The University of Texas at Austin
Department of Electrical and Computer Engineering

Real-Time Systems / Real-Time Operating Systems

EE445M/ECE380L.12, Spring 2022

Midterm Exam Solutions

Date: March 24, 2022

UT EID: ________________

Printed Name: ____
Last, First

Your signature is your promise that you have not cheated and will not cheat on this exam, nor will you help others to cheat on this exam:

Signature: ____

Instructions:
- Open book, open notes and open web.
- No electronic devices other than your laptop/PC (cell phones off and stowed away).
- You are allowed to access any resource on the internet, but no electronic communication other than with instructors.
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided.
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

Problem 1	30	
Problem 2	10	
Problem 3	30	
Problem 4	20	
Problem 5	10+10	
Total	100+10	

Problem 1 (30 points): Cooperative Context Switching

Given the following basic cooperative round-robin context switch implementation:

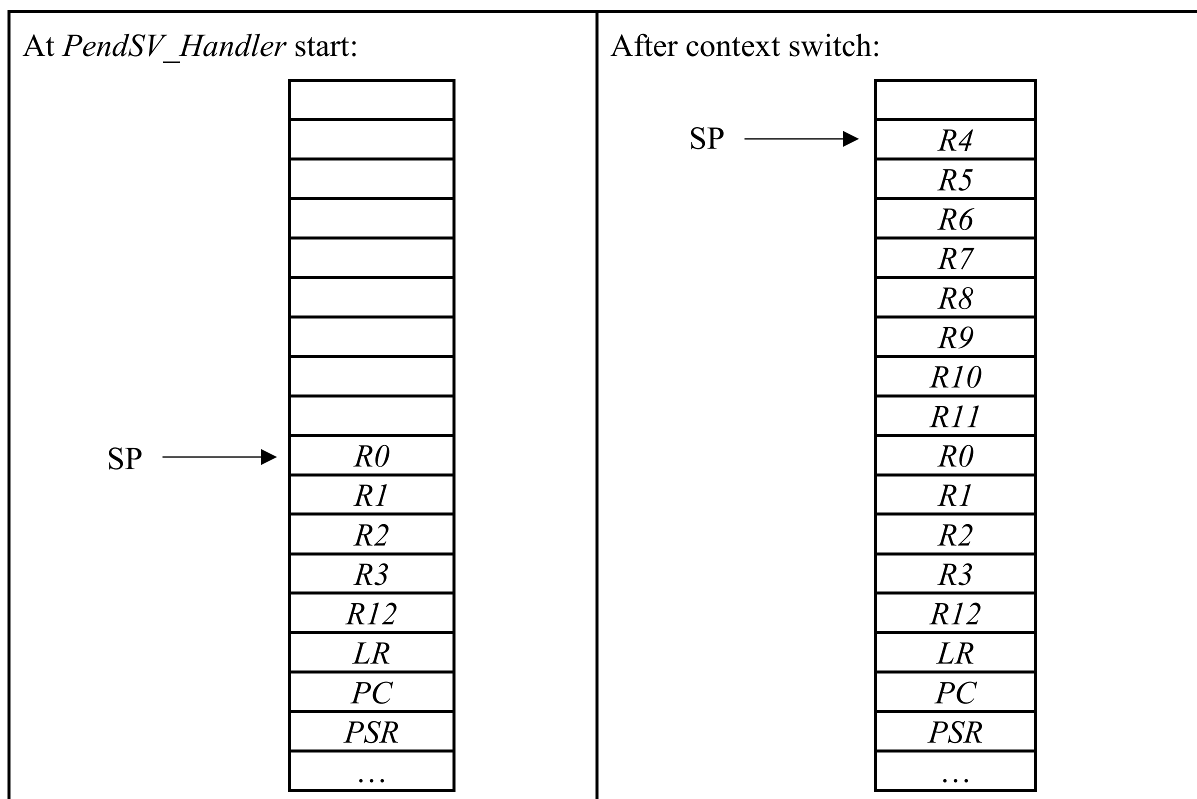
```
struct TCB {
  long *sp;
  struct TCB *next;
}

struct TCB *RunPt;


void OS_Suspend(void) {
  NVIC_INT_CTRL_R=0x10000000;
}
```

```
PendSV_Handler
  CPSID   I
  PUSH    {R4-R11}
  LDR     R0, =RunPt
  LDR     R1,[R0]
  STR     SP,[R1]
  LDR     R1,[R1,#4]
  STR     R1,[R0]
  LDR     SP,[R1]
  POP     {R4-R11}
  CPSIE   I
  BX      LR
```

a) Is this implementation correct or does it have bugs? Show any needed bug fixes.

This implementation is correct, there are no bugs.

b) Assuming a given thread's stack state right before calling *OS_Suspend*, show what additional information is stored on the stack at the time when the first line of the *PendSV_Handler* is executed and when the context switch to the next thread is complete.



c) What will be the first instruction being executed by the next thread when the context switch is complete, i.e. the *PendSV_Handler* exits?

The PendSV_Handler will return to the instruction after it was triggered, i.e. to the final BX LR instruction in OS_Suspend.

d) The PendSV interrupt normally needs to have the lowest priority. Why is that? What happens if the PendSV has a higher priority than another interrupt in the system?

If PendSV has a higher priority, it may interrupt another interrupt handler, i.e. it will be a nested interrupt situation. In that case, R0-R3, R12, LR, PC and PSR are saved twice on the stack. When the PendSV_Handler then switches to another thread, it will only have one set of auto-saved registers on the stack and the PendSV_Handler will try to return to a foreground thread instead of another interrupt. This will lead to a hard fault due to a corrupted machine state (handler vs. thread mode).

Also, it's not a good idea to context switch in the middle of another interrupt handler. This will make for a potentially very long-running interrupt service routine.

e) Why does the PendSV_*Handler* disable interrupts during its execution? What would happen if the CPSID/CPSIE instructions at the beginning and end of the handler were removed?

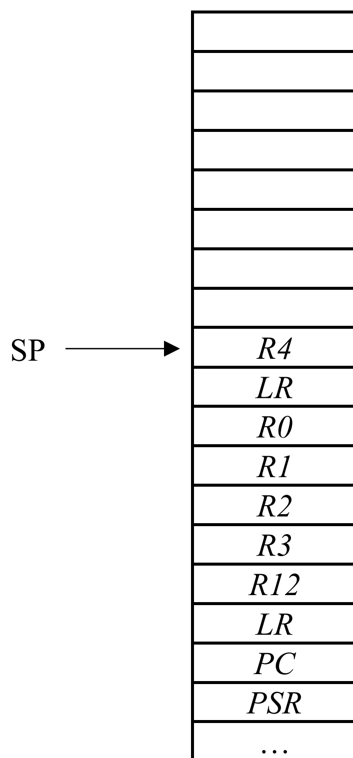
The PendSV_Handler modifies the global shared RunPt variable. This creates potential race conditions and critical sections with other code that accesses RunPt, e.g. if another interrupt handler (background thread) calls OS_AddThread. Interrupts are disabled in the PendSV_Handler to provide mutual exclusion and protect such critical sections.

Problem 2 (10 points): Preemptive Context Switching

Given the basic round-robin context switch implementation from Problem 1, a developer wants to turn his cooperative OS into a preemptive one using the following SysTick implementation:

```
void SysTick_Handler(void) {
0x00000708 B510      PUSH        {r4,lr}
    PendSV_Handler();
0x0000070A F7FFFDEE  BL.W        PendSV_Handler
} // end SysTick_Handler
0x0000070E BD10      POP         {r4,pc}
```

Will this work? Why or why not? Assuming a given state of the stack at the time when a SysTick interrupt is triggered, show the information on the stack when the first line of the *PendSV_Handler* that is called from the *SysTick_Handler* is executed. Explain whether this stack state is a potential problem or not, and under what conditions. If there are potential problems, is there a way to have the *SysTick_Handler* call the *PendSV_Handler* directly that works under all conditions?



This will not work in general. The stack state is incompatible with and different from the stack state when entering the PendSV_Handler normally, e.g. if OS_Suspend is called by a foreground thread (see Problem 1). This will lead to problems (popping the wrong information off the stack) if a PendSV_Handler that is called from the Systick_Handler tries to switch to a thread that was previously switched out by a regular PendSV_Handler triggered via OS_Suspend, and vice versa.

There is a way to have the SysTick_Handler call the PendSV_Handler directly. It requires removing the PUSH and POP instructions and then branching to instead of calling the PendSV_Handler, i.e. replacing the BL with a B instruction while making sure that this branch is the last instruction of the SysTick_Handler (it won't return).

This, however, also requires adjusting the interrupt priorities for SysTick and PendSV to have the same (lowest) priority and thus make sure that a SysTick can not interrupt the PendSV_Handler. Otherwise, if a SysTick were to occur e.g. right after enabling interrupts but before the final BX LR in the PendSV_Handler, it leads to a nested PendSV situation similar to Problem 1d).

Problem 3 (30 points): Scheduling

You are given a system with 4 tasks listed below.

Task	Execution Time	Period
T1	0.25 ms – 1 ms	5 ms
T2	0.5 ms – 1 ms	6 ms
T3	1 ms – 2 ms	10 ms
T4	3 ms – 6 ms	15 ms

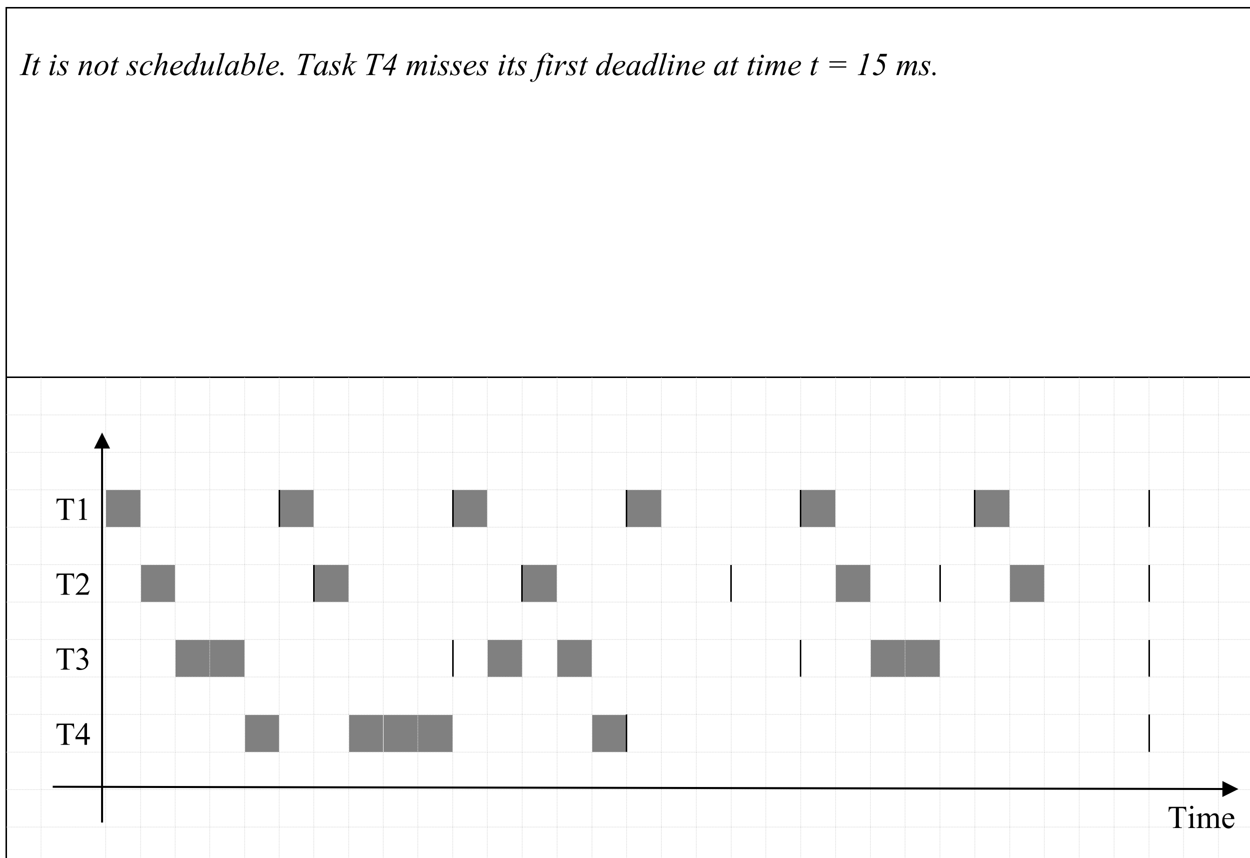
a) What is the best-case and worst-case CPU utilization running these tasks?

Best case: 0.25/5 + 0.5/6 + 1/10 + 3/15 = (1.5 + 2.5 + 3 + 6) / 30 = 13 / 30 = 43.33%

Worst case: 1/5 + 1/6 + 2/10 + 6/15 = (6 + 5 + 6 + 12) / 30 = 29 / 30 = 96.67%

b) Is this task set schedulable using RMS? Show the worst-case RMS schedule at the critical instant. If RMS is unable to meet any deadlines, specify which deadlines are missed.

It is not schedulable. Task T4 misses its first deadline at time t = 15 ms.

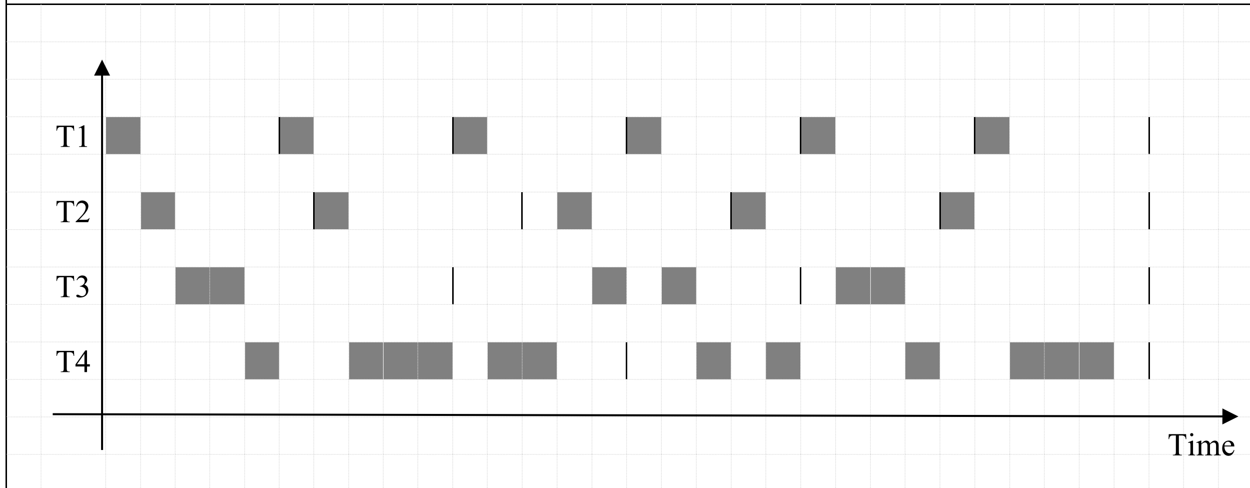


*Name:*______________________________

c) Is this task set schedulable using EDF? Show the worst-case EDF schedule at the critical instant. If EDF is unable to meet any deadlines, specify which deadlines are missed.

Taskset is schedulable. No deadlines missed.

Note that there are multiple possible schedules. Schedule below gives priority to lower task ID in case of same deadlines.



d) You profile your OS running the EDF schedule and find that the context switch overhead is extremely high. What is the maximum overhead for the context switch that will still allow your EDF schedule from c) to run? You can assume that context switch overhead is only incurred when the OS actually switches to a different task.

There are 23 context switches in the EDF schedule above (this depends on which EDF schedule is chosen), and 1ms of time left. Also, every task has at least 1ms slack before its deadline. Hence, the maximal context switch delay that is possible is 1/23 = 0.43ms.

Problem 4 (20 points): Synchronization and Deadlocks

a) Given a round-robin scheduler, if you want to minimize CPU utilization, what semaphore implementation would you use? Why?

Blocking semaphores. They require significantly fewer CPU cycles than spinning semaphores.

b) Given a round-robin scheduler, if you want to minimize jitter, what semaphore implementation would you use? Why?

Spinlock semaphores. They generally have interrupts disabled for a shorter maximum time.

c) Assume a system that doesn't have semaphores and only uses *StartCritical()* and *EndCritical()* to protect critical sections, can this system ever have deadlocks?

There can not be any deadlocks as there can be no hold-and-wait situation.

d) Assuming we implement a deadlock detector that runs as lowest-priority task and checks whether no other task is running or sleeping, i.e. all other tasks are waiting on semaphores, will this detector be able to detect all deadlocks? Why or why not?

No, it will not detect all deadlocks. It will only detect deadlocks that involve all tasks. There can be deadlocks that only involve a subset of tasks while other tasks keep on running.

Problem 5 (10+10 points): Bulk Synchronization

A bulk or barrier semaphore waits until all threads have reached a checkpoint before letting any thread proceed. In other words, all threads must enter *OS_WaitBulkSema* before any thread is allowed to leave the function. Given below is an implementation of a bulk semaphore that is cyclic, i.e. can be used more than once (and *OS_WaitBulkSema* can be called multiple times per thread). This implementation has several bugs. Identify all the bugs in the code and explain what erroneous behavior will occur with the current implementation. Bonus points for providing fixes to the bugs.

```
struct bulk_sema = {
  sema_t mutex;
  sema_t barrier;
  int count; // Threads waiting
  int n; // total # of threads
 };
 typedef bulk_sema bsema_t;
```

```
OS_SignalMulti(sema_t *sema, int value)
{
  int i;
  for (i=0; i<value; i++)
    OS_Signal(sema);
}
```

```
void OS_InitBulkSema(bsema_t *sema,
                     int n)
{
  // # of participating threads
  sema->n = n;

  // initialize barrier & mutex
  OS_InitSemaphore(&sema->barrier,
                   0);
  OS_InitSemaphore(&sema->mutex,
                   1);

  // no waiting threads yet
  sema->count = 0;

}
```

```
OS_WaitBulkSema(bsema_t *sema) {
  OS_bWait(&sema->mutex);

  // Increase # of waiting threads
  sema->count++;

  // All threads have reached?
  if (sema->count == sema->n)
  {
    // Let all threads progress
    OS_SignalMulti(&sema->barrier,
                   sema->n);

    // And reset for next use
    sema->count = 0;
  }

  // Finally wait ourselves
  OS_Wait(&sema->barrier);

  OS_bSignal(&sema->mutex);
}
```

This implementation has two bugs:

1) *Deadlock when waiting for the barrier while holding the mutex. Holding the mutex means no other task can enter the function, so the barrier can never be notified.*
   -> Fix: Move OS_bSignal(&sema->mutex) to right before OS_Wait (&sema->barrier)
   Note that moving releasing the mutext any earlier time would lead to a race condition on modifications and checking/resetting of the count variable, which need to be atomic.
2) *There is a race condition when releasing the barrier. A thread that is released through the barrier may call OS_WaitBulkSema again and then grab the barrier up to n times before other threads have a chance to go through it, effectively locking them out.*
   -> The fix for this is more complicated. One needs to not let any thread into the barrier until all threads have gone through it. This can be done by having another barrier semaphore that counts down until all threads are through before being released for the next iteration.