

CS167 Homework Assignment 2 Solutions

Fall 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.*

In the Unix approach, characters are erased (and the result echoed back to the terminal) almost immediately after the erase character is typed. But in the VMS approach there could be a rather disconcerting lengthy delay between when the erase character is typed and the character is actually erased on the display, since the thread consuming the character may be doing other things when the erase character is typed.

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?)*

It is saved in the kernel stack of the real machine.

- b. *Suppose the VMM decides that the interrupt should be handled by a virtual machine. Explain how it makes this happen.*

The VMM must first determine which type of interrupt the real interrupt should be mapped to on the virtual machine. It then finds the virtual machine's interrupt-vector register and determines the address of the interrupt handler in the virtual machine. Then it copies the virtual machine's register context, which had been saved on real machines kernel stack and pushes it on the current kernel stack of the virtual machine. Finally it passes control to the virtual machine, in virtual privileged mode, at the address determined from its interrupt vector.

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));
}

```

It does not work. Suppose thread 1 calls *mutex_lock*, successfully locks the mutex, and returns. Thread 2 calls *mutex_lock* for the same mutex and blocks. Thread 1 now calls *mutex_unlock* and thus leaves the mutex in the unlocked state and wakes up thread 2. However, before thread 2 runs, thread 3 calls *mutex_lock*, finds the mutex unlocked, locks it, and returns. Thread 2 now runs, returns from *thread_switch*, sets the mutex to the locked state, and returns. Thus both threads 2 and 3 proceed as if they have the mutex exclusively. Total disaster ensues.

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.)*

As described in the text, *thread_switch* must be passed a locked spin lock that's protecting the wait queue that the blocking thread has joined. Thus this spin lock is not unlocked in blocking lock, but passed in the locked state to *thread_switch*. Note that *thread_switch* must not unlock the spin lock until the blocking thread is no longer running — otherwise we might still run into the problem that the blocking thread is running on two processors at once. However, the spin lock that's passed to *thread_switch* is on the calling thread's stack. Once that thread is no longer running, *thread_switch* has switched to the new thread's stack and thus the argument spin lock is no longer directly accessible. We might consider copying the argument to a static local variable (it certainly wouldn't improve things to copy it to a non-static local variable). However, since *thread_switch* can be called simultaneously by threads running on different processors, this wouldn't work, since there would be just one copy shared by all processors. So, what we might do is to create a new field within each thread's control block called *wait_spinlock*. The caller of *thread_switch* copies its spinlock into this location of the next thread's control block, and then that thread unlocks it prior to returning from *thread_switch*. The modified code for *thread_switch* is below:

```

void thread_switch(spinlock_t *s) {
    thread_t NextThread, OldCurrent;

    NextThread = dequeue(RunQueue);
    NextThread->wait_spinlock = s;
    OldCurrent = CurrentThread;
    CurrentThread = NextThread;
    swapcontext(&OldCurrent->context, &NextThread->context);
    spin_unlock(wait_spinlock);

    // We're now in the new thread's context
}

```

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.)*

They should be saved on the stack of the interrupted thread. This way this thread's stack contains all the thread's context, allowing the thread to be scheduled independently of the interrupt 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.)*

`CurrentThread` must be set to refer to the interrupt-handling thread and the interrupted thread must be put back into the list of runnable threads.

- 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.)*

The purpose of a DPC is to allow an action to be performed without interfering with the handling of further occurrences of the interrupt that generated it. This might be done by having the interrupt threads run at a high priority. DPCs could be handled by one or more threads that run at a lower priority. When an interrupt thread generates a DPC, it might append the request to a DPC request queue that is served by the lower-priority DPC threads.

- 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.*

When there's a clock interrupt, its thread will preempt whatever thread is running at the moment. Assuming that the interrupted thread is put back on the list of runnable threads so that it is next chosen to run *after* all other threads of equal priority, then nothing else needs to be done to implement time slicing.