1. Consider an architecture for which all sensitive instructions are privileged instructions (the IBM 360 is an example of such an architecture; the x86 is not). On such architectures it is possible to run an operating system known as a virtual machine monitor (VMM): it supports virtual machines in which operating systems designed to run on the real machine can run, unmodified. The OS running in the virtual machine is called a guest OS. Suppose we run a copy of the VMM itself as the guest OS, and it, in turns, runs a guest OS in one of its virtual machines. This is known as nested virtualization.

Let’s say that OS₀ is running on the real hardware, OS₁ is running as guest OS on OS₀, and, in general, OSᵢ₋₁ is running as a guest OS on OSᵢ. Suppose, for some i>0, a sensitive instruction (which, of course, is also a privileged instruction) is executed within OSᵢ. Thus, since the real machine is running in user mode, a privileged-instruction trap occurs and the (real) hardware invokes the trap handler of OS₀. Assuming the virtual machine running OSᵢ is running in (virtual) privileged mode, explain the sequence of events that must occur to process this instruction. In particular, explain what happens in each of the OSₖ’s, for k<i. Your answer should not be long or complicated; there is no need to supply any code. You don’t need to supply the details of what happens when, say, a virtual machine monitor emulates the effect of a trap in one of its virtual machines; you may simply say that the VMM effects the trap in the virtual machine. You may assume that each VMM is supporting exactly one virtual machine.

Hint: a virtual machine is either in virtual privileged mode or virtual user mode, i.e., it should behave as if it is in either privileged mode or user mode. The virtual machine monitor implementing it keeps track of which mode the virtual machine “thinks” it’s in, but, as far as the VMM is concerned, the virtual machine is always in (real) user mode. Thus if a privileged instruction is executed in the virtual machine, the VMM is notified by a privileged-instruction trap. If the VMM is running on the real machine, the trap is generated by the real hardware. But if the VMM is, itself, running in a virtual machine, what causes the trap to happen? When the VMM responds to the trap, its response depends upon whether the virtual machine it is supporting thinks it is in privileged mode or user mode.

Note that for all k<i, the virtual machine running OSₖ is in user mode. Thus OS₀, seeing that the virtual machine running OS₁ is in user mode, will effect a privileged-instruction trap in that virtual machine. OS₁ will be invoked to handle the trap. If it is supporting a virtual machine (running OS₂), then, if that virtual machine is in user mode, it will effect a privileged-instruction trap in OS₂. So, for k<i-1, OSₖ will effect a privileged-instruction trap in OSₖ₊₁.

Since OS₁’s virtual machine is in privileged mode, the OS₁ will emulate the sensitive instruction, with its virtual machine in privileged mode, making sure that it is proper for OS₁ to be executing it, and adjusting its effect as necessary. This will most likely cause OS₁ to execute a sensitive (and therefore privileged) instruction. Thus the procedure just outlined for OS₁’s executing a privileged instruction will be repeated with OS₁. This will continue until it is OS₀ that’s executing a privileged instruction, but since it’s running in privileged mode on the real hardware, there will be no further privileged-instruction traps.

2. A standard feature on many x86-based systems has been a programmable interval timer (PIT). As used by operating systems, it is given an initial positive value. It then counts down at some frequency (say 1 MHz) until it reaches zero. When this happens it sends a clock interrupt to the processor, resets itself to the initial value, and counts down again, ad infinitum. For the following questions, you may assume a uniprocessor system. The expected answers are relatively short.
a. Assume that the PIT’s initial value is set just once, when the system boots. Explain how the PIT might be used by an operating system to drive a time-slice-based scheduler, where each thread is given a time slice of one centisecond (hundredth of a second), and clock interrupts happen every millisecond (thousandth of a second).

One approach is for the clock interrupt to maintain a 64-bit (so that overflow is not a problem) global counter that starts at zero and is incremented by one at every clock interrupt. When a thread starts its time slice, the global-timer value at which its time slice will be over is computed. Once the clock-interrupt handler sets the counter to that value, the thread is forced to yield the processor to the next thread.

b. Our operating system is now running on a virtual machine. We would like its threads to get one-centisecond time slices of virtual time, i.e., even though several centiseconds of real time may have elapsed, the time slice is not over until the virtual machine has been running for a centisecond. Explain what is done on the virtual machine monitor (VMM) to ensure that this happens. Note that it’s not necessary for these time slices to be measured exactly, as long as they are on average the correct length (with low variance).

The VMM’s clock-interrupt handler is notified of clock interrupts every millisecond of real time. If the VM is running when the clock interrupt occurs, the VMM simulates the occurrence of a clock interrupt on the VM by saving its state on the VM’s current kernel stack and invoking its clock-interrupt handler.

3. We would like to adapt stride scheduling for use with multithreaded processes. The intent is that each process would purchase a certain number of tickets, as in standard stride scheduling. These tickets would be used to determine the processor time to be allocated to the process as a whole. Within a process, the tickets would be distributed among the process’s threads. For example, a four-threaded process might purchase 10 tickets and distribute two tickets to each of three threads, and four tickets to the fourth thread. Another process, say a two-threaded process, might purchase six tickets, with one going to one of its threads and five to the other. In general, an n-threaded process must purchase at least n tickets so that all of its threads can get processor time. The scheduler, when deciding which thread to run, will first determine which process to select a thread from based on per-processor meters governed by the number of tickets held by each process. Then, given the process, it will decide which thread within the process to run based on per-thread meters governed by the number of tickets held by each thread within the process.

Assuming the two aforementioned processes are the only active processes, one will have a meter running at 1/10 the “fair” meter rate, and the other will have a meter running at 1/6 the “fair” rate. When the first (four-threaded) process is chosen to execute (i.e., when its meter has a lower value than the other process’s), whichever of its threads that has the lowest value on its meter will run. The process’s meter will be incremented by 1/10 of a clock tick, the thread’s meter by 1/n of a clock tick, where n is the number of tickets given to that thread. Thread and process meters are initialized based on the number of tickets, as described in lecture.

a. Between the two processes (again, assuming there are no other active processes) there are 16 tickets outstanding. With normal stride scheduling, with only per-thread meters as explained in lecture, after 16 clock ticks the value of each thread’s meter would be increased by 1. Would this be different using the adapted stride scheduling described above? Explain.

After 16 clock ticks, the meters on the two processes would each have been increased by one, with first having been chosen during 10 of those ticks and the other during 6. Thus the four-threaded process would have had 10 ticks for its threads to run. Its four-ticket thread would have run during four of those ticks, with its meter increasing by ¼ tick each.
time. The other three threads would have each run during two of those ticks, with their meters increasing by $\frac{1}{2}$ tick each time. And, similarly, the two-threaded process would have its five-ticket thread run during five of its six ticks, with its meter increasing by $\frac{1}{5}$ tick each time, and the other thread running once, with its meter increasing by one. Thus the effect on each thread’s meter would be the same as with normal stride scheduling.

**b. In normal stride scheduling, the thread with five tickets would run first. Which thread runs first with adapted stride scheduling?**

Since the four-threaded process, with 10 tickets, is selected first, its four-ticket thread would be the first to run.

**c. Suppose we make a further modification to stride scheduling, allowing each process to distribute any number of tickets to its threads, regardless of the number of tickets it, itself, has purchased. Thus, for example, our second (two-threaded) process might distribute one ticket to each of its threads, rather than distributing six tickets to them. After 16 clock ticks, how much actual processor time would each of these threads have run?**

The two-threaded process, as before, would be selected for six of the 16 ticks. Since its threads had paid equal bribes and thus have meters running at identical rates, each would run for three ticks.