Question 1 – Monitors

Here's a sample solution:

Monitor:

```java
int numInEntry
bool doorHeld
condition wentThru

method holder:
    numInEntry++
    delay 5
    if doorHeld:
        go through
    else if numInEntry == 1
        open door, go through,
        close door
    else:
        doorHeld = True
        while numInEntry > 1:
            wait(wentThru)
        doorHeld = False
    numInEntry--
    signal wentThru
end method
```

```java
method non-holder:
    numInEntry++
    delay 5 [cross entryway]
A    if doorHeld:
    go through
else:
B    open door, go through,
    close door
    numInEntry--
    signal wentThru
end method
```

Notes:

A. You need to test *after* you finish delaying. (i.e. the stick figure crosses the entryway first, then checks the door, rather than checking the state of the door 5 seconds before arriving at it)

B. The way the question is phrased, non-holders always go through in exactly 5 seconds – if there is no one holding the door they open it, go through, and close it. (they don't wait for a following door-holder)
**Question 2 – Context switching**

The figure to the right shows most of the steps. Note that proc1 calls getline, but that getline then returns to the beginning of proc2, losing the first line of input. The full sequence is:

q4.1 call do_switch, pushing &tmp (300C) and sp[0] (1FFC) and a return address. We won't worry about what initial stack pointer value is.

do_switch.1,2 save current stack pointer into tmp.

do_switch 3,4 set SP to 1FFC and return, popping 1000 (proc1) as the return address

proc1.1,2 – push buf_1 (3200) and call getline. SP=1FF8 on entry to getline.

give_1 – call get_term_in, which copies the first line of input into buf_g (3100) and returns 1.

give_2,3,4 – since 'curr' starts at 0, we take the 'if' and call do_switch(&sp[0],sp[1]) where sp[1] = 2FFC

do_switch.1,2 – save current SP (1FEC) into sp[0]

do_switch.3,4 – load SP with 2FFC and return, popping 2000 (proc2) as the return address.

proc2.1,2 – push buf_2 (3300) and call getline, pushing return address

getline.1 – call get_term_in(buf_g), reading 2nd line into buf_g (3100) and returning 1

getline.2 – 'curr' = 1, so we skip the 'if' body

give_5,6 – copy input string from buf_g (3100) to buf_2 (3300) and return

proc2.3,4 – print 2nd line of input, loop back to top of proc2

proc2.1,2 – call getline

give_1 – call get_term_in, which blocks waiting for a 3rd line of input.
Question 3 – Markov Models

(a) Label the arcs with rates in transitions/minute:

(b) Give the balance equations and solve for the fraction of time the system is in states I, A, B, and Q.

The balance equations for each of the 4 states are:

1. \((1+3)P_I = 6P_Q + 3P_B\)
2. \(1P_I = 4P_A\)
3. \(4P_A + 5P_B = 5P_Q\)
4. \(3P_I = (3+5)P_B\)

We'll omit equation 3, and add the probability sum constraint:

5. \(P_I + P_A + P_B + P_Q = 4P_A\)

Since this was assigned on an in-class test, it can be easily solved by back-substitution:

6. \(P_A = \frac{1}{4} P_I\) from (2)
7. \(P_B = \frac{3}{8} P_I\) from (4)
8. \(4P_I = 6P_Q + \frac{3\cdot 3}{8} P_I\) from (1,7) \(\rightarrow\) \(P_Q = \frac{23}{48} P_I\)
9. \(1 + \frac{1}{4} + \frac{3}{8} + \frac{23}{48}\) \(\rightarrow\) \(P_I = \frac{48}{95}\) \(\rightarrow\) \(P_A = \frac{12}{95}\), \(P_B = \frac{24}{95}\), \(P_Q = \frac{23}{95}\)
Question 4 – Virtual Memory

We begin execution in the following state:

PC = 05000000
SP = BFFF0FFC

The code to be paged in contains starts with the following 3 instructions (we'll assume they're 4 bytes each):

MOV *(06000100), EAX
PUSH EAX
MOV EAX, *(06000300)

and I asked for the following events: (a) instruction attempts, (b) instruction completions, (c) page faults, (d) page allocation, (e) PTE updates.

1. Instruction attempt, 05000000 (MOV *,EAX)
2. page fault on instruction access
3. allocate page 5, set 00004[0A0] = 00005
4. allocate page 6 (code page), set 00005[000] = 00006, read-only
5. read code from disk to page 6
6. instruction attempt, 05000000 (MOV *(EAX)
7. instruction completes
8. instruction attempt 05000004 (PUSH EAX)
9. page fault, data write to BFFF0FF8 (stack)
10. allocate page 7, set 00004[2FF] = 00007
11. allocate page 8 (data page), fill with zeros, set 00007[3FF] = 00008, read/write
12. instruction attempt, PUSH EAX
13. instruction completion
14. instruction attempt, MOV EAX,*(...)
15. page fault – no write access
16. allocate page 00009, copy page 00002 to 00009, set 00003[000] = 00009, read/write
17. update 00001[000] from 00002 read-only to 00002 read/write.

Note that in step 6/7 we've faulted in the code page, and the data page is already mapped read-only. In step 16 we have to remember to copy 00002 to 00009 (it's not copy-on-write if you don't copy), and afterwards we restore the writability of page 06000xxx in address space 1, as it's no longer shared.

In practice step 17 might be deferred until a later fault, and probably requires a reference count so we know that page 00002 is not longer shared between multiple address spaces.