Appendix A

530128
END

Function Prototypes

This appendix contains the function prototypes for the standard ISO C, POSIX, and UNIX System functions described in the text. Often, we want to see only the arguments to a function ("Which argument is the file pointer for `fgets`?") or only the return value ("Does `sprintf` return a pointer or a count?"). These prototypes also show which headers need to be included to obtain the definitions of any special constants and to obtain the ISO C function prototype to help detect any compile-time errors.

The page number reference for each function prototype appears to the right of the first header file listed for the function. The page number reference gives the page containing the prototype for the function. That page should be consulted for additional information on the function.

Some functions are supported by only a few of the platforms described in this text. In addition, some platforms support function flags that other platforms don't support. In these cases, we usually list the platforms for which support is provided. In a few cases, however, we list platforms that lack support.

```
void      abort(void);
```
<stdlib.h> p. 365
This function never returns

```
int       accept(int sockfd, struct sockaddr *restrict addr,
              socklen_t *restrict len);
```
<sys/socket.h> p. 608
Returns: file (socket) descriptor if OK, –1 on error

```
int       access(const char *path, int mode);
```
<unistd.h> p. 102
mode: R_OK, W_OK, X_OK, F_OK
Returns: 0 if OK, –1 on error

```
int       aio_cancel(int fd, struct aiocb *aiocb);
```
<aio.h> p. 514
Returns: AIO_ALLDONE, AIO_CANCELED, AIO_NOTCANCELED, or –1 on error

```
int       aio_error(const struct aiocb *aiocb);
```
<aio.h> p. 513
Returns: 0 if operation succeeded, EINPROGRESS if operation is still in progress, error code if operation failed, or –1 on error

```
int       aio_fsync(int op, struct aiocb *aiocb);
```
<aio.h> p. 513
Returns: 0 if OK, –1 on error

```
int       aio_read(struct aiocb *aiocb);
```
<aio.h> p. 512
Returns: 0 if OK, –1 on error

```
ssize_t   aio_return(const struct aiocb *aiocb);
```
<aio.h> p. 513
Returns: result of asynchronous operation, –1 on error

```
int       aio_suspend(const struct aiocb *const list[], int nent,
                      const struct timespec *timeout);
```
<aio.h> p. 514
Returns: 0 if OK, –1 on error

```
int       aio_write(struct aiocb *aiocb);
```
<aio.h> p. 512
Returns: 0 if OK, –1 on error

```
unsigned
int       alarm(unsigned int seconds);
```
<unistd.h> p. 338
Returns: 0 or number of seconds until previously set alarm

```
int       atexit(void (*func)(void));
```
<stdlib.h> p. 200
Returns: 0 if OK, nonzero on error

```
int        bind(int sockfd, const struct sockaddr *addr, socklen_t len);
                    <sys/socket.h>                                          p. 604
                    Returns: 0 if OK, –1 on error

void      *calloc(size_t nobj, size_t size);
                    <stdlib.h>                                              p. 207
                    Returns: non-null pointer if OK, NULL on error

speed_t    cfgetispeed(const struct termios *termptr);
                    <termios.h>                                             p. 692
                    Returns: baud rate value

speed_t    cfgetospeed(const struct termios *termptr);
                    <termios.h>                                             p. 692
                    Returns: baud rate value

int        cfsetispeed(struct termios *termptr, speed_t speed);
                    <termios.h>                                             p. 692
                    Returns: 0 if OK, –1 on error

int        cfsetospeed(struct termios *termptr, speed_t speed);
                    <termios.h>                                             p. 692
                    Returns: 0 if OK, –1 on error

int        chdir(const char *path);
                    <unistd.h>                                              p. 135
                    Returns: 0 if OK, –1 on error

int        chmod(const char *path, mode_t mode);
                    <sys/stat.h>                                            p. 106
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, –1 on error

int        chown(const char *path, uid_t owner, gid_t group);
                    <unistd.h>                                              p. 109
                    Returns: 0 if OK, –1 on error

void       clearerr(FILE *fp);
                    <stdio.h>                                               p. 151

int        clock_getres(clockid_t clock_id, struct timespec *tsp);
                    <sys/time.h>                                            p. 190
                    clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC,
                              CLOCK_PROCESS_CPUTIME_ID,
                              CLOCK_THREAD_CPUTIME_ID
                    Returns: 0 if OK, –1 on error
```

```
int       clock_gettime(clockid_t clock_id, struct timespec *tsp);
```
<sys/time.h> p. 189
clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID
Returns: 0 if OK, –1 on error

```
int       clock_nanosleep(clockid_t clock_id, int flags,
              const struct timespec *reqtp,
              struct timespec *remtp);
```
<time.h> p. 375
clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID
flags: TIMER_ABSTIME
Returns: 0 if slept for requested time, error number on failure

```
int       clock_settime(clockid_t clock_id, const struct timespec *tsp);
```
<sys/time.h> p. 190
clock_id: CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID
Returns: 0 if OK, –1 on error

```
int       close(int fd);
```
<unistd.h> p. 66
Returns: 0 if OK, –1 on error

```
int       closedir(DIR *dp);
```
<dirent.h> p. 130
Returns: 0 if OK, –1 on error

```
void      closelog(void);
```
<syslog.h> p. 470

```
unsigned
char      *CMSG_DATA(struct cmsghdr *cp);
```
<sys/socket.h> p. 645
Returns: pointer to data associated with cmsghdr structure

```
struct
cmsghdr   *CMSG_FIRSTHDR(struct msghdr *mp);
```
<sys/socket.h> p. 645
Returns: pointer to first cmsghdr structure associated with the msghdr structure, or NULL if none exists

```
unsigned
int       CMSG_LEN(unsigned int nbytes);
                    <sys/socket.h>                                    p. 645
                    Returns: size to allocate for data object nbytes large

struct
cmsghdr  *CMSG_NXTHDR(struct msghdr *mp, struct cmsghdr *cp);
                    <sys/socket.h>                                    p. 645
                    Returns: pointer to next cmsghdr structure associated with
                             the msghdr structure given the current cmsghdr
                             structure, or NULL if we're at the last one

int       connect(int sockfd, const struct sockaddr *addr,
                  socklen_t len);
                    <sys/socket.h>                                    p. 605
                    Returns: 0 if OK, -1 on error

int       creat(const char *path, mode_t mode);
                    <fcntl.h>                                         p. 66
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: file descriptor opened for write-only if OK,
                             -1 on error

char     *ctermid(char *ptr);
                    <stdio.h>                                         p. 694
                    Returns: pointer to name of controlling terminal on success,
                             pointer to empty string on error

int       dprintf(int fd, const char *restrict format, ...);
                    <stdio.h>                                         p. 159
                    Returns: number of characters output if OK, negative value
                             if output error

int       dup(int fd);
                    <unistd.h>                                        p. 79
                    Returns: new file descriptor if OK, -1 on error

int       dup2(int fd, int fd2);
                    <unistd.h>                                        p. 79
                    Returns: new file descriptor if OK, -1 on error

void      endgrent(void);
                    <grp.h>                                           p. 183

void      endhostent(void);
                    <netdb.h>                                         p. 597
```

```
void      endnetent(void);
                  <netdb.h>                                          p. 598

void      endprotoent(void);
                  <netdb.h>                                          p. 598

void      endpwent(void);
                  <pwd.h>                                            p. 180

void      endservent(void);
                  <netdb.h>                                          p. 599

void      endspent(void);
                  <shadow.h>                                         p. 182
                  Platforms: Linux 3.2.0, Solaris 10

int       execl(const char *path, const char *arg0, ...
                /* (char *) 0 */ );
                  <unistd.h>                                         p. 249
                  Returns: -1 on error, no return on success

int       execle(const char *path, const char *arg0, ... /* (char *) 0,
                 char *const envp[] */ );
                  <unistd.h>                                         p. 249
                  Returns: -1 on error, no return on success

int       execlp(const char *filename, const char *arg0, ...
                 /* (char *) 0 */ );
                  <unistd.h>                                         p. 249
                  Returns: -1 on error, no return on success

int       execv(const char *path, char *const argv[]);
                  <unistd.h>                                         p. 249
                  Returns: -1 on error, no return on success

int       execve(const char *path, char *const argv[],
                 char *const envp[]);
                  <unistd.h>                                         p. 249
                  Returns: -1 on error, no return on success

int       execvp(const char *filename, char *const argv[]);
                  <unistd.h>                                         p. 249
                  Returns: -1 on error, no return on success

void      _Exit(int status);
                  <stdlib.h>                                         p. 198
                  This function never returns
```

```
void      _exit(int status);
                  <unistd.h>                                    p. 198
                  This function never returns

void      exit(int status);
                  <stdlib.h>                                    p. 198
                  This function never returns

int       faccessat(int fd, const char *path, int mode, int flag);
                  <unistd.h>                                    p. 102
                  mode: R_OK, W_OK, X_OK, F_OK
                  flag: AT_EACCESS
                  Returns: 0 if OK, -1 on error

int       fchdir(int fd);
                  <unistd.h>                                    p. 135
                  Returns: 0 if OK, -1 on error

int       fchmod(int fd, mode_t mode);
                  <sys/stat.h>                                  p. 106
                  mode: S_IS[UG]ID, S_ISVTX,
                        S_I[RWX](USR|GRP|OTH)
                  Returns: 0 if OK, -1 on error

int       fchmodat(int fd, const char *path, mode_t mode, int flag);
                  <sys/stat.h>                                  p. 106
                  mode: S_IS[UG]ID, S_ISVTX,
                        S_I[RWX](USR|GRP|OTH)
                  flag: AT_SYMLINK_NOFOLLOW
                  Returns: 0 if OK, -1 on error

int       fchown(int fd, uid_t owner, gid_t group);
                  <unistd.h>                                    p. 109
                  Returns: 0 if OK, -1 on error

int       fchownat(int fd, const char *path, uid_t owner,
                   gid_t group, int flag);
                  <unistd.h>                                    p. 109
                  flag: AT_SYMLINK_NOFOLLOW
                  Returns: 0 if OK, -1 on error

int       fclose(FILE *fp);
                  <stdio.h>                                     p. 150
                  Returns: 0 if OK, EOF on error
```

```
int     fcntl(int fd, int cmd, ... /* int arg */ );
                <fcntl.h>                                          p. 82
                cmd: F_DUPFD, F_DUPFD_CLOEXEC, F_GETFD,
                     F_SETFD, F_GETFL, F_SETFL, F_GETOWN,
                     F_SETOWN, F_GETLK, F_SETLK, F_SETLKW
                Returns: depends on cmd if OK, –1 on error

int     fdatasync(int fd);
                <unistd.h>                                         p. 81
                Returns: 0 if OK, –1 on error
                Platforms: Linux 3.2.0, Solaris 10

void    FD_CLR(int fd, fd_set *fdset);
                <sys/select.h>                                     p. 503

int     FD_ISSET(int fd, fd_set *fdset);
                <sys/select.h>                                     p. 503
                Returns: nonzero if fd is in set, 0 otherwise

FILE   *fdopen(int fd, const char *type);
                <stdio.h>                                          p. 148
                type: "r", "w", "a", "r+", "w+", "a+"
                Returns: file pointer if OK, NULL on error

DIR    *fdopendir(int fd);
                <dirent.h>                                         p. 130
                Returns: pointer if OK, NULL on error

void    FD_SET(int fd, fd_set *fdset);
                <sys/select.h>                                     p. 503

void    FD_ZERO(fd_set *fdset);
                <sys/select.h>                                     p. 503

int     feof(FILE *fp);
                <stdio.h>                                          p. 151
                Returns: nonzero (true) if end of file on stream,
                         0 (false) otherwise

int     ferror(FILE *fp);
                <stdio.h>                                          p. 151
                Returns: nonzero (true) if error on stream, 0 (false) otherwise

int     fexecve(int fd, char *const argv[], char *const envp[]);
                <unistd.h>                                         p. 249
                Returns: –1 on error, no return on success
```

```
int        fflush(FILE *fp);
                        <stdio.h>                                    p. 147
                        Returns: 0 if OK, EOF on error

int        fgetc(FILE *fp);
                        <stdio.h>                                    p. 150
                        Returns: next character if OK, EOF on end of file or error

int        fgetpos(FILE *restrict fp, fpos_t *restrict pos);
                        <stdio.h>                                    p. 158
                        Returns: 0 if OK, nonzero on error

char      *fgets(char *restrict buf, int n, FILE *restrict fp);
                        <stdio.h>                                    p. 152
                        Returns: buf if OK, NULL on end of file or error

int        fileno(FILE *fp);
                        <stdio.h>                                    p. 164
                        Returns: file descriptor associated with the stream, -1 on error

void       flockfile(FILE *fp);
                        <stdio.h>                                    p. 443

FILE      *fmemopen(void *restrict buf, size_t size,
                    const char *restrict type);
                        <stdio.h>                                    p. 171
                        type: "r", "w", "a", "r+", "w+", "a+"
                        Returns: stream pointer if OK, NULL on error

FILE      *fopen(const char *restrict path, const char *restrict type);
                        <stdio.h>                                    p. 148
                        type: "r", "w", "a", "r+", "w+", "a+"
                        Returns: file pointer if OK, NULL on error

pid_t      fork(void);
                        <unistd.h>                                   p. 229
                        Returns: 0 in child, process ID of child in parent, -1 on error

long       fpathconf(int fd, int name);
                        <unistd.h>                                   p. 42
                        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
                        Returns: corresponding value if OK, -1 on error
```

```
int       fprintf(FILE *restrict fp, const char *restrict format, ...);
                  <stdio.h>                                              p. 159
                  Returns: number of characters output if OK, negative value
                        if output error

int       fputc(int c, FILE *fp);
                  <stdio.h>                                              p. 152
                  Returns: c if OK, EOF on error

int       fputs(const char *restrict str, FILE *restrict fp);
                  <stdio.h>                                              p. 153
                  Returns: non-negative value if OK, EOF on error

size_t    fread(void *restrict ptr, size_t size, size_t nobj,
                FILE *restrict fp);
                  <stdio.h>                                              p. 156
                  Returns: number of objects read

void      free(void *ptr);
                  <stdlib.h>                                             p. 207

void      freeaddrinfo(struct addrinfo *ai);
                  <sys/socket.h>                                         p. 599
                  <netdb.h>

FILE     *freopen(const char *restrict path, const char *restrict type,
                  FILE *restrict fp);
                  <stdio.h>                                              p. 148
                  type: "r", "w", "a", "r+", "w+", "a+"
                  Returns: file pointer if OK, NULL on error

int       fscanf(FILE *restrict fp, const char *restrict format, ...);
                  <stdio.h>                                              p. 162
                  Returns: number of input items assigned, EOF if input
                        error or end of file before any conversion

int       fseek(FILE *fp, long offset, int whence);
                  <stdio.h>                                              p. 158
                  whence: SEEK_SET, SEEK_CUR, SEEK_END
                  Returns: 0 if OK, –1 on error

int       fseeko(FILE *fp, off_t offset, int whence);
                  <stdio.h>                                              p. 158
                  whence: SEEK_SET, SEEK_CUR, SEEK_END
                  Returns: 0 if OK, –1 on error
```

```
int       fsetpos(FILE *fp, const fpos_t *pos);
                  <stdio.h>                                          p. 158
                  Returns: 0 if OK, nonzero on error

int       fstat(int fd, struct stat *buf);
                  <sys/stat.h>                                       p. 93
                  Returns: 0 if OK, -1 on error

int       fstatat(int fd, const char *restrict path,
                  struct stat *restrict buf, int flag);
                  <sys/stat.h>                                       p. 93
                  flag: AT_SYMLINK_NOFOLLOW
                  Returns: 0 if OK, -1 on error

int       fsync(int fd);
                  <unistd.h>                                         p. 81
                  Returns: 0 if OK, -1 on error

long      ftell(FILE *fp);
                  <stdio.h>                                          p. 158
                  Returns: current file position indicator if OK, -1L on error

off_t     ftello(FILE *fp);
                  <stdio.h>                                          p. 158
                  Returns: current file position indicator if OK,
                        (off_t)-1 on error

key_t     ftok(const char *path, int id);
                  <sys/ipc.h>                                        p. 557
                  Returns: key if OK, (key_t)-1 on error

int       ftruncate(int fd, off_t length);
                  <unistd.h>                                         p. 112
                  Returns: 0 if OK, -1 on error

int       ftrylockfile(FILE *fp);
                  <stdio.h>                                          p. 443
                  Returns: 0 if OK, nonzero if lock can't be acquired

void      funlockfile(FILE *fp);
                  <stdio.h>                                          p. 443

int       futimens(int fd, const struct timespec times[2]);
                  <sys/stat.h>                                       p. 126
                  Returns: 0 if OK, -1 on error
```

```
int       fwide(FILE *fp, int mode);
                    <stdio.h>                                    p. 144
                    <wchar.h>
                    Returns: positive if stream is wide oriented, negative if stream
                             is byte oriented, or 0 if stream has no orientation

size_t    fwrite(const void *restrict ptr, size_t size, size_t nobj,
                 FILE *restrict fp);
                    <stdio.h>                                    p. 156
                    Returns: number of objects written

const
char     *gai_strerror(int error);
                    <netdb.h>                                    p. 600
                    Returns: a pointer to a string describing the error

int       getaddrinfo(const char *restrict host,
                      const char *restrict service,
                      const struct addrinfo *restrict hint,
                      struct addrinfo **restrict res);
                    <sys/socket.h>                               p. 599
                    <netdb.h>
                    Returns: 0 if OK, nonzero error code on error

int       getc(FILE *fp);
                    <stdio.h>                                    p. 150
                    Returns: next character if OK, EOF on end of file or error

int       getchar(void);
                    <stdio.h>                                    p. 150
                    Returns: next character if OK, EOF on end of file or error

int       getchar_unlocked(void);
                    <stdio.h>                                    p. 444
                    Returns: the next character if OK, EOF on end of file or error

int       getc_unlocked(FILE *fp);
                    <stdio.h>                                    p. 444
                    Returns: the next character if OK, EOF on end of file or error

char     *getcwd(char *buf, size_t size);
                    <unistd.h>                                   p. 136
                    Returns: buf if OK, NULL on error

gid_t     getegid(void);
                    <unistd.h>                                   p. 228
                    Returns: effective group ID of calling process
```

```
char     *getenv(const char *name);
                    <stdlib.h>                                    p. 210
                    Returns: pointer to value associated with name,
                             NULL if not found

uid_t     geteuid(void);
                    <unistd.h>                                    p. 228
                    Returns: effective user ID of calling process

gid_t     getgid(void);
                    <unistd.h>                                    p. 228
                    Returns: real group ID of calling process

struct
group    *getgrent(void);
                    <grp.h>                                       p. 183
                    Returns: pointer if OK, NULL on error or end of file

struct
group    *getgrgid(gid_t gid);
                    <grp.h>                                       p. 182
                    Returns: pointer if OK, NULL on error

struct
group    *getgrnam(const char *name);
                    <grp.h>                                       p. 182
                    Returns: pointer if OK, NULL on error

int       getgroups(int gidsetsize, gid_t grouplist[]);
                    <unistd.h>                                    p. 184
                    Returns: number of supplementary group IDs if OK,
                             -1 on error

struct
hostent  *gethostent(void);
                    <netdb.h>                                     p. 597
                    Returns: pointer if OK, NULL on error

int       gethostname(char *name, int namelen);
                    <unistd.h>                                    p. 188
                    Returns: 0 if OK, -1 on error

char     *getlogin(void);
                    <unistd.h>                                    p. 275
                    Returns: pointer to string giving login name if OK,
                             NULL on error
```

```
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>                                    p. 600
                      <netdb.h>
                      flags: NI_DGRAM, NI_NAMEREQD, NI_NOFQDN,
                           NI_NUMERICHOST, NI_NUMERICSCOPE,
                           NI_NUMERICSERV
                      Returns: 0 if OK, nonzero on error

struct
netent    *getnetbyaddr(uint32_t net, int type);
                      <netdb.h>                                         p. 598
                      Returns: pointer if OK, NULL on error

struct
netent    *getnetbyname(const char *name);
                      <netdb.h>                                         p. 598
                      Returns: pointer if OK, NULL on error

struct
netent    *getnetent(void);
                      <netdb.h>                                         p. 598
                      Returns: pointer if OK, NULL on error

int       getopt(int argc, char * const argv[], const char *options);
                      <fcntl.h>                                         p. 662
                      extern int opterr, optind, optopt;
                      extern char *optarg;
                      Returns: the next option character, or -1 when all options
                           have been processed

int       getpeername(int sockfd, struct sockaddr *restrict addr,
                      socklen_t *restrict alenp);
                      <sys/socket.h>                                    p. 605
                      Returns: 0 if OK, -1 on error

pid_t     getpgid(pid_t pid);
                      <unistd.h>                                        p. 294
                      Returns: process group ID if OK, -1 on error

pid_t     getpgrp(void);
                      <unistd.h>                                        p. 293
                      Returns: process group ID of calling process
```

```
pid_t     getpid(void);
                    <unistd.h>                                    p. 228
                    Returns: process ID of calling process

pid_t     getppid(void);
                    <unistd.h>                                    p. 228
                    Returns: parent process ID of calling process

int       getpriority(int which, id_t who);
                    <sys/resource.h>                              p. 277
                    which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER
                    Returns: nice value between -NZERO and NZERO-1 if OK,
                             -1 on error

struct
protoent *getprotobyname(const char *name);
                    <netdb.h>                                     p. 598
                    Returns: pointer if OK, NULL on error

struct
protoent *getprotobynumber(int proto);
                    <netdb.h>                                     p. 598
                    Returns: pointer if OK, NULL on error

struct
protoent *getprotoent(void);
                    <netdb.h>                                     p. 598
                    Returns: pointer if OK, NULL on error

struct
passwd   *getpwent(void);
                    <pwd.h>                                       p. 180
                    Returns: pointer if OK, NULL on error or end of file

struct
passwd   *getpwnam(const char *name);
                    <pwd.h>                                       p. 179
                    Returns: pointer if OK, NULL on error

struct
passwd   *getpwuid(uid_t uid);
                    <pwd.h>                                       p. 179
                    Returns: pointer if OK, NULL on error
```

```
int       getrlimit(int resource, struct rlimit *rlptr);
                    <sys/resource.h>                                    p. 220
                    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)
                    Returns: 0 if OK, –1 on error

char      *gets(char *buf);
                    <stdio.h>                                           p. 152
                    Returns: buf if OK, NULL on end of file or error

struct
servent   *getservbyname(const char *name, const char *proto);
                    <netdb.h>                                           p. 599
                    Returns: pointer if OK, NULL on error

struct
servent   *getservbyport(int port, const char *proto);
                    <netdb.h>                                           p. 599
                    Returns: pointer if OK, NULL on error

struct
servent   *getservent(void);
                    <netdb.h>                                           p. 599
                    Returns: pointer if OK, NULL on error

pid_t     getsid(pid_t pid);
                    <unistd.h>                                          p. 296
                    Returns: session leader's process group ID if OK,
                             –1 on error

int       getsockname(int sockfd, struct sockaddr *restrict addr,
                      socklen_t *restrict alenp);
                    <sys/socket.h>                                      p. 605
                    Returns: 0 if OK, –1 on error
```

```
int        getsockopt(int sockfd, int level, int option, void *restrict val,
                      socklen_t *restrict lenp);
```
<sys/socket.h> p. 624
Returns: 0 if OK, –1 on error

```
struct
spwd      *getspent(void);
```
<shadow.h> p. 182
Returns: pointer if OK, NULL on error
Platforms: Linux 3.2.0, Solaris 10

```
struct
spwd      *getspnam(const char *name);
```
<shadow.h> p. 182
Returns: pointer if OK, NULL on error
Platforms: Linux 3.2.0, Solaris 10

```
int        gettimeofday(struct timeval *restrict tp,
                        void *restrict tzp);
```
<sys/time.h> p. 190
Returns: 0 always

```
uid_t      getuid(void);
```
<unistd.h> p. 228
Returns: real user ID of calling process

```
struct
tm        *gmtime(const time_t *calptr);
```
<time.h> p. 192
Returns: pointer to broken-down time, NULL on error

```
int        grantpt(int fd);
```
<stdlib.h> p. 723
Returns: 0 on success, –1 on error

```
uint32_t   htonl(uint32_t hostint32);
```
<arpa/inet.h> p. 594
Returns: 32-bit integer in network byte order

```
uint16_t   htons(uint16_t hostint16);
```
<arpa/inet.h> p. 594
Returns: 16-bit integer in network byte order

```
const
char     *inet_ntop(int domain, const void *restrict addr,
                    char *restrict str, socklen_t size);
                       <arpa/inet.h>                                    p. 596
                       Returns: pointer to address string on success, NULL on error

int      inet_pton(int domain, const char *restrict str,
                   void *restrict addr);
                       <arpa/inet.h>                                    p. 596
                       Returns: 1 on success, 0 if the format is invalid, or –1 on error

int      initgroups(const char *username, gid_t basegid);
                       <grp.h>    /* Linux & Solaris */                 p. 184
                       <unistd.h> /* FreeBSD & Mac OS X */
                       Returns: 0 if OK, –1 on error

int      ioctl(int fd, int request, ...);
                       <unistd.h>     /* System V */                    p. 87
                       <sys/ioctl.h>  /* BSD and Linux */
                       Returns: –1 on error, something else if OK

int      isatty(int fd);
                       <unistd.h>                                       p. 695
                       Returns: 1 (true) if terminal device, 0 (false) otherwise

int      kill(pid_t pid, int signo);
                       <signal.h>                                       p. 337
                       Returns: 0 if OK, –1 on error

int      lchown(const char *path, uid_t owner, gid_t group);
                       <unistd.h>                                       p. 109
                       Returns: 0 if OK, –1 on error

int      link(const char *existingpath, const char *newpath);
                       <unistd.h>                                       p. 116
                       Returns: 0 if OK, –1 on error

int      linkat(int efd, const char *existingpath, int nfd,
                const char *newpath, int flag);
                       <unistd.h>                                       p. 116
                       flag: AT_SYMLINK_NOFOLLOW
                       Returns: 0 if OK, –1 on error
```

```
int       lio_listio(int mode,
                     struct aiocb *restrict const list[restrict],
                     int nent, struct sigevent *restrict sigev);
                      <aio.h>                                          p. 515
                      mode: LIO_NOWAIT, LIO_WAIT
                      Returns: 0 if OK, –1 on error

int       listen(int sockfd, int backlog);
                      <sys/socket.h>                                   p. 608
                      Returns: 0 if OK, –1 on error

struct
tm       *localtime(const time_t *calptr);
                      <time.h>                                         p. 192
                      Returns: pointer to broken-down time, NULL on error

void      longjmp(jmp_buf env, int val);
                      <setjmp.h>                                       p. 215
                      This function never returns

off_t     lseek(int fd, off_t offset, int whence);
                      <unistd.h>                                       p. 67
                      whence: SEEK_SET, SEEK_CUR, SEEK_END
                      Returns: new file offset if OK, –1 on error

int       lstat(const char *restrict path,
                struct stat *restrict buf);
                      <sys/stat.h>                                     p. 93
                      Returns: 0 if OK, –1 on error

void     *malloc(size_t size);
                      <stdlib.h>                                       p. 207
                      Returns: non-null pointer if OK, NULL on error

int       mkdir(const char *path, mode_t mode);
                      <sys/stat.h>                                     p. 129
                      mode: S_IS[UG]ID, S_ISVTX,
                            S_I[RWX](USR|GRP|OTH)
                      Returns: 0 if OK, –1 on error

int       mkdirat(int fd, const char *path, mode_t mode);
                      <sys/stat.h>                                     p. 129
                      mode: S_IS[UG]ID, S_ISVTX,
                            S_I[RWX](USR|GRP|OTH)
                      Returns: 0 if OK, –1 on error
```

```
char     *mkdtemp(char *template);
                    <stdlib.h>                                    p. 169
                    Returns: pointer to directory name if OK, NULL on error

int      mkfifo(const char *path, mode_t mode);
                    <sys/stat.h>                                  p. 553
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, -1 on error

int      mkfifoat(int fd, const char *path, mode_t mode);
                    <sys/stat.h>                                  p. 553
                    mode: S_IS[UG]ID, S_ISVTX,
                          S_I[RWX](USR|GRP|OTH)
                    Returns: 0 if OK, -1 on error

int      mkstemp(char *template);
                    <stdlib.h>                                    p. 169
                    Returns: file descriptor if OK, -1 on error

time_t   mktime(struct tm *tmptr);
                    <time.h>                                      p. 192
                    Returns: calendar time if OK, -1 on error

void     *mmap(void *addr, size_t len, int prot, int flag, int fd,
              off_t off);
                    <sys/mman.h>                                  p. 525
                    prot: PROT_READ, PROT_WRITE, PROT_EXEC,
                          PROT_NONE
                    flag: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
                    Returns: starting address of mapped region if OK,
                             MAP_FAILED on error

int      mprotect(void *addr, size_t len, int prot);
                    <sys/mman.h>                                  p. 527
                    Returns: 0 if OK, -1 on error

int      msgctl(int msqid, int cmd, struct msqid_ds *buf);
                    <sys/msg.h>                                   p. 562
                    cmd: IPC_STAT, IPC_SET, IPC_RMID
                    Returns: 0 if OK, -1 on error

int      msgget(key_t key, int flag);
                    <sys/msg.h>                                   p. 562
                    flag: IPC_CREAT, IPC_EXCL
                    Returns: message queue ID if OK, -1 on error
```

```
ssize_t   msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
                  <sys/msg.h>                                          p. 564
                  flag: IPC_NOWAIT, MSG_NOERROR
                  Returns: size of data portion of message if OK, –1 on error

int       msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
                  <sys/msg.h>                                          p. 563
                  flag: IPC_NOWAIT
                  Returns: 0 if OK, –1 on error

int       msync(void *addr, size_t len, int flags);
                  <sys/mman.h>                                         p. 528
                  flag: MS_ASYNC, MS_INVALIDATE, MS_SYNC
                  Returns: 0 if OK, –1 on error

int       munmap(void *addr, size_t len);
                  <sys/mman.h>                                         p. 528
                  Returns: 0 if OK, –1 on error

int       nanosleep(const struct timespec *reqtp,
                    struct timespec *remtp);
                  <time.h>                                             p. 374
                  Returns: 0 if slept for requested time, –1 on error

int       nice(int incr);
                  <unistd.h>                                           p. 276
                  Returns: new nice value – NZERO if OK, –1 on error

uint32_t  ntohl(uint32_t netint32);
                  <arpa/inet.h>                                        p. 594
                  Returns: 32-bit integer in host byte order

uint16_t  ntohs(uint16_t netint16);
                  <arpa/inet.h>                                        p. 594
                  Returns: 16-bit integer in host byte order

int       open(const char *path, int oflag, ... /* mode_t mode */ );
                  <fcntl.h>                                            p. 62
                  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)
                  Returns: file descriptor if OK, –1 on error
                  Platforms: O_FSYNC flag on FreeBSD 8.0 and Mac OS X 10.6.8
```

```
int       openat(int fd, const char *path, int oflag, ...
                  /* mode_t mode */ );
                        <fcntl.h>                                   p. 62
                        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)
                        Returns: file descriptor if OK, –1 on error
                        Platforms: O_FSYNC flag on FreeBSD 8.0 and Mac OS X 10.6.8

DIR       *opendir(const char *path);
                        <dirent.h>                                  p. 130
                        Returns: pointer if OK, NULL on error

void      openlog(const char *ident, int option, int facility);
                        <syslog.h>                                  p. 470
                        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>                                   p. 173
                        Returns: stream pointer if OK, NULL on error

FILE      *open_wmemstream(wchar_t **bufp, size_t *sizep);
                        <wchar.h>                                   p. 173
                        Returns: stream pointer if OK, NULL on error

long      pathconf(const char *path, int name);
                        <unistd.h>                                  p. 42
                        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
                        Returns: corresponding value if OK, –1 on error

int       pause(void);
                        <unistd.h>                                  p. 338
                        Returns: –1 with errno set to EINTR
```

```
int       pclose(FILE *fp);
                  <stdio.h>                                          p. 541
                  Returns: termination status of popen cmdstring, –1 on error

void      perror(const char *msg);
                  <stdio.h>                                          p. 15

int       pipe(int fd[2]);
                  <unistd.h>                                         p. 535
                  Returns: 0 if OK, –1 on error

int       poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
                  <poll.h>                                           p. 506
                  Returns: count of ready descriptors, 0 on timeout, –1 on error

FILE     *popen(const char *cmdstring, const char *type);
                  <stdio.h>                                          p. 541
                  type: "r", "w"
                  Returns: file pointer if OK, NULL on error

int       posix_openpt(int oflag);
                  <stdlib.h>                                         p. 722
                  <fcntl.h>
                  oflag: O_RWDR, O_NOCTTY
                  Returns: file descriptor of next available PTY master if OK,
                           –1 on error

ssize_t   pread(int fd, void *buf, size_t nbytes, off_t offset);
                  <unistd.h>                                         p. 78
                  Returns: number of bytes read, 0 if end of file, –1 on error

int       printf(const char *restrict format, ...);
                  <stdio.h>                                          p. 159
                  Returns: number of characters output if OK, negative value
                           if output error

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>                                     p. 506
                  Returns: count of ready descriptors, 0 on timeout, –1 on error

void      psiginfo(const siginfo_t *info, const char *msg);
                  <signal.h>                                         p. 379
```

```
void      psignal(int signo, const char *msg);
                  <signal.h>                                        p. 379
                  <siginfo.h>    /* on Solaris */

int       pthread_atfork(void (*prepare)(void), void (*parent)(void),
                     void (*child)(void));
                  <pthread.h>                                       p. 458
                  Returns: 0 if OK, error number on failure

int       pthread_attr_destroy(pthread_attr_t *attr);
                  <pthread.h>                                       p. 427
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                 int *detachstate);
                  <pthread.h>                                       p. 428
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getguardsize(const pthread_attr_t
                                   *restrict attr,
                                 size_t *restrict guardsize);
                  <pthread.h>                                       p. 430
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getstack(const pthread_attr_t *restrict attr,
                           void **restrict stackaddr,
                           size_t *restrict stacksize);
                  <pthread.h>                                       p. 429
                  Returns: 0 if OK, error number on failure

int       pthread_attr_getstacksize(const pthread_attr_t
                                   *restrict attr,
                                 size_t *restrict stacksize);
                  <pthread.h>                                       p. 430
                  Returns: 0 if OK, error number on failure

int       pthread_attr_init(pthread_attr_t *attr);
                  <pthread.h>                                       p. 427
                  Returns: 0 if OK, error number on failure

int       pthread_attr_setdetachstate(pthread_attr_t *attr,
                                 int detachstate);
                  <pthread.h>                                       p. 428
                  detachstate: PTHREAD_CREATE_DETACHED,
                               PTHREAD_CREATE_JOINABLE
                  Returns: 0 if OK, error number on failure
```

```
int     pthread_attr_setguardsize(pthread_attr_t *attr,
                                  size_t guardsize);
```
<pthread.h> p. 430
Returns: 0 if OK, error number on failure

```
int     pthread_attr_setstack(const pthread_attr_t *attr,
                              void *stackaddr, size_t *stacksize);
```
<pthread.h> p. 429
Returns: 0 if OK, error number on failure

```
int     pthread_attr_setstacksize(pthread_attr_t *attr,
                                  size_t stacksize);
```
<pthread.h> p. 430
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```
<pthread.h> p. 441
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_getpshared(const pthread_barrierattr_t
                                       *restrict attr,
                                       int *restrict pshared);
```
<pthread.h> p. 441
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_init(pthread_barrierattr_t *attr);
```
<pthread.h> p. 441
Returns: 0 if OK, error number on failure

```
int     pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                       int pshared);
```
<pthread.h> p. 441
pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED
Returns: 0 if OK, error number on failure

```
int     pthread_barrier_destroy(pthread_barrier_t *barrier);
```
<pthread.h> p. 418
Returns: 0 if OK, error number on failure

```
int     pthread_barrier_init(pthread_barrier_t *restrict barrier,
                             const pthread_barrierattr_t *
                               restrict attr,
                             unsigned int count);
```
<pthread.h> p. 418
Returns: 0 if OK, error number on failure

```
int     pthread_barrier_wait(pthread_barrier_t *barrier);
```
<pthread.h> p. 419

Returns: 0 or PTHREAD_BARRIER_SERIAL_THREAD if OK, error number on failure

```
int     pthread_cancel(pthread_t tid);
```
<pthread.h> p. 393

Returns: 0 if OK, error number on failure

```
void    pthread_cleanup_pop(int execute);
```
<pthread.h> p. 394

```
void    pthread_cleanup_push(void (*rtn)(void *), void *arg);
```
<pthread.h> p. 394

```
int     pthread_condattr_destroy(pthread_condattr_t *attr);
```
<pthread.h> p. 440

Returns: 0 if OK, error number on failure

```
int     pthread_condattr_getclock(const pthread_condattr_t
                                  *restrict attr,
                                clockid_t *restrict clock_id);
```
<pthread.h> p. 441

Returns: 0 if OK, error number on failure

```
int     pthread_condattr_getpshared(const pthread_condattr_t
                                    *restrict attr,
                                  int *restrict pshared);
```
<pthread.h> p. 440

Returns: 0 if OK, error number on failure

```
int     pthread_condattr_init(pthread_condattr_t *attr);
```
<pthread.h> p. 440

Returns: 0 if OK, error number on failure

```
int     pthread_condattr_setclock(pthread_condattr_t *attr,
                                clockid_t clock_id);
```
<pthread.h> p. 441

Returns: 0 if OK, error number on failure

```
int     pthread_condattr_setpshared(pthread_condattr_t *attr,
                                  int pshared);
```
<pthread.h> p. 440

pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED

Returns: 0 if OK, error number on failure

```
int     pthread_cond_broadcast(pthread_cond_t *cond);
```
<pthread.h> p. 415
Returns: 0 if OK, error number on failure

```
int     pthread_cond_destroy(pthread_cond_t *cond);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_cond_init(pthread_cond_t *restrict cond,
                  const pthread_condattr_t *restrict attr);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_cond_signal(pthread_cond_t *cond);
```
<pthread.h> p. 415
Returns: 0 if OK, error number on failure

```
int     pthread_cond_timedwait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex,
                      const struct timespec
                        *restrict timeout);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_cond_wait(pthread_cond_t *restrict cond,
                  pthread_mutex_t *restrict mutex);
```
<pthread.h> p. 414
Returns: 0 if OK, error number on failure

```
int     pthread_create(pthread_t *restrict tidp,
                const pthread_attr_t *restrict attr,
                void *(*start_rtn)(void *),
                void *restrict arg);
```
<pthread.h> p. 385
Returns: 0 if OK, error number on failure

```
int     pthread_detach(pthread_t tid);
```
<pthread.h> p. 397
Returns: 0 if OK, error number on failure

```
int     pthread_equal(pthread_t tid1, pthread_t tid2);
```
<pthread.h> p. 385
Returns: nonzero if equal, 0 otherwise

```
void    pthread_exit(void *rval_ptr);
```
<pthread.h> p. 389

```
void     *pthread_getspecific(pthread_key_t key);
                    <pthread.h>                                          p. 449
                    Returns: thread-specific data value or NULL if no value has
                             been associated with the key

int      pthread_join(pthread_t thread, void **rval_ptr);
                    <pthread.h>                                          p. 389
                    Returns: 0 if OK, error number on failure

int      pthread_key_create(pthread_key_t *keyp,
                            void (*destructor)(void *));
                    <pthread.h>                                          p. 447
                    Returns: 0 if OK, error number on failure

int      pthread_key_delete(pthread_key_t key);
                    <pthread.h>                                          p. 448
                    Returns: 0 if OK, error number on failure

int      pthread_kill(pthread_t thread, int signo);
                    <signal.h>                                           p. 455
                    Returns: 0 if OK, error number on failure

int      pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
                    <pthread.h>                                          p. 431
                    Returns: 0 if OK, error number on failure

int      pthread_mutexattr_getpshared(const pthread_mutexattr_t
                                        *restrict attr,
                                      int *restrict pshared);
                    <pthread.h>                                          p. 431
                    Returns: 0 if OK, error number on failure

int      pthread_mutexattr_getrobust(const pthread_mutexattr_t
                                       *restrict attr,
                                     int *restrict robust);
                    <pthread.h>                                          p. 432
                    Returns: 0 if OK, error number on failure

int      pthread_mutexattr_gettype(const pthread_mutexattr_t
                                     *restrict attr,
                                   int *restrict type);
                    <pthread.h>                                          p. 434
                    Returns: 0 if OK, error number on failure

int      pthread_mutexattr_init(pthread_mutexattr_t *attr);
                    <pthread.h>                                          p. 431
                    Returns: 0 if OK, error number on failure
```

```
int     pthread_mutexattr_setpshared(pthread_mutexattr_t *attr,
                                     int pshared);
```
<pthread.h> p. 431

pshared: PTHREAD_PROCESS_PRIVATE,
PTHREAD_PROCESS_SHARED

Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_setrobust(pthread_mutexattr_t *attr,
                                    int robust);
```
<pthread.h> p. 432

robust: PTHREAD_MUTEX_ROBUST,
PTHREAD_MUTEX_STALLED

Returns: 0 if OK, error number on failure

```
int     pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```
<pthread.h> p. 434

type: PTHREAD_MUTEX_NORMAL,
PTHREAD_MUTEX_ERRORCHECK,
PTHREAD_MUTEX_RECURSIVE,
PTHREAD_MUTEX_DEFAULT

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_consistent(pthread_mutex_t * mutex);
```
<pthread.h> p. 433

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_destroy(pthread_mutex_t *mutex);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_init(pthread_mutex_t *restrict mutex,
                           const pthread_mutexattr_t *restrict attr);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_lock(pthread_mutex_t *mutex);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
                                const struct timespec *restrict tsptr);
```
<pthread.h> p. 407
<time.h>

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_trylock(pthread_mutex_t *mutex);
```
<pthread.h> p. 400

Returns: 0 if OK, error number on failure

```
int     pthread_mutex_unlock(pthread_mutex_t *mutex);
                <pthread.h>                                          p. 400
                Returns: 0 if OK, error number on failure

int     pthread_once(pthread_once_t *initflag, void (*initfn)(void));
                <pthread.h>                                          p. 448
                pthread_once_t initflag = PTHREAD_ONCE_INIT;
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
                <pthread.h>                                          p. 439
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_getpshared(const pthread_rwlockattr_t
                                        *restrict attr,
                                      int *restrict pshared);
                <pthread.h>                                          p. 440
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
                <pthread.h>                                          p. 439
                Returns: 0 if OK, error number on failure

int     pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                      int pshared);
                <pthread.h>                                          p. 440
                pshared: PTHREAD_PROCESS_PRIVATE,
                         PTHREAD_PROCESS_SHARED
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
                <pthread.h>                                          p. 409
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,
                            const pthread_rwlockattr_t
                              *restrict attr);
                <pthread.h>                                          p. 409
                Returns: 0 if OK, error number on failure

int     pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
                <pthread.h>                                          p. 410
                Returns: 0 if OK, error number on failure
```

```
int     pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                                   const struct timespec
                                     *restrict tsptr);
```
<pthread.h> p. 413
<time.h>
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock,
                                   const struct timespec
                                     *restrict tsptr);
```
<pthread.h> p. 413
<time.h>
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
int     pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```
<pthread.h> p. 410
Returns: 0 if OK, error number on failure

```
pthread_t pthread_self(void);
```
<pthread.h> p. 385
Returns: thread ID of the calling thread

```
int     pthread_setcancelstate(int state, int *oldstate);
```
<pthread.h> p. 451
state: PTHREAD_CANCEL_ENABLE,
PTHREAD_CANCEL_DISABLE
Returns: 0 if OK, error number on failure

```
int     pthread_setcanceltype(int type, int *oldtype);
```
<pthread.h> p. 453
type: PTHREAD_CANCEL_DEFERRED,
PTHREAD_CANCEL_ASYNCHRONOUS
Returns: 0 if OK, error number on failure

```
int     pthread_setspecific(pthread_key_t key, const void *value);
                <pthread.h>                                          p. 449
                Returns: 0 if OK, error number on failure

int     pthread_sigmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
                <signal.h>                                           p. 454
                how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
                Returns: 0 if OK, error number on failure

int     pthread_spin_destroy(pthread_spinlock_t *lock);
                <pthread.h>                                          p. 417
                Returns: 0 if OK, error number on failure

int     pthread_spin_init(pthread_spinlock_t *lock, int pshared);
                <pthread.h>                                          p. 417
                pshared: PTHREAD_PROCESS_PRIVATE,
                         PTHREAD_PROCESS_SHARED
                Returns: 0 if OK, error number on failure

int     pthread_spin_lock(pthread_spinlock_t *lock);
                <pthread.h>                                          p. 418
                Returns: 0 if OK, error number on failure

int     pthread_spin_trylock(pthread_spinlock_t *lock);
                <pthread.h>                                          p. 418
                Returns: 0 if OK, error number on failure

int     pthread_spin_unlock(pthread_spinlock_t *lock);
                <pthread.h>                                          p. 418
                Returns: 0 if OK, error number on failure

void    pthread_testcancel(void);
                <pthread.h>                                          p. 453

char   *ptsname(int fd);
                <stdlib.h>                                           p. 723
                Returns: pointer to name of PTY slave if OK, NULL on error

int     putc(int c, FILE *fp);
                <stdio.h>                                            p. 152
                Returns: c if OK, EOF on error

int     putchar(int c);
                <stdio.h>                                            p. 152
                Returns: c if OK, EOF on error
```

```
int       putchar_unlocked(int c);
```
<stdio.h> p. 444
Returns: c if OK, EOF on error

```
int       putc_unlocked(int c, FILE *fp);
```
<stdio.h> p. 444
Returns: c if OK, EOF on error

```
int       putenv(char *str);
```
<stdlib.h> p. 212
Returns: 0 if OK, nonzero on error

```
int       puts(const char *str);
```
<stdio.h> p. 153
Returns: non-negative value if OK, EOF on error

```
ssize_t   pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```
<unistd.h> p. 78
Returns: number of bytes written if OK, –1 on error

```
int       raise(int signo);
```
<signal.h> p. 337
Returns: 0 if OK, nonzero on error

```
ssize_t   read(int fd, void *buf, size_t nbytes);
```
<unistd.h> p. 71
Returns: number of bytes read if OK, 0 if end of file, –1 on error

```
struct
dirent    *readdir(DIR *dp);
```
<dirent.h> p. 130
Returns: pointer if OK, NULL at end of directory or error

```
ssize_t   readlink(const char *restrict path, char *restrict buf,
                   size_t bufsize);
```
<unistd.h> p. 123
Returns: number of bytes read if OK, –1 on error

```
ssize_t   readlinkat(int fd, const char* restrict path,
                     char *restrict buf, size_t bufsize);
```
<unistd.h> p. 123
Returns: number of bytes read if OK, –1 on error

```
ssize_t   readv(int fd, const struct iovec *iov, int iovcnt);
```
<sys/uio.h> p. 521
Returns: number of bytes read if OK, 0 if end of file, –1 on error

```
void     *realloc(void *ptr, size_t newsize);
```
<stdlib.h> p. 207

Returns: non-null pointer if OK, NULL on error

```
ssize_t  recv(int sockfd, void *buf, size_t nbytes, int flags);
```
<sys/socket.h> p. 612

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)

Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or –1 on error

```
ssize_t  recvfrom(int sockfd, void *restrict buf, size_t len, int flags,
             struct sockaddr *restrict addr,
             socklen_t *restrict addrlen);
```
<sys/socket.h> p. 613

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)

Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or –1 on error

```
ssize_t  recvmsg(int sockfd, struct msghdr *msg, int flags);
```
<sys/socket.h> p. 613

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)

Returns: length of message in bytes, 0 if no messages are available and peer has done an orderly shutdown, or –1 on error

```
int      remove(const char *path);
```
<stdio.h> p. 119

Returns: 0 if OK, –1 on error

```
int      rename(const char *oldname, const char *newname);
```
<stdio.h> p. 119

Returns: 0 if OK, –1 on error

```
int     renameat(int oldfd, const char *oldname, int newfd,
                 const char *newname);
```
<stdio.h> p. 119
Returns: 0 if OK, –1 on error

```
void    rewind(FILE *fp);
```
<stdio.h> p. 158

```
void    rewinddir(DIR *dp);
```
<dirent.h> p. 130

```
int     rmdir(const char *path);
```
<unistd.h> p. 130
Returns: 0 if OK, –1 on error

```
int     scanf(const char *restrict format, ...);
```
<stdio.h> p. 162
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
void    seekdir(DIR *dp, long loc);
```
<dirent.h> p. 130

```
int     select(int maxfdp1, fd_set *restrict readfds,
               fd_set *restrict writefds, fd_set *restrict exceptfds,
               struct timeval *restrict tvptr);
```
<sys/select.h> p. 502
Returns: count of ready descriptors, 0 on timeout, –1 on error

```
int     sem_close(sem_t *sem);
```
<semaphore.h> p. 580
Returns: 0 if OK, –1 on error

```
int     semctl(int semid, int semnum, int cmd, ...
               /* union semun arg */ );
```
<sys/sem.h> p. 567
cmd: IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT, GETZCNT, GETVAL, SETVAL, GETALL, SETALL
Returns: (depends on command), –1 on error

```
int     sem_destroy(sem_t *sem);
```
<semaphore.h> p. 582
Returns: 0 if OK, –1 on error

```
int       semget(key_t key, int nsems, int flag);
                  <sys/sem.h>                                          p. 567
                  flag: IPC_CREAT, IPC_EXCL
                  Returns: semaphore ID if OK, –1 on error

int       sem_getvalue(sem_t *restrict sem, int *restrict valp);
                  <semaphore.h>                                        p. 582
                  Returns: 0 if OK, –1 on error

int       sem_init(sem_t *sem, int pshared, unsigned int value);
                  <semaphore.h>                                        p. 582
                  Returns: 0 if OK, –1 on error

int       semop(int semid, struct sembuf semoparray[], size_t nops);
                  <sys/sem.h>                                          p. 568
                  Returns: 0 if OK, –1 on error

sem_t    *sem_open(const char *name, int oflag, ... /* mode_t mode,
                  unsigned int value */ );
                  <semaphore.h>                                        p. 579
                  flag: IPC_CREAT, IPC_EXCL
                  Returns: pointer to semaphore if OK, SEM_FAILED on error

int       sem_post(sem_t *sem);
                  <semaphore.h>                                        p. 582
                  Returns: 0 if OK, –1 on error

int       sem_timedwait(sem_t *restrict sem,
                    const struct timespec *restrict tsptr);
                  <semaphore.h>                                        p. 581
                  <time.h>
                  Returns: 0 if OK, –1 on error

int       sem_trywait(sem_t *sem);
                  <semaphore.h>
                  Returns: 0 if OK, –1 on error

int       sem_unlink(const char *name);
                  <semaphore.h>                                        p. 580
                  Returns: 0 if OK, –1 on error

int       sem_wait(sem_t *sem);
                  <semaphore.h>                                        p. 581
                  Returns: 0 if OK, –1 on error
```

```
ssize_t   send(int sockfd, const void *buf, size_t nbytes, int flags);
                    <sys/socket.h>                                          p. 610
                    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)
                    Returns: number of bytes sent if OK, –1 on error

ssize_t   sendmsg(int sockfd, const struct msghdr *msg, int flags);
                    <sys/socket.h>                                          p. 611
                    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)
                    Returns: number of bytes sent if OK, –1 on error

ssize_t   sendto(int sockfd, const void *buf, size_t nbytes, int flags,
                 const struct sockaddr *destaddr, socklen_t destlen);
                    <sys/socket.h>                                          p. 610
                    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)
                    Returns: number of bytes sent if OK, –1 on error

void      setbuf(FILE *restrict fp, char *restrict buf);
                    <stdio.h>                                               p. 146

int       setegid(gid_t gid);
                    <unistd.h>                                              p. 258
                    Returns: 0 if OK, –1 on error

int       setenv(const char *name, const char *value, int rewrite);
                    <stdlib.h>                                              p. 212
                    Returns: 0 if OK, –1 on error
```

```
int     seteuid(uid_t uid);
                <unistd.h>                                          p. 258
                Returns: 0 if OK, –1 on error

int     setgid(gid_t gid);
                <unistd.h>                                          p. 256
                Returns: 0 if OK, –1 on error

void    setgrent(void);
                <grp.h>                                             p. 183

int     setgroups(int ngroups, const gid_t grouplist[]);
                <grp.h>     /* Linux */                             p. 184
                <unistd.h> /* FreeBSD, Mac OS X, and Solaris */
                Returns: 0 if OK, –1 on error

void    sethostent(int stayopen);
                <netdb.h>                                           p. 597

int     setjmp(jmp_buf env);
                <setjmp.h>                                          p. 215
                Returns: 0 if called directly, nonzero if returning from
                    a call to longjmp

int     setlogmask(int maskpri);
                <syslog.h>                                          p. 470
                Returns: previous log priority mask value

void    setnetent(int stayopen);
                <netdb.h>                                           p. 598

int     setpgid(pid_t pid, pid_t pgid);
                <unistd.h>                                          p. 294
                Returns: 0 if OK, –1 on error

int     setpriority(int which, id_t who, int value);
                <sys/resource.h>                                    p. 277
                which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER
                Returns: 0 if OK, –1 on error

void    setprotoent(int stayopen);
                <netdb.h>                                           p. 598

void    setpwent(void);
                <pwd.h>                                             p. 180
```

```
int     setregid(gid_t rgid, gid_t egid);
                <unistd.h>                                          p. 257
                Returns: 0 if OK, –1 on error

int     setreuid(uid_t ruid, uid_t euid);
                <unistd.h>                                          p. 257
                Returns: 0 if OK, –1 on error

int     setrlimit(int resource, const struct rlimit *rlptr);
                <sys/resource.h>                                    p. 220
                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)
                Returns: 0 if OK, –1 on error

void    setservent(int stayopen);
                <netdb.h>                                           p. 599

pid_t   setsid(void);
                <unistd.h>                                          p. 295
                Returns: process group ID if OK, –1 on error

int     setsockopt(int sockfd, int level, int option, const void *val,
                   socklen_t len);
                <sys/socket.h>                                      p. 624
                Returns: 0 if OK, –1 on error

void    setspent(void);
                <shadow.h>                                          p. 182
                Platforms: Linux 3.2.0, Solaris 10

int     setuid(uid_t uid);
                <unistd.h>                                          p. 256
                Returns: 0 if OK, –1 on error
```

```
int       setvbuf(FILE *restrict fp, char *restrict buf, int mode,
                  size_t size);
                      <stdio.h>                                          p. 146
                      mode: _IOFBF, _IOLBF, _IONBF
                      Returns: 0 if OK, nonzero on error

void     *shmat(int shmid, const void *addr, int flag);
                      <sys/shm.h>                                        p. 574
                      flag: SHM_RND, SHM_RDONLY
                      Returns: pointer to shared memory segment if OK, -1 on error

int       shmctl(int shmid, int cmd, struct shmid_ds *buf);
                      <sys/shm.h>                                        p. 573
                      cmd: IPC_STAT, IPC_SET, IPC_RMID,
                           SHM_LOCK (Linux 3.2.0, Solaris 10),
                           SHM_UNLOCK (Linux 3.2.0, Solaris 10)
                      Returns: 0 if OK, -1 on error

int       shmdt(const void *addr);
                      <sys/shm.h>                                        p. 574
                      Returns: 0 if OK, -1 on error

int       shmget(key_t key, size_t size, int flag);
                      <sys/shm.h>                                        p. 572
                      flag: IPC_CREAT, IPC_EXCL
                      Returns: non-negative shared memory ID if OK, -1 on error

int       shutdown(int sockfd, int how);
                      <sys/socket.h>                                     p. 592
                      how: SHUT_RD, SHUT_WR, SHUT_RDWR
                      Returns: 0 if OK, -1 on error

int       sig2str(int signo, char *str);
                      <signal.h>                                         p. 380
                      Returns: 0 if OK, -1 on error
                      Platforms: Solaris 10

int       sigaction(int signo, const struct sigaction *restrict act,
                    struct sigaction *restrict oact);
                      <signal.h>                                         p. 350
                      Returns: 0 if OK, -1 on error

int       sigaddset(sigset_t *set, int signo);
                      <signal.h>                                         p. 344
                      Returns: 0 if OK, -1 on error
```

```
int     sigdelset(sigset_t *set, int signo);
```
<signal.h> p. 344
Returns: 0 if OK, –1 on error

```
int     sigemptyset(sigset_t *set);
```
<signal.h> p. 344
Returns: 0 if OK, –1 on error

```
int     sigfillset(sigset_t *set);
```
<signal.h> p. 344
Returns: 0 if OK, –1 on error

```
int     sigismember(const sigset_t *set, int signo);
```
<signal.h> p. 344
Returns: 1 if true, 0 if false, –1 on error

```
void    siglongjmp(sigjmp_buf env, int val);
```
<setjmp.h> p. 356
This function never returns

```
void   (*signal(int signo, void (*func)(int)))(int);
```
<signal.h> p. 323
Returns: previous disposition of signal if OK, SIG_ERR on error

```
int     sigpending(sigset_t *set);
```
<signal.h> p. 347
Returns: 0 if OK, –1 on error

```
int     sigprocmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```
<signal.h> p. 346
how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
Returns: 0 if OK, –1 on error

```
int     sigqueue(pid_t pid, int signo, const union sigval value)
```
<signal.h> p. 376
Returns: 0 if OK, –1 on error

```
int     sigsetjmp(sigjmp_buf env, int savemask);
```
<setjmp.h> p. 356
Returns: 0 if called directly, nonzero if returning from a call to siglongjmp

```
int       sigsuspend(const sigset_t *sigmask);
```
<signal.h> p. 359
Returns: –1 with errno set to EINTR

```
int       sigwait(const sigset_t *restrict set, int *restrict signop);
```
<signal.h> p. 454
Returns: 0 if OK, error number on failure

```
unsigned
int       sleep(unsigned int seconds);
```
<unistd.h> p. 373
Returns: 0 or number of unslept seconds

```
int       snprintf(char *restrict buf, size_t n,
                   const char *restrict format, ...);
```
<stdio.h> p. 159
Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

```
int       sockatmark(int sockfd);
```
<sys/socket.h> p. 626
Returns: 1 if at mark, 0 if not at mark, –1 on error

```
int       socket(int domain, int type, int protocol);
```
<sys/socket.h> p. 590
type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET
Returns: file (socket) descriptor if OK, –1 on error

```
int       socketpair(int domain, int type, int protocol, int sockfd[2]);
```
<sys/socket.h> p. 630
type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET
Returns: 0 if OK, –1 on error

```
int       sprintf(char *restrict buf, const char *restrict format, ...);
```
<stdio.h> p. 159
Returns: number of characters stored in array if OK, negative value if encoding error

```
int       sscanf(const char *restrict buf,
                 const char *restrict format, ...);
```
<stdio.h> p. 162
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
int       stat(const char *restrict path, struct stat *restrict buf);
                    <sys/stat.h>                                          p. 93
                    Returns: 0 if OK, –1 on error

int       str2sig(const char *str, int *signop);
                    <signal.h>                                            p. 380
                    Returns: 0 if OK, –1 on error
                    Platforms: Solaris 10

char     *strerror(int errnum);
                    <string.h>                                            p. 15
                    Returns: pointer to message string

size_t    strftime(char *restrict buf, size_t maxsize,
                   const char *restrict format,
                   const struct tm *restrict tmptr);
                    <time.h>                                              p. 192
                    Returns: number of characters stored in array if room,
                          0 otherwise

size_t    strftime_l(char *restrict buf, size_t maxsize,
                     const char *restrict format,
                     const struct tm *restrict tmptr, locale_t locale);
                    <time.h>                                              p. 192
                    Returns: number of characters stored in array if room,
                          0 otherwise

char     *strptime(const char *restrict buf, const char *restrict format,
                   struct tm *restrict tmptr);
                    <time.h>                                              p. 195
                    Returns: pointer to one character past last character parsed,
                          NULL otherwise

char     *strsignal(int signo);
                    <string.h>                                            p. 380
                    Returns: a pointer to a string describing the signal

int       symlink(const char *actualpath, const char *sympath);
                    <unistd.h>                                            p. 123
                    Returns: 0 if OK, –1 on error

int       symlinkat(const char *actualpath, int fd, const char *sympath);
                    <unistd.h>                                            p. 123
                    Returns: 0 if OK, –1 on error

void      sync(void);
                    <unistd.h>                                            p. 81
```

```
long      sysconf(int name);
                  <unistd.h>                                          p. 42
                  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
                  Returns: corresponding value if OK, –1 on error

void      syslog(int priority, char *format, ...);
                  <syslog.h>                                          p. 470

int       system(const char *cmdstring);
                  <stdlib.h>                                          p. 265
                  Returns: termination status of shell

int       tcdrain(int fd);
                  <termios.h>                                         p. 693
                  Returns: 0 if OK, –1 on error

int       tcflow(int fd, int action);
                  <termios.h>                                         p. 693
                  action: TCOOFF, TCOON, TCIOFF, TCION
                  Returns: 0 if OK, –1 on error

int       tcflush(int fd, int queue);
                  <termios.h>                                         p. 693
                  queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
                  Returns: 0 if OK, –1 on error
```

```
int        tcgetattr(int fd, struct termios *termptr);
```
<termios.h> p. 683
Returns: 0 if OK, –1 on error

```
pid_t      tcgetpgrp(int fd);
```
<unistd.h> p. 298
Returns: process group ID of foreground process group if OK, –1 on error

```
pid_t      tcgetsid(int fd);
```
<termios.h> p. 299
Returns: session leader's process group ID if OK, –1 on error

```
int        tcsendbreak(int fd, int duration);
```
<termios.h> p. 693
Returns: 0 if OK, –1 on error

```
int        tcsetattr(int fd, int opt, const struct termios *termptr);
```
<termios.h> p. 683
opt: TCSANOW, TCSADRAIN, TCSAFLUSH
Returns: 0 if OK, –1 on error

```
int        tcsetpgrp(int fd, pid_t pgrpid);
```
<unistd.h> p. 298
Returns: 0 if OK, –1 on error

```
long       telldir(DIR *dp);
```
<dirent.h> p. 130
Returns: current location in directory associated with *dp*

```
time_t     time(time_t *calptr);
```
<time.h> p. 189
Returns: value of time if OK, –1 on error

```
clock_t    times(struct tms *buf);
```
<sys/times.h> p. 280
Returns: elapsed wall clock time in clock ticks if OK, –1 on error

```
FILE       *tmpfile(void);
```
<stdio.h> p. 167
Returns: file pointer if OK, NULL on error

```
char       *tmpnam(char *ptr);
```
<stdio.h> p. 167
Returns: pointer to unique pathname, NULL on error

```
int       truncate(const char *path, off_t length);
                  <unistd.h>                                          p. 112
                  Returns: 0 if OK, –1 on error

char     *ttyname(int fd);
                  <unistd.h>                                          p. 695
                  Returns: pointer to pathname of terminal, NULL on error

mode_t    umask(mode_t cmask);
                  <sys/stat.h>                                        p. 104
                  Returns: previous file mode creation mask

int       uname(struct utsname *name);
                  <sys/utsname.h>                                     p. 187
                  Returns: non-negative value if OK, –1 on error

int       ungetc(int c, FILE *fp);
                  <stdio.h>                                           p. 151
                  Returns: c if OK, EOF on error

int       unlink(const char *path);
                  <unistd.h>                                          p. 117
                  Returns: 0 if OK, –1 on error

int       unlinkat(int fd, const char *path, int flag);
                  <unistd.h>                                          p. 117
                  flag: AT_REMOVEDIR
                  Returns: 0 if OK, –1 on error

int       unlockpt(int fd);
                  <stdlib.h>                                          p. 723
                  Returns: 0 on success, –1 on error

int       unsetenv(const char *name);
                  <stdlib.h>                                          p. 212
                  Returns: 0 if OK, –1 on error

int       utimensat(int fd, const char *path,
                    const struct timespec times[2], int flag);
                  <sys/stat.h>                                        p. 126
                  flag: AT_SYMLINK_NOFOLLOW
                  Returns: 0 if OK, –1 on error

int       utimes(const char *path, const struct timeval times[2]);
                  <sys/time.h>                                        p. 127
                  Returns: 0 if OK, –1 on error
```

```
int     vdprintf(int fd, const char *restrict format, va_list arg);
                <stdarg.h>                                          p. 161
                <stdio.h>
```
Returns: number of characters output if OK, negative value if output error

```
int     vfprintf(FILE *restrict fp, const char *restrict format,
                va_list arg);
                <stdarg.h>                                          p. 161
                <stdio.h>
```
Returns: number of characters output if OK, negative value if output error

```
int     vfscanf(FILE *restrict fp, const char *restrict format,
                va_list arg);
                <stdarg.h>                                          p. 163
                <stdio.h>
```
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
int     vprintf(const char *restrict format, va_list arg);
                <stdarg.h>                                          p. 161
                <stdio.h>
```
Returns: number of characters output if OK, negative value if output error

```
int     vscanf(const char *restrict format, va_list arg);
                <stdarg.h>                                          p. 163
                <stdio.h>
```
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
int     vsnprintf(char *restrict buf, size_t n,
                const char *restrict format, va_list arg);
                <stdarg.h>                                          p. 161
                <stdio.h>
```
Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

```
int     vsprintf(char *restrict buf, const char *restrict format,
                va_list arg);
                <stdarg.h>                                          p. 161
                <stdio.h>
```
Returns: number of characters stored in array if OK, negative value if encoding error

```
int       vsscanf(const char *restrict buf, const char *restrict format,
                  va_list arg);
```
<stdarg.h> p. 163
<stdio.h>
Returns: number of input items assigned, EOF if input error or end of file before any conversion

```
void      vsyslog(int priority, const char *format, va_list arg);
```
<syslog.h> p. 472
<stdarg.h>
Platforms: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10

```
pid_t     wait(int *statloc);
```
<sys/wait.h> p. 238
Returns: process ID if OK, 0, or –1 on error

```
int       waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```
<sys/wait.h> p. 244
idtype: P_PID, P_PGID, P_ALL
options: WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED
Returns: 0 if OK, –1 on error
Platforms: Linux 3.2.0, Solaris 10

```
pid_t     waitpid(pid_t pid, int *statloc, int options);
```
<sys/wait.h> p. 238
options: WCONTINUED, WNOHANG, WUNTRACED
Returns: process ID if OK, 0, or –1 on error

```
pid_t     wait3(int *statloc, int options, struct rusage *rusage);
```
<sys/types.h> p. 245
<sys/wait.h>
<sys/time.h>
<sys/resource.h>
options: WNOHANG, WUNTRACED
Returns: process ID if OK, 0, or –1 on error
Platforms: 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> p. 245
<sys/wait.h>
<sys/time.h>
<sys/resource.h>
options: WNOHANG, WUNTRACED
Returns: process ID if OK, 0, or –1 on error
Platforms: 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>                                    p. 72
                    Returns: number of bytes written if OK, –1 on error

ssize_t   writev(int fd, const struct iovec *iov, int iovcnt);
                    <sys/uio.h>                                   p. 521
                    Returns: number of bytes written if OK, –1 on error
```

This page intentionally left blank

Appendix B

Miscellaneous Source Code

B.1 Our Header File

Most programs in the text include the header `apue.h`, shown in Figure B.1. It defines constants (such as `MAXLINE`) and prototypes for our own functions.

Most programs need to include the following headers: `<stdio.h>`, `<stdlib.h>` (for the `exit` function prototype), and `<unistd.h>` (for all the standard UNIX function prototypes). So our header automatically includes these system headers, along with `<string.h>`. This also reduces the size of all the program listings in the text.

```
/*
 * Our own header, to be included before all standard system headers.
 */
#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>      /* some systems still require this */
#include <sys/stat.h>
#include <sys/termios.h>    /* for winsize */
```

```
#if defined(MACOS) || !defined(TIOCGWINSZ)
#include <sys/ioctl.h>
#endif

#include <stdio.h>      /* for convenience */
#include <stdlib.h>     /* for convenience */
#include <stddef.h>     /* for offsetof */
#include <string.h>     /* for convenience */
#include <unistd.h>     /* for convenience */
#include <signal.h>     /* for SIG_ERR */

#define MAXLINE 4096            /* max line length */

/*
 * Default file access permissions for new files.
 */
#define FILE_MODE   (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

/*
 * Default permissions for new directories.
 */
#define DIR_MODE    (FILE_MODE | S_IXUSR | S_IXGRP | S_IXOTH)

typedef void    Sigfunc(int);   /* for signal handlers */

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

/*
 * Prototypes for our own functions.
 */
char    *path_alloc(size_t *);              /* Figure 2.16 */
long     open_max(void);                    /* Figure 2.17 */

int      set_cloexec(int);                  /* Figure 13.9 */
void     clr_fl(int, int);
void     set_fl(int, int);                  /* Figure 3.12 */

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

void     pr_mask(const char *);             /* Figure 10.14 */
Sigfunc *signal_intr(int, Sigfunc *);       /* Figure 10.19 */

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

void     sleep_us(unsigned int);            /* Exercise 14.5 */
ssize_t  readn(int, void *, size_t);        /* Figure 14.24 */
ssize_t  writen(int, const void *, size_t); /* Figure 14.24 */

int      fd_pipe(int *);                    /* Figure 17.2 */
int      recv_fd(int, ssize_t (*func)(int,
                const void *, size_t));     /* Figure 17.14 */
```

```
int      send_fd(int, int);                  /* Figure 17.13 */
int      send_err(int, int,
                  const char *);             /* Figure 17.12 */
int      serv_listen(const char *);          /* Figure 17.8 */
int      serv_accept(int, uid_t *);          /* Figure 17.9 */
int      cli_conn(const char *);             /* Figure 17.10 */
int      buf_args(char *, int (*func)(int,
                  char **));                 /* Figure 17.23 */

int      tty_cbreak(int);                    /* Figure 18.20 */
int      tty_raw(int);                       /* Figure 18.20 */
int      tty_reset(int);                     /* Figure 18.20 */
void     tty_atexit(void);                   /* Figure 18.20 */
struct termios  *tty_termios(void);          /* Figure 18.20 */

int      ptym_open(char *, int);             /* Figure 19.9 */
int      ptys_open(char *);                  /* Figure 19.9 */
#ifdef  TIOCGWINSZ
pid_t    pty_fork(int *, char *, int, const struct termios *,
                  const struct winsize *);   /* Figure 19.10 */
#endif

int     lock_reg(int, int, int, off_t, int, off_t); /* Figure 14.5 */

#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);     /* Figure 14.6 */

#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 *, ...);         /* Appendix B */
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 *, ...);         /* Appendix B */
```

```
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);        /* parent/child from Section 8.9 */
void    TELL_PARENT(pid_t);
void    TELL_CHILD(pid_t);
void    WAIT_PARENT(void);
void    WAIT_CHILD(void);

#endif  /* _APUE_H */
```

Figure B.1 Our header: `apue.h`

The reasons we include our header before all the normal system headers are to allow us to define anything that might be required by headers before they are included, to control the order in which header files are included, and to allow us to redefine anything that needs to be fixed up to hide the differences between systems.

B.2 Standard Error Routines

Two sets of error functions are used in most of the examples throughout the text to handle error conditions. One set begins with `err_` and outputs an error message to standard error. The other set begins with `log_` and is intended for daemon processes (Chapter 13) that probably have no controlling terminal.

The reason for defining our own error functions is to let us write our error handling with a single line of C code, as in

```
if (error condition)
        err_dump(printf format with any number of arguments);
```

instead of

```
if (error condition) {
        char  buf[200];

        sprintf(buf, printf format with any number of arguments);
        perror(buf);
        abort();
}
```

Our error functions use the variable-length argument list facility from ISO C. See Section 7.3 of Kernighan and Ritchie [1988] for additional details. Be aware that this ISO C facility differs from the `varargs` facility provided by earlier systems (such as SVR3 and 4.3BSD). The names of the macros are the same, but the arguments to some of the macros have changed.

Figure B.2 summarizes the differences between the various error functions.

Function	Adds string from strerror?	Parameter to strerror	Terminate?
err_dump	yes	errno	abort();
err_exit	yes	explicit parameter	exit(1);
err_msg	no		return;
err_quit	no		exit(1);
err_ret	yes	errno	return;
err_sys	yes	errno	exit(1);
err_cont	yes	explicit parameter	return;
log_msg	no		return;
log_quit	no		exit(2);
log_ret	yes	errno	return;
log_sys	yes	errno	exit(2);
log_exit	yes	explicit parameter	exit(2);

Figure B.2 Our standard error functions

Figure B.3 shows the error functions that output to standard error.

```
#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable aruments */

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

/*
 * Nonfatal error related to a system call.
 * Print a message and return.
 */
void
err_ret(const char *fmt, ...)
{
    va_list     ap;

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

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
void
err_sys(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
```

```
    va_end(ap);
    exit(1);
}

/*
 * Nonfatal error unrelated to a system call.
 * Error code passed as explict parameter.
 * Print a message and return.
 */
void
err_cont(int error, const char *fmt, ...)
{
    va_list     ap;

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

/*
 * Fatal error unrelated to a system call.
 * Error code passed as explict parameter.
 * Print a message and terminate.
 */
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);
}

/*
 * Fatal error related to a system call.
 * Print a message, dump core, and terminate.
 */
void
err_dump(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    err_doit(1, errno, fmt, ap);
    va_end(ap);
    abort();        /* dump core and terminate */
    exit(1);        /* shouldn't get here */
}

/*
 * Nonfatal error unrelated to a system call.
```

```
 * Print a message and return.
 */
void
err_msg(const char *fmt, ...)
{
    va_list     ap;

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

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
void
err_quit(const char *fmt, ...)
{
    va_list     ap;

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

/*
 * Print a message and return to caller.
 * Caller specifies "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);     /* in case stdout and stderr are the same */
    fputs(buf, stderr);
    fflush(NULL);       /* flushes all stdio output streams */
}
```

Figure B.3 Error functions that output to standard error

Figure B.4 shows the `log_XXX` error functions. These require the caller to define the variable `log_to_stderr` and set it nonzero if the process is not running as a daemon. In this case, the error messages are sent to standard error. If the `log_to_stderr` flag is 0, the `syslog` facility (Section 13.4) is used.

```
/*
 * Error routines for programs that can run as a daemon.
 */

#include "apue.h"
#include <errno.h>      /* for definition of errno */
#include <stdarg.h>     /* ISO C variable arguments */
#include <syslog.h>

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

/*
 * Caller must define and set this: nonzero if
 * interactive, zero if daemon
 */
extern int  log_to_stderr;

/*
 * Initialize syslog(), if running as daemon.
 */
void
log_open(const char *ident, int option, int facility)
{
    if (log_to_stderr == 0)
        openlog(ident, option, facility);
}

/*
 * Nonfatal error related to a system call.
 * Print a message with the system's errno value and return.
 */
void
log_ret(const char *fmt, ...)
{
    va_list     ap;

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

/*
 * Fatal error related to a system call.
 * Print a message and terminate.
 */
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);
}

/*
 * Nonfatal error unrelated to a system call.
 * Print a message and return.
 */
void
log_msg(const char *fmt, ...)
{
    va_list     ap;

    va_start(ap, fmt);
    log_doit(0, 0, LOG_ERR, fmt, ap);
    va_end(ap);
}

/*
 * Fatal error unrelated to a system call.
 * Print a message and terminate.
 */
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);
}

/*
 * Fatal error related to a system call.
 * Error number passed as an explicit parameter.
 * Print a message and terminate.
 */
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);
}

/*
 * Print a message and return to caller.
 * Caller specifies "errnoflag" and "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);
    }
}
```

Figure B.4 Error functions for daemons

Appendix C

Solutions to Selected Exercises

Chapter 1

1.1 For this exercise, we use the following two arguments for the `ls`(1) command: `-i` prints the i-node number of the file or directory (we say more about i-nodes in Section 4.14), and `-d` prints information about a directory instead of information on all the files in the directory.

Execute the following:

```
$ ls -ldi /etc/. /etc/..          -i says print i-node number
 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 /. /..                  both . and .. have i-node number 2
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /./
      2 drwxr-xr-x  19 root    4096 Jan 15 07:25 /../
```

1.2 The UNIX System is a multiprogramming, or multitasking, system. Other processes were running at the time this program was run.

1.3 Since the *msg* argument to `perror` is a pointer, `perror` could modify the string that *msg* points to. The qualifier `const`, however, says that `perror` does not modify what the pointer points to. On the other hand, the error number

argument to `strerror` is an integer, and since C passes all arguments by value, the `strerror` function couldn't modify this value even if it wanted to. (If the handling of function arguments in C is not clear, you should review Section 5.2 of Kernighan and Ritchie [1988].)

1.4 During the year 2038. We can solve the problem by making the `time_t` data type a 64-bit integer. If it is currently a 32-bit integer, applications will have to be recompiled to work properly. But the problem is worse. Some file systems and backup media store times in 32-bit integers. These would need to be updated as well, but we still need to be able to read the old format.

1.5 Approximately 248 days.

Chapter 2

2.1 The following technique is used by FreeBSD. The primitive data types that can appear in multiple headers are defined in the header `<machine/_types.h>`. For example:

```
#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_ */
```

In each of the headers that can define the `size_t` primitive system data type, we have the sequence

```
#ifndef  _SIZE_T_DECLARED
typedef __size_t        size_t;
#define _SIZE_T_DECLARED
#endif
```

This way, the `typedef` for `size_t` is executed only once.

2.3 If `OPEN_MAX` is indeterminate or ridiculously large (i.e., equal to `LONG_MAX`), we can use `getrlimit` to get the per-process maximum for open file descriptors. Since the per-process limit can be modified, we can't cache the value obtained from the previous call (it might have changed). See Figure C.1.

```
#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("can't get file limit");
        if (rl.rlim_max == RLIM_INFINITY)
            openmax = OPEN_MAX_GUESS;
        else
            openmax = rl.rlim_max;
    }
    return(openmax);
}
```

Figure C.1 Alternative method for identifying the largest possible file descriptor

Chapter 3

3.1 All disk I/O goes through the kernel's block buffers (also called the kernel's buffer cache). The exception to this is I/O on a raw disk device, which we aren't considering. (Some systems also provide a *direct I/O* option to allow applications to bypass the kernel buffers, but we aren't considering this option either.) Chapter 3 of Bach [1986] describes the operation of this buffer cache. Since the data that we `read` or `write` is buffered by the kernel, the term *unbuffered I/O* refers to the lack of automatic buffering in the user process with these two functions. Each `read` or `write` invokes a single system call.

3.3 Each call to `open` gives us a new file table entry. However, since both `open`s reference the same file, both file table entries point to the same v-node table entry. The call to `dup` references the existing file table entry. We show this in Figure C.2. An `F_SETFD` on `fd1` affects only the file descriptor flags for `fd1`, but an `F_SETFL` on `fd1` affects the file table entry that both `fd1` and `fd2` point to.

3.4 If `fd` is 1, then the `dup2(fd, 1)` returns 1 without closing file descriptor 1. (Remember our discussion of this in Section 3.12.) After the three calls to `dup2`, all three descriptors point to the same file table entry. Nothing needs to be closed.

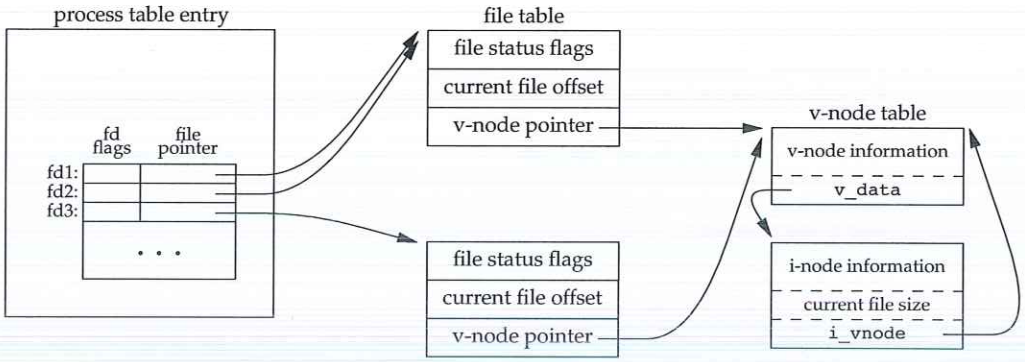


Figure C.2 Result of dup and open

If `fd` is 3, however, after the three calls to `dup2`, four descriptors are pointing to the same file table entry. In this case, we need to close descriptor 3.

3.5 Since the shells process their command line from left to right, the command

```
./a.out > outfile 2>&1
```

first sets standard output to `outfile` and then dups standard output onto descriptor 2 (standard error). The result is that standard output and standard error are set to the same file. Descriptors 1 and 2 both point to the same file table entry. With

```
./a.out 2>&1 > outfile
```

however, the dup is executed first, causing descriptor 2 to be the terminal (assuming that the command is run interactively). Then standard output is redirected to the file `outfile`. The result is that descriptor 1 points to the file table entry for `outfile`, and descriptor 2 points to the file table entry for the terminal.

3.6 You can still `lseek` and `read` anywhere in the file, but a `write` automatically resets the file offset to the end of file before the data is written. This makes it impossible to `write` anywhere other than at the end of file.

Chapter 4

4.1 If `stat` is called, it always tries to follow a symbolic link (Figure 4.17), so the program will never print a file type of "symbolic link." For the example shown in the text, where `/dev/cdrom` is a symbolic link to `/dev/sr0`, `stat` reports that `/dev/cdrom` is a block special file, not a symbolic link. If the symbolic link points to a nonexistent file, `stat` returns an error.

4.2 All permissions are turned off:

```
$ umask 777
$ date > temp.foo
$ ls -l temp.foo
----------  1 sar          29 Feb  5 14:06 temp.foo
```

4.3 The following shows what happens when user-read permission is turned off:

```
$ date > foo
$ chmod u-r foo                    turn off user-read permission
$ ls -l foo                        verify the file's permissions
--w-r--r--  1 sar          29 Feb  5 14:21 foo
$ cat foo                          and try to read it
cat: foo: Permission denied
```

4.4 If we try, using either open or `creat`, to create a file that already exists, the file's access permission bits are not changed. We can verify this by running the program from Figure 4.9:

```
$ rm foo bar                       delete the files in case they already exist
$ date > foo                       create them with some data
$ date > bar
$ chmod a-r foo bar                turn off all read permissions
$ ls -l foo bar                    verify their permissions
--w-------  1 sar          29 Feb  5 14:25 bar
--w-------  1 sar          29 Feb  5 14:25 foo
$ ./a.out                          run program from Figure 4.9
$ ls -l foo bar                    check permissions and sizes
--w-------  1 sar           0 Feb  5 14:26 bar
--w-------  1 sar           0 Feb  5 14:26 foo
```

Note that the permissions didn't change but that the files were truncated.

4.5 The size of a directory should never be 0, since there should always be entries for dot and dot-dot. The size of a symbolic link is the number of characters in the pathname contained in the symbolic link, and this pathname must always contain at least one character.

4.7 The kernel has a default setting for the file access permission bits when it creates a new `core` file. In this example, it was `rw-r--r--`. This default value may or may not be modified by the `umask` value. The shell also has a default setting for the file access permission bits when it creates a new file for redirection. In this example, it was `rw-rw-rw-`, and this value is always modified by our current `umask`. In this example, our `umask` was 02.

4.8 We can't use `du`, because it requires either the name of the file, as in

```
du tempfile
```

or a directory name, as in

```
du .
```

But when the `unlink` function returns, the directory entry for `tempfile` is gone. The `du .` command just shown would not account for the space still taken by

`tempfile`. We have to use the `df` command in this example to see the actual amount of free space on the file system.

4.9 If the link being removed is not the last link to the file, the file is not removed. In this case, the changed-status time of the file is updated. But if the link being removed is the last link to the file, it makes no sense to update this time, because all the information about the file (the i-node) is removed with the file.

4.10 We recursively call our function `dopath` after opening a directory with `opendir`. Assuming that `opendir` uses a single file descriptor, this means that each time we descend one level, we use another descriptor. (We assume that the descriptor isn't closed until we're finished with a directory and call `closedir`.) This limits the depth of the file system tree that we can traverse to the maximum number of open descriptors for the process. Note that the `nftw` function as specified in the XSI option of the Single UNIX Specification allows the caller to specify the number of descriptors to use, implying that it can close and reuse descriptors.

4.12 The `chroot` function is used by the Internet File Transfer Protocol (FTP) program to aid in security. Users without accounts on a system (termed *anonymous FTP*) are placed in a separate directory, and a `chroot` is done to that directory. This prevents the user from accessing any file outside this new root directory.

In addition, `chroot` can be used to build a copy of a file system hierarchy at a new location and then modify this new copy without changing the original file system. This could be used, for example, to test the installation of new software packages.

Only the superuser can execute `chroot`, and once you change the root of a process, it (and all its descendants) can never get back to the original root.

4.13 First, call `stat` to fetch the three times for the file; then call `utimes` to set the desired value. The value that we don't want to change in the call to `utimes` should be the corresponding value from `stat`.

4.14 The `finger`(1) command calls `stat` on the mailbox. The last-modification time is the time that mail was last received, and the last-access time is when the mail was last read.

4.15 Both `cpio` and `tar` store only the modification time (`st_mtime`) in the archive. The access time isn't stored, because its value corresponds to the time the archive was created, since the file has to be read to be archived. The `-a` option to `cpio` has it reset the access time of each input file after the file has been read. This way, the creation of the archive doesn't change the access time. (Resetting the access time, however, does modify the changed-status time.) The changed-status time isn't stored in the archive, because we can't set this value on extraction even if it was archived. (The `utimes` function and its related functions, `futimens` and `utimensat`, can change only the access time and the modification time.)

When the archive is read back (extracted), `tar`, by default, restores the modification time to the value in the archive. The `m` option to `tar` tells it to not

restore the modification time from the archive; instead, the modification time is set to the time of extraction. In all cases with `tar`, the access time after extraction will be the time of extraction.

In contrast, `cpio` sets the access time and the modification time to the time of extraction. By default, it doesn't try to set the modification time to the value on the archive. The `-m` option to `cpio` has it set both the access time and the modification time to the value that was archived.

4.16 The kernel has no inherent limit on the depth of a directory tree. Nevertheless, many commands will fail on pathnames that exceed `PATH_MAX`. The program shown in Figure C.3 creates a directory tree that is 1,000 levels deep, with each level being a 45-character name. We are able to create this structure on all platforms; however, we cannot obtain the absolute pathname of the directory at the 1,000th level using `getcwd` on all platforms. On Mac OS X 10.6.8, we can never get `getcwd` to succeed while in the directory at the end of this long path. The program is able to retrieve the pathname on FreeBSD 8.0, Linux 3.2.0, and Solaris 10, but we have to call `realloc` numerous times to obtain a buffer that is large enough. Running this program on Linux 3.2.0 gives us

```
$ ./a.out
getcwd failed, size = 4096: Numerical result out of range
getcwd failed, size = 4196: Numerical result out of range
...                          418 more lines
getcwd failed, size = 45896: Numerical result out of range
getcwd failed, size = 45996: Numerical result out of range
length = 46004
                             the 46,004-byte pathname is printed here
```

We are not able to archive this directory, however, using `cpio`. It complains that many of the filenames are too long. In fact, `cpio` is unable to archive this directory on all four platforms. In contrast, we can archive this directory using `tar` on FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8. However, we are unable to extract the directory hierarchy from the archive on Linux 3.2.0.

4.17 The `/dev` directory has write permissions turned off to prevent a normal user from removing the filenames in the directory. This means that the `unlink` attempt fails.

Chapter 5

5.2 The `fgets` function reads up through and including the next newline *or* until the buffer is full (leaving room, of course, for the terminating null). Also, `fputs` writes everything in the buffer until it hits a null byte; it doesn't care whether a newline is in the buffer. So, if `MAXLINE` is too small, both functions still work; they're just called more often than they would be if the buffer were larger.

If either of these functions removed or added the newline (as `gets` and `puts` do), we would have to ensure that our buffer was big enough for the largest line.

```
#include "apue.h"
#include <fcntl.h>

#define DEPTH    1000           /* directory depth */
#define STARTDIR "/tmp"
#define NAME     "alonglonglonglonglonglonglonglonglongname"
#define MAXSZ    (10*8192)

int
main(void)
{
    int     i;
    size_t  size;
    char    *path;

    if (chdir(STARTDIR) < 0)
        err_sys("chdir error");

    for (i = 0; i < DEPTH; i++) {
        if (mkdir(NAME, DIR_MODE) < 0)
            err_sys("mkdir failed, i = %d", i);
        if (chdir(NAME) < 0)
            err_sys("chdir failed, i = %d", i);
    }

    if (creat("afile", FILE_MODE) < 0)
        err_sys("creat error");

    /*
     * The deep directory is created, with a file at the leaf.
     * Now let's try to obtain its pathname.
     */
    path = path_alloc(&size);
    for ( ; ; ) {
        if (getcwd(path, size) != NULL) {
            break;
        } else {
            err_ret("getcwd failed, size = %ld", (long)size);
            size += 100;
            if (size > MAXSZ)
                err_quit("giving up");
            if ((path = realloc(path, size)) == NULL)
                err_sys("realloc error");
        }
    }
    printf("length = %ld\n%s\n", (long)strlen(path), path);

    exit(0);
}
```

Figure C.3 Create a deep directory tree

5.3 The function call

```
printf("");
```

returns 0, since no characters are output.

5.4 This is a common error. The return value from `getc` and `getchar` is an `int`, not a `char`. `EOF` is often defined to be –1, so if the system uses signed characters, the code normally works. But if the system uses unsigned characters, after the `EOF` returned by `getchar` is stored as an unsigned character, the character's value no longer equals –1, so the loop never terminates. The four platforms described in this book all use signed characters, so the example code works on these platforms.

5.5 Call `fsync` after each call to `fflush`. The argument to `fsync` is obtained with the `fileno` function. Calling `fsync` without calling `fflush` might do nothing if all the data were still in memory buffers.

5.6 Standard input and standard output are both line buffered when a program is run interactively. When `fgets` is called, standard output is flushed automatically.

5.7 An implementation of `fmemopen` for BSD-based systems is shown in Figure C.4.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

/*
 * Our internal structure tracking a memory stream
 */
struct memstream
{
    char    *buf;   /* in-memory buffer */
    size_t  rsize;  /* real size of buffer */
    size_t  vsize;  /* virtual size of buffer */
    size_t  curpos; /* current position in buffer */
    int     flags;  /* see below */
};

/* flags */
#define MS_READ     0x01    /* open for reading */
#define MS_WRITE    0x02    /* open for writing */
#define MS_APPEND   0x04    /* append to stream */
#define MS_TRUNCATE 0x08    /* truncate the stream on open */
#define MS_MYBUF    0x10    /* free buffer on close */

#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) {            /* "a" mode */
        ms->vsize = ms->curpos;
    } else if (ms->flags & MS_TRUNCATE) {   /* "w" mode */
        ms->vsize = 0;
    } else {                                /* "r" mode */
```

```
        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);  /* error */
            flags |= MS_READ;
            break;

        case 'w':
            if (flags != 0)
                return(0);  /* error */
            flags |= MS_WRITE|MS_TRUNCATE;
            break;

        case 'a':
            if (flags != 0)
                return(0);  /* error */
            flags |= MS_APPEND;
            break;

        case '+':
            if (flags == 0)
                return(0);  /* error */
            flags |= MS_READ|MS_WRITE;
            break;

        case 'b':
            if (flags == 0)
                return(0);  /* error */
            break;

        default:
            return(0);      /* error */
        }
    }
```

```
    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);

    /* can only read from curpos to 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);
}
```

Figure C.4 Implementation of `fmemopen` for BSD systems

Chapter 6

6.1 The functions to access the shadow password file on Linux and Solaris are discussed in Section 6.3. We can't use the value returned in the `pw_passwd` field by the functions described in Section 6.2 to compare an encrypted password, since that field is not the encrypted password. Instead, we need to find the user's entry in the shadow file and use its encrypted password field.

On FreeBSD and Mac OS X, the password file is shadowed automatically. In the `passwd` structure returned by `getpwnam` and `getpwuid` on FreeBSD 8.0, the `pw_passwd` field contains the encrypted password, but only if the caller's effective user ID is 0. On Mac OS X 10.6.8, the encrypted password is not accessible using these interfaces.

6.2 The program in Figure C.5 prints the encrypted password on Linux 3.2.0 and Solaris 10. Unless this program is run with superuser permissions, the call to `getspnam` fails with an error of `EACCES`.

```
#include "apue.h"
#include <shadow.h>

int
main(void)      /* Linux/Solaris version */
{
    struct spwd *ptr;

    if ((ptr = getspnam("sar")) == NULL)
        err_sys("getspnam error");
    printf("sp_pwdp = %s\n", ptr->sp_pwdp == NULL ||
      ptr->sp_pwdp[0] == 0 ?  "(null)" : ptr->sp_pwdp);
    exit(0);
}
```

Figure C.5 Print encrypted password under Linux and Solaris

Under FreeBSD 8.0, the program in Figure C.6 prints the encrypted password if the program is run with superuser permissions. Otherwise, the value returned in `pw_passwd` is an asterisk. On Mac OS X 10.6.8, asterisks are printed regardless of the permissions with which it is run.

```
#include "apue.h"
#include <pwd.h>

int
main(void)      /* FreeBSD/Mac OS X version */
{
    struct passwd   *ptr;

    if ((ptr = getpwnam("sar")) == NULL)
        err_sys("getpwnam error");
```

```
    printf("pw_passwd = %s\n", ptr->pw_passwd == NULL ||
      ptr->pw_passwd[0] == 0 ?  "(null)" : ptr->pw_passwd);
    exit(0);
}
```

Figure C.6 Print encrypted password under FreeBSD and Mac OS X

6.5 The program shown in Figure C.7 prints the date in a format similar to the `date` command.

```
#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("time error");
    if ((tm = localtime(&caltime)) == NULL)
        err_sys("localtime error");
    if (strftime(line, MAXLINE, "%a %b %d %X %Z %Y\n", tm) == 0)
        err_sys("strftime error");
    fputs(line, stdout);
    exit(0);
}
```

Figure C.7 Print the time and date in a format similar to `date`(1)

Running this program gives us

```
$ ./a.out                          author's default is US/Eastern
Wed Jul 25 22:58:32 EDT 2012
$ TZ=US/Mountain ./a.out           U.S. Mountain time zone
Wed Jul 25 20:58:32 MDT 2012
$ TZ=Japan ./a.out                 Japan
Thu Jul 26 11:58:32 JST 2012
```

Chapter 7

7.1 It appears that the return value from `printf` (the number of characters output) becomes the return value of `main`. To verify this theory, change the length of the string printed and see if the new length matches the return value. Note that not all systems exhibit this property. Also note that if you enable the ISO C extensions in `gcc`, then the return value is always 0, as required by the standard.

7.2 When the program is run interactively, standard output is usually line buffered, so the actual output occurs when each newline is output. If standard output were directed to a file, however, it would probably be fully buffered, and the actual output wouldn't occur until the standard I/O cleanup is performed.

7.3 On most UNIX systems, there is no way to do this. Copies of `argc` and `argv` are not kept in global variables like `environ` is.

7.4 This provides a way to terminate the process when it tries to dereference a null pointer, a common C programming error.

7.5 The definitions are

```
typedef void    Exitfunc(void);

int atexit(Exitfunc *func);
```

7.6 `calloc` initializes the memory that it allocates to all zero bits. ISO C does not guarantee that this is the same as either a floating-point 0 or a null pointer.

7.7 The heap and the stack aren't allocated until a program is executed by one of the `exec` functions (described in Section 8.10).

7.8 The executable file (`a.out`) contains symbol table information that can be helpful in debugging a `core` file. To remove this information, use the `strip`(1) command. Stripping the two `a.out` files reduces their size to 798,760 and 6,200 bytes.

7.9 When shared libraries are not used, a large portion of the executable file is occupied by the standard I/O library.

7.10 The code is incorrect, since it references the automatic integer `val` through a pointer after the automatic variable is no longer in existence. Automatic variables declared after the left brace that starts a compound statement disappear after the matching right brace.

Chapter 8

8.1 To simulate the behavior of the child closing the standard output when it exits, add the following line before calling `exit` in the child:

```
fclose(stdout);
```

To see the effects of doing this, replace the call to `printf` with the lines

```
i = printf("pid = %ld, glob = %d, var = %d\n",
    (long)getpid(), glob, var);
sprintf(buf, "%d\n", i);
write(STDOUT_FILENO, buf, strlen(buf));
```

You need to define the variables `i` and `buf` also.

This assumes that the standard I/O stream `stdout` is closed when the child calls `exit`, not the file descriptor `STDOUT_FILENO`. Some versions of the standard I/O library close the file descriptor associated with standard output, which would cause the `write` to standard output to also fail. In this case, dup standard output to another descriptor, and use this new descriptor for the `write`.

8.2 Consider Figure C.8.

```
#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("vfork error");
    /* child and parent both return */
}

static void
f2(void)
{
    char    buf[1000];      /* automatic variables */
    int     i;

    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

Figure C.8 Incorrect use of `vfork`

When `vfork` is called, the parent's stack pointer points to the stack frame for the `f1` function that calls `vfork`. Figure C.9 shows this. `vfork` causes the child to execute first, and the child returns from `f1`. The child then calls `f2`, and its stack frame overwrites the previous stack frame for `f1`. The child then zeros out the automatic variable `buf`, setting 1,000 bytes of the stack frame to 0. The child returns from `f2` and then calls `_exit`, but the contents of the stack beneath the stack frame for `main` have been changed. The parent then resumes after the call

to `vfork` and does a return from `f1`. The return information is often stored in the stack frame, and that information has probably been modified by the child. After the parent resumes, what happens with this example depends on many implementation features of your UNIX system (where in the stack frame the return information is stored, what information in the stack frame is wiped out when the automatic variables are modified, and so on). The normal result is a `core` file, but your results may differ.

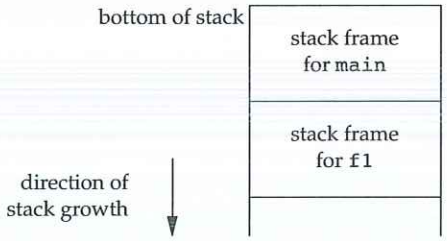


Figure C.9 Stack frames when `vfork` is called

8.4 In Figure 8.13, we have the parent write its output first. When the parent is done, the child writes its output, but we let the parent terminate. Whether the parent terminates or whether the child finishes its output first depends on the kernel's scheduling of the two processes (another race condition). When the parent terminates, the shell starts up the next program, and this next program can interfere with the output from the previous child.

We can prevent this from happening by not letting the parent terminate until the child has also finished its output. Replace the code following the `fork` with the following:

```
else if (pid == 0) {
    WAIT_PARENT();          /* parent goes first */
    charatatime("output from child\n");
    TELL_PARENT(getppid()); /* tell parent we're done */
} else {
    charatatime("output from parent\n");
    TELL_CHILD(pid);        /* tell child we're done */
    WAIT_CHILD();           /* wait for child to finish */
}
```

We won't see this happen if we let the child go first, since the shell doesn't start the next program until the parent terminates.

8.5 The same value (`/home/sar/bin/testinterp`) is printed for `argv[2]`. The reason is that `execlp` ends up calling `execve` with the same *pathname* as when we call `execl` directly. Recall Figure 8.15.

8.6 The program in Figure C.10 creates a zombie.

```
#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("fork error");
    else if (pid == 0)      /* child */
        exit(0);

    /* parent */
    sleep(4);
    system(PSCMD);

    exit(0);
}
```

Figure C.10 Create a zombie and look at its status with ps

Zombies are usually designated by ps(1) with a status of 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
```

Chapter 9

9.1 The `init` process learns when a terminal user logs out, because `init` is the parent of the login shell and receives the `SIGCHLD` signal when the login shell terminates.

For a network login, however, `init` is not involved. Instead, the login entries in the `utmp` and `wtmp` files, and their corresponding logout entries, are usually written by the process that handles the login and detects the logout (`telnetd` in our example).

Chapter 10

10.1 The program terminates the first time we send it a signal. The reason is that the `pause` function returns whenever a signal is caught.

10.3 Figure C.11 shows the stack frames.

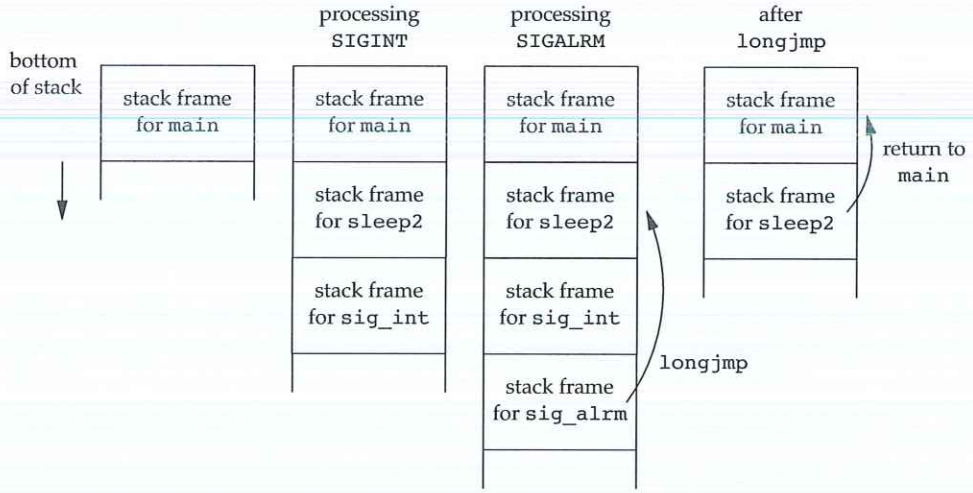


Figure C.11 Stack frames before and after `longjmp`

The `longjmp` from `sig_alrm` back to `sleep2` effectively aborts the call to `sig_int`. From this point, `sleep2` returns to main (recall Figure 10.8).

10.4 We again have a race condition, this time between the first call to `alarm` and the call to `setjmp`. If the process is blocked by the kernel between these two function calls, the alarm goes off, the signal handler is called, and `longjmp` is called. But since `setjmp` was never called, the buffer `env_alrm` is not set. The operation of `longjmp` is undefined if its jump buffer has not been initialized by `setjmp`.

10.5 See "Implementing Software Timers" by Don Libes (*C Users Journal*, vol. 8, no. 11, Nov. 1990) for an example. A copy of this paper is available online at `http://www.kohala.com/start/libes.timers.txt`.

10.7 If we simply called `_exit`, the termination status of the process would not show that it was terminated by the `SIGABRT` signal.

10.8 If the signal was sent by a process owned by some other user, the process has to be set-user-ID to either root or to the owner of the receiving process, or the `kill` attempt won't work. Therefore, the real user ID provides more information to the receiver of the signal.

10.10 On one system used by the author, the value for the number of seconds increased by 1 about every 60–90 minutes. This skew occurs because each call to `sleep` schedules an event for a time in the future, but is not awakened exactly when that event occurs (because of CPU scheduling). In addition, a finite amount of time is required for our process to start running and call `sleep` again.

A program such as the `cron` daemon has to fetch the current time every minute, as well as to set its first sleep period so that it wakes up at the beginning of the next minute. (Convert the current time to the local time and look at the `tm_sec` value.) Every minute, it sets the next sleep period so that it'll wake up at the next minute. Most of the calls will probably be `sleep(60)`, with an occasional `sleep(59)` to resynchronize with the next minute. But if at some point the process takes a long time executing commands or if the system gets heavily loaded and scheduling delays hold up the process, the sleep value can be much less than 60.

10.11 Under Linux 3.2.0, Mac OS X 10.6.8, and Solaris 10, the signal handler for `SIGXFSZ` is never called. But `write` returns a count of 24 as soon as the file's size reaches 1,024 bytes.

When the file's size has reached 1,000 bytes under FreeBSD 8.0, the signal handler is called on the next attempt to write 100 bytes, and the `write` call returns –1 with `errno` set to `EFBIG` ("File too big").

On all four platforms, if we attempt an additional `write` at the current file offset (the end of the file), we will receive `SIGXFSZ` and `write` will fail, returning –1 with `errno` set to `EFBIG`.

10.12 The results depend on the implementation of the standard I/O library: how the `fwrite` function handles an interrupted `write`.

On Linux 3.2.0, for example, when we use the `fwrite` function to write a large buffer, the `fwrite` function calls `write` directly for the same number of bytes. While in the middle of the `write` system call, the alarm fires, but we don't see the signal until the write completes. It appears as if the kernel is blocking the signal while we are in the middle of the `write` system call.

In contrast, on Solaris 10, the `fwrite` function calls `write` in 8 KB increments until the entire amount is written. When the alarm fires, it is caught, interrupting the call to `fwrite`. After we return from the signal handler, we return to the loop inside the `fwrite` function and continue writing in 8 KB increments.

Chapter 11

11.1 A version of the program that allocates the memory dynamically instead of using an automatic variable is shown in Figure C.12.

```
#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("  structure at 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("can't allocate memory");
    fp->a = 1;
    fp->b = 2;
    fp->c = 3;
    fp->d = 4;
    printfoo("thread:\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, "can't create thread 1");
    err = pthread_join(tid1, (void *)&fp);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printfoo("parent:\n", fp);
    exit(0);
}
```

Figure C.12 Correct use of thread return value

11.2 To change the thread ID of a pending job, the reader–writer lock must be held in write mode to prevent anyone from searching the list while the ID is being changed. The problem with the way the interfaces are currently defined is that the ID of a job can change between the time that the job is found with `job_find` and the job is removed from the list by calling `job_remove`. This problem can be solved by embedding a reference count and a mutex inside the `job` structure and having `job_find` increment the reference count. The code that changes the ID can then avoid any job in the list that has a nonzero reference count.

11.3 First of all, the list is protected by a reader–writer lock, but the condition variable needs a mutex to protect the condition. Second, the condition each thread should wait to be satisfied is that there is a job for it to process, so we need to create a per-thread data structure to represent this condition. Alternatively, we can embed the mutex and condition variable in the `queue` structure, but this means that all worker threads will wait on the same condition. If there are many worker threads, we can run into a *thundering herd* problem, whereby many threads are awakened without work to do, resulting in a waste of CPU resources and increased lock contention.

11.4 It depends on the circumstances. In general, both can be correct, but each alternative has drawbacks. In the first sequence, the waiting threads will be scheduled to run after we call `pthread_cond_broadcast`. If the program is running on a multiprocessor, some threads will run and immediately block because we are still holding the mutex (recall that `pthread_cond_wait` returns with the mutex held). In the second sequence, a running thread can acquire the mutex between steps 3 and 4, invalidate the condition, and release the mutex. Then, when we call `pthread_cond_broadcast`, the condition will no longer be true, and the threads will run needlessly. This is why the awakened threads must recheck the condition and not assume that it is true merely because `pthread_cond_wait` returned.

Chapter 12

12.1 This is not a multithreading problem, as one might first guess. The standard I/O routines are indeed thread-safe. When we call `fork`, each process gets a copy of the standard I/O data structures. When we run the program with standard output attached to a terminal, the output is line buffered, so every time we print a line, the standard I/O library writes it to our terminal. However, if we redirect the standard output to a file, then the standard output is fully buffered. The output is written when the buffer fills or the process closes the stream. When we `fork` in this example, the buffer contains several printed lines not yet written, so when the parent and the child finally flush their copies of the buffer, the initial duplicate contents are written to the file.

12.3 Theoretically, if we arrange for all signals to be blocked when the signal handler runs, we should be able to make a function async-signal safe. The problem is

that we don't know whether any of the functions we call might unmask a signal that we've blocked, thereby making it possible for the function to be reentered through another signal handler.

12.4 On FreeBSD 8.0, the program drops core. With `gdb`, we are able to see that the program initialization calls pthread functions, which call `getenv` to find the value of the `LIBPTHREAD_SPINLOOPS` and `LIBPTHREAD_YIELDLOOPS` environment variables. However, our thread-safe version of `getenv` calls back into the pthread library while it is in an intermediate, inconsistent state. In addition, the thread initialization functions call `malloc`, which, in turn, call `getenv` to find the value of the `MALLOC_OPTIONS` environment variable.

To get around this problem, we could make the reasonable assumption that program start-up is single threaded, and use a flag to indicate whether the thread initialization had been completed by our version of `getenv`. While this flag is false, our version of `getenv` can operate as the non-reentrant version does (and avoid all calls to pthread functions and `malloc`). Then we could provide a separate initialization function to call `pthread_once`, instead of calling it from inside `getenv`. This requires that the program call our initialization function before calling `getenv`. This solves our problem, because this can't be done until the program start-up initialization completes. After the program calls our initialization function, our version of `getenv` operates in a thread-safe manner.

12.5 We still need `fork` if we want to run a program from within another program (i.e., before calling `exec`).

12.6 Figure C.13 shows a thread-safe `sleep` implementation that uses `select` to delay for the specified amount of time. It is thread-safe because it doesn't use any unprotected global or static data and calls only other thread-safe functions.

12.7 The implementation of a condition variable most likely uses a mutex to protect its internal structure. Because this is an implementation detail and therefore hidden, there is no portable way for us to acquire and release the lock in the fork handlers. Since we can't determine the state of the internal lock in a condition variable after calling `fork`, it is unsafe for us to use the condition variable in the child process.

Chapter 13

13.1 If it calls `chroot`, the process will not be able to open `/dev/log`. The solution is for the daemon to call `openlog` with an *option* of `LOG_NDELAY`, before calling `chroot`. This opens the special device file (the UNIX domain datagram socket), yielding a descriptor that is still valid, even after a call to `chroot`. This scenario is encountered in daemons, such as `ftpd` (the File Transfer Protocol daemon), that specifically call `chroot` for security reasons but still need to call `syslog` to log error conditions.

```
#include <unistd.h>
#include <time.h>
#include <sys/select.h>

unsigned
sleep(unsigned seconds)
{
    int n;
    unsigned slept;
    time_t start, end;
    struct timeval tv;

    tv.tv_sec = seconds;
    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 >= seconds)
        return(0);
    return(seconds - slept);
}
```

Figure C.13 A thread-safe implementation of `sleep`

13.4 Figure C.14 shows a solution.

```
#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, "no login name\n");
        else
            fprintf(fp, "login name: %s\n", p);
    }
    exit(0);
}
```

Figure C.14 Call `daemonize` and then obtain login name

The results depend on the platform. Recall that `daemonize` closes all open file descriptors and then reopens the first three to `/dev/null`. This means that the process won't have a controlling terminal, so `getlogin` won't be able to look in the `utmp` file for the process's login entry. Thus, on Linux 3.2.0 and Solaris 10, we find that a daemon has no login name.

Under FreeBSD 8.0 and Mac OS X 10.6.8, however, the login name is maintained in the process table and copied across a `fork`. This means that the process can always get the login name, unless the parent didn't have one to start out (such as `init` when the system is bootstrapped).

Chapter 14

14.1 The test program is shown in Figure C.15.

```
#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);

    /*
     * Create a file.
     */
    if ((fd = open("lockfile", O_RDWR|O_CREAT, 0666)) < 0)
        err_sys("can't open/create lockfile");

    /*
     * Read-lock the file.
     */
    if ((pid1 = fork()) < 0) {
        err_sys("fork failed");
    } else if (pid1 == 0) { /* child */
        if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
            err_sys("child 1: can't read-lock file");
        printf("child 1: obtained read lock on file\n");
        pause();
        printf("child 1: exit after pause\n");
```

```
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * Parent continues ... read-lock the file again.
 */
if ((pid2 = fork()) < 0) {
    err_sys("fork failed");
} else if (pid2 == 0) { /* child */
    if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
        err_sys("child 2: can't read-lock file");
    printf("child 2: obtained read lock on file\n");
    pause();
    printf("child 2: exit after pause\n");
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * Parent continues ... block while trying to write-lock
 * the file.
 */
if ((pid3 = fork()) < 0) {
    err_sys("fork failed");
} else if (pid3 == 0) { /* child */
    if (lock_reg(fd, F_SETLK, F_WRLCK, 0, SEEK_SET, 0) < 0)
        printf("child 3: can't set write lock: %s\n",
          strerror(errno));
    printf("child 3 about to block in write-lock...\n");
    if (lock_reg(fd, F_SETLKW, F_WRLCK, 0, SEEK_SET, 0) < 0)
        err_sys("child 3: can't write-lock file");
    printf("child 3 returned and got write lock????\n");
    pause();
    printf("child 3: exit after pause\n");
    exit(0);
} else {        /* parent */
    sleep(2);
}

/*
 * See if a pending write lock will block the next
 * read-lock attempt.
 */
if (lock_reg(fd, F_SETLK, F_RDLCK, 0, SEEK_SET, 0) < 0)
    printf("parent: can't set read lock: %s\n",
      strerror(errno));
else
    printf("parent: obtained additional read lock while"
```

```
            " write lock is pending\n");
    printf("killing child 1...\n");
    kill(pid1, SIGINT);
    printf("killing child 2...\n");
    kill(pid2, SIGINT);
    printf("killing child 3...\n");
    kill(pid3, SIGINT);
    exit(0);
}
```

Figure C.15 Determine record-locking behavior

On FreeBSD 8.0, Linux 3.2.0, and Mac OS X 10.6.8, the behavior is the same: additional readers can starve pending writers. Running the program gives us

```
child 1: obtained read lock on file
child 2: obtained read lock on file
child 3: can't set write lock: Resource temporarily unavailable
child 3 about to block in write-lock...
parent: obtained additional read lock while write lock is pending
killing child 1...
child 1: exit after pause
killing child 2...
child 2: exit after pause
killing child 3...
child 3: can't write-lock file: Interrupted system call
```

On Solaris 10, readers don't starve waiting writers. In this case, the parent is unable to obtain a read lock because there is a process waiting for a write lock.

14.2 Most systems define the `fd_set` data type to be a structure that contains a single member: an array of long integers. One bit in this array corresponds to each descriptor. The four `FD_` macros then manipulate this array of longs, turning specific bits on and off and testing specific bits.

One reason that the data type is defined to be a structure containing an array and not simply an array is to allow variables of type `fd_set` to be assigned to one another with the C assignment statement.

14.3 In the good ol' days, most systems allowed us to define the constant `FD_SETSIZE` before including the header `<sys/select.h>`. For example, we could write

```
#define FD_SETSIZE  2048
#include <sys/select.h>
```

to define the `fd_set` data type to accommodate 2,048 descriptors. Unfortunately, things aren't that simple anymore. To use this technique with contemporary systems, we need to do several things:

1. Before we include any header files, we need to define whatever symbol prevents us from including `<sys/select.h>`. Some systems might protect

the definition of the `fd_set` type with a separate symbol. We need to define this, too.

For example, on FreeBSD 8.0, we need to define `_SYS_SELECT_H_` to prevent the inclusion of `<sys/select.h>` and we need to define `_FD_SET` to prevent the inclusion of the definition for the `fd_set` data type.

2. Sometimes, for compatibility with older applications, `<sys/types.h>` defines the size of the `fd_set`, so we need to include it first, then undefine `FD_SETSIZE`. Note that some systems use `__FD_SETSIZE` instead.
3. We need to redefine `FD_SETSIZE` (or `__FD_SETSIZE`) to the maximum file descriptor number we want to be able to use with `select`.
4. We need to undefine the symbols we defined in step 1.
5. Finally, we can include `<sys/select.h>`.

Before we run the program, we need to configure the system to allow us to open as many file descriptors as we might need so that we can actually make use of `FD_SETSIZE` file descriptors.

14.4 The following table lists the functions that do similar things.

`FD_ZERO`	`sigemptyset`
`FD_SET`	`sigaddset`
`FD_CLR`	`sigdelset`
`FD_ISSET`	`sigismember`

There is not an `FD_xxx` function that corresponds to `sigfillset`. With signal sets, the pointer to the set is always the first argument, and the signal number is the second argument. With descriptor sets, the descriptor number is the first argument, and the pointer to the set is the next argument.

14.5 Figure C.16 shows an implementation using `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);
}
```

Figure C.16 Implementation of `sleep_us` using `select`

Figure C.17 shows an implementation using `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);
}
```

Figure C.17 Implementation of `sleep_us` using `poll`

As the BSD `usleep`(3) manual page states, `usleep` uses the `nanosleep` function, which doesn't interfere with timers set by the calling process.

14.6 No. What we would like to do is have `TELL_WAIT` create a temporary file and use 1 byte for the parent's lock and 1 byte for the child's lock. `WAIT_CHILD` would have the parent wait to obtain a lock on the child's byte, and `TELL_PARENT` would have the child release the lock on the child's byte. The problem, however, is that calling `fork` releases all the locks in the child, so the child can't start off with any locks of its own.

14.7 A solution is shown in Figure C.18.

```
#include "apue.h"
#include <fcntl.h>

int
main(void)
{
    int i, n;
    int fd[2];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    set_fl(fd[1], O_NONBLOCK);

    /* write 1 byte at a time until pipe is full */
    for (n = 0; ; n++) {
        if ((i = write(fd[1], "a", 1)) != 1) {
            printf("write ret %d, ", i);
            break;
        }
    }
    printf("pipe capacity = %d\n", n);
    exit(0);
}
```

Figure C.18 Calculation of pipe capacity using nonblocking writes

The following table shows the values calculated for our four platforms.

Platform	Pipe Capacity (bytes)
FreeBSD 8.0	65,536
Linux 3.2.0	65,536
Mac OS X 10.6.8	16,384
Solaris 10	16,384

These values can differ from the corresponding `PIPE_BUF` values, because `PIPE_BUF` is defined to be the maximum amount of data that can be written to a pipe *atomically*. Here, we calculate the amount of data that a pipe can hold independent of any atomicity constraints.

14.10 Whether the program in Figure 14.27 updates the last-access time for the input file depends on the operating system and the type of file system in which the file resides. On all four platforms, the last-access time is updated when the file resides in the default file system type for the given operating system.

Chapter 15

15.1 If the write end of the pipe is never closed, the reader never sees an end of file. The pager program blocks forever reading from its standard input.

15.2 The parent terminates right after writing the last line to the pipe. The read end of the pipe is automatically closed when the parent terminates. But the parent is probably running ahead of the child by one pipe buffer, since the child (the pager program) is waiting for us to look at a page of output. If we're running a shell, such as the Korn shell, with interactive command-line editing enabled, the shell probably changes the terminal mode when our parent terminates and the shell prints a prompt. This undoubtedly interferes with the pager program, which has also modified the terminal mode. (Most pager programs set the terminal to noncanonical mode when awaiting input to proceed to the next page.)

15.3 The `popen` function returns a file pointer because the shell is executed. But the shell can't execute the nonexistent command, so it prints

```
sh: line 1: ./a.out: No such file or directory
```

on the standard error and terminates with an exit status of 127 (although the value depends on the type of shell). `pclose` returns the termination status of the command as it is returned by `waitpid`.

15.4 When the parent terminates, look at its termination status with the shell. For the Bourne shell, Bourne-again shell, and Korn shell, the command is `echo $?`. The number printed is 128 plus the signal number.

15.5 First add the declaration

```
FILE    *fpin, *fpout;
```

Then use `fdopen` to associate the pipe descriptors with a standard I/O stream, and set the streams to be line buffered. Do this before the `while` loop that reads from standard input:

```
if ((fpin = fdopen(fd2[0], "r")) == NULL)
    err_sys("fdopen error");
if ((fpout = fdopen(fd1[1], "w")) == NULL)
    err_sys("fdopen error");
if (setvbuf(fpin, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
if (setvbuf(fpout, NULL, _IOLBF, 0) < 0)
    err_sys("setvbuf error");
```

The `write` and `read` in the `while` loop are replaced with

```
if (fputs(line, fpout) == EOF)
    err_sys("fputs error to pipe");
if (fgets(line, MAXLINE, fpin) == NULL) {
    err_msg("child closed pipe");
    break;
}
```

15.6 The `system` function calls `wait`, and the first child to terminate is the child generated by popen. Since that's not the child that `system` created, it calls `wait` again and blocks until the `sleep` is done. Then `system` returns. When `pclose` calls `wait`, an error is returned, since there are no more children to `wait` for. Then `pclose` returns an error.

15.7 Although the details vary by platform (see Figure C.19), `select` indicates that the descriptor is readable. After all the data has been read, `read` returns 0 to indicate the end of file. But with `poll`, the `POLLHUP` event is returned, and this can happen while there is still data to be read. Once we have read all the data, however, `read` returns 0 to indicate the end of file. After all the data has been read, the `POLLIN` event is not returned, even though we need to issue a `read` to receive the end-of-file notification (the return value of 0).

Operation	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
`select` on `read` end of pipe with `write` end closed	R/W/E	R	R/W	R/W/E
`poll` on `read` end of pipe with `write` end closed	R/HUP	HUP	INV	HUP
`select` on `write` end of pipe with `read` end closed	R/W/E	R/W	R/W	R/W
`poll` on `write` end of pipe with `read` end closed	R/HUP	W/ERR	INV	HUP

Figure C.19 Pipe behavior with `select` and `poll`

The conditions shown in Figure C.19 include R (readable), W (writable), E (exception), HUP (hangup), ERR (error), and INV (invalid file descriptor). With an output descriptor that refers to a pipe that has been closed by the reader, `select` indicates that the descriptor is writable. But when we call `write`, the `SIGPIPE` signal is generated. If we either ignore this signal or return from its

signal handler, `write` fails with `errno` set to `EPIPE`. With `poll`, however, the behavior varies by platform.

15.8 Anything written by the child to standard error appears wherever the parent's standard error would appear. To send standard error back to the parent, include the shell redirection `2>&1` in the *cmdstring*.

15.9 The `popen` function `fork`s a child, and the child executes the shell. The shell in turn calls `fork`, and the child of the shell executes the command string. When *cmdstring* terminates, the shell is waiting for this to happen. The shell then exits, which is what the `waitpid` in `pclose` is waiting for.

15.10 The trick is to open the FIFO twice: once for reading and once for writing. We never use the descriptor that is opened for writing, but leaving that descriptor open prevents an end of file from being generated when the number of clients goes from 1 to 0. Opening the FIFO twice requires some care, as a nonblocking open is required. We have to do a nonblocking, read-only open first, followed by a blocking open for write-only. (If we tried a nonblocking open for write-only first, it would return an error.) We then turn off nonblocking for the read descriptor. Figure C.20 shows the code for this.

```
#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("mkfifo error");
    if ((fdread = open(FIFO, O_RDONLY | O_NONBLOCK)) < 0)
        err_sys("open error for reading");
    if ((fdwrite = open(FIFO, O_WRONLY)) < 0)
        err_sys("open error for writing");
    clr_fl(fdread, O_NONBLOCK);
    exit(0);
}
```

Figure C.20 Opening a FIFO for reading and writing, without blocking

15.11 Randomly reading a message from an active queue would interfere with the client–server protocol, as either a client request or a server's response would be lost. To read the queue, all that is needed is for the process to know the identifier for the queue and for the queue to allow world-read access.

15.13 We never store actual addresses in a shared memory segment, since it's possible for the server and all the clients to attach the segment at different addresses.

Instead, when a linked list is built in a shared memory segment, the list pointers should be stored as offsets to other objects in the shared memory segment. These offsets are formed by subtracting the start of the shared memory segment from the actual address of the object.

15.14 Figure C.21 shows the relevant events.

Parent i set to	Child i set to	Shared value set to	update returns	Comment
		0		initialized by mmap
	1			child runs first, then is blocked
0				parent runs
		1		
			0	then parent is blocked
		2		child resumes
			1	
	3			then child is blocked
2				parent resumes
		3		
			2	then parent is blocked
		4		
			3	
	5			then child is blocked
4				parent resumes

Figure C.21 Alternation between parent and child in Figure 15.33

Chapter 16

16.1 Figure C.22 shows a program that prints the system's byte order.

```
#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("little-endian\n");
    else if (*cp == 4)
        printf("big-endian\n");
    else
        printf("who knows?\n");
    exit(0);
}
```

Figure C.22 Determine byte order on system

16.3 For each endpoint we will be listening on, we need to bind the proper address and record an entry in an `fd_set` structure corresponding to each file descriptor. We will use `select` to wait for connect requests to arrive on multiple endpoints. Recall from Section 16.4 that a passive endpoint will appear to be readable when a connect request arrives on it. When a connect request does arrive, we will accept the request and process it as before.

16.5 In the `main` procedure, we need to arrange to catch `SIGCHLD` by calling our `signal` function (Figure 10.18), which will use `sigaction` to install the handler specifying the restartable system call option. Next, we need to remove the call to `waitpid` from our `serve` function. After `forking` the child to service the request, the parent closes the new file descriptor and resumes listening for additional connect requests. Finally, we need a signal handler for `SIGCHLD`, as follows:

```
void
sigchld(int signo)
{
    while (waitpid((pid_t)-1, NULL, WNOHANG) > 0)
        ;
}
```

16.6 To enable asynchronous socket I/O, we need to establish socket ownership using the `F_SETOWN fcntl` command, and then enable asynchronous signaling using the `FIOASYNC ioctl` command. To disable asynchronous socket I/O, we simply need to disable asynchronous signaling. The reason we mix `fcntl` and `ioctl` commands is to find the methods that are most portable. The code is shown in Figure C.23.

```
#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);
}
```

Figure C.23 Enable and disable asynchronous socket I/O

Chapter 17

17.1 Regular pipes provide a byte stream interface. To detect message boundaries, we'd have to add a header to each message to indicate the length. But this still involves two extra copy operations: one to write to the pipe and one to read from the pipe. A more efficient approach is to use the pipe only to signal the main thread that a new message is available. We can use a single byte for this purpose. With this approach, we need to move the `mymesg` structure to the `threadinfo` structure and use a mutex and a condition variable to prevent the helper thread from reusing the `mymesg` structure until the main thread is done with it. The solution is shown in Figure C.24.

```
#include "apue.h"
#include <poll.h>
#include <pthread.h>
#include <sys/msg.h>
#include <sys/socket.h>

#define NQ      3       /* number of queues */
#define MAXMSZ  512     /* maximum message size */
#define KEY     0x123   /* key for first message queue */

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("msgrcv error");
        tip->len = n;
        pthread_mutex_lock(&tip->mutex);
        if (write(tip->fd, "a", sizeof(char)) < 0)
            err_sys("write error");
        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("msgget error");

        printf("queue ID %d is %d\n", i, qid[i]);

        if (socketpair(AF_UNIX, SOCK_DGRAM, 0, fd) < 0)
            err_sys("socketpair error");
        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("pthread_cond_init error");
        if (pthread_mutex_init(&ti[i].mutex, NULL) != 0)
            err_sys("pthread_mutex_init error");
        if ((err = pthread_create(&tid[i], NULL, helper,
          &ti[i])) != 0)
            err_exit(err, "pthread_create error");
    }
```

```
    for (;;) {
        if (poll(pfd, NQ, -1) < 0)
            err_sys("poll error");
        for (i = 0; i < NQ; i++) {
            if (pfd[i].revents & POLLIN) {
                if ((n = read(pfd[i].fd, &c, sizeof(char))) < 0)
                    err_sys("read error");
                ti[i].m.mtext[ti[i].len] = 0;
                printf("queue id %d, message %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);
}
```

Figure C.24 Poll for XSI messages using pipes

17.3 A *declaration* specifies the attributes (such as the data type) of a set of identifiers. If the declaration also causes storage to be allocated, it is called a *definition*.

In the `opend.h` header, we declare the three global variables with the `extern` storage class. These declarations do not cause storage to be allocated for the variables. In the `main.c` file, we define the three global variables. Sometimes, we'll also initialize a global variable when we define it, but we typically let the C default apply.

17.5 Both `select` and `poll` return the number of ready descriptors as the value of the function. The loop that goes through the `client` array can terminate when the number of ready descriptors has been processed.

17.6 The first problem with the proposed solution is that there is a race between the call to `stat` and the call to `unlink` where the file can change. The second problem is that if the name is a symbolic link pointing to the UNIX domain socket file, then `stat` will report that the name is a socket (recall that the `stat` function follows symbolic links), but when we call `unlink`, we will actually remove the symbolic link instead of the socket file. To solve this problem, we should use `lstat` instead of `stat`, but this doesn't solve the first problem.

17.7 The first option is to send both file descriptors in one control message. Each file descriptor is stored in adjacent memory locations. The following code shows this:

```
struct msghdr msg;
struct cmsghdr *cmptr;
int *ip;
```

```
if ((cmptr = calloc(1, CMSG_LEN(2*sizeof(int)))) == NULL)
     err_sys("calloc error");
msg.msg_control = cmptr;
msg.msg_controllen = CMSG_LEN(2*sizeof(int));
/* continue initializing 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;
```

This approach works on all four platforms covered in this book. The second option is to pack two separate `cmsghdr` structures into a single message:

```
struct msghdr msg;
struct cmsghdr *cmptr;

if ((cmptr = calloc(1, 2*CMSG_LEN(sizeof(int)))) == NULL)
     err_sys("calloc error");
msg.msg_control = cmptr;
msg.msg_controllen = 2*CMSG_LEN(sizeof(int));
/* continue initializing 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;
```

Unlike the first approach, this method works only on FreeBSD 8.0.

Chapter 18

18.1 Note that you have to terminate the `reset` command with a line feed character, not a return, since the terminal is in noncanonical mode.

18.2 It builds a table for each of the 128 characters and sets the high-order bit (the parity bit) according to the user's specification. It then uses 8-bit I/O, handling the parity generation itself.

18.3 If you happen to be on a windowing terminal, you don't need to log in twice. You can do this experiment between two separate windows. Under Solaris, execute `stty -a` with standard input redirected from the terminal window running `vi`. This shows that `vi` sets MIN to 1 and TIME to 1. A call to `read` will wait for at least one character to be typed, but after that character is entered, `read` waits only one-tenth of a second for additional characters before returning.

Chapter 19

19.1 Both servers, `telnetd` and `rlogind`, run with superuser privileges, so their calls to `chown` and `chmod` succeed.

19.2 Execute `pty -n stty -a` to prevent the slave's `termios` structure and `winsize` structure from being initialized.

19.4 Unfortunately, the `F_SETFL` command of `fcntl` doesn't allow the read–write status to be changed.

19.5 There are three process groups: (1) the login shell, (2) the `pty` parent and child, and (3) the `cat` process. The first two process groups constitute a session with the login shell as the session leader. The second session contains only the `cat` process. The first process group (the login shell) is a background process group, and the other two are foreground process groups.

19.6 First, `cat` terminates when it receives the end of file from its line discipline. This causes the PTY slave to terminate, which causes the PTY master to terminate. This in turn generates an end of file for the `pty` parent that's reading from the PTY master. The parent sends `SIGTERM` to the child, so the child terminates next. (The child doesn't catch this signal.) Finally, the parent calls `exit(0)` at the end of the `main` function.

The relevant output from the program shown in Figure 8.29 is

```
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 This can be done with the shell's `echo` command and the `date`(1) command, all in a subshell:

```
#!/bin/sh
( echo "Script started on " `date`;
  pty "${SHELL:-/bin/sh}";
  echo "Script done on " `date` ) | tee typescript
```

19.8 The line discipline above the PTY slave has echo enabled, so whatever `pty` reads on its standard input and writes to the PTY master gets echoed by default. This echoing is done by the line discipline module above the slave even though the program (`ttyname`) never reads the data.

Chapter 20

20.1 Our conservative locking in `_db_dodelete` is meant to avoid race conditions with `db_nextrec`. If the call to `_db_writedat` were not protected with a write lock, it would be possible to erase the data record while `db_nextrec` was reading that data record: `db_nextrec` would read an index record, determine

that it was not blank, and then read the data record, which could be erased by `_db_dodelete` between the calls to `_db_readidx` and `_db_readdat` in `db_nextrec`.

20.2 Assume that `db_nextrec` calls `_db_readidx`, which reads the key into the index buffer for the process. This process is then stopped by the kernel, and another process runs. This other process calls `db_delete`, and the record being read by the other process is deleted. Both its key and its data are rewritten in the two files as all blanks. The first process resumes and calls `_db_readdat` (from `db_nextrec`) and reads the all-blank data record. The read lock by `db_nextrec` allows it to do the read of the index record, followed by the read of the data record, as an atomic operation (with regard to other cooperating processes using the same database).

20.3 With mandatory locking, other readers and writers are affected. Other reads and writes are blocked by the kernel until the locks placed by `_db_writeidx` and `_db_writedat` are removed.

20.5 By writing the data record before the index record, we protect ourselves from generating a corrupt record if the process should be killed in between the two writes. If the process were to write the index record first, but be killed before writing the data record, then we'd have a valid index record that pointed to invalid data.

Chapter 21

21.5 Here are some hints. There are two places to check for queued jobs: the printer spooling daemon's queue and the network printer's internal queue. Take care to prevent one user from being able to cancel someone else's print job. Of course, the superuser should be able to cancel any job.

21.7 We don't need to prod the daemon, because we don't need to reread the configuration file until we need to print a file. The `printer_thread` function checks whether it needs to reread the configuration file before each attempt to send a job to the printer.

21.9 We need to null-terminate the string we write to the job file (recall that `strlen` doesn't include the terminating null byte when it calculates the length of a string). There are two simple approaches: either we can add 1 to the number of bytes we write, or we can use the `dprintf` function instead of calling `sprintf` and `write`.

This page intentionally left blank

Bibliography

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.

A paper introducing the Mach operating system.

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.

A paper describing the Service Management Facility (SMF) in Solaris, which provides a framework for starting and monitoring administrative processes, and recovering from failures affecting the services they provide.

Adobe Systems Inc. 1999. *PostScript Language Reference Manual, Third Edition*. Addison-Wesley, Reading, MA.

The language reference manual for PostScript.

Aho, A. V., Kernighan, B. W., and Weinberger, P. J. 1988. *The AWK Programming Language*. Addison-Wesley, Reading, MA.

A complete book on the awk programming language. The version of awk described in this book is sometimes called "nawk" (for 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*, pp. 13–22, Pittsburgh, PA.

A description of the AT&T Tuxedo Transaction Processing System.

Arnold, J. Q. 1986. "Shared Libraries on UNIX System V," *Proceedings of the 1986 Summer USENIX Conference*, pp. 395–404, Atlanta, GA.

Describes the implementation of shared libraries in SVR3.

AT&T. 1989. *System V Interface Definition, Third Edition.* Addison-Wesley, Reading, MA.

This four-volume set specifies the source code interface and runtime behavior of System V. The third edition corresponds to SVR4. A fifth volume, containing updated versions of commands and functions from volumes 1–4, was published in 1991. Currently out of print.

AT&T. 1990a. *UNIX Research System Programmer's Manual, Tenth Edition, Volume I.* Saunders College Publishing, Fort Worth, TX.

The version of the *UNIX Programmer's Manual* for the 10th Edition of the Research UNIX System (V10). This volume contains the traditional UNIX System manual pages (Sections 1–9).

AT&T. 1990b. *UNIX Research System Papers, Tenth Edition, Volume II.* Saunders College Publishing, Fort Worth, TX.

Volume II for the 10th Edition of the Research UNIX System (V10) contains 40 papers describing various aspects of the system.

AT&T. 1990c. *UNIX System V Release 4 BSD/XENIX Compatibility Guide.* Prentice Hall, Englewood Cliffs, NJ.

Contains manual pages describing the compatibility library.

AT&T. 1990d. *UNIX System V Release 4 Programmer's Guide: STREAMS.* Prentice Hall, Englewood Cliffs, NJ.

Describes the STREAMS system in SVR4.

AT&T. 1990e. *UNIX System V Release 4 Programmer's Reference Manual.* Prentice Hall, Englewood Cliffs, NJ.

This is the programmer's reference manual for the SVR4 implementation for the Intel 80386 processor. It contains Sections 1 (commands), 2 (system calls), 3 (subroutines), 4 (file formats), and 5 (miscellaneous facilities).

AT&T. 1991. *UNIX System V Release 4 System Administrator's Reference Manual.* Prentice Hall, Englewood Cliffs, NJ.

This is the system administrator's reference manual for the SVR4 implementation for the Intel 80386 processor. It contains Sections 1 (commands), 4 (file formats), 5 (miscellaneous facilities), and 7 (special files).

Bach, M. J. 1986. *The Design of the UNIX Operating System.* Prentice Hall, Englewood Cliffs, NJ.

A book on the details of the design and implementation of the UNIX operating system. Although actual UNIX System source code is not provided in this text (since it was proprietary to AT&T at the time), many of the algorithms and data structures used by the UNIX kernel are presented and discussed. This book describes SVR2.

Bolsky, M. I., and Korn, D. G. 1995. *The New KornShell Command and Programming Language, Second Edition.* Prentice Hall, Englewood Cliffs, NJ.

A book describing how to use the Korn shell, both as a command interpreter and as a programming language.

Bovet, D. P., and Cesati, M. 2006. *Understanding the Linux Kernel, Third Edition.* O'Reilly Media, Sebastopol, CA.

A book that describes the Linux 2.6 kernel architecture.

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, DC.

Describes changes made to the virtual memory implementation of SVR4 to improve its performance, especially for `fork` and `exec`.

Comer, D. E. 1979. "The Ubiquitous B-Tree," *ACM Computing Surveys*, vol. 11, no. 2, pp. 121–137 (June).

A good, comprehensive paper on B-trees.

Date, C. J. 2004. *An Introduction to Database Systems, Eighth Edition*. Addison-Wesley, Boston, MA.

A comprehensive overview of database systems.

Evans, J. 2006. "A Scalable Concurrent `malloc` Implementation for FreeBSD," *Proceedings of BSDCan*.

A paper describing the `jemalloc` implementation of the dynamic memory allocation library used in 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).

A paper describing the extendible hashing technique.

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.

Describes an alternative library function to traverse a file system hierarchy.

Gallmeister, B. O. 1995. *POSIX.4: Programming for the Real World*. O'Reilly & Associates, Sebastopol, CA.

Describes the real-time interfaces in the POSIX standard.

Garfinkel, S., Spafford, G., and Schwartz, A. 2003. *Practical UNIX & Internet Security, Third Edition*. O'Reilly & Associates, Sebastopol, CA.

A detailed book on UNIX System security.

Ghemawat, S., and Menage, P. 2005. "TCMalloc: Thread-Caching Malloc."

A brief description of Google's `TCMalloc` memory allocator. The description is available at `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.

Describes the implementation of shared libraries in 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.

Describes the initial implementation of the `mmap` function and related issues in the virtual memory design.

Goodheart, B. 1991. *UNIX Curses Explained*. Prentice Hall, Englewood Cliffs, NJ.

A complete reference on terminfo and the `curses` library. Currently out of print.

Hume, A. G. 1988. "A Tale of Two Greps," *Software Practice and Experience*, vol. 18, no. 11, pp. 1063–1072.

An interesting paper that discusses performance improvements in `grep`.

IEEE. 1990. *Information Technology—Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language].* IEEE (Dec.).

This was the first of the POSIX standards, and it defined the C language systems interface standard, based on the UNIX operating system. It is often called POSIX.1. It is now part of the Single UNIX Specification published by The Open Group [2008].

ISO. 1999. *International Standard ISO/IEC 9899—Programming Language C.* ISO/IEC.

The official standard for the C language and the standard libraries. Although this was replaced by a new version of the standard in 2011, the systems described in this book still conform to the 1999 version of the standard.

PDF versions of this standard can be purchased online at either `http://www.ansi.org` or `http://www.iso.org`.

ISO. 2011. *International Standard ISO/IEC 9899, Information Technology—Programming Languages—C.* ISO/IEC.

The latest version of the official standard for the C language and the standard libraries, which replaces the 1999 version.

PDF versions of this standard can be purchased online at either `http://www.ansi.org` or `http://www.iso.org`.

Kernighan, B. W., and Pike, R. 1984. *The UNIX Programming Environment.* Prentice Hall, Englewood Cliffs, NJ.

A general reference for additional details on UNIX programming. This book covers numerous UNIX commands and utilities, such as `grep`, `sed`, `awk`, and the Bourne shell.

Kernighan, B. W., and Ritchie, D. M. 1988. *The C Programming Language, Second Edition.* Prentice Hall, Englewood Cliffs, NJ.

A book on the ANSI standard version of the C programming language. Appendix B contains a description of the libraries defined by the ANSI standard.

Kerrisk, M. 2010. *The Linux Programming Interface.* No Starch Press, San Francisco, CA.

If you thought this book was long, here is one that is half again as big, but focuses only on the Linux programming interface.

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.

A description of the original v-node implementation.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition.* Addison-Wesley, Boston, MA.

Describes sorting and searching algorithms.

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.

A description of an alternative to the standard I/O library. The library is available at `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.

An alternative to the standard I/O library based on mapped files.

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.

An entire book on the 4.3BSD operating system. This book describes the Tahoe release of 4.3BSD. Currently out of print.

Lennert, D. 1987. "How to Write a UNIX Daemon," *;login:*, vol. 12, no. 4, pp. 17–23 (July/August).

Describes how to write a daemon in the UNIX System.

Libes, D. 1990. "expect: Curing Those Uncontrollable Fits of Interaction," *Proceedings of the 1990 Summer USENIX Conference*, pp. 183–192, Anaheim, CA.

A description of the `expect` program and its implementation.

Libes, D. 1991. "expect: Scripts for Controlling Interactive Processes," *Computing Systems*, vol. 4, no. 2, pp. 99–125 (Spring).

This paper presents numerous `expect` scripts.

Libes, D. 1994. *Exploring Expect*. O'Reilly & Associates, Sebastopol, CA.

A complete book on using the `expect` program.

Lions, J. 1977. *A Commentary on the UNIX Operating System*. AT&T Bell Laboratories, Murray Hill, NJ.

Describes the source code of the 6th Edition UNIX System. Available only to AT&T employees, contractors, and interns, although copies leaked outside of AT&T.

Lions, J. 1996. *Lions' Commentary on UNIX 6th Edition*. Peer-to-Peer Communications, San Jose, CA.

Describes the 6th Edition UNIX System in a publicly available version of the 1977 classic.

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.

A paper describing the linear hashing technique.

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.

An entire book on the 4.4BSD operating system.

McKusick, M. K., and Neville-Neil, G. V. 2005. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, Boston, MA.

An entire book on the FreeBSD operating system, version 5.2.

McDougall, R., and Mauro, J. 2007. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, Upper Saddle River, NJ.

A book on the internals of the Solaris 10 operating system. Also covers OpenSolaris.

Morris, R., and Thompson, K. 1979. "UNIX Password Security," *Communications of the ACM*, vol. 22, no. 11, pp. 594–597 (Nov.).

A description of the history of the design of the password scheme used in UNIX Systems.

Nemeth, E., Snyder, G., Seebass, S., and Hein, T. R. 2001. *UNIX System Administration Handbook, Third Edition*. Prentice Hall, Upper Saddle River, NJ.

A book with many details on administering a UNIX system.

The Open Group. 2008. *The Single UNIX Specification, Version 4*. The Open Group, Berkshire, UK.

The POSIX and X/Open standards combined into a single reference.

The HTML version can be viewed for free online at `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.

A description of the Plan 9 operating system, developed in the same department where the UNIX System was invented.

Plauger, P. J. 1992. *The Standard C Library*. Prentice Hall, Englewood Cliffs, NJ.

A complete book on the ANSI C library. It contains a complete C implementation of the library.

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).

Describes the IPC facilities provided by the Ninth Edition Research UNIX System, developed at AT&T Bell Laboratories. The features are built on the stream input–output system and include full-duplex pipes, the ability to pass file descriptors between processes, and unique client connections to servers. A copy of this paper also appears in AT&T [1990b].

Rago, S. A. 1993. *UNIX System V Network Programming*. Addison-Wesley, Reading, MA.

A book that describes the networking programming environment of UNIX System V Release 4, which is based on STREAMS.

Raymond, E. S., ed. 1996. *The New Hacker's Dictionary, Third Edition*. MIT Press, Cambridge, MA.

Lots of computer hacker terms defined.

Salus, P. H. 1994. *A Quarter Century of UNIX*. Addison-Wesley, Reading, MA.

A history of the UNIX System from 1969 to 1994.

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.

A modification of the `db`(3) library from 4.4BSD that implements transactions.

Seltzer, M., and Yigit, O. 1991. "A New Hashing Package for UNIX," *Proceedings of the 1991 Winter USENIX Conference*, pp. 173–184, Dallas, TX.

A description of the `dbm`(3) library and its implementations, and a newer hashing package.

Singh, A. 2006. *Mac OS X Internals: A Systems Approach*. Addison-Wesley, Upper Saddle River, NJ.

Roughly 1,600 pages on the design of the Mac OS X operating system.

Stevens, W. R. 1990. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, NJ.

A detailed book on network programming under the UNIX System. The contents of the first edition of this book differ greatly from later editions.

Stevens, W. R., Fenner, B., and Rudoff, A. M. 2004. *UNIX Network Programming, Volume 1, Third Edition*. Addison-Wesley, Boston, MA.

A detailed book on network programming under UNIX System. Redesigned and split into two volumes in the second edition and updated in the third edition.

Stonebraker, M. R. 1981. "Operating System Support for Database Management," *Communications of the ACM*, vol. 24, no. 7, pp. 412–418 (July).

Describes operating system services and how they affect database operation.

Strang, J. 1986. *Programming with curses*. O'Reilly & Associates, Sebastopol, CA.

A book on the Berkeley version of `curses`.

Strang, J., Mui, L., and O'Reilly, T. 1988. *termcap & terminfo, Third Edition*. O'Reilly & Associates, Sebastopol, CA.

A book on `termcap` and `terminfo`.

Sun Microsystems. 2005. *STREAMS Programming Guide*. Sun Microsystems, Santa Clara, CA.

Describes STREAMS programming on the Solaris platform.

Thompson, K. 1978. "UNIX Implementation," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1931–1946 (July–Aug.).

Describes some of the implementation details of Version 7.

Vo, Kiem-Phong. 1996. "Vmalloc: A General and Efficient Memory Allocator," *Software Practice and Experience*, vol. 26, no. 3, pp. 357–374.

Describes a flexible memory allocator.

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.

Describes TOCTTOU weaknesses in the UNIX file system interface.

Weinberger, P. J. 1982. "Making UNIX Operating Systems Safe for Databases," *The Bell System Technical Journal*, vol. 61, no. 9, pp. 2407–2422 (Nov.).

Describes some problems in implementing databases in early UNIX systems.

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.

Describes a memory allocation algorithm suitable for a wide variety of applications.

Williams, T. 1989. "Session Management in System V Release 4," *Proceedings of the 1989 Winter USENIX Conference*, pp. 365–375, San Diego, CA.

Describes the session architecture in SVR4, on which the POSIX.1 interfaces were based. This includes process groups, job control, and controlling terminals. Also describes the security concerns of existing approaches.

X/Open. 1989. *X/Open Portability Guide*. Prentice Hall, Englewood Cliffs, NJ.

A set of seven volumes covering commands and utilities (Vol. 1), system interfaces and headers (Vol. 2), supplementary definitions (Vol. 3), programming languages (Vol. 4), data management (Vol. 5), window management (Vol. 6), networking services (Vol. 7). Although out of print, this has been replaced by the Single UNIX Specification [Open Group 2008].

This page intentionally left blank

Index

The function subentries labeled "definition of" point to where the function prototype appears and, when applicable, to the source code for the function. Functions defined in the text that are used in later examples, such as the `set_fl` function in Figure 3.12, are included in this index. The definitions of functions that are part of the larger examples (Chapters 17, 19, 20, and 21) are also included to help in going through these examples. Also, significant functions and constants that occur in any of the examples in the text, such as `select` and `poll`, are also included in this index. Trivial functions that occur frequently, such as `printf`, are sometimes not referenced when they occur in examples.

This page intentionally left blank

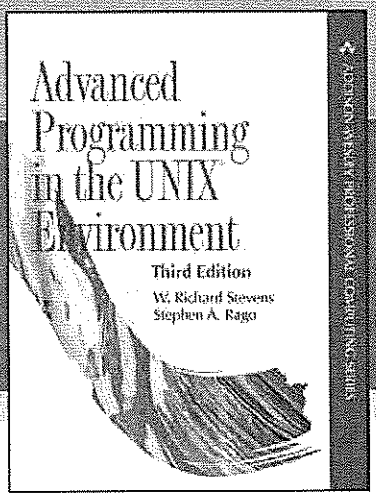
Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.



Safari Books Online

FREE Online Edition

Your purchase of ***Advanced Programming in the UNIX® Environment, Third Edition,*** includes access to a free online edition for 45 days through the **Safari Books Online** subscription service. Nearly every Addison-Wesley Professional book is available online through **Safari Books Online**, along with thousands of books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly Media, Prentice Hall, Que, Sams, and VMware Press.

Safari Books Online is a digital library providing searchable, on-demand access to thousands of technology, digital media, and professional development books and videos from leading publishers. With one monthly or yearly subscription price, you get unlimited access to learning tools and information on topics including mobile app and software development, tips and tricks on using your favorite gadgets, networking, project management, graphic design, and much more.

Activate your FREE Online Edition at informit.com/safarifree

STEP 1: Enter the coupon code: LDMRWBI.

STEP 2: New Safari users, complete the brief registration form. Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition, please e-mail customer-service@safaribooksonline.com