

Tutorial on Threads Programming with Python

Norman Matloff and Francis Hsu*
University of California, Davis
©2003-2005, N. Matloff

December 9, 2005

Contents

1	Why Use Threads?	3
2	What Are Threads?	3
2.1	Processes	3
2.2	Threads Are Process-Like, But with a Big Difference	4
3	Thread Managers	4
3.1	Kernel-Level Thread Managers	5
3.2	User-Level Thread Managers	5
3.3	Comparison	5
3.4	The Python Thread Manager	5
4	Python Threads Modules	6
4.1	The <code>thread</code> Module	6
4.2	The <code>threading</code> Module	11
5	Condition Variables	14
5.1	General Ideas	14
5.2	Event Example	15
5.3	Other <code>threading</code> Classes	16

*Francis, a graduate student, wrote most of Section 7.

6	The Effect of Timesharing	17
6.1	Code Analysis	18
6.2	Execution Analysis	19
7	The Queue Module	20
8	Debugging Threads with PDB	23
8.1	Using PDB	23
8.2	RPDB	24
9	Non-Pre-emptive Threads in Python	25

1 Why Use Threads?

Threads play a major role in applications programming today. For example, most Web servers are threaded, as are many Java GUI programs.

Here are the major reasons for using threads:

- parallel computation:

If one has a multiprocessor machine and one's threading system allows it, threads enable true parallel processing, with the goal being substantial increases in total processing speed. Threading has become the standard approach to programming on such machines.

- parallel I/O:

I/O operations are slow relative to CPU speeds. A disk seek, for instance, takes milliseconds, while a machine instruction takes nanoseconds. While waiting for a seek to be done, we are wasting CPU time, when we could be executing literally millions of machine instructions.

By putting each one of several I/O operations in a different thread, we can have these operations done in parallel, both with each other and with computation, which does use the CPU.

Most Web servers, for example, are threaded. This allows operations for many different clients to be performed in parallel.

- asynchronous I/O:

Many threaded applications is that they deal with **asynchronous** actions. In a Web server which is serving many accesses simultaneously, we don't know which one will complete next. In a GUI program, we may not know whether the user's next action will be to use the mouse or use the keyboard. By having a separate thread for each action—a separate thread for each Web access, separate threads for mouse and keyboard, etc.—we may be able to write code which is clearer, more convenient and more efficient.

- clarity of program code:

In some cases, program code may be clearer by writing it in a threaded manner, with each thread handling a separate aspect of the program.

- modularity/reusability of program code:

Threading may help make code more modular and reusable.

2 What Are Threads?

2.1 Processes

If your knowledge of operating systems is rather sketchy, you may find this section useful.

Modern operating systems (OSs) use **timesharing** to manage multiple programs which appear to the user to be running simultaneously. Assuming a standard machine with only one CPU, that simultaneity is only an illusion, since only one program can run at a time, but it is a very useful illusion.

Each program that is running counts as a **process** in UNIX terminology (or a **task** in Windows). Multiple copies of a program, e.g. running multiple simultaneous copies of the **vi** text editor, count as multiple processes. The processes “take turns” running. One process might run for a few milliseconds, then be interrupted by the timer hardware, causing the OS to run. The OS saves the current state of the interrupted process so it can be resumed later, then selects the next process to give a turn to. This is known as a **context switch**; the context in which the CPU is running has switched from one process to another. This cycle repeats. Any given process will keep getting turns, and eventually will finish. A turn is called a **quantum** or **timeslice**.

The OS maintains a **process table**, listing all current processes. Each process will be shown as currently being in either Run state or Sleep state. Let’s explain the latter first. Think of an example in which the program reaches a point at which it needs to read input from the keyboard. That might take a long time, so the OS ends this turn prematurely, and marks the process as being in Sleep state.¹ So, being in Sleep state means that the process is waiting for some event to occur; we say that the process is **blocked**.

Being in Run state does not mean that the process is currently running. It merely means that this process is ready to run, i.e. eligible for a turn. Each time a turn ends, the OS will choose one of the processes in Run state to be given the next turn. If a process is in Sleep state but the event it was waiting for occurs, the OS will change its state to Run.

If you wish to get more information on processes in operating systems, see <http://heather.cs.ucdavis.edu/~matloff/50/PLN/OSOverview.pdf>.

2.2 Threads Are Process-Like, But with a Big Difference

A thread is like a UNIX process. In fact, threads are sometimes called “lightweight” processes, because threads occupy much less memory, and take less time to create, than do processes.

But a major difference between processes and threads is that although each thread has its own local variables, just as is the case for a process, the global variables of the parent program are shared by all threads, and serve as the main method of communication between the threads.²

One of the implications of this is that environmental variables, such as current working directory, are shared by all threads. If one thread changes such a variable, then the other threads are subject to the new value.

3 Thread Managers

The thread manager acts like a “mini-operating system” Just like a real OS maintains a table of processes, a thread system’s thread manager maintains a table of threads. When one thread gives up the CPU, or has its turn pre-empted (see below), the thread manager looks in the table for another thread to activate. Whichever thread is activated will then resume execution at the line at which it had left off, i.e. the line at which it had relinquished or lost control.

Just as a process is either in Run state or Sleep state, the same is true for a thread. A thread is either ready to be given a turn to run, or is waiting for some event.

¹Note that the reason the OS even gets a chance to run here is that the program made a call to the OS to do the keyboard input.

²It is possible to share globals among UNIX processes, but very painful.

In a compiled language like C or C++, the thread manager (or at least the upper level of it) is in a library, which is linked to the application program. There are many such libraries in use on various platforms. In the UNIX world, many of these are popular public-domain (i.e. free) packages such as **pthread**s and **pth**. The Windows OS has a proprietary threads system, but it is rather similar to **pthread**s.

In an interpreted language, the threads manager is in the interpreter itself. The manager may in turn call a more basic C/C++ threads library, making the picture more complex. We will not pursue this point here, but it should be kept in mind that this means that threads behavior on so-called “platform-independent” languages like Java or Python may in fact depend quite a bit on the underlying platform.

3.1 Kernel-Level Thread Managers

Here each thread really is like a process, and for example will show up on UNIX systems when one runs the **ps** process-list command. The threads manager is the OS. The different threads set up by a given application program take turns running, just like processes do. When a thread is activated, it is given a timeslice, at the end of which a hardware interrupt from a timer causes control of the CPU to be transferred to the thread manager; we say that the thread has been **pre-empted**. Or, if a thread reaches a point at which it needs to wait for some event, it will voluntarily relinquish its timeslice.

This kind of thread system is used in the UNIX **pthread**s system, as well as in Windows threads.

3.2 User-Level Thread Managers

User-level thread systems are “private” to the application. Running the **ps** command on a UNIX system will show only the original application running, not all the threads it creates. Here the threads are not pre-empted; on the contrary, a given thread will continue to run until it voluntarily gives up control of the CPU, either by calling some “yield” function or by calling a function by which it requests a wait for some event to occur.³

3.3 Comparison

Kernel-level threads have the advantage that they can be used on multiprocessor systems, thus achieving true parallelism between threads. This is a major advantage.

On the other hand, in my opinion user-level threads also have a major advantage in that they allow one to produce code which is much easier to write, is easier to debug, and is cleaner and clearer. This in turn stems from the non-preemptive nature of user-level threads; application programs written in this manner typically are not cluttered up with lots of lock/unlock calls (details on these below), which are needed in the pre-emptive case.

3.4 The Python Thread Manager

In a sense, Python threads are a blend of the kernel- and user-level approaches, though much more of the latter than the former. The Python interpreter keeps track of how long the current thread has executing, in

³In typical user-level thread systems, an external event, such as an I/O operation or a signal, will also cause the current thread to relinquish the CPU.

terms of the number of Python **byte code** instructions have executed.⁴ When that reaches a certain number, by default 100, another thread will be given a turn.⁵ Such a switch will also occur if a thread reaches an I/O statement. Thus Python threads are semi-pre-emptive.

On the other hand, Python does “piggyback” on top of the OS’ underlying threads system (or other external threads system). In other words, Python does make use of the services of that latter system.

I mentioned above that non-pre-emptive threading is nice because one can avoid the code clutter of locking and unlocking (details of lock/unlock below). In Python’s case, it will not quite be so simple. There is some degree of atomicity: The interpreter will treat any Python virtual machine instruction as indivisible, thus not needing locks in that case. But the bottom line will be that unless you know the virtual machine well, you should use locks at all times.

Python maintains a Global Interpreter Lock to ensure that only one thread has access to any Python object in a program at a time. Among other things, this means that one cannot actually achieve true parallelism when one uses Python threading on a multiprocessor machine. One can, though, achieve “parallel I/O,” e.g. do network operations simultaneously, is a very common application of threads.

4 Python Threads Modules

Python threads are accessible via two modules, **thread.py** and **threading.py**. The former is more primitive, thus easier to learn from.

4.1 The thread Module

The example here involves a client/server pair.⁶ It does nothing useful, but is a simple illustration of the principles. We set up two invocations of the client; they keep sending letters to the server; the server concatenates all the letters it receives.

Here is the client, **clnt.py**:

```
1 # simple illustration of thread module
2
3 # two clients connect to server; each client repeatedly sends a letter k,
4 # which the server appends to a global string v and reports it to the
5 # client; k = '' means the client is dropping out; when all clients are
6 # gone, server prints the final string v
7
8 # this is the client; usage is
9
10 #   python clnt.py server_address port_number
11
12 import socket # networking module
13 import sys
14
```

⁴This is the “machine language” for the Python virtual machine.

⁵This number is settable, via a call to **sys.setcheckinterval()**.

⁶It is preferable here that the reader be familiar with basic network programming. See my tutorial at <http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf>. However, the comments preceding the various network calls would probably be enough for a reader without background in networks to follow what is going on.

```

15 # create Internet TCP socket
16 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
17
18 host = sys.argv[1] # server address
19 port = int(sys.argv[2]) # server port
20
21 # connect to server
22 s.connect((host, port))
23
24 while(1):
25     # get letter
26     k = raw_input('enter a letter:')
27     s.send(k) # send k to server
28     # if stop signal, then leave loop
29     if k == '': break
30     v = s.recv(1024) # receive v from server (up to 1024 bytes)
31     print v
32
33 s.close() # close socket

```

And here is the server, **svr.py**:

```

1 # simple illustration of thread module
2
3 # multiple clients connect to server; each client repeatedly sends a
4 # letter k, which the server adds to a global string v and echos back
5 # to the client; k = '' means the client is dropping out; when all
6 # clients are gone, server prints final value of v
7
8 # this is the server
9
10 import socket # networking module
11 import sys
12
13 import thread
14
15 # note the globals v and nclnt, and their supporting locks, which are
16 # also global; the standard method of communication between threads is
17 # via globals
18
19 # function for thread to serve a particular client
20 def serveclient(c):
21     global v,nclnt,vlock,nclntlock
22     while 1:
23         # receive letter from client, if it is still connected
24         k = c.recv(1)
25         if k == '': break
26         # update v in an atomic manner
27         vlock.acquire()
28         v += k
29         vlock.release()
30         # send new v back to client
31         c.send(v)
32     c.close()
33     nclntlock.acquire()
34     nclnt -= 1
35     nclntlock.release()

```

```

36
37 # set up Internet TCP socket
38 lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
39
40 port = int(sys.argv[1]) # server port number
41 # bind lstn socket to this port
42 lstn.bind('', port)
43 # start listening for contacts from clients (at most 2 at a time)
44 lstn.listen(5)
45
46 # initialize total, v
47 v = ''
48 # set up a lock to guard v
49 vlock = thread.allocate_lock()
50
51 # nclnt will be the number of clients still connected
52 nclnt = 2
53 # set up a lock to guard nclnt
54 nclntlock = thread.allocate_lock()
55
56 # accept calls from the clients
57 for i in range(nclnt):
58     # wait for call, then get a new socket to use for this client,
59     # and get the client's address/port tuple (though not used)
60     (clnt,ap) = lstn.accept()
61     # start thread for this client, with serveclient() as the thread's
62     # function, with parameters clnt; note that parameter set must be
63     # a tuple; in this case, the tuple is of length 1, so a comma is
64     # needed
65     thread.start_new_thread(serveclient,(clnt,))
66
67 # shut down the server socket, since it's not needed anymore
68 lstn.close()
69
70 # wait for both threads to finish
71 while nclnt > 0: pass
72
73 print 'the final value of v is', v

```

Make absolutely sure to run the programs before proceeding further.⁷ Here is how to do this:

I'll refer to the machine on which you run the server as **a.b.c**, and the two client machines as **u.v.w** and **x.y.z**.⁸ First, on the server machine, type

```
python srvr.py 2000
```

and then on each of the client machines type

```
python clnt.py a.b.c 2000
```

(You may need to try another port than 2000, anything above 1023.)

⁷You can get them from the **.tex** source file for this tutorial, located wherever you picked up the **.pdf** version.

⁸You could in fact run all of them on the same machine, with address name **localhost** or something like that, but it would be better on separate machines.

Input letters into both clients, in a rather random pattern, typing some on one client, then on the other, then on the first, etc. Then finally hit Enter without typing a letter to both clients to end the session.

The client code is straight network operations, no threading. But the server is threaded, setting up one thread for each of the two clients.

The reason for threading the server is that the inputs from the clients will come in at unpredictable times. At any given time, the server doesn't know which client will send input next, and thus doesn't know on which client to call `recv()`. One way to solve this problem is by having threads, which run "simultaneously" and thus give the server the ability to read from whichever client has sent data.⁹

So, let's see the technical details. We start with the "main" program.¹⁰

```
vlock = thread.allocate_lock()
```

Here we set up a **lock variable** which guards `v`. We will explain later why this is needed. Note that in order to use this function and others we needed to import the **thread** module.

```
ncInt = 2
ncIntlock = thread.allocate_lock()
```

We will need a mechanism to insure that the "main" program, which also counts as a thread, will be passive until both application threads have finished. The variable `ncInt` will serve this purpose. It will be a count of how many clients are still connected. The "main" program will monitor this, and wrap things up later when the count reaches 0.

```
thread.start_new_thread(serveclient, (cInt,))
```

Having accepted a client connection, the server sets up a thread. This is done via `thread.start_new_thread()`. The first argument is the name of the application function which the thread will run, in this case `serveclient()`. The second argument is a tuple consisting of the set of arguments for that application function. As noted in the comment, this set is expressed as a tuple, and since in this case our tuple has only one component, we use a comma to signal the Python interpreter that this is a tuple.

So, here we are telling Python's threads system to call our function `serveclient()` with the argument `cInt`; the thread becomes "active" immediately. This does not mean that it starts executing right away, though. All that happens is that the threads manager adds this new thread to its list of threads, and marks its current state as runnable, as opposed to being in a state of waiting for some event.

By the way, this gives us a chance to show how clean and elegant Python's threads interface is compared to what one would need in C/C++. For example, in **pthread**, the function analogous to `thread.start_new_thread()` has the signature

```
pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes,
               void *(*thread_function)(void *), void *arguments);
```

⁹Another solution is to use nonblocking I/O. See this example in that context in <http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf>

¹⁰Just as you should write the main program first, you should read it first too, for the same reasons.

What a mess! For instance, look at the types in that third argument: A pointer to a function whose argument is pointer to `void` and whose value is a pointer to `void` (all of which would have to be `cast` when called). It's such a pleasure to work in Python, where we don't have to be bothered by low-level things like that.

```
while nclnt > 0: pass
```

The statement says that as long as at least one client is still active, do nothing. Sounds simple, and it is, but you should consider what is really happening here.

Remember, the three threads—the two client threads, and the “main” one—will take turns executing, with each turn lasting a brief period of time. Each time “main” gets a turn, it will loop repeatedly on this line. This serves our purpose of not allowing “main” to go past this line until both clients are gone.

But on the other hand all that empty looping in “main” is wasted. What we would really like is a way to prevent the “main” function from getting a turn at all until the two clients are gone. There are ways to do this which you will see later, but we have chosen to remain simple for now.

Now consider the function `serveclient()`. Any thread executing this function will deal with only one particular client, the one corresponding to the connection `c` (an argument to the function). So this `while` loop does nothing but read from that particular client. If the client has not sent anything, the thread will block on the line

```
k = c.recv(1)
```

This thread will then be marked as being in Sleep state by the thread manager, thus allowing the other client thread a chance to run. If neither client thread can run, then the “main” thread keeps getting turns. When a user at one of the clients finally types a number, the corresponding thread unblocks, and resumes execution.

Next comes the most important code for the purpose of this tutorial:

```
vlock.acquire()  
v += k  
vlock.release()
```

Here we are worried about a **race condition**. Suppose for example `v` is currently `'abx'`, and Client 0 sends `k` equal to `'g'`. The concern is that this thread's turn might end in the middle of that addition to `v`, say right after the Python interpreter had formed `'abxg'` but before that value was written back to `v`. This could be a big problem. The next thread might get to the same statement, take `v`, still equal to `'abx'`, and append, say, `'w'`, making `v` equal to `'abxw'`. Then when the first thread gets its next turn, it would finish its interrupted action, and set `v` to `'abxg'`—which would mean that the `'w'` from the other thread would be lost.

All of this hinges on whether the operation

```
v += k
```

is interruptible. Could a thread's turn end somewhere in the midst of the execution of this statement? If not, we say that the operation is **atomic**. If the operation were atomic, we would not need the lock/unlock operations surrounding the above statement.

One could do research on the Python virtual machine to find out whether the above statement is atomic. I did so, and it appears to me that it is *not* atomic.¹¹ Moreover, it's safer not to take a chance, especially since Python compilers could vary or the virtual machine could change; after all, we would like our Python source code to work even if the machine changes.

So, we need the lock/unlock operations:

```
vlock.acquire()  
v += k  
vlock.release()
```

The lock, **vlock** here, can only be held by one thread at a time. When a thread executes this statement, the Python interpreter will check to see whether the lock is locked or unlocked right now. In the latter case, the interpreter will lock the lock and the thread will continue, and will execute the statement which updates **v**. It will then release the lock, i.e. the lock will go back to unlocked state.

If on the other hand, when a thread executes **acquire()** on this lock when it is locked, i.e. held by some other thread, its turn will end and the interpreter will mark this thread as being in Sleep state, waiting for the lock to be unlocked. When whichever thread currently holds the lock unlocks it, the interpreter will change the blocked thread from Sleep state to Run state.

Note again that if our threads were non-preemptive, we would not need these locks.

Note also the crucial role being played by the global nature of **v**. Global variables are used to communicate between threads. In fact, recall that this is one of the reasons that threads are so popular—easy access to global variables. Thus the dogma so often taught in beginning programming courses that global variables must be avoided is wrong; on the contrary, there are many situations in which globals are necessary and natural.¹²

The same race-condition issues apply to the code

```
nclntlock.acquire()  
nclnt -= 1  
nclntlock.release()
```

4.2 The threading Module

This is the same network client/server application considered in Section 4.1. The client program stays the same, since it didn't involve threads in the first place. Here is the new server code:

```
1 # simple illustration of threading module  
2
```

¹¹Here's how to look at the Python virtual machine code: Running Python in interactive mode, I first imported the module **dis** ("disassembler"). I then imported the server module, which I had in a file **svr.py**, by typing

```
import(svr)
```

(I first needed to add the usual **if __name__ == '__main__':** code, so that the program wouldn't execute upon being imported.)

You can get a list of Python virtual machine instructions in *Python: the Complete Reference*, by Martin C. Brown, pub. by Osborne, 2001. But if you have background in assembly language, you can probably guess what the code is doing anyway.

It appears that the Python statement in question compiles to four Python virtual machine instructions, not one.

¹²I think that dogma is presented in a far too extreme manner anyway. See <http://heather.cs.ucdavis.edu/~matloff/globals.html>.

```

3 # multiple clients connect to server; each client repeatedly sends a
4 # value k, which the server adds to a global string v and echos back
5 # to the client; k = '' means the client is dropping out; when all
6 # clients are gone, server prints final value of v
7
8 # this is the server
9
10 import socket # networking module
11 import sys
12 import threading
13
14 # class for thread to serve a particular client
15 class srvr(threading.Thread):
16     # v and vlock are now class variables
17     v = ''
18     vlock = threading.Lock()
19     id = 0 # next available thread number
20     def __init__(self,clntsock):
21         # invoke constructor of parent class
22         threading.Thread.__init__(self)
23         # add instance variables
24         self.myid = srvr.id
25         srvr.id += 1
26         self.myclntsock = clntsock
27     # this function is what the thread actually runs; the required name
28     # is run(); threading.Thread.start() calls threading.Thread.run(),
29     # which is always overridden, as we are doing here
30     def run(self):
31         while 1:
32             # receive letter from client, if it is still connected
33             k = self.myclntsock.recv(1)
34             if k == '': break
35             # update v in an atomic manner
36             srvr.vlock.acquire()
37             srvr.v += k
38             srvr.vlock.release()
39             # send new v back to client
40             self.myclntsock.send(srvr.v)
41             self.myclntsock.close()
42
43 # set up Internet TCP socket
44 lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
45 port = int(sys.argv[1]) # server port number
46 # bind lstn socket to this port
47 lstn.bind(('', port))
48 # start listening for contacts from clients (at most 2 at a time)
49 lstn.listen(5)
50
51 nclnt = int(sys.argv[2]) # number of clients
52
53 mythreads = [] # list of all the threads
54 # accept calls from the clients
55 for i in range(nclnt):
56     # wait for call, then get a new socket to use for this client,
57     # and get the client's address/port tuple (though not used)
58     (clnt,ap) = lstn.accept()
59     # make a new instance of the class srvr
60     s = srvr(clnt)
61     # keep a list all threads

```

```

62     mythreads.append(s)
63     # threading.Thread.start calls threading.Thread.run(), which we
64     #     overrode in our definition of the class srvr
65     s.start()
66
67 # shut down the server socket, since it's not needed anymore
68 lstn.close()
69
70 # wait for all threads to finish
71 for s in mythreads:
72     s.join()
73
74 print 'the final value of v is', srvr.v

```

Again, let's look at the main data structure first:

```
class srvr(threading.Thread):
```

The **threading** module contains a class **Thread**, which represent one thread. A typical application will subclass this class, for two reasons. First, we will probably have some application-specific variables or methods to be used. Second, the class **Thread** has a member method **run()** which is almost always overridden, as you will see below.

Consistent with OOP philosophy, we might as well put the old globals in as class variables:

```
v = ''
vlock = threading.Lock()
```

Note that class variable code is executed immediately upon execution of the program, as opposed to when the first object of this class is created. So, the lock is created right away.

```
id = 0
```

This is to set up ID numbers for each of the threads. We don't use them here, but they might be useful in debugging.

```
def __init__(self,clntsock):
    ...
    self.myclntsock = clntsock

# ``main`` program
...
    (clnt,ap) = lstn.accept()
    s = srvr(clnt)

```

The “main” program, in creating an object of this class for the client, will pass as an argument the socket for that client. We then store it as a member variable for the object.

```
def run(self):
    ...

```

As noted earlier, the **Thread** class contains a member method **run()**. This is a dummy, to be overridden with the application-specific function to be run by the thread. It is invoked by the method **Thread.start()**, in the “main” program. As you can see above, it is pretty much the same as the previous code in Section 4.1 which used the **thread** module, adapted to the class environment.

One thing that is quite different in this program is the way used to end it:

```
for s in mythreads:
    s.join()
```

The **join()** method in the class **Thread** blocks until the given thread exits. The overall effect of this loop, then, is that the “main” program will wait at that point until all the threads are done. This is a much cleaner approach than what we used earlier, and it is also more efficient, since the “main” will not be given any turns in which it wastes time looping around doing nothing, as in the program in Section 4.1:

```
while nclnt > 0: pass
```

Here we maintained our own list of threads. However, we could also get one via the call **threading.enumerate()**. If placed after the **for** loop in our server code above, for instance as

```
print threading.enumerate()
```

we would get output like

```
[<_MainThread(MainThread, started)>, <srvr(Thread-1, started)>,
<srvr(Thread-2, started)>]
```

5 Condition Variables

5.1 General Ideas

We saw in the last section that **threading.Thread.join()** avoids the need for wasteful looping in **main()**, while the latter is waiting for the other threads to finish. In fact, it is very common in threaded programs to have situations in which one thread needs to wait for something to occur in another thread. Again, in such situations we would not want the waiting thread to engage in wasteful looping.

The solution to this problem is **condition variables**. Most threads systems allow these, with Python’s **threading** package being no exception. As is typical with Python in so many things, Python makes it easier to use condition variables.

At the first level, there is the class **threading.Condition**, which corresponds well to the condition variables available in most threads systems. However, at this level condition variables are rather cumbersome to use, as not only do we need to set up condition variables but we also need to set up locks to guard them.

So, Python offers a higher-level class, **threading.Event**, which is just a wrapper for **threading.Condition**, but does all the lock operations behind the scenes, alleviating the programmer from this work.

5.2 Event Example

Following is an example of the use of **threading.Event**. It searches a given network host for servers at various ports on that host. As noted in the comment, the threaded operation used here would make more sense if many hosts were to be scanned, rather than just one, as each **connect()** operation does take some time. But even on the same machine, if a server is active but busy enough that we never get to connect to it, it may take a long for the attempt to timeout. It is common to set up Web operations to be threaded for that reason. We could also have each thread check a block of ports on a host, not just one, for better efficiency.

```
1 # portscanner.py: checks for active ports on a given machine; would be
2 # more realistic if checked several hosts at once
3
4 # usage: python portscanner.py host maxthreads
5
6 import sys, threading, socket
7
8 class scanner(threading.Thread):
9     tlist = [] # list of all current scanner threads
10    maxthreads = int(sys.argv[2]) # max number of threads we're allowing
11    evnt = threading.Event() # event to signal OK to create more threads
12    lck = threading.Lock() # lock to guard tlist
13    def __init__(self,tn,host):
14        threading.Thread.__init__(self)
15        self.threadnum = tn # thread ID/port number
16        self.host = host # checking ports on this host
17    def run(self):
18        s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
19        try:
20            s.connect((self.host, self.threadnum))
21            print "%d: successfully connected" % self.threadnum
22            s.close()
23        except:
24            print "%d: connection failed" % self.threadnum
25            # thread is about to exit; remove from list, and signal OK if we
26            # had been up against the limit
27            scanner.lck.acquire()
28            scanner.tlist.remove(self)
29            print "%d: now active --" % self.threadnum, scanner.tlist
30            if len(scanner.tlist) == scanner.maxthreads-1:
31                scanner.evnt.set()
32                scanner.evnt.clear()
33            scanner.lck.release()
34    def newthread(pn,hst):
35        scanner.lck.acquire()
36        sc = scanner(pn,hst)
37        scanner.tlist.append(sc)
38        scanner.lck.release()
39        sc.start()
40        print "%d: starting check" % pn
41        print "%d: now active --" % pn, scanner.tlist
42    newthread = staticmethod(newthread)
43
44 def main():
45     host = sys.argv[1]
46     for i in range(1,100):
47         scanner.lck.acquire()
48         print "%d: attempting check" % i
```

```

49     # check to see if we're at the limit before starting a new thread
50     if len(scanner.tlist) >= scanner.maxthreads:
51         # too bad, need to wait until not at thread limit
52         print "%d: need to wait" % i
53         scanner.lck.release()
54         scanner.evnt.wait()
55     else:
56         scanner.lck.release()
57         scanner.newthread(i,host)
58     for sc in scanner.tlist:
59         sc.join()
60
61 if __name__ == '__main__':
62     main()

```

As you can see, when `main()` discovers that we are at our self-imposed limit of number of active threads, we back off by calling `threading.Event.wait()`. At that point `main()`—which, recall, is also a thread—blocks. It will not be given any more timeslices for the time being. When some active thread exits, we have it call `threading.Event.set()` and `threading.Event.clear()`. The threads manager reacts to the former by moving all threads which had been waiting for this event—in our case here, only `main()`—from Sleep state to Run state; `main()` will eventually get another timeslice.

The call to `threading.Event.clear()` is crucial. Without it, any subsequent call to `threading.Event.wait()` would immediately return, even though the condition has not been met yet.

Note carefully the use of locks. The `main()` thread adds items to `tlist`, while the other threads delete items (delete themselves, actually) from it. These operations must be guarded by locks.

I've put in a lot of extra `print` statements so that you can get an idea as to how the threads' execution is interleaved. Try running the program.¹³ But remember, the program may appear to hang for a long time if a server is active but so busy that the attempt to connect times out.

5.3 Other threading Classes

The function `Event.set()` “wakes” all threads that are waiting for the given event. That didn't matter in our example above, since only one thread (`main()`) would ever be waiting at a time in that example. But in more general applications, we sometimes want to wake only one thread instead of all of them. For this, we can revert to working at the level of `threading.Condition` instead of `threading.Event`. There we have a choice between using `notify()` or `notifyAll()`.

The latter is actually what is called internally by `Event.set()`. But `notify()` instructs the threads manager to waken just one of the waiting threads (we don't know which one).

The class `threading.Semaphore` offers semaphore operations. Other classes of advanced interest are `threading.RLock` and `threading.Timer`.

¹³Disclaimer: Not guaranteed to be bug-free.

6 The Effect of Timesharing

Our earlier examples were **I/O-bound**, meaning that most of its time is spent on input/output. This is a very common type of application of threads.

As mentioned before, another common use for threads is to parallelize **compute-bound** programs, i.e. programs that do a lot of computation. This is useful if one has a **multiprocessor** machine. Unfortunately, as also mentioned, this parallelization is not possible in Python at the moment. However, the compute-bound example here will serve to illustrate the effects of timesharing.

Following is a Python program that finds prime numbers using threads. Note carefully that it is not claimed to be efficient at all; it is merely an illustration of the concepts.

```
1  #!/usr/bin/env python
2
3  import sys
4  import math
5  import thread
6
7  def dowork(tn): # thread number tn
8      global n,prime,nexti,nextilock,nstarted,nstartedlock,donelock
9      donelock[tn].acquire()
10     nstartedlock.acquire()
11     nstarted += 1
12     nstartedlock.release()
13     lim = math.sqrt(n)
14     nk = 0
15     while 1:
16         nextilock.acquire()
17         k = nexti
18         nexti += 1
19         nextilock.release()
20         if k > lim: break
21         nk += 1
22         if prime[k]:
23             r = n / k
24             for i in range(2,r+1):
25                 prime[i*k] = 0
26     print 'thread', tn, 'exiting; processed', nk, 'values of k'
27     donelock[tn].release()
28
29 def main():
30     global n,prime,nexti,nextilock,nstarted,nstartedlock,donelock
31     n = int(sys.argv[1])
32     prime = (n+1) * [1]
33     nthreads = int(sys.argv[2])
34     nstarted = 0
35     nexti = 2
36     nextilock = thread.allocate_lock()
37     nstartedlock = thread.allocate_lock()
38     donelock = []
39     for i in range(nthreads):
40         d = thread.allocate_lock()
41         donelock.append(d)
42         thread.start_new_thread(dowork,(i,))
43     while nstarted < 2: pass
44     for i in range(nthreads):
```

```

45     donelock[i].acquire()
46     print 'there are', reduce(lambda x,y: x+y, prime) - 2, 'primes'
47
48 if __name__ == '__main__':
49     main()

```

6.1 Code Analysis

So, let's see how the code works.

The algorithm is the famous Sieve of Erathosthenes: We list all the numbers from 2 to **n**, then cross out all multiples of 2 (except 2), then cross out all multiples of 3 (except 3), and so on. The numbers which get crossed out are composite, so the ones which remain at the end are prime.

Line 32: We set up an array **prime**, which is what we will be “crossing out.” The value 1 means “not crossed out,” so we start everything at 1. (Note how Python makes this easy to do, using list “multiplication.”)

Line 33: Here we get the number of desired threads from the command line.

Line 34: The variable **nstarted** will show how many threads have already started. This will be used later, in Lines 43-45, in determining when the **main()** thread exits. Since the various threads will be writing this variable, we need to protect it with a lock, on Line 37.

Lines 35-36: The variable **nexti** will say which value we should do “crossing out” by next. If this is, say, 17, then it means our next task is to cross out all multiples of 17 (except 17). Again we need to protect it with a lock.

Lines 39-42: We create the threads here. The function executed by the threads is named **dowork()**. We also create locks in an array **donelock**, which again will be used later on as a mechanism for determining when **main()** exits (Line 44-45).

Lines 43-45: There is a lot to discuss here. To start, first look back at Line 50 of **srvr.py**, our earlier example. We didn't want the main thread to exit until the two child threads were done.¹⁴ So, Line 50 is a **busy wait**, repeatedly doing nothing (**pass**). That's a waste of time—each time the main thread gets a turn to run, it repeatedly executes **pass** until its turn is over.

We'd like to avoid such waste in our primes program, which we do in Lines 43-45. To understand what those lines do, look at Lines 10-12. Each child thread increments a count, **nstarted**; meanwhile, on Line 43 the main thread is wasting time executing **pass**.¹⁵ But as soon as the last thread increments the count, the main thread leaves its busy wait and goes to Line 44.¹⁶

Back in each child thread, the thread acquires its **donelock** lock on Line 9, and doesn't release it until Line 27, when the thread is done. Meanwhile, the main thread is waiting for those locks, in Lines 44-45. *This is very different from the wait it did on Line 43.* In the latter case, the main thread just spun around, wasting time by repeatedly executing **pass**. By contrast, in Lines 44-45, the main thread isn't wasting time—because it's not executing at all.

¹⁴The effect of the main thread ending earlier would depend on the underlying OS. On some platforms, exit of the parent may terminate the child threads, but on other platforms the children continue on their own.

¹⁵In reading the word *meanwhile* here, remember that the threads are taking turns executing, 10 Python virtual machine instructions per turn. Thus the word *meanwhile* only refers to concurrency among the threads, not simultaneity.

¹⁶Again, the phrase *as soon as* should not be taken literally. What it really means is that after the count reaches **nthreads**, the next time the main thread gets a turn, it goes to Line 44.

To see this, consider the case of $i = 0$. The call to **acquire** in Line 45 will block. From this point on, the thread manager within the Python interpreter will not give the main thread any turns, until finally child thread 0 executes Line 27. At that point, the thread manager will notice that the lock which had just been released was being awaited by the main thread, so the manager will “waken” the main thread, i.e. resume giving it turns. Of course, then i will become 1, and the main thread will “sleep” again.

Note carefully the roles of Lines 9-12 and 43. Without them, the main thread might be able to execute Line 45 with $i = 0$ before child thread 0 executes Line 12. If the same thing happened with $i = 1$, then the main thread would exit prematurely.

So, we’ve avoided premature exit while at the same time allowing only minimal time wasting by the main thread.

Line 13: We need not check any “crosser-outers” that are larger than \sqrt{n} .

Lines 15-25: We keep trying “crosser-outers” until we reach that limit (Line 20). Note the need to use the lock in Lines 16-19. In Line 22, we check the potential “crosser-outer” for primeness; if we have previously crossed it out, we would just be doing duplicate work if we used this k as a “crosser-outer.”

6.2 Execution Analysis

Note that I put code in Lines 21 and 26 to measure how much work each thread is doing. Here k is the “crosser-outer,” i.e. the number whose multiples we are crossing out. Line 21 tallies how many values of k this thread is handling. Let’s run the program and see what happens.

```
% python primes.py 100 2
thread 0 exiting; processed 9 values of k
thread 1 exiting; processed 0 values of k
there are 25 primes
% python primes.py 10000 2
thread 0 exiting; processed 99 values of k
thread 1 exiting; processed 0 values of k
there are 1229 primes
% python primes.py 10000 2
thread 0 exiting; processed 99 values of k
thread 1 exiting; processed 0 values of k
there are 1229 primes
% python primes.py 100000 2
thread 1 exiting; processed 309 values of k
thread 0 exiting; processed 6 values of k
there are 9592 primes
% python primes.py 100000 2
thread 1 exiting; processed 309 values of k
thread 0 exiting; processed 6 values of k
there are 9592 primes
% python primes.py 100000 2
thread 1 exiting; processed 311 values of k
thread 0 exiting; processed 4 values of k
there are 9592 primes
% python primes.py 1000000 2
thread 1 exiting; processed 180 values of k
thread 0 exiting; processed 819 values of k
there are 78498 primes
% python primes.py 1000000 2
```

```

thread 1 exiting; processed 922 values of k
thread 0 exiting; processed 77 values of k
there are 78498 primes
% python primes.py 1000000 2
thread 0 exiting; processed 690 values of k
thread 1 exiting; processed 309 values of k
there are 78498 primes

```

This is really important stuff. For the smaller values of **n** like 100, there was so little work to do that thread 0 did the whole job before thread 1 even got started. Thread 1 got more chance to run as the size of the job got longer. The imbalance of work done, if this occurred on a multiprocessor system, is known as the **load balancing** problem.

Note also that even for the larger jobs there was considerable variation from run to run. How is this possible, given that the size of a turn is fixed at a certain Python byte code instructions? The answer is that although the turn size is constant, the delay before a thread is created is random, due to the fact that the Python threads system makes use of an underlying threads system (in this case **pthreads** on Linux). In many of the runs above, for instance, thread 0 was started first and thus did the lion's share of the work, but in some cases thread 1 was started first.

7 The Queue Module

Threaded applications often have some sort of work queue data structure. When a thread becomes free, it will pick up work to do from the queue. When a thread creates a task, it will add that task to the queue.

Clearly one needs to guard the queue with locks. But Python provides the **Queue** module to take care of all the lock creation, locking and unlocking, and so on, so that we don't have to bother with it.

Here is an example of its use:

```

1 # pqsort.py: threaded quicksort
2 # sorts an array with a fixed pool of worker threads
3
4 # disclaimer: does NOT produce a speedup, even on multiprocessor
5 # machines, as Python threads cannot run simultaneously
6
7 # adapted by Francis Hsu from Prof. Norm Matloff's
8 # Shared-Memory Quicksort in Introduction to Parallel Programming
9 # http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProc.pdf
10
11 import threading, Queue, random
12
13 class pqsort:
14     ''' threaded parallel quicksort '''
15     nsingletons = 0          # used to track termination
16     nsingletonslock = None
17
18     def __init__(self, a, numthreads = 5):
19         ''' quicksorts array a in parallel with numthreads threads '''
20         jobs = Queue.Queue() # job queue
21         pqsort.pqsorter.numthreads = 0 # thread creation count
22         self.threads = []    # threads
23         pqsort.nsingletons = 0 # count of positions that are sorted

```

```

24                                     # done sorting when == to len(a)
25 pgsort.nsingletonslock = threading.Lock()
26
27 jobs.put((0,len(a)))
28
29 for i in range(numthreads):         # spawn threads
30     t = pgsort.pgsorter(a, jobs)
31     self.threads.append(t)
32     t.start()
33
34 for t in self.threads:               # wait for threads to finish
35     t.join()
36
37 def report(self):
38     for t in self.threads:
39         t.report()
40
41 class pgsorter(threading.Thread):
42     ''' worker thread for parallel quicksort '''
43     numthreads = 0                   # thread creation count
44
45     def __init__(self, a, jobs):
46         self.a = a                  # array being handled by this thread
47         self.jobs = jobs            # Queue of sorting jobs to do
48         pgsort.pgsorter.numthreads += 1 # update count of created threads
49         self.threadid = self.numthreads # unique id of this thread
50         self.loop = 0               # work done by thread
51
52         threading.Thread.__init__(self)
53
54     def run(self):
55         ''' thread loops taking jobs from queue until none are left '''
56         while pgsort.nsingletons < len(self.a):
57             try:
58                 job = self.jobs.get(True,1) # get job
59                                         # Queue handles the locks for us
60             except:
61                 continue
62
63             if job[0] >= job[1]:         # partitioning an array of 1
64                 pgsort.nsingletonslock.acquire()
65                 pgsort.nsingletons+=1
66                 pgsort.nsingletonslock.release()
67                 continue
68
69             self.loop +=1
70             m = self.separate(job)      # partition
71
72             self.jobs.put((job[0], m))  # create new jobs to handle the
73             self.jobs.put((m+1, job[1])) # new left and right partitions
74
75     def separate(self, (low, high)):
76         ''' quicksort partitioning with first element as pivot '''
77         pivot = self.a[low]
78         last = low
79         for i in range(low+1,high):
80             if self.a[i] < pivot:
81                 last += 1
82                 self.a[last], self.a[i] = self.a[i], self.a[last]

```

```

83         self.a[low], self.a[last] = self.a[last], self.a[low]
84         return last
85
86     def report(self):
87         print "thread", self.threadid, "visited array", self.loop , "times"
88
89 def main():
90     ''' pqsort timesharing analysis '''
91     for size in range(10):
92         a = range(100*(size+1))
93         shufflesort(a)
94
95 def shufflesort(a):
96     #shuffle array
97     for i in range(len(a)):
98         r = random.randint(i, len(a)-1)
99         (a[i], a[r]) = (a[r], a[i])
100
101     #sort array
102     s = pqsort(a)
103     print "For sorting an array of size", len(a)
104     s.report()
105
106 if __name__ == '__main__':
107     main()

```

By the way, let's see how the load balancing went:

```

% python pqsort.py
For sorting an array of size 100
thread 1 visited array 88 times
thread 2 visited array 12 times
thread 3 visited array 0 times
thread 4 visited array 0 times
thread 5 visited array 0 times
For sorting an array of size 200
thread 1 visited array 189 times
thread 2 visited array 0 times
thread 3 visited array 11 times
thread 4 visited array 0 times
thread 5 visited array 0 times
For sorting an array of size 300
thread 1 visited array 226 times
thread 2 visited array 74 times
thread 3 visited array 0 times
thread 4 visited array 0 times
thread 5 visited array 0 times
For sorting an array of size 400
thread 1 visited array 167 times
thread 2 visited array 112 times
thread 3 visited array 41 times
thread 4 visited array 58 times
thread 5 visited array 22 times
For sorting an array of size 500
thread 1 visited array 249 times
thread 2 visited array 125 times
thread 3 visited array 100 times

```

```
thread 4 visited array 17 times
thread 5 visited array 9 times
For sorting an array of size 600
thread 1 visited array 87 times
thread 2 visited array 185 times
thread 3 visited array 120 times
thread 4 visited array 105 times
thread 5 visited array 103 times
For sorting an array of size 700
thread 1 visited array 295 times
thread 2 visited array 278 times
thread 3 visited array 54 times
thread 4 visited array 32 times
thread 5 visited array 41 times
For sorting an array of size 800
thread 1 visited array 291 times
thread 2 visited array 217 times
thread 3 visited array 52 times
thread 4 visited array 204 times
thread 5 visited array 36 times
For sorting an array of size 900
thread 1 visited array 377 times
thread 2 visited array 225 times
thread 3 visited array 113 times
thread 4 visited array 128 times
thread 5 visited array 57 times
For sorting an array of size 1000
thread 1 visited array 299 times
thread 2 visited array 233 times
thread 3 visited array 65 times
thread 4 visited array 249 times
thread 5 visited array 154 times
```

8 Debugging Threads with PDB

Debugging is always tough with parallel programs, including threads programs. It's especially difficult with pre-emptive threads; those accustomed to debugging non-threads programs find it rather jarring to see sudden changes of context while single-stepping through code. Tracking down the cause of deadlocks can be very hard. (Often just getting a threads program to end properly is a challenge.)

Another problem which sometimes occurs is that if you issue a “next” command in your debugging tool, you may end up inside the internal threads code. In such cases, use a “continue” command or something like that to extricate yourself.

8.1 Using PDB

Unfortunately, threads debugging is even more difficult in Python, at least with the basic PDB debugger. One cannot, for instance, simply do something like this:

```
pdb.py buggyprog.py
```

because the child threads will not inherit the PDB process from the main thread. You can still run PDB in the latter, but will not be able to set breakpoints in threads.

What you can do, though, is invoke PDB from *within* the function which is run by the thread, by calling `pdb.set_trace()` at one or more points within the code:

```
import pdb
pdb.set_trace()
```

In essence, those become breakpoints.

For example, in our program above, we could add a PDB call at the beginning of the loop in `serveclient()`:

```
while 1:
    import pdb
    pdb.set_trace()
    # receive letter from client, if it is still connected
    k = c.recv(1)
    if k == '': break
```

You then run the program directly through the Python interpreter as usual, NOT through PDB, but then the program suddenly moves into debugging mode on its own. At that point, one can then step through the code using the `n` or `s` commands, query the values of variables, etc.

PDB's `c` (“continue”) command still works. Can one still use the `b` command to set additional breakpoints? Yes, but it might be only on a one-time basis, depending on the context. A breakpoint might work only once, due to a scope problem. Leave the scope where we invoked PDB causes removal of the trace object.

Of course, you can get fancier, e.g. setting up “conditional breakpoints,” something like:

```
debugflag = int(sys.argv[1])
...
if debugflag == 1:
    import pdb
    pdb.set_trace()
```

Then, the debugger would run only if you asked for it on the command line. Or, you could have multiple `debugflag` variables, for activating/deactivating breakpoints at various places in the code.

Moreover, once you get the `(Pdb)` prompt, you could set/reset those flags, thus also activating/deactivating breakpoints.

Note that local variables which were set before invoking PDB, including parameters, are not accessible to PDB.

Make sure to insert code to maintain an ID number for each thread. This really helps when debugging.

8.2 RPDB

A threads-capable variant of PDB is RPDB. RPDB has the additional advantage that one can submit debugging commands in a separate screen when debugging a terminal-based application.

I've written a mini-tutorial on RPDB, accessible at <http://heather.cs.ucdavis.edu/~matloff/rpdb.html>.

9 Non-Pre-emptive Threads in Python

Pre-emptive threading is a pain.

It is possible to use Python generators to implement non-pre-emptive threads systems in Python. One example of this is the SimPy discrete-event system, <http://simpy.sourceforge.net/>.