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

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.6, Spring 2016

Midterm Solutions

Date: March 24, 2016

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

Problem 1 (20 points): Critical Sections and Deadlock

a) Given the following two routines that can be called from any user thread. Does this code have any critical sections or reentrancy issues? Justify, and list and all such cases in the code.

```
void Dbg_Out(int v) {
  int save;
  save = GPIO_PORTF_DATA_R;
  GPIO_PORTF_DATA_R = v;
  Dbg_Record(v);
  GPIO_PORTF_DATA_R = save;
}
```

```
int Dbg_Dump[MAX_DUMP];
unsigned Dbg_Count = 0;

void Dbg_Record(int v) {
  if(Dbg_Cnt==MAX_DUMP) return;
  Dbg_Dump[Dbg_Cnt] = v;
  Dbg_Cnt = Dbg_Cnt + 1;
}
```

Yes, two issues:

(1) The Dbg_Record() function is not reentrant, it performs a potentially concurrent write access to Dbg_Dump and a read-modify-write access to the global Dbg_Cnt variable.

(2) The Dbg_Out() function is also not reentrant. For one, it calls the non-reentrant Dbg_Record() routine. In addition, it also has a critical section itself when performing a read-modify-write sequence on the globally shared Port F resource.

b) Now consider the following code. Does this code have any critical sections, reentrancy or deadlock issues? Justify your answer, and list all such cases in the code.

```
void Dbg_Out(int v) {
  int save;
  DisableInterrupts();
  save = GPIO_PORTF_DATA_R;
  GPIO_PORTF_DATA_R = v;
  Dbg_Record(v);
  GPIO_PORTF_DATA_R = save;
  EnableInterrupts();
}
```

```
int Dbg_Dump[MAX_DUMP];
unsigned Dbg_Count = 0;

void Dbg_Record(int v) {
  if(Dbg_Cnt==MAX_DUMP) return;
  DisableInterrupts();
  Dbg_Dump[Dbg_Cnt] = v;
  Dbg_Cnt = Dbg_Cnt + 1;
  EnableInterrupts();
}
```

The core of Dbg_Record() is now mutually exclusive and thus reentrant, but the overall routine still has a race condition in the MAX_DUMP check: more than one thread may pass the check and thus overflow the dump buffer if they are preempted between the check and disabling interrupts.

Dbg_Out() still has a critical section making it non-reentrant: Since the Dbg_Record() call will unconditionally enable interrupts at the end of its execution, an interrupt and context switch to another thread can occur after the call. As such, it is not guaranteed that the `GPIO_PORTF_DATA_R = save` *statement will finish before another thread enters Dbg_Out().*

c) Now consider the following code. Does this code have any critical sections, reentrancy or deadlock issues? Justify your answer, and list all such cases in the code.

```
void Dbg_Out(int v) {
  int save;
  DisableInterrupts();
  save = GPIO_PORTF_DATA_R;
  GPIO_PORTF_DATA_R = v;
  Dbg_Record(v);
  GPIO_PORTF_DATA_R = save;
  EnableInterrupts();
}
```

```
int Dbg_Dump[MAX_DUMP];
unsigned Dbg_Count = 0;
sema_t Dbg_Mutex = 1;

void Dbg_Record(int v) {
  if(Dbg_Cnt==MAX_DUMP) return;
  OS_bWait(&Dbg_Mutex);
  Dbg_Dump[Dbg_Cnt] = v;
  Dbg_Cnt = Dbg_Cnt + 1;
  OS_bSignal(&Dbg_Mutex);
}
```

It depends on how OS_bWait/bSignal() are implemented:

If those internally use Enable/DisableInterrupts(), the same critical section as in 1b) still exists.

Otherwise, i.e. if they use `LDREX/STREX`*, except for the MAX_DUMP race condition, both routines are mutually exclusive and thus reentrant. However, the code then has potential for deadlocks: Dbg_Record() can be interrupted and preempted while holding the Dbg_Mutex semaphore. If another thread then calls Dbg_Out(), it will block on Dbg_Mutex when calling Dbg_Record() while interrupts (and hence all context switches) are disabled.*

d) Provide another solution that avoids all critical sections and deadlocks. Prove that your solution is deadlock-free. List all deadlock conditions and show that at least one of them is violated.

Two possible solutions:

(1) *Replace all Enable/DisableInterrupts() in solution b) with calls to Start/EndCritical() instead (such that Dbg_Record() properly saves and restores interrupt conditions on entry/exit).*
(2) *Replace Enable/DisableInterrupts() in solution c) with OS_bWait/bSignal() on another mutex specific to Port F.*

Deadlock conditions:

(i) *Mutual exclusion*
(ii) *Hold and wait*
(iii) *Circular wait*
(iv) *No preemption of wait*

Solution (1) violates (ii) and (iii). There is no waiting/blocking of threads. Once a thread has disabled interrupts, it has acquired exclusive access to the whole machine.

Solution (2) violates (iii). Assuming mutexes are not accessed otherwise, threads either only ever access one mutex (in Dbg_Record), or the two mutexes are always acquired in the same order (in Dbg_Out).

Problem 2 (20 points): Priority 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)	10ms	30ms
Warning (W)	20ms	40ms
Engine (E)	10ms	60ms

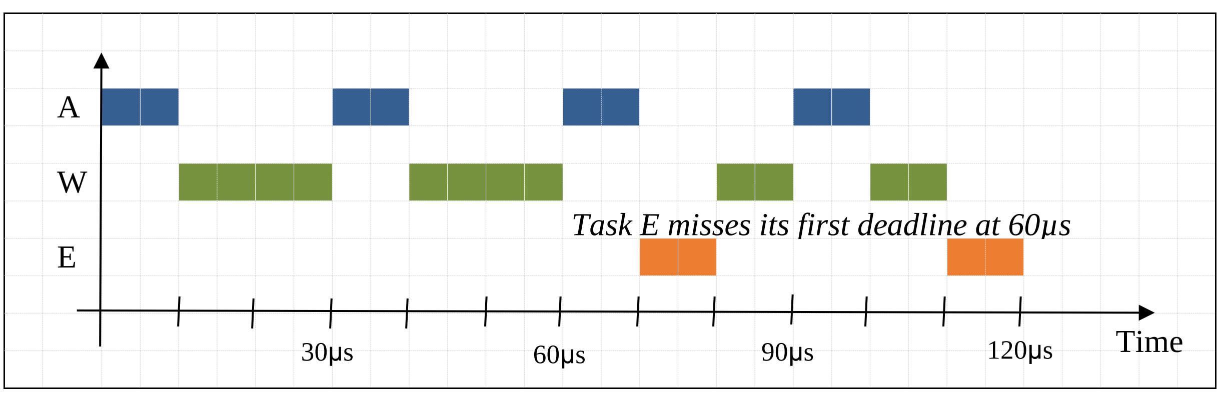
a) Assign priorities to tasks to implement a rate monotonic scheduling (RMS) strategy.

Assign priorities inversely proportional to task periods:
Highest priority: A
Medium priority: W
Lowest priority: E

b) What is the processor utilization when executing this task set.

Utilization = 10/30 + 20/40 + 10/60 = 1/3 + 1/2 + 1/6 = 100%

c) Draw the schedule of task executions over time. Assume that all tasks become ready to execute, i.e. start their first period at time zero. Draw one iteration of the schedule until it starts repeating. Is the task set schedulable, i.e. do all task finish their execution before the start of their next period (=deadline)?



Problem 3 (25 points): OS Sleep Support

a) Given the basic round-robin OS kernel code below, show the necessary modifications (insertions and/or deletions) to add *OS_Sleep()* functionality. Keep your implementation simple, i.e. you are not required to optimize for performance.

```
struct tcb {
  long *sp;
  struct tcb *next;

  unsigned long sleep;

}

struct tcb* RunPt;
```

```
#define ContextSwitch() (NVIC_INT_CTRL_R=0x10000000) // trigger PendSV
void DisableInterrupts(void);
void EnableInterrupts(void);
long StartCritical(void);
void EndCritical(long sr);

void OS_Sleep(unsigned long delay) {

  RunPt->sleep = (delay / SYSTICK_PERIOD);  // set thread to sleep

  ContextSwitch();    // OS_Suspend();

}

void SysTick_Handler(void) {

  struct tcb* pt;

  pt = RunPt;         // decrement all sleeping threads
  do {
    if(pt->sleep) pt->sleep = pt->sleep – 1;
    pt = pt->next;
  } while(pt != RunPt);

  ContextSwitch();

}
```

```
PendSV_Handler

    CPSID   I

    PUSH    {R4-R11}

    LDR     R0, =RunPt

    LDR     R1, [R0]

    STR     SP, [R1]

next
    LDR     R1, [R1,#4]
                           LDR R2, [R1,#8]  ; load sleep value
                           CMP R2, #0       ; zero?
                           BEQ next         ; no -> keep looking

    STR     R1, [R0]

    LDR     SP, [R1]

    POP     {R4-R11}

    CPSIE   I

    BX      LR
```

b) How does your OS implementation behave when all threads are sleeping, i.e. what happens when all but one thread currently sleep and the last active/running thread calls *OS_Sleep()*?

Note that there are many different solutions for question a).

For the specific solution outlined above, the PendSV_Handler will go into an endless loop once triggered from the last active thread's OS_Sleep() call. This solution could be made to behave as expected by ensuring that PendSV has the lowest interrupt priority and then temporarily enabling interrupts in the PendSV_Handler while looking for a non-sleeping thread, such that the Systick_Handler can be fired to keep counting down sleep times until one thread is woken up (sleep value becomes non-zero and loop in PendSV_Handler thus exits).

The proper way to handle this situation, however, is to always have one never-ending, always-active so-called Idle Task running (with lowest priority in case of priority scheduling). This takes care of any situation in which all threads are sleeping, inactive (killed), blocked, etc.

Problem 4 (20 points): Thread Exit and Kill

Assume a basic round-robin OS kernel (as shown in Problem 3) with the following *OS_AddThread()* implementation:

```
long* SetInitialStack(long *sp, void (*entry)(void)) {
  *(sp)   = (long)0x01000000L;  /* xPSR */
  *(--sp) = (long)entry;        /* PC  */
  *(--sp) = (long)0x14141414L;  /* R14 */    = (long)OS_Kill;
  *(--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;
}

int OS_AddThread(void(*task)(void)) { long sr;
  struct tcb* newPt;

  sr = StartCritical();


  newPt = AllocTcb();  // get TCB & stack, set SP to top of stack
  if(!newPt) { EndCritical(sr); return 0; }


  newPt->sp = SetInitialStack(newPt->sp, task);

  newPt->next = RunPt->next;
  RunPt->next = newPt;


  EndCritical(sr);
  return 1;
}
```

Given the following user code:

```
void main(void) {
  int i;
  OS_Init();
  OS_AddThread(Thread1);
  ...
  OS_Launch();
}
```

```
Thread1

    ...

    BX  LR       <--- PC
```

a) What happens when *Thread1* exits normally (without calling *OS_Kill()*)? In other words, if the current PC points to the last `BX LR` return instruction in *Thread1*, where will the branch go to and what line of code will be executed next? Hint: think about the value that will be in the `LR` register during execution of *Thread1* and thus when `BX LR` is executed.

The link register (LR) will point to the value of R14 restored from the initial stack of the thread, i.e. 0x14141414. As such, the thread will branch to address 0x14141414 when executing the final `BX lR` instruction, i.e. to an undefined address.

b) Modify the OS kernel code shown above such that *OS_Kill()* will be automatically executed whenever a thread added to the OS exits normally. You are only allowed to make modifications to the kernel but not the user code, i.e. the *OS_xxx* interface must remain as is and must work for an arbitrary number of threads. If you need additional code, show it in the box below.

Two possible solutions:

(1) Simply replace the initial value of R14/LR with the address of the OS_Kill function to branch to when executing the final `BX LR` in a thread.

(2) Write a wrapper routine that takes as argument a pointer of the main thread function and then ensures that OS_Kill() is called after that thread function finishes:

```
void OS_ThreadWrapper(viod(*task)(void)) {
  *task();
  OS_Kill(); // never returns
}
```

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:

```
long* SetInitialStack(long *sp, void (*entry)(void)) {
  *(sp)   = (long)0x01000000L;  /* xPSR */
  *(--sp) = ~~(long)entry;~~          (long)OS_ThreadWrapper;
  *(--sp) = (long)0x14141414L;
  *(--sp) = (long)0x12121212L;
  *(--sp) = (long)0x03030303L;
  *(--sp) = (long)0x02020202L;
  *(--sp) = (long)0x01010101L;
  *(--sp) = ~~(long)0x000000000;~~    (long)entry;
...
```

Problem 5 (15 points): Synchronization and Deadlock

a) Given the two threads below, can any deadlock occur? If yes, why (show an execution sequence leading to deadlock)? If not, why not (prove that there is no deadlock)?

```
void Thread1(void) {
  OS_bWait(&file_mutex);
  ...
  OS_bWait(&memory_mutex);
  ...
  OS_bSignal(&memory_mutex);
  ...
  OS_bSignal(&file_mutex);
}
```

```
void Thread2(void) {
  OS_bWait(&memory_mutex);
  ...
  If(debug) {
    OS_bWait(&file_mutex);
    ...
    OS_bSignal(&file_mutex);
  }
  ...
  OS_bSignal(&memory_mutex);
}
```

Yes, deadlock can occur. Possible sequence:

1) *Thread 1 acquires file_mutex*
2) *Thread 2 acquires memory_mutex*
3) *Thread 2 blocks on acquiring file_mutex (debug condition is true)*
4) *Thread 1 blocks on acquiring memory_mutex.*

b) Given the two threads below, can any deadlock occur? If yes, why (show an execution sequence leading to deadlock)? If not, why not (prove that there is no deadlock)?

```
void Thread1(void) {
  OS_bWait(&file_mutex);
  ...
  OS_Wait(&available);
  ...
  OS_bSignal(&file_mutex);
}
```

```
void Thread2(void) {
  OS_bWait(&file_mutex);
  ...
  If(new_data) {
    OS_Signal(&available);
  }
  ...
  OS_bSignal(&file_mutex);
}
```

Yes, deadlock can occur. Possible sequence:

1) *Thread 1 acquires file_mutex*
2) *Thread 1 blocks on available*
3) *Thread 2 blocks on file_mutex*

c) Assume that the code is running on top of an OS that uses priority scheduling and a priority ceiling protocol for all semaphores. For each of the examples above, can any deadlock still occur? If yes, why (show an execution sequence leading to deadlock)? If not, why not (prove that there is no deadlock)?

Example a) *No, deadlock cannot occur any more. As soon as one thread acquires either mutex, its priority will be raised to the maximum and it can not be preempted any more until it releases that semaphore. The priority ceiling protocol is equivalent to disabling interrupts, except that actual interrupts and interrupt-driven background threads can still preempt any foreground thread.*
Example b) *Deadlock can still occur, priority ceilings do not prevent this type of deadlock when actual synchronization (not only mutual exclusion) is involved.*