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

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	

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();















  EnableInterrupts();






  return semaRes;
}
```

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.

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))
{
 *(sp) = (long)0x01000000L;
 *(--sp) = (long)entry;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--sp) = (long)0L;
 *(--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);

  …

  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,=Dmp
 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?

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.

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?

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?

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?

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?

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?

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?

*Name:*______________________________

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);
  }
}
```

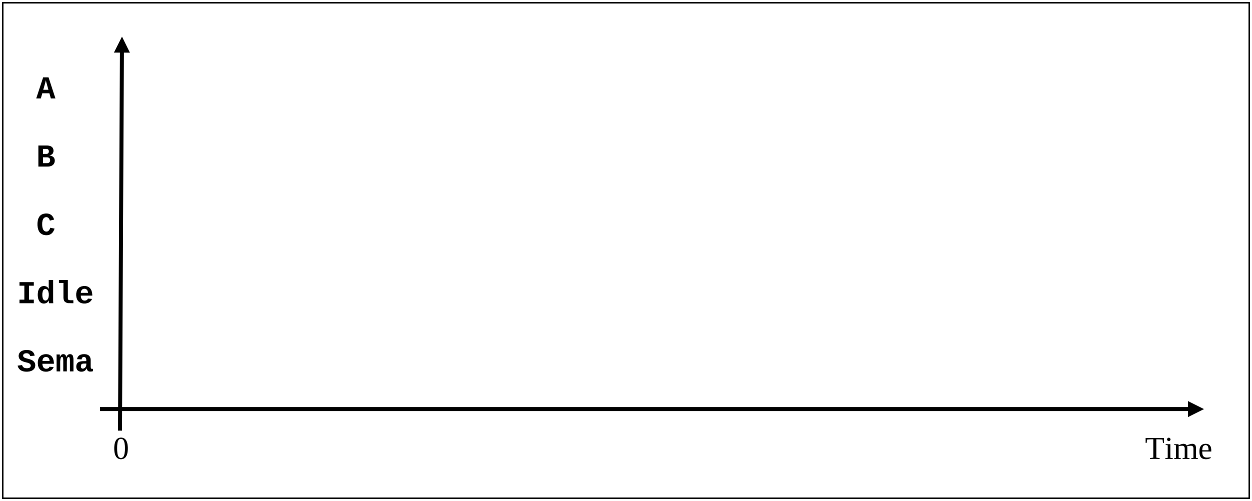
a) OS suing a round-robin scheduler with spinlock semaphores and threads running in the round-robin order of *ThreadA-ThreadB-ThreadC-Idle*.

A
B
C
Idle
Sema
0
Time

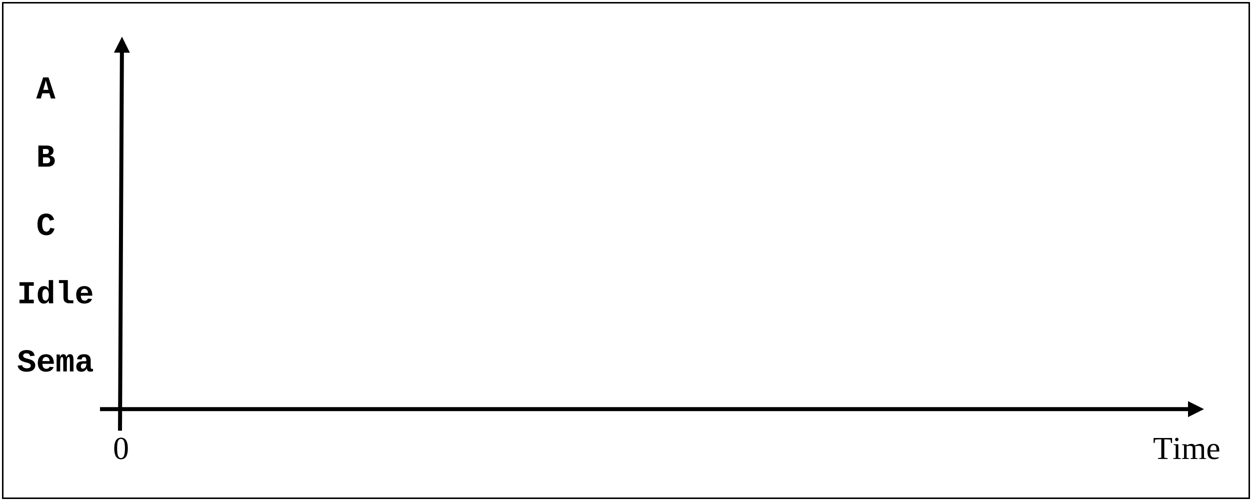
b) OS using a round-robin scheduler with spinlock semaphores and threads running in the round-robin order of *ThreadC-ThreadB-ThreadA-Idle*.

A
B
C
Idle
Sema
0
Time

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



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


A
B
C
Idle
Sema
0
Time