

CS167 Homework Assignment 2

Due 10pm October 23, 2009

1. In Unix systems, as shown in slide VIII-8 and on text page 4-7, the processing of the erase character (typically backspace) is done in the interrupt context. In some other systems, most notably Digital's VMS (which was the primary ancestor of Windows NT), such processing was done in the context of the thread reading from the terminal. Explain why the Unix approach makes more sense than the VMS approach.
2. Assume that in a particular computer architecture there is a special register, accessible only in privileged mode, that points to an interrupt vector containing the addresses of the routines that are to handle each of the various types of interrupts known to the architecture. For example, if an unmasked interrupt of type i occurs, then the current processor context is pushed onto the current kernel stack and control transfers to the routine whose address is in the i th component of the vector pointed to by the register.

We would like to virtualize the architecture. Suppose a (real) interrupt occurs while a virtual machine is running.

- a. In which stack is the processor's context saved? (It's not sufficient to say "kernel stack." How is this stack designated?)
 - b. Suppose the VMM decides that the interrupt should be handled by a virtual machine. Explain how it makes this happen.
3. The following code is an alternative to the implementation of mutexes in slide XI-23 (Section 5.1.2 in the text). Does it work? Explain why or why not.

```
mutex_lock(mutex_t *mut) {
    if (mut->locked) {
        enqueue(mut->wait_queue, CurrentThread);
        thread_switch();
    }
    mut->locked = 1;
}

mutex_unlock(mutex_t *mut) {
    mut->locked = 0;
    if (!queue_empty(mut->wait_queue))
        enqueue(RunQueue, dequeue(mut->wait_queue));
}
```

4. The final implementation of *blocking_lock* on page 5-17 (slide XI-29) requires some changes to *thread_switch*. Show how *thread_switch* must be modified. (Note: this is trickier than it might seem at first glance.)
5. We have a new architecture for interrupt handling. There are n possible sources of interrupts. A bit vector is used to mask them: if bit i is 1, then interrupt source i is masked. The operating system employs n threads to handle interrupts, one per interrupt source. When interrupt i occurs, thread i handles it and interrupt source i is automatically masked. When the thread completes handling of the interrupt, interrupt source i is unmasked. Thus if interrupt source i attempts to send an interrupt while a previous interrupt from i is being handled, the new interrupt is masked

until the handling of the previous one is completed. In other words, each interrupt thread handles one interrupt at a time.

Threads are scheduled using a simple priority-based scheduler. It maintains a list of runnable threads (the exact data structure is not important for this problem). There's a global variable *CurrentThread* that refers to the currently running thread.

- a. When an interrupt occurs, on which stack should the registers of the interrupted thread be saved? Explain. (Hint: there are two possibilities: the stack of the interrupted thread and the stack of the interrupt-handling thread.)
- b. After the registers are saved, what further actions are necessary so that the interrupt-handling thread and the interrupted thread can be handled by the scheduler? (Hint: consider the scheduler's data structures.)
- c. Recall that Windows employs DPCs (deferred procedure calls) so that interrupt handlers may have work done when there is no other interrupt handling to be done. How could this be done in the new architecture? (Hint: it's easily handled in the new architecture.)
- d. If there are multiple threads at the same priority, we'd like their execution to be time-sliced — each runs for a certain period of time, then yields to the next. In Windows, this is done by the clock interrupt handler's requesting a DPC, which forces the current thread to yield the processor. Explain how such time-slicing can be done on the new architecture.