NUS
National University
of Singapore

RTOS & RTX

Ravi Suppiah
Lecturer, NUS SoC

Content of this Module

- We cover the following topics in this module

    - Why RTOS
    - Compare with Super loop
    - RTX
    - CMSIS

Real-Time Operating Systems

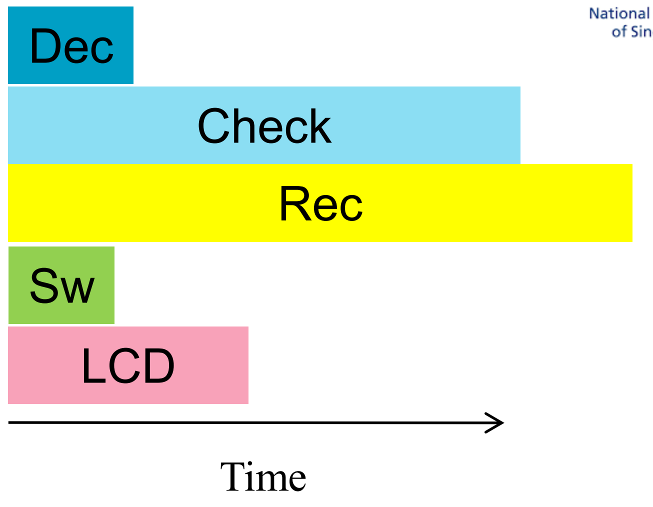
- Real-time operating systems (RTOS) are designed to meet harsh timing constraints
  - Hard real-time – critical tasks which have to be completed on time
  - Soft real-time – may continue finishing the task even missing the deadline
- Industrial applications: robots, aircraft control …
- Key design requirements:
  - Predictability and determinism
  - Speed – fast enough while keeping high predictability and determinism
  - Responsive to user control
  - Fail-safety
  - Advanced scheduling and memory allocation

Why RTOS on embedded systems?

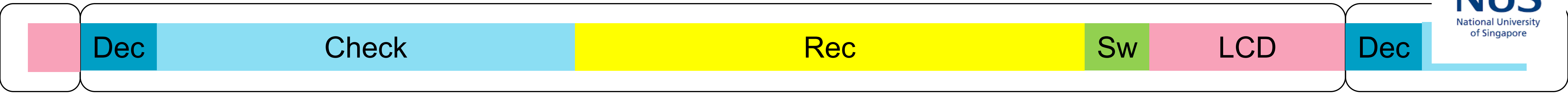
- Although it is possible to implement everything in a huge sequential loop…
  - Uses lengthy interrupt service routine (ISR)
  - Needs to keep synchronization between ISRs
  - Poor predictability (nested ISRs) and extensibility
  - Change of the ISR or the Super-Loop ripple through the entire system
- RTOS: all computation requests are encapsulated into tasks and scheduled on demand
  - Better program flow and event response
  - Multitasking
  - Concise ISRs thus deterministic
  - Better communication
  - Better resource management
  - Easier to develop applications

A case in point

- GPS based Speed Camera Alarm and Moving Map
  - Sounds alarm when approaching a speed camera
  - Display's vehicle position on LCD
  - Also logs driver's position information
  - Hardware: GPS, user switches, speaker, LCD, flash memory
- Tasks:
  - Dec: Decode GPS sentence to find current vehicle position.
  - Check: Check to see if approaching any speed camera locations. Takes longer as the number of cameras increases.
  - Rec: Record position to flash memory. Takes a long time if erasing a block.
  - Sw: Read user input switches.
  - LCD: Update LCD with map.
- How to implement in a super loop?
  - Run tasks in the same order every time?
  - Allow preemption?



Super-loop



- Simple but…
  - Always run the same schedule, regardless of changing conditions and relative importance of tasks.
  - All tasks run at the same rate. Changing rates requires adding extra calls to the function.
  - Maximum delay is the sum of all task run times. Polling/execution rate is equal to 1/maximum delay.
- What if we receive GPS position right after Rec starts running?
- Delays
  - Have to wait for Rec, Sw, LCD before we start decoding position with Dec.
  - Have to wait for Rec, Sw, LCD, Dec, Check before we know if we are approaching a speed camera!

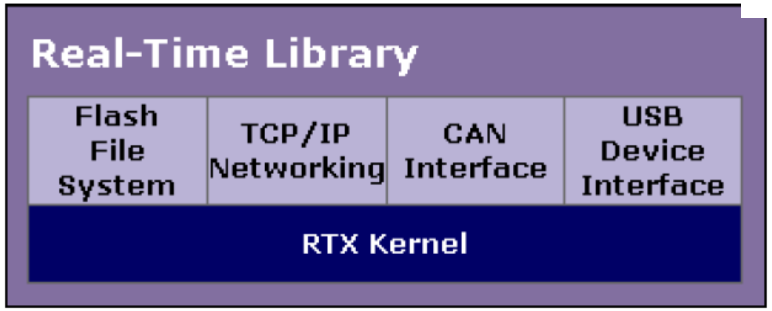
```
while (1){
    Dec();
    Check();
    Rec();
    Sw();
    LCD();
}
```

RTX

- Royalty-free, deterministic, open source RTOS
- High-Speed real-time operation with low interrupt latency
- Flexible Scheduling: round-robin, pre-emptive, and collaborative
- Small footprint for resource constrained systems
- Compatible with ARM cores (from ARM7, ARM9 to Cortex-M processors) and software tools (Keil MDK-ARM)
- Support for multithreading and thread-safe operation
- Kernel aware debug support in Keil MDK-ARM
- Dialog-based setup using µVision Configuration Wizard

RTX Structure

- Keil Real-Time Library (RTL)
  - RTX Kernel
  - Flash file system
  - Networking
  - CAN interface
  - USB device interface



Real-Time Library

Flash File System

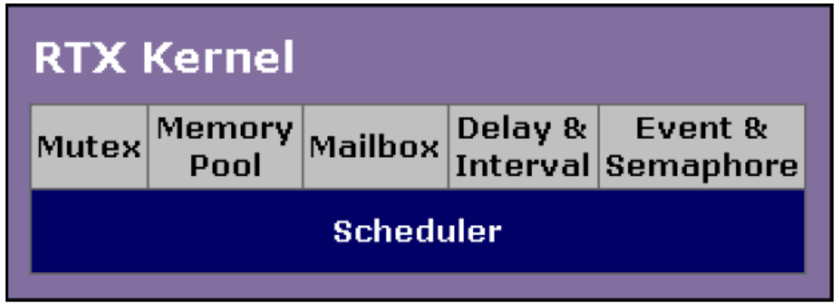
TCP/IP Networking

CAN Interface

USB Device Interface

RTX Kernel

- RTX Kernel
  - Scheduler is the core of the RTX kernel
  - Supports for mutex, memory pool, mailbox, timing functions, events and semaphores



RTX Kernel

Mutex

Memory Pool

Mailbox

Delay & Interval

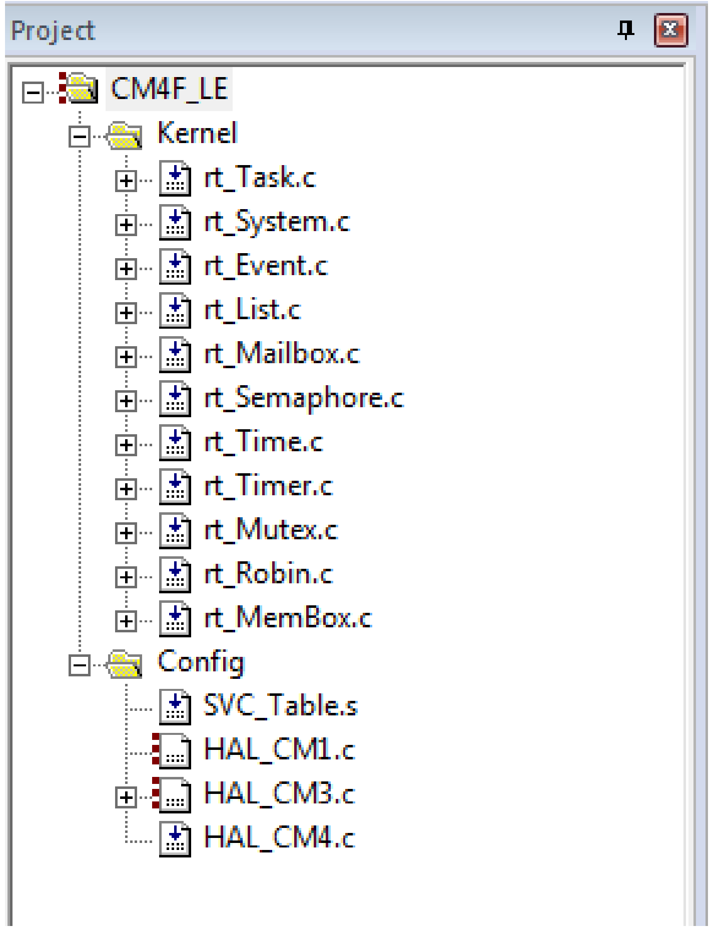
Event & Semaphore

Scheduler

Source code of RTX

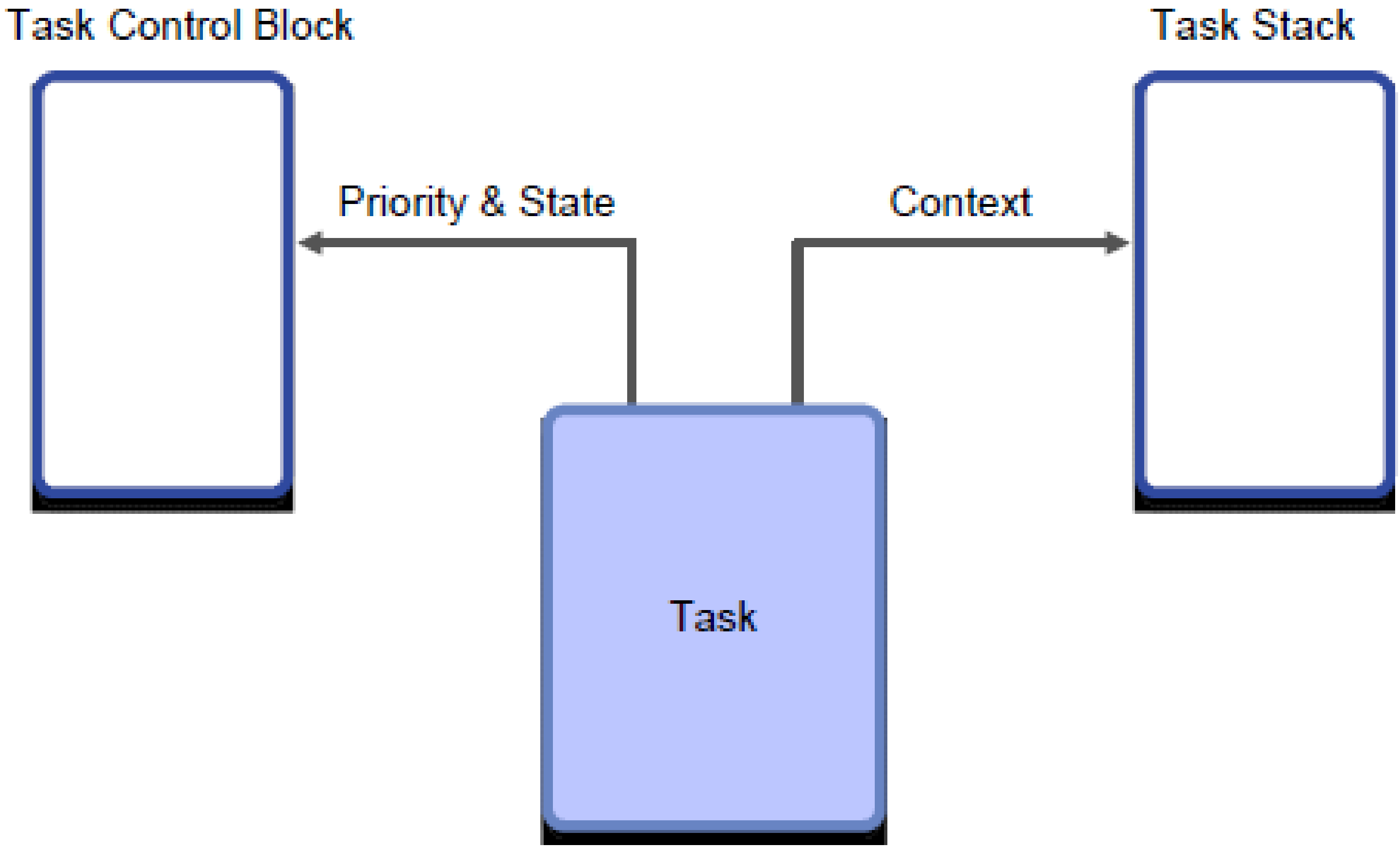
- By default C:\Keil_v5\ARM\RL\RTX\SRC\CM (Keil uVision 5)

- Or through the project file RTX_Lib_CM located in C:\Keil_v5\ARM\RL\RTX



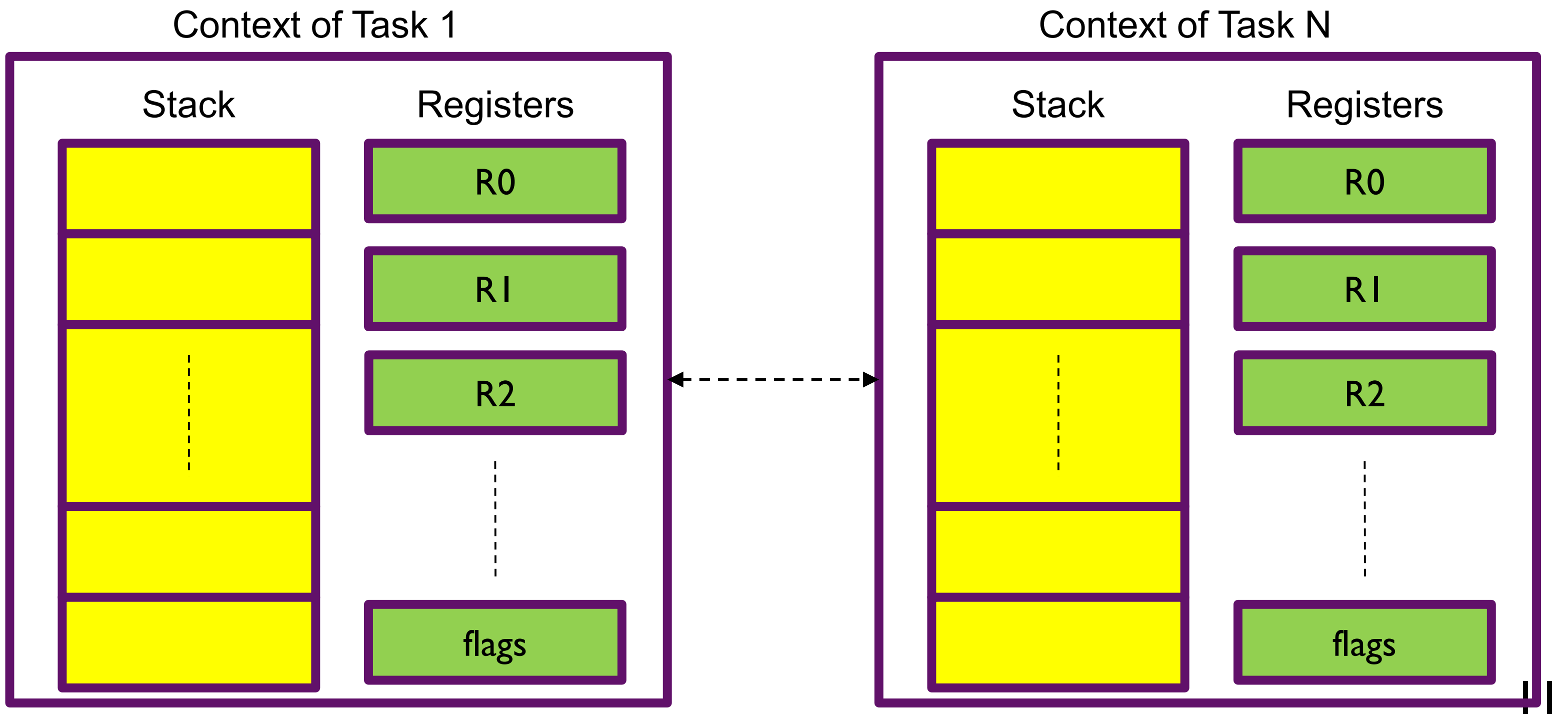
```
CM4F_LE
  Kernel
    rt_Task.c
    rt_System.c
    rt_Event.c
    rt_List.c
    rt_Mailbox.c
    rt_Semaphore.c
    rt_Time.c
    rt_Timer.c
    rt_Mutex.c
    rt_Robin.c
    rt_MemBox.c
  Config
    SVC_Table.s
    HAL_CM1.c
    HAL_CM3.c
    HAL_CM4.c
```

Task Creation and Deletion



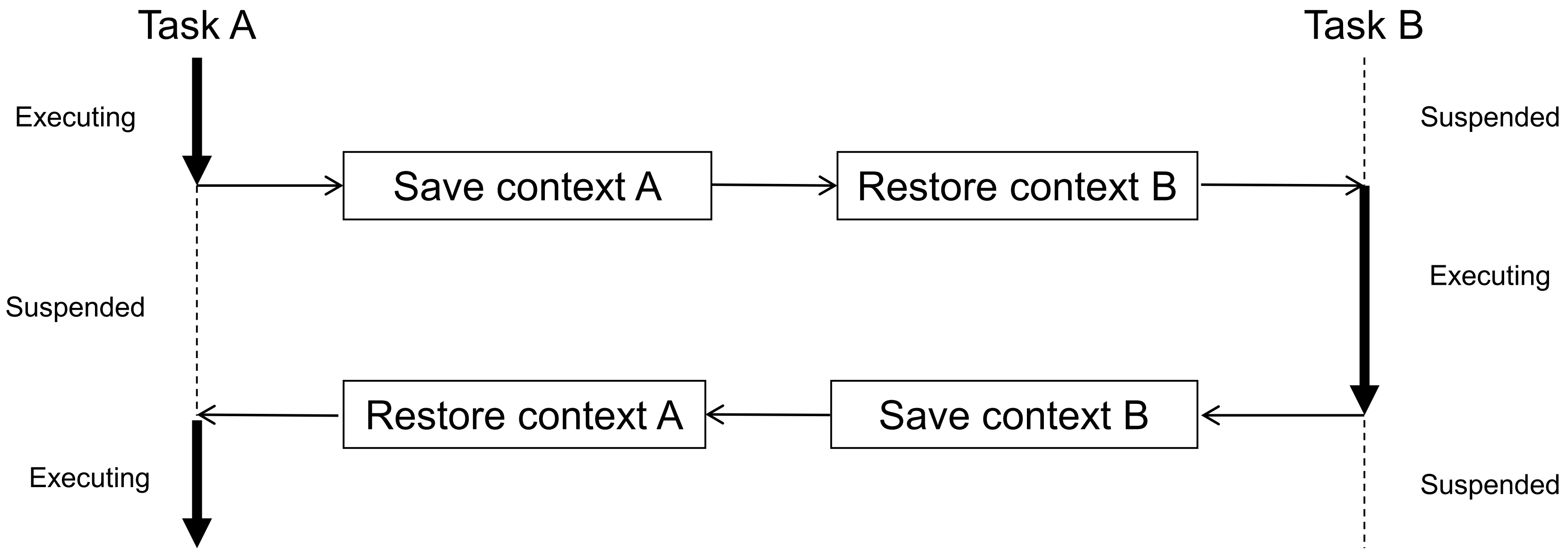
Task Context

- The context includes memory for the thread's stack and copies of the contents of all the CPU registers used by the thread.



Context Switching

- A context switch from Thread A to Thread B first saves all CPU registers in context A and then reloads all CPU registers from context B.
- Since the CPU register set includes the stack pointer (SP) and the program counter (PC), reloading context B in effect causes a reactivation of thread B’s stack and a return to where it left off when it was suspended.

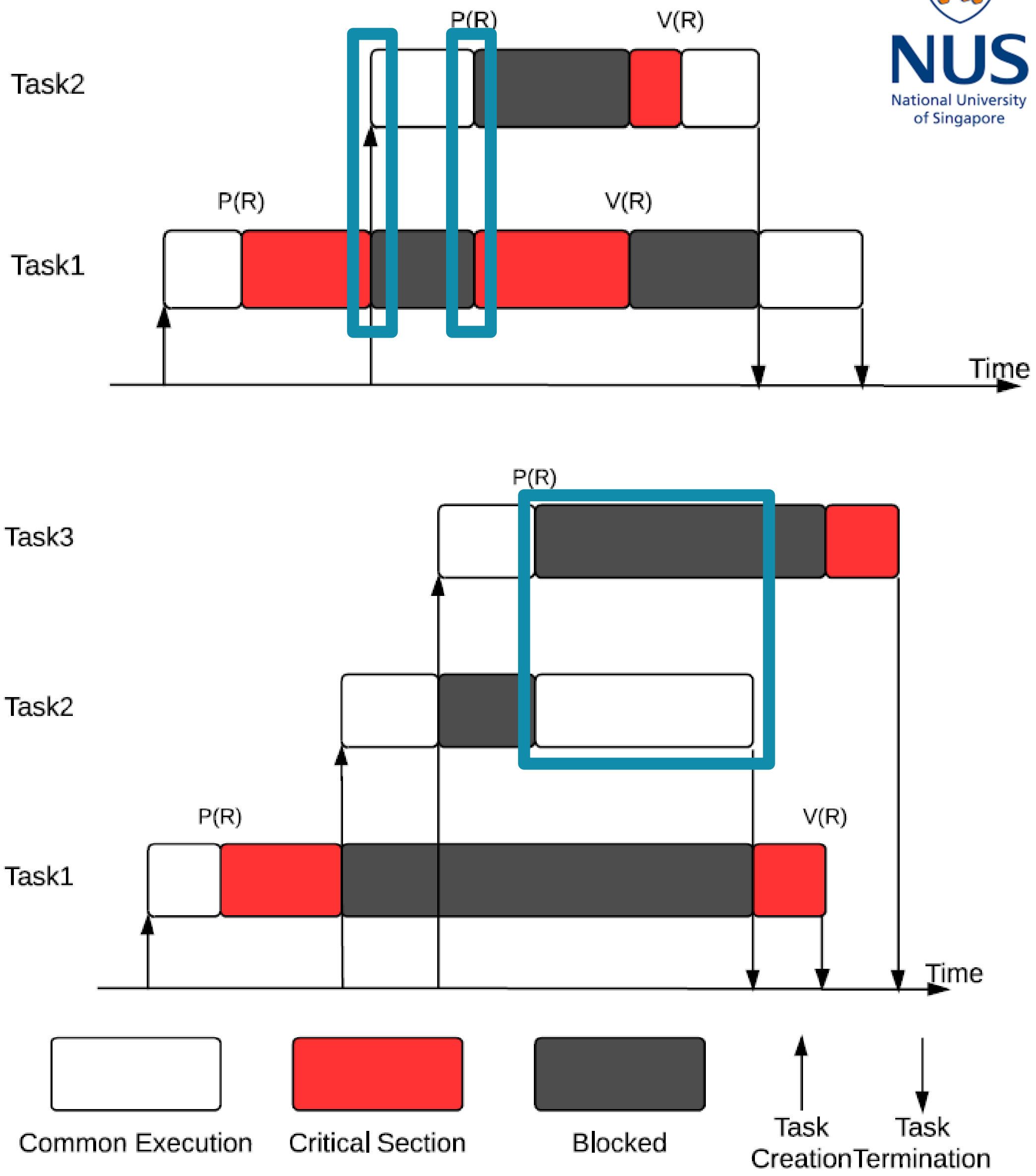


Prioritized Scheduling

- Prioritized Scheduling means that task with higher priority should be dispatched and can preempt lower priority tasks
- The problem is dependency. For instance, can a task in its critical section be preempted?
  - If no, any problem?
  - If yes, any problem?
  - The point of prioritized scheduling is based on the emergency level of the task, instead of whether it's in a critical section. But you have to pay extra attention to critical sections as otherwise they may cause conflict between tasks.
- **Priority inversion**

Preemptive Critical Section

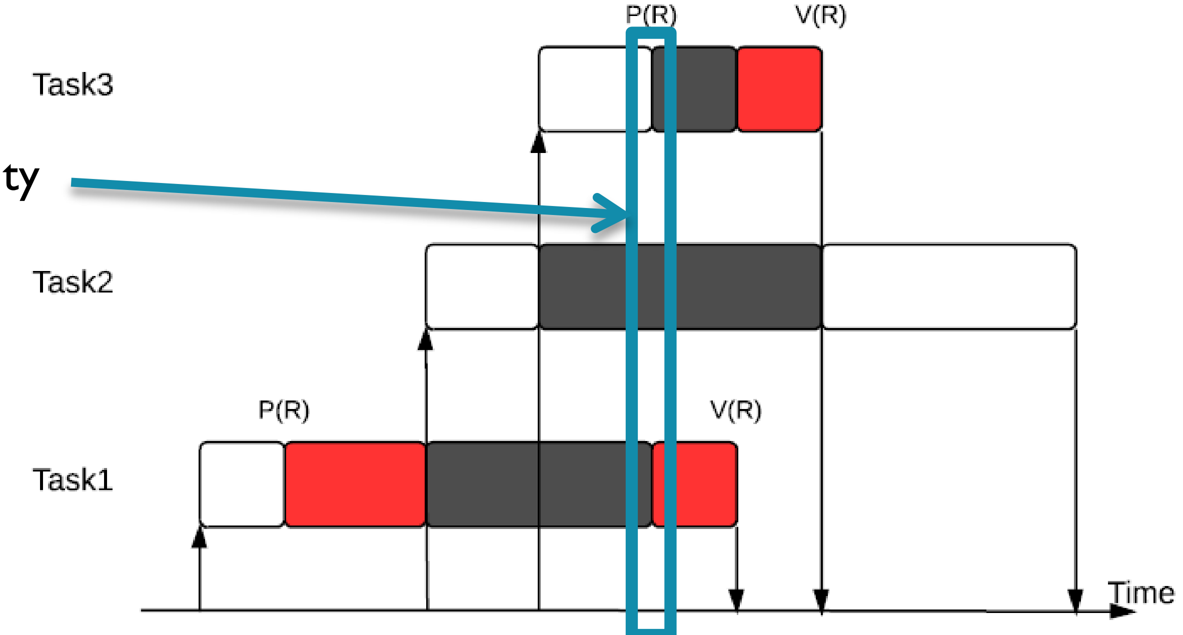
- Priority(T_N)>Priority(T_M),if N>M
- Seems okay?
- But what if a higher priority task would like to enter the same critical section? P(R) blocks task 2 if task 1 has the mutex (recall the requirements for mutex). Still acceptable as contaminating CS may cause even worse disaster, better avoid that and let task 1 finish first.
- Blocking time seems to be bounded by the maximum length of critical section of lower priority tasks. But no. See the second example.
- Priority inversion: even though task 2 is not requesting R, it blocks task 3!



Priority Inheritance

- Applied in many RTOS including RTX.
- Blocking time is now bounded by the maximum (sum) length of critical section of lower priority tasks.
- The idea is to elevate the priority of low priority task (if it blocks high priority task) to the highest priority of tasks blocked by it.
- And resume its original priority when it exits the critical section.

Task 1 inherits task 3's priority



Priority Ceiling

- 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’.

- PCP always raises the priority, whether a higher-priority thread is blocked or not.

RTX Scheduling Options

- Pre-emptive scheduling
  - Each task has a **different priority** and will run until it is pre-empted or has reached a blocking OS call.
- Round-Robin scheduling
  - Each task has the **same priority** and will run for a fixed period, or time slice, or until it has reached a blocking OS call.
  - If quantum expired, state will be changed to READY.
- Co-operative multi-tasking
  - Each task has the **same priority** and the Round-Robin is disabled. Each task will run until it reached a blocking OS call or voluntarily yields the CPU.

- The default scheduling option for RTX is **Round-Robin Pre-emptive**. Prioritized RR.

The End!

- Next, we will look at Synchronization…