Question 1 – Markov models

At work, Wally sits at his desk drinking coffee, occasionally getting up to either go to the bathroom or go to the cafeteria for another cup of coffee. On the way back to his cube (from either the cafeteria or the bathroom), there is a 50% probability that he will stop to complain about work to Alice before returning to his cube.

On average Wally gets a cup of coffee once for every hour that he sits at his desk, and goes to the bathroom once for every two hours he sits at his desk. It takes on average 15 minutes to get coffee in the cafeteria, 10 minutes to use the bathroom, and 10 minutes to complain to Alice. Wally's transitions between these activities are memoryless.

a) Add transition rates (in events/hour) to the diagram of the Markov model for Wally's day.

Here we note that the mean time spent in a state before transitioning to another one is the inverse of the transition rate. Thus:

- 2 hours in cube per bathroom break = 0.5 breaks/hour
- 10 minutes (mean time for bathroom or complaining) = 6 transitions / hour. Thus if Wally is currently in the bathroom, there is approximately a 10% (6/60) chance that he will exit in the next minute.
- 15 minutes = 4 transitions/hour

b) Set up the balance equations and solve for the fraction of time that Wally spends in states D (desk), C, B, and A.

Note that the flow rate across an edge is the steady state probability of the source state times the edge transition rate, and flow into a state must equal flow out of that state. Thus:

\[
(0.5 + 1)D = 3B + 6A + 2C \quad (1)
\]
\[
4C = 1D \quad (2)
\]
\[
6B = 0.5D \quad (3)
\]
\[
6A = 3B + 2C \quad (4)
\]

and \[A + B + C + D = 1 \quad (5)\]
We can solve this by back-substitution:

\[
\begin{align*}
B &= \frac{D}{12} \quad \text{(from #3)} \\
C &= \frac{D}{4} \quad \text{(#2)} \\
A &= \frac{D}{24} + \frac{D}{8} \quad \text{(#6,7,4)(8)} \\
(1/24 + 1/8 + 1/12 + 1/4 + 1) D &= 1 \quad \text{(from #5..8)}
\end{align*}
\]

giving:

\[
\begin{align*}
35/24 D &= 1 \\
D &= 24/35 \quad (0.6857), \ C = 6/35 \quad (0.1714), \ B = 2/35 \quad (0.0571), \ A = 3/35 \quad (0.0857)
\end{align*}
\]

c) During an 8 hour day, on average how many times does Wally complain to Alice? There are several valid ways to derive this result; please explain your calculation.

You can either (i) count the number of times Wally finishes complaining to Alice, or (ii) the number of times he starts.

(i) The steady state probability of state A (complaining to Alice) is 0.0857 or 3/35; across an 8 hour day he spends 24/35 hours complaining to Alice. At an average complaint duration of 1/6 of an hour, that corresponds to 24·3/35 or 144/35 (4.114) complaints per day,

(ii) The arrival rate into state A is \( P_{B \rightarrow A} \cdot r_{B \rightarrow A} + P_{C \rightarrow A} \cdot r_{C \rightarrow A} \) which is 3·2/35 + 2·6/35, or 18/35; across an 8-hour day this adds up to 144/35 times that he stops by Alice's cube to complain.

**Question 2 - Synchronization**

An infinite line of taxis with ID numbers 1, 2, ... is lined up on Huntington Ave; each taxi can hold two people. As students arrive individually at the cab stand they enter the first taxi in line; when the taxi is full (i.e. two students) it leaves. If the taxi is not full - i.e. it is only holding one student - then it waits however long it needs to until another student arrives.

We simulate this with a function `taxi()`. (note that this function might better be called “student_arrival”) The arrival of a student at the taxi stand is represented by a thread calling `taxi();` the function blocks and does not return until the corresponding taxi leaves. The function returns an integer giving the ID number of the taxi which the student left in.

Give pseudo-code for a monitor or semaphore implementation of this function.

The solution to this problem needs to keep track of two pieces of state – the ID number of the current taxi, and whether it is empty or holding a student. Simple semaphore and monitor solutions are very similar, using either a monitor condition or a semaphore initialized to 0 for the first student to wait on:

```java
Semaphore lock = semaphore(1) 
Semaphore inTaxi = semaphore(0) 
Boolean firstStudent = false 
Int taxiID = 1 

function student_arrival returns int <taxi ID>: 
    wait(lock) 
    firstStudent = NOT firstStudent 
    myTaxiID = taxiID 
    if firstStudent 
        signal(lock) 
        wait(inTaxi) 
    else 
        tmp = taxiID++ 
        signal(inTaxi) 
        signal(lock) 
    return myTaxiID
```

```java
monitor taxiStand: 
    Boolean firstStudent = false 
    Int taxiID = 1 
    Condition inTaxi 

method student_arrival returns int <taxi ID>: 
    firstStudent = NOT firstStudent 
    myTaxiID = taxiID 
    if firstStudent 
        wait(inTaxi) 
    else 
        signal(inTaxi) 
        taxiID++ 
        return myTaxiID
```
Points to note:

- In the semaphore case you need a binary semaphore (i.e. a mutex) to protect the shared variables, but you only need one.
- The first student needs to drop that lock before going to sleep
- In both the monitor and semaphore solution, the first student can’t trust the value of taxiID across the wait(), as the second student thread will update it during the wait. (the 'taxiID++' statement could be moved to the first student thread, after the wait(), but there’s a race between the second thread returning from wait() and a third student entering.
- Finally, there’s a simple way to combine the taxiID and firstStudent variables – if you number students from 0, then taxiID = 1+floor(studentNum/2), and firstStudent = isEven(studentNum).

**Question 3 – Virtual Memory**

[...]

The first two instructions at location 08000000 are as follows:

```assembly
08000000: MOV *(09000070), EAX
08000004: PUSH EAX
```

Page faults occur when the instruction fetch or a data read/write refer to an address that is not currently mapped in the page tables; when the page fault returns, the faulting instruction is re-tried.

Give the sequence of events from process start until completion of the second instruction, where each event is one of:

- instruction attempt – specify instruction address
- page fault – specify (a) address of instruction, (b) address of the faulting access.
- page allocation – specify physical page number returned
- page table updated – specify (a) physical page modified, (b) index [0...1023] of entry written, (c) page number written into entry
- disk read – specify physical page written to.

1. Instruction attempt at address 0x08000000
2. Page fault on instruction fetch – PC = 0x08000000, fault address = 0x08000000
   At this point there is no virtual memory mapped, so anything that the CPU tries to do is going to fault.

   The page number of the fault address is 08000, which splits as so:

   | 0000 | 1000 | 00 | 00 | 0000 | 0000 |

   where the top 10 bits are 00 0010 0000, or 020 hex. To map this address we need to allocate a page, and another page for the 2nd level page table.

3. Page allocation, returns 00001, for the page table
4. Page table update – update root table (00000), index 020, value (page pointed to) of 00001
5. Page allocation, returns 00002, for the data page
6. Page table update – update page 00001 at index 000 with value 00002.
7. Now we can read the program in from disk to physical page 00002...
8. and retry: instruction attempt at 0x08000000. This time the instruction fetch succeeds, but...
9. Page fault: instruction address 08000000, data address 09000070
10. Again we need a page table: page allocation returns 00003
11. Page table update: update 00000 at index 024, value written = 00003
12. Data page: page allocation returns 00004
13. Page table update: update 00003, index 0, value written = 00004
14. Instruction attempt at 08000000, and finally completes
15. Instruction attempt at 08000004
16. Page fault – instruction addr 08000004, fault address FFF00FFC (FFF003= splits into 3FF, 300)
17. Page allocation returns 00005
18. Page table update – update root (00000) index 3FF = 00005
19. Page allocation returns 00006
20. Page table update – update 00006 index 300 = 00006

The final configuration looks like this:

![Page Configuration Diagram]

**Question 4 – Context switching**

*The following diagram shows stacks, variables, and code for two threads switching via the yield12() function of homework 1.*

sp1: 0x800 = ...
sp2: 0x804 = 200
0x200,204,208 = 500, 7, 12
SP = 118, PC = 300

d0_switch(sp1, sp2_ptr):
0x400 MOV *(SP+8),EAX // EAX = sp2_ptr
0x404 MOV SP,*EAX    // *sp2_ptr = SP
0x408 MOV *(SP+4),EAX // EAX = sp1
0x40C MOV EAX,SP     // SP = sp1
0x410 RET

d0_yield12():
0x300  PUSH 0x800 // &sp2
0x304  PUSH *(0x804) // sp1
0x308  CALL 0x400 // d0_switch(sp1, &sp2)
0x30C  ADD 8,SP

0x500  POP EAX // EAX = stack.pop()
0x504  POP EBX // EBX = stack.pop()

--- DONE ---

a) Starting with the stack pointer and program counter given above, the CPU will execute 10 instructions before reaching “---DONE---”, with PC=0x508. Give values for SP, EAX, and EBX at this point, and write in the values which have been pushed on the thread one stack (the left-hand one) in the proper locations.

[...]

[Image of the diagram]
The 10 instructions and their effects are:

1. 300 PUSH 0x800  \( \text{SP}=114, *114 = 800, \text{PC}=304 \)
2. 304 PUSH *(0x804)  \( \text{SP}=110, *110 = 200, \text{PC}=308 \)
3. 308 CALL 0x400  \( \text{SP}=10C, *10C=30C, \text{PC}=400 \)
4. 400 MOV *(SP+8),EAX  \( \text{EAX}=800, \text{PC}=404 \)
   \( \text{note that SP+8 = 114, and *(SP+8) is the first argument, &sp1} \)
5. 404 MOV SP, *EAX  \( \text{SP}=10C, *10C=30C, \text{PC}=408 \)
6. 408 MOV *(SP+4),EAX  \( \text{EAX}=200, \text{PC}=410 \)
7. 40C MOV EAX, SP  \( \text{SP}=200, \text{PC}=410 \)
8. 410 RET  \( \text{PC}=500, \text{SP}=204 \)
9. 500 POP EAX  \( \text{EAX}=7, \text{SP}=208, \text{PC}=504 \)
10. 504 POP EBX  \( \text{EBX}=12, \text{SP}=20C, \text{PC}=508 \)

Thus SP=20C, EAX=7, EBX=12

b) For some reason we wish to add a third argument and a return value to do_switch:

\[ \text{val} = \text{do\_switch(new\_sp, old\_sp\_ptr, arg)}; \]

and have it always return the value of this argument as the return value. Describe how this could be done. (there are multiple methods, but the simplest and most general one is the best answer)

Here I forgot to mention that by convention the return value of a function is placed in the EAX register before the RET instruction. Sorry.

Assuming that we call do_switch from thread 1, switch to thread 2, and then switch back to thread 1. We note that in the first call to do_switch, 'arg' is already on thread 1's stack, at *(SP+12). So we can just wait until we return back to thread 1 (which happens at address 40C, whether for thread 1 or any other thread), and then load *(SP+12) to EAX to be our return value.

Alternately we could push an extra copy on the stack, and then pop it as a return value before returning. Storing it in a global only received partial credit, as this doesn't scale to >2 threads.