

# Appendix A

## Function Prototypes

530128  
END

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>  
This function never returns
```

p. 365

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

p. 608

int	<b>access</b> (const char *path, int mode); <i>&lt;unistd.h&gt;</i> <i>mode: R_OK, W_OK, X_OK, F_OK</i> <i>Returns: 0 if OK, -1 on error</i>	p. 102
int	<b>aio_cancel</b> (int fd, struct aiocb *aiocb); <i>&lt;aio.h&gt;</i> <i>Returns: AIO_ALLDONE, AIO_CANCELED,  AIO_NOTCANCELED, or -1 on error</i>	p. 514
int	<b>aio_error</b> (const struct aiocb *aiocb); <i>&lt;aio.h&gt;</i> <i>Returns: 0 if operation succeeded, EINPROGRESS if  operation is still in progress, error code if  operation failed, or -1 on error</i>	p. 513
int	<b>aio_fsync</b> (int op, struct aiocb *aiocb); <i>&lt;aio.h&gt;</i> <i>Returns: 0 if OK, -1 on error</i>	p. 513
int	<b>aio_read</b> (struct aiocb *aiocb); <i>&lt;aio.h&gt;</i> <i>Returns: 0 if OK, -1 on error</i>	p. 512
ssize_t	<b>aio_return</b> (const struct aiocb *aiocb); <i>&lt;aio.h&gt;</i> <i>Returns: result of asynchronous operation, -1 on error</i>	p. 513
int	<b>aio_suspend</b> (const struct aiocb *const list[], int nent, const struct timespec *timeout); <i>&lt;aio.h&gt;</i> <i>Returns: 0 if OK, -1 on error</i>	p. 514
int	<b>aio_write</b> (struct aiocb *aiocb); <i>&lt;aio.h&gt;</i> <i>Returns: 0 if OK, -1 on error</i>	p. 512
unsigned int	<b>alarm</b> (unsigned int seconds); <i>&lt;unistd.h&gt;</i> <i>Returns: 0 or number of seconds until previously set alarm</i>	p. 338
int	<b>atexit</b> (void (*func)(void)); <i>&lt;stdlib.h&gt;</i> <i>Returns: 0 if OK, nonzero on error</i>	p. 200

int	<b>bind(int sockfd, const struct sockaddr *addr, socklen_t len);</b>	
	<sys/socket.h>	p. 604
	Returns: 0 if OK, -1 on error	
void	<b>*calloc(size_t nobj, size_t size);</b>	
	<stdlib.h>	p. 207
	Returns: non-null pointer if OK, NULL on error	
speed_t	<b>cgetattrspeed(const struct termios *termptr);</b>	
	<termios.h>	p. 692
	Returns: baud rate value	
speed_t	<b>cgetospeed(const struct termios *termptr);</b>	
	<termios.h>	p. 692
	Returns: baud rate value	
int	<b>cfsetispeed(struct termios *termptr, speed_t speed);</b>	
	<termios.h>	p. 692
	Returns: 0 if OK, -1 on error	
int	<b>cfsetospeed(struct termios *termptr, speed_t speed);</b>	
	<termios.h>	p. 692
	Returns: 0 if OK, -1 on error	
int	<b>chdir(const char *path);</b>	
	<unistd.h>	p. 135
	Returns: 0 if OK, -1 on error	
int	<b>chmod(const char *path, mode_t mode);</b>	
	<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	<b>chown(const char *path, uid_t owner, gid_t group);</b>	
	<unistd.h>	p. 109
	Returns: 0 if OK, -1 on error	
void	<b>clearerr(FILE *fp);</b>	
	<stdio.h>	p. 151
int	<b>clock_gettime(clockid_t clock_id, struct timespec *tsp);</b>	
	<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	<b>clock_gettime</b> (clockid_t <i>clock_id</i> , struct timespec * <i>tsp</i> );		
	<sys/time.h>		p. 189
	<i>clock_id</i> : CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID		
	Returns: 0 if OK, -1 on error		
int	<b>clock_nanosleep</b> (clockid_t <i>clock_id</i> , int <i>flags</i> , const struct timespec * <i>reqtp</i> , struct timespec * <i>remtp</i> );		
	<time.h>		p. 375
	<i>clock_id</i> : CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID		
	<i>flags</i> : TIMER_ABSTIME		
	Returns: 0 if slept for requested time, error number on failure		
int	<b>clock_settime</b> (clockid_t <i>clock_id</i> , const struct timespec * <i>tsp</i> );		
	<sys/time.h>		p. 190
	<i>clock_id</i> : CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, CLOCK_THREAD_CPUTIME_ID		
	Returns: 0 if OK, -1 on error		
int	<b>close</b> (int <i>fd</i> );		
	<unistd.h>		p. 66
	Returns: 0 if OK, -1 on error		
int	<b>closedir</b> (DIR * <i>dp</i> );		
	<dirent.h>		p. 130
	Returns: 0 if OK, -1 on error		
void	<b>closelog</b> (void);		
	<syslog.h>		p. 470
unsigned char	<b>*CMMSG_DATA</b> (struct cmsghdr * <i>cp</i> );		
	<sys/socket.h>		p. 645
	Returns: pointer to data associated with cmsghdr structure		
struct cmsghdr	<b>*CMMSG_FIRSTHDR</b> (struct msghdr * <i>mp</i> );		
	<sys/socket.h>		p. 645
	Returns: pointer to first cmsghdr structure associated with the msghdr structure, or NULL if none exists		

unsigned			
int	<b>CMSG_LEN</b> (unsigned int nbytes);		
	<sys/socket.h>		p. 645
	Returns: size to allocate for data object nbytes large		
struct			
cmsghdr	* <b>CMSG_NXTHDR</b> (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	<b>connect</b> (int sockfd, const struct sockaddr *addr, socklen_t len);		
	<sys/socket.h>		p. 605
	Returns: 0 if OK, -1 on error		
int	<b>creat</b> (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	* <b>ctermid</b> (char *ptr);		
	<stdio.h>		p. 694
	Returns: pointer to name of controlling terminal on success, pointer to empty string on error		
int	<b>dprintf</b> (int fd, const char *restrict format, ...);		
	<stdio.h>		p. 159
	Returns: number of characters output if OK, negative value if output error		
int	<b>dup</b> (int fd);		
	<unistd.h>		p. 79
	Returns: new file descriptor if OK, -1 on error		
int	<b>dup2</b> (int fd, int fd2);		
	<unistd.h>		p. 79
	Returns: new file descriptor if OK, -1 on error		
void	<b>endgrent</b> (void);		
	<grp.h>		p. 183
void	<b>endhostent</b> (void);		
	<netdb.h>		p. 597

void	<b>endnetent(void);</b>	<netdb.h>	p. 598
void	<b>endprotoent(void);</b>	<netdb.h>	p. 598
void	<b>endpwent(void);</b>	<pwd.h>	p. 180
void	<b>endservent(void);</b>	<netdb.h>	p. 599
void	<b>endspent(void);</b>	<shadow.h> Platforms: Linux 3.2.0, Solaris 10	p. 182
int	<b>execl(const char *path, const char *arg0, ... /* (char *) 0 */ );</b>	<unistd.h>	p. 249 Returns: -1 on error, no return on success
int	<b>execle(const char *path, const char *arg0, ... /* (char *) 0, char *const envp[] */ );</b>	<unistd.h>	p. 249 Returns: -1 on error, no return on success
int	<b>execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );</b>	<unistd.h>	p. 249 Returns: -1 on error, no return on success
int	<b>execv(const char *path, char *const argv[]);</b>	<unistd.h>	p. 249 Returns: -1 on error, no return on success
int	<b>execve(const char *path, char *const argv[], char *const envp[]);</b>	<unistd.h>	p. 249 Returns: -1 on error, no return on success
int	<b>execvp(const char *filename, char *const argv[]);</b>	<unistd.h>	p. 249 Returns: -1 on error, no return on success
void	<b>_Exit(int status);</b>	<stdlib.h>	p. 198 This function never returns

---

void	<b>_exit(int status);</b>	<unistd.h>	p. 198
		This function never returns	
void	<b>exit(int status);</b>	<stdlib.h>	p. 198
		This function never returns	
int	<b>faccessat(int fd, const char *path, int mode, int flag);</b>	<unistd.h>	p. 102
		mode: R_OK, W_OK, X_OK, F_OK	
		flag: AT_EACCESS	
		Returns: 0 if OK, -1 on error	
int	<b>fchdir(int fd);</b>	<unistd.h>	p. 135
		Returns: 0 if OK, -1 on error	
int	<b>fchmod(int fd, mode_t mode);</b>	<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	<b>fchmodat(int fd, const char *path, mode_t mode, int flag);</b>	<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	<b>fchown(int fd, uid_t owner, gid_t group);</b>	<unistd.h>	p. 109
		Returns: 0 if OK, -1 on error	
int	<b>fchownat(int fd, const char *path, uid_t owner, gid_t group, int flag);</b>	<unistd.h>	p. 109
		flag: AT_SYMLINK_NOFOLLOW	
		Returns: 0 if OK, -1 on error	
int	<b>fclose(FILE *fp);</b>	<stdio.h>	p. 150
		Returns: 0 if OK, EOF on error	

int	<b>fcntl</b> (int <i>fd</i> , int <i>cmd</i> , ... /* int arg */ );	<fcntl.h>	p. 82
	<i>cmd</i> : 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 <i>cmd</i> if OK, -1 on error		
int	<b>fdatasync</b> (int <i>fd</i> );	<unistd.h>	p. 81
	Returns: 0 if OK, -1 on error		
	Platforms: Linux 3.2.0, Solaris 10		
void	<b>FD_CLR</b> (int <i>fd</i> , fd_set * <i>fdset</i> );	<sys/select.h>	p. 503
int	<b>FD_ISSET</b> (int <i>fd</i> , fd_set * <i>fdset</i> );	<sys/select.h>	p. 503
	Returns: nonzero if <i>fd</i> is in set, 0 otherwise		
FILE	* <b>fdopen</b> (int <i>fd</i> , const char * <i>type</i> );	<stdio.h>	p. 148
	<i>type</i> : "r", "w", "a", "r+", "w+", "a+"		
	Returns: file pointer if OK, NULL on error		
DIR	* <b>fdopendir</b> (int <i>fd</i> );	<dirent.h>	p. 130
	Returns: pointer if OK, NULL on error		
void	<b>FD_SET</b> (int <i>fd</i> , fd_set * <i>fdset</i> );	<sys/select.h>	p. 503
void	<b>FD_ZERO</b> (fd_set * <i>fdset</i> );	<sys/select.h>	p. 503
int	<b>feof</b> (FILE * <i>fp</i> );	<stdio.h>	p. 151
	Returns: nonzero (true) if end of file on stream, 0 (false) otherwise		
int	<b>ferror</b> (FILE * <i>fp</i> );	<stdio.h>	p. 151
	Returns: nonzero (true) if error on stream, 0 (false) otherwise		
int	<b>execve</b> (int <i>fd</i> , char *const <i>argv</i> [], char *const <i>envp</i> []);	<unistd.h>	p. 249
	Returns: -1 on error, no return on success		

int	<b>fflush(FILE *fp);</b> <stdio.h> Returns: 0 if OK, EOF on error	p. 147
int	<b>fgetc(FILE *fp);</b> <stdio.h> Returns: next character if OK, EOF on end of file or error	p. 150
int	<b>fgetpos(FILE *restrict fp, fpos_t *restrict pos);</b> <stdio.h> Returns: 0 if OK, nonzero on error	p. 158
char	<b>*fgets(char *restrict buf, int n, FILE *restrict fp);</b> <stdio.h> Returns: buf if OK, NULL on end of file or error	p. 152
int	<b>fileno(FILE *fp);</b> <stdio.h> Returns: file descriptor associated with the stream, -1 on error	p. 164
void	<b>flockfile(FILE *fp);</b> <stdio.h>	p. 443
FILE	<b>*fmemopen(void *restrict buf, size_t size,</b> <b>const char *restrict type);</b> <stdio.h> <b>type: "r", "w", "a", "r+", "w+", "a+"</b> Returns: stream pointer if OK, NULL on error	p. 171
FILE	<b>*fopen(const char *restrict path, const char *restrict type);</b> <stdio.h> <b>type: "r", "w", "a", "r+", "w+", "a+"</b> Returns: file pointer if OK, NULL on error	p. 148
pid_t	<b>fork(void);</b> <unistd.h> Returns: 0 in child, process ID of child in parent, -1 on error	p. 229
long	<b>fpathconf(int fd, int name);</b> <unistd.h> <b>name:</b> _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	p. 42

int	<b>fprintf</b> (FILE *restrict <i>fp</i> , const char *restrict <i>format</i> , ...); <stdio.h>	p. 159
	>Returns: number of characters output if OK, negative value if output error	
int	<b>fputc</b> (int <i>c</i> , FILE * <i>fp</i> ); <stdio.h>	p. 152
	>Returns: <i>c</i> if OK, EOF on error	
int	<b>fputs</b> (const char *restrict <i>str</i> , FILE *restrict <i>fp</i> ); <stdio.h>	p. 153
	>Returns: non-negative value if OK, EOF on error	
size_t	<b>fread</b> (void *restrict <i>ptr</i> , size_t <i>size</i> , size_t <i>nobj</i> , FILE *restrict <i>fp</i> ); <stdio.h>	p. 156
	>Returns: number of objects read	
void	<b>free</b> (void * <i>ptr</i> ); <stdlib.h>	p. 207
void	<b>freeaddrinfo</b> (struct addrinfo * <i>ai</i> ); <sys/socket.h> <netdb.h>	p. 599
FILE	<b>*freopen</b> (const char *restrict <i>path</i> , const char *restrict <i>type</i> , FILE *restrict <i>fp</i> ); <stdio.h>	p. 148
	<i>type</i> : "r", "w", "a", "r+", "w+", "a+" >Returns: file pointer if OK, NULL on error	
int	<b>fscanf</b> (FILE *restrict <i>fp</i> , const char *restrict <i>format</i> , ...); <stdio.h>	p. 162
	>Returns: number of input items assigned, EOF if input error or end of file before any conversion	
int	<b>fseek</b> (FILE * <i>fp</i> , long <i>offset</i> , int <i>whence</i> ); <stdio.h>	p. 158
	<i>whence</i> : SEEK_SET, SEEK_CUR, SEEK_END >Returns: 0 if OK, -1 on error	
int	<b>fseeko</b> (FILE * <i>fp</i> , off_t <i>offset</i> , int <i>whence</i> ); <stdio.h>	p. 158
	<i>whence</i> : SEEK_SET, SEEK_CUR, SEEK_END >Returns: 0 if OK, -1 on error	

int	<b>fsetpos(FILE *fp, const fpos_t *pos);</b> <stdio.h> Returns: 0 if OK, nonzero on error	p. 158
int	<b>fstat(int fd, struct stat *buf);</b> <sys/stat.h> Returns: 0 if OK, -1 on error	p. 93
int	<b>fstatat(int fd, const char *restrict path,               struct stat *restrict buf, int flag);</b> <sys/stat.h> <i>flag:</i> AT_SYMLINK_NOFOLLOW Returns: 0 if OK, -1 on error	p. 93
int	<b>fsync(int fd);</b> <unistd.h> Returns: 0 if OK, -1 on error	p. 81
long	<b>ftell(FILE *fp);</b> <stdio.h> Returns: current file position indicator if OK, -1L on error	p. 158
off_t	<b>ftello(FILE *fp);</b> <stdio.h> Returns: current file position indicator if OK, (off_t)-1 on error	p. 158
key_t	<b>ftok(const char *path, int id);</b> <sys/ipc.h> Returns: key if OK, (key_t)-1 on error	p. 557
int	<b>ftruncate(int fd, off_t length);</b> <unistd.h> Returns: 0 if OK, -1 on error	p. 112
int	<b>ftrylockfile(FILE *fp);</b> <stdio.h> Returns: 0 if OK, nonzero if lock can't be acquired	p. 443
void	<b>funlockfile(FILE *fp);</b> <stdio.h>	p. 443
int	<b>futimens(int fd, const struct timespec times[2]);</b> <sys/stat.h> Returns: 0 if OK, -1 on error	p. 126

---

<code>int        <b>fwide(FILE *fp, int mode);</b></code>	<code>&lt;stdio.h&gt;</code>	p. 144
<code>&lt;wchar.h&gt;</code>		
Returns: positive if stream is wide oriented, negative if stream is byte oriented, or 0 if stream has no orientation		
<code>size_t      <b>fwrite(const void *restrict ptr, size_t size, size_t nobj,</b></code>	<code>FILE *restrict fp);</code>	
<code>&lt;stdio.h&gt;</code>		
Returns: number of objects written		
<code>const</code>		
<code>char        *gai_strerror(int error);</code>	<code>&lt;netdb.h&gt;</code>	p. 600
Returns: a pointer to a string describing the error		
<code>int        <b>getaddrinfo(const char *restrict host,</b></code>	<code>const char *restrict service,</code>	
<code>const struct addrinfo *restrict hint,</code>		
<code>struct addrinfo **restrict res);</code>		
<code>&lt;sys/socket.h&gt;</code>		
<code>&lt;netdb.h&gt;</code>		
Returns: 0 if OK, nonzero error code on error		
<code>int        <b>getc(FILE *fp);</b></code>	<code>&lt;stdio.h&gt;</code>	p. 150
Returns: next character if OK, EOF on end of file or error		
<code>int        <b>getchar(void);</b></code>	<code>&lt;stdio.h&gt;</code>	p. 150
Returns: next character if OK, EOF on end of file or error		
<code>int        <b>getchar_unlocked(void);</b></code>	<code>&lt;stdio.h&gt;</code>	p. 444
Returns: the next character if OK, EOF on end of file or error		
<code>int        <b>getc_unlocked(FILE *fp);</b></code>	<code>&lt;stdio.h&gt;</code>	p. 444
Returns: the next character if OK, EOF on end of file or error		
<code>char        *<b>getcwd(char *buf, size_t size);</b></code>	<code>&lt;unistd.h&gt;</code>	p. 136
Returns: <i>buf</i> if OK, NULL on error		
<code>gid_t        <b>getegid(void);</b></code>	<code>&lt;unistd.h&gt;</code>	p. 228
Returns: effective group ID of calling process		

---

char	<b>*getenv(const char *name);</b>	<stdlib.h>	p. 210
	Returns: pointer to value associated with <i>name</i> , NULL if not found		
uid_t	<b>geteuid(void);</b>	<unistd.h>	p. 228
	Returns: effective user ID of calling process		
gid_t	<b>getgid(void);</b>	<unistd.h>	p. 228
	Returns: real group ID of calling process		
struct group	<b>*getgrent(void);</b>	<grp.h>	p. 183
	Returns: pointer if OK, NULL on error or end of file		
struct group	<b>*getgrgid(gid_t gid);</b>	<grp.h>	p. 182
	Returns: pointer if OK, NULL on error		
struct group	<b>*getgrnam(const char *name);</b>	<grp.h>	p. 182
	Returns: pointer if OK, NULL on error		
int	<b>getgroups(int gidsetsize, gid_t grouplist[]);</b>	<unistd.h>	p. 184
	Returns: number of supplementary group IDs if OK, -1 on error		
struct hostent	<b>*gethostent(void);</b>	<netdb.h>	p. 597
	Returns: pointer if OK, NULL on error		
int	<b>gethostname(char *name, int namelen);</b>	<unistd.h>	p. 188
	Returns: 0 if OK, -1 on error		
char	<b>*getlogin(void);</b>	<unistd.h>	p. 275
	Returns: pointer to string giving login name if OK, NULL on error		

int	<b>getnameinfo</b> (const struct sockaddr *restrict <i>addr</i> , socklen_t <i>alen</i> , char *restrict <i>host</i> , socklen_t <i>hostlen</i> , char *restrict <i>service</i> , socklen_t <i>servlen</i> , unsigned int <i>flags</i> ); <sys/socket.h> <netdb.h>	p. 600
	<i>flags</i> : NI_DGRAM, NI_NAMEREQD, NI_NOFQDN, NI_NUMERICHOST, NI_NUMERICSCOPE, NI_NUMERICSERV	
	Returns: 0 if OK, nonzero on error	
struct netent	* <b>getnetbyaddr</b> (uint32_t <i>net</i> , int <i>type</i> ); <netdb.h>	p. 598
	Returns: pointer if OK, NULL on error	
struct netent	* <b>getnetbyname</b> (const char * <i>name</i> ); <netdb.h>	p. 598
	Returns: pointer if OK, NULL on error	
struct netent	* <b>getnetent</b> (void); <netdb.h>	p. 598
	Returns: pointer if OK, NULL on error	
int	<b>getopt</b> (int <i>argc</i> , char * const <i>argv</i> [], const char * <i>options</i> ); <fcntl.h> extern int <i>opterr</i> , <i>optind</i> , <i>optopt</i> ; extern char * <i>optarg</i> ;	p. 662
	Returns: the next option character, or -1 when all options have been processed	
int	<b>getpeername</b> (int <i>sockfd</i> , struct sockaddr *restrict <i>addr</i> , socklen_t *restrict <i>alenp</i> ); <sys/socket.h>	p. 605
	Returns: 0 if OK, -1 on error	
pid_t	<b>getpgid</b> (pid_t <i>pid</i> ); <unistd.h>	p. 294
	Returns: process group ID if OK, -1 on error	
pid_t	<b>getpgrp</b> (void); <unistd.h>	p. 293
	Returns: process group ID of calling process	

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

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

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

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

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

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

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

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

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

int	<b>getrlimit</b> (int <i>resource</i> , struct rlimit * <i>rlptr</i> ); <i>&lt;sys/resource.h&gt;</i> <i>resource:</i> 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)	p. 220
	Returns: 0 if OK, -1 on error	
char	<b>*gets</b> (char * <i>buf</i> ); <i>&lt;stdio.h&gt;</i>	p. 152
	Returns: <i>buf</i> if OK, NULL on end of file or error	
struct servent	<b>*getservbyname</b> (const char * <i>name</i> , const char * <i>proto</i> ); <i>&lt;netdb.h&gt;</i>	p. 599
	Returns: pointer if OK, NULL on error	
struct servent	<b>*getservbyport</b> (int <i>port</i> , const char * <i>proto</i> ); <i>&lt;netdb.h&gt;</i>	p. 599
	Returns: pointer if OK, NULL on error	
struct servent	<b>*getservent</b> (void); <i>&lt;netdb.h&gt;</i>	p. 599
	Returns: pointer if OK, NULL on error	
pid_t	<b>getsid</b> (pid_t <i>pid</i> ); <i>&lt;unistd.h&gt;</i>	p. 296
	Returns: session leader's process group ID if OK, -1 on error	
int	<b>getsockname</b> (int <i>sockfd</i> , struct sockaddr * <i>restrict addr</i> , <i>socklen_t *restrict alenp</i> ); <i>&lt;sys/socket.h&gt;</i>	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>
                  Returns: 0 if OK, -1 on error
                  p. 624

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

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

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

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

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

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

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

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

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 efds, const char *existingpath, int nfds,		
const char *newpath, int flag);	<unistd.h>	
	flag: AT_SYMLINK_NOFOLLOW	p. 116
	Returns: 0 if OK, -1 on error	

int	<b>lio_listio</b> (int <i>mode</i> , struct aiocb *restrict const <i>list</i> [restrict], int <i>nent</i> , struct sigevent *restrict <i>sigev</i> );  <i>&lt;aio.h&gt;</i>	p. 515
	<i>mode</i> : LIO_NOWAIT, LIO_WAIT Returns: 0 if OK, -1 on error	
int	<b>listen</b> (int <i>sockfd</i> , int <i>backlog</i> );  <i>&lt;sys/socket.h&gt;</i>	p. 608
	Returns: 0 if OK, -1 on error	
struct tm	<b>*localtime</b> (const time_t * <i>calptr</i> );  <i>&lt;time.h&gt;</i>	p. 192
	Returns: pointer to broken-down time, NULL on error	
void	<b>longjmp</b> (jmp_buf <i>env</i> , int <i>val</i> );  <i>&lt;setjmp.h&gt;</i>	p. 215
	This function never returns	
off_t	<b>lseek</b> (int <i>fd</i> , off_t <i>offset</i> , int <i>whence</i> );  <i>&lt;unistd.h&gt;</i>	p. 67
	<i>whence</i> : SEEK_SET, SEEK_CUR, SEEK_END Returns: new file offset if OK, -1 on error	
int	<b>lstat</b> (const char *restrict <i>path</i> , struct stat *restrict <i>buf</i> );  <i>&lt;sys/stat.h&gt;</i>	p. 93
	Returns: 0 if OK, -1 on error	
void	<b>*malloc</b> (size_t <i>size</i> );  <i>&lt;stdlib.h&gt;</i>	p. 207
	Returns: non-null pointer if OK, NULL on error	
int	<b>mkdir</b> (const char * <i>path</i> , mode_t <i>mode</i> );  <i>&lt;sys/stat.h&gt;</i>	p. 129
	<i>mode</i> : S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH) Returns: 0 if OK, -1 on error	
int	<b>mkdirat</b> (int <i>fd</i> , const char * <i>path</i> , mode_t <i>mode</i> );  <i>&lt;sys/stat.h&gt;</i>	p. 129
	<i>mode</i> : S_IS[UG]ID, S_ISVTX, S_I[RWX](USR GRP OTH) Returns: 0 if OK, -1 on error	

char	<b>*mkdtemp(char *template);</b>	<stdlib.h>	p. 169
Returns: pointer to directory name if OK, NULL on error			
int	<b>mkfifo(const char *path, mode_t mode);</b>	<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	<b>mkfifoat(int fd, const char *path, mode_t mode);</b>	<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	<b>mkstemp(char *template);</b>	<stdlib.h>	p. 169
Returns: file descriptor if OK, -1 on error			
time_t	<b>mktime(struct tm *tmptr);</b>	<time.h>	p. 192
Returns: calendar time if OK, -1 on error			
void	<b>*mmap(void *addr, size_t len, int prot, int flag, int fd,</b>	<sys/mman.h>	p. 525
off_t off); 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	<b>mprotect(void *addr, size_t len, int prot);</b>	<sys/mman.h>	p. 527
Returns: 0 if OK, -1 on error			
int	<b>msgctl(int msqid, int cmd, struct msqid_ds *buf);</b>	<sys/msg.h>	p. 562
cmd: IPC_STAT, IPC_SET, IPC_RMID Returns: 0 if OK, -1 on error			
int	<b>msgget(key_t key, int flag);</b>	<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	<b>openat</b> (int <i>fd</i> , const char * <i>path</i> , int <i>oflag</i> , ... /* mode_t <i>mode</i> */ );	
	<fcntl.h>	p. 62
	<i>oflag</i> : 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 <i>mode</i> : 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	<b>*opendir</b> (const char * <i>path</i> );	
	<dirent.h>	p. 130
	Returns: pointer if OK, NULL on error	
void	<b>openlog</b> (const char * <i>ident</i> , int <i>option</i> , int <i>facility</i> );	
	<syslog.h>	p. 470
	<i>option</i> : LOG_CONS, LOG_NDELAY, LOG_NOWAIT, LOG_ODELAY, LOG_PERROR, LOG_PID	
	<i>facility</i> : 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	<b>*open_memstream</b> (char ** <i>bufp</i> , size_t * <i>sizep</i> );	
	<stdio.h>	p. 173
	Returns: stream pointer if OK, NULL on error	
FILE	<b>*open_wmemstream</b> (wchar_t ** <i>bufp</i> , size_t * <i>sizep</i> );	
	<wchar.h>	p. 173
	Returns: stream pointer if OK, NULL on error	
long	<b>pathconf</b> (const char * <i>path</i> , int <i>name</i> );	
	<unistd.h>	p. 42
	<i>name</i> : _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	<b>pause</b> (void);	
	<unistd.h>	p. 338
	Returns: -1 with errno set to EINTR	

int	<b>pclose(FILE *fp);</b>	<stdio.h>	p. 541
Returns: termination status of <code>popen cmdstring</code> , -1 on error			
void	<b>perror(const char *msg);</b>	<stdio.h>	p. 15
int	<b>pipe(int fd[2]);</b>	<unistd.h>	p. 535
Returns: 0 if OK, -1 on error			
int	<b>poll(struct pollfd fdarray[], nfds_t nfds, int timeout);</b>	<poll.h>	p. 506
Returns: count of ready descriptors, 0 on timeout, -1 on error			
FILE	<b>*popen(const char *cmdstring, const char *type);</b>	<stdio.h>	p. 541
type: "r", "w"			
Returns: file pointer if OK, NULL on error			
int	<b>posix_openpt(int oflag);</b>	<stdlib.h>	p. 722
<fcntl.h>			
oflag: O_RDWR, O_NOCTTY			
Returns: file descriptor of next available PTY master if OK, -1 on error			
ssize_t	<b>pread(int fd, void *buf, size_t nbytes, off_t offset);</b>	<unistd.h>	p. 78
Returns: number of bytes read, 0 if end of file, -1 on error			
int	<b>printf(const char *restrict format, ...);</b>	<stdio.h>	p. 159
Returns: number of characters output if OK, negative value if output error			
int	<b>pselect(int maxfdp1, fd_set *restrict readfds,</b>	fd_set *restrict writefds, fd_set *restrict exceptfds,	
const struct timespec *restrict tsptx,			
const sigset(SIG_SETSIG);			
<sys/select.h>			
Returns: count of ready descriptors, 0 on timeout, -1 on error			
void	<b>psiginfo(const siginfo_t *info, const char *msg);</b>	<signal.h>	p. 379

void	<b>psignal</b> (int <i>signo</i> , const char * <i>msg</i> );		
	<signal.h>		
	<siginfo.h> /* on Solaris */		p. 379
int	<b>pthread_atfork</b> (void (* <i>prepare</i> )(void), void (* <i>parent</i> )(void),		
	void (* <i>child</i> )(void));		
	<pthread.h>		p. 458
	Returns: 0 if OK, error number on failure		
int	<b>pthread_attr_destroy</b> (pthread_attr_t * <i>attr</i> );		
	<pthread.h>		p. 427
	Returns: 0 if OK, error number on failure		
int	<b>pthread_attr_getdetachstate</b> (const pthread_attr_t * <i>attr</i> ,		
	int * <i>detachstate</i> );		
	<pthread.h>		p. 428
	Returns: 0 if OK, error number on failure		
int	<b>pthread_attr_getguardsize</b> (const pthread_attr_t		
	*restrict <i>attr</i> ,		
	size_t *restrict <i>guardsize</i> );		
	<pthread.h>		p. 430
	Returns: 0 if OK, error number on failure		
int	<b>pthread_attr_getstack</b> (const pthread_attr_t *restrict <i>attr</i> ,		
	void **restrict <i>stackaddr</i> ,		
	size_t *restrict <i>stacksize</i> );		
	<pthread.h>		p. 429
	Returns: 0 if OK, error number on failure		
int	<b>pthread_attr_getstacksize</b> (const pthread_attr_t		
	*restrict <i>attr</i> ,		
	size_t *restrict <i>stacksize</i> );		
	<pthread.h>		p. 430
	Returns: 0 if OK, error number on failure		
int	<b>pthread_attr_init</b> (pthread_attr_t * <i>attr</i> );		
	<pthread.h>		p. 427
	Returns: 0 if OK, error number on failure		
int	<b>pthread_attr_setdetachstate</b> (pthread_attr_t * <i>attr</i> ,		
	int <i>detachstate</i> );		
	<pthread.h>		p. 428
	<i>detachstate</i> : 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	<b>pthread_barrier_wait</b> (pthread_barrier_t *barrier); <pthread.h>	p. 419
	Returns: 0 or PTHREAD_BARRIER_SERIAL_THREAD if OK, error number on failure	
int	<b>pthread_cancel</b> (pthread_t tid); <pthread.h>	p. 393
	Returns: 0 if OK, error number on failure	
void	<b>pthread_cleanup_pop</b> (int execute); <pthread.h>	p. 394
void	<b>pthread_cleanup_push</b> (void (*rtn)(void *), void *arg); <pthread.h>	p. 394
int	<b>pthread_condattr_destroy</b> (pthread_condattr_t *attr); <pthread.h>	p. 440
	Returns: 0 if OK, error number on failure	
int	<b>pthread_condattr_getclock</b> (const pthread_condattr_t *restrict attr, clockid_t *restrict clock_id); <pthread.h>	p. 441
	Returns: 0 if OK, error number on failure	
int	<b>pthread_condattr_getpshared</b> (const pthread_condattr_t *restrict attr, int *restrict pshared); <pthread.h>	p. 440
	Returns: 0 if OK, error number on failure	
int	<b>pthread_condattr_init</b> (pthread_condattr_t *attr); <pthread.h>	p. 440
	Returns: 0 if OK, error number on failure	
int	<b>pthread_condattr_setclock</b> (pthread_condattr_t *attr, clockid_t clock_id); <pthread.h>	p. 441
	Returns: 0 if OK, error number on failure	
int	<b>pthread_condattr_setpshared</b> (pthread_condattr_t *attr, int pshared); <pthread.h>	p. 440
	<i>pshared:</i> PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED	
	Returns: 0 if OK, error number on failure	

int	<b>pthread_cond_broadcast</b> (pthread_cond_t *cond); <pthread.h> Returns: 0 if OK, error number on failure	p. 415
int	<b>pthread_cond_destroy</b> (pthread_cond_t *cond); <pthread.h> Returns: 0 if OK, error number on failure	p. 414
int	<b>pthread_cond_init</b> (pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr); <pthread.h> Returns: 0 if OK, error number on failure	p. 414
int	<b>pthread_cond_signal</b> (pthread_cond_t *cond); <pthread.h> Returns: 0 if OK, error number on failure	p. 415
int	<b>pthread_cond_timedwait</b> (pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict timeout); <pthread.h> Returns: 0 if OK, error number on failure	p. 414
int	<b>pthread_cond_wait</b> (pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex); <pthread.h> Returns: 0 if OK, error number on failure	p. 414
int	<b>pthread_create</b> (pthread_t *restrict tidp, const pthread_attr_t *restrict attr, void *(*start_rtn)(void *), void *restrict arg); <pthread.h> Returns: 0 if OK, error number on failure	p. 385
int	<b>pthread_detach</b> (pthread_t tid); <pthread.h> Returns: 0 if OK, error number on failure	p. 397
int	<b>pthread_equal</b> (pthread_t tid1, pthread_t tid2); <pthread.h> Returns: nonzero if equal, 0 otherwise	p. 385
void	<b>pthread_exit</b> (void *rval_ptr); <pthread.h>	p. 389

void	<b>*pthread_getspecific(pthread_key_t key);</b>	<pthread.h>	p. 449
	Returns: thread-specific data value or NULL if no value has been associated with the key		
int	<b>pthread_join(pthread_t thread, void **rval_ptr);</b>	<pthread.h>	p. 389
	Returns: 0 if OK, error number on failure		
int	<b>pthread_key_create(pthread_key_t *keyp,</b> <b>              void (*destructor)(void *));</b>	<pthread.h>	p. 447
	Returns: 0 if OK, error number on failure		
int	<b>pthread_key_delete(pthread_key_t key);</b>	<pthread.h>	p. 448
	Returns: 0 if OK, error number on failure		
int	<b>pthread_kill(pthread_t thread, int signo);</b>	<signal.h>	p. 455
	Returns: 0 if OK, error number on failure		
int	<b>pthread_mutexattr_destroy(pthread_mutexattr_t *attr);</b>	<pthread.h>	p. 431
	Returns: 0 if OK, error number on failure		
int	<b>pthread_mutexattr_getpshared(const pthread_mutexattr_t</b> <b>                        *restrict attr,</b> <b>                        int *restrict pshared);</b>	<pthread.h>	p. 431
	Returns: 0 if OK, error number on failure		
int	<b>pthread_mutexattr_getrobust(const pthread_mutexattr_t</b> <b>                        *restrict attr,</b> <b>                        int *restrict robust);</b>	<pthread.h>	p. 432
	Returns: 0 if OK, error number on failure		
int	<b>pthread_mutexattr_gettype(const pthread_mutexattr_t</b> <b>                        *restrict attr,</b> <b>                        int *restrict type);</b>	<pthread.h>	p. 434
	Returns: 0 if OK, error number on failure		
int	<b>pthread_mutexattr_init(pthread_mutexattr_t *attr);</b>	<pthread.h>	p. 431
	Returns: 0 if OK, error number on failure		

int	<b>pthread_mutexattr_setpshared</b> (pthread_mutexattr_t *attr, int pshared);	
	<b>&lt;pthread.h&gt;</b>	p. 431
	pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED	
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutexattr_setrobust</b> (pthread_mutexattr_t *attr, int robust);	
	<b>&lt;pthread.h&gt;</b>	p. 432
	robust: PTHREAD_MUTEX_ROBUST, PTHREAD_MUTEX_STALLED	
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutexattr_settype</b> (pthread_mutexattr_t *attr, int type);	
	<b>&lt;pthread.h&gt;</b>	p. 434
	type: PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_ERRORCHECK, PTHREAD_MUTEX_RECURSIVE, PTHREAD_MUTEX_DEFAULT	
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutex_consistent</b> (pthread_mutex_t * mutex);	
	<b>&lt;pthread.h&gt;</b>	p. 433
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutex_destroy</b> (pthread_mutex_t *mutex);	
	<b>&lt;pthread.h&gt;</b>	p. 400
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutex_init</b> (pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);	
	<b>&lt;pthread.h&gt;</b>	p. 400
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutex_lock</b> (pthread_mutex_t *mutex);	
	<b>&lt;pthread.h&gt;</b>	p. 400
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutex_timedlock</b> (pthread_mutex_t *restrict mutex, const struct timespec *restrict tspr);	
	<b>&lt;pthread.h&gt;</b>	p. 407
	<b>&lt;time.h&gt;</b>	
	Returns: 0 if OK, error number on failure	
int	<b>pthread_mutex_trylock</b> (pthread_mutex_t *mutex);	
	<b>&lt;pthread.h&gt;</b>	p. 400
	Returns: 0 if OK, error number on failure	

int	<b>pthread_mutex_unlock</b> (pthread_mutex_t *mutex); <pthread.h> Returns: 0 if OK, error number on failure	p. 400
int	<b>pthread_once</b> (pthread_once_t *initflag, void (*initfn)(void)); <pthread.h> pthread_once_t initflag = PTHREAD_ONCE_INIT; Returns: 0 if OK, error number on failure	p. 448
int	<b>pthread_rwlockattr_destroy</b> (pthread_rwlockattr_t *attr); <pthread.h> Returns: 0 if OK, error number on failure	p. 439
int	<b>pthread_rwlockattr_getpshared</b> (const pthread_rwlockattr_t *restrict attr, int *restrict pshared); <pthread.h> Returns: 0 if OK, error number on failure	p. 440
int	<b>pthread_rwlockattr_init</b> (pthread_rwlockattr_t *attr); <pthread.h> Returns: 0 if OK, error number on failure	p. 439
int	<b>pthread_rwlockattr_setpshared</b> (pthread_rwlockattr_t *attr, int pshared); <pthread.h> pshared: PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED Returns: 0 if OK, error number on failure	p. 440
int	<b>pthread_rwlock_destroy</b> (pthread_rwlock_t *rwlock); <pthread.h> Returns: 0 if OK, error number on failure	p. 409
int	<b>pthread_rwlock_init</b> (pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr); <pthread.h> Returns: 0 if OK, error number on failure	p. 409
int	<b>pthread_rwlock_rdlock</b> (pthread_rwlock_t *rwlock); <pthread.h> Returns: 0 if OK, error number on failure	p. 410

```
int      pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock,
                                     const struct timespec
                                     *restrict tspr);
          <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 tspr);
          <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	<b>pthread_setspecific(pthread_key_t key, const void *value);</b>	<pthread.h>	p. 449
Returns: 0 if OK, error number on failure			
int	<b>pthread_sigmask(int how, const sigset_t *restrict set,</b>	<b>sigset_t *restrict oset);</b>	p. 454
<signal.h>			
<i>how:</i> SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK			
Returns: 0 if OK, error number on failure			
int	<b>pthread_spin_destroy(pthread_spinlock_t *lock);</b>	<pthread.h>	p. 417
Returns: 0 if OK, error number on failure			
int	<b>pthread_spin_init(pthread_spinlock_t *lock, int pshared);</b>	<pthread.h>	p. 417
<i>pshared:</i> PTHREAD_PROCESS_PRIVATE, PTHREAD_PROCESS_SHARED			
Returns: 0 if OK, error number on failure			
int	<b>pthread_spin_lock(pthread_spinlock_t *lock);</b>	<pthread.h>	p. 418
Returns: 0 if OK, error number on failure			
int	<b>pthread_spin_trylock(pthread_spinlock_t *lock);</b>	<pthread.h>	p. 418
Returns: 0 if OK, error number on failure			
int	<b>pthread_spin_unlock(pthread_spinlock_t *lock);</b>	<pthread.h>	p. 418
Returns: 0 if OK, error number on failure			
void	<b>pthread_testcancel(void);</b>	<pthread.h>	p. 453
char	<b>*ptsname(int fd);</b>	<stdlib.h>	p. 723
Returns: pointer to name of PTY slave if OK, NULL on error			
int	<b>putc(int c, FILE *fp);</b>	<stdio.h>	p. 152
Returns: <i>c</i> if OK, EOF on error			
int	<b>putchar(int c);</b>	<stdio.h>	p. 152
Returns: <i>c</i> if OK, EOF on error			

int	<b>putchar_unlocked</b> (int <i>c</i> ); <stdio.h> Returns: <i>c</i> if OK, EOF on error	p. 444
int	<b>putc_unlocked</b> (int <i>c</i> , FILE * <i>fp</i> ); <stdio.h> Returns: <i>c</i> if OK, EOF on error	p. 444
int	<b>putenv</b> (char * <i>str</i> ); <stdlib.h> Returns: 0 if OK, nonzero on error	p. 212
int	<b>puts</b> (const char * <i>str</i> ); <stdio.h> Returns: non-negative value if OK, EOF on error	p. 153
ssize_t	<b>pwrite</b> (int <i>fd</i> , const void * <i>buf</i> , size_t <i>nbytes</i> , off_t <i>offset</i> ); <unistd.h> Returns: number of bytes written if OK, -1 on error	p. 78
int	<b>raise</b> (int <i>signo</i> ); <signal.h> Returns: 0 if OK, nonzero on error	p. 337
ssize_t	<b>read</b> (int <i>fd</i> , void * <i>buf</i> , size_t <i>nbytes</i> ); <unistd.h> Returns: number of bytes read if OK, 0 if end of file, -1 on error	p. 71
struct dirent	<b>*readdir</b> (DIR * <i>dp</i> ); <dirent.h> Returns: pointer if OK, NULL at end of directory or error	p. 130
ssize_t	<b>readlink</b> (const char *restrict <i>path</i> , char *restrict <i>buf</i> , size_t <i>bufsize</i> ); <unistd.h> Returns: number of bytes read if OK, -1 on error	p. 123
ssize_t	<b>readlinkat</b> (int <i>fd</i> , const char* restrict <i>path</i> , char *restrict <i>buf</i> , size_t <i>bufsize</i> ); <unistd.h> Returns: number of bytes read if OK, -1 on error	p. 123
ssize_t	<b>readv</b> (int <i>fd</i> , const struct iovec * <i>iov</i> , int <i>iovcnt</i> ); <sys/uio.h> Returns: number of bytes read if OK, 0 if end of file, -1 on error	p. 521

---

```

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	<b>renameat</b> (int <i>oldfd</i> , const char * <i>oldname</i> , int <i>newfd</i> , const char * <i>newname</i> ); <stdio.h> Returns: 0 if OK, -1 on error	p. 119
void	<b>rewind</b> (FILE * <i>fp</i> ); <stdio.h>	p. 158
void	<b>rewinddir</b> (DIR * <i>dp</i> ); <dirent.h>	p. 130
int	<b>rmdir</b> (const char * <i>path</i> ); <unistd.h> Returns: 0 if OK, -1 on error	p. 130
int	<b>scanf</b> (const char *restrict <i>format</i> , ...); <stdio.h> Returns: number of input items assigned, EOF if input error or end of file before any conversion	p. 162
void	<b>seekdir</b> (DIR * <i>dp</i> , long <i>loc</i> ); <dirent.h>	p. 130
int	<b>select</b> (int <i>maxfdp1</i> , fd_set *restrict <i>readfds</i> , fd_set *restrict <i>writefd</i> s, fd_set *restrict <i>exceptfd</i> s, struct timeval *restrict <i>tvptr</i> ); <sys/select.h> Returns: count of ready descriptors, 0 on timeout, -1 on error	p. 502
int	<b>sem_close</b> (sem_t * <i>sem</i> ); <semaphore.h> Returns: 0 if OK, -1 on error	p. 580
int	<b>semctl</b> (int <i>semid</i> , int <i>semnum</i> , int <i>cmd</i> , ... /* union semun arg */ ); <sys/sem.h> <i>cmd</i> : IPC_STAT, IPC_SET, IPC_RMID, GETPID, GETNCNT, GETZCNT, GETVAL, SETVAL, GETALL, SETALL Returns: (depends on command), -1 on error	p. 567
int	<b>sem_destroy</b> (sem_t * <i>sem</i> ); <semaphore.h> Returns: 0 if OK, -1 on error	p. 582

int	<b>semget(key_t key, int nsems, int flag);</b>		
	<sys/sem.h>		p. 567
	flag: IPC_CREAT, IPC_EXCL		
	Returns: semaphore ID if OK, -1 on error		
int	<b>sem_getvalue(sem_t *restrict sem, int *restrict valp);</b>		
	<semaphore.h>		p. 582
	Returns: 0 if OK, -1 on error		
int	<b>sem_init(sem_t *sem, int pshared, unsigned int value);</b>		
	<semaphore.h>		p. 582
	Returns: 0 if OK, -1 on error		
int	<b>semop(int semid, struct sembuf semoparray[], size_t nops);</b>		
	<sys/sem.h>		p. 568
	Returns: 0 if OK, -1 on error		
sem_t	<b>*sem_open(const char *name, int oflag, ... /* mode_t mode,</b>		
	<b>unsigned int value */ );</b>		
	<semaphore.h>		p. 579
	flag: IPC_CREAT, IPC_EXCL		
	Returns: pointer to semaphore if OK, SEM_FAILED on error		
int	<b>sem_post(sem_t *sem);</b>		
	<semaphore.h>		p. 582
	Returns: 0 if OK, -1 on error		
int	<b>sem_timedwait(sem_t *restrict sem,</b>		
	<b>const struct timespec *restrict tspr);</b>		
	<semaphore.h>		p. 581
	<time.h>		
	Returns: 0 if OK, -1 on error		
int	<b>sem_trywait(sem_t *sem);</b>		
	<semaphore.h>		
	Returns: 0 if OK, -1 on error		
int	<b>sem_unlink(const char *name);</b>		
	<semaphore.h>		p. 580
	Returns: 0 if OK, -1 on error		
int	<b>sem_wait(sem_t *sem);</b>		
	<semaphore.h>		p. 581
	Returns: 0 if OK, -1 on error		

```

ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags); p. 610
    <sys/socket.h>
    flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL
           MSG_CONFIRM (Linux 3.2.0),
           MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,
                           Mac OS X 10.6.8, Solaris 10),
           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                           Mac OS X 10.6.8, Solaris 10),
           MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
           MSG_MORE (Linux 3.2.0)
    Returns: number of bytes sent if OK, -1 on error

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags); p. 611
    <sys/socket.h>
    flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL
           MSG_CONFIRM (Linux 3.2.0),
           MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,
                           Mac OS X 10.6.8, Solaris 10),
           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                           Mac OS X 10.6.8, Solaris 10),
           MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
           MSG_MORE (Linux 3.2.0)
    Returns: number of bytes sent if OK, -1 on error

ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags, p. 610
    const struct sockaddr *destaddr, socklen_t destlen);
    <sys/socket.h>
    flags: MSG_EOR, MSG_OOB, MSG_NOSIGNAL
           MSG_CONFIRM (Linux 3.2.0),
           MSG_DONTROUTE (FreeBSD 8.0, Linux 3.2.0,
                           Mac OS X 10.6.8, Solaris 10),
           MSG_DONTWAIT (FreeBSD 8.0, Linux 3.2.0,
                           Mac OS X 10.6.8, Solaris 10),
           MSG_EOF (FreeBSD 8.0, Mac OS X 10.6.8),
           MSG_MORE (Linux 3.2.0)
    Returns: number of bytes sent if OK, -1 on error

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

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

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

```

---

int	<b>seteuid(uid_t uid);</b>	<unistd.h>	p. 258
		Returns: 0 if OK, -1 on error	
int	<b>setgid(gid_t gid);</b>	<unistd.h>	p. 256
		Returns: 0 if OK, -1 on error	
void	<b>setgrent(void);</b>	<grp.h>	p. 183
int	<b>setgroups(int ngroups, const gid_t grouplist[]);</b>	<grp.h> /* Linux */ <unistd.h> /* FreeBSD, Mac OS X, and Solaris */	p. 184
		Returns: 0 if OK, -1 on error	
void	<b>sethostent(int stayopen);</b>	<netdb.h>	p. 597
int	<b>setjmp(jmp_buf env);</b>	<setjmp.h>	p. 215
		Returns: 0 if called directly, nonzero if returning from a call to longjmp	
int	<b>setlogmask(int maskpri);</b>	<syslog.h>	p. 470
		Returns: previous log priority mask value	
void	<b>setnetent(int stayopen);</b>	<netdb.h>	p. 598
int	<b>setpgid(pid_t pid, pid_t pgid);</b>	<unistd.h>	p. 294
		Returns: 0 if OK, -1 on error	
int	<b>setpriority(int which, id_t who, int value);</b>	<sys/resource.h>	p. 277
		which: PRIO_PROCESS, PRIO_PGRP, PRIO_USER	
		Returns: 0 if OK, -1 on error	
void	<b>setprotoent(int stayopen);</b>	<netdb.h>	p. 598
void	<b>setpwent(void);</b>	<pwd.h>	p. 180

int	<b>setregid(gid_t rgid, gid_t egid);</b> <unistd.h> Returns: 0 if OK, -1 on error	p. 257
int	<b>setreuid(uid_t ruid, uid_t euid);</b> <unistd.h> Returns: 0 if OK, -1 on error	p. 257
int	<b>setrlimit(int resource, const struct rlimit *rlptr);</b> <sys/resource.h> 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	p. 220
void	<b>setservent(int stayopen);</b> <netdb.h>	p. 599
pid_t	<b>setsid(void);</b> <unistd.h> Returns: process group ID if OK, -1 on error	p. 295
int	<b>setsockopt(int sockfd, int level, int option, const void *val, socklen_t len);</b> <sys/socket.h> Returns: 0 if OK, -1 on error	p. 624
void	<b>setspent(void);</b> <shadow.h> Platforms: Linux 3.2.0, Solaris 10	p. 182
int	<b>setuid(uid_t uid);</b> <unistd.h> Returns: 0 if OK, -1 on error	p. 256

int	<b>setvbuf(FILE *restrict fp, char *restrict buf, int mode, size_t size);</b>	<stdio.h>	p. 146
		mode: _IOFBF, _IOLBF, _IONBF	
		Returns: 0 if OK, nonzero on error	
void	<b>*shmat(int shmid, const void *addr, int flag);</b>	<sys/shm.h>	p. 574
		flag: SHM_RND, SHM_RDONLY	
		Returns: pointer to shared memory segment if OK, -1 on error	
int	<b>shmctl(int shmid, int cmd, struct shmid_ds *buf);</b>	<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	<b>shmrdt(const void *addr);</b>	<sys/shm.h>	p. 574
		Returns: 0 if OK, -1 on error	
int	<b>shmget(key_t key, size_t size, int flag);</b>	<sys/shm.h>	p. 572
		flag: IPC_CREAT, IPC_EXCL	
		Returns: non-negative shared memory ID if OK, -1 on error	
int	<b>shutdown(int sockfd, int how);</b>	<sys/socket.h>	p. 592
		how: SHUT_RD, SHUT_WR, SHUT_RDWR	
		Returns: 0 if OK, -1 on error	
int	<b>sig2str(int signo, char *str);</b>	<signal.h>	p. 380
		Returns: 0 if OK, -1 on error	
		Platforms: Solaris 10	
int	<b>sigaction(int signo, const struct sigaction *restrict act, struct sigaction *restrict oact);</b>	<signal.h>	p. 350
		Returns: 0 if OK, -1 on error	
int	<b>sigaddset(sigset_t *set, int signo);</b>	<signal.h>	p. 344
		Returns: 0 if OK, -1 on error	

int	<b>sigdelset</b> (sigset_t *set, int signo); <signal.h> Returns: 0 if OK, -1 on error	p. 344
int	<b>sigemptyset</b> (sigset_t *set); <signal.h> Returns: 0 if OK, -1 on error	p. 344
int	<b>sigfillset</b> (sigset_t *set); <signal.h> Returns: 0 if OK, -1 on error	p. 344
int	<b>sigismember</b> (const sigset_t *set, int signo); <signal.h> Returns: 1 if true, 0 if false, -1 on error	p. 344
void	<b>siglongjmp</b> (sigjmp_buf env, int val); <setjmp.h> This function never returns	p. 356
void	(* <b>signal</b> (int signo, void (*func)(int)))(int); <signal.h> Returns: previous disposition of signal if OK, SIG_ERR on error	p. 323
int	<b>sigpending</b> (sigset_t *set); <signal.h> Returns: 0 if OK, -1 on error	p. 347
int	<b>sigprocmask</b> (int how, const sigset_t *restrict set, sigset_t *restrict oset); <signal.h> how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK Returns: 0 if OK, -1 on error	p. 346
int	<b>sigqueue</b> (pid_t pid, int signo, const union sigval value) <signal.h> Returns: 0 if OK, -1 on error	p. 376
int	<b>sigsetjmp</b> (sigjmp_buf env, int savemask); <setjmp.h> Returns: 0 if called directly, nonzero if returning from a call to <b>siglongjmp</b>	p. 356

int	<b>sigsuspend</b> (const sigset_t *sigmask); <signal.h> Returns: -1 with errno set to EINTR	p. 359
int	<b>sigwait</b> (const sigset_t *restrict set, int *restrict signop); <signal.h> Returns: 0 if OK, error number on failure	p. 454
unsigned int	<b>sleep</b> (unsigned int seconds); <unistd.h> Returns: 0 or number of unslept seconds	p. 373
int	<b>snprintf</b> (char *restrict buf, size_t n, const char *restrict format, ...); <stdio.h> Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error	p. 159
int	<b>sockatmark</b> (int sockfd); <sys/socket.h> Returns: 1 if at mark, 0 if not at mark, -1 on error	p. 626
int	<b>socket</b> (int domain, int type, int protocol); <sys/socket.h> type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET Returns: file (socket) descriptor if OK, -1 on error	p. 590
int	<b>socketpair</b> (int domain, int type, int protocol, int sockfd[2]); <sys/socket.h> type: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET Returns: 0 if OK, -1 on error	p. 630
int	<b>sprintf</b> (char *restrict buf, const char *restrict format, ...); <stdio.h> Returns: number of characters stored in array if OK, negative value if encoding error	p. 159
int	<b>sscanf</b> (const char *restrict buf, const char *restrict format, ...); <stdio.h> Returns: number of input items assigned, EOF if input error or end of file before any conversion	p. 162

int	<code>stat(const char *restrict path, struct stat *restrict buf);</code>	p. 93
	<sys/stat.h>	
	Returns: 0 if OK, -1 on error	
int	<code>str2sig(const char *str, int *signop);</code>	p. 380
	<signal.h>	
	Returns: 0 if OK, -1 on error	
	Platforms: Solaris 10	
char	<code>*strerror(int errnum);</code>	p. 15
	<string.h>	
	Returns: pointer to message string	
size_t	<code>strftime(char *restrict buf, size_t maxsize,</code>	
	<code>const char *restrict format,</code>	
	<code>const struct tm *restrict tmpr);</code>	
	<time.h>	p. 192
	Returns: number of characters stored in array if room,	
	0 otherwise	
size_t	<code>strftime_l(char *restrict buf, size_t maxsize,</code>	
	<code>const char *restrict format,</code>	
	<code>const struct tm *restrict tmpr, locale_t locale);</code>	
	<time.h>	p. 192
	Returns: number of characters stored in array if room,	
	0 otherwise	
char	<code>*strptime(const char *restrict buf, const char *restrict format,</code>	
	<code>struct tm *restrict tmpr);</code>	
	<time.h>	p. 195
	Returns: pointer to one character past last character parsed,	
	NULL otherwise	
char	<code>*strsignal(int signo);</code>	p. 380
	<string.h>	
	Returns: a pointer to a string describing the signal	
int	<code>symlink(const char *actualpath, const char *sympath);</code>	
	<unistd.h>	
	Returns: 0 if OK, -1 on error	
int	<code>symlinkat(const char *actualpath, int fd, const char *sympath);</code>	
	<unistd.h>	
	Returns: 0 if OK, -1 on error	
void	<code>sync(void);</code>	p. 81
	<unistd.h>	

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

p. 42

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

p. 693

Returns: 0 if OK, -1 on error

```
int       tcflush(int fd, int queue);
          <termios.h>
          queue: TCIFLUSH, TCOFLUSH, TCIOFLUSH
```

p. 693

Returns: 0 if OK, -1 on error

int	<b>tcgetattr</b> (int <i>fd</i> , struct termios * <i>termpptr</i> ); <termios.h> Returns: 0 if OK, -1 on error	p. 683
pid_t	<b>tcgetpgrp</b> (int <i>fd</i> ); <unistd.h> Returns: process group ID of foreground process group if OK, -1 on error	p. 298
pid_t	<b>tcgetsid</b> (int <i>fd</i> ); <termios.h> Returns: session leader's process group ID if OK, -1 on error	p. 299
int	<b>tcsendbreak</b> (int <i>fd</i> , int <i>duration</i> ); <termios.h> Returns: 0 if OK, -1 on error	p. 693
int	<b>tcsetattr</b> (int <i>fd</i> , int <i>opt</i> , const struct termios * <i>termpptr</i> ); <termios.h> <i>opt</i> : TCSANOW, TCSADRAIN, TCSAFLUSH Returns: 0 if OK, -1 on error	p. 683
int	<b>tcsetpgrp</b> (int <i>fd</i> , pid_t <i>pgrp_id</i> ); <unistd.h> Returns: 0 if OK, -1 on error	p. 298
long	<b>telldir</b> (DIR * <i>dp</i> ); <dirent.h> Returns: current location in directory associated with <i>dp</i>	p. 130
time_t	<b>time</b> (time_t * <i>calptr</i> ); <time.h> Returns: value of time if OK, -1 on error	p. 189
clock_t	<b>times</b> (struct tms * <i>buf</i> ); <sys/times.h> Returns: elapsed wall clock time in clock ticks if OK, -1 on error	p. 280
FILE	<b>*tmpfile</b> (void); <stdio.h> Returns: file pointer if OK, NULL on error	p. 167
char	<b>*tmpnam</b> (char * <i>ptr</i> ); <stdio.h> Returns: pointer to unique pathname, NULL on error	p. 167

---

int	<b>truncate</b> (const char *path, off_t length); <unistd.h> Returns: 0 if OK, -1 on error	p. 112
char	<b>*ttyname</b> (int fd); <unistd.h> Returns: pointer to pathname of terminal, NULL on error	p. 695
mode_t	<b>umask</b> (mode_t cmask); <sys/stat.h> Returns: previous file mode creation mask	p. 104
int	<b>uname</b> (struct utsname *name); <sys/utsname.h> Returns: non-negative value if OK, -1 on error	p. 187
int	<b>ungetc</b> (int c, FILE *fp); <stdio.h> Returns: c if OK, EOF on error	p. 151
int	<b>unlink</b> (const char *path); <unistd.h> Returns: 0 if OK, -1 on error	p. 117
int	<b>unlinkat</b> (int fd, const char *path, int flag); <unistd.h> flag: AT_REMOVEDIR Returns: 0 if OK, -1 on error	p. 117
int	<b>unlockpt</b> (int fd); <stdlib.h> Returns: 0 on success, -1 on error	p. 723
int	<b>unsetenv</b> (const char *name); <stdlib.h> Returns: 0 if OK, -1 on error	p. 212
int	<b>utimensat</b> (int fd, const char *path, const struct timespec times[2], int flag); <sys/stat.h> flag: AT_SYMLINK_NOFOLLOW Returns: 0 if OK, -1 on error	p. 126
int	<b>utimes</b> (const char *path, const struct timeval times[2]); <sys/time.h> Returns: 0 if OK, -1 on error	p. 127

```
int      vdprintf(int fd, const char *restrict format, va_list arg);          p. 161
        <stdarg.h>
        <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);                      p. 161
        <stdarg.h>
        <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);                     p. 163
        <stdarg.h>
        <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);                   p. 161
        <stdarg.h>
        <stdio.h>
        Returns: number of characters output if OK, negative
                value if output error

int      vscanf(const char *restrict format, va_list arg);                   p. 163
        <stdarg.h>
        <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);                    p. 161
        <stdarg.h>
        <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);                      p. 161
        <stdarg.h>
        <stdio.h>
        Returns: number of characters stored in array if OK, negative
                value if encoding error
```

<pre>int      vsscanf(const char *restrict buf, const char *restrict format,                  va_list arg);          &lt;stdarg.h&gt;          &lt;stdio.h&gt;</pre> <p>Returns: number of input items assigned, EOF if input error or end of file before any conversion</p>	p. 163
<pre>void     vsyslog(int priority, const char *format, va_list arg);          &lt;syslog.h&gt;          &lt;stdarg.h&gt;</pre> <p>Platforms: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10</p>	p. 472
<pre>pid_t    wait(int *statloc);</pre> <p>&lt;sys/wait.h&gt;</p> <p>Returns: process ID if OK, 0, or -1 on error</p>	p. 238
<pre>int      waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);          &lt;sys/wait.h&gt;</pre> <p><i>idtype:</i> P_PID, P_PGID, P_ALL <i>options:</i> WCONTINUED, WEXITED, WNOHANG, WNOWAIT, WSTOPPED</p> <p>Returns: 0 if OK, -1 on error</p> <p>Platforms: Linux 3.2.0, Solaris 10</p>	p. 244
<pre>pid_t    waitpid(pid_t pid, int *statloc, int options);          &lt;sys/wait.h&gt;</pre> <p><i>options:</i> WCONTINUED, WNOHANG, WUNTRACED</p> <p>Returns: process ID if OK, 0, or -1 on error</p>	p. 238
<pre>pid_t    wait3(int *statloc, int options, struct rusage *rusage);          &lt;sys/types.h&gt;          &lt;sys/wait.h&gt;          &lt;sys/time.h&gt;          &lt;sys/resource.h&gt;</pre> <p><i>options:</i> WNOHANG, WUNTRACED</p> <p>Returns: process ID if OK, 0, or -1 on error</p> <p>Platforms: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10</p>	p. 245
<pre>pid_t    wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);          &lt;sys/types.h&gt;          &lt;sys/wait.h&gt;          &lt;sys/time.h&gt;          &lt;sys/resource.h&gt;</pre> <p><i>options:</i> WNOHANG, WUNTRACED</p> <p>Returns: process ID if OK, 0, or -1 on error</p> <p>Platforms: FreeBSD 8.0, Linux 3.2.0, Mac OS X 10.6.8, Solaris 10</p>	p. 245

```
ssize_t    write(int fd, const void *buf, size_t nbytes);
           <unistd.h>
           Returns: number of bytes written if OK, -1 on error
```

p. 72

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

p. 521

*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      pts_open(char *);                            /* Figure 19.9 */
#ifndef 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 <code>strerror</code> ?	Parameter to <code>strerror</code>	Terminate?
<code>err_dump</code>	yes	<code>errno</code>	<code>abort();</code>
<code>err_exit</code>	yes	explicit parameter	<code>exit(1);</code>
<code>err_msg</code>	no		<code>return;</code>
<code>err_quit</code>	no		<code>exit(1);</code>
<code>err_ret</code>	yes	<code>errno</code>	<code>return;</code>
<code>err_sys</code>	yes	<code>errno</code>	<code>exit(1);</code>
<code>err_cont</code>	yes	explicit parameter	<code>return;</code>
<code>log_msg</code>	no		<code>return;</code>
<code>log_quit</code>	no		<code>exit(2);</code>
<code>log_ret</code>	yes	<code>errno</code>	<code>return;</code>
<code>log_sys</code>	yes	<code>errno</code>	<code>exit(2);</code>
<code>log_exit</code>	yes	explicit parameter	<code>exit(2);</code>

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 arguments */

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 explicit 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 explicit 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));
    strncat(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));
    strncat(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_DEFINED
typedef __size_t size_t;
#define __SIZE_T_DEFINED
#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 opens 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.

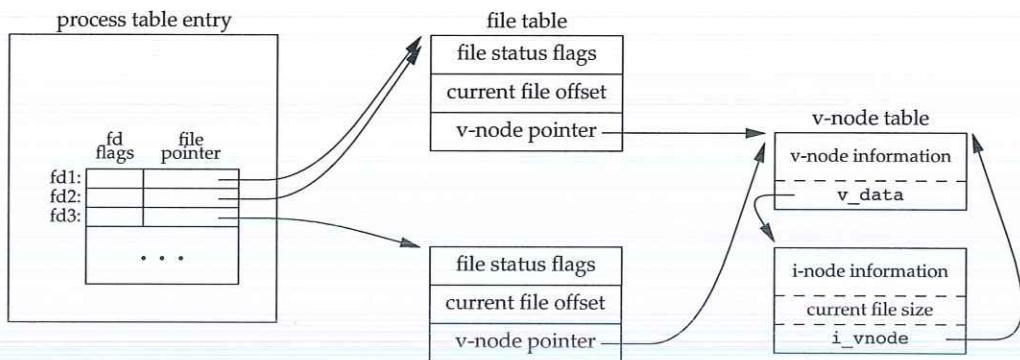


Figure C.2 Result of dup and open

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
$ ls -l foo
--w-r--r-- 1 sar 29 Feb 5 14:21 foo
$ cat foo
cat: foo: Permission denied
```

*turn off user-read permission  
verify the file's permissions  
and try to read it*

- 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
$ date > foo
$ date > bar
$ chmod a-r foo bar
$ ls -l foo bar
--w----- 1 sar 29 Feb 5 14:25 bar
--w----- 1 sar 29 Feb 5 14:25 foo
$ ./a.out
$ ls -l foo bar
--w----- 1 sar 0 Feb 5 14:26 bar
--w----- 1 sar 0 Feb 5 14:26 foo
```

*delete the files in case they already exist  
create them with some data  
turn off all read permissions  
verify their permissions  
run program from Figure 4.9  
check permissions and sizes*

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
...
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->vsizer = ms->curpos;
    } else if (ms->flags & MS_TRUNCATE) { /* "w" mode */
        ms->vsizer = 0;
    } else { /* "r" mode */
        if (ms->buf)
            free(ms->buf);
        ms->buf = NULL;
        ms->curpos = 0;
    }
}
```

```
        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_MBUF)
        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	<i>author's default is US/Eastern</i>
Wed Jul 25 22:58:32 EDT 2012	
\$ TZ=US/Mountain ./a.out	<i>U.S. Mountain time zone</i>
Wed Jul 25 20:58:32 MDT 2012	
\$ TZ=Japan ./a.out	<i>Japan</i>
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;

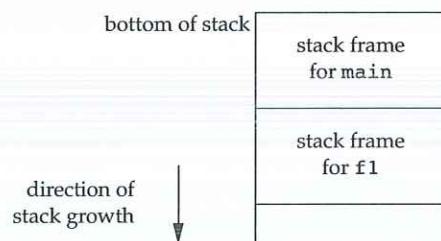
    for (i = 0; i < sizeof(buf); i++)
        buf[i] = 0;
}
```

---

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 `execvp` ends up calling `execve` with the same *pathname* as when we call `execv` directly. Recall Figure 8.15.

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

```
#include "apue.h"

#ifndef 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.

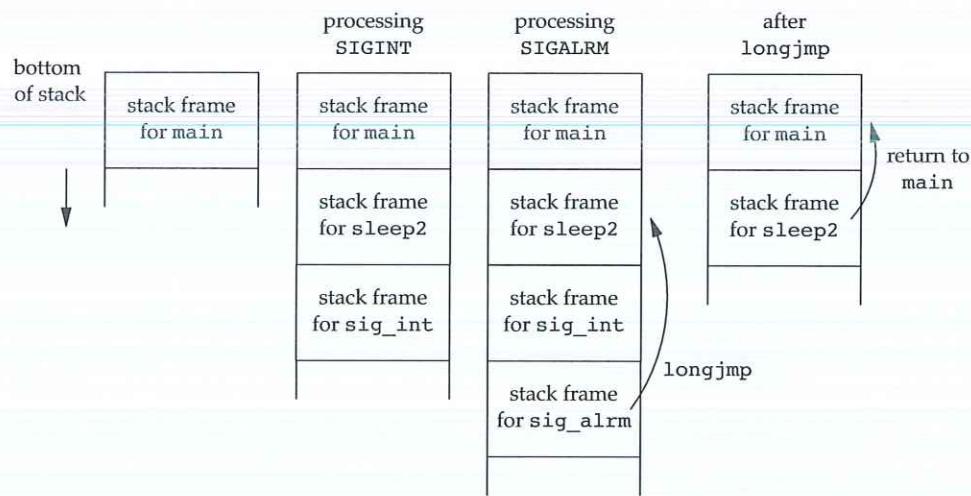


Figure C.11 Stack frames before and after `longjmp`

The `longjmp` from `sig_alarm` 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_alarm` 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.

<code>FD_ZERO</code>	<code>sigemptyset</code>
<code>FD_SET</code>	<code>sigaddset</code>
<code>FD_CLR</code>	<code>sigdelset</code>
<code>FD_ISSET</code>	<code>sigismember</code>

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
<code>select</code> on read end of pipe with write end closed	R/W/E	R	R/W	R/W/E
<code>poll</code> on read end of pipe with write end closed	R/HUP	HUP	INV	HUP
<code>select</code> on write end of pipe with read end closed	R/W/E	R/W	R/W	R/W
<code>poll</code> 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 forks 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	1	0		initialized by mmap child runs first, then is blocked parent runs
		1		then parent is blocked child resumes
		2	0	then child is blocked parent resumes
	3	3	1	then parent is blocked child resumes
		4	2	then child is blocked parent resumes
		5	3	then parent is blocked child 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, `telnetsd` 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 (`ttynname`) 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 Technologies (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_f1` 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.

- #!, *see* interpreter file
  - ., *see* current directory
  - ., ., *see* parent directory
- 386BSD, xxxi, 34
- 3BSD, 234
- 4.1BSD, 525
- 4.2BSD, 18, 34, 81, 121, 129–130, 183, 277, 326, 329, 469, 502, 508, 521, 525, 589
- 4.3BSD, xxxi, 33–34, 36, 49, 201, 257, 267, 289, 313, 318, 329, 366, 482, 535, 735, 898, 951
  - Reno, xxxi, 34, 76
  - Tahoe, xxxi, 34, 951
- 4.4BSD, xxvi, xxxi, 21, 34, 74, 112, 121, 129, 149, 234, 329, 535, 589, 735, 744, 951
- a2ps program, 842
- abort function, 198, 236, 241, 272, 275, 313, 317–319, 331, 365–367, 381, 447, 900
  - definition of, 365–366
- absolute pathname, 5, 8, 43, 50, 64, 136, 141–142, 260, 553, 911
- accept function, 148, 331, 451, 608–609, 615, 617, 635, 639–640, 648, 817
  - definition of, 608
- access function, 102–104, 121, 124, 331, 452
  - definition of, 102
- Accetta, M., 35
- accounting
  - login, 186–187
  - process, 269–275
- acct function, 269
- acct structure, 270, 273
- acctcom program, 269
- accton program, 269, 274
- ACORE constant, 271, 273–274
- Adams, J., 293
- add\_job function, 814, 820, 823, 827
  - definition of, 820
- add\_option function, 831, 834
  - definition of, 831
- addressing, socket, 593–605
- addrinfo structure, 599–603, 614, 616, 618, 620, 622, 800, 802, 804, 807, 813–814, 816, 819, 833

- a**
- `add_worker` function, 814, 824, 828
    - definition of, 828
  - adjustment on exit, semaphore, 570–571
  - Adobe Systems, 825, 947
  - advisory record locking, 495
  - AES (Application Environment Specification), 32
  - AEXPND constant, 271
  - `AF_INET` constant, 590–591, 595–596, 598, 601, 603–604, 802, 808
  - `AF_INET6` constant, 590, 595–596, 601
  - `AF_IPX` constant, 590
  - `AF_LOCAL` constant, 590
  - `AFORK` constant, 270–271, 273
  - `AF_UNIX` constant, 590, 601, 630, 632, 635, 637, 640–641, 941
  - `AF_UNSPEC` constant, 590, 601
  - `agetty` program, 290
  - Aho, A. V., 262, 947
  - `AI_ALL` constant, 603
  - `AI_CANONNAME` constant, 603, 616, 618, 623, 802
  - `AI_NUMERICHOST` constant, 603
  - `AI_NUMERICSERV` constant, 603
  - `aio_cancel` function, 514–515
    - definition of, 514
  - `aiocb` structure, 511, 517–518
  - `aio_error` function, 331, 513, 515, 519–520
    - definition of, 513
  - `aio_fsync` function, 512–513, 520
    - definition of, 513
  - `<aio.h>` header, 29
  - `AIOLISTIO_MAX` constant, 515–516
  - `AIOMAX` constant, 515–516
  - `AIOPRIO_DELTA_MAX` constant, 515–516
  - `aio_read` function, 512–513, 515, 518
    - definition of, 512
  - `aio_return` function, 331, 513, 519–520
    - definition of, 513
  - `aio_suspend` function, 331, 451, 514, 520
    - definition of, 514
  - `aio_write` function, 512–513, 515, 519
    - definition of, 512
  - `AI_PASSIVE` constant, 603
  - `AI_V4MAPPED` constant, 600, 603
  - AIX, 35, 334
  - `alarm` function, 313, 317, 331–332, 335, 338–343, 357, 373–374, 381–382, 620–621, 924
    - definition of, 338
  - `alloca` function, 210
  - Almquist, K., 4
  - `already_running` function, 475–478
    - definition of, 474
  - `ALTWERASE` constant, 676, 682, 685
  - American National Standards Institute, *see* ANSI
  - Andrade, J. M., 560, 947
  - ANSI (American National Standards Institute), 25
  - ANSI C, xxx–xxxii
  - Apple Computer, xxi, xxvi
  - Application Environment Specification, *see* AES
  - `apue_db.h` header, 745, 753, 757, 761
  - `apue.h` header, 7, 9–10, 247, 324, 489–490, 635, 755, 895–898
  - Architecture, UNIX, 1–2
  - `argc` variable, 815
  - `ARG_MAX` constant, 40, 43, 47, 49, 251
  - arguments, command-line, 203
  - `argv` variable, 663
  - Arnold J. Q., 206, 947
  - `<arpa/inet.h>` header, 29, 594
  - `asctime` function, 192
  - `<assert.h>` header, 27
  - assignment-allocation character, 162
  - `ASU` constant, 271, 273
  - asynchronous I/O, 501, 509–520
  - asynchronous socket I/O, 627
  - async-signal safe, 330, 446, 450, 457, 461–462, 927
  - `at` program, 259, 472
  - `atd` program, 259, 465
  - `AT_EACCESS` constant, 103
  - `atexit` function, 40–41, 43, 200, 202, 226, 236, 394, 731, 920
    - definition of, 200
  - `ATEXIT_MAX` constant, 40–41, 43, 49, 52
  - `AT_FDCWD` constant, 65, 94, 102, 106, 110, 116–117, 120, 123–124, 127, 129, 553
  - `atoi` function, 766, 839–840
  - `atol` function, 765–767, 818, 823
  - atomic operation, 39, 44, 59, 63, 77–79, 81, 116, 149, 359, 365, 488, 553, 566, 568, 570, 945
  - `AT_REMOVEDIR` constant, 117
  - `AT_SYMLINK_FOLLOW` constant, 116
  - `AT_SYMLINK_NOFOLLOW` constant, 94, 106, 110, 127
  - AT&T, xix, 6, 33, 174, 336, 507, 948
  - automatic variables, 205, 215, 217, 219, 226
  - avoidance, deadlock, 402–407
  - `awk` program, 44, 46, 262–264, 552, 950
  - `AXSIG` constant, 271, 273–274
  
  - `B0` constant, 692
  - `B110` constant, 692
  - `B115200` constant, 692
  - `B1200` constant, 692
  - `B134` constant, 692
  - `B150` constant, 692

**B1800** constant, 692  
**B19200** constant, 692  
**B200** constant, 692  
**B2400** constant, 692  
**B300** constant, 692  
**B38400** constant, 692  
**B4800** constant, 692  
**B50** constant, 692  
**B57600** constant, 692  
**B600** constant, 692  
**B75** constant, 692  
**B9600** constant, 692  
Bach, M. J., xix, xxxii, 74, 81, 112, 116, 229, 907, 948  
background process group, 296, 300, 302, 304, 306–307, 309, 321, 369, 377, 944  
backoff, exponential, 606  
Barkley, R. E., 949  
barrier attributes, 441–442  
barriers, 418–422  
basename function, 442  
bash program, 85, 372  
**.bash\_login** file, 289  
**.bash\_profile** file, 289  
Bass, J., 485  
baud rate, terminal I/O, 692–693  
Berkeley Software Distribution, *see* BSD  
bibliography, alphabetical, 947–953  
big-endian byte order, 593, 791  
bind function, 331, 604, 609, 624–625, 634–635, 637–638, 641  
    definition of, 604  
/bin/false program, 179  
/bin>true program, 179  
<bits/signum.h> header, 314  
block special file, 95, 138–139  
Bolsky, M. I., 548, 948  
Bostic, K., xxxii, 33, 74, 112, 116, 525, 951  
    Keith, 229, 236  
Bourne, S. R., 3  
Bourne shell, 3, 53, 90, 210, 222, 289, 299, 303, 372, 497, 542, 548, 702, 935, 950  
Bourne-again shell, 3–4, 53, 85, 90, 210, 222, 289, 300, 548  
Bovet, D. P., 74, 948  
BREAK character, 677, 682, 685, 688, 690, 694, 708  
BRKINT constant, 676, 685, 688, 706–708  
BS0 constant, 685  
BS1 constant, 685  
BSD (Berkeley Software Distribution), 34, 65, 111, 175, 286, 289, 291, 293, 296–297, 299, 482, 501, 509–511, 532, 596–597, 630, 726–727, 734, 742  
BSD Networking Release 1.0, xxxi, 34  
BSD Networking Release 2.0, xxxi, 34  
BSD/386, xxxi  
BSDLY constant, 676, 684–685, 689  
\_\_BSD\_VISIBLE constant, 473  
bss segment, 205  
buf\_args function, 656–658, 668–670, 897  
    definition of, 657  
buffer cache, 81  
buffering, standard I/O, 145–147, 231, 235, 265, 367, 552, 721, 752  
BUFSIZ constant, 49, 147, 166, 220  
build\_start function, 814, 817, 822  
    definition of, 822  
BUS\_ADRALN constant, 353  
BUS\_ADRERR constant, 353  
BUS\_OBJERR constant, 353  
byte order, xxii, 593–594, 792, 810, 825, 831, 834, 842, 861, 865  
    big-endian, 593, 791  
    little-endian, 593  
C, ANSI, xxx–xxxii  
    ISO, 25–26, 153, 950  
C shell, 3, 53, 222, 289, 299, 548  
c99 program, 58, 70  
cache  
    buffer, 81  
    page, 81  
CAE (Common Application Environment), 32  
calendar time, 20, 24, 59, 126, 189, 191–192, 264, 270  
calloc function, 207–208, 226, 544, 760, 920  
    definition of, 207  
cancellation point, 451  
canonical mode, terminal I/O, 700–703  
Carges, M. T., 560, 947  
cat program, 89, 112, 123, 301, 304, 734–735, 748, 944  
catclose function, 452  
catgets function, 442, 452  
catopen function, 452  
CBAUDEXT constant, 675, 685  
cbreak terminal mode, 672, 704, 708, 713  
cc program, 6, 57, 206  
CCAR\_OFLOW constant, 675, 685, 689  
cc\_t data type, 674  
CCTS\_OFLOW constant, 675, 685  
cd program, 136  
CDSR\_OFLOW constant, 675, 685  
CDTR\_IFLOW constant, 675, 685  
Cesati, M., 74, 948

**cgetispeed** function, 331, 677, 692  
   definition of, 692  
**cgetospeed** function, 331, 677, 692  
   definition of, 692  
**csetispeed** function, 331, 677, 692  
   definition of, 692  
**csetospeed** function, 331, 677, 692  
   definition of, 692  
 character special file, 95, 138–139, 699  
**CHAR\_BIT** constant, 37–38  
**CHARCLASS\_NAME\_MAX** constant, 39, 49  
**CHAR\_MAX** constant, 37–38  
**CHAR\_MIN** constant, 37–38  
**chdir** function, 8, 121, 135–137, 141, 222, 288,  
   331, 468, 912  
   definition of, 135  
 Chen, D., 949  
**CHILD\_MAX** constant, 40, 43, 49, 233  
**chmod** function, 106–108, 121, 125, 331, 452, 558,  
   641, 944  
   definition of, 106  
**chmod** program, 99–100, 559  
**chown** function, 55, 109–110, 120–121, 125, 288,  
   331, 452, 558, 944  
   definition of, 109  
**chroot** function, 141, 480, 910, 928  
**CIBAUDEXT** constant, 675, 685  
**CIGNORE** constant, 675, 685  
 Clark, J. J., xxxii  
**CLD\_CONTINUED** constant, 353  
**CLD\_DUMPED** constant, 353  
**CLD\_EXITED** constant, 353  
**CLD\_KILLED** constant, 353  
**CLD\_STOPPED** constant, 353  
**CLD\_TRAPPED** constant, 353  
**clearenv** function, 212  
**clearerr** function, 151  
   definition of, 151  
**cli\_args** function, 656–658, 668–669  
   definition of, 658  
**cli\_conn** function, 636–637, 640, 659, 665, 897  
   definition of, 636, 640  
**client\_add** function, 662, 665, 667  
   definition of, 661  
**client\_alloc** function, 661–662, 668  
   definition of, 660  
**client\_cleanup** function, 814, 824, 829  
   definition of, 829  
**client\_del** function, 665, 667  
   definition of, 661  
 client-server model, 479–480, 585–587  
**client\_thread** function, 814, 817, 824  
   definition of, 824  
**CLOCAL** constant, 318, 675, 685  
**clock** function, 58–59  
 clock tick, 20, 42–43, 49, 59, 270, 280  
**clock\_getres** function, 190  
   definition of, 190  
**clock\_gettime** function, 189–190, 331, 408, 414,  
   437, 439  
   definition of, 189  
**clockid\_t** data type, 189  
**CLOCK\_MONOTONIC** constant, 189  
**clock\_nanosleep** function, 373–375, 437, 439,  
   451, 462  
   definition of, 375  
**CLOCK\_PROCESS\_CPUTIME\_ID** constant, 189  
**CLOCK\_REALTIME** constant, 189–190, 408, 437,  
   439, 581  
**clock\_settime** function, 190, 439  
   definition of, 190  
**CLOCKS\_PER\_SEC** constant, 59  
**clock\_t** data type, 20, 58–59, 280  
**CLOCK\_THREAD\_CPUTIME\_ID** constant, 189  
**clone** function, 229  
**close** function, 8, 52, 61, 66, 80–81, 124, 128, 331,  
   451, 468, 474, 492, 532, 537–539, 544, 550, 553,  
   560, 577–578, 587, 592–593, 609, 616, 618,  
   625, 638–639, 641, 654–655, 657, 665,  
   667–669, 725–726, 728–729, 739–740, 761,  
   823, 826–827, 829, 833, 837  
   definition of, 66  
**closedir** function, 5, 7, 130–135, 452, 698, 823,  
   910  
   definition of, 130  
**closeelog** function, 452, 470  
   definition of, 470  
**close-on-exec** flag, 80, 83, 252–253, 479–480, 492  
**clrasync** function, definition of, 940  
**clr\_f1** function, 85, 482–483, 896, 937  
**clri** program, 122  
**cmsgcred** structure, 648–651  
**CMSG\_DATA** function, 645–646, 648, 650, 652  
   definition of, 645  
**CMSG\_FIRSTHDR** function, 645, 652  
   definition of, 645  
**cmsghdr** structure, 645–647, 649, 651  
**CMSG\_LEN** function, 645–647, 649, 651  
   definition of, 645  
**CMSG\_NXTHDR** function, 645, 650, 652  
   definition of, 645  
**CMSPAR** constant, 675, 685, 690  
**codes**, option, 31  
**COLL\_WEIGHTS\_MAX** constant, 39, 43, 49  
**COLUMNS** environment variable, 211  
 Comer, D. E., 744, 949

- command-line arguments, 203  
 Common Application Environment, *see* CAE  
 Common Open Software Environment, *see* COSE  
 communication, network printer, 789–843  
`<complex.h>` header, 27  
`comp_t` data type, 59  
 Computing Science Research Group, *see* CSRG  
 condition variable attributes, 440–441  
 condition variables, 413–416  
`connect` function, 331, 451, 605–608, 610–611,  
     621, 635, 641–642  
     definition of, 605  
 connection establishment, 605–609  
`connect_retry` function, 607, 614, 800, 808, 834  
     definition of, 606–607  
 controlling  
     process, 296–297, 318  
 terminal, 63, 233, 252, 270, 292, 295–298, 301,  
     303–304, 306, 309, 311–312, 318, 321, 377, 463,  
     465–466, 469, 480, 680, 685, 691, 694, 700, 702,  
     716, 724, 726–727, 898, 953  
 cooked terminal mode, 672  
 cooperating processes, 495, 752, 945  
 Coordinated Universal Time, *see* UTC  
 coprocesses, 548–552, 721, 737  
 copy-on-write, 229, 458  
 core dump, 74, 928  
`core` file, 111, 124, 275, 315, 317, 320, 332, 366, 681,  
     703, 909, 920, 922  
 COSE (Common Open Software Environment), 32  
 count, link, 44, 59, 114–117, 130  
`cp` program, 141, 528  
`cpio` program, 127, 142, 910–911  
`<cpio.h>` header, 29  
 CR terminal character, 678, 680, 703  
`CR0` constant, 685  
`CR1` constant, 685  
`CR2` constant, 685  
`CR3` constant, 685  
`CRDLY` constant, 676, 684–685, 689  
`CREAD` constant, 675, 686  
`creat` function, 61, 66, 68, 79, 89, 101, 104, 118,  
     121, 125, 149, 331, 451, 491, 825–826, 909, 912  
     definition of, 66  
 creation mask, file mode, 104–105, 129, 141, 169,  
     233, 252, 466  
`cron` program, 259, 382, 465, 470, 472–474, 925  
`CRTSCTS` constant, 675, 686  
`CRTS_IFLOW` constant, 675, 686  
`CRTSXOFF` constant, 675, 686  
`crypt` function, 287, 298, 304, 442  
`crypt` program, 298, 700  
`CS5` constant, 684, 686  
`CS6` constant, 684, 686  
`CS7` constant, 684, 686  
`CS8` constant, 684, 686, 706–708  
`.cshrc` file, 289  
`CSIZE` constant, 675, 684, 686, 706–707  
`csopen` function, 653–654  
     definition of, 654, 659  
 CSRG (Computing Science Research Group), xx,  
     xxvi, 34  
`CSTOPB` constant, 675, 686  
`ctermid` function, 442, 452, 694, 700–701  
     definition of, 694  
`ctime` function, 192  
`<ctype.h>` header, 27  
`cu` program, 500  
`cupsd` program, 465, 793  
 current directory, 4–5, 8, 13, 43, 50, 65, 94, 100,  
     115–117, 120, 127, 130, 135–137, 178, 211, 233,  
     252, 315, 317, 466  
 Curses, 32  
`curses` library, 712–713, 949, 953  
`cuserid` function, 276  
  
 daemon, 463–480  
     coding, 466–469  
     conventions, 474–479  
     error logging, 469–473  
`daemonize` function, 466, 468, 480, 616, 618, 623,  
     664, 815, 896, 929–930  
     definition of, 467  
 Dang, X. T., 206, 949  
 Darwin, xxii, xxvii, 35  
`dash` program, 372  
 data, out-of-band, 626  
 data segment  
     initialized, 205  
     uninitialized, 205  
 data transfer, 610–623  
 data types, primitive system, 58  
 database library, 743–787  
     coarse-grained locking, 752  
     concurrency, 752–753  
     fine-grained locking, 752  
     implementation, 746–750  
     performance, 781–786  
     source code, 753–781  
 database transactions, 952  
 Date, C. J., 753, 949  
 date functions, time and, 189–196  
`date` program, 192, 196, 371, 919, 944

- DATEMSK environment variable, 211  
**db** library, 744, 952  
**DB** structure, 756–758, 760–762, 765–768, 773, 776, 782  
  `_db_alloc` function, 757, 760–761  
    definition of, 760  
  `_db_close` function, 745, 749, 754, 761  
    definition of, 745, 761  
  `_db_delete` function, 746, 752, 754, 768–769, 771, 945  
    definition of, 746, 768  
  `_db_dodelete` function, 757, 768–769, 772, 776, 780–781, 787, 944–945  
    definition of, 769  
  `_db_fetch` function, 745, 748–749, 752, 754, 762, 767  
    definition of, 745, 762  
  `_db_find_and_lock` function, 757, 762–763, 767–768, 774–775, 777, 786  
    definition of, 763  
  `_db_findfree` function, 757, 775, 777–778, 781  
    definition of, 777  
  `_db_free` function, 757–758, 761  
    definition of, 761  
**DBHANDLE** data type, 749, 754, 757, 761–762, 768, 774, 779  
  `_db_hash` function, 757, 764, 787  
    definition of, 764  
**DB\_INSERT** constant, 745, 749, 754, 774, 776  
**dbm** library, 743–744, 952  
**dbm\_clearerr** function, 442  
**dbm\_close** function, 442, 452  
**dbm\_delete** function, 442, 452  
**dbm\_error** function, 442  
**dbm\_fetch** function, 442, 452  
**dbm\_firstkey** function, 442  
**dbm\_nextkey** function, 442, 452  
**dbm\_open** function, 442, 452  
**dbm\_store** function, 442, 452  
**dbm\_nextrec** function, 746, 750, 752, 754, 769, 779, 781, 787, 944–945  
    definition of, 746, 779  
**db\_open** function, 745–746, 749, 752, 754–757, 759–761, 781  
    definition of, 745, 757  
  `_db_readdat` function, 757, 762, 768, 780, 945  
    definition of, 768  
  `_db_readidx` function, 757, 764–765, 778, 780, 945  
    definition of, 765  
  `_db_readptr` function, 757, 763, 765, 770, 775–777, 787  
    definition of, 765  
**DB\_REPLACE** constant, 745, 754, 774  
**db rewind** function, 746, 754, 760, 779, 781  
  definition of, 746, 779  
**DB\_STORE** constant, 745, 754, 774  
**db\_store** function, 745, 747, 749, 752, 754, 769, 771, 774, 781, 787  
  definition of, 745, 774  
  `_db_writedata` function, 757, 769, 771–772, 775–777, 781, 787, 944–945  
    definition of, 771  
  `_db_writeidx` function, 522, 757, 759, 770, 772, 775–776, 781, 787, 945  
    definition of, 772  
  `_db_writeptr` function, 757, 759, 770, 773, 775–776, 778  
    definition of, 773  
**dcheck** program, 122  
**dd** program, 275  
**deadlock**, 234, 402, 490, 552, 721  
  avoidance, 402–407  
  record locking, 490  
**Debian Almquist shell**, 4, 53  
**Debian Linux distribution**, 4  
**delayed write**, 81  
**DELAYTIMER\_MAX** constant, 40, 43  
**descriptor set**, 503, 505, 532, 933  
**detachstate** attribute, 427–428  
**/dev/fd** device, 88–89, 142, 696  
**/dev/fd/0** device, 89  
**/dev/fd/1** device, 89, 142  
**/dev/fd/2** device, 89  
**device number**  
  major, 58–59, 137, 139, 465, 699  
  minor, 58–59, 137, 139, 465, 699  
**device special file**, 137–139  
**/dev/klog** device, 470  
**/dev/kmem** device, 68  
**/dev/log** device, 470, 480, 928  
**/dev/null** device, 73, 86, 304  
**/dev/stderr** device, 89, 697  
**/dev/stdin** device, 89, 697  
**/dev/stdout** device, 89, 697  
**dev\_t** data type, 59, 137–138  
**devtmpfs** file system, 139  
**/dev/tty** device, 298, 304, 312, 694, 700, 740  
**/dev/tty1** file, 290  
**/dev/zero** device, 576–578  
**df** program, 141, 910  
**DIR** structure, 7, 131, 283, 697, 822  
**directories**  
  files and, 4–8  
  hard links and, 117, 120  
  reading, 130–135

directory, 4  
 current, 4–5, 8, 13, 43, 50, 65, 94, 100, 115–117,  
 120, 127, 130, 135–137, 178, 211, 233, 252, 315,  
 317, 466  
 file, 95  
 home, 2, 8, 135, 211, 288, 292  
 ownership, 101–102  
 parent, 4, 108, 125, 129  
 root, 4, 8, 24, 139, 141, 233, 252, 283, 910  
 Directory Services daemon, 185  
 dirent structure, 5, 7, 131, 133, 697, 822  
 <dirent.h> header, 7, 29, 131  
 dirname function, 442  
 DISCARD terminal character, 678, 680, 687  
 dlclose function, 452  
 dlerror function, 442  
 <dlfcn.h> header, 29  
 dlopen function, 452  
 do\_driver function, 732, 739  
 definition of, 739  
 Dorward, S., 229, 952  
 DOS, 57, 65  
 dot, *see* current directory  
 dot-dot, *see* parent directory  
 dprintf function, 159, 452, 945  
 definition of, 159  
 drand48 function, 442  
 DSUSP terminal character, 678, 680, 688  
 dtruss program, 497  
 du program, 111, 141, 909  
 Duff, T., 88  
 dup function, 52, 61, 74, 77, 79–81, 148, 164, 231,  
 331, 468, 492–493, 592–593, 907–908, 921  
 definition of, 79  
 dup2 function, 64, 79–81, 90, 148, 331, 539, 544,  
 550–551, 592, 618–619, 655, 728–729,  
 739–740, 907–908  
 definition of, 79

E2BIG error, 564  
 EACCES error, 14–15, 474, 487, 499, 918  
 EAGAIN error, 16, 376, 474, 482, 484, 487, 496–497,  
 499, 514, 563, 569–570, 581, 609, 627  
 EBADF error, 52, 916  
 EBUSY error, 16, 400, 410, 418  
 ECANCELED error, 515  
 ECHILD error, 333, 351, 371, 546  
 ECHO constant, 676, 686–687, 701, 705–707, 731  
 echo program, 203  
 ECHOCTL constant, 676, 686  
 ECHOE constant, 676, 686–687, 701, 731  
 ECHOK constant, 676, 687, 701, 731  
 ECHOKE constant, 676, 687  
 ECHONL constant, 676, 687, 701, 731  
 ECHOPRT constant, 676, 686–687  
 ed program, 367, 369–370, 496–497  
 EDEADLK error, 418  
 EEXIST error, 121, 558, 584  
 EFBIG error, 925  
 effective  
 group ID, 98–99, 101–102, 108, 110, 140, 183,  
 228, 233, 256, 258, 558, 587  
 user ID, 98–99, 101–102, 106, 110, 126, 140, 228,  
 233, 253, 256–260, 276, 286, 288, 337, 381, 558,  
 562, 568, 573, 586–587, 637, 640, 809, 918  
 efficiency  
 I/O, 72–74  
 standard I/O, 153–156  
 EIDRM error, 562–564, 568–570, 579  
 EINPROGRESS error, 519–520, 608  
 EINTR error, 16, 265–266, 301, 327–329, 339, 359,  
 370, 502, 508, 514, 545–546, 563–564,  
 569–570, 620  
 EINVAL error, 42, 47–48, 345, 389, 543, 545–546,  
 705–707, 774, 914  
 EIO error, 309, 321, 823–824, 826–827  
 Ellis, M., xxxii  
 ELOOP error, 121–122  
 EMFILE error, 544, 546  
 EMSGSIZE error, 610  
 ENAMETOOLONG error, 65, 637, 640  
 encrypt function, 442  
 endgrent function, 183–184, 442, 452  
 definition of, 183  
 endhostent function, 452, 597  
 definition of, 597  
 endnetent function, 452, 598  
 definition of, 598  
 endprotoent function, 452, 598  
 definition of, 598  
 endpwent function, 180–181, 442, 452  
 definition of, 180  
 endservent function, 452, 599  
 definition of, 599  
 endspent function, 182  
 definition of, 182  
 endutxent function, 442, 452  
 ENFILE error, 16  
 ENOBUFS error, 16  
 ENOENT error, 15, 170, 445, 745, 774  
 ENOLCK error, 16  
 ENOMEM error, 16, 914  
 ENOMSG error, 564  
 ENOSPC error, 16, 445  
 ENOTDIR error, 592

**E**NOTRECOVERABLE error, 433  
**E**NOTTY error, 683, 693  
**e**nviron variable, 203–204, 211, 213, 251, 255,  
 444–445, 450, 920  
 environment list, 203–204, 233, 251, 286–288  
 environment variable, 210–213  
     **COLUMNS**, 211  
     **DATEMSK**, 211  
     **HOME**, 210–211, 288  
     **IFS**, 269  
     **LANG**, 41, 211  
     **LC\_ALL**, 211  
     **LC\_COLLATE**, 43, 211  
     **LC\_CTYPE**, 211  
     **LC\_MESSAGES**, 211  
     **LC\_MONETARY**, 211  
     **LC\_NUMERIC**, 211  
     **LC\_TIME**, 211  
     **LD\_LIBRARY\_PATH**, 753  
     **LINES**, 211  
     **LOGNAME**, 211, 276, 288  
     **MAILPATH**, 210  
     **MALLOC\_OPTIONS**, 928  
     **MSGVERB**, 211  
     **NLSPATH**, 211  
     **PAGER**, 539, 542–543  
     **PATH**, 100, 211, 250–251, 253, 260, 263, 265,  
 288–289  
     **POSIXLY\_CORRECT**, 111  
     **PWD**, 211  
     **SHELL**, 211, 288, 737  
     **TERM**, 211, 287, 289  
     **TMPDIR**, 211  
     **TZ**, 190, 192, 195–196, 211, 919  
     **USER**, 210, 288  
**E**NXIO error, 553  
**E**OF constant, 10, 151–152, 154, 164, 175, 545,  
 547–548, 550–551, 664, 730, 913  
**E**OF terminal character, 678, 680, 686–687, 700, 703  
**E**OL terminal character, 678, 680, 687, 700, 703  
**E**OL2 terminal character, 678, 680, 687, 700, 703  
**E**OWNERDEAD error, 432  
**E**PERM error, 256  
**E**Pipe error, 537, 937  
**E**poch, 20, 22, 126, 187, 189–190, 640  
**E**RANGE error, 50  
**E**RASE terminal character, 678, 680, 686–687,  
 702–703  
**E**RASE2 terminal character, 678, 681  
**err**\_cont function, 897, 899  
     definition of, 900  
**err**\_dump function, 366, 767, 897, 899  
     definition of, 900  
**err**\_exit function, 809, 897, 899  
     definition of, 900  
**err**\_msg function, 897, 899  
     definition of, 901  
**errno** variable, 14–15, 42, 50, 55, 65, 67, 81,  
 120–121, 144, 256, 265, 277, 301, 309, 314, 321,  
 327–328, 330–331, 333, 337, 339, 345, 351,  
 359, 371, 376, 380, 384, 386, 446–447, 454, 471,  
 474, 482, 484, 487, 499, 502, 508, 513–514, 537,  
 546, 553, 564, 568, 579, 581, 584, 592, 608–610,  
 627, 637–640, 683, 693, 745, 805, 899, 925, 937  
**<errno.h>** header, 14–16, 27  
**error**  
     handling, 14–16  
     logging, daemon, 469–473  
     recovery, 16  
     routines, standard, 898–904  
     TOCTTOU, 65, 250, 953  
**err**\_quit function, 7, 279, 815, 897, 899, 912  
     definition of, 901  
**err**\_ret function, 897, 899, 912  
     definition of, 899  
**err**\_sys function, 7, 279, 897, 899  
     definition of, 899  
**E**PIPE error, 67, 592  
**E**SRCH error, 337  
     /etc/gettydefs file, 290  
     /etc/group file, 17–18, 177, 185–186  
     /etc/hosts file, 186, 795  
     /etc/init directory, 290  
     /etc/inittab file, 290  
     /etc/master.passwd file, 185  
     /etc/networks file, 185–186  
     /etc/passwd file, 2, 99, 135, 177–178, 180, 182,  
 185–186  
     /etc/printer.conf file, 794–795, 799  
     /etc/protocols file, 185–186  
     /etc/pwd.db file, 185  
     /etc/rc file, 189, 291  
     /etc/services file, 185–186  
     /etc/shadow file, 99, 185–186  
     /etc/spwd.db file, 185  
     /etc/syslog.conf file, 470  
     /etc/termcap file, 712  
     /etc/ttys file, 286  
**E**TIME error, 800, 805  
**E**TIMEDOUT error, 407, 413, 415, 581, 800  
**E**vans, J., 949  
**E**WOULDBLOCK error, 16, 482, 609, 627  
**E**XDEV error, 120  
**exec** function, 10–11, 13, 23, 39–40, 43, 79, 82,  
 100, 121, 125, 197, 201, 203, 225, 229, 233–234,  
 249–257, 260–261, 264–266, 269–271, 275,  
 277, 282–283, 286–288, 290–292, 294, 305,

**325, 372, 457, 479, 492, 527, 533, 538, 541, 557, 585, 653–654, 658–659, 669, 716–717, 721, 723, 727, 739, 742, 920, 928, 949**  
**exec1 function, 249–251, 261, 265–266, 272, 274–275, 283, 288, 331, 370–371, 539, 544, 550–551, 618, 655, 737, 922**  
**definition of, 249**  
**execle function, 249–251, 254, 287, 331**  
**definition of, 249**  
**execlp function, 12–13, 19, 249–251, 253–254, 264–265, 283, 740, 922**  
**definition of, 249**  
**execv function, 249–251, 331**  
**definition of, 249**  
**execve function, 249–251, 253, 331, 922**  
**definition of, 249**  
**execvp function, 249–251, 253, 731–732**  
**definition of, 249**  
**exercises, solutions to, 905–945**  
**\_Exit function, 198, 201, 236–237, 239, 331, 365, 367, 388, 447**  
**definition of, 198**  
**\_exit function, 198, 201, 235–239, 265–266, 282–283, 331, 365, 367, 370, 381, 388, 447, 921, 924**  
**definition of, 198**  
**exit function, 7, 150, 154, 198–202, 226, 231, 234–239, 246, 249, 265, 271–272, 274–275, 283, 288, 330, 365–366, 388, 447, 466, 542, 705, 732, 742, 817, 830, 895, 920–921, 944**  
**definition of, 198**  
**exit handler, 200**  
**expect program, 720, 739–740, 951**  
**exponential backoff, 606**  
**ext2 file system, 129**  
**ext3 file system, 129**  
**ext4 file system, 73, 86, 129, 465**  
**EXTPROC constant, 676, 687**  
  
**faccessat function, 102–104, 331, 452**  
**definition of, 102**  
**Fagin, R., 744, 750, 949**  
**Fast-STREAMS, Linux, 534**  
**fatal error, 16**  
**fchkdir function, 135–137, 592**  
**definition of, 135**  
**fchmod function, 106–108, 120, 125, 331, 452, 498, 592**  
**definition of, 106**  
**fchmodat function, 106–108, 331, 452**  
**definition of, 106**  
  
**fchown function, 109–110, 125, 331, 452, 592**  
**definition of, 109**  
**fchownat function, 109–110, 331, 452**  
**definition of, 109**  
**fclose function, 148–150, 172–174, 199, 201, 365, 367, 452, 545, 701, 803**  
**definition of, 150**  
**fcntl function, 61, 77, 80–87, 90, 112, 148, 164, 252–253, 331, 451–452, 480, 482, 485–490, 492, 494–495, 510–511, 592, 626–627, 783, 785, 939, 944**  
**definition of, 82**  
**<fcntl.h> header, 29, 62**  
**fd\_dasync function, 81, 86–87, 331, 451, 513, 592**  
**definition of, 81**  
**FD\_CLOEXEC constant, 63, 79, 82–83, 252, 480**  
**FD\_CLR function, 503–504, 665, 933**  
**definition of, 503**  
**FD\_ISSET function, 503–504, 665, 817, 933**  
**definition of, 503**  
**fdopen function, 148–150, 159, 544, 936**  
**definition of, 148**  
**fdopendir function, 130–135**  
**definition of, 130**  
**fd-pipe, 653–654, 656, 658**  
**fd\_pipe function, 630, 655, 739, 896**  
**definition of, 630**  
**fd\_set data type, 59, 503–504, 532, 664, 805, 814, 816–817, 932–933, 939**  
**FD\_SET function, 503–504, 664–665, 805, 816, 933**  
**definition of, 503**  
**FD\_SETSIZE constant, 933**  
**FD\_SETSIZE constant, 504, 932–933**  
**F\_DUPFD constant, 81–83, 592**  
**F\_DUPFD\_CLOEXEC constant, 82, 592**  
**FD\_ZERO function, 503–504, 664, 805, 933**  
**definition of, 503**  
**feature test macro, 57–58, 84**  
**Fenner, B., 157, 291, 470, 589, 953**  
**<fenv.h> header, 27**  
**feof function, 151, 157**  
**definition of, 151**  
**ferror function, 10, 151, 154, 157, 273, 538, 543, 550**  
**definition of, 151**  
**fexecve function, 249–250, 253, 331**  
**definition of, 249**  
**FF0 constant, 687**  
**FF1 constant, 687**  
**FFDLY constant, 676, 684, 687, 689**  
**fflush function, 145, 147, 149, 172, 174–175, 366, 452, 547–548, 552, 702, 721, 901, 904, 913**  
**definition of, 147**

**F\_FREESP** constant, 112  
**fgetc** function, 150–151, 154–155, 452  
  definition of, 150  
**F\_GETFD** constant, 82–83, 480, 592  
**F\_GETFL** constant, 82–85, 592  
**F\_GETLK** constant, 82, 486–490  
**F\_GETOWN** constant, 82–83, 592, 626  
**fgetpos** function, 157–159, 452  
  definition of, 158  
**fgets** function, 10, 12, 19, 150, 152–156, 168,  
  174–175, 214, 216, 452, 538, 543, 548,  
  550–552, 616, 622, 654, 738, 753, 803, 845, 911,  
  913, 936  
  definition of, 152  
**fgetwc** function, 452  
**fgetws** function, 452  
**FIFOs**, 95, 534, 552–556  
**file**  
  access permissions, 99–101, 140  
  block special, 95, 138–139  
  character special, 95, 138–139, 699  
  descriptor passing, 587, 642–652  
  descriptors, 8–10, 61–62  
  device special, 137–139  
  directory, 95  
  group, 182–183  
  holes, 68–69, 111–112  
  mode creation mask, 104–105, 129, 141, 169,  
  233, 252, 466  
  offset, 66–68, 74, 77–78, 80, 231–232, 494, 522,  
  747–748, 908  
  ownership, 101–102  
  pointer, 144  
  regular, 95  
  sharing, 74–77, 231  
  size, 111–112  
  times, 124–125, 532  
  truncation, 112  
  types, 95–98  
**FILE** structure, 131, 143–144, 151, 164, 168,  
  171–172, 220, 235, 273, 443–444, 538,  
  542–543, 545, 547, 622, 701, 754, 803, 914, 929  
**file system**, 4, 113–116  
  `devtmpfs`, 139  
  `ext2`, 129  
  `ext3`, 129  
  `ext4`, 73, 86, 129, 465  
  `HFS`, 87, 113, 116  
  `HSFS`, 113  
  `PCFS`, 49, 57, 113  
  `S5`, 65  
  `UFS`, 49, 57, 65, 113, 116, 129  
**filename**, 4  
  truncation, 65–66  
**FILENAME\_MAX** constant, 38  
**fileno** function, 164, 545, 701, 913  
  definition of, 164  
**\_FILE\_OFFSET\_BITS** constant, 70  
**FILEPERM** constant, 800, 825  
**files and directories**, 4–8  
**FILESIZEBITS** constant, 39, 44, 49  
**find** program, 124, 135, 252  
**finger** program, 141, 179, 910  
**FIOASYNC** constant, 627, 939–940  
**FIOSETOWN** constant, 627  
**FIPS**, 32–33  
**Flandrena, B.**, 229, 952  
**<float.h>** header, 27, 38  
**flock** function, 485  
**flock** structure, 486, 489–490, 494  
**flockfile** function, 443–444  
  definition of, 443  
**FLUSHO** constant, 676, 680, 687  
**fmemopen** function, 171–175, 913  
  definition of, 171  
**fmtmsg** function, 211, 452  
**<fmtmsg.h>** header, 30  
**FNDELAY** constant, 482  
**<fnmatch.h>** header, 29  
**F\_OK** constant, 102  
**follow\_link** function, 48  
**fopen** function, 6, 144, 148–150, 165, 220, 273,  
  452, 538–539, 542, 701, 803, 929  
  definition of, 148  
**FOPEN\_MAX** constant, 38, 43  
**foreground process group**, 296, 298, 300–303, 306,  
  311, 318–322, 369, 377, 680–682, 685, 689, 710,  
  719, 741, 944  
**foreground process group ID**, 298, 303, 677  
**fork** function, 11–13, 19, 23, 77, 228–237,  
  241–243, 245–249, 254, 260–261, 264–266,  
  269–272, 274–275, 277–278, 282, 286, 288,  
  290–292, 294, 296, 304, 307–308, 312, 326,  
  331, 334, 370–372, 381, 457–462, 466–469,  
  471, 479, 491–493, 498–500, 527, 533–539,  
  541, 544, 546, 550, 557, 565, 577, 585, 588,  
  618–619, 642, 653–655, 658–659, 669–670,  
  716, 721, 723–724, 726–728, 732, 739, 781,  
  922–923, 927–928, 930–931, 934, 937, 939, 949  
  definition of, 229  
**fork1** function, 229  
**forkall** function, 229  
**Fowler, G. S.**, 135, 949, 953  
**fpathconf** function, 37, 39, 41–48, 53–55, 65,  
  110, 452, 537, 679  
  definition of, 42

**FPE\_FLTDIV** constant, 353  
**FPE\_FLTINV** constant, 353  
**FPE\_FLTOVF** constant, 353  
**FPE\_FLTRES** constant, 353  
**FPE\_FLTSUB** constant, 353  
**FPE\_FLTUND** constant, 353  
**FPE\_INTDIV** constant, 353  
**FPE\_INTOVF** constant, 353  
**fpos\_t** data type, 59, 157  
**fprintf** function, 159, 452  
     definition of, 159  
**fputc** function, 145, 152, 154–155, 452  
     definition of, 152  
**fputs** function, 146, 150, 152–156, 164, 168,  
     174–175, 452, 543, 548, 550, 701, 901, 904, 911,  
     919, 926, 936  
     definition of, 153  
**fputwc** function, 452  
**fputws** function, 452  
**F\_RDLCK** constant, 486–487, 489–490, 897,  
     930–931  
**fread** function, 150, 156–157, 269, 273, 452  
     definition of, 156  
**free** function, 163, 174, 207–209, 330, 332, 401,  
     403–405, 407, 437–438, 450, 697, 762, 829,  
     833, 837, 842, 917  
     definition of, 207  
**freeaddrinfo** function, 599, 833  
     definition of, 599  
**FreeBSD**, xxi–xxii, xxvi–xxvii, 3–4, 21, 26–27,  
     29–30, 34–36, 38, 49, 57, 60, 62, 64, 68, 70, 81,  
     83, 88, 95, 102, 108–111, 121, 129, 132, 138,  
     175, 178, 182, 184–185, 187–188, 209–212,  
     222, 225, 229, 240, 245, 253, 257, 260, 262, 269,  
     271, 276–277, 288–289, 292, 298, 303, 310,  
     314–316, 319, 322, 329, 334, 351, 355, 358, 371,  
     373, 377, 379–380, 385, 388, 393, 396, 409,  
     426–427, 433, 439, 473, 485, 492–493, 497,  
     499, 503, 527, 534, 539, 561, 567, 572, 576,  
     594–595, 607, 611–613, 627, 634, 648–649,  
     652, 675–678, 685–691, 716, 724, 726–727,  
     740–741, 744, 799, 911, 918, 930, 932–933,  
     935–936, 949, 951  
**freopen** function, 144, 148–150, 452  
     definition of, 148  
**frequency scaling**, 785  
**fscanf** function, 162, 452  
     definition of, 162  
**fsck** program, 122  
**fseek** function, 149, 157–159, 172, 452  
     definition of, 158  
**fseeko** function, 157–159, 172, 452  
     definition of, 158  
**F\_SETFD** constant, 82, 85, 90, 480, 592, 907  
**F\_SETFL** constant, 82–83, 85, 90, 511, 592, 627,  
     907, 944  
**F\_SETLK** constant, 82, 486–488, 490, 494, 897,  
     930–931  
**F\_SETLKW** constant, 82, 486, 488, 490, 897, 931  
**F\_SETOWN** constant, 82–83, 510, 592, 626–627, 939  
**fsetpos** function, 149, 157–159, 172, 452  
     definition of, 158  
**fstat** function, 4, 93–95, 120, 331, 452, 494, 498,  
     518, 529–530, 535, 586, 592, 698, 759, 808, 833  
     definition of, 93  
**fstatat** function, 93–95, 331, 452  
     definition of, 93  
**fsync** function, 61, 81, 86–87, 175, 331, 451, 513,  
     517, 528, 592, 787, 913  
     definition of, 81  
**ftell** function, 157–159, 452  
     definition of, 158  
**ftello** function, 157–159, 452  
     definition of, 158  
**ftok** function, 557–558  
     definition of, 557  
**ftp** program, 472, 928  
**ftruncate** function, 112, 125, 331, 529–530, 592  
     definition of, 112  
**ftrylockfile** function, 443–444  
     definition of, 443  
**fts** function, 132  
**ftw** function, 122, 130–135, 141  
<ftw.h> header, 30  
**full-duplex pipes**, 534  
     named, 534  
     timing, 565  
**function prototypes**, 845–893  
**functions, system calls versus**, 21–23  
**F\_UNLCK** constant, 486–487, 489–490, 897  
**funlockfile** function, 443–444  
     definition of, 443  
**funopen** function, 175, 915  
**futimens** function, 125–128, 331, 452, 910  
     definition of, 126  
**fwide** function, 144  
     definition of, 144  
**fwprintf** function, 452  
**fwrite** function, 150, 156–157, 382, 452, 925  
     definition of, 156  
**F\_WRLCK** constant, 486–487, 489–490, 494, 897,  
     931  
**fwscanf** function, 452

- gai\_strerror** function, 600, 616, 619, 621, 623  
     definition of, 600
- Gallmeister, B. O.**, 949
- Garfinkel, S.**, 181, 250, 298, 949
- gather write**, 521, 644
- gawk** program, 262
- gcc** program, 6, 26, 58, 919
- gdb** program, 928
- gdbm library**, 744
- generic pointer**, 71, 208
- getaddrinfo** function, 452, 599–601, 603–604,  
     614–616, 619, 621, 623, 802, 808  
     definition of, 599
- getaddrlist** function, 800, 802, 804, 808, 815  
     definition of, 802
- GETALL** constant, 568
- getc** function, 10, 150–156, 164–165, 452,  
     701–702, 913  
     definition of, 150
- getchar** function, 150–151, 164, 175, 452, 547, 913  
     definition of, 150
- getchar\_unlocked** function, 442, 444, 452  
     definition of, 444
- getconf** program, 70
- getc\_unlocked** function, 442, 444, 452  
     definition of, 444
- getcwd** function, 50, 135–137, 142, 208, 452,  
     911–912  
     definition of, 136
- getdate** function, 211, 442, 452
- getdelim** function, 452
- getegid** function, 228, 331  
     definition of, 228
- getenv** function, 204, 210–212, 442, 444–446,  
     449–450, 462, 539, 928  
     definition of, 210
- getenv\_r** function, 445–446
- geteuid** function, 228, 257, 268, 331, 650, 809  
     definition of, 228
- getgid** function, 17, 228, 331  
     definition of, 228
- getgrent** function, 183–184, 442, 452  
     definition of, 183
- getgrgid** function, 182, 442, 452  
     definition of, 182
- getgrgid\_r** function, 443, 452
- getgrnam** function, 182, 442, 452  
     definition of, 182
- getgrnam\_r** function, 443, 452
- getgroups** function, 184, 331  
     definition of, 184
- gethostbyaddr** function, 597, 599
- gethostbyname** function, 597, 599
- gethostent** function, 442, 452, 597  
     definition of, 597
- gethostid** function, 452
- gethostname** function, 39–40, 43, 188, 452,  
     616–618, 623, 815  
     definition of, 188
- getline** function, 452
- getlogin** function, 275–276, 442, 452, 480,  
     929–930  
     definition of, 275
- getlogin\_r** function, 443, 452
- getmsg** function, 740
- getnameinfo** function, 452, 600  
     definition of, 600
- GETNCNT** constant, 568
- getnetbyaddr** function, 442, 452, 598  
     definition of, 598
- getnetbyname** function, 442, 452, 598  
     definition of, 598
- getnetent** function, 442, 452, 598  
     definition of, 598
- get\_newjobno** function, 814, 820, 825, 843  
     definition of, 820
- getopt** function, 442, 452, 662–664, 669, 730–731,  
     807–808  
     definition of, 662
- getpass** function, 287, 298, 700, 702–703  
     definition of, 701
- getpeername** function, 331, 605  
     definition of, 605
- getpgid** function, 293–294  
     definition of, 294
- getpgrp** function, 293, 331  
     definition of, 293
- GETPID** constant, 568
- getpid** function, 11, 228, 230, 235, 272, 308, 331,  
     366, 378, 387, 474, 650, 939  
     definition of, 228
- getppid** function, 228–229, 331, 491, 732  
     definition of, 228
- get\_printaddr** function, 800, 804, 819  
     definition of, 804
- get\_printserver** function, 800, 804, 808  
     definition of, 804
- getpriority** function, 277  
     definition of, 277
- getprotobyname** function, 442, 452, 598  
     definition of, 598
- getprotobynumber** function, 442, 452, 598  
     definition of, 598
- getprotoent** function, 442, 452, 598  
     definition of, 598
- getpwent** function, 180–181, 442, 452

- definition of**, 180  
**getpwnam** function, 177–181, 186, 276, 287,  
 330–332, 442, 452, 816, 918  
 definition of, 179–180  
**getpwnam\_r** function, 443, 452  
**getpwuid** function, 177–181, 186, 275–276, 442,  
 452, 809, 918  
 definition of, 179  
**getpwuid\_r** function, 443, 452  
**getresgid** function, 257  
**getresuid** function, 257  
**getrlimit** function, 53, 220, 224, 466–467,  
 906–907  
 definition of, 220  
**getrusage** function, 245, 280  
**gets** function, 152–153, 911  
 definition of, 152  
**getservbyname** function, 442, 452, 599  
 definition of, 599  
**getservbyport** function, 442, 452, 599  
 definition of, 599  
**getservent** function, 442, 452, 599  
 definition of, 599  
**getsid** function, 296  
 definition of, 296  
**getsockname** function, 331, 605  
 definition of, 605  
**getsockopt** function, 331, 624–625  
 definition of, 624  
**getspent** function, 182  
 definition of, 182  
**getspnam** function, 182, 918  
 definition of, 182  
**gettimeofday** function, 190, 278, 414, 421, 437,  
 439  
 definition of, 190  
**getty** program, 238, 286–288, 290, 472  
**gettytab** file, 287  
**getuid** function, 17, 228, 257, 268, 275–276, 331  
 definition of, 228  
**getutxent** function, 442, 452  
**getutxid** function, 442, 452  
**getutxline** function, 442, 452  
**GETVAL** constant, 568  
**getwo** function, 452  
**getwchar** function, 452  
**GETZCNT** constant, 568  
**Ghemawat**, S., 949  
**GID**, *see* group ID  
**gid\_t** data type, 59  
**Gingell**, R. A., 206, 525, 949  
**Gitlin**, J. E., xxxii  
**glob** function, 452  
**global variables**, 219  
**<glob.h>** header, 29  
**gmtime** function, 191–192, 442  
 definition of, 192  
**gmtime\_r** function, 443  
**GNU**, 2, 289, 753  
**GNU Public License**, 35  
**Godsil**, J. M., xxxii  
**Goodheart**, B., 712, 949  
**Google**, 210  
**goto, nonlocal**, 213–220, 355–358  
**Grandi**, S., xxxii  
**grantpt** function, 723–725  
 definition of, 723  
**grep** program, 20, 174, 200, 252, 950  
**group** file, 182–183  
**group ID**, 17, 255–260  
 effective, 98–99, 101–102, 108, 110, 140, 183,  
 228, 233, 256, 258, 558, 587  
 real, 98, 102, 183, 228, 233, 252–253, 256, 270,  
 585  
 supplementary, 18, 39, 98, 101, 108, 110,  
 183–184, 233, 252, 258  
**group structure**, 182  
**<grp.h>** header, 29, 182, 186  
**guardsize** attribute, 427, 430  
  
**hack**, 303, 842  
**half-duplex pipes**, 534  
**handle\_request** function, 656, 665–666, 668  
 definition of, 657, 668  
**hard link**, 4, 114, 117, 120, 122  
**hard links and directories**, 117, 120  
**hcreate** function, 442  
**hdestroy** function, 442  
**headers**  
 optional, 30  
 POSIX required, 29  
 standard, 27  
 XSI option, 30  
**heap**, 205  
**Hein**, T. R., xxxii, 952  
**Hewlett-Packard**, 35, 835  
**HFS** file system, 87, 113, 116  
**Hogue**, J. E., xxxii  
**holes**, file, 68–69, 111–112  
**home directory**, 2, 8, 135, 211, 288, 292  
**HOME** environment variable, 210–211, 288  
**Honeyman**, P., xxxii  
**hostent** structure, 597  
**hostname** program, 189

**HOST\_NAME\_MAX** constant, 40, 43, 49, 188, 615–618, 622–623, 800, 815  
**HP-UX**, 35  
**hsearch** function, 442  
**HSFS** file system, 113  
**htonl** function, 594, 810, 824–827, 834  
  definition of, 594  
**hton** function, 594, 831, 834  
  definition of, 594  
**HTTP** (Hypertext Transfer Protocol), 792–793  
**Hume**, A. G., 174, 950  
**HUPCL** constant, 675, 687  
**Hypertext Transfer Protocol**, *see* HTTP

**IBM** (International Business Machines), 35  
**ICANON** constant, 676, 678, 680–682, 686–687, 691, 703, 705–707  
**iconv\_close** function, 452  
*<iconv.h>* header, 29  
**iconv\_open** function, 452  
**ICRNL** constant, 676, 680, 688, 700, 706–708  
**identifiers**  
  IPC, 556–558  
  process, 227–228  
**IDXLEN\_MAX** constant, 779  
**IEC** (International Electrotechnical Commission), 25  
**IEEE** (Institute for Electrical and Electronic Engineers), xx, 26–27, 950  
**IEXTEN** constant, 676, 678, 680–682, 688, 706–708  
**I\_FIND** constant, 725–726  
**IFS** environment variable, 269  
**IGNBRK** constant, 676, 685, 688  
**IGNCR** constant, 676, 680, 688, 700  
**IGNPAR** constant, 676, 688, 690  
**ILL\_BADSTK** constant, 353  
**ILL\_COPROC** constant, 353  
**ILL\_ILLADR** constant, 353  
**ILL\_ILLOPC** constant, 353  
**ILL\_ILLOPN** constant, 353  
**ILL\_ILLTRP** constant, 353  
**ILL\_PRVOPC** constant, 353  
**ILL\_PRVREG** constant, 353  
**Illumos**, xxi  
**IMAXBEL** constant, 676, 688  
**implementation differences**, password, 184–185  
**implementations**, UNIX System, 33  
**INADDR\_ANY** constant, 605  
**in\_addr\_t** data type, 595  
**incore**, 74, 152  
**INET6\_ADDRSTRLEN** constant, 596  
**inet\_addr** function, 596  
**INET\_ADDRSTRLEN** constant, 596, 603–604  
**inetd** program, 291, 293, 465, 470, 472  
**inet\_ntoa** function, 442, 596  
**inet\_ntop** function, 596, 604  
  definition of, 596  
**inet\_pton** function, 596  
  definition of, 596  
**INFOTIM** constant, 508  
**init** program, 187, 189, 228, 237–238, 246, 270, 286–291, 293, 307, 309, 312, 320, 337, 379, 464–465, 475, 923, 930  
**initgroups** function, 184, 288  
  definition of, 184  
**initialized data segment**, 205  
**init\_printer** function, 814, 816, 819, 833  
  definition of, 819  
**init\_request** function, 814, 816, 818  
  definition of, 818  
**initserver** function, 615–617, 619, 622–623, 800, 816  
  definition of, 609, 625  
**inittab** file, 320  
**INLCR** constant, 676, 688  
**i-node**, 59, 75–77, 94, 108, 113–116, 120, 124, 127, 130–131, 138–139, 179, 253, 698, 905, 910  
**ino\_t** data type, 59, 114  
**INPCK** constant, 676, 688, 690, 706–708  
**in\_port\_t** data type, 595  
**Institute for Electrical and Electronic Engineers**, *see* IEEE  
**int16\_t** data type, 831  
**Intel**, xxii  
**International Business Machines**, *see* IBM  
**International Electrotechnical Commission**, *see* IEC  
**International Standards Organization**, *see* ISO  
**Internet Printing Protocol**, *see* IPP  
**Internet worm**, 153  
**interpreter file**, 260–264, 283  
**interprocess communication**, *see* IPC  
**interrupted system calls**, 327–330, 343, 351, 354–355, 365, 508  
**INT\_MAX** constant, 37–38  
**INT\_MIN** constant, 37–38  
**INTR** terminal character, 678, 681, 688, 701  
*<inttypes.h>* header, 27  
**I/O**  
  asynchronous, 501, 509–520  
  asynchronous socket, 627  
  efficiency, 72–74  
  library, standard, 10, 143–175  
  memory-mapped, 525–531  
  multiplexing, 500–509

- nonblocking, 481–484
- nonblocking socket, 608–609, 627
- terminal, 671–713
- unbuffered, 8, 61–91
- I\_OBUFSZ** constant, 836
- ioctl** function, 61, 87–88, 90, 297–298, 322, 328–329, 452, 482, 510, 562, 592, 627, 674, 710–711, 718–719, 725–728, 730, 740–742, 939–940
  - definition of, 87
  - \_I\_OFBF** constant, 147
  - \_IOLBF** constant, 147, 166, 220
  - \_IO\_LINE\_BUF** constant, 165
  - \_IONBF** constant, 147, 166
  - \_IO\_UNBUFFERED** constant, 165
- iovec** structure, 41, 43, 521, 611, 646–647, 649, 651, 655, 659, 765, 771–772, 832, 836
- IOV\_MAX** constant, 41, 43, 49, 521
- IPC** (interprocess communication), 533–588, 629–670
  - identifiers, 556–558
  - key, 556–558, 562, 567, 572
  - XSI, 556–560
- IPC\_CREAT** constant, 558, 632, 941
- IPC\_EXCL** constant, 558
- IPC\_NOWAIT** constant, 563–564, 569–570
- ipc\_perm** structure, 558, 562, 567, 572, 587
- IPC\_PRIVATE** constant, 557–558, 575, 586, 588
- ipcrm** program, 559
- IPC\_RMID** constant, 562–563, 568, 573–575
- ipcs** program, 559, 588
- IPC\_SET** constant, 562–563, 568, 573
- IPC\_STAT** constant, 562–563, 568, 573
- IPP** (Internet Printing Protocol), 789–792
- ipp.h** header, 843
- ipp\_hdr** structure, 798, 832, 834, 838, 842
- IPPROTO\_ICMP** constant, 591
- IPPROTO\_IP** constant, 591, 624
- IPPROTO\_IPV6** constant, 591
- IPPROTO\_RAW** constant, 591, 602
- IPPROTO\_TCP** constant, 591, 602, 624
- IPPROTO\_UDP** constant, 591, 602
- I\_PUSH** constant, 725–726
- IRIX**, 35
- isalpha** function, 516
- isatty** function, 679, 695, 698–699, 711, 730, 738
  - definition of, 695
- isdigit** function, 839–840
- I\_SETSIG** constant, 510
- ISIG** constant, 676, 678, 680–682, 688, 706–708
- ISO** (International Standards Organization), xx, xxxi, 25–27, 950
- ISO C**, 25–26, 153, 950
- <**iso646.h**> header, 27
- is\_read\_lockable** function, 490, 897
- isspace** function, 839–840
- ISTRIP** constant, 676, 688, 690, 706–708
- is\_write\_lockable** function, 490, 897
- IUCLC** constant, 676, 688
- IUTF8** constant, 676, 689
- IXANY** constant, 676, 689
- IXOFF** constant, 676, 681–682, 689
- IXON** constant, 676, 681–682, 689, 706–708
- jemalloc**, 210
- jmp\_buf** data type, 216, 218, 340, 343
- job control**, 299–303
  - shell, 294, 299, 306–307, 325, 358, 377, 379, 734–735
    - signals, 377–379
  - job structure, 812–813, 820–821, 832
  - job\_append** function, definition of, 411
  - job\_find** function, 927
    - definition of, 412
  - job\_insert** function, definition of, 411
  - job\_remove** function, 927
    - definition of, 412
- Jolitz, W. F., 34
- Joy, W. N., 3, 76
- jsh** program, 299
- Karels, M. J., 33–34, 74, 112, 116, 229, 236, 525, 951
- kernel**, 1
- Kernighan, B. W., xx, xxxii, 26, 149, 155, 162, 164, 208, 262, 898, 906, 947, 950
- Kerrisk, M., 950
- key, IPC, 556–558, 562, 567, 572
- key\_t** data type, 557, 633
- kill** function, 18, 272, 308, 314, 325, 331, 335–338, 353, 363, 366–367, 376, 378, 381, 455, 679, 681, 702, 732–733, 924, 932
  - definition of, 337
- kill** program, 314–315, 321, 325, 551
- KILL** terminal character, 678, 681, 687, 702–703
- kill\_workers** function, 814, 828–830
  - definition of, 828
- Kleiman, S. R., 76, 950
- Knuth, D. E., 422, 764, 950
- Korn, D. G., 3, 135, 174, 548, 948–950, 953
- Korn shell, 3, 53, 90, 210, 222, 289, 299, 497, 548, 702, 733–734, 737, 935, 948
- Kovach, K. R., 560, 947
- Krieger, O., 174, 531, 951

- 164a** function, 442  
**LANG** environment variable, 41, 211  
**<langinfo.h>** header, 29  
**last** program, 187  
**launchctl** program, 293  
**launchd** program, 228, 259, 289, 292, 465  
**layers**, shell, 299  
**LC\_ALL** environment variable, 211  
**LC\_COLLATE** environment variable, 43, 211  
**LC\_CTYPE** environment variable, 211  
**lchown** function, 109–110, 121, 125  
  definition of, 109  
**LC\_MESSAGES** environment variable, 211  
**LC\_MONETARY** environment variable, 211  
**LC\_NUMERIC** environment variable, 211  
**L\_ctermid** constant, 694  
**LC\_TIME** environment variable, 211  
**ld** program, 206  
**LDAP** (Lightweight Directory Access Protocol), 185  
**LD\_LIBRARY\_PATH** environment variable, 753  
**ldterm** STREAMS module, 716, 726  
leakage, memory, 209  
least privilege, 256, 795, 816  
Lee, M., 206, 949  
Lee, T. P., 949  
Leffler, S. J., 34, 951  
Lennert, D., 951  
Lesk, M. E., 143  
**lgamma** function, 442  
**lgammaf** function, 442  
**lgammal** function, 442  
Libes, D., 720, 924, 951  
**<libgen.h>** header, 30  
libraries, shared, 206–207, 226, 753, 920, 947  
Lightweight Directory Access Protocol, *see* LDAP  
**limit** program, 53, 222  
limits, 36–53  
  C, 37–38  
  POSIX, 38–41  
  resource, 220–225, 233, 252, 322, 382  
  runtime indeterminate, 49–53  
  XSI, 41  
**<limits.h>** header, 27, 37, 39, 41, 49–50  
Linderman, J. P., xxxii  
line control, terminal I/O, 693–694  
**LINE\_MAX** constant, 39, 43, 49  
**LINES** environment variable, 211  
link  
  count, 44, 59, 114–117, 130  
  hard, 4, 114, 117, 120, 122  
  symbolic, 55, 94–95, 110–111, 114, 118, 120–123,  
    131, 137, 141, 186, 908–909  
link function, 79, 115–119, 121–122, 125, 331, 452  
  definition of, 116  
**linkat** function, 116–119, 331, 452  
  definition of, 116  
**LINK\_MAX** constant, 39, 44, 49, 114  
**lint** program, 200  
Linux, xxi–xxii, xxv, xxvii, 2–4, 7, 14, 21, 26–27,  
  29–30, 35–38, 49, 52, 57, 60, 62, 64–65, 70, 73,  
  75–76, 86–89, 102, 108–111, 121–122, 129,  
  132, 138, 173, 178, 182, 184–185, 187–188,  
  205, 209, 211–212, 222, 226, 229, 240, 244–245,  
  253, 257, 259–260, 262, 269, 271, 274,  
  276–277, 288–290, 293, 298, 303, 306,  
  314–316, 318–320, 322, 329, 334–335, 351,  
  354–355, 358, 371, 373, 377, 379–380, 385,  
  388, 392, 396, 409, 426–427, 432–433, 439,  
  462, 464–465, 473–474, 485, 496–497, 503,  
  522, 530–531, 534, 559, 561, 567, 571–573,  
  575–576, 578, 583, 594–596, 607, 611–613,  
  627, 634, 648–650, 652, 675–678, 684–691,  
  693, 716, 724, 726–727, 740–741, 744, 753,  
  783, 793, 799, 911, 918, 925, 930, 932, 935–936  
Linux Fast-STREAMS, 534  
LinuxThreads, 388  
**lio\_listio** function, 452, 515  
  definition of, 515  
**LIO\_NOWAIT** constant, 515  
Lions, J., 951  
**LIO\_WAIT** constant, 515  
**listen** function, 331, 605, 608–609, 625, 635, 638,  
  800  
  definition of, 608  
little-endian byte order, 593  
Litwin, W., 744, 750, 951  
**LLONG\_MAX** constant, 37  
**LLONG\_MIN** constant, 37  
**ln** program, 115  
LNEXT terminal character, 678, 681  
locale, 43, 192, 211  
**localeconv** function, 442  
**<locale.h>** header, 27  
**localtime** function, 190–192, 194–195, 264, 408,  
  442, 452, 919  
  definition of, 192  
**localtime\_r** function, 443, 452  
**lockf** function, 451–452, 485  
**lockf** structure, 493  
**lockfile** function, 473–474  
  definition of, 494  
locking  
  database library, coarse-grained, 752  
  database library, fine-grained, 752  
**locking** function, 485

- `lock_reg` function, 489, 897, 930–931  
   definition of, 489
- locks  
   reader-writer, 409–413  
   spin, 417–418
- `lock_test` function, 489–490, 897  
   definition of, 489
- `log` function, 470
- `LOG_ALERT` constant, 472
- `LOG_AUTH` constant, 472
- `LOG_AUTHPRIV` constant, 472
- `LOG_CONS` constant, 468, 471
- `LOG_CRIT` constant, 472
- `LOG_CRON` constant, 472
- `LOG_DAEMON` constant, 468, 472
- `LOG_DEBUG` constant, 472
- `LOG_EMERG` constant, 472
- `LOG_ERR` constant, 472, 474–476, 478–479,  
   615–619, 622–623, 902–903
- `log_exit` function, 817, 898–899  
   definition of, 903
- `LOG_FTP` constant, 472
- `logger` program, 471
- login accounting, 186–187  
   `.login` file, 289
- login name, 2, 17, 135, 179, 187, 211, 275–276, 290,  
   480, 930  
   root, 16
- `login` program, 179, 182, 184, 187, 251, 254, 256,  
   276, 287–290, 292, 472, 700, 717, 738
- `LOG_INFO` constant, 472, 476, 478
- `LOGIN_NAME_MAX` constant, 40, 43, 49
- logins  
   network, 290–293  
   terminal, 285–290
- `LOG_KERN` constant, 472
- `LOG_LOCAL0` constant, 472
- `LOG_LOCAL1` constant, 472
- `LOG_LOCAL2` constant, 472
- `LOG_LOCAL3` constant, 472
- `LOG_LOCAL4` constant, 472
- `LOG_LOCAL5` constant, 472
- `LOG_LOCAL6` constant, 472
- `LOG_LOCAL7` constant, 472
- `LOG_LPR` constant, 472
- `LOG_MAIL` constant, 472
- `log_msg` function, 897, 899  
   definition of, 903
- `LOGNAME` environment variable, 211, 276, 288
- `LOG_NDELAY` constant, 471, 928
- `LOG_NEWS` constant, 472
- `LOG_NOTICE` constant, 472
- `log_open` function, 664, 898  
   definition of, 902
- `LOG_PERROR` constant, 471
- `LOG_PID` constant, 471, 664
- `log_quit` function, 830, 898–899  
   definition of, 903
- `log_ret` function, 898–899  
   definition of, 902
- `log_sys` function, 804, 898–899  
   definition of, 902
- `LOG_SYSLOG` constant, 472
- `log_to_stderr` variable, 664, 807, 813, 902, 904
- `LOG_USER` constant, 472, 664
- `LOG_WARNING` constant, 472
- `LONG_BIT` constant, 38  
   `longjmp` function, 355, 358  
   `longjmp` function, 197, 213, 215–219, 225,  
   330–331, 340–341, 343, 355–358, 365, 381, 924  
   definition of, 215
- `LONG_MAX` constant, 37, 52–53, 60, 420, 906–907
- `LONG_MIN` constant, 37
- `loop` function, 663–664, 666, 670, 732, 742  
   definition of, 666, 732
- `lp` program, 585, 793
- `lpc` program, 472
- `lpd` program, 472, 793
- `lpshed` program, 585, 793
- `lrand48` function, 442
- `ls` program, 5–8, 13, 107–108, 112, 123, 125, 131,  
   135, 139, 141, 177, 179, 559, 905
- `lseek` function, 8, 59, 61, 66–70, 77–79, 88, 91,  
   149, 158, 331, 452, 462, 486, 489, 498, 592, 670,  
   765–766, 768, 771, 773, 779, 819, 908  
   definition of, 67
- `lstat` function, 93–97, 121–122, 133, 141, 331,  
   452, 942  
   definition of, 93
- `L_tmpnam` constant, 168
- Lucchina, P., xxii
- Mac OS X, xxi–xxii, xxvi–xxvii, 3–4, 17, 26–27,  
   29–30, 35–36, 38, 49, 57, 60, 62, 64, 70, 83,  
   87–88, 102, 108–111, 113, 121, 129, 132, 138,  
   175, 178, 182, 184–185, 187–188, 193, 209,  
   211–212, 222, 228, 240, 244–245, 260, 262, 269,  
   271, 276–277, 288–289, 292–293, 298, 303,  
   314–317, 319, 322, 329, 334, 351, 355, 371, 373,  
   377, 379–380, 385, 388, 393, 396, 409,  
   426–427, 464–465, 485, 497, 503, 522, 534,  
   559, 561, 567, 572, 576, 594, 607, 611–613, 627,  
   634, 648, 675–678, 685–691, 716, 724,  
   726–727, 740–741, 744, 793, 799, 911, 918, 925,  
   930, 932, 935–936

Mach, xxii, xxvi–xxvii, 35, 947  
**<machine/\_types.h>** header, 906  
macro, feature test, 57–58, 84  
MAILPATH environment variable, 210  
main function, 7, 150, 155, 197–200, 202, 204,  
  215–217, 226, 236–237, 249, 283, 330–332,  
  357–358, 468, 654, 656, 663, 729, 739, 811, 814,  
  817, 824, 830, 833, 919, 921, 939, 944  
major device number, 58–59, 137, 139, 465, 699  
major function, 138–139  
make program, 300  
makethread function, 436, 438–439  
mallinfo function, 209  
malloc function, 21–23, 51, 136, 145, 174,  
  207–210, 213, 330, 332, 392, 400–401, 403,  
  405, 429, 437, 447, 450, 575, 616, 618, 623,  
  646–647, 650–651, 661–662, 666, 696,  
  760–761, 815, 820, 828, 839, 926, 928  
  definition of, 207  
MALLOC\_OPTIONS environment variable, 928  
mallopt function, 209  
mandatory record locking, 495  
Mandrake, xxvii  
MAP\_ANON constant, 578  
MAP\_ANONYMOUS constant, 578  
MAP\_FAILED constant, 529, 577  
MAP\_FIXED constant, 526–527  
MAP\_PRIVATE constant, 526, 528, 578  
MAP\_SHARED constant, 526–529, 576–578  
**<math.h>** header, 27  
Mauro, J., 74, 112, 116, 951  
MAX\_CANON constant, 39, 44, 47, 49, 673  
MAX\_INPUT constant, 39, 44, 49, 672  
MAXPATHLEN constant, 49  
MB\_LEN\_MAX constant, 37  
mbstate\_t structure, 442  
McDougall, R., 74, 112, 116, 951  
McIlroy, M. D., xxxii  
McKusick, M. K., xxxii, 33–34, 74, 112, 116, 229,  
  236, 525, 951  
MD5, 181  
MDMBUF constant, 675, 685, 689  
memccpy function, 155  
memcpy function, 530–531, 916  
memory  
  allocation, 207–210  
  layout, 204–206  
  leakage, 209  
  shared, 534, 571–578  
memory-mapped I/O, 525–531  
memset function, 172–173, 614, 616, 618, 621, 623  
Menage, P., 949  
message queues, 534, 561–565  
  timing, 565  
**<mgetty>** program, 290  
MIN terminal value, 687, 703–704, 708, 713, 943  
minor device number, 58–59, 137, 139, 465, 699  
minor function, 138–139  
mkdir function, 101–102, 120–122, 125, 129–130,  
  331, 452, 912  
  definition of, 129  
**mkdir** program, 129  
**mkdirat** function, 129–130, 331, 452  
  definition of, 129  
**mkdtemp** function, 167–171, 452  
  definition of, 169  
**mkfifo** function, 120–121, 125, 331, 452, 553, 937  
  definition of, 553  
**mkfifo** program, 553  
**mkfifoat** function, 331, 452, 553  
  definition of, 553  
**mknod** function, 120–121, 129, 331, 452, 553  
**mknodat** function, 331, 452, 553  
**mkstemp** function, 167–171, 452  
  definition of, 169  
**mktimes** function, 190, 192, 195, 452  
  definition of, 192  
**mlock** function, 221  
**mmap** function, 174, 221, 429, 481, 525, 527,  
  529–532, 576–578, 587, 592, 949  
  definition of, 525  
**modem**, xx, xxvii, 285, 287, 297, 318, 328, 481, 508,  
  671, 674–675, 685, 687, 689, 692  
**mode\_t** data type, 59  
**<monetary.h>** header, 29  
Moran, J. P., 525, 949  
**more** program, 543, 748  
Morris, R., 181, 951  
**mount** program, 102, 129, 139, 496  
mounted STREAMS-based pipes, 534  
**mprotect** function, 527  
  definition of, 527  
**mq\_receive** function, 451  
**mq\_send** function, 451  
**mq\_timedreceive** function, 451  
**mq\_timedsend** function, 451  
**<mqueue.h>** header, 30  
**mrand48** function, 442  
**MS\_ASYNC** constant, 528  
**MSG\_CONFIRM** constant, 611  
**msgctl** function, 558–559, 562  
  definition of, 562  
**MSG\_CTRUNC** constant, 613  
**MSG\_DONTROUTE** constant, 611  
**MSG\_DONTWAIT** constant, 611  
**MSG\_EOF** constant, 611

- MSG\_EOR** constant, 611, 613  
**msgget** function, 557–562, 632–633, 941  
     definition of, 562  
**msghdr** structure, 611, 613, 644, 646–647, 649, 651  
**MSG\_MORE** constant, 611  
**MSG\_NOERROR** constant, 564, 631, 941  
**MSG\_NOSIGNAL** constant, 611  
**MSG\_OOB** constant, 611–613, 626  
**MSG\_PEEK** constant, 612  
**msgrecv** function, 451, 558–559, 561, 564, 585, 631,  
     941  
     definition of, 564  
**msgsnd** function, 451, 558, 560–561, 563–565, 633  
     definition of, 563  
**MSG\_TRUNC** constant, 612–613  
**MSGVERB** environment variable, 211  
**MSG\_WAITALL** constant, 612  
**MS\_INVALIDATE** constant, 528  
**msqid\_ds** structure, 561–562, 564  
**MS\_SYNC** constant, 528, 530  
**msync** function, 451, 528, 530  
     definition of, 528  
**Mui**, L., 712, 953  
multiplexing, I/O, 500–509  
**munmap** function, 528–529  
     definition of, 528  
**mutex** attributes, 430–439  
**mutex** timing comparison, 571  
**mutexes**, 399–409  
**mv** program, 115  
**myftw** function, 133, 141
- named full-duplex pipes, 534  
**NAME\_MAX** constant, 38–39, 44, 49, 55, 65, 131  
**nanosleep** function, 373–375, 437, 439, 451, 462,  
     837, 934  
     definition of, 374  
**Nataros**, S., xxxii  
Native POSIX Threads Library, *see* NPTL  
**nawk** program, 262  
**NCCS** constant, 674  
**ndbm** library, 744  
**<ndbm.h>** header, 30  
**Nemeth**, E., xxxii, 952  
**<netdb.h>** header, 29, 186  
**netent** structure, 598  
**<net/if.h>** header, 29  
**<netinet/in.h>** header, 29, 595, 605  
**<netinet/tcp.h>** header, 29  
Network File System, Sun Microsystems, *see* NFS  
Network Information Service, *see* NIS  
network logins, 290–293
- network printer communication, 789–843  
Neville-Neil, G. V., 74, 112, 116, 951  
**newgrp** program, 183  
**nfds\_t** data type, 507  
**\_NFILE** constant, 51  
**NFS** (Network File System, Sun Microsystems), 76,  
     787  
**nftw** function, 122, 131–132, 135, 442, 452, 910  
**NGROUPS\_MAX** constant, 39, 43, 49, 183–184  
**nice** function, 276–277, 279  
     definition of, 276  
**nice value**, 252, 276–277, 279  
Nievergelt, J., 744, 750, 949  
**NIS** (Network Information Service), 185  
**NIS+**, 185  
**NL** terminal character, 678, 680–681, 687, 700, 703  
**NLO** constant, 689  
**NL1** constant, 689  
**NL\_ARGMAX** constant, 39  
**NLDLY** constant, 676, 684, 689  
**nlink\_t** data type, 59, 114  
**n1\_langinfo** function, 442  
**NL\_LANGMAX** constant, 41  
**NL\_MSGMAX** constant, 39  
**NL\_SETMAX** constant, 39  
**NLSPATH** environment variable, 211  
**NL\_TEXTMAX** constant, 39  
**<n1\_types.h>** header, 29  
nobody login name, 178–179  
**NOFILE** constant, 51  
**NOFLSH** constant, 676, 689  
**NOKERNINFO** constant, 676, 682, 689  
**nologin** program, 179  
nonblocking  
     I/O, 481–484  
     socket I/O, 608–609, 627  
noncanonical mode, terminal I/O, 703–710  
nonfatal error, 16  
nonlocal goto, 213–220, 355–358  
**NPTL** (Native POSIX Threads Library), xxiii, 388  
**ntoh1** function, 594, 811, 825, 842  
     definition of, 594  
**ntohs** function, 594, 604, 842  
     definition of, 594  
**NULL** constant, 823  
null signal, 314, 337  
**NZERO** constant, 41, 276–278
- O\_ACCMODE** constant, 83–84  
**O\_APPEND** constant, 63, 66, 72, 77–78, 83–84, 149,  
     497, 511

- O\_ASYNC** constant, 83, 511, 627  
**O\_CLOEXEC** constant, 63  
**O\_CREAT** constant, 63, 66, 79, 89, 121, 125, 474, 496–498, 517–518, 529, 558, 579–580, 584, 749, 758, 818, 930  
**OCRNL** constant, 676, 689  
**od** program, 69  
**O\_DIRECT** constant, 150  
**O\_DIRECTORY** constant, 63  
**O\_DSYNC** constant, 64, 83, 513  
**O\_EXCL** constant, 63, 79, 121, 558, 580, 584  
**O\_EXEC** constant, 83  
**OFDEL** constant, 676, 684, 689  
**off\_t** data type, 59, 67–70, 157–158, 772  
**OFILL** constant, 676, 684, 689  
**O\_FSYNC** constant, 64, 83–84  
**OLCUC** constant, 676, 689  
Olson, M., 952  
**O\_NDELAY** constant, 36, 63, 482  
**ONLCR** constant, 676, 690, 731, 738  
**ONLRET** constant, 676, 690  
**ONOCR** constant, 676, 690  
**O\_NOCTTY** constant, 63, 297–298, 466, 723–724, 726  
**ONOEOT** constant, 676, 690  
**O\_NOFOLLOW** constant, 63  
**O\_NONBLOCK** constant, 36, 63, 83–84, 482–483, 496, 498, 553, 611–612, 934, 937  
**open** function, 8, 14, 61–66, 77, 79, 83, 89, 91, 100–101, 103–104, 112, 118, 120–125, 127–128, 137, 148–150, 283, 287, 297–298, 331, 451, 468, 470, 474, 482, 492–493, 495–498, 517–518, 525, 529, 553, 556, 558, 560, 577–578, 585, 588, 592, 653, 656–657, 669–670, 685, 723, 725–726, 745, 757–758, 808, 818, 823, 833, 907, 909, 930, 937  
definition of, 62  
Open Group, The, xxi, xxvi, 31, 196, 950  
Open Software Foundation, *see* OSF  
**openat** function, 62–66, 331, 451  
definition of, 62  
**opend.h** header, 656, 660, 942  
**opendir** function, 5, 7, 121, 130–135, 252–253, 283, 452, 697, 822, 910  
definition of, 130  
**openlog** function, 452, 468, 470–471, 480, 902, 928  
definition of, 470  
**OPEN\_MAX** constant, 40, 43, 49, 51–53, 60, 62, 906  
**open\_max** function, 466, 544, 546, 666, 896  
definition of, 52, 907  
**open\_memstream** function, 171–174  
definition of, 173  
OpenServer, 485  
OpenSolaris, xxi  
OpenSS7, 534  
**open\_wmemstream** function, 171–174  
definition of, 173  
**OPOST** constant, 676, 690, 706–708, 710  
**optarg** variable, 663  
**opterr** variable, 663  
**optind** variable, 808  
option codes, 31  
options, 53–57  
socket, 623–625  
**optopt** variable, 663  
Oracle Corporation, xxi–xxii, 35  
**O\_RDONLY** constant, 62, 83–84, 100, 103, 517–518, 529, 654, 808, 833, 937  
**O\_RDWR** constant, 62, 83–84, 100, 128, 468, 474, 498, 517–518, 529, 577, 723, 725, 749, 818, 930  
O'Reilly, T., 712, 953  
orientation, stream, 144  
orphaned process group, 307–309, 469, 735  
**O\_RSYNC** constant, 64, 83  
**O\_SEARCH** constant, 63, 83  
OSF (Open Software Foundation), 31–32  
**O\_SYNC** constant, 63–64, 83–84, 86–87, 513, 520  
**O\_TRUNC** constant, 63, 66, 100, 112, 125, 127–128, 149, 496, 498, 517–518, 529, 749  
**O\_TTY\_INIT** constant, 64, 683, 722  
out-of-band data, 626  
ownership  
directory, 101–102  
file, 101–102  
**O\_WRONLY** constant, 62, 83–84, 100, 937  
OXTABS constant, 676, 690
- packet mode, pseudo terminal, 740  
page cache, 81  
page size, 573  
pagedaemon process, 228  
PAGER environment variable, 539, 542–543  
PAGESIZE constant, 40, 43, 49  
PAGE\_SIZE constant, 41, 43, 49  
P\_ALL constant, 244  
PARENB constant, 675, 688, 690, 706–708  
parent  
directory, 4, 108, 125, 129  
process ID, 228, 233, 237, 243, 246, 252, 287–288, 309, 464  
PAREXT constant, 675, 690  
parity, terminal I/O, 688  
PARMRK constant, 676, 685, 688, 690  
PARODD constant, 675, 685, 688, 690, 713

- Partridge, C., xxxii  
 passing, file descriptor, 587, 642–652  
*passwd* program, 99, 182, 720  
*passwd* structure, 177, 180, 332, 809, 814, 918  
 password  
     file, 177–181  
     implementation differences, 184–185  
     shadow, 181–182, 196, 918  
 PATH environment variable, 100, 211, 250–251,  
     253, 260, 263, 265, 288–289  
 path\_alloc function, 133, 137, 896, 912  
     definition of, 50  
 pathconf function, 37, 39, 41–48, 50–51, 53–55,  
     57, 65, 110, 121, 452, 537  
     definition of, 42  
 PATH\_MAX constant, 38–39, 44, 49–50, 142, 911  
 pathname, 5  
     absolute, 5, 8, 43, 50, 64, 136, 141–142, 260, 553,  
     911  
     relative, 5, 8, 43–44, 50, 64–65, 135, 553  
     truncation, 65–66  
 pause function, 324, 327–328, 331, 334, 338–343,  
     356, 359, 365, 374, 451, 460, 711, 924, 930–931  
     definition of, 338  
 \_PC\_2\_SYMLINKS constant, 55  
 \_PC\_ASYNC\_IO constant, 55  
 \_PC\_CHOWN\_RESTRICTED constant, 55  
 \_PC\_FILESIZEBITS constant, 42, 44  
 PCFS file system, 49, 57, 113  
 pckt STREAMS module, 716, 740  
 \_PC\_LINK\_MAX constant, 42, 44  
 pclose function, 267, 452, 541–548, 616, 622,  
     935–937  
     definition of, 541, 545  
 \_PC\_MAX\_CANON constant, 42, 44, 47  
 \_PC\_MAX\_INPUT constant, 42, 44  
 \_PC\_NAME\_MAX constant, 42, 44  
 \_PC\_NO\_TRUNC constant, 55, 57  
 \_PC\_PATH\_MAX constant, 43–44, 51  
 \_PC\_PIPE\_BUF constant, 44  
 \_PC\_PRIO\_IO constant, 55  
 \_PC\_SYMLINK\_MAX constant, 44  
 \_PC\_SYNC\_IO constant, 55  
 \_PC\_TIMESTAMP\_RESOLUTION constant, 42, 44  
 \_PC\_VDISABLE constant, 54–55, 679  
 PENDIN constant, 676, 690  
 Pentium, xxii, xxvii  
 permissions, file access, 99–101, 140  
 perror function, 15–16, 24, 334, 379, 452, 600, 905  
     definition of, 15  
 pggrp structure, 311–312  
 PID, *see* process ID  
 pid\_t data type, 11, 59, 293, 384  
 Pike, R., 229, 950, 952  
 pipe function, 125, 148, 331, 535, 537–538, 540,  
     544, 546, 550, 565, 630, 934  
     definition of, 535  
 PIPE\_BUF constant, 39, 44, 49, 532, 537, 554–555,  
     935  
 pipes, 534–541  
     full-duplex, 534  
     half-duplex, 534  
     mounted STREAMS-based, 534  
     named full-duplex, 534  
     timing full-duplex, 565  
 Pippenger, N., 744, 750, 949  
 Plan 9 operating system, 229, 952  
 Plauger, P. J., 26, 164, 323, 952  
 pointer, generic, 71, 208  
 poll function, 319, 330–331, 343, 451, 481,  
     501–502, 506–509, 531–532, 560, 586, 588,  
     592, 608–609, 627, 631–632, 659, 664,  
     666–668, 718, 732, 742, 933–934, 936–937, 942  
     definition of, 506  
 POLLERR constant, 508  
 pollfd structure, 507, 632, 666, 668, 934, 941  
 <poll.h> header, 29, 507  
 POLLHUP constant, 508, 667–668, 936  
 POLLIN constant, 508, 632, 666–668, 936, 941–942  
 polling, 246, 484, 501  
 POLLNVAL constant, 508  
 POLLOUT constant, 508  
 POLLPRI constant, 508  
 POLLRDBAND constant, 508  
 POLLRDNORM constant, 508  
 POLLWRBAND constant, 508  
 POLLWRNORM constant, 508  
 popen function, 23, 242, 249, 267, 452, 541–548,  
     587–588, 615, 619, 622–623, 935–937  
     definition of, 541, 543  
 port number, 593, 595–596, 598–601, 605  
 Portable Operating System Environment for  
     Computer Environments, IEEE, *see* POSIX  
 POSIX (Portable Operating System Environment  
     for Computer Environments, IEEE), xix,  
     xxxi, 26–30, 33, 265, 561, 674  
 POSIX semaphores, 579–584  
 POSIX.1, xxvi, xxxi, 4, 9, 27, 38, 41, 50, 53, 57–58,  
     88, 257, 262, 329, 367–368, 384, 533, 546, 553,  
     589, 617, 744, 950  
 POSIX.2, 262  
 \_POSIX2\_SYMLINKS constant, 55  
 \_POSIX\_ADVISORY\_INFO constant, 31  
 \_POSIX\_AIO\_LISTIO\_MAX constant, 515  
 \_POSIX\_AIO\_MAX constant, 515  
 \_POSIX\_ARG\_MAX constant, 39–40

---

\_POSIX\_ASYNCHRONOUS\_IO constant, 54, 57  
 \_POSIX\_ASYNC\_IO constant, 55  
 \_POSIX\_BARRIERS constant, 54, 57  
 \_POSIX\_CHILD\_MAX constant, 39–40  
 \_POSIX\_CHOWN\_RESTRICTED constant, 55, 57,  
     110  
 \_POSIX\_CLOCKRES\_MIN constant, 38  
 \_POSIX\_CLOCK\_SELECTION constant, 54, 57  
 \_POSIX\_CPUTIME constant, 31, 189  
 \_POSIX\_C\_SOURCE constant, 57–58, 84, 240  
 \_POSIX\_DELAYTIMER\_MAX constant, 39–40  
 posix\_fadvise function, 452  
 posix\_fallocate function, 452  
 \_POSIX\_FSYNC constant, 31  
 \_POSIX\_HOST\_NAME\_MAX constant, 39–40  
 \_POSIX\_IPV6 constant, 31  
 \_POSIX\_JOB\_CONTROL constant, 57  
 \_POSIX\_LINK\_MAX constant, 39  
 \_POSIX\_LOGIN\_NAME\_MAX constant, 39–40  
 POSIXLY\_CORRECT environment variable, 111  
 posix\_madvise function, 452  
 \_POSIX\_MAPPED\_FILES constant, 54, 57  
 \_POSIX\_MAX\_CANON constant, 39  
 \_POSIX\_MAX\_INPUT constant, 39  
 \_POSIX\_MEMLOCK constant, 31  
 \_POSIX\_MEMLOCK\_RANGE constant, 31  
 \_POSIX\_MEMORY\_PROTECTION constant, 54, 57  
 \_POSIX\_MESSAGE\_PASSING constant, 31  
 \_POSIX\_MONOTONIC\_CLOCK constant, 31, 189  
 \_POSIX\_NAME\_MAX constant, 39, 580  
 \_POSIX\_NGROUPS\_MAX constant, 39  
 \_POSIX\_NO\_TRUNC constant, 55, 57, 65  
 \_POSIX\_OPEN\_MAX constant, 39–40  
 posix\_openpt function, 452, 722–725  
     definition of, 722  
 \_POSIX\_PATH\_MAX constant, 39–40, 696–697  
 \_POSIX\_PIPE\_BUF constant, 39  
 \_POSIX\_PRIO\_IO constant, 55  
 \_POSIX\_PRIORITIZED\_IO constant, 31  
 \_POSIX\_PRIORITY\_SCHEDULING constant, 31  
 \_POSIX\_RAW\_SOCKETS constant, 31  
 \_POSIX\_READER\_WRITER\_LOCKS constant, 55,  
     57  
 \_POSIX\_REALTIME\_SIGNALS constant, 55, 57  
 \_POSIX\_RE\_DUP\_MAX constant, 39  
 \_POSIX\_RTSIG\_MAX constant, 39–40  
 \_POSIX\_SAVED\_IDS constant, 57, 98, 256, 337  
 \_POSIX\_SEMAPHORES constant, 55, 57  
 \_POSIX\_SEM\_NSEMS\_MAX constant, 39–40  
 \_POSIX\_SEM\_VALUE\_MAX constant, 39–40  
 \_POSIX\_SHARED\_MEMORY\_OBJECTS constant, 31  
 \_POSIX\_SHELL constant, 57  
 \_POSIX\_SIGQUEUE\_MAX constant, 39–40  
 \_POSIX\_SOURCE constant, 57  
 \_POSIX\_SPAWN constant, 31  
 posix\_spawn function, 452  
 posix\_spawnp function, 452  
 \_POSIX\_SPIN\_LOCKS constant, 55, 57  
 \_POSIX\_SPORADIC\_SERVER constant, 31  
 \_POSIX\_SSIZE\_MAX constant, 39  
 \_POSIX\_STREAM\_MAX constant, 39–40  
 \_POSIX\_SYMLINK\_MAX constant, 39  
 \_POSIX\_SYMLOOP\_MAX constant, 39–40  
 \_POSIX\_SYNCHRONIZED\_IO constant, 31  
 \_POSIX\_SYNC\_IO constant, 55  
 \_POSIX\_THREAD\_ATTR\_STACKADDR constant,  
     31, 429  
 \_POSIX\_THREAD\_ATTR\_STACKSIZE constant,  
     31, 429  
 \_POSIX\_THREAD\_CPUTIME constant, 31, 189  
 \_POSIX\_THREAD\_PRIO\_INHERIT constant, 31  
 \_POSIX\_THREAD\_PRIO\_PROTECT constant, 31  
 \_POSIX\_THREAD\_PRIORITY\_SCHEDULING  
     constant, 31  
 \_POSIX\_THREAD\_PROCESS\_SHARED constant,  
     31, 431  
 \_POSIX\_THREAD\_ROBUST\_PRIO\_INHERIT  
     constant, 31  
 \_POSIX\_THREAD\_ROBUST\_PRIO\_PROTECT  
     constant, 31  
 \_POSIX\_THREADS constant, 55, 57, 384  
 \_POSIX\_THREAD\_SAFE\_FUNCTIONS constant,  
     55, 57, 442  
 \_POSIX\_THREAD\_SPORADIC\_SERVER constant,  
     31  
 \_POSIX\_TIMEOUTS constant, 55  
 \_POSIX\_TIMER\_MAX constant, 39–40  
 \_POSIX\_TIMERS constant, 55, 57  
 \_POSIX\_TIMESTAMP\_RESOLUTION constant, 44  
 posix\_trace\_event function, 331  
 \_POSIX\_TTY\_NAME\_MAX constant, 39–40  
 posix\_TYPED\_mem\_open function, 452  
 \_POSIX\_TYPED\_MEMORY\_OBJECTS constant, 31  
 \_POSIX\_TZNAME\_MAX constant, 39–40  
 \_POSIX\_V6\_ILP32\_OFF32 constant, 70  
 \_POSIX\_V6\_ILP32\_OFFBIG constant, 70  
 \_POSIX\_V6\_LP64\_OFF64 constant, 70  
 \_POSIX\_V6\_LP64\_OFFBIG constant, 70  
 \_POSIX\_V7\_ILP32\_OFF32 constant, 70  
 \_POSIX\_V7\_ILP32\_OFFBIG constant, 70  
 \_POSIX\_V7\_LP64\_OFF64 constant, 70  
 \_POSIX\_V7\_LP64\_OFFBIG constant, 70  
 \_POSIX\_VDISABLE constant, 55, 57, 678–679  
 \_POSIX\_VERSION constant, 57, 188  
 PowerPC, xxi–xxii, xxvii  
 P\_PGID constant, 244

**PPID**, *see* parent process ID  
**P\_PID** constant, 244  
**pr** program, 753  
**prctl** program, 559  
**pread** function, 78, 451, 461–462, 592  
     definition of, 78  
**Presotto, D. L.**, xxxii, 229, 952  
**pr\_exit** function, 239–241, 266–268, 281, 283, 372, 896  
     definition of, 240  
**primitive system data types**, 58  
**print** program, 794, 801, 820, 824–825, 834, 843  
**printd** program, 794, 843  
**printer communication, network**, 789–843  
**printer spooling**, 793–795  
     source code, 795–842  
**printer\_status** function, 814, 837–838, 843  
     definition of, 838  
**printer\_thread** function, 814, 832, 945  
     definition of, 832  
**printf** function, 10–11, 21, 150, 159, 161–163, 175, 192, 194, 219, 226, 231, 235, 283, 309, 330, 349, 452, 552, 919–920  
     definition of, 159  
**print.h** header, 815, 820, 825  
**printreq** structure, 801, 809–810, 812, 820, 822–824, 827  
**printresp** structure, 801, 809, 811, 824–827  
**PRIOPGRP** constant, 277  
**PRIOPPROCESS** constant, 277  
**PRIOPUSER** constant, 277  
**privilege, least**, 256, 795, 816  
**pr\_mask** function, 356–357, 360–361, 896  
     definition of, 347  
`/proc`, 136, 253  
**proc** structure, 311–312  
**process**, 11  
     accounting, 269–275  
     control, 11, 227–283  
     ID, 11, 228, 252  
     ID, parent, 228, 233, 237, 243, 246, 252, 287–288, 309, 464  
     identifiers, 227–228  
     relationships, 285–312  
     scheduling, 276–280  
     system, 228, 337  
     termination, 198–202  
     time, 20, 24, 59, 280–282  
**process group**, 293–294  
     background, 296, 300, 302, 304, 306–307, 309, 321, 369, 377, 944  
     foreground, 296, 298, 300–303, 306, 311, 318–322, 369, 377, 680–682, 685, 689, 710, 719, 741, 944  
     ID, 233, 252  
     ID, foreground, 298, 303, 677  
     ID, session, 304  
     ID, terminal, 303, 463  
     leader, 294–296, 306, 312, 465–466, 727  
     lifetime, 294  
     orphaned, 307–309, 469, 735  
**processes**, cooperating, 495, 752, 945  
**process-shared** attribute, 431  
     .profile file, 289  
**program**, 10  
**PROT\_EXEC** constant, 525  
**PROT\_NONE** constant, 525  
**protoent** structure, 598  
**prototypes**, function, 845–893  
**PROT\_READ** constant, 525, 529, 577  
**PROT\_WRITE** constant, 525, 529, 577  
**PR\_TEXT** constant, 801, 810, 825, 835–836  
**ps** program, 237, 283, 303, 306–307, 463–465, 468–469, 480, 736, 923  
**pselect** function, 331, 451, 501, 506  
     definition of, 506  
**pseudo terminal**, 715–742  
     packet mode, 740  
     remote mode, 741  
     signal generation, 741  
     window size, 741  
**psiginfo** function, 379–380, 452  
     definition of, 379  
**psignal** function, 379–380, 452  
     definition of, 379  
**ptem** STREAMS module, 716, 726  
**pthread** structure, 385  
**pthread\_atfork** function, 457–461  
     definition of, 458  
**pthread\_attr\_destroy** function, 427–429  
     definition of, 427  
**pthread\_attr\_getdetachstate** function, 428  
     definition of, 428  
**pthread\_attr\_getguardsize** function, 430  
     definition of, 430  
**pthread\_attr\_getstack** function, 429  
     definition of, 429  
**pthread\_attr\_getstacksize** function, 429–430  
     definition of, 430  
**pthread\_attr\_init** function, 427–429  
     definition of, 427  
**pthread\_attr\_setdetachstate** function, 428  
     definition of, 428  
**pthread\_attr\_setguardsize** function, 430  
     definition of, 430  
**pthread\_attr\_setstack** function, 429  
     definition of, 429

**pthread\_attr\_setstacksize** function, 429–430  
     definition of, 430  
**pthread\_attr\_t** data type, 427–428, 430, 451  
**pthread\_barrierattr\_destroy** function, 441  
     definition of, 441  
**pthread\_barrierattr\_getpshared** function, 441  
     definition of, 441  
**pthread\_barrierattr\_init** function, 441  
     definition of, 441  
**pthread\_barrierattr\_setpshared** function, 441  
     definition of, 441  
**pthread\_barrier\_destroy** function, 418–419  
     definition of, 418  
**pthread\_barrier\_init** function, 418–419, 421  
     definition of, 418  
**PTHREAD\_BARRIER\_SERIAL\_THREAD** constant, 419, 422  
**pthread\_barrier\_t** data type, 419  
**pthread\_barrier\_wait** function, 419–423  
     definition of, 419  
**pthread\_cancel** function, 393, 451, 453, 828  
     definition of, 393  
**PTHREAD\_CANCEL\_ASYNCHRONOUS** constant, 453  
**PTHREAD\_CANCEL\_DEFERRED** constant, 453  
**PTHREAD\_CANCEL\_DISABLE** constant, 451  
**PTHREAD\_CANCELED** constant, 389, 393  
**PTHREAD\_CANCEL\_ENABLE** constant, 451  
**pthread\_cleanup\_pop** function, 394–396, 827, 829  
     definition of, 394  
**pthread\_cleanup\_push** function, 394–396, 824  
     definition of, 394  
**pthread\_condattr\_destroy** function, 440  
     definition of, 440  
**pthread\_condattr\_getclock** function, 441  
     definition of, 441  
**pthread\_condattr\_getpshared** function, 440  
     definition of, 440  
**pthread\_condattr\_init** function, 440  
     definition of, 440  
**pthread\_condattr\_setclock** function, 441  
     definition of, 441  
**pthread\_condattr\_setpshared** function, 440  
     definition of, 440  
**pthread\_condattr\_t** data type, 441  
**pthread\_cond\_broadcast** function, 415, 422–423, 927  
     definition of, 415  
**pthread\_cond\_destroy** function, 414, 462  
     definition of, 414  
**pthread\_cond\_init** function, 414, 462, 941  
     definition of, 414  
**PTHREAD\_COND\_INITIALIZER** constant, 413, 416, 455, 814  
**pthread\_cond\_signal** function, 415–416, 456, 821, 942  
     definition of, 415  
**pthread\_cond\_t** data type, 413, 416, 455, 814, 940  
**pthread\_cond\_timedwait** function, 414–415, 434, 440–441, 451  
     definition of, 414  
**pthread\_cond\_wait** function, 414–416, 434, 451, 456, 832, 927, 941  
     definition of, 414  
**pthread\_create** function, 385–388, 390–392, 395, 397, 421, 427–428, 456, 460, 477, 632, 817, 926, 941  
     definition of, 385  
**PTHREAD\_CREATE\_DETACHED** constant, 428  
**PTHREAD\_CREATE\_JOINABLE** constant, 428  
**PTHREAD\_DESTRUCTOR\_ITERATIONS** constant, 426, 447  
**pthread\_detach** function, 396–397, 427  
     definition of, 397  
**pthread\_equal** function, 385, 412  
     definition of, 385  
**pthread\_exit** function, 198, 236, 389–391, 393–396, 447, 824–829  
     definition of, 389  
**pthread\_getspecific** function, 449–450  
     definition of, 449  
**<pthread.h>** header, 29  
**pthread\_join** function, 389–391, 395–396, 418, 451, 926  
     definition of, 389  
**pthread\_key\_create** function, 447–448, 450  
     definition of, 447  
**pthread\_key\_delete** function, 447–448  
     definition of, 448  
**PTHREAD\_KEYS\_MAX** constant, 426, 447  
**pthread\_key\_t** data type, 449  
**pthread\_kill** function, 455  
     definition of, 455  
**pthread\_mutexattr\_destroy** function, 431, 445  
     definition of, 431  
**pthread\_mutexattr\_getpshared** function, 431  
     definition of, 431  
**pthread\_mutexattr\_getrobust** function, 432  
     definition of, 432  
**pthread\_mutexattr\_gettype** function, 434

**definition of**, 434  
**pthread\_mutexattr\_init** function, 431, 438, 445  
**definition of**, 431  
**pthread\_mutexattr\_setpshared** function, 431  
**definition of**, 431  
**pthread\_mutexattr\_setrobust** function, 432  
**definition of**, 432  
**pthread\_mutexattr\_settype** function, 434, 438, 445  
**definition of**, 434  
**pthread\_mutexattr\_t** data type, 430–431, 438, 445  
**pthread\_mutex\_consistent** function, 432–433, 571  
**definition of**, 433  
**PTHREAD\_MUTEX\_DEFAULT** constant, 433–434  
**pthread\_mutex\_destroy** function, 400–401, 404, 407  
**definition of**, 400  
**PTHREAD\_MUTEX\_ERRORCHECK** constant, 433–434  
**pthread\_mutex\_init** function, 400–401, 403, 405, 431, 438, 445, 941  
**definition of**, 400  
**PTHREAD\_MUTEX\_INITIALIZER** constant, 400, 403, 405, 408, 416, 431, 449, 455, 459, 813–814  
**pthread\_mutex\_lock** function, 400–401, 403–404, 406–408, 416, 422–423, 432, 438, 445, 450, 456, 459–460, 820–821, 828–830, 832–833, 941–942  
**definition of**, 400  
**PTHREAD\_MUTEX\_NORMAL** constant, 433–434  
**PTHREAD\_MUTEX\_RECURSIVE** constant, 433–434, 438, 445  
**PTHREAD\_MUTEX\_ROBUST** constant, 432  
**PTHREAD\_MUTEX\_STALLED** constant, 432  
**pthread\_mutex\_t** data type, 400–401, 403, 405, 408, 416, 438, 445, 449, 455, 459, 813–814, 940  
**pthread\_mutex\_timedlock** function, 407–409, 413  
**definition of**, 407  
**pthread\_mutex\_trylock** function, 400, 402  
**definition of**, 400  
**pthread\_mutex\_unlock** function, 400–401, 403–404, 406–407, 416, 422–423, 438–439, 445, 450, 456, 460, 820–821, 828–830, 832–833, 941–942  
**definition of**, 400  
**pthread\_once** function, 445, 448, 450, 928  
**definition of**, 448  
**PTHREAD\_ONCE\_INIT** constant, 445, 448–449  
**pthread\_once\_t** data type, 445, 449  
**PTHREAD\_PROCESS\_PRIVATE** constant, 417, 431, 442  
**PTHREAD\_PROCESS\_SHARED** constant, 417, 431, 442, 571  
**pthread\_rwlockattr\_destroy** function, 439  
**definition of**, 439  
**pthread\_rwlockattr\_getpshared** function, 440  
**definition of**, 440  
**pthread\_rwlockattr\_init** function, 439  
**definition of**, 439  
**pthread\_rwlockattr\_setpshared** function, 440  
**definition of**, 440  
**pthread\_rwlockattr\_t** data type, 439  
**pthread\_rwlock\_destroy** function, 409–410  
**definition of**, 409  
**pthread\_rwlock\_init** function, 409, 411  
**definition of**, 409  
**PTHREAD\_RWLOCK\_INITIALIZER** constant, 409  
**pthread\_rwlock\_rdlock** function, 410, 412, 452  
**definition of**, 410  
**pthread\_rwlock\_t** data type, 411  
**pthread\_rwlock\_timedrdlock** function, 413, 452  
**definition of**, 413  
**pthread\_rwlock\_timedwrlock** function, 413, 452  
**definition of**, 413  
**pthread\_rwlock\_tryrdlock** function, 410  
**definition of**, 410  
**pthread\_rwlock\_trywrlock** function, 410  
**definition of**, 410  
**pthread\_rwlock\_unlock** function, 410–412  
**definition of**, 410  
**pthread\_rwlock\_wrlock** function, 410–412, 452  
**definition of**, 410  
**pthreads**, 27, 229, 384, 426  
**pthread\_self** function, 385, 387, 391, 824  
**definition of**, 385  
**pthread\_setcancelstate** function, 451  
**definition of**, 451  
**pthread\_setcanceltype** function, 453  
**definition of**, 453  
**pthread\_setspecific** function, 449–450  
**definition of**, 449  
**pthread\_sigmask** function, 453–454, 477, 815  
**definition of**, 454  
**pthread\_spin\_destroy** function, 417  
**definition of**, 417

**pthread\_spin\_init** function, 417  
     definition of, 417  
**pthread\_spin\_lock** function, 418  
     definition of, 418  
**pthread\_spin\_trylock** function, 418  
     definition of, 418  
**pthread\_spin\_unlock** function, 418  
     definition of, 418  
**PTHREAD\_STACK\_MIN** constant, 426, 430  
**pthread\_t** data type, 59, 384–385, 387, 390–391,  
     395, 411, 421, 428, 456, 460, 476, 632, 812, 814,  
     824, 829, 926, 941  
**pthread\_testcancel** function, 451, 453  
     definition of, 453  
**PTHREAD\_THREADS\_MAX** constant, 426  
**ptrdiff\_t** data type, 59  
**ptsname** function, 442, 723–725  
     definition of, 723  
**pty** program, 309, 715, 720–721, 727, 729–742, 944  
**pty\_fork** function, 721, 724, 726–730, 732, 739,  
     741–742  
     definition of, 727  
**ptym\_open** function, 724, 726–728, 897  
     definition of, 724–725  
**ptys\_fork** function, 897  
**ptys\_open** function, 724, 726–728, 897  
     definition of, 724–725  
Pu, C., 65, 953  
**putc** function, 10, 152–156, 247–248, 452, 701  
     definition of, 152  
**putchar** function, 152, 175, 452, 547–548  
     definition of, 152  
**putchar\_unlocked** function, 442, 444, 452  
     definition of, 444  
**putc\_unlocked** function, 442, 444, 452  
     definition of, 444  
**putenv** function, 204, 212, 251, 442, 446, 462  
     definition of, 212  
**putenv\_r** function, 462  
**puts** function, 152–153, 452, 911  
     definition of, 153  
**pututxline** function, 442, 452  
**putwc** function, 452  
**putwchar** function, 452  
**PWD** environment variable, 211  
**<pwd.h>** header, 29, 177, 186  
**pwrite** function, 78–79, 451, 461–462, 592  
     definition of, 78

Quarterman, J. S., 33–34, 74, 112, 116, 229, 236, 525,  
     951  
QUIT terminal character, 678, 681, 688, 702

race conditions, 245–249, 339, 784, 922, 924  
Rago, J. E., xxvii  
Rago, S. A., xxxii, 88, 157, 290, 952  
**raise** function, 331, 336–338, 365  
     definition of, 337  
**rand** function, 442  
raw terminal mode, 672, 704, 708, 713, 732, 734  
Raymond, E. S., 952  
**read** function, 8–10, 20, 59, 61, 64, 71–72, 78, 88,  
     90–91, 111, 124–125, 130, 145, 154–156, 174,  
     301, 308–309, 328–331, 342–343, 364–365,  
     378, 451, 462, 470, 482–483, 495–496,  
     498–502, 505–506, 508–509, 513, 517,  
     523–525, 530–531, 536–537, 540–541,  
     549–551, 553, 556, 587, 590, 592, 610, 612, 654,  
     656, 665–667, 672, 702–704, 708–709,  
     732–733, 738, 740, 748, 752, 765, 767–768,  
     805–806, 811, 818, 823, 836–838, 907–908,  
     936, 943  
     definition of, 71  
**read**, scatter, 521, 644  
**readdir** function, 5, 7, 130–135, 442, 452, 697, 823  
     definition of, 130  
**readdir\_r** function, 443, 452  
reader-writer lock attributes, 439–440  
reader-writer locks, 409–413  
reading directories, 130–135  
**readlink** function, 121, 123–124, 331, 452  
     definition of, 123  
**readlinkat** function, 123–124, 331, 452  
     definition of, 123  
**read\_lock** function, 489, 493, 498, 897  
**readmore** function, 814, 837, 840–841  
     definition of, 837  
**readn** function, 523–524, 738, 806, 811, 896  
     definition of, 523–524  
**readv** function, 41, 43, 329, 451, 481, 521–523,  
     531, 592, 613, 644, 752, 766  
     definition of, 521  
**readw\_lock** function, 489, 759, 763, 780, 897  
**real**  
     group ID, 98, 102, 183, 228, 233, 252–253, 256,  
     270, 585  
     user ID, 39–40, 43, 98–99, 102, 221, 228, 233,  
     252–253, 256–260, 270, 276, 286, 288, 337,  
     381, 585, 924  
**realloc** function, 50, 174, 207–208, 213, 661–662,  
     666, 761, 838, 840, 911–912  
     definition of, 207  
record locking, 485–499  
     advisory, 495  
     deadlock, 490  
     mandatory, 495

- timing comparison, 571  
**recv** function, 331, 451, 592, 612–615, 626–627  
  definition of, 612  
**recv\_fd** function, 642–644, 650, 655, 660, 896  
  definition of, 642, 647  
**recvfrom** function, 331, 451, 613, 620–623  
  definition of, 613  
**recvmsg** function, 331, 451, 613, 644, 646–648, 651  
  definition of, 613  
**recv\_ufd** function, 650  
  definition of, 651  
**RE\_DUP\_MAX** constant, 39, 43, 49  
reentrant functions, 330–332  
**regcomp** function, 39, 43  
**regexec** function, 39, 43  
*<regex.h>* header, 29  
register variables, 217  
regular file, 95  
relative pathname, 5, 8, 43–44, 50, 64–65, 135, 553  
reliable signals, 335–336  
remote mode, pseudo terminal, 741  
**remove** function, 116–119, 121, 125, 452  
  definition of, 119  
**remove\_job** function, 814, 822, 832  
  definition of, 822  
**rename** function, 119–121, 125, 331, 452  
  definition of, 119  
**renameat** function, 119–120, 331, 452  
  definition of, 119  
**replace\_job** function, 814, 821, 837  
  definition of, 821  
**REPRINT** terminal character, 678, 681, 687, 690, 703  
**reset** program, 713, 943  
resource limits, 220–225, 233, 252, 322, 382  
restarted system calls, 329–330, 342–343, 351, 354, 508, 700  
**restrict** keyword, 26, 93, 123, 146, 148, 152–153, 156, 158–159, 161–163, 190, 192, 195, 346, 350, 385, 400, 409, 414, 428–432, 434, 440–441, 454, 502, 506, 596, 599–600, 605, 608, 613, 624  
**rewind** function, 149, 158, 168, 452  
  definition of, 158  
**rewinddir** function, 130–135, 452  
  definition of, 130  
**rfork** function, 229  
Ritchie, D. M., xx, 26, 143, 149, 155, 162, 164, 208, 898, 906, 950, 952  
**RLIM\_INFINITY** constant, 221, 468  
**rlimit** structure, 220, 224, 467, 907  
**RLIMIT\_AS** constant, 221–223  
**RLIMIT\_CORE** constant, 221–223, 317  
**RLIMIT\_CPU** constant, 221–223  
**RLIMIT\_DATA** constant, 221–223  
**RLIMIT\_FSIZE** constant, 221–223, 382  
**RLIMIT\_INFINITY** constant, 224, 907  
**RLIMIT\_MEMLOCK** constant, 221–223  
**RLIMIT\_MSGQUEUE** constant, 221, 223  
**RLIMIT\_NICE** constant, 221, 223  
**RLIMIT\_NOFILE** constant, 221–223, 467, 907  
**RLIMIT\_NPROC** constant, 221–223  
**RLIMIT\_NPTS** constant, 221, 223  
**RLIMIT\_RSS** constant, 222–223  
**RLIMIT\_SBSIZE** constant, 222–223  
**RLIMIT\_SIGPENDING** constant, 222, 224  
**RLIMIT\_STACK** constant, 222, 224  
**RLIMIT\_SWAP** constant, 222, 224  
**RLIMIT\_VMEM** constant, 222, 224  
**rlim\_t** data type, 59, 223  
**rlogin** program, 717, 741–742  
**rlogind** program, 717, 734, 741, 944  
**rm** program, 559, 663  
**rmdir** function, 117, 119–120, 125, 129–130, 331  
  definition of, 130  
**robust** attribute, 431, 571  
**R\_OK** constant, 102–103  
root  
  directory, 4, 8, 24, 139, 141, 233, 252, 283, 910  
  login name, 16  
**routed** program, 472  
**rpcbind** program, 465  
RS-232, 674, 685–686  
**rsyslogd** program, 465, 480  
**RTSIG\_MAX** constant, 40, 43  
Rudoff, A. M., 157, 291, 470, 589, 953  
**runacct** program, 269  
**S5** file system, 65  
**sa** program, 269  
**sac** program, 290  
Sacken, J., xxxii  
**SAF** (Service Access Facility), 290  
safe, async-signal, 330, 446, 450, 457, 461–462, 927  
**sa\_handler** structure, 376  
**SA\_INTERRUPT** constant, 351, 354–355  
**s\_alloc** function, 584  
Salus, P. H., xxxii, 952  
**SA\_NOCLDSTOP** constant, 351  
**SA\_NOCLDWAIT** constant, 333, 351  
**SA\_NODEFER** constant, 351, 354  
Santa Cruz Operation, *see SCO*  
**SA\_ONSTACK** constant, 351

**SA\_RESETHAND** constant, 351, 354  
**SA\_RESTART** constant, 329, 351, 354, 508–509  
**SA\_SIGINFO** constant, 336, 350–353, 376, 512  
**saved**  
  set-group-ID, 56, 98, 257  
  set-user-ID, 56, 98, 256–260, 288, 337  
**S\_BANDURG** constant, 510  
**sbrk** function, 21–23, 208, 221  
**\_SC\_AIO\_MAX** constant, 516  
**\_SC\_AIO\_PRIO\_DELTA\_MAX** constant, 516  
**scaling**, frequency, 785  
**scanf** *configfile* function, 803–804  
  definition of, 803  
**scandir** function, 452  
**scanf** function, 150, 162–163, 452  
  definition of, 162  
**\_SC\_ARG\_MAX** constant, 43, 47  
**\_SC\_ASYNCHRONOUS\_IO** constant, 57  
**\_SC\_ATEXIT\_MAX** constant, 43  
**scatter read**, 521, 644  
**\_SC\_BARRIERS** constant, 57  
**\_SC\_CHILD\_MAX** constant, 43, 221  
**\_SC\_CLK\_TCK** constant, 42–43, 280–281  
**\_SC\_CLOCK\_SELECTION** constant, 57  
**\_SC\_COLL\_WEIGHTS\_MAX** constant, 43  
**\_SC\_DELAYTIMER\_MAX** constant, 43  
**SCHAR\_MAX** constant, 37–38  
**SCHAR\_MIN** constant, 37–38  
**<sched.h>** header, 29  
**scheduling**, process, 276–280  
**\_SC\_HOST\_NAME\_MAX** constant, 43, 616, 618, 623, 815  
**Schwartz**, A., 181, 250, 298, 949  
**\_SC\_IO\_LISTIO\_MAX** constant, 516  
**\_SC iov\_MAX** constant, 43  
**\_SC\_JOB\_CONTROL** constant, 54, 57  
**\_SC\_LINE\_MAX** constant, 43  
**\_SC\_LOGIN\_NAME\_MAX** constant, 43  
**\_SC\_MAPPED\_FILES** constant, 57  
**SCM\_CREDENTIALS** constant, 649–652  
**SCM\_CREDS** constant, 649–650, 652  
**SCM\_CREDTYPE** constant, 650, 652  
**\_SC\_MEMORY\_PROTECTION** constant, 57  
**SCM\_RIGHTS** constant, 645–646, 650, 652  
**\_SC\_NGROUPS\_MAX** constant, 43  
**\_SC\_NZERO** constant, 276, 278  
**SCO** (Santa Cruz Operation), 35  
**\_SC\_OPEN\_MAX** constant, 43, 52, 221, 907  
**\_SC\_PAGESIZE** constant, 43, 527  
**\_SC\_PAGE\_SIZE** constant, 43, 527  
**\_SC\_READER\_WRITER\_LOCKS** constant, 57  
**\_SC\_REALTIME\_SIGNALS** constant, 57  
**\_SC\_RE\_DUP\_MAX** constant, 43  
**script** program, 715, 719–720, 734, 736–737, 741–742  
**\_SC\_RTSIG\_MAX** constant, 43  
**\_SC\_SAVED\_IDS** constant, 54, 57, 98, 256  
**\_SC\_SEMAPHORES** constant, 57  
**\_SC\_SEM\_NSEMS\_MAX** constant, 43  
**\_SC\_SEM\_VALUE\_MAX** constant, 43  
**\_SC\_SHELL** constant, 57  
**\_SC\_SIGQUEUE\_MAX** constant, 43  
**\_SC\_SPIN\_LOCKS** constant, 57  
**\_SC\_STREAM\_MAX** constant, 43  
**\_SC\_SYMLOOP\_MAX** constant, 43  
**\_SC\_THREAD\_ATTR\_STACKADDR** constant, 429  
**\_SC\_THREAD\_ATTR\_STACKSIZE** constant, 429  
**\_SC\_THREAD\_DESTRUCTOR\_ITERATIONS** constant, 426  
**\_SC\_THREAD\_KEYS\_MAX** constant, 426  
**\_SC\_THREAD\_PROCESS\_SHARED** constant, 431  
**\_SC\_THREADS** constant, 57, 384  
**\_SC\_THREAD\_SAFE\_FUNCTIONS** constant, 57, 442  
**\_SC\_THREAD\_STACK\_MIN** constant, 426  
**\_SC\_THREAD\_THREADS\_MAX** constant, 426  
**\_SC\_TIMER\_MAX** constant, 43  
**\_SC\_TIMERS** constant, 57  
**\_SC\_TTY\_NAME\_MAX** constant, 43  
**\_SC\_TZNAME\_MAX** constant, 43  
**\_SC\_V7\_ILP32\_OFF32** constant, 70  
**\_SC\_V7\_ILP32\_OFFBIG** constant, 70  
**\_SC\_V7\_LP64\_OFF64** constant, 70  
**\_SC\_V7\_LP64\_OFFBIG** constant, 70  
**\_SC\_VERSION** constant, 50, 54, 57  
**\_SC\_XOPEN\_CRYPT** constant, 57  
**\_SC\_XOPEN\_REALTIME** constant, 57  
**\_SC\_XOPEN\_REALTIME\_THREADS** constant, 57  
**\_SC\_XOPEN\_SHM** constant, 57  
**\_SC\_XOPEN\_VERSION** constant, 50, 54, 57  
**<search.h>** header, 30  
**sed** program, 950  
**Seebass**, S., 952  
**seek** function, 67  
**SEEK\_CUR** constant, 67, 158, 486, 494–495, 766  
**seekdir** function, 130–135, 452  
  definition of, 130  
**SEEK\_END** constant, 67, 158, 486, 494–495, 771–773, 781  
**SEEK\_SET** constant, 67, 158, 172, 486, 494–495, 498, 759, 762–763, 765–766, 768–773, 775–780, 818–819, 930–931  
**SEGV\_ACCERR** constant, 353  
**SEGV\_MAPERR** constant, 353  
**select** function, 330–331, 343, 451, 481, 501–509, 531–532, 560, 586, 588, 592, 608–609, 626–627, 631–632, 659, 664–666, 668, 718,

732, 742, 805–806, 816–817, 928–929, 933, 936, 939, 942  
     definition of, 502  
 Seltzer, M., 744, 952  
 semaphore, 57, 534, 565–571  
     adjustment on exit, 570–571  
     locking timing comparison, 571, 583  
*<semaphore.h>* header, 29  
 sembuf structure, 568–569  
 sem\_close function, 580, 584  
     definition of, 580  
 semctl function, 558, 562, 566–568, 570  
     definition of, 567  
 sem\_destroy function, 582  
     definition of, 582  
 SEM\_FAILED constant, 584  
 semget function, 557–558, 566–567  
     definition of, 567  
 sem\_getvalue function, 582  
     definition of, 582  
 semid\_ds structure, 566–568  
 sem\_init function, 582  
     definition of, 582  
 SEM\_NSEMS\_MAX constant, 40, 43  
 semop function, 452, 559, 567–570  
     definition of, 568  
 sem\_open function, 579–580, 582, 584  
     definition of, 579  
 sem\_post function, 331, 581–582, 584  
     definition of, 582  
 sem\_t structure, 582  
 sem\_timedwait function, 451, 581–582  
     definition of, 581  
 sem\_trywait function, 581, 584  
 semun union, 567–568  
 SEM\_UNDO constant, 569–570, 580, 583  
 sem\_unlink function, 580–581, 584  
     definition of, 580  
 SEM\_VALUE\_MAX constant, 40, 43, 580  
 sem\_wait function, 451, 581–582, 584  
     definition of, 581  
 send function, 331, 451, 592, 610, 616, 626–627  
     definition of, 610  
 send\_err function, 642–644, 653, 656–657, 668–669, 897  
     definition of, 642, 644  
 send\_fd function, 642–645, 649, 653, 656–657, 669, 897  
     definition of, 642, 646, 649  
 sendmsg function, 331, 451, 611, 613, 644–646, 650, 670  
     definition of, 611  
 sendto function, 331, 451, 610–611, 620, 622–623  
     definition of, 610  
 S\_ERROR constant, 510  
 serv\_accept function, 636–638, 641, 648, 659, 665, 667–668, 897  
     definition of, 636, 638  
 servent structure, 599  
 Service Access Facility, *see* SAF  
 Service Management Facility, *see* SMF  
 serv\_listen function, 636–637, 659, 664–665, 667, 670, 897  
     definition of, 636–637  
 session, 295–296  
     ID, 233, 252, 296, 311, 463–464  
     leader, 295–297, 311, 318, 464–466, 469, 726–727, 742, 944  
     process group ID, 304  
 session structure, 310–311, 318, 464  
 set  
     descriptor, 503, 505, 532, 933  
     signal, 336, 344–345, 532, 933  
 SETALL constant, 568, 570  
 setasync function, definition of, 939  
 setbuf function, 146–147, 150, 171, 175, 247–248, 701, 930  
     definition of, 146  
 set\_cloexec function, 615, 617, 622, 896  
     definition of, 480  
 setegid function, 258  
     definition of, 258  
 setenv function, 212, 251, 442  
     definition of, 212  
 seteuid function, 258–260  
     definition of, 258  
 set\_f1 function, 86, 482–483, 498, 896, 934  
     definition of, 85  
 setgid function, 256, 258, 288, 331, 816  
     definition of, 256  
 setgrent function, 183–184, 442, 452  
     definition of, 183  
 set-group-ID, 98–99, 102, 107–108, 110, 129, 140, 233, 253, 317, 496, 546, 723  
     saved, 56, 98, 257  
 setgroups function, 184  
     definition of, 184  
 sethostent function, 452, 597  
     definition of, 597  
 sethostname function, 189  
 setitimer function, 317, 320, 322, 381  
 \_setjmp function, 355, 358  
 setjmp function, 197, 213, 215–219, 225, 340, 343, 355–356, 358, 381, 924  
     definition of, 215  
*<setjmp.h>* header, 27

**setkey** function, 442  
**setlogmask** function, 470–471  
  definition of, 470  
**setnetent** function, 452, 598  
  definition of, 598  
**setpgid** function, 294, 331  
  definition of, 294  
**setpriority** function, 277  
  definition of, 277  
**setprotoent** function, 452, 598  
  definition of, 598  
**setpwent** function, 180–181, 442, 452  
  definition of, 180  
**setregid** function, 257–258  
  definition of, 257  
**setreuid** function, 257  
  definition of, 257  
**setrlimit** function, 53, 220, 382  
  definition of, 220  
**setservent** function, 452, 599  
  definition of, 599  
**setsid** function, 294–295, 297, 310–311, 331, 464–467, 724, 727–728  
  definition of, 295  
**setsockopt** function, 331, 624–625, 651  
  definition of, 624  
**setspent** function, 182  
  definition of, 182  
**settimeofday** function, 190  
**setuid** function, 98, 256, 258, 260, 288, 331, 816  
  definition of, 256  
**set-user-ID**, 98–99, 102, 104, 107–108, 110, 129, 140, 182, 233, 253, 256–257, 259, 267, 317, 546, 585–586, 653, 924  
  saved, 56, 98, 256–260, 288, 337  
**setutxent** function, 442, 452  
**SETVAL** constant, 568, 570  
**setvbuf** function, 146–147, 150, 171, 175, 220, 552, 721, 936  
  definition of, 146  
**SGI** (Silicon Graphics, Inc.), 35  
**SGID**, *see* set-group-ID  
**SHA-1**, 181  
shadow passwords, 181–182, 196, 918  
<shadow.h> header, 186  
**S\_HANGUP** constant, 510  
Shannon, W. A., 525, 949  
shared  
  libraries, 206–207, 226, 753, 920, 947  
  memory, 534, 571–578  
sharing, file, 74–77, 231  
shell, *see* Bourne shell, Bourne-again shell, C shell, Debian Almquist shell, Korn shell, TENEX C shell  
**SHELL** environment variable, 211, 288, 737  
shell, job-control, 294, 299, 306–307, 325, 358, 377, 379, 734–735  
shell layers, 299  
shells, 3  
**S\_HIPRI** constant, 510  
**shmat** function, 559, 573–576  
  definition of, 574  
**shmatt\_t** data type, 572  
**shmctl** function, 558, 562, 573–575  
  definition of, 573  
**shmctl** function, 574  
  definition of, 574  
**shmget** function, 557–558, 572, 575  
  definition of, 572  
**shmid\_ds** structure, 572–574  
**SHMLBA** constant, 574  
**SHM\_LOCK** constant, 573  
**SHM\_RDONLY** constant, 574  
**SHM\_RND** constant, 574  
**SHRT\_MAX** constant, 37  
**SHRT\_MIN** constant, 37  
**shutdown** function, 331, 592–593, 612  
  definition of, 592  
**SHUT\_RD** constant, 592  
**SHUT\_RDWR** constant, 592  
**SHUT\_WR** constant, 592  
**SI\_ASYNCIO** constant, 353  
**S\_IFBLK** constant, 134  
**S\_IFCHR** constant, 134  
**S\_IFDIR** constant, 134  
**S\_IFIFO** constant, 134  
**S\_IFLNK** constant, 114, 134  
**S\_IFMT** constant, 97  
**S\_IFREG** constant, 134  
**S\_IFSOCK** constant, 134, 634  
**sig2str** function, 380–381  
  definition of, 380  
**SIG2STR\_MAX** constant, 380  
**SIGABRT** signal, 236, 240–241, 275, 313, 317–319, 365–367, 381, 924  
**sigaction** function, 59, 323, 326, 329–331, 333, 335–336, 349–355, 366, 370, 374, 376, 455, 468, 476, 478–479, 510, 621, 815, 939  
  definition of, 350  
**sigaction** structure, 350, 354–355, 366, 369, 374, 376, 379, 467, 476, 478, 621, 814  
**sigaddset** function, 331, 344–345, 348, 360, 362–363, 370, 374, 378, 456, 478–479, 701, 815, 933  
  definition of, 344–345  
**SIGALRM** signal, 313–314, 317, 330–332, 338–340, 342–343, 347, 354, 356–357, 364–365, 373–374, 621

**sigaltstack** function, 351  
**sig\_atomic\_t** data type, 59, 356–357, 361–363, 732  
**SIG\_BLOCK** constant, 346, 348, 360, 362–363, 370, 374, 454, 456, 477, 701, 815  
**SIGBUS** signal, 317, 352–353, 527, 530  
**SIGCANCEL** signal, 317  
**SIGCHLD** signal, 238, 288, 315, 317, 331–335, 351–353, 367–368, 370–371, 377, 471, 501, 546, 723, 923, 939  
    semantics, 332–335  
**SIGCLD** signal, 317, 332–336  
**SIGCONT** signal, 301, 309, 317, 337, 377, 379  
**sigdelset** function, 331, 344–345, 366, 374, 933  
    definition of, 344–345  
**SIG\_DFL** constant, 323, 333, 350–351, 366, 378–379, 476  
**sigemptyset** function, 331, 344, 348, 354–355, 360, 362–363, 369–370, 374, 378, 456, 467, 476, 478, 621, 701, 815, 933  
    definition of, 344  
**SIGEMT** signal, 317–318  
**SIG\_ERR** constant, 19, 324, 334, 340–343, 348, 354–356, 360–361, 363, 368, 550, 709, 711, 733  
**sigevent** structure, 512  
**SIGEV\_NONE** constant, 518  
**sigfillset** function, 331, 344, 366, 477, 933  
    definition of, 344  
**SIGFPE** signal, 18, 240–241, 317–318, 352–353  
**SIGFREEZE** signal, 317–318  
**sigfunc** data type, 354–355, 896  
**SIGHUP** signal, 308–309, 317–318, 468, 475–479, 546, 815, 830, 843  
**SIG\_IGN** constant, 323, 333, 350, 366, 369, 379, 467, 815  
**SIGILL** signal, 317–318, 351–353, 366  
**SIGINFO** signal, 317–318, 682, 689  
**siginfo** structure, 244, 283, 351–352, 376, 379, 381, 512  
**SIGINT** signal, 18–19, 300, 314, 317, 319–320, 340–341, 347, 359–361, 364–365, 367–370, 372, 455–457, 546, 679, 681, 685, 688–689, 701–702, 709, 930, 932  
**SIGIO** signal, 83, 317, 319, 501, 509–510, 627  
**SIGIOT** signal, 317, 319, 365  
**sigismember** function, 331, 344–345, 347–348, 933  
    definition of, 344–345  
**sigjmp\_buf** data type, 356  
**SIGJVM1** signal, 317  
**SIGJVM2** signal, 317  
**SIGKILL** signal, 272, 275, 315, 317, 319, 321, 323, 346, 380, 735  
**siglongjmp** function, 219, 331, 355–358, 365  
    definition of, 356  
**SIGLOST** signal, 317  
**SIGLWP** signal, 317, 319, 321  
**signal** function, 18–19, 59, 308, 323–326, 329–335, 339–343, 348–349, 354–356, 360–361, 363, 368, 378, 510, 550, 709, 711, 939  
    definition of, 323, 354  
**signal mask**, 336  
**signal set**, 336, 344–345, 532, 933  
**<signal.h>** header, 27, 240, 314, 324, 344–345, 380  
**signal\_intr** function, 330, 355, 364, 382, 508, 733, 896, 930  
    definition of, 355  
**signals**, 18–19, 313–382  
    blocking, 335  
    delivery, 335  
    generation, 335  
    generation, pseudo terminal, 741  
    job-control, 377–379  
    null, 314, 337  
    pending, 335  
    queueing, 336, 349, 376  
    reliable, 335–336  
    unreliable, 326–327  
**signal\_thread** function, 814, 830  
    definition of, 830  
**sigpause** function, 331  
**sigpending** function, 331, 335, 347–349  
    definition of, 347  
**SIGPIPE** signal, 314, 317, 319, 537, 550–551, 553, 556, 587, 611, 815, 936  
**SIGPOLL** signal, 317, 319, 501, 509–510  
**sigprocmask** function, 331, 336, 340, 344, 346–349, 360, 362–364, 366, 370, 374, 378–379, 453–454, 456, 701  
    definition of, 346  
**SIGPROF** signal, 317, 320  
**SIGPWR** signal, 317–318, 320  
**sigqueue** function, 222, 331, 353, 376–377  
    definition of, 376  
**SIGQUEUE\_MAX** constant, 40, 43, 376  
**SIGQUIT** signal, 300, 317, 320, 347–349, 361–362, 367, 370, 372, 456–457, 546, 681, 689, 702, 709  
**SIGRTMAX** constant, 376  
**SIGRTMIN** constant, 376  
**SIGSEGV** signal, 314, 317, 320, 332, 336, 352–353, 393, 527  
**sigset** function, 331, 333  
**sigsetjmp** function, 219, 355–358  
    definition of, 356  
**SIG\_SETMASK** constant, 346, 348–349, 360, 362–364, 366, 370, 374, 454, 456, 701

**sigset\_t** data type, 59, 336, 344, 347–348, 360–361, 363, 366, 369, 374, 378, 454–456, 701, 813  
**SIGSTKFLT** signal, 317, 320  
**SIGSTOP** signal, 315, 317, 320, 323, 346, 377  
**SIGSUSP** signal, 689  
**sigsuspend** function, 331, 340, 359–365, 374, 451  
  definition of, 359  
**SIGSYS** signal, 317, 320  
**SIGTERM** signal, 315, 317, 321, 325, 476–479, 709, 732–733, 742, 815, 830, 944  
**SIGTHAW** signal, 317, 321  
**SIGTHR** signal, 319  
**sigtimedwait** function, 451  
**SIGTRAP** signal, 317, 321, 351, 353  
**SIGTSTP** signal, 300, 308, 317, 320–321, 377–379, 680, 682, 701, 735  
**SIGTTIN** signal, 300–301, 304, 309, 317, 321, 377, 379  
**SIGTTOU** signal, 301–302, 317, 321, 377, 379, 691  
**SIG\_UNBLOCK** constant, 346, 349, 378, 454  
**SIGURG** signal, 83, 314, 317, 319, 322, 510–511, 626  
**SIGUSR1** signal, 317, 322, 324, 347, 356–358, 360–361, 363–364, 501  
**SIGUSR2** signal, 317, 322, 324, 363–364  
**sigval** structure, 352  
**SIGVTALRM** signal, 317, 322  
**sigwait** function, 451, 454–455, 457, 475, 477, 830  
  definition of, 454  
**sigwaitinfo** function, 451  
**SIGWAITING** signal, 317, 322  
**SIGWINCH** signal, 311, 317, 322, 710–712, 718–719, 741–742  
**SIGXCPU** signal, 221, 317, 322  
**SIGXFSZ** signal, 221, 317, 322, 382, 925  
**SIGXRES** signal, 317, 322  
Silicon Graphics, Inc., *see* SGI  
**SI\_MESGQ** constant, 353  
Singh, A., 112, 116, 952  
Single UNIX Specification, *see* SUS  
  Version 3, *see* SUSv3  
  Version 4, *see* SUSv4  
single-instance daemons, 473–474  
**S\_INPUT** constant, 510  
**SI\_OCSPGRP** constant, 627  
**SI\_QUEUE** constant, 353  
**S\_IRGRP** constant, 99, 104, 107, 140, 149, 473, 896  
**S\_IROTH** constant, 99, 104, 107, 140, 149, 473, 896  
**S\_IRUSR** constant, 99, 104, 107, 140, 149, 169, 473, 818, 896  
**S\_IWGXG** constant, 107, 639  
**S\_IWRCO** constant, 107, 639  
**S\_IWRCU** constant, 107, 584, 639  
**S\_ISBLK** function, 96–97, 139  
**S\_ISCHR** function, 96–97, 139, 698  
**S\_ISDIR** function, 96–97, 133, 698  
**S\_ISFIFO** function, 96–97, 535, 552  
**S\_ISGID** constant, 99, 107, 140, 498  
**S\_ISLINK** function, 96–97  
**S\_ISREG** function, 96, 808  
**S\_ISSOCK** function, 96–97, 639  
**S\_ISUID** constant, 99, 107, 140  
**S\_ISVTX** constant, 107–109, 140  
**SI\_TIMER** constant, 353  
**SI\_USER** constant, 353  
**S\_IWGRP** constant, 99, 104, 107, 140, 149  
**S\_IWOTH** constant, 99, 104, 107, 140, 149  
**S\_IWUSR** constant, 99, 104, 107, 140, 149, 169, 473, 818, 896  
**S\_IXGRP** constant, 99, 107, 140, 498, 896  
**S\_IXOTH** constant, 99, 107, 140, 896  
**S\_IXUSR** constant, 99, 107, 140, 169, 896  
size, file, 111–112  
size program, 206–207, 226  
sizeof operator, 231  
**size\_t** data type, 59–60, 71, 507, 772, 906  
**\_SLBF** constant, 166  
**sleep** function, 230, 234, 243, 246, 272, 274, 308, 331, 339–342, 348, 372–375, 381–382, 387, 391–392, 439, 451, 460, 504, 532, 606–607, 923, 925, 928, 931, 936  
  definition of, 373–374, 929  
**sleep** program, 372  
**sleep2** function, 924  
**sleep\_us** function, 532, 896  
  definition of, 933–934  
SMF (Service Management Facility), 293  
**S\_MSG** constant, 510  
**\_SNBF** constant, 165  
Snow Leopard, xxi  
**snprintf** function, 159, 901, 904  
  definition of, 159  
Snyder, G., 952  
**sockaddr** structure, 595–597, 605–607, 609, 622, 625, 635, 637, 639, 641, 800  
**sockaddr\_in** structure, 595–596, 603  
**sockaddr\_in6** structure, 595–596  
**sockaddr\_un** structure, 634–638, 640–642  
**socketmark** function, 331, 626  
  definition of, 626  
**SOCK\_DGRAM** constant, 590–591, 602, 608, 612, 621, 623, 632, 941  
**socket**  
  addressing, 593–605  
  descriptors, 590–593  
  I/O, asynchronous, 627

I/O, nonblocking, 608–609, 627  
 mechanism, 95, 534, 587, 589–628  
 options, 623–625  
**socket** function, 148, 331, 590, 592, 607, 609, 621,  
 625, 637–638, 640–641, 808  
 definition of, 590  
**socketpair** function, 148, 331, 629–630, 632,  
 634, 941  
 definition of, 630  
 sockets, UNIX domain, 629–642  
 timing, 565  
**socklen\_t** data type, 606–607, 609, 622, 625, 800  
**SOCK\_RAW** constant, 590–591, 602  
**SOCK\_SEQPACKET** constant, 590–591, 602, 605,  
 609, 612, 625  
**SOCK\_STREAM** constant, 319, 590–591, 602, 605,  
 609, 612, 614–616, 618–619, 625, 630, 635,  
 637, 640, 802, 808, 816  
 Solaris, xxi–xxii, xxv, xxvii, 3–4, 26–27, 29–30,  
 35–36, 38, 41, 48–49, 57–60, 62, 64–65, 70, 76,  
 88, 102, 108–113, 121–122, 129, 131–132, 138,  
 178, 182, 184–188, 208–209, 211–212, 222,  
 225, 229, 240, 242, 244–245, 260, 277, 288, 290,  
 293, 296, 298, 303, 314, 316–323, 329,  
 334–335, 351, 355, 371, 373, 377, 379–380,  
 385, 388, 392, 396, 409, 426–427, 432, 439, 471,  
 485, 496–497, 499, 503, 530–531, 534, 559,  
 561, 563, 565, 567, 572–573, 576, 592, 594,  
 607–608, 611–613, 627, 634, 648, 675–678,  
 684–691, 693, 700, 704, 716–717, 723–724,  
 726–727, 740–741, 744, 799, 911, 918, 925, 930,  
 932, 935–936, 951  
**SOL\_SOCKET** constant, 624–625, 645–646,  
 650–652  
 solutions to exercises, 905–945  
**SOMAXCONN** constant, 608  
**SO\_OOBINLINE** constant, 626  
**SO\_PASSCRED** constant, 651  
**SO\_REUSEADDR** constant, 625  
 source code, availability, xxx  
**S\_OUTPUT** constant, 510  
 Spafford, G., 181, 250, 298, 949  
**spawn** function, 234  
<**spawn.h**> header, 30  
 spin locks, 417–418  
 spooling, printer, 793–795  
**sprintf** function, 159, 549, 616, 622, 640, 655,  
 657, 659, 668–669, 759, 772–773, 803,  
 818–819, 822–823, 825–827, 833–835, 837,  
 845, 945  
 definition of, 159  
**spwd** structure, 918  
**squid** login name, 178  
**S\_RDBAND** constant, 510  
**S\_RDNORM** constant, 510  
**sscanf** function, 162, 549, 551, 802–803  
 definition of, 162  
**ssh** program, 293  
**sshd** program, 465  
**SSIZE\_MAX** constant, 38, 41, 71  
**ssize\_t** data type, 39, 59, 71  
 stack, 205, 215  
**stackaddr** attribute, 427  
**stacksize** attribute, 427  
 standard error, 8, 145, 617  
 standard error routines, 898–904  
 standard input, 8, 145  
 standard I/O  
 alternatives, 174–175  
 buffering, 145–147, 231, 235, 265, 367, 552, 721,  
 752  
 efficiency, 153–156  
 implementation, 164–167  
 library, 10, 143–175  
 streams, 143–144  
 versus unbuffered I/O, timing, 155  
 standard output, 8, 145, 617  
 standards, 25–33  
 differences, 58–59  
 START terminal character, 678, 680–682, 686, 689,  
 693  
**stat** function, 4, 7, 65, 93–95, 97, 99, 107,  
 121–122, 124, 126–128, 131, 138, 140–141,  
 170, 331, 452, 586, 592, 628, 639–640, 670, 698,  
 908, 910, 942  
 definition of, 93  
**stat** structure, 93–96, 98, 111, 114, 124, 140, 147,  
 167, 170, 498, 518, 529, 535, 552, 557, 586, 638,  
 697–698, 757, 807, 832  
 static variables, 219  
 STATUS terminal character, 678, 682, 687, 689, 703  
<**stdarg.h**> header, 27, 162–163, 755, 758  
<**stdbool.h**> header, 27  
\_\_**STDC\_IEC\_559\_\_** constant, 31  
<**stddef.h**> header, 27, 635  
**stderr** variable, 145, 483, 731, 901  
**STDERR\_FILENO** constant, 62, 145, 618–619, 643,  
 648, 652, 729  
**stdin** variable, 10, 145, 154, 214, 216, 550–551,  
 654  
**STDIN\_FILENO** constant, 9, 62, 67, 72, 145, 308,  
 378, 483, 539, 544, 549–550, 619, 655–656,  
 679, 684, 709, 711, 728, 730–732, 739–740  
<**stdint.h**> header, 27, 595  
<**stdio.h**> header, 10, 27, 38, 51, 145, 147, 151,  
 164, 168, 694, 755, 895

- <stdlib.h> header, 27, 208, 895
- stdout variable, 10, 145, 154, 247–248, 275, 901, 921, 930
- STDOUT\_FILENO constant, 9, 62, 72, 145, 230, 235, 378, 483, 537, 544, 549–550, 614, 618–620, 654–656, 729, 733, 739–740, 921
- Stevens, D. A., xxxii
- Stevens, E. M., xxxii
- Stevens, S. H., xxxii
- Stevens, W. R., xx, xxv–xxvi, xxxii, 157, 291, 470, 505, 589, 717, 793, 952–953
- sticky bit, 107–109, 117, 140
- stime function, 190
- Stonebraker, M. R., 743, 953
- STOP terminal character, 678, 680–682, 686, 689, 693
- str2sig function, 380
  - definition of, 380
- strace program, 497
- Strang, J., 712, 953
- strchr function, 767
- stream orientation, 144
- STREAM\_MAX constant, 38, 40, 43, 49
- STREAMS, xxii, 88, 143, 501–502, 506, 508, 510, 534, 560, 565, 648, 716–717, 722, 726, 740
- streams, memory, 171–174
- STREAMS module
  - ldterm, 716, 726
  - pckt, 716, 740
  - ptem, 716, 726
  - ttcompat, 716, 726
- streams, standard I/O, 143–144
- STREAMS-based pipes, mounted, 534
  - timing, 565
- strerror function, 15–16, 24, 380, 442, 452, 471, 474, 478–479, 600, 615–618, 621–622, 657, 669, 823–827, 830, 833–834, 842, 899, 901, 904, 906, 931
  - definition of, 15
- strerror\_r function, 443, 452
- strftime function, 190, 192–196, 264, 408, 452, 919
  - definition of, 192
- strftime\_1 function, 192
  - definition of, 192
- <string.h> header, 27, 895
- <strings.h> header, 29
- strip program, 920
- strlen function, 12, 231, 945
- strncasecmp function, 840
- strncpy function, 809
- Strong, H. R., 744, 750, 949
- <stropts.h> header, 508, 510
- strptime function, 195
  - definition of, 195
- strsignal function, 380, 830
  - definition of, 380
- strtok function, 442, 657–658
- strtok\_r function, 443
- strtol function, 278, 633
- stty program, 301, 691–692, 702, 713, 943
- Stumm, M., 174, 531, 951
- S\_TYPEISMQ function, 96
- S\_TYPEISSEM function, 96
- S\_TYPEISSHM function, 96
- su program, 472
- submit\_file function, 807, 809, 811
  - definition of, 809
- SUID, *see* set-user-ID
- Sun Microsystems, xxi–xxii, xxvii, 33, 35, 76, 740, 953
- SunOS, xxxi, 33, 206, 330, 354
- superuser, 16
- supplementary group ID, 18, 39, 98, 101, 108, 110, 183–184, 233, 252, 258
- SUS (Single UNIX Specification), xxi, xxvi, 28, 30–33, 36, 50, 53–54, 57–58, 60–61, 64, 69, 78, 88, 94, 105, 107, 109, 131, 136, 143, 157, 163, 168–169, 180, 183, 190–191, 196, 211–212, 220–221, 234, 239, 244–245, 262, 293, 296, 311, 315, 322, 330, 333, 352, 354, 410, 425, 429–431, 442, 469–472, 485, 496, 501, 507, 509, 521, 527–528, 533–534, 559, 561, 565–566, 572–573, 583, 596, 607, 610, 612, 623, 627, 645, 662, 674, 678, 683, 722–724, 744, 910, 950, 953
- SUSP terminal character, 678, 680, 682, 688, 701
- SUSv3 (Single UNIX Specification, Version 3), 32
- SUSv4 (Single UNIX Specification, Version 4), 32, 88, 132, 143, 153, 168–169, 189, 314, 319–320, 336, 375–376, 384, 442, 501, 509–510, 525, 533, 571, 579
- SVID (System V Interface Definition), xix, 32–33, 948
- SVR2, 65, 187, 317, 329, 336, 340–341, 712, 948
- SVR3, 76, 129, 201, 299, 313, 317, 319, 326, 329, 333, 336, 496, 502, 507, 898, 948
- SVR3.0, xxxi
- SVR3.1, xxxi
- SVR3.2, xxxi, 36, 81, 267
- SVR4, xxii, xxxi–xxxii, 3, 21, 33, 35–36, 48, 63, 65, 76, 121, 187, 209, 290, 296, 299, 310, 313, 317, 329, 333, 336, 469, 502, 507–508, 521, 712, 722, 744, 948–949, 953
- swapper process, 227
- S\_WRBAND constant, 510
- S\_WRNORM constant, 510

**symbolic link**, 55, 94–95, 110–111, 114, 118, 120–123, 131, 137, 141, 186, 908–909  
**symlink** function, 123–124, 331, 452  
  definition of, 123  
**symlinkat** function, 123–124, 331, 452  
  definition of, 123  
**SYMLINK\_MAX** constant, 39, 44, 49  
**SYMLOOP\_MAX** constant, 40, 43, 48–49  
**sync** function, 61, 81, 452  
  definition of, 81  
**sync** program, 81  
**synchronization mechanisms**, 86–87  
**synchronous write**, 63, 86–87  
**<sys/acct.h>** header, 269  
**sysconf** function, 20, 37, 39, 41–48, 50–54, 57, 59–60, 69, 98, 201, 221, 256, 276, 280–281, 384, 425–426, 429, 431, 442, 516, 527, 616, 618, 623, 800, 815, 907  
  definition of, 42  
**sysctl** program, 315, 559  
**sysdef** program, 559  
**<sys/disklabel.h>** header, 88  
**<sys/filio.h>** header, 88  
**<sys/ipc.h>** header, 30, 558  
**<sys/iso/signal\_iso.h>** header, 314  
**syslog** function, 452, 465, 468–472, 474–476, 478–480, 615–619, 622–623, 901, 904, 928  
  definition of, 470  
**syslogd** program, 470–471, 473, 475, 479–480  
**<syslog.h>** header, 30  
**<sys/mkdev.h>** header, 138  
**<sys/mman.h>** header, 29  
**<sys/msg.h>** header, 30  
**<sys/mtio.h>** header, 88  
**<sys/param.h>** header, 49, 51  
**<sys/resource.h>** header, 30  
**<sys/select.h>** header, 29, 501, 504, 932–933  
**<sys/sem.h>** header, 30, 568  
**<sys/shm.h>** header, 30  
**sys\_siglist** variable, 379  
**<sys signal.h>** header, 314  
**<sys/socket.h>** header, 29, 608  
**<sys/sockio.h>** header, 88  
**<sys/stat.h>** header, 29, 97  
**<sys/statvfs.h>** header, 29  
**<sys/sysmacros.h>** header, 138  
**system calls**, 1, 21  
  interrupted, 327–330, 343, 351, 354–355, 365, 508  
  restarted, 329–330, 342–343, 351, 354, 508, 700  
  tracing, 497  
  versus functions, 21–23  
**system** function, 23, 129, 227, 249, 264–269, 281–283, 349, 367–372, 381, 451, 538, 542, 923, 936  
  definition of, 265–266, 369  
  return value, 371  
**system identification**, 187–189  
**system process**, 228, 337  
**System V**, xxv, 87, 464, 466, 469, 475, 482, 485, 500–501, 506, 509–510, 722, 726  
**System V Interface Definition**, *see SVID*  
**<sys/time.h>** header, 30, 501  
**<sys/times.h>** header, 29  
**<sys/ttycom.h>** header, 88  
**<sys/types.h>** header, 29, 58, 138, 501, 557, 933  
**<sys/uio.h>** header, 30  
**<sys/un.h>** header, 29, 634  
**<sys/utsname.h>** header, 29  
**<sys/wait.h>** header, 29, 239

**TAB0** constant, 691  
**TAB1** constant, 691  
**TAB2** constant, 691  
**TAB3** constant, 690–691  
**TABDLY** constant, 676, 684, 689–691  
**Tankus, E.**, xxxii  
**tar** program, 127, 135, 142, 910–911  
**<tar.h>** header, 29  
**tcdrain** function, 322, 331, 451, 677, 693  
  definition of, 693  
**tcflag\_t** data type, 674  
**tcflow** function, 322, 331, 677, 693  
  definition of, 693  
**tcflush** function, 145, 322, 331, 673, 677, 693  
  definition of, 693  
**tcgetattr** function, 331, 674, 677, 679, 683–684, 691–692, 695, 701, 705–707, 722, 730–731  
  definition of, 683  
**tcgetpgrp** function, 298–299, 331, 674, 677  
  definition of, 298  
**tcgetsid** function, 298–299, 674, 677  
  definition of, 299  
**TCIFLUSH** constant, 693  
**TCIOFF** constant, 693  
**TCIOFLUSH** constant, 693  
**TCION** constant, 693  
**TCMalloc**, 210, 949  
**TCOFLUSH** constant, 693  
**TCOOFF** constant, 693  
**TCOON** constant, 693  
**TCSADRAIN** constant, 683  
**TCSAFLUSH** constant, 679, 683, 701, 705–707  
**TCSANOW** constant, 683–684, 728, 731

- tcsendbreak** function, 322, 331, 677, 682, 693–694  
 definition of, 693
- tcsetattr** function, 322, 331, 673–674, 677, 679, 683–684, 691–692, 701, 705–707, 722, 728, 731, 738  
 definition of, 683
- tcsetpgrp** function, 298–299, 301, 303, 322, 331, 674, 677  
 definition of, 298
- tee** program, 554–555
- tell** function, 67
- TELL\_CHILD** function, 247–248, 362, 491, 498, 532, 539, 541, 577, 898  
 definition of, 363, 540
- telldir** function, 130–135  
 definition of, 130
- TELL\_PARENT** function, 247, 362, 491, 532, 539, 541, 577, 898, 934  
 definition of, 363, 540
- TELL\_WAIT** function, 247–248, 362, 491, 498, 532, 539, 577, 898, 934  
 definition of, 363, 540
- telnet** program, 292–293, 500, 738–739, 742
- telnetd** program, 291–292, 500–501, 717, 734, 923, 944
- tempnam** function, 169
- TENEX C shell, 3
- TERM environment variable, 211, 287, 289
- termcap**, 712–713, 953
- terminal  
 baud rate, 692–693  
 canonical mode, 700–703  
 controlling, 63, 233, 252, 270, 292, 295–298, 301, 303–304, 306, 309, 311–312, 318, 321, 377, 463, 465–466, 469, 480, 680, 685, 691, 694, 700, 702, 716, 724, 726–727, 898, 953  
 identification, 694–700  
 I/O, 671–713  
 line control, 693–694  
 logins, 285–290  
 mode, cbreak, 672, 704, 708, 713  
 mode, cooked, 672  
 mode, raw, 672, 704, 708, 713, 732, 734  
 noncanonical mode, 703–710  
 options, 683–691  
 parity, 688  
 process group ID, 303, 463  
 special input characters, 678–682  
 window size, 311, 322, 710–712, 718, 727, 741–742  
 termination, process, 198–202
- terminfo**, 712–713, 949, 953
- termio** structure, 674
- <**termio.h**> header, 674
- termios** structure, 64, 311, 674, 677–679, 683–684, 692–693, 695, 701, 703–706, 708, 722, 727, 730–732, 738, 741–742, 897, 944
- <**termios.h**> header, 29, 88, 674
- text segment, 204
- <**tgmath.h**> header, 27
- Thompson, K., 75, 181, 229, 743, 951–953
- thread-fork interactions, 457–461
- thread\_init** function, 445
- threads, 14, 27, 229, 383–423, 578  
 cancellation options, 451–453  
 concepts, 383–385  
 control, 425–462  
 creation, 385–388  
 I/O, 461–462  
 reentrancy, 442–446  
 synchronization, 397–422  
 termination, 388–397
- thread-signal interactions, 453–457
- thread-specific data, 446–451
- thundering herd, 927
- tick, clock, 20, 42–43, 49, 59, 270, 280
- time  
 and date functions, 189–196  
 calendar, 20, 24, 59, 126, 189, 191–192, 264, 270  
 process, 20, 24, 59, 280–282  
 values, 20
- time** program, 20
- TIME** terminal value, 687, 703–704, 708, 713, 943
- time** function, 189–190, 194, 264, 331, 357, 639–640, 919, 929  
 definition of, 189
- <**time.h**> header, 27, 59
- timeout** function, 439, 462
- TIMER\_ABSTIME** constant, 375
- timer\_getoverrun** function, 331
- timer\_gettime** function, 331
- TIMER\_MAX** constant, 40, 43
- timer\_settime** function, 331, 353
- times, file, 124–125, 532
- times** function, 42, 59, 280–281, 331, 522  
 definition of, 280
- timespec** structure, 94, 126, 128, 189–190, 375, 407–408, 413–414, 437–438, 506, 832
- time\_t** data type, 20, 59, 94, 189, 192, 196, 906
- timeval** structure, 190, 278, 414, 421, 437, 503, 506, 805–806, 929, 933
- timing  
 full-duplex pipes, 565  
 message queues, 565  
 read buffer sizes, 73

- read/write versus mmap, 530
- standard I/O versus unbuffered I/O, 155
- STREAMS-based pipes, 565
- synchronization mechanisms, 86–87
- UNIX domain sockets, 565
  - `writev` versus other techniques, 522
- timing comparison, mutex, 571
  - record locking, 571
  - semaphore locking, 571, 583
- `TIOCGWINSZ` constant, 710–711, 719, 730, 897
- `TIOCpkt` constant, 740
- `TIOCREMOTE` constant, 741
- `TIOCSCTTY` constant, 297–298, 727–728
- `TIOCSIG` constant, 741
- `TIOCSIGNAL` constant, 741
- `TIOCSWINSZ` constant, 710, 718, 728, 741
- `tip` program, 713
- `tm` structure, 191, 194, 408, 919
- `TMPDIR` environment variable, 211
- `tmpfile` function, 167–171, 366, 452
  - definition of, 167
- `TMP_MAX` constant, 38, 168
- `tmpnam` function, 38, 167–171, 442
  - definition of, 167
- `tms` structure, 280–281
- `TOCTTOU` error, 65, 250, 953
- Torvalds, L., 35
- `TOSTOP` constant, 676, 691
- `touch` program, 127
- tracing system calls, 497
- transactions, database, 952
- `TRAP_BRKPT` constant, 353
- `TRAP_TRACE` constant, 353
- `tread` function, 800, 805–806, 825, 838–839
  - definition of, 805
- `treadn` function, 800, 806, 824
  - definition of, 806
- Trickey, H., 229, 952
- `truncate` function, 112, 121, 125, 474
  - definition of, 112
- truncation
  - file, 112
  - filename, 65–66
  - pathname, 65–66
- `truss` program, 497
- `ttcompat` STREAMS module, 716, 726
- `tty` structure, 311
- `tty_atexit` function, 705, 731, 897
  - definition of, 708
- `tty_cbreak` function, 704, 709, 897
  - definition of, 705
- `ttymon` program, 290
- `ttynname` function, 137, 276, 442, 452, 695–696, 699
  - definition of, 695, 698
  - `TTY_NAME_MAX` constant, 40, 43, 49
  - `ttynname_r` function, 443, 452
  - `tty_raw` function, 704, 709, 713, 731, 897
    - definition of, 706
  - `tty_reset` function, 704, 709, 897
    - definition of, 707
  - `tty_termios` function, 705, 897
    - definition of, 708
  - `type` attribute, 431
  - `typescript` file, 719, 737
  - `TZ` environment variable, 190, 192, 195–196, 211, 919
  - `TZNAME_MAX` constant, 40, 43, 49
  - `tzset` function, 452
  - 
  - Ubuntu, xxii, 7, 26, 35, 290
  - `UCHAR_MAX` constant, 37–38
  - `ucontext_t` structure, 352
  - `ucred` structure, 649, 651
  - UFS file system, 49, 57, 65, 113, 116, 129
  - UID, *see* user ID
  - `uid_t` data type, 59
  - `uint16_t` data type, 595
  - `uint32_t` data type, 595
  - `UINT_MAX` constant, 37–38
  - `ulimit` program, 53, 222
  - `ULLONG_MAX` constant, 37
  - `ULONG_MAX` constant, 37
  - UltraSPARC, xxii, xxvii
  - `umask` function, 104–107, 222, 331, 466–467
    - definition of, 104
  - `umask` program, 105, 141
  - `uname` function, 187, 196, 331
    - definition of, 187
  - `uname` program, 188, 196
  - unbuffered I/O, 8, 61–91
  - unbuffered I/O timing, standard I/O versus, 155
  - `ungetc` function, 151–152, 452
    - definition of, 151
  - `ungetwc` function, 452
  - uninitialized data segment, 205
  - `<unistd.h>` header, 9, 29, 53, 62, 110, 442, 501, 755, 895
  - UNIX Architecture, 1–2
  - UNIX domain sockets, 629–642
    - timing, 565
  - UNIX System implementations, 33
  - Unix-to-Unix Copy, *see* UUCP
  - UnixWare, 35

- unlink function, 114, 116–119, 121–122, 125, 141, 169–170, 331, 366, 452, 497, 553, 637, 639, 641, 823, 826–827, 837, 909, 911, 937, 942  
 definition of, 117
- unlinkat function, 116–119, 331, 452  
 definition of, 117
- un\_lock function, 489, 759–760, 762, 768, 770–771, 773, 777–778, 780, 897
- unlockpt function, 723–725  
 definition of, 723
- Unrau, R., 174, 531, 951
- unreliable signals, 326–327
- unsetenv function, 212, 442  
 definition of, 212
- update program, 81
- update\_jobno function, 814, 819, 832, 843  
 definition of, 819
- Upstart, 290
- uptime program, 614–615, 617, 619–620, 622–623, 628  
 $\_USE\_BSD$  constant, 473
- USER environment variable, 210, 288
- user ID, 16, 255–260  
 effective, 98–99, 101–102, 106, 110, 126, 140, 228, 233, 253, 256–260, 276, 286, 288, 337, 381, 558, 562, 568, 573, 586–587, 637, 640, 809, 918  
 real, 39–40, 43, 98–99, 102, 221, 228, 233, 252–253, 256–260, 270, 276, 286, 288, 337, 381, 585, 924
- USHRT\_MAX constant, 37
- usleep function, 532, 934
- UTC (Coordinated Universal Time), 20, 189, 192, 196
- utime function, 331
- UTIME\_NOW constant, 126
- utimensat function, 125–128, 331, 452, 910  
 definition of, 126
- UTIME OMIT constant, 126–127
- utimes function, 125–128, 141, 331, 452, 910  
 definition of, 127
- utmp file, 186–187, 276, 312, 734, 923, 930
- utmp structure, 187
- utmpx file, 187  
 $\<utmpx.h>$  header, 30
- utsname structure, 187–188, 196
- UUCP (Unix-to-Unix Copy), 188
- uucp program, 500
- V7, 329, 726
- va\_arg function, 758
- va\_end function, 758, 899–903
- va\_list data type, 758, 899–903
- /var/account/acct file, 269
- /var/adm/pacct file, 269
- <varargs.h> header, 162
- variables  
 automatic, 205, 215, 217, 219, 226  
 global, 219  
 register, 217  
 static, 219  
 volatile, 217, 219, 340, 357
- /var/log/account/pacct file, 269
- /var/log/wtmp file, 187
- /var/run/utmp file, 187
- va\_start function, 758, 899–903
- VDISCARD constant, 678
- vdprintf function, 161, 452  
 definition of, 161
- VDSUSP constant, 678
- VEOF constant, 678–679, 704
- VEOL constant, 678, 704
- VEOL2 constant, 678
- VERASE constant, 678
- VERASE2 constant, 678
- vfork function, 229, 234–236, 283, 921–922
- vfprintf function, 161, 452  
 definition of, 161
- vfscanf function, 163  
 definition of, 163
- vfwprintf function, 452
- vi program, 377, 497, 499, 672, 711–713, 943
- VINTR constant, 678–679
- viqw program, 179
- VKILL constant, 678
- VLNEXT constant, 678
- VMIN constant, 703–705, 707
- v-node, 74–76, 78, 136, 312, 493–494, 642, 907, 950
- vnode structure, 311–312
- Vo, K. P., 135, 174, 949–950, 953
- volatile variables, 217, 219, 340, 357
- vprintf function, 161, 452  
 definition of, 161
- vQUIT constant, 678
- vread function, 525
- VREPRINT constant, 678
- vscanf function, 163  
 definition of, 163
- vsnprintf function, 161, 901  
 definition of, 161
- vsprintf function, 161, 471  
 definition of, 161
- vsscanf function, 163  
 definition of, 163
- vSTART constant, 678

**VSTATUS** constant, 678  
**VSTOP** constant, 678  
**VSUSP** constant, 678  
**vsyslog** function, 472  
     definition of, 472  
**VT0** constant, 691  
**VT1** constant, 691  
**VTDLY** constant, 676, 684, 689, 691  
**VTIME** constant, 703–705, 707  
**VWERASE** constant, 678  
**vwprintf** function, 452  
**vwwrite** function, 525

**wait** function, 231–232, 237–246, 249, 255, 264,  
     267, 280, 282–283, 301, 317, 328–329, 331,  
     333–335, 351, 368, 371–372, 451, 471, 499,  
     546, 588, 936  
     definition of, 238  
**Wait**, J. W., xxxii  
**wait3** function, 245  
     definition of, 245  
**wait4** function, 245  
     definition of, 245  
**WAIT\_CHILD** function, 247, 362, 491, 532, 539, 577,  
     898, 934  
     definition of, 363, 540  
**waitid** function, 244–245, 283, 451  
     definition of, 244  
**WAIT\_PARENT** function, 247–248, 362, 491, 498,  
     532, 539, 577, 898  
     definition of, 363, 540  
**waitpid** function, 11–13, 19, 23, 237–245, 254,  
     261, 265–267, 282, 285, 294, 301, 315, 329, 331,  
     370–371, 451, 498, 538, 545–546, 587–588,  
     618, 935, 937, 939  
     definition of, 238  
**wall** program, 723  
**wc** program, 112  
**<wchar.h>** header, 27, 144  
**wchar\_t** data type, 59  
**WCONTINUED** constant, 242, 244  
**WCOREDUMP** function, 239–240  
**wctomb** function, 442  
**wcsftime** function, 452  
**wcsrtombs** function, 442  
**wstombs** function, 442  
**wtomb** function, 442  
**<wctype.h>** header, 27  
**Weeks**, M. S., 206, 949  
**Wei**, J., 65, 953  
**Weinberger**, P. J., 76, 262, 743, 947, 953  
**Weinstock**, C. B., 953

**WERASE** terminal character, 678, 682, 685–687,  
     703  
**WEXITED** constant, 244  
**WEXITSTATUS** function, 239–240  
**who** program, 187, 734  
**WIFCONTINUED** function, 239  
**WIFEXITED** function, 239–240  
**WIFSIGNALED** function, 239–240  
**WIFSTOPPED** function, 239–240, 242  
**Williams**, T., 310, 953  
**Wilson**, G. A., xxxii  
**window size**  
     pseudo terminal, 741  
     terminal, 311, 322, 710–712, 718, 727, 741–742  
**winsize** structure, 311, 710–711, 727, 730, 732,  
     742, 897, 944  
**Winterbottom**, P., 229, 952  
**WNCHANG** constant, 242, 244  
**WNOWAIT** constant, 242, 244  
**W\_OK** constant, 102  
**Wolff**, R., xxxii  
**Wolff**, S., xxxii  
**WORD\_BIT** constant, 38  
**wordexp** function, 452  
**<wordexp.h>** header, 29  
**worker\_thread** structure, 812–813, 828–829  
**working directory**, *see* current directory  
**worm**, Internet, 153  
**wprintf** function, 452  
**Wright**, G. R., xxxii  
**write**  
     delayed, 81  
     gather, 521, 644  
     synchronous, 63, 86–87  
**write** program, 723  
**write** function, 8–10, 20–21, 59, 61, 63–64,  
     68–69, 72, 77–79, 86–88, 90, 125, 145–146,  
     156, 167, 174, 230–231, 234, 247, 328–329,  
     331, 342–343, 378, 382, 451, 474, 482–484,  
     491, 495–498, 502, 505, 509, 513, 517,  
     522–526, 530–532, 537–538, 540, 549–551,  
     553, 555, 560, 565, 587, 590, 592, 610, 614, 620,  
     643, 654–655, 672, 752, 760, 773, 810, 819, 826,  
     836, 907–908, 921, 925, 934, 936–937, 945  
     definition of, 72  
**write\_lock** function, 489, 493, 498, 818, 897  
**writen** function, 523–524, 644, 732–733, 738,  
     810–811, 824–827, 836, 896  
     definition of, 523–524  
**writev** function, 41, 43, 329, 451, 481, 521–523,  
     531–532, 592, 611, 644, 655, 660, 752, 771, 773,  
     832, 836  
     definition of, 521

`writew_lock` function, 489, 491, 759, 763, 769,  
771–772, 777, 787, 897

`wscanf` function, 452

`WSTOPPED` constant, 244

`WSTOPSIG` function, 239–240

`WTERMSIG` function, 239–240

`wtmp` file, 186–187, 312, 923

Wulf, W. A., 953

`WUNTRACED` constant, 242

x86, xxi

`xargs` program, 252

`XCASE` constant, 691

Xenix, 33, 485, 726

`xinetd` program, 293

`X_OK` constant, 102

X/Open, xxvi, 31, 953

X/Open Curses, 32

X/Open Portability Guide, 31–32

Issue 3, *see XPG3*

Issue 4, *see XPG4*

`_XOPEN_CRYPT` constant, 31, 57

`_XOPEN_IOV_MAX` constant, 41

`_XOPEN_NAME_MAX` constant, 41

`_XOPEN_PATH_MAX` constant, 41

`_XOPEN_REALTIME` constant, 31, 57

`_XOPEN_REALTIME_THREADS` constant, 31, 57

`_XOPEN_SHM` constant, 57

`_XOPEN_SOURCE` constant, 57–58

`_XOPEN_UNIX` constant, 30–31, 57

`_XOPEN_VERSION` constant, 57

XPG3 (X/Open Portability Guide, Issue 3), xxxi,

33, 953

XPG4 (X/Open Portability Guide, Issue 4), 32, 54

XSI, 30–31, 53–54, 57, 94, 107, 109, 131–132, 143,

161, 163, 168–169, 180, 183, 211–212, 220, 222,

239, 242, 244–245, 252, 257, 276, 293, 315, 317,

322, 329, 333, 350–352, 377, 429, 431, 442,

469–472, 485, 521, 526, 528, 534, 553,

562–563, 566, 571, 576, 578, 587–588, 666,

676, 685, 687, 689–691, 722, 724, 744, 910

XSI IPC, 556–560

`XTABS` constant, 690–691

Yigit, O., 744, 952

zombie, 237–238, 242, 283, 333, 351, 923

*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.

#### About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

# informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram  
IBM Press | Que | Prentice Hall | Sams  
SAFARI BOOKS ONLINE



THE TRUSTED TECHNOLOGY LEARNING SOURCE



**InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

▲Addison-Wesley

Cisco Press

EXAM/CRAM

IBM  
Press.

QUE'

PRENTICE  
HALL

SAMS

| Safari<sup>®</sup>

## LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters.  
Visit [informit.com/newsletters](http://informit.com/newsletters).
- Access FREE podcasts from experts at [informit.com/podcasts](http://informit.com/podcasts).
- Read the latest author articles and sample chapters at [informit.com/articles](http://informit.com/articles).
- Access thousands of books and videos in the Safari Books Online digital library at [safari.informit.com](http://safari.informit.com).
- Get tips from expert blogs at [informit.com/blogs](http://informit.com/blogs).

Visit [informit.com/learn](http://informit.com/learn) to discover all the ways you can access the hottest technology content.

### Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit [informit.com/socialconnect](http://informit.com/socialconnect).



THE TRUSTED TECHNOLOGY LEARNING SOURCE



▲Addison-Wesley

Cisco Press

EXAM/CRAM

IBM  
Press.

QUE'

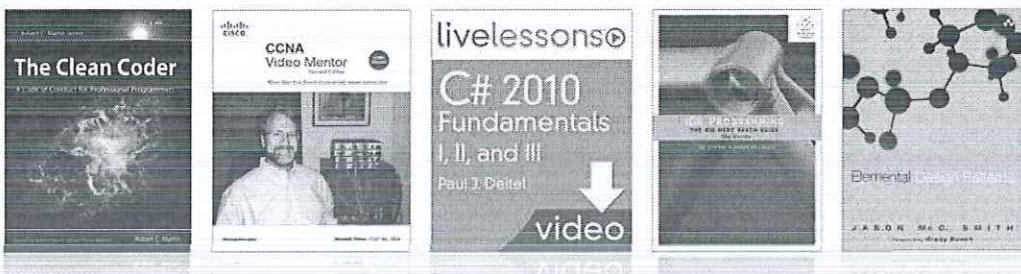
PRENTICE  
HALL

SAMS

| Safari<sup>®</sup>

## Try Safari Books Online FREE for 15 days

Get online access to Thousands of Books and Videos



**Safari**  
Books Online

FREE 15-DAY TRIAL + 15% OFF\*

[informit.com/safaritrial](http://informit.com/safaritrial)

### ► Feed your brain

Gain unlimited access to thousands of books and videos about technology, digital media and professional development from O'Reilly Media, Addison-Wesley, Microsoft Press, Cisco Press, McGraw Hill, Wiley, Wrox, Prentice Hall, Que, Sams, Apress, Adobe Press and other top publishers.

### ► See it, believe it

Watch hundreds of expert-led instructional videos on today's hottest topics.

## WAIT, THERE'S MORE!

### ► Gain a competitive edge

Be first to learn about the newest technologies and subjects with Rough Cuts pre-published manuscripts and new technology overviews in Short Cuts.

### ► Accelerate your project

Copy and paste code, create smart searches that let you know when new books about your favorite topics are available, and customize your library with favorites, highlights, tags, notes, mash-ups and more.

\* Available to new subscribers only. Discount applies to the Safari Library and is valid for first 12 consecutive monthly billing cycles. Safari Library is not available in all countries.



AdobePress

Cisco Press



IBM  
Press



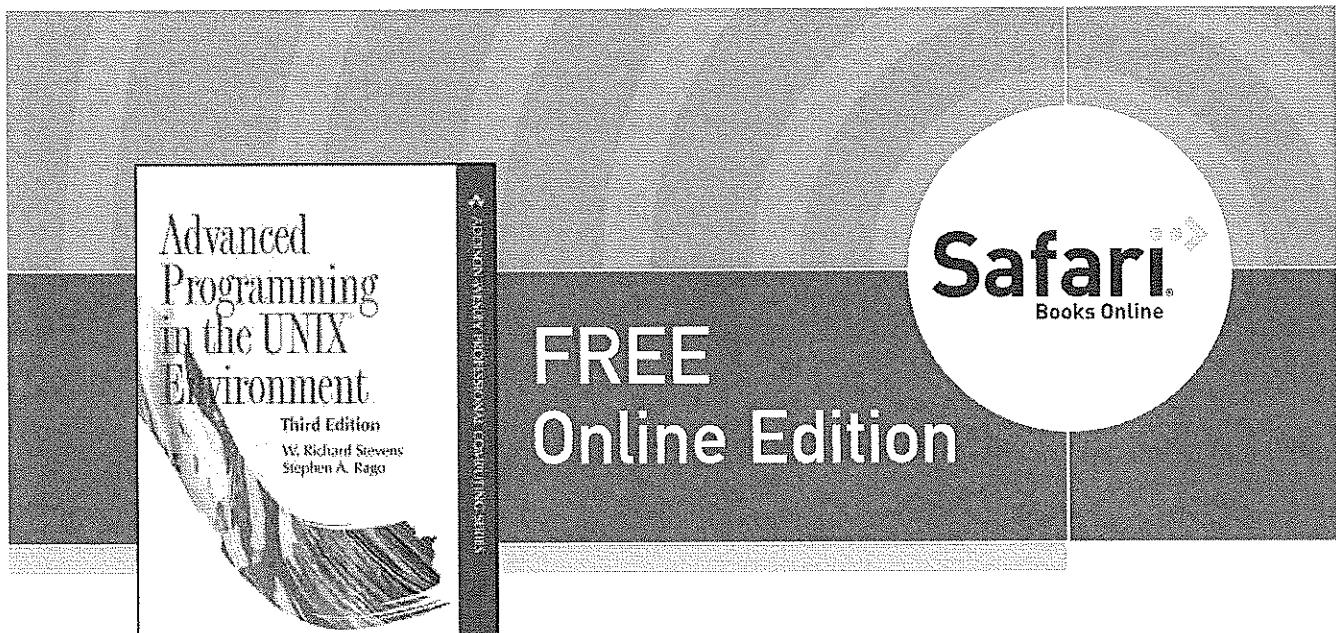
PEARSON  
IT Certification



SAMS

VMware PRESS





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](http://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](mailto:customer-service@safaribooksonline.com)



