С.Л. РОМАНОВ

ПРОГРАММИРОВАНИЕ ДЛЯ ОПЕРАЦИОННОЙ СИСТЕМЫ UNIX

Министерство образования и науки Российской Федерации
Балтийский государственный технический университет «Военмех»
Кафедра информационных систем и компьютерных технологий

С.Л. РОМАНОВ

ПРОГРАММИРОВАНИЕ ДЛЯ ОПЕРАЦИОННОЙ СИСТЕМЫ UNIX

Учебное пособие

Под редакцией Н.Н. Смирновой

Санкт-Петербург
2011

УДК 004.451.9(075.8)
Р69

Романов, С.Л.

Р69 Программирование для операционной системы Unix: учебное пособие / С.Л. Романов; под ред. Н.Н. Смирновой; Балт. гос. техн. ун-т. – СПб., 2011. – 74 с.

ISBN 978-5-85546-624-9

Рассматриваются организация многозадачности и основные способы межпроцессного взаимодействия в программмах для операционных систем семейства Unix с помощью системных вызовов и библиотечных функций.

Предназначено для студентов всех специальностей, изучающих дисциплину «Операционные системы» и выполняющих лабораторные работы по одноименному курсу.

УДК 004.451.9(075.8)

Р е ц е н з е н т ы: канд. техн. наук, доц. каф. НЗ БГТУ *В.Ю. Емельянов*; канд. техн. наук, доц. каф. распределённых интеллектуальных систем Института международных образовательных программ *Е. В. Потехина*

Утверждено
редакционно-издательским
советом университета

ISBN 978-5-85546-624-9

П р е д и с л о в и е

В данном учебном пособии рассмотрены основные механизмы реализации многозадачности и межпроцессного взаимодействия в операционной системе (ОС) Unix. Все примеры программ написаны на языке Си.

Для изучения материала требуются знания на уровне продвинутого пользователя ОС Unix (работа в командной строке, знание механизма прав доступа, управление процессами).

Практически все рассматриваемые здесь системные вызовы и библиотечные функции при возникновении ошибки помещают ее код в переменную errno; это действие специально не отмечается при рассмотрении вызовов и функций. Значения кодов ошибок с описаниями можно найти в руководстве, вызываемом командой *man*. Приведенные описания системных вызовов и библиотечных функций достаточно полные; некоторые, редко используемые возможности не описаны. Для получения наиболее подробного описания следует использовать команду man; в некоторых случаях при вызове man нужно указывать номер секции руководства перед искомым термином. Например, команда *man kill* выдаст информацию о команде *kill* (секция 1), а *man 2 kill* – о системном вызове *kill* (секция 2). Полезная для программиста информация содержится в секциях 2 (системные вызовы), 3 (функции библиотеки libc), 4 (устройства), 5 (форматы файлов и протоколы), 7 (соглашения, макросы, определения типов и т.п.).

Во всех приведенных примерах предполагается, что текущий каталог указан в пути *PATH*. Если это не так, то его следует

добавить, непример командами *PATH=$PATH*:. ; *export PATH*. В качестве программы-оболочки применяется bash. Используется вызов Си-компилятора cc, стандартный в Unix. В Linux- и BSD-системах вместо него можно использовать gcc.

Пособие состоит из семи разделов и приложения. В первом разделе рассмотрены основы многозадачности в Unix, создание дочерних процессов, использование кода завершения процесса, во втором – передача данных между процессами через неименованный канал и организация конвейера из нескольких процессов с перенаправлением потоков ввода-вывода. В третьем разделе рассмотрена работа с именованными каналами. Четвертый раздел посвящен использованию разделяемой памяти. В пятом рассматривается механизм семафоров и особенности его реализации в Unix, в шестом детально описана работа с потоковыми сокетами: локальными и IP-сокетами. В седьмом разделе рассмотрены механизм сигналов и иерархия процессов в Unix и основы написания процессов-демонов.

1. ОСНОВЫ МНОГОЗАДАЧНОСТИ

1.1. Основные понятия. Вызов fork()

Все системы семейства Unix являются многозадачными, поддерживающими параллельное исполнение многих процессов. ***Процессом*** называется программа в момент выполнения. ***Образ процесса*** состоит из кода (текста) программы, инициализированных и неинициализированных данных и стека. ***Контекстом процесса*** называется информация, необходимая для перевода процесса в режим выполнения. В Unix-системах контекст процесса состоит из состояния пользовательского уровня (адресное пространство процесса и окружение времени выполнения) и состояния уровня ядра (параметры планирования, управление ресурсами, идентификация процесса). Контекст процесса включает в себя всю информацию, используемую ядром для предоставления сервисов процессу. Пользователь может создавать процессы, управлять их исполнением и получать извещение об изменении состояния исполнения процесса.

Каждый процесс может создавать (порождать) другие процессы, называемые ***дочерними процессами***. Дочерние процесссы, в свою очередь, могут также порождать новые процессы. В результате образуется дерево процессов, в корне которого находится процесс **init** – процесс инициализации системы. Кроме того, каждый процесс принадлежит к определенной группе и сессии. Подробнее иерархия процессов в ОС Unix рассмотрена в подразд. 7.3.

Информация обо всех процессах, идущих в системе, хранится в таблице процессов. В частности, для каждого процесса хранятся следующие данные:

1. Уникальный номер процесса – **PID** (Process IDentifier – идентификатор процесса) – целое положительное число, однозначно идентифицирующее процесс в системе. Процесс init всегда имеет PID, равный 1.
2. Номер группы процесса **PGID** – определяет принадлежность процесса к определенной группе.
3. Номер родительского процесса **PPID** (Parent PID). У процесса init значение PPID всегда равно 0.

4. Реальный номер пользователя, запустившего процесс, – **UID**.

5. Эффективный (действующий) номер пользователя – **EUID**. Он может отличаться от UID при исполнении файла с установленным SETUID-битом или после исполнения системного вызова setuid().

6. Реальный номер группы – **GID**.

7. Эффективный (действующий) номер группы – **EGID**, который может отличаться от GID при исполнении файла с установленным SETGID-битом или после исполнения системного вызова setgid().

Номера UID, EUID, GID, EGID используются для проверки прав доступа процесса к файлам и некоторым другим ресурсам. Отметим, что программы, исполняемые файлы которых имеют установленный SETUID (SETGID)-бит прав доступа, называют ***setuid-программами (setgid-программами)***.

Для создания нового процесса в ОС Unix предназначен системный вызов fork(), объявленный в заголовочном файле unistd.h следующим образом:

```
pid_t     fork(void);
```

Тип pid_t – это тип идентификатора процесса, объявленный как int. Системный вызов fork() порождает ***копию процесса-родителя***. Созданный процесс отличается от исходного только значениями PID и PPID. При удачном завершении fork() возвращает процессу-родителю номер (PID) созданного процесса, а дочернему процессу – значение 0; при неуспехе fork() возвращает значение - 1. Отметим, что открытые файлы (т.е. их дескрипторы) ***наследуются*** дочерним процессом от родительского.

Рассмотрим пример простой программы, демонстрирующей работу fork(). Программа создает два процесса: родительский и дочерний. Каждый из них выводит свой номер PID, а дочерний процесс – еще и номер своего родителя (PPID).

```
/* Файл fork1.c  -  Два процесса */
#include <unistd.h>
#include <stdio.h>

void main(void)
{
 int p;
```

```
printf("До fork(): PID = %d\n", getpid());
 if ( (p = fork()) < 0 ) printf("Ошибка fork()\n");
 else
       if (p!=0)
       /* Родительский  процесс */
       {
            printf("Родительский процесс:PID=%d\n", getpid());
            wait(); /* Ожидание завершения дочернего процесса */
            printf("Родительский процесс завершен\n");
       }
       else    {
       /* Дочерний процесс */
            printf("Дочерний процесс : PID=%d, PPID=%d," \
                                                  getpid(), getppid());
            }
}
```

Если скомпилировать программу и запустить ее, увидим на экране следующую информацию:

```
До fork(): PID = 395
Родительский процесс: PID=395
Дочерний процесс: PID=396, PPID=395
Родительский процесс завершен
user@localhost$
```

Числовые значения при разных запусках могут отличаться от указанных. Обратите внимание на равенство номеров PID родительского и PPID дочернего процессов. В программе использованы системные вызовы getpid() и getppid(), объявленные в unistd.h так:

```
pid_t        getpid(void);
pid_t        getppid(void);
```

Они возвращают номер самого вызвавшего процесса (PID) и номер его родительского процесса (PPID).

Также в программе использован системный вызов wait(), блокирующий вызвавший процесс до завершения одного из его потомков. Системный вызов wait() имеет аргумент и возвращамое значение, не использованные в данной программе. Более подробно wait() будет рассмотрен далее. В данной программе он используется, чтобы гарантировать, возвращение в оболочку bash после завершения всех процессов. Что может произойти, если не

использовать wait()? Рассмотрим следующий пример – программу, создающую два процесса. Затем родительский процесс выводит сообщение "Родительский процесс" дважды с интервалом в 1 секунду, а дочерний – сообщение "Дочерний процесс, PPID=nnn" пять раз также с интервалом в 1 секунду:

```
/* fork2.c - Два процесса без ожидания завершения дочернего */
#include <unistd.h>
#include <stdio.h>

void main(void)
{
 int i,p;

 if ( (p=fork())<0 ) printf("fork() error\n");
 else
        if ( p!=0 )
          /* Родительский процесс */
          for(i=0;i<2;i++) {
            printf("Родительский процесс\n"); sleep(1);
          }
        else
          /* Дочерний процесс */
          for(i=0;i<5;i++) {
            printf("Дочерний процесс, PPID=%d\n", getppid());
            sleep(1);
          }
}
```

Запустив программу, увидим на экране следующий результат:

```
Родительский процесс
Дочерний процесс, PPID=208
Родительский процесс
Дочерний процесс, PPID=208
user@localhost$ Дочерний процесс, PPID=1
Дочерний процесс, PPID=1
Дочерний процесс, PPID=1
```

Курсор останется на пустой строке внизу. Поясним, что произошло. После запуска программы и создания дочернего процесса процессы начали выводить свои сообщения; bash в это время ждал завершения запущенной им программы. После завершения родительского процесса bash вышел из ожидания и вывел своё приглашение; однако дочерний процесс всё ещё

работал и вывел три своих строки, одна из которых оказалась выведена сразу за приглашением bash'а. Эта программа наглядно демонстрирует полную независимость исполнения процессов. Обратите внимание, что вначале родительским процессом дочернего считается процесс с номером 208 (настоящий родитель), а после завершения последнего – процесс с номером 1 (init). Это общая закономерность. В ОС семейства Unix, когда родительский процесс завершается раньше дочернего, дочернему присваивается номер PPID=1.

1.2. Запуск новой программы. Семейство exec()

В нашем примере все процессы порождались одной и той же программой. Естественно, для полноценной реализации многозадачности должна иметься возможность запуска в качестве нового процесса другой программы. Такая возможность реализуется применением вместе с fork() семейства функций exec(). Они также описаны в unistd.h следующим образом:

```
int execl(const char *PATHNAME, const char *ARG, ...);
int execlp(const char *NAME, const char *ARG, . . . );
int execle(const char *PATHNAME, const char *ARG,  ... ,
                                char * const ENVP[]);
int execv(const char *PATHNAME, char * const ARGV[]);
int execvp(const char *NAME, char * const ARGV[]);
int execve(const char *PATHNAME, char *const ARGV[],
                                char * const ENVP[]);
```

Из этих функций только execve() является системным вызовом, а остальные – реализованными на его основе библиотечными функциями. Для программиста эта разница практически несущественна.

Любая функция exec() ***заменяет образ текущего процесса на новый***, загружая и запуская указанный исполнимый файл. Запущенная программа наследует PID и открытые файлы; посланные процессу сигналы сбрасываются, устанавливаются исходные (как по умолчанию) варианты реакции на сигналы. Разница между функциями семейства exec() заключается в аргументах. PATHNAME – это имя файла, возможно, с указанием пути. Файл ищется по обычному правилу. NAME – это просто имя файла, поиск которого выполняется так же, как команды в

оболочке (если не содержит символа '/', то поиск выполняется в каталогах, перечисленных в переменной PATH, иначе – по обычному правилу). В execl(), execle(), execlp() аргумент ARG – это строки-аргументы, передаваемые программе. Как обычно, нулевой (первый по порядку) аргумент – это путь/имя самой программы, а последним в списке должен быть NULL, например:

```
execl("/home/user/prg",   "/home/user/prg",   "Arg#1",   NULL);
        Сама программа     Арг. 0              Арг. 1     Конец списка
```

Аргументы могут передаваться и в виде указателя на массив строк const char *ARGV[] – в функциях execv(), execvp(), execve(). Этот массив должен содержать строки, аналогичные вышеприведенному списку, например: ARGV[0]="/home/user/prg", ARGV[1]="Argument#1", ARGV[2]=NULL. Этот способ передачи удобно использовать при передаче вызываемой программе аргументов вызвавшей программы.

Последний из аргументов exec() - ENVP – это указатель на массив строк-переменных окружения: const char *ENVP[]. Этот массив также должен иметь последним элементом NULL. Обычно ENVP используется для передачи (наследования) строк окружения из вызвавшей программы (из 3-го аргумента функции main()). Можно передать и "пустое" окружение, указав в качестве значения ENVP NULL.

Особый случай загрузки программы функцией exec() – указание в качестве исполняемой программы ***скрипта.*** Скрипт должен содержать первую строку вида

```
#!Интерпретатор [аргументы]
```

где Интерпретатор – это полное имя (с путем) интерпретатора, которому предназначен скрипт. Происходит запуск указанного интерпретатора, как по команде

```
Интерпретатор [аргументы] Имя_скрипта
```

(то есть загружается и исполняется код программы-интерпретатора, который, в свою очередь, исполняет указанный скрипт.)

Отметим, что все функции exec() при успешном исполнении *не возвращают результат*, поскольку происходит замещение кода, содержащего вызов exec(). При неуспешном

завершении (например, если не найден файл-программа или отсутствуют права на его исполнение) exec() возвращает результат -1. Проверять его нет необходимости: если код старого процесса продолжает исполняться после exec(), значит, произошла ошибка.

Рассмотрим использование функции семейства exec() - execl(). Программа осуществляет вызов стандартной утилиты ls с аргументами – опцией "-l" и именем каталога "/". Ниже приведен текст программы.

```
/* Файл exec_ls.c */
#include <stdio.h>
#include <unistd.h>

void main(void)
{
        printf("Запускаем ls ...\n");
        execl("/bin/ls", "/bin/ls", "-l", "/", NULL);
        printf("Ошибка exec()!\n");  exit(1);
}
```

После запуска программы увидим на экране сообщение программы о запуске ls и содержимое корневого каталога в расширенном формате, например:

```
Запускаем ls ...
drwxr-xr-x 2 root   root  4096    2010-07-27 15:44  bin
drwxr-xr-x 2 root   bin   4096    2010-09-26 20:15  boot
drwxr-xr-x 2 root   root  40960   2010-10-24 16:00  dev
```

и т.д.

В случае ошибки произойдет завершение программы с выводом сообщения "Ошибка exec()!". В этом можно убедиться, например, заменив первый аргумент execl() на "/bin/not_ls".

1.3. Код завершения программы. Процессы-зомби

Рассмотрим более подробно системный вызов wait(). Он объявлен в sys/wait.h как

```
pid_t        wait(int *exit_status)
```

Системный вызов wait() блокирует вызвавший процесс до тех пор, пока не завершится один из его потомков. Возвращаемое

значение функции – это номер завершившегося процесса. Кроме того, wait() может вернуть значение exit_status – *статус завершения* процесса. Статус завершения – это целое число, содержащее код возврата и информацию о причине завершения процесса. Код возврата – некоторое целое число, указанное при завершении программы (например, с помощью вызова exit(код_возврата)) и обычно используемое для указания успешности или неуспешности решения процессом его задачи. Например, такие утилиты, как cp или mv, возвращают код 0 при успешном завершении и при ошибке – некоторое ненулевое значение, зависящее от причины ошибки.

Статус завершения (exit_status) несет, кроме кода возврата, дополнительную информацию о причине завершения процесса. Для извлечения этой информации служат следующие макросы (аргументом всех этих макросов является *значение* кода завершения, а не указатель на него!):

1. WIFEXITED(exit_status) – возвращает "истину", если процесс завершился нормально, т.е. с помощью exit(), _exit() либо возвратом из функции main().
2. WEXITSTATUS(exit_status) – возвращает значение кода возврата (фактически, 8 младших битов кода завершения). Этот макрос можно вычислять, только если WIFEXITED() вернул значение "истина".
3. WIFSIGNALED(exit_status) – возвращает истину, если процесс завершился в результате получения сигнала (подробнее сигналы рассматриваются в разд. 4).
4. WTERMSIG(exit_status) – возвращает номер сигнала, вызвавшего завершение процесса. Этот макрос можно вычислять, если WIFSIGNALED() вернул значение "истина".

Рассмотрим пример. Первая программа запускает вторую, которая запрашивает у пользователя некоторое целое число и возвращает его в качестве кода завершения, после чего первая программа выводит полученный код.

```
/* Файл retcode_main.c */

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
```

```
int main()
{
        int p,code;

        if((p=fork())<0) { printf("Ошибка fork()!\n"); return 1; }
         if(p==0) { /* В дочернем процессе запустим retcode. */
             execl("retcode", "retcode", NULL);
             printf("Ошибка запуска 'retcode'!\n");
             return 123; /* Код завершения дочернего процесса */
        }
        else { /* Родительский процесс */
             wait(&code);
             if(WIFEXITED(code))
                  printf("Дочерний процесс завершен с кодом=%d\n",
                                  WEXITSTATUS(code));
             else  if(WIFSIGNALED(code))
                  printf("\nДочерний процесс завершен сигналом %d\n",
                                  WTERMSIG(code));
             else  printf("\nДочерний процесс завершен\n");
             return 0;
        }
}
```

Скомпилируем эту программу командой

```
cc -o retcode_main retcode_main.c
```

и получим исполнимый файл retcode_main.

Вторая программа:

```
/* Файл retcode.c */
#include <stdio.h>
#include <unistd.h>

main()
{
        int code;

        printf("Введите код (0..255):");
        scanf("%d", &code);
        exit(code); /* Можно написать и "return code;" */
}
```

Скомпилируем эту программу командой cc -o retcode retcode.c и получим исполнимый файл retcode. Теперь запустим программу retcode_main:

```
bash$ retcode_main
Введите код (0..255): 36
Дочерний процесс завершен с кодом=36
```

Таким образом, код, введенный в программе retcode, передан в вызвавшую программу retcode_main. Запустим программу еще раз:

```
bash$ retcode_main
Введите код (0..255):
```

Не вводя код, переключимся на другую консоль (с тем же пользовательским именем) или вызовем еще одно окно консоли (например, xterm в X Window). В новой консоли узнаем, какие процессы исполняются данным пользователем:

```
bash$ ps –a
      PID      TTY        TIME  CMD
      . . . . . .
      347      pts/1    00:00:00retcode_main
      348      pts/1    00:00:00retcode
      . . . . . .
```

(Показана не вся выводимая ps информация; значения PID процессов retcode_main и retcode могут быть другими.) Теперь уничтожим процесс retcode командой kill 348. Вернувшись на первую консоль, увидим следующее:

```
Дочерний процесс завершен сигналом 15
```

Сигнал 15 – это сигнал SIGTERM, посылаемый по умолчанию командой kill.

Изменим обе программы. В родительском процессе сделаем паузу перед wait(), а дочерний будет возвращать постоянное значение 100. Новые программы:

```
/* Файл retcode_main1.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    int p,code;
```

```
    if((p=fork())<0) { printf("Ошибка fork()\n"); return 1; }
    if(p==0) {
        execl("retcode","retcode",NULL); /* Запуск "retcode" */
        printf("Ошибка запуска 'retcode'\n");
        return 123; /* Код завершения */
    }
    else { /* Родительский процесс */
        printf("Нажмите Enter для продолжения ...\n");
        getc(stdin); /* Просто ждем нажатия Enter */
        wait(&code);
        if(WIFEXITED(code))
            printf("Дочерний процесс завершен с кодом=%d\n",
                WEXITSTATUS(code));
        else if(WIFSIGNALED(code))
            printf("\nДочерний процесс завершен сигналом %d\n",
                WTERMSIG(code));
        else printf("\nДочерний процесс завершен\n");
        return 0;
    }
}
```

Вторая программа очень простая:

```
/* Файл retcode1.c */

int  main(void)
{
        return 100;
}
```

Скомпилируем обе программы командами `cc -o retcode_main retcode_main1.c` и `cc -o retcode retcode1.c` и запустим первую:

```
bash$ retcode_main
Нажмите Enter для продолжения ...
```

Переключимся теперь на другую консоль и посмотрим состояние процессов:

```
bash$ ps –a
PID      TTY         TIME  CMD
. . . . . .
347      pts/1     00:00:00retcode_main
348      pts/1     00:00:00retcode <defunct>
. . . . . .
```

(Как и ранее, показана только информация, относящаяся к нашим процессам.) Дочерний процесс (retcode) имеет пометку "<defunct>". Это означает, что он является процессом-зомби.

Процесс-зомби – это завершившийся процесс, информация о котором все еще хранится системой. В таком виде он будет существовать до тех пор, пока его родительский процесс не прочитает код его завершения (например, вызовом wait()) или до завершения самого родительского процесса.

Проверим это. Вернемся на первую консоль и нажмем <Enter>, продолжая исполнение процесса retcode_main (родительского). Получим сообщение:

```
Дочерний процесс завершен с кодом=100
```

Командой ps можно убедиться, что оба наших процесса отсутствуют.

Кроме wait(), существует аналогичный системный вызов waitpid(). Он также объявлен в sys/wait.h :

```
pid_t waitpid(pid_t pid, int *exit_status, int options);
```

Данный системный вызов блокирует (приостанавливает) текущий процесс до завершения его процесса-потомка, указанного аргументом pid:

1) pid < -1 – ожидание завершения любого дочернего процессса, чей номер группы процесса (PGID) равен абсолютному значению аргумента pid;

2) pid = -1 – ожидание завершения любого дочернего процесса аналогично вызову wait();

3) pid=0 – ожидание завершения любого дочернего процесса, чей PGID равен PGID вызвавшего процесса (т.е. входящего в ту же группу процессов);

4) pid>0 – ожидание завершения дочернего процесса с номером, равным pid.

Второй аргумент (exit_status) используется для получения статуса завершения процесса, аналогично аргументу вызова wait().

Третий аргумент – это опции. Может быть указан 0 или несколько опций, объединенных с помощью побитового ИЛИ ('|'),

из которых отметим одну. Опция WNOHANG вызывает немедленный возврат (без ожидания) системного вызова со значением pid завершившегося потомка или 0, если таких нет. Возвращаемыми значениями waitpid() могут быть: pid завершившегося процесса, -1 при возникновении ошибки, 0 в вышерассмотренном случае.

Контрольные вопросы

1. Что такое процесс? Как он идентифицируется в ОС семейства Unix?
2. Какое действие выполняет системный вызов fork()?
3. Для чего служат функции семейства exec()? В чём разница между этими функциями?
4. Какова последовательность действий для запуска некоторой программы в качестве дочернего процесса?
5. Что произойдёт, если в качестве первого аргумента exec*() – имени файла запускаемой программы указать имя скрипта?
6. Что такое код завершения программы? Как программа задаёт свой код завершения?
7. Каково назначение системного вызова wait()?
8. Что такое процесс-зомби?

2. НЕИМЕНОВАННЫЕ КАНАЛЫ

2.1. Создание и использование канала

Каналы в ОС Unix – это механизм передачи данных между процессами. Работа с каналом, в принципе, подобна работе с файлами. Вначале канал создаётся с помощью системного вызова pipe(), описанного в unistd.h как int pipe(int fd[2]); Этот системный вызов при успешном завершении создает два файловых дескриптора: fd[0] и fd[1] и возвращает 0. При ошибке pipe() возвращает значение -1.

Через созданный канал возможна передача данных. Дескриптор fd[0] служит для чтения, а fd[1] – для записи. Операции чтения и

записи выполняются с помощью read() и write() аналогично работе с файлами. При этом система при необходимости осуществляет блокировку процессов. Например, если процесс пытается прочитать больше данных, чем имеется в канале, то он блокируется до тех пор, пока другой процесс не запишет достаточное количество данных в канал или не закроет "пишущий" конец канала. Закрытие канала осуществляется вызовом close(), как и для файлов.

В принципе, один процесс может выполнять операции и чтения, и записи с одним и тем же каналом (передавая данные самому себе), однако на практике применяется вариант, когда один процесс записывает данные в канал (используя дескриптор fd[1]), а второй читает их из канала (используя дескриптор fd[0]). Неиспользуемые дескрипторы (fd[0] в пишущем и fd[1] в читающем процессе) закрывают.

Работа с каналом демонстрируется следующей программой:

```
/* Файл pipedemo.c */
#include <stdio.h>
#include <unistd.h>

#define MAXLEN 100

main()
{
	int fd[2];
	int p, l;
	char s[MAXLEN];

	/* Создаем канал */
	if (pipe(fd)<0) { printf("Ошибка pipe()!\n"); return 1; }

	/* Процесс "раздваивается". Созданный канал наследуется
	   обоими процессами. */
	if( (p=fork()) < 0 ) {
	   printf("Ошибка fork() error\n"); return 1;
	}

	if(p==0) { /* Дочерний процесс - писатель */
	   close(fd[0]);
	   printf("Введите строку:");
	   if(fgets(s, MAXLEN, stdin)!=NULL) {
```

```
            l = strlen(s) + 1; /* длина строки + '\0' */
            /* Запись s[] в канал */
            if( write(fd[1], s, l) != l ) {
                printf("Ошибка записи в канал!\n");
                return 1;
            }
        }
        return 0; /* Конец дочернего процесса */
    }
    else { /* Родительский процесс - читатель */
        close(fd[1]);
        if( ( l = read(fd[0], s, MAXLEN) ) < 0 ) {
            printf("Ошибка чтения из канала!\n");
            return 1;
        }
        if(l)
            printf("Строка=%s\n", s);
        else
            printf("Конец файла!\n");
        return 0; /* Конец родительского процесса */
    }
}
```

Программа порождает два процесса, один из которых запрашивает у пользователя некоторую строку и записывает ее в канал, а второй читает строку из канала и выводит ее на экран. Пример работы:

```
bash$ cc -o pipedemo pipedemo.c
bash$ pipedemo
Введите строку: ABCDEF
Строка=ABCDEF
```

Если в ответ на приглашение "Введите строку" нажать Ctrl+D ("конец файла" в Unix), то на экране увидим сообщение «Конец файла!»

Вместо текста через канал могут передаваться любые данные. Число используемых каналов практически ограничено только системой, т.е. процессы могут создавать несколько каналов и обмениваться данными через них (например, по одному каналу передавать данные из 1-го процесса во 2-й, а по другому – из 2-го в 1-й).

2.2. Перенаправление потоков ввода-вывода. Конвейер

Одним из мощных механизмов, используемых в ОС Unix, является возможность перенаправлять потоки ввода-вывода. Любой процесс, запускаемый в ОС Unix, имеет три стандартных потока: ввода (stdin), вывода (stdout), ошибок (stderr). Первые два из них часто называют просто "стандартный ввод" и "стандартный вывод". Программа-оболочка предоставляет следующие возможности:

1) перенаправление из файла на стандартный ввод программы,

2) перенаправление стандартного вывода программы в файл,

3) перенаправление стандартного потока ошибок в файл,

4) связывание стандартного вывода одной программы со стандартным вводом другой (конвейер).

Более подробно перенаправление рассматривается в руководствах по работе из командной строки. Здесь мы рассмотрим, как реализовать перенаправление в программах на Си.

Все виды перенаправления можно выполнить с помощью системного вызова dup2(), объявленного в unistd.h как

```
int   dup2(int oldfd, int newfd);
```

Системный вызов dup2(oldfd, newfd) заставляет дескриптор newfd указывать на тот же файл, что и oldfd. Если файл newfd был открыт, то он вначале закрывается. Иными словами, дескриптор newfd связывается с oldfd. Если вызов dup2() завершается неудачно, то он возвращает значение -1; при успехе возвращается значение нового дескриптора (newfd=oldfd). Дескрипторы стандартных потоков равны: stdin=0, stdout=1, stderr=2. Теперь, если у нас имеются дескрипторы некоторых файлов fin и fout, открытых для чтения и записи соответственно, то возможны следующие действия:

1) dup2(fin, 0) – перенаправить из файла fin на стандартный ввод;

2) dup2(fout, 1) – перенаправить стандартный вывод в файл fout;

3) dup2(fout, 2) – перенаправить стандартный поток ошибок в файл fout.

Используя в качестве файловых дескрипторов дескрипторы созданного ранее канала, можно реализовать конвейер.

Рассмотрим организацию конвейера на следующем примере. Допустим, нужно вывести список первых 10 файлов каталога /bin в расширенном формате. Это можно реализовать следующей командной строкой: ls -l /bin | head (утилита head в данном случае выводит 10 первых строк из текста, полученного со стандартного ввода). Организуем такой же конвейер программой на языке Си:

```
/* Файл pipedemo2.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

main()
{
    int p[2];  /* Дескрипторы конвейера */
    int newp1, newp2; /* PID дочерних процессов */

    /* Создание канала */
    if (pipe(p)<0) {
        printf("Ошибка создания канала\n");
        return 1;
    }

    /* Создание 1-го дочернего процесса */
    if ( (newp1=fork()) < 0 ) {
        printf("Ошибка 1-го fork()\n");
        return 2;
    }
    if (newp1==0) { /* 1-й дочерний процесс */
        close(p[0]); /* Закрываем дескриптор "чтения канала" */
        dup2(p[1], 1); /* Связываем канал со станд. выводом */
        /* Запускаем "ls -l /bin" */
        execlp("ls", "ls", "-l", "/bin", NULL);
        printf("Ошибка запуска ls\n");
        return 3;
    }
    else { /* Родительский процесс */
        if ( (newp2=fork()) < 0 ) {
            printf(" Ошибка 2-го fork()\n");
            return 2;
        }
```

```
        if(newp2==0) { /* 2-й дочерний процесс */
            close(p[1]); /* Закрываем дескриптор "записи" */
            dup2(p[0], 0); /* Связываем канал со станд. вводом */
            execlp("head", "head", NULL); /* Запускаем "head" */
            printf("Ошибка запуска head\n");
            return 3;
        }
    }
    /* Родительский процесс */
    /* Очевидно, что последним из потомков завершится второй
        (head), поэтому ждем завершения именно его. Статус
        завершения не используется (указан NULL).
        Опции waitpid() также 0. */
    waitpid(newp2,NULL,0);
}
```

Эта программа демонстрирует в упрощенном виде механизм, используемый программами-оболочками для создания конвейеров.

Контрольные вопросы

1. Что такое неименованный канал? Как он создается?
2. Что делает системный вызов dup2()?
3. Какие стандартные потоки ввода-вывода имеет программа в Unix?
4. Как выполнить перенаправление стандартных потоков?
5. Что такое конвейер? Как организовать конвейер в Си-программе?

3. ИМЕНОВАННЫЕ КАНАЛЫ (FIFO)

Именованные каналы – это еще один механизм передачи данных в ОС Unix. Их также называют **FIFO** (First In – First Out – первый вошел – первый вышел), что отражает принцип функционирования канала: данные, записанные в канал первыми, будут и прочитаны из канала первыми.

Работа с именованными каналами аналогична неименованным каналам, рассмотренным в предыдущем разделе. Возможны также запись в канал и чтение из канала. Обычно один процесс осуществляет запись, а другой чтение. Однако имеется и существенное

различие: именованный канал "виден" в файловой системе как файл специального типа, поэтому для работы с ним применимы обычные операции: open(), close(), read(), write() и т.д. Нет необходимости передавать из одного процесса в другой дескриптор открытого канала, как в случае организации конвейера. Вместо этого взаимодействующим процессам достаточно знать имя канала. Следовательно, через именованные каналы возможна простая передача данных между процессами, не являющимися родственниками.

Создать именованный канал можно с помощью системного вызова mkfifo() (рассмотрен далее) или команды mkfifo <имя канала>, например:

```
bash$ mkfifo fifo1
bash$ ls -l mkfifo
prw-r--r--   1   user   users   0   2010-11-09 14:04 fifo1
```

Буква "p" (pipe) в начале строки указывает тип созданного файла – FIFO pipe. Рассмотрим теперь работу канала. Отдадим команду bash$ cat fifo1. Поскольку канал изначально пуст, cat заблокируется при попытке чтения канала. Теперь из другой консоли (или другого окна терминала), находясь в том же каталоге, где создан канал fifo1, отдадим команду bash$ echo "Hello" > fifo1.

Команда echo поместила в канал текст "Hello" и признак конца файла, после чего команда cat разблокировалась, вывела текст на экран и завершилась. В этом легко убедиться, вернувшись на первую консоль (или окно терминала). Этот пример показывает, что передача данных через именованные каналы доступна не только Си-программам, но и скриптам, и простым командам.

В программе на языке Си создание именованного канала осуществляется с помощью системного вызова mkfifo(), объявленного в sys/stat.h:

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

Вызов mkfifo() создает именованный канал с именем pathname. Права доступа (режим доступа) к созданному файлу устанавливаются равными mode & ~umask, где umask – маска прав доступа процесса (обычно 022). При успешном завершении

mkfifo() возвращает значение 0, при ошибке – значение -1. После создания именованного канала с ним работают, как с обычным файлом. Именованные каналы автоматически не уничтожаются; для их уничтожения используется вызов unlink(), как и для обычных файлов.

Продемонстрируем работу с именованными каналами следующими двумя программами. Первая программа (сервер) после запуска создает именованный канал с именем fifo0, открывает его для чтения и ждет поступления текста по каналу. Поступающий текст программа-сервер выводит на экран; при поступлении сообщения "END" программа завершается и удаляет файл именованного канала. Вторая программа (клиент) открывает именованный канал и выводит в него текстовые сообщения, задаваемые в качестве аргументов. Программа-сервер приведена ниже:

```
/* Файл fifo_server.c */
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <string.h>

#define FIFO_NAME "fifo0"
#define EXIT_MESSAGE "END\n"
#define BUFLEN 100

main()
{
      FILE *f;
      char str[BUFLEN];

      if(mkfifo(FIFO_NAME,0600)<0) {
          printf("Ошибка создания FIFO '%s'\n", FIFO_NAME);
          return 1;
      }

      if( (f=fopen(FIFO_NAME, "r")) == NULL ) {
          printf("Невозможно открыть '%s' - файл удален?\n", \
                    FIFO_NAME);
          return 2;
      }
      for(;;) {
          if( fgets(str, BUFLEN, f) != NULL ) {
              printf("\nПРИНЯТО:%s\n", str);
              if(strcmp(str, EXIT_MESSAGE)==0) break;
          }
      }
```

```
    fclose(f);
    /* Удалим fifo-файл */
    if( unlink(FIFO_NAME) <0 )
        printf("Ошибка удаления FIFO-файла!\n");
    return 0;
}
```

Скомпилируем ее командой bash$ cc -o server fifo_server.c. Программа-клиент:

```
/* Файл fifo_client.c */
#include <stdio.h>

#define FIFO_NAME "fifo0"
#define BUFLEN 100

main(int argc, char *argv[])
{
    FILE *f;
    int i;

    /* Открытие FIFO-файла. Режим "r+" для проверки
        существования файла. */
    if( (f=fopen(FIFO_NAME, "r+")) == NULL) {
        printf("Файл  FIFO '%s' не найден\n", FIFO_NAME);
        return 1;
    }

    for(i=1;i<argc;i++) {
        fputs(argv[i], f); /* Записываем i-й аргумент */
        fputc('\n', f); /* и "конец строки" после него */
    }
    fclose(f);
    return 0;
}
```

Скомпилируем ее командой bash$ cc -o client fifo_client.c. Запустим программу-сервер в фоновом режиме командой server &. Если выдается сообщение о невозможности создания FIFO, удалите в текущем каталоге файл fifo0 и повторите команду. Можно убедиться в существовании FIFO-файла, набрав команду ls -l fifo0. На экране увидим следующее (вместо user и users могут быть другие имена):

```
prw------- 1 user users0 2010-11-09 19:13          fifo0
```

Теперь можно проверить работу программ, запуская программу-клиент и наблюдая выводимые сообщения:

```
bash$ client Alpha Beta Gamma
ПРИНЯТО: Alpha
ПРИНЯТО: Beta
ПРИНЯТО: Gamma
bash$ client END
ПРИНЯТО: END
[1]+       Done                 server
```

Отсылать сообщения "серверу" можно и так:

```
bash$ echo "Message" > fifo0
```

Контрольные вопросы

1. Что такое именованные каналы? Как они создаются?
2. Виден ли именованный канал в каталоге файлов?
3. Будет ли существовать именованный канал после завершения создавшей его программы?
4. Каковы различия в сфере использования именованных и неименованных каналов?

4. РАЗДЕЛЯЕМАЯ ПАМЯТЬ

Разделяемая память (shared memory) – это некоторая область памяти, совместно используемая несколькими процессами. Использование разделяемой памяти позволяет осуществить передачу данных между процессами с большой скоростью.

Последовательность работы с разделяемой памятью следующая. Вначале один из процессов выделяет некоторый объем памяти, называемый *сегментом***.** В системе каждый сегмент разделяемой памяти идентифицируется номером – *ключом сегмента.* Для каждого сегмента определены владелец и группа и имеются права доступа, задаваемые девятью битами (по три бита для владельца, группы и остальных пользователей) аналогично правам доступа к файлу (право исполнения не используется). Внутри процесса для доступа к сегменту используется другой номер – *идентификатор сегмента*. Для выделения нового сегмента или получения доступа к сегменту с известным ключом служит системный вызов shmget().

Перед использованием выделенного сегмента процесс должен выполнить *подключение сегмента* с помощью системного вызова shmat(). Системный вызов shmat() возвращает указатель на начало подключенного сегмента, который можно использовать обычным образом.

Завершив работу с сегментом, процесс *отсоединяет* его системным вызовом shmdt(). Когда дальнейшая необходимость в выделенном сегменте исчезает, его следует *удалить*, используя системный вызов shmctl(). Сегмент не будет удален автоматически, даже если все работавшие с ним процессы завершились!

Рассмотрим системные вызовы для работы с разделяемой памятью более подробно.

Системный вызов shmget() описан в sys/shm.h следующим образом:

```
int     shmget(key_t key, size_t size, int shm_flags);
```

Он возвращает идентификатор сегмента, связанного со значением ключа key или -1 при ошибке. Новый сегмент размером size, округленным вверх до значения, кратного PAGE_SIZE, создается в следующих случаях:

1) указано значение key, равное IPC_PRIVATE. Система сама выберет уникальное значение ключа;

2) значение key не равно IPC_PRIVATE, сегментов с таким значением key не существует и среди флагов указан флаг IPC_CREAT (т.е. IPC_CREAT & shm_flags не равно 0).

Для вновь созданного сегмента устанавливаются владелец и группа равными эффективным (действующим) UID и GID текущего процесса и права доступа к сегменту как 9 младших битов аргумента shm_flags. Если ни первое, ни второе не выполнено, то система пытается предоставить уже имеющийся сегмент с ключом key, проверяя наличие такого сегмента и права доступа к нему для вызвавшего процесса.

Значение аргумента shm_flags – это объединенные с помощью побитового ИЛИ (|) права доступа к создаваемому сегменту и два возможных флага:

IPC_CREAT – "создать сегмент",

IPC_EXCL – используется вместе с IPC_CREAT и заставляет вызов вернуть "ошибку", если сегмент уже создан.

Системный вызов shmat() выполняет *подключение* сегмента. Он также объявлен в sys/shm.h :

```
void        *shmat(int id, void *address, int flags);
```

При успешном завершении возвращается адрес сегмента – указатель на сегмент с идентификатором id; при ошибке – указатель (void *)-1. Адрес для подключения сегмента в адресное пространство вызвавшего процесса может быть выбран системой автоматически, если указать аргумент address, равный NULL. Если этот аргумент не равен NULL, то сегмент подключается по заданному адресу, определяемому из значения address и флагов flags по некоторому правилу. Первый способ (с address=NULL) является предпочтительным и обеспечивает лучшую переносимость программ. Значение аргумента flags – это объединенные по операции "побитовое ИЛИ" (|) флаги, из которых отметим SHM_RDONLY. Если этот флаг указан, то сегмент подключается только для чтения и для успешного подключения процессу достаточно иметь только право чтения данного сегмента; по умолчанию же сегмент подключается для чтения и записи, и для успешного подключения процессу необходимо иметь права чтения и записи данного сегмента.

Для *отключения* сегмента, ранее подключенного по адресу (указателю) address, служит системный вызов shmdt(), объявленный в sys/shm.h:

```
int  shmdt(void * address);
```

При успешном завершении он возвращает 0, при ошибке – -1.

Для получения информации о сегменте, установки владельца, группы и прав доступа к сегменту, а также для удаления сегмента служит системный вызов shmctl(), описанный в sys/shm.h так:

```
int  shmctl(int id, int cmd, struct shmid_ds *buf);
```

При успешном завершении действия он возвращает 0, при ошибке – значение -1. Аргумент id – это идентификатор сегмента, над которым производится операция; cmd указывает команду – операцию над сегментом и может принимать следующие значения:

1) IPC_STAT – получить информацию о сегменте;

2) IPC_SET – установить "свойства" сегмента;

3) IPC_RMID- удалить сегмент;

4) SHM_LOCK – запретить выгрузку сегмента (т.е. заблокировать его в оперативной памяти);

5) SHM_UNLOCK – разрешить выгрузку сегмента.

Две последние команды используются только в ОС Linux.

Разберем действие команд более подробно. IPC_STAT и IPC_SET позволяют получить или изменить свойства (права доступа и т.п.) сегмента. Информация о сегменте передается через структуру struct shmid_ds, наиболее важные поля которой следующие:

```
struct shmid_ds {
        struct ipc_perm shm_perm; /* См. описание ниже */
        int shm_segsz;              /* Размер сегмента в байтах */
        . . . . /* Остальная часть структуры не рассматривается */
};
```

Входящая сюда структура ipc_perm описана так:

```
struct ipc_perm {
        key_t __key; /* Ключ сегмента */
        unsigned short uid; /* UID владельца сегмента */
        unsigned short gid; /* GID владельца сегмента */
        . . . . . .
        unsigned mode; /* 9 мл. бит - права доступа к сегменту */
        . . . . . .
};
```

Команда IPC_STAT заносит в структуру *buf информацию о сегменте; вызывающий ее процесс должен иметь право чтения заданного сегмента. Команда IPC_SET изменяет UID и GID владельца сегмента и права доступа к сегменту на указанные в структуре *buf; для ее выполнения вызывающий процесс должен быть создателем или владельцем сегмента или иметь права суперпользователя (root).

Команда IPC_RMID отмечает удаляемый сегмент. В качестве указателя *buf в системном вызове shmctl() указывается NULL. Отмеченный сегмент будет уничтожен после последнего его отсоединения (shmdt()).

Команда SHM_LOCK (SHM_UNLOCK) запрещает (разрешает) выгрузку сегмента (т.е. принадлежащих ему страниц виртуальной памяти) на диск (в swap-раздел или файл). Запрет выгрузки сегмента ускоряет доступ к нему.

Отметим следующие особенности работы с разделяемой памятью. После fork() дочерний процесс наследует подключенные сегменты разделяемой памяти. После вызова exec() все подключенные сегменты расширенной памяти отключаются от процесса. После завершения процесса вызовом exit() все подключенные сегменты также отключаются от процесса. Как после exec(), так и после exit() сегменты только отключаются, но *не уничтожаются*.

Из командной строки информацию о выделенных сегментах разделяемой памяти можно получить командой ipcs -m.

Рассмотрим в качестве примера передачу от одного процесса другому некоторой символьной строки. Передача осуществляется с помощью информационной структуры my_info, имеющей два поля: string – буфер для передаваемой строки и flag – флаг, ненулевое значение которого указывает, что строка передана. Информационная структура передается через сегмент разделяемой памяти со статическим ключом, указанным в файле defs.h, подключаемом к обеим программам. Тексты программ:

```
/* Файл defs.h */
/* Ключ сегмента */
#define SEG_KEY 0x12345678

/* Макс. длина передаваемой строки */
#define MAXSTRLEN  100

typedef struct {
        int  flag;
        char    string[MAXSTRLEN];
}       my_info;

/* Файл sh_main.c */
#include <stdio.h>
#include <sys/shm.h>

#include "defs.h"
```

```
main()
{
        int shmid; /* ID сегмента */
        my_info *ptr;  /* Указатель на сегмент */
        struct shmid_ds shm_info;  /* Информация о сегменте */
        int size; /* Размер выделяемой памяти */

        /* Создаем сегмент разделяемой памяти. 0600 - права
            доступа: чтение(400) + запись(200) для владельца. */
        shmid = shmget(SEG_KEY, sizeof(my_info),  \
                                IPC_CREAT | IPC_EXCL | 0600)
        if (shmid < 0 ) {
            printf("Ошибка создания сегмента!\n"); return 1;
        }
        printf("Создан сегмент с ID = %d\n", shmid);
        /* Подключим сегмент. Адрес сегмента помещается в ptr.*/
        ptr = (my_info *) shmat(shmid, NULL, 0);
        if ( ptr == (my_info *) -1 ) {
            printf("Ошибка присоединения сегмента!\n");
            goto destroy_segment;
        }
        /* Получим информацию о созданном сегменте */
        if ( shmctl(shmid, IPC_STAT, &shm_info) < 0 ) {
            printf("Ошибка shmctl(IPC_STAT)!\n");
            goto destroy_segment;
        }
        size = shm_info.shm_segsz;
        printf("Размер %d байт, требовалось %d байт\n", \
                size, sizeof(my_info));
        printf("Ключ сегмента=0x%lx\n", shm_info.shm_perm.__key);
        /* Сбрасываем флаг передачи строки */
        ptr -> flag = 0;
        /* Ждем, пока строка не передана вторым процессом */
        while( ptr->flag == 0 ) ;
        printf("Принята строка:%s\nНажмите Enter\n",ptr->string);
        /* Отключаем сегмент */
        shmdt(ptr);

destroy_segment: /* Уничтожение сегмента */
        shmctl(shmid, IPC_RMID, NULL);
        return 0;
}

/* Файл sh_send.c */
#include <stdio.h>
#include <sys/shm.h>

#include "defs.h"
```

```
main()
{
	my_info *ptr;/* Указатель на сегмент */
	int shmid;	/* ID сегмента */

	/* Получаем ID сегмента с известным ключом SEG_KEY */
	shmid = shmget(SEG_KEY, sizeof(my_info), 0600);
	if( shmid < 0 ) {
	    printf("Ошибка shmget() - сегмент не был создан?\n");
	    return 1;
	}
	printf("Найден сегмент. ID=%d\n", shmid);
	/* Подключение сегмента */
	ptr = shmat(shmid,NULL, 0);
	if( ptr == (my_info *) -1 ) {
	    printf("Ошибка подключения сегмента!\n");
	    return 1;
	}
	/* Вводим с клавиатуры строку */
	printf("Введите строку:");
	fgets(ptr->string, MAXSTRLEN, stdin);
	/* Устанавливаем флаг "строка передана" */
	ptr->flag = 1;
	/* Отключаем сегмент разделяемой памяти */
	shmdt(ptr);
	return 0;
}
```

Скомпилируем обе программы командами cc -o sh_main sh_main.c и cc -o sh_send sh_send.c.

Теперь запустим sh_main в фоновом режиме. На экране увидим следующее (значение ID сегмента может быть другим):

```
bash$ sh_main &
Создан сегмент с ID = 655363
Размер 104 байт, требовалось 104 байт
Ключ сегмента 0x12345678
```

Нажмите <Enter>, чтобы получить приглашение оболочки. Посмотрим теперь информацию о разделяемой памяти, набрав команду ipcs –m:

```
------ Shared Memory Segments -------
key            shmid      owner      perms      bytes      nattch status
. . . . . .
0x12345678  655363     user       600        104        1
. . . . . .
```

Смысл полей key и shmid очевиден. Поле owner показывает владельца сегмента, поле bytes – размер сегмента, nattch – количество подключений. Запустим теперь вторую программу и в ответ на её приглашение введем любую строку (например, QWERTY):

```
bash$ sh_send
Сегмент найден. ID=655363
Введите строку: QWERTY
bash$ Принята строка: QWERTY
Нажмите Enter
[1]+ Done                    sh_main
bash$
```

Здесь выведенный программой sh_main текст "Принята строка ..." наложился на приглашение оболочки. Обратите внимание на равенство ID сегмента в обеих программах. С помощью команды ipcs -m можно убедиться, что ранее созданный сегмент уничтожен. Если запустить программу, создающую сегмент (sh_main), а затем уничтожить ее (например, с помощью kill), то можно убедиться (командой ipcs -m) в том, что сегмент остался существовать после завершения создавшего его процесса. Чтобы уничтожить сегмент из командной строки, используйте команду ipcrm -m <ID сегмента>.

В данных программах использован *статический ключ* для доступа к сегменту. Статическое назначение ключа исключает необходимость передавать значение ключа между процессами, но накладывает некоторые ограничения на программы. Например, в нашем примере невозможно запустить два экземпляра программы sh_main. Второе ограничение – перед использованием некоторого значения ключа программист должен убедиться, что это значение уже не используется другими программами в системе. Решить данные проблемы можно с помощью *динамического выделения ключа сегмента*, однако при этом возникает задача передать значение идентификатора сегмента из процесса – создателя сегмента остальным процессам, использующим данный сегмент. Значение идентификатора сегмента может быть передано самыми различными способами, например через аргументы программы или через переменные окружения. Рассмотрим вариант предыдущего примера, использующий динамическое выделение ключа. Для передачи значения идентификатора сегмента используются аргументы второй программы:

```
/* Файл defs2.h */
/* Максимальная длина передаваемой строки */
#define MAXSTRLEN  100

typedef struct {
        int  flag;
        char    string[MAXSTRLEN];
}       my_info;


/* Файл sh_main2.c */
#include <stdio.h>
#include <sys/shm.h>

#include "defs2.h"

main()
{
        int shmid; /* ID сегмента */
        my_info *ptr;  /* Указатель на подключенный сегмент */
        struct shmid_ds shm_info;
        int size; /* Размер сегмента */

        /*   Создание сегмента разделяемой памяти с динамически
             выделенным ключом */
        shmid = shmget(IPC_PRIVATE, sizeof(my_info), \
                        IPC_CREAT | IPC_EXCL | 0600);
        if ( shmid  < 0) {
             printf("Ошибка создания сегмента!\n"); return 1;
        }
        /* Подключение сегмента. */
        ptr = (my_info *) shmat(shmid, NULL, 0);
        if ( ptr == (my_info *)(-1) ) {
             printf("Ошибка подключения сегмента!\n");
             goto destroy_segment;
        }
        if ( shmctl(shmid, IPC_STAT, &shm_info) < 0 ) {
             printf("Ошибка shmctl(IPC_STAT) !\n");
             goto destroy_segment;
        }
        size = shm_info.shm_segsz;
        printf("Создан сегмент размером %d байт, "\
             "требовалось %d байт\n", size, sizeof(my_info));
        /* Сбрасываем флаг "строка передана" */
        ptr -> flag = 0;
        printf("ID созданного сегмента = %d\n", shmid);
        /* Ждем, пока строка не передана */
        while( ptr->flag == 0 ) ;
```

```
        printf("Принята строка: %s\nНажмите Enter\n",
                                        ptr->string );

        /* Отключение сегмента */
        shmdt(ptr);

destroy_segment:
        shmctl(shmid, IPC_RMID, NULL);
        return 0;
}


/* Файл sh_send2.c */
#include <stdio.h>
#include <sys/shm.h>

#include "defs.h"

main(int argc, char *argv[])
{
        my_info  *ptr;
        int shmid;

        if (argc<2) {
            printf("Использование: sh_main2   ID_сегмента\n");
            return 1;
        }
        /* Получение ID сегмента из 1-го аргумента. */
        shmid = atoi(argv[1]);
        /* Подключение сегмента */
        ptr = shmat(shmid,NULL, 0);
        if( ptr == (my_info *) -1 ) {
            printf("Невозможно подключить сегмент с ID=%d\n",
                                                    shmid);
            return 2;
        }
        printf("Введите строку:");
        fgets(ptr->string, MAXSTRLEN, stdin);
        /* Устанавливаем флаг "строка передана" */
        ptr->flag = 1;
        /* Отключаем сегмент */
        shmdt(ptr);
        return 0;
}
```

Скомпилируем обе программы командами cc -o sh_main2 sh_main2.c и cc -o sh_send2 sh_send2.c.

Теперь запустим sh_main2 в фоновом режиме. На экране увидим следующее (значение ID сегмента может быть другим):

```
bash$ sh_main2 &
Размер 104 байт, требовалось 104 байт
ID созданного сегмента = 294915
```

Нажмите <Enter>, чтобы получить приглашение оболочки. Посмотрим теперь информацию о разделяемой памяти командой ipcs -m. Увидим на экране:

```
------ Shared Memory Segments -------
key            shmid      owner   perms    bytes   nattch    status
. . . . . .
0x00000000  294915    user      600      104     1
. . . . . .
```

Смысл полей выводимой таблицы тот же, что и в предыдущем примере. Значение ключа (key) при динамическом получении всегда будет равно 0, поэтому для доступа к сегменту используется значение идентификатора сегмента.

Запустим теперь вторую программу (sh_send2) с аргументом, равным ID сегмента, выведенным первой программой (в данном случае 294915). В ответ на приглашение введем любую строку (например, ASDFGH):

```
bash$ sh_send2 294915
Введите строку: ASDFGH
bash$ Принята строка: ASDFGH
Нажмите Enter
[1]+ Done                 sh_main2
bash$
```

Данный пример демонстрирует работу с разделяемой памятью с динамическим выделением ключа сегмента. Можно запустить несколько экземпляров sh_main2 и убедиться, что каждый из них создаст свой сегмент разделяемой памяти. Затем можно несколько раз вызвать sh_send2, указывая значения ID сегментов для разных экземпляров sh_main2.

Контрольные вопросы

1. Что такое разделяемая память? Какова последовательность работы с ней?
2. Как создаются и подключаются сегменты разделяемой памяти?
3. Что такое статический и динамический ключ сегмента разделяемой памяти? В чем преимущества и недостатки использования каждого из них?
4. Как уничтожаются сегменты разделяемой памяти?
5. Какой командой можно получить информацию о существующих сегментах разделяемой памяти?
6. Как из командной строки уничтожить сегмент разделяемой памяти?
7. Можно ли запускать рассмотренные программы sh_main (sh_main2) и sh_send (sh_send2) от имен разных пользователей? Если нет, то почему? Что нужно изменить в программах, чтобы это было возможно?

5. СЕМАФОРЫ

Семафор – это механизм организации совместного доступа нескольких процессов к общим ресурсам; семафор имеет некоторое целочисленное значение (рассмотрим здесь вариант, когда значение семафора только положительное (беззнаковое)). Также имеется очередь ожидающих на семафоре процессов. С семафором возможны две основные операции: **P-операция** или "закрытие" семафора и **V-операция** или "открытие" семафора.

P-операция выполняет следующее:

- если значение семафора было равно 0, то вызвавший операцию процесс блокируется и помещается в очередь ожидающих на семафоре процессов;
- если значение семафора не равно 0, то оно уменьшается на 1 и вызвавший процесс продолжается.

V-операция выполняет следующее:

- если очередь ожидающих процессов не пуста, то из нее выбирается и разблокируется один процесс;
- иначе значение семафора увеличивается на 1.

Важно, что любая операция над семафором должна выполняться как *единое непрерывное (атомарное)* действие.

Применяются семафоры следующим образом. Перед доступом к разделяемому ресурсу каждый процесс вызывает P-операцию

над семафором. В результате один из процессов получает доступ к ресурсу, а остальные блокируются на семафоре. После завершения работы с ресурсом процесс вызывает V-операцию, разрешая таким образом другим процессам получить доступ к разделяемому ресурсу.

В ОС Unix/Linux работа с семафорами реализована системными вызовами semget(), semop(), semctl(), semtimedop() и имеет много дополнительных возможностей по сравнению с рассмотренным выше простым семафором; в частности, можно изменить семафор на значение, отличающееся от +1/-1, и совершить несколько операций над набором семафоров одним системным вызовом. Далее рассматриваются основные возможности механизма семафоров в ОС Unix/Linux и реализация с их помощью простого семафора.

Семафоры в Unix объединяются в наборы; одиночный семафор можно реализовать как «набор из одного семафора». В программе набор семафоров при проведении операций над ним идентифицируется его номером – *идентификатором набора семафоров*. Вначале один из процессов, использующих данный набор семафоров, создает его с помощью вызова semget(). Остальные процессы, использующие данный набор семафоров, также используют semget() для получения доступа к тому же набору. Далее для работы с набором семафоров используются вызовы semctl() – для установки начальных значений семафора и semop() или semtimedop() – для реализации P/V-операций. После завершения работы набор семафоров уничтожается с помощью системного вызова semctl().

Для каждого набора семафоров определены владелец, группа и права доступа (9 бит) аналогично файловой системе ОС Unix.

Системный вызов semget() возвращает идентификатор набора семафоров, связанного с заданным ключом key. Он объявлен в sys/sem.h следующим образом:

```
int  semget(int key, int nsems, int semflags);
```

При успешном завершении возвращается номер набора (неотрицательное число), при ошибке – значение -1.

Аргумент key – это ключ, идентифицирующий данный набор семафоров в системе. Может быть указано либо конкретное значение, либо IPC_PRIVATE для создания нового уникального набора. Аргумент nsems – это количество семафоров в создаваемом наборе; если semget() используется для получения доступа к уже существующему набору семафоров, то его значение произвольно (можно указать 0). Третий аргумент semflags – это набор флагов, объединенных с помощью битового ИЛИ ('|'). В девяти младших битах semflags указываются права доступа к создаваемому набору семафоров. Указание флага IPC_CREAT означает "создать набор семафоров", а флага IPC_EXCL вместе с ним – "вернуть значение "ошибка", если набор с таким ключом уже был создан".

С каждым семафором набора связана следующая структура данных:

```
struct {
    unsigned short semval;      /* Значение семафора */
    unsigned short semzcnt;     /* К-во процессов, ожидающих
                                   нулевого значения семафора */
    unsigned short semncnt;     /* К-во процессов, ожидающих
                                   увеличения значения семафора */
    pid_t sempid;               /* Процесс, исполнивший послед-
                                   нюю операцию с семафором */
}
```

Структуры данных, связанные с каждым набором семафоров, *не инициализируются* при создании набора вызовом semget(). Для инициализации используется вызов semctl().

Системный вызов semctl() используется для установки значений структур данных, связанных с семафорами, для чтения значений этих структур и для удаления набора семафоров. Он объявлен в sys/sem.h следующим образом:

```
int  semctl(int semid, int semnum, int cmd, ... );
```

Он может иметь три или четыре аргумента. При ошибке возвращается значение -1, при успешном завершении – некоторое неотрицательное значение, зависящее от указанной операции cmd.

Аргумент semid – это идентификатор набора семафоров, над которым производится операция. Третий аргумент cmd определяет операцию, проводимую над набором семафоров.

Аргумент semnum задает номер семафора в наборе при выполнении некоторых операций. Семафоры в наборе нумеруются, начиная с 0.

Операция IPC_RMID вызывает *уничтожение* набора семафоров и разблокировку процессов, заблокированных на них. Значение аргумента semnum игнорируется (может быть любым). Выполнить эту операцию может только создатель или владелец набора семафоров, а также процесс, имеющий права суперпользователя (root).

Следующие рассматриваемые операции позволяют прочитать или изменить значения полей структур данных семафоров. Естественно, для чтения или изменения вызывающий процесс должен иметь соответствующие права доступа к набору семафоров.

Операции GETVAL, GETNCNT, GETZCNT, GETPID позволяют получить значение поля структуры семафора с номером semnum (semval, semncnt, semzcnt, sempid соответственно). Возвращаемым значением semctl() будет значение требуемого поля.

В остальных операциях вызов semctl() имеет четыре аргумента и принимает вид semctl(semid, semnum, cmd, arg), где четвертый аргумент имеет тип union semun:

```
union semun {
      int val;                  /* значение для SETVAL */
      struct semid_ds *buf;     /* буфер для IPC_STAT, IPC_SET */
      unsigned short *array;    /* массив для GETALL, SETALL */
      /* Linux-специфическая часть */
      struct seminfo *__buf;    /* Буфер для IPC_INFO */
}
```

Каждое из полей объединения используется определенными операциями (cmd). Объединение union semun не объявлено в стандартных заголовочных файлах; его следует объявить самостоятельно, выбрав из приведенных выше необходимые поля.

Операция SETVAL устанавливает значение поля semval семафора с номером semnum. Процессы, заблокированные на данном семафоре, разблокируются, если значение semval становится равным 0 или увеличивается.

Операция SETALL устанавливает значения поля semval всех семафоров набора по значениям элементов массива arg.array. Как и при SETVAL, возможна разблокировка процессов.

Операция GETALL выполняет чтение значений поля semval всех семафоров в массив arg.array.

Операции IPC_STAT, IPC_SET используются для чтения и изменения структуры описания набора семафоров (например, изменения прав доступа) и здесь не рассматриваются.

Для выполнения основных операций над семафорами (P- и V-операции) служит системный вызов semop(), объявленный в sys/sem.h следующим образом:

```
int     semop(int semid, struct sembuf *sops, unsigned nsops);
```

При успешном завершении semop() возвращает 0, при ошибке – значение -1.

Первый аргумент semid – это идентификатор набора семафоров, с которым производится операция, второй аргумент sops – указатель на массив структур, описывающих действия над семафорами набора; третий аргумент nsops – размер этого массива.

Структура struct sembuf описывает действие над семафором массива и имеет следующие поля:

```
struct sembuf {
    unsigned short sem_num;     /* Номер семафора в наборе */
    short sem_op;               /* Операция над семафором */
    short sem_flg;              /* Флаги операции */
}
```

Операция над семафором задается целым числом со знаком. Если это число больше 0, то оно добавляется к полю semval семафора (возможно, вызывая разблокировку процессов, ранее заблокированных на данном семафоре). Если значение sem_op меньше 0, то возможны два варианта:

1) если абсолютное значение sem_op меньше значения поля semval семафора, то semval уменьшается на абсолютное значение sem_op и вызвавший процесс продолжается;

2) если абсолютное значение sem_op больше значения поля semval семафора, то вызвавший процесс блокируется до тех пор, пока значение semval не станет больше или равно sem_op. Когда

это произойдет, значение semval будет уменьшено на абсолютное значение sem_op и вызвавший процесс продолжится.

Разблокировка процесса может произойти также в результате уничтожения набора семафоров. В этом случае системный вызов semop() возвращает значение ошибки, а код ошибки (errno) будет равен EIDRM.

Если заблокированный на семафоре получает сигнал, то он разблокируется и системный вызов semop() возвращает ошибку с кодом errno=EINTR.

Существуют два флага операции. Флаг IPC_NOWAIT запрещает блокировку процесса; вместо блокировки системный вызов semop() вернет значение "ошибка" и установит errno=EAGAIN. Флаг SEM_UNDO указывает, что операция над семафором должна быть отменена (undo), когда процесс завершится.

Практически для реализации P- и V-операций используются два значения sem_op: -1 для P-операции ("закрытия" семафора) и 1 для V-операции ("открытия" семафора).

Состояние используемых в системе наборов семафоров можно узнать командой ipcs.

Рассмотрим пример использования семафоров для доступа к разделяемому ресурсу. Как и в разд. 4, создадим две программы, одна из которых ("сервер") будет читать из общей разделяемой памяти строки и выводить их на экран, а вторая ("клиент") будет запрашивать у пользователя эти строки и записывать их в разделяемую память. Если в предыдущем примере программа-сервер после вывода строки на экран сразу же завершала свою работу, то здесь для завершения работы программы-сервера следует передать ему строку "EXIT". В качестве разделяемого ресурса, доступ к которому регулируется семафором, выступает общая разделяемая память. Признаком наличия строки в разделяемой памяти является ненулевой первый байт строки. Тексты программ приведены ниже:

```
/* Файл sem_defs.h */
/* Ключ разделяемой памяти */
#define SEG_KEY 0x12345678

/* Ключ набора семафоров */
#define SEM_KEY 0x12345678
```

```
/* Максимальная длина передаваемой строки */
#define MAXSTRLEN  100

/* union для вызова semctl() */
union semun {
        int val;
} ;


/* Файл sem_main.c - программа-"сервер" */
#include <stdio.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#include "sem_defs.h"

main()
{
        int shmid; /* ID сегмента разделяемой памяти */
        char *str; /* Указатель на принимаемую строку */
        int semid; /* ID набора семафоров */
        int retcode = 1; /* Код возврата из программы */
        union semun semcd;  /* Используется semctl() */
        struct sembuf semoper[1];  /* Используется semop() */
        int exitflag=1; /* Флаг выхода из цикла приема строк */

        /* Создание сегмента разделяемой памяти */
        if ( (shmid = shmget(SEG_KEY, MAXSTRLEN+1,
                        IPC_CREAT | IPC_EXCL | 0600)) < 0) {
            printf("Ошибка shmget()!\n"); return 1;
        }

        /* Подключение сегмента разделяемой памяти */
        if((str=(char *)shmat(shmid,NULL,0))==(char *)(-1)) {
            printf("shmat() error\n"); goto destroy_segment;
        }
        /* Создание набора семафоров */
        if ( (semid = semget(SEM_KEY, 1,
                                IPC_CREAT | IPC_EXCL | 0600)) < 0 ) {
            printf("Ошибка semget()!\n");
            goto destroy_segment;
        }
        /* Установка начального значения семафора, равного 1 */
        semcd.val = 1;
        if (semctl(semid, 0, SETVAL, semcd) < 0) {
            printf("Ошибка semctl()!\n");
```

```
        goto destroy_semaphore;
    }
    /* Цикл приема строк через разделяемую память */
    while(exitflag) {
        /* Обеспечиваем монопольный доступ к ресурсу
        ( разделяемому сегменту памяти ) */
        /* P-операция */
        semoper[0].sem_num = 0; /* Номер семафора в наборе */
        semoper[0].sem_op = -1; /* Операция */
        semoper[0].sem_flg = 0;
        semop(semid, semoper, 1); /* Выполнение операции */

        if (str[0]) { /* Если строка имеется ... */
            printf("Принята строка: %s\n", str);
            /* Проверка: передана команда выхода? */
            if (strncmp(str, "EXIT", 4) == 0)  {
                exitflag = 0;
                printf("ВЫХОД.\n");
            }

            /* Сбрасываем признак наличия строки */
            str[0] = 0;
        }

        /* Освобождаем ресурс */
        /* V-операция */
        semoper[0].sem_num = 0; /* Номер семафора в наборе */
        semoper[0].sem_op = 1;  /* Операция */
        semoper[0].sem_flg = 0;
        semop(semid, semoper, 1); /* Выполнение операции */
    } /* Конец цикла приема строк */

    /* Нормальное завершение */
    /* Отсоединение сегмента */
    shmdt(str);
    retcode = 0;

destroy_semaphore:
    /* Уничтожение набора семафоров */
    semctl(semid, IPC_RMID, 0);

destroy_segment:
    /* Уничтожение сегмента разделяемой памяти */
    shmctl(shmid, IPC_RMID, NULL);
    return retcode;
}
```

```
/* Файл sem_send.c – программа-клиент */
#include <stdio.h>
#include <unistd.h>
#include <sys/shm.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include "sem_defs.h"

main()
{
	int shmid; /* ID сегмента разделяемой памяти */
	char *str; /* Передаваемая строка */
	int semid; /* ID набора семафоров */
	struct sembuf semoper[1]; /* Используется semop() */

	/* Получение доступа к разделяемой памяти */
	if ( (shmid = shmget(SEG_KEY, MAXSTRLEN+1, 0600)) < 0) {
		printf("Ошибка shmget()!\n"); return 1;
	}
	/* Подключение сегмента разделяемой памяти */
	if((str=(char *)shmat(shmid,NULL,0))==(char *)(-1)) {
		printf("shmat() error\n"); return 1;
	}
	/* Получение доступа к набору семафоров */
	if ( (semid = semget(SEM_KEY, 1, 0600)) < 0 ) {
		printf("Ошибка semget()!\n");
		return 2;
	}

	/* Получение монопольного доступа к разделяемой памяти */
	/* P-операция */
	semoper[0].sem_num = 0; /* Номер семафора в наборе */
	semoper[0].sem_op = -1; /* операция */
	semoper[0].sem_flg = 0;
	semop(semid, semoper, 1); /* Выполнение операции */
	/* Ввод строки с клавиатуры */
	printf("Введите строку: ");
	fgets(str, MAXSTRLEN, stdin);
	/* Освобождаем ресурс (разделяемую память) */
	/* V-операция */
	semoper[0].sem_num = 0; /* Номер семафора в наборе */
	semoper[0].sem_op = 1; /* операция */
	semoper[0].sem_flg = 0;
	semop(semid, semoper, 1); /* Выполнить операцию */
	/* Отключение сегмента разделяемой памяти */
	shmdt(str);
	return 0;
}
```

Скомпилируем обе программы командами cc -o main sem_main.c и cc -o send sem_send.c. Запустим программу-сервер в фоновом режиме командой main &. Если при запуске программы возникают ошибки, то следует попытаться удалить сегмент разделяемой памяти и набор семафоров командой ipcrm –M 0x12345678 –S 0x12345678 (здесь 0x12345678 – ключ разделяемой памяти (после –M) и набора семафоров (после –S)). Если это не помогает, нужно изменить значение ключей сегмента разделяемой памяти и набора семафоров (SEG_KEY и SEM_KEY) в файле sem_defs.h и перекомпилировать программы sem_main.c и sem_send.c.

После запуска программы main с помощью команды ipcs можно убедиться в наличии сегмента разделяемой памяти и набора семафоров с ключами 0x12345678:

```
bash$ ipcs –ms
------ Shared Memory Segments --------
key             shmid   owner   perms   bytes   nattch   status
0x12345678      32769   user    600     101     1
. . . . . .
------ Semaphore Arrays --------
key             semid   owner   perms   nsems
0x12345678      32768   user    600     1
```

Теперь запустим программу-клиент (send) и в ответ на её приглашение введем строку (например, TEST):

```
bash$ send
Введите строку: TEST
Принята строка: TEST
```

Нажмите клавишу <Enter>, чтобы получить приглашение bash. Еще раз запустим программу send:

```
bash$ send
Введите строку:
```

В этот раз не будем вводить строку, а вызовем еще одно окно терминала (или перейдем на другую консоль) и там попытаемся вызвать еще одну копию той же программы send:

```
bash$ send
```

Видим, что программа не выдала приглашения на ввод строки. Это произошло потому, что ввод строки происходит после прохождения семафора, который оказался заблокирован первой запущенной программой send. Таким образом, действительно обеспечен *монопольный доступ* к разделяемому ресурсу. Вернемся в окно терминала (или на консоль), где была запущена первая программа send, и в ответ на приглашение введем строку ABCDEF. После нажатия клавиши <Enter> увидим на экране

```
Принята строка: TEST
```

Перейдем на второе окно терминала (или на консоль), где была запущена вторая программа send. Увидим, что эта программа теперь вывела приглашение на ввод строки. Это произошло потому, что первая программа send освободила доступ к разделяемому ресурсу и дала возможность второй программе пройти семафор (строго говоря, программа main также периодически закрывает и открывает семафор, но, поскольку это происходит очень быстро, пользователь, запускающий send, этого не замечает). Введем в ответ на приглашение строку EXIT. Нажав клавишу <Enter> и перейдя к первому окну терминала (или первой консоли), увидим на экране следующие сообщения от программы-сервера (main):

```
Принята строка: EXIT
ВЫХОД.
[1]+ Done                    main
```

Таким образом, программа-сервер завершила свою работу. С помощью команды ipcs –ms можно убедиться, что и сегмент разделяемой памяти, и набор семафоров, использованные в этих программах, удалены.

В рассмотренных программах для получения доступа к заданному набору семафоров использован статический ключ набора семафоров. В качестве альтернативы можно использовать динамически выделяемый ключ с передачей дескриптора набора семафоров из программы, создавшей набор (в нашем случае main), всем остальным использующим его набор программам. Использование динамического ключа набора семафоров полностью аналогично использованию динамического ключа при работе с разделяемой памятью, рассмотренной в разд. 4.

Приведенные программы – учебные, демонстрирующие принцип использования семафоров для доступа к разделяемому ресурсу. Они не являются оптимальными, например не вынесены из циклов инвариантные присваивания (для большей наглядности реализации операций с семафором). Вторая возможность оптимизации – использование дополнительного семафора для блокировки программы-сервера до получения строки вместо использованного цикла постоянной проверки.

Контрольные вопросы

1. Что такое семафор? Каковы основные операции над семафором?
2. Как создать или подключить существующий набор семафоров?
3. Как задать начальные значения семафоров?
4. Как реализовать P-операцию над семафором средствами Unix?
5. Как реализовать V-операцию над семафором средствами Unix?
6. Как уничтожить набор семафоров? Что произойдет при этом с процессами, заблокированными на семафорах данного набора?
7. Как из командной строки получить информацию о наборах семафоров?
8. Как из командной строки уничтожить заданный набор семафоров?

6. СОКЕТЫ

6.1. Общие сведения

Сокеты – это универсальный способ взаимодействия процессов. Их можно использовать для передачи данных как локально (в пределах одной системы), так и между удаленными системами через сеть. Сокеты можно классифицировать следующим образом:

1. По используемому протоколу. Существует и поддерживается в Unix/Linux множество протоколов. Здесь рассматриваются два: **Unix-сокеты**, обеспечивающие локальное взаимодействие процессов в пределах одной системы и **IP-сокеты**, обеспечивающие по IP-протоколу как локальное, так и удаленное взаимодействие.

2. По способу взаимодействия можно также выделить несколько типов, из которых рассмотрим *потоковые сокеты* –

обеспечивающие упорядоченную, надежную (гарантированную), двустороннюю передачу данных в виде потока байтов, основанную на соединениях. Обычно для передачи используется протокол TCP.

Отметим, что существуют дейтаграммные (датаграммные) сокеты, обеспечивающие быструю, но не гарантированную передачу данных в виде пакетов фиксированного максимального размера; дейтаграммные сокеты не требуют создания соединений.

Общая последовательность взаимодействия через потоковые сокеты процессов – сервера и клиента – следующая (в скобках указаны используемые системные вызовы):

1. Сервер создает сокет (socket()) и присваивает ему адрес или имя (bind()).
2. Сервер включает *прослушивание* сокета (listen()).
3. При готовности к соединению сервер выполняет прием запроса на подключение (accept()). Если запросов нет, сервер обычно блокируется на вызове accept() (возможны не рассматриваемые здесь варианты).
4. Клиент создает свой сокет (socket()) и, зная адрес или имя сокета сервера, выполняет запрос на подключение к нему (connect()).
5. Получив запрос на подключение, система сервера создает новый сокет, разблокирует accept() в серверном процессе и передает дескриптор вновь созданного сокета как результат accept(). Исходный сокет остается в режиме прослушивания.
6. Имея дескрипторы сокетов, сервер и клиент могут обмениваться данными, используя обычные вызовы read() и write().

Отметим, что поступающие на заданный сокет сервера запросы на подключение образуют очередь. Системный вызов accept() извлекает из очереди один из запросов; чтобы принять остальные, следует повторно вызвать accept().

Рассмотрим используемые системные вызовы более подробно.

Создание сокетов осуществляется системным вызовом socket(), объявленным в sys/socket.h:

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

При успешном завершении socket() возвращает дескриптор созданного сокета, при ошибке – значение -1.

Первый аргумент domain задает семейство протоколов, используемых для связи через сокет. Для локальных Unix-сокетов задается как PF_UNIX или PF_LOCAL (синонимы), для IP-сокетов – PF_INET для протокола IPv4 или PF_INET6 для протокола IPv6.

Второй аргумент type определяет тип сокета. Для потоковых сокетов задается значение SOCK_STREAM.

Третий аргумент protocol указывает конкретный протокол передачи данных. Здесь можно указать значение 0, тогда выбор протокола произойдет автоматически.

Отметим, что существует вызов socketpair(), создающий пару неименованных сокетов для локального взаимодействия.

После создания сокета ему нужно присвоить имя или адрес, идентифицирующие данный сокет для других процессов. Для этого используется системный вызов bind(), объявленный в sys/socket.h:

```
int bind(int sdesc, const struct sockaddr *address, socklen_t len);
```

Вызов bind() возвращает 0 при успешном завершении и -1 при ошибке.

Аргумент sdesc – это дескриптор сокета, полученный ранее вызовом socket(); структура struct sockaddr *address определяет адрес или имя сокета, а socklen_t – размер этой структуры.

В качестве второго аргумента могут передаваться указатели на структуры следующих типов:

1. Для локальных сокетов используется структура struct sockaddr_un, объявленная в sys/un.h. В ней в поле sun_family записывается константа AF_UNIX, а в поле sun_path (char sun_path[...]) – путь+имя сокета, например, с помощью strcpy().

2. Для IP-сокетов используется структура struct sockaddr_in, описанная в netinet/in.h, в которой имеются поля:

sin_family – определяет семейство адресов, используется значение AF_INET;

sin_addr и sin_port – определяют IP-адрес и порт. В них порядок записи байтов может отличаться от используемого в машине, поэтому для формирования значений этих полей следует использовать библиотечные функции (htons() для номера порта,

inet_makeaddr() и т.п. для IP-адреса). Подробнее их использование рассматривается далее, в подразд 6.3.

Последний аргумент – размер (длину) структуры sockaddr, можно задать, используя sizeof().

Системный вызов listen() включает прослушивание сокета. Он объявлен в sys/socket.h:

```
int   listen(int sdesc, int maxqueuelen);
```

Он возвращает значение 0 при успехе и -1 при ошибке.

Первый аргумент sdesc – это дескриптор сокета (полученный от socket()). Второй аргумент maxqueuelen задает максимальное количество запросов на подключение в очереди к данному сокету.

Системный вызов connect() отправляет запрос подключения к заданному сокету. Он также объявлен в sys/socket.h:

```
Int  connect( int sdesc, const struct sockaddr *serv_addr,
                        socklen_t len);
```

Первый аргумент sdesc – дескриптор сокета, полученный от socket(). Второй и третий аргументы аналогичны аргументам вызова bind(), с той разницей, что они указывают существующий адрес для подключения (а bind() задает новый адрес сокета).

Системный вызов accept() служит для приема сервером запроса на подключение к сокету. Он объявлен в sys/socket.h:

```
int     accept( int sdesc, struct sockaddr *addr, socklen_t *len);
```

При успешном выполнении он извлекает первый запрос из очереди запросов на подключение к сокету, создает новый сокет с большинством свойств исходного сокета и возвращает дескриптор созданного сокета. Также accept() заполняет структуру struct sockaddr *addr данными, аналогичными указываемым в качестве входных в bind() и connect(). В *len возвращается размер этой структуры. Если эти данные не нужны (обычно), то в качестве второго и третьего аргументов указывается NULL.

Создаваемые и именованные bind() локальные сокеты видны в файловой системе как файлы типа "socket".

6.2. Локальные сокеты

Рассмотрим две программы: сервер и клиент, использующие локальный сокет. Программа-сервер создает сокет и далее выводит поступающие через него сообщения. Программа-клиент выводит в сокет сообщения, указываемые в командной строке. Тексты программ приведены ниже:

```
/* Файл sdefs.h */

/* Имя локального сокета */
#define SOCKETNAME "mysocket"

/* Сообщение, уничтожающее процесс-сервер */
#define EXIT_MESSAGE "EXIT"

/* Программа-сервер - файл sserver.c */
#include <stdio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <string.h>

#include "sdefs.h"

/* Размер буфера для сообщения */
#define BUFLEN 100
/* Длина очереди запросов к сокету */
#define QUEUE_LEN 5

main()
{
    int sdesc; /* Дескриптор основного (слушающего) сокета */
    int asock; /* Дескриптор сокета соединения */
    struct sockaddr_un saddr; /* Структура - адрес сокета */

    int len, stat = 1; /* Длина сообщения и код завершения */
    char buf[BUFLEN]; /* Буфер сообщения */

    /* Создание сокета */
    if ( (sdesc = socket(PF_UNIX, SOCK_STREAM, 0)) < 0 ) {
        printf("Ошибка socket()!\n"); return 1;
    }

    /* Присвоение сокету имени */
    saddr.sun_family = AF_UNIX;
```

```
    strcpy(saddr.sun_path, SOCKETNAME);
    if ( bind(sdesc, (struct sockaddr *)&saddr,
                        sizeof(saddr) ) < 0 ) {
        printf("Ошибка bind()! (Сокет уже существует?)\n");
        return 1;
    }

    /* Включение прослушивания сокета */
    if ( listen(sdesc, QUEUE_LEN) < 0 ) {
        printf("Ошибка listen() !\n");
        unlink(SOCKETNAME); return 1;
    }

    /* Цикл приема сообщений через сокет */
    for(;;) {
        /* Прием соединения; создается новый сокет для него */
        if ( (asock = accept(sdesc, NULL, NULL) ) < 0 ) {
            printf("Ошибка accept()!\n"); break;
        }
        /* Чтение сообщения из нового сокета */
        if ( (len = read(asock, buf, BUFLEN-1) ) < 0 ) {
            printf("Ошибка чтения из сокета!\n"); break;
        }
        /* Закрытие нового сокета. */
        close(asock);

        buf[len] = 0; /* Добавление символа '\0' для гарантии
                        правильного завершения строки */
        printf("Принято: <%s>\n", buf);

        /* Если принято сообщение "EXIT", завершить работу */
        if(!strcmp(buf, EXIT_MESSAGE)) {
            stat = 0; break;
        }
    }

    /* Закрытие сокета */
    close(sdesc);
    /* Удаление файла сокета */
    unlink(SOCKETNAME);
    /* Завершение программы */
    return stat;
}


/* Программа-клиент. Файл sclient.c */
#include <stdio.h>
#include <unistd.h>
```

```
#include <sys/socket.h>
#include <sys/un.h>
#include <string.h>

#include "sdefs.h"

main(int argc, char *argv[])
{
      int sdesc;
      struct sockaddr_un saddr;

      /* Проверка аргументов и вывод справки */
      if (argc<2) {
          printf("Использование: %s  Сообщение\n"\
              "Сообщение для закрытия сервера: %s\n",
              argv[0], EXIT_MESSAGE);
          return 1;
      }

      /* Создание сокета */
      if( ( sdesc = socket(PF_UNIX, SOCK_STREAM, 0) ) < 0 ) {
          printf("Ошибка socket()!\n"); return 1;
      }
      /* Соединение с сокетом сервера */
      saddr.sun_family = PF_UNIX;
      strcpy(saddr.sun_path, SOCKETNAME);
      if( connect(sdesc, (struct sockaddr *) &saddr,
                                  sizeof(saddr) ) < 0 ) {
          printf("Ошибка соединения! (сервер не запущен?)\n");
          return 1;
      }
      if ( write( sdesc, argv[1], strlen(argv[1])+1 ) < 0 ) {
          printf("Ошибка записи в сокет!\n");
          return 1;
      }

      close(sdesc); /* Закрытие сокета */
      return 0;
}
```

Скомпилируем обе программы командами cc -o sserver sserver.c и cc -o sclient sclient.c. Запустим программу-сервер в фоновом режиме командой sserver &. Проверим теперь существование файла-сокета:

```
bash$ ls -l mysocket
srwxr-xr-x      1 user users     0    2010-11-21 20:48 mysocket
```

Первая буква в строке ('s') означает тип файла – сокет.

Запустим теперь программу-клиент, указав в качестве аргументов некоторые сообщения. Пронаблюдаем вывод "Принято <.....>" от программы-сервера:

```
bash$ sclient 'Connected!"
Принято: <Connected!>
bash$ sclient 'Unix socket works!'
Принято: <Unix socket works!>
bash$ sclient EXIT
Принято: <EXIT>
[1]+      Done                sserver
```

6.3. IP-сокеты

В отличие от локальных сокетов, IP-сокеты можно использовать для организации удаленного взаимодействия процессов через сеть. Адрес сокета состоит из двух частей: сетевого адреса (это IP-адрес) и порта – номера порта, используемого сокетом.

IP-адрес представляет собой целое число, идентифицирующее данный компьютер в сети. Рассмотрим здесь IP-адреса версии 4 (IP v4) как более компактные (IP v6 отличается только большей длиной адреса). IP-адрес версии v4 имеет длину 4 байта. Обычно его записывают в "точечной" форме, записывая десятичные значения каждого байта и разделяя их точкой; например "192.168.1.2". Кроме того, существует система доменных имен (DNS), позволяющая записывать адреса в более наглядном виде, например "mail.ru". Для указания вызову bind() IP-адрес можно сформировать с помощью следующих библиотечных функций: inet_aton() при использовании "точечной" записи адреса, gethostbyname() при использовании DNS-записи и ряда других. Не следует указывать IP-адрес непосредственно числовой константой или целой переменной, поскольку порядок записи байтов в конкретной машине может отличаться от принятого в сети. Например, в процессорах ряда Intel первым идет младший байт, а в сети принят порядок процессоров Motorola: первым идет старший байт. Использование библиотечных функций позволяет сделать программу более мобильной.

Функция inet_aton() объявлена в netinet/in.h следующим образом:

```
int     inet_aton(const char *cp, struct in_addr *inp);
```

Она преобразует IP-адрес из точечной формы записи в виде строки const char *cp в числовую, принятую в сети форму. Полученный адрес помещается в структуру struct in_addr *inp, содержащую единственное поле – unsigned long int s_addr. Полученную структуру целиком следует скопировать в поле sin_addr структуры sockaddr_in, используемой вызовом bind().

Использование портов позволяет обмениваться данными через один IP-адрес нескольким различным клиентам. Номера портов меньше 1024 называются *зарезервированными*. Многие из них используются стандартными сетевыми службами и протоколами, например, протокол FTP использует порты 20 и 21, SSH – порт 22, telnet – порт 23, HTTP – порт 80. Порт обычно указывают после адреса через двоеточие, например, "127.0.0.1:80" означает "порт 80 по IP-адресу 127.0.0.1".

По тем же причинам, что и для IP-адресов, для преобразования номера порта в значение, используемое в сети, следует использовать библиотечную функцию htons(), объявленную в netinet/in.h:

```
uint16_t        htons(uint16_t hostshort);
```

Она преобразует число-аргумент hostshort в форму с порядком байтов, принятым в сети.

Рассмотрим пример работы с IP-сокетами. Принцип организации соединения и передачи данных остается тем же, что и для локальных сокетов, но вместо имени локального сокета в вызове bind() нужно указать IP-адрес и порт. Кроме того, когда при создании соединения (accept()) создается новый сокет, ему назначается тот же IP-адрес, но новое уникальное значение порта. Это позволяет обслуживать несколько запросов к одному сокету.

Программа представляет собой пример простейшего HTTP-сервера. При обмене по HTTP-протоколу браузер вначале обычно передает запрос "GET /", т.е. "получить главную страницу сайта". Распознав этот запрос, наш сервер передает через тот же сокет текст простой web-страницы. В качестве адреса сокета используется 127.0.0.1 – стандартный адрес loop-интерфейса, т.е. адрес самого компьютера, поэтому программа работоспособна даже при отсутствии локальной сети. Текст программы с комментариями:

```
/* Файл ipserver.c */
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>

/* IP-адрес и порт, используемые программой */
#define MY_IPADDR       "127.0.0.1"
#define MY_PORT         2080

/* Размер буфера для приема запроса */
#define REQLEN 256
/* Размер очереди запросов */
#define QUEUE_LEN 5

/* HTML-текст простой web-странички */
#define HTTP_TEXT " <HTML> Test </HTML> \n"

main()
{
    int  sdesc; /* Дескриптор основного сокета */
    int  asock; /* Дескрипторы соединения */
    struct  sockaddr_in  saddr;  /* Адрес и порт сокета */
    struct  in_addr   ipaddr;  /* Для получения IP-адреса */

    int len;
    char req[REQLEN];  /* Буфер приема запросов */

    /* Создаем сокет */
    if ( (sdesc = socket(PF_INET, SOCK_STREAM, 0)) < 0 ) {
        printf("Ошибка socket()!\n"); return 1;
    }

    /* Преобразуем IP-адрес из точечной в числовую форму */
    if ( inet_aton(MY_IPADDR, &ipaddr) < 0 ) {
        printf("Ошибка inet_aton() (ош. адрес?)\n"); return 1;
    }

    /* Заполняем поля структуры адреса сокета */
    saddr.sin_family = AF_INET;  /* Семейство адресов */
    saddr.sin_addr = ipaddr;  /* IP-адрес */
    saddr.sin_port = htons(MY_PORT);  /* Номер порта */

    /* Устанавливаем адрес сокета */
    if ( bind(sdesc, (struct sockaddr *)&saddr,
                                    sizeof(saddr) ) < 0 ) {
        printf("Ошибка bind()!\n"); return 1;
    }
```

```
    /* Включаем прослушивание сокета */
    if ( listen(sdesc, QUEUE_LEN) < 0 ) {
        printf("Ошибка listen()!\n"); return 1;
    }

    /* Бесконечный цикл приема запросов.
        Выход из цикла - по нажатию Ctrl+C или при ошибках. */
    for(;;) {
        /* Принимаем запрос соединения */
        if ( (asock = accept(sdesc, NULL, NULL) ) < 0 ) {
            printf("Ошибка accept()!\n"); break;
        }
        if ( (len = read(asock, req, REQLEN-1) ) < 0 ) {
            printf("Ошибка read()!\n"); break;
        }
        req[len] = 0; /* Гарантия завершения строки */
        printf("Принят запрос: <%s>\n", req);
        /* Это запрос браузера ? */
        if( strncmp(req, "GET /", 5) == 0 ) {
            printf("Отсылаю HTML-текст\n");
            write(asock, HTTP_TEXT, strlen(HTTP_TEXT)+1);
        }
        /* Закрываем новый сокет */
        close(asock);
    } /* Конец цикла приема запросов */
}
```

Скомпилируем программу командой cc -o ipserver ipserver.c и запустим ее командой ipserver. Если возникнут ошибки bind(), то следует заменить номер порта (2080) в программе на некоторый другой.

Теперь в другой консоли или другом окне запустим браузер и укажем в строке адреса http://127.0.0.1:2080. Браузер отобразит страницу с одним словом "Test". В консоли (окне), где был запущен наш сервер, можно наблюдать текст запроса, пришедшего от сервера, например:

```
. . . . . .
Принят запрос: <GET / HTTP/1.0
Host: 127.0.0.1:2080
Accept:text/html, text/plain, text/sgml, video/mpeg, image/j>
Отсылаю HTML-текст
. . . . . .
```

Обратите внимание, что после "GET /" могут идти дополнительные параметры, зависящие от используемого браузера. Поэтому в программе выполняется проверка принятого запроса с помощью функции strncmp(), а не strcmp().

Для завершения работы нашего мини-сервера нажмите Ctrl+C в консоли (окне терминала), где он был запущен.

Контрольные вопросы

1. Что такое сокет? Какие виды сокетов вы знаете?
2. Какова последовательность работы с потоковыми сокетами на стороне сервера и на стороне клиента?
3. Что делают системные вызовы socket(), bind(), connect(), listen() и accept()?
4. Виден ли локальный сокет в каталоге файлов?
5. Что такое IP-адрес и номер порта? Для чего нужен номер порта?
6. Как в Си-программе указать IP-адрес и номер порта? Почему для этого следует использовать библиотечные функции?
7. Почему возможно параллельное обслуживание через IP-сокет нескольких запросов к одному и тому же адресу/порту?

7. СИГНАЛЫ. ИЕРАРХИЯ ПРОЦЕССОВ

7.1. Понятие сигнала

Сигнал в ОС Unix/Linux – это некоторое сообщение, отправляемое одним процессом другому процессу. Процесс, получивший сигнал, может отреагировать на него одним из следующих действий:

1. *Завершиться*. Процесс, получивший сигнал, завершается.
2. *Завершение и дамп памяти*. Процесс, получивший сигнал, завершается, и содержимое памяти записывается на диск для последующего анализа.
3. *Игнорировать*. Процесс не выполняет никаких действий. Не всякий сигнал можно игнорировать.
4. *Перехватить и обработать*. При получении сигнала работа процесса асинхронно прерывается и вызывается процедура-обра-

ботчик сигнала, после завершения работы которой исполнение процесса может быть продолжено. Адрес вызова процедуры (функции) обработчика предварительно сообщается системе процессом с помощью системного вызова sigaction(). Отметим, что на время обработки сигнала процедурой-обработчиком повторное поступление того же сигнала блокируется; блокировка снимается после завершения работы процедуры-обработчика (выхода из нее).

Вариант реакции предварительно выбирается процессом с помощью системного вызова sigaction(). Большинство сигналов можно заблокировать. Блокировка отличается от игнорирования тем, что поступивший, но заблокированный сигнал запоминается и остается в "подвешенном" (pending) состоянии и может быть принят (разблокирован) в дальнейшем, в то время как при игнорировании поступивший сигнал просто отбрасывается.

Кроме указанных четырех вариантов реакции, сигналом процесс может быть *остановлен* (без уничтожения), и затем другой сигнал может *продолжить* его выполнение. Данные действия не выбираются процессом, а предопределены в системе для нескольких сигналов; процесс может заблокировать (запретить) некоторые из них.

Сигналы идентифицируются по их номеру. В программах и при описаниях сигналов обычно пользуются не числовыми значениями номеров сигналов, а символическими константами – именами сигналов. Например, сигнал с номером 9 имеет имя SIGKILL.

Кроме передачи из некоторого процесса, некоторые ситуации в системе приводят к передаче процессам сигналов. Например, нажатие клавиш <Ctrl+C> приводит к отправке сигнала SIGINT, обращение по недействительному или запрещенному для процесса адресу вызывает отправку процессу сигнала SIGSEGV. Фактически в подобных ситуациях сигнал отправляет процессу ядро системы.

Полный список сигналов и их значений можно просмотреть командой man 7 signal. Описание некоторых сигналов приведено в следующей таблице.

Некоторые сигналы Unix

Имя сигнала	Номер сигнала	Действие по умолчанию	Причина сигнала
SIGHUP	1	Завершение	Отключение (hang-up) управляющего терминала или завершение процесса, управляющего терминалом. Некоторые процессы-демоны при получении данного сигнала выполняют повторное чтение своих файлов конфигурации
SIGINT	2	Завершение	Прерывание от клавиатуры (Ctrl+C)
SIGQUIT	3	Дамп	Завершение (quit) от клавиатуры (Ctrl+\)
SIGILL	4	Дамп	Недопустимая команда процессора
SIGABRT	6	Дамп	Аварийное завершение (процесс вызвал функцию abort())
SIGFPE	8	Дамп	Ошибка операции с плавающей точкой
SIGKILL	9	Завершение	Отправление сигнала KILL
SIGSEGV	11	Дамп	Обращение по недействительному адресу памяти
SIGPIPE	13	Завершение	"Broken pipe" – запись в канал, откуда никто не читает
SIGALRM	14	Завершение	Сигнал таймера, установленного вызовом alarm()
SIGTERM	15	Завершение	Сигнал завершения
SIGUSR1	10*	Завершение	1-й определяемый пользователем сигнал
SIGUSR2	12*	Завершение	2-й определяемый пользователем сигнал
SIGCHLD	17*	Игнорировать	Дочерний процесс остановлен или завершен
SIGCONT	18*	Продолжить	Продолжить остановленный процесс
SIGSTOP	19*	Остановка	Остановить процесс
SIGTSTP	20*	Остановка	Остановка процесса с терминала (Ctrl+Z)
SIGTTIN	21*	Остановка	Ввод с терминала для фонового процесса
SIGTTOUT	22*	Остановка	Вывод на терминал для фонового процесса

Сигналы, значения которых помечены "*", имеют разные значения в разных реализациях Unix; в таблице приведено их значение в Linux для платформы i386.

Сигналы SIGKILL и SIGSTOP не могут быть перехвачены, заблокированы или игнорированы.

Для отправки сигнала служит системный вызов kill(), объявленный в signal.h:

```
int  kill (pid_t pid, int sig);
```

Вызов kill() посылает сигнал с номером sig процессу или группе процессов. При успешном завершении kill() возвращает значение 0, при ошибке – значение -1. Процесс или процессы, которым направлен сигнал, определяются аргументом pid следующим образом:

1) если pid>0, то сигнал посылается процессу с номером pid;

2) если pid=0, то сигнал посылается каждому процессу в группе текущего процесса;

3) если pid= -1, то сигнал посылается каждому процессу в системе, кроме процесса **init**;

4) если pid< -1, то сигнал посылается каждому процессу группы с номером (-pid).

Чтобы процесс имел право отсылать сигнал другим процессам, должно быть выполнено одно из следующих условий:

1. Процесс имеет права суперпользователя (root), т.е. его UID или EUID равны 0.

2. Реальный (UID) или эффективный (EUID) номера пользователей отсылающего процесса должен быть равен реальному (UID) или установленному SETUID-битом номеру каждого из процессов-получателей сигнала.

3. Для отсылки сигнала SIGCONT достаточно, чтобы отсылающий и принимающий сигнал процессы принадлежали к одной сессии. Понятие сессии рассматривается в подразд. 7.3.

Проверить возможность отсылки сигнала можно, указав аргумент sig = 0. Отсылки сигнала не произойдет, а по возвращенному kill() значению можно судить о достаточности прав у текущего процесса.

7.2. Обработка и блокировка сигналов

Для определения варианта реакции процесса на полученный сигнал служит системный вызов sigaction(), объявленный в signal.h следующим образом:

```
Int  sigaction (int sig_num, const struct sigaction *act, struct sigaction *oldact);
```

Входные параметры: sig_num – номер сигнала, *act – структура типа struct sigaction, определяющая действие – реакцию на сигнал (рассмотрена ниже). В *oldact возвращается предыдущее

значение структуры, определяющей реакцию на сигнал; если это значение не нужно, то следует указать значение oldact = NULL. При успешном завершении sigaction() возвращает значение 0, при ошибке - значение -1.

Структура sigaction, определяющая политику реагирования на сигнал, описана так:

```
struct sigaction {
        void (*sa_handler)(int );
        void (*sa_action)(int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restorer)(void);
}
```

Поля sa_handler и sa_action задают действие, выполняемое при получении сигнала. В поле sa_handler можно записать следующее:

- SIG_DFL – реакция по умолчанию;
- SIG_IGN – игнорировать сигнал;
- указатель на *функцию-обработчик* сигнала. Функция-обработчик – это обычная функция, принимающая единственный аргумент типа int – номер поступившего сигнала и возвращающая void.

Поле sa_action служит той же цели, но функция-обработчик принимает другие аргументы, здесь не рассматриваемые. Не следует одновременно задавать значения и поля sa_handler, и поля sa_action.

Поле sa_mask задает маску сигналов, используемую во время исполнения обработчика сигнала.

В поле sa_flags записывается значение, формируемое с помощью побитового ИЛИ ('|') из нескольких констант, изменяющих некоторые особенности реакции на сигнал (например, использовать во время обработки сигнала другой стек). Обычно достаточно записать в это поле значение 0.

Поле sa_restorer является устаревшим и не используется.

Маска сигналов – это битовое поле типа sigset_t, определяющее, какие из сигналов будут ***блокированы***. Для работы с битовым полем этого типа предназначены следующие библиотечные функции, объявленные в signal.h:

int sigemptyset(sigset_t *set) – очищает маску сигналов;

int sigfillset(sigset_t *set) – устанавливает маску сигналов для всех сигналов;

int sigaddset(sigset_t *set, int sig_num) – добавляет к маске сигнал с номером sig_num;

int sigdelset(sigset_t *set, int sig_num) – удаляет из маски сигнал с номером sig_num;

int sigismember(const sigset_t *set, int sig_num) – проверяет наличие в маске сигнала с номером sig_num; возвращает значения: 0 – сигнала нет в маске, 1 – сигнал указан в маске, -1 – ошибка.

Во всех функциях аргумент set – это битовое поле, с которым производится операция. Функции возвращают значение 0 при успешном завершении, -1 при ошибке.

Для блокирования/разблокирования сигналов служит системный вызов sigprocmask(), объявленный в signal.h следующим образом:

```
Int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Аргумент how определяет, как следует использовать указанную в *set маску сигналов. Можно указать следующие варианты:

SIG_BLOCK – добавить к блокируемым сигналы, указанные в маске;

SIG_UNBLOCK – снять блокировку сигналов, указанных в маске;

SIG_SETMASK – установить набор блокируемых сигналов, равный маске;

В *oldset заносится предыдущее значение маски блокировки сигналов; если оно не нужно, следует указать значение oldset=NULL.

Рассмотрим пример программы, использующей обработку сигнала. Установим собственный обработчик сигнала SIGINT (нажатие клавиш Ctrl+C) и заблокируем сигнал SIGTERM (завершение процесса). Текст программы.

```
/* Файл sig.c */
#include <signal.h>
#include <stdio.h>
```

```
/* Функция-обработчик сигнала SIGINT */
void ctrl_c_handler(int sig) {
        printf("Нажато Ctrl+C !\n");
}

main()
{
        struct sigaction act;
        sigset_t mask;

        /* Очистим маску сигналов */
        sigemptyset(&mask);
        /* Добавим к маске сигнал SIGTERM */
        sigaddset(&mask, SIGTERM);
        /* Установим блокировку по маске */
        sigprocmask(SIG_SETMASK, &mask, NULL);
        sigaddset(&mask, SIGINT);
        /* Заполним поля структуры act */
        act.sa_mask = mask;
        act.sa_handler = &ctrl_c_handler;
        act.sa_flags = 0;
        /* Установим обработчик сигнала */
        if ( sigaction(SIGINT, &act, NULL) == -1 ) {
             printf("Ошибка sigaction()!\n");
             return 1;
        }

        /* Бесконечный цикл */
        for (;;) ;
}
```

Скомпилируем программу командой cc -o sig sig.c и запустим ее командой sig. Нажимая клавиши Ctrl+C (например, три раза), увидим на экране сообщения вида

```
Нажато Ctrl+C !
Нажато Ctrl+C !
Нажато Ctrl+C !
```

Вызовем еще одно окно терминала (или перейдем на другую консоль) и попытаемся уничтожить запущенный процесс sig командой killall sig. Легко убедиться (например, командой ps u | grep sig), что процесс sig не уничтожен. Это произошло потому, что сигнал SIGTERM, отправляемый процессу командой kill, в процессе заблокирован. Чтобы его уничтожить, пошлем неблокируемый сигнал SIGKILL командой killall -KILL sig. В окне терминала или консоли, где был запущен процесс sig, увидим сообщение Killed, подтверждающее уничтожение процесса.

7.3. Иерархия процессов

Как уже говорилось ранее, в ОС Unix процессы организованы в группы. Группы процессов используются для управления доступом к терминалам и для обеспечения возможности посылки сигналов нескольким "родственным" процессам. Принадлежность процесса к группе определяется номером его группы процессов – PGID (Process Group ID). Дочерний процесс наследует номер группы родительского процесса. Ядро ОС Unix предоставляет простые механизмы изменения процессов своей группы или группы своих процессов-потомков. Создание новой группы процессов осуществляется просто, с помощью системного вызова setpgid() или setpgrp(). Значением номера созданной группы становится PID процесса-создателя группы, а процесс-создатель становится *лидером группы процессов*.

Изменение процессом номера своей группы процессов осуществляется также с помощью системных вызовов setpgrp() и setpgid() – процесс переходит в другую группу. Системные вызовы setpgrp() и setpgid() описаны в unistd.h следующим образом:

```
int  setpgid(pid_t pid, pid_t pgid);
int  setpgrp(void);
```

Вызов setpgid() устанавливает для процесса, указанного номером pid, номер группы pgid. Если указано значение pid=0, то используется номер вызвавшего процесса. Если указано значение pgid=0, то в качестве номера группы используется PID вызвавшего процесса. Таким образом, можно создать новую группу процессов, в которой вызвавший процесс станет лидером. Вызов setpgrp() равносилен вызову setpgid(0,0). Оба вызова возвращают значение 0 при успешном завершении и значение -1 при ошибке. Ошибку вызывают следующие запрещенные действия:

1) попытка изменить номер группы дочернего процесса, когда дочерний процесс уже исполнил вызов execve() или входит в другую сессию, чем родительский процесс;

2) попытка изменить номер группы процесса, являющегося лидером сессии;

3) попытка изменить номер группы процесса, не являющегося ни вызвавшим, ни дочерним процессом.

Группу процессов часто рассматривают как *задание (job)*, которым манипулируют программы-оболочки (sh, bash и т.п.). Распространенный тип задания, создаваемого оболочкой, – это *конвейер* из нескольких процессов, соединенных каналами (pipe), так что выход первого процесса становится входом второго, выход второго – входом третьего и т.д. Оболочка создает такое задание, применяя fork() для каждого процесса в конвейере, помещая затем все эти процессы в одну отдельную группу.

Процесс может послать сигнал как одному процессу, так и целой группе процессов. Процесс, принадлежащий к некоторой группе, может принимать сигналы, вызывающие приостановку, возобновление исполнения, прерывание или уничтожение всей группы.

Каждому терминалу присвоен номер группы процессов (PGID). Нормально он равен номеру группы процессов, связанной с терминалом. Программа-оболочка может создавать несколько групп процессов, связанных с одним терминалом; этот терминал будет *управляющим терминалом* для каждого процесса из этих групп. Процесс может читать из дескриптора управляющего терминала (т.е., с терминала), только если номер группы процессов терминала соответствует номеру группы процессов процесса. В противном случае при попытке чтения процесс будет блокирован. Изменяя PGID терминала, оболочка может переключать терминал между несколькими заданиями. Этот механизм и есть управление заданиями, реализуемое, например, в bash командами fg и bg.

Аналогично объединению в группы несколько родственных процессов могут быть объединены в сессию (session). Сессия – это набор из одного или более процессов; сессия может быть связана с устройством терминала. Главное назначение сессий – объединение вместе login-оболочки пользователя и процессов, порождаемых ею, а также создание изолированного окружения для процессов-демонов и их потомков. Любой процесс, не являющийся лидером группы процессов, может создать новую сессию с помощью системного вызова setsid(), становясь *лидером сессии* и единственным процессом в ней. Вместе с новой сессией всегда создается и новая группа процессов, номер PGID которой равен PID процесса-создателя сессии, и процесс-создатель сессии становится лидером этой группы. Таким образом, из механизма создания сессий следует, что все процессы из одной группы процессов всегда принадлежат к одной сессии.

Сессия может иметь связанный с ней управляющий терминал, используемый по умолчанию для связи с пользователем. Только процесс-лидер сессии может назначить (выделить) терминал для сессии, становясь таким образом *управляющим процессом* терминала. Каждый терминал может служить управляющим терминалом только для одной сессии одновременно. Когда некоторый процесс создает новую сессию и становится ее лидером, он отсоединяется от своего управляющего терминала, если такой имелся до создания новой сессии.

В качестве примера можно привести последовательность запуска системы (в несколько упрощенном виде). После загрузки ядра ОС запускается процесс init, который, в свою очередь, запускает несколько процессов связи с терминалом (например, getty). Эти процессы становятся лидерами сессий, выполняют подключение к своему терминалу и вызывают процесс login, обеспечивающий вход пользователя в систему. После входа пользователя login-процесс вызывает командный интерпретатор (например, bash) и пользователь начинает работать в системе. Так образуется сессия, в которой лидером является login-процесс. В эту сессию будут входить, кроме него и командного интерпретатора, процессы, запущенные пользователем с помощью команд. Эти процессы могут объединяться в несколько групп.

Процессом-демоном (daemon) в ОС Unix называют фоновый процесс, исполняющий некоторую работу и не связанный с терминалом. Например, процессами-демонами являются web-сервер (httpd), процесс lpd, обеспечивающий спулинг вывода на принтер, и pppd-процесс, поддерживающий протокол связи "точка-точка" (Point-to-Point Protocol). Существует соглашение, по которому имена программ, исполняемых как демоны, имеют на конце букву "d". Например, вышеперечисленные httpd, lpd, pppd, демон обеспечения связи по протоколу FTP-ftpd и другие. Однако это соглашение соблюдается не всегда.

Для всех процессов-демонов характерно отсутствие связи с терминалом, поэтому главным способом передачи им команд является передача сигналов. Чтобы иметь возможность передать сигнал системным вызовом kill() или командой kill, процесс или пользователь должен иметь соответствующие права. Практически часто используется отсылка сигналов от суперпользователя (root), например, командой sudo kill Сигнал PID_получателя.

Рассмотрим в общих чертах механизм запуска процесса-демона из командной строки (или скрипта). Вначале запущенный процесс исполняется в той же сессии, что и оболочка, из которой он был запущен, и связан с некоторым управляющим терминалом. Чтобы разорвать обе эти связи, используется системный вызов setsid(), описанный в unistd.h:

```
pid_t       setsid(void);
```

Системный вызов setsid() создает новую сессию и в ней новую группу процессов. Лидером новой сессии и новой группы становится вызвавший процесс, и номера новой сессии и группы процессов будут установлены равными его номеру PID. Вызвавший процесс и созданная сессия не будут иметь управляющего терминала. Возвращаемое значение setsid(): при успехе – номер созданной сессии (положительное число), при ошибке – значение -1. Единственная причина возникновения ошибки – равенство значения PID вызвавшего процесса и любого из существующих в системе номеров групп процессов; обычно это происходит, когда вызвавший процесс является лидером какой-либо группы процессов.

После исполнения setsid() процесс исполняется в новой сессии, став процессом-демоном. На его исполнение не будут влиять события, происходящие с оболочкой, откуда он был запущен, например, выход пользователя из системы. Рассмотрим следующую программу:

```
/* Файл dmn.c */
#include <unistd.h>
#include <stdio.h>

main()
{
    switch(fork()) {
        case -1: /* Ошибка fork() */
            printf("Ошибка fork()!\n"); return 1;
        default: /* Родительский процесс */
            return 0;
        case 0: /* Дочерний процесс */
            if(setsid()<0) {
                printf("setsid() error!\n");
                return 2;
            }
```

```
            else {
                /* Бесконечный цикл */
                for (;;) ;
            }
    }
}
```

Вначале программа исполняет системный вызов fork(), после которого исполняются уже два процесса: родительский, который сразу завершается, и дочерний. Дочерний процесс гарантированно не является лидером группы процессов (для чего и исполнялся вызов fork()) и может успешно исполнить вызов setsid(). После исполнения системного вызова setsid() процесс отключается от терминала и становится лидером новой сессии. Теперь он полностью независим от программы-оболочки, из которой была запущена наша программа.

Скомпилируем программу командой cc -o dmn dmn.c и запустим ее командой dmn. Просмотрим список процессов, включая процессы, не связанные с терминалом, например командой ps ux, и убедимся, что процесс dmn исполняется. Завершим теперь текущий сеанс работы (закрыв окно терминала или сеанс на текстовой консоли) и повторно войдем в систему под тем же именем пользователя. Снова просмотрим список процессов (командой ps ux) и убедимся, что процесс dmn все еще существует (исполняется). Это пример простейшего процесса-демона. Чтобы завершить его исполнение, отправим ему сигнал SIGTERM командой killall dmn.

Рассмотренную программу можно рассматривать как «заготовку» для написания более сложных программ-демонов. Существует библиотечная функция daemon(), выполняющая действия, аналогичные начальным (от fork() до for(;;)) в нашей программе. Функция daemon() описана в unistd.h следующим образом:

```
int  daemon(int nochdir, int noclose);
```

Если аргумент nochdir равен 0, то текущий каталог процесса меняется на корневой (‘/’). Если аргумент noclose равен 0, то стандартные потоки ввода, вывода и ошибок перенаправляются в /dev/null. Функция возвращает значение 0 при успешном выполнении и значение -1 при ошибке. С использованием данной функции наша программа приобретет более простой вид:

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

main()
{
      if ( daemon(0,0) == -1 ) {
          printf("Ошибка daemon() !\n");
          return 1;
      }
          /* Бесконечный цикл */
      for (;;) ;
}
```

Скомпилировав и запустив эту программу, легко убедиться, что она функционирует полностью аналогично предыдущей.

Контрольные вопросы

1. Что такое сигнал в ОС семейства Unix?
2. Как отправить сигнал процессу из Си-программы? Из командной строки (или скрипта)? Как отправить сигнал группе процессов?
3. Каковы варианты реакции процесса на получение сигнала?
4. Как заблокировать сигнал? Чем отличается блокировка сигнала от игнорирования сигнала?
5. Может ли процесс послать сигнал любому процессу? Каковы ограничения на пересылку сигналов между процессами?
6. Что такое группы процессов? Как реализуется переключение терминала между заданиями?
7. Что такое сессия? Как создать новую сессию? Любой ли процесс может это сделать?
8. Что такое процессы-демоны? Как создать процесс-демон? Как можно передать команду процессам-демонам (например, «перечитать файл конфигурации»)?

Библиографический список

1. *Операционная* система LINUX: Начальный курс пользователя: учебное пособие / Н.Н.Смирнова, Т.В.Панова, В.В.Касаткин; Балт. гос. техн. ун-т. СПб., 2005. 61 с.

2. *Таненбаум, Э.* Современные операционные системы. 2-е изд. / Э. Таненбаум. СПб.: Питер, 2005. 1038 с.

3. *Таненбаум, Э.* Операционные системы. Разработка и реализация. Классика CS. 3-е изд. / Э. Таненбаум, А. Вудхалл. СПб.: Питер, 2007. 704 с.

4. *Керниган, Б.* Язык программирования Си: пер. с англ. 3-е изд., испр. / Б. Керниган, Д. Ритчи. СПб.: Невский Диалект, 2001. 352 с.

ПРИЛОЖЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ СИСТЕМНЫХ ВЫЗОВОВ И БИБЛИОТЕЧНЫХ ФУНКЦИЙ

Вызов или библиотечная функция	Описан в файле	Действие
fork()	unistd.h	Создает копию процесса
getpid()	unistd.h	Возвращает номер процесса (PID)
getppid()	unistd.h	Возвращает номер родительского процесса (PPID)
execl() execle() execlp() execv() execvp() execve()	unistd.h	Заменяет образ текущего процесса на новый (запускает исполнимый файл). Различаются наборами аргументов
wait()	sys/wait.h	Ждет завершения любого дочернего процесса
waitpid()	sys/wait.h	Ждет завершения указанного процесса
pipe()	unistd.h	Создает неименованный канал (pipe)
dup2()	unistd.h	Связывает дескрипторы файлов
mkfifo()	sys/stat.h	Создает именованный канал (FIFO)
shmget()	sys/shm.h	Создает сегмент разделяемой памяти или возвращает дескриптор уже созданного сегмента
shmat()	sys/shm.h	Подключает сегмент разделяемой памяти
shmdt()	sys/shm.h	Отключает сегмент разделяемой памяти
shmctl()	sys/shm.h	Получает или изменяет информацию о сегменте разделяемой памяти; удаляет сегмент
semget()	sys/sem.h	Создает набор семафоров или возвращает дескриптор уже созданного набора
semctl()	sys/sem.h	Устанавливает или читает значение структур данных, связанных с семафорами; удаляет набор семафоров
semop()	sys/sem.h	Выполняет P/V-операции с семафорами (подъем/опускание)

Вызов или библиотечная функция	Описан в файле	Действие
socket()	sys/socket.h	Создает сокет
bind()	sys/socket.h (дополнительно sys/un.h для Unix-сокетов, netinet/in.h для IP-сокетов)	Назначает имя/адрес сокета
listen()	sys/socket.h	Включает прослушивание сокета
accept()	sys/socket.h	Принимает запрос на подключение к сокету
connect()	sys/socket.h	Выполняет подсоединение к существующему сокету
inet_aton()	netinet/in.h	Преобразует IP-адрес из точечной формы записи в значение, принятое в сети
htons()	netinet/in.h	Преобразует число в значение, принятое в сети
kill()	signal.h	Отправляет сигнал процессу или группе процессов
sigaction()	signal.h	Определяет реакцию процесса на полученный сигнал
sigemptyset()	signal.h	Очищает маску сигналов
sigfillset()	signal.h	Устанавливает маску для всех сигналов
sigaddset()	signal.h	Добавляет сигнал к маске
sigdelset()	signal.h	Удаляет сигнал из маски
sigismember()	signal.h	Проверяет наличие сигнала в маске
sigprocmask()	signal.h	Блокирует/разблокирует сигналы по маске
setpgid()	unistd.h	Изменяет номер группы (PGID) процесса; может создать группу процессов
setpgrp()	unistd.h	Создает новую группу процессов
setsid()	unistd.h	Создает новую сессию
daemon()	unistd.h	Обеспечивает исполнение программы как «демона»

О Г Л А В Л Е Н И Е

Романов Сергей Леонидович

Программирование для операционной системы Unix

Редактор *Г.М. Звягина*
Корректор *Л.А. Петрова*
Подписано в печать 27.06.2011. Формат бумаги 60х84/16. Бумага документная.
Печать трафаретная. Усл. печ. л. 4,3. Тираж 100 экз. Заказ № 153.
Балтийский государственный технический университет
Типография БГТУ
190005, С.-Петербург, 1-я Красноармейская ул., д. 1