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

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.12, Spring 2018

Final Exam Solutions

Date: May 10, 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	20	
Problem 2	15	
Problem 3	20 + 10	
Problem 4	25	
Problem 5	10	
Problem 6	5	
Problem 7	5	
Total	100 + 10	

*Name:*______________________________

Problem 1 (20 points): Miscellaneous

a) Does it make sense to give a CPU-bound thread higher priority for disk I/O than an I/O bound thread? Why? Explain your answer briefly.

True, for the same reason that I/O-bound threads should get higher priority for the CPU. If a CPU-bound thread must wait for I/O behind I/O-bound threads, we could end up in a situation where all of the threads are waiting for I/O and the CPU is idle. Better to finish the I/O for the CPU-bound threads as quickly as possible so they can get keep the CPU busy; this increases our chances of keeping all of the system components busy at all times.

b) A virtual memory system with paging can have external fragmentation, true or false? Why or why not?

False. Paging is technique in which logical memory is broken into blocks of same size called pages, such that the physical address space of a process can be non-contiguous.

c) Virtual address sizes must be the same as physical address sizes, true or false, and why? What determines the size of virtual and physical addresses?

False. The size of physical addresses is determined by the hardware. The size of virtual addresses by the width of page table entries, so it can be either larger or smaller than physical addresses.

d) Describe two situations in which spinlock (busy waiting) semaphores can be more appropriate than blocking ones.

Very short critical sections

Tightly interacting threads (very short expected wait times)

When ultra-low latency / very short reaction time is required

Implementing locks on multiprocessors with true concurrency, where priorities don't matter.

e) Assume you have a system in which there can be a circular wait pattern in the resource-allocation graph. Is this a definite sign of a deadlock? Why or why not?

No. This is not a definite sign of deadlock. The other conditions for deadlock must be satisfied (hold-and-wait, mutual exclusion and no preemption). While a circular wait pattern implies both mutual exclusion (i.e. waiting in the first place) and hold-and-wait, the no preemption condition may still not hold.

f) Can a system with all jobs having the same priority ever suffer from the priority inversion problem? Why or why not?

No. The priority inversion problem refers to the problem where lower priority jobs end up blocking higher priority jobs from running. If all jobs have the same priority, then clearly this cannot happen.

g) Consider an RTOS implementing a priority scheduler, where two tasks having the same priority is not allowed (such as in uCOS). Does the OS ever require a context switch in the default periodic SysTick handler?

No. In pure priority scheduling, a context switch is only necessary if a thread with higher priority than the currently running one wakes up, which cannot happen during normal SysTick. With one exception: when SysTick handles sleep expiration, a higher priority thread can be woken up as part of that, which then requires a context switch in the SysTick handler.

Problem 2 (15 points): Thread Synchronization

Malek, Shailesh, and Shan go to an Indian restaurant at a busy time of the day. The waiter apologetically explains that the restaurant can provide only two pairs of spoons (for a total of four spoons) to be shared among the three people. Shailesh proposes that all four spoons be placed in an empty glass at the center of the table and that each diner should obey the following protocol (where `spoon` is a semaphore that is initialized to 4):

```
while (!had_enough_to_eat()) {
  OS_Wait(&spoon);
  OS_Wait(&spoon);
  eat();
  OS_Signal(&spoon);
  OS_Signal(&spoon);
}
```

a) Can this dining plan lead to a deadlock? Explain your answer.

No. Deadlock cannot occur because there are enough spoons to guarantee that, even if all 3 diners grab one spoon and then get preempted, at least one of the three diners will be able to get one additional, i.e. the two spoons that he/she needs. Once the diner finishes, he/she will release the spoons for someone else to use. So, eventually everyone finishes.

b) Can there ever be a deadlock depending on the number of spoons? If so, show such a deadlock scenario. What is the minimum number of spoons for there not to be a deadlock?

Yes. There can be a deadlock depending on the number of spoons.

Example of deadlock scenario with only 3 available spoons and 3 persons each requiring 2 spoons to eat: It might happen that each person gets one spoon, then is preempted and is blocked forever waiting for the other spoon to be available. This will result into deadlock.

In general: the minimum number of spoons required to avoid deadlock is 4 (clearly, there is a deadlock with 3, but none with 4).

c) Suppose now that instead of three there will be an arbitrary number of D diners. Furthermore, each diner $d = 1..D$ may require a different number of S_d spoons to eat. For example, it is possible that one of the diners is an octopus, who for some reason refuses to begin eating before acquiring $S_{octopus} = 8$ spoons. What is the smallest number of spoons needed to ensure that deadlock can not occur?

$\sum_d S_d - D + 1$. *This guarantees that every diner can get all but one of the spoons it needs, with one additional spoon to guarantee that at least one diner gets all of the spoons it needs.*

Problem 3 (20 + 10 points): Synchronization Primitives

In class, we discussed that atomic test-and-set and more generalized compare-and-swap operations can provide minimal canonical primitives to solve any synchronization problem, and that such operations have thus historically been directly implemented in hardware in many machines.

a) Our ARM does not provide built-in test-and-set or compare-and-swap operations. Shown below is C code for a software implementation on the ARM. This code is not atomic and thus has race conditions. Describe possible race conditions. Modify the code to make it atomic. You can assume that standard OS primitives are available:

```
    void DisableInterrupts(void);
    void EnableInterrupts(void);
    long StartCritical(void);
    void EndCritical(long);
```

```
int compare_and_swap(int* dst, int expected, int new) {

     int cur;

     long sr = StartCritical();

     cur = *dst;

     if(cur == expected) {

          *dst = new;

     }

     EndCritical(sr);

     return cur;

}

int test_and_set(int* dst) {    // binary version

     return compare_and_swap(dst, 0, 1);

}
```

*Race condition if thread gets preempted after make a local copy of *dst in cur, and another thread updates *dst in the middle. Then both threads may assume that cur == expected, return the same cur, and update *dst with different new values non-deterministically.*

b) Implement binary spin-lock semaphores in C using only test-and-set or compare-and-swap primitives.

```
void OS_bInitSemaphore(int *semaPt) { // initialize to be unlocked

     *semaPt = 0;  // 0 is unlocked, 1 is locked
     alt.: *semaPt = 1;  // traditional 1 is unlocked, 0 is locked

}
void OS_bWait(int *semaPt) {


     while(test_and_set(semaPt) == 1) ;
     alt.: while(compare_and_swap(semaPt, 1, 0) == 0);

}
void OS_bSignal(int* semaPt) {

     *semaPt = 0;  // can use compare_and_swap, but atomic as is
     alt.: *semaPt = 1;  // can use test_and_set(), but not needed

}
```

c) In addition to semaphores themselves, test-and-set and compare-and-swap primitives allow implementing so-called non-blocking or lock-free data structures that are made thread-safe without traditional coarse-grain and expensive blocking using semaphores. Shown below is the code for inserting a TCB into the OS ready queue as called from *OS_AddThread()* shown in the midterm. Modify (add/delete/replace) the code to make it thread-safe using only compare-and-swap or test-and-set.

```
void Q_Insert(struct TCB **queue, struct TCB *new) {

     do {
       new->next = *queue;

     ~~*queue = new;~~
     } while(compare_and_swap(queue, new->next, new) != new->next);

}
```

d) **(Required for graduate students, extra credit for undergraduates)** What are advantages and disadvantages of lock-free programming versus use of semaphores? When can and should it be used, when can it not be used?

Advantages: Avoids many issues with locking and semaphores, such as deadlocks (deadlock free if lock-free is used exclusively and not mixed with semaphores) or priority inversions. Also can be less overhead and higher performance. And can be used in interrupt handlers.

*Disadvantages: Hard in the general case (e.g .try implementing lock-free removing from a queue…). It is also a form of busy-waiting, so wastes cycles if there is a lot of contention for a resource. Note, however, that waiting/spinning in lock-free programming is *not*about blocking, i.e. threads actively waiting on each other (spinning until some shared variable update is performed by another thread, as is done in the semaphore implementation). It is just waiting until the resource is not contended so an update can hbe performed without conflict (spinning until there is *no* update by another thread, as is in the queue case). Which should always be brief, i.e. the default assumption is for the update to succeed and conflicts requiring wait/spinning to be the exception. This is different from actively locking out other threads, where the default assumption is that there will be a conflict that needs blocking/suspension.*

e) **(Required for graduate students, extra credit for undergraduates)** Instead of native test-and-set or compare-and-swap, the ARM provides hardware support for synchronization through `LDREX/STREX` instructions. Implement compare-and-swap in assembly using `LDREX/STREX.`

```
compare_and_swap ; pointer in R0, expected in R1, new in R2


     LDREX      R3,[R0]

     CMP        R3,R1

     BNE        end

     STREX      R12,R2,[R0]

     CMP        R12,#0

     BNE        compare_and_swap

end

     MOV        R0,R3



     BX    LR
```

Problem 4 (25 points): Filesystems

Assume you have a disk with a filesystem using indexed allocation, where the first blocks on the disk store the directory information and the global index table.

a) Assuming a disk with 65536 blocks of 512 byte each, what is the size of the index table and how many blocks on disk does it occupy? What is the largest disk size supported if the index table must fit into one disk block? What if it needs to fit into half a block?

Size of index table & number of index blocks for 65536 block disk?	*16-bit indices to represent 65536 blocks. 65536 entries in the index table. For a total of 65536*2 = 128kB, which is 128kB/512B = 256 blocks.*
Largest disk size with 1 block index table?	*512 bytes can hold 512*1 or 256*2 index table. A disk with 512 blocks would require 2-byte indices. So only 256 entries fit, for a 256-block or 256*512 = 128kB disk.*
Largest disk size with ½ block index table?	*A half block can hold a 256*1 index table. 1-byte indices are enough for a 256 block disk. So same as above, a 256-block or 128kB disk.*

b) What is the reliability of this filesystem? Can the files on disk be partially or completely recovered if i) the directory block(s), ii) the index table block(s) or iii) any other block(s) on disk are damaged? What can and cannot be recovered in each case?

Block(s) lost	Recoverable?
i) Directory	*No files can be recovered. Only used vs. unused data blocks can be recovered from the index table, but mapping of blocks to files is lost.*
ii) Index table	*No files can be recovered. No data can be recovered. Mapping of files to blocks on disk is lost. Information about used vs. unused blocks on disk is lost.*
iii) Other	*Files can be partially recovered. Only the damaged block in corresponding files are lost.*

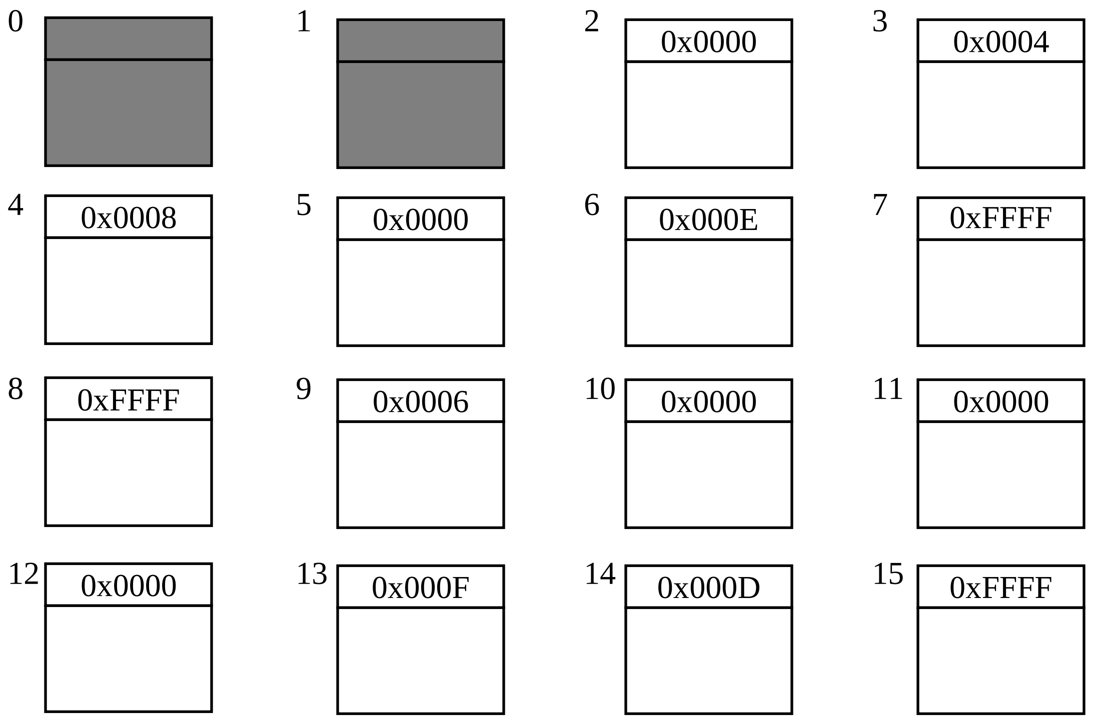
c) To improve reliability (while maintaining fast random access capabilities), indexed allocation can be combined with linked allocation. Shown below is a disk using such a combined filesystem, where the first two blocks on disk normally store the directory and index table, but have been damaged. The first two bytes of each regular block otherwise contain a pointer to the address of the next block in the file, 0xFFFF if the block is the last one, or 0x0000 if the block does not belong to a file (is empty). Assuming the directory and index blocks have both been damaged, what can be recovered from the remaining disk information? For the specific disk content below, recover as much information as possible to fill in the missing directory and index table information.

Directory

File name	Start block	Start index	Size (blocks)
Can not be recovered	*3*	*0*	*3*
	9	*3*	*5*
	7	*8*	*1*

Index Table

	Block
0	*3*
1	*4*
2	*8*
3	*9*
4	*6*
5	*14*
6	*13*
7	*15*
8	*7*
9	
10	
11	
12	
13	
14	
15	



d) How would the recovered directory and index table be different when block 14 would also be unreadable in the example above?

Second file (starting at block 9) would be recovered as two separate files:
One starting at block 9 (index 3) with size of 3 blocks (last block marked as damaged).
Second one starting at block 13 (index 6) with size of 2 blocks.

Alternatively:
First file would have one less block. Index table entry 5 would be removed and all entries below (6-8) would move up by one (to position 5-7). Directory start indices to be adjusted accordingly.

Problem 5 (10 points): Relocation

Given the following simple user program to be loaded and executed by the OS:

```
// Display_Message prototype
#include "display.h"

int live = 0;
char* msg = "Hello";

void inc(void)
{
  live++;
}

int main(void)
{
  inc();
  Display_Message(0, 0, msg, live);
}
```

For each of the following two disassembled program data and code regions as generated by different compiler variants, is the program position-independent, i.e. will it be able to execute as is when loaded into an arbitrary location in memory? If not, indicate which lines of code are not and show how the code will need to be relocated by the OS at load time. You can assume that the OS will set R9 appropriately when launching the process to execute the program.

*Name:*_______________________________

a)

```
0x20000000 0000      DCW           0x0000
0x20000004 …         DCB           “Hello”,0
```

```
    Display_Message
0x00000092 F8DFC004  LDR           r12,[pc,#4]  ; @0x0000009A
0x00000096 4760      BX            r12

0x0000009A ~~00000000  DCD           0x00000000~~   <-- Needs patch

    inc
0x00000110 4802      LDR           r0,[pc,#8]  ; @0x0000011C
0x00000112 6800      LDR           r0,[r0,#0x00]
0x00000114 1C40      ADDS          r0,r0,#1
0x00000116 4901      LDR           r1,[pc,#4]  ; @0x0000011C
0x00000118 6008      STR           r0,[r1,#0x00]
0x0000011A 4770      BX            lr

0x0000011C ~~20000000  DCD           0x20000000~~   <-- Needs patch

    main
0x00000120 B510      PUSH          {r4,lr}
0x00000122 F7FFFFF5  BL            inc (0x00000110)
0x00000126 4804      LDR           r0,[pc,#16]  ; @0x00000138
0x00000128 4A04      LDR           r2,[pc,#16]  ; @0x0000013C
0x0000012A 2100      MOVS          r1,#0x00
0x0000012C 6803      LDR           r3,[r0,#0x00]
0x0000012E 4608      MOV           r0,r1
0x00000130 F7FFFFAF  BL            Display_Message (0x00000092)
0x00000134 2000      MOVS          r0,#0x00
0x00000136 BD10      POP           {r4,pc}

0x00000138 ~~20000000  DCD           0x20000000~~   <-- Needs patch
0x0000013C ~~20000004  DCD           0x20000004~~   <-- Needs patch
```

Not position independent. Hard-coded pointers to addresses in the data segment as marked above need to be patched on loading, when actual location of data segment is known. Likewise, hard-coded pointer to address of function called by Display_Message needs to be patched to point to the actual function to be called.

*Name:*______________________________

b)

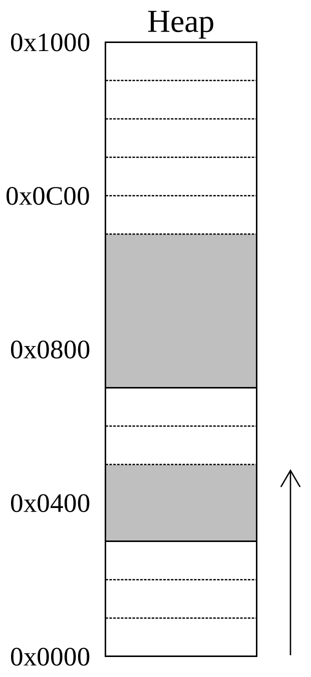
```
0x20000004 0000      DCW          0x0000
0x20000008 …         DCB          “Hello”,0
```

```
    Display_Message
0x00000040 DF2A      SVC          0x2A
0x00000042 4770      BX           lr

    inc
0x000000BC 4803      LDR          r0,[pc,#12]  ; @0x000000CC
0x000000BE 4448      ADD          r0,r0,r9
0x000000C0 6800      LDR          r0,[r0,#0x00]
0x000000C2 1C40      ADDS         r0,r0,#1
0x000000C4 4901      LDR          r1,[pc,#4]  ; @0x000000CC
0x000000C6 4449      ADD          r1,r1,r9
0x000000C8 6008      STR          r0,[r1,#0x00]
0x000000CA 4770      BX           lr

0x000000CC 00000004  DCD          0x00000004

    main
0x000000D0 B510      PUSH         {r4,lr}
0x000000D2 F7FFFFF3  BL           inc (0x000000BC)
0x000000D6 4805      LDR          r0,[pc,#20]  ; @0x000000EC
0x000000D8 4448      ADD          r0,r0,r9
0x000000DA 4A05      LDR          r2,[pc,#20]  ; @0x000000F0
0x000000DC 444A      ADD          r2,r2,r9
0x000000DE 2100      MOVS         r1,#0x00
0x000000E0 6803      LDR          r3,[r0,#0x00]
0x000000E2 4608      MOV          r0,r1
0x000000E4 F7FFFFAC  BL           Display_Message (0x00000040)
0x000000E8 2000      MOVS         r0,#0x00
0x000000EA BD10      POP          {r4,pc}

0x000000EC 00000004  DCD          0x00000004
0x000000F0 00000008  DCD          0x00000008
```

Program is fully position independent. All references to data segment are relative to R9. All absolute references to other code locations replaced by SVC style calls.

Problem 6 (5 points): Heap

Consider a 4kB heap in the state as shown below after the given sequence of `malloc()` and `free()` calls has been executed. Assume that the heap manager does not require any overhead for extra meta-data, and that the heap is allocated from bottom to top, i.e. a block always ends up being placed at the bottom of its chosen free space region. What allocation strategy (first/best/worst fit) does the heap use?



```
p1 = Heap_Malloc(1792);
p2 = Heap_Malloc(1024);
Heap_Free(p1);
p3 = Heap_Malloc(768);
p4 = Heap_Malloc(512);
Heap_Free(p3);
```

p1 and p2 malloc are the same for any fit.
Then, p3 malloc is either first or worst fit (since it ends up in the larger of the two free spaces).
Then, p4 malloc is either first or best fit (since it ends up in what is now the smaller of the two free spaces).

Hence, the only fit that satisfies all cases is first fit.

Problem 7 (5 points): CAN

Consider a CAN network with 4 microcontrollers in which microcontroller M0 periodically sends messages with ID 14 to microcontroller M1 every 31ms, and microcontroller M2 periodically sends messages with ID 4 to microcontroller M3 every 11ms. What is the maximum jitter experienced by M1 and M3 in receiving their messages? Show the result as a function of the frame delay t_f (= time to complete a single message transfer).

Message with ID4 has higher priority than message with ID14, so messages with ID14 can be delayed by up to t_f.

However, the CAN bus is not preemptive, so in the worst case, if a message with ID4 wants to be sent right after a message with ID14 has started, it will need to wait until the bus is free, so can also be delayed by up to t_f.

Max jitter in both cases is t_f.