The shell creates a new process each time it starts up a program in response to a command. For example, the command line:

```
$ cat file1 file2
```

results in the shell creating a process especially to run the `cat` command. The slightly more complex command line:

```
$ ls | wc -l
```

results in two processes being created to run the commands `ls` and `wc` concurrently. (In addition, the output of the directory listing program `ls` is **piped** into the word count program `wc`.)

ecause a process corresponds to an execution of a program, processes should not be confused with the programs they run; indeed, several processes can concurrently run the same program. For example, several users may be running the same editor program simultaneously, each invocation of the one program counting as a separate process.

Any UNIX process may in turn start other processes. This gives the UNIX process environment a hierarchical structure paralleling the directory tree of the file system. At the top of the process tree is a single controlling process, an execution of an extremely important program called `init`, which is ultimately the ancestor of all system and user processes.

For the programmer, UNIX provides a handful of system calls for process creation and manipulation. Excluding the various facilities for inter-process communication, the most important of these are:

`fork`	Used to create a new process by duplicating the calling process. `fork` is the basic process creation primitive.
`exec`	A family of library routines and one system call, each of which performs the same underlying function: the transformation of a process by overlaying its memory space with a new program. The differences between `exec` calls lie mainly in the way their argument lists are constructed.
`wait`	This call provides rudimentary process synchronization. It allows one process to wait until another related process finishes.
`exit`	Used to terminate a process.

In the rest of this chapter we will discuss the UNIX process in general and these four important calls in particular.

5.2 Creating processes

5.2.1 The `fork` system call

The fundamental process creation primitive is the `fork` system call. It is the mechanism which transforms UNIX into a multitasking system.

Usage

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

A successful call to `fork` causes the kernel to create a new process which is a (more or less) exact duplicate of the calling process. In other words the new process runs a copy of the same program as its creator, the variables within this having the same values as those within the calling process, with a single important exception, which we shall discuss shortly.

The newly created process is described as the **child process**, the one that called `fork` in the first place is called, not unnaturally, its **parent**.

After the call the parent process and its newly created offspring execute concurrently, both processes resuming execution at the statement immediately after the call to `fork`.

For those used to a purely sequential programming environment, the idea of `fork` can be a little difficult at first. Figure 5.1 demonstrates the notion more clearly. The figure centres around three lines of code, consisting of a call to `printf`, followed by a call to `fork` and then another call to `printf`.

There are two sections to the figure, *Before* and *After*. The *Before* section shows things prior to the invocation of `fork`. There is a single process labelled *A* (we are using the label *A* purely for convenience; it means nothing to the system). The arrow labelled *PC* (for program counter) shows the statement currently being executed. Since it points to the first `printf`, the rather trivial message `One` is displayed on standard output.

The *After* section shows the situation immediately following the call to `fork`. There are now two processes *A* and *B* running together. Process *A* is the same as in the *Before* part of the figure. *B* is the new process spawned by the call to `fork`. With one major exception, the value of `pid`, it is a copy of *A* and is executing the same program as *A*, hence the duplication of the three lines of source code in the figure. Using the terminology we introduced above, *A* is the parent process while *B* is its child.

The two *PC* arrows in this part of the figure show that the next statement executed by both parent and child after the `fork` invocation is a call to `printf`. In other words, both *A* and *B* pick up at the same point in the program code, even though *B* is new to the system. The message `Two` is therefore displayed twice.

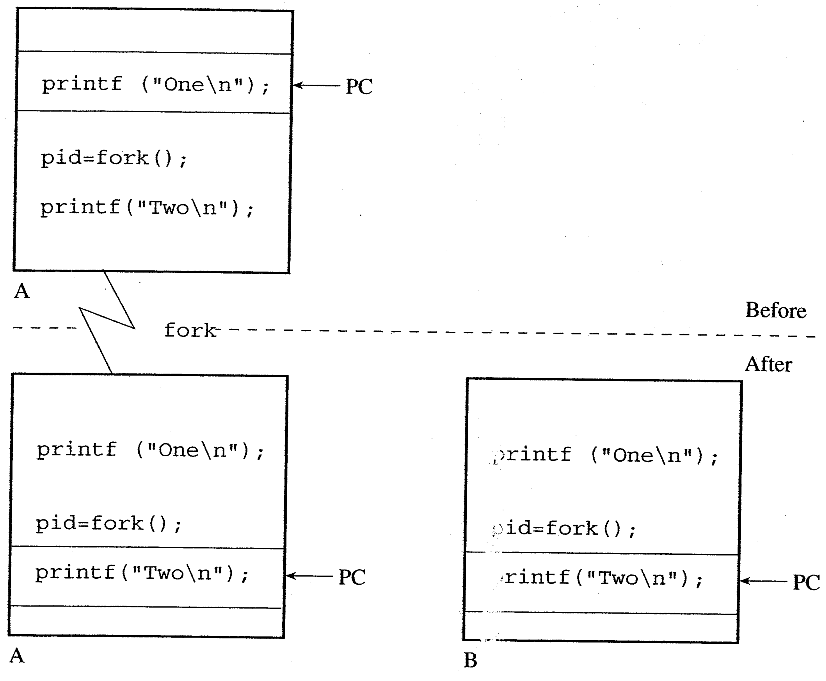

Figure 5.1 *The* `fork` *call.*

Process-ids

As you can see from the usage description at the beginning of this section, `fork` is called without arguments and returns a `pid_t`. The special `pid_t` type is defined in `<sys/types.h>` and is typically integer. An example call would be:

```
#include <unistd.h>     /* includes pid_t definition */

pid_t pid;
  .
  .
  .
pid = fork();
```

It is the value of `pid` that distinguishes child and parent. In the parent, `pid` is set to a non-zero, positive number. In the child it is set to zero. Because the return value in parent and child differ, the programmer is able to specify different actions for the two processes.

The number returned to the parent in `pid` is called the **process-id** of the child process. It is this number that identifies the new process to the system, rather like a user-id identifies a user. Since all processes are born through a call to `fork`, every

UNIX process has its own process-id, which at any one time is unique to that process. A process-id can therefore best be thought of as the identifying signature of its process.

The following short program demonstrates the action of `fork`, and the use of the process-id, perhaps a little more clearly:

```
/* spawn -- demonstrates fork */
#include <unistd.h>

main()
{
 pid_t pid;    /* hold process-id in parent */

 printf("Just one process so far\n");
 printf("Calling fork...\n");

 pid = fork();  /* create new process */

 if(pid == 0)
       printf("I'm the child\n");
 else if(pid > 0)
       printf("I'm the parent, child has pid %d\n", pid);
 else
       printf("Fork returned error code, no child\n");
}
```

There are three branches to the `if` statement after the `fork`. The first specifies the child process, corresponding to a zero value for the variable `pid`. The second gives the action for the parent process, corresponding to a positive value for `pid`. The third branch deals, implicitly, with a negative value (in fact −1) for `pid`, which can arise when `fork` has failed to create a child. This can indicate that the calling process has tried to breach one of two limits; the first is a system-wide limit on the number of processes; the second restricts the number of processes an individual user may run simultaneously. In both circumstances, the error variable `errno` contains the error code `EAGAIN`. Also notice that, since the two processes generated by the example program will run concurrently and without synchronization, there is no guarantee that the output from parent and child will not become confusingly intermingled.

Before we move on, it is worth discussing why `fork` is a useful call, since it may seem a little pointless in isolation. The essential point is that `fork` becomes valuable when combined with other UNIX facilities. For example, it is possible to have a child and parent process perform different but related tasks, cooperating by using one of the UNIX inter-process communication mechanisms such as signals or pipes (described in later chapters). Another facility often used in combination with `fork` is the `exec` system call, which we shall discuss in the next section, and which enables other programs to be executed.

Exercise 5.1 A program can call `fork` several times. Similarly, each child process can use `fork` to spawn children of its own. To prove this, write a program which creates two subprocesses. Each of these should then create one subprocess of its own. After each fork, each parent process should use printf to display the process-ids of its offspring.

5.3 Running new programs with exec

5.3.1 The exec family

If fork was the only process creation primitive available to the programmer, UNIX would be a little boring since only copies of the same program could be created. Thankfully, a member of the exec family can be used to initiate the execution of a new program. Figure 5.2 shows the family tree for the exec calls. The main difference between them is the way parameters can be passed. As you can see from the figure, all of these functions eventually make a call to execve – the real system call.

The following usage description shows most of the members of the family. (We will return to execle and execve shortly.)

Usage

```
#include <unistd.h>

/* The execl family of calls must be given the arguments as
   a NULL terminated list */

/* execl must be given a valid pathname for the executable */
int execl(const char *path,
          const char *arg0, ..., const char *argn,
          (char *)0);

/* execlp only needs the filename of the executable */
int execlp(const char *file,
          const char *arg0, ..., const char *argn,
          (char *)0);

/* The execv family of calls must be given an array of arguments */

/* execv must be given a valid pathname for the executable */
int execv(const char *path, char *const argv[]);

/* execvp only needs the filename of the executable */
int execvp(const char *file, char *const argv[]);
```

All varieties of exec perform the same function: they transform the calling process by loading a new program into its memory space. If the exec is successful the calling program is completely overlaid by the new program, which is then

$$\begin{tikzcd}
\texttt{execl} \arrow[d] & \texttt{execle} \arrow[dd] & \texttt{execlp} \arrow[d] \\
\texttt{execv} \arrow[dr] & & \texttt{execvp} \arrow[dl] \\
& \texttt{execve} &
\end{tikzcd}$$

Figure 5.2 *The* exec *family tree.*

started from its beginning. The result can be regarded as a new process, but one that retains the same process-id as the calling process.

It is important to stress that exec does not create a new subprocess to run concurrently with the calling process. Instead the old program is obliterated by the new. So, unlike fork there is no return from a successful call to exec.

To simplify matters, we spotlight just one of the exec calls, namely execl.

All parameters for execl are character pointers. The first, path in the usage description, gives the name of the file containing the program to be executed; with execl this must be a valid pathname, absolute or relative. The file itself must also contain a true program or a shell script with execute permission. (The system tells whether a file contains a program by looking at its first two bytes or so. If these contain a special value, called a **magic number**, then the system treats the file as a program.) The second parameter arg0 is, by convention, the name of the program or command stripped of any preceding pathname element. This and the remaining variable number of arguments (arg1 to argn) are available to the invoked program, corresponding to command line arguments within the shell. Indeed, the shell itself invokes commands by using one of the exec calls in conjunction with fork. Because the argument list is of arbitrary length, it must be terminated by a null pointer to mark the end of the list.

As always a short example is worth a thousand words, and the following program uses execl to run the directory listing program ls:

```
/* runls -- use "execl" to run ls */

#include <unistd.h>

main()
{
 printf("executing ls\n");

 execl("/bin/ls", "ls", "-l", (char *)0);

 /* If execl returns, the call has failed, so ... */
 perror("execl failed to run ls");
 exit(1);
}
```

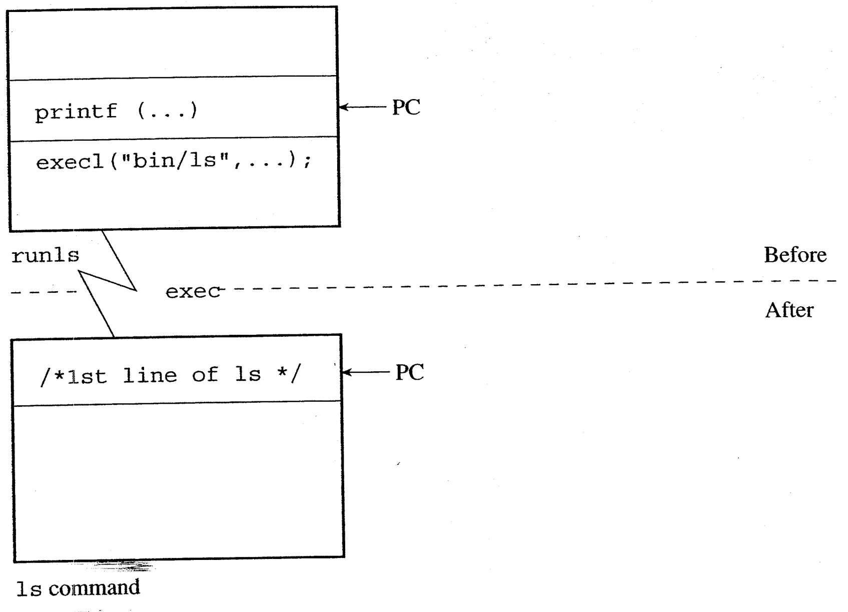

Figure 5.3 *The* exec *call.*

The action of this example program is shown in Figure 5.3. The *Before* section shows the process immediately before the call to `execl`. The *After* section shows the transformed process, after the call to `execl`, which now runs the `ls` program. The program counter *PC* points at the first line of `ls`, indicating that `execl` causes the new program to start from its beginning.

Notice that in the example, the call to `execl` is followed by an unconditional call to the library routine `perror`. This reflects the way a successful call to `execl` (and for that matter all its relations) obliterates the calling program. If the calling program does survive and `execl` returns, then an error must have occurred. As a corollary to this, when `execl` and its relatives do return, they will always return −1.

The `execv`, `execlp` *and* `execvp` *calls*

The other forms of `exec` give the programmer flexibility in the construction of parameter lists. `execv` takes just two arguments: the first (`path` in the usage description above) is a string containing the pathname of the program to be executed. The second (`argv`) is an array of strings, declared as:

```
char * const argv[];
```

The first member of this array points, again purely by convention, to the name of the program to be executed (excluding any pathname prefix). The remaining members point to any additional arguments for the program. Since this list is of indeterminate length, it must always be terminated by a null pointer.

The next example uses `execv` to run the same `ls` command as the previous example:

```
/* runls2 -- uses execv to run ls */

#include <unistd.h>

main()
{
 char * const av[]={"ls","-l",(char *)0};

 execv("/bin/ls", av);

 /* again - getting this far implies error */
 perror("execv failed");
 exit(1);
}
```

`execlp` and `execvp` are almost identical to `execl` and `execv`. The main difference is that the first argument for both `execlp` and `execvp` is a simple filename, not a pathname. The path prefix for this filename is found by a search of the directories denoted by the shell environment variable `PATH`. `PATH` of course can be simply set at shell level with a sequence of commands such as:

```
$ PATH=/bin:/usr/bin:/usr/keith/mybin
$ export PATH
```

This means that both the shell and `execvp` will search for commands first in `/bin`, then `/usr/bin` and finally `/usr/keith/mybin`.

Exercise 5.2 In what circumstances would you use `execv` instead of `execl`?

Exercise 5.3 Assume that `execvp` and `execlp` do not exist. Write subroutine equivalents using `execl` and `execv`. The parameters for these routines should consist of a list of directories and a set of command line arguments.

5.3.2 Accessing arguments passed with `exec`

Any program can gain access to the arguments in the `exec` call that invoked it through parameters passed to the `main` function of the program. These parameters can be used by defining the program's `main` function as follows:

```
main(int argc, char **argv)
{
 /* body of program */
}
```

This should look familiar to many of you, since the same technique is used for accessing arguments from the command line that started a program, another indication that the shell itself uses `exec` to start processes. (We have, not unreasonably, assumed a knowledge of command line parameters in a few of the preceding examples and exercises. This section should therefore clarify things for those who had any problems with these.)

In the `main` function declaration above, `argc` is an integer count of the number of arguments. `argv` points to the array of arguments themselves. So, if a program is executed through a call to `execvp` as follows:

```
char * const argin[] = {"command","with","arguments",(char *)0};

execvp("prog", argin);
```

then within `prog` we would find the following conditions holding true:

```
argc == 3

argv[0] == "command"

argv[1] == "with"

argv[2] == "arguments"

argv[3] == (char *)0
```

As a simple demonstration of this technique, consider the next program, which prints its arguments, excluding its first, on standard output:

```
/* myecho -- echo command line arguments */

main(int argc, char **argv)
{
 while(--argc > 0)
       printf("%s ", *++argv);

 printf("\n");
}
```

If this program is invoked with the following program fragment:

```
char * const argin[]={"myecho", "hello", "world", (char *)0};

execvp(argin[0], argin);
```

then `argc` in `myecho` would be set to 3, and the following output would result:

```
hello world
```

which is the same result as would be obtained by the shell command:

```
$ myecho hello world
```

Exercise 5.4 Write `waitcmd`, a program which, when a file changes, executes an arbitrary command. It should pick up both the name of the file to watch and the command to execute from its command line arguments. The calls `stat` and `fstat` can be used to monitor the file. The program should not unnecessarily waste system resources; therefore use the standard `sleep` subroutine (introduced in Exercise 2.16) to make `waitcmd` pause for a decent interval after it has examined the file. How should it cope if the file does not exist initially?

5.4 Using `exec` and `fork` together

`fork` and `exec` combined offer the programmer a powerful tool. By forking, then using `exec` within the newly created child, a program can run another program within a subprocess and without obliterating itself. The following example shows how. In it we also introduce a simple error routine called `fatal` and, rather prematurely, the system call `wait`. This system call makes a process wait for its child to finish what it is doing – a common experience of any parent. It is discussed in detail later.

```
/* runls3 -- run ls in a subprocess */

#include <unistd.h>

main()
{
 pid_t pid;

 switch(pid = fork()){
 case -1:
       fatal("fork failed");
       break;
 case 0:
       /* child calls exec */
       execl("/bin/ls", "ls", "-l", (char *)0);
       fatal("exec failed");
       break;
```

```
  default:
        /* parent uses wait to suspend execution
         * until child finishes
         */
        wait((int *)0);
        printf("ls completed\n");
        exit(0);
  }
}
```

fatal simply uses perror to display a message, then exits. (The break following the call to fatal insures the code against any future changes to the subroutine.) fatal is implemented as follows:

```
int fatal(char *s)
{
 perror(s);
 exit(1);
}
```

Again, we will use a diagram for clarity in explaining the program's action, in this case Figure 5.4. This is divided into three parts: *Before* fork, *After* fork and *After* exec.

In the initial state, *Before* fork, there is a single process *A* and the program counter *PC* points at the fork statement, indicating that this is the next statement due to be executed.

After the fork call, there are two processes *A* and *B*. *A*, the parent process, is executing the wait system call. This will cause the execution of *A* to be suspended until *B* terminates. Meanwhile, *B* is using execl to load the ls command.

What happens next is shown in the *After* exec part of Figure 5.4. Process *B* has been transformed and now executes the ls program. The program counter for *B* has been set to the first statement of ls. Because *A* is waiting for *B* to terminate, its *PC* arrow has not changed position.

You should now be able to see the outline of some of the mechanisms employed by the shell. For example, when a command is executed, in the normal fashion, the shell uses fork, exec and wait as above. When a command is placed in the background the call to wait is omitted until later and both shell and command processes run concurrently.

The docommand *example*

UNIX provides a library routine called system which allows a shell command to be executed from within a program. Using fork and exec we will implement a rudimentary version of this called docommand. We will invoke a standard shell (identified by the pathname /bin/sh) as an intermediary, rather than attempt to run the command directly. This allows docommand to take advantage of the features offered by the shell such as filename expansion. The -c argument used in the

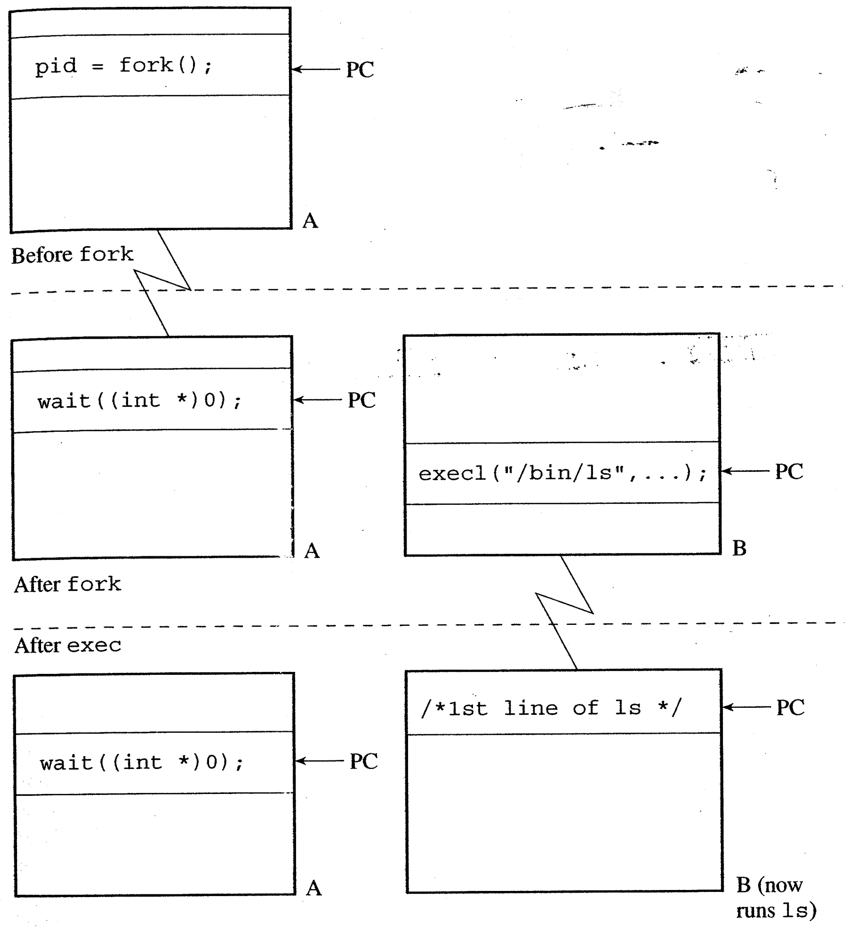

Figure 5.4 *The* fork *and* exec *calls combined.*

invocation of the shell tells it to take commands from the next string argument, rather than standard input.

```
/* docommand -- run shell command, first version */

#include <unistd.h>

int docommand(char *command)
{
 pid_t pid;
```

```
    if ((pid = fork()) < 0 )
          return (-1);

    if (pid == 0) /* child */
    {
          execl("/bin/sh", "sh", "-c", command, (char *)0);
          perror("execl");
          exit(1);
    }

    /* code for parent */
    /* wait until child exits */
          wait((int *)0);
          return(0);
}
```

This, it must be said, is only a first approximation to the proper library routine `system`. For example, if the end user of the program hit the interrupt key while the shell command was running, then both the command and the calling program would be stopped. There are ways to circumvent this, but we will delay discussing these until the next chapter.

5.5 Inherited data and file descriptors

5.5.1 `fork`, files and data

A child process created with `fork` is an almost perfect copy of its parent. In particular all variables within the child will retain the values they held in the parent (the one exception is the return value from `fork` itself). Since the data available to the child is a *copy* of that available to the parent, and occupies a different absolute place in memory, it is important to realize that subsequent changes in one process will not affect the variables in the other.

Similarly, all the files open in the parent are also open in the child; the child maintaining its own copy of the file descriptors associated with each file. However, files kept open across a call to `fork` remain intimately connected in child and parent. This is because the read–write pointer for each file is shared between the child and the parent. This is possible because the read–write pointer is maintained by the system; it is not embedded explicitly within the process itself. Consequently, when a child process moves forward in a file, the parent process will also find itself at the new position. The following short program demonstrates this. In it we use the routine `fatal` introduced earlier in this chapter, and a new routine called `printpos`. In addition, we are assuming the existence of a file called `data` which is at least 20 characters in length.

```
/* proc_file -- shows how files are handled across forks */
/* assume "data" is at least 20 chars long */
```

```
#include <unistd.h>
#include <fcntl.h>

main()
{
 int fd;
 pid_t pid;                      /* process-id */
 char buf[10];                   /* buffer to hold file data */

 if(( fd = open("data", O_RDONLY)) == -1)
       fatal("open failed");

 read(fd, buf, 10);              /* advance file pointer */

 printpos("Before fork", fd);

 /* now create two processes */
 switch(pid = fork()){
 case -1:                        /* error */
       fatal("fork failed");
       break;
 case 0:                         /* child */
       printpos("Child before read", fd);
       read(fd, buf, 10);
       printpos("Child after read", fd);
       break;
 default:                        /* parent */
       wait((int *)0);
       printpos("Parent after wait", fd);
 }
}
```

`printpos` simply displays the current position within a file together with a short message. It can be implemented as follows:

```
/* print position in file */
int printpos(const char *string, int filedes)
{
 off_t pos;

 if(( pos = lseek(filedes, 0, SEEK_CUR)) == -1)
       fatal("lseek failed");
 printf("%s:%ld\n", string, pos);
}
```

When we ran this example we obtained the following results, fairly conclusive proof that the read–write pointer is shared by both processes:

```
Before fork:10
Child before read:10
Child after read:20
Parent after wait:20
```

Exercise 5.5 Write a program that demonstrates that program variables in a parent and child process have the same initial values but are independent of each other.

Exercise 5.6 Discover what happens within a parent process when a child closes a file descriptor inherited across a fork. In other words, does the file remain open in the parent, or is it closed there?

5.5.2 `exec` and open files

Open file descriptors are also normally passed across calls to `exec`. That is, files open in the original program are kept open when an entirely new program is started through `exec`. The read–write pointers for such files are unchanged by the `exec` call. (It obviously makes no sense to talk about the values of variables being preserved across an `exec` call, since the original and newly loaded programs will in general be entirely different.)

However, the all-purpose, all-weather routine `fcntl` can be used to set the **close-on-exec** flag associated with a file. If this is on (the default is off), then the file is closed when any member of the `exec` family is invoked. The following code fragment shows how the close-on-exec flag is enabled:

```
#include <fcntl.h>

.
.
.

int fd;

fd = open("file", O_RDONLY);

.
.
.

/* set close-on-exec flag on */
fcntl(fd, F_SETFD, 1);
```

The close-on-exec flag can be turned off with:

```
fcntl(fd, F_SETFD, 0);
```

Its value can be obtained as follows:

```
res = fcntl(fd, F_GETFD, 0);
```

The integer `res` will be 1 if the close-on-exec flag is on for the file descriptor `fd`, 0 otherwise.

5.6 Terminating processes with the `exit` system call

Usage

```
#include <stdlib.h>

void exit(int status);
```

The `exit` system call is an old friend, but we can now place it into its proper context. It is used to terminate a process, although a process will also stop when it runs out of program by reaching the end of the `main` function, or when `main` executes a `return` statement.

The single, integer argument to `exit` is called the process' **exit status**, the low-order eight bits of which are available to the parent process, providing it has executed a `wait` system call (more details of this in the next section). The value returned through `exit` in this way is normally used to indicate the success or failure of the task performed by the process. By convention a process returns zero on normal termination, some non-zero value if something has gone wrong.

As well as stopping the calling process, `exit` has a number of other consequences: most importantly, all open file descriptors are closed. If the parent process has executed a `wait` call, as in the last example, it will be restarted. `exit` will also call any programmer-defined exit handling routines and perform what are generally described as clean-up actions. These can, for example, be concerned with buffering in the Standard I/O Library. A programmer can also set at least 32 exit handling routines with the `atexit` function.

Usage

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

The `atexit` routine registers the function pointed to by `func`, to be called with no parameters. Each of the exit handling functions recorded by `atexit` will be called on exit in the reverse order to which they were set.

For completeness, we should also mention the system call `_exit`, which is distinguished from `exit` by the leading underscore in its name. This is used in exactly the same way as `exit`. However, it circumvents the clean-up actions we described earlier. The `_exit` call should be avoided by most programmers.

Exercise 5.7 The exit status of a program can be obtained by using the '$?' variable within the shell, for example:

```
$ ls nonesuch
 nonesuch: No such file or directory
$ echo $?
 2
```

Write a program called `fake` which uses the integer value of its first argument as its exit status. Using the method outlined above, try out `fake` using a variety of arguments, including negative and large values. Is `fake` a useful program?

5.7 Synchronizing processes

5.7.1 The `wait` system call

Usage

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

As we have already seen briefly, `wait` temporarily suspends the execution of a process while a child process is running. Once the child has finished, the waiting parent is restarted. If more than one child is running then `wait` returns as soon as any one of the parent's offspring exits.

`wait` is often called by a parent process just after a call to `fork`. For example:

```
.
.
.
int status;
pid_t cpid;

cpid = fork(); /*create new process */

if(cpid == 0){

 /* child */
 /* do something .... */

}else{

 /* parent, so wait for child */
 cpid = wait(&status);
 printf("The child %d is dead\n",cpid);

}
.
.
.
```

This combination of `fork` and `wait` is most useful when the child process is intended to run a completely different program by calling `exec`.

The return value from `wait` is normally the process-id of the exiting child. If `wait` returns `(pid_t)-1`, it can mean that no child exists and in this case `errno` will contain the error code `ECHILD`. Being able to tell if any child has terminated individually means that the parent process can sit in a loop waiting for each of its offspring. When the parent realizes that all the children have terminated, it can continue.

`wait` takes one argument, `status`, a pointer to an integer. If the pointer is `NULL` then the argument is simply ignored. If, however, `wait` is passed a valid pointer, `status` will contain useful status information when `wait` returns. Normally this information will be the exit-status of the child passed through `exit`.

The following program `status` demonstrates how `wait` is used in these circumstances:

```
/* status -- how to get hold of a child's exit status */

#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
 pid_t pid;
 int status, exit_status;

 if((pid = fork())< 0)
       fatal("fork failed");

 if(pid == 0)             /* child */
 {
       /* now call the library routine sleep
        * to suspend execution for 4 seconds
        */
       sleep(4);
       exit(5);           /* exit with non-zero value */
 }

 /* getting this far means this is the parent */
 /* so wait for child */

 if((pid = wait(&status)) == -1)
 {
       perror("wait failed");
       exit(2);
 }

 /* test to see how the child died */
 if(WIFEXITED(status))
```

```
  {
        exit_status = WEXITSTATUS(status);
        printf("Exit status from %d was %d\n",pid, exit_status);
  }

  exit(0);
}
```

The value returned to the parent via `exit` is stored in the high-order eight bits of the integer `status`. For these to be meaningful the low-order eight bits must be zero. The macro `WIFEXITED` (defined in `<sys/wait.h>`) tests to see if this is in fact the case. The macro `WEXITSTATUS` returns the value stored in the high-order bits of `status`. If `WIFEXITED` returns 0 then it means that the child was stopped in its tracks by another process, using a communication method called a **signal**. Signals will be discussed in Chapter 6.

Exercise 5.8 Adapt the `docommand` routine so that it returns the `exit` status of the command it executes. What should happen if the `wait` call involved returns −1?

5.7.2 Waiting for a particular child: `waitpid`

The `wait` system call allows a parent to wait for any child. However, if the parent wants to be more particular it can use the `waitpid` system call to wait for a specific child process.

Usage

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

The first argument, `pid` specifies the process-id of the child process that the parent wishes to wait for. If this is set to −1 and the `options` argument is set to 0 then `waitpid` behaves exactly the same as `wait`, since −1 denotes an interest in any child process. If `pid` is greater than 0 then the parent will wait for the child with a process-id of `pid`. The second argument, `status`, will hold the status of the child when `waitpid` returns.

The final argument, `options`, can take a variety of values defined in `<sys/wait.h>`. The most useful of these options is `WNOHANG`. This allows `waitpid` to sit in a loop monitoring a situation but not blocking if the child process is still running. If `WNOHANG` is set then `waitpid` will return 0 if the child has not yet terminated.

The functionality of `waitpid` with the `WNOHANG` option can be demonstrated by rewriting the previous example. This time the parent process checks to see if the child has finished. If not, it outputs a message to say that it is still waiting, it then sleeps for a second and again calls `waitpid` to see if the child has completed. Notice that the child process gets its process-id by calling `getpid`. We will explain this in Section 5.10.1.

```
/* status2 --
 * how to obtain a child's exit status using waitpid */

#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>

main()
{
 pid_t pid;
 int status, exit_status;

 if((pid = fork())< 0)
       fatal("fork failed");

 if(pid == 0)            /* child */
 {
       /* now call the library routine sleep
        * to suspend execution for 4 seconds
        */
       printf("Child %d sleeping...\n",getpid());
       sleep(4);
       exit(5);          /* exit with non-zero value */
 }

 /* getting this far means this is the parent */
 /* so see if the child has exited if not sleep for */
 /* one second and recheck */

 while(waitpid(pid, &status, WNOHANG) == 0)
 {
       printf("Still waiting...\n");
       sleep(1);
 }

 /* test to see how the child died */
 if(WIFEXITED(status))
 {
       exit_status = WEXITSTATUS(status);
       printf("Exit status from %d was %d\n",pid, exit_status);
 }

 exit(0);
}
```

Running this program produced the following output:

```
Still waiting...
Child 12857 sleeping...
Still waiting...
Still waiting...
Still waiting...
Exit status from 12857 was 5
```

5.10.4 Sessions and session-ids

Each process group in turn belongs to a session. A session is really about a process's connection to a **controlling terminal**. When users log on, the processes and process groups they explicitly or implicitly create will belong to a session linked to their current terminal. A session is typically a collection of a single foreground process group using the terminal and one or more background process groups. A session is identified by a **session-id** of type `pid_t`.

A process can obtain its current session-id with a call to `getsid` as follows:

Usage

```
#include <sys/types.h>
#include <unistd.h>

pid_t getsid(pid_t pid);
```

If `getsid` is passed a value of 0 then it returns the session-id of the calling process otherwise the session-id of the process identified by `pid` is returned.

The idea of a session is useful with background or **daemon** processes. A daemon process is simply a process which does not have a controlling terminal. An example is `cron`, which executes commands at specified times and dates. A daemon can set itself to be in a session *without a controlling terminal* by calling the `setsid` system call, and moving itself into a new session.

Usage

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

If the calling process is not a process group leader then a new process group and session will be created and the process-id of the calling process will become the session-id. It will also have no controlling terminal. The daemon process will now be in a strange state in that it will be the only process in the process group contained in the new session, and its `pid` will also be the process group-id and the session-id.

The `setsid` function will fail if the calling process is already a process group leader returning `(pid_t)-1`.

5.10.5 The environment

The **environment** of a process is simply a collection of null-terminated strings, represented within a program as a null-terminated array of character pointers. By convention, each environment string has the form:

name = something

A programmer can make direct use of the environment of a process by adding an extra parameter `envp` to the parameter list of the `main` function within a program. The following program fragment shows the type of `envp`:

```
main(int argc, char **argv, char **envp)
{
 /* do something */
}
```

As a trivial example, the next program simply prints out its environment and exits:

```
/* showmyenv.c -- show environment */

main(int argc, char **argv, char **envp)
{
 while(*envp)
       printf("%s\n",*envp++);
}
```

Running this program on one machine produced the following results:

```
CDPATH=:..:/
HOME=/usr/keith
LOGNAME=keith
MORE=-h -s
PATH=/bin:/etc:/usr/bin:/usr/cbin:/usr/lbin
SHELL=/bin/ksh
TERM=vt100
TZ=GMT0BST
```

This layout may well look familiar to you. It is the environment of the shell process that invoked the `showmyenv` program and includes important variables used by the shell such as `HOME` and `PATH`.

What this example shows is that the default environment of a process is the same as the process that created it through a call to `exec` or `fork`. Since the environment is passed on in this way, it allows information to be recorded semi-permanently which would otherwise have to be respecified for each new process. The `TERM` environment variable, which stores the current terminal type, is a good example of how useful this can be.

To actually specify a new environment for the process you must use one of two new members of the `exec` family: `execle` and `execve`. These are called as follows:

```
execle(path, arg0, arg1, ..argn, (char *)0, envp);
```

and:

```
execve(path, argv, envp);
```

These duplicate, respectively, the actions of the system calls execv and execl. The one difference is the addition of the envp parameter, which is a null-terminated array of character pointers that specifies the environment for the new program. The next example uses execle to pass a new environment to the showmyenv program:

```
/* setmyenv.c -- set environment for program */

main()
{
 char *argv[2], *envp[3];

 argv[0] = "showmyenv";
 argv[1] = (char *)0;

 envp[0] = "foo=bar";
 envp[1] = "bar=foo";
 envp[2] = (char *)0;

 execve("./showmyenv", argv, envp);

 perror("execve failed");
}
```

Although it is acceptable to use the parameters passed to the main function, the preferred way a process can gain access to its environment is through the global variable:

```
extern char **environ;
```

A standard library function getenv can be used to scan environ for the name of an environment variable in the form name = string.

Usage

```
#include <stdlib.h>

char *getenv(const char *name);
```

Here the argument to getenv is the name of the variable you wish to find. If the search is successful, getenv returns a pointer to the value part of the string, otherwise it returns the NULL pointer. The following code shows an example of its use:

```
/* find the value of the PATH environment variable */

#include <stdlib.h>
```

```
main()
{
 printf("PATH=%s\n", getenv("PATH"));
}
```

A companion routine putenv is also provided to change or extend the environment. It is called along the following lines:

```
putenv("NEWVARIABLE = value");
```

putenv returns zero if it was successful. It alters the environment pointed to by environ. It does not alter the envp pointer in the current main function.

5.10.6 The current working directory

As we saw in Chapter 4, each process is associated with a current working directory. The initial setting for the current working directory is inherited across the fork or exec that started the process. To put it another way, a process is initially placed in the same directory as its parent.

The fact that the current working directory is a per-process attribute is important. If a child process changes position by calling chdir (defined in Chapter 4), the current working directory in the parent process is unchanged. For this reason the standard cd command is actually a 'built-in' command within the shell itself and does not correspond to a program.

5.10.7 The current root directory

Each process is also associated with a root directory used in absolute pathname searches. As with the current working directory, the root directory of a process is initially determined by that of its parent. UNIX provides a system call for changing a process' idea of where the start of the file system hierarchy is. This system call is chroot.

Usage

```
#include <unistd.h>

int chroot(const char *path);
```

path points to a pathname naming a directory. If chroot succeeds, path will become the starting point for those file searches that begin with a '/' (for the calling process only, the system as a whole is not affected). chroot returns −1 and the root directory remains unchanged if the call fails. The calling process must have the appropriate privileges to change the root directory.

Exercise 5.12 Add the cd command to the smallsh command processor.

Exercise 5.13 Write your own version of the getenv function.

5.10.8 User- and group-ids

Each process is associated with a real user-id and a real group-id. These are always the user and current group-ids of the user who invoked the process.

The effective user- and group-ids are used to determine whether a process can access a file. More often than not, these are the same as the real user- and group-ids. However, a process, or one of its ancestors, can have its set-user-id or set-group-id permission bit set. For example, if the program file's set-user-id bit is set, then when the program is invoked through a call to exec, the effective user-id of the process becomes that of the file owner, not that of the user who started the process.

There are several system calls available for obtaining the user- and group-ids associated with a process. The following program fragment demonstrates these:

```
#include <unistd.h>

main()
{
 uid_t uid, euid;
 gid_t gid, egid;

 /* get real user-id */
 uid = getuid();

 /* get effective user-id */
 euid = geteuid();

 /* get real group-id */
 gid = getgid();

 /* get effective group-id */
 egid = getegid();
}
```

Two calls are also available for setting the effective user- and group-ids of a process:

```
#include <unistd.h>

uid_t newuid;
gid_t newgid;
.
.
.
/* set effective user-id */
status = setuid(newuid);
```

```
/* set effective group-id */
status = setgid(newgid);
```

A process invoked by a non-privileged user (that is, anybody who is not superuser) can only reset its effective user- and group-ids back to their real counterparts. Superuser as usual is allowed free rein. The return value from both routines is 0 on successful completion, −1 otherwise.

Exercise 5.13 Write a routine that obtains the real user- and group-ids of the calling process, then writes the ASCII equivalents of these onto the end of a log file.

5.10.9 File size limits: ulimit

There is a per-process limit on the size of a file that can be created with the write system call. This limit also covers the situation where a pre-existing file, shorter than the limit, is extended.

The file size limit can be manipulated with the ulimit system call.

Usage

```
#include <ulimit.h>

long ulimit(int cmd, [long newlimit]);
```

To obtain the current file size limit, a programmer can call ulimit with the parameter cmd set to UL_GETFSIZE. The value returned is in units of 512-byte blocks.

To change the file size limit, a programmer should set cmd to UL_SETFSIZE and place the new file size limit, again in 512-byte blocks, into newlimit. For example:

```
if(ulimit(UL_SETFSIZE, newlimit) < 0)
 perror("ulimit failed");
```

Only superuser can actually increase a file size limit in this way. Processes that have the effective user-ids of other users are, however, allowed to decrease their limit.

5.10.10 Process priorities: nice

The system decides the proportion of cpu time a particular process is allocated partly on the basis of an integer **nice** value. Nice values range from 0 to a system-dependent maximum. The higher the number, the lower the process' priority.

Socially aware processes can lower their priority, and thus allocate more resources to other processes, by using the `nice` system call. This takes one argument, a positive increment to be added to the current nice value; for example:

```
nice(5);
```

Superuser (and only superuser) processes can increase their priority by using a negative value as the `nice` call parameter. `nice` is a useful call if all you want to do is calculate π to a hundred million places and not impact the system's responsiveness to other users. `nice` is an old call. Recent optional real-time extensions in *POSIX* give far more fine grain control. However, this is a specialist topic which we will not consider further.