

Часть 2. Системное программирование в UNIX

1. Работа с файловой структурой UNIX

1.1. Интерфейсы прикладного программирования UNIX и POSIX

Операционная система UNIX имеет набор API функций, построенных на основе системных вызовов функций ядра операционной системы. Большинство API функций работает в режиме ядра – защищенном режиме, позволяющему процессу получать доступ к данным ядра операционной системы. После завершения выполнения системного вызова API функции, процесс возвращается в режим пользователя. Переключения контекстов во время вызова системных API функций требует дополнительного времени, по сравнению с вызовом функций библиотеки, но предоставляет значительно больше возможностей для выполнения системных операций.

Большинство функций используют стандарт POSIX.1 и POSIX.1b, в целом аналогичных API функциям UNIX.

Функции API возвращают значение `-1` в случае не успешного выполнения операции. Конкретное значение ошибки содержится в глобальной переменной `errno` описанной в файле `errno.h`. Тестовую строку, содержащую системное сообщение об ошибке, возвращает функция `strerror()`. Наиболее часто встречающиеся ошибки приведены в табл. 1.1.

Табл. 1.1 Значение кодов ошибок

<code>errno</code>	Описание ошибки
<code>EACCESS</code>	Процесс не имеет права на выполнение операции.
<code>EPERM</code>	Вызывающий процесс не имеет прав привилегированного пользователя.
<code>ENOENT</code>	Используется недопустимое имя файла.
<code>BADF</code>	Используется недопустимый дескриптор файла.
<code>EINTR</code>	Вызов функции прерван внешним сигналом.
<code>EAGAIN</code>	Один из используемых в процессе выполнения API временно не доступен (занят).
<code>ENOMEM</code>	Ошибка динамического выделения памяти.
<code>EIO</code>	Ошибка ввода/вывода.
<code>EPIPE</code>	Попытка записи в канал, для которого не существует читающего процесса.
<code>EFAULT</code>	Ошибка адреса одного из аргументов.
<code>ENOEXEC</code>	Ошибка запуска процесса.
<code>ESCHILD</code>	Процесс не имеет дочерних процессов, завершения которых он мог бы ждать.

1.2. Работа с файлами в операционной системе UNIX

Для представления информации о файле в UNIX используется структура `st_t` описанная в файле `sys/stat.h`. Структура `st_t` инициализируется и заполняется функцией `stat()` или `fstat()`.

Структура `stat` описана следующим образом:

```
struct stat
{
    dev_t    st_dev ;    /* идентификатор файловой системы    */
    ino_t    st_ino ;    /* номер индексного дескриптора файла */
    mode_t   st_mode ;   /* тип файла и флаги доступа         */
    nlink_t  st_nlink ;  /* счетчик жестких ссылок            */
    uid_t    st_uid ;   /* UDI владельца файла              */
    gid_t    st_gid ;   /* GID владельца файла              */
    dev_t    st_rdev ;   /* номера устройств                 */
    off_t    st_size ;  /* размер файла                      */
    time_t   st_atime ;  /* время последнего доступа         */
    time_t   st_mtime ;  /* время последней модификации      */
    time_t   st_ctime ;  /* время последнего изменения статуса*/
}
```

Максимальное значение одновременно открытых файлов определяется значением `OPEN_MAX` (`_POSIX_OPEN_MAX`), определенных в `limits.h`.

Порядок действий при открытии файла выглядит следующим образом:

1. Поиск в таблице дескрипторов первой свободной позиции. Если свободная позиция не обнаружена, то возвращается значение `-1`. Иначе, в качестве дескриптора файла, возвращается номер задействованной позиции в таблице.

2. Поиск незадействованной позиции в таблице файлов. Если свободная позиция не обнаружена, то возвращается значение `-1`. Если в таблице файлов найдена свободная позиция, то выполняются следующие действия:

- в выделенной записи таблицы дескрипторов делается ссылка на найденную запись в таблице файлов, аналогичная ссылка на номер в таблице файлов делается на номер в таблице дескрипторов;
- в записи таблицы файлов формируется указатель текущей позиции в файле (смещение от начала файла);
- в записи таблицы файлов записывается режим, в котором был открыт файл;
- значение счетчика количества дескрипторов связанных с файлом в таблице файлов устанавливается в 1;
- значение счетчика ссылок индексного дескриптора (указывает, сколько записей таблицы файлов указывают на данный дескриптор) увеличивается на 1.

При закрытии файла выполняются следующие действия:

1. отметить запись в таблице дескрипторов как свободную;
2. уменьшить значение счетчика таблицы файлов на 1, если значение не равно 0, то переход на пп 6;
3. отметить запись в таблице файлов как свободную;
4. уменьшить значение счетчика в таблице индексных дескрипторов на 1, если значение не равно 0, то переход на пп 6;
5. если значение счетчика жестких ссылок равно 0, то позиция в таблице помечается свободной и освобождается вся физическая память, выделенная для хранения файла;
6. ядро передает управление процессу и возвращает код успешного выполнения.

Работа с файлами реализуются через потоки или через дескриптор файла. Работа через указатели потоков более эффективна при больших объемах последовательных операций чтения/записи в файл. Это является следствием того, что библиотечные функции Си, предназначенные для работы с файлами через потоки ввода/вывода, используют буферизацию.

Работа файлами через дескриптор файла является более эффективным для приложений осуществляющих произвольный доступ к данным из файла.

Однако, на практике для эти два способа оказываются сильно связаны между собой. Функция `foren()` для фактического открытия файла обращается к функции `open()`, работающей через дескрипторы.

Ниже представлены наиболее часто использующиеся функции, описанные в модуле `stdio.h`:

FILE * fopen(char *NameFile, char *open_mode)

Функция `fopen()` открывает файл указанной в строке `NameFile`. Режим открытия файла определяется строкой `open_mode`. Допустимы следующие режимы открытия файлов:

“r”	– для чтения;	“r+”	– для чтения в монопольном режиме;
“a”	– для добавления в конец файла;	“a+”	– для добавления в монопольном режиме;
“w”	– для записи;	“w+”	– для записи в монопольном режиме;

В случае если функции `fopen()` не удалось открыть файл, то она возвращает значение `NULL`.

Пример использования функции `fopen()`:

```
FILE *FP ;
FP = fopen("test.txt","r") ;
if(FP == NULL)
{
    printf("\n Ошибка открытия файла") ;
    . . . /* Обработка ошибки открытия файла */
}
```

int fileno(FILE *FP)

Функция `fileno()` определяет дескриптор файла по указателю на поток. Запись `FILE` содержит дескриптор открытого файла, которая может быть получена и использована для выполнения операций чтения/записи в файл через дескриптор.

Например, для рассмотренного выше примера возможно выполнение следующих действий:

```
FILE *FP ;
int handle, A[10] ;
FP = fopen("test.txt","w") ;
if(FP == NULL)
    {   printf("\n Ошибка открытия файла") ;
        ... /* Обработка ошибки открытия файла */
    }
handle = fileno(FP) ;
write(handle, A, sizeof(A)) ;
```

FILE * fdopen(int handle, char *open_mode)

Функция `fdopen()` создает указатель на поток ввода/вывода для файла по номеру дескриптора. В целом функция аналогична функции `fopen()`, однако файл повторно не открывается, а только создается указатель в таблице потоков для уже существующего значения дескриптора. После выполнения функции программист может работать с файлами одновременно и через дескриптор и через указатель потока.

Например:

```
int handle ;
FILE *FP ;
float r1, r2 ;
handle = open("test.dat", O_BINARY) ;
FP = fdopen(handle, "r") ;
read(handle, &r1, sizeof(float) ;
fscanf(FP, "%f", &r2) ;
```

1.3. Работа структурой каталогов

С точки зрения реализации файловой структуры, каталог – это служебный файл с информацией о файлах и подкаталогах. Каждая запись описывается структурой `struct direct`.

```
struct direct
{
    unsigned short int d_fileno;
    char d_name[14];
}
```

Для работы со структурой каталогов в Си предусмотрен набор функций, в целом реализуемых через системные вызовы `open()`, `read()`, `write()`, `lseek()`, `close()`. Функции работы с каталогами определены в `direct.h` (`sys/dir.h` в BSD UNIX).

int mkdir(char *name, mode_t mode)

Функция `mkdir()` создает каталог с указанным именем и атрибутами, которые как и создаваемые файлы маскируются значением `umask`. В случае не успешного завершения вызова функция возвращает значение `-1`.

```
Char pathname[] = "/home/user/new_dir";
```

```
If(mkdir(pathname, mode) != 0)
{
    printf("Error opening directory %s", pathname);
}
```

DIR *opendir(char *name_dir)

Функция `opendir()` возвращает указатель на поток связанный со служебным файлом каталога. В случае если функции `opendir()` не сумеет открыть каталог, то возвращается значение `NULL`.

```
dirent * readdir(DIR *dir_id)
```

Функция `readdir()` выдает указатель на следующий активный элемент каталога. Указатели на неактивные элементы каталога не выдаются. При достижении конца каталога или при выявлении некорректной позиции в каталоге возвращается пустой указатель.

Функция `telldir()` выдает текущую позицию в указанном потоке каталога.

Функция `seekdir()` устанавливает позицию для последующей операции `readdir()` над потоком каталога. Данная позиция совпадает с той, которая была получена в результате выполнения операции `telldir()`, вычислившей значение `loc`. Значения, которые возвращает `telldir()`, корректны только в том случае, если каталог не сжимался и не расширялся. Такая проблема не возникает в случае версии 5, но может возникнуть для некоторых других типов файловых систем.

void rewinddir(DIR *dir_id)

Макрос `rewinddir()` переустанавливает в начало позицию в указанном потоке каталога. В связи с тем, что `rewinddir()` реализована в виде макроса, к ней нельзя применить операцию вычисления адреса функции.

int closedir(DIR *dir_id)

Функция `closedir()` закрывает указанный поток каталога и освобождает структуру `DIR`.

```
int rmdir(char *name)
```

Функция `rmdir()` удаляет каталог. **Внимание!** Удаляемый каталог должен быть пустым.

Например, рассмотрим фрагмент программы для поиска в каталоге элемента `name`:

```
dirp = opendir (".");
while ((dp = readdir (dirp)) != NULL)
if (strcmp (dp->d_name, name) == 0)
{
    closedir (dirp);
    return FOUND;
}
closedir (dirp);
return NOT_FOUND;
```

1.4. Работа с жесткими ссылками

Жесткая ссылка – это путь к файлу в операционной системе UNIX. Обычно файлы имеют только одну жесткую ссылку – полное имя файла. Новую жесткую ссылку можно создать с помощью команды `ln`, например:

```
ln /stud/pim.doc /users/test.dat.
```

После выполнения этой команды, пользователи могут работать с этим файлом и как с `/stud/pim.doc`, и как `/users/test.dat`.

Кроме жестких ссылок, команда `ln` позволяет создавать и символические ссылки. Разница между жесткими и символическими ссылками представлена в табл. 1.2.

Таблица 1.2. Жесткие и символические ссылки

Жесткая ссылка	Символическая ссылка
Не создает нового индексного дескриптора.	Создает новый индексный дескриптор.
Не позволяет привилегированному пользователю ссылаться на каталоги	Может ссылаться на каталоги.
Не может ссылаться на файлы в других файловых системах	Может ссылаться на файлы в других файловых системах.
Увеличивает значение счетчика жестких ссылок соответствующего индексного дескриптора.	Не изменяет значение счетчика жестких ссылок соответствующего индексного дескриптора.

1.5. Функции работы с файлами

int open(char *file_name, int access_mode, mode_t mode_file)

Функция open() открывает файл для работы через дескриптор файла. Прототипы функции описаны в sys/types.h и fcntl.h.

Значения аргумента access_mode приведены в табл. 1.3.

Таблица 1.3 Флаги режима открытия файла

Флаг режима доступа	Описание
O_RDONLY	Открывает файл только для чтения.
O_WRONLY	Открывает файл только для записи.
O_RDWR	Открывает файл и для чтения и для записи.
O_APPEND	Открывает файл для добавления.
O_CREAT	Режим создания файла. Если файл уже существует, то флаг не влияет.
O_EXCL	Используется совместно с флагом O_CREAT для завершения работы функции open() в случае если создаваемый файл уже существует.
O_TRUNC	Открыть файл с усечением.
O_NOBLOCK	Запрещает режим блокировки операций чтения и записи в файл. Внимание! Флаг применим только к файлам устройств и FIFO файлам.
O_NOCTTY	Запрещает использовать файл терминального устройства в качестве управляющего терминала вызывающего процесса. Внимание! Флаг применим только к файлам терминальных устройств.

Аргумент mode_file функции open() используется только при флаге O_CREAT и задает атрибуты создаваемого файла.

Возможные значения mode_file приведены в табл. 1.4.

Табл. 1.4. Флаги атрибутов создаваемого файла

Флаг режима доступа	Описание

В случае не успешного завершения функция open() возвращает значение -1. Значение дескриптора успешно открытого файла находится в пределах от 0 до OPEN_MAX-1.

Например:

```
int handle ;  
handle = open("test.dat", O_BINARY) ;  
if( handle == -1)  
{
```

```
printf(“\n Ошибка открытия файла”);
...
}
```

Атрибуты создаваемого файла автоматически маскируются значением `umask` вызывающего процесса. Значение `umask` указывает те атрибуты файла, которые необходимо исключить. Значение `umask` передается процессам наследникам, и может быть модифицировано функцией `umask()`.

mode_t umask(mode_t new_mask)

Функции `umask()` передается новое значение маски, при этом она возвращает старое значение маски.

Например:

```
mode_t old_mask ;
old_mask = umask( S_IXGRP | S_IWOTH | S_IXOTH );
```

При создании файла функций `open()` файл будет создан с атрибутами определенными выражением: `mode_file & ~umask`.

size_t read(int handle, void *buf, size_t size)

Функция `read()` читает данные из файла, заданного дескриптором `handle`. Указанное значением `size` количество байт читается в переменную по адресу `buf`. Функция `read()` возвращает количество прочитанных байт.

Например:

```
int A[20], N ;
size_t size_read ;
...
read(handle, &N, sizeof(N)) ;
size_read = read(handle, A, sizeof(A)) ;
if( size_read < sizeof(A))
{
    printf(“\n Прочитано меньше байт чем ожидалось “) ;
    ...
}
```

В связи тем что `size_t` определен как `int`, максимальный размер блока, читаемого из файла не может превышать максимального значения типа `int`.

Одной из возможных причин, почему функция `read()` прочитала не полный объем данных, может быть прерывания при поступлении сигнала. В том случае, если выполнение функции `read()` было прервано сигналом, операционная система (кроме BSD) не перезапустит его автоматически.

size_t write(int handle, void *buf, size_t size)

Функция `write()` записывает указанное количество байт (`size`) из переменной расположенной по адресу `buf` в файл открытый через

дескриптор `handle`. Функция `write()` возвращает количество успешно записанных байт.

Например:

```
int A[20], N, size_write ;  
...  
write(handle, &N, sizeof(N)) ;  
size_write = write(handle, A, sizeof(A)) ;
```

int fcntl(int handle, int cmd, ...)

Функция `fcntl()` позволяет получать и устанавливать флаги управления доступом для файла, заданного дескриптором `handle`. Например, если файл открыт для чтения, а пользователю необходимо внести в него изменения, или если необходимо организовать работу с файлом через различные дескрипторы.

Значение `cmd` может быть одним из приведенных в табл. 1.5.

Табл. 1.5. Аргументы функции `fcntl()`

Значение <code>cmd</code>	Описание
<code>F_GETFL</code>	Возвращает режим открытия файла.
<code>F_SETFL</code>	Изменяет режим открытия файла на указанный.
<code>F_GETFD</code>	Возвращает значение флага <code>close-on-exec</code> .
<code>F_SETFD</code>	Устанавливает или сбрасывает, в зависимости от третьего аргумента (1 или 0) флаг <code>close-on-exec</code> .
<code>F_DUPFD</code>	Создает дубликат дескриптора файла.

Например:

```
int handle, A[20] ;  
handle = open("test.dat", O_BINARY) ;  
read(handle, A, sizeof(A)) ;  
fcntl(handle, F_SETFL, O_RDWR | O_BINARY) ;  
write(handle, A, sizeof(A)) ;
```

off_t lseek(int handle, off_t pos, int base)

Функция `lseek()` позволяет позиционироваться по файлу. Указатель для файла `handle` перемещается в позицию `pos` отсчитываемую от значения `base`. Допустимые значения `base` приведены в табл. 1.6.

Табл. 1.6. Аргументы функции `lseek()`

Значение <code>base</code>	Описание
<code>SEEK_SET</code>	Начало файла.
<code>SEEK_CUR</code>	Текущая позиция в файле.
<code>SEEK_END</code>	Конец файла.

Функция `lseek()` возвращает значение позиции, в которую был перемещен указатель файла.

int link(char *name, char *new_name)

Функция `link()` создает новую ссылку на существующий файл. Операционная система UNIX не позволяет создавать жесткие ссылки на файлы, находящиеся в других файловых системах. Для создания жесткой ссылки на каталог необходимо иметь права привилегированного пользователя. Если операция выпалилась не успешно, то функция возвращает значение `-1`.

```
if(link("/path/to/file", "/path/to/link") != 0 )
{
    printf("Error creating link");
}
```

int unlink(char *name)

Функция `unlink()` удаляет жесткую ссылку на файл. При этом если ссылка являлась последней (счетчик жестких ссылок стал равным 0), то удаляется и сам файл. Если вызов функции закончился не успешно, то возвращается значение `-1`.

Функция `unlink()` может работать с каталогами, только в случае если процесс обладает правами привилегированного пользователя.

```
If(unlink("/file/to/delete") != 0)
{
    printf("Error");
}
```

int rename(char *old_name, char *new_name)

Функция `rename()` переименовывает файл. При этом требуется, что бы файлы находились в одной файловой системе. В случае не успешного выполнения функция возвращает значение `-1`.

```
If(rename("/path/to/file/oldname", "/path/to/file/newname") != 0)
{
    printf("Error");
}
```

int stat(char *name, struct stat *atrib)

int fstat(int handle, struct stat *atrib)

Функции `stat()` и `fstat()` позволяют получить атрибуты файла. Для функции `stat()` файл передается именем, а для функции `fstat()` – дескриптором файла. В случае если у процесса не доступа к файлу, то функции возвращают

значение -1 . В случае успешного выполнения функции заполняют поля структуры `struct stat`, которая описана следующим образом:

```

struct stat
{
    dev_t    t_dev ;    /* идентификатор файловой системы          */
    ino_t    st_ino ;   /* номер индексного дескриптора файла      */
    mode_t   st_mode ;  /* тип файла и флаги доступа              */
    nlink_t  st_nlink ; /* счетчик жестких ссылок                  */
    uid_t    st_uid ;   /* идентификатор владельца файла          */
    gid_t    st_gid ;   /* идентификатор группы                     */
    dev_t    st_rdev ;  /* номер устройства                        */
    off_t    st_size ;  /* размер файла в байтах                   */
    time_t   st_atime ; /* время последнего доступа                */
    time_t   st_mtime ; /* время последней модификации            */
    time_t   st_ctime ; /* время последнего изменения статуса      */
}

stat    attrib;
if(stat("filename", &attrib) != 0)
{
    printf("Error");
}
printf("Filesize %d bytes", attrib.st_size);

```

int lstat(char *name, struct stat *atrib)

Функция полностью аналогична функции `stat()` но работает не только с файлами, но и с символическими ссылками (выдает данные не по файлу, на который ссылается ссылка, по самой ссылке).

int access(char *name, int flag)

Функция `access()` проверяет наличие файла и прав доступа на этот файл у пользователя. Значение параметра `flag` определяет тип запроса к файлу. В случае положительного ответа, функция возвращает значение 0 , иначе значение -1 . Возможные значения параметра `flag` приведены в табл. 1.7.

Табл. 1.7. Запросы к атрибутам файла

Значение flag	Запрос к файлу
F_OK	Проверить существует ли файл.
R_OK	Есть ли у вызывающего процесса право на чтение.
W_OK	Есть ли у вызывающего процесса право на запись.
X_OK	Есть ли у вызывающего процесса право на выполнение.

Значение `flag` может быть получено путем побитового сложения отдельных атрибутов, например: `flag = F_OK | R_OK`.

```

If(access(“filename”, X_OK) != 0)
{
    printf(“You cannot execute this file”);
}

```

int close(int handle)

Функция close() закрывает файл, заданный дескриптором handle. В случае не удачного завершения вызова, возвращается значение -1, а переменная errno устанавливается в код ошибки. По умолчанию ядро операционной системы UNIX при завершении процесса закрывает все открытые им файлы.

```

If(close(fp) != 0)
{
    printf(“Error closing file”);
}

```

int fcntl(int handle, int cmd_flag, struct flock *f_flock)

Функция fcntl() позволяет блокировать файл, заданный дескриптором handle. Операционная система UNIX позволяет организовывать доступ к файлу со стороны многих процессов. При этом может иметь место попытки работы с одним и тем же участком файла. Решение этой ситуации возможно за счет блокировки файлов, или областей файлов.

Конкретный режим блокировки задается значением cmd_flag, приведенные в табл. 1.8.

Табл. 1.8. Режимы блокировки файлов

Значение cmd_flag	Назначение
F_SETLKW	Блокировать файл.
F_SETLKW	Блокирует файл и вызывающий процесс на время пока действует блокировка на файл.
F_GETLKW	Дает запрос, о том какой процесс заблокировал файл.

Указание, какую область файла необходимо блокировать, заносится в поля структуры struct flock, описанной следующим образом:

```

struct flock
{
    short    l_type ;    /* тип блокировки                */
    short    l_whence ; /* адрес следующего поля        */
    off_t    l_start ;  /* адрес начала блокировки      */
    off_t    l_len ;    /* размер заблокированной области */
    pid_t    l_pid ;    /* идентификатор блокирующего процесса */
}

```

Тип блокировки задается значением поля l_type, значения которого приведены в табл. 1.9.

Табл.1.9.Значения поля l_type

Значение l_type	Действие
F_RDLCK	Установить блокировку по чтению.
F_WRLCK	Установить блокировку по записи.
F_UNLCK	Снять блокировку указанной области.

Если установить значение поля l_len равном 0, то будет заблокирована вся область до конца файла.

Внимание! Все установленные процессом блокировки снимаются при его завершении.

```

Int owner_id = fcntl(fp, F_GETOWN);
If(owner_id == -1)
{
    printf("No owner?");
}

```

2. Процессы в операционной системе UNIX

Процесс – это программа, находящаяся в стадии выполнения. Процесс в ОС UNIX создается с помощью системного вызова fork(). Создаваемый процесс получает копи сегментов стека, данных, кода и таблицу дескрипторов родительского процесса.

Родительский процесс может приостановить свое выполнение с помощью функций wait() и waitpid() до завершения процесса наследника. В случае независимого продолжения процесса родителя и наследника родительский процесс может получать информацию о состоянии процесса наследника с помощью функций signal() и sigaction().

Завершение выполнения процесса выполняется с помощью функции _exit(). Традиционно считается, что нулевой код завершения процесса является успешным.

Вызов exec() позволяет запустить другую программу, вместо вызывающего процесса. В результате данного вызова идентификатор процесса и другая служебная информация, включая дескрипторы открытых каталогов и файлов (кроме тех у которых установлен флаг close-on-exec – закрыть по завершении процесса).

pid_t fork(void)

Функция fork() создает дочерний процесс. В случае успешного выполнения возвращается идентификатор созданного процесса. В случае не успешного выполнения возвращается значение -1 и устанавливается значение переменной errno. Текстовый вариант сообщения об ошибке может быть получено с помощью функции perror().

```

Pid_t pid_id = fork();
If(pid_id == -1)
{
    printf("Error creating process");
}

```

void _exit(int exit_code)

Функция `_exit()` завершает выполнение процесса, при этом освобождаются все сегменты процесса и закрываются все дескрипторы открытых файлов.

Соответствующая завершаемому процессу запись в таблице процессов сохраняется для предоставления возможности родительскому процессу получить код завершения процесса.

В качестве кода завершения процесса используются значения от 0 до 255.

pid_t wait(int *status_p)

pid_t waitpid(pid_t child_pid, int *status_p, int option)

Функции `wait()` и `waitpid()` приостанавливают выполнение родительского процесса до завершения вызванного дочернего процесса. Данные вызовы позволяют получить код завершения процесса и освободить запись о процессе в таблице процессов.

Функция `wait()` приостанавливает выполнение родительского процесса до получения сигнала или завершения одного из дочерних процессов. В случае завершения дочернего процесса функция `wait()` возвращает код завершения процесса. Если в момент вызова функции `wait()` родительский процесс не имеет ни одного созданного им дочернего процесса, то возвращается значение `-1`.

Функция `waitpid()` позволяет получить более полную информацию о завершаемом процессе. В зависимости от значения аргумента `child_pid`, можно уточнить, завершения какого из дочерних процессов ждет родительский процесс. Возможные значения `child_pid` приведены в табл. 2.1.

Табл. 2.1. Аргументы функции `waitpid()`

Значение <code>child_pid</code>	Завершения какого дочернего процесса ждет родительский процесс
ИД дочернего процесса	Завершения дочернего процесса с идентификатором равным указанному значению.
-1	Завершения любого дочернего процесса.
0	Завершения любого дочернего процесса, принадлежащего к одной группе с родительским процессом.
Отрицательное значение (кроме -1)	Завершения любого дочернего процесса, идентификатор группы которого совпадает с абсолютным значением <code>child_pid</code> .

Значение аргумента `option` позволяет задать режимы управления заданиями и блокировки. В переменную `status_p`, передаваемую через

указатель записывается код завершения процесса устанавливаемый в функции `_exit()` дочернего процесса. В случае если код завершения контролировать не требуется, то передается значение `NULL`.

```
int execl(char *name, char *arg,)  
int execlp(char *name, char *arg)  
int execl_e(char *name, char *arg, char **env)  
int execv(char *name, char **arg)  
int execvp(char *name, char **arg)  
int execve(char *name, char **arg, char **env)
```

Функции `exec()` позволяют запустить на выполнение новый процесс. При этом новый процесс заменяет в памяти сегменты кода, данных и стека вызывающего процесса, но не создает новую запись в таблице процессов. Функции имеют незначительные отличия по особенностям передачи параметров. Аргумент `*name` соответствует имени вызываемого файла. Если имя файла не начинается с символа `'\'`, то некоторые из приведенных функций позволяют использовать значение переменной окружения среды `PATH`. Аргументы запуска программы `arg` позволяют задать второй аргументы функции `main(int argc, char **arg [, char **env])` вызываемой программы. В зависимости от конкретной функции аргумент передается либо в виде одной строки символов (отдельные аргументы разделяются пробелом, как в командной строке), либо в виде массива строк, заканчивающихся значением `NULL`.

Аргумент `**env` позволяет передать параметры окружения среды `UNIX` в вызываемый процесс.

Использование аргументов функции `main()` позволяет передать в вызываемый процесс параметры из родительского процесса.

```
int system(char *cmd)
```

Функция `system()` вызывает команду, или запускает файл, указанный в аргументе `cmd`. В случае если необходимо вызвать несколько команд за одно обращение к функции `system()`, они пишутся через точку с запятой.

Функция `system()` вызывает `fork()` для создания процесса. Порожденный процесс в свою очередь вызывает `execlp()` и передает аргументы процессу.

Функция `system()` ожидает получение информации от `waitpid()` сигнала о том что порожденный процесс завершился вызовом `_exit()`. В случае неуспешного завершения порожденного процесса функция `system()` возвращает значение `-1`.

```
int pipe(int fifo[2])
```

Функция `pipe()` создает коммуникационный канал между двумя взаимосвязанными процессами. При этом создается служебный файл канала, который автоматически удаляется при закрытии файловых дескрипторов использующихся для работы с каналом.

В большинстве версий UNIX канал может быть только однонаправленным. Дескриптор `fifo[0]` используется для записи данных в канал, а дескриптор `fifo[1]` для чтения данных из канала связи. Использование двух направленных каналов связи, допустимое в отдельных версиях UNIX, обычно не рекомендуется, так как приводит к непереносимости программы. Так же не рекомендуется работать через один канал связи более чем с двумя процессами, так как это может привести к неоднозначным результатам.

В отношении канала связи не допустимо использование функции `lseek()`. Выпорка данных производится только последовательно. Физически данные удаляются из канала сразу после прочтения. Максимальный размер буфера, выделяемого для канала связи определяется значением `PIPE_BUF`.

Для передачи значений `fifo[]` процессу наследнику, или нескольким наследникам обычно функция `pipe()` вызывается до создания процессов. В этом случае, процесс наследник, созданный с помощью функции `fork()`, имеет копию дескрипторов родительского процесса.

При передаче данных по каналу связи, принимающая сторона может проверить отсутствие данных в канале связи по значению, возвращаемому функцией `read()` – функция вернула меньшее число байт, чем Вы заказали прочесть.

Если принимающая сторона, закрыла канал связи, то передающему процессу будет направлено сообщение о том, что канал разорван – `SIGPIPE`.

В случае успешного завершения вызова функция `pipe()` возвращает значение 0, иначе, значение `-1` и устанавливает значение переменной `errno` в код ошибки.

pid_t getpid(void)
pid_t getppid(void)
pid_t getpgrp(void)
pid_t getuid(void)
pid_t geteuid(void)
pid_t getgid(void)
pid_t getegid(void)

Приведенный выше набор функций, позволяет получить атрибуты процесса.

Назначение функций приведено в табл. 2.2

Табл. 2.2. Функции получения атрибутов процессов

Функция	Назначение
pid_t getpid(void)	Получить идентификатор текущего процесса.
pid_t getppid(void)	Получить идентификатор родительского процесса.
pid_t getpgrp(void)	Получить идентификатор группы вызывающего процесса.
pid_t getuid(void)	Получить реальный идентификатор владельца вызывающего процесса.

pid_t geteuid(void)	Получить реальный идентификатор группы вызывающего процесса.
pid_t getgid(void)	Значение атрибута eUID вызывающего процесса (используются для определения прав на доступ к файлам).
pid_t getegid(void)	Значение атрибута eGID вызывающего процесса (используются для определения прав на доступ к файлам).

Например, рассмотрим программу вывода информации о процессе:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main(void)
{
    printf("\n идентификатор текущего процесса: %d", (int) getpid() );
    printf("\n идентификатор родительского процесса: %d", (int) getppid() );
    printf("\n идентификатор группы вызыв. процесса: %d", (int) getpgrp() );
    printf("\n реальный идентификатор владельца: %d", (int) getuid() );
    printf("\n реальный идентификатор группы: %d", (int) getgid() );
    printf("\n значение eUID: %d", (int) geteuid() );
    printf("\n значение eGID: %d", (int) getegid() );
}
int setsid(void)
int setpgid(pid_t pid, pid_t pgid)
int setuid(uid_t uid)
int seteuid(uid_t uid)
int setgid(gid_t gid)
int setegid(gid_t gid)
```

Приведенный выше набор функций позволяет устанавливать отдельные атрибуты процессов. Некоторые атрибуты процессов изменять не допускается, например: идентификаторы самого процесса и родительского процесса. Назначение функций приведено в табл. 2.3.

Табл. 2.3. Функции установки атрибутов процессов

Функция	Назначение
int setsid(void)	Отсоединение процесса. В этом случае процесс, работает не зависимо от родительского процесса.
int setpgid(pid_t pid, pid_t pgid)	Делает процесс лидером новой группы процессов.
int setuid(uid_t uid)	Измерить реальный и эффективный UID на указанный.
int seteuid(uid_t uid)	Изменить эффективный идентификатор владельца процесса.

int setgid(gid_t gid)	Изменяет значение eUID вызывающего процесса.
int setegid(gid_t gid)	Изменяет эффективный идентификатор группы вызывающего процесса.

В случае успешного завершения вызова функция pipe() возвращает значение 0, иначе, значение -1 и устанавливает значение переменной errno в код ошибки.

/* Пример коммуникации процессов при помощи программных каналов
* (трубы, pipes).

* Данная программа превращается в две программы,
* соединенные трубами в таком порядке:

```
*
*      stdout          stdin
*      /----- PIP1 -----> cmd2
*      cmd1 <----- PIP2 -----/
*      stdin          stdout
*/
```

/* файл LOOP_strt.c */

#include <stdio.h>

#define eq(s1,s2) (strcmp(s1,s2) == 0) /* истина, если строки равны */

#define SEP "---" /* разделитель команд при наборе */

main(c, v) char **v;

{

char **p, **q;

int pid;

int PIP1[2]; /* труба cmd1-->cmd2 */

int PIP2[2]; /* труба cmd2-->cmd1 */

if(c==1){

printf("Call: strt cmd1... %s cmd2...\n", SEP);

exit(1);

}

/* разбор аргументов */

v++;

/* в p - аргументы первой команды */

p = v;

while(*v && !eq(*v, SEP))

v++;

*v = NULL;

v++;

/* в q - аргументы второй команды */

q = v;

pipe(PIP1); /* создаем две трубы */

pipe(PIP2); /* PIP[0] - открыт на чтение, PIP[1] - на запись */

```

if( pid = fork()){ /* развилка: порождаем процесс */
/* ПОРОЖДЕННЫЙ ПРОЦЕСС */
    fprintf( stderr, "сын=%s pid=%d\n", p[0], getpid());
    /* перенаправляем stdout нового процесса в PIP1 */
    dup2( PIP1[1], 1 );
    close( PIP1[1] );
    /* канал чтения мы не будем использовать */
    close( PIP1[0] );
    /* перенаправляем stdin из PIP2 */
    dup2( PIP2[0], 0 );
    close( PIP2[0] );
    /* канал записи мы не будем использовать */
    close( PIP2[1] );
    /* начинаем выполнять программу, содержащуюся в
    * файле p[0] с аргументами p (т.е. cmd1)
    */
    execvp( p[0], p );
    /* возврата из сисвызова exec не бывает */
}
else{
/* ПРОЦЕСС-РОДИТЕЛЬ */
    fprintf( stderr, "отец=%s pid=%d\n", q[0], getpid());
    /* перенаправляем stdout в PIP2 */
    dup2( PIP2[1], 1 );
    close( PIP2[1] ); close( PIP2[0] );
    /* перенаправляем stdin из PIP1 */
    dup2( PIP1[0], 0 );
    close( PIP1[0] ); close( PIP1[1] );
    /* запускаем cmd2 */
    execvp( q[0], q );
}
}
}

```

3. Сигналы в операционной системе UNIX

Сигналы инициализируются событиями и посылаются для уведомления о произошедшем событии в процесс. Источникам сигналов могут выступать события в аппаратуре компьютера или в операционной системе. Так же возможен обмен сигналами между процессами. Сигналы в UNIX являются программным аналогом системы прерываний компьютера.

Список сигналов приведен в табл. 3.1. Полный список сигналов используемых в операционной системе UNIX содержится в файле signal.h.

Табл. 3.1. Сигналы в операционной системе UNIX

Сигнал	Событие	Файл core
SIGALRM	Наступление тайм-аута таймера сигналов. Может генерироваться функцией alarm().	Нет
SIGABRT	Аварийное завершение процесса. Генерируется функцией abort().	Да
SIGFPE	Недопустимая математическая операция.	Да
SIGHUP	Разрыв связи с управляющим терминалом.	Нет
SIGILL	Недопустимая машинная команда.	Да
SIGINT	Прерывание процесса, обычно генерируется нажатием Ctrl+C.	Да
SIGKILL	Уничтожение процесса. Может генерироваться с помощью команды kill.	Да
SIGPIPE	Недопустимая запись в канал связи.	Да
SIGQUIT	Выход из процесса. Генерируется при нажатии клавиш Ctrl+\	Да
SIGEGV	Ошибка сегментации.	Да
SIGTERM	Завершение процесса. Может генерироваться с помощью команды kill.	Да
SIGUSR1	Определяется пользователем.	Нет
SIGUSR2	Определяется пользователем.	Нет
SIGCHID	Посылается в родительский процесс при завершении дочернего процесса.	Нет
SIGCONT	Возобновление остановленного процесса.	Нет
SIGSTOP	Остановка процесса.	Нет
SIGTTIN	Остановка фонового процесса в случае запроса данных от управляющего терминала.	Нет
SIGSTP	Остановка процесса нажатием клавиш Ctrl+Z.	Нет
SIGTTOU	Остановка фонового процесса в случае попытки вывести данные на свой управляющий терминал.	Нет

Пользователь может реализовать собственную обработку вызванного сигнала. Многие сигналы, вызывающие завершение процесса формируют файл core, содержащий образ завершеного процесса. Данный образ позволяет проанализировать наличие смысловой ошибки в программе.

Если в процесс посылается много экземпляров сигнала, то пользовательский обработчик обслуживает только первый из них, а остальные сигналы обрабатывает стандартный обработчик. При этом нет гарантии что процесс перехватит все сигналы, кроме того возможно ситуация когда процесс получит повторный экземпляр сигнала, до завершения обработки первого экземпляра. При этом возникает вариант соревнования обработки различных экземпляров сигнала. Это является одним из главных недостатков операционной системы UNIX.

3.2. Функции для работы с сигналами

Функция `signal()`

Прототип функции `signal()`, описанной в `signal.h`, выглядит следующим образом:

```
void *signal (int signal_num, void (*handler)(int))
```

Значение `signal_num` – идентификатор сигнала, возможные значения которых приведены в табл. 11.1. Пользовательский обработчик сигнала указывается через указатель. Аргументом функции является целое число.

Например:

```
#include <signal.h>
#include <stdio.h>
/* Функция обработчик сигнала */
void sign_term(int sig_num)
{
    ...
    printf("\n получен сигнал SIGTERM");
}
void main(void)
{
    ...
    signal(SIGTERM, sign_term);
    ...
}
```

В файле `signal.h` описаны макросы указывающие необходимость игнорировать сигнал или выполнить действие по умолчанию.

```
#define SIG_DFL    void (*) (int) 0
```

```
#define SIG_IGN    void (*) (int) 1
```

Макрос `SIG_DFL` показывает, что необходимо использовать стандартный системный обработчик.

Макрос `SIG_IGN` указывает сигнал, который необходимо проигнорировать.

Пример использования данных макросов приведен ниже:

```
signal(SIGINT, SIG_IGN);
signal(SIGSEGV, SIG_DFL);
```

Функция `signal()` возвращает адрес старого обработчика событий. Его значение используется для возвращения системного обработчика события. Например:

```
void    *old_handler;
/* Устанавливаем режим игнорировать сигнал прерывания процесса по
Cntl+C */
old_handler = signal(SIGINT, SIG_IGN);
...
/* Восстанавливаем стандартный обработчик сигнала */
```

signal(SIGINT, old_handler) ;

Для UNIX System V.3, V4 используется функция sigset() полностью идентичная функции signal(). В последних версиях для задания пользовательских версий появилась функция sigaction().

Сигнальная маска

В операционной системе UNIX можно задать сигнальную маску, блокирующие указанные сигналы для передачи в процесс.

int sigprocmask(int cmd, sigset_t *new_mask, sigset_t *old_mask)

Режим работы функции sigprocmask() задается с помощью аргумента cmd, возможные значения приведены в табл. 3.2.

Табл. 3.2. Аргументы функции sigprocmask()

Аргумент cmd	Выполняемые действия
SIG_SETMASK	Установить новое значение сигнальной маски.
SIG_BLOCK	Добавить сигналы к уже установленной сигнальной маски.
SIG_UNBLOCK	Удалить из сигнальной маски указанные сигналы.

Новое значение сигнальной маски (или изменения в сигнальной маске) содержатся в переменной new_mask, передаваемой в функцию в виде указателя. Старое значение маски сигналов, сохраняется в переменной old_mask, передаваемой так же через указатели.

Структура sigset_t определена в файле signal.h следующим образом:

```
typedef unsigned long sigset_t;
```

В операционной системе UNIX есть целый набор функций для работы переменными масок сигналов.

int sigemptyset(sigset_t *new_mask)

Функция sigemptyset() сбрасывает все сигналы, указанные в маске new_mask.

int sigaddset(sigset_t *new_mask, int signal_num)

Функция sigaddset() устанавливает в маске new_mask сигнал с номером signal_num.

int sigdelset(sigset_t *new_mask, int signal_num)

Функция sigdelset() очищает в маске new_mask сигнал с номером signal_num.

int sigfillset(sigset_t *new_mask)

Функция устанавливает все флаги сигналов в указанной маске.

В случае не успешного выполнения все перечисленные выше функции возвращают значение -1, а в случае успешного выполнения – значение 0.

int sigismember(sigset_t *new_mask, int sign_num)

Функция возвращает значение 1 если в маске new_mask установлен сигнал с номером sign_num. Если флаг указанного сигнала не установлен, то возвращается значение 0.

int sigpending(sigset_t *mask)

Функция sigpending() определяет наличие сигналов, ожидающих данный процесс. Флаги сигналов записываются в переменную mask. В случае не успешного выполнения функция возвращает значение -1.

Функция посылки сигнала в процесс kill()

int kill(pid_t pid, int signal_num)

Функция kill() посылает сигнал с номером, заданном в переменной signal_num, в указанный аргументом pid процесс. Возможные значения аргументов приведены в табл. 3.3.

Табл. 3.3. Аргументы функции kill()

Значения pid	Кому передается сигнал
Положительное число	Процессу с указанным значением идентификатора.
0	Сигнал посылается всем процессам, чей GID совпадает с идентификатором группы вызывающего процесса.
-1	Сигнал посылается во все процессы, реальный владельца которых совпадает с эффективным идентификатором вызывающего процесса. В случае если владелец вызывающего процесса является привилегированным пользователем, то сигнал посылается во все процессы (кроме ядра системы – значения идентификаторов 0 и 1).
Отрицательное число	Сигнал посылается во все процессы, чей GID совпадает с абсолютным значением аргумента pid.

Функция alarm()

unsigned int alarm(unsigned int time_interval)

Функция alarm() посылает процессу сигнал SIGALRM через указанное число секунд. При вызове, функция возвращает число секунд, оставшиеся до посылки сигнала. Если при вызове функции alarm() установить значение time_interval равно 0, то таймер отключается.

Функция alarm() позволяет приостановить выполнения процесса на указанное время. Однако с помощью нее можно устанавливать время только в секундах, а так же можно работать только с одним таймером. Если этого недостаточно, то можно воспользоваться функциями settimet() и gettimet() позволяющими задавать время в миллисекундах и работать сразу с несколькими (до 3-х) таймерами.

int settimet(int which, struct itimerval *val, struct itimerval *old)

int gettimet(int which, struct itimerval *old)

Тип таймера определяется значением аргумента which, возможные значения приведены в табл. 3.4.

Табл. 3.4. Типы таймеров

Значение which	Тип таймера
ITIMER_REAL	Таймер реального времени. По истечении установленного времени посылается сигнал SIGALARM.
ITIMER_VIRTUAL	Таймер времени, затраченного процессом на задачу пользователя. По истечении времени посылается сигнал SIGVTALARM.
ITIMER_PROF	Таймер общего времени, затраченного процессом на задачу пользователя и на системные задачи. По истечении времени посылается сигнал SIGPROF.

Структура itimerval определена следующим образом:

```
struct itimerval
{
    struct timeval    it_interval ; /* интервал таймера          */
    struct timeval    it_value ;   /* текущее значение таймера */
}
```

Структура timeval описана следующим образом:

```
struct timeval
{
    time_t            tv_sec;
    suseconds_t       tv_usec;
}
```

4. Взаимодействие процессов

4.1. Функции работы с сообщениями

Операционная система UNIX позволяет организовать взаимодействие нескольких процессов. Организация взаимодействия процессов для различных версий операционной системы UNIX различно.

В адресном пространстве ядра операционной системы создается таблица очередей сообщений. Работа с сообщениями по структуре аналогична работе с файлами.

Функции работы с сообщениями описаны в модуле ipc.h, msg.h и types.h.

Функция msgget()

int msgget(key_t key, int flag)

Функция msgget() открывает очередь сообщения, идентификатор которой задан значением key. Если значение key равно IPC_PRIVATE, то создается очередь сообщений для эксклюзивного использования вызывающего процесса.

Значение аргумента `flag` указывает на требуемое от функции `msgget()` действие. Возможные значения аргумента `flag` приведены в табл. 4.1.

Табл. 4.1. Значения аргумента функции `msgget()`

Аргумент <code>flag</code>	Действие выполняемое функцией <code>msgget()</code>
0	Возвращает значение дескриптора очереди сообщений с номером <code>key</code> .
<code>IPC_CREAT</code>	Создает новую очередь с указанным идентификатором <code>key</code> . При этом необходимо указывать права доступа, например <code>IPC_CREAT 0644</code> .
<code>IPC_EXCL</code>	Используется совместно с флагом <code>IPC_CREAT</code> , если пользователю необходимо обязательно создать новую очередь сообщений. При наличии этого флага, если очередь уже существует, то функция возвращает значение <code>-1</code> .

В случае не успешного выполнения функция `msgget()` возвращает значение `-1` и устанавливает соответствующее значение переменной ошибки `errno`. Возможные сообщения об ошибке при работе с функцией `msgget()` приведены в табл. 4.2.

Табл. 4.2. Значения ошибок при вызове функции `msgget()`

Значение <code>errno</code>	Причина возникновения ошибки
<code>ENOSPC</code>	Превышен лимит системы – <code>MSGMNI</code>
<code>ENOENT</code>	Нет очереди с указанным номером <code>key</code> и нет указаний на создание новой очереди.
<code>EEXIST</code>	Установлены флаги <code>IPC_EXCL</code> и <code>IPC_CREAT</code> , а очередь сообщений с номером <code>key</code> уже существует.
<code>EACCESS</code>	У вызывающего процесса нет прав на доступ к данной очереди сообщений.

Функция `msgsnd()`

`int msgsnd(int msgfd, char *msg, int len, int flag)`

Сообщение, записанное в поле структуры `msg`, длиной `len` байт посылается в очередь сообщений с дескриптором `msgfd`.

Формат сообщения задается в виде структуры:

```
struct
{
    long  mtype          ; /* тип сообщения          */
    char  mtext[MSGMAX] ; /* буфер для текста сообщения */
}
```

Значение аргумента `flag` указывает тип вызова – блокирующий или не блокирующий. Рекомендуется использовать деблокирующий вызов функции, для этого необходимо установить значение `flag = 0`.

В случае не успешного выполнения операции функция `msgsnd()` возвращает значение `-1`.

Функция `msgrcv()`

`int msgsnd(int msgfd, void *msg, int len, int mtype, int flag)`

Функция `msgsnd()` принимает сообщение типа `mtype` из очереди сообщений с дескриптором `msgfd`. Сообщение записывается в структуру `msg`, аналогичную описанной выше.

```
struct msg_msg
{
    struct list_head m_list;
    long m_type;
    int m_ts;      /* message text size */
    struct msg_msgseg* next;
    void *security;
};
```

Возможные значения типа сообщений `mtype` приведены в табл. 4.3.

Табл. 4.3. Типы сообщений

Значение <code>mtype</code>	Тип сообщения
0	Самое старое сообщение любого типа из указанной очереди.
Положительное число	Принять самое старое сообщение указанного типа.
Отрицательное число	Принять самое старое сообщение из очереди с минимальным номером, меньшим или равным абсолютному значению аргумента <code>mtype</code> .

Максимальный размер принимаемого сообщения определяется значением аргумента `len`. Если принимаемое из очереди сообщение, имеет больший размер, то возвращается код не успешного завершения.

Аргумент `flag` указывает на режим блокировки вызова. Рекомендуется разрешать блокировать процесс, для этого необходимо устанавливать значение `flag` в 0. Для запрещения блокирования сообщения необходимо указать флаг `IPC_NOWAIT`.

В случае успешного завершения функция `msgsnd()` возвращает количество прочитанных из очереди сообщений байт, в случае не успешного завершения возвращается значение `-1`.

Функция `msgctl()`

`int msgctl(int msgfd, int cmd, struct msqid_ds *mbuf)`

Функция позволяет получать и изменять параметры очереди сообщений, заданной дескриптором `msgfd`. Аргумент `cmd` указывает на конкретное действие. Возможные варианты приведены в табл. 4.4.

Табл. 4.4. Аргументы функции msgctl()

Значение cmd	Действие
IPC_STAT	Записать параметры очереди сообщений в структуру mbuf.
IPC_SET	Изменить параметры очереди на указанные в структуре mbuf. Для выполнения данной операции пользователь должен иметь права привилегированного пользователя, быть создателем или владельцем очереди.
IPC_RMID	Удалить очередь из системы. Для выполнения данной операции пользователь должен иметь права привилегированного пользователя, быть создателем или владельцем очереди.

Структура struct msqid_ds описана следующим образом:

```
struct msqid_ds
{
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* первое сообщение в очереди*/
    struct msg *msg_last; /* последнее сообщение в очереди */
    __kernel_time_t msg_stime; /* последнее msgsnd время */
    __kernel_time_t msg_rtime; /* последнее msgrcv время */
    __kernel_time_t msg_ctime; /* последнее change время*/
    unsigned long msg_lbytes; /* Reuse junk fields for 32 bit */
    unsigned long msg_lqbytes; /* ditto */
    unsigned short msg_cbytes; /* кол-во байт в сообщении */
    unsigned short msg_qnum; /* кол-во сообщений в очереди */
    unsigned short msg_qbytes; /* max number of bytes on queue */
    __kernel_ipc_pid_t msg_lspid; /* pid последнего msgsnd */
    __kernel_ipc_pid_t msg_lrpid; /* последний pid */
}

```

В случае успешного выполнения функция возвращает значение 0, иначе возвращается значение -1.

4.2. Пример приложения клиент/сервер

```
/*Функция "эхо", использующая файлы сообщений */
/*файл mes.h******/
#include "commun.h"
#include <sys/ipc.h>
#include <sys/msg.h>
#define MKEY1 8001 /*ключ 1: клиент пишет, сервер
```

```

        читает */
#define MKEY2 8002    /*ключ 2: клиент читает, сер-
        вер пишет */
#define PERM 0600    /*разрешение на доступ */
#define NORMAL 1     /*обычное сообщение */
#define FIN 2        /*завершающее сообщение */
/*файл client.c *****/
#include "mes.h"

clientipc()
{
    int rid, wid; /*идентификаторы файлов сообщений */
/*получение идентификаторов, связанных с ключами */
    wid = msgget((key_t) MKEY1, 0);
    rid = msgget((key_t) MKEY2, 0);
/*обращение к циклу чтения-записи */
    client(rid, wid);
    exit(0);
}

/*функция приема-передачи */
client(rid, wid)
int rid; /*идентификатор файла чтения */
int wid; /*идентификатор файла записи */
{
    struct msgbuf {
        long mtype;
        char mtext[TAI LLEMAXI] } buf;
/*цикл приема-передачи буферов */
    for (i=0; i<nbuf; i++) {
        buf.mtype = NORMAL;
        retour = msgsnd(wid, &buf, lbuf, 0);

        retour = msgrcv(rid, &buf, lbuf, 0, 0);
    }
/*посылка nbuf FIN для остановки сервера */
    buf.mtype = FIN;
    retour = msgsnd(wid, &buf, 0, 0);
}

/*файл server.c *****/
#include "mes.h"

serveuripc()
{

```

```

int rid, wid; /*идентификаторы файлов сообщений */

/*создание файлов сообщений */
rid = msgget((key_t) MKEY1, PERM|IPC_CREAT);
wid = msgget((key_t) MKEY2, PERM|IPC_CREAT);
/*обращение к циклу чтения-записи*/
serveur(rid, wid);
/*удаление файлов сообщений, поскольку при завершении
работы процесса они не уничтожаются */
msgctl(rid, IPC_RMID, 0);
msgctl(wid, IPC_RMID, 0);
}

/*функция приема-передачи */
serveur(rid, wid)
int rid; /*идентификатор файла чтения*/
int wid; /*идентификатор файла записи*/
{
/*обработка, симметричная по отношению к клиенту */
.....
/*остановка, если оказалось, что тип=FIN */
if (buf.mtype == FIN) return;
}

```

5. Семафоры

5.1. Семафоры в операционной системе UNIX

Семафоры в операционной системе UNIX позволяют синхронизировать множество процессов. Семафоры объединяются в группы и хранятся в адресном пространстве ядра операционной системы, в том числе и после завершения создавшего их процесса. В случае удаления семафоров, ядро активизирует все процессы, заблокированы с использованием удаленных семафоров. Функции работы с семафорами и системные переменные определены в sem.h. Имена системных переменных, ограничивающих работу с семафорами приведены в табл. 5.1.

Табл. 5.1. Системные ограничения на работу с семафорами

Имя переменной	Назначение
SEMMNI	Максимальное число наборов семафоров
SEMMNS	Максимальное число семафоров во всех наборах.
SEMMSL	Максимальное число семафоров в одном наборе.
SEMOPM	Максимальное число семафоров в наборе, над которым можно выполнять одновременно действия.

Информация о семафоре хранится в структуре `ipc_perm`, которая описана следующим образом:

```
struct ipc_perm
{
    __kernel_key_t    key;
    __kernel_uid_t    uid;
    __kernel_gid_t    gid;
    __kernel_uid_t    cuid;
    __kernel_gid_t    cgid;
    __kernel_mode_t   mode;
    unsigned short    seq;
};
```

5.2. Функции работы с семафорами

int semget(key_t key, int num_sem, int flag)

Функция `semget()` открывает или создает набор семафоров с идентификатором `key`. Функция возвращает дескриптор для работы с набором семафоров.

Возможные аргумента `flag` представлены в табл. 5.2.

Табл. 5.2. Аргументы `semget()`

Значение <code>flag</code>	Действия
0	Вернуть значение дескриптора указанного набора семафоров.
IPC_CREAT	Создать новый набор семафоров. Число создаваемых семафоров передается в аргументе <code>num_sem</code> . Значение флага IPC_CREAT побитово складывается с необходимыми правами доступа к набору.
IPC_EXCL	Проверка создания новой очереди сообщений. Если создаваемая очередь уже существует, то возвращает значение -1. Используется совместно с флагом IPC_CREAT.

В случае неуспешного выполнения функция возвращает значение -1.

int semop(int semfd, struct sembuf *ptr, int len)

Функция `semop()` позволяет менять значение семафора в наборе с указанным дескриптором `semfd`.

Структура `sembuf`, содержащая информацию о семафоре и об операции над ним, описана следующим образом:

```
struct sembuf
{
    short sem_num ; /* индекс семафора          */
    short sem_op ; /* операция над семафором          */
};
```

```

short sem_flg ; /* флаг операции над семафором */
}

```

Количество структур, на которые указывает указатель `prt`, передается в аргументе `len`.

Операция над семафором определяется значением поля `sem_op`. Возможные значения приведены в табл. 5.3.

Табл. 5.3. Операции над семафорами

Значение поля <code>sem_op</code>	Выполняемая операция
Положительное число X	Увеличить значение указанного семафора на указанное число.
Отрицательное число $-X$	Уменьшить значение семафора на указанную величину.
0	Проверить равенство значения семафора на 0.

В случае успешного выполнения операции функция `semop()` возвращает значение 0, а в случае не успешного выполнения возвращается значение -1 .

int semctl(int semfd, int num, int cmd, union seun arg)

Функция `semctl()` позволяет получать и изменять параметры набора семафоров с указанным значением дескриптора `semfd` и номера семафора `num`. Возможные операции над семафорами, задаваемые значением аргумента `cmd`, приведены в табл. 5.4.

Табл. 5.4. Операции над семафорами

Значение <code>cmd</code>	Операция над семафорами
IPC_STAT	Копировать управляющие параметры в буфер <code>arg</code> .
IPC_SET	Установить новые параметры семафоров заданные в буфере <code>arg</code> .
IPC_RMID	Удалить семафор из системы.
GETALL	Скопировать все значения семафоров в массив.
SETALL	Установить значения семафоров на указанные в массиве.
GETVAL	Возвратить значение семафора с номером <code>num</code> .
SETVAL	Установить значение семафора с номером <code>num</code> на указанное в поле <code>val</code> .
GETPID	Возвратить идентификатор процесса, который последним выполнял операцию с семафором с номером <code>num</code> .
GETNCNT	Возразить количество заблокированных процессов, ожидающих увеличения семафора с номером <code>num</code> .
GETZCNT	Возразить количество заблокированных процессов, ожидающих обращение семафора с номером <code>num</code> в ноль.

Для выполнения операций, связанных с изменением значений параметров семафоров процесс должен иметь права привилегированного пользователя, быть создателем или владельцем семафора.

Объединение `semun` описана следующим образом:

```
union semun
{
    int          val ;      /* значение семафора          */
    struct semid_ds *buf ;  /* управляющие параметры семафоров */
    ushort       *array ;  /* массив значений семафоров      */
}
```

В случае не удачного завершения функция `setctl()` возвращает значение `-1`.

5.3. Пример взаимодействия процессов с использованием семафоров

```
/*Функция "эхо", использующая семафоры */

/*файл mem.h          *****/
#include "commun.h"
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#define MEMKEY 7899    /*ключ, связанный с разделяемой памя-
                      тью */
#define SEMKEY1 9001  /*ключ, связанный с семафором 1 */
#define SEMKEY2 9002  /*ключ, связанный с семафором 2 */
#define PERM 0600     /*разрешение на доступ */

/*файл client.c      *****/
#include "mem.h"

clientipc()
{
    int memid;        /*идентификатор разделяемой памяти */
    int semclient, semserveur; /*идентификаторы семафоров */

    /*получение идентификаторов, связанных с ключами для
    разделяемой памяти*/
    memid = shmget((key_t) MEMKEY, lbuf, 0);
    /*получение идентификаторов, связанных с ключами для
    семафоров */
    semserveur = semget((key_t) SEMKEY1, 1, 0);
    semclient = semget((key_t) SEMKEY2, 1, 0);
    /*обращение к циклу чтения-записи */
```



```

    client(memid, semclient, semserveur);
}

/*функция приема-передачи */
client(memid, semclient, semserveur)
int memid;          /*идентификатор разделяемой памяти*/
int semclient;      /*идентификатор семафора */
int semserveur;     /*идентификатор семафора */
{
    char *pbuf;     /*указатель на начало разделяемой памяти */

/*определение адреса разделяемой памяти */
    pbuf = (char *) shmat(memid, 0, 0);
/*цикл приема-передачи буферов */
    for (i=0; i<nbuf; i++) {
/*ожидание на семафоре клиента (освобождаемого сервером,
    разрешающим клиенту писать) */
        P(semclient);
/*освобождение семафора сервера (разрешение серверу читать)*/
        V(semserveur);
/*при приеме сообщений клиент и сервер меняются ролями */
        P(semclient);
        V(semserveur);
    }
/*для указания серверу на то, что он должен остановиться,
    в первый байт буфера заносится 0 */
    P(semclient);
    *pbuf = 0;
    V(semserveur);
}

/*файл serveur.c      *****/
#include "mem.h"

serveuripc()
{
    int memid;          /*идентификатор памяти */
    int semclient, semserveur; /*идентификатор семафоров */
    union semun {
        int val;
        struct semid_ds *buf;
        ushort *array;
    } semctl_arg; /* структура управления семафором */

/*создание идентификаторов, связанных с ключом для

```

```

разделяемой памяти */
memid = shmget((key_t) MEMKEY, TAILLEMAXI, PERM|IPC_CREAT);
/*создание идентификаторов, связанных с ключами для
семафоров */
semserveur = semget((key_t) SEMKEY1, 1, PERM|IPC_CREAT);
semclient = semget((key_t) SEMKEY2, 1, PERM|IPC_CREAT);
/*инициализация семафоров */
semctl_arg.val = 0;
semctl(semserveur, 0, SETVAL, semctl_arg);
semctl(semclient, 0, SETVAL, semctl_arg);
/*обращение к циклу чтения-записи */
serveur(memid, semclient, semserveur);
/*отказ от разделяемой памяти и семафоров */
shmctl(memid, IPC_RMID, 0);
semctl(semserveur, 1, IPC_RMID, 0);
semctl(semclient, 1, IPC_RMID, 0);
}

```

```

/*функция приема-передачи */
serveur(memid, semclient, semserveur)
int memid; /*идентификатор разделяемой памяти */
int semclient; /*идентификатор семафора */
int semserveur; /*идентификатор семафора */
{
/*обработка, симметричная по отношению к клиенту */
.....
/*выход, если установлен флаг окончания */
if (*pbuf == 0) {
return;
}
}

```

```

/*файл sem.c *****/
#include "mem.h"

```

```

/*функция, реализующая операции над семафорами */
static void semcall(sid, op)
int sid; /*идентификатор */
int op; /*операция */
{
struct sembuf sb;
sb.sem_num=0;
sb.sem_op=op;
sb.sem_flg=0;
semop(sid, &sb, 1);
}

```

```

}

/*установка семафора */
void P(sid)
int sid;      /*идентификатор */
{
    semcall(sid, -1);
}

/*сброс (освобождение) семафора */
void V(sid)
int sid;      /*идентификатор */
{
    semcall(sid, 1);
}

```

6. Интерфейс транспортного уровня

Для взаимодействия процессов в рамках компьютерной сети в операционной системе UNIX используются гнезда. Гнезда предоставляют возможность работать с протоколами TCP, UDP и обращаться непосредственно по IP адресу компьютера.

Протокол TCP используется для создания виртуального канала, а UDP – дейтограммам. Виртуальный канал предусматривает установление соединения между гнездами и последовательную передачу данных, что обеспечивает высокую надежность передачи данных. Соединение типа дейтограмма работают быстрее чем виртуальный канал, но при этом не выполняется условие последовательного передачи данных, что снижает надежность передачи данных. Дейтограммное соединение используется в приложениях, где важно быстроедействие и физический канал связи обеспечивает высокую надежность передачи данных.

6.1. Функции для работы каналами связи

int socket(int domain, int type, int protocol)

Функция socket() для указанного пользователем домена гнездо заданного типа с указанным протоколом.

Аргумент domain и type определяет тип домена и гнезда. Возможные значения аргументов приведены в табл. 6.1.

Табл. 6.1. Аргументы функции socket()

Значение аргумента	Использование
--------------------	---------------

domain	
AF_UNIX	Домены системы UNIX
AF_INET	Internet домены
Значение аргумента type	Использование
SOCK_STREAM	Передача сообщений в виде упорядоченного двунаправленного потока байтов с предварительным установлением соединения.
SOCK_DGRAM	Взаимодействие с помощью дейтаграмм. Сообщения передаются без установления предварительного соединения.
SOCK_SEQPACKET	Передача сообщений в виде упорядоченного двунаправленного потока байтов фиксированной максимальной длины с предварительным установлением соединения.

Значение аргумента `protocol` позволяет выбрать используемый при соединении протокол. Если значение `protocol = 0`, то протокол выбирается операционной системой.

В случае успешного выполнения функция `socket()` возвращает значение дескриптора гнезда. В случае неуспешного выполнения возвращается значение `-1`.

int bind(int sid, struct sockaddr *name, int len)

Функция `bind()` присваивает символическое имя содержащееся в структуре `name`, с указанным размером `len`, гнезду с дескриптором `sid`.

Для операционной системы UNIX имя гнезда задается аналогично имени файла. Структура `sockaddr`, для домена UNIX, описана следующим образом:

```
struct sockaddr
{
short  sun_family ; /* тип операционной системы – AF_UNIX      */
char  sun_path[] ; /* символическое имя                                           */
}
```

В случае домена Internet имя должно состоять из имени компьютера и номера порта. Аргумент `name` в этом случае имеет следующий формат:

```
struct sockaddr_in
{
short      sin_family ; /* тип операционной системы – AF_INET */
u_short   sin_port ;   /* номер порта                       */
struct in_addr sin_addr ; /* имя хоста удаленного компьютера  */
}
```

В случае успешного завершения функция `bind()` возвращает значение `0`, а в случае не успешного выполнения, значение `-1`.

int listen(int sid, int size)

Функция `listen()` создает гнездо со стороны серверного процесса для соединений типа виртуальный канал. Функции передаются – дескриптор гнезда – `sid`, и `size` – максимальное число запросов на установленное соединение ($size \leq 5$).

В случае успешного завершения функция `listen()` возвращает значение 0, а в случае не успешного выполнения, значение `-1`.

`int connect(int sid, struct sockaddr *name, int len)`

Функция `connect()` вызывается клиентским гнездом для установления связи с серверным гнездом. Дескриптор гнезда задается аргументом `sid`. Имя серверного гнезда задается аргументом `name`, имеющим размер `len`.

В случае успешного завершения функция `connect()` возвращает значение 0, а в случае не успешного выполнения, значение `-1`.

`int accept(int sid, struct sockaddr *name, int len)`

Функция `accept()` устанавливает соединение серверного гнезда с подключающимся клиентским гнездом. Дескриптор гнезда с символическим именем, записанным в структуре `name`, размера `len`, передается в аргументе `sid`.

В случае успешного завершения функция возвращает дескриптор нового гнезда, предназначенного для эксклюзивного обмена сообщениями между сервером и клиентом. В случае не успешного выполнения функция возвращает значение `-1`.

`int send(int sid, char *buf, int len, int flag)`

Функция `send()` пересылает сообщение, содержащееся в буфере `buf`, длиной `len` байт в гнездо заданное дескриптором `sid`. Значение аргумента `flag` для обычных сообщений устанавливается в значение 0. Для привилегированных сообщений указывается `MSG_OOB`.

В случае успешного завершения функция возвращает число байт, записанных в канал связи. В случае не успешного завершения функция возвращает `-1`.

`int recv(int sid, char *buf, int len, int flag)`

Функция `recv()` принимает сообщение из гнезда с указанным дескриптором `sid`. Сообщение записывается в буфер `buf`, максимальная длина принимаемого сообщения задается значением `len`. Аргумент `flag` определяет тип принимаемого сообщения. Если установлено значение `MSG_OOB`, то принимаются только привилегированные сообщения. Если `flag = 0`, то принимаются все сообщения. Если значение `flag = MSG_PEEK` то прочитанное сообщение остается в очереди.

В случае успешного завершения функция возвращает число байт, прочитанных из канала связи. В случае не успешного завершения функция возвращает `-1`.

int shutdown(int sid, int mode)

Функция shutdown() завершает работу канала связи, с указанным дескриптором sid. Возможные значения аргумента mode приведены в табл. 6.2.

Табл. 6.2. Режимы закрытия канала связи

Аргумент mode	Режим закрытия канала связи
0	Закрыть гнездо для чтения.
1	Закрыть гнездо для записи.
2	Полностью закрыть гнездо для чтения и для записи.

В случае успешного завершения функция shutdown() возвращает значение 0, а в случае не успешного выполнения, значение -1.

6.2. Работа с виртуальным каналом связи

Работа с виртуальным каналом связи представлена на рис. 6.1.

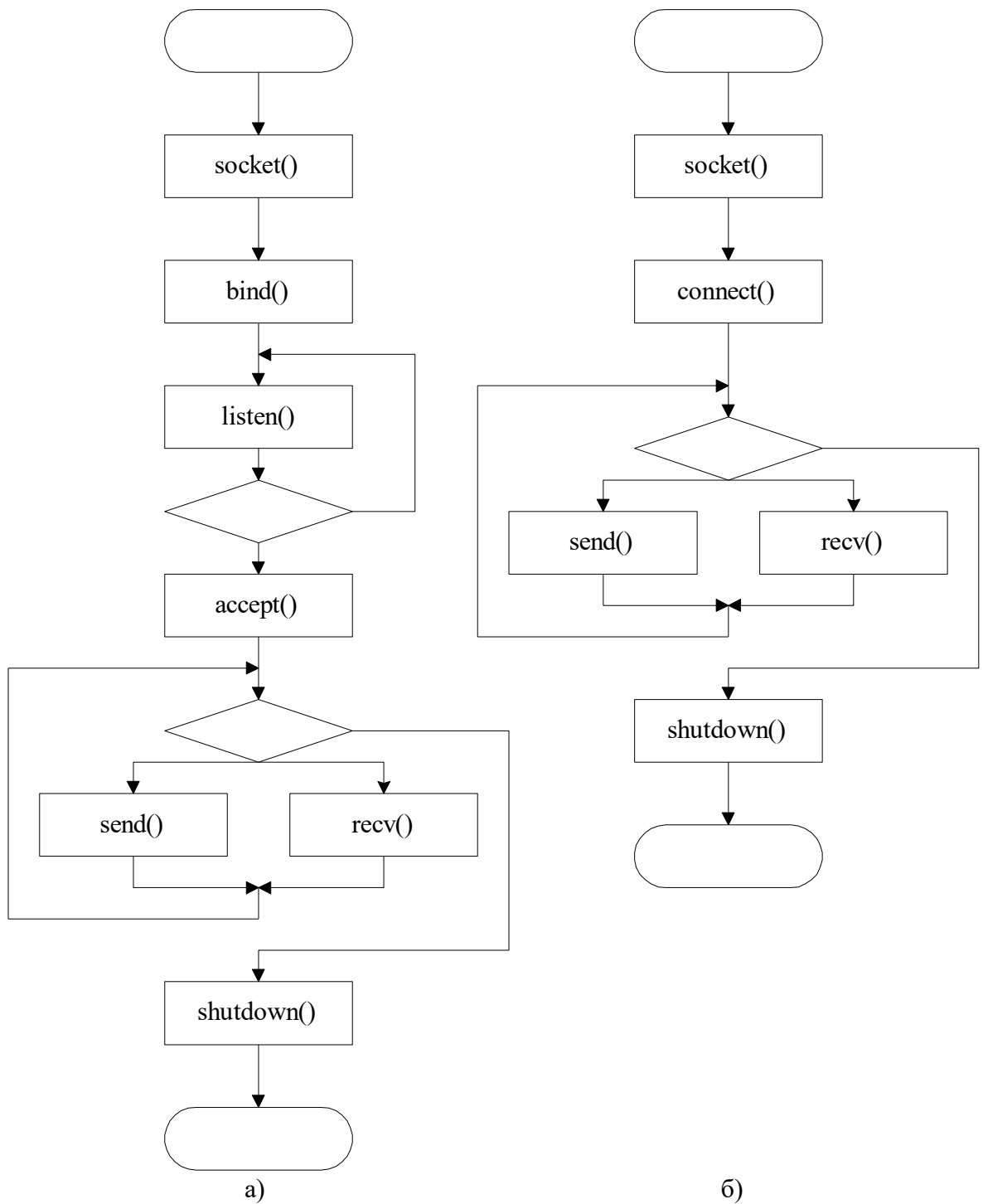


Рис. 6.1. Работа с виртуальным каналом связи
 а) – серверная часть, б) – клиентская часть

6.3. Работа с дейтаграммный канал связи

Работа с дейтаграммным каналом связи представлена на рис. 6.2.

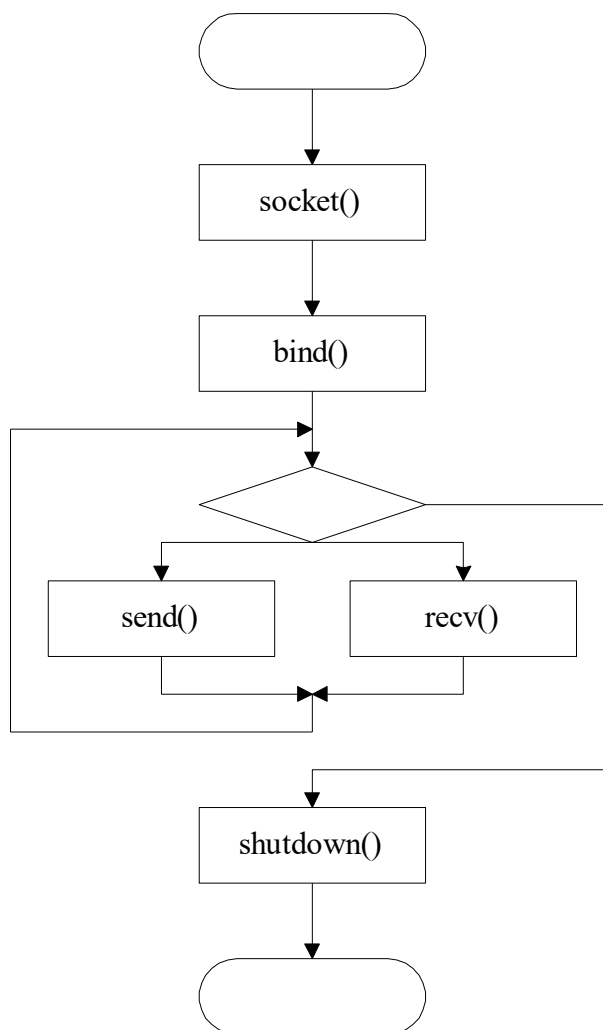


Рис. 6.2. Алгоритм работы процессов через дейтаграмные гнезда

6.4. Пример реализации клиент–серверного приложения

Server.cpp

```

#include <errno.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main( int argc, char **argv )
{
    struct sockaddr_in local;
    SOCKET s;
    SOCKET s1;
    int rc;
    char buf[ 1024 ];

```



```

s = socket( PF_INET, SOCK_STREAM, 0 );

bzero( &local, sizeof( local ) );
local.sin_family = AF_INET;
local.sin_port = htons( 9000 );
local.sin_addr.s_addr = htonl( INADDR_ANY );
if ( bind( s, ( struct sockaddr * )&local,
          sizeof( local ) ) < 0 )
    error( 1, errno, "Error" );
if ( listen( s, NLISTEN ) < 0 )
    error( 1, errno, "listen failed" );
s1 = accept( s, NULL, NULL );

for ( ;; )
{
    rc = recv( s1, buf, sizeof( buf ), 0 );
    if ( rc < 0 )
        error( 1, errno, "recv failed" );
    if ( rc == 0 )
        error( 1, 0, "Client disconnected\n" );
    rc = send( s1, buf, rc, 0 );
    if ( rc < 0 )
        error( 1, errno, "send failed" );
}
}

```

Client.cpp

```

#include <errno.h>
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main( int argc, char **argv )
{
    struct sockaddr_in peer;
    SOCKET s;
    char lin[1024];
    int rc;

    bzero( sap, sizeof( *peer ) );
    peer ->sin_family = AF_INET;
    peer ->sin_addr.s_addr = htonl( INADDR_ANY );

```

```

peer->sin_port = htons( 9000);

s = socket( AF_INET, SOCK_STREAM, 0 );

if ( connect( s, ( struct sockaddr * )&peer,
             sizeof( peer ) ) )
    error( 1, errno, "connect failed" );

rc = recv( s, lin, sizeof( lin ) - 1, 0 );
if(rc > 0)
    printf(“%s”, lin);
else
    printf(“Error”);

exit( 0 );
}

```

7. Удаленные вызовы процедур

Механизм удаленных вызовов процедур позволяет вызывать функции на удаленном компьютере. Механизм удаленного вызова процедур рассмотрен в (____).

7.1. Компилятор grscgen

Компилятор grscgen обеспечивает разработку RPC приложений.