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

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.12, Spring 2018

Midterm

Date: March 20, 2018

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

Problem 1 (25 points): Critical Sections, Reentrance and Deadlock

Given the following implementation of a simple heap that allocates blocks of fixed 1kB size:

```
long Heap[HEAP_BLKS][256];


void Heap_Init(void) {
  unsigned i;

  for(i = 0; i < HEAP_BLKS; i++) {

    Heap[i][0] = -1;

  }

}

void Heap_Free(long* blk) {

  blk[-1] = -1;

}
```

```
long* Heap_Alloc(void) {
  unsigned i;

  for(i = 0; i < HEAP_BLKS; i++) {

    if(Heap[i][0] < 0) break;

  }

  if(i == HEAP_BLKS) {

    return 0; // heap full

  }

  Heap[i][0] = 0x0000DEAD;

  return &Heap[i][1];
}
```

a) Is this code thread-safe, reentrant and deadlock-free? Point out all and any critical sections, reentrance issues or deadlocks

b) Fix the heap by showing the necessary modifications (insertions, deletions, replacements) of the code above to make it thread-safe, reentrant and deadlock-free. Your solution should introduce the least jitter using standard binary semaphores:

```
void OS_InitSemaphore(sema4type *semaPt, long value);
void OS_bWait(sema4type *semaPt);
void OS_bSignal(sema4type *semaPt);
```

c) Given the code for a priority scheduler that uses the corrected heap from b) with blocking semaphores, where the priority queue management (*Q_xxx()*) functions themselves are not thread-safe/reentrant, does this code have any critical sections, non-reentrance or deadlocks? List all issues and indicate any code changes necessary to fix the code with minimal intrusion and jitter. Is a solution purely using semaphores possible? If yes, show it. If not, why not?

```
struct tcb {
  long *sp;
  struct TCB *next;
  unsigned priority;
  long stack[128];
}

struct TCB *RunPt; // current
struct TCB *ReadyQ = 0;

int OS_AddThread(void
 (*task)(void), unsigned prio) {
  struct TCB *new;

  DisableInterrupts();

  new = (struct TCB*)Heap_Alloc();
  if(!new) {
    return 0;
  }

  new->sp = InitStack(
             &new->stack[127],
             task);
  new->priority = prio;

  Q_Insert(&ReadyQ, new);

  EnableInterrupts();

  return 1;
}
```

```
void OS_Kill(void) {

  DisableInterrupts();

  Q_Remove(&ReadyQ, RunPt);

  Heap_Free(RunPt);

  ContextSwitch(); // trig. PendSV

  EnableInterrupts();

}
void Systick_Handler(void) {

  if(ReadyQ != RunPt) {

    ContextSwitch(); // PendSV
    // switches to ReadyQ

  }

}
```

Problem 2 (15 points): OS Core Functionality

Assume a basic priority-based OS kernel (as shown in Problem 1) with the following *InitStack()* implementation and given user code:

```
long* InitStack(long *sp, void (*entry)(void)) {
  *(sp)   = (long)0x01000000L;  /* xPSR */
  *(--sp) = (long)entry;        /* PC  */
  *(--sp) = (long)entry;        /* R14 */
  *(--sp) = (long)0x12121212L;  /* R12 */
  *(--sp) = (long)0x03030303L;  /* R3  */
  *(--sp) = (long)0x02020202L;  /* R2  */
  *(--sp) = (long)0x01010101L;  /* R1  */
  *(--sp) = (long)0x000000000;  /* R0  */
  *(--sp) = (long)0x11111111L;  /* R11 */
  *(--sp) = (long)0x10101010L;  /* R10 */
  *(--sp) = (long)0x09090909L;  /* R9  */
  *(--sp) = (long)0x08080808L;  /* R8  */
  *(--sp) = (long)0x07070707L;  /* R7  */
  *(--sp) = (long)0x06060606L;  /* R6  */
  *(--sp) = (long)0x05050505L;  /* R5  */
  *(--sp) = (long)0x04040404L;  /* R4  */
  return sp;
}
```

```
void Thread1(void) {
  static int i = 0;
  ST7735_Message(0, 0,
           "Hello world", i++);
}
```

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

a) What will be the behavior of this program, i.e. what is the sequence of events in the system and what messages will appear on the ST7735 display, if any?

b) Describe minimal modifications of the OS to allow parameters being passed to threads via:
`OS_AddThread(void (*task)(int), unsigned prio, int param)`
as shown in the example below.

```
void Thread1(int i) {
  ST7735_Message(0, 0,
            "Hello world", i);
} // should print "Hello world: 42"
```

```
void main(void) {
  OS_Init();
  OS_AddThread(Thread1, 0, 42);
  OS_Launch(); // does not return
}
```

Problem 3 (25 points): Periodic Real-Time Tasks

As discussed in class, many real-time systems use a periodic task model in which foreground user tasks (such as main control algorithms) are expected to execute periodically with a well-defined and fixed (e.g. based on control theory) rate (and corresponding deadline).

a) Develop an implementation of a periodic *PID* task that executes with a fixed period *P* (in ms). You can assume that the following typical OS time management functions are available:

```
unsigned long OS_MsTime(void); // system time in ms
void OS_Sleep(unsigned long sleepTime);  // in ms
```

Bonus points for an implementation that maximizes accuracy and minimizes jitter.

```
void PID(void) {


  while(1) {



    PIDWork();




  }

}
```

b) Many RTOSs will provide a native realization of periodic (foreground) tasks. List the changes needed to extend a typical OS (such as the one from Problems 1-2) to provide periodic tasks:

```
OS_AddPeriodicForegroundThread(void (*task)(void), …,
                                        unsigned long period);
```

as shown below. Sketch (only) the modifications/extensions of basic OS routines as needed.

```
// run every 10ms
void PID(void) {
  PID_Work();
}
```

```
void main(void) {
  OS_Init();
  OS_AddPeriodicForegroungThread(PID, 5, 10);
  OS_Launch(); // does not return
}
```

c) Assuming your priority-scheduled system is running a sporadic/aperiodic foreground task T_i with execution time E_i at each priority level $0 \leq i < N$, what is the maximum jitter experienced by a periodic foreground task T_m at priority m? Assume that there are no background threads.

Problem 4 (15 points): Semaphores

Given the following spin-lock semaphore implementation:

```
struct Sema4 {
     long value;

};
typedef struct Sema4 Sema4Type;

void OS_InitSemaphore(
 Sema4Type *semaPt, int value)
{
  int sr;

  sr = StartCritical();

  semaPt->value = value;

  EndCritical(sr);

}
```

```
void OS_Wait(Sema4Type *semaPt) {
  int sr;

  sr = StartCritical();

  while(*semaPt <= 0){

     EndCritical(sr);

     sr = StartCritical();
  }

  (*semaPt)--;

  EndCritical(sr);

}

void OS_bSignal(Sema4Type *semaPt) {
  int sr;

  sr = StartCritical();


  semaPt->value = 1;


  EndCritical(sr);


}
```

a) This code has bugs. List and explain all the issues in the code, and describe the minimal changes required to fix each issue (you don't need to show modifications in the code above).

b) Modify the semaphore implementation such that it can avoid deadlocks if, by bad design, one of the threads happens to call *OS_bWait()* twice, i.e. calls *OS_bWait()* a second time on the same semaphore while it is already holding that semaphore as show in the example case below. Show the minimal necessary modifications (insertions, deletions, replacements) of the semaphore code such that this example executes without deadlocking (while preserving intended semaphore semantics).

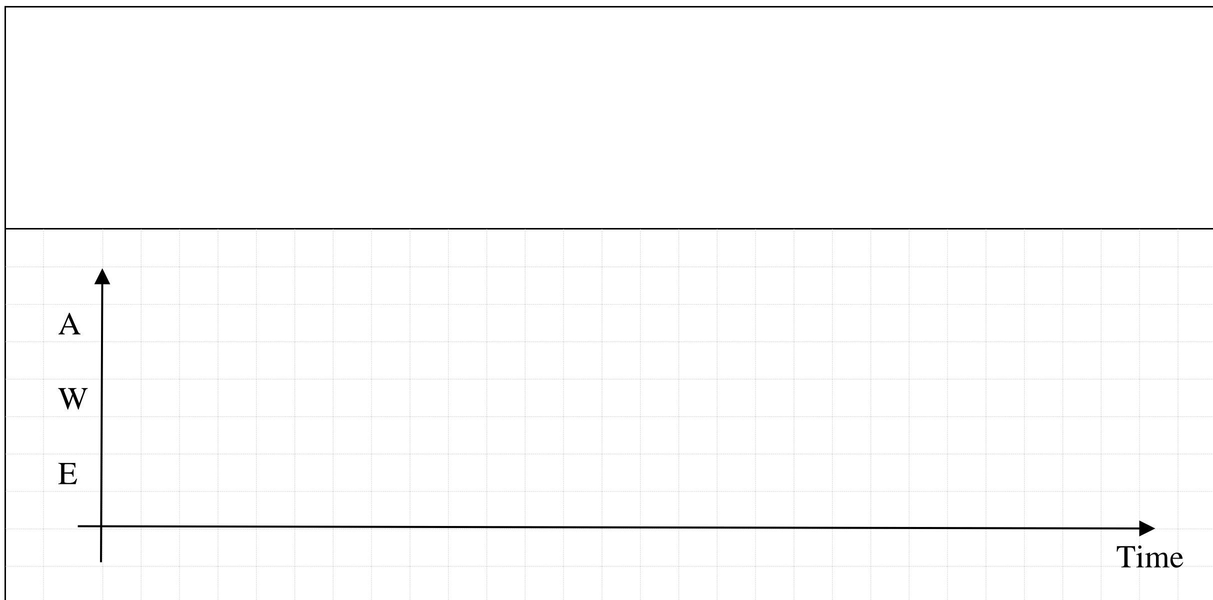
```
Sema4Type display_mutex;

void Display_DrawLine(int x1, int y1, int x2, int y2) {
  OS_bWait(&display_mutex);
  …
  OS_bSignal(&display_mutex);
}

void Display_DrawLogo(void) {
   OS_bWait(&display_mutex);
   …
   Display_DrawLine(0, 0, 144, 42);
   …
   OS_bSignal(&mutex);
}

void Thread1(void) {
  …
  Display_DrawLogo();
  …
}
```

Problem 5 (20 points): Real-Time Scheduling

Consider a real-time system running three periodic tasks with the following periods (= deadlines) and execution times. You can assume zero context switch and interrupt overhead.

Task	Execution Time	Period
Airbag (A)	1ms	3ms
Warning (W)	1ms	6ms
Engine (E)	*e*	4ms

a) What is the maximum execution time *e* of taks E such that this task set is schedulable using an RMS strategy without missing deadlines? What is the processor utilization in this case? Draw one iteration of the resulting schedule.



b) What is the maximum execution time *e* of taks E such that this task set is schedulable using an EDF strategy without missing deadlines? What is the processor utilization in this case? Draw one iteration of the resulting schedule.