Processes

In Lab 11 we will look at an advanced topic -- how to create and control processes on our own. Today we will just get started with some of the terminology.

First of all, a *process* is a set of instructions to be executed one after another. So far all of our programs have been single processes.

Computers can run more than one process at a time. While my laptop is displaying these slides, it is also keeping track of the time, and running other programs, some of which I have asked it to run and some of which are for the operating system.

A single processor can only run one process at a time. It appears that the processor is doing many things at the same time because it does a little bit of one, a little bit of the next, and so forth, and the "time slices" are so small and run so quickly that it appears they are happening at the same time.

However, most modern computers, even personal laptops, contain multiple processors.

There are two big reasons for writing programs that use multiple processes:

- If you design the program right, one part of the program can get work done while another part is waiting for something, such as user input or access to a file.
- Multiple processes can accomplish a big task more quickly than a single process.

The workflow for this is

a) Create a single function that does what we want a process to do.
b) Create a process to run that function.

Running Processes in Python

We usually run Python programs within the Idle application. This doesn't work with programs that spawn new processes; Idle uses its own processes and interferes with the creation of new ones. The easiest way to run the programs we will create in Lab 11 is to navigate in a terminal (or shell) window to the right folder and run the program foo.py with command

> python3 foo.py

On Windows I can run these programs by right-clicking on the program file and choosing *python* from the menu. You should be able to do something similar on Macs.

To see the program output on Windows I need to replace the usual call to main( ) with

```
if __name__ == "__main__":
    main()
    input()
```

This tells the operating system to run my main() function in the "__main__" process to which the standard output stream is attached. This happens by default in Linux systems, so you don't need to do this for the Lab 11 programs, but it doesn't hurt to include it.

Now, back to Python

We will make use of two Python modules:

os (which contains information from the operating system)
multiprocessing

As usual, before we can use modules we need to import them.

The primary class we will use is in the multiprocessing module. Its name is Process. The Process constructor takes many arguments that have default values. We will use only two of them:

multiprocessing.Process(target, args)

target is the name of the function you want to run in the process.

args is a **tuple** with the argument values for the function. Even if there is only one argument, put a comma after it.

Constructing a Process object creates a process to run the target function. We then run the process by calling its start( ) method.

For example, here is a short program that creates and runs a process:

```
import multiprocessing

def printer( stringToPrint, numTimes):
        for i in range(numTimes):
                print(stringToPrint)

def main():
       p = multiprocessing.Process(target=printer,
                args = ("This is fun!", 5) )
        p.start()

if __name__ == "__main__":
   main()
   input()
```

This gets more interesting if we change main( ) to

```
def main():
    p = multiprocessing.Process(target=printer,
            args = ("She loves me.", 5) )
    q = multiprocessing.Process(target=printer,
            args = ("She loves me not.", 5) )
    p.start()
    q.start()
```

Now the two processes interleave, producing results such as

She loves me.
She loves me not.
She loves me not.
She loves me not.
She loves me.
She loves me not.
She loves me.
She loves me.
She loves me not.
She loves me.