Midterm Exam (Remotely Proctored) Results for Test Student

View Log (https://utexas.instructure.com/courses/1276512/quizzes/1355524/submissions/15277696/log)

ⓘ The following questions need review:

- **Question 3**
- **Question 4**

Score for this quiz: **75** out of 100 *

Submitted Apr 4 at 1:18pm

This attempt took 6 minutes.

Question 1

20 / 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 {
             uint16_t tid;

             uint32_t sp;

             struct TCB *next;

             uint32_t sleepTime;

             uint16_t notBlockec

}
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]
      CMP R1, #0

        BNE     next
        LDRH    R1, [R0, #14]
        CMP     R1, #1
        BNE     next
      LDR R1, =RunPt

        STR     R0, [R1]
        LDR     R2, =CurrentThread
        LDRH    R1, [R0]
        STRH    R1, [R2]
      LDR SP, [R0, #2]

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

Answer 1:

Correct!

uint16_t tid;

Answer 2:

Correct!

uint32_t sp;

Answer 3:

Correct!

struct TCB *next;

Answer 4:

Correct!

uint32_t sleepTime;

Answer 5:

Correct!

uint16_t notBlocked;

Answer 6:

Correct!

CMP R1, #0

Answer 7:

Correct!

LDR R1, =RunPt

Answer 8:

Correct!

LDR SP, [R0, #2]

Additional Comments:

Question 2

4 / 4 pts

What scheduling strategy does the context switch routine in Question 1 above use? Round-robin

Is this a cooperative or preemptive OS? We don't know. Dep

Answer 1:

Correct!

Round-robin

Answer 2:

Correct!

We don't know. Depends on how PendSV is triggered.

Additional Comments:

Question 3

15 / 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}
```

savedSP ---> 0x00000047

```
     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)
       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
```

0x05050505
0x06060606
0x07070707
0x08080808
0x09090909
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?

Your Answer:

The program will continue at address 0x00000F86 (PC stored second to last on the exception stack) with value 0x47 in R4 (stored at the top of the stack as pushed during context switch in PendSV_handler). Hex 0x47 is ASCII 'G". As such, the thread will continue by outputting 'G', 'H', ...

Once it reaches 'Z', the loop will exit and the thread will first pop R4 and LR from the stack and then reach it's last statement (BX LR). The LR popped from the stack (last item on the stack), however, points back to the beginning of the thread (that goes back to the way the initial stack for the thread was setup by the OS when creating the thread - R4 and LR are pushed at the beginning and popped at the end). As a result, the thread will relaunch, and it will start from 'A', 'B', ... again and loop as such forever.

Additional Comments:

Question 4

10 / 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.

Your Answer:

Answer to (a): GCD of F1, F2 and F3 = 25ms reload. Tasks 1, 2, 3 happen every 10, 5, 4 timer interrupts.

Answer to (b): Thread 3 experiences highest jitter of E1+E2 = 17ms because when all three threads line up, thread 3 must wait for both higher priority threads to complete)

Additional Comments:

Question 5

15 / 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? spinlock

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

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

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;
  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);
}
```

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

How can the semaphore implementation be modified to fix this issue? There is no fix, spinl

Answer 1:

Correct!

spinlock

Answer 2:

Correct!

non-cooperative

Answer 3:

Correct!

Insert OS_Suspend() call into OS_Wait() while-loop (between enable and disable interrupts)

Answer 4:

Correct!

Interrupts are enabled inside OS_Wait, making Dump() non-reentrant (critical section on access to DebugCnt)

Answer 5:

Correct!

There is no fix, spinlock semaphores can not be mixed with Disable/EnableInterrupts

Additional Comments:

Question 6

8 / 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	blocked ▾	continue
ReadCount=2, WriteCount=1	blocked ▾	continue
ReadCount=0, WriteCount=1	blocked ▾	blocked ▾
ReadCount=0, WriteCount=0	continue ▾	continue ▾

Answer 1:

Correct!

blocked

Answer 2:

Correct!

continue

Answer 3:

Correct!

blocked

Answer 4:

Correct!

continue

Answer 5:

Correct!

blocked

Answer 6:

Correct!

blocked

Answer 7:

Correct!

continue

Answer 8:

Correct!

continue

Additional Comments:

Question 7

20 / 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);
          OS_InitSemaphore(&rdr,1); ▾

          OS_InitSemaphore(&s2,1); ▾

          N/A ▾
```

```
ROpen(){
          OS_Wait(&rdr); ▾

  OS_Wait(&mutex);
          N/A ▾

  ReadCount++;
  if(ReadCount==1)
    OS_Wait(&wrt);
          N/A ▾

  OS_Signal(&mutex);
          OS_Signal(&rdr); ▾

}
```

```
RClose(){
          N/A ▾

  OS_Wait(&mutex);
          N/A ▾

  ReadCount--;
  if(ReadCount==0)
    OS_Signal(&wrt);
          N/A ▾

  OS_Signal(&mutex);
          N/A ▾

}
```

```
WOpen(){
          OS_Wait(&s2); ▾

  WriteCount++;
```

```
WClose(){
          N/A ▾

  OS_Signal(&wrt);
```

```
        if(WriteCount==1)  ▾

          OS_Wait(&rdr);  ▾

        OS_Signal(&s2);  ▾

  OS_Wait(&wrt);
        N/A  ▾

}
```

```
        OS_Wait(&s2);  ▾

  WriteCount--;
        if(WriteCount==0)  ▾

          OS_Signal(&rdr);  ▾

        OS_Signal(&s2);  ▾

}
```

Answer 1:

Correct!

OS_InitSemaphore(&rdr,1);

Answer 2:

Correct!

OS_InitSemaphore(&s2,1);

Answer 3:

Correct!

N/A

Answer 4:

Correct!

OS_Wait(&rdr);

Answer 5:

Correct!

N/A

Answer 6:

Correct!

N/A

Answer 7:

Correct!

OS_Signal(&rdr);

Answer 8:

Correct!

N/A

Answer 9:

Correct!

N/A

Answer 10:

Correct!

N/A

Answer 11:

Correct!

N/A

Answer 12:

Correct!

OS_Wait(&s2);

Answer 13:

Correct!

if(WriteCount==1)

Answer 14:

Correct!

OS_Wait(&rdr);

Answer 15:

Correct!

OS_Signal(&s2);

Answer 16:

Correct!

N/A

Answer 17:

Correct!

N/A

Answer 18:

Correct!

OS_Wait(&s2);

Answer 19:

Correct!

if(WriteCount==0)

Answer 20:

Correct!

OS_Signal(&rdr);

Answer 21:

Correct!

OS_Signal(&s2);

Additional Comments:

Question 8

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	blocked ▾	continue ▾
ReadCount=2, WriteCount=1	blocked ▾	blocked ▾
ReadCount=0, WriteCount=1	blocked ▾	blocked ▾
ReadCount=0, WriteCount=0	continue ▾	continue ▾

Answer 1:

Correct!

blocked

Answer 2:

Correct!

continue

Answer 3:

Correct!

blocked

Answer 4:

Correct!

blocked

Answer 5:

Correct!

blocked

Answer 6:

Correct!

blocked

Answer 7:

Correct!

continue

Answer 8:

Correct!

continue

Additional Comments:

Fudge Points: --

Final Score: 100 out of 100

Update Scores