Topics Needing Clarification

In grading Assignment 1, I realized that there are several topics that are not clearly understood by a majority of those who submitted the assignment. The purpose of this short document is to clarify those areas.

Timers. The timer referred to in the question is not the same thing as the hardware or system clock. A *timer* is associated with the process currently running on the CPU, and it is used by the operating system to implement time-slicing of the CPU. It is controlled by the operating system with hardware support. When a process is scheduled onto the CPU, the operating system sets the timer equal to a specified number of *ticks*. The length of a tick depends on the operating system and hardware. On a fast, modern processor such as a Pentium, a tick might be a millisecond. The timer is decremented with each tick until it becomes zero, at which point a timer interrupt occurs and the process is removed from the CPU. The decrementing may be done in hardware or software. On older systems each tick caused an interrupt to decrement the timer in software. Modern processors have special hardware timers that decrement themselves. A user process should not be able to modify the timer because if it could, it could prevent the operating system from ever running.

Zeroing memory. Every process needs the ability to clear memory, or fill a range of addresses with zeros. On UNIX, the `bzero()` and `memset()` library functions do exactly this. The questions is, should this instruction be privileged? The answer is that it depends on what other protections exist in the CPU. Most modern processors have support for memory protection in the form of either segmentation or base-limit registers, usually in a separate memory management unit or MMU. In modern CPUs, when an address is generated, it is a logical address, not a physical one, and the corresponding physical address is generated by the MMU, which does the required checking. Discussion of how this is done has to wait until we cover memory management. If a processor does have an MMU, then the instruction to clear memory does not need to be privileged, because if the range of address will be validated in the MMU. If it does not, then the kernel would have to validate the range before zeroing the memory, so it would have to be privileged.

Traps. A *trap* is a specific type of exception. An *exception* is an interrupt generated by the processor as a result of executing an instruction. The exception may be due to an erroneous or anomalous condition, such as dividing by zero or trying to access a part of memory outside of the process's legal range. Such exceptions are known as *faults*. Traps are sometimes purposely executed to force the kernel to run. Traps can be used as a means of implementing *system calls*. Traps are the principal method of implementing debuggers as well. When a break-point is set in a program, a trap is inserted into the code. Traps must be executable in user mode, otherwise a user program could never relinquish the CPU voluntarily to let the kernel run.

Changing the mode bit. If a user process were able to change the mode from user to privileged, then there would be no distinction between user mode and privileged mode, since all instructions could be executed by user processes. Obviously then, this instruction must be privileged and can only be executed by a process running with kernel privileges, i.e., it can only be done by a process that has the privilege to do it. This sounds like a paradox: only privileged processes can execute the instruction that allows them to acquire the privilege to execute privileged instructions. Obviously, there has to be a catch here. Roughly speaking, two values are required to implement dual mode, a Current Privilege Level, associated with the running process, and a Code Privilege Level, associated with the code to be executed in privileged mode. Suppose 0 and 1 are the two values each can have and that 1 means kernel privilege and 0 means user privilege. When a user process needs the kernel to do something for it, it issues a call that causes a trap that causes kernel code to run briefly in user mode. The kernel code has Code Privilege 1, which is greater than 0, so it is allowed to change the mode, setting Current Privilege Level to 1. When it finishes, it sets it back to 0 before letting a

user process run. It is actually much more complex than this, and the terms I use are my own, but this is the basic idea.

On dual-mode in the 0x86 series. The point of this question was to get you to investigate exactly how dual mode works in the 0x86 series of processors. It is not as simple as it was in the Motorola series. Intel had to make sure that what they did in the 386 was backwards compatible with all of those DOS programs that ran amok on the unprotected earlier CPUs. So they created a *real mode*, a *protected mode*, and a *virtual real mode*. Real mode is the mode of the earlier CPUS like the 8088, completely unprotected. Legacy DOS programs could only run in real mode. Protected mode provided four different levels of privilege, which they called *rings*. Ring 0 is the most privileged and ring 3, the least. Modern operating systems run in protected mode. Systems like UNIX that use a two-level privilege system use rings 0 and 3. In Linux for example, the kernel runs (mostly) in ring 0 and user mode is in ring 3. The set of operations available to ring 3 code is restricted by hardware-enforced security mechanisms such as segmentation, paging, and I/O privilege restrictions.