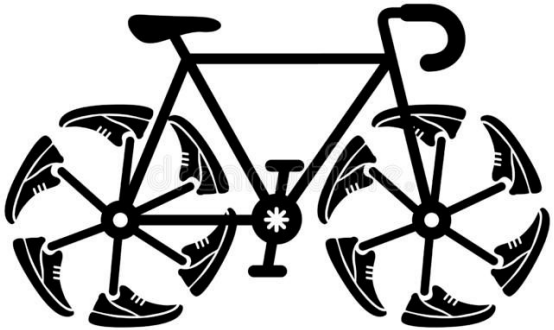
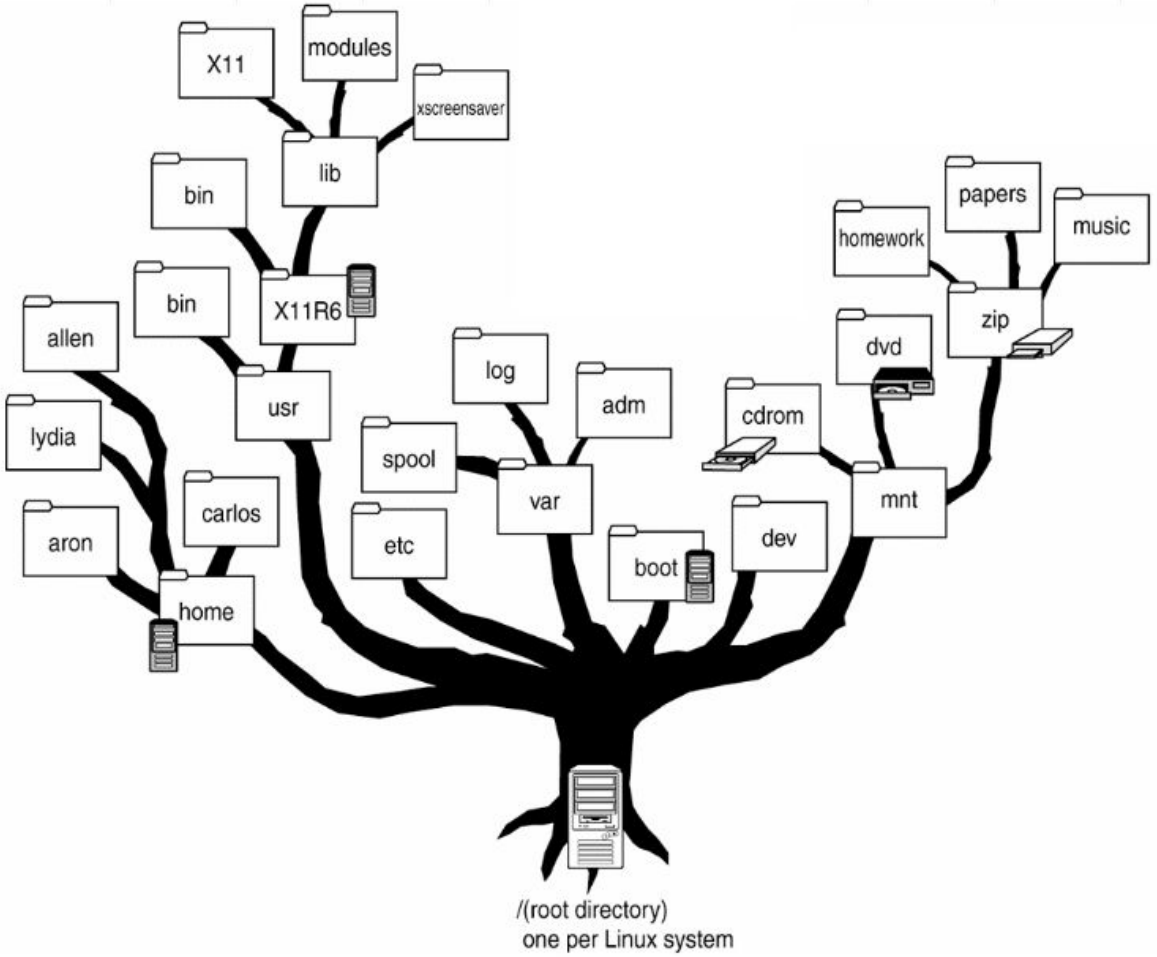
Программирование в Linux



Файлы и файловые системы


X11
modules
xscreensaver
lib
bin
bin
X11R6
allen
usr
lydia
carlos
aron
home
log
adm
spool
var
etc
boot
cdrom
dvd
mnt
dev
homework
papers
music
zip
/(root directory)
one per Linux system

Все есть файл

- Файл
- Каталог
- Сокеты
- Устройства
- Объекты ядра

Единый интерфейс на уровне системных вызовов

```
$ cat /proc/cpuinfo | head -10
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
stepping        : 9
microcode       : 0xd6
cpu MHz         : 2310.532
cache size      : 6144 KB
physical id     : 0
```

Системный вызов open

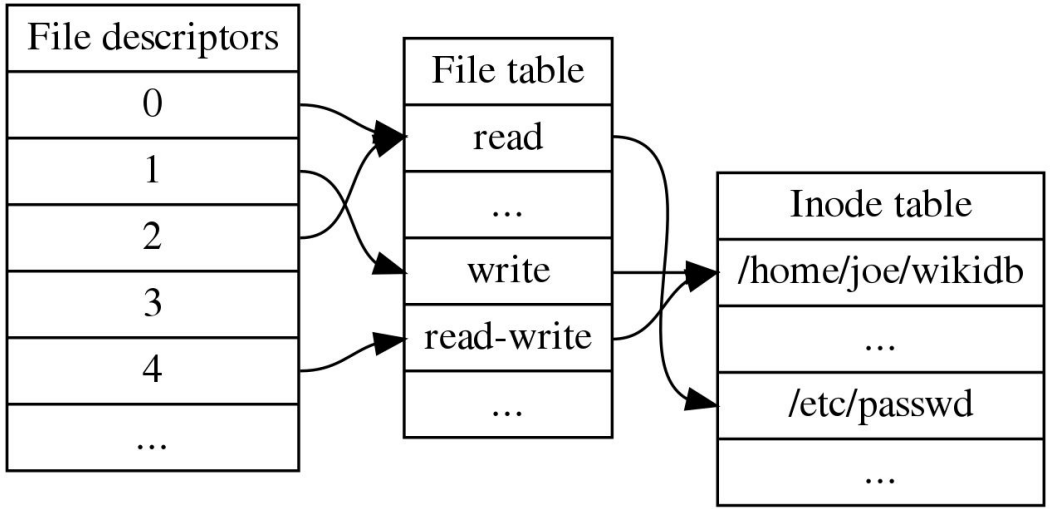
```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

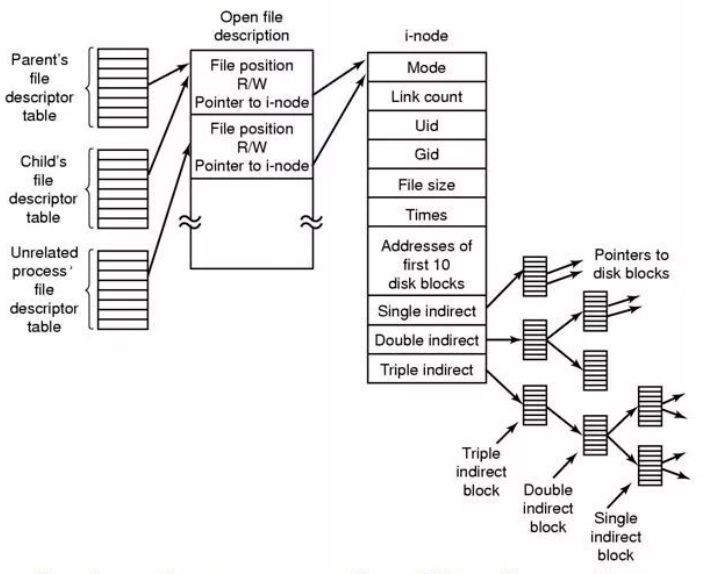
```
fd = open("/home/ds/file.txt", ...)
```

1. Открыть файл-каталог "/"
2. Найти в нем элемент home. Узнать, какой файловой системе он принадлежит
3. Открыть "/home". Найти в нем элемент "ds"
4. Открыть "/home/ds" Найти в нем file.txt — вернуть файловый дескриптор

- Операционная система поддерживает таблицу открытых файлов
- Счетчик ссылок открытых файлов (список процессов, использующих файл)
- Файловые дескрипторы уникальны внутри процесса
- На уровне fs файлы определяются через i-node



i-node (индексный дескриптор)

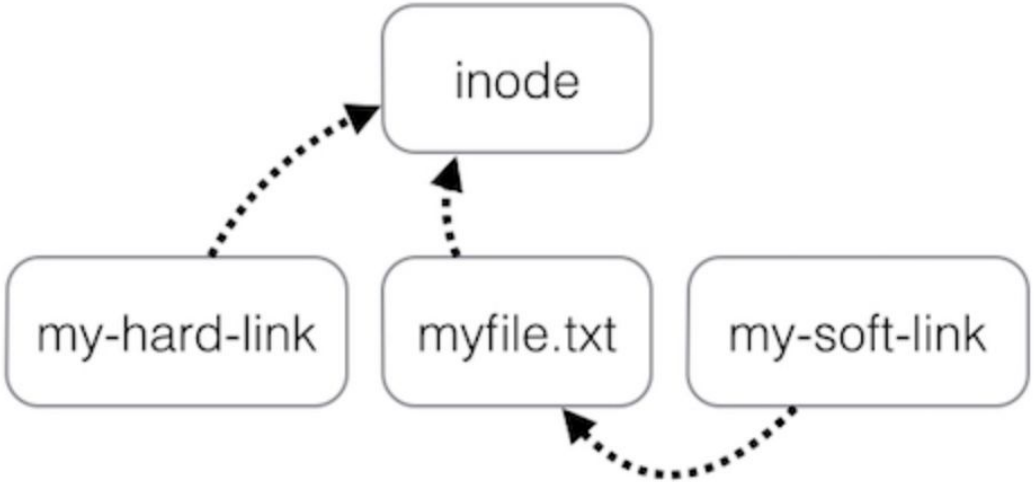


- Каждому файлу в fs соответствует inode
- inode содержит метаинформацию о файле
- inode косвенно указывает, где расположены данные на устройстве
- i-node не содержит имени файла — оно записано в файле-каталоге

Ссылки

Сырое содержимое файла-каталога — список элементов вида: <имя> <inode>

У одного и того же файла может быть несколько имен!
В i-node хранится счетчик “жестких” ссылок на нее
Жесткие ссылки не могут пересекать границы файловых систем
Мягкая ссылка — файл, хранящий имя другого файла



Информация о файле и типы файлов

```c
int main(int argc, char *argv[]){
    struct stat sb;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // fstat — с файловым дескруптором
    // stat  — с путем, но следует по симлинкам
    if (lstat(argv[1], &sb) == -1) {
        perror("lstat");
        exit(EXIT_FAILURE);
    }

    printf("ID of containing device:  [%lx,%lx]\n",
            (long) major(sb.st_dev), (long) minor(sb.st_dev));

    printf("File type:                ");

    switch (sb.st_mode & S_IFMT) {
        case S_IFBLK:  printf("block device\n");            break;
        case S_IFCHR:  printf("character device\n");        break;
        case S_IFDIR:  printf("directory\n");               break;
        case S_IFIFO:  printf("FIFO/pipe\n");               break;
        case S_IFLNK:  printf("symlink\n");                 break;
        case S_IFREG:  printf("regular file\n");            break;
        case S_IFSOCK: printf("socket\n");                  break;
        default:       printf("unknown?\n");                break;
    }

    printf("I-node number:            %ld\n", (long) sb.st_ino);
    printf("Mode:                     %lo (octal)\n",(unsigned long)sb.st_mode);

    printf("Link count:               %ld\n", (long) sb.st_nlink);
    printf("Ownership:                UID=%ld   GID=%ld\n",
            (long) sb.st_uid, (long) sb.st_gid);

    printf("Preferred I/O block size: %ld bytes\n",
            (long) sb.st_blksize);
    printf("File size:                %lld bytes\n",
            (long long) sb.st_size);
    printf("Blocks allocated:         %lld\n",
            (long long) sb.st_blocks);

    printf("Last status change:       %s", ctime(&sb.st_ctime));
    printf("Last file access:         %s", ctime(&sb.st_atime));
    printf("Last file modification:   %s", ctime(&sb.st_mtime));


    return EXIT_SUCCESS;
}
```

Что можно делать с "файлами"

- **open, openat, creat** - открыть (создать, если нет)
  - успешность зависит от выставленных флагов
  - для особых комбинаций флагов есть обертки
  - возвращают файловые дескрипторы
- **link, linkat, symlink, symlinkat -** создавать ссылки
- **read / write -** последовательности байт
- **lseek -** позиционирование
- **mmap/munmap -** отображать в память
- удалять - **unlink, unlinkat**
- **ioctl -** делать что-то, выходящее за рамки read/write

Все эти методы работают с файловыми дескрипторами.

Дескрипторы конечны и их надо закрывать -- **close**

```cpp
#include <sys/unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>

#define PATH_MAX 100

int main() {
    auto tmp_file_fd = open(".",
                            O_TMPFILE | O_RDWR,
                            S_IRUSR | S_IWUSR);

    if (tmp_file_fd < 0){
        perror("open");
    }

    auto file = fdopen(tmp_file_fd, "w");
    fprintf(file, "hello");

    char path[PATH_MAX] = {};
    snprintf(path, PATH_MAX,  "/proc/self/fd/%d", tmp_file_fd);

    if (linkat(AT_FDCWD, path, AT_FDCWD, "./hello.txt",
                                         AT_SYMLINK_FOLLOW)) {
        perror("linkat");
    }


    fclose(file);
    // код ниже всегда напечатает ошибку!
    // дескриптор уже закрыт!
    if (close(tmp_file_fd)) {
        perror("close");
    }
}
```

Файл - абстракция. А абстракции могут течь!

Не работает

```
...

#define PATH_MAX 100

int main() {
    auto dir_fd = open("/home/dmis",
                       O_DIRECTORY | O_RDONLY,
                       S_IRUSR);

    if (dir_fd < 0){
        perror("open");
    }

    auto file = fdopen(dir_fd, "r");

    if (!file){
        perror("fdopen");
        exit(-1);
    }

    char buffer[PATH_MAX + 1];
    fgets(buffer, PATH_MAX, file);

    if (errno){
        perror("fgets");
        exit(-1);
    }

    fclose(file);
    return EXIT_SUCCESS;
}
```

Работает

```
...
#include <dirent.h>
#include <iostream>

int main() {
    auto dir_fd = open("/home/dmis",
                       O_DIRECTORY | O_RDONLY,
                       S_IRUSR);

    if (dir_fd < 0){
        perror("open");
    }

    auto dir = fdopendir(dir_fd);

    if (!dir){
        perror("fdopendir");
        exit(-1);
    }

    while (auto dir_entry = readdir(dir)) {
        std::cout << dir_entry->d_name << "\n";
    }

    closedir(dir);
    return EXIT_SUCCESS;
}
```

Обработка ошибок

```c
#include <errno.h>
#include <string.h>

errno = 0; // функции почти никогда не очищают код!
//вызываем любую, потенциально падающую функцию
if (errno) {
  perror("text"); // пишет в stderr "text: текст ошибки"
  printf("%s", strerror(errno)); // пишет текст ошибки
}
```

Никогда не забывайте про обработку ошибок!

```c
auto child_pid = fork();

if (child_pid == 0) {
  // дочерний процесс что-то делает
} else {
  ...
  kill(child_pid, SIGTERM); // если fork не удался (-1),
  //                  будет broadcast по всем процессам в системе!
}
```