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

# Real-Time Systems / Real-Time Operating Systems

**EE445M/EE380L.12, Spring 2019**

---

## Midterm

**Date:** March 28, 2019

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 and open notes.
- No calculators or any electronic devices (turn cell phones off).
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided.
- *Anything outside the boxes will be ignored in grading.*
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

| | | |
|---|---|---|
| **Problem 1** | 20 | |
| **Problem 2** | 25 | |
| **Problem 3** | 15 | |
| **Problem 4** | 20 | |
| **Problem 5** | 20 | |
| **Total** | 100 | |

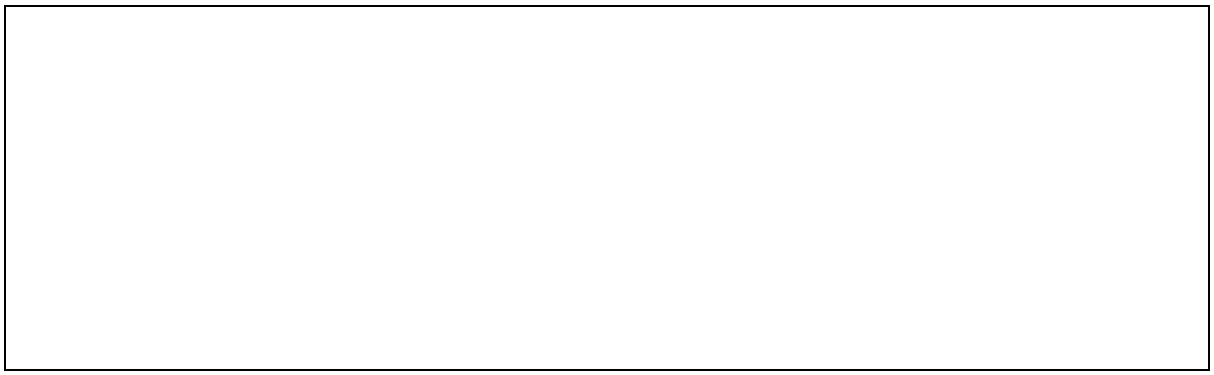
**Problem 1 (20 points): Context Switching**

If you recall the original context switch code shown in class (and used in the lab), you might have noticed that the stack pointer (SP) was never saved on the stack. Instead, it was stored in each thread's TCB. Why? The following implementation stores the SP on the stack just like all of the other registers. Although we only show the modified version of the context switch routine and TCB below, you can assume that *OS_Init*() and *OS_Start*() have been modified accordingly.

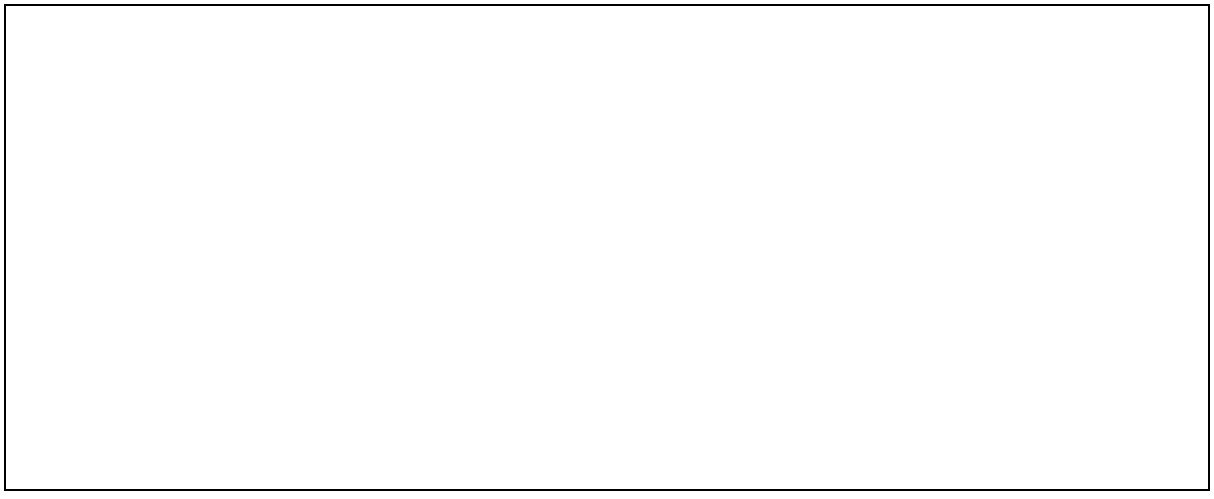
```
struct TCB {
  struct TCB *next;
  long stack[128];
}
typedef struct TCB tcb;


tcb *runPt;
```

```
PendSV_Handler   ; Saves R0-R3,R12,LR,PC,PSR
  CPSID I        ; Disable interrupts
  PUSH {R4-R11}  ; Save R4 - R11
  PUSH {SP}      ; Save SP
  LDR  R0,=RunPt ; R0 = pointer to RunPt
  LDR  R1,[R0]   ; R1 = RunPt
  LDR  R1,[R1]   ; R1 = RunPt->next
  STR  R1,[R0]   ; RunPt = R1
  POP  {SP}      ; Restore SP
  POP  {R4-R11}  ; Restore R4 - R11
  CPSIE I        ; Enable interrupts
  BX   LR     ; Restores R0-R3,R12,LR,PC,PSR
```

a) What is wrong with the above implementation? What will happen if you try to run it?



b) Can you think of a way to fix the code to have a working context switch without making changes or additions to the TCB or any global data structure? List all your assumptions and show changes you need to make to the context switch routine. Hint: This is harder, so do this last. There are multiple possible solutions.



**Problem 2 (25 points): Multi-Threaded Programming**

For each of the examples below, does the code have any potential race conditions or deadlocks in a multi-threaded environment? Either describe why the code is correct, i.e. free of any such bugs, or list all issues and fix the code such that it becomes free of any of them. Assume that unless noted otherwise, all variables, semaphores and data structures have been properly initialized.

a) Assume this is the only code in the module, where *get_seconds* is the only routine that can be called by external threads:

```
static struct time {
  unsigned int h;
  unsigned int m;
} total_time = { TOTAL_HOURS, TOTAL_MINUTES };



unsigned long get_seconds(void)
{


  return ( total_time.h * 60 + total_time.m ) * 60;


}
```

b) What about this case, assuming *set_time* is the only routine that can be called externally:

```
static struct time {
  unsigned int h;
  unsigned int m;
} total_time = { 0, 0 };



void set_time(unsigned int h, unsigned int m)
{


  total_time.m = m;

  total_time.h = h;


}
```

c) What about if the module now contains both of these the routines as the (only) externally callable ones?

```
static struct time {
  unsigned int h;
  unsigned int m;
} total_time = { 0, 0 };



void set_time(unsigned int h, unsigned int m)
{


  total_time.m = m;

  total_time.h = h;


}


unsigned long get_seconds(void)
{



   return ( total_time.h * 60 + total_time.m ) * 60;



}
```

d) Finally, what about this abridged (incomplete) code of a very special OS:

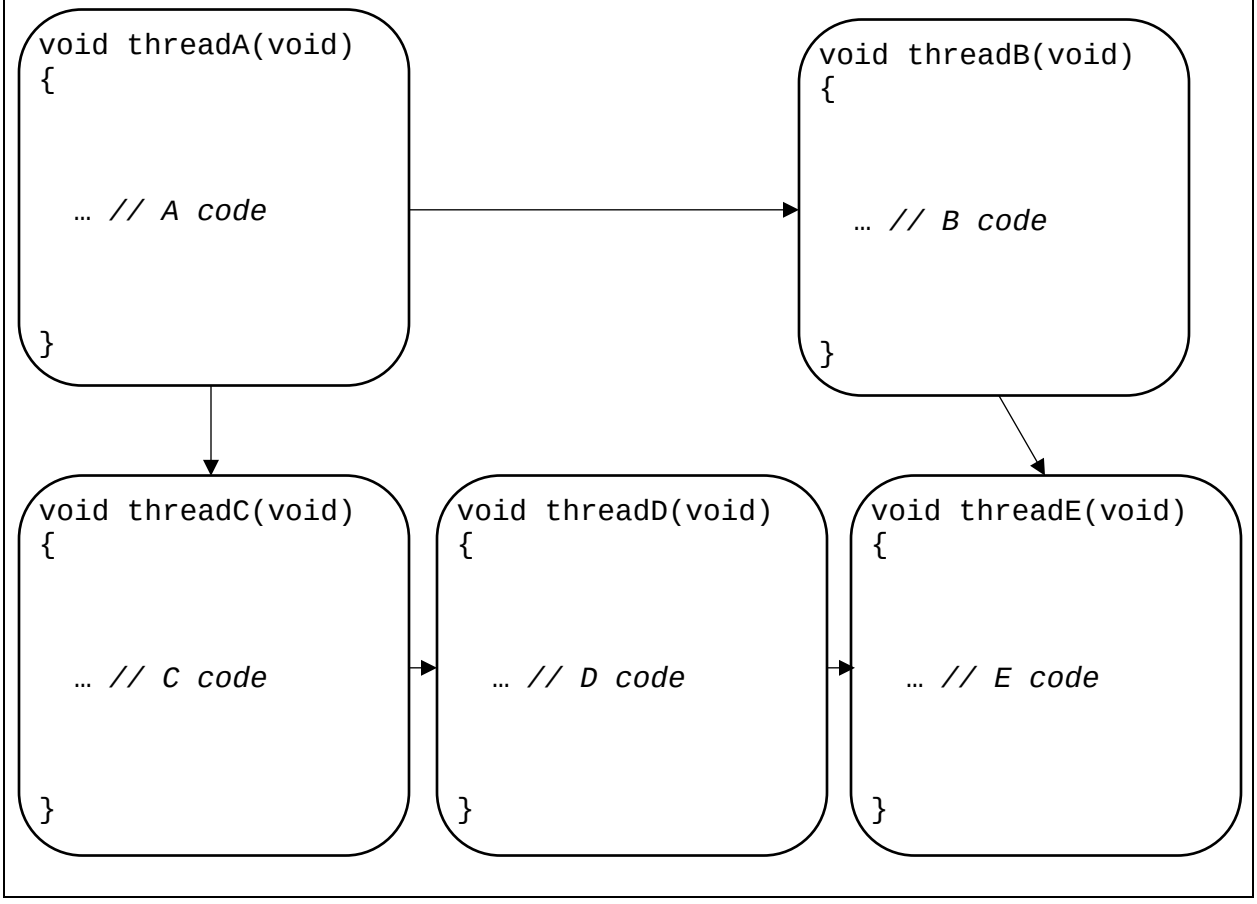
```
tcb_type *runPt = 0;
tcb_type tcb[3];
int numThreads = 0;

void AddDefault(void)
{
  tcb_type *t;

  OS_bWait(&runList_mutex);

  if(runPt) {
    return;
  }

  OS_bWait(&tcb_mutex);

  t = &tcb[numThreads++];

  OS_bSignal(&tcb_mutex);

  t->next = 0;
  runPt = t;

  OS_bSignal(&runList_mutex);

}
```

```
void Add(void)
{
  tcb_type *t;

  OS_bWait(&tcb_mutex);

  if(numThreads < 3) {

    t = &tcb[numThreads++];

    OS_bWait(&runList_Mutex);

    t->next = runPt;
    runPt = t;

    OS_bSignal(&runList_Mutex);

  }

  OS_bSignal(&tcb_mutex);

}
```

**Problem 3 (15 points): Thread Synchronization**

In many applications, tasks or threads will have dependencies in the form of predecessor-successor relationships in which a task is only allowed to execute once all its predecessors have finished execution. Such dependency relationships can be expressed in the form of a so-called task graph. Using only semaphores and regular C statements/variables, complete the code below to implement the given, intended task graph and task dependencies.

```
// Global variables and initialization code
```




Does your solution use the smallest number of semaphores, or could the pattern be realized using fewer semaphores than you showed above? If so, how many are minimally needed?

**Problem 4 (20 points): Scheduling**

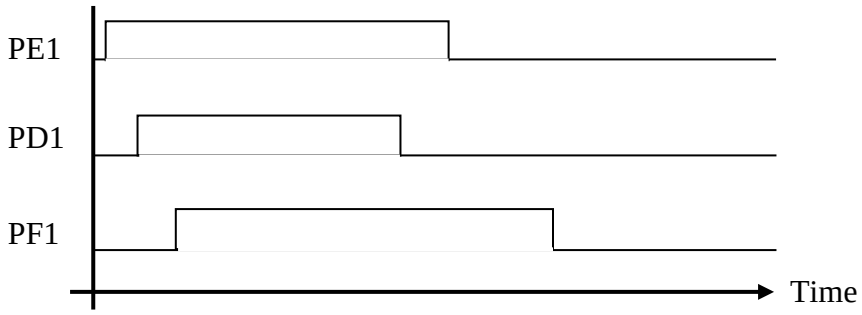
For each of the following cases, assume that the given threads were added before calling *OS_Launch*(). The OS scheduling policy is unknown. On launching the OS, the given profiling waveform is captured. What are all the possible scheduling policies that the OS could be using? For each case, identify the following based on the given profiling information: (1) whether a round-robin and/or priority scheduling could have resulted in the given waveform, and (2) the relative thread priorities, if any. Assume that thread execution times are all longer than one time slice. Furthermore, assume that all semaphores are initialized to one.

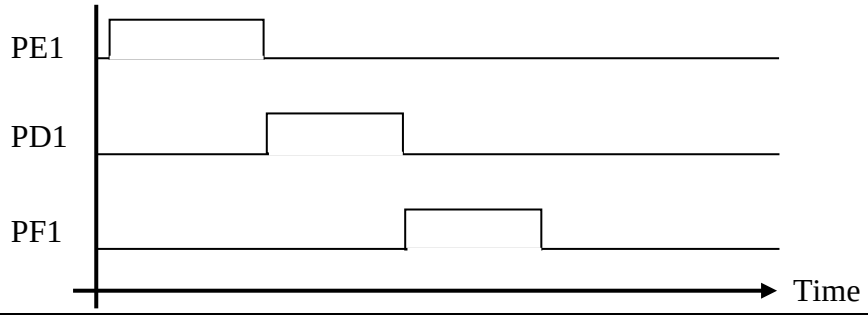
a)

```
void thread1a() {
   PE1 = 0x02;
   foo1a();
   PE1 = ~0x02;
   OS_Kill();
}
```

```
void thread2a(){
  PD1 = 0x02;
  foo2a();
  PD1 = ~0x02;
  OS_Kill();
}
```

```
void thread3a(){
  PF1 = 0x02;
  foo3a();
  PF1 = ~0x02:
  OS_Kill();
}
```



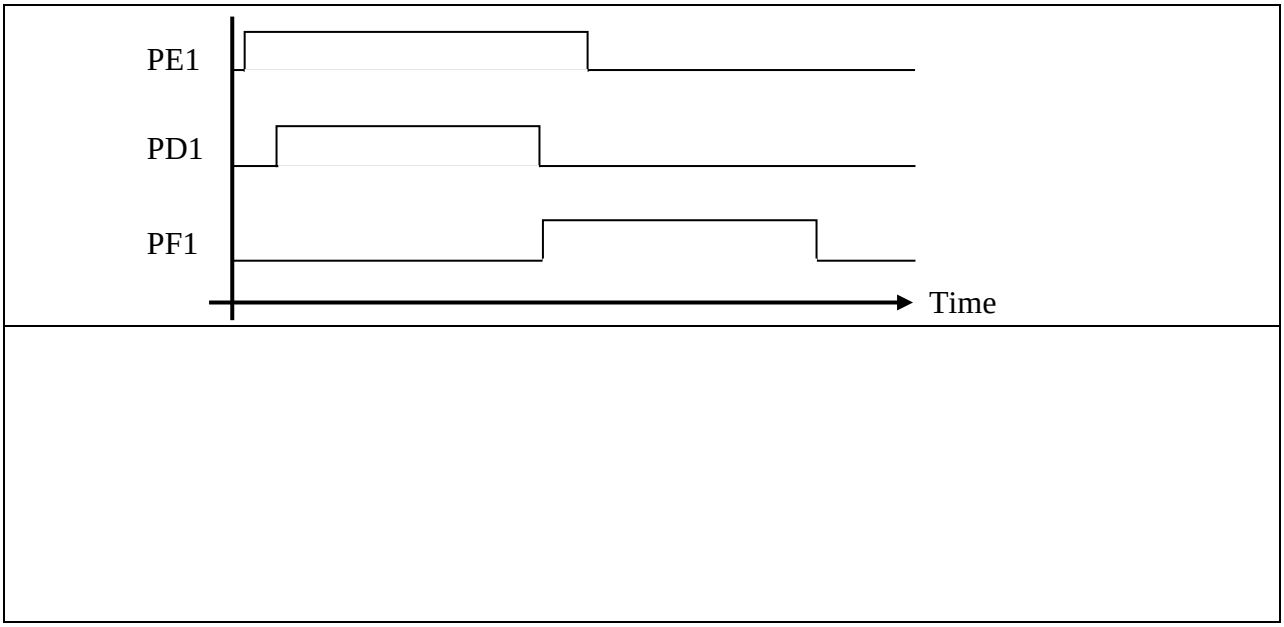



b)

```
void thread1b() {
   PE1 = 0x02;
   foo1b();
   OS_Sleep(3);
   PE1 = ~0x02;
   OS_Kill();
}
```

```
void thread2b(){
   PD1 = 0x02;
   foo2b();
   PD1 = ~0x02;
   OS_Kill();
}
```

```
void thread3b(){
  PF1 = 0x02;
  foo3b();
  PF1 = ~0x02:
  OS_Kill();
}
```

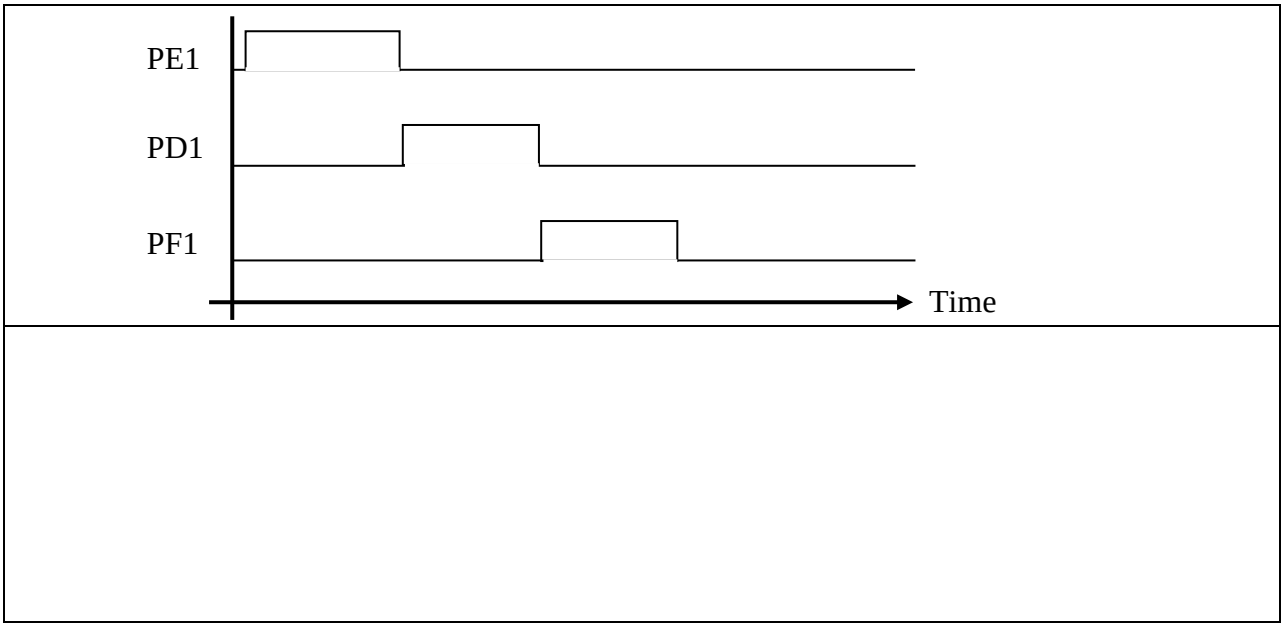



c)

```
void thread1c() {
  OS_bWait(&mutex);
  PE1 = 0x02;
  foo1c();
  PE1 = ~0x02;
  OS_bSignal(&mutex);
  OS_Kill();
}
```

```
void thread2c() {
   OS_bWait(&mutex);
   PD1 = 0x02;
   foo2c();
   PD1 = ~0x02;
   OS_bSignal(&mutex);
   OS_Kill();
}
```

```
void thread3c() {
  OS_bWait(&mutex);
  PF1 = 0x02;
  foo3c();
  PF1 = ~0x02;
  OS_bSignal(&mutex);
  OS_Kill();
}
```



**Problem 5 (20 points): Miscellaneous**

a) What are the tradeoffs in setting the time slice your OS runs on? What are the effects of having a very long (say 1s) or a very short (say 1ns) time slices vs. the default 1-10ms?

b) Is there a need for a SysTick interrupt in a purely priority scheduled OS? If not, why not? If so, under what conditions and/or for what functionality is a SysTick absolutely needed?

c) Assume you have a critical section that is protected by a spin-lock semaphore implemented using disabling/enabling of interrupts internally. Isn't this equivalent to just removing the semaphore and ensuring mutual exclusion by disabling/enabling interrupts for the critical section instead? What is the difference? Under what conditions is using semaphores better than using disabling/enabling of interrupts for mutual exclusion, and vice versa?

d) In class we discussed semaphores using a priority ceiling protocol to avoid priority inversion problems. Aren't such semaphores equivalent to just enabling/disabling of interrupts? What is the difference?