1 Вычислительная система

1. Прикладное программное обеспечение;
2. Системное программное обеспечение. Граница с прикладным ПО довольно размыта;
3. Системы программирования (компиляторы, форматы представления данных...
   Программы для написания программ и библиотеки);
4. Абстрактная машина (абстрактный вычислитель), вычислительные виртуальные ресурсы (например, файл, программа);
5. Управление физическими ресурсами;
6. Аппаратное обеспечение.

Ядро, включая драйверы, лежит на уровнях 3-5, но 3-ий уровень захвачен не полностью. Драйверы предоставляют абстракцию, самостоятельно оперируя прерываниями и адресами.

Операционная система лежит на уровнях 2-5. Debian kFreeBSD является примером ОС, в которой на ядре FreeBSD предоставляется тот же системный интерфейс, что и в системах семейства Linux.

Механизм системных вызовов

Есть всякие регистры, прерывания, assembler, но как это относится к программе на C? Если не использовать ничего дополнительного, то придётся под каждую конкретную машину переписывать код из-за разницы в этих интерфейсах.

Для этого используют библиотеки. Есть библиотека libc, которая принудительно прилинковывается к каждой программе. Эта библиотека предоставляет интерфейс для взаимодействия с «внутренностями» операционной системы.

POSIX — интерфейс системных вызовов, чья реализация содержится в libc.

1.1 Уровень аппаратного обеспечения

- Материнская плата
  - разъём шины питания;
  - разъёмы оперативной памяти;
  - разъём(ы) под процессор(ы);
  - разъёмы расширений PCI-E (Peripheral component interconnect - Express). Раньше чаще использовались другие разъёмы: AGP (Accelerated Graphics Port), PCI, ISA (Industry Standard Architecture). Шина ISA была аналоговой, т.е. сигнал не дискретизировался. Остальные же шины уже были цифровые.
  - разъёмы для «медленных устройств». Это разъёмы, к которым подключаются устройства, связанные с человеком. Сейчас большинство «медленных устройств» переехали на
  - USB (Universal Serial Bus).
  - SATA (Serial ATA) — последовательный интерфейс обмена данными с накопителями, SCSI «скази» (Small Computers System Interface) — интерфейс для общения компьютера с периферийными устройствами.
  - BIOS (Basic Input-Output System). Эта микросхема подключена ко всем шинам.
  - батарейка, которая позволяет сохранять астрономическое время, например, для запуска компьютера в определённое время.

На материнской плате есть набор шин:

- Процессор - RAM («северный мост»).
- RAM - контроллер периферии.
- Процессор - контроллер периферии («Южный мост», обычно PCI-E). Остальные шины являются абонентами этой шины.

Эти три шины называются быстрыми шинами, так как контроллер PCI-E. В системе Linux при помощи команд `lspci` и `lsusb` можно посмотреть набор устройств, подключённых к соответствующей шине.

Компьютер суть есть иерархия шин, к которым подключаются все устройства.

Архитектура фон Неймана

Компьютер фон Неймана — исполнитель некоторой программы, состоящей из команд.

- Центральный процессор
  - АЛУ (Арифметико-логическое устройство)
  - УУ (устройство управления)
- Оперативная память, подключённая к процессору;
- Внешние устройства, подключённые к процессору.

В компьютере фон Неймана есть понятие **такт** — время, за которое выполняется одна команда. Команды выбираются из оперативной памяти в некотором порядке.

Принципы фон Неймана:

1. Принцип хранимой программы (программа хранится в оперативной памяти).
2. Принцип адресности. Оперативная память разбита на набор ячеек, одна ячейка называется **словом**.
3. Принцип двоичного кодирования.
4. Принцип однородности. Команды и данные хранятся в одной и той же памяти и внешне в памяти неразличимы.

Эта архитектура небезопасна, так как программа может затереть абсолютно всё, включая коды всех программ.

Сейчас для защиты памяти используют механизм виртуальной памяти и тегированную память. Тегированная память отличается от обычной тем, что слово представляет из себя не только адрес, но и некоторой тег, который предоставляет некоторую служебную информацию.

Гарвардская архитектура

Эта архитектура отличается от фон Неймановской тем, что инструкции и память физически разделены на разные устройства хранения и каналы к этим устройствам.
Она не используется, так как никак нельзя загрузить новую программу, но зато она безопасна, в отличие от фон Неймановской.

1.2 Память

1. Оперативная память
2. Внешнее запоминающее устройство
   - HDD
     - Floppy
   - SSD
   - Оптика
3. Кэш процессора
4. Регистровая память

1.2.1 Время жизни информации

Тактовая частота — частота тактов.
Такт — минимальное время, необходимое для того, чтобы фиксировать состояние процессора, чтобы получить гарантированное значение сигнала на всех входах и выходах.
Генератор тактовой частоты — может дискретно изменять тактовую частоту.
Время жизни в регистровой памяти — время, на протяжении которого процессор включен. Регистровая память работает со скоростью процессора.

Два варианта:

- Кэш находится в процессоре. Работает медленнее регистровой памяти, время жизни — время, на протяжении которого процессор включен.
- Кэш — некоторая микросхема, тесно связанная с процессором. Время жизни — время, на протяжении которого процессор включен.

Выключая машину, оперативная память может продолжать существовать, при условии, что она сама потребляет электричество.

Характеристики ОП:

- Частота шины
- taccess — время чтения
- twrite — время записи (время до освобождения шины)
- trecycle — время повторного обращения (время, после которого можно обратиться к той же ячейке, если до этого что-то записали в эту ячейку или извлекали из нее информацию).
- trefresh — время, через которое нужно повторно записывать в ячейку значение. (это необходимо, так как происходит утечка заряда из одной ячейки в другую, то есть 0 может стать 1 или 1 станет 0, таким образом, нужно обновить значение заряда).

2 Процессор

Instruction pointer — регистр, задающий номер ячейки в ОП для извлечения оттуда слова, которое содержит некоторую команду.

Стадии работы процессора:

1. Чтение инструкции (это делает устройство управления)
2. Декодирование (Decoder)
3. Выполнение инструкции АЛУ (при этом могут быть использованы дополнительные данные из ОП, а также регистры)
4. Сохранение результатов (в ОП и в регистрах)

2.1 Системы команд

Команда:

- Код операции
- Типы аргументов
- Аргументы (для каждого выделено обычно либо 32 бит, либо 64 бит памяти, в зависимости от архитектуры процессора; иногда можно оптимизировать хранимые данные: под аргумент отводят меньшее количество бит)
  - Число (интерпретируется либо как константа, либо как адрес в памяти; будем пока считать, что адрес задан номером ячейки)
  - Регистр

В *microcode* задается реализация макрокоманд.

Системы команд:

1. CISC(Complex Instruction Set Computing) — команды могут иметь переменный размер.
2. RISC(Reduced Instruction Set Computing) — пытается уменьшить размер команды. Размер команды почти всегда фиксирован. Команда *prefetch* позволяет извлечь из памяти заданное количество команд и выполнять их потом параллельно.

Классификация команд по количеству аргументов:

- Одноадресные. Операции происходят над единственным аргументом, результат записывается в регистр.
- Двухадресные. Один из операндов используется для записи результата(обычно в первый).
- Трехадресные. Результат записывается в третий операнд(вычисления происходят над первыми двумя).
- Безадресные. Извлекают аргументы из памяти и кладут обратно в память. (Стековая машина)

2.1.1 Типы даннах в архитектуре

- Адреса
- Целочисленные типы. В скобках указаны суффиксы, которые дописываются к командам, чтобы понимать, какой длины аргумент используется.
  - 1 byte — байт (b), `char` в C
  - 2 bytes — слово (w), `short` в C
  - 4 bytes — удвоенное слово (d), `int` в C
  - 8 bytes — учетверённое слово (q), `long int` в C
  - 16 bytes — 128 bit (o), `long long int` в C

  Для хранения отрицательных чисел используется так называемый дополнительный код. Для получения числа $-A$ мы вычитаем из него 1 и инвертируем все биты.
- Типы с плаваюзей точкой.
  - `float` (d)
  - `double` (q)
  - `long double` (o)

  Для представления этих типов используется запись вида <знаковый бит>:<знаковый бит экспоненты>:<7/10 бит экспоненты>:<мантисса>. Мантисса при этом хранится не в дополнительном а в прямом коде. То есть у чисел с плавающей точкой есть +0 и -0. Всякие случаи вида хранения нуля, $\pm\infty$ описаны в стандарте IEEE 754. 0 хранится всеми нулями. `NaN` хранится как все единицы. ∞ хранится как знак бесконечности в знаке, все единицы в порядке и все нули в мантиссе.
- Строки. Непрерывныя последовательность байт, которую можно целиком перенести в другую область памяти.
- Биты. Их можно хранить двумя разными способами. Можно считать, что младшие биты хранятся на младших адресах, а можно наоборот. То есть число можно читать "слева направо"и "справо налево"в порядке увеличения адресов.
  - Little-endian — старшие биты хранятся в байтах со старшими адресами
  - Big-endian — старшие биты хранятся в байтах с младшими адресами

2.1.2 Виды регистров

- РОН (регистры общего назначения) — могут использоваться командами
- Служебные регистры — на них лежит управление работой процессора
  - IP — instruction pointer. Read-only регистр. Может называться IP (16bit), EIP (Extended IP, 32bit), RIP (Rich IP, 64bit)
  - Регистры флагов (FLAGS, EFLAGS, RFLAGS) — некоторые флаги после выполнения команды. Например, после переполнения при сложении будет выставлен флаг переполнения.
    - DF (Direction Flag), который задаёт направление операции, операции перемещения строки или побитового сдвига.
    - ZF (Zero Flag), обычно используется в счётчиках цикла, выставляется, если последняя операция дала 0. Для этого в ассемблере используется JE, которая проверяет ZF и, если он установлен в 1, то переходит по указанной метке

  * Бит чётности
  * Бит переноса
  * И другие

2.1.3 Виды команд

- Побитовые:
  - XOR
  - AND
  - OR
  - ROL (специфична для Intel) — циклический сдвиг
  - SHL, SHR — битовый сдвиг влево и вправо соответственно.
- Арифметические:
  - MUL — умножение
  - DIV — деление
  - ADD — сложение
  - SUB — вычитание
  - MOD — остаток по модулю

  Для всех этих операций есть I-версия
- Перемещение память-регистр:
  - MOV
  - CMP операция сравнения, выставляет флаги
  - Команды расширения: CBW (b $\rightarrow$ w) и другие. Добавление суффикса E к команде означает сохранение знака. Сокращение же производится автоматически при перемещении из большего типа данных в меньший банальным «отрубанием» всего лишнего.
- Операции переходов:
  - JMP — переход на указанный адрес. Если у нас память слишком большая, а разрядность, например, 32 бита, то для длинного прыжка нужно использовать farjmp, то есть изменить область view, после чего сделать JMP.
  - JE — условный переход при флаге равенства нуля
  - JG — условный переход при флаге больше нуля
  - JL — условный переход при флаге меньше нуля
  - JGE — условный переход при флаге больше или равно нуля
  - JLE — условный переход при флаге меньше или равно нуля
  - Частным случаем перехода является функция CALL, которая вызывает функцию при помощи стека. Им управляют 2 регистра общего назначения SP и BP. Регистр SP (Stack Pointer) указывает на вершину стека. Перед вызовом процедуры аргументы нужно положить в стек в каком-то порядке, перемещаем SP на вершину. BP указывает на начало «стопки аргументов». После выделяется память в стеке для выполнения процедуры. ENTER перемещает SP на столько, на сколько нужно для выделения памяти для процедуры. LEAVE возвращает SP на место.

    Функция CALL проделывает все вышеуказанные действия и переходит на начало функции. RET же, производит обратное действие

- Работа со стеком:
  - PUSH
  - POP
  - PUSHA — добавить все регистры общего назначения в стек
  - POPA — убрать все регистры общего назначения из стека

2.1.4 Выравнивание памяти

При обращении к памяти разным количеством байт могут получаться разные эффекты. При обращении к одному байту, адрес к которому мы обращаемся, не важен. Если же мы обращаемся двумя и большим числом байт, то при несовпадении делимости номера адреса, можно или получить или деградацию производительности или ошибку обращения к памяти.

В *C* при объявлении

```
struct s
{
    char a;
    int b;
}
```

переменные будут размещены в том порядке, в котором они указаны в структуре. Если размещать четырёхбайтовый `int` сразу после однобайтового `char`'а, то при каждом обращении к `b` нужно будет делать два запроса к памяти. Вероятно, что итоговый размер будет 6 байт. Но можно заставить компилятор (используя определённые опции)

3 Виртуальная память

1. Сегментная организация. При такой организации вся память разбита на некоторые сегменты, у которых хранятся адреса. Адрес делится на **базу сегмента** и **смещение внутри сегмента**. Таким образом, перемещения между сегментами делятся на два типа:

   - «короткие» переходы — в адресе изменяется только 32-битное смещение внутри сегмента.
   - «длинные» переходы — меняется также и база сегмента.

   У этой организации есть применение в защите памяти. Если мы попытаемся выйти за границу сегмента, то случится аппаратная ошибка, которая впоследствие обработана операционной системой, вместо порчи каки-либо данных. Так, к защите памяти относятся

   (a) Размер сегмента и база — предотвращают выход за границу используемой памяти

   (b) Тип сегмента
   - системный
   - сегмент кода — можно обращаться только к инструкциям, нельзя обращаться как к данным
   - сегмент данных (возможно, read-only)
   - сегмент стека (в современных — частный случай сегмента данных). Обычно не подразумевает запись/чтение, используются для стековых операций

   (c) Уровень привелегий. Процессор может находится в одном из уровней привелегий (хранится в регистре флагов). Привилегии как-то ограничивают набор доступных инструкций и адресов. Программа пользователя обычно работает на самом непривелегированном уровне

(user-mode). Самым привелегированным является kernel-mode, он работает на 0-ом уровне привилегий.

В архитектуре Intel вся информация о сегментах хранится в таблице сегментов, доступной по регистру `gdtr`.

Глобальные переменные описываются в сегменте данных. Константы описываются в read-only сегменте данных. В архитектуре Intel они хранятся в сегменте кода.

Язык ASSEMBLER

Этот язык — способ записи команд в мнемоническом виде. После компиляции этого кода получается файл `.o` с объектным кодом.

Для работы с `.o` есть `binutils`:

- `nm` — выводит все символы из объектного файла
- `ld` — преобразует набор объектных файлов в исполняемый файл с расширением `.ELF` (Linux), `MZ` (Windows), `.COFF`. Отличия заключаются в способе хранения программы
- `as` — принимает файлы `.s` (`.asm`) d файлы `.o`.
- `objdump` — по объектному файлу генерирует код assembler, который, в свою очередь, при компиляции даст исходный объектный файл
- `addr2line` или `addrtoline` — говорит, к какой строчке объектного файла относится данный адрес

В дальнейшем речь будет идти о `gas` (GNU Assembler).
В `gas` всё, что начитается с точки, является так называемой директивой.

1. Секции
   - `.text` — инструкции
   - `.data` — переменные
   - `.rodata` — read-only data
   - `.bss` — неинициализированные данные, принудительно заполнено нулями.
2. Описания переменных
   - `.byte, ...`. Например, `.byte 'a','b'` задаёт двухбайтовую переменную в которой первый байт соответствует 'a', а второй — 'b'.
   - `.word` — 2 байта
   - `.long` (`.int`) — 4 байта
   - `.quad` — восемь байт
   - `.double` — 8 байт, число с плавающей точкой
   - `.ascii` — строка без ноля (`.ascii "abc"`)
   - `.string` — нуль-терминированная строка
   - Управление выравниванием. Директива `.align num` гарантирует, что следующий адрес будет кратен num
   - `.space num` — захватывает num байт. `lcom`, `com` делают то же самое, `lcom` позволяет задать имя.
3. Команды

- При написании чего-либо (адреса или названия переменной) без символа `$` интерпретируется как адрес, если добавить `$`, будет вычислено число. Регистры используются через `%`, например, `%r15`

  `movg arr, %rdi;`

  Записать значение по адресу `arr` в регистр `rdi`
- Косвенная адресация. В gas задаётся круглыми скобками. Синтаксис `-4(%rdi)` означает интерпретацию содержимого регистра как адреса, применение к этому адресу смещения и получения значения.

4. Метки

   - `label: ...` — метка, на которую линковщик не обратит внимания.

3.1 Соглашение о передаче параметров

Пусть есть абстрактный язык программирования ProgL. Как передавать параметры функции, написанной на этом языке?

1. Можно класть параметры в стек. А в каком порядке? Пусть наша функция имеет вид `func(a, b, c, d)`.

   (a) порядок push для аргументов.
   - Можно класть аргументы в стек в том порядке, в котором они идут в вызове функции (снизу вверх).
   - Можно наоборот, сверху вниз, в обратном порядке поступления аргументов. В этом случае на вершине стека будет лежать первый аргумент (как в C)

   (b) порядок очищения стека.
   - вызывающий очищает стек (как в C)
   - вызываемый очищает стек

2. Для передачи параметров можно использовать *регистры общего назначения*. В архитектуре Intel таких регистров 16. В языке C можно передавать `char, short, int, long, long long`. С типом `long long` возникает проблема: он не помещается в один регистр. Для этого используются $\begin{bmatrix}\texttt{rax}\\ \texttt{rbx}\end{bmatrix}$. Таким образом, у нас становится на два регистра меньше. Всё, что не поместилось в регистры, помещается на стек, как в п.1.

3. Возвращаемое значение

   - на вершину стека
   - на регистры (как в C)

3.2 Позиционно зависимый/независимый код

Когда может быть нужен позиционно независимый код (который может быть размещён по любому адресу в виртуальной памяти и он будет исполняться нормально)? **Почти всегда**.

А как добиться позиционной независимости?

1. Избегать прямой адресации.

   В архитектуре amd64 есть возможность чтения регистра `rip`, который содержит адрес следующей выполняемой инструкции. На его основе построен метод относительной адресации RIP-relative.

2. Также, в случае подключения библиотеки в процессе выполнения программы, вызов библиотечной функции должен осуществляться не по обычному адресу, а по адресу перехода из таблицы символов. Если мы знаем, что функция будет лежать третьей в таблице символов, то код будет выглядеть так:

   ```
   mov T, %rdi
   call 3(%rdi)
   ```

   В случае отстутствия нормальной функции, будет осуществлён переход на функцию-заглушку или произведена динамическая линковка, которая заключается в проходе по таблице символов и поиске нужных функций в файловой системе в библиотеках, указанных при компиляции с ключом `-l`.

А когда может пригодиться позиционно **зависимый** код?

1. Загрузчик ОС
2. Сама операционная система, а именно, ядро в привилегированном режиме. В частности, для драйверов устройств

3.3 Таблица сегментов

В таблице сегментов описание сегментов содержит

1. бит представленности (1, если этот сегмент можно использовать)
2. база — начало сегмента в оперативной памяти
3. размер сегмента в у.е.
4. размер у.е., а точнее, номер этого размера Не может быть произвольным, есть заранее определённый набор возможных значений.
5. тип
6. права доступа
7. уровень привилегий

3.4 Страничная организация памяти

Страничная организация памяти может существовать независимо от сегментной.

Виртуальная память делится на некоторые диапазоны страниц. Минимальный размер каждого диапазона — одна страница.

Информация о страницах хранится в таблице страниц.

Каждый адрес разбивается на части:

1. Информация о записи в каталоге страниц 0 уровня
2. Информация о записи в каталоге страниц 1 уровня
3. ...
4. номер страницы
5. смещение внутри страницы

Для получения значения в адресе производится просмотр соответствующего диапазона в каталоге 0-го уровня. Если диапазон используется, то происходит переход в каталог страниц 1-го уровня и такой-же опрос диапазона адресов, но диапазон уже меньше. Если диапазон оказался неиспользуемым, то создаётся ошибка page fault.

Такой спуск продолжается до тех пор, пока не будет найдена нужная страница или не произойдёт page fault.

Для ускорения этого поиска используется TLB-кэш. В нём хранится старшая часть виртуального адреса и соответствующий ей адрес в оперативной памяти. Поиск в первую очередь производится по этому кэшу и только потом — по иерархии каталогов.

В каждой записи каталога/таблицы страниц хранится следующая информация:

1. бит представленности
2. «бит грязности» — были ли операции записи. В дальнейшем используется для своппинга
3. бит доступа
4. права
5. уровень привилегий
6. (если каталог) размер страницы
7. адрес в оперативной памяти

Чем же плоха такая структура? Она плоха тем, что при одном обращении к адресу, происходит негарантированное количество реальных обращений к каталогам страниц. Также тратится много оперативной памяти на хранение самой структуры каталогов

Запрет сброса TLB

Некоторые страницы можно пометить **несбрасываемыми**

3.5 Сегментно-страничная организация

При такой организации виртуальный адрес в первую очередь проверяется **блоком сегментации**, после чего попадает в блок «страничности», из которого идёт запрос к оперативной памяти. Каждый блок может по своему усмотрению изменить формат адреса.

4 Работа с периферией

- Порты ввода-вывода. Есть специальные команды:

  ```
  in  <port>, <data>
  out <data>, <port>
  ```

- Отображение адреса. Некоторая часть пространства виртуальных адресов выдаётся под внешние устройства.

  Например, в случае с видеокартой, при записи в виртуальный адрес, на самом деле происходит обращение к собственной памяти видеокарты посредством соответствующей шины. В реальности размер такого выделенного сегмента фиксирован, и при передаче большого массива данных приходится передавать данные по частям.

- Асинхронный режим. Реализуется при помощи DMA (Direct Memory Access). При асинхронном режиме есть отображение с виртуальной памяти на память внешнего устройства, а также внешнее устройство связано с оперативной памятью. Механизм общения выглядит так:

1. Модификация таблицы страниц по защите диапазона адресов оперативной памяти.
2. Формирование команды внешнему устройству. В отображаемые адреса кладём следующую информацию:
   - Номер прерывания
   - Код операции
   - Диапазон адресов физической памяти
3. Возврат процессора к нормальному режиму
4. Прерывание. Чтение данных из диапазона отображаемых адресов (информация о том, как завершилась операция). Выполняется обработчиком прерывания.
5. Переписывание данных из «DMA»-диапазона в структуры ядра. При этом обработчик прерывания должен понимать, куда нужно положить эти данные (например, данные с жёсткого диска и из сети нужно класть в соответствующие буферы).
6. Изменение таблиц страниц с целью снятия блокировки с виртуального адреса. Этот шаг необязательно выполняется, например при изначальной инициализации «DMA» диапазона.

5 Прерывания и их обработка

Все прерывания пронумерованы

- Регистр прерываний. При прерывании регистр выставляется в соответствующее значение.
- Регистр маски прерывания. При выставлении нужных битов в 1 можно замаскировать прерывание и процессор не будет обращать на него внимания
- Вектор прерываний. Изначально предоставляется BIOS, после чего переписывается ОС и используется для своих нужд. Ссылка на физическую память этого вектора лежит в регистре `idtr`. Это некоторая таблица, содержащая адреса входов в обработчики прерываний.

У прерываний есть приоритет. Обычно, чем больше номер прерывания, тем ниже у него приоритет.

APIC

На материнской плате находится контроллер прерываний. На каждом процессоре есть LPIC — локальный контроллер прерываний. При поступлении сигнала с внешнего устройства на контроллер прерываний прерывание по системе round-robin передаётся процессорам. После был создана система MSI, затем MSIX.

5.1 Передача прерывания на процессор

После передачи прерывания процессору (в LPIC) и понимания, что данное прерывание не замаскировано, происходит следующее:

1. Аппаратная обработка прерывания:
   (a) Сохраняем в стек `RIP` (место выполнения), `FLAGS`, регистр маски, сегментные регистры;
   (b) Регистр маски выставляется в 1 (все возможные прерывания маскируются);
   (c) Вызываем обработчик прерывания с нулевым уровнем привилегий (должен быть обеспечен ядром ОС).

2. Программная обработка прерывания:
   (a) Складываем в стек регистры, которыми пользуемся;
   (b) Производим «срочные действия» (например, если ранее пришло прерывание от таймера);
   (c) Размаскируем прерывания, которые не будут мешать;
   (d) Длительная обработка прерывания;
   (e) Восстановление регистров;
   (f) Вызов инструкции `iret` — возврат в нормальный режим работы (или в предыдущий обработчик).

5.2 Создание прерываний

Для создания прерывания есть инструкция `int` (interrupt)

```
int <interruption #>
```

Прерывания делятся на

- Внутренние прерывания;
- Внешние прерывания.

Внутренние прерывания обычно более приоритетны.
Примеры внутренних прерываний:

- Деление на 0
- page fault
- access error
- trap — необходимость отладки
- illegal insruction
- bad memory access
- ...

5.3 Прерывание таймера

5.3.1 Watch dog

В процессоре есть регистр `NMI`. Со временем число в этом регистре инкрементируется или декрементируется. В тот момент, когда этот регистр стал нулём, создаётся немаскируемое прерывание. Если это случится, мы получим BSOD. Если получится, будет создан дамп памяти. Это нужно для отладки операционной системы.

Такой же таймер есть на материнской плате. Он используется для отладки внешних устройств. Если он занулится, то будет перезагрузка питанием.

Диапазоны виртуальной памяти

Если мы находимся в привилегированном режиме ядра, то ты мы сами должны позаботиться, чтобы *NULL* указывал на никуда неотображаемый адрес.

Схема виртуальной памяти ядра:

1. Данные ядра
   - NULL
   - Адреса input-output
   - DMA
   - Код ядра
   - Данные
2. Куча ядра
   - Организация кучи («дерево»)
   - Диапазон адресов
3. Некоторый пустой промежуток
4. Стек ядра

   Стек ядра и стек обычной программы не должны пересекаться.

kmalloc — захватывает кусок данных в куче ядра

Схема виртуальной памяти обычной программы:

1. Ядро
2. Код
3. Данные
4. Куча
5. Пустое пространство
6. Стек

6 Процесс

Процесс — программа, обладающая некоторыми правами на ресурсы.

Контекст процесса:

1. Аппаратная составляющая (процесс ей пользуется только во время своего выполнения)
   - Регистры
   - Адреса в ОП
2. Пользовательская составляющая
   - Код процесса

- Данные процесса

3. Системная составляющая
    - Таблицы страниц
    - Целочисленные значения
    - Ограничения
    - Таблица открытых файлов
    - Список обработчиков сигналов
    - Сигнальная маска и вектор сигналов, очередь сигналов
    - Список IPC-объектов
    - Список аргументов `main`

Исполняемый файл (ELF, MZ):

1. Код
2. Данные
3. Служебная информация

Целочисленные значения:

- nice — приоритет процесса
- pid — уникальный идентификатор процесса
- uid — идентификатор пользователя, от имени которого запускается процесс
- gid — идентификатор группы, от имени которой запускается процесс
- euid — «эффективный» идентификатор пользователя
- egid — «эффективный» идентификатор группы
- cgroupid — задает множество ограничений на ресурсы
- sessionid — идентификатор сессии (нужен для массового завершения процессов)
- pgid — идентификатор процессной группы (используется для работы с терминалом)

Пример использования euid и egid:
passwd — изменяет пароль. etc/shadow — здесь хранится пароль. Мы запускаемся с gid = 0, uid = 0. Таким образом, euid и egid нужны для того, чтобы понять кому мы меняем пароль.

- getcwd — позволяет получить текущий каталог
- setcwd — позволяет задать текущий каталог
- chroot — позволяет сменить корень процесса

Ограничения на ресурсы, доступные процессу:

- Жесткие
- Мягкие

Ресурсы, которые могут быть доступны процессу:

1. Размер виртуальной памяти
2. Размер стека
3. Количество открытых файлов
4. Размер файла core (файла дампа)
5. Размер pipe
6. Число создаваемых процессов

Классификация ядер

1. Монолитное ядро
2. Расширяемое ядро (реализуется с помощью модулей)

   Модуль ядра — некий объектный код, который компилируется отдельно от ядра, но обладает жестким интерфейсом вызовов и способом описания (имя модуля, зависимости).

   insmode — подгружает модули в ОП
3. Микроядро
   - Понятие процесса
   - Внутренняя шина

Монолитное ядро работает быстрее, чем расширяемое. Но при этом сложно надежное написание, затруднена отладка.

6.1 Жизненный цикл процесса

1. Создание процесса. Процесс создаётся обычно при помощи `fork()`, но на самом деле созданием процессов можно управлять на более тонком уровне при помощи системного вызова `clone(...)`, а `fork` является неким макросом над ним.
2. Попадание в очередь готовых к выполнению. В этой очереди все процессы сортируются согласно приоритету. У приоритета есть статическая и динамическая составляющие. Динамическая задаётся ядром, статическая — частично ядром, частично пользователем.
3. Ядро выбирает процесс с наименьшим значеним приоритета и назначает его на выполнение.

   Из состояния выполнения можно вернуться в очередь готовых к выполнению, а можно перейти в одну из следующих стадий

   - Стадия «зомби». В этом состоянии процесс выполняться не будет, тело процесса не существует, но есть информация о том, как завершился процесс и всё ещё занята строчка в таблице процессов.

     После этого процесс может перейти в состояние полного завершения
   - Состояние остановки (по получению сигнала).

     Отсюда можно попасть в очередь готовых к выполнению, а оттуда можно перейти как к выполнению, так и в состояние зомби и даже обратно в состояние остановки. В состояние зомби из очереди готовых к выполнению процесс может перейти при получении `SIGKILL` от ядра

- Ввод-вывод. В это время работа процесса приостановлена, но он считается активным. Самостоятельно (без использования `read` и `write`) перейти в эту стадию процесс может только при своппинге, если нужная область памяти выкачена на внешний диск.

  После этого можно перейти либо в очередь готовых к выполнению, либо в состояние «зомби»

6.2 Создание процесса

Процесс создаётся при помощи системного вызова `clone(...)` или же макроса над ним — `fork()`. Они возвращают `pid_t` — pid порождённого процесса. `fork` создаёт новый процесс, который является копией родителя.

При создании процесса **не копируются**:

- Вектор и очередь сигналов
- `pid`, `ppid`

Вся остальная информация копируется, в том числе и открытые файлы. При этом курсор в этих файлах будет общий, т.е. при сдвиге курсора в одном из процессов, курсор в другом процессе также сдвинется. Но при открытии новых файлов, они будут независимы.

Процесс состоит из

- Инструкций
- Информации
- Стека процесса

При использовании `fork` инструкции не меняются, а информация в памяти вместе со стеком копируется. Если копировать всю информацию сразу, то это будет долго. Для этого используется ленивое копирование.

При ленивом копировании страницы виртуальной памяти копируются и помечаются read-only. Пока происходят только запросы на чтение, всё нормально. Как только происходит запрос на запись, операционная система получает page fault и выделяет новому процессу память, модифицируя таблицу страниц, снимая флаг read-only у родительского процесса.

`vfork` не запускает ленивое копирования, а просто блокирует родительский процесс до момента смены тела нового процесса.

6.3 Смена тела процесса

- Мягкий способ — `mmap`. Берётся некий бинарный файл и отображаем его на определённые адреса в виртуальной памяти.
- Тоже мягкий способ — динамически подгружаемые библиотеки — `dlopen` — расширяет сегмент кода и интерпретирует часть файла как код, добавляя его в сегмент кода.
- Жёсткий способ — `execve`.

  ```
  int execve(const char *filename, char *const argv[], char *const envp[]);
  ```

  При данном способе смены тела процесса сегмент кода полностью заменяется на содержимое файла.

Переменная окружения — строка вида `<имя переменной>=<значение>`
Библиотека `libc` предоставляет различные надстройки над `execve` вида `exelp`, `execle` и другие.

6.4 Об «отцовстве» процессов

В системе существует некоторая древовидная иерархия процессов с корнем в процессе 1 (init). Имеется две сущности: идентификатор сессии `sesid`, который создаётся при входе в систему. Сразу при входе в систему создаётся процесс, который является лидером сессии. Обычно это shell, но, в целом, это может быть любая другая программа.

У всех процессов, которые наследуются от лидера сессии, один и тот же `sesid`.

Примерно также имеется идентификатор группы `gid`. Сменить идентификатор сессии можно только на свой `pid`, тогда как идентификатор группы ещё и на `pid` отца.

А что делать, если процесс осиротел? В этом случае его отцом становится init, который его `wait`'ает.

Если же процесс умер, но у него остались потомки, то init начинает их истреблять. Сначала он делает это вежливо (`SIGTERM`), а потом грубо (`SIGKILL`).

6.5 Переключение процессов

6.5.1 Режимы переключения

В случае, когда процессов много, а ядер мало, нужно уметь переключаться между процессами. Для этого есть несколько решений:

- Пакетный режим. В этом режиме процессу выставляется предельное время исполнения, по истечении которого процесс принудительно завершается.

  Такая система хороша в случае, когда переключение между процессами слишком дорогостоящая операция (например, при больших вычислениях на кластерах).

- Режим разделения времени. Каждому процессу выделяется некий квант времени. Операционная система по прерываниям таймера смотрит в очередь готовых на выполнение и при наличии процесса с меньшим приоритетом инициализирует новый процесс, «откладывая» старый в очередь на готовых к выполнению.

- Режим реального времени. В этом режиме на каждую операцию устанавливается так называемый директивный интервал. В общем случае это интервал, в который операция должна выполниться. При этом должна быть гарантия, что операция успеет выполниться за данный интервал. В таких системах у процессов нет приоритетов как таковых, а всё планируется заранее и переключение происходит по границам директивных интервалов.

  У директивного интервала может не быть правой границы, но левая должна быть всегда.

6.5.2 Механизм переключения

При переключении ядро должно сделать следующее:

1. Сохранить состояние регистров
2. Подгрузить таблицу страниц нового процесса в оперативную память
3. Модифицируем таблицы процессов, меняем `pid` активного процесса
4. Понижение прав и переход на отложенный адрес в другом процессе

6.5.3 Квант времени планирования процессов

Δt — квант времени планирования. Это непрерывный интервал, внутри которого переключение процессов не допускается. Это параметр ядра, то есть устанавливается при его компиляции.

Можно сделать

1. Маленький промежуток

   - Отзывчивость системы. Помогает на ноутбуках и телефонах. Например, процесс, отлавливающий нажатие элемента на экране, довольно быстро среагирует на нажатие. Это называется **io-intensive** — интенсивный ввод-вывод.

2. Средний промежуток — некоторый компромисс между большим и малым промежутком.

3. Большой промежуток. Если сделать интервал слишком большим, то получится пакетный режим, поэтому такой промежуток всё же исчисляется микросекундами.

   - Помогает для больших вычислений. Тратится мало времени на переключение процессов, процесс может долго заниматься одним вычислением, не теряя при этом кеш-память.

     То есть помогает для серверного режима и для трудоёмких вычислений.

6.5.4 Приоритеты

Приоритет процесса состоит из

- Динамической составляющей
  - Динамический приоритет, понижающийся со временем нахождения в очереди. Если процесс недавно выполнялся, то эта составляющая устанавливается большой и понижается со временем.

    При `fork`'е, у порождённых процессов выставляется маленькая динамическая составляющая, поэтому при завершении кванта времени будут сначала выполнены именно они, а затем, после уравнивания приоритетов, отцовский процесс.
- Статической составляющей
  - Приоритет пользователя (у пользователя 0 приоритет должен быть немного выше, чем у обычных пользователей)
  - Приоритет, назначаемый пользователем
  - Число, добавляемое ядром, чтобы приоритет пользовательских процессов был больше, чем у внутренних процессов ядра

6.5.5 Процессы реального времени в системах с разделением времени

Это процессы, при возникновении которых все остальные процессы будут остановлены. Его нельзя прерывать никаким процессам, кроме процессов ядра.

Пример необходимости такого процесса — запись на CD-ROM. Если диск будет раскручен, лазер зажжён, а данный перестанут поступать из-за переключения процесса, то болванка будет испорчена.

7 Потоки (threads)

Потоки — это некоторая идеология. В этой идеологии внутри процесса создаются так называемые «нити», которые исполняются в этих самых потоках, но не являются независимыми процессами, а являются лишь частью процесса. Это поток команд, который выполняется в адресном пространстве процесса. На практике, в системе Linux это почти обычные процессы, созданные, например, операцией `clone` со специальными параметрами.

Количество этих нитей обычно совпадает с количеством ядер в машине (например, на 4-ядерном процессоре без Hyoer-Threading'а делают 4 «нити»). В Linux количество нитей ограничено количеством процессов, которые можно порождать.

Для создания «нитей» можно использовать встроенные интерфейсы, параметры компилятора (`#pragma ...`) а также библиотечную функцию `pthread_create`. Функция, с которой создаётся процесс, должна возвращать `void*`, обычно возвращает `NULL`, а результат выполнения получают через глобальные переменные

У нити и исходного процесса

- Сегменты данных **идентичны**, при изменении переменной в одной нити, она меняется везде
- Сегменты кода **идентичны**, как и при `fork`
- Стеки **разные**. При создании новой «нити» создаётся новый стек. На вершину стека кладётся указатель на функцию, которая используется для создания потока.

Это почти то же самое, что и при `exec`'е, за исключением того, что происходит переключение на на динамический линковщик, а сразу на реализацию функции.

7.1 Потоки в POSIX

Для использования потоков есть библиотека `<pthread.h>`. Использование выглядит примерно так:

```
pthread_t thread;
pthread_create(pthread_t *thread, attr, body, void* args);
```

В аргументах передаётся указатель на переменную потока, адрес функции и аргументы, с которыми будет вызвана функция. После порождения потока и его завершения, нужна синхронизация. Для этого используется `pthread_join(thread, void*)`, которая дождётся завершения завершения потока и по второму аргументу получит результат выполнения функции.

Я тебя породил, я тебя и убью.

Поток можно уничтожить при помощи `pthread_cancel(thread)`.

Поток может и сам завершиться досрочно при помощи `pthread_exit(void*)`.

Поток можно отсоединить от родительского процесса при помощи `pthread_detach(thread)`.

8 Сигналы

8.1 Реакция на сигналы

1. Игнорирование. В этом случае он возникает и... убивается операционной системой. Происходит ничего
2. Завершение без дампа памяти. Выполняется операционной системы
3. Завершение с дампом памяти
4. Остановка процесса
5. Продолжение работы
6. Вызов пользовательской функции-обработчика

Есть два «неприличных» сигнала:

- `SIGSTOP` — принудительная остановка процесса. В режиме остановки он может быть убит только при помощи `SIGKILL`, но за одним исключением. Если процесс находится в режиме ввода-вывода и ядро не может гарантировать согласованность завершения с точки зрения оперативной памяти
- `SIGKILL` — принудительное завершение процесса.

8.2 Сигнальная маска

Обычные сигналы можно маскировать. Маска сигналов (или вектор сигналов), является числом-битовой маской и принадлежит ядру.

Передача сигнала процессу происходит при переключении на этот процесс, кроме сигналов `SIGSTOP` и `SIGKILL`, так как при обоих этих сигналах нет смысла передавать управление процессу.

Для порождаемых потоков маска сигналов установлена в маскирование всех сигналов, сигнал передаётся в основной поток. Сигналы обрабатываются только им.

8.3 Сигналы реального времени

Сигналы реального времени складываются в очередь. Номера этих сигналов расположены между `SIGRTMIN` и `SIGRTMAX` включительно. Обработка сигналов происходит в порядке их появления в очереди.

8.4 Обработчики сигналов

Обработчикам сигналов разрешается пользоваться только системными вызовами или более высокоуровневыми объектами, которые не используются в основной программе. Корректность работы гарантируется только для системных вызовов.

При наличии функции-обработчика, при передаче управления процессу, вызывается этот самый обработчик и этот сигнал маскируется.

- Старый интерфейс

  ```
  void handler(int)
  ```

- Новый интерфейс

  ```
  void handler(int, sig_info_t, void*...)
  ```

При этом функция-обработчик выполняется в собственном стеке. По завершении функции-обработчика ядро размаскирует сигнал и передаёт управление обратно в функцию.

Не исключено, что до завершения обработчика придёт другой сигнал. В этом случае будет вызван следующий обработчик

9 Виртуальные машины (уровень виртуализации)

Виртуализация — создание видимости того, что исполняемый код выполняется на нескольких машинах, при том, что на самом деле всё происходит на одном и том же железе.

9.1 Уровни виртуализации

1. Аппаратная виртуализация. Обычно используется для разделения аппаратных ресурсов между запускаемыми операционными системами. Например, при запуске нескольких ОС оперативная память может быть поделена на куски и каждой запущенной ОС будет выдан некоторый диапазон адресов, создана «иллюзия» наличия только выданного куска оперативной памяти.

2. Виртуализация с hypervisor. Задача hypervisor'а — распределение запросов к ресурсам машины от ядер запускаемых ОС. Ядра при этом модифицируются для возможности общения с hypervisor'ом.

Сам hypervisor — тоже в некотором смысле ОС, но урезанная — без управления процессами, без файловой системы и т.д.

После запуска загрузчик первым делом запускает hypervisor.

Такие виртуальные машины работают несколько медленнее, чем с аппаратной виртуализацией, за счёт наличия прослойки hypervisor'a, потребляющей некоторые ресурсы

3. Виртуальная машина в user mode. В этом случае виртуальные машины запускаются как пользовательские процессы в основной ОС.

   При таком подходе все экземпляры виртуальных ОС могут получить доступ ко всем физическим ресурсам машины одновременно. Также при такой виртуализации на виртуальных машинах доступен тот же набор инструкций, что и на основной машине

4. Полная эмуляция другой архитектуры. В этом случае скорость уже не важна, но важна полнота эмуляции, используется в основном разработчиками этих самых архитектур.

10 Управление оперативной памятью ядром

Для поддержки отображения страниц на физические адреса в ядре есть «таблица» для управления физической памятью. В этой таблице хранится список диапазонов. Внутри каждого диапазона может быть множество объектов, привязанных к этому диапазону. Объектом, например, может быть

- Операция ввода-вывода, зарегистрированная пользователем
- память, выделенная пользовательскому процессу (тогда в объекте будет храниться ссылка на контекст процесса) и другое.

С каждым диапазоном связана величина `lru` (last recently used). Это два числа — `r` и `w` — последнее чтение и последняя запись. Эти величины хранятся в некоторой внутренней системе отсчёта времени, например, в количестве тактов.

10.1 Swapping

Swap — это некоторый диапазон на внешнем запоминающем устройстве, в котором содержатся несколько диапазонов адресов ОП. Используется в случае, когда место в оперативной памяти начинает заканчиваться.

Диапазоны бывают двух типов:

1. Чистый — когда в оперативной памяти и на внешнем устройстве хранится идентичная информация.
2. Грязный — когда информация в оперативной памяти не синхронизированна с внешним устройством.

Страничный демон

Страничный демон — процесс, просыпающийся по внутреннему таймеру ядра. Он инициирует процессы записи на внешнее устройство после операции записи в оперативную память, освобождая таким образом набор диапазонов в оперативной памяти.

Есть несколько алгоритма, по которым выбираются диапазоны для выгрузки в swap:

- LRU (Last Recently Used) — выгрузка последнего использованного диапазона
- NRU (Not Recently Used). Собирает некоторую статистику и выбрасывает наименее использованный диапазон
- «Алгоритм часов» — учитывает и частоту чтения, и частоту записи, и выбрасывает (при прочих равных) то, что было реже прочитано.

10.2 Что делать, если кончилась память?

`vm.overcommit_memory` и `vm.overcommit_ratio` — величины, которые можно менять в `/etc/sysctl.conf`. В `vm.overcommit_memory` можно записать три значения

- 0 — ядро не выпадет в Kernel Panic и будет позволять программам выделять памяти больше, чем физической. Будет происходить ленивое выделение памяти. Но когда память кончится включится добрый OOM Killer.

  При помощи некоторых эвристических алгоритмов будет выбран процесс с ненулевым эффективным идентификатором пользователя и наибольшим использованием памяти... и будет убит.

- 1 — Ядро выпадает в Kernel Panic

- 2 — Считается величина $swap+mem\cdot\frac{ratio}{100}$, которая является допустимым объёмом используемой памяти. Если мы *случайно* выставим `vm.overcommit_ratio` хотя бы 100, то при переполнении реальной памяти... будет вызван OOM killer.

slab-алгоритм, baddy-алгоритм.

Kernel space

В ядре существует три «множества» памяти:

1. Начало виртуальной памяти
2. Область DMA
3. Прочее

Buddy-алгоритм и slab-алгоритм

`buddy-алгоритм` стремится минимизировать количество «дырок» в ОП.
Память поделена на куски размера 2^n. Далее выделение памяти M байт происходит так:

1. Часть делится пополам, пока она не станет минимального размера, который все еще не меньше M байт.
2. M байт записываются в соответствующую ячейку.
3. При освобождении данной ячейки она объединяется с соседней в одну ячейку большего размера.

`slab` характеризуется размером и типом объекта. Когда `slab` нужно расширить, то запускается `buddy-алгоритм`.

Работа с памятью на уровне процесса

Существует несколько функций для работы с памятью:

- malloc — вызывает функцию по работе с кучей. Если в куче недостаточно места для выделения данного объема памяти, то происходит системный вызов `brk(size)` — он увеличивает сегмент данных на `size`. `dcmalloc` — алгоритм реализации `malloc`'а. Более высокоуровневые реализации: `tcmalloc, jemalloc`.

- calloc — то же самое, что `malloc`, только отображает виртуальную память на таблицу страниц, которая отображается на `read only` страницу в ОП из нулей. Если произошла запись, то данный диапазон адресов уже не отображается на страницу из нулей, а создается новое отображение на другую часть ОП.

- realloc — манипулирует с таблицей страниц виртуальной памяти. При вызове данной функции происходит копирование таблицы страниц, то есть наблюдалась такая ситуация: диапазоны адресов отображаются через таблицу страниц на ОП, нужно добавить еще кусок к данному диапазону, но этот кусок перекрывается с другим, таким образом, нужно перезаписать таблицу страниц и добавить к ней нужный кусок (ведь мы хотели получить непрерывный кусок памяти).
- free — освобождает память.

Виртуальная файловая система

Виртуальная файловая система имеет иерархическую структуру.

С ней можно выполнять операции монтирования и размонтирования.

Монтирование — момент, когда ядро использует тройку <драйвер устройства, драйвер файловой системы, точка монтирования>, чтобы создать связку между файловой системой, файлом устройства и точкой монтирования.

Типы объектов файловой системы

1. Файл
2. Каталог
3. Файл устройства

   `/proc` и `/sys` состоят полностью из файлов устройств
4. Символическая ссылка — некоторый файл, в котором прописан путь до данного файла
5. PF_UNIX (для socket())
6. FIFO (для mkfifo()) — то же самое, что и `pipe`, только с именем

Обобщенный индексный дескриптор — представление типа объекта файловой системы. Этот дескриптор должен существовать для любого объекта в файловой системе.

Что же записано в дескрипторе:

1. Тип
2. Права доступа: u(rwx), g(rwx), o(rwx), suid bit, t bit, acl (access control list) — ассоциирует шаблон пользователя (группы) с группой (пользователем)
3. Владелец — uid
4. Группа — gid
5. Даты чтения и модификации
6. Файл устройства
7. Ссылка на данные устройства, где расположен физический индексный дескриптор

Системный вызов `stat` возвращает обобщенный индексный дескриптор.

Каталог с именем «..» — ссылка на каталог выше по иерархии, с именем «.» — ссылка на текущий каталог.

В каталоге хранится:

1. Имя файла
2. Ссылка на индексный дескриптор
3. Тип файла

Список системных вызовов для еще не открытого файла:

- stat — возвращает индексный дескриптор
- open — открывает файл
- link — создает символическую ссылку
- unlink — удаляет символическую ссылку
- mmap — отображает файлы в ОП

Список системных вызовов для открытого файла:

- close — закрывает файл
- read — читает из файла
- write — записывает в файл
- fstat
- lseek

Файлы устройств

Для файлов устройств в индексном устройстве также записаны:

- Мажорный номер — определяет шину, на которой записано устройство
- Минорный номер — номер устройства на шине

Для регистрации устройств используется программа `udev`(раньше — скрипт MAKEDEV). Просматривает `/sys/bus` и, если были какие-то изменения, создает файлы устройств.

Например, при подключении USB-устройства ядро подгружает драйвер устройства, далее создаются указатели на реализации функций для работы с файлом устройства как объектом файловой системы. Ссылка на вектор с реализациями функций лежит в таблице с минорными номерами.

11 Файловая система

11.1 Где может храниться файловая система

Файловая система может храниться на **блочном устройстве**. Это устройство, у которого определены операции чтения и записи, но обратиться к одному байту нельзя, а можно читать и записывать только **блоками**.

Блочные устройства могут быть физическими и виртуальными.

Физические:

- HDD
- SSD
- Оптика (Blu-ray, DVD, ...)

- Магнитная лента

Все устройства делятся на устройства

- Последовательного доступа — к ним относятся магнитные ленты и, в некоторой степени, оптические диски.

  Чтобы добраться до нужной ячейки, нужно пролистать всё до этой самой ячейки.

- Произвольного доступа

11.1.1 Магнитная лента

Лента устроена как, собственно, лента и головка записи-чтения. Лента делится на дорожки, которые делятся на биты. У ленты есть свойство повреждаться, поэтому нужен достаточно надежный способ записи.

Надёжность зависит от плотности записи, которая зависит от таких характеристик как зерно лента. Для надежности некоторые паттерны записываются несколько раз.

Файловая система чаще хранится на паре лента+SSD, где на SSD хранится оглавление того, что лежит на ленте.

11.1.2 Оптика

Диск состоит из круговых дорожек и радиальных секторов.
Проблемы:

1. Одноразовая запись
2. Перезапись конечное число раз, много меньшее, чем у SSD

Файловая система для оптических дисков — ISO 9660. Эта файловая система похожа на систему для ленты. В этой файловой системе есть так называемые сессии. В начале диска хранится оглавление и копия оглавления. Остальное хранится как файлы и директории на диске, причём они не бьются на блоки.

В случае необходимости удаления файл не стирается, а просто «забывается», а при изменении просто записывается новая копия файла, снова забывая про старую.

Есть другая файловая система — UDF. Эта система блочная. Файлы записываются по блокам, система предназначена для изменения файлов и возможности перезаписи. В этой системе файл не копируется полностью, а обновляется только небольшое число блоков.

11.1.3 SSD

SSD имеет ограниченное количество циклов перезаписи. Для решения этой проблемы при последовательной записи блоки перемешиваются как можно более случайно для равномерного износа.

Что же хорошо и плохо хранить на SSD?

- **Плохо**. swap-раздел. Казалось бы, всё будет хорошо, веь большая скорость — идеально для swap'а. Но при этом нужно быть готовым, что устройство быстро выйдет из строя.
- Хорошо — корень и `/usr`, так как они часто читаются и редко записываются

11.1.4 HDD

Жёсткий диск устроен как набор поверхностей, над которыми висят прикреплённые к общему кронштейну головки чтения-записи. У данных на жёстком диске есть три координаты (поверхность, дорожка, сектор). Понятно, как определить поверхность и дорожку, а как определить сектор?

Сектор определяется при помощи отсчёта нужного количества миллисекунд.

Плохо в жестких дисках то, что перемещение головки занимает много времени, а быстрее перемещать нельзя из-за проблемы перепада температур.

Жёсткий диск сам оптимизирует количество перемещений головки, раньше этим занималась операционная система.

Для более простого с точки зрения ОС доступа была придумана такая вещь, как LBA (Linear Block Array).

Виртуальные блочные устройства

11.1.5 Таблицы разделов

В этом случае какой-то участок LBA-устройства выделяется на хранение таблицы разделов, а остальное пространство делится на части, которые и называются разделами. Далее предполагается, что на этих разделах будет находиться файловая система.

Множество разделов для файловой системы полезно тем, что позволяет локализовать урон при bad block, а также

В мире Intel есть две основные системы:

- MBR (Master Boot Record) — поддерживает лишь четыре раздела. Нельзя сделать MBR больше, чем на 1 TB.

  extended раздел — раздел со своей собственной MBR.

  Устроена она так:

  - 512 байт загрузчика
  - MBR
  - MBR (копия)
  - Разделы. Для хранения типа раздела используют 1 байт.

- GPT (GUID Partition Table). Поддерживает вложенные extended-разделы, копия хранится в конце, а не в начале. Для хранения типа раздела используют уже 2 байта. Есть два важных типа разделов: bios boot data и efi boot.

  Также у каждого раздела есть подпись, которая как-то обеспе

11.1.6 Сетевые блочные устройства

```
/dev/nfs
/dev/nvrom...
```

11.1.7 Блочные устройства в оперативной памяти

Минус — при выключении питания теряются данные

11.1.8 Много дисков как одно устройство

Есть два главных типа таких устройств:

- RAID 0-6. Остальные номера — комбинации. В RAID есть определение «полосы»
  - RAID 0. Создаётся устройство с размером — суммой размеров входящих в него устройств. При этом появляется явление расслоения памяти, когда одна полоса делится на части и части записываются на разные диски (например, в порядке 1, 2, 3, 1, 2,...). В этом случае скорость записи-чтения теоретически может увеличиться в n раз. Проблема в том, что если один из дисков выйдет из строя, то всё устройство прекращает своё существование. Удобно для создания каких-нибудь tmp
  - RAID 1. Предполагает два диска. Размер совпадает с размером каждого из дисков в массиве. Запись происходит сразу на два диска, причём при поломке одного из дисков система происходит перевод массива в режим read-only или происходит подключение spare-диска, информация при этом не теряется.
  - RAID 2. Содержит n дисков. Информация как-то более-менее равномерно размазана по всем дискам. Информация хранится в виде массива и кодами Хэмминга, восстанавливающими ошибки.
  - RAID 3. Один из дисков используется как диск с контрольными суммами (блок чётности), где хранятся XOR-суммы данных на остальных дисках. Так, при помощи контрольных сумм можно будет восстановить любой повреждённый диск. Проблема в неравномерной нагрузке на диски.
  - RAID 4. RAID 4, но деление полосы на байты, а не на блоки.
  - RAID 5. RAID 3, но контрольные суммы распределены по дискам.
  - RAID 6. RAID 5, но 2 контрольные суммы. Таким образом гарантируется восстановление после выхода из строя двух жёстких дисков.
- LVM. Создаётся виртуальное блочное устройство Logical Volume. Он состоит из участков других блочных устройств. Также есть такая вещь, как Volume group, которая может состоять из нескольких Logical Volume и множества блочных устройств. В дальнейшем блочные устройства можно распределять по Logical Volume.

  Эта система предназначена для сжатия и расширения файловой системы, когда это необходимо. Например, в больших дата-центрах могут использовать LVM натянутый на RAID 6.

12 Процесс загрузки

1. Аппаратная стадия
2. Работа BIOS
3. Загрузчик 1-й стадии
4. Загрузчик 2-й стадии
5. Старт ядра
6. Старт init
7. Разворачивание служб и пользовательских программ

12.0 Аппаратная стадия

Обычно в эту стадию не задействуется ЦП, эта стадия — прерогатива материнской платы. Цель этой стадии — проверка того, что подключено к материнской плате (наличие процессора и памяти. проверка памяти). Затем настраивается контроллер прерываний, чтобы BIOS мог работать с ними.

12.1 Работа BIOS

BIOS содержит в себе некоторые настройки. BIOS грузится в оперативную память, используется ЦП. Грузятся некоторые настройки, например, производится опрос устройств, которые были записаны в системный раздел памяти, для поиска загрузчика 1-й стадии. Или переводит процессор в режим наибольшей совместимости

12.2 Загрузчик 1-й стадии

Запускает загрузчик 2-й стадии.

12.3 Загрузчик 2-й стадии

Перевод процессора обратно в нужный режим. Обычно принимает что-то в формате файловых систем, что передавать ядру и как его размещать в оперативной памяти.

Также он может искать драйвера в файловой системе.

После размещения ядра, управление передаётся ему и загрузчик перестаёт существовать

12.4 Старт ядра

Разворачиваются все необходимые таблицы, распознаёт и конфигурирует устройства так, как ему надо. Затем монтируется корневая файловая системаи запускается init.

12.5 Старт init

В случае Linux есть systemd и sysvinit.

12.6 Разворачивание служб и пользовательских программ

13 ext*, UFS, (NTFS?), FAT

В этих файловых системах файлы хранятся как набор разрозненных блоков в разных местах

13.1 FAT (File Allocation Table)

В файловой системе от начала диска лежат:

- массив свободных-занятых блоков (File Allocation Table)

  1. Блок свободен
  2. Занят файлом
  3. Системный — обычно хранилище swap, системные блоки должны идти подряд
  4. bad-block

  Также в этом массиве хранится различная метаинформация, например, размер блока. Сама таблица является суперблоком.

- Копия FAT
- Первый блок после копии FAT — корневой каталог файловой системы.

Блок файловой системы FAT поделен на две области: данные и, в конце блока, ссылка на следующий блок. Ввиду такой системы в FAT нет нормального индексного дескриптора, а информация о размере файла хранится в каталоге.

Из-за разделения файлов на блоки есть явление фрагментации, которое, к слову, теряет свою негативную сторону при использовании SSD.

Большие файлы хранить невозможно из-за нескольких ограничений:

- Размер ссылки на следующий блок (ограничение на размер устройства)
- Размер описания размера файла в каталоге.
- Размер блока

13.2 Другие файловые системы

В какой-то момент люди поняли, что FAT ущербен и начали придумывать что-то другое. Была придумана следующая схема:

- Суперблок

  1. размер файловой системы в блоках
  2. размер блока (должен быть кратен размеру блока блочного устройства, актуально на всех системах)
  3. дата последнего монтирования и место монтирования
  4. label — метка файловой системы (например, на флешках «KINGSTON 32GB»)
  5. UUID (Universally Unique Identifier) — уникальный идентификатор экземпляра созданной файловой системы, вероятность совпадения с другим UUID, созданным в мире, крайне мала. Может быть использован вместо label'а для монтирования файловой системы
  6. дата последней проверки проверки целостности
  7. была ли файловая система размонтирована (если нет, то произошёл сбой, например, отключение питания, перед повторным монтированием обязательна проверка целостности)

8. битовый массив свободных-занятых блоков.
9. число индексных дескрипторов

Всё дисковое пространство делится на группы блоков, в начале каждой из которых лежит суперблок. Такой подход обеспечивает большую «выживаемость» данных по сравнению с FAT.

- Суперблок группы

  Некоторая часть суперблока, которая записана в каждой группе после суперблока. В основном предназначен для хранения массива свободных-занятых блоков данной группы.

- Массив индексных дескрипторов

  Содержит в себе метаинформацию о группе блоков

13.2.1 Немного про индексные дескрипторы

В индексном дескрипторе, помимо информации, обсуждённой ранее, хранится также и счётчик ссылок. В UNIX-системах на одну область файловой системы может хранится несколько так называемых **жёстких ссылок**. По сути это имена, которыми называется данный файловый дескриптор, информация об этих именах хранится в файлах каталогов.

В момент удаления файла, счётчик ссылок уменьшается на 1, а при достижении 0 происходит «лёгкий ад»

Также в индексном дескрипторе есть такая вещь, как xattr. В нем хранится метаинформация, котрая не предусмотрена базовой системой.

13.3 Регулярные файлы и каталоги

Информация в индексном дескрипторе регулярного файла:

1. Размер в байтах
2. Способ хранения в блоках данных. В UNIX используется массив номеров блоков, которые заняты данным файлом. До тех пор, пока для хранения файла хватает, условно, 10 блоков, то для нумерации используется этот массив напрямую. Иначе следующий блок за этими 10 интерпретируется не как блок данных, а блок с массивом номеров блоков.

   Если же и этого не хватает, то следующий блок указывает на массив номеров блоков, которые интерпретируются так же, как и предыдущий блок — массивы номеров действительных блоков. И таким образом это дерево блоков может разрастаться вплоть до границы массива номеров блоков в индексном дескрипторе. Размеры в блоках так будут составлять 10, 110, 1110, ...

13.4 А чего бы ещё навернуть?

13.4.1 Файловые системы с дырками

Где-то в суперблоке можно завести специальные блоки, которые часто повторяются в файловой системе (например, блок из одних нулей). При попытке чтения такого блока драйвером файловой системы будут выданы нули без обращения к диску. Если же файловая система достаточно умная, то она может обнаружить, что есть много однотипных блоков, пометить их дырявыми и ссылаться на один и тот же блок.

13.5 Файл каталога

Содержимой одной записи файла каталога:

- offset_next — смещение до следующей записи
- Тип объекта
- Имя объекта — имеет фиксированный размер
- inode

Файл каталога состоит из множества таких записей.
Удаление файла из каталога происходит путём... Изменения offset'а предыдущей записи для «перескока» через удаляемый файл. Из-за этого файлы каталогов имеют неприятное свойство со временем расти.

13.6 Проверка целостности

С этой точки зрения существует 3 типа файловых систем:

1. Без файла журнала
2. С файлом журнала. Этот файл журнала реализует аналог транзакции из области баз данных, в котором хранится информация о текущей операции. После завершения операции файл журнала очищается.

   Если же операция не была завершена, то при проверке целостности изменения будут откачены.
3. С разделом журнала.

   В более продвинутых системах, например, в больших БД, журнал хранит все изменения файловой системы с возможностью отката до любого предыдущего состояния. По этому принципу работает zfs — довольно новая система, предназначенная для хранения огромных данных, и, возможно, UFS2. И, конечно же, jfs (journal file system) от IBM. Также существует Razerfs.

Как же проходит проверка целостности?

1. При наличии, разбираемся с журналом
2. Суперблок копируется в оперативную память (или «сумма» всех суперблоков)
3. Строим в оперативной памяти массив (не битовый) по числу блоков файловой системы и инициализируем нулями
4. Проход по области индексных дескрипторов. Если дескриптор используется, то выставляем в массив блоков 1, иначе оставляем 0

   Конфликты могут быть вида:

   - 0 в суперблоке, 1 в массиве (в суперблоке не используется). В этом случае лучше сказать, что на самом деле индексный дескриптор не используется.
   - 1 в суперблоке, 0 в массиве (в суперблоке используется). В этой ситуации нужно пройти по списку каталогов и если он хоть где-то встречается, то нужно его пометить как используемый, иначе грохнуть в суперблоке.
5. Проход по индексным дескрипторам, инкрементируя значение в массиве у всех блоков, на которые ссылается данный индексный дескриптор

   Конфликты могут быть вида:

- 0 в суперблоке, 1 в массиве (в суперблоке не используется). В этом случае суперблок помечается 1.
- 1 в суперблоке, 0 в массиве (в суперблоке используется). Можно просто стереть блок. Но эту операцию нужно делать в последний момент.

  Эта ситуация может может возникнуть в 2-х случаях: в случае повреждения суперблока (тогда всё нормально) и в случае повреждения индексного дескриптора. Тогда должен возникнуть блок или первого, или третьего типа.
- 1 в супер блоке, >1 в массиве. В этой ситуации делается копия блока и один или оба блока помещаются в lost and found. При наличии таких конфликтов все блоки типа 2 помечаются 1 и помещаются в lost and found.