Concurrency

Here's a really short Python program:

```
input('Enter something: ')
```

If you run this, you'll see the program sit there and wait for you to type something in. It can't do anything else until you enter something. A lot of times we would want our program to be able to do some other work while it is waiting for input. The tool for doing this is called a *thread*.

Threads are sort of like mini-processes. They aren't true processes because they don't stand on their own. They are always created by a parent process, and they share the address space of their parent. But they do get scheduled separately by the OS scheduler, so once the OS context switches away from one thread, another has a chance to run. On multicore CPUs, multiple threads from a program may be able to run at the same time.

Here is sort of a "Hello World" of using threads in Python:

```
from threading import *

def f():
    print('hello from thread 1')

def g():
    print('hello from thread 2')

t1 = Thread(target=f)
t2 = Thread(target=g)

t1.start()
t2.start()
```

There are two threads, `t1` and `t2`. Each thread has its own code to run given by the `target` option. The `start` method starts each thread running. The OS will choose which thread to run first. In all likelihood, `t1` will run first, but there's no guarantee of that.

Here is another example. If you want to try running it yourself, it's easier to copy and paste from the HTML version of these notes. Copy-pasting from the pdf often messes up the spacing.

```
from threading import *

def f():
    p = 2
    while True:
        for i in range(2, p):
            if p % i == 0:
                break
        else:
            primes.append(p)
        p += 1


def g():
    while True:
        input('Enter something: ')
        print('While you were typing, last prime found was', primes[-1])

primes = []

t1 = Thread(target=f)
```

```
t2 = Thread(target=g)

t1.start()
t2.start()
```

One of the threads in this program is continually generating prime numbers. The other one is continually asking the user to enter something. Both are running at the same time, sort of. The OS is context-switching back and forth between them (and other processes on the system).

About threading

Remember that one of the early innovations in operating systems was the realization was that while one process is doing I/O another can be using the CPU. Threads allow it so that while one part of your program is doing I/O, another part of your program can be using the CPU. A nice example of this is a program that has to download a bunch of webpages. If we download them sequentially, then if one of the downloads is slow, all of the downloads after it will have to wait until it is done. If we do each download in a separate thread, one slow download won't affect the others.

On a multicore system, you can use threads in order to have your program run on multiple cores at once. However, not in Python – Python has this annoying feature called the GIL (global interpreter lock), which is kind of important to the internals of Python, and it unfortunately makes it impossible for more than one thread to be running at the same time.

On a single core system, threads will give a speedup if some of them are doing a lot of I/O. But if all your threads are doing a lot of computation, then threads won't give a speedup, and in fact can even slow things down a little because of the overhead of switching processes. On a multicore system, threads will give a speedup even if they are all doing computation, as long as the number of threads does not exceed the number of cores.

Why threads? Why not just use multiple processes? Threads share their parent's address space, so it's easier for them to communicate with each other. They can share variables. Trying to get two separate processes to communicate is more painful.

A problem with threading

Here is a very important threading program.

```
from threading import *

def f():
    global count
    for i in range(1000000):
        count += 1

def g():
    global count
    for i in range(1000000):
        count += 1

count = 0

t1 = Thread(target=f)
t2 = Thread(target=g)

t1.start()
t2.start()
```

```
t1.join()
t2.join()

print(count)
```

There are two threads, `t1` and `t2`. Each one runs the exact same code. They both add 1 to a global counter variable 1,000,000 times. In theory, when both are done, the counter should equal 2,000,000. Note that the join method near the bottom basically says to wait until both threads are done. Once they are both done, the program prints out the value of `count`, which should hopefully equal 2,000,000.

Except it doesn't. I ran the program 5 times and these are the five different outputs I got: 1218764, 1425491, 1335768, 1292715, 1168259. These numbers are quite random. Try the program yourself to see. You'll end up getting different numbers entirely. If you do get 2,000,000, then try increasing 1,000,000 to 10,000,000.

Understanding why this weirdness happens is key to understanding what can go wrong with threads and concurrency. And the solution is probably not something that you would expect. It has to do with the fact that `count += 1` at the machine level is not a single operation. It is actually three operations: load, add, store. First the system has to load the value of `count` from memory and put it into a CPU register. Then it adds 1 to that value, and then it stores that value back in memory.

If a context switch happens in the middle of this process, things can go awry. Suppose thread `t1` runs first. It adds 1 to count a bunch of times until eventually the OS decides to context switch away from it. Let's suppose that it gets up to `count=8000` and is about to add 1 to the count. So it loads the value of `count` into a register, adds 1 to get 8001, and is just about to store that value back in `count`, when the context switch happens. The OS puts thread `t1` into a state of suspended animation at this point. It's frozen right before it can update the value of `count` to 8001.

Suppose that the context switch takes us to thread `t2`. It reads the value of `count`, which is currently 8000 and starts adding 1 to it. Maybe it gets to run for 10000 iterations and when the OS context switches away from it, `count` is now up to 18000. Then the OS context-switches back to `t1`. It picks up where it left off, which was to write the value 8001 into `count`. It does that, and with that all of the work `t2` did has been erased! The `count` variable is now at 8001 instead of 18001.

This is precisely what causes the random output when we run the program above. Sometimes the context switches happen at okay times and nothing bad happens, but sometimes they happen in the middle of a `count += 1` operation and then progress is lost. So instead of getting to 2,000,000, we end up getting to some value considerably less, like 1,218,764. Here is what happens in the order it happens:

1. Thread 1 runs first and adds 1 to count 8000 times. So the count is 8000.
2. Thread 1 gets interrupted in middle of an addition. It is frozen in between adding and storing the new value 8001.
3. Thread 2 runs and adds 1 to the count 10,000 times. The value of count is now 18000.
4. Thread 2 is interrupted and Thread 1 gets to run again.
5. Thread 1 finishes what it was trying to do earlier, which is to store 8001 in `count`. All of Thread 2's work is therefore destroyed.

This is our first example of what is called a *race condition*. A race condition is a programming problem where the output of the program is dependent on the exact state of a the computer or a particular sequence of events. In this case, the results of the program vary depending on exactly when the OS decides to context switch between threads.

Locks

Race conditions often arise when two or more threads share a common resource. In this case, the common resource is the `count` variable. The part of the code in which the shared resources is accessed is called a *critical* *section*. In this example, it's the `count += 1` line.

The solution to this problem is some type of *mutual exclusion*, where only one thread is allowed to be in the critical section at any given point in time. We will learn several ways to accomplish this. The first way is called a *lock*. Here is how we would use the lock in counting program above:

```python
def f():
    global count
    for i in range(1000000):
        lock.acquire()
        count += 1
        lock.release()

def g():
    global count
    for i in range(1000000):
        lock.acquire()
        count += 1
        lock.release()

count = 0

lock = Lock()

t1 = Thread(target=f)
t2 = Thread(target=g)

t1.start()
t2.start()

t1.join()
t2.join()

print(count)
```

We create the lock with the line `lock=Lock()`. In each of the two functions, we surround the critical section, `count += 1`, with `lock.acquire()` and `lock.release()`. This guarantees that only one thread can ever be updating the count at any one time. What happens is whenever one of the threads wants to update the count variable, it first checks if the lock is set. If it isn't set, then it sets the lock, goes into the critical section to update the count, and then unsets the lock. If the lock is set, then the thread will have to wait until the lock is free again before it can enter the critical section.

How locks are implemented

Because this is a course in OS, we want to know how the OS actually does this locking procedure. One way is called *test and set*.

The basic idea is the operating system uses a boolean flag variable for the lock. When a thread wants to enter the critical section, it checks if the flag is true or not. If it's false, then it enters the critical section and sets the flag to true. When exiting, it sets it back to false again. If the flag is true when it wants to enter the critical section, then it waits for the flag to become false before entering.

A nice analogy for this is a porta-potty. You never want more than one person at a time in one of those things. They have a little sign on the front that indicates if it is occupied or not. When someone wants to use the porta-potty, they check if it's free, go in if it is, set the sign to busy, and when they're done, they set the sign back to free. If the porta-potty is busy, then the person will wait until it becomes free before going in.

There is still a potential race condition with test and set. What if the thread checks the flag, sees that it's true, and right before it can set the flag to true, it gets interrupted by a context switch? Another thread then gets to run and goes in the critical section. A context switch back to the first thread happens, and it picks up where it left off, which was to set the flag to true and enter the critical section. Now we have two threads in the critical section.

In the porta-potty example, this would be like Person A checking the occupied sign on the porta-potty and seeing it is set to unoccupied. Just as they are about the enter, they get distracted by a text on their phone. While they are distracted, another person walks into the porta-potty. After Person A is done with their text, they go into the porta-potty without looking at the sign (because last they checked it said it was unoccupied). And now there are two people in there.

The solution to this is that a lot of CPUs have a specific instruction that does a test-and-set operation in a single, *atomic* operation. "Atomic" means that it can't be broken up. The test and the set both happen at the same time and no context switch can come between them.

Spin locks and futexes

Locks are usually implemented as either *spin locks* or as *futexes*. They differ in what they do if the flag is set. With a spin lock, a thread will sit and "spin" waiting, constantly checking the flag to see if it's false. With a futex, the process will go to sleep and wait for the OS to notify it when the flag becomes false. In fact, the OS puts all threads waiting for the lock on a queue.

Sticking with the porta-potty example, a spin lock is where someone waiting for the porta-potty sits and stares at it until the sign says it is unoccupied. A futex would be where there's an attendant (the OS) who will send you off to a waiting room and come to get you when the porta-potty is free again.

Here are some pros and cons of each approach:

- Spinning can be really wasteful on a single core system, especially if there are several threads all waiting for the same lock. For example, suppose we have threads A, B, C, D, and E, and the OS is running them round-robin. Let's say A has the lock and the others all want it. So A is running for a while and then its time slice ends while it is still using the lock. The OS switches to B. B will spend its entire time slice in a while loop constantly checking the lock to see if it's free. Of course it won't be free because it can't be freed up until A gets to run again. So that whole time slice is just wasted CPU cycles.

  Similarly, C, D, and E all waste their time slices. If each has a 20 ms time slice, then 80 of the 100 ms are totally wasted with no useful computation getting done. However, with a futex, all the other threads would be asleep, and A would get to run until it was done with the lock, at which point the OS could wake up the others. There would not be any CPU cycles wasted spinning.

- Spin locks can also be unfair. If there are several threads all waiting for the same lock, the one that gets it first will depend on whatever is happening with the OS scheduler. With a futex, the queue is first-come, first served.

- Spinning is easy to program and doesn't require much special work from the OS. A futex requires the OS to setup and maintain a queue. This setup can be expensive in terms of CPU cycles. If the lock is only held for a short while, especially on a multicore system, this overhead can be more trouble than it's worth.

Both approaches are used in modern operating systems.