

is necessary because some UNIX implementations deactivate the signal handler when it's called. By explicitly reactivating it, you ensure that it gets called again when the next child process terminates. (It won't happen in this example, but it will in real servers.)

The call to `signal.signal()` establishes the signal handler. The first argument is the signal of interest, and the second one names the function that should be called when it arrives. That function must accept two arguments: the signal number and an optional stack frame.

The remainder of the program is fairly typical. When you run the program, you'll see output like this:

```
$ ./zombiesol.py
Before the fork, my PID is 2931
Child sleeping 5 seconds...
Hello from the parent. The child will be PID 2932
Sleeping 10 seconds...
Reaped child process 2932
Sleep done.
```

You'll notice that the parent reaps the child process only five seconds into its sleep, since that's how long it takes before the child process terminates. The signal handler is called immediately.

You might also notice that the parent process never finishes its sleep. There's a special case with `time.sleep()` in that if any signal handler is called, the sleep will terminate immediately, rather than continue waiting the remaining amount of time. Since you'll rarely need to use `time.sleep()` with networking code, this shouldn't be an issue.

Solving the Zombie Problem with Polling

Another approach to solving the zombie problem is to periodically check for zombie children. This method doesn't involve a signal handler, and as such, will not cause problems for `sleep()`. Signal handlers can also cause problems with I/O functions on some operating systems, which is a larger problem for network clients.

Here's another solution to the zombie problem. Instead of using a signal handler, it will periodically try to collect any zombie processes.

```
#!/usr/bin/env
# Zombie probl

import os, tin

def reap():
    """Try to
    while 1:
        try:
            r
        except:
            b:
        print

    print "Before

    pid = os.fork
    if pid:
        print "He
        print "Pa
        time.sleep
        print "Pa
        reap()
        print "Pa
        time.sleep
        print "Pa
    else:
        print "Ch
        time.sleep
        print "Ch
```

This prog
function is ve
process woul
While there v
since new on
When yo

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

import os, time

def reap():
    """Try to collect zombie processes, if any."""
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
        except:
            break
        print "Reaped child process %d" % result[0]

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 "Parent sleeping 60 seconds..."
    time.sleep(60)
    print "Parent sleep done."
    reap()
    print "Parent sleeping 60 seconds..."
    time.sleep(60)
    print "Parent sleep done."
else:
    print "Child sleeping 5 seconds..."
    time.sleep(5)
    print "Child terminating."
```

This program will simply call `reap()` to gather up the child processes. This function is very similar to the signal handler in the previous example. A server process would probably call `reap()` at the bottom of its primary `accept()` loop. While there will sometimes be zombie processes out there, they won't build up, since new ones would be created only after cleaning up the older ones.

When you run this problem, you'll see output like this:

```

$ ./zombiepoll.py
Before the fork, my PID is 3667
Child sleeping 5 seconds...
Hello from the parent. The child will be PID 3668
Parent sleeping 60 seconds...
Child terminating.
Parent sleep done.
Reaped child process 3668
Parent sleeping 60 seconds...
Parent sleep done.

```

If you run the program, you'll notice several differences between it and the previous one. First of all, the child process wasn't reaped immediately when it terminated. Secondly, the call to `time.sleep()` wasn't interrupted. Finally, if you do a `ps` during the 55 seconds between the time the child exits and the time it's reaped, you'll see it listed as a zombie. But you can see that it's been cleaned up during the last 60 seconds of the program.

Forking Servers

Forking is most commonly used for network servers. I presented code for several different servers in Chapter 3, but each sample shared a common problem: It could only serve one client at a time. This is rarely an acceptable limitation, and forking is one of the most common ways to solve the problem. The concepts demonstrated earlier can be applied to the server code. Here's an example of an echo server that uses forking. Because it uses forking, it can echo text back to several clients at once.

```

#!/usr/bin/env python
# Echo Server with Forking - Chapter 20 - echoserver.py
# Compare to echo server in Chapter 3

import socket, traceback, os, sys

def reap():
    # Collect any child processes that may be outstanding
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
            if not result[0]: break

```

```

except:
    break
print "Reaped child process %d" % result[0]

host = ''
port = 51423                                # Bind to all interfaces

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

print "Parent at %d listening for connections" % os.getpid()

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Clean up old children.
    reap()

    # Fork a process for this connection.
    pid = os.fork()

    if pid:
        # This is the parent process. Close the child's socket
        # and return to the top of the loop.
        clientsock.close()
        continue
    else:
        # From here on, this is the child.
        s.close()                                # Close the parent's socket

        # Process the connection

```

```

try:
    print "Child from %s being handled by PID %d" % \
        (clientsock.getpeername(), os.getpid())
    while 1:
        data = clientsock.recv(4096)
        if not len(data):
            break
        clientsock.sendall(data)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

# Close the connection

try:
    clientsock.close()
except KeyboardInterrupt:
    raise
except:
    traceback.print_exc()

# Done handling the connection. Child process must terminate
# and not go back to the top of the loop.
sys.exit(0)

```

Let's look at this program, which is the TCP echo server from Chapter 3 with forking added in. Now it can handle multiple clients simultaneously.

First, the function `reap()` is defined similarly to the previous examples. However, there's an additional test: to see whether or not the PID returned by `waitpid()` is zero. In the previous cases, this test was skipped, since we always knew that `reap()` was called when there was at least one zombie process, but that might not be the case here.

Then, the code proceeds unmodified until after the call to `accept()`. The first new call is to `reap()`. This will clean up any zombie processes that have terminated since the last time a client connected. Next, the program forks and uses the usual `if pid` design.

If the process post-fork is the parent, it will close the child's socket and return to the top of the loop with `continue` to list for more connections. If we're in the child process, it closes the parent process's socket and then processes the connection as usual. However, there's a change at the end—the child calls `sys.exit(0)` when it's done processing. This is vitally important. If it didn't do this, execution would

return to the t
tions as well a
the client clos
client termina
Try runni
that it echoes
status messag

```

$ ./echoserver
Parent at 1627
Child from ('1
Child from ('1

```

This show
processes.

Locking

A simple prog
system. Howe
you have to be
connection at

For instar
be a problem
or corrupted,

To solve th
is most frequ
process to per
that uses lock

```

#!/usr/bin/env
# Locking serv
# NOTE: lastac

```

```
import socket,
```

```
def getlastacc
    """Given a
    from that
    never an a

```

return to the top of the while loop, and the child would try to accept new connections as well as the parent. In this particular case, it will generate an error since the client closed its copy of the master socket. The `sys.exit()` makes sure that the client terminates when it should.

Try running the program. You can then connect to port 51423 and observe that it echoes text back to you. On the console, the server will print out some status messages. Here's what it looked like for me:

```
$ ./echoserver.py
Parent at 16271 listening for connections
Child from ('127.0.0.1', 37708) being handled by PID 16273
Child from ('127.0.0.1', 37709) being handled by PID 16285
```

This shows two incoming connections being handled by two different processes.

Locking

A simple program like an echo server never needs to write to any files on the local system. However, this isn't necessarily the case for all servers. When using forking, you have to be wary of concurrency issues that don't occur if you only service one connection at once.

For instance, if part of the task of your server is to write lines to a file, it would be a problem to have two servers writing to the file at once. Changes could be lost or corrupted, and the two processes could overwrite each other's changes.

To solve this problem, you'll need to use locking. In forking programs, locking is most frequently used to control access to files. Locking lets you force only one process to perform certain actions at a time. Here's an example of a forking server that uses locking:

```
#!/usr/bin/env python
# Locking server with Forking - Chapter 20 - lockingserver.py
# NOTE: lastaccess.txt will be overwritten!

import socket, traceback, os, sys, fcntl, time

def getlastaccess(fd, ip):
    """Given a file descriptor and an IP, finds the date of last access
    from that IP in the file and returns it. Returns None if there was
    never an access from that IP."""
```

```

# Acquire a shared lock. We don't care if others are reading the file
# right now, but they shouldn't be writing it.
fcntl.flock(fd, fcntl.LOCK_SH)

try:
    # Start at the beginning of the file
    fd.seek(0)

    for line in fd.readlines():
        fileip, accesstime = line.strip().split("|")
        if fileip == ip:
            # Got a match -- return it
            return accesstime
    return None
finally:
    # Make sure the lock is released no matter what
    fcntl.flock(fd, fcntl.LOCK_UN)

def writelastaccess(fd, ip):
    """Update file noting new last access time for the given IP."""

    # Acquire an exclusive lock. Nobody else can modify the file
    # while it's being used here.
    fcntl.flock(fd, fcntl.LOCK_EX)
    records = []

    try:
        # Read the existing records, *except* the one for this IP.
        fd.seek(0)
        for line in fd.readlines():
            fileip, accesstime = line.strip().split("|")
            if fileip != ip:
                records.append((fileip, accesstime))

        fd.seek(0)

        # Write them back out, *plus* the one for this IP.
        for fileip, accesstime in records + [(ip, time.asctime())]:
            fd.write("%s|%s\n" % (fileip, accesstime))
        fd.truncate()
    finally:
        # Release the lock no matter what
        fcntl.flock(fd, fcntl.LOCK_UN)

```

```

def reap():
    """Collect all dead children"""
    while 1:
        try:
            result, _ = os.waitpid(-1, os.WNOHANG)
            if result != 0:
                break
        except:
            pass
        print "Reaped %d children" % result

host = ''
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
fd = open("lastaccess.txt", "w")

while 1:
    try:
        clientsocket, _ = s.accept()
        except KeyboardInterrupt:
            raise
        except:
            traceback.print_exc()
            continue

        # Clean up old records
        reap()

        # Fork a process to handle the client
        pid = os.fork()

        if pid:
            # This is the parent process
            # and return the client socket
            clientsocket.close()
            continue
        else:
            # From here on, this is the child process
            s.close()

```

```

def reap():
    """Collect any waiting child processes."""
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
            if not result[0]: break
        except:
            break
        print "Reaped child process %d" % result[0]

host = ''
port = 51423

# Bind to all interfaces

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)
fd = open("lastaccess.txt", "w+")

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Clean up old children.
    reap()

    # Fork a process for this connection.
    pid = os.fork()

    if pid:
        # This is the parent process. Close the child's socket
        # and return to the top of the loop.
        clientsock.close()
        continue
    else:
        # From here on, this is the child.
        s.close()
        # Close the parent's socket

```

```

# Process the connection

try:
    print "Got connection from %s, servicing with PID %d" % \
        (clientsock.getpeername(), os.getpid())
    ip = clientsock.getpeername()[0]
    clientsock.sendall("Welcome, %s.\n" % ip)
    last = getlastaccess(fd, ip)
    if last:
        clientsock.sendall("I last saw you at %s.\n" % last)
    else:
        clientsock.sendall("I've never seen you before.\n")

    writelastaccess(fd, ip)
    clientsock.sendall("I have noted your connection at %s.\n" % \
        getlastaccess(fd, ip))

except (KeyboardInterrupt, SystemExit):
    raise
except:
    traceback.print_exc()

# Close the connection

try:
    clientsock.close()
except KeyboardInterrupt:
    raise
except:
    traceback.print_exc()

# Done handling the connection. Child process *must* terminate
# and not go back to the top of the loop.
sys.exit(0)

```

This is a fairly basic server. It simply notes the last time a connection was received from a given IP and notes that in a file. The algorithm used to do that is rather inefficient and vulnerable to *race conditions*—situations in which the outcome depends on which process happens to get to the data first.

To com
getlastacce
This reques
shared lock
function, b
same time,
If anothe
process will
is a *blocking*
At the e
argument o
held. It's *vit*
in *deadlock*
automatica

TIP N
whethe
run. A c
use try
to be sk

The wr
except tha
exclusive lo
the file. Tha
well as othe
After th
writes it ba
first acquir
The answer
then betwe
acquired, a
know about
would be lo
Let's lo
./lockingse
example of

To combat that, it uses `fcntl.flock()` to restrict access to the file. The `getlastaccess()` function starts out by calling `fcntl.flock(fd, fcntl.LOCK_SH)`. This requests a shared lock on the file. Any number of processes can hold a shared lock as long as no process holds an exclusive lock. That's fine for this function, because it's only reading. It's OK if other processes are reading at the same time, but you don't want to be reading while someone else is writing.

If another process tries to acquire a lock while this process holds it, the other process will stall at the `flock()` call until the lock can be acquired. Therefore, this is a *blocking* call because execution is blocked until a lock is acquired.

At the end of `getlastaccess()`, `flock()` is called again, this time with an argument of `LOCK_UN`, which means "unlock" and effectively releases the lock held. It's *vital* that all acquired locks must be released. Failure to do so can result in *deadlock*, where processes are waiting on each other. The only time a lock is automatically released for you is when your process terminates.

TIP Notice that the unlocking occurs in a `finally` clause. This means that whether an exception was caught or not, the unlocking command is always run. A common error is to fail to use `try...finally` around locks. Unless you use `try...finally`, an unexpected exception can cause the `unlock` command to be skipped, resulting in *deadlock*.

The `writelastaccess()` function uses a pattern similar to `getlastaccess()`, except that it acquires an exclusive lock with `LOCK_EX`. When a process holds an exclusive lock, it guarantees that no other process can have a lock of any type on the file. That's what you want here, since you want to lock out all the other readers as well as other instances of `writelastaccess()`.

After the lock is acquired, `writelastaccess()` loads the file from disk, then writes it back out with the new information. You may be wondering why I didn't first acquire a shared lock for reading, followed by an exclusive lock for writing. The answer is that this would introduce a race condition. If I used that approach, then between the time the lock for reading is released and the lock for writing is acquired, another process could have written out data. My process would not know about this data (having just read the file premodification), and the change would be lost. That's why it's important to use a single lock for this entire function.

Let's look at what this program does when it's run. You can just use `./lockingserver.py` to start it. Then, you can telnet to the server. Here's an example of a client-side session:

```

$ telnet localhost 51423
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome, 127.0.0.1.
I've never seen you before.
I have noted your connection at Thu Jul 1 06:06:42 2004.
Connection closed by foreign host.
$ telnet localhost 51423
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Welcome, 127.0.0.1.
I last saw you at Thu Jul 1 06:06:42 2004.
I have noted your connection at Thu Jul 1 06:08:44 2004.
Connection closed by foreign host.

```

Here, the first time the client connected, the server didn't have a record of it in its `lastaccess.txt` file. It recorded the connection time. For the second connection, the server reports the saved connection time and records the new connection time.

While these connections were occurring, the server was reporting this:

```

$ ./lockingserver.py
Got connection from ('127.0.0.1', 37742), servicing with PID 16848
Reaped child process 16848
Got connection from ('127.0.0.1', 37743), servicing with PID 16850

```

In this particular case, the second child process wasn't yet reaped even though it had terminated. When a third child would connect, it would be reaped.

Error Handling

Strange as it may seem, `os.fork()` can fail. This is rare but does happen. The cause of a failure would be a resource limitation of some kind—the operating system may be out of memory, it may be out of space in its process table, or you may run up against a limit on the maximum number of processes set by an administrator.

There's no good way to deal with this situation. If you don't check for an error, a failure on `os.fork()` will terminate the program. For a client, that's OK, but for a server, it means your server completely dies.

A better way is to kill off just the one connection that caused the problem, and hope that the administrator notices the problem or that the thing causing the problem (a wayward program, for instance) goes away. If so, then when later

clients connect, not be restarted.

Remember to be more specific, there's an error, that's why you're

Here's a mo with `os.fork()`

```

#!/usr/bin/env py
# Echo Server wi
# errorserver.py

```

```
import socket, t
```

```

def reap():
    while 1:
        try:
            resu
            if n
        except:
            brea
            print "R

```

```

host = ''
port = 51423

```

```

s = socket.socke
s.setsockopt(soc
s.bind((host, po
s.listen(1)

```

```

while 1:
    try:
        clientso
    except Keybo
        raise
    except:
        tracebac
        continue

```

```

# Clean up o
reap()

```

clients connect, the fork should succeed. This way, the server process itself need not be restarted.

Remember at the beginning of the chapter I said that `fork()` returns twice. To be more specific, `fork()` either returns twice or raises an exception due to an error. If there's an error, there's no PID returned and execution doesn't fork off—after all, that's why you're getting the exception.

Here's a modified version of the forking echo server that handles problems with `os.fork()`:

```
#!/usr/bin/env python
# Echo Server with Forking and Forking Error Detection - Chapter 20
# errorserver.py

import socket, traceback, os, sys

def reap():
    while 1:
        try:
            result = os.waitpid(-1, os.WNOHANG)
            if not result[0]: break
        except:
            break
        print "Reaped child process %d" % result[0]

host = ''                                # Bind to all interfaces
port = 51423

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind((host, port))
s.listen(1)

while 1:
    try:
        clientsock, clientaddr = s.accept()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()
        continue

    # Clean up old children.
    reap()
```

```

# Fork a process for this connection.
try:
    pid = os.fork()
except:
    print "BAD THING HAPPENED: fork failed"
    clientsock.close()
    continue

if pid:
    # This is the parent process. Close the child's socket
    # and return to the top of the loop.
    clientsock.close()
    continue
else:
    print "New child", os.getpid()
    # From here on, this is the child.
    s.close()                                # Close the parent's socket

    # Process the connection

    try:
        print "Got connection from", clientsock.getpeername()
        while 1:
            data = clientsock.recv(4096)
            if not len(data):
                break
            clientsock.sendall(data)
    except (KeyboardInterrupt, SystemExit):
        raise
    except:
        traceback.print_exc()

    # Close the connection

    try:
        clientsock.close()
    except KeyboardInterrupt:
        raise
    except:
        traceback.print_exc()

    # Done handling the connection. Child process *must* terminate
    # and not go back to the top of the loop.
    sys.exit(0)

```

You'll notice that if `fork()` fails, the server returns to the top of the loop and never be used, and it has to wait over it. More importantly, it has time in which for each discarded case, Python's garbage collector is bad, but it's bad.

This program is a server that listens for connections and then sends the connection to the client. If the server cannot handle the connection, it sends the alternative. If the master process is busy, say, three minutes, it could cause the server to be at all. A few clients could cause the server to be at all.

Unfortunately, that's easily done. There are restrictions on the system problem.

Summary

Most server programs are several methods. The easiest is this. The easiest is

To fork, you need the process ID of the

When a process is in the system until it is using forking must minutes. One way of polling, and periodic

Forking server incoming connections file descriptors the

You'll notice that this program is mostly the same as the previous example. If `fork()` fails, the server displays an error message, closes the client socket, and returns to the top of the loop. It's important to close that client socket—it will never be used, and this ensures that the client knows not to try communicating over it. More importantly, imagine a scenario in which there was a prolonged time in which `fork()` fails—perhaps the system has run out of memory. It might have to turn away thousands of client requests. If it doesn't close those sockets, each discarded request will continue consuming resources. (In this particular case, Python's garbage collector will likely keep that problem from getting very bad, but it's bad practice to rely upon that behavior).

This program is also notable for what it *doesn't* do. It sends no message whatsoever to the client. The client will simply see a connection reset by peer message if the server cannot fork. This isn't particularly friendly to the client, but consider the alternative. If the server cannot fork, everything it does is taking place in the master process. If it takes a while to communicate with a poorly connected client—say, three minutes—then during that time the server isn't accepting connections at all. A few clients that are attempting to connect when the server can't fork could cause the server to be rendered effectively no better than if it had crashed.

Unfortunately, testing your `os.fork()` error-handling code isn't something that's easily done. Causing `os.fork()` to fail means enforcing administrative restrictions on process counts (not always easily done), or actually causing a system problem.

Summary

Most server programs have a need to handle more than one client at once. There are several methods available to the server designer who wants to accomplish this. The easiest is forking, which is available primarily on Linux and UNIX platforms.

To fork, you call `os.fork()`, which returns twice. That function returns the process ID of the child to the parent, and returns 0 to the child.

When a process terminates, information about its termination remains on the system until its parent calls `wait()` or `waitpid()` on it. Therefore, programs using forking must make sure to call `wait()` or `waitpid()` when a child process terminates. One way to do that is via a signal handler. Alternatively, you could use polling, and periodically check for terminated child processes.

Forking servers usually will use `fork()` to create a new process to handle each incoming connection. It's important for both the parent and child to close any file descriptors that won't be used in that particular process.

If files will be modified, locking is important. Locking prevents data corruption that could occur if multiple processes attempt to modify a file at once, or if one process reads a file while another is writing to it.

The `os.fork()` function can raise an exception if the system cannot perform a fork. Though rare, this exception must be handled to prevent a server crash.

FORKING, WHERE requests to be separate processes. Threading. The single process ever need an explained by applications t

In some other connection between receives upload server may ne are updating that use fork(With threads, just running one thread, all global variable the program. V so changing a

However, between threads always good. mess up the t that threads d and debug.

Through communication p doesn't turn o threading in E issues. Next, y concludes with