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 Solutions

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?

The implementation never switches the stack and hence context between threads.

While the runPt will be updated, the OS will always continue to run the same thread.

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.

One possible solution:
Save the stack pointer at the very top of the stack space, effectively reducing the available stack size by one word (since the stack grows from the bottom). In assembly:

```
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
  STR  SP,[R1,#4]; Save SP in stack[0]
  LDR  R1,[R1]   ; R1 = RunPt->next
  STR  R1,[R0]   ; RunPt = R1
  ~~POP  {SP}      ; Restore SP~~
  LDR  SP,[R1,#4]; Restore SP from stack[0]
  POP  {R4-R11}  ; Restore R4 - R11
  CPSIE I        ; Enable interrupts
  BX   LR     ; Restores R0-R3,R12,LR,PC,PSR
```

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;


}
```

Only read accesses to the shared variable, so no race condition, i.e. function is re-entrant.
No sempahores, so no deadlocks.

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 };

sema_t s = 1;

void set_time(unsigned int h, unsigned int m)
{
  OS_bWait(&s);

  total_time.m = m;

  total_time.h = h;

  OS_bSignal(&s);
}
```

Multi-step write to a shared data structure has race condition leading to inconsistencies in the total_time when called by multiple threads, i.e. function is not re-entrant. Critical section comprised of write access sequence needs to be protected by semaphores. Only one semaphore, so no deadlock.

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 };

sema_t s = 1;

void set_time(unsigned int h, unsigned int m)
{
  OS_bWait(&s);

  total_time.m = m;

  total_time.h = h;

  OS_bSignal(&s);
}


unsigned long get_seconds(void)
{
   unsigned long t;

   OS_bWait(&s);

   t = ( total_time.h * 60 + total_time.m ) * 60;

   OS_bSignal(&s);

   return t;
}
```

In addition to the race condition and critical section within the non-reentrant set_time() function, there is now also a race condition and critical section between multi-step writes and reads in set_time() and get_seconds(). E.g. if get_seconds() is preempted and set_time() is called in between the reading of total_time.h and total_time.m, get_seconds() will return an incorrect result.

Hence, both functions now need to be protected by the mutex semaphore to ensure mutual exclusion of set_time() with itself and set_time() with get_seconds().As a side effect, get_seconds() will also be mutually exclusively with multiple calls of itself now.

*Name:*______________________________

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) {
    OS_bSignal(&runList_mutex);
    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(&runList_mutex);
  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);
 OS_bSignal(&runList_mutex);
}
```

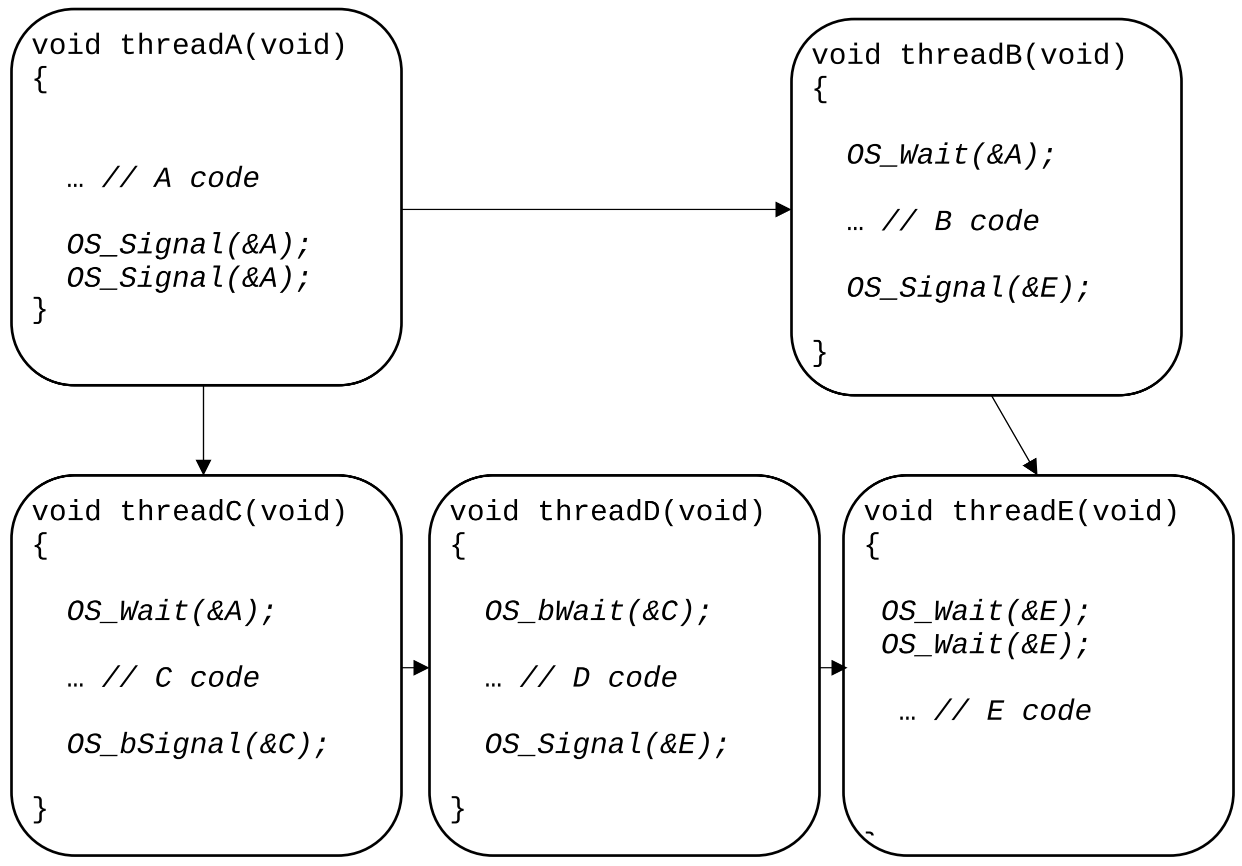
Two issues:
1) runList_mutex needs to be released when returning from AddDefault() early.
2)Potential circular hold and wait, i.e. deadlock between AddDefault() and Add(). Need to acquire semaphores in the same order in both functions. Two possible ways to do that:
(a) Change Add() to acquire runList_mutex before tcb_mutex (solution outlined above), or
(b) Change AddDefault() to acquire tcb_mutex before runList_mutex. This requires both mutexes to be released when returning early, though!
Both of these solutions have the downside of unnecessarily locking semaphores in cases when the if() statements succeed (and hence locking would have not been necessary). But no other way to solve.

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

sema_t A = 0;
sema_t C = 0;
sema_t E = 0;
```



```
void threadA(void)
{


  … // A code

  OS_Signal(&A);
  OS_Signal(&A);
}
```

```
void threadB(void)
{

  OS_Wait(&A);

  … // B code

  OS_Signal(&E);

}
```

```
void threadC(void)
{

  OS_Wait(&A);

  … // C code

  OS_bSignal(&C);

}
```

```
void threadD(void)
{

  OS_bWait(&C);

  … // D code

  OS_Signal(&E);

}
```

```
void threadE(void)
{

 OS_Wait(&E);
 OS_Wait(&E);

  … // E 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?

General solution uses one binary semaphore per dependency arc, i.e. per pair of threads.

Above is shown solution with 3 semaphores, which is minimum. However, this requires A and E semaphores used above for realizing fork and join dependencies to be counting semaphores.

Problem 4 (20 points): Scheduling

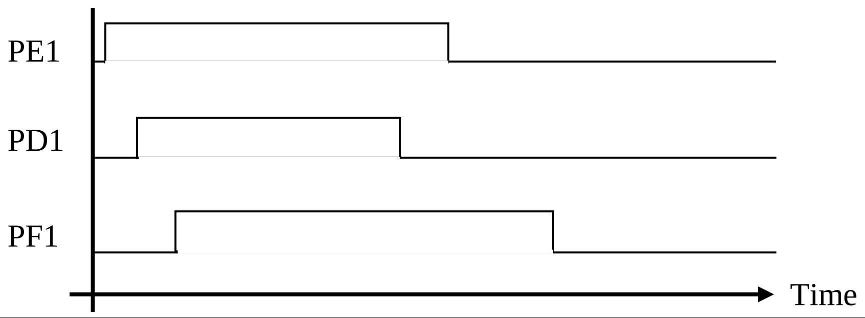
For each of the following cases, assume that the given threads were added before calling *OS_Launch*(). The OS scheduling policy is unknown. On launching the OS, the given profiling waveform is captured. What are all the possible scheduling policies that the OS could be using? For each case, identify 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)

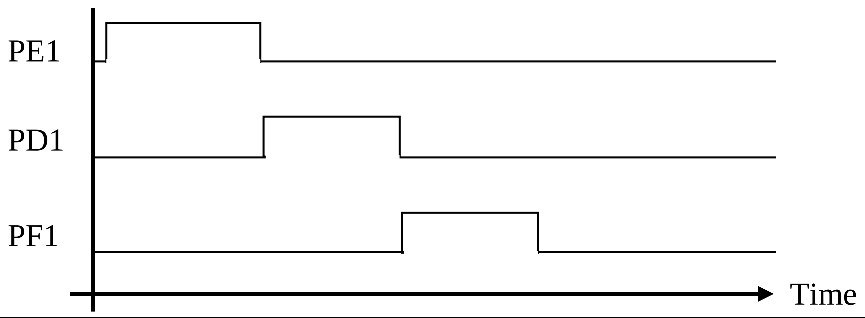
```
void thread1a() {
   PE1 = 0x02;
   foo1a();
   PE1 = ~0x02;
   OS_Kill();
}
```

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

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



All threads run simultaneously, i.e. this must be a round-robin scheduler.



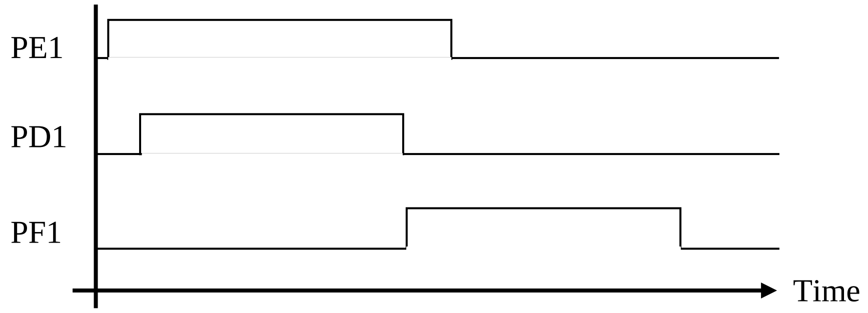
Threads never run at the same time, i.e. must be a priority scheduler.

Priorities: Thread1a > Thread2a > Thread3a

*Name:*________________________________

b)

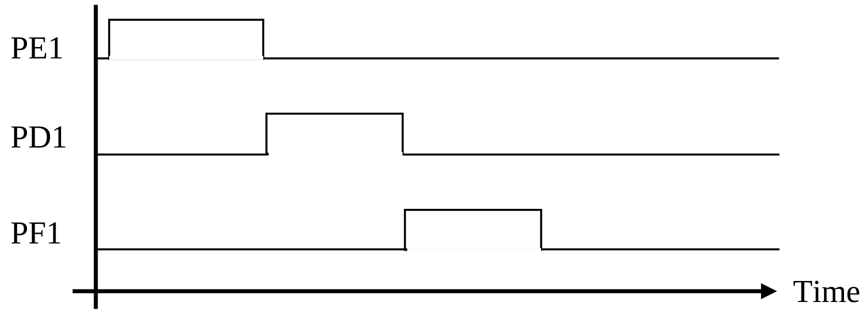
`void thread1b() {`
`   PE1 = 0x02;`
`   foo1b();`
`   OS_Sleep(3);`
`   PE1 = ~0x02;`
`   OS_Kill();`
`}`
	`void thread2b(){`
	`   PD1 = 0x02;`
	`   foo2b();`
	`   PD1 = ~0x02;`
	`   OS_Kill();`
	`}`
		`void thread3b(){`
		`  PF1 = 0x02;`
		`  foo3b();`
		`  PF1 = ~0x02:`
		`  OS_Kill();`
		`}`



Thread2b and Thread3b don't run at the same time, i.e. strictly by priority. Thread1b starts running first but then sleeps, so Thread2b can kick in.
=> Priority scheduler with priorities: Thread1b >= Thread2b > Thread3b

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();`
		`}`



Mutual exclusion between threads ensures by itself, i.e. independent of the scheduler that none of them run at the same time. So this can be either a round-robin or a priority scheduler.

In the round-robin case, under the assumption n of a thread order Thread1c -> Thread2c -> Thread3c

In the priority case, with priorities: Thread1c > Thread2c > Thread3c

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 slice vs. the default 1-10ms?

Tradeoff between frequent context switching overhead vs. fairness/reaction time in making progress in all threads. With a time slice of 1ns, the OS would do pretty much nothing but context switching. With a time slice of 1s, threads would be able to hold the CPU for a long time and only make very bursty progress or would not be able to react very quickly to events, i.e. the illusion of having parallelism would be gone.

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?

In a priority OS, context switches are only possible and necessary when there is some change in the state of threads in the system. Specifically, if the currently running thread kills itself, blocks or sleeps, or if a higher-priority thread is released or otherwise added to the system. However, any such change in OS state can only be triggered by a call to an OS routine, where the context switch can in turn be triggered from within each OS routine as necessary but without requiring Systick.

With one exception: Systick is needed for OS_Sleep() functionality to count down sleep timers of threads and add sleeping threads back (and then trigger a context switch if it is of higher priority than the currently running one) once their sleep timer expires.

Systick is also needed in case of round-robin scheduling of threads with the same priority.

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?

The difference is that interrupts are only disabled for a very brief period of time within the semaphore, but not for the whole duration of the critical section. As such, semaphores are preferred for long critical sections.

Conversely, using disabling/enabling of interrupts is better than using semaphores when the critical section is very short (as short or shorter than the critical section internal to the semaphore). In this case, using semaphores just adds extra unnecessary overhead.

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?

Priority ceiling raises the priority of a thread holding a semaphore to the highest level. This prevents a thread being in a critical section from being interrupted by any other thread. In that sense, it is equivalent to just disabling interrupts globally. However, the difference is that disabling interrupts also prevents any background thread/interrupt handler from running. By contrast, priority ceiling semaphores don't.
1