CS167 Homework Assignment 1 Solutions

Spring 2016

1. The figure below illustrates, for the x86 architecture, a pair of stack frames and their linkages. What you are to do for this problem is to show how these linkages (or a suitable modification) can be made to work when successive stack frames do not occupy contiguous storage locations. This might be necessary, say, when a routine discovers that it is near the end of the space allocated for its stack, and allocates additional storage for more stack elsewhere. Then when it calls a subroutine, the new stack frame is in the newly allocated stack extension. Of course, when the subroutine returns, the caller resumes execution in its original stack.

The caller, in such a circumstance, executes code based on the following:

```
... pushl $size_of_additional_stack_space
    call malloc         # call malloc to allocate the stack extension
    addl $8, %esp       # remove malloc’s arguments from the stack
    # %eax now contains the address of the bottom of the stack extension
    addl $size_of_additional_stack_space, %eax
    # %eax now contains the address of the top of the stack extension
    movl %eax, %esp     # %esp now refers to the new stack
    pushl $args         # push subr’s arguments on the (new) stack
    call subr
    addl $4, %esp       # remove subr’s arguments from new stack
    # add code to switch back to the original stack
```
Subr’s code is the following (this is the standard code for entering and exiting a subroutine on the x86 architecture):

```
pushl %ebp            # save %ebp (frame pointer) on the stack
movl %esp, %ebp      # set %ebp to point to the new end of stack
                    # some registers might be saved here
subl $8, %esp        # allocate 8 bytes for local variables
                    # the body of subr goes here
movl $0, %eax        # subr returns 0
                    # restore any registers that were saved
movl %ebp, %esp      # set %esp to refer to the current frame
                    # (popping local variables off the stack)
popl %ebp            # restore the caller’s frame pointer
ret                  # return to the caller (set %eip to point to the
                    # instruction after the call)
```

What must be added to the caller’s code (subr must not be modified) so that, on return from subr, the caller’s execution resumes on its original stack? Note that it might be necessary to insert code somewhere before the call to subr. Recall that the notation n(%eax) is used to refer to the contents of the location whose address is the contents of %eax plus n. Don’t worry about freeing the storage for the stack extension. Points will not be taken off for not getting the x86 assembler syntax exactly right.

The modified caller’s code is as follows. Note that the original code restores %ebp and %eip correctly. What is done here is, in addition, to restore %esp. An alternative approach would be to compute the value of %esp from the restored %ebp, making it not necessary to save %esp on the new stack.

```
... pushl $size_of_additional_stack_space
    call malloc          # call malloc to allocate the stack extension
    addl $8, %esp       # remove malloc’s arguments from the stack
                    # %eax now contains the address of the bottom of the stack extension
    addl $size_of_additional_stack_space, %eax
                    # %eax now contains the address of the top of the stack extension
    movl %esp, 0(%eax)   # save a copy of %esp on the new stack
    subl $4, %eax       # adjust new stack to account for pushed %esp
    movl %eax, %esp     # %esp now refers to the new stack
    pushl $args         # remove subr’s arguments from new stack
    call subr
    addl $4, %esp       # pop arguments off of new stack
    movl 0(%esp), %esp  # restore original stack pointer
```

2. As was shown in class, Unix systems support a large number of special characters that receive special processing by the OS. One such character is the “suspend” character (ctrl-Z by default), that causes a SIGTSTP signal to be sent to the application process. The default action for the signal is to suspend execution of the process. The system responds to the suspend character (by sending the signal) as soon as possible after it is typed in. In older Unix systems there was another special character known as the “delayed suspend” character (ctrl-Y by default) that had the same effect as the suspend character, except that the signal is sent when the application process consumes the character, rather than right away. (That
The suspend character is processed in the interrupt context as the character is received from the terminal. It is not placed on any of the queues, but processed right away. Processing involves deleting the contents of both the partial-line and completed-line queues and sending a signal to the process. For the delayed-suspend character, when it arrives the interrupt handler deletes the contents of the partial-line queue and puts the character at the end of the completed-line queue. Then, when the application thread reads input and reads the line containing (only) the delayed-suspend character, it sends a signal to itself.

3. 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. 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. The DPC interrupt handler calls thread_switch to switch to the context of the next thread. Explain why this approach doesn’t work on the VAX architecture.

Since interrupts are handled on the interrupt stack, the DPC handler would itself be running on the interrupt stack. If it called thread_switch, the current stack context would be saved and the stack context of the next thread (saved in an earlier call to thread_switch) would be restored. But, since there’s only one interrupt stack per processor, it’s likely that the two contexts are on the same stack, thus one thread would overwrite the stack context of the other.

b. 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.

Rather than request a DPC when the current thread is to be preempted, an interrupt handler might set the AST-request flag in the saved copy of the current thread’s control register. Then when this register is restored as part of getting back into the context of the interrupted thread, an AST would occur on the thread’s stack. The AST handler could then safely call thread_switch.

4. 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. 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.)

The simplest approach is to check for pending signals when the thread is about to return to user mode, after completing either a system call or interrupt processing.

b. 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. 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, since we need to make sure the thread immediately returns to user mode. Thus we need to get the thread to a point where it can immediately return to user mode (in particular, its kernel stack must be unwound). Explain how this might be done. (Hint: consider the use of setjmp and longjmp: see slides XXIII-25 and XXIII-26 from CS 33 where they are called sigsetjmp and siglongjmp (http://cs.brown.edu/courses/csci0330/lecture/23signals1.pdf).)

To unwind the stack, the thread should call longjmp. To establish a point to which the stack should be unwound, it should call setjmp. So, when the thread enters a system call, it calls setjmp, saving its state in a jmpbuf in its thread control block:

```c
if (setjmp(tcb->jmpbuf) == 1) {
    return;
} /* start processing system call */
```

On return from wait, if it was “unwaited”, it longjmps:

```c
if (wait(...) == UNWAITED) {
    longjmp(tcb->jmpbuf, 1);
}
```