# CG2271 Real Time Operating Systems
# Tutorial 10

Question 1

Suppose that an operating system does not have any way to see in advance if it is safe to read from a file, pipe or device (i.e. to see if a read results in a block), but allows clocks to be set that interrupt blocked system calls. Is it possible to implement a threads package in user space under these circumstances? Discuss.

**In a user-space thread package, the OS is only aware of processes, but not the threads that run within the process. On the other hand, the clock will trigger an interrupt which hands control over to the OS. Since the OS is unaware of the threads that are running within each process, it's not possible for the OS to hand control over to a new thread within the process.**

**However if the OS is able to hand control back to the user-space thread run-time (e.g. by registering a "call-back" address with the OS), then it is possible to implement the threads package since the run-time can now select another thread to run.**

**Actual practical implementation will be very complicated because of the need to save registers, stacks, program states, etc.**

Question 2

Assume that the following code is the only code in the system that uses the variable `iShareDeviceXData`. The routine `vGetDatafromDeviceX` is an interrupt routine. Now suppose that instead of disabling all interrupts in `vTaskZ` as shown below, we disable only the device X interrupt, allowing all other interrupts. Will this still protect the `iSharedDeviceXData` variable? If not, why not? If so, what are the advantages (if any) and disadvantages (if any) of doing this compared to disabling all interrupts?

```
int iSharedDeviceXData;

void interrupt vGetDataFromDeviceX(void)
{
      iShareDeviceXData = !! Get data from device X hardware
      !! reset hardware
}

void vTaskZ(void) /* Low priority task */
{
      int iTemp;
      while(1)
      {
            ……
```

```
                !!disable interrupts
                iTemp = iSharedDeviceXData;
                !! enable interrupts
                !! compute with iTemp
        }
}
```

***iSharedDeviceXData* is protected.**

**Implications of allowing other interrupts:**

**i) Higher priority interrupts can now interrupt vGetDatafromDeviceX.**

   **a. Advantage: Higher priority devices can still be serviced.**
   **b. Disadvantage: If the higher priority handler interrupts vGetDatafromDeviceX at the start, and if it takes longer than the deadline for reading device X, data from device X can be lost.**
   **c. This however does not affect iShareDeviceXData.**

**ii) The task swapper still works!**

   **a. Advantage: Can still process higher priority tasks.**
   **b. Disadvantage: vTaskZ, being a low priority task, may not be able to compute with *iTemp* for a long time.**
   **c. Disadvantage:**

**vTaskZ is a low priority task; it can be pre-empted just after it disables the interrupt for device X. If there are sufficient high priority tasks, the interrupt for device X can be disabled indefinitely, causing data from device X to be lost.**

## Question 3

Show, on the two (unrelated) C!! code fragments below running on a pre-emptive RTOS, under what circumstances they might fail, and what the effect of the failure(s) is/are going to be. You should also indicate on the code fragment how to correct the problem, if possible.

### C!! Code Fragment 1

```
int stack[MAX_SIZE];
int sp = 0;

void push(int data)
{
      if(sp < MAX_SIZE)
      {
            disable_interrupts();
            stack[sp++] = data;
            enable_interrupts();
      }
```

```
    }

    int pop()
    {
          if(sp > 0)
          {
                disable_interrupts();
                return stack[--sp];
                enable_interrupts();
          }
    }
```

**Suppose IRQ 1 has lower priority than IRQ2. In push, suppose currently sp=MAX_SIZE-1:**

- **i)** **IRQ-1 calls push. Just after the if(sp<MAX_SIZE) check (which will pass), IRQ-2 triggers.**
- **ii)** **IRQ-1 is interrupted by IRQ-2, which again checks that sp<MAX_SIZE. Interrupts are then disabled, and sp is incremented by 1. The value of sp is now MAX_SIZE.**
- **iii)** **Control returns to IRQ-1. Currently sp is MAX_SIZE, so IRQ-1 now writes to stack[MAX_SIZE]. Since the range of stack is 0 to MAX_SIZE-1, this will exceed the array and corrupt memory.**

**Similar problem with pop except that IRQ-1 writes to stack[-1]. Fix by moving disable_interrupts to above the if.**

## <u>C!! Code Fragment 2</u>

```
int sharedData1, sharedData2;

/* All of these are initialized
in main, which is omitted here */
OSSemaphore sema1, sema2;
OSMailBox mb1, mb2;

void task1()
{
   int data;
   recvmsg(mb2, &data);
   take_sema(sema2);
   !! Do some processing on sharedData2
   sendmsg(mb2, sharedData2);
   take_sema(sema1);
   !! Do some processing on sharedData
   sendmsg(mb1, sharedData);
   release_sema(sema1);
   release_sema(sema2);
}
```

```
void task2()
{
   take_sema(sema1);
   !! Do some processing on sharedData
   sendmsg(mb2, sharedData1);
   take_sema(sema2);
   !! Do some processing on sharedData2
   recvmsg(mb2, sharedData2);
   recvmsg(mb1, &sharedData1);
   release_sema(sema1);
   release_sema(sema2);
}
```

**If Task2 has a higher priority, it will take semaphores 1, take semaphore 2, do a sendmsg to mb2, then block at the recvmesg.**

**Task1 then starts, and tries to take sema2, causing it to block. It never reaches the sendmsg to mb2, which will cause task2 to unblock and release both sema1 and sema2. Thus there is a deadlock.**

**The problem can be resolved by both tasks releasing their semaphores as soon as they finish using the shared data.**

## Question 4

a. Explain the similarities and differences between processes and threads. In particular, explain why threads are more desirable than processes when we need multiple execution streams.

   **Processes and threads are similar in that they are both schedulable sequences of instructions with their own states. The difference is in what is included in the states. For processes the state includes registers, stacks, open file pointers, memory, etc, whereas for threads the state includes mostly just the registers andstack pointer.**

   **Threads are more desirable because they are lighter. I.e. there's less overheads in creating, scheduling and destroying them. However because they share memory, race and re-entrancy issues appear.**

b. Referring to the context saving and restoring macros (portSaveContext and portRestoreContext) in FreeRTOS, are tasks in FreeRTOS treated like threads or processes? I.e. given two tasks in FreeRTOS, are they more similar to threads or processes? Explain your answer.

   **Tasks in FreeRTOS are treated more like threads. When a new task is created space is allocated only to store the registers, SP and PC. There is no duplication of memory or file pointers of the parent as in the case of processes.**