P1. User and kernel. (30 points) These are true/false questions: please answer T or F for each, in the box to the left.

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.
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).

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:
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.
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."

**FAST**

```java
incoming(Request r) {
}
Request getNextRequest(Class c) {
}
```

**CLEAN**

```java
incoming(Request r) {
}
Request getNextRequest(Class c) {
}
```
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: 

b) How many context switches? Enter a number: 

c) Trace CLEAN: 

d) How many context switches? Enter a number: 

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? 

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?