CHAPTER 20

Forking

VIRTUALLY ALL AUTHORS of servers, and many authors of clients, need to write programs that can effectively handle multiple network connections simultaneously. As an example, consider a web server. If your server could only handle one connection at a time, you could only be transmitting a single page at a time. If you have a large file on your server and a user on a slow link is downloading it, that user could completely tie up your server for an hour or more. During that time, nobody else would be able to view any pages on that server. Virtually all servers want to be able to serve more than one client at once.

To serve multiple clients simultaneously, you need to have some way to handle several network connections at once. Python provides three primary ways to meet that objective: forking, threading, and asynchronous I/O (also known as nonblocking sockets). I'll cover all three: forking in this chapter, threading in Chapter 21, and asynchronous I/O in Chapter 22.

Of these three, forking is probably the easiest to understand and use. However, it's not completely portable; forking may be unavailable on platforms that aren't derived from UNIX.

Forking involves *multitasking*—the ability to run multiple processes at once, or to simulate that ability. In this chapter, you'll learn how to apply forking to your programs. First, you'll learn about how forking works with your operating system and some common pitfalls to avoid. Next, you'll see how to apply forking to server programs. Finally, the chapter will conclude with information on locking and error handling.

Understanding Processes

Forking is tied in closely with the operating system's nature of a process. A *process* is usually defined as "an executing instance of a program." When you start up an editor such as Emacs, the operating system creates a new process that runs it. When Emacs terminates, that process goes away. If you open up two copies of Emacs, there will be two Emacs processes running. Although they both may be instances of `/usr/bin/emacs` and may be started up the same way, they may be doing different things—perhaps editing different files. Each process is distinct.

Each process has a unique identification number called a *process ID* (PID). The operating system assigns the PID to a process when it starts. In the preceding Emacs example, the two Emacs processes would each have a unique PID.

> **NOTE** This chapter focuses on UNIX and Linux platforms, since those are platforms for which forking is best supported. The information contained in this chapter may not apply to other operating systems such as Windows.
>
> However, although the details may differ in important ways, all multitasking operating systems (including Windows) have some notion of a process, even if they don't refer to it by that name. Single-tasking operating systems, such as DOS, will usually have no notion of a process.

You can gather information about running processes by using the `ps` command. The syntax for `ps` differs from one UNIX to the next. Here's an example from Linux, which should also work on any BSD operating system and AIX:

```
$ ps x
  PID TTY      STAT   TIME COMMAND
19817 ?        S      0:00 /bin/sh /usr/bin/startkde
19866 ?        Ss     0:00 /usr/bin/ssh-agent startkde
19877 ?        Ss     0:02 kdeinit: Running...
19880 ?        S      0:18 kdeinit: dcopserver --nosid
19882 ?        S      0:01 kdeinit: klauncher
19885 ?        S      0:26 kdeinit: kded
19966 ?        S     34:11 /usr/bin/artsd -F 5 -S 4096 -a alsa -s 60 -m artsmess
...
12096 ?        S      0:00 xterm
12097 pts/668  Ss     0:00 -bash
12154 pts/668  S+     0:00 emacs -nw letter.txt
12155 ?        S      0:00 xterm
12156 pts/669  Ss     0:00 -bash
12163 pts/669  S+     0:00 emacs -nw report.txt
```

The first column in the `ps` output shows the PID of a given process. The last column, in most cases, shows what program that process is executing. In this example, the first processes listed correspond to the graphical environment KDE. They represent things such as the sound system. I've cut out several dozen other processes for this example.

Farther down, you can see two different versions of Emacs running as in the previous example. One has PID 12154 and the other has PID 12163. The first was originally started to edit `letter.txt` and the second was started to edit `report.txt`.

Each process has unique attributes. For instance, PID 12154 may have an open file descriptor for `letter.txt` while PID 12163 may have an open file descriptor for `report.txt`. Processes can also have unique environment variables, data in memory, and open network connections.

The process is the fundamental unit of multitasking. Several processes may be running simultaneously. For instance, my two Emacs processes could be running at the same time as a web browser, a file downloading process, a data analysis process, and a CD burning process. A single process doesn't have more than one thing executing at once. Threading, discussed in Chapter 21, can blur that line.

Understanding fork()

The system call used to implement forking is called `fork()`. It's a very unique call. Most functions will return exactly once (with or without a value). The `sys.exit()` function never returns since it terminates the program. By contrast, Python's `os.fork()` is the only function that actually returns *twice*. After calling `fork()`, there are two copies of your program running at once. But the second copy doesn't restart from the beginning; both copies continue directly after the call to `fork()`—the process's entire address space is copied. Errors are possible, and `os.fork()` could raise an exception; see the "Error Handling" section in this chapter for details.

The `fork()` call returns the process ID (PID) of the newly created process to the original ("parent") process. To the new ("child") process, it returns a PID of 0. Therefore, logic like this is common:

```
def handle():
    pid = os.fork()
    if pid:
        # Parent
        close_child_connections()
        handle_more_connections()
    else:
        # Child
        close_parent_connections()
        process_this_connection()
```

When I said before that `os.fork()` is the only function that returns twice, that's not entirely accurate. I could write the following:

```
def dothefork():
    pid = os.fork()
    if pid:
        return "server"
    else:
        return "client"
```

In this instance, `dothefork()` would actually return twice as well. It should be noted, though, that any function that returns twice does, at some point, call `os.fork()` to make that possible.

Forking is one of the most common and best-understood methods of multitasking, and using forks is especially common for servers, whereby the server typically forks for each new incoming request.

After a fork, each process has a distinct address space. Modifying a variable in one process will not modify it in another, and that is a key difference from threads (discussed in the next chapter). This leaves your code less vulnerable to errors that may cause the server process for one connection to interfere with that of another.

Forking is used, on UNIX systems, for more than just network purposes. For instance, the typical way (and what Python does under the hood when you call `os.system()`) to execute a program is to fork and then use one of the `os.exec...()` functions to start the new program. The parent process can then continue on, monitoring the child, or it can opt to have its execution blocked until the child terminates by using one of the `wait()` functions (which will be described later in the section "The Zombie Problem").

However, forking is a fairly low-level operation. The process of actually doing a fork takes a little bit of work to make sure that you're doing everything the operating system expects of you.

Duplicated File Descriptors

There are several side effects of forking. One of the most obvious is that of duplicated file descriptors. A file descriptor can refer to things such as a socket, a file on disk, a terminal (standard input/output/error), or certain other file-like objects.

Since a forked copy of a process is an exact copy, it inherits all the file descriptors and sockets that the parent process had. So you wind up with a situation in which both the parent and child process have a connection open to a single remote host.

That's bad for several reasons. One is that if both processes try to communicate over that socket, the result will likely be garbled. Another is that a call to `close()` doesn't actually close the connection until *both* processes have called it.

Therefore, protocols (such as FTP) that use the closing of a socket as a signal that some action has completed will be broken unless sockets are closed both places. Some authors do, on occasion, exploit the fact that two processes can access the socket, but this requires great care and is quite rare.

The solution to this problem is to have whichever process doesn't need a socket close it immediately after forking. For the typical case of a server that forks a new process to handle each incoming request, you'll notice that the parent process will close the socket for the child, and the child will close the master listening socket that the parent uses. This will ensure proper operation for both processes.

Zombie Processes

The semantics of `fork()` are built around the assumption that the parent process is interested in finding out when and how a child process terminated. For instance, a shell script is interested in finding out the exit code from a program that is run. A parent process can find out not just the exit code, but also if a process crashed or terminated due to a signal. The way a parent gathers this information is via `os.wait()` or a similar call.

During the time between the termination of the child process, and the time the parent calls `wait()` in it, the child process is said to be a *zombie* process. It's no longer executing, yet certain memory structures are still present in order to permit the parent to `wait()` on it.

For most servers, the information returned by `wait()` is irrelevant. If a worker process dies, the server will not do anything different; it should still go on servicing requests from other clients.

However, you must still call `wait()` on the child process at some point after it terminates. Otherwise, system resources will be consumed by the vast amount of zombie processes, which could eventually render the server machine unusable.

The operating system makes that job fairly easy, though. Each time a child process terminates, it sends the `SIGCHLD` signal to its parent process. (A signal is a rudimentary way to inform a process of certain events.) The parent process can set a signal handler to receive `SIGCHLD` and clean up any children that have terminated. While this sounds tricky, I'll show you an example in the "The Zombie Problem" section later in this chapter that can accomplish this very easily.

If the parent process dies before its children, the children will continue running. The system re-parents them, setting their parent to be `init` (process 1). The `init` process will then take care of cleaning up zombies.

Performance

You may think that using `fork()` is a slow proposition since it must copy over all of a server each time a client connects. In reality, the performance hit of `fork()` is insignificant and unnoticeable to all but the most heavily loaded systems.

Most modern operating systems, such as Linux, implement `fork()` with copy-on-write memory. That means that memory isn't actually copied until it needs to be (when one process or the other modifies it). The call to `fork()` itself is usually virtually instantaneous.

The `fork()` call is used all over in the system. For instance, when you're using a shell and type `ls`, the shell will fork a copy of itself, and the new process will invoke `ls`. A similar thing happens if you click an icon to launch a program in a graphical environment. The desktop manager or window manager will fork itself, and then call `exec()` to start the new program. When you call `os.system()` from a Python program, there's an internal call for `fork()` and `exec()` in the same manner.

Extremely heavily loaded systems that serve many brief connections, such as web servers for very popular sites, may not want to put up with even the small overhead of forking. These servers sometimes use a *forked pool*, in which the forking is done in advance and processes are reused. They might also choose to use asynchronous I/O, which has no per-process overhead, or threading, which has less of an overhead. For general-purpose use, forking remains a good choice.

Forking First Steps

Here's a simple first example of forking. It's going to fork, and both processes will display some messages.

```
#!/usr/bin/env python
# First fork example - Chapter 20 - firstfork.py

import os, time

print "Before the fork, my PID is", os.getpid()

if os.fork():
    print "Hello from the parent.  My PID is", os.getpid()
else:
    print "Hello from the child.  My PID is", os.getpid()

time.sleep(1)
print "Hello from both of us."
```

This program will print out its process ID prior to forking. Then, because `fork()` returns twice, the parent and child each print out a unique message, and they both fall out of the `if`, wait for one second, then display a greeting. Here's what the output looks like:

```
$ ./firstfork.py
Before the fork, my PID is 2700
Hello from the child.  My PID is 2701
Hello from the parent.  My PID is 2700
 ... one second later ...
Hello from both of us.
Hello from both of us.
```

On some systems, you may observe that the order of the parent and child messages is different, and they may be different each time you run the program. The operating system makes no guarantee about that, as in fact, both processes should be executing simultaneously.

Notice how `Hello from both of us` is displayed twice, even though it occurs in the code only once. That's because, by the time the execution reaches that point, there are actually two copies of the program running.

The Zombie Problem

Let's take a look at the aforementioned zombie problem in action. The UNIX command `ps` shows a list of active processes. Here's an example that will demonstrate the zombie problem. While it's running, open up another terminal session and take a look at the state of processes.

```
#!/usr/bin/env python
# Zombie problem demonstration - Chapter 20 - zombieprob.py

import os, time

print "Before the fork, my PID is", os.getpid()

pid = os.fork()
if pid:
    print "Hello from the parent.  The child will be PID %d" % pid
    print "Sleeping 120 seconds..."
    time.sleep(120)
```

The child process will terminate immediately after the fork (`fork()` returns PID 0 for the child, so it will fail the `if` test, and there's nothing else for it to do). The parent doesn't clean it up, but rather waits around for a while. Run the program as follows:

```
$ ./zombieprob.py
Before the fork, my PID is 2719
Hello from the parent.  The child will be PID 2720
Sleeping 120 seconds...
```

Now, in another terminal session, inspect the results without stopping the program:

```
$ ps ax | grep 2719
 2719 pts/2    S      0:00 python ./zombieprob.py
$ ps ax | grep 2720
 2720 pts/2    Z      0:00 [python] <defunct>
```

You can see that the child process is a zombie; the `Z` in the third column, as well as the `<defunct>` at the end of the output, indicate that. Once the parent terminates, you'll be able to confirm that neither process exists. The shell cleans up the parent process, and the child process gets re-parented to `init`, which will clean it up.

The Role of init

The `init` program is always the first process that runs on the system and always has PID 1. Its main roles are starting up and shutting down the system. In this case, there's another special role for `init`. If a process dies, and there are still children of it out there on the system (zombie or not), the operating system will change that process's parent to be PID 1—`init`. The `init` program will watch for zombie children in the same way that normal processes will, so these processes will get cleaned up.

Solving the Zombie Problem with Signals

Here's a program that solves the zombie problem:

```
#!/usr/bin/env python
# Zombie problem solution - Chapter 20 - zombiesol.py

import os, time, signal

def chldhandler(signum, stackframe):
    """Signal handler.  Runs on the parent and is called whenever
    a child terminates."""
    while 1:
         # Repeat as long as there are children to collect.
       try:
            result = os.waitpid(-1, os.WNOHANG)
        except:
            break
        print "Reaped child process %d" % result[0]
    # Reset the signal handler so future signals trigger this function
    signal.signal(signal.SIGCHLD, chldhandler)

# Install signal handler so that chldhandler() gets called whenever
# child process terminates.
signal.signal(signal.SIGCHLD, chldhandler)

print "Before the fork, my PID is", os.getpid()

pid = os.fork()
if pid:
    print "Hello from the parent.  The child will be PID %d" % pid
    print "Sleeping 10 seconds..."
    time.sleep(10)
    print "Sleep done."
else:
    print "Child sleeping 5 seconds..."
    time.sleep(5)
```

First, the program defines the signal handler `chldhandler()`. This function is called whenever `SIGCHLD` is received. It has a simple loop calling `os.waitpid()`. The first argument to `os.waitpid()`, –1, means to wait for any terminated child process, and the second tells it to return immediately if no more terminated processes exist. If there are child processes waiting, `waitpid()` returns a tuple of a process's PID and exit information. Otherwise, it raises an exception. The act of using `wait()` or `waitpid()` to collect information about terminated processes is called *reaping*.

The call is in a loop because a single `SIGCHLD` could indicate multiple child processes have died. Finally, after the loop, the signal handler is reactivated. This