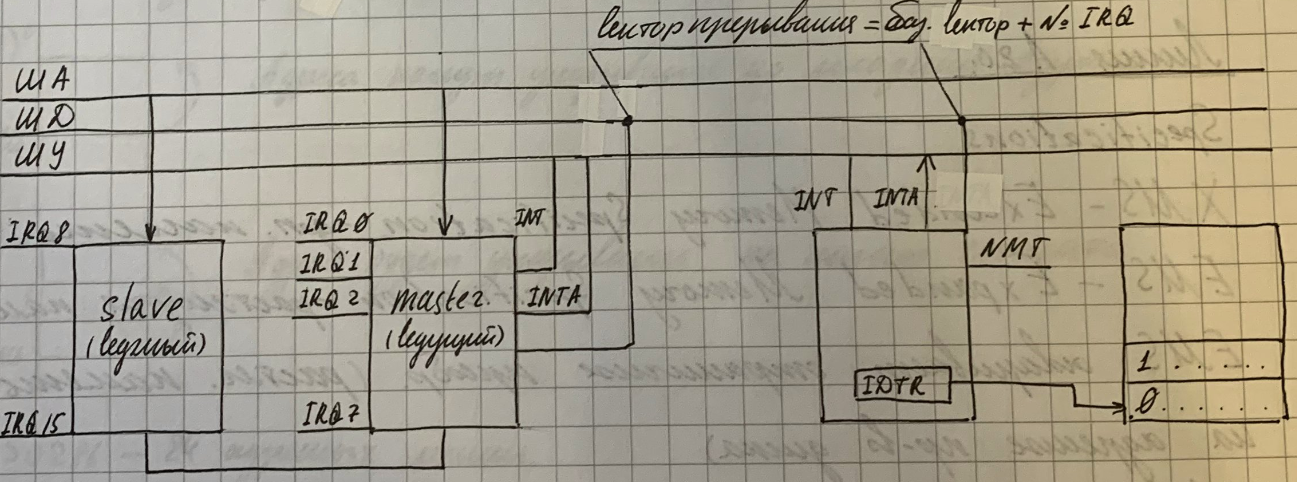
**OC – определение, место ОС в системе программного обеспечения ЭВМ. Ресурсы вычислительной системы.  Режимы  ядра  и  задачи:  переключение  в  режим  ядра  –  классификация  событий.  Процесс ,  как  единица  декомпозиции системы** ОС – комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами ,  которые используют эти ресурсы в вычислениях.   
Классификация ОС:  1.Однопрограммная пакетной обработки (в оперативной памяти м.б. только 1 прикладная программа);  2.Мультипрограммная пакетной обработки (в оперативной памяти одновременно много программ);  3.Мультипрограммная с разделением времени (в оперативной памяти одновременно большое число  программ, процессорное время – квантуется, чтобы  обеспечить фиксированное время ответа <= 3 с);  4. Реального времени (Главное – время отклика. Организует работу   
вычислительной системы в темпе, обеспечивающем  обслуживание  некоторого    
внешнего  процесса  не  зависимо  от  вычислительной  системы.  Время  отклика    
системы  <=  время  поступления  запроса  на  ответ.  В  системах  реального  времени используется система абсолютных приоритетов.  a. Жесткая система (строгое обеспечение времени ответа).  b. Гибкая система (цифровые аудио и мультимедийные средства) ); 5.Сетевые (серверные); 6. Многопроцессорные;   7. Встроенные (в ТВ, микроволновки);  8. ОС для смарт‐карт. | ОС ПК – многопроцессорная система с разделением времени. Ресурс – любой из компонентов вычислительной системы и предоставляемые им возможности.  (время CPU, объем ОЗУ, каналы I/O, timer, данные, ключи защиты, реентерабельные коды ОС (коды, не  модифицирующие сами себя (доступна повторная входимость).)Процесс – программа в стадии выполнения. Является единицей декомпозицией системы, именно ему выделяются ресурсы системы.  Пользовательский режим ‐ наименее привилегированный режим, поддерживаемый ОС; он не имеет прямого доступа к оборудованию и у него ограниченный доступ к памяти.  Режим ядра ‐ привилегированный режим. Т.е. части ОС, которые исполняются в режиме ядра, такие как драйверы    
устройств и подсистемы типа диспетчера виртуальной памяти, имеют прямой доступ ко всей аппаратуре и памяти. Различия в работе программ пользовательского режима и режима ядра поддерживаются аппаратными средствами компьютера (а именно ‐ процессором).  | Переключение процесса в режим ядра (прерывания):  
1. Системные вызовы (программные прерывания) - вызываются искусственно с помощью соответствующей команды из программы (int), предназначены для выполнения некоторых действий ОС (фактически запрос на услуги ОС) (обращение к ЗУ);   
 2. Исключения - реакция микропроцессора на нестандартную ситуацию, возникшую внутри микропроцессора во время выполнения некоторой команды (Исправимые - приводят к вызову определенного менеджера системы, в  результате работы которого может быть продолжена работа процесса, например отсутствие нужной информации в оперативной памяти; Неисправимые - ошибки в коде. Например, деление на 0)  
3. Аппаратные прерывания (прерывания от системного таймера, прерывание от горячих клавиш, от устройств ввода/вывода)

**Три режима работы компьютера на базе процессоров Intel**   
1. **Реальный режим**  (или режим реальных адресов) ‐ это название было дано прежнему способу адресации  памяти после появления 286‐го процессора, поддерживающего защищённый режим.  Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов, 20‐разрядный адрес –  сегмент/смещение). Минимальная адресная единица памяти – байт.   FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).  Компьютер начинает работать в реальном режиме. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров.  1‐проц.  режим под управлением MS‐DOS (главное  –  минимизация  памяти,  занимаемой  ОС  =>  нет  многозадачности).  *Таблица векторов прерываний: 0‐256 б.:*  резидентная часть DOS; место резидентных программ; сегменты программы; pool или heap (куча); транзитая часть DOS;  Резидентная часть DOS   *1 Мб*   
2. **Защищенный  режим** –  многопроцессный  режим.  В памяти компьютера одновременно находится большое число программ  с  квантованием  процессорного  времени  с  виртуальной  памятью.  Управляет защищенным режимом ОС с разделением времени (Windows,  Linux).  В защищенном режиме 4 уровня привилегий, ядро ОС – на 0‐м. Создан для работы нескольких независ. программ. Для обеспечения совместной работы нескольких задач необходимо защитить их от взаимного влияния, взаимод. задач д. регулироваться. Разработан фирмой Digital Equipments для 32‐разрядных компьютеров VAX‐11

3. **Специальный режим защищенного режима (V86)** - как задачи выполняются ОС реального режима, в  кажд. из кот. выполн. по 1 прогр. реального режима. Многозадачный режим с поддержкой виртуальной памяти.  процессор  фактически  продолжает  использовать  схему  преобразования  адресов  памяти  и  средства  мультизадачности  защищённого  режима.  В  виртуальном  режиме  используется  трансляция  страниц  памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в  виртуальном режиме. Каждая из этих задач может иметь собственное адресное пространство, каждое  размером  в  1  мегабайт .  Все  задачи  виртуального  режима  обычно  выполняются  в  третьем,  наименее  привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор автоматически  переключается  из  виртуального  режима  в  защищённый.  Поэтому  все  прерывания  отображаются  в  операционную систему, работающую в защищённом режиме.

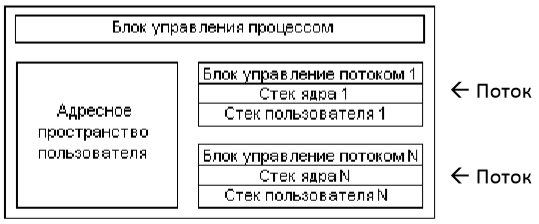
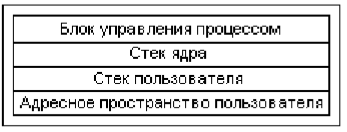
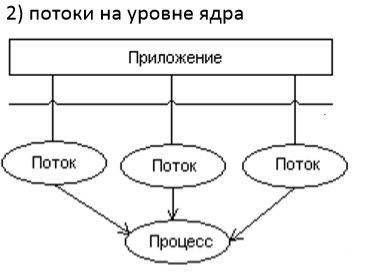
**Классификация операционных систем и их особенности. Иерархическая машина. Виртуальная машина.** ОС – комплект программ, которые совместно управляют ресурсами вычислительной системы и процессами ,  которые используют эти ресурсы в вычислениях.   
Классификация ОС:  1.Однопрограммная пакетной обработки (в оперативной памяти м.б. только 1 прикладная программа);  2.Мультипрограммная пакетной обработки (в оперативной памяти одновременно много программ);  3.Мультипрограммная с разделением времени (в оперативной памяти одновременно большое число  программ, процессорное время – квантуется, чтобы  обеспечить фиксированное время ответа <= 3 с);  4. Реального времени (Главное – время отклика. Организует работу   
вычислительной системы в темпе, обеспечивающем  обслуживание  некоторого    
внешнего  процесса  не  зависимо  от  вычислительной  системы.  Время  отклика    
системы  <=  время  поступления  запроса  на  ответ.  В  системах  реального  времени используется система абсолютных приоритетов.  a. Жесткая система (строгое обеспечение времени ответа).  b. Гибкая система (цифровые аудио и мультимедийные средства) ); 5.Сетевые (серверные); 6. Многопроцессорные;   7. Встроенные (в ТВ, микроволновки);  8. ОС для смарт‐карт. | ОС ПК – многопроцессорная система с разделением времени. | Иерархическая машина – ОС разбивается на функции и определяется место этих функций по удаленности от  аппаратной части  
  
Виртуальная машина – совокупность команд машины и команд ОС, кот . м . использовать программы для  получения сервиса ОС

**XMS, линия A20  – адресное заворачивание.** XMS  (eXtended  Memory  Specification,  спецификация  расширенной  памяти) –  спецификация  Mirosoft  на  расширенную память (XMS 2.0), позволяющая DOS‐программам с помощью диспетчера расширенной памяти  (XMM) использовать расширенную память ПК на процессорах 80286 и более новых. (добавленная, продленная) XMSоговаривает все  вопросы, связанные с дополнительной памятью (сверх 1 Мб).  EMS (Expanded Memory Specification, спецификация отображаемой памяти) –стандарт, разработанный в 1985 г.  фирмами Lotus, Intel и Microsoft для доступа из DOS к областям памяти выше  
1 Мбайт в системах на базе  процессоров 80386 и более поздних. (расширенная, растягиваемая)  | 0-640 Кб:  основная память (conventional) – память, доступная  
DOS и программам реального режима.  Стандартная память является самой дефицитной в PC, когда речь идет о работе в среде ОС типа MS‐DOS. На ее  небольшой  
объем (типовое значение 640 Кбайт) претендуют и BIOS, и ОС реального  
режима, а остатки отдаются  прикладному  ПО.  Стандартная  память  используется  для  векторов  прерываний,  области  переменных  BIOS;  области DOS; предоставляется пользователю (до 638 Кбайт).  • 640‐1024 Кб:  UMA (upper memory area)  
 – обл. верхней памяти – зарезервирована для системных нужд  Размещаются обл. буферной памяти адаптеров (пр. – видеопамять) и постоянная память (BIOS с  
расширениями).  • C 1024 Кб –…(защ.)  XMA (extended memory area)  
 – непосредственно доступна только в защищенном  режиме для компьютеров с процессорами 286 и выше.  • 1024‐1088 Кб (Реал): HMA (high memory area)   
– верхняя область памяти (1‐й сегмент размером 64 Кбайт,  расположенный выше мегабайтной отметки памяти PC с операционной системой MS‐DOS) Единственная    
область расширенной памяти, доступная 286+ в реальном режиме при открытом вентиле Gate A20.  | XMS – программная спецификация использования дополнительной   
памяти DOS‐программами для компьютеров  на процессорах  286 и выше . Позволяет  
программе получить в распоряжение одну или несколько областей  дополнительной  
памяти,  а  также  использовать  область HMA.  Распределением  областей  ведает  диспетчер  расширенной памяти ‐ драйвер HIMEM.SYS. Диспетчер позволяет захватить или освободить область HMA (65 520  байт, начиная с 100000h), а также управлять вентилем линии адреса A20. Функции XMS позволяют программе: определить размер максимального доступного блока памяти; захватить или освободить блок памяти; копировать данные из одного блока в другой, причем участники копирования могут быть блоками как  стандартной, так и дополнительной памяти в любых сочетаниях; запереть блок памяти (запретить копирование) и отпереть его; изменить размер выделенного блока.  Спецификации EMS и XMS отличаются по принципу действия: в EMS для доступа к дополнительной памяти  выполняется отображение (страничная переадресация) памяти, а в XMS ‐ копирование блоков данных.

Адресное заворачивание: Процессор  в  реальном  режиме  поддерживает  адресное    
пространство  до  1Мбайт.  Адресное  пространство  разбито на сегменты по 64Кбайт. 20‐битный базовый адрес сегмента вычисляется сдвигом значения селектора на  4 бита влево. Данные внутри сегмента адресуются 16‐битным смещением. | В реальном режиме существует 2 вида адресного заворачивания.  В 8086 – 20 линий адреса, заворачивание на начало. 220 = 1 Мб = FFFFF    
Если  в  реальном  режиме  открыть  линию  А20  (21‐ю  линию),  то  в  реальном  режиме  станет  доступно  дополнительно 64 Кб памяти. Линия А20 – для совсместимости (для заворачивания).  По умолчанию компьютер начинает работу в реальном режиме. Для обеспечения адресного заворачивания А20  сброшена – заземлена. Для перехода в защищенный режим надо открыть А20. | В реальном режиме формирования   
линейного адреса есть возможность адресовать пространство между 1Мб и    
1Мб+64Кб (например, указав в качестве селектора 0FFFFh, а в качестве смещения 0FFFFh, мы получим линейный  адрес 10FFEFh).  
  
  
**Прерывания: классификация; аппаратные прерывания ‐ последовательность операций при выполнении  аппаратного прерывания. Прерывания точные и неточные** Классификация прерываний (в зависимости от источника) :  
1.программные (системные вызовы) – вызываются искусственно с помощью соответствующей команды из программы  (int),  предназначены  для  выполнения  некоторых  действий  ОС  (фактически  запрос  на  услуги  ОС),  является  синхронным событием; 2. аппаратные – возникают как реакция микропроцессора на физический сигнал от некоторого устройства (клавиатура,  системные  часы,  мышь ,  жесткий  
диск  и  т .д.),  по  времени  возникновения  эти  прерывания  асинхронны, т .е.  происходят в случайные моменты времени (от таймеров, горячие клавиши, от устройств вв/вв); 3. исключения  –  являются  реакцией  микропроцессора  на  нестандартную   
 ситуацию,  возникшую  внутри  микропроцессора во время выполнения некоторой команды программы (деление на ноль, прерывание по флагу TF  (трассировка)), являются синхронным событием.) Исправимые – приводят к вызову определенного менеджера системы, в результате работы которого может  быть продолжена работа процесса (пр.: страничная неудача с менеджером памяти). Неисправимые – в случае сбоя или в случае ошибки программы (пр.: ошибка адресации). В этом случае  процесс завершается | Механизм реализации аппаратных прерываний: Когда  устройство  заканчивает  свою  работу,  оно  инициирует  прерывание  (если  они  разрешены  ОС).  Для  этого  устройство  посылает  сигнал  на  выделенную  этому  устройству  специальную  линию  шины.  Этот  сигнал  распознается  контроллером  прерываний.  При  отсутствии  других  необработанных  запросов  прерывания контроллер обрабатывает его сразу.  Если при обработке прерывания поступает запрос  от  устройства  с  более  низким  приоритетом,  то  новый  запрос  игнорируется,  а  устройство  будет  удерживать сигнал прерывания на шине, пока он  не обработается.  Контроллер  прерываний  посылает  по  шине  вектор  прерывания,  который  формируется  как  сумма  базового  вектора  и  №  линии  IRQ  (в  реальном  режиме  базовый  вектор  =  8h,  в  защищенном – первые 32 строки IDT отведены под  исключения => базовый вектор = 20h). С помощью  вектора прерывания дает нам смещении в IDT, из  которой мы получаем точку входа в обработчик.  Вскоре  после  начала  своей  работы  процедура  обработки  прерываний  подтверждает  получение  прерывания, записывая определенное значение в  порт  контроллера  прерываний.  Это  подтвержд.  разреш. контроллеру издавать новые прерывания. Точные прерывания – прерывание, оставляющее машину в строго определенном состоянии. Обладает св‐ми:  1. Счетчик команд указывает на команду (текущая), до которой все команды выполнены. 2. Ни одна команда после текущей не выполнена.  Не говорится,  что  команды  после  текущей  не  могли  начать  выполнение,    
а  то ,  что  все  изменения ,  связанные  с  этими  командами, должны быть отменены.  3. Состояние текущей команды известно (аппаратн. – обычно не нач.   
вып., при искл. – привела к исключ.)   
  
  
  
  
**Защищенный режим: системные таблицы – GDT, IDT, теневые регистры** Защищенный режим – 32х‐разрядный, поддерж.  многопоточность  и  многопроцессность.    
В  отличие от реального режима здесь доступно 4  Гб памяти (в реальном диапазон адресов памяти  ограничен  1  мб).  В  защищ .  реж .  4  уровня  привилег.  Ядро ОС   
находится на 0‐м уровне.  GDT (global descriptor table) – таблица , которая  описывает  сегменты  основной  памяти  ОС.  В  системе только 1 GDT. На начальный адрес GDT  указывает GDT Register (32 разрядный).  IDT  (interrupt  descriptor  table)    
–  таблица ,  предназначенная  для  хранения  адресов  обработчиков  прерываний.    
Базовый  адрес IDT  помещен  IDT  Register.  IDT  столько ,  сколько  процессоров.   
 LDT  (local  descriptor  table)  –  таблица ,  которая  описывает адресное пространство   
процесса. В LDT  Register находится смещение до соответствующего  дескриптора  в  GDT,  описывающего  сегмент,  в  котором находится LDT.  Таблиц  
 LDT столько, сколько процессов. | Формат селектора (явл. ID сегмента):  Индекс  равен  (кратен)  8  и  является  смещением в таблице дескрипторов.   Обязательно наличие 0‐го дескриптора.  0  и 1  биты –  Requested  Privilege  Level,  показывает  на  каком    
уровне  привилегии  работаем (00 – нулевой уровень). 2 бит –  Table indicator,   
0 – адрес в GDT, 1 – в LDT   Селектор указывает на дескриптор сегмента  в таблице дескриптора.  | Формат дескриптора сегмента (GDT):  A – access – бит доступа   
к сегменту  (устан. аппаратно при доступе к сегменту).  Тип – опред. поля доступа:      w: для сегмента кода:    0:  чтение  запрещено  (не  касается      выборки команд)    1: чтение разрешено           для сегмента данных:    0: модиф. запрещ., 1: модиф. разреш.  S – опред., что описывает дескриптор;  DPL – уровень привилегий   
 P – бит присутствия, исп. для работы с ВП.    (0‐сегмента нет в ВП, 1 – есть.)  
 D – бит разрядности операндов и адресов (0 ‐ 16‐разрядные , 1 – 32‐разр.)    
G – бит гранулярности (0 – размер сегмента задан в байтах, 1 – в страницах по 4 Кб).  Процесс не м. выйти за размер своего сегмента – контроль ОС (защита адресных простр‐в сегментов др. от др.).  К дескрипторам GDT и LDT мы обращаемся с помощью селекторов , к дескриптору IDT мы обращаемся по смещению ,  которое берем из прерывания.  | Обработчик прерывания: IDTR (указывает на начало  
 IDT) + смещение из прерывания = дескриптор в IDT.  Из дескриптора в IDT берем селектор. С помощью селектора узнаем, с какой таблицей мы работаем.  1. Если работаем с GDT, то с помощью селектора получаем дескриптор сегмента, в котором находится наш обработчик, в  этом сегменте с помощью смещения из дескриптора в   
IDT мы получаем точку входа в обработчик прерывания.  2. Если работаем с LDT,  
 то с помощью LDTR (в котором у нас смещение до дескриптора сегмента в GDT, в котором находится  LDT) находим этот дескриптор, получаем сегмент. В этом сегменте   
находится нужная LDT, в ней с помощью селектора  получаем  дескриптор  сегмента  в  котором  находится  наш  обработчик ,  в  этом  сегменте  с  помощью  смещения  из  дескриптора в IDT получаем точку входа в обработчик прерывания.

| В процессоре каждому из сегментных регистров (CS, DS, SS, ES, FS, GS) сопоставлен теневой регистр (дескриптора). Он не  доступен программисту и загружается (параллельно) автоматически из таблицы дескрипторов соотв. сегмента чтобы реже  обращаться к ОП.

\*картинка!!!

**Понятие  процесса.  Процесс  как  единица  декомпозиции  системы.  Процессы  и  потоки .  Типы  потоков .  Диаграмма состояний процесса. Планирование и диспетчеризация.** Процесс ‐ программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).   М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.    
Поток  (thread,  нить )  ‐  часть  последовательного  кода  процесса,  которая  м.  выполняться  параллельно с другими частями кода. Не имеет своего адресного пространства.   
Однопоточная и многопоточная модель процесса:  
  
Типы потоков:  
 (1. быстрота переключения; Каждая нить имеет свой контекст; Возм. распараллел. при асинхр. системн. вызовах; Управл. потоками заним. спец. библиотека. | 2 – Независимый поток управ.; выполн. в контексте некоторого процесса; Процесс остаётся владельцем ресурсов)  
Диаграмма состояний процесса:  
  
Классификация планирования: Системы бывают с/без: переключения, вытеснения, приоритетов; Классификация приоритетов: относительные/абсолютные, статические/динамические  | Процесс  выполняется  от  начала  и  до  конца  при  получении    
процессорного  времени  (для  систем  с  однопрограммной монопольной обработкой), либо процесс выполняется до тех пор , пока он не запросит IO,  после чего   
он блокируется в ожидании завершения IO (для систем с мультипрограммной обработкой). Т .о.,  процесс выполняется произв. кол‐во времени (зависит от характера процесса) -> нет гарантир. времени отклика.  Вытеснение  основано  на  системе  с  приоритетами .  Если  все  процессы  равноправны,  то  вытеснения  нет.

**Обеспечение моноп. доступа к разделяемым данным в задаче "писатели‐читатели", используя Win32 API**#include "stdafx.h"

#include <stdio.h>

#include <windows.h>

const DWORD sleep\_time\_for\_writer = 50; // milliseconds for writer

const DWORD sleep\_time\_for\_reader = 30; // milliseconds for reader

const int readers\_count = 5;

const int writers\_count = 3;

const int iterations = 5;

HANDLE mutex; // это дескрипторы

HANDLE can\_read;

HANDLE can\_write;

HANDLE writers[writers\_count];

HANDLE readers[readers\_count];

volatile LONG active\_readers = 0;

bool writing = false;

int value = 0;

void start\_read()

{

if (writing || WaitForSingleObject(can\_write, 0) == WAIT\_OBJECT\_0) {

WaitForSingleObject(can\_read, INFINITE); // захватывают мьютекс

}

//WaitForSingleObject(mutex, INFINITE);

InterlockedIncrement(&active\_readers); //инкремент количества читателей

SetEvent(can\_read); // перевод в свободное состояние

}

void stop\_read()

{

InterlockedDecrement(&active\_readers);

if (active\_readers == 0) {

SetEvent(can\_write);

}

ReleaseMutex(mutex); // освобождает мьютекс

}

void start\_write()

{

WaitForSingleObject(mutex, INFINITE);

if (writing || active\_readers > 0) {

WaitForSingleObject(can\_write, INFINITE);

}

writing = true;

ReleaseMutex(mutex);

}

void stop\_write()

{

writing = false;

if (WaitForSingleObject(can\_read, 0) == WAIT\_OBJECT\_0) {

SetEvent(can\_read);

}

else

SetEvent(can\_write);

}

DWORD WINAPI reader(LPVOID) // точка входа в процесс reader

{

while (value < writers\_count \* iterations) {

start\_read();

//Sleep(sleep\_time\_for\_reader);

printf("R #%ld <---- %d\n", GetCurrentThreadId(), value);

stop\_read();

Sleep(sleep\_time\_for\_reader);

}

return 0; // возврат управления, очистка всех ресурсов

}

DWORD WINAPI writer(LPVOID) // точка входа в процесс writer

{

for (int i = 0; i < iterations; i++) {

start\_write();

//Sleep(sleep\_time\_for\_writer);

printf("W #%ld ----> %ld\n", GetCurrentThreadId(), ++value);

stop\_write();

Sleep(sleep\_time\_for\_writer);

}

return 0; //возврат управления, очистка всех ресурсов

}

int init\_handles()

{

// атрибут защиты, начальное состояние, имя

mutex = CreateMutex(NULL, FALSE, NULL); // (1) - параметр защиты, если null - то по умолчанию; (2) - false, вызвавший поток не владеет мьютексом, (3) - создаётся без имени

if (mutex == NULL) {

perror("!!! Can't create mutex");

return EXIT\_FAILURE;

}

// атрибут защиты, с автосбросом, начальное состояние, имя

can\_read = CreateEvent(NULL, FALSE, TRUE, NULL); // false - самосброс (когда флаг события - true, то он сбрасывается)

if (can\_read == NULL) {

perror("!!! Can't create event can read");

return EXIT\_FAILURE;

}

can\_write = CreateEvent(NULL, FALSE, TRUE, NULL);

if (can\_write == NULL) {

perror("!!! Can't create event can write");

return EXIT\_FAILURE;

}

return EXIT\_SUCCESS;

}

int create\_threads()

{

for (int i = 0; i < writers\_count; i++) {

// атрибут защиты,размер стека потока, адрес функции потока, значение любое, флаг запуска, идентификатор потока

writers[i] = CreateThread(NULL, 0, writer, NULL, 0, NULL);

if (writers[i] == NULL) {

perror("!!! Can't create writer");

return EXIT\_FAILURE;

}

}

for (int i = 0; i < readers\_count; i++) {

readers[i] = CreateThread(NULL, 0, reader, NULL, 0, NULL);

if (readers[i] == NULL) {

perror("!!! Can't create reader");

return EXIT\_FAILURE;

}

}

return EXIT\_SUCCESS;

}

int main()

{

int code = EXIT\_SUCCESS;

if ((code = init\_handles()) != EXIT\_SUCCESS) {

return code;

}

if ((code = create\_threads()) != EXIT\_SUCCESS) { // создаём потоки

return code;

}

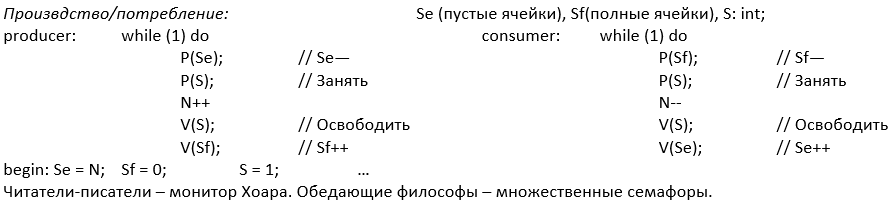
WaitForMultipleObjects(writers\_count, writers, TRUE, INFINITE); //ждёт либо все (true), либо хотя бы один (false) объект из массива

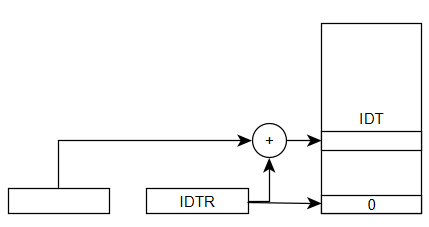
WaitForMultipleObjects(readers\_count, readers, TRUE, INFINITE); // функця ждёт, когда освободятся все объекты из набора

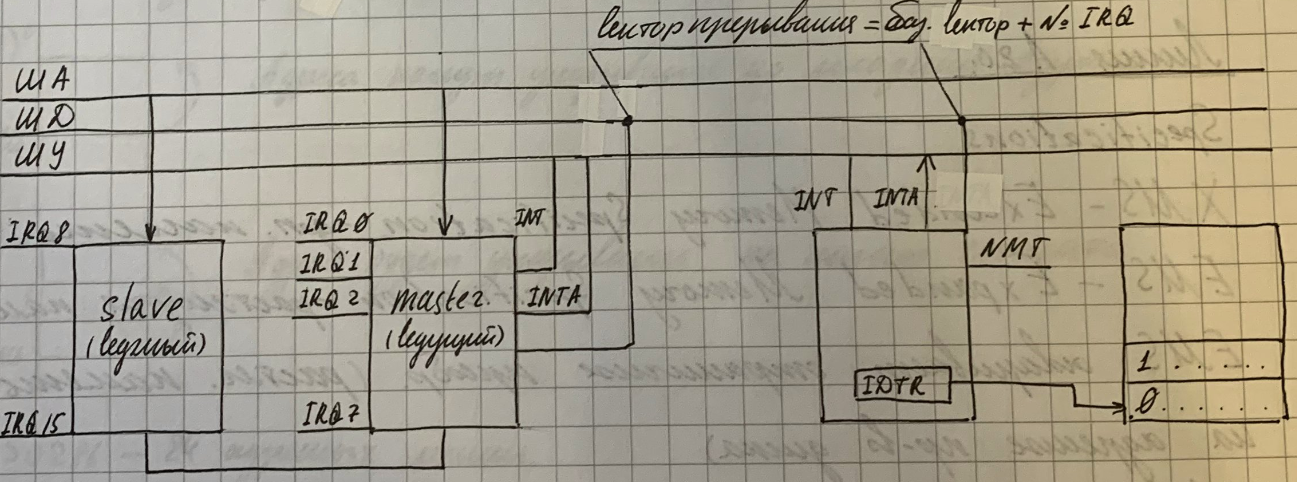
CloseHandle(mutex); CloseHandle(can\_read); CloseHandle(can\_write);

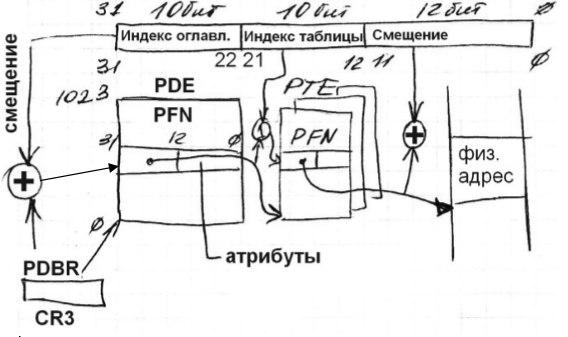
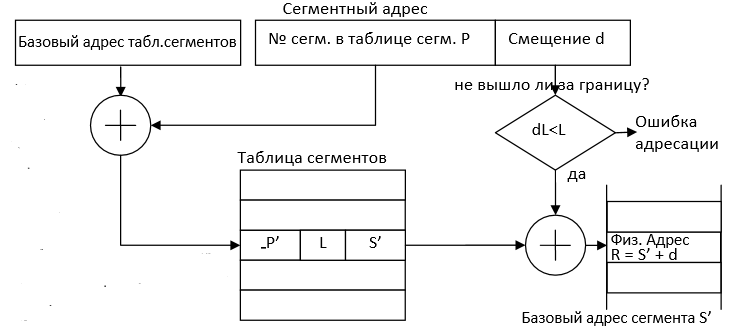
return code;

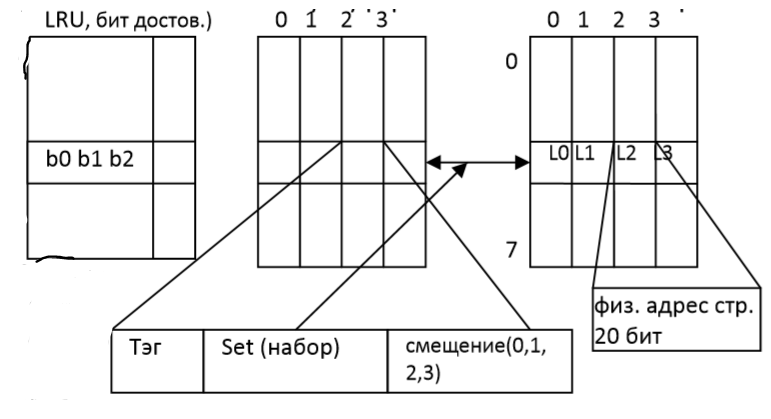
}

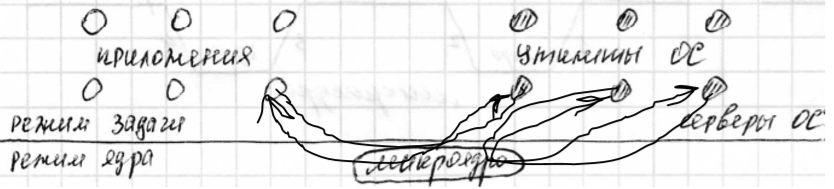
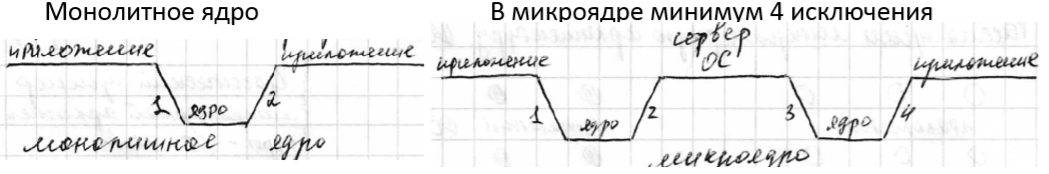
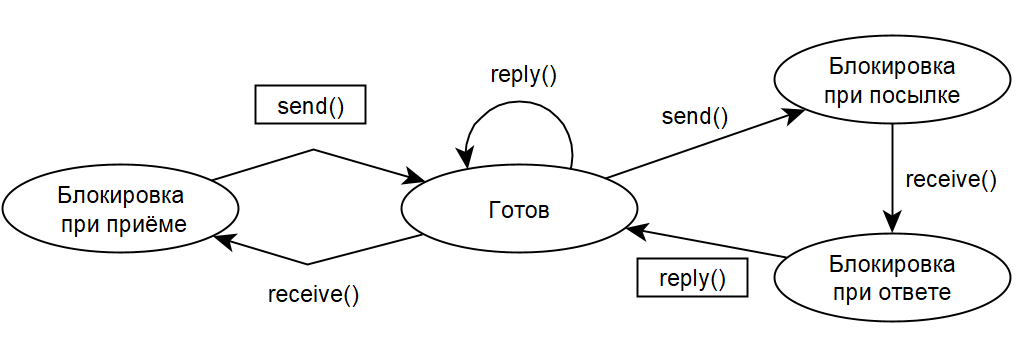
**Взаимоисключение и синхронизация процессов и потоков. Семафоры: определение, виды, примеры.** Процессам часто нужно взаимодействовать друг с другом,  
 например , один процесс может передавать  данные другому процессу, или несколько  
 процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов. Она связана с потерей доступа к параметрам  
из‐за их  некорректного разделения. В каждый момент времени на процессоре выполняется 1 процесс. Каждому процессу выделяется квант процессорного времени. Разделяемый ресурс ‐ переменная, к которой обращаются разные процессы.  Критическая секция – область кода, из которой осуществляется доступ к разделяемому ресурсу.  Монопольное использование – если процесс получил доступ к разделяемому ресурсу , то др . процесс не м .  получить доступ к этому ресурсу. Активное ожидание на процессоре – ситуация, когда процесс занимает процессорное время, проверяя  
значение  флага  (занятости  ресурса  другим  процессом).  Активное ожидание    
на  процессоре  является  неэффективным  использованием процессорного времени.   
Возможные варианты развития событий:  1. Возможно, что оба процесса пройдут цикл ожидания и попадут в свои критические секции ‐ ок; 2. Возможно бесконечное откладывание (зависание) – ситуация , когда разделённый ресурс снова  захватывается тем же процессом; 3. Тупик (deadlock, взаимоблокировка) – ситуация, когда оба процесса установили флаги занятости и  ждут. Т.е. каждый ожидает освобождения ресурса, занятого другим процессом | Семафор – неотрицательная защищённая  
переменная S, над которой определено 2 неделимые операции:  Р (пропустить) и V (освободить). Защищенность семафора означает, что значение семафора м. изменяться только операциями P и V.  Необходимо обеспечить монопольный доступ процесса к разделяемому ресурсу до тех пор, пока процесс его   
не освободит. Все алгоритмы программной реализации обобщил Дейкстра, введя понятие семафора.  1. Операция P(S): S = S – 1. Декремент семафора (если он возможен). Если S = 0, то процесс, пытающийся  выполнить операцию Р, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его  освобождает другой процесс, выполняющий операцию V(S); 2. Операция V(S): S = S + 1. Инкремент S одним неделимым действием (последовательность непрерывных  действий: инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других  процессов. Если S = 0, то V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожид. на семафоре.  P(S) и V(S) есть неделимые (атомарные) операции.   
Суть: процесс пытающий выполнить операцию P(S) блокируется, становится в очередь ожидания данного семафора, освобождает  его другой процесс, который выполняет V(S). Таким образом исключается активное ожидание.  Семафоры   
устраняют активное ожидание на процессоре. Семафоры бывают бинарные (S от 0 и 1) , считающие (S от 0 до n) , множественные (набор считающих семафоров). Процесс может создать семафор и изменять его. Удалить семафор может только процесс, создавший его, либо  привилегированный процесс. В Windows после освобождения на семафоре приоритет процесса повышается.  При проверке флага мы не переходим в режим ядра. | Примеры использования: Производство‐потребление – считающие – буфер пуст и полон, один бинарный (монитор – кольцевой буфер).   


**Режимы работы процессоров Intel Pentium. Прерывания в защищенном режиме (таблица ID).**  Реальный режим (или режим реальных адресов) ‐это название было дано прежнему способу адресации памяти после появления 286‐го процессора, поддерживающего защищённый режим. Реальный режим поддерживается аппаратно. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров. Однопроцессорный режим под управлением MS DOS (нет многозадачности). Компьютер начинает работать в реальном режиме. 16 разрядов , 20‐разрядный адрес – сегмент/смещение. Диапазон адресов памяти ограничен 1МБ (2^20 = FFFFF = 1024 Кб = 1Мб). Минимальная адресная единица памяти – байт. | Защищенный  режим  –  многопроцессный  режим. 32х‐разрядный. Увеличение адресного пространства до 4ГБ. Возможность работать в виртуальном адресном пространстве. Организация многозадачного режима с параллельным выполнением нескольких программ(процессов); (многопоточность и многопроцессность); 4х уровненная система привилегий; Страничная организация памяти (повышает уровень защиты задач друг от друга и эффективность их выполнения).   
Для обращения к сегменту в сегментный регистр заносится селектор(а не сегментный адрес). В селектор входит номер(индекс) соответствующего сегмента дескриптора. Процессор по этому номеру находит нужный дескриптор, извлекает из него базовый адрес сегмента и, прибавляя смещение (указанное в конкретной команде)(относительный адрес), формирует адрес ячейки памяти. !Смещение берется из команды!  
Теневые регистры - регистры, которые недоступны программисту; Они автоматически загружаются процессором из таблицы дескрипторов каждый раз, когда процессор инициализирует соответствующий сегментный регистр. Теневые регистр ≠ КЭШ. Теневые рег. находятся в процессоре. Сделаны для сокращения обращений программы к оперативной памяти. В процессоре для каждого из сегментных регистров имеется теневой регистр дескриптора. | Специальный  режим  защищенного  режима  (V86)  –  процессор фактически  продолжает использовать схему преобразования адресов  памяти  и  средства  мультизадачности  защищённого  режима.  В  виртуальном  режиме  используется  трансляция  страниц  памяти.  Это  позволяет  в  мультизадачной  операционной  системе  создавать несколько задач, работающих в виртуальном режиме. Каждая из этих задач может иметь собственное  адресное пространство, каждое размером в 1 мегабайт. Все задачи виртуального режима обычно выполняются в  третьем, наименее привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор  автоматически переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в  операционную систему, работающую в защищённом режиме.   | Прерывания в защищенном режиме:  IDT  (interrupt  descriptor  table)  –  таблица,  предназначенная  для  хранения  адресов  обработчиков  прерываний.  Базовый  адрес IDT  помещен  IDT  Register.  IDT  столько ,  сколько  процессоров. IDTR  (указывает  на  начало  IDT)  +  смещение  из  прерывания = дескриптор в IDT.  Из дескриптора в IDT берем селектор. С помощью  селектора узнаем, с какой таблицей мы работаем.  Если  работаем  с  GDT,  то  с  помощью  селектора  получаем  дескриптор  сегмента,  в  котором  находится  наш  обработчик, в этом сегменте с помощью смещения из дескриптора в IDT мы получаем точку входа в обработчик  прерывания. Если работаем с LDT, то с помощью LDTR (в котором у нас смещение до дескриптора сегмента в  GDT, в котором находится LDT) находим этот дескриптор,  
 получаем сегмент. В этом сегменте находится нужная  LDT, в ней с помощью   
селектора получаем дескриптор сегмента, в котором находится наш обработчик, в этом  сегменте с помощью смещения из дескриптора в IDT получаем точку входа в обработчик прерывания.   


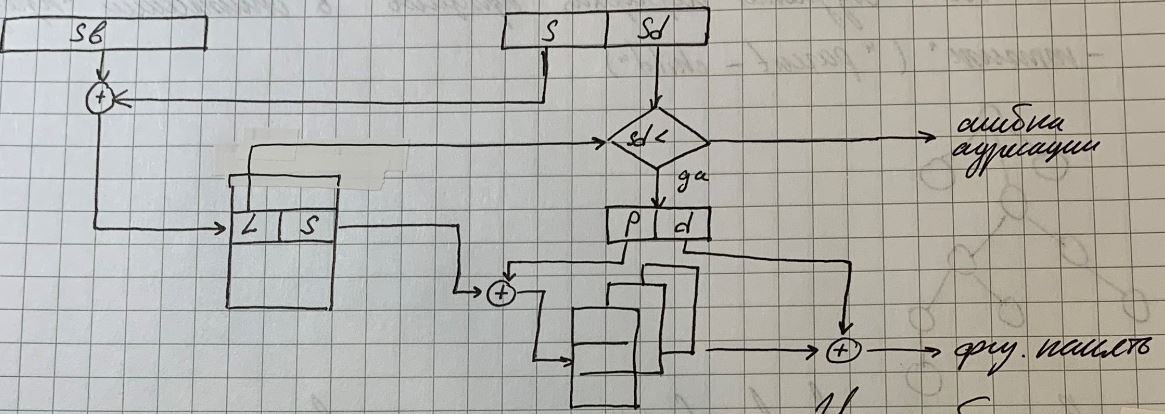
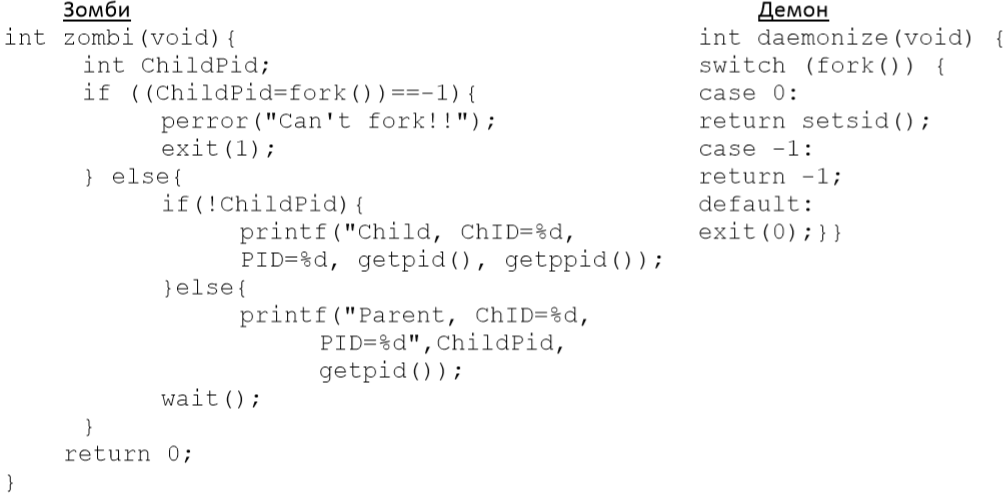
**Методы организации ввода‐вывода: программируемый, с прерываниями, прямой доступ к памяти.**   
1) Программируемый. Когда процессору встречается команда вв /выв, выполняя эту команду, процессор вызывает  библиотечную функцию, кот . переводит процессор в режим ядра, осуществляет передачу команды вв /выв. Контроллер  выполняет необходимые действия, затем устанавливает соответствующие биты в регистрах состояний вв/выв. Процессор д.  периодически  проверять  состояние  этих  битов, т .е.  опрашивать  соответствующие  регистры  контроллера.  На  процессор  возложено непосредственное управление вв/выв: опрос битов , пересылка команд чтения/записи, передача данных. Т .о.  необходимо иметь следующий набор команд: команды управления (исп. для инициализации внешних устройств, указывают им, что делать); команды анализа состояния (проверка состояний флагов вв/выв); передача данных (для чт/зап команд или данных из/в внешн. устр. в/из регистры процессора). Недостаток: процессор постоянно опрашивает флаги – остается мало времени для остального.  | 2. С использованием прерываний. Прерывание от устройства вв/выв поступает, когда оно завершило процесс вв/выв.  Процессор  освобождается  от  проверки  флагов  и  м.  переключиться  на  др.  работу.  Метод  требует  включ .  в  состав  ОС  контроллера прерываний. Контроллер  прерываний  посылает  по  шине  управления сигнал .  В  конце  выполнения  каждой  команды  процессор  проверяет  входной  сигнал с шины управления (если прерывания не  замаскированы  в  ОС).  Если  получен  сигнал  –  посылается  ответный  сигнал  контроллеру  прерываний,  в  ответ  контроллер  прерываний  формирует и посылает вектор прерывания. Вектор  передается по шине данных. Полученный вектор  используется для процедуры обработки прерыв. Недостатки:  Кажд. прерыв. вызывает остановку выполнен. текущих процессов и сохранение аппаратного  контекста,  любое  слово ,  передающ .  между  памятью  и  устройством, д .  пройти  через  регистры  процессора.  Но  все  равно  намного  эффективнее  программируемого, т .к.  искл .  активное ожидание на процессоре. | 3.Прямой доступ к памяти Для управления ПДП  подключ. контроллер ПДП  (программно  управл.  устр .  –  спецпроцессор).  Устройству  вв /выв  или  каналу временно передается управление памятью (обычно на время,  равное 1 циклу памяти), благодаря чему слово или группа слов м.б.  записана  в  память .  Процессор  своей  памяти  не  имеет,  поэтому  все  осущ. через ОП. Передача – через регистры процессора.

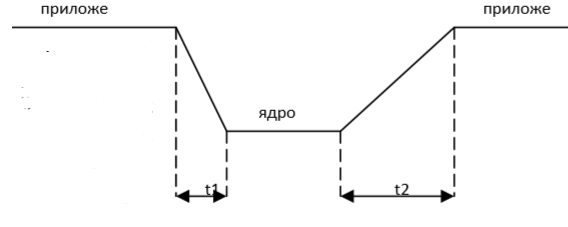
**Защищенный режим. EMS. Преобразование адреса при страничном преобразовании в процессорах Intel** Защищенный  режим  –  многопроцессный  режим. 32х‐разрядный. Увеличение адресного пространства до 4ГБ. Возможность работать в виртуальном адресном пространстве. Организация многозадачного режима с параллельным выполнением нескольких программ(процессов); (многопоточность и многопроцессность); 4х уровненная система привилегий; Страничная организация памяти (повышает уровень защиты задач друг от друга и эффективность их выполнения).   
Для обращения к сегменту в сегментный регистр заносится селектор(а не сегментный адрес). В селектор входит номер(индекс) соответствующего сегмента дескриптора. Процессор по этому номеру находит нужный дескриптор, извлекает из него базовый адрес сегмента и, прибавляя смещение (указанное в конкретной команде)(относительный адрес), формирует адрес ячейки памяти. !Смещение берется из команды! |   
EMS (Expanded Memory Specification, спецификация отображаемой памяти) – стандарт, разработанный в 1985 г.  фирмами Lotus, Intel и Microsoft для доступа из DOS к областям памяти выше 1 Мбайт в системах на базе  процессоров 80386 и более поздних. (расширенная, растягиваемая). Страничное преобразование может быть включено или не включено; сегментное преобразование всегда включено. Если страничное преобразование включено, то используется пейджинг страниц, а не свопинг. Размер сегмента равен размеру страницы (4Кб). Система EMS в основном предназначена для хранения данных. Если преобразование занимает несколько сегментов, значит должно быть столько таблиц страниц, сколько у него сегментов. PDE – page directory entry (каталог); PFN – page frame number; PTE – page table entry; PDBR – page directory base register (CR3)   
В Intel кэш TLB хранятся адреса страниц, к которым были последние последние обращения.  
  
  
**Управление  памятью:  распределение  памяти  сегментами  по  запросам;  стратегии  выделения  памяти;  фрагментация**  При  страничной  организации  виртуальное  адресное  пространство  процесса  делится  механически  на  равные части. Это не позволяет дифференцировать способы доступа к разным частям программы (сегментам). | Виртуальное  адресное  пространство  процесса  делится  на  сегменты,  размер  которых  определяется  программистом с учетом смыслового значения содерж. в них информации. Отдельный сегмент м. представлять  собой подпрограмму, массив данных и т.д. При загрузке процесса часть сегментов помещ. в оперативную память  (для каждого из этих сегментов ОС выбирает подходящий участок свободной памяти), а часть сегментов размещ.  в дисковой памяти. Сегменты одной программы могут занимать в оперативной памяти несмежные участки.  Необх. контролир.  размер сегмента.  Необх. контролир. выход за   размер сегмента.  Каждый сегмент описывается   дескриптором сегмента.  (содержит поле адреса, с которого  сегмент  начинается  и  поле  длины  сегмента). Благодаря этому можно  осуществлять контроль:  1) размещения сегментов без наложен. др. на др.  2) обращается  ли  код  исполняющейся  задачи  за  пределы текущего сегмента.  В дескрипторе содержатся также данные о правах   доступа к сегменту (запрет на модификацию, можно ли его предоставлять другой задаче).   
  
Типы организации таблиц сегментов: 1) Единая таблица  – у сегмента есть единственное имя, при этом дескриптор содерж . список прав доступа  каждого пользователя сегмента; 2) Локальные таблицы – каждая локальная таблица описывает определ. адресное  пространство (опред. среду). В разн. средах 1 и тот же сегмент имеет разн. имена.    
3) Локальные  таблицы +  глобальные  таблицы  –  каждый  сегмент  имеет  2  ID  –  локальный и глобальный. Локальная таблица ссылается на дескриптор сегмента в  глобальной таблице. Физич. хар‐ки описываютсяв главной таблице.   
Алгоритмы, используемые при замещении сегментов: 1) Самый широкий; 2) Самый узкий; 3) Первый подходящий | Способы распределения памяти:  1)Связанное распределение  (Одиночное – программа загруж. в память целиком в последовательные адреса. Запрещ. обращ.  программы в ОП ОС (регистр границы); Статическими разделами – размер разделов определяется в момент загрузки ОС; Динамическими разделами –разделы до начала выполнения задачи не устанавливаются); 2) Несвязанное выделение (Несвязанные разделы опред. размера (страницы (д.б. таблицы страниц); Поделить память на логические единицы (сегменты) )

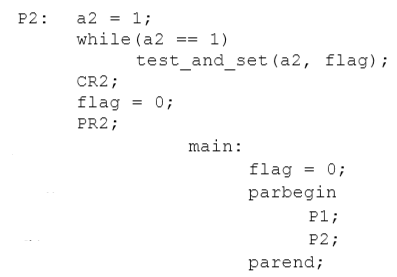
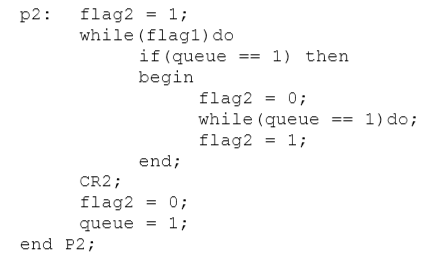
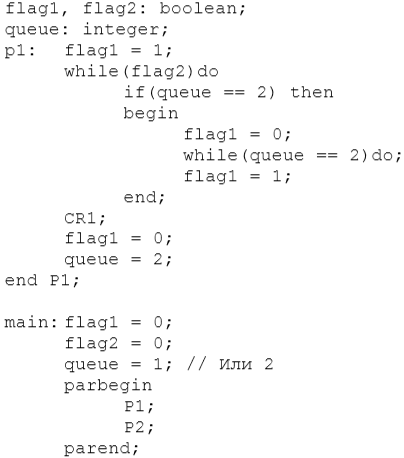
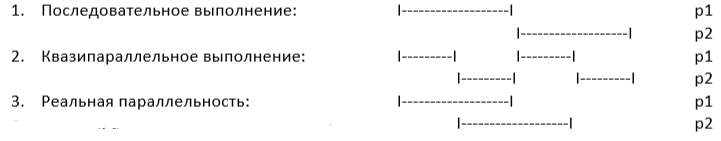
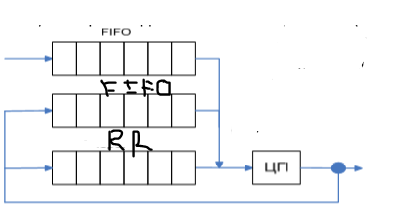
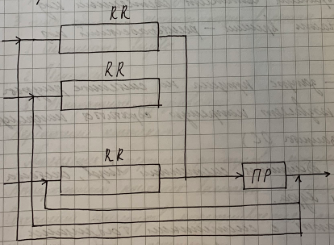
**Режимы работы компьютера IBM PC, кэши TLB и данных** Реальный режим (или режим реальных адресов) ‐это название было дано прежнему способу адресации памяти после появления 286‐го процессора, поддерживающего защищённый режим. Реальный режим поддерживается аппаратно. Необходим для обеспечения функционирования программ, разработанных для старых моделей, в новых моделях микропроцессоров. Однопроцессорный режим под управлением MS DOS (нет многозадачности). Компьютер начинает работать в реальном режиме. 16 разрядов , 20‐разрядный адрес – сегмент/смещение. Диапазон адресов памяти ограничен 1МБ (2^20 = FFFFF = 1024 Кб = 1Мб). Минимальная адресная единица памяти – байт. | Защищенный  режим  –  многопроцессный  режим. 32х‐разрядный. Увеличение адресного пространства до 4ГБ. Возможность работать в виртуальном адресном пространстве. Организация многозадачного режима с параллельным выполнением нескольких программ(процессов); (многопоточность и многопроцессность); 4х уровненная система привилегий; Страничная организация памяти (повышает уровень защиты задач друг от друга и эффективность их выполнения).   
Для обращения к сегменту в сегментный регистр заносится селектор(а не сегментный адрес). В селектор входит номер(индекс) соответствующего сегмента дескриптора. Процессор по этому номеру находит нужный дескриптор, извлекает из него базовый адрес сегмента и, прибавляя смещение (указанное в конкретной команде)(относительный адрес), формирует адрес ячейки памяти. !Смещение берется из команды!  
Теневые регистры - регистры, которые недоступны программисту; Они автоматически загружаются процессором из таблицы дескрипторов каждый раз, когда процессор инициализирует соответствующий сегментный регистр. Теневые регистр ≠ КЭШ. Теневые рег. находятся в процессоре. Сделаны для сокращения обращений программы к оперативной памяти. В процессоре для каждого из сегментных регистров имеется теневой регистр дескриптора. | Специальный режим защищенного режима (V86): ОС реального режима (не DOS, а спец код выполнения) запускаются как задачи. V86 - специальный режим защищенного режима. ОС реального режима (не DOS, а спец код выполнения) запускаются как задачи. Создается VM реального режима, в каждой из которых выполняется по 1 программе реального времени. V86 присуще: многозадачность, поддержка виртуальной памяти, у каждой задачи собственное адресное пространство размером 1МБ. Процессор фактически продолжает использовать схему преобразования адресов памяти и средства мультизадачности защищённого режима. В виртуальном режиме используется трансляция страниц памяти. Это позволяет в мультизадачной операционной системе создавать несколько задач, работающих в виртуальном режиме. Все задачи виртуального режима обычно выполняются в третьем, наименее привилегированном кольце защиты. Когда в такой задаче возникает прерывание, процессор переключается из виртуального режима в защищённый. Поэтому все прерывания отображаются в операционную систему, работающую в защищённом режиме. | TLB (translation look‐aside buffer, буфер быстрого преобразования адреса,  буфер предварительной трансляции) –  таблица в блоке управл. памятью, отвечающая за преобразование виртуальных адресов в физические (в чипе). TLB представляет собой четырехканальную ассоциативный по множеству буфер – кэш. В КЭШе TLB хранятся  адреса  
страниц, к кот. были посл. обращения.  В  блоке  данных  находится  8  наборов  по  4  элемента данных в каждом. Элемент данных в TLB  состоит из 20 битов старшего порядка физического  адреса.  Эти 20  битов  могут  интерпретироваться  как  базовый  адрес  страницы,  который  по  опр.  имеет 12 очищенных битов   
младшего порядка.  TLB транслирует линейный адрес в физический и работает  только со старшими 20 битами каждого из них; младшие  12 битов (представляющие собой смещение в странице)  одинаковы как для линейного адреса, так и для физич.   Блоку элементов данных соответствует блок элементов  достоверности, атрибутов и тега (признака). Если соответствие найдено среди тегов выбранного набора, а соответствующий бит достоверности равен 1, то  линейный адрес транслируется заменой старших 20 битов на 20 битов соответствующего элемента данных.  Каждому набору   
соответствует три бита LRU: они отслеживают используемость данных в наборе и проверяются  при необходимости в новом элементе (а также следят за достоверностью всех элементов в наборе). b0 b1 b2 –  для реализации алгоритма псевдо‐LRU, послед бит – бит достоверности. При очистке кэша или сбросе процессора  все биты достоверности сбрасываются в 0. При записи ищется любая недостоверная строка.  
 

**Классификация структур ядер ОС. Особенности ОС с микроядром. Модель клиент‐сервер. Три состояния  процесса при передаче сообщений. Достоинства и недостатки микро‐ядерной архитектуры.** ОС разбивается на несколько уровней, причем обращение через уровень невозможно (непрозр. интерфейс).   В качестве ядра выделяется самый низкий уровень распределения аппаратных ресурсов процессам самой ОС  (непосредственное обращение с аппаратурой).  Существует два типа структур ядер: 1. Монолитное ядро: ядро – это все, что выполняется в режиме ядра; ядро представляет собой единую программу с модульной структурой (выделены функции, такие как  планировщик, файловая система, драйверы, менеджеры памяти); при изменении любой функции нужно перекомпилировать все ядро; Для ОС с монолитным ядром характерна система прерываний; Монолитное – программа, имеющая модульную структуру (состоит из подпрограмм); 2. Микроядро: В микроядерной архитектуре все компоненты ОС являются самостоятельными программами которые выполняются в разных адресных пространствах; Взаимодействие между компонентами ОС выполняются с помощью посылки и приема сообщений; 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) и диспетчеризация потоков (простейший процесс имеет один поток). Микроядерная  архитектура  основана  на  модели клиент‐сервер (К серверам ОС относятся: сервер файлов, процессов, безопасности, виртуальной памяти)  
  
  
Модель «клиент-сервер»: Система рассматривается как совокупность двух групп процессов: процессы‐серверы, предоставляющие набор сервисов и процессы‐клиенты, запрашивающие сервисы. Принято считать, что данная модель работает на уровне транзакций (запрос и ответ – неделимая операция).  *Плюсы:* Высокая  степень  модульности  ядра  ОС  (упрощает  добавление  новых  компонентов). М .  загружать  и  выгружать нов. драйверы, файловые системы и т. д., т.о. упрощается процесс отладки компонентов ядра; Компоненты ядра ОС принципиально не отличаются от пользовательских программ -> для их отладки  можно применять обычные средства; Повышается    
надежность  системы, т .к.  ошибка  на  уровне  непривилег.  программы  <  опасна ,  чем  отказ  на  уровне режима ядра.  *Минусы:* Микроядерная архитектура ОС вносит  дополнительные  накладные  расходы,  связанные с передачей сообщений, что  сущ. влияет на производительность; Для того чтобы микроядерная ОС   по скорости не уступала ОС на базе монолитного ядра, треб. очень аккуратно проектировать разбиение  системы на компоненты, стараясь минимизировать взаимодействие между ними.   


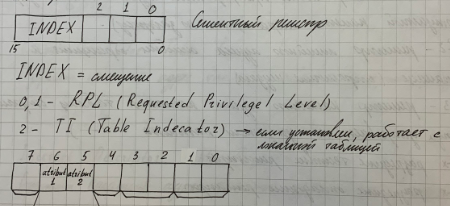
**Unix: команды fork(), wait(), exec(), pipe(), signal().** Unix создавалась как ОС разделения времени.  Базовое  понятие  Unix  –  процесс (единица  декомпозиции  ОС,  программа  времени  выполнения ).  Процесс рассматривается  как  виртуальная    
машина  с  собств.  адресным  пространством,  выполн .  пользов.  прогр., предоставл. набор услуг. Процесс может находиться в двух состояниях – «задача» (процесс выполняет  собственный  код)  и «система»  или «ядро»(выполняет  реентерабельный  код  ОС).  Процессы  все  время  переходят «пользователь» 🡨🡪 «система». Unix – ОC с динамическим управлением процессами | FORK(): fork() создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0. Блокировки файлов и сигналы, ожидающие обработки, не наследуются. В старых UNIX системах код предка копировался в адресное пространство потомка. Оптимизация: Под Linux fork реализован с помощью "копирования страниц при записи" (copy-on-write, COW): копируется таблица страниц для потомка, но ссылаются на адресное пространство предка. Права доступа меняется на read-only, устанавливается флаг copy-on-write. Если предок/потомок пытаются изменить страницу, то возникает исключение по правам доступа. Выполняя это исключение супервизор обнаружит флаг copy-on-write и создаст копию страницы в адресном пространстве того процесса, который пытался ее изменить. | WAIT(): Системный вызов wait приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Считывает код завершения потомка. Код завершения записывается по адресу, переданному в качестве параметра. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби" ("zombie")), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Cистемный вызов wait() блокирует родительский процесс до момента завершения дочернего. При этом процесс-предок получает статус завершения процесса-потомка. **#include<sys/types.h> #include <sys/wait.h> pid\_t wait(int \*stat\_loc);** Вызов возвращает PID дочернего процесса. Обычно это дочерний процесс, который завершился. Сведения о состоянии позволяют родительскому процессу определить статус завершения дочернего процесса, т.е. значение, возвращенное из функции main потомка или переданное функции exit(). Есть еще один системный вызов, который можно применять для ожидания дочернего процесса. Он называется waitpid() и применяется для ожидания завершения определенного процесса. Аргумент pid — конкретный дочерний процесс, окончания которого нужно ждать. Если он равен –1, waitpid() возвращает информацию о любом дочернем процессе. | EXEC(): Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов exec(), параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов exec() создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в exec(), но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов exec() создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц. В результате системного вызова exec() адресное пространство процесса будет заменено на адресное пространство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы. Например: системный вызов exec() запускает команду ls.  
if (fork()==0) wait(0); else execl("ls", "ls", 0); /\* порожденный процесс \*/ |   
PIPE(): Системный вызов pipe() создает неименованный программный канал. Неименованные программные каналы могут использоваться для обмена сообщениями между процессами родственниками. В отличие от именованных программных каналов неименованные не имеют идентификатора, но имеют дескриптор. Процесс-потомок наследует все дескрипторы открытых файлов процесса-предка, в том числе и неименованных программных каналов. Программные каналы имеют встроенные средства взаимоисключения — массив файловых дескрипторов: из канала нельзя читать, если в него пишут, и в канал нельзя писать, если из него читают. Для этого определяется массив файловых дескрипторов, как показано в примере:  
int fd[2]; pipe(fd);  
if (( pid=fork())<0)  
{ err\_sys(“Error fork()”);}  
else if (pid==0){ /\*child\*/ close(fd[0]); write(fd[1], … ); }  
else { /\*parent\*/ close(fd[1]); read(fd[0], … ); }  
| SIGNAL(): Сигнал - способ информирования процесса ядром о происшествии какого-то события. Если возникает несколько однотипных событий, процессу будет подан только один сигнал. Сигнал означает, что произошло событие, но ядро не сообщает сколько таких событий произошло. Установить реакцию на поступление сигнала можно с помощью системного вызова signal *func = signal(snum, function);*  
snum - номер сигнала, а function - адрес функции, которая должна быть выполнена при поступлении указанного сигнала. Возвращаемое значение - адрес функции, которая будет реагировать на поступление сигнала. С помощью системного вызова kill() можно сгенерировать сигналы и передать их другим процессам. *kill(pid, snum);* где pid - идентификатор процесса, а snum - номер сигнала, который будет передан процессу. Обычно kill() используется для того, чтобы принудительно завершить ("убить") процесс.  
# include <iostream.h> # include <signal.h> void catch\_sig(int sig\_numb){ signal(sig\_numb, catch\_sig); cout<<”catch\_sig”<<sig\_numb<<endl;}  
int main(void){ signal(SIGNERM, catch\_sig); signal(SIGINT, SIG\_IGN); signal(SIGSEGV, SIG\_DFL);}

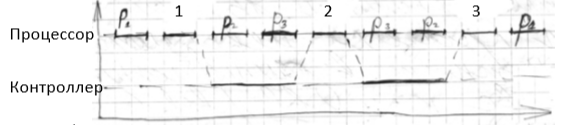
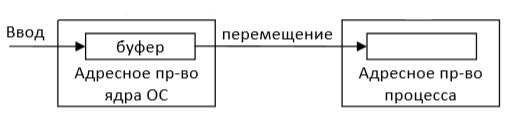
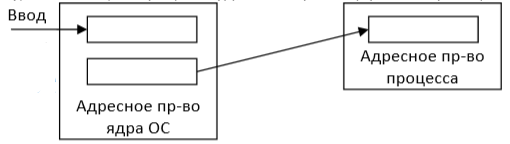
**Виртуальная память: сегментно-страничное распределение   
памяти по запросам. Достоинства и недостатки** Виртуальная память – память,  
 размер которой превышает размер реального физического пространства.  Используется адресное пространство диска как область свопинга или пейджинга, т.е. для   
временного хранения  областей памяти. Подходы к реализации управления   
виртуальной памятью: 1. страничное распределение памяти по запросам; 2. сегментное распределение памяти по запросам; 3. сегментно ‐ страничное   
распределение памяти по запросам. Сегмент представляется в виде совокупности   
страниц, что позволяет устранить проблемы , связанные с  перекомпоновкой  и  ограничением  размера  сегмента. В  системе  с  сегментно‐страничной  организацией    
применяется  трехкомпонентная  (трехмерная)  адресация.  Виртуальный  адрес  определяется  как  упорядоченная  тройка v=(s,p,d),  где s‐  номер  сегмента,  p‐  номер страницы в сегменте, d‐ смещение в странице, по которому находится нужный элемент.   
Обращение к виртуальному адресу при отсутствии его в физической  памяти  может  вызвать  следующие  действия  связанные  с  соответствующими типами прерываний или особых случаев: 1. Прерывание  по  особому  случаю  при  связывании.    
Данный  тип  прерывания возникает при отложенном связывании; 2. Сегмент,   
к которому пытается обратиться процесс, отсутствует в   основной памяти. Менеджер памяти найдет нужный «виртуальный» сегмент, сформирует для него таблицу страниц; 3. Когда сегмент находится в памяти, обращение к таблице страниц может показать,  
 что нужная страница отсутствует в памяти. Менеджер памяти возьмет, найдет нужную страницу во внешней памяти и загрузит ее в основную память; 4. Как и при чисто   
сегментном распределении, адрес виртуальной памяти может выйти за границу сегмента. В этом  случае произойдет прерывание по выходу за границу сегмента; 5. Если   
контроль  по  признакам  доступа  к  сегменту  показывает ,  что  операция ,  запрашиваемая  по  указанному  виртуальному адресу, не разрешена, произойдет прерывание по защите сегмента. | + страничного:  легко реализовать, алгоритм LRU в этом   
достаточно эффективен; - страничного: сложность коллективного использования   
+ сегментного: легко реализовать коллективное использование, т.к. сегмент является   
логической единицей деления памяти; - сегментного:  необх. корректировки таблицы   
дескрипторов всех процессов при изменении размеров сегментов; сложн. при загрузке   
новых сегментов (в памяти должно сущ. адр. пространство необходимого размера); фрагментация  
  
  
  
**Unix: концепция процессов;  процессы ‐ «сироты», процессы «зомби», демоны; пример** Unix создавалась как ОС разделения времени.  Базовое  понятие  Unix  
  –  процесс (единица  декомпозиции  ОС,  программа  времени  выполнения ).  Процесс рассматривается  как  виртуальная  машина  с  собств.  адресным  пространством, выполн .  пользов.  прогр., предоставл. набор услуг. Процесс может находиться в двух состояниях – «задача» (процесс выполняет  собственный  код)  и «система»  или «ядро»(выполняет  реентерабельный  код  ОС).  Процессы  все  время  переходят «пользователь» 🡨🡪 «система». Unix – ОC с динамическим управлением процессами | FORK(): fork() создает процесс-потомок, который отличается от родительского только значениями PID (идентификатор процесса) и PPID (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в 0. Блокировки файлов и сигналы, ожидающие обработки, не наследуются. В старых UNIX системах код предка копировался в адресное пространство потомка. Оптимизация: Под Linux fork реализован с помощью "копирования страниц при записи" (copy-on-write, COW): копируется таблица страниц для потомка, но ссылаются на адресное пространство предка. | WAIT(): Системный вызов wait приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Считывает код завершения потомка. Код завершения записывается по адресу, переданному в качестве параметра. Если дочерний процесс к моменту вызова функции уже завершился (так называемый "зомби" ("zombie")), то функция немедленно возвращается | EXEC(): Чаще всего нет смысла в выполнении двух одинаковых процессов и потомок сразу выполняет системный вызов exec(), параметрами которого является имя исполняемого файла и, если нужно, параметры, которые будут переданы этой программе. Говорят, что системный вызов exec() создает низкоуровневый процесс: создаются таблицы страниц для адресного пространства программы, указанной в exec(), но программа на выполнение не запускается, так как это не полноценный процесс, имеющий идентификатор и дескриптор. Системный вызов exec() создает таблицу страниц для адресного пространства программы, переданной ему в качестве параметра, а затем заменяет старый адрес новой таблицы страниц. В результате системного вызова exec() адресное пространство процесса будет заменено на адресное пространство новой программы, а сам процесс будет возвращен в режим задачи с установкой указателя команд на первую выполняемую инструкцию этой программы. Например: системный вызов exec() запускает команду ls.  
if (fork()==0) wait(0); else execl("ls", "ls", 0); /\* порожденный процесс \*/ |   
Процесс-сирота: Системный вызов fork() создает новый процесс – процесс-потомок. Связь родитель – потомок создает иерархию процессов ( см. указатели на потомков в struct proc ). Если родительский процесс завершается раньше своих потомков, то в системе выполняется так называемое усыновление: процесс-потомок усыновляется процессом с идентификатором 1 ( процессом «открывшим» терминал и создавшим терминальную группу). В Linux Ubuntu процесс-потомок усыносляется процессом-посредником systemd –user. | Процессы «зомби»: Применение вызова fork() для создания процессов может оказаться очень полезным, но необходимо отслеживать дочерние процессы. Когда дочерний процесс завершается, связь его с родителем сохраняется до тех пор, пока родительский процесс в свою очередь не завершится нормально, или не вызовет wait(). Следовательно, запись о дочернем процессе не исчезает из таблицы процессов немедленно. Становясь неактивным, дочерний процесс все еще остается в системе, поскольку его код завершения должен быть сохранен, на случай если родительский процесс в дальнейшем вызовет wait. Процесс-зомби – это процесс, у которого отобраны все ресурсы, кроме последнего – строки в таблице процессов. | Демон - процессы спящие большую часть времени; компьютерная программа в системах класса UNIX, запускаемая самой системой и работающая в фоновом режиме без прямого взаимодействия с пользователем. Демоны обычно запускаются во время загрузки системы. Типичные задачи демонов: серверы сетевых протоколов (HTTP, FTP, электронная почта и др.), управление оборудованием, поддержка очередей печати, управление выполнением заданий по расписанию и т. д.   
В техническом смысле демоном считается процесс, который не имеет управляющего терминала.  


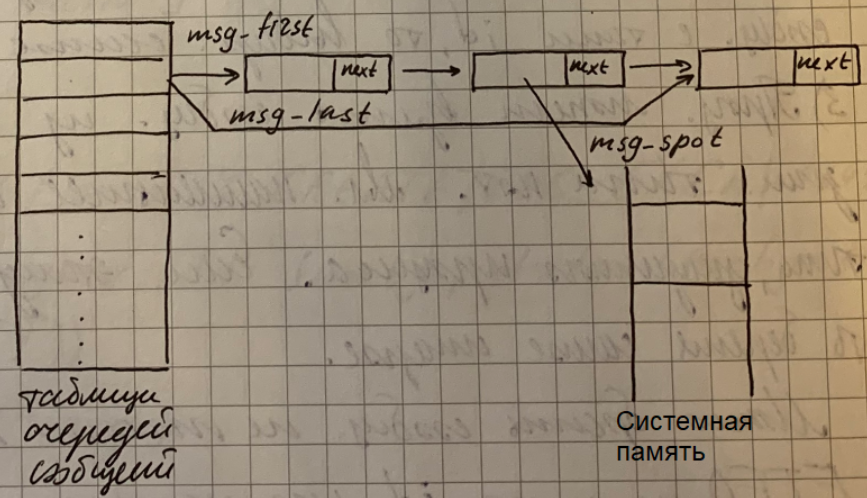
**ОС с монолит. ядром. Переключение в режим ядра. Система прерываний. Точные и неточные прерывания** Монолитное ядро – программа, состоящая из подпрограмм   
(имеющая модульную структуру), содержащих в  себе все функции ОС, включая планировщик, файловую систему, драйверы, менеджеры памяти. Поскольку это 1  программа, то она имеет 1 адресное пространство -> все её подпрограммы имеют доступ ко всем её внутренним  структурам.  Такие  ОС  делятся  на 2  части –  резидентную  и  нерезидентную.  Любые  изменения  приводят  к  необходимости перекомпилирования всей ОС. Пример: ОС UNIX, хотя она имеет минимизированное ядро.  Часть функций вынесены за пределы ядра. В Unix вынесены в shell. Следующие функции остаются в ядре:    
Управление процессами нижнего уровня, (диспетчеризация);Управление памятью, упр.  
физ. памятью.; Драйверы устройств, кот непосредственно взаимодействуют   
с клавиатурой. | Производительность: Выполнение  вызова  в  ОС  с  монолитным    
ядром  требует 2  переключений в режим ядра (приложение – ядро – приложение); Без  учёта  времени  передачи  самих  сообщений,  в  ОС  с  микроядром  системный  вызов  требует  как  минимум 4  переключений  плюс  время,  проводимое  в  блокировке  при  передаче сообщений (приложение – микроядро – сервер ОС –  микроядро – приложение).  
   
Переключение процесса в режим ядра: 1. Аппаратные  прерывания   
Прерывания  от  действия  операторов  (Ctrl+Alt+Del),  а  так  же  от  схем  компьютера  правильности  работы  +  контроль  уровня  напряжения  в  сети.  Аппаратные  прерывания:  сигналы  от  внешних  устройств поступают на контроллер прерывания, причем эти прерывания не зависят от выполняемого процесса,  т.е процесс вполне может переключиться на выполнение какого‐либо другого процесса. Аппаратные прерывания    
обрабатываются в системном контексте, при этом доступ в адресное пространство процесса им не нужен, т.е. им  не нужен доступ к контексту процесса. Обработчик   
прерывания не обращается к контексту процесса.   
2. Исключения. Являются синхронным событием, возникают в процессе выполнения программы, при арифметическом переполнении, делении на 0 и т.д. Бывают исправимые (процесс м. продолжаться с той же команды, в которой   
произошло исключение) и неисправимые (заканчиваются завершением программы).  
3. Системные вызовы Можно представить как программный интерфейс, предоставляемый ядром системы пользовательским процессам. Эти функции называются API функциями. При системных вызовах сначала вызывается библиотечная функция, кот. передает № сист. вызова в стек пользователя и вызывает спец. инструкцию системного прерывания, кот. меняет режим задачи на режим ядра и передает управлению обработчику системного вызова. Системные вызовы выполняются в режиме ядра. Имеют доступ к адресному пространству и управляющим структурам, вызвавшего их процесса, также они м. обращаться к стеку ядра этого процесса | Точные прерывания – прерывание, оставляющее машину в строго   
определенном состоянии. Обладает св‐ми: Счетчик команд указывает на команду  
 (текущая), до которой все команды выполнены; Ни одна команда после текущей не   
выполнена; Состояние текущей команды известно

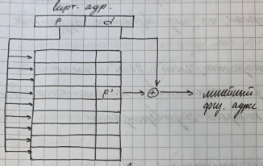
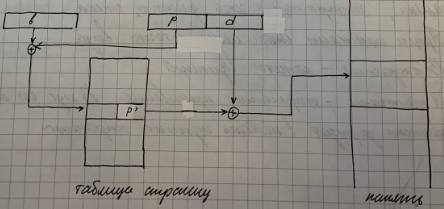
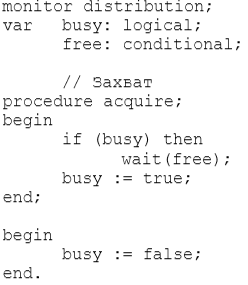
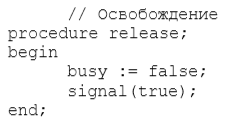
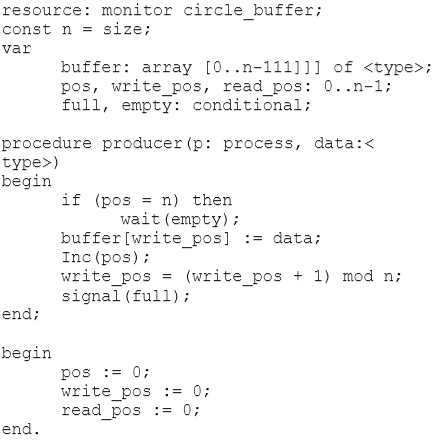
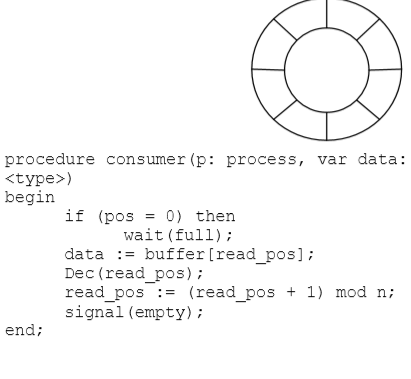
**Читатели-писатели; решение с использованием семафоров Дейкстра для ОС Unix**Существует набор процедур, обращающихся базе данных, чтобы получить оттуда информацию, и существуют процедуры, которые имеют право изменять содержимое базы данных. Критическим ресурсом является конкретное поле записи в базе данных. Поскольку процесс-писатель изменяет данные, он должен иметь монопольный доступ к БД. Очевидно, что монопольный доступ надо устанавливать на уровне конкретного поля структуры, а не всей БД. Каждый момент времени может работать только 1 писатель. Нет смысла ограничивать количество процессов-читателей, так как они не измен. содержимое БД. Когда число читателей = 0, писатель получает возможность начать работу. Новый читатель не м. начать работу, пока не завершится писатель. Когда читателю необходимо провести чтение, но вызывает start\_read(), когда завершить stop\_read(). Он м. читать, если нет писателя, который изменяет в данный момент данные, а также если нет писателя в очереди. Второе условие необходимо, чтобы предотвратить бесконечное откладывание писателей. signal(c\_read) заканчивается start\_read() , который пробуждает следующего читателя, ожидающего в очереди. Следующий читатель начинает работать и т.д. Возникает цепная реакция читателей, продолжающаяся, пока очереди есть неактивированные читатели. Так как они не мешают друг другу, то читатели могут выполняться реально параллельно, Такая цепная реакция обслуживается по в stop\_read() читателей принципу FIFO. количество уменьшается на 1 и может стать равным 0. В этот момент вырабатывается сигнал «можно писать». Процессы-писатели для начала работы вызывают start\_write(). Для обеспечения монопольного доступа к полю БД, если есть процесс-писатель или другой активный писатель, писатель переводится в состояние ожидания, пока с\_write не равно значению «истина». Получив возможность работать, писатель присваивает логической переменной wrt значение «истина», тем самым блокируя доступ других писателей к данному полю. Чтобы не возникло бесконечного откладывания читателей, писатель проверяет, нет ли ожидающих читатетей. Если они есть, то читатель из очереди активизируется. Если нет, то подается сигнал о возможности работы следующего писателя.  
#define WRITERS 3 #define READERS 5 #define P -1 #define V 1   
#define ACT\_READER 0 #define ACT\_WRITER 1 #define BIN\_ACT\_WRITER 2  
#define WAIT\_WRITER 3   
const int PERM = S\_IRWXU | S\_IRWXG | S\_IRWXO; struct sembuf start\_read[] = { { WAIT\_WRITER, 0, 1 }, { ACT\_WRITER, 0, 1 }, { ACT\_READER, V, 1 } }; struct sembuf stop\_read[] = { {ACT\_READER, P, 1} }; struct sembuf start\_write[] = { { WAIT\_WRITER, V, 1 }, { ACT\_READER, 0, 1 }, { BIN\_ACT\_WRITER, P, 1 }, { ACT\_WRITER, V, 1 }, { WAIT\_WRITER, P, 1 } }; struct sembuf stop\_write[] = { { ACT\_WRITER, P, 1 }, { BIN\_ACT\_WRITER, V, 1 }};void writer(int semid, int\* shm, int num){  
 while (1) { semop(semid, start\_write, 5); (\*shm)++; // критическая секция printf("process %d Writer #%d ----> %d\n", getpid(),num, \*shm); semop(semid, stop\_write, 2); sleep(2); }}  
void reader(int semid, int\* shm, int num) {   
while (1) { semop(semid, start\_read,3); printf("\tprocess %d Reader #%d <---- %d\n",getpid(), num, \*shm); semop(semid, stop\_read,1); sleep(1); }}  
int main() { int shm\_id;   
 if ((shm\_id = shmget(IPC\_PRIVATE, 4, IPC\_CREAT | PERM)) == -1) //выделяем разд. память { perror("Unable to create a shared area.\n"); exit( 1 ); }   
 int \*shm\_buf = (int\*)shmat(shm\_id, 0, 0); // получаем ее адрес   
if (shm\_buf == (void\*) -1) { perror("Can't attach memory"); exit( 1 ); } (\*shm\_buf) = 0; int sem\_id;  
 if ((sem\_id = semget(IPC\_PRIVATE, 4, IPC\_CREAT | PERM)) == -1) // 4 набора семафоров { perror("Unable to create a semaphore.\n"); exit( 1 ); }   
int ctrl = semctl(sem\_id, BIN\_ACT\_WRITER, SETVAL, 1); // устанавливает значения  
 if ( ctrl == -1) { perror( "Can't set semaphor`s values." ); exit( 1 ); }   
pid\_t pid = -1;   
for (int i = 0; i < WRITERS && pid != 0; i++) // инициализируем writers   
{ pid = fork(); if (pid == -1) { perror("Writer's fork error.\n"); exit( 1 ); } if (pid == 0) // если дочерний процесс, то запускаем функцию писателей { writer(sem\_id, shm\_buf, i); } }   
for (int i = 0; i < READERS && pid != 0; i++) { pid = fork(); if (pid ==  
 -1){ perror("Reader's fork error.\n"); exit( 1 ); } if (pid == 0) { reader(sem\_id, shm\_buf, i); } }  
 if (shmdt(shm\_buf) == -1) { perror( "Can't detach shared memory" ); exit( 1 ); } if (pid != 0) { int \*status; for (int i = 0; i < WRITERS + READERS; ++i) // ждем завершение всех читателей и писателей { wait(status); } if (shmctl(shm\_id, IPC\_RMID, NULL) == -1) { perror( "Can't free memory!" ); exit( 1 ); } }   
return 0; }  
  
  
  
  
 **Процессы: организация монопольного доступа – реализация взаимоисключения в помощью команды testand‐set, алгоритм Деккера.**     
Процессы: организация монопольного доступа – реализация взаимоисключения в помощью команды test and‐set, алгоритм Деккера.  | Аппаратная реализация  
 взаимоисключения (test\_and\_set).  Эта команда является машинной и неделимой, т.е.  
 ее  нельзя прервать. Она одновременно производит проверку и установку ячейки   
памяти, называемой ячейкой  блокировки: читает значение лог. переменной В,   
копирует его в А, а затем устанавливает для В значение True.  test\_and\_set(a, b):  a = b; b = true; В Windows это называется спин‐блокировкой.  
   
Flag = true, когда один из процессов – в своем критическом участке. main: Теоретически не исключено бесконечное откладывание, но вероятность -> 0 (т.к. test\_and\_set – машинная неделимая команда и выполняется очень быстро.    
Бесконечное откладывание – ситуация, когда разделённый ресурс снова захватывается тем же процессом. | Программная реализация (алгоритм Деккера).  Деккер   
предложил способ свободный от бесконечного откладывания. ( queue – очередь процесса входить в критическую секцию. )  
  
Недостаток обоих методов – активное ожидание на процессоре.  Активное  ожидание  –  ситуация ,  когда  процесс  занимает  процессорное  время ,  проверяя  значение    
флага.  Активное ожидание на процессоре является неэффективным использованием   
процессорного времени.  
  
  
**Unix: разделяемая память (shemget, shmat, семафоры(struct sem, semop, semget)).**   
Разделяемые сегменты – средство взаимодействия процессов через разделяемое адресное пространство. Т.к. адресное пространство защищено, процессы могут взаимодействовать только через ядро. Подключается к адресному пространству процесса (виртуальный). + быстрота передачи информации. Осуществляется мэппинг, но не копирование => повыш. быстродействие. Разделяемая память не имеет средств взаимоисключения. В ядре создается таблица разделяемых сегментов. |   
*int shmget*(key\_t key, size\_t size, int shmflg); Возвращает идентификатор общего сегмента памяти, связанного с ключом, значение которого задано аргументом key. Если сегмента, связанного с таким ключом, нет и в параметре shmflg имеется значение IPC\_CREATE или значение ключа задано IPC\_PRIVATE, создается новый сегмент. Значение ключа IPC\_PRIVATE гарантирует уникальность идентификации нового сегмента. | void \*shmat(int shmid, const void \*shmaddr, int shmflg); Присоединяет разделяемый сегмент памяти, определяемый идентификатором shmid к адресному пространству процесса. Если значение аргумента shmaddr равно нулю, то сегмент присоединяется по виртуальному адресу, выбираемому системой. Если значение аргумента shmaddr ненулевое, то оно задает виртуальный адрес, по которому сегмент присоединяется. | Семафор - неотрицательная защищённая переменная, над которой определено 2 неделимые операции: пропустить(P)(захватить) и освободить(V). Семафоры в системе поддерживаются таблицей семафоров в ядре (ядре ОС), одна на систему. В этой таблице отслеживаются все создаваемые в системе наборы семафоров. Unix поддерживают наборы считающих семафоров. Доступ к отдельному семафору – по индексу. Каждый дескриптор содержит следующие данные о наборе семафоров: 1) имя (целое число, присваивается процессом, который создал набор); 2) UID; 3) права доступа (для user, group, others); 4) кол-во семафоров в наборе; 5) последнее время обращения; 6) время последнего изменения управляющих параметров набора; 7) указатель на массив семафоров. На семафорах определены три операции (в отличие от семафоров Дейкстры): инкремент - освобождает ресурс; проверка освобождения ресурса без попытки захвата; декремент - захват ресурса; Флаги для sembuf: IPC\_NO\_WAIT – информирует ядро о нежелании процесса переходить в состояние ожидания; SEM\_UNDO – указывает ядру, что необходимо отслеживать изменения значения семафора операцией semop.   
struct sembuf { short sem\_num; short sem\_op; short sem\_flags; }  
int semget(key\_t key, int nsems, int semflg); Системный вызов возвращает идентификатор набора семафоров, связанный с аргументом key. Создается новый набор из семафоров nsems , если значение key равно IPC\_PRIVATE.  
int semop(int semid, struct sembuf \*sops, unsigned nsops); Функция одной неделимой операцией изменяет элементамы из набора семафоров semid. Каждый из элементов nsops(длина массива) в массиве sops определяет операцию, производимую над семафором в структуре struct sembuf.  
#include <sys/types.h> #include<sys/ipe.h> #include<sys/sem.h>  
struct sembuf sbuf[2] = {{0, -1, SEM\_UNDO|IPC\_NOWAIT}, {1, 0, 1}};  
int main(){  
int fd = semget(666, 2, IPC\_CREATE | S\_IRWXU | S\_IRWXG | S\_IRWXO);  
if (fd == -1){perror("semget error"); exit(1);}  
if (semop(fd, sbuf, 2) == -1) { perror("semop error"); }  
return 0; }  
  
  
  
**Управление процессорами: планирование и диспетчеризация, алгоритмы планирования – классификация;  приоритетное планирование, планирование в современных системах.** Процесс ‐ программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).   М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.  Аппаратный контекст процесса – сост. регистров.  Полный контекст процесса – сост. регистров + сост. памяти.  Планирование – управление распределением ресурсов ЦП между разл. конкурирующими процессами путем  передачи им управления согласно некот. стратегии планирования.  Диспетчеризация – выделение процессу процессорного времени.  Программа‐планировщик – программа, отвечающая за управление использованием совместного ресурса.   
  
Классификация алгоритмов планирования: с/без переключения, с/без вытеснения, c/без приоритетов; | Бесконечное откладывание – ситуация , когда процесс никогда   
не получает необх . для выполнения ресурсов  (точнее, кванта времени). Возникает, когда диспетчер всегда отдаёт квант др. процессу, т.к. его приоритет >.  | Алгоритмы планирования в системе разделения времени: 1) Round Robin (RR): циклическое планирование. очередь процессов (FIFO). Когда у текущего процесса истекает квант, он опять помещается в конец очереди.   
2) Адаптивное планирование (многоуровневые очереди): несколько очередей с приоритетами. Новые и “проснувшиеся” процессы попадают в первую очередь. Если они не успевают завершиться или заблокироваться на вводе/выводе, то попадают в следующую очередь и т.д. Последняя очередь работает по схеме RR.  
  
 3) Адаптивное планирование (по памяти): ​ дополнительно оценивается объем занимаемой в данный момент процессом физической памяти и необходимый объем для дальнейшего выполнения. В случае, если процесс может получить этот объем. то он выполняется. Если нет – откладывается; 4) Windows: т.к. в ОС процессы неравнозначны, система разделения времени пытается выделить процессорное время справедливо: учитывается время ожидания, простоя, приоритет блокировки. Процесс требующий много процессорного времени окажется в самой низкой очереди. Интерактивные (мышь, клавиатура) будут в высшей очереди.  
 | Алгоритмы планирования в системе пакетной обработки: 1) FIFO - очередь. Без переключения. Без приоритетов; 2) SJF (shortest job first) - может возникнуть бесконечное откладывание. Необходима априорная информация о времени выполнения. Статический приоритет. Без переключения; 3) SRT (shortest remaining time) - с вытеснением. Выполняющийся процесс прерывается, если в очереди появляется процесс с меньшим временем   
выполнения (меньше оставшегося до завершения времени); 4) HRV (Highest   
Response Ratio) - наиболее высокое относительное время ответа –  
c динамическими приоритетами. Используется в UNIX. Чем больше ожидание, тем выше приоритет. Позволяет избежать бесконечного откладывания. (tw - время ожидания, ts - запрошенное время обслуживания) priority = (tw-ts)/ts

**Защищ. режим работы ПК с процессорами Intel (486, .). Уровни привилегий. Сист. таблицы: GDT, LDT, IDT.**  Защищенный режим – 32х‐разрядный, поддерж.    
многопоточность  и  многопроцессность.  В  отличие от реального режима здесь доступно 4  Гб памяти (в реальном диапазон адресов памяти  ограничен  1  мб ).  В  защищ .  реж .  4  уровня  привилег. Ядро ОС находится на 0‐м уровне. | GDT (global descriptor   
table) – таблица , которая  описывает  сегменты  основной  памяти  ОС.  В  системе только 1 GDT. На начальный адрес GDT  указывает GDT Register (32 разрядный).  | IDT  (interrupt  descriptor  table)  –  таблица ,  предназначенная  для  хранения  адресов  обработчиков  прерываний.  Базовый  адрес IDT  помещен  IDT  Register.  IDT  столько ,сколько  процессоров. |  LDT  (local  descriptor  table)  –  таблица ,  которая  описывает   
адресное пространство процесса. В LDT  Register находится смещение до соответствующего  дескриптора  в  GDT,  описывающего  сегмент,  в  котором находится LDT.   Таблиц LDT столько, сколько процессов.  | Формат селектора (явл. ID сегмента):    
Индекс  равен  (кратен)  8  и  является  смещением в таблице дескрипторов.   Обязательно наличие 0‐го дескриптора.  0  и 1  биты –  Requested  Privilege  Level,  показывает    
на  каком  уровне  привилегии  работаем (00 – нулевой уровень). 2 бит –  Table indicator, 0 – адрес в GDT, 1 – в LDT   Селектор указывает на дескриптор сегмента  в таблице дескриптора.  | Формат дескриптора сегмента (GDT):  A – access – бит доступа  
 к сегменту;  Тип – опред. поля доступа; DPL – уровень привилегий;  P – бит  
 присутствия, исп. для работы с ВП;  D – бит разрядн. операндов и адр. (0 – 16, 1 – 32);  G – бит  гранулярности (0 – размер  сегмента  задан в байтах, 1 – в страницах по 4 Кб).  Процесс не м. выйти за размер своего сегмента –  контроль  ОС  (защита  адресных    
простр‐в  сегментов  др.  от  др .).  К  дескр .  GDT  и LDT  обращаемся с помощью   
селекторов, к дескр.  IDT  обращаемся  по  смещению,  кот .  берем  из  прерывания.  Обработчик  прерывания:  IDTR  (указывает  на  начало  IDT)  +  смещение из прерывания = дескриптор в IDT. Из дескр. в IDT берем селектор. С помощью селектора узнаем, с какая табл.  | 1) Если работаем с GDT, то с помощью селектора получаем дескриптор   
сегмента, в котором находится наш обработчик, в  этом сегменте с помощью смещения из дескриптора в IDT мы получаем точку входа в обработчик прерывания; 2) Если   
работаем с LDT, то с помощью LDTR (в котором у нас смещение до дескриптора сегмента в GDT, в котором находится  LDT) находим этот дескриптор, получаем сегмент. В этом сегменте находится нужная LDT, в ней с помощью селектора  получаем  дескриптор  сегмента  в  котором  находится  наш  обработчик ,  в  этом  сегменте  с  помощью  смещения  из  дескриптора в IDT получаем точку входа в обработчик прерывания.  |  
В процессоре каждому из сегментных регистров (CS, DS, SS, ES, FS, GS) сопоставлен теневой регистр (дескриптора). Он не  доступен программисту и загружается   
параллельно автоматически из таблицы дескрипторов соотв. сегмента чтобы реже   
обращ. к ОП.  | Уровни привилегий (кольца защиты): В защищенном   
режиме предусмотрен механизм защиты с помощью системы уровней  привилегий. Существует 4 уровня привилегий: от 0 до 3. 0‐ой уровень – предоставляет макс.   
привилегии, используется для  ядра OC. 3‐ий уровень – исп. для прикладных   
программ. Каждому сегменту программы придется опред. уровень привилег.,  указываемый в поле DPL (Descriptor Privilage Level). Уровень привилег., указанный в    
дескрипторе,  назначается  всем  обьектам,  входящим  в  данный  сегмент.  Уровень    
привилегий  выполняемого  в  данный  момент сегмента команд называется текущим уровнем привилегий – CPL(Current Privilage Level).   Вся  система  привилегий    
основана  на  сравнении  CPL  выполняемой  программы с уровнями привилегий DPL   
сегментов, к которым она обращается. В реальном режиме уровней привилегий нет.

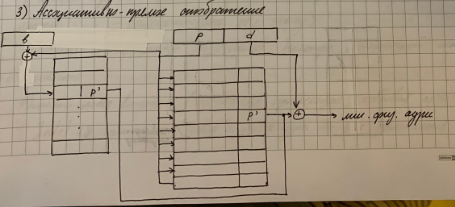


**Подсистема ввода ‐ вывода: синхронный и асинхронный ввод‐вывод.** Задача системы вв/выв – обеспечить взаимодействие процессора с внешними устройствами. Синхронный  – приложение, запросившее операцию вв/выв.   
блокируется до завершения этой операции. 1 – обработка системного вызова; 2 – процедура вв/выв; 3 – завершение операции вв/выв   
  
Асинхронный – приложение, подавшее запрос на вв/выв, может какое‐то время продолжать выполнение. (\* – синхронизация (мьютекс, событие,   
порт завершения‐  completion port).)   
При вв/выв используется буферизация. Одинарный буфер – выделяется в пространстве ядра ОС. Перед вв/выв процессу назначается буфер.  Сначала данные помещаются в буфер, затем  копируются в адресное пространство процесса.  Операция  вв/выв  выполняется  быстрее, т .к.  процесс не блокиров. – все данные получ. сразу).   
Двойная буферизация: Опережающее считывание.  Исключаются проблемы, т.к. буфер находится в системной (невыгружаемой) области памяти.  Схемы передачи информации: блок‐ориентированная (поточно‐ориент.), побайтно‐ориент. (построчно).  Если процесс пытается поместить в буфер строку, а он занят, то процесс будет блокирован (пр‐во/потребл.)  Используется двойная буферизация. Проблема буферизации остро встает в ОС реальн. вр.  
   
2 способа перемещения:  1. Копирование – передача последов. байт   в обл. памяти.    
2. Мэпинг – получение указателя на адресное пространство в ядре ОС.  | Порт завершения – механизм сообщения по токам о завершении операции вв/выв. Если файл сопоставить с  портом завершения, то по окончании операции вв/выв в очередь порта завершения став. пакет завершения.  Приложение проверяет наличие пакета завершения в порте завершения и т.о. узнает о факте завершения. | Базовым понятием в UNIX является процесс. Процесс  ‐ программа в стадии выполнения, единица  диспетчеризации. В UNIX принято рассматривать процесс как виртуальную машину с собственным адресным  пространством. Эта виртуальная машина выполняет программу, предоставляя набор услуг.  Средства взаимодействия процессов: программные каналы (именованные и неименованные), сигналы,  разделяемая память, очереди сообщений, семафоры, ввод/вывод с отображением в память  Программный канал – это специальный буфер, который создается в системной области памяти.  Они  описываются в соответствующей системной таблице.  Информация в канал записывается по принципу FIFO и не модифицируется. В канал нельзя писать, если его  читают и наоборот. 2 типа программных каналов: именованные, неименованные. 

**Сигналы. Разделяемые сегменты. Сообщения.** 1) Именованные каналы создаются mknod name p; 2) pipe()  создает неименованный   
программный канал. (есть дескриптор, но нет id)  Предок и потомок могут обмениваться сообщениями с помощью неименованного программного канала.  1 процесс пишет, другой считывает из трубы – симплексная связь.  Канал является средством передачи информации, который не имеет собственных средств синхронизации.  Программный канал буферизуется на 3х уровнях.Если процесс пытается записать более 4 Кб данных,  
то труба буфирезуется во времени, приостанавливая  «писателя», пока все данные не будут прочитаны.  | Техника сигналов поддерживается ОС и отвечает стандартным  
требованиям системы прерываний, к которой  относятся обработка системных исключений, снешних и внутренних прер‐й, а также маскируемых и  немаскируемых прер‐й. В UNIX процессы могут порождать и принимать сигналы. Сигналы:  синхронные(порождены процессом) и асинхронные(внешним действием: польз или ядром).signal() Для изменения хода выполнения программы. Необходимо написать свой обработчик (в зависимости от  того был получен сигнал или нет выполняются разные действия) | Разделяемые сегменты – средство взаимодействия процессов через   
разделяемое адресное пространство. Т.к.  адресное пространство защищено, процессы могут взаимодействовать только через ядро. Подключается к  адресному пространству процесса (виртуальный).  + быстрота передачи информации. Осуществляется мэппинг, но не копирование => повыш. быстродействие.  Разделяемая память не имеет средств   
взаимоисключения.  В ядре создается таблица разделяемых сегментов.  int shmget(key\_t key, int size, int shmflg); Создается новый разделяемый сегмент памяти с размером size (округленным до размера, кратного PAGE\_SIZE), если значение key равно IPC\_PRIVATE или если значение key не равно IPC\_PRIVATE и нет идентификатора, соответствующего key. | Шаблон сообщения описывает Struct msgbuf {long mytype; char mytext[MSG\_MAX];} Ядро только выполняет размещение и выборку сообщений. Менеджер ресурсов отслеживает число очередей. При посылке сообщения производится копирование текста сообщение в адресное пространство ядра системы. При получении – обратная операция. Недостаток: двойное копирование. Система поддерживает системную таблицу очередей сообщений. Когда процесс передает сообщение в очередь, ядро создает для него новую запись и помещает её в конец связного списка записей соответствующих сообщением указанной очереди. В каждой такой записи указывается: тип сообщения, размер в байтах, указатель на область данных ядра системы, в которой фактически находится сообщение. Когда какой либо процесс выбирает сообщение из очереди, он может выбрать: самое старое сообщение независимо от его типа; по указанному id; взять сообщение, числовое значение типа которого является наименьшим

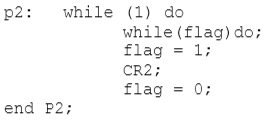
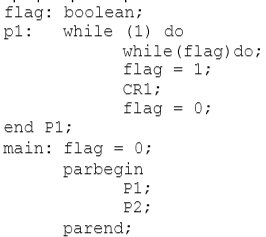
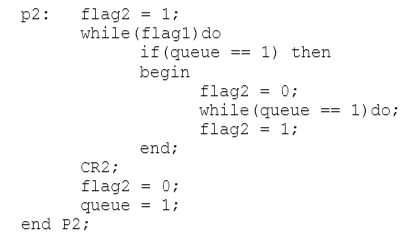
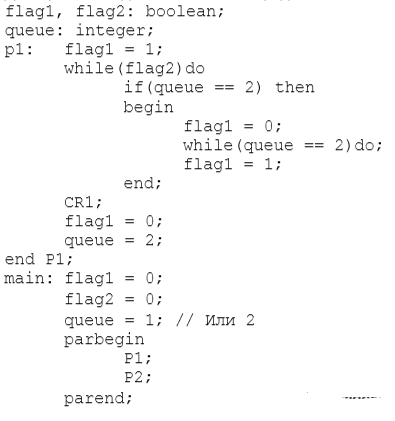
**Средства взаимодействия процессов: мониторы – простой монитор, монитор "кольцевой буфер"**  Использование семафоров часто приводит к взаимоблокировке.  
 Понятие «монитор» предложил Хоар. Монитор – языковая конструкция, состоящая  
 из структур данных и подпрограмм, использующих данные  структуры.  Монитор    
защищает  данные.  Доступ  к  данным монитора  могут  получить  только  п/п    
монитора. В  каждый момент времени в мониторе м. находиться только 1 процесс. Монитор является ресурсом.  Процесс, захвативший монитор, – процесс в мониторе, процесс, ожидающий в очереди, – процесс на мониторе.    Используется  переменная  типа  «событие»  для  каждой  причины  перевода  процесса  в  состояние  блокировки. Над переменной – 2 операции: wait – открывает доступ к монитору, задержив. выполнение   
процесса. Оно д.б. восстановлено операцией; signal другого процесса. Чтобы   
 гарантировать,  что  процесс ,  обратившийся  к  ресурсу ,  когда‐нибудь  его  получит,  
  необходимо ,  чтобы  процесс, находящийся в ожидании имел больший приоритет, чем вновь пытающийся обратиться в монитор.  Мониторы м. входить в состав языка программирования (например, C#).  | Простой монитор – предназначен для разделения  
1 единственного ресурса произвольному числу процессов .  Монитор, обслуживающий произвольное количество процессов, ограничен длиной своей очереди.   
   
Монитор  «кольцевой  буфер»  –  решает  задачу «производство‐потребление»,  то  есть  существуют  процессыпроизводители  и  процессы‐потребители,  а  также  буфер –  массив  заданного  размера,  куда  производители  помещают данные, а потребители считывают оттуда данные в том порядке, в котором они помещались (FIFO –  “первым вошел, первым вышел”).  
 

**Защищенный режим: перевод компьютера из реального режима в защищенный и обратно.** Реальный режим  (или режим реальных адресов) ‐ это название было дано прежнему способу адресации  памяти после появления 286‐го процессора,   
поддерживающего защищённый режим.  Реальный режим поддерживается аппаратно. Работает идентично 8086 (16 разрядов , 20‐разрядный адрес –  сегмент/смещение).   
Минимальная адресная единица памяти – байт.   2^20 = FFFFF = 1024 Кб = 1Мб (объем доступного адресного пространства).  Необходим  для  обеспечения  функционирования  программ,  разработанных  для  старых  моделей,  в  новых  моделях микропроцессоров. Компьютер начинает работать в реальном режиме. | Защищенный режим – 32х‐разрядный, поддерж. многопоточность и многопроцессность. В отличие от  реального режима здесь доступно 4 Гб памяти (в реальном диапазон адресов памяти ограничен 1 мб). В защищ.  реж. 4 уровня привилег. Ядро ОС находится на 0‐м уровне.  | Уровни привилегий (кольца защиты): В защищенном режиме предусмотрен механизм защиты с помощью  системы уровней привилегий. Существует 4 уровня привилегий: от 0 до 3. 0‐ой уровень – предоставляет макс.  привилегии,  используется  для  ядра  OC.   
3‐ий  уровень  –  исп .  для  прикладных  программ.  Каждому  сегменту  программы  
придется опред. уровень привилег., указываемый в поле DPL (Descriptor Privilage Level, уровень  привилег. дескр.) его дескр. Уровень привилег., указанный в дескрипторе, назначается всем обьектам, входящим  в данный сегмент. Уровень привилегий выполняемого в данный момент сегмента команд называется текущим  уровнем  привилегий  –  CPL(Current  Privilage  Level).  Он  определяется  полем  RPL  селектора  сегмента  команд,  загружаемого в CS. Вся система привилегий основана на сравнении CPL выполняемой программы с уровнями  привилегий DPL сегментов, к которым она обращается. В реальном режиме уровней привилегий нет.  | Переход в защищ. режим: Чтобы перейти в защищенный режим, достаточно установить бит РЕ — нулевой бит в управляющем регистре CR0,  и процессор немедленно окажется в защищенном режиме. Единственное дополнительное требование, которое  предъявляет Intel,  чтобы в этот момент все прерывания, включая немаскируемое, были отключены.   Инициализируем  GDT,  IDT  загружаем  с  помощью  привилегированных  команд  в  регистры ,  перенастраиваем  контроллер прерываний на новый базовый вектор 20h, запрещаем прерывания устанавливаем бит РЕ в 1.    
Переключение в защищенный режим:  1. Открыть адресную линию A20.  2. Подготовить в оперативной памяти глобальную таблицу дескрипторов GDT. В этой таблице должны быть  созданы дескрипторы для всех сегментов, которые будут нужны программе сразу после того , как она  переключится в защищённый режим. Впоследствии, находясь в защищённом режиме, программа может  модифицировать GDT (если, она в нулевом кольце защиты).  3. Подготовить в оперативной памяти таблицу   
дескрипторов прерываний IDT.  4. Для обеспечения возможности возврата из   
защищённого режима в реальный необходимо записать адрес  возврата в   
реальный режим в область данных BIOS по определенному адресу, а также записать в CMOSпамять  в  ячейку  0Fh  код 5.  Этот  код  обеспечит  после  выполнения  сброса  процессора  передачу  управления по адресу, подготовленному нами в области данных BIOS по этому адресу.  5. Запомнитьв оперативной памяти содержимое  
 сегментных регистров, которые необходимо сохранить  для возврата в реальный режим, в частности, указатель стека реального режима. 6.Запретить  все  маскируемые  и  немаскируемые  прерывания.  Сохранить  маски  прерываний .  Перепрограммировать контроллер прерываний.  7. Загрузить регистр GDTR и IDTR.  8. Перейти в защищенный  
 режим (установить бит РЕ — нулевой бит в управляющем регистре CR0 в 1).  9. Загрузить новый селектор в регистр CS.  10. Загрузить сегментные   
регистры селекторами на соответствующие дескрипторы.  11. Разрешить прерывания. |  
Возвращение в реальный режим:  1. Переключиться в реальный режим    
2. Сбросить очередь предвыборки, загрузить CS реальным сегментным адресом    
3. Задать регистры для работы в реальном режиме    
4. Загрузка IDTR (Interrupt Descriptor Table Register)   
для реального режима    
5. Разрешаем немаскируемые прерывания    
6. Разрешить маскируемые прерывания   
  
  
  
**Виртуальная память: распределение памяти страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество.**  
Virtual Memory - система при которой рабочее пространство процесса частично располагается в основной памяти и частично во вторичной. При обращении к какой либо памяти, система аппаратными средствами определяет, присутствует ли область физической памяти, если отсутствует, то генерируется прерывание, это позволяет супервизору передать необходимые данные из вторичной в основную. Используется адресное пространство диска как область свопинга или педжинга, т.е. для временного хранения областей памяти. Виртуальная память - память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или пейджинга, т.е. для временного хранения областей памяти. | Подходы к реализации управления виртуальной памятью: 1. страничное распределение памяти по запросам; 2. сегментное распределение памяти по запросам; 3. сегментно - страничное распределение памяти по запросам   
Страница - является единицей физического деления памяти. Её размер устанавливается системой. Сегмент - является единицей логического деления памяти. Её размер определяется объемом кода. сегментное распределение памяти по запросам | Распределение памяти страницами по запросам: Адресное пространство процесса и адресное пространство физической памяти делится на блоки равного размера. Блоки, на которые делится адресное пространство процесса называют страницами, а блоки на которые делится физическая память кадрами, фреймами или блоками. Процесс копируется в страничный файл в области свопинга, таким образом, для него создается виртуальное адресное пространство. Соответственно размер виртуального адресного пространства может превышать объем физической памяти. Для возможности отображения страниц на соответствующие блоки физической памяти необходимо аппаратно поддерживаемое преобразование адресов (иначе слишком долго).  
+ легко реализовать, алгоритм LRU в этом эффект; -сложность коллективного использования. | Свойство локальности-использовании ассоциативного, буфера на 8 адресов при страничном преобразовании дает 90% скорости полностью ассоциативной памяти благодаря свойству локальности. Берется из того, что в наших программах чаще встречается следование, чем ветвление. Локальность бывает двух типов: временнАя(процесс обратившийся к одной странице наиболее вероятно в следующую единицу времени обратится к этой-же странице) и пространственная (процесс обратившийся к одной странице наиболее вероятно обратится к соседним страницам). Рабочее множество - набор страниц которые процесс должен держать в памяти. Если это набор не будет загружен возникнет трешинг страниц (постоянная загрузка и выгрзка одних и тех же страниц)  
(1 – прямое отображение; 2 – ассоциативное отображение; 3 – ассоц.-прям. отображ.)



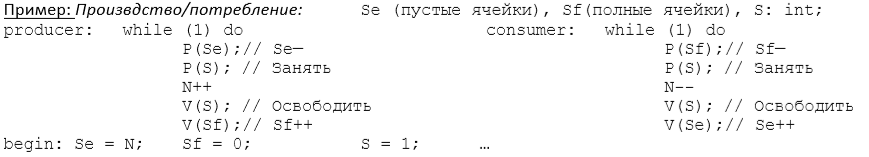
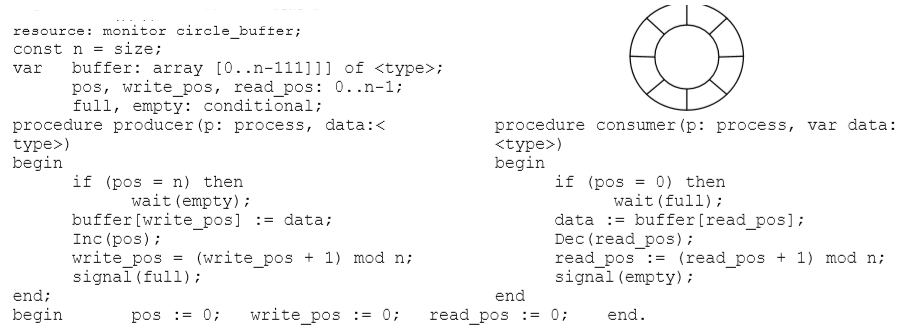
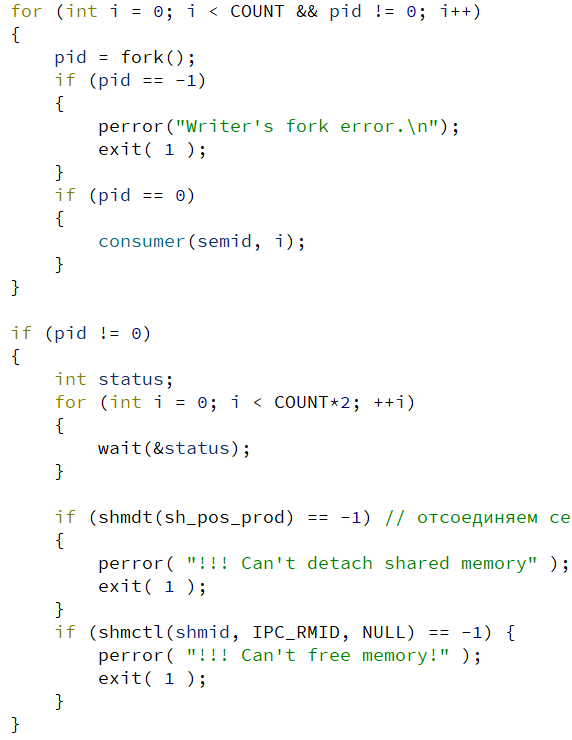
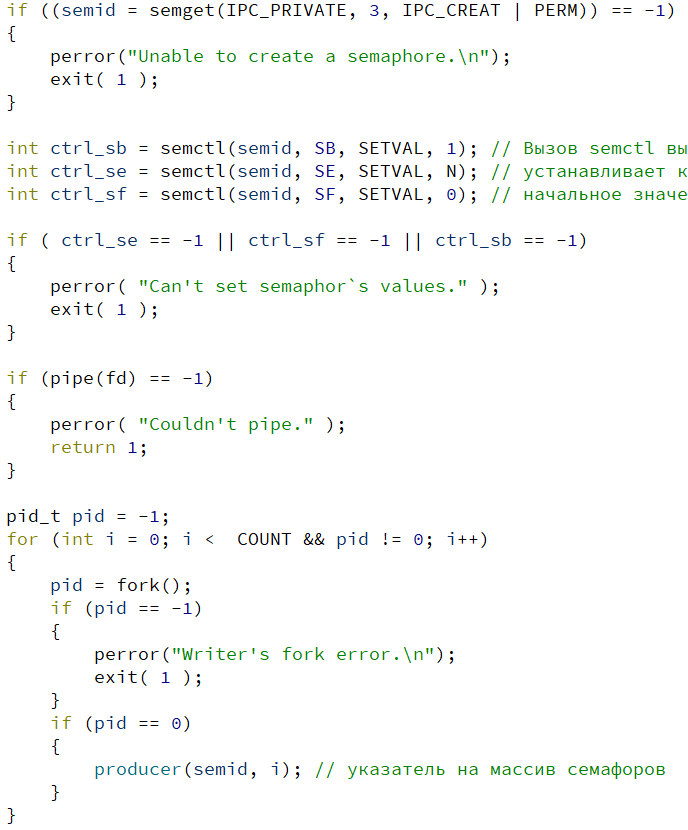
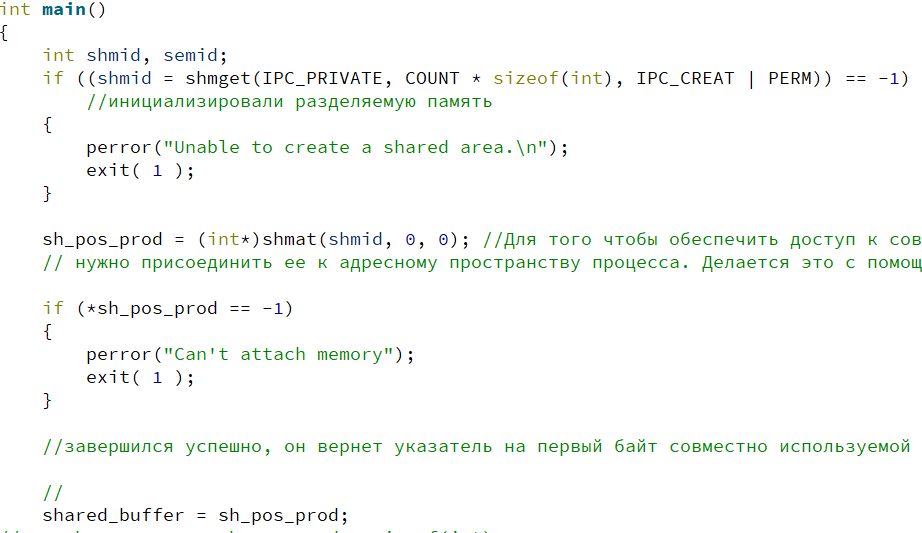
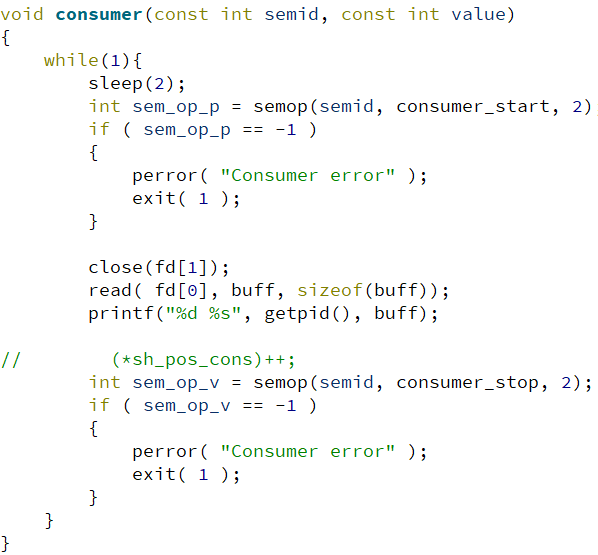
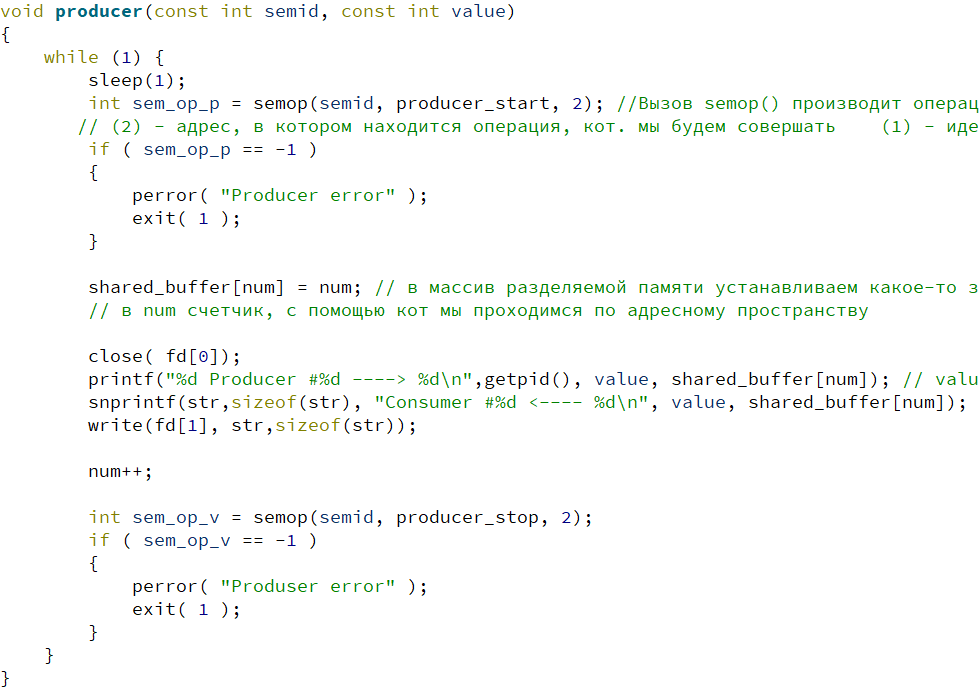
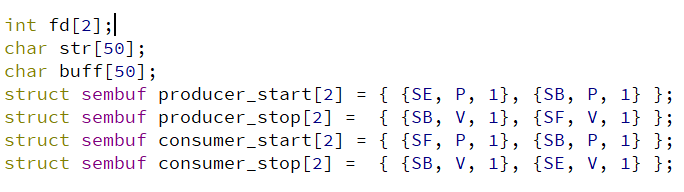
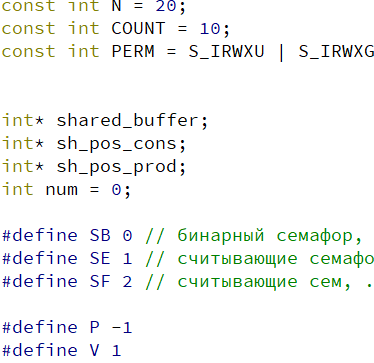
**Прерывание от системного таймера в защищенном режиме: функции** UNIX: Прерывание таймера имеет второй приоритет (после прерывания по сбою питания).    
Функции обработчика прерывания таймера:  Инкремент счетчика таймера: Вызов процедуры обновления статистики использования процессора текущим   
процессом; Пробуждение в нужные моменты времени системных процессов   
(например, swapper и pagedaemon); Поддержка профилирования выполнения  
 процессов в режимах ядра и задачи при помощи драйвера  параметров; Вызов обработчиков отложенных вызовов.; Декремент значения кванта процессорного   
времени.; После истечения кванта процессорного времени:  1) Посылка текущему  
 процессу сигнала SIGXCPU, если тот превысил выделенный ему квант  процессорного времени;  2)Вызов функций планировщика (пересчет приоритетов).Т.к. некоторые из задач не требуют выполнения на каждом тике , то вводится понятие основного тика  (равен n тикам), часть задач выполняется только при основном тике.   
Windows: В  многопроцессорной  системе  каждый  процессор  получает    
прерывания  системного  таймера,  но  обновление  значения  системного  таймера  в    
результате  обработки  этого  прерывания  осуществляется  только  одним процессором.  Однако  все  процессоры  используют  это  прерывание  для  измерения  кванта  времени,  выделенного потоку, и для вызова процедуры планирования по истечении этого кванта. Функции обработчика прерывания таймера:  Инкремент счетчика таймер; Вызов процедуры сбора статистики использования процессорного времени; Вызов обработчиков отложенных вызовов; Декремент значения кванта процессорного   
времени; После истечения кванта процессорного времени:  
Посылка текущему потоку сигнала по превышении кванта процессорного времени;   
Вызов функций, относящихся к работе диспетчера ядра (пересчет приоритетов);Постановка DPC (Deferred Procedure Call – отложенный вызов процедуры) в очередь, чтобы  инициировать диспетчеризацию потоков 

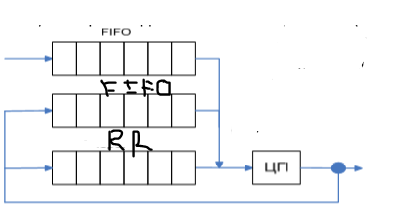
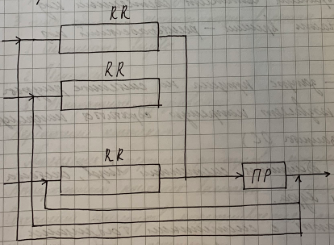
**Процессы: синхронизация процессов и алгоритмы взаимного исключения в распределенных системах.**  Процесс ‐ программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).   М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.    В  распределенных  системах  процессы  не  имеют  общей  памяти  (например,  сети ).  Могут  синхронизироваться только сообщениями.  Взаимодействие  по  системе «клиент-cервер».   
Выдел . процессы серверы, предоставляющие набор сервисов, и процессы‐клиенты, пользующиеся предоставленными сервисами.  send <список значений> to <адресат>,   где <список значений> – значения набора парам. в момент посылки  receive <список переменных> from <адресат>  <адресат> – указывается адресат сообщения и отправитель    Считается ,  что  модель «клиент‐сервер»  работает  на  уровне  транзакций   
(условно  неделимая  последовательность действий – перерывы не приводят к изменению данных, обеспеч. целостность данных).  | Простейший протокол обмена «клиент‐сервер»: 1. Запрос – клиент запраш. сервер для обраб. запр; 2. Ответ – сервер   
возвращает результат операции; 3. Подтверждение  –  клиент  подтверждает    
прием  сообщения от сервера; 4. Клиент запрашивает: «Сервер доступен?»;   
5. Если сервер доступен, то посылает: «Я доступен»; 6. Если сервер   
недоступен (занят), то может послать  сообщение: «Перезвоните»; 7. Адрес неверен.  
 Процесс с зад. ID в сист. отсутств. RPC (remote procedure call) –   
вызов удаленной процедуры.  | Проблема синхронизации: Процессам часто нужно  
 взаимодействовать друг с другом, например, 1 процесс может передавать данные др.  процессу, или несколько процессов могут обрабатывать данные из общего файла. В этих сл. возникает проблема  синхронизации процессов. Она связана с потерей доступа к параметрам из‐за их некорректного разделения.  Критический ресурс ‐ разделенная  
 переменная, к которой обращаются разные процессы.  Критическая секция ‐ строки кода, в кот. происходит обращение к критическому ресурсу.  Необходимо обеспечить монопольный доступ процесса к критическому ресурсу до тех пор пока процесс его не  освободит. Т.е. чтобы не могли одновременно войти в крит. секцию | Алгоритм синхронизации логических часов (алг. Лампорта): Для  2‐х  произв.    
событий  вводятся  понятия  «случилось  до»,  «случилось  после».  Время приема НЕ м.б. < времени посылки сообщения.  1. Каждому сообщ. припис. время отправки   
по локальным часам отправителя.  2. Получатель сравнивает это время со своим   
временем. Если собств. время <  полученного, то собств. время устанавливается на > времени полученного.  | Алгоритмы взаимоисключения в распределенных системах:   
1) Централизованный алгоритм . Процесс‐координатор (м.б. выбран процесс с наиб.   
сетевым адресом)  отвечает  за  возможность  вхождения  в  критические  секции.  Как  только  процесс  готов  войти  в  критич.  область ,  он  посылает  сообщение,  где  указывает ID  крит .  секции .Процесс  сможет  зайти  в  крит.  секцию  только  после  ответа  координатора. Координатор проверяет, не находится ли к‐л. др. процесс в крит.   
области (по полученным сообщениям).  Если находится, то корд. не посылает ответ, а ставит запрос в очередь. После освобождения крит. области процессу б.  выслан ответ‐подтверждение. Если координатор аварийно завершился  необходимо выбрать новый координатор. Если кл. процесс обнаружит отсутств. корд . (время ожид ответа > кртитич . времени), то процесс инициирует выбор нового  координатора. Самый эффект. алг  
2) Распределенный  алгоритм.  Процесс ,  желающий  войти  в  критический  участок,   
формирует  запрос ,  содержащий  ID  критического  участка,  свой  номер  и  время  полокальным  часам.  Запрос  посылается  всем  процессам  в  системе.  Полагается, что передача сообщений является надежной. Когда др. процесс получ. сообщ., то:  a) Если получат. не находится и не собирается входить в критический участок, то он посыл. ответ‐разреш.  b.) Если получатель уже находится в критическом участке, то он   
не отвечает и ставит запрос в очередь.  c) Если получатель желает войти в   
критическую секцию, то он сравнивает свое время и время в запросе. Если время  в запросе меньше его собственного времени (собств. – позже) – (a), иначе (b).  Процесс входит в критическую обл., если получ. n‐1 разреш. (от всех, кроме себя). После выхода  
посылает разрешение  всем процессам в очереди. Необходима посылка и получение n‐1 сообщений.   
3)Алгоритм Token Ring. Самый распространенный стандарт локальных сетей. Все процессы образуют логическое кольцо  (направленное). Кажд. процесс знает № своей позиции и № ближайшего к нему процесса. При инициализ. Token (спец.  сообщение) к 0, передается от процесса n‐1 к n. Когда процесс получает Token, он смотрит,  
 не требуется ли ему войти в  критическую секцию. Если надо – входит (удерживая Token), если нет – посылает Token дальше. Если желающих нет,  Token циркулирует по кольцу с большой скоростью. Кол‐во сообщений – от 1 до ∞ (если ни 1 не входил).   
4)Модифицированный Token Ring. Если процесс заинтересован в передаче данных, то при  получении Tokena, Token изымается из кольца. Процесс посылает в кольцо свой Token, содержащий адрес источника и  адрес получателя. Передача Tokena = копирование. При обнаружении в Tokene своего адреса, процесс копир. Token в  свой буфер и вставл . в него свое подтверждение приема. Процесс , пославший Token и получивший подтверждение  изымает Token и посылает др. Token, чтобы процессы могли обрабатывать свои данные

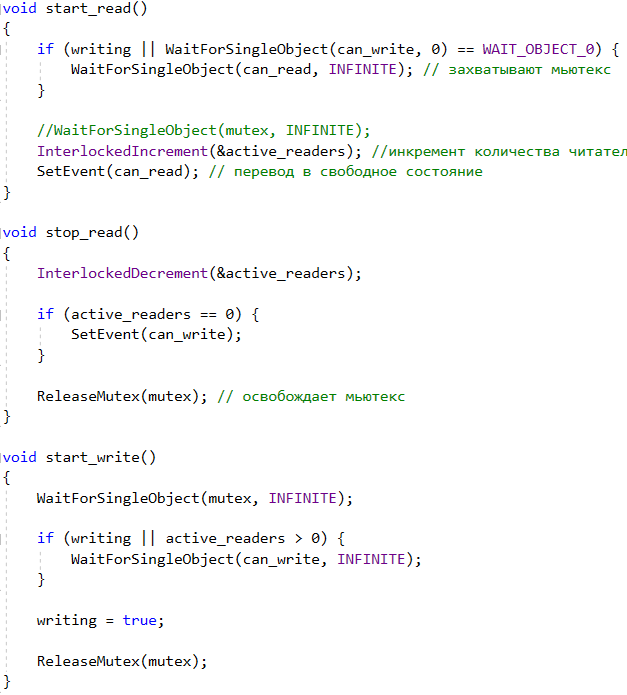
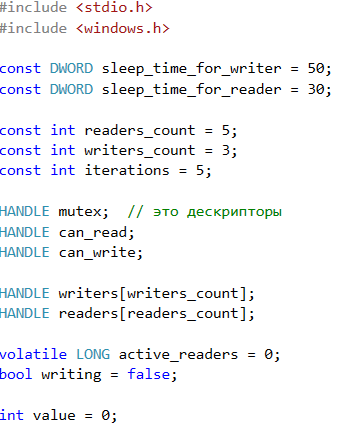
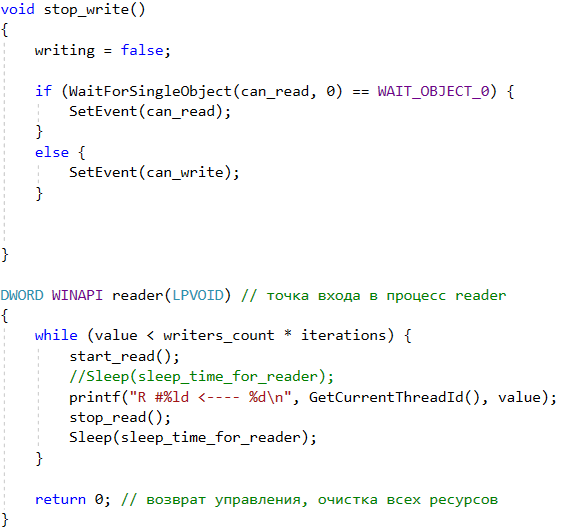
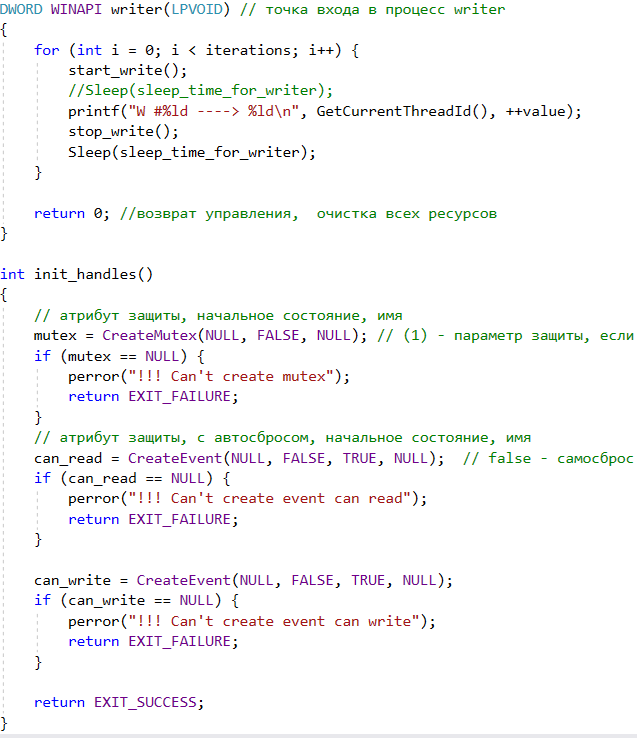
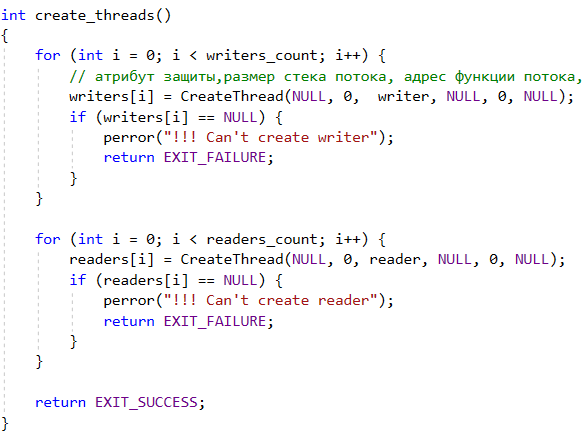
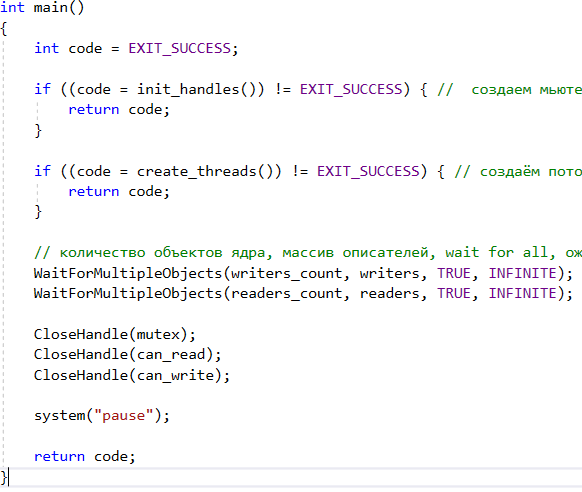
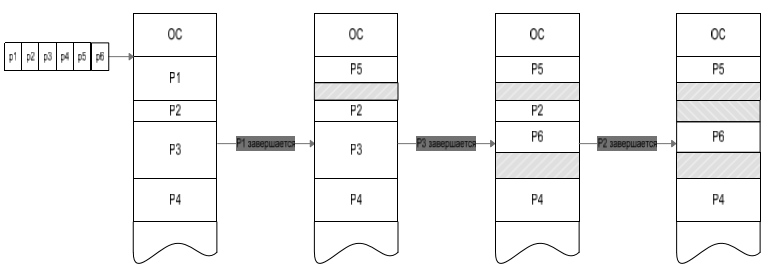
**Взаимодействие процессов: монопольное использование – программная реализация взаимоисключения,  взаимоисключение с помощью семафоров; сравнение – достоинства и недостатки** Процессам часто нужно взаимодействовать друг с другом , например, один процесс может передавать данные  другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает  проблема синхронизации процессов. Она связана с потерей доступа к параметрам из‐за их некорректного   
разделения. В  каждый момент времени на процессоре выполняется 1 процесс. Каждому процессу выделяется квант процесс. времени.  Разделяемый ресурс ‐ переменная, к которой обращаются разные процессы.  Критическая секция – область кода, из которой осуществляется доступ к разделяемому ресурсу.  Монопольное использование – если процесс получил доступ к разделяемому ресурсу, то др. процесс не м. получить доступ.  Необходимо обеспечить монопольный доступ процесса к разделяемому ресурсу до тех пор, пока процесс его не освободит.  Все алгоритмы программной реализации обобщил Дейкстра, введя понятие семафора.  Активное ожидание на процессоре – ситуация , когда процесс занимает процессорное время , проверяя значение флага  (занятости  ресурса  другим  процессом).  Активное  ожидание  на  процессоре  является  неэффективным  использованием  процессорного времени. (пример реализации и программная   
реализация алгоритма Деккера (способ свободный от бесконечного откладывания.) НИЖЕ!!!) | Семафор – неотрицательная защищённая переменная S, над которой  
 определено 2 неделимые операции:  Р (от датск. passeren ‐ пропустить) и V (от датск. vrygeven ‐ освободить). Защищенность семафора означает, что значение  семафора м. изменяться только операциями P и V.  1. Операция P(S): S = S – 1. Декремент   
семафора (если он возможен). Если S = 0, то процесс, пытающийся выполнить  операцию Р, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой  процесс, выполняющий операцию V(S).    
2.Операция V(S): S = S + 1. Инкремент S одним неделимым действием (последовательность непрерывных действий:  инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0,  то V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожид. на семафоре.   
  
  
1)  
  
2)

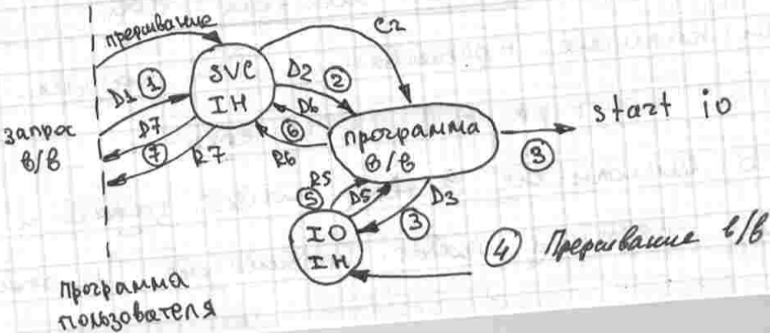
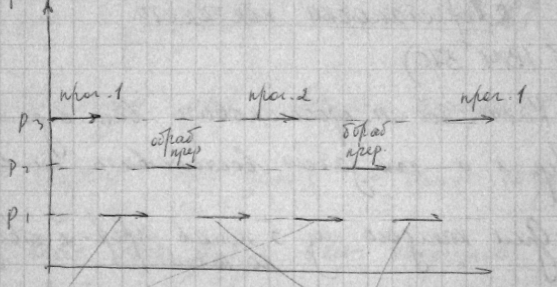
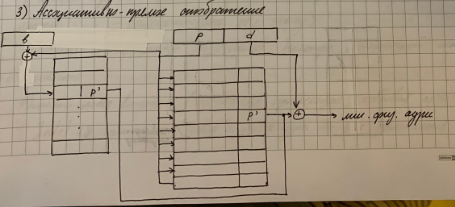
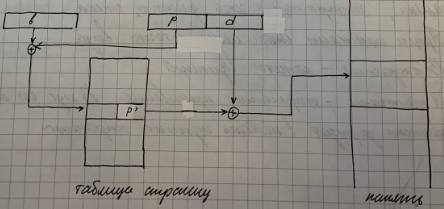
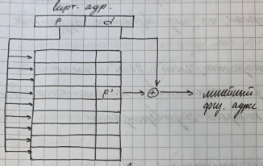
**Тупики :  определение  тупиковой  ситуации  для  повторно  используемых  ресурсов,  четыре  условия  возникновения тупика, обход тупиков ‐ алгоритм банкира**   
Повторно‐используемые ресурсы – количество в системе постоянно и при   
использовании они не изменяются  (или редко): аппаратура (ОП, ЦП), реентерабельные коды, системные таблицы (изменения в них могут вноситься  только супервизором), процедуры ОС (так как они являются реентерабельными).  Потребляемые ресурсы – количество в ОС переменно и произвольно: сообщения. Процесс может создать любое  количество сообщений. Процесс получения сообщения заканчивается его уничтожением.  Тупик  –  ситуация ,  возникающая  в  результате  монопольного  использования  разделяемых  ресурсов,  когда  процесс, владея ресурсом, запрашивает другой ресурс , занятый непосредственно или через цепочку запросов  другими процессами, ожидающими освобождение ресурса, занятого 1‐м процессом | Условия возникновения тупика в   
системе:1. Усл. взаимоисключения (процессы требуют предоставления права   
монопольного использования ресурсов); 2. Усл. ожидания ресурса  
 (процесс удерживает занимаемые им ресурсы и ожидает выделения доп. ресурсов).  
3. Усл. неперераспределяемости (ресурсы нельзя отобрать у процесса, их   
использующего, – только вернет сам); 4. Условие кругового ожидания (существует   
кольцевая цепь процессов, в которой каждый процесс удерживает  за собой один или более ресурсов, которые необходимы следующему в этой цепи процессу).    
Методы борьбы с тупиками: предотвращение (исключение), обход (недопущение), обнаружение и восстановл.  Обход или недопущение тупиков связаны с анализом запросов. Предполаг., что тупик потенциально возможен,  но в ОС создаются такие условия, при которых тупик становится невозможен . Очевидно , что анализ ситуации  связан с анализом запросов ресурсов процессами.  | Алгоритм Банкира: В качестве банкира   
выступает менеджер ресурсов . Процессы указывают в своих заявках, макс . потребность в  ресурсах данного типа. Процесс не м. затребовать больше ресурсов, чем указано в его заявке. Условия:  1) число процессов в ОС фиксировано; 2) число ресурсов в ОС   
фиксировано; 3) процесс не м. запросить > ресурсов, чем есть в ОС ; 4) процесс не м. запросить > ресурсов, чем в его заявке; 5) всех распределенных ресурсов данного   
класса не  м.б. > ∑ ресурсов данного класса в ОС Алгоритм: 1. Каждый запрос   
проверяется на отношение к количеству ресурсов в системе; 2) Каждая заявка   
проверяется относительно суммы всех заявок на ресурсы; 3) Менеджер ресурсов   
при получении очередной заявки в ОС ищет такую последовательность процессов, кот. м.  гарантированно завершиться (ПРИМЕР НИЖЕ!!!) | Состояние безопасное,   
если: Существует процесс, кот. завершится; Существует процесс, которому хватит   
всех единиц от освободившегося 1‐го; Существует i‐й процесс, все i‐1 процессы  
освободили все ресурсы, всех свободных ресурсов хватит при  макс. запросе i‐го процесса. 

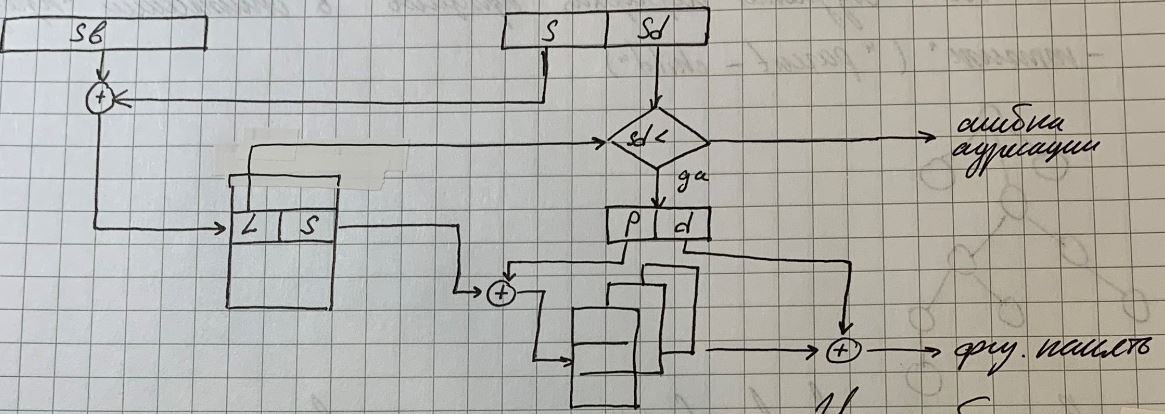
**Win32 API :  CreateThread(),  WaitForSingleObject(), WaitForMultipleObject().**   
Поток – часть последовательного кода процесса, которая может выполняться || с другими частями кода.    
HANDLE CreateThread(   
LPSECURITY\_ATTRIBUTES lpThreadAttributes, // дескриптор защиты   
SIZE\_T dwStackSize, // начальный размер стека LPTHREAD\_START\_ROUTINE lpStartAddress, // функция потока LPVOID lpParameter, // параметр потока   
DWORD dwCreationFlags, // опции создания   
LPDWORD lpThreadId // идентификатор потока );  
Если  функция  завершается  успешно,  величина  возвращаемого  значения  –  дескриптор  нового  потока .  Если  функция завершается с ошибкой, величина возвращаемого значения ‐ ПУСТО (NULL).  Поток создается с приоритетом потока THREAD\_PRIORITY\_NORMAL.  Когда поток заканчивает работу, объект потока приобретает сигнального состояния, удовлетворяя любые потоки,  которые ждали объект. Объект потока остается в системе, до тех пор, пока не поток закончит работу , и все  дескрипторы к нему не будут закрыты через вызов CloseHandle. **|** DWORD WINAPI WaitForSingleObject  
(    \_\_in  HANDLE hHandle,   // дескриптор объекта    \_\_in  DWORD dwMilliseconds  // время ожидания  );  Ожидает и возвращают значение тогда, когда происходит одно из ниже перечисленного:  Указанный объект находится в сигнальном состояние. ; Интервал  времени  простоя  истекает . Интервал  времени  простоя  может  быть    
установлен  в   INFINITE  (БЕСКОНЕЧНО), чтобы определить, что ожидание будет непрерывным.  Возвращает событие, заставившее функцию завершиться:  •  WAIT\_ABANDONED ‐ поток, владевший объектом, завершился, не переведя объект в сигнальн. состояние  •  WAIT\_OBJECT\_0 ‐ Объект перешел в сигнальное состояние  •  WAIT\_TIMEOUT ‐ Истек срок ожидания  •  WAIT\_FAILED ‐ Произошла ошибка (например, получено неверное значение hHandle)**|** DWORD WINAPI WaitForMultipleObjects  
(    \_\_in  DWORD nCount,     // количество дескрипторов    \_\_in  const HANDLE \*lpHandles,  // массив дескрипторов    \_\_in  BOOL bWaitAll,      // ждать всех (true) или хотя бы одного (false)    \_\_in  DWORD dwMilliseconds   // время ожидания  );  Возвращаемые значения те же, за исключением одного: WAIT\_OBJECT\_N, где N – объект, заставивший функцию  завершиться.**|** В  Windows  реализован  механизм  мьютексов  (mutex  
  –  mutual  exception  –  взаимоисключение).  Может  использоваться параллельными процессами. Также в Windows имеется системный вызов CRITICAL\_SECTION (в  Рихтере:  CRITICAL\_SECTION  –  структура  данных,  а  вот EnterCriticalSection()  и LeaveCriticalSection()  и  есть  уже  системные вызовы). Эти системные вызовы предназначены только для потоков.  Мьютексы, семафоры и события позволяют синхронизировать потоки, используемые в разных процессах, чтоб  одно приложение могло уведомить другое об окончании той или иной операции.  Объекты ядра мьютксы гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Мьютексы  ведут  точно  также  как  критические  секции,  однако  они  являются  объектами  пользовательского  режима .  Недостаток – невозможность синхронизации потоков разных процессов.  Функции API: CreateMutex() — создать мьютекс; WaitForSingleObject() — ждать освобождения мьютекса, eventa  или семафора, ReleaseMutex() — освободить мьютекс.Пример: HANDLE Writing, CanRead, Writer;      
CanRead = CreateEvent(NULL, false, false, NULL);    
Writing = CreateMutex(NULL, false, NULL);      
WaitForSingleObject(Writing, INFINITE);      
ReleaseMutex(Writing);    SetEvent(CanRead);      
Writer = CreateThread(NULL, NULL, Procedure\_name, Param, NULL, NULL);

**Процессы:  бесконечное  откладывание ,  зависание ,  тупиковая  ситуация  ‐  анализ  на  примере  задачи  об  обедающих философах. Считающие и множественные семафоры. Мониторы: монитор кольцевой буфер.** Процесс ‐ программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).   Асинхронные процессы – процессы , каждый из которых выполняется с собственной скоростью . Все процессы в системе  являются асинхронными, то есть нельзя сказать, когда процесс придёт в какую‐то точку.  1. Возможно, что оба процесса пройдут цикл   
ожидания и попадут в свои критичские секции ‐ ок ; 2. Бесконечное откладывание   
(зависание) – ситуация, когда разделённый ресурс снова захв. тем же процессом;  
 3. Тупик (deadlock, взаимоблокировка) – ситуация , когда оба процесса установили   
флаги занятости и ждут . Т .е. каждый  ожидает освобождения ресурса, занятого другим процессом.  Проблема обедающих философов.   
Существуют только 3 схемы действия философов:  1. пытается взять обе вилки сразу.   
Если удается, он может есть. -> Бесконечное откладывание; 2. берет левую вилку   
и пытается взять правую (левую держит в руке) -> Тупик;   3. берет левую вилку,   
и если не удается взять правую, то кладет левую вилку обратно -> Ok  | Семафор – неотрицательная защищённая переменная S, над которой определено 2 неделимые операции:  Р (от датск. passeren ‐ пропустить) и V (от датск. vrygeven ‐ освободить). Защищенность семафора означает, что значение  семафора м. изменяться только операциями P и V.  1. Операция P(S): S = S – 1. Декремент семафора   
(если он возможен). Если S = 0, то процесс, пытающийся выполнить  операцию Р, будет заблокирован на семафоре в ожидании, пока S не станет больше 0. Его освобождает другой  процесс, выполняющий операцию V(S).    
2. Операция V(S): S = S + 1. Инкремент S одним неделимым действием   
(последовательность непрерывных действий:  инкремент, выборка и запоминание). Во время операции к семафору нет доступа для других процессов. Если S = 0,  то V(S) приведёт к S = 1. Это приведёт к активизации процесса, ожид. на семафоре.  P(S) и V(S) есть неделимые (атомарные) операции.  Суть:  процесс  пытающий  выполнить  операцию  P(S)  блокируется ,  становится  в  очередь  ожидания  данного  семафора,  освобождает его другой процесс, который выполняет V(S). Таким образом исключается активное ожидание.  Семафоры поддерживаются ОС‐ой. Семафоры устраняют активное   
ожидание на процессоре.  Семафоры бывают: бинарные (S принимает значения 0 и 1), считающие (S принимает значения от 0 до n), множественные  (набор считающих семафоров). Все семафоры 1 набора м. устанавливаться 1 операцией.  В Windows после освобождения на семафоре приоритет процесса повыш. Изменение S м. рассматривать как событие в ОС.Монитор – языковая конструкция, состоящая из структур данных и подпрограмм, использующих данные структуры.  Монитор защищает данные. Доступ к данным монитора могут получить только п/п монитора. В каждый момент времени в  мониторе м. находиться только 1 процесс. Монитор является ресурсом. Монитор «кольцевой буфер» –   
решает задачу «производство‐потребление», то есть существуют процессы‐производители и  процессы‐потребители, а также буфер – массив заданного   
размера, куда производители помещают данные, а потребители  считывают оттуда данные в том порядке, в котором они помещались (FIFO).  
 **Синхронизация процессов ОС Linux на примере задачи «производство-потребление»**  


**Процессы:  процесс  как  единица  декомпозиции  системы.  Контекст  процесса.  Переключение  контекста.  Классификация  алгоритмов  планирования;  алгоритм  адаптивного  планирования;    ситуация  ‐  бесконечное  откладывание – причины возникновения** Процесс ‐ программа в стадии выполнения. Единица декомпозиции ОС (именно ему выделяются ресурсы ОС).   М. делиться на потоки, программист созд. в своей программе потоки, которые выполняются квазипараллельно.Аппаратный контекст процесса – сост. регистров.  Полный контекст процесса – сост. регистров + сост. памяти.  Планирование – управл. распредел. ресурсов ЦП между разл. конкур. процессами путем  передачи им управления согласно некот. стратегии планирования. Диспетчеризация  – выделение процессу процессорного времени.  Контекст  выполнения  –  Функции  ядра  могут  выполняться  в  контексте  процесса ,  либо  в  системном  контексте.  При  выполнении в контексте процесса ядро функционирует от имени текущего процесса, имея доступ к данным тек. процесса,  стеку ядра, адресному пространству процесса (всё это – user area). Ядро м. заблокировать процесс.  При  выполнении  в  контексте  ядра,  оно  обслуживает  прерывание  от  внешних  устройств  и  выполняет  пересчёт  приор.  процессов. Всё это выполн. в сист. контексте (контексте прерываний). Переключ. – мультизадачность.  Процесс может быть разделён на параллельно выполняющиеся потоки (threads).  Программа‐планировщик – программа,   
отвечающая за управление использованием совместного ресурса.  Последовательное выполнение – Квазипараллельное выполнение – Реальная параллельность. Планирование: с/без вытеснения, с/без переключения, с/без приоритетов. Бесконечное откладывание – ситуация, когда процесс никогда не получает необх. для выполнения ресурсов (точнее, кванта  времени). Возникает, когда диспетчер всегда отдаёт квант др. процессу, т.к. его приоритет >.  | Классификация алгоритмов планирования: с/без переключения, с/без вытеснения, c/без приоритетов; | Бесконечное откладывание – ситуация , когда процесс никогда   
не получает необх . для выполнения ресурсов  (точнее, кванта времени). Возникает, когда диспетчер всегда отдаёт квант др. процессу, т.к. его приоритет >.  | Алгоритмы планирования в системе разделения времени: 1) Round Robin (RR): циклическое планирование. очередь процессов (FIFO). Когда у текущего процесса истекает квант, он опять помещается в конец очереди.   
2) Адаптивное планирование (многоуровневые очереди): несколько очередей с приоритетами. Новые и “проснувшиеся” процессы попадают в первую очередь. Если они не успевают завершиться или заблокироваться на вводе/выводе, то попадают в следующую очередь и т.д. Последняя очередь работает по схеме RR.  
  
 3) Адаптивное планирование (по памяти): ​ дополнительно оценивается объем занимаемой в данный момент процессом физической памяти и необходимый объем для дальнейшего выполнения. В случае, если процесс может получить этот объем. то он выполняется. Если нет – откладывается; 4) Windows: т.к. в ОС процессы неравнозначны, система разделения времени пытается выделить процессорное время справедливо: учитывается время ожидания, простоя, приоритет блокировки. Процесс требующий много процессорного времени окажется в самой низкой очереди. Интерактивные (мышь, клавиатура) будут в высшей очереди.  
 | Алгоритмы планирования в системе пакетной обработки: 1) FIFO - очередь. Без переключения. Без приоритетов; 2) SJF (shortest job first) - может возникнуть бесконечное откладывание. Необходима априорная информация о времени выполнения. Статический приоритет. Без переключения; 3) SRT (shortest remaining time) - с вытеснением. Выполняющийся процесс прерывается, если в очереди появляется процесс с меньшим временем   
выполнения (меньше оставшегося до завершения времени); 4) HRV (Highest   
Response Ratio) - наиболее высокое относительное время ответа –  
c динамическими приоритетами. Используется в UNIX. Чем больше ожидание, тем выше приоритет. Позволяет избежать бесконечного откладывания. (tw - время ожидания, ts - запрошенное время обслуживания) priority = (tw-ts)/ts

**Синхронизация  потоков  в  ОС Windows:  мьютексы,  события ;  пояснить  особенности  использования  на  примере задачи «читатели и писатели».**   
   
   
   
Мьютекс (mutex – mutually exclusive (взаимно исключающий)) ‐ механизм синхронизации, используемый для упорядоч.  доступа к ресурсам. Этот объект обеспечивает исключительный (exclusive) доступ к охраняемому блоку кодов. Также  мьютекс  позволяет  установить  время  блокировки  ресурса.  Мьютекс  предоставляет  доступ  к  объекту  любому  из  потоков, если в данный момент объект не занят, и запоминает текущее состояние объекта. Если объект занят , то  мьютекс запрещает доступ.   
Однако можно подождать освобождения объекта с помощью функции WaitForSingleObject,  в которой роль управляющего объекта выполняет тот же мьютекс.   HANDLE CreateMutex ( LPSECURITY\_ATTRIBUTES lpMutexAttributes, // атрибут безопастности BOOL bInitialOwner, // флаг начального владельца LPCTSTR lpName // имя объекта  
 ); | Событие (event) ‐ это объект, выполняющий роль переключателя.   
У него есть только два состояния: вкл и выкл.  Событие  создается  и  помещается  в  коде  соответствующего  потока,  где  ведется  наблюдение  за  состоянием  объекта. Если объект события выключен, ждущие его потоки блокируются.   HANDLE CreateEvent ( LPSECURITY\_ATTRIBUTES lpEventAttributes, // атрибут защиты BOOL bManualReset, // тип сброса TRUE - ручной BOOL bInitialState, // начальное состояние TRUE - сигнальное LPCTSTR lpName // имя обьекта);   
  
  
  
  
**Управление  памятью:  выделение  памяти  разделами  фиксированного  размера,  выделение  памяти  разделами переменного размера, стратегии выделения памяти, фрагментация памяти** В современных системах имеется иерархия памяти.  Чем ближе к процессору, тем быстрее должна быть память.  Существует вертикальное управление памятью связанное с передачей данных с уровня на уровень.   Существует горизонтальное управление связанное с управление конкретным уровнем.  Распределение памяти разделами фиксированного размера.  Существует 3 типа разделов небольшой, средний и  большой. Размер определялся до работы системы и не менялся.  Минусы: свободные, не используемые участки памяти; в одной очереди может быть заданий много, а в  другой не быть совсем. Выход: 1 очередь – но тогда возникает не эффективное использование памяти. В процессе работы системы размеры блоков были известны и не менялись, что позволяло использовать  абсолютные адреса. Если адресное пространство процесса не помещалось в блок фиксированного размера, то  такой процесс откладывался. Для каждого задания ОС было известно априорно , сколько времени оно будет  выполняться | Распределение памяти разделами переменной длины   
  
В  результате получаем фрагментацию – ситуацию, когда в результате многократной загрузки‐выгрузки  появляется большое количества небольших свободных   
участков памяти, в которые ничего нельзя записать. ОС  должна обладать средствами, которые позволяли бы ей объединять свободные участки ОП. Существует три  стратегии выбора раздела для загрузки задания:  1) Выбирается первый попавшийся,  
 подходящий по размеру ; 2) Выбор самого «тесного» (больше всего  
 соответствующего по размеру, оставляющего меньше свободного  места);   
3) Выбор самого «широкого» (В оставшееся адресное пространство можно   
загрузить ещё одно задание) 

**Тупики : Обнаружение тупиков для повторно используемых ресурсов методом   
редукции графа , способы  представления графа и методы восстановление   
работоспособности системы** Условия возникновения тупика в системе:    
1) Условие взаимоисключения (процессы треб. предоставления права монопольного  
 использования ресурсов)  2) Условие  ожидания  ресурса  (процесс  удерживает  уже   
 выделенные  ресурсы  и  ожидает  выделения  дополнительных ресурсов)    
3) Условие неперераспределяемости ресурсов (ресурсы нельзя отобрать у процесса,  
 их использующего, до тех  пор, пока процесс сам не вернёт их системе)  4) Условие кругового ожидания (существует кольцевая цепь процессов, в которой каждый процесс удерживает  за собой один или более ресурсов, которые необходимы следующему в этой цепи процессу) | Обнаружение тупиков и восстановление   
работоспособности. Формализуем задачу: будем рассматривать  систему как декартово произведение множества состояний, где под состоянием понимается состояние ресурса  (свободен  или  распределён).  При  этом  состояние  может  измениться  процессом  в  результате  запроса  и  последующего получения ресурса, а также в результате освобождения процессом занимаемого им ресурса. Если  в системе процесс не может ни получить, ни вернуть ресурс, то говорят, что система находится в тупике, то есть  не может поменять своё состояние в результате выделения или освобождения ресурса. Определить, что какое‐то  количество процессов находится в тупике можно при   
помощи графовой модели Холдта.  Граф  L = (X, U, P) задан,  если  даны   
 множества  вершин X ≠∅   и  множество  рёбер  U≠∅ ,  а  также  инцидентор    
(трехместный  предикат)  Р,  причём  высказвание  P(x,u,y)   означает   
высказывание  «Ребро  и  соединяет вершину х с вершиной y», а также удовлетворяет двум условиям: 1) предикат Р определён на всех таких упорядоченных тройках (х,и,у),  
 для которых  yx ∈ X ,  и  u∈ U; 2) каждое ребро, соединяющее какую‐либо упорядоченную пару вершин x и у кроме неё может соединять  только обратную пару у, х.  | Дуга – ребро, соединяющее х с у, но не у с х.  Дуги бывают   
двух видов: запросы и выделения. Таким образом модель Холдта представляет собой  
двудольный  (бихроматический)  граф ,  где Х  разбивается  на  подмножество    
вершин‐процессов  π = {p1, p2, …, pn}  и  подмножество вершин‐ресурсов  ρ={r1, r2, …, rn} . π, ρ: X = ρ ∪ π, ρ ∩ π = ∅.  Приобретение (выделение) – дуга  (r,p) где  r ∈ ρ, p ∈ π.  
Запрос – дуга  (p,r), где r ∈ ρ, p ∈ π.  Обнаружить  процесс,  попавший  в  тупик,   
можно методом  редукци (сокращения)  графа.  Формализуем  процедуру сокращения:   
1) Граф сокращается по вершине рi, если эта рi не является ни заблокированной,  
 ни изолированной, путём  удаления всех рёбер, входящих в рi и выходящих из неё.   
2) Процедура  сокращения  соответствует  действиям  процессов  по  приобретению    
запрошенных  ранее  ресурсов  и  последующего  освобождения  всех  занимаемых  процессом  ресурсов.  В  этом  случае Pi  становится изолированной вершиной. | Представление графов:  1) Матрица ; 2) Связный список. В простейшем варианте двудольный (бихроматический) граф м.б. описан 2 матрицами:1. матрица запросов (отраж. запросов проц.) – A = {p, r}  2. матрица распредел. (кол‐во рес, выдел. к‐л. проц.) – B = {r, p}  | Восстановление  тупика.  Система    
д . поддерживать ср‐ва приостановки возобновления. Последов. прекращ.  
процессов, попавших в тупик, в опред. порядке, пока пока не б. достаточно ресурсов для устранения тупика.  1. М. отбирать ресурсы у всех процессов;     
2. М. отбирать ресурсы только у процессов, попавших в тупик.  Минимизация м.б.  
осущ. на приор. проц:  • на основе цены повт. запуска проц. от текущей точки  • на основе внешней цены  Д.б.  корректное  освобождение  ресурса  –  процесс  д.  вернуться  к  состоянию (точке),  предшеств .  запросу  на  данный ресурс  
 (перед кажд. запросом необх. сохр. сост. процесса) – откат  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
**Виртуальная память: распределение памяти страницами по запросам, свойство локальности, анализ страничного поведения процессов, рабочее множество.**  
Virtual Memory - система при которой рабочее пространство процесса частично располагается в основной памяти и частично во вторичной. При обращении к какой либо памяти, система аппаратными средствами определяет, присутствует ли область физической памяти, если отсутствует, то генерируется прерывание, это позволяет супервизору передать необходимые данные из вторичной в основную. Используется адресное пространство диска как область свопинга или педжинга, т.е. для временного хранения областей памяти. Виртуальная память - память, размер которой превышает размер реального физического пространства. Используется адресное пространство диска как область свопинга или пейджинга, т.е. для временного хранения областей памяти. | Подходы к реализации управления виртуальной памятью: 1. страничное распределение памяти по запросам; 2. сегментное распределение памяти по запросам; 3. сегментно - страничное распределение памяти по запросам   
Страница - является единицей физического деления памяти. Её размер устанавливается системой. Сегмент - является единицей логического деления памяти. Её размер определяется объемом кода. сегментное распределение памяти по запросам | Распределение памяти страницами по запросам: Адресное пространство процесса и адресное пространство физической памяти делится на блоки равного размера. Блоки, на которые делится адресное пространство процесса называют страницами, а блоки на которые делится физическая память кадрами, фреймами или блоками. Процесс копируется в страничный файл в области свопинга, таким образом, для него создается виртуальное адресное пространство. Соответственно размер виртуального адресного пространства может превышать объем физической памяти. Для возможности отображения страниц на соответствующие блоки физической памяти необходимо аппаратно поддерживаемое преобразование адресов (иначе слишком долго).  
+ легко реализовать, алгоритм LRU в этом эффект; -сложность коллективного использования. | Сегмент представляется в виде совокупностие страниц, что позволяет устранить проблемы, связанные с перекомпоновкой и ограничением размера сегмента. В системе с сегментно-страничной организацией применяется трехкомпонен. (трехмерная) адресация. Виртуальный адрес определяется как упорядоченная тройка v=(s,p,d), где s- номер сегмента, р- номер страницы в сегменте, d-смещение в странице, по которому находится нужный элемент.  
+ страничного: легко реализовать, алгоритм LRU достаточно эффективен;   
- страничного: сложность коллективного использования. | + сегментного: легко реализовать коллективное использование, т.к. сегмент является лог. единицей деления памяти; -сегментного: необх. корректировки таблицы дескрипторов всех процессов при изменении размеров сегментов; сложн. при загрузке новых сегментов; Фрагментация  
(1 – прямое отображение; 2 – ассоциативное отображение; 3 – ассоц.-прям. отображ.)   




**Ядро ОС: многопоточное ядро; взаимоисключение в ядре – спин ‐ блокировки**Системные  процессы  (выполняющиеся  в  режиме  ядра)  также  нуждаются  в  механизме  взаимоисключения .  Критическими секциями ядра являются разделы кода ядра, в которых , например , модифицируются данные  диспетчера  ядра  или  DPC  ((Deferred  Procedure  Call)  отложенный  вызов  процедуры).  Наибольшую  проблему  представляют прерывания, так как они могут возникнуть в любой момент, например в момент обновления БД  ядра, при этом обработчик прерывания может также модифицировать ту же самую БД ядра.  DPC – управляющий объект ядра , который описывает запрос и  позволяет отложить обработку запроса до  повышения IRQL до уровня DPC/dispatch (диспетчеризации)     (на примере Windows)  Пользоваться запретом прерываний нельзя.  В ядре выполняется большое число потоков. В частности , каждый драйвер представляет отдельный поток.  Необходимо корректно решать задачу взаимоисключения.Не урегулирован вопрос монопольного  использования указателя на конец  очереди.  В однопроцессорных и  многопроцессорных системах проблема  монопольного использования решается  взаимоисключением. Механизм “spin lock”  В ядре возможно только постоянно в  цикле проверять переменные блокировки  (активное ожидание). Другие способы не возможны | SpinLock: В  основе  лежит  команда test\_and\_set. Простейшая  форма  механизмов  в  ядре  базируется  на  аппаратной  реализации.    
Существуют  еще  более  простые  механизмы:  InterLockedIncrement,  InterLockedDecriment,  InterLockedExchange. При реализации этих команд блокируется многопроцессорная шина на время операции,  чтобы  другой  процесс  не  мог  выполнить  команды.    
Основная  проблема  в  ядре –  прерывания.  Но  запрет  прерываний не является хорошим выходом. В Win2000 запрещены только некоторые прерывания, обработчик  которых использует этот же ресурс.  Пусть код работает на IRQL\_Passive. Этот код захватил spin‐блокировку. После  этого код прерван кодом  с  уровнем IRQL\_Dispatch, который  
 пытается захватить ту же spin‐блокировку. Для этого он входит в бесконечный  цикл   
ожидания, в котором проверяет значение переменной и блокирует все действия в системе. Этот цикл таким  образом никогда не закончится.  Для  корректной  работы,  необходимо  обеспечить  выполнение  определенных действий при захвате спинблокировки:   
не позволить коду с более высоким уровнем IRQL прерывать код с более низким – повышать  текущий уровень IRQL в момент захвата спин‐блокировки до некоторого  
уровня, который связан с этой спинблокировкой.  При освобождении уровень понижается до прежнего значения.   
void spin\_lock(spin\_lock\_t \*c) // c - condition - переменная типа условие  
{while(test-and-set(c) != 0) /ресурс занят/}  
void spin\_unlock(spin\_lock\_t \*c) {c = 0;}  
Блокировка шины памяти происходит т.к. на одном процессе крутится spin\_lock, а остальные процессы не могут обратиться к памяти. Данная проблема решается с помощью 2х вложенных циклов:void spin\_lock(spin\_lock\_t \*c){  
while(test-and-set(c) != 0) // команда устроена таким образом, что шина данных будет блокированна   
while(\*c != 0); //внутренний цикл выполняется путем проверки переменной с, без захвата шины данных }

**Прерывания в последовательности ввода\вывода**    
Прерывание сигнализирует о наступлении некоторого события. При программируемом вводе/выводе CPU  должен ждать готовности устройства и постоянно опрашивать флаг его состояния (тратятся циклы процессорного  времени). При вводе/выводе с прерыванием CPU передает контроллеру команду ввода/вывода и переходит к выполнению  другой работы. Когда контроллер будет готов обменяться данными с CPU, он пошлет CPU сигнал прерывания  (сообщит, что его необходимо обслужить).  | Чтение: CPU генерирует команду READ, сохраняет содержимое счетчика команд и других своих регистров и переходит к  выполнению других операций. В конце каждого цикла команды CPU проверяет наличие прерываний. При поступлении прерывания от  контроллера ввода/вывода, CPU сохраняет информацию о выполняемой в текущий момент задаче и выполняет  программу, обрабатывающую прерывание. При этом он считывает слова из регистров контроллера  ввода/вывода на шину данных и заносит их в память. По завершении обработки прерывания он восстанавливает  контекст программ, от которой поступило прерывание и продолжает выполнение.   
  
(С – вызвать или разбудить, R – возвратить или разбудить, D – данные, IH – interrupt handler, SVC -–super visor call) | Ввод/вывод с прерываниями более эффективен , чем программируемый, так как процессор освобождается от опроса флагов. Вся передача осуществляется через регистры CPU. | Программа 1, запросив операцию вв/выв была блокирована обработчиком прерывания, но перед этим система определила причину, вызвала обработчик => драйвер => инициализация операции вв/выв=> обработчик прерывания. Затем анализ правильности передачи данных. Затем процессор переключился на другую программу пока не получит прерывание. Затем повышает приоритет процесса.