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];

Sema4Type Heap_Mutex;

void Heap_Init(void) {
  unsigned i;

  OS_InitSemaphore(&Heap_Mutex,1);

  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;

  OS_bWait(&Heap_Mutex);

  for(i = 0; i < HEAP_BLKS; i++) {
    if(Heap[i][0] < 0) break;
  }

  if(i == HEAP_BLKS) {
    OS_bSignal(&Heap_Mutex);

    return 0; // heap full
  }

  Heap[i][0] = 0x0000DEAD;

  OS_bSignal(&Heap_Mutex);

  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

Heap_Alloc has RMW sequence on heap blocks (element 0), not reentrant.

Technically, there is also potential race condition between Heap_Init (W) or Heap_Free (W) and Heap_Alloc (RMW). However, since the MW sequence in Heap_Alloc only happens if the R was negative, and since the other routines can only set to negative, there is no actual race condition. To be on the safe side, it is ok to use mutex around writes in Heap_Init/Free.

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;

  DisableInterrupts();

  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) {

  opt.: DisableInterrupts();

  if(ReadyQ != RunPt) {

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

  }

  opt.: EnableInterrupts();
}
```

No reentrance issues since OS_AddThread and OS_Kill disable interrupts (and SysTick can never reenter itself).

Deadlock or critical section: if another thread is preempted while holding Heap_Mutex and OS_AddThread is called, it will either block on Heap_Mutex with interrupts disabled (deadlock) or, if the mutex implementation uses Disable/EnableInterrupts internally, Q_Insert will be run with interrupts enabled (critical section). Similarly, with OS_Kill if Heap_Free uses Heap_Mutex.

Code also has a bug, missing DisableInterrupts in if(!new) return.
Potential race condition: if there are interrupts with higher priority than SysTick that can call OS_AddThread. But in the worst case this just results in an extra context switch, so not a real issue.

Semaphores can not be used if OS_AddThread can be called from background threads/interrupt handlers. For this reason, even the Heap can not really use semaphores either. Note that if Heap_Free uses semaphores, the deadlock/critical section in OS_Kill can not be fixed by moving the Disable/EnableInterrupts() – Q_Remove and Heap_Free need to be atomic – the only solution is to not use semaphores in the heap. And in both cases, it would be better to use Start/EndCritical instead of Disable/EnableInterrupts. (But this wasn't asked for, so both solutions are acceptable.)

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?

Thread1 will return back to its beginning when it returns/exits (on `BX LR`*), so it will repeatedly output*
Hello world: 0
Hello world: 1
….

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
}
```

In OS_AddThread, pass the parameter to InitStack:

```
new->sp = InitStack(…, param);
```

Then, in InitStack, initialize register R0 with that value:

```
long* InitStack(long *sp, void (*entry)(void), int param) {
   …
   *(--sp) = (long)0x000000000;  /* R0  */
   …
```

such that it gets passed into Thread1's first function parameter when Thread1 gets switched in and executed.

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) {

  unsigned long last;

  while(1) {

    last = OS_MsTime();

    PIDWork();

    OS_Sleep(P –
         (OS_MsTime() – last));

  }

}
```

Alternative: more accurate, using absolute time-triggered schedule:

```
void PID(void) {

  while(1) {

    PIDWork();

    OS_Sleep(P –
       OS_MsTime() % P);

  }
}
```

Hybrid: set `last` *on first call (outside while), then* `last = last + P;` *inside while loop for a fixed schedule from first call on.*

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_AddPeriodicForegroundThread(PID, 5, 10);
  OS_Launch(); // does not return
}
```

Many solutions possible, here are just a few:

1) Use a wrapper around each task that realizes one of the solutions from a).

```
void OS_ThreadWrapper(void(*task)(void), unsinged long P) {
    while(1) {
      *task();
      OS_Sleep(…);
     }
    }
```

Then, replace the PC entry point stored on the initial stack with the address of OS_ThreadWrapper() and set the initial value of R0 to the actual thread entry address (where the while(1) in the wrapper can also be replaced by solution akin to Problem 2):

```
long* InitiStack(long *sp, void (*entry)(void), unsigned long P) {
      *(sp)   = (long)0x01000000L;  /* xPSR */
      *(--sp) = (long)entry;          (long)OS_ThreadWrapper;
      …
      *(--sp) = (long)0x00000000;     (long)entry;  /* R0  */
      *(--sp) = (long)0x00000000;     P;  /* R1  */
```

2) Maintain a list of periodic thread functions and their periods in a global data structure and setup a periodic background thread (OS_AddPeriodicThread) or use the regular SysTick_Handler to call OS_AddThread on the start of a new period (e.g. (OS_MsTime() % P) == 0) for any such thread. In addition, setup the initial stack to call OS_Kill when the thread exits. This has several disadvantages: thread creation and killing overhead every period, and possible overloading of the system with multiple instances of the same thread if a one does not finish before the end of its period.

3) Extend the TCB with a period entry and a pointer to the thread function, both of which are initialized by OS_AddPeriodicForegroundThread:

```
struct tcb {
    …
    unsigned long P;  // defaults to 0
    void (*task)(void);  // defaults to 0
    …
}
```

Then, either setup the initial stack to ensure that a special OS_EndPeriod function is called when the thread exits/returns (BX LR), e.g. set the stack up to call OS_Kill (in all cases, for all threads) and let OS_Kill call OS_EndPeriod for periodic threads:

```
void OS_Kill(void) {
  if (RunPt->P) {
      return OS_EndPeriod();
```

```
  }
  …  // regular OS_Kill
}
long* InitStack(long *sp, void (*entry)(void)) {
    *(sp)   = (long)0x01000000L;  /* xPSR */
    *(--sp) = (long)entry;        /* PC */
    *(--sp) = (long)OS_Kill;  or  (long)OS_EndPeriod; /* LR */
    …
```

Then, in OS_EndPeriod, move the thread TCB from the ready list/queue into a special wait list/queue (or mark the thread as waiting for its next period in the TCB and don't schedule such threads), and trigger a context switch:

```
void OS_EndPeriod(void) {
  Q_Remove(&ReadyQ, RunPt);
  Q_Insert(&WaitQ, RunPt);
  ContextSwitch(); // should never return
}
```

Finally, in the regular SysTick_Handler, go through the list of waiting threads and check whether their start of the next period has been reached (e.g. by OS_MsTime()%P, by remembering and checking difference t to `last` *run time, or by computing remaining ticks in OS_EndPeriod and counting down similar to sleep handling). For any thread that has, move it back from the wait into the ready queue and (important!) reset the stack pointer and call* `InitStack()` *to re-initialize the stack and re-start the* `task` *in the TCB from its beginning when it is switched in next time:*

```
void SysTick_Handler(void) {
  …
  cur = WaitQ;
  while (cur) {
    if((OS_MsTime() % cur->P) == 0) {
      tmp = cur; cur = cur->next;
      Q_Remove(&WaitQ, tmp);
      Q_Insert(&ReadyQ, tmp);
      tmp->sp = InitStack(&tmp->stack[127], tmp->task);
    } else {
      cur = cur -> next;
    }
  }
  …
}
```

4) *Use the regular OS_Sleep mechanism within the OS kernel itself: extend the TCB and setup the initial stack to call OS_EndPeriod as above. Then, realize an OS_Sleep in OS_EndPeriod, while also making sure to jump back to the beginning of the thread (instead of regularly exiting) and to return to OS_EndPeriod when the thread exits again next time.*

Simple C solution:

```
void OS_EndPeriod(void) {
  while (1) {
      OS_Sleep(…);   // one of the solutions, see above/below
      *(RunPt->task)();  // call thread, let it return back here
  }
}
```

*Alternatively, if OS_EndPeriod is called via from OS_Kill (which overwrites LR in the BL call) setup link register to point back to OS_Kill or OS_EndPeriod (does not matter which) and then branch unconditionally into thread code (*not* using call/BL, which would override LR), needs assembly:*

```
OS_EndPeriod
    LDR  R0,=RunPt
    …                   ; compute OS_Sleep parameter
    BL   OS_Sleep
    LDR  LR,=OS_Kill/EndPeriod ; only needed if called from OS_Kill
    LDR  R1,[R0,#ofs]  ; load thread pointer from TCB
    BX   R1            ; branch to thread, never returns here
```

For computation of OS_Sleep parameter, if using variant with `last` *(see a)), extend the TCB:*

```
struct tcb {
    …
    unsigned long P;  // defaults to 0
    void (*task)(void);  // defaults to 0
    unsigned long last;
    …
}
```

Initialize in `OS_AddPeriodicForegroundThread()`*:*

```
new->last = OS_MsTime();
```

And call OS_Sleep and update `last` *after OS_Sleep() returns as follows:*

```
OS_Sleep(RunPt->P – (OS_MsTime() – RunPt->last));
RunPt->last = OS_MsTime();
```

or (absolute, fixed), after simplifying code (move increment before sleep and reduce equation):

```
RunPt->last = RunPt->last + RunPt->P;
OS_Sleep(RunPt->last – OS_MsTime());
```

(There should be error checking for `last < OS_MsTime()` *in case of deadline misses)*

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.

Relative jitter = $\sum_{i \leq m} E_i$ *+ OS_Sleep granularity*

Absolute jitter: depends on sleep strategy in a). With absolute/fixed time schedule, same as relative jitter. With relative schedule, errors can accumulate, so absolute jitter is potentially unbounded!

Problem 4 (15 points): Semaphores

Given the following spin-lock semaphore implementation:

```
struct Sema4 {
     long value;

    struct tcb *holder;

};
typedef struct Sema4 Sema4Type;

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

  sr = StartCritical();

  semaPt->value = value;

  semaPt->holder = 0;

  EndCritical(sr);

}
```

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

  sr = StartCritical();

  if(RunPt != semaPt->holder) {

    while(semaPt->value <= 0){
       EndCritical(sr);
       sr = StartCritical();
    }

    (semaPt->value)--;

    semaPt->holder = RunPt;
  }
  EndCritical(sr);
}

void  OS_bSignal(Sema4Type  *semaPt)
{
  int sr;

  sr = StartCritical();
  semaPt->value = 1;

  semaPt->holder = 0;

  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).

Firstly, there were a few (unintended) typos, specifically `*semaPt` *instead of* `semaPt->value`.

The main conceptual issue: OS_Wait uses EndCritical()/StartCritical() inside the while loop. If OS_Wait is called with interrupts disabled, it will result in a deadlock.

But also: a spinlock OS_Wait should not be called with interrupts disabled in any case. Either there will be a deadlock as above or, if Enable/DisableInterrupts() is used, interrupts will be enabled and critical sections re-opened when they shouldn't be.

The only proper way to fix both issues is to use an LDREX/STREX implementation.

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

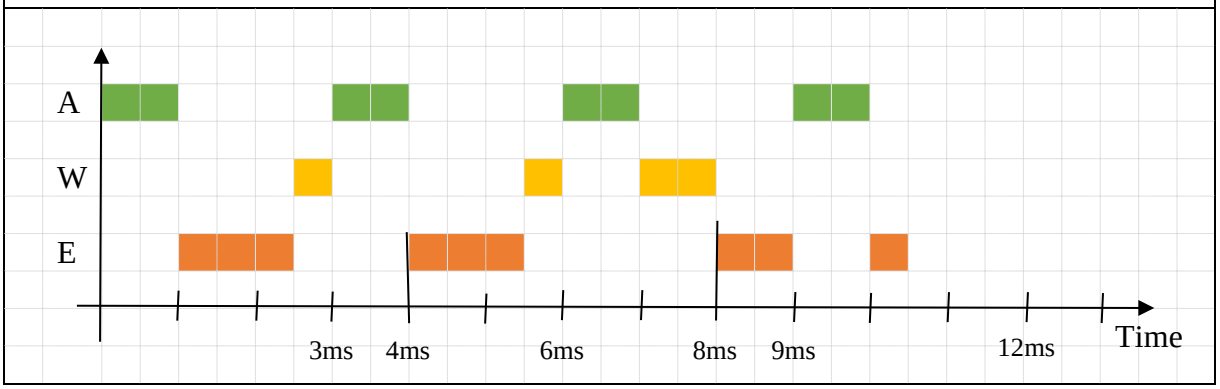
Consider a real-time system running three periodic tasks with the following periods (= deadlines) and execution times. You can assume zero context switch and interrupt overhead.

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.

e = 1.5ms

Utilization = 4/12 + 2/12 + 4.5/12 = 10.5/12 = 87.5%



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.

EDF must be able to reach 100% utilization, so e = 2ms.

Different solutions possible, this assumes that for tasks with same deadline, priority is given to smallest period: