Tutorial on Threads Programming with Python

Norman Matloff and Francis Hsu*
University of California, Davis

May 5, 2009

Contents

*Francis, a graduate student, wrote most of Section 6.

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 settings in which using threads has been founded convenient and/or efficient:

- *Programs with asynchronous events:*

  Here the program must be ready for various events, but does not know the order in which they might occur. For example, in Sections 3.1 and 3.2, we have a network server connected to several clients. The server does not know from which client the next message will arrive. So, we have the server create a separate thread for each client, with each thread handling only its client.

- *Programs whose peformance can be improved through latency hiding:*

  Here the program is doing multiple I/O operations, each having long **latency**, i.e. delay in response. We'd like to perform useful work while waiting for the response, so we have different threads for each I/O action. This way, although the latency is still there, it is "hidden" by doing other useful work in parallel.

  For example, in Section 4.2, each thread performs a separate network operation.

- *Computation-intensive programs:*

  If our program is a long-running mathematical computation, it can really benefit from having several processors, e.g. two processors in the case of dual-core machines. By having our program set up a different thread for each processor, we have the potential for substantial speedup, due to the parallelization of the computation. An example is in Section 5.

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. This is changing, as for example dual-core CPU chips have become common in home PCs. But even then, the principle is the same, as there typically will be more processes than CPUs, so that some of the simultaneity is illusory.

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, of fixed size, say for concreteness 30 milliseconds. After a process has run for 30 milliseconds, a hardware timer emits an interrupted which causes the OS to run. We say that the process has been **pre-empted**. 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. Since user programs make calls to the OS to do I/O, that causes a jump to the OS, prematurely ending this process' turn. The OS marks the process as being in Sleep state, meaning not currently eligible for turns, since it is waiting for the I/O to occur. 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.

Note carefully that a process may also be in Sleep state if it is waiting for some non-I/O event to occur. There are calls one can make from a process in which we say, "Don't run this process again until some other process has set a certain variable" or the like.

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 process, and may even be a process, depending on the thread system. In fact, threads are sometimes called "lightweight" processes, because threads occupy much less memory, and take less time to create, than do processes.

Just as processes can be interrupted at any time, the same is generally true for threads. I say "generally" because there are various kinds of qualifying statements and exceptions to this, to be discussed in Section 7, but for Python it's true. The point is that in general one must be very careful in this regard. In particular, proper use of lock variables is crucial, as we will see.

Also, just as one process may create one or more child processes, e.g. using **fork()** in Unix, with threading, our program creates one or more child threads. By the way, the parent is also a thread.

On the other hand, a major difference between ordinary 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 in a threaded environment are shared by all threads, and serve as the main method of communication between the threads.[1]

2.3 Python Threads

Python's thread system builds on the underlying OS threads. Thus are thus pre-emptible. Note, though, that Python adds its own threads manager on top of the OS thread system; see Section 7.

[1] It is possible to share globals among Unix processes, but very painful.

3 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, so we will start with it.

3.1 The `thread` Module

The example here involves a client/server pair.[2] As you'll see from reading the comments at the start of the files, the program 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.

Only the server needs to be threaded. It will have one thread for each client.

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

```
1  # simple illustration of thread module
2
3  # two clients connect to server; each client repeatedly sends a letter,
4  # stored in the variable k, which the server appends to a global string
5  # v, and reports v to the client; k = '' means the client is dropping
6  # out; when all clients are 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
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, **srvr.py**:

```
1  # simple illustration of thread module
2
3  # multiple clients connect to server; each client repeatedly sends a
```

[2] It 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.

```
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, c
20 def serveclient(c):
21    global v,nclnt,vlock,nclntlock
22    while 1:
23       # receive letter from c, if it is still connected
24       k = c.recv(1)
25       if k == '': break
26       # concatenate v with k in an atomic manner, i.e. with protection
27       #    by locks
28       vlock.acquire()
29       v += k
30       vlock.release()
31       # send new v back to client
32       c.send(v)
33    c.close()
34    nclntlock.acquire()
35    nclnt -= 1
36    nclntlock.release()
37
38 # set up Internet TCP socket
39 lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40
41 port = int(sys.argv[1])  # server port number
42 # bind lstn socket to this port
43 lstn.bind(('', port))
44 # start listening for contacts from clients (at most 2 at a time)
45 lstn.listen(5)
46
47 # initialize concatenated string, v
48 v = ''
49 # set up a lock to guard v
50 vlock = thread.allocate_lock()
51
52 # nclnt will be the number of clients still connected
53 nclnt = 2
54 # set up a lock to guard nclnt
55 nclntlock = thread.allocate_lock()
56
57 # accept calls from the clients
58 for i in range(nclnt):
59    # wait for call, then get a new socket to use for this client,
60    #    and get the client's address/port tuple (though not used)
61    (clnt,ap) = lstn.accept()
62    # start thread for this client, with serveclient() as the thread's
63    #    function, with parameter clnt; note that parameter set must be
64    #    a tuple; in this case, the tuple is of length 1, so a comma is
65    #    needed
66    thread.start_new_thread(serveclient,(clnt,))
67
68 # shut down the server socket, since it's not needed anymore
69 lstn.close()
70
71 # wait for both threads to finish
```

```
72  while nclnt > 0: pass
73
74  print 'the final value of v is', v
```

Make absolutely sure to run the programs before proceeding further.[3] 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**.[4] 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.)

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 one of the clients to end the session for that client, type a few more characters in the other client, and then end that session too.

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 sent 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.[5].

So, let's see the technical details. We start with the "main" program.[6]

```
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.

```
nclnt = 2
nclntlock = 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 **nclnt** 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,(clnt,))
```

[3] You can get them from the **.tex** source file for this tutorial, located wherever your picked up the **.pdf** version.

[4] You 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.

[5] Another solution is to use nonblocking I/O. See this example in that context in `http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf`

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

Having accepted a a client connection, the server sets up a thread for serving it, 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()**, supplying that function with the argument **clnt**. The thread becomes "active" immediately, but this does not mean that it starts executing right away. All that happens is that the threads manager adds this new thread to its list of threads, and marks its current state as Run, as opposed to being in a Sleep state, 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 **pthreads**, 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);
```

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.

Now consider our statement

```
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. But all that empty looping in "main" is wasted time. 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 letter, the corresponding thread unblocks, i.e. the threads manager changes its state to Run, so that it will soon resume 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. I did this, using the methods described in Section 7.4, and it appears to me that the above statement is *not* atomic.

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 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.[7]

The same race-condition issues apply to the code

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

[7] I think that dogma is presented in a far too extreme manner anyway. See `http://heather.cs.ucdavis.edu/~matloff/globals.html`.

3.2 The `threading` Module

The program below treats the same network client/server application considered in Section 3.1, but with the more sophisticated **threading** module. 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
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 threads, subclassed from threading.Thread class
15 class srvr(threading.Thread):
16    # v and vlock are now class variables
17    v = ''
18    vlock = threading.Lock()
19    id = 0  # I want to give an ID number to each thread, starting at 0
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**, any instance of which represents 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 meant to be 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 or in future enhancement of the code.

```
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()**, called here in the main program. As you can see above, it is pretty much the same as the previous code in Section 3.1 which used the **thread** module, adapted to the class environment.

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

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

The **join()** method in the class **Thread** blocks until the given thread exits. (The threads manager puts the main thread in Sleep state, and when the given thread exits, the manager changes that state to Run.) The overall effect of this loop, then, is that the main program will wait at that point until all the threads are done. They "join" the main program. This is a much cleaner approach than what we used earlier, and it is also more efficient, since the main program will not be given any turns in which it wastes time looping around doing nothing, as in the program in Section 3.1 in the line

```
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)>]
```

4 Condition Variables

4.1 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**. As the name implies, these are variables used by code to wait for a certain condition to occur. Most threads systems include provisions for these, and Python's **threading** package is exception.

The **pthreads** package, for instance, has a type **pthread_cond** for such variables, and has functions such as **pthread_cond_wait()**, which a thread calls to wait for an event to occur, and **pthread_cond_signal()**, which another thread calls to announce that the event now has occurred.

But as is typical with Python in so many things, it is easier for us to use condition variables in Python than in C. 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 extra locks to guard them. This is necessary in any threading system, but it is a nuisance to deal with.

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

4.2 `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. (This is called a **port scanner**.) As noted in a 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.

The use of threads is aimed at checking many ports in parallel, one per thread. The program has a self-imposed limit on the number of threads. If **main()** is ready to start checking another port but we are at the thread limit, the code in **main()** waits for the number of threads to drop below the limit. This is accomplished by a condition wait, implemented through the **threading.Event** class.

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

```
36          scanner.lck.release()
37      def newthread(pn,hst):
38          scanner.lck.acquire()
39          sc = scanner(pn,hst)
40          scanner.tlist.append(sc)
41          scanner.lck.release()
42          sc.start()
43          print "%d:  starting check" % pn
44          print "%d:  now active --" % pn, scanner.tlist
45      newthread = staticmethod(newthread)
46
47  def main():
48      host = sys.argv[1]
49      for i in range(1,100):
50          scanner.lck.acquire()
51          print "%d:  attempting check" % i
52          # check to see if we're at the limit before starting a new thread
53          if len(scanner.tlist) >= scanner.maxthreads:
54              # too bad, need to wait until not at thread limit
55              print "%d:  need to wait" % i
56              scanner.lck.release()
57              scanner.evnt.wait()
58          else:
59              scanner.lck.release()
60          scanner.newthread(i,host)
61      for sc in scanner.tlist:
62          sc.join()
63
64  if __name__ == '__main__':
65       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. The word *clear* here means that **threading.Event.clear()** is clearing the occurence of the event. Without this, 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 atomic, and thus 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.[8] 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.

4.3 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()**.

[8] Disclaimer: Not guaranteed to be bug-free.

The latter is actually what is called internally by **Event.set()**. But **notify()** instructs the threads manager to wake 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**.

5 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 multicore machine. Unfortunately, as to be discussed, this parallelization is not possible in standard Python. The latter is CPython, the most common implementation, based on C. Another population implementation is Jython, implemented in Java, and still another is Microsoft's Iron Python. Jython and Iron Python do not have the parallelization constraint, and we will discuss this in more detail later. In any case, 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. Note too that we are using the simple **thread** module, rather than **threading**.

```
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 < nthreads: 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()
```

5.1 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, recall that in **srvr.py**, our example in Section 3.1, we didn't want the main thread to exit until the child threads were done.[9] So, Line 50 was 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.

Here in our primes program, a premature exit by **main()** result in printing out wrong answers. On the other hand, we don't want **main()** to engage in a wasteful busy wait. We could use **join()** from **threading.Thread** for this purpose, as we have before, but here we take a different tack: We set up a list of locks, one for each thread, in a list **donelock**. Each thread initially acquires its lock (Line 9), and releases it when the thread finishes its work (Lin 27). Meanwhile, **main()** has been waiting to acquire those locks (Line 45). So, when the threads finish, **main()** will move on to Line 46 and print out the program's results.

[9]The 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.

But there is a subtle problem (threaded programming is notorious for subtle problems), in that there is no guarantee that a thread will execute Line 9 before **main()** executes Line 45. That's why we have a busy wait in Line 43, to make sure all the threads acquire their locks before **main()** does. Of course, we're trying to avoid busy waits, but this one is quick.

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."

5.2 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 it occurrs on a multiprocessor system with truly concurrent threads (not ours here), 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 number of 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.

6 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.

The function **put()** in Queue adds an element to the end of the queue, while **get()** will remove the head of the queue, again without the programmer having to worry about race conditions. Note that **put()**, at least in its default form, will block if the queue is currently empty.

Below is an example of its use, an in-place quicksort. Items in the queue are pairs of list indices, with for example (12,29) representing the task of sorting the sublist between indices 12 and 29.

```
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 standard 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       pqsort.nsingletonslock = threading.Lock()
26
27       jobs.put((0,len(a)))
28
29       for i in range(numthreads):    # spawn threads
30          t = pqsort.pqsorter(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 pqsorter(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            pqsort.pqsorter.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 pqsort.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                  pqsort.nsingletonslock.acquire()
65                  pqsort.nsingletons+=1
66                  pqsort.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
```

7 Threads Internals

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 where it had left off, i.e. where its last turn ended.

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. The thread manager will keep track of these states, decide which thread to run when another has lost its turn, etc.

7.1 Kernel-Level Thread Managers

Here each thread really is a process, and for example will show up on Unix systems when one runs the appropriate **ps** process-list command, say **ps axH**. The threads manager is then the OS.

The different threads set up by a given application program take turns running, among all the other processes.

This kind of thread system is is used in the Unix **pthreads** system, as well as in Windows threads.

7.2 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.[10]

A typical example of a user-level thread system is **pth**.

7.3 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.

7.4 The Python Thread Manager

Python "piggybacks" on top of the OS' underlying threads system. A Python thread is a real OS thread. If a Python program has three threads, for instance, there will be three entries in the **ps** output.

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

However, Python's thread manager imposes further structure on top of the OS threads. It keeps track of how long a thread has been executing, in terms of the number of Python **byte code** instructions that have executed.[11] When that reaches a certain number, by default 100, the thread's turn ends. In other words, the turn can be pre-empted either by the hardware timer and the OS, or when the interpreter sees that the thread has executed 100 byte code instructions.

7.4.1 The GIL

In the case of CPython (but not Jython or Iron Python) Most importantly, there is a global interpreter lock, the famous (or infamous) GIL. It is set up to ensure that only one thread runs at a time, in order to facilitate easy garbage collection.

Suppose we have a C program with three threads, which I'll call X, Y and Z. Say currently Y is running. After 30 milliseconds (or whatever the quantum size has been set to by the OS), Y will be interrupted by the timer, and the OS will start some other process. Say the latter, which I'll call Q, is a different, unrelated program. Eventually Q's turn will end too, and let's say that the OS then gives X a turn. From the point of view of our X/Y/Z program, i.e. ignoring Q, control has passed from Y to X. The key point is that the point within Y at which that event occurs is random (with respect to where Y is at the time), based on the time of the hardware interrupt.

By contrast, say my Python program has three threads, U, V and W. Say V is running. The hardware timer will go off at a random time, and again Q might be given a turn, *but* definitely neither U nor W will be given a turn, because the Python interpreter had earlier made a call to the OS which makes U and W wait for the GIL to become unlocked.

Let's look at this a little closer. The key point to note is that the Python interpreter itself is threaded, say using **pthreads**. For instance, in our X/Y/Z example above, when you ran **ps axH**, you would see three Python processes/threads. I just tried that on my program **thsvr.py**, which creates two threads, with a command-line argument of 2000 for that program. Here is the relevant portion of the output of **ps axH**:

```
28145 pts/5    Rl     0:09 python thsvr.py 2000
28145 pts/5    Sl     0:00 python thsvr.py 2000
28145 pts/5    Sl     0:00 python thsvr.py 2000
```

What has happened is the Python interpreter has spawned two child threads, one for each of my threads in **thsvr.py**, in addition to the interpreter's original thread, which runs my **main()**. Let's call those threads UP, VP and WP. Again, these are the threads that the OS sees, while U, V and W are the threads that I see—or think I see, since they are just virtual.

The GIL is a **pthreads** lock. Say V is now running. Again, what that actually means on my real machine is that VP is running. VP keeps track of how long V has been executing, in terms of the number of Python **byte code** instructions that have executed. When that reaches a certain number, by default 100, UP will release the GIL by calling **pthread_mutex_unlock()** or something similar.

The OS then says, "Oh, were any threads waiting for that lock?" It then basically gives a turn to UP or WP (we can't predict which), which then means that from my point of view U or W starts, say U. Then VP and WP are still in Sleep state, and thus so are my V and W.

[11] This is the "machine language" for the Python virtual machine.

So you can see that it is the Python interpreter, not the hardware timer, that is determining how long a thread's turn runs, relative to the other threads in my program. Again, Q might run too, but within this Python program there will be no control passing from V to U or W simply because the timer went off; such a control change will only occur when the Python interpreter wants it to. This will be either after the 100 byte code instructions or when U reaches an I/O operation or other wait-event operation.

So, the bottom line is that while Python uses the underlying OS threads system as its base, it superimposes further structure in terms of transfer of control between threads.

7.4.2 Implications for Randomness and Need for Locks

I mentioned in Section 7.2 that non-pre-emptive threading is nice because one can avoid the code clutter of locking and unlocking (details of lock/unlock below). Since, barring I/O issues, a thread working on the same data would seem to always yield control at exactly the same point (i.e. at 100 byte code instruction boundaries), Python would seem to be deterministic and non-pre-emptive. However, it will not quite be so simple.

First of all, there is the issue of I/O, which adds randomness. There may also be randomness in how the OS chooses the first thread to be run, which could affect computation order and so on.

Finally, there is the question of atomicity in Python operations: 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.

7.4.3 The GIL Controversy, and the New `multiprocessing` Module

CPython's GIL is the subject of much controversy. As you can see, it prevents running true parallel threaded CPython[12] on multiprocessor machines, such as the multicore machines that are now commonplace, thus limiting performance.

That might not seem to be too severe a restriction—after all if you really need the speed, you probably won't use a scripting language in the first place. But a number of people take the point of view that, given that they have decided to use Python no matter what, they would like to get the best speed subject to that restriction. So, there has been much grumbling about the GIL.

However, the creator of Python, Guido Van Rossum, points out that it really is a nonissue. If one wants true parallel processing with Python on a multiprocore machine, one should use ordinary processes, not threads.

Until recently, that solution was somewhat less attractive in that, as noted earlier, it is difficult to share variables at the ordinary process level (whether in Python, C or whatever). That problem has now been solved, though, by the new **multiprocessing** module. It provides an interface very close to that of the **threading** module, so shared variables are not a problem.

Moreover, one can run a program across machines! In other words, the **multiprocessing** module allows to run several threads not only on the different cores of one machine, but on many machines at once, in cooperation in the same manner that threads cooperate on one machine. By the way, this idea is similar to something I did for Perl some years ago (PerlDSM: A Distributed Shared Memory System for Perl. *Proceedings of PDPTA 2002*, 63-68).

[12] Note the qualifier *threaded*.

A Debugging Threaded Programs

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.

A.1 Forcing PDB to Debug Threaded Programs

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 **srvr.py** in Section 3.1, 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.

A.2 RPDB2 and Winpdb

The Winpdb debugger (`www.digitalpeers.com/pythondebugger/`),[13] is very good. Among other things, it can be used to debug threaded code, curses-based code and so on, which many debuggers can't. Winpdb is a GUI front end to the text-based RPDB2, which is in the same package. I have a tutorial on both at `http://heather.cs.ucdavis.edu/~matloff/winpdb.html`.

B 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/`.

[13] No, it's not just for Microsoft Windows machines, in spite of the name.