



Конспект лекций курса

# ОПЕРАЦИОННЫЕ СИСТЕМЫ

Всеволод Дёмкин  
КПИ, Киев, 2015

# Литература

- Шеховцов В.А. - Операційні системи
- Валерій Габрусев, В. Лапінський, О. Нестеренко - Основи операційних систем. Ядро, процес, потік.
- Э. Таненбаум - Операционные системы: Разработка и реализация
- Основы ОС
- Основы ОС, практикум
- Erik Helin, Adam Renberg - The little book about OS development
- Silberschatz, Galvin, Gagne - Operating Systems Concepts with Java
- Operating Systems Course in MIT
- Advanced Linux Programming

# Введение

## ОС

ОС — это комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как **интерфейс** между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для **управления** устройствами и вычислительными процессами, эффективного распределения **ресурсов** между вычислительными процессами и организации надёжных вычислений.

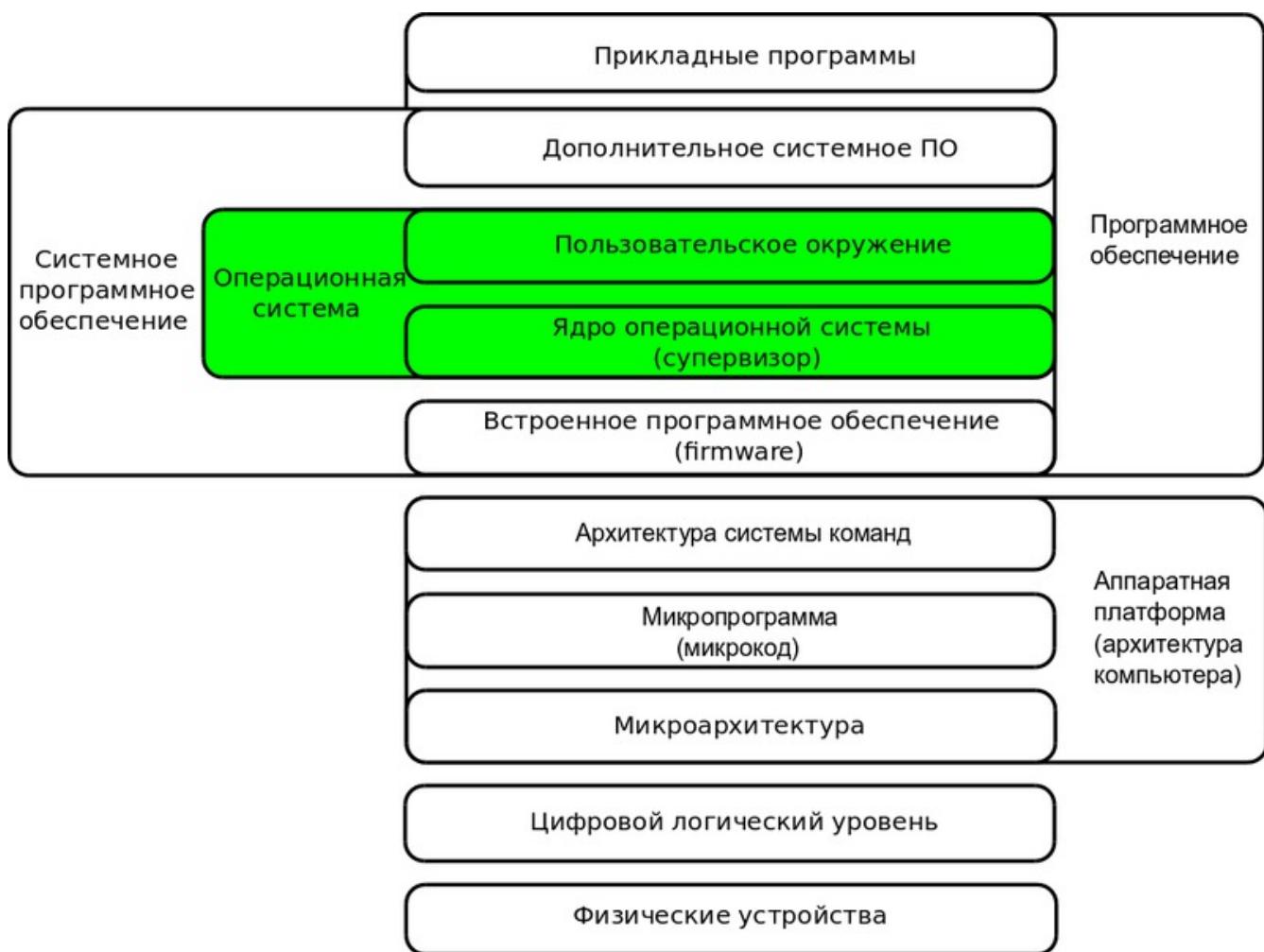


Рис. 1.1. Место ОС

### Задачи ОС:

- управление аппаратной частью (менеджер ресурсов)
- абстракция аппаратной части (виртуальная машина)

- изоляция приложений от аппаратной части (во избежание порчи)

## Программа в памяти

### Программа Hello World:

```
#include <stdio.h>
#include <unistd.h>
int main ()
{
    printf("Hello World");
    exit(0);
}
```

### Один из вариантов дизассемблирования:

```
.section .rodata
.LC0:
.string "Hello World"
.text
.globl main
.type main,@function
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    subl $12, %esp
    pushl $.LC0
    call printf
    addl $16, %esp
    subl $12, %esp
    pushl $0
    call exit
```

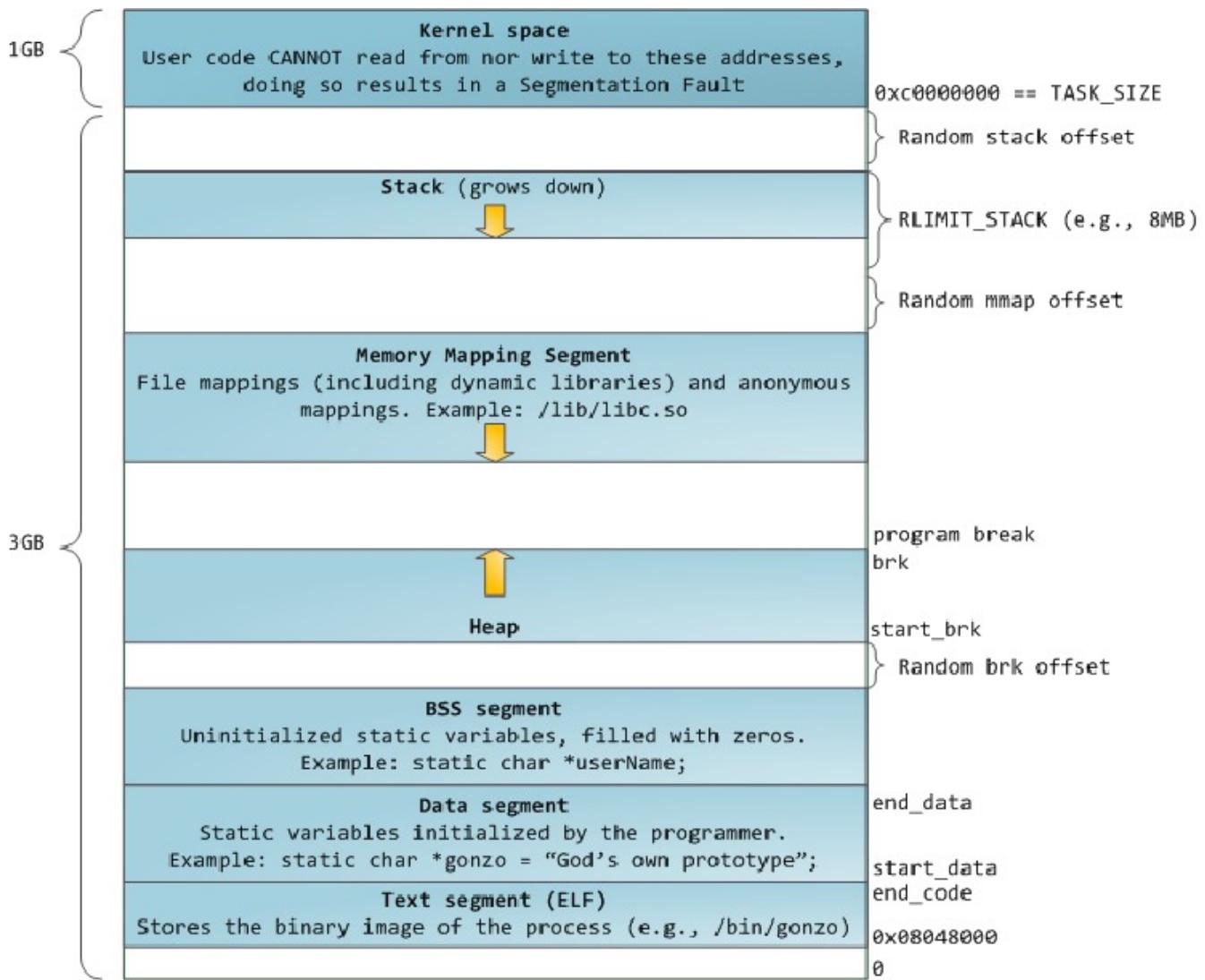


Рис. 1.2. Программа в памяти

## Ядро ОС

Ядро ОС — это центральная часть операционной системы, обеспечивающая приложениям координированный доступ к ресурсам компьютера, таким как процессорное время, память, внешнее аппаратное обеспечение, внешнее устройство ввода и вывода информации. Также обычно ядро предоставляет сервисы файловой системы и сетевых протоколов.

Ядро — тоже программа.

## Варианты реализации ядра:

- МОНОЛИТНОЕ: одна монолитная программа в памяти — +простота, +скорость, -ошибки, -перекомпиляция

- модульное: монолитная программа, предоставляющая интерфейс загрузки и выгрузки доп.модулей — снимает проблему перекомпиляции
- микроядро: несколько программ, которые взаимодействуют через передачу сообщений — +изоляция, +слабая связность, -сложность, -скорость
- наноядро: ядро только управляет ресурсами (обработка прерываний)
- экзоядро: наноядро с координацией работы процессов
- гибридное

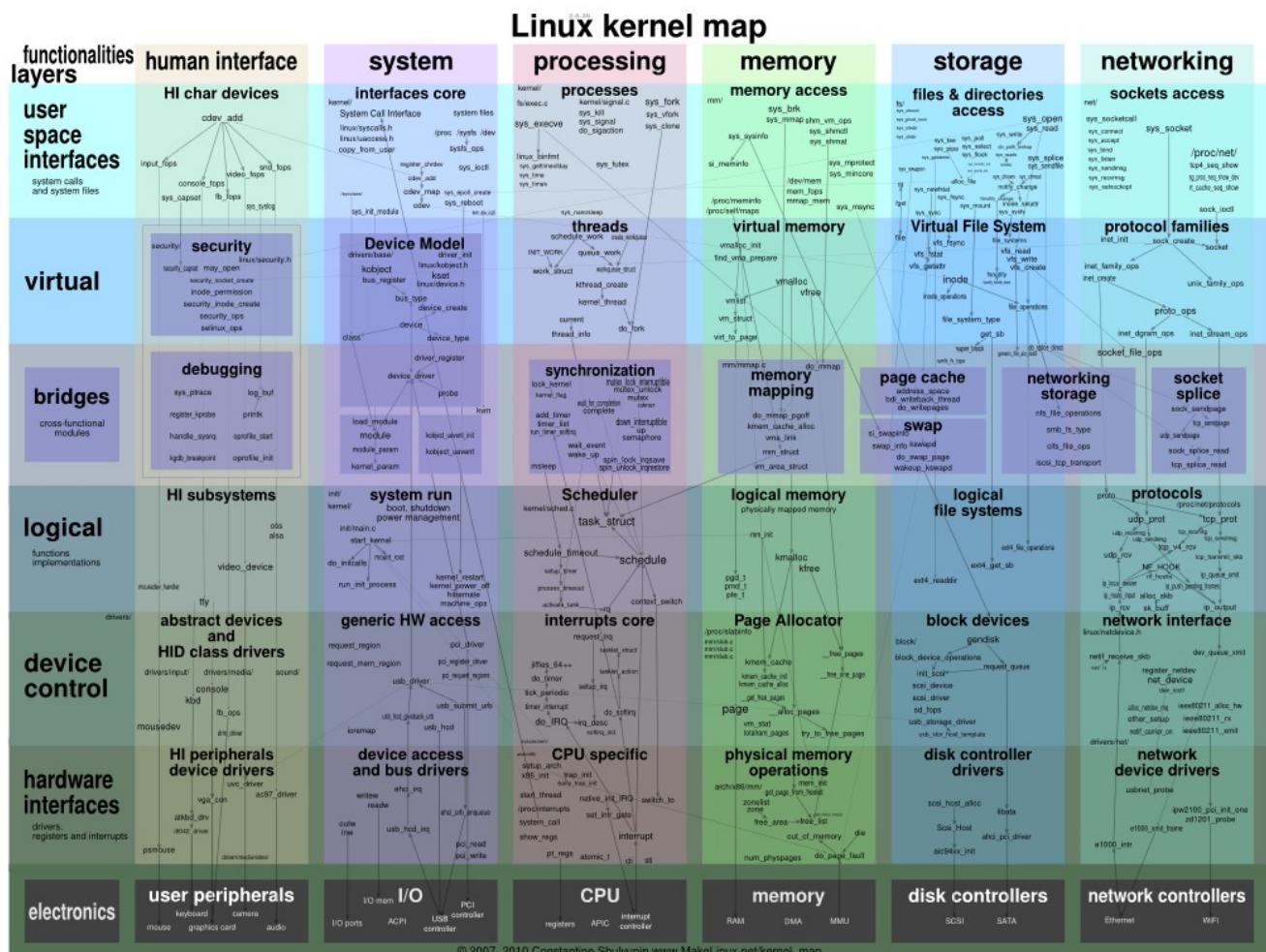


Рис. 1.3. Ядро Linux, [http://www.makelinux.net/kernel\\_map/](http://www.makelinux.net/kernel_map/)

## Аппаратная архитектура

### Различные архитектуры:

- фон Неймана
- Гарвардская
- стековые машины

- Lisp Machine
- FPGA
- и другие

## Архитектура фон Неймана

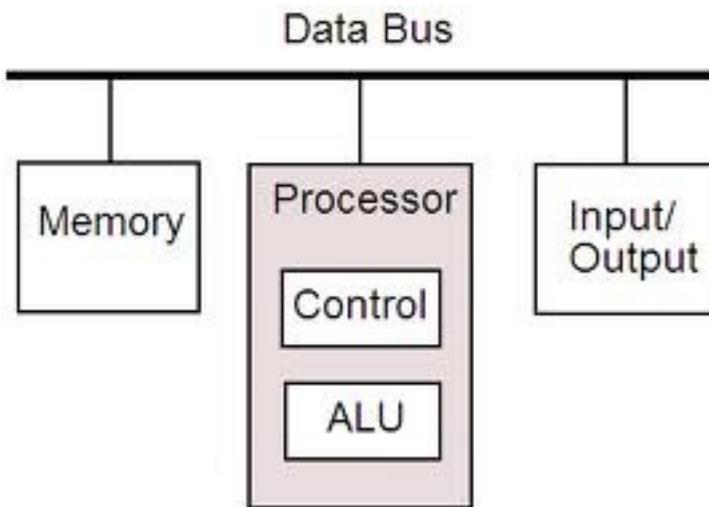


Рис. 1.4. Архитектура фон Неймана

## Принципы фон Неймана:

### Двоичного кодирования

Согласно этому принципу, вся информация, поступающая в ЭВМ, кодируется с помощью двоичных сигналов (двоичных цифр, битов) и разделяется на единицы, называемые словами.

### Однородности памяти

Программы и данные хранятся в одной и той же памяти. Поэтому ЭВМ не различает, что хранится в данной ячейке памяти — число, текст или команда. Над командами можно выполнять такие же действия, как и над данными.

### Адресуемости памяти

Структурно основная память состоит из пронумерованных ячеек; процессору в произвольный момент времени доступна любая ячейка. Отсюда следует возможность давать имена областям памяти, так, чтобы к

хранящимся в них значениям можно было бы впоследствии обращаться или менять их в процессе выполнения программы с использованием присвоенных имен.

Последовательного программного управления

Предполагает, что программа состоит из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности.

Жесткости архитектуры

Неизменяемость в процессе работы топологии, архитектуры, списка команд.

## Von Neumann Bottleneck:

Совместное использование шины для памяти программ и памяти данных приводит к узкому месту архитектуры фон Неймана, а именно ограничению пропускной способности между процессором и памятью по сравнению с объемом памяти. Из-за того, что память программ и память данных не могут быть доступны в одно и то же время, пропускная способность является значительно меньшей, чем скорость, с которой процессор может работать. Это серьезно ограничивает эффективное быстродействие при использовании процессоров, необходимых для выполнения минимальной обработки на больших объемах данных. Процессор постоянно вынужден ждать необходимых данных, которые будут переданы в память или из памяти. Так как скорость процессора и объем памяти увеличивались гораздо быстрее, чем пропускная способность между ними, узкое место стало большой проблемой, серьезность которой возрастает с каждым новым поколением процессоров.



Рис. 1.5. Иерархия памяти в компьютере

## Архитектура x86

[http://en.wikibooks.org/wiki/X86\\_Assembly/X86\\_Architecture](http://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture)

### Основные особенности:

- набор инструкций процессора (CICS), инструкции двух типов: арифметико-логические (ADD, AND, ...) и управляющие (MOV, JMP, ...)
- ограниченное количество регистров: несколько регистров общего назначения (A, B, C, D), размер которых равен длине машинного слова, и несколько специальных регистров (IP, FLAGS, ...). АЛУ процессора может работать только с регистрами общего назначения. УУ процессора занимается модификацией значений регистров или перемещением данных между регистрами и памятью

### Режимы работы процессора:

- реальный: прямая адресация памяти
- защищенный: косвенная адресация памяти с использованием модуля управления памятью (MMU)
- и другие вспомогательные

### Работа процессора:

```
while (1) {
```

```
execute_instruction(read_memory(IP));  
// IP - регистр-указатель инструкции  
}
```

## Виртуальная машина

Виртуальная машина — это (эффективный) изолированный дубликат реальной машины.

[Критерии эффективной виртуализации Попека-Голдберга](#)

### Возможные задачи:

- эмуляция различных архитектур
- реализация языка программирования
- увеличение переносимости кода
- исследования производительности ПО или новой компьютерной архитектуры
- оптимизации использования ресурсов компьютеров
- защита информации и ограничение возможностей программ (песочница)
- внедрение вредоносного кода для управления инфицированной системой
- моделирование информационных систем различных архитектур на одном компьютере
- упрощение администрирования

### Типы:

- Системная (гипервизор)
- Процессная

### Типы гипервизоров:

- автономный
- на основе базовой ОС
- гибридный

## POSIX

POSIX - Portable Operating System Interface for Unix — Переносимый интерфейс операционных систем Unix — набор стандартов, описывающих интерфейсы

между операционной системой и прикладной программой.

Открытые стандарты — это стандарты, которые публикуются в открытых источниках и, как правило, имеют одну или несколько (часто обязательным является наличие минимум двух) эталонных реализаций (*reference implementation*). Также, как правило, такие стандарты разрабатываются в рамках четко определенного процесса.

Интерфейс — совокупность правил (описаний, соглашений, протоколов), обеспечивающих взаимодействие устройств и программ в вычислительной системе или сопряжение между системами. Это внешнее представление, абстракция какого-то информационного объекта. Интерфейс разделяет методы внешнего взаимодействия и внутренней работы. Один объект может иметь несколько интерфейсов для разных "потребителей". Интерфейс — это средство трансляции между сущностями внешней и внутренней для объекта среды. Интерфейс — это форма косвенного взаимодействия. Связанно с концепцией кибернетики "черный ящик".

Протокол — набор соглашений интерфейса логического уровня, которые определяют обмен данными между различными программами. Эти соглашения задают единообразный способ передачи сообщений и обработки ошибок при взаимодействии программного обеспечения разнесённой в пространстве аппаратуры, соединённой тем или иным интерфейсом. Это набор правил взаимодействия между объектами. Эти правила определяют синтаксис, семантику и синхронизацию взаимодействия. Протокол может существовать в форме конвенции (неформального) или стандарта (формального набора правил).

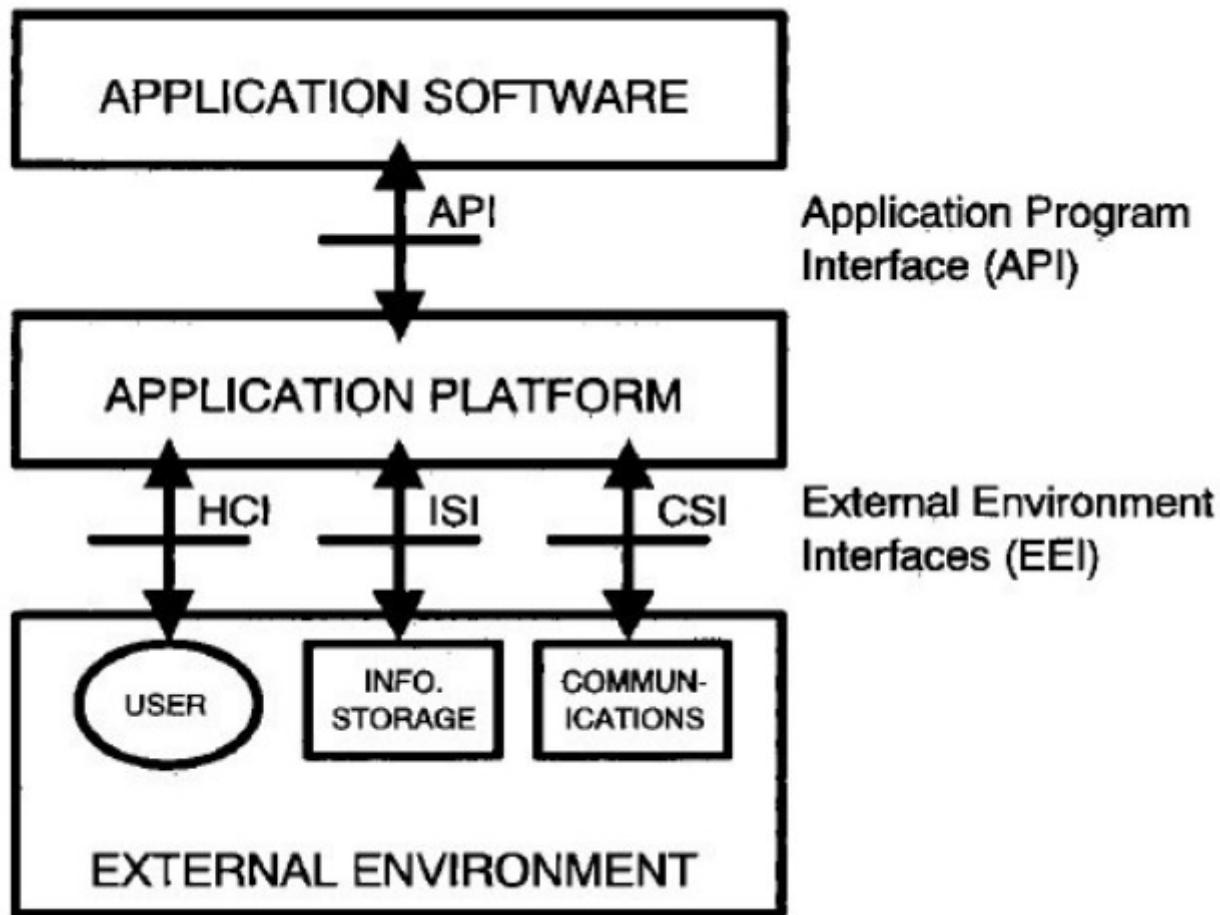


Рис. 1.6. Модель POSIX

## Задачи POSIX:

- содействовать облегчению переноса кода прикладных программ на иные платформы
- способствовать определению и унификации интерфейсов заранее при проектировании, а не в процессе их реализации
- сохранить по возможности и учитывать все главные, созданные ранее и используемые прикладные программы
- определять необходимый минимум интерфейсов прикладных программ, для ускорения создания, одобрения и утверждения документов
- развивать стандарты в направлении обеспечения коммуникационных сетей, распределенной обработки данных и защиты информации
- рекомендовать ограничение использования бинарного (объектного) кода для приложений в простых системах

## Принципы открытой системы:

- переносимость приложений (на уровнях: кода и программы), данных и персонала
- интероперабельность
- расширяемость
- масштабируемость

## Фольклор

Системное программирование — одна из старейших областей компьютерной инженерии. Поэтому она включает в себя не только формальные знания, такие как алгоритмы, стандарты и результаты исследований, но также и накопленные неформальные, культурные и социальные знания.

## Примеры:

Закон Постела (принцип здравости) — относится к организации взаимодействия между системами. Один из принципов, лежащих в основе Интернета

Будьте консервативны в том, что отправляете, и либеральны в том, что принимаете.

Закон конвертирования программ Завинского — описывает развитие любой сложной программной системы

Каждая программа пытается расширяться до тех пор, пока не сможет читать почту. Те программы, которые не могут этого сделать, заменяются теми, которые могут.

Десятое правило программирования Гринспена — описывает развитие любой сложной программной системы, основанной на статических языках

Любая достаточно сложная программа на C или Fortran содержит реализацию половины Common Lisp, которая является ad hoc, неформально-специфицированной, полной багов и медленной.

# Литература

- [A Crash Course in Modern Hardware](#)
- [The C Programming Language](#) или [Learn C The Hard Way](#)
- [The Unix Programming Environment](#)
- [Tanenbaum–Torvalds debate](#)

# Взаимодействие с аппаратной частью

## Обзор работы ОС с аппаратной частью

ОС реализуется поверх аппаратной архитектуры, которая определяет способ взаимодействия и набор ограничений. Взаимодействие происходит напрямую — через выдачу инструкций процессору, — и косвенно — через обработку прерываний устройств. С момента начала загрузки ОС указатель на текущую инструкцию процессора (регистр IP) устанавливается в начальную инструкцию исполняемого кода ОС, после чего управление работой процессора переходит к ОС.

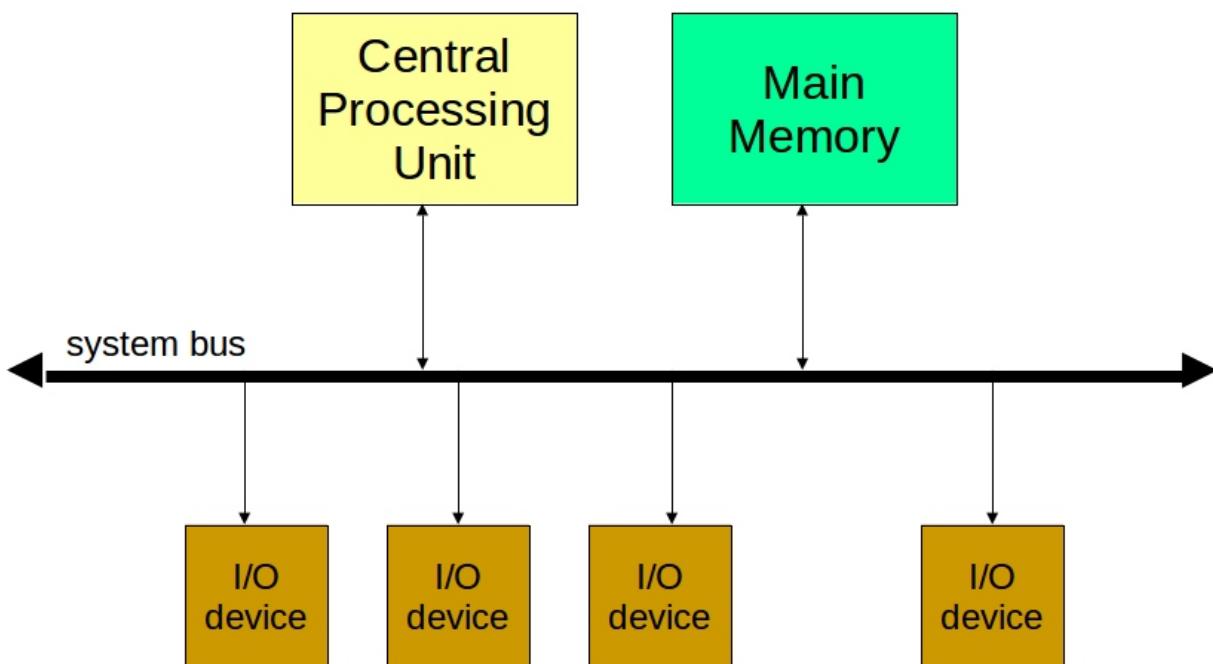


Рис. 2.1. Общий вид аппаратной архитектуры

Взаимодействие процессора с внешними устройствами (также называемое вводом-выводом) возможно только через адресуемую память. Существуют 2 схемы передачи данных между памятью и устройствами: PIO (Programmed IO — ввод-вывод через процессор) и DMA (Direct Memory Access — прямой доступ к памяти). В основном используется второй вариант, который полагается на отдельный контроллер, что позволяет разгрузить процессор от управления вводом-выводом и таким образом ускорить работу системы в целом.

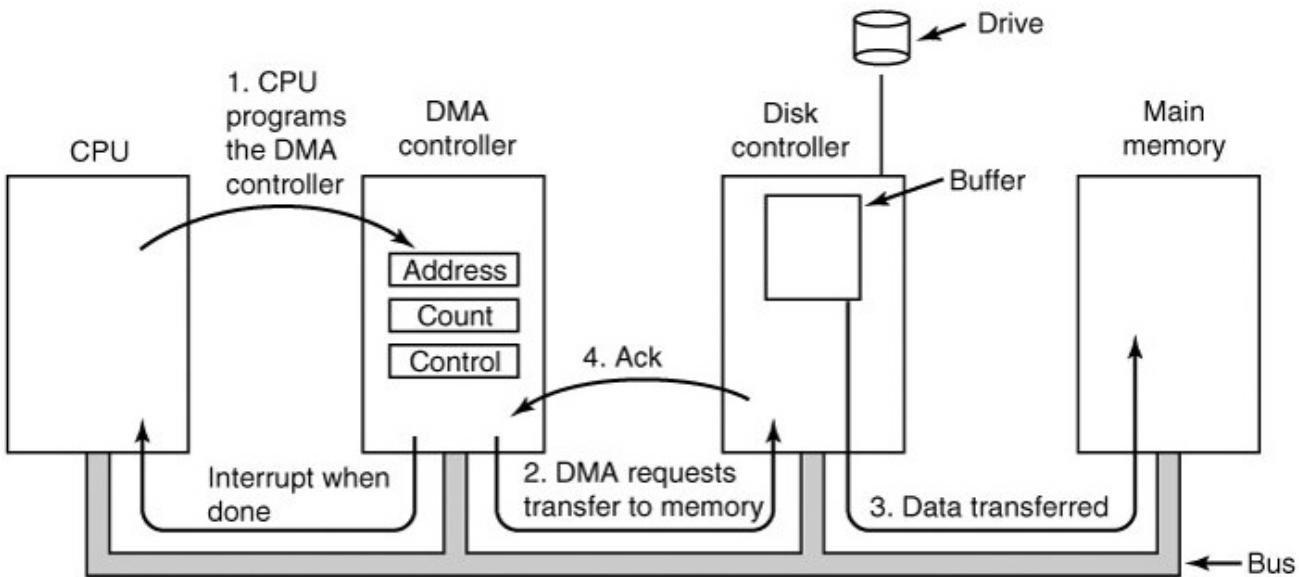


Рис. 2.2. Схема ввода-вывода через DMA

## Драйверы устройств

Драйвер устройства — это компьютерная программа, которая реализует механизм управления устройством и позволяет программам более высокого уровня взаимодействовать с конкретным устройством, не зная его команд и других параметров функционирования. Драйвер осуществляет свои функции посредством команд контроллера устройства и обработки прерываний, приходящих от него. Как правило, драйвер реализуется как часть (модуль) ядра ОС, т.к.:

- обработка прерываний драйвером требует задействования функций ОС
- справедливая и эффективная утилизация устройства требует участия ОС
- посылка неверных команд или их последовательностей, а также несоблюдение других условий работы с устройством может вывести его из строя

Драйверы устройств делятся на 3 основных класса:

- Символьные — работают с устройствами, позволяющими передавать данные по 1 символу (байту): как правило, различные консоли, модемы и т.п.
- Блочные — работают с устройствами, позволяющими осуществлять буферизированный ввод-вывод: например, различными дисковыми накопителями
- Сетевые

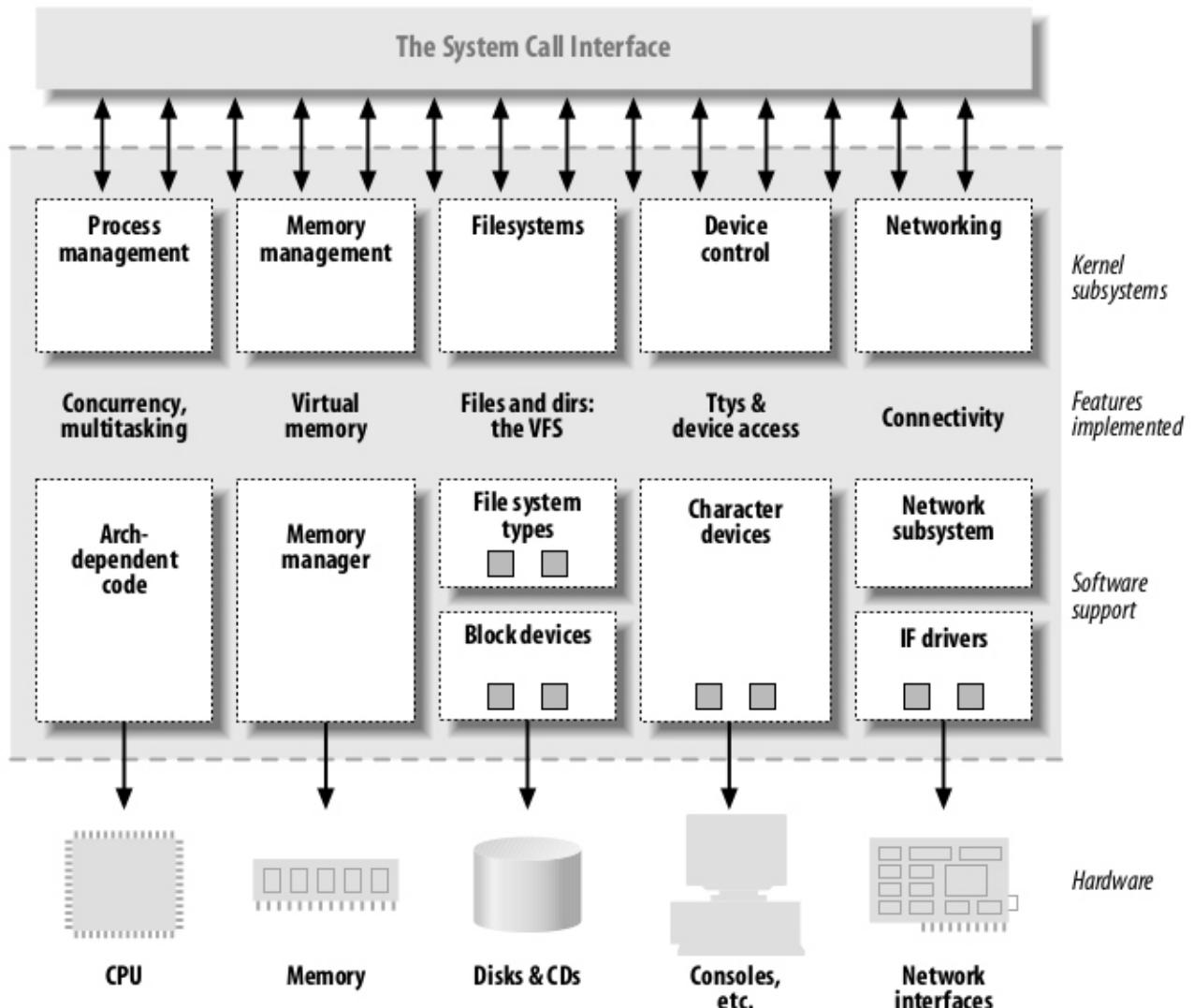


Рис. 2.3. Классы драйверов

Драйверы отдельных устройств объединяются ОС в классы, которые предоставляют единообразный абстрактный интерфейс программам более высокого уровня. В общем этот интерфейс называется Уровнем абстракции оборудования (Hardware abstraction layer, HAL).

## Время в компьютере

Для работы ОС использует аппаратный таймер, который работает на заданной тактовой частоте (на данный момент, как правило 100 Гц). В Linux 1 цикл такого таймера называется *jiffies*. При этом современные процессоры работают на тактовой частоте порядка ГГц, взаимодействие с памятью происходит с частотой порядка десятков МГц, с диском и сетью — порядка сотен КГц. В целом, это создает определенную иерархию операций в компьютере по порядку времени, требуемого для их выполнения. Эффективная работа ядра ОС

основывается на знании о том, какие операции допустимы на каком из уровней иерархии.

# Прерывания

## Аппаратные прерывания

Все операции ввода-вывода требуют продолжительного времени на свое выполнение, поэтому выполняются в асинхронном режиме. Т.е. после выполнения инструкции, вызывающей ввод-вывод, процессор не ждет его завершения, а переключается на выполнение других инструкций. Когда ввод-вывод завершается, устройство сигнализирует об этом посредством прерывания. Такое прерывание называется аппаратным (жестким) или же асинхронным.

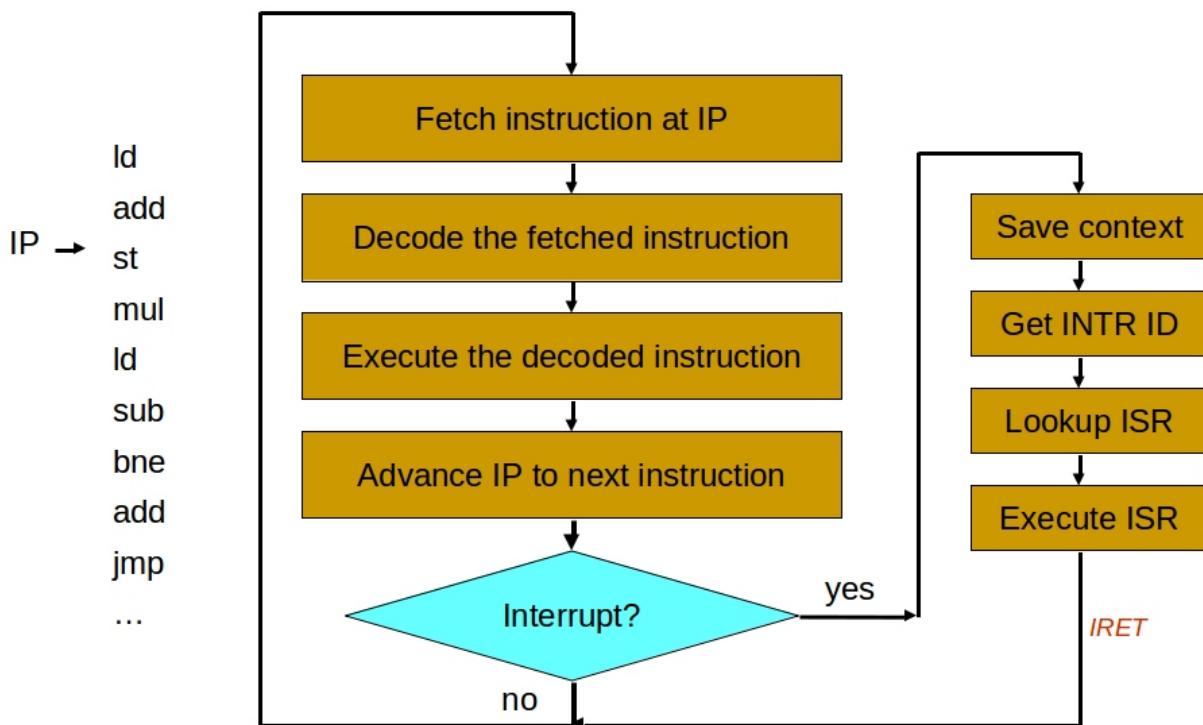


Рис. 2.4. Алгоритм работы процессора

В общем, **прерывание** — это сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

PIC (Programmable Interrupt Controller) — программируемый контроллер

прерываний) — это специальное устройство, которое обеспечивает сигнализацию о прерываниях процессору.

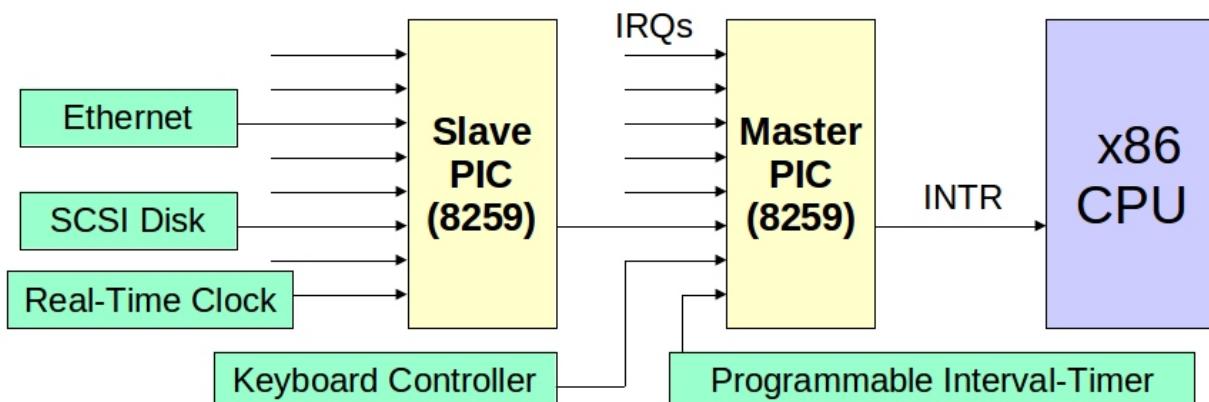


Рис. 2.5. Пример реализации PIC

## Программные прерывания

Помимо асинхронных прерываний процессоры поддерживают также синхронные прерывания двух типов:

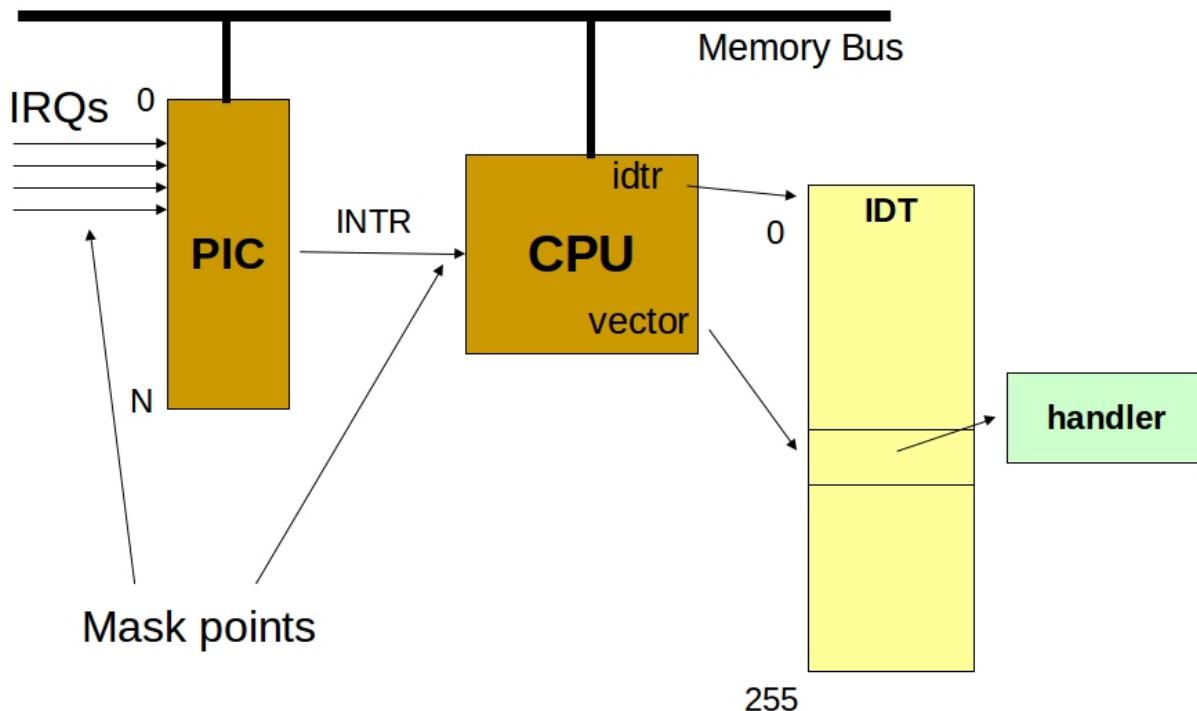
- Исключения (Exceptions): ошибки (fault) — предполагают возможность восстановления, ловушки (trap) — сигналы, которые посылаются после выполнения команды и используются для остановки работы процессора (например, при отладке), и сбои (abort) — не предполагают восстановления
- Программируемые прерывания

В архитектуре x86 предусмотрено 256 синхронных прерываний, из которых первые 32 — это зарезервированные исключения, остальные могут быть произвольно назначены ОС. Примеры стандартных исключений в архитектуре x86 с их номерами:

```

0: divide-overflow fault
6: Undefined Opcode
7: Coprocessor Not Available
11: Segment-Not-Present fault
12: Stack fault
13: General Protection Exception
14: Page-Fault Exception
  
```

## Схема обработки прерываний



255

Рис. 2.6. Общая схема системы обработки прерываний

Каждое прерывание имеет уникальный номер, который используется как смещение в таблице обработчиков прерываний. Эта таблица хранится в памяти компьютера и на ее начало указывает специальный регистр процессора - IDT (Interrupt Descriptor Table).

При поступлении сигнала о прерывании его нужно обработать как можно быстрее, для того, чтобы дать возможность производить ввод-вывод другим устройствам. Поэтому процессор сразу переключается в режим обработки прерывания. По номеру прерывания процессор находит процедуру-обработчик в таблице IDT и передает ей управление. Обработчик прерывания, как правило, разбит на 2 части: верхнюю (top half) и нижнюю (bottom half).

Верхняя часть выполняет только тот минимальный набор операций, который необходим, чтобы передать управление дальше. Этот набор включает:

- подтверждение прерывания (которое разрешает приход новых прерываний)
- точное определение устройства, от которого пришло прерывание
- инициализация процедуры обработки нижней части и постановка ее в очередь на исполнение

Процедура нижней части обработчика выполняет копирование данных из буфера устройства в память.

## Контексты

Обработчики прерываний работают в т.н. **атомарном контексте**. Переключение контекста — это процесс сохранения состояния регистров процессора в память и установки новых значений для регистров. Можно выделить как минимум 3 контекста:

- Атомарный, который в свою очередь часто разбит на контекст обработки аппаратных прерываний и программных. В атомарном контексте у процессора нет информации о том, какая программа выполнялась до этого, т.е. нет связи с пользовательской средой. В атомарном контексте нельзя вызывать блокирующие операции, в том числе sleep.
- Ядерный — контекст работы функций самого ядра ОС.
- Пользовательский — контекст работы функций пользовательской программы, из которого нельзя получить доступ к памяти ядра.

## Домены безопасности



Рис. 2.7. Кольца процессора

Для поддержки разграничения доступа к критическим ресурсам большинство архитектур реализуют концепцию **доменов безопасности** или же **колец процессора** (CPU Rings). Вся память компьютера промаркована номером кольца безопасности, к которому она относится. Инструкции, находящиеся в памяти, относящейся к тому или иному кольцу, в качестве operandов могут использовать только адреса, которые относятся к кольцам по номеру не более

данного. Т.е. программы в кольце 0 имеют максимальные привилегии, а в наибольшем по номеру кольце — минимальные.

На x86 архитектуре колец 4: от 0 до 3. ОС с монолитным или модульным ядром (такие, как Linux) загружаются в кольцо 0, а пользовательские программы — в кольцо 3. Остальные кольца у них не задействованы. В случае микроядерных архитектур некоторые подсистемы ОС могут загружаться в кольца 1 и/или 2.

В соответствии с концепцией доменов безопасности все операции работы с устройствами могут выполняться только из круга 0, т.е. они реализованы в коде ОС и предоставляются пользовательским программам через интерфейс системных вызовов.

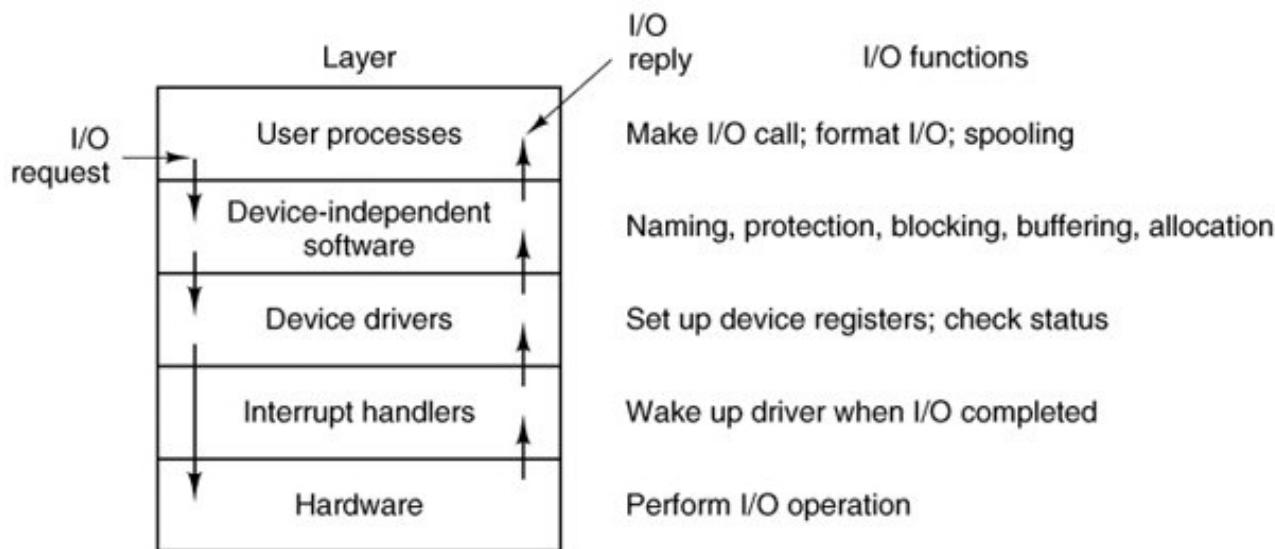


Рис. 2.8. Уровни обработки ИО-запроса

Ввод-вывод является продолжительной операцией по шкале процессорного времени. Поэтому оно блокирует вызвавший его процесс для того, чтобы не вызывать простоя процессора.

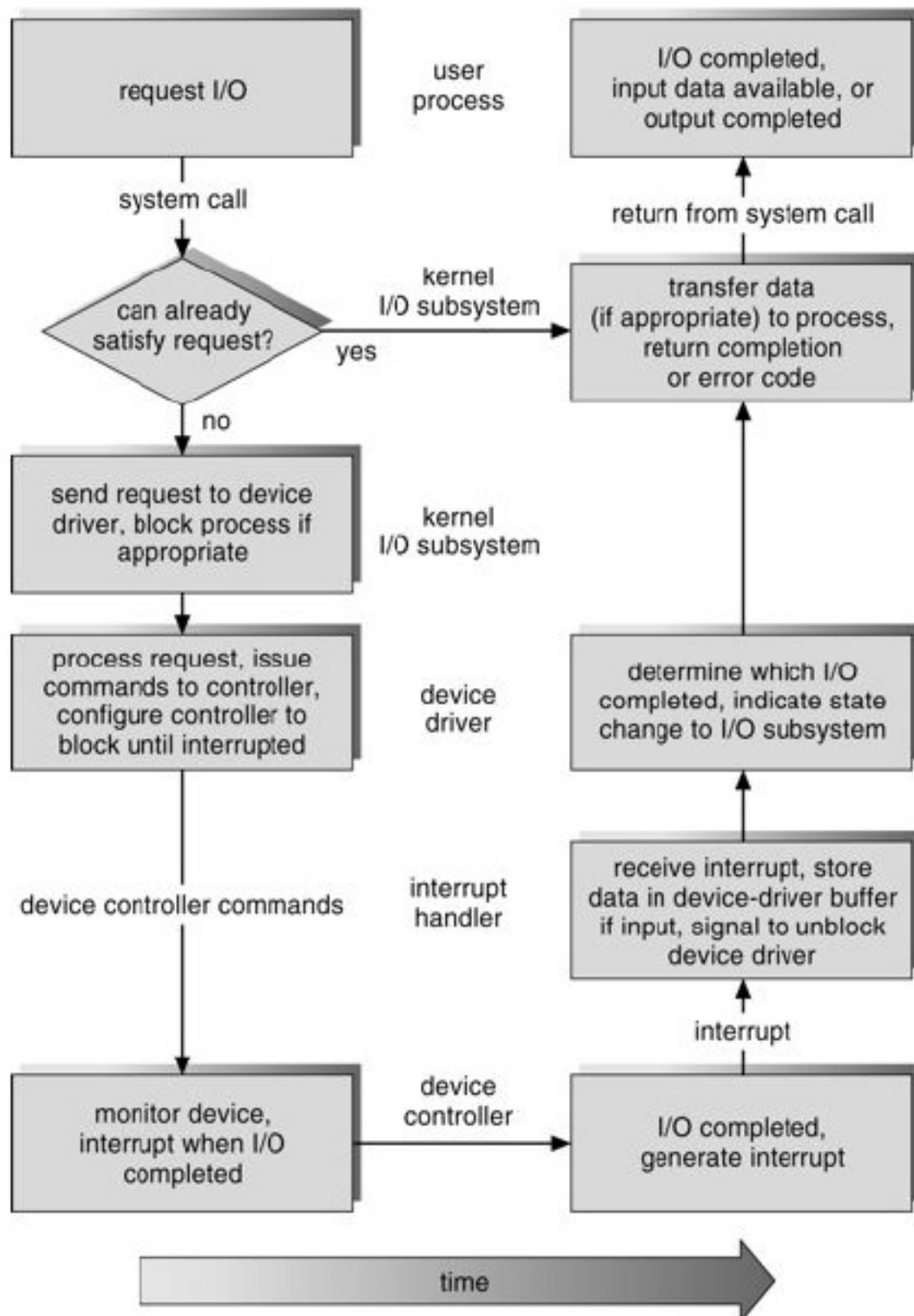


Рис. 2.9. Алгоритм ввода-вывода

# Загрузка

Загрузка (bootstrapping) — букв. вытягивание себя за собственные шнурки — процесс многоступенчатой инициализации ОС в памяти компьютера, который проходит через такие этапы:

1. Инициализация прошивки (firmware).
2. Выбор ОС-кандидата на загрузку.
3. Загрузка ядра ОС. На компьютерах с архитектурой x86 этот этап состоит из двух подэтапов:
  1. Загрузка в реальном режиме процессора.
  2. Загрузка в защищенном режиме.
4. Загрузка компонентов системы в пользовательском окружении.

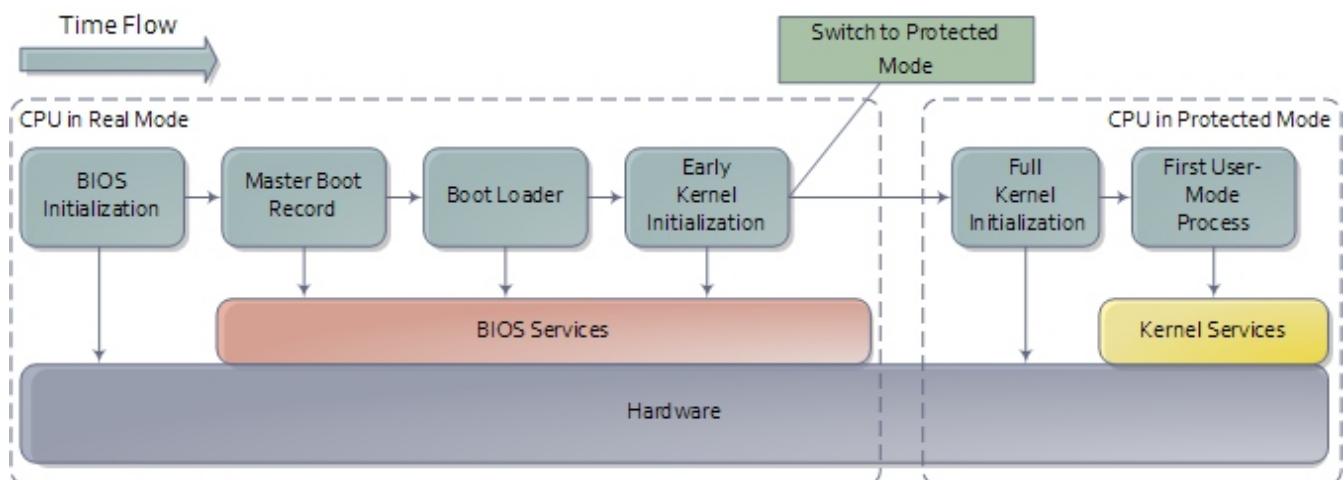


Рис. 2.10. Процесс загрузки ОС на архитектуре x86

## Прошивка

Прошивка (Firmware) — это программа, записанная в ROM-память компьютера. На компьютерах общего назначения прошивка выполняет функцию инициализации аппаратной части и передачи управления ОС. Распространенными интерфейсами прошивок являются варианты BIOS, OpenBootProm и EFI.

BIOS — это стандартный для x86 интерфейс прошивки. Он имеет множество исторических ограничений.

Алгоритм загрузки из BIOS:

1. Power-On Self-Test (POST) — тест работоспособности процессора и

памяти после включения питания.

2. Проверка дисков и выбор загрузочного диска.
3. Загрузка Главной загрузочной записи (Master Boot Record, MBR) загрузочного диска в память и передача управления этой программе. MBR может содержать от 1 до 4 записей о разделах диска.
4. Выбор загрузочного раздела. Загрузка загрузочной программы (т.н. 1-й стадии загрузчика) из загрузочной записи выбранного раздела.
5. Выбор ОС для загрузки. Загрузка загрузочной программы самой ОС (т.н. 2-й стадии загрузчика).
6. Загрузка ядра ОС в реальном режиме работы процессора. Большинство современных ОС не могут полностью загрузиться в реальном режиме (из-за жестких ограничений по памяти: для ОС доступно меньше 1МБ памяти). Поэтому загрузчик реального режима загружает в память часть кода ОС и переключает процессор в защищенный режим.
7. Окончательная загрузка ядра ОС в защищенном режиме.

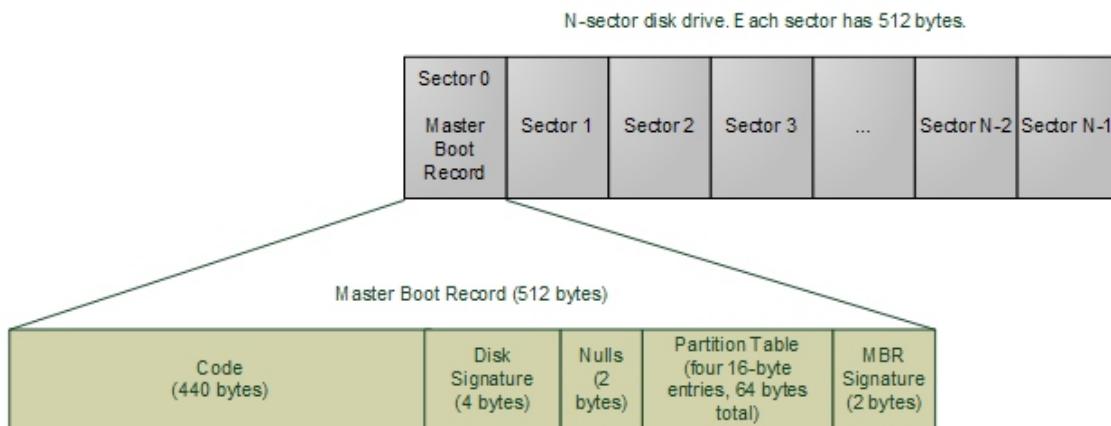


Рис. 2.11. Главная загрузочная запись (MBR)

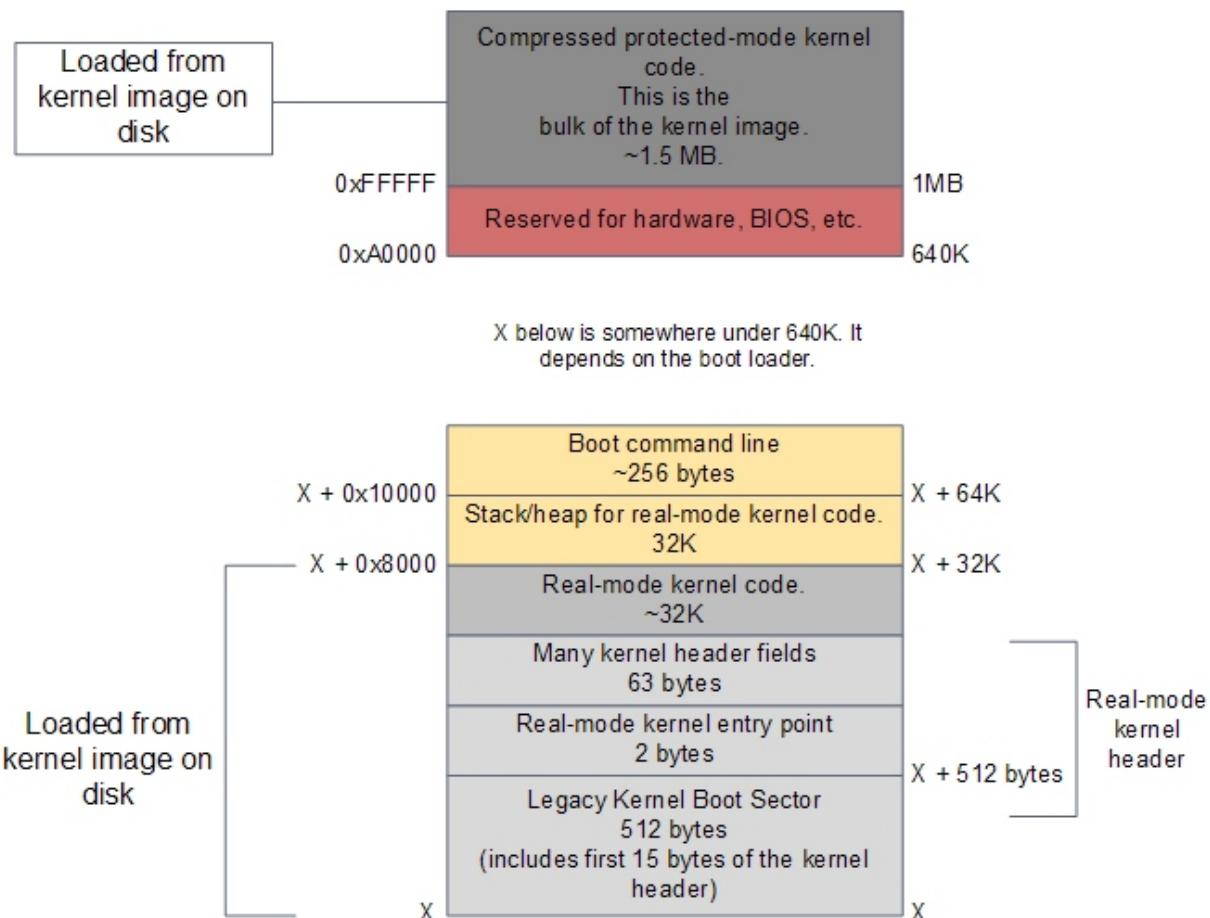


Рис. 2.12. Память компьютера после загрузки ОС

## Процесс init

После завершения загрузки ядра ОС, запускается первая программа в пользовательском окружении. В основанных на Unix системах это процесс номер 0, который называется *idle*. Концептуально этот процесс работает так:

```
while (1) {
    ; // do nothing
}
```

Такой процесс нужен, потому что процессором должны постоянно выполняться какие-то инструкции, он не может просто ждать.

Процессом номер 1 в Unix-системах является процесс *init*, который запускает все сервисы ОС, которые работают в пользовательском окружении. Это такие сервисы, как графическая оболочка, различные сетевые сервисы, сервис периодического выполнения задач (*cron*) и др. Конфигурация этих сервисов и

последовательности их загрузки выполняется с помощью shell-скриптов, находящихся в директориях /etc/init.d/, /etc/rc.d/ и др.

## Литература

- [PC Architecture](#)
- [Interrupts and Exceptions](#)
- [Interrupt Handling Contexts](#)
- [Jiffies](#)
- [Latency Numbers Every Programmer Should Know](#)
- Software Illustrated series by Gustavo Duarte:
  - [CPU Rings, Privilege, and Protection](#)
  - [How Computers Boot Up](#)
  - [The Kernel Boot Process](#)
- [Linux Device Drivers, Third Edition](#)
- [The Linux Kernel Driver Model: The Benefits of Working Together](#)
- [Writing Device Drivers in Linux](#)
- [Another Level of Indirection](#)
- [x86 DOS Boot Sector Written in C](#)

# Бинарный интерфейс

## Бинарный интерфейс приложений (ABI)

Бинарный интерфейс приложений — набор формальных спецификаций и неформальных соглашений в рамках программной платформы, обеспечивающих исполнение программ на платформе. Бинарному интерфейсу должны следовать все программы, однако вся работа по его реализации ложится на компилятор и, как правило, прозрачна для прикладных программистов.

Бинарный интерфейс включает:

- спецификацию типов данных: размеры, формат, последовательность байт (endiannes)
- форматы исполняемых файлов
- соглашения о вызовах
- формат и номера системных вызовов
- и др.

# Ассемблер

Ассемблер — это низкоуровневый язык, позволяющий непосредственно кодировать инструкции программной платформы (ОС или виртуальной машины).

Отличия ассемблера от языков более высокого уровня:

- отсутствие единого стандарта, т.е. у каждой архитектуры свой ассемблер
- программа на Ассемблере довольно прямо отражается в бинарный код объектного (исполняемого) файла; обратное преобразование — из объектного файла в ассемблерный код называется дизассемблированием
- в ассемблере нет понятия переменных, а также управляющих конструкций: выполнение программы происходит за счет манипуляции данными в регистрах и памяти напрямую, а также вызова других инструкций процессора
- отсутствие каких-либо проверок целостности данных (например, проверки типов)

Синтаксис ассемблера составляют:

- литералы — константные значения, которые представляют сами себя (числа, адреса, строки, регистры)
- команды — мнемоники для записи соответствующих инструкций процессора
- метки — имена для адресов памяти
- директивы компилятора — инструкции компилятору, описывающие различные аспекты создания из программы исполняемого файла (разрядность и секции программы, экспортируемые символы и т.д.) — не записываются напрямую в исполняемую программу
- макросы — конструкции для записи повторяющихся блоков кода, в результате выполнения которых на этапе компиляции выполняется подстановка этих кусков кода вместо имени макроса

Общепринятыми являются 2 синтаксиса ассемблера:

- AT&T
- Intel

Наиболее распространенные ассемблеры для архитектуры x86: NASM, GAS, MASM, TASM. Часть из них являются кросс-платформенными, т.е. работают в разных ОС, а часть — только в какой-либо одной ОС или группе ОС.

## Регистры процессора

Команды ассемблера позволяют напрямую манипулировать регистрами процессора. Регистр — это небольшой объем очень быстрой памяти (как правило, размером в 1 машинное слово), размещённой на процессоре. Он предназначен для хранения результатов промежуточных вычислений, а также некоторой информации для управления работой процессора. Так как регистры размещены непосредственно на процессоре, доступ к данным, хранящимся в них, намного быстрее доступа к данным в оперативной памяти.

Все регистры можно разделить на две группы: **пользовательские и системные**. Пользовательские регистры используются при написании "обычных" программ. В их число входят основные программные регистры, а также регистры математического сопроцессора, регистры MMX, XMM (SSE, SSE2, SSE3) и т.п. К системным регистрам относятся регистры управления, регистры управления памятью, регистры отладки, машинно-специфичные регистры MSR и другие.

## Адресация памяти

Ассемблером поддерживаются разные способы адресации памяти:

- непосредственная
- прямая (абсолютная)
- косвенная (базовая)
- автоинкрементная/автодекрементная
- регистровая
- относительная (в случае использования сегментной организации виртуальной памяти)

Порядок байтов (endianness) в машинном слове определяет последовательность записи байтов: от старшего к младшему (**big-endian**) или от младшего к старшему (**little-endian**).

## Стек

(Более правильное название используемой структуры данных — **стопка** или **магазин**. Однако, исторически прижилось заимствованное название стек).

Стек (stack) — это часть динамической памяти, которая используется при вызове функций для хранения ее аргументов и локальных переменных. В архитектуре x86 стек растет вниз, т.е. вершина стека имеет самый маленький адрес. Регистр SP (Stack Pointer) указывает на текущую вершину стека, а регистр BP (Base Pointer) указывает на т.н. базу, которая используется для разделение стека на логические части, относящиеся к одной функции — **фреймы** (кадры). Помимо обычных инструкций работы с памятью и регистрами (таких как `mov`), дополнительно для манипуляции стеком используются инструкции `push` и `pop`, которые заносят данные на вершину стека и забирают данные с вершины. Эти инструкции также осуществляют изменение регистра SP.

Как правило, в программах на высокоуровневых языках программирования нет кода для работы со стеком напрямую, а это делает за кадром компилятор, реализуя определенные соглашения о вызовах функций и способы хранения локальных переменных. Однако функция `alloca` библиотеки `stdlib` позволяет программно выделять память на стеке.

Вызов функции высокогоуровневого языка создает на стеке новый фрейм, который содержит аргументы функции, адрес возврата из функции, указатель на начало предыдущего фрейма, а также место под локальные переменные.

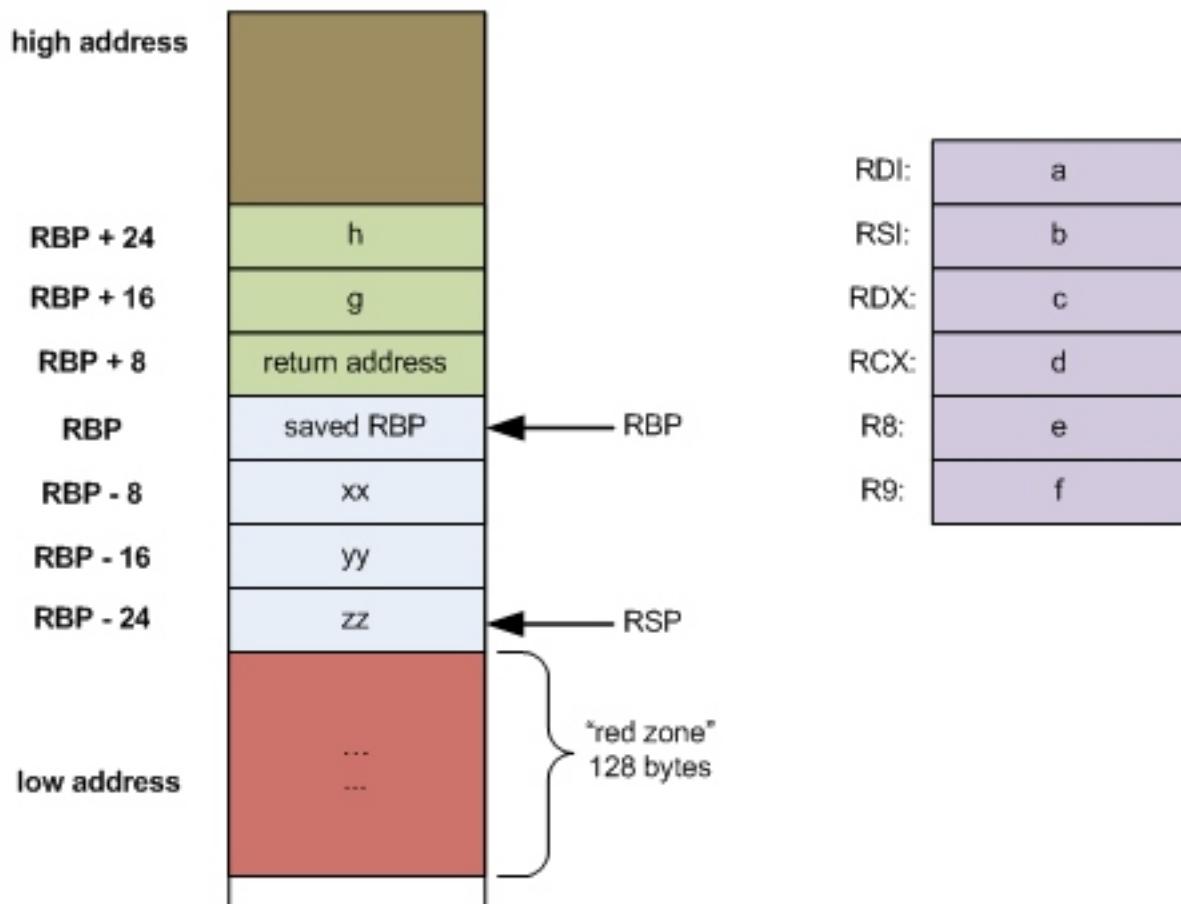


Рис. 3.1. Вид фрейма стека при вызове в рамках AMD64 ABI

В начале работы программы в стеке выделен только 1 фрейм для функции `main` и ее аргументов — числового значения `argc` и массива указателей переменной длины `argv`, каждый из которых записывается на стек по отдельности, а также переменных окружения.

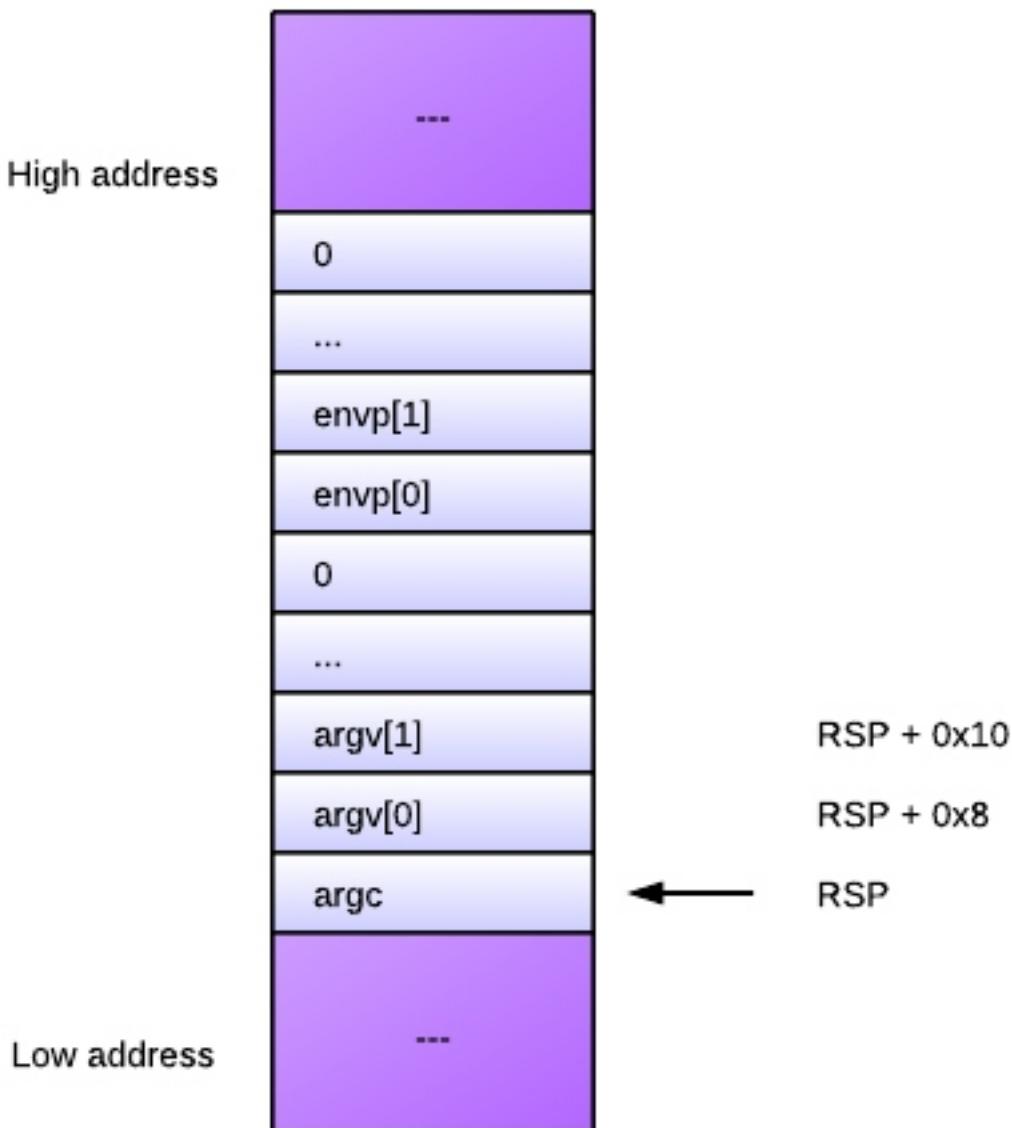


Рис. 3.2. Вид стека после вызова функции main

## Соглашения о вызовах

Соглашение о вызовах — это схема, в соответствии с которой вызов функции высокуровневого языка программирования реализуется в исполняемом коде. Соглашение о вызовах отвечает на вопросы:

- Как передаются аргументы и возвращаются значения? Они могут передаваться либо в регистрах, либо через стек, либо и так, и так (гибридная схема).
- Как распределяется работа (по манипуляции стеком вызовов) междузывающей и вызванной стороной?

В принципе, соглашения не являются жестким стандартом, и программа не обязана следовать тому или иному соглашению для собственных функций (поэтому программы на ассемблере не всегда следуют ему), однако компиляторы создают исполняемые файлы в соответствии с тем или иным соглашением. Кроме того, соглашения о вызовах для библиотечных функций может отличаться от соглашения для системных вызовов: например, аргументы системных вызовов могут передаваться через регистры, а библиотечных функций — через стек.

Распространенные соглашения:

- cdecl — общепринятое соглашение для программ на языке С в архитектуре IA32: параметры кладутся на стек справо-налево,зывающая функция отвечает за очистку стека после возврата из вызванной функции
- stdcall — стандартное соглашение для Win32: параметры кладутся на стек справо-налево, функция может использовать регистры EAX, ECX и EDX, вызванная функция отвечает за очистку стека перед возвратом
- fastcall — нестандартные соглашения, в которых передача одного или более параметров происходит через регистры для ускорения вызова
- pascal — параметры кладутся на стек слева-направо (противоположно cdecl) и вызванная функция отвечает за очистку стека перед возвратом
- thiscall — соглашение для C++
- safecall — соглашение для COM/OLE
- syscall
- optlink
- AMD64 ABI
- Microsoft x86 calling convention

Пример вызова в соответствии с соглашением cdecl:

Код на языке С:

```
// вызываемая функция
int callee(int a, int b, int c) {
    int d;
    d = a + b + c;
    ...
    return d;
}
//зывающая функция
```

```
int caller(void) {
    int rez = callee(1, 2, 3);
    return rez + 5;
}
```

Сгенерированный компилятором ассемблерный код:

```
// в вызывающей функции
pushl %ebp // сохранение указателя на предыдущий фрейм
movl %esp,%ebp
// запись аргументов в стек справа-налево
pushl $3
pushl $2
pushl $1
call callee
addl $12,%esp // очистка стека
addl $5,%eax // результат вызова – в регистре ЕАХ
leave
ret
// в вызванной функции callee(1, 2, 3)
subl $4,%esp // выделение места под переменную d
movl %eax,4(%ebp) // достаем аргумент a
movl %ecx,8(%ebp) // достаем аргумент b
addl %eax,%ecx // результат сложения остается в %eax
movl %ecx,12(%ebp) // достаем аргумент c
addl %eax,%ecx // результат сложения остается в %eax
movl (%esp),%eax // присваиваем значение d
// ...
movl %eax,(%esp) // записываем d в %eax
leave // эквивалент movl $ebp,$esp; pop $ebp
ret
```

## Системные вызовы

В общем, **системный вызов** — это произвольная функция, которая реализуется ядром ОС и доступна для вызова из пользовательской программы. При выполнении системного вызова происходит переключение контекста из пользовательского в ядерный. Поэтому на уровне команд процессора системный вызов выполняется не как обычный вызов функции (инструкция CALL), а с помощью программного прерывания (в Linux это прерывание номер 80) или же с помощью инструкции SYSENTER (более современный вариант).

Пример выполнения системного вызова write с помощью программного прерывания:

```

mov eax, 4          ; specify the sys_write function code
                    ; (from OS vector table)
mov ebx, 1          ; specify file descriptor stdout(1)
mov ecx, str        ; move start address of string message
                     ; to ecx register
mov edx, str_len    ; move length of message (in bytes)
int 80h             ; tell kernel to perform
                     ; the system call we just set up

```

Пример выполнения системного вызова write с помощью инструкции SYSENTER:

```

push str_len
push str
push 1
push 4
push ebp
mov ebp, esp
sysenter

```

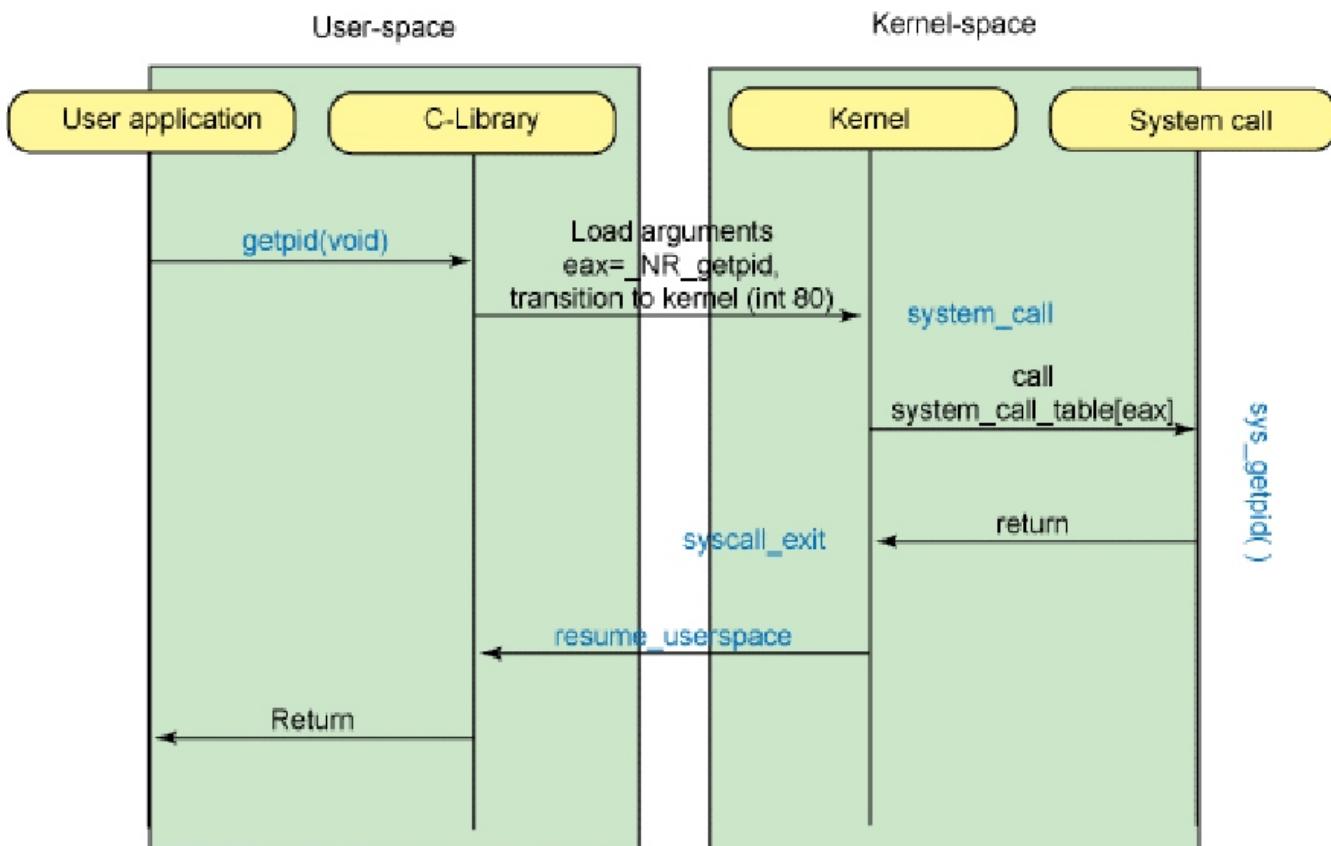


Рис. 3.3. Схема выполнения системного вызова

Стандартная библиотека libc реализует свои функции поверх системных

вызовов.

## Литература

- [Ассемблер в Linux для программистов С](#)
- [Ассемблеры для Linux: Сравнение GAS и NASM](#)
- [Why Registers Are Fast and RAM Is Slow](#)
- [x86 Registers](#)
- [Kernel command using Linux system calls](#)
- [The Linux Kernel: System Calls](#)
- [Sysenter Based System Call Mechanism in Linux 2.6](#)
- [Reverse Engineering for Beginners](#)

# Управление памятью

## Аппаратное управление памятью

Большинство компьютеров используют большое количество различных запоминающих устройств, таких как: ПЗУ, ОЗУ, жесткие диски, магнитные носители и т.д. Все они представляют собой виды памяти, которые доступны через разные интерфейсы. Два основных интерфейса — это прямая адресация процессором и файловые системы. Прямая адресация подразумевает, что адрес ячейки с данными может быть аргументом инструкций процессора.

Режимы работы процессора x86:

- реальный — прямой доступ к памяти по физическому адресу
- защищенный — использование виртуальной памяти и колец процессора для разграничения доступа к ней

## Виртуальная память

Виртуальная память — это подход к управлению памятью компьютером, который скрывает физическую память (в различных формах, таких как: оперативная память, ПЗУ или жесткие диски) за единым интерфейсом, позволяя создавать программы, которые работают с ними как с единым непрерывным массивом памяти с произвольным доступом.

Решаемые задачи:

- поддержка изоляции процессов и защиты памяти путём создания своего собственного виртуального адресного пространства для каждого процесса
- поддержка изоляции области ядра от кода пользовательского режима
- поддержка памяти только для чтения и с запретом на исполнение
- поддержка выгрузки не используемых участков памяти в область подкачки на диске (свопинг)
- поддержка отображённых в память файлов, в том числе загрузочных модулей
- поддержка разделяемой между процессами памяти, в том числе с копированием-при-записи для экономии физических страниц

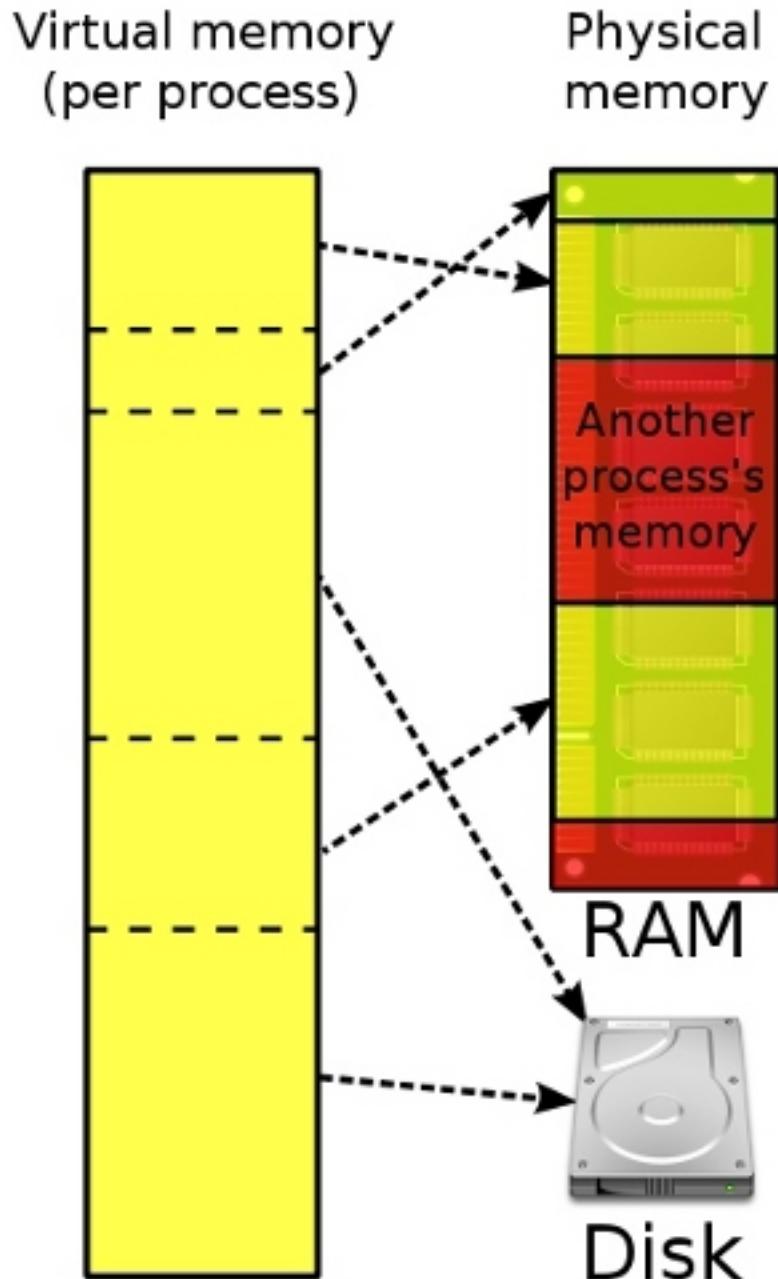


Рис. 4.1. Абстрактное представление виртуальной памяти

Виды адресов памяти:

- физический - адрес аппаратной ячейки памяти
- логический - виртуальный адрес, которым оперирует приложение

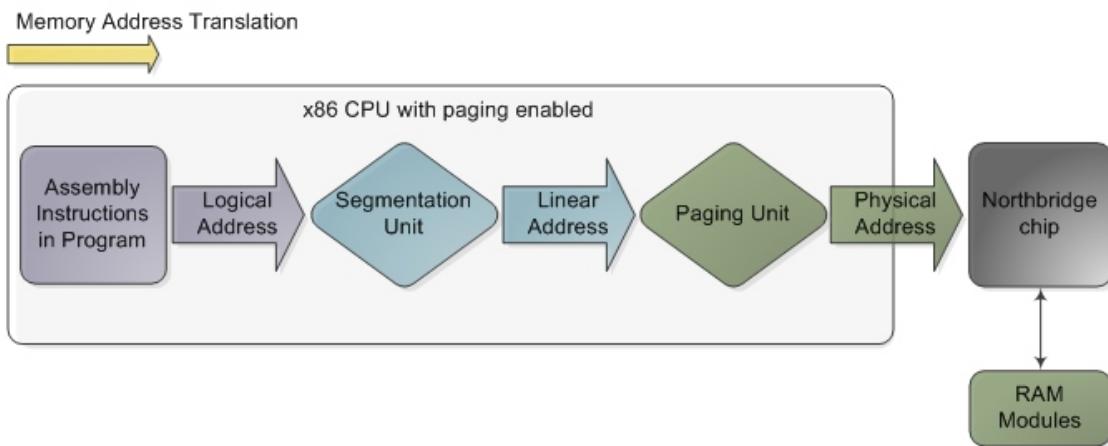


Рис. 4.2. Трансляция логического адреса в физический

За счет наличия механизма виртуальной памяти компиляторы прикладных программ могут генерировать исполняемый код в рамках упрощенной абстрактной линейной модели памяти, в которой вся доступная память представляется в виде непрерывного массива [машинных слов](#), адресуемого с 0 до максимально возможного адреса для данной разрядности ( $2^N$ , где N - количество бит, т.е. для 32-разрядной архитектуры максимальный адрес —  $2^{32} = \#FFFFFF$ ). Это значит что результирующие программы не привязаны к конкретным параметрам запоминающих устройств, таких как их объем, режим адресации и т.д.

Кроме того, этот дополнительный уровень позволяет через тот же самый интерфейс обращения к данным по адресу в памяти реализовать другие функции, такие как обращение к данным в файле (через механизм mmap) и т.д. Наконец, он позволяет обеспечить более гибкое, эффективное и безопасное управление памятью компьютера, чем при использовании физической памяти напрямую.

На аппаратном уровне виртуальная память, как правило, поддерживается специальным устройством — [Модулем управления памятью](#).

## Страницчная организация памяти

Страницчная память — способ организации виртуальной памяти, при котором единицей отображения виртуальных адресов на физические является регион постоянного размера — страница.

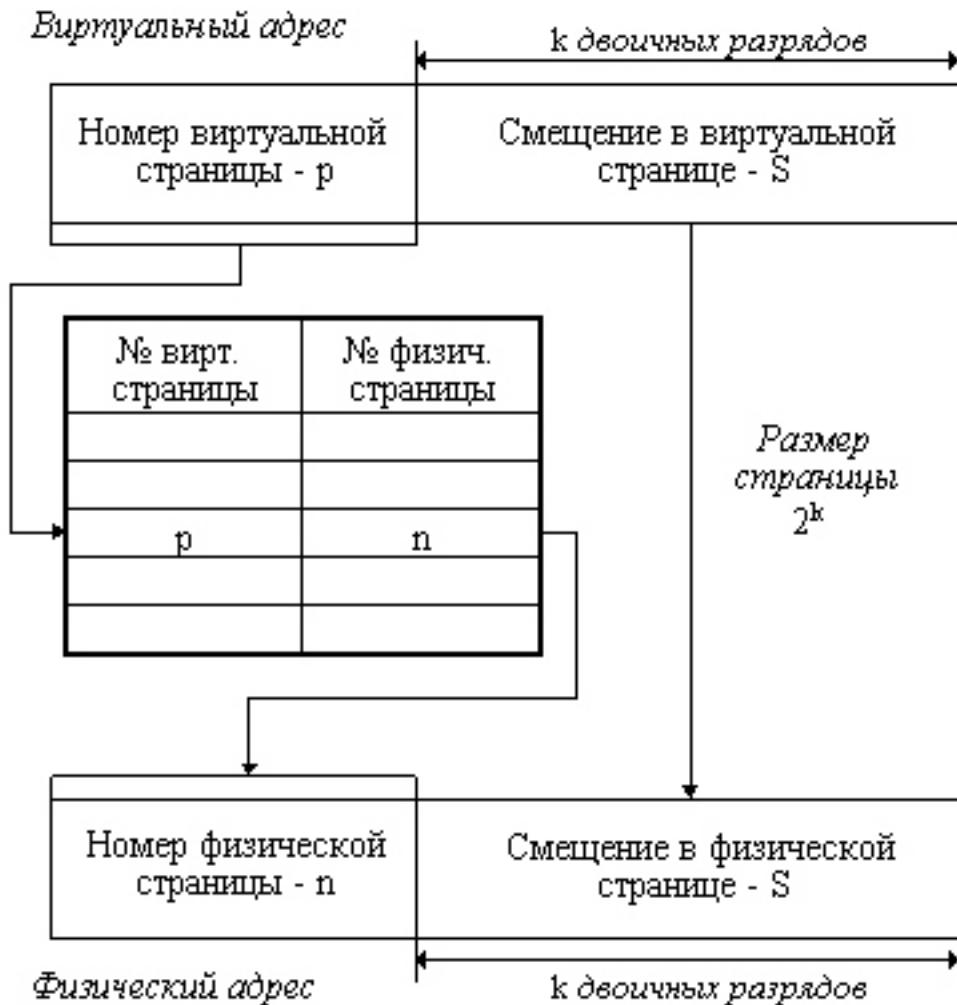


Рис. 4.3. Трансляция адреса в страничной модели

При использовании страничной модели вся виртуальная память делится на  $N$  страниц таким образом, что часть виртуального адреса интерпретируется как номер страницы, а часть — как смещение внутри страницы. Вся физическая память также разделяется на блоки такого же размера — **фреймы**. Таким образом в один фрейм может быть загружена одна страница. **Свопинг** — это выгрузка страницы из памяти на диск (или другой носитель большего объема), который используется тогда, когда все фреймы заняты. При этом под свопинг попадают страницы памяти неактивных на данный момент процессов.

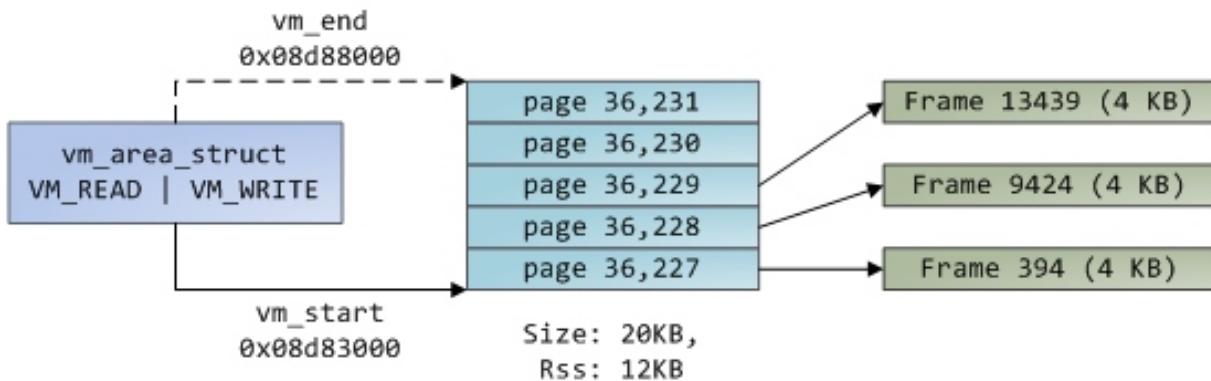


Рис. 4.4. Память процесса в страничной модели

Таблица соответствия фреймов и страниц называется таблицей страниц. Она одна для всей системы. Запись в таблице страниц включает служебную информацию, такую как: индикаторы доступа только на чтение или на чтение/запись, находится ли страница в памяти, производилась ли в нее запись и т.д. Страница может находиться в трех состояниях: загружена в память, выгружена в swap, еще не загружена в память (при изначальном выделении страницы она не всегда сразу размещается в памяти).

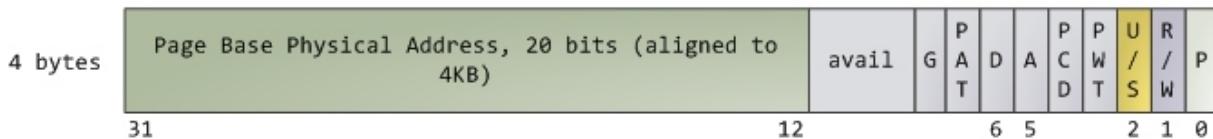


Рис. 4.5. Запись в таблице страниц

Размер страницы и количество страниц зависит от того, какая часть адреса выделяется на номер страницы, а какая на смещение. К примеру, если в 32-разрядной системе разбить адрес на две равные половины, то количество страниц будет составлять  $2^{16}$ , т.е. 65536, и размер страницы в байтах будет таким же, т.е. 64 КБ. Если уменьшить количество страниц до  $2^{12}$ , то в системе будет 4096 страниц по 1МБ, а если увеличить до  $2^{20}$ , то 1 миллион страниц по 4КБ. Чем больше в системе страниц, тем больше занимает в памяти таблица страниц, соответственно работа процессора с ней замедляется. А поскольку каждое обращение к памяти требует обращения к таблице страниц для трансляции виртуального адреса, такое замедление очень нежелательно. С другой стороны, чем меньше страниц и, соответственно, чем они больше по объему — тем больше потери памяти, вызванные внутренней фрагментацией страниц, поскольку страница является единицей выделения памяти. В этом заключается диллема оптимизации страничной памяти. Она особенно актуальна при переходе к 64-разрядным архитектурам.

Для оптимизации страничной памяти используются следующие подходы:

- специальный кеш — TLB (translation lookaside buffer) — в котором хранится очень небольшое число (порядка 64) наиболее часто используемых адресов страниц (основные страницы, к которым постоянно обращается ОС)
- многоуровневая (2, 3 уровня) таблица страниц — в этом случае виртуальный адрес разбивается не на 2, а на 3 (4,...) части. Последняя часть остается смещением внутри страницы, а каждая из остальных задает номер страницы в таблице страниц 1-го, 2-го и т.д. уровней. В этой схеме для трансляции адресов нужно выполнить не 1 обращение к таблице страниц, а 2 и более. С другой стороны, это позволяет сворачивать таблицы страниц 2-го и т.д. уровней, и подгружать в память только те таблицы, которые нужны текущему процессу в текущий момент времени или же даже кэшировать их. А каждая из таблиц отдельного уровня имеет существенно меньший размер, чем имела бы одна таблица, если бы уровень был один

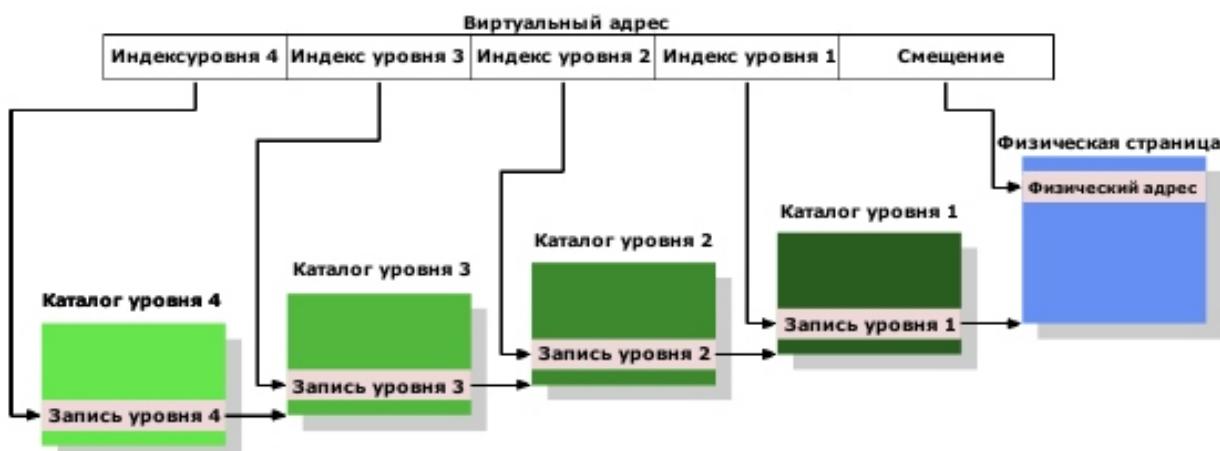


Рис. 4.6. Многоуровневая система страниц

- инвертированная таблица страниц — в ней столько записей, сколько в системе фреймов, а не страниц, и индексом является номер фрейма: а число фреймов в 64- и более разрядных архитектурах существенно меньше теоретически возможного числа страниц. Проблема такого подхода — долгий поиск виртуального адреса. Она решается с помощью таких механизмов как: хеш-таблицы или кластерные таблицы страниц

# Сегментная организация памяти

Сегментная организация виртуальной памяти реализует следующий механизм: вся память делиться на сегменты фиксированной или произвольной длины, каждый из которых характеризуется своим начальным адресом — **базой** или **селектором**. Виртуальный адрес в такой системе состоит из 2-х компонент: **базы** сегмента, к которому мы хотим обратиться, и **смещения** внутри сегмента.



Рис. 4.7. Представление сегментной модели виртуальной памяти

## Историческая модель сегментации в архитектуре x86

В архитектуре x86 сегментная модель памяти была впервые реализована на

16-разрядных процессорах 8086. Используя только 16 разрядов для адреса давало возможность адресовать только  $2^{16}$  байт, т.е. 64КБ памяти. В то же время стандартный размер физической памяти для этих процессоров был 1МБ. Для того, чтобы иметь возможность работать со всем доступным объемом памяти и была использована сегментная модель. В ней у процессора было выделено 4 специализированных регистра CS (сегмент кода), SS (сегмент стека), DS (сегмент данных), ES (расширенный сегмент) для хранения базы текущего сегмента (для кода, стека и данных программы).

Физический адрес в такой системе расчитывался по формуле:

```
addr = base << 4 + offset
```

Это приводило к возможности адресовать большие адреса, чем 1МБ — т.н. [Gate A20](#).

См. также: [http://en.wikipedia.org/wiki/X86\\_memory\\_segmentation](http://en.wikipedia.org/wiki/X86_memory_segmentation)

## Плоская модель сегментации

32-разрядный процессор 80386 мог адресовать  $2^{32}$  байт памяти, т.е. 4ГБ, что более чем перекрывало доступные на тот момент размеры физической памяти, поэтому изначальная причина для использования сегментной организации памяти отпала.

Однако, помимо особого способа адресации сегментная модель также предоставляет механизм защиты памяти через **кольца безопасности процессора**: для каждого сегмента в таблице сегментов задается значение допустимого уровня привилегий (DPL), а при обращении к сегменту передается уровень привилегий текущей программы (запрошенный уровень привилегий, RPL) и, если  $RPL > DPL$  доступ к памяти запрещен. Таким образом обеспечивается защита сегментов памяти ядра ОС, которые имеют  $DPL = 0$ . Также в таблице сегментов задаются другие атрибуты сегментов, такие как возможность записи в память, возможность исполнения кода из нее и т.д.

Таблица сегментов каждого процесса находится в памяти, а ее начальный адрес загружается в регистр LDTR процессора. В регистре GDTR процессора хранится указатель на глобальную таблицу сегментов.

В современных процессорах x86 используется "Плоская модель сегментации", в которой база всех сегментов выставлена в нулевой адрес.

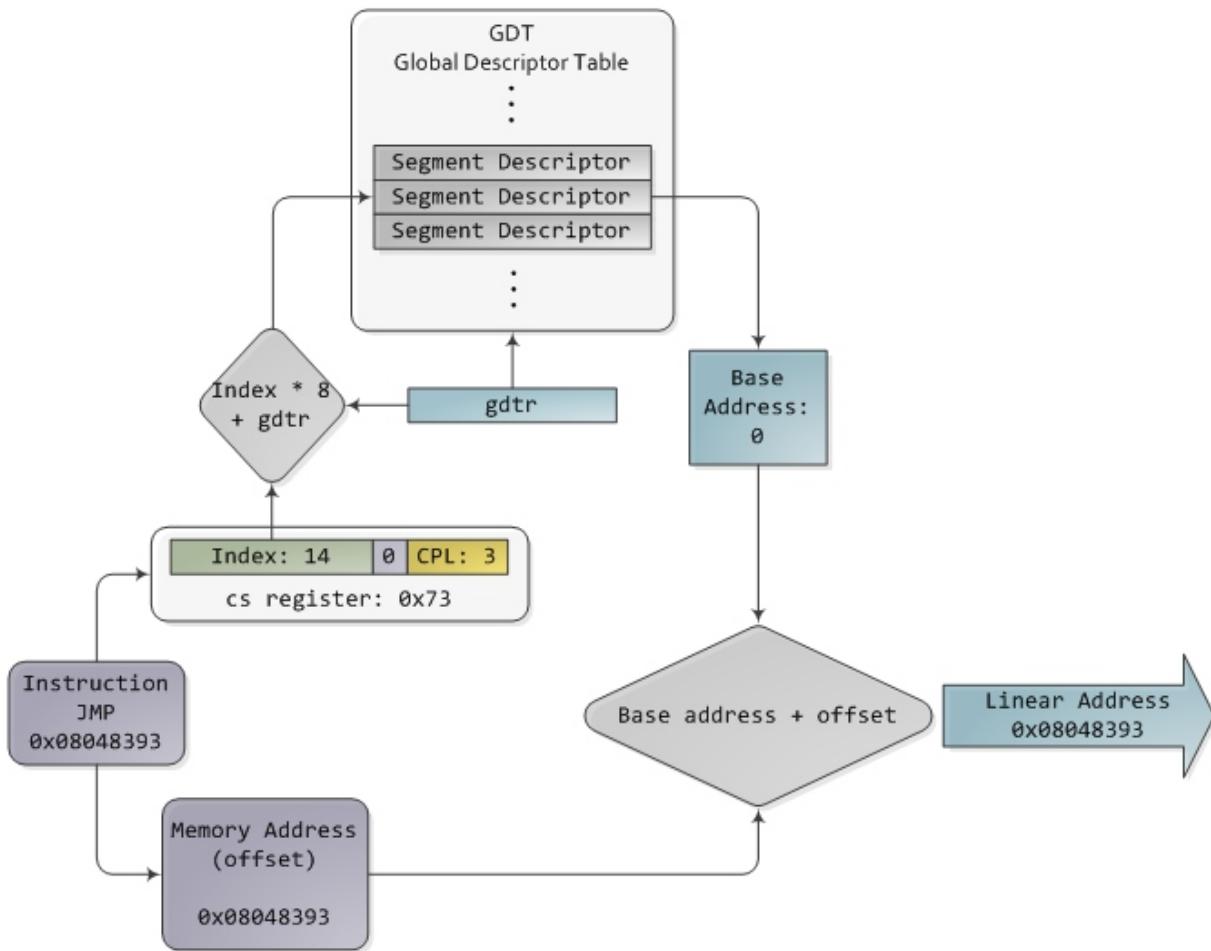


Рис. 4.8. Плоская модель сегментации

# Виртуальная память в архитектуре x86

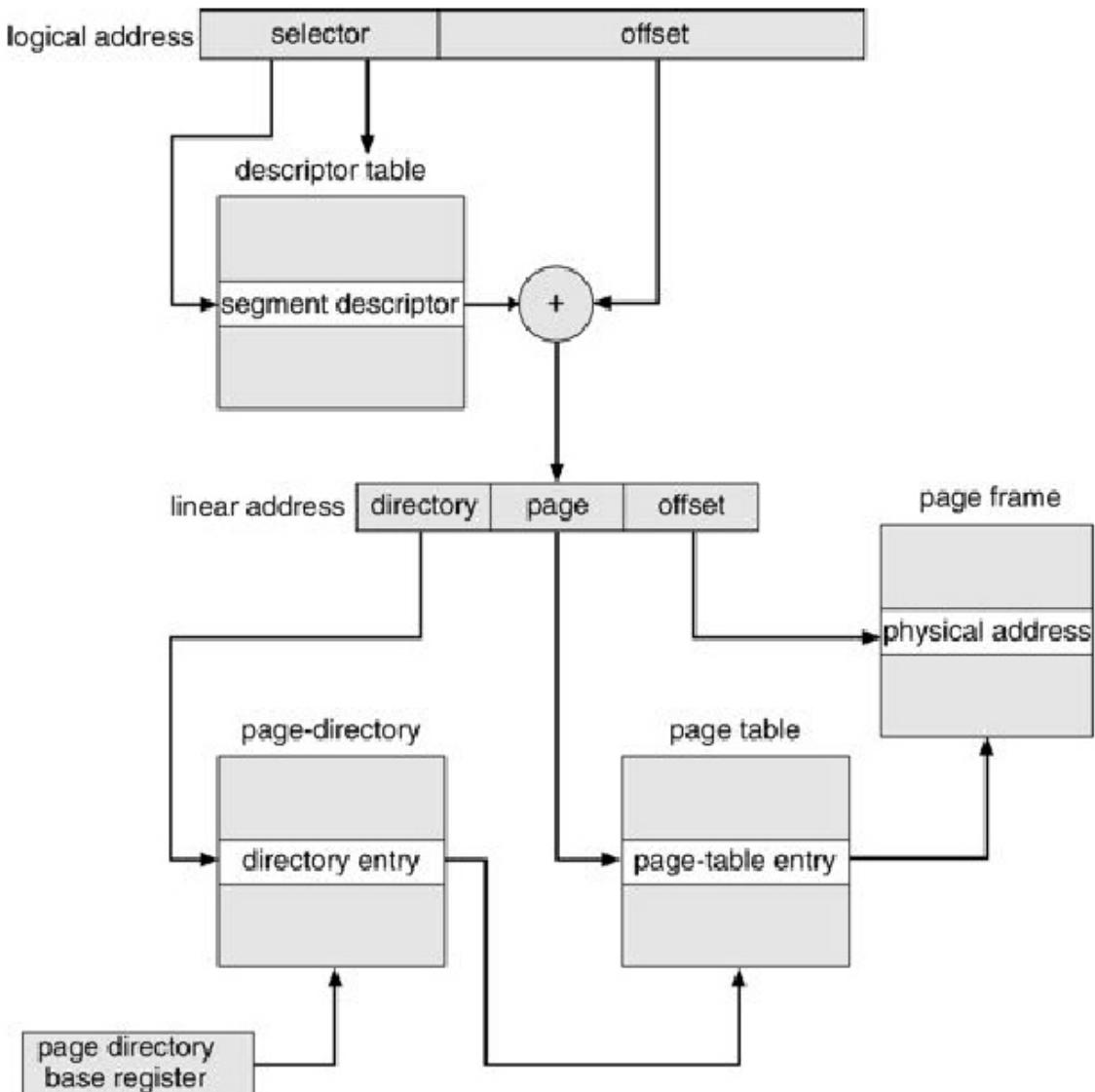


Рис. 4.9. Трансляция адреса в архитектуре x86

Системные вызовы для взаимодействия с подсистемой виртуальной памяти:

- `brk`, `sbrk` - для увеличения сегмента памяти, выделенного для данных программы
- `mmap`, `memmap`, `munmap` - для отображения файла или устройства в память
- `protect` - изменение прав доступа к областям памяти процесса

Пример выделение памяти процессу:

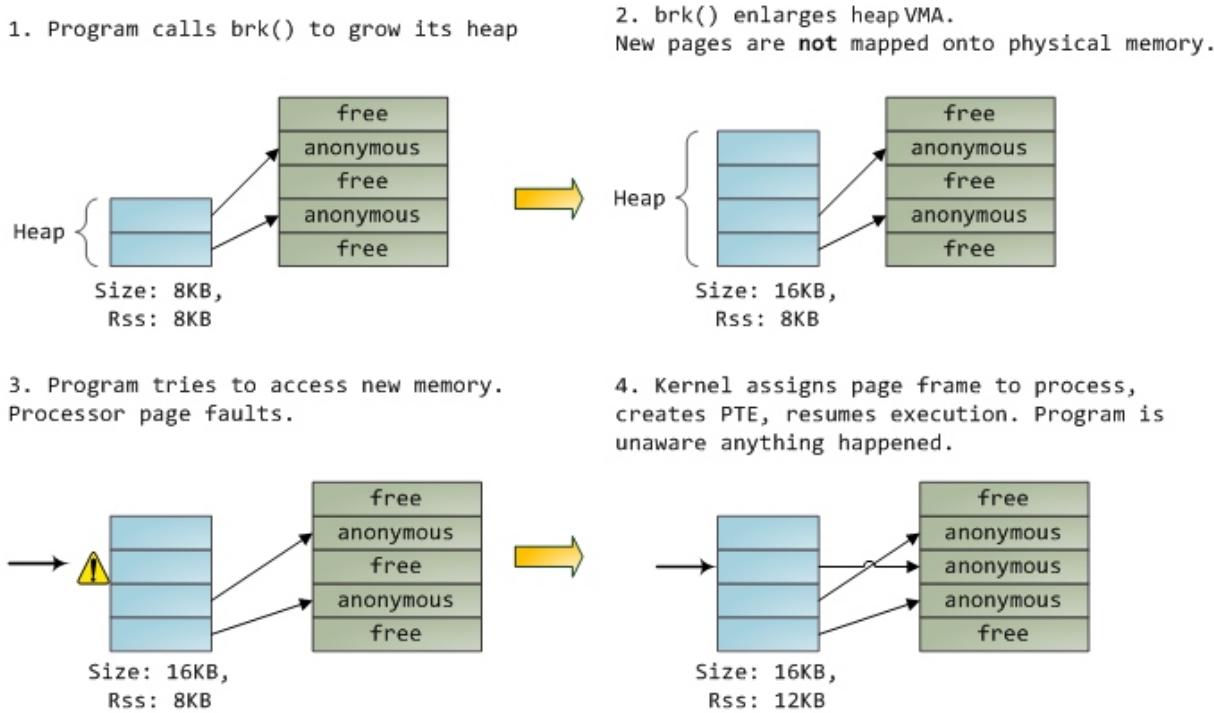


Рис. 4.10. Ленивое выделение памяти при вызове brk

## Алгоритмы выделения памяти

Эффективное выделение памяти предполагает быстрое (за 1 или несколько операций) нахождение свободного участка памяти нужного размера.

Способы учета свободных участков:

- битовая карта (bitmap) — каждому блоку памяти (например, странице) ставится в соответствие 1 бит, который имеет значение занят/свободен
- связный список — каждому непрерывному набору блоков памяти одного типа (занят/свободен) ставится в соответствие 1 запись в связном списке блоков, в которой указывается начало и размер участка
- использование нескольких связных списков для участков разных размеров — см. алгоритм [Buddy allocation](#)

## Кэширование

Кеш — это компонент компьютерной системы, который прозрачно хранит данные так, чтобы последующие запросы к ним могли быть удовлетворены быстрее. Наличие кеша подразумевает также наличие запоминающего устройства (гораздо) большего размера, в которых данные хранятся

изначально. Запросы на получение данных из этого устройства **прозрачно** проходят через кеш в том смысле, что если этих данных нет в кеше, то они запрашиваются из основного устройства и параллельно записываются в кеш. Соответственно, при последующем обращении данные могут быть извлечены уже из кеша. За счет намного меньшего размера кеш может быть сделан намного быстрее и в этом основная цель его существования.

По принципу записи данных в кеш выделяют:

- сквозной (write-through) — данные записываются синхронно и в кеш, и непосредственно в запоминающее устройство
- с обратной записью (write-back, write-behind) — данные записываются в кеш и иногда синхронизируются с запоминающим устройством

По принципу хранения данных выделяют:

- полностью ассоциативные
- множественно-ассоциативные
- прямого соответствия

L1 Cache – 32KB, 8-way set associative, 64-byte cache lines  
1. Pick cache set (row) by index

36-bit memory location as interpreted by the L1 cache:

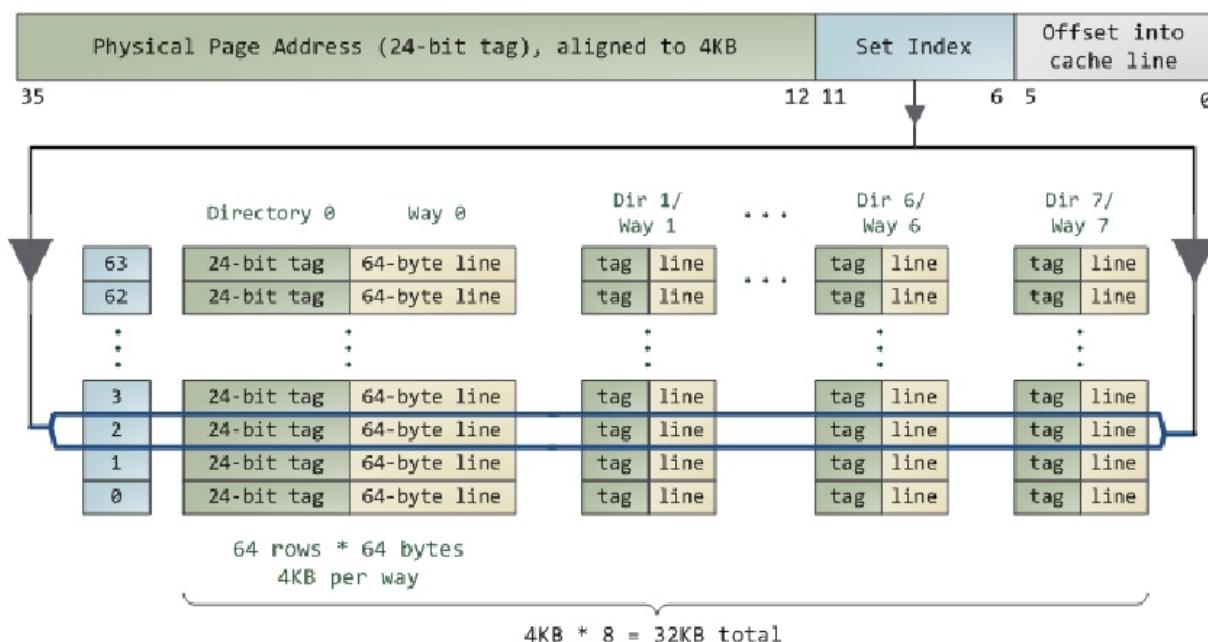


Рис. 4.11. Пример множественно-ассоциативного кеша в архитектуре x86

## 2. Search for matching tag in the set

36-bit memory location as interpreted by the L1 cache:

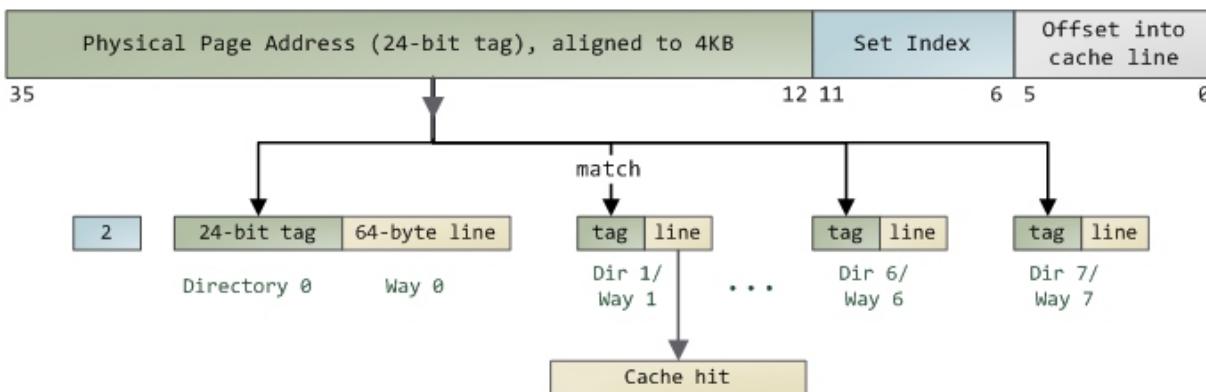


Рис. 4.12. Поиск в множественно-ассоциативном кеше

## Алгоритмы замещения записей в кеше

Поскольку любой кеш всегда меньше запоминающего устройства, всегда возникает необходимость для записи новых данных в кеш удалять из него ранее записанные. Эффективное удаление данных из кеша подразумевает удаление наименее востребованных данных. В общем случае нельзя сказать, какие данные являются наименее востребованными, поэтому для этого используются эвристики. Например, можно удалять данные, к которым происходило наименьшее число обращений с момента их загрузки в кеш (least frequently used, **LFU**) или же данные, к которым обращались наименее недавно (least recently used, **LRU**), или же комбинация этих двух подходов (**LRFU**).

Кроме того, аппаратные ограничения по реализации кеша часто требуют минимальных расходов на учет служебной информации о ячейках, которой является также и использование данных в них. Наиболее простым способом учета обращений является установка 1 бита: было обращение или не было. В таком случае для удаления из кеша может использоваться алгоритм **часы** (или **второго шанса**), который по кругу проходит по всем ячейкам, и выгружает ячейку, если у нее бит равен 0, а если 1 — сбрасывает его в 0.

Более сложным вариантом является использование аппаратного счетчика для каждой ячейки. Если этот счетчик фиксирует число обращений к ячейке, то это простой вариант алгоритма LFU. Он обладает следующими недостатками:

- может произойти переполнение счетчика (а он, как правило, имеет очень небольшую разрядность) — в результате будет утрачена вся информация об обращениях к ячейке

- данные, к которым производилось множество обращений в прошлом, будут иметь высокое значение счетчика даже если за последнее время к ним не было обращений

Для решения этих проблем используется механизм **старения**, который предполагает периодический сдвиг вправо одновременно счетчиков для всех ячеек. В этом случае их значения будут уменьшаться (в 2 раза), сохраняя пропорцию между собой. Это можно считать вариантом алгоритм LRFU.

## Литература

- Управление памятью
- Виртуальная память
- [What Every Programmer Should Know About Memory](#)
- [The Memory Management Reference](#)
- Software Illustrated series by Gustavo Duarte:
  - [How The Kernel Manages Your Memory](#)
  - [Memory Translation and Segmentation](#)
  - [Getting Physical With Memory](#)
  - [What Your Computer Does While You Wait](#)
  - [Cache: a place for concealment and safekeeping](#)
  - [Page Cache, the Affair Between Memory and Files](#)
- [Memory Allocators 101](#)
- [How tcmalloc Works](#)
- [How Bad Can 1GB Pages Be?](#)
- [How Misaligning Data Can Increase Performance 12x by Reducing Cache Misses](#)
- [Real Mode Memory Management](#)
- [Memory Testing from Userspace Programs](#)
- [How L1 and L2 CPU caches work](#)

- Redis latency spikes and the Linux kernel

# Исполняемые файлы

## Программа в памяти (в Linux)

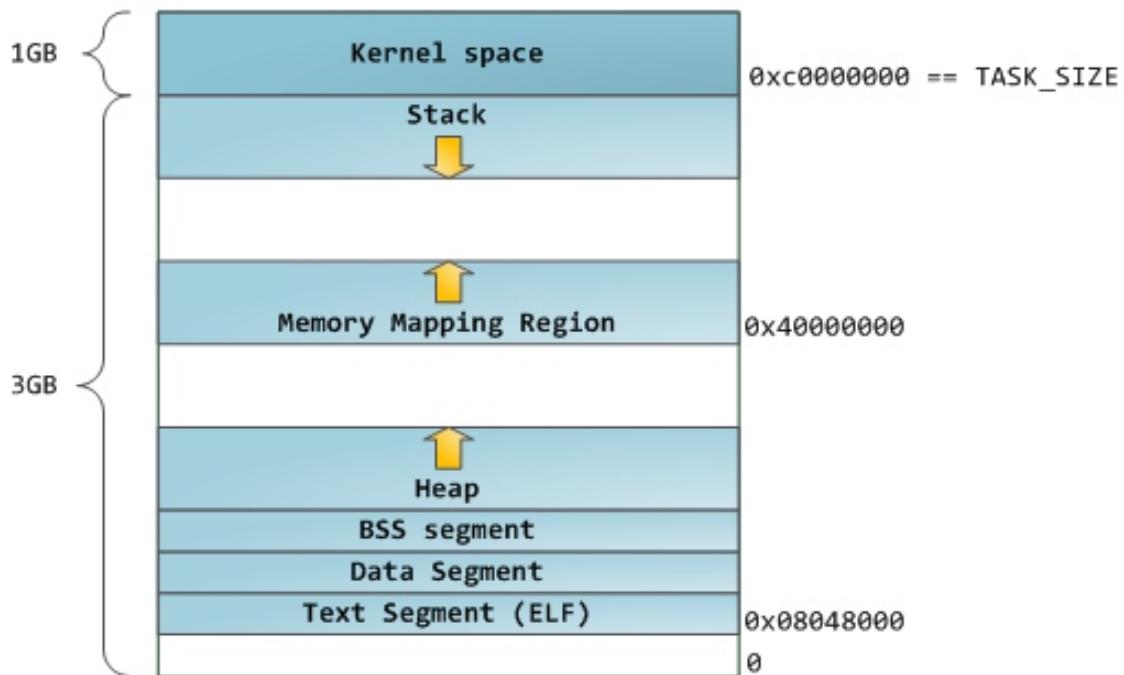


Рис. 5.1. Программа в памяти

Выполнение программы начинается с системного вызова `exec`, которому передается путь к файлу с бинарным кодом программы. `exec` — это интерфейс к загрузчику программ ОС, который загружает секции программы в память в зависимости от формата исполняемого файла, а также выделяет дополнительные секции динамической памяти. После загрузки память программы продолжает быть разделенной на отдельные секции. Указатели на начало/конец и другие свойства каждой секции находятся в структуре `mm_struct` текущего процесса.

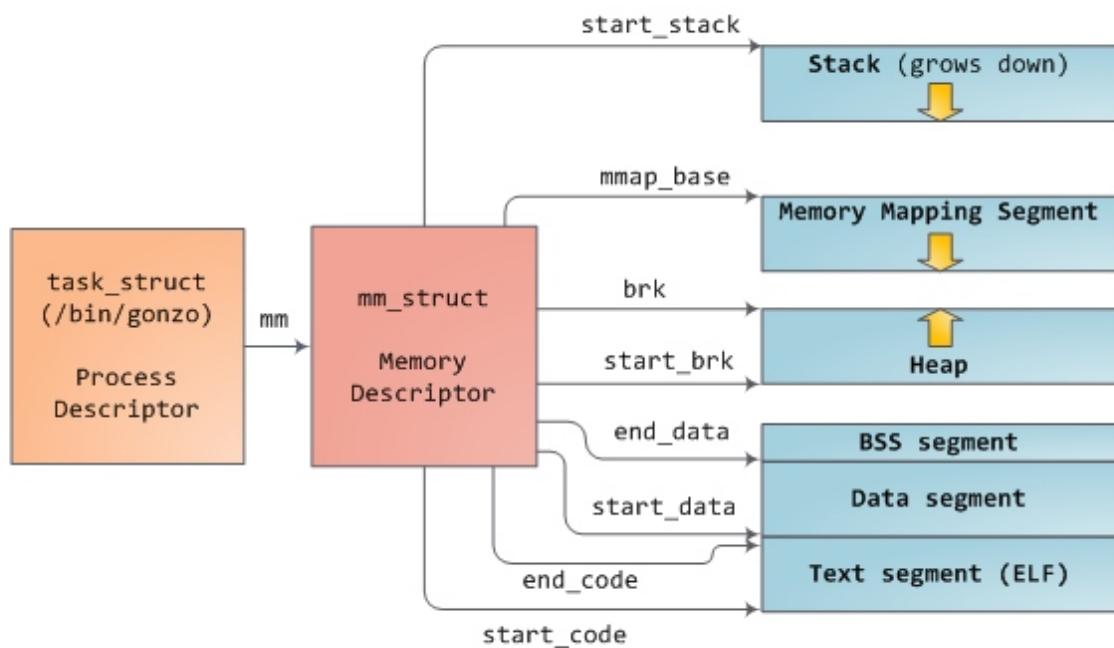


Рис. 5.2. Сегменты памяти процесса

Для загрузки отдельных сегментов в память используется системный вызов `mmap`.

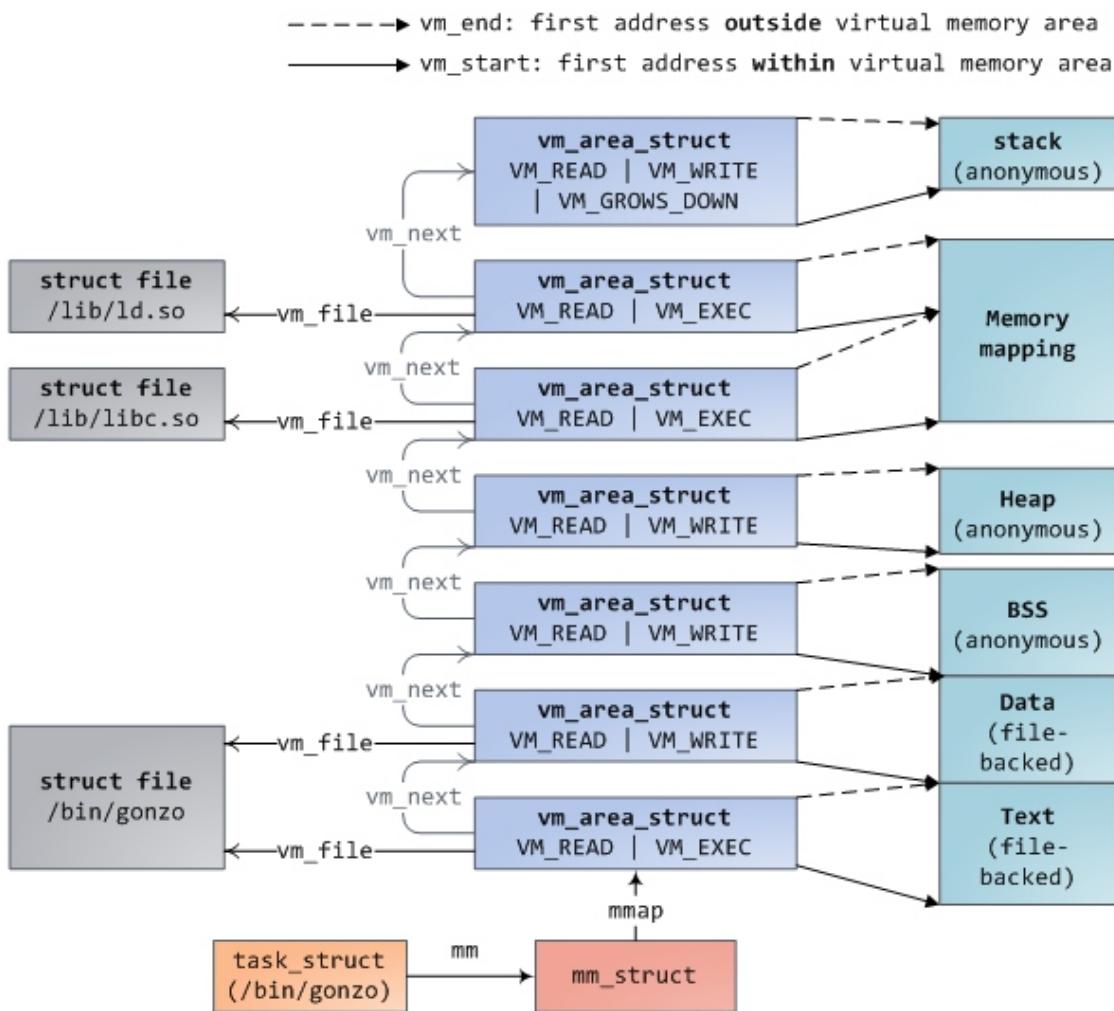


Рис. 5.3. Более подробная схема сегментов памяти процесса

## Статическая память программы

Статическая память программы — это часть памяти, которая является отображением кода объектного файла программы. Она инициализируется загрузчиком программ ОС из исполняемого файла (способ инициализации зависит от конкретного формата исполняемого файла).

Она включает несколько секций, среди которых общераспространенными являются:

- секция `text` — секция памяти, в которую записываются сами инструкции программы

- секция `data` — секция памяти, в которую записываются значения статических переменных программы
- секция `bss` — секция памяти, в которой выделяется место для записи значений объявленных, но не инициализированных в программе статических переменных
- секция `rodata` — секция памяти, в которую записываются значения констант программы
- секция таблицы символов — секция, в которой записаны все внешние (экспортируемые) символы программы с адресами их местонахождения в секциях `text` или `data` программы

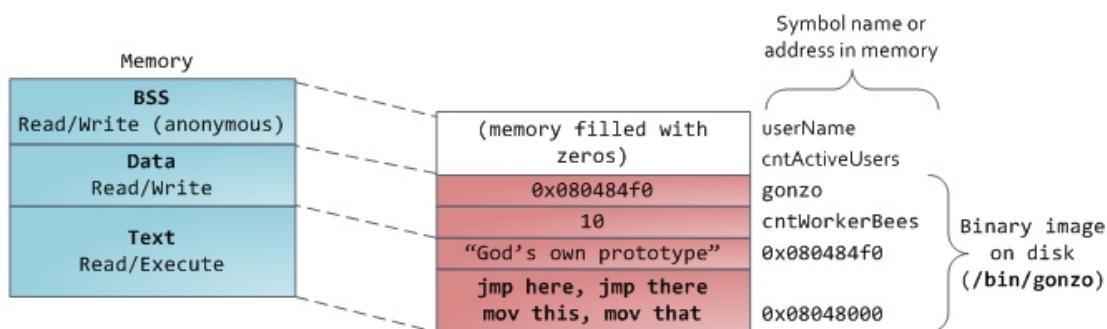


Рис. 5.4. Статическая память программы

## Динамическая память программы

Динамическая память выделяется программе в момент ее создания, но ее содержимое создается программой по мере ее выполнения. В области динамической памяти используется 3 стандартные секции, помимо которых могут быть и другие.

- стек (stack)
- куча (heap)
- сегмент отображаемой памяти (memory map segment)

Для выделения дополнительного объема динамической памяти используется системный вызов `brk`.

## Стек

(Более правильное название используемой структуры данных — **стопка** или **магазин**. Однако, исторически прижилось заимствованное название стек).

Стек (stack) — это часть динамической памяти, которая используется при

вызове функций для хранения ее аргументов и локальных переменных. В архитектуре x86 стек растет вниз, т.е. вершина стека имеет самый маленький адрес. Регистр SP (Stack Pointer) указывает на текущую вершину стека, а регистр BP (Base Pointer) указывает на т.н. базу, которая используется для разделение стека на логические части, относящиеся к одной функции — **фреймы** (кадры). Помимо обычных инструкций работы с памятью и регистрами (таких как `mov`), дополнительно для манипуляции стеком используются инструкции `push` и `pop`, которые заносят данные на вершину стека и забирают данные с вершины. Эти инструкции также осуществляют изменение регистра SP.

Как правило, в программах на высокоровневых языках программирования нет кода для работы со стеком напрямую, а это делает за кадром компилятор, реализуя определенные соглашения о вызовах функций и способы хранения локальных переменных. Однако функция `alloca` библиотеки `stdlib` позволяет программе выделять память на стеке .

Вызов функции высокоровневого языка создает на стеке новый фрейм, который содержит аргументы функции, адрес возврата из функции, указатель на начало предыдущего фрейма, а также место под локальные переменные.

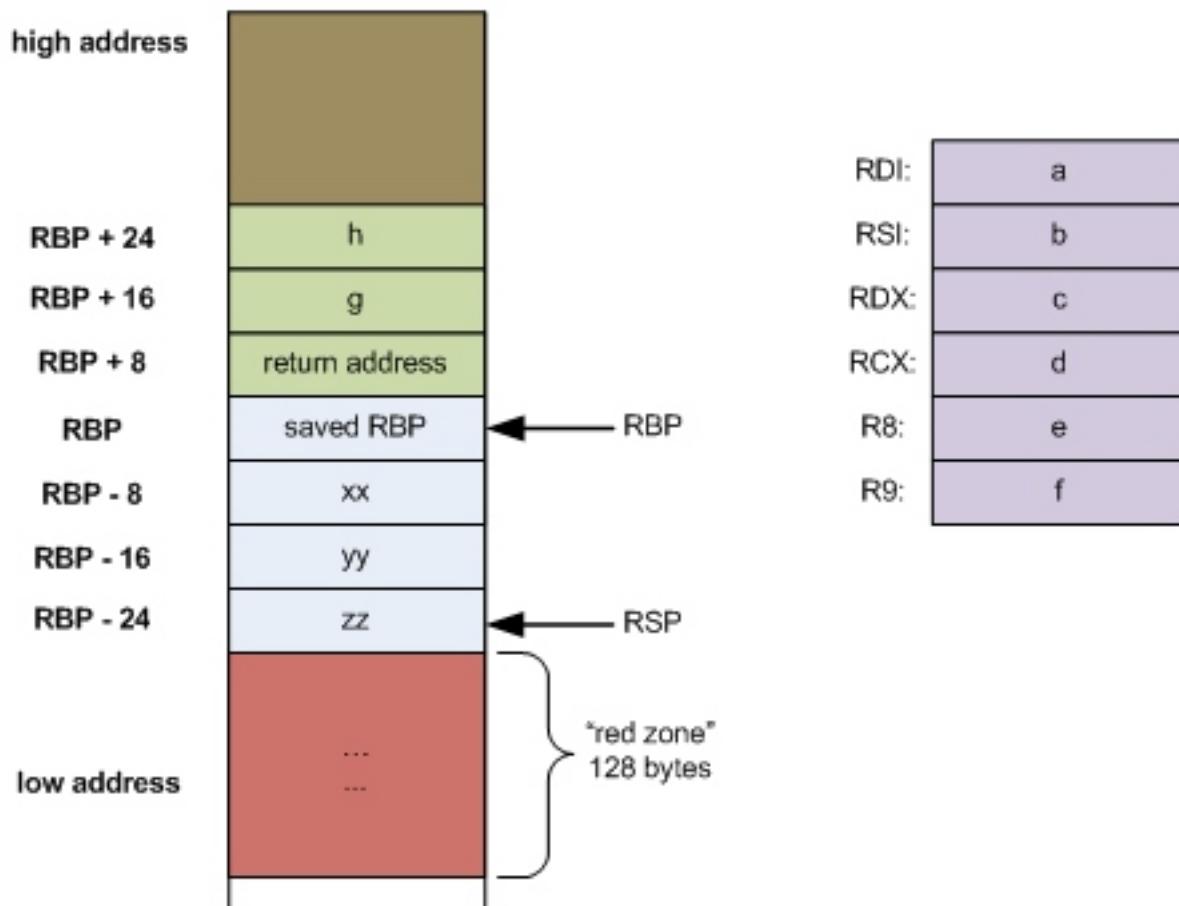
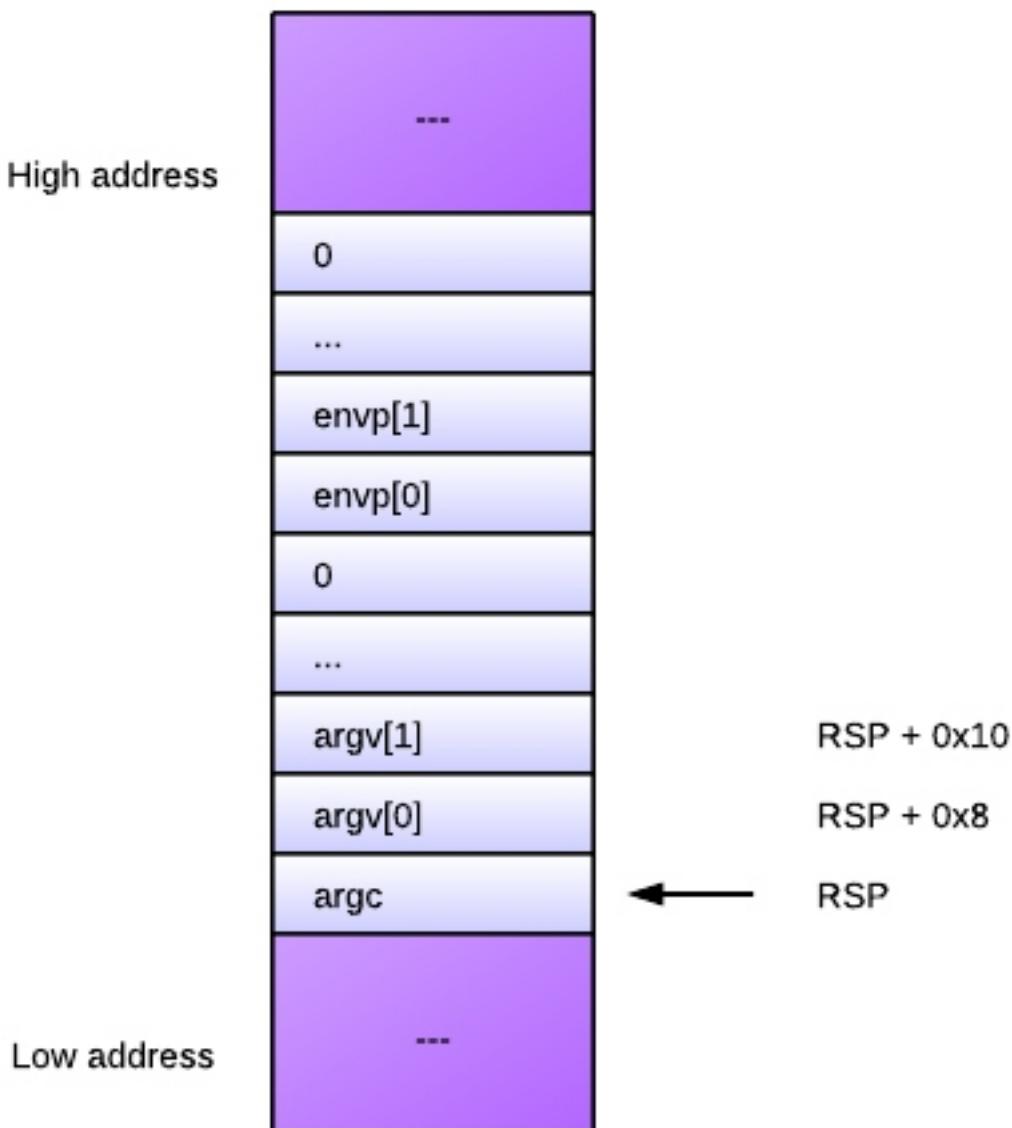


Рис. 5.5. Вид фрейма стека при вызове в рамках AMD64 ABI

В начале работы программы в стеке выделен только 1 фрейм для функции `main` и ее аргументов — числового значения `argc` и массива указателей переменной длины `argv`, каждый из которых записывается на стек по отдельности, а также переменных окружения.

Рис. 5.6. Вид стека после вызова функции `main`

## Куча

Куча (heap) — это часть динамической памяти, предназначенная для выделения участков памяти произвольного размера. Она в первую очередь используется для работы с массивами неизвестной заранее длины (буферами), структурами

и объектами.

Для управления кучей используется подсистема выделения памяти (memory allocator), интерфейс к которому — это функции `malloc/calloc` и `free` в `stdlib`.

Основные требования к аллокатору памяти:

- минимальное используемое пространство и фрагментация
- минимальное время работы
- максимальная **локальность** памяти
- максимальная настраиваемость
- максимальная совместимость со стандартами
- максимальная переносимость
- обнаружение наибольшего числа ошибок
- минимальные аномалии

Многие языки высокого уровня реализуют более высокоуровневый механизм управления памятью поверх системного аллокатора — автоматическое выделение памяти со сборщиком мусора. В этом случае в программы не производится вызов функции `malloc`, а управление памятью осуществляется среда исполнения программы.

Варианты реализации сборки мусора:

- подсчет ссылок
- трассировка/с выставлением флагов (Mark and Sweep)

## Сегмент файлов, отображаемых в память

Сегмент файлов, отображаемых в память — это отдельная область динамической памяти, которая используется для эффективно работы с файлами, а также для подключения участков памяти других программ с помощью вызова `mmap`.

## Исполняемые файлы

В результате компиляции программы в машинный код создается исполняемый файл, т.е. файл, содержащий непосредственно инструкции процессора.

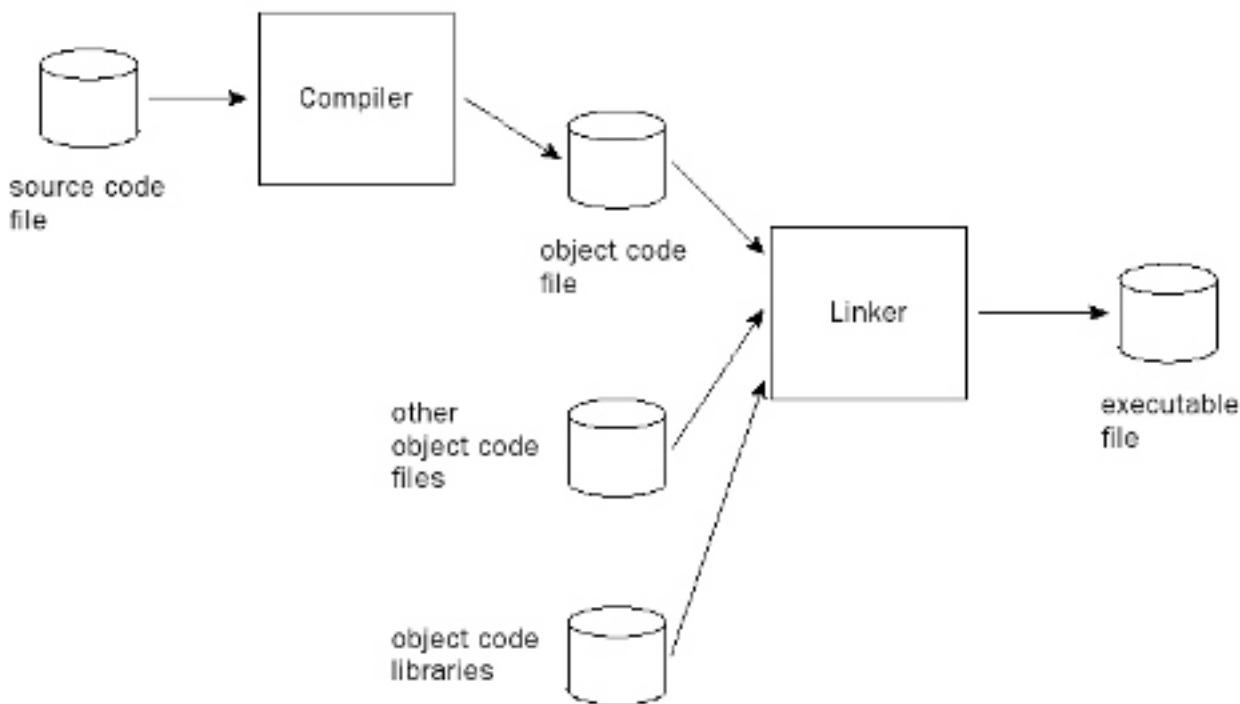


Рис. 5.7. Этапы создания исполняемого файла

Типы исполняемых файлов:

- объектный файл (object file) — файл, преобразованный компилятором, но не приведенный окончательно к виду исполняемого файла в одном из форматов исполняемых файлов
- исполняемая программа (executable) — файл в одном из форматов исполняемых файлов, который может быть запущен загрузчиком программ ОС
- разделяемая библиотека (shared library) — программа, которая не может быть запущена самостоятельно, а подключается (компилятором) как часть других программ
- снимок содержимого памяти (core dump) — снимок состояния памяти программы в момент ее исполнения — позволяет продолжить исполнение программы с того места, на котором он был создан

## Форматы исполняемых файлов

Формат исполняемых файлов — это определенная структура бинарного файла, которую формируют компилятор и компоновщик программы, и которая используется загрузчиком программ ОС.

В рамках формата исполняемых файлов описывается:

- способ задания секций файла, их количество и порядок
- метаданные, их типы и размещение в файле
- каким образом файл будет загружаться: по какому адресу в памяти, в какой последовательности
- способ описания импортируемых и экспортируемых символов
- ограничения на размер файла и т.п.

Распространенные форматы:

- .COM
- a.out
- COFF
- DOS MZ Executable
- Windows PE
- Windows NE
- ELF

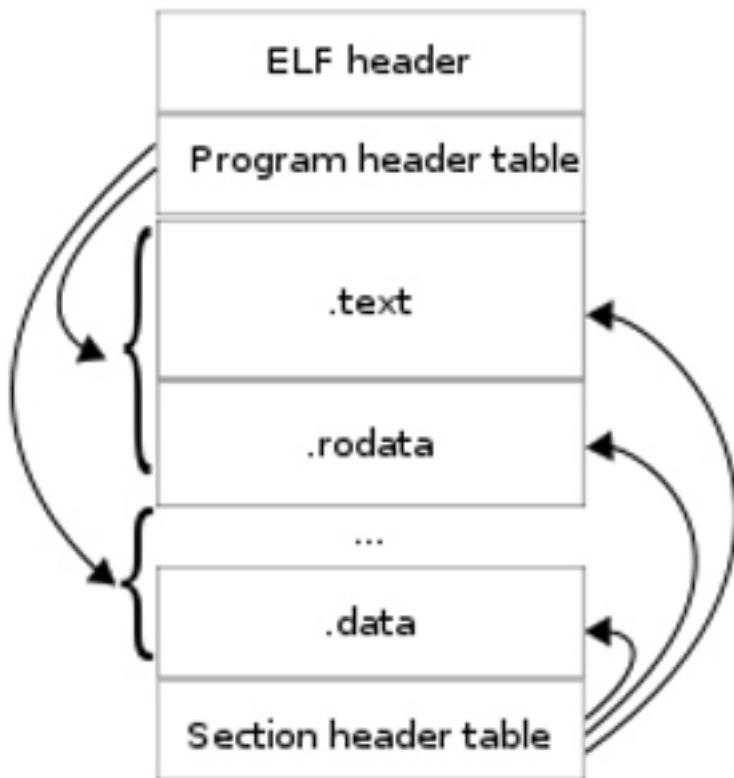


Рис. 5.8. Формат ELF

Формат ELF (Executable and Linkable Format) — стандартный формат исполняемых файлов в Linux. Файл в этом формате содержит:

- заголовок файла
- таблицу заголовков сегментов программы
- таблицу заголовков секций программы
- блоки данных

Сегменты программы содержат информацию, используемую загрузчиком программы, а секции — используемую компоновщиком.

Также в исполняемом файле может записываться отладочная информация. Некоторые форматы имеют встроенную поддержку для этого, а для некоторых форматов используются дополнительные, такие как:

- stabs
- DWARF

## Библиотеки

Библиотеки содержат функции, выполняющие типичные действия, которые могут использоваться другими программами. В отличие от исполняемой программы библиотека не имеет точки входа (функции `main`) и предназначена для подключения к другим программам или библиотекам. Стандартная библиотека C (`libc`) — первая и основная библиотека любой программы на C.

Библиотеки могут подключаться к программе в момент:

- сборки - build time (такие библиотеки называются статическими)
- загрузки - load time
- исполнения - run time

Разделяемые библиотеки — это библиотеки, которые подключаются в момент загрузки или исполнения программы и могут разделяться между несколькими программами в памяти для экономии памяти. Помимо этого они не включаются в код программы и таким образом не увеличивают его объем. С другой стороны, они в большей степени подвержены проблеме конфликта версий зависимостей разных компонент (в применении к библиотекам также называемой DLL hell).

Способы подключения разделяемых библиотек в Unix:

- релокации времени загрузки программы

- позиционно-независимый код (PIC)

Релокации времени загрузки программы используют специальную секцию исполняемого файла — таблицу релокации, в которой записываются преобразования, которые нужно произвести с кодом библиотеки при ее загрузке. Ее недостатки — увеличение времени загрузки программы из-за необходимости переписывания кода библиотеки для применения всех релокаций на этом этапе, а также невозможность сделать секцию кода библиотеки разделяемой в памяти из-за того, что релокация для каждой программы применяется по-разному, т.к. библиотека загружается в память по разным виртуальным адресам.

Позиционно-независимый код использует таблицу глобальных отступов (Global Offset Table, GOT), в которой записываются адреса всех экспортруемых символов библиотеки. Его недостаток — это замедление всех обращений к символам библиотеки из-за необходимости выполнять дополнительное обращение к GOT.

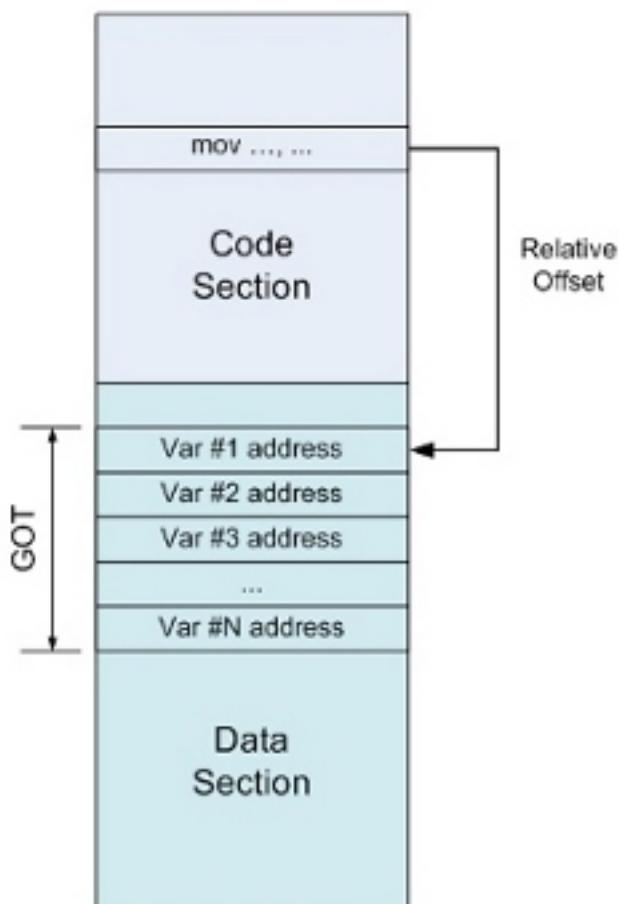
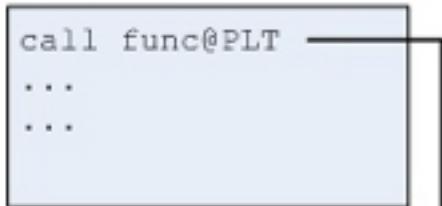


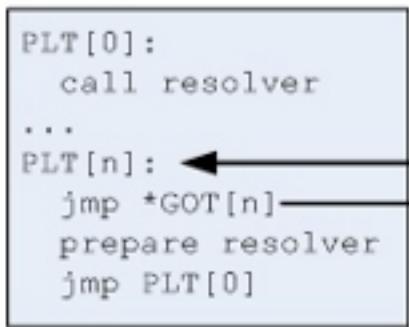
Рис. 5.9. Таблица глобальных отступов

Для поддержки позднего связывания функций через механизм "трамплина" также используется таблица компоновки процедур (Procedure Linkage Table, PLT).

### Code:



### PLT:



### GOT:



### Code:



Рис. 5.10. Реализация трамплина при вызове функции с помощью таблицы компоновки процедур

## Виртуальные машины

Виртуальная машина — это программная реализация реального компьютера, которая исполняет программы.

Применения виртуальных машин:

- увеличение переносимости кода
- исследование и оптимизация программ
- эмулятор
- песочница
- виртуализация
- платформа для R&D языков программирования

- платформа для R&D различных компьютерных систем
- скрытие программ (вирусы)

Типы:

- системная — полная эмуляция компьютера
- процессная — частичная эмуляция компьютера для одного из процессов ОС

## Системные ВМ

Виды системных ВМ:

- Гипервизор/монитор виртуальных машин: тип 1 (на голом железе) и тип 2 (на ОС-хозяине)
- Паравиртуализация

Требования Попека и Голдберга для эффективной виртуализации:

- Все чувствительные инструкции аппаратной архитектуры являются привилегированными
- Нет временных ограничений на выполнение инструкций (рекурсивная виртуализация)

Примеры: VMWare, VirtualBox, Xen, KVM, Qemu, Linux LXC containers, Solaris zones

## Процессные ВМ

Процессные ВМ функционируют по принципу 1 процесс – 1 экземпляр ВМ и, как правило, предоставляют интерфейс более высокого уровня, чем аппаратная платформа.

Код программы для таких ВМ компилируется в промежуточное представление (**байт-код**), который затем интерпретируется ВМ. Часто в них также используется JIT-компиляция байт-кода в родной код.

Варианты реализации:

- Стек-машина (0-операнд)
- Аккумулятор-машина (1-операнд)
- Регистровая машина (2- или 3-операнд)

Примеры: JVM, .Net CLR, Parrot, LLVM, Smalltalk VM, V8

# Литература

- [Anatomy of a Program in Memory](#)
- [How is a binary executable organized](#)
- [Inside Memory Management](#)
- [x86 Registers](#)
- [Stack frame layout on x86-64](#)
- [Doug Lea's malloc](#)
- [Visualizing Garbage Collection Algorithms](#)
- [Demystifying Garbage Collectors](#)
- Eli Benderski on Static and Dynamic Object Code in Linux:
  - [How Statically Linked Programs Run on Linux](#)
  - [Load-time relocation of shared libraries](#)
  - [Position Independent Code \(PIC\) in shared libraries](#)
  - [Position Independent Code \(PIC\) in shared libraries on x64](#)
- [How To Write Shared Libraries](#)
- [Executable and Linkable Format \(ELF\)](#)
- [Why do Windows functions all begin with a pointless MOV EDI, EDI instruction?](#)

# Управление процессами

## Процесс

Процесс – это адресное пространство и единая нить управления. (Устаревшее определение)

Более точно понятие процесса включает в себя:

- Программу, которая исполняется
- Ее динамическое состояние (регистровый контекст, состояние памяти и т.д.)
- Доступные ресурсы (как индивидуальные для процесса, такие как дескрипторы файлов, так и разделяемые с другими)

В ОС структура Процесс (Process control block) — одна из ключевых структур данных. Она содержит всю информацию о процессе, необходимую разным подсистемам ОС. Эта информация включает:

- PID (ID процесса)
- PPID (ID процесса-родителя)
- путь и аргументы, с которым запущен процесс
- программный счетчик
- указатель на стек
- и др.

Ниже приведена небольшая часть этой структуры в ОС Linux:

```
#include<sched.h>
struct task_struct {
    /* Состояние:
     * -1 - заблокированный,
     * 0 - готовность,
     * >0 - остановленный */
    volatile long state;
    void *stack;
    unsigned long flags;
    int prio, static_prio, normal_prio;
    struct list_head run_list;
    const struct sched_class *sched_class;
    ...
/* task state */
```

```
    struct linux_binfmt *binfmt;
    long exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;
    pid_t pid;
    pid_t tgid;
    struct task_struct *real_parent;
    struct task_struct *parent;
    struct list_head children;
    struct list_head sibling;
    struct task_struct *group_leader;
    struct timespec start_time;
    struct timespec real_start_time;
    ...
/* process credentials */
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
    struct group_info *group_info;
    kernel_cap_t cap_effective,
                cap_inheritable, cap_permitted;
    unsigned keep_capabilities:1;
    struct user_struct *user;
    ...
/* open file information */
    struct files_struct *files;
/* namespace */
    struct nsproxy *nsproxy;
/* signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;
    ...
};

};
```

## Нить управления

Нить управления (thread) — это одна логическая цепочка выполнения команд. В одном процессе может быть как одна нить управления, так и несколько (в системах с поддержкой многопоточности — multithreading).

ОС предоставляет интерфейс для создания нитей управления и в этом случае берет на себя их планирование наравне с планированием процессов. В стандарте POSIX описан подобный интерфейс, который реализован в библиотеке **PThreads**. Нити, предоставляемые ОС, называются **родными** (native). Однако любой процесс может организовать управление нитями внутри

себя независимо от ОС (фактически, в рамках одной родной нити ОС). Такой подход называют **зелеными** или **легковесными** нитями.

Волокно (fiber) — легковесная нить, которая работает в системе кооперативной многозадачности (см. ниже).

Преимущества родных нитей:

- не требуют дополнительных усилий по реализации
- используют стандартные механизмы планирования ОС
- блокировка и реакция на сигналы ОС происходит в рамках нити, а не всего процесса

Преимущества зеленых нитей:

- потенциально меньшие накладные расходы на создание и поддержку
- не требуют переключения контекста при системных вызовах, что дает потенциально большое быстродействие
- гибкость: процесс может реализовать любую стратегию планирования таких нитей

## Виды процессов

Процессы могут запускаться для разных целей: - выполнения каких-то одноразовых действий (например, скрипты) - выполнения задач под управлением пользователя (интерактивные процессы, такие как редактор) - беспрерывной работы в фоновом режиме (сервисы или демоны, такие как сервис терминала или почтовый сервер)

Процесс-демон — это процесс, который запускается для долгосрочной работы в фоновом режиме, отключается от запустившего его терминала (его стандартные потоки ввода-вывода закрываются либо перенаправляются в лог-файл) и меняет свои полномочия на минимально необходимые.

Управление таким процессом, обычно, осуществляется с помощью механизма сигналов ОС.

# Жизненный цикл процесса



Рис. 6.1. Жизненный цикл процесса

## Порождение процесса

Все процессы ОС, за исключением первого процесса, который запускается после загрузки ядра, имеют родителя. Создание нового процесса требует инициализации структуры PCB и запуска (постановки на планирование) нити управления процесса. Основным требованием к этим операциям является скорость выполнения. Инициализация структуры PCB с нуля является затратной операцией, кроме того порожденному процессу, как правило, требуется доступ к некоторым ресурсам (таким как потоки ввода-вывода) родительского процесса. Поэтому обычно структура нового процесса создается методом **клонирования** структуры родителя. Альтернативой является загрузка предварительно инициализированной структуры из файла и ее модификация.

Модель **fork/exec** — это модель двухступенчатого порождения процесса в Unix-системах. На первой ступени с помощью системного вызова `fork` создается идентичная копия текущего процесса (для обеспечения быстродействия, как правило, через механизм копирования-при-записи - `copy-on-write`, COW). На втором этапе с помощью операции `exec` в память созданного процесса загружается новая программа. В этой модели процесс-родитель имеет возможность дождаться завершения дочернего процесса с помощью системных вызовов семейства `wait`. Разбивка этой операции на два этапа дает возможность легко порождать идентичные копии процесса (например, для масштабирования приложения - такой способ применяется в сетевых серверах), а также гибко управлять ресурсами,

доступными дочернему процессу.

## Завершение процесса

По завершению процесс возвращает целочисленный код возврата (exit code) — результат выполнения функции `main`. В Unix-системах код возврата, равный 0, сигнализирует об успехе, все остальные говорят об ошибке (разработчик приложения волен произвольно сопоставлять ошибки возвращаемым значениям).

Процесс может завершиться следующим образом:

- нормально: вызвав системный вызов `exit` или выполнив `return` из функции `main` (что приводит к вызову `exit` в функции `libc`, которая запустила `main`)
- ошибочно: если выполнение процесса вызывает критическую ошибку (Segmentation Fault, General Protection Exception, Division by zero или другие аппаратные исключения)
- принудительно: если процесс завершается ОС, например, при нехватки памяти, а также, если он не обрабатывает какой-либо из посланных ему сигналов (в том числе, сигнал `KILL`, который невозможно обработать, из-за чего посылка этого сигнала всегда приводит к завершению процесса)

Используя функции семейства `wait`, один процесс может ожидать завершения другого. Это часто используется в родительских процессах, которым нужно получить информацию о завершении своих потомков, чтобы продолжить работу. Вызовы `wait` являются блокирующими — функция не завершится, пока не завершится процесс, которого она ждёт.

Если потомок завершается, но родительский процесс не вызывает `wait`, потомок становится т.н. процессом **зомби**. Это завершившиеся процессы, информация о завершении которых никем не запрошена. Впрочем, после завершения родительского процесса все его потомки переходят к процессу с PID 1, т.е. `init`. Он самостоятельно очищает информацию, оставшуюся после зомби.

Пример программы, порождающей новый процесс и ожидающей его завершения:

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

```
int main() {
    // Клонируем текущий процесс
    pid_t childpid = fork();
    /* Переменная childpid будет существовать
     * и в оригинальном процессе, и в его клоне,
     * но в дочернем процессе она будет равна 0 */
    if (!childpid) {
        // Это дочерний процесс
        char* command[3] = {"./bin/echo", "Hello, world!", NULL};
        execvp(command[0], command);
        /* Если не произошло никаких ошибок, execvp() не
         * завершается
         * и программа никогда не достигнет этого участка
         * кода */
        exit(EXIT_FAILURE);
    } else if (childpid == -1) {
        // fork() возвращает -1 в случае ошибки
        fprintf(stderr, "Can't fork, exiting...\n");
        exit(EXIT_FAILURE);
    } else {
        // Это родительский процесс
        exit(EXIT_SUCCESS);
    }
    return 0;
}
```

## Работа процесса

В мультипроцессных системах все процессы исполняются на процессоре не все время своей работы, а только его часть. Соответственно, можно выделить:

- общее время нахождения процесса в системе: от момента его запуска до завершения
- (чистое) время исполнения процесса
- время ожидания

В состояние ожидания процесс может перейти либо при поступлении прерывания от процессора, либо после вызова самим процессом блокирующей операции, либо после добровольной передачи процессом управления ОС (вызов планировщика `schedule` при вытесняющей многозадачности либо же операция `yield` при кооперативной многозадачности).

При переходе процесса в состояние ожидания происходит **переключение**

**контекста** и запуск на процессоре кода ядра ОС (кода обработки прерываний или кода планировщика). Переключение контекста подразумевает сохранение в памяти содержания связанных с исполняемым процессом регистров и загрузка в регистры значений для следующего процесса.

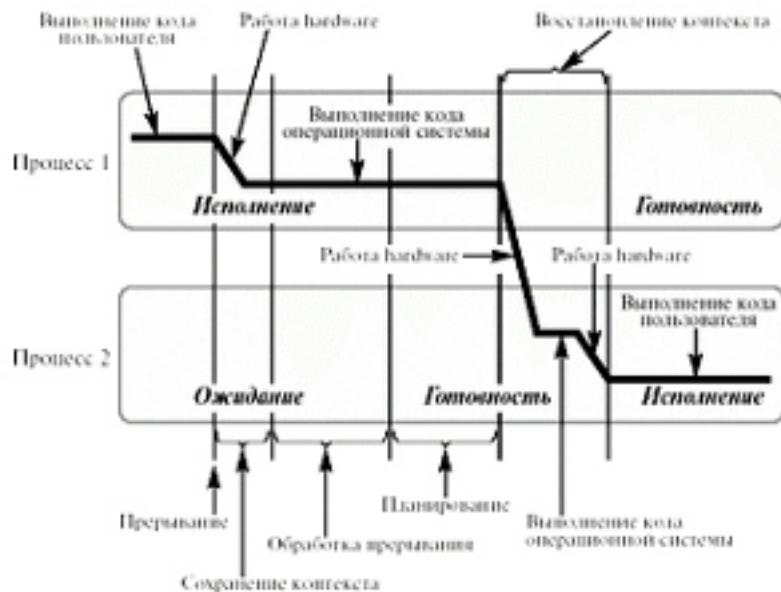


Рис. 6.2. Переключение контекста

В системах с вытесняющей многозадачностью прерывание CLOCK INTERRUPT вызывает планировщик ОС, который переводит процесс в состояние ожидания, если отведенное ему на исполнение время истекло.

Завершение блокирующих операций знаменуется прерыванием, при обработке которого ОС переводит заблокированный процесс в состояние готовности к работе.

## Планирование процессов

Многозадачность — это свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов.

Виды многозадачности:

- вытесняющая
- невытесняющая
- кооперативная (подвид невытесняющей)

Вытесняющая многозадачность подразумевает наличие в ОС специальной

программы-планировщика процессов, который принимает решение о том, какой процесс должен выполняться и сколько времени отвести ему на выполнение. После завершения отведенного времени процесс принудительно прерывается и управление передается другому процессу.

При невытесняющей многозадачности процессы работают поочередно, причем переключение происходит по завершению всего процесса или логического блока в его рамках. Кооперативная многозадачность — это вариант невытесняющей многозадачности, в которой только сам процесс может сигнализировать ОС о готовности передать управление.

## Алгоритмы планирования процессов

Планирование процессов применяется в системах с вытесняющей многозадачностью.

Требования к алгоритмам планирования:

- справедливость
- эффективность (в смысле утилизации ресурсов)
- стабильность
- масштабируемость
- минимизация времени: выполнения, ожидания, отклика

Алгоритмы планирования для выбора следующего процесса на исполнение, как правило, используют **приоритет** процесса. Приоритет может определяться статически (один раз для процесса) или же динамически (пересчитываться на каждом шаге планирования).

## Алгоритм Первый пришел — первый обслужен (FCFS)



Рис. 6.3. Алгоритм FCFS

Это простой алгоритм с наименьшими накладными расходами. Это алгоритм со статическим приоритетом, в качестве которого выступает время прихода процесса. Это наименее стабильный алгоритм, который не может гарантировать приемлемое время отклика в интерактивных системах, поэтому он применяется только в системах batch-обработки.

## Алгоритм Карусель (Round Robin)

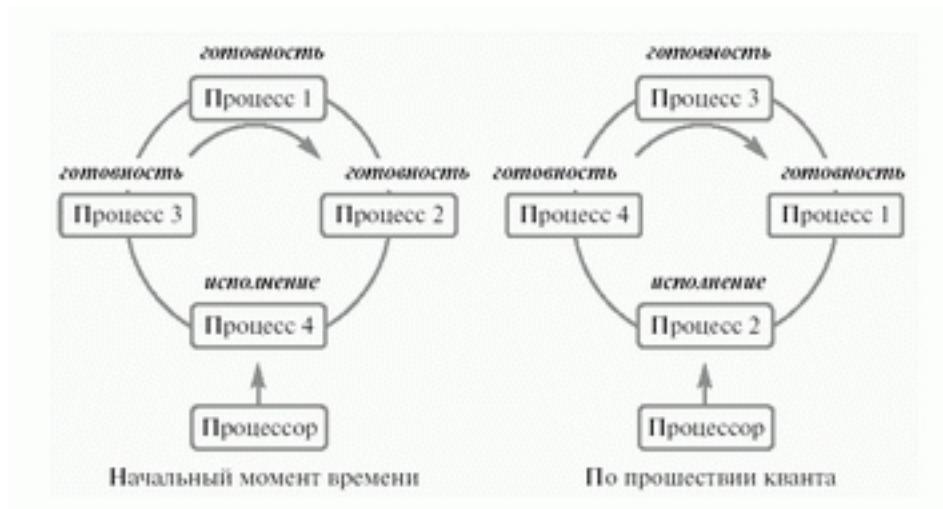


Рис. 6.4. Алгоритм Карусель

Этот алгоритм предполагает попеременное выполнение всех процессов в течение одинакового кванта времени, после завершения которого независимо от состояния процесса он прерывается и управление переходит к следующему процессу. Этот алгоритм является наиболее стабильным и простым. Он вообще не использует приоритет процесса.

## Алгоритм справедливого планирования

В основе этого алгоритма лежит принцип: из всех кандидатов на выполнение должен выбираться тот, у которого отношение чистого времени фактической работы к общему времени нахождения в системе наименьший. Иными словами, этот алгоритм использует динамический приоритет, который вычисляется по формуле  $r = t / T$  (где  $t$  - чистое время исполнения,  $T$  - время прошедшее от запуска процесса; чем меньше значение  $r$ , тем приоритет выше).

# Алгоритм Многоуровневая очередь с обратной связью

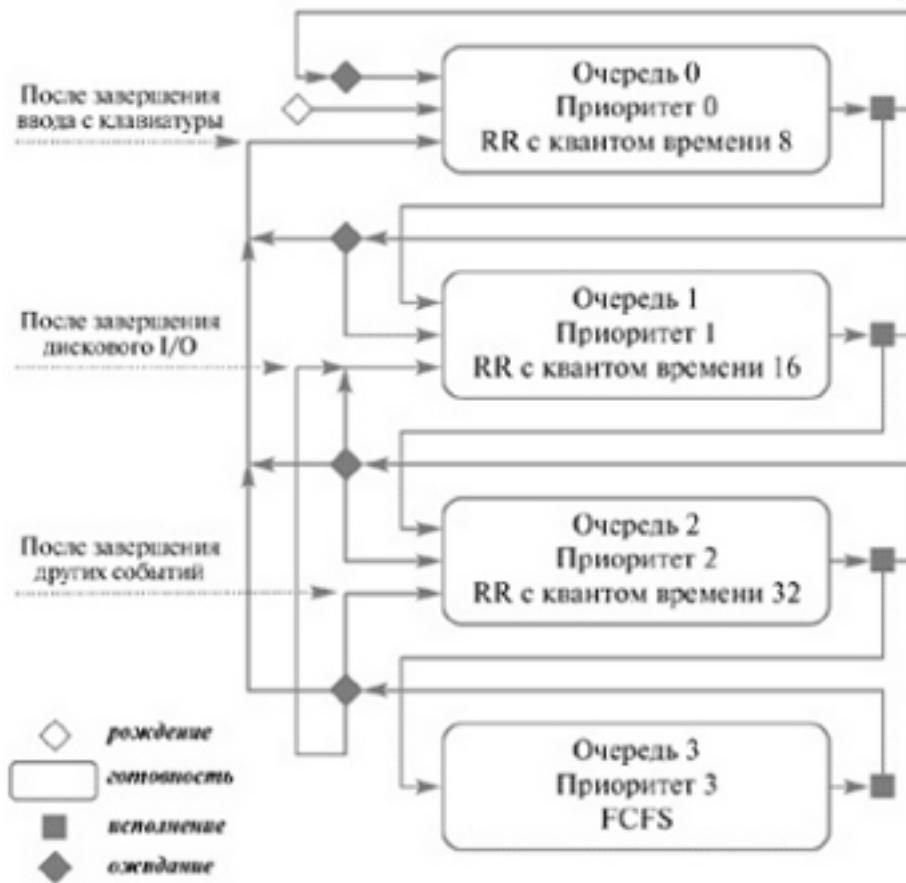


Рис. 6.5. Алгоритм Многоуровневая очередь с обратной связью

В качестве примера более сложного адаптивного алгоритма могут служить многоуровневые очереди с обратной связью, которые принимают решение о приоритете процесса на основе времени, которое необходимо ему для завершения работы или текущего логического блока.

## Реальные алгоритмы

Алгоритмы, применяемые в реальных системах, делятся на алгоритмы для интерактивных систем и алгоритмы для систем реального времени. Алгоритмы для систем реального времени всегда имеют ограничения на время завершения отдельных операций, поэтому им для планирования необходима дополнительная информация, которой, как правило, нет в интерактивных системах — например, время до завершения процесса.

В основе реальных алгоритмов лежат базовые алгоритмы, перечисленные выше, но также они используют некоторые дополнительные параметры такие

как:

- эпохи (временные интервалы, в конце которых накопленная для планирования информация сбрасывается)
- группировку процессов по классам (мягкого реального времени, процессы ядра, интерактивные, фоновые и т.д.) для обеспечения лучшего времени отклика системы

Кроме того, в таких системах учитываются технологии Симметричный мультипроцессинг (Symmetrical Multiprocessing, SMP) и Одновременная многопоточность (Symulteneous Multithreading, SMT), при которых несколько ядер процессора или несколько логических потоков исполнения в процессоре работают с общей памятью.

## Межпроцессное взаимодействие (IPC)

### Цели взаимодействия:

- модульность (путь Unix: маленькие кусочки, слабо связанные между собой, которые делают что-то одно и делают это очень хорошо)
- масштабирование
- совместное использование данных
- разделение привилегий
- удобство

### Типы взаимодействия:

- через разделяемую память
- обмен сообщениями
  - сигнальный
  - канальный
  - широковещательный

## Взаимодействие через разделяемую память

Самый быстрый и простой способ взаимодействия, при котором процессы записывают и считывают данные из общей области памяти. Он не требует никаких накладных расходов, но подразумевает наличие договоренности о формате записываемых данных. Проблемы этого подхода:

- необходимость блокирующей синхронизации для обеспечения неконфликтного доступа к общей памяти
- увеличение логической связности между отдельными процессами
- не возможность масштабироваться за рамками памяти одного компьютера

## Обмен сообщениями

Передача сообщений обладает прямо противоположными свойствами и считается более предпочтительным способом организации взаимодействия в общем случае. Сообщения могут передаваться как индивидуально, так и в рамках выделенной сессии обмена сообщениями.

## Сигнальный способ взаимодействия

Сигнальный способ — это вариант взаимодействия через передачу сообщений, который предполагает возможность отправки только заранее определенных сигналов, которые не имеют никакой нагрузки в виде данных. Таким образом сигналы могут передавать информацию только о предварительно заданном наборе событий. Такая система является простой, но не способна обслуживать все варианты взаимодействия. Поэтому она часто применяется для обслуживания критических сценариев работы.

Системный вызов `kill` позволяет посыпать сигналы процессам Unix. Среди них есть зарезервированные сигналы, такие как:

- TERM - запрос на завершение процесса
- HUP - запрос на перезапуск процесса
- ABRT - запрос на отмену текущей операции (генерируется ОС при нажатии `Ctrl-C`)
- PIPE - сигнал о закрытии конвеера другим процессом
- KILL - сигнал о принудительном завершении процесса
- и др.

Процес в Unix обязан обработать пришедший ему сигнал, иначе ОС принудительно завершает его работу.

## Канальный способ взаимодействия

Канальный способ — это вариант взаимодействия через передачу сообщений, при котором между процессами устанавливается канал соединения, в рамках которого передаются сообщения. Этот канал может быть как односторонним (сообщения идут только от одного процесса к другому), так и двусторонним.

Pipe (конвеер, анонимный канал) — односторонний канал, который позволяет процессам передавать данные в виде потока байт. Это самый простой способ взаимодействия в Unix, имеющий специальный синтаксис в командных оболочках (`proc1 | proc2`, что означает, что данные из процесса `proc1` передаются в `proc2`). Анонимный канал создается системным вызовом `pipe`, который принимает на вход массив из двух чисел и записывает в них два дескриптора (один из них открыт на запись, а другой — на чтение).

Особенности анонимных каналов:

- данные передаются построчно
- не задан формат сообщений, т.е. процессы сами должны "договариваться" о нем
- ошибка в канале приводит к посылке сигнала PIPE к процессу, который пытался выполнить чтение или запись в него

Именованный канал (named pipe) создается с помощью системного вызова `mkfifo`. Фактически, он является правильным заменителем для обмена данными через временные файлы, поскольку тот обладает следующими недостатками:

- использование медленного диска вместо более быстрой памяти
- расход места на диске (в то время как при обмене данными через FIFO после считывания они стираются); более того, место на диске может закончиться
- у процесса может не быть прав создать файл или же файл может быть испорчен/удален другим процессом

## Модель Акторов

Модель акторов Хьюита — это теоретическая модель, исследующая взаимодействие независимых программных агентов.

Актор — это независимый легковесный процесс, который взаимодействует с другими процессами только через передачу сообщений. В этой модели процессы не используют разделяемую память.

Эта модель лежит в основе языка программирования Erlang, а также библиотека для организации распределенной работы Java-приложений Akka.

## Литература

- [POSIX: командная оболочка и системные вызовы](#)
- [Advanced Linux Programming: Processes](#)
- [Daemons, Signals, and Killing Processes](#)
- [Taxonomy of UNIX IPC Methods](#)
- [Understanding the Linux Kernel - 10. Process Scheduling](#)
- [The Linux Process Scheduler](#)
- [Linux Kernel 2.4 Internals - 2. Process and Interrupt Management](#)
- [ULE: A Modern Scheduler For FreeBSD](#)
- [How Linux 3.6 nearly broke PostgreSQL](#)
- [How Erlang does scheduling](#)

# Синхронизация

## Проблема синхронизации

Синхронизация в компьютерных системах — это координация работы процессов таким образом, чтобы последовательность их операций была предсказуемой. Как правило, синхронизация необходима при совместном доступе к разделяемым ресурсам. **Критическая секция** — часть программы, в которой есть обращение к совместно используемым данным. При нахождении в критической секции двух (или более) процессов, возникает состояние гонки/состязания за ресурсы. **Состояние гонки** — это состояние системы, при котором результат выполнения операций зависит от того, в какой последовательности будут выполняться отдельные процессы в этой системе, но управлять этой последовательностью нет возможности. Иными словами, это противоположность правильной синхронизации.

Для избежания гонок необходимо выполнение таких условий:

- **Взаимного исключения**: два процесса не должны одновременно находиться в одной критической области
- **Прогресса**: процесс, находящийся вне критической области, не может блокировать другие процессы
- **Ограниченнего ожидания**: невозможна ситуация, в которой процесс вечно ждет попадания в критическую область
- Также в программе, в общем случае, не должно быть предположений о скорости или количестве процессоров

### Пример условий гонок: i++

```
movl i, $eax // i - метка адреса переменной i в памяти
incl $eax
movl $eax, i
```

Поскольку процессор не может модифицировать значения в памяти непосредственно, для этого ему нужно загрузить значение в регистр, изменить его, а затем снова выгрузить в память. Если в процессе выполнения этих трех инструкций процесс будет прерван, может возникнуть непредсказуемый результат, т.е. имеет место условие гонки.

## Классические задачи синхронизации

Классические задачи синхронизации — это модельные задачи, на которых исследуются различные ситуации, которые могут возникать в системах с разделяемым доступом и конкуренцией за общие ресурсы. К ним относятся задачи: Производитель-потребитель, Читатели-писатели, Обедающие философы, Спящий парикмахер, Курильщики сигарет, Проблема Санта-Клауса и др.

Задача **Производитель-потребитель** (также известна как задача ограниченного буфера): 2 процесса — производитель и потребитель — работают с общим ресурсом (буфером), имеющим максимальный размер N. Производитель записывает в буфер данные последовательно в ячейки 0,1,2,..., пока он не заполниться, а потребитель читает данные из буфера в обратном порядке, пока он не опустеет. Запись и считывание не могут происходить одновременно.

## Наивное решение

```
int buf[N];
int count = 0;
void producer() {
    while (1) {
        int item = produce_item();
        while (count == N - 1)
            /* do nothing */;
        buf[count] = item;
        count++;
    }
}
void consumer() {
    while (1) {
        while (count == 0)
            /* do nothing */;
        int item = buf[count - 1];
        count--;
        consume_item(item);
    }
}
int main() {
    make_thread(&producer);
    make_thread(&consumer);
}
```

Проблема синхронизации в этом варианте: если производитель будет прерван потребителем после того, как запишет данные в буфер `buf[count] = item`, но

до того, как увеличит счетчик, то потребитель считает из буфера элемент перед только что записанным, т.е. в буфере образуется дырка. После того как производитель таки увеличит счетчик, счетчик как раз будет указывать на эту дырку. Симметричная проблема есть и у потребителя.

Также у этой задачи может быть множество модификаций: например, количество производителей и потребителей может быть больше 1. В этом случае добавляются новые проблемы синхронизации.

Еще одна проблема этого решения — это бессмысленная трата вычислительных ресурсов: циклы `while (count == 0) /* do nothing */ ;` — т.н. **занятое ожидание** (busy loop, busy waiting) или же **поллинг** (pooling) — это ситуация, когда процесс не выполняет никакой полезной работы, но занимает процессор и не дает в это время работать на нем другим процессам. Таких ситуаций нужно по возможности избегать.

## Алгоритмы программной синхронизации

Программный алгоритм синхронизации — это алгоритм взаимного исключения, который не основан на использовании специальных команд процессора для запрета прерываний, блокировки шины памяти и т. д. В нем используются только общие переменные памяти и цикл для ожидания входа в критическую секцию исполняемого кода. В большинстве случаев такие алгоритмы не эффективны, т.к. используют поллинг для проверки условий синхронизации.

Пример программного алгоритма — алгоритм Петерсона:

```
int interested[2];
int turn;
void enter_section(int process_id) {
    int other = 1 - process_id;
    interested[process_id] = 1;
    turn = other;
    while (turn == other && interested[other])
        /* busy waiting */;
}
void leave_section(int process_id) {
    interested[process_id] = 0;
}
```

См. также алгоритм Деккера, алгоритм пекарни Лемпорта и др.

# Аппаратные инструкции синхронизации

Аппаратные инструкции синхронизации реализуют **атомарные** примитивные операции, на основе которых можно строить механизмы синхронизации более высокого уровня. Атомарность означает, что вся операция выполняется как целое и не может быть прерывана посередине. Атомарные примитивные операции для синхронизации, как правило, выполняют вместе 2 действия: запись значения и проверку предыдущего значения. Это дает возможность проверить условие и сразу записать такое значение, которое гарантирует, что условие больше не будет выполняться.

## Try-and-set lock (TSL)

Инструкции типа try-and-set записывают в регистр значение из памяти, а в память — значение 1. Затем они сравнивают значение в регистре с 0. Если в памяти и был 0 (т.е. доступ к критической области был открыт), то сравнение пройдет успешно, и в то же время в память будет записан 1, что гарантирует, что в следующий раз сравнение уже не будет успешным, т.е. доступ закроется.

Реализация критической секции с помощью TSL:

```
enter_region:  
    TSL REGISTER, LOCK  
    CMP REGISTER, 0  
    JNE ENTER_REGION  
    RET  
leave_region:  
    MOV LOCK, 0  
    RET
```

То же самое на C:

```
void lock(int *lock) {  
    while (!test_and_set(lock))  
        /* busy waiting */;  
}
```

## Compare-and-swap (CAS)

Инструкции типа compare-and-swap записывают в регистр новое значение и при этом проверяют, что старое значение в регистре равно запомненному ранее значению.

В x86 называется CMPXCHG.

Аналог на С:

```
int compare_and_swap(int* reg, int oldval, int newval) {  
    int old_reg_val = *reg;  
    if (old_reg_val == oldval)  
        *reg = newval;  
    return old_reg_val;  
}
```

Проблема АВА (ABBA): CAS инструкции не могут отследить ситуацию, когда значение в регистре было изменено на новое, а потом снова было возвращено к предыдущему значению. В большинстве случаев это не влияет на работу алгоритма, а в тех случаях, когда влияет, необходимо использовать инструкции с проверкой на такую ситуацию, такие как LL/SC.

## Другие аппаратные инструкции

- Двойной CAS
- Fetch-and-add
- Load-link/store-conditional (LL/SC)

## Системные механизмы синхронизации

С помощью аппаратных инструкций можно реализовать более высокоровневые конструкции, которые могут ограничивать доступ в критическую область, а также сигнализировать о каких-то событиях.

Самым простым вариантом ограничения доступа в критическую область является **переменная-замок**: если ее значение равно 0, то доступ открыт, а если 1 — то закрыт. У нее есть 2 атомарные операции:

- заблокировать (**lock**) — проверить, что значение равно 0, и устанавливает его в 1 или же ждать, пока оно не станет 0
- разблокировать (**unlock**), которая устанавливает значение в 1

Также полезной может быть операция попробовать заблокировать (**trylock**), которая не ждет пока значение замка станет 0, а сразу возвращает ответ о невозможности заблокировать замок.

## Спинлок

Спинлок — это замок, ожидание при блокировке которого реализовано в виде

занятого ожидания, т.е. поток "крутился" в цикле, ожидая разблокировки замка.

Реализация на ассемблере с помощью CAS:

```
lock: # 1 = locked, 0 = unlocked.
      dd 0
spin_lock:
      mov eax, 1
loop:
      xchg eax, [lock]
      # Atomically swap the EAX register with
      # the lock variable.
      # This will always store 1 to the lock,
      # leaving previous value in the EAX register
      test eax, eax
      # Test EAX with itself. Among other things, this
      # sets the processor's Zero Flag if EAX is 0.
      # If EAX is 0, then the lock was unlocked and
      # we just locked it. Otherwise, EAX is 1
      # and we didn't acquire the lock.
      jnz loop
      # Jump back to the XCHG instruction if Zero Flag
      # is not set, the lock was locked,
      # and we need to spin.
      ret
spin_unlock:
      mov eax, 0
      xchg eax, [lock]
      # Atomically swap the EAX register with
      # the lock variable.
      ret
```

Использование спинлока целесообразно только в тех областях кода, которые не могут вызвать блокировку, иначе все время, отведенное планировщиком потоку, ожидающему на спинлоке, будет потрачено на ожидание и при этом другие потоки не будут работать.

## Семафоры

Семафор — это примитив синхронизации, позволяющий ограничить доступ к критической секции только для N потоков. При этом, как правило, семафор позволяет реализовать это без использования занятого ожидания.

Концептуально семафор включает в себя счетчик и очередь ожидания для

потоков. Интерфейс семафора состоит из двух основных операций: опустить (**down**) и поднять (**up**). Операция опустить атомарно проверяет, что счетчик больше 0 и уменьшает его. Если счетчик равен 0, поток блокируется и ставиться в очередь ожидания. Операция поднять увеличивает счетчик и посыпает ожидающим потокам сигнал пробудиться, после чего один из этих потоков сможет повторить операцию опустить.

Бинарный семафор — это семафор с  $N = 1$ .

## Мьютекс (mutex)

Мьютекс — от словосочетания *mutual exclusion*, т.е. взаимное исключение — это примитив синхронизации, напоминающий бинарный семафор с дополнительным условием: разблокировать его должен тот же поток, который и заблокировал.

Реализация мьютекса с помощью примитива CAS:

```
void acquire_mutex(int *mutex) {
    while (cas(mutex, 1, 0)) /* busy waiting */;
}
void release_mutex(int *mutex) {
    *mutex = 1;
}
```

Стоит отметить, что не все системы предоставляют гарантии того, что мьютекс будет разблокирован именно заблокировавшим его потоком. В приведенном выше примере это условие как раз не проверяется.

Мьютекс с возможностью повторного входа (*re-entrant mutex*) — это мьютекс, который позволяет потоку несколько раз блокировать его.

## RW lock

RW lock — это особый вид замка, который позволяет разграничить потоки, которые выполняют только чтение данных, и которые выполняют их модификацию. Он имеет операции заблокировать на чтение (**rdlock**), которая может одновременно выполняться несколькими потоками, и заблокировать на запись (**wlock**), которая может выполняться только 1 потоком. Также правильные реализации RW-замка позволяют избежать проблемы инверсии приоритетов (см. ниже).

## Переменные условия и мониторы

**Монитор** — это механизм синхронизации в объектно-ориентированном программировании, при использовании которого объект помечается как синхронизированный и компилятор добавляет к вызовам всех его методов (или только выделенных синхронизированных методов) блокировку с помощью мьютекса. При этом код, использующий этот объект, не должен заботиться о синхронизации. В этом смысле монитор является более высокоуровневой конструкцией, чем семафоры и мьютексы.

**Переменная условия** (condition variable) — примитив синхронизации, позволяющий реализовать ожидание какого-то события и оповещение о нем. Над ней можно выполнять такие действия:

- ожидать (**wait**) сообщения о каком-то событии
- сигнализировать событие всем потокам, ожидающим на данной переменной. Сигнализация может быть блокирующей (**signal**) — в этом случае управление переходит к ожидающему потоку, — и неблокирующей (**notify**) — в этом случае управление остается у сигнализирующего потока

Большинство мониторов поддерживают внутри себя использование переменных условия. Это позволяет нескольким потокам заходить в монитор и передавать управление друг другу через эту переменную.

См. [http://en.wikipedia.org/wiki/Monitor\\_\(synchronization\)](http://en.wikipedia.org/wiki/Monitor_(synchronization))

## Интерфейс синхронизации

POSIX Threads (Pthreads) — это часть стандарта POSIX, которая описывает базовые примитивы синхронизации, поддерживаемые в Unix системах. Эти примитивы включают семафор, мьютекс и переменные условия.

**Futex** (быстрый замок в пользовательском пространстве) — это реализация мьютекса в Unix-системах, которая оптимизирована для минимального использования функций ядра ОС, за счет чего достигается более быстрая работа с ним. С помощью фьютексов в Linux реализованы семафоры и переменные условия.

Над фьютексами можно делать такие базовые операции:

- ожидать — `wait(addr, val)` — проверяет, что значение по адресу `addr` равно `val`, и, если это так, то переводит поток в состояние ожидания, а иначе продолжает работу (т.е. входит в критическую область)

- разбудить — `wake(addr, n)` — посыпает  $n$  потокам, ожидающим на фьютексе по адресу `addr` оповещение о необходимости проснуться

## Проблемы синхронизации

Помимо условий гонки и занятого ожидания неправильная синхронизация может привести к следующим проблемам:

**Тупик/взаимная блокировка** (deadlock) — ситуация, когда 2 или более потоков ожидают разблокировки замков друг от друга и не могут продвинуться. Простейший пример кода, который может вызвать взаимную блокировку:

```
void thread1() {  
    acquire(mutex1);  
    do_something1();  
    acquire(mutex2);  
    do_something_else1();  
    release(mutex2);  
    release(mutex1);  
}  
void thread2() {  
    acquire(mutex2);  
    do_something2();  
    acquire(mutex1);  
    do_something_else2();  
    release(mutex1);  
    release(mutex2);  
}
```

**Живой блок** (livelock) — ситуация с более, чем двумя потоками, при которой потоки ожидают разблокировки друг от друга, при этом могут менять свое состояние, но не могут продвинуться глобально в своей работе. Такая ситуация более сложная, чем тупик и возникает значительно реже: как правило, она связана с временными особенностями работы программы.

**Инверсия приоритета** — ситуация, когда более поток с большим приоритетом вынужден ожидать потоки с меньшим приоритетом из-за неправильной синхронизации.

**Голодание** — ситуация, когда поток не может получить доступ к общему ресурсу и не может продвинуться. Такая ситуация может быть следствием как тупика, так и инверсии приоритета.

## Способы предотвращения тупиковых ситуаций

Все способы борьбы с тупиками не являются универсальными и могут работать только при определенных условиях. К ним относятся:

- монитор тупиков, который следит за тем, чтобы ожидание на каком-то из замков не длилось слишком долго, и рестартует ожидающий поток в таком случае
- нумерация замков и их блокировка только в монотонном порядке
- Алгоритм банкира

## Неблокирующая синхронизация

Неблокирующая синхронизация — это группа подходов, которые ставят своей целью решить проблемы синхронизации альтернативным путем без явного использования замков и основанных на них механизмов. Эти подходы создают новую **конкурентную парадигму** программирования, что можно сравнить с появлением структурной парадигмы как отрицанием подхода к написанию программ с использованием низкоуровневой конструкции *goto* и перехода к использованию структурных блоков: итерация, условное выражение, цикл.

### Ничего общего (**shared-nothing**)

Архитектуры программ без общего состояния рассматривают вопросы построения систем из взаимодействующих компонент, которые не имеют разделяемых ресурсов и обмениваются информацией только через передачу сообщений. Такие системы, как правило, являются намного менее связными, и поэтому лучше поддаются масштабированию и являются менее чувствительными к отказу отдельных компонент.

Теоретические работы на этот счет: Взаимодействующие параллельные процессы (Communicating Parallel Processes, CPP) и модель Акторов.

Практическая реализация этой концепции — язык Erlang. В этой модели единицей вычисления является легковесный процесс, который имеет "почтовый ящик", на который ему могут отправляться сообщения от других процессов, если они знают его ID в системе. Отправка сообщений является неблокирующей (асинхронной), а прием является синхронным: т.е. процесс может заблокироваться в ожидании сообщения. При этом время блокировки может быть ограничено программно.

### CSP

Взаимодействующие последовательные процессы (Communicating Sequential

Processes, CSP) — это еще один подход к организации взаимодействия без использования замков. Единицами взаимодействия в этой модели являются процессы и каналы. В отличие от модели CPP, пересылка данных через канал в этой модели происходит, как правило, синхронно, что дает возможность установить определенную последовательность выполнения процессов. Данная концепция реализована в языке программирования Go.

## Программная транзакционная память

Транзакция — это группа последовательных операций, которая представляет собой логическую единицу работы с данными. Транзакция может быть выполнена либо целиком и успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта. В теории баз данных существует концепция ACID, которая описывает условия, которые накладываются на транзакции, чтобы они имели полезную семантику. **A** означает **атомарность** — транзакция должна выполняться как единое целое. **C** означает **целостность** (consistency) — в результате транзакции будут либо изменены все данные, с которыми работает транзакция, либо никакие из них. **I** означает **изоляция** — изменения данных, которые производятся во время транзакции, станут доступны другим процессам только после ее завершения. **D** означает **сохранность** — после завершения транзакции система БД гарантирует, что ее результаты сохраняются в долговременном хранилище данных. Эти свойства, за исключением, разве что, последнего могут быть применимы не только к БД, но и к любым операциям работы с данными. Программная система, которая реализует транзакционные механизмы для работы с разделяемой памятью — это Программная транзакционная память (Software Transactional Memory, STM). STM лежит в основе языка программирования Clojure, а также доступна в языках Haskell, Python (в реализации PyPy) и других.

## Литература

- [A Beautiful Race Condition](#)
- [Java's Atomic and volatile, under the hood on x86](#)
- [Mutexes and Condition Variables using Futexes](#)
- [Common Pitfalls in Writing Lock-Free Algorithms](#)
- [Values and Change](#)

- Beautiful Concurrency
- Programming Erlang: 8. Concurrent Programming

# Файловая система

## Файловая система

Файловая система (ФС) — это компонент ОС, отвечающий за постоянное хранение данных.

Задачи:

- хранение данных в потенциально неограниченных объемах
- долгосрочное сохранение данных (persistence)¶
- одновременная доступность данных многим процессам

В отличие от оперативной памяти ФС являются постоянной памятью, поэтому на первое место для них выходят сохранность и доступность данных, а затем уже стоит быстродействие.

Некоторые из видов файловых систем:

- Дисковые
- Виртуальные (в памяти и не только)
- Сетевые
- Распределенные
- Мета ФС (ФС, которые используют другие ФС для организации хранения данных, а сами добавляют особую логику работы)

### [Список ФС](#)

Файловый интерфейс — это простой и удобный способ работы с данными, поэтому многие ОС распространяют его на другие объекты, помимо файлов данных. Например, в Unix все устройства подключаются в системе как специальные файлы (символико-специфичные — для устройств с последовательным вводом-выводом; и блочно-специфичные — для устройств с буферизированным вводом-выводом). Кроме того, в Linux есть специальные виртуальные файловые системы: `sysfs`, которая оперирует файлами, отображающими системные структуры данных из памяти ядра, а также `tmpfs`, которая позволяет создавать файлы в оперативной памяти. Такой подход привел к тому, что Unix стал известен как **файл-центричная** ОС, хотя есть другие ОС, которые еще далее продвинулись в этом. Например, такие объекты, как сокеты (см. лекцию про работу с сетью) в Unix имеют собственные системные вызовы, в то время как в системе Plan 9 (которая стала развитием

идей Unix) они доступны через тот же файловый интерфейс.

## Файл

Файл — это именованная область диска. (Неверное и устаревшее определение). Верное определение: Файл — это объект файловой системы, содержащий информацию о размещении данных в хранилище. При этом хранилище может быть как физическим запоминающим устройством (диском, магнитной лентой и т.д.), так и виртуальным устройством, за которым стоит оперативная память, какое-либо устройство ввода-вывода, сетевое соединение или же другая файловая система.

Основные операции над файлами:

- `create` — создание
- `delete` — удаление
- `open` — открытие (на запись, чтение или же на то и другое вместе)
- `close` — закрытие
- `read` — чтение
- `write` — запись
- `append` — запись в конец
- `seek` — переход на заданную позицию для последующего чтения/записи
- `getattr` — получение атрибутов файла
- `setattr` — установка атрибутов файла
- `lock` — блокировка файла для эксклюзивной работы с ним (в большинстве ОС блокировка файлов реализуется в виде рекомендательных, а не обязательных замков)

[Файловый дескриптор](#) — это уникальный для процесса номер в таблице дескрипторов процесса (которая хранится в ядре ОС), который операционная система предоставляет ему, чтобы выполнять указанные операции с файлом. Системы Windows оперируют схожей концепцией — описатель файла (*file handle*).

Помимо указанных выше операций в Unix используются следующие важные системные вызовы для работы с файловыми дескрипторами:

- `dup` — создание копии записи в таблице дескрипторов с новым индексом, указывающим на тот же файл
- `dup2` — "перенаправление" одной записи на другую

- `fcntl` — управление нестандартными атрибутами и свойствами файлов, которые могут поддерживаться теми или иными файловыми системами

## Директория

Директория — это объект файловой системы, содержащий информацию о структуре и размещении файлов. Часто это тоже файл, только особый.

В большинстве ФС директории объединяются в дерево директорий, таким образом создавая **иерархическую** систему хранения информации. Это дерево имеет корень, который в Unix системах называется `/`. Положение файла в этом дереве — это **путь** к нему от корня — т.н. **абсолютный путь**. Кроме того, можно говорить об **относительном пути** от выбранной директории к другому объекту ФС. Относительный путь может содержать особое обозначение `..`, которое указывает на предка (родителя) директории в дереве директорий.

Логическое дерево директорий может объединять больше одной ФС.

Включение ФС в это дерево называется **монтированием**. В результате монтирования корень ФС привязывается к какой-то директории внутри дерева. Для первой монтируемой системы ее корень привязывается к корню самого дерева.

В Unix также реализована операция `chroot`, которая позволяет изменить корень текущего дерева на одну из директорий внутри него. В рамках одного сеанса работы, выполнив ее, уже нельзя вернуться назад, т.е. получить доступ ко всем узлам дерева, которые не являются потомками нового корня ФС.

Существует два подхода к привязке файлов к директориям: в большинстве ФС эта привязка является косвенной — через использование **ссылки** на отдельный объект, хранящий метаданные файла. Такие ссылки называются **жесткими** (hard link).

В таких системах один файл может иметь несколько ссылок на себя из разных директорий. В такой системе создание нового файла происходит в 2 этапа: сначала выполняется операция `create`, которая создает сам файловый объект, а затем `link`, которая привязывает его к конкретной директории.

Операция `link` может выполняться несколько раз, и каждый файл хранит количество ссылок на себя из разных директорий. Удаление файла происходит с точностью дооборот: выполняется операция `unlink`, после чего файл перестает быть привязанным к директории. Если при этом его счетчик ссылок становится равным 0, то система выполняет над файловым объектом операцию `delete`. Однако есть и более примитивные системы, в которых метаданные о файле хранятся прямо в директории. Фактически, в таких ФС реализована однозначная привязка файла к единственной директории.

В обоих видах систем часто поддерживается также концепция **символических ссылок** (реализуемых в виде специальных файлов), которые ссылаются не непосредственно на файловый объект ФС, а на объект, находящийся по определенному пути. В этом случае ФС уже не может обеспечить проверку существования такого объекта и управление другими его свойствами, как это делается для жестких ссылок, но, с другой стороны, символические ссылки позволяют ссылаться на файлы в одном логическом дереве директорий, но за пределами текущей ФС.

Операции над директориями:

- `create` — создать
- `delete` — удалить
- `readdir (list)` — получить список непосредственных потомков данной директории (файлов и других директорий)<sup>9</sup>
- `opendir` — открыть директорию для изменения информации в ней
- `closedir` — закрыть директорию
- `link/unlink`

## Схемы размещения файлов

Схема размещения файлов — это общий подход к хранению данных и метаданных файла на запоминающем устройстве. Основные критерии эффективности той или иной схемы — это утилизация диска, быстродействие операций чтения, записи и поиска, а также устойчивость к сбоям.

**Фрагментация** — это отсутствие технической возможности использовать часть пространства хранилища данных из-за того или иного размещения данных в нем. Виды фрагментации:

- внешняя — невозможность использования целых блоков
- внутренняя — невозможность использования частей отдельных блоков

Внутренняя фрагментация неизбежна в любом хранилище, которое оперирует блоками большего размера, чем единица хранения (например, жесткий диск хранит данные побайтно, но файлы могут начинаться только с начала логического блока, как правило, имеющего размеры порядка килобайт). Возможность появления внешней фрагментации зависит от схемы размещения данных.

## Непрерывная/последовательная схема

В этой схеме файл хранится как одна непрерывная последовательность блоков. Для указания размещения файла достаточно задать адрес его начального блока и общий размер.

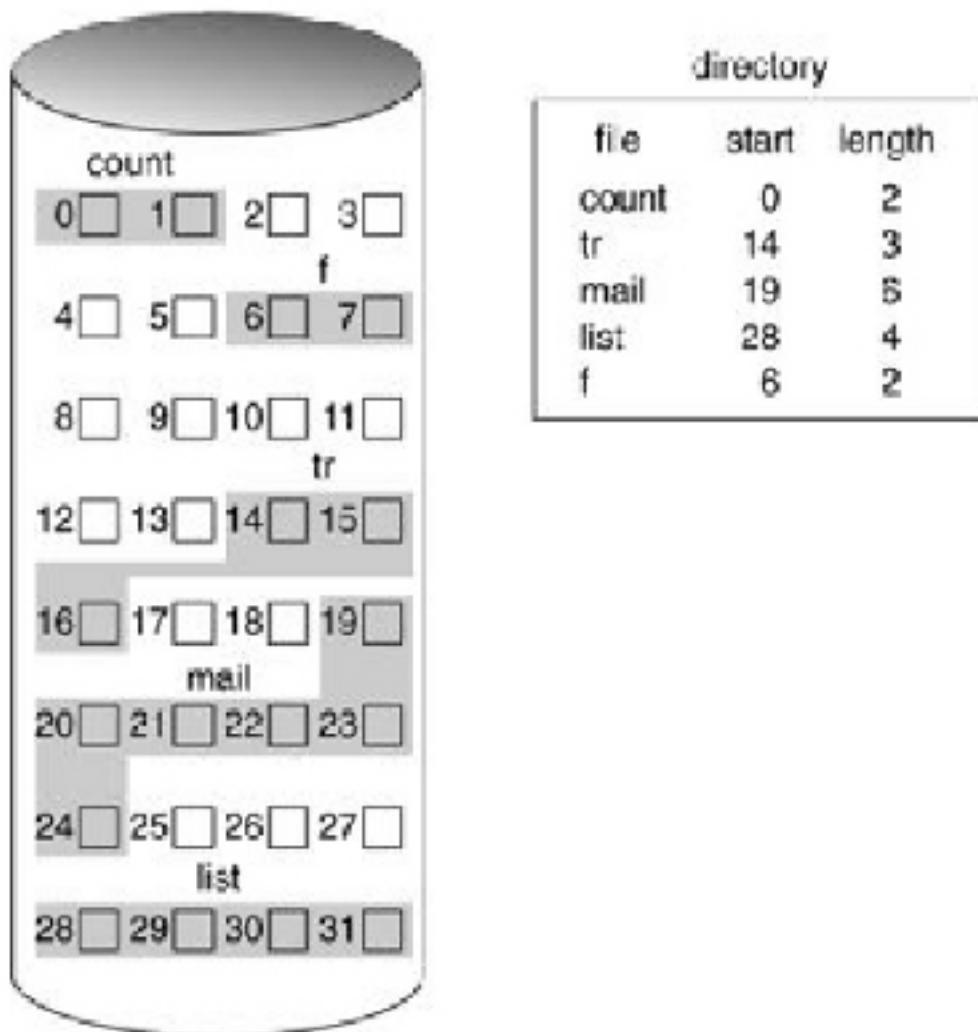


Рис. 8.1. Непрерывная схема размещения файлов

Преимущества:

- легко реализовать
- самая лучшая производительность

Недостатки:

- внешняя фрагментация
- необходимость реализовать учет дырок

- проблемы с ростом размера файла

Это самая простая схема размещения. Она идеально подходит для любых носителей, который предполагают использование только в режиме для чтения, или же существенное (на порядки) превышение числа операций чтения над записью.

## Схема размещения связным списком

В этой схеме блоки файла могут находиться в любом месте диска и не быть упорядочены каким-то образом, но каждый блок должен хранить ссылку на последующий за ним. Для указания размещения файла достаточно задать адрес его первого блока.

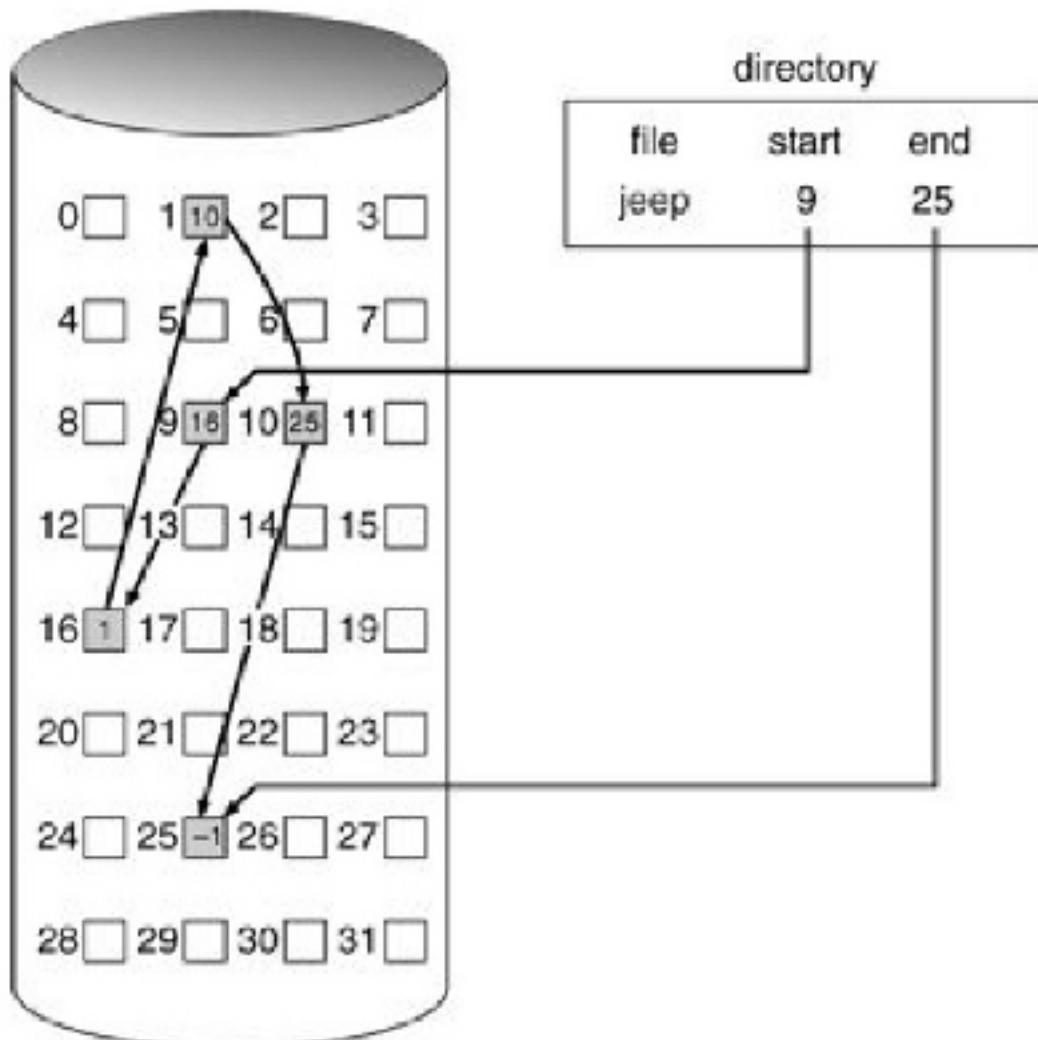


Рис. 8.2. Схема размещения файлов связным списком

Преимущества:

- нет внешней фрагментации
- простота реализации

Недостатки:

- самая худшая производительность (особенно на магнитных дисках)
- плохая устойчивость к ошибкам: повреждение одного блока в середине файла приведет к тому, что все последующие за ним блоки будут недоступны
- не круглая цифра (в степенях 2) доступного места в блоке из-за использования части пространства блока для хранения указателя на следующий блок

Поэтому в чистом виде такая схема редко применяется. Перечисленные недостатки в основном устраняет **табличная** реализация этой схемы. В ней информация о следующих блоках связного списка выносится из самих блоков в отдельную таблицу, которая размещается в заранее заданном месте диска. Каждая запись в таблице хранит номер следующего блока для данного файла (или же индикатор того, что данный блок не занят либо же испорчен). Такая технология называется **Таблицей размещения файлов** (см. [FAT](#)). Недостатком табличной схемы является централизация всех метаданных в таблице, что приводит к опасности потери всей ФС в случае непоправимого повреждения таблицы, а также к тому, что для больших дисков эта таблица будет иметь большой объем, а она должна все время быть полностью доступной в памяти.

## Индексная схема

В этой схеме блоки разделяют на 2 типа: те, которые используются для хранения данных файла, и те, которые хранят метаданные — т.н. индексные узлы (**inode**). Таким образом, в отличие от предыдущей схемы, метаданные о файлах хранятся распределено в ФС, что делает этот подход более эффективным и отказоустойчивым. Кроме того, эта схема соответствует собственно иерархической модели ФС и тривиально поддерживает операции `link/unlink`: директория хранит ссылки на индексные узлы привязанных к ней файлов. Поскольку индексные узлы — это обычные блоки диска, к ним применяются те же методы работы, что и для обычных блоков (например, кеширование).

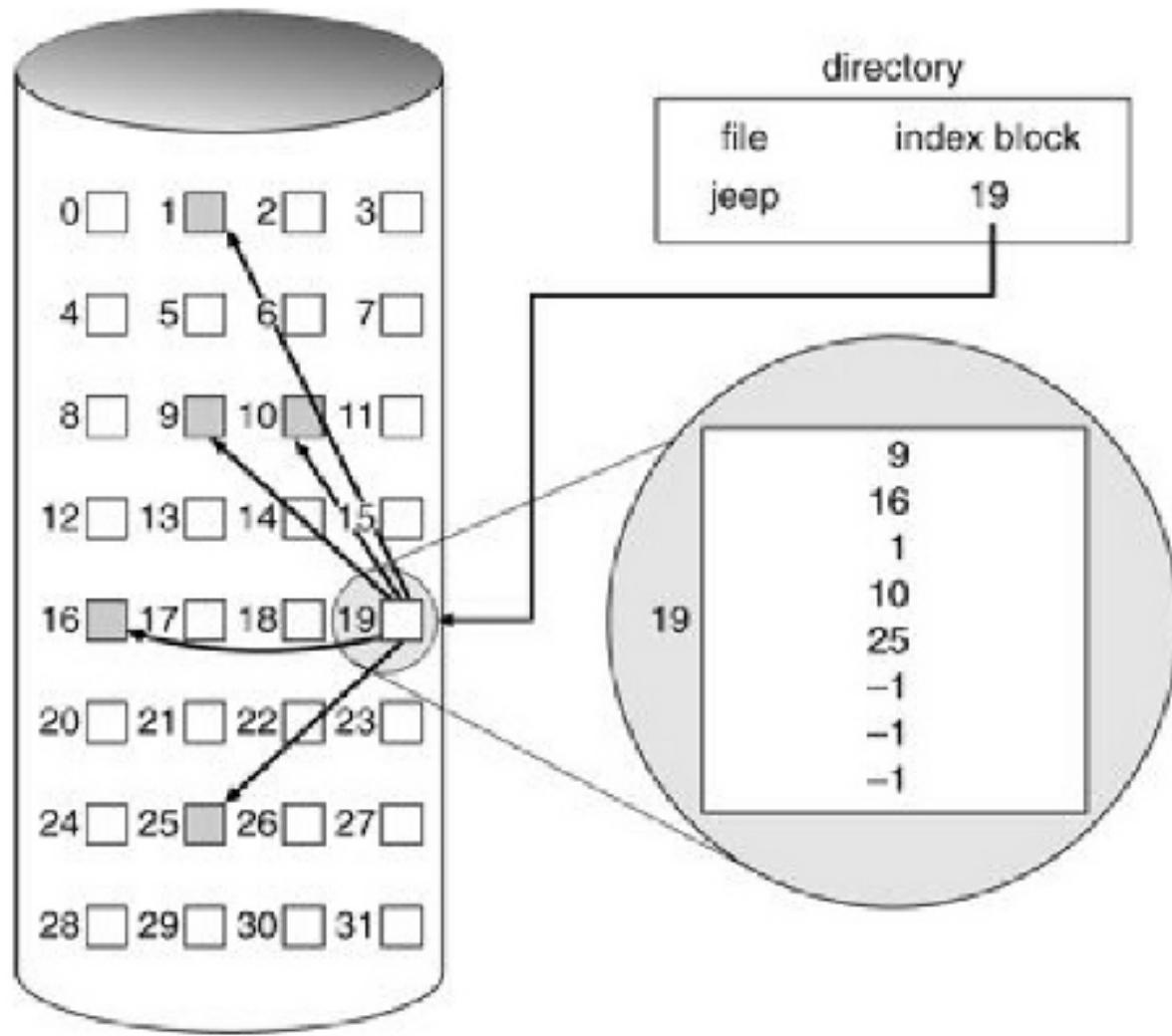


Рис. 8.3. Индексная схема размещения файлов

**Преимущества:**

- концептуальное соответствие между логической и физической схемой хранения
- большая эффективность и быстродействие
- большая отказоустойчивость

**Недостатки:**

- фиксированный размер индексного узла, что налагает ограничения на размер файла (для решения этой проблемы используются непрямые inode'ы, которые хранят ссылки не на блоки данных, а на inode'ы следующего уровня)

# Оптимизация работы ФС

Важным параметром, влияющим на оптимизацию ФС является средний размер файла. Он сильно зависит от сценариев использования системы, но проводятся исследования, которые замеряют это число для типичной файловой системы. В 90-х годах средний размер файла составлял 1КБ, в 2000-ных — 2 КБ, на данный момент — 4КБ и более.

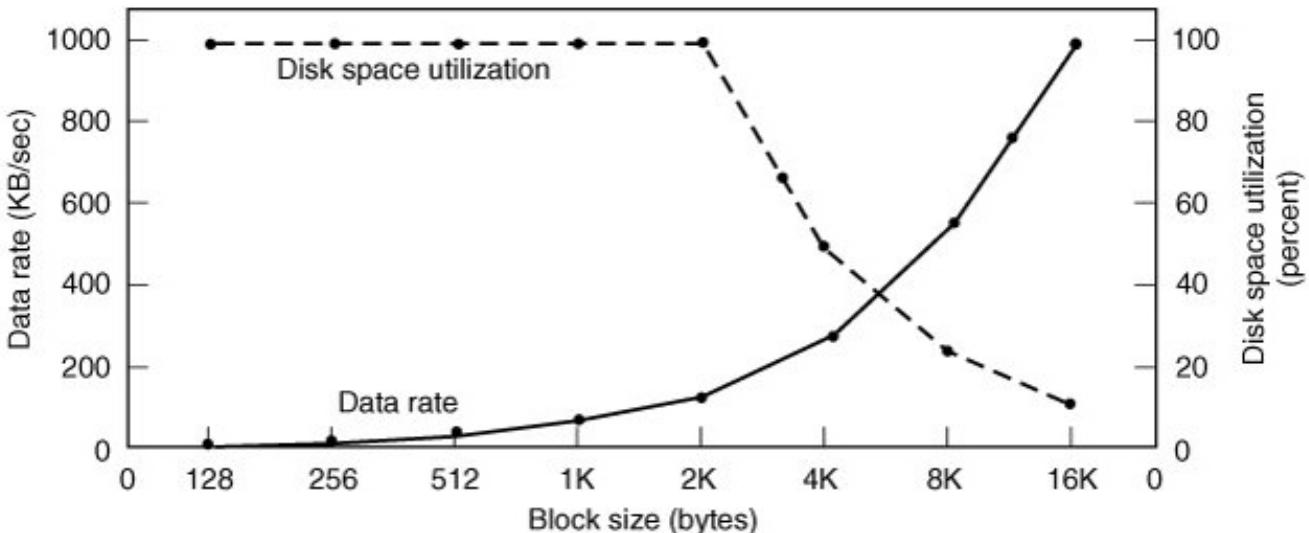


Рис. 8.4. Связь размера блока с быстродействием и утилизацией диска при среднем размере файла в системе 2 КБ

Важный метод оптимизации ФС — это кеширование. В отличие от кеширования процессора в случае дисковых хранилищ точный LRU возможен, но иногда он может быть даже вреден. Поэтому для дисков часто используется **сквозной кеш** (изменение данных в нем сразу же вызывает изменение данных на диске), хотя он менее эффективен. Хотя в Unix системах для увеличения быстродействия исторически принято было использовать операцию sync, которая периодически, а не постоянно, сохраняла изменения данных в дисковом кеше на носитель.

Также оптимизация очень существенно зависит от физических механизмов хранения данных, т.к. профиль нагрузки, например, для магнитного и твердотельных дисков совершенно разные, не говоря о ФС, работающих по сети.

Подходы к оптимизации ФС, использующих жесткий диск:

- чтение блоков наперед (однако зависит от шаблона использования: последовательный или произвольный доступ к данным в файле)

- уменьшение свободного хода дисковой головки за счет помещения блоков в один цилиндр
- использование [ФС на основе журнала](#)

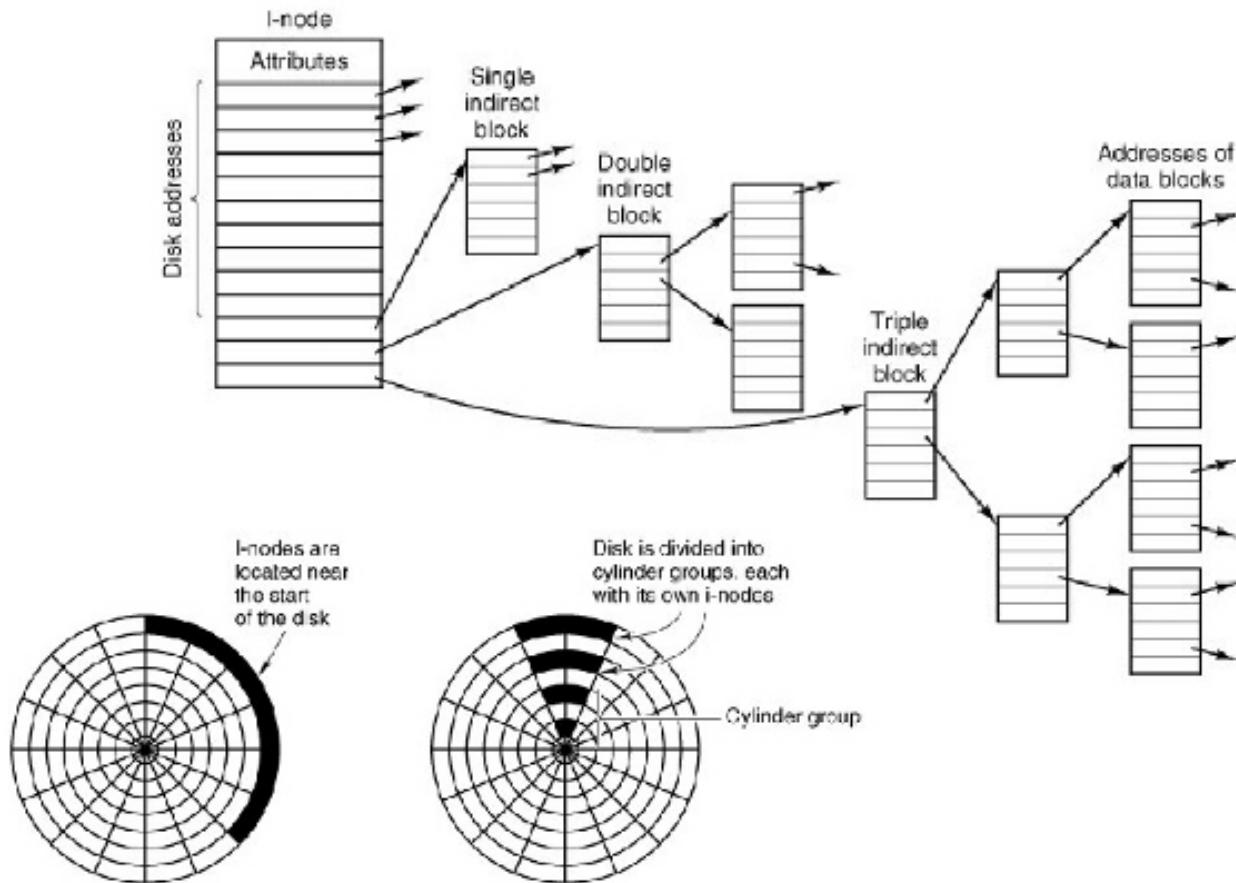


Рис. 8.5. Подходы к оптимизации индексной схемы

## Литература

- [Another Level of Indirection](#)
- [Inferno OS Namespaces](#)
- [The Google File System](#)
- [Everything you never wanted to know about file locking](#)
- [Building the next generation file system for Windows: ReFS](#)

# Взаимодействие с сетью

"Сеть — это компьютер" (лозунг корпорации Sun)

Поддержка работы с сетью на первый взгляд не является функцией ядра ОС, однако на практике большинство ОС реализуют ее в ядре. Для этого есть несколько причин:

- работа с сетью нужна подавляющему большинству нетривиальных программ, поэтому логично, что ОС должна предоставить для них сетевой сервис, абстрагированный от разнородных аппаратных средств и низкоуровневых протоколов поддержки соединения

Любая программа стремится расширяться до тех пор, пока с его помощью не станет возможно читать почту. Те программы, которые не расширяются настолько, заменяются теми, которые расширяются. (Закон оборачивания софта Jamie Zawinski)

- работа с сетью должна быть быстрой
- ограничения безопасности, связанные с работой с сетью
- относительная простота реализации: небольшой набор стандартных протоколов, среди которых основные — это IP, TCP и UDP

Также в основе реализации компьютерных сетей лежит Принцип устойчивости (Закон Постела):

Будьте консервативными в том, что отправляете, и либеральными в том, что принимаете от других.

## "Заблуждения" программистов про сеть

Разработка распределенных программ, использующих сеть, отличается от разработки программ, работающих на одном компьютере. Эти различия выражены в следующем списке т.н. "заблуждений" программистов про сеть:

- сеть надежна
- расходы на транспорт нулевые
- задержка нулевая
- часы синхронизированы
- пропускная способность неограниченна

- топология сети неизменна
- сеть гомогенна
- есть только один администратор
- сеть безопасна

## Модель OSI

[Сетевая модель OSI](#) — это теоретическая эталонная модель сетевого взаимодействия открытых систем. В ней реализован принцип разделения забот (separation of concerns), который выражен в том, что взаимодействие происходит на 7 разных уровнях, каждый из которых отвечает за решение одной проблемы:

- 7й - Прикладной (application) — доступ к сетевым службам прикладных приложений, данные представляются в виде "запросов" (requests)
- 6й - Представления (presentation) — кодирование и шифрование данных
- 5й - Сеансовый (session) — управление сеансом связи
- 4й - Транспортный (transport) — связь между конечными пунктами (которые не обязательно связаны непосредственно) и надежность, данные представляются в виде "сегментов" (datagrams)
- 3й - Сетевой (network) — определение маршрута и логическая адресация, обеспечение связи в рамках сети, данные представляются в виде "пакетов" (packets)
- 2й - Канальный (data link) — физическая адресация, обеспечение связи точка-точка, данные представляются в виде "кадров" (frames)
- 1й - Физический (physical) — работа со средой передачи, сигналами и двоичными данными (битами)

При обеспечении связи между узлами (хостами) данные проходят процесс "погружения" с прикладного уровня на физический на отправителе и обратный процесс на получателе.

## Стек протоколов TCP/IP

На практике доминирующей моделью сетевого взаимодействия является стек протоколов TCP/IP, который в целом соответствует модели OSI, однако не регламентирует обязательное наличие всех уровней в ней. Как следует из названия, обязательными протоколами в ней являются TCP (или его альтернатива UDP), а также IP, которые реализуют транспортный и сетевой уровень модели OSI.

Уровни TCP/IP стека:

- 4й - Прикладной уровень (Process/Application) — соответствует трем верхним уровням модели OSI (однако, не обязательно реализует функциональность их всех)
- 3й - Транспортный уровень (Transport) — соответствует транспортному уровню модели OSI
- 2й - Межсетевой уровень (Internet) — соответствует сетевому уровню модели OSI
- 1й - Уровень сетевого доступа (Network Access) — соответствует двум нижним уровням модели OSI

В этой модели верхний и нижний уровни включают в себя несколько уровней модели OSI и в разных случаях они могут быть реализованы как одним протоколом взаимодействия, так и несколькими (соответствующими отдельным уровням). Например, протокол HTTP реализует уровни прикладной и представления, а протокол TLS — сеансовый и представления, а в сочетании между собой они могут покрыть все 3 верхних уровня. При этом протокол HTTP работает и самостоятельно, и в этом случае, поскольку он не реализует сеансовый уровень, HTTP-соединения называют "stateless", т.е. не имеющими состояния.

Модель TCP/IP также называют песочными часами, поскольку посередине в ней находится один протокол, IP, а протоколы под ним и над являются очень разнообразными и покрывают разные сценарии использования.

Стандартизация протокола посередине дает большую гибкость низкоуровневым протоколам (которая нужна из-за наличия разных способов соединения) и высокоуровневым (нужную из-за наличия разных сценариев работы).

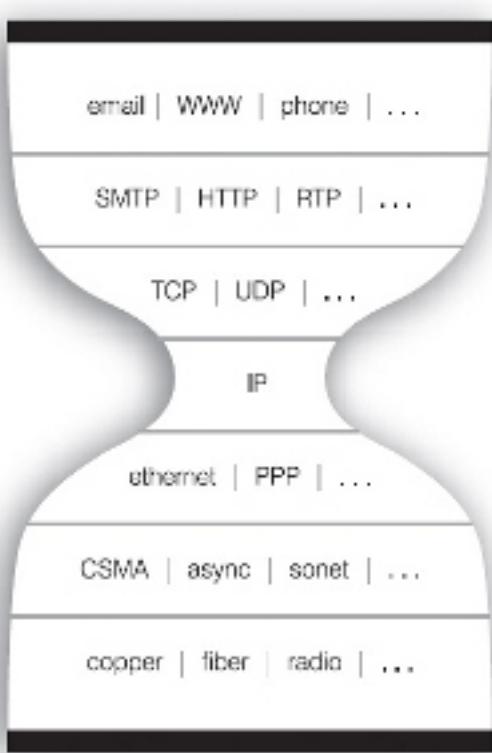


Рис. 9.1. TCP/IP стек как песочные часы

## Интерфейс BSD сокетов

Интерфейс сокетов — это де-факто стандарт взаимодействия прикладной программы с ядром ОС — точка входа в сеть для приложения. Он соединяет прикладной уровень стека TCP/IP, который реализуется в пользовательском пространстве, с нижним уровнями, которые, как правило, реализуются в ядре ОС.

Сокеты расчитаны на работу в клиент-серверной парадигме взаимодействия: активный клиент подключается к пассивному серверу, который способен одновременно обрабатывать множество клиентских соединений. Для идентификации сервера при сокетном соединении используется пара IP-адрес—порт. **Порт** — это уникальное в рамках одного хоста число, как правило, ограниченное в диапазоне 1-65535. Порты делятся на привилегированные (1-1024), которые выделяются для приложений с разрешения администратора системы, и все остальные — доступные любым приложениям без ограничений. Большинство стандартных прикладных протоколов имеют стандартные номера портов: 80 — HTTP, 25 — SMTP, 22 — SSH, 21 — FTP, 53 — DNS. Один порт может одновременно использовать только один процесс ОС.

Сокет — это файлоподобный объект, поддерживающий следующие операции:

- создание — в результате в программе появляется соответствующий файловый дескриптор
- подключение — выполняется по-разному для клиента и сервера
- отключение
- чтение/запись
- конфигурация

Основные системные вызовы для работы с сокетами:

- `socket` — создание сокета
- `connect` — инициация клиентского соединения
- `bind` — привязка сокета к порту
- `listen` — перевод сокета в пассивный режим прослушивания порта (актуально только для TCP соединений)
- `accept` — принятие соединения от клиента (который вызвал операцию `connect`) — это блокирующая операци, которая ждет поступления нового соединения
- `read/write` или же `send/recv` — запись/чтение данных в сокет
- `recvfrom/sendto` — аналогичные операции для UDP сокетов
- `setsockopt` — установка параметров сокета
- `close` — закрытие сокета

Поскольку сокеты — это, фактически, интерфейс для погружения на третий уровень TCP/IP-стека, сокеты не предоставляют механизмов для управления кодированием данных и сенсами работы приложений — они просто позволяют передать "сырой" поток байт.

## Общая схема взаимодействия через сокет

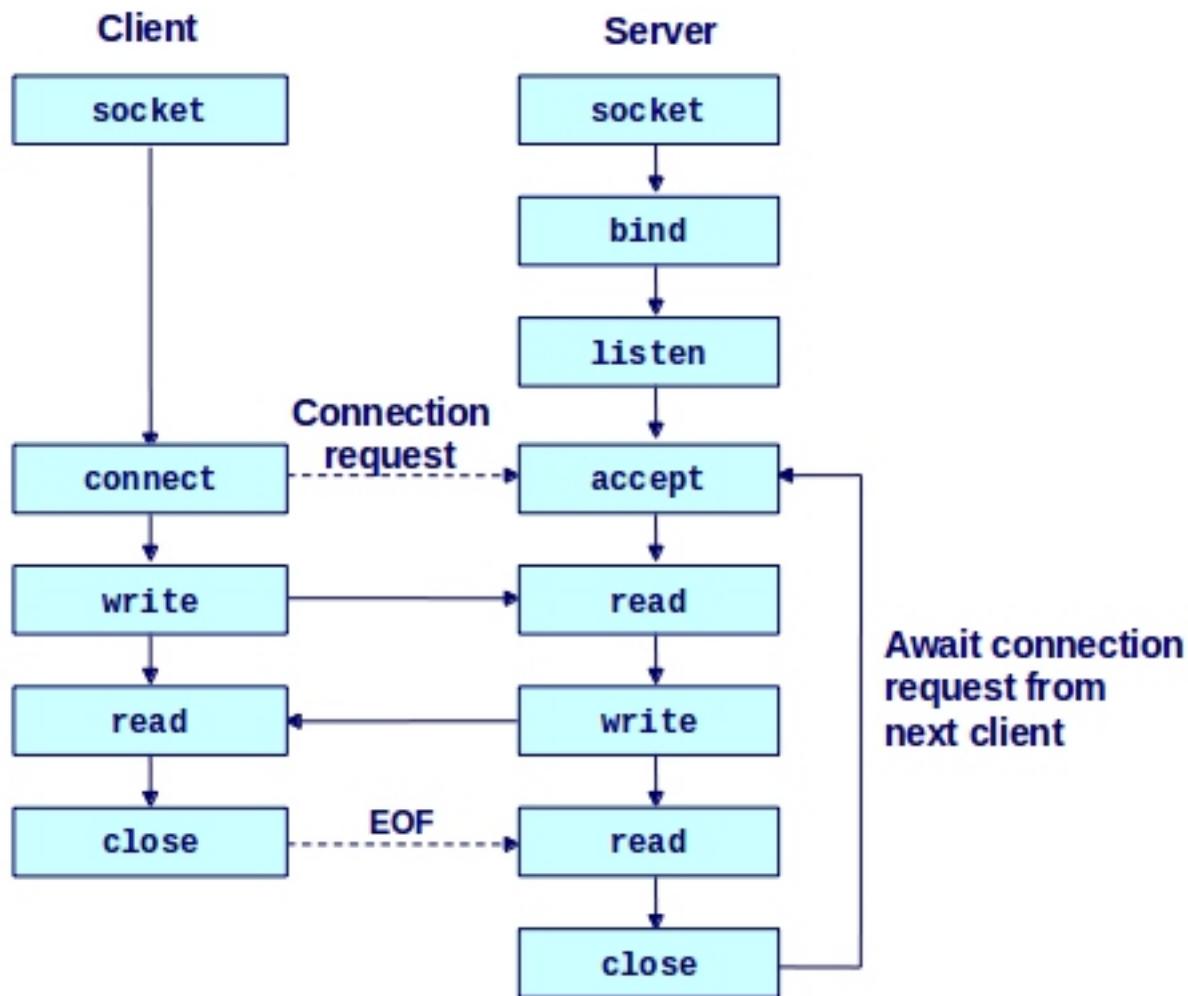


Рис. 9.2. Общая схема взаимодействия через сокет для TCP соединения

Как видно из схемы, на сервере для установления TCP соединения нужно выполнить 3 операции:

- bind захватывает порт, после чего другие процессы не смогут занять его для себя
- listen (для TCP соединения) переводит его в режим прослушивания, после чего клиенты могут инициировать подключения к нему
- однако, пока на сервере на выполнен accept, клиентские соединения будут ожидать в очереди (backlog) сокета, ограничения на которую могут быть заданы в вызове listen

Выполнение accept приводит к появлению еще одного объекта сокета, который отвечает текущему клиентскому соединению. При этом серверный сокет может

принимать новые соединения.

После выполнения accept сервер может реализовать несколько сценариев обслуживания клиента:

- эксклюзивный вариант — обслуживание происходит в том же потоке, который выполнил accept, поэтому другие клиенты ждут завершения соединения в очереди
- 1 поток на соединение — сразу же после выполнения accept создается новый поток, куда передается появившийся сокет, и дальнейшая коммуникация происходит в этом потоке, который закрывается по завершению соединения. Тем временем сервер может принимать новые соединения. Такая схема является наиболее распространенной. Ее основной недостаток — это большие накладные расходы на каждое соединение (отдельный поток ОС)
- неблокирующий ввод-вывод — при этом в одном потоке сервер принимает соединение, а в другом потоке работает т.н. цикл событий (event loop), в котором происходит асинхронная обработка всех принятых клиентских соединений

## Неблокирующий (асинхронный) ввод-вывод

Сокеты поддерживают как синхронный, так и асинхронный ввод-вывод. Асинхронный IO является критической функцией для создания эффективных сетевых серверов. Для поддержки асинхронного ввода вывода у сокетов (как и у других файловых дескрипторов) есть параметр `O_NONBLOCK`, который можно установить с помощью системного вызова `fcntl`. После перевода файлового дескриптора в неблокирующий режим, с сокетом можно работать с помощью системных вызовов `select` и `poll`, которые позволяют для группы файловых дескрипторов узнать, какие из них готовы к чтению/записи. Альтернативой `poll` являются специфичные для отдельных систем операции, которые реализованы более эффективно, но не являются портабельными: `epoll` в Linux, `kqueue` во FreeBSD и др.

## ZeroMQ (0MQ)

Развитием парадигмы сокетных соединений за рамки модели взаимодействия клиент-сервер является технология ZeroMQ, которая предоставляет усовершенствованный интерфейс сокетов с поддержкой большего количества протоколов взаимодействия, а также с поддержкой других схем работы:

- публикация-подписка (pub-sub)

- тяни-толкай (push-pull)
- дилер-маршрутизатор (dealer-router)
- эксклюзивная пара
- наконец, схема запрос-ответ (req-resp) — это классическая клиент-серверная схема соединения

См. [ZeroMQ - Super Sockets](#)

## RPC и сетевые архитектуры

Распределенная программа использует для взаимодействия определенный протокол, который также можно рассматривать как интерфейс вызова процедур удаленно (RPC — remote procedure call). Фактически, интерфейс RPC реализует прикладной уровень модели OSI, но для его поддержки также необходимо в той или иной мере реализовать протоколы уровня представления и, иногда, сеансового уровня. Уровень представления решает задачу передачи данных в рамках гетерогенной (т.е. состоящей из различных компонент) сети в "понятной" форме. Для этого нужно учитывать такие аспекты, как старшинство байт (endianness), кодировки для текстовых данных, представления композитных данных (коллекций, структур) и т.д. Еще одной задачей RPC-уровня часто является нахождение сервисов (service discovery).

Реализация RPC может быть основана на собственном (ad hoc) или же каком-то из стандартных протоколов прикладного уровня и представления. Например, реализация RPC по методологии REST использует стандартные протоколы HTTP в качестве транспортного и JSON/XML для представления (сериализации). XML/RPC або JSON/RPC — это ad hoc RPC, которые используют XML или JSON для представления данных. Протоколы ASN.1 и Thrift — это бинарные протоколы, которые определяют реализацию всех 3-х уровней.

В форматах сериализации существует 2 дилеммы: бинарные и текстовые форматы, а также статические (использующие схему) и динамические (без схемы, schema-less).

Распространенные форматы сериализации включают:

- JSON — текстовый динамический формат
- XML — тестовый формат с опциональной схемой
- Protocol Buffers — бинарный статический формат
- MessagePack — основанный на JSON бинарный формат

- Avro — основанный на JSON формат со схемой
- EDN (extensible data notation) — текстовый динамический формат

Сетевые приложения могут быть реализованы в виде разных сетевых архитектур. Ключевым параметром для каждой архитектуры является уровень централизации: от полностью централизованных — **клиент-сервер** — до полностью децентрализованных — **peer-to-peer/P2P**. Важными моделями между этими двумя крайностями является модель сервисно-ориентированной архитектуры (**SOA**), а также модель **клиент-очередь-клиент**.

## Литература

- [Tour of the Black Holes of Computing: Network Programming](#)
- [Beej's Guide to Network Programming](#)
- [Socket System Calls](#)
- [Мультиплексирование ввода-вывода](#)

# Безопасность

## Общие принципы безопасности

Информационная безопасность — это непрерывный процесс защиты информационных систем от угроз трех видов:

- несанкционированный доступ (НСД) и использование
- нарушение целостности, конфиденциальности, подлинности и других характеристик данных
- нарушение доступности и/или полноценного функционирования информационной системы

Подсистема безопасности не является выделенным компонентом ОС и ее практически невозможно добавить в систему пост-фактум, т.е. она должна быть встроена с самого начала.

Принципы создания безопасной системы:

- принцип безопасных настроек по умолчанию
- принцип валидации данных, поступающих от других участников системы
- принцип полной медации — всегда проверка текущих прав
- принцип наименьших привилегий — выдавать права, достаточные для выполнения только требуемых операций и не более того
- принцип разделения привилегий — по возможности, требовать согласия нескольких участников для выполнения операций
- принцип экономии на механизме — механизмы защиты должны быть максимально простыми из возможных и реализовываться на самом низком из возможных уровне
- принцип минимального общего между разными участниками системы
- принцип открытого дизайна — архитектура, реализация и используемые алгоритмы в системе должны быть известны, а в секрете могут держаться только ограниченные по объему авторизационные данные (ключи, пароли и тп.)
- принцип психологической приемлемости

Основные сервисы системы безопасности описываются аббревиатурой AAA:

- Аутентификация (Authentication) — установление "личности" стороны, с которой происходит взаимодействие

- Авторизация (Authorization) — проверка прав на выполнение каких-либо операций в системе
- Аудит (Accounting) — учет операций, связанных с системой безопасности (для возможности последующего расследования и установления причин проблемы)

Способы (факторы) аутентификации:

- по паролю
- по вопросу безопасности
- по одноразовому паролю
- по жетону (уникальным числом)
- с помощью сертификата ЭЦП

Аутентификация может быть как однофакторной, так и многофакторной.

## Механизмы работы системы безопасности

В системе безопасности рассматриваются 3 сущности: участники системы (субъекты, пользователи), ресурсы (например, файлы) и права доступа.

Единицей управления является **домен защиты** — это пара объект и права доступа. Домен может соответствовать одному субъекту или их группе.

Матрица контроля доступа — это теоретическая модель, которая описывает матрицу, которая приводит в соответствие все ресурсы системы со всеми ее субъектами. В ячейках этой матрицы находятся права доступа конкретного пользователя/роли/группы к конкретному ресурсу. Такая матрица позволяет описать все права доступа в системе, однако ее практическое применение не эффективно.

На практике используются 2 следующих подхода:

- списки контроля доступа (Access Control Lists, ACL), которые предполагают хранение и учет прав на уровне ресурсов, т.е., фактически, по столбцам этой матрицы
- мандатные системы (Capability или C-list), которые предполагают хранение прав доступа у субъектов системы, т.е. по строкам матрицам

В системе, основанной на ACL, для каждого ресурса определен список субъектов с их правами. Например, в ФС Unix у каждой директории и файла определены 3 типа субъектов: пользователь-владелец, группа-владелец и все остальные,— а также 3 типа прав: чтение, запись и выполнение. Другим

примером ACL является список правил межсетевого экрана, ресурсами в которых являются хосты/подсети и/или возможность обращения к определенным портам/использования определенных протоколов. В такой системе вместо прав доступа устанавливаются действия экрана при обращении: разрешить, запретить, ограничить и т.д. При этом, учитывая потенциальную неограниченность различных субъектов-участников сети, в такой системе в основном используются обобщенные субъекты: все хосты, все внешние хосты, все хосты из определенной подсети и т.д.

В системах на основе мандатов мандат выдается отдельно для каждого субъекта на каждое право доступа. Мандат, как правило, реализуется как числовой жетон (token), который выдается субъекту системой безопасности. Этот жетон может быть:

- просто уникальным числом, которое записывается в базу данных для кортежа пользователь, домен безопасности. Проблема такого способа в потенциально неограниченном количестве записей в системе с большим количеством ресурсов и/или субъектов. В такой системе автентифицированному субъекту достаточно предоставить жетон для того, чтобы идентифицировать ресурс, к которому он хочет получить доступ, и получить доступ
- криптографической величиной, полученной применением односторонней функции к паре домен безопасности и секретный ключ, известный только ядру системы,  $c = f(\text{domain}, \text{key})$ . В этом случае для проверки мандата субъект должен предоставить не только сам жетон, но и идентификатор ресурса и права доступа. Проверка будет производится повторным вычислением значения функции над теми же аргументами. Безопасность системы основана на невозможности получить то же значение жетона без знания секретного ключа. В этой системе затруднен отзыв отдельных мандатов, поскольку для отзыва мандата нужно либо изменить идентификатор ресурса, что повлияет на всех субъектов, имеющих к нему доступ, либо изменить ключ, который, как правило, уникален для пользователя, но отзыв ключа приведет к отзыву всех мандатов этого пользователя. Для решения этой проблемы используются т.н. непрямые объекты.

Хотя с точки зрения модели Матрицы прав доступа в обоих системах хранится одна и та же информация, эта модель описывает только статические характеристики системы и не описывает ее поведения в динамике.

При рассмотрении динамики работы системы безопасности на основе мандатов обладают следующими преимуществами перед списками контроля

доступа:

- отсутствие необходимости использования общего пространства имен ресурсов, известного всем субъектам системы
- возможность более гранулярного учета прав: в системе на основе списка контроля доступа администраторам системы необходимо исчерпывающее знание о всех субъектах системы — поскольку это психологически неприемлимо, субъекты обычно агрегируются более общими сущностями — **принципалами** безопасности

В то же время в мандатных системах более трудно решить следующие проблемы:

- ограничить передачу мандата от одного субъекта другому (такая возможность также может быть и полезным свойством системы)
- отзыв мандатов, особенно выборочный отзыв отдельных прав, а не всех мандатов для какого-то субъекта

Во многих реальных системах используется комбинация обоих подходов: например, в файловой системе Unix ACL используются для первичного контроля прав, а для проверки текущих прав используются мандат, который выдается после первичной авторизации, проверка которого намного эффективнее.

Системы на основе мандата часто используются для создания т.н. "песочниц", например для выполнения кода, полученных из недоверенных источников — поскольку в такой системе проще реализовать принцип "по-умолчанию без доступа".

## Реализация системы безопасности

Аппаратная платформа предоставляет такие базовые механизмы для поддержки системы безопасности:

- **кольца процессора** (CPU rings), которые используются в сегментной организации памяти
- привилегированные инструкции (в некоторых plataформах)
- **рандомизация адресного пространства программы, защита стека** и т.п.

Используя эти примитивы ОС выстраивает систему безопасности. Основа этой системы в большинстве ОС:

- это выполнение всех критических операций в ядре ОС и предоставление ограниченного интерфейса к ним через механизм системных вызовов
- поділ користувачів на адміністраторів (в Unix-системах: особливий користувач root) і звичайних користувачів

## Литература

- [Secure Systems Design Principles](#)
- [Defensive Programming](#)
- [CWE/SANS Top 25 Most Dangerous Software Errors](#)
- [Capability Myths Demolished](#)
- [Classic Buffer Overflow Explained](#)
- [Priviledge Escalation Bug in Linux](#)
- [How to Exploit an XSS](#)

# Unix

## История Unix

UNIX появился в 1973 (начал разрабатываться в 1969) в Bell Labs. Первая целевая платформа — миникомпьютеры DEC (PDP-7).

В Unix были созданы такие технологии, как: язык C, оператор pipe (|) для взаимодействия между процессами, интерфейс сокетов BSD и многие другие.

UNIX изначально был условно открытой системой, достаточно удобной для портирования на другие архитектуры. Поэтому довольно быстро появились разные ветви (варианты) Unix'ов. Первой такой веткой (fork'ом) стал Берклиевский дистрибутив (BSD) в 1977 году. В то же время, лицензия UNIX не давала возможности неограниченного изменения и модификации системы, с чем были связаны многие юридические конфликты. В конце концов, на данный момент сформировалось несколько закрытых коммерческих версий Unix'a, несколько открытых версий, а также ряд Unix-подобных систем, созданных с нуля (прежде всего, GNU/Linux).

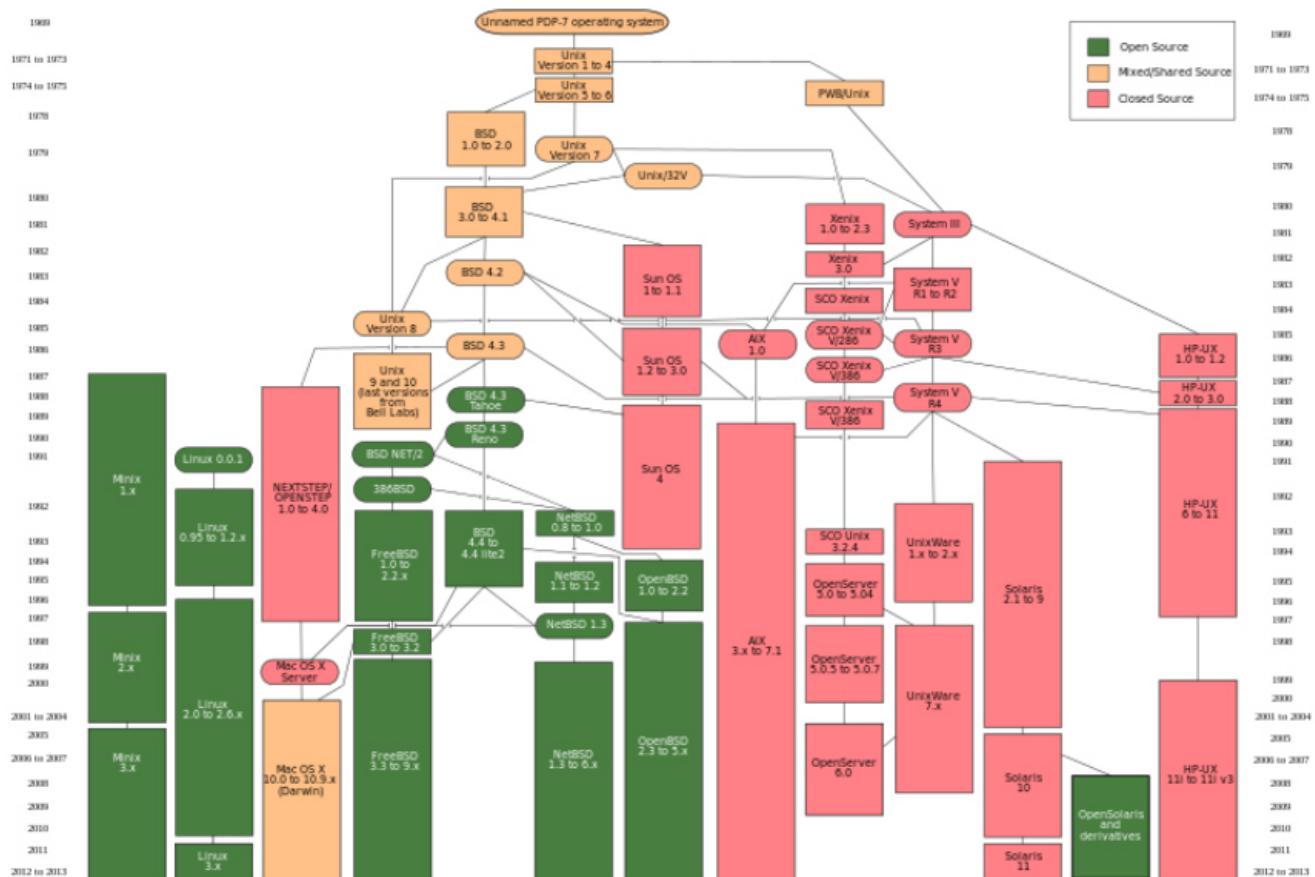


Рис. 11.1. Семейство потомков Unix и Unix-подобных ОС

# Основные вариации Unix'ов

## Коммерческие вариации

Коммерческие версии Unix ведут свою родословную от т.н. Unix System V. Практически все они прекратили свое развитие. Основные представители:

- SCO UnixWare (прекращен)
- Sun Solaris (прекращен)
- IBM AIX
- HP-UX

На данный момент развивается в качестве ОС для дата-центров только illumos — открытый наследник Sun Solaris, который был самой продвинутой и активно изменяющейся версией Unix'a.

Ключевые технологии illumos:

- ФС ZFS
- технология изоляции и создания т.н. песочниц Solaris Zones
- система интроспекции DTrace
- технология виртуализации Kernel Virtual Machine

## Варианты BSD

Берклевский дистрибутив развивался как альтернатива System V Unix и с момента выхода в свет версии 4.3 Tahoe стал полностью открытым и бесплатным. С дистрибутивами BSD связана одна из самых распространенных open-source лицензий — лицензия BSD. BSD-версии Unix продолжают активно развиваться.

Представители BSD семейства:

- FreeBSD
- OpenBSD
- NetBSD
- DragonflyBSD

На основе дистрибутива FreeBSD было создано ядро Darwin, которое используется в MacOS X.

## GNU/Linux

Linux — это Unix и не Unix — полностью новая система, созданная на основе принципов Unix, как их понимал Линус Торвальдс. Толчком для написания системы стали распространение учебного варианта Unix'a MINIX, созданного Таненбаумом, а также появление процессора Intel 386 с поддержкой плоской сегментной модели, что радикально упростило написание ядра ОС.

Linux обошел другие варианты Unix'a на волне open-source за счет использования наработок движения GNU (компилятор gcc, утилиты core-utils, редактор Emacs и др.) и лицензии GPL, которая требует открытия доработок системы на тех же условиях, что и оригинальной системы.

Linux — это ядро, на основании которого создается дистрибутив — набор системных утилит и других программ, необходимых для работы системы, таких как графическая оболочка, офисные приложения и т.п. Существуют десятки дистрибутивов GNU/Linux, среди которых наиболее распространены ветки Red Hat (Red Hat Enterprise Linux, Fedora, Suse, CentOS) и Debian (Debian, Ubuntu, Mint).

Специфическим дистрибутивом Linux является ОС Google Android.

## Plan 9

Plan 9 была академической попыткой в рамках лаборатории Bell Labs создать наследника Unix, в котором бы были исправлены его основные недостатки. Она оказалась классическим примером синдрома "второй системы", и не получила распространения.

Ключевые решения:

- всё – действительно файл
- отдельные пространства имен
- упразднение суперпользователя

## Ключевые решения Unix

- открытая система (man, POSIX, open source)
- файл-центричность и текст-центричность
  - "Small pieces, loosely joined"
  - развитые средства IPC
- иерархическая файловая система с примитивной моделью безопасности

- плоское API системных вызовов, основанное на С
  - дополнительные возможности спрятаны в ioctl, fnctl, и т.п.
- пользователь root
- модель запуска процессов fork/exec
- развитая система управления зависимостями

## Поддержка GUI в Unix

Исторически ядро Unix разрабатывалось без поддержки графического интерфейса, который на тот момент еще не был развит. По мере развития различных вариантов графической аппаратной части в Unix было создано решения для поддержки и работы с ними — протокол X11, разработанный рабочей группой в университете MIT.

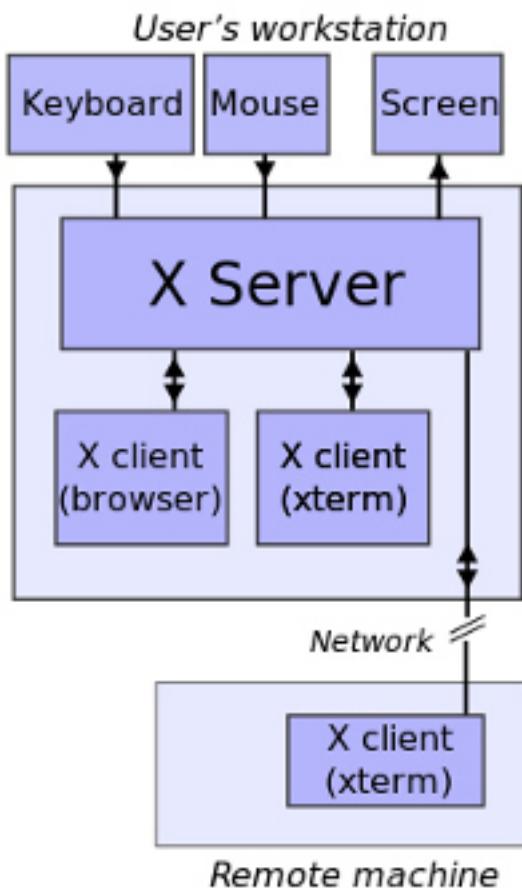


Рис. 11.2. Схема взаимодействия в рамках протокола X11

X11 протокол имеет различные реализации:

- xfree86
- X.org

- Wayland

На базе X11 протокола, как правило, строятся более высокоуровневые графические фреймворки (GUI toolkits). Самые распространенные фреймворки в Unix-среде:

- GTK
- Qt

Оконные менеджеры решают задачу управления интерфейсом приложений в рамках единой концепции (на данный момент принята концепция Рабочего стола или [WIMP](#)). Как правило, оконные менеджеры строятся на основе графических фреймворков. Например, самый распространенный менеджер Gnome использует GTK, KDE построен на основе Qt.

Другие оконные менеджеры:

- обычные: Xfce, Sawfish
- плиточные: ion3, XMonad

## Управление зависимостями

В условиях создания новых приложений с использованием созданной ранее базы программных библиотек, важной и сложной задачей, которую решает любой дистрибутив Unix (а также, на данный момент и среди всех языков программирования), является управление зависимостями. Для этого (почти) каждый дистрибутив Unix имеет специальную программу — пакетный менеджер, который отвечает за ведение базы существующих библиотек и их версий, зависимостей и совместимости между версиями, а также мест хранения исходного или объектного кода, из которых можно загрузить ту или иную версию.

Различают менеджеры зависимостей, которые работают на уровне исходного кода (с его последующей сборкой) и на уровне бинарных файлов.

Самые распространенные пакетные менеджеры:

- менеджеры на основе формата APT в Debian
- менеджеры на основе формата RPM в Red Hat
- BSD ports и система Portage Gentoo Linux

Также, большинство сред языков программирования предоставляют свой особый менеджер пакетов. Например:

- Maven в Java
- RubyGems в Ruby
- NPM в Node.js

## Принципы разработки под Unix

В книге "Искусство программирования под Unix" ([TAOUP](#)) Эрик Реймонд сформулировал следующие принципы хорошего тона, которые предпочтительно применяются при разработке как самих частей Unix систем, так и программ для Unix:

- Модульности (Modularity): создавать простые части, связываемые чистыми интерфейсами
- Ясности (Clarity): ясные решения лучше хитрых и умных
- Композиции (Composition): создавать программы, которые могут быть связанными с другими программами
- Разделения (Separation): разделять политику и механизм, интерфейс и реализацию
- Простоты (Simplicity): придумывать самое простое решение, добавлять сложность только по необходимости
- Бережливости (Parsimony): создавать большие программы, только когда продемонстрированно, что другие не справятся
- Прозрачности (Transparency): продумывать программы с тем, чтобы сделать интроспекцию и отладку возможной и максимально простой
- Прочности (Robustness): прочность — это дитя прозрачности и простоты
- Представлений (Representation): вкладывать знания в данные, чтобы логика программы была тупой и надежной
- Наименьшего удивления (Least Surprise): при проектировании интерфейсов всегда нужно выбирать наименее неожиданный вариант
- Тишины (Silence): когда у программы нет ничего удивительного, чтобы сообщить, она не должна сообщать ничего
- Починки (Repair): стараться починить все, что можно, но когда это не удается — падать, падать громко и как можно раньше
- Экономии (Economy): время программиста дорого стоит, его лучше сберечь и потратить машинное время
- Генерации (Generation): по возможности избегать кодирования, когда можно написать программу, которая будет писать другие программы

- Оптимизации (Optimization): прототипировать перед шлифовкой, добиться работы программы перед ее оптимизацией
- Разнообразия (Diversity): не доверять никаким утверждениям о единственном верном пути
- Расширяемости (Extensibility): проектировать на будущее — оно наступит раньше, чем мы думаем

## Критика Unix

Наряду с фанатами Unix существуют и Unix-ненависники, которые даже написали собственную книгу — [Руководство Unix-ненависника](#).

Вот некоторые типичные претензии к Unix системам:

- Слабая поддержка GUI
- Слабая поддержка сценариев работы обычного пользователя (т.н. Desktop сценарии)
- Недостаточное следование модели “всё – файл”
- Примитивная модель безопасности: простой ACL, пользователь root
- Примитивная файловая система
- Дополнительные возможности спрятаны в ioctl, fnctl, ...
- Устаревший системный язык (C)

## Литература

- [The Art of Unix Programming](#)
- [The UNIX-HATERS Handbook](#)
- [The Daemon, the GNU & the Penguin](#)

# Windows

## История Windows

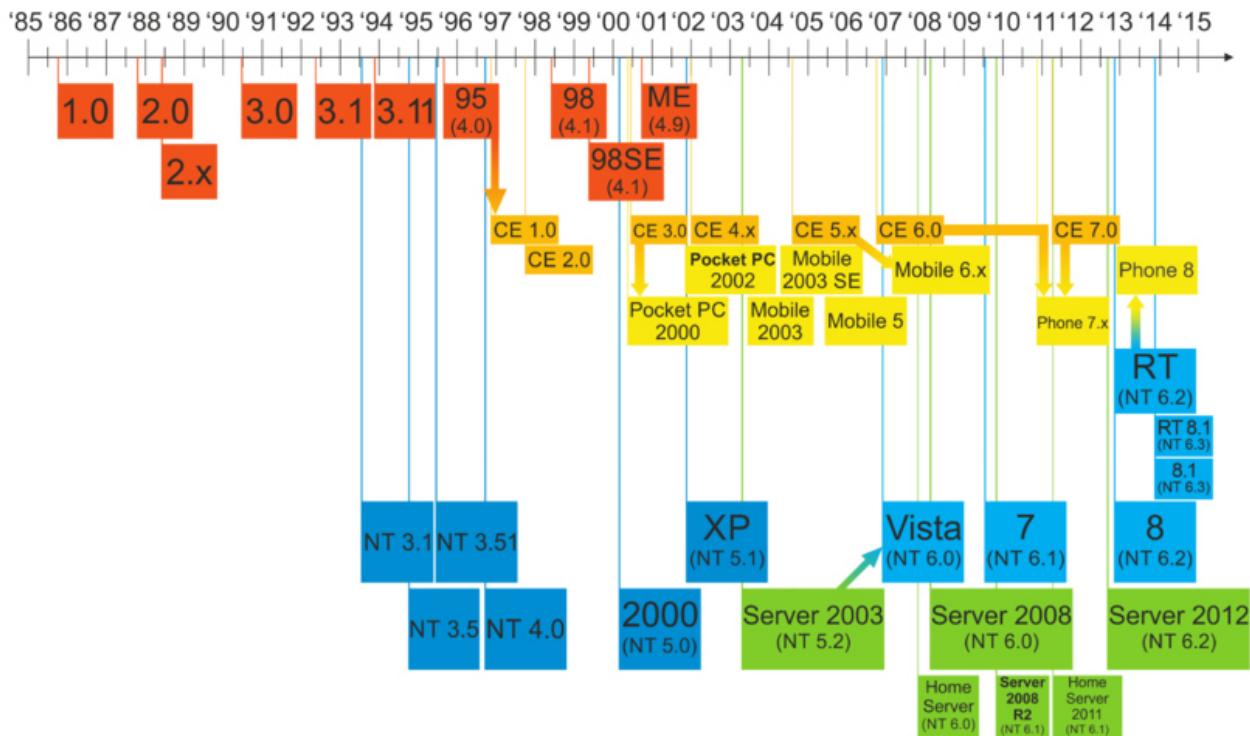


Рис. 12.1. Хронология развития Windows

Windows — это семейство ОС, которое развивалось постепенно, начиная с переделки ОС CP/M в ОС MS-DOS. MS-DOS, как и CP/M, был примитивной однопользовательской ОС для только появлявшихся персональных компьютеров. Он был создан в рамках существующих на тот момент аппаратных ограничений этих устройств (16-битная архитектура, малые доступные ресурсы, такие как мощность процессора, объемы памяти и постоянных хранилищ). Изначально эта система должна была загружаться с дискет емкостью порядка 750 Кб, работать в текстовом графическом режиме и управлять памятью до 1 МБ. Система Windows изначально была графической оболочкой поверх MS-DOS.

Вместе с быстрым развитием возможностей ПК начала быстро развиваться и Windows, вскоре упервшись в ограничения своей архитектуры. Windows 95/98 стала 32-разрядной версией системы, которая еще не использовала такие стандартные технологии поддержки многопользовательской работы, как разделение на режим ядра и пользователя, вытесняющую многозадачность, ФС с разделением прав доступа и т.п.

Реализацией современных концепций ОС стало ядро Windows NT, которое послужило основой систем Windows 2000/XP/7/8 и Windows Server.

## Windows NT

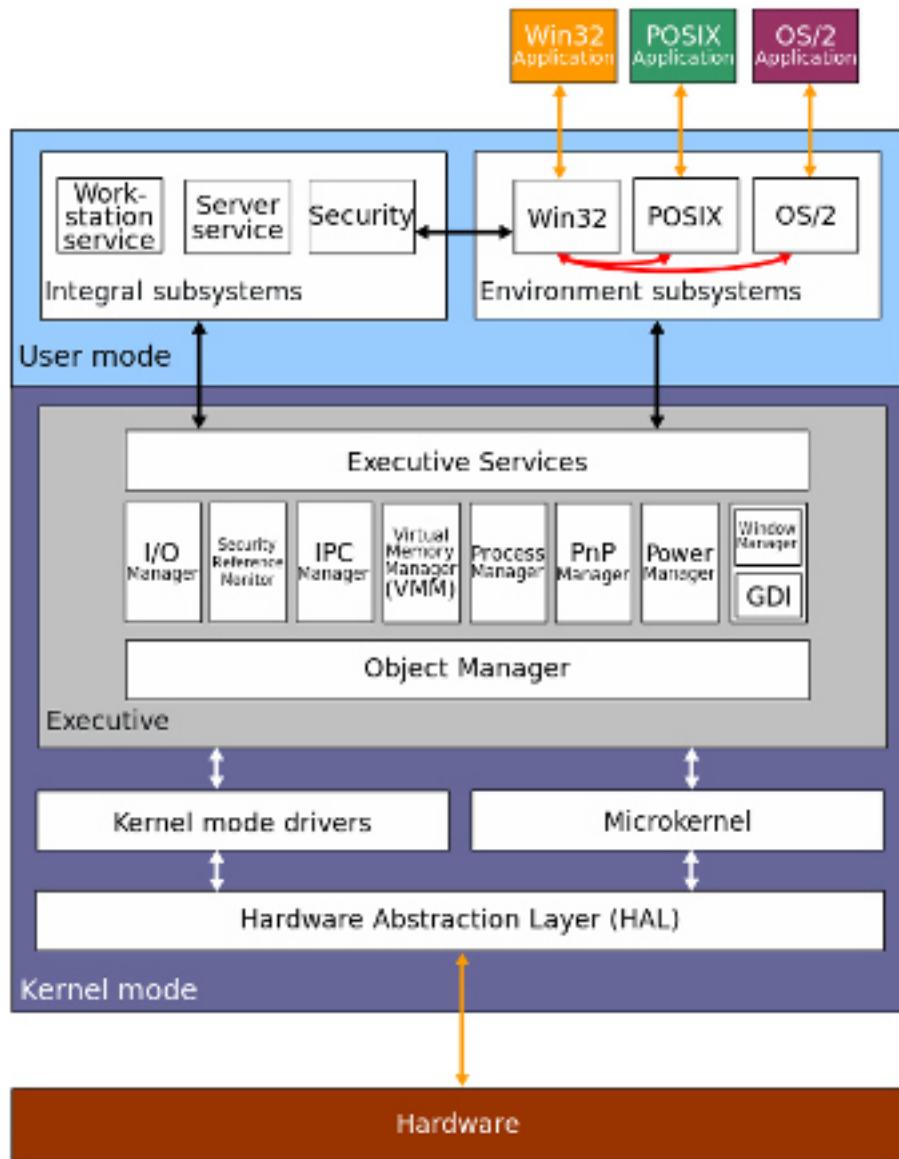


Рис. 12.2. Архитектура Windows NT

Windows NT — это название современного ядра Windows.

Его основные характеристики:

- разделение на режим ядра и пользователя
- гибридное ядро, включающее микроядро, уровень абстракции устройств и драйверов (HAL) и сервисы ОС, работающие в ядерном режиме (Exe-

tive), а также сервисы ОС, работающие в пользовательском режиме:  
окружение (Environment) и интеграции (Integral)

- поддержка вытесняющей многозадачности
- поддержка SMT/SMP
- ввод-вывод на основе пакетного и асинхронного режимов

Микроядро выполняет следующие функции:

- синхронизация
- планирование выполнения нитей и обработки прерываний
- перехват исключений
- инициализацию драйверов при запуске системы

Исполнительные сервисы ОС (Executive) включают:

- ввод-вывод (в т.ч. взаимодействие с графическими устройствами через интерфейс GDI)
- управление устройствами (в т.ч. питанием, поддержка Plug-n-play устройств)
- управление объектами ядерного режима
- управление процессами и межпроцессорным взаимодействием
- управление памятью и кешами
- управление конфигурацией (через системный реестр)
- безопасность

Сервисы окружения ядра WinNT поддерживают 3 режима работы:

- Win32-совместимый (с поддержкой MSDOS и Win16 приложений), включающий также оконный менеджер - сервис csrss.exe
- Posix-совместимый
- OS/2-совместимый

## Ключевые решения Windows

- привязка к архитектуре x86 (Windows не поддерживает других архитектур, помимо ARM)
- GUI-центричность
- Windows API
- Управление конфигурацией через реестр

- Большое внимание обратной совместимости
- Системный язык C++, затем C#

## Критика Windows

- закрытая система
- проблемы с композицией программ
- проблемы с безопасностью
- непригодность для многих сценариев работы (прежде всего, как высокопроизводительной серверной ОС)

## Литература

- [Mark Russinovich & David Solomon - Windows Internals](#)
- [The Infamous Windows “Hello World” Program](#)
- [How Microsoft Lost the API War](#)