CS-537: Midterm (Spring 2016)
The Timeline

Please Read All Questions Carefully!
There are eighteen (18) total numbered pages, with fourteen (14) questions.

Please put your FULL NAME (mandatory) on THIS page only.

Name: [MASTER]
Grading Page

<table>
<thead>
<tr>
<th>Points</th>
<th>Total Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>10</td>
</tr>
<tr>
<td>Q2</td>
<td>10</td>
</tr>
<tr>
<td>Q3</td>
<td>10</td>
</tr>
<tr>
<td>Q4</td>
<td>10</td>
</tr>
<tr>
<td>Q5</td>
<td>10</td>
</tr>
<tr>
<td>Q6</td>
<td>10</td>
</tr>
<tr>
<td>Q7</td>
<td>10</td>
</tr>
<tr>
<td>Q8</td>
<td>10</td>
</tr>
<tr>
<td>Q9</td>
<td>10</td>
</tr>
<tr>
<td>Q10</td>
<td>10</td>
</tr>
<tr>
<td>Q11</td>
<td>10</td>
</tr>
<tr>
<td>Q12</td>
<td>10</td>
</tr>
<tr>
<td>Q13</td>
<td>10</td>
</tr>
<tr>
<td>Q14</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>140</td>
</tr>
</tbody>
</table>
Directions

Welcome to the 537 Midterm! It shouldn’t be too hard, unless, of course, you haven’t prepared! Then it will be hard, unless you are a genius.

The theme of the exam is the timeline. Timelines show the behavior of things over time (raturally), and so in this exam we will mostly look at timelines to see what the system is doing, or create timelines to describe various behaviors.

Please read each question carefully.

Exam-taking strategy should be: easiest-problem-first. This scheduling discipline will ensure you finish as much of the exam as possible, and also builds your confidence. Don’t get stuck working on one hard problem.

Good luck!
1. Base And Bounds Blues

A system uses simple base/bounds to virtualize address spaces. In each of the traces below, your job is to fill in the missing values of either virtual addresses, physical addresses, base register, or bounds register. The bounds holds the size of the valid region of the address space. In some cases, you won’t be able to precisely know the values, so just put as precise of an answer as you can (i.e., perhaps a range, not a single number).

All numbers are in decimal format.

<table>
<thead>
<tr>
<th>Virtual Address --&gt; Physical Address</th>
<th>Base?</th>
<th>Bounds?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>2999</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address --&gt; Physical Address</th>
<th>Base?</th>
<th>Bounds?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>100</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>1999</td>
<td>2999</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>3000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address --&gt; Physical Address</th>
<th>Base?</th>
<th>Bounds?</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3400</td>
<td>3000</td>
</tr>
<tr>
<td>2000</td>
<td>5300</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>5301</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>6300</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address --&gt; Physical Address</th>
<th>Base?</th>
<th>Bounds?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6050</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>6150</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>6050</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>[fault]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address --&gt; Physical Address</th>
<th>Base?</th>
<th>Bounds?</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>9000</td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>[fault]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address --&gt; Physical Address</th>
<th>Base?</th>
<th>Bounds?</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>10001</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>3001</td>
<td></td>
</tr>
<tr>
<td>2001</td>
<td>3002</td>
<td></td>
</tr>
</tbody>
</table>
2. Deadlock Or Not Deadlock

Some of the following arrangement of threads, and the locks that they will grab, can lead to deadlock. Which ones?

Thread 1: will try to grab locks 1 and 2 in an arbitrary order.
Thread 2: will try to grab locks 1 and 2 in fixed order, 1 then 2.

Could this deadlock?

**YES**

Thread 1: will try to grab locks 1 and 2 in fixed order, 1 then 2.
Thread 2: will try to grab locks 1 and 2 in fixed order, 2 then 1.

Could this deadlock?

**YES**

Thread 1: will try to grab locks 1 and 2 in fixed order, 1 then 2.
Thread 2: will try to grab locks 1, 2 and 3 in fixed order, 1 then 2 then 3.

Could this deadlock?

(No cycle)

Thread 1: will try to grab locks 1 and 2 in arbitrary order.
Thread 2: will try to grab locks 2 and 3 in arbitrary order.
Thread 3: will try to grab locks 1 and 3 in arbitrary order.

Could this deadlock?

**YES**

Thread 1: will try to grab locks 1 and 2 in arbitrary order.
Thread 2: will try to grab locks 2 and 3 in arbitrary order.
Thread 3: will try to grab lock 1.

Could this deadlock?

**NO**

Thread 1: will try to grab locks 1 and 2 in fixed order, 1 then 2.
Thread 2: will try to grab locks 2 and 3 in fixed order, 2 then 3.
Thread 3: will try to grab locks 1 and 3 in fixed order, 1 then 3.

Could this deadlock?

**NO**

(ordered acquire)

Thread 1: will try to grab locks 1 and 2 in fixed order, 1 then 2.
Thread 2: will try to grab locks 2 and 3 in fixed order, 2 then 3.
Thread 3: will try to grab locks 1 and 2 in arbitrary order.

Could this deadlock?

**YES**
3. Fork Exec Wait Oh My

Remember fork(), exec(), and wait()? In this question, we trace when each of these system calls is CALLED, and when they RETURN. Thus, write down something like "fork called" if fork() has been called, and "fork returned" if it returns. A particular answer may have more than one action.

System call(s) called/returned: (if there are any)

- Process A is running; it starts to make a child process B (a nearly exact copy of itself)
  - FORK CALL

- Process A then continues running just after the OS has been told to create B
  - FORK RETURN

- Process B runs for the first time
  - FORK RETURN

- Process B now overlays its address space with a new program and starts running in its main()
  - EXEC CALL

- Process B runs for a while, then A runs for a while, then B, etc.

- Process A now ensures that it will be notified when B exits, blocking until that is the case.
  - WAIT CALL

B now creates C

C starts running

C tries to overlay its address space but fails to do so and decides to exit

- EXEC CALL, EXEC RETURN

- PROCESS B runs again, then creates D

B and D run, each doing some disk I/O

- FORK RETURN, FORK CALL

- FORK RETURN, FORK RETURN

B finally exits and A runs

- WAIT RETURN
4. Hardware Or Software Or User Program Or Give Up

This question is about the Limited Direct Execution protocol. Some of these steps are performed by the OS; some are handled by hardware (HW); some are in the user program itself (USER). Mark (by circling the correct answer), in the timeline below, which steps are taken by OS, HW, or USER program in this example of a process being created, running, issuing a system call, and exiting.

<table>
<thead>
<tr>
<th>Step</th>
<th>OS</th>
<th>HW</th>
<th>USER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create entry for process list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Allocate memory for program</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load program into memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Setup user stack with argv</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fill kernel stack with reg/PC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execute return-from-trap instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>restore regs from kernel stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>switch to user mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set PC to main()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Start running in main()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call a system call</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execute trap instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>save regs to kernel stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>switch to kernel mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set PC to OS trap handler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handle trap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do work of syscall</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>execute return-from-trap instruction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>restore regs from kernel stack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>switch to user mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set PC to instruction after earlier trap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call exit() system call</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5. Physical Memory Is Just A Cache

Here is a timeline of virtual memory references, given by the virtual page number:

\[ 0 \ 1 \ 4 \ 0 \ 1 \ 3 \ 0 \ 1 \ 4 \ 1 \]

Your job is simple: for each scenario below, determine whether the virtual page access will lead to a HIT ("H") as the page is FOUND in the memory of the system or a MISS ("m") as it is not (the page must be retrieved from the disk’s swap space).

All pages begin on disk; no pages are in memory at the start (and thus must be referenced to be brought into memory).

Policy FIFO, Cache Size 3

\[
\begin{array}{cccccccc}
0 & 1 & 4 & 0 & 1 & 3 & 0 & 1 \\
M & M & M & H & H & M & M & M \\
\end{array}
\]

Policy FIFO, Cache Size 5

\[
\begin{array}{cccccccc}
0 & 1 & 4 & 0 & 1 & 3 & 0 & 1 \\
M & M & M & M & H & H & H & H \\
\end{array}
\]

Policy LRU, Cache Size 3

\[
\begin{array}{cccccccc}
0 & 1 & 4 & 0 & 1 & 3 & 0 & 1 \\
M & M & M & H & H & M & M & H \\
\end{array}
\]

Policy LRU, Cache Size 1000

\[
\begin{array}{cccccccc}
0 & 1 & 4 & 0 & 1 & 3 & 0 & 1 \\
M & M & M & H & H & M & H & H \\
\end{array}
\]

[Notice: Same result because cache is big]
6. Producers And Consumers: We Need Both

Assume the following producer/consumer implementation for the famous bounded buffer problem.

```c
int buffer[max];

void *producer(void *arg) {
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);    // p1
        while (count == max)           // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        put(i);                        // p4
        Pthread_cond_signal(&full);    // p5
        Pthread_mutex_unlock(&mutex);  // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Pthread_mutex_lock(&mutex);    // c1
        while (count == 0)             // c2
            Pthread_cond_wait(&full, &mutex); // c3
        int tmp = get();              // c4
        Pthread_cond_signal(&empty);  // c5
        Pthread_mutex_unlock(&mutex);  // c6
        printf("%d\n", tmp);
    }
}
```

Assume further that the only way a thread stops running is when it explicitly blocks in either a condition variable or lock (in other words, no untimely interrupts switch from one thread to the other).

Also assume there are NO SPURIOUS WAKEUPS from wait().

In the following, show which lines of code run given the particular scenario. For example:

Thread Pa: plp2p4p5p6plp2p3
Thread Ca: c1c2c4c5c6c1c2

You can also show this more concisely if you like, e.g.:

Pa: 1,2,4,5,6,1,2,3
Ca: 1,2,4,5,6,1,2

Trace 1: 1 producer (Pa), 1 consumer (Ca), max=1. Producer Pa runs first. Stop when consumer has consumed one entry.

Pa: 1 2 4 5 6 1 2 3
Ca: 1 2 4

Trace 2: 1 producer (Pa), 1 consumer (Ca), max=3. Producer Pa runs first. Stop when consumer has consumed one entry.

Pa: 1 2 4 5 6 1 2 4 5 6 1 2 4 5 6 1 2 3
Ca: 1 2 4

3 times
void *producer(void *arg) {
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == max) // p1
            Pthread_mutex_lock(&mutex);
        Pthread_cond_wait(&empty, &mutex); // p2
        put(i); // p3
        Pthread_cond_signal(&fill); // p4
        Pthread_mutex_unlock(&mutex); // p5
    }
}

void *consumer(void *arg) {
    while (1) {
        Pthread_mutex_lock(&mutex);
        while (count == 0) // c1
            Pthread_mutex_lock(&mutex);
        Pthread_cond_wait(&fill, &mutex); // c2
        int tmp = get(); // c3
        Pthread_cond_signal(&empty); // c4
        Pthread_mutex_unlock(&mutex); // c5
    }
}

Trace 3: 1 producer (Pa), 1 consumer (Ca), max=1. Consumer Ca runs first. Stop when consumer has consumed one entry.
Pa: 123456123
Ca: 123

Trace 4: 1 producer (Pa), 2 consumers (Ca, Cb), max=1. Consumer Ca runs first, then Cb, then Pa. Stop when producer Pa has produced an entry.
Pa: 124
Ca: 123
Cb: 123

Trace 5: Now we change the "while" loops to "if" statements. Show a trace of behavior, using one producer and two consumers, where this leads to problems (assume max is set to whatever you like).
Pa: 123456123
Ca: 123
Cb: unit

Trace 6: Now we use "while" loops but only one condition variable, not two as above. Show a trace of behavior, using one producer and two consumers, where this leads to problems (assume max is set to whatever you like).
Pa: 123456123
Ca: unit
Cb: 123

7. Remember Scheduling

The following timelines show a set of jobs arriving, and then being executed on a processor. What is the TURNAROUND TIME for job A?

Assume A arrives at the beginning of the time unit where the "*" is on the timeline, A ends at the end of the time unit where "x" is and that each tick moves time forward 1 time unit.

Example

ABABA
* x

ABABABABAB
* x

BBABABABABAB
* x

AAAAABBBBB
* x

BBBBBBBBAAA
* x

BBBBBBBBBBB
* x

ABCBABCABCA
* x

BBBBBAAAA
* x

This means that A starts at time=0, runs right away, and then finishes at time=5. B runs from time=1-2 and time=3-4.

Turnaround Time (A)?

<table>
<thead>
<tr>
<th>Time</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
</tr>
</tbody>
</table>

Now do the same for RESPONSE TIME for A (note that the traces are not identical to those above):

Response Time (A)?

<table>
<thead>
<tr>
<th>Time</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
</tr>
</tbody>
</table>

Don't grade (same as)
8. Reverse Engineering The Page Table

In this problem, we consider address translation in a system with a simple linear page table (an array of page table entries, or PTEs).

Parameters:
- Virtual address space size is 32KB
- Page size is 4KB
- Physical memory size is 64KB

Here is a trace of virtual addresses and the physical addresses they translate to (or perhaps an invalid access):

VA 0x1063 → PA 0x2063
VA 0x67b4 → PA 0x67b4
VA 0x584a → PA 0x584a
VA 0x4dfe → Invalid
VA 0x388a → Invalid
VA 0xc6b → PA 0x2c6b
VA 0x50a9 → PA 0xe0a9
VA 0x0bc6 → Invalid
VA 0x2a9f → PA 0x9a9f
VA 0x742b → Invalid
VA 0xb5e → Invalid
VA 0x5597 → PA 0xe597

Can you reconstruct the page table entries from this? For each entry that you can construct, please do so; otherwise, mark down the entry as "UNKNOWN".

Format: Valid bit followed by Physical Frame Number (PFN)
[ 1 or 0 then PFN ]

<table>
<thead>
<tr>
<th>Page Table Entry 0:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table Entry 1:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table Entry 2:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table Entry 3:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table Entry 4:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table Entry 5:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0xE</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table Entry 6:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table Entry 7:</th>
<th>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9. Segmentation Is Fun Until The Stack Segment

Segmentation is another approach to supporting virtual memory. In this question, you will examine some timelines of virtual memory addresses and try to set the base and bounds registers, per segment, correctly so as to **NEVER GENERATE a SEGMENTATION FAULT**, and to make sure that the virtual addresses in the trace get translated to the proper physical address.

All other virtual addresses (not seen in the trace) should generate a SEGMENTATION FAULT.

Here we assume a simple segmentation approach that splits the virtual address space into two segments. The top bit of the virtual address determines which segment it is in.

Segment 0 acts like a code and heap segment; the heap grows towards higher addresses.

Segment 1 acts like a stack segment; it grows backwards towards lower addresses. For this segment, we follow convention that the book follows: the base register points to the physical address one past the last byte of the stack.

In both segments, the bounds (or limit) register just contains the "size" of the segment, i.e., the number of bytes valid.

**Trace 1:** Assume a 16-byte (4-bit) virtual address space (tiny!).
- Virtual address trace: 0,1,2,3,15,14,13 (all of these accesses should be valid)
- Virtual address 1 translates to physical address 101
- Virtual address 13 translates to physical address 998

<table>
<thead>
<tr>
<th>Segment 0 Base?</th>
<th>100</th>
<th>Segment 1 Base?</th>
<th>1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment 0 Limit?</td>
<td>4</td>
<td>Segment 1 Limit?</td>
<td>3</td>
</tr>
</tbody>
</table>

**Trace 2:** Assume a 64-byte (6-bit) virtual address space.
- Virtual address trace: 0,1,63 (all of these accesses should be valid)
- Virtual address 1 translates to physical address 1001
- Virtual address 63 translates to physical address 899

<table>
<thead>
<tr>
<th>Segment 0 Base?</th>
<th>1000</th>
<th>Segment 1 Base?</th>
<th>900</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment 0 Limit?</td>
<td>2</td>
<td>Segment 1 Limit?</td>
<td>1</td>
</tr>
</tbody>
</table>

**Trace 3:** Assume a 8-byte (3-bit) virtual address space.
- Virtual address trace: 0,1,2,3 (all of these accesses should be valid)
- Virtual address 3 translates to physical address 100

<table>
<thead>
<tr>
<th>Segment 0 Base?</th>
<th>97</th>
<th>Segment 1 Base?</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment 0 Limit?</td>
<td>4</td>
<td>Segment 1 Limit?</td>
<td>0</td>
</tr>
</tbody>
</table>
10. Spin Locks For The Win

Assume we are using a spinlock to protect a critical section. In the timelines below, "c" means a thread is doing some computation; "S" means a thread is spinning on a lock, waiting for it to become available; "A" means a thread acquired a lock; "R" means a thread has released the lock.

Describe, in words, what happened in each of the following timelines. In some cases, the timelines shouldn't be possible. If so, say so, and describe why. In all cases, there is ONLY ONE LOCK that is being used.

Thread 1: cccccAcccccRcccccc Thread 2: ccccccAccccccRcccccc

What happened?

T₁ acquired, released lock; T₂ did same

Thread 1: cccccAccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
11. The Dreaded MLFQ Question

The MLFQ (multi-level feedback queue) is a scheduling discipline. It consists of a number of rules. First, circle the rules that are actually part of the final MLFQ policy:

1. If Priority(A) > Priority(B), A runs (B doesn’t).
2. If Priority(A) < Priority(B), A runs (B doesn’t).
3. If Priority(A) = Priority(B), A & B run in round-robin fashion.
4. If Priority(A) = Priority(B), A runs to completion, then B.
5. When a job enters the system, it is placed at the highest priority (the topmost queue).
6. When a job enters the system, it is placed in any queue.
7. When a job enters the system, it is placed in the lowest queue.
8. Once a job uses up its time slice at a given level, its priority is reduced.
9. Once a job uses up its time slice at a given level, it moves to the end of the round-robin queue.
10. Once a job uses up its time slice at a given level, it exits.
11. After some time period, move each job up to a higher priority queue.
12. After some time period, move each job down to a lower priority queue.
13. After some time period, move all the jobs in the system to the top-most (highest-priority) queue.

Now, write down the rule (or rules) that come into play in each of the following example traces of MLFQ behavior (use the numbers from above). Note that * marks when A arrives, if the information is relevant.

Rule(s)?

Q3: A
Q2: AA
Q1: AAAAAAAAAA ...

Q3: A
Q2: AA
Q1: BBBBBBB

Q3: A
Q2: AA
Q1: BBBBBBB AAAAAAAA ...

Q3: A
Q2: AA
Q1: BBBBBBB AAAAAAAA ...

Q3: A
Q2: AA
Q1: BBBBBBB AAAAAAAA ...

Q3: AB
Q2: AABB
Q1: AABBAAABB AABBBB...
12. The Even More Dreaded Multi-Level Page Table

Assume you have a 15-bit virtual address, with page size = 32 bytes. Assume further a two-level page table, with a page directory which points to pieces of the page table. Each page directory entry is 1 byte, and consists of a valid bit and PFN of the page of the page table. Each page table entry is similar: a valid bit followed by the PFN of the physical page where the desired data resides.

The page directory resides in physical page 18.

The following physical page contents are made available to you:

- Page 10: 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
- Page 18: 7f c0 ea f9 ed 8b db ba d9 c1 84 8a b3 7f da eb 9a 85 ab 87 e5 97 b1 df 86 ec e7 ad f2 b9 d5 f8
- Page 30: 1b 03 11 1e 12 16 18 0f 08 10 0a 1a 0b 0e 17 19 1b 14 07 1a [c] 16 17 0f 0f 12 04 14 1a 05
- Page 57: a1 7f 7f 7f 7f 7f 7f 7f 9e 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
- Page 90: 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f 7f
- Page 98: 16 0d 18 10 02 0e 01 1c 1d 0a 09 17 06 05 05 0a 13 1d 06 1d 11 1b 19 04 14 03 00 3c 17 11 05 1a
- Page 126: 16 0e 14 07 07 01 0c 11 03 05 0c 00 19 05 1c 11 09 02 13 01 0a 1e 19 16 13 17 1b 03 1b 1e 12

In translating virtual address 0x3a3a, which physical pages are accessed?

18, 90, 98

What is the final data value returned?

0x00

In translating virtual address 0x74f6, which physical pages are accessed?

18, 57, 30

What is the final data value returned?

0x1c
13. Too Many Forking Questions

We have a system with three processes (A, B, C) and a single CPU. Processes can be in one of three states: Running, Ready, Blocked. If the process does not exist (yet), or has exited, just put a "---" down or leave the entry blank.

Below is a timeline of process behavior. Fill in the states of each process in the diagram:

Process A is loaded into memory and starts executing in main().

Process A calls fork() and creates Process B (but A, the parent, keeps running).

Process A issues a request to the disk; B starts executing at the return from fork().

B calls fork(), creating Process C; B keeps running.

B’s timeslice expires; C runs.

A’s I/O completes (but there are no other changes)

C waits for user input. A runs.
14. You Put the T In TLB

The following question traces TLB behavior over time. Each question will give you a few assumptions; you should then produce a series of hits ("H") and misses ("m"). The string "mHmH" would mean "TLB miss" followed by a "TLB hit" followed by a "miss" followed by a "hit".

In all cases, ignore instruction references (i.e., do not worry about their effects on the TLB).

Also, always assume the array (discussed below) is PAGE ALIGNED.

Assume you have a 1-entry TLB. Assume you access contiguous 4-byte integers in a large array, starting at index 0 and going to the max size. Assume the page size is 32-bytes. What is the hit/miss pattern for that access pattern?

Pattern:

\[
\text{[m HHH HHHH] repeated (until done)}
\]

Assume you have a 2-entry TLB with LRU replacement of TLB entries. Assume you access contiguous 4-byte integers in a large array, again starting at 0. Assume the page size is 32-bytes. What is the hit/miss pattern for that access pattern?

Pattern:

\[
\text{[m HH HHH HHH] repeated (until done) (same as above)}
\]

Assume you have a 1-entry TLB. Assume you access *every other* 4-byte integer in a large contiguous array, starting at index=0, then index=2, etc. Assume the page size is 32-bytes. What is the hit/miss pattern for that access pattern?

Pattern:

\[
\text{[m HHH] repeated (until done)}
\]

Assume you have a 16-entry TLB with FIFO replacement. Assume you repeatedly access all 4-byte integers in a small contiguous array of 24 integers, in a loop. Assume the page size is 32-bytes. What is the hit/miss pattern for that access pattern, for the *fourth* run through the loop?

Pattern:

\[
\text{[H] repeated 24 times (all hits)}
\]

\[
\text{3 pages}
\]