Системное программирование

Лекция 10

Асинхронный ввод/вывод

Сокеты UNIX

Мультиплексирование ввода/вывода

Одной из самых медленных операций является коммуникация между субъектами (пользователем и процессом или несколькими процессами).

Зачастую, большую часть времени занимает ожидание новых данных.

В клиент-серверных приложениях сервер должен обрабатывать сообщения сразу от нескольких клиентов. 2 простых решения затратны:

1. схема 1 соединение – 1 поток [потоки большую часть времени простаивают];
2. неблокирующие операции (O_NONBLOCK/MSG_DONTWAIT)
   + цикл активного опроса [тратится излишнее время CPU].

Решением является **мультиплексирование ввода/вывода** – объединение операций нескольких операций ввода/вывода в одну.

В роли такой операции может выступать ожидание новых данных.

Вызов poll (пример)

Вызовы `poll()` блокирует вызывающий поток до тех пор, пока хотя бы один дескриптор из заданного набора не будет готов к выполнению желаемой операции ввода/вывода.

```
int poll (struct pollfd *fds, nfds_t nfds, int timeout);
```

Аргументы:

`fds` – массив структур, представляющих интересующие дескрипторы;
`nfds` – размер массива `fds`;
`timeout` – таймаут в секундах (-1 - нет ограничения);

Вызов возвращает число дескрипторов, для которых выполняются заданные условия, или 0 – если сработал таймаут.

Вызов не предназначен для работы с обычными файлами – только с каналами/сокетами/файлами устройств.

См. также: `ppoll()`

Структура pollfd

```
struct pollfd {
    int   fd;       /* дескриптор */
    short events;   /* ожидаемые события */
    short revents; /* произошедшие события */
};
```

Для каждого дескриптора в поле `events` указываются интересующие условия в виде битовой маски: `POLLIN` (наличие новых данные, чтение не заблокируется), `POLLOUT` (появление места в канале, запись не заблокируется) и др.

При возврате из `poll()` соответствующие запрошенным условиям флаги устанавливаются в поле `revents`.

Кроме того, в случае ошибки в `revents` могут выставляться флаги `POLLHUP` (обрыв канала), `POLLERR`(иная ошибка), `POLLNVAL` (невалидный дескриптор).

Использование poll

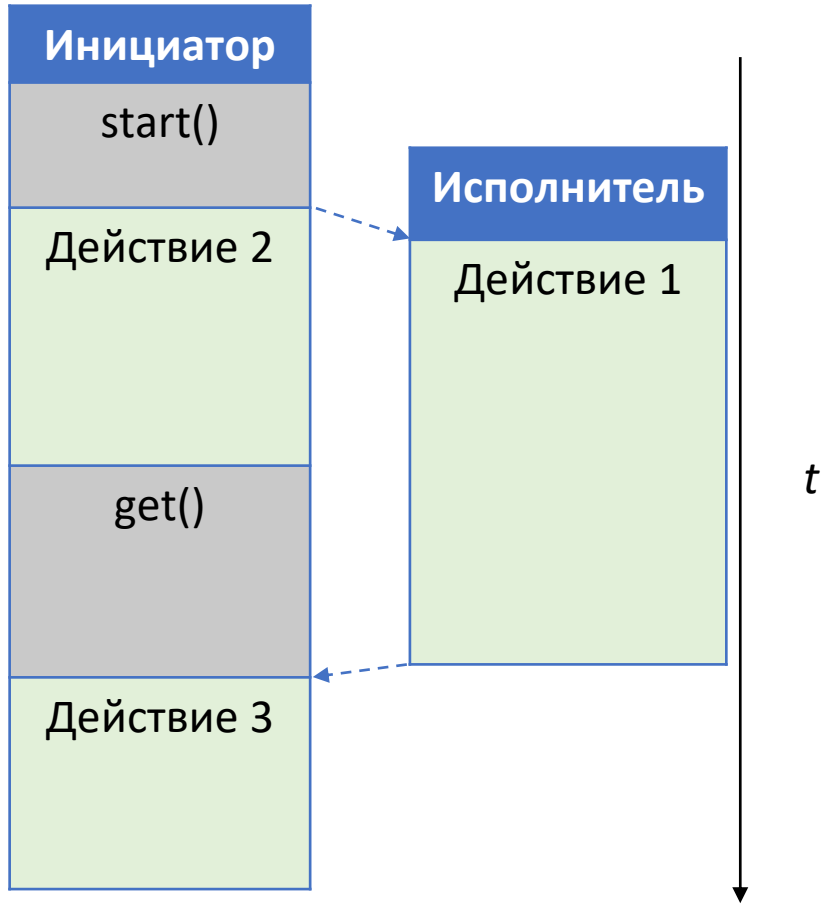
1. Составить массив структур `pollfd` и выставить в них флаги интересующих условий.
2. Вызвать `poll()` или `ppoll()`;
3. Если результат < 0 и `errno==EINTR` – обработать прерывание, GOTO 2;
4. Если результат == 0 – обработать таймаут, GOTO2;
5. Для каждой структуры `pollfd` :
   1. Проверить флаги в `revents`;
   2. Если выставлен флаг условия – выполнить операцию;
   3. Если выставлен флаг ошибки – обработать ошибку;
6. GOTO 2.

Асинхронные операции

Операция называется **асинхронной**, если инициация операции и выполнение операции являются отдельными действиями.

Чаще всего асинхронные операции применяются для повышения производительности – когда есть некоторая долгая операция, которую можно отдать для исполнения кому-то еще.

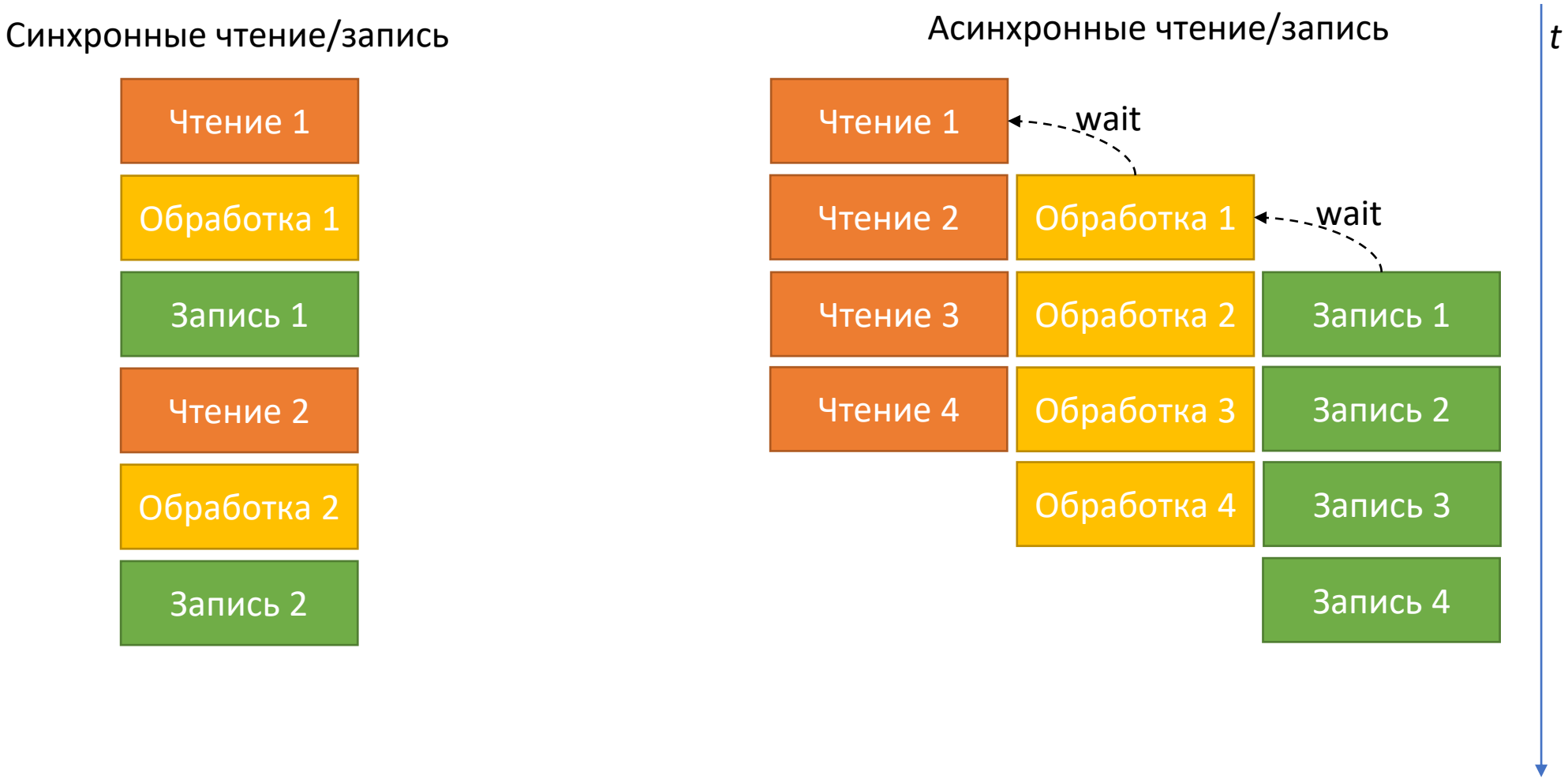
Исполнителем операции обычно выступает другой поток или ядро ОС.




```
int result = synchronous_operation();
```

```
start_asynchronous_operation();
/*…*/
int result = get_operation_result();
```

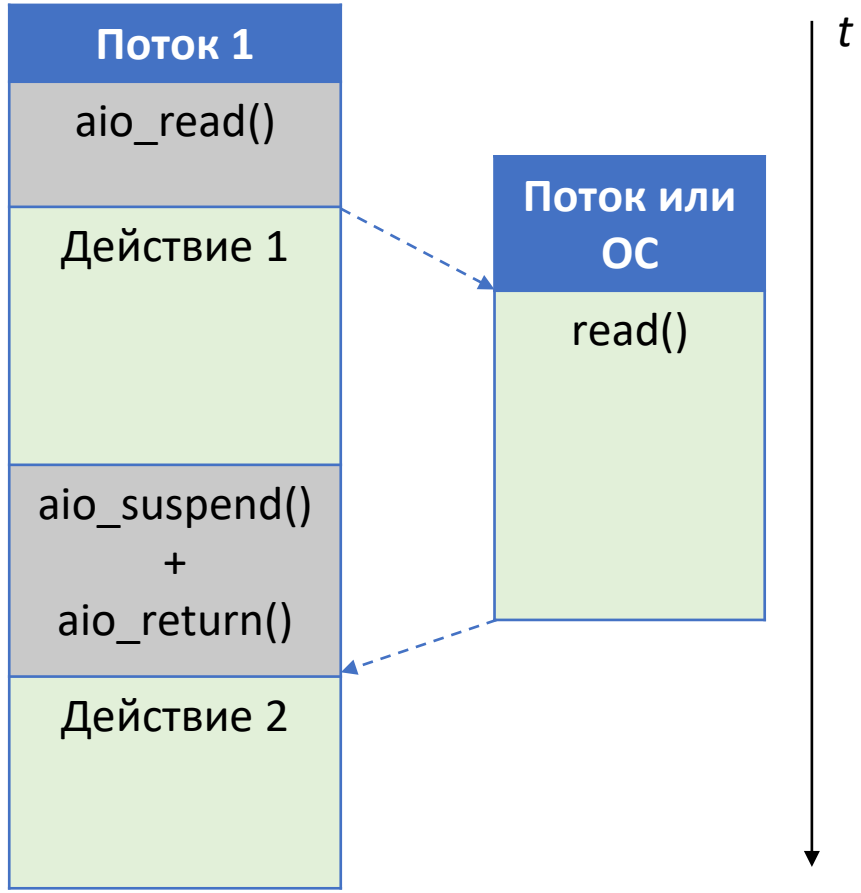
Синхронное и асинхронное чтение/запись



Асинхронные операции

В UNIX за асинхронные операции ввода-вывода отвечают функции с префиксом `aio_*()` из заголовочного файла `<aio.h>`.

Реализация данных функций зависит от системы. Они могут быть реализованы в пространстве пользователя поверх потоков, или в пространстве ядра – в этом случае системной будут представлены не-POSIX системные вызовы для асинхронного ввода-вывода, которые будут использоваться за кадром.



Структура aiocb

Для описания операции ввода-вывода используется структура aiocb (async I/O control block);

```
struct aiocb {
  int             aio_fildes;     /* Дескриптор файла */
  off_t           aio_offset;     /* Позиция в файле (но см. O_APPEND) */
  volatile void*  aio_buf;        /* Буфер */
  size_t          aio_nbytes;     /* Размер буфера */
  int             aio_reqprio;    /* Приоритет */
  struct sigevent aio_sigevent;   /* Способ извещения об окончании */
  int             aio_lio_opcode; /* Тип операции [для lio_listio()]*/
};
```

Извещение об окончании операции

Поле `aio_sigevent` задает способ извещения об окончании операции. Значением поля должен быть указатель на структуру `sigevent`:

```
struct sigevent {
  int          sigev_notify; // Способ извещения
  int          sigev_signo;  // Сигнал
  union sigval sigev_value;  // Данные, посылаемые при извещении
  /*Функция-обработчик извещения*/
  void(*sigev_notify_function) (union sigval);
  void* sigev_notify_attributes; /*Атрибуты потока*/
}
```

Извещение об окончании операции

Поле `sigev_notify` должно содержать одну из именованных констант:

- `SIGEV_NONE` – извещение не посылается,
- `SIGEV_SIGNAL` – процессу посылается сигнал из поля `sigev_signo;`
- `SIGEV_THREAD` – по завершении операции в отдельном потоке выполняется функция из поля `sigev_notify_function`. Атрибуты потока устанавливаются из `sigev_notify_attributes.`

Вместе с извещением передается значение из поля `sigev_value`, которое позволяет принимающей стороне получить дополнительную информацию, необходимую для обработки.

Асинхронное чтение-запись

Чтение/запись инициируются функциями `aio_read/aio_write()`.

```
int aio_read(struct aiocb* aiocbp);
int aio_write(struct aiocb* aiocbp);
```

При асинхронном чтении-записи текущее смещение в файле игнорируется, операции начинаются по смещению из поля `aiocbp->aio_offset`. Смещение после выполнения асинхронной операции не определено – отсюда следует, что нельзя смешивать асинхронные и неасинхронные операции.

Структура, на которую указывает `*aiocbp` не должна изменяться до конца операции!

Проверка статуса асинхронной операции

Узнать состояние асинхронной операции ввода-вывода можно функцией `aio_error()`.

```
int aio_error(const struct aiocb* aiocbp);
```

Если операция завершилась успешно, функция вернет 0.

Если операция еще выполняется, функция вернет `EINPROGRESS`.

Если операция отменена, функция вернет `ECANCELED`.

Любое другое значение означает ошибку операции и будет равно значению из `errno`, если бы операция выполнялась синхронно.

Отмена асинхронной операции

Операция может быть отменена функцией `aio_cancel()`.

```
int aio_cancel(int fd, struct aiocb* aiocbp);
```

Если `aiocbp==NULL`, то будут отменены все операции, связанные с данным дескриптором в `fd`. Иначе будет отменена только конкретная операция

Если операции были успешно отменены, возвращается `AIO_CANCELED`.

Если хотя бы одна операция не была отменена, возвращается `AIO_NOTCANCELED`.

Если все операции уже завершились, возвращается `AIO_ALLDONE`.

При ошибке возвращается -1.

Ожидание асинхронной операции

Дождаться завершения асиинхронной операции ввода-вывода можно функцией `aio_suspend()`.

```
int aio_suspend(const aiocb* const *aiocb_list, int nitems,
                const timespec* timeout);
```

Функция блокирует поток до тех пор, пока не завершится одна из операций, указанных в массиве aiocb_list (nitems задает размер массива). Опционально модно указать максимальное время ожидания в параметре timeout.

Функция может быть прервана сигналом, в этом случае она возвращает -1 с errno==EINTR.

Получение результата асинхронной операции (пример)

Результат асинхронной операции можно получить функцией `aio_return()`.

```
ssize_t aio_return(struct aiocb* aiocbp);
```

Функция не блокирует поток! Если операция еще не завершена, то результат не определен.

Отсюда следует, что нужно сначала убедиться, что операция завершена вызовами `aio_suspend/aio_error()`.

Функция возвращает то же самое, что и вызовы `read/write` – число успешно считанных/записанных байт. Если в ходе операции произошла ошибка, то функция вернет -1 и установит `errno` соответствующим образом.

Сокеты домена UNIX

Сокеты домена UNIX (UNIX Domain Sockets, `AF_UNIX`) являются средством межпроцессного взаимодействия.

- Тип `SOCK_STREAM` создает сокет UNIX с семантикой обычного файла (через сокет передается поток данных). До начала передачи сокет должен установить соединение через `connect()`.
- Тип `SOCK_DGRAM` создает сокет UNIX с семантикой очереди сообщений (через сокет передаются отдельные сообщения). Сообщения могут происходить от нескольких процессов. POSIX не гарантирует ни порядок передачи, ни надежность передачи сообщений.
- Тип `SOCK_SEQPACKET` создает сокет UNIX с семантикой очереди сообщений (через сокет передаются отдельные сообщения). До начала передачи сокет должен установить соединение через `connect()`. *Гарантируется сохранность порядка передачи.*

Структура sockaddr_un

Структура `sockaddr_un` используется для указания адреса сокета UNIX в качестве 2 аргумента функций `bind()` и `connect()`.

```
struct sockaddr_un {
  sa_family_t sun_family;    /* =AF_UNIX */
  char        sun_path[108]; /* абсолютный путь */
};
```

Сокет создается в качестве файла специального типа, но открыть его через `open()` нельзя.

Вызовы sendmsg и recvmsg

Для работы с сокетами UNIX можно использовать `send()/sendto()` и `recv()/recvfom()`.

Кроме того, следующая пара методов позволяет получать вместе с сообщениями *дополнительную информацию*:

```
ssize_t sendmsg(int sockfd, const msghdr* msg, int flags);
ssize_t recvmsg(int sockfd, msghdr* msg, int flags);
```

Вызовы возвращают размер принятого/отправленного сообщения.

Если для записи данных недостаточно места, в `msg->msg_flags` после `recvmsg()` будет установлен флаг `MSG_TRUNC` (если нет места для обычных данных) или `MSG_CTRUNC` (если нет места для специальных данных).

Структура msghdr

```
struct msghdr {
  void*      msg_name;       /* Буфер адреса источника */
  socklen_t  msg_namelen;    /* размер буфера  msg_name*/
  iovec*     msg_iov;        /* Массив буферов данных */
  size_t     msg_iovlen;     /* Длина msg_iov */
  void*      msg_control;    /* специальные данные */
  size_t     msg_controllen; /* длина спец. данных */
  int        msg_flags;      /* флаги*/
};
```

Структура iovec

При отправке сообщения в поле `msghdr.msg_iov` передается массив буферов ввода/вывода. В одном сообщении могут быть собраны данные из нескольких буферов.

```
struct iovec {      /* Буфер для ввода/вывода */
  void* iov_base;   /* Адрес буфера */
  size_t iov_len;   /* Размер буфера */
};
```

Структура cmsghdr

Буфер специальных данных передается в поле `msghdr.msg_control`. В одном сообщении могут быть переданы несколько сообщений специальных данных, но все они должны быть в одном буфере. Каждое такое сообщение должно начинаться с заголовка, описываемого структурой `cmsghdr`.

```
struct cmsghdr {
  size_t cmsg_len;    /* Размер данных с учетом заголовка*/
  int    cmsg_level;  /* Исходный протокол передачи */
  int    cmsg_type;   /* Тип данных */
};
```

Структура cmsghdr

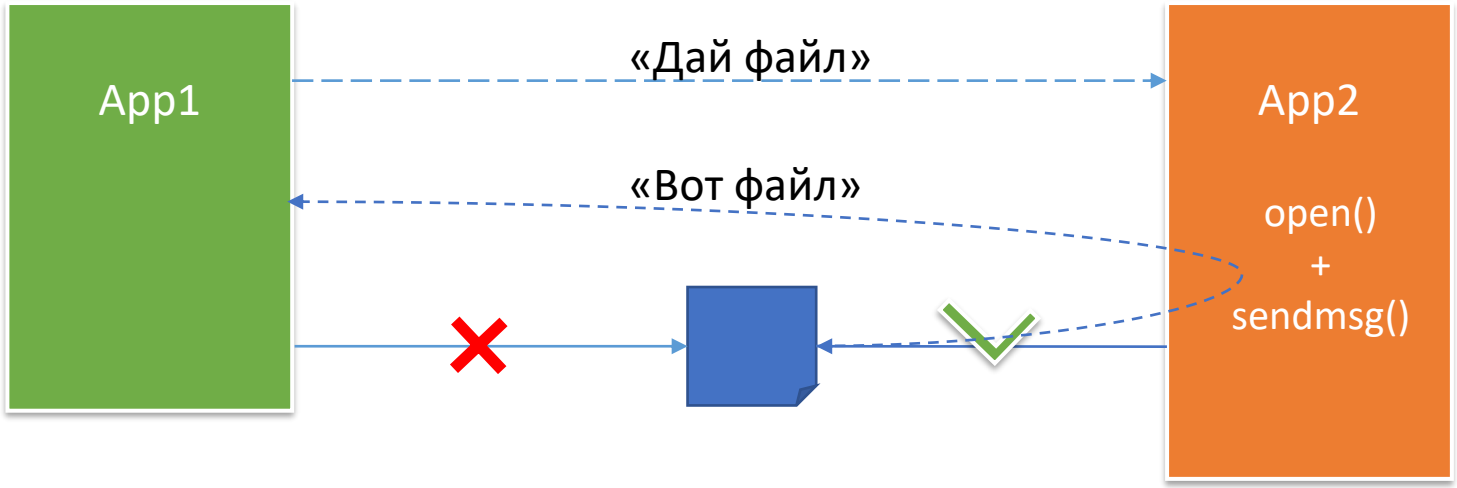
Для облегчения работы с буферами сообщений специальных данных используется ряд макросов:

```
/*получить указатель на первое сообщение спец. данных или 0*/
cmsghdr* CMSG_FIRSTHDR(msghdr* msgh);

/*получить указатель на следующее сообщение, если оно есть*/
cmsghdr* CMSG_NXTHDR( msghdr* msgh, cmsghdr* cmsg);

/*рассчитать общий размер заголовка и данных*/
size_t CMSG_SPACE(size_t length);

/*возвращает указатель на данные после заголовка*/
unsigned char* CMSG_DATA(struct cmsghdr* cmsg);
```

Передача дескрипторов файлов (пример)

Особенностью сокетов домена UNIX является возможность пересылки между процессами дескрипторов открытых файлов.

При этом, поскольку проверка прав происходит в момент открытия файла, возможно передать дескриптор открытого файла процессу, который формально не имеет привилегий на чтение/запись в файл.



Аутентификация по сокету (пример)

Помимо пересылки файловых дескрипторов, через UNIX-сокет в виде дополнительных данных могут отправлены идентификаторы пользователя и группы текущего процесса.

При этом отправляющий процесс *не может подделать* эти идентификаторы (но он может выбирать, какой из идентификаторов отправлять – реальный, эффективный или сохраненный).

Принимающий процесс может проверить отправителя и принять/отклонить запрос.

Для отправки и приема идентификаторов необходимо установить опцию сокета SO_PASSCRED

```
int t = 1;
setsockopt(sockfd, SOL_SOCKET, SO_PASSCRED, &t, sizeof t);
```