Shared Data

The examples we have seen so far have the processes completely independent of each other. In many problems the processes need to coordinate by sharing some of the variables.

The multiprocessing module has a class for this:
multiprocessing.ctypes.RawValue( )

The names here are getting really long. To simplify this, we will import the modules in a different way.

If we say

```
        import multiprocessing
```

then we can use things in the multiprocessing module, but we need to prefix the multiprocessing name:

```
        multiprocessing.Process( )
```

If we say

```
        from multiprocessing import *
```

we can use them without prefixing the module name.

In the past when we have used the random module we said

```
import random
```

and used it as

```
x = random.randint(0, 10)
```

If we said instead

```
from random import *
```

how would we call the randint function?

A) `x = randint(0, 10)`
B) `x = random.randint(0, 10)`
C) `x = .randint(0, 10)`
D) `x = rand.int(0, 10)`

To make a shared variable that represents an integer, we use

```
r = RawValue( "i", <integer value> )
```

such as

```
r = RawValue( "i", 0 )
```

This needs to be created outside of the function the processes will work on, and passed as an argument to them.

<FirstSharedExample.py>

You need to be careful about how you work with shared data because the processes can modify it asynchronously.

<SecondSharedExample.py>

<ThirdSharedExample.py>

Locks

The last example had 10 processes each incrementing a shared variable 10 times. The final value of the variable should be 100 greater than its starting value but it almost never is. Several processes each read the value of the variable, perhaps this is 55, and they each write the next value, 56, into it. Instead of incrementing the variable they are overwriting its value.

Imagine what would happen if deposits to a bank account worked this way -- you start with $100, deposit $50 and then deposit $25 and find that your final balance is only $75. If we are going to write programs where processes share data we need to have some way to guarantee data integrity.

There are several solutions to this. We are going to use a simple solution called a Lock.   Lock is a class in the multiprocessing module. The constructor takes no arguments, so we create a lock with Lock( ).

There are only two methods for the Lock class: acquire( ) and release( ). Acquiring the lock puts it in its locked state; releasing it unlocks it.

The important thing about a lock is that when a process tries to acquire it, the process is put on hold until the lock becomes available. So if a function has code

```
    blah blah
    lock.acquire( )
        critical section
    lock.release( )
```

then only one process at a time can execute the critical section. Every other process that wants to execute this section has to wait until the lock is released so they can acquire it.

To make this work we generally create the lock outside of the function and pass it in as an argument. For example, the function that increments a shared variable i might be

```
def F(r, lock):
        for i in range(0, 10):
                lock.acquire()
                r.value =  r.value + 1
                print( "Process %d set r to %d" %
                                (current_process().pid, r.value))
                lock.release()
```

We need to be careful to match acquisitions and releases of locks. This code will never finish running:

```
def F(r, lock):
    for i in range(0, 10):
        lock.acquire()
        r.value =  r.value + 1
        print( "Process %d set r to %d" %
                    (current_process().pid, r.value))
    lock.release()
```

Why not???