ECEn 425 Fall 2019
Midterm 1 Solution

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

a. **T F** 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 relatively slow.

b. **T F** Wait states are a mechanism to suspend the processor while it waits for interrupts.

False. Wait states are a mechanism that allows the processor to insert extra bus cycles in its transactions with slower memory devices. See the discussion in the text beginning on page 55.

c. **T F** Many microprocessors have an interrupt pin associated with an interrupt that cannot be disabled.

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

d. **T F** The 8086 interrupt model implemented in our simulator includes 8 hardware interrupts.

True. See the class slides and the online documentation.

e. **T F** The **bp** register normally points to a memory location containing the base address of the previous stack frame.

True. See the class slides on stack frame layouts and your own code.

f. **T F** Shared data problems arise only when shared variables are modified within multiple functions.

False. In virtually all of the examples in the text and class slides, a single routine writes the shared variables and a single routine reads the shared variables. The challenge is to make a sequence of accesses atomic rather than to avoid multiple writers.

g. **T F** In systems using an RTOS, each ISR has its own stack.

False. Each ISR uses the stack of the currently executing task.

h. **T F** C's **volatile** keyword warns the compiler that a variable may be changed by something the compiler can't see.

True. From page 103: "The volatile keyword, part of the C standard, allows you to warn your compiler that certain variables may change because of interrupt routines or other things the compiler doesn't know about."

i. **T F** The worst-case latency for a particular interrupt includes the time to run higher priority ISRs.

True. Lots on this in the class slides, and see page 104 in the text.

j. **T F** If ISRs can interrupt tasks, then the operating system is necessarily using *preemptive scheduling*.

False. Preemptive scheduling means that the highest priority ready *task* is next to execute, even if the current task has not voluntarily given up the CPU. This form of scheduling is handled by the RTOS. In contrast, the scheduling of ISRs is handled by the hardware, and that mechanism always gives interrupt code priority over regular (non-interrupt) code – provided, of course, that interrupts are enabled.

k. **T F** If a YAK task is in the Blocked state, it must have previously been in the Running state.

True. YAK is a conventional RTOS, in that each task is created in the Ready state, and it blocks only "because it decides for itself that it has run out of things to do" (p. 140). Thus, it had to run in order to be in the Blocked state.

l. **T F** When a system using YAK powers up, the kernel runs before a single instruction of user code is executed.

False. The first thing that runs in any compiled C code is the body of the main() function, and this is in user code in YAK (and other RTOSes).

m. **T F** According to our text, disabling task switches is "the most targeted way" of protecting shared data.

False. See page 168. The "most targeted way" is actually using semaphores.

n. **T F** ISRs may not use semaphores as signaling devices to communicate with task code.

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

o. **T F** The phone-tapping software installed on Vodafone systems by intruders was described as a *rootkit*.

True. According to the "Athens Affair" paper, a rootkit is software that alters the operation of system code and tries to keep you from detecting its presence. Why is this relevant? Some rootkits on desktop systems have made headlines over the years, and this paper describes "the first time a rootkit has been observed on a special-purpose system." Given the increasing popularity of real-time embedded systems, we'll probably see more examples of rootkits in such systems over the years. Seems that security in embedded systems might be a good area of specialization….

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

In an RTOS, a semaphore is a globally visible shared variable accessed only indirectly through kernel functions (creating, obtaining, and releasing it) that can be used to enforce mutual exclusion or to provide a signal for synchronization. If the function is called to obtain a semaphore and it is not available, the caller will be blocked until the semaphore becomes available (when it is released by the task holding it).

b. *reentrant function*

A function is reentrant if it can be called by more than one task (or ISR) and still work correctly in all cases, even with context switches that result in two or more active calls to the same function on different task stacks. This is critical in creating multi-tasking application code, and it is achieved by applying the three rules described in our text: (1) not using global variables in a non-atomic way, (2) not using the hardware in a non-atomic way, and (3) not calling functions unless they are also reentrant.

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 as it processes 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) i    j    k    m    n
```

```
(b) i    j    k    m    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) Describe the actions of the 8086 hardware in response to an asserted, enabled interrupt.

First, there is some handshaking with the PIC to get the interrupt priority level, and then that value is used to access the interrupt vector table for a segment-offset pair (two 16-bit values) that constitutes the address of the ISR to transfer control to. The hardware saves 3 16-bit values on the current stack (IP, code segment register, and the flags) and then starts fetching instructions from the corresponding ISR. That's a total of two 16-bit reads, three 16-bit writes before a single instruction in the ISR is executed.

5. (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 pretty heavy-handed, but that is a different issue. Starvation was certainly detected.)

6. (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).

7. (8 pts) A particular ISR completes all of the actions below (in some order) when it runs. Circle all those actions which must take place before calling YKExitISR().

call interrupt handler	send the EOI command to the PIC
restore context	call YKEnterISR()
disable interrupts	execute an IRET instruction
save context	enable interrupts

See the discussion of ISR actions on the Interrupt Mechanism web page. A typical ISR performs these actions in this order (remember that the hardware will have disabled interrupts before it runs): (1) the context is saved, (2) YKEnterISR() is called, (3) interrupts are enabled (so higher priority interrupts can be serviced while the handler is running), (4) the handler is called and it runs to completion, (5) interrupts are disabled (for what amounts to a critical section at the end of the ISR), (6) the EOI command is sent to the PIC, (7) YKExitISR() is called (which calls the scheduler, possibly causing a different task to be dispatched at this point, otherwise returning to the ISR so that control will return to the task that was interrupted), (8) the context of the interrupt task is restored, and (9) the IRET instruction is executed, returning control to the interrupted task. Thus, all of these except for restoring context and executing an IRET must have already taken place before YKExitISR() is called.

8. (8 pts) Assuming normal YAK functionality, when the dispatcher runs for the first time, which of these kernel functions must already have been called at least once? (Circle each correct answer.)

YKNewTask()	YKRun()
YKScheduler()	YKEnterISR()
YKInitialize()	YKSemCreate()
YKDelayTask()	YKTickHandler()

By "normal YAK functionality" I mean according to the YAK specifications on the web page and the examples of YAK code used in class. The dispatcher runs for the first time within YKRun() at the end of main(), since this call causes task code to run for the first time. At that point (assuming correctly written code), main() will have called YKInitialize() and YKRun(). The code in main() will also have created at least one task, so YKNewTask() will have been called at least twice at that point, including the call to create the idle task. (The YAK specs state: "At least one user-defined task must be created with a call to YKNewTask() before YKRun() is called.") Similarly, YKRun() calls YKScheduler() which calls the dispatcher, so YKScheduler() will have been called exactly once. (From the YAK webpage: "This routine [YKRun()] causes the scheduler to run for the first time (which then calls the dispatcher).")

Until YKRun() is called, the system is not really ready to start normal operations, and you probably don't want to start dealing with interrupts, so it would be a bad thing if YKEnterISR() or YKTickHandler() had already run once. YKSemCreate() should have been called if the application requires a semaphore (best to do initialization of semaphores in main()), but not all applications require semaphores. Finally, YKDelayTask() will not have been called yet since it is called only by task code and no task has yet executed.

9. (10 pts) Circle each YAK function below that must include a call to the scheduler in its source code, according to the kernel specifications.

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

In general, we need a call to the scheduler at the end of any kernel function that might have changed the status of any task in the system. (The call to the scheduler could be conditional, based on whether it is called from interrupt code or task code, for example, but it will be present in the source code.) This certainly includes functions typically called by task code, including YKNewTask(), YKSemPost(), YKDelayTask(), and YKSemPend(). Let's consider the other cases: YKInitialize() is called only once, at the start of main() in application code. It does not call the scheduler because the system doesn't start executing task code until YKRun() executes, and that is the last thing done by the code in main(). YKRun() calls the scheduler to get task execution started. YKEnterISR() and YKExitISR() are called in ISR code while interrupts are disabled. We don't call the scheduler in the former, but we do in the latter (if we're returning to task code) to make sure we're executing the highest priority ready task. YKSemCreate() cannot change the status of any task, so it need not call the scheduler. (As discussed in class, it is a good idea for application code to create all semaphores in main() before calling YKRun().) Finally, YKTickHandler() is a special case because it can change the status of a task but we do NOT want to call the scheduler at this point. We have to execute the current ISR code to the point of executing the EOI command or our interrupt priority levels will be messed up – the call to the scheduler in YKExitISR() will ensure that we execute the highest priority ready task when we finish the ISR.

10. (6 pts) What solutions exist for the shared-data problem when the sharing is between task and ISR code?

Of the primary alternatives we discussed (disabling interrupts, using semaphores, locking the scheduler), it should be clear that disabling interrupts is the only alternative when sharing is between task and interrupt code. (Interrupt code cannot take a semaphore, since it cannot be blocked in the same way that task code is blocked. This means there is no way to keep the ISR from running in the case when the task code is in its critical section. Locking the scheduler has no impact on whether interrupt code executes or not.) One should not, however, neglect the other alternatives discussed in the text beginning on page 107. These include double buffering, the use of a circular queue, and repeatedly reading the value until it is the same twice. They can be thought of as "programming tricks" that certainly could be made to work, but they probably wouldn't be your first choices without compelling reasons to avoid disabling interrupts.

11. (6 pts) Why is it essential that semaphores be referenced indirectly (using pend and post, for example) in application code rather than having tasks and ISRs access the semaphore data structures directly?

There are two main points here.

1. If we access the semaphore value directly, we'll have a shared data problem on the semaphore, and avoiding shared data problems is often the motivation for using semaphores to begin with. We could still make this work, by disabling interrupts, reading and writing the semaphore, and then enabling interrupts, but you'd have to do this with EVERY access to the semaphore in the code. This would be painful. Far better to encapsulate the access details *within* reentrant functions that can be called with little if any hassle throughout your code.
2. If we access the semaphore value directly, what do we do when it is not available? Pend will transparently cause the caller to block – it would be tricky to get that functionality without another RTOS function comparable to pend. (This might take the form of a "block me until this happens" function, but why not simplify the code and do it through pend?) Trickier, how do you know if some other task is blocked on the semaphore when it is released with a post (so that it should be made ready)? Under no circumstances should task code be poking around the ready and blocked lists. It is far better to let post and pend take care of all of this behind the scenes.