

Низкоуровневый ввод-вывод, часть 5

Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение,
2019

Содержание лекции

- 1 Файловый ввод-вывод
 - Открытие файлов
 - Чтение данных
 - Запись данных
- 2 Синхронный ввод-вывод
- 3 Закрытие файлов
- 4 Позиционирование, усечение

Файловый ввод-вывод

- Ядро поддерживает попроцессный список открытых файлов, называемый файловой таблицей.
- Таблицы индексируется с помощью неотрицательных целых чисел, называемых файловыми дескрипторами (часто они именуются сокращенно `fd`).
- Каждая запись в списке содержит информацию об открытом файле, в частности указатель на хранимую в памяти копию файлового дескриптора и ассоциированных с ним метаданных.
- К метаданным относятся, в частности, файловая позиция и режимы доступа.

Стандартные дескрипторы

- Каждый процесс традиционно имеет не менее трех открытых файловых дескрипторов: 0, 1 и 2.
- Файловый дескриптор 0 соответствует стандартному вводу (`stdin`), дескриптор 1 — стандартному выводу (`stdout`), дескриптор 2 — стандартной ошибке (`stderr`).
- Библиотека C не ссылается непосредственно на эти целые числа, а предоставляет препроцессорные определения `STDIN_FILENO`, `STDOUT_FILENO` и `STDERR_FILENO` для каждого из вышеописанных вариантов соответственно.
- Как правило, `stdin` подключен к терминальному устройству ввода (обычно это пользовательская клавиатура), а `stdout` и `stderr` — к дисплею терминала.

Системный вызов open()

Открытие файла и получение файлового дескриптора осуществляются с помощью системного вызова open():

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

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

Системный вызов open() ассоциирует файл, на который указывает имя пути name с файловым дескриптором, возвращаемым в случае успеха. В качестве файловой позиции указывается его начало (нуль), и файл открывается для доступа в соответствии с заданными флагами (параметр flags).

Флаги для открытия файла

- Аргумент `flags` — это поразрядное ИЛИ, состоящее из одного или нескольких флагов, определяющее режим доступа к файлу.
- Режим доступа может иметь одно из следующих значений: `O_RDONLY`, `O_WRONLY` или `O_RDWR`.

```
int fd;  
  
fd = open ("/home/kidd/madagascar", O_RDONLY);  
if (fd == -1)  
    /* ошибка */
```

- Если файл открыт только для чтения, в него невозможно что-либо записать, и наоборот.
- Процесс, осуществляющий системный вызов `open()`, должен иметь права, чтобы получить запрашиваемый доступ.

Открытие файла

Некоторые дополнительные флаги аргумента `flags` для изменения поведения `open()`.

- `O_APPEND`. Режим дозаписи - перед каждым актом записи файловая позиция будет обновляться и устанавливаться в текущий конец файла.
- `O_CREAT`. Если файл, обозначаемый именем `name`, не существует, то ядро создаст его.
- `O_TRUNC`. Если файл уже существует, является обычным файлом и заданные для него флаги допускают запись, то файл будет усечен до нулевой длины.

Открытие файла

Дополнительные флаги аргумента `flags` для изменения поведения `open()`.

- `O_ASYNC`. Когда указанный файл станет доступным для чтения или записи, генерируется специальный сигнал (по умолчанию `SIGIO`). Этот флаг может использоваться только при работе с `FIFO`, каналами, сокетами и терминалами, но не с обычными файлами.
- `O_SYNC`. Файл будет открыт для синхронного вводавывода. Никакие операции записи не завершатся, пока данные физически не окажутся на диске.

Права доступа создаваемых файлов

Аргумент `mode` требуется, если задан флаг `O_CREAT`.

```
int fd;  
fd = open (file, O_WRONLY | O_CREAT | O_TRUNC,  
           S_IWUSR | S_IRUSR | S_IWGRP | S_IRGRP | S_IROTH);  
if (fd == -1)  
    /* ошибка */
```

- при создании файла аргумент `mode` задает права доступа к этому новому файлу;
- режим доступа не проверяется при данном конкретном открытии файла;
- аргумент `mode` является UNIX-последовательностью битов, регламентирующей доступ.

Функция creat()

Комбинация `O_WRONLY | O_CREAT | O_TRUNC` настолько распространена, что существует специальный системный вызов, обеспечивающий именно такое поведение:.

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

```
int creat (const char *name, mode_t mode);
```

В большинстве архитектур Linux `creat()` является системным вызовом, хотя его можно легко реализовать и в пользовательском пространстве:

```
int creat (const char *name, int mode)
{
    return open (name, O_WRONLY | O_CREAT | O_TRUNC, mode);
}
```

Системный вызов read()

```
#include <unistd.h>
```

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

- Каждый вызов считывает не более len байт в памяти, на которые содержится указание в buf.
- Считывание происходит с текущим значением смещения, в файле, указанном в fd.
- При успешном вызове возвращается количество байтов, записанных в buf.
- При ошибке вызов возвращает -1 и устанавливает errno.
- Файловая позиция продвигается в зависимости от того, сколько байтов было считано с fd.
- Если объект, указанный в fd, не имеет возможности позиционирования (например, это файл символьного устройства), то считывание всегда начинается с «текущей» позиции.

Системный вызов read()

```
unsigned long word;  
ssize_t nr;  
  
/* считываем пару байт в 'word' из 'fd' */  
nr = read (fd, &word, sizeof (unsigned long));  
if (nr == -1)  
    /* ошибка */
```

- вызов может вернуться, считав не все байты из len;
- могут возникнуть ошибки, требующие исправления, но не проверяемые и не обрабатываемые в коде.

Последствия вызова `read()`

- вызов возвращает значение, равное `len`. Все `len` считанных байтов сохраняются в `buf`.
- вызов возвращает значение, меньшее чем `len`, но большее чем ноль. Считанные байты сохраняются в `buf`. Ошибка возникает в середине процесса, становится доступно значение, большее 0, но меньшее `len`. Конец файла был достигнут ранее, чем было прочитано заданное количество байтов. При повторном вызове (в котором соответствующим образом обновлены значения `len` и `buf`) оставшиеся байты будут считаны в оставшуюся часть буфера либо укажут на причину проблемы.
- Вызов возвращает 0. Это означает конец файла. Считывать больше нечего.
- Вызов блокируется, поскольку в текущий момент данные недоступны. Этого не происходит в неблокирующем режиме.

Последствия вызова `read()`

- Вызов возвращает `-1`, а `errno` присваивается `EINTR`. Это означает, что сигнал был получен прежде, чем были считаны какие-либо байты. Вызов будет повторен.
- Вызов возвращает `-1`, а `errno` присваивается `EAGAIN`. Это означает, что вызов блокировался потому, что в настоящий момент нет доступных данных, и запрос следует повторить позже. Это происходит только в неблокирующем режиме.
- Вызов возвращает `-1`, а `errno` присваивается иное значение, нежели `EINTR` или `EAGAIN`. Это означает более серьезную ошибку. Простое повторение вызова в данном случае, скорее всего, не поможет.

Считывание всех байт)

```
ssize_t ret;

while (len != 0 && (ret = read (fd, buf, len)) != 0) {
    if (ret == -1) {
        if (errno == EINTR)
            continue;
        perror ("read");
        break;
    }

    len -= ret;
    buf += ret;
}
```

Неблокирующее считывание)

Неблокирующий ввод-вывод - вызов `read()` не блокируется при отсутствии доступных данных, вместо этого - немедленный возврат вызова, указывающий, что данных действительно нет.

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
    if (errno == EINTR)
        goto start; /* вот незадача */
    if (errno == EAGAIN)
        /* повторить вызов позже */
    else
        /* ошибка */
}
```


Системный вызов write()

```
#include <unistd.h>
```

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

- при вызове write() записывается некоторое количество байтов, меньшее или равное тому, что указано в count.
- запись начинается с buf, установленного в текущую файловую позицию.
- при успешном выполнении возвращается количество записанных байтов, а файловая позиция обновляется соответственно.
- при ошибке возвращается -1 и устанавливается соответствующее значение errno.

Простейший пример использования

```
const char *buf = "My ship is solid!";
ssize_t nr;

/* строка, находящаяся в 'buf', записывается в 'fd' */
nr = write (fd, buf, strlen (buf));
if (nr == -1)
    /* ошибка */
```

Проверка всех ошибок

```
unsigned long word = 1720;
size_t count;
ssize_t nr;

count = sizeof (word);
nr = write (fd, &word, count);
if (nr == -1)
    /* ошибка, проверить errno */
else if (nr != count)
    /* возможна ошибка, но значение 'errno' не установлено */
```

Режимы записи и ошибки

Режим дозаписи:

- Когда дескриптор `fd` открывается в режиме дозаписи (с флагом `O_APPEND`), запись начинается не с текущей позиции дескриптора файла, а с точки, в которой в данный момент находится конец файла.

Неблокирующая запись:

- Когда дескриптор `fd` открывается в неблокирующем режиме (с флагом `O_NONBLOCK`), а запись в том виде, в котором она выполнена, в нормальных условиях должна быть заблокирована, системный вызов `write()` возвращает `-1` и устанавливает `errno` в значение `EAGAIN`. Запрос следует повторить позже.

Если значение `count` превышает `SSIZE_MAX`, то результат вызова `write()` не определен.

Синхронный ввод-вывод

fsync()

Метод, позволяющий гарантировать, что данные окажутся на диске - использовать системный вызов `fsync()`.

```
#include <unistd.h>
```

```
int fsync (int fd);
```

- Вызов `fsync()` гарантирует, что все грязные данные, ассоциированные с конкретным файлом, на который отображается дескриптор `fd`, будут записаны на диск.
- Файловый дескриптор `fd` должен быть открыт для записи.
- Вызов заносит на диск как данные, так и метаданные (цифровые отметки о времени создания файла и другие атрибуты).
- Вызов `fsync()` не вернется, пока жесткий диск не сообщит, что все данные и метаданные оказались на диске.

Коды ошибок fsync, fdatasync

В случае успеха возвращается 0, в противном возвращается -1.

Устанавливается значение errno:

- EBADF - указанный дескриптор файла не является допустимым дескриптором, открытым для записи;
- EINVAL - указанный дескриптор файла отображается на объект, не поддерживающий синхронизацию;
- EIO - при синхронизации произошла низкоуровневая ошибка вводавывода.

Синхронный ввод-вывод

`sync()`

обеспечивает синхронизацию всех буферов, имеющихя на диске

```
#include <unistd.h>

void sync (void);
```

- функция не имеет ни параметров, ни возвращаемого значения.
- функция всегда завершается успешно, и после ее возврата все буферы — содержащие как данные, так и метаданные — гарантированно оказываются на диске.

Флаг синхронизации

Флаг `O_SYNC` может быть передан вызову `open()`. Этот флаг означает, что все операции ввода-вывода, осуществляемые с этим файлом, должны быть синхронизированы.

```
int fd;

fd = open (file, O_WRONLY | O_SYNC);
if (fd == -1) {
    perror ("open");
    return -1;
}
```

- запросы на считывание синхронизированы **всегда**.
- вызовы `write()`, как правило, не синхронизируются.

Флаг `O_SYNC` можно рассмотреть в следующем ключе: он принудительно выполняет неявный вызов `fsync()` после каждой операции `write()` перед возвратом вызова.

Системный вызов `close()` - разорвать связь между дескриптором и файлом, который с ним ассоциирован

```
#include <unistd.h>
```

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

- `close()` отменяет отображение открытого файлового дескриптора `fd` и разрывает связь между файлом и процессом.
- ядро свободно может переиспользовать дескриптор как возвращаемое значение для последующих вызовов `open()` или `creat()`.
- успешное выполнение - возвращается 0, иначе -1.

```
if (close (fd) == -1)  
    perror ("close");
```

- Заккрытие файла никак не связано с актом сбрасывания файла на диск.

Системный вызов `lseek()` предназначен для установки в заданное значение файловой позиции конкретного файлового дескриптора.

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

off_t lseek (int fd, off_t pos, int origin);
```

Аргумент `origin`:

- `SEEK_CUR` - текущая файловая позиция дескриптора `fd` установлена в его текущее значение плюс `pos`. Последний может иметь отрицательное, положительное или нулевое значение. Если `pos` равен нулю, то возвращается текущее значение файловой позиции.
- `SEEK_END` - текущая файловая позиция дескриптора `fd` установлена в текущее значение длины файла плюс `pos`, который может иметь отрицательное, положительное или нулевое значение. Если `pos` равен нулю, то смещение устанавливается в конец файла.

Системный вызов `lseek()` предназначен для установки в заданное значение файловой позиции конкретного файлового дескриптора.

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

off_t lseek (int fd, off_t pos, int origin);
```

Аргумент `origin`:

- `SEEK_SET` - текущая файловая позиция дескриптора `fd` установлена в `pos`. Если `pos` равен нулю, то смещение устанавливается в начало файла.

Системный выход lseek()

Файловая позиция дескриптора fd получает значение 1825:

```
off_t ret;  
  
ret = lseek (fd, (off_t) 1825, SEEK_SET);  
if (ret == (off_t) -1)  
    /* ошибка */
```

Системный выход lseek()

Установить файловую позицию дескриптора fd в конец файла:

```
off_t ret;  
  
ret = lseek (fd, 0, SEEK_END);  
if (ret == (off_t) -1)  
    /* ошибка */
```

Системный выход lseek()

Вызов lseek() возвращает обновленную файловую позицию, поэтому его можно использовать для поиска текущей файловой позиции:

```
int pos;

pos = lseek (fd, 0, SEEK_CUR);
if (pos == (off_t) -1)
    /* ошибка */
else
    /* 'pos' – это текущая позиция fd */
```