CS61A Notes – Week 12 – Concurrency

On your computer, you often have multiple programs running at the same time. You might have your internet browser open reading the latest Piazza posts, your instant messaging client open to talk to your friends about the new homework assignment, Pandora open to stream some tunes, and maybe about a hundred other programs running. But you only have one computer, and one CPU! How can you do so many things at once?

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, creating the illusion that the programs are running 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 and 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.

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

- Load certain values out of memory
- Compute a new value
- Store something back into memory

This is best illustrated by example. Look at the decomposition of Python statements below, and make sure you understand them.

```
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
```

Questions:

```
>>> 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)
```

Question 1)
What are all the possible pairs of printed statements that could arise from executing the following 2 lines? Assume that the 2 function calls are executed in parallel.

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

Question 2)
Suppose that Steven, Aki, and Eric decide to pool some money together:

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

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

Steven: balance = balance + 10
Aki: balance = balance - 20
Eric: balance = balance - (balance / 2)

a. 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.

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

Protecting shared state
Parallel computing would not be very useful if unexpected and "incorrect" results occurred often when running our programs. Luckily, there are ways to make sure that certain pieces of code are executed without being interrupted.

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. In order for a parallel computation to behave correctly in concurrent situations, the critical sections need to have atomicity - a guarantee that these sections will not be interrupted by any other code.

We will discuss several methods of synchronizing and serializing our code but they all share the same underlying idea - they use variables in shared state as signals that all processes understand and respect. 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.

Locks

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

While a lock is acquired by a process, any other process that tries to perform the acquire() action will automatically be made to wait until the lock becomes free. This way, only one process can acquire a lock at a time. For a lock to protect a particular set of variables, all the processes need to be programmed to follow a rule: no process will access any of the shared variables unless it owns that particular lock. In effect, all the processes need to "wrap" their manipulation of the shared variables in acquire() and release() statements for that lock.

Question 1)

Protect the critical section of the make_withdraw function by acquiring and releasing the lock.

```
>>> from threading import Lock
>>> def make_withdraw(balance):
        balance_lock = Lock()
        def withdraw(amount):
            nonlocal balance
            if amount > balance:
                print("Insufficient funds")
            else:
                balance = balance - amount
                print(balance)
        return withdraw
```

Question 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
```

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 natural number. Each time acquire is called, this state is decremented by 1, and each time release is called, the state is incremented by 1. The semaphore has a

restriction that its state must be above 0. If we try to acquire the semaphore while its state is 0, then we will wait until the acquire becomes legal (that is, until someone calls release).
As a comprehensive example of how to use locks and semaphores, 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()
    lock.acquire()
    cars.append(car)
    lock.release()
def exit_garage(car):
    # entrances call this function when a car exits
    open_spots.release()
    lock.acquire()
    cars.remove(car)
    lock.release()
```

Deadlock
Deadlock is a situation that occurs when two or more processes are stuck, waiting for each other to finish.

Fill in the following code such that if compute() and anti_compute() are run in parallel, then deadlock might occur.

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

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 in order 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))
```

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

Question 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 have N many processors to run the computation on, how long does this version of the code take?

```
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(len(L)):
    # 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]
```

Question 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):
```

Question 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()
```

```
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.
        # YOUR CODE HERE
```

```
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.
        # YOUR CODE HERE
```