These are sample questions previous midterms of mine.

**Context Switching**

For this question you need to describe the behavior of a simple system with two processes, similar to one described in the first lecture or in Homework 1 question 4:

The operating system has two system calls:

```plaintext
putchar(c)
c = getchar()
```

Inside the operating system, each process has a task control block (tcb1 for process 1 and tcb2 for process 2) with the following fields:

- **next** – used to join the tasks in a ring, so that yield() can always move to the next task
- **sp** – saved stack pointer, as in HW 1 question 4
- **input** – single-character input buffer, holding either a value received from the serial port or <null>
- **index** – 1 for tcb1, 2 for tcb2

The OS has a variable `current` which points to the TCB of the currently running task.

When a character arrives at a serial port, one of the two interrupt handler functions is called (in the OS):

```plaintext
interrupt1:
tcbl.input = <character>

interrupt2:
tcb2.input = <character>
```

We assume that the programs read input fast enough that the character is received by the user process before the next input interrupt arrives.
The OS has the following additional functions:

\[
\text{yield:}
\]
\[
\begin{align*}
&\text{push registers} \\
&\text{current->sp = <stack_ptr>} \\
&\text{current = current->next} \\
&<\text{stack_ptr}> = \text{current->sp} \\
&\text{pop registers} \\
&\text{return}
\end{align*}
\]

\[
\text{getchar:}
\]
\[
\begin{align*}
&\text{while true:} \\
&\quad \text{if current->input == <null>} \\
&\quad \quad \text{yield()} \\
&\quad \text{else} \\
&\quad \quad \text{break} \\
&\quad \text{tmp = current->input} \\
&\quad \text{current->input = <null>} \\
&\quad \text{return tmp}
\end{align*}
\]

\[
\begin{align*}
&\text{putchar(c):} \\
&\quad \text{if current->index == 1} \\
&\quad \quad \text{send c to port 1} \\
&\quad \text{if current->index == 2} \\
&\quad \quad \text{send c to port 2}
\end{align*}
\]

The user processes are each running the following program:

\[
\text{char buf[80] (forget about buffer overflow for this program)}
\]
\[
\text{while true:} \\
\quad \text{getline(buf)} \\
\quad \text{putline(buf)}
\]

with the following library functions:

\[
\text{getline(ptr):}
\]
\[
\begin{align*}
&\text{while true:} \\
&\quad \text{c = getchar()} \\
&\quad \text{*ptr++ = c} \\
&\quad \text{if (c == '\n')} \\
&\quad \quad \text{break} \\
&\quad \text{*ptr = 0} \\
&\text{return}
\end{align*}
\]

\[
\text{putline(ptr):}
\]
\[
\begin{align*}
&\text{while *ptr != 0:} \\
&\quad \text{putchar(*ptr++)}
\end{align*}
\]

Finally, input arrives in the following sequence:

1. port 2: 'a'  
2. port 1: 'x'  
3. port 2: '"n'  
4. port 2: 'b'  
5. port 1: 'y'  
6. port 1: '"n'  

Simulate on paper the operation of the computer for this input. Assume that at the start of the simulation \textit{current} is pointing to tcb1, both processes are ready to call \textit{getline}, and both tcb1.input and tcb2.input are <null>.

For each of the 6 input events (2/a, 1/x, 2/"n, 2/b, 1/y, 1/"n) you will need to write out the program steps (in the main process function, the library, the OS, and the interrupt handler) which occur after that event, identifying (a) which process is running and (b) which function the line is in for each step.

You \textbf{don't} have to write out all the steps in \textit{yield()} - just write 'current = tcb1' or 'current = tcb2' to indicate whether it switched from process 2 to 1 or vice versa. You don't have to write the 'while true' lines, and you don't have to write out all the steps in \textit{putline()} - just tell me what string it's writing out.

Note that once all the work to handle an event is finished, you end up in a state where tcb1.input and tcb2.input are both <null>, both processes have called into \textit{getchar()}, and they are both going to “spin” yielding to each other. At this point write “spin” and go on to the next input event.
Here are the steps in response to the first event:

event: port 2, <a>
interrupt2: tcb2.input = <a>

proc. 1: (main) getline()
      1: (getline) getchar()
      1: (getchar) if current->input == <null>   - yes, it is
      1: (getchar) yield(): current = tcb2

proc. 2: (main) getline()
      2: (getline) getchar()
      2: (getchar) if current->input == <null>   - no, it's <a>
      2: (getchar) tmp = <a>, current->input = <null>
      2: (getline) if c == 'n' - no, it's <a>
      2: (getline) getchar()
      2: (getchar) if current->input == <null> - yes, it is
      2: (getchar) yield(): current = tcb1

This is one of the longer steps; you have 5 more to go.
Markov models - Dilbert

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.

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.

Synchronization – 3-way rendezvous

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.
Synchronization – 3-way rendezvous

For this question you will need to implement what I will call a 3-way synchronizer using a monitor. It has 3 methods: method1(), method2(), and method3(). It waits for threads to call each of the methods, with each passing an integer argument, and then all three methods return, with a return value equal to the average of the three inputs. A graphical example is shown below, with threads A, B, and C calling method1, method2, and method3 in that order.

You will need to give detailed pseudo-code, listing the monitor instance variables and describing the steps in each method in sufficient detail so that I can determine that it works correctly.

**Partial credit**: a monitor which works correctly if three threads A, B, and C call methods 1, 2, and 3 in any order.

**Full credit**: a monitor which works correctly if additional threads call method<i> while method</i> is still waiting to be called.

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.

Initial configuration:

![Initial Configuration Diagram]

**Context switching**

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

![Context Switching Diagram]

do_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

yield12():
0x300  PUSH 0x800 // &sp2
0x304  PUSH *(0x804) // sp1
0x308  CALL 0x400 // do_switch(sp1, &sp2)
0x30C  ADD 8,SP

0x500  POP EAX // EAX = stack.pop()
0x504  POP EBX // EBX = stack.pop()
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 pass 'arg' from the calling thread to the thread that gets switched to. Describe how this could be done.