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

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.12, Fall 2020

Midterm Exam Solutions

Date: October 22, 2020

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 calculators or any electronic devices other than your laptop/PC (turn cell phones off).
- 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	10	
Problem 4	10	
Problem 5	10	
Problem 6	30	
Total	100	

*Name:*_______________________________

Problem 1 (20 points): Semaphores

a) Given the following function template, fill in the rest of the code to design a spinlock implementation of an *OS_Wait_Or* function that will wait for two semaphores and acquire the first one that becomes available. The function should return a pointer to the acquired semaphore. If both semaphores are available, the function can acquire and return any of the two.

```
Sema4Type* OS_Wait_Or(Sema4Type* semaA, Sema4Type* semaB) {
  Sema4Type *semaRes;



  DisableInterrupts();


  while ((*semaA) <= 0 && (*semaB) <= 0) {
    EnableInterrupts();
    DisableInterrupts();
  }
  if ((*semaA) > 0) {
   semaRes = semaA;
  } else {
   semaRes = semaB;
  }
  (*semaRes) = (*semaRes) – 1;



  EnableInterrupts();






  return semaRes;
}
```

*Name:*______________________________

b) Given the following program using *OS_Wait_Or*:

```
OS_InitSemaphore(&A,1);
OS_InitSemaphore(&B,1);
OS_InitSemaphore(&C,1);
OS_InitSemaphore(&D,1);
```

`TaskA(){`
` sema4* t1;`
` sema4* t2;`
` t1 = OS_Wait_OR(&A,&B);`
` t2 = OS_Wait_OR(&C,&D);`
` ...`
 	
` OS_Signal(t2);`
` OS_Signal(t1);`
`}`
	`TaskB(){`
	` sema4* t1;`
	` sema4* t2;`
	` t1 = OS_Wait_OR(&A,&C);`
	` t2 = OS_Wait_OR(&C,&D);`
	` ...`
	
	` OS_Signal(t2);`
	` OS_Signal(t1);`
	`}`
		`TaskC(){`
		` sema4* t1;`
		` sema4* t2;`
		` t1 = OS_Wait_OR(&A,&C);`
		` t2 = OS_Wait_OR(&B,&D);`
		` ...`
		
		` OS_Signal(t2);`
		` OS_Signal(t1);`
		`}`

Is there any possible interleaving that could cause deadlock? If yes, please show an example that could be a deadlock. If not, justify your answer.

No, there is no deadlock. The only circular hold-and-wait can be from tasks holding A, B and C in their first wait statements. But then there is always semaphore D available for at least one task to continue and break the hold.

Problem 2 (20 points): OS Core

Assume a basic priority-based OS kernel with the following *InitStack()* implementation used by *OS_AddThread()* and a given user code running on top of the OS.

OS code:

```
long* InitStack(long *sp,
          void (*entry)(void)
          uint32_t threadID)
{
 *(sp) = (long)0x01000000L;
 *(--sp) = (long)entry;
 *(--sp) = (long)~~0L;~~ entry;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)~~0L;~~ threadID;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 return sp;
}
```

```
int NumThreads = 0;

int OS_AddThread(void(*task)(int),
                uint32_t priority)
{
  struct TCB* NewPt;

  …  // allocate TCB and stack,
  …  // insert in list

  NewPt->priority = priority;
  NewPt->threadID = NumThreads++;
  NewPt->sp = InitStack(NewPt->sp,
                 Task,
                 NewPt->threadID);

  …

  return 1;
}
```

User code:

```
 ; int Dump[100];
Dump
 DSD 100
 ; int DmpCnt = 0;
DmpCnt
 DCD 0

 ; void Thread1(int i) {
Thread1
 ; Dump[DmpCnt++] = i;
 LDR R1,=DmpCnt
 LDR R2,[R1]
 LDR R3,=Dump
 STR R0,[R3,R2 LSL 2]
 ADD R2,R2,#1
 ; if(DmpCnt >= 100) DmpCnt = 0;
 CMP R2,#100
 BLT end
 MOV R2,#0
end
 STR R2,[R1]
 ; }
 BX LR
```

```
void Idle(int i) {
  // Do nothing
}

void main(void) {
 OS_Init();
 OS_AddThread(&Thread1, 0);
 OS_AddThread(&Idle, 1);
 OS_Launch(); // does not return
}
```

a) What will be the behavior of this program and what values will be stored in the Dump array when run on the given OS kernel?

The program will store one value of 0 into Dump[0] and then crash.

The stack is initialized to pass value 0 via R0 into the argument of each thread. In addition, the LR register of each thread (and in fact all registers) are initialized to zero.

In this user program, Thread1 has highest priority and runs first. It will put it's argument into Dump[0] and then try to exit via BX LR. This will branch to address 0, which will trigger a hard fault.

b) Show the modifications necessary in the OS code to: (1) pass the thread ID into each thread as first parameter, and (2) execute all threads in an endless loop. You don't need to list the full code again, just indicate the lines that need to be changed or added/removed in the OS code listing above. You are not allowed to make any changes to the user code, only the OS kernel.

(1) Need to pass the thread ID into threads via register R0, and

(2) Need to setup the LR register of each thread to point back to the thread entry, such that the BX LR at the end loops back to the thread's first instruction.

Both is accomplished by passing the threadID into InitStack and then setting up the initial registers accordingly. See modifications indicated in code above.

Problem 3 (10 points): Race Conditions and Reentrance

a) Does the user program as given in Problem 2 above have any race condition? Why or why not? If not, how can the race condition be resolved?

The user program has global variables that are accessed by Thread1 in a Read-Modify-Write sequence. However, since there is only one Thread1 accessing the globals, there is no race condition.

b) Is the *Thread1()* function in the user program from Problem 2 reentrant? Why or why not? If not, how can it be made reentrant?

The Thread1 function is not re-entrant. If it were executed by two threads (e.g. by launching and adding Thread1 twice via OS_AddThread), there would be a race condition accessing the global Dump and DumpCnt variables, i.e. Thread1 has a critical section with itself.

To make Thread1 re-entrant, the critical section has to be resolved by making the Read-Modify-Write and Write sequences to DumpCnt and Dump mutually exclusive, e.g. by adding a mutex semaphore around the code:

```
Sema4type mutex = 0;

 void Thread1(int i) {
  OS_bWait(&mutex);
  Dump[DmpCnt++] = i;
  if(DmpCnt >= 100) DmpCnt = 0;
  OS_bSignal(&mutex)
}
```

Problem 4 (10 points): Background Threads

Given an OS that uses PendSV for interrupt-based context switching where PendSV has the lowest interrupt priority and the following *OS_Sleep* implementation as well as user program:

OS_Sleep code:

```
void OS_Sleep(unit32_t sleep)
{
  // Mark thread as sleeping
  RunPt->sleepTime = sleep;
  // And switch to next thread
  OS_Suspend();
}

void OS_Suspend(void)
{
  ContextSwitch();
}

void ContextSwitch(void) {
  // Trigger PendSV
  NVIC_INT_CTRL_R=0x10000000;
}
```

User code:

```
void ThreadA(void) {
  while(1) { OS_Wait(&Sema); }
}
void ThreadB(void) {
  while(1) { printf("Hello\n"); }
}
void BGThreadC() {
  OS_Signal(&Sema);
  OS_Sleep(5);
}

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

void main(void) {
 OS_Init();
 OS_AddPeriodicThread(&BGThreadC,
                     TIME_10MS, 0);
 OS_AddThread(&ThreadA, 0);
 OS_AddThread(&ThreadB, 1);
 OS_AddThread(&Idle, 2);
 OS_Launch(); // does not return
}
```

a) What will happen when *OS_Signal* is called in the background thread *BGThreadC*? Is there any issue? If so, explain the issue and behavior. If not, why is there no issue?

There is no issue with calling OS_Signal from a background thread. All OS_Signal does is increment the semaphore and if there is a waiting thread, unblock it, and potentially trigger a context switch if the thread being woken up has a higher priority than the currently running foreground thread. Since PendSV has a lower priority than the background interrupt, the PendSV context switch will be executed as soon as the background interrupt handler exits.

*Name:*______________________________

b) What will happen when *OS_Sleep* is called in the background thread *BGThreadC*? Is there any issue? If so, explain the issue and behavior. If not, why is there no issue?

If OS_Sleep is called from a background thread, it will not actually sleep the background thread but result in the sleep being effectively handled/assigned as if it were called from the currently running foreground thread.

When a background thread is running, RunPt points to the foreground thread that was running when the background timer interrupt got triggered. As such, OS_Sleep will mark that foreground thread as sleeping and will then trigger a context switch. In the same way as in OS_Signal, the context switch will be executed as soon as the background interrupt handler exits. The end result is that the foreground thread that was running will be suspended and marked as sleeping until it gets woken up again. In other words, the foreground thread ends up sleeping, not the background thread.

Problem 5 (10 points): Miscellaneous

a) Assume two periodic real-time task sets, one with 3 tasks and 50% CPU utilization, and the other with 5 tasks and 85% CPU utilization. Are these task sets schedulable under an RMS or EDF strategy? Why or why not?

An EDF scheduler is able to schedule both task sets.

An RMS scheduler is guaranteed to be able to schedule the first task set, but for the second task set it will depend on the specific periods and deadlines of each task (an analysis of the critical instant will need to be performed to determine whether the task set is schedulable).

b) What is the role of the time slice in a priority scheduled OS? What effect will changing the time slice have on system operation?

The time slice does not have any scheduling effect in a strict priority scheduled OS as there is no need for regular, timer-based thread preemption. The only time a thread switch can occur and a running thread hence needs to be preempted is if the state of the system changes in any way, i.e. a higher priority thread enters or wakes up. However, all the system state changes are under the control of the OS and it can trigger context switches as needed.

The only aspect the time slice may influence is the granularity of the system tick used for counting sleep or wait times. I.e. it may influence the granularity of possible sleep times. And if context switches are triggered every time, a finer slice leads to more overhead.

Plus, round-robin scheduling of threads with the same priority.

Problem 6 (30 points): Scheduling

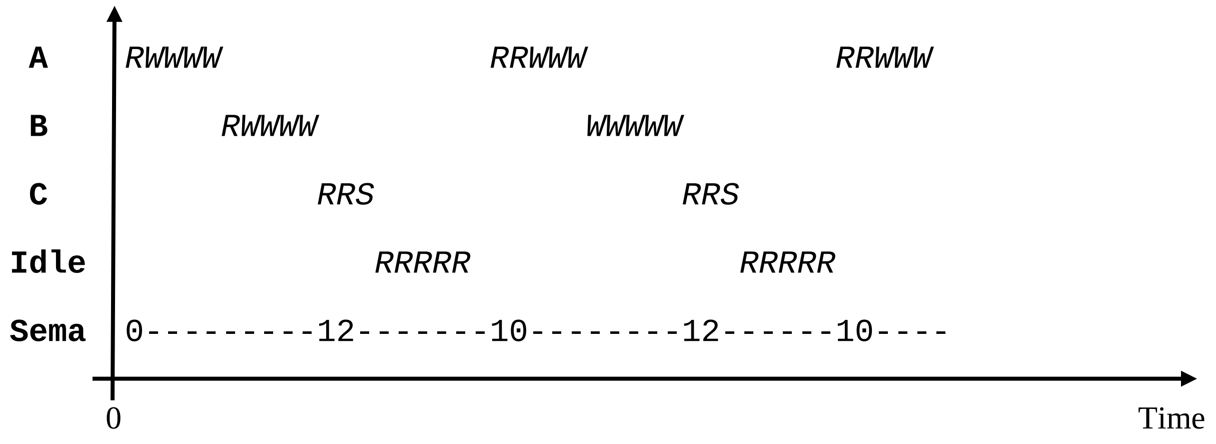
Suppose you have the following three threads in your system. You can assume that the semaphore is initialized to zero and that there is another `Idle` thread in the OS that does nothing (executes a `while(1){}` loop) and runs last in the sequence or with lowest priority. For each of the OS implementations below, show the order and sequence of events and thread executions until the behavior repeats. Indicate any changes in the value of the semaphore `Sema` and threads changing between active/ready (A), running (R), waiting (W) and sleeping (S) states (indicate spinning as waiting). The time slice is 5ms and the first thread starts execution at time 0.

```
ThreadA(){
  while(1) {
    PD0 ^= 0x01;
    OS_Wait(&Sema);
    PD0 ^= 0x01;
  }
}
```

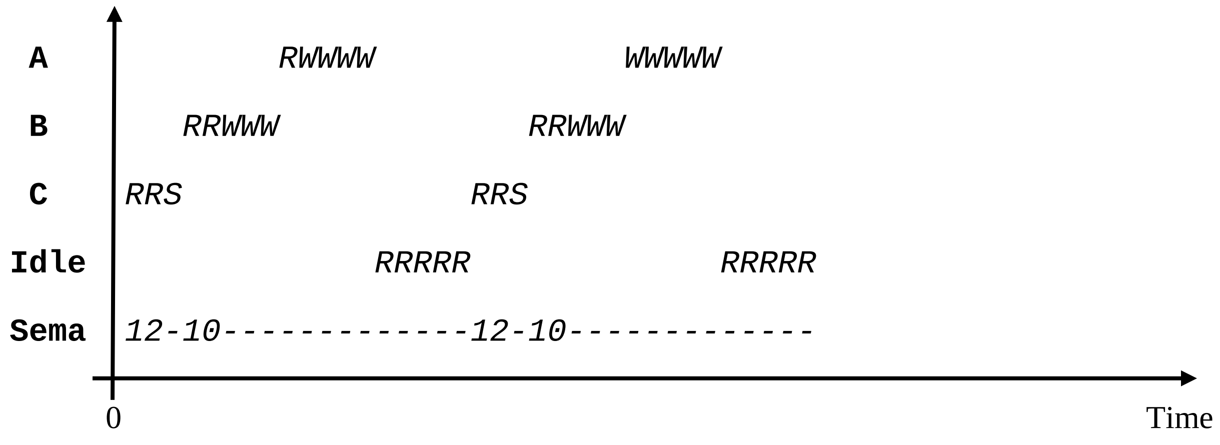
```
ThreadB(){
  while(1) {
    PD1 ^= 0x02;
    OS_Wait(&Sema);
    PD1 ^= 0x02;
  }
}
```

```
ThreadC() {
  while(1) {
    OS_Signal(&Sema);
    OS_Signal(&Sema);
    OS_Sleep(5);
  }
}
```

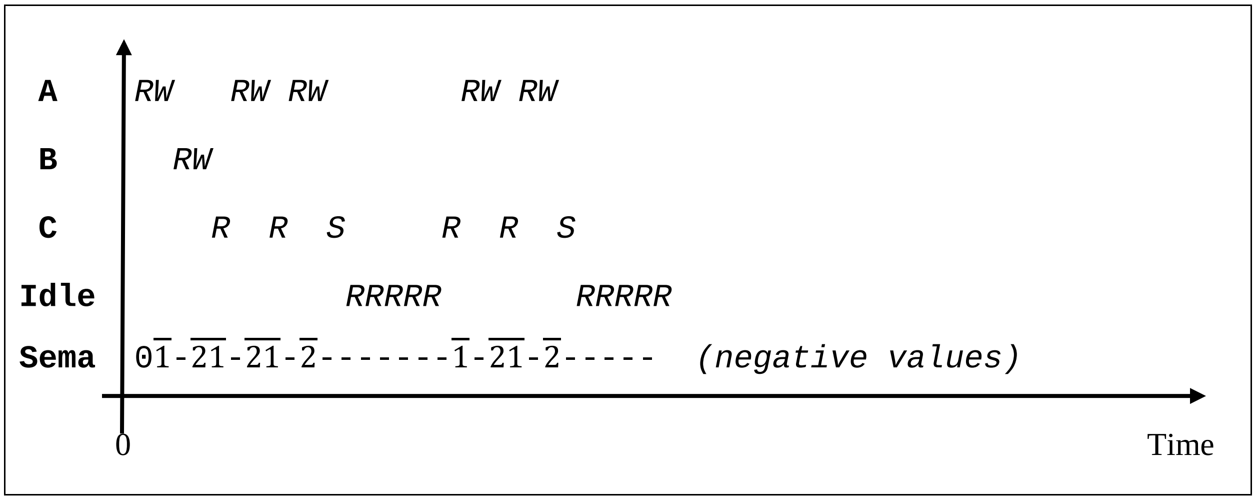
a) OS using a round-robin scheduler with spinlock semaphores and threads running in the round-robin order of *ThreadA-ThreadB-ThreadC-Idle*.



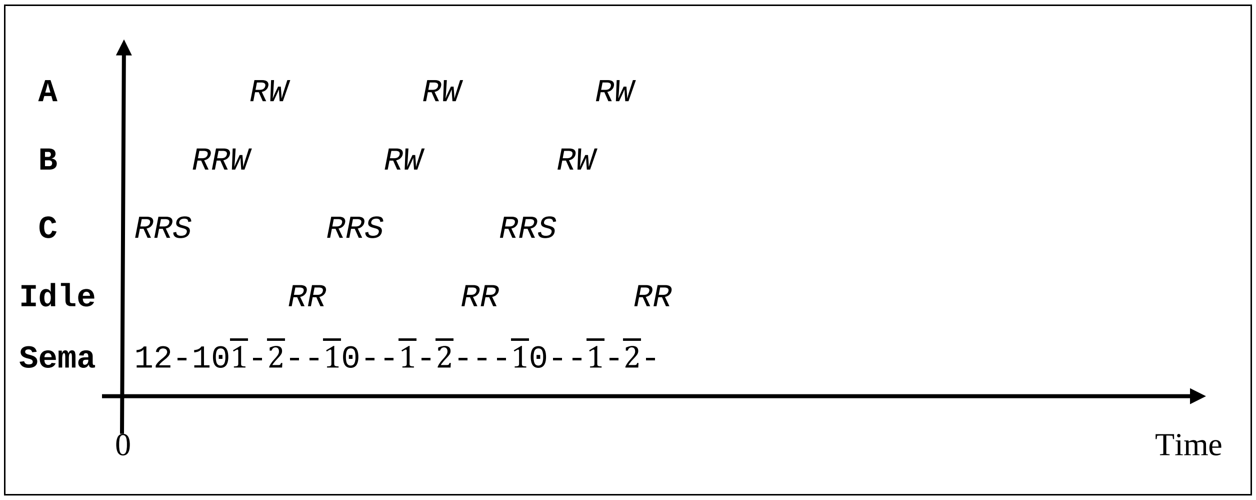
b) OS using a round-robin scheduler with spinlock semaphores and threads running in the round-robin order of *ThreadC-ThreadB-ThreadA-Idle*.



c) OS suing a priority scheduler with blocking semaphores and thread priorities of (highest to lowest) *ThreadA-ThreadB-ThreadC-Idle*.


A RW RW RW RW RW
B RW
C R R S R R S
Idle RRRRR RRRRR
Sema 01̅-2̅1̅-2̅1̅-2̅-------1̅-2̅1̅-2̅----- (negative values)
0
Time

d) OS using a priority scheduler with blocking semaphores and thread priorities of (highest to lowest) *ThreadC-ThreadB-ThreadA-Idle*.


A RW RW RW
B RRW RW RW
C RRS RRS RRS
Idle RR RR RR
Sema 12-101̅-2̅--1̅0--1̅-2̅---1̅0--1̅-2̅-
0
Time