ECEn 425 Fall 2017
Midterm 1 Solution

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

a. In most flash memories, data can only be written in blocks that are much larger than a word or byte.

True. See page 37.

b. DMA is circuitry that moves data directly between I/O devices and memory without CPU intervention.

True. See the top of page 79.

c. Watchdog timers are restarted under software control.

True. Software resets or restarts the timer, which should never expire in normal operation. If it does expire, the software is assumed to have crashed, and the timer resets the processor, which allows execution to start over.

d. In our target architecture, 8 distinct interrupt request lines (IRQs) are connected directly to the 8086.

False. Only one hardware interrupt line is connected to the processor, which would make it quite a bit more complicated to figure out what device generated the interrupt and needs attention if it were not for the PIC that simplifies things considerably for the CPU. The PIC has 8 distinct hardware interrupt lines (see class slides) and handshakes with the CPU to make it appear in many respects as if the 8 IRQ lines were attached directly, but the interface is still more complicated because they are separate devices. One example: having to execute the EOI command at the end of an ISR. (This just allows the PIC to keep track of what the CPU is doing as it responds to interrupts.)

e. A critical section is any code sequence that must be atomic for the system to work properly.

True. See page 98 and the class slides.

f. In our class labs, each ISR must explicitly disable interrupts at lower priority levels.

False. Look at your code for lab3 and lab4. We never have to explicitly modify the interrupt mask register in this manner. It is one of the nice freebies that the PIC does for us. After calling the handler, each ISR does disable interrupts, but this affects all interrupts and not just those at lower priority levels.

g. Our simulator includes the capability of listing the addresses of the most recently interrupted instructions.

True. Check out the “hist” command. If you didn’t know this existed, you really should review the commands once in a while. If you are having some strange behavior with interrupts, this can save you hours of debugging.

h. The only actions performed in YKInitialize() are initializing kernel data structures.

False. According the YAK specs, it not only does initialization and bookkeeping, but it must also create the idle task.

i. The failure of the Mars Rover was related to priority inversion.

True. See the class slides. This is another example of the risks of using semaphores.

j. An RTOS with priority inheritance necessarily has task priorities that change at run-time.

True. As page 165 of the text points out, priority inheritance involves the temporary boosting of task priorities, so they change at run-time.

k. Any function that does not write global variables is necessarily reentrant.

False. It isn't just *writes* to global variables we need to be concerned about, nor is it just global variable accesses that make functions non-reentrant. See page 149 for the three rules we need to apply to determine if a function is reentrant.

l. In systems using an RTOS, each ISR has its own stack.

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

m. In RTOS code, if tasks A and B are deadlocked, neither will have a TCB on the ready list.

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

n. In YAK, a value greater than 0 indicates that the semaphore is available.

True. This is fairly standard practice for semaphores, and it should be clear from the pseudo code for YKSemPend that we discussed in class. (See the class slides.)

o. The "Athens Affair" perpetrators installed on-the-fly patches in over 25 separate code blocks.

True. On page 31, the article points out that 29 (of about 1760 total) code blocks had a series of patches applied to support the functionality the perpetrators wanted. In the process of making these run-time changes to the executing code, it would have been very easy to bring down the system. Whoever did this really knew the code.

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

From page 97 of the text, a part of a program is atomic if it cannot be interrupted, or split into pieces. Obviously we are making a portion of our code atomic when we disable interrupts. The trick is to recognize when we need to do it.

b. *hard real time systems*

In a real-time system, actions must be taken in a timely way, usually before specific deadlines pass. A hard real-time system is one in which no deadlines can be missed without severe consequences, including possibly the failure of the entire system.

3. (10 pts) From a log file of a system running the YAK kernel, you notice that, at a particular point in time, control switches from task A to task B. Circle each action below that could have caused the switch to task 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 moved itself to the ready list.	A delayed itself.
An interrupt handler forced A to block.	A created a new task.
An interrupt handler posted to a semaphore.	An interrupt handler pended on a semaphore.

Let's consider the possibilities in order.

1. **B was made ready by the tick handler.** We don't know anything about the relative priorities of A and B. B may have been made ready by the tick handler, and then it would preempt A if B were higher priority, so this action should be circled.
2. **B forced A to block.** There is simply no way that B can force A to block – there is no kernel function that a task can call to do this to another task, and B wasn't running anyway. This action should not be circled.
3. **B moved itself to the ready list.** This is not possible. A task never runs unless it is already in the ready list, so there is no way a task can move itself to the ready list. This action should not be circled.
4. **An interrupt handler forced A to block.** This is not possible. There is no kernel function that an ISR can call to force a task to block. This action should not be circled.
5. **An interrupt handler posted to a semaphore.** Interrupt handlers can post to semaphores, and B may have been blocked on that semaphore and also have higher priority than A. This action should be circled.
6. **A pended on a semaphore.** Task A could have pended on a semaphore that was not available, causing it to block and allowing B to run. This action should be circled.
7. **A posted to a semaphore.** If A posted to a semaphore that B was blocked on, then the post would have caused B to unblock and B would preempt A if B had higher priority. This action should be circled.
8. **A delayed itself.** Task A could have delayed itself, allowing B to run. This action should be circled.
9. **A created a new task.** Task A could have created task B, and task B would then preempt task A if it had higher priority. This action should be circled.
10. **An interrupt handler pended on a semaphore.** It is forbidden to have interrupt handlers pend on semaphores (interrupt code can't be blocked like task code), so this is not something you'd observe in an operational system with any BYU grads involved. This action should not be circled.

4. (5 pts) Describe the actions of the 8086 <u>hardware</u> 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. (6 pts) In a non-preemptive RTOS, should there ever be shared data problems between tasks? Why or why not?

See problem 10 in Chapter 6. In a non-preemptive RTOS you still have tasks and ISRs, but the tasks run until they are interrupted, in which case control returns to the same task, or until they voluntarily give up the CPU (by delaying themselves or blocking until some shared resource becomes available). Assuming the code is well written (think BYU-grad quality), tasks would never cause themselves to block while in a critical section, so there <u>should</u> never be any shared data problems between tasks. (In poorly written code, anything could happen, but without preemption the shared data problem – between tasks – is certainly much easier to avoid.)

6. (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. All of these must have taken place before YKExitISR() can be called: save context, call YKEnterISR(), enable interrupts, call interrupt handler, disable interrupts, and send EOI command to PIC. Only after the call to YKExitISR() can the context be restored and the IRET instruction executed.

7. (6 pts) Relative to the other software architectures described in Chapter 5, what are the main advantages and disadvantages of using an RTOS?

Some major advantages:

1. With preemption, tasks don't have to wait for each other to complete, improving response time for high priority tasks. Moreover, adding a new lower-priority task doesn't affect response time of higher-priority tasks.
2. The system takes care of scheduling – user code just assigns task priorities and the system runs what is most important. Nothing in user code has to decide what to do next.
3. The system provides very useful blocking, unblocking, delaying, and signaling functionality.
4. There are lots of kernels available, commercial and otherwise.

Some major disadvantages:

1. The user software is more complex. (You can mess up with semaphores in a whole bunch of ways, for example.) You have to design tasks, assign priorities, identify critical sections, solve shared data problems, write ISRs and handlers, and the fun doesn't stop there. (The RTOS itself has some complexity, as you are discovering in your design, but application developers will just buy a kernel from a vendor, so this is not an issue for the software they have to design and debug.)
2. There is more runtime overhead in an RTOS. (Those kernel routines occupy program memory, and they take time to run.)

8. (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 still be conditional, based on whether it is called from interrupt code or task code, for example.) 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 would be a good idea for the 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.

9. (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.

10. (5 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 easy to see 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. 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 our 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 at EVERY access to the semaphore in the

code. What are the chances we mess up somewhere? 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 calling pend. (You'd have to call some "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 marked ready)? Under no circumstances should task code be poking around the ready and blocked lists. Just let the post and pend take care of all of this behind the scenes.