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.


**Question 1 (continued)**

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

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.
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.
Question 3 – Virtual Memory

A process has the following 32-bit address space consisting of 3 defined memory pages:

- 0xFFF00xxx – (i.e addresses 0xFFF00000 through 0xFFF00FFF) – stack. Allocated on demand as a zero-filled page.
- 0x09000xxx – data, allocated on demand as a zero-filled page.
- 0x08000xxx – code, paged in on demand from file /bin/myprogram

There is a memory allocator which allocates physical memory pages in sequence – i.e 00000, 00001, 00002, etc. (note that no pages get freed in this exercise) Newly allocated pages have already been zeroed.

The operating system initializes this process with the following state:

- program counter: 08000000
- stack pointer: FFF01000

It allocates a single physical page (page 00000) for the page directory, but leaves all entries in the directory empty (i.e invalid) Thus at process start the operating system knows the memory map for the process, but the hardware does not.

The first two instructions at location 08000000 are as follows. (note that PUSH decrements the stack pointer by sizeof(argument) – 4 in this case – and then stores the indicated value. In C: int *stack; *(--stack) = value;)

- 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.

There should be 3 page faults, 6 page allocations, 6 page table updates, and one disk read.

(diagram and space for answer on next page)
Initial configuration:

```
| PC: 08000000 |
| SP: FFF0800 |
| CR3: 00000 |
```

order of events:

```
| Page 00000 |
| index: 0 |
| ... |
| 1023 |
```
Question 4 – Context switching

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

Note that (a) all instructions and data are assumed to be 4 bytes, (b) PUSH X is equivalent to *(--SP) = X; i.e it decrements the stack pointer before storing the value, and CALL pushes the address of the instruction immediately after the CALL before jumping to the called address. (conversely, POP X reads from the location pointed to by the stack pointer and stores it in X before incrementing the stack pointer; RET reads the return address from the location pointed to by the stack pointer before incrementing it.)

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.

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

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

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)
Definitions and other helpful material

Semaphore: A semaphore is an object with an initial value N and two methods: down() and up(). Calls to down() can block, while calls to up() do not block. Note that you don't implement semaphores - you instantiate them:

\[ S = \text{semaphore}(N) \] -- create a semaphore with initial value N

Semaphore behavior is defined thusly:
Given a semaphore S with initial value N: at any point in time, if M threads have called S.up(), then no more than M+N threads have returned from S.down().

Monitor: A monitor is a particular kind of user-defined class, which has:
- regular variables
- methods
- conditions --- this is what normal classes don't have

Threads can:
- enter the monitor by calling any of the methods
- leave the monitor by returning from a method
- leave the monitor by calling wait()
- re-enter the monitor by returning from wait()

If there are any threads waiting on condition C:
- C.signal() will release one of them and
- C.broadcast() will release all of them.
- If there are no threads waiting, neither C.signal() nor C.broadcast() have any effect.

Only one thread can be in the monitor (i.e. entered but not left) at any given time. If thread A is in the monitor and thread B tries to (a) call a method or (b) return from wait(), thread B will block until thread A leaves the monitor. (or longer – see below)

When a thread calls signal() or broadcast() it does not leave the monitor, so that you can be sure that it will continue to run before any thread that it woke up.

However, if you wake thread A via signal() or broadcast(), and thread B tries to enter the monitor by calling a method, you don't know whether A or B will get to run first.

Hexadecimal numbers: Hex numbers are base 16, with numerals 0-9, A, B, C, D, E and F. The following is a table converting between decimal, hexadecimal, and binary:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>1010</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>1011</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>1100</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>1101</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>1110</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>1111</td>
</tr>
</tbody>
</table>