Introduction to Threads Programming with Python

Norman Matloff
University of California, Davis

May 6, 2003

Contents

1 What Are Threads?

The overall goal of **threads** is to make it convenient to write programs which run multiple tasks, and to do so efficiently.

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

A thread is like a UNIX process, and in fact is sometimes called a "lightweight" process. From a programming point of view, the difference 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.[1] In terms of resource usage, threads occupy much less memory, and take less time to create.

2 Overview of Thread Operations

In a compiled language like C or C++, a threads package is accessed via calls to a library. There are many such libraries available, including popular public-domain (i.e. free) ones such as **pthreads** and **pth**. 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.

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 control.

Thread systems are either **kernel-level** or **user-level**. Let's consider former case first. 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 quantum of time, 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**. This kind of thread system is is used in the UNIX **pthreads** system, as well as in Windows threads.

User-level thread systems, on the other hand, 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.[2]

Kernel-level threads have the advantage that they can be used on multiprocessor systems. On the other hand, in my opinion user-level threads have enormous advantage that they allow one to produce code which is much easier to write, easier to debug, cleaner and clearer. This in turn stems from the non-preemptive

[1] It is possible to share globals among UNIX processes, but very painful. By the way, if you need a quick overview/review of operating systems, see `http://heather.cs.ucdavis.edu/~matloff/50/PLN/OSOverview.pdf`.

[2] 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.

nature of user-level threads; application programs written in this manner typically are not cluttered up with lots of lock/unlock calls, which are needed in the pre-emptive case.

3 Python Threads

Even though Python's threads mechanisms are built on the underlying platform's threads system, this is basically transparent to the application programmer. The interpreter keeps track of how long the current thread has executing, in terms of the number of Python **byte code** instructions have executed.[3] When that reaches a certain number, by default 10, another thread will be given a turn.[4] Such a switch will also occur if a thread reaches an I/O statement. Thus Python threads are pre-emptive.[5]

Internally, Python maintains a Global Interpreter Lock to ensure that only one thread has access to the interpreter at a time.

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

4 First Example

The example 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 numbers to the server; the server totals all the numbers it receives.

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

```
1  # simple illustration client of thread module
2
3  # two clients connect to server; each client repeatedly sends a
4  # value k, which the server adds to a global value v and echoes back
5  # to the client; k = 0 means the client is dropping out; when all
6  # clients gone, server prints the final value of v
7
8  # the values k are sent as single bytes for convenience; it's assumed
9  # that neither k nor v ever exceed 255
10
11 # this is the client; usage is
12
13 #    python clnt.py server_address port_number
14
15 import socket
16 import sys
17
18 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
19
20 host = sys.argv[1]  # server address
```

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

[4] This number is settable, via a call to **sys.setcheckinterval()**.

[5] Starting with Python 2.2, though, one can use **generators** as a non-preemptive alternative to threads.

[6] It is assumed here that the reader is familiar with basic network programming. See my tutorial at `http://heather.cs.ucdavis.edu/~matloff/Python/PyNet.pdf`.

```
port = int(sys.argv[2])  # server port
s.connect((host, port))

while(1):
   d = raw_input('input k ')
   k = int(d)
   s.send(chr(k))  # send k as one byte
   if k == 0: break
   d = s.recv(1)  # receive v as one byte
   print ord(d)  # print v

s.close()
```

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

```
# simple illustration client of thread module

# multiple clients connect to server; each client repeatedly sends a
# value k, which the server adds to a global value g and echoes back
# to the client; k = 0 means the client is dropping out; when all
# clients gone, server prints final value of g

# the values k are sent as single bytes for convenience; it's assumed
# that neither k nor g ever exceeds 255

# this is the server

import socket
import sys

import thread

# thread to serve a client
def serveclient(c,i):
   global v,nclnt,nclntlock
   while 1:
      d = c.recv(1)
      k = ord(d)
      if k == 0:
         break
      vlock.acquire()
      v += k
      vlock.release()
      c.send(chr(v))
   c.close()
   vlock.acquire()
   nclnt -= 1
   vlock.release()

lstn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
port = int(sys.argv[1])
lstn.bind(('', port))
lstn.listen(5)

v = 0
vlock = thread.allocate_lock()
nclnt = 2
```

```
43 nclntlock = thread.allocate_lock()
44
45 for i in range(2):
46    (clnt,addr) = lstn.accept()
47    thread.start_new_thread(serveclient,(clnt,i))
48
49 # wait for both threads to exit
50 while nclnt > 0: pass
51
52 lstn.close()
53 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 1024.)

Input numbers into both clients, in a rather random pattern, then finally input 0 to both 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 sent input next, and thus doesn't know on which client to call **recv()**. This problem is solved by having threads, which run "simultaneously" and thus give the server the ability to read from whichever client has sent data.

So, let's see the technical details.

Lines 35-38: Here we set up the server, typical network operations.

Line 40: The variable **v** holds the running total mentioned in our earlier description of what the program does.

Line 41 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 in Line 16.

Lines 42-43: 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 when the count reaches 0 (Line 50).

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

[8] 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.

and it will turn out to need a lock too, which we name **ncltnlock**.

Lines 46-47: The server accepts a client connection, and then sets up a thread. The latter operation is done via **thread.start_new_thread()**, whose first argument is the name of the application function which the thread will run, and whose second argument is a tuple consisting of the set of arguments for that application function. So, here we are telling Python's threads system to call our function **serveclient()** (defined on Line 19) with the arguments **clnt** and **i**; the thread becomes active immediately.

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).

Line 50: 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 Line 50. This does serve our purpose of not allowing "main" to go past this line until both clients are gone, but on the other hand all that looping 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, but we have chosen the simplest way here.

Lines 21-29: Remember, this thread will deal with only one particular client, the one corresponding to the connection **c** (an argument to the function, in Line 19). So this **while** loop does nothing but read from that particular client. If the client has not sent anything, the thread will block on Line 22, 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.

Lines 22-23: Note that I read just one character, and then call **ord()** on it. This was a trick of convenience. To explain it, look at Lines 25-27 of the client, **clnt.py**: I read in the number **k**, as usual, on Line 25, and convert it from a string to a number on Line 26. But the funny stuff is on Line 27, which says, "Send the character whose code is **k**." If **k** is, say, 65, then 'A' will b sent. Remember, all this means is that the bit string 01000001 will go onto the network, so all I've done is tricked Python into sending an integer (65 here), which would normally occupy 4 bytes, in just 1 byte. That will work as long as the integer is at most 255, a restriction which is no problem for me in this little illustrative example.[9]

Lines 26-28: We are worried about a **race condition**. Suppose for example **v** is currently 12, and Client 0 sends **k** equal to 3. Line 27, when compiled to byte code, will be maybe two or three lines long. What if this thread's turn ends right in the middle of that code fragment? That thread will have found that **v** is 12, and will be ready to store 15 back into **v**. But if the turn ends, then Client 1 may send **k** equal to 9, say, and write 19 in **v**. When that turn ends, the first thread will execute again—and write 15 to **v**, a disaster.

[9] ASCII code only goes up to 127, but **ord()** will still produce values up to 255.

To deal with this, on Line 26, we call the **thread** module's lock **acquire()** function. The lock is initially unlocked. The first thread to execute this line will lock the lock, and if the other thread tries to execute this line, that thread will simply block until the first thread unlocks the lock by executing Line 28. That way we insure that not more than one thread is on Line 27 at a time.

Note 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 natural.[10]

Lines 31-33: The same race-condition issues apply here.

5 Second Example

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

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, i.e. a machine with more than one CPU, and if the threads system takes advantage of that. With several CPUs, several threads can run at the same time, rather than taking turns as is the normal case. Thus one could get a big speedup in the computation by using threads.

Another reason to use threads is for programming style purposes. A computation might naturally break down into several different parts, and it might be easier to conceptualize if we implement those parts as threads.

Here is a Python program that finds prime numbers using threads:

```
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     nstartedlock.acquire()
10    nstarted += 1
11    nstartedlock.release()
12    donelock[tn].acquire()
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
```

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

```
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()
```

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, 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.[11] 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 9-11. Each child thread increments a count, **nstarted**; meanwhile, on Line 43 the main thread is wasting time executing **pass**.[12] But as soon as the last thread increments the count, the main thread leaves its busy wait and goes to Line 44.[13]

Then back in each child thread, the thread acquires its **donelock** lock on Line 12, 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.

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 9-11 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."

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
```

[11] 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.

[12] In 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.

[13] Again, 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.

```
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 10 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).

6 Debugging

Debugging is always tough with parallel programs, including threads programs. It is especially difficult in Python, at least with the basic PDB debugger. One cannot, for instance, 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 is invoke PDB from *within* a child thread. In our primes-finding program, for instance, you could place the following code just before, say, Line 15:

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

Each thread will pause at Line 15, where you can set breakpoints, etc. However, it may be difficult to use.

7 Advanced Python Threads

Python's other threads module is **threading**. Most "Pythonistas" (as they call themselves) prefer this package, and it is indeed more convenient than, albeit more complex than, the basic **thread** module. Here we will merely present an overview.[14]

First of all, **threading** separates the creation of a thread from the actual start of its execution. For instance, in the primes example, the creation/start of the threads might be written as

```
threadlist = []
for i in range(nthreads):
   t = threading.Thread(target=dowork,args=(i,))
   threadlist.append(t)
for i in range(nthreads):
   threadlist[i].start()
```

This separation is actually a big advantage. Many threads applications have problems with threads starting "too early," and the separation shown here solves that problem neatly.

Here's another convenience: In our primes program above, we went to a lot of trouble to arrange things so that the main thread would not have to do (much of) a busy wait on its way to exiting. Using **threading**, this would be much easier, e.g. as:

```
for i in range(nthreads):
   threadlist[i].join()
```

which means that the main thread will not proceed any further until each child thread "joins" it, by exiting. Again, the main thread will "sleep" while waiting for this.

The **threading** module also has wait/signal operations, enabling one thread to "sleep" until "signaled" by another, and much more.

[14] By the way, the **threading** module is loosely modeled after Java threads.