Starting New Processes

You can cause a program to run from inside another program and thereby create a new process by using the system library function.

#include <stdlib.h>

int system (const char *string);

The system function runs the command passed to it as a string and waits for it to complete. The command is executed as if the command $ **sh -c string** has been given to a shell. system returns 127 if a shell can't be started to run the command and -1 if another error occurs. Otherwise, system returns the exit code of the command.

Try It Out system

You can use system to write a program to run ps. Though this is not tremendously useful in and of itself, you'll see how to develop this technique in later examples. (We don't check that the system call actually worked for the sake of simplicity in the example.)

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
printf("Running ps with system\n");
system("ps ax");
printf("Done.\n");
exit(0);
}
```

When you compile and run this program, system1.c, you get something like the following:

```
$ ./system1
Running ps with system
PID TTY STAT TIME COMMAND
1 ? Ss 0:03 init [5]
...
1262 pts/1 Ss 0:00 /bin/bash
1273 pts/2 S 0:00 su -
1274 pts/2 S+ 0:00 -bash
1463 pts/2 SN 0:00 oclock
1465 pts/1 S 0:01 emacs Makefile
1480 pts/1 S+ 0:00 ./system1
1481 pts/1 R+ 0:00 ps ax
Done.
```

Because the system function uses a shell to start the desired program, you could put it in the background by changing the function call in system1.c to the following:

system("ps ax &"); When you compile and run this version of the program, you get something like

```
$ ./system2
Running ps with system
PID TTY STAT TIME COMMAND
1 ? S 0:03 init [5]
...
```

Done.
$ 1274 pts/2 S+ 0:00 -bash
1463 pts/1 SN 0:00 oclock
1465 pts/1 S 0:01 emacs Makefile
1484 pts/1 R 0:00 ps ax
469
Chapter 11: Processes and Signals

How It Works

In the first example, the program calls system with the string "ps ax", which executes the ps program.

The program returns from the call to system when the ps command has finished. The system function can be quite useful but is also limited. Because the program has to wait until the process started by the call to system finishes, you can't get on with other tasks.

In the second example, the call to system returns as soon as the shell command finishes. Because it's a request to run a program in the background, the shell returns as soon as the ps program is started, just as would happen if you had typed

$ **ps ax &**

at a shell prompt. The system2 program then prints Done. and exits before the ps command has had a chance to finish all of its output. The ps output continues to produce output after system2 exits and in this case does not include an entry for system2.

In general, using system *is a far from ideal way to start other processes, because it invokes the desired program using a shell. This is both inefficient, because a shell is started before the program is started, and also quite dependent on the installation for the shell and environment that are used. In the next section, you see a much better way of invoking programs, which should almost always be used in preference to the* system *call.*

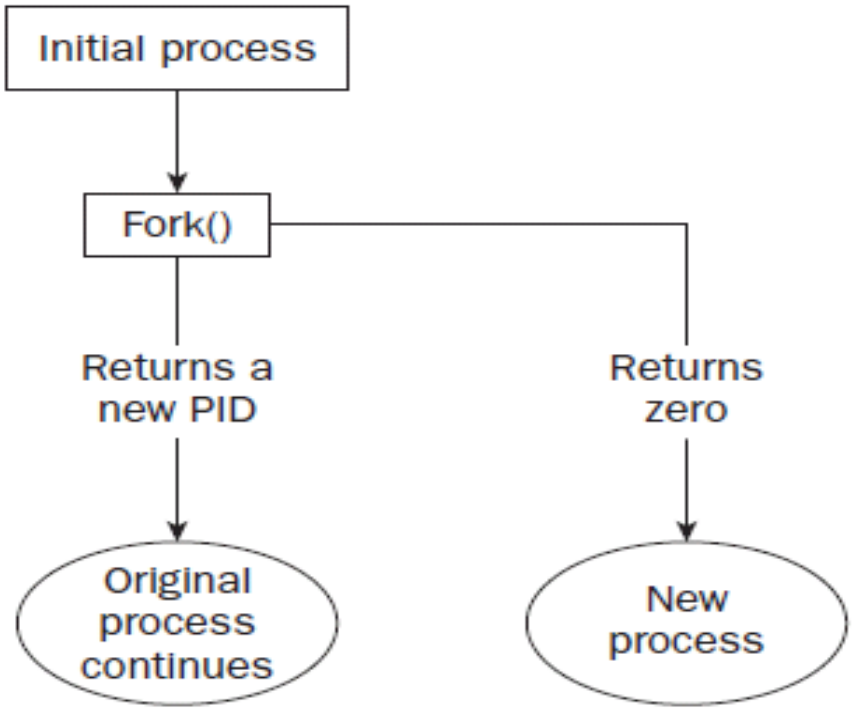
Duplicating a Process Image

To use processes to perform more than one function at a time, you can either use threads, or create an entirely separate process from within a program, as init does, rather than replace the current thread of execution, as in the exec case.

You can create a new process by calling fork. This system call duplicates the current process, creating a new entry in the process table with many of the same attributes as the current process. The new process is almost identical to the original, executing the same code but with its own data space, environment, and file descriptors. Combined with the exec functions, fork is all you need to create new processes.

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

As you can see in Figure 11-2, the call to fork in the parent returns the PID of the new child process. The new process continues to execute just like the original, with the exception that in the child process the call to fork returns 0. This allows both the parent and child to determine which is which.



If fork fails, it returns -1. This is commonly due to a limit on the number of child processes that a parent may have (CHILD_MAX), in which case errno will be set to EAGAIN. If there is not enough space for an entry in the process table, or not enough virtual memory, the errno variable will be set to ENOMEM.

A typical code fragment using fork is

```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
break;
default : /* We are parent */
break;
}
```

Try It Out fork

Let's look at a simple example, fork1.c:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
pid_t pid;
char *message;
int n;
```

```
printf("fork program starting\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
message = "This is the child";
n = 5;
break;
default:
message = "This is the parent";
n = 3;
break;
}
for(; n > 0; n--) {
puts(message);
sleep(1);
}
exit(0);
}
```

This program runs as two processes. A child is created and prints a message five times. The original process (the parent) prints a message only three times. The parent process finishes before the child has printed all of its messages, so the next shell prompt appears mixed in with the output.

```
$ ./fork1
fork program starting
This is the child
This is the parent
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

How It Works

When fork is called, this program divides into two separate processes. The parent process is identified by a nonzero return from fork and is used to set a number of messages to print, each separated by one second.

Waiting for a Process

When you start a child process with fork, it takes on a life of its own and runs independently. Sometimes, you would like to find out when a child process has finished. For example, in the previous program, the parent finishes ahead of the child and you get

some messy output as the child continues to run. You can arrange for the parent process to wait until the child finishes before continuing by calling wait.

#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);

The wait system call causes a parent process to pause until one of its child processes is stopped. The call returns the PID of the child process. This will normally be a child process that has terminated. The status information allows the parent process to determine the exit status of the child process, that is, the value returned from main or passed to exit. If stat_loc is not a null pointer, the status information will be written to the location to which it points.

You can interpret the status information using macros defined in sys/wait.h, shown in the following table.

Try It Out wait

In this Try It Out, you modify the program slightly so you can wait for and examine the child process exit status. Call the new program wait.c.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
pid_t pid;
char *message;
int n;
int exit_code;
printf("fork program starting\n");
pid = fork();
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
message = "This is the child";
n = 5;
exit_code = 37;
break;
default:
message = "This is the parent";
n = 3;
exit_code = 0;
break;
}
for(; n > 0; n--) {
```

```
puts(message);
sleep(1);
}
This section of the program waits for the child process to finish.
if (pid != 0) {
int stat_val;
pid_t child_pid;
child_pid = wait(&stat_val);
printf("Child has finished: PID = %d\n", child_pid);
if(WIFEXITED(stat_val))
printf("Child exited with code %d\n", WEXITSTATUS(stat_val));
else
printf("Child terminated abnormally\n");
}
exit(exit_code);
}
```

When you run this program, you see the parent wait for the child.

```
$ ./wait
fork program starting
This is the child
This is the parent
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 1582
Child exited with code 37
$
```

How It Works

The parent process, which got a nonzero return from the fork call, uses the wait system call to suspend its own execution until status information becomes available for a child process. This happens when the child calls exit; we gave it an exit code of 37. The parent then continues, determines that the child terminated normally by testing the return value of the wait call, and extracts the exit code from the status information.

Zombie Processes

Using fork to create processes can be very useful, but you must keep track of child processes. When a child process terminates, an association with its parent survives until the parent in turn either terminates normally or calls wait. The child process entry in the process table is therefore not freed up immediately.

Although no longer active, the child process is still in the system because its exit code needs to be stored in case the parent subsequently calls wait. It becomes what is known as defunct, or a *zombie process*.

You can see a zombie process being created if you change the number of messages in the fork example program. If the child prints fewer messages than the parent, it will finish first and will exist as a zombie until the parent has finished.

Try It Out Zombies

fork2.c is the same as fork1.c, except that the number of messages printed by the child and parent processes is reversed. Here are the relevant lines of code:

```
switch(pid)
{
case -1:
perror("fork failed");
exit(1);
case 0:
message = "This is the child";
n = 3;
break;
default:
message = "This is the parent";
n = 5;
break;
}
```

How It Works

If you run the preceding program with ./fork2 & and then call the ps program after the child has finished but before the parent has finished, you'll see a line such as this. (Some systems may say <zombie> rather than <defunct>.)

```
$ ps –al
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
004 S 0 1273 1259 0 75 0 - 589 wait4 pts/2 00:00:00 su
000 S 0 1274 1273 0 75 0 - 731 schedu pts/2 00:00:00 bash
000 S 500 1463 1262 0 75 0 - 788 schedu pts/1 00:00:00 oclock
000 S 500 1465 1262 0 75 0 - 2569 schedu pts/1 00:00:01 emacs
000 S 500 1603 1262 0 75 0 - 313 schedu pts/1 00:00:00 fork2
003 Z 500 1604 1603 0 75 0 - 0 do_exi pts/1 00:00:00 fork2 <defunct>
000 R 500 1605 1262 0 81 0 - 781 - pts/1 00:00:00 ps
```

If the parent then terminates abnormally, the child process automatically gets the process with PID 1 (init) as parent. The child process is now a zombie that is no longer running but has been inherited by init because of the abnormal termination of the parent process. The zombie will remain in the process table until collected by the init process. The bigger the table, the slower this procedure. You need to avoid zombie processes, because they consume resources until init cleans them up.

There's another system call that you can use to wait for child processes. It's called waitpid, and you can use it to wait for a specific process to terminate.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

The pid argument specifies the PID of a particular child process to wait for. If it's -1, waitpid will return information for any child process. Like wait, it will write status

information to the location pointed to by stat_loc, if that is not a null pointer. The options argument allows you to modify the behavior of waitpid. The most useful option is WNOHANG, which prevents the call to waitpid from suspending execution of the caller. You can use it to find out whether any child processes have terminated and, if not, to continue. Other options are the same as for wait.
So, if you wanted to have a parent process regularly check whether a specific child process has terminated, you could use the call
waitpid(child_pid, (int *) 0, WNOHANG);
This will return zero if the child has not terminated or stopped, or child_pid if it has. waitpid will return -1 on error and set errno. This can happen if there are no child processes (errno set to ECHILD), if the call is interrupted by a signal (EINTR), or if the option argument is invalid (EINVAL).