CG2271 Real Time Operating Systems
Tutorial 9

Question 1

Assume that messages in your RTOS consist of void pointers, that sendmsg places the void pointer passed to it on a queue, and that rcvmesg returns the void pointer it retrieved from the queue. What is wrong with the following code?

```
void vLookForInputTask(void)
{
      ….
      if(!! A key has been pressed on the keyboard)
            vGetKey();
}

void vGetKey(void)
{
      char ch;
      ch = !! Get key from the keyboard;
      /* Send key to keyboard command handler task */
      sndmsg(KEY_MBOX, &ch, PRIORITY_NORMAL);
}

void vHandleKeyCommandsTask(void)
{
      char *p_chLine;
      char ch;
      while(1)
      {
            /* Wait for key to be received */
            p_chLine = rcvmsg(KEY_MBOX, WAIT_FOREVER);
            ch = *p_chLine;
            !! Do stuff with ch
      } /* while */
}
```

1. vGetKey passes the address of ch to the mailbox. But ch is a local variable that is destroyed once vGetKey exits.

2. When vHandleKeyCommandsTask gets the address to ch, the memory location used to hold ch might already be overwritten by something else

3. As a result this de-reference might access rubbish!

Question 2

This code uses the following *AMX* functions to create and use events:

```
ajevcre(AMXID *p_amxidGroup, unsigned int uValueInit,
               char *p_chTag);
```

- Creates a group of 16 events, and returns a handle to the events in *p_amxidGroup*. The events group is also given a four-character tag given in *p_chTag*. The events are initialized to "set" or "reset" depending on the value of *uValueInit*.

```
ajevsig(AMXID amxidGroup, unsigned int uMask,
         unsigned int uValueNew);
```

- Sets and resets the events in the group indicated by *amxidGroup*. The *uMask* parameter indicates which of the 16 events in the group should be set or reset, depending on the parameter *uValueNew*.

```
ajevwat(AMXID amxidGroup, unsigned int uMask, unsigned int uValue,
               int iMatch, long lTimeOut);
```

- Causes the task to wait for one or more events in the group indicated by *amxidGroup*. The *umask* parameter indicates which of the 16 events in the group the task should wait for. *uValue* indicates whether the task wants to wait for the selected events to be "set" or "reset", while *iMatch* specifies whether the task should unblock when ALL or AT LEAST ONE of the indicated events is set or reset. *lTimeOut* indicates how long the task is willing to wait.

Given the information above, rewrite the following code with semaphores.

```
/* Handle for trigger group of events */
AMXID amxidTrigger;

/* Constants for use in the group */
#define TRIGGER_MASK       0x0001
#define TRIGGER_SET              0x0001
#define TRIGGER_RESET            0x0000

void main(void)
{
   ……
   /* Create an event group with the trigger and keyboard events set */
   ajevcre(&amxidTrigger, 0, "EVTR");
   ……
}

void interruptvTriggerISR(void)
{
         /* User pulled trigger. Set the event */
         ajevsig(amxidTrigger, TRIGGER_MASK, TRIGGER_SET);
}
```

```
void vScanTask(void)
{
        ……
    while(1)
    {
        /* Wait for user to pull trigger */
        ajevwat(amxidTrigger, TRIGGER_MASK, TRIGGER_SET,
              WAIT_FOR_ANY, WAIT_FOREVER);

        /* Reset the trigger event */
        ajevsig(amxidTrigger, TRIGGER_MASK, TRIGGER_RESET);
        !! turn on hardware scanner
    } /* while */
}
```

Re-written Code:

```
OSSemaphore sem;
void main(void)
{
  ……
 OSSemCreate(&sem, 0); /* Initsemaphore to
 0 */
 ……
}


void interrupt interruptvTriggerISR(void)
{
    /* User pulled trigger. Set the event */

    OSSemRelease(&sem); /* Release the semaphore */
}

void vScanTask(void)
{
         ……
         while(1)
         {
            /* Wait for user to pull trigger */
            OSSemTake(&sem); /* Try to take a semaphore */
            !! turn on hardwarescanner
         }  /* while */
}
……
```

Question 3

What does the term “re-entrancy” mean? In particular, what does it mean when we say that a routine is “re-entrant”? Demonstrate, with an example, the problems associated with a non re-entrant routine, and how the routine can be made re-entrant. GIYF. ☺

Consider a routine AddOne:

```
void AddOne()
{
        statusCount+=1;
}
```

In assembly this would be broken down to:

```
LOAD R0, statusCount        ; Load contents of statusCount to R0
ADD R0, R0, 1               ; Increment R0 by 1
STORE statusCount, R0       ; Write updated value to statusCount
```

This routine is called by both Task1 and Task2. It is possible for Task 1 to get pre-empted while it is still inside this routine, and Task 2 gets started up. Furthermore it is possible that Task 2 also calls AddOne before Task1 completely exits it (because Task1 got pre-empted). A routine is “re-entrant” if, in this situation, it behaves exactly as though Task1 had fully exited AddOne before Task2 entered.

We can demonstrate that the example above is not re-entrant. Suppose statusCount is currently 3:

Description	Task1 code exec	Task2 code exec	Value
Task 1 calls AddOne	LOAD R0, statusCount	-	statusCount=3, R0=3
	ADD R0, R0, 1	-	statusCount=3, R0=4
Task1 gets pre-empted.			
Task2 calls AddOne	-	LOAD R0, statusCount	statusCount=3, R0=3
	-	ADD R0, R0, 1	statusCount=3, R0=4
	-	STORE statusCount, R0	statusCount=4, R0=4
Execution returns to Task1			
	STORE statusCount, R0	-	statusCount=4, R0=4

Note that the final result of statusCount=4 is wrong, because Task1 would’ve updated statusCount to 4, and then Task2 should’ve updated it to 5.

We can make addOne re-entrant by requiring a task to acquire a semaphore before entering the task. Rewrite AddOne to:

OSSemaphore   addsema=1;

```
void AddOne
{
        OSGetSema(&addsema);
        statusCount+=1;
        OSReleaseSema(&addsema);
}
```

Suppose this compiles to:

```
!! Get semaphore addsema
LOAD R0, statusCount            ; Load contents of statusCount to R0
ADD R0, R0, 1                   ; Increment R0 by 1
STORE statusCount, R0           ; Write updated value to statusCount
!! Release semaphore addsema
```

The execution trace is now:

Description	Task1 code exec	Task2 code exec	Value
Execution begins			addsema=1
Task 1 calls AddOne	!! Get semaphore	-	addsema=0
	LOAD R0, statusCount	-	statusCount=3, R0=3
	ADD R0, R0, 1	-	statusCount=3, R0=4
Task1 gets pre-empted.			
Task2 calls AddOne	-	!! Get semaphore	addsema=0
Task2 is blocked. Control returns to Task1			
	STORE statusCount,R0	-	statusCount=4, R0=4
	!! Release semaphore	-	addsema=1
Control is handed back to Task2, addsema=0			
	-	LOAD R0, statusCount	statusCount=4, R0=4
	-	ADD R0, R0, 1	statusCount=4, R0=5
	-	STORE statusCount, R0	statusCount=5 R0=5
	-	!!Release semaphore	addsema=1

We now get the correct result of statusCount=5, and AddOne is now re-entrant.

<u>Question 4</u>

Is this function re-entrant?

```
int strlen(char *p_sz)
{
   int iLength;

   iLength = 0;
```

```
    while(*p_sz != ‘\0’)
    {
          ++iLength;
          ++p_sz;
    }

    return iLength;
}
```

Yes and no.
YES:
On its own, the variables *iLength* and *p_sz* are local, and hence even if *strlen* gets pre-empted halfway, and a second task calls *strlen*, it works fine. Each call to *strlen* will have its own copy of *iLength* and *p_sz* on it's the calling task's stack. Since the RTOS manages each task's stack separately, there will therefore not be a conflict.
NO:
The problem is that *p_sz* may point to a buffer that is shared between *Task1* and *Task2*. When *Task1* is pre-empted halfway through *strlen* and *Task2* is started, it's possible that *Task2* changes the buffer that *p_sz* points to. When *Task1* resumes, *strlen* is really now counting the characters of a different string than when it first started.

Question 5

Consider the statement: “In a non-preemptive RTOS, tasks cannot interrupt each other. Therefore there are no data sharing problems amongst task.” Would you agree with this?

No. Consider two tasks Task A and Task B. Task B depends on Task A computing a particular value in *sharedData* before Task B can start its job. Task A meanwhile depends on an event occurring before it can compute *sharedData*:

Task A:

!!Does some stuff:
!! Wait on Event E
!! Compute sharedData

If Event E has not occurred, Task A BLOCKs while Task B starts.

Task B:

!! Use sharedData; /* Wrong! Task A hasn’t computed sharedData yet! */

Hence there is still some need to coordinate between two tasks. There is also an issue of interrupt handlers corrupting data that Task A and/or Task B are sharing with it.