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

Real-Time Systems / Real-Time Operating Systems

ECE445M/ECE380L.12, Spring 2023

Midterm Solutions

Date: March 2, 2023

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	20	
Problem 2	20	
Problem 3	20	
Problem 4	25	
Problem 5	15	
Total	100	

Problem 1 (20 points): Miscellaneous

a) What are two advantages of preemptive schedulers over cooperative schedulers? What are some advantages of cooperative schedulers?

Preemptive scheduling advantages: does not depend on cooperation, e.g. in case of misbehaving applications, potential for quicker reaction times since OS can preempt any time it needs to.

Cooperative scheduling advantages: precise control over scheduling points, less overhead

b) Describe what happens in your Lab 2/Lab 3 OS when you call *OS_Sleep()* from a background thread.

OS_Sleep will trigger a context switch/suspend either in the middle of the interrupt handler or right after it finishes (if suspend just triggers a PendSV). In the former case, the stack gets corrupted. In the latter case, it will end up sleeping the thread that was interrupted by the background thread.

c) Describe what happens when you call *OS_Wait()* from a background thread with spinlock semaphores? What about with blocking semaphores?

For spinlock semaphores, it will potentially end up looping forever inside the interrupt handler (unless there is a higher priority interrupt that calls OS_Signal to wakeup the same semaphore).

For blocking semaphores, it will mark the thread that was interrupt as blocked in the TCB/blocked list and then trigger a context switch/suspend from within the interrupt handler. Similar to OS_Sleep, this will either corrupt the stack or will end up blocking the thread that was interrupted by the background thread.

d) There are several ways to make critical sections atomic. What is the difference between using *Disable*/*EnableInterrupts*, *Start*/*EndCritical*, and semaphores for synchronization?

Disable/EnableInterrupts unconditionally disables/enables interrupts.

Start/EndCritical remembers the interrupt enable status before disabling and then restores instead of unconditionally enabling.

Sempahores do not disable interrupts in the critical section, i.e. only lock out threads that want to access the same semaphore, not all other foreground and background threads. But introduce the potential for deadlocks (with multiple semaphores).

Problem 2 (20 points): Context Switching

Given the following TCB structure:

```
typedef struct TCB TCB_t;
struct TCB {
    uint32_t  id;         /* thread ID */
    uint32_t* sp;         /* stack pointer */
    TCB_t* next;          /* pointer to the next TCB */
    TCB_t* previous;      /* pointer to the previous TCB */
};
TCB_t RunPt;
```

a) Write assembly code for a context switch not done in an interrupt handler (i.e. not using *PendSV* or any other interrupt) using the TCB defined above assuming a round-robin scheduler.

```
ContextSwitch
     CPSID I
     PUSH  {R0-R12, LR}   ; push registers
     LDR   R0, =RunPt     ; load address of RunPt
     LDR   R1, [R0]       ; load RunPt
     STR   SP, [R1,#4]    ; save stack pointer
     LDR   R1, [R1,#8]    ; load next TCB pointer
     STR   R1, [R0]       ; save new RunPt
     LDR   SP, [R1,#4]    ; load new stack pointer
     POP   {R0-R12, LR}   ; pop registers
     CPSIE I
     BX    LR
```

In reality, the PSR should also be saved in this case. This can be done using MRS/MSR instructions, e.g., as follows, but was not required for this question:

```
ContextSwitch
     CPSID I
     PUSH  {R0-R12, LR}   ; push base registers
     MRS   R0,PSR         ; save PSR
     PUSH  {R0}
     LDR   R0, =RunPt     ; load address of RunPt
     LDR   R1, [R0]       ; load RunPt
     STR   SP, [R1,#4]    ; save stack pointer
     LDR   R1, [R1,#8]    ; load next TCB pointer
     STR   R1, [R0]       ; save new RunPt
     LDR   SP, [R1,#4]    ; load new stack pointer
     POP   {R0}           ; restore PSR
     MSR   PSR,R0
     POP   {R0-R12, LR}   ; pop registers
     CPSIE I
     BX    LR
```

b) Write assembly code for a context switch in the *PendSV Handler* using the TCB defined above assuming a round-robin scheduler.

```
PendSV_Handler
     CPSID I
     PUSH  {R4-R11}        ; push remaining registers
     LDR   R0, =RunPt      ; load address of RunPt
     LDR   R1, [R0]        ; load RunPt
     STR   SP, [R1,#4]     ; save stack pointer
     LDR   R1, [R1,#8]     ; load next TCB pointer
     STR   R1, [R0]        ; save new RunPt
     LDR   SP, [R1,#4]     ; load new stack pointer
     POP   {R4-R11}        ; pop remaining registers
     CPSIE I
     BX    LR              ; return from interrupt
```

Problem 3 (20 points): Synchronization

Given the application below that runs on an OS using a strict priority scheduler with blocking sempahores:

```
uint32_t Count1 = 0;
uint32_t Count2 = 0;
uint32_t Count3 = 0;
```

```
sema_t mutex;
sema_t lock;
```

```
void Thread1(void) {
  OS_Sleep(50);
  while(1){
    OS_Wait(&mutex);
    OS_Wait(&lock);
    Count1++;
    OS_Sleep(10);
    OS_Signal(&lock);
    OS_Signal(&mutex);
  }
}

void Thread2(void) {
  while(1) {
     OS_Wait(&lock);
     OS_Wait(&mutex);
     Count2++;
     OS_Sleep(50);
     OS_Signal(&mutex);
     OS_Signal(&lock);
  }
}
```

```
void Thread3(void) {
  OS_Sleep(10);
  while(1) {
    Count3++;
  }
}

void Idle(void) {
  while(1) {
    WaitforInterrupt();
}

main {
  OS_Init()
  OS_InitSemaphore(&mutex, 1);
  OS_InitSemaphore(&lock, 1);
  OS_AddThread(Thread1,0);
  OS_AddThread(Thread2,2);
  OS_AddThread(Thread3,1);
  OS_AddThread(Idle,3); // lowest prio
  OS_Launch(2MS_TIME);
}
```

a) The code above has a bug. Find and describe the bug, and describe possible solutions.

The code above will run into a priority inversion that will prevent Thread1 from executing. This happens because Thread1 and Thread3 first sleep and then Thread2 grabs the semaphores and goes to sleep itself. Once Thread3 wakes up, it will run indefinitely and Thread2 will never run again to release the semaphores. Thus, when Thread1 wakes up, it will also be blocked waiting on these semaphores indefinitely. As such, only Thread3 will run in the end.

To fix priority inversion, one can use priority ceiling or priority inheritance protocols to increase the priority of Thread2 holding the semaphores to at least the highest priority of threads that are waiting on the semaphore (Thread1). That way Thread3 will be preempted to let Thread2 run and release the sempaphores, which in turn will allow Thread1 to also run.

Other solutions involve changing the application, e.g. changing the priority of the threads statically. But the code then has other bugs, see b).

b) Now assume that a student removes *Thread3*. This code still has a bug. Explain what issue remains and how it might be fixed. Is there any solution that fixes both a) and b) without changing the application?

This removes the priority inversion, but there is still the possibility of a deadlock in this code. This can happen when threads wakeup and are scheduled and interleaved in such a way that Thread1 will wake up and preempt Thread2 right at the point where Thread2 has grabbed the lock but not yet the mutex semaphore. This is very unlikely to happen, but can potentially occur the first time Thread1 wakes up since both threads wake up at around the same time depending on how time slices align. Then Thread1 will try to grab the semaphore for mutex causing deadlock. The way to fix this is to acquire semaphores in the same order in Thread1 and Thread2.

A solution that fixes both the priority inversion problem in a) and the deadlock in a) and b) is to use a priority ceiling protocol for sempahores. In this case, as soon as Thread2 grabs lock, it will be elevated to the highest system priority and can then not be interrupted by Thread1 any more.

Problem 4 (25 points): Scheduling

You are given a system with 3 tasks listed below.

Task	Execution Time	Period
T1	2ms	5ms
T2	3ms	10ms
T3	4ms	15ms

a) What is the CPU utilization running these tasks? Is a system with this CPU utilization guaranteed to be schedulable by RMS and/or EDF?

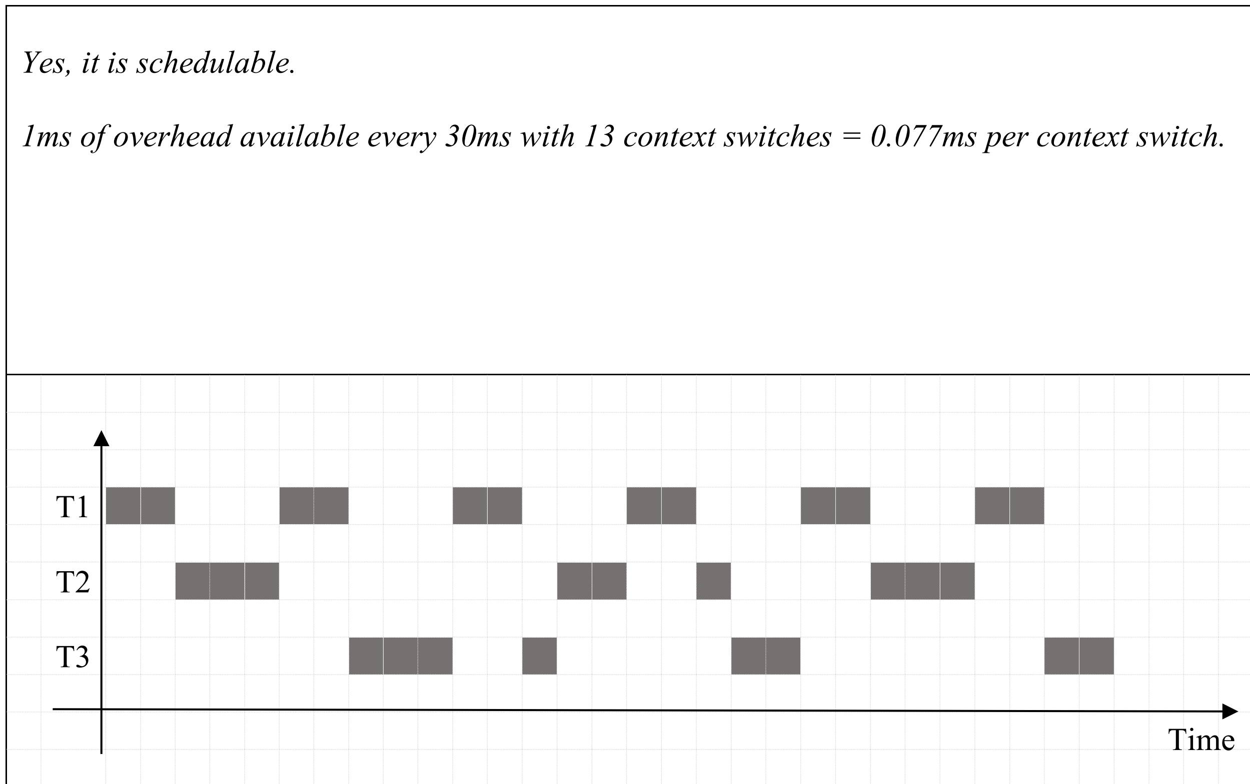
Utilization: 2/5 + 3/10 + 4/15 = 0.967

This is guaranteed to be schedulable by EDF (<100%) but not RMS.

b) Show the EDF schedule on the diagram below. Is this task set schedulable by EDF? If so, what is the maximum overhead that context switching can take. If not, specify what deadline is missed.

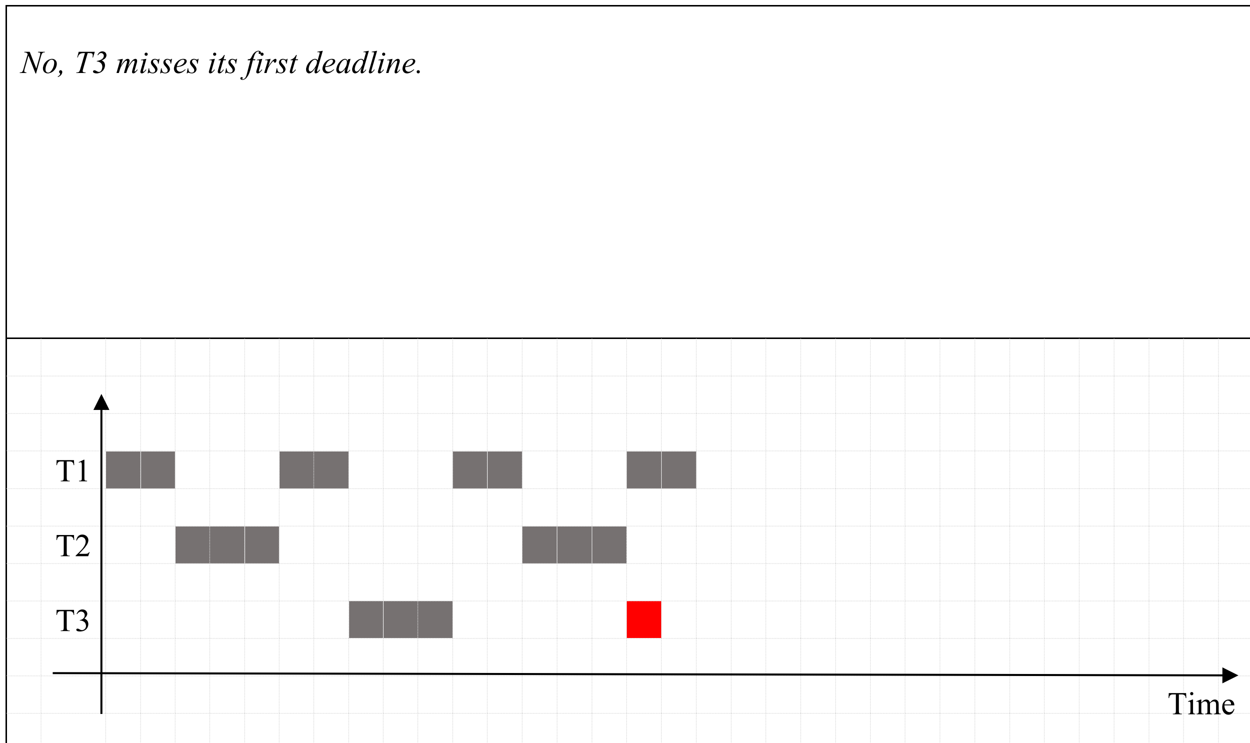
Yes, it is schedulable.

1ms of overhead available every 30ms with 13 context switches = 0.077ms per context switch.



c) Show the RMS schedule on the diagram below. Is this task set schedulable by RMS? If so, what is the maximum overhead that context switching can take. If not, specify what deadline is missed.

No, T3 misses its first deadline.



Problem 5 (15 points): Semaphores

a) Implement a spinlock semaphore that will call *OS_DeadlockResart* after a certain amount of time (DEAD_LOCK_TIME, given in ms) if the semaphore can not be acquired before. You may call *OS_MsTime*, which returns the system time in ms counting up. Multiple threads can call *OS_Wait* on the same or different semaphores. Do not add anything to the TCB.

```
void OS_Wait(Sema4Type *semaPt) {
     DisableInterrupts();
     uint32_t timeWaitedStart = OS_MsTime();
     While ((*semaPt) == 0) {
          EnableInterrupts();
          uint32_t currentTime = OS_MsTime();
          If ((timeWaitedStart-currentTime) > DEAD_LOCK_TIME) {
               OS_DeadlockRestart();
          OS_Suspend();
          DisableInterrupts();
     }
     (*semaPt) = (*semaPt) - 1;
     EnableInterrupts();
}
```

b) What would you need to do to implement a similar deadlock checker for a blocking semaphore? Assume that the blocking semaphore functions are implemented using a linked list per semaphore. (You do not need to code this, but simply explain/write some pseudo code.)

To implement a checker on a blocked semaphore, one would need to setup a timer to periodically check how long a thread has been blocked and when it reaches a certain time it calls OS_DeadlockRestart.

To implement the exact same semantics as in a), one would need to add a field to the TCB that stores the time when a thread was blocked, and the compare that time to the current time on every timer tick to check whether the limit has been exceeded.

Alternatively, one can just impose a maximum time that a semaphore can be held. In that case, a field is added to the semaphore that notes the time when a semaphore is first acquired, and check this value on every timer tick. When OS_Signal is called semaphore is released), the the time value in the semaphore is reset.