Первое поколение компьютеров

ЭНИАК: ~ 20 тыс. электронных ламп, ежемесячно заменялось 2000 ламп

Первое поколение компьютеров

Элементная база

Первое поколение компьютеров

Элементная база	электронно-вакуумные лампы
Временной период	конец 1940-х – начало 1950-х годов

в 1946 г. в Пенсильванском университете США была разработана вычислительная машина ENIAC (Electronic Numerical Integrator and Computer), которая считается одной из первых электронных вычислительных машин — ЭВМ.

Особенности

- **однопользовательский, персональный режим**
- **зарождение класса сервисных, управляющих программ**
- **зарождение языков программирования**

Приоритетное направление: военные задачи

Операционные системы

Введение (часть 1)

1. Развитие вычислительной техники
2. Основы архитектуры вычислительной системы

Второе поколение компьютеров

Элементная база

Второе поколение компьютеров

Элементная база	полупроводниковые приборы: диоды и транзисторы
Временной период	конец 1950-х – вторая половина 1960-х годов

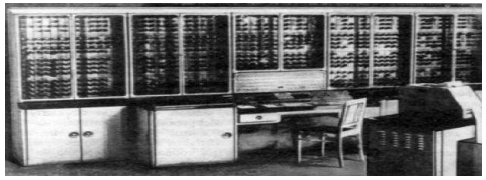
Особенности

- **пакетная обработка заданий**
- **мультипрограммирование**
- **языки управления заданиями**
- **файловые системы**
- **виртуальные устройства**
- **операционные системы**

Приоритетное направление: управление бизнес-процессами

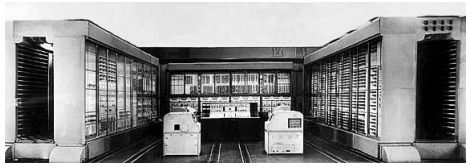
Первое поколение компьютеров

«Урал-1»



Первое поколение компьютеров

«Стрела»: быстродействие: 2000 трехадресных команд в секунду, основной такт — 500 мкс



Третье поколение компьютеров

IBM-360

Третье поколение компьютеров

Элементная база

Третье поколение компьютеров

Элементная база	интегральные схемы
Временной период	конец 1960-х – начало 1970-х годов

Особенности

- **аппаратная унификация узлов и устройств**
- **создание семейств компьютеров**
- **унификация компонентов программного обеспечения**

Второе поколение компьютеров

БЭСМ-6

Четвёртое поколение компьютеров

Apple 1 (1976г.)

Стив Джобс и Стив Возняк Apple 1: 4 Кбайт RAM, 8-разрядный микропроцессор MOS 6502 1 МГц

Четвёртое поколение компьютеров

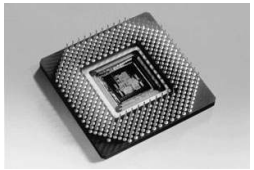
Altair-8800 (1974г.)

На базе микропроцессора Intel-8080 (1974г.). Программы вводились переключением тумблеров на передней панели, а результаты считывались со светодиодных индикаторов. **Объем памяти – 256 байт**

Пол Аллен и Бил Гейтс (Micro-soft) (1975г.) создали интерпретатор языка Basic **(4кб)**

Четвёртое поколение компьютеров

Элементная база



Первый микропроцессор Intel-4004 (1971г.) - 2250 элементов.

Первый универсальный микропроцессор Intel-8080 (1974г.) - 4500 элементов (основа для создания первых ПК).

16-битный микропроцессор Motorolla-68000 (1979 г.) - 70 000 элементов.

Первый 32-битный микропроцессор Hewlett Packard (1981 г.) - 450 тыс. элементов.

Четвёртое поколение компьютеров

Элементная база	большие и сверхбольшие интегральные схемы
Временной период	начало 1970-х – настоящее время

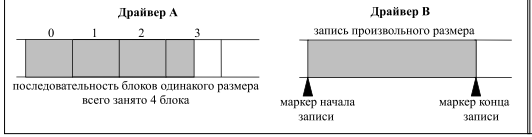
Особенности

- **«дружественность» пользовательских интерфейсов**
- **сетевые технологии**
- **безопасность хранения и передачи данных**

Управление физическими ресурсами вычислительной системы

систематизация и стандартизация правил программного использования физических ресурсов
(уровень драйверов физических устройств)

Драйвер физического устройства — программа, основанная на использовании команд управления конкретного физического устройства и предназначенная для организации работы с данным устройством.



Аппаратный уровень вычислительной системы

Характеристики физических ресурсов (устройств)

- правила программного использования
- производительность и/или емкость
- степень занятости или используемости

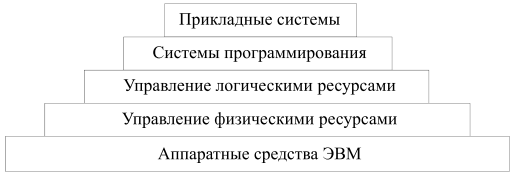
Средства программирования, доступные на аппаратном уровне

- система команд компьютера
- аппаратные интерфейсы программного взаимодействия с физическими ресурсами

Основы архитектуры вычислительной системы

Вычислительная система — совокупность аппаратных и программных средств, функционирующих в единой системе и предназначенных для решения задач определенного класса.

Структура вычислительной системы:



Четвёртое поколение компьютеров

Osborne I : один из первых ноутбуков (1980)

Восьмиразрядный процессор Zilog Z80A. Объем оперативной памяти составлял 64 Кбайта; Два пятидюймовых дисковода и модем.

Управление логическими (виртуальными) ресурсами

Средства программирования, доступные на уровнях управления ресурсами ВС

- система команд компьютера
- программные интерфейсы драйверов устройств (как физических, так и виртуальных)

Управление логическими (виртуальными) ресурсами

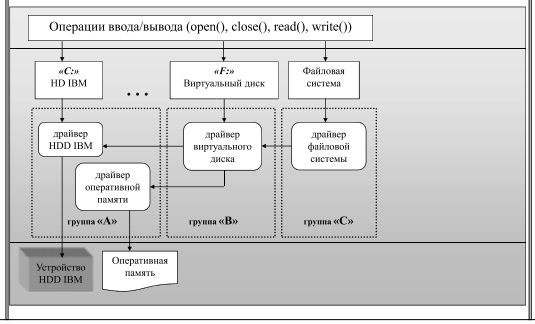
Характеристики ресурсов вычислительной системы

Ресурсы вычислительной системы — совокупность всех физических и виртуальных ресурсов.

Одной из характеристик ресурсов вычислительной системы является их **конечность -** возможна конкуренция за обладание ресурсом между его программными потребителями.

Операционная система — это комплекс программ, обеспечивающий управление ресурсами вычислительной системы.

Управление логическими (виртуальными) ресурсами



Управление логическими (виртуальными) ресурсами

Логическое (виртуальное) устройство (ресурс) — устройство (ресурс), некоторые эксплуатационные характеристики которого (возможно все) реализованы программным образом.

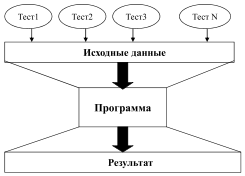
Драйвер логического (виртуального) ресурса — программа, обеспечивающая существование и использование соответствующего ресурса.

Жизненный цикл программы: тестирование и отладка

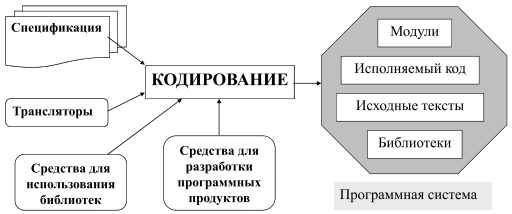
Тестирование — процесс проверки правильности функций программы на заранее определённых наборах входных данных (**тестах** или **тестовых нагрузках**).

Тестовое покрытие —набор тестов, наиболее полно покрывающих функции системы**.**

Отладка — процесс поиска, локализации и исправления зафиксированных при тестировании и эксплуатации программных ошибок.



Жизненный цикл программы: кодирование



Средства для разработки программных продуктов

- Средства автоматизации получения объектных модулей
- Отслеживание даты обновления
- Средства поддержки коллективной работы и контроль версий

Жизненный цикл программы: проектирование

Основные понятия:

Объектная ВС — ВС, на которой предполагается работа программного комплекса.

Инструментальная ВС — ВС, на которой будет вестись разработка программного комплекса.

Стандарты жизненного цикла программы в ВС опираются на современные представления о разработке программ.

Системы программирования

Система программирования — это комплекс программ, обеспечивающий поддержание жизненного цикла программы в вычислительной системе.

Жизненный цикл программы в вычислительной системе

1. Проектирование
2. Кодирование
3. Тестирование и отладка
4. Ввод программной системы в эксплуатацию (внедрение) и сопровождение

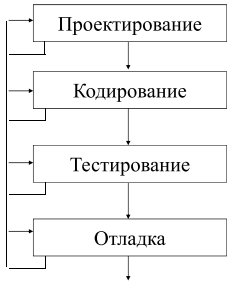
Модели разработки программных систем: каскадная итерационная модель

Преимущества

- В идеале максимальное удовлетворение заказчика продукта

Недостатки

- Недетерменированность времени и затрат



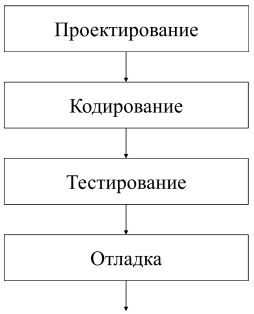
Модели разработки программных систем: каскадная модель

Преимущества

- Детерменированность времени и затрат

Недостатки

- Возможно устаревание к моменту реализации



Системы программирования

Современные технологии разработки программного обеспечения: модели разработки программных систем

1. Каскадная модель
2. Каскадная итерационная модель
3. Спиральная модель

Жизненный цикл программы: внедрение и сопровождение

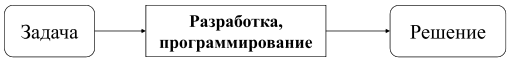
Внедрение — установка и первичная настройка программного комплекса на объектную ВС.

Сопровождение — исправление недочётов внедрения и проектирования программного комплекса.

Прикладные системы

Прикладная система — программная система, ориентированная на решение или автоматизацию решения задач из конкретной предметной области.

Первый этап развития прикладных систем



Системы программирования

Система программирования — это комплекс программ, обеспечивающий технологию автоматизации проектирования, кодирования, тестирования, отладки и сопровождения программного обеспечения.

Средства программирования, доступные на уровне системы программирования

- программные средства и компоненты системы программирования, обеспечивающие поддержание жизненного цикла программы.

Системы программирования: история

Начало 50-х годов XX века	Система программирования или система автоматизации программирования включала в себя ассемблер (или автокод) и загрузчик, появление библиотек стандартных программ и макрогенераторов.
Середина 50-х – начало 60-х годов XX века	Появление и распространение языков программирования высокого уровня (Фортран, Алгол-60, Кобол и др.). Формирование концепций модульного программирования.
Середина 60-х годов – начало 90-х XX века	Развитие интерактивных и персональных систем, появление и развитие языков объектно-ориентированного программирования.
90-е XX века – настоящее время	Появление промышленных средств автоматизации проектирования программного обеспечения, CASE-средств (*Computer-Aided Software/System Engineering*), унифицированного языка моделирования UML.

Модели разработки программных систем: спиральная модель

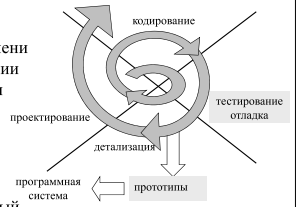
Наиболее современная и перспективная модель.

Преимущества

- Детерминированность времени выполнения каждой итерации
- Возможность рассмотрения системы до её завершения

Недостатки

- Недетерминизированность времени и затрат на конечный продукт (изначально)



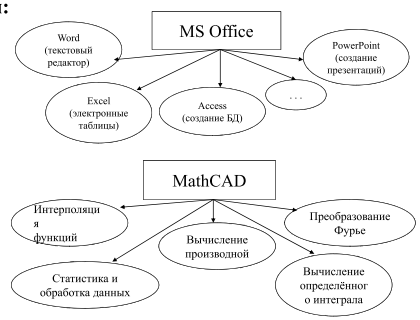
Прикладные системы

Основные тенденции в развитии современных прикладных систем

- Стандартизация моделей автоматизируемых бизнес-процессов
  - **B2B** (Business To Business)
  - **B2C** (Business To Customer)
  - **ERP** (Enterprise Resource Planning)
  - **CRM** (Customer Relationship Management)
- Открытость систем
  - **API** — Application Programming Interface

Прикладные системы

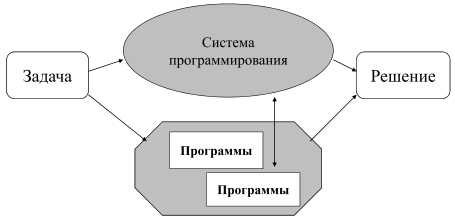
Примеры:



Прикладные системы

Третий этап развития прикладных систем

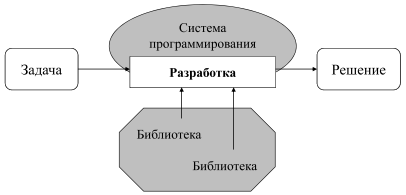
Третий этап характеризуется появлением *пакетов прикладных программ*



Прикладные системы

Второй этап развития прикладных систем

Второй этап — развитие систем программирования и появление средств создания и использования библиотек программ



Операционные системы

Введение (часть 2)

3. Основы компьютерной архитектуры
   - 3.1. Компьютер фон Неймана
   - 3.2. Аппарат прерываний
   - 3.3. Внешние устройства
   - 3.4. Аппаратная поддержка ОС и систем программирования

Выводы

Базовые определения и понятия:

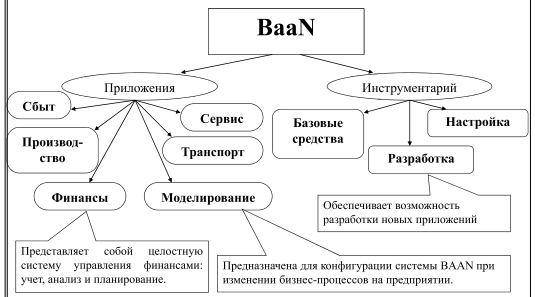
- Вычислительная система
- Физические ресурсы (устройства)
- Драйвер физического устройства
- Логические или виртуальные ресурсы (устройства)
- Драйвер логического (виртуального) ресурса
- Ресурсы вычислительной системы
- Операционная система
- Жизненный цикл программы в вычислительной системе
- Система программирования
- Прикладная система

Выводы

Пользователь и уровни структурной организации ВС

Прикладные системы	+ набор функциональных средств прикладной системы.
Системы программирования	+ трансляторы языков высокого уровня, библиотеки...
Управление логическими (виртуальными) устройствами	+ интерфейсы драйверов виртуальных устройств.
Управление физическими устройствами	+ интерфейсы драйверов физических ресурсов
Аппаратные средства	Система команд, аппаратные интерфейсы программного управления физическими устройствами

Прикладные системы



Оперативное запоминающее устройство

Использование содержимого поля служебной информации

1. Контроль за целостностью данных

При записи слова в память контрольная сумма бит = 9 (1001b) ⇒ тег = **1**.

1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1

Ошибки нет

тег

При чтении машинного слова 16 бит вычисляется сумма бит = 9 (1001b) и сверяется со значением тега.

1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	0	1

Ошибка

При чтении машинного слова 16 бит вычисляется сумма бит = 8 (1000b), а тег = 1 ⇒ Сбой в работе ОЗУ

1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	1

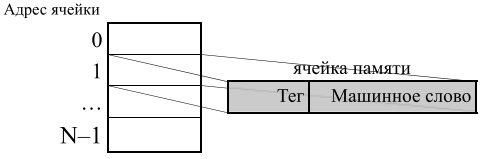
Ошибка не видна

При чтении машинного слова 16 бит вычисляется сумма бит = 7 (111b), и тег = 1 ⇒ Ошибка будет не выявлена

Пример контроля за целостностью данных по четности

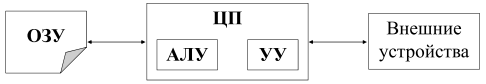
Оперативное запоминающее устройство

ОЗУ предназначено для хранения программы, выполняющейся в компьютере.
Тег — поле служебной информации.
Машинное слово — поле программно изменяемой информации.



Компьютер фон Неймана

Структура, основные компоненты компьютера фон Неймана



Принципы построения компьютера фон Неймана

1. Принцип двоичного кодирования
2. Принцип программного управления
3. Принцип хранимой программы

Компьютер фон Неймана

Историческая справка

- Джон фон Нейман (John Von Neumann)
- EDVAC (Electronic Discrete Variable Computer — Электронный Компьютер Дискретных Переменных)
- «Предварительный доклад о компьютере EDVAC» (A First Draft Report on the EDVAC)
- Джон Мочли (John Mauchly) и Джон Преспер Эккерт (John Presper Eckert)
- ENIAC (Electronic Numerical Integrator And Computer)

Оперативное запоминающее устройство

Расслоение памяти

Сравнение времени доступа

Последовательное чтение машинных слов по адресам:	ОЗУ без расслоения	ОЗУ с расслоением
A	~ t_{cycle}	~ t_{access}
A+1	~ t_{cycle}	~ t_{access}
A+2	~ t_{cycle}	~ t_{access}
Суммарное время:	~ $3*t_{cycle}$	~ $3*t_{access}$

Схема работы

Оперативное запоминающее устройство

Расслоение памяти

К независимых банков памяти, где $K = 2^L$.

Оперативное запоминающее устройство

Производительность оперативной памяти — скорость доступа процессора к данным, размещенным в ОЗУ.

- **Время доступа (access time, t_{access})** — время между запросом на чтение слова из оперативной памяти и получением содержимого этого слова.
- **Длительность цикла памяти (cycle time, t_{cycle})** — минимальное время между началом текущего и последующего обращения к памяти.

($t_{cycle} > t_{access}$)

Оперативное запоминающее устройство

Использование содержимого поля служебной информации

2. Контроль доступа к командам/данными
3. Контроль доступа к машинным типам данных

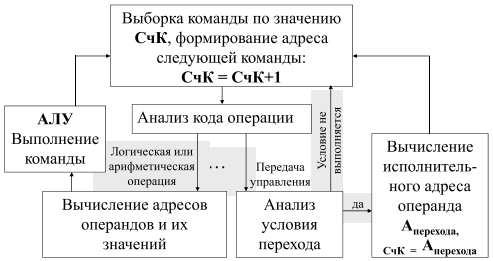
Центральный процессор

Кэш-память (cache memory) первого уровня (L1)

1. Обмен данными между кэшем и оперативной памятью осуществляется **блоками** фиксированного размера
2. **Адресный тег** блока — содержит служебную информацию о блоке (соответствие области ОЗУ, свободен/занят блок, …)
3. Нахождение данных в кэше – **попадание** (**hit**). Если искомых данных нет в кэше, то фиксируется **промах (cache miss)**
4. При возникновении промаха происходит обновление содержимого кэша **— вытеснение**. Стратегии вытеснения:
   - случайный выбор блока
   - вытеснение наименее «популярного» блока (LRU — Least-Recently Used)
5. Вытеснение кэша данных:
   - **сквозное кэширование** (write-through caching)
   - **кэширование с обратной связью** (write-back cache) — тег модификации (dirty bit)

Центральный процессор

Рабочий цикл процессора

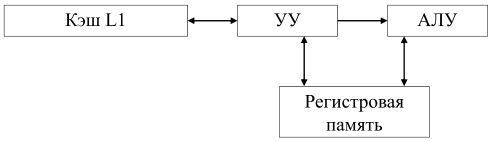


Центральный процессор

Регистровая память

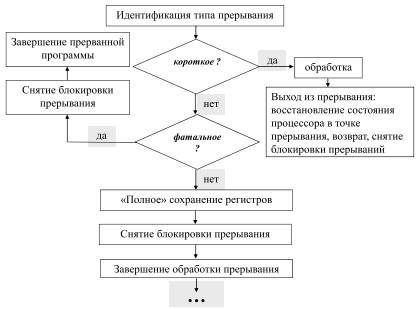
Центральный процессор

Устройство управления (control unit) — координация выполнения команд.
Арифметико-логическое устройство (arithmetic/logic unit) —выполнение команд, арифметической или логической обработки операндов.
Регистровая память (register memory) — совокупность устройств памяти процессора - регистров.
Кэш-память (cache memory) — буферизация работы процессора с оперативной памятью.



Аппарат прерываний

Программный этап обработки прерываний



Аппарат прерываний

Этап аппаратной обработки прерываний

Аппарат прерываний

Прерывание — событие в компьютере, при возникновении которого в процессоре происходит предопределенная последовательность действий.

Типы прерываний

- **Внутренние** — инициируются схемами контроля работы процессора
- **Внешние** — события, возникающие в компьютере в результате взаимодействия центрального процессора с внешними устройствами

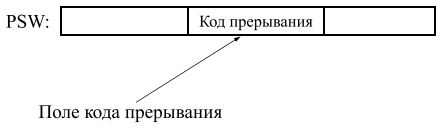
Центральный процессор

Использование кэш-памяти (cache memory)

- Сокращается количество обращений к ОЗУ
- Существенно увеличивается скорость доступа к памяти в случае использования ОЗУ с «расслоением»
- Усложнение логики процессора

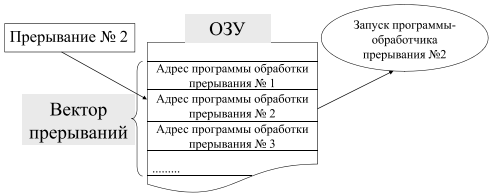
Аппарат прерываний

Модель организации прерываний с использованием регистра «слово состояние процессора»



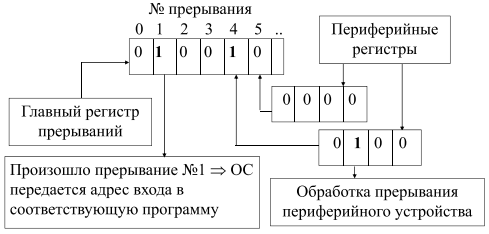
Аппарат прерываний

Модель организации прерываний с использованием «вектора прерываний»



Аппарат прерываний

Модель организации прерываний с использованием «регистра прерываний»



Аппарат прерываний

Модели организации прерываний

1. С помощью регистра прерываний
   Имеется специальный регистр прерываний, в определённые биты которого при возникновении прерываний заносятся «1».
2. С помощью вектора прерываний
   В фиксированной области памяти находится вектор адресов обработчиков прерываний.
3. С помощью слова состояния процессора
   Имеется специальный регистр, в котором хранится состояние процессора. При возникновении прерывания в определённую часть данного регистра заносится код прерывания.

Устройство последовательного доступа

Магнитная лента

Внешние устройства

Внешние запоминающие устройства (ВЗУ)

Обмен данными:

- записями фиксированного размера — **блоками**
- записями произвольного размера

Доступ к данным:

- операции чтения и записи (жесткий диск, CD-RW)
- только операции чтения (CD-ROM, DVD-ROM, …)

Последовательного доступа:

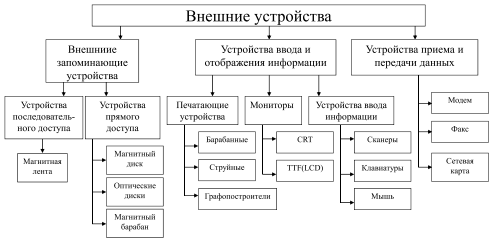
- Магнитная лента

Прямого доступа:

- Магнитные диски
- Магнитный барабан
- Магнито-электронные ВЗУ прямого доступа

Внешние устройства

Частичная иерархия внешних устройств



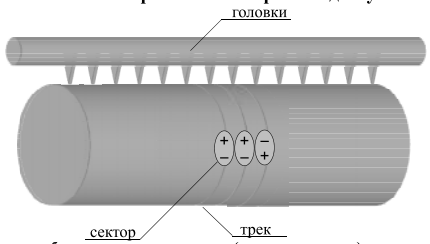
Внешние устройства

Модели синхронизации при обмене с внешними устройствами

[1] Примечание: процесс выполняется до возникновения следующего прерывания

Устройство прямого доступа

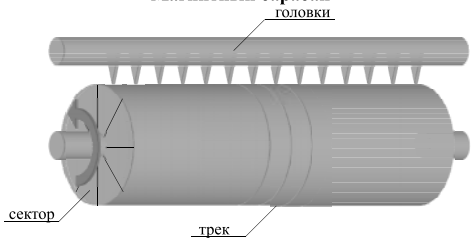
Магнитно-электронные ВЗУ прямого доступа



Операции, необходимые для начала чтения (позиционирования)

Устройство прямого доступа

Магнитный барабан



Операции, необходимые для начала чтения (позиционирования)

1. Поворот для совмещения головки с началом сектора

Устройство прямого доступа

Магнитные диски

Операции, необходимые для начала чтения (позиционирования)

1. Установка головки на требуемую дорожку
2. Поворот для совмещения головки с началом сектора

Аппаратная поддержка программных систем и мультипрограммного режима

Проблемы:

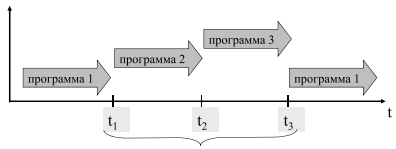
- Вложенные обращения к подпрограммам
- Накладные расходы при смене обрабатываемой программы
- Перемещаемость программы по ОЗУ
- Фрагментация памяти

Базовая аппаратная поддержка мультипрограммного режима

1. Аппарат **защиты памяти**
2. **Специальный режим операционной системы** (привилегированный режим или режим супервизора)
3. Аппарат прерываний (как минимум, **прерывание по таймеру**)

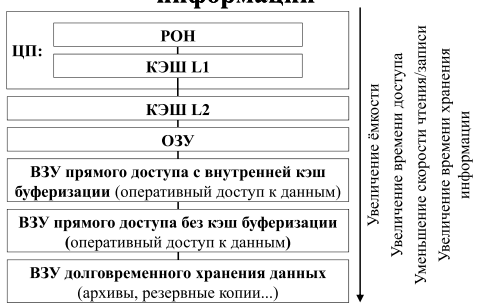
Аппаратная поддержка ОС и систем программирования

Мультипрограммный режим — режим, при котором возможна организация переключения выполнения с одной программы на другую.

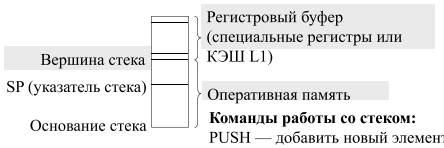


время обмена программы 1 (операции ввода/вывода)

Иерархия устройств хранения информации



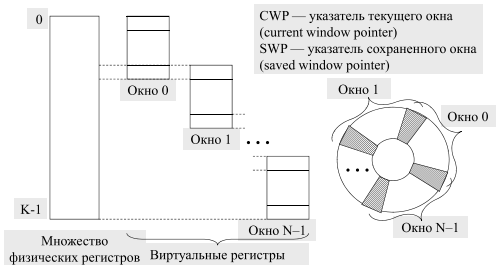
Аппаратная организация системного стека



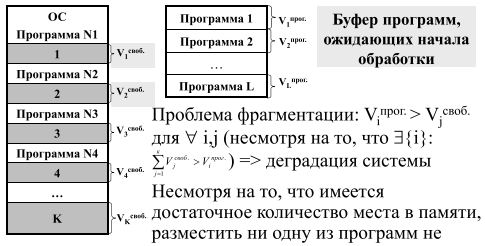
Команды работы со стеком:
PUSH — добавить новый элемент
POP — удалить элемент из вершины стека

Использование системного стека может частично решать проблему минимизации накладных расходов при смене обрабатываемой программы и/или обработке прерываний.

Регистровые окна (Register windows)



Фрагментация памяти



Проблема фрагментации: $V_i^{прог.} > V_j^{своб.}$ для $\forall$ i,j (несмотря на то, что $\exists\{i\}: \sum_{j=1}^{} V_j^{своб.} > V_i^{прог.}$) => деградация системы

Несмотря на то, что имеется достаточное количество места в памяти, разместить ни одну из программ не удаётся.

Перемещаемость программы по ОЗУ

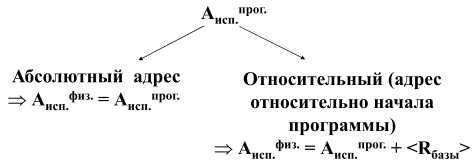
Соответствие адресов, используемых в программе, области ОЗУ, в которой будет размещена данная программа

Виртуальная память. Базирование

Базирование адресов — отображение виртуального адресного пространства программы в физическую память «один в один».

Виртуальная память. Базирование

Базирование адресов — решение проблемы **перемещаемости** программы по ОЗУ.

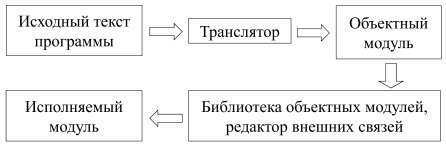


Виртуальная память. Базирование

Аппарат виртуальной памяти — аппаратные средства компьютера, обеспечивающие преобразование (установление соответствия) программных адресов, используемых в программе с адресами физической памяти, в которой размещена программа во время выполнения.

Базирование адресов — реализация одной из моделей аппарата виртуальной памяти.

Виртуальная память. Базирование



В исполняемом модуле используется программная (логическая или виртуальная) адресация

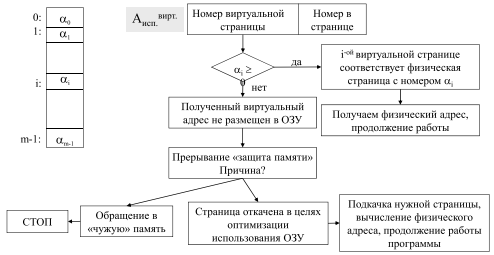
Проблема – установление соответствия между программной адресацией и физической памятью

Операционные системы

Введение (часть 3)

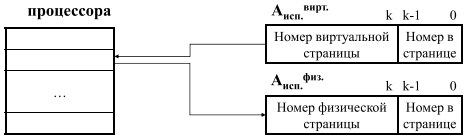
Виртуальная память. Страничная организация памяти

Модельный пример организации страничной виртуальной памяти



Виртуальная память. Страничная организация памяти

Таблица страниц процессора



Преобразование виртуального адреса в физический — замена номера виртуальной страницы на соответствующий номер физической страницы

Виртуальная память. Страничная организация памяти

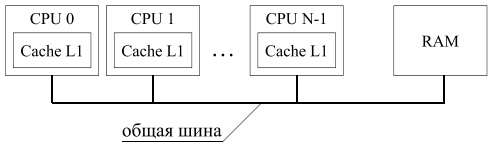
Страницы - блоки фиксированного размера. Размер страницы — 2^k

UMA и SMP системы

UMA: характеристики доступа любого процессорного элемента в любую точку ОЗУ не зависят от конкретного элемента и адреса (Все процессоры равноценны относительно доступа к памяти).

SMP-системы являются подвидом UMA-систем.

SMP-системы



Иерархия MIMD-систем

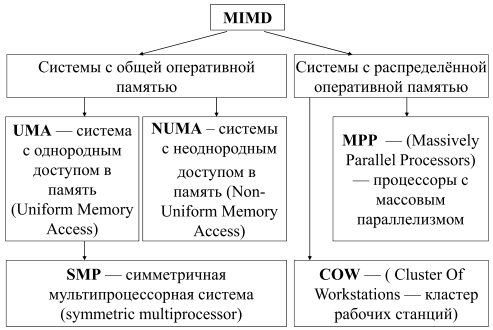
Системы делятся по принципу организации работы с ОЗУ

В системе с общей оперативной памятью имеется ОЗУ, и любой процессорный элемент имеет доступ к любой точке общего ОЗУ, то есть любой адрес может быть исполнительным для любого процессора

Виды систем с общей оперативной памятью

- UMA — Uniform Memory Access
  - SMP — Symmetrical MultiProcessing
- NUMA — Non-Uniform Memory Access

Иерархия MIMD-систем



Классификация архитектур многопроцессорных ассоциаций

Автор классификации: Майкл Флинн (M. Flynn)

- Поток инструкций (команд)
- Поток данных
- **ОКОД** (**SISD** — Single Instruction, Single Data stream) — компьютер с единственным ЦП
- **ОКМД** (**SIMD** — Single Instruction, Multiple Data stream) — матричная обработка данных
- **МКОД** (**MISD** — Multiple Instruction, Single Data stream) — ?
- **МКМД** (**MIMD —** Multiple Instruction, Multiple Data stream) — множество процессоров одновременно выполняют различные последовательности команд над своими данными

NUMA-системы

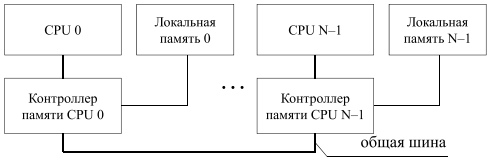
Проблема синхронизации кэша - несколько способов её решения:

- использовать процессоры без кэша (использовать только Cache L2)
- использовать **ccNUMA** (NUMA-системы с когерентными кэшами)

ccNUMA сложнее, но позволяет строить системы из сотен процессорных элементов.

NUMA-системы

NUMA: Процессорные элементы работают на общем адресном пространстве, но характеристики доступа процессора к ОЗУ зависят от того, куда он обращается.



SMP-системы

Преимущества SMP

- Простота реализации

Недостатки SMP

- Задержки при доступе к памяти
- Система с явной централизацией — общая шина является «узким местом»
- Проблема синхронизации кэша (решением является кэш-память с отслеживанием)
- Ограничение на количество процессорных элементов (как следствие централизации)

SMP-системы

Синхронизация кэша.

Поведение кэш-памяти с отслеживанием при чтении/записи

Операции	Локальный кэш	Кэш других процессоров
Промах при чтении	Запись из памяти в кэш	Ничего
Попадание при чтении	Использование кэша	Ничего (операция «не видна»)
Промах при записи	Запись в память	Соответствующая запись в кэше удаляется
Попадание при записи	Запись в память и кэш	Соответствующая запись в кэше удаляется

MPP-системы

Преимущества MPP

- Высокая эффективность при решении определённого класса задач

Недостатки MPP

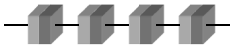
- Высокая стоимость
- Узкая специализация

MPP-системы

MPP — Специализированные дорогостоящие ВС. Эти компьютеры могут выстраиваться, процессорные элементы могут объединяться в различные топологии: макроконвейер, n-мерный гиперкуб и др.

Примеры топологий MPP-систем

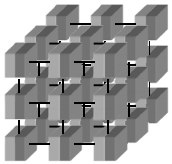
Макроконвейер:



Процессорный элемент с локальной памятью

Межэлементные коммуникации, определяющие топологию

3-мерный гиперкуб:



Иерархия MIMD-систем

Системы с распределённой оперативной памятью представляются как объединение компьютерных узлов, каждый из которых состоит из процессора и ОЗУ, непосредственный доступ к которой имеет только «свой» процессорный элемент. Класс наиболее перспективных систем.

Виды систем с распределённой оперативной памятью

- MPP — Massively Parallel Processors
- COW — Cluster Of Workstations

NUMA-системы

Преимущества NUMA

- Степень параллелизма выше, чем в SMP

Недостатки NUMA

- Централизация (ограничение ресурсом шины)
- Использование когерентных кэшей загружает шину служебной информацией

Недостатки ccNUMA

- Загрузка общей шины служебной информацией

Линии связи / каналы

Организация канала

- Коммутируемый канал
  Физически коммутируемая линия может иметь различные топологии
- Выделенный канал
  Обеспечивает доступ к системе на постоянной основе.
  Преимущества: отсутствие отказа, детерминированное качество соединения

Организация канала

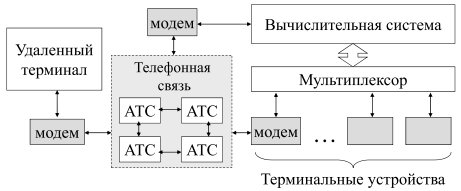
- Канал точка-точка
  Один абонент общается с одним абонентом (подключение к удалённому терминалу без мультиплексирования)
- Многоточечный канал
  Подключение терминала осуществляется через локальный мультиплексор

Направление движения информации

- Симплексные каналы
- Дуплексные каналы
- Полудуплексные каналы

Терминальные комплексы

Терминальный комплекс — многомашинная ассоциация, предназначенная для организации массового доступа удаленных и локальных пользователей к ресурсам некоторой вычислительной системы.



COW- кластеры

Преимущества COW

- «прозрачность» архитектуры
- относительная «универсальность» - возможность применения для решения широкого круга задач

Проблемы COW

- Топология

COW-кластеры

- Кластер как вычислительный узел (высокопроизводительная система)
- Кластеры, которые обеспечивают надёжность (сохранение работоспособности при возможном снижении производительности)

Компьютерные сети

Модели построения компьютерной сети:

- Сеть коммутации каналов
- Сеть коммутации сообщений
- Сеть коммутации пакетов

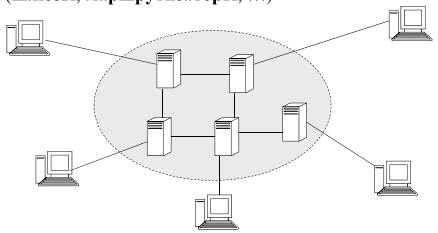
Сообщение — логически целостная порция данных, имеющая произвольный размер).

Взаимодействие абонентов осуществляется сеансами связи. **Сеанс связи** состоит из обмена сообщениями между абонентами. Начало/завершение сеанса связи.

Компьютерные сети

Составляющие компьютерной сети

- Абонентские или основные компьютеры — **хосты**
- Коммуникационные или вспомогательные компьютеры (**шлюзы**, **маршрутизаторы**, …)



Свойства компьютерных сетей

1. Большое число связанных узлов, обеспечивающих решение определённых задач
2. Возможность распределения обработки информации
3. Расширяемость сети (сеть должна обеспечивать развитие сети по протяжённости, по расширению пропускной способности каналов, по составу и производительности узлов, входящих в состав сети)
4. Применение симметричных интерфейсов обмена информации внутри сети (возможность произвольного распределеня функций внутри сети)

Компьютерные сети

Компьютерная сеть — объединение компьютеров (вычислительных систем), взаимодействующих через коммуникационную среду.

Коммуникационная среда — каналы и средства передачи данных.

Модель ISO/OSI организации взаимодействия в сети

7. Прикладной уровень
6. Представительский уровень
5. Сеансовый уровень
4. Транспортный уровень
3. Сетевой уровень
2. Канальный уровень
1. Физический уровень

Сеть коммутации пакетов

Каждое сообщение разбивается на блоки фиксированного размера - **пакеты**. Структура пакета (заголовок, данные)

Преимущества

- Так как известна топология сети и характеристики её элементов, то возможно определение требований в коммутационных узлах
  ⇒ возможна оценка размера буфера и времени доставки пакетов

Недостатки

- Увеличение трафика из-за наличия заголовочной информации
- Проблема сборки пакетов

Сеть коммутации сообщений

Взаимодействие представляется в виде последовательности обменов сообщениями.

Преимущества

- Отсутствие занятости канала на недетерминированный промежуток времени

Недостатки

- Сообщения могут быть произвольного размера - необходимость наличия средств буферизации неопределённых характеристик
- Необходимость в специализированном коммуникационном оборудовании и ПО
- Повтор передачи всего сообщения в случае сбоя

Сеть коммутации каналов

Сеть коммутации каналов обеспечивает выделение коммуникаций абонентам на весь сеанс связи.

Преимущества

- После установления соединения сеть находится в состоянии готовности
- Требования к коммуникационному оборудованию минимальны
- Минимизируются накладные расходы по передаче данных
- Детерминированная пропускная способность

Недостатки

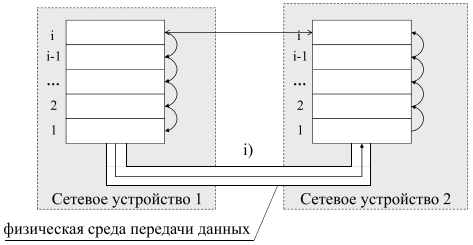
- Требование избыточности сети
- Период ожидания соединения (канала) недетерминирован
- Неэффективное использование выделенного канала
- В случае сбоя или отказа повторная передача информации

Соответствие модели ISO/OSI модели семейства протоколов TCP/IP

Уровень модели TCP/IP	Уровень модели ISO/OSI
1. Уровень доступа к сети Специфицирует доступ к физической сети.	Канальный уровень Физический уровень
2. Межсетевой уровень В отличие от сетевого уровня модели OSI, не устанавливает соединений с другими машинами.	Сетевой уровень

Семейство протоколов TCP/IP

Логическое взаимодействие сетевых устройств по i-ому протоколу



Основные понятия

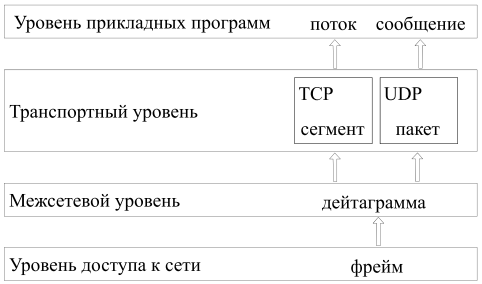
Протокол (правила взаимодействия одноименных уровней) — формальное описание сообщений и правил, по которым сетевые устройства (вычислительные системы) осуществляют обмен информацией. Правила взаимодействия одноимённых (одноранговых) уровней сети.

Интерфейс — правила взаимодействия вышестоящего уровня с нижестоящим.

Служба (сервис) — набор операций, предоставляемых нижестоящим уровнем вышестоящему.

Стек протоколов — перечень разноуровневых протоколов, реализованных в системе.

Взаимодействие между уровнями протоколов TCP/IP



Свойства протоколов семейства TCP/IP

- Открытые (доступные для использования) стандарты протоколов
- Независимость от аппаратного обеспечения сети передачи данных
- Общая схема именования сетевых устройств
- Стандартизованные протоколы прикладных программ

Соответствие модели ISO/OSI модели семейства протоколов TCP/IP

Уровень модели TCP/IP	Уровень модели ISO/OSI
4. Уровень прикладных программ Состоит из прикладных программ и процессов, использующих сеть и доступных пользователю. В отличие от модели OSI, прикладные программы сами стандартизуют представление данных.	Уровень прикладных программ Уровень представления данных

Соответствие модели ISO/OSI модели семейства протоколов TCP/IP

Уровень модели TCP/IP	Уровень модели ISO/OSI
3. Транспортный уровень Обеспечивает доставку данных от компьютера к компьютеру, обеспечивает средства для поддержки логических соединений между прикладными программами. В отличие от транспортного уровня модели OSI, в функции транспортного уровня TCP/IP не всегда входят контроль за ошибками и их коррекция. TCP/IP предоставляет два разных сервиса передачи данных на этом уровне. Протокол TCP, UDP.	Сеансовый уровень Транспортный уровень

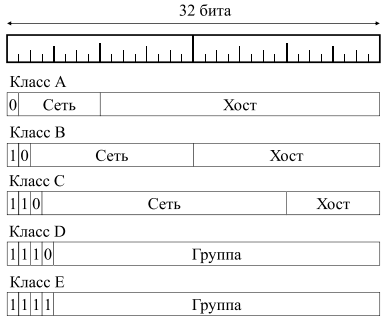
Дейтаграммы

Дейтаграмма — пакет протокола IP.

Шлюз — устройство, передающее пакеты между различными сетями.

Маршрутизация — процесс выбора шлюза или маршрутизатора.

Система адресации протокола IP



Межсетевой уровень. Протокол IP

- Функции протокола IP
  - формирование дейтаграмм
  - поддержание системы адресации
  - обмен данными между транспортным уровнем и уровнем доступа к сети
  - организация маршрутизации дейтаграмм
  - разбиение и обратная сборка дейтаграмм
- IP — протокол БЕЗ логического установления соединения
- Протокол IP НЕ обеспечивает обнаружение и исправление ошибок

Уровень доступа к сети

На уровне доступа к сети протоколы обеспечивают систему средствами для передачи данных другим устройствам в сети

Пример

Протокол Ethernet — разработка исследовательской компании Xerox (1976 год). Единая шина — широковещательная сеть. Для сетевых устройств обеспечивается множественный доступ, с контролем и обнаружений конфликтов (Carrier Sense Multiple Access with Collision Detection — CSMA/CD)

Операционные системы

Введение (часть 4)

Уровень прикладных программ

На основе TCP базируются прикладные протоколы, которые обеспечивают либо доступ и работу с заведомо корректной информацией, которая осуществляется в сети Интернет.

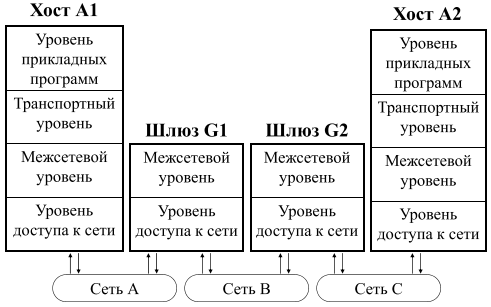
- Протоколы, опирающиеся на TCP
  - TELNET (Network Terminal Protocol)
  - FTP (File Transfer Protocol)
  - SMTP (Simple Mail Transfer Protocol)
- Протоколы, опирающиеся на UDP
  - DNS (Domain Name Service)
  - RIP (Routing Information Protocol)
  - NFS (Network File System)

Транспортный уровень

- **Протокол контроля передачи (TCP**, Transmission Control Protocol) обеспечивает надежную доставку данных с обнаружением и исправлением ошибок и с установлением логического соединения.
- **Протокол пользовательских дейтаграмм (UDP**, User Datagram Protocol) отправляет пакеты с данными, не контролируя их доставку.

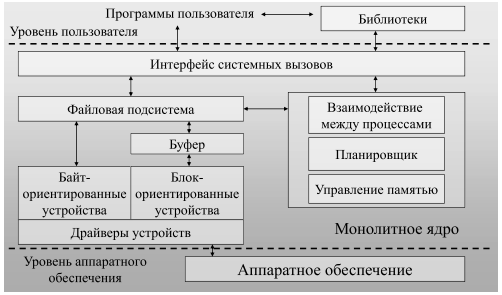
TCP обеспечивает последовательную передачу пакетов, контроль доставки пакетов, отработку сбоев.

Маршрутизация дейтаграмм



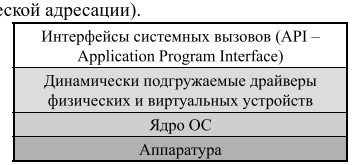
Структура ОС

Пример структурной организации классической системы UNIX



Структура ОС

Ядро (**Kernel**) — резидентная часть ОС, работающая в режиме супервизора (обычно работает в режиме физической адресации).



Динамически подгружаемые драйверы устройств:
- резидентные / нерезидентные
- работают в пользовательском / привилегированном режиме

Системный вызов — обращение к ОС за предоставление той или иной функции (возможности, услуги, сервиса).

Требования к ОС

- Надежность
  Количество ошибок должно быть минимизировано
- Защита
  Предусмотрение защиты информации и ресурсов от несанкционированного доступа
- Эффективность
  Удовлетворение критериям эффективности
- Предсказуемость
  Известны заранее проблемы и последствия различных действий, устойчивость к форс-мажору

Базовые понятия

Операционная система — комплекс программ, обеспечивающий контроль за существованием, распределением и использованием ресурсов ВС.

Процесс — совокупность машинных команд и данных, исполняющаяся в рамках ВС и обладающая правами на владение некоторым набором ресурсов.

Пакетная ОС

Переключение выполнения процессов происходит:

- выполнение процесса завершено
- возникло прерывание
- зацикливания процесса

Типы операционных систем

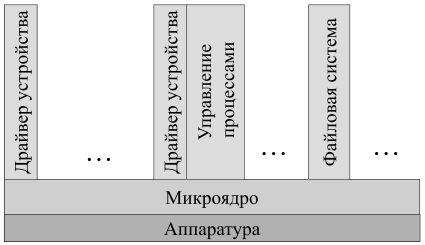
- Пакетная ОС
- Системы разделения времени
- ОС реального времени

Логические функции ОС

- Управление процессами
- Управление ОП
- Планирование
- Управление устройствами и ФС
- Сетевое взаимодействие
- Безопасность

Структура ОС

Микроядерная архитектура



Сетевые, распределенные ОС

Распределённая ОС — ОС, функционирующая на многопроцессорном/многомашинном комплексе, в котором на каждом из узлов функционирует своё ядро, а также система, обеспечивающая распределение возможностей (ресурсов) ОС.

Сетевые, распределенные ОС

Сетевая ОС — ОС, которая обеспечивает функции распределения приложений в сети

ОС реального времени

Системы реального времени являются специализированными системами, в которых все функции планирования ориентированы на обработку фиксированного набора событий за время, не превосходящее некоторого предельного значения.

Системы разделения времени

Квант времени ЦП — некоторый фиксированный ОС промежуток времени работы ЦП.

Переключение выполнения процессов происходит:

- исчерпался выделенный квант времени
- выполнение процесса завершено
- возникло прерывание
- зацикливания процесса

Работа с сигналами

```
#include <sys/types.h>
#include <signal.h>

int kill (pit_t pid, int sig);
```

pid – идентификатор процесса, которому посылается сигнал

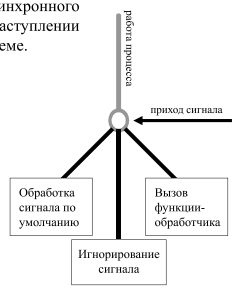
sig – номер посылаемого сигнала

При удачном выполнении возвращает 0, в противном случае возвращает -1

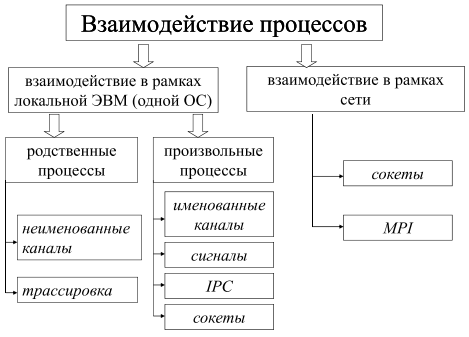
Сигналы

Сигнал – средство асинхронного уведомления процесса о наступлении некоторого события в системе.

Примеры сигналов <signal.h>
- SIGINT (2)
- SIGQUIT (3)
- SIGKILL (9)
- SIGALRM (14)
- SIGCHLD (18)



Взаимодействие процессов



Реализация взаимодействия процессов

Пример. Двухпроцессный вариант программы “будильник”.

```
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void alr(int s)
{
    printf(“\n Быстрее!!! \n”);
}

int main(int argc, char **argv)
{   char s[80];
    int pid;
    signal(SIGALRM, alr);
    if  (pid=fork()) {/*”отец”*/}
     else {/*”сын”*/}
    return 0;
}
```

Пример. Программа “будильник”.

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void alrm (int s)
{
  printf(“\n жду имя \n”);
    alarm(5);
}

int main(int argc, char **argv)
{  char s[80];
   signal(SIGALRM, alrm); alarm(5);
   printf(“Введите имя \n”);
   for (;;) {
         printf(“имя:”);
         if (gets(s) != NULL) break;
   };
   printf(“OK! \n”);
   return 0;
}
```

Пример. Обработка сигнала.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
int count=1;

int main(int argc, char **argv)
{  signal (SIGINT, SigHndlr);
   while (1);/*”тело программы”*/
   return 0;
}

  void SigHndlr (int s)
  {printf("\n I got SIGINT %d time(s) \n", count ++);
   if (count==5)
       signal (SIGINT, SIG_DFL);
  }
```

Работа с сигналами

```
#include <signal.h>
void (*signal  ( int sig, void (*disp) (int))) (int)
```

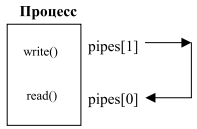
sig – номер сигнала, для которого устанавливается реакция

disp – либо определенная пользователем функция – обработчик сигнала, либо одна из констант:
SIG_DFL – обработка по умолчанию
SIG_IGN - игнорирование

При успешном завершении функция возвращает указатель на предыдущий обработчик данного сигнала.

Пример. Использование канала.

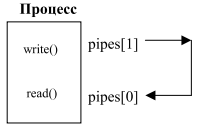
```
int main(int argc, char **argv)
{
   char *s=”chanel”;
   char buf[80];
   int pipes[2];
   pipe(pipes);
   write(pipes[1],s,strlen(s)+1);
   read(pipes[0],buf,strlen(s)+1);
   close(pipes[0]);
   close(pipes[1]);
   printf(“%s\n”,buf);
}
```



Неименованные каналы. Системный вызов pipe( )

#include <unistd.h>
int pipe (int *pipes);

pipes[1] – запись в канал
pipes[0] – чтение из канала



Неименованные каналы.

Пример. Двухпроцессный вариант программы “будильник”.

```
/*”отец”*/

for (;;) {
   sleep(5);
   kill(pid, SIGALRM);
}
```

```
/*”сын”*/
printf(“Введите имя \n”);
for (;;) {
   printf(“имя:”);
   if (gets(s) != NULL) break;
}
printf(“OK!\n”);
kill(getppid(), SIGKILL);
```

Пример. Совместное использование сигналов и каналов – «пинг-понг».

```
void SigHndlr (int s)
{
  if (cnt < MAX_CNT)
  {
      read(fd[0], &cnt, sizeof(int));
      printf("%d \n", cnt);
      cnt++;
      write(fd[1], &cnt, sizeof(int));
      kill(target_pid, SIGUSR1);
  } …
```

Пример. Совместное использование сигналов и каналов – «пинг-понг».

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

#define MAX_CNT 100
int target_pid, cnt;
int fd[2];
int status;
```

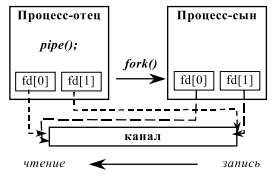
Пример. Реализация конвейера.

```
#include <stdio.h>
int main(int argc, char **argv)
{
  int fd[2];

  pipe(fd);
  if(fork()) {  dup2(fd[1],1);
                close(fd[1]);
                close(fd[0]);
                execl(“/usr/bin/print”,”print”,0);
                }
  dup2(fd[0],0);
  close(fd[0]);
  close(fd[1]);
  execl(“/usr/bin/wc”,”wc”,0);
}
```

Пример. Типовая схема взаимодействия процессов с использованием канала.

```
int main(int argc, char **argv)
{
   int fd[2];
   pipe(fd);
   if(fork()) {    close(fd[0]);
                   write (fd[1], …);
                   …
                   close(fd[1]);
                   …
              }
   else {close(fd[1]);
         while(read(fd[0],…)) {…}
         …
   }
}
```



Именованные каналы.

Пример. Совместное использование сигналов и каналов – «пинг-понг».

```
…
  } else { /* процесс – сын */
      read(fd[0], &cnt, sizeof(int)); /* старт синхр*/
      target_pid = getppid();
      write(fd[1], &cnt, sizeof(int));
      kill(target_pid, SIGUSR1);
      for(;;);
  }
}
```

Пример. Совместное использование сигналов и каналов – «пинг-понг».

```
int main(int argc, char **argv)
{  pipe(fd);
   signal(SIGUSR1, SigHndlr);
   cnt = 0;
   if (target_pid = fork()) { /* процесс – родитель*/
       write(fd[1], &cnt, sizeof(int));
       while(wait(&status)== -1);
       printf("Parent is going to be terminated\n");
       close(fd[1]);
       close(fd[0]);
       return 0; …
```

Пример. Совместное использование сигналов и каналов – «пинг-понг».

```
… else
   if (target_pid == getppid()) /* процесс – сын*/
   {
       printf("Child is going to   be terminated\n");
       close(fd[1]);
       close(fd[0]);
       exit(0);
   } else /* процесс – родитель */
       kill(target_pid, SIGUSR1);
}
```

Пример. «Клиент-сервер». Процесс-клиент:

```
#include <stdio.h>          #include <sys/stat.h>
#include <sys/types.h>      #include <sys/file.h>
```

```
int main(int argc, char **argv)
{
int fd;
int pid = getpid( );
fd = open ("fifo", O_RDWR);
write (fd, &pid, sizeof(int));
close (fd);
}
```

Пример. «Клиент-сервер». Процесс-сервер:

```
int main(int argc, char **argv)
{  int fd;
   int pid;
   mkfifo("fifo", FILE_MODE | 0666);
   fd = open ("fifo", O_RDONLY | O_NONBLOCK);
   while  ( read (fd, &pid, sizeof(int) ) != -1) {
        printf ("Server %d  got message from %d !\n",
                                              getpid(), pid);
   …
   }
   close (fd);
   unlink ("fifo");
}
```

Пример. «Клиент-сервер». Процесс-сервер:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
```

Именованные каналы. Создание.

```
int  mkfifo (char *pathname, mode_t mode);
```

pathname – имя создаваемого канала

mode – права доступа + режимы открытия

- блокировка при подключении
- использование флагов:
  - **O_RDONLY** открытие «на чтение»;
  - **O_RDWR** открытие «на чтение+запись»;
  - **O_NONBLOCK** – открытие без блокирования;
  - .....

Системный вызов ptrace()

```
#include <sys/ptrace.h>
int ptrace(int cmd, int pid, int addr, int data);
```

cmd=**PTRACE_TRACEME** вызывает сыновний процесс, позволяя трассировать себя

cmd=**PTRACE_PEEKDATA** чтение слова из адресного пространства отлаживаемого процесса

cmd=**PTRACE_PEEKUSER** чтение слова из контекста процесса (из пользовательской составляющей, содержащейся в <sys/user.h>)

cmd=**PTRACE_POKEDATA** запись данных в адресное пространство процесса-потомка

cmd=**PTRACE_POKEUSER** запись данных в контекст трассируемого процесса.

Главный - Подчиненный

int ptrace(int cmd, int pid, int addr, int data);

cmd – код команды:

• **группа команд чтения** (сегмент кода, сегмент данных, контекст процесса)

• **группа команд записи** (сегмент кода, сегмент данных, контекст процесса)

• **группа команд управления** (продолжить выполнение, продолжить выполнение с заданного адреса, включить «шаговый режим», завершить процесс, разрешить трассировку)

Главный - Подчиненный

```
#include <sys/ptrace.h>
```

int ptrace(int cmd, int pid, int addr, int data);

cmd – код выполняемой команды

pid – идентификатор процесса-потомка

addr – некоторый адрес в адресном пространстве процесса-потомка

data – слово информации.

Взаимодействие «главный-подчинённый».

Пример.

```
int main(int argc, char **argv)
{
   return argc/0;
}
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/wait.h>
```

Схема установки контрольной точки по адресу A_{BrPnt}

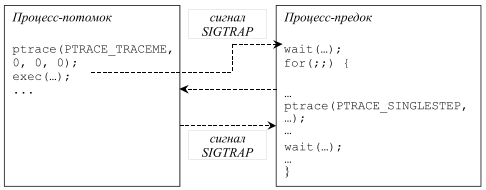
Установка контрольной точки
Статус отлаживаемого процесса (ОП) **ВЫПОЛНЕНИЕ**
→ послать **Sigtrap**
→ ждем остановку ОП + анализ точки остановки (статус ОП **ОЖИДАНИЕ**)
→ чтение в адресном пространстве ОП, сохранение (N_{BrPnt}, $< A_{BrPnt} >$)
→ запись **BrPnt** в A_{BrPnt}
→ продолжить с точки останова

Приход в контрольную точку
Статус (ОП) **ВЫПОЛНЕНИЕ**
→ ждем остановки ОП, остановка (статус ОП **ОЖИДАНИЕ**)
→ чтение информации из контекста, анализ точки остановки
→ контрольная точка (совпадение адреса остановки + причины остановки)
→ действия по отладке ОП в состоянии **ОЖИДАНИЯ**
→ .....

Снятие контрольной точки
Статус (ОП) **ОЖИДАНИЕ**
→ восстанавливаем содержимое A_{BrPnt} (N_{BrPnt}, $< A_{BrPnt} >$)
→ продолжить с адреса A_{BrPnt}

«Движение» через контрольную точку
Статус (ОП) **ОЖИДАНИЕ**
→ восстанавливаем содержимое A_{BrPnt} (N_{BrPnt}, $< A_{BrPnt} >$)
→ включаем «шаговый» режим
→ продолжить с адреса A_{BrPnt}
→ ждем остановки ОП (анализ точки остановки)
→ запись **BrPnt** в A_{BrPnt}
→ продолжаем с точки остановки
→ .....

Общая схема трассировки процессов



Системный вызов ptrace()

```
#include <sys/ptrace.h>
int ptrace(int cmd, int pid, int addr, int data);
```

cmd=**PTRACE_GETREGS,PTRACE_GETFREGS** чтение регистров общего назначения

cmd=**PTRACE_SETREGS,PTRACE_SETFREGS** запись в регистры общего назначения

cmd=**PTRACE_CONT** возобновление выполнения трассируемого процесса

cmd=**PTRACE_SYSCALL, PTRACE_SINGLESTEP** возобновляется выполнение трассируемой программы, но снова останавливается после выполнения одной инструкции

cmd=**PTRACE_KILL** завершение выполнения трассируемого процесса

Общие концепции

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

key_t ftok (char * filename, char proj)

Параметры

filename — строка, содержащая имя файла

proj — добавочный символ

Система межпроцессного взаимодействия IPC

Пример.

```
…
  while (1) {
      wait( &status );
      ptrace(PTRACE_GETREGS, pid, &REG, &REG);
      printf("signal = %d, status = %#x, EIP=%#x,
              ESP=%#x\n“ , WSTOPSIG(status), status,
      REG.eip, REG.esp);
      if (WSTOPSIG(status) != SIGTRAP) {
              if (!WIFEXITED(status))
                      ptrace (PTRACE_KILL, pid, 0, 0);
              break;
      }
      ptrace (PTRACE_CONT, pid, 0, 0);
  }
}
```

Пример.

```
int main(int argc, char *argv[])
{
   pid_t pid;
   int status;
   struct user_regs_struct REG;
   if  ((pid = fork()) == 0) {
        ptrace(PTRACE_TRACEME, 0, 0, 0);
        execl(“son”, ”son”, 0);
   }
   …
```

Создание/доступ к очереди сообщений

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/message.h>

int msgget (key_t key, int msgflag)

Параметры

key — уникальный идентификатор ресурса

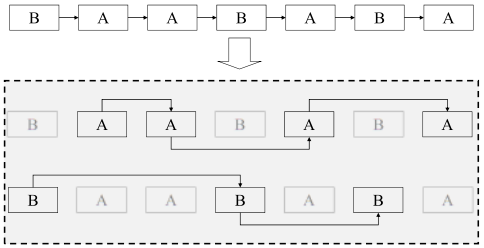
msgflag — флаги, управляющие поведением вызова

Возвращаемое значение

В случае успеха возвращается положительный дескриптор очереди, в случае неудачи возвращается –1.

Очередь сообщений

- Организация очереди сообщений по принципу **FIFO**
- Использование типов сообщений



IPC: очередь сообщений

1. Общие концепции
2. Создание/доступ к очереди сообщений
3. Отправка сообщений
4. Получение сообщений
5. Управление очередью сообщений
6. Пример. Использование очереди сообщений
7. Пример. Очередь сообщений .Модель «клиент-сервер»

Общие концепции

- **<ResName>get** (key, …, flags) — создание/подключение
- Флаги создания/подключения:
  - **IPC_PRIVATE** (доступность только породившему процессу)
  - **IPC_CREAT** (создать новый или подключиться к существующему)
  - **IPC_EXCL** (+ IPC_CREAT создание только нового)
  - **…**

Получение сообщений

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgrcv (int msqid, void * msgp, size_t msgsz, long msgtyp, int msgflg)

Параметры

msgflg — побитовое сложение флагов

- IPC_NOWAIT — если сообщения в очереди нет, то возврат –1
- MSG_NOERROR — разрешение получать сообщение, даже если его длина превышает емкость буфера

Возвращаемое значение

В случае успеха возвращает количество прочитанных байтов в теле сообщения.

Получение сообщений

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgrcv (int msqid, void * msgp, size_t msgsz, long msgtyp, int msgflg)

Параметры

msqid — дескриптор очереди

msgp — указатель на буфер

msgsz — размер тела сообщения

msgtyp — тип сообщения, которое процесс желает получить

Отправка сообщений

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd (int msqid, const void * msgp, size_t msgsz, int msgflg)

Параметры

msgsz — размер тела сообщения

msgflg

- 0 процесс блокируется, если для посылки сообщения недостаточно системных ресурсов
- **IPC_NOWAIT** – работа без блокировки (возврат –1)

Отправка сообщений

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgsnd (int msqid, const void * msgp, size_t msgsz, int msgflg)

Параметры

msqid — дескриптор очереди, полученный в результате вызова msgget()

msgp — указатель на буфер:

- **long msgtype —**тип сообщения
- **char msgtext[] —** данные (тело сообщения)

Использование очереди сообщений

Основной процесс

```
...
switch ( str [ 0 ] ) {
   case 'a' : case 'A' :
      Message . mtype = 1 ;
      msgsnd ( msgid, ( struct
         msgbuf * ) ( & Message ),
         strlen ( str ) + 1, 0 ) ;
      break ;
   case 'b' : case 'B' :
      Message . mtype = 2 ;
      msgsnd(msgid, ( struct
         msgbuf * ) ( & Message ),
         strlen ( str ) + 1, 0 ) ;
      break ;
   case 'q' : case 'Q' :
      Message . mtype = 1 ;
      msgsnd ( msgid, ( struct msgbuf * )
         ( & Message ), strlen ( str ) + 1, 0 ) ;
      Message . mtype = 2 ;
      msgsnd ( msgid, ( struct msgbuf * )
         ( & Message ), strlen ( str ) + 1, 0 ) ;
      sleep ( 10 ) ;
      msgctl ( msgid, IPC_RMID, NULL ) ;
      exit ( 0 ) ;
   default :
      break ;
   }
}
```

Использование очереди сообщений

Основной процесс

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
#include <stdio.h>
struct
{
   long mtype ;
   char Data [ 256 ] ;
} Message ;

int main ( int argc, char ** argv )
{
   key_t key;
   int msgid;
   char str [ 256 ] ;

   key = ftok ( "/usr/mash", 's' ) ;
   msgid = msgget ( key, 0666 |
      IPC_CREAT | IPC_EXCL ) ;
   for ( ; ; )
   {
      gets ( str ) ;
      strcpy ( Message . Data, str ) ;
      ...
```

Управление очередью сообщений

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msgid_ds *buf)

Параметры

buf — структура, описывающая параметры очереди. Тип **msgid_ds** описан в заголовочном файле <sys/message.h>, и представляет собой структуру, в полях которой хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возвращаемое значение

В случае успеха возвращается 0.

Управление очередью сообщений

Необходимые заголовочные файлы и прототип

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msgid_ds * buf)

Параметры

msgid — дескриптор очереди

cmd — команда

- **IPC_STAT** — скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре **buf**
- **IPC_SET** — заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре **buf**
- **IPC_RMID** — удалить очередь

IPC: разделяемая память

Создание общей памяти

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

int shmget (key_t key, int size, int shmflg)

Параметры

key — ключ для доступа к разделяемой памяти

size — размер области памяти

shmflg — флаги управляющие поведением вызова

Возвращаемое значение

дескриптор области памяти, в случае неудачи — –1.

Пример: «Клиент-сервер»

Клиент

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int main ( int argc, char ** argv )
{
   struct {
      long mestype;
      long mes;
   } messageto;
   struct {
      long mestype;
      char mes[100];
   } messagefrom;
   key_t key;
   int mesid;
   long pid = getpid () ;
   key = ftok ( "example", 'r' ) ;
   mesid = msgget ( key, 0666 ) ;
   messageto . mestype = 1 ;
   messageto . mes = pid ;
   msgsnd (mesid, & messageto, sizeof
      (messageto) – sizeof ( long ), 0 ) ;
   msgrcv ( mesid, & messagefrom, sizeof (
      messagefrom ) – sizeof ( long ), pid, 0 ) ;
   printf ( "%s", messagefrom . mes ) ;
   return 0 ;
}
```

Пример: «Клиент-сервер»

Сервер

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>
int main ( int argc, char ** argv )
{   struct {
      long mestype ;
      char mes [ 100 ] ;
   } messageto ;
   struct {
      long mestype ;
      long mes ;
   } messagefrom ;
   key_t key ;
   int mesid ;
   key = ftok ( "example", 'r' ) ;
   mesid = msgget ( key, 0666 | IPC_CREAT |
      IPC_EXCL ) ;
   while ( 1 )
   {
      msgrcv ( mesid, & messagefrom, sizeof (
         messagefrom ) – sizeof ( long ), 1, 0 ) ;
      messageto . mestype = messagefrom . mes ;
      strcpy ( messageto . mes, "Message for client" ) ;
      msgsnd ( mesid, & messageto, sizeof (messageto
         ) – sizeof ( long ),      0 ) ;
   }
   return 0 ;
   }
```

Использование очереди сообщений

Процесс-приемник A

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
#include <stdio.h>
struct {
   long mtype ;
   char Data [ 256 ] ;
} Message ;

int main ( int argc, char ** argv )
{
   key_t key ;
   int msgid ;
   key = ftok ( "/usr/mash", 's' ) ;
   msgid = msgget ( key, 0666 ) ;
   for ( ; ; ) {
      msgrcv ( msgid, ( struct msgbuf * )
         ( & Message ), 256, 1, 0 ) ;
      printf ( "%s", Message . Data ) ;
      if ( Message . Data [ 0 ] == 'q' || Message .
         Data [ 0 ] == 'Q' )
         break ;
   }
   exit () ; }
```

Пример. Работа с общей памятью в рамках одного процесса

```
int main ( int argc, char ** argv )
{
    key_t key;
    char * shmaddr ;

    key = ftok ( "/tmp/ter", 'S' ) ;
    shmid = shmget ( key, 100, 0666 | IPC_CREAT | IPC_EXCL ) ;
    shmaddr = shmat ( shmid, NULL, 0 ) ;
    putm ( shmaddr ) ;
    .....
    shmctl ( shmid, IPC_RMID, NULL ) ;
    exit () ;
}
```

Управление разделяемой памятью

Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl ( int shmid, int cmd, struct shmid_ds * buf )
```

Параметры

shmid — дескриптор области памяти

cmd:

- **IPC_STAT** — скопировать структуру, описывающую управляющие параметры области памяти
- **IPC_SET** — заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре **buf**.
- **IPC_RMID** удалить
- **SHM_LOCK, SHM_UNLOCK** — блокировать или разблокировать область памяти.

buf — структура, описывающая управляющие параметры области памяти.

Отключение от разделяемой памяти

Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt ( char * shmaddr )
```

Параметры

shmaddr — адрес прикрепленной к процессу памяти, который был получен при вызове shmat()

Возвращаемое значение

В случае успешного выполнения функция возвращает 0, в случае неудачи — –1.

Доступ к разделяемой памяти

Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char * shmat ( int shmid, char * shmaddr, int shmflg )
```

Параметры

shmid — дескриптор области памяти

shmaddr — «адрес подключения» - адрес, начиная с которого необходимо подсоединить разделяемую память (>0 или =0)

shmflg — флаги, например, **SHM_RDONLY** подсоединяемая область будет использоваться только для чтения

Возвращаемое значение

Возвращает адрес, начиная с которого будет отображаться присоединяемая разделяемая память. При неудаче возвращается –1.

Операции над семафором

Значение семафора с номером **num** равно **val**.

```
struct sembuf
{   short sem_num ;   /* номер семафора    */
    short sem_op ;    /* операция          */
    short sem_flg ;   /* флаги операции    */
}
```

Если sem_op ≠ 0 **то**
    пока (val+sem_op < 0) [процесс блокирован пока]
    val=val+sem_op
Если sem_op = 0 **то**
    пока (val ≠ 0) [процесс блокирован]
    [возврат из вызова]

Операции над семафором

Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop ( int semid, struct sembuf  *cmd_buf , size_t nops )
```

Параметры

semid — дескриптор ресурса

cmd_buf — массив из элементов типа **sembuf**

nops — количество элементов в массиве **cmd_buf**

Создание/доступ к семафору

Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget ( key_t key, int nsems, int semflag )
```

Параметры

key — уникальный идентификатор ресурса

nsems — количество семафоров

semflag — флаги

Возвращаемое значение

Возвращает целочисленный дескриптор созданного разделяемого ресурса, либо -1, если ресурс не удалось создать.

IPC: массив семафоров

Использование разделяемой памяти и семафоров

1-ый процесс

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX 256

int main ( int argc, char ** argv )
{
    key_t key;
    int semid, shmid ;
    struct sembuf sops ;
    char * shmaddr ;
    char str [ NMAX ] ;
    key = ftok ( "/usr/ter/exmpl", 'S' ) ;
    semid = semget ( key, 1, 0666 | IPC_CREAT |
        IPC_EXCL ) ;
    shmid = shmget ( key, NMAX, 0666 |
        IPC_CREAT | IPC_EXCL ) ;
    shmaddr = shmat(shmid, NULL, 0 ) ;
```

Пример. Использование разделяемой памяти и семафоров

Управление массивом семафоров

Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int semid, int num, int cmd, union  semun arg)
```

```
<sys/sem.h>
union semun
{
   int val ;  /* значение одного семафора */
   struct semid_ds * buf ; /* параметры массива семафоров в
       целом  (количество, права доступа, статистика доступа)*/
   unsigned short * array ;     /* массив значений семафоров */
}
```

Управление массивом семафоров

Необходимые заголовочные файлы и прототип

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int semid, int num, int cmd, union  semun arg)
```

Параметры

semid — дескриптор массива семафоров

num — номер семафора в массиве

cmd — операция

- **IPC_SET** изменить значение, параметры семафора
- **IPC_RMID** удалить массив семафоров
- и др.

arg — управляющие параметры

Возвращаемое значение

Возвращает значение, соответствующее выполнявшейся операции (по умолчанию 0), в случае неудачи — –1

Использование разделяемой памяти и семафоров

2-ой процесс

```
    sops.sem_flg = 0;
    do  {
        printf ( "Waiting… \n" ) ;
        sops . sem_op = –2 ;
        semop ( semid, & sops, 1 ) ;
        strcpy ( str, shmaddr ) ;
        if ( str [ 0 ] == 'Q' )
            shmdt ( shmaddr ) ;
        sops . sem_op = –1 ;
        semop ( semid, & sops, 1 ) ;
        printf ( "Read from shared memory: %s\n", str ) ;
    } while ( str [ 0 ] != 'Q' ) ;
    return 0 ;
}
```

Использование разделяемой памяти и семафоров

2-ой процесс

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX 256

int main ( int argc, char ** argv )
{
    key_t key ;
    int semid, shmid ;
    struct sembuf sops ;
    char * shmaddr ;
    char str [ NMAX ] ;
    key = ftok ( "/usr/ter/exmpl", 'S' ) ;
    semid = semget ( key, 1, 0666 ) ;
    shmid = shmget ( key, NMAX, 0666 ) ;
    shmaddr = shmat ( shmid, NULL, 0 ) ;
    sops.sem_num = 0 ;
```

Использование разделяемой памяти и семафоров

1-ый процесс

```
…
semctl ( semid, 0, SETVAL, (int) 0 ) ;
sops . sem_num = 0 ;
sops . sem_flg = 0 ;
do
{
    printf( "Введите строку:" ) ;
    if ( fgets ( str, NMAX, stdin ) ==
        NULL )    strcpy ( str, "Q" ) ;
    strcpy ( shmaddr, str ) ;
    sops . sem_op = 3 ;
    semop ( semid, & sops, 1 ) ;
    sops . sem_op = 0 ;
    semop ( semid, & sops, 1 ) ;
}
while ( str [ 0 ] != 'Q' ) ;
shmdt ( shmaddr ) ;
shmctl ( shmid, IPC_RMID,
            NULL);
semctl ( semid, 0, IPC_RMID, (int) 0) ;
return 0 ;
}
```

Взаимодействие процессов: сокеты

Связывание

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int bind (int sockfd, struct sockaddr * myaddr, int addrlen) ;

Параметры

addrlen — размер структуры **sockaddr («доменный» адрес сокета**).

Возвращаемое значение

В случае успешного связывания **bind** возвращает 0, в случае ошибки — –1.

Связывание

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int bind (int sockfd, struct sockaddr * myaddr, int addrlen) ;

Параметры

sockfd — дескриптор сокета

myaddr — указатель на структуру, содержащую адрес сокета

Структура адреса для домена **AF_INET**

```
#include <netinet/in.h>
struct sockaddr_in {
  short sin_family ;          /* == AF_INET */
  u_short sin_port ;          /* port number */
  struct in_addr sin_addr ;   /* host IP address */
  char sin_zero [ 8 ] ;       /* not used */
} ;
```

Связывание

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int bind (int sockfd, struct sockaddr * myaddr, int addrlen) ;

Параметры

sockfd — дескриптор сокета

myaddr — указатель на структуру, содержащую адрес сокета

Структура адреса для домена **AF_UNIX**

```
#include <sys/un.h>
struct sockaddr_un {
  short sun_family ;          /* == AF_UNIX */
  char sun_path [ 108 ] ;
} ;
```

Создание сокета

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol) ;

Параметры

domain — коммуникационный домен:
- **AF_UNIX**
- **AF_INET**

type — тип сокета:
- **SOCK_STREAM** — виртуальный канал
- **SOCK_DGRAM** — датаграммы

protocol — протокол:
- **0** — автоматический выбор протокола
- **IPPROTO_TCP** — протокол TCP (AF_INET)
- **IPPROTO_UDP** — протокол UDP (AF_INET)

Подтверждение соединения

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr * addr, int * addrlen) ;

Параметры

sockfd — дескриптор сокета

addr — указатель на структуру, в которой возвращается адрес клиентского сокета, с которым установлено соединение (если адрес клиента не интересует, передается NULL).

addrlen — возвращается реальная длина этой структуры. максимальный размер очереди запросов на соединение.

Возвращаемое значение

- дескриптор нового сокета, соединенного с сокетом клиентского процесса.

Запрос на соединение

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr * serv_addr, int addrlen) ;

Параметры

sockfd — дескриптор сокета

serv_addr — указатель на структуру, содержащую адрес сокета, с которым производится соединение

addrlen — реальная длина структуры

Возвращаемое значение

В случае успешного связывания функция возвращает 0, в случае ошибки — –1. Код ошибки заносится в **errno**.

Прослушивание сокета

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int listen (int sockfd, int backlog);

Параметры

sockfd — дескриптор сокета

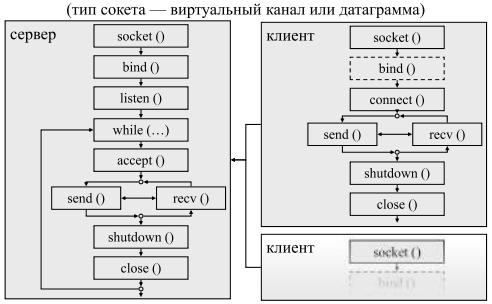
backlog — максимальный размер очереди запросов на соединение

Возвращаемое значение

В случае успешного обращения функция возвращает 0, в случае ошибки — –1. Код ошибки заносится в **errno**.

Предварительное установление соединения

(тип сокета — виртуальный канал или датаграмма)



Прием и передача данных

- **Read()**
- **Write()**

Прием и передача данных

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>

int send (int sockfd, const void * msg, int len, unsigned int flags) ;	**int recv (int sockfd, void * buf, int len, unsigned int flags) ;**

Возвращаемое значение

Функция возвращает количество переданных байт в случае успеха и –1 в случае неудачи. Код ошибки при этом устанавливается в **errno**.	В случае успеха функция возвращает количество считанных байт, в случае неудачи –1.

Прием и передача данных

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>

int send (int sockfd, const void * msg, int len, unsigned int flags) ;	**int recv (int sockfd, void * buf, int len, unsigned int flags) ;**

Параметры

flags — может содержать комбинацию специальных опций.

MSG_OOB — флаг сообщает ОС, что процесс хочет осуществить прием/передачу экстренных сообщений.

MSG_PEEK — При вызове **recv ()** процесс может прочитать порцию данных, не удаляя ее из сокета. Последующий вызов recv вновь вернет те же самые данные.

Прием и передача данных

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>

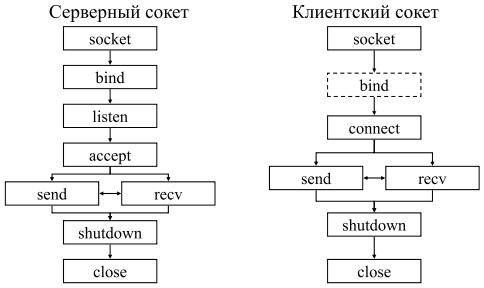
int send (int sockfd, const void * msg, int len, unsigned int flags) ;	**int recv (int sockfd, void * buf, int len, unsigned int flags) ;**

Параметры

sockfd — дескриптор сокета, через который передаются данные

msg — сообщение	**buf** — указатель на буфер для приема данных
len — длина сообщения	**len** — первоначальная длина буфера

Схема работы с сокетами с установлением соединения



Завершение работы с сокетом

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>
int shutdown (int sockfd, int mode) ;
(закрытие соединения)
int close (int fd) ;
(закрытие сокета)

Параметры

sockfd — дескриптор сокета

mode — режим закрытия соединения

= 0, сокет закрывается для чтения

= 1, сокет закрывается для записи

= 2, сокет закрывается и для чтения, и для записи

Прием и передача данных

Необходимые заголовочные файлы и прототип

#include <sys/types.h>
#include <sys/socket.h>

int sendto (int sockfd, const void * msg, int len, unsigned int flags, const struct sockaddr * to, int tolen) ;

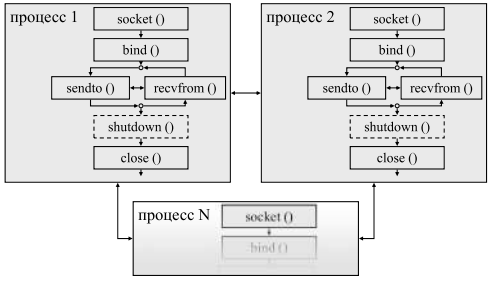
- Такие же, как и у рассмотренных раньше
- Такие же, как и у рассмотренных раньше
- указатель на структуру, содержащую адрес получателя
- размер структуры to

int recvfrom (int sockfd, void * buf, int len, unsigned int flags, struct sockaddr * from, int * fromlen) ;

- Такие же, как и у рассмотренных раньше
- указатель на структуру с адресом отправителя
- размер структуры from

Сокеты без предварительного соединения

(тип сокета — датаграмма)



```
…
unlink ( own_addr . sun_path ) ;            /* связываем сокет */
if ( bind ( sockfd, ( struct sockaddr * ) & own_addr, sizeof (
    own_addr . sun_family ) + strlen ( own_addr . sun_path ) ) < 0 )
{
  printf ( “can't bind socket!” ) ;
  return 0 ;
}
if ( ! is_server ) {                        /* это — клиент  */
  memset ( & party_addr, 0, sizeof ( party_addr ) ) ;
  party_addr . sun_family = AF_UNIX ;
  strcpy ( party_addr . sun_path, SADDRESS ) ;
  printf ( “type the string: ” ) ;
  …
```

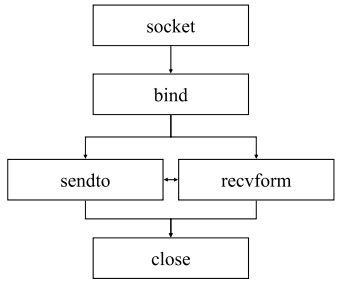
```
int main ( int argc, char ** argv)
{
 struct sockaddr_un party_addr,
    own_addr ;
 int sockfd ;
 int is_server ;
 char buf [ BUFLEN ] ;
 int party_len ;
 int quitting ;
 if (argc != 2) {
   printf( "Usage: %s
    client|server.\n", argv [ 0 ] ) ;
   return 0 ;
 }
 …
```

```
…
quitting = 1;
is_server = ! strcmp ( argv [ 1 ] ,
   “server” ) ;
memset ( & own_addr, 0, sizeof (
   own_addr ) ) ;
own_addr . sun_family =
   AF_UNIX ;
strcpy ( own_addr . sun_path,
   is_server ? SADDRESS :
   CADDRESS ) ;
if ( ( sockfd = socket ( AF_UNIX,
   SOCK_DGRAM, 0 ) ) < 0 ) {
  printf ( “can't create socket\n” ) ;
  return 0 ;
}
…
```

Пример. Работа с локальными сокетами
AF_UNIX

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#include <string.h>
#define SADDRESS "mysocket"
#define CADDRESS "clientsocket"
#define BUFLEN  40
```

Схема работы с сокетами без установления соединения




```
  ...
  /* посылаем ответ */
  if ( sendto ( sockfd, buf, strlen ( buf ) + 1, 0, ( struct sockaddr * ) &
    party_addr, party_len ) != strlen ( buf ) + 1 )
  {
    printf ( “server: error writing socket!\n” ) ;
    return 0 ;
  }
  if ( quitting )
    break ;
 }  /* while ( 1 ) */
 close ( sockfd ) ;
 return 0 ;
}
```

```
...
while ( 1 ) { /* получаем строку от клиента и выводим на печать */
  party_len = sizeof ( party_addr ) ;
  if ( recvfrom ( sockfd, buf, BUFLEN, 0, ( struct sockaddr * ) &
    party_addr, & party_len ) < 0 )
  {
    printf ( “server: error reading socket!” ) ;
    return 0 ;
  }
  printf ( “server: received from client: %s \n”, buf ) ;
  /* не пора ли выходить? */
  quitting = ( ! strcmp ( buf, “quit” ) ) ;
  if ( quitting ) strcpy ( buf, “quitting now!” ) ;
  else
    if ( ! strcmp ( buf, “ping!” ) ) strcpy ( buf, “pong!” ) ;
    else strcpy ( buf, “wrong string!” ) ;
  …
```

```
  …
  printf ( "client: server answered: %s\n", buf ) ;
  if ( quitting )
    break ;
  printf ( “type the string: ” ) ;
 }                                          /* while */
 close ( sockfd ) ;
 return 0 ;
}                                           /* if (!is_server), клиент */
…
```

```
…
while ( gets ( buf ) ) {                    /* не пора ли выходить? */
 quitting = ( ! strcmp ( buf, “quit” ) ) ;
  /* считали строку и передаем ее серверу */
 if ( sendto ( sockfd, buf, strlen ( buf ) + 1, 0, ( struct sockaddr * )
    & party_addr, sizeof ( party_addr . sun_family ) + strlen (
    SADDRESS ) ) != strlen ( buf ) + 1 ) {
  printf ( “client: error writing socket!\n” ) ;
  return 0 ;
 }
 if ( recvfrom ( sockfd, buf, BUFLEN, 0, NULL, 0 ) <0 ) {
  printf ( "client: error reading socket!\n“ ) ;
  return 0 ;
 }
 …
```

```
…
while ( 1 ) {
  memset ( & party_addr, 0, sizeof ( party_addr ) ) ;
  party_len = sizeof ( party_addr ) ; /* создаем соединение */
  if ( ( newsockfd = accept ( sockfd, ( struct sockaddr * ) & party_addr,
    & party_len ) ) < 0 ) {
    printf ( “error accepting connection!” ) ;
    return 0 ;
  }
  if ( ! fork () ) {
    /*это — сын, он обрабатывает запрос и посылает ответ*/
    close ( sockfd ) ; /*  этот сокет сыну не нужен */
    if ( ( len = recv ( newsockfd, & buf, BUFLEN, 0 ) ) < 0 ) {
      printf ( “error reading socket!” ) ;
      return 0 ;
    }
    …
```

```
…
/* связываем сокет */
memset ( & own_addr, 0, sizeof ( own_addr ) ) ;
own_addr . sin_family = AF_INET ;
own_addr . sin_addr . s_addr = INADDR_ANY ;
own_addr . sin_port = htons ( PORTNUM ) ;
if ( bind ( sockfd, ( struct sockaddr * ) & own_addr, sizeof ( own_addr
    ) ) < 0 ) {
  printf ( “can't bind socket!” ) ;
  return 0 ;
}
/* начинаем обработку запросов на соединение */
if ( listen ( sockfd, BACKLOG ) < 0 ) {
  printf ( “can't listen socket!” ) ;
  return 0 ;
}
…
```

```
int main ( int argc, char ** argv )
{
 struct sockaddr_in own_addr, party_addr ;
 int sockfd, newsockfd, filefd ;
 int party_len ;
 char buf [ BUFLEN ] ;
 int len ;
 int i ;
 /* создаем сокет */
 if ( ( sockfd = socket ( AF_INET, SOCK_STREAM, 0 ) ) < 0 )
 {
  printf ( “can't create socket\n” ) ;
  return 0 ;
 }
 …
```

Пример. Работа с локальными сокетами
AF_INET (`GET /<имя файла>`)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#define PORTNUM 8080
#define BACKLOG 5
#define BUFLEN 80
#define FNFSTR "404 Error
File Not Found "
#define BRSTR "Bad
Request "
```

Управление процессами

```
    …
    /* читаем из файла порции данных и посылаем их клиенту */
    while ( len = read ( filefd, & buf, BUFLEN ) )
      if ( send ( newsockfd, buf, len, 0 ) < 0 ) {
        printf ( “error writing socket!” ) ;
        return 0 ;
      }
    close ( filefd ) ;
    shutdown ( newsockfd, 1 ) ;
    close ( newsockfd ) ;
    return 0 ;
  } /* процесс — отец. Он закрывает новый сокет и продолжает
       прослушивать старый */
  close ( newsockfd ) ;
} /* while (1) */
}
```

```
…
for ( i =5; buf [ i ] && ( buf [ i ] > ‘ ’ ) ; i++ ) ;
buf [ i ] = 0 ;
/* открываем файл */
if ( ( filefd = open ( buf + 5, O_RDONLY ) ) < 0) {
  /* нет файла! */
  if ( send ( newsockfd, FNFSTR, strlen ( FNFSTR ) + 1, 0) !=
      strlen ( FNFSTR ) + 1 ) {
    printf ( “error writing socket!” ) ;
    return 0 ;
  }
  shutdown ( newsockfd, 1 ) ;
  close ( newsockfd ) ;
  return 0 ;
}
…
```

```
…
/* разбираем текст запроса */
printf ( "received: %s \n", buf ) ;
if ( strncmp ( buf, "GET /", 5 ) ) { /* плохой запрос! */
  if ( send ( newsockfd, BRSTR, strlen ( BRSTR ) + 1, 0 ) != strlen
      ( BRSTR ) + 1 ) {
    printf ( “error writing socket!” ) ;
    return 0 ;
  }
  shutdown ( newsockfd, 1 ) ;
  close ( newsockfd ) ;
  return 0 ;
}
…
```

Модель пакетной однопроцессной системы

0. Поступление процесса в очередь на начало обработки ЦП (процесс попадает в БВП)
1. Начало обработки процесса на ЦП (из БВП в БОП)
2. Завершение выполнения процесса, освобождение системных ресурсов.

Модельная ОС

1. **Буфер ввода процессов (БВП)** — пространство, в котором размещаются и хранятся сформированные процессы с момента их образования, до момента начала выполнения.
2. **Буфер обрабатываемых процессов (БОП)** — буфер для размещения процессов, находящихся в системе в мультипрограммной обработке.

Жизненный цикл процесса

Основной из задач ОС является поддержание жизненного цикла процесса.

Типовые этапы обработки процесса в системе

Жизненный цикл процесса в системе:

- Образование (порождение) процесса
- Обработка (выполнение) процесса
- Ожидание (по тем или иным причинам) постановки на выполнение
- Завершение процесса

Определение процесса. Основные понятия

Процесс — совокупность машинных команд и данных, которая обрабатывается в рамках вычислительной системы и обладает правами на владение некоторым набором *ресурсов*.

Ресурсы могут принадлежать только одному процессу, либо ресурсы могут разделяться между процессами — **разделяемые ресурсы**.

Выделение ресурсов процессу

- предварительная декларация — до начала выполнения
- динамическое пополнение списка принадлежащих процессу ресурсов

Количество допустимых процессов в системе — *ресурс ВС.*

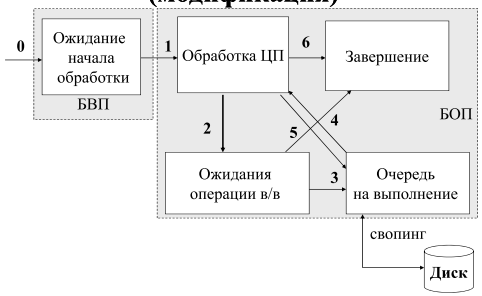
Типы процессов

- **«полновесные» процессы**
- **«легковесные» процессы**

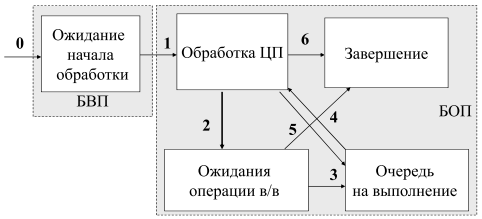
«Полновесные процессы» — процессы, выполняющиеся внутри защищенных участков оперативной памяти.

«Легковесные процессы» (нити) — работают в мультипрограммном режиме одновременно с активировавшей их задачей и используют ее виртуальное адресное пространство.

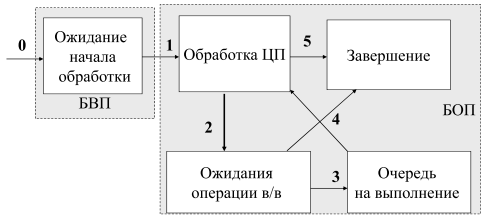
Модель ОС с разделением времени (модификация)



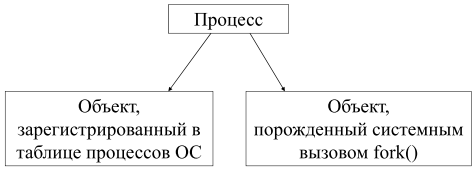
Модель ОС с разделением времени



Модель пакетной мультипроцессной системы



Определение процесса Unix



Контекст процесса

Контекст процесса — совокупность данных, характеризующих актуальное состояние процесса.

- Пользовательская составляющая — текущее состояние программы (совокупность машинных команд и данных, размещенных в ОЗУ)
- Системно-аппаратная составляющая
  - информация идентификационного характера (PID процесса, PID «родителя»…)
  - информация о содержимом регистров, настройках аппаратных интерфейсов, режимах работы процессора и т.п.
  - информация, необходимая для управления процессом (состояние процесса, приоритет).

Понятие «процесс»

Понятие «**процесс**» включает в себя следующее:

- исполняемый код
- собственное адресное пространство, которое представляет собой совокупность виртуальных адресов, которые может использовать процесс
- ресурсы системы, которые назначены процессу ОС
- хотя бы одну выполняемую нить

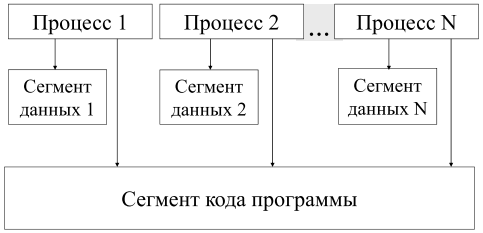
Типы процессов

Однонитевая организация процесса — «один процесс — одна нить»:

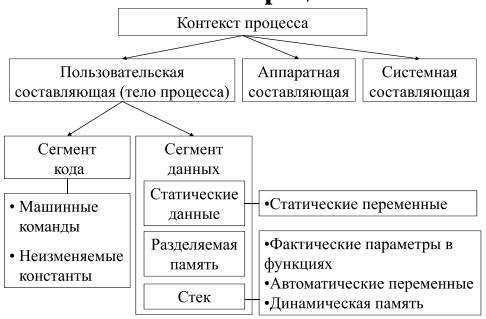
Многонитевая организация процесса:

Контекст процесса

Разделение сегмента кода



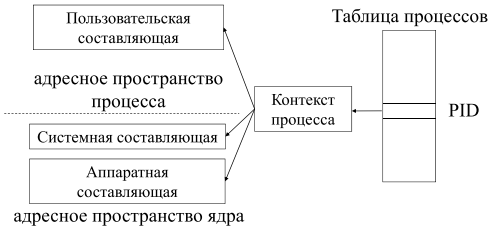
Контекст процесса



Определение процесса в UNIX

Процесс в UNIX — объект, зарегистрированный в таблице процессов UNIX.

Идентификатор процесса (PID)



Создание нового процесса

Составляющие контекста, наследуемые при вызове fork()

- Окружение
- Файлы, открытые в процессе-отце
- Способы обработки сигналов
- Разрешение переустановки эффективного идентификатора пользователя
- Разделяемые ресурсы процесса-отца
- Текущий рабочий каталог и домашний каталоги
- …

Создание нового процесса

#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);

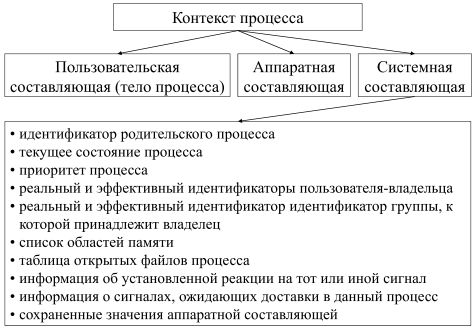
- При удачном завершении возвращается:
  - сыновнему процессу значение 0
  - родительскому процессу PID порожденного процесса
- При неудачном завершении возвращается –1, код ошибки устанавливается в переменной **errno**
- Заносится новая запись в таблицу процессов
- Новый процесс получает уникальный идентификатор
- Создание контекста для сыновнего процесса

Второе определение процесса в UNIX

Процесс в UNIX — это объект, порожденный системным вызовом **fork()**.

Системный вызов — обращение процесса к ядру ОС за выполнением тех или иных действий.

Контекст процесса



Семейство системных вызовов exec()

#include <unistd.h>
int execl (const char *path, char *arg0, …, char *argn, 0);

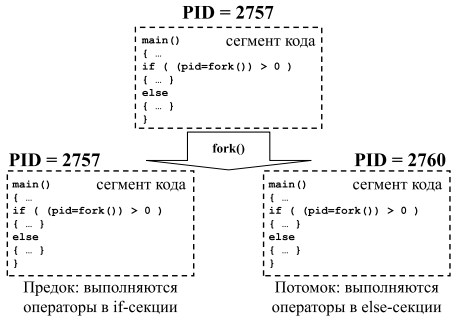
- **path** — имя файла, содержащего исполняемый код программы
- **arg0** — имя файла, содержащего вызываемую на выполнение программу
- **arg1, …, argn** — аргументы программы, передаваемые ей при вызове

Возвращается:	
в случае ошибки	–1

Пример

```
int main ( int argc, char **argv )
{
   printf ( "PID = %d; PPID = %d \n", getpid(), getppid() );
   fork ();
   printf ( "PID = %d; PPID = %d \n", getpid(), getppid() );
   return 0;
}
```

Схема создания нового процесса

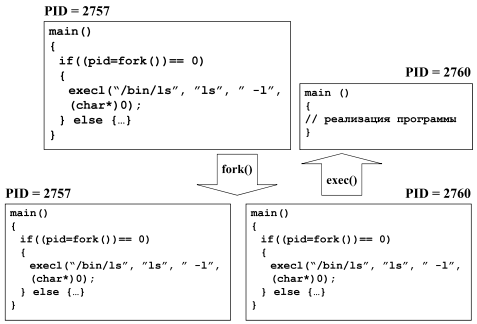


Создание нового процесса

Составляющие контекста, не наследуемые при вызове fork()

- Идентификатор процесса (PID)
- Идентификатор родительского процесса (PPID)
- Сигналы, ждущие доставки в родительский процесс
- Время посылки ожидающего сигнала, установленное системным вызовом **alarm()**
- Блокировки файлов, установленные родительским процессом

Использование схемы fork-exec



Пример

```
#include <unistd.h>

int main ( int argc, char **argv )
{
   …
   /* тело программы */
   …
   execl ( "/bin/ls", "ls", "-l", (char*) 0 );
   /* или execlp ( "ls" ,"ls", "-l", (char*) 0 ); */
   printf ( "это напечатается в случае неудачного обращения к предыдущей функции, к примеру, если не был найден файл ls.\n" );
   …
}
```

Семейство системных вызовов exec()

Сохраняются:

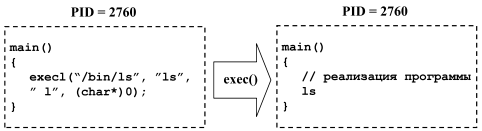
- Идентификатор процесса
- Идентификатор родительского процесса
- Таблица дескрипторов файлов
- Приоритет и большинство атрибутов

Изменяются:

- Режимы обработки сигналов
- Эффективные идентификаторы владельца и группы
- Файловые дескрипторы (закрытие некоторых файлов)

Семейство системных вызовов exec()

Схема работы системного вызова exec()



Получение информации о завершении своего потомка

#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);

status по завершению содержит:

- в старшем байте — код завершения процесса-потомка (**пользовательский код завершения процесса**)
- в младшем байте — индикатор причины завершения процесса-потомка, устанавливаемый ядром UNIX (**системный код завершения процесса**)

Возвращается: PID завершенного процесса или –1 в случае ошибки или прерывания

Завершение процесса

- Освобождается сегмента кода и сегмента данных процесса
- Закрываются все открытые дескрипторы файлов
- Если у процесса имеются потомки, их предком назначается процесс с идентификатором 1
- Освобождается большая часть контекста процесса (кроме статуса завершения и статистики выполнения)
- Процессу-предку посылается сигнал SIGCHLD

Завершение процесса

#include <unistd.h>
void _exit (int status);

status :

= 0 при успешном завершении

0 при неудаче (возможно, номер варианта)

Завершение процесса

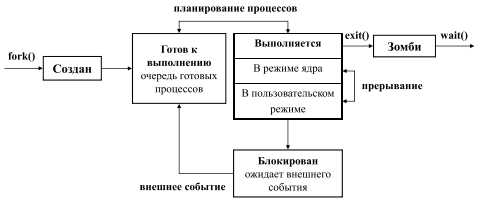
- Системный вызов **_exit()**
- Выполнение оператора **return**, входящего в состав функции **main()**
- Получение сигнала

Начальная загрузка

Начальная загрузка — загрузка ядра системы в оперативную память, запуск ядра.

- Чтение нулевого блока системного устройства аппаратным загрузчиком
- Поиск и считывание в память файла /unix
- Запуск на исполнение файла /unix

Жизненный цикл процессов



Пример. Использование системного вызова wait()

```
#include <stdio.h>
int main ( int argc, char **argv )
{
   int i;
   for ( i=1; i<argc; i++ ) {
      int status;
      if ( fork () > 0 ) {
         wait( &status );
         printf( "process-father\n" );
         continue;
      }
      execlp ( argv[i], argv[i], 0 );
      exit ();
   }
}
```

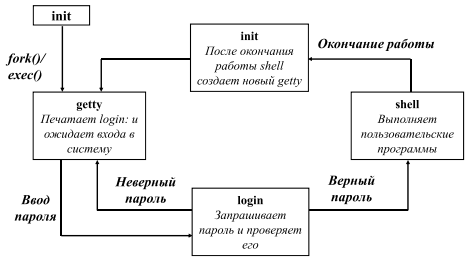
file prog1 prog2 prog3

<текст от prog1>
process-father
<текст от prog2>
process-father
<текст от prog3>
process-father

Получение информации о завершении своего потомка

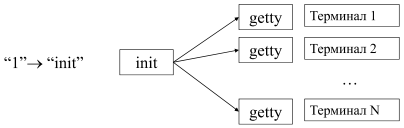
- Приостановка родительского процесса до завершения (остановки) какого-либо из потомков
- После передачи информации о статусе завершения предку, все структуры, связанные с процессом-«зомби» освобождаются, удаляется запись о нем из таблицы процессов

Схема дальнейшей работы системы



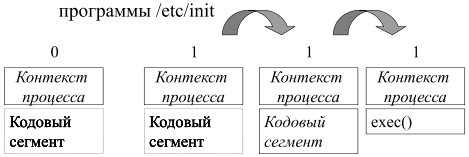
Инициализация системы

- Замена команды процесса “1” кодом из файла /etc/init (запуск exec())
- Подключение интерпретатора команд к системной консоли
- Создание многопользовательской среды



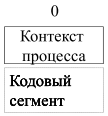
Инициализация системы

- Создание ядром первого процесса
  - Копируется процесс “0” (запись таблицы процессов)
  - Создание области кода процесса “1”
  - Копирование в область кода процесса “1” программы, реализующей системный вызов **exec()**, который необходим для выполнения программы /etc/init



Инициализация Unix системы

- Начальная инициализация компонентов компьютера (установка часов, инициализация контроллера памяти и пр.)
- Инициализация системных структур данных
- Инициализация процесса с номером “0”:
  - не имеет кодового сегмента
  - существует в течении всего времени работы системы

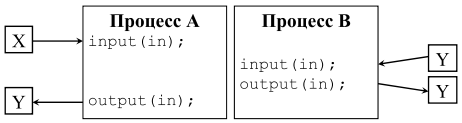


Требование мультипрограммирования

Результат выполнения процессов не должен зависеть от порядка переключения выполнения между процессами.

Гонки (race conditions) между процессами.

```
void echo ()
{
  char in;
  input ( in ) ;
  output ( in ) ;
}
```



Разделение ресурсов

Разделение ресурса — совместное использование несколькими процессами ресурса ВС.

Критические ресурсы — разделяемые ресурсы, которые должны быть доступны в текущий момент времени только одному процессу.

Критическая секция или **критический интервал** часть программы (фактически набор операций), в которой осуществляется работа с критическим ресурсом.

Параллельные процессы

Параллельные процессы — процессы, выполнение (обработка) которых хотя бы частично перекрывается по времени.

- **Независимые процессы** используют независимое множество ресурсов
- **Взаимодействующие процессы** используют ресурсы совместно, и выполнение одного процесса может оказать влияние на результат другого

Управление процессами

3. Взаимодействие процессов: синхронизация, тупики
   - 3.1. Разделение ресурсов
   - 3.2. Взаимное исключение
     - 3.2.1. Проблемы реализации взаимного исключения
     - 3.2.2. Способы реализации взаимного исключения
       - 3.2.2.1. Семафоры Дейкстры
       - 3.2.2.2. Мониторы
       - 3.2.2.3. Обмен сообщениями
   - 3.3. Примеры

Семафоры Дейкстры

Семафоры Дейкстры — формальная модель синхронизации, предложенная голландским учёным Эдсгером В. Дейкстрой

Операции, определённые над семафорами

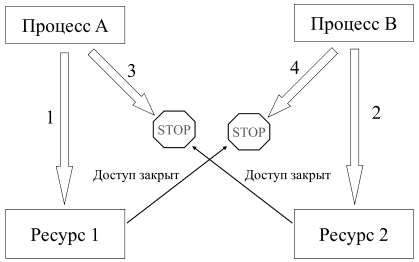
- Down (S) или P (S) – **P**roberen (проверить)
- Up (S) или V (S) – **V**erhogen (увеличить)

Способы реализации взаимного исключения

Способы, которые позволяют работать с разделяемыми ресурсами. Существуют как аппаратные, так и алгоритмические модели.

- Запрещение прерываний и специальные инструкции
- Алгоритм Петерсона
- Активное ожидание
- **Семафоры Дейкстры**
- **Мониторы Хоара**
- **Обмен сообщениями**

Тупики (Deadlocks)



Взаимное исключение

Взаимное исключение — способ работы с разделяемым ресурсом, при котором в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

- **Тупики (deadlocks)**
- **Блокирование (дискриминация)**

Тупик — ситуация, при которой из-за некорректной организации доступа и разделения ресурсов происходит взаимоблокировка.

Блокирование — доступ одного из процессов к разделяемому ресурсу не обеспечивается из-за активности других, более приоритетных процессов.

Обмен сообщениями

- Синхронизация
  - send/receive блокирующие
  - send/receive неблокирующими
- Адресация
  - Прямая (ID процесса)
  - Косвенная (почтовый ящик, или очередь сообщений)

Обмен сообщениями

Синхронизация и передача данных:

- для однопроцессорных систем и систем с общей памятью
- для распределенных систем (когда каждый процессор имеет доступ только к своей памяти)

Функции:
- send (destination, message)
- receive (source, message)

Мониторы Хоара

Монитор Хоара — совокупность процедур и структур данных, объединенных в программный модуль специального типа.

- Структуры данных монитора доступны только для процедур, входящих в этот монитор
- Процесс «входит» в монитор по вызову одной из его процедур
- В любой момент времени внутри монитора может находиться не более одного процесса

Использование двоичного семафора для организации взаимного исключения

Двоичный семафор — семафор, максимальное значение которого равно 1.

процесс 1
```
int semaphore;
…
down ( semaphore ) ;
/*критическая секция процесса 1*/
…
up ( semaphore ) ;
…
```

процесс 2
```
int semaphore;
…
down ( semaphore ) ;
/*критическая секция процесса 2*/
…
up ( semaphore ) ;
…
```

«Обедающие философы»

Правильное решение с использованием семафоров

```
# define  N  5
# define  LEFT  ( i – 1 ) % N
# define  RIGHT  ( i + 1 ) % N
# define  THINKING  0
# define  HUNGRY  1
# define  EATING  2

typedef  int  semaphore ;
int state [ N ] ;
semaphore mutex = 1 ;
semaphore s [ N ] ;
```

«Обедающие философы»

Простейшее решение

```
#define N 5

void philosopher ( int i )
{
  while (TRUE)
  {
    think () ;
    take_fork ( i ) ;
    take_fork ( ( i + 1 ) % N ) ;
    eat () ;
    put_fork ( i ) ;
    put_fork ( ( i + 1 )  % N ) ;
  }
  return;
}
```

«Обедающие философы» (доступ равноправных процессов к разделяемому ресурсу)

Классические задачи синхронизации процессов

1. Обедающие философы
2. Задача о читателях и писателях
3. Задача о спящем парикмахере

«Спящий парикмахер» (клиент-сервер с ограничением на длину очереди)

«Читатели и писатели» (задача резервирования ресурса)

```
typedef int semaphore ;
int rc = 0 ;
semaphore mutex = 1 ;
semaphore db = 1 ;

void reader ( void )
{
 while ( TRUE )
 {
  down ( & mutex ) ;
  rc++ ;
  if ( rc == 1 ) down ( & db ) ;
  up ( & mutex ) ;
  read_data_base () ;
  down ( & mutex ) ;
  rc-- ;
  if ( rc ==0 ) up ( & db ) ;
  up ( & mutex ) ;
  use_data_read () ;
 }
}

void writer ( void )
{
  while ( TRUE )
  {
    think_up_data () ;
    down ( & db ) ;
    write_data_base () ;
    up ( & db ) ;
  }
}
```

«Обедающие философы»

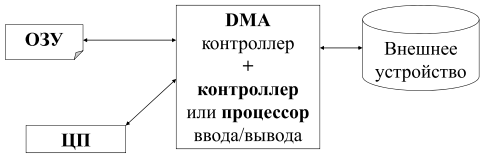
Правильное решение с использованием семафоров

```
void philosopher ( int i )
{ while ( TRUE )
  {
  think () ;
    take_forks ( i ) ;
    eat ();
    put_forks ( i ) ;
  }
}
void put_forks ( int i )
{
  down ( & mutex ) ;
  state[i] = THINKING ;
  test ( LEFT ) ;
  test ( RIGHT ) ;
  up ( & mutex ) ;
}
```

```
void take_forks ( int i )
{
  down ( & mutex ) ;
  state [ i ] =  HUNGRY ;
  test ( i ) ;
  up( & mutex ) ;
  down ( & s [ i ] ) ;
}

void test ( int i )
{
  if ( ( state [ i ] == HUNGRY )
  && ( state [ LEFT ] != EATING )
  && ( state [ RIGHT ] != EATING ))
  {
  state [ i ] = EATING ;
  up ( & s [ i ] ) ;
  }
}
```

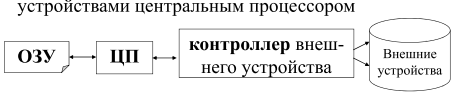
Управление внешними устройствами. Архитектура



4. Использование **контроллера прямого доступа** к памяти (DMA) при обмене.
5. Управление внешними устройствами с использованием **процессора или канала ввода/вывода**.

Управление внешними устройствами. Архитектура

1. **Непосредственное управление** внешними устройствами центральным процессором



2. **Синхронное управление** внешними устройствами использованием контроллеров внешних устройств
3. **Асинхронное управление** внешними устройствами с использованием **контроллеров внешних устройств**

Операционные системы

Управление внешними устройствами

```
#define CHAIRS 5
typedef int semaphore ;
semaphore customers = 0 ;

void barber ( void )
{
  while ( TRUE )
  {
    down ( & customers ) ;
    down ( & mutex ) ;
    waiting = wating – 1 ;
    up ( & barbers ) ;
    up ( & mutex ) ;
    cut_hair () ;
  }
}
```

```
semaphore barbers = 0 ;
semaphore mutex = 1 ;
int waiting = 0 ;

void customer ( void )
{
  down ( & mutex ) ;
  if ( waiting < CHAIRS )
  {
    waiting = waiting + 1 ;
    up ( & customers ) ;
    up ( & mutex ) ;
    down ( barbers ) ;
    get_haircut () ;
  }
  else
  {
    up ( & mutex ) ;
  }
}
```

Планирование дисковых обменов

Shortest Service Time First — «жадный» алгоритм — на каждом шаге поиск обмена с минимальным перемещением

SSTF

Путь головки	L
15 → 14	1
14 → 11	3
11 → 7	4
7 → 4	3
4 → 35	31
35 → 40	5
общ. 47 средн. 7,83	

LIFO

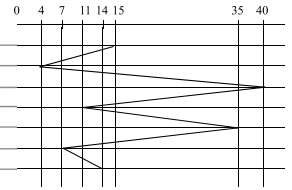
Путь головки	L
15 → 14	1
14 → 7	7
7 → 35	28
35 → 11	24
11 → 40	29
40 → 4	36
общ. 126 средн. 20,83	

Планирование дисковых обменов

Рассмотрим модельную ситуацию:
головка HDD позиционирована на дорожке 15
Очередь запросов к дорожкам: 4, 40, 11, 35, 7, 14

FIFO

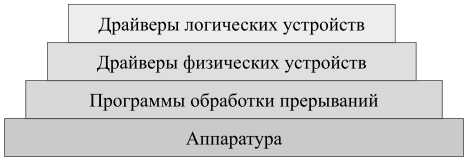
Путь головки	L
15 → 4	11
4 → 40	36
40 → 11	29
11 → 35	24
35 → 7	28
7 → 14	7
общ. 135 средн. 22,5	



Программное управление внешними устройствами

- унификация программных интерфейсов доступа к внешним устройствам (унификация именования, абстрагирование от свойств конкретных устройств)
- обеспечение конкретной модели синхронизации при выполнении обмена (синхронный, асинхронный обмен)
- обработка возникающих ошибок (индикация ошибки, локализация ошибки, попытка исправления ситуации);
- буферизация обмена
- обеспечение стратегии доступа к устройству (распределенный доступ, монопольный доступ);
- планирование выполнения операций обмена

Программное управление внешними устройствами



RAID системы

RAID (Redundant Array of Independent (Inexpensive) Disks) — избыточный массив независимых (недорогих) дисков.

RAID система — набор физических дисковых устройств, рассматриваемых операционной системой, как единое дисковое устройство (данные распределяются по физическим устройствам, образуется избыточная информация, используемая для контроля и восстановления информации).

Планирование дисковых обменов

N-step-SCAN

Разделение очереди на подочереди длины ≤ N запросов каждая (из соображений FIFO). Последовательная обработка очередей. Обрабатываемая очередь не обновляется. Обновление очередей, отличных от обрабатываемой.

Борьба с «залипанием» головки (интенсивный обмен с одной и той же дорожкой).

Планирование дисковых обменов

C-SCAN

Путь головки	L
15 → 4	11
4 → 7	3
7 → 11	4
11 → 14	3
14 → 35	21
35 → 40	5
общ. 47 средн. 7,83	

Циклическое сканирование
Сканирование в одном направлении. Ищем минимальный номер дорожки, затем «движемся наверх»

Планирование дисковых обменов

PRI — алгоритм, основанный на приоритетах процессов.

SCAN

Путь головки	L
15 → 35	20
35 → 40	5
40 → 14	26
14 → 11	3
11 → 7	4
7 → 4	3
общ. 61 средн. 10,16	

«Лифт» — сначала «движение» в одну сторону до «упора», затем в другую, также до «упора»

Для ∀ набора запросов
перемещений ≤ 2 × число_дорожек

Уровни RAID

RAID 5 (распределенная четность — циклическое распределение «четности»)

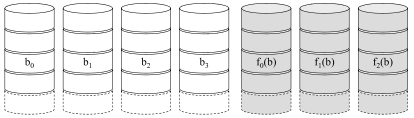
Уровни RAID

RAID 4
Пример: 4 диска данных, один – четности:

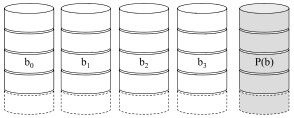
Изначально:
X4(i) = X3(i) XOR X2(i) XOR X1(i) XOR X0(i)

После обновления полосы на диске X1:
$X4_{new}(i) = X4(i)$ XOR X1(i) XOR $X1_{new}(i)$

Уровни RAID



RAID 2 избыточность с кодами Хэмминга



RAID 3 (четность с чередующимися битами)

Пример: 4 диска данных, один — четности: Потеря данных на первом диске

X4(i) = X3(i) XOR X2(i) XOR X1(i) XOR X0(i)

X1(i) = X4(i) XOR X3(i) XOR X2(i) XOR X0(i)

Уровни RAID

RAID 0 (без избыточности)

RAID 1 (зеркалирование)

Файлы устройств

- Содержимое файлов устройств размещается исключительно в соответствующем индексном дескрипторе
- Структура ИД файла устройства:
  - «Старший номер» (major number) устройства
  - Тип файла устройства
  - «Младший номер» (minor number) устройства
- Системные таблицы драйверов устройств:
  - **bdevsw**
  - **cdevsw**

Файлы устройств, драйверы

- Иерархия драйверов
- Специальные файлы устройств (/dev)
  - Байт-ориентированные устройства
  - Блок-ориентированные устройства

UNIX: Работа с внешними устройствами

1. Файлы устройств, драйверы
   1.1. Файлы устройств
   1.2. Системные таблицы драйверов устройств
   1.3. Ситуации, вызывающие обращение к функциям драйвера
   1.4. Включение/удаление драйверов в системе
2. Организация обмена данных с файлами
3. Буферизация при блок-ориентированном обмене
4. Борьба со сбоями

Уровни RAID

RAID 6 (двойная избыточность — циклическое распределение четности с использованием двух схем контроля: N+2 дисков)

Организация обмена данными с файлами

- Таблица индексных дескрипторов открытых файлов (размещается в памяти ядра ОС)
- Таблица файлов (размещается в памяти ОС)
- Таблица открытых файлов

Включение/удаление драйверов в систему

- «Жёсткое», статистическое встраивание драйверов в код ядра
- Динамическое включение драйвера в систему
  - Загрузка и динамическое связывание драйвера с кодом ядра
  - Инициализация драйвера и соответствующего ему устройства

Ситуации, вызывающие обращение к функциям драйвера

- Старт системы, определение ядром состава доступных устройств
- Обработка запроса ввода/вывода
- Обработка прерывания, связанного с данным устройством
- Выполнение специальных команд управления

Системные таблицы драйверов устройств

- Запись таблицы — **коммутатор устройства**
- Типовой набор точек входа в драйвер:
  - **βopen(), βclose()**
  - **βread(), βwrite()**
  - **βioctl()**
  - **βintr()**
  - **βstrategy()**

Борьба со сбоями

- Наличие параметра, определяющего периоды времени, через которые осуществляется сброс системных данных, который может оперативно меняться
- Пользовательская команда SYNC
- Избыточность системы, позволяющая восстанавливать информацию

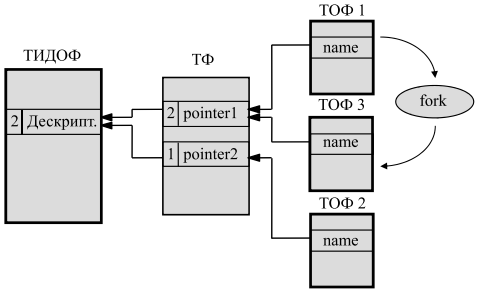
Буферизация при блок-ориентированном обмене

- Оптимизация работы ОС, за счет минимизации реальных обращений к физическому устройству
- Недостатки:
  - Критичность к несанкционированным отключениям питания
  - Разорванность во времени факта обращения к системе за обменом и реальным обменом

Буферизация при блок-ориентированном обмене

1. Поиск заданного блока в буферном пуле. Нашли-переходим на шаг 4
2. Поиск буфера в буферном пуле для чтения и размещения заданного блока
3. Чтение блока в найденный буфер
4. Изменение счётчика времени во всех буферах
5. Содержимое буфера передаётся в качестве результата

Пример



Одиночное непрерывное распределение

Необходимые аппаратные средства

- Регистр границ + режим ОС / режим пользователя
- Если ЦП в режиме пользователя попытается обратиться в область ОС, то возникает прерывание

Алгоритмы: очевидны.

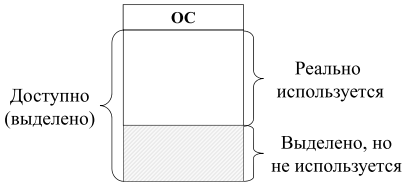
Достоинства: простота.

Недостатки

1. Часть памяти не используется
2. Процессом/заданием память занимается все время выполнения
3. Ограничение на размеры задания

Одиночное непрерывное распределение

Основные концепции



Управление оперативной памятью

Стратегии и методы управления

1. Одиночное непрерывное распределение
2. Распределение разделами
3. Распределение перемещаемыми разделами
4. Страничное распределение
5. Сегментное распределение
6. Сегментно-страничное распределение

План рассмотрения стратегий управления

1. Основные концепции
2. Необходимые аппаратные средства
3. Основные алгоритмы
4. Достоинства, недостатки

Управление оперативной памятью

Основные задачи

1. Контроль состояния каждой единицы памяти (свободна/распределена)
2. Стратегия распределения памяти (кому, когда и сколько памяти должно быть выделено)
3. Выделение памяти (выбор конкретной области, которая должна быть выделена)
4. Стратегия освобождения памяти (процесс освобождает, ОС “забирает” окончательно или временно)

Распределение неперемещаемыми разделами

Алгоритмы

Модель статического определения разделов

Б. Одна входная очередь процессов

1. Освобождение раздела ⇒ поиск (в начале очереди) первого процесса, который может разместиться в разделе.
   Проблема: большие разделы ↔ маленькие процессы
2. Освобождение раздела ⇒ поиск процесса максимального размера, не превосходящего размер раздела.
   Проблема: дискриминация «маленьких» процессов
3. Оптимизация варианта 2. Каждый процесс имеет счетчик дискриминации. Если значение счетчика процесса ≥ K, то обход его в очереди невозможен

Распределение неперемещаемыми разделами

Алгоритмы

Модель статического определения разделов

А. Сортировка входной очереди процессов по отдельным очередям к разделам. Процесс размещается в разделе минимального размера, достаточного для размещения данного процесса. В случае отсутствия процессов в каких-то подочередях — неэффективность использования памяти.

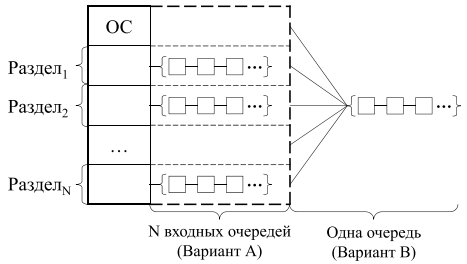
Распределение неперемещаемыми разделами

Необходимые аппаратные средства

1. Два регистра границ
2. Ключи защиты (PSW)

Распределение неперемещаемыми разделами

Основные концепции



Распределение перемещаемыми разделами

Достоинства

1. Ликвидация фрагментации

Недостатки

1. Ограничение размером физической памяти
2. Затраты на перекомпоновку

Распределение перемещаемыми разделами

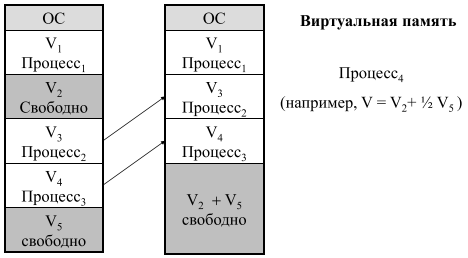
Необходимые аппаратные средства

1. Регистры границ + регистр базы
2. Ключи + регистр базы

Алгоритмы:

Распределение перемещаемыми разделами

Основные концепции



Распределение неперемещаемыми разделами

Достоинства

1. Простое средство организации мультипрограммирования
2. Простые средства аппаратной поддержки
3. Простые алгоритмы

Недостатки

1. Фрагментация
2. Ограничение размерами физической памяти
3. Весь процесс размещается в памяти — возможно неэффективное использование

Страничное распределение

Алгоритмы и организация данных

Размер и организация таблицы страниц ???

Модельная структура записи таблицы страниц

ε	δ	γ	β	α	Номер физической страницы

α — присутствие/отсутствие
β — защита (чтение, чтение/запись, выполнение)
γ — изменения
δ — обращение (чтение, запись, выполнение)
ε — блокировка кэширования

Страничное распределение

Необходимые аппаратные средства

1. Полностью аппаратная таблица страниц (стоимость, полная перегрузка при смене контекстов, скорость преобразования)
2. Таблица страниц в ОЗУ + Регистр начала таблицы страниц в памяти (простота, управление смены контекстов, медленное преобразование)
3. Гибридные решения

Страничное распределение

Основные концепции

Таблица страниц — отображение номеров виртуальных страниц на номера физических.

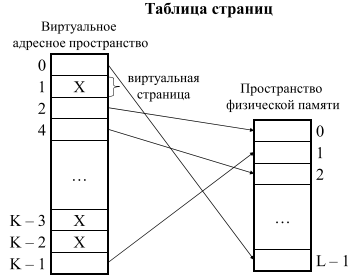
Проблемы

1. Размер таблицы страниц (количество 4KB страниц при 32-х разрядной адресации — 1 000 000; любой процесс имеет собственную таблицу страниц)
2. Скорость отображения

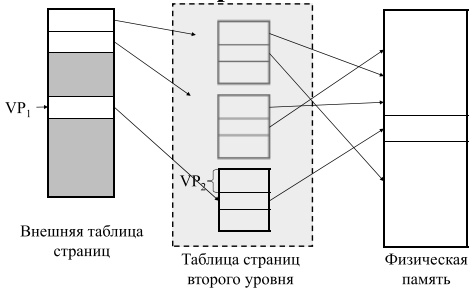
Страничное распределение

Основные концепции

Таблица страниц

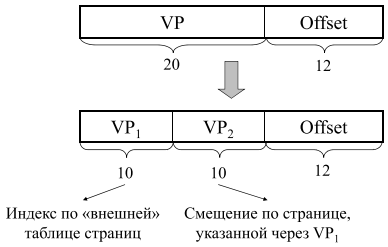


Иерархическая организация таблицы страниц



Иерархическая организация таблицы страниц

Двухуровневая организация



Иерархическая организация таблицы страниц

Проблема — размер таблицы страниц.

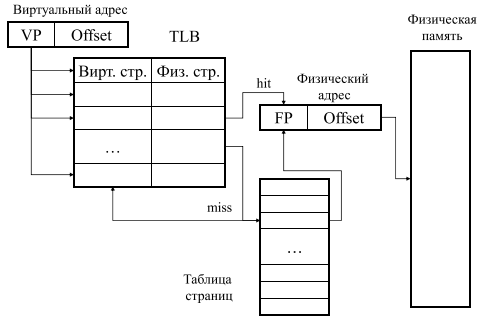
Объем виртуальной памяти современного компьютера — $2^{32}…2^{64}$ байт

Пример

$V_{вирт.} = 2^{32}$
$V_{стр.} = 2^{12}$ (4KB) ⇒ Количество виртуальных страниц — 2^{20} (**много**)

Решение — использование многоуровневых таблиц страниц (2^x, 3^x, 4^x)

TLB (Translation Lookaside Buffer)



Замещение страниц

Алгоритм

1. При запуске процесса M и R для всех страниц процесса обнуляются
2. По таймеру происходит обнуление всех битов R
3. При возникновении страничного прерывания ОС делит все страниц на классы:
   - Класс 0: M=0, R=0
   - Класс 1: M=1, R=0
   - Класс 2: M=0, R=1
   - Класс 3: M=1, R=1
4. Случайная выборка страницы для удаления в непустом классе с минимальным номером

Замещение страниц

Проблема загрузки «новой» страницы в память. Необходимо выбрать страницу для удаления из памяти (с учетом ее модификации и пр.)

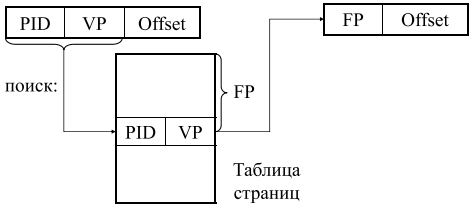
Алгоритм NRU (Not Recently Used — не использовавшийся в последнее время)

Используются биты статуса страницы в записях таблицы страниц

R — обращение
M — изменение
} устанавливаются аппаратно

обнуление — программно (ОС)

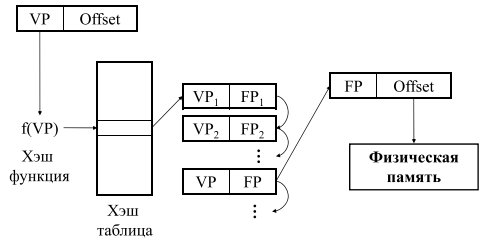
Инвертированные таблицы страниц



Проблема — поиск по таблице (хэширование)

Решение проблемы перезагрузки таблицы страниц при смене обрабатываемых ЦП процессов

использование хэш-таолиц (функция расстановки)



Замещение страниц

Алгоритм NFU (Not Frequently Used — редко использовавшаяся страница)

Для каждой физической страницы i – программный счетчик $Count_i$

0. Изначально $Count_i$ – обнуляется для всех i.

1. По таймеру $Count_i = Count_i + R_i$

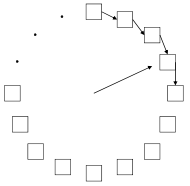
Выбор страницы с минимальным значением $Count_i$

Недостатки

- «помнит» старую активность
- при большой активности, возможно переполнение счетчика

Замещение страниц

Алгоритм «Часы»



1. Если R = 0, то выгрузка страницы и стрелка на позицию вправо
2. Если R = 1, то R обнуляется, стрелка на позицию вправо и на п.1

Замещение страниц

Алгоритм FIFO

«Первым прибыл — первым удален» — простейший вариант FIFO. Проблема «справедливости»

Модификация алгоритма (алгоритм вторая попытка)

1. Выбирается самая «старая страница». Если R=0, то она заменяется
2. Если R = 1, то R обнуляется, обновляется время загрузки страницы в память (т.е. переносится в конец очереди). На п.1

Замещение страниц

Стратегия: лучше выгрузить измененную страницу, к которой не было обращений как минимум в течение 1 «тика» таймера, чем часто используемую страницу

Сегментно-страничная организация памяти

Основные концепции:

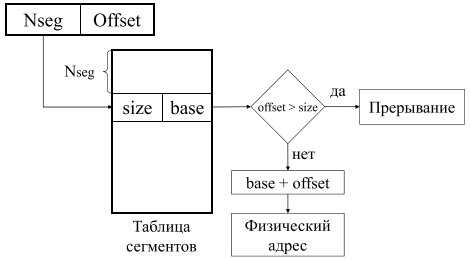
Nseg	Npage	Offset

Логический адрес

Сегментная организация памяти

Необходимые аппаратные средства

Виртуальный адрес:



Сегментная организация памяти

Основные концепции

- Виртуальное адресное пространство представляется в виде совокупности сегментов
- Каждый сегмент имеет свою виртуальную адресацию (от 0 до N – 1)
- Виртуальный адрес: <номер_сегмента, смещение>

Замещение страниц

Модификация NFU — алгоритм старения

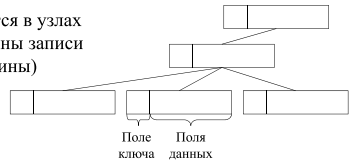
Модификация:

1. Значение счетчика сдвигается на 1 разряд вправо
2. Значение R добавляется в крайний левый разряд счетчика

Структурная организация файлов

1. Файл, как **последовательность байтов**
2. Файл, как **последовательность записей переменной длины**
3. Файл, как **последовательность записей постоянной длины**
4. Иерархическая организация файлов (дерево)

Записи находятся в узлах дерева (возможны записи переменной длины)



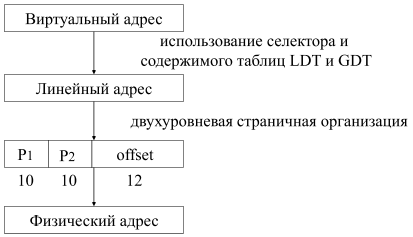
Операционные системы

Файловые системы (часть 1)

1. Базовые методы организации ФС
   - 1.1. Общие концепции
     - 1.1.1. Структурная организация файлов
     - 1.1.2. Атрибуты файла
     - 1.1.3. Работа с файлами
     - 1.1.4. Модельная организация каталогов ФС
   - 1.2. Подходы в практической реализации ФС
     - 1.2.1. Модели реализации файлов
     - 1.2.2. Таблица размещения файловой системы
     - 1.2.3. Модели организации каталогов
     - 1.2.4. Соответствие имени и содержимого файла
     - 1.2.5. Учёт свободных блоков ФС
     - 1.2.6. Квотирование пространства ФС
   - 1.3. Надёжность файловой системы
   - 1.4. Проверка файловой системы

Сегментно-страничная организация памяти

Необходимые аппаратные средства



Сегментно-страничная организация памяти

Модельный пример (Intel):

Виртуальный адрес:

Селектор	Offset

Nseg	Локализация	Защита

Таблицы локальных дескрипторов (сегменты доступные для данного процесса) LDT (Local Descriptor Table)

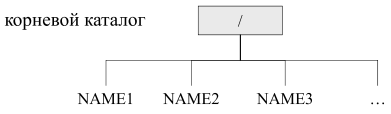
Таблица глобальных дескрипторов (разделяемые между процессами сегменты) GDT (Global Descriptor Table)

Каждая запись LDT и GDT – полная информация о сегменте (адрес базы, размер и т.д.).

Модельная организация каталогов файловых систем

Каталог — компонент файловой системы, содержащий информацию о содержащихся в файловой системе файлах. Каталоги являются специальным видом файлов.

Модель одноуровневой файловой системы



Типовые программные интерфейсы работы с файлами

open — открытие / создание файла

close — закрытие

read / write — читать, писать (относительно положения указателя чтения / запись)

delete — удалить файл из файловой системы

seek — позиционирование указателя чтение/запись

rename — переименование файла

read / write _attributes — чтение, модификация атрибутов файла

Основные сценарии работы с файлами

Начало

Открытие файла (регистрация в системе возможности работы процесса с содержимым файла)

Работа с содержимым файла, с атрибутами файла

Завершение

Закрытие файла — информация системе о завершении работы процесса с открытым файлом

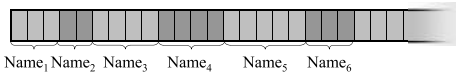
Файловый дескриптор — системная структура данных, содержащая информацию об актуальном состоянии открытого файла.

Атрибуты файла

- Имя
- Права доступа
- Персонификация (создатель, владелец)
- Тип файла
- Размер записи
- Размер файла
- Указатель чтения / записи
- Время создания
- Время последней модификации
- Время последнего обращения
- Предельный размер файла
- …

Модели реализации файлов

Непрерывные файлы



Достоинства

- Простота реализации
- Высокая производительность

Недостатки

- Фрагментация свободного пространства
- Увеличение размера существующего файла

Подходы в практической реализации файловой системы

Структура «системного» диска

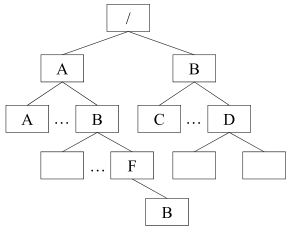
Блок физического HDD ➔ Блок виртуального HDD
Блок файловой системы
Блок файла

Модельная организация каталогов файловых систем

Иерархические файловые системы

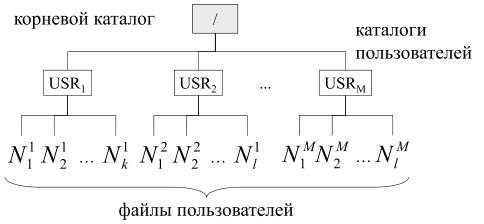
Понятия

- имя файла
- полное имя файла
- относительное имя
- домашний каталог
- текущий каталог



Модельная организация каталогов файловых систем

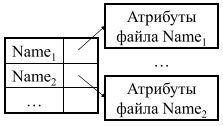
Модель двухуровневой файловой системы



Модели организации каталогов

$Name_1$	Атрибуты
$Name_2$	Атрибуты
…	

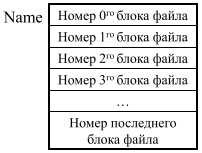
Записи каталога фиксированного размера, содержат имя файла и все его атрибуты.



Каталог содержит имя файла и ссылку на атрибуты файла. Размер атрибутов может варьироваться.

Организация «длинных» имен файлов?

Индексные узлы (дескрипторы)



Индексный узел (дескриптор) — системная структура данных, содержащая информацию о размещении блоков конкретного файла в файловой системе.

Достоинства

- Нет необходимости в размещении в ОЗУ информации всей FAT о всех файлах системы, в памяти размещаются атрибуты, связанные только с открытыми файлами

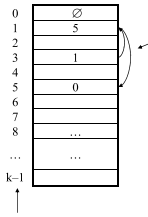
Недостатки

- Размер файла и размер индексного узла (в общем случае прийти к размерам таблицы размещения).

Решение: — ограничение размера файла
— иерархическая организация индексных узлов

Таблица размещения файловой системы

(File Allocation Table — FAT)



начальный блок файла $Name_1$

номера блоков файловой системы

Достоинства

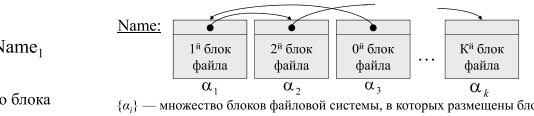
- Возможность использования всего блока для хранения данных файла
- Оптимизация прямого доступа (при полном или частичном размещении таблицы в ОЗУ)

Недостатки

- Желательно размещение всей таблицы в ОЗУ (проблема размера, например для 60 GB раздела и блоков размером 1KB потребуется 60 000 000*4B = 240 MB)

Модели реализации файлов

Файлы, имеющие организацию связанного списка



$\{\alpha_i\}$ — множество блоков файловой системы, в которых размещены блоки файла Name.

Достоинства

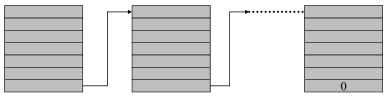
- Отсутствие фрагментации свободного пространства (за исключением блочной фрагментации)
- Простота реализации
- Эффективный последовательный доступ

Недостатки

- Сложность (неэффективность) организации прямого доступа
- Фрагментация файла по диску
- Наличие ссылки в блоке файла (ситуации чтения 2-х блоков при необходимости чтения данных объемом один блок)

Учет свободных блоков файловой системы

Связный список свободных блоков



При использовании связного списка свободных блоков в ОЗУ размещается первый блок списка.

Координация использования пространства внешней памяти

Проблема — размер блока файловой системы.

«Большой блок»:

- эффективность обмена
- существенная внутренняя фрагментация (неэффективное использование пространства ВП)

«Маленький блок»:

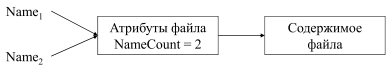
- эффективное использование пространства ВП
- фрагментация данных файла по диску

Проблема — определение оптимального размера блока.

Взаимнооднозначное соответствие: имя файла — содержимое файла?

2. Содержимому файла может соответствовать два и более имен файла.

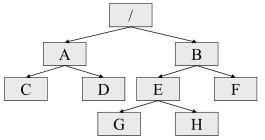
2.1 «Жесткая» связь



2.2 “Символическая” связь

Взаимнооднозначное соответствие: имя файла — содержимое файла

1. Содержимому любого файла соответствует единственное имя файла



Надежность файловой системы

Стратегии архивирования

- **Физическая архивация**
  - «Один в один»
  - Интеллектуальная физическая архивация (копируются только использованные блоки файловой системы)
  - Проблема обработки дефектных блоков
- **Логическая архивация** — копирование файлов (а не блоков), модифицированных после заданной даты.

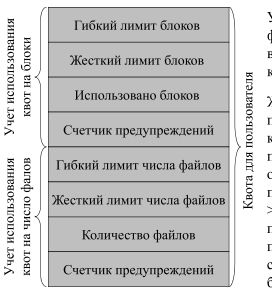
Надежность файловой системы

- Потеря информации в результате аппаратного или программного сбоя
- Случайное удаление файлов

⇒ **Резервное копирование (архивирование):**

- Копируются не все файлы файловой системы (избирательность архивирования по типам файлов)
- Инкрементное архивирование (резервное копирование) — единожды создается «полная» копия, все последующие включают только обновленные файлы
- Использование компрессии при архивировании (риск потери всего архива из-за ошибки в чтении/записи сжатых данных)
- Проблема архивирования «на ходу» (во время копирования происходят изменения файлов, создание, удаление каталогов и т.д.)
- Распределенное хранение резервных копий

Квотирование пространства файловой системы



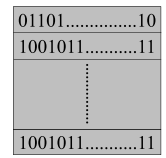
Учет количества и размеров тех файлов, у которых атрибут владельца соответствует конкретному пользователю.

Жесткие лимиты не превышаются никогда. Гибкие квоты можно превышать, но после этого включается обратный счетчик предупреждений. Пока счетчик > 0, при каждой регистрации пользователя в системе от получает предупреждение, если счетчик = 0, пользователь блокируется.

Использование битового массива

Состояние любого блока определяется содержимым бита с номером каждого блока.

Если блок свободен, бит равен 1, занят — 0.



Пример

Для 16GB HDD потребуется 2048 блоков для хранения битового массива

Проверка целостности файловой системы

2.

	0	1	2	3	4	5
Таблица занятых блоков	1	0	0	1	0	1
Таблица свободных блоков	0	0	1	0	1	0

Пропавший блок

Оставить как есть, но система «замусоривается».

Добавить в список свободных блоков файловой системы.

3.

	0	1	2	3	4	5
Таблица занятых блоков	1	0	0	1	0	1
Таблица свободных блоков	0	1	2	0	1	0

Дубликат свободного блока – пересоздание списка свободных блоков.

Проверка целостности файловой системы

Варианты анализа таблиц

1.

	0	1	2	3	4	5
Таблица занятых блоков	1	1	0	1	0	1
Таблица свободных блоков	0	0	1	0	1	0

Непротиворечивость файловой системы соблюдена.

Проверка целостности файловой системы

Контроль непротиворечивости блоков файловой системы:

Модельная стратегия контроля

1. Формируются две таблицы:
   - таблица занятых блоков
   - таблица свободных блоков

(размеры таблиц соответствуют размеру файловой системы — число записей равно числу блоков ФС)

Изначально все записи таблиц обнуляются.

2. Анализируется список свободных блоков. Для каждого номера свободного блока увеличивается на 1 соответствующая ему запись в таблице свободных
3. Анализируются все индексные узлы. Для каждого блока, встретившегося в индексном узле, увеличивается его счетчик на 1 в таблице занятых блоков
4. Анализ содержимого таблиц и коррекция ситуаций

Проверка целостности файловой системы

Проблема — при аппаратных или программных сбоях возможна потеря информации:

- потеря модифицированных данных в «обычных» файлах
- потеря системной информации (содержимое каталогов, списков системных блоков, индексные узлы и т.д.)

Контроль целостности или непротиворечивости файловой системы.

Организация ФС UNIX. Виды файлов

Файл UNIX — это специальным образом именованный набор данных, размещенный в системе.

Виды файлов

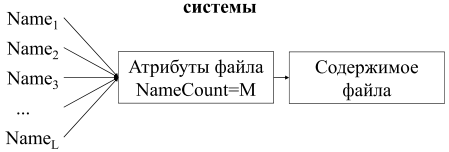
- Обычный файл (regular file)
- Каталог (directory)
- Специальный файл устройств (special device file)
- Именованный канал (FIFO)
- Ссылка (link)
- Сокет (socket)

Операционные системы

Файловые системы (часть 2)

Проверка целостности файловой системы

Контроль непротиворечивости файлов файловой системы



Возможны варианты:

1. L = M — все в порядке
2. L > M
3. L < M

⇒ NameCount = L

Проверка целостности файловой системы

4.

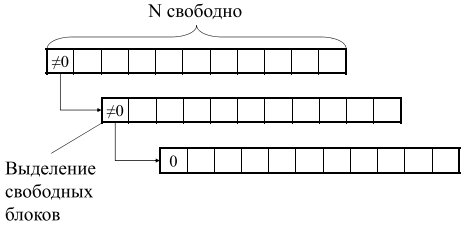
	0	1	2	3	4	5
Таблица занятых блоков	1	2	0	1	0	1
Таблица свободных блоков	0	0	1	0	1	0

Дубликат занятого блока ⇒ автоматическое решение максимально затруднено, имеет место потеря информации в одном из файлов.

Действие

1. $Name_1$ → копируется $Name_1^2$
2. $Name_2$ → копируется $Name_2^2$
3. Удаляются $Name_1$, $Name_2$
4. Запускается переопределение списка свободных блоков
5. Обратное переименование файлов и фиксация факта их возможной проблемности

Работа с массивами номеров свободных блоков



Модель версии System V

Структура ФС

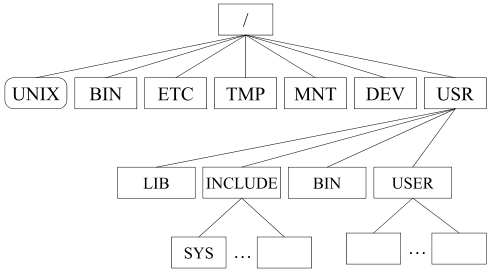
Суперблок	Область индексных дескрипторов	Блоки

Суперблок файловой системы содержит оперативную информацию о текущем состоянии файловой системы, а также данные о параметрах настройки.

Индексный дескриптор — специальная структура данных ФС, которая ставится во взаимнооднозначное соответствие с каждым файлом.

Блоки — свободные, занятые под системную информацию, занятые файлами.

Логическая структура каталогов



Права доступа

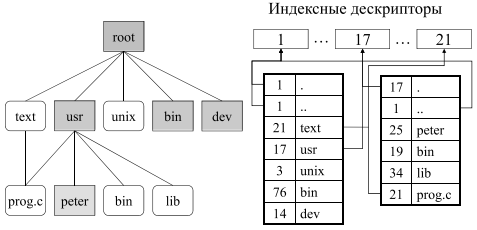
Категории пользователей

- Пользователь
- Группа
- Все пользователи системы

Права

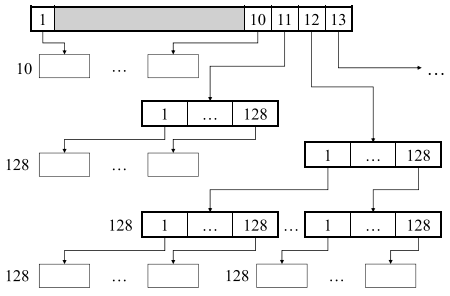
- На чтение
- На запись
- На исполнение

Файл каталог

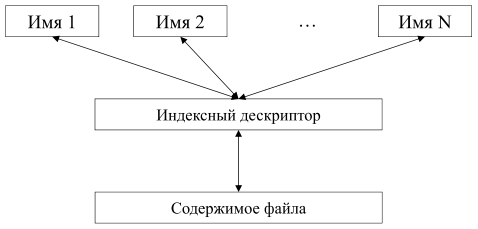


Адресация блоков файла

Модельный пример:
блок 512b; ссылка на блок 4b



Индексные дескрипторы



Работа с массивом свободных ИД

- Освобождение ИД
  - Есть свободное место — номер → элемент массива
  - Нет свободного места — номер «забывается»
- Запрос ИД
  - Поиск в массиве
    - Массив пустой — обновление массива
    - Массив не пустой — OK

Модель версии FFS BSD

- Стратегия размещения
- Внутренняя организация блоков
- Выделение пространства для файла
- Структура каталога FFS

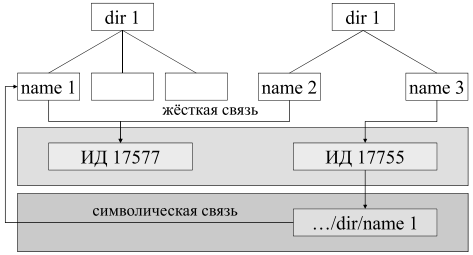
Недостатки ФС модели версии System V

- Концентрация важной информации в суперблоке
- Проблема надежности
- Фрагментация файла по диску
- Ограничения на возможную длину имени файла

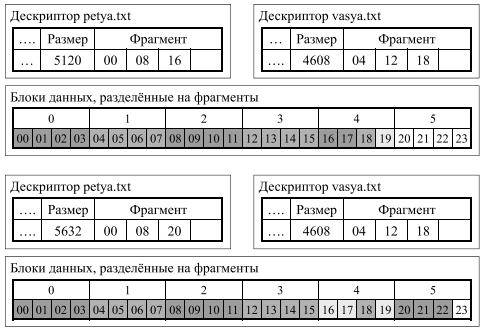
Достоинства ФС модели версии System V

- Оптимизация в работе со списками номеров свободных индексных дескрипторов и блоков
- Организация косвенной адресации блоков файлов

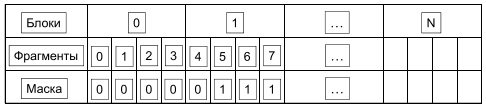
Установление связей



Выделение пространства для файла

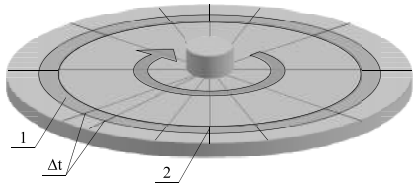


Внутренняя организация блоков

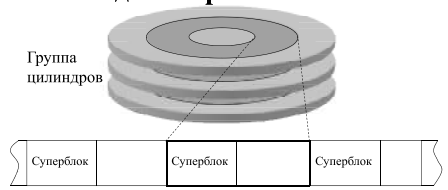


Стратегия размещения

- Размещение каталога
- Равномерность использования блоков данных
- Размещение последовательных блоков файлов



Модель версии FFS BSD



- копия суперблока
- информация о свободных блоках (битовый массив) и о свободных индексных дескрипторах
- массив индексных дескрипторов (ИД)
- блоки файлов

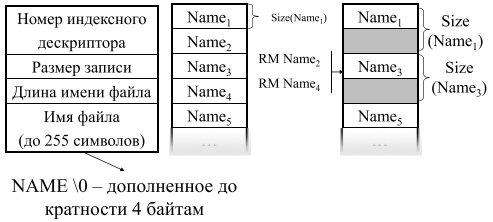
Блокировка доступа к содержимому файла

Возможность блокирования области файла любого размера.

Системный вызов **fcntl ()**

- **Исключающая блокировка** (**exclusive lock**) — «жесткая» блокировка (область может быть заблокирована единственный раз). Блокировка с монополизацией.
- **Распределенная блокировка** (**shared lock**) — «мягкая» блокировка (возможны пересечения заблокированных областей). Рекомендательная блокировка.

Структура каталога FFS



NAME \0 – дополненное до кратности 4 байтам

Фрагментация каталога → Дефрагментация

Прямой поиск → Кэширование имен файлов