Системные вызовы для доступа к файлам

Основные понятия и действия

Это произведение доступно по лицензии
Creative Commons “Attribution-ShareAlike” (“Атрибуция — На тех же условиях”) 3.0 Непортированная.
http://creativecommons.org/licenses/by-sa/3.0/deed.ru

Основные действия

При работе с содержимым файла чаще всего выполняются следующие действия:

- **чтение** (read) — данные читаются из файла в заданный буфер;
  (*буфер* — область памяти, предназначенная для обмена с файлом)

Основные действия

При работе с содержимым файла чаще всего выполняются следующие действия:

- **чтение** (read) — данные читаются из файла в заданный буфер;
  (*буфер* — область памяти, предназначенная для обмена с файлом)
- **запись** (write) — данные записываются из буфера в файл;

Основные действия

При работе с содержимым файла чаще всего выполняются следующие действия:

- **чтение** (read) — данные читаются из файла в заданный буфер;
  (*буфер* — область памяти, предназначенная для обмена с файлом)
- **запись** (write) — данные записываются из буфера в файл;
- **перемещение текущей позиции** (lseek) — перемещается текущая позиция чтения-записи без передачи данных.

Как задать аргументы?

Представим условный прототип системного вызова read:

```
resultType read( откуда, куда, сколько );
```

Задать “**куда**” — просто, что указатель на область памяти, куда поместить прочитанные данные (например, “void *buf”).

“**Сколько**” — это просто число, показывающее, сколько байт мы просим прочитать (например, “unsigned size”).

Но как задать “**откуда**”? Будет ли разумно задавать в этом качестве **имя файла**?
Например, “const char *filename”.

Следствия использования имени файла для указания “куда”

Если задать при каждом системном вызове чтения имя файла, то операционной системе каждый раз придётся:

- искать этот файл в файловой системе;
- проверять права доступа;
- находить, где расположены блоки, выделенные данному файлу;
- определить каким-то образом, где мы читали в предыдущий раз;
- ...

Это неоправданно долго. Вывод — эти действия надо выполнять один раз **перед** использованием файла.

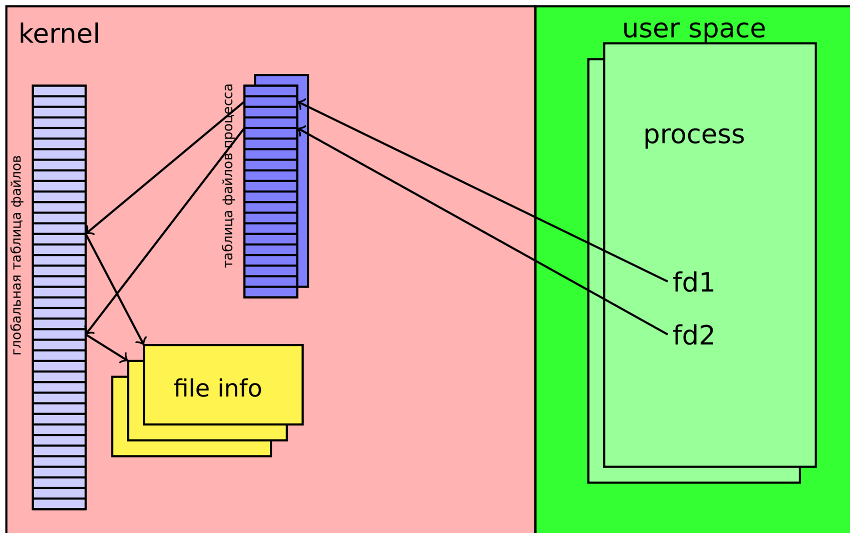
Эти действия называются “открытием файла”. И для выполнения этих действий используется системный вызов “**open**”.

Но кто будет хранить информацию об открытых файлах? Ядро или программа?

Если эта информация будет принадлежать программе пользователя, то пользователь сможет произвольно изменять эти данные — это недопустимо.

Поэтому информацию об открытых файлах хранит **ядро**.

Информация об открытых файлах



Файловый дескриптор

Все операции с файлом после открытия после его открытия совершаются с помощью файлового дескриптора — номера открытого файла в процессе. Обозначается **fd** — *file descriptor*.
Процесс может получить файловый дескриптор:

- открыв файл системным вызовом “open”;
- получив дескриптор от родителя.

Обычно программы получают от родителя 3 открытых файла с дескрипторами:

- 0 — stdin = **стандартный ввод**. Используется в getchar, scanf ...
- 1 — stdout = **стандартный вывод**. Используется в putchar, printf ...
- 2 — stderr = **стандартный вывод об ошибках**.

Системный вызов “open”
описание

Для открытия файла используется системный вызов “open” (man 2 open):

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

file — имя файла. Просто строка языка C. Может быть константной строкой (например `"myfile.txt"`), именем символьного массива, где эта стока содержится...
flags — битовый набор флагов. Указывает, что и как мы собираемся делать с файлом.
mode — права доступа к создаваемому файлу.

Возвращаемое значение:
-1 — признак ошибки (номер причины — в переменной `errno`), > 0 — **файловый дескриптор** (fd).

Системный вызов “open”
Флаги открытия

Режим открытия файла определяется набором бит в целочисленном аргументе **flags**. Один из трёх флагов должен быть указан обязательно:

- **O_RDONLY** — читать
- **O_WRONLY** — писать
- **O_RDWR** — читать и писать

Дополнительно могут быть указаны файлы

- **O_CREAT** — создать файл, если его нет;
- **O_EXCL** — выдать ошибку,если файл существует;
- **O_TRUNC** — удалить старое содержимое;
- **O_NOFOLLOW** — не следовать по символическим ссылкам;
- **O_APPEND** — дописывать в конец файла;
- . . .

Если файл в результате системного вызова “open” действительно создаётся, (задан флаг O_CREAT, файла не было и его удалось создать) то необходимо задать права доступа к этому файлу. Для этого используются младшие 9 бит аргумента “mode_t mode” (mode_t — какой-то целочисленный тип).
3 группы по 3 бита: (владелец)(группа)(остальные).
В каждой группе значение бит, начиная со старшего:
r (4) — разрешение читать,
w (2) — разрешение писать,
x (1) — разрешение выполнять.
Можно указывать в виде восьмеричного литерала, например 0640, 0755, 0600. А можно в виде набора флагов, например S_IRUSR | S_IWUSR | S_IRGRP.
Если не указан флаг O_CREAT, то третий аргумент можно не указывать.

Обработка ошибок

errno, strerror

Если произошла ошибка — open возвращает -1.

!!!

Пользователь должен иметь возможность узнать причину ошибки.

Код причины ошибки — в целочисленной переменной errno (описана в <errno.h>).

Преобразовать код в строку — функция
const char* strerror(int errnum)
(описана в <string.h>)

В консольных приложениях сообщения об ошибках следует выводить на stderr — **стандартный вывод ошибок**.

Пример использования системного вызова “open”

```
const char *fn = "hidden.data";
int fd1 = open( fn1, O_RDONLY );
if( fd1 < 0 ) {
  fprintf( stderr, "Fail to open file \"%s\%: %s\n",
           fn, strerror(errno) );
  exit(1);
}

int fd2 = open( "x.dat", O_WRONLY | O_CREAT | O_TRUNC,
                0640 );

int fd3 = open( "z.txt", O_WRONLY | O_CREAT | O_EXCL,
                 0600 );

int fd4 = open( "a b c", O_RDWR | O_APPEND );
```

На каждый открытый файл расходуются системные ресурсы. Когда файл больше не нужен — его надо **закрыть**.

```
int close( int fd );
```

Аргумент:
fd — файловый дескриптор файла, который надо закрыть.

Возвращает:
0 = Ок
-1 — ошибка, причина — в errno.

При завершении процесса все его открытые файлы закрываются автоматически.

Для чтения из файла используется системный вызов “read”:

```
ssize_t read(int fd, void *buf, size_t len);
```

Аргументы:
fd — из какого файла (дескриптор)
buf — куда (указатель на область памяти)
len — сколько байт.
size_t — беззнаковое целое, например unsigned, unsigned long.
size_t — знаковое целое, например int, long.

Возвращает
-1 — ошибка (причина — в errno);
0 — конец файла (EOF);
>0 — сколько байт прочитал (**может быть <len**).

Пример чтения из файла

```
#define BUFSZ 1024
char buf[BUFSZ];
int fdr = open( "my_file.txt", O_RDONLY );
if( fdr == -1 ) { handle_error(); }

int r = read( fdr, buf, BUFSZ );
if( r < 0 ) {
  fprintf( stderr, "Fail␣to␣read␣file␣my_file.txt:␣%s\n",
           strerror(errno) );
  return 2;
}
if( r == 0 ) {
  printf( "EOF␣detected!\n" );
  return 0;
}
printf( "Read␣%d␣bytes,␣first␣is␣%02X\n", r, r[0] );
```

Для записи в файл используется системный вызов “write”:

```
ssize_t write(int fd, const void *buf, size_t len);
```

Аргументы:
fd — в какой файла (дескриптор)
buf — откуда (указатель на область памяти)
len — сколько байт.

Возвращает
−1 — ошибка;
0 — ошибки нет, но записано 0 байт;
>0 — сколько байт записал (**может быть <len**).

Пример записи в файл

```
// ... many checks here ...
int fdw = open( argv[1], O_WRONLY | O_CREAT, 0644 );
if( fdw < 0 ) { handle_error(); }

int w = write( fdw, "ABCDE", 4 );
if( w < 0 ) {
  fprintf( stderr, "Fail␣to␣write␣to␣file␣\"%s\":␣%s\n",
           argv[1], strerror(errno) );
  return 2;
}
if( w == 0 ) {
  printf( "Fail␣to␣write␣w/o␣error???\n" );
  return 0;
}
printf( "Written␣%d␣bytes\n", w );
```

Для перемещения текущей позиции чтения/записи используется системный вызов “lseek”:

```
off_f lseek(int fd, off_t offset, int whence);
```

Аргументы:
fd — в каком файле (дескриптор)
offset — смещение — на сколько байт (знаковое)
whence — относительно какой точки смещаться.
SEEK_SET — от начала файла;
SEEK_END — от конца;
SEEK_CUR — от текущей позиции

Возвращает
-1 — ошибка; ≥ 0 — смещение от начала файла.

На 64-битной операционной системе тип “off_t” — всегда 64-х битное целое. Допускает перемещение на $\pm 2^{63} = 9223372036854775808$ байт = 8589934592 GB.

На 32-битной операционной системе по умолчанию тип “off_t” — 32-х битное целое. Допускает перемещение на $\pm 2^{31} = 2147483648$ байт = 2 GB.

Для использования в этом случае 64-х бит:

1. Определить макрос препроцессора _FILE_OFFSET_BITS как 64, и “off_t” станет 64-х битным целым, и будет использоваться системный вызов “lseek64” под именем “lseek”.

2. Или определить макрос препроцессора _LARGEFILE64_SOURCE, использовать тип “off64_t” и системный вызов “lseek64”.

С помощью “lseek” можно переместится за конец файла (но не за начало). При этом:

Попытка чтения приведёт к состоянию “конец файла”.

Запись будет произведена как обычно, а в пропущенном пространстве будет “дыра”.

При чтении из “дыры” будут прочитаны символы с кодом ‘0’ (двоичные нули).

Место для файла выделяется только под реально записанные блоки.

Для получения информации о файле используются разновидности системного вызова **stat**:

```
int  stat( const char *name, struct stat *st );
int lstat( const char *name, struct stat *st );
int fstat( int fd , struct stat *st );
```

name — имя файла;
fd — файловый дескриптор
st — указатель на структуру типа stat, в которую записывается информация о файле.

Возвращает: 0 — Ok −1 — error.

stat и **lstat** отличаются только тогда, когда файл – символическая ссылка. Тогда **lstat** возвращает информацию о ссылке, а **stat** — о файле, ка который ссылка ссылается.

Структура “stat”

Структура stat описана там же, где и сам stat:

```
#include <sys/stat.h> // в нём:
struct stat {
  dev_t st_dev;      // устройство (major, minor)
  ino_t st_ino;      // номер индексного дескриптора
  nlink_t st_nlinks;// колво- имен файла
  dev_t st_rdev;     // описываемое устройство
  off_t st_size; // размер в байтах (off_t- 2 типа)
  blksize_t st_blksize; // размер блока в байтах
  blkcnt_t st_blocks;   // колво- выделенных блоков
  uid_t st_uid;          // id владельца
  gid_t st_gid;         // id группы
  mode_t st_mode;       // тип и права доступа
  time_t st_atime, st_mtime, st_ctime ; // времена
     // последнего доступа, изменения,
     // и изменения индексного дескриптора
}; //  и могут быть еще поля
```

Системный вызов "stat"

Поле "st_mode" структуры "stat"

```
mode_t st_mode;        // тип и права доступа
```

Поле "st_mode" структуры "stat" состоит из двух частей:

- младшие 12 бит — права доступа к файлу. Выделение этих бит — st.st_mode & 07777. Для проверки можно использовать специальные макросы для отдельных бит и групп, например S_IRWXU — все биты владельца файла, S_IXGRP — бит разрешения выполнения для группы, S_IROTH — бит разрешения чтения для остальных ...
- старшие биты — тип файла. Набор бит, зависимых от платформы.

Пример:

```
if( (st.st_mode & 022) ) {
   printf( "Group or other can write!\n" );
}
```

Для проверки типа файла следует использовать следующие макросы:

- **S_ISREG(mode)** — истина, если обычный файл
- **S_ISDIR(mode)** — истина, если каталог
- **S_ISLNK(mode)** — истина, если символическая ссылка
- **S_ISBLK(mode)** — истина, если блочное устройство
- **S_ISCHR(mode)** — истина, если символьное устройство.
- **S_ISSOCK(mode)** — истина, если сокет
- **S_ISPIPE(mode)** — истина, если именованный канал

Пример:

```
if( S_ISDIR(st.st_mode) ) {
  printf( "Is a directory\n" );
}
```

Системный вызов “stat”

Поля “st_atime” “st_mtime” “st_ctime” структуры “stat”

```
time_t st_atime; // время последнего доступа
time_t st_mtime; // время последнего изменения
time_t st_ctime; // время изменения inode
```

Время в “time_t” хранится в секундах с начала эпохи (1 января 1970 года).

Простейший перевод в строку:

```
char *ctime(const time_t *timep); // из <time.h>

printf( "Modification time is %s\n",
        ctime(st.st_mtime) );
```

Перевод “time_t”в структуру “tm”:

```
struct tm* localtime( const time_t *t ); // <time.h>
struct tm {
  int tm_sec;    // секунды
  int tm_min;    // минуты
  int tm_hour;   // часы
  int tm_mday;   // день месяца 1-31
  int tm_mon;    // месяц 0-11
  int tm_year;   // год-1900
  int tm_wday;   // день недели, Вск0=
  int tm_yday;   // день в году 0-365
  int tm_isdst;  // флаг летнего времени
};

struct tm *t = localtime( st.st_atime );
printf( "mtime: %d:%d:%d (mtime_file_atu_example)\n",
         t->hour, t->min, t->sec );
```

Для удобного и правильного форматирования даты и времени следует использовать функцию “strftime”:

```
size_t strftime( char *s, size_t max,
                 const char *format,
                 const struct tm *t );
```

s — буфер, в который будет записана строка;
max — размер буфера;
format — строка формата со спецификаторами;
t — указатель на структуру с датой и временем.

Спецификаторы формата выглядят как в “printf”, но с другим смыслом, например; “%B” — полное название месяца, “%d” — день в месяце, . . .

Один из самых полезных – “%c” — дата и время по локальным стандартам.

Для получения информации о файле из командной строки используется команда “stat”

```
stat [OPTION]... FILE...
```

По умолчанию показывает всю информацию о файле/файлах, например

```
stat /usr /etc/group /dev/sda1
```

Можно выбрать конкретную информацию:

```
stat -c '%s' /etc/group
```

Маскировка бит доступа создаваемых файлов

umask

На самом деле, при создании новых файлов третий аргумент системного вызова “open” используется не напрямую. Из переданного значения **сбрасываются** те биты, которые **установлены** в *umask*. Для управления *umask* есть одноимённый системный вызов:

```
mode_t umask(mode_t mask);
```

Получает новое значение *umask*, а возвращает старое.

Например, если третий аргумент сист. вызова “open” был 0666, а umask — 0027, то у файла будут биты доступа 0640.
Есть одноимённая команда — **umask**.