Solver: Shuvam Nandi

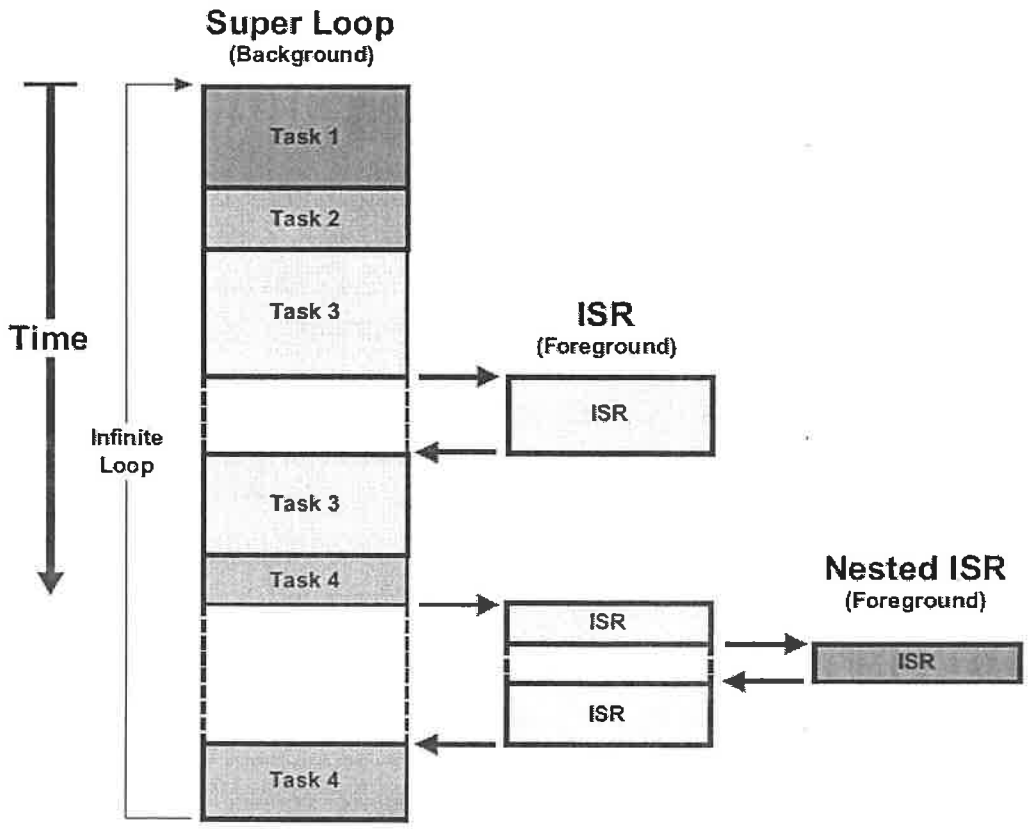
Email Address: shuvam001@e.ntu.edu.sg

1. (a)
   (i) The various features of Cortex-M3 processor are:
   - **Instruction Set**
     Cortex-M3 can execute only the mixed 16/32 bits Thumb-2 (T2) instructions, which consist of the classic Thumb 16-bit instructions and Thumb-2 32-bit instructions. Many common operations could be executed using a single 16-bit Thumb instruction.
   - **Operating Modes**
     Cortex-M3 supports two operating modes:
     1. Handler Mode
        - Mode when program is running in foreground
        - Entered as a result of an exception
        - Always executes with Privileged access level, where there is unrestricted access to all system resources
     2. Thread Mode
        - Mode when program is running in the background
        - Entered upon reset, and can be re-entered as the result of return from an exception
        - Could execute in both Privileged and User access levels
   - **Access Levels**
     Cortex-M3 supports two access levels:
     1. Privileged Access
        - Available when executing in both Handler and Thread modes
        - State into which the processor enters upon reset where processor has unrestricted access to all system resources and can use all supported instructions
     2. User Access
        - Available when executing in Thread mode only
        - Process restricted from accessing certain system resources, e.g. configuration registers for system
        - Commonly used by application programs launched by the operating system
   - **Stacks**
     Cortex-M uses a full descending stack. Cortex-M implements two stacks, the main stack and the process stack, with independent copies of the stack pointer. In Handler mode, the processor always uses the main stack, whereas in Thread mode, the CONTROL register controls whether the processor uses the main stack or the process stack.

(ii) The Cortex-M3 processor supports up to 255 exceptions, with the first 15 exception numbers (1 to 15) reserved for System exceptions. The rest (16 to 255) can be used by vendor-specific interrupt signals, which are asserted indirectly through the Nested Vector Interrupt Controller (NVIC).

Most high-volume and low-cost microcontroller-based applications (e.g. microwave ovens, toys, etc.) are designed as foreground/background systems. In this, the **foreground** typical comprises the interrupt service routines, or in the case of ARM processor, the exception handling routines. These routines handle asynchronous external events in a timely manner. In this mode, the system would require to be running in Handler mode and with Privileged Access level. The stack in use would always be the main stack. The **background** is an infinite loop (super-loop) that uses the remaining CPU cycle to perform less time critical tasks. It keeps running until it is interrupted because of an exception or interrupt. In this execution mode, the system would most likely be running in Thread mode with User or Privileged Access level. Privileged access might be given to perform some configurations or setup. If it is running with User Access level, process stack would be used; otherwise main stack would be in use.

The following diagram shows a foreground-background system with servicing of interrupts:



During power up, Cortex-M3 runs in Thread mode with Privileged access level, to enable setting up various registers for configuration. When an exception (i.e. interrupt) occurs, the background loop is halted, and the processor jumps to the foreground. The operating mode, access level and stack are changed to be able to service the exception. The current state of the system is stored in the stack frame so that the system could return to the same state once the exception is serviced.

(b)

The relevant registers of the Cortex-M3 processor and their respective functions are as follows:

- General Purpose Registers: **R0-R12**
  Registers **R0** to **R7** are also known as low registers, and can be used by both 16-bit and 32-bit Thumb-2 instructions
  Registers **R8** to **R12** are also known as high registers, and can only be used by 32-bit Thumb-2 instructions only
- Stack Pointer (SP): **R13**
  R13 is used as the Stack Pointer which points to the active stack which could either be SP_main or SP_process
- Link Register (LR): **R14**
  R14 stores the address of the return address when the program execution jumps to a procedure/function at a specific register address
- Program Counter (PC): **R15**
  R15 is used as the register which stores the address of the instruction to be read and executed

(c)

The various features and techniques provided by the Cortex-M3 processor to enable efficient handling of multiple interrupts are as follows:

- **Interrupt Handling Schemes**
  Cortex-M3 provides three schemes to handle interrupts:
  - Non-nested Interrupt
    Interrupts are handled separately and sequentially. When entering the ISR, all interrupts are first disabled, after which the incoming request is processed. After the ISR completes its task, interrupts are re-enabled, and control is returned to the main application. This is the simplest method and is not suitable for real-time sensitive applications commonly found in embedded systems.
  - Nested Interrupt
    In this scheme, another interrupt (typically of a higher priority) is allowed to invoke when servicing an interrupt. ISR re-enables

interrupts before the handler has fully serviced the current interrupt. Multiple interrupts can hence be handled in a nested manner, with the latest one (with high priority) getting serviced first.

- Re-entrant Nested Interrupt
  In this scheme, multiple interrupt invocations are allowed, like the conventional nested version, but all are of the same origin, i.e. they occur in a recursive manner. Once the re-entered (latest) invocation completes, the previously invoked instance will continue execution.

- **Interrupt Priority**
  Cortex-M3 processor incorporates a Nested Vector Interrupt Controller (NVIC) to support exceptions and interrupts handling, including prioritized exceptions and nested interrupts. With the interrupt controller, interrupt prioritization can be achieved in hardware or software.
- **Disable Interrupts**
  Cortex-M3 allows disabling interrupts using the Priority Mask (PRIMASK) and Base Priority Mask (BASEPRI) registers.
- **Tail-chaining**
  This mechanism speeds up interrupt servicing by skipping the overhead of state saving and restoration between interrupts.
- **Late-arriving**
  Cortex-M3 uses preemption to ensure efficient interrupt handling. If a higher priority interrupt occurs during the state saving (within the first 12 cycles) of another lower priority interrupt/exception the existing context state saving will continue, but vector fetching is re-started using the vector of the late-arriving higher priority interrupt. On exiting this handler of the late arriving interrupt, the normal tail chaining rule applied to service the lower priority interrupt that was pending.

2. (a)

```
#include <stdio.h>
int main()
{
    float x = 0.4, y = 0.8;    Line 4
    if (x && y > 0.8)          Line 5
        printf("Pass");        Line 6
    else                       Line 7
        printf("Fail");        Line 8
    return 0;
}
```

To perform the AND operation in line 5, x and y must be binary numbers.

Converting the floating numbers x and y in line 4 to their IEEE standard representations, 0.4 = 0100 and 0.8 = 1000.

Therefore, x && y = 0100 && 1000
= 0000

Since 0000 < 1000 (0 < 0.8), the `if` condition in line 6 returns false and the execution goes to the else statement. Thus, the output result is **Fail**.

(b)
Embedded systems programming is different from developing applications on a desktop computers. Key characteristics of an embedded system as compared to PCs are:

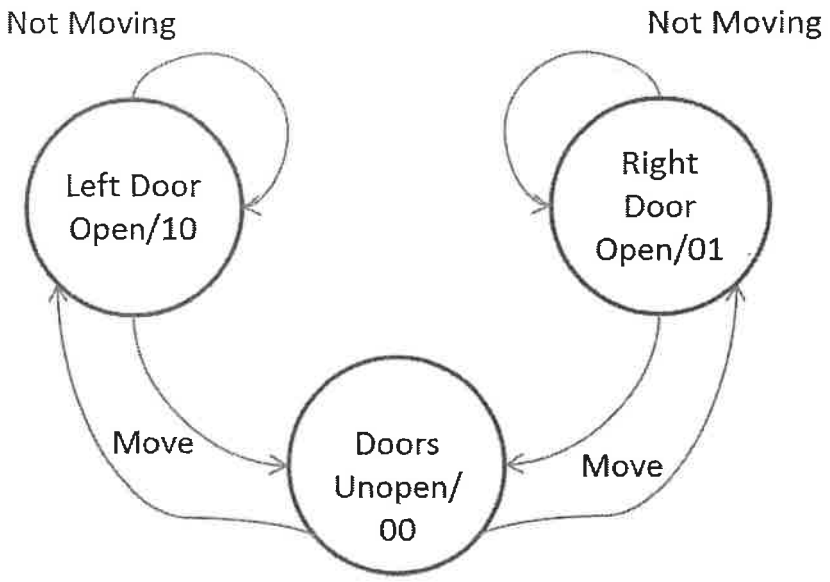
- Embedded devices have resources constraints (limited ROM, RAM, stack space and processing power)
- Components are smaller and less power consuming in embedded devices
- More tied to the hardware

Two distinctive characteristics of embedded C based software development as compared conventional C based software development are:

1. **Code Size:** Governed by available programming memory and use of programming language
2. **Code Speed**: Governed by the processing power and timing constraints

(c)
(i) Assuming that the train starts at station number 1, where the right door would be open (initial state: Right Door Open). The scenario can be represented by the following table:

State	Input		Next State	Output	
	Train Status	Station		Left Door Open	Right Door Open
Left Door Open	Not Moving	Even	Doors Unopen	Yes	No
Right Door Open	Not Moving	Odd	Doors Unopen	No	Yes
Doors Unopen	Moving	-	Left Door Open / Right Door Open	No	No



Above is a diagram representing the described Finite State Machine.

(ii) This design is based on Moore Finite State Machine, since the output of the FSM only depends on the current state of the machine and not on the input to the machine.

3. (a) x+=35

No, the above statement cannot be executed as an atomic operation. This operation actually represents addition of 35 to the variable x and storing it into x again. In a processor which follows load/store architecture, this operation can be broken down into four steps:

1. Load constant 35 into register 1
2. Load variable x from memory into register 2
3. Add values in registers 1 and 2 into register 2
4. Store register 2 value to memory location of x

Thus, it is incorrect to say that this is an atomic operation.

(b)

A reentrant function is one if it can be interrupted while already in the process of executing. That is, a function is reentrant if it can be interrupted in the middle of execution (for example, by a signal or context switch) and invoked again after the interruption execution completes.

Yes, the given function is reentrant. A context switch might occur when `compute_square()` is computing `num1 * num1`. It uses local variable num1 and for each thread there would be a separate copy of the variable. Such a function can be

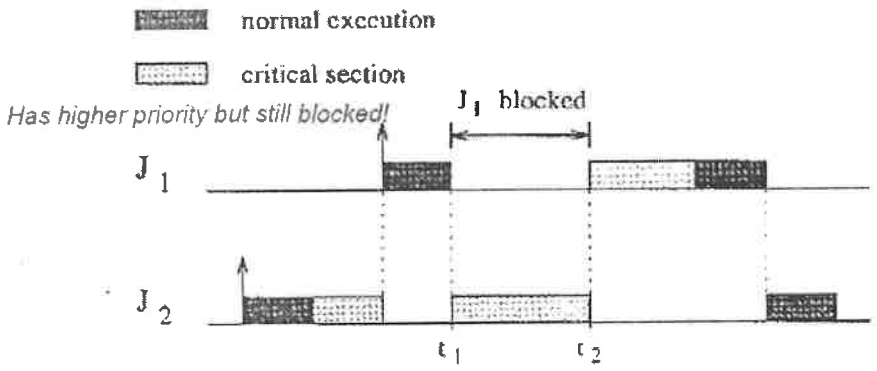
invoked simultaneously by multiple threads if and only if each invocation references or provides unique data and inputs. It follows the following guidelines of being a reentrant function:

- Does not access mutable global or function-static variables.
- Does not self-modify code.
- Does not invoke another function that is itself non-reentrant.

(c)

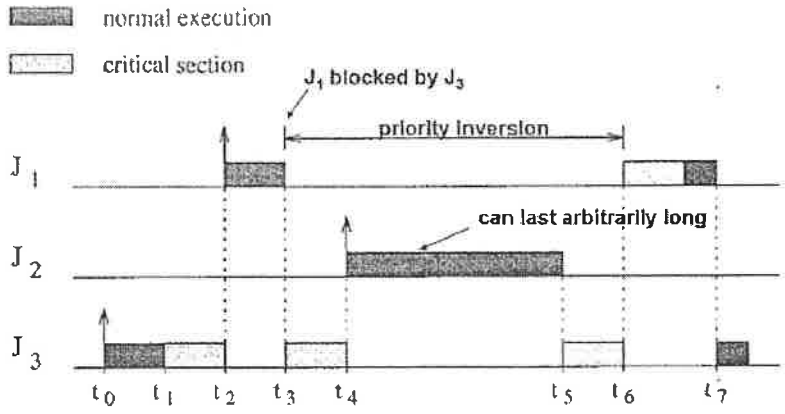
Priority inversion is a problematic scenario in task scheduling, where unavoidable blocking causes a higher priority task getting indirectly pre-empted by a lower priority task, effectively "inverting" the relative priorities of the two tasks.

In the diagram below, J1 has higher priority than J2. The execution of critical section of J1 is stalled due to the execution of critical section of J2.



In a system running multiple threads with different levels of priority, a medium-priority thread preempts a low-priority thread during inversion, and acquires a resource which a higher-priority thread requires for execution. This delays the thread with the higher priority. It is called "unbounded" because it will persist as long as the medium-priority thread has all the resources it needs to continue running, which is unrelated to the resource shared by the other two threads and can be unpredictable.

In the diagram below, J2 is a task of medium priority which is blocking the highest priority task of the system J1 to be blocked without a bound, since it can hold the resource for an unbounded period of time.



(d) Two schemes used by an Operating System to prevent priority inversion are:

- **Priority Inheritance Protocol (PIP):** In PIP, whenever a thread fails to acquire a resource, the OS looks up which thread owns the resource. If the owner has a priority lower than the thread now blocked, the OS temporarily raises the priority of the owner to match that of the blocked thread until the resource is released.
- **Priority Ceiling Protocol (PCP):** Priority of the low-priority thread is raised immediately when it acquires a shared resource and restored to its original value when it releases the resource. The temporary priority is a value predetermined by the programmer as the highest among all the threads that access the same resource and is referred to as the '**priority ceiling**'.

4. (a)

(i) Given, Task A was created with priority level 3 before Task B with priority level 6. init is initialized with a value 1 and cntr with a value 0. The output of the program would be:

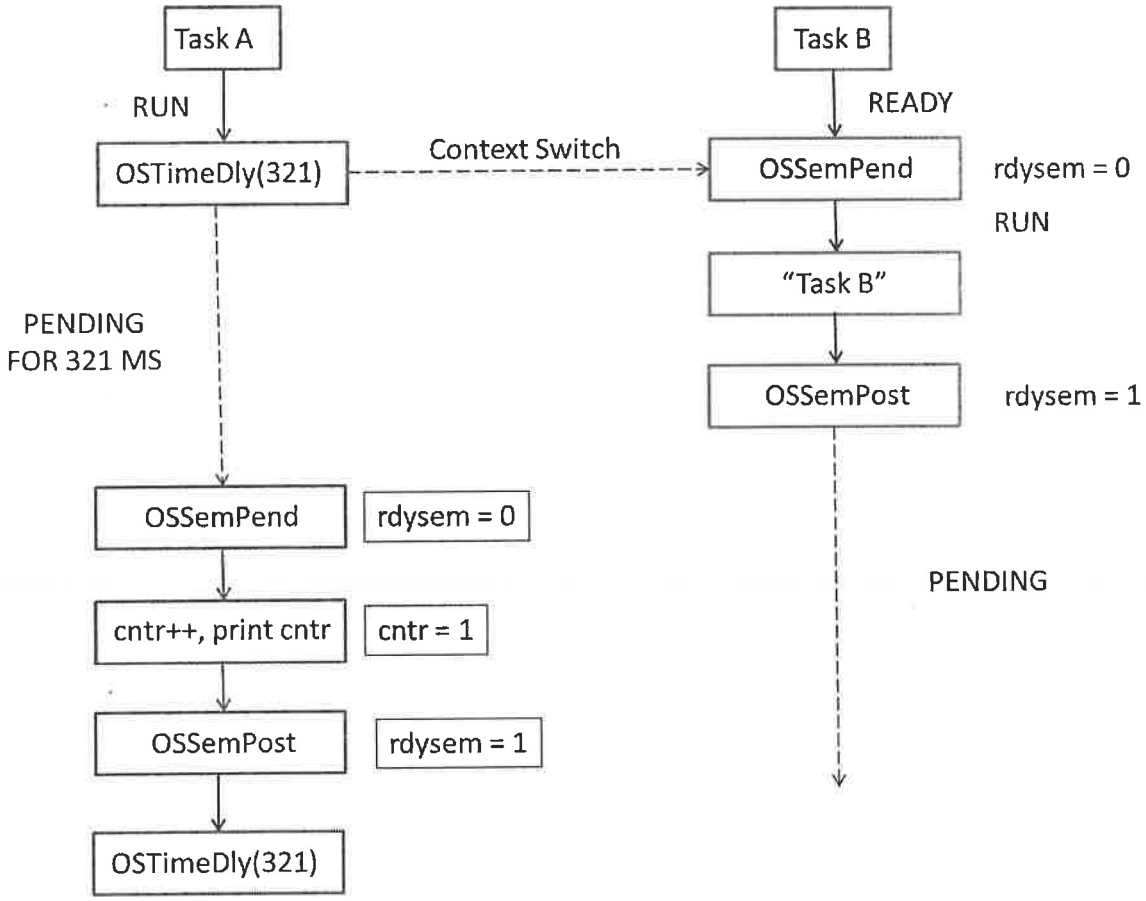
Task B1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 and so on...

This is because Task A has higher priority than Task B, and will run first. After entering the while() loop, it will give up the CPU to task B because of the OSTimeDly() call. This will cause Task B to start running, and since init is 1, it will print "Task B" and make init 0. Now, Task B will execute OSSemPend and acquire the semaphore rdysem. If Task A becomes ready to execute now, it will have to wait for Task B to finish its critical section since semaphore rdysem is acquired by Task B. Even if Task A gets to run, it will not be able to print "Task A" since init is 0.

When Task B releases the semaphore, Task A will acquire the semaphore and get to run, increasing cntr by 1, making it 1. Task A will then print the value of cntr. This follows and cntr's value increases by 1 in each execution of Task A.

This will repeat until cntr becomes 5, in which case the value of cntr will be set to 0 by Task B. Therefore, the value of cntr increases from 0 to 5, then becomes 0 again. This repeats endlessly.

(ii) The below diagram shows the state-transition diagram to show the changes in the states of both TaskA and TaskB:



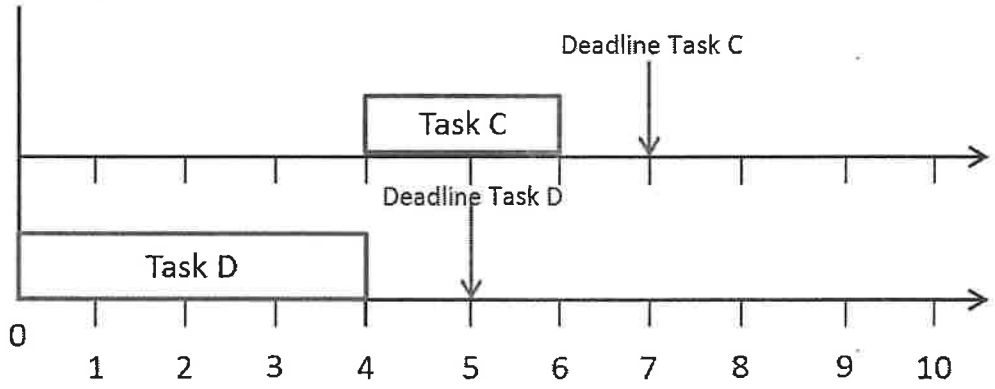
(iii) The value of cntr variable starts from 0. The execution begins with Task B, followed by Task A, then Task B again, followed by Task A once more. This pattern of execution keeps repeating. Task A increases the value of cntr in each exection while Task B checks if cntr==5, and if it is, it sets cntr to 0. The following table shows how the value of cntr increases with the execution of tasks A and B.

Task A	Task B
1 (1st execution)	0 (1st execution)
2 (2nd execution)	1 (2nd execution)
3 (3rd execution)	2 (3rd execution)
4 (4th execution)	3 (4th execution)
5 (5th execution)	4 (5th execution)
1 (6th execution)	0 (6th execution)

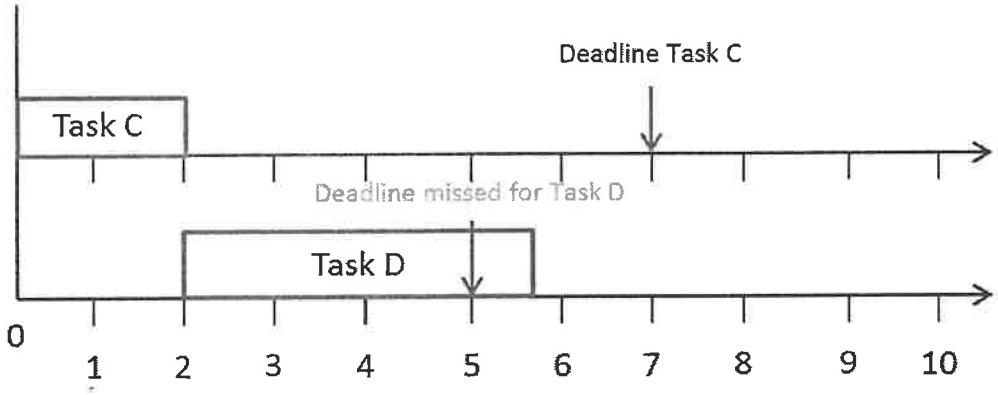
When cntr becomes 5, cntr will be set to 0 by Task B. Therefore, the value of cntr increases from 0 to 5 then becomes 0 again. This repeats endlessly.

(b)
Given two tasks,
Task C: Computation Time = 2 sec, Deadline = 7 sec
Task D: Computation Time = 4 sec, Deadline = 5 sec

According to Deadline-driven scheduling, higher priority should be assigned to Task D and a lower priority should be given to Task C. This is because Task D has an earlier deadline at time = 5 seconds. Following this schedule, Task D will run first and finish execution at time = 4 seconds, and Task C will finish at time = 6 seconds, and both tasks will be able to meet their respective deadlines. This is shown in the timing diagram given below:



If this were not followed, and Task C was allowed to run first, the deadline for Task D would have been missed, as shown below:



For reporting of errors and errata, please visit pypdiscuss.appspot.com
Thank you and all the best for your exams! ☺