1. The standard C library provides the pair of routines setjmp and longjmp that, respectively, save and restore the context of the caller. A typical use of these routines is as follows:

```c
jmp_buf jmpbuf;
int TimedInput() {
    signal(SIGALRM, timeout);
    if (setjmp(jmpbuf) == 0) {
        alarm(30);
        GetInput(); /* possible long wait for input */
        HandleInput();
        return 0;
    } else
        return 1;
}

void timeout() {
    longjmp(jmpbuf, 1); /* jump back to the call to setjmp */
}
```

The call to setjmp saves the context of the caller in jmpbuf and returns 0. The program then arranges, by calling alarm, for a SIGALRM signal to occur in 30 seconds. It then calls GetInput, which will return once the user types in a line of input. However, if the signal occurs before the input is typed in, the thread is forced to invoke the signal handler, timeout. The handler calls longjmp, which causes the thread to restore the context that had been saved in jmpbuf. This context includes the thread’s stack pointer (esp), frame pointer (ebp), and instruction pointer (eip). This thread now executes as if it had just returned from setjmp (a second time!). However, what setjmp returns is given by the second argument to longjmp — 1. For more details, see the man pages for setjmp and longjmp.

Assume in this problem that we are using the 32-bit x86 architecture. The code below, disassembled from a program that uses setjmp and longjmp, shows the current implementation of setjmp in glibc. It’s provided for your reference. Understanding it completely is not required for you to answer this problem. Included is code that “mangles” two pointer values. This is done to thwart certain overflow attacks (covered in CS 166). What mangling does is to effectively encrypt the pointers, using as a key a thread-specific value stored in memory referred to by the segment register gs. These values are easily decrypted by longjmp. For the purposes of this problem, you may ignore such mangling and simply assume it doesn’t happen.

```
<_setjmp+0>: xor  $eax,$eax      # set eax to 0.
<_setjmp+2>: mov  0x4(%esp),%edx # put jmpbuf addr in edx.
<_setjmp+6>: mov  %ebx,(%edx)  # save ebx in jmpbuf.
<_setjmp+8>: mov  %esi,0x4(%edx) # save esi in jmpbuf.
<_setjmp+11>: mov  %edi,0x8(%edx) # save edi in jmpbuf.
<_setjmp+14>: lea  0x4(%esp),%ecx # determine what the contents
                            # of esp were before the call
                            # to setjmp.
```
a. [10%] Lecture slide II-26 shows how to implement thread_switch, a routine that switches from one thread to another. Show how this could be done using setjmp to save the context of the calling thread and longjmp to restore the context of the target thread. (Hint: this is relatively easy and involves just a few lines of C code. You may assume that each thread has a control block and that the control block of the current thread is pointed to by CurrentThread. You may add whatever fields you think necessary to these control blocks.)

Assume that each thread control block contains a jmpbuf. Then switch may be implemented as follows:

```c
void thread_switch(thread_t *next) {
    if (setjmp(CurrentThread->jmpbuf) == 0) {
        CurrentThread = next;
        longjmp(next, 1);
    }
}
```

b. [15%] In part a, you assumed that the thread being switched to already existed and had saved its context via an earlier call to thread_switch. For this part, show how to set up a new thread so that
it can start executing via a call (by another thread) to thread_switch. So, implement a (short) routine:

```c
typedef void *(*start_routine_t)(void *);
void start_thread(void *new_stack, thread_t *new_thread,
                   start_routine_t start, void *arg);
```

Assume the caller has allocated the new thread’s stack and thread control block (new_thread), and passes them, along with the address of the new thread’s first function (start) and its first argument. The purpose of start_thread is to initialize jmpbuf and the stack, and return to the caller. Note that if new_thread returns from its first function, it should then call pthread_exit with the value returned as its argument — make sure that you arrange for this to happen. (Hint: use setjmp to do a first cut at initializing the new thread’s jmpbuf, then adjust its values. Recall that pthread_exit does not return!)

You may assume the existence of the routine:
```c
void *CopyFrame(void *esp, void *ebp, int nargs, void *dest_stack);
```

It copies the stack frame starting at esp to the location pointed to by dest_stack. The end of the stack frame is at ebp+n, where n is (nargs+1) sizeof(void *) (this accounts for both the return address and the arguments). In other words, the stack frame includes the arguments being passed to it. Note that dest_stack is the address of the low end of the stack. The frame is copied into the high end of the stack, and thus CopyFrame knows how big stacks are. It returns the address of the low end of the copy of the stack frame, i.e., the address to which esp should point.

Note that the values of the edi, esi, and ebx registers saved in the jmpbuf are not important. You may refer to the fields in the jmpbuf symbolically, as in jmpbuf.oldesp and jmpbuf.ebp.

```c
void start_thread(stack_t *stack, thread_t *tcb,
                  start_routine_t fproc, void *farg) {
    if (setjmp((tcb)->jmpbuf) == 1) {
        // the new thread has just been switched to
        pthread_exit(fproc(farg));
    }
    // tcb->jmpbuf refers to the calling thread’s stack. We must
    // make it refer to the new thread’s stack. In particular, ebp
    // and the old esp must refer to the stack frame in which the
    // new thread begins execution. This stack frame should be a
    // a copy of the calling thread’s stack frame, since the
    // arguments passed on the calling thread’s stack are used in
    // the new thread. Thus the initial contents of the new stack
    // are copied from the calling thread’s stack.
    void *current_oldesp = tcb->jmpbuf->oldesp;
    void *current_ebp = tcb->jmpbuf->ebp;
    tcb->jmpbuf->oldesp = CopyFrame(current_oldesp, current_ebp,
                                     4, stack);

    unsigned diff = (unsigned)(tcb->jmpbuf->oldesp -
                                (unsigned)current_oldesp);
    tcb->jmpbuf->ebp = (void *)((unsigned)current_ebp + diff);
}
```
2. [20%] The final implementation of blocking_lock on page 174 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:

```c
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(CurrentThread->wait_spinlock);

    // We’re now in the new thread’s context
}
```

3. [25%] 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 this feature is no longer supported is perhaps some indication of how useful it was.) The suspend and the delayed-suspend characters have the effect, when they are processed, of deleting all characters currently waiting to be read by the user application that arrived before them. Explain, in terms of the diagram on page 133 of the text and slide VI-14, how both special characters are implemented. Are these characters placed in either of the queues? Which one(s)? In which context (interrupt or thread (and which thread)) are the characters processed?

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.
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.

Since interrupts are handled on the interrupt stack, the DPC handler would itself be running on the interrupt stack, rather than on the interrupted thread’s stack. If it called thread_switch, the current stack context (in particular the stack pointer and other relevant registers) would be saved in the interrupted thread’s thread control block, and the stack context of the next thread (saved in an earlier call to thread_switch) would be restored from its thread control block. But, since there’s only one interrupt stack per processor, if the next thread had itself been preempted by a DPC on the same processor as the current interrupted thread, the two contexts would refer to the same stack (the current processor’s interrupt stack), and thus one thread would overwrite the stack frames of the other.

   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.

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.