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

Real-Time Systems / Real-Time Operating Systems

EE445M/EE380L.12, Spring 2020

Final Exam Solutions

Date: May 13, 2020

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, open notes and open web.
- No calculators or any electronic devices other than your laptop/PC (turn cell phones off).
- You are allowed to access any resource on the internet, but no electronic communication other than with instructors.
- Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided.
- For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

Problem 1	15	
Problem 2	15	
Problem 3	15	
Problem 4	20	
Problem 5	10	
Problem 6	15	
Problem 7	10	
Total	100	

Problem 1 (15 points): LDREX/STREX

Given the following semaphore implementation using LDREX/STREX:

```
OS_Wait ; R0 points to counter
   LDREX    R1, [R0] ; counter
   SUBS     R1, #1   ; counter -1,
   ITT      PL       ; ok if >= 0
   STREXPL R2,R1,[R0]; try update
   CMPPL    R2, #0   ; succeed?
   BNE      OS_Wait  ; no, try again
   BX       LR
```

```
OS_Signal ; R0 points to counter
   LDREX    R1, [R0]  ; counter
   ADD      R1, #1    ; counter + 1
   STREX    R2,R1,[R0]; try update
   CMP      R2, #0    ; succeed?
   BNE      OS_Signal ; no, again
   BX       LR
```

a) Is this a spinlock or blocking semaphore implementation?

Spinlock

b) Is this a cooperative or non-cooperative semaphore implementation?

Non-cooperative

c) If cooperative, how can it be made non-cooperative, and if non-cooperative, how can it be made cooperative?

Insert a call to OS_Suspend() before looping back on failed update in OS_Wait:

```
   CMPPL R2, #0
   BE end
  PUSH LR
  BL OS_Suspend
  POP LR
  B OS_Wait
end
  BX LR
```

d) Given the following code from the midterm, where *Dump()* can be called by multiple threads:

```
char DebugDump[DUMP_SIZE];
unsigned int DebugCnt = 0;

void Dump(char c) {
  long sr;
  sr = StartCritical();
  DebugDump[DebugCnt++] = c;
  UART_OutChar(c);
  if(DebugCnt >= DUMP_SIZE)
    DebugCnt = 0;
  EndCritical(sr);
}
```

```
long UARTSema = 1;

void UART_OutChar(char c) {
  OS_Wait(&UARTSema);
  // UART code here
  ...
  OS_Signal(&UARTSema);
}
```

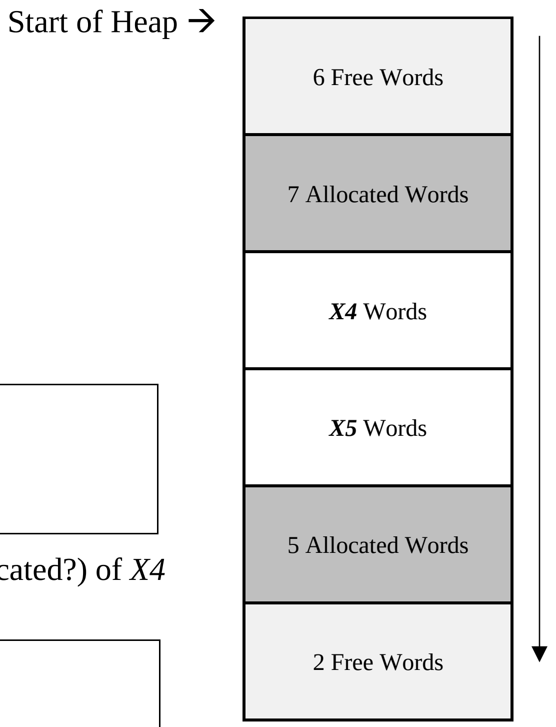
Is there still an issue with this code using the LDREX/STREX semaphore implementation above? If so, what is the issue? If not, why not?

Code is reentrant, i.e. there is no more critical section. But it instead deadlocks. OS_Wait is called with interrupts disabled. If the semaphore is not available, it will spin forever.

Problem 2 (15 points): Heap

Given below is a partially complete series of heap accesses and the corresponding heap state (not drawn to scale). Accesses to the heap are done using *Heap_Alloc(X)*, which allocates *X* **words**. Each allocated block on the heap requires 2 additional words of metadata overhead, i.e. if *X* words are requested, *X*+2 words are actually allocated on the heap. The heap is 32 words total.

```
Heap_Init();
P1 = Heap_Alloc(4);
P2 = Heap_Alloc(7);
P3 = Heap_Alloc(8);
Heap_Free(X1);
P4 = Heap_Alloc(5);
Heap_Free(X2);
P5 = Heap_Alloc(X3);
```



a) What are the values of *X1*, *X2*, and *X3*?

X1 = P2, X2 = P1, X3 = 3

b) What are the values and allocation status (free or allocated?) of *X4* and *X5*?

X4 = 2 words free, X5 = 10 words allocated

c) What heap algorithm is used? How do you know?

When P4 is allocated, it goes into slot that used to be P2, i.e. either first or worst fit.

When P5 is allocated, it goes into the second, larger slot, i.e. worst fit. So worst fit it is.

d) What is the largest *X6* that a *Heap_Alloc(X6)* can still allocate after the sequence of calls above?

Largest free block is 6 words, so X6 = 4 including overhead.

Problem 3 (15 points): File system

Assume a FAT file system with 3 files currently in it. A zero in the File Allocation Table (FAT) signifies the end of a file. The disk size is 2kB, and consists of 16 blocks. Assume that all disk metadata (directory and FAT) is stored in the first two blocks.

a) Given the following Directory and FAT, fill in the missing information.

Many solutions possible, here are listed only a few

File Allocation Table

Index	Data
0	X
1	X
2	*8 / 0 / 11*
3	6
4	12
5	0
6	7
7	*0 / 11 / 0*
8	3
9	2
10	*15 / 15 / 15*
11	10
12	9
13	0
14	0
15	14

Directory

File Name	Size	Start of file
A	4	*11 / 4 / 8*
B	8	*4 / 8 / 4*
C	1	*13*
Free space	*1*	5

b) What is the size of each disk block?

2048 / 16 = 128 bytes

c) What is the largest new file that can be created on the disk in its current state as above?

1 block = 128 bytes of free space

d) How much space does the FAT occupy on disk? What is the minimum space needed per directory entry?

FAT entries are 4 bits (0-15). A total of 16 x 4 bits = 8 bytes.

Each directory entry requires one 4-bit start index, 8 bits for size (when supporting up to 2kB files at byte granularity) and N bytes for name.

Problem 4 (20 points): 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.

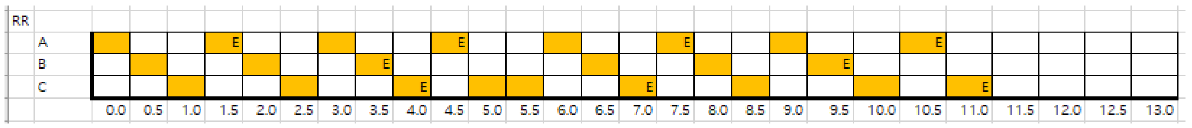
Task	**Execution time**	**Period**
Task A	10ms	30ms
Task B	15ms	60ms
Task C	15ms	40ms

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

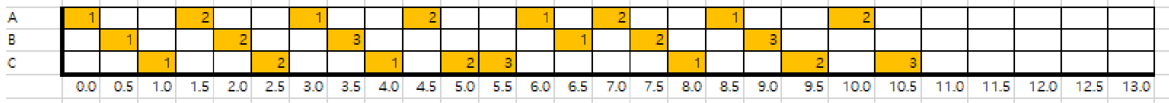
$\frac{1}{3}+\frac{1}{4}+\frac{3}{8} = 95.8\%$

b) Draw the schedule of task executions over time using a round-robin scheduler with a 5ms time slice. 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 tasks finish their execution before the start of their next period (=deadline)?

It depends on how you schedule round robin, if you schedule it with A-B-C like normal, it cannot meet the deadline.



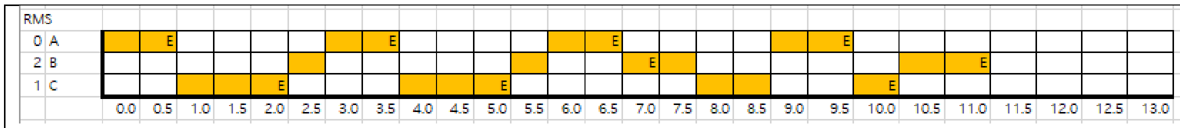
Another reasonable solution: when it misses the deadline, skip computation like below.



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

A- High, C-Mid, B-Low

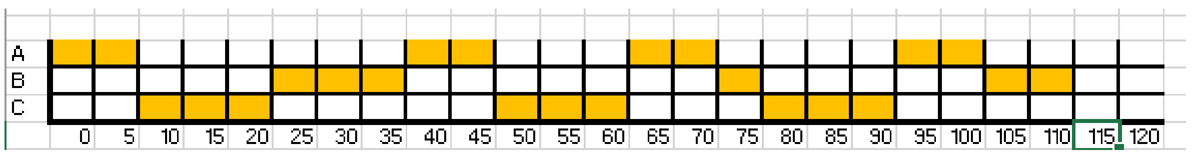
d) Draw the schedule of task executions over time with a RMS scheduler. Assume that all tasks become ready to execute, i.e. start their first period at time zero. Draw one iteration of the schedule until it starts repeating. Is the task set schedulable, i.e. do all task finish their execution before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



Task B cannot meet the deadline
Similar as solution b), skipping the computation when missing deadline is correct, too.

e) Finally, would an EDF scheduler be able to run this task set such that all tasks finish their execution before the start of their next period (=deadline)? Why or why not?

CPU utilization is below 100%, an EDF scheduler is guaranteed to be able to schedule this task set. EDF schedule looks like this:

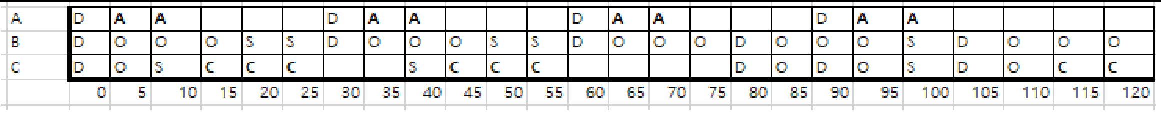


Problem 5 (10 points): Ethernet

Now consider a real-time system with three devices connected over an Ethernet network. Each device will need to send an Ethernet packets with given sizes (transmission times) periodically as indicated below. The system uses a real-time variant of CSMA/CD invented by a 445M student in which devices are assigned pre-determined and fixed backoff times as shown below. Ethernet collisions will occur if two devices start sending at the same time. Assume it takes 5ms for all devices to detect the collision after it has occurred.

Device	Transmission time	Period	Backpff Time
Device A	10ms	30ms	0ms
Device B	15ms	60ms	15ms
Device C	15ms	40ms	5ms

a) Draw the timeline of data transmission over the Ethernet bus. You can use terminology such as *S* for carrier sensing, *D* for collision detection and *O* for backoff. Draw the timeline for one hyperperiod. Do all tasks finish their transmission before the start of their next period (=deadline)?



A	D	A	A				D	A	A				D	A	A				D	A	A				
B	D	O	O	O	S	S	D	O	O	O	S	S	D	O	O	O	D	O	O	O	S	D	O	O	O
C	D	O	S	C	C	C			S	C	C	C					D	O	D	O	S	D	O	C	C
	0	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100	105	110	115	120

No, C cannot meet the deadline, and B never send packet.

b) Instead of fixed backoff times, would a dynamic backoff assignment, e.g. following an EDF-like strategy be ever a possibility in Ethernet?

No, an EDF strategy requires a central scheduler to observe all deadlines and dynamically pick the currently closest one. There is no central entity in Ethernet that has this type of system knowledge.

Problem 6 (15 points): SVC Handler

Given the following assembly code of a basic SVC handler.

```
SVC_Handler

    LDR     R12,[SP,#24]   ; Return PC




    LDRH    R12,[R12,#-2]  ; SVC instruction is 2 bytes




    BIC     R12,#0xFF00    ; Decode instruction


    MOV     R0,R12

    LDM     SP,{R1-R3}~~{R0-R3}~~      ; Get any parameters


    PUSH    {LR}

    BL      OS_Call

    POP     {LR}


    STR     R0,[SP]        ; Store return value




    BX      LR             ; Return from exception
```

a) The code has a bug. Where is the bug and what is needed to fix it? Modify the code above to fix the bug.

LR register needs to be save (PUSH LR) and restored (POP LR) before/after BL call.

*Name:*______________________________

b) Assume an *OS_Call* routine that expects the ID of the OS routine to call as the first parameter and then itself internally calls the correct OS routine as follows:

```
int OS_Call(int id, int para1, int para2, int para3) {
  switch(id) {
    case 0: return OS_func1(para1);
    case 1: return OS_func2((void*)para1, (short)para2);
    case 3: return OS_func3();
    …
  }
  return 0;
}
```

Show the modifications of the *SVC_Handler* code needed to correctly call an *OS_Call* with this behavior.

c) What is the maximum number of OS function parameters supported by the SVC handler from a) and your modified handler from b)? Can this approach be extended to support additional and even an arbitrary number of parameters? If so, how? Describe the changes necessary to the user code triggering the SVC traps and/or the SVC handler inside the OS.

Only the four parameters can generally be passed via registers R0-R3.

If R0 carries the call ID, then only 3 additional parameters can be passed via registers.

Additional parameters can be support by passing them via the stack.

No change to user code required, default AAPCS passes parameters via the stack.

Since additional data automatically pushed by the interrupt hardware when triggering the SVC will sit at the top of the stack, the SVC_Handler needs to be modified to get the parameters from further down the stack and push them onto the top of the stack.

```
  PUSH {LR,R4,R5}
  ADD R4,SP,#32   ; top of original stack before exception
  ADD R4,R4, #n*4    ; start from N parameters down
  LDR R5,[R4]       ; load parameter from stack
  PUSH {R5}         ; and push onto stack
  SUB R4,R4, #4     ; next para
  LDR R5,[R4]       ; load parameter from stack
  PUSH {R5}         ; and push onto stack
 …
  BL OS_Call
  POP {R5}         ; pop fake parameters from stack
  POP {R5}
  …
  POP {LR,R4,R5}
```

Problem 7 (10 points): Virtual Memory

In class we discussed virtual memory and paging, and that paging effectively solves the fragmentation problem similar to how indexed allocation did for filesystems, where the page table is equivalent to the index table.

a) Could a linked allocation be used for memory systems instead? Why is this a good or bad idea? What kind of hardware support would be needed in the MMU, i.e. what would the MMU need to do on every load or store for this to work?

This would replace the page table with a linked list organization of pages in memory. Memory overhead is the same.

But for a linked allocation of a process' contiguous address space, the MMU would have to traverse the linked list of pages in memory on every load and store. This would create a lot of additional memory accesses in the typical/average case. The overhead is larger than the overhead of accessing page tables in the worst case.

However, similar to normal paging, this overhead could be reduced by caching the most recent traversal results, i.e. mappings of virtual address to page indices in memory similar to a TLB. In case of TLB hits, the schemes would be equivalent.

There is, however, no advantage of doing a linked allocation. In file systems, a linked allocation removes the need for a central index table stored in memory, which has reliability problems (e.g. in case of power outages etc.). In case of virtual memory, page tables are also stored in memory, so they don't have such disadvantages.

b) How about a FAT-like system for virtual memory management?

This would replace the page table with a FAT. Again, with the same memory overhead.

This would mean organizing the page tables as linked lists. This would then require a linked list traversal to do page table lookups. Since page tables are normally also stored in memory, this would also incur additional memory accesses (that can again be avoided by caching, which will work more effectively in this case than a linked allocation).

There is also no advantage of using a FAT style. In file systems, FAT avoids the fragmentation problem in the index table. In virtual memory, we avoid that problem by not having a single shared page table, but a separate page table per process. The equivalent of that in file systems is what Linux does: each file has its own, separate index table (stored in special inodes on the disk).