

CS167 Homework Assignment 1

Due February 14, 2018

1. [25%] The text, on page 101, describes how to switch from one thread to another (this also appears in slide II-24). The implicit assumption is that the thread being switched to in *thread_switch* has sometime earlier yielded the processor by calling *thread_switch* itself. Suppose, however, that this thread is newly created and is being run for the first time. Thus when its creator calls *thread_switch* to enter its context, it should start execution as if its first routine had just been called. Show what the initial contents of its stack should be to make this happen (i.e., please draw a diagram showing the stack). Assume an x86-like assembler language, as used in Chapter 3 of the text. Also, assume the IA-32 calling convention, in which arguments are on the stack.
2. [25%] To fix implementation of *blocking_lock* on page 174 (slide III-32) so that it works, the spin lock should remain locked until *thread_switch* has switched to another thread. Thus *blocking_lock* should pass a pointer to the spin lock to *thread_switch*, where it will be unlocked once the context of the next thread has been entered. Show how to modify *thread_switch* to make this happen. (Note: this is trickier than it might seem at first glance.)
3. In this question we explore some of the details of how signals are implemented in Unix. We saw in class what is done to force a thread to call its signal handler after it notices it has been signaled. What we're discussing here is how the thread notices. Assume that when a thread is signaled, a bit is set in its thread control block (in the kernel) indicating it's been signaled. You may also assume the existence of a routine *InvokeSignal*, called by a thread running in kernel mode, that determines what the thread's response to a signal should be and sets up the thread's user stack, if necessary, to make that response happen. Once *InvokeSignal* returns, the thread then immediately returns from kernel mode to user mode — its user stack has been set up so the right thing happens. The questions here are: when is this routine called, and how does the thread know to call it? (Don't answer them yet; we'll guide you through them.) Though it's important, in this problem you need not describe how the return values from system calls are set up.
 - a. [12%] Since it's important that a signaled thread call *InvokeSignal* itself, and that it call it while in kernel mode, a signaled thread should check whether it's been signaled at some point when it is running in kernel mode. Assuming the thread is either currently running or is runnable (i.e., on the run queue), describe at what points in the thread's execution it should check if it's been signaled and then call *InvokeSignal*. You may assume there are periodic clock interrupts. (Hint: there may be a number of places where this could happen, but we want to make things as simple as possible, but still viable.)
 - b. [13%] Suppose now the signaled thread is sleeping, i.e., waiting on some wait queue in the kernel, and we'd like to wake it up and force it to deal with the signal. In particular, assume the thread has called *wait* (from within the kernel). Normally, when the thread returns from *wait*, it may assume what it's been waiting for has happened (this is not *pthread_cond_wait!*). However, whoever signals the thread, upon discovering the thread is sleeping, will call *unwait* to force the thread to wake up and return from the *wait* call. Thus *wait* returns some special value indicating that it's returning because of an *unwait*, rather than returning because what it's waiting for has happened. The thread should, at this point, check to see if it's been signaled. However, this is not a good place to call *InvokeSignal* — it should be called just before the thread returns to user mode, but the thread may well be deep in nested function calls. Somehow, we must arrange for it to return from all these functions, call *InvokeSignal*, then return to user mode. Explain how this might be done. (Hint:

consider the use of *setjmp* and *longjmp*: see slides XXII-25 and XXII-26 from CS 33 where they are called *sigsetjmp* and *siglongjmp* (<http://cs.brown.edu/courses/csci0330/lecture/22signals1.pdf>.)

4. In the x86 architecture, interrupts are processed using the current thread's kernel stack. However, on the VAX architecture, there was a per-processor interrupt stack on which interrupts were processed. Thus, when an interrupt occurred, the current context was saved on the processor's interrupt stack and the interrupt handler was run on that interrupt stack.
- a. [13%] We discussed in class how Windows uses its DPC facility to implement preemption: when the current thread is to be preempted (perhaps because the current thread's time slice has expired) a DPC (deferred procedure call) is requested. When there are no higher-priority interrupts pending or being processes, a DPC interrupt occurs and the interrupt handler simply calls *thread_switch* to switch from the context of the interrupted thread to the context of the next thread. Explain why this approach doesn't work on the VAX architecture. (Hint: on which stack does the DPC interrupt handler run (and on which it calls *thread_switch*)? Recall that *thread_switch* saves the stack context of the caller and restores that of the thread being switched to. You may assume that the value of the interrupt stack pointer is reset to the top of the interrupt stack on return to user mode.)
 - b. [12%] Instead, the VAX architecture had a feature called *asynchronous system traps* (ASTs), which is similar, but not identical, to the Windows APC (asynchronous procedure call). Among the processor registers is a special control register containing various flags, one of which is the AST-request flag: when this is set, a trap occurs and the AST handler is called on the *kernel stack of the current thread*. This register is among those saved when an interrupt occurs and is restored when an interrupted thread is resumed. When an interrupt handler is entered, a copy of this register is loaded that has the AST-request flag cleared. Explain how ASTs might be used to implement preemption on the VAX architecture, achieving the same effect as the DPC approach on the x86 architecture. (Hint: the contents of saved registers can be modified before they are restored.)