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 (naturally), 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 hold 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>
</tr>
</thead>
<tbody>
<tr>
<td>0  1000</td>
</tr>
<tr>
<td>100  1100</td>
</tr>
<tr>
<td>1999  2999</td>
</tr>
<tr>
<td>2000  [fault]</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>Virtual Address -&gt; Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>____ ?  0900</td>
</tr>
<tr>
<td>____ ?  1100</td>
</tr>
<tr>
<td>____ ?  3000</td>
</tr>
<tr>
<td>____ ?  [fault]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Virtual Address -&gt; Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>9000  10001</td>
</tr>
<tr>
<td>100  1101</td>
</tr>
<tr>
<td>2000  3001</td>
</tr>
<tr>
<td>2001  3002</td>
</tr>
</tbody>
</table>
2. 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)

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

Process B runs for the first time

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

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.

B now creates C

C starts running

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

Process B runs again, then creates D

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

B finally exits and A runs
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>
4. 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

0 1 4 0 1 3 0 1 4 1

Policy FIFO, Cache Size 5

0 1 4 0 1 3 0 1 4 1

Policy LRU, Cache Size 3

0 1 4 0 1 3 0 1 4 1

Policy LRU, Cache Size 1000

0 1 4 0 1 3 0 1 4 1
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(&fill); // p5
        Pthread_mutex_unlock(&mutex); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Pthread_mutex_lock(&mutex); // c1
        while (count == 0) // c2
            Pthread_cond_wait(&fill, &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: p1p2p4p5p6p1p2p3
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:
Ca:
```

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

```
Pa:
Ca:
```
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_mutex_unlock(&mutex); // p6
    }
}

void *consumer(void *arg) {
    while (1) {
        Pthread_mutex_lock(&mutex); // c1
        while (count == 0) // c2
            Pthread_cond_wait(&fill, &mutex); // c3
        int tmp = get(); // c4
        Pthread_mutex_unlock(&mutex); // c6
    }
}

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

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:
Ca:
Cb:

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:
Ca:
Cb:

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:
Ca:
Cb:
6. 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" and that each tick moves time forward 1 time unit.

Example

```
ABABA
*  x

ABABABABABAB
*  x

BBABABABABAB
*  x

AAAAABBBBB
*  x

BBBBBBBBAAAAA
*  x

ABCBABCABCA
*  x

BBBBBBBBAAAAA
*  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) ?

Now do the same for RESPONSE TIME for A:

```
ABABABABAB
*  x

BBABABABABAB
*  x

AAAAABBBBB
*  x

BBBBBBBBAAAAA
*  x

ABCBABCABCA
*  x

BBBBBBBBAAAAA
*  x
```
7. 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 0xe84a
VA 0x4dfe --> Invalid
VA 0x388a --> Invalid
VA 0x1c6b --> PA 0x2c6b
VA 0x50a9 --> PA 0xe0a9
VA 0x0bc6 --> Invalid
VA 0x2a9f --> PA 0x9a9f
VA 0x742b --> Invalid
VA 0x4b5e --> 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>Valid</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Table Entry 0:</td>
<td></td>
</tr>
<tr>
<td>Page Table Entry 1:</td>
<td></td>
</tr>
<tr>
<td>Page Table Entry 2:</td>
<td></td>
</tr>
<tr>
<td>Page Table Entry 3:</td>
<td></td>
</tr>
<tr>
<td>Page Table Entry 4:</td>
<td></td>
</tr>
<tr>
<td>Page Table Entry 5:</td>
<td></td>
</tr>
<tr>
<td>Page Table Entry 6:</td>
<td></td>
</tr>
<tr>
<td>Page Table Entry 7:</td>
<td></td>
</tr>
</tbody>
</table>
8. 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: ccccAccccRccccc
Thread 2: ccccccAccccRccccc
What happened?

Thread 1: ccccAccccccc cccccRccccc
Thread 2: cccccSSSSSSSSSSSSSSSSSSSSS
What happened?

Thread 1: ccccAcc cccccR
Thread 2: ccccccAccccR
What happened?

Thread 1: ccccAcc
Thread 2: cccSSSSSSSSSSSSSSS ... (forever)
What happened?

Thread 1: AR AR AR AR ....
Thread 2: AR AR AR AR
What happened?

Thread 1: cccSSSSSSSS SSSSSSSSS ... (forever)
Thread 2: ccccccSSSSSS SSSSSSSSS ... (forever)
What happened?
9. 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: AAAAAAAAAAAA ...

*  
Q3: A
Q2: AA
Q1: BBBBB

*  
Q3: A
Q2: AA
Q1: BBBBB BBAAAAA ...

Q3:  
Q2:  
Q1: BBABBBAAABBBB

Q3: AB  
Q2: ABB  
Q1: AAABBBBAAABBBB AAAABBBB...
10. Too Much Forking

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:

<table>
<thead>
<tr>
<th>Event Description</th>
<th>A</th>
<th>State?</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process A is loaded into memory and starts executing in main().</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process A calls fork() and creates Process B (but A, the parent, keeps running)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process A issues a request to the disk; B starts executing at the return from fork().</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B calls fork(), creating Process C; B keeps running.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B’s timeslice expires; C runs.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A’s I/O completes (but there are no other changes)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C waits for user input. A runs.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
11. 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:

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:

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:

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:
12. Reverse Engineering Memory Management

The above trace of accesses to physical memory was generated by this user-space code that loops over an array of 4-byte integers:

```c
for(i=0; i<16; i++) {
    sum += array[i*64];
}
```

Only data reads are shown in the trace (i.e., memory accesses caused by instruction fetch are not shown). Assume "i" and "sum" are kept in registers, so only the accesses to array trigger memory accesses. Answer the following (provide explanations):

(a) what is the page size?

(b) Which is consistent with the accesses: a 4-level page table? or segmented paging?

(c) Is a TLB in use?
13. Lookout for Locality!

Assume all pages are initially swapped out to disk, and the process may only have 3 pages resident in memory at a time. The OS uses the LRU replacement policy with no prefetch. Consider the following timeline of accesses:

(a) is the above timeline, fill the circles that will be hits to memory

(b) what is the hit rate?

(c) assuming a disk read takes 10 ms, and a memory read takes 0 ns (yes, that’s zero time), what is the AMAT (average memory access time) for this workload?

(d) note the exact same address is never accessed twice, yet there are still some hits. What type of locality makes this possible?

(e) how would changing the page size affect the hit rate in this workload?
In order to debug your spinlock performance, you add some printf calls to your spinlock:

```c
void lock(lock_t *lock) {
    printf("enter\n");
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
    printf("return\n");
}
```

```c
void unlock(lock_t *lock) {
    printf("unlock\n");
    lock->flag = 0;
}
```

You also modify your kernel (which always uses round-robin scheduling) to print "CS" on every context switch. You observe the following timeline, annotated with printf output:

(a) suppose you add a yield() call inside your spin loop. Redraw the timeline (again with printf annotations):

(b) suppose you start using kernel-supported queue locks instead of the spinlock. Redraw the timeline (again with printf annotations):