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

Real-Time Systems / Real-Time Operating Systems

EE445M/ECE380L.12, Spring 2022

Final Exam Solutions

Date: May 14, 2022

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 electronic devices other than your laptop/PC (cell phones off and stowed away).
- 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	20	
Problem 2	15	
Problem 3	5	
Problem 4	20	
Problem 5	20	
Problem 6	20	
Total	100	

Problem 1 (20 points): Context Switching

Context switching overhead can limit performance on systems with many tasks and a lot of context switches. Given the following context switch implementation:

```
PendSV_Handler
  CPSID I
  PUSH  {R4-R11}
  LDR   R0, =RunPt
  LDR   R1, [R0]
  STR   SP, [R1]
  LDR   R1, [R1, #4]
  STR   R1, [R0]
  LDR   SP, [R1]
  POP   {R4-R11}
  CPSIE I
  BX    LR
```

a) How long (in ns) does it take to perform a context switch in this implementation? Assume each Load/Store takes 3 cycles, Push/Pop are 3 cycles per register, and all other instructions take 1 cycle. Also assume a 80 MHz clock and 10 cycles to enter/exit the interrupt not including pushes/pops.

*10 + 1 + 3*16 + 6*3 + 3*16 + 2 + 10 = 137 cycles*

*137 cycles * 12.5ns/cycle = 1712.5 ns*

b) A way to save context switch time is to avoid the interrupt overhead and perform context switches in a cooperative manner using *OS_Suspend* calls realized as regular subroutines, as we did in the very first context switch implementation shown in class. What are the issues and limitations with such an approach? Under what conditions can context switches be executed this way (without using an interrupt and interrupt handler)?

An interrupt handler will automatically save registers that are not accessible otherwise through a plain software context switch, specifically the PSR. As such, such an approach can only work if context switches are executed at locations where the value of the PSR does not need to be saved, i.e. is not needed by any instruction that follows after the swtich.

Otherwise, general issues with cooperative scheduling: requires user or compile to insert OS_Suspend calls and provides no guarantees against buggy or malicious code taking over the whole CPU. Also, providing real-time guarantees and scheduling is up to the user/compiler then.

c) Another way to reduce context switch overhead is to reduce the number of or completely avoid saving any registers on the stacks. Suppose we only PUSH and POP a subset (or none) of the registers R4-R11 in the *PendSV_Handler*, what are the issues and limitations with such an approach? Under what conditions can context switches be executed in this way?

Similar to the PSR, if a register is not saved during the context switch, its value will generally be lost after returning back to the same thread. Hence, any register that is actively in use, i.e. holds a value that will be needed by a subsequent instruction (i.e. that is "alive") will need to be saved at context switch time. As such, such an approach can only work if either the registers that are alive are known (and saved) in the context switch routine, or if context switches are performed at locations where it is guaranteed that only the registers that are saved by the context switch carry live values.

d) An approach that combines solutions from b) and c) is called *co-routines*. In co-routines, lightweight context switches are performed as cooperative subroutine calls that save only a subset of registers, inserted at appropriate locations in the code (determined by the user or compiler) where it is guaranteed that respective conditions are satisfied. Assuming a co-routine approach where locations are chosen such that no registers need to be saved at all, implement such a co-routine variant of *OS_Suspend*. What is the context switch delay in this case?

```
OS_Suspend
  ; need to save LR and SP
  ; at minimum
  PUSH  {LR}
  LDR   R0, =RunPt
  LDR   R1, [R0]
  STR   SP, [R1]
  LDR   R1, [R1, #4]
  STR   R1, [R0]
  LDR   SP, [R1]
  POP   {LR}
  BX    LR
```

*3 + 6*3 + 3+ 1 = 25 cycles*

*25 * 12.5 = 312.5 ns*

If counting the OS_Suspend call itself:

26 cycles, i.e. 325ns

Problem 2 (15 points): Scheduling

Below is the list of tasks for those tasks from Problem 3 in the midterm. In the midterm, you examined the context switching overhead of this schedule using standard algorithms.

Task	Execution Time	Period
T1	1 ms	5 ms
T2	1 ms	6 ms
T3	2 ms	10 ms
T4	6 ms	15 ms

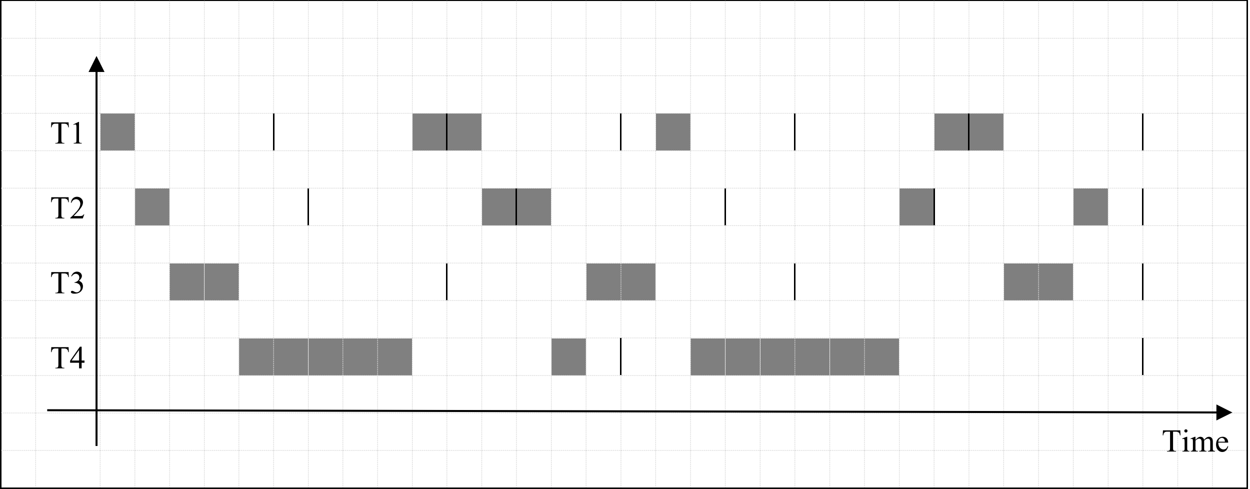
a) Are EDF or RMS optimal to minimize the number of context switches? Why or why not?

No, neither EDF nor RMS can generally guarantee to minimize the number of context switches. Its goal is to optimize for deadline violations. It does not take into account context switches and context switch overheard.

b) Draw a schedule that minimizes the number of context switches while meeting all deadlines. What is the maximum context switch overhead (in ms) such that the tasks are schedulable?

What we are ideally looking for here is a so-called non-preemptive schedule. However, there is no non-preemptive schedule for this task set that meets all deadlines. Below is a schedule that minimizes preemptions (there are other possible solutions).

Depending on whether a start of a new period of the already running task counts as a context switch or not, there are 15 or 18 context switches in this schedule. With 1ms of slack, 1/15 = 0.067ms or 1/18 = 0.056ms maximum context switch delay (note that there are schedules with 18 context switches that have at least 1ms of slack before every deadline).



Problem 3 (5 points): SVC Handler

Assume an OS that implements *OS_Kill* called from user threads via SVC traps. Shown below is a SVC Handler and a skeleton of your *OS_Kill* function. This implementation has an error in it. What is the error, how does it manifest itself and why is it happening. How can you correct it?

```
SVC Handler
  LDR R12, [SP, #24]
  LDRH R12, [R12, #-2]
  BIC R12, #0xFF00
  LDR  SP, {R0-R3}
  …
  BL OS_Kill
  …
  STR R0, [SP]
  BX LR
```

```
OS_Kill() {
  DisableInterrupt();

  // Remove from run list
  runPt->prev->next = runPt->next;
  runPt->next->prev = runPt->prev;

  // Switch to next thread
  OS_Suspend();
  EnableInterrupts();

  // Prevent running dead thread
  for(;;);
}
```

The problem is the endless loop at the end of OS_Kill. If OS_Kill is called from the SVC_Handler, the loop and hence the SVC_Handler will never return, i.e. the system will be stuck and blocked in an interrupt handler and hang.

The solution is to remove the endless loop at the end of OS_Kill.

Problem 4 (20 points): Process Loading

a) Is position-independent code and data sufficient to address relocation when dynamically loading processes like in Lab 5? Why or why not?

No, position-independence can only address relocation of the process code and data segments themselves. It does not resolve the address translation for accesses to OS code and data, the locations of which are generally not known (or preferably not hardcoded) when a process is compiled. As such, we need a separate mechanism to address references to OS code and data.

b) In Lab 5, the ELF loader performed patching (aka relocation) at load time. Was this necessary or could we have handled all of the patched cases via SVC traps and an *SVC_Handler*?

SVC traps can not be used for any OS calls that block, e.g. that wait on a semaphore (such as the STT7735_Message call in Lab 5). For such calls, we need to perform relocation via patching to call the OS routines directly.

c) Alternatively, could we have used patching to replace all SVC traps and the *SVC_Handler?*

Yes, we could use patching to resolve all OS calls. However, performing OS calls in an interrupt context has other benefits that would be lost, including protection (handler vs. thread mode and use of PSP vs. MSP). In practice, we want to perform all OS kernel calls that need access to protected hardware in an interrupt context, while carefully separating out OS calls that can run in user mode (and which in turn can use semaphores for waiting, etc.)

d) In a virtual memory system, where each process has its own private virtual address space, are position independence, patching/relocation and/or SVC traps still required?

In a virtual memory system, process code and data segments do not need to be position independent. Since every process gets its own address space, a process can be loaded at the location it was compiled for as defined in the ELF file.

However, any code and data that is meant to be shared among different processes and that is stored in pages that get mapped into the virtual address space of different processes at potentially different locations needs to be compiled to be position-independent. This applies to shared libraries such as DLLs in Windows.

This does normally not apply to OS kernel code and data, which is always mapped into the same locations in each process, but where we still don't want to or simply can not make those locations know to the processes when they are compiled. As such, OS calls still need to be resolved using traps or patching.

Problem 5 (20 points): Networking

Assume you want to implement a system transfering files between two TM4Cs that are connected via Ethernet and CAN interfaces. Each TM4C has an SD card connected through the SPI protocol.

a) When your SPI bus is running at a clock rate of 4MHz, how long does it take and what is hence the effective bandwidth for reading a contiguous block of 64MB of data from the SD card. Assume zero command-response delay (NCR=0) and an immediate data start (i.e. zero SD card latency and data packet delay).

64MB equals 64,000,000/512 = 125,000 blocks of 512 bytes each. 4Mbit/s means 2us/byte.
*Using single-block SPI transfers: 125,000 * (6 + 1 + 512 + 3) * 2us = 130,500,000 us*
*Using a multi-block SPI transfer: 6 + 1 + (512+3) * 125,000 * 2us = 128,750,007 us*

For a BW of 64MB/130.5s = 0.49 MB/s or 64MB/128.750007s = 0.497 MB/s

b) Assuming a CAN 2.0A bus running at a baud rate of 1Mbit/s using 11bit IDs and assuming no bit stuffing is needed, how long does it take and what is thus the effective bandwidth for sending 64MB of data?

8 data bytes per CAN frame means 64,000,000 / 8 = 8,000,000 CAN frames needed.

*8,000,000 * (11 + 64 + 36) * 1us = 888,000,000 us*

For a BW of 64MB/888s = 0.072 MB/s

c) Assuming an Ethernet physical layer baud rate of 10Mbit/s, how long does it take and what is this the effective bandwidth for sending 64MB of data? You can assume that there are no other machines on the Ethernet network.

1500 bytes per Ethernet frame means 64,000,000 / 1,500 = 42,667 Ethernet frames.
10Mbit/s means 0.8us/byte.

*(42,666 * (7+1+12+2+1500+4) + (7+1+12+2+1000+4)) * 0.8us = 52,087,473.6 us*

For a BW of 64MB/52.0874736s = 1.23 MB/s

d) How long does it take to first read 64MB of data from the SD card and then send it through CAN 2.0A or Ethernet, and what is the effective bandwidth for each case? Which network interface should you use for the file transfer, and why? Is there any way to reduce transfer time and achieve a higher effective bandwidth? If so, how and what is the maximum bandwidth?

Assuming single-block SPI transfers:

CAN: 130.5s + 888s = 1018.5s for a BW of 0.063 MB/s
Ethernet: 130.5s + 52.02s = 182.52s for a BW of 0.35 MB/s
-> Chose Ethernet since it is much faster.

Can increase effective bandwidth by overlapping and pipelining of SPI reads and Ethernet transfers, e.g. using DMA to read SPI in the background while sending Ethernet in the foreground. Effective bandwidth in that case is the smaller of SPI and CAN or Ethernet bandwidths, i.e. 0.072MB/s for CAN or 0.497MB/s for Ethernet.

e) Now, assume that there are 40 TM4Cs connected to each other using CAN and Ethernet, where half of them are sending a 64MB file to another TM4C at the same time. How long does it take for all 20 transfer to be complete and what is this the effective system bandwidth? Which network should you use to connect the TM4Cs, CAN or Ethernet? Justify your answer.

*For CAN, transfers of different TM4Cs will be serialized on the bus according the arbitration priorities, i.e. the total transfer time is simply 20 times the transfer time per 64MB chunk, i.e. 20 * 888s + 130.5s (for SPI reads in each TM4C) = 17,890.5s with a BW of 20*64/17890.5 = 0.071MB/s.*

For Ethernet, total transfer time will depend on how collisions on the Ethernet bus are resolved. In general, if 20 masters try to send at the same time, the Ethernet bandwidth will degrade significantly. Hence, CAN is likely the faster approach.

Problem 6 (20 points): Filesystem

You are asked to design a filesystem that supports SD cards of up to 32GiB (2^{35} bytes) size, where each block holds 512 bytes. The filesystem should support at least 60 files, where you can assume that each directory entry takes up 8 bytes.

a) How many SD card blocks need to be reserved for the directory?

*60 * 8 = 480 bytes, i.e. 1 block.*

b) You choose to use linked allocation. What is the largest file size (in bytes) you can support and why? You don't need to write out the actual number, just the expression showing how it is derived and computed is enough.

A 32GiB disk will have 2^{26} = 67,108,864 blocks.

To be able to address all blocks, 32 bit, i.e. 4 bytes per index/link are needed.

This leaves 508 bytes for usable data per block.

*Accounting for the reserved directory block, this leaves 67,108,863 * 508 = 34,091,302,404 bytes of usable disk space, which can all be allocated to a single file.*

c) You choose instead to use a FAT file system. What is the largest file size (in bytes) you can support and why? Again, it is ok to express file size as formula instead of final number.

Again, FAT table entries need to be 4 bytes wide.

The FAT needs to have 1 entry per disk block, i.e. 2^{26} entries.

Total FAT size is 2^{28} bytes. As such, the FAT requires 2^{19} = 524,288 blocks on the disk.

Together with the directory block, this leaves $2^{26} - 2^{19} – 1$ = 66,584,575 blocks of usable disk space. This translates into $2^{35} – 2^{28} – 512$ = 34,091,302,400 of usable space that is available for a single file.

*Name:*______________________________

d) Assuming that you can cache the directory, one FAT and one data block in memory, given the following sequence of file system accesses, what are the best-case and worst-case number of SD card reads for the two file systems (linked, FAT). Assume that a FAT block remains in memory until it is evicted by another FAT block load. The command *Read(file* f, int X)* should return 1 byte at the *X*th byte position in file *f*.

```
// Open file with read permission
File* f = fopen("./testfile.txt", "r")

char A = Read(f,100);
char AA = Read(f,int(A));
char B = Read(f,1000);
char BB = Read(f,int(B));
char C = Read(f,2000);
char CC = Read(f,int(C));
```

Linked:
Read directory
Read first block of f, value of A can be 0 to 255, so next read is same block.
Read second block of f, value of B again 0 to 255
Read first block of f
Read second block of f
Read third block of f
Read fourth block of f, value of C again 0 to 255
Read first block of f
-> 8 accesses

FAT, best case is all FAT entries of f are in the same block:
Read directory
Read first block of f, access twice in memory
Read FAT block that contains first link of f
Read second block of f
Read first block of f
Read fourth block of f
Read first block of f
-> 7 accesses

FAT, worst case is that FAT entries are spread across different blocks:
Read directory
Read first block of f, access twice in memory
Read FAT block that contains first link of f
Read second block of f
Read first block of f
Read FAT block that contains second link of f
Read FAT block that contains third link of f
Read fourth block of f
Read first block of f
-> 9 accesses