ECEn 425 Fall 2018
Midterm 1 Solution

1. (30 pts) Mark each of the following true or false by circling T or F, respectively.

a. **T F** In many embedded systems, ROM can be accessed more quickly than RAM.

False. ROM is almost always slower than RAM. See the discussion in the text on pages 38 and 39.

b. **T F** Different parts of the address space may each use a different number of wait states.

True. See text, page 56. The system may employ different memory chips that have different access times. Setting the proper number of wait states for each portion of memory is one more thing that software would need to take care of.

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

False. They do a hardware reset. See pages 71-72. A reset is more reliable than an interrupt because interrupts may be disabled, or the interrupt vector table may have been overwritten (unintentionally by buggy software). The whole point of a watchdog timer is to create a failsafe that will get the system operational again the software has crashed.

d. **T F** When 8086 hardware responds to an interrupt, it pushes exactly 32 bits of information on the stack.

False. It pushes three 16-bit values: flags, CS, and IP, for a total of 48 bits of information.

e. **T F** In a hard real-time system, failure to meet a single response time deadline may result in system failure.

True. See the definitions in the class slides. Designers cannot allow the system to miss a single deadline.

f. **T F** Our interrupt code must inform the PIC when the end of each ISR is reached.

True. This is the price we pay on our platform for having an external PIC that handles interrupts. Our ISRs must execute the EOI near the end of each ISR. See the online documentation and the class slides.

g. **T F** Our simulator includes the capability of stopping execution when a particular register value is changed.

True. See the online documentation for the class tools. Knowing what functionality exists in the simulator can save you time in chasing down obscure bugs.

h. **T F** The function-queue-scheduling architecture described in the book uses preemptive scheduling.

False. Preemption involves suspending the current task and switching to a higher-priority task as soon as it unblocks or enters the ready state. At some future point in time, the suspended task will get to resume right where it left off. This kind of preemption happens in an RTOS, but it does not happen in the function-queue-scheduling architecture. Entries are placed in the function-queue in priority order, but once something starts to execute, it runs to completion before the next task starts. That is not preemptive scheduling.

i. **T 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.

j. **T F** The operation 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.

k. **T F** Priority inversion is a form of deadlock.

False. In priority inversion, a higher priority task is waiting for a lower priority task to release a resource that it holds exclusively. Deadlock requires a circular dependence such that no task can proceed. The two problems are unrelated.

l. **T F** Correctly written task code is always a C function that will eventually terminate and return.

False. Correctly written RTOS tasks are never called, they generally do not terminate (unless they delete themselves using a kernel function) and they never return.

m. **T F** In any system running YAK, the first task to be dispatched is always the idle task.

False. The idle task is the first task *created*, but it would be the first task *dispatched* (in YKRun) only if the application did not create at least one other task in main, and that would not be good programming practice.

n. **T F** Any task that calls YKSemPend() must also call YKSemPost() or the code will not execute properly.

False. Semaphores can be used for signaling (as opposed to protecting critical sections), in which case the task receiving the signal just calls the pend function. See class slides and page 162 in the text.

o. **T F** 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 to work correctly* 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 voluntarily suspend execution. The lower-priority task is simply suspended, and it gets to resume when there is no higher priority task in the ready list.

3. (8 pts) Suppose you are examining a log file on an embedded system running the YAK kernel. You notice that, at a particular point in time, control switches from task A to task B. The priority of B is lower than the priority of task A. Circle each of the following that could have caused the switch to B.

B was made ready by the tick handler.	A pended on a semaphore.
B forced A to block.	A posted to a semaphore.
B changed its status to ready.	A delayed itself.
An interrupt handler posted to a semaphore.	An interrupt handler pended on a semaphore.

This is very similar to a problem on the midterm from last year, but we've been given the added information that B is lower priority than A, and that changes the possible answers. Let's consider them in order: if the only thing that happened was B being made ready by the tick handler, we would not find it running, since A has higher priority. There is simply no way that B can force A to block – tasks can't do this, and B wasn't running anyway. Similarly, B can't change its own status to ready, as this is an operation performed by the RTOS, and B wasn't running anyway. If an interrupt handler posted to a semaphore, that could cause B to unblock, but that won't make B run because B has lower priority than A. Task A could have pended on a semaphore that was not available, causing it to block and allowing B to run. If A posted to a semaphore, that might have made a higher priority task run, but it could not make B run because B has lower priority. Task A could have delayed itself, allowing B to run. Finally, it is forbidden to have interrupt handlers pend on semaphores, so this is not something you'd observe in a system designed by BYU grads.

4. (6 pts) Circle each of the 6 variables below that is stored on the stack.

```
int i;
static int j;
void f(int k, int *p){
    int l;
    static int m;
    ...
}
```

You should have circled k, p, and l. (The variable p is certainly stored on the stack; we have no idea what it points to, but the question asks about where the variable itself is stored.) Both i and j are global variables, and m is actually a global that will be stored at a compiler-generated label so it can't be accessed by anything outside this routine. However, each iteration or instantiation of this function will use the same global variable when m is referenced, something that is definitely a problem in the context of reentrant functions.

5. (6 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.

6. (6 pts) What are the three things a function must NOT do if it is to be classified as reentrant?

The function must not (1) use shared variables in a non-atomic way, (2) call a non-reentrant function, or (3) use hardware in a non-atomic way. See discussion in Section 6.2.

7. (5 pts) Can you increase the throughput of a system (e.g., rate at which it processes data) by increasing the frequency of the timer interrupt (corresponding to the tick interrupt in our simulator)? Why or why not?

The short answer here is no. There is something close to a fixed overhead associated with the interrupt code for each timer tick, and ramping up the tick frequency simply runs that code more often, resulting in more overhead. If the processor was otherwise busy running task code, this overhead will actually reduce performance. Why isn't the rate of data processing connected to the timer tick? You could build a system where this happens, but it wouldn't make a lot of sense. The inputs the system is processing typically show up via some interface device that likely has its own interrupt. To increase throughput, you want to increase the efficiency of responding to those interrupts and processing the associated data, not add overhead in the form of more frequent tick interrupts.

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) Circle all the functions below which *must be* reentrant in any YAK kernel.

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. (8 pts) A key responsibility of RTOS application code is to specify the size of each task stack. Circle each of the following that does NOT in any way affect the maximum size required for a task stack.

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 (how many instructions they take to execute) does not in any way affect the stack size required.

11. (5 pts) Some RTOSs can change *task* priorities dynamically. Is it possible to dynamically change the priority of an *interrupt* associated with a particular input device? Explain how this could be done, or why it cannot be done.

In conventional microcontrollers, interrupt priority levels are determined by which pin on the microprocessor (or PIC in the case of the 8086) the device is connected to. If you want to change the priority level of a particular device, you have to rewire it and change its physical connection. Note that changing the interrupt vector table simply changes the software routine (ISR) that runs when an interrupt at a particular priority level is responded to – it does not change the priority level associated with interrupts caused by a particular device, so long as it is connected to the same input pin on the microprocessor.