CONCURRENCY AND MAPREDUCE 15

COMPUTER SCIENCE 61A

August 3, 2014

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.

When a computer sees a line of code in Python, it is usually too complex for the computer to do it all in one step. To understand how a computer executes a program, we use the basic model of a computer with a processor and memory. A computer uses *memory* to store variables and their values, and the *processor* to process and compute data and values.

However, the issue is that the processor itself cannot remember more than a handful of values, and memory cannot do any calculation with the values that it stores. So a computer must coordinate between the two by breaking down Python expressions into steps that fall under three categories:

- **Load** values from memory to the processor
- **Compute** new values in the processor
- **Store** values from the processor into memory

For example, the following three lines in Python break down into the following computer steps:

```
Python Code:          Machine Instructions:
y = 5                 store 5 -> y

x = y * 3 + 2         load y: 5
                      compute 5*3: 15
                      compute 15 + 2: 17
                      store 17 -> x

print(x)              load x: 17
                      compute (call function print) 17
```

1.2 Parallelism

The basic idea behind parallelism is efficiency through multi-tasking. Imagine that you (a processor) are at your apartment, and you have several tasks (threads) that you need to complete: buying groceries, doing laundry, cleaning up the living space and calling your ISP to get your internet fixed. You could do this serially, but you can end up wasting some time (such as waiting for the wash and dry cycles to finish, or listening to hold music for an hour) that could have been better spent doing other things. By doing your tasks in parallel, you can get everything done more quickly than if you had done things serially.

Of course, imagine how much faster everything will get done if you got your apartment-mate to help you out as well.

Computers parallelize by having their processor(s) rapidly alternate between multiple threads, by executing a random number of steps before switching to another random thread and repeating the process. However, when two processes are run in parallel, it is not clear which one will start first, or when the processor will switch from one to the other.

For example, consider the following:

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

def two():
    print('CS 61A')
```

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

Possibility 1	Possibility 2	Possibility 3
hello	hello	cs61a
world	cs61a	hello
cs61a	world	world

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

1.3 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)
```

Solution: [2, "Insufficient..."] [3, "Insufficient..."] [2,3] [3,2] [2,2] [3,3] [2,-5] [3,-5] [-5,-5]

2. Suppose that Jonathan, Ajeya, and Matthew decide to pool some money together:

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

Now suppose Jonathan deposits $10, Ajeya withdraws $20, and Matthew withdraws half of the money in the account by executing the following commands:

```
Jonathan: balance = balance + 10
Ajeya: balance = balance - 20
Matthew: 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**.

Solution: 35, 40, 45, 50

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

Solution: 25, 30, 35, 40, 45, 50, 55, 60, 70, 80, 90, 110

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
```

Solution: 12, 22, 2, 11

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

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

Solution: y is always 3, x can be 8, 10, 3, 6, and 5 (which happens when thread 1 interrupts between the loading of x and the loading of y of thread 2)

6. Starting with `x = 1`,

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

Solution: 202, 3, 101, 2, 102

2 Shared State

As you can see, problems can 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()
```

Solution:

```
    def withdraw(amount):
        nonlocal balance
        balance_lock.acquire()    # acquire here
        if amount > balance:
            print('Insufficient funds')
        else:
            balance = balance - amount
            print(balance)
        balance_lock.release()    # release here
    return withdraw
```

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
```

Solution: [2, "Insufficient..."] [3, "Insufficient..."]

2.3 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:

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

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

def two():
    lock2.acquire()
    lock1.acquire()
    print('cs61a')
    lock2.release()
    lock1.release()
```

Visually, it looks like this:

		`two()` `# two starts`
`one()`	`# one starts`	`lock2.acquire` `# acquires lock2`
`lock1.acquire()`	`# acquires lock1`	`lock1.acquire()` `# can't acquire lock1`
`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.

2.4 Questions

1. Modify the following code such that `compute` and `anti_compute` will avoid deadlock, 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()
```

Solution: One solution is to make `compute`'s acquire/release pattern be:

```
x.acquire
y.acquire
y.release
x.release
```

And `anti_compute`'s acquire/release pattern be:

```
x.acquire
y.acquire
y.release
x.release
```

2.5 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`?

   Solution: $O(n^4)$

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?

   ```
   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]
```

Solution: $O(n^3)$

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):
```

Solution:

```
        self.__messages_lock.acquire()
        self.__messages.append(message)
        self.__messages_lock.release()
        self.__semaphore.release()
```

```
    def receive(self):
```

Solution:

```
        self.__semaphore.acquire()
        self.__messages_lock.acquire()
        message = self.__messages.pop()
        self.__messages_lock.release()
        return message
```

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.
```

Solution:

```
bank_receiver.send((account_num, action, amount, my_receiver))
result = my_receiver.receive()
print(result)
```

```
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."""
```

Solution:

```
msg = my_receiver.receive()
if (msg[1] == 'deposit'):
    bank.accounts[msg[0]].deposit(msg[2])
    msg[3].send('OK')
elif (msg[1] == 'withdraw'):
    bank.accounts[msg[0]].withdraw(msg[2])
    msg[3].send('OK')
else:
    msg[3].send('Bad Command')
```

3 MapReduce (Optional)

The core idea behind efficient parallelism is splitting up work using threads among multiple processors. The examples that we've talked about so far have involved a computer with one or more processors. But why limit ourselves to one computer?

MapReduce uses the compututational power of a cluster of nodes (or a network of computers) to process large amounts of data much more quickly than just one could. There are three phases in the MapReduce execution model:

- **Map Phase -** A *mapper* is applied to the inputs, and it emits key-value pairs
- **Sort Phase -** All the intermediate key-value pairs are grouped by keys.
- **Reduce Phase -** For each intermediate key, a *reducer* is applied to "accumulate" all the values associated with the key.

MapReduce coordinates the work in the cluster by breaking up the input into smaller subproblems and assigning them to each node, assigning new work when nodes finish their jobs, and rescheduling work if a node runs into an error or is working slowly. When nodes finish their work, they emit the results so that another node doing the next step can process those values. The process continues until finally the entire job is done.

3.1 Mappers and Reducers

Since the MapReduce framework is an abstraction, all we need to do is provide a *mapper* function, which will emit key-value pairs from the inputs, and a *reducer* function, which will combine all the intermediate key-value pairs with the same key value.

Suppose for example we wanted to count the number occurences of each word in Shakespeare's works. Our mapper should emit each word once:

```
#!/usr/bin/env python3

import sys
from ucb import main
from mapreduce import emit

def emit_words(line):
    for word in line.split():
        emit(word, 1)

for line in sys.stdin:
    emit_words(line)
```

Then each key-value pair is just an occurrence of each word. So then after our key-value pairs are sorted by key, all our reducer needs to do is count the number of occurrences of each word:

```
#!/usr/bin/env python3

import sys
from ucb import main
from mapreduce import emit, group_values_by_key

for key, value_iterator in group_values_by_key(sys.stdin):
    emit(key, sum(value_iterator))
```

3.2 Questions

1. Suppose instead of counting the occurence of each word, we just wanted the count of the number of total words as well as the number of lines. How would the `map` and `reduce` functions be written?

   Map function:

   ```
   def emit_words(line):
   ```

   Solution:

   ```
       for word in line:
           emit('word', 1)
       emit('line', 1)
   ```

   ```
   for line in sys.stdin:
       emit_words(line)
   ```

   Reduce function:

   ```
   for key, value_iterator in group_values_by_key(sys.stdin):
   ```

   Solution:

   ```
       emit(key, sum(value_iterator))
   ```

2. In Project 2, we processed quite a large number of tweets, and were able to analyze how states in the U.S. felt about certain words. But data that we analyzed doesn't come close to qualifying as big data. If we wanted to analyze all the tweets in the U.S. from a year, or even just a week, our implementation quickly becomes infeasible. We could, however, approach this problem using MapReduce instead. Finish the implementation of the `map` and `reduce` functions.

   Map function:

   ```
   def emit_tweet_sent(closest_state, sentiment):
   ```

   Solution:

   ```
       emit(closest_state, sentiment)
   ```

   ```
   for tweet in sys.stdin:
       closest_state, sentiment = analyze_tweet(tweet)
       emit_tweet_sent(closest_state, sentiment)
   ```

   Reduce function:

   ```
   for key, value_iterator in group_values_by_key(sys.stdin):
   ```

   Solution:

   ```
       total, count = 0, 0
       for sent in value_iterator:
           total += sent
           count += 1
       emit(key, total/count)
   ```

3. As part of a project, you and your team have measured the atmospheric concentration of various greenhouse gases in various cities around the globe. You want to process the data by finding the average atmospheric concentration of each greenhouse gas component in each city. Finish the implementation of the `map` and `reduce` functions.

   Map function:

   ```
   def emit_city_greenhouse_concentration(city, measurements):
   ```

   Solution:

```
    for i in range(len(gases)):
        emit((city, gases[i]), measurements[i])
```

```
gases = ['CO2', 'H2O', 'CH4', 'NO2', 'O3']
for line in sys.stdin:
    city, CO2, H2O, CH4, NO2, O3 = line.split()
    emit_city_gas_concentration(city, (CO2, H2O, CH4, NO2, O3))
```

Reduce function:

```
for key, value_iterator in group_values_by_key(sys.stdin):
```

Solution:

```
    total, count = 0, 0
    for measurement in value_iterator:
        total += measurement
        count += 1
    emit(key, total/count)
```