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

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:

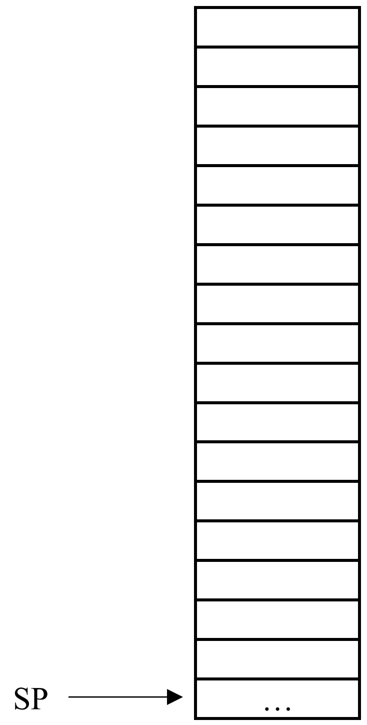
```
struct TCB {
  long *sp;
  struct TCB *next;
}

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
```

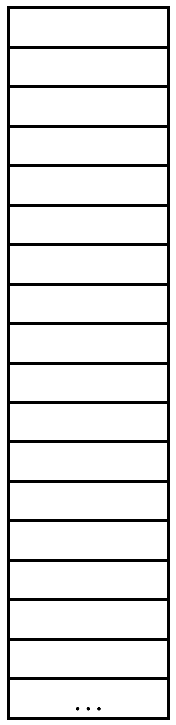
a) Is this implementation correct or does it have bugs? Show any needed bug fixes.

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.

At *PendSV_Handler* start:



After context switch:



*Name:*______________________________

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?

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?

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?

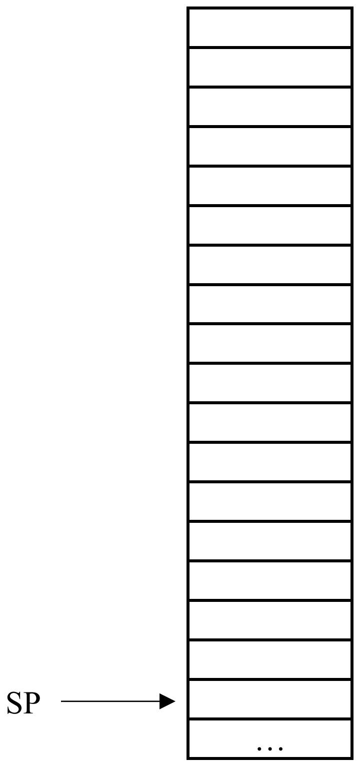
*Name:*______________________________

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?



*Name:*________________________________

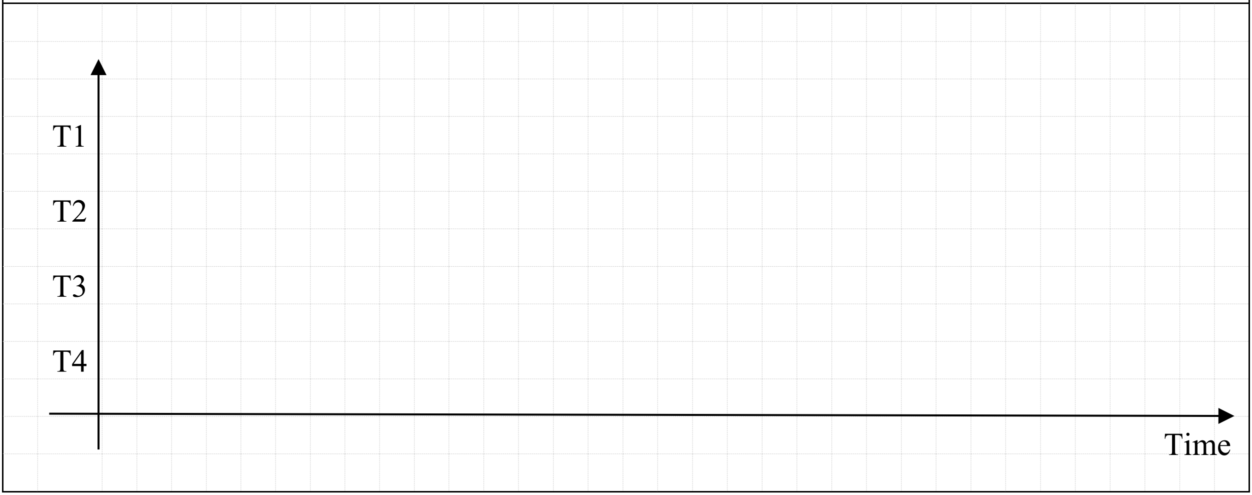
Problem 3 (30 points): Scheduling

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

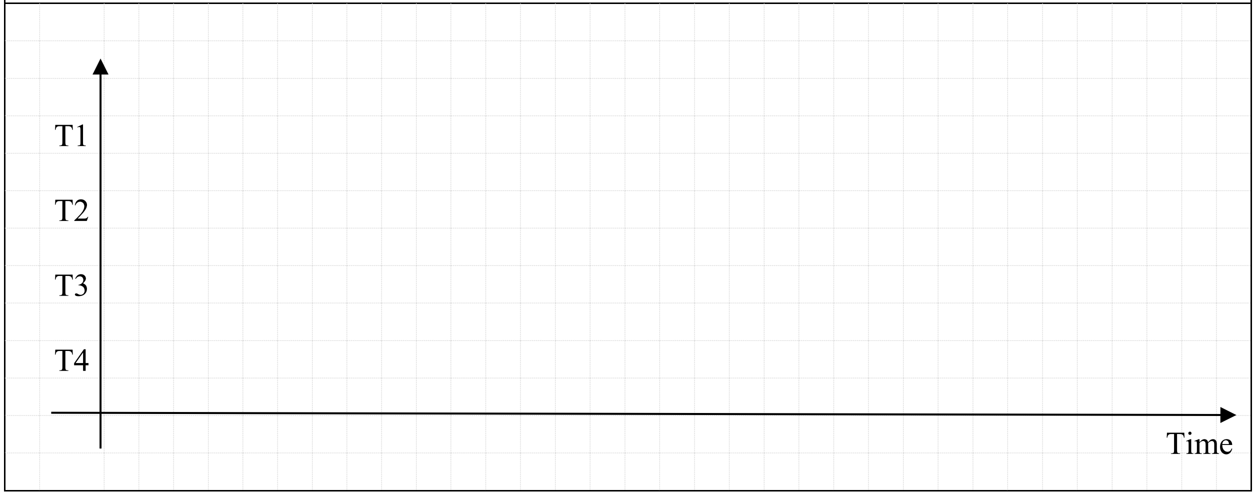
a) What is the best-case and worst-case CPU utilization running these tasks?

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.



*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.



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.

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?

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

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?

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?

*Name:*______________________________

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);
}
```

*Name:*________________________________