ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА» (САМАРСКИЙ УНИВЕРСИТЕТ)

ПРОГРАММИРОВАНИЕ МНОГОЗАДАЧНОСТИ В UNIX

Самара 2017

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

ПРОГРАММИРОВАНИЕ МНОГОЗАДАЧНОСТИ В UNIX

Составитель К.Е. Климентьев

Самара
Издательство Самарского университета
2017

УДК 004.451.46
ББК 32.973

Составитель ***К.Е.Климентьев***

Рецензент: к.т.н., доцент А.В. Баландин

Программирование мнозадачности в UNIX: [Электронный ресурс]: метод. указания / сост. *К.Е. Климентьев*. - Самара: Изд-во Самарского университета, 2017. – 44 с. : ил. Электрон. текстовые и граф. дан. (Кбайт).- 1 эл. опт. диск (CD-ROM).

Методические указания к лабораторному практикуму по курсу «Системы реального времени»

Предназначены для студентов, изучающих в рамках направления подготовки 09.03.01 «Информатика и вычислительная техника» курс «Системы реального времени» и прочие курсы аналогичной тематики .

Содержат необходимый теоретический и справочный материал для выполнения лабораторных работ. Также могут быть использованы в курсовом проектировании и при разработке выпускной квалификационной работы.

Подготовлены на кафедре информационных систем и технологий.

УДК 004.056.55
ББК 32.973

Оглавление

Задание на лабораторную работу

Цель: практическое изучение методов и средств многозадачного программирования в UNIX-подобных операционных системах.

Задание: требуется написать, отладить и продемонстрировать преподавателю систему из нескольких независимых процессов (**A**) или потоков (**B**), совместно решающих квадратное уравнение (**L**), вычисляющих гипотенузу по двум катетам (**M**) или дисперсию выборки из трех чисел (**N**). Процессы или потоки должны отображать на экране ход своего выполнения в виде отладочных сообщений. Система должны состоять из:

- главного процесса или потока, принимающего с клавиатуры исходные данные и выводящего на экран результат;
- нескольких служебных процессов или потоков, способных по отдельности выполнять элементарные арифметические действия - сложение, вычитание, умножение, деление, вычисление квадратного корня и т.п.

При этом использовать способ синхронизации **C**, **D**, **E** или **F** и способ передачи данных между процессами или потоками - **G**, **H**, **I, J** или **K**.

Таблица 1. Индивидуальные задания на лабораторную работу

Вариант	Задание	Вариант	Задание	Вариант	Задание
1	ACIL	10	AJL	19	ACJL
2	ADJM	11	AHKM	20	ADIM
3	BCGN	12	BCKN	21	BCJN
4	AFKL	13	AFGL	22	ADHL
5	ACJM	14	ACHM	23	AFGM
6	BDJN	15	BDGN	24	BDKN
7	ADKL	16	ADIL	25	ACGL
8	AFIM	17	AFJM	26	ADHM
9	BEKN	18	BEJN	27	BEGN

Условные обозначения:

- **A** – независимые процессы;
- **B** – потоки одного процесса;
- **C** – мьютексы;
- **D** – семафоры;
- **E** – условные переменные;
- **F** – сигналы;
- **G** – обычные файлы;
- **H** – разделяемая память;
- **I** – каналы FIFO;
- **J** – сокеты;
- **K** – очереди сообщений;
- **L** – квадратное уравнение: $(x_1, x_2) = (-b \pm \sqrt{D}) / 2a;\ D = b^2 - 4ac;$
- **M** – гипотенуза по двум катетам: $c = \sqrt{a^2 + b^2}$;
- **N** – выборочная дисперсия трех чисел: $\tilde{\sigma}_x^2 = \frac{1}{2}\sum_{i=1}^{3}(x_i - \bar{x})^2;\ \bar{x} = \frac{1}{3}\sum_{i=1}^{3} x_i$.

Теоретическая часть

1.1. Основные определения

Многозадачность – это режим работы операционной системы, при котором параллельно могут выполняться несколько программ (задач). Для создания эффекта одновременности в условиях, когда количество задач больше количества процессоров, задачи выполняются поочередно короткими фрагментами. Существуют две разновидности многозадачности:

- *кооперативная* – при которой задача сама принимает решение о передаче управления другой задаче;
- *вытесняющая* – при которой операционная система через небольшие *кванты времени* принудительно прерывает выполнение одной задачи, чтобы передать управление другой.

В современных версиях операционных систем UNIX реализована вытесняющая многозадачность.

Процесс – задача, монопольно занимающая выделенное ей адресное пространство и владеющая некоторым набором ресурсов. Все процессы конкурируют друг с другом за процессорное время. Они могут порождать и уничтожать друг друга, соответственно, существуют *процессы-предки* и *процессы-потомки*. Самым первым процессом в UNIX, являющимся прямым или косвенным предком всех остальных, является «init». Процессы, не имеющие прямого предка (например, он уже завершен), называются «*зомби-процессами*».

Потоки – задачи, разделяющие общее адресное пространство. Наличие их является требованием стандарта POSIX. Однако потоки реализованы не во всех версиях UNIX и UNIX-подобных систем.

В библиотеки UNIX включены многочисленные функции, обеспечивающие синхронизацию задач и обмен данными между ними. Однако развитие UNIX-подобных операционных систем выполняется различными группами разработчиков, вследствие чего отдельные механизмы в конкретной версии операционной системы могут отсутствовать. В данных методических указаниях описаны две группы функций, отчасти дублирующие друг друга по функциональным возможностям:

- реализованные в классическом стиле операционной системы UNIX SYSTEM 5;
- реализованные в соответствии с требованиями стандарта POSIX.

1.2. Компиляция и отладка программ

Рекомендуется компилировать программы при помощи компилятора GCC, входящего в состав большинства бесплатных UNIX-подобных систем, таких как Free BSD, Linux и пр. Возможно и использование компилятора Watcom C/C++, поставляемого вместе с QNX и Mac OS X. В принципе,

допустимо использование и иных языков программирования, таких как XDS Modula-2 for Linux.

Примеры команд компиляции:

- **cc myprog.c** – простая компиляция, результат попадет в файл «a.out»;
- **cc myprog.c –o myprog** – с указанием выходного файла;
- **cc –lpthreads myprog.c** – с подключением потоковой библиотеки;
- **cc –lrt myprog.c** – с подключением библиотек асинхронного в/в;
- **cc –lm myprog.c** – с подключением математической библиотеки;
- **cc –lGL -lglut myprog.c** – с подключением библиотек OpenGL;
- **cc myprog.c –L/usr/X11R6/Lib –lX11** – с подключением библиотеки XLib;
- **cc –Wa, –ahls=myprog.lst myprog.c** – с получением ассемблерного листинга;
- **cc –c myprog.c** – в объектный модуль, без компоновки;
- **cc main.o obj1.o obj2.o** – компоновка из нескольких объектных модулей;
- **cc –g myprog.c** - с включением отладочной информации.

Для отладки программ целесообразно использовать значение переменной **errno**, которая при возникновении ошибки в ядре операционной системы содержит числовой код этой ошибки. Коды наиболее часто встречающихся ошибок см. в Приложении А.

Так же можно воспользоваться следующими функциями.

char* strerror(int errnum) – возвращает указатель на строку, содержащую текстовое описание ошибки (на английском языке) для указанного числового кода. Прототип в «string.h». Параметр:

errnum – числовой код ошибки.

void perror(const char *str) - возвращает указатель на строку, содержащую текстовое описание ошибки (на английском языке) для указанного числового кода и одновременно выводит этот текст на стандартное устройство вывода ошибок **stderr** (по умолчанию это экран консоли).

Пример применения.

```
#include <stdio.h>
#include <errno.h>
main() {
 int f;
 f =open(…);
 if (errno==EINVAL) printf("\nФункции передан неверный аргумент");
…
}
```

2. Создание и уничтожение процессов

Для программирования с использованием функций, описанных в этом разделе, необходимо подключить файлы «**stdlib.h**» или «**unistd.h**».

Простейший способ запуска процесса – использование функции **system()**, использующей в качестве «посредника» интерпретатор команд пользователя. Таким образом, работа функции **system()** идентична «ручному» запуску процесса из командной строки консоли.

int system (char *s) – выполняет команду пользователя, заданную в строке s. Возвращает –1 в случае ошибки или 127 в случае невозможности запуска командного интерпретатора.

Следующая группа функций обеспечивают запуск процесса и работу с ним при помощи прямого запроса к ядру операционной системы, минуя интерпретатор команд.

pid_t fork(void) – создает точную копию адресного пространства процесса, вызвавшего функцию, и запускает в нем дочернюю копию текущего (родительского) процесса с команды, следующей за fork(). Родительская копия также продолжает выполнение с той же точки. Возвращает:

- для родительского процесса - системный идентификатор процесса PID, который будет загружен в это адресное пространство;
- для дочернего процесса - 0.

```
int execlp( const char *file, const char *arg0,
                ... const char *argN,(char *) NULL );
int execvp( const char *file, char *argv[] );
int execl( const char *path, const char *arg0,
                ... const char *argN,(char *) NULL );
int execv( const char *path, char *argv[] );
int execle( const char *path, const char *arg0,
                 ... const char *argN,(char *)NULL, char * envp[] );
int execve( const char *path, char *argv[], char *envp[] )
```

- загружают во вновь созданное (при помощи fork()) адресное пространство программу и запускают ее на выполнение. Параметры:

- file – строка имени файла, находящегося в текущем каталоге или в одном из умалчиваемых каталогов;
- path – полное имя файла, включая путь к нему;
- env, envp – массив (строка) переменных среды;
- argn – указатель на количество параметров, передаваемых процессу;
- argv, arg0, arg1… - строки параметров, передаваемых процессу.

Все эти функции в случае ошибки возвращают –1. При этом переменная errno может принять значения: ENOEXEC – неверный формат программы; ENOMEM – не хватает памяти.

pid_t wait (int *status_ptr) – переводит родительский процесс в состояние ожидания завершения любого дочернего процесса. Для использования этой функции необходимо подключить «wait.h».

pid_t getpid(void) – возвращает для обратившегося процесса его системный идентификатор PID.

void exit (int status) – завершает работу текущего процесса, возвращая процессу-родителю статус. Автоматически закрываются все открытые файлы, освобождаются захваченные области памяти и т.п.

void _exit (int status) – завершает работу текущего процесса, возвращая процессу-родителю статус. «Сборка мусора» не выполняется.

Примеры работы с процессами.

```
// Запуск процесса
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

pid_t p, q;
char *e[]={"",""};

main() {
 p = fork(); /*Копирование адресного пространства и процесса*/
 if (p)      /* Родитель получает PID  ребенка*/
   wait(&q); /* Ждать завершения потомков */
 else        /* Ребенок получает 0 */
   execv("./hello", e); /* Ребенок замещает себя другой программой*/
}
```

// Завершение процесса

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
pid_t p, q;
char *e[]={"",""};
main() {
 p = fork(); /*Копирование адресного пространства и процесса*/
 if (p) {    /* Родитель получает PID  ребенка*/
  sleep(10); /* Задержка на 10 сек */
  kill (p, SIGKILL);
  }
 else        /* Ребенок получает 0 */
   execv("./dull", e); /* Ребенок замещает себя другой программой*/
}
```

3. Потоки

Потоки (или нити, треды) – задачи, использующие общее адресное пространство.

3.1. Потоки в стиле SYSTEM V

Потоки запускаются при помощи функции:

int clone (int (*fn) (void *), void *child_stack, int flags, void *arg) – создает точную копию адресного пространства процесса, вызвавшего функцию, и запускает в нем дочернюю копию текущего (родительского) процесса с начала процедуры, указанной в параметре fn. Таким образом, clone() запускает параллельный процесс, а в нем – только указанный поток.

Параметры:

- fn – процедура, поток которой запускается в дочернем процессе;
- child_stack – размер стека для запускаемого потока;
- flags – младший байт содержит сигнал (обычно, 0 или SIGCHLD), который посылается родительскому процессу после завершения потока; старший байт – битовые флаги CLONE_PARENT – новый потомок будет копией старого родителя; CLONE_FS – потомок разделяет с родителем файловую систему; CLONE_FILES – потомок разделяет с родителем открытые файловые дескрипторы; CLONE_SIGHAND – потомок разделяет с родителем обработчики сигналов; CLONE_VM – потомок разделяет с родителем память; CLONE_PID – потомок имеет с родителем общий PID; CLONE_THREAD – потомок размещается в той же группе потоков, что и ролдитель;
- arg – ссылка на параметры, передаваемые в поток.

Возвращает:

- для родительского процесса - системный идентификатор процесса PID, который будет загружен в это адресное пространство;
- -1 в случае ошибки.

Внимание: в случае применения CLONE_VM обработчик сигналов является нереентерабельным.

Пример запуска потока методом клонирования.

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

/*Глобальные данные*/
char stack[10000], i, j;

/*Код и данные потока*/
```

```
int mythread(void * thread_arg) {
 for(int i=0;i<10;i++){
  sleep(1);
  printf("\nMythread: i=%d",i);
 }
 printf("Thread1 finished\n");
}

/*Код и данные «главного» потока*/
int main() {
 clone(mythread,(void*)
    (stack+10000-1),CLONE_VM|CLONE_FS|CLONE_FILES|LONE_SIGHAND,NULL);
for(j=0;j<10;j++){
 sleep(1);
 printf("Main thread: j=%d\n",j);
 }
}
```

3.2. Потоки в стиле POSIX

Потоки в стиле POSIX не реализованы в некоторых «классических» UNIX-системах. Для работы с такими потоками в Linux необходимо подключение заголовочного файла «**pthread.h**» и библиотеки «**pthread**» (например, «**cc proba.c -pthread**»). Используются функции:

int pthread_create(pthread_t * thread, pthread_attr_t *attr, void* (*start_routine) (void*), void * arg) - создаёт новый поток. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметры:

- thread – возвращаемый идентификатор потока (если NULL, то не возвращается);
- attr – атрибуты запуска (0 – по умолчанию);
- start_routine – адрес функции потока;
- arg – аргумент, передаваемый функции.

Возвращает 0 в случае успеха.

pthread_t pthread_self(void) - возвращает потоку собственный идентификатор.

void pthread_exit(void *retval) - завершает текущий поток. Параметр:

- retval – адрес для кода возврата.

int pthread_join(pthread_t th, void **thr_return) - ожидает завершения потока. Параметры:

- ht – идентификатор потока;
- thr_return – указатель на код возврата.

int pthread_kill (pthread_t thread, int signo) - посылает потоку сигнал. Параметры:

- thread – идентификатор потока;

- signo – код сигнала.

int sched_yield(void) – передать управление другим потокам, не ожидая завершения кванта. При успешном выполнении возвращает 0, а в случае неудачи –1.

int pthread_cancel(pthread_t thread_id) – принудительно завершает работу потока. Параметр:

- thread – идентификатор потока;

void pthread_cleanup_push(void (*handler) (void *), void *arg) – устанавливает обработчик, вызываемый перед принудительным завершением текущего потока. Параметры:

- handler – функция обработчика;
- arg – аргумент, передаваемый обработчику.

void pthread_cleanup_pop(int execute) – сбрасывает ранее установленный обработчик, вызываемый перед принудительным завершением текущего потока. Параметр:

- execute – вызвать ли обработчик перед сбросом.

Пример запуска потоков средствами POSIX Pthreads.

```
/*К омпиляция: gcc primer.c -lpthreads */
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

void *func(void *arg) {
 int loc_id = * (int *) arg;
 while (1) {
  printf("Thread %i is running\n", loc_id);
  sleep(1);
 }
}

int main() {
 int id1, id2, result;
 pthread_t thread1, thread2;
 id1 = 1;
 result = pthread_create(&thread1, NULL, func, &id1);
 pthread_detach(thread1);
 id2 = 2;
 result = pthread_create(&thread2, NULL, func, &id2);
 pthread_detach(thread2);
}

/*Пример с общим адрсным пространством, но разными процедурами*/
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
```

```
void *func1() { /* Поток 1 */
 int i;
 for (i=0;i<10;i++) { printf("Thread 1 is running\n"); sleep(1); }
}

void *func2() { /* Поток 2 */
 int i;
for (i=0;i<10;i++) { printf("Thread 2 is running\n"); sleep(1); }
}

int main() {
 int result, status1, status2;
 pthread_t thread1, thread2;
 result = pthread_create(&thread1, NULL, func, NULL);
 result = pthread_create(&thread2, NULL, func, NULL);
 pthread_join(thread1, &status1);
 pthread_join(thread2, &status2);
 printf(”\nПотоки завершены с %d и %d”, status1, status2);
}
```

4. Синхронизация задач
4.1. Сигналы

Сигналы – это программный аналог прерывания. Сигнал может быть послан от одного процесса другому или всем сразу. В соответствии с требованиями POSIX возможно так же послать сигнал отдельному потоку – см. описание функции **pthread_kill()**. Сигналы посылаются со стороны операционной системы процессам, при выполнении которых обнаружены исключительная ситуация. Имена и числовые значения сигналов определены в файле «**signal.h**» (см. Приложение Б).

Для работы с сигналами используются следующие функции:

int kill (pid_t pid, int sig) – посылает указанный сигнал процессу с указанным идентификатором. Параметры:

- pid – идентификатор процесса (частные случаи: 0 – всем процессам из той же группы, -1 – всем процессам, кому данный процесс имеет право посылать сигналы);
- sig – код сигнала.

Возвращает 0 в случае успеха или –1 в случае ошибки.

void abort(void) – послать сигнал SIGABRT**.**

int raise (int sig) – послать указанный сигнал самому себе. Возвращает 0 в случае успеха.

unsigned alarm(unsigned secs) – запускает «будильник», посылающий самому себе сигнал SIGALRM через указанный интервал времени. Параметр:

secs - интервал времени.

Возвращает время до срабатывания ранее запущенного «будильника» или 0, если «будильник» еще не запущен.

int sigqueue(pid_t pid, int signum, const union sigval value) – посылает указанный сигнал указанному процессу, но если процесс уже обрабатывает другой сигнал, этот будет поставлен в очередь. Параметры:

- pid – идентификатор процесса (частные случаи: 0 – всем процессам из той же группы, -1 – всем процессам, кому данный процесс имеет право посылать сигналы);
- sig – код сигнала;
- value – список значений, передаваемых с сигналом. Тип:

```
union sigval {
  int sival_int;      // Значение
  void *sival_ptr; // Указатель на следующий
 }
```

int sigaction (int sig, const struct sigaction *restrict act, struct sigaction *restrict oact) – позволяет устанавливать и изменять способ обработки сигнала в стиле стандарта POSIX. Параметры:

- sig – обрабатываемый сигнал;
- restrict_act – новый способ обработки сигнала;
- restrict_oact – старый способ обработки сигнала.

Тип sigaction:

```
struct sigaction {
  void (*sa_handler) (int);// Указатель на простой обработчик
  sigset_t sa_mask;          // Блокируемые во время обработки сигналы
  int sa_flags;              // Флаги, влияющие на поведение сигнала
  void (*sa_sigaction) (int, siginfo_t *, void *); // Указатель на обработчик
}
```

int sigemptyset(sigset *set) – обнуляет маску, блокирующую сигналы. Все сигналы будут заблокированы. Параметр:

- set – маска, должна быть передана в sigaction();

Возвращает 0 в случае успеха и –1 в случае ошибки.

int sigfillset(sigset *set) – заполняет маску, блокирующую сигналы. Все сигналы будут разрешены. Параметр:

- set – маска, должна быть передана в sigaction();

Возвращает 0 в случае успеха и –1 в случае ошибки.

int sigaddset(sigset_t *set, int signum) – выставляет в маске бит разрешения определенного сигнала. Сигнал будет разблокирован. Параметры:

- set – маска, должна быть передана в sigaction();
- signum – номер сигнала.

Возвращает 0 в случае успеха и –1 в случае ошибки.

int sigismember(sigset_t *set, int signum) – проверяет блокировку сигнала. Параметры:

- set – маска;
- signum – номер сигнала.

Возвращает: 1 – сигнал разблокирован, 0 – сигнал заблокирован и –1 в случае ошибки.

int sigdelset(sigset_t *set, int signum) – сбрасывает в маске бит разрешения определенного сигнала. Сигнал будет заблокирован. Параметры:

- set – маска, должна быть передана в sigaction();
- signum – номер сигнала.

Возвращает 0 в случае успеха и –1 в случае ошибки.

int pause (void) – вызывает задержку текущего процесса до прихода любого сигнала. В случае ошибки возвращает –1.

int sigwait(const sigset_t *set, int *signum) – вызывает ожидание конкретного сигнала. Параметры:

- set – набор ожидаемых сигналов;
- signum – номер полученного сигнала.

В случае успеха возвращает 0, иначе возвращает код ошибки.

int signal (int signum, void (*act)(int)) – устанавливает упрощенный обработчик указанного сигнала. После прихода сигнала все установки сбрасываются. Параметры:

- signum – номер ожидаемого сигнала;
- act – процедура обработки сигнала.

Возвращает старое действие, установленное для сигнала, или SIG_ERR в случае ошибки.

int sigset (int signum, void (*act)(int)) – устанавливает обработчик указанного сигнала. Параметры:

- signum – номер ожидаемого сигнала;
- act – процедура обработки сигнала.

Возвращает старое действие, установленное для сигнала, или SIG_ERR в случае ошибки.

Пример работы с сигналами.

```
#include <stdio.h> /* Родительский процесс */
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
pid_t p;
char *e[]={"",""};
```

```
main() {
 p = fork();
 if (p) { /* parent */
   sleep(1);
   kill(p,SIGUSR1);
   }
 else  /* child */
   execv("./sighand", e);
 }
```

```
#include <stdio.h> /*Дочерний процесс*/
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

volatile int flag = 0;

void sigint_handler(int sig) {
 flag = sig;
}

int main(void) {
    struct   sigaction sa;
    sigset_t set;

    sigemptyset(&set);         // Clear set of signals
    sigaddset(&set, SIGUSR1); // Add signal to set
    sa.sa_handler = sigint_handler;
    sa.sa_flags = 0;
    sa.sa_mask = set;
    sigaction(SIGUSR1, &sa, NULL);
    while (!flag);             // Wait for signal
    printf("\nReceived signal %d", flag);
    }
```

4.2. Семафоры

Семафор – наиболее универсальный механизм, обеспечивающий синхронизацию задач. В UNIX-системах существуют, по крайней мере, два набора функций для работы с семафорами.

4.2.1. Семафоры в стиле System V

Эти функции сохранились в UNIX-системах с 1970-годов и обеспечивают «традиционную» работу с семафорами. Их описания находятся в заголовочных файлах «**sys/ipc.h**» и «**sys/sem.h**».

key_t ftok (const char *path, int id) – создает ключ для объекта ядра (семафора, очереди сообщений и т.п.) операционной системы. Параметры:

- path – имя объекта (существующий и доступный файл);
- id – желаемый номер объекта (>0).

Возвращает –1 в случае ошибки или заполненный key_t.

int semget (key_t key, int nsems, int flags) – создает набор (неициализированных!) семафоров. Параметры:

- key – ключ для набора семафоров;
- nsems – количество семафоров в наборе;
- flags – битовые флаги (IPC_CREAT – создать новый семафор, IPC_EXCL – не создавать копию существующего семафора).

Возвращает –1 в случае ошибки или идентификатор семафора.

int semctl (int semid, int semnum, int cmd, union semun arg) - позволяет выполнять операции, определенные командой cmd над набором семафоров, указанным в semid. Первый семафор в наборе обозначен величиной 0 для semnum. Параметры:

- semid – идентификатор семафора;
- semnum – номер семафора;
- cmd – действие над семафорами (IPC_STAT - получить информацию о наборе семафоров; IPC_SET - переустановить характеристики набора семафоров; IPC_RMID - удалить из системы набор семафоров; SETVAL - устанавливает значение семафорной переменной; GETVAL – получает значение семафорной переменной; GETALL - получает значение для всех семафоров в наборе; SETALL – устанавливает значение для всех семафоров в наборе; GETPID – получает PID последнего процесса, обратившегося к семафору);
- arg – аргумент команды типа «semun»:

```
union semun {
        int val;                    // Целое число
        struct semid_ds *buf;       // semid
        unsigned short array        // массив байтов
        }
                и
struct semid_ds {
        struct ipc_perm;            // Права доступа
        unsigned short sem_nsems;   // Размер набора
        time_t sem_otime;           // Последнее обращение к semop
        time_t sem_ctime            // Последнее обращение к semctl
        }.
```

int semop (int semid, struct sembuf *sops, size_t nsops) – управляет семафором или группой семафоров. Параметры:

- semid – идентификатор семафора;
- sembuf – буфер операций типа «sembuf»:

```
struct sembuf {
        unsigned short sem_num; // Номер семафора
        short sem_op;           // Операция
```

short sem_flg // Флаги (рекомендуется SEM_UNDO)
},

возможны операции: >0 – увеличить значение семафора на sem_op; 0 – процесс блокируется, пока семафор не обнулится; <0 – процесс блокируется, пока текущее значение семафора не сравняется или не превысит |sem_op|, если это уже достигнуто, то |sem_op| вычитается из значения семафора.

- nsops - количество операций в буфере.

Пример семафоров в стиле SYSTEM 5.

```
#include <sys/ipc.h> /*System V semaphore*/
#include <sys/sem.h>
#include <sys/types.h>
#include <stdio.h>

void *mythread() { /*New thread*/
 key_t k1;
 int s1;
 struct sembuf b1;

 fprintf(stdout,"Start..."); fflush(stdout);
 k1 = ftok(".", 0);
 s1 = semget(k1, 1, 0644 | IPC_EXCL);
 b1.sem_num = 0;          /* Number of semaphore   */
 b1.sem_op  = 0;          /* Wait while s1<>0      */
 b1.sem_flg = 0;
 fprintf(stdout, "\nWait..."); fflush(stdout);
 semop(s1, &b1, 1);
 fprintf(stdout, "\ndone!\n"); fflush(stdout);
}

int main() {
 int s;
 struct sembuf b;
 int u;
 key_t k;
 pthread_t t;
 k = ftok(".", 0);
 s = semget(k, 1, 0644 | IPC_CREAT | IPC_EXCL);
 u = 1;
 semctl(s, 0, SETVAL, u);
 pthread_create( t, 0,  &mythread, 0);
 sleep(5);
 u = 0;
 semctl(s, 0, SETVAL, u );
 sleep(1);
 semctl(s, 0, IPC_RMID, NULL);
 }
```

4.2.2 Семафоры в стиле POSIX

Эти функции появились в UNIX-системах в связи с требованиями стандарта POSIX. Их описания находятся в заголовочном файле «**semaphore.h**». Они присутствуют не во всех UNIX-системах.

sem_t *sem_open (char*name, int flags, mode_t mode, unsigned int value) создает новый или открывает существующий семафор. Возвращает указатель на sem_t, или -1 в случае неудачи. Параметры:

- name – имя семафора (должно начинаться с символа «/»);
- flags - 0, если семафор уже существует; O_CREAT – создать новый; O_EXCL – возвратить ошибку, если семафор существует.
- mode - это права доступа;
- init_value – начальное значение семафора.

int sem_getvalue (sem_t *idp, int*valuep) - определяет текущее значение семафора. Всегда возвращает 0. Параметры:

- idp – семафор;
- valuep – значение семафора.

int sem_wait (sem_t *idp) - уменьшает значение семафора на единицу. Если его значение меньше или равно 0, то вызывающий процесс блокируется. Функция всегда возвращает 0. Параметр:

- idp – семафор.

int sem_trywait (sem_t *idp) – если значение >0, то уменьшает значение семафора на единицу и возвращает 0; если значение уже меньше или равно 0, то возвращает EAGAIN. Параметр:

- idp – семафор.

int sem_post (sem_t *idp) - увеличивает значение семафора на единицу. В случае успеха возвращает 0, в случае слишком большого значения возвращает *ERANGE*. Параметр:

- idp – семафор.

int sem_close (sem_t *idp) – отключает семафор. В случае успеха возвращает 0, а в случае неудачи -1. Параметр:

- idp – семафор.

int sem_unlink (char *name) - удаляет семафор из системы. В случае успешного выполнения функция возвращает 0, а в случае неудачи -1. Параметр:

- name – имя семафора (должно начинаться с символа «/»);

int sem_init (sem_t *addr, int pshared, unsigned int value) – создает неименованный семафор. Параметры:

- addr – область памяти под семафор;
- pshared – доступность для других процессов;
- value – начальное значение.

В случае успешного выполнения данная функция возвращает 0, а в случае неудачи: EINVAL превышено максимально возможное количество семафоров в системе; ENOSYS аргумент pshared не равен нулю.

int sem_destroy (sem_t* idp) – удаляет неименованный семафор, созданный при помощи sem_init(). В случае успеха возвращает 0, а если есть заблокированные семафором задачи, то EBUSY.

Пример семафоров в стиле POSIX.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>

sem_t empty, full;
int data;

void *Producer(void *arg){            // Сначала empty=1 full=0
 while(1)  {
  sem_wait(&empty);                    // empty:=empty-1
  data++;
  sem_post(&full);                     // full:=full+1
 }
}

void *Consumer(void *arg){ // Сначала empty=1 full=0
 while(1)  {
  sem_wait(&full);    // full:=full-1
  printf("\nData=%d", data);
  sem_post(&empty);    // empty:=empty+1
 }
}

int main(){
 pthread_t thread1, thread2;
 sem_init(&empty,SHARED,1); // Установлен в 1
 sem_init(&full,SHARED,0); // Установлен в 0
 data = 0;
 pthread_create(&thread1, NULL, &Producer, NULL);
 pthread_create(&thread2, NULL, &Consumer, NULL);
 sleep(60);
 pthread_cancel(thread1);
 pthread_cancel(thread2);
```

4.3. Мьютексы

Мьютекс – двоичный семафор, способный принимать значения «свободно» и «занятно». Наличия мьютексов требует стандарт POSIX. Для работы с потоками необходимо подключение заголовочного файла «**pthread.h**». Для синхронизации используются функции:

int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr) - инициализирует мьютекс. Параметры:

- mutex - возвращаемый идентификатор мьютекса;
- mutexattr – это указатель на атрибуты или 0, если по умолчанию. Атрибуты: PTHREAD_MUTEX_INITIALIZIER – запрещает потоку повторно захватывать критическую секцию; PTHREAD_RECURSIVE_ MUTEX_INITIALIZIER_NP – разрешает потоку многократно захватывать критическую секцию, при этом увеличивается счетчик захватов; PTHREAD_ERRORCHECK_MUTEX_INITIALIZIER_NP – запрещает снимать блокировку потоку, который ее не создавал.

int pthread_mutex_lock (pthread_mutex_t *mutex) – попытка захвата мьютекса. Если мьютекс уже захвачен, то поток блокируется до его освобождения. Параметр:

- mutex – идентификатор мьютекса.

int pthread_mutex_trylock (pthread_mutex_t *mutex) – не блокирующая попытка захвата мьютекса. Если мьютекс уже захвачен, то возвращает код ошибки EBUSY, иначе 0. Параметр:

- mutex – идентификатор мьютекса.

int pthread_mutex_unlock (pthread_mutex_t *mutex) - освобождает мьютекс. Параметр:

- mutex – идентификатор мьютекса.

int pthread_mutex_destroy (pthread_mutex_t *mutex) – удаляет мьютекс. Параметр:

- mutex – идентификатор мьютекса.

Пример использования мьютексов.

```
#include <stdlib.h> /* Just mutex */
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

int data = 0;
pthread_mutex_t mutex;

void *Producer() {
  while(1)  {
   fprintf(stderr, "\n1-st Wait");
   pthread_mutex_lock(&mutex);
   fprintf(stderr, "\nPing: %d", data++);
   pthread_mutex_unlock(&mutex);
   sleep(1);
   }
  }

void *Consumer() {
 while(1)  {
  fprintf(stderr, "\n2-nd Wait");
  pthread_mutex_lock(&mutex);
```

```
  fprintf(stderr, "\nPong: %d", data--);
  pthread_mutex_unlock(&mutex);
  sleep(1);
 }
}

int main() {
  pthread_t thread1, thread2;
  pthread_mutex_init(&mutex, NULL);
  pthread_create(&thread1, NULL, &Producer, NULL);
  pthread_create(&thread2, NULL, &Consumer, NULL);
  sleep(7);
  pthread_cancel(thread1);
  pthread_cancel(thread2);
 }
```

4.4. Переменные состояния

Переменные состояния – это «флаги», доступные одновременно всем задачам. Как известно, алгоритмы синхронизации, проверяющие единственный флаг готовности, в общем случае некорректны, т.к. могут приводить к взаимоблокировкам и «гонкам». Поэтому в реальности переменные состояния используются как признаки активации и деактивации мьютексов.

int pthread_cond_init (pthread_cond_t *cond, pthread_condattr_t *cond_attr) - инициализирует переменную состояния. Параметры:

- cond – возвращаемый идентификатор переменной состояния;
- cond_attr – атрибуты переменной состояния.

int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex) - блокирует поток на переменной состояния, ожидая его наступления. Параметры:

- cond – идентификатор переменной состояния;
- mutex – идентификатор мьютекса.

int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime) - блокирует поток на переменной состояния на определенное время. Параметры:

- cond – идентификатор переменной состояния;
- mutex – идентификатор мьютекса;
- abstime – время.

int pthread_cond_signal (pthread_cond_t *cond) – сигнализирует о наступлении состояния и снимает блокировку с потока, ожидающего переменную состояния. Параметр:

- cond – идентификатор переменной состояния.

int pthread_cond_broadcast (pthread_cond_t *cond) - снимает блокировку со всех потоков, ожидающих переменную состояния. Параметр:

- cond – идентификатор переменной состояния.

int pthread_cond_destroy (pthread_cond_t *cond) – удаляет переменную состояния. Параметр:

- cond – идентификатор переменной состояния.

Пример применения переменных состояния.

```
#include <stdlib.h> /* Condition with mutex */
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
int data;
pthread_mutex_t mutex;
pthread_cond_t cond;

void *Producer() {
 while(1)  {
  fprintf(stderr, "\n1-st Wait...");
  pthread_mutex_lock(&mutex);
  fprintf(stderr, "\nPing: %d", data++);
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
  sleep(1);
  }
 }
 void *Consumer() {
  while(1)  {
   fprintf(stderr, "\n2-nd Wait...");
   pthread_mutex_lock(&mutex);
   pthread_cond_wait(&cond, &mutex);
   fprintf(stderr,"\nPong: %d", data--);
   pthread_mutex_unlock(&mutex);
   sleep(1);
 }
}
int main()  {
 pthread_t thread1, thread2;
 pthread_cond_init(&cond, NULL);
 pthread_mutex_init(&mutex, NULL);
 data = 0;
 pthread_create(&thread1, NULL, &Producer, NULL);
 pthread_create(&thread2, NULL, &Consumer, NULL);
 sleep(7);
 pthread_cond_destroy(&cond);
}
```

5. Информационный обмен между задачами

По умолчанию, каждый процесс выполняется в своем виртуальном адресном пространстве и, для обмена информацией с другими процессами, ему предоставляются операционной системой специальные средства.

5.1. Файловый ввод-вывод

Система организации файлов – базовый механизм межзадачного обмена. Он обеспечивает доступ не только к наборам данных, размещенных на носителях информации, но и к устройствам, объектам ядра операционной системы и т.п.

int open(char *path, int flags, mode_t perms) – открывает существующий или создает новый файл. Указатель чтения-записи устанавливается в 0. Возвращает дескриптор открытого файла или –1 в случае ошибки. Параметры:

- path – имя файла;
- flags – битовые флаги доступа (O_CREAT – создать новый, O_EXCL – при создании существующего файла вернуть ошибку, O_APPEND – только для добавления, O_RDONLY – только для чтения, O_WRONLY – только для записи, O_RDWR – для чтения-записи, O_BINARY – в двоичном режиме, O_TEXT – в текстовом режиме, O_TRUNC – обрезать до 0 длины);
- perms – (S_IWRITE – разрешение записи, S_IREAD – разрешение чтения).

int creat(char *path, mode_t perms) – создает новый файл. Возвращает дескриптор созданого файла или –1 в случае ошибки. Параметры:

- path – имя файла;
- perms – (S_IWRITE – разрешение записи, S_IREAD – разрешение чтения).

ssize_t write(int fd, void *buf, size_t nbytes) – записывает данные в файл или устройство. Указатель чтения-записи перемещается автоматически. Возвращает количество успешно записанных байтов или –1. Параметры:

- fd – дескриптор файла или устройства;
- buf – буфер данных;
- nbytes – количество байтов для записи.

ssize_t read(int fd, void *buf, size_t nbytes) – читает данные из файла или устройства. Указатель чтения-записи перемещается автоматически. Возвращает количество успешно прочитанных байтов или –1. Параметры:

- fd – дескриптор файла или устройства;
- buf – буфер данных;
- nbytes – количество байтов для чтения.

int close(int fd) – закрывает файл или устройство. Параметр:

- fd – дескриптор файла или устройства.

off_t lseek(int fd, off_t pos, int whence) – изменяет положение указателя чтения-записи в файле. Параметры:

- fd – дескриптор файла;
- pos – относительное смещение;
- whence - точка отсчета (SEEK_SET – от начала файла, SEEK_CUR – от текущей позиции, SEEK_END – от конца файла).

Пример файлового ввода-вывода.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main() { // Чтение даты создания BIOS
 int r, f, i;
 unsigned char c;
 f = open("/dev/mem", O_RDONLY, 0);
 lseek(f, 0xFFFF5, SEEK_SET);
 for (i=0;i<8;i++) {
 r = read(f, &c, sizeof(char));
 write(1, &c, sizeof(char)); // STDIN_FILENO=0, STDOUT_FILENO=1, STDERR_FILENO=2
 }
 close(f);
}
```

5.2. Буферизованный ввод-вывод

Этот тип файлового ввода-вывода использует концепцию потоков данных, что позволяет управлять буферизацией данных и форматировать передаваемую информацию. В частности, для доступа к консоли операционная система открывает и передает программе стандартные потоки **stdin** (для ввода с клавиатуры), **stdout** (для вывода на экран) и **stderr** (для вывода сообщений об ошибках). Возможна ассоциация потока с файлом, строкой, портом последовательного ввода-вывода и т.п.

5.2.1. Файловые операции с потоками ввода-вывода

Используются функции:

FILE *fopen(const char *filename, const char *mode) – открывает поток, ассоциируя его с файлом или устройством. Параметры:

- filename – имя файла или устройства;
- mode – строка, описывающая режим открытия (“r” – для чтения; “w” – для записи; “b” – доступ как к потоку байтов, если не указать, то будут пропускаться нули, символы конца абзаца, табуляции и пр.; “a” – запись будет выполняться в конец).

Возвращает дескриптор открытого потока или 0 в случае ошибки.

int fclose (FILE *stream) – закрывает поток. Параметр:

- stream – дескриптор потока.

Возвращает 0 в случае успеха или признак EOF в случае ошибки.

int fflush(FILE *stream) – принудительно передает данные потока, накопленные во внутреннем буфере операционной системы, на конкретное устройство ввода-вывода. Параметр:

- stream – дескриптор потока.

Возвращает 0 в случае успеха или признак EOF в случае ошибки.

size_t fread(void *buffer, size_t size, size_t count, FILE *stream) – считывает из потока данные. Параметры:

- buffer – массив под данные;
- size – размер одной порции;
- count – количество порций;
- stream – дескриптор потока.

Возвращает объем реально переданных данных.

size_t fwrite(void *buffer, size_t size, size_t count, FILE *stream) – записывает в поток данные.

- buffer – массив под данные;
- size – размер одной порции;
- count – количество порций;
- stream – дескриптор потока.

Возвращает объем реально переданных данных.

int feof(FILE *stream) – сообщает о конце потока. Параметр:

- stream – дескриптор потока.

Возвращает 0 в случае конца.

int ferror(FILE *stream) – сообщает об ошибке при операции ввода-вывода в поток. Если ошибка возникла и нужно продолжать работу, ее можно сбросить при помощи функции **clearer(FILE *stream).** Параметр:

- stream – дескриптор потока.

Возвращает 0 в случае отсутствия ошибок.

int fseek(FILE *stream, long offset, int origin) – устанавливает новую позицию чтения-записи в потоке. Параметры:

- stream – дескриптор потока;
- offset – смещение;
- origin – база, от которой отсчитывается смещение (SEEK_SET – начало потока, SEEK_CUR - текущая позиция, SEEK_END – конец потока).

Возвращает 0 в случае отсутствия ошибок.

long ftell(FILE *stream) – сообщает текущую позицию чтения-записи в потоке. Параметр:

- stream – дескриптор потока.

Возвращает искомую позицию или –1 в случае ошибки.

Пример буферизованного файлового ввода-вывода.

```
#include <stdio.h>
main() { // Побайтовое копирование файла описания пользователей
 FILE *f1, *f2;
 unsigned char c;
 f1 = fopen("/etc/passwd", "r+b"); // Readonly + binary
 f2 = fopen("./file.txt", "w");
 while (1) {
 if (feof(f1)) break;
 fread( &c, sizeof(unsigned char), 1, f1  );
 fwrite( &c, sizeof(unsigned char), 1, f2 );
 fprintf(stdout, "%c", c);  // stdin, stdout, stderr
}
fclose(f1);
fclose(f2);
}
```

5.2.2 Форматирование данных в потоках ввода-вывода

Все функции этой группы имеют частные разновидности, использующие предварительно назначенные потоки. Например, разновидностями функции **fprintf()** являются:

- **fprintf(FILE *stream, char *formats, <список параметров>)** – позволяет назначить и использовать для ввода-вывода любой поток;
- **printf(char *formats, <список параметров>)** – использует в качестве потока поток stdout, который по умолчанию назначен на окно консоли, но может быть переназначен, например, на файл или порт последовательного ввода-вывода;
- **sprintf(char *s, char *formats, <список параметров>) –** использует в качестве потока указанную строку (массив байтов);
- **cprintf(char *formats, <список параметров>) –** использует в качестве потока окно консоли без возможности переназначения;
- **vprintf(char *formats, va_list <список параметров>) –** аналогична printf(), но вместо списка параметров передается указатель на список параметров, который инициализируется при помощи va_start(va_list ptr, char *format) и удаляется при помощи va_end(va_list ptr).

fgets(char *s, int n, FILE *stream) – считывает из потока строку.
Параметры:
- s – массив байтов под считываемую строку;
- stream – дескриптор потока.

int fgetc(FILE *stream) – считывает из потока один символ.
Параметр:
- stream – дескриптор потока.

Возвращает код символа или EOF при ошибке.

int ungetc(char ch, FILE *stream) – возвращает в поток считанный символ. Теперь символ может быть считан повторно.

Параметры:

- ch – код символа;
- stream – дескриптор потока.

Возвращает ch или EOF при ошибке.

fputs(FILE *stream, char *string) – записывает в поток строку.

Параметр:

- stream – дескриптор потока.

fprintf(FILE *stream, char *formats, <список параметров>) – выводит в поток данные в соответствии со строкой формата.

Параметры:

- stream – дескриптор потока;
- formats – строка форматов («d» –знаковое целое, «u» - беззнаковое целое, «x» - целое в 16-ричном формате, «f» - вещественное, «s» - строка, «c» - символ и пр.);
- <список параметров> - параметры, значения которых выводятся в поток.

fscanf(FILE *stream, char *formats, <список параметров>) – считывает из потока данные в соответствии со строкой формата.

Параметры:

- stream – дескриптор потока;
- formats – строка форматов («d» –знаковое целое, «u» - беззнаковое целое, «x» - целое в 16-ричном формате, «f» - вещественное, «s» - строка, «c» - символ и пр.);

<список параметров> - параметры, значения которых выводятся в поток.

5.3. Каналы

Каналы – механизм межзадачного обмена с последовательной структурой доступа (очередью типа FIFO – «первый пришел, первый вышел»). Чтение и запись в них осуществляются при помощи файловых функций **read()** и **write()**. Открытые дескрипторы закрываются при помощи **close()**. При их использовании в программе необходимо подключить файл «**unistd.h**». Типы данных описаны в «**sys/types.h**» и «**sys/stat.h**».

int pipe(int pfd[2]) – создает неименованный канал с двумя открытыми дескрипторами. Возможна передача их значений родительскому процессу в виде строковых параметров вызова. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметр:

- pfd – массив дескрипторов (pfd[0] – для записи, pfd[1] – для чтения).

int mkfifo(char *path, mode_t perm) – создает именованный канал, доступ к которому затем выполняется из разных задач при помощи файловых операций open() с противоположными правами доступа, а так же read(), write() и close(). Функция open() с флагом O_RDOINLY будет заблокирована, пока не будет выполнена функция open() с O_WRONLY, и наоборот. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметры:

- path – имя создаваемого канала;
- perm – битовые флаги доступа к каналу.

Пример использования каналов.

```
#include <stdlib.h>
#include <stdio.h>
int pd[2]; // Pipe descriptorsint
main()  {
 int d1=1234, d2;
 pipe(pd); // create pipe
 write(pd[1], &d1, sizeof(int));
 read(pd[0], &d2, sizeof(int));
 printf("%u", d2);  close(pd[0]);  close(pd[1]);
}
```

```
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>

main () {
int f1, f2, d1=1234, d2, q;
q=mkfifo("/tmp/tralala", 0x777); // Create
f2 = open("/tmp/tralala", O_RDONLY|O_NONBLOCK); // !!!
f1 = open("/tmp/tralala", O_WRONLY);
q = write(f1, &d1, sizeof(int));close(f1);
q = read(f2, &d2, sizeof(int));
close(f2);
printf("%u", d2);
}
```

5.4. Очереди сообщений

Сообщение – это объект, состоящий из: уникального идентификатора (номера, типа и т.п.) и прикрепленных к нему данных (например, текста). Задачи могут посылать их друг другу, причем еще не прочитанные сообщения не пропадают, а накапливаются в очереди.

5.4.1. Очереди сообщений в стиле SYSTEM V

Для программирования очередей сообщений могут понадобиться включаемые файлы «sys/types.h», «sys/ipc.h» и «sys/msg.h». Очередь сообщений в стиле SYSTEM V является объектом, имеющим уникальный идентификатор (токен). Операционная система поддерживает несколько стандартных очередей,

кроме того, существует возможность создавать и использовать новые. Буфер сообщения должен иметь следующую структуру:

```
struct msgbuf {
   long mtype;        /* тип сообщения */
   char mtext[1];     /* текст сообщения */
};
```

Если mtype=0, то работа идет с первым в очереди сообщением любого типа; если mtype>0, то с первым по счету сообщением указанного типа; если mtype<0, то – первое сообщение с номером, который равен или меньше указанного.

- **key_t ftok(const char *path, int id)** – см. раздел «Семафоры в стиле System V»

int msgget(key_t key, int flags) – возвращает идентификатор очереди сообщений, соответствующей указанному токену. Возвращает идентификатор или –1 в случае ошибки. Параметры:

- key_t key – токен;
- int flags – битовые флаги (IPC_CREAT – создать новую очередь, 9 младших битов – биты прав доступа).

int msgsnd(int msqid, const void *msgp, size_t msgsize, int flags) – записывает собщение в очередь. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметры:

- msqid – идентификатор очереди;
- msgp – сообщение;
- msgsize – размер сообщения;
- flags – битовые флаги.

ssize_t msgrcv(int msqid, void *msgp, size_t mtextsize, long msgtype, int flags) – считывает сообщение из очереди. Возвращает число считанных байтов или –1 в случае ошибки. Параметры:

- msqid – идентификатор очереди;
- msgp – сообщение;
- msgsize – размер сообщения;
- msgtype – тип запрашиваемого сообщения;
- flags – битовые флаги.

int msgctl(int msgid, int cmd, struct msgid_ds *data) – управляет очередью сообщений. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметры:

- msgid – идентификатор очереди;

- cmd – команда (IPC_CMD – удалить очередь, IPC_STAT – заполнить data сведениями об очереди, IPC_SET – установить для очереди параметры из data);
- data – параметры очереди.

Структура блока параметров:

```
struct msgid_ds {
 struct ipc_perm msg_perm; /* Права доступа */
 msgqnum_t msg_qnum;       /* Текущее количество сообщений */
 msglen_t msg_qbytes;      /* Максимально допустимый размер */
 pid_t msg_lspid;          /* ID последнего процесса-отправителя*/
 pid_t msg_lrpid;          /* ID  последнего процесса-читателя */
 time_t msg_stime;         /* Время последней отправки */
 time_t msg_rtime;         /* Время последнего приема */
 time_t msg_ctime;         /* Время последнего изменения параметров */
}
```

Пример.

```
// System 5 Queues
#include<stdio.h>
#include<errno.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<sys/types.h>
#include<sys/stat.h>

#define TYPE 12345
struct msgbuf { long type; char msg[50]; };

Sender() {
 struct msgbuf buf;
 int q;
 key_t t;
 t = ftok(".",0x12);
 q = msgget( t, 0666 | IPC_CREAT );
 buf.type = TYPE;
 buf.msg[0] = 'H'; buf.msg[1] = 'e';
 buf.msg[2] = 'l'; buf.msg[3] = 'l';
 buf.msg[4] = 'o'; buf.msg[5] = 0;
 msgsnd( q, &buf, sizeof(struct msgbuf)-sizeof(long), 0);
}

Receiver() {
 int t, q, n;
 struct msgbuf buf;
 struct msqid_ds d;
 t = ftok(".",0x12);
 q = msgget( t, 0666 );
 n = msgrcv( q, &buf, sizeof(struct msgbuf)-sizeof(long),
             TYPE, MSG_NOERROR | IPC_NOWAIT );
 printf("\nReceived %d bytes , message=%s\n", n, buf.msg);
 n = msgctl( q, IPC_RMID, NULL);
}
```

```
main() {
 Sender();
 Receiver();
}
```

5.4.2. Очереди сообщений в стиле POSIX

Эти функции появились в UNIX-системах в связи с требованиями стандарта POSIX. Они присутствуют не во всех UNIX-системах. Для их использования в программе может понадобиться включаемый файл «mqueue.h».

mqd_t mq_open(char *name, int flags [, mode_t perms, struct mq_attr]) – открывает или создает очередь сообщений. Возвращает дескриптор очереди или –1 в случае ошибки. Параметры:

- name – имя очереди;
- flags – битовые флаги доступа (O_CREAT – создать новую, O_EXCL – при создании существующей очереди вернуть ошибку, O_RDONLY – только для чтения, O_WRONLY – только для записи, O_RDWR – для чтения-записи, O_BINARY – в двоичном режиме, O_TEXT – в текстовом режиме, O_NONBLOCK – без блокирования операций чтения при опустошении или записи при переполнении очереди);
- perms – разрешения (S_IWRITE – разрешение записи, S_IREAD – разрешение чтения);
- mq_attr – возможно NULL или в соответствии со структурой:

```
struct mq_attr {
        long mq_flags;      /* Флаги */
        long mq_maxmsg;     /* Максимальное число сообщений */
        long mq_msgsize;    /* Максимальный размер сообщений */
        long mq_msgs;       /* Количество сообщений в очереди */
}
```

int mq_close(mqd_t mqd) – закрывает очередь сообщений. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметр:

- mqd – дескриптор очереди.

int mq_unlink(const char *name) – удаляет очередь сообщений. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметр:

- name – имя удаляемой очереди.

int mq_send(mqd_t mqd, char *msg, size_t msgsize, unsigned priority) – помещает сообщение в очередь в соответствии с приоритетом. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметры:

- mqd - дескриптор очереди;

- msg - сообщение;
- msgsize - размер сообщения;
- priority - приоритет (от 0 до 31).

int mq_receive(mqd_t mqd, char *msg, size_t msgsize, unsigned *priorityp) – извлекает сообщение из очереди. Возвращает размер сообщения в случае успеха или –1 в случае ошибки. Параметры:

- mqd - дескриптор очереди;
- msg - буфер для сообщения;
- msgsize - размер буфера;
- prioritytyp - приоритет принятого сообщения или NULL.

Пример использования очередей сообщений.

```
// POSIX Queues // gcc program.c -lrt
#include <errno.h>
#include <stdio.h>
#include <mqueue.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define QSIZE 16
void Sender() {
 mqd_t m1;
 static struct mq_attr m;
 m.mq_maxmsg  = 10;
 m.mq_msgsize = QSIZE;
 m.mq_flags = 0;
 m.mq_curmsgs = 0;
 m1 = mq_open("/qwerty", O_CREAT|O_WRONLY, 0644, &m);
 mq_send(m1, "Hello!", 7, 30);
 mq_close(m1);
}
void Receiver() {
 mqd_t m2;
 static struct mq_attr m;
 char buf[QSIZE];
 int p;
 m2 = mq_open("/qwerty", O_RDONLY /*, 0644, &m*/ );
 mq_receive(m2, buf, QSIZE, &p);
 printf("\nMessage = %s", buf);
 mq_close(m2);
}
main() {
 mq_unlink("/qwerty");
 Sender();
 Receiver();
 mq_unlink("/qwerty");
}
```

5.5. Разделяемая память

Разделяемая память – фрагмент адресного пространства, к которому имеют доступ несколько процессов одновременно. Доступ к этой области

осуществляется при помощи указателей. При программировании используются включаемые файлы «sys/ipc.h» и «sys/shm.h».

5.5.1. Разделяемая память в стиле SYSTEM 5

int shmget(key_t key, size_t size, int flags) – создает общую область. Возвращает идентификатор области в случае успеха или –1 в случае ошибки. Параметры:

- key - токен объекта (см. описание функции ftok() в разделе «Очереди сообщений в стиле SYSTEM V»);
- size - размер вновь создаваемой области;
- flags – флаги (IPC_CREAT – создать новую область, IPC_PRIVATE – защищенная область).

void *shmat(int shmid, const void *shmaddr, int flags) – подключение программы к разделяемой области. Возвращает указатель на начало общей области в случае успеха или –1 в случае ошибки. Параметры:

- shmid – идентификатор общей области;
- shmaddr – желаемый адрес области или NULL;
- flags – флаги.

int shmdt (const void *shmaddr) – отключение программы от разделяемой области. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметр:

- shmaddr – указатель на область.

5.5.2. Разделяемая память в стиле POSIX

Эти функции появились в UNIX-системах в связи с требованиями стандарта POSIX. Они присутствуют не во всех UNIX-системах.

int shm_open(const char *name, int flags, mode_t perms) – открывает объект разделяемой памяти. Возвращает дескриптор области в случае успеха или –1 в случае ошибки. Параметры:

- name – имя области памяти;
- flags – битовые флаги(O_CREAT – создать новый, O_EXCL – при создании существующего объекта вернуть ошибку, O_APPEND – только для добавления, O_RDONLY – только для чтения, O_WRONLY – только для записи, O_RDWR – для чтения-записи, O_TRUNC – обрезать до 0 длины);
- perms – права доступа (S_IWRITE – разрешение записи, S_IREAD – разрешение чтения).

int shm_unlink(const char name) – удаляет ранее созданный объект разделяемой памяти. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметр:

- name – имя объекта.

int ftruncate(int fd, off_t length) - устанавливает размер объекта, заданного дескрптором. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметры:

- fd – дескритор объекта;
- length – новый размер объекта.

void *mmap(vpid *addr, size_t len, int prot, int flags, int fd, off_t off) – отображает страницы памяти, принадлежащие объекту, в адресное пространство процесса. Возвращает указатель на сегмент или MAP_FAILED в случае ошибки. Параметры:

- addr – желаемый адрес отображения (или NULL);
- len – размер сегмента;
- prot – защита сегмента (PROT_NONE – доступ запрещен, PROT_READ – для чтения, PROT_WRITE – для записи, PROT_EXEC – для исполнения);
- flags – флаги (MAP_SHARED – изменения доступны другим процессам, MAP_PRIVATE – видны только самому процессу);
- fd – дескриптор объекта;
- off – смещение в объекте разделяемой памяти.

int nunmap(vpid *addr, size_t len) – отключает отображение страниц памяти объекта в память процесса. Возвращает 0 в случае успеха или –1 в случае ошибки. Параметры:

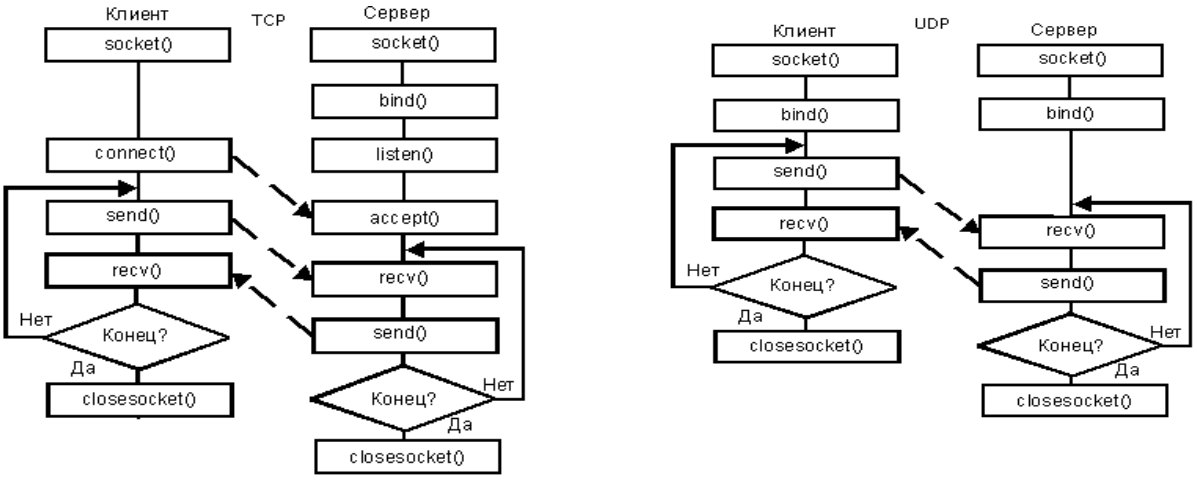
- addr – указатель на сегмент;
- len – длина сегмента.

Пример использования разделяемой памяти.

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
main() {
 int r, f, i;
 char *m;
 f = open("/dev/mem", O_RDONLY, 0);
 m = (char *) mmap( 0, 0xFFFFF, PROT_READ, MAP_SHARED, f, 0);
 for (i=0xFFFF5;i<0xFFFFD;i++)  write(1, &m[i], 1);
 munmap( m, 0xFFFFF);
 close(f);
}
```

6. Сокеты

Сокеты Беркли – это универсальный интерфейс для обмена данными, использующий сетевые протоколы. Один из участников информационного обмена обязательно должен быть сервером, обслуживающим запросы, а остальные – клиентами. Общая структура алгоритма взаимодействия проиллюстрирована на рис. 1.



а) По протоколу TCP

Б) По протоколу UDP

Рис.1. Варианты взаимодействия сервера и клиента

int socket (int domain, int type, int protocol) – создает новый (неинициализированный) сокет. Параметры:

- domain – тип используемого адреса (PF_UNIX – локальный адрес, PF_INET или AF_INET – адрес в формате IPv4, PF_INET6 или AF_INET6 – в формате IPv6), при этом флаги с «AF» требуют указания конкретного протокола, а флаги с «PF» позволяют системе выбрать протокол самостоятельно;
- type – тип данных (SOCK_STREAM – поток для TCP, SOCK_DGRAM – дейтаграмма для UDP, SOCK_RAW – формируемый вручную, SOCK_SEQPACKET – последовательные пакеты);
- protocol – тип протокола (IPPROTO_TCP, IPPROTO_SCTP, IPPROTO_UDP, IPPROTO_DCCP или 0 – по умолчанию).

Возвращает дескриптор сокета или -1 – в случае неудачи.

int close (int sock) - закрывает сокет, освобождая все ресурсы. Параметр:

- sock – дескриптор открытого ранее сокета.

Возвращает 0 в случае успеха или –1 в случае неудачи.

int shutdown(int soc, int how) – запретить работу сокета. Параметры:

- sock – дескриптор открытого ранее сокета;
- how – способ запрета (0 - запретить чтение, 1 - запретить запись, 2 - запретить и то и другое).

int connect(int sock, struct socketaddr *serv_addr, int adrlen) - пытается установить соединение с сокетом иного сетевого узла. Параметры:

- sock – дескриптор открытого ранее сокета;
- serv_addr – идентификатор узла, с которым устанавливается соединение, заданный в виде:

```
struct sockaddr {
 unsigned short sa_family; // Тип используемого адреса, см. socket()
 char sa_data[14];          //  Адрес и связанные с ним служебные данные
}
```

Формат поля sa_data зависит от типа адреса. Например, для случая IP-адресации используется следующая структура:

```
struct sockaddr_in {
 short int sin_family; // Тип используемого адреса, см. socket()
 unsigned short int sin_port; // Порт
 unsigned long s_addr; // конкретный IP-адрес или INADDR_ANY – «127.0.0.*»
 unsigned char siz_zer[8]; // Нули
}
```

где

```
 struct in_addr { u_long s_addr; };
```

- adrlen – длина идентификатора.

Возвращает 0 в случае удачи, иначе -1.

int bind (int sock, struct sockaddr *serv_addr, int adrlen) - устанавливает соответствие между сокетом и адресом текущего узла. Параметры – см. описание функции connect(). Возвращает 0 в случае удачи, иначе -1.

int listen(int sock, int backlog) - ожидает запрос соединения от иного сокета. Параметры:

- sock – дескриптор ранее открытого сокета;
- backlog – максимально допустимый размер очереди сообщений.

Возвращает 0 в случае успеха, иначе -1.

int accept (int sock, struct sockaddr *addr, int *adrlen) - устанавливает соединение с другим «слушающим» сокетом, указанным в параметре sock. Параметры - см. описание функции connect(). Внимание: последний параметр – не целая переменная, а ссылка на нее!

Возвращает дескриптор сокета в случае успеха, иначе -1.

int send (int sock, void *buf, int bufsize, int flags) - посылает данные через сокет, который (или с которым) уже установлено соединение. Параметры:

- sock – дескриптор открытого ранее сокета;

- buf - буфер данных;
- bufsize – количество передаваемых данных;
- flags – битовые флаги (MSG_OOB – «срочные» данные, MSG_DONTROUTE – передавать без маршрутизации).

Возвращает количество передаваемых (не переданных!) байтов или –1 в случае неуспеха.

int recv (int sock, void *buf, int bufsize, int flags) - считывает данные из сокета. Параметры:

- sock – хэндл открытого ранее сокета;
- buf – буфер данных;
- bufsize – размер буфера;
- flags – битовые флаги (MSG_OOB – «срочные» данные, MSG_PEEK – прочитать данные, не удаляя их из входной очереди).

Возвращает количество прочитанных байтов (возможно, 0) или -1 в случае ошибки.

struct hostent *gethostbyname(const char *name) - возвращает информацию о сетевом узле, заданном при помощи символического имени. Параметр:

- name – строка символического имени (например, «pop3.hotmail.com»).

Возвращает 0 в случае ошибки или заполненную структуру

```
struct hostent {
   char  *  h_name;        // «Официальное» имя узла
   char  * * h_aliases;    // Массив альтернативных имен
   int  h_addrtype;        // Тип адреса
   int   h_length;         // Длина каждого адреса
   char  ** h_addr_list;   //  Массив числовых адресов
};
```

struct hostent *gethostbyaddr (char *addr, int len, int type) - возвращает информацию о сетевом узле, заданном при помощи числового адреса. Параметры:

- addr – указатель на числовой адрес;
- len – длина адреса;
- type – тип адреса.

Возвращает 0 в случае ошибки или заполненную структуру hostent (см. описание функции gethostbyname()).

int gethostname (char * name, int namelen) - возвращает имя текущего узла сети. Параметры:

- name – строка с именем;
- namelen – длина имени.

В случае успеха возвращает 0.

Группа функций **ntohs()**, **htons()**, **ntohl()** и **htonl()** - преобразуют порядок байтов в числовых данных к виду, используемому в сетях, и обратно:

- **uint32_t ntohs (uint32_t netshort)** – 16-битовое число из сетевого в «нормальный»;
- **uint32_t htons (uint32_t hostshort**) – 16-битовое число из «нормального» в сетевой;
- **uint32_t ntohl (uint32_t netlong)** – 32-битовое число из сетевого в «нормальный»;
- **uint32_t htonl (uint32_t hostlong)** - 32-битовое число из «нормального» в сетевой.

unsigned long inet_addr (const char *cp) - преобразует строку с IP-адресом вида «AAA.BBB.CCC.DDD» в числовую форму. Параметр:

- cp – строка с адресом.

Возвращает числовое значение адреса или –1 в случае ошибки. Если нужен адрес «255.255.255.255», который совпадает с -1, то использовать **inet_aton()**.

char *inet_ntoa (struct in_addr in) - преобразует IP-адрес, представленный в числовой форме, в строку вида «AAA.BBB.CCC.DDD». Параметр:

- in – числовой IP-адрес.

Пример применения «локальных» сокетов.

```
/*Сервер должен быть запущен в 1-ой консоли  */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <errno.h>
#define PORT 6969
#define BUFSIZE 16
int sock;
struct sockaddr_in addr;
char buf[BUFSIZE];
int bytes_read, total = 0;
int main() {
 sock = socket(AF_INET, SOCK_DGRAM, 0);
 addr.sin_family = AF_INET;
 addr.sin_port = htons(PORT);
 addr.sin_addr.s_addr = htonl(INADDR_ANY);
 bind(sock, (struct sockaddr *)&addr, sizeof(addr));
while(1) {
  bytes_read = recvfrom(sock, buf, BUFSIZE, 0, NULL, NULL);
  total = total + bytes_read;
  buf[bytes_read] = '\0';
  fprintf(stdout, buf);fflush(stdout);
  if (total>BUFSIZE) break;
 }
  printf("\nTotal %d bytes received", total);
  return 0;
}

/* Клиент должен быть запущен во 2-ой консоли */
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#define PORT 6969
char ss[]={"Hello world"};
int sock;
struct sockaddr_in addr;
int main() {
 sock = socket(AF_INET, SOCK_DGRAM, 0);
 addr.sin_family = AF_INET;
 addr.sin_port = htons(PORT);
 addr.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
 connect(sock, (struct sockaddr *)&addr, sizeof(addr));
 send(sock, ss, strlen(ss), 0);
 close(sock);
 return 0;
}
```

Литература

1. Гриффитс А. GCC. Настольная книга пользователей, программистов и системных администраторов. – К.:ООО «ТИД ДС», 2004. – 624 с.

2. Джонсон М., Троан Э. Разработка приложений в среде Linux. – М.: ООО «И.Д.Вильямс», 2007. – 544 с.

3. Лав Р. Linux. Системное программирование. – СПб.: Питер, 2014. – 448 с.

4. Митчелл М., Оулдем Д., Самьюэд А. Программирование для Linux. Профессиональный подход. – 288 с.

5. Робачевский А.М. Операционная система UNIX. – СПб.: БХВ-Петербург, 2002. – 528 с.

6. Роббинс А. Linux: программирование в примерах. – М.: КУДИЦ-ОБРАЗ, 2005. – 656 с.

7. Стивенс У. UNIX. Взаимодействие процессов. – СПб.: Питер, 2003. - 576 с.

8. Фуско Дж. Linux. Руководство программиста. – СПб.: Питер, 2011. – 448 с.

9. Рочкинд М. Программирование для UNIX. – СПб.: БХВ-Петербург, 2005. – 704 с.

Приложение А. Коды ошибок

В таблице приведены числовые значения ошибок только для Linux. В иных UNIX-подобных операционных системах они могут отличаться.

Ошибка	Номер	Причина
EPERM	1	Операция не разрешена
ENOENT	2	Нет такого файла или каталога
ESRCH	3	Нет такого процесса
EINTR	4	Прерванный системный вызов
EIO	5	Ошибка ввода/вывода
ENXIO	6	Устройство не подключено
E2BIG	7	Слишком длинный список аргументов
ENOEXEC	8	Неверный формат выполняемого файла
EBADF	9	Неверный дескриптор файла
ECHILD	10	Нет порожденных процессов
EDEADLK	11	Предотвращена взаимная блокировка
ENOMEM	12	Невозможно выделить блок памяти
EACCES	13	Доступ запрещен
EFAULT	14	Неверный адрес
ENOTBLK	15	Требуется блочное устройство
EBUSY	16	Устройство занято
EEXIST	17	Файл существует
EXDEV	18	Ссылка между устройствами
ENODEV	19	Операция не поддерживается устройством
ENOTDIR	20	Это не каталог
EISDIR	21	Это каталог
EINVAL	22	Неверный аргумент
ENFILE	23	Слишком много открытых файлов в системе
EMFILE	24	Слишком много открытых файлов
ENOTTY	25	Вызов ioctl не соответствует устройству
ETXTBSY	26	Текстовый файл занят
EFBIG	27	Файл слишком большой
ENOSPC	28	Нет места на устройстве
ESPIPE	29	Недопустимое позиционирование
EROFS	30	Файловая система только для чтения
EMLINK	31	Слишком много ссылок
EPIPE	32	Разорваный конвейер
EDOM	33	Цифровой аргумент вне области определения
ERANGE	34	Результат слишком велик
EAGAIN, WOULDBLOCK	35	Ресурс временно недоступен
EINPROGRESS	36	Операция в процессе выполнения
EALREADY	37	Операция уже выполняется
ENOTSOCK	38	Канальная операция не на канале
EDESTADDRREQ	39	Требуется адрес приемника
EMSGSIZE	40	Сообщение слишком длинное
EPROTOTYPE	41	Неверный протокол для канала
ENOPROTOOPT	42	Протокол недоступен
EPROTONOSUPPORT	43	Протокол не поддерживается
ESOCKTNOSUPPORT	44	Тип канала не поддерживается
EOPNOTSUPP	45	Операция не поддерживается
EPFNOSUPPORT	46	Семейство протоколов не поддерживается

EAFNOSUPPORT	47	Адреса не поддерживаются семейством протоколов
EADDRINUSE	48	Адрес уже используется
EADDRNOTAVAIL	49	Невозможно назначить запрошенный адрес
ENETDOWN	50	Сеть не работает
ENETUNREACH	51	Сеть недостижима
ENETRESET	52	Сеть разорвала соединение по сбросу
ECONNABORTED	53	Программа вызвала разрыв соединения
ECONNRESET	54	Соединение сброшено другой стороной
ENOBUFS	55	Нет свободных буферов
EISCONN	56	Канал уже подключен
ENOTCONN	57	Канал не подключен
ESHUTDOWN	58	Посылка невозможна после отключения канала
ETOOMANYREFS	59	Слишком много ссылок: невозможно сплести
ETIMEDOUT	60	Время для соединения вышло
ECONNREFUSED	61	Соединение отвергнуто
ELOOP	62	Слишком много уровней ссылок
ENAMETOOLONG	63	Слишком длинное имя файла
EHOSTDOWN	64	Компьютер не работает
EHOSTUNREACH	65	Нет пути до компьютера
ENOTEMPTY	66	Каталог не пуст
EPROCLIM	67	Слишком много процессов
EUSERS	68	Слишком много пользователей
EDQUOT	69	Превышено ограничение на использование диска
ESTALE	70	Устаревший дескриптор NFS
EREMOTE	71	Слишком много не-локальных уровней в пути
EBADRPC	72	Неверная структура RPC
ERPCMISMATCH	73	Неверная версия RPC
EPROGUNAVAIL	74	RPC программа недоступна
EPROGMISMATCH	75	Неверная версия программы
EPROCUNAVAIL	76	Неверная процедура для программы
ENOLCK	77	Невозможно выполнить захват
ENOSYS	78	Функция не действует
EFTYPE	79	Неверный тип или формат файла
EAUTH	80	Ошибка аутентификации
ENEEDAUTH	81	Требуется аутентификация
EIPSEC	82	Ошибка обработки IPSec
ENOATTR	83	Атрибут не найден
EILSEQ	84	Неверная последовательность байтов
ENOMEDIUM	85	Носитель не найден
EMEDIUMTYPE	86	Неверный тип носителя
EOVERFLOW	87	Значение слишком велико для этого типа данных
ECANCELED	88	Операция отменена
EIDRM	89	Идентификатор удален
ENOMSG	90	Нет сообщения запрошенного типа
ENOTSUP	91	Не поддерживается

Приложение Б. Сигналы

В таблице приведены числовые значения сигналов только для Linux. В иных UNIX-подобных операционных системах они могут отличаться.

Сигнал	Номер	Причина
SIGABRT	6	Сигнал аварийного завершения процесса
SIGALRM	14	Срабатывание будильника
SIGCHLD	20	Завершение, остановка или продолжение дочернего процесса
SIGCONT	18	Продолжение приостановленного процесса
SIGFPE	8	Некорректная арифметическая операция
SIGILL	4	Некорректная команда
SIGINT	2	Сигнал прерывания, поступивший с терминала
SIGKILL	9	Уничтожение процесса (нельзя игнорировать)
SIGPIPE	13	Попытка записи в канал, из которого никто не читает
SIGQUIT	3	Сигнал выхода, поступивший с терминала
SIGSEGV	11	Некорректное обращение к памяти
SIGSTOP	19	Остановка выполнения (нельзя игнорировать)
SIGTERM	15	Сигнал терминирования
SIGTSTP	20	Сигнал остановки, поступивший с терминала
SIGTTIN	21	Попытка чтения из фонового процесса
SIGTTOU	22	Попытка записи в фоновый процесс
SIGUSR1,SIGUSR2	10,12	Определяемые пользователем сигналы
SIGPOLL	29	Опрашиваемое событие
SIGPROF	27	Срабатывание таймера профилирования
SIGSYS	21	Некорректный системный вызов
SIGTRAP	5	Попадание в точку трассировки/прерывания
SIGVTALRM	26	Срабатывание виртуального таймера
SIGXCPU	24	Исчерпан лимит процессорного времени
SIGXFSZ	25	Превышено ограничение на размер файлов

Методические материалы

ПРОГРАММИРОВАНИЕ МНОГОЗАДАЧНОСТИ В UNIX

Методические указания

Составитель ***Климентьев Константин Евгеньевич***

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ имени академика С.П.Королева»
(Самарский университет)
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34

Изд-во Самарского университета
443086, Самара, Московское шоссе, 34.