A Прототипы функций

Это приложение содержит прототипы функций, определяемых стандартами ISO C и POSIX, а также функций UNIX, описанных в этой книге. Часто необходимо узнать, какие аргументы принимает та или иная функция («В каком аргументе функции `fgets` передается указатель на структуру `FILE`?») или что она возвращает («Что возвращает функция `sprintf` — указатель или счетчик?»). В описаниях прототипов указаны заголовочные файлы, которые нужно подключить для получения определений всех специальных констант и прототипов функций ISO C, что поможет в диагностике ошибок времени компиляции.

Для каждой функции справа от первого заголовочного файла указывается номер страницы в книге, где приводился прототип этой функции. Там же следует искать дополнительную информацию о ней.

Некоторые функции поддерживаются не всеми платформами, описанными в этой книге. Кроме того, некоторые платформы поддерживают флаги функций, не поддерживаемые другими платформами. Обычно мы будем перечислять платформы, которые поддерживают ту или иную функциональность. Однако в отдельных случаях будут перечислены платформы, на которых поддержка отсутствует.

```
void      abort(void);
```
`<stdlib.h>` с. 434

Эта функция никогда не возвращает управление

```
int       accept(int sockfd, struct sockaddr *restrict addr,
              socklen_t *restrict len);
```
`<sys/socket.h>` с. 697

Возвращает дескриптор файла (сокета) в случае успеха, –1 — в случае ошибки

```
int       access(const char *path, int mode);
```
`<unistd.h>` с. 147

mode: `R_OK`, `W_OK`, `X_OK`, `F_OK`

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       aio_cancel(int fd, struct aiocb *aiocb);
```
с. 594

<aio.h>

Возвращает AIO_ALLDONE, AIO_CANCELED или AIO_NOTCANCELED в случае успеха, –1 — в случае ошибки

```
int       aio_error(const struct aiocb *aiocb);
```
с. 598

<aio.h>

Возвращает 0 в случае успеха, EINPROGRESS — если выполнение операции продолжается, код ошибки — если операция потерпела неудачу, –1 — в случае ошибки

```
int       aio_fsync(int op, struct aiocb *aiocb);
```
с. 592

<aio.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       aio_read(struct aiocb *aiocb);
```
с. 592

<aio.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
ssize_t   aio_return(const struct aiocb *aiocb);
```
с. 593

<aio.h>

Возвращает результат асинхронной операции в случае успеха, –1 — в случае ошибки

```
int       aio_suspend(const struct aiocb *const list[], int nent,
                      const struct timespec *timeout);
```
с. 593

<aio.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       aio_write(struct aiocb *aiocb);
```
с. 592

<aio.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
unsigned
int       alarm(unsigned int seconds);
```
с. 406

<unistd.h>

Возвращает 0 или число секунд, оставшихся до истечения установленного интервала времени

```
int       atexit(void (*func)(void));
```
с. 255

<stdlib.h>

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

```
int       bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```
<sys/socket.h> с. 692
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
void      *calloc(size_t nobj, size_t size);
```
<stdlib.h> с. 262
Возвращает непустой указатель в случае успеха, NULL — в случае ошибки

```
speed_t   cfgetispeed(const struct termios *termptr);
```
<termios.h> с. 785
Возвращает значение скорости в бодах

```
speed_t   cfgetospeed(const struct termios *termptr);
```
<termios.h> с. 785
Возвращает значение скорости в бодах

```
int       cfsetispeed(struct termios *termptr, speed_t speed);
```
<termios.h> с. 785
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       cfsetospeed(struct termios *termptr, speed_t speed);
```
<termios.h> с. 785
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       chdir(const char *path);
```
<unistd.h> с. 184
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       chmod(const char *path, mode_t mode);
```
<sys/stat.h> с. 151
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       chown(const char *path, uid_t owner, gid_t group);
```
<unistd.h> с. 155
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
void      clearerr(FILE *fp);
```
<stdio.h> с. 201

```
int     clock_getres(clockid_t clock_id, struct timespec *tsp);
```
<sys/time.h> с. 244

clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     clock_gettime(clockid_t clock_id, struct timespec *tsp);
```
<sys/time.h> с. 243

clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     clock_nanosleep(clockid_t clock_id, int flags,
            const struct timespec *reqtp,
            struct timespec *remtp);
```
<time.h> с. 445

clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

flags: TIMER_ABSTIME

Возвращает 0, если установленное время истекло, или код ошибки — в случае неудачи

```
int     clock_settime(clockid_t clock_id, const struct timespec *tsp);
```
<sys/time.h> с. 244

clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     close(int fd);
```
<unistd.h> с. 109

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     closedir(DIR *dp);
```
<dirent.h> с. 179

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
void    closelog(void);
```
<syslog.h> с. 545

```
unsigned
char      *CMSG_DATA(struct cmsghdr *cp);
```
<sys/socket.h> с. 735

Возвращает указатель на данные, связанные со структурой cmsghdr

```
struct
cmsghdr   *CMSG_FIRSTHDR(struct msghdr *mp);
```
<sys/socket.h> с. 735

Возвращает указатель на первую структуру cmsghdr, связанную со структурой msghdr, или NULL — если таковой не существует

```
unsigned
int       CMSG_LEN(unsigned int nbytes);
```
<sys/socket.h> с. 735

Возвращает объем памяти, который необходимо выделить для хранения объекта размером nbytes

```
struct
cmsghdr   *CMSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp);
```
<sys/socket.h> с. 735

Возвращает указатель на следующую структуру cmsghdr, связанную со структурой msghdr, которую представляет текущая структура cmsghdr, или NULL — если таковой не существует

```
int       connect(int sockfd, const struct sockaddr *addr, socklen_t len);
```
<sys/socket.h> с. 694

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       creat(const char *path, mode_t mode);
```
<fcntl.h> с. 108

mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)

Возвращает дескриптор файла, открытый только для записи, в случае успеха, –1 — в случае ошибки

```
char      *ctermid(char *ptr);
```
<stdio.h> с. 787

Возвращает указатель на имя управляющего терминала в случае успеха, указатель на пустую строку — в случае ошибки

```
int     dprintf(int fd, const char *restrict format, ...);
                        <stdio.h>                                   с. 210
```
Возвращает число выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int     dup(int fd);
                        <unistd.h>                                  с. 123
```
Возвращает новый дескриптор файла в случае успеха, –1 — в случае ошибки

```
int     dup2(int fd, int fd2);
                        <unistd.h>                                  с. 123
```
Возвращает новый дескриптор файла в случае успеха, –1 — в случае ошибки

```
void    endgrent(void);
                        <grp.h>                                     с. 236
```

```
void    endhostent(void);
                        <netdb.h>                                   с. 685
```

```
void    endnetent(void);
                        <netdb.h>                                   с. 686
```

```
void    endprotoent(void);
                        <netdb.h>                                   с. 687
```

```
void    endpwent(void);
                        <pwd.h>                                     с. 232
```

```
void    endservent(void);
                        <netdb.h>                                   с. 687
```

```
void    endspent(void);
                        <shadow.h>                                  с. 235
```
Платформы: Linux 3.2.0, Solaris 10

```
int     execl(const char *path, const char *arg0, ... /* (char *) 0 */ );
                        <unistd.h>                                  с. 309
```
Возвращает –1 в случае ошибки, в случае успеха не возвращает управление

```
int     execle(const char *path, const char *arg0, ...
         /* (char *) 0, char *const envp[] */ );
```
<unistd.h> с. 309

Возвращает –1 в случае ошибки, в случае успеха не возвращает управление

```
int     execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );
```
<unistd.h> с. 309

Возвращает –1 в случае ошибки, в случае успеха не возвращает управление

```
int     execv(const char *path, char *const argv[]);
```
<unistd.h> с. 309

Возвращает –1 в случае ошибки, в случае успеха не возвращает управление

```
int     execve(const char *path, char *const argv[], char *const envp[]);
```
<unistd.h> с. 309

Возвращает –1 в случае ошибки, в случае успеха не возвращает управление

```
int     execvp(const char *filename, char *const argv[]);
```
<unistd.h> с. 309

Возвращает –1 в случае ошибки, в случае успеха не возвращает управление

```
void    _Exit(int status);
```
<stdlib.h> с. 253

Эта функция никогда не возвращает управление

```
void    _exit(int status);
```
<unistd.h> с. 253

Эта функция никогда не возвращает управление

```
void    exit(int status);
```
<stdlib.h> с. 253

Эта функция никогда не возвращает управление

```
int     faccessat(int fd, const char *path, int mode, int flag);
```
<unistd.h> с. 147

mode: `R_OK`, `W_OK`, `X_OK`, `F_OK`

flag: `AT_EACCESS`

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fchdir(int fd);
```
<unistd.h> с. 184
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fchmod(int fd, mode_t mode);
```
<sys/stat.h> с. 151
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fchmodat(int fd, const char *path, mode_t mode, int flag);
```
<sys/stat.h> с. 151
mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)
flag: AT_SYMLINK_NOFOLLOW
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fchown(int fd, uid_t owner, gid_t group);
```
<unistd.h> с. 155
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fchownat(int fd, const char *path, uid_t owner,
             gid_t group, int flag);
```
<unistd.h> с. 155
flag: AT_SYMLINK_NOFOLLOW
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fclose(FILE *fp);
```
<stdio.h> с. 200
Возвращает 0 в случае успеха, EOF — в случае ошибки

```
int     fcntl(int fd, int cmd, ... /* int arg */ );
```
<fcntl.h> с. 126
cmd: F_DUPFD, F_DUPFD_CLOEXEC, F_GETFD,
F_SETFD, F_GETFL, F_SETFL, F_GETOWN,
F_SETOWN, F_GETLK, F_SETLK, F_SETLKW
Возвращаемое значение зависит от аргумента *cmd* в случае успеха, –1 — в случае ошибки

```
int     fdatasync(int fd);
```
<unistd.h> с. 125
Возвращает 0 в случае успеха, –1 — в случае ошибки
Платформы: Linux 3.2.0, Solaris 10

```
void      FD_CLR(int fd, fd_set *fdset);
```
<sys/select.h> с. 582

```
int       FD_ISSET(int fd, fd_set *fdset);
```
<sys/select.h> с. 582

Возвращает ненулевое значение, если *fd* имеется в наборе, 0 — в противном случае

```
FILE      *fdopen(int fd, const char *type);
```
<stdio.h> с. 197

type: "r", "w", "a", "r+", "w+", "a+"

Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки

```
DIR       *fdopendir(int fd);
```
<dirent.h> с. 179

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
void      FD_SET(int fd, fd_set *fdset);
```
<sys/select.h> с. 582

```
void      FD_ZERO(fd_set *fdset);
```
<sys/select.h> с. 582

```
int       feof(FILE *fp);
```
<stdio.h> с. 201

Возвращает ненулевое значение (истина), если достигнут конец файла, 0 (ложь) — в противном случае

int **ferror**(FILE **fp*);

<stdio.h> с. 201

Возвращает ненулевое значение (истина), если при работе с потоком возникла ошибка, 0 (ложь) — в противном случае

```
int       fexecve(int fd, char *const argv[], char *const envp[]);
```
<unistd.h> с. 309

Возвращает –1 в случае ошибки, в случае успеха не возвращает управление

```
int fflush(FILE *fp);
```
<stdio.h> с. 197

Возвращает 0 в случае успеха, EOF — в случае ошибки

```
int fgetc(FILE *fp);
```
<stdio.h> с. 201

Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки

```
int fgetpos(FILE *restrict fp, fpos_t *restrict pos);
```
<stdio.h> с. 210

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

```
char *fgets(char *restrict buf, int n, FILE *restrict fp);
```
<stdio.h> с. 203

Возвращает указатель на *buf* в случае успеха, NULL — по достижении конца файла или в случае ошибки

```
int       fileno(FILE *fp);
```
<stdio.h> с. 216

Возвращает дескриптор файла, ассоциированный с потоком в случае успеха, –1 — в случае ошибки

```
void      flockfile(FILE *fp);
```
<stdio.h> с. 515

```
FILE      *fmemopen(void *restrict buf, size_t size,
                    const char *restrict type);
```
<stdio.h> с. 224

type: "r", "w", "a", "r+", "w+", "a+"

Возвращает указатель на поток в случае успеха, NULL — в случае ошибки

```
FILE      *fopen(const char *restrict path, const char *restrict type);
```
<stdio.h> с. 197

type: "r", "w", "a", "r+", "w+", "a+"

Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки

```
pid_t     fork(void);
```
<unistd.h> с. 286

Возвращает 0 в дочернем процессе, идентификатор дочернего процесса — в родительском процессе, –1 — в случае ошибки

```
long      fpathconf(int fd, int name);
```
<unistd.h> с. 82

name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED, _PC_FILESIZEBITS, _PC_LINK_MAX, _PC_MAX_CANON, _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC, _PC_PATH_MAX, _PC_PIPE_BUF, _PC_PRIO_IO, _PC_SYMLINK_MAX, _PC_SYNC_IO, _PC_TIMESTAMP_RESOLUTION, _PC_2_SYMLINKS, _PC_VDISABLE

Возвращает соответствующее значение в случае успеха, –1 — в случае ошибки

```
int       fprintf(FILE *restrict fp, const char *restrict format, ...);
```
<stdio.h> с. 210

Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int       fputc(int c, FILE *fp);
```
<stdio.h> с. 202

Возвращает символ *c* в случае успеха, EOF — в случае ошибки

```
int       fputs(const char *restrict str, FILE *restrict fp);
```
<stdio.h> с. 204

Возвращает неотрицательное значение в случае успеха, EOF — в случае ошибки

```
size_t    fread(void *restrict ptr, size_t size, size_t nobj, FILE *restrict fp);
```
<stdio.h> с. 207

Возвращает количество прочитанных блоков

```
void      free(void *ptr);
```
<stdlib.h> с. 262

```
void      freeaddrinfo(struct addrinfo *ai);
```
<sys/socket.h> с. 688
<netdb.h>

```
FILE    *freopen(const char *restrict path, const char *restrict type,
                 FILE *restrict fp);
```
с. 197

<stdio.h>

type: "r", "w", "a", "r+", "w+", "a+"

Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки

```
int     fscanf(FILE *restrict fp, const char *restrict format, ...);
```
с. 214

<stdio.h>

Возвращает количество введенных элементов, EOF — по достижении конца файла или в случае ошибки перед выполнением преобразования

```
int     fseek(FILE *fp, long offset, int whence);
```
с. 209

<stdio.h>

whence: SEEK_SET, SEEK_CUR, SEEK_END

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fseeko(FILE *fp, off_t offset, int whence);
```
с. 209

<stdio.h>

whence: SEEK_SET, SEEK_CUR, SEEK_END

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fsetpos(FILE *fp, const fpos_t *pos);
```
с. 210

<stdio.h>

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

```
int     fstat(int fd, struct stat *buf);
```
с. 137

<sys/stat.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fstatat(int fd, const char *restrict path,
                struct stat *restrict buf, int flag);
```
с. 137

<sys/stat.h>

flag: AT_SYMLINK_NOFOLLOW

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     fsync(int fd);
```
с. 125

<unistd.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
long      ftell(FILE *fp);
```
<stdio.h> с. 209

Возвращает значение текущей позиции в файле в случае успеха, –1L — в случае ошибки

```
off_t     ftello(FILE *fp);
```
<stdio.h> с. 209

Возвращает значение текущей позиции в файле в случае успеха, (off_t)-1 — в случае ошибки

```
key_t     ftok(const char *path, int id);
```
<sys/ipc.h> с. 639

Возвращает значение ключа в случае успеха, (key_t)-1 — в случае ошибки

```
int       ftruncate(int fd, off_t length);
```
<unistd.h> с. 158

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       ftrylockfile(FILE *fp);
```
<stdio.h> с. 515

Возвращает 0 в случае успеха, ненулевое значение — если блокировка не может быть установлена

```
void      funlockfile(FILE *fp);
```
<stdio.h> с. 515

```
int       futimens(int fd, const struct timespec times[2]);
```
<sys/stat.h> с. 174

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       fwide(FILE *fp, int mode);
```
<stdio.h> с. 193

<wchar.h>

Возвращает положительное значение, если поток ориентирован на работу с многобайтными (wide) символами, отрицательное — с однобайтными символами и 0 — если поток не имеет ориентации

```
size_t    fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
```
<stdio.h> с. 207

Возвращает количество записанных блоков

```
const
char      *gai_strerror(int error);
```
<netdb.h> с. 689

Возвращает указатель на строку с описанием ошибки

```
int       getaddrinfo(const char *restrict host,
                      const char *restrict service,
                      const struct addrinfo *restrict hint,
                      struct addrinfo **restrict res);
```
<sys/socket.h> с. 688

<netdb.h>

Возвращает 0 в случае успеха, ненулевой код ошибки — в случае неудачи

```
int       getc(FILE *fp);
```
<stdio.h> с. 201

Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки

```
int       getchar(void);
```
<stdio.h> с. 201

Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки

```
int       getchar_unlocked(void);
```
<stdio.h> с. 516

Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки

```
int       getc_unlocked(FILE *fp);
```
<stdio.h> с. 516

Возвращает следующий символ в случае успеха, EOF — по достижении конца файла или в случае ошибки

```
char      *getcwd(char *buf, size_t size);
```
<unistd.h> с. 185

Возвращает указатель на *buf* в случае успеха, NULL — в случае ошибки

```
gid_t     getegid(void);
```
<unistd.h> с. 285

Возвращает эффективный идентификатор группы вызывающего процесса

```
char      *getenv(const char *name);
```
<stdlib.h> с. 266

Возвращает указатель на значение переменной окружения с именем *name*, NULL — если переменная не найдена

```
uid_t     geteuid(void);
```
<unistd.h> с. 285

Возвращает эффективный идентификатор пользователя вызывающего процесса

```
gid_t     getgid(void);
```
<unistd.h> с. 285

Возвращает реальный идентификатор группы вызывающего процесса

```
struct
group     *getgrent(void);
```
<grp.h> с. 236

Возвращает указатель в случае успеха, NULL — по достижении конца файла или в случае ошибки

```
struct
group     *getgrgid(gid_t gid);
```
<grp.h> с. 235

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
struct
group     *getgrnam(const char *name);
```
<grp.h> с. 235

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
int       getgroups(int gidsetsize, gid_t grouplist[]);
```
<unistd.h> с. 237

Возвращает количество идентификаторов дополнительных групп в случае успеха, –1 — в случае ошибки

```
struct
hostent   *gethostent(void);
```
<netdb.h> с. 685

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
int       gethostname(char *name, int namelen);
```
<unistd.h> с. 242

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
char      *getlogin(void);
```
<unistd.h> с. 337

Возвращает указатель на строку с именем пользователя в случае успеха, NULL — в случае ошибки

```
int       getnameinfo(const struct sockaddr *restrict addr,
                      socklen_t alen, char *restrict host,
                      socklen_t hostlen, char *restrict service,
                      socklen_t servlen, unsigned int flags);
```
<sys/socket.h> с. 689

<netdb.h>

flags: NI_DGRAM, NI_NAMEREQD, NI_NOFQDN, NI_NUMERICHOST, NI_NUMERICSCOPE, NI_NUMERICSERV

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

```
struct
netent    *getnetbyaddr(uint32_t net, int type);
```
<netdb.h> с. 686

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
struct
netent    *getnetbyname(const char *name);
```
<netdb.h> с. 686

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
struct
netent    *getnetent(void);
```
<netdb.h> с. 686

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
int       getopt(int argc, const * const argv[], const char *options);
```
<fcntl.h> с. 752

```
extern int optind, opterr, optopt;
extern char *optarg;
```

Возвращает символ следующей опции или –1 — если все параметры обработаны

```
int       getpeername(int sockfd, struct sockaddr *restrict addr,
                      socklen_t *restrict alenp);
```
<sys/socket.h> с. 693

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
pid_t     getpgid(pid_t pid);
```
<unistd.h> с. 356

Возвращает идентификатор группы процессов в случае успеха, –1 — в случае ошибки

```
pid_t     getpgrp(void);
```
<unistd.h> с. 356

Возвращает идентификатор группы процессов вызывающего процесса

```
pid_t     getpid(void);
```
<unistd.h> с. 285

Возвращает идентификатор процесса вызывающего процесса

```
pid_t     getppid(void);
```
<unistd.h> с. 285

Возвращает идентификатор родительского процесса

```
int       getpriority(int which, id_t who);
```
<sys/resource.h> с. 339

which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER

Возвращает значение коэффициента уступчивости между –NZERO и NZERO–1 в случае успеха, –1 — в случае ошибки

```
struct
protoent  *getprotobyname(const char *name);
```
<netdb.h> с. 687

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
struct
protoent  *getprotobynumber(int proto);
```
<netdb.h> с. 687

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
struct
protoent  *getprotoent(void);
```
<netdb.h> с. 687

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
struct
passwd    *getpwent(void);
```
<pwd.h> с. 232

Возвращает указатель в случае успеха, NULL — в случае ошибки или по достижении конца файла

```
struct
passwd    *getpwnam(const char *name);
```
<pwd.h> с. 232

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
struct
passwd    *getpwuid(uid_t uid);
```
<pwd.h> с. 232

Возвращает указатель в случае успеха, NULL — в случае ошибки

```
int       getrlimit(int resource, struct rlimit *rlptr);
```
<sys/resource.h> с. 277

resource: RLIMIT_CORE, RLIMIT_CPU,
RLIMIT_DATA, RLIMIT_FSIZE,
RLIMIT_NOFILE, RLIMIT_STACK,
RLIMIT_AS (FreeBSD 8.0, Linux 3.2.0, Solaris 10),
RLIMIT_MEMLOCK (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8),
RLIMIT_MSGQUEUE (Linux 3.2.0),
RLIMIT_NICE (Linux 3.2.0),
RLIMIT_NPROC (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8),
RLIMIT_NPTS (FreeBSD 8.0),
RLIMIT_RSS (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8),
RLIMIT_SBSIZE (FreeBSD 8.0),
RLIMIT_SIGPENDING (Linux 3.2.0),
RLIMIT_SWAP (FreeBSD 8.0),
RLIMIT_VMEM (Solaris 10)

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
char      *gets(char *buf);
                    <stdio.h>                                          с. 203
                    Возвращает указатель на buf в случае успеха, NULL —
                    по достижении конца файла или в случае ошибки

struct
servent   *getservbyname(const char *name, const char *proto);
                    <netdb.h>                                          с. 687
                    Возвращает указатель в случае успеха, NULL — в слу-
                    чае ошибки

struct
servent   *getservbyport(int port, const char *proto);
                    <netdb.h>                                          с. 687
                    Возвращает указатель в случае успеха, NULL — в слу-
                    чае ошибки

struct
servent   *getservent(void);
                    <netdb.h>                                          с. 687
                    Возвращает указатель в случае успеха, NULL — в слу-
                    чае ошибки

pid_t     getsid(pid_t pid);
                    <unistd.h>                                         с. 359
                    Возвращает идентификатор группы процессов ли-
                    дера сеанса в случае успеха, –1 — в случае ошибки

int       getsockname(int sockfd, struct sockaddr *restrict addr,
                      socklen_t *restrict alenp);
                    <sys/socket.h>                                     с. 693
                    Возвращает 0 в случае успеха, –1 — в случае ошибки

int       getsockopt(int sockfd, int level, int option, void *restrict val,
                     socklen_t *restrict lenp);
                    <sys/socket.h>                                     с. 714
                    Возвращает 0 в случае успеха, –1 — в случае ошибки

struct
spwd      *getspent(void);
                    <shadow.h>                                         с. 235
                    Возвращает указатель в случае успеха, NULL — в слу-
                    чае ошибки
                    Платформы: Linux 3.2.0, Solaris 10
```

```
struct
spwd      *getspnam(const char *name);
                    <shadow.h>                                          с. 235
                    Возвращает указатель в случае успеха, NULL — в слу-
                    чае ошибки
                    Платформы: Linux 3.2.0, Solaris 10

int       gettimeofday(struct timeval *restrict tp,
                    void *restrict tzp);
                    <sys/time.h>                                        с. 244
                    Всегда возвращает 0

uid_t     getuid(void);
                    <unistd.h>                                          с. 285
                    Возвращает реальный идентификатор пользователя
                    вызывающего процесса

struct
tm        *gmtime(const time_t *calptr);
                    <time.h>                                            с. 246
                    Возвращает указатель на структуру с временем, раз-
                    ложенным на составляющие, NULL — в случае ошибки

int       grantpt(int fd);
                    <stdlib.h>                                          с. 815
                    Возвращает 0 в случае успеха, –1 — в случае ошибки

uint32_t  htonl(uint32_t hostint32);
                    <arpa/inet.h>                                       с. 682
                    Возвращает 32-разрядное целое с сетевым порядком
                    байтов

uint16_t  htons(uint16_t hostint16);
                    <arpa/inet.h>                                       с. 682
                    Возвращает 16-разрядное целое с сетевым порядком
                    байтов

const
char      *inet_ntop(int domain, const void *restrict addr,
                  char *restrict str, socklen_t size);
                    <arpa/inet.h>                                       с. 684
                    Возвращает указатель на строку с адресом в случае
                    успеха, NULL — в случае ошибки
```

```
int     inet_pton(int domain, const char *restrict str,
                  void *restrict addr);
```
<arpa/inet.h> с. 684

Возвращает 1 в случае успеха, 0 — в случае неверного формата, –1 — в случае ошибки

```
int     initgroups(const char *username, gid_t basegid);
```
<grp.h> /* Linux и Solaris */ с. 237

<unistd.h> /* FreeBSD и Mac OS X */

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     ioctl(int fd, int request, ...);
```
<unistd.h> /* System V */ с. 132

<sys/ioctl.h> /* BSD и Linux */

Возвращает –1 в случае ошибки, любое другое значение — в случае успеха

```
int     isatty(int fd);
```
<unistd.h> с. 788

Возвращает 1 (истина), если это терминальное устройство, 0 (ложь) — в противном случае

```
int     kill(pid_t pid, int signo);
```
<signal.h> с. 405

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     lchown(const char *path, uid_t owner, gid_t group);
```
<unistd.h> с. 155

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     link(const char *existingpath, const char *newpath);
```
<unistd.h> с. 163

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     linkat(int efd, const char *existingpath, int nfd,
               const char *newpath, int flag);
```
<unistd.h> с. 163

flag: AT_SYMLINK_NOFOLLOW

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       lio_listio(int mode,
                     struct aiocb *restrict const list[restrict],
                     int nent, struct sigevent *restrict sigev);
```
<aio.h> с. 594

mode: LIO_NOWAIT, LIO_WAIT

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       listen(int sockfd, int backlog);
```
<sys/socket.h> с. 696

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
struct
tm        *localtime(const time_t *calptr);
```
<time.h> с. 246

Возвращает указатель на структуру с временем, разложенным на составляющие, NULL — в случае ошибки

```
void      longjmp(jmp_buf env, int val);
```
<setjmp.h> с. 272

Эта функция никогда не возвращает управление

```
off_t     lseek(int fd, off_t offset, int whence);
```
<unistd.h> с. 109

whence: SEEK_SET, SEEK_CUR, SEEK_END

Возвращает новую текущую позицию в файле в случае успеха, –1 — в случае ошибки

```
int       lstat(const char *restrict path,
          struct stat *restrict buf);
```
<sys/stat.h> с. 137

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
void      *malloc(size_t size);
```
<stdlib.h> с. 262

Возвращает непустой указатель в случае успеха, NULL — в случае ошибки

```
int       mkdir(const char *path, mode_t mode);
```
<sys/stat.h> с. 177

mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       mkdirat(int fd, const char *path, mode_t mode);
```
<sys/stat.h> с. 177

mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
char      *mkdtemp(char *template);
```
<stdlib.h> с. 221

Возвращает указатель на имя каталога в случае успеха, NULL — в случае ошибки

```
int       mkfifo(const char *path, mode_t mode);
```
<sys/stat.h> с. 634

mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       mkfifoat(int fd, const char *path, mode_t mode);
```
<sys/stat.h> с. 634

mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       mkstemp(char *template);
```
<stdlib.h> с. 221

Возвращает дескриптор файла в случае успеха, –1 — в случае ошибки

```
time_t    mktime(struct tm *tmptr);
```
<time.h> с. 246

Возвращает календарное время в случае успеха, –1 — в случае ошибки

```
void      *mmap(void *addr, size_t len, int prot, int flag, int fd,
              off_t off);
```
<sys/mman.h> с. 605

prot: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE

flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE

Возвращает начальный адрес отображенной области в случае успеха, MAP_FAILED — в случае ошибки

```
int       mprotect(void *addr, size_t len, int prot);
```
<sys/mman.h> с. 608

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       msgctl(int msqid, int cmd, struct msqid_ds *buf);
```
<sys/msg.h> с. 645

cmd: IPC_STAT, IPC_SET, IPC_RMID

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       msgget(key_t key, int flag);
```
<sys/msg.h> с. 644

flag: IPC_CREAT, IPC_EXCL

Возвращает идентификатор очереди сообщений в случае успеха, –1 — в случае ошибки

```
ssize_t   msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```
<sys/msg.h с. 647

flag: IPC_NOWAIT, MSG_NOERROR

Возвращает размер блока данных сообщения в случае успеха, –1 — в случае ошибки

```
int       msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```
<sys/msg.h> с. 645

flag: IPC_NOWAIT

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       msync(void *addr, size_t len, int flags);
```
<sys/mman.h> с. 608

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       munmap(void *addr, size_t len);
```
<sys/mman.h> с. 609

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       nanosleep(const struct timespec *reqtp,
                    struct timespec *remtp);
```
<time.h> с. 444

Возвращает 0, если установленное время истекло, –1 — в случае неудачи

```
int       nice(int incr);
```
<unistd.h> с. 339

Возвращает новое значение коэффициента уступчивости минус NZERO в случае успеха, –1 — в случае ошибки

```
uint32_t  ntohl(uint32_t netint32);
```
<arpa/inet.h> с. 682

Возвращает 32-разрядное целое с аппаратным порядком байтов

```
uint16_t  ntohs(uint16_t netint16);
```
<arpa/inet.h> с. 682

Возвращает 16-разрядное целое с аппаратным порядком байтов

```
int       open(const char *path, int oflag, ... /* mode_t mode */ );
```
<fcntl.h> с. 104

oflag: O_RDONLY, O_WRONLY, O_RDWR, O_EXEC,
O_SEARCH;
O_APPEND, O_CLOEXEC, O_CREAT,
O_DIRECTORY, O_DSYNC, O_EXCL,
O_NOCTTY, O_NOFOLLOW, O_NONBLOCK,
O_RSYNC, O_SYNC, O_TRUNC, O_TTY_INIT

mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)

Возвращает дескриптор файла в случае успеха, –1 — в случае ошибки

Платформы: флаг O_FSYNC — для FreeBSD 8.0 и Mac OS X 10.6.8

```
int       openat(int fd, const char *path, int oflag, ... /* mode_t mode */);
```
<fcntl.h> с. 104

oflag: O_RDONLY, O_WRONLY, O_RDWR, O_EXEC, O_SEARCH;
O_APPEND, O_CLOEXEC, O_CREAT,
O_DIRECTORY, O_DSYNC, O_EXCL,
O_NOCTTY, O_NOFOLLOW, O_NONBLOCK,
O_RSYNC, O_SYNC, O_TRUNC, O_TTY_INIT

mode: S_IS[UG]ID, S_ISVTX, S_I[RWX](USR|GRP|OTH)

Возвращает дескриптор файла в случае успеха, –1 — в случае ошибки

Платформы: флаг O_FSYNC — для FreeBSD 8.0 и Mac OS X 10.6.8

```
DIR       *opendir(const char *path);
```
<direct.h> с. 179

Возвращает указатель на структуру DIR в случае успеха, NULL — в случае ошибки

```
void      openlog(const char *ident, int option, int facility);
```
<syslog.h> с. 545

option: LOG_CONS, LOG_NDELAY, LOG_NOWAIT, LOG_ODELAY, LOG_PERROR, LOG_PID

facility: LOG_AUTH, LOG_AUTHPRIV, LOG_CRON, LOG_DAEMON, LOG_FTP, LOG_KERN, LOG_LOCAL[0-7], LOG_LPR, LOG_MAIL, LOG_NEWS, LOG_SYSLOG, LOG_USER, LOG_UUCP

```
FILE      *open_memstream(char **bufp, size_t *sizep);
```
<stdio.h> с. 226

Возвращает указатель на поток ввода/вывода в случае успеха, NULL — в случае ошибки

```
FILE      *open_wmemstream(wchar_t **bufp, size_t *sizep);
```
<wchar.h> с. 226

Возвращает указатель на поток ввода/вывода в случае успеха, NULL — в случае ошибки

```
long      pathconf(const char *path, int name);
```
<unistd.h> с. 82

name: _PC_ASYNC_IO, _PC_CHOWN_RESTRICTED, _PC_FILESIZEBITS, _PC_LINK_MAX, _PC_MAX_CANON, _PC_MAX_INPUT, _PC_NAME_MAX, _PC_NO_TRUNC, _PC_PATH_MAX, _PC_PIPE_BUF, _PC_PRIO_IO, _PC_SYMLINK_MAX, _PC_SYNC_IO, _PC_TIMESTAMP_RESOLUTION, _PC_2_SYMLINKS, _PC_VDISABLE

Возвращает соответствующее значение в случае успеха, –1 — в случае ошибки

```
int       pause(void);
```
<unistd.h> с. 407

В случае ошибки возвращает –1 и код ошибки EINTR в переменной errno

```
int       pclose(FILE *fp);
```
<stdio.h> с. 623

Возвращает код завершения команды cmdstring функции popen, –1 — в случае ошибки

```
void      perror(const char *msg);
```
<stdio.h> с. 49

```
int       pipe(int fd[2]);
```
<unistd.h> с. 616

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```
<poll.h> с. 585

Возвращает количество дескрипторов, готовых к выполнению операции, 0 — в случае истечения времени тайм-аута, –1 — в случае ошибки

```
FILE      *popen(const char *cmdstring, const char *type);
```
<stdio.h> с. 623

type: `"r"`, `"w"`

Возвращает указатель на структуру `FILE` в случае успеха, `NULL` — в случае ошибки

```
int       posix_openpt(int oflag);
```
<stdlib.h> с. 815

<fcntl.h>

oflag: `O_RWDR`, `O_NOCTTY`

Возвращает дескриптор следующего доступного ведущего PTY в случае успеха, –1 — в случае ошибки

```
ssize_t   pread(int fd, void *buf, size_t nbytes, off_t offset);
```
<unistd.h> с. 122

Возвращает количество прочитанных байтов, 0 по достижении конца файла, –1 — в случае ошибки

```
int       printf(const char *restrict format, ...);
```
<stdio.h> с. 210

Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int       pselect(int maxfdp1, fd_set *restrict readfds,
              fd_set *restrict writefds, fd_set *restrict exceptfds,
              const struct timespec *restrict tsptr,
              const sigset_t *restrict sigmask);
```
<sys/select.h> с. 584

Возвращает количество дескрипторов, готовых к выполнению операции, 0 — в случае истечения тайм-аута, –1 — в случае ошибки

```
void     psiginfo(const siginfo_t *info, const char *msg);
                  <signal.h>                                          с. 450

void     psignal(int signo, const char *msg);
                  <signal.h>                                          с. 450
                  <siginfo.h> /* в Solaris */

int      pthread_atfork(void (*prepare)(void), void (*parent)(void),
                    void (*child)(void);
                  <pthread.h>                                         с. 531
                  Возвращает 0 в случае успеха, код ошибки — в слу-
                  чае неудачи

int      pthread_attr_destroy(pthread_attr_t *attr);
                  <pthread.h>                                         с. 498
                  Возвращает 0 в случае успеха, код ошибки — в слу-
                  чае неудачи

int      pthread_attr_getdetachstate(const pthread_attr_t *attr,
                           int *detachstate);
                  <pthread.h>                                         с. 499
                  Возвращает 0 в случае успеха, код ошибки — в слу-
                  чае неудачи

int      pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                              size_t *restrict guardsize);
                  <pthread.h>                                         с. 501
                  Возвращает 0 в случае успеха, код ошибки — в слу-
                  чае неудачи

int      pthread_attr_getstack(const pthread_attr_t *restrict attr,
                           void **restrict stackaddr,
                           size_t *restrict stacksize);
                  <pthread.h>                                         с. 500
                  Возвращает 0 в случае успеха, код ошибки — в слу-
                  чае неудачи

int      pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                          size_t *restrict stacksize);
                  <pthread.h>                                         с. 501
                  Возвращает 0 в случае успеха, код ошибки — в слу-
                  чае неудачи
```

```
int     pthread_attr_init(pthread_attr_t *attr);
```
<pthread.h> с. 498

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```
<pthread.h> с. 499

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```
<pthread.h> с. 501

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_attr_setstack(const pthread_attr_t *attr,
                  void *stackaddr, size_t *stacksize);
```
<pthread.h> с. 500

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_attr_setstacksize(pthread_attr_t *attr,
                size_t stacksize);
```
<pthread.h> с. 501

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```
<pthread.h> с. 513

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_barrierattr_getpshared(const pthread_barrierattr_t
                    *restrict attr,
                  int *restrict pshared);
```
<pthread.h> с. 513

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_barrierattr_init(pthread_barrierattr_t *attr);
```
<pthread.h> с. 513

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                       int pshared);
```
<pthread.h> с. 513

pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_barrier_destroy(pthread_barrier_t *barrier);
```
<pthread.h> с. 491

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_barrier_init(pthread_barrier_t *restrict barrier,
                             const pthread_barrierattr_t *restrict attr,
                             unsigned int count);
```
<pthread.h> с. 491

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_barrier_wait(pthread_barrier_t *barrier);
```
<pthread.h> с. 492

Возвращает 0 или PTHREAD_BARRIER_SERIAL_THREAD в случае успеха, код ошибки — в случае неудачи

```
int     pthread_cancel(pthread_t tid);
```
<pthread.h> с. 465

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
void    pthread_cleanup_pop(int execute);
```
<pthread.h> с. 465

```
void    pthread_cleanup_push(void (*rtn)(void *), void *arg);
```
<pthread.h> с. 465

```
int     pthread_condattr_destroy(pthread_condattr_t *attr);
```
<pthread.h> с. 512

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_condattr_getclock(const pthread_condattr_t *restrict attr,
                                  clockid_t *restrict clock_id);
```

<pthread.h> с. 512

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_condattr_getpshared(const pthread_condattr_t *restrict attr,
                                    int *restrict pshared);
```

<pthread.h> с. 512

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_condattr_init(pthread_condattr_t *attr);
```

<pthread.h> с. 512

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_condattr_setclock(pthread_condattr_t *attr,
                                  clockid_t clock_id);
```

<pthread.h> с. 512

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_condattr_setpshared(pthread_condattr_t *attr,
                                    int pshared);
```

<pthread.h> с. 512

pshared: PTHREAD_PROCESS_PRIVATE,
PTHREAD_PROCESS_SHARED

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_cond_broadcast(pthread_cond_t *cond);
```

<pthread.h> с. 487

Возвращает 0 в случае успеха, код ошибки в случае неудачи

```
int     pthread_cond_destroy(pthread_cond_t *cond);
```

<pthread.h> с. 486

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_cond_init(pthread_cond_t *restrict cond,
                  pthread_condattr_t *restrict attr);
```
<pthread.h> с. 486

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_cond_signal(pthread_cond_t *cond);
```
<pthread.h> с. 487

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_cond_timedwait(pthread_cond_t *restrict cond,
                       pthread_mutex_t *restrict mutex,
                       const struct timespec *restrict timeout);
```
<pthread.h> с. 486

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_cond_wait(pthread_cond_t *restrict cond,
                  pthread_mutex_t *restrict mutex);
```
<pthread.h> с. 486

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_create(pthread_t *restrict tidp,
                const pthread_attr_t *restrict attr,
                void *(*start_rtn)(void), void *restrict arg);
```
<pthread.h> с. 457

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_detach(pthread_t tid);
```
<pthread.h> с. 468

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_equal(pthread_t tid1, pthread_t tid2);
```
<pthread.h> с. 456

Возвращает ненулевое значение, если идентификаторы равны, 0 — в противном случае

```
void    pthread_exit(void *rval_ptr);
```
<pthread.h> с. 460

```
void      *pthread_getspecific(pthread_key_t key);
                    <pthread.h>                                          с. 522
```
Возвращает адрес области памяти с локальными данными потока или NULL, если ключ *key* не был ассоциирован с локальными данными

```
int       pthread_join(pthread_t thread, void **rval_ptr);
                    <pthread.h>                                          с. 461
```
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_key_create(pthread_key_t *keyp, void (*destructor)(void *);
                    <pthread.h>                                          с. 519
```
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_key_delete(pthread_key_t key);
                    <pthread.h>                                          с. 520
```
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_kill(pthread_t thread, int signo);
                    <signal.h>                                           с. 528
```
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
                    <pthread.h>                                          с. 502
```
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_mutexattr_getpshared(const pthread_mutexattr_t *restrict attr,
                                       int *restrict pshared);
                    <pthread.h>                                          с. 503
```
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_mutexattr_getrobust(const pthread_mutexattr_t *restrict attr,
                                      int *restrict robust);
                    <pthread.h>                                          с. 503
```
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict attr,
                                  int *restrict type);
```

<pthread.h> с. 505

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

<pthread.h> с. 502

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

<pthread.h> с. 503

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                                    int robust);
```

<pthread.h> с. 503

robust: PTHREAD_MUTEX_ROBUST,
PTHREAD_MUTEX_STALLED

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

<pthread.h> с. 505

type: PTHREAD_MUTEX_NORMAL,
PTHREAD_MUTEX_ERRORCHECK,
PTHREAD_MUTEX_RECURSIVE,
PTHREAD_MUTEX_DEFAULT

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutex_consistent(pthread_mutex_t *mutex);
```

<pthread.h> с. 504

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutex_destroy(pthread_mutex_t *mutex);
```

<pthread.h> с. 472

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutex_init(pthread_mutex_t *restrict mutex,
                    const pthread_mutexattr_t *restrict attr);
```
<pthread.h> с. 472

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutex_lock(pthread_mutex_t *mutex);
```
<pthread.h> с. 472

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                        const struct timespec *restrict tsptr);
```
<pthread.h> с. 479

<time.h>

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutex_trylock(pthread_mutex_t *mutex);
```
<pthread.h> с. 472

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_mutex_unlock(pthread_mutex_t *mutex);
```
<pthread.h> с. 472

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_once(pthread_once_t *initflag, void (*initfn)(void);
```
<pthread.h> с. 521

pthread_once_t *initflag* = PTHREAD_ONCE_INIT;

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```
<pthread.h> с. 511

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr,
                            int *restrict pshared);
```
<pthread.h> с. 511

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```
<pthread.h> с. 511

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                      int pshared);
```
<pthread.h> с. 511

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```
<pthread.h> с. 481

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                            const pthread_rwlockattr_t *restrict attr);
```
<pthread.h> с. 481

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```
<pthread.h> с. 482

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                                   const struct timespec *restrict tsptr);
```
<pthread.h> с. 485

<time.h>

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                                   const struct timespec *restrict tsptr);
```
<pthread.h> с. 485

<time.h>

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```
<pthread.h> с. 482

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```
<pthread.h> с. 482
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```
<pthread.h> с. 482
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```
<pthread.h> с. 482
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
pthread_t pthread_self(void);
```
<pthread.h> с. 456
Возвращает идентификатор вызывающего потока

```
int       pthread_setcancelstate(int state, int *oldstate);
```
<pthread.h> с. 523
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_setcanceltype(int type, int *oldtype);
```
<pthread.h> с. 526
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_setspecific(pthread_key_t key, const void *value);
```
<pthread.h> с. 522
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_sigmask(int how, const sigset_t *restrict set,
              sigset_t *restrict oset);
```
<signal.h> с. 527
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int       pthread_spin_destroy(pthread_spinlock_t *lock);
```
<pthread.h> с. 490
Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_spin_init(pthread_spinlock_t *lock, int pshared);
```
<pthread.h> с. 490

pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_spin_lock(pthread_spinlock_t *lock);
```
<pthread.h> с. 490

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_spin_trylock(pthread_spinlock_t *lock);
```
<pthread.h> с. 490

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
int     pthread_spin_unlock(pthread_spinlock_t *lock);
```
<pthread.h> с. 490

Возвращает 0 в случае успеха, код ошибки — в случае неудачи

```
void    pthread_testcancel(void);
```
<pthread.h> с. 526

```
char    *ptsname(int fd);
```
<stdlib.h> с. 816

Возвращает указатель на имя ведомого PTY в случае успеха, NULL — в случае ошибки

```
int     putc(int c, FILE *fp);
```
<stdio.h> с. 202

Возвращает символ *c* в случае успеха, EOF — в случае ошибки

```
int     putchar(int c);
```
<stdio.h> с. 202

Возвращает символ *c* в случае успеха, EOF — в случае ошибки

```
int     putchar_unlocked(int c);
```
<stdio.h> с. 516

Возвращает символ *c* в случае успеха, EOF — в случае ошибки

```
int       putc_unlocked(int c, FILE *fp);
```
<stdio.h> с. 516

Возвращает символ *c* в случае успеха, EOF — в случае ошибки

```
int       putenv(char *str);
```
<stdlib.h> с. 268

Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

```
int       puts(const char *str);
```
<stdio.h> с. 204

Возвращает неотрицательное значение в случае успеха, EOF — в случае ошибки

```
ssize_t   pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```
<unistd.h> с. 122

Возвращает количество записанных байтов в случае успеха, –1 — в случае ошибки

```
int       raise(int signo);
```
<signal.h> с. 405

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
ssize_t   read(int fd, void *buf, size_t nbytes);
```
<unistd.h> с. 113

Возвращает количество прочитанных байтов в случае успеха, 0 — по достижении конца файла, –1 — в случае ошибки

```
struct
dirent    *readdir(DIR *dp);
```
<dirent.h> с. 179

Возвращает указатель в случае успеха, NULL — по достижении конца каталога или в случае ошибки

```
ssize_t   readlink(const char *restrict path, char *restrict buf,
                   size_t bufsize);
```
<unistd.h> с. 171

Возвращает количество прочитанных байтов в случае успеха, –1 — в случае ошибки

```
ssize_t   readlinkat(int fd, const char *restrict path,
                     char *restrict buf, size_t bufsize);
```

<unistd.h> с. 171

Возвращает количество прочитанных байтов в случае успеха, –1 — в случае ошибки

```
ssize_t   readv(int fd, const struct iovec *iov, int iovcnt);
```

<sys/uio.h> с. 600

Возвращает количество прочитанных байтов в случае успеха, 0 — по достижении конца файла, –1 — в случае ошибки

```
void      *realloc(void *ptr, size_t newsize);
```

<stdlib.h> с. 262

Возвращает непустой указатель в случае успеха, NULL — в случае ошибки

```
ssize_t   recv(int sockfd, void *buf, size_t nbytes, int flags);
```

<sys/socket.h> с. 701

flags: MSG_PEEK, MSG_OOB, MSG_WAITALL,
MSG_CMSG_CLOEXEC (Linux 3.2.0),
MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0, Solaris 10),
MSG_ERRQUEUE (Linux 3.2.0),
MSG_TRUNC (Linux 3.2.0)

Возвращает длину сообщения в байтах, 0 — если нет доступных сообщений и на другом конце соединения была запрещена операция записи, –1 — в случае ошибки

```
ssize_t   recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
                   struct sockaddr *restrict addr,
                   socklen_t *restrict addrlen);
```

<sys/socket.h> с. 702

flags: MSG_PEEK, MSG_OOB, MSG_WAITALL,
MSG_CMSG_CLOEXEC (Linux 3.2.0),
MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0, Solaris 10),
MSG_ERRQUEUE (Linux 3.2.0),
MSG_TRUNC (Linux 3.2.0)

Возвращает длину сообщения в байтах, 0 — если нет доступных сообщений и на другом конце соединения запрещена операция записи, –1 — в случае ошибки

```
ssize_t   recvmsg(int sockfd, struct msghdr *msg, int flags);
                  <sys/socket.h>                                    с. 702
                  flags: MSG_PEEK, MSG_OOB, MSG_WAITALL,
                         MSG_CMSG_CLOEXEC (Linux 3.2.0),
                         MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                                       Solaris 10),
                         MSG_ERRQUEUE (Linux 3.2.0),
                         MSG_TRUNC (Linux 3.2.0)
                  Возвращает длину сообщения в байтах, 0 — если
                  нет доступных сообщений и на другом конце соеди-
                  нения запрещена операция записи, –1 — в случае
                  ошибки

int       remove(const char *path);
                  <stdio.h>                                         с. 165
                  Возвращает 0 в случае успеха, –1 — в случае ошибки

int       rename(const char *oldname, const char *newname);
                  <stdio.h>                                         с. 166
                  Возвращает 0 в случае успеха, –1 — в случае ошибки

int       renameat(int oldfd, const char *oldname, int newfd,
                   const char *newname);
                  <stdio.h>                                         с. 166
                  Возвращает 0 в случае успеха, –1 — в случае ошибки

void      rewind(FILE *fp);
                  <stdio.h>                                         с. 209

void      rewinddir(DIR *dp);
                  <dirent.h>                                        с. 179

int       rmdir(const char *path);
                  <unistd.h>                                        с. 178
                  Возвращает 0 в случае успеха, –1 — в случае ошибки

int       scanf(const char *restrict format, ...);
                  <stdio.h>                                         с. 214
                  Возвращает количество введенных элементов, EOF —
                  по достижении конца файла или в случае ошибки
                  перед выполнением преобразования
```

```
void     seekdir(DIR *dp, long loc);
                 <dirent.h>                                          с. 179

int      select(int maxfdp1, fd_set *restrict readfds,
                fd_set *restrict writefds,
                fd_set *restrict exceptfds, struct timeval *restrict tvptr);
                 <sys/select.h>                                      с. 580
                 Возвращает количество дескрипторов, готовых
                 к выполнению операции, 0 — по истечении тайм-
                 аута, –1 — в случае ошибки

int      sem_close(sem_t *sem);
                 <semaphore.h>                                       с. 665
                 Возвращает 0 в случае успеха, –1 — в случае ошибки

int      semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
                 <sys/sem.h>                                         с. 651
                 cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID,
                      GETNCNT, GETZCNT, GETVAL, SETVAL,
                      GETALL, SETALL
                 Возвращаемое значение зависит от типа команды,
                 –1 — в случае ошибки

int      sem_destroy(sem_t *sem);
                 <semaphore.h>                                       с. 668
                 Возвращает 0 в случае успеха, –1 — в случае ошибки

int      semget(key_t key, int nsems, int flag);
                 <sys/sem.h>                                         с. 651
                 flag: IPC_CREAT, IPC_EXCL
                 Возвращает идентификатор семафора в случае успе-
                 ха, –1 — в случае ошибки

int      sem_getvalue(sem_t *restrict sem, int *restrict valp);
                 <semaphore.h>                                       с. 668
                 Возвращает 0 в случае успеха, –1 — в случае ошибки

int      sem_init(sem_t *sem, int pshared, unsigned int value);
                 <semaphore.h>                                       с. 667
                 Возвращает 0 в случае успеха, –1 — в случае ошибки

int      semop(int semid, struct sembuf semoparray[], size_t nops);
                 <sys/sem.h>                                         с. 652
                 Возвращает 0 в случае успеха, –1 — в случае ошибки
```

```
sem_t     *sem_open(const char *name, int oflag, ...
                    /* mode_t mode, unsigned int value */ );
```
с. 664

<semaphore.h>

flag: IPC_CREAT, IPC_EXCL

Возвращает указатель на семафор в случае успеха, SEM_FAILED — в случае ошибки

```
int       sem_post(sem_t *sem);
```
с. 667

<semaphore.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict tsptr);
```
с. 667

<semaphore.h>

<time.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       sem_trywait(sem_t *sem);
```
с. 666

<semaphore.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       sem_unlink(const char *name);
```
с. 666

<semaphore.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       sem_wait(sem_t *sem);
```
с. 666

<semaphore.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
ssize_t   send(int sockfd, const void *buf, size_t nbytes, int flags);
```
с. 698

<sys/socket.h>

flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL,
MSG_CONFIRM (Linux 3.2.0),
MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10),
MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10),
MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
MSG_MORE (Linux 3.2.0)

Возвращает количество переданных байтов в случае успеха, –1 — в случае ошибки

```
ssize_t   sendmsg(int sockfd, const struct msghdr *msg, int flags);
```
с. 700

<sys/socket.h>

flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL,
MSG_CONFIRM (Linux 3.2.0),
MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10),
MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10),
MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
MSG_MORE (Linux 3.2.0)

Возвращает количество переданных байтов в случае успеха, –1 — в случае ошибки

```
ssize_t   sendto(int sockfd, const void *buf, size_t nbytes, int flags,
                 const struct sockaddr *destaddr, socklen_t destlen);
```
с. 699

<sys/socket.h>

flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL,
MSG_CONFIRM (Linux 3.2.0),
MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10),
MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10),
MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
MSG_MORE (Linux 3.2.0)

Возвращает количество переданных байтов в случае успеха, –1 — в случае ошибки

```
void      setbuf(FILE *restrict fp, char *restrict buf);
```
с. 196

<stdio.h>

```
int       setegid(gid_t gid);
```
с. 319

<unistd.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       setenv(const char *name, const char *value, int rewrite);
```
с. 268

<stdlib.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       seteuid(uid_t uid);
```
с. 319

<unistd.h>

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     setgid(gid_t gid);
                <unistd.h>                                          с. 316
                Возвращает 0 в случае успеха, –1 — в случае ошибки

void    setgrent(void);
                <grp.h>                                             с. 236

int     setgroups(int ngroups, const gid_t grouplist[]);
                <grp.h> /* в Linux */                               с. 237
                <unistd.h> /* в FreeBSD, Mac OS X и Solaris */
                Возвращает 0 в случае успеха, –1 — в случае ошибки

void    sethostent(int stayopen);
                <netdb.h>                                           с. 685

int     setjmp(jmp_buf env);
                <setjmp.h>                                          с. 272
                Возвращает 0, если вызывается непосредственно, ненулевое значение — если возврат произошел вследствие вызова longjmp

int     setlogmask(int maskpri);
                <syslog.h>                                          с. 545
                Возвращает предыдущее значение маски приоритета журналируемых сообщений

void    setnetent(int stayopen);
                <netdb.h>                                           с. 686

int     setpgid(pid_t pid, pid_t pgid);
                <unistd.h>                                          с. 357
                Возвращает 0 в случае успеха, –1 — в случае ошибки

int     setpriority(int which, id_t who, int value);
                <sys/resource.h>                                    с. 340
                which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER
                Возвращает 0 в случае успеха, –1 — в случае ошибки

void    setprotoent(int stayopen);
                <netdb.h>                                           с. 687

void    setpwent(void);
                <pwd.h>                                             с. 232
```

```
int     setregid(gid_t rgid, gid_t egid);
                <unistd.h>                                          с. 318
```
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     setreuid(uid_t ruid, uid_t euid);
                <unistd.h>                                          с. 318
```
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     setrlimit(int resource, const struct rlimit *rlptr);
                <sys/resource.h>                                    с. 277
```
resource: `RLIMIT_CORE`, `RLIMIT_CPU`,
`RLIMIT_DATA`, `RLIMIT_FSIZE`,
`RLIMIT_NOFILE`, `RLIMIT_STACK`,
`RLIMIT_AS` (FreeBSD 8.0, Linux 3.2.0, Solaris 10),
`RLIMIT_MEMLOCK` (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8),
`RLIMIT_MSGQUEUE` (Linux 3.2.0),
`RLIMIT_NICE` (Linux 3.2.0),
`RLIMIT_NPROC` (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8),
`RLIMIT_NPTS` (FreeBSD 8.0),
`RLIMIT_RSS` (FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8),
`RLIMIT_SBSIZE` (FreeBSD 8.0),
`RLIMIT_SIGPENDING` (Linux 3.2.0),
`RLIMIT_SWAP` (FreeBSD 8.0),
`RLIMIT_VMEM` (Solaris 10)
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
void    setservent(int stayopen);
                <netdb.h>                                           с. 687
```

```
pid_t   setsid(void);
                <unistd.h>                                          с. 358
```
Возвращает идентификатор группы процессов в случае успеха, –1 — в случае ошибки

```
int     setsockopt(int sockfd, int level, int option, const void *val,
            socklen_t len);
                <sys/socket.h>                                      с. 713
```
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
void    setspent(void);
                <shadow.h>                                          с. 235
```
Платформы: Linux 3.2.0, Solaris 10

```
int     setuid(uid_t uid);
```
<unistd.h> с. 316
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);
```
<stdio.h> с. 196
mode: _IOFBF, _IOLBF, _IONBF
Возвращает 0 в случае успеха, ненулевое значение — в случае ошибки

```
void    *shmat(int shmid, const void *addr, int flag);
```
<sys/shm.h> с. 658
flag: SHM_RND, SHM_RDONLY
Возвращает указатель на сегмент разделяемой памяти в случае успеха, –1 — в случае ошибки

```
int     shmctl(int shmid, int cmd, struct shmid_ds *buf);
```
<sys/shm.h> с. 658
cmd: IPC_STAT, IPC_SET, IPC_RMID,
SHM_LOCK (Linux 3.2.0, Solaris 10),
SHM_UNLOCK (Linux 3.2.0, Solaris 10)
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     shmdt(void *addr);
```
<sys/shm.h> с. 659
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     shmget(key_t key, size_t size, int flag);
```
<sys/shm.h> с. 657
flag: IPC_CREAT, IPC_EXCL
Возвращает неотрицательный идентификатор сегмента разделяемой памяти в случае успеха, –1 — в случае ошибки

```
int     shutdown(int sockfd, int how);
```
<sys/socket.h> с. 680
how: SHUT_RD, SHUT_WR, SHUT_RDWR
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sig2str(int signo, char *str);
```
<signal.h> с. 451
Возвращает 0 в случае успеха, –1 — в случае ошибки
Платформы: Solaris 10

```
int     sigaction(int signo, const struct sigaction *restrict act,
            struct sigaction *restrict oact);
```
<signal.h> с. 418

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sigaddset(sigset_t *set, int signo);
```
<signal.h> с. 412

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sigdelset(sigset_t *set, int signo);
```
<signal.h> с. 412

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sigemptyset(sigset_t *set);
```
<signal.h> с. 412

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sigfillset(sigset_t *set);
```
<signal.h> с. 412

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sigismember(const sigset_t *set, int signo);
```
<signal.h> с. 412

Возвращает 1, если утверждение истинно, 0 — если ложно, –1 — в случае ошибки

```
void    siglongjmp(sigjmp_buf env, int val);
```
<setjmp.h> с. 425

Эта функция никогда не возвращает управление

```
void    (*signal(int signo, void (*func)(int)))(int);
```
<signal.h> с. 389

Возвращает предыдущую диспозицию сигнала в случае успеха, SIG_ERR — в случае ошибки

```
int     sigpending(sigset_t *set);
```
<signal.h> с. 416

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sigprocmask(int how, const sigset_t *restrict set,
            sigset_t *restrict oset);
```
<signal.h> с. 414

how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       sigqueue(pid_t pid, int signo, const union sigval value)
```
<signal.h> с. 446

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       sigsetjmp(sigjmp_buf env, int savemask);
```
<setjmp.h> с. 425

Возвращает 0, если вызывается непосредственно, ненулевое значение — если возврат произошел вследствие вызова siglongjmp

```
int       sigsuspend(const sigset_t *sigmask);
```
<signal.h> с. 428

Возвращает –1, с кодом ошибки EINTR в переменной errno

```
int       sigwait(const sigset_t *restrict set, int *restrict signop);
```
<signal.h> с. 528

Возвращает 0 в случае успеха и код ошибки — в случае неудачи

```
unsigned
int       sleep(unsigned int seconds);
```
<unistd.h> с. 442

Возвращает 0 или количество секунд, оставшихся до окончания приостановки

```
int       snprintf(char *restrict buf, size_t n,
                   const char *restrict format, ...);
```
<stdio.h> с. 210

Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение — в случае ошибки преобразования

```
int       sockatmark(int sockfd);
```
<sys/socket.h> с. 716

Возвращает 1, если достигнут маркер, 0 — если нет, –1 — в случае ошибки

```
int       socket(int domain, int type, int protocol);
```
<sys/socket.h> с. 677

type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET,

Возвращает дескриптор файла (сокета) в случае успеха, –1 — в случае ошибки

```
int     socketpair(int domain, int type, int protocol, int sockfd[2]);
```
<sys/socket.h> с. 719

type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     sprintf(char *restrict buf, const char *restrict format, ...);
```
<stdio.h> с. 210

Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение — в случае ошибки преобразования

```
int     sscanf(const char *restrict buf, const char *restrict format, ...);
```
<stdio.h> с. 214

Возвращает количество введенных элементов, EOF — по достижении конца файла или в случае ошибки перед выполнением преобразования

```
int     stat(const char *restrict path, struct stat *restrict buf);
```
<sys/stat.h> с. 137

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     str2sig(const char *str, int *signop);
```
<signal.h> с. 451

Возвращает 0 в случае успеха, –1 — в случае ошибки

Платформы: Solaris 10

```
char    *strerror(int errnum);
```
<string.h> с. 49

Возвращает указатель на строку сообщения

```
size_t  strftime(char *restrict buf, size_t maxsize,
                 const char *restrict format,
                 const struct tm *restrict tmptr);
```
<time.h> с. 246

Возвращает количество символов, сохраненных в массиве, если достаточно места, 0 — в противном случае

```
size_t  strftime_l(char *restrict buf, size_t maxsize,
                   const char *restrict format,
                   const struct tm *restrict tmptr, locale_t locale);
```
<time.h> с. 246

Возвращает количество символов, сохраненных в массиве, если достаточно места, 0 — в противном случае

```
char      *strptime(const char *restrict buf, const char *restrict format,
                    struct tm *restrict tmptr);
```
<time.h> с. 249

Возвращает указатель на символ, находящийся за последним проанализированным символом, NULL — в противном случае

```
char      *strsignal(int signo);
```
<string.h> с. 450

Возвращает указатель на строку с описанием сигнала

```
int       symlink(const char *actualpath, const char *sympath);
```
<unistd.h> с. 170

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       symlinkat(const char *actualpath, int fd, const char *sympath);
```
<unistd.h> с. 170

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
void      sync(void);
```
<unistd.h> с. 125

```
long      sysconf(int name);
```
<unistd.h> с. 82

name: _SC_ARG_MAX, _SC_ASYNCHRONOUS_IO,
_SC_ATEXIT_MAX, _SC_BARRIERS,
_SC_CHILD_MAX, _SC_CLK_TCK,
_SC_CLOCK_SELECTION, _SC_COLL_WEIGHTS_MAX,
_SC_DELAYTIMER_MAX, _SC_HOST_NAME_MAX,
_SC_IOV_MAX, _SC_JOB_CONTROL, _SC_LINE_MAX,
_SC_LOGIN_NAME_MAX, _SC_MAPPED_FILED,
_SC_MEMORY_PROTECTION, _SC_NGROUPS_MAX,
_SC_OPEN_MAX, _SC_PAGESIZE,
_SC_PAGE_SIZE, _SC_READER_WRITER_LOCKS,
_SC_REALTIME_SIGNALS, _SC_RE_DUP_MAX,
_SC_RTSIG_MAX, _SC_SAVED_IDS,
_SC_SEMAPHORES, _SC_SEM_NSEMS_MAX,
_SC_SEM_VALUE_MAX, _SC_SHELL,
_SC_SIGQUEUE_MAX, _SC_SPIN_LOCKS,
_SC_STREAM_MAX, _SC_SYMLOOP_MAX,
_SC_THREAD_SAFE_FUNCTIONS,

```
                    _SC_THREADS, _SC_TIMER_MAX,
                    _SC_TIMERS, _SC_TTY_NAME_MAX,
                    _SC_TZNAME_MAX, _SC_VERSION,
                    _SC_XOPEN_CRYPT, _SC_XOPEN_REALTIME,
                    _SC_XOPEN_REALTIME_THREADS, _SC_XOPEN_SHM,
                    _SC_XOPEN_VERSION
```

Возвращает соответствующее значение в случае успеха, –1 — в случае ошибки

```
void    syslog(int priority, const char *format, ...);
                <syslog.h>                                          с. 545

int     system(const char *cmdstring);
                <stdlib.h>                                          с. 326
```

Возвращает код завершения командной оболочки

```
int     tcdrain(int fd);
                <termios.h>                                         с. 786
```

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     tcflow(int fd, int action);
                <termios.h>                                         с. 786
```

action: `TCOOFF`, `TCOON`, `TCIOFF`, `TCION`

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     tcflush(int fd, int queue);
                <termios.h>                                         с. 786
```

queue: `TCIFLUSH`, `TCOFLUSH`, `TCIOFLUSH`

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     tcgetattr(int fd, struct termios *termptr);
                <termios.h>                                         с. 776
```

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
pid_t   tcgetpgrp(int fd);
                <unistd.h>                                          с. 361
```

Возвращает идентификатор группы процессов переднего плана в случае успеха, –1 — в случае ошибки

```
pid_t   tcgetsid(int fd);
                <termios.h>                                         с. 362
```

Возвращает идентификатор группы процессов лидера сеанса в случае успеха, –1 — в случае ошибки

```
int     tcsendbreak(int fd, int duration);
```
<termios.h> с. 786

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     tcsetattr(int fd, int opt, const struct termios *termptr);
```
<termios.h> с. 776

opt: TCSANOW, TCSADRAIN, TCSAFLUSH

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     tcsetpgrp(int fd, pid_t pgrpid);
```
<unistd.h> с. 361

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
long    telldir(DIR *dp);
```
<dirent.h> с. 179

Возвращает значение текущей позиции в каталоге, ассоциированное с *dp*

```
time_t  time(time_t *calptr);
```
<time.h> с. 243

Возвращает значение текущего времени в случае успеха, –1 — в случае ошибки

```
clock_t times(struct tms *buf);
```
<sys/times.h> с. 342

Возвращает значение общего времени выполнения процесса в тактах в случае успеха, –1 — в случае ошибки

```
FILE    *tmpfile(void);
```
<stdio.h> с. 222

Возвращает указатель на структуру FILE в случае успеха, NULL — в случае ошибки

```
char    *tmpnam(char *ptr);
```
<stdio.h> с. 222

Возвращает указатель на строку с уникальным именем файла

```
int     truncate(const char *path, off_t length);
```
<unistd.h> с. 158

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
char      *ttyname(int fd);
```
<unistd.h> с. 788

Возвращает указатель на строку с именем специального файла устройства терминала, NULL — в случае ошибки

```
mode_t    umask(mode_t cmask);
```
<sys/stat.h> с. 149

Возвращает предыдущее значение маски режима создания файлов

```
int       uname(struct utsname *name);
```
<sys/utsname.h> с. 241

Возвращает неотрицательное значение в случае успеха, –1 — в случае ошибки

```
int       ungetc(int c, FILE *fp);
```
<stdio.h> с. 202

Возвращает *c* в случае успеха, EOF — в случае ошибки

```
int       unlink(const char *path);
```
<unistd.h> с. 164

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       unlinkat(int fd, const char *path, int flag);
```
<unistd.h> с. 164

flag: AT_REMOVEDIR

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       unlockpt(int fd);
```
<stdlib.h> с. 815

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       unsetenv(const char *name);
```
<stdlib.h> с. 268

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int       utimensat(int fd, const char *path,
              const struct timespec times[2], int flag);
```
<sys/stat.h> с. 174

flag: AT_SYMLINK_NOFOLLOW

Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     utimes(const char *path, const struct timeval times[2]);
                <sys/time.h>                                          с. 175
```
Возвращает 0 в случае успеха, –1 — в случае ошибки

```
int     vdprintf(int fd, const char *restrict format, va_list arg);
                <stdarg.h>                                            с. 213
                <stdio.h>
```
Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int     vfprintf(FILE *restrict fp, const char *restrict format, va_list arg);
                <stdarg.h>                                            с. 213
                <stdio.h>
```
Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int     vfscanf(FILE *restrict fp, const char *restrict format, va_list arg);
                <stdarg.h>                                            с. 216
                <stdio.h>
```
Возвращает количество введенных элементов, `EOF` — в случае ошибки ввода или по достижении конца файла перед выполнением преобразования

```
int     vprintf(const char *restrict format, va_list arg);
                <stdarg.h>                                            с. 213
                <stdio.h>
```
Возвращает количество выведенных символов в случае успеха, отрицательное значение — в случае ошибки

```
int     vscanf(const char *restrict format, va_list arg);
                <stdarg.h>                                            с. 216
                <stdio.h>
```
Возвращает количество введенных элементов, `EOF` — в случае ошибки ввода или по достижении конца файла перед выполнением преобразования

```
int     vsnprintf(char *restrict buf, size_t n,
                  const char *restrict format, va_list arg);
                <stdarg.h>                                            с. 213
                <stdio.h>
```
Возвращает количество символов, сохраненных в массиве, если буфер имеет достаточный размер, отрицательное значение — в случае ошибки преобразования

```
int     vsprintf(char *restrict buf, const char *restrict format,
                 va_list arg);
```
<stdarg.h> с. 213
<stdio.h>
Возвращает количество символов, сохраненных в массиве, в случае успеха, отрицательное значение — в случае ошибки преобразования

```
int     vsscanf(const char *restrict buf, const char *restrict format,
                va_list arg);
```
<stdarg.h> с. 216
<stdio.h>
Возвращает количество введенных элементов, EOF — в случае ошибки ввода или по достижении конца файла перед выполнением преобразования

```
void    vsyslog(int priority, const char *format, va_list arg);
```
<syslog.h> с. 549
<stdarg.h>
Платформы: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10

```
pid_t   wait(int *statloc);
```
<sys/wait.h> с. 297
Возвращает идентификатор процесса в случае успеха, 0 или –1 — в случае ошибки

```
int     waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```
<sys/wait.h> с. 303
idtype: P_PID, P_PGID, P_ALL
options: WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED
Возвращает 0 в случае успеха, –1 — в случае ошибки
Платформы: Linux 3.2.0, Solaris 10

```
pid_t   waitpid(pid_t pid, int *statloc, int options);
```
<sys/wait.h> с. 297
options: WCONTINUED, WNOHANG, WUNTRACED
Возвращает идентификатор процесса в случае успеха, 0 или –1 — в случае ошибки

```
pid_t     wait3(int *statloc, int options, struct rusage *rusage);
                    <sys/types.h>                                          с. 304
                    <sys/wait.h>
                    <sys/time.h>
                    <sys/resource.h>
                    options: WNOHANG, WUNTRACED
                    Возвращает идентификатор процесса в случае успе-
                    ха, 0 или –1 — в случае ошибки
                    Платформы: FreeBSD 8.0, Linux 3.2.0,
                    Mac OS X 10.6.8, Solaris 10

pid_t     wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
                    <sys/types.h>                                          с. 304
                    <sys/wait.h>
                    <sys/time.h>
                    <sys/resource.h>
                    options: WNOHANG, WUNTRACED
                    Возвращает идентификатор процесса в случае успе-
                    ха, 0 или –1 — в случае ошибки
                    Платформы: FreeBSD 8.0, Linux 3.2.0,
                    Mac OS X 10.6.8, Solaris 10

ssize_t   write(int fd, const void *buf, size_t nbytes);
                    <unistd.h>                                             с. 115
                    Возвращает количество записанных байтов в случае
                    успеха, –1 — в случае ошибки

ssize_t   writev(int fd, const struct iovec *iov, int iovcnt);
                    <sys/uio.h>                                            с. 600
                    Возвращает количество записанных байтов в случае
                    успеха, –1 — в случае ошибки
```

B Различные исходные тексты

B.1. Наш заголовочный файл

Большинство программ в книге подключают заголовочный файл `apue.h`, содержимое которого приводится в листинге B.1. В нем определяются значения констант (таких, как `MAXLINE`) и прототипы наших собственных функций.

Большинство программ должны также подключать следующие заголовочные файлы: `<stdio.h>`, `<stdlib.h>` (где определен прототип функции `exit`) и `<unistd.h>` (содержащий прототипы всех стандартных функций UNIX). Поэтому наш заголовочный файл автоматически подключает эти системные заголовочные файлы, а также файл `<string.h>`. Это позволило также сократить размер листингов в книге.

Листинг B.1. Наш заголовочный файл apue.h

```
/*
 * Наш заголовочный файл, который подключается перед любыми
 * стандартными системными заголовочными файлами
 */
#ifndef _APUE_H
#define _APUE_H

#define _POSIX_C_SOURCE 200809L

#if defined(SOLARIS)         /* Solaris 10 */
#define _XOPEN_SOURCE 600
#else
#define _XOPEN_SOURCE 700
#endif

#include <sys/types.h>   /* некоторые системы требуют этого заголовка */
#include <sys/stat.h>
#include <sys/termios.h> /* структура winsize */

#if defined(MACOS) || !defined(TIOCGWINSZ)
```

```
#include <sys/ioctl.h>
#endif

#include <stdio.h>   /* для удобства */
#include <stdlib.h>  /* для удобства */
#include <stddef.h>  /* макрос offsetof */
#include <string.h>  /* для удобства */
#include <unistd.h>  /* для удобства */
#include <signal.h>  /* константа SIG_ERR */

#define MAXLINE 4096 /* максимальная длина строки */

/*
 * Права доступа по умолчанию к создаваемым файлам.
 */
#define FILE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Права доступа по умолчанию к создаваемым каталогам.
 */
#define DIR_MODE (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void Sigfunc(int); /* обработчики сигналов */

#define min(a,b) ((a) < (b) ? (a) : (b))
#define max(a,b) ((a) > (b) ? (a) : (b))

/*
 * Прототипы наших собственных функций.
 */
char     *path_alloc(size_t *);              /* листинг 2.3 */
long      open_max(void);                    /* листинг 2.4 */

int       set_cloexec(int);                  /* листинг 13.5 */
void      clr_fl(int, int);
void      set_fl(int, int);                  /* листинг 3.5 */

void      pr_exit(int);                      /* листинг 8.5 */

void      pr_mask(const char *);             /* листинг 10.10 */
Sigfunc  *signal_intr(int, Sigfunc *);       /* листинг 10.12 */

void      daemonize(const char *);           /* листинг 13.1 */

void      sleep_us(unsigned int);            /* упражнение 14.5 */
ssize_t   readn(int, void *, size_t);        /* листинг 14.9 */
ssize_t   writen(int, const void *, size_t); /* листинг 14.9 */

int       fd_pipe(int *);                    /* листинг 17.1 */
int       recv_fd(int, ssize_t (*func)(int,
                  const void *, size_t));    /* листинг 17.10 */
int       send_fd(int, int);                 /* листинг 17.9 */
int       send_err(int, int,
```

```
                const char *);            /* листинг 17.8 */
int       serv_listen(const char *);      /* листинг 17.5 */
int       serv_accept(int, uid_t *);      /* листинг 17.6 */
int       cli_conn(const char *);         /* листинг 17.7 */
int       buf_args(char *, int (*func)(int,
                char **));                /* листинг 17.19 */

int       tty_cbreak(int);                /* листинг 18.10 */
int       tty_raw(int);                   /* листинг 18.10 */
int       tty_reset(int);                 /* листинг 18.10 */
void      tty_atexit(void);               /* листинг 18.10 */
struct    termios *tty_termios(void);     /* листинг 18.10 */

int       ptym_open(char *, int);         /* листинг 19.1 */
int       ptys_open(char *);              /* листинг 19.1 */
#ifdef TIOCGWINSZ
pid_t     pty_fork(int *, char *, int, const struct termios *,
             const struct winsize *);     /* листинг 19.2 */
#endif

int       lock_reg(int, int, int, off_t, int, off_t); /* листинг 14.2 */
#define read_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_RDLCK, (offset), (whence), (len))
#define readw_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_RDLCK, (offset), (whence), (len))
#define write_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_WRLCK, (offset), (whence), (len))
#define writew_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLKW, F_WRLCK, (offset), (whence), (len))
#define un_lock(fd, offset, whence, len) \
            lock_reg((fd), F_SETLK, F_UNLCK, (offset), (whence), (len))

pid_t     lock_test(int, int, off_t, int, off_t); /* листинг 14.3 */

#define is_read_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_RDLCK, (offset), (whence), (len)) == 0)
#define is_write_lockable(fd, offset, whence, len) \
            (lock_test((fd), F_WRLCK, (offset), (whence), (len)) == 0)

void      err_msg(const char *, ...);     /* приложение В */
void      err_dump(const char *, ...) __attribute__((noreturn));
void      err_quit(const char *, ...) __attribute__((noreturn));
void      err_cont(int, const char *, ...);
void      err_exit(int, const char *, ...) __attribute__((noreturn));
void      err_ret(const char *, ...);
void      err_sys(const char *, ...) __attribute__((noreturn));

void      log_msg(const char *, ...);     /* приложение В */
void      log_open(const char *, int, int);
void      log_quit(const char *, ...) __attribute__((noreturn));
void      log_ret(const char *, ...);
void      log_sys(const char *, ...) __attribute__((noreturn));
void      log_exit(int, const char *, ...) __attribute__((noreturn));
```

```
void      TELL_WAIT(void); /* предок/потомок из раздела 8.9 */
void      TELL_PARENT(pid_t);
void      TELL_CHILD(pid_t);
void      WAIT_PARENT(void);
void      WAIT_CHILD(void);
#endif /* _APUE_H */
```

Наш заголовочный файл подключается первым, перед всеми обычными системными заголовочными файлами, потому что это позволяет нам дать определения, которые могут потребоваться другим заголовочным файлам, установить порядок подключения заголовочных файлов, а также переопределить некоторые значения, чтобы сгладить и скрыть различия между системами.

B.2. Стандартные процедуры обработки ошибок

В большинстве наших примеров используются два набора функций обработки ошибочных ситуаций. Один включает функции с именами, начинающимися с префикса `err_`, они выводят сообщения в стандартный поток вывода сообщений об ошибках. Другой включает функции с именами, начинающимися с префикса `log_`, они предназначены для использования в процессах-демонах (глава 13), которые, как правило, не имеют управляющего терминала.

Эти наборы функций позволяют обрабатывать ошибочные ситуации всего одной строчкой в программе, например:

```
if (условие ошибки)
        err_dump(формат в стиле printf с любым количеством аргументов);
```

вместо

```
if (условие ошибки) {
        char buf[200];

        sprintf(buf, формат в стиле printf с любым количеством
                     аргументов);
        perror(buf);
        abort();
}
```

Наши функции обработки ошибок используют возможность передачи списка аргументов переменной длины, которая определяется стандартом ISO C. Дополнительные сведения вы найдете в разделе 7.3 [Kernighan and Ritchie, 1988]. Важно понимать, что функциональная возможность передачи списка аргументов переменной длины из стандарта ISO C отличается от функциональности `varargs`, которая предоставлялась ранними версиями системы (такими, как SVR3 и 4.3BSD). Имена макроопределений остались теми же, но аргументы некоторых из них изменились.

В табл. B.1 показаны различия между разными функциями обработки ошибок.

Таблица В.1. Наши стандартные функции обработки ошибок

Функция	Добавляет строку от strerror?	Аргументы для strerror	Завершает процесс?
err_dump	Да	errno	abort();
err_exit	Да	Явный параметр	exit(1);
err_msg	Нет		return;
err_quit	Нет		exit(1);
err_ret	Да	errno	return;
err_sys	Да	errno	exit(1);
log_msg	Нет		return;
log_quit	Нет		exit(2);
log_ret	Да	errno	return;
log_sys	Да	errno	exit(2);

В листинге В.2 приводятся функции обработки ошибок, которые выводят сообщения в стандартный поток вывода сообщений об ошибках.

Листинг В.2. Функции обработки ошибок, которые выводят сообщения в стандартное устройство вывода сообщений об ошибках

```
#include "apue.h"
#include <errno.h>  /* определение переменной errno */
#include <stdarg.h> /* список аргументов переменной длины ISO C */

static void err_doit(int, int, const char *, va_list);

/*
 * Обрабатывает нефатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
err_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_sys(const char *fmt, ...)
{
```

```
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    exit(1);
}

/*
 * Обрабатывает нефатальные ошибки, не связанные с системными вызовами.
 * Код ошибки передается в аргументе.
 * Выводит сообщение и возвращает управление.
 */
void
err_cont(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, не связанные с системными вызовами.
 * Код ошибки передается в аргументе.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_exit(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, error, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение, создает файл core и завершает работу процесса.
 */
void
err_dump(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort(); /* записать дамп памяти в файл и завершить процесс */
    exit(1); /* этот вызов никогда не должен выполниться */
}
```

```
/*
 * Обрабатывает нефатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
err_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
err_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(0, 0, fmt, ap);
    va_end(ap);
    exit(1);
}

/*
 * Выводит сообщение и возвращает управление в вызывающую функцию.
 * Вызывающая функция определяет значение флага "errnoflag".
 */
static void
err_doit(int errnoflag, int error, const char *fmt, va_list ap)
{
    char    buf[MAXLINE];

    vsnprintf(buf, MAXLINE-1, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf)-1, ": %s",
                 strerror(error));
    strcat(buf, "\n");
    fflush(stdout); /* в случае, когда stdout и stderr - одно и то же устройство */
    fputs(buf, stderr);
    fflush(NULL);   /* сбрасывает все выходные потоки */
}
```

В листинге В.3 приводятся исходные тексты функций семейства `log_XXX`. Они требуют, чтобы в вызывающем процессе была определена глобальная переменная `log_to_stderr`. Эта переменная должна содержать ненулевое значение, если процесс выполняется не как демон. В этом случае сообщения выводятся в стандарт-

ный поток вывода сообщений об ошибках. Если `log_to_stderr` содержит 0, для вывода сообщений используется функция `syslog` (раздел 13.4).

Листинг В.3. Функции обработки ошибок для демонов

```
/*
 * Процедуры обработки ошибок для программ, которые могут работать как демоны.
 */
#include "apue.h"
#include <errno.h>  /* определение переменной errno */
#include <stdarg.h> /* список аргументов переменной длины ISO C */
#include <syslog.h>

static void log_doit(int, int, int, const char *, va_list ap);

/*
 * В вызывающем процессе должна быть определена и установлена эта переменная:
 * ненулевое значение - для интерактивных программ, нулевое – для демонов
*/
extern int log_to_stderr;

/*
 * Инициализировать syslog(), если процесс работает в режиме демона.
 */
void
log_open(const char *ident, int option, int facility)
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Обрабатывает нефатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение, соответствующее содержимому переменной errno,
 * и возвращает управление.
 */
void
log_ret(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, errno, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_sys(const char *fmt, ...)
{
```

```
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, errno, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Обрабатывает нефатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и возвращает управление.
 */
void
log_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, 0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Обрабатывает фатальные ошибки, не связанные с системными вызовами.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_quit(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, 0, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}

/*
 * Обрабатывает фатальные ошибки, связанные с системными вызовами.
 * Номер ошибки передается в параметре.
 * Выводит сообщение и завершает работу процесса.
 */
void
log_exit(int error, const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(1, error, LOG_ERR, fmt, ap);
    va_end(ap);
    exit(2);
}
```

```
/*
 * Выводит сообщение и возвращает управление в вызывающую функцию.
 * Вызывающая функция должна определить значения аргументов
 * "errnoflag" и "priority".
 */
static void
log_doit(int errnoflag, int error, int priority, const char *fmt,
         va_list ap)
{
    char    buf[MAXLINE];

    vsnprintf(buf, MAXLINE-1, fmt, ap);
    if (errnoflag)
        snprintf(buf+strlen(buf), MAXLINE-strlen(buf)-1, ": %s",
                 strerror(error));
    strcat(buf, "\n");
    if (log_to_stderr) {
        fflush(stdout);
        fputs(buf, stderr);
        fflush(stderr);
    } else {
        syslog(priority, "%s", buf);
    }
}
```

C Варианты решения некоторых упражнений

Глава 1

1.1 Для решения этого упражнения используем следующие два аргумента команды `ls`(1): `-i` — для вывода номеров индексных узлов файлов и каталогов (более подробно об индексных узлах рассказывается в разделе 4.14), и `-d` — для вывода информации только о каталогах.

В результате получим следующее:

```
$ ls -ldi /etc/. /etc/..        ключ –i заставляет выводить номера индексных
                                узлов
 162561 drwxr-xr-x  66 root    4096 Feb 5 03:59 /etc/./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /etc/../
$ ls -ldi /. /..                оба каталога . и .. имеют один и тот же номер
                                inode – 2
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /./
drwxr-xr-x  19 root    4096 Jan 15 07:25 /../
```

1.2 UNIX является многозадачной системой. Следовательно, между запусками нашей программы были запущены какие-то другие процессы.

1.3 Аргумент *msg* функции `perror` является указателем, поэтому `perror` может изменить содержимое строки, на которую указывает аргумент *msg*. Однако атрибут `const` говорит о том, что `perror` не изменяет строку, на которую ссылается указатель. С другой стороны, аргумент с кодом ошибки в функции `strerror` является целым числом, а так как он передается по значению, функция `strerror` не сможет изменить его, даже если захочет. (Если вы не совсем понимаете, как передаются и обрабатываются аргументы функций в языке C, обратитесь к разделу 5.2 [Kernighan and Ritchie, 1988].)

1.4 В 2038 году. Проблема может быть решена за счет увеличения размера типа `time_t` до 64 бит. Если это будет сделано для корректной работы всех приложений, использующих 32-разрядное представление, их необходимо будет

пересобрать. Но на самом деле проблема гораздо глубже. Некоторые файловые системы и носители, предназначенные для хранения резервных копий, используют 32-разрядное представление времени. Их также придется обновить, но при этом сохранить совместимость с устаревшим форматом.

1.5 Примерно 248 дней.

Глава 2

2.1 В FreeBSD используется следующий способ. Элементарные типы данных, которые могут быть объявлены в нескольких заголовочных файлах, определяются в файле `<machine/_types.h>`. Например:

```
#ifndef _MACHINE__TYPES_H_
#define _MACHINE__TYPES_H_

typedef int          __int32_t;
typedef unsigned int __uint32_t;
...
typedef __uint32_t   __size_t;
...
#endif /* _MACHINE__TYPES_H_ */
```

В каждом заголовочном файле, где может определяться элементарный системный тип данных `size_t`, можно использовать такую последовательность:

```
#ifndef _SIZE_T_DECLARED
typedef __size_t      size_t;
#define _SIZE_T_DECLARED
#endif
```

При таком подходе инструкция `typedef` для типа `size_t` будет выполнена всего один раз.

2.3 Если значение константы `OPEN_MAX` не определено или чрезвычайно велико (то есть равно `LONG_MAX`), для получения максимально возможного количества открытых файловых дескрипторов для процесса можно использовать функцию `getrlimit`. Учитывая, что предел для процесса может быть изменен, мы не можем повторно использовать значение, полученное в результате предыдущего вызова (так как он мог измениться). Решение приводится в листинге С.1.

Листинг С.1. Альтернативный способ определения максимально возможного количества файловых дескрипторов

```
#include "apue.h"
#include <limits.h>
#include <sys/resource.h>

#define OPEN_MAX_GUESS 256
```

```
long
open_max(void)
{
    long openmax;
    struct rlimit rl;

    if ((openmax = sysconf(_SC_OPEN_MAX)) < 0 ||
                  openmax == LONG_MAX) {
        if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
            err_sys("невозможно получить значение предела");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}
```

Глава 3

3.1 Все дисковые операции ввода/вывода выполняются с использованием буферов блоков, расположенных в пространстве ядра (которые также известны как буферный кэш ядра). Исключением являются операции ввода/вывода с неструктурированными дисковыми устройствами, которые мы не рассматривали. (Некоторые системы также поддерживают *непосредственные операции ввода/вывода* с дисковыми устройствами, чтобы дать приложениям возможность производить ввод/вывод в обход буферов в ядре, но мы не рассматривали такую возможность.) Работа буферного кэша описана в главе 3 [Bach, 1986]. Поскольку читаемые или записываемые данные буферизуются ядром, термин *небуферизованный ввод/вывод* скорее означает отсутствие автоматической буферизации в пользовательском процессе при использовании функций `read` и `write`. Каждая из этих функций обращается к единственному системному вызову.

3.3 Каждый вызов функции `open` создает новую запись в таблице файлов. Но так как обе операции открывают один и тот же файл, обе записи в таблице файлов будут указывать на одну и ту же запись в таблице виртуальных узлов. Вызов `dup` создаст еще одну ссылку на существующую запись в таблице файлов. Диаграмма, соответствующая данной ситуации, показана на рис. С.1. Функция `fcntl` с аргументами `F_SETFD` и `fd1` воздействует только на флаги дескриптора `fd1`. Но с аргументами `F_SETFL` и `fd1` она будет воздействовать на запись в таблице файлов и тем самым на оба дескриптора — `fd1` и `fd2`.

3.4 Для `fd` со значением 1 вызов `dup2(fd, 1)` вернет 1, оставив открытым дескриптор 1. (Вспомните обсуждение из раздела 3.12.) После выполнения трех вызовов `dup2` все три дескриптора будут ссылаться на одну и ту же запись в таблице файлов. Ни один из дескрипторов не будет закрыт. Однако для `fd` со значением 3, после третьего вызова `dup2`, на одну и ту же запись

в таблице файлов будут ссылаться уже четыре дескриптора. В этом случае нужно закрыть дескриптор с номером 3.

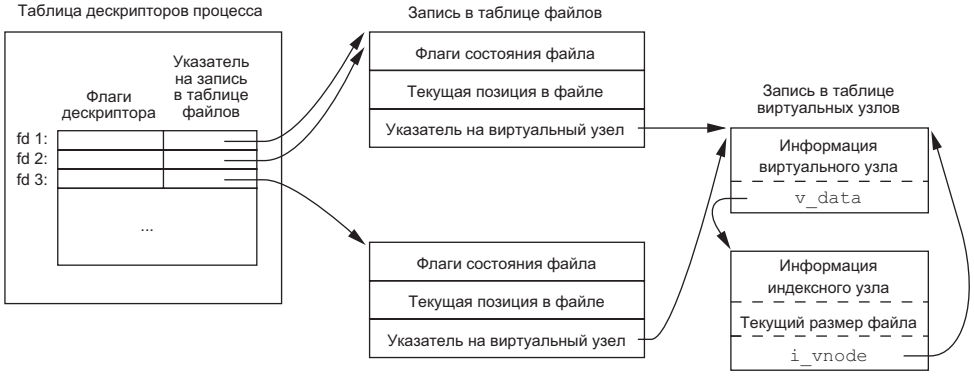

Рис. С.1. Результат работы функций dup и open

3.5 Поскольку командные оболочки обрабатывают аргументы командной строки слева направо, команда

```
./a.out > outfile 2>&1
```

сначала перенаправит стандартный вывод в файл `outfile`, а затем продублирует его на дескриптор с номером 2 (стандартный вывод сообщений об ошибках). В результате все, что будет выводиться в стандартный вывод и в стандартный вывод сообщений об ошибках, попадет в один и тот же файл. Дескрипторы 1 и 2 будут ссылаться на одну и ту же запись в таблице файлов. Однако команда

```
./a.out 2>&1 > outfile
```

сначала вызовет функцию `dup`, и в результате дескриптор с номером 2 будет ссылаться на терминал (предполагается, что команда запущена в интерактивном режиме). А затем стандартный вывод будет перенаправлен в файл `outfile`. В результате дескриптор с номером 1 будет ссылаться на запись в таблице файлов, которая соответствует файлу `outfile`, а дескриптор с номером 2 — на запись, которая соответствует терминалу.

3.6 Вы по-прежнему сможете использовать функцию `lseek` и читать данные из произвольного места в файле, но вызов функции `write` автоматически произведет переход в конец файла перед записью данных. В этом случае вы не сможете записать данные в произвольное место в файле.

Глава 4

4.1 Функция `stat` всегда пытается следовать по символическим ссылкам (табл. 4.9), поэтому программа никогда не выведет строку «символическая

ссылка». Для приведенного примера, где файл `/dev/cdrom` является символической ссылкой на файл `/dev/sr0`, функция `stat` укажет, что файл `/dev/cdrom` является специальным файлом блочного устройства, а не символической ссылкой. Если символическая ссылка ссылается на несуществующий файл, функция `stat` вернет признак ошибки.

4.2 Все биты прав доступа окажутся сброшены:

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
----------  1 sar     29 Feb  5 14:06 temp.foo
```

4.3 Следующий пример показывает, что произойдет, если сбросить бит user-read:

```
$ date > foo
$ chmod u-r foo                 сбросить бит user-read
$ ls -l foo                     проверить права доступа к файлу
--w-r--r--  1 sar     29 Feb  5 14:21 foo
$ cat foo                       и попытаться прочитать его
cat: foo: Permission denied
```

4.4 Если попытаться с помощью функции `open` или `creat` создать файл, который уже существует, права доступа к файлу не изменятся. Мы можем убедиться в этом, запустив программу из листинга 4.3:

```
$ rm foo bar                    удалить файлы, если они существуют
$ date > foo                    создать их и наполнить данными
$ date > bar
$ chmod a-r foo bar             сбросить биты права на чтение для всех
$ ls -l foo bar                 проверить права доступа
--w-------  1 sar    29 Feb  5 14:25 bar
--w-------  1 sar    29 Feb  5 14:25 foo
$ ./a.out                       запустить программу из листинга 4.3
$ ls -l foo bar                 проверить права доступа и размеры файлов
--w-------  1 sar     0 Feb  5 14:26 bar
--w-------  1 sar     0 Feb  5 14:26 foo
```

Обратите внимание, что права доступа не изменились, но файлы были усечены.

4.5 Размер каталога никогда не может быть равен 0, поскольку файлы каталогов содержат по крайней мере две записи — ссылки на каталоги `.` и `..`. Размер файла символической ссылки определяется количеством символов в имени файла и пути к нему, а имя файла всегда содержит хотя бы один символ.

4.7 При создании файла `core` ядро по умолчанию использует определенные значения битов прав доступа. В данном примере это `rw-r--r--`. Это значение может модифицироваться, а может не модифицироваться значением `umask`. Командная оболочка также определяет значения битов прав доступа по умолчанию, которые устанавливаются для файлов, созданных в резуль-

тате перенаправления. В данном примере это `rw-rw-rw-`, а это значение всегда модифицируется текущим значением `umask`. В данном примере значением `umask` было число 02.

4.8 Мы не можем воспользоваться командой `du`, так как она требует указать имя файла, например

```
du tempfile
```

или имя каталога:

```
du .
```

Но после возврата из функции `unlink` запись в каталоге для файла `tempfile` исчезает. Команда `du .` не смогла бы показать, что содержимое файла `tempfile` по-прежнему продолжает занимать дисковое пространство. В этом примере мы должны использовать команду `df`, чтобы увидеть фактический объем свободного дискового пространства.

4.9 При удалении ссылки, которая не является последней, сам файл не удаляется. В этом случае обновляется время последнего изменения файла. Но если удаляется последняя ссылка на файл, обновление времени последнего изменения теряет всякий смысл, поскольку вся информация о файле (индексный узел) удаляется вместе с файлом.

4.10 Мы рекурсивно вызываем функцию `dopath` после открытия каталога функцией `opendir`. Предположим, что `opendir` использует единственный дескриптор — в этом случае каждый раз, спускаясь на один уровень вглубь иерархии дерева каталогов, мы используем другой дескриптор. (Если исходить из предположения, что дескрипторы не закрываются, пока не будет закончен обзор дерева каталогов и не будет вызвана функция `closedir`.) Это ограничивает глубину дерева каталогов, на которую мы можем погрузиться, максимальным количеством одновременно открытых дескрипторов. Обратите внимание: в расширениях XSI стандарта Single UNIX Specification определено, что функция `nftw` позволяет вызывающему процессу задать максимальное количество используемых дескрипторов, допуская закрытие и повторное использование дескрипторов.

4.12 Функция `chroot` используется в Интернете на серверах FTP для повышения безопасности. Пользователи, не имеющие учетных записей в системе (так называемые *анонимные пользователи FTP*), попадают в отдельный каталог, и этот каталог делается корневым с помощью функции `chroot`. Это предотвращает возможность доступа к файлам, расположенным за пределами нового корневого каталога.

Кроме того, функция `chroot` может использоваться для создания копии дерева каталогов на новом месте, чтобы затем изменять эту новую копию, не опасаясь внести изменения в оригинальную файловую систему. Это полезно, например, для тестирования результатов установки новых программных пакетов.

Только суперпользователь может вызвать функцию `chroot`, и после изменения корневого каталога процесс и все его потомки никогда не смогут вернуться к первоначальному корню файловой системы.

4.13 Прежде всего необходимо вызвать функцию `stat`, чтобы получить три значения времени для файла, затем вызвать `utimes`, чтобы изменить требуемое значение. Значение, которое не должно изменяться в результате вызова `utimes`, должно соответствовать значению, полученному от функции `stat`.

4.14 Команда `finger`(1) использует функцию `stat` для определения атрибутов времени почтового ящика. Время последнего изменения соответствует времени прибытия последнего электронного письма, а время последнего обращения — времени, когда в последний раз была прочитана почта.

4.15 Обе утилиты, `cpio` и `tar`, сохраняют в архиве только время последнего изменения (`st_mtime`). Время последнего обращения не сохраняется, поскольку его значение соответствует времени создания архива, так как для этого архиватор должен прочитать содержимое файла. Ключ `-a` команды `cpio` позволяет переустановить время последнего обращения для каждого прочитанного файла. В результате создание архива не влечет изменения времени последнего обращения. (Однако переустановка времени последнего обращения к файлу приводит к изменению времени последнего изменения статуса.) Время последнего изменения статуса не сохраняется в архиве, так как при извлечении файла из архива нет возможности запросить его значение, даже если бы оно было сохранено в архиве. (Функция `utimes` и родственные ей `futimens` и `utimensat` могут изменять только время последнего изменения файла и время последнего обращения к файлу.)

Когда архиватор `tar` извлекает файлы из архива, он по умолчанию восстанавливает время последнего изменения извлекаемых файлов. С помощью ключа `m` можно указать утилите `tar`, что она не должна восстанавливать время последнего изменения файла, тогда в качестве времени последнего изменения будет использоваться время извлечения из архива. При использовании архиватора `tar` время последнего обращения к файлу после его извлечения из архива в любом случае будет установлено равным времени извлечения.

Архиватор `cpio`, напротив, в качестве времени последнего изменения и времени последнего обращения устанавливает время извлечения из архива. По умолчанию он не пытается восстановить прежнее время последнего изменения файла, сохраненное в архиве. При использовании архиватора `cpio` для восстановления значений времени последнего обращения и времени последнего изменения, сохраненных в архиве, следует использовать ключ `-m`.

4.16 Ядро изначально не имеет ограничений на глубину вложенности каталогов. Но большинство команд завершаются ошибкой, если полные имена файлов или каталогов превышают длину `PATH_MAX`. Программа в листинге С.2 создает дерево каталогов, состоящее из 1000 уровней вложенности, на каждом уровне каталог имеет имя длиной 45 символов. Можно создать эту структуру на любой платформе, однако ни на одной из платформ нельзя получить

абсолютное полное имя каталога на тысячном уровне с помощью функции `getcwd`. В Mac OS X 10.6.8 мы никогда не сможем получить полное имя самого последнего каталога в таком длинном пути. В FreeBSD 8.0, Linux 3.2.0 и Solaris 10 программа сможет получить полное имя последнего каталога, но нам придется много раз вызвать функцию `realloc`, чтобы разместить буфер достаточно большого размера. Запуск этой программы в Linux 3.2.0 дал следующие результаты:

```
$ ./a.out
ошибка вызова функции getcwd, размер = 4096: Numerical result too large
ошибка вызова функции getcwd, размер = 4196: Numerical result too large
...                              еще 418 строк
ошибка вызова функции getcwd, размер = 45896: Numerical result too large
ошибка вызова функции getcwd, размер = 45996: Numerical result too large
длина = 46004
                                 здесь было выведено имя длиной 46 004 байт
```

Мы не сможем заархивировать это дерево каталогов с помощью архиватора `cpio`. Он выведет сообщение о слишком длинном имени файла. `cpio` не сможет заархивировать этот каталог ни на одной из четырех платформ. Напротив, `tar` сможет заархивировать этот каталог в FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8. Но в Linux 3.2.0 не получится извлечь это дерево каталогов из архива.

Листинг С.2. Создание дерева каталогов с глубокой вложенностью

```
#include "apue.h"
#include <fcntl.h>

#define DEPTH       1000     /* глубина вложенности */
#define STARTDIR    "/tmp"
#define NAME        "alonglonglonglonglonglonglonglonglongname"
#define MAXSZ       (10*8192)

int
main(void)
{
    int     i, size;
    size_t  size;
    char   *path;

    if (chdir(STARTDIR) < 0)
        err_sys("ошибка вызова функции chdir");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("ошибка вызова функции mkdir, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("ошибка вызова функции chdir, i = %d", i);
    }

    if (creat("afile", FILE_MODE) < 0)
```

```
            err_sys("ошибка вызова функции creat");

        /*
         * Дерево каталогов с большой глубиной вложенности создано,
         * в каталоге создан файл. Теперь попробуем получить его полное имя.
         */
        path = path_alloc(&size);
        for ( ; ; ) {
            if (getcwd(path, size) != NULL) {
                break;
            } else {
                err_ret("ошибка вызова функции getcwd, размер = %ld", (long)
                        size);
                size += 100;
                if (size > MAXSZ)
                    err_quit("превышено наше ограничение");
                if ((path = realloc(path, size)) == NULL)
                    err_sys("ошибка вызова функции realloc");
            }
        }
        printf("длина = %ld\n%s\n", (long)strlen(path), path);

        exit(0);
    }
```

4.17 Для каталога `/dev` все биты права на запись сброшены, что не позволит обычному пользователю удалять файлы из каталога. Это означает, что вызов функции `unlink` будет завершаться неудачей.

Глава 5

5.2 Функция `fgets` будет читать символы, пока не встретится символ перевода строки или пока буфер не заполнится (с учетом места, которое необходимо оставить для завершающего нулевого символа). Функция `fputs` будет выводить данные из буфера, пока не встретит завершающий нулевой символ — она не обращает внимания на символы перевода строки, которые могут находиться в буфере. То есть если значение `MAXLINE` будет слишком маленьким, обе функции по-прежнему будут работать, просто они будут вызываться намного чаще, чем при использовании буфера большого размера.

Если бы любая из этих функций удаляла или добавляла символ перевода строки (как это делают функции `gets` и `puts`), нам пришлось бы предусматривать размещение буферов достаточно большого объема, чтобы вместить самую длинную строку.

5.3 Вызов

```
printf("");
```

вернет 0, потому что не выводит ни одного символа.

5.4 Это достаточно распространенная ошибка. Возвращаемое значение функций `getc` и `getchar` имеет тип `int`, а не `char`. Зачастую константа `EOF` определена как –1, и поэтому, если в системе тип `char` имеет знак, этот код будет работать нормально. Но если в системе тип `char` не имеет знака, возвращаемое значение `EOF`, полученное от `getchar`, будет сохранено в переменной с беззнаковым типом `char` и перестанет быть равным –1, вследствие чего цикл никогда не закончится. На всех четырех платформах, описываемых в данной книге, тип `char` имеет знак, поэтому данный пример будет корректно работать на всех этих платформах.

5.5 Вызывать функцию `fsync` после каждого вызова `fflush`. Аргумент функции `fsync` можно получить вызовом функции `fileno`. Вызов `fsync` без обращения к `fflush` может не дать ожидаемого результата, если данные все еще находятся во внутренних буферах приложения.

5.6 Когда программа работает в интерактивном режиме, стандартные потоки ввода и вывода буферизуются построчно. Когда вызывается функция `fgets`, содержимое потока стандартного вывода сбрасывается автоматически.

5.7 Реализация `fmemopen` для BSD-систем представлена в листинге С.3.

Листинг С.3. Реализация функции fmemopen для BSDсистем

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/*
 * Внутренняя структура для слежения за потоком ввода/вывода в памяти
 */
struct memstream
{
    char   *buf;    /* буфер в памяти */
    size_t  rsize;  /* фактический размер буфера */
    size_t  vsize;  /* виртуальный размер буфера */
    size_t  curpos; /* текущая позиция в буфере */
    int     flags;  /* см. ниже */
};

/* флаги */
#define MS_READ      0x01 /* открыть для чтения */
#define MS_WRITE     0x02 /* открыть для записи */
#define MS_APPEND    0x04 /* открыть для добавления в конец */
#define MS_TRUNCATE  0x08 /* усечь при открытии */
#define MS_MYBUF     0x10 /* освободить буфер при закрытии */

#ifndef MIN
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif

static int    mstream_read(void *, char *, int);
static int    mstream_write(void *, const char *, int);
```

```
static fpos_t mstream_seek(void *, fpos_t, int);
static int    mstream_close(void *);
static int    type_to_flags(const char *__restrict type);
static off_t  find_end(char *buf, size_t len);

FILE *
fmemopen(void *__restrict buf, size_t size,
         const char *__restrict type)
{
    struct memstream *ms;
    FILE *fp;

    if (size == 0) {
        errno = EINVAL;
        return(NULL);
    }
    if ((ms = malloc(sizeof(struct memstream))) == NULL) {
        errno = ENOMEM;
        return(NULL);
    }
    if ((ms->flags = type_to_flags(type)) == 0) {
        errno = EINVAL;
        free(ms);
        return(NULL);
    }
    if (buf == NULL) {
        if ((ms->flags & (MS_READ|MS_WRITE)) !=
          (MS_READ|MS_WRITE)) {
            errno = EINVAL;
            free(ms);
            return(NULL);
        }
        if ((ms->buf = malloc(size)) == NULL) {
            errno = ENOMEM;
            free(ms);
            return(NULL);
        }
        ms->rsize = size;
        ms->flags |= MS_MYBUF;
        ms->curpos = 0;
    } else {
        ms->buf = buf;
        ms->rsize = size;
        if (ms->flags & MS_APPEND)
            ms->curpos = find_end(ms->buf, ms->rsize);
        else
            ms->curpos = 0;
    }
    if (ms->flags & MS_APPEND) {          /* режим "a" */
        ms->vsize = ms->curpos;
    } else if (ms->flags & MS_TRUNCATE) { /* режим "w" */
        ms->vsize = 0;
    } else {                              /* режим "r" */
```

```
        ms->vsize = size;
    }
    fp = funopen(ms, mstream_read, mstream_write,
                 mstream_seek, mstream_close);
    if (fp == NULL) {
        if (ms->flags & MS_MYBUF)
            free(ms->buf);
        free(ms);
    }
    return(fp);
}

static int
type_to_flags(const char *__restrict type)
{
    const char *cp;
    int flags = 0;

    for (cp = type; *cp != 0; cp++) {
        switch (*cp) {
        case 'r':
            if (flags != 0)
                return(0); /* ошибка */
        flags |= MS_READ;
        break;

        case 'w':
            if (flags != 0)
                return(0); /* ошибка */
            flags |= MS_WRITE|MS_TRUNCATE;
            break;

        case 'a':
            if (flags != 0)
                return(0); /* ошибка */
            flags |= MS_APPEND;
            break;

        case '+':
            if (flags == 0)
                return(0); /* ошибка */
            flags |= MS_READ|MS_WRITE;
            break;

        case 'b':
            if (flags == 0)
                return(0); /* ошибка */
            break;

        default:
            return(0); /* ошибка */
        }
    }
```

```
    return(flags);
}

static off_t
find_end(char *buf, size_t len)
{
    off_t off = 0;

    while (off < len) {
        if (buf[off] == 0)
            break;
        off++;
    }
    return(off);
}

static int
mstream_read(void *cookie, char *buf, int len)
{
    int nr;
    struct memstream *ms = cookie;

    if (!(ms->flags & MS_READ)) {
        errno = EBADF;
        return(-1);
    }
    if (ms->curpos >= ms->vsize)
        return(0);

    /* прочитать можно только от текущей позиции до vsize */
    nr = MIN(len, ms->vsize - ms->curpos);
    memcpy(buf, ms->buf + ms->curpos, nr);
    ms->curpos += nr;
    return(nr);
}

static int
mstream_write(void *cookie, const char *buf, int len)
{
    int nw, off;
    struct memstream *ms = cookie;

    if (!(ms->flags & (MS_APPEND|MS_WRITE))) {
        errno = EBADF;
        return(-1);
    }
    if (ms->flags & MS_APPEND)
        off = ms->vsize;
    else
        off = ms->curpos;
    nw = MIN(len, ms->rsize - off);
    memcpy(ms->buf + off, buf, nw);
    ms->curpos = off + nw;
```

```
    if (ms->curpos > ms->vsize) {
        ms->vsize = ms->curpos;
        if (((ms->flags & (MS_READ|MS_WRITE)) ==
          (MS_READ|MS_WRITE)) && (ms->vsize < ms->rsize))
            *(ms->buf + ms->vsize) = 0;
    }
    if ((ms->flags & (MS_WRITE|MS_APPEND)) &&
      !(ms->flags & MS_READ)) {
        if (ms->curpos < ms->rsize)
            *(ms->buf + ms->curpos) = 0;
        else
            *(ms->buf + ms->rsize - 1) = 0;
    }
    return(nw);
}

static fpos_t
mstream_seek(void *cookie, fpos_t pos, int whence)
{
    int off;
    struct memstream *ms = cookie;

    switch (whence) {
    case SEEK_SET:
        off = pos;
        break;
    case SEEK_END:
        off = ms->vsize + pos;
        break;
    case SEEK_CUR:
        off = ms->curpos + pos;
        break;
    }
    if (off < 0 || off > ms->vsize) {
        errno = EINVAL;
        return -1;
    }
    ms->curpos = off;
    return(off);
}

static int
mstream_close(void *cookie)
{
    struct memstream *ms = cookie;

    if (ms->flags & MS_MYBUF)
        free(ms->buf);
    free(ms);
    return(0);
}
```

Глава 6

6.1 Функции доступа к теневому файлу паролей в Linux и Solaris обсуждались в разделе 6.3. Мы не можем для сравнения с зашифрованным паролем использовать значение, возвращаемое в поле `pw_passwd` функциями, описанными в разделе 6.2, поскольку это поле не содержит зашифрованного пароля. Чтобы получить зашифрованный пароль, нужно отыскать требуемую учетную запись в теневом файле паролей и извлечь из нее зашифрованный пароль.

В FreeBSD и Mac OS X автоматически используется теневой файл паролей. В FreeBSD 8.0, в структуре `passwd`, возвращаемой функциями `getpwnam` и `getpwuid`, поле `pw_passwd` содержит зашифрованный пароль, но только если вызывающий процесс имеет эффективный идентификатор пользователя 0. В Mac OS X 10.6.8 зашифрованный пароль нельзя получить с помощью этих функций.

6.2 Программа из листинга С.4 выводит зашифрованный пароль в Linux 3.2.0 и Solaris 10. Если эту программу запустит обычный пользователь, вызов `getspnam` завершится неудачей с кодом ошибки `EACCES`.

Листинг С.4. Вывод зашифрованного пароля в ОС Linux и Solaris

```
#include "apue.h"
#include <shadow.h>

int
main(void) /* версия для Linux/Solaris */
{
    struct spwd *ptr;

    if ((ptr = getspnam("sar")) == NULL)
        err_sys("ошибка вызова функции getspnam");
    printf("sp_pwdp = %s\n", ptr->sp_pwdp == NULL ||
           ptr->sp_pwdp[0] == 0 ? "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

В листинге С.5 приводится исходный текст программы, которая выведет зашифрованный пароль в FreeBSD 8.0, если запустить ее с привилегиями суперпользователя. В иных случаях в поле `pw_passwd` возвращается символ «звездочки». В Mac OS X 10.6.8 зашифрованный пароль выводится в любом случае, независимо от привилегий, с которыми запущена программа.

Листинг С.5. Вывод зашифрованного пароля в ОС FreeBSD и Mac OS X

```
#include "apue.h"
#include <pwd.h>

int
main(void) /* версия для FreeBSD/Mac OS X */
{
    struct passwd *ptr;
```

```
    if ((ptr = getpwnam("sar")) == NULL)
        err_sys("ошибка вызова функции getpwnam");
    printf("pw_passwd = %s\n", ptr->pw_passwd == NULL ||
           ptr->pw_passwd[0] == 0 ? "(null)" : ptr->pw_passwd);
    exit(0);
}
```

6.5 Программа из листинга С.6 выводит текущее время и дату в формате утилиты `date`.

Листинг С.6. Вывод текущего времени и даты в формате утилиты date

```
#include "apue.h"
#include <time.h>

int
main(void)
{
    time_t      caltime;
    struct tm  *tm;
    char        line[MAXLINE];

    if ((caltime = time(NULL)) == -1)
        err_sys("ошибка вызова функции time");
    if ((tm = localtime(&caltime)) == NULL)
        err_sys("ошибка вызова функции localtime");
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("ошибка вызова функции strftime");
    fputs(line, stdout);
    exit(0);
}
```

Запустив эту программу, мы получили следующее:

```
$ ./a.out                       часовой пояс автора по умолчанию US/Eastern
Wed Jul 25 22:58:32 EDT 2012
$ TZ=US/Mountain ./a.out        U.S. часовой пояс штата Монтана
Wed Jul 25 20:58:32 MDT 2012
$ TZ=Japan ./a.out              Япония
Thu Jul 26 11:58:32 JST 2012
```

Глава 7

7.1 Похоже, что возвращаемое значение функции `printf` (количество выведенных символов) стало возвращаемым значением функции `main`. Чтобы проверить это предположение, измените длину выводимой строки и посмотрите, как изменится возвращаемое значение. Такое поведение наблюдается не во всех системах. Отметьте также, что если разрешить применение расширений ISO C в компиляторе `gcc`, возвращаемое значение всегда будет равно 0, как того требует стандарт.

7.2 Когда программа работает в интерактивном режиме, стандартный вывод обычно буферизуется построчно так, что фактический вывод происходит

только при выводе символа перевода строки. Но если стандартный поток вывода перенаправлен в файл, ему, скорее всего, будет назначен режим полной буферизации, и фактический вывод не будет производиться до освобождения ресурсов стандартной библиотеки ввода/вывода.

7.3 В большинстве версий UNIX это невозможно. Копии `argc` и `argv` не сохраняются в глобальных переменных, как, например, `environ`.

7.4 Это дает возможность аварийно завершить процесс при попытке обратиться к памяти по пустому указателю, что является достаточно распространенной ошибкой при программировании на языке С.

7.5 Вот эти определения:

```
typedef void    Exitfunc(void);
int atexit(Exitfunc *func);
```

7.6 Функция `calloc` инициализирует выделяемую память, обнуляя все биты. Стандарт ISO C не гарантирует, что в результате это даст числа с плавающей точкой, равные 0, или пустые указатели.

7.7 Куча и стек не размещаются в памяти, пока программа не будет запущена одной из функций семейства `exec` (описывается в разделе 8.10).

7.8 Выполняемый файл (`a.out`) содержит отладочную информацию, которая может оказаться полезной при анализе файла `core`. Удалить эту информацию можно командой `strip`(1). Удаление отладочной информации из двух файлов `a.out` помогло уменьшить их размеры до 798 760 и 6200 байт.

7.9 Когда не используются разделяемые библиотеки, большую часть выполняемого файла занимает стандартная библиотека ввода/вывода.

7.10 Этот код содержит ошибку, поскольку пытается вернуть ссылку на переменную `val` с автоматическим классом размещения после того, как переменная перестала существовать. Автоматические переменные, объявленные после левой открывающей скобки, с которой начинается составной оператор, не видны за правой закрывающей скобкой.

Глава 8

8.1 Чтобы смоделировать ситуацию закрытия стандартного вывода при завершении дочернего процесса, добавьте следующую строку перед вызовом функции `exit` в дочернем процессе:

```
fclose(stdout);
```

Чтобы увидеть, как действует эта строка, замените вызов функции `printf` строками

```
i = printf("pid = %ld, glob = %d, var = %d\n",
           (long)getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

Вам также потребуется определить переменные `i` и `buf`.

Здесь предполагается, что стандартный поток `stdout` будет закрыт, когда дочерний процесс вызовет функцию `exit`, но дескриптор `STDOUT_FILENO` останется открытым. Некоторые версии стандартной библиотеки ввода/вывода при закрытии стандартного потока вывода закрывают и файловый дескриптор, в результате функция `write` также будет завершаться неудачей. В этом случае с помощью функции `dup` продублируйте стандартный вывод на какой-либо другой дескриптор и используйте его в функции `write`.

Рассмотрим программу в листинге С.7.

Листинг С.7. Некорректное использование функции vfork

```
#include "apue.h"

static void f1(void), f2(void);

int
main(void)
{
    f1();
    f2();
    _exit(0);
}

static void
f1(void)
{
    pid_t   pid;

    if ((pid = vfork()) < 0)
        err_sys("ошибка вызова функции vfork");
    /*
     * Оба процесса, дочерний и родительский, выполняют возврат
     * в вызывающую функцию.
     */
}

static void
f2(void)
{
    char    buf[1000]; /* переменные с автоматическим классом размещения */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

К моменту вызова функции `vfork` указатель стека в родительском процессе будет содержать адрес кадра функции `f1`, которая вызвала `vfork`. Это показано на рис. С.2. После вызова `vfork` дочерний процесс первым получает управление и выполняет возврат из функции `f1`. После этого потомок вызы-

вает функцию `f2` и кадр стека этой функции накладывается на предыдущий кадр стека функции `f1`. Затем дочерний процесс забивает нулями 1000 байт автоматической переменной `buf`, размещенной на стеке. Далее дочерний процесс выполняет возврат из `f2` и вызывает `_exit`, но содержимое стека ниже кадра стека функции `main` уже изменилось. После этого родительский процесс возобновляет работу и производит возврат из функции `f1`. Адрес возврата из функции чаще всего хранится на стеке, но эта информация наверняка уже изменена дочерним процессом. Что может произойти с родительским процессом в данном примере, во многом зависит от различных особенностей реализации конкретной версии UNIX (где в стеке хранится адрес возврата из функции, какая информация на стеке будет уничтожена при изменении содержимого автоматической переменной и т. п.). Типичный результат — аварийное завершение родительского процесса с созданием файла `core`, но у вас результаты могут быть иными.

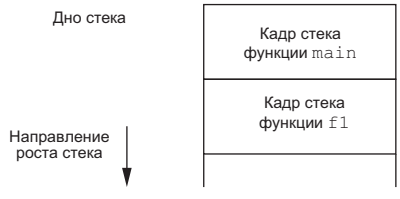

Рис. С.2. Содержимое стека в момент вызова функции vfork

8.4 В листинге 8.7 мы заставляли родительский процесс начинать вывод первым. Когда родительский процесс заканчивал вывод, свою строку начинал выводить дочерний процесс, но при этом мы разрешали родительскому процессу завершить работу, не дожидаясь завершения потомка. Что произойдет раньше, завершение работы родительского процесса или завершение вывода дочерним процессом, зависит от реализации алгоритма планирования процессов в ядре (еще одна разновидность гонки за ресурсами). Когда завершается родительский процесс, командная оболочка запускает следующую программу, и вывод этой программы смешивается с выводом дочернего процесса, запущенного предыдущей программой.

Мы можем предотвратить эту ситуацию, запретив родительскому процессу завершать работу раньше, чем дочерний процесс завершит вывод своей строки. Замените код, следующий за вызовом функции `fork`, следующим фрагментом:

```
else if (pid == 0) {
    WAIT_PARENT();          /* родительский процесс стартует первым */
    charatatime("от дочернего процесса\n");
    TELL_PARENT(getppid()); /* сообщить родителю о завершении вывода */
} else {
    charatatime("от родительского процесса\n");
    TELL_CHILD(pid);       /* сообщить потомку о завершении вывода */
    WAIT_CHILD();          /* подождать, пока потомок завершит вывод */
}
```

Мы не сможем наблюдать подобный эффект, если позволим дочернему процессу стартовать первым, поскольку командная оболочка не запустит следующую программу, пока не завершится родительский процесс.

8.5 Аргумент `argv[2]` будет иметь то же значение (`/home/sar/bin/testinterp`). Это объясняется тем, что работа функции `execlp` завершается вызовом `execve` с тем же значением аргумента *pathname*, что и при непосредственном обращении к функции `execl` (рис. 8.2).

8.6 Программа из листинга С.8 создает процесс-зомби.

Листинг С.8. Создает процесс-зомби, состояние которого можно затем проверить с помощью ps

```
#include "apue.h"

#ifdef SOLARIS
#define PSCMD "ps -a -o pid,ppid,s,tty,comm"
#else
#define PSCMD "ps -o pid,ppid,state,tty,command"
#endif

int
main(void)
{
    pid_t   pid;

    if ((pid = fork()) < 0)
        err_sys("ошибка вызова функции fork");
    else if (pid == 0) /* потомок */
        exit(0);

    /* предок */
    sleep(4);
    system(PSCMD);

    exit(0);
}
```

Обычно команда `ps` обозначает процессы-зомби с помощью символа `Z`.

```
$ ./a.out
 PID  PPID S TT       COMMAND
2369  2208 S pts/2    -bash
7230  2369 S pts/2    ./a.out
7231  7230 Z pts/2    [a.out] <defunct>
7232  7230 S pts/2    sh -c ps -o pid,ppid,state,tty,command
7233  7232 R pts/2    ps -o pid,ppid,state,tty,command
```

Глава 9

9.1 Процесс `init` знает, когда пользователь производит выход из системы с терминала, потому что является родительским процессом по отношению к ко-

мандной оболочке входа и получает сигнал `SIGCHLD`, когда она завершает работу.

Однако в случае входа в систему через сетевое соединение процесс `init` никак не задействован. Записи в файлы `utmp` и `wtmp` о входе в систему и выходе из системы обычно записываются процессом, который обслуживает вход в систему и определяет момент выхода (в нашем случае — сервер `telnetd`).

Глава 10

10.1 Программа завершит работу, когда мы пошлем ей первый сигнал. Дело в том, что функция `pause` возвращает управление сразу, как только будет перехвачен какой-либо сигнал.

10.3 Схема состояния стека приводится на рис. С.3. Вызов функции `longjmp` из `sig_alrm` выполняет переход обратно в функцию `sleep2`, прерывая работу функции `sig_int`. В этой точке `sleep2` возвращает управление функции `main`.

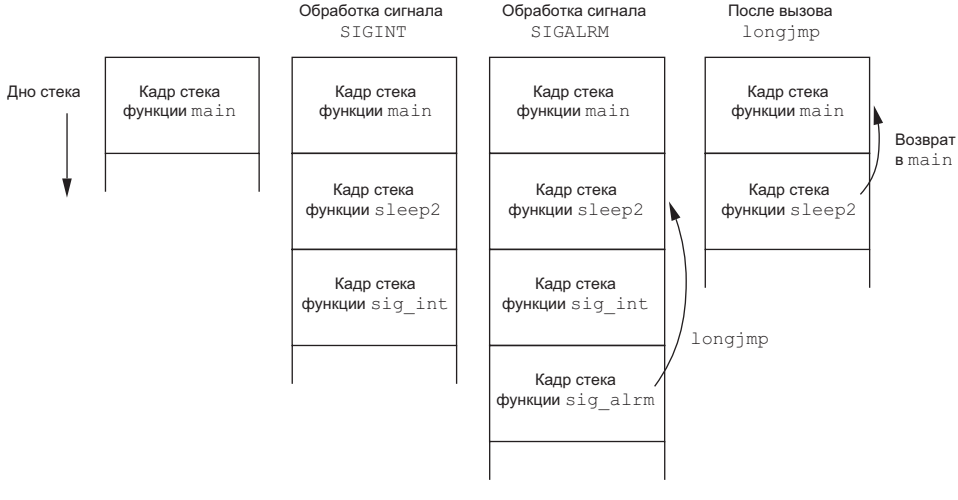

Рис. С.3. Состояние стека до и после вызова функции longjmp

10.4 Мы снова столкнулись с состоянием гонки за ресурсами, на этот раз между первым вызовом функции `alarm` и вызовом функции `setjmp`. Если ядро заблокирует процесс между этими двумя вызовами и истечет время таймаута, процесс получит сигнал и для его обработки вызовет обработчик сигнала, который, в свою очередь, вызовет функцию `longjmp`. Но так как `setjmp` еще не вызывалась, буфер `env_alrm` не будет заполнен корректными значениями. Поведение функции `longjmp` не определено, если буфер перехода не был инициализирован функцией `setjmp`.

10.5 За примерами обращайтесь к статье Дона Либеса (Don Libes) «Implementing Software Timers» (C Users Journal, vol. 8, no. 11, Nov 1990). Электронная ко-

пия этой статьи доступна по адресу *http://www.kohala.com/start/libes.timers.txt*.

10.7 Если просто вызвать функцию `_exit`, по коду завершения процесса не будет видно, что он завершился по сигналу `SIGABRT`.

10.8 Если сигнал был послан процессом, который принадлежит некоторому другому пользователю, этот процесс должен иметь сохраненный set-user-ID, равный идентификатору суперпользователя или идентификатору пользователя владельца процесса, принимающего сигнал, иначе функция `kill` не сможет послать сигнал. Поэтому реальный идентификатор несет больше информации для процесса, принимающего сигнал.

10.10 В одной из систем, используемых автором, значение количества секунд увеличивалось на 1 каждые 60–90 минут. Это отклонение обусловлено тем, что каждый вызов `sleep` планирует событие в будущем, но момент пробуждения процесса не совсем точно соответствует запланированному (из-за нагрузки на центральный процессор). Кроме того, некоторый объем времени требуется, чтобы возобновить работу процесса после приостановки и опять вызвать функцию `sleep`.

Такие программы, как `cron`, получают текущее время каждую минуту и в первый раз задают время приостановки таким, чтобы возобновить работу в начале следующей минуты (преобразуя текущее время в локальное и извлекая значение поля `tm_sec`). Каждую минуту они устанавливают величину очередного периода приостановки так, чтобы процесс возобновил работу в начале следующей минуты. Обычно это будут вызовы `sleep(60)` и изредка, для синхронизации с текущим временем, `sleep(59)`. Но иногда, когда выполнение запланированных команд занимает продолжительное время или при высокой нагрузке на систему, может быть выбрано меньшее значение аргумента функции `sleep`.

10.11 В Linux 3.2.0, Mac OS X 10.6.8 и Solaris 10 обработчик сигнала `SIGXFSZ` никогда не будет вызван. Но функция `write` вернет число 24, как только размер файла превысит 1024 байт. В FreeBSD 8.9 и Mac OS X 10.6.8, когда размер файла достигнет 1000 байт, обработчик сигнала будет вызван при следующей же попытке записать очередные 100 байт, а функция `write` вернет значение –1 с кодом ошибки `EFBIG` (File too big — файл слишком велик) в переменной `errno`.

10.12 Результат зависит от реализации стандартной библиотеки ввода/вывода: от того, как функция `fwrite` обрабатывает прерывание системного вызова `write`.

В Linux 3.2.0, например, когда функция `fwrite` используется для записи большого буфера, она вызывает `write` непосредственно, передавая ей то же количество байтов. Если в процессе выполнения системного вызова `write` поступит сигнал `SIGALRM`, приложение не получит его, пока `write` не завершит запись. По всей видимости ядро блокирует сигнал, если он поступает в процессе выполнения системного вызова `write`.

В Solaris 10, напротив, функция `fwrite` передает данные системному вызову `write` блоками по 8 Кбайт. Поэтому сигнал `SIGALRM` в этой системе может прервать выполнение функции `fwrite`. После возврата из обработчика сигнала управление будет передано во внутренний цикл функции `fwrite` и она продолжит запись данных блоками по 8 Кбайт.

Глава 11

11.1 Версия программы, которая выделяет область динамической памяти вместо использования автоматических переменных, приводится в листинге С.9.

Листинг С.9. Корректное использование возвращаемого значения потока

```
#include "apue.h"
#include <pthread.h>

struct foo {
    int a, b, c, d;
};

void
printfoo(const char *s, const struct foo *fp)
{
    fputs(s, stdout);
    printf("  структура по адресу 0x%lx\n", (unsigned long)fp);
    printf("  foo.a = %d\n", fp->a);
    printf("  foo.b = %d\n", fp->b);
    printf("  foo.c = %d\n", fp->c);
    printf("  foo.d = %d\n", fp->d);
}

void *
thr_fn1(void *arg)
{
    struct foo *fp;

    if ((fp = malloc (sizeof(struct foo))) == NULL)
        err_sys("невозможно выделить область динамической памяти");
    fp->a = 1;
    fp->b = 2;
    fp->c = 3;
    fp->d = 4;
    printfoo("поток:\n", fp);
    return((void *)fp);
}

int
main(void)
{
    int err;
    pthread_t tid1;
    struct foo *fp;
```

```
    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "невозможно создать поток 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "невозможно присоединить поток 1");
    printfoo("родительский процесс:\n", fp);
    exit(0);
}
```

11.2 Чтобы изменить идентификатор потока для задания, ожидающего обработки, необходимо блокировку чтения/записи установить в режиме для записи, чтобы предотвратить возможность поиска по списку, пока не будет произведено изменение идентификатора. Проблема, связанная с текущим определением интерфейсов, заключается в том, что идентификатор задания может измениться между моментом, когда задание будет найдено функцией `job_find`, и моментом, когда задание будет исключено из списка функцией `job_remove`. Эту проблему можно решить добавлением счетчика ссылок и мьютекса в структуру `job`; тогда функция `job_find` должна будет увеличивать счетчик ссылок, а код, который производит изменение идентификатора, сможет пропускать те задания в списке, которые имеют ненулевой счетчик ссылок.

11.3 Во-первых, список защищен блокировкой чтения/записи, но переменная состояния должна быть под защитой мьютекса. Во-вторых, каждый поток должен ожидать появления задания для обработки на своей собственной переменной состояния, поэтому нам придется создать для каждого потока структуру данных, которая представляла бы это состояние. Как вариант, можно ввести переменную состояния и мьютекс в структуру `queue`, но это означало бы, что все рабочие потоки ожидали бы на одной и той же переменной состояния. При большом количестве рабочих потоков мы могли бы столкнуться с проблемой *гремящего стада* (thundering herd), когда большое количество потоков возобновляют работу, но для них не находится заданий и в результате они впустую расходуют ресурсы процессора, ужесточая борьбу за обладание блокировкой.

11.4 Это зависит от обстоятельств. Вообще оба варианта могут работать вполне корректно, но каждый из них имеет свои недостатки. В первом случае ожидающие потоки будут запланированы на возобновление работы после вызова `pthread_cond_broadcast`. Если программа работает в многопроцессорной среде, некоторые запущенные потоки окажутся сразу же заблокированными, потому что мьютекс все еще заперт (не забывайте, что `pthread_cond_wait` возвращает управление с запертым мьютексом). Во втором случае работающий поток может успеть захватить мьютекс между действиями 3 и 4, среагировать на изменение состояния, сделав его недействительным, и освободить мьютекс. Затем, когда будет вызвана `pthread_cond_broadcast`, состояние больше не будет истинным, и поток отработает понапрасну. По этой причине поток всегда должен перепроверять истинность состояния, а не полагаться на то, что оно истинно, просто потому, что функция `pthread_cond_wait` вернула управление.

Глава 12

12.1 Эта проблема не связана с многопоточной архитектурой приложения, как может показаться на первый взгляд. Процедуры стандартной библиотеки ввода/вывода в действительности являются безопасными в контексте потоков. Когда мы называем функцию `fork`, каждый процесс получает отдельную копию структур данных стандартной библиотеки ввода/вывода. При запуске программы со стандартным выводом, присоединенным к терминалу, вывод будет буферизоваться построчно, поэтому каждый раз, когда мы выводим строку, стандартная библиотека ввода/вывода будет записывать ее в устройство терминала. Однако если перенаправить стандартный вывод в файл, библиотека выберет для него режим полной буферизации. Фактическая запись в файл будет произведена только при заполнении буфера или при закрытии потока. В этом примере к моменту вызова функции `fork` буфер уже содержит несколько еще не записанных в файл строк, поэтому, когда родительский и дочерний процессы наконец сбросят свои копии буферов, первоначальное их содержимое будет записано в файл дважды.

12.3 Теоретически, заблокировав доставку всех сигналов при вызове обработчика сигнала, мы могли бы сделать функцию безопасной в контексте обработки асинхронных сигналов. Проблема в том, что мы не знаем, не разблокирует ли какая-либо функция, к которой мы обращаемся, какой-либо из заблокированных сигналов, сделав тем самым возможным повторное вхождение в обработчик другого сигнала.

12.4 В FreeBSD 8.0 программа завершилась аварийно с созданием файла `core`. С помощью отладчика `gdb` удалось определить, что программа застряла в бесконечном цикле инициализации. В процессе инициализации программа вызывала функции инициализации потоков, которые обращаются к функции `getenv`, чтобы получить значения переменных окружения `LIBPTHREAD_SPINLOOPS` и `LIBPTHREAD_YIELDLOOPS`. Однако наша потокобезопасная реализация `getenv` вызывает функции из библиотеки `pthread`, находясь в промежуточном, противоречивом состоянии. Кроме того, функции инициализации потоков пытаются вызвать `malloc`, которая, в свою очередь, вызывает `getenv`, чтобы получить значение переменной окружения `MALLOC_OPTIONS`.

Чтобы обойти эту проблему, можно по умолчанию полагать, что программа запускается как однопоточная, и сообщать нашей версии `getenv` о необходимости инициализации потока с помощью флага. При ложном значении флага наша версия `getenv` может действовать подобно нереентерабельной версии (и тем самым избежать вызовов функций из библиотеки `pthread` и `malloc`). Также можно было бы реализовать отдельную функцию инициализации, вызывающую `pthread_once`, чтобы не вызывать ее из `getenv`. При такой организации программа должна будет вызывать данную функцию инициализации перед вызовом `getenv`. Это решает проблему, потому что `gentenv` не будет вызвана, пока программа не завершит инициализацию. А после вызова функции инициализации наша версия `getenv` будет действовать потокобезопасным образом.

12.5 Функция `fork` по-прежнему необходима, если мы захотим запустить одну программу из другой (то есть вызывать `fork` перед вызовом `exec`).

12.6 В листинге С.10 приводится потокобезопасная реализация функции `sleep`, которая для организации задержки использует функцию `select`. Она безопасна в многопоточной среде потому, что не использует никаких незащищенных глобальных или статических данных и вызывает только безопасные функции.

12.7 Реализация переменной состояния, скорее всего, использует мьютекс для защиты ее внутренней структуры. Поскольку это уже относится к области реализации конкретных версий UNIX и скрыто от нас, какого-либо переносимого способа захватить или отпустить блокировку в момент ветвления процесса не существует. Поскольку мы не можем определить состояние внутренней блокировки в переменной состояния после вызова функции `fork`, использование переменных состояния в дочернем процессе будет небезопасным.

Листинг С.10. Реализация потокобезопасной функции sleep

```
#include <unistd.h>
#include <time.h>
#include <sys/select.h>

unsigned
sleep(unsigned nsec)
{
    int n;
    unsigned slept;
    time_t start, end;
    struct timeval tv;

    tv.tv_sec = nsec;
    tv.tv_usec = 0;
    time(&start);
    n = select(0, NULL, NULL, NULL, &tv);
    if (n == 0)
        return(0);
    time(&end);
    slept = end - start;
    if (slept >= nsec)
        return(0);
    return(nsec - slept);
}
```

Глава 13

13.1 Если процесс вызовет функцию `chroot`, он не сможет открыть устройство `/dev/log`. Решение заключается в том, чтобы вызвать функцию `openlog` с флагом `LOG_NDELAY` в аргументе *option* перед обращением к функции `chroot`. В результате демон откроет специальный файл устройства (сокет дейтаграмм из домена UNIX), что даст ему дескриптор, который останется

действительным даже после вызова `chroot`. С подобным алгоритмом можно столкнуться в таких демонах, как `ftpd` (демон службы передачи файлов по протоколу FTP), где функция `chroot` используется из соображений безопасности, но для регистрации ошибок в системном журнале используется `syslog`.

13.3 Решение приводится в листинге С.11. Результат зависит от платформы. Вспомните, что функция `daemonize` закрывает все дескрипторы файлов и снова открывает первые три на устройстве `/dev/null`. Это означает, что процесс не имеет управляющего терминала, в результате функция `getlogin` не сможет отыскать запись о процессе в файле `utmp`. Поэтому в Linux 3.2.0 и Solaris 10 можно обнаружить, что демоны не имеют имени пользователя.

Однако в FreeBSD 8.0 и Mac OS X 10.6.8 имя пользователя сохраняется в таблице процессов и копируется в дочерний процесс при вызове функции `fork`. Это означает, что процесс всегда может узнать имя пользователя, если только он не был запущен одним из процессов, которые не имеют имени пользователя (как, например, процесс `init`).

Листинг С.11. Вызов функции daemonize и попытка определить имя пользователя

```
#include "apue.h"

int
main(void)
{
   FILE *fp;
   char *p;

   daemonize("getlog");
   p = getlogin();
   fp = fopen("/tmp/getlog.out", "w");
   if (fp != NULL) {
      if (p == NULL)
         fprintf(fp, "процесс не имеет имени пользователя\n");
      else
         fprintf(fp, "имя пользователя: %s\n", p);
   }
   exit(0);
}
```

Глава 14

14.1 Тестовая программа приводится в листинге С.12.

Листинг С.12. Проверка поведения механизма блокировки записей в файле

```
#include "apue.h"
#include <fcntl.h>
#include <errno.h>

void
```

```
sigint(int signo)
{
}

int
main(void)
{
    pid_t pid1, pid2, pid3;
    int fd;

    setbuf(stdout, NULL);
    signal_intr(SIGINT, sigint);

    /*
     * Создать файл.
     */
    if ((fd = open("lockfile", O_RDWR|O_CREAT, 0666)) < 0)
        err_sys("невозможно открыть/создать файл блокировки");

    /*
     * Установить блокировку для чтения.
     */
    if ((pid1 = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid1 == 0) { /* потомок */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("потомок 1: невозможно заблокировать "
                    "файл для чтения");
        printf("потомок 1: установлена блокировка для чтения\n");
        pause();
        printf("потомок 1: выход после паузы\n");
        exit(0);
    } else { /* предок */
        sleep(2);
    }

    /*
     * Родительский процесс продолжается ...
     * снова установить блокировку для чтения.
     */
    if ((pid2 = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid2 == 0) { /* потомок */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("потомок 2: невозможно заблокировать "
                    "файл для чтения");
        printf("потомок 2: установлена блокировка для чтения\n");
        pause();
        printf("потомок 2: выход поле паузы\n");
        exit(0);
    } else { /* родительский процесс */
        sleep(2);
    }
```

```
    /*
     * Родительский процесс продолжается ... блокируется
     * при попытке установить блокировку для записи.
     */
    if ((pid3 = fork()) < 0) {
        err_sys("ошибка вызова функции fork");
    } else if (pid3 == 0) { /* потомок */
        if (lock_reg(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
            printf("потомок 3: невозможно заблокировать "
                   "файл для записи:%s\n",
                   strerror(errno));
        printf("потомок 3: останов, пока не получит блокировку...\n");
        if (lock_reg(fd, F_SETLKW, F_WRLCK, 0, SEEK_SET, 0) < 0)
            err_sys("потомок 3: невозможно заблокировать "
                    "файл для записи");
        printf("потомок 3 сумел установить блокировку для записи???\n");
        pause();
        printf("потомок 3: выход после паузы\n");
        exit(0);
    } else { /* родительский процесс */
        sleep(2);
    }

    /*
     * Проверить, будет ли заблокирована попытка получить
     * блокировку для записи очередной попыткой установки
     * блокировки для чтения.
     */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        printf("родитель: невозможно заблокировать файл для чтения: %s\n",
               strerror(errno));
    else
        printf("родитель: установлена дополнительная "
               "блокировка для чтения,"
               " запрос на установку блокировки для записи ожидает\n");
    printf("останавливается потомок 1...\n");
     kill(pid1, SIGINT);
     printf("останавливается потомок 2...\n");
     kill(pid2, SIGINT);
     printf("останавливается потомок 3...\n");
     kill(pid3, SIGINT);
     exit(0);
}
```

В FreeBSD 8.0, Linux 3.2.0 и Mac OS X 10.6.8 был получен одинаковый результат: дополнительные читающие процессы могут оставить пишущие процессы ни с чем. Запустив программу, мы получили следующие результаты:

```
потомок 1: установлена блокировка для чтения
потомок 2: установлена блокировка для чтения
потомок 3: невозможно заблокировать файл для записи: Resource temporarily
```

```
unavailable
потомок 3 пытается установить блокировку для записи
родитель: установлена дополнительная блокировка для чтения, запрос
на установку блокировки для записи ожидает
останавливается потомок 1...
потомок 1: выход после паузы
останавливается потомок 2...
потомок 2: выход после паузы
останавливается потомок 3...
потомок 3: невозможно заблокировать файл для записи: Interrupted system call
```

В Solaris 10 читающие процессы не способны заблокировать пишущие процессы. В данном примере родительский процесс не сможет получить блокировку на чтение, потому что имеются процессы, ожидающие получения блокировки для записи.

14.2 В большинстве систем тип данных `fd_set` определен как структура, которая содержит всего одно поле — массив длинных целых чисел. Каждый бит в этом массиве соответствует одному дескриптору. Макросы `FD_` работают с этим массивом длинных целых чисел, включая, выключая и возвращая состояние отдельных битов.

Одна из причин, по которым этот тип данных был объявлен как структура, содержащая массив, а не просто как массив, заключается в том, что это дает возможность присваивать значение одной переменной типа `fd_set` другой переменной типа `fd_set` обычным оператором присваивания языка C.

14.3 В старые добрые времена большинство систем допускало возможность определения константы `FD_SETSIZE` перед подключением заголовочного файла `<sys/select.h>`. Например, с помощью инструкций

```
#define FD_SETSIZE 2048
#include <sys/select.h>
```

можно определить размер типа `fd_set` таким, чтобы он мог вместить 2048 дескрипторов. К сожалению, это больше невозможно. Чтобы добиться подобного в современных системах, необходимо:

1. Прежде чем подключать какие-либо заголовочные файлы, необходимо определить символ, предотвращающий подключение `<sys/select.h>`. Некоторые системы могут защищать определение типа `fd_set` отдельным символом. Нам также нужно определить его. Например, чтобы предотвратить подключение `<sys/select.h>` в FreeBSD 8.0, необходимо определить символ `_SYS_SELECT_H_` и также определить символ `_FD_SET`, чтобы предотвратить включение определения типа `fd_set`.
2. Иногда, для совместимости со старыми приложениями, заголовочный файл `<sys/types.h>` определяет размер типа `fd_set`, поэтому необходимо сначала подключить его, а затем удалить символ `FD_SETSIZE`. Обратите внимание, что в некоторых системах вместо `FD_SETSIZE` используется символ `__FD_SETSIZE`.

3. Далее следует переопределить символ `FD_SETSIZE` (или `__FD_SETSIZE`), указав желаемое максимальное значение количества дескрипторов, которое может использоваться с функцией `select`.
4. Затем необходимо удалить символ, определенный на шаге 1.
5. И наконец, подключить `<sys/select.h>`.

Прежде чем запускать программу, следует настроить систему, чтобы она позволяла открывать столько дескрипторов, сколько потребуется, и мы действительно могли использовать `FD_SETSIZE` дескрипторов.

14.4 В следующей таблице перечислены функции, которые решают сходные задачи.

`FD_ZERO`	`sigemptyset`
`FD_SET`	`sigaddset`
`FD_CLR`	`sigdelset`
`FD_ISSET`	`sigismember`

В семействе `FD_XXX` нет функции, которая соответствовала бы функции `sigfillset`. При работе с сигналами указатель на набор сигналов всегда передается в первом аргументе, а номер сигнала — во втором. При работе с наборами дескрипторов в первом аргументе передается номер дескриптора, а в следующем — указатель на набор дескрипторов.

14.5 В листинге С.13 показана реализация с использованием функции `select`.

Листинг С.13. Реализация функции sleep_us на основе функции select

```
#include "apue.h"
#include <sys/select.h>

void
sleep_us(unsigned int nusecs)
{
    struct timeval tval;

    tval.tv_sec = nusecs / 1000000;
    tval.tv_usec = nusecs % 1000000;
    select(0, NULL, NULL, NULL, &tval);
}
```

В листинге С.14 показана аналогичная реализация с использованием функции `poll`.

Листинг С.14. Реализация функции sleep_us на основе функции poll

```
#include <poll.h>

void
sleep_us(unsigned int nusecs)
{
```

```
    struct pollfd dummy;
    int timeout;

    if ((timeout = nusecs / 1000) <= 0)
        timeout = 1;
    poll(&dummy, 0, timeout);
}
```

Как утверждает страница справочного руководства `usleep`(3) в BSD, функция usleep использует в своей работе `nanosleep`. Она корректно взаимодействует с другими таймерами, установленными вызывающим процессом, и не прерывается в случае перехвата сигнала.

14.6 Нет. В этом случае `TELL_WAIT` должна была бы создать временный файл длиной в два байта, где один байт отводится для родительского и один байт — дочернего процесса. Функция `WAIT_CHILD` могла бы заставить родительский процесс ожидать снятия блокировки с байта дочернего процесса, а `TELL_PARENT` — снимать блокировку с байта дочернего процесса. Проблема, однако, в том, что функция `fork` снимает все блокировки в дочернем процессе, поэтому дочерний процесс не может быть запущен с какими-либо установленными блокировками.

14.7 Решение приводится в листинге С.15.

Листинг С.15. Подсчет емкости неименованного канала с помощью неблокирующей операции записи

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    int i, n;
    int fd[2];

    if (pipe(fd) < 0)
        err_sys("ошибка вызова функции pipe");
    set_fl(fd[1], O_NONBLOCK);

    /*
     * Записывать по 1 байту, пока канал не заполнится.
     */
    for (n = 0; ; n++) {
        if ((i = write(fd[1], "a", 1)) != 1) {
            printf("функция write вернула число %d, ", i);
            break;
        }
    }
    printf("емкость канала = %d\n", n);
    exit(0);
}
```

В следующей таблице показаны значения, полученные на наших четырех платформах.

Платформа	Емкость канала в байтах
FreeBSD 8.0	65 536
Linux 3.2.0	65 536
Mac OS X 10.6.8	16 384
Solaris 10	16 384

Эти значения могут отличаться от значения константы `PIPE_BUF`, поскольку эта константа определяет максимальный объем данных, которые можно записать в канал атомарно. Здесь же мы получили объем данных, которые могут находиться в канале, не принимая во внимание атомарность их записи.

14.10 Изменит ли программа из листинга 14.10 время последнего обращения к исходному файлу, зависит от операционной системы и типа файловой системы, в которой размещается файл. На всех четырех платформах, рассматриваемых в книге, время последнего обращения к файлу обновляется, если он располагается в файловой системе, используемой по умолчанию этими платформами.

Глава 15

15.1 Если конец канала, открытый для записи, не будет закрыт, процесс, читающий данные из канала, никогда не увидит признака конца файла. Поэтому программа постраничного просмотра окажется «навечно» заблокированной в операции чтения со стандартного ввода.

15.2 Родительский процесс завершится сразу после записи в канал последней строки. Конец канала, открытый для чтения, автоматически закроется по завершении родительского процесса. Но родительский процесс наверняка опережает потомка на один буфер, поскольку дочерний процесс (программа постраничного просмотра) ожидает, пока пользователь не просмотрит выведенную перед ним страницу. Если запустить программу в командной оболочке, такой как Korn shell, которая работает в диалоговом режиме, оболочка наверняка изменит режим терминала по завершении работы родительского процесса и выведет свое приглашение. Это несомненно повлияет на программу постраничного просмотра, так как она тоже изменяет режим терминала. (Большинство программ постраничного просмотра в ожидании перехода к следующей странице переводят терминал в неканонический режим.)

15.3 Функция `popen` вернет указатель на структуру `FILE`, потому что она запустит командную оболочку. Но сама командная оболочка не сможет выполнить несуществующую команду и потому выведет строку

```
sh: line 1: ./a.out: No such file or directory
```

в стандартное устройство вывода сообщений об ошибках и завершится с кодом 127 (впрочем, код завершения зависит от типа командной оболочки). Функция `pclose` вернет код завершения команды, который будет получен от функции `waitpid`.

15.4 После завершения родительского процесса посмотрите код его завершения. В Bourne shell, Bourne-again shell и Korn shell это можно сделать с помощью команды `echo $?`. Она выведет число, равное сумме числа 128 и номера сигнала.

15.5 Прежде всего, нужно добавить объявление

```
FILE *fpin, *fpout;
```

Затем с помощью функции `fdopen` связать дескриптор канала с потоком ввода/вывода и назначить ему построчный режим буферизации. Сделать это необходимо перед входом в цикл `while`, где производится чтение из стандартного ввода:

```
if ((fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("ошибка вызова функции fdopen");
if ((fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("ошибка вызова функции fdopen");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("ошибка вызова функции setvbuf");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("ошибка вызова функции setvbuf");
```

Обращения к функциям `read` и `write` в цикле заменить строками

```
if (fputs(line, fpout) == EOF)
    err_sys("ошибка вывода в канал");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("дочерний процесс закрыл канал");
    break;
}
```

15.6 Функция `system` вызовет `wait`, и первым завершится дочерний процесс, запущенный функцией `popen`. Поскольку это не тот потомок, который был запущен функцией `system`, она снова вызовет функцию `wait` и заблокируется, пока не завершится работа команды `sleep`. После этого функция `system` вернет управление. Когда `pclose` вызовет `wait`, она вернет признак ошибки, поскольку все дочерние процессы уже завершили работу. Вслед за ней и сама `pclose` вернет признак ошибки.

15.7 Функция `select` отметит дескриптор как доступный для чтения. Когда функция `read` будет вызвана после чтения всех данных из канала, она вернет 0 в качестве признака конца файла. В случае с функцией `poll` будет возвращено событие `POLLHUP`, а оно может быть возвращено, даже если в канале еще имеются данные, доступные для чтения. Однако когда функция `read` прочитает все данные, она вернет 0 как признак конца файла. После чтения всех данных событие `POLLIN` возвращено не будет, даже если нам еще только

предстоит прочитать признак конца файла (возвращаемое значение 0 функции `read`).

Таблица С.1. Поведение функций select и poll при работе с каналами

Операция	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Вызов `select` в читающем процессе, когда пишущий процесс закрыл свой дескриптор	R/W/E	R	R/W	R/W/E
Вызов `poll` в читающем процессе, когда пишущий процесс закрыл свой дескриптор	R/HUP	HUP	INV	HUP
Вызов `select` в пишущем процессе, когда читающий процесс закрыл свой дескриптор	R/W/E	R/W	R/W	R/W
Вызов `poll` в пишущем процессе, когда читающий процесс закрыл свой дескриптор	R/HUP	W/ERR	INV	HUP

Сокращения в табл. С.1: R (readable — доступно для чтения), W (writable — доступно для записи), E (exception — исключение), HUP (hangup — разрыв связи), ERR (error — ошибка) и INV (invalid file descriptor — недопустимый дескриптор файла). Для дескриптора, ссылающегося на канал, который был закрыт читающим процессом, `select` сообщит, что дескриптор доступен для записи. Но когда будет вызвана функция `write`, система сгенерирует сигнал `SIGPIPE`. Если сигнал игнорируется программой или обработчик сигнала вернет управление, `write` вернет признак ошибки с кодом `EPIPE` в переменной `errno`. Однако поведение функции `poll` в подобной ситуации может отличаться в разных системах.

15.8 Все, что будет выведено дочерним процессом в стандартный вывод сообщений об ошибках, будет отправлено в стандартный вывод сообщений об ошибках родительского процесса. Чтобы отправить данные со стандартного вывода сообщений об ошибках родительскому процессу, включите в *cmdstring* операцию перенаправления `2>&1`.

15.9 Функция `popen` создает дочерний процесс, а он запускает командный интерпретатор. Командный интерпретатор, в свою очередь, вызывает `fork`, и новый дочерний процесс командного интерпретатора запускает командную строку. Родительский командный интерпретатор дожидается, когда *cmdstring* завершится, и также завершает работу, чего, в свою очередь, ожидает функция `waitpid` в `pclose`.

15.10 Хитрость заключается в том, что канал FIFO надо открыть дважды: один раз для чтения и один раз для записи. Мы вообще не используем дескриптор, открытый для записи, но оставляем его открытым для предотвращения генерации признака конца файла, когда количество клиентов

уменьшается с 1 до 0. Открытие FIFO в два приема требует некоторых дополнительных действий, так как оно должно производиться в неблокирующем режиме. Сначала мы должны открыть FIFO только для чтения в неблокирующем режиме, а затем вызвать `open` в блокирующем режиме, чтобы открыть канал только для записи. (Если мы сначала попытаемся открыть FIFO в неблокирующем режиме только для записи, функция `open` вернет признак ошибки.) Затем мы должны сбросить флаг неблокирующего режима в дескрипторе, открытом для чтения. В листинге С.16 показано, как это делается.

Листинг С.16. Открытие канала FIFO для чтения и записи без блокировки процесса

```
#include "apue.h"
#include <fcntl.h>

#define FIFO "temp.fifo"

int
main(void)
{
    int fdread, fdwrite;

    unlink(FIFO);
    if (mkfifo(FIFO, FILE_MODE) < 0)
        err_sys("ошибка вызова функции mkfifo");
    if ((fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("ошибка открытия для чтения");
    if ((fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("ошибка открытия для записи");
    clr_fl(fdread, O_NONBLOCK);
    exit(0);
}
```

15.11 Беспорядочное чтение сообщений из активной очереди может повлечь за собой конфликты между сервером и клиентом из-за несоблюдения протокола обмена, так как в этом случае могут быть утеряны либо запросы клиента, либо отклики сервера. Чтобы получить возможность чтения из очереди, процесс должен знать ее идентификатор, а очередь должна иметь установленный бит world-read (доступ на чтение для всех остальных).

15.12 Мы никогда не должны хранить фактические адреса в сегменте разделяемой памяти, поскольку существует вероятность, что сервер и все клиенты подключат этот сегмент к различным адресам. Вместо адресов в связанном списке, который строится в сегменте разделяемой памяти, следует использовать величины смещений объекта от начала сегмента разделяемой памяти. Эти смещения формируются путем вычитания адреса начала сегмента разделяемой памяти из адреса объекта.

15.14 В табл. С.2 приводится схема происходящих событий.

Таблица С.2. Чередование периодов работы родительского и дочернего процессов из листинга 15.12

Значение i в родителе	Значение i в потомке	Разделяемое значение	Возвращаемое значение update	Комментарий
		0		Инициализируется функцией `mmap`
	1			Потомок запускается первым и затем блокируется
0				Запускается родитель
		1		
			0	Затем родитель блокируется
		2		Потомок возобновляет работу
			1	
	3			Затем потомок блокируется
2				Родитель возобновляет работу
		3		
			2	Затем родитель блокируется
		4		
			3	
	5			Затем потомок блокируется
4				Родитель возобновляет работу

Глава 16

16.1 В листинге С.17 приводится программа, которая определяет порядок байтов для аппаратной архитектуры, на которой она запущена.

Листинг С.17. Определение порядка байтов

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

int
main(void)
```

```
{
    uint32_t i = 0x04030201;
    unsigned char *cp = (unsigned char *)&i;

    if (*cp == 1)
        printf("обратный (little-endian)\n");
    else if (*cp == 4)
        printf("прямой (big-endian)\n");
    else
        printf("неизвестный?\n");
    exit(0);
}
```

16.3 Каждый из сокетов, который будет принимать запросы на соединение, должен быть привязан к своему адресу, и для каждого дескриптора должна быть создана соответствующая запись в структуре `fd_set`. Для приема запросов на соединение на нескольких адресах мы будем использовать функцию `select`. В разделе 16.4 уже говорилось, что по прибытии запроса на соединение дескриптор сокета будет отмечен как доступный для чтения. Прибывающие запросы на соединение мы будем принимать и обслуживать, как и прежде.

16.5 Для этого нужно установить обработчик сигнала `SIGCHLD`, обратившись к функции `signal` (листинг 10.12), которая устанавливает обработчик сигнала с помощью функции `sigaction`, позволяющей определить возможность перезапуска прерванных системных вызовов. Затем следует убрать вызов `waitpid` из функции `serve`. После запуска дочернего процесса родитель закрывает новый дескриптор и переходит к ожиданию новых запросов на соединение. И наконец, нам нужен сам обработчик сигнала `SIGCHLD`:

```
void
sigchld(int signo)
{
    while (waitpid((pid_t)-1, NULL, WNOHANG) > 0)
        ;
}
```

16.6 Чтобы разрешить асинхронный режим работы сокета, необходимо назначить процесс владельцем сокета с помощью команды `F_SETOWN` функции `fcntl` и разрешить асинхронную доставку сигнала с помощью команды `FIOASYNC` функции `ioctl`. Чтобы запретить асинхронный режим работы сокета, достаточно просто запретить асинхронную доставку сигнала. Смешивание вызовов функций `fcntl` и `ioctl` необходимо для обеспечения переносимости. Код функций приводится в листинге С.18.

Листинг С.18. Функции разрешения и запрещения асинхронного режима работы сокета

```
#include "apue.h"
#include <errno.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
```

```
#if defined(BSD) || defined(MACOS) || defined(SOLARIS)
#include <sys/filio.h>
#endif

int
setasync(int sockfd)
{
    int n;

    if (fcntl(sockfd, F_SETOWN, getpid()) < 0)
        return(-1);
    n = 1;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}

int
clrasync(int sockfd)
{
    int n;

    n = 0;
    if (ioctl(sockfd, FIOASYNC, &n) < 0)
        return(-1);
    return(0);
}
```

Глава 17

17.1 Обычные каналы обеспечивают доступ к данным как к потоку байтов. Для определения границ сообщений в каждое из них необходимо добавить заголовок, где указать длину сообщения. Но при этом все еще сохраняется необходимость выполнять две дополнительные операции копирования: одну — для записи в канал и одну — для чтения из канала. Намного эффективнее использовать канал только для передачи главному потоку сигнала о доступности нового сообщения. Для этого достаточно использовать один байт. При использовании такого решения нам потребуется переместить структуру `mymesg` в структуру `threadinfo` и задействовать мьютекс и переменную состояния, чтобы помешать вспомогательному потоку повторно использовать структуру `mymesg`, пока она не будет обработана главным потоком. Реализация этого решения представлена в листинге С.19.

Листинг С.19. Проверка наличия сообщений XSI с использованием каналов

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ     3      /* количество очередей */
```

```
#define MAXMSZ 512    /* максимальный размер сообщения */
#define KEY    0x123  /* ключ для первой очереди сообщений */

struct mymesg {
    long       mtype;
    char       mtext[MAXMSZ+1];
};

struct threadinfo {
    int             qid;
    int             fd;
    int             len;
    pthread_mutex_t mutex;
    pthread_cond_t  ready;
    struct mymesg   m;
};

void *
helper(void *arg)
{
                        int n;
    struct threadinfo   *tip = arg;

    for(;;) {
        memset(&tip->m, 0, sizeof(struct mymsg));
        if ((n = msgrcv(tip->qid, &tip->m, MAXMSZ, 0,
                        MSG_NOERROR)) < 0)
            err_sys("ошибка вызова функции msgrcv");
        tip->len = n;
        pthread_mutex_lock(&tip->mutex);
        if (write(tip->fd, "a", sizeof(char)) < 0)
            err_sys("ошибка вызова функции write");
        pthread_cond_wait(&tip->ready, &tip->mutex);
        pthread_mutex_unlock(&tip->mutex);
    }
}

int
main()
{
    char                c;
    int                 i, n, err;
    int                 fd[2];
    int                 qid[NQ];
    struct pollfd       pfd[NQ];
    struct threadinfo   ti[NQ];
    pthread_t           tid[NQ];

    for (i = 0; i < NQ; i++) {
        if ((qid[i] = msgget((KEY+i), IPC_CREAT|0666)) < 0)
            err_sys("ошибка вызова функции msgget");

        printf("очередь %d получила идентификатор %d\n", i, qid[i]);
```

```
        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("ошибка вызова функции socketpair");
        pfd[i].fd = fd[0];
        pfd[i].events = POLLIN;
        ti[i].qid = qid[i];
        ti[i].fd = fd[1];
        if (pthread_cond_init(&ti[i].ready, NULL) != 0)
            err_sys("ошибка вызова функции pthread_cond_init");
        if (pthread_mutex_init(&ti[i].mutex, NULL) != 0)
            err_sys("ошибка вызова функции pthread_mutex_init");
        if ((err = pthread_create(&tid[i], NULL, helper,
                                  &ti[i])) != 0)
            err_exit(err, "ошибка вызова функции pthread_create");
    }

    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
            err_sys("ошибка вызова функции poll");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, &c, sizeof(char))) < 0)
                    err_sys("ошибка вызова функции read");
                ti[i].m.mtext[ti[i].len] = 0;
                printf("очередь: %d, сообщение: %s\n", qid[i],
                       ti[i].m.mtext);
                pthread_mutex_lock(&ti[i].mutex);
                pthread_cond_signal(&ti[i].ready);
                pthread_mutex_unlock(&ti[i].mutex);
            }
        }
    }
    exit(0);
}
```

17.3 *Объявление* определяет атрибуты (такие, как тип данных) набора идентификаторов. Если объявление предполагает выделение памяти под объявленные объекты, то такое объявление называется *определением*.

В заголовочном файле `opend.h` мы объявляем три глобальные переменные с классом хранения `extern`. Эти объявления не подразумевают выделения памяти для хранения значений переменных. В файле `main.c` мы определяем три глобальные переменные. Иногда определение глобальной переменной может сопровождаться ее инициализацией, но мы, как правило, позволяем языку С инициализировать ее значением по умолчанию.

17.5 Обе функции, `select` и `poll`, возвращают количество дескрипторов, готовых к выполнению операции. Цикл обхода массива `client` может быть завершен раньше, когда число обработанных дескрипторов достигнет значения, полученного от функции `select` или `poll`.

17.6 Первая проблема заключается в состоянии гонки в интервале времени между вызовами `stat` и `unlink`, в течение которого файл может изменить-

ся. Вторая проблема проявляется, когда имя представляет символическую ссылку на файл сокета домена UNIX — функция `stat` сообщит, что это сокет (функция `stat` следует по символическим сылкам), но функция `unlink` удалит саму символическую ссылку, а не файл сокета. Решить последнюю проблему можно, заменив вызов функции `stat` вызовом `lstat`, но это не решает первую проблему.

17.7 Первый способ — отправить оба дескриптора в одном управляющем сообщении. Все файловые дескрипторы хранятся в смежных областях памяти. Это демонстрирует следующий код:

```
struct msghdr msg;
struct cmsghdr *cmptr;
int *ip;

if ((cmptr = calloc(1, CMSG_LEN(2*sizeof(int)))) == NULL)
    err_sys("ошибка вызова функции calloc");

msg.msg_control = cmptr;
msg.msg_controllen = CMSG_LEN(2*sizeof(int));
/* продолжение инициализации msghdr... */
cmptr->cmsg_len = CMSG_LEN(2*sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
ip = (int *)CMSG_DATA(cmptr);
*ip++ = fd1;
*ip = fd2;
```

Данный прием можно использовать на всех четырех платформах, обсуждаемых в этой книге. Второй способ — упаковать две отдельные структуры `cmsghdr` в одно сообщение:

```
struct msghdr msg;
struct cmsghdr *cmptr;

if ((cmptr = calloc(1, 2*CMSG_LEN(sizeof(int)))) == NULL)
    err_sys("ошибка вызова функции calloc");
msg.msg_control = cmptr;
msg.msg_controllen = 2*CMSG_LEN(sizeof(int));
/* продолжение инициализации msghdr... */
cmptr->cmsg_len = CMSG_LEN(sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmptr) = fd1;
cmptr = CMPTR_NXTHDR(&msg, cmptr);
cmptr->cmsg_len = CMSG_LEN(sizeof(int));
cmptr->cmsg_level = SOL_SOCKET;
cmptr->cmsg_type = SCM_RIGHTS;
*(int *)CMSG_DATA(cmptr) = fd2;
```

В отличие от первого, этот способ работает только в FreeBSD 8.0.

Глава 18

18.1 Обратите внимание: поскольку терминал находится в неканоническом режиме, ввод команды `reset` должен завершаться символом перевода строки, а не символом возврата каретки.

18.2 Она строит таблицу для каждого из 128 символов и затем устанавливает самый старший бит (бит четности) в соответствии с указаниями пользователя. После этого она использует 8-разрядный ввод/вывод, самостоятельно обслуживая бит четности.

18.3 Если вы используете терминал с оконной системой, вам не нужно входить в систему дважды. Вы можете проделать этот эксперимент в двух отдельных окнах. В Solaris запустите команду `stty -a`, перенаправив стандартный ввод окна, в котором запущен редактор `vi`. Это позволит увидеть, что `vi` устанавливает параметры `MIN` и `TIME` в значение 1. Вызов функции `read` будет ожидать ввода хотя бы одного символа, но когда символ будет введен, функция `read`, прежде чем вернуть управление, будет ждать ввода дополнительных символов не дольше одной десятой доли секунды.

Глава 19

19.1 Оба сервера, `telnetd` и `rlogind`, работают с привилегиями суперпользователя, поэтому могут без ограничений пользоваться функциями `chown` и `chmod`.

19.2 Запустите `pty -n stty -a`, чтобы предотвратить инициализацию структур `termios` и `winsize` подчиненного терминала.

19.4 К сожалению, команда `F_SETFL` функции `fcntl` не позволяет изменять состояние режима «для чтения и для записи».

19.5 Здесь присутствуют три группы процессов: (1) командная оболочка входа, (2) дочерний и родительский процессы программы `pty`, (3) процесс `cat`. Первые две группы составляют единый сеанс, в котором в качестве лидера выступает командная оболочка входа. Второй сеанс содержит только процесс `cat`. Первая группа процессов (командная оболочка входа) является группой процессов фонового режима, а две другие — группами процессов переднего плана.

19.6 Первым завершится процесс `cat`, когда получит от своего модуля дисциплины обслуживания терминала признак конца файла. Это приведет к завершению ведомого PTY, что вызовет завершение ведущего PTY. Это, в свою очередь, приведет к тому, что родительский процесс, который получает ввод от ведущего PTY, получит признак конца файла. Родительский процесс пошлет сигнал `SIGTERM` дочернему процессу, вследствие чего дочерний процесс прекратит работу. (Дочерний процесс не перехватывает этот сигнал.) И наконец, родительский процесс вызовет функцию `exit(0)` в конце функции `main`.

Ниже приводится вывод программы из листинга 8.16, соответствующий данному случаю.

```
cat     e =     270, chars =     274, stat =    0:
pty     e =     262, chars =      40, stat =   15: F     X
pty     e =     288, chars =     188, stat =    0:
```

19.7 Сделать это можно с помощью команд `echo` и `date`(1), запустив их в подоболочке:

```
#!/bin/sh
( echo "Сбор данных запущен " `date`;
  pty "${SHELL:/bin/sh}";
  echo " Сбор данных завершен " `date` ) | tee typescript
```

19.8 В модуле дисциплины обслуживания терминала, расположенном выше ведомого PTY, разрешен эхо-вывод, поэтому все, что читает PTY со своего стандартного ввода и записывает в ведущий PTY, по умолчанию выводится в виде эха. Эхо-вывод производится модулем дисциплины обслуживания терминала, расположенным выше подчиненного PTY, даже если программа (`ttyname`) не читает данные.

Глава 20

20.1 Наш консерватизм в установке блокировки в функции `_db_dodelete` обусловлен стремлением избежать состояния гонки в функции `db_nextrec`. Если вызов `_db_writedat` не будет защищен блокировкой, может возникнуть ситуация, когда запись с данными окажется стерта во время ее чтения функцией `db_nextrec`: функция `db_nextrec` может прочитать индексную запись, убедиться, что она не пуста, и приступить к чтению записи с данными, которая может быть стерта функцией `_db_dodelete` между вызовами `_db_readidx` и `_db_readdat` в `db_nextrec`.

20.2 Предположим, что `db_nextrec` вызывает `_db_readidx`, которая читает индекс в буфер процесса. Затем процесс приостанавливается ядром, и управление передается другому процессу. Другой процесс вызывает `db_delete` и удаляет запись, прочитанную первым процессом. Обе записи — ключ и данные — оказываются затертыми пробелами. Затем управление возвращается первому процессу, который вызывает `_db_readdat` (из `db_nextrec`) и читает запись с данными, затертую пробелами. Блокировка для чтения, устанавливаемая в `db_nextrec`, позволяет выполнить чтение индексной записи и записи с данными атомарно (относительно других процессов, использующих ту же самую базу данных).

20.3 Использование принудительных блокировок окажет влияние на другие читающие и пишущие процессы. Они заблокируются ядром, пока не будут сняты блокировки, установленные функциями `_db_writeidx` и `_db_writedat`.

20.5 Используя такой порядок записи (сначала данные, потом индекс), мы защищаем файлы базы данных от повреждения в случае, если процесс завершится между двумя операциями записи. Если процесс сначала запишет индексную запись и будет неожиданно завершен перед записью данных, мы

получим корректную индексную запись, которая указывает на некорректные данные.

Глава 21

21.5 Несколько подсказок. Проверять наличие заданий можно в двух местах: в очереди демона печати и во внутренней очереди сетевого принтера. Вы должны не допустить, чтобы один пользователь получил возможность отменить задание печати другого пользователя. Разумеется, суперпользователь должен иметь возможность отменить печать любого задания.

21.7 Этого не требуется, потому что демону не нужно повторно читать конфигурационный файл, пока не появится задание для печати. Функция `printer_thread` проверяет необходимость повторного чтения файла конфигурации перед каждой попыткой отправить задание принтеру.

21.9 Достаточно записать строку, оканчивающуюся нулевым байтом (не забывайте, что функция `strlen` не учитывает нулевой байт при определении длины строки). Существует два простых решения: либо добавлять 1 к счетчику записываемых байтов, либо использовать функцию `dprintf` вместо пары функций `sprintf` и `write`.

Список литературы

Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. 1986. «Mach: A New Kernel Foundation for UNIX Development», Proceedings of the 1986 Summer USENIX Conference, pp. 93–113, Atlanta, GA.

Введение в операционную систему Mach.

Adams, J., Bustos, D., Hahn, S., Powell, D., and Praza, L. 2005. «Solaris Service Management Facility: Modern System Startup and Administration», Proceedings of the 19th Large Installation System Administration Conference (LISA'05), pp. 225–236, San Diego, CA.

Описание механизма управления службами (Service Management Facility, SMF) в ОС Solaris, обеспечивающего основу для запуска и мониторинга администрируемых процессов, а также восстановления служб после сбоев.

Adobe Systems Inc. 1999. PostScript Language Reference Manual, Third Edition. Addison-Wesley, Reading, MA.

Справочное руководство по языку PostScript.

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1988. The AWK Programming Language. Addison-Wesley, Reading, MA.

Замечательная книга по языку программирования awk. Версия awk, описываемая в книге, иногда называется nawk («new awk»).

Andrade, J. M., Carges, M. T., and Kovach, K. R. 1989. «Building a Transaction Processing System on UNIX Systems», Proceedings of the 1989 USENIX Transaction Processing Workshop, vol. May, pp. 13–22, Pittsburgh, PA.

Описание системы обработки запросов AT&T Tuxedo.

Arnold, J. Q. 1986. «Shared Libraries on UNIX System V», Proceedings of the 1986 Summer USENIX Conference, pp. 395–404, Atlanta, GA.

Описание реализации разделяемых библиотек в SVR3.

AT&T. 1989. System V Interface Definition, Third Edition. Addison-Wesley, Reading, MA.

Этот четырехтомник описывает интерфейсы исходного кода System V и ее поведение во время выполнения. Третья редакция соответствует SVR4. Пятый том содержит обновленные версии команд и функций из томов 1–4, был издан в 1991 году. В настоящее время тираж распродан.

AT&T. 1990a. UNIX Research System Programmer's Manual, Tenth Edition, Volume I. Saunders College Publishing, Fort Worth, TX.

Версия «Руководства программиста UNIX» для 10-й редакции Research UNIX System (V10). В этой книге содержатся традиционные для UNIX страницы справочного руководства (разделы 1–9).

AT&T. 1990b. UNIX Research System Papers, Tenth Edition, Volume II. Saunders College Publishing, Fort Worth, TX.

Том II руководства программиста для UNIX Version 10 (V10) содержит 40 статей, описывающих различные аспекты системы.

AT&T. 1990c. UNIX System V Release 4 BSD/XENIX Compatibility Guide. Prentice Hall, Englewood Cliffs, NJ.

Содержит страницы справочного руководства, описывающие библиотеку совместимости.

AT&T. 1990d. UNIX System V Release 4 Programmer's Guide: STREAMS. Prentice Hall, Englewood Cliffs, NJ.

Описывает систему STREAMS в SVR4.

AT&T. 1990e. UNIX System V Release 4 Programmer's Reference Manual. Prentice Hall, Englewood Cliffs, NJ.

Справочное руководство программиста к реализации SVR4 для процессора Intel 80386. Содержит разделы 1 (команды), 2 (системные вызовы), 3 (подпрограммы), 4 (форматы файлов) и 5 (различные возможности).

AT&T. 1991. UNIX System V Release 4 System Administrator's Reference Manual. Prentice Hall, Englewood Cliffs, NJ.

Справочное руководство системного администратора к реализации SVR4 для процессора Intel 80386. Содержит разделы 1 (команды), 4 (форматы файлов), 5 (различные возможности) и 7 (специальные файлы).

Bach, M. J. 1986. The Design of the UNIX Operating System. Prentice Hall, Englewood Cliffs, NJ.

Книга подробно описывает архитектуру и реализацию операционной системы UNIX. Хотя исходный код UNIX и не приводится (поскольку в то время он был собственностью AT&T), все же в книге представлено большое количество алгоритмов и структур данных, используемых ядром UNIX. Эта книга описывает SVR2.

Bolsky, M. I., and Korn, D. G. 1995. The New KornShell Command and Programming Language, Second Edition. Prentice Hall, Englewood Cliffs, NJ.

Книга описывает работу с командной оболочкой Korn shell — как с командным интерпретатором, так и с языком программирования.

Bovet, D. P. и Cesati, M. Understanding the Linux Kernel, Third Edition. O'Reilly Media, Sebastopol, CA.[1]

Chen, D., Barkley, R. E., and Lee, T. P. 1990. «Insuring Improved VM Performance: Some No-Fault Policies», Proceedings of the 1990 Winter USENIX Conference, pp. 11–22, Washington, D.C.

Описывает изменения, внесенные в реализацию виртуальной памяти SVR4 для повышения производительности (главным образом функций `fork` и `exec`).

Comer, D. E. 1979. «The Ubiquitous B-Tree», ACM Computing Surveys, vol. 11, no. 2, pp. 121–137 (June).

Хорошая подробная статья о двоичных деревьях.

Date, C. J. 2004. An Introduction to Database Systems, Eighth Edition. Addison-Wesley, Boston, MA.[2]

Обширный обзор систем управления базами данных.

Evans, J. 2006. «A Scalable Concurrent malloc Implementation for FreeBSD», Proceedings of BSDCan.

Описание реализации `jemalloc` — библиотеки для работы с динамической памятью, используемой в ОС FreeBSD.

Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R. 1979. «Extendible Hashing — A Fast Access Method for Dynamic Files», ACM Transactions on Databases, vol. 4, no. 3, pp. 315–344 (September).

Статья, описывающая методику расширяемого хеширования.

Fowler, G. S., Korn, D. G., and Vo, K. P. 1989. «An Efficient File Hierarchy Walker», Proceeding of the 1989 Summer USENIX Conference, pp. 173–188, Baltimore, MD.

Описывает альтернативную библиотеку функций для обхода дерева каталогов файловой системы.

Gallmeister, B. O. 1995. POSIX.4: Programming for the Real World. O'Reilly & Associates, Sebastopol, CA.

Описывает интерфейсы реального времени стандарта POSIX.

Garfinkel, S., Spafford, G., and Schwartz, A. 2003. Practical UNIX & Internet Security, Third Edition. O'Reilly & Associates, Sebastopol, CA.

Подробная книга о безопасности операционной системы UNIX.

[1] *Даниель Бовет, М. Чезати.* Ядро Linux. 3-е изд. BHV-Санкт-Петербург, 2007, ISBN: 978-5-94157-957-0.

[2] *К. Дж. Дейт.* Введение в системы баз данных. 8-е изд. Вильямс, 2006, ISBN: 5-8459-0788-8.

Ghemawat, S., and Menage, P. 2005. «TCMalloc: Thread-Caching Malloc».

Краткое описание диспетчера памяти TCMalloc, разработанного в компании Google. Доступно также в электронном виде по адресу *http://goog-perftools.sourceforge.net/doc/tcmalloc.html*.

Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. 1987. «Shared Libraries in SunOS», Proceedings of the 1987 Summer USENIX Conference, pp. 131–145, Phoenix, AZ.

Описывает реализацию разделяемых библиотек в SunOS.

Gingell, R. A., Moran, J. P., and Shannon, W. A. 1987. «Virtual Memory Architecture in SunOS», Proceedings of the 1987 Summer USENIX Conference, pp. 81–94, Phoenix, AZ.

Описывает первоначальную реализацию функции `mmap` и проблемы, связанные с архитектурой виртуальной памяти.

Goodheart, B. 1991. UNIX Curses Explained. Prentice Hall, Englewood Cliffs, NJ.

Полное руководство по `terminfo` и библиотеке `curses`. В настоящее время тираж распродан.

Hume, A. G. 1988. «A Tale of Two Greps», Software Practice and Experience, vol. 18, no. 11, pp. 1063–1072.

Интересная статья, в которой обсуждается вопрос повышения производительности утилиты `grep`.

IEEE. 1990. Information Technology – Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]. IEEE (Dec.).

Это был первый из стандартов POSIX, и он определял стандартные системные интерфейсы языка программирования C на основе ОС UNIX. Нередко он называется POSIX.1. В настоящее время входит в состав стандарта Single UNIX Specification, опубликованного The Open Group [2008].

ISO. 1999. International Standard ISO/IEC 9899 — Programming Language C. ISO/IEC.

Официальный стандарт языка программирования C и его библиотек. В 2011 году вышла новая версия стандарта, тем не менее все системы, описываемые в этой книге, все еще соответствуют версии стандарта 1999 года.

Электронную версию стандарта в формате PDF можно получить по адресам *http://www.ansi.org* и *http://www.iso.org*.

ISO. 2011. International Standard ISO/IEC 9899, Information Technology — Programming Languages — C. ISO/IEC.

Последняя версия официального стандарта языка программирования C и его библиотек. Эта версия пришла на смену версии 1999 года.

Электронную версию стандарта в формате PDF можно получить по адресам *http://www.ansi.org* и *http://www.iso.org*.

Kernighan, B. W., and Pike, R. 1984. The UNIX Programming Environment. Prentice Hall, Englewood Cliffs, NJ.[1]

Общее руководство по программированию в UNIX. Книга охватывает множество команд и утилит UNIX, таких как `grep`, `sed`, `awk` и `Bourne shell`.

Kernighan, B. W., and Ritchie, D. M. 1988. The C Programming Language, Second Edition. Prentice Hall, Englewood Cliffs, NJ.[2]

Книга о версии ANSI языка программирования C. Приложение B содержит описание библиотек, определяемых стандартом ANSI.

Kerrisk, M. 2010. The Linux Programming Interface. No Starch Press, San Francisco, CA.

Если данная книга показалась вам слишком длинной, то эта книга вдвое больше, но она описывает только программные интерфейсы Linux.

Kleiman, S. R. 1986. «Vnodes: An Architecture for Multiple File System Types in Sun Unix», Proceedings of the 1986 Summer USENIX Conference, pp. 238–247, Atlanta, GA.

Описание оригинальной реализации концепции виртуальных узлов.

Knuth, D. E. 1998. The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, Boston, MA.[3]

Описывает алгоритмы сортировки и поиска.

Korn, D. G., and Vo, K. P. 1991. «SFIO: Safe/Fast String/File IO», Proceedings of the 1991 Summer USENIX Conference, pp. 235–255, Nashville, TN.

Описание альтернативной библиотеки ввода/вывода. Библиотека доступна по адресу *http://www.research.att.com/sw/tools/sfio*.

Krieger, O., Stumm, M., and Unrau, R. 1992. «Exploiting the Advantages of Mapped Files for Stream I/O», Proceedings of the 1992 Winter USENIX Conference, pp. 27–42, San Francisco, CA.

Альтернатива стандартной библиотеке ввода/вывода, основанная на отображаемых файлах.

[1] *Б. Керниган, Р. Пайк*. UNIX. Программное окружение. Символ-Плюс, 2012, ISBN: 5-93286-029-4.

[2] *Б. Керниган, Д. Ритчи*. Язык программирования Си». 2-е изд. Вильямс, 2017, ISBN: 978-5-8459-1874-1, 0-13-110362-8.

[3] *Дональд Э*. Кнут. Искусство программирования. Т. 3. Сортировка и поиск». 2-е изд. Вильямс, 2014, ISBN: 978-5-8459-0082-1, 0-201-89685-0.

Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. 1989. The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley, Reading, MA.

Книга целиком посвящена операционной системе 4.3BSD. Описывает версию Tahoe 4.3BSD. В настоящее время тираж распродан.

Lennert, D. 1987. «How to Write a UNIX Daemon», ;login:, vol. 12, no. 4, pp. 17–23 (July/August).

Рассказывает о написании демонов для UNIX.

Libes, D. 1990. «expect: Curing Those Uncontrollable Fits of Interaction», Proceedings of the 1990 Summer USENIX Conference, pp. 183–192, Anaheim, CA.

Описание программы `expect` и ее реализации.

Libes, D. 1991. «expect: Scripts for Controlling Interactive Processes», Computing Systems, vol. 4, no. 2, pp. 99–125 (Spring).

В статье представлены многочисленные сценарии для программы `expect`.

Libes, D. 1994. Exploring Expect. O'Reilly & Associates, Sebastopol, CA.

Книга по работе с программой `expect`.

Lions, J. 1977. A Commentary on the UNIX Operating System. AT&T Bell Laboratories, Murray Hill, NJ.

Описывает исходные тексты 6-й редакции UNIX (6th Edition UNIX System). Доступна только для специалистов и служащих AT&T, хотя некоторые копии просочились за пределы AT&T.

Lions, J. 1996. Lions' Commentary on UNIX 6th Edition. Peer-to-Peer Communications, San Jose, CA.

Общедоступная версия классического издания 1977 года описывает 6-ю редакцию ОС UNIX.

Litwin, W. 1980. «Linear Hashing: A New Tool for File and Table Addressing», Proceedings of the 6th International Conference on Very Large Databases, pp. 212–223, Montreal, Canada.

Статья, описывающая метод линейного хеширования.

McKusick, M. K., Bostic, K., Karels, M. J., and Quarterman, J. S. 1996. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, Reading, MA.

Книга целиком посвящена операционной системе 4.4BSD.

McKusick, M. K., and Neville-Neil, G. V. 2005. The Design and Implementation of the FreeBSD Operating System. Addison-Wesley, Boston, MA.[1]

Книга целиком посвящена операционной системе FreeBSD 5.2.

McDougall, R., and Mauro, J. 2007. Solaris Internals. Solaris 10 and OpenSolaris Kernel Architecture, Second Edition. Prentice Hall, Upper Saddle River, NJ.

Книга о внутреннем устройстве операционной системы Solaris 10. Охватывает также версию OpenSolaris.

Morris, R., and Thompson, K. 1979. «UNIX Password Security», Communications of the ACM, vol. 22, no. 11, pp. 594–597 (Nov.).

Описание истории развития схемы паролей, используемой в системах UNIX.

Nemeth, E., Snyder, G., Seebass, S., and Hein, T. R. 2001. UNIX System Administration Handbook, Third Edition. Prentice Hall, Upper Saddle River, NJ.[2]

Книга, в которой подробно рассматривается администрирование UNIX.

The Open Group. 2008. The Single UNIX Specification, Version 4. The Open Group, Berkshire, UK.

Стандарты POSIX и X/Open, объединенные в один справочник.

Электронную версию в формате HTML можно найти по адресу *http://www.opengroup.org*.

Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. 1995. «Plan 9 from Bell Labs», Plan 9 Programmer's Manual Volume 2. AT&T, Reading, MA.

Описание операционной системы Plan 9, разработанной в том же подразделении, что и система UNIX.

Plauger, P. J. 1992. The Standard C Library. Prentice Hall, Englewood Cliffs, NJ.

Книга о библиотеке ANSI C. Содержит полную реализацию библиотеки языка C.

Presotto, D. L., and Ritchie, D. M. 1990. «Interprocess Communication in the Ninth Edition UNIX System», Software Practice and Experience, vol. 20, no. S1, pp. S1/3–S1/17 (June).

Описывает возможности IPC, предоставляемые 9-й редакцией UNIX (Ninth Edition Research UNIX System), разработанной в AT&T Bell Laboratories. Функциональные возможности основаны на потоковой системе ввода/вывода и включают дуплексные каналы, передачу файловых дескрипторов между про-

[1] *Маршалл К. Маккузик, Джордж В.* Невилл-Нил. FreeBSD: архитектура и реализация. КУДИЦ-Образ, 2006, ISBN: 5-9579-0103-2.

[2] *Э. Немет, Г. Снайдер, Т. Р. Хейн.* UNIX: Руководство системного администратора. Для профессионалов / 4-е изд. Вильямс, 2017, ISBN: 978-5-8459-2006-5.

цессами и создание уникальных соединений между клиентами и серверами. Копия этой статьи имеется также в [8].

Rago, S. A. 1993. UNIX System V Network Programming. Addison-Wesley, Reading, MA.

Книга описывает программирование в сетевом окружении UNIX System V Release 4, основанное на использовании механизмов STREAMS.

Raymond, E. S., ed. 1996. The New Hacker's Dictionary, Third Edition. MIT Press, Cambridge, MA.

Определения множества терминов из лексикона хакера.

Salus, P. H. 1994. A Quarter Century of UNIX. AddisonWesley, Reading, MA.

История развития UNIX с 1969 по 1994 год.

Seltzer, M., and Olson, M. 1992. «LIBTP: Portable Modular Transactions for UNIX», Proceedings of the 1992 Winter USENIX Conference, pp. 9–25, San Francisco, CA.

Модификация библиотеки `db`(3) из 4.4BSD, которая реализует механизм транзакций.

Seltzer, M., and Yigit, O. 1991. «A New Hashing Package for UNIX», Proceedings of the 1991 Winter USENIX Conference, pp. 173–184, Dallas, TX.

Описание библиотеки `dbm`(3) и ее реализации, а также новейшего пакета хеширования.

Singh, A. 2006. Mac OS X Internals: A Systems Approach. Addison-Wesley, Upper Saddle River, NJ.

Примерно 1600 страниц с описанием архитектуры операционной системы Mac OS X.

Stevens, W. R. 1990. UNIX Network Programming. Prentice Hall, Englewood Cliffs, NJ.[1]

Книга подробно описывает программирование сетевых приложений для UNIX. Первое издание очень сильно отличается по своему содержанию от более поздних изданий.

Stevens, W. R., Fenner, B., and Rudoff, A. M. 2004. UNIX Network Programming, Volume 1, Third Edition. Addison-Wesley, Boston, MA.[2]

Подробно описывается программирование сетевых приложений для UNIX. Переработана и разбита на два тома во втором издании, дополнена в третьем.

[1] *Стивенс У.* UNIX: Разработка сетевых приложений. Питер, 2003, ISBN: 5-318-00535-7.

[2] *Стивенс У., Феннер Б., Рудофф Э.* UNIX: Разработка сетевых приложений. Питер, 2006, ISBN: 5-94723-991-4.

Stonebraker, M. R. 1981. «Operating System Support for Database Management», Communications of the ACM, vol. 24, no. 7, pp. 412–418 (July).

Описывает службы операционной системы и их влияние на работу базы данных.

Strang, J. 1986. Programming with curses. O'Reilly & Associates, Sebastopol, CA.

Книга о версии библиотеки `curses` из Беркли.

Strang, J., Mui, L., and O'Reilly, T. 1988. termcap & terminfo, Third Edition. O'Reilly & Associates, Sebastopol, CA.

Книга посвящена `termcap` и `terminfo`.

Sun Microsystems. 2005. STREAMS Programming Guide. Sun Microsystems, Santa Clara, CA.

Описывает STREAMS программирование на платформе Solaris.

Thompson, K. 1978. «UNIX Implementation», The Bell System Technical Journal, vol. 57, no. 6, pp. 1931–1946 (July–Aug.).

Описывает некоторые аспекты реализации Version 7.

Vo, Kiem-Phong. 1996. «Vmalloc: A General and Efficient Memory Allocator», Software Practice and Experience, vol. 26, no. 3, pp. 357–374.

Описывает гибкий диспетчер динамической памяти.

Wei, J., and Pu, C. 2005. «TOCTTOU Vulnerabilities in UNIX_Style File Systems: An Anatomical Study», Proceedings of the 4th USENIX Conference on File and Storage Technologoes (FAST'05), pp. 155–167, San Francisco, CA.

Описывает недостатки TOCTTOU в интерфейсе файловой системы UNIX.

Weinberger, P. J. 1982. «Making UNIX Operating Systems Safe for Databases», The Bell System Technical Journal, vol. 61, no. 9, pp. 2407–2422 (Nov.).

Описывает некоторые проблемы реализации баз данных в ранних версиях UNIX.

Weinstock, C. B., and Wulf, W. A. 1988. «Quick Fit: An Efficient Algorithm for Heap Storage Allocation», SIGPLAN Notices, vol. 23, no. 10, pp. 141–148.

Описывает алгоритм управления динамической памятью, который подходит для широкого круга приложений.

Williams, T. 1989. «Session Management in System V Release 4», Proceedings of the 1989 Winter USENIX Conference, pp. 365–375, San Diego, CA.

Описывает архитектуру сеанса в SVR4, на которой были основаны интерфейсы POSIX.1. Рассматриваются группы процессов, управление заданиями, управляющие терминалы и вопросы безопасности существующих механизмов.

X/Open. 1989. X/Open Portability Guide. Prentice Hall, Englewood Cliffs, NJ.

Издание состоит из семи томов, которые охватывают команды и утилиты (том 1), системные интерфейсы и заголовочные файлы (том 2), дополнительные определения (том 3), языки программирования (том 4), управление данными (том 5), управление окнами (том 6), сетевые службы (том 7). Хотя это издание в настоящее время отсутствует в продаже, его заменяет Single UNIX Specification [Open Group, 2008].