Лекция 24
Работа с астрономическим временем

Время в Unix

- Тип для хранения времени: time_t
- Тип знаковый, допускаются отрицательные значения
- Время хранится в UTC
- Число секунд, прошедших с 1970-01-01 00:00:00 UTC
- Leap seconds не учитываются (т.е. в сутках всегда 86400 секунд)

Недостатки

- Если sizeof(time_t) == 4, то проблема переполнения time_t (проблема 2038 года)
- Точность в 1с часто недостаточна
- Необходимость специальной обработки leap seconds

Работа со временем

```
#include <time.h>

time_t time(time_t *tptr);


struct tm *localtime_r(time_t *tptr, struct tm *res);
struct tm *gmtime_r(time_t *tptr, struct tm *res);


time_t mktime(struct tm *ptm);
time_t timegm(struct tm *ptm); /* non-standard */
```

Struct tm

```
struct tm {
    int tm_sec;         /* seconds */
    int tm_min;         /* minutes */
    int tm_hour;        /* hours */
    int tm_mday;        /* day of the month */
    int tm_mon;         /* month */
    int tm_year;        /* year */
    int tm_wday;        /* day of the week */
    int tm_yday;        /* day in the year */
    int tm_isdst;       /* daylight saving time */
};
```

mktime

- Работает в локальной временной зоне
- Для корректной обработки летнего времени при вызове mktime поле tm_isdst должно быть -1
- Возвращает -1, если время не представимо в time_t
- Нормализует значения полей tm_year, tm_mon, tm_mday, tm_hour, tm_min, tm_sec
- Заполняет tm_wday, tm_yday

Более точное время

```
struct timeval {
  time_t      tv_sec;     /* seconds */
  suseconds_t tv_usec;    /* microseconds */
};


int gettimeofday(struct timeval *tv, struct timezone *tz);


struct timespec {
  time_t tv_sec;        /* seconds */
  long   tv_nsec;       /* nanoseconds */
};
```

Внутреннее устройство файловой системы

Файловый дескриптор

- Fd – универсальный способ доступа к разного типа ресурсам:
- Операции: read, write, close, dup, epoll, fcntl, ioctl
- Fd – целое число
- Индексы лучше указателей: fd – индекс в таблицу в ядре

Таблица файловых дескрипторов

- Хранится для каждого процесса
- Находится в include/linux/fdtable.h

```
struct fdtable
{
    unsigned int max_fds;
    struct file **fd;                  // file pointer
    unsigned long *close_on_exec; //CLOEXEC bitset
    unsigned long *open_fds;       // opened bitset
};
```

struct file

- Состояние открытого файла
- Находится в исходном коде ядра Linux в include/linux/fs.h

```
struct file
{
    atomic_long_t  f_count; // счетчик ссылок
    unsigned int   f_flags; // флаги open
    fmode_t        f_mode;  // внутр. флаги
    loff_t         f_pos;   // текущее смещение
    // много всего еще
};
```

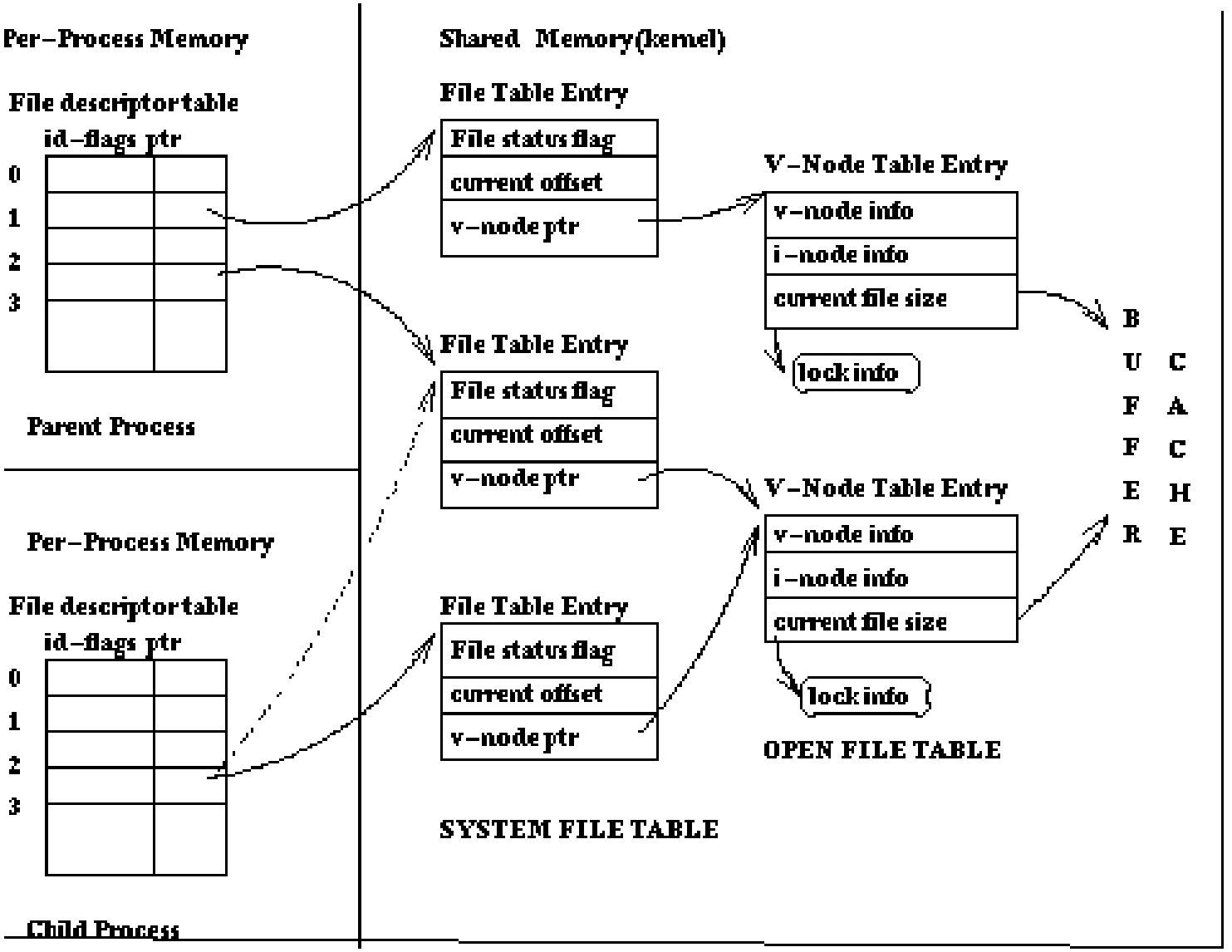
Подсчет ссылок

- При выполнении open (создание первого ф. д.): f_count = 1
- При копировании ф. д. (dup* или fork):
  ++f_count
- При закрытии ф. д. (close)
  ```
  if (--f_count == 0) {
    // реально закрыть файл:
    // сохранить несохраненные данные,
    // освободить ресурсы ядра
  }
  ```
- Подсчет ссылок — эффективный способ управления ресурсами в случае ациклических графов

Разделение открытого файла

- Все ф. д. - копии разделяют (имеют общую) следующую информацию:
  - Режим открытия файла
  - Текущую позицию в файле
- Открытый файл закрывается, когда закрывается последний ф. д.-копия
- Каждый ф. д. - копия имеет свое значение флага O_CLOEXEC

Структуры ядра



Одновременная работа с файлами

- В Unix если одновременно несколько процессов работают с копиями одного и того же файлового дескриптора или с одним и тем же файлом, и операции чтения, и операции записи разрешены без ограничений
- Процессы должны сами согласовать свое поведение, чтобы избежать порчи данных
- Варианты: флаг O_APPEND, рекомендательные блокировки, обязательные блокировки

Атомарность с файлами

- POSIX не гарантирует атомарности чтения/записи при работе с файлами
- Реально Linux записывает/считывает данные небольшого (зависит от типа ФС, около 1KiB) размера атомарно, то есть при записи данные двух процессов не перемешаются
- **НО! Запись/чтение данных и изменение значения текущей позиции в совокупности могут быть не атомарны! В современных ядрах – атомарны.**

Чтение/запись с файлами

- Процесс 1

  write(fd, "123\n", 4);

- Процесс 2

  write(fd, "456\n", 4);

- Два возможных результата:

  123
  456

- Или

  456
  123

Блокировки файлов (file locking)

- Advisory (рекомендательная) — для процессов, которые добровольно соглашаются соблюдать блокировки
  - Процесс может игнорировать блокировки других процессов
- Mandatory (обязательная) — для любых процессов
  - Не везде поддерживаются
  - Требуют специального монтирования файловой системы

Типы блокировки

- Read (shared) — блокировка на чтение. Несколько процессов могут заблокировать ресурс на чтение, при условии, что нет блокировок на запись
- Write (exclusive) — единственный процесс блокирует на запись, нет блокировок на чтение
- Если требуемый тип блокировки не может быть немедленно удовлетворен, процесс переводится в состояние ожидания или блокировка завершаеся ошибкой

Системный вызов fcntl

```
struct flock {
    short l_type;    /* F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;  /* SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
(F_GETLK only) */
};


struct flock flk;

int res = fcntl(fd, OPER, &flk);

// OPER – один из F_SETLK, F_SETLKW, F_GETLK
```

Операции fcntl

- F_SETLK
  - F_RDLCK — заблокировать на чтение
  - F_WRLCK — заблокировать на запись
  - F_UNLCK — разблокировать
  - Если операция невозможна, возвращается ошибка EACCESS или EAGAIN
- F_SETLKW
  - Те же операции блокировки
  - Если операция невозможна, процесс блокируется
- F_GETLK
  - Проверить возможность блокировки
  - Если блокировка невозможна, получить информацию о процессе

Особенности fcntl

- Per-process, т. е. каждый процесс может иметь только один тип блокировки на каждый конкретный байт файла
- Advisory, т. е. системные вызовы read и write не проверяют наличие и тип блокировки
- При закрытии любого файлового дескриптора, связанного с файлом, в этом процессе блокировки процесса сбрасываются

Mandatory locking

- В Linux необходимо выполнить следующее:
  - Разрешить обязательные блокировки при монтировании
    mount DEVICE PATH -o mand
  - Файл не должен иметь разрешение исполнения на группу (бит 010 прав)
  - Файл должен иметь sgid бит (02000)
- Тогда read write будут проверять блокировку файлов и переводить процесс в состояние ожидания в случае конфликта

Квотирование

- Можно квотировать (ограничить число):
  - Индексных дескрипторов (т. е. файлов)
  - Блоков данных (т. е. суммарный размер файлов)
- Типы квоты:
  - Hard — ограничение не может быть превышено
  - Soft — ограничение может быть превышено на ограниченное время (grace period)
- Квоты могут применяться к пользователю и группе

Управление квотированием

- Квотирование включается при монтировании файловой системы
  mount DEV PATH -o usrquota,grpquota
- Базы данных квотирования создаются и управляются с помощью
  quotacheck OPTIONS
- Квота для пользователя (группы) редактируется с помощью
  edquota

Типы файловых систем

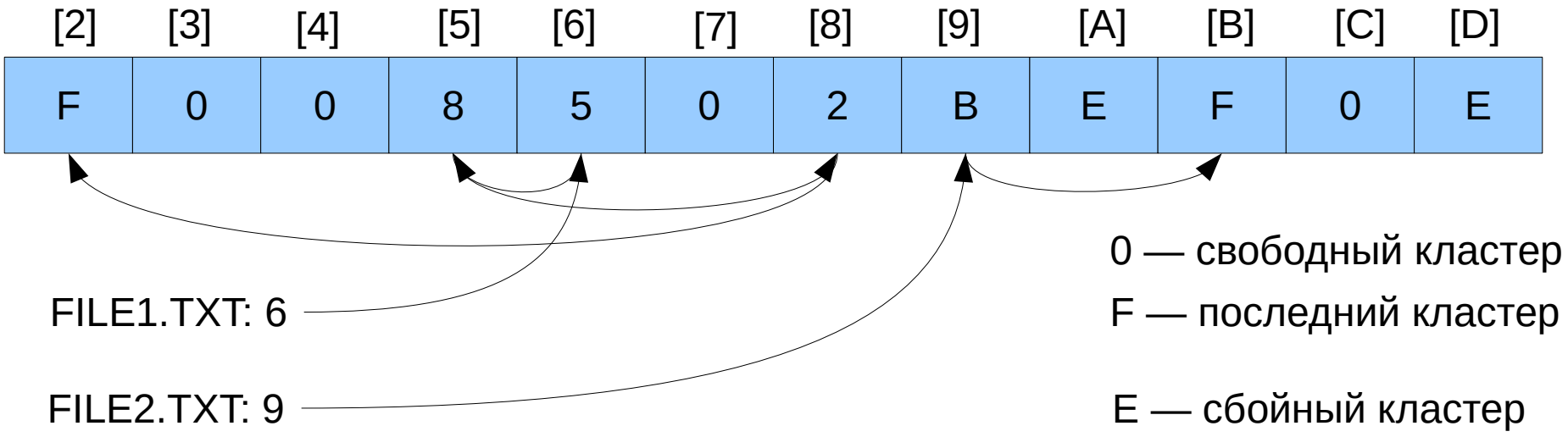
- Поддержка иерархии каталогов (иерархические/не иерархические)
- Поддержка журналирования
- Поддержка особенностей хранения данных (dvd, flash)

Файловая система RT-11

- Одноуровневая иерархия файлов
- Файлы хранятся в непрерывных областях области данных диска
- Имена файлов — 6 + 3 заглавные латинские буквы и цифры (кодируется в 6 байтах)
- Записи о файлах в каталоге диска располагаются в порядке размещения файлов в области данных диска, специальные записи для «дыр»

FAT

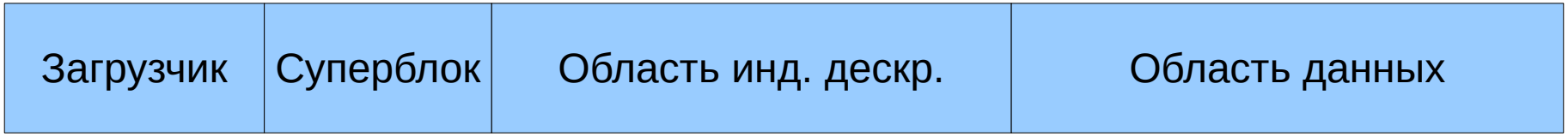
- Иерархическая файловая система
- Имена файлов: 8 + 3 (символы занимают один байт)
- Таблица размещения файлов:



FAT

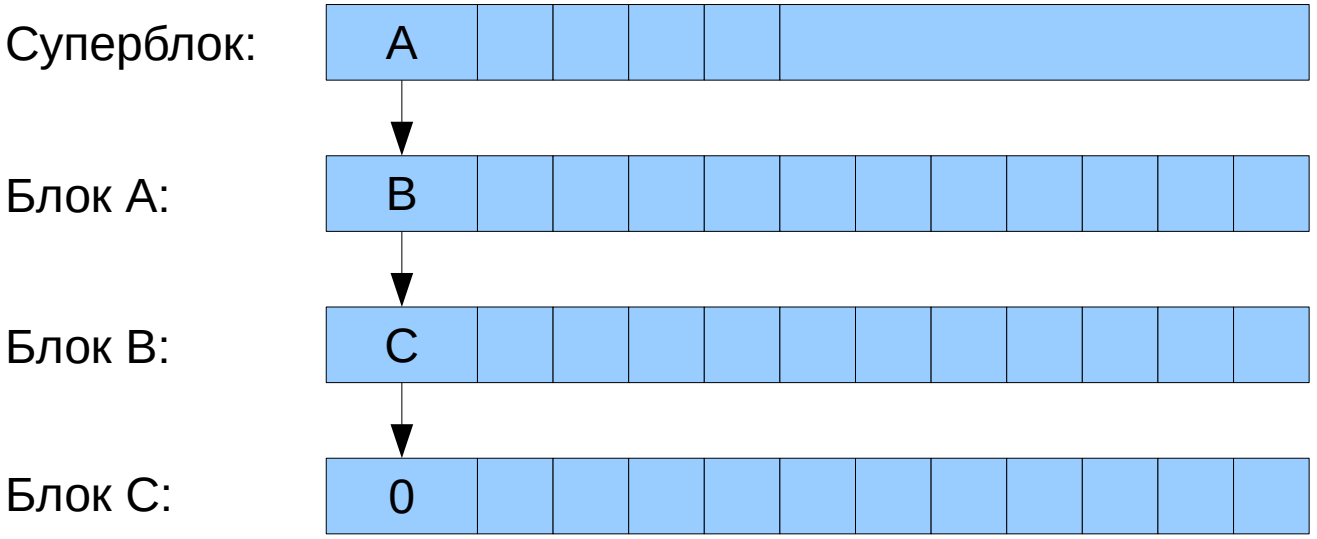
- Метаинформация о файле хранится в записях каталога
- FAT16: Максимальный размер кластера — 32 KiB (при размере блока 512 байт — 64 блока на кластер)
- Для надежности на диске хранится две копии FAT
- Для эффективной работы в памяти приходится держать FAT целиком
- Фрагментация файлов

UNIX System V FS (s5fs)



- Суперблок хранит информацию о файловой системе:
  - Размер файловой системы в блоках
  - Размер области индексных дескрипторов (inode) в блоках
  - Число свободных блоков и инд. дескр.
  - Номер первого свободного инд. дескр.
  - Список свободных блоков данных (частично)
- Загружается в память при монтировании

Список свободных блоков



- При удалении блока он добавляется в начало списка
- При выделении блока он берется из начала списка

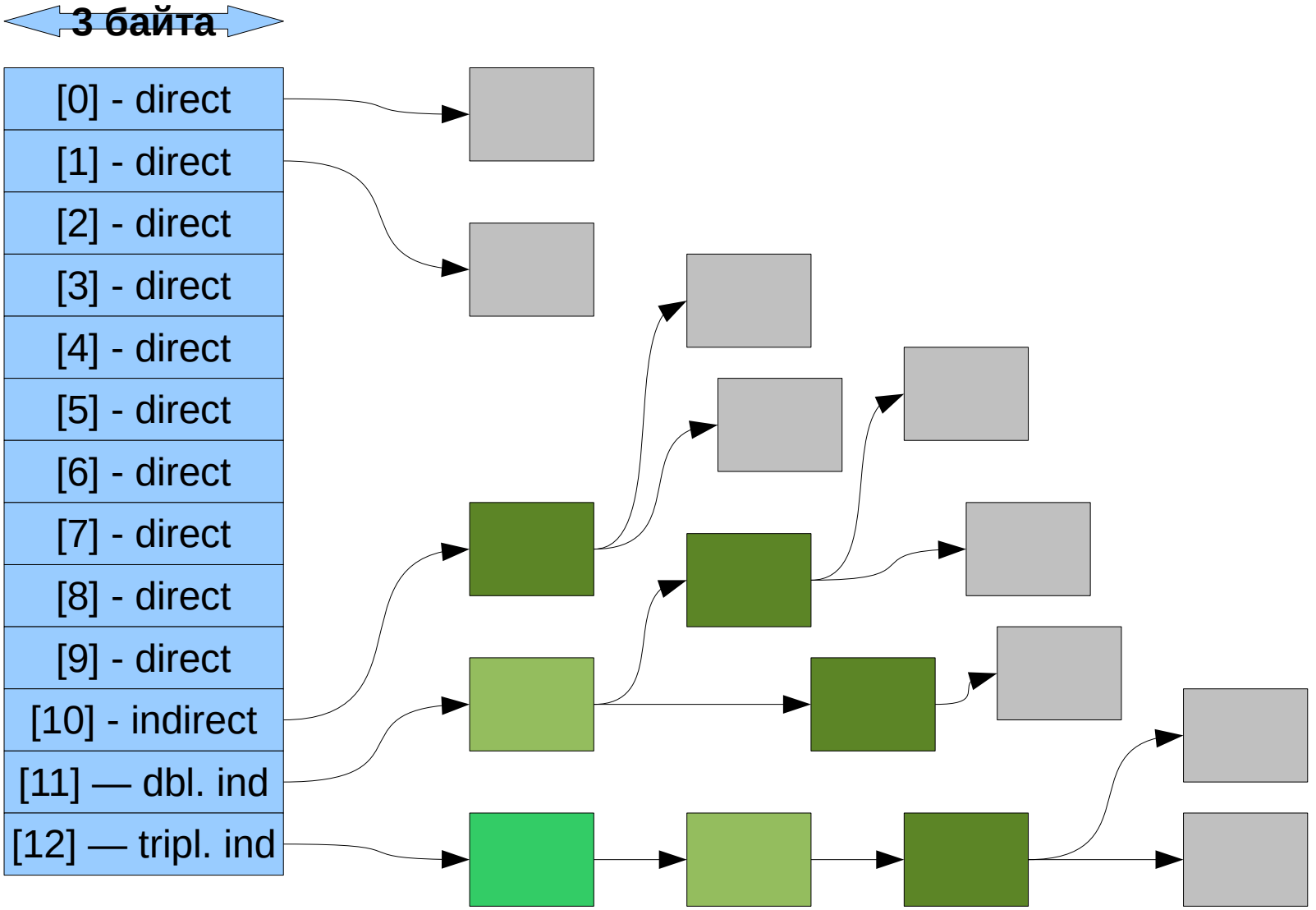
Индексный дескриптор (inode)

Поле	Размер	Описание
di_mode	2	Права доступа и тип файла
di_nlinks	2	Число ссылок на этот и. д.
di_uid	2	Идентификатор пользовател
di_gid	2	Идентификатор группы
di_size	4	Размер файла
di_addr	39	Массив адресов блоков данных
di_gen	1	Поколение
di_atime	4	Время посл. доступа к файлу
di_mtime	4	Время модификации файла
di_ctime	4	Время создания файла

Размер — 64 байта

Индексный дескриптор в памяти содержит дополнительные поля!

Массив адресов блоков



Размеры файлов

- Размер блока — 512 байт
- Один номер блока – 3 байта
- 10 непоср. номеров блоков — 5 KiB
- Номер косв. Блока – 512/3 = 170 номеров блоков
  - Итого: 10 + 170 = 180 блоков
- Номер двойного косв. блока - 170^2 блоков
  - Итого: 10 + 170 + 170^2 = ~14 MiB
- Номер тройного косв. блока - 170^3 блоков
  - Итого: 10 + 170 + 170^2 + 170^3 = ~2.5GiB
- Максимальный размер файла в системных вызовах – INT_MAX = 2GiB

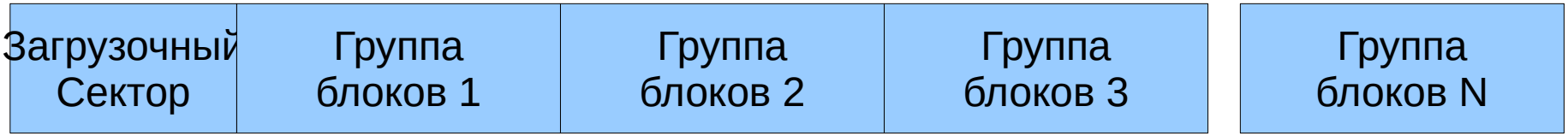
Структура каталога

- Каталог — файл, содержащий список файлов и каталогов
- Каждая запись в каталоге — 16 байт
  - Имя файла — 14 байтов
  - Номер индексного дескриптора — 2 байта

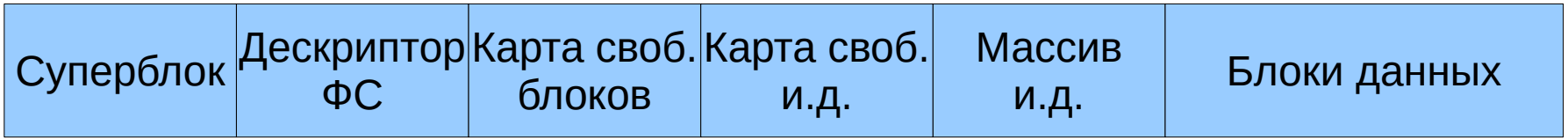
Недостатки

- Суперблок может быть поврежден
- Размер блока недостаточный (низкая скорость передачи)
- Блоки файлов и каталогов разбросаны по диску
- Индексные дескрипторы находятся далеко от блоков данных

Файловая система Ext2 (Linux)



Группа блоков:



- Размер блока данных: 1024, 2048, 4096 байт
- Номера индексных дескрипторов и блоков — 32 битные беззнаковые
- Размер индексного дескриптора — 128 байт
- Запись в каталоге имеет переменный размер (до 256 с)

Ext2 (1993)

- В inode хранится 12 прямых ссылок на блоки, indirect, double indirect, triple indirect (каждая — 32 бита)
- Если длина symlink < 60 байт, то он хранится в inode
- Свободные блоки хранятся в битовом множестве

Журналирование

- Обеспечение целостности файловой системы в случае краха ОС или сбоя питания
- Журнал — специальная область на диске
- Каждая операция, модифицирующая данные, выполняется в три стадии:
  - В журнал записывается операция (с флагом невыполненной)
  - Выполняется операция
  - Операция в журнале помечается как выполненная

Ext3 (2001)

- Совместима снизу вверх с ext2
- Обеспечивает журналирование
- Индексирование больших каталогов (htree)
- Максимальный размер файла увеличен до 2 TiB = 2 * 1024 GiB

Ext4 (2008)

- Макс. размер файловой системы до 2^60
- Макс. размер файла — 16TiB
- Extends вместо отображения блоков
  - До 4 на каждый inode, 128 MiB каждый
  - Оптимизация размещения на диске больших файлов
- Отметки времени с точностью до наносекунд
- 34 бита на секундную часть времени

Управление памятью

Виртуальная адресация

- Для многопроцессной обработки требуется защита памяти: процесс не должен иметь неавторизованный доступ к памяти других процессов и ядра
- Адреса ячеек памяти данных и программы, используемые в процессе, не обязаны совпадать с адресами в физической памяти (ОЗУ)
- Адреса ячеек памяти для процесса — **виртуальные адреса**
- Адреса ячеек памяти в оперативной памяти — **физические адреса**

Виртуальная адресация (память)

- Программно-аппаратный механизм трансляции виртуальных адресов в физические
- Аппаратная часть — отображение виртуальных адресов в физические в «обычной» ситуации — должно быть очень быстрой, так как необходимо для выполнения каждой инструкции
- Программная часть — подготовка отображения к работе, обработка исключительных ситуаций

Модели виртуальной адресации

- Модель база+смещение
  - Два регистра для процесса: регистр базы (B), регистр размера (Z)
  - Пусть V — виртуальный адрес (беззнаковое значение), если V >= Z — ошибка доступа к памяти, иначе
  - P — физический адрес, P = B + V

Сегментная адресация

- Каждый процесс состоит из нескольких сегментов: сегмент кода, сегмент стека, сегмент данных1, сегмент данных2
- Для каждого сегмента хранятся свои базовый адрес и размер
- У каждого сегмента свои права доступа, например:
  - Код: чтение + выполнение
  - Стек: чтение + запись
- Сегмент может отсутствовать в оперативной памяти и подгружаться по требованию

Страничная адресация

- Все пространство виртуальных адресов разбивается на страницы **равного размера**
- Каждая страница виртуальной памяти отображается на физическую память независимо от других
- Каждая страница имеет права доступа независимо от других страниц
- Страница может быть отмечена как неотображенная или отсутствующая в памяти
- При невозможности аппаратно отобразить виртуальную страницу в физическую — Page Fault

Отображение страниц

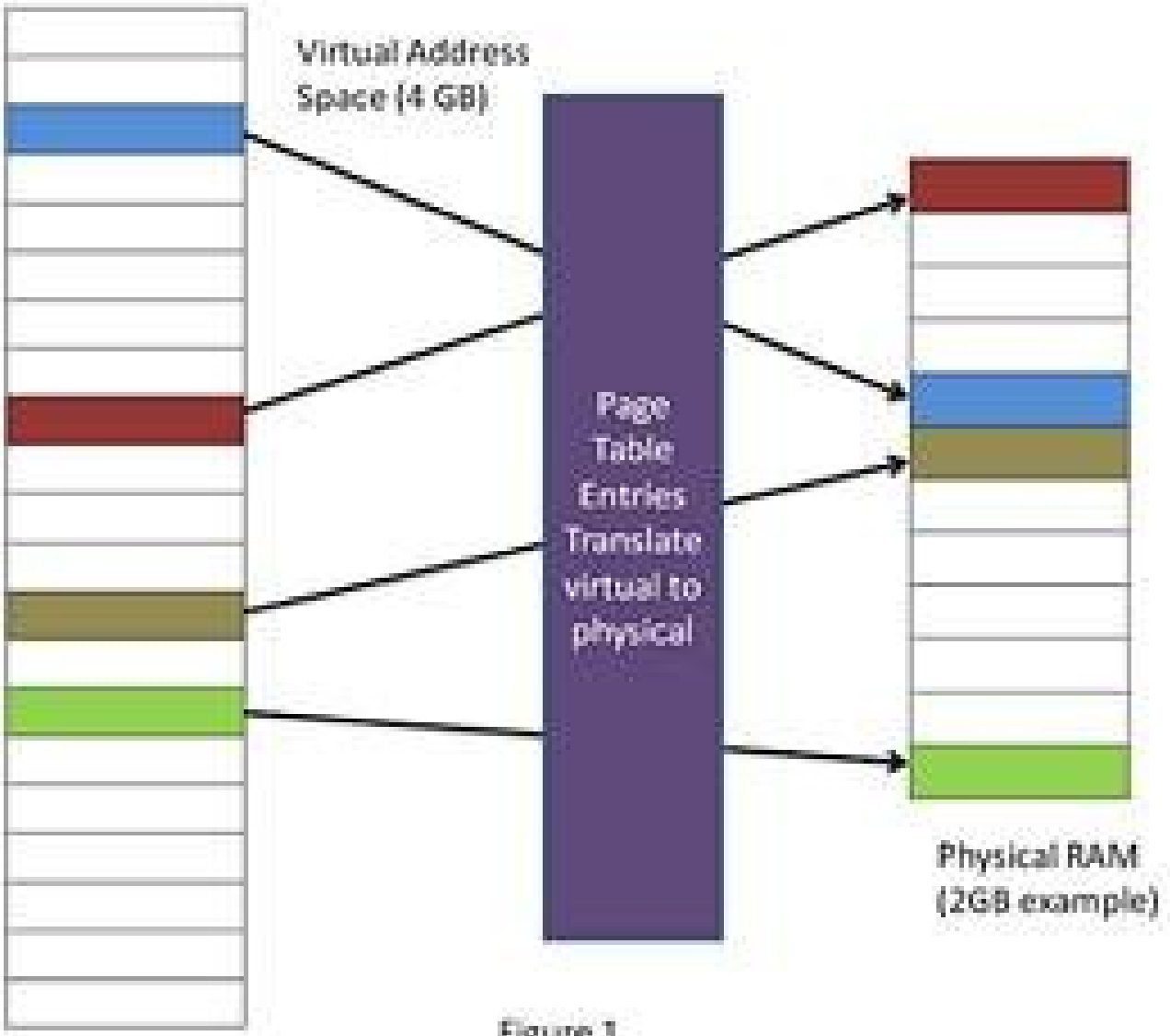
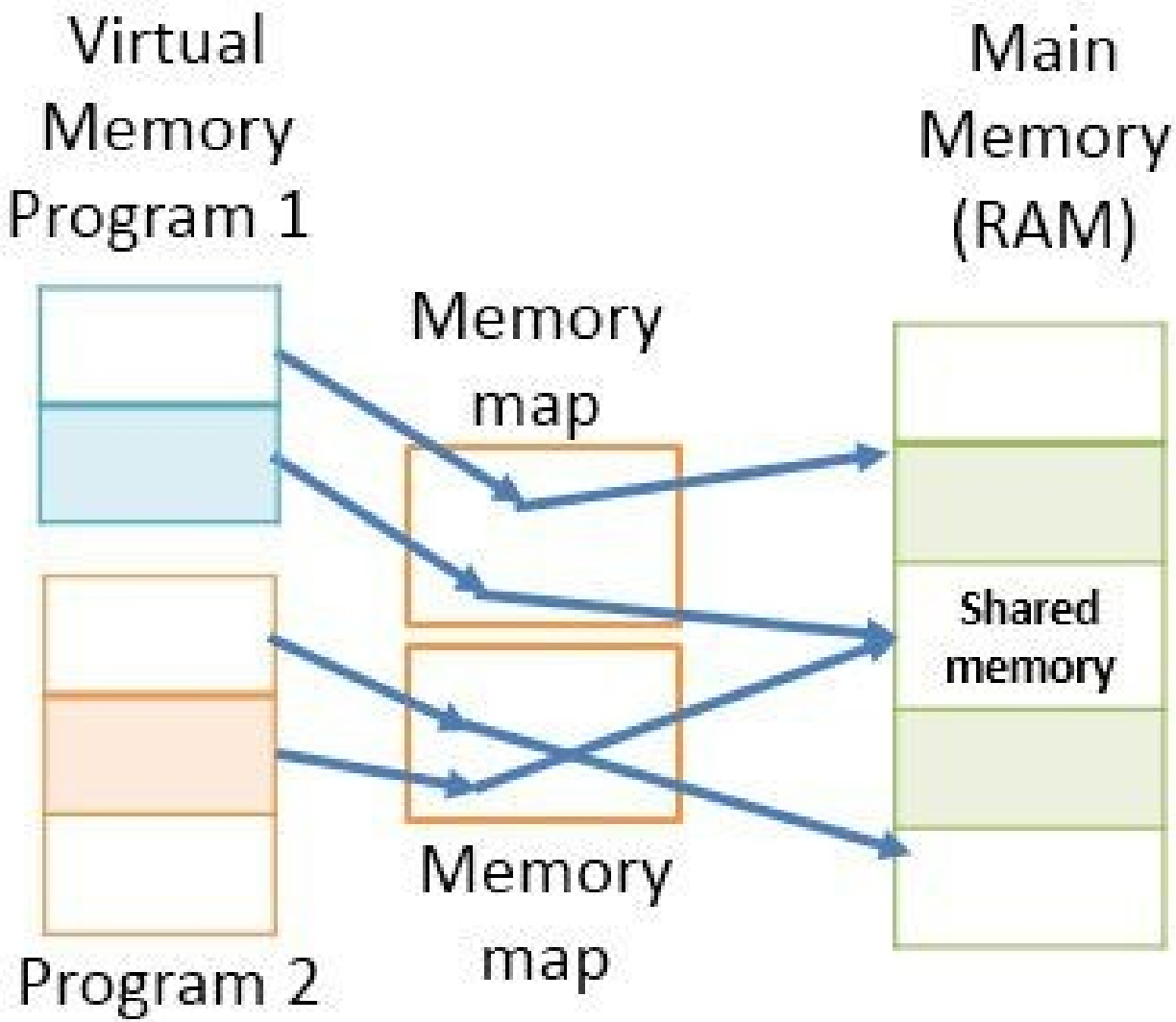
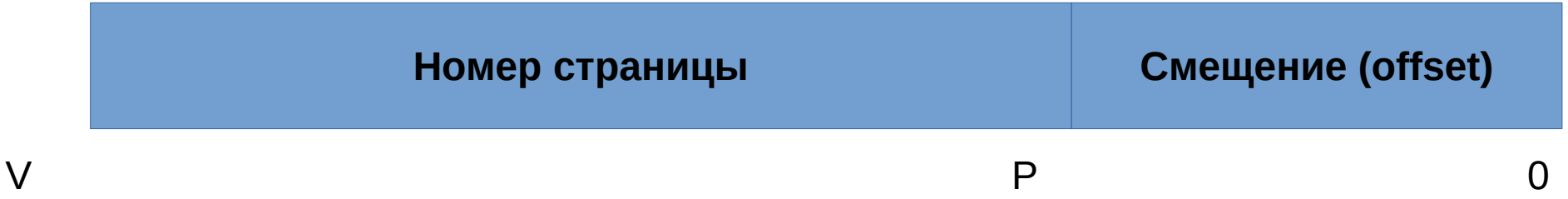


Figure 1

Разделяемые страницы



Виртуальный адрес



- V — количество бит виртуального адреса
- P — количество бит на смещение в страницу
- (V - P) — количество бит на номер страницы
- Для x86: V = 32, P = 12, V-P = 20
  - Виртуальное адресное пространство 4GiB
  - Размер страницы — 4096 байт (4KiB)
  - 2^20 (~1 Mi виртуальных страниц)

Двухуровневая таблица страниц (x86)

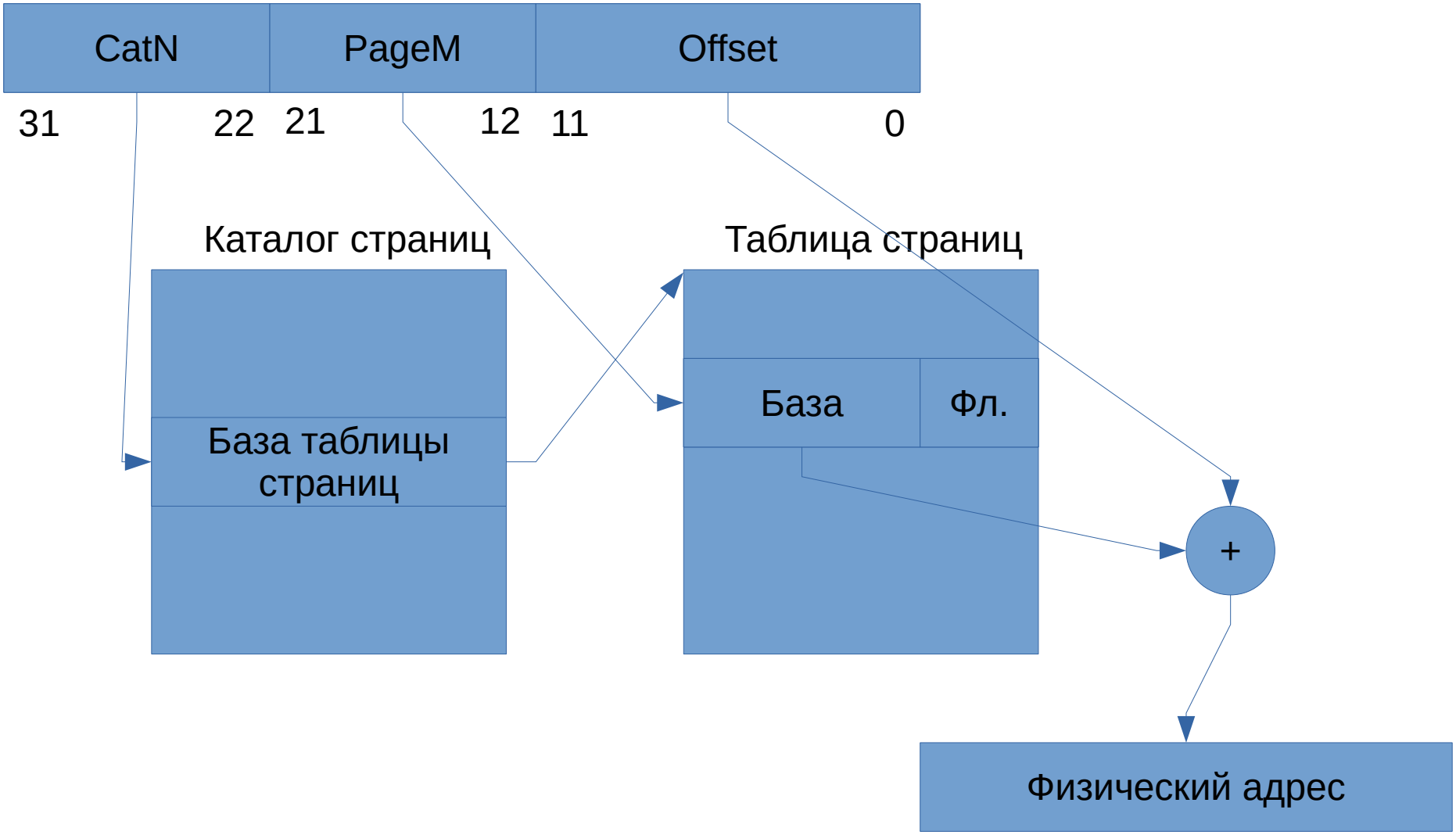


Таблица страниц

- Регистр процессора CR3 указывает на начало каталога страниц
- X86 — двухуровневая таблица страниц, размер страницы — 4KiB, в каталоге страниц 1024 записи, в каждой таблице страниц 1024 записи, одна запись — 4 байта
- X64 — четырехуровневая таблица страниц, размер страницы — 4KiB, в таблице каждого уровня 512 записей, одна запись — 8 байт.

Элемент таблицы страниц (x86)

- P — страница присутствует в ОЗУ
- R — право на запись в страницу
- U — доступна из user-space
- C — кеширование страницы запрещено
- W — разрешена сквозная (write-through) запись
- A — к странице было обращение
- D — (dirty) страница была модифицирована
- G — страница глобальная

Трансляция адресов x86

```
#define PAGE_SIZE  4096
#define TABLE_SIZE 1024
unsigned translate(unsigned va)
{
  unsigned *catalog = CR3;
  unsigned *table = catalog[va >> 22] & -PAGE_SIZE;
  unsigned phys = table[(va >> 12) & (TABLE_SIZE - 1)]
& -PAGE_SIZE;
  return phys + (va & (PAGE_SIZE - 1));
}
```

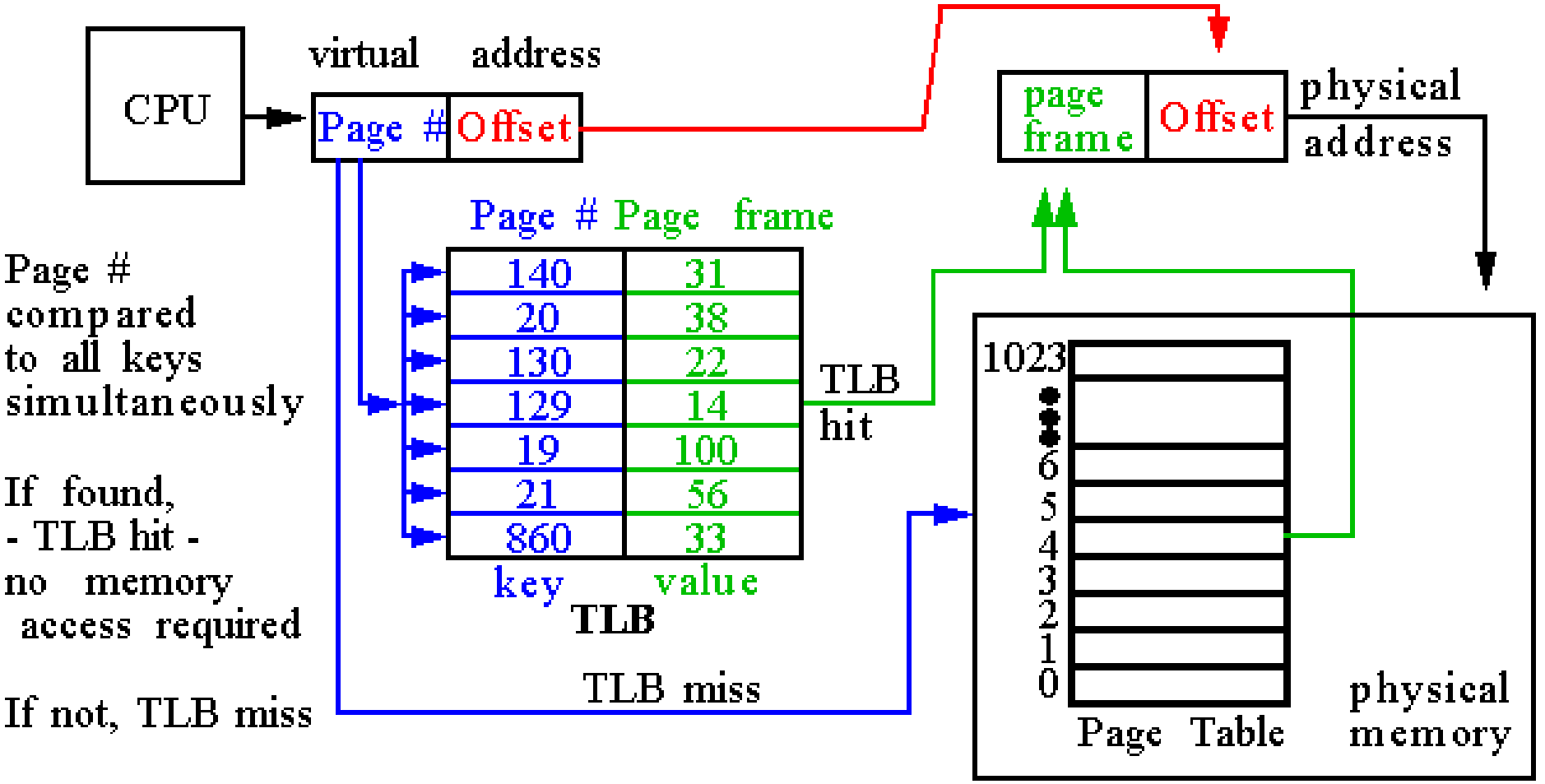
Доступ к странице

- Если страница отсутствует в ОЗУ (P == 0), обращение к странице → Page Fault
- Если в user-space и U == 0 → Page Fault
- Если записываем в страницу и R == 0 → Page Fault
- Устанавливаем флаг «accessed» (A = 1)
- Если записываем, устанавливаем флаг «dirty» (D = 1)

TLB (Translation Lookaside Buffer)

- Двухуровневая таблица страниц может потребовать 2 вспомогательных обращения к памяти (а 4-уровневая – 4!!!)
- TLB — кэш-память для отображения виртуального адреса в физический
- TLB может быть многоуровневым и разделенным: для Intel Nehalem:
  - 64 записи в L1 DTLB
  - 128 записей в L1 ITLB
  - 512 записей в L2 TLB

TLB


CPU
virtual address
Page # Offset
page frame
Offset
physical address
Page # Page frame
140 31
20 38
130 22
129 14
19 100
21 56
860 33
key value
TLB
TLB hit
Page # compared to all keys simultaneously
If found, - TLB hit - no memory access required
If not, TLB miss
TLB miss
1023
6
5
4
3
2
1
0
Page Table
physical memory

PageFault

- Исключение PageFault не обязательно ошибка в программе
- Допустимые ситуации:
  - Страница данных откачана в swap (P == 0)
  - Страница кода не загружена из файла (P == 0)
  - Запись в страницу созданную для copy-on-write (R == 0)
- Обработчик исключения определяет причину PageFault. Если PageFault произошел из-за ошибки, ошибка передается в программу

Итог: зачем нужна страничная виртуальная память

- Каждый процесс (выполняемая программа) имеет свое виртуальное адресное пространство – упрощение управлением памятью на уровне процесса (стек, куча, нити)
- Права доступа к памяти могут гибко настраиваться (read, write, execute)
- Процессы изолированы друг от друга
- Но несколько процессов могут разделять (использовать) одну и ту же страницу физического ОЗУ (shared pages)
- Программа (и вообще адресное пространство отдельного процесса) не обязана располагаться последовательно в физической памяти
- Виртуальной памяти может быть больше физической

Что почитать

- http://rus-linux.net/lib.php?name=/MyLDP/hard/memory/memory.html
-