Parallel Processes in Python Documentation

CSInParallel Project

August 02, 2015

CONTENTS

1 For Instructors **2**

2 For Students **3**

3 Learning Goals **4**
- 3.1 Base Knowledge . . . 4
- 3.2 Conceptual/Applied Knowledge . . . 4

4 Prerequisites and Assumed Level **5**

5 Contents **6**
- 5.1 Basics of Processes with Python . . . 6
- 5.2 Creating multiple child processes . . . 7
- 5.3 Creating multiple child processes, part 2 . . . 8
- 5.4 Execution Order and Resource Contention . . . 9
- 5.5 Solution to Exercise . . . 12
- 5.6 Communication . . . 13
- 5.7 Queue Exercise Hint . . . 14
- 5.8 Queue Exercise Solution . . . 15
- 5.9 Coordination of Processes . . . 16
- 5.10 Process Coordination, Part 2 . . . 17

Author:

Steven Bogaerts
Department of Computer Science
DePauw University
602 S. College Ave, Greencastle, IN 46135, U.S.A.

CHAPTER

ONE

FOR INSTRUCTORS

This module examines some key concepts of parallelism that have particularly accessible Python implementations, making it suitable for an introductory computer science course (CS1). It considers how to create processes to work in parallel, how to use locks to manage access to shared resources, and how to pass data between processes. This is done using the `multiprocessing` module in Python.

Note: The code examples in this module conform to **Python 2**.

CHAPTER

TWO

FOR STUDENTS

You are about to study some key ideas of parallel computation. More specifically, we'll build on the knowledge of Python you've already gained, to see how to create programs that can make use of special computer hardware to literally do multiple things at the same time. We'll learn how to do this in Python using the `multiprocessing` module, but the ideas we explore here are broadly applicable. Thus they will serve a nice foundation for more study in future courses.

Start at the beginning section linked in the Contents below and work your way through, trying examples and stopping to complete code that you are asked to work on along the way. You should work through this tutorial by having your Python programming environment available as you read– this is designed for you to learn by doing.

CHAPTER

THREE

LEARNING GOALS

3.1 Base Knowledge

Students should be able to:

- Describe what a process is and how a running program can involve many processes.
- Describe why order of execution of multiple idependent processes is not guaranteed.
- Describe the challenge of shared resources in parallel computation.
- Describe the purpose of interprocess communication.

3.2 Conceptual/Applied Knowledge

Students should be able to:

- Write embarrassingly parallel programs in Python using the multiprocessing module.
- Manage access to a shared resource via a lock.
- Write programs involving simple interprocess communication via a shared queue.

CHAPTER

FOUR

PREREQUISITES AND ASSUMED LEVEL

At a minimum, students should have introductory-level experience in printing, variables, tuples, and writing and using functions with keyword arguments. Some basic understanding of object-oriented programming is also helpful: classes, methods, object state, constructors, and method invocation. For example, it would be quite sufficient for students to be able to write a "Student" class with major and GPA fields and supporting accessors and mutators.

CHAPTER

FIVE

CONTENTS

5.1 Basics of Processes with Python

Think about a time you worked with other people on some task. For example, you might have worked with some friends to shovel a driveway or complete a class project. You split the task into pieces, and each person worked at the same time to get the job done more quickly than would be possible by yourself. This is parallelism. In computing, *parallelism* can be defined as the use of multiple processing units working together to complete some task. There are many different kinds of hardware that can serve as a "processing unit", but the principle is the same: a task is broken into pieces in some way, and the processing units cooperate on those pieces to get the task done.

Every computing device has a central processing unit (CPU) that handles the running of a program. Have you heard of desktop computers or mobile devices being described as "dual-core" or "quad-core"? This is a reference to the number of processing units available on the CPU of that device. A computer with a dual-core CPU has two cores – two processing units – capable of working at the same time. Similarly, a quad-core CPU has four cores.

The challenge is that these cores don't get used to their greatest benefit automatically. Programs need to be written in a particular way to make effective use of the available cores. In this course module, we'll explore the use of the multiprocessing module in Python to write programs that can execute on multiple cores.

Before we dive into programming, let's consider what a *process* is. While the details can be rather complex and dependent on many factors, the big picture is simple. We can think of a process as a running program. A process has to keep track of what line in the code will be executed next, and what variable values are set. On a single-core processor, only one process actually runs at a time. This is in contrast to a multicore processor, in which multiple processes can be executed literally at the same time (limited by the number of cores, of course).

5.1.1 Making a Process

We are now ready to work with the `multiprocessing` module itself. Let's consider the code below. First note that we use `from multiprocessing import *` to gain access to the multiprocessing module. This will give us access to many useful tools, including the `current_process` function and the `Process` class used in this example.

```python
from multiprocessing import *

def sayHi():
    print "Hi from process", current_process().pid

def procEx():
    print "Hi from process", current_process().pid, "(parent process)"

    otherProc = Process(target=sayHi, args=())
    otherProc.start()
```

This code follows a common pattern: a *parent* process creates one or more *child* processes to do some task. In this example, suppose we call `procEx`. The first line in that function prints a simple message about what process is running. This is done by calling the function `current_process` that is defined in the multiprocessing module. The `current_process` function returns a Process object representing the currently running process. Every Process object has a public field called **pid**, which stands for "process identifier". Thus `current_process().pid` returns the pid for the currently running process. This is what gets printed.

Proceeding to the next line of the `procEx` function, observe that the `Process` constructor is called, passing two arguments by name. The purpose of this constructor call is to create a new `Process` object to be executed. The `target` argument specifies what function should be executed when the process under construction is actually started. The `args` argument is a tuple of the arguments to pass to the target function; since the `sayHi` target function takes no arguments, args is an empty tuple in this example.

It is important to note that by calling the `Process` constructor, we have *created* a `Process` object, but we have not yet *started* a new process. That is, the process exists, but is not available to be run yet. The process is actually started with the last line of `procEx`. The `start` method is defined in the `Process` class. It changes the state of the `Process` object on which it is called, such that the process is made available for execution.

So to summarize, there are two steps to make a child process do some task: A `Process` object must be created using the constructor, and then started using the `start` method.

So what does the child process do? It executes the `sayHi` function, as specified in the target argument of the `Process` constructor call. Thus it simply prints a message showing its pid. Note we use the same `current_process().pid` code here as in the parent, but this code will be executed by the child process, not the parent, and so the pid will be different. If you call the `procEx` method, you should receive output similar to the following:

```
Hi from process 3988 (parent process)
Hi from process 4828
```

Of course, your pids will likely be different, since these numbers are arbitrarily assigned by the operating system.

Try the code

`Download basic1.py` and try the above example on your system.

5.1.2 Making Multiple Processes

Let's extend what we've just looked at a little bit with a short exercise. Copy the code from the previous example and modify it to create three processes, each of which says "hi". Try this on your own now, before reading on.

5.2 Creating multiple child processes

A solution to the previous exercise is the following:

```
def procEx2():
    print "Hi from process", current_process().pid, "(parent process)"

    p1 = Process(target=sayHi, args=())
    p2 = Process(target=sayHi, args=())
    p3 = Process(target=sayHi, args=())
    p1.start()
    p2.start()
    p3.start()
```

Here we make three different `Process` objects. It is important to note that each process uses the same `sayHi` function defined before, but each process executes that function independent of the others. Each child process will print its own unique pid.

Let's push this a little further now, using a sayHi2 function that takes an argument. Observe the following code:

```
def sayHi2(n):
    print "Hi", n, "from process", current_process().pid

def manyGreetings():
    print "Hi from process", current_process().pid, "(main process)"

    name = "Jimmy"
    p1 = Process(target=sayHi2, args=(name,))
    p2 = Process(target=sayHi2, args=(name,))
    p3 = Process(target=sayHi2, args=(name,))

    p1.start()
    p2.start()
    p3.start()
```

Note in the `manyGreetings` function that we create three `Process` objects, but this time the `args` argument is not an empty tuple, but rather a tuple with a single value in it. (Recall that the comma after name is used in single-element tuples to distinguish them from the other use of parentheses: syntactic grouping.) With the `args` tuple set up in this way, `name` is passed in for `n` in the `sayHi2` function. So the result here is that each of the three child processes has the same name, "Jimmy", which is included in the child process's output. Of course, we could trivially pass distinct names to the children by adjusting the `args` tuple accordingly.

Try the code

`Download manyGreetings.py` and try the above example on your system.

5.2.1 Variable Number of Processes

Let's try another exercise now. Write a function that first asks for your name, and then asks how many processes to spawn. That many processes are created, and each greets you by name and gives its pid. Try this on your own before moving on. *Hint*: use a loop to create the number of desired child processes.

5.3 Creating multiple child processes, part 2

Here is a possible solution for creating a variable number of child processes:

```
def manyGreetings2():
    name = raw_input("Enter your name: ")
    numProc = input("How many processes? ")

    for i in range(numProc):
        (Process(target=sayHi2, args=(name,))).start()
```

5.3.1 Anonymous Processes

After obtaining the user's name and desired number of processes, we create and start that many Process objects with a loop. Note in this case that the single line of the loop body could also be written as two lines as follows:

```
p = Process(target=sayHi2, args=(name,))
p.start()
```

We can say that the one-line version includes the use of *anonymous* Process objects. They are anonymous since the individual objects are never stored in variables for later use. They are simply created and started immediately. The one-line version might look confusing at first, but note that (Process(target=sayHi2, args=(name,))) creates a Process object. We're then just calling the start method on that Process object, instead of storing it in a variable and calling start on that variable. For our purposes, the end result is the same.

Now, consider the following:

```
for i in range(numProc):
    pi = Process(target=sayHi2, args=(name,))
    pi.start()
```

This would work as well, as it merely substitutes variable p in the previous example for pi. However, it is important to point out that this code does **not** actually create several variables, p0, p1, p2, etc. Sometimes this kind of mistake happens because we're working in a different context now – parallel programming – but it's important to remember that the same programming principles you've already learned continue to apply here. For example, consider the following example:

```
for a in range(10):
    grade = 97
```

Clearly this code does not create the variables gr0de, gr1de, gr2de, etc. Similarly, then, pi does not become p0, p1, p2, etc.

Another important question can be considered in reviewing the `manyGreetings2` code above again. Which approach is better, the explicit use of p, or the anonymous version given in the original? It depends. In this example, we don't need access to the `Process` objects later, so there's no need to store them. So the anonymous version is acceptable in that regard. But we might also think about which version we find to be more readable. To an extent this may be a matter of personal opinion, but it is something that should be considered in programming.

5.4 Execution Order and Resource Contention

In addition to a pid, we also have the option of naming each child process. Any provided name is stored in the public name field defined in the `Process` class. For example, consider the following code:

```
def sayHi3(personName):
    print "Hi", personName, "from process", current_process().name, "- pid", current_process().pid

def manyGreetings3():
    print "Hi from process", current_process().pid, "(parent process)"

    personName = "Jimmy"
    for i in range(10):
        Process(target=sayHi3, args=(personName,), name=str(i)).start()
```

If we run `manyGreetings3`, the parent process says "Hi", and then creates and starts ten child processes. Each child process runs `sayHi3`, which requires a personName argument, specified in the args parameter of the Process constructor call. We also include one new argument in the Process constructor call: `name`. This `name` argument should be a string, and in this example we just use the string representation of the loop index variable i. Thus when a child process executes `sayHi3`, it has access to the `personName` given as an argument, and also has access to the `name` field provided in the call to the `Process` constructor.

Try to predict what will happen when you run the `manyGreetings3` function. Your first guess might be the following (with arbitrary pids, of course):

```
Hi from process 3988 (main process)
Hi Jimmy from process 0 pid 5164
Hi Jimmy from process 1 pid 5236
Hi Jimmy from process 2 pid 6884
Hi Jimmy from process 3 pid 3652
Hi Jimmy from process 4 pid 1060
Hi Jimmy from process 5 pid 1767
Hi Jimmy from process 6 pid 5812
Hi Jimmy from process 7 pid 4732
Hi Jimmy from process 8 pid 3564
Hi Jimmy from process 9 pid 4332
```

It's possible that the processes will print out very nicely and in order like the above, but it's extremely unlikely. First note that each core of the CPU is a *scarce resource*, meaning there aren't typically enough cores for every process to use one whenever it wants. On a quad-core system, for example, up to four processes can execute at once. If there are more than four processes wanting to execute, some will need to wait.

So the operating system maintains a list of waiting processes. When a core becomes available, the operating system chooses another process to execute on that core. But the process created first is not necessarily the next one chosen. That is, just because we *create and start* the processes in the order 0 through 9 in our program, it doesn't mean that the operating system will choose them to *execute* in the order 0 through 9. So, for example, we might expect output like this:

```
Hi from process 3988 (parent process)
Hi Jimmy from process 8 pid 3564
Hi Jimmy from process 2 pid 6884
Hi Jimmy from process 6 pid 5812
Hi Jimmy from process 0 pid 5164
Hi Jimmy from process 3 pid 3652
Hi Jimmy from process 9 pid 4332
Hi Jimmy from process 4 pid 1060
Hi Jimmy from process 7 pid 4732
Hi Jimmy from process 1 pid 5236
Hi Jimmy from process 5 pid 1767
```

In fact, any ordering of the child processes' messages is possible. The only thing we know for certain is that the message from the parent process will show up first, since our code specifies that that should happen before we create any child processes. There are ways to ensure that certain tasks are completed before other tasks, as we'll see later in this module. But by default, *child processes execute in arbitrary order*, and parallel programs must be designed with this in mind.

Unfortunately, we still haven't captured what will likely happen when we run the code given above. Go ahead and run it now and see. Results will vary, but you may see something very garbled up like the following:

```
Hi from process 3988 (main process)
Hi HJHimmyiHii   Jimmy  from process 0 -JfJ pid 5164
immyrom process 4 - pid 4332
immy from process  from process 7 - pid8  5236- pid
 3564
Hi Jimmy from process 1 - pidH 6884
i Jimmy from process 5 - pid 3652
Hi Jimmy from process 5 - pid 1060
Hi Jimmy from process 2 - pid 176
Hi Jimmy from process 3 - pid 5812
Hi Jimmy from process 9 - pid 4732
```

What's going on? The first thing to realize is that the CPU cores are not the only scarce resource here. Standard output – where printing occurs – is also scarce. More specifically, standard output is a single *shared* resource that multiple processes are trying to access at the same time. So the processes have to take turns. As it is, our program is not forcing the processes to take turns in any reasonable way. How can we fix this? We'll use a *lock*.

5.4.1 Using a Lock to Control Printing

One excellent way to begin our study of locks is by analogy to a concept from the novel by William Golding (or the 1963 and 1990 film adaptations). The novel tells the story of a group of boys shipwrecked on a deserted island with no adult survivors. Before an eventual breakdown into savagery, the boys conduct regular meetings to decide on issues facing the group. The boys quickly realize that, left unchecked, such meetings will be unproductive as multiple boys wish to speak at the same time. Thus a rule is developed: Only the boy that is holding a specially-designated conch shell is allowed to speak. When that boy is finished speaking, he relinquishes the conch so that another boy may speak. Thus order is maintained at the meetings as long as the boys abide by these rules. We can also imagine what would happen if this conch were not used: chaos in meetings as the boys try to shout above each other. (And in fact this does happen in the story.)

It requires only a slight stretch of the events in this novel to make an analogy to the coordination of multiple processes accessing a shared resource, like standard output. In programming terms, each boy is a separate process, having his own things he wishes to say at the meeting. But the air around the meeting is a shared resource - all boys speak into the same space. So there is contention for the shared resource that is this space. Control of this shared resource is handled via the single, special conch shell. The conch shell is a *lock* – only one boy may hold it at a time. When he releases it, indicating that he is done speaking, some other boy may pick it up. Boys that are waiting to pick up the conch are not allowed to say anything – they just have to wait until whoever has the conch releases it. Of course, several boys may be waiting for the conch at the same time, and only one of them will actually get it next. So some boys might have to continue to wait through multiple speakers.

The following code shows the analogous idea in Python.

```python
def sayHi4(lock, name):
    lock.acquire()
    print "Hi", name, "from process", current_process().pid
    lock.release()

def manyGreetings4():
    lock1 = Lock()

    print "Hi from process", current_process().pid, "(main process)"

    for i in range(10):
        Process(target=sayHi4, args=(lock1, "p"+str(i))).start()
```

At the start of `manyGreetings4`, the constructor of the `Lock` class is called, with the resulting object stored in the variable `lock1`. This single `Lock` object, along with a distinct name, is passed to each of the child processes. Each child process wants to print something when it executes `sayHi4`. But print writes to `stdout` (standard output), a single resource that is shared among all the processes. So when multiple processes all want to print at the same time, their output would be jumbled together were it not for the lock, which ensures that only one process is able to execute its print at a time.

How does the lock accomplish this? Through the use of the acquire and release methods, both defined in the Lock class. Suppose process A acquires the lock and begins printing. If processes B, C, and D then execute their acquire calls while A has the lock, then B, C, and D each must wait. That is, each will *block* on its acquire call. Once A

releases the lock, one of the processes blocked on that lock acquisition will arbitrarily be chosen to acquire the lock and print. That process will then release the lock so that another blocked process can proceed, and so on.

Note that the lock must be created in the parent process and then passed to each child – this way each child process is referring to the same lock. The alternative, in which each child constructs its own lock, would be analogous to each boy bringing his own conch to a meeting. Clearly this wouldn't work.

As in the previous example, the order of execution of the processes is still arbitrary. That is, the acquisition of the lock is arbitrary, and so subsequent runs of the code are likely to produce different orderings. It is not necessarily the process that was created first, or that has been waiting the longest, that gets to acquire the lock next.

Try the code

Download `manyGreetings4.py` and try the above example on your system.

5.4.2 You try it: Digging Holes

Let us now try an exercise extending the concept of locks above. Imagine that you have 10 hole diggers, named A, B, C, D, E, F, G, H, I, and J. Think of each of these as a process, and write a function assignDiggers() that creates 10 processes with these worker names working on hole 0, 1, 2, ..., 9, respectively. Each one should print a message about what it's doing. When you're done, you should get output like the following (except that the order will be arbitrary):

```
Hiddy-ho!  I'm worker G and today I have to dig hole 6
Hiddy-ho!  I'm worker A and today I have to dig hole 0
Hiddy-ho!  I'm worker C and today I have to dig hole 2
Hiddy-ho!  I'm worker D and today I have to dig hole 3
Hiddy-ho!  I'm worker F and today I have to dig hole 5
Hiddy-ho!  I'm worker I and today I have to dig hole 8
Hiddy-ho!  I'm worker H and today I have to dig hole 7
Hiddy-ho!  I'm worker J and today I have to dig hole 9
Hiddy-ho!  I'm worker B and today I have to dig hole 1
Hiddy-ho!  I'm worker E and today I have to dig hole 4
```

Try to complete this exercise before moving on.

5.5 Solution to Exercise

A solution is to the hole-digging exercise is as follows:

```python
def dig(workerName, holeID, lock):
    lock.acquire()
    print "Hiddy-ho!  I'm worker", workerName, "and today I have to dig hole", holeID
    lock.release()

def assignDiggers():
    lock = Lock()
    workerNames = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J"]

    for holeID in range(len(workerNames)):
        Process(target=dig, args=(workerNames[holeID], holeID, lock)).start()
```

The `assignDiggers` function creates a single Lock object and a list of the worker names. A process is started for each worker, passing the appropriate name, assigned hole, and the lock. Each worker attempts to acquire the lock, and is only allowed to print once that acquisition succeeds. After printing, the worker releases the lock so that another worker may print.

This exercise is also a good demonstration of the strengths and limitations of different approaches to looping. The solution shown above uses what can be referred to as a "loop-by-index" approach, in which the holeID index is the loop variable. An alternative would be a "loop-by-element" (for-each loop) approach, like this:

```
... # Other code as before
for workerName in workerNames:
    Process(target=dig, args=(workerName, workerNames.index(workerName), lock)).start()
```

The loop-by-element approach, however, is not as effective, because the worker-Names.index(workerName) requires a fair amount of extra work to execute. While the actual execution time will be nearly instantaneous in both approaches for a small list, it is nevertheless a good idea to reiterate the general principle of using the right programming constructs for maximum efficiency. You don't want to fall into the trap of using a less efficient choice on a larger list of data in some other circumstance, or in a circumstance where you might execute such a loop over and over many times, where the time used will add up.

5.6 Communication

In most real-world applications of parallelism, some amount of communication between processes is required. We have already seen one way in which processes can communicate: a parent process can send data to children through the args parameter of the Process constructor. Now we are ready to look at a more flexible means of communication.

The Queue class (pronounced like the letter "Q") defines a Queue object that a parent can pass to children so that multiple processes have access to it. A queue can be thought of as a collection of data. Any process can put data onto the queue using the put method, and take data off the queue using the get method. Thus one process could do a put, and another a get, in order to transmit data. If a process attempts a get when there is nothing on the queue, then the process will wait (*block*) on the line of code where the get occurred until some other process does a put on the queue.

Let's look at this in the following example:

```
import time

def greet(q):
    print "(child process) Waiting for name..."
    name = q.get()
    print "(child process) Well, hi", name

def sendName():
    q = Queue()

    p1 = Process(target=greet, args=(q,))
    p1.start()

    time.sleep(5) # wait for 5 seconds
    print "(parent process) Ok, I'll send the name"
    q.put("Jimmy")
```

Note the use of a queue for communication between processes, which in this case is the variable `q`, which is a Python multiprocessing `Queue` object. When `sendName` is run, the following output results:

```
(child process) Waiting for name...
(parent process) Ok, I'll send the name
(child process) Well, hi Jimmy
```

At the start of the sendName function, the Queue constructor is called, with the resulting Queue object stored in the variable q. This object is passed to the child process. So the child process is in fact using the same queue as the parent. The child is started, and then the parent does nothing for 5 seconds, via the time.sleep(5) command. In the meantime, since the child has started, the first print in greet is executed, followed by the call to get. The child's get is a *blocking* command. This means that the child process will go to sleep until it has a reason to wake up – in this case, that there is something to get off the queue. Since the parent sleeps for 5 seconds, the child ends up blocking for approximately 5 seconds as well. Finally the parent process sends the string "Jimmy", the child process unblocks and stores "Jimmy" in the variable name, and prints its final message.

Try the code

`Download sendName.py` and try the above example on your system.

5.6.1 Extended Communication Via a Queue

Let's try some quick practice now that you've worked through the previous example. Copy the code above as a basis for `greet2` and `sendName2`. Modify the code so that `greet2` expects to receive 5 names, which are sent by `sendName2`. Each function should accomplish this by sending/receiving one name at a time, in a loop. Spend some time on this before moving on.

5.7 Queue Exercise Hint

Did you get your code working? The exercise may feel challenging at first because the context is new. Let's try to organize our thoughts a bit with some pseudocode:

```
'''
def greet2():
    for 5 times
        get name from queue
        say hello

def sendName2():
    queue
    make a child process, give it the queue
    start it

    for 5 times
        sleep for a bit
        put another name in the queue
'''
```

When we work in psuedocode, it frees us from having to think about new syntax and other details all at once. Instead we're free to get some big picture ideas down first. In the above psuedocode you can see that the parent process, in `sendName2`, will make a queue and a child process. It will then loop five times, sending one piece of data at a time to the child via the queue. The child, in `greet2`, will also loop five times, getting something from the queue and printing. Recall that if the child attempts to get something from the queue when there's nothing there, it will block until something is available to get. If you didn't already solve this problem, try again now, using the pseudocode as a guide.

With this pseudocode developed, the actual code comes much more easily.

5.8 Queue Exercise Solution

Recall the English pseudocode for our original simple examle problem of sending 5 pieces of data from a parent to a child process:

```
'''
def greet2():
    for 5 times
        get name from queue
        say hello

def sendName2():
    queue
    make a child process, give it the queue
    start it

    for 5 times
        sleep for a bit
        put another name in the queue
'''
```

Here is a Python solution that follows the structure of the pseudocode very closely. It's just a matter of filling in the syntax we're learning for queues, along with a review of working with processes and how we can sleep for a randomly defined amount of time.

```
from random import randint

def greet2(q):
    for i in range(5):
        print
        print "(child process) Waiting for name", i
        name = q.get()
        print "(child process) Well, hi", name

def sendName2():
    q = Queue()

    p1 = Process(target=greet2, args=(q,))
    p1.start()

    for i in range(5):
        time.sleep(randint(1,4))
        print "(main process) Ok, I'll send the name"
        q.put("George"+str(i))
```

Try the code

Download `sendName2.py` and try the above example on your system.

Once you are comfortable with this example of using queues to communicate data and coordinate the handling of that data using `put` and `get`, you will be ready to look at some other coordination mechanisms in the next section.

5.9 Coordination of Processes

5.9.1 The Join Method

In parallel programming, a *join* operation instructs the executing process to block until the process on which the join is called completes. For example, if a parent process creates a child process in variable `p1` and then calls `p1.join()`, then the parent process will block on that join call until p1 completes. One important point to emphasize again in this example is that the *parent* process blocks, not the process on which join is called (`p1`). Hence the careful language at the start of this paragraph: the executing process blocks until the process on which the join is called completes.

The word "join" can be confusing sometimes. The following example provides an analogy of the parent process waiting (using join) for a "slowpoke" child process to catch up.

```
def slowpoke(lock):
    time.sleep(10)
    lock.acquire()
    print "Slowpoke: Ok, I'm coming"
    lock.release()

def haveToWait():
    lock = Lock()
    p1 = Process(target=slowpoke, args=(lock,))
    p1.start()
    print "Waiter: Any day now..."

    p1.join()
    print "Waiter: Finally! Geez."
```

The child process is slow due to the `time.sleep(10)` call. Note also the use of a lock to manage the shared use of stdout.

It should be pointed out, however, that join is not always necessary for process coordination. Often a similar result can be obtained by blocking on a queue get, as described in the previous section and later in this section.

Try the code

`Download haveToWait.py` and try the above example on your system.

5.9.2 Obtaining a Result from a Single Child

While earlier examples demonstrated a parent process sending data to a child via a queue, this example allows us to practice the other way around: a child that performs a computation which is then obtained by the parent. Consider two functions: `addTwo-Numbers`, and `addTwoPar`. `addTwoNumbers` takes two numbers as arguments, adds them, and places the result on a queue (which was also passed as an argument). `addTwoPar` asks the user to enter two numbers, passes them and a queue to addTwo-Numbers in a new process, waits for the result, and then prints it.

Consider the following starting code:

```
def addTwoNumbers(a, b, q):
    # time.sleep(5) # In case you want to slow things down to see what is happening.
    q.put(a+b)

def addTwoPar():
    x = input("Enter first number: ")
    y = input("Enter second number: ")
    q = Queue()
    p1 = Process(target=addTwoNumbers, args=(x, y, q))
    p1.start()
```

The parent passes two numbers inputted by the user, and a shared queue, to a child process, `p1`, which will execute `addTwoNumbers`. The child process puts the sum of the numbers onto the queue, with an optional sleep call before, to slow the computation down for illustrative purposes.

Here is an exercise for you to consider now: starting with the above code, which you can `download as addTwoPar.py`, write code to make the parent obtain the result from the child and print it. Do not move on until you have done this.

5.10 Process Coordination, Part 2

5.10.1 Solution to Exercise

The completed exercise is as follows:

```
def addTwoNumbers(a, b, q):
    # sleep(5) # In case you want to slow things down to see what is happening.
    q.put(a+b)

def addTwoPar():
    x = input("Enter first number: ")
    y = input("Enter second number: ")

    q = Queue()
    p1 = Process(target=addTwoNumbers, args=(x, y, q))
    p1.start()

    # p1.join()
    result = q.get()
    print "The sum is:", result
```

As you can see, it requires only a small addition. The parent must call the get method on the queue. Once the child has put something on the queue, the parent's get will succeed, and the variable result will get a value and be printed.

Did you attempt to use join in your solution, as in the commented-out line in the above solution? In this example the join is not harmful, but is not required. This is because the get will already cause the parent process to block until data is on the queue. So there's no need for the parent process to wait for the child to be finished with a join as well. The get already causes the required wait.

5.10.2 Using a Queue to Merge Multiple Child Process Results

The following example is a nice extension of the one above.

```
from multiprocessing import *
from random import randint
import time
def addManyNumbers(numNumbers, q):
    s = 0
    for i in range(numNumbers):
        s = s + randint(1, 100)
    q.put(s)
```

```
def addManyPar():
    totalNumNumbers = 1000000

    q = Queue()
    p1 = Process(target=addManyNumbers, args=(totalNumNumbers/2, q))
    p2 = Process(target=addManyNumbers, args=(totalNumNumbers/2, q))
    p1.start()
    p2.start()

    answerA = q.get()
    answerB = q.get()
    print "Sum:", answerA + answerB
```

Here, the task is to add up a large number of random numbers. This is accomplished by creating two child processes, each of which is responsible for half of the work. Note that a shared queue, plus half the total number of numbers, is passed to each child. Each child creates many random numbers and adds them up, putting the final sum on the queue. The parent makes two get calls, to obtain the result from each child. Note that the parent will likely block on at least the first get call, since it will need to wait until one of the children finishes and places its result on the queue.

Here's an interesting question to consider: which child's result will be in answerA and which in answerB? The answer is that this is indeterminate. Whichever child process finishes first will have its answer in answerA, and the other will be in answerB. This is not a problem for commutative merging operations, like the addition of this example, but of course could be a complication for non-commutative merging.

Try the code

`Download addManyPar.py` and try the above example on your system.

5.10.3 Conclusion

Of course there are many places you could go next, but here we have seen an introduction to parallel programming in Python using the multiprocessing module. We've explored the parent-child model of parallel programming, in which the parent creates many child processes to perform some task. We've seen how shared resources lead to a need for locks to ensure uninterrupted access. Finally, we've seen how to pass data between processes, both via the Process constructor's `args` argument, and also through the use of a shared queue.