1. User programs may execute a special instruction to enter kernel mode.
2. The kernel may read or write data in user space (the user program's virtual address space).
3. Your heap manager library invokes the kernel to allocate virtual memory.
4. A buggy heap library might cause the process to crash while executing user code outside of the library.
5. A buggy heap library might cause the operating system to crash.
6. A buggy kernel might cause the operating system to crash.
7. A CPU interrupt may occur when the CPU core is in kernel mode.
8. A CPU interrupt may occur when the CPU core is in user mode.
9. A CPU core in user mode may execute a special instruction to disable interrupts.
10. A CPU core in kernel mode may execute a special instruction to disable interrupts.
11. A CPU interrupt may cause the kernel to initiate a thread context switch.
12. A load or store to an allocated (with malloc) heap block may raise a CPU fault.
13. A load or store to a released (with free) heap block may raise a CPU fault.
14. A classic (single-threaded) Unix process may block when the CPU is in kernel mode.
15. A classic (single-threaded) Unix process may block when the CPU is in user mode.
P1. User and kernel (30 points)

1. User programs may execute a special instruction to enter kernel mode. It is the trap instruction (callsys, syscall). 60%

2. The kernel may read or write data in user space (the user program's virtual address space). It is copyin/copyout, as for the read/write system calls. That is one reason why it is useful to run in kernel mode within the process VAS. 85%

3. Your heap manager library invokes the kernel to allocate virtual memory. With the mmap or sbrk system call.

4. A buggy heap library might cause the process to crash while executing user code outside of the library. For example, the heap library might doubly allocate a block and cause the user program to overwrite its own heap blocks.

5. A buggy heap library might cause the operating system to crash. “The kernel must be bulletproof.” 84%

6. A buggy kernel might cause the operating system to crash. E.g., null pointer in the kernel causes “blue screen of death”.

7. A CPU interrupt may occur when the CPU core is in kernel mode. Kernel runs with interrupts enabled (except when it disables them for short sections), and interrupts may be nested. 55%

8. A CPU interrupt may occur when the CPU core is in user mode. E.g., timer interrupts drive involuntary yields (preempts) for timeslicing.

9. A CPU core in user mode may execute a special instruction to disable interrupts. It can try but disabling interrupts is a privileged op.

10. A CPU core in kernel mode may execute a special instruction to disable interrupts. This is what we mean by disable_interrupts in p1.

11. A CPU interrupt may cause the kernel to initiate a thread context switch. E.g., timeslicing with timer interrupts.

12. A load or store to an allocated (with malloc) heap block may raise a CPU fault. Tricky: valid, but may raise a page fault. 49%

13. A load or store to a released (with free) heap block may raise a CPU fault. Tricky: valid, but may raise a page fault. 78% got this, but the 49% on the previous question suggests that many got it right for the wrong reasons.

14. A classic (single-threaded) Unix process may block when the CPU is in kernel mode. Processes block while executing syscall code (traps) or fault handler code (e.g., page fault). 54%

15. A classic (single-threaded) Unix process may block when the CPU is in user mode. Must enter kernel to block: context switch to another process is not possible in user mode. 19%
P2. Hex mess (30 points)
Consider a simple page table for a simple model machine. This machine has a byte-addressable memory with 256 256-byte pages/frames, so an address (virtual or physical) fits in one 16-bit word. A page table entry (PTE) is also one 16-bit word, consisting of (in order) a zero bit (unused for this example), a valid/present bit (set iff the PTE has a valid translation), a write-enable bit, an execute-enable bit (these permission bits are set iff the corresponding permission is enabled) and a 12-bit page frame number (PFN).

```
| PTE: | 0 | V | W | X | PFN (12 bits) |
```

Assume the OS kernel initializes a process page table sparsely with contents as shown to the right for the text, global, and stack regions: all unspecified regions are undefined and their PTEs are zero.

Consider the five instructions listed, each with its virtual address operand. What does the machine do for each instruction? Enter into the corresponding box an E if the instruction executes, an F if it gives a page fault, a P if it gives a protection fault, or an S for a segmentation fault. Only one outcome is possible for each instruction. Write any additional assumptions outside of the boxes:

Finally, when you are done, enter all five outcomes in one string with no punctuation in the box below:

```
F EEPP
```

About two-thirds of the class got this right or missed one (24+12) or maybe two (11). Those of you who are guessing: these are easy concepts that are the key to understanding virtual memory. See the diagram on the next page.
P2. Hex mess (30 points)

1. **F**: missing page: fault
2. **E**: all good
3. **E**: all good
4. **P**: execute is disabled (stack)
5. **P**: write is disabled (text)

### Step 1. Leftmost two hex digits of VA give the VPN

- **F**
- **E**
- **E**
- **P**
- **P**

### Step 2. Index the page map by VPN to get PTE.

<table>
<thead>
<tr>
<th>VPN (8 bits)</th>
<th>Offset (8 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x000</td>
</tr>
<tr>
<td>0x5</td>
<td>0x108</td>
</tr>
<tr>
<td>0x1</td>
<td>0x000</td>
</tr>
<tr>
<td>0x5</td>
<td>0x080</td>
</tr>
<tr>
<td>0x5</td>
<td>0x091</td>
</tr>
<tr>
<td>0x40</td>
<td>0x26a1</td>
</tr>
<tr>
<td>0x2</td>
<td>0x02b</td>
</tr>
<tr>
<td>0x6</td>
<td>0xa2c</td>
</tr>
<tr>
<td>0x6</td>
<td>0x032</td>
</tr>
<tr>
<td>0xf0</td>
<td>0x2000</td>
</tr>
<tr>
<td>0xf1</td>
<td>0x2000</td>
</tr>
<tr>
<td>0xf2</td>
<td>0x6120</td>
</tr>
<tr>
<td>0xf3</td>
<td>0x6119</td>
</tr>
</tbody>
</table>

### Step 3. Get bits (valid, protection) from leftmost hex digit of PTE, and decode.

<table>
<thead>
<tr>
<th>Page table</th>
<th>PTE bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 0x000</td>
<td>0: 0x0 Miss</td>
</tr>
<tr>
<td>1: 0x5108</td>
<td>1: 0x5 VX</td>
</tr>
<tr>
<td>2: 0x1000</td>
<td>2: 0x1 Miss</td>
</tr>
<tr>
<td>3: 0x5080</td>
<td>3: 0x5 VX</td>
</tr>
<tr>
<td>4: 0x5091</td>
<td>4: 0x5 VX</td>
</tr>
<tr>
<td>0x40</td>
<td>0x2 Miss</td>
</tr>
<tr>
<td>41</td>
<td>0x2 Miss</td>
</tr>
<tr>
<td>42</td>
<td>0x6 VW</td>
</tr>
<tr>
<td>43</td>
<td>0x6 VW</td>
</tr>
<tr>
<td>0xf0</td>
<td>0x2 Miss</td>
</tr>
<tr>
<td>f1</td>
<td>0x2 Miss</td>
</tr>
<tr>
<td>f2</td>
<td>0x6 VW</td>
</tr>
<tr>
<td>f3</td>
<td>0x6 VW</td>
</tr>
</tbody>
</table>

### Step 4. Determine from PTE bits if translation is valid (present) or missing.

### Step 5. If valid, check PTE protection bits against requested mode of access.
P3. Spell it out (30 points)
The thread API for p1 consists of functions thread_XXX(), where each XXX begins with one of the following letters: BUSYCLEW. To get the BUSYCLEW acronym, we ignore libinit and suppose that a thread calls thread_exit (E) when it returns from its start function. List a string of capital letters in BUSYCLEW order for all thread operations that do (or might do) each of the following:

1. Block sometimes
2. Block every time
3. Place a thread on the ready queue
4. Place multiple threads on the ready queue
5. Initiate a context switch
6. Grant ownership of a lock
7. Release ownership of a lock
8. Modify a lock queue
9. Complete a deadlock
10. Terminate (exit) the process

P4. Three-thread monte (30 points)
Suppose that three threads with IDs {0, 1, 2} all invoke each of the following numbered pseudocode program snippets. What do the resulting schedules print? Assume that the threads are created in ID order for each snippet, that “print(id)” prints the ID of the current thread, and that the thread primitives behave exactly as in p1, with the mandated deterministic FIFO behavior (i.e., no preemption). List all possible outputs for each schedule in the corresponding box on the left, on separate lines, as strings of digits with no punctuation.
P3. Spell it out (30 points)
This question tested whether you were conversant with the basic p1 thread primitives. I got about a dozen or so test-takers who did not follow directions: letters out of order, punctuation, full names. Since I had to decode their 100+ answers manually, I shaved some points here and there where convenient and was generally less forgiving.

About half the questions had 75% of the points awarded. For 6 grant ownership of a lock, lots of people forgot wait or unlock (handoff locks, a p1t detail). A few other trouble spots: 3 place a thread on the ready queue, 5 initiate a context switch, 9 complete a deadlock, 10 exit the process. These were all in the 50%-60% range. Any switch out of the running state (block, exit, yield:YLEW) initiates a context switch, and any of these could terminate the process if there are no ready threads to switch to—except yield, which always leaves a ready thread to switch to. To complete a deadlock, a thread must block (LW). For operations that place a thread on the ready queue, about 40% of the class missed one or two of the five (BUSYC+W). Most trickily, W does an unlock, which can add a thread to the ready queue and confer lock ownership (via handoff). The + in 3.3 and 3.6 means that I did not deduct points for missing the W. I imagined people thought W "calls" U and L, and so listing U and L is good enough to cover the W case.

P4. Three-thread Monte (30 points)
People complained about this, but answers earned 80-85% of the points for the first two. Cycle was more involved so scores were a little lower. People got confused about the FIFO behavior of CVs, which causes threads to wake in the order they wait, and some got confused about yielding while holding a lock. Only 20% of the Cycle answers were fully correct. About 10% of all answers had multiple schedules, which lost some points because the schedules are all deterministic under the assumptions.

**Cycle:**
- T0 gets lock, waits on 1,1
- T1 gets lock, wakes T0, waits on 1,1
- T2 gets lock, wakes T1, waits on 1,1, ready q: T1 T0
- T0 runs and prints (0)
- T0 yields with lock, ready q: T0 T1
- T1 runs
- T1 slams into lock, T0 runs, ready q: empty
- T0 loops, wakes T2, waits on 1,1 (waking T1 with lock)
- T2 runs, slams into lock, ready q: T1
- T1 runs with lock, prints (01) and yields to nobody
- T1 loops, wakes T0, waits on 1,1 (waking T2 with lock), ready q: T2 T0
- T0 runs, slams into lock
- T2 runs, prints (012) and yields to nobody, ready q empty
- T2 loops, wakes T1, waits on 1,1 (waking T0 with lock), ready q: T0 T1
- T1 runs, slams into lock
- T0 runs, prints (0120) and yields to nobody
- T0 unlocks (waking T1 with lock), exits, printing 00 (012000)
- T1 runs, prints (0120001) and yields to nobody
- T1 unlocks, exits printing 11 (012000111)
P5. Thread multi-pool (80 points).
Consider a server S that processes incoming requests. Each request R is one of three kinds (or types, or classes: 1, 2, 3). The type of each request R is visible to the code as R.class. S has a fixed set of servicer threads such that each servicer T is associated with exactly one class C, and processes requests only of its class C. S has a receiver thread that receives incoming requests and queues them for servicing. This problem asks you to write code to synchronize the receiver and servicers.

a) (20+20 = 40 points) The receiver calls procedure incoming() for each incoming request. Each servicer calls getNextRequest(class c) to accept an incoming request. Write two versions of these procedures, a FAST version and a CLEAN version. In FAST, your goal is to minimize the number of unnecessary context switches. CLEAN must conform to the Java restriction that each monitor has a single condition (CV). As always: "any kind of pseudocode is fine as long as its meaning is clear."

The code is identical for the two versions, except that CLEAN uses a single CV and a broadcast. Nothing else changes, since the servicers loop before leaping. The FAST version is faster because it allows dispatching directly to a servicer thread that can handle the request. The code is independent of the number of servicers: it follows the ThreadPool pattern, but with per-class queues rather than a single queue. Unlike Soda Machine or Pipes, a ThreadPool has no reason for incoming() to wait: typically a server or other network device drops or rejects incoming traffic when a queue is full.
P5. Thread multi-pool continued. Answer these questions about the problem and your solutions.

b) Traces (10+10 = 20 points). Suppose deterministic p1t threads with no preemption, and with one core and one servicer for each class. Consider the thread schedule to handle a sequence of three incoming requests with classes 1, 2, 3. Assume that the receiver produces the next request only after all servicer threads are idle, and then yields. List the thread XXX operations called in the schedule, in order, as a string of capital letters in BUSYCLEW order (as in P3). Each call of the thread library from the user program is represented by one letter. For example, each schedule starts with CCCC to create the receiver and servicer threads.

a) Trace FAST: CCCC LW LW LW LSUY ULW LSUY ULW LSUY ULW

b) How many context switches? Enter a number: 9

c) Trace CLEAN: CCCC LW LW LW LBUY ULWWW LBUY WULWW LBUY WWULW

d) How many context switches? Enter a number: 15

c) Keeping busy (20 points). Suppose server S has four CPU cores. How many servicers for S to to be “live”? “Live” means: S can process any incoming request R if S has an idle core to process R, independent of the class of the request. Enter a number in each box.

a) What is the smallest number of servicers (total) that could ensure liveness, presuming that they do not block while processing a request? 12

b) What is the smallest number of servicers (total) that could ensure liveness, presuming that they block half the time (e.g., for I/O) while processing a request? 24
P5. Thread multi-pool continued. Answer these questions about the problem and your solutions.

Trace FAST:

```
CCCC LW LW LW LSUY ULW LSUY ULW LSUY ULW
```

How many context switches? Enter a number: 9

The idea here was that Gradescope could recognize correct answers automatically by string-matching. Correct answers would assure full credit on P5a, reducing manual grading overhead. It did not happen that way. I awarded about 70% of the points on this part, but only a few answers were exactly what I was looking for.

The trace must have the prefix CCCC LWLWLW to create the threads and quiesce the servicers. Then the receiver dispatches each incoming() request with a LSUY sequence, and a servicer that picks up the request unlocks with a U before returning the request for servicing, and then awaits the next request with LW. I took points if major pieces were missing, e.g., no waits, no locking, no signals, locks without unlocks. For the context switches: one for each LW to quiesce, then two for each request, plus or minus one: see the spaces in the trace above.

CLEAN trace is the same, but signal turns to broadcast (LSUY→LBUY) and all the servicers wake up, so it takes WW for the spurious ones to go back to sleep. So the broadcasts for requests 1,2,3 generate ULWWW, WULWW, WWULW. That results in two extra context switches per request.

Any reasonable answer was accepted, and partial credit here was generous. Unfortunately, blank answers (8%) could not benefit from partial credit. You should always give your best shot, even if it is only a partial answer. Otherwise your score cannot reflect what you know and the purpose of the exam is defeated.

Trace CLEAN:

```
CCCC LW LW LW LBUY ULWWW LBUY WULWW LBUY WWULW
```

How many context switches? Enter a number: 15

c) Keeping busy (20 points).

The trick here is that if the incoming requests are all of the same type, then you need one servicer per class per core to be sure you always have a thread to process the next request if there is a core idle. If the threads are blocked half the time then you need more to ensure that you have (in expectation) one non-blocked thread per class if there is an idle core. Doubling will do it, but I accepted a range of values to increase the thread count by 50% to 2x or a little more. About a third of the class increased the thread count suitably for blocking but missed the workload dependency. That was worth partial credit.