CONCURRENCY 7b

COMPUTER SCIENCE 61A

August 8, 2013

1 Concurrency

On your computer, you often use multiple programs at the same time. You might be reading Piazza posts on your internet browser, talking to friends through instant messaging, streaming music from Spotify, and maybe using a hundred other programs. But you only have one computer, and one CPU! How can so many programs run all at once?

Since a computer program is a series of instructions and a *process* is the execution of those instructions, what actually happens is that the CPU switches between different processes very quickly, doing a little for each process before moving on to the next. This creates the illusion that the programs are running concurrently.

This is because a process can contain many *threads*, all of which access the memory and instructions allocated to a certain process. In a multi-threaded process, different threads can be working on different sections of the code, or on different inputs to the process, "concurrently."

Today's discussion is on *parallelism*, and the *concurrency* issues that might arise from having multiple programs run simultaneously.

As a general note, parallel computing is extremely important in computer science today, because it is becoming harder to increase the speed of processors due to physical heat limitations. To achieve more processing power, computers now utilize multiple processors - your computer probably has at least two "cores" in its processor.

1.1 Decomposing a Python Statement

Before we can talk about concurrency, we need to first understand what happens under the hood when the interpreter executes a Python statement.

Typically, a Python statement can cause the computer to

- **Retrieve** values from memory

- **Compute** new values

- **Store** values into memory

For example:

```
y = 5               store 5 -> y


x = y * 3           load y: 5
                    compute y*3: 15
                    store 15 -> x


print(x)            load x: 15
                    compute (call function print) 15
```

When two processes are run in parallel, it is not clear which one will start first, or when the CPU will switch from one to the other.

For example, consider the following:

```
def one():
    print('hello')
    print('world')

def two():
    print('jom magrotker')
```

Running `one()` and `two()` in parallel can yield any one of the following possibilities:

Possibility 1	Possibility 2	Possibility 3
hello	hello	jom magrotker
world	jom magrotker	hello
jom magrotker	world	world

Exactly what the interpreter prints will be different for every computer, and will even be different for different trials on the same computer!

1.2 Questions

1. Consider the following code:

```
>>> def make_withdraw(balance):
        def withdraw(amount):
            nonlocal balance
            if amount > balance:
                print('Insufficient funds')
            else:
                balance = balance - amount
                print(balance)
        return withdraw
>>> w = make_withdraw(10)
```

What are all the possible pairs of printed statements that could arise from executing the following 2 lines *in parallel*?

```
>>> w(8)
>>> w(7)
```

2. Suppose that Steven, Joy, and Albert decide to pool some money together:

```
>>> balance = 100
```

Now suppose Steven deposits $10, Joy withdraws $20, and Albert withdraws half of the money in the account by executing the following commands:

```
Steven: balance = balance + 10
Joy: balance = balance - 20
Albert: balance = balance - (balance / 2)
```

List all the different possible values for balance after these three transactions have been completed, assuming that the banking system **forces the three processes to run sequentially in some order**.

3. What are some other values that could be produced if **the system allows the processes to be interleaved**?

For the following questions, what are possible values that x and y could have if the given two threads are run concurrently?

4. Starting with x = 1,

```
>>> x = x * 2                              >>> x = x + 10
```

5. Starting with x = 1, y = 1,

```
>>> x = x + 5                              >>> x = x + y + 1
>>> y = 3
```

6. Starting with x = 1,

```
>>> if x % 2 == 0:                         >>> x = x * 2
...      x = x + 1
... else:
...      x = x + 100
```

2   Shared State

Problems arise in parallel computation when one process influences another during critical sections of a program.

Critical sections are sections of code that need to be executed as if they were a single instruction, but are actually made of several statements. For a parallel computation to behave correctly, the critical sections need to have *atomicity* – a guarantee that these sections will not be interrupted by any other code.

There are several methods of synchronizing our code, but they all share the same idea – each method has some way of signalling to other processes that they are currently handling shared data, and that other processes should not modify the data.

Think of traffic lights at an intersection: the only thing that stops a driver at a red light is the shared understanding that everyone will stop when they see a red light; there is **no physical mechanism** that actually makes people stop!

2.1 Locks

Locks are shared objects that are used to signal that shared state is being read or modified. They are also referred to as a *mutexes* (short for "mutual exclusion"). In Python, a process can acquire and release a lock, using the `acquire` and `release` methods respectively. Consider the following code:

```
>>> from threading import Lock
>>> lock = Lock()
>>> def one():
        lock.acquire()     # 'one' acquires 'lock'
        print('hello')
        print('world')
        lock.release()     # 'one' releases 'lock'
>>> def two():
        lock.acquire()     # 'two' acquires 'lock'
        print('jom')
        lock.release()     # 'two' releases 'lock'
```

While a lock is acquired by a process, any other process that tries to perform the `acquire` action will be required to wait until the lock becomes free. Only one process can acquire a lock at a time.

For a lock to "protect" a set of variables, all code blocks that deal with those variables should be surrounded by `acquire()` and `release()` calls.

Let's assume that `one()` and `two()` run in parallel, with `one()` starting first. Visually, it looks like this:

```
1 | one()                # one starts
2 | lock.acquire()       # acquires lock   two()               # two starts
3 | print('hello')                         lock.acquire()      # two can't acquire lock
4 | print('world')
5 | lock.release()       # releases lock
6 |                                        print('jom')        # two can acquire lock
7 |                                        lock.release()      # releases lock
```

2.2 Questions

1. Rewrite the `make_withdraw` function from the previous question such that the critical sections are protected by the lock.

   ```
   >>> from threading import Lock
   >>> def make_withdraw(balance):
           balance_lock = Lock()
   ```

2. What are the possible pairs of printed values if the following code is now run?

   ```
   >>> w = make_withdraw(10)
   >>> w(8) #these 2 lines are executed in parallel
   >>> w(7) #these 2 lines are executed in parallel
   ```

2.3 Semaphores

Sometimes we want to grant more than one thread access to a resource. We use a synchronization construct called a *semaphore* to accomplish this. Semaphores are similar to locks, except that they can be acquired multiple times up to a limit.

The state of the semaphore starts at some positive integer, n, and follows these rules:

- Each time `acquire` is called, n *decreases* by 1.

- Each time `release` is called, n *increases* by 1.

- If a process tries to acquire the semaphore while n is 0, then that process will be required to wait until n is greater than 0 (i.e. another process called `release()`).

Here is an example of how we would model a parking garage with a specific number of parking spots and two entrances using a semaphore. In our parking garage,

- We are given car objects that want to enter and exit the garage.

- A car cannot enter unless it is guaranteed to have a spot.

- We want to keep a list of cars currently in the garage.

- A car can always exit the garage.

```
TOTAL_SPOTS = 10
open_spots = Semaphore(TOTAL_SPOTS)
cars_lock = Lock()
cars = []
def enter_garage(car):
    # entrances call this function when a car wants to enter
    open_spots.acquire()
    cars_lock.acquire()
    cars.append(car)
    cars_lock.release()
def exit_garage(car):
    # entrances call this function when a car exits
    cars_lock.acquire()
    cars.remove(car)
    open_spots.release()
    cars_lock.release()
```

If a car wants to enter the garage, it must acquire the semaphore. Once ten cars have entered the garage, the semaphore's count goes down to 0. This prevents another car from entering until a car currently in possession of the semaphore has released it.

2.4 Deadlock

Deadlock is a situation that occurs when two or more processes are stuck, waiting for each other to finish. The following two functions will deadlock if run concurrently:

```python
lock1 = Lock()
lock2 = Lock()

def foo():
    lock1.acquire()
    lock2.acquire()
    print('hello')
    print('world')
    lock1.release()
    lock2.release()

def bar():
    lock2.acquire()
    lock1.acquire()
    print('jom')
    lock2.release()
    lock2.release()
```

Visually, it looks like this:

```
1 |                                      | two()              # two starts
2 | one()            # one starts        | lock2.acquire      # acquires lock2
3 | lock1.acquire()  # acquires lock1    | lock1.acquire()    # can't acquire lock1
4 | lock2.acquire()  # can't acquire lock2|
```

one has to wait for two to release lock2 before one can continue – but two has to wait for one to release lock1 before two can release lock2! Thus, the two functions are stuck in deadlock.

When writing programs that utilize concurrency, you have to design your code in such a way that it avoids deadlock.

2.5 Question

1. Modify the following code such that `compute` and `anti_compute` will avoid dead-lock, but will still be able to run in parallel without corrupting data.

```
>>> x_lock = Lock()
>>> y_lock = Lock()
>>> x = 1
>>> y = 0
>>> def compute():
        x_lock.acquire()
        y_lock.acquire()
        y = x + y
        x = x * x
        x_lock.release()
        y_lock.release()
>>> def anti_compute():
        y_lock.acquire()
        x_lock.acquire()
        y = y - x
        x = sqrt(x)
        y_lock.release()
        x_lock.release()
```

2.6 Message Passing

An alternative way for handling concurrent computation is to avoid sharing memory altogether, thus avoiding the problems we've seen above. Instead, we let computations behave independently, but give them a controlled way in which that can send messages to each other to coordinate.

Suppose for example that we want to map a function `foo` onto the elements of a list, but the `foo` function takes a very long time. Instead of running `foo` on just one element at a time, we could run `foo` on all the cores inside our computer. The different threads of computation could then pass their results back to the main thread which can put them together again.

Without concurrency, we might have the following code:

```
def foo(n):
    # Do something that takes a really long time
    ...
L = list(range(n))
M = list(map(foo, L))
```

1. If foo is $O(n^3)$, how long does this code take for input size is n?

2. Consider instead the code below, which uses (an unwritten) `MessageReceiver` class which implements a method for sending and receiving messages. The idea is that n many independent processes get run which will pass their results back to the main thread. This thread will then receive those results and put the answers together.

   Assuming that you now have n many processors to run the computation on, what is the running time of the entire process?

   ```python
   def run_computation_thread(main_thread, n):
       result = foo(n)
       main_thread.send((n, result))
   L = list(range(n))
   M = list(range(n))
   main_thread = MessageReceiver()

   # Start n many threads running
   for i in range(n):
       # This creates an independent process which will run
       # the target function passed to it
       thread = Thread(target = \
               lambda: run_computation_thread(main_thread, L[i]))
       thread.start()# Receive results from all the threads

   for i in range(n):
       # Get the next result, or wait until someone sends us one.
       result = main_thread.receive()
       M[result[0]] = result[1]
   ```

3. The advantage of message passing is that nowhere in the code do we actually need to worry about critical sections or deadlocks. However, unlike `Locks` and `Semaphores`, Python doesn't have a built-in class for message passing. Instead, fill in a definition for `MessageReceiver` which implements the send/receive methods using `Locks` and `Semaphores`.

```
class MessageReceiver:
    def __init__(self):
        self.__semaphore = Semaphore(0)
        self.__messages_lock = Lock()
        self.__messages = []


    def send(self, message):








    def receive(self):
```

4. We saw above how we can use Locks to avoid incorrect results when multiple ATMs try to withdraw from an account simultaneously. Now consider how we might solve a similar problem using message passing. Suppose we have a number of processes acting as ATMs running concurrently with the main Bank process. Make sure you understand why this code doesn't have critical sections the same way the Lock example did.

```
class Account:
    def __init__(self):
        self.balance = 0
    def deposit(self, x):
        self.balance += x
    def withdraw(self, x):
        self.balance -= x
class Bank:
    def __init__(self):
        self.accounts = {}
    def create_account(account_num):
        self.accounts[account_num] = Account()
```

```python
def atm_process(bank_receiver):
    my_receiver = MessageReceiver()
    while True:
        account_num = input('Account Number: ')
        action = input('withdraw/deposit: ')
        amount = input('Amount: ')
        # Send a command to the bank and get a result back.
        # Print the result.
```

```python
def bank_process(bank):
    my_receiver = MessageReceiver()
    while True:
        """ Receive a message and process it. Send the string
        'OK' back to the ATM if the command was valid (i.e.
        was 'deposit' or 'withdraw') and send back
        'Bad Command' otherwise."""
```