# Part I - 20 points each

For yes/no questions, be sure to provide a brief justification.

1. A microprocessor supports a single hardware timer. Suppose instruction SET_TIMER permits one to set the timer value[1]. Does this need to be a privileged instruction?
2. When a process executes a TRAP or is interrupted, the operating system uses a separate stack located in memory unavailable to user processes to execute any operating system code rather than the stack of the current process. Why might operating systems designers select this type of implementation?
3. During an interrupt, how does the operating system select the interrupt service routine to run?
4. Processes P3 and P7 are executing. The system has two types of pending I/O requests. P9 and P11 are waiting on secondary storage reads. P18 is waiting for a network write to complete. P25 and P19 are awaiting assignment of the CPU. Draw a queueing diagram for these processes. Draw an arrow showing where P18 will go once it completes. Write the state that each process is in next to the process bubble.
5. Process P92 has user-level threads. Process P33 has kernel-level threads.
    a. Will P33 run in supervisory mode?
    b. Which process will have faster context switches?
    c. What happens when a thread in P92 or P33 block?

# Part II – 100 points

This is a small programming assignment designed to give you experience in using POSIX threads. Be sure to read the material on POSIX threads which is accessible through Blackboard in the course documents as well as information

[1] Note that this is a conceptual question. Timers are usually controlled by memory-mapped registers, something that we have not yet studied.

in the FAQ. Do not implement a critical section for this assignment. We have not yet covered critical sections and this assignment is designed such that they are not needed. You will have an opportunity to work with critical sections in a subsequent assignment.

It is frequently the case with GUI programs that a progress bar displays the progress that a task has made towards the completion of a task. You will write a subroutine for a "poor person's" progress bar which displays how much progress has been made towards a specific goal. The poor person's progress bar is textual in nature and alleviates you of the need to learn X Window programming, although you are welcome to create a real progress bar if you wish. Should you do this, encapsulate your code so that the grader is not distracted by the additional information needed for X Windows.

You are to write a function called progress_monitor that monitors the progress of a task in a separate thread. Function progress_monitor should take a single argument which is a pointer to the following structure:

```
typedef struct {
        long *  CurrentStatus;
        long    InitialValue;
        long    TerminationValue;
} PROGRESS_STATUS;
```

- CurrentStatus is a pointer to a long which represents the current status of the computation being tracked. *We will refer to the long integer to which this variable dereferences as the progress indicator.* Another thread will be modifying the value to which this points as a task is completed (see below).
- InitialValue is the starting value for the computation.
- TerminationValue is the value at which the computation is complete.

You may assume that TerminationValue >= Progress Indicator >= InitialValue.

The function progress_monitor is only invoked once. It loops until the progress bar is complete and exits. Each time function progress_monitor is allocated the CPU, it should compute the percentage of the task that has been

completed and add to a progress bar of 40 hyphen ("-") characters depending upon the amount of progress that has been made.

As an example of this, let us suppose that InitialValue = 10, TerminationValue = 60, and the progress indicator (*CurrentStatus) = 50. If the progress_monitor thread is scheduled under these conditions, 80% (40/50ths) of the task has been completed. Consequently, .80 * 40 = 32 hyphens should be displayed:

```
--------------------------------
```

When new hyphens need to be added, print them without a line feed character, so that the user will see a smooth progression of hyphens printing on their terminal. Note that putchar()/printf()/cout() typically buffer their output and only make a system call for output once the buffer is full. Request that they be printed immediately by using fflush(stdout) for C or cout.flush() for C++. (If you desire, you may print every 10th hyphen as a +, it will make it easier for you to count the number of characters in the progress bar.)

When the progress indicator has reached the termination value, the thread will print a linefeed and exit the thread. Remember that it is possible that you may need to print more than one hyphen at a time if more than an additional 1/40th of the task has completed since the last time that progress_monitor was scheduled.

The task that you will be measuring the progress on for this program is simple. You will need to write a program which given a file name determines the number of words in the file. Your Makefile should compile the program to output file wordcount. A sample session might appear as follows:

```
edoras> wordcount filename
---------------------------------------
There are 2583298 words in filename.

or

edoras> wordcount filename
---------+---------+---------+---------+
There are 2583298 words in filename.
```

You will be required to write at least three functions although you are certainly encouraged to use more if you see your program becoming unwieldy:

- main - Takes a command line argument of the filename to be counted. Main calls wordcount with either the filename or the file descriptor of the file. Appropriate error handling should be present (i.e error messages of "no file specified", or "could not open file"). After wordcount returns, it prints the number of words.

- progress_monitor - As described above.

- wordcount - Returns a long integer with the number of words and takes a file descriptor or filename as input (your choice). If you choose to pass in a file descriptor and you decide to use high-level input (e.g. the file descriptor returned from fopen as opposed to open) you may add an additional argument indicating the size of the file as that is easier to obtain from the file name using lstat [see below]). If you select to pass in a filename, you will need to open the file and provide error handling if needed. wordcount will spawn a progress_monitor thread with a populated PROGRESS_STATUS structure as the argument. PROGRESS_STATUS should contain:

    - *CurrentStatus - A pointer to a long used by wordcount to store the number of bytes processed so far.

    - InitialValue = 0

    - TerminationValue = Number of bytes in file. (See man fstat or lstat for how to obtain this. fstat requires a file descriptor from a low-level I/O call: e.g. open, whereas lstat uses a filename. If you are using high-level I/O, either use lstat or open the file first with the low-level I/O, call fstat, then close it.)

  It will then read one character a time, updating the number of bytes processed and counting the number of words in the file. We will define a word as a non-zero length sequence of non whitespace characters (i.e. tab, space, linefeed, newline, etc.). You may find it useful to use the library routine iswspace (see man page). You can check your results by using the UNIX command wc (see man page) which provides information about the number of bytes, words, and lines.

Once it is done counting, it waits for the progress_monitor thread to exit and returns the number of words counted.

In addition to the functions described above, you will need to provide a Makefile (see the class FAQ for a tutorial) which can be used with make to compile your program.

The directory ~mroch/lib/cs570/text/ascii/ contains text files from the last assignment along with a conglomeration of many books in the file big.txt. Use these to test your program along with any other files you wish. For the smaller works of Austen and Poe, you are unlikely to see your progress bar increment as the data file is small enough to count it too quickly.

# What to Turn In

You must turn in:

- Paper copy of your work including the program, Makefile, output and the questions. Pair programmers should turn in a single package containing separately answered questions and a single copy of the program.
- All students must fill out and sign either the single or pair affidavit and attach it to your work. A grade of zero will be assigned if the affidavit is not turned in.
- An electronic submission of programs shall be made in addition to the paper copy. A program comparison algorithm will be used to detect cases of program plagiarism.

Remember that all assignments are due at the beginning of class, and the policy on late assignments is described in the syllabus.