ECEn 425 Fall 2016
Midterm 1 Solution

1. (30 pts) Mark each of the following true or false.

a. Normally, ROM can be accessed more quickly than RAM.

False. See the discussion in the text on pages 38 and 39.

b. According to our text, flash memory is unsuitable for storing rapidly changing data.

True. See page 39. Because you have to write data a whole block at a time, the writing process is pretty slow.

c. When watchdog timers expire, they typically generate a high-priority interrupt.

False. They assert RESET on the microprocessor. (See page 71.) There is a big difference between reset and an interrupt; if the software is hung up, interrupts could be disabled, in which case the interrupt would be ignored and the watchdog timer would have no effect. The reset can't be ignored.

d. The action of reading a global variable is always *atomic*.

False. If the variable size exceeds the word size of the machine, it may take multiple accesses just to read a variable. See page 100-101 and the class slides.

e. When 8086 hardware responds to an interrupt, it pushes exactly 32 bits of information on the stack.

False. It pushes three 16-bit words of information. See the lab info pages on the interrupt mechanism.

f. In our programs, memory at address **bp+4** generally holds a local variable within the current function.

False. The memory location at bp+4 actually holds the first incoming argument or parameter for the current function.

g. According to our text, the private context of each task includes a stack.

True. See Section 6.2.

h. Most microprocessors have an interrupt pin associated with an interrupt that cannot be disabled.

True. This is the nonmaskable interrupt discussed on page 89.

i. Our interrupt code must inform the PIC when the end of each ISR is reached.

True. This is an unfortunate consequence of having an external PIC that handles interrupts. The price we pay is making sure we take care of the EOI at the end of each ISR. See the online documentation and the class slides.

j. In an RTOS, a blocked task does not consume any CPU time.

True. See page 140. The ability to block a task that doesn't have anything to do – and have it not consume any CPU time – is a definite advantage of using an RTOS, even though it introduces other complications.

k. In any system running YAK, the first task dispatched is the idle task.

False. It may be the first task that is created, but it would be very unusual for it to be the first task that is dispatched – that actually runs. Look through the application code for Lab 4, for example.

l. In RTOS code, if tasks A and B are deadlocked, both will have a TCB on the ready list.

False. If A and B are deadlocked, both are blocked waiting for a resource held by the other. See the class slides.

m. ISRs may not use semaphores as signaling devices to communicate with task code.

False. See the example in Figure 6.16, for example.

n. Using techniques that include run-time software patches, some computer systems run continuously for decades.

True. This is an interesting tidbit from the "Athens Affair" paper. Some of the computers running telephone switches (stuff that you never want to shut down, if possible) have run for decades, but one has to be able to update them with new software for new functionality, and that is where the patches come in. The intruders modified 29 different code blocks via run-time patches. Whoever did it knew the code rather well.

o. In the "Athens Affair", code executing on individual cell phones was hacked.

False. The code that was hacked was on the Erickson switches in cellular base stations.

2. (6 pts) Define the following terms:
a. *critical section*

A sequence of instructions that must be atomic for the system to work correctly. See page 98. Note that you could take an arbitrary sequence of instructions, disable interrupts at the start, enable interrupts at the end, and it would clearly be atomic, but that doesn't make it a critical section. In other words, its not *being atomic,* but the fact that it *must be atomic or something breaks* that makes it a critical section.

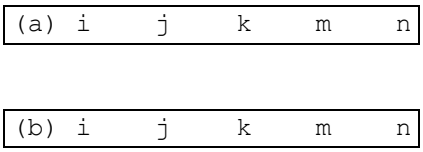
b. *preemption*

Preemption involves the (involuntary) suspension of the current thread of execution to run another that is deemed to be more important. In an RTOS, this occurs when a higher-priority task is executed right away (requiring a context switch) as soon as it becomes ready without waiting for the current task to reach a convenient point in its execution (other than having interrupts enabled) or to otherwise indicate in any way that it is willing to be switched out. The lower-priority task is simply suspended, and it gets to resume when there is no higher priority task in the ready list.

3. (10 pts) In box (a) below, circle each variable that is assigned to a fixed location in memory (not varying at run time) by the compiler/assembler in processing the code below. In box (b), circle each variable for which space will be allocated in the stack frame created by function f. Assume the conventions used by our class tools.

```
int i;
static int j;
```

```
void f(int k)
{
      int m;
      static int n;
      ...
}
```



a. Variables i, j and n will be at fixed locations. Variables k and m will be in dynamically allocated stack frames, and hence at potentially different memory locations for each call to f().
b. The only variable for which space is allocated in f's stack frame is m. (With a more sophisticated compiler, it could also be register allocated.) Variable k was placed on the stack by the caller, and hence is in the caller's stack frame.

4. (5 pts) Assuming that nested interrupts are supported, what components or factors contribute to the latency of the interrupt with the 2nd highest priority?

Let's consider the most significant ones: The first factor is the length of time interrupts might be disabled. Second is the delay in executing the highest priority interrupt code in its entirety (ISR+handler). Third is the time to run this ISR and handler to the point of taking whatever actions we consider to be a suitable response to the interrupt.

5. (5 pts) For software development tools that allow the coding of interrupt routines in C, what will the compiler add to the compiled code that would not be present in a normal C function?

First, think about the structure of a compiled C function, and compare that with the structure of the ISRs you wrote in assembly. Both save some context, do some work, and return. If the compiler is to transform a C function into an ISR, what must it add? Most obviously (see page 91 in the text), the interrupt routine must save and restore the entire context; functions need only save the caller-saved registers that they will use, but interrupt routines can't clobber *any* registers. Interrupt routines use a special return statement (iret on the x86) that must be used at the end. Using the iret and saving/restoring the entire context are absolute essentials. Some other things need to happen in the interrupt routine, but they could be done by the users code in the "interrupt" function. These include: turning interrupts back on to allow interrupt nesting, turning interrupts off before the context is restored, executing an EOI instruction to keep the PIC up to date, and making sure that the interrupt nesting level count is kept consistent (done in YAK by calling YKEnterISR() and YKExitISR()).

6. (8 pts) For a future 425 lab, you will need to determine how big to make the stack for a task. Which of the following do NOT in any way affect the maximum size that a task's stack might grow to? (Circle all that are correct.)

Number of registers in the CPU	Maximum function call nesting in task code
Number of interrupt priority levels	Number of tasks in the system
Return value types of functions called by task	Number of parameters in functions called by task
Size and type of local variables in task code	Length of ISR code in instructions

Let's consider the choices in order. First, recall that each ISR has to save a full context on the stack, and the size of that context is determined by the number of registers in the CPU. (This is the downside of running on, say, a CPU with, say, 128 general-purpose registers.) If interrupts can be nested, the total number of ISR-generated stack frames that can be on the stack at the same time is the number of unique priority levels of interrupts. Functions return values in registers, so this does not affect stack size. Local variables in task code are allocated on the stack, so their number, size, and type need to be taken into consideration. (If your task function has a large array declared local to the function, you'd definitely better pay attention.) Each function called by the task function will put another frame on the stack, and so you'd better consider the maximum possible depth of nested

function calls in your code. The size of task A's stack is not affected by the number of other tasks in the system, since they have their own stacks. Parameters to called functions are placed on the stack, so this also affects maximum stack size. Finally, the length of ISR's (in instructions) does not in any way affect the stack size required.

7. (6 pts) What is priority inversion? Sketch a scenario (showing tasks and events) in which it occurs. Describe one way that an RTOS can address this problem.

See pages 164-165 of the text, as well as the class slides. In essence, a lower-priority task holds a semaphore that a higher priority task blocks on, but a third task (with priority between the other two) keeps the lower-priority task from running and releasing the semaphore. Why the name? In this scenario, once the higher-priority task blocks on the resource, the true importance of the task holding that resource is actually higher than the statically assigned priority indicates. To address this problem, many RTOSs provide pend functions that implement *priority inheritance*, so that the priority of the lower-priority task holding the semaphore is temporarily boosted to the level of the higher-priority waiting task. The priority will be reset to its original value as soon as it posts to the semaphore it holds (or otherwise releases what the higher priority task is waiting for).

8. (10 pts) Circle each YAK function below that can cause a task to be added to the ready list.

YKNewTask()	YKRun()
YKSemPost()	YKEnterISR()
YKInitialize()	YKSemCreate()
YKDelayTask()	YKTickHandler()
YKExitISR()	YKSemPend()

The scenario is that there is a new TCB in the ready list (perhaps more conceptual than literal in your kernel code) when this function finishes that wasn't there when the function started. YKNewTask should be circled: it causes a new TCB to be created for the task in question, and all tasks start in the ready state. YKSemPost should be circled: the post to the semaphore may unblock another task waiting for that semaphore. YKInitialize creates the idle task, so it should also be circled. YKDelayTask causes a TCB to be removed from the ready list, but it cannot cause a new TCB to be placed in the ready list. YKExitISR causes the scheduler to run, but it does not otherwise affect the status of any task in the system. Similarly, YKRun calls the scheduler, but never makes a task "ready". YKEnterISR does not affect task states in any way, nor does YKSemCreate. YKTickHandler can cause a delayed task to be made ready, so it should be circled. Finally, YKSemPend can cause a task to block, but it cannot cause a new task to appear in the ready list.

9. (10 pts) Which of these routines must be reentrant in any YAK kernel? (Circle all that are correct)

YKInitialize()	YKRun()
YKNewTask()	YKEnterMutex()
YKDelayTask()	YKEnterISR()
YKSemPend()	YKTickHandler()
YKIdleTask()	YKDispatcher()

How do we know if a function must be reentrant? Here's the key: is there ever a chance that, while one task or ISR is in a call to the function, somehow we have a context switch or interrupt and another task or ISR makes another call to the same function? If so, it must be reentrant. If, on the other hand, we can convince ourselves that this could NEVER happen, we don't have to make it reentrant. Let's consider the alternatives in order:

- YKInitialize. Never called by ISRs or task code. Does NOT need to be reentrant.

- YKNewTask. Can be called by multiple tasks (and perhaps even ISRs, although this would be bad form). MUST be reentrant.
- YKDelayTask. Can be called by multiple tasks. MUST be reentrant.
- YKSemPend. Can be called by multiple tasks (for same or different semaphores). MUST be reentrant.
- YKIdleTask. Never called by ISRs or task code. (Tasks are only dispatched, never called.) Does NOT need to be reentrant.
- YKRun. Never called by task code or ISRs. Does NOT need to be reentrant.
- YKEnterMutex. Can be called all over the place by tasks and handlers and kernel functions. MUST be reentrant. (Fortunately, it is so simple that it is hard to make it non-reentrant.)
- YKEnterISR. This function is called only within ISRs, and the YAK specs say it must be called BEFORE interrupts are enabled. Thus, there is no way that we can enter a second call to this function before the first finishes. Does NOT need to be reentrant.
- YKTickHandler. This function is called in only one place, namely the tick ISR. Even if interrupts are reenabled, the interrupt at this same priority level will not be responded to by the hardware until this ISR finishes, so there is no way we can reenter this function while we're still in a call to it. Does NOT need to be reentrant.
- YKDispatcher. This function is called only from the scheduler (see the online YAK specs). As was mentioned multiple times in class, it is certainly possible to code a kernel in such a way that the scheduler (and hence dispatcher) are only ever called with interrupts already disabled. (My kernel works this way.) Hence, it does NOT need to be reentrant.

10. (5 pts) Could a watchdog timer be used to detect task starvation? Explain how this could be done, or why it would not be possible.

It can be done. Suppose we want to detect task starvation for task Q. We set up a watchdog timer, initialize it to a carefully determined value, and then write the task code to reset the watchdog timer ("pet the watchdog") on a regular basis when the task runs. If the timer ever expires, we'll know that task Q did not get the processor time it needed. (Resetting the entire system when this happens is admittedly quite heavy-handed, but that is a different issue. Starvation was certainly detected.)

11. (5 pts) Tasks M and N share an array, each must often update many elements in the array, and no other code in the system accesses the array. Of the three shared-data protection mechanisms discussed in the text (disabling interrupts, using semaphores, locking the scheduler), which is most appropriate for this situation? Justify your answer.

You can certainly take a different approach (and defend it), but I think the best choice is using semaphores, despite the associated risks. If the updates are frequent and lengthy, disabling interrupts is probably not a good idea, given the negative impact on interrupt response times across the board. Locking the scheduler is probably better than disabling interrupts since it affects only tasks, but unfortunately it affects all tasks. Semaphores provide "the most targeted way to protect data" (p. 168) and that targeting is important in this case. Remember, however, that semaphores present certain challenges of their own and must be used carefully to avoid problems.