Chapter 12

System calls and library functions

This section contains the following system calls and library functions, listed in alphabetical order:

accept	fchown	ioctl	pipe
alarm	fcntl	kill	read
bind	fork	lchown	readdir
bzero	fstat	link	setegid
chdir	ftruncate	listen	seteuid
chmod	getegid	lseek	setgid
chown	geteuid	lstat	setpgid
close	getgid	memset	setuid
closedir	gethostbyname	mkdir	signal
connect	gethostname	mkfifo	socket
dup	getpgid	mknod	stat
dup2	getpid	nice	sync
execl	getppid	ntohl	truncate
execlp	getuid	ntohs	unlink
execv	htonl	open	wait
execvp	htons	opendir	write
exit	inet_addr	pause	
fchmod	inet_ntoa	perror	

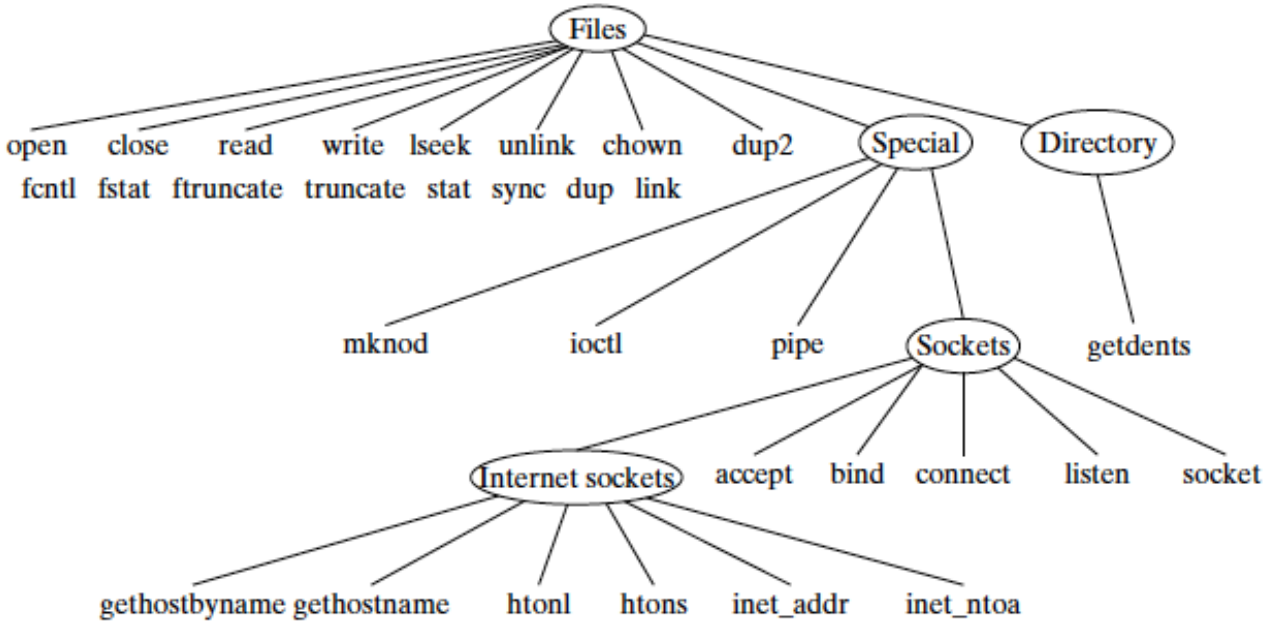

Figure 12–1 File management system call hierarchy.

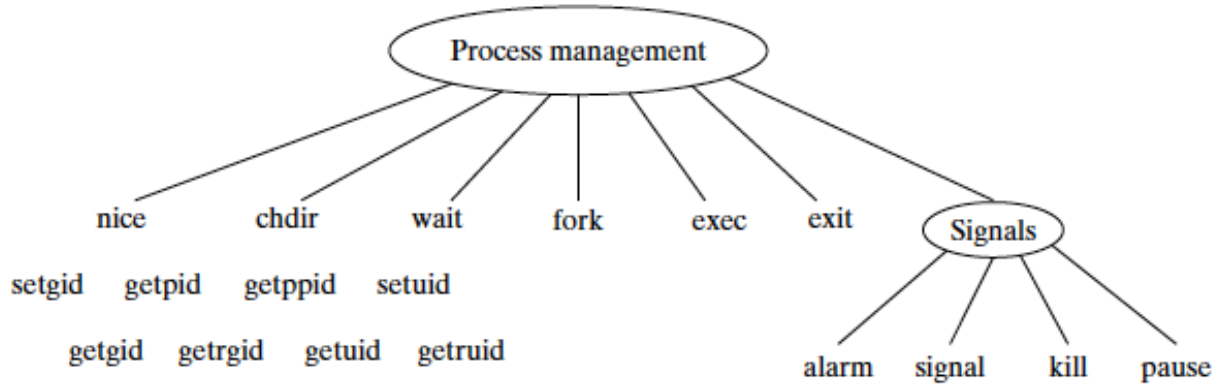

Figure 12–2 Process management system call hierarchy.

Figure 12–3 Error handling hierarchy.

Library Function: void **perror** (char* *str*)

perror () displays the string *str*, followed by a colon, followed by a description of the last system call error. If there is no error to report, it displays the string "Error 0." Actually, perror () isn't a system call—it's a standard C library function.

Figure 12–4 Description of the perror () system call.

```
#define     EPERM       1  /* Operation not permitted */
#define     ENOENT      2  /* No such file or directory */
#define     ESRCH       3  /* No such process */
#define     EINTR       4  /* Interrupted system call */
#define     EIO         5  /* I/O error */
```

Value	Meaning
0	standard input (stdin)
1	standard output (stdout)
2	standard error (stderr)

Figure 12–5 File descriptor values for standard I/O channels.

Name	Function
open	Opens/creates a file.
read	Reads bytes from a file into a buffer.
write	Writes bytes from a buffer to a file.
lseek	Moves to a particular offset in a file.
close	Closes a file.
unlink	Removes a file.

Figure 12–7 Linux system calls for basic I/O operations.

System Call: int **open** (char* *fileName*, int *mode* [, int *permissions*])

open () allows you to open or create a file for reading and/or writing. *fileName* is an absolute or relative pathname and *mode* is a bitwise or'ing of a read/write flag together with zero or more miscellaneous flags. *permissions* is a number that encodes the value of the file's permission flags, and should only be supplied when a file is being created. It is usually written using the octal encoding scheme described in Chapter 3, "GNU Utilities for Nonprogrammers." The *permissions* value is affected by the process's umask value, described in Chapter 5, "The Linux Shells." The values of the predefined read/write and miscellaneous flags are defined in "/usr/include/fcntl.h". The read/write flags are as follows:

FLAG	MEANING
O_RDONLY	Open for read-only.
O_WRONLY	Open for write-only.
O_RDWR	Open for read and write.

The miscellaneous flags are as follows:

FLAG	MEANING
O_APPEND	Position the file pointer at the end of the file before each write ().
O_CREAT	If the file doesn't exist, create the file, and set the owner ID to the process's effective user ID. The umask value is used when determining the initial permission flag settings.
O_EXCL	If O_CREAT is set and the file exists, then open () fails.
O_NONBLOCK or O_NDELAY	This setting works only for named pipes. If set, an open for read-only will return immediately, regardless of whether the write end is open, and an open for write-only will fail if the read end isn't open. If clear, an open for read-only or write-only will block until the other end is also open.
O_TRUNC	If the file exists, it is truncated to length zero.

open () returns a non-negative file descriptor if successful; otherwise, it returns -1.

Figure 12–10 Description of the open () system call.

System Call: ssize_t **read** (int *fd*, void* *buf*, size_t *count*)

[*Note*: This synopsis describes how read () operates when reading a regular file. For information on reading from special files, please refer to later sections of this chapter.]

read () copies *count* bytes from the file referenced by the file descriptor *fd* into the buffer *buf*. The bytes are read from the current file position, which is then updated accordingly.

read () copies as many bytes from the file as it can, up to the number specified by *count*, and returns the number of bytes actually copied. If a read () is attempted after the last byte has already been read, it returns 0, which indicates end-of-file.

If successful, read () returns the number of bytes that it read; otherwise, it returns -1.

Figure 12–11 Description of the read () system call.

System Call: ssize_t **write** (int *fd*, void* *buf*, size_t *count*)

[*Note*: This synopsis describes how write () operates when writing to a regular file. For information on writing to special files, please refer to later sections of this chapter.]

write () copies *count* bytes from a buffer *buf* to the file referenced by the file descriptor *fd*. The bytes are written at the current file position, which is then updated accordingly. If the O_APPEND flag was set for *fd*, the file position is set to the end of the file before each write.

write () copies as many bytes from the buffer as it can, up to the number specified by *count*, and returns the number of bytes actually copied. Your process should always check the return value. If the return value isn't *count*, then the disk probably filled up and no space was left.

If successful, write () returns the number of bytes that were written; otherwise, it returns -1.

Figure 12–12 Description of the write () system call.

System Call: off_t **lseek** (int *fd*, off_t *offset*, int *mode*)

lseek () allows you to change a descriptor's current file position. *fd* is the file descriptor, offset is a long integer, and mode describes how *offset* should be interpreted. The three possible values of mode are defined in "/usr/include/stdio.h," and have the following meaning:

VALUE	MEANING
SEEK_SET	*offset* is relative to the start of the file.
SEEK_CUR	*offset* is relative to the current file position.
SEEK_END	*offset* is relative to the end of the file.

lseek () fails if you try to move before the start of the file.

If successful, lseek () returns the current file position; otherwise, it returns -1.

Figure 12–13 Description of the lseek () system call.

System Call: int **close** (int *fd*)

close () frees the file descriptor *fd*. If *fd* is the last file descriptor associated with a particular open file, the kernel resources associated with the file are deallocated. When a process terminates, all of its file descriptors are automatically closed, but it's better programming practice to close a file when you're done with it. If you close a file descriptor that's already closed, an error occurs.

If successful, close () returns zero; otherwise, it returns -1.

Figure 12–14 Description of the close () system call.

System Call: int **unlink** (const char* *fileName*)

unlink () removes the hard link from the name *fileName* to its file. If *fileName* is the last link to the file, the file's resources are deallocated. In this case, if any process's file descriptors are currently associated with the file, the directory entry is removed immediately but the file is only deallocated after all of the file descriptors are closed. This means that an executable file can unlink itself during execution and still continue to completion.

If successful, unlink () returns zero; otherwise, it returns -1.

Figure 12–15 Description of the unlink () system call.

Name	Function
stat	Obtains status information about a file.
fstat	Works just like stat.
readdir	Obtains directory entries.

Figure 12–16 Advanced Linux I/O system calls.

System Call: int **stat** (const char* *name*, struct stat* *buf*)
int **lstat** (const char* *name*, struct stat* *buf*)
int **fstat** (int *fd*, struct stat* *buf*)

stat () fills the buffer *buf* with information about the file *name*. The **stat** structure is defined in "/usr/include/sys/stat.h". lstat () returns information about a symbolic link itself rather than the file it references. fstat () performs the same function as stat (), except that it takes the file descriptor of the file to be stat'ed as its first parameter. The **stat** structure contains the following members:

NAME	MEANING
st_dev	the device number
st_ino	the inode number
st_mode	the permission flags
st_nlink	the hard link count
st_uid	the user ID
st_gid	the group ID
st_size	the file size
st_atime	the last access time
st_mtime	the last modification time
st_ctime	the last status change time

There are some predefined macros defined in "/usr/include/sys/stat.h" that take **st_mode** as their argument and return true (1) for the following file types:

MACRO	RETURNS TRUE FOR FILE TYPE
S_ISDIR	directory
S_ISCHR	character-oriented special device
S_ISBLK	block-oriented special device
S_ISREG	regular file
S_ISFIFO	pipe

The time fields may be decoded using the standard C library asctime () and localtime () subroutines.

stat () and fstat () return 0 if successful and -1 otherwise.

Figure 12–18 Description of the stat () system call.

Library Function: DIR * **opendir** (char * *fileName*)
struct dirent * **readdir** (DIR **dir*)
int **closedir** (DIR **dir*)

opendir () opens a directory file for reading and returns a pointer to a stream descriptor which is used as the argument to readdir () and closedir (). readdir () returns a pointer to a **dirent** structure containing information about the next directory entry each time it is called. closedir () is used to close the directory. The **dirent** structure is defined in the system header file "/usr/include/dirent.h"

NAME	MEANING
d_ino	the inode number
d_off	the offset of the next directory entry
d_reclen	the length of the directory entry structure
d_name	the filename

opendir () returns the directory stream pointer when successful, NULL when not successful. readdir () returns 1 when a directory entry has been successfully read, 0 when the last directory entry has already been read, and -1 in the case of an error. closedir () returns 0 on success, -1 on failure.

Figure 12–19 Description of the opendir (), readdir (), and closedir () library functions.

Name	Function
chown	Changes a file's owner and/or group.
chmod	Changes a file's permission settings.
dup	Duplicates a file descriptor.
dup2	Similar to dup.
fchown	Works just like chown.
fchmod	Works just like chmod.
fcntl	Gives access to miscellaneous file characteristics.
ftruncate	Works just like truncate.
ioctl	Controls a device.
link	Creates a hard link.
mknod	Creates a special file.
sync	Schedules all file buffers to be flushed to disk.
truncate	Truncates a file.

Figure 12–20 Linux file management system calls.

System Call: int **chown** (const char* *fileName*, uid_t *ownerId*, gid_t *groupId*)
int **lchown** (const char* *fileName*, uid_t *ownerId*, gid_t *groupId*)
int **fchown** (int *fd*, uid_t *ownerId*, gid_t *groupId*)

chown () causes the owner and group IDs of *fileName* to be changed to *ownerId* and *groupId*, respectively. A value of -1 in a particular field means that its associated value should remain unchanged.

Only a super-user can change the ownership of a file, and a user may change the group only to another group that he/she is a member of. If *fileName* is a symbolic link, the owner and group of the link are changed instead of the file that the link is referencing.

fchown () is just like chown () except that it takes an open descriptor as an argument instead of a filename.

lchown () changes the ownership of a symbolic link itself rather than the file the link references.

They both return -1 if unsuccessful, and 0 otherwise.

Figure 12–21 Description of the chown () system call.

System Call: int **chmod** (const char* *fileName*, int *mode*)
int **fchmod** (int *fd*, mode_t *mode*);

chmod () changes the mode of *fileName* to *mode*, where *mode* is usually supplied as an octal number as described in Chapter 3, "GNU Utilities for Nonprogrammers." The "set user ID" and "set group ID" flags have the octal values 4000 and 2000, respectively. To change a file's mode, you must either own it or be a super-user.

fchmod () works just like chmod () except that it takes an open file descriptor as an argument instead of a filename.

They both return -1 if unsuccessful, and 0 otherwise.

Figure 12–22 Description of the chmod () system call.

System Call: int **fcntl** (int *fd*, int *cmd*, int *arg*)

fcntl () performs the operation encoded by *cmd* on the file associated with the file descriptor *fd*. *arg* is an optional argument for *cmd*. Here are the most common values of *cmd*:

VALUE	OPERATION
F_SETFD	Set the close-on-exec flag to the lowest bit of *arg* (0 or 1).
F_GETFD	Return a number whose lowest bit is 1 if the close-on-exec flag is set, and 0 otherwise.
F_GETFL	Return a number corresponding to the current file status flags and access modes.
F_SETFL	Set the current file status flags to *arg*.
F_GETOWN	Return the process ID or process group that is currently set to receive SIGIO/SIGURG signals. If the returned value is positive, it refers to a process ID. If it's negative, its absolute value refers to a process group.
F_SETOWN	Set the process ID or process group that should receive SIGIO/SIGURG signals to *arg*. The encoding scheme is as described for F_GETOWN.

fcntl () returns -1 if unsuccessful.

Figure 12–24 Description of the fcntl () system call.

System Call: int **ioctl** (int *fd*, int *cmd*, int *arg*)

ioctl () performs the operation encoded by *cmd* on the file associated with the file descriptor *fd*. *arg* is an optional argument for *cmd*. The valid values of *cmd* depend on the device that *fd* refers to, and are typically documented in the manufacturer's operating instructions. I therefore supply no examples for this system call.

ioctl () returns -1 if unsuccessful.

Figure 12–25 Description of the ioctl () system call.

System Call: int **link** (const char* *oldPath*, const char* *newPath*)

link () creates a new label *newPath* and links it to the same file as the label *oldPath*. The hard link count of the associated file is incremented by one. If *oldPath* and *newPath* reside on different physical devices, a hard link cannot be made and link () fails. For more information about hard links, consult the description of **ln** in Chapter 4, "GNU Utilities for Power Users."

link () returns -1 if unsuccessful, and 0 otherwise.

Figure 12–26 Description of the link () system call.

System Call: int **mknod** (const char* *fileName*, mode_t *type*, dev_t *device*)
int **mkdir** (const char* *fileName*, mode_t *mode*)

mknod () creates a new regular, directory, or special file called *fileName* whose type can be one of the following:

VALUE	MEANING
S_IFDIR	directory
S_IFCHR	character-oriented special file
S_IFBLK	block-oriented special file
S_IFREG	regular file
S_IFIFO	named pipe

Only a super-user can use mknod () to create directories or special files.

mkdir () creates a directory with permission setting created by a logical AND of *mode* with the process's current umask setting.

It is now typical to use the mkdir () system call to create directories and mkfifo () (see below) to make named pipes rather than mknod ().

Both mknod () and mkdir () return -1 if unsuccessful, and 0 otherwise.

Figure 12–27 Description of the mknod () and mkdir () system calls.

Library Function: int **mkfifo** (const char* *fileName*, mode_t *mode*)

mkfifo () creates a named pipe called *fileName* with permission setting created by a logical AND of *mode* with the process's current umask setting.

Figure 12–28 Description of the mkfifo () library function.

System Call: void **sync** ()

sync () schedules all of the file system buffers to be written to disk. For more information on the buffer system, consult Chapter 13, "Linux Internals." sync () should be performed by any programs that bypass the file system buffers and examine the raw file system.

sync () always succeeds.

Figure 12–29 Description of the sync () system call.

System Call: int **truncate** (const char* *fileName*, off_t *length*)
int **ftruncate** (int *fd*, off_t *length*)

truncate () sets the length of the file *fileName* to be *length* bytes. If the file is longer than *length*, it is truncated. If it is shorter than *length*, it is padded with ASCII nulls.

ftruncate () works just like truncate () except that it takes an open file descriptor as an argument instead of a filename.

They both return -1 if unsuccessful, and 0 otherwise.

Figure 12–30 Description of the truncate () system call.

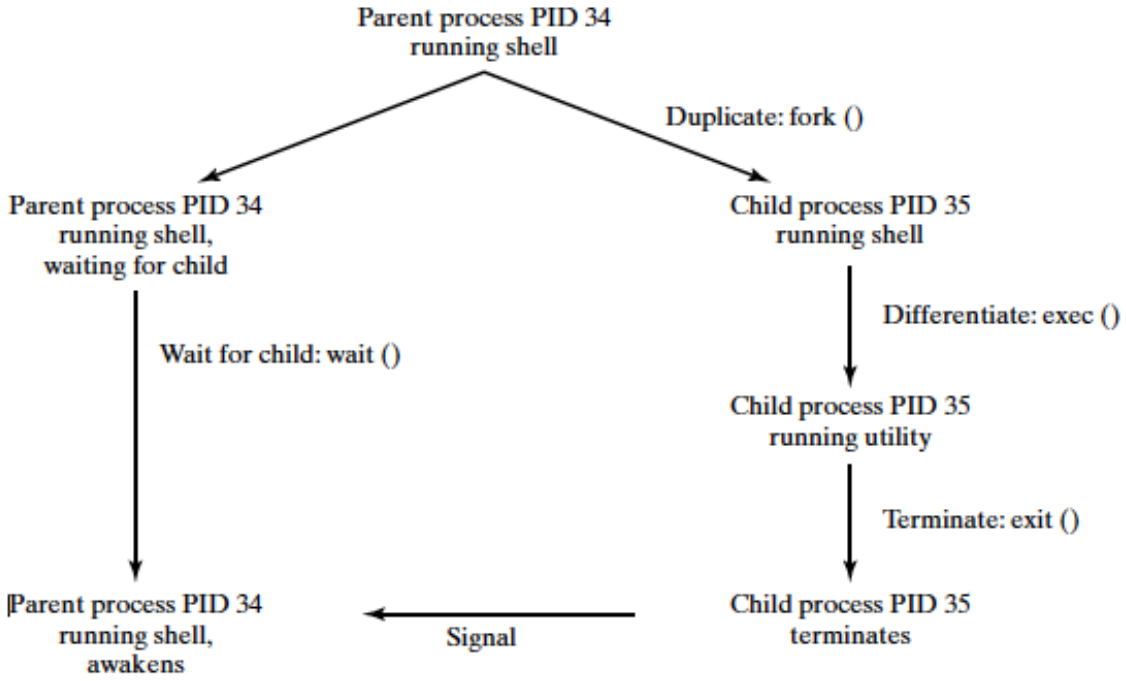

Figure 12–31 How a shell runs a utility.

System Call: pid_t **fork** (void)

fork () causes a process to duplicate. The child process is an almost-exact duplicate of the original parent process; it inherits a copy of its parent's code, data, stack, open file descriptors, and signal table. However, the parent and child have different process ID numbers and parent process ID numbers.

If fork () succeeds, it returns the PID of the child to the parent process, and returns 0 to the child process. If it fails, it returns -1 to the parent process, and no child is created.

Figure 12–33 Description of the fork () system call.

System Call: pid_t **fork** (void)

fork () causes a process to duplicate. The child process is an almost-exact duplicate of the original parent process; it inherits a copy of its parent's code, data, stack, open file descriptors, and signal table. However, the parent and child have different process ID numbers and parent process ID numbers.

If fork () succeeds, it returns the PID of the child to the parent process, and returns 0 to the child process. If it fails, it returns -1 to the parent process, and no child is created.

Figure 12–33 Description of the fork () system call.

System Call: pid_t **getpid** (void)
pid_t **getppid** (void)

getpid () and getppid () return a process's ID and parent process's ID numbers, respectively. They always succeed. The parent process ID number of PID 1 is 1.

Figure 12–34 Description of the getpid () and getppid () system calls.

```
$ cat myfork.c                  ...list the program.
#include <stdio.h>
main ()
{
 int pid;
 printf ("I'm the original process with PID %d and PPID %d.\n",
         getpid (), getppid ());
 pid = fork (); /* Duplicate. Child and parent continue from here */
 if (pid != 0) /* pid is non-zero, so I must be the parent */
   {
     printf ("I'm the parent process with PID %d and PPID %d.\n",
              getpid (), getppid ());
     printf ("My child's PID is %d\n", pid);
   }
 else /* pid is zero, so I must be the child */
   {
     printf ("I'm the child process with PID %d and PPID %d.\n",
              getpid (), getppid ());
   }
 printf ("PID %d terminates.\n", getpid () ); /* Both processes
execute this */
}
$ ./myfork                      ...run the program.
I'm the original process with PID 13292 and PPID 13273.
I'm the parent process with PID 13292 and PPID 13273.
My child's PID is 13293.
I'm the child process with PID 13293 and PPID 13292.
PID 13293 terminates.           ...child terminates.
PID 13292 terminates.           ...parent terminates.
$ _
```

```
main ()
{
 int pid;
  printf ("I'm the original process with PID %d and PPID %d.\n",
         getpid (), getppid ());
 pid = fork (); /* Duplicate. Child and parent continue from here */
 if (pid != 0) /* Branch based on return value from fork () */
   {
     /* pid is nonzero, so I must be the parent */
     printf ("I'm the parent process with PID %d and PPID %d.\n",
              getpid (), getppid ());
     printf ("My child's PID is %d\n", pid);
   }
 else
   {
     /* pid is zero, so I must be the child */
     sleep (5); /* Make sure that the parent terminates first */
     printf ("I'm the child process with PID %d and PPID %d.\n",
              getpid (), getppid ());
   }
  printf ("PID %d terminates.\n", getpid () ); /* Both processes
execute this */
}
$ ./orphan                        ...run the program.
I'm the original process with PID 13364 and PPID 13346.
I'm the parent process with PID 13364 and PPID 13346.
PID 13364 terminates.
I'm the child process with PID 13365 and PPID 1.    ...orphaned!
PID 13365 terminates.
$ _
```

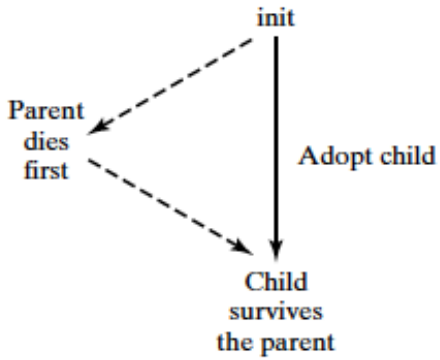

Figure 12–35 Process adoption.

System Call: pid_t **wait** (int* *status*)

wait () causes a process to suspend until one of its children terminates. A successful call to wait () returns the pid of the child that terminated and places a status code into *status* that is encoded as follows:

If the rightmost byte of status is zero, the leftmost byte contains the low eight bits of the value returned by the child's call to exit () or return ().

If the rightmost byte is nonzero, the rightmost seven bits are equal to the number of the signal that caused the child to terminate, and the remaining bit of the rightmost byte is set to 1 if the child produced a core dump.

If a process executes a wait () and has no children, wait () returns immediately with -1. If a process executes a wait () and one or more of its children are already zombies, wait () returns immediately with the status of one of the zombies.

Figure 12–37 Description of the wait () system call.

System Call: void **exit** (int *status*)

exit () closes all of a process's file descriptors, deallocates its code, data, and stack, and then terminates the process. When a child process terminates, it sends its parent a SIGCHLD signal and waits for its termination code *status* to be accepted. Only the lower eight bits of *status* are used, so values are limited to 0–255. A process that is waiting for its parent to accept its return code is called a *zombie* process. A parent accepts a child's termination code by executing wait (), which is described shortly.

The kernel ensures that all of a terminating process's children are orphaned and adopted by "init" by setting their PPID to 1. The "init" process always accepts its children's termination codes.

exit () never returns.

Figure 12–36 Description of the exit () system call.

Library Function: int **execl** (const char* *path*, const char* *arg0*, const char* *arg1*, ..., const char* *argn*, NULL)

int **execv** (const char* *path*, const char* *argv*[ ])

int **execlp** (const char* *path*, const char* *arg0*, const char* *arg1*,..., const char* *argn*, NULL)

int **execvp** (const char* *path*, const char* *argv*[ ])

The exec family of library functions replace the calling process's code, data, and stack from the executable whose pathname is stored in *path*.

execvl () is identical to execlp (), and execv () is identical to execvp (), except that execl () and execv () require the absolute or relative pathname of the executable file to be supplied, whereas execlp () and execvp () use the $PATH environment variable to find *path*.

If the executable is not found, the system call returns -1; otherwise, the calling process replaces its code, data, and stack from the executable and starts to execute the new code. A successful call to any of the exec system calls never returns.

execl () and execlp () invoke the executable with the string arguments pointed to by *arg1*..*argn*. *arg0* must be the name of the executable file itself, and the list of arguments must be null terminated.

execv () and execvp () invoke the executable with the string arguments pointed to by *argv*[1]..*argv*[n], where *argv*[n+1] is NULL. *argv*[0] must be the name of the executable file itself.

Figure 12–38 Description of the exec family of library functions.

System Call: int **chdir** (const char* *pathname*)

chdir () sets a process's current working directory to the directory *pathname*. The process must have execute permission from the directory to succeed.

chdir () returns 0 if successful; otherwise, it returns -1.

Figure 12–39 Description of the chdir () system call.

Library Function: int **nice** (int *delta*)

nice () adds *delta* to a process's current priority value. Only a super-user may specify a *delta* that leads to a negative priority value. Legal priority values lie between -20 and +19. If a *delta* is specified that takes a priority value beyond a limit, the priority value is truncated to the limit.

If nice () succeeds, it returns the new nice value; otherwise it returns -1. Note that this can cause problems, since a nice value of -1 is legal.

Figure 12–40 Description of the nice () library function.

System Call: uid_t **getuid** ()
uid_t **geteuid** ()
gid_t **getgid** ()
gid_t **getegid** ()

getuid () and geteuid () return the calling process's real and effective user ID, respectively. getgid () and getegid () return the calling process's real and effective group ID, respectively. The ID numbers correspond to the user and group IDs listed in the "/etc/passwd" and "/etc/group" files.

These calls always succeed.

Figure 12–41 Description of the getuid (), geteuid (), getgid (), and getegid () system calls.

System Call: int **setuid** (uid_t *id*)
int **seteuid** (uid_t *id*)
int **setgid** (gid_t *id*)
int **setegid** (gid_t *id*)

seteuid () and setegid () set the calling process's effective user and group ID, respectively. setuid () and setgid () set the calling process's effective and real user and group ID, respectively, to the specified value.

These calls succeed only if executed by a super-user, or if *id* is the real or effective user (group) ID of the calling process. They return 0 if successful; otherwise, they return -1.

Figure 12–42 Description of the setuid (), seteuid (), setgid (), and setegid () system calls.

Macro	#	Default action	Description
SIGHUP	1	quit	Hangup or death of controlling process.
SIGINT	2	quit	Keyboard interrupt.
SIGQUIT	3	core	Quit.
SIGILL	4	core	Illegal instruction.
SIGABRT	6	core	Abort.
SIGFPE	8	core	Arithmetic exception.
SIGKILL	9	quit	Kill (cannot be caught, blocked, or ignored).
SIGUSR1	10	quit	User-defined signal.
SIGSEGV	11	core	Segmentation violation (out of range address).
SIGUSR2	12	quit	User-defined signal.
SIGPIPE	13	quit	Write on a pipe or other socket with no one to read it.
SIGALRM	14	quit	Alarm clock.
SIGTERM	15	quit	Software termination signal (default signal sent by *kill*).
SIGCHLD	17	ignore	Status of child process has changed.
SIGCONT	18	none	Continue if stopped.
SIGSTOP	19	stop	Stop (suspend) the process.
SIGTSTP	20	stop	Stop from the keyboard.
SIGTTIN	21	stop	Background read from tty device.
SIGTTOU	22	stop	Background write to tty device.

Figure 12–44 POSIX signals.

Macro	#	Default action	Description
SIGHUP	1	quit	Hangup or death of controlling process.
SIGINT	2	quit	Keyboard interrupt.
SIGQUIT	3	core	Quit.
SIGILL	4	core	Illegal instruction.
SIGABRT	6	core	Abort.
SIGFPE	8	core	Arithmetic exception.
SIGKILL	9	quit	Kill (cannot be caught, blocked, or ignored).
SIGUSR1	10	quit	User-defined signal.
SIGSEGV	11	core	Segmentation violation (out of range address).
SIGUSR2	12	quit	User-defined signal.
SIGPIPE	13	quit	Write on a pipe or other socket with no one to read it.
SIGALRM	14	quit	Alarm clock.
SIGTERM	15	quit	Software termination signal (default signal sent by *kill*).
SIGCHLD	17	ignore	Status of child process has changed.
SIGCONT	18	none	Continue if stopped.
SIGSTOP	19	stop	Stop (suspend) the process.
SIGTSTP	20	stop	Stop from the keyboard.
SIGTTIN	21	stop	Background read from tty device.
SIGTTOU	22	stop	Background write to tty device.

Figure 12–44 POSIX signals.

System Call: void (***signal** (int *sigCode*, void (**func*)(int))) (int)

signal () allows a process to specify the action that it will take when a particular signal is received. The parameter *sigCode* specifies the number of the signal that is to be reprogrammed, and *func* may be one of several values:

- SIG_IGN, which indicates that the specified signal should be ignored and discarded.
- SIG_DFL, which indicates that the kernel's default handler should be used.
- an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives.

The valid signal numbers are included from "/usr/include/signal.h" (and the other header files that includes, the actual signal definitions are in "/usr/include/asm/signal.h" on my Linux machine). The signals SIGKILL and SIGSTP may not be reprogrammed. A child process inherits the signal settings from its parent during a fork (). When a process performs an exec (), previously ignored signals remain ignored but installed handlers are set back to the default handler.

With the exception of SIGCHLD, signals are not stacked. This means that if a process is sleeping and three identical signals are sent to it, only one of the signals is actually processed.

signal () returns the previous *func* value associated with *sigCode* if successful; otherwise it returns -1.

Figure 12–46 Description of the signal () system call.

System Call: int **kill** (pid_t *pid*, int *sigCode*)

kill () sends the signal with value *sigCode* to the process with PID *pid*. kill () succeeds and the signal is sent as long as at least one of the following conditions is satisfied:

- The sending process and the receiving process have the same owner.
- The sending process is owned by a super-user.

There are a few variations on the way that kill () works:

- If *pid* is 0, the signal is sent to all of the processes in the sender's process group.
- If *pid* is -1 and the sender is owned by a super-user, the signal is sent to all processes, including the sender.
- If *pid* is -1 and the sender is not a super-user, the signal is sent to all of the processes owned by the same owner as the sender, excluding the sending process.
- If the *pid* is negative and not -1, the signal is sent to all of the processes in the process group.

Process groups are discussed later in this chapter. If kill () manages to send at least one signal successfully, it returns 0; otherwise, it returns -1.

Figure 12–48 Description of the kill () system call.