Chapter 9

Interfacing to the OS

The most basic operation of any programming language is to interact with the operating system under which it is running. With a command-line–based operating system, such as Unix or Windows, interaction should include the extraction of other elements and text supplied on the command line. While we're at it, it's probably a good idea to actually parse the contents.

Beyond these basics, there are other more complex operations, including starting new processes, executing external programs, and handling multithreaded operations on those systems that support them.

Python provides a unified entry point for most of these operations in the **sys** and **os** modules. There are also separate modules that handle other systems, such as **time** and **getopt**. We'll have a look at all of these modules in this chapter and provide pointers to other chapters in this section that handle similar systems.

Working with the System (sys Module)

In a strict sense, the **sys** module provides tools for communicating directly with the system under which Python is being executed. Although it does provide some generic functionality (including access to the command-line arguments), its primary purpose is to help the programmer determine the environment under which their script is being executed.

Getting Command-line Arguments

In Python the command-line arguments are available through the **sys.argv** array. As with other arrays, counting starts at zero, with the first element containing the name of the script and elements from index one onwards containing any arguments supplied to the script.

For example, the script

```
import sys
count=0
for argument in sys.argv:
    print "Argument %d is %s" % (count,argument)
    count += 1
```

generates the following output when executed:

```
$ python argv.py hello this is a test
Argument 0 is argv.py
Argument 1 is hello
Argument 2 is this
Argument 3 is is
Argument 4 is a
Argument 5 is test
```

Knowing the name of the script can be useful, especially if you want to report an error to the user.

Parsing Command-line Arguments

You may want to supply arguments to your script that can be parsed just like those supplied to other programs, such as:

```
process.py -y -g --debug --output=process.out myfile.txt
```

In this case you need to use the **getopt** module. The module provides a single function called **getopt** that processes the arguments, placing the information into some more convenient forms. The basic format of the function is

```
getopt(args, options [, long_options])
```

where **args** is the list of arguments you want to parse, and **options** is a string containing the single-letter arguments you want to interpret. If you append a colon to a letter within the **options** string, then the argument will accept an additional argument as data to the argument. If supplied, the **long_options** should be a list of strings defining the words, rather than single letters, which should be identified and parsed. Note that in the arguments list, word-based arguments must be prefixed with a double, rather than a single, hyphen. The = suffix to an argument in **long_options** causes **getopt** to interpret the following argument as additional data.

The **getopt** function returns two objects. The first is a list of tuples containing the parsed argument and its value (if it has one); the second object is a list of the remaining, unparsed arguments.

For example:

```
>>> import getopt
>>> args = ['-a','-x','.bak','--debuglevel','99','file1','file2']
>>> opts, remargs = getopt.getopt(args,'ax:', ['debuglevel='])
>>> opts
[('-a', ''), ('-x', '.bak'), ('--debuglevel', '99')]
>>> remargs
['file1', 'file2']
```

The **getopt** function raises an exception (**GetoptError**) if an unrecognized option is seen in the argument list, or if the argument is expecting additional data but doesn't receive it.

Standard Filehandles

All processes on all platforms potentially support three standard filehandles: those which can be used to communicate with the standard input (the keyboard or terminal),

standard output (the monitor or terminal), or standard error (usually the monitor/terminal). For users of Unix, you can equate these handles to the <, >, and 2> redirection operators. Under Mac OS, these filehandles are emulated by the Python interpreter.

You can access the standard filehandles in Python using the **sys.stdin**, **sys.stdout**, and **sys.stderr** objects. These are file objects, so you must use the file methods defined in Table 10.1 in Chapter 10.

The **print** statement sends its output to the equivalent of **sys.stdout** so the statements

```
print 'Busy doin nothin'
```

and

```
sys.stdout.write('Busy doin nothin\n')
```

produce the same result.

Redirecting Output

It's often useful to redirect the output of one of the standard filehandles to another location. For example, if you wanted to trace the execution of a script, and you were using **sys.stderr** to send messages to the usual error channel, you could redirect the output from within your script.

To redirect any filehandle you need to open a file using the built-in **open()** function to create a new file object and either assign it to a new object and then assign that object to one of the standard filehandles, or assign the object directly, which is what we do here:

```
import sys
sys.stderr = open('error.log','w')
sys.stderr.write('Error log!\n')
sys.stderr.close()
```

Original Filehandles

If you reassign any of the standard filehandles within Python, the original objects as they were at the start of the scripts execution are available using the **sys.__stdin__**, **sys.__stdout__**, and **sys.__stderr__** objects.

Assuming we'd redirected the output to another file using the trick in the previous example, we could put standard error back to normal using:

```
sys.stderr = sys.__stderr__
```

Terminating Execution

Your script naturally terminates when the Python interpreter sees the end of the script—when there are no more statements to execute, then the execution stops. You can also force the termination of a script using the **sys.exit()** function:

```
Python 2.1 (#2, Apr 29 2001, 14:36:04)
[GCC 2.95.3 20010315 (release)] on sunos5
Type "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.exit()
$
```

You can supply an optional argument to the function, which can either be an integer or an object. If it's an integer, then by convention zero indicates successful execution while any nonzero value indicates an error. If you supply an object, then the object is written (using **str()**) to **sys.stderr** and a value of 1 is returned to the caller. If the object is the special value **None** then a zero is returned instead.

The actual implementation of the **sys.exit()** function causes Python to raise a **SystemExit** exception. This forces any **finally** clauses of any outstanding **try** statements to be honored and executed, and it's possible to trap the exception in order to perform cleanup operations (closing files, network/database connections, etc.) before the script finally terminates.

Note *An alternative way to quit is to actually raise the* ***SystemExit*** *exception; it'll have the same effect as calling* ***sys.exit()****.*

Tracing Termination

Although you could use the **SystemExit** exception to trap a termination request and perform some cleanup functions, it's much better to use **sys.exitfunc()**. You use this to define the name of a function to be called when **sys.exit()** is called. For example:

```
import sys

def cleanup():
    print "Had enough, deciding to leave\n"

sys.exitfunc = cleanup
```

For a more extensive system, use the **atexit** module; see the online documentation for more information.

Interpreter Information

In addition to the functions and structures already seen, the sys module also provides an interface to a number of variables that hold various pieces of information about the current instances of the Python interpreter. These variables are summarized in Table 9-1.

Python sys Variable	Description
builtin_module_names	A tuple containing the names of the all the modules built into the Python executable.
copyright	The copyright message from the current Python interpreter.
exec_prefix	Directory where the platform-dependent Python library files are kept.
executable	Returns the path to the executable file holding the current Python interpreter.
maxint	The largest integer supported by the **integer** object type.
platform	The platform identifier string determined during installation/configuration. For example, 'sunos5' or 'mac'.
prefix	Directory where platform-independent Python library files are kept.
ps1, ps2	The strings containing the primary and secondary prompts used when using the interpreter interactively. By default these are set to >>> and ... respectively.
version	The version of the interpreter running the current script. The return value is a string of the form "2.0 (#71, Oct 22 2000, 22:09:24) [CW PPC w/GUSI2 w/THREADS]". The string contains the major version, minor revision, the build date, and the platform and options configured when the interpreter was built.

Table 9-1. *Interpreter and Other Variables Supported by the* ***sys*** *Module*

Module Search Path

Python modules are looked for within a standard search path that is determined during compilation time. For example, the following list comes from a Python 2.0 installation under Solaris 8:

```
['', '/usr/local/lib/python2.0',
'/usr/local/lib/python2.0/plat-sunos5',
'/usr/local/lib/python2.0/lib-tk',
'/usr/local/lib/python2.0/lib-dynload',
'/usr/local/lib/python2.0/site-packages']
```

The list of directories is stored within the **sys.path** array object. If you want to add other directories to the list of those searched when you use the **import** statement, then simply modify the **sys.path** array:

```
import sys
sys.path.append('/home/mc/lib')
import smtplib
```

To insert the paths so that they get searched in preference to the standard path, use the **insert()** method:

```
import sys
sys.path.insert(0,'/home/mc/lib')
import mymodule
```

Once loaded, you can determine which modules are currently loaded using the **sys.modules** object. This is a dictionary that lists the module name and where it was loaded. For example, here's a list from a Linux installation taken *before* any user modules have been imported:

```
{'os.path': <module 'posixpath' from
'/usr/local/lib/python2.1/posixpath.pyc'>,
'os': <module 'os' from '/usr/local/lib/python2.1/os.pyc'>,
'readline': <module 'readline' (built-in)>,
'exceptions': <module 'exceptions' (built-in)>,
'__main__': <module '__main__' (built-in)>,
'posix': <module 'posix' (built-in)>,
'sys': <module 'sys' (built-in)>,
'__builtin__': <module '__builtin__' (built-in)>,
'site': <module 'site' from '/usr/local/lib/python2.1/site.pyc'>,
'signal': <module 'signal' (built-in)>,
'UserDict': <module 'UserDict' from
'/usr/local/lib/python2.1/UserDict.pyc'>,
```

```
'posixpath': <module 'posixpath' from
'/usr/local/lib/python2.1/posixpath.pyc'>,
'stat': <module 'stat' from '/usr/local/lib/python2.1/stat.pyc'>}
```

Note that the format of each dictionary pair is specific: the key contains the name of the module, and the corresponding value contains information about the module's real name and whether it is a built-in module or loaded from an external file.

Working with the Operating System (os Module)

Interacting with the operating system includes everything from determining the environment in which the interpreter and your script are running, the user and process environment, and also controlling and communicating with external aspects such as files and file systems.

In Python most of this functionality is handled by the **os** module, although some other modules provide generic interfaces to other systems. The **os** module is not itself the provider of functionality—instead it creates the necessary links between the **os** namespace and built-in modules such as **posix** (for Unix/Windows NT and 2000) or **mac**, which provide the real functionality on a system-dependent basis. This allows the **os** module to act as a cross-platform module that provides all of the core APIs for communicating with the host system.

You can determine which of the various OS-dependent modules has been loaded by examining the **os.name** variable. For example:

```
Python 2.0 (#71, Oct 22 2000, 22:09:24)  [CW PPC w/GUSI2 w/THREADS] on mac
Type "copyright", "credits" or "license" for more information.
>>> import os
>>> os.name
'mac'
```

The **os.path** variable holds the name of the module that should be used for handling platform-independent pathname operations. You can import the module required directly using

```
import os.path
```

Manipulating Environment Variables

The **os.environ** dictionary provides access to the environment variables of the current process (the Python interpreter). The **os.environ** variable is actually a mapping object,

which means that we can access the information just like a dictionary, and we can also modify the real environment values using standard dictionary statements and expressions. Changing the environment will affect the Python interpreter and any programs we start from the interpreter through **os.system()** or **os.exec()**, and it will also be used as the environment for any children we create using **os.fork()**.

During startup the contents of **os.environ** are populated with the current environment. You can access the information just as would a standard dictionary:

```
print os.environ['PATH']
```

You can also assign a value to specific keys within **os.environ** to create and/or set the value of an environment variable. For example, to modify the value of the environments **PATH** variable, which defines the list of directories used to search for an external program, we would use

```
os.environ['PATH'] = '/bin:/sbin:/usr/sbin:/usr/bin:/usr/local/bin'
```

Alternatively, you can set the value of an environment variable using the **os.putenv()** function:

```
os.putenv('PATH', 'C:\\Python;C:\\WINNT\\system32;C:\\WINNT')
```

Note that although assigning a value to the mapping object provided by **os.environ** automatically calls **os.putenv()**, the **os.putenv()** function does automatically update **os.environ**. This may mean that changes made to the environment using **os.putenv()** are not reflected in **os.environ** even though the real environment has actually been modified.

Line Termination

Because the **os** module is loaded on a system-dependent basis, it also provides a handy way for setting and providing information that is system dependent. Most of this functionality is handled through the different functions and other system-dependent modules loaded by **os**.

Line termination is a perennial problem with any cross-platform–compatible language. Python solves this by automatically setting the value of the **os.linesep** variable to the correct line termination used by the current platform. The actual value is a string, either the single character newline ('\n') for POSIX systems, carriage return ('\r') for Mac OS machines, or the carriage-return/newline sequence ('\r\n') used by Windows.

The value is used by **print** when outputting information to the screen and by the **readline()** method when reading information from a file. In most situations you won't need to refer to this value. But if you are writing a cross-platform script and using file

objects and **write()** methods to output textual data, then you should use the **os.linesep** value at the end of each statement. For example:

```
file.write('Some other text' + os.linesep)
```

This will ensure that whatever operating system you are currently executing on you will be using the correct line termination.

Process Environment

To get user ID and other process-specific information, you need to use one of a series of functions from the **os** module. A list of these functions is in Table 9-2.

Python Function	Description
chdir(path)	Changes the current working directory to **path**.
getcwd()	Returns the path of the current working directory.
getegid()	Returns the effective group ID. Unix only.
geteuid()	Returns the effective user ID. Unix only.
getgid()	Returns the real group ID. Unix only.
getpgrp()	Returns the ID of the current process group. Unix only.
getpid()	Returns the process ID of the current process. Unix/Windows only.
getppid()	Returns the ID of the parent process. Unix only.
getuid()	Returns the real user ID. Unix only.
putenv(var, value)	Sets the value of the environment variable name **var** to **value**. See "Manipulating Environment Variables" earlier in this chapter for more information. Unix/Windows only.
setgid(gid)	Sets the group ID to **gid**. Unix only. Requires Superuser privileges.
setpgrp()	Creates a new process group for the current process. Returns the ID of the new process group. Unix only. Requires Superuser privileges.

Table 9-2. *Process Information Functions in the* ***os*** *Module*

Python Function	Description
setpgid(pid, pgrp)	Assigns the process ID **pid** to be a member of the group **pgrp**. Unix only. Requires Superuser privileges.
setsid()	Creates a new session and returns the newly created session ID. Sessions are used with terminals and the shells that support them to allow multiple applications to be executed without the use of a windowing system. Unix only.
setuid(uid)	Sets the user ID of the current process. Unix only. Requires Superuser privileges.
strerror(errno)	Returns the error message associated with the error number in **code**. Unix/Windows only.
umask(mask)	Sets the umask of the current process to **mask**. Unix/Windows only.
uname()	Returns a tuple of strings containing the system name, node name, release, version, and machine for the current system. Unix only.

Table 9-2. *Process Information Functions in the* **os** *Module* (continued)

Process Execution and Management

You can execute and manage processes within Python. Execution involves either replacing the Python interpreter, starting a new process or communicating with a process either to read or write to the process, or both. We can also install signal handlers to control how our process reacts when it is sent a signal. Most of this functionality is handled in Python by the **os** module.

Running External Commands

Although we can do most things within Python using some form of module or extension, there are times when we need to communicate with the outside world by running some external command.

The os.exec*() Functions The basic Python function for execution an external command is **execv**. There are also variations on the same function that also allow you to supply a dictionary of environment variables (**execve()**, **execvpe()**) or to search the

environment path in **os.environ['PATH']** for the specified application to execute (**execvp()**, **execvpe()**). The format for each function is shown below:

```
os.execv(path, args)
os.execve(path, args, env)
os.execvp(path, args)
os.execvpe(path, args, env)
```

The **path** should be the full path to the file that you want to execute. When using **os.execvp()** or **os.execvpe()**, the environment path directories are searched. Alternatively you can set a different path using the **os.defpath** variable. The **args** argument should be a list (or tuple) of arguments supplied to the program being called. The **env** argument should be a dictionary of environment variables: these will be used in place of the environment being used by the Python interpreter.

Here are some examples of running all of the above functions:

```
os.execv('/bin/ls', ('-la'))
os.execve('/usr/local/bin/cvs', ('commit'), {'CVSROOT':'/export/cvs'})
os.execvp('ls',('-la'))
os.execvpe('cvs', ('commit'), {'CVSROOT':'/export/cvs'})
```

Note that in *all* cases the **exec*()** series of commands entirely replace the Python interpreter. If you execute the following from within a Python script (without using **os.fork()**), the Python interpreter and your script will terminate and the **ls** program will execute in its place:

```
os.execv('/bin/ls', ('-la'))
```

If you want to start an additional process, you will need to use either **os.fork()** to start a new child process, **os.spawn*()** (Unix/Windows only) or use the **os.system()** command.

In addition to the base versions there are also multiple argument versions that can be used with a variable number of arguments rather than using a tuple or list argument.

os.execl(path, arg0, arg1, ...)	Equivalent to **os.execv(path, (arg0, arg1, ...)**
os.execle(path, arg0, arg1, ..., env)	Equivalent to **os.execve(path, (arg0, arg1, ...), env)**
os.execlp(path, arg0, arg1, ...)	Equivalent to **os.execvp(path, (arg0, arg1, ...))**

In all cases an **OSError** exception is raised if the program in **path** cannot be found, with additional arguments containing the precise error, for example:

```
>>> import os
>>> os.execv('nothing',('',))
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
OSError: [Errno 2] No such file or directory
```

Starting a New Process To start a new process, rather than replace the current process, you need to use the **system()** function:

```
import os
os.system('emacs')
```

The return value from **os.system()** is the exit status of the called program. Under Windows, however, the value returned is always zero. If you want to read the output from a program, then use the Python **os.popen()** function instead (see "Communicating with an External Process").

Alternatively, under Unix and Windows you can use the **os.spawn*()** functions. They work in a similar fashion to the **os.exec*()** function but create a new process, rather than replacing the existing process. There are two forms:

```
os.spawnv(mode, path, args)
os.spawnvw(mode, path, args, env)
```

where **path** is the location of the application you want to spawn, and **args** is a list or tuple containing the arguments for the application. When using **os.spawnve()**, the **env** argument is a dictionary that is used to populate the environment for the spawned application.

In both cases the **mode** is a constant, as defined by the **os** module and listed in Table 9-3.

Communicating with an External Process

To read the output from an external command you need to use the **os.popen()** function. It opens a pipe to a program call so that you can read the information output from the file or write the information to the file. For example, the code below opens a connection to the **ls** external command to get a list of files, which we then read and print out (for a better alternative, see the **glob** module in Chapter 11):

```
import os
dir = os.popen('ls -al','r')
while(1):
    line = dir.readline()
    if line:
        print line,
    else:
        break
```

The **os.popen()** function works exactly the same as the built-in **open** function, returning a file object that we can read from in bytes or lines, as we have here. The actual format for the **os.popen()** function is

```
os.popen(command [, mode [, bufsize]])
```

The **command** should be the string to be executed by the shell, which is started when the function is called. The **mode** should be "r" for reading or "w" for writing; it defaults to "r" if no **mode** is supplied. The **bufsize** is the buffer size to be used when reading from the command, just like the **bufsize** argument to Python's **open()** function. The **os.popen()** function is only supported under Unix and Windows.

Unfortunately the **os.popen()** function works only in one direction; you can only read or write to the program that you called, you cannot read and write to the process. For this, you need the **popen2()** function in the **popen2** module.

The **popen.popen2()** function spawns a new process and returns the child standard output and input filehandles:

```
(child_stdout, child_stdin) = popen2(cmd [, bufsize [, mode]])
```

popen3() returns the standard output, input and error filehandles:

```
(child_stdout, child_stdin, child_stderr)
                          = popen3(cmd [, bufsize, [, mode]])
```

Finally the **popen4()** function returns a combined stdout/stderr filehandle:

Constant	Description
P_WAIT	Executes the program and waits for it to terminate before returning control to the calling program.
P_NOWAIT	Executes the program and immediately returns a process handle.
P_NOWAITO	Identical to **P_NOWAIT**.
P_OVERLAY	Executes the program, replacing the current process—that is, it works like the **exec*()** functions.
P_DETACH	Executes the program and detaches from it; the new process continues to run but cannot use **wait()** to wait for the process to terminate.

Table 9-3. *Modes for Spawning New Applications*

```
(child_stdout_and_error, child_stdin) = popen4(cmd[, bufsize[, mode ] ])
```

Once opened we can read and write to/from **child_stdout**, **child_stdin** or **child_stderr** using the same methods as for a normal file.

The **popen2.Popen3()** function returns an instance of the **Popen3** class:

```
child = Popen3(cmd [, capturestderr [, bufsize]])
```

The **capturestderr** argument if true forces the instance to capture standard error as well as input and output. The default is not to capture standard error. The new instance has the following methods and attributes:

child.poll()	Returns the exit code of the child or –1 if the child is still running
child.wait()	Waits for the child process to terminate, returning the exit code
child.fromchild	The file object that captures the child's standard output
child.tochild	The file object that sends input to the child's standard input
child.childerr	The file object that captures the child's standard error

Finally, there's also a **Popen4** class that works in the same way as the **popen4()** function, returning a combined standard out/standard error filehandle.

```
Popen4(cmd[, bufsize ])
```

Creating Child Processes

Lots of servers and other processes create "children" during their execution. The parent is normally some kind of listening program or service dispatcher—for example a web server—and when it receives a request or network connection it then creates a child process to service the request. In this way it's easier to handle the requests of a number of clients without resorting to a round robin or loop approach to servicing requests.

The operating system **fork()** function creates a new child process. In Python the **os.fork()** function is just an alias for the operating system version. What actually happens when **os.fork()** is called is that the current process is duplicated, and the **os.fork()** function returns zero within the child process while returning the ID of the child process in the parent. Within Python, to handle this just check the return value:

```
pid = os.fork()
if not pid:
    # Start of child process
```

```
else:
    # Continuation of parent process
```

For example, the script below forks off a new process and sends a message:

```
import os,time

pid = os.fork()
if not pid:
    for step in range(10):
        print 'Hello from the child!'
        time.sleep(1)
else:
    for step in range(10):
        print 'Hello from the parent!'
        time.sleep(1)
```

When executed, this script generates the following output:

```
Hello from the parent!
Hello from the child!
Hello from the child!
Hello from the parent!
...
Hello from the parent!
Hello from the parent!
Hello from the child!
Hello from the child!
Hello from the parent!
```

As you can see, from within the same script we are effectively executing two different processes. In essence, we end up with a quasi-multithreaded system for executing multiple processes from within a single process. Note that any open filehandles—including network sockets—are duplicated when **os.fork()** is called. This can be useful in situations where you want to accept data on an open filehandle in the parent but provide the child with the information. Once the process has forked, close the filehandles in the parent and let the child deal with them.

Waiting for Children

When you fork new processes and they eventually die, you will need to wait for the processes to exit cleanly to ensure that they do not remain in the process table as

"zombies." This process is called *reaping*. There are two functions that will wait for a process to terminate, **os.wait()** and **os.waitpid()**.

```
(pid, exitcode) = os.wait([pid])
(pid, exitcode) = os.waitpid(pid, options)
```

The **pid** should be the process ID of the process that you want to wait for. If you do not specify a **pid** to the **wait()** function, the process will wait until all of the children have died. For example, to wait for child processes to terminate at the end of a script you might simply use

```
os.wait()
```

Note that the parent process blocks (pauses) until the process you are waiting for has completed.

If you want to avoid this blocking, you need to use the **waitpid()** function. The **options** argument accepts either zero, which forces normal operation identical to calling **os.wait()**, or the **os.WNOHANG** if you want the call to immediately return. Using **waitpid()** in this manner allows you to call a reaping process as part of a standard loop within the parent. For example:

```
import os

while 1:
    # wait for a connection
    if accepted:
        pid = os.fork()
    if not pid:
        # do the child stuff
    else:
        os.waitpid(pid, os.WNOHANG)
```

In a true multiprocess environment, you'd probably add the process ID to an array and then process each in order.

A much better solution is to create a signal handler. Each child process sends a **SIGCHLD** signal to its parent when it terminates. You can trap this using a signal handler, which then calls **wait()** or **waitpid()** for you. For example, the following signal handler is probably the easiest way to reap old processes:

```
def child_handler(signum, frame):
    os.wait()
```

See "Signals" later in this chapter for more information on how to create and install different signal handlers.

Getting Exit States

Both the **wait()** and **waitpid()** functions return a tuple consisting of the process ID of the child that terminated and its exit status. The exit status is a 16-bit number; the low byte of this is the signal number that killed the process, and the high byte is the exit status.

```
(newpid, exitcode) = os.waitpid(pid, os.WNOHANG)
```

On its own, the **exitcode** is not very useful, but the **os** module defines a number of functions that check the **exitcode** against known values to determine how the child terminated. These functions are **WIFSTOPPED()**, **WIFSIGNALED()**, **WIFEXITED()**, **WEXITSTATUS()**, **WSTOPSIG()**, and **WTERMSIG()** and return true if the **exitcode** matches the given condition.

For example, to check if a child process terminated because it was sent another signal we might use

```
import os

while(1):
    (newpid, exitcode) = os.waitpid(pid, os.WNOHANG)
    if (WIFSIGNALED(exitcode)): break
```

Signals

Signals provide a method for signaling a particular event to a process. They are not a communication mechanism: you cannot transmit information—all you can do is signal a particular event. Signals are used on a variety of operating systems, primarily those that are Unix or POSIX 1003.1 compatible (including Windows NT/2000), and are usually used as a way of managing a process or its execution. For example, the **KILL** signal tells the operating system to terminate the process. Others can be used to initiate a particular function; For example, sending the **HUP** signal to the **named** DNS server forces it to reload its configuration file.

The exact list of signals supported on your system is entirely OS and even OS version dependent. As a rough guide I've included a list in Table 9-4 of POSIX signals that should be supported by most operating systems.

All signals have a default operation: the **SIGSEGV** signal usually produces a core dump of the process under Unix, for example. Some signals can also be trapped, as in the example given for **named** above. For more information on trapping signals see the section "Signal Handlers" later in this chapter. Support under non-Unix operating systems is very limited; for example Windows supports only the ABRT, FPE, ILL, INT, SEGV, and TERM signals.

POSIX 1003.1 Name	Description
SIGABRT	Abnormal termination
SIGALRM	The timer set by the **alarm** function has expired
SIGFPE	Arithmetic exceptions; for example, divide overflow or divide by zero
SIGHUP	Hang-up detected on the controlling terminal or death of a controlling process
SIGILL	Illegal instruction indicating a program error
SIGINT	Interrupt signal (special character from the keyboard or signal from another application)
SIGKILL	Termination signal; cannot be caught or ignored
SIGPIPE	Attempt to write to a pipe with no application reading from it
SIGQUIT	Quit signal (special character from the keyboard or signal from another application)
SIGSEGV	Attempt to access an invalid memory address
SIGTERM	Termination signal (from another application or OS)
SIGUSR1	Application-defined (user-defined) signal
SIGUSR2	Application-defined (user-defined) signal
SIGCHLD	A child process terminated or stopped
SIGCONT	Continue the process if currently stopped
SIGSTOP	Stop signal; stops the specified process
SIGTSTP	Stop signal from special character from keyboard
SIGTTIN	A read was attempted from the controlling terminal by a background process
SIGTTOU	A write was attempted to the controlling terminal by a background process

Table 9-4. *POSIX Signals*

Sending Signals

The Python **os.kill()** function sends a signal to an existing process:

```
kill(pid, signal)
```

The **signal** module contains the symbolic constants for the different signals with each constant in the form **SIG***. For example we could send the alarm signal to the current process using

```
kill(os.getpid(), signal.SIGALRM)
```

See Table 9-4 for a full list of the POSIX signals or see the **signal(5)** manual page under Unix or the MSDN library under Windows for information on which signals are supported on your system.

Signal Handlers

To trap a signal you must install what is known as a *signal handler*. This is just a function that will be called when the signal is received by the process. In Python we install a signal handler using the **signal.signal()** function. It accepts two arguments: the signal to be trapped and the function that will be called:

```
signal.signal(signal, handler)
```

For example, to install a system handler for the **SIGALRM** signal:

```
import signal,time,sys

def alarm_handler(signum, frame):
    print "Wake Up!"
    sys.exit()

signal.signal(signal.SIGALRM, alarm_handler)

signal.alarm(5)
print "Going to sleep... "
time.sleep(10)
print "Now I'm up. "
```

The **signal.alarm()** function installs an alarm timer for the given number of seconds. Once the time is up, the OS automatically sends a **SIGALRM** signal to the process. In this case we set an alarm for 5 seconds, while waiting for 10 seconds. If we run the

script we should get the "Wake Up!" message after about 5 seconds, without seeing the "Now I'm up" message as the script terminates at the end of the signal handler function.

In all cases a signal handler must accept two arguments. The first argument are **signum**, the signal raised when the handler was called, and **frame**, which is a frame object describing the Python execution stack at the point the signal was raised. We'll be looking at frames in more detail in Chapter 24: they are used by the Python interpreter to manage the execution of a script.

If you don't supply a function that accepts two arguments when installing the signal handler, then a **TypeError** exception is raised.

Getting the Current Signal Handler You can get the name of the current signal handler installed for a particular signal using

```
handler = signal.getsignal(signal.SIGALRM)
```

The returned object is callable, so we could immediately invoke the handler using

```
handler()
```

Disabling a Signal Handler The **signal** module defines two standard handlers that can be used to modify the behavior of the signal handling process. The **signal.SIG_IGN** handler forces Python to ignore the specified signal, while the **signal.SIG_DFL** handler causes Python to invoke the default signal handler.

We can ignore a signal like this:

```
signal.signal(SIGALRM, signal.SIG_IGN)
```

or set the signal to its default operation (as defined by the operating system) using

```
signal.signal(SIGQUIT, signal.SIG_DFL)
```

Note that you cannot ignore signals such as KILL. They are always handled by your OS.

User/Group Information

The user and group information stored in the Unix **/etc/passwd** and **/etc/group{s}** files is available through the **grp** and **pwd** modules. In each case the functions raise a **KeyError** if the group does not exist. See Table 9-5 for a list of the supported functions.

For example, to get the home directory and shell for the current user we might use

Function	Description
grp.getgrgid(gid)	Returns a tuple containing the information for the group matching **gid**. The tuple consists of the group name, group password, group id, and a list of the members of the group. For example **grp.getgrgid(0)** returns: **('root', '', 0, ['root'])**.
grp.getgrnam(name)	Identical to **getgrgid()** but looks for a group matching **name**.
grp.getgrall()	Returns a list of tuples, where each tuple contains the information returned by **getgrgid()**.
pwd.getpwuid(uid)	Returns the user information for the user ID **uid**. Information is returned as a tuple containing username, password, user ID, group ID, gecos (full name and/or contact detail), home directory, and shell. For example **pwd.getpwuid(0)** returns **('root', 'x', 0, 1, 'Super-User', '/', '/usr/bin/bash')**.
pwd.getpwnam(name)	Identical to **getpwuid()** but looks for a user matching **name**.
pwd.getpwall()	Returns a list of tuples, where each tuple contains the information returned by **getpwuid()**.

Table 9-5. *Getting User and Group Entries with Python*

```
import pwd,os

pwinfo = pwd.getpwuid(os.getuid())
name = pwinfo[0]
fullname, homedir, shell = pwinfo[4:]
print "You are %s (%s)\nHome Directory is %s\nShell is %s" \
        % (name, fullname, homedir, shell)
```

You might also want to investigate the **getpass** and **crypt** modules, which provide a safe way of getting a password from a terminal and for encrypting a text password into the encrypted version that is stored in the **/etc/passwd** file. For example, the script below prompts and then checks the current user's password:

```
import getpass,pwd,crypt
```

```
password = getpass.getpass()
realpw = pwd.getpwnam(getpass.getuser())[1]
if realpw == crypt.crypt(password,password[:2]):
    print "Password validated"
else:
    print "Password invalid
```


Multithreading

Threads are a relatively new mode of operation for many platforms and programming languages. Threads are essentially a method for executing a number of different functions *simultaneously* within a given process without resorting to using **fork()** (which implies a serious overhead as the process is duplicated) or using less efficient methods such as the **select()** function (see Chapter 10). Before we look at the specifics of threads, though, it's worth taking a look at how modern multitasking operating systems operate.

How Multitasking Works

If you look at a typical modern operating system, you'll see that it's designed to handle the execution of a number of processes simultaneously. The method for employing this is either through *cooperative multitasking* or *preemptive multitasking*. In both cases, the actual method for executing a number of processes simultaneously is the same—the operating system literally switches between applications every fraction of a second, suspending the previous application and then resuming the next one in a round-robin fashion. So, if the operating system has 20 concurrent processes, each one will be executed for a fraction of a second before being suspended again and having to wait for 19 other processes to do their work before getting a chance to work again.

The individual processes are typically unaware of this switching, and the effects on the application are negligible—most applications couldn't care less whether they were working as a single process or as part of a multiprocessing environment, because the operating system controls their execution at such a low level.

The two different types of multitasking—cooperative and preemptive—describe how the operating system controls the applications that are executing. With *cooperative* multitasking, all processes are potentially given the same amount of execution time as all others. What actually happens is that a given process executes until it calls a system function, at which time the OS is able to switch to a different process. This means that it's possible for an application never to relinquish its control over the processor. Some operating systems are more specific and provide the "main" application with the bulk of the processor time (say 80 percent), and the "background" applications with equal amounts of the remainder (20 percent). This is the model used by the Mac OS (but not MacOS X), and it allows the GUI environment to give the most time to the application the user is currently employing.

Preemptive multitasking is much more complex. Instead of just arbitrarily sharing the processor time between all of the processes that are executing, an operating system

with preemptive multitasking gives the most processor time to the process that requires it. The operating system does this by monitoring the processes that are running and manages the execution of each process; those with higher priorities get more time, and those with the lowest priorities get the least. Because we can control the priorities of the processes, we have much greater control over how different processes are executed. On a database server, for example, you'd want to give the database process the highest priority to ensure the speed of the database. Preemptive multitasking is one of the main features of many server-oriented operating systems, including Unix, Linux, and NT-based Windows implementations, including Windows 2000 and NT itself. It's also more commonly used in client-oriented operating systems; Windows 95/98/NT Workstation/2000 Professional, Mac OS X, and BeOS all support preemptive multiprocessing.

The different multitasking solutions also determine the different hardware types that can be used with an operating system. Cooperative multitasking is really only practical on a single-processor system. This is because of the round-robin approach, which requires that the process resides on the same processor for its entire duration. Although it's possible (and indeed frequently common) to find preemptive multitasking on single processor machines, they do not handle as well as you might expect. In fact in some quarters the requirement of a preemptive system has helped to drive processor design— the SPARC, SuperSPARC, and UltraSPARC processors used by Sun have all been specially designed with multitasking in mind. They employ multiple register sets to allow the processor to easily switch between multiple tasks.

With preemptive multitasking, multiprocessor solutions are the best hardware platform. Because the operating system knows how much time each process requires, it can assign individual processes to different processors depending on how busy each processor is. This helps to make the best use of the available processor capacity and to spread the load more effectively. However, the division of labor is only on a process-by-process basis, so if you have one particularly intensive process, it can only be executed on a single processor, even if it there is spare capacity on other processors in the system.

From Multitasking to Multithreading

With a multitasking operating system, there seem to be a number of processes all executing concurrently. In reality, of course, each process is running for a fraction of a second, and potentially many times a second, to give the impression of a real multitasking environment with lots of individual processors working on their own application.

For each process, there is an allocation of memory within the addressing space supported by the operating system that needs to be tracked, and for multiuser operating systems, such as Unix, there are also permission and security attributes and, of course, the actual code for the application itself. Tracking all of this information is a full-time job— under Unix there are a number of processes that keep an eye on all of this information, in addition to the core kernel process that actually handles many of the requests.

If an individual process wants to be able to perform a number of tasks concurrently, there are two possible solutions. The first solution is a round-robin approach, as used by the main operating system, but without the same level of control. Each function that needs to be executed is called in sequence in a loop, but because we can't arbitrarily terminate a function mid-execution, there are often problems with "lag-time"—if a function has a large amount of information to process, then its execution will hold up the entire loop.

For file processing, you can get around this by using **select** and parsing fixed blocks of information for each file. In this instance, we only process the information from the files that have supplied (or require) more data, and provided we read only a single line or a fixed-length block of data, the time to process each request should be relatively small.

For solutions that require more complex multitasking facilities, the only other alternative is to **fork** a new process specifically to handle the processing event. Because **fork** creates a new process, its execution and priority handling can be controlled by the parent operating system. This is usually the solution used by network services, such as Apache and IMAP or POP3 daemons. When a client connects to the server, it forks a new process designed to handle the requests of the client.

The problem with forking a new process is that it is a time-consuming and very resource-hungry process, and it doesn't really solve the problem: it gives the OS yet another process to manage and allot time to. Creating a new process implies allocating a new block of memory and creating a new entry in the process table used by the operating system's scheduler to control each process's execution. To give you an idea of the resource implications, a typical Apache process takes up about 500K. If 20 clients connect all at the same time, it requires the allocation of 10MB of memory and the duplication of the main image into each of the 20 new processes.

In most situations, we don't actually need most of the baggage associated with a new process. With Apache, a forked process doesn't need to read the configuration file—it's already been done for us, and we don't need to handle any of the complex socket handlers. We need only the ability to communicate with the client socket we are servicing.

This resource requirement puts unnecessary limits on the number of concurrent clients that can be connected at any one time—it is dependent on the available memory and ultimately the number of processes that the operating system can handle. The actual code required to service the client requests could be quite small, say 20K. Using multiprocessing on a system with 128MB might limit the number of clients to around 200—not a particularly large number for a busy website. To handle more requests than that, you'd need more memory and probably more processors: switching between 200 processes on a single CPU is not recommended because the amount of time given to each process during a single pass (executing each process once) would be very small, and therefore it could take minutes for a single process to service even a small request.

This is where threads come in. A thread is like a slimmed-down process—in fact they are often called "lightweight processes." The thread runs within the confines of the parent process and normally executes just one function from the parent. Creating a new thread doesn't mean allocating large areas of memory (there's probably room

within the parent's memory allocation) or require additions to the operating system's schedule tables, either. In our web server example, rather than forking a new process to handle the client, we could instead create a new thread using the function that handles client requests.

By using multithreading, we can therefore get the multiprocessing capability offered by the parent operating system, but within the confines of a single process. Now an individual process can execute a number of functions simultaneously, or alternatively execute the same function a number of times, just as you would with our web server.

On an operating system that supports preemptive multitasking and multithreading, we get the prioritizing system on the main process and an internal "per-process" multitasking environment. On a multiprocessor system, most operating systems will also spread the individual threads from a single process across all of the processors. So, if we have one particularly intensive process, it can use all of the available resources and processors by splitting its operation into a number of individual threads. We no longer get into a situation where one process is tied to one processor, or into the situation where we have to create large numbers of duplicates to service the requests of a number of machines.

Threading is, of course, very OS-specific. Even now, there are only a handful of operating systems that provide the functionality to a reasonable level, and some require additional or different libraries to enable the functionality. Most of the operating systems that support threading are either Unix based (Solaris, AIX, HP-UX, some Linux distributions, BSD, Mac OS X) or Windows based (Windows 98/NT/2000/Me), but there are exceptions such as BeOS and even Mac OS.

Comparing Threads to Multiple Processes

The major difference between multithreaded and multiprocess applications is directly related to the relative resource cost, which we've already covered. Using **fork** to create duplicate instances of the same process requires a lot of memory and processor time. The overhead for a new thread is only slightly larger than the size of the function you are executing, and unless you are passing around huge blocks of data between threads, it's not inconceivable to be able to create hundreds or on a suitable system thousands of threads within a single process.

The only other difference is the level of control and communication that you can exercise over the threads. When you **fork** a process, you are limited in how you can communicate with the child. To exchange information, you'll need to open pipes to communicate with your children and this becomes unwieldy with a large number of children. If you simply want to control the children, you are limited to using signals to either kill or suspend the processes—there's no way to reintegrate the threads into the main process, or to arbitrarily control their execution without using signals.

Comparing Threads to select()

The **select** function provides an excellent way of handling the data input and output from a number of filehandles concurrently, but this is where the comparison ends. It's

not possible, in any way, to use **select** for anything other than communicating with filehandles, and this limits its effectiveness for concurrent processing.

On the other hand, with threads you can create a new thread to handle any aspect of your process's execution, including, but not limited to, communication with filehandles. For example, with a multidiscipline calculation you might create new threads to handle the different parts of the calculation.

Threads and Python

Python supports threads on Windows, Solaris, BeOS, some Linux variants, and most other Unix variants that support the POSIX thread library (pthread). By default, threads are disabled in the Python source distribution: you'll need to manually add support by modifying the Python **Setup** file in the Extensions directory. Check the installation files that come with Python for more information.

The scheduling of threads and thread switching are controlled by an interpreter lock that controls thread execution. This ensures that only a single thread can be executed by the current instance of the interpreter at any time. We can also only switch between threads between the execution of individual byte codes. (See Chapter 23 for more information on byte code and how it affects execution.) You can control the number of bytecode instructions that are executed for each thread before execution switches to the next thread using the **sys.setcheckinterval()** function. The default is 10 bytecode instructions.

Also be careful when using Python extensions written in C/C++ that may or may not be written to be thread compatible. If they are not thread compatible, the process will block until the function call has completed, irrespective of the **sys.setcheckinterval()** settings.

You should also be aware of the effects of using signals when employing threads. Signals handlers can be caught and executed only by the main thread, not by any additional threads in the script. This obviously also implies that you cannot send a signal to an individual thread.

Basic Threads

The **thread** module provides a very simplified method for creating threads and managing locks on data structures. All threads in Python work by creating a new thread whose sole responsibility is to execute the function they are supplied with during execution. This means that if you have a piece of code that you executed within a separate thread, you will need to place it into a function before you create a thread to execute.

Creating New Threads

The core function is **start_new_thread()**:

```
start_new_thread(func, args [,keywordargs])
```

The function creates a new thread that starts executing the function **func**, which is called using the arguments **args** and the keyword arguments **keywordargs** by using the **apply()** function. The thread starts immediately; if you want to delay the execution of a given thread, you can do this using the **threading** module, covered later in this chapter in the "Advanced Threads" section. If there is an error in the function and the exception has not been handled, then a stack trace is printed and the thread exits, but other threads, including the parent, continue to execute. Exceptions are not propagated up to the parent.

However, when the parent thread terminates, the continued execution of the threads is entirely OS dependent. Some will allow the threads to continue until they reach the end of their function. Others will kill all the threads and terminate the entire program. Check your system's documentation to see how it treats parentless threads.

For example, here's a script that displays the time, updated every 10 seconds, using a thread to handle the update process.

```
import thread, time

def display_time(interval, prefix=''):
    while 1:
        print prefix,time.ctime(time.time())
        time.sleep(interval)

thread.start_new_thread(display_time, (10, 'The is now'))
while 1:
    pass
```

The function itself is just a loop that prints out the time (using an optional prefix) before pausing for the specified interval. Note that the arguments to the thread function have to be supplied as a tuple. We also have to set up a loop in the main thread that does nothing so that the parent process doesn't die, as this would also kill off the thread we created.

Thread Control

To terminate a thread, you either let the thread's function terminate with a return, or, to force termination, you can call the **thread.exit()** function. Also be aware that the **sys.exit()** function *when called from within a thread* calls the **thread.exit()** function, which terminates only the thread, not the entire script.

All threads are also given a unique integer identity number that you can determine within a thread using the **get_ident()** function. This is especially useful if you want to log thread execution. (We'll see an example of this when we look at object access across threads.) Note that thread IDs are not sequential and there is no relation between the threads of a process and the process ID.

Object Locking

Threads share and have access to all of the global objects and data structures within the parent, so you must use extreme caution when updating these structures from multiple threads. Although it's difficult to demonstrate, the effects of updating the same data structure at the same time in two different threads is unlikely to achieve the desired result. Although there are different methods of controlling access to individual variables within the thread system, they all rely on the same basic premise of a "lock." This is a thread-safe object that can be used to control access to another object from within multiple threads. The **thread** module provides only a very simple object locking mechanism, called a *mutex*, for mutual exclusion; for more advanced methods see "Advanced Threads" later in this chapter.

The **allocate_lock()** function creates a lock object (of type **LockType**) that can be used to set or determine the lock for a given object. The relationship between the lock and the object is not automatic; you need to know which lock object relates to each variable before using the system. For example, if we create a variable **counter** we might create a **counter_lock** object that we'll use to control access to that variable:

```
counter = 0
counter_lock = thread.allocate_lock()
```

If we want to make changes to **counter** we need to first acquire the lock handled by **counter_lock**. The **acquire()** method attempts this operation for us:

```
lock.acquire([waitflag])
```

By default **waitflag** is zero, indicating that the method should return immediately whether the lock could be acquired or not. If **waitflag** is nonzero then the thread will block until the lock has been acquired. In either case, you *must* supply a value if you want to check its return status. The method returns zero if the lock could not be acquired and 1 if it could. For example, to acquire a lock and then change the value of our counter we could use

```
if (counter_lock.acquire(0)):
    counter += 1
```

The **counter_lock** is now locked and requests by other threads to acquire the lock will always fail. Although it's possible for other threads to still modify the **counter** object, they would be doing so outside of the authority of the lock and could corrupt the information in the object. Mutexes are essentially just a protocol; all the threads need to follow the protocol for the system to work. To actually release the lock we need to

use the **release()** method on our **counter_lock** object so that other threads can acquire the lock and change the value:

```
counter_lock.release()
```

If you want to check the status of a lock without using the **acquire()** method you can also use the **locked()** method. This returns 1 if the object is locked or zero if not.

Putting this all together, the script below is a simple counter with three threads. The first thread is the display thread that simply outputs the current counter value without making modifications. There are also two threads, A and B, that acquire the lock on a variable, increase its value and then hold on to the lock for a short period before releasing it.

```
import thread, time

counter = 0

counter_lock = thread.allocate_lock()

def thread_incra():
    while 1:
        print "Thread A(%d): Executing" % (thread.get_ident(),)
        if (counter_lock.acquire(0)):
            global counter
            print "Thread A: Acquired lock...pausing..."
            counter += 1
            time.sleep(20)
            counter_lock.release()
        else:
            print "Thread A: Couldn't get lock"
        time.sleep(5)

def thread_incrb():
    while 1:
        print "Thread B(%d): Executing" % (thread.get_ident(),)
        if (counter_lock.acquire(0)):
            global counter
            print "Thread B: Acquired lock...pausing..."
            counter += 5
            time.sleep(10)
            counter_lock.release()
        else:
            print "Thread B: Couldn't get lock"
        time.sleep(5)

def thread_display():
```

```
    while 1:
        print "Display thread(%d): Counter is %d" % (thread.get_ident(), counter)
        time.sleep(5)

thread.start_new_thread(thread_display, ())
thread.start_new_thread(thread_incra, ())
thread.start_new_thread(thread_incrb, ())
while 1:
    pass
```

If we execute the script we get the following output:

```
Display thread(1026): Counter is 0
Thread A(2051): Executing
Thread A: Acquired lock...pausing...
Thread B(3076): Executing
Thread B: Couldn't get lock
Thread B(3076): Executing
Thread B: Couldn't get lock
Display thread(1026): Counter is 1
Thread B(3076): Executing
Thread B: Couldn't get lock
Display thread(1026): Counter is 1
Thread B(3076): Executing
Display thread(1026): Counter is 1
Thread B: Couldn't get lock
Display thread(1026): Counter is 1
Thread B(3076): Executing
Thread B: Acquired lock...pausing...
Thread A(2051): Executing
Thread A: Couldn't get lock
Display thread(1026): Counter is 6
Display thread(1026): Counter is 6
Thread A(2051): Executing
Thread A: Acquired lock...pausing...
Display thread(1026): Counter is 7
Thread B(3076): Executing
Thread B: Couldn't get lock
Display thread(1026): Counter is 7
Thread B(3076): Executing
Thread B: Couldn't get lock
```

You can see here how first thread A acquires the lock, updates the counter, and then holds on to the lock while thread B continues to try to execute and acquire the lock—then the process is reverse as thread B gets the lock and thread A fails to get the lock. Simultaneously, the display thread goes on outputting the current counter value.

Advanced Threads

Although the **thread** module should suffice for most thread implementations, it will not suit everybody. There is no way in the **thread** module, for example, to control the operation or execution of the threads from within the parent. It's also limited in the way in which we can control access to variables, including the way we communicate between threads without using a global variable or the resources or namespace of the parent thread.

The **threading** module solves this by providing a more advanced method of initiating a new thread and a number of lock systems, semaphores, and other interthread communication systems to better help you control and communicate between your parent and threads.

We'll have a look at the specifics shortly; all of the solutions are based on a number of new object classes, but there are general utility functions within the module to allow you to monitor the current situation. These functions are

activeCount()	Returns the number of currently active **Thread** objects.
currentThread()	Returns the **Thread** object for the currently executing thread. This can be useful within a thread as a method of determining information about the thread from within itself.
enumerate()	Returns a list of all the currently active **Thread** objects, regardless of their execution status.

The **threading** module and its classes are actually built on top of the main **thread** module. Although **threading** doesn't support any different solutions for the creation of threads or locking mechanisms, what it does do is augment the basic methods to support the functionality offered by many OS/language interfaces.

Thread Objects

The core of the system is the **Thread** class. This defines a separate thread and is used to execute a function in the same way as the **thread.start_new_thread()** function. The basic format for creating a new Class instance is

```
Thread(group=None, target=None, name=None, args=(), kwargs())
```

The **target**, **args**, and **kwargs** arguments are identical the **func**, **args**, and **keywordargs** arguments to **thread.start_new_thread()**. The **group** argument is reserved for future extension and for the moment can be safely ignored. The **name** argument is the name that you want to give to your thread. Names are merely notational and have no bearing on the execution or control of your threads. The default name is "Thread-N" where N is a sequential number.

For example, we could create a new **Thread** object for our counter thread in the earlier example using

```
my_display_thread = Thread(None, thread_display, None, (), {})
```

The result is a **Thread** object that supports the following methods:

t.start()	When a thread is created it is not immediately executed; you must call the **start()** method for the function to actually be executed. What actually happens is that **start()** calls the threads **run()** method, which itself calls the function you specified when the **Thread** object was created.
t.run()	Called when the thread starts. By default this calls the function specified when the object was created but you can also overload this method with the statements you want executed in new classes based on **Thread**.
t.getName()	Gets the name of the thread.
t.setName()	Sets the name of the thread.
t.isAlive()	Returns 1 if the thread is alive (that is, currently executing) or zero otherwise. Threads in which the **start()** method has not been called or that have terminated will return zero.
t.setDaemon()	You can daemonize a thread by calling the **setDaemon()** method *before* calling **start()**. Daemonized threads continue to execute in the same way that a daemonized program (as controlled by the **os.setpgprp()** function) continues to execute after the main program has finished. Using **setDaemon()** you can therefore start a multithreaded server just like the **named** or **httpd** daemons under Unix, or in a similar way to services under Windows NT/2000.

Lock Objects

The **threading** module augments the basic lock mechanism with two new lock objects, the primitive lock and the reentrant lock.

Primitive Locks Primitive locks work in the same way as the basic lock mechanism in **thread**. They have two states, lock or unlocked, and you can acquire or release a lock using the **acquire()** and **release()** methods. To create a new primitive lock:

```
mylock = Lock()
```

The **acquire()** works in a similar way to the **acquire()** method on a **thread** lock, but by default the function blocks (as opposed to returning instantly). To change this behavior, supply a single argument of zero to the function. As before, the method returns 1 if the operation was successful or zero if the lock could not be acquired.

```
mylock.acquire(1)
```

To release the lock, use the **release()** method:

```
mylock.release()
```

Reentrant Locks A reentrant lock is to the **Lock** class, but the same lock can be acquired and released a number of times within the same thread. Normally, once a lock has been acquired by a thread it cannot be acquired again until the lock has been released. Using a reentrant lock, we can release and acquire a lock within a nested set of statements. Only the outermost **release()** operation resets the lock to its unlocked state.

The way the system works is that calls to **acquire()** increment the lock state. If the lock is already acquired by the current thread, then calling **acquire()** again only increments the count and returns immediately with a true value to the caller. When you call **release()**, the counter is decremented—it's only when the counter reaches zero that the lock is properly relinquished.

To create a reentrant lock use the **RLock** class:

```
myrlock = RLock()
```

The **acquire()** and **release()** methods work the same as for the **Lock** class.

Condition Variables

Condition variables are a more advanced form of the locking mechanisms already described. Rather than simply providing a lock mechanism, a condition variable can be used to indicate a particular change of state or event to a thread. For example, in a situation where one thread generates output for another thread to process, the processing thread may be waiting to access the data in a shared variable. Meanwhile, the producer thread has to wait for the processor thread to complete before adding new data.

Using a simple **Lock** or **RLock** class in this case would be complex. The condition variable gets around this by simplifying the process for determining the state and for notifying a waiting thread that the variable is available to use.

You create a new condition variable using the **Condition** constructor:

```
mycondition = Condition([lock])
```

The optional **lock** argument should be an instance of the **Lock** or **RLock** classes. If none is specified, then a new **RLock** instance is created. The new object supports the following methods:

mycondition.acquire(*args)	Acquires the underlying lock, calling the correct **acquire()** method according to the underlying lock. The **args** if supplied are passed verbatim to the lock's **acquire()** method.
mycondition.release()	Releases the underlying lock.
mycondition.wait([timeout])	Waits for notification from another thread either for the period specified in **timeout** or indefinitely. Once called, the underlying lock is released, and the thread pauses until it receives a **notify()** or **notifyAll()** event from another thread for the given lock. Once the event has been received, the lock is reacquired and the method returns. If the operation times out, the lock is reacquired and execution continues as normal.
mycondition.notify()	Sends a notification event to a thread currently **waiting**. Note that the operation does not actually release the lock; you need to call both **notify()** and **release()** to allow the other thread to continue.
mycondition.notifyAll()	Notifies all threads waiting.

As an example, the simple script below passes messages from one thread to another to be printed. We use **notify()** in the sending thread to indicate to a waiting thread that a new message is ready to printed. Obviously this is a simplistic example, but it demonstrates how easily we can communicate the status of a variable between two threads.

```
import threading, time

mycondition = threading.Condition()
mymessage = ''

def thread_send():
    global mymessage
    counter = 0
    while 1:
        mycondition.acquire()
```

```
            mymessage = 'New Message: '+str(counter)
            counter += 1
            mycondition.notify()
            mycondition.release()
            time.sleep(5)

def thread_receive():
    global mymessage
    while 1:
        mycondition.acquire()
        if mymessage:
            print "Display:",mymessage
            mymessage = ''
            mycondition.wait()
        mycondition.release()

threada = threading.Thread(None, thread_send, None)
threadb = threading.Thread(None, thread_receive, None)

threada.start()
threadb.start()

while 1:
    pass
```

Semaphores

A semaphore is a basic counter-based locking mechanism. Calls to **acquire()** decrement the counter and calls to **release()** increment the counter. If the counter ever reaches zero, then the **acquire()** method blocks until another thread calls **release()**. Semaphores can be used to indicate a particular status for a given thread or other object. For example, you might use a semaphore in conjunction with an array to indicate how many items are in the array—that way, once the number of items in the array reaches zero you can block a thread from executing and removing information from the array.

To create a semaphore use the **Semaphore** class:

```
mysemaphore = threading.Semaphore([value])
```

The option **value** argument indicates the initial value for the semaphore; the default is 1. The value should be a positive integer.

mysemaphore.acquire([blocking])	The **acquire()** method attempts to acquire the semaphore. If the semaphore's counter is larger than zero, **acquire()** decrements the count and returns immediately. If the counter is already at zero, then the method blocks until another thread calls **release()**. The **blocking** argument works as for other locks.
mysemaphore.release()	Increases the semaphore's internal counter. If the counter is zero, calling **release()** unblocks a waiting thread.

Events

Events are used to communicate between threads by signaling a specific event that other threads can wait for. An event object has an internal flag that can be set or cleared at will. In addition, a **wait()** method allows you to wait until the flag is set. To create an event object:

```
myevent = Event()
```

The internal flag is initially set to false. The methods supported by an **Event** instance are

myevent.isSet()	Returns true if the internal flag is true.
myevent.set()	Sets the internal flag to true.
myevent.clear()	Clears the internal flag, setting it to false.
myevent.wait([timeout])	Waits for the internal flag of **myevent** to become true. The methods waits indefinitely, or until the **timeout** (specified as a floating point value, treated as seconds) expires. Obviously if the flag is already set to true then the method returns immediately.

Queues

Although the different messaging methods we have already seen work adequately in most situations, they do not resolve all of issues, especially when dealing with threads that need to communicate large volumes of data between one another. Although we can share objects using one of the methods shown above (effectively sharing information), especially if the provider and consumer threads work at different rates, or deal with different volumes of data, then we end up in contention situations that become difficult to resolve.

The **Queue** module provides a solution. It provides a class that sets up a multiprovider and multiconsumer FIFO (First In, First Out) queue. Using the object we can add new requests to the queue, and process requests from the queue without worrying about processing rates, job sizes, or other limitations, and also proceed without the problems of manually dealing with object locks.

To create a new queue, we need to create an instance of the **Queue** class:

```
myqueue = Queue(maxsize)
```

The **maxsize** defines the maximum number of items that can be placed into the queue. If the **maxsize** is less than or equal to zero, then there is no upper limit. Choosing a suitable queue size is important, as choosing the wrong size could mean you end up with too few spaces available for new objects; or if the value is set very high, or infinite, then you run the risk of allowing your threads to consume ever increasing amounts of resource.

Although there are ways to control how many items are supplied to the queue it's probably best to size it according to about half the size of maximum number of items you expect to process. For example a program that processes about 10,000 elements should probably set a **maxsize** of about 5000. If the size of your queue elements is particularly large, then factor the size according to the maximum amount of space you want to allocate to your application.

An instance of **Queue** supports the following methods:

myqueue.qsize()	Returns the approximate size of the queue; an exact figure will always be difficult to determine since other threads may be updating the queue.
myqueue.empty()	Returns 1 if the queue is empty or zero if not.
myqueue.full()	Returns 1 if the queue is full or zero if not.
myqueue.put(item [, block])	Puts the object **item** into the queue. If **block** is supplied and is a positive value, then the caller blocks until a free slot is available. If not supplied, or if **block** is zero, then a **Full** exception is raised if the queue is full.
myqueue.put_nowait(item)	Equivalent to **myqueue.put(item, 0)**.
myqueue.get([block])	Gets an item from the queue. If **block** is supplied and a positive value, then the caller blocks until an item is available. If not supplied, or if **block** is zero, then an **Empty** exception is raised if the queue is empty.
myqueue.get_nowait()	Equivalent to **myqueue.get(0)**.