Ordering Information:

Python How to Program
The Complete Python Training Course

- View the complete **Table of Contents**
- Read the **Preface**
- Download the **Code Examples**

To view all the Deitel products and services available, visit the Deitel Kiosk on InformIT at **www.informIT.com/deitel**.

To follow the Deitel publishing program, sign-up now for the *DEITEL™ BUZZ ONLINE* e-mail newsletter at **www.deitel.com/newsletter/subscribeinformIT.html**.

To learn more about our **Python programming courses** or any other Deitel instructor-led corporate training courses that can be delivered at your location, visit **www.deitel.com/training**, contact our Director of Corporate Training Programs at (978) 461-5880 or e-mail: **christi.kelsey@deitel.com.**

Note from the Authors: This article is an excerpt from Chapter 18, Section 18.2 of *Python How to Program*. In this article, we discuss the mechanics of process creation and management in Python. Process management is a task common to programs such as shells and simple Web servers. We introduce functions **`os.fork`** and **`os.wait`**. We also introduce the concept of asynchronously executing processes and the unpredictability of their relative execution speeds. Readers should be familiar with basic Python programming, modules and exception handling. The code examples included in this article show readers programming examples using the DEITEL™ signature LIVE-CODE™ Approach, which presents all concepts in the context of complete working programs followed by the screen shots of the actual inputs and outputs.

18.2 `os.fork` Function

Creating new processes is useful in applications that can perform multiple tasks in parallel. For example, the Apache Web server (prior to version 2.0) used multiple processes to handle multiple client requests simultaneously. Each of these processes was an identical copy of the main Apache process. In this case, making identical copies of the main Apache process was useful, because each of these processes performed the same task (i.e., the serving of Web pages to clients).

One way to create a new process is to use function **`os.fork`**, which is available only on POSIX-compliant systems (e.g., most versions of UNIX and Linux). Module **`os`** on the Windows version of Python does not define function **`os.fork`**, because Windows does not support the creation of new processes by using **`fork`**. Instead, Windows applications programmers typically use multithreaded programming techniques to accomplish concurrency tasks.

Common Programming Error 18.1

Attempting to execute a Python program that invokes **`os.fork`** *on a Windows machine causes an* **`AttributeError`** *exception because the* **`os`** *module for Windows does not define function* **`fork`**.

Portability Tip 18.1

Function **`os.fork`** *is unavailable for Windows versions of Python.*

Figure 18.1 describes how function **`os.fork`** creates a new process. Each time a program executes, the operating system creates a new process to run the program's instructions (Step 1). A process also may cause the operating system to create a new process by calling **`os.fork`**. The *parent process* is the process that invokes **`os.fork`**. Any process that the parent process *forks* (creates) is a *child process*. Each process has a unique *process id* number, or *pid*, that identifies the process. When a process invokes function **`fork`**, the operating system creates a new child process that is essentially identical to the parent (original) process (Step 2). The child process inherits copies of many values, such as global variables and environment variables, from the parent process. The only difference between the two processes is the return value of **`fork`**: The **`child`** process receives a return value of 0, and the parent process receives the child's *pid* as the return value. After the call to function **`fork`**, the two processes execute the same program concurrently, starting with the line of code that follows the invocation of **`fork`**. The parent and child processes execute concurrently and independently of one another (i.e., they execute "asynchronously"). Figure 18.2 illustrates an example of function **`os.fork`**.

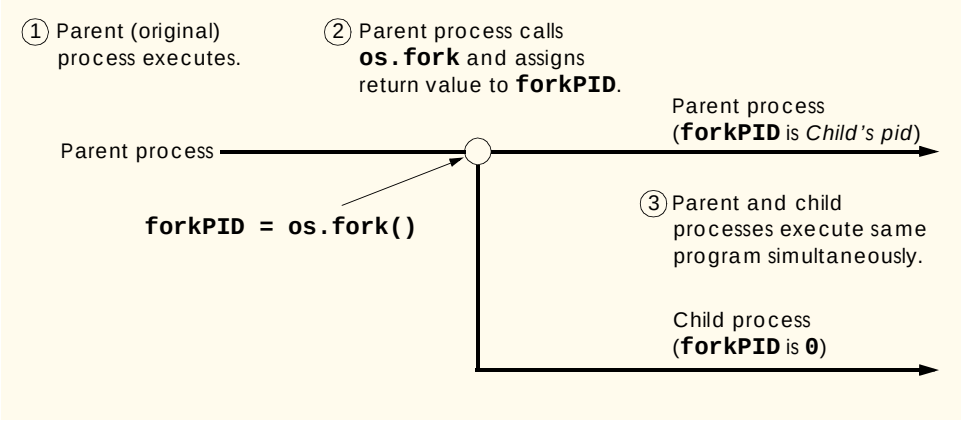

Fig. 18.1 `os.fork` creates a new process.

```
1  # Fig. 18.2: fig18_02.py
2  # Using fork to create child processes.
3
4  import os
5  import sys
6
7  processName = "parent"  # only the parent is running now
8
9  print "Program executing\n\tpid: %d, processName: %s" \
10    % ( os.getpid(), processName )
11
12 # attempt to fork child process
13 try:
14    forkPID = os.fork()  # create child process
15 except OSError:
16    sys.exit( "Unable to create new process." )
17
18 if forkPID != 0:  # am I parent process?
19    print "Parent executing\n" + \
20       "\tpid: %d, forkPID: %d, processName: %s" \
21       % ( os.getpid(), forkPID, processName )
22
23 elif forkPID == 0:  # am I child process?
24    processName = "child"
25    print "Child executing\n" + \
26       "\tpid: %d, forkPID: %d, processName: %s" \
27       % ( os.getpid(), forkPID, processName )
28
29 print "Process finishing\n\tpid: %d, processName: %s" \
30    % ( os.getpid(), processName )
```

Fig. 18.2 `os.fork` used to create child processes. (Part 1 of 2.)

```
Program executing
        pid: 5428, processName: parent
Parent executing
        pid: 5428, forkPID: 5429, processName: parent
Process finishing
        pid: 5428, processName: parent
Child executing
        pid: 5429, forkPID: 0, processName: child
Process finishing
        pid: 5429, processName: child
```

```
Program executing
        pid: 5430, processName: parent
Child executing
        pid: 5431, forkPID: 0, processName: child
Process finishing
        pid: 5431, processName: child
Parent executing
        pid: 5430, forkPID: 5431, processName: parent
Process finishing
        pid: 5430, processName: parent
```

```
Program executing
        pid: 5888, processName: parent
Child executing
Parent executing
        pid: 5888, forkPID: 5889, processName: parent
Process finishing
        pid: 5888, processName: parent
        pid: 5889, forkPID: 0, processName: child
Process finishing
        pid: 5889, processName: child
```

Fig. 18.2 **`os.fork`** used to create child processes. (Part 2 of 2.)

Line 7 initializes variable **`processName`** to **`"parent"`** to indicate that the current process is the parent process. Lines 9–10 print the *pid* and **`processName`** of the parent process. A process can access its *pid* by calling function **`os.getpid`**. Line 14 then calls function **`os.fork`** to create a duplicate of the current process. If the operating system is unable to create a new process, function **`os.fork`** raises an **`OSError`** exception, and line 16 exits the program; otherwise, the operating system creates a new process. Both copies of the process—parent and child—continue execution from the point at which the child process was created (line 14), but each process executes in a separate memory space.

If a program must perform different tasks in the parent and child processes, that program can use **`if`** statements to test the value that **`fork`** returns in each process. The processes then can perform the appropriate tasks based on the results of those **`if`** statements. Recall that **`fork`** returns 0 in the newly created child process; whereas in the parent pro-

cess, **fork** returns the child's *pid*, which must be a positive integer. In the example of Fig. 18.2, the parent process performs different tasks than those that the child process performs. If the executing process is the parent, the return value from fork is the child's pid, and the condition on line 18 evaluates to true. The process then executes the parent-specific code on lines 19–21. If the executing process is the child, **forkPID** is 0 and the condition on line 18 evaluates to false. This prevents the child process from executing the parent-specific code. Instead, the condition on line 23 evaluates to true and the child-specific code in lines 24–27 executes.

The child process changes its copy of variable **processName** to the value **"child"** (line 24). If a program modifies the variable's value in one process, the value of the variable in the other process does not change, because each process contains a separate variable called **processName**.

Each process outputs its *pid*, the value of variable **forkPID** and the value of variable **processName**. Notice in the sample outputs that the child's *pid* matches the value of **forkPID** in the parent process. The parent process can use the child's *pid* to manage the child process, by calling other functions available in module **os**.

Figure 18.2 is followed by three sample executions of the program. Notice that the first two sample outputs differ. After the child process is created by calling **os.fork** (line 14), both processes—parent and child—proceed independently as *asynchronous* concurrent processes. Asynchronous means that they operate independently of one another without synchronization. Concurrent means that they can proceed in parallel (i.e., they execute at the same time). Thus, we cannot predict the relative speeds of the child process and parent process. For this reason, the output of Fig. 18.2 will differ on each execution. Sometimes, the parent process will execute line 19 before the child process executes line 25, and sometimes the child process will execute first. Notice in the last sample output, the parent process executes line 29 while the child process is executing line 25! This underscores the concurrency of the processes.

Another reason the output differs in the sample outputs is because each time a new process is run, the operating system assigns it a unique *pid*. Thus, the *pid* of each process—parent and child—changes with each execution of the program.

In some cases, the parent process must wait for a child process to finish before the parent can proceed. For example, a child process might perform a calculation whose result the parent requires before the parent can continue executing. Function ***os.wait*** (available only on POSIX-compatible systems) waits for any one of the parent's child processes to complete before allowing the parent process to continue its execution. This function returns a two-element tuple that contains the *pid* of the finished child and the child's *exit status*—an integer that indicates the state in which the child process exited. An exit status of 0 indicates that the child process completed successfully; a positive integer indicates that the child process terminated with some error. Function **os.wait** raises an **OSError** exception if there are no children. Figure 18.3 demonstrates function **os.wait**.

```
1  # Fig. 18.3: fig18_03.py
2  # Demonstrates the wait function.
3
```

Fig. 18.3 **os.wait** used to wait for a child process. (Part 1 of 3.)

```
 4  import os
 5  import sys
 6  import time
 7  import random
 8
 9  # generate random sleep times for child processes
10  sleepTime1 = random.randrange( 1, 6 )
11  sleepTime2 = random.randrange( 1, 6 )
12
13  # parent ready to fork first child process
14  try:
15     forkPID1 = os.fork()  # create first child process
16  except OSError:
17     sys.exit( "Unable to create first child. " )
18
19  if forkPID1 != 0:  # am I parent process?
20
21     # parent ready to fork second child process
22     try:
23        forkPID2 = os.fork()  # create second child process
24     except OSError:
25        sys.exit( "Unable to create second child." )
26
27     if forkPID2 != 0:  # am I parent process?
28        print "Parent waiting for child processes...\n" + \
29           "\tpid: %d, forkPID1: %d, forkPID2: %d" \
30           % ( os.getpid(), forkPID1, forkPID2 )
31
32        # wait for any child process
33        try:
34           child1 = os.wait()[ 0 ]  # wait returns one child's pid
35        except OSError:
36           sys.exit( "No more child processes." )
37
38        print "Parent: Child %d finished first, one child left." \
39           % child1
40
41        # wait for another child process
42        try:
43           child2 = os.wait()[ 0 ]  # wait returns other child's pid
44        except OSError:
45           sys.exit( "No more child processes." )
46
47       print "Parent: Child %d finished second, no children left." \
48           % child2
49
50     elif forkPID2 == 0:  # am I second child process?
51        print """Child2 sleeping for %d seconds...
52        \tpid: %d, forkPID1: %d, forkPID2: %d""" \
53           % ( sleepTime2, os.getpid(), forkPID1, forkPID2 )
54        time.sleep( sleepTime2 )  # sleep to simulate some work
55
56  elif forkPID1 == 0:  # am I first child process?
57     print """Child1 sleeping for %d seconds...
```

Fig. 18.3 `os.wait` used to wait for a child process. (Part 2 of 3.)

```
58          \tpid: %d, forkPID1: %d""" \
59          % ( sleepTime1, os.getpid(), forkPID1 )
60      time.sleep( sleepTime1 )  # sleep to simulate some work
```

```
Child2 sleeping for 4 seconds...
        pid: 9578, forkPID1: 9577, forkPID2: 0
Child1 sleeping for 5 seconds...
        pid: 9577, forkPID1: 0
Parent waiting for child processes...
        pid: 9576, forkPID1: 9577, forkPID2: 9578
Parent: Child 9578 finished first, one child left.
Parent: Child 9577 finished second, no children left.
```

```
Parent waiting for child processes...
        pid: 9579, forkPID1: 9580, forkPID2: 9581
Child1 sleeping for 1 seconds...
        pid: 9580, forkPID1: 0
Child2 sleeping for 5 seconds...
        pid: 9581, forkPID1: 9580, forkPID2: 0
Parent: Child 9580 finished first, one child left.
Parent: Child 9581 finished second, no children left.
```

```
Parent waiting for child processes...
Child1 sleeping for 4 seconds...
        pid: 9583, forkPID1: 0
Child2 sleeping for 3 seconds...
        pid: 9584, forkPID1: 9583, forkPID2: 0
        pid: 9582, forkPID1: 9583, forkPID2: 9584
Parent: Child 9584 finished first, one child left.
Parent: Child 9583 finished second, no children left.
```

Fig. 18.3 **`os.wait`** used to wait for a child process. (Part 3 of 3.)

The program creates two child processes, and the parent waits for both child processes to complete execution before the parent process terminates. Each child invokes function **`time.sleep`** to sleep for a random number of seconds (calculated on lines 10–11). The calls to function **`sleep`** make it seem that the child processes perform some work.

Line 15 creates the first child process, and line 23 creates the second child process. The outer **`if`** statement (line 19) evaluates **`forkPID1`**, the return value from the first call to **`fork`** (line 15). If the parent process is executing, the code in lines 21–48 executes. The parent forks a new child (line 23), assigns the return value to **`forkPID2`** and checks the value of this variable to determine if this is still the parent running (line 27).

After creating the second child, the parent prints a message to indicate that the parent will wait for its children (lines 28–30). The parent then calls function **`os.wait`** (line 34). The parent process waits, or *blocks*, until either one of its child processes has finished. Note that this is not necessarily the first child that the parent created. Due to the nature of concurrent, asynchronous processes, the second child process may complete execution first.

Function **os.wait** often is useful when a parent must use the results of the child process's task before the parent can continue its own task (e.g., a parent process that sends an e-mail that contains text output provided by a child process).

The first child process (lines 57–60) prints its *pid* and the value of **forkPID1**. It then calls function **time.sleep** (line 60), with an argument that specifies the length of time in seconds for which the process should remain asleep. We pass a random value (**sleepTime1**) to simulate performing some work. Although programmers cannot rely on function **time.sleep** to provide synchronization, when you run the program, notice that the parent does waits the correct number of seconds before terminating. After the first child process wakes up, it completes execution and terminates successfully. The second child process performs similar tasks (lines 51–54). This second process prints a message and sleeps for a random amount of time before terminating.

Function **os.wait** causes the parent to wait for any one of its child processes to complete execution. Function **os.wait** then returns a tuple containing the *pid* and exit status of the child process that completed, and the parent process resumes execution. When one child process terminates, the parent wakes up and prints a message to the screen that shows the *pid* of the child process that completed (lines 38–39). Recall that processes execute concurrently and asynchronously, so we cannot predict which child process will complete first.[1] Each call to **os.wait** causes the parent to wait for only one of its children. Line 43 calls function **os.wait** a second time, which causes the parent to block again until the remaining child process completes. When the remaining child terminates, the parent prints a message to the screen, which displays the *pid* of that child (lines 47–48).

Notice that the order in which the processes execute differs with each execution, as shown in the three sample outputs for Fig. 18.3. In the first output, the second child process (with pid 9578) completes execution first. This reminds us once again that all of the processes execute concurrently and asynchronously.

1. If a parent must wait for a particular child process to finish, the parent can call function **os.waitpid** and specify the child process's *pid*. See the Python documentation for more detail.