Midterm Exam (Remotely Proctored)

ⓘ This is a preview of the published version of the quiz

Started: Apr 2 at 10:15pm

Quiz Instructions

Your taking this exam 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.

- Open book and open notes.
- No 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.

This class uses **Proctorio (http://getproctorio.com/)** for remote proctoring of exams. To use Proctorio, you must use Google Chrome as your Web browser and install the Proctorio Chrome extension. Furthermore, your computer must have a decent amount of free RAM (close any unneeded applications) and compute power as well as a working webcam and microphone. For more information see: **Proctorio Student Guide.**

Since this an open-book/open-notes exam, I have disabled all the browser lock-down options. Proctorio will only be used to monitor and record your video, audio, screen and web traffic to check e.g. that you are not communicating with others. Only I will be able to view that information.

If you run into trouble before or during the exam, click the Proctorio extension icon (upper, right-hand corner) to access live chat help or call 1-866-948-9248.

Question 1 — 20 pts

Given the partially completed context switch routine below, fill in the 3 blanks in the context switch and the TCB fields. The TCB contains the following five fields, but you must determine the size of each variable and their order: *savedSP* (saved stack pointer), *sleeptime* (nonzero if sleeping), not*Blocked* (0 when blocked on semaphore), *next* (pointer to next TCB), and *tid* (unique thread ID).

Global variables:

```
struct TCB {
            [          ]

            [          ]

            [          ]

            [          ]

            [          ]

}
struct TCB *RunPt;       // currently running thread
uint16_t CurrentThread;  // TID of current thread
```

Context switch routine:

```
PendSV_Handler
        CPSID I
        PUSH    {R4-R11}
        LDR     R0, =RunPt
        LDR     R0, [R0]
        STR     SP, [R0, #2]
next    LDR     R0, [R0, #6]
        LDR     R1, [R0, #10]
                ____________


        BNE     next
        LDRH    R1, [R0, #14]
        CMP     R1, #1
        BNE     next
                ____________


        STR     R0, [R1]
        LDR     R2, =CurrentThread
        LDRH    R1, [R0]
        STRH    R1, [R2]
                ____________


        POP     {R4-R11}
        CPSIE I
        BX      LR
```

Question 2

4 pts

What scheduling strategy does the context switch routine in Question 1 above use? _____

Is this a cooperative or preemptive OS? _____

Question 3

15 pts

Given the following program and data memory dump of the C/compiled assembly code and stack of a *ThreadA* that is currently switched out by the OS, i.e. not occupying the CPU.

```
   void ThreadA(void){
0x00000F80 B510      PUSH          {r4,lr}
     char count = 'A";
0x00000F82 2441      MOVS          r4,#0x41
     do {
0x00000F84 BF00      NOP
       UART_OutChar(count);
0x00000F86 4620      MOV           r0,r4
0x00000F88 F000F98B  BL.W          UART_OutChar (0x000012A0)
```

savedSP --->

0x00000047
0x05050505
0x06060606
0x07070707
0x08080808
0x09090909

```
      count++;
0x00000F8C 1C64      ADDS          r4,r4,#1
    } while(count <= 'Z');
0x00000F8E 2C5A      CMP           r4,#0x5A
0x00000F90 DDF8      BLE           0x00000F86
  }
0x00000F94 E8BD4010  POP           {r4,lr}
0x00000F98 4770      BX            lr
```

0x10101010
0x11111111
0x00000046
0x4000C000
0x02020202
0x00000343
0x12121212
0x00000F8C
0x00000F86
0x81000000
0x04040404
0x00000F80

What will happen, i.e. what will get executed and what output will appear on the terminal the next time the thread is switched in by the OS? What will happen with this thread and what terminal output will be produced by it in the course of its remaining execution?

HTML Editor

B I U A A 12pt Paragraph

0 words

Question 4

10 pts

Assume a system running the following three periodic background threads (where a smaller number is higher priority).

Background Thread	Priority	Period / Frequency	Execution Time
A	1	F1 = 4Hz	E1 = 9ms
B	2	F2 = 8Hz	E2 = 8ms
C	3	F3 = 10Hz	E3 = 13ms

(a) Describe how you can run this system using only one hardware timer. Specifically, briefly describe/sketch what the timer interrupt handler will need to do at a high level (**do not** show detailed code) and what the timer reload value will need to be initialized to.

(b) What is the maximum jitter experienced in this system? By which thread and why? You can ignore any other interrupts including context switches for foreground threads.

In all case, make sure to show how you derived your results and any numbers, i.e. show how the values are computed as general functions of F1...F3 and E1...E3.

HTML Editor

12pt Paragraph

0 words

Question 5

15 pts

Given the following semaphore implementation:

```
void OS_Wait(long *s) {
 long sr;
 sr = StartCritical();
  while((*s) <= 0){
    EnableInterrupts();
    DisableInterrupts();
  }
  (*s) = (*s) - 1;
  EndCritical(sr);
}
```

```
void OS_Signal(long *s) {
  long sr;
  sr = StartCritical();
  (*s) = (*s) + 1;
  EndCritical(sr);
}
```

Is this a spinlock or blocking semaphore implementation?

Is this a cooperative or non-cooperative semaphore implementation?

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

Given the following code, where *Dump()* can be called by multiple threads for debugging purposes:

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

void Dump(char c) {
  long sr;
```

```
long UARTSema = 1;

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

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

```
  ...
  OS_Signal(&UARTSema);
}
```

What is the issue with this code given the semaphore implementation above?

How can the semaphore implementation be modified to fix this issue?

Question 6

8 pts

Given the following code for the Readers-Writers problem discussed in class:

```
ReadCount = 0;
WriteCount = 0;
OS_InitSemaphore(&mutex,1);
OS_InitSemaphore(&wrt,1);
```

```
ROpen(){
  OS_Wait(&mutex);
  ReadCount++;
  if(ReadCount==1)
    OS_Wait(&wrt);
  OS_Signal(&mutex);
}
```

```
RClose(){
  OS_Wait(&mutex);
  ReadCount--;
  if(ReadCount==0)
    OS_Signal(&wrt);
  OS_Signal(&mutex);
}
```

```
WOpen(){
  WriteCount++;
  OS_Wait(&wrt);
}
```

```
WClose(){
  OS_Signal(&wrt);
  WriteCount--;
}
```

Suppose all readers and writers use the same file. Given each of the following program states on the left side of the table, when a new thread calls *WOpen* or *ROpen*, would the new thread be blocked because or allowed to continue? Assume that for all the cases, the *mutex* is currently not held and none of the active readers or writers (i.e. readers/writers that were not blocked) is inside any of the above functions .

State	*WOpen*	*ROpen*
ReadCount=2, WriteCount=0	[Select]	[Select]
ReadCount=2, WriteCount=1	[Select]	[Select]
ReadCount=0, WriteCount=1	[Select]	[Select]
ReadCount=0, WriteCount=0	[Select]	[Select]

Question 7

20 pts

A problem with the traditional Readers-Writers solution is that writers may suffer starvation. While the writer is waiting for the semaphore, other readers may come in and the writer may never be able to enter. Modify the code to prevent this problem. Other readers should no longer be able to start using the file when a writer waits for the *wrt* semaphore. In other words, we want writers to have higher priority than readers. Please fill in the blanks to complete such an implementation. If you think a line is not necessary, please select **N/A**.

```
ReadCount = 0;
WriteCount = 0;
OS_InitSemaphore(&mutex,1);
OS_InitSemaphore(&wrt,1);
```

[Select]

[Select]

[Select]

```
ROpen(){
```

[Select]

```
  OS_Wait(&mutex);
```

[Select]

```
  ReadCount++;
  if(ReadCount==1)
    OS_Wait(&wrt);
```

[Select]

```
  OS_Signal(&mutex);
```

[Select]

```
}
```

```
RClose(){
```

[Select]

```
  OS_Wait(&mutex);
```

[Select]

```
  ReadCount--;
  if(ReadCount==0)
    OS_Signal(&wrt);
```

[Select]

```
  OS_Signal(&mutex);
```

[Select]

```
}
```

```
WOpen(){
```

[Select]

```
  WriteCount++;
```

[Select]

[Select]

```
WClose(){
```

[Select]

```
  OS_Signal(&wrt);
```

[Select]

```
  WriteCount--;
```

[Select]

[Select]

```
  OS_Wait(&wrt);
```

[Select]

```
}
```

[Select]

[Select]

```
}
```

Question 8

8 pts

With the update Readers-Writers implementation from Question 7, given each program state on the left side of the table, when a new *WOpen* or *ROpen* is launched, would the new thread be blocked or be allowed to continue? Again assume that all readers and writers use the same file and that none of the active readers and writers (i.e. readers/writers that were not blocked) is currently in any of the functions.

	WOpen	*ROpen*
ReadCount=2, WriteCount=0	[Select]	[Select]
ReadCount=2, WriteCount=1	[Select]	[Select]
ReadCount=0, WriteCount=1	[Select]	[Select]
ReadCount=0, WriteCount=0	[Select]	[Select]