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

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.6, Spring 2015

Midterm Solutions

Date: March 12, 2015

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	10	
Problem 2	15	
Problem 3	20	
Problem 4	20	
Problem 5	10	
Problem 6	25+10	
Total	100+10	

Problem 1 (10 points): Reentrance

Given the following C code and the assembly code generated by the compiler for a library function that converts an integer into a bit string:

```
// bit.h
const char* bit2str(int b);
```

```
; bit.s
bit2str:
  MOV   r1,r0
  MOVS  r0,#0x00
  LDR   r3,[pc,#28]  ; @0x0564
  STRB  r0,[r3,#0x20]
  MOVS  r2,#0x1F
  B     0x0000055C
0x054E:
  AND   r0,r1,#0x01
  ADDS  r0,r0,#0x30
  LDR   r3,[pc,#12]  ; @0x0564
  STRB  r0,[r3,r2]
  ASRS  r1,r1,#1
  SUBS  r2,r2,#1
0x055C:
  CMP   r2,#0x00
  BGE   0x0000054E
  LDR   r0,[pc,#0]  ; @0x0564
  BX    lr
0x0564:
  DCW   0x1004
  DCW   0x2000
```

```
// bit.c

static char buf[33];

// Example use case:
// printf("V: %s\n", bit2str(v));

const char* bit2str(int b) {
  int i;
  buf[32] = 0;
  for(i=31; i>=0; i--) {
    buf[i] = '0' + (b & 0x01);
    b >>= 1;
  }
  return buf;
}
```

Is the *bit2str()* function reentrant? If yes, why? If not, explain why not (provide a counterexample), and indicate how could it be changed to make it reentrant.

No, it is not reentrant. The function includes a write-read access to a shared, global variable.

To make it reentrant, we need to change the library interface such that the buffer is owned by the caller:

const char bit2str(char* buf, int b);*

Note that using a mutex semaphore to protect the global access will not work here – the function returns a (read-only) pointer to the shared resource. As such. the critical section extends beyond the end of the function into the caller until the last time the pointer is accessed..

Problem 2 (15 points): Stack Size

In class, we talked about estimation of stack sizes. Given the C code and assembly code generated by the compiler as shown below. Assume that the *ADC_In()* and *GPIO_Out()* functions are driver code that only accesses hardware registers.

```
Thread (0x0544):
  SUB   sp,SP,#0x1000
0x054A:
  BL.W  ADC_In (0x0000040C)
  MOV   r1,r0
  MOV   r0,sp
  BL.W  filter (0x0000055E)
  MOV   r4,r0
  BL.W  GPIO_Out (0x00000410)
  B     0x0000054A
filter (0x055E):
  PUSH  {r4-r6,lr}
  MOV   r4,r0
  MOV   r6,r1
  MOVW  r5,#0x3FF
  B     0x00000576
0x056A:
  SUBS  r0,r5,#1
  LDR   r0,[r4,r0,LSL #2]
  STR   r0,[r4,r5,LSL #2]
  SUBS  r5,r5,#1
0x0576:
  CMP   r5,#0x00
  BGT   0x0000056A
  STR   r6,[r4,#0x00]
  MOV   r0,r4
  BL.W  fir (0x00000584)
  POP   {r4-r6,pc}
fir (0x0584):
  PUSH  {r4,lr}
  MOV   r2,r0
  MOVS  r0,#0x00
  MOVS  r1,#0x00
  B     0x0000059E
0x058E:
  LDR   r3,[pc,#24]  ; @0x000005A8
  LDR   r3,[r3,r1,LSL #2]
  LDR   r4,[r2,r1,LSL #2]
  MLA   r0,r3,r4,r0
  ADDS  r1,r1,#1
0x059E:
  CMP   r1,#0x400
  BLT   0x0000058E
  POP   {r4,pc}
0x05A8:
  DCW   0x0600
  DCW   0x2000
```

```
const int h[1024] = {
  …
};

int fir(int x[]) {
  int i;
  int r = 0;
  for(i=0; i<1024; i++)
    r += h[i] * x[i];
  return r;
}

int filter(int x[], int v) {
  int i;
  for(i=1023; i>0; i--)
     x[i] = x[i-1];
  x[0] = v;
  return fir(x);
}

void Thread(void) {
  int v;
  int x[1024];

  while(1) {
    v = ADC_In();
    GPIO_Out(filter(x, v));
  }
}
```

a) Are the *fir()* and *filter()* functions reentrant? Why or why not?

Yes, both are reentrant. They do not have any write access to any global resource.fir() accesses a shared variable (h[]), but read only, so no critical section.

b) Draw the function-by-function call graph for the *Thread*.

```
Thread -> ADC_In
       |-> filter -> fir
       \-> GPIO_Out
```

c) Determine the minimum amount of stack space needed to avoid stack overflow when running the *Thread* as a foreground thread in a preemptively scheduled system. Show your work and explain how you arrive at the result.

Stack needs of individual functions:

*fir: 2*4 = 8 bytes*
*filter: 4*4 = 16 bytes*
*Thread: 1024 * 4 = 4096 bytes*
ADC_In and GPIO_Out are assumed to require no stack space

With this, at the deepest point of the call graph, i.e. when executing fir(), the stack will have 8+16+4096 = 4120

*In addition, if we get preempted while executing fir(), we need an additional 16*4=64 bytes to save the current thread's context (16 register).*

So the minimum stack size needed for Thread is 4184 bytes.

Problem 3 (20 points): Weighted Round-Robin Scheduler

You are asked to implement a system that uses a weighted, preemptive round-robin scheduler. Each foreground thread is thereby associated with an integer *weight* parameter that specifies the number of time slices the thread is supposed to run before switching over to the next thread in sequence. Starting from the basic round-robin OS code, show the necessary modifications (insertions and/or deletions) to add weighted functionality. Maintain a constant SysTick interrupt period, i.e. you are not allowed to change the reload value. Assume that non-cooperative spinlock sempahores are used and no sleeping functionality is needed.

```
struct tcb {
  long *sp;
  struct tcb *next;
  unsigned int weight;  // greater than 0

  unsinged int ticks;

}

struct tcb* RunPt;
```

```
SysTick_Handler

    CPSID   I

    PUSH    {R4-R11}

    LDR     R0, =RunPt

    LDR     R1, [R0]
                           LDR   R2,[R1,#12] ; load ticks
                           SUBS  R2,#1       ; decrement
                           STR   R2,[R1,#12] ; and write back
                           BNE   skip        ; if > 0, keep running
                           LDR   R2,[R1,#8]  ; else reset w/ weight
                           STR   R2,[R1,#12] ; and write back -> ticks
    STR     SP, [R1]                         ; and perform switch

    LDR     R1, [R1,#4]

    STR     R1, [R0]

    LDR     SP, [R1]

skip:

    POP     {R4-R11}

    CPSIE   I

    BX      LR
```

Problem 4 (20 points): Real-Time Performance

Consider a priority scheduled real-time system running three interrupt-triggered foreground tasks with the following priorities and worst-case execution times. All tasks are sporadic/aperiodic with at least 100µs between consecutive activations of the same task. You can assume zero context switch and interrupt overhead.

Task	**Priority**	**Execution Time**
Airbag	High	10µs
Warning	Medium	20µs
Engine	Low	30µs

a) What is the worst-case latency (time between triggering the interrupt and the task starting to execute) and worst-case response time (between interrupt trigger and task finishing execution) for each task?

	Max. Latency	Max. Response Time
Airbag	*0µs*	*10µs*
Warning	*10µs*	*30µs*
Engine	*30µs*	*60µs*

b) Now consider that the *Warning* and *Engine* tasks access a shared resource that is protected with a blocking mutex semaphore. Assuming each task does not hold the mutex for longer than 5µs, what are the worst-case latencies and response times?

	Max. Latency	Max. Response Time
Airbag	*0µs*	*10µs*
Warning	*10µs*	*35µs*
Engine	*30µs*	*60µs*

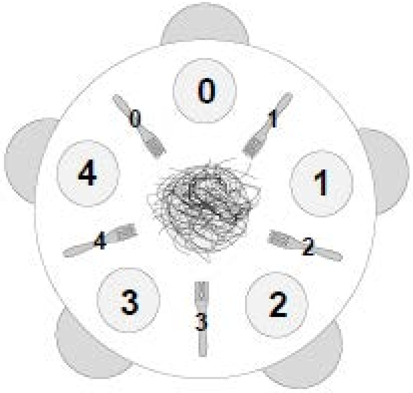
c) Assuming instead that the *Airbag* and *Engine* tasks access a shared mutex for no longer than 5µs each, what are the worst-case latencies and response times?

	Max. Latency	Max. Response Time
Airbag	*0µs*	*35µs*
Warning	*10µs*	*30µs*
Engine	*30µs*	*60µs*

d) What do we call the effect that causes changes in latencies/response times between a)-c)?

Priority inversion

Problem 5 (10 points): Dining Philosophers



In class, we talked about the classical Dining Philosophers problem. Assume five philosophers are sitting at a round table with five plates and five forks. The philosophers continuously alternate between eating and thinking. To eat, a philosopher needs two forks. The synchronization problem is that each fork can only be held by one philosopher at a time. The goal is to find a solution in which it is guaranteed that no philosopher will starve, while allowing as many philosophers to eat at the same time as possible.

Given the following coding of this problem in which philosophers are represented by threads and forks represent shared resources protected by semaphores:

```
Thread0() {
 for(;;) {
  think();
  wait(&f0);
  wait(&f1);
  eat();
  signal(&f1);
  signal(&f0);
 }
}
```

```
Thread1() {
 for(;;) {
  think();
  wait(&f1);
  wait(&f2);
  eat();
  signal(&f2);
  signal(&f1);
 }
}
```

```
Thread2() {
 for(;;) {
  think();
  wait(&f2);
  wait(&f3);
  eat();
  signal(&f3);
  signal(&f2);
 }
}
```

```
Thread3() {
 for(;;) {
  think();
  wait(&f3);
  wait(&f4);
  eat();
  signal(&f4);
  signal(&f3);
 }
}
```

```
Thread4() {
 for(;;) {
  think();
  wait(&f4);
  wait(&f0);
  eat();
  signal(&f0);
  signal(&f4);
 }
}
```

Is this a valid solution to the problem that satisfies all constraints? If yes, prove it. It no, explain why not (provide a counterexample), and show modified code that provides a valid solution.

The code has a potential deadlock. Assume the each thread grabs its first semaphore and then gets preempted before being able to acquire its second semaphore. At that point, there will be a circular hold-and-wait dependency chain.

There are two solutions to this problem:

1) *Remove the circular wait condition by making sure all threads access all semaphores in the same order. In other words, change Thread4 to*

```
Thread4() {
 for(;;) {
  think();
  wait(&f0);
  wait(&f4);
  …
```

2) *Introduce an extra counting semaphore to make sure that no more than 4 threads are trying eat at the same time. Introduce a counting semaphore “limit”, initialized to 4:*

```
ThreadX() {
 for(;;) {
  think();
  wait(&limit);
  wait(&fx);
  wait(&fx+1);
  …
```

Problem 6 (25+10 points): Synchronization

a) Consider a problem in which we want to synchronize two foreground threads such that each thread can only proceed beyond a certain point once it is guaranteed that the other thread has also arrived at its synchronization point. This is called a *rendezvous* pattern. In other words, using only semaphores and regular C statements/variables, complete the following code such that *a2()* executes after *b1()*, and *b2()* executes after *a1()*:

```
// Global variables and semaphores


sema4_t a = 0;
sema4_t b = 0;
```

```
void ThreadA(void) {



  a1();

  //rendezvous here



  bSignal(&a);
  bWait(&b);







  a2();



}
```

```
void ThreadB(void) {



  b1();

  //rendezvous here



  bSignal(&b);
  bWait(&a);







  b2();



}
```

b) The generalization of a rendezvous with *N* threads is called a *barrier*. Many operating systems will provide a native barrier synchronization primitive. Show the C implementation of a *OS_Barrier()* function that provides a spinlock realization of barrier synchronization. Also demonstrate how to use your *OS_Barrier()* function in the following code, such that each thread only executes *b()* once it is guaranteed that all other threads have finished executing *a()*.You can assume that *N* is known and given at compile time. Hint: start from the C implementation of regular spinlock counting semaphores and show the minimally necessary modifications to turn it into a barrier.

```
void OS_Barrier(  sema4_t*  b                        )
{
                        or
  DisableInterrupts();          DisableInterrupts();
  *b -= 1;                      *b -= 1;
  while(*b > 0) {               EnableInterrupts();
    EndableInterrupts();        while(*b > 0) {}
    DisableInterrupts();
  }
  EnableInterrupts();




}
```

```
#define N …

// Global variables and semaphores/barriers

sema4_t b = N;


void Thread(void) {



  a();

  // barrier here

  OS_Barrier(       &b                         );

  b();


}

void main(void) {
  int i;
  OS_Init();
  for(i=0; i<N; i++) { OS_AddThread(&Thread); }
  OS_Launch();
}
```

c) **(Required for graduate students, extra credit for undergraduates)** A barrier functionality can also be realized with standard semaphores. Using only sempahores and regular C statements/variables, complete the following code to realize a barrier among *N* threads. Hint: think about how you can realize the equivalent code of your *OS_Barrier()* function from b) with just regular variables and semaphores.

```
#define N …

// Global variables and semaphores


unsigned int count = N;
sema4_t mutex = 1;
sema4_t turnstile = 0;





void Thread(void) {


  a();

  // barrier here

  bWait(&mutex);
  count -= N;
  if(!count) bSignal(&turnstile); // Can be outside mutex
  bSignal(&mutex);                  // if so, can potentially be
                                    // replaced by busy-waiting
  bWait(&turnstile);                //  while(count) {};
  bSignal(&turnstile);              // but that will not work
                                    // with priority-based OS

  // Note that this does not allow the barrier to be reused.
  // The turnstile ends up in a wrong state afterwards.

  // For a reusable solution, and many other tips and tricks
  // around synchronization issues, see
  //   Allen B. Downey, The Little Book of Semaphores
  //   http://greenteapress.com/semaphores/

  b();

}

void main(void) {
  int i;
  OS_Init();
  for(i=0; i<N; i++) { OS_AddThread(&Thread); }
  OS_Launch();
}
```