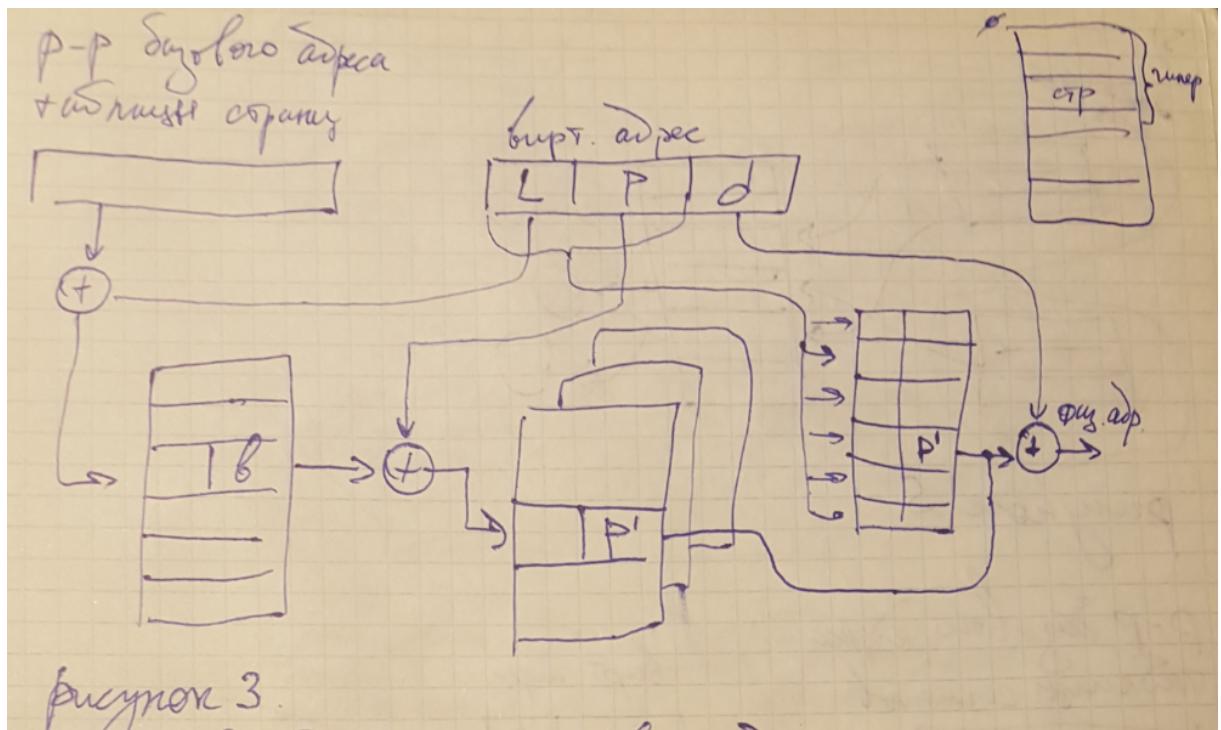


Рисунок 6.11 — pic

Copy-on-write решает проблемы с коллективным доступом. Страницы мы должны разделять. Разделение выполняется по принципу Copy-on-write. Если процесс захочет записать в страницу, то в его адресном пространстве будет создана копия страницы. Основная претензия к страничному ??? была снята. Intel поддерживает страницами по запросам и сегментами по запросам. Наличие регистра CR3 доказывает наличие независимого табличного преобразования.

### 6.5.2 Схема преобразования сегментами по запросам

Страница – физическое деление памяти. Сегмент – логическое деление памяти. Размер сегмента определяется размером программного кода. При сегментном преобразование необходимо проверять, не вышел ли процесс за собственное адресное пространство. Процесс имеет таблицу сегментов. Виртуальный адрес делится на 2 части: сегмент, смещение. В дескрипторе сегмента присутствует размер сегмента, при формировании адреса выполняется проверка.

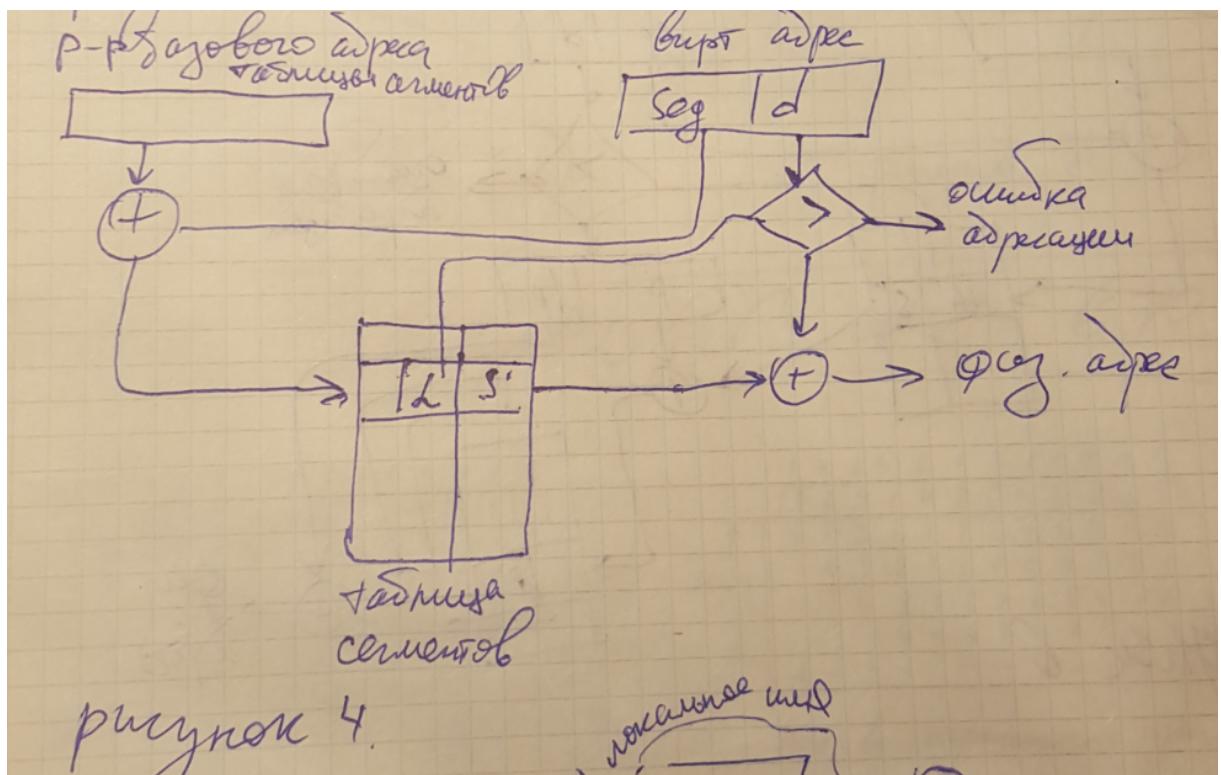


Рисунок 6.12 — pic

Организация ???:

- а) единая таблица. У сегментов в системе есть глобальное имя.
- б) локальные таблицы. ????. Сегмент имеет локальное имя.
- в) локальная таблица + глобальная. У каждого процесса есть возможность работать по его локальному имени и то же время есть глобальная таблица, описывающая сегменты памяти и обращения к сегментам памяти выполняются по глобальному имени.

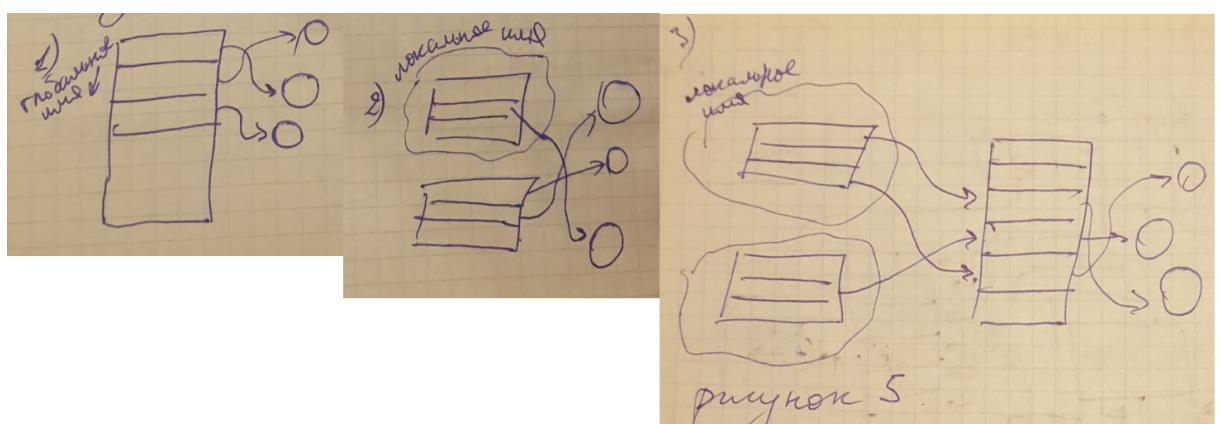


Рисунок 6.13 — pic

### 6.5.3 Управление памяти сегментами, поделенными на страницы

В выделении памяти разделами переменного размера были проблемы с фрагментацией памяти. Наши сегменты – перемещаемые, но надо поменять физические базовые адреса. Отредактировать все таблицы или глобальную таблицу. Большой объем работ, связанных с объединением свободной памяти. Цена вопроса? Намучавшийся с сегментами, люди придумали управление памяти сегментами, поделенными на страницы. Адресное пространство процесса делится на сегменты, но размер сегмента должен быть кратен размеру страницы. Есть таблицы сегментов, но в ней будет находиться базовый адрес таблицы страницы сегмента. Если программа поделена на несколько сегментов, то будет несколько таблицы страниц. Смещение делится на 2 части: страницу и смещение в странице. В таблице сегментов находится базовый адрес таблицы страниц этого сегмента.

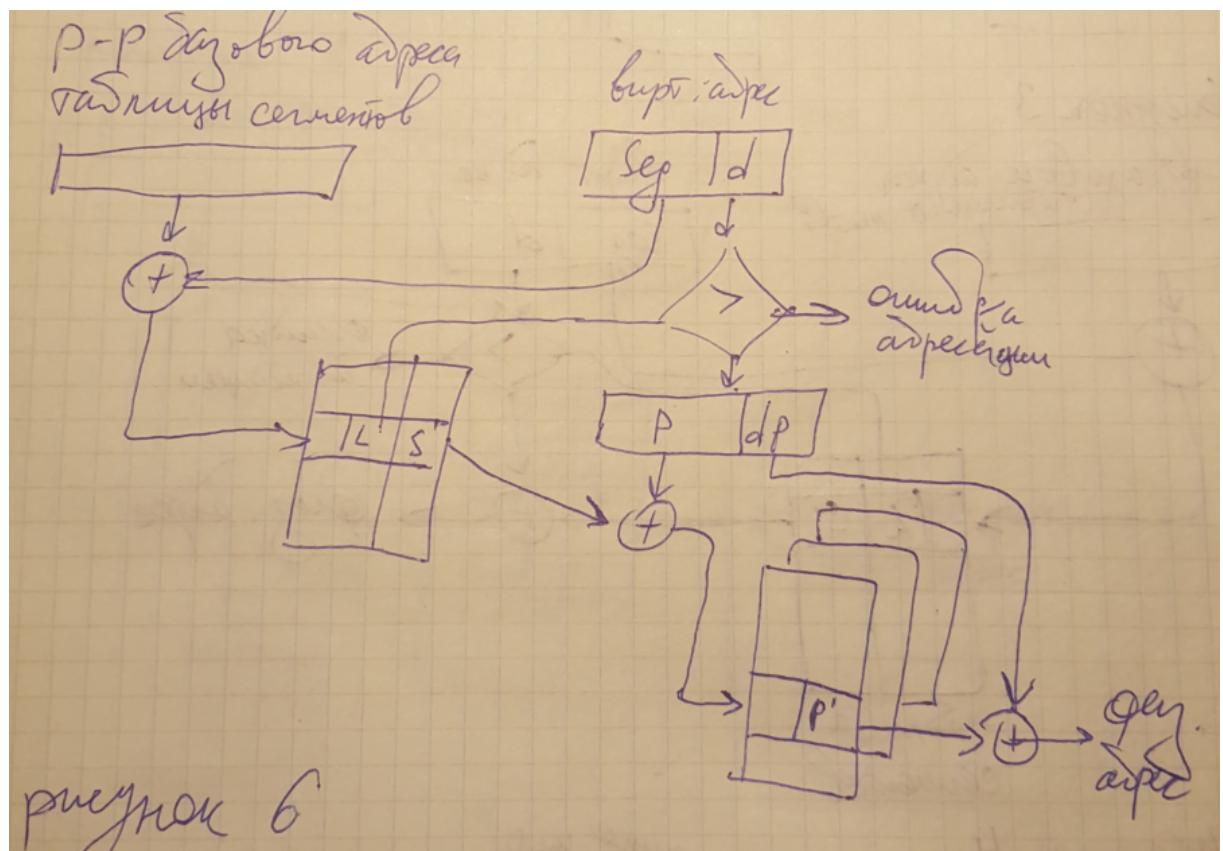


Рисунок 6.14 — pic

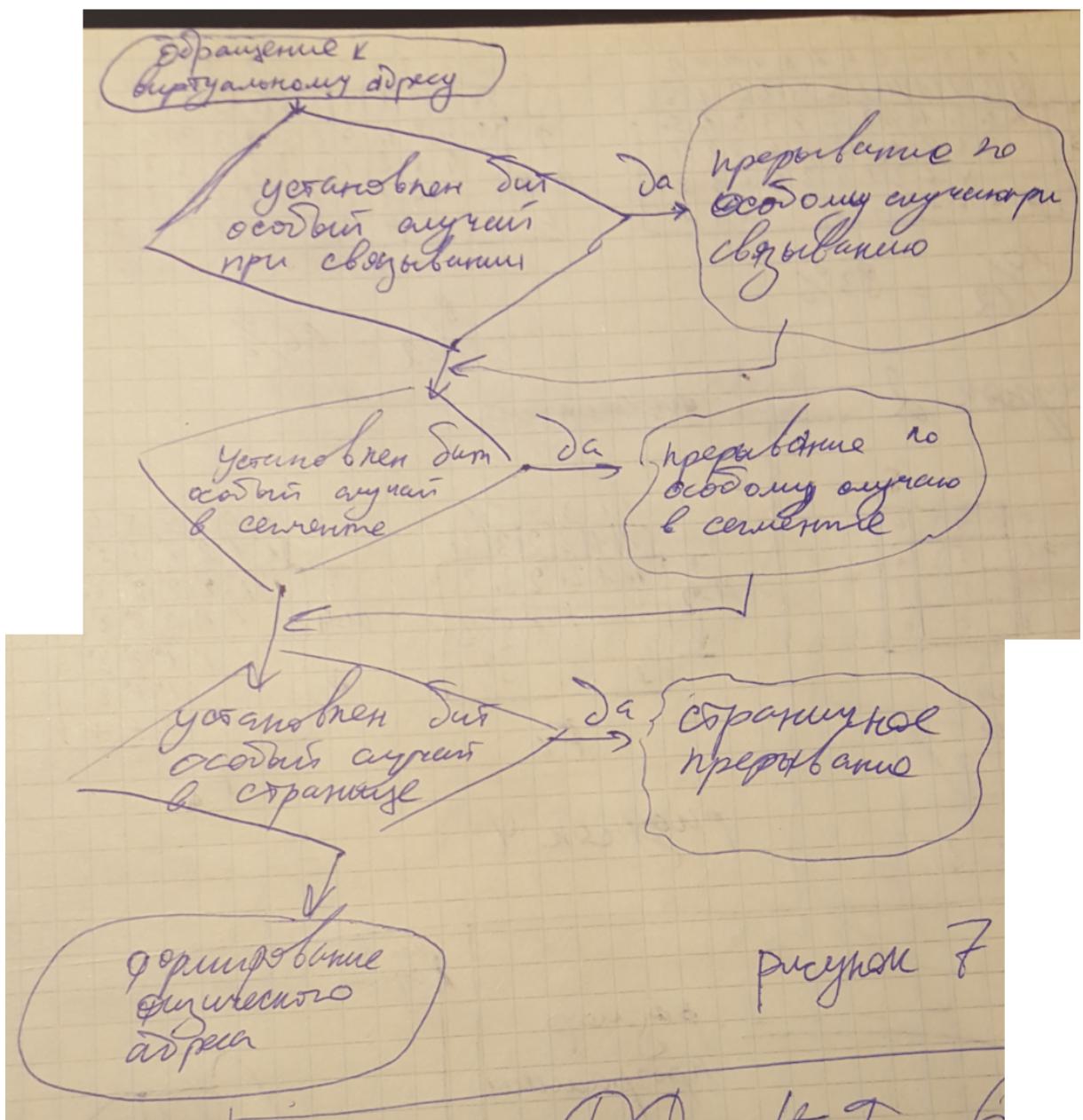


Рисунок 6.15 — pic

Речь идет об отложенном связывании (linking). Программа поделена на сегменты, сегмент линкуется только когда к нему происходит обращение и выполняется в процессе выполнения программы. Прерывание по особому случаю при связывании: появление в таблице сегментов данного процесса дескриптора данного сегмента. Для данного сегмента будет создана таблица страниц и это будет сделано по прерыванию по особому случаю в сегменте. И уже по страницочному прерыванию, используя таблицу страниц сегмента менеджер памяти сможет загрузить необходимую страницу в физическую память. В результате получим адрес начала ??? физической памяти и сложив со смещением получим линейный адрес. Свопинг заменяется пейджингом. Остается физическое деление памяти (сегменты делятся на страницы). При использовании copy-on-write преимущество данной схемы в ноль.

### 6.5.4 Алгоритмы Page replacement (замещение страниц)

#### 6.5.4.1 Выталкивание случайной страницы

Низкие накладные расходы. Не является дискриминационной. Но можем вытолкнуть только что загруженную страницу или часто используемую страницу.

#### 6.5.4.2 FIFO

Реализуется или присваиванием странице временной метки (когда страница загружена в память), или организуется связный список (вновь загруженная страница поступает в хвост. Страница для выталкивания берется из головы). Исключается только что загруженная страница. Свойство «аномалия fifo»: априорное предположение о том, что если увеличить объем доступной памяти, то кол-во страничных прерываний должно сократиться. Для некоторых траекторий процессов не подтверждается. Увеличение объема памяти ведет к росту страничных прерываний.

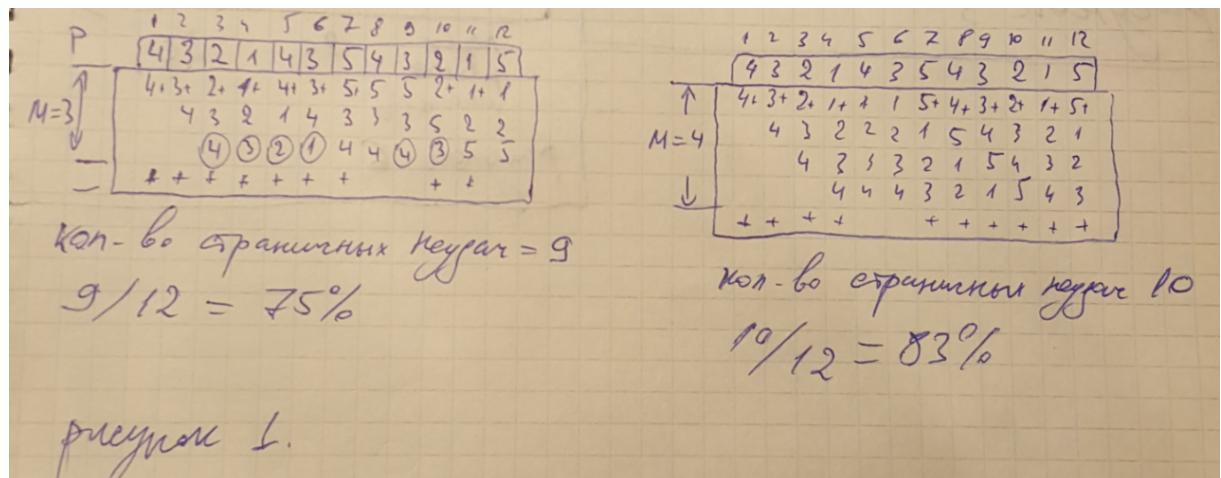


Рисунок 6.16 — pic

#### 6.5.4.3 LRU

Может быть организован с помощью временных меток или связанного списка. При каждом обращении к странице временная метка обновляется. Если связный список, то страница переносится в хвост. Никогда не будет вытолкнута часто используемая страница. Но алгоритм требует больших накладных расходов. В основе этого алгоритма лежит эвристическое правило, что если обращение было к странице, то следующее обращение будет к этой же странице. Связано со свойством локальности, характерным для наших программ. Проведем моделирование на этой же модели.

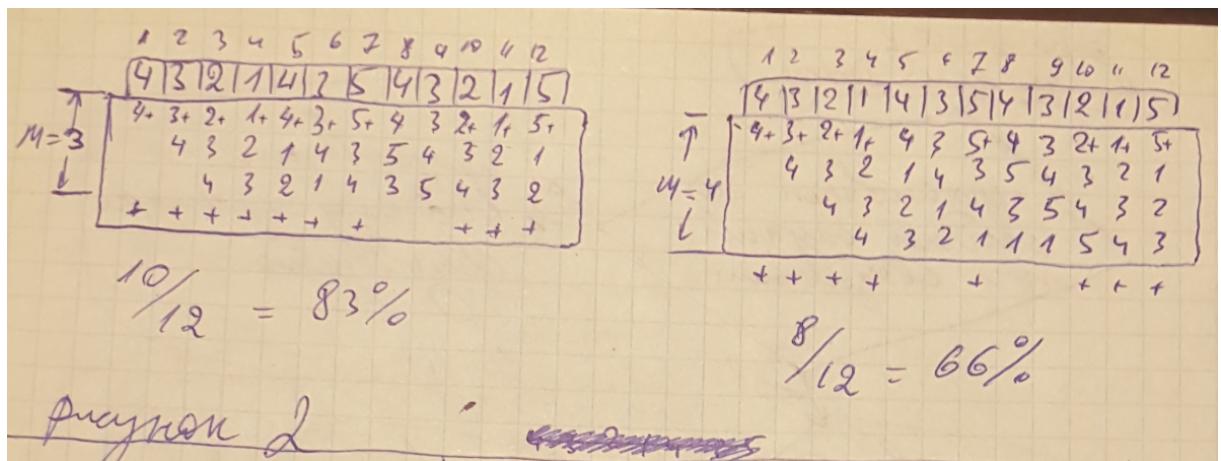


Рисунок 6.17 — pic

Свойство включения – первые три строки совпадают. Алгоритм относится к классу стековых алгоритмов. Никогда не приведет к росту числа страниценных прерываний при увеличении памяти. Крайне затратный. Связан с постоянным редактированием. В чистом виде не используется. Используется аппроксимация LRU.

#### 6.5.4.4 Алгоритм NUR

Страница не используемая в последнее время.

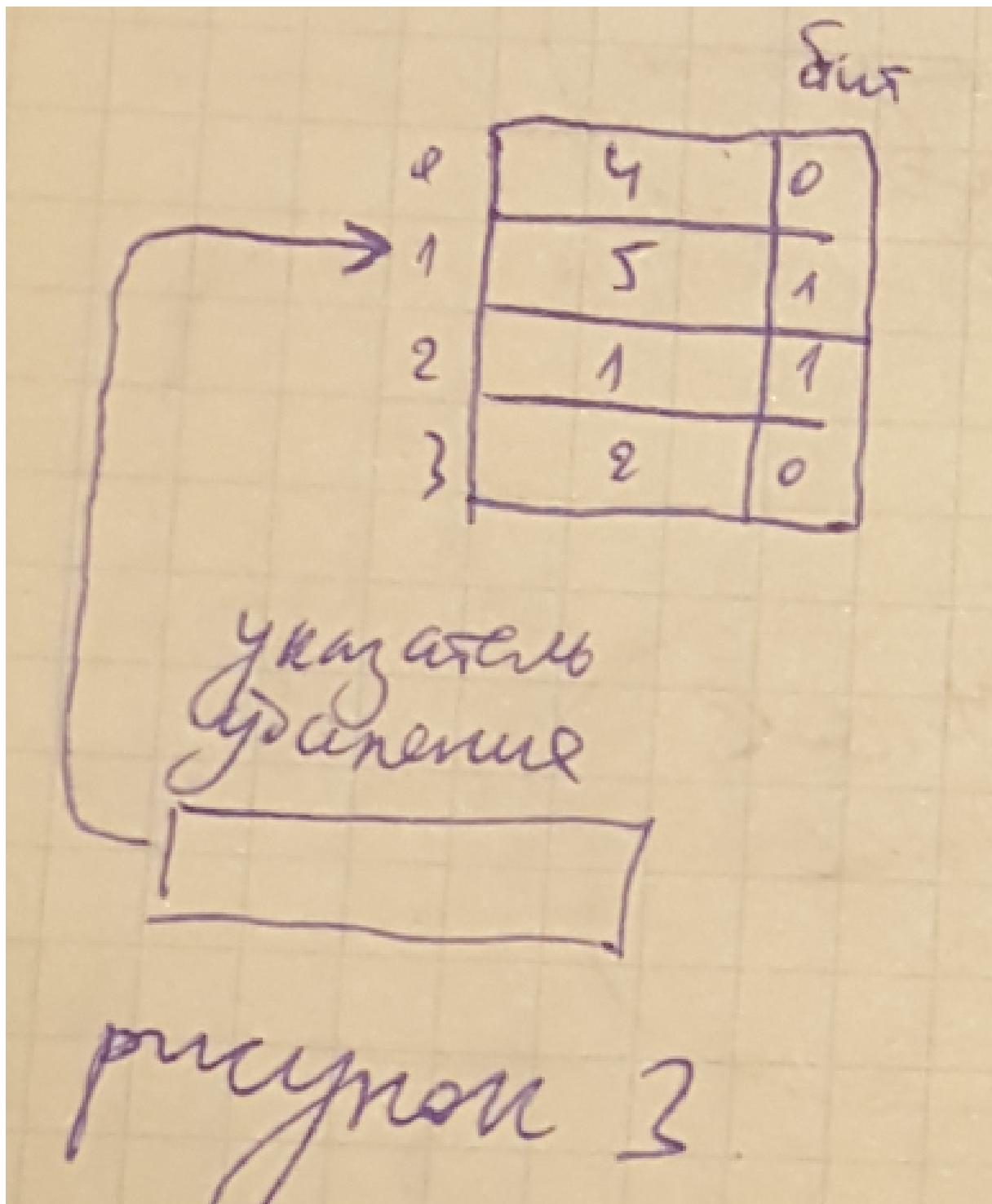


Рисунок 6.18 — pic

момент загрузки 5 страницы в первый кадр. Указатель удаления установлен на первый кадр. Если в дальнейшем потребуется удалить страницу, то проверку битов обращения будут начинать со второго кадра. Каждой странице приписывается бит обращения. В какие то моменты времени все биты сбрасываются в ноль. Затем при обращении к странице бит обращения устанавливается в 1. Для вытеснения ищется первая страница, у которой бит обращения = 0. Есть бит обращения. Так же

важен бит модификация. Если модификации не было, то точная копия находится на диске и её не надо копировать (мы избавляемся от необходимости копирования).

Бит обращения	Бит модификации
0	0
0	1
1	0
1	1

Выгодно выгружать не модифицируемые страницы.

#### 6.5.4.5 LFU наименее часто используемая страница

Эта стратегия близка к LRU. В ней контролируется частота обращения к странице. Может быть вытеснена только что загруженная страница.

#### 6.5.4.6 Метод связанных пар

Эти рассуждения связаны с размером страницы. На конкретной странице выполняется только часть команд. Чем больше размер страницы, тем меньшее количество команд будет выполнено. Если адресное пространство процесса поделено на небольшие страницы, то возрастает размер таблицы страниц. При этом таблица страниц должна находиться в оперативной памяти.

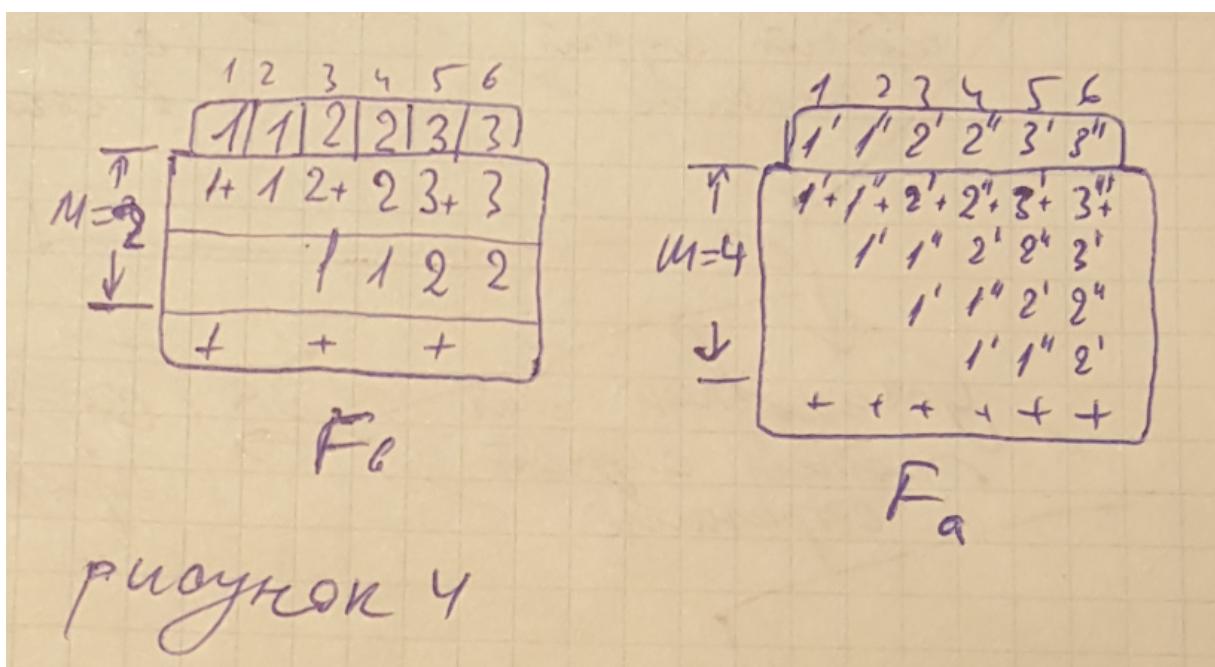


Рисунок 6.19 — pic

В 1973 году Медник показал, что для LRU существуют траектории страниц такого вида, для которых отношение числа страничных прерывания  $F_a/F_b$  будет

стремиться к  $M/N + 1$ , где  $N$  - размер страницы.  $1024/4 + 1 = 257$ . Наиболее часто употребим размер страницы 4 кб.

### 6.5.5 Вытеснения

Глобальное вытеснение – вытеснение любой страницы любого процесса. Локальное вытеснение – страница для вытеснения выбирается из пула загруженных страниц данного процесса.

**Страницное поведение программ. Производительность.**

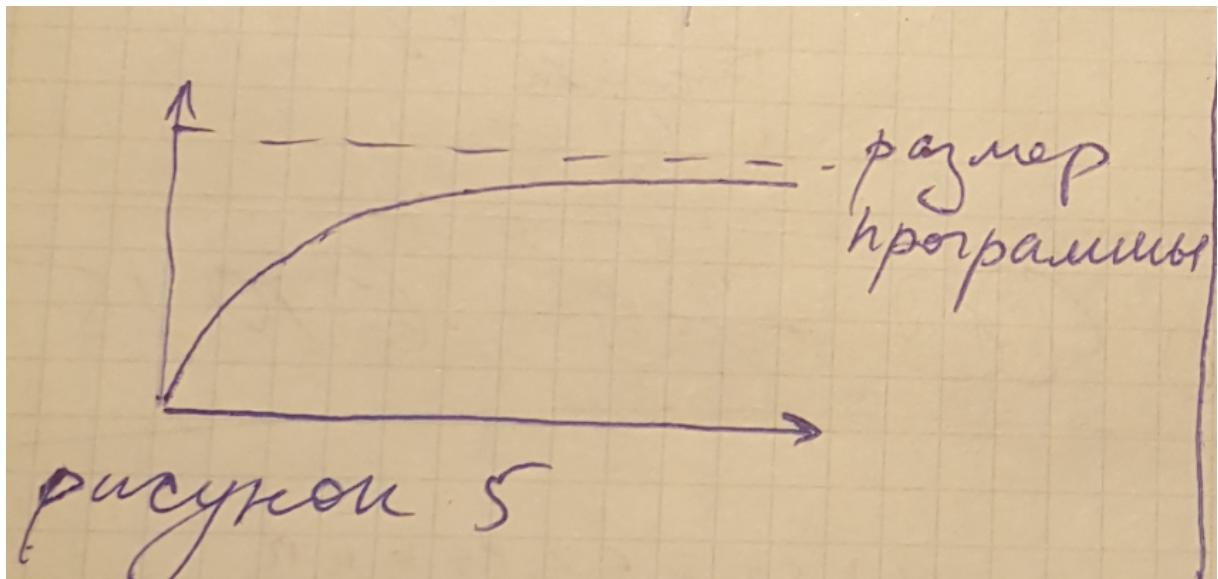


Рисунок 6.20 – График, который показывает процент страниц, к которым процесс обращается в течении своей жизни.

Минимально необходимая память: Страница кода (точка входа), страница сегмента данных, страница сегмента стека. Получив квант процессора, процесс начинает интенсивно подкачивать страницы. За время существования процесс обратится к большей части своих страниц. Этот перегиб связан с тем, что процесс для своего нормального выполнения, без страницных прерываний (увеличивают время выполнения процесса). Если процессу удается загрузить в память все страницы, к которым он обращается, то он будет выполняться без страницных прерываний. Этот факт получил название – теория рабочего множества. Поведение программы, во время выполнения, с точки зрения загрузки страниц, не является стабильным. Страницное поведение процесса не является стабильным. Денниг 1968 году предложил в качестве локальной меры производительности взять число страниц, к которым программа обращается за интервал времени  $\Delta t$ .  $W(t, dt)$  – рабочее множество. Если процессу удается загрузить в память всё рабочее множество, то процесс будет выполняться без страницных прерываний, т.е. эффективно. Если процесс не сможет загрузить всё свое рабочее множество (страницы, которые ему необходимы в данный промежуток

времени  $\Delta t$ ), то возникнет подкачка одних и тех же страниц, названная **трешингом** [thrashing].

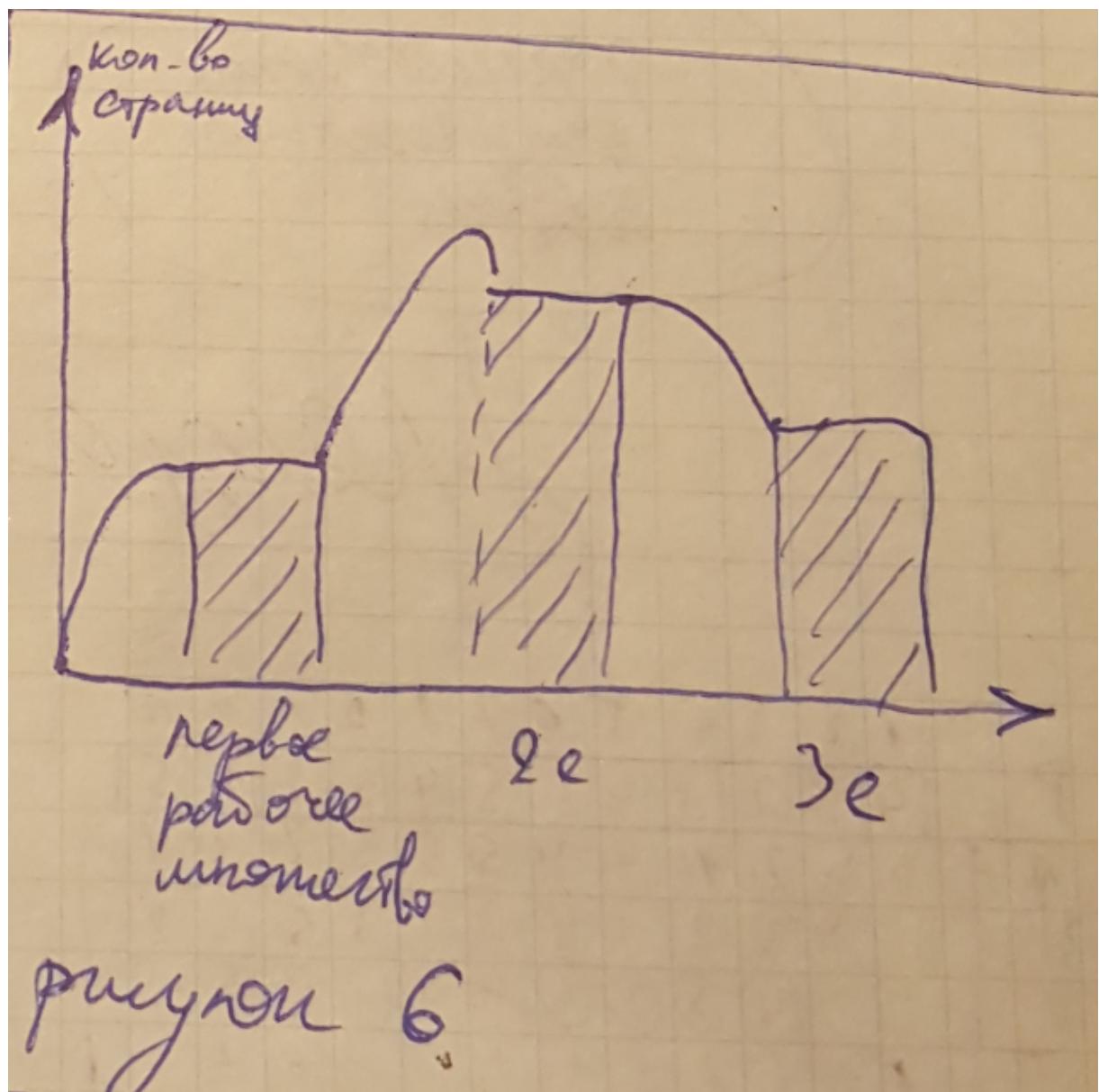


Рисунок 6.21 — pic

В процессе жизни процесса он обращается к разным набором страниц. Загрузив рабочее множество, процесс будет выполняться без страничных прерываний. Затем ему нужно перейти в другое рабочее множество.

Речь идет о системах общего назначения. Чтобы процесс эффективно выполнялся не должно быть страничных прерываний. Страницное прерывание – затратное действие в системе, связанное с работой менеджера памяти. В Unix есть страницный демон. В Windows есть что-то похожее. Страницный демон – вызывается таймером (Вызов отложенный, таймер инициирует действие. Он не может выполняться по кванту, так как будет поздно.) сканирует страницы и выгружает те страницы, к которым дольше не было обращения, чтобы, когда произойдет страницное прерывание как можно быстрее загрузить новую страницу. Процесс переходит с одного рабочего множества на другое. На иллюстрации (предыдущая лекция) выделяются горбики. В этот момент в памяти находятся страницы из старого и нового рабочего множества. Майкрософт: в их теории присутствует понятие рабочего множества. С классическим рабочим множеством это не имеет ни чего общего. Они могут заранее выделить рабочее множество. Имеется ввиду квота. Связано это с локальным и глобальным вытеснением. Процесс может держать в памяти до 10 страниц. Если процесс исчерпал свою квоту, а ему еще нужно, то он не может это сделать. В системе возникнет трешинг, т.е. резкий рост числа страницных прерываний (будет выгружать и загружать одни и те же страницы). Если возник рост кол-во страницных прерываний, то система увеличивает квоту. Все эти исследования были в 70-х годах. У IBM было практическое доказательство рабочего множества. Процесс никогда одновременно не обращается ко всем страницам, но в процессе жизни процесс обратится к максимальному кол-ву своих страниц.

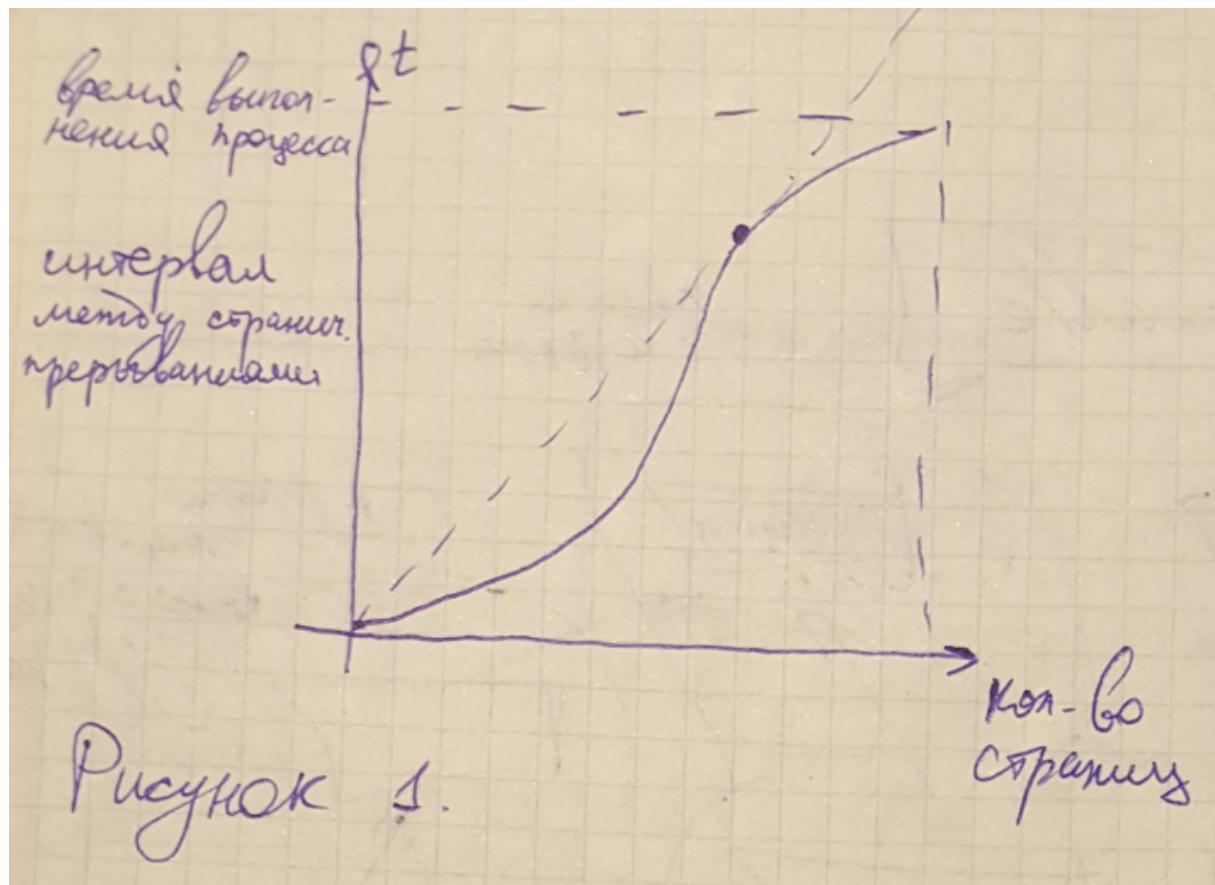


Рисунок 6.22 — pic

Интервал между страничными прерываниями линейно зависит от кол-ва страниц, но потом происходит перегиб. Он является следствием того, что в памяти находится все рабочее множество процесса. Эту кривую называют временем жизни.

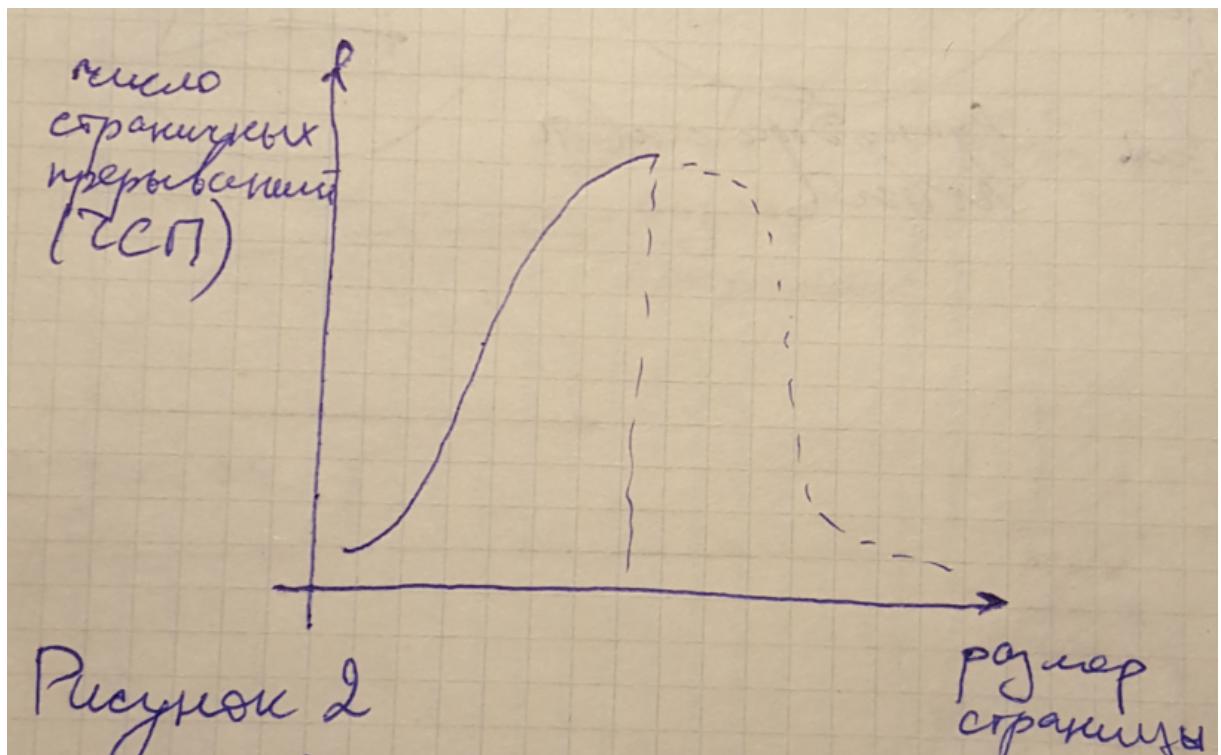


Рисунок 6.23 – pic

Показывает ЧСП в зависимости от размера страницы или ????. На большой странице будет выполняться меньшее кол-во команд. Наш лимит – размер программы.

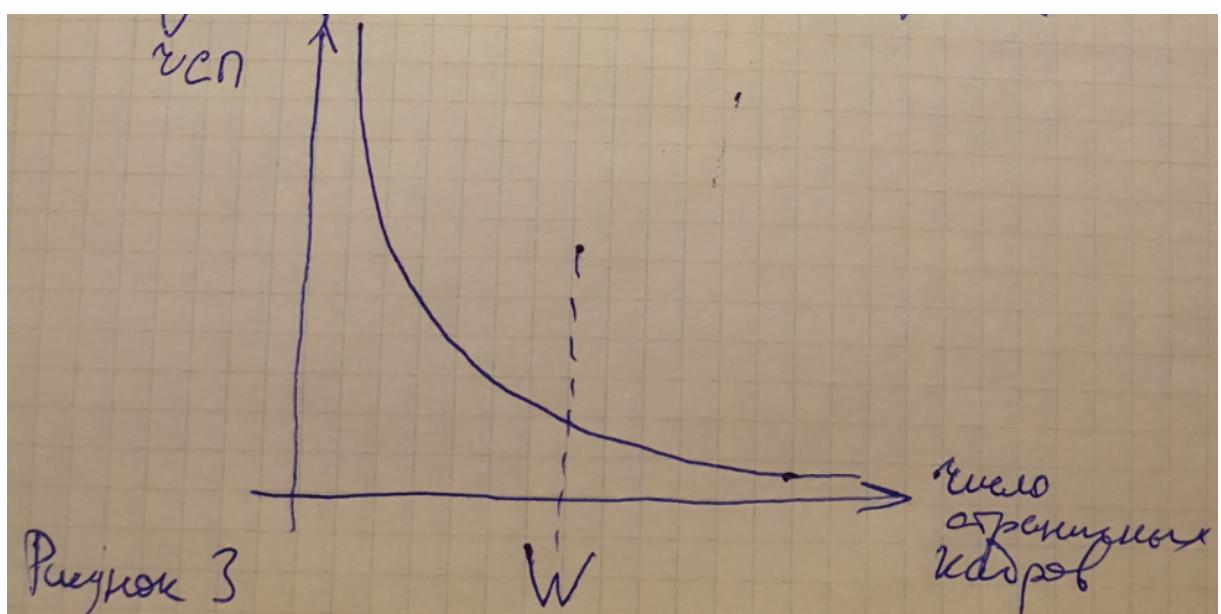


Рисунок 6.24 – pic

W – рабочее множество. Размер страницы фиксирован. Если квота маленькая – то ЧСП резко возрастает.

## 6.6 Процессоры Intel

Intel поддерживает управление памятью сегментами по запросам (регистры GDTR – содержит начальный адрес таблицы глобальных дескрипторов , LDTR - содержит начальный адрес таблицы локальных дескрипторов) для страницы CR3 – адрес где произошло страницное прерывание. РЕ можно отключить. Сегментное преобразование отключить нельзя. Для процесса выделяется один единственный дескриптор, который описывает сегмент размером 4гб.

4гб – виртуальное адресное пространство (ВАП).

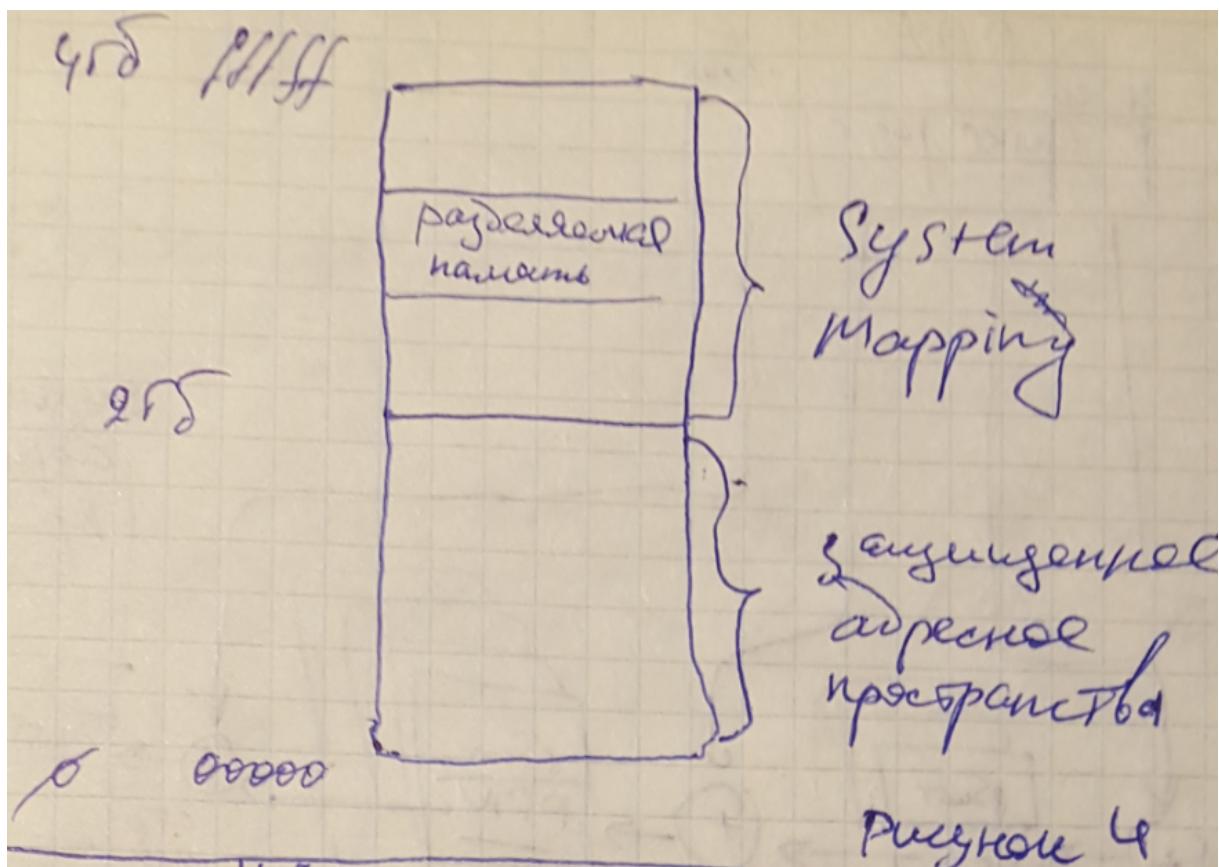


Рисунок 6.25 — Наиболее употребимый способ деления ВАП

Никто не копирует систему, Mapping – отображение. В ней существует область разделяемой памяти – область данных ядра системы, которая может разделяться процессами. Ни один процесс не может обратиться в адресное пространство другого процесса. Процессы взаимодействуют через адресное пространство ядра системы через системные вызовы. Чтобы система могла работать с этими страницами, необходимо создать таблицу страниц.

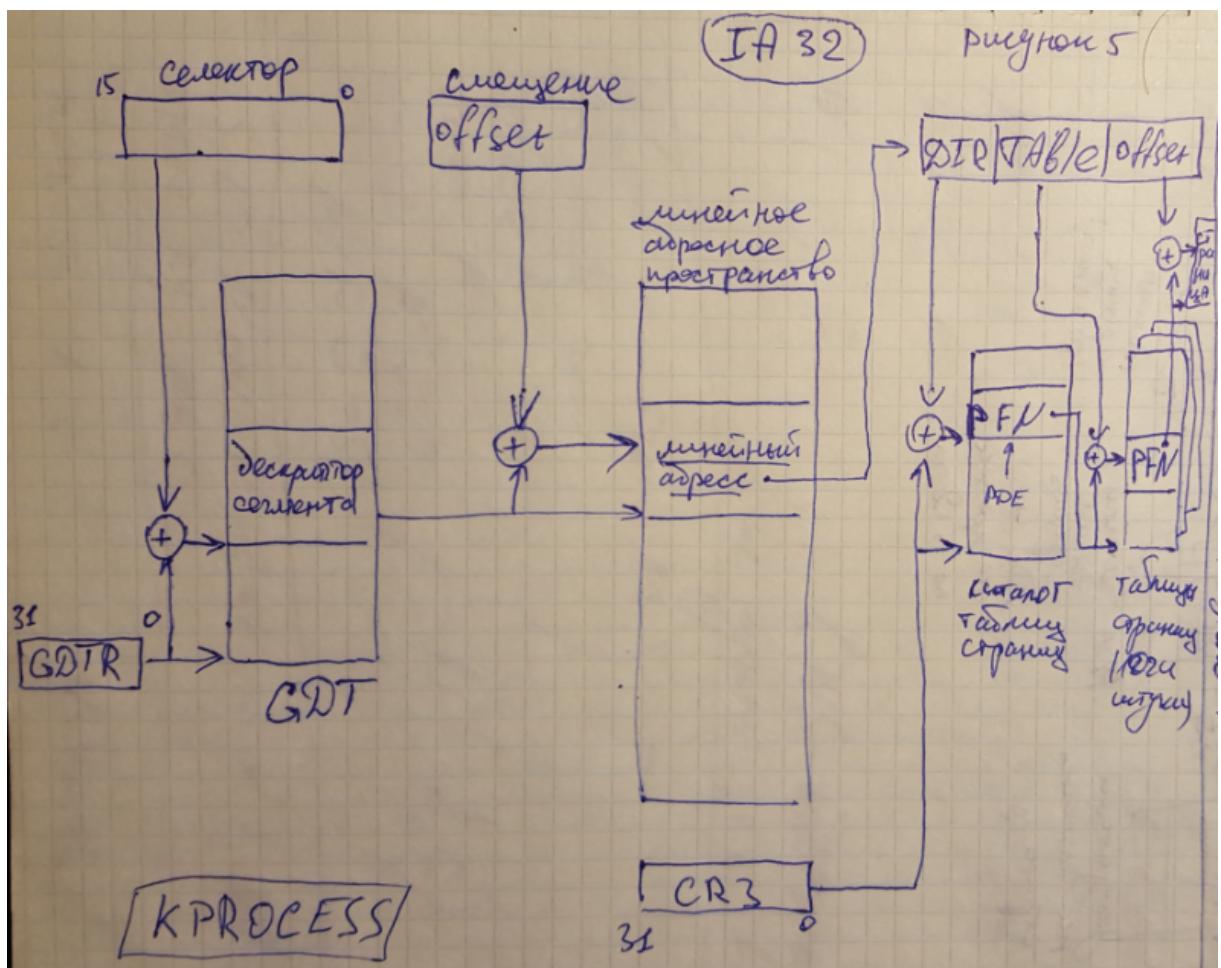


Рисунок 6.26 — pic

*offset* берем из программы. Из регистров *SI*, *DI*, *BP*, *SP*. Каждая программа имеет логическое адресное пространство. Следующее виртуальное (линейное). Каталог таблиц страниц (КТС) содержит дескриптор таблиц. КТС один на систему и может содержать 1024 строки. PDE - 4 байта. PFE (page frame number) – таблица страниц. 512 таблиц на систему, 512 – на процесс. В регистре *CR3* базовый адрес каталога таблиц страниц. Адрес (в регистре *cr3*) берется из дескриптора процесса. На шине адреса будет всегда находиться линейный адрес, т.е. адрес байта физической памяти. К преобразованиям шина адреса не имеет никакого отношения. Начиная с «пентиум про» поддерживают режим PAE (Physical AE). В режиме PAE виртуальный адрес делится на 4 поля.

31, 30	индекс	индекс	12 бит
индекс указателя на каталог страниц.	кataloga страниц	таблиц страниц	смещение offset

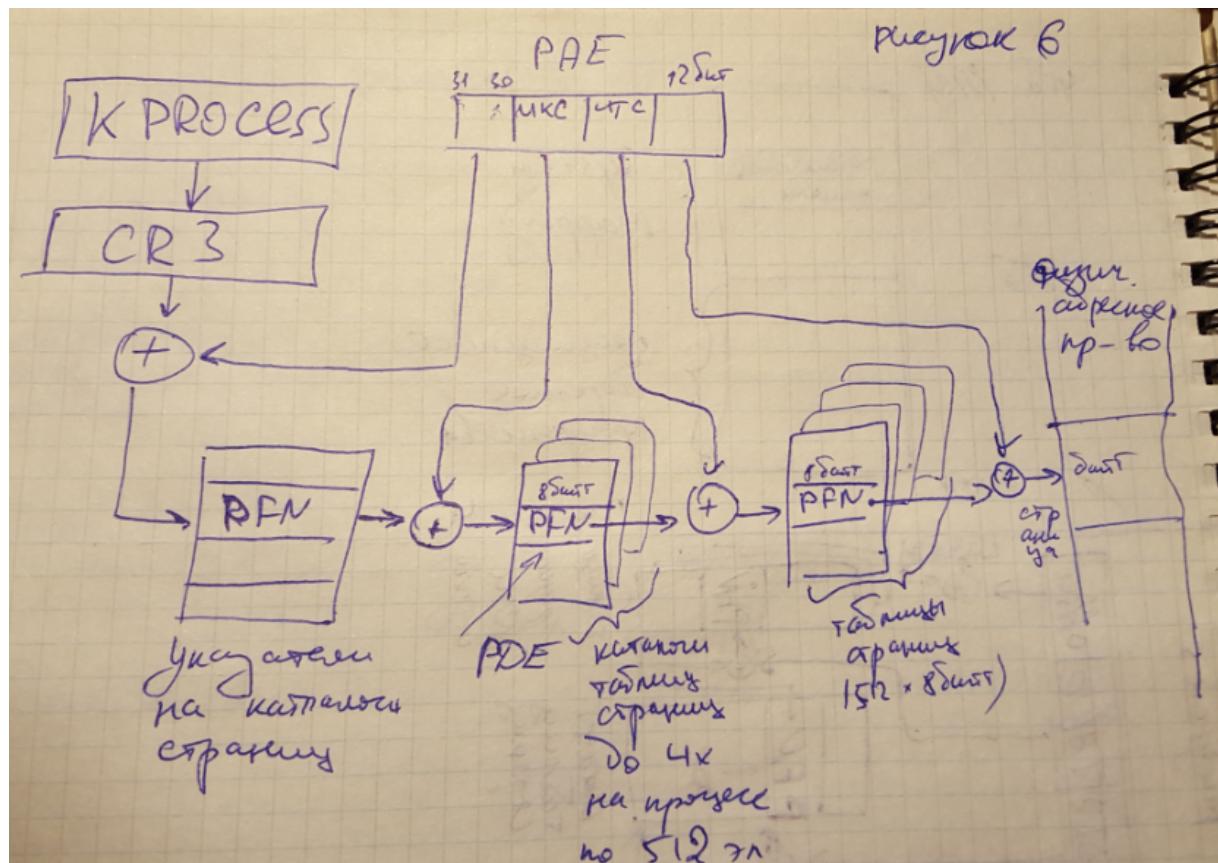


Рисунок 6.27 — pic

У процесса может быть 4 каталога таблиц страниц. Существует специальная версия 32 разрядного ядра Windows с поддержкой PAE. Ядро это называется *Ntkrnlpa.exe*. Кол-во преобразований больше, но обратиться можем к большей памяти. Страницы могут подгружаться. В любой системе оптимизирована загрузка страницы. Большая оперативная память позволяет держать больше программ.

## 7 Взаимодействие параллельных процессов

Чтобы не терять значение разделяемой переменной, необходимо обеспечить монопольный доступ (когда процесс получает доступ к критической секции (КС) по разделяемой переменной, то системой больше ни какой процесс не сможет получить доступ к этой же КС.)

Монопольный доступ обеспечивается методами взаимоисключения, делятся на:

- а) программные
- б) аппаратные
- в) с помощью семафоров
- г) мониторы.

В общем случае задача взаимоисключения решается для N процессов.

### 7.1 Программная реализация

С использованием флага, который проверяется в цикле.

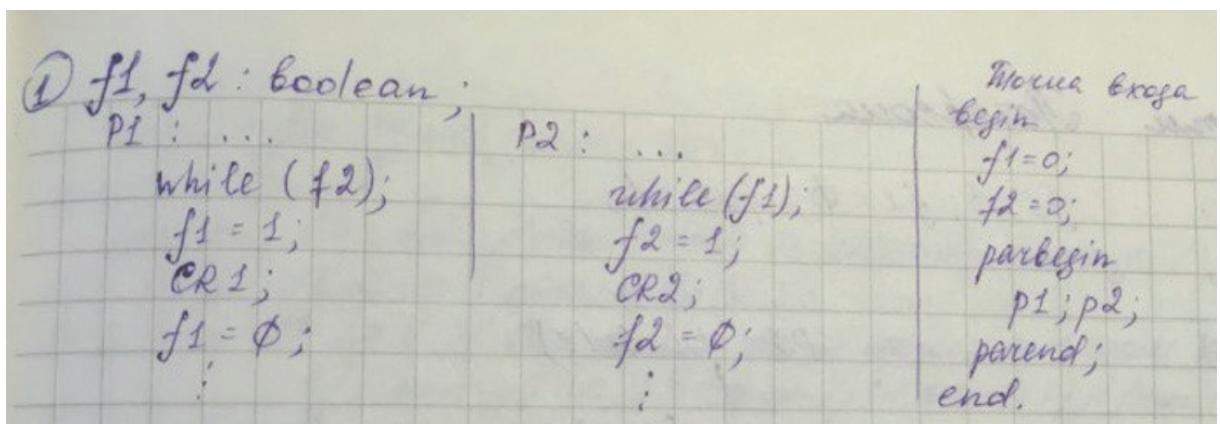


Рисунок 7.1 — вариант 1

В 7.1 второй процесс мог не успеть установить флаг, тогда первый процесс также пройдет цикл while, следовательно снова получаем одновременный доступ.

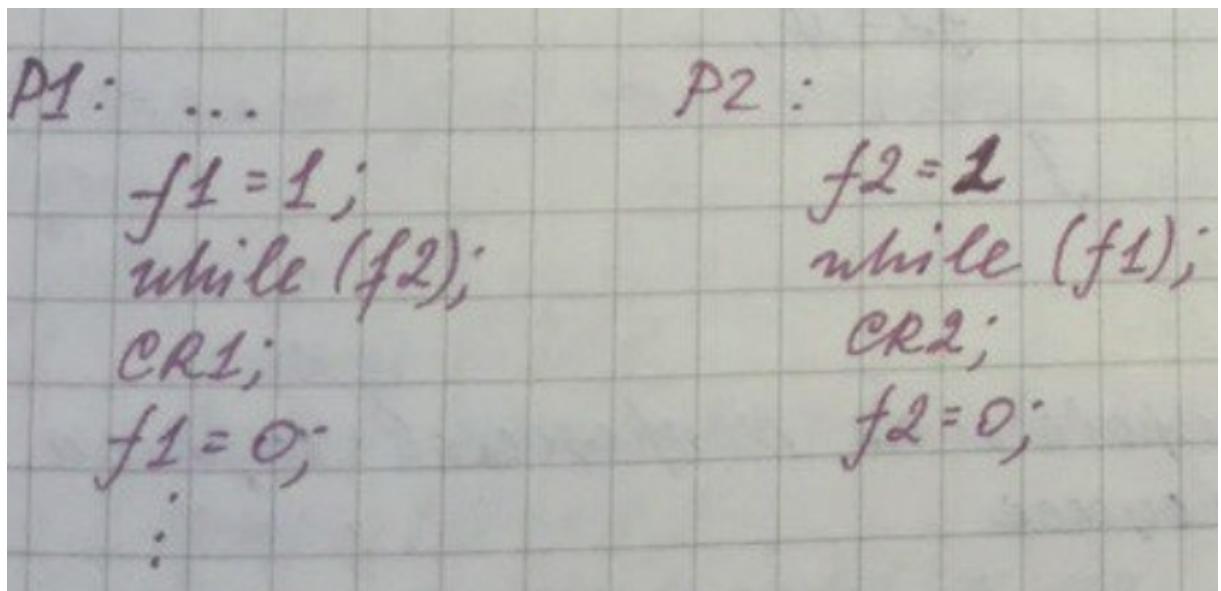


Рисунок 7.2 — вариант 2

В 7.2 зацикливание в проверке флага!

### 7.1.1 Алгоритм Дейкера

Дейкер ввел дополнительную переменную ("чья очередь?") и 2 флага. Алгоритм обеспечивает взаимоисключений.

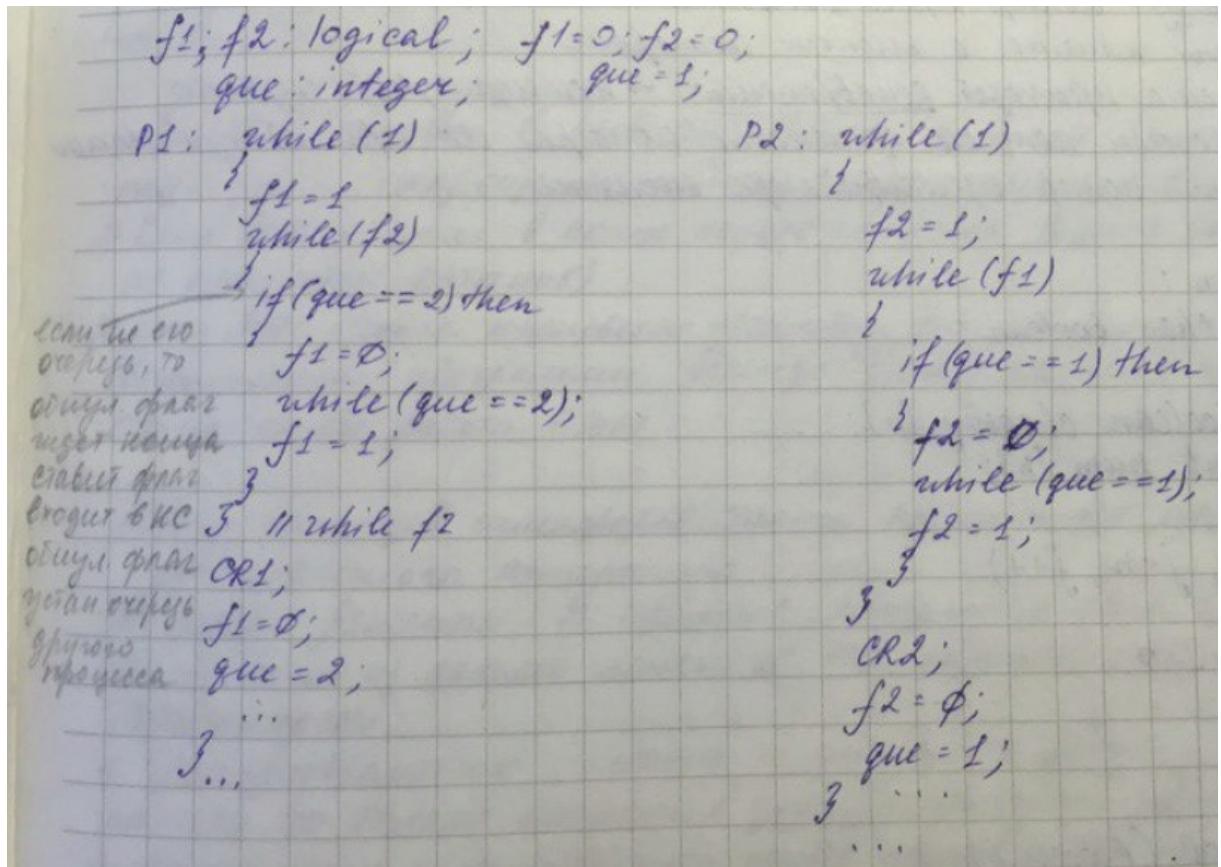


Рисунок 7.3 — Алгоритм Дейкера

### 7.1.2 Алгоритм Петерсона

Этот алгоритм можно пролонгировать на N процессов и рассматривать значение "que" как номер процесса.

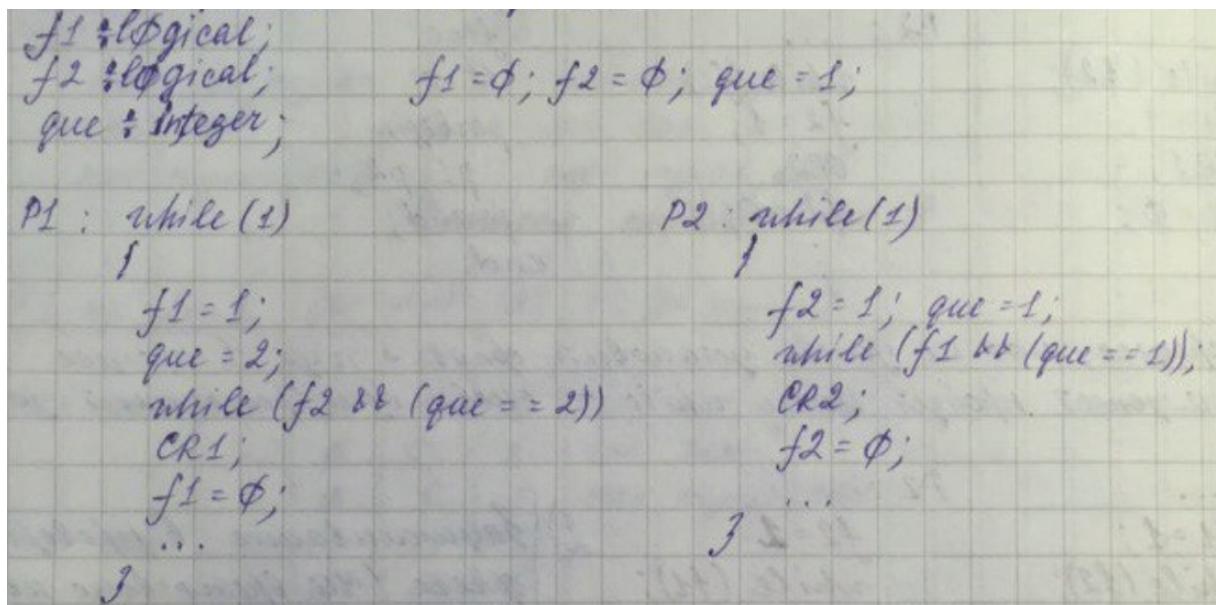


Рисунок 7.4 — Алгоритм Петерсона

### 7.1.3 Алгоритм Bakery (Лампорта)

Решает задачу взаимоисключения для N процессов. Каждому приходящему клиенту выдается жетон с номером. Каждому новому - номер строго больше. Продавец обслуживает клиента с меньшим номером. Если 2 клиента приходят одновременно, то возникает коллизия, которую нужно разрешить: процессы получают одинаковые номера, но при обслуживании, процесс, у которого номер идентификатора меньше

?????????????????????????????????????????????????????????????????????????  
?????????????????????????????????????????????????????????????????????????????  
?????????????????????????????????????????????????????????????????????????????

## 7.2 Аппаратная реализация

Если речь идет о об однопроцессорной системе, в которой параллельное выполнение происходит через квази-параллельность, то процесс выполняется до тех пор, пока им не будет запрошен сервис ОС или до прерывания. Если отменить прерывания на то время, пока процесс находится в КС, то обеспечивается взаимоисключение, но это невозможно.

В машине IBM370 машинная команда TEST\_AND\_SET реализует одновременную проверку и установку содержимого ячейки памяти, которая называется **байт блокировки**. Эта команда является неделимой, т.е. последовательность действий, которую выполняет эта команда прервать нельзя. Она читает значение логической переменной *b*, копирует в *a*, устанавливает для *b* значение "истинно".

Рассмотрим случай, когда процесс 1 (P1) хочет войти в КС, и процесс 2 (P2) уже в ней. P1 устанавливает C1 в 1 и входит в цикл проверки f. Т.к. P2 в КС, то f = 1. Т.о. P1 находится в своем цикле ожидания, пока P2 не выйдет из своего критического участка.

Этот способ реализации приводит к тому, что вероятность бесконечного откладывания очень мала, т.к. процесс выходит из КС и устанавливает флаг в false, то скорей всего другой процесс перехватит инициативу и сделает f = true.

TEST\_AND\_SET машинная команда, используемая в спин блокировках. Спин блокировка - циклическое выполнение проверки, обычно используется в ядре. Часто эта команда возвращает предыдущее значение объекта.

```

1 void spin_lock(spin_lock_t *c) {
2     while (testandset(c) != 0) {}
3 }
4
5 void spin_unlock(spin_lock_t c) {
6     c := 0;
7 }
```

TEST\_AND\_SET в основных системах связана с блокировкой шины данных. Но длительный уцикл может привести к занятию шины одной нитью и приведет к существенному снижению производительности системы. Решение - через 2 вложенных цикла.

```

1 void spin_lock(spin_lock_t *c) {
2     while (testandset(c) != 0) {
3         while (*c != 0) {}
4     }
5 }
```

Если переменная занята, то выполняется вложенный цикл, в котором проверка переменной  $C$  выполняется без захвата шины данных.

### 7.3 Семафоры Дейкстры

1965 год - первая работа Дейкстры, связанная с семафорами. Семафор - неотрицательная защищенная переменная  $S \geq 0$ , на которой определены 2 операции  $P(S)$  - пропустить, и  $V(S)$  - освободить. Переменная защищена, т.к. м.б. изменена только тими двумя неделимыми операциями.

Семафор  $(0, 1)$  - бинарный.

Операция  $P(S)$  - декремент. Процесс, вызвавший операцию  $P(S)$  для  $S = 0$  будет заблокирован на семафоре.

Операция  $V(S)$  - инкремент, т.е. процесс разблокирует другой процесс, если было  $S = 0$ .

Заблокированный процес не выполняет цикл активного ожидания, т.е. семафоры Дейкстры решили проблему активного ожидания. Но заблокировать/разблокировать процесс может только ядро. Вызов команд  $P(S)$  и  $V(S)$  приводит к переводу системы в режим ядра. Считывающий семафор - принимает целые неотрицательные значения.

```
1 //P1: ...
2 P(S);
3 CR1;
4 V(S);
5
6 //P2: ...
7 P(S);
8 CR2;
9 V(S);
```

Осуществляется переход в режим ядра при захвате и освобождении семафора, т.е. происходит переключение контекста. Считывающие семафоры могут принимать не отрицательные значения. Современные ОС поддерживают наборы считающих семафоров. По своей семантике наборы семафоров похожи на массивы (массивы семафоров). У набора семафоров есть важное свойство: одной неделимой операцией можно изменить значения всех семафоров или части семафоров. Если процесс использует большое кол-во разных семафоров то сложно уследить за состоянием семафоров, то такие программы могли висеть в тупиках.

Другие средства взаимоисключения: мьютекс.

#### 7.4 Мьютесксы

Между семафорами и мьютексами есть серьезные различия:

- a) Мьютекс имеет хозяина, только процесс, захвативший мьютекс, может его освободить. Семафор может освободить любой другой процесс, зная идентификатор.
- б) Чтобы обеспечить для мьютекса свойство владельца, если мьютекс пытается захватить более приоритетный процесс, то временно повышается приоритет процесса хозяина.
- в) Мьютекс не может быть случайно удален, т.е. если он захвачен кем то, то удалить процесс-хозяин – нельзя.

#### 7.5 Задача обедающих философов

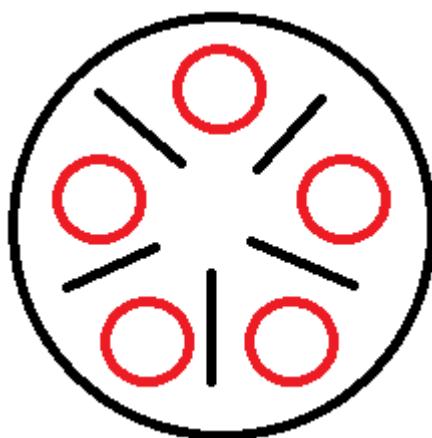


Рисунок 7.5 — pic

Возможны три сценария действия философов (эти ситуации – негативные в системе):

- a) Голодание: Каждый пытается взять левую и правую вилку, если удается, то он начинает есть. Через какое то время кладет вилки обратно.
- б) Тупик: Философ берет правую вилку, удерживает её и пытается взять левую.
- в) Зависание: Философ берет правую вилку, если не может взять левую , то кладет обратно. Захват и освобождение одних и тех же ресурсов. Это похоже на трешинг – загрузка и выгрузка одних и тех же страниц.

```

1  var
2      forks: array[1..5] of semafort;
3      i: integer;
4  begin
5      i := 5;
6      repeat
7          forks[i] := 1;
8          i := i - 1;
9      until i = 0;
10
11 cobegin
12     1:
13     begin
14         left := 1;
15         right := 2;
16         ...
17     end;
18     ...
19     5:
20     begin
21         left := 1;
22         right := 2;
23     repeat
24         // думает
25         P(forks[left], forks[right]);
26         // ест
27         V(forks[left], forks[right]);
28         ...
29     forever;
30 coend;

```

Рисунок 7.6 — listing

## 7.6 Задача производство потребление

Один процесс производит данные, другой процесс – считывает эти данные. Один – кладет произведенные данные в ячейки буфера, другой – считывает из ячеек буфера. Дейкстра предложил решение на трех семафорах, двух считающих и одном бинарном.

*SB* – бинарный семафор.

*SE* – считает пустые ячейки.

*SF* – число заполненных ячеек.

```
1 program semaphore;
2 SE, SF, SB: int;
3
4 SE = N;
5 SF = 0;
6 SB = 1;
7
8 producer: ...
9     P(SE);
10    P(SB);
11    N = N + 1;
12    V(SB);
13    P(SF);
14    ...
15
16 consumer:
17     P(SF);
18     P(SB);
19     N = N - 1;
20     V(SB);
21     P(SF);
22     ...
```

Рисунок 7.7 – listing

Если покупатель работает быстрее, то покупатель будет блокирован на *SF*, так как нет заполненных ячеек.

Если продавец работает быстрее, то при попытке декрементировать *SE*, продавец будет блокирован. Изменение *N* – критическая секция.

Дейкстра решил с использованием отдельных семафоров. Но так же можно решить с помощью набора семафоров.

## 7.7 Мониторы

Язык параллельного программирования: Ада.

Цель монитора – структурировать средства взаимоисключения.

Идея – создание механизма, который бы соответствующим образом унифицировал взаимодействие параллельных процессов. Монитор представляет синтаксическую конструкцию. Монитор содержит данные и процедуры, которые могут изменять эти данные. Говорят, что монитор защищает свои данные, так как изменить данные монитора можно только с помощью процедур монитора. Монитор может предоставляться языком, а может быть системным средством. Монитор гарантирует, что в каждый момент времени процедуры монитора могут использовать только один процесс. Если процесс успешно вызвал процедуру монитора, то говорят, что такой процесс находится в мониторе, остальные процессы ставятся к монитору в очередь. Классические мониторы используют 2 системных вызова wait и signal, эти системные вызовы определены на переменной типа «условие».

Рассмотрим пример простого монитора, потом кольцевого, а потом монитор Хоара (читатели-писатели).

### 7.7.1 Простой монитор

– обеспечивает выделение единственного ресурса произвольному числу процессов. [3]

```

1  RESOURSE: MONITOR
2  var
3      busy: logical;
4      x: conditional;
5
6  procedure ecquire; {захватить}
7  begin
8      if busy then wait(x);
9      busy := true;
10 end;
11
12 procedure release; {освободить}
13 begin
14     busy := false;
15     signal(x);
16 end;
17
18 {начальные установки}
19 begin
20     busy := false;
21 end;

```

Рисунок 7.8 – listing

Монитор обслуживает произвольное число процессов, кол-во которых ограничено длинной очереди. `signal(x)` проверяет очередь процессов к монитору и выбирает один для запуска на выполнение.

### 7.7.2 Монитор кольцевой буфер

Также задача производство потребление. Понятие синхронизация процессов – связана с выравниванием скоростей на каком то отрезке деятельности. Взаимоисключение процессов - ???.

```

1  RESOURSE: MONITOR
2  var
3      bcircle: array[1..n] of <type>;
4      pos: 0..n;
5      wp: 0..n-1; //запись в позицию
6      rp: 0..n-1;//чтение из pos
7      buf_empty, buf_full: conditional;
8
9  procedure producer(data: <type>);
10 begin
11     if (pos) then
12         wait(buf_empty);
13     bcircle[wp] := data;
14     pos := pos + 1;
15     wp := (wp + 1) mod n;
16     signal(buf_full);
17 end;
18
19 procedure consumer(var data: <type>);
20 begin
21     if (pos = 0) then
22         wait(buf_full);
23     data := bcircle(rp);
24     pos := pos - 1;
25     rp := (rp + 1) mod n;
26     signal(buf_empty);
27 end
28
29 {начальные условия}
30 begin
31     pos = 0;
32     wp = 0;
33     rp = 0;
34 end;

```

Рисунок 7.9 — listing

Механизм кольцевого буфера используется для организации работы spooler.  
SPOOL – Simultaneous Peripheral Operations On Line – для вывода текста на принтер.

**Задача читатели – писатели**. Бытовая интерпретация – система продажи билетов на самолеты, когда бронируются места, конкретный день и рейс. Состоит из 2 этапов:

- a) чтение информации, указываем вылет прилет, дату ...
- b) бронирование.

Место – критическая переменная. Когда бронируем – мы выступаем как писатель. Писатель (по конкретной разделяемой переменной) может быть только один.

### 7.7.3 Монитор Хоара

Если кто то пишет или есть писатели, ждущие своей очереди, то читатель переводится в состояние ожидание на переменной типа условие `canread`. Если читатель минует блокировку, то увеличивается активных читателей и читатель посыпает сигнал читателям, которые ожидают своей очереди на чтение. Цепная реакция читателей. Каждый читатель инициирует других читателей. Читатели могут работать параллельно. По завершении чтения процесс вызывает `stopread()`, уменьшается колво активных читателей, и если их число равно 0, то посыпается сигнал – можно писать. Проверка очереди гарантирует, что не будет бесконечного откладывания ????. Когда процесс писатель начинается, вызывается `startwrite()`. Бесконечное откладывание читателей исключается в `stopwrite()`. Эти функции есть в UNIX. Если писатель готов писать – надо дать ему эту возможность.

```

1 int readers = 0; //кол-во активных читателей
2 boolean writelock = false;
3 conditional canwrite, canread;
4
5 monitorentry void startread() {
6     //queue очередь ждущих писателей
7     if (writelock || queue(canwrite)) {
8         wait(canread);
9     }
10    ++readers;
11    signal(canread);
12 }
13
14 monitorentry void stopread() {
15     --readers;
16     if (readers == 0) {
17         signal(canwrite);
18     }
19 }
20
21 monitorentry void startwrite() {
22     if (readers > 0 || writelock) {
23         wait(canwrite);
24     }
25     writelock = true;
26 }
27
28 monitorentry void stopwrite() {
29     writelock = false;
30     if (queue(canread)) {
31         signal(canread);
32     } else {
33         signal(canwrite);
34     }
35 }

```

Рисунок 7.10 — listing

**ЛР №3** максимально использовать основное свойство набора семафоров – одной неделимой операцией можно изменять все или часть семафоров набора. Объ-

являем набор из трех семафоров. Наборы семафоров семантически реализованы как массивы. Два семафора – как считающие, и один – как бинарный.

- 1) Реализация производство-потребление на трех семафорах решение Дейкстры.
- 2) Читатели-писатели на семафорах и разделяемой памяти.

Монитор Хоара читатели – писатели под Windows под win32 без оконного интерфейса. Открываем книгу [4]. Можно посмотреть про потоки. Реализуем на event. В Windows – с автосбросом и со сбросом вручную. Потребуются оба типа. Нужно подумать, как использовать мьютекс.

Моделируем простейшую ситуацию, когда потоки разделяют одну глобальную переменную. Каждый писатель просто инкрементирует разделяемую переменную. При работе программы должно быть видно: какой писатель, что записал и что читают читатели. Чтобы они не работали последовательно, ставим рандомные задержки.

## 7.8 Проблема спящего парикмахера

(с точки зрения асинхронных процессов и ресурсов, которые эти процессы захватывают). Относится к классическим задачам в синхронизации. Аналогия – над гипотетическим одним парикмахером. Одно рабочее место и приемная с несколькими стульями. Когда парикмахер заканчивает подстригать клиента и идет в приемную, чтобы посмотреть, если ли ждущие клиенты. Если есть – то приглашает одного и начинает стричь. Если ждущих клиентов нет – то возвращается в свое кресло и спит. Каждый приходящий клиент смотрит, что делает парикмахер. Если спит – то будет его и садится в кресло, если парикмахер работает – то клиент идет в приемную, и если в ней есть свободный стул и занимает её. Если стула нет – клиент уходит. Нет критических ситуаций при беглом рассмотрении, но если рассмотреть подробнее, то можно выделить критические действия, связанные с тем, что действия клиентов и парикмахера занимают неизвестное кол-во времени и они - параллельные. Например: клиент входит и видит, что парикмахер работает, тогда он идет в приемную. Пока он идет со своей скоростью, а парикмахер заканчивает и идет в приемную быстрее чем клиент, придя – обнаруживают пустую приемную, идет спать, а клиент уверенный что парикмахер работает, идет ждать. Другой пример: два клиента могут прийти в одной и тоже время, когда в приемной есть один свободный стул. Они видят, что парикмахер работают и оба пытаются занять единственный стул.

Рандеву – особый тип взаимодействия. В классическом виде реализована в языке Ада. (применяется в банковском деле и в аэропортах). Работа с параллельными процессами в языке Ада ?? называется task. Task – специальный вид модуля, который может работать независимо от других модулей. Task имеют средство связи друг с другом. Task имеют спецификацию и тело. Задачи не могут быть отранслированы самостоятельно, они связываются между собой при помощи входов entry. Если одна

задача выдала обращение к входу и оно принимается другой задачей, то обе задачи теряют свою независимость, т.е. между ними устанавливается randevu. Пока randevu действует – они синхронизированы. В механизме randevu отсутствует симметрия. Если одна задача инициирует randevu, то вторая задача может принять, а может и не принять вызов первой задачи. Randevu происходит только тогда, когда задача, к которой произошел вызов – принимает вызов. Randevu начинается с согласования фактических и формальных параметров. Далее оно продолжается выполнением операторов, которые располагаются между ключевыми словами do – be end. Во время randevu task'и могут обмениваться информацией через список параметров. Последовательность операторов выполняется от имени обоих задач. В результате синхронизированные задачи получают результат выполнения последовательности операторов одновременно!!!!

**Итог:** рассмотрели блокирующие и освобождающие системные вызовы, в отличие от randevu Ады, всё что мы рассмотрели – блокирующие и ????. При этом в системах самым общим способом взаимодействия является передача сообщений. Система также предоставляет системные вызовы для передачи сообщений (send receive). Передача сообщений в самом общем виде.

**Проблемы при передаче сообщений** : рассмотрим два процесса, выполняющихся параллельно. Возьмем один процесс на одном компе, другой на другом. Они должны обменяться сообщениями. Один процесс выполнялся некоторое время. В какой то момент времени он посыпало сообщение-запрос другому процессу, который также выполняется и что то делает.

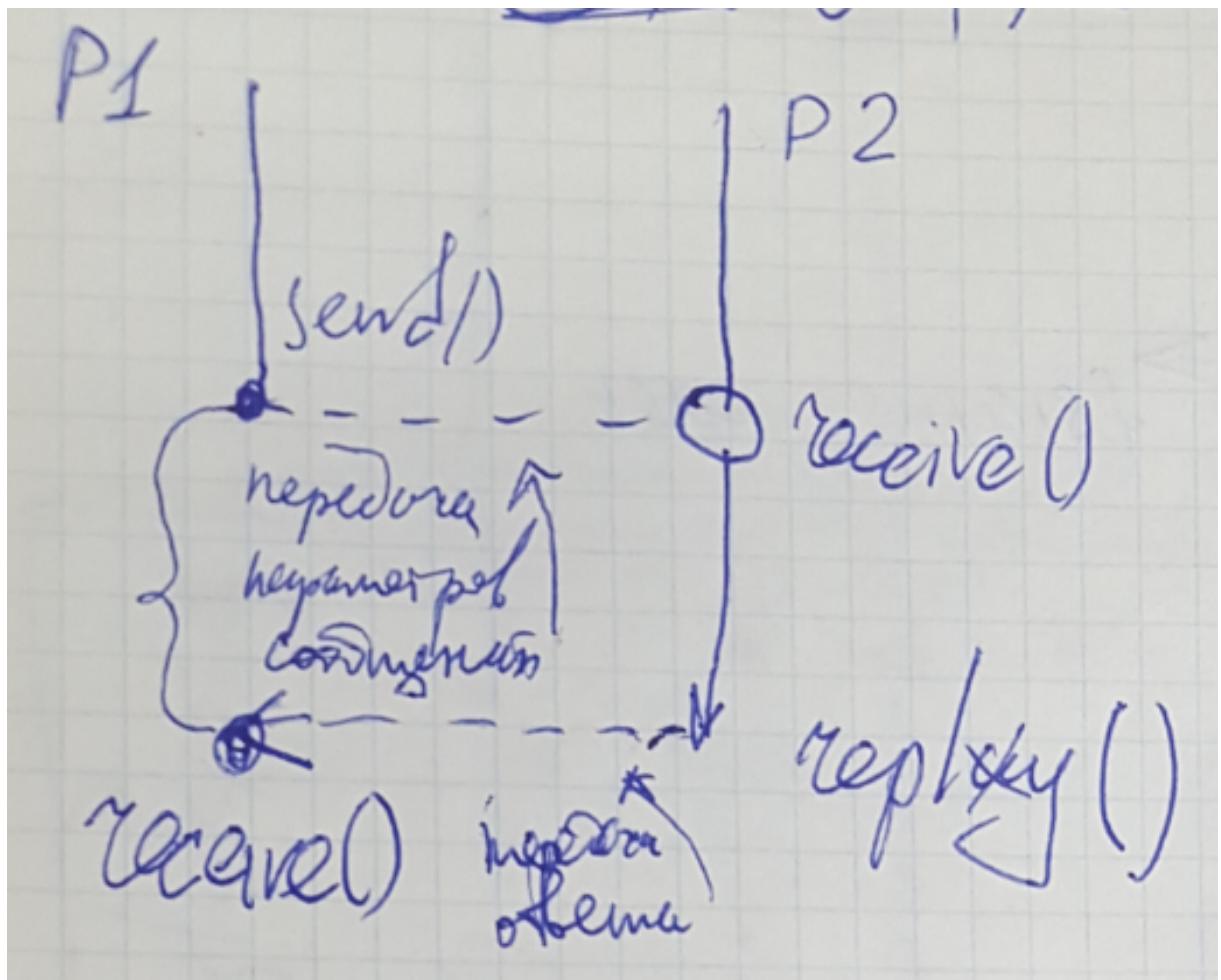


Рисунок 7.11 — pic

Выделяются три состояния блокировки процесса при передаче и приеме сообщений.

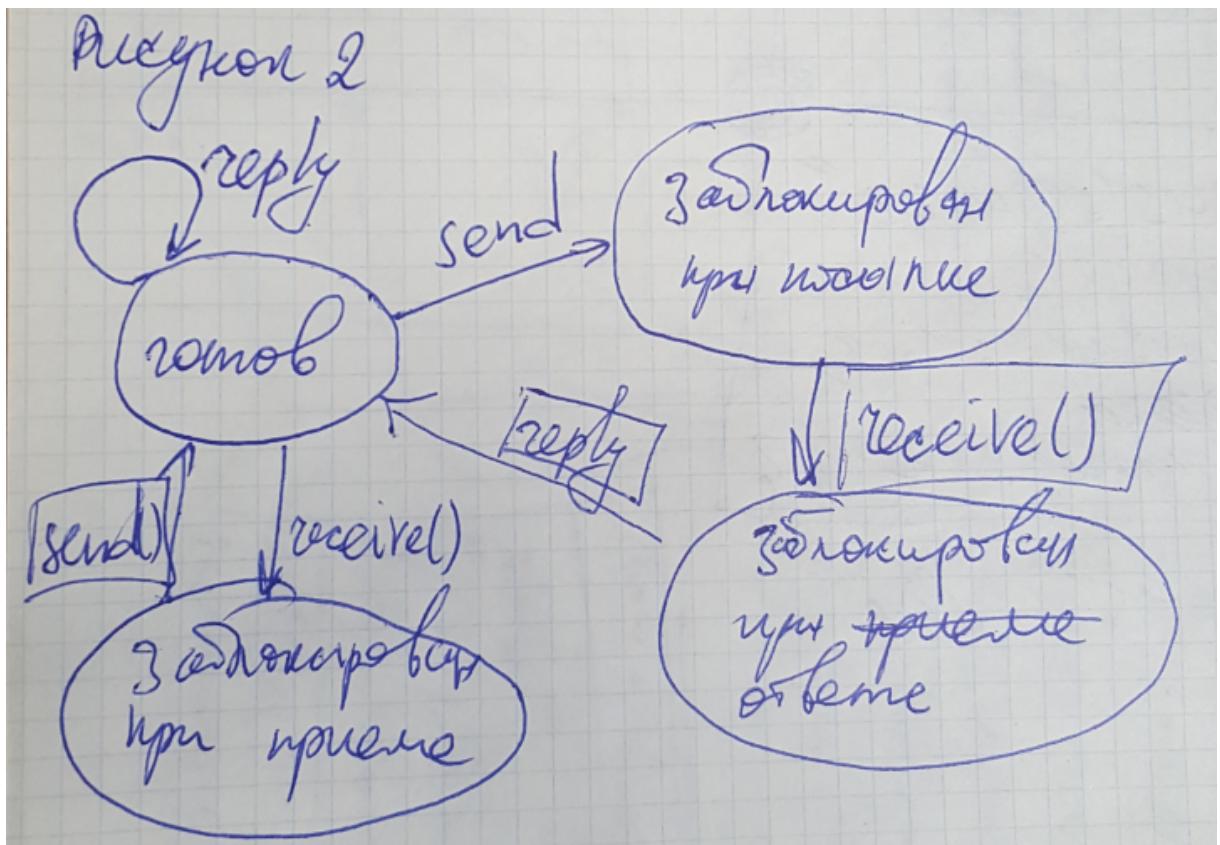


Рисунок 7.12 — диаграмма три состояния блокировки при передаче сообщения

На 7.12 в рамочке – системные вызовы, выполненные другим процессом. Все три состояния блокировки могут возникнуть при строгом протоколе взаимодействия, когда процесс должен ждать. Параметры в сигнатуре функций позволяет на этапе компиляции контролировать правильность типов.

Как только другой процесс вызовет `receive()`, процесс, пославший сообщение, будет разблокирован. Он будет заблокирован, пока не будет обработан и не вызван `replay()`.

## 8 Средство межпроцессного взаимодействия Unix

Синхронизация – связь между процессами, при котором один процесс не может выполняться, начиная с определенного места до тех пор, пока другой процесс не достигнет своей определенной точки. Пример: один процесс записывает данные в буфер. Оба процесса должны быть синхронизированы.

IPC (средство межпроцессного взаимодействия Unix).

### 8.1 Сигналы

Соответствуют определенным событиям в системе. Механизм сигналов позволяет процессам реагировать. Получение некоторым процессом сигнала означает завершиться. Реакция процесса на принимаемый сигнал зависит от того, как процесс определил свое поведение. У процессов есть для этого сигнальные маски. Потомок – наследует сигнальную маску.

Системный вызов `int kill(int pid, int sig)`. Если процесс вызовет `kill(getpid(), sigalarm)` – будет подан сигнал этому же процессу. Системный вызов `void (*signal(int sig, void (*handle)(int)))(int)` – реакцией на сигнал будет вызов `handle`. Сигналы посылаются группе процессов. При этом в зависимости от определенных значений в параметрах в соответствующих вызовов может быть разный состав групп, но вообще это процесс, вызвавший многократно `fork` создает группу (подчинение сверху вниз). Предок – создатель группы процессов. Т.е. процессы родственники могут получать в системе одни и те же сигналы. Важнейший сигнал – уничтожение процесса. Системный вызов `signal` возвращает сигнал на предыдущий обработчик данного сигнала в результате его можно использовать для восстановления обработчика сигнала, написав следующее:

```

1 #include <signal.h>
2 int main () {
3     void (*old_handler)(int) = signal(SIGINT, sig_IGN);
4     //действия
5     signal(SIGINT, old_handler);
6 }
7
8 #include <iostream.h>
9 #include <signal.h>
10 void catch_sig(int sig_num) {
11     signal(signum, catch_sig);
12     cout << "catch_sig" << sig_name << endl;
13 }
14 int main () {
15     signal(SIGTERM, catch_sis);
16     signal(SIGINT, SIG_IGN);
17     signal(SIGSEGV, SIG_DFL);
18     return 0;
19 }
```

Рисунок 8.1 — listing

Перехват сигнала SIGTERM, реакция на SIGSEGV должна быть установлена стандартным образом. Не рекомендуется использовать `signal` при разработке переносимых приложений, так как его поведение в System V отличается от поведения в BSD. POSIX – portable operating system interface based on unix – интерфейс переносимых ОС. POSIX.1 FIPS – федеральный американский стандарт. В BSD – классическое средство взаимодействия – сокеты. В Европе разработан x/open portability Guide (XPG3, XPG4) для разработки переносимого ПО. Содержат POSIX 1, POSIX 2, ANSI C. В POSIX определены системные вызовы SING LONG JUMP (получив определенный сигнал можно перейти в определенную точку ПО).

## 8.2 Семафоры

Unix поддерживают наборы считающих семафоров. Доступ к отдельному семафору – по индексу. Первый семафор – нулевой индекс. Семафоры в системе поддерживаются таблицей семафоров в ядре, одна на систему. В этой таблице отслеживаются все создаваемые в системе наборы семафоров.

Каждый дескриптор содержит следующие данные о наборе семафоров:

- имя (целое число, присваивается процессом, который создал набор). Другие процессы по этому имени могут открыть набор и получить дескриптор, чтобы получить доступ.

- б) UID (создателя набора и идентификатор его группы, процесс может удалять набор и изменять управляющие параметры, если его UID совпадает с UID в дескрипторе семафора)
- в) права доступа (для user, group, others)
  - г) кол-во семафоров в наборе
  - д) последнее время обращения
  - е) время последнего изменения управляющих параметров набора
  - ж) указатель на массив семафоров.

Операции над набором семафоров:

- а) `semget()` - создания семафоров;
- б) `semctl()` - изменение параметров;
- в) `semop()` - изменение состояния.

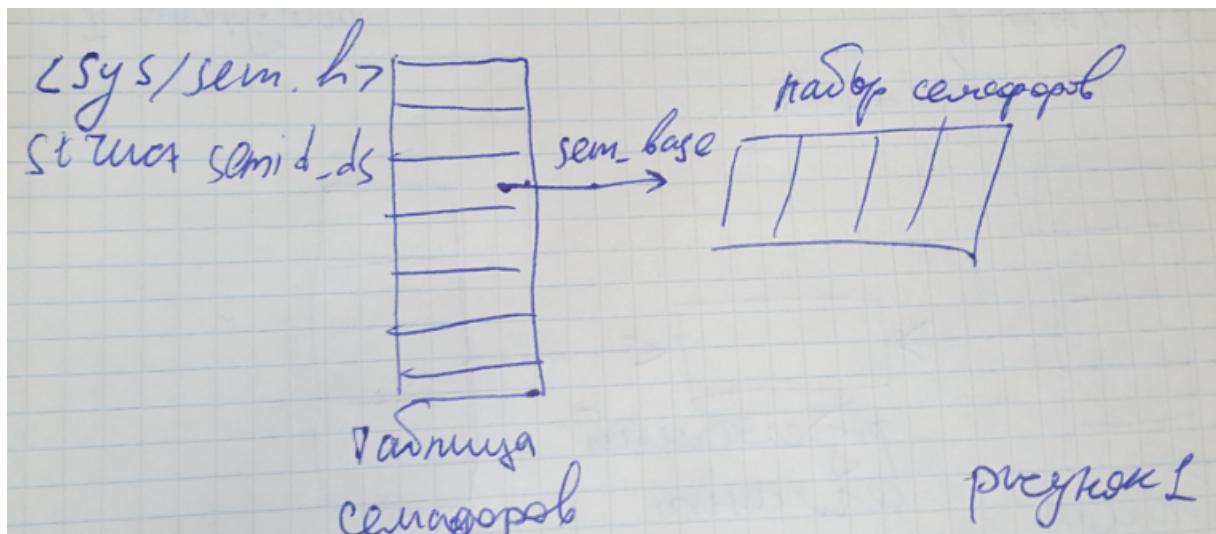


Рисунок 8.2 — pic

Листинг 8.1 — Структура `semid_ds`

```

1 struct semid_ds {
2     ushort sem_index;
3     short sem_op;
4     short sem_fe;
5 }
```

На семафорах определены три операции (в отличие от семафоров Дейкстры):

- а) `sem_op > 0` – инкремент - освобождает ресурс
- б) `sem_op = 0` – проверка освобождения ресурса без попытки захвата
- в) `sem_op < 0` – декремент – захват ресурса

```

1 #include <sys/types.h>
2 #include <sys/ipe.h>
3 #include <sys/sem.h>
4
5 struct semid_ds sbuf[2] = {{0, -1, SEM_UNDO | IPC_NOWAIT}, {1, 0, 1}};
6
7 int main () {
8     int perms = S_IRWXU | S_IRWXG | S_IROTH;
9     int fd = semget(100, 2, IPC_CREATE | perms);
10    if (fd == -1) {
11        perror("semget");
12        exit(1);
13    }
14    if (semop(fd, sbuf, 2) == -1) perror("semop");
15    return 0;
16}

```

Рисунок 8.3 — Демонстрируем работу набора семафоров

Для всех категорий пользователей устанавливаются полные права. Если системный вызов `semget()` выполнился успешно, то процесс вызывает системный вызов `semop()`, который передает массив структур `sbuf`, каждая структура определяет действие на одном семафоре наборе. На первом семафоре – декремент, на втором – проверяется на ноль. На семафорах допустимо использовать специальные флаги `IPC_NO_WAIT` – информирует ядро о нежелании процесса переходить в состояние ожидания. Наличие флага объясняется желанием избежать блокировки всех процессов в очереди к семафору, если захвативший семафор процесс завершится аварийно или получит сигнал `KILL`, который перехватить нельзя. В результате уничтожаемый процесс не сможет осуществить освобождение семафора. И если не предпринять соответствующих мер, то все ожидающие будут заблокированы навечно. `SEMONGO` – указывает ядру, что необходимо отслеживать изменения значения семафора операцией `semop`. И при завершении процесса система ликвидирует сделанные изменения, чтобы процессы небыли заблокированы навечно.

### 8.3 PIPE Программные каналы

Существуют программные каналы двух типов: именованные и не именованные. Поддерживаются, как специальные файлы.

- неименованные: не имеют идентификатора. Имеют дескриптор. Ими могут пользоваться только процессы родственники, так как при `fork` наследуются дескрипторы всех открытых файлов.
- именованные – имеют имя.

Программный канал является буфером в системной области памяти. Труба буферизуется на трех уровнях:

- a) В системной области памяти.
- б) В файловой системе помечается буквой р
- в) При переполнении системной памяти, буферы, имеющие большое время существования, переписываются на диск, при этом используются стандартные функции работы с файлами. Если процесс пытается записать больше 4096 байт, то труба буферизуется во времени.

#### **8.4 Файлы, отображаемые в памяти**

Связано с механизмом одноуровневой памяти. Рассмотрели схемы управления виртуальной памяти, наличие на диске специальной области свопинга. Туда вытесняются страницы из оперативной памяти.

Один и тот же файл может находить в системе в трех копиях:

- а) оперативной памяти;
- б) области свопинга;
- в) файловая система.

Если осуществлять свопинг в область адресного пространства файла, то дополнительная область свопинга не нужна. Этот подход был реализован в системе AS400/OS400 [5]. Этот механизм получил называние – файлы, отображаемые в памяти. В Windows экзешники выполняются, как файлы, отображаемые в памяти.

В прошлой лекции была ошибка. Надо так:

```

1 #include <sys/types.h>
2 #include <sys/ipc.h>
3 #include <sys/sem.h>
4
5 int semget(key_t key, int nsems, int semflag)
6 key: IPC_PRIVATE, IPC_CREATE

```

Рисунок 8.4 — listing

8.4 возвращает идентификатор, и его передаем системному вызову semop()

## 8.5 Программные каналы

Неименованный – файловой системе не виден, но для него создается дескриптор, который наследуется процессами потомками. В результате процессы родственники могут обмениваться сообщениями. Программный канал буферизуется на трех уровнях (В системной области памяти, но если память заполнена, то наиболее старые буфера отправляются на диск. Если больше 4096 байт, то он будет заблокирован)

```
mknod(name, IFFIFO ACCESS, 0);|
```

ACCESS должно быть определено, например 0666

Листинг 8.2 — Создание канала

```
1 # ls | ws -1
```

в результате 8.2 создается канал. Если программный канал не может быть создан, нам возвратят -1. В своих программах нужно проверять.

## 8.6 Сегменты разделяемой памяти

Особенность в том, что данное системное средство взаимодействия процессов, позволяет множеству процессов отображать системное адресное пространство выделенное под разделяемый сегмент на своё адресное пространство, т.е. созданный разделяемый сегмент подключается с помощью указателя к виртуальному адресному пространству процесса. РП была реализована как средство повышения производительности при обработки сообщений, как универсальное средство в силу того, что разделяемый сегмент подключается ???, при записи сообщения в разделяемый сегмент и при чтении – копирование не выполняется. Так данное средство позволяет осуществить обмен сообщениями быстро, но РП не имеет средств взаимоисключения.

Поэтому РП используются совместно с семафрами. В адресном пространстве ядра имеется таблица разделяемых сегментов (= РП).

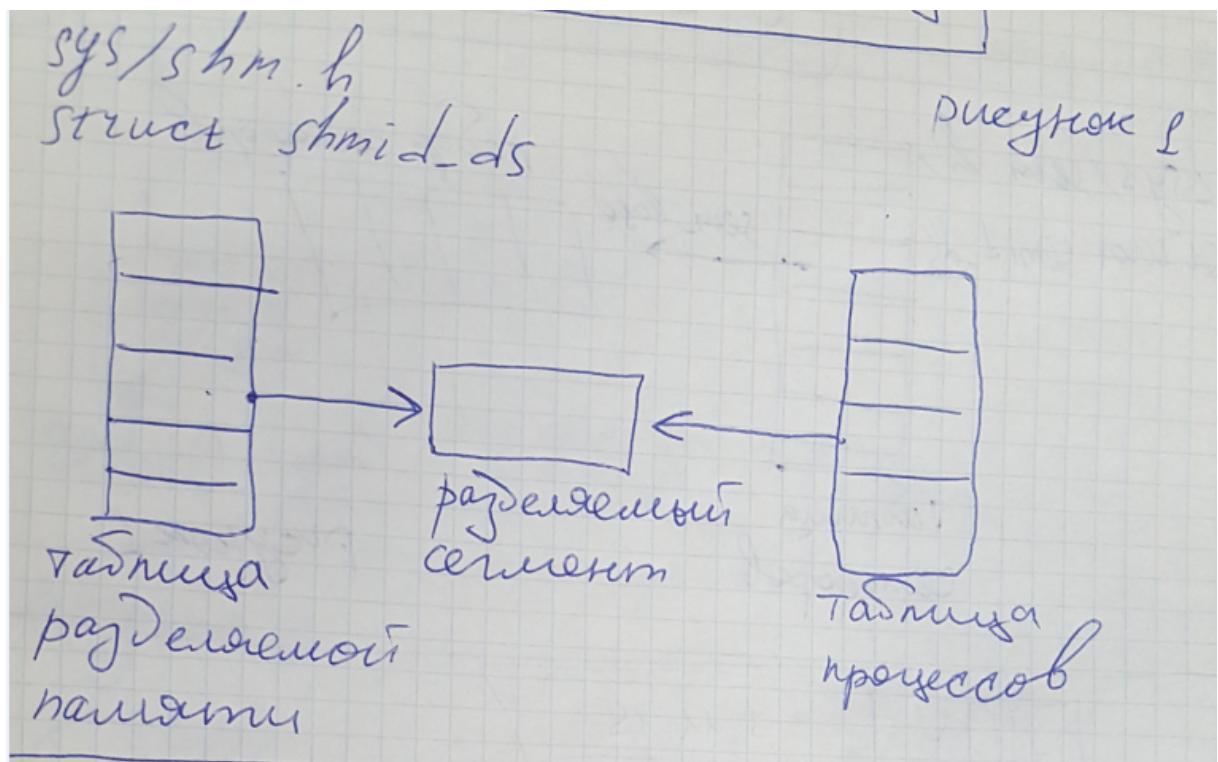


Рисунок 8.5 — pic

Процессы, дескрипторы которых в таблице процессов, получают указатель на разделяемый сегмент ???.  
`shmget()`;

Изменить параметры созданного разделяемого сегмента поможет `shmctl()`;  
`shmat()` `shmdt()` - ???.

```
1 int *buf;
2 if (shm_id = shmget(IPC_PRIVATE,
3     (MAX_SLOTS + 2) * sizeof(int), 0600) == -1)
4 {
5     return fprintf(stderr, "shmget");
6 }
7 buf = (int *) shmat(shm_id, NULL, 0);
8 if (buf == (int *) -1) {
9     perror("shmat");
10    exit(1);
11 }
```

Рисунок 8.6 — listing

```

1  char *buf;
2  if (shmid = shmget(IPC_PRIVATE,
3      (MAX_SLOTS + 2) * sizeof(int), 0600) == -1)
4  {
5      return fprintf(stderr, "shmget");
6  }
7  buf = (char *) shmat(shmid, NULL, 0);
8  if (buf == (char *) -1) {
9      perror("shmat");
10     exit(1);
11 }
12 strcpy(buf, "Hello");

```

Рисунок 8.7 — listing

Таблица 8.1 — В системе есть ограничения на создание РП

Системное ограничение	значение
SHMMNI	Максимальное число разделяемых сегментов, которые могут существовать одновременно
SHMMIN	Минимальный размер разделяемой области памяти в байтах
SHMMAX	

## 8.7 Очереди сообщений

У pipe должен быть и читатель и писатель. Сложные правила взятия сообщения из очереди. Система поддерживает системную таблицу очередей сообщений.

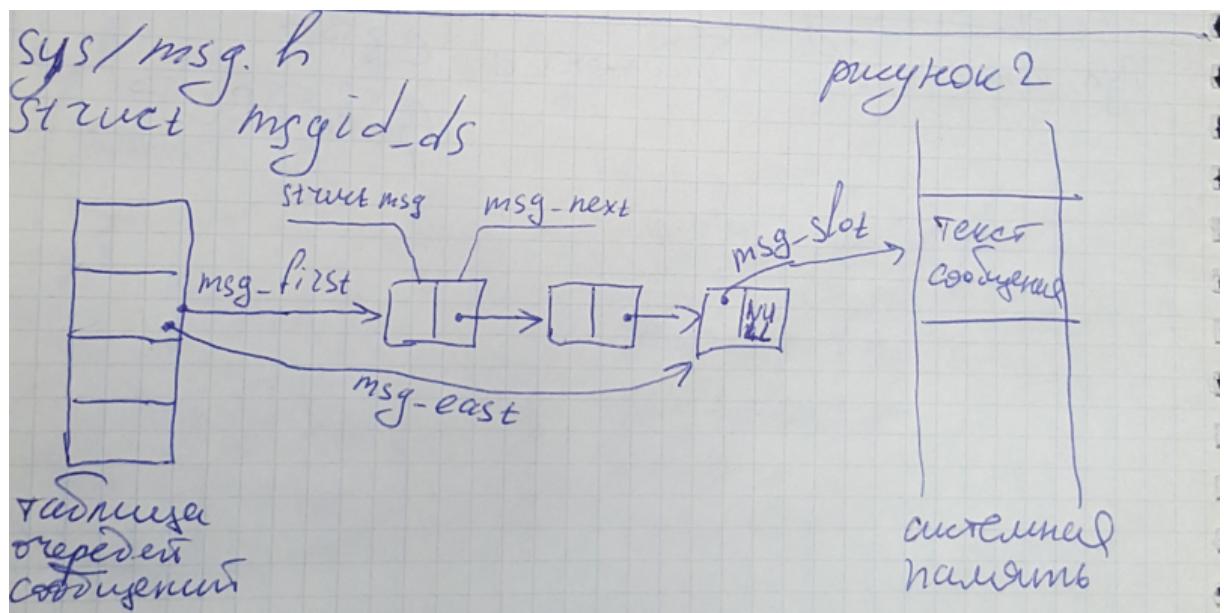


Рисунок 8.8 — pic

Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает её в конец связного списка записей соответствующих сообщений указанной очереди. В каждой такой записи указывается тип сообщения, размер в байтах, указатель на область данных ядра системы, в которой фактически находится сообщение. Ядро копирует сообщение из пространства отправителя в область данных ядра системы, после чего процесс отправитель может завершиться. Сообщение остается доступным для чтения другим процессам. Когда какой либо процесс выбирает сообщение из очереди, ядро копирует его в адресное пространство этого процесса, после чего запись удаляется (сообщение - потребляемый ресурс, когда получено – перестает существовать).

Процесс может выбрать сообщение из очереди несколькими способами:

- Взять самое старое сообщение, независимо от его типа.
- Если идентификатор сообщения совпадает с идентификатором, который указал процесс. Если существует несколько сообщений с таким идентификатором, то берется самое старое. (идентификатор = тип)
- Выбрать сообщение, числовое значение типа которого ??? меньшее или равное значению типа, указанному процессом.

Самое главное, процесс, отправивший сообщение, может завершиться. Процесс, заинтересованный в получении сообщения, может взять его несколькими способами, Он может гибко действовать с сообщениями и он не будет блокирован. Так, если вернуться к диаграмме трех состояний блокировки процесса при передачи сообщений, используя очередь сообщений, мы можем избавить процесс от блокировок. Но цена – получаем не то, что нужно, ???.

```

1 struct msg_buf {
2     Long mtype;
3     char mtext[MSGTXTLEN];
4 } msg;
5 int msgid = msgget(IPC_PRIVATE, MSGPERM | IPC_CREATE | IPC_EXCL);
6 int rc = msgsnd(msgid, &msg, sizeof(msg.mtext), 0);

```

Рисунок 8.9 — listing

Windows поддерживает разделяемые сегменты. Определены файлы, отображаемые в память, и установкой параметров можно регламентировать его как РП. Файлы отображаемые в память позволяют отображать адресное пространство в оперативную память. Для того, чтобы избавиться от области свопинга. Windows поддерживает POSIX.

## 9 Синхронизация в распределенных системах

Рассмотрим работы программы make OC UNIX. В Unix большие программы разбиваются на несколько файлов. Очевидно, что изменения, вносимые в один файл, ??? компиляция только этого файла. Программа make проверяет время последней модификации всех исходных файлов и всех объектных файлов программы. Если исходный файл имеет время последней модификации больше, чем время последней модификации объектного соответствующего файла, то make считает, что он был изменен и перекомпилирует его. Рассмотрим ситуацию в распределённой системе. Есть несколько компов. На одном мы редактируем, на другом – перекомпилируем.

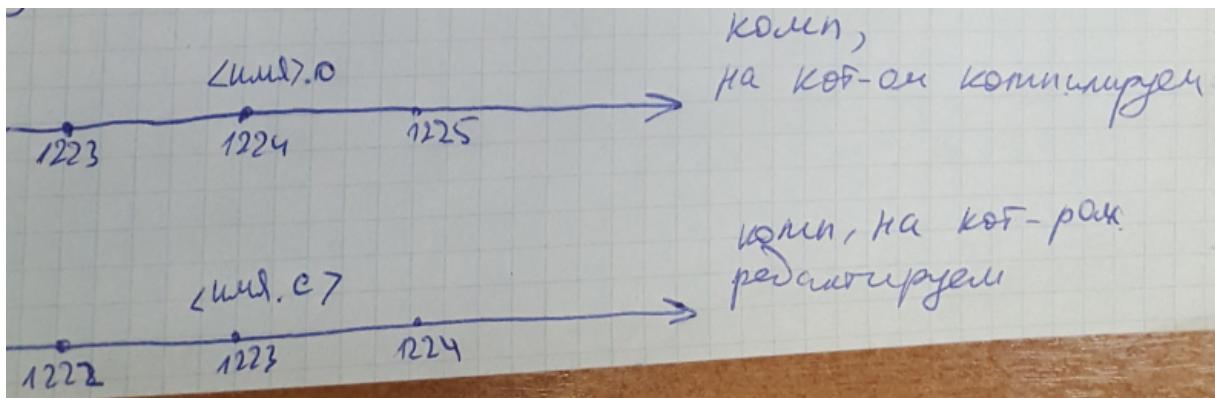


Рисунок 9.1 — pic

Связано с локальными часами. На компах кварцевые генераторы. Любое техническое устройство имеет определенные характеристики, в том числе кварцевый генератор имеет допустимую ошибку генерации тиков. Погрешность –  $10^{-5}$ . Если таймер генерирует 60 прерываний / секунду, то в час он сгенерирует 216 000 тиков, конкретная машина может выдать значения тиков в следующем диапазоне: 215996 – 216002 за час. Вывод – не можем синхронизироваться по локальным часам.

Способ синхронизации – часы. Работают на импульсах тактового генератора. Они имеют точность, поэтому имеют свойство спешить или отставать. В развитых странах есть служба точного времени. Интервал между 2 последовательными солнечными переходами называется солнечным днем. В результате измерений была определена солнечная секунда (min solar second). В 1948 году были изобретены атомные часы, обладающие высокой стабильностью, и было определено, что за одну секунду будет совершено 9192631770 переходов цезия133. 50 лабораторий мира имеют такие часы. Международное бюро усредняет эти данные и выдает время по атомным часам. TAI - International Atomic Time. Средний солнечный день усредняется. Атомная секунда не меняется. Полдень будет наступать раньше, чем реальный полдень. В результате было решено использовать потерянные секунды. Всякий раз, когда разница возрастает до 800мс, то добавляется секунда. Так перешли на UTC. Большинство электрических компаний положили в основу это универсальное согласованное время UTC. Когда бюро объявляет потерянную секунду, то частоту герцевых часов меняют с 60->61 или 50->51. В США есть институт точного времени. Они имеют коротковолновую станцию. В начале каждой секунды осуществляют рассылку.

Один комп оборудован приемником WWV, остальные подводятся. Алгоритмы синхронизации:

- a) Cristian's
- б) Berkeley
- в) усредняющие

В распределенных системах стоит задача синхронизации. Представим локальную сеть, на каждом компе есть локальное время, но они подводят 9.2.

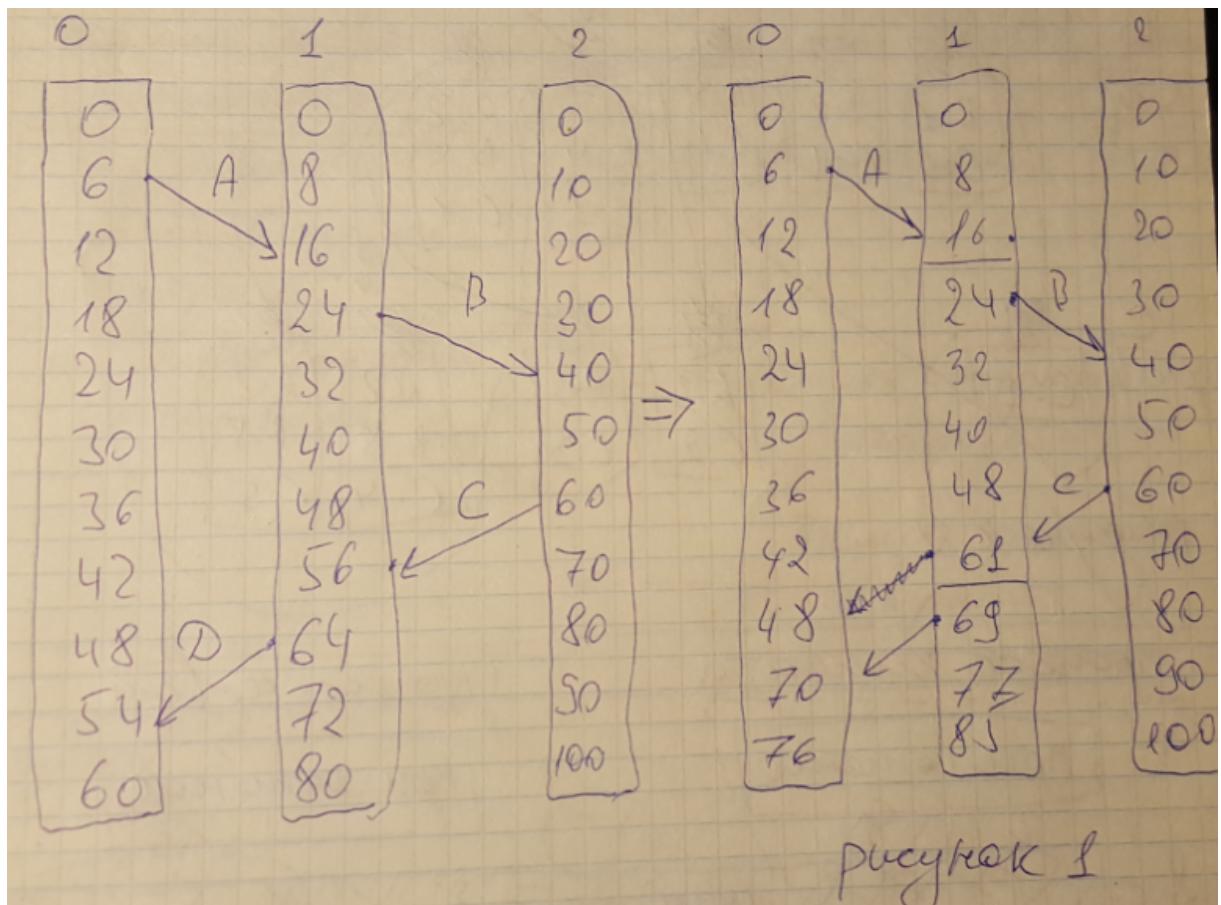


Рисунок 9.2 — pic

Есть алгоритмы синхронизации по логическим часам. Предложил в 1978 году Лампорт. Должно выполняться соотношение: случилось до – случилось после. Если событие  $A$  произошло до события  $B$ , то  $tA < tB$ . Это мы гарантировать не можем. Лампорт предложил при взаимодействии параллельных процессов, которые обмениваются сообщениями добавлять в это сообщение время отправки по локальным часам. Процесс, который получает сообщение, смотрит, а какое его локальное время и устанавливает свое время равное времени пришедшего в сообщении + 1, если его время меньше.

Проблема – ситуация, когда время двух событий одинаковое. Возникает неоднозначность. В 1988 году разработан алгоритм векторные часы. Передаем не просто значения локального счетчика и корректировать собственный локальный счетчик, тут формируется вектор, на 9.3 показан процесс передачи сообщений тремя процессами и показано, что для последующего состояния системы состояния  $B:4$  – исходное. Это не время, это событие (посылка и отправка сообщения). Каждое сообщение содержит локальный счетчик.

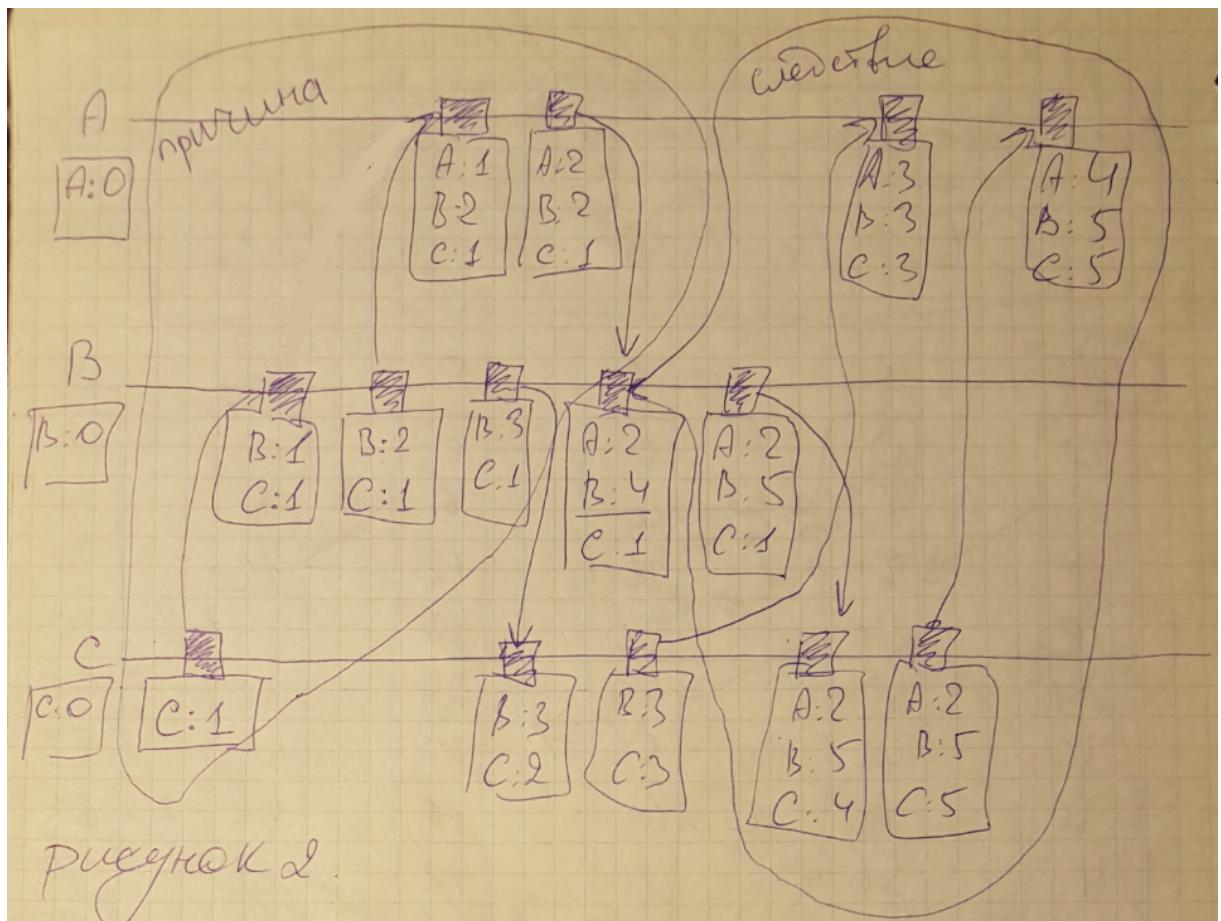


Рисунок 9.3 — алгоритм векторные часы

В распределенных системах возникает задача разделения ресурсов.

## 9.1 Алгоритмы взаимного исключения

### 9.1.1 Централизованный алгоритм (забияки)

с выбором нового координатора. Самый надежный. Предполагает наличие некоторого процесса координатора. Обычно в качестве процесса координатора выбирается процесс, который имеет наибольший сетевой номер. Когда какой-то процесс хочет войти в свою критическую секцию он посыпает сообщение запрос координатору, указывая критическую секцию, в которую он хочет войти и ждет от координатора разрешения. Если в этот момент ни один процесс не находится в критической секции, то координатор посыпает сообщение разрешение. Если некоторый процесс уже находится в критической секции, то никакой ответ не посыпается, а сообщение запрашивающего процесса ставится в очередь. Затем по мере освобождения критической секции просматривается очередь и процессу будет послано сообщение разрешение. Координатор координирует вход в критическую секцию. Чтобы сохранить работоспособность системы, при обнаружении любым из процессов отсутствие координатора этот процесс инициализирует выборы нового координатор путем посылки

сообщения-запроса со своим номером. Если номер процесса получившего сообщение запрос о начале выборов больше, то он посыпает назад подтверждение прием и сам инициализирует новые выборы. В результате выполнения такое цепочки действий выбирается новый координатор и им становится процесс с большим сетевым номером.

### 9.1.2 Распределенный алгоритм

Когда процесс хочет войти в свой критический участок по конкретному ресурсу он формирует сообщение, в котором указывает идентификатор критической секции, свой номер и своё локальное время. Это сообщение посыпается остальным процессам. При этом предполагается, что получение такого сообщения надежно, оно подтверждается получением сообщения. Когда процесс получает такое сообщение запрос, его действия зависят от того, в каком состоянии относительно критической секции он сам находится.

Допустимы три ситуации:

- а) процесс получивший сообщение не находится и не собирается входить в данную критическую секцию, тогда он посыпает процессу, приславшему сообщение, ответ с разрешением;
- б) процесс, получивший сообщение, уже находится в критической секции по данному ресурсу. В этом случае он не посыпает ничего и ставит это сообщение в очередь;
- в) процесс, получивший сообщение сам хочет войти в данную критическую секцию, тогда он сравнивает временную отметку в полученном сообщении с временной отметкой его собственного сообщения. Если время поступившего к нему сообщения меньше его собственного времени (его запрос возник позже) то он посыпает сообщение разрешение, иначе ничего не посыпает и ставит пришедшее сообщение в очередь.

Процесс может войти в критическую секцию, если получит N-1 сообщение-разрешение от других процессов. Если любой из процессов завершился, то уже не получить N-1 сообщение.

### 9.1.3 Алгоритм Token ring

Формируется логическое кольцо, в котором каждый процесс знает номер своих соседей. 1984 год фирма IBM. Token – специальное сообщение, которое создается для конкретной критической секции. Такой token циркулирует по кольцу. Когда процесс получает token от своего соседа, он анализирует, не требуется ли ему войти в данную критическую секцию, если нет – он сразу посыпает token дальше, если ему

нужно войти в критическую секцию, он входит в неё и удерживает token. Если у нас  $M$  критических ресурсов, в кольце будет циркулировать  $M$  токенов.

## 9.2 RPC (вызов удаленных процедур)

??? Для какой-то функции от своего имени. RPC (вызов удаленных процедур) был предложен юниксоидами и этот механизм предназначен для удаленного доступа, реализован таким образом, что вызов этой удаленной функции выполняется также как вызов локальной функции. Такое взаимодействие выполняется по схеме клиент-сервер. Процесс, вызывающий RPC, является клиентским, а процесс, который выполняет РПЦ функцию – является серверным. RPC является низкоуровневым средством взаимодействия, от этого не становится менее интересным. У RPC есть особенности, главное из которых является то, что механизм RPC скрывает от пользователя этот механизм удаленного доступа. Пользователь RPC может не заморачиваться на то, что он обращается к удаленной машине. РПЦ появились в 80-е.

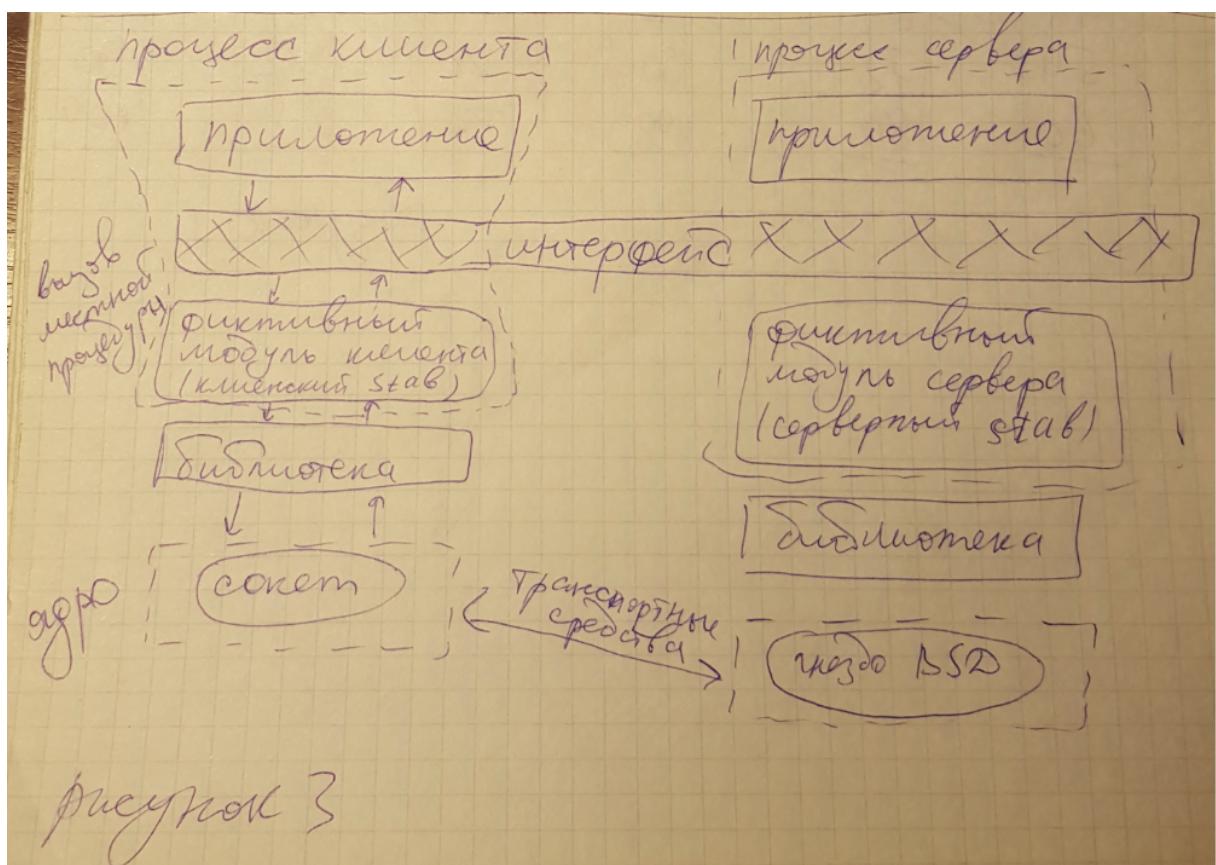


Рисунок 9.4 — Методология RPC

После того, как клиентский стаб (заглушка, нереальный вызов) вызван программой клиентом, Он заполняет буфер отправленным сообщением, затем параметры этого сообщения преобразуются в определенный формат и формируется пакет/несколько пакетов. После чего осуществляется переключение в режим ядра, сохраняется контекст процесса, ядро копирует сообщение/пакет в свое адресное пространство

и выполняет отправку пакета серверу. На стороне сервера полученное сообщение распаковывается, ядро определяет, какому серверному стабу адресовано это сообщение. Если такой стаб на стороне сервера имеется, то выполняется копирование пакета в его буфер. После чего выполняется переключение контекста, который произошел в результате вызова серверным стабом системного вызова `receive`. После того, как контекст восстановлен, сообщение восстановлено серверным стабом, приложение переходит к выполнению этого запроса RPC и формирует ответное сообщение.

Проблемы, возникающие при таком взаимодействии, является указание в клиентском приложении сетевого адреса сервера. Такой подход имеет недостаток, т.е. его крайнюю не гибкость, а именно, сервер может быть перемещен или может быть увеличено число серверов, кроме того, может быть изменен интерфейс. В результате всех этих изменений необходимо заново перекомпилировать программу, которая использует жесткий адрес. Чтобы избежать такие проблемы, в некоторых распределенных системах существует динамическое связывание (биндинг).

### 9.3 Неделимые транзакции

Транзакция – последовательность операций над одним/несколькими объектами базы данных, которые переводят систему из одного целостное состояние в другое целостное состояние. Модель неделимой транзакции пришла из бизнеса. Один процесс объявляет, что хочет начать транзакцию с одним или более процессов. Инициатор транзакции объявляет, что он хочет завершить транзакцию. Если все процессы с ним соглашаются, то результат фиксируется. Если один/более процессов отказываются/потерпели крах, тогда все изменения возвращаются к исходному состоянию. Системные вызовы: `begin`, `abort`, `and`, `read`, `write`. Транзакции обладают свойствами упорядоченности, неделимости, постоянством. Неделимость – промежуточные результаты не видны.

Существует два глобальных подхода к реализации неделимых транзакций:

- a) Создание для процессов, участвующих в транзакции, индивидуального рабочего пространства, в которых должны находиться копии всех необходимых файлов и объектов, участвующих в транзакции. Все изменения, выполняемые в процессе транзакции, выполняются в этих копиях. Исходные файлы и объекты не изменяются в процессе транзакции. Если транзакция фиксируется, то сделанные изменения копируются в исходные объекты и файлы. Недостаток: большое количество копий.
- б) Список намерений. Заключается он в том, что модифицируются сами файлы, не существует дополнительных копий. Все изменения вносятся в журнал регистрации. Там отмечается, какая транзакция делает изменение, какой файл меняется, какой блог редактируется, а так же старое и новое значения. Только после успешной записи в журнал изменяется сам файл. Если транзакция фиксируется, об этом

делается запись в журнале регистрации, но старые значения все равно сохраняются. Если транзакция прерывается, то инфо из журнала используется для приведения файла в исходное состояние. В распределенных системах транзакция может потребовать взаимодействие процессов на разных машинах. На каждой машине хранятся ????. Для достижения этого в распределенных системах используется протокол двухфазной фиксации транзакций. Это самый распространённый протокол. Суть протокола: один из процессов выполняет функции координатора. Координатор начинает транзакцию и делает запись в своем локальном журнале. После этого он посыпает подчиненным процессам, участвующим в транзакции, сообщение «приготовиться».

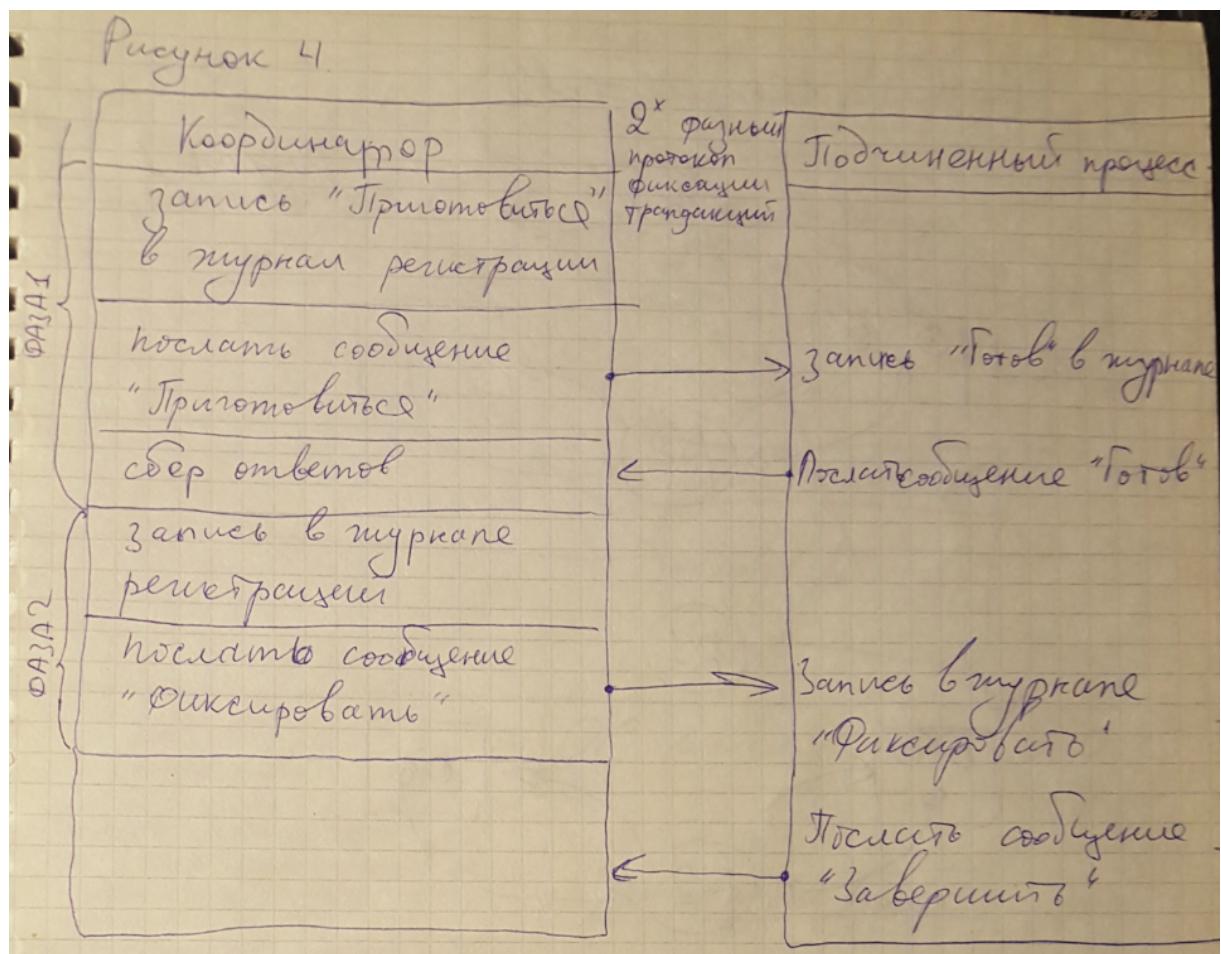


Рисунок 9.5 — pic

#### 9.4 Тупики

Теория тупиков в [6]. Глава про тупики, автор доказывает ряд теорем, которые позволяют обнаруживать тупики.

Таблица 9.1 — Тупик

Процесс №1	Процесс №2
Запрос ресурса P1	
Получение ресурса P1	
	Запрос ресурса P2
	Получение ресурса P2
Запрос ресурса P2	
	Запрос ресурса P1

Тупик – ситуация, возникающая в результате монопольного использования ресурсов, когда процесс, владея ресурсом, запрашивает другой ресурс, занятый непосредственно/через цепочку запросов процессом, который ожидает освобождения ресурсов, занятых первым процессом.

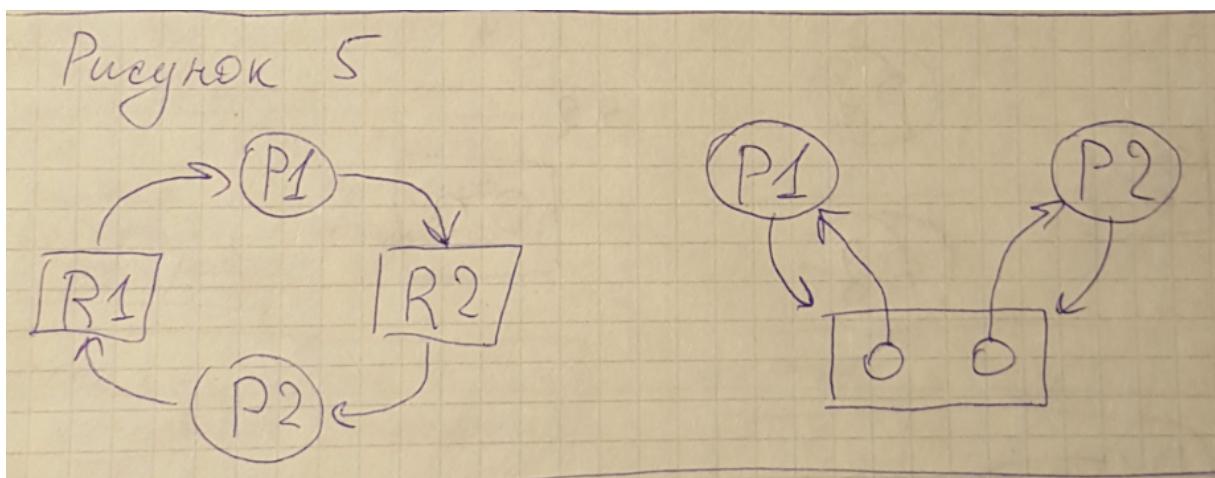


Рисунок 9.6 — Граф Тупика для двух единичных ресурсов

Типы ресурсов:

- а) потребляемые ресурсы (сообщения, когда получено, перестает существовать);
- б) повторно используемые ресурсы (аппаратные ресурсы системы, системные таблицы, реентерабельный код).

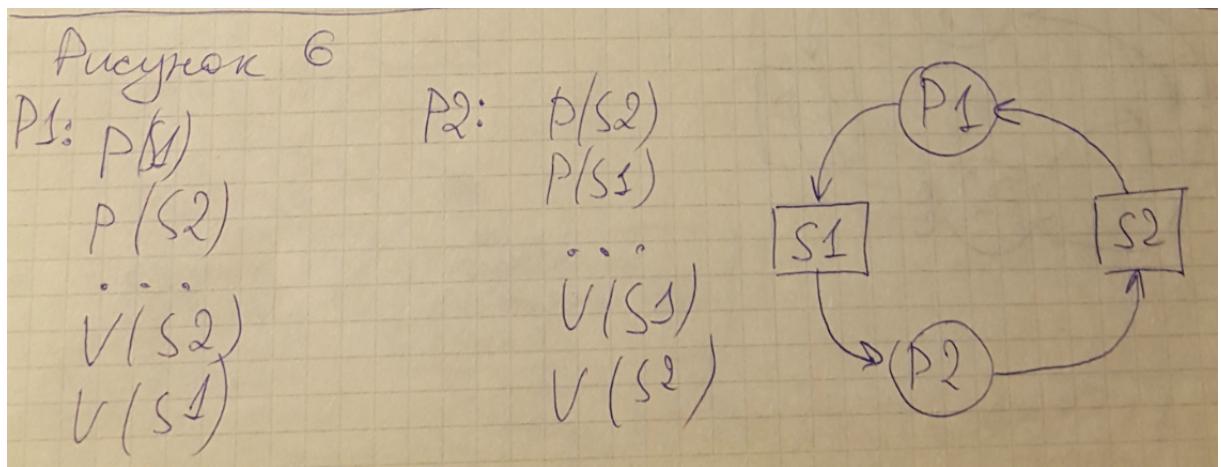


Рисунок 9.7 — рис

Условия возникновения тупика:

- а) взаимоисключение, когда процессы монопольно используют предоставляемые им ресурсы;
- б) ожидание, связано с тем, что процесс удерживает занятые им ресурсы и ожидает предоставления доп. ресурсов для возможности продолжения выполнения;
- в) неперераспределяемость, у процесса нельзя отобрать ресурсы до завершения процессов или до того момента, когда процесс сам освободит занимаемые ресурсы;
- г) круговое ожидание, возникает замкнутая цепь запросов процессов к ресурсам, в котором каждый процесс занимает ресурс, необходимый следующему процессу в цепочке, для продолжения выполнения.

### Борьба с тупиками (3 метода)

- недопущение, создание в системе такой ситуации, когда тупики в принципе невозможны.

Одна из причин должна быть устранена. Ханвендер в своей работе доказал, что если устраниТЬ хотя бы одну причину, то тупик не возникнет. Согласно этой стратегии существует три основных подхода:

1. Если процесс сразу запрашивает все необходимые ему ресурсы. Так было в первых системах. Бесконечное откладывание!

2.

- обход, тупики возможны, но в системе прикладываются усилия, чтобы их обойти;

- обнаружение тупиков, они возникают, и задача — обнаружить, а после — восстановить работоспособность системы;

- недопущение ???

В тупик попадают взаимодействующие процессы.

2. Упорядочиваем ресурсы. Ресурсы делятся на классы. Каждому такому классу присваивается номер и выполняется следующее правило: процессы могут запрашивать ресурсы из классов с номерами большими, чем номера классов, к которым принадлежат классы, которые уже удерживают. Если он запрашивает ресурс с номером меньше, чем те, которые он уже удерживает, то он должен освободить все, что удерживает и запросить всё заново. Так он вырождается в первый способ (ему нужно всё опять запросить разом).

3. У процессов можно отбирать ресурсы, устраниет свойство неперераспределимости ресурсов. Если процесс, в процессе запроса, не может получить ресурс, то он должен освободить уже занятые ресурсы. Этот подход может привести к запросу и освобождению одних и тех же ресурсов (трешинг).

**Обход тупиков**. Единственным классическим подходом к обходу тупиков является алгоритм банкира, предложенный Дейкстрой. Банкиры дают заём. Задача банкира – дать заём тем, кто может их вернуть, но часто заемщик не может вернуть заем сразу, да и то, ему нужны еще займы. Банкир – менеджер системы. Заемы – процессы, делающие заявки на ресурсы, заявка процесса должна отображать максимальную потребность в ресурсе каждого класса. В процессе выполнения, процесс не может затребовать ресурсов больше, чем указал в заявке. Кол-во процессов – фиксировано. В системе должна быть информация, для этого определяются специальные структуры, при этом процесс запрашивает ресурсы по единице. МР гарантирует, что в системе тупик не возникнет. Каждый запрос проверяется по отношению к кол-ву ресурсов данного типа, имеющихся в системе. Анализируя ситуацию, МР ищет такую последовательность процессов, которая с учетом максимальных потребностей в ресурсе данного типа может завершиться. Если такая последовательность есть, то система не находится в тупике и состояние системы надежно, в результате запрошенные единицы ресурса выделяются процессу.

Таблица 9.2 — Надежное состояние

процессы	текущее распределение	свободные единицы	заявка
p1	1		4
p2	3		5
p3	5		9
		2	

9.3 является надежной и безопасной относительно тупика, так как в таблице имеется последовательность процессов, которая может завершиться. Если текущее

состояние системы надежно, то все последующие состояния системы будут надежными. Например, если система из состояния в 9.3 перейдет в следующее состояние ??

Таблица 9.3 — Надежное состояние

процессы	текущее распределение	свободные единицы	заявка
p1	2		4
p2	3		5
p3	5		9
		1	

В такой ситуации нет последовательности. Но процессы могут и не запросить максимально заявленное кол-во ресурсов. Состояние системы является безопасным, если существует последовательность процессов такая, что первый процесс в последовательности обязательно завершится, так как если он запросит максимально заявленное кол-во ресурсов, то в системе хватит ресурсов. Второй процесс может завершиться, если первый процесс завершится и вернет занимаемые им ресурсы, и в сумме этих ресурсов будет достаточно для удовлетворения потребностей. И так далее. Когда процесс делает новый запрос, менеджер ресурсов должен найти последовательность успешно завершенных процессов, и только в этом случае запрос будет удовлетворен. Запрос процесса, переводящий систему в ненадежное состояние – откладывается. Так как система постоянно поддерживается в надежном состоянии, то за конечное время все запросы будут завершены и все процессы смогут завершить свою работу, но каждый раз требуется исследовать  $n!$  последовательностей, что предполагает большие затраты на выполнение данного алгоритма. Таким образом, алгоритма банкира имеет теоретическое значение. Это важно для систем реального времени, которые управляют внешними по отношению к системе процессами. Чаще всего это системы автопилота, ставятся в автомобили. Не нужно путать систему реального времени с системой разделения времени.

Предлагаются подходы аппроксимации алгоритма банкира. Наиболее известная, это

**алгоритм Габермана**. МР поддерживает массив:

$S[0..r - 1] // r$  - число единиц ресурса

$S[i] = r - 1$  для всех  $i : 0 \leq i \leq r$

заявил - С, удерживает - Н, и запрашивает 1, то  $S[i]$  декрементируется для всех  $i : 0 \leq i \leq C - H$ . Если какое то из  $S[i]$  становится отрицательным, то такое состояние системы становится опасным, относительно тупика. Затратность этого алгоритма значительно меньше, чем  $N^2$ .

**Третий способ обнаружения тупиков** : опасная цепь запросов обнаруживается с помощью графов. Для обнаружения тупиков используется модель Хомта, представляет собой двудольный (бихроматический) граф, разбивается на два подмножества, множество вершин процессов, множество вершин ресурсов. Вершины соединяются дугами, причем одна дуга не соединяет вершины одного и того же множества. Граф направленный. Дуга, которая выходит из вершины подмножества вершин ресурсов к вершине из подмножества вершин процессов, называется выделением ресурса. И дуга, которая выходит из вершины процесса и входит в вершину ресурса называется запросом. Тупики обнаруживаются методом редукции графа. Редукция заключается в ??? дуг графа. Если все вершины становятся изолированными, то система не находится в тупике.

Рисунок 1

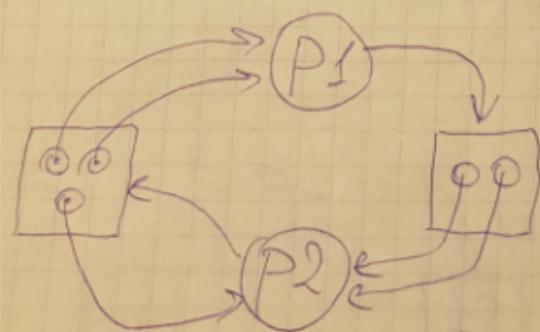
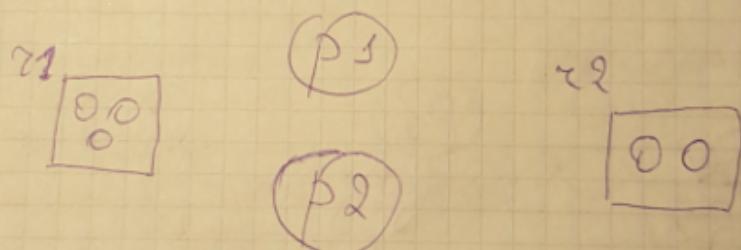
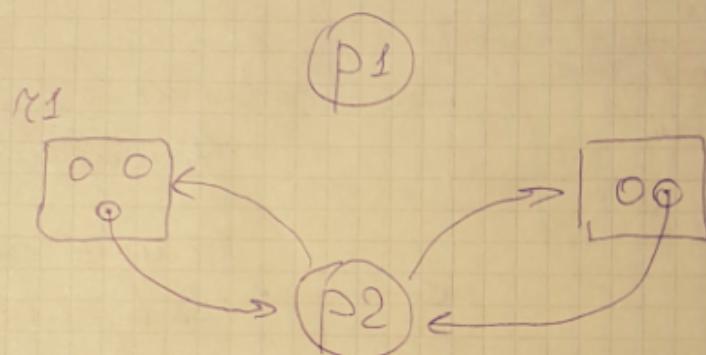
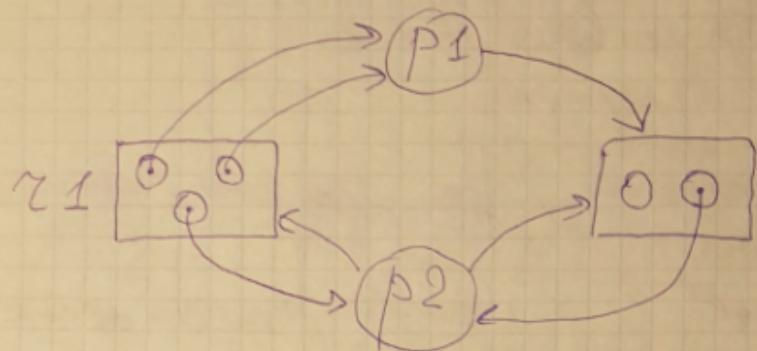


Рисунок 9.8 — pic

Данный граф 9.8 можно сократить по вершине 1, так эта вершина становится изолированной.

Шоу было доказано ряд теорем:

- a) Граф является полностью сокращаемым, если существует такая последовательность сокращения, которая устраняет все дуги. Если граф нельзя полностью сократить, то анализируемое состояние является тупиковым.
- б) Цикл, в графе повторно используемых ресурсов, является необходимым условием тупика.
- в) Если состояние  $S$  не является состоянием тупика, то состояние  $T$ , в которое система переходит из  $S$  и  $T$  есть состояние тупика, то в ??? когда операция процесса  $P_i$  является запросом.  $P_i$  находится в тупике в состоянии  $T$ . Т.е. тупик в системе может возникнуть только в результате запроса.

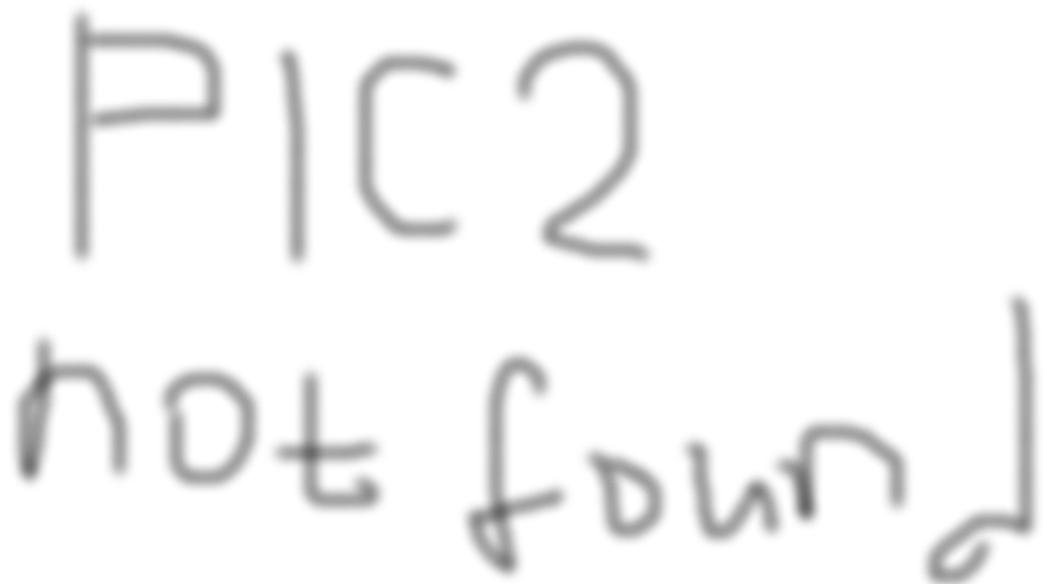


Рисунок 9.9 — демонстрирует несокращаемый граф

Представление графов в системе: двудольный бихроматический граф может быть представлен двумя матрицами, матрицей запросов  $Ap, r$  ( $a_{ij}$  – запрос  $i$  процесса на единицу  $j$  ресурса) и матрицей распределения  $Br, p$  ( $b_{ij}$  – кол-во единиц  $i$  ресурса выделенное  $j$  процессу). Можно также делать связный список. Очевидно, что и матрицы, и списки должны быть монопольно используемыми, это представление двудольного графа.

Алгоритмы обнаружения:

1. Метод прямого обнаружения заключается в просмотре по порядку матрицы и там где это возможно производится сокращение, действия аналогичные сокращению дуг графа, до тех пор, пока нельзя будет сделать еще сокращений. Процессы, оставшиеся после всех сокращения, находятся в тупике. Для самого плохого случая, когда

сокращения выполняются в порядке, обратном следованию процессов, число проверок будет  $n * (n + 1)/2$ . Каждая проверка требует просмотра  $N$  ресурсов, таким образом время для выполнения данных действий будет пропорционально  $m * n^2$

2. Алгоритм с вектором свободных ресурсов  $F = [..f_j]$ , где  $f_j$  – кол-во свободных единиц ресурсов. Тогда кол-во распределенных единиц  $j$  ресурсов, и кол-во свободных единиц  $j$  ресурса равно кол-ву единиц этого ресурса в системе.

$$r_j = f_j + \sum_j b_{ji}$$

Основан на анализе запросов путем сравнения векторов. Пусть имеется два вектора  $C$  и  $D$ .  $C \leq D : ci \leq di$  для  $1 \leq i \leq n$ . Когда ??? и строка запросов этого процесса меньше или равна вектора  $F$  то такой запрос может быть удовлетворен.

Обнаружение тупиков выполняется методом редукции графа (сокращения графа). Сократить дугу можно в том случае, если запрос процесса может быть удовлетворен. В результате редукции могут образоваться изолированные вершины, и если все вершины графа изолированные, то система не находится в тупике. Если остались не сокращенные вершины, то они в тупике. Мы уже писали три теоремы. Тупик – замкнутая цепь запросов, может возникнуть только в результате запроса. Обнаружение тупиков может представляться в виде матрицы распределения и матрицы запросов. Также необходимо иметь инфо о свободных ресурсах системы (вектор свободных единиц ресурса  $F$ )  $F = [\dots f_i \dots]$ ,  $f_i$  - количество свободных единиц  $i$  ресурса. Если просуммировать все выделенные единицы  $i$  ресурса и сложить со свободными единицами ресурса, то получим кол-во единиц данного ресурса в системе. Первый лобовой алгоритм – метод прямого обнаружения. Просмотр по порядку матрицы запросов, и там где можно, производятся сокращения. Более эффективный – на анализе векторов. Пусть имеется два вектора  $C$  и  $D$ .  $C \leq D$  (каждый элемент вектора  $C$  меньше или равен элементу вектора  $D$ ). Тогда, если  $i$  процесс запросил  $j$  ресурс и при этом строка запроса  $i$  процесса меньше или равна вектору  $F$ , то такой запрос может быть удовлетворен. Процесс, запрос которого может быть удовлетворён, может завершиться, и освободить занимаемые им ресурсы, и сократить граф по вершине данного ресурса. Более эффективно, если хранить дополнительную информацию о запросе. Для каждого ресурса хранятся запросы, упорядоченные по размеру. Для каждого процесса  $i$ , определяется счетчик ожидания, обозначим  $W_i$ , этот счетчик содержит число типов ресурсов, которые вызвали блокировку процесса (счетчик отражает число типов ресурсов, на которых процесс заблокирован) Среди заблокированных процессов ищется цикл запросов. Алгоритм Бенсусан и Мерфи описан в книге Медника-Донована. В книге – особенности реализации упр. памятью, процессорами, процессами, взаимодействие процессов, проблема тупиков. Книга не устарела, так как архитектура компов – не изменилась.

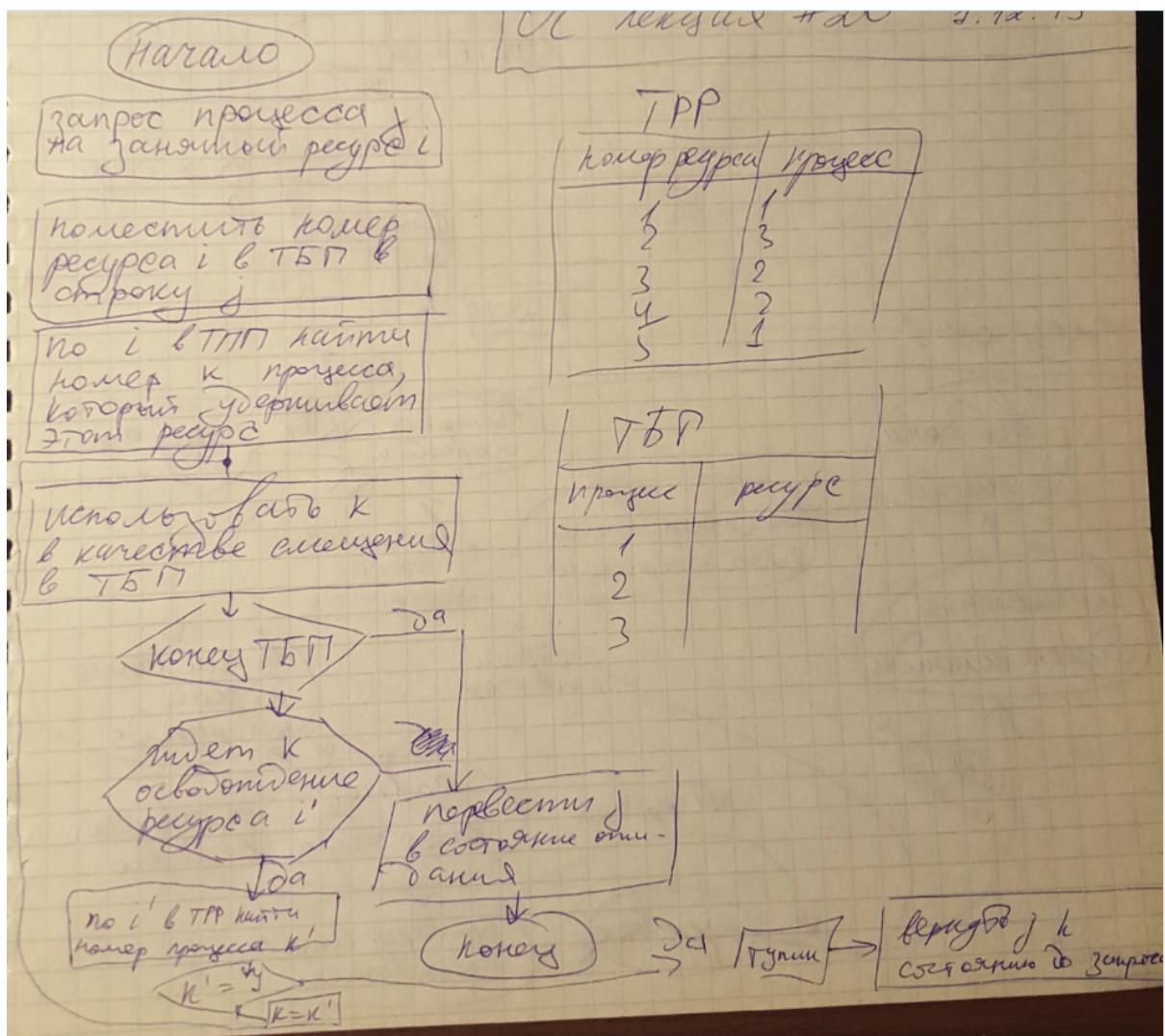


Рисунок 9.10 — pic

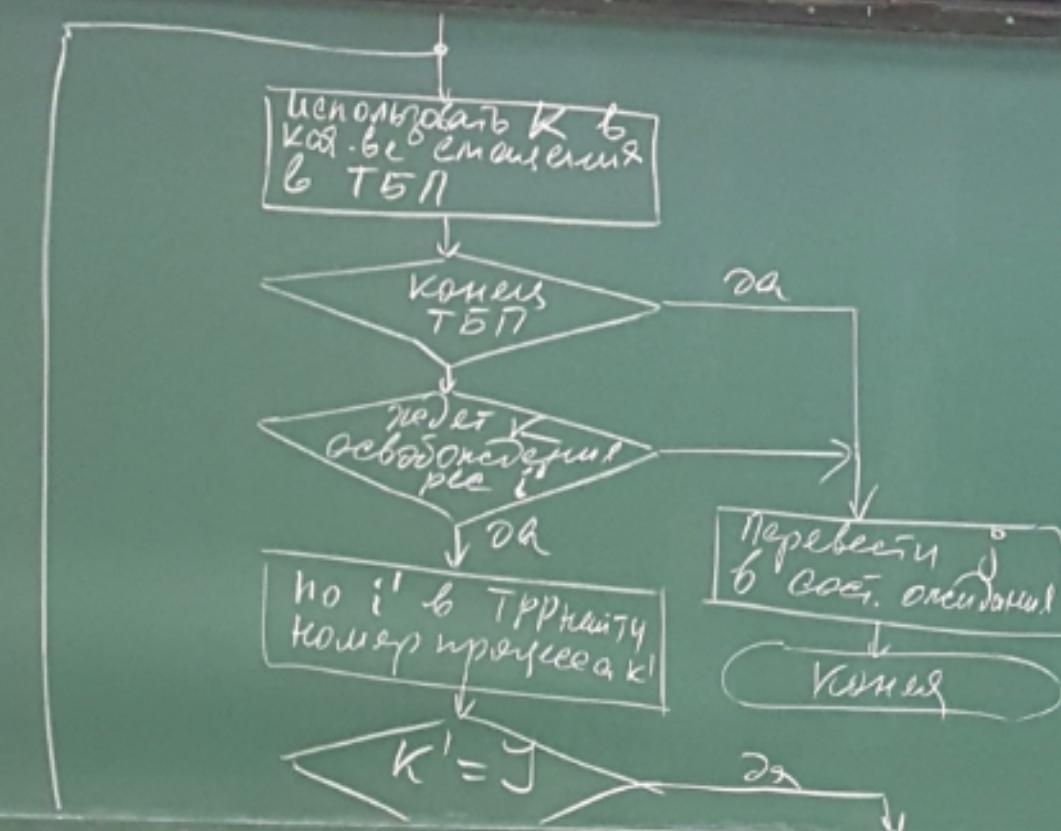
# Kazano

Запуск процесса  
на замкнутой реальности

Поместить коммерческую реальность в ТБР  
в сценарий

Но и в ТБР находит  
коммерческий процесс  
который управляет  
этот реальный

TPP	
коды реальности	реальность
1	1
2	3
3	2
4	2
5	1



$$K = K'$$

Типик

вернуть K  
сост.-но до запроса

ТБП – таблица блокированных процессов. ТРР – таблица распределенных ресурсов.

Таблица 9.4 – tbl

1	p1 запросил и получил r1
2	p2 запросил и получил r3
3	p3 запросил и получил r2
4	p2 запросил и получил r4
5	p1 запросил и получил r5
6	<p>p1 запросил и получил r3  <math>j = 1, i = 3 \rightarrow k = 2</math>          блокируем процесс 1 на ресурсе 3</p>
7	<p>p2 запросил и получил r2  <math>j = 2, i = 2 \rightarrow k = 3</math>          блокируем процесс 2 на ресурсе 2</p>
8	<p>p3 запросил и получил r5  <math>j = 3, i = 5 \rightarrow k = 1</math>          по к в ТБП видимо, что <math>i' = 3 \rightarrow k' = 2 \rightarrow</math>  <math>k = 2 \rightarrow i' = 2 \rightarrow k' = 3 \rightarrow</math> обнаружили замкнутую цепь запросов</p>

У алгоритма много ограничений. Ресурсы – в единственном экземпляре. Сток – любая вершина из подмножества процессов, которая не имеет ребер, выходящих из неё. Т.е. процесс получил все ресурсы и у него нет запросов. Последовательное исключение ребер, направленных в сток. Если длительное время не выполняются необходимые действия, то можно предположить, что система в тупике. На мейнфрейме такое предположение более реально по сравнению с распределенной системой, так как в РС задержки непредсказуемы. Если обнаружили, что система в тупике, то выполняются действия, которые мы обсудили. Другой подход. Если тупик возник в результате запроса, то можно анализировать состояние системы по ??. Для непрерывного обнаружения анализируется каждый запрос. При этом надо проверить, не возникает ли при этом запросе цикл по вершине  $r_i$ . Два подхода выхода из тупика:

- Прекращение выполнения процессов. Приводит к потере проделанной процессами работы. Определить перечень процессов, попавших в тупик и последовательно завершать их. Освобождаются ресурсы этих процессов. Можно за основу брать приоритет процессов, но со временем он уменьшается, в результате можем завершить процесс, который долго работал и сделал много работы. Другой подход – выключить процесс, который не попали в тупик.
- Перехват ресурсов. Тупик возникает в результате запроса на отсутствующий свободный ресурс. Если перехватить нужные ресурсы у других процессов, то их можно выделить тупиковым процессам. Чтобы

перехватить ресурс, нужно фиксировать состояния процессов и возвращать какой-то процесс к состоянию до получения конкретного ресурса. Какие системы вообще не могут попадать в тупик? – реального времени. Это системы, управляющие внешними процессами или объектами. Существует два аспекта: надежность и живучесть. Система должна уметь перестраиваться.

## **10 Архитектура вычислительных систем с точки зрения ядра**

Рассмотрим монолитное и микроядро.

Существует две структуры ядер:

- a) Монолитное – программа, имеющая модульную структуру. Т.е. состоит из подпрограмм. Единственным способом, изменить функциональность ядра, это перекомпиляция ядра.
- б) В микроядерной архитектуре все компоненты ОС являются самостоятельными программами и возможно выполняются в разных адресных пространствах. Взаимодействие между такими модулями ОС выполняется по модели клиент – сервер. Взаимодействие осуществляется путем передачи/приема сообщений. Основная функция ядра – осуществление взаимодействия процессов с помощью сообщений.

## 10.1 Монолитное ядро

Переходим к подробному изучению монолитного ядра. Примеры таких систем: Windows. Windows 2000 – не является ОС на основе микроядра в классическом понимании (стр. 25). ОС Матч реализует крошечное ядро микроядра, но оно не эффективно. В Unix/Linux ядро - минимизировано. Для ОС с монолитным ядром характерна система прерываний.

Принято различать такие синхронные события:

а) системные вызова (выполняются в процессе выполнения программы, которой требуется сервис системы; частыми являются обращение процессов к внешним устройствам (если речь идет об интерактивных процессах, то это клавиатура и мышь))

б) исключения:

1) исправимые (страничные прерывания, которое возникает при отсутствии страницы в физической памяти, но в результате обработки этого исключения, вызывается менеджер памяти, который загрузит её в память)

2) неисправимые, приводят к завершении программы с сообщением об ошибке.

Асинхронные события в системе: - аппаратные прерывания. Большую группу составляют прерывания от внешних устройств ввода/вывода. Такое прерывание возникает, когда устройство завершает операцию ввода вывода.

В шинной архитектуре (также есть канальная) внешними устройствами управляют контроллеры или адаптеры. Контроллер – устройство находящееся в внешнем устройстве, а адаптер – на материнской плате. Контроллер – программно управляемое устройство, в нем имеется набор регистров и некоторая логика. Контроллер получает от процессора команду, выполняя которую, контроллер берет на себя управление операцией ввода/вывода. По завершении операции ввода/вывода контроллер посыпает на вход контроллера прерываний сигнал. В шине передаются: данные (собственно данные и команды), адреса (обеспечивают обращение к данным или командам), сигналы управления. В наших системах два адресных пространства (??? и портов ввода/вывода, это подтверждаются командами IN/OUT).

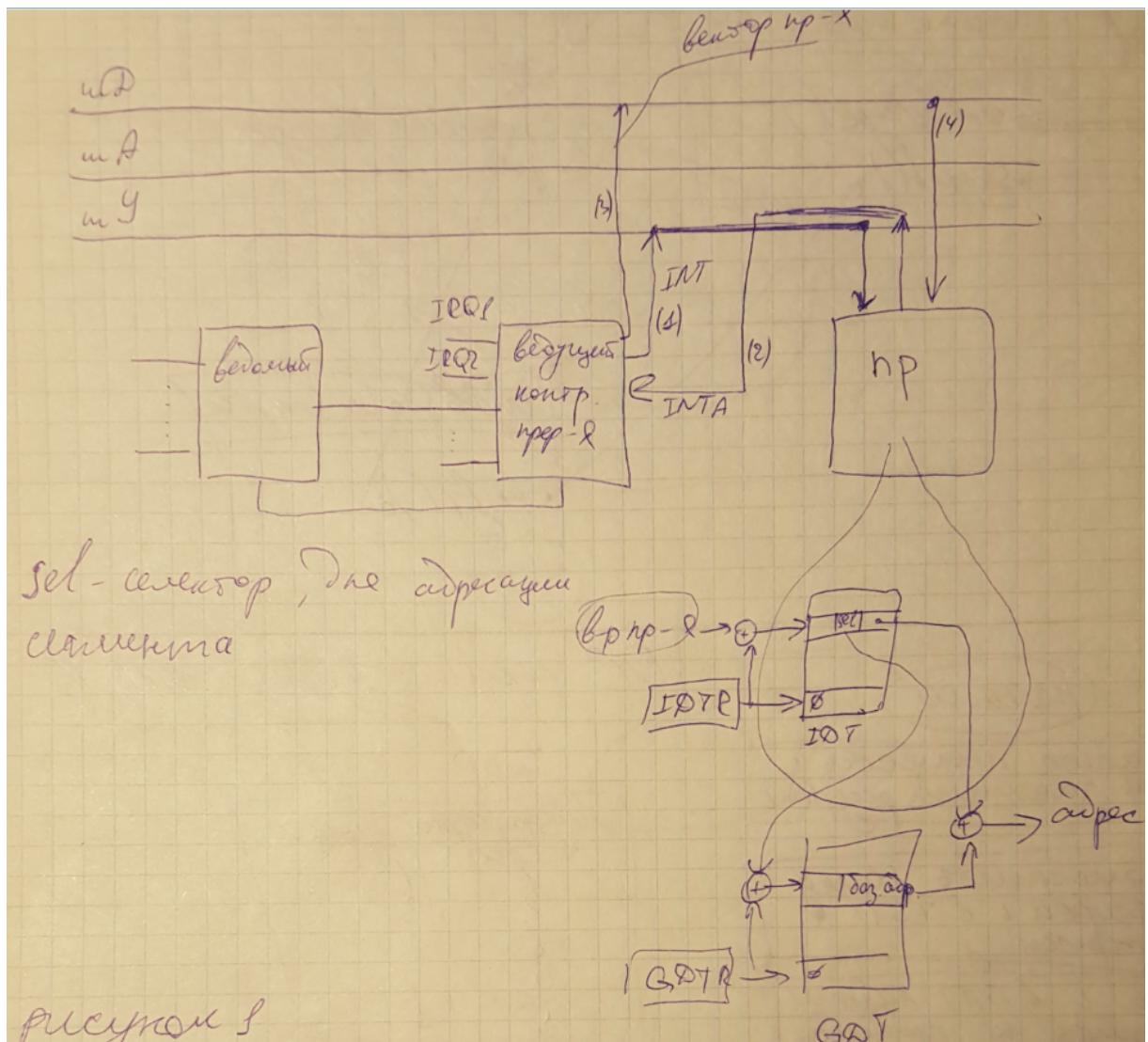


Рисунок 10.1 — pic

Прерывание системного таймера – единственное независимое от процессора действие в системе. В многопроцессорной системе разделения времени на таймер возлагается задача декремента кванта (найти инфу про то, что частота большая, а тики всего лишь 18 раз в секунду)

Процессор проверяет наличие прерывания на своей ножке в конце цикла выполнения каждой команды.

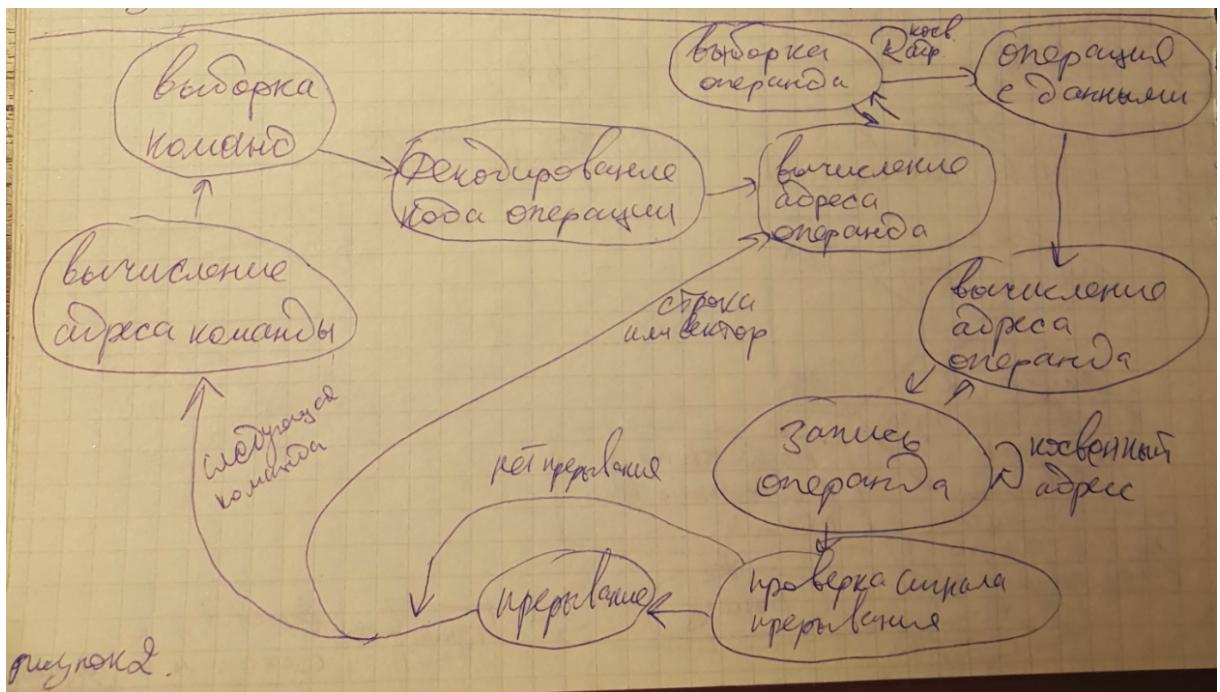


Рисунок 10.2 — рис

Если пришло прерывание, то сохраняется аппаратный контекст выполняемой программы. В счетчики команд устанавливается адрес обработчика прерываний. Система переключается в режим ядра. Обработчики прерываний в системе это одна из точек входа драйвера. Это программа, которая в системе управляет работой внешнего устройства до какого то момента. Предназначен, чтобы послать и сформировать команду для устройства. Другая задача драйвера – получить от устройства данные и вернуть их процессу, который запросил эти данные. Драйверы – многовходовые программы. Один из точек входа – обработчик прерывания. Ни одна программа не может на прямую обратиться к устройствам ввода/вывода. Сделано из соображений безопасности. Иначе любой процесс может обратиться в любое место ОС, в частности к системным таблицам.

Оси позволяют изменять функциональность без перекомпиляции ядра. В windows реализовано с помощью иерархии драйверов (стек драйверов). Можно написать свой драйвер, зарегистрировать в системе, и система будет обращаться к нему. В этом стеке есть разные типы драйверов. На нижнем уровне – функциональный драйвер. Пишется разработчиками устройств. Только они знают формат данных. Имеется драйвер фильтр нижнего уровня и драйвер фильтр верхнего уровня. Изменив соответствующие коды в драйверах – фильтрах, можем изменить функциональность устройства. Linux предоставляет возможность написания драйверов, загружаемые модули ядра.

## 10.2 Микроядерная архитектура

Микроядро – модуль ОС обеспечивающий взаимодействие между процессами, диспетчеризацию процессов, первичную обработку прерываний и низкоуровневое управление памятью. Микроядро реализует взаимодействие с аппаратным уровнем вычислительной системы (внешние устройства или ОЗУ). Все остальные – самостоятельные процессы, работающие, возможно, в разных адресных пространствах. ОС состоит из набора программ, эти процессы должны взаимодействовать с процессами пользователей и друг с другом. Такое взаимодействие выполняется по модели клиент – сервер. Процессы ОС предоставляют сервис пользовательским процессам, а также могут предоставлять сервис друг другу. Взаимодействие таких процессов выполняется с помощью передачи сообщений.

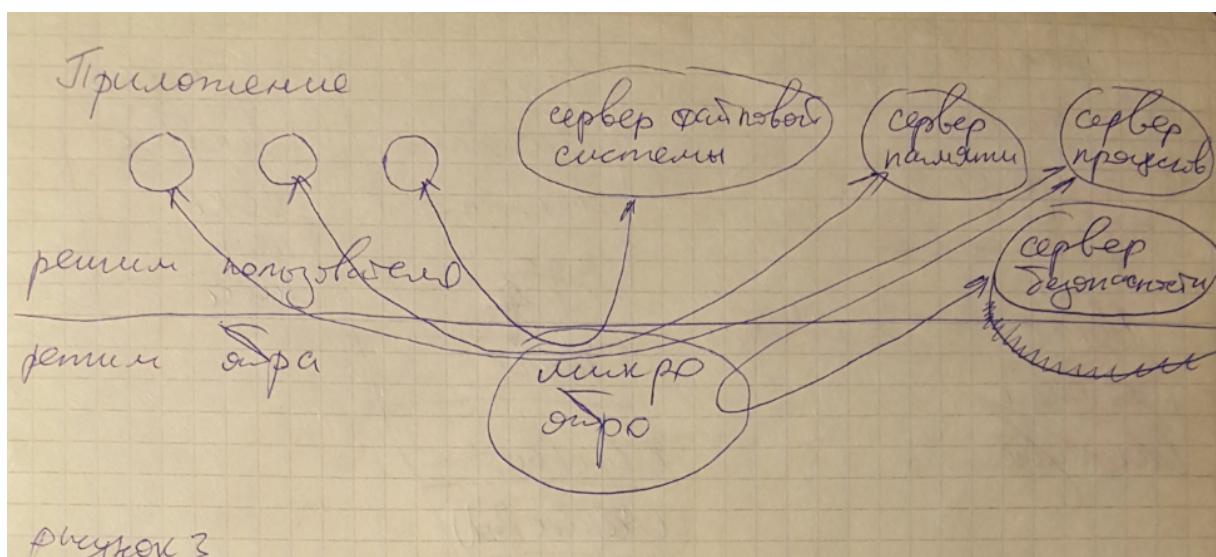


Рисунок 10.3 – pic

Большую часть работы, которую выполняет ОС выполняют отдельные программы, выполняющиеся в режиме пользователя.

В режиме пользователя:

Сервер процессов: постановка процессов в очередь, пересчет приоритетов. Таким образом, в микроядре остаются низкоуровневые функции.

Эффективность микроядерной архитектуры: см. диаграмму трех состояний блокировки процесса при передаче сообщений. Там сказали, что время этих блокировок нельзя предсказать, это вероятностные вещи. Очевидно, что передача сообщений связана с неконтролируемыми задержками. А так же контролируемыми временными затратами, в частности в микроядерной архитектуре больше переключений в режим ядра (как минимум в 2 раза).

рисунок 4

микроядро

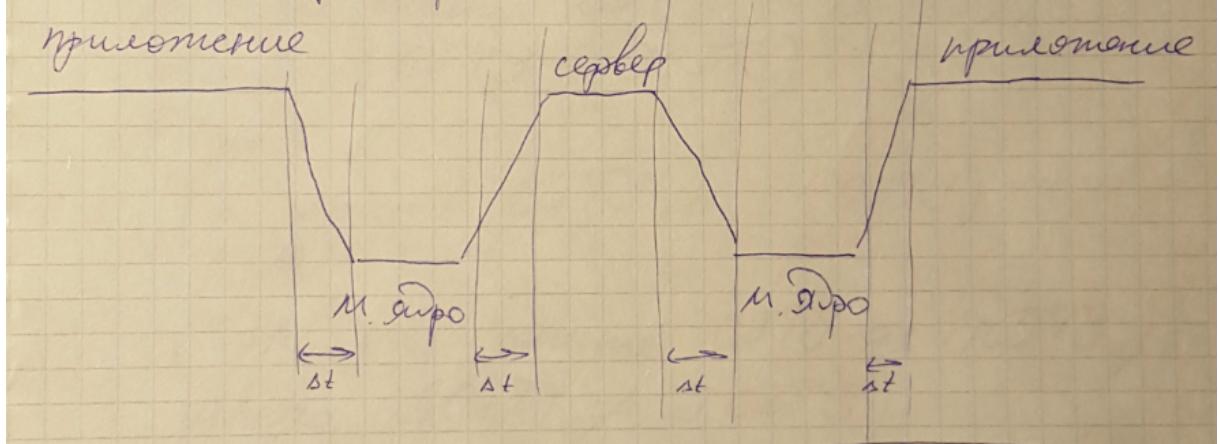


Рисунок 10.4 — pic

В результате микроядерная архитектура является не эффективной.

Основа микроядерной архитектуры: сервисы операционной системы реализованы в собственных адресных пространствах. В микроядре остается функция непосредственного управления процессором. 2 функция микроядра связана с управлением памятью. Управление памятью в большинстве современных систем выполняется страницами по запросам. Самый нижний уровень управления памятью – выделение физической страницы.

Процессор и ОЗУ – суть компьютера. Остальное – внешнее устройство. В микроядре остаются низкоуровневые функции (обработка прерываний).

Планирование процессов может выполняться в режиме пользователя. Это пересчет приоритетов процессов в зависимости от различных ситуаций в системе. Система с вытеснением: если пришел более приоритетный процесс, то он вытеснит выполняющийся процесс.

Менеджер памяти. Какие страницы можно вытеснить? А ядру дается команда на пейджинг.

Все ОС выполняют одни и те же функции из-за одинаковой архитектуры компонентов. Микроядерная – крайне неэффективная архитектура, так как у нас есть предсказуемые задержки, как минимум в 2 раза из-за переключения полного контекста. Почему интерес к микроядерным архитектурам не угасает? Поддается модификации без изменения кода ядра. Чтобы добавить новую функциональность – достаточно написать свой сервис. Благодаря возможности модификации без изменения всей ОС. Реализовано в ОС Mach.

### 10.2.1 ОС Mach

Оперирует набором абстракций. Базовые: процесс, поток. В Mach определены такие абстракции:

а) процессы (имеет виртуальное адресное пространство; в этом ад.пр. находится код, данные и несколько стеков; процесс – единица декомпозиции системы, так как именно процесс является владельцем ресурсов);

б) потоки (поток – непрерывная часть кода программы, которая может выполняться параллельно с другими частями кода; поток – независимо планируемый контекст выполнения, разделяющий единое адресное пространство процесса с другими потоками; единица планирования – поток; потоку принадлежит аппаратный контекст и счетчик команд; в процессе может быть один поток)

в) объекты памяти (может состоять из 1 или несколько страниц; в основе управления памяти – кластеризация (заранее загрузим следующие страницы); составляет основу управления виртуальной памятью; выполняется страницами по запросам; для загрузки отсутствующей страницы посыпается сообщение серверу памяти, который

выполняется в режиме задачи, получив ответ от сервера, сможет выполнить загрузку нужных страниц в оперативную память)

г) порты – защищенный почтовый ящик, который способен поддерживать очередь сообщений. (очередь сообщений – упорядоченный список сообщений)

д) сообщения (межпроцессное взаимодействие осуществляется с помощью сообщений; для передачи сообщений определена абстракция порт)

Система поддерживает несколько типов портов:

а) порт процесса (используется для взаимодействия с ядром; многие функции ядра процесс вызывает путем отправки сообщений в порт процесса)

б) порт загрузки (используется при старте системы; процесс init читает из порта загрузки инфо о именах .. обеспечивающие более важные сервисы)

в) порт особых ситуаций (используется при передаче сообщений об ошибках процессу)

г) зарегистрированные порты (для взаимодействия процессов со стандартными в системе серверами; процесс может предоставить другому процессу посыпать/получать сообщения через один из принадлежащих ему портов; такая возможность реализуется с помощью мандата, который включает указатель на порт и список прав которыми обладает другой процесс по отношению к данному порту; может выполнить команду send или только команду resive);

В Mach процесс может называться task.

## 11 Операционные системы реального времени

Операционные системы реального времени (ОСРВ) QNX. Применять понятие реального времени к физическим устройствам бессмысленно. Мы и они живут в реальном времени. Это понятие возникло для операционных систем (может быть и ПО спец назначения).

Понятие от Posix 1003.1: реальное время в операционных системах – способность операционной системы обеспечить требуемый уровень сервиса в определенный промежуток времени.

Ключевым отличием ядра ОСРВ является детерминированность (основанный на строгом контроле времени). Детерминированность определяется тем, что они управляются внешними системами/устройствами. Они завязаны на характеристиках внешних устройств. Это и есть детерминированность. Её следствием является требование обеспечения операционной системы соответствующих сервисов за определенные промежутки времени. Т.е. в ОСРВ главным критерием эффективность является обеспечение временных характеристик вычислительного процесса. Любая система реального времени должна реагировать на сигналы управляемого объекта в течении заданных временных ограничений. Очевидно – в ОСРВ планирование имеет особое

значение. Однако следует учитывать, что в ОСРВ как правило, набор выполняемых задач известен заранее. В ОСРВ часто имеется информация о временах выполнения задач, моментах активизации и предельных допустимых интервалах ожидания ответов. Такая информация может существовать в хорошо изученных/детерминированных системах. Достигается это путем выполнения большого кол-ва экспериментов с оценкой всех параметров системы. Также часто это достигается путем моделирования системы или эмуляции, когда одна операционная система работает как другая ОС. В результате снимаются все параметры тщательным образом. О системе реального времени известно все. В силу этого выбираются алгоритмы планирования, пересчет приоритетов.

Различаются системы реального времени:

- а) гибкие (система резервирования билетов; если из-за временных задержек оператору не удается зарезервировать билет, это не страшно так как запрос может быть повторен);
- б) жесткие (система управления атомным реактором; временные задержки/отказ системы недопустимы)

В системах реального времени использовались статические приоритеты. В настоящее время в ОСРВ используется подходы:

- а) статические приоритеты в зависимости от значимости процессов;
- б) динамические приоритеты (пересчитываются).

Классический алгоритм планирования независимых задач для жестких систем реального времени с одним процессором является

**алгоритм 1973 года Лью и Лейланд .** Основан на следующих предположениях:

- а) запросы на выполнение всех задач набора имеют жесткие временные ограничения на ответ системы и являются периодическими;
- б) все задачи – независимы. т.е. между любой парой задач не существует ограничений на порядок выполнения или взаимного исключения;
- в) срок выполнения каждой задачи равен её периоду  $P_i$ ;
- г) максимальное время выполнения каждой задачи известно и постоянно  $C_i$ ;
- д) время переключения контекста можно игнорировать;
- е) максимальный суммарный коэффициент загрузки процессора X  $C_i/P_i$  при существовании  $N$  задач не превосходит  $N*(2^{1/n} - 1)$  при стремление N к бесконечности и приблизительно равно  $\ln(2)$ , т.е. 0.7

Суть алгоритма: задачам назначаются статические приоритеты в зависимости от времени выполнения. Задачи с наименьшим временем получают наивысший

приоритет. При соблюдении всех ограничений алгоритм гарантирует выполнение временных ограничений для всех задач для всех ситуаций. Если периоды повторения задач кратны периоду самой короткой задачи то требование к максимальному коэффициенту нагрузки процессора – смягчается, и он может доходить до единицы.

У Таненбаума это отображается следующей формулой:

$N$  процессов,  $M$  периодических событий. Событие с номером  $i$  поступает с периодом  $p_i$  и на его обработку уходит время  $c_i$ . Тогда своевременную обработку всех потоков обеспечит выполнение след условия:

$$\sum_{i=1}^m ci/pi \leq 1$$

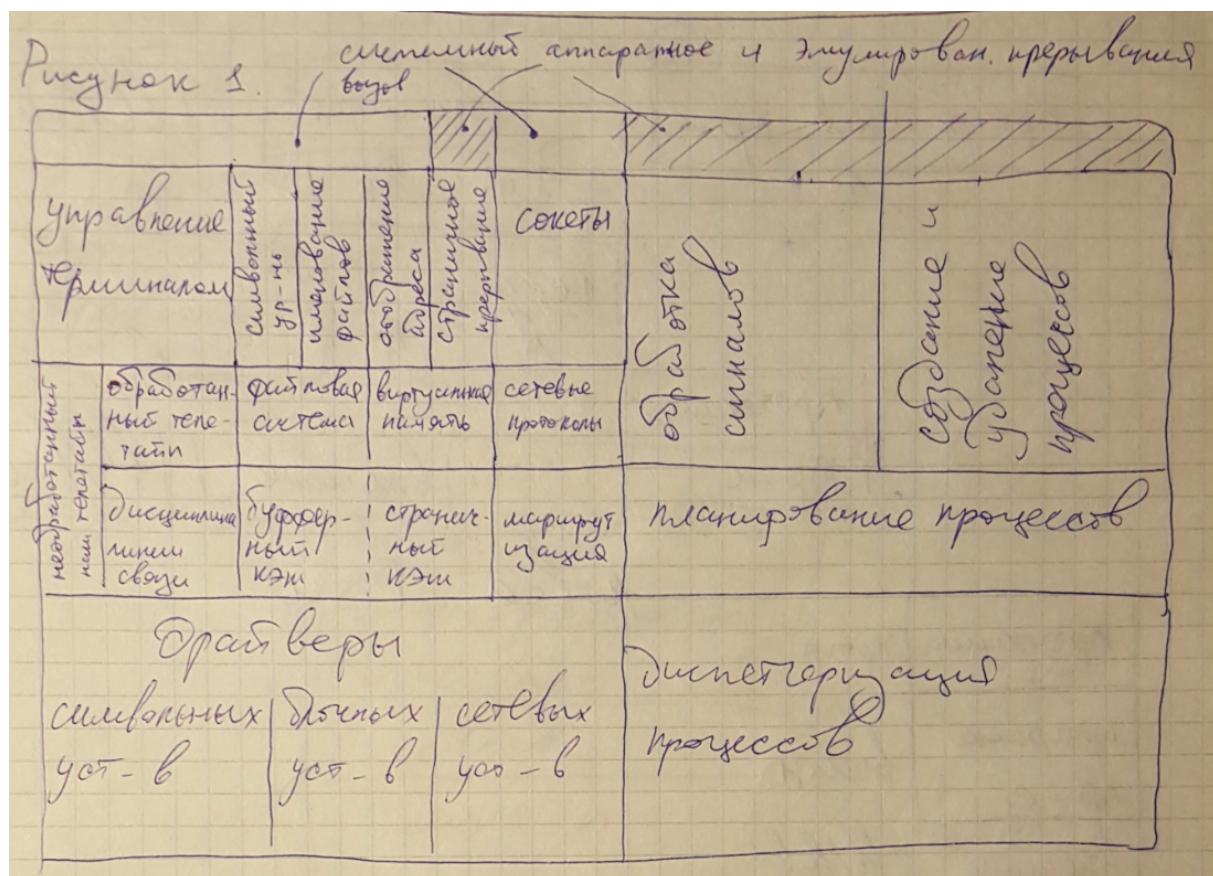


Рисунок 11.1 – Структура ядра QNX 4.4 BCD

Иерархическая организация с монолитным ядром. Все уровни показывают, что остается в микроядре, и что остается режиме пользователя. Она базируется на Unix. Планирование – постановка процессов в очередь в соответствии с их приоритетами. Диспетчеризация – непосредственное выделение кванта процессорного времени. Драйверы – для управления внешними устройствами. Сокеты – средство взаимодействия процессов в Linux BCD.

## 12 Прерывания

Существуют 2 подхода к реализации обработки прерываний в системе:

- запрет прерываний на время выполнения обработчика прерывания. Запрет прерываний на длительное время в системе – невозможен. Поэтому обработчики прерываний делятся на быстрые и медленные, а медленные делятся на две части: верхнюю и нижнюю половину. Обработчик таймера может инициализировать последующее выполнение планировщика выполнения. Верхняя половина считывает в буфер информацию из контроллера и заканчивается инициализация последующего выполнения нижней половины обработчика. В Юникс поддерживается несколько типов вторых половин. В windows тоже самое, но через DPC.

6) вложенные прерывания.

### Вложенные прерывания

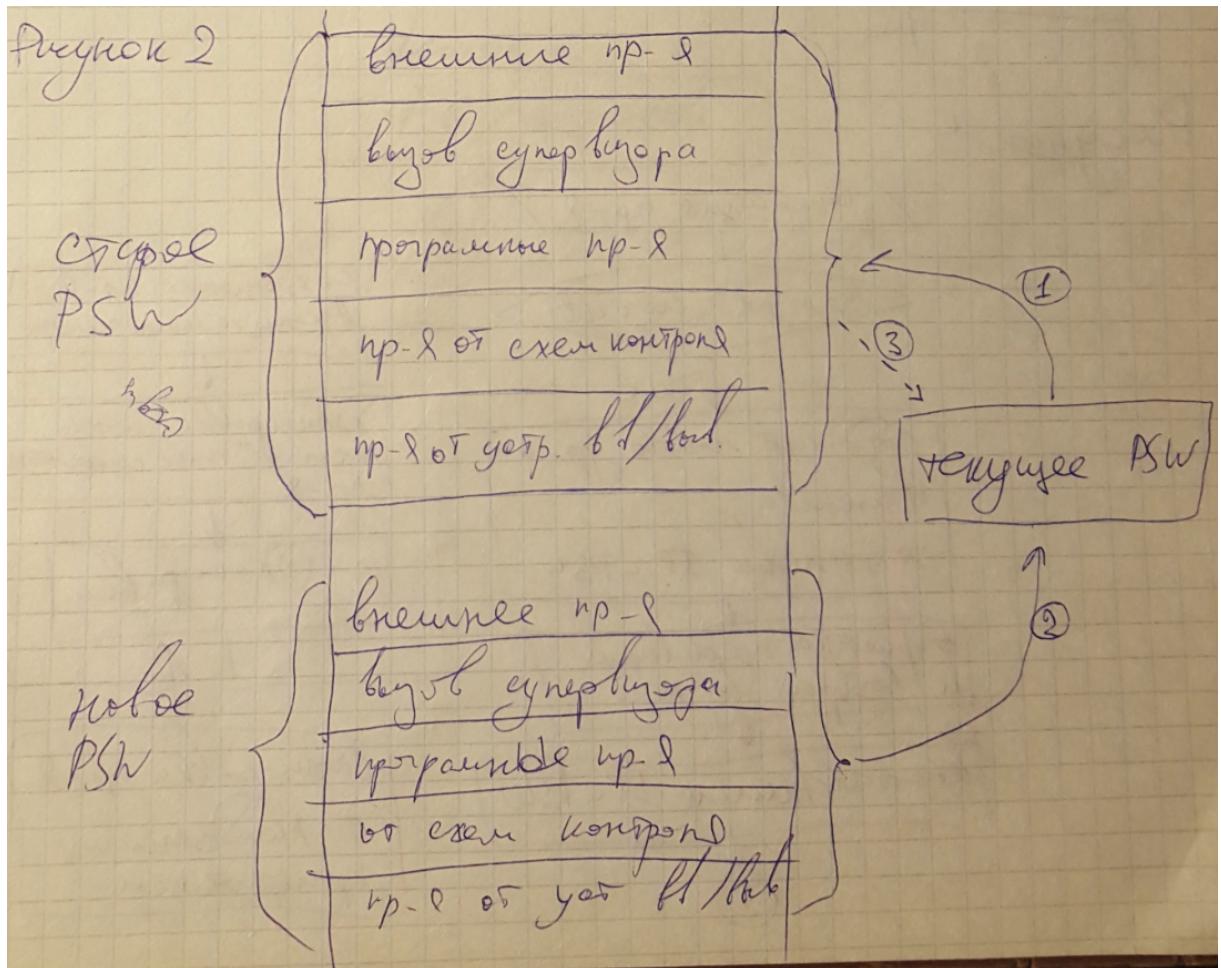


Рисунок 12.1 — pic

PSW - аппаратный контекст.

Механизм прерывания реализуется аппаратно (шаги):

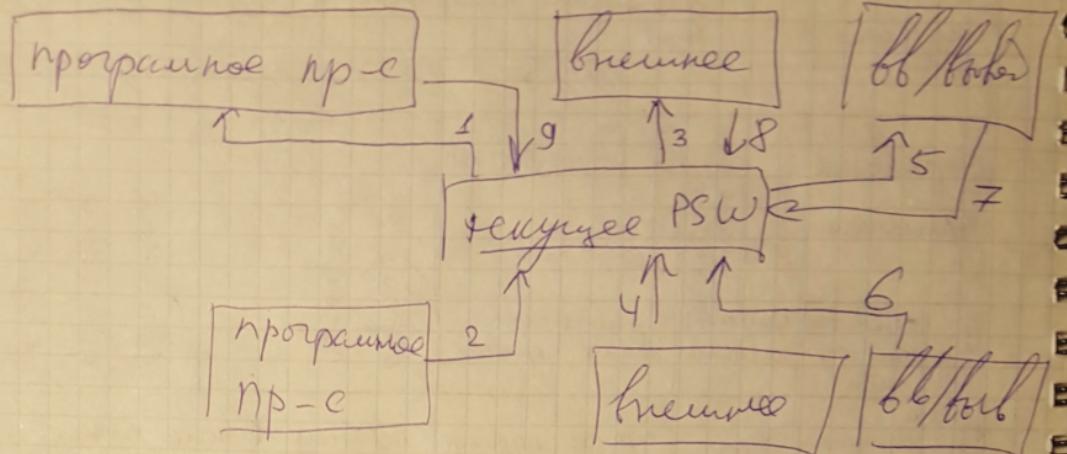
- текущее PSW записывается в место старого PSW
- новое PSW загружается в качестве текущего
- по завершению обработки прерывания старое PSW загружается в место текущего, что дает возможность продолжить выполнение прерванной программы.

Пример: пусть одновременно возникло три сигнала прерывания:

- программное прерывание (5 приоритет)
- внешнее (3)
- устройство ввода/вывода (2)

Рисунок 3

старое PSW



текущее PSW



Рисунок 12.2 — pic

Процесс обработки прерывания. Аппаратные прерывания. Основная функция аппаратных прерываний – информировать процессор о завершении операций ввода/вывода, что позволяет процессору отключиться на какое то время от управления устройством (распараллеливание функций). Это асинхронные события. Система не знает, в каком месте выполнения потока инструкций произойдет прерывание. В результате прерывания в системах x86 через таблицу дескрипторов прерываний осуществляется вызов соответствующего обработчика прерываний, называемый ISR (interrupt service routine). Выполняется в режиме ядра в системном контексте, так как прерванный процесс не имеет отношения к возникшему прерыванию, обработчик этого прерывания не должен обращаться к контексту процесса. По этой причине он не обладает правом блокировки. Прерывание все равно оказывает некоторое влияние на выполнение текущего процесса, а именно, время потраченное на обработку прерывания является частью выделенного процессу кванта. Обработчик прерывания системного таймера использует тики текущего процесса и поэтому нуждается в доступе к его структуре proc. Контекст процесса не полностью защищен от доступа к адресному пространству процесса.

Прерывания могут возникнуть в результате возникновения множества независимых событий в системе. Поэтому реальна ситуация, когда во время одного прерывания, возникает другое. При этом прерывание аппаратного таймера должно обслуживаться сразу. Вводится поддержка различных уровней приоритета Interrupt priority level. Уровни IPL сравниваются и устанавливаются на аппаратном уровне. Ядро может повысить приоритет некоторых критических инструкций. На некоторых аппаратных платформах поддерживается глобальный стек прерываний, используемый всеми обработчиками. На платформах, не имеющих такого стека, действуют стеки ядра текущего процесса. Должен быть обеспечен механизм изоляции стека ядра от обработчика. Ядро помещает в стек уровень контекста перед вызовом обработчика.

### Алгоритм обработки прерывания

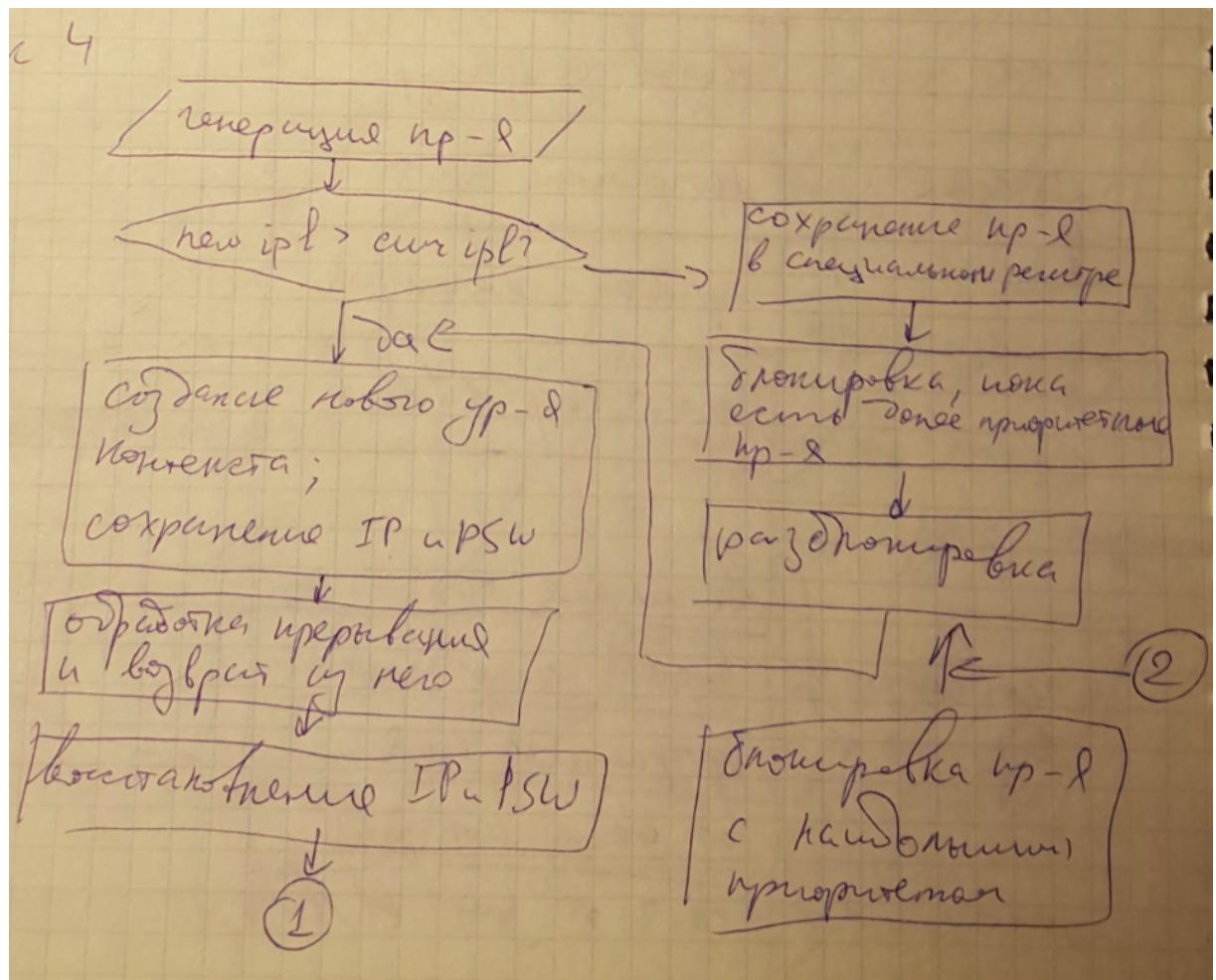


Рисунок 12.3 — Алгоритм обработки прерывания

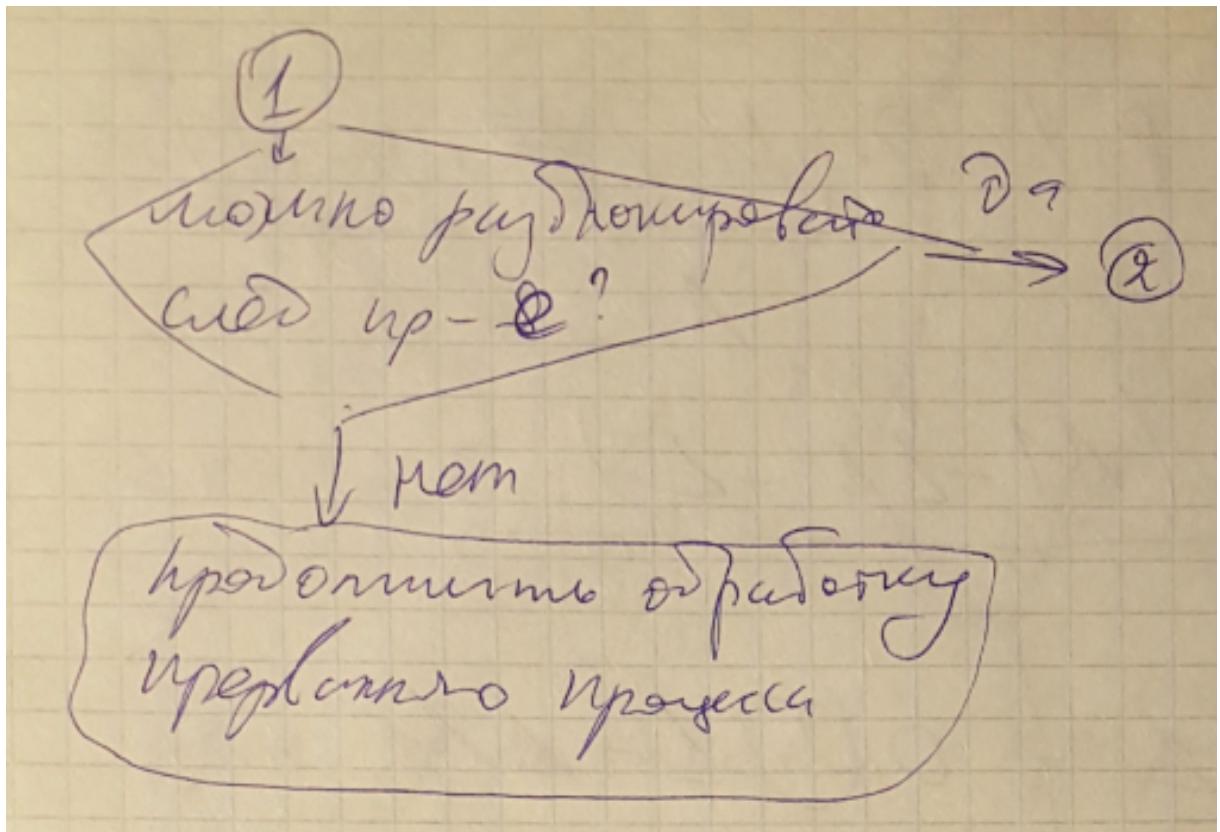


Рисунок 12.4 – Алгоритм обработки прерывания

Обработчики прерываний входят в состав драйверов соответствующих устройств. Они регистрируются в этих драйверах.

#### Реализация обработки прерываний в Windows.

Завершая передачу данных, контроллер устройства генерирует прерывание. Это прерывание в x86 приходит на выделенную ножку контроллера прерываний. После того, как прерывание получено, начинает действовать ядро, а именно диспетчер ввода/вывода и драйвер устройств. В результате процессор передает управление (в windows обработчик ловушки ядра или ISR) в соответствии с таблицей диспетчирования прерываний.

ISR в windows выполняется в два этапа:

1 этап. На уровне device irql (на этом этапе сохраняется название устройства и запрещаются дальнейшие прерывания от него)  
 DPC – deferred procedure call (отложенный вызов процедуры).

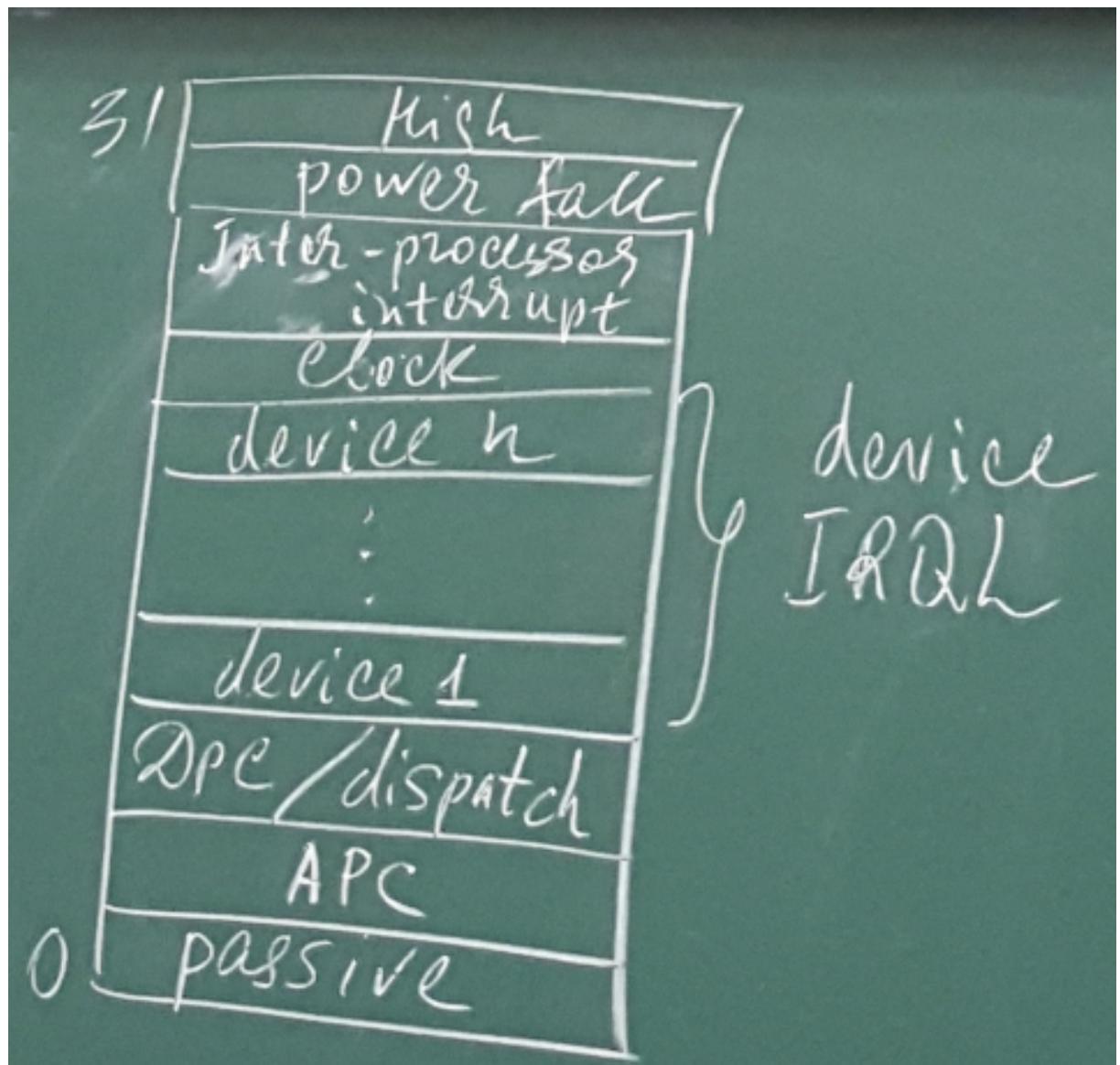


Рисунок 12.5 — В Windows NT применяется 32 уровня IRQL

2 этап. ISR помещает DPC в очередь на выполнение, после чего завершает прерывание и завершается.

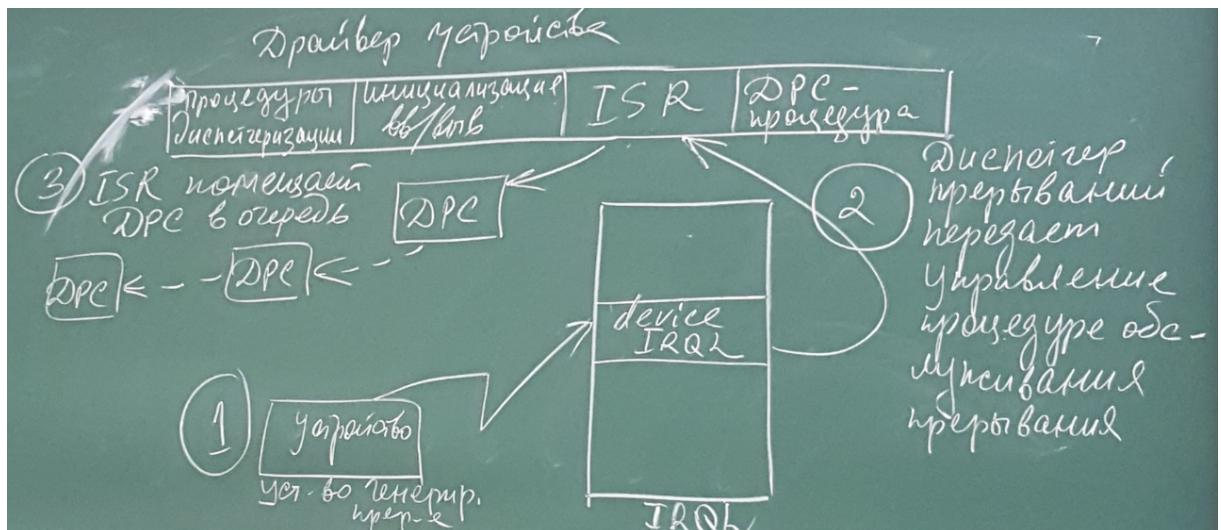


Рисунок 12.6 — Первая часть обработки

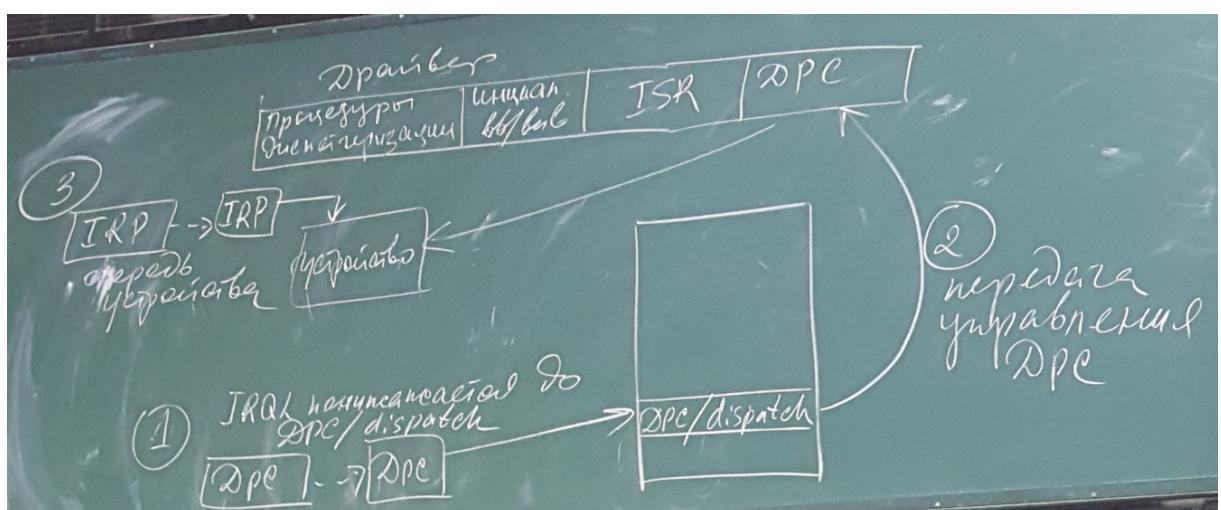


Рисунок 12.7 — Вторая часть обработки

На первой фазе устройство генерирует прерывание, определяется уровень данного прерывания irql, вызывается обработчик соответствующего прерывания ISR, который входит в состав драйвера. ISR должна выполнять минимум действий, связано это с тем, что часть действий – критические, и должны выполняться, как неделимые. Основное действие – сохранение информации о состоянии прерывания. Второе действие – инициализация отложенного действия. ДПС процедура начинает обработки следующего запроса из очереди устройства IRP (пакет), после чего заканчивает обработку прерывания. В системе столько очередей DPC, сколько процессоров. DPC – объект. По умолчанию ядро помечает DPC объекты в конец DPC очереди того процессора, на котором выполнялось ISR. Однако драйвер устройства может указать приоритет DPC низкий, средний, высокий, а также может направить DPC в очередь конкретного процессора. Низкий – когда очередь DPC превышает пороговое значение. DPC применяется, когда истекает квант. При каждом тике системного

таймера генерируется прерывание IRQL clock (приоритет его выше всех девайсов, это видно на 12.5. Обработчик прерывания таймера обновляет системное время и уменьшает квант. Когда счетчик кванта обнуляется, ядру может понадобиться перераспределить процессорное время. По 12.5 задача перераспределения имеет ???.

Обработчик прерывания таймера ставит DPC в очередь для того, чтобы инициализировать диспетчеризацию потоков, после чего завершает свою работу и IRQL процессора понижается. По истечению кванта пересчитывать приоритеты – поздно. Оно осуществляется по главному тику. Таймер инициализирует постановкой соответствующего объекта DPC в очередь процессору. И когда обработчик таймера завершается, когда код обработчика прерывания завершился, IRQL процессора понижается. Сначала находилось на уровне clock. Power выше, чтобы система сохранила работоспособность, всё завершается, выполняет сохранение состояний, чтобы потом могли сохранить работу. Если DPC имеет приоритет ниже приоритета всех аппаратных прерываний, то сначала обрабатываются все возникшие аппаратные прерывания, а затем генерируется прерывание DPC. На уровне passive могут выполняться некоторые потоки ядра и процессы пользователя.

Реализация обработки прерываний в системе:

- a) запрещение прерывание (для многопроцессорных систем запрещение прерываний выполняется на локальном процессоре; в системе столько таблиц дескрипторов прерываний (таблиц диспетчеризации прерываний) сколько процессоров, так как каждый процессор должен адресовать возникающее прерывание; только прерывание от системного таймера выполняется одним процессором и такой процессор принято называть главным, несмотря на симметричность системы).
- б) вложенные прерывания (предполагают наличие приоритетов и в соответствии с этими приоритетами более высокоприоритетные прерывания вытесняют менее приоритетные; контекст сохраняется либо в стековой ??? либо в стеке).

Файловые системы. Современные системы отличаются только файловыми системами. Имеют уровни иерархии. Самый верхний – символный уровень именования файлов. Доступ к дискам осуществляется через подсистему ввода/вывода. ФС позволяет единообразно работать с устройствами.

## 13 ОС реального времени

Определение Posix: Реальное время в операционных системах – способность ОС обеспечить требуемый уровень сервиса в определенный промежуток времени. Перефразируем: ОС должна обеспечить возможность обслуживать внешние по отношению к системе системы за определенный промежуток времени. У нас есть некое устройство/система устройств и мы должны обеспечить управление ими. В системах общего назначения windows/Linux обеспечивают управление процессами реального времени, однако ключевым отличием сервисов ядра ОС реального времени является детерминированный характер их работы, основанный на строгом контроле времени выполнения определенных задач. Под детерминированностью следует понимать хорошо изученные системы, т.е. все системы реального времени являются системами специального назначения. Даже QNX, когда настроена на выполнение конкретным объектом (имеет все возможности) становится системой специального назначения. Про этот объект следует знать все о процессах и потоках.

Различаются два типа реального времени:

- a) жесткое (предполагает ответ системы в строго отведенный интервал времени);
- б) мягкое (возможны задержки).

Системы реального времени строятся по определенным принципам:

а) алгоритмы, позволяющие обеспечивать быстрое переключение потоков (переключение контекста должно выполняться быстро, должно поддерживаться аппаратно)

- б) минимальные временные задержки

Способы построения систем РВ:

а) на основе событийной синхронизации, которая предполагает переключение задач только тогда, когда возникает событие с более высоким приоритетом. Такой приоритет называется приоритетом планирования.

- б) квантование.

В системах реального времени главным критерием эффективности является обеспечение временных характеристик вычислительно процесса. В этих системах планирование имеет особое значение. Любая система РВ должна реагировать на сигналы управляемого объекта в течение заданных временных интервалов. Необходимость тщательного планирования работ облегчается тем, что весь набор выполняемых задач известен заранее. Чтобы система могла эффективно управлять набором задач в системе должна иметься информация не только о времени выполнения, но и о текущих затратах времени, о моментах активации задач и о предельно допустимых интервалах ожидания ответа. В системах реального времени применяются как статические приоритеты и алгоритмы планирования на статических приоритетах, так и динамические приоритеты и соответствующие алгоритмы. Появились недавно, связано это с ростом мощности компов.

Система резервирования билетов – мягкая система. Если из-за временных нарушений оператору не удается зарезервировать билет, то можно повторить заново.

QNX NEUTRINO v6.2 - ОС с микроядерной архитектурой построенная на модели клиент-сервер. Процесс в этой системе взаимодействуют с помощью сообщений. Поддерживает виртуальную память. Управление виртуальной памятью является затратным.

QNX обеспечивают все составляющие систем РВ:

- а) многозадачность;
- б) диспетчеризацию потоков на основе приоритетов;
- в) быстрое переключение контекста;
- г) вложенные прерывания

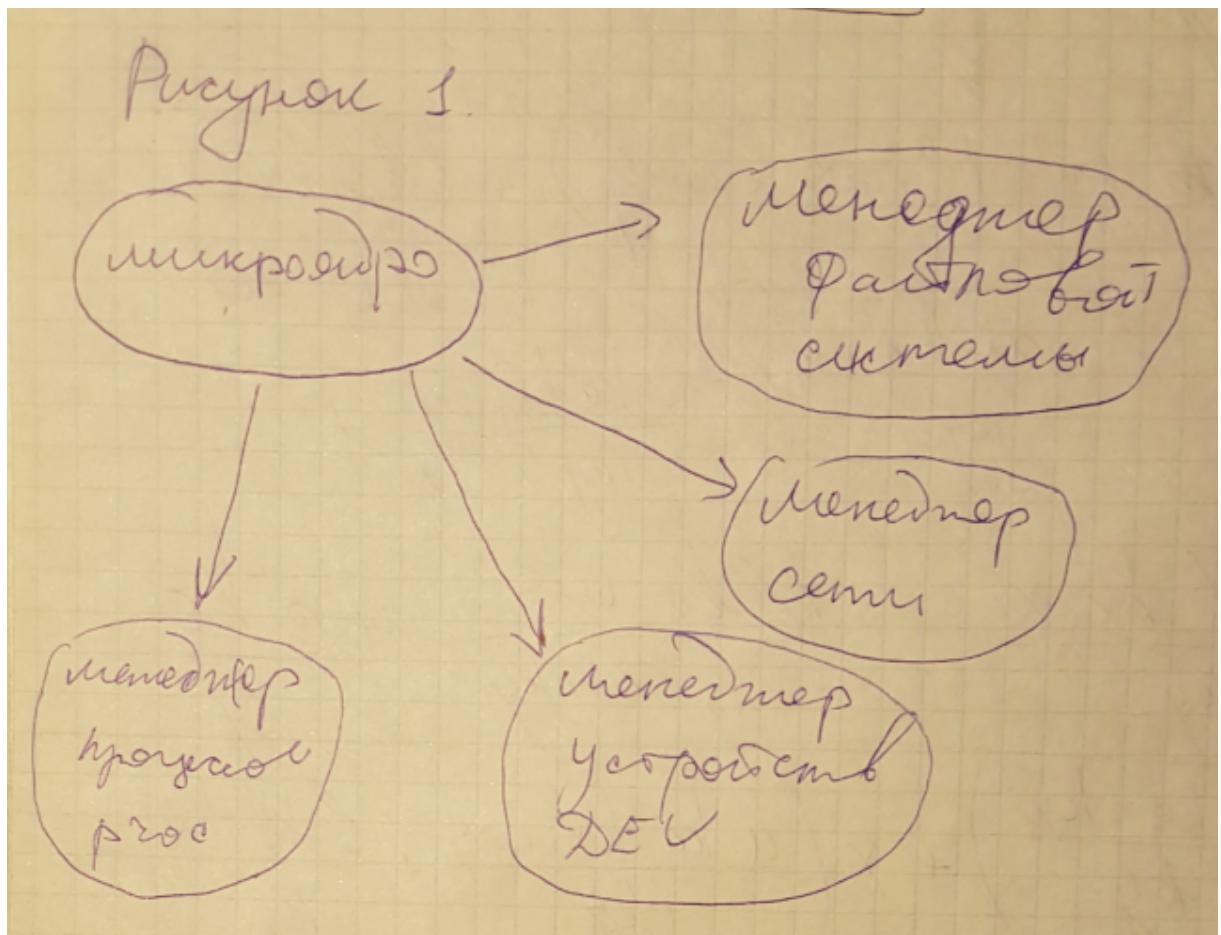


Рисунок 13.1 — Структура

Особенность микроядра - в отдельные модули вынесена поддержка управления дисками, управления сетевыми возможностями, при этом запуск и работа модулей аналогична пользовательским процессам.

Ядро выполняет функции:

- передачу сообщений (обеспечивает маршрутизацию сообщений);
- диспетчеризация потоков (простейший процесс имеет один поток).

Переключение потоков и процессов мало чем отличается. Процесс в QNX выполняется в собственном виртуальном адресном пространстве. Само ядро никогда не получает управление в результате диспетчеризации. Код ядра выполняется только как следствие системных вызовов, исключений или аппаратных прерываний. Единица диспетчеризации – поток.

Существует три события, в результате которых выполняется диспетчеризация потоков:

- вытеснение;
- блокирование (выполняемый поток может вызвать функцию, которая приведет к его блокировке);

в) уступка.

Поток в системе может иметь два состояния:

- а) готов;
- б) заблокирован.

Планирование потоков выполняется на основе системы приоритетов. Для супер-юзера выделен специальный диапазон приоритетов. Весь: 0-255. Пользовательский: 0 – 63.

Поддерживает дисциплины планирования:

- а) FIFO (поток выполняется пока не будет вытеснен более приоритетным/заблокированым/добровольно отдаст);
- б) RR (поток выполняется в течении определенного кванта; квант в 4 раза больше тактового интервала; тактовый интервал 1мс в процессорах с тактовой частотой 40МГц; в x86 квант будет 4мс);
- в) спорадическая (случайное) (вводится понятие приоритет переднего плана foreground и фоновый background)

Для управления спорадическим подходом используется:

- а) кол-во времени, за которое поток может выполниться;
- б) до какого уровня можно понизить приоритет
- в) период пополнения, когда поток может расходовать свой бюджет выполнения;
- г) максимально число текущих пополнений.

Рисунок 2

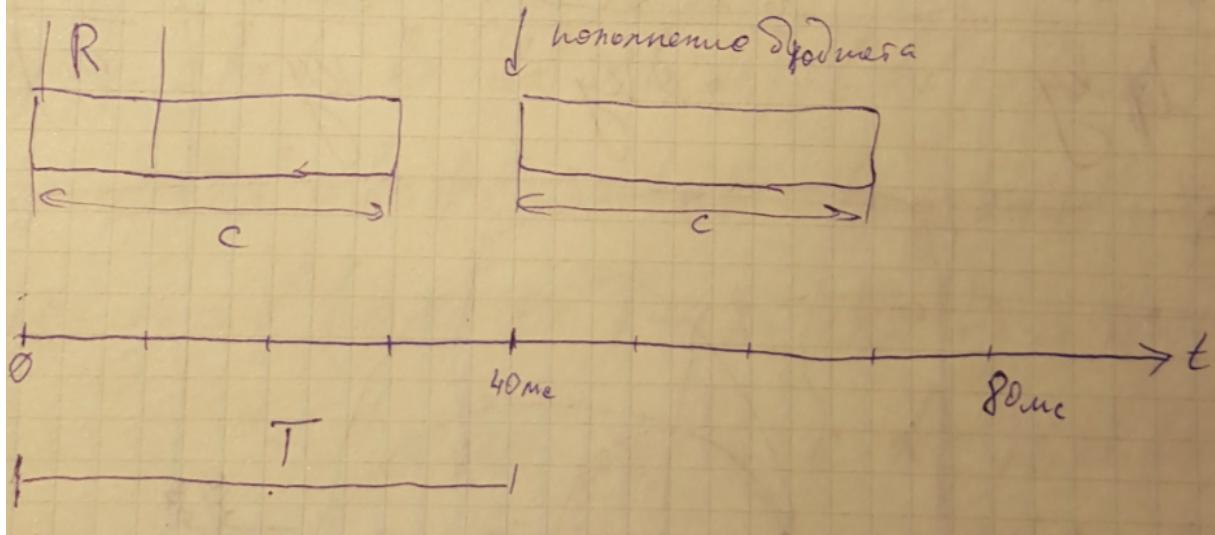


Рисунок 13.2 — pic

$C$  – начальный бюджет выполнения потока, который расходуется потоком в процессе выполнения. Бюджет пополняется с периодичностью . Когда поток блокируется, израсходованная часть потока  $R$  пополняется. Если в каком-то из драйверов возникла ошибка, то это не приводит к перезапуску системы, а приводит к перезапуску драйвера.

## 14 Точные и неточные прерывания

Связано с суперскалярными процессорами.

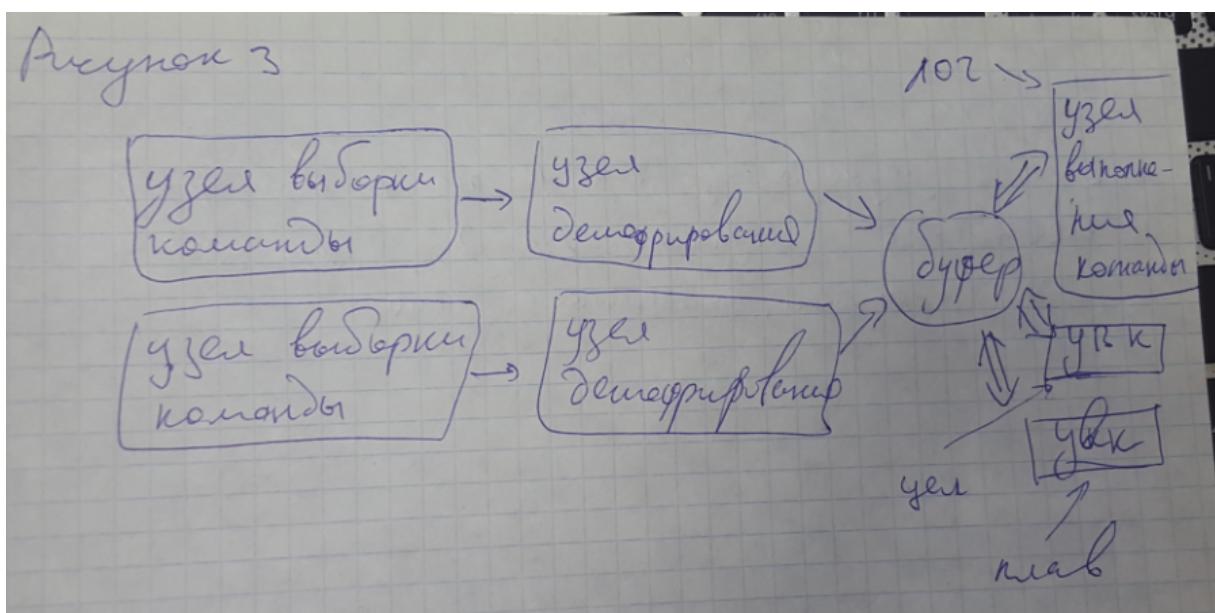


Рисунок 14.1 — Диаграмма выполнения команды

Команды могут выполняться не последовательно. В структуре с конвейером различные команды находятся на разных стадиях выполнения. В момент прерывания значение счетчика команд может не отражать истинной границы между выполненными и еще не выполненными командами. Скорее всего, он будет указывать на следующую команду, какую нужно считать из памяти. Соответственно при возврате и прерывания ОС не может просто начать заполнять конвейер с команды указанной в адресе счетчика команд. Она должна определить последнюю выполненную команду, что требует проведения серьезного анализа процессора. В суперскалярной еще хуже, поскольку команды могут выполняться не в порядке их расположения в памяти, нет четкой границы между невыполненными и выполненными. Может возникнуть такая ситуация: команды 1,2,3,5,8 – выполнены, 4,6,7,9,10 – не выполнены. Счетчик команд может указывать на 9, 10 или 11 команды. Вводится понятие **точного прерывания** – это прерывание, которое оставляет машину в строго определенном состоянии.

Свойства точного прерывания:

- a) счетчик команд указывает на команду, до которой все команды полностью выполнены;
- б) ни одна команда, после той, на которую указывает счетчик команд – не выполнены;
- в) состояние команды, на которую указывает счетчик команд – известно.

Ничего не говорится о том, что команды после той, на которую указывает счетчик команд, не могли начаться, утверждается только, что все изменения, связанные с выполнением этих команд, должны быть отменены до начала обработки прерывания.

## **Заключение**

Поправки, исправления, дополнения жду в виде pull-реквестов в  
<https://github.com/iproha94/BmstuOperatingSystems>

## **Список использованных источников**

1. *С. Медник, Дж. Донован. IBM 360 / Дж. Донован. С. Медник.*
2. *Кузнецов, Сергей. Блеск и нищета легковесных процессов. — <http://www.osr.ru/cw/1996/31/13520/>. — 1996.*
3. *Пол Дейтел, Харви Дейтел. Как программировать на С / Харви Дейтел Пол Дейтел. — 2014.*
4. *Рихтер, Джейффири. Windows для профессионалов. / Джейффири Рихтер. — 2001.*
5. *Солтис, Фрэнк. Основы AS/400 / Фрэнк Солтис.*
6. *Шоу, Алан С. Логическое проектирование операционных систем / Алан С. Шоу. — 1981.*