



REAL-TIME OPERATING SYSTEMS

MAESTRÍA EN SISTEMAS INTELIGENTES MULTIMEDIA

INSTRUCTOR: CARLOS ENRIQUE DIAZ GUERRERO



AUTHOR: CARLOS ENRIQUE DIAZ GUERRERO

7/10/2018

1

OVERVIEW

- Real-Time Systems Concepts
- Real-Time Operating Systems Concepts
- Kernel Structure and Task Management
- Time Management
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- References

OVERVIEW

- **Real-Time Systems Concepts**
- Real-Time Operating Systems Concepts
- Kernel Structure and Task Management
- Time Management
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- References



REAL-TIME SYSTEMS CONCEPTS

Real-Time System Definition



REAL-TIME SYSTEMS CONCEPTS

- Real-Time System. The correctness of the system depends on:
 - The logical **result** of the computation **and**
 - The **time** at which the results are produced

Deliver the right answer at the right **time**!

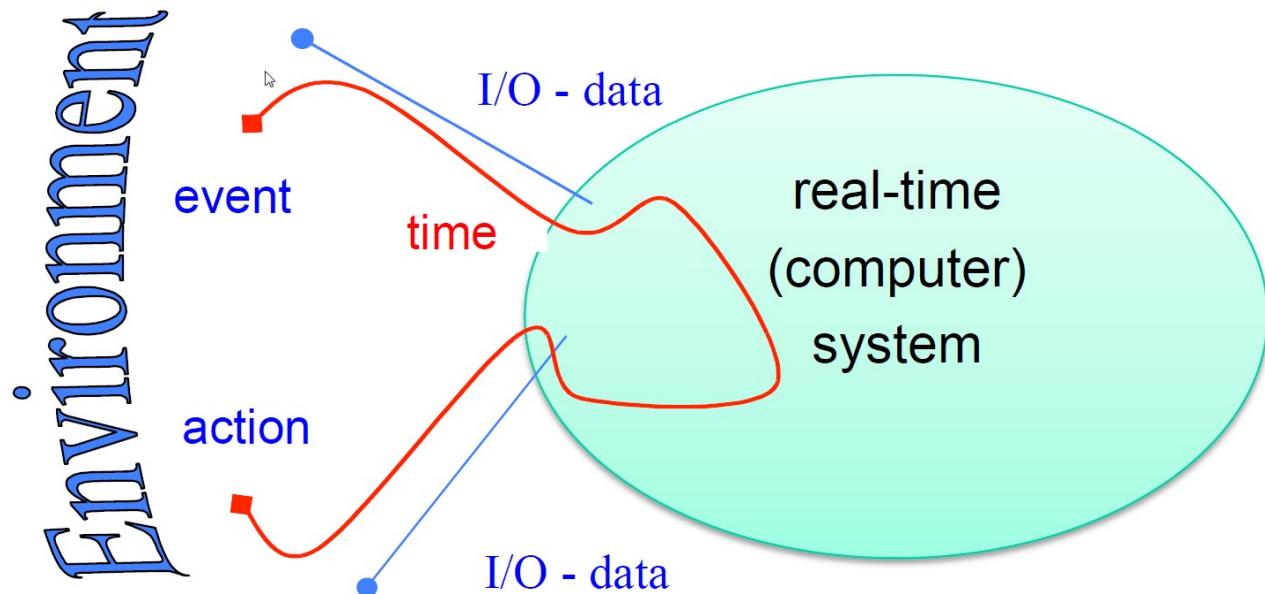
- “Fast” doesn’t necessarily mean real-time
- Too early or too late is not good

REAL-TIME SYSTEMS CONCEPTS

- Computer-world, e.g. PC
 - Reduced interaction with real-world
→ Main interaction is only with the user
 - Computer controls the speed of the user
 - User can be halted, annoyed
 - **“Computer-Time”**
- Real-world
 - Most interactions come from the environment
 - Environment controls the speed of the system
 - Environment can NOT be halted
 - Environment timings can NOT be modified
 - **“Real-Time”**

REAL-TIME SYSTEMS CONCEPTS

- Real-Time Systems shall react to the environment events as they occur, at their speed.



Real-Time Systems Types

- Timing constraints of the system (**deadlines**) not met:
→ System Failure
- Missing one or more **deadlines** is acceptable??
 - **Hard** Real-Time: Not acceptable. May lead to a catastrophe of the system.
E.g. Pacemaker device: Time failure may cause the death of a person.
 - **Soft** Real-Time: Acceptable (Up to some degree).
E.g. Video decoder: Time failure leads to user annoyance

REAL-TIME SYSTEMS CONCEPTS

- A Real-Time System may have components of mixed categories.
 - E.g., Some sub-functions classified as hard real-time while others classified as soft real-time.

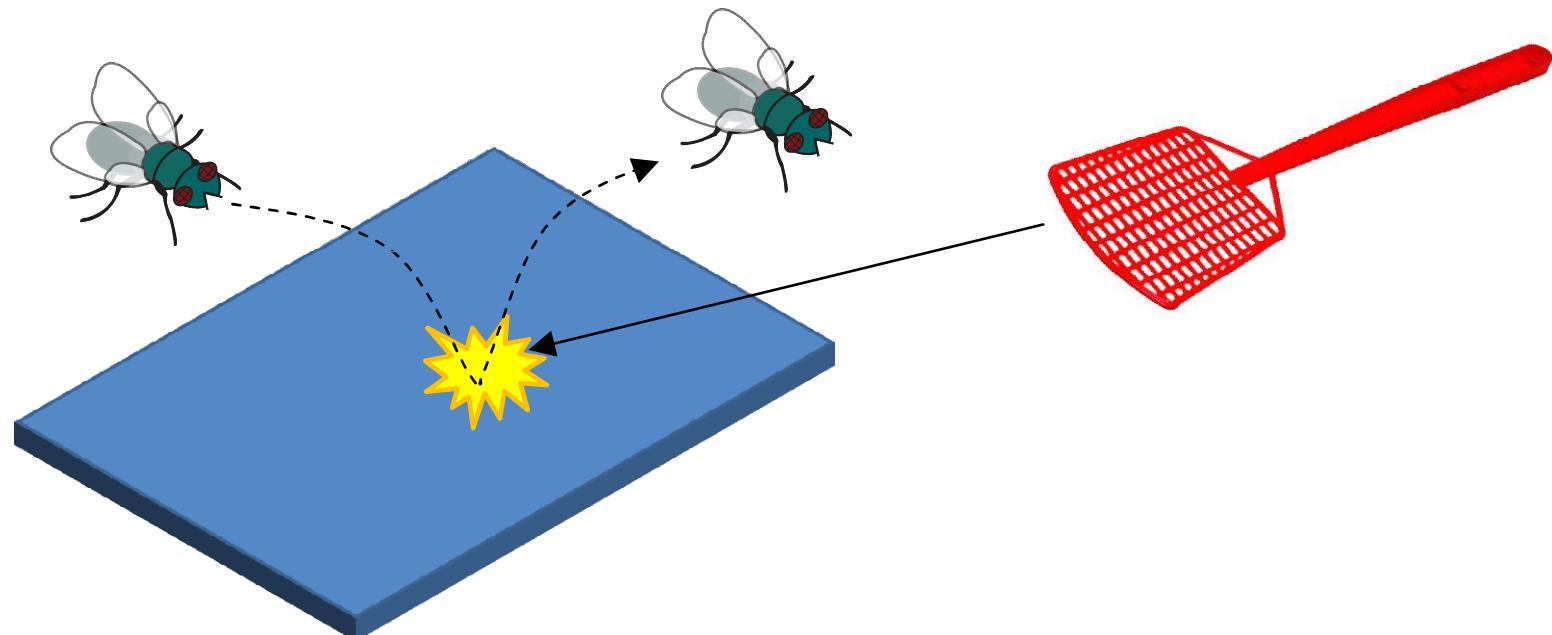


REAL-TIME SYSTEMS CONCEPTS

Real-Time Systems Examples...

REAL-TIME SYSTEMS CONCEPTS

- Fly-Swatter: Aim is to splash the fly...



- To Fast: Fly notice the swatter and flies-away
- To Slow: Fly escapes

REAL-TIME SYSTEMS CONCEPTS

■ Airbag System



- Activates 100 ms after crash
- Inflates in 20 ms
- Deflates immediately

- Too early: Deflated before passenger protection
- Too late: No passenger protection or collides passenger during inflation

REAL-TIME SYSTEMS CONCEPTS

■ Video player



- Decoding not in time:
 - Partial decoding
 - Lost of “macroblocks”
 - Chopping motion

REAL-TIME SYSTEMS CONCEPTS



How to... ?:

- Manage time properly
 - Guarantee deadlines
- Handle system complexity:
 - Multiple functions in a single device
 - E.g. Mobile phones
 - Functions with different time requirements
- Manage HW resources
- React in a timely fashion to errors
-

- The answer:

Real-Time Operating System

Perhaps the most important component in a real-time system.

OVERVIEW

- Real-Time Systems Concepts
- **Real-Time Operating Systems Concepts**
- Kernel Structure and Task Management
- Time Management
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- References

Endless-Loop programs

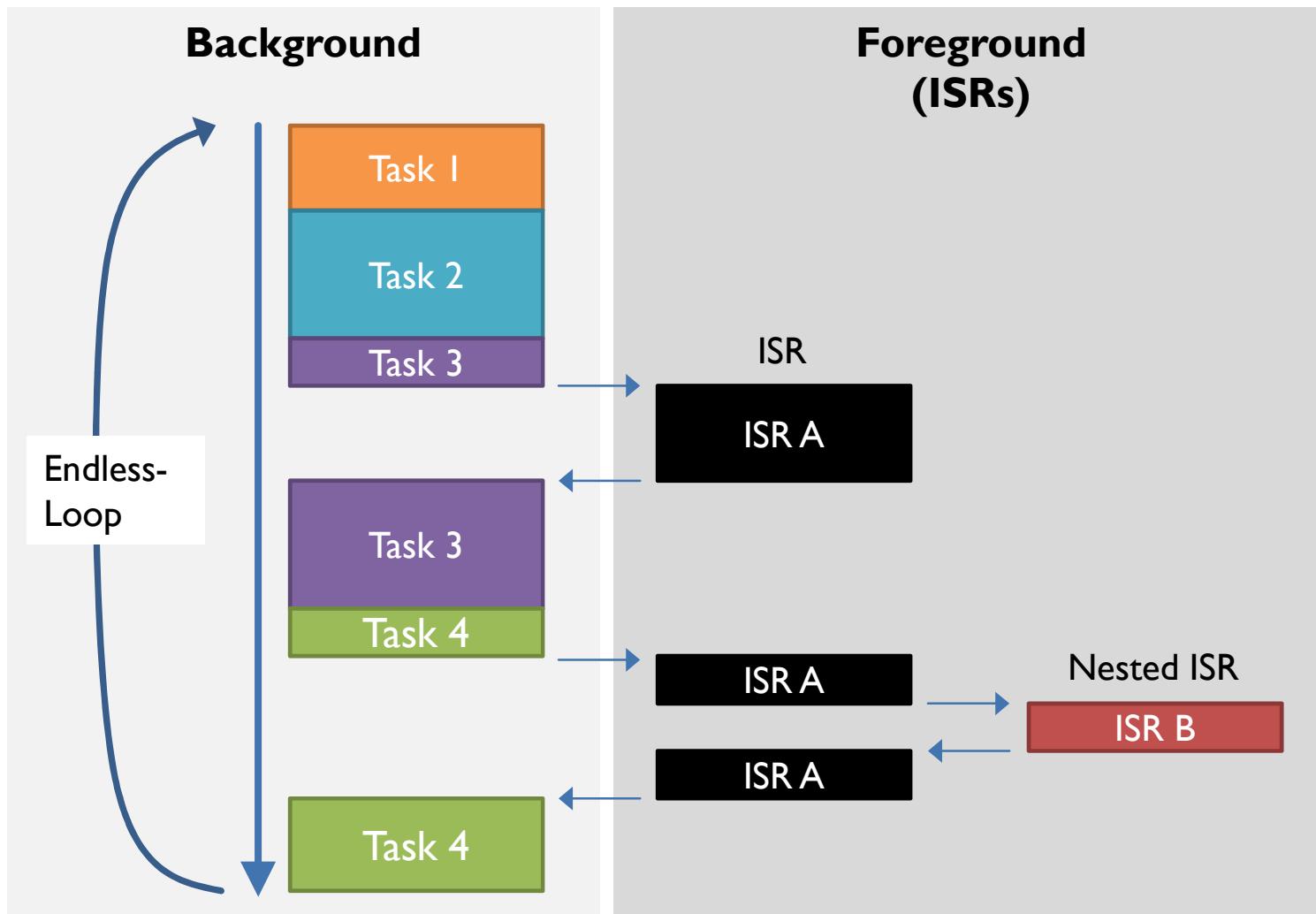
vs.

RTOS-based programs

Endless-Loop programs

- A.k.a. Background/Foreground programs
- Application is implemented at two levels of hierarchy:
 - Background (Task-Level)
 - Foreground (Interrupt-Level)
- Before OS this was the common way to implement “real-time” programs.

REAL-TIME OPERATING SYSTEMS CONCEPTS



REAL-TIME OPERATING SYSTEMS CONCEPTS

```
void main(){
    /* initialization code */

    for(;;){
        /* Endless-Loop */
        app_task_1();
        app_task_2();
        app_task_3();
        app_task_4();
    }
}
```

```
void app_task_1(){
    /* code for task 1... */
}

void app_task_2(){
    /* code for task 2... */
}

void app_task_3(){
    /* code for task 3... */
}

void app_task_4(){
    /* code for task 4... */
}

void ISR_A(){
    /* Handler for interrupt A */
}

void ISR_B(){
    /* Handler for interrupt B */
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

■ **Background Level (Task-Level)**

- Functions (tasks) called inside an infinite-loop
- Functions get called sequentially
- Time on which a given function will be called again is rather non-deterministic
 - Depends on the execution time of the other functions.
 - This time varies depending on the executed path (E.g. Which if/else branch has been taken)
 - Depends on occurrence and execution time of foreground functions (interrupts)
- Hard to have time control over the desired functions

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Foreground Level (Interrupt-Level)**
 - Functions called within an ISR context.
 - ISRs always interrupt the background functions so this level has higher priority.
 - Events are handled by interrupts
 - Time-Critical functions need to be handled by ISRs.

REAL-TIME OPERATING SYSTEMS CONCEPTS

■ **Disadvantages:**

- Very hard to have control of time
- System will most likely have long ISRs
 - Care needed for nesting interrupts
 - May cause long interrupt-suppression times → Missing events.
- Difficult to handle a “complex” system (many functionalities in the same system)

■ **Advantages:**

- Very low resource usage (RAM/ROM)
 - Low Cost
- No additional overhead compared to a system with an RTOS.

■ **Common usage:**

- Low-end, low-cost products like microwaves, toys, etc.



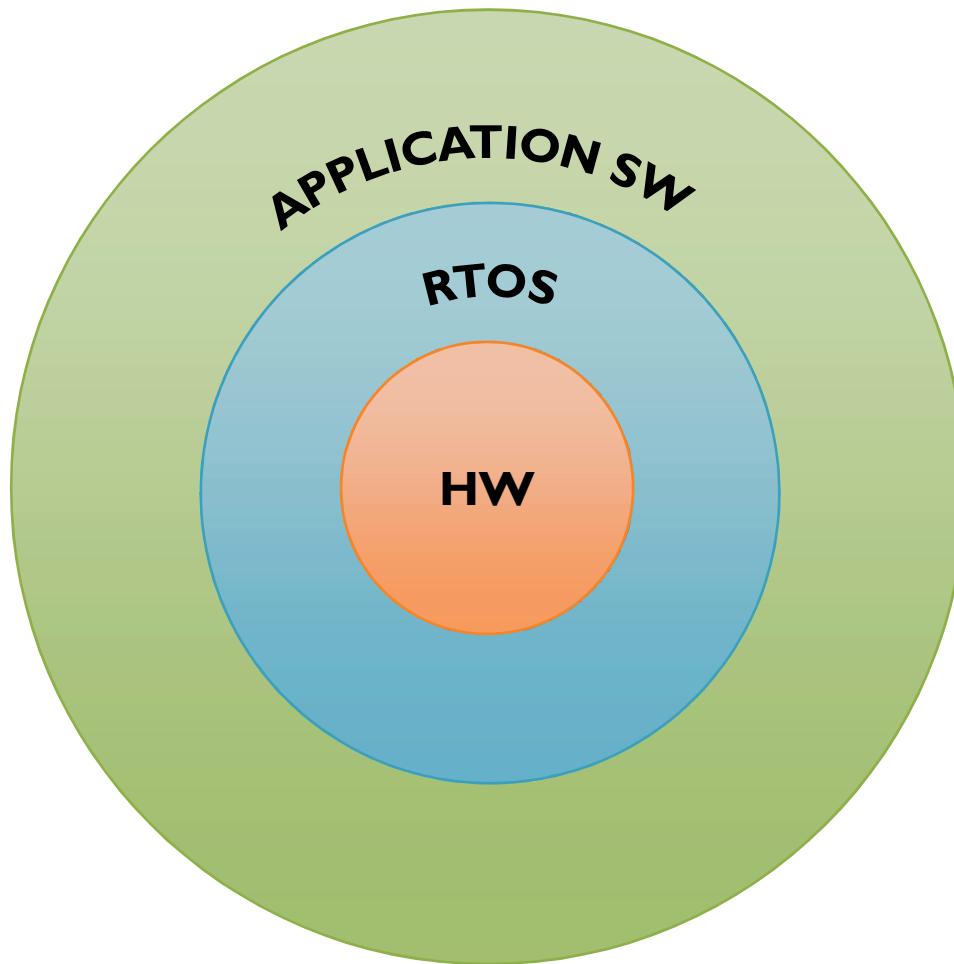
REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Endless-Loop** systems are not enough for most of the current real-time systems.
- Need something else...

Operating System based programs

- An Operating System (**OS**) is a program that:
 1. Controls the execution of **application** programs
 2. Acts as an **interface** between the hardware and the software application layers
- A Real-Time Operating System (**RTOS**) is a type of OS that has been designed to guarantee the timing requirements of a real-time system.

REAL-TIME OPERATING SYSTEMS CONCEPTS



This is an ideal layering...

In reality RTOS normally
do not cover 100% of the
Hardware Abstraction
Layer

REAL-TIME OPERATING SYSTEMS CONCEPTS

I. RTOS as an application programs manager:

- Determines when and which application programs to **execute**
- Provides **timing** management
 - E.g., Delays
- Provides **events** management
- Provides **communication** and **synchronization** artifacts
- Provides logging mechanisms
 - E.g., Stack usage, CPU usage
- Provides mechanisms for errors handling
 - E.g. Stack over/under flow

REAL-TIME OPERATING SYSTEMS CONCEPTS

2. RTOS as HW-SW interface:

- HW Resources management
 - Mutexes, etc.
- First level of hardware abstraction
 - At least some basic HW abstraction is normally provided: ISRs, timers, atomic operations artifacts

REAL-TIME OPERATING SYSTEMS CONCEPTS

- RTOS does all that by means of:
 - RTOS **Services**
 - RTOS **APIs**

REAL-TIME OPERATING SYSTEMS CONCEPTS

- RTOS advantages (compared to endless-loop systems)
 - Significantly eases time management of different application functions
 - Better handling of complexity
 - Easier to guarantee deadlines (determinism)
 - Multitasking (described later...)
 - Improves re-usability (faster time-to-market)
 - ...
- RTOS drawbacks (compared to endless-loop systems):
 - Need more HW resources
 - RAM, ROM is required for the RTOS itself
 - May not fit in a very small microcontrollers
 - Add additional overhead to the system
 - E.g., Context switching takes time

- Simple example: Button **Debouncing**
 - How to implement this in an endless-loop system?
 - What about an RTOS-bases system?
 - Which one is easier?

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Some popular RTOS:
 - OSEK OS
 - μ C/OS
 - FreeRTOS
 - VxWorks



REAL-TIME OPERATING SYSTEMS CONCEPTS

RTOS Common Concepts





REAL-TIME OPERATING SYSTEMS CONCEPTS

Task definition

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Real-time application programs are generally implemented by splitting the **job** to be done into several **tasks**

REAL-TIME OPERATING SYSTEMS CONCEPTS

■ Task:

- A task is a **container** for the application code
- Basic building block of software written under RTOS
- In most RTOS a task is just a sub-routine (e.g., a function)
- Is a simple program that thinks it has the CPU all for itself
- Each task execution can be “**independent**” from each other.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- FreeRTOS Task example:

```
void TaskA( void *pvParameters )
{
    /* Some run once-only code here... */
    for( ;; )
    {
        /* Task application code here. */
        appFunction1();
        appFunction2();
        /* ... */
    }

    /* Tasks must not attempt to return from their implementing
    function or otherwise exit. If it is necessary for a task to
    exit then have the task call vTaskDelete( NULL ) to ensure
    its exit is clean. */
    vTaskDelete( NULL );
}
```

REAL-TIME OPERATING SYSTEMS CONCEPTS

- μC/OS Task example:

```
static void TaskA( void *p_arg )
{
    OS_ERR err;
    CPU_TS ts;

    /* Some run once-only code here... */

    while(1)
    {
        /* Task application code here. */
        appFunction1();
        appFunction2();

        /* OS service call to suspend the task
         * for 100 ms (this will happen periodically) */
        OSTimeDlyHMSM( (CPU_INT16U) 0,
                        (CPU_INT16U) 0,
                        (CPU_INT16U) 0,
                        (CPU_INT32U) 100,
                        (OS_OPT) OS_OPT_TIME_HMSM_STRICT,
                        (OS_ERR *)&err );
    }
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- OSEK OS Task example:

```
TASK(TaskA)
{
    ****start user code*****
    appFunction1();
    appFunction2();
    ****end user code****

    TerminateTask();
}
```

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Types of Tasks:**

- Periodic
- Aperiodic
- Sporadic
- Background



REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Periodic Task:**

- Need to be called periodically (have a periodic **arrival time**).
- Each one can be seen as an endless-loop.
- Most of the RT application code is contained in this kind of task.
- Deterministic time.
- Timings can be easily guaranteed



REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Aperiodic Task:**

- No arrival time known.
 - Soft: No execution deadline.
 - Firm: With deadline
- High time-critical functions are implemented in this kind of tasks.
 - Where “latency” of *reaction* is important
- Special care to guarantee timings.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Sporadic Task:**

- Aperiodic task but with minimum arrival time known.

- **Background Task:**

- Runs when no periodic, aperiodic or sporadic task is executed (use CPU's left overs).
 - No time guarantees at all
 - Good for long, non-critical calculations, e.g., RAM/ROM check.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Although there are many task types...
- **Periodic** Tasks are commonly the base for real-time scheduling algorithms.

REAL-TIME OPERATING SYSTEMS CONCEPTS

■ Tasks i timing **parameters**:

- Arrival time (a_i): Time on which a given task is expected to start executing (task gets ready to run).
A.k.a., release time (r_i), request time.
- Task Period (T_i)
- Computation Time (C_i)
- Worst-Case Execution Time (WCET)
- Absolute Deadline (d_i)
- Relative Deadline (D_i)
- Slack Time (S_i). A.k.a., laxity
- Response time (R_i)

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Tasks, Jobs, Processes, Threads???
 - Terminology used diversely across the literature
- Terms Processes / threads: more commonly used in non real-time OS
 - E.g., Linux, Windows, Mac Os
- Terms Tasks / Jobs: more commonly used for RTOS
 - E.g., OSEK, FreeRTOS, µC/OS, etc.

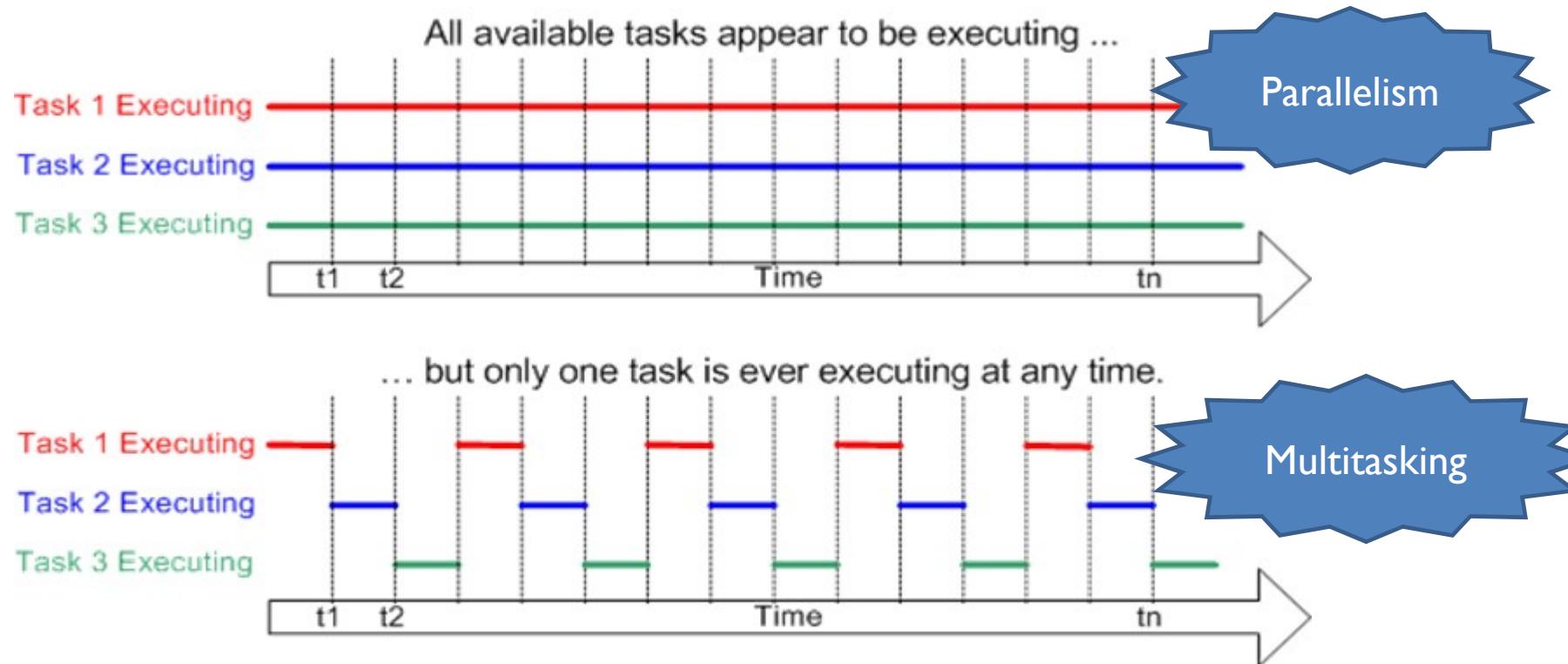


REAL-TIME OPERATING SYSTEMS CONCEPTS

Multitasking

REAL-TIME OPERATING SYSTEMS CONCEPTS

■ Multitasking



REAL-TIME OPERATING SYSTEMS CONCEPTS

■ Multitasking

- Multiple tasks are performed during the same period of time
- Multitasking does not necessarily mean that they are executing at exactly the same instant (**parallelism**)
 - Multitasking doesn't necessarily implies parallelism.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Mutitasking in a single-CPU system
 - Only one task running at any point in time.
 - Need to **schedule** which task shall be running and which ones shall be waiting for the CPU.
 - Tasks executing in turns fast enough:
→ **Illusion** of parallelism.
- Mutitasking in a multi-CPU system
 - RTOS design complexity really challenging but..
 - Out of scope for this course..



REAL-TIME OPERATING SYSTEMS CONCEPTS

Task Scheduling



REAL-TIME OPERATING SYSTEMS CONCEPTS

■ Tasks **Scheduling**

- Algorithm to decide **which task** should be executing at any particular time
- Performed by the RTOS kernel **scheduler** (detailed later...)



REAL-TIME OPERATING SYSTEMS CONCEPTS

- Desired characteristics of an RTOS **scheduler**:
 - Determinism
 - Responsiveness
 - User Control
 - Reliability
 - Fail-soft operation

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Types of schedulers*:**
 - Non Preemptive (cooperative)
 - Preemptive
- *This classification is based on the scheduler's **decision mode**: When is the scheduling algorithm triggered / allowed?

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Scheduler algorithms can also be characterized by:
 - Their **selection function**: Criteria to select the next task for execution?
 - Their tasks **priority** assignation:
 - **Fixed** priority: Tasks priorities are assigned “offline”
 - **Dynamic** priority: Task priorities assigned in run-time or “online”

REAL-TIME OPERATING SYSTEMS CONCEPTS

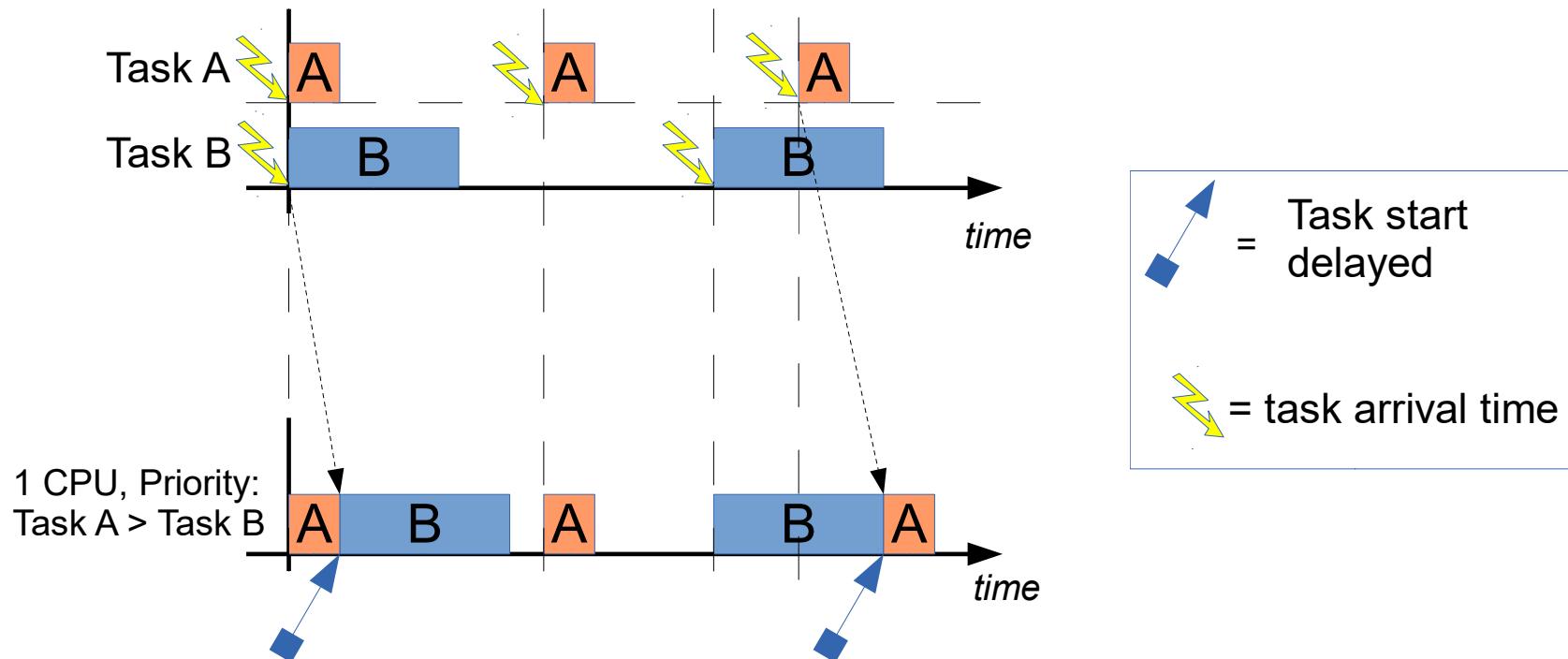
- **Task priority:** The order of importance of a task regarding its execution within an RTOS scheduler.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Non preemptive** scheduling:
 - Decision mode: scheduling is NOT allowed **during** the execution of a task.
 - Running Task “dictates” when to trigger the scheduling.
 - Tasks voluntarily release the CPU.
 - Every task is executed from **beginning** to **end** without interruption of any other task or until they **suspend** themselves.
 - A.k.a. cooperative scheduling

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Non preemptive** scheduling



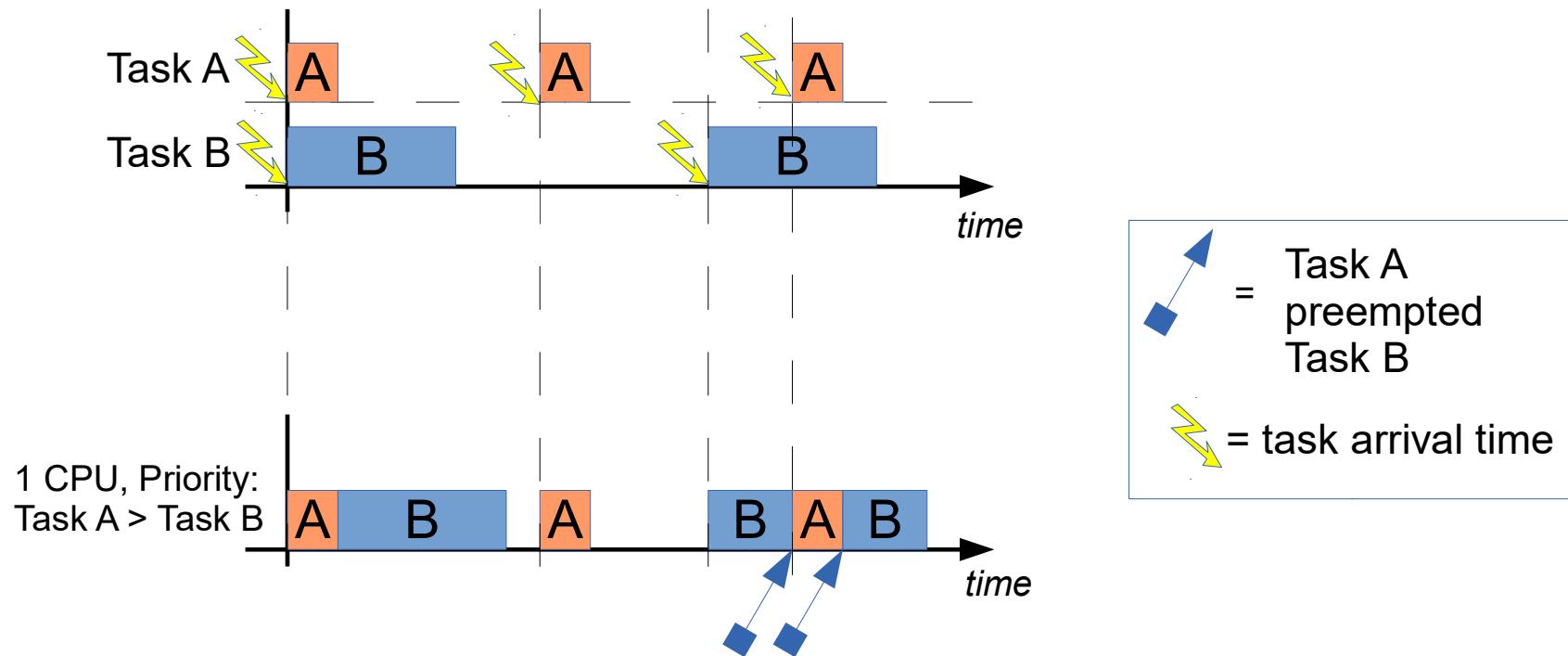
REAL-TIME OPERATING SYSTEMS CONCEPTS

■ **Preemptive** scheduling:

- Decision mode: scheduling is allowed at **any** time during the execution of any task.
- The scheduler “dictates” when a given task is to be executed.
- A task being executed can be interrupted by the scheduler and this last may decide to start executing another task or to continue the execution of the interrupted one.
- Tasks interleaving.

REAL-TIME OPERATING SYSTEMS CONCEPTS

■ **Preemptive** scheduling



REAL-TIME OPERATING SYSTEMS CONCEPTS

■ **Non preemptive** scheduling:

- Simpler
- Less overhead
- No data-sharing problems



- A task may monopolize the CPU
- Failure in single task → all system failure
- Harder to control timings → **soft** real-time systems.



REAL-TIME OPERATING SYSTEMS CONCEPTS

■ Preemptive scheduling:

- More fair utilization of the CPU
- Better Reaction Time
- Failure in single task → Other tasks can continue
- Easier to control timings → **hard** real-time systems



- More complex
- Higher overhead
- Data-sharing problems



REAL-TIME OPERATING SYSTEMS CONCEPTS

- RTOS may combine both non-preemptive and preemptive scheduling
 - Tasks are assigned to either a non-preemptive or a preemptive group.
 - RTOS scheduler manages each group “independently”
 - RTOS needs to handle the coexistence of both strategies

REAL-TIME OPERATING SYSTEMS CONCEPTS

Common scheduling algorithms

- Non Preemptive:
 - **FCFS**: First Come First Served
 - **SPN**: Shortest Process Next
 - **HRRN**: Highest Response Ratio Next
- Preemptive:
 - **RMS**: Rate Monotonic Scheduling (offline/fixed priorities)
 - **EDF**: Earliest Deadline First (online/dynamic priorities)
 - **RR**: Round-Robin (all task have the same priority)
 - **SRT**: Shortest Remaining Time First (preemptive version of SPN)



REAL-TIME OPERATING SYSTEMS CONCEPTS

Task State Models

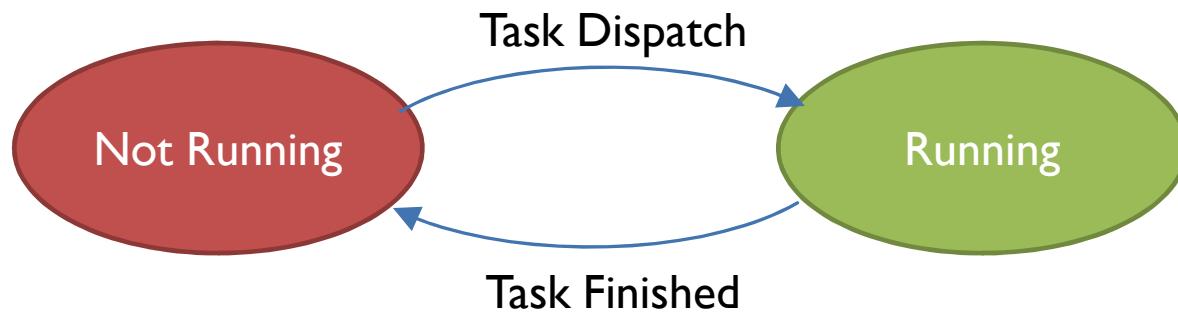


REAL-TIME OPERATING SYSTEMS CONCEPTS

- Based on the scheduling algorithms, tasks can be in different **states**:
 - Running, ready to run, blocked, ...

REAL-TIME OPERATING SYSTEMS CONCEPTS

- The simplest model of a task considers only **two states**
 - Running
 - Not running

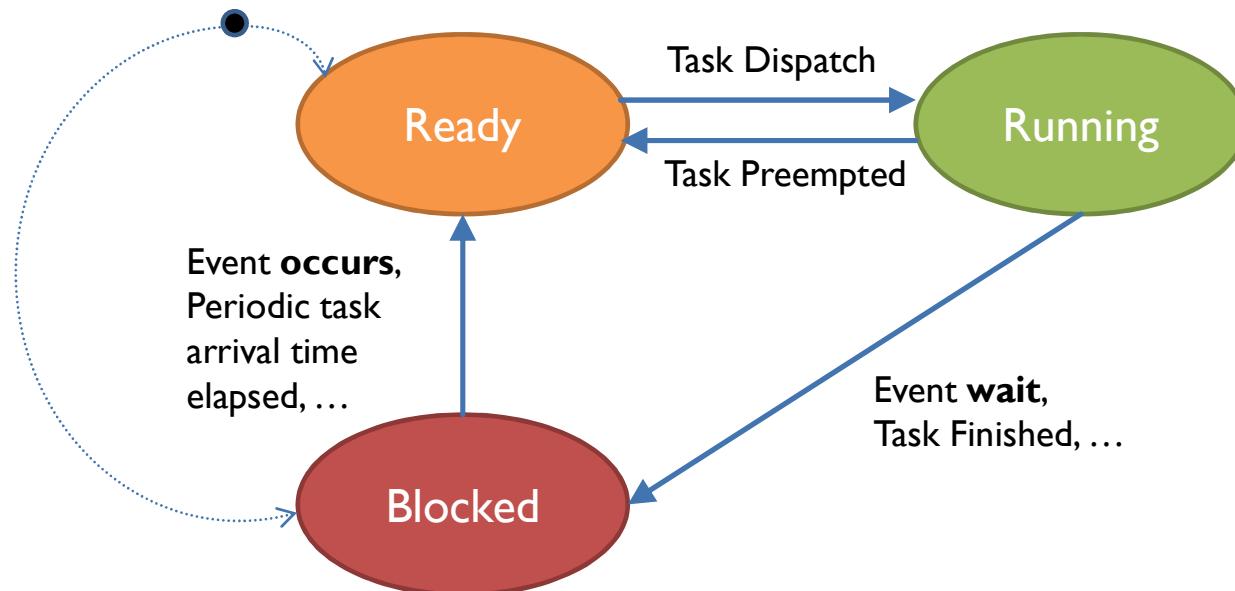


REAL-TIME OPERATING SYSTEMS CONCEPTS

- Model too simplistic for an RTOS
 - Cannot handle the situation of one or more tasks being “ready for execution” but not being executed.
 - Cannot model “preemption”
 - May be sufficient for the simplest FCFS scheduling..

REAL-TIME OPERATING SYSTEMS CONCEPTS

- The **three states** task model
 - **Running** (being executed)
 - **Ready** to run (but not yet being executed)
 - **Blocked** / Waiting (waiting for an event to occur in order to become *ready*)





REAL-TIME OPERATING SYSTEMS CONCEPTS

- This is the most common task model for an RTOS
- Those three states are **conceptual** states.
- Actual RTOS implementations normally define more states.



REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Running state:**
 - Task in this state is the one being executed by the CPU
- Only **one** task can be in a **running** state at a time
 - Remember that in this course we only consider single CPU systems

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Ready state:**
 - The task is ready to be executed but is waiting for the scheduling algorithm to *choose* it for execution.
- When a **running** task is **preempted**, it transitions to the **ready** state. The task by itself “didn’t want” to be stopped and hence it is still ready to run.
- More than one task may be in the ready state at a given time.
- The RTOS keeps track of which tasks are in **ready** state by the so-called **ready list**.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Blocked/Waiting state:**

- The task is blocked (not running) and not ready to run, it is indeed waiting for something to occur in order to resume execution.

- Tasks normally wait for:

- An event for which it has itself been blocked for, e.g., waiting for an I/O resource, waiting for a semaphore to be unlocked, etc.
 - Some RTOS implement task periodicity by calling an RTOS “delay” function with the task period time as the delay parameter (see the μ C/OS task code example). The time expiration is the “event” the task will be waiting for.

REAL-TIME OPERATING SYSTEMS CONCEPTS

... Blocked/Waiting state cnt'd...

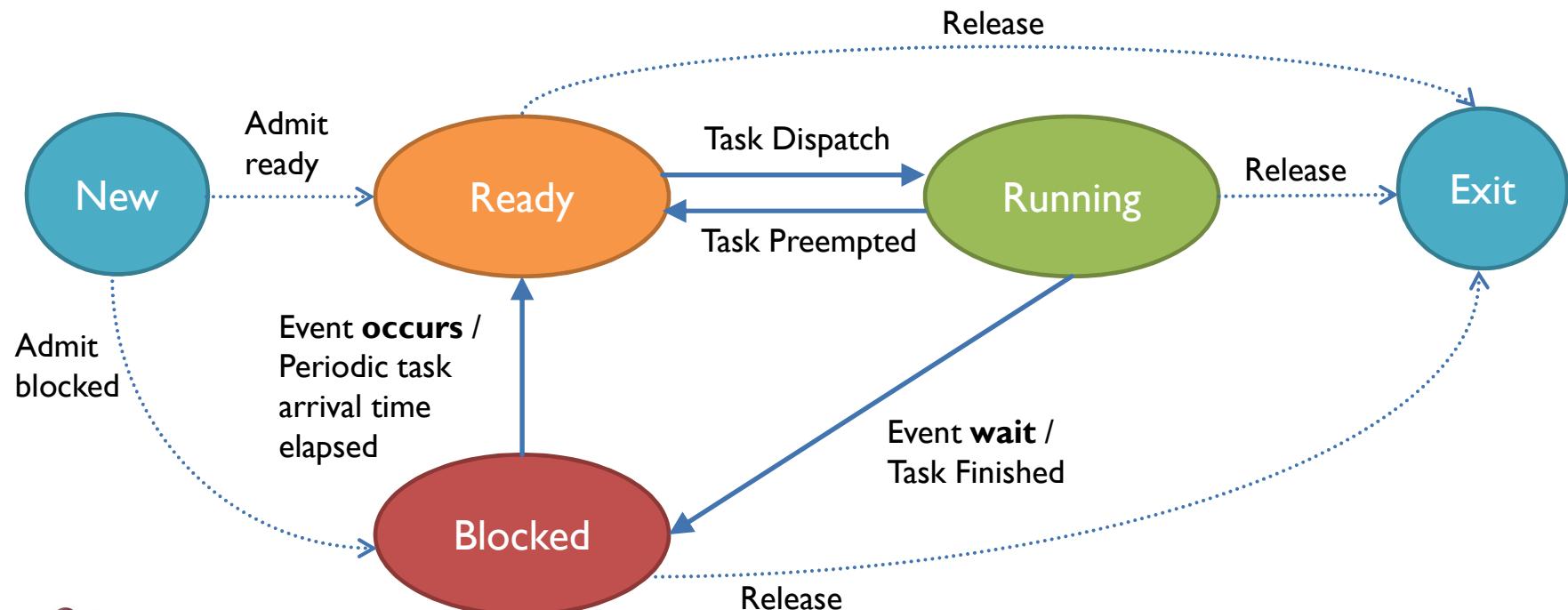
- Once the event occurs, the task transitions to the **ready** state.
- They do not transition directly to the **running** state because is the scheduler's responsibility to do so.
- The terms **blocked** and **waiting** in this context are indistinctly used.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- The **initial** state of a Task under this model can be either “ready” or “Blocked/Waiting”.
 - If task is initially set to “Blocked/Waiting”, then the *waiting* event shall be defined as well at init phase.

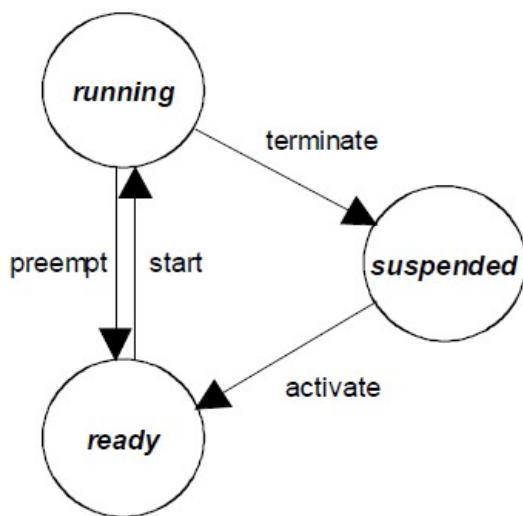
REAL-TIME OPERATING SYSTEMS CONCEPTS

- Some RTOS implementations add two additional states:
 - **New**: The task object has been created (allocated in memory, etc.) but it has not yet been admitted to the ready list. E.g., because it has not been initialized, etc.
 - **Exit**: The task object is no longer considered by the scheduling algorithm. E.g., because it is in a deadlock and the RTOS decides to ignore it, etc.

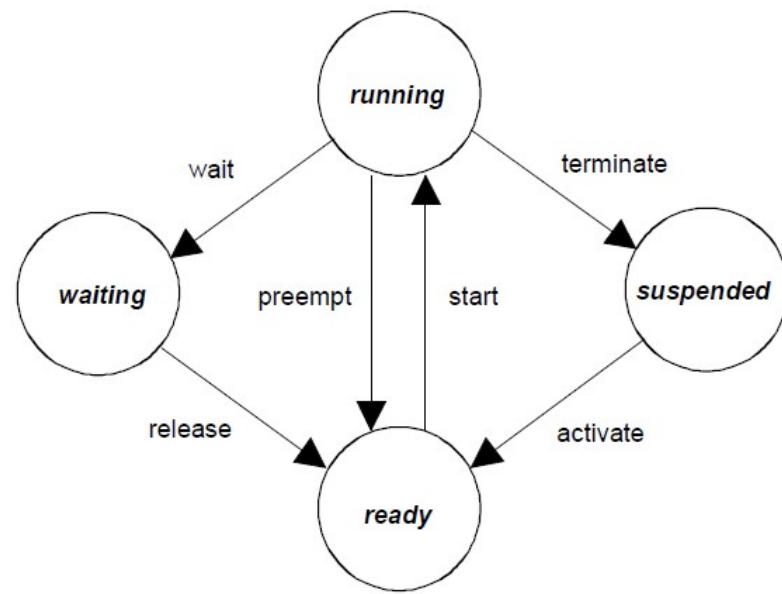


REAL-TIME OPERATING SYSTEMS CONCEPTS

- Example: OSEK OS Task models



Normal Task



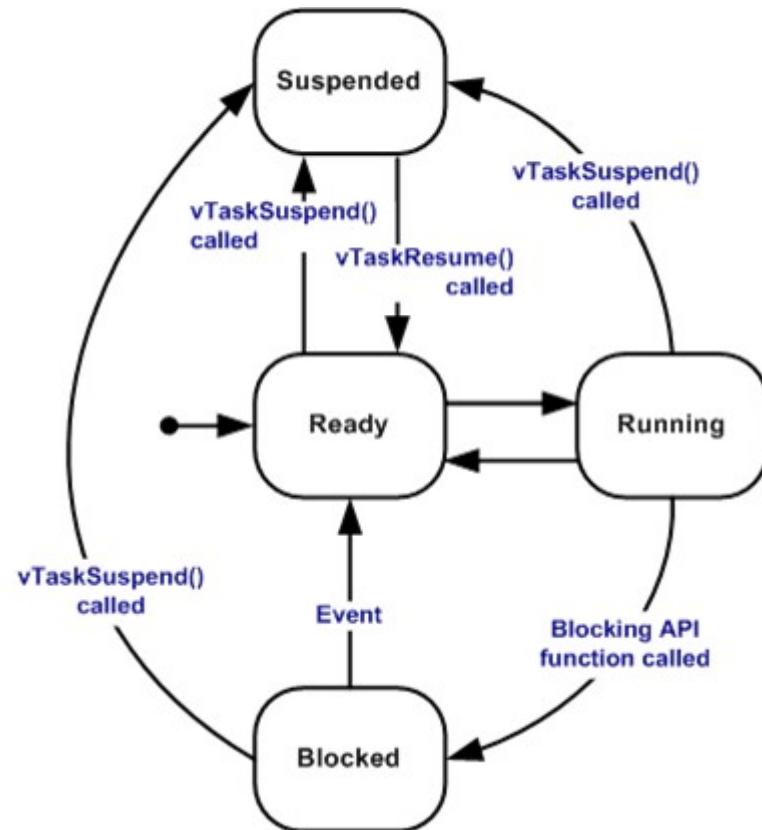
Extended Task

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... OSEK OS Task models
- OSEK OS splits the “blocking” state in two states: **suspended** and **waiting**.
- OSEK OS distinguishes when the task is blocked due its termination (suspended state) or because it has put itself to wait for something before termination (waiting state).
- If a task is able to “suspend itself” it is called an **extended task** otherwise it is called a *normal* task.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Example: FreeRTOS Task model

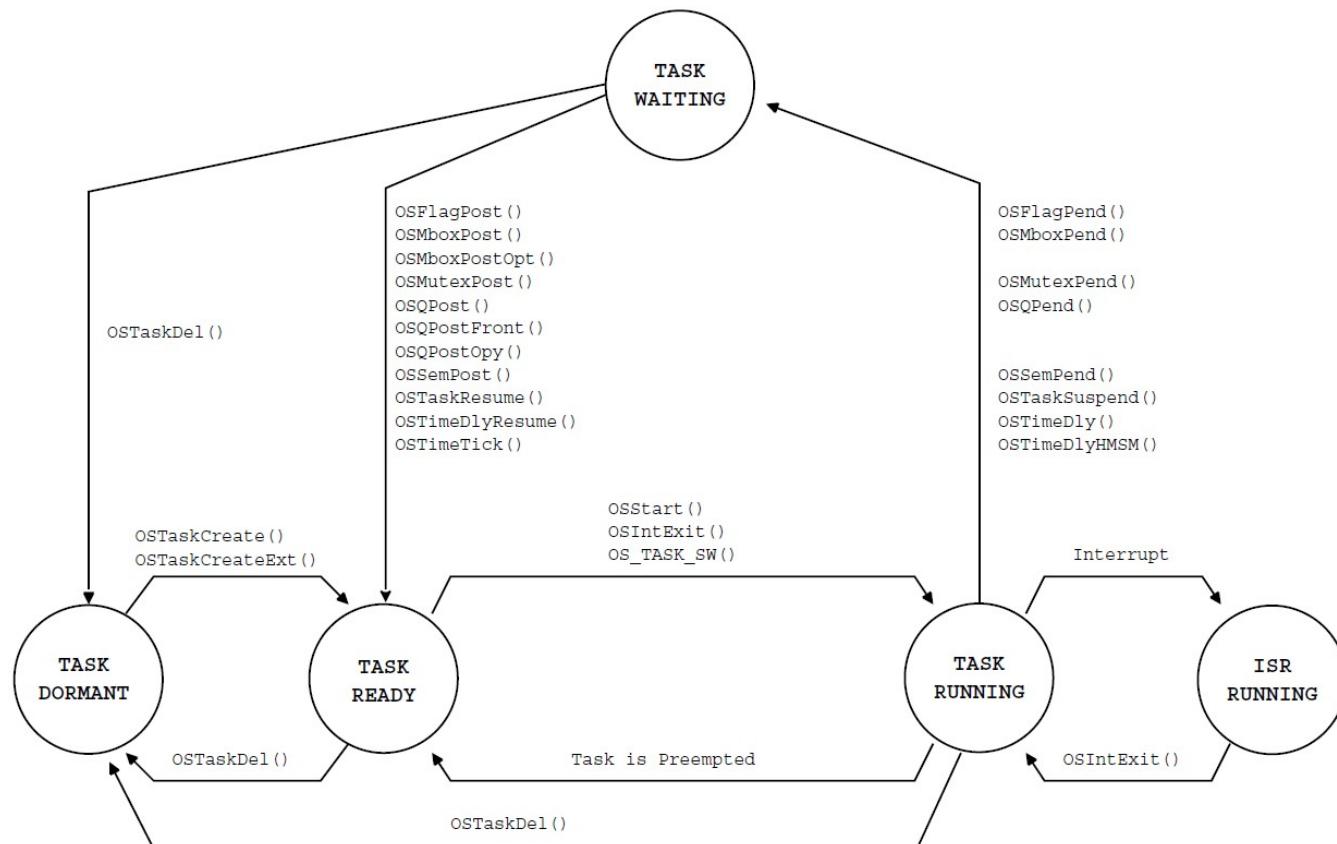


REAL-TIME OPERATING SYSTEMS CONCEPTS

- ...FreeRTOS Task model...
 - Similarly than OSEK OS, FreeRTOS distinguishes when the task has been blocked due to its termination or because it has suspended itself.
 - The “Blocked” state of FreeRTOS is equivalent to the “waiting” state of OSEK OS.

REAL-TIME OPERATING SYSTEMS CONCEPTS

■ Example: µC/OS II Task model





REAL-TIME OPERATING SYSTEMS CONCEPTS

Tasks Schedulability





REAL-TIME OPERATING SYSTEMS CONCEPTS

- Periodic tasks are the basis for real-time scheduling
- Common periodic task parameters:
 - T_i Period of task i
 - a_i Arrival time of task i . Time at which task i may start to execute (it gets ready to run).
 - D_i Relative Deadline of task i . Relative from start of period. Time at which the task i must be completed.
 - C_i Computation time. Time required to completely execute task i as if it was never interrupted.

REAL-TIME OPERATING SYSTEMS CONCEPTS

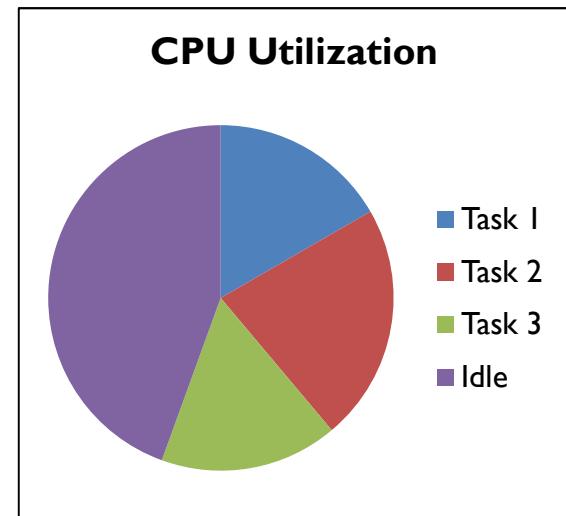
CPU Utilization

- Portion of CPU used for execution of a task set.
- Utilization of a single task i :

$$U_i = \frac{C_i}{T_i}$$

- Utilization of task set with n tasks:

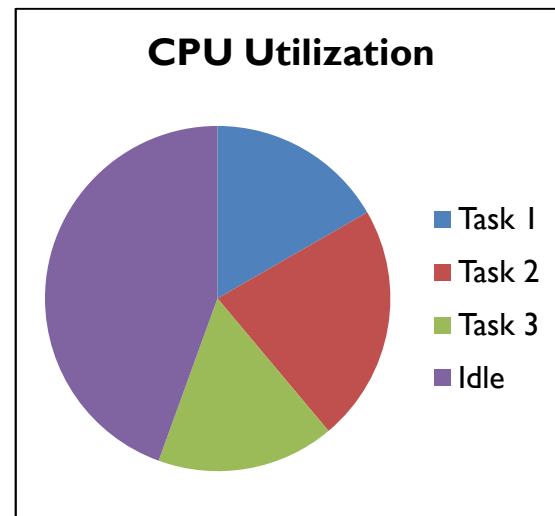
$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} = \sum_{i=1}^n \frac{C_i}{T_i}$$



REAL-TIME OPERATING SYSTEMS CONCEPTS

Schedulability

- Definition: A task set is **schedulable** when all tasks meet their deadlines during the lifetime of the task set (system).



REAL-TIME OPERATING SYSTEMS CONCEPTS

Schedulability conditions based on utilization

- **Necessary condition**

- If it doesn't hold the taskset is for sure not schedulable
- If it holds we can't yet tell whether it is schedulable or not

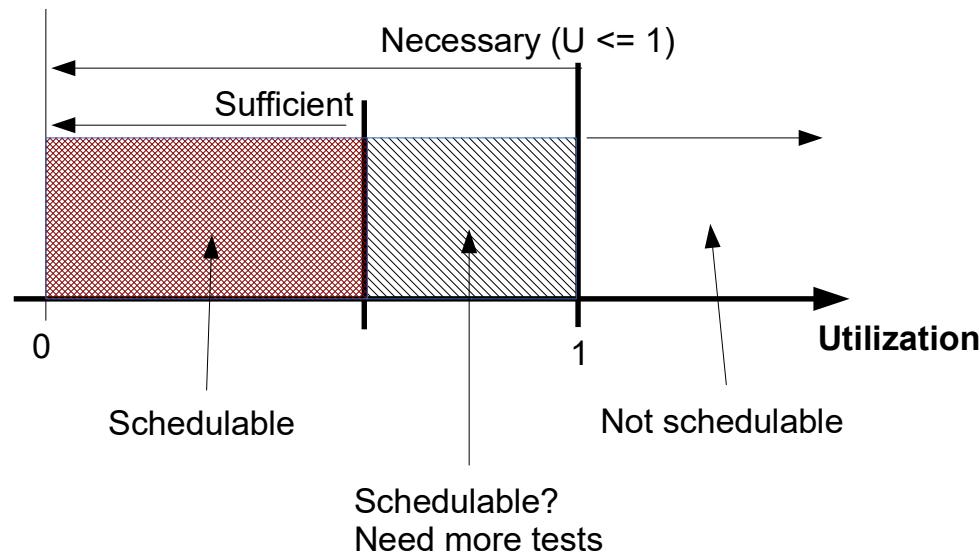
$$U \leq 1$$

- **Sufficient condition:**

- If it holds the task set is schedulable.
- If it doesn't hold then we don't know.
- Mathematical expression depends on the specific scheduling algorithm..

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Schedulability conditions ...
- **Necessary and sufficient condition**
 - If both condition hold then we are sure that the taskset is schedulable!
 - This is what we are looking for!.



REAL-TIME OPERATING SYSTEMS CONCEPTS

Critical Instance Schedulability Test

■ **Critical Instance**

- The worst case **response** time for all tasks is given when all tasks are **released** at the same time.
- Under these conditions, if tasks meet the first deadlines (the first periods), they will do so in the future!



REAL-TIME OPERATING SYSTEMS CONCEPTS

Task set **Hyper-period**

- **Hyper-period**

- The hyper-period is the **Least Common Multiple (LCM)** of the period of all tasks in the task set.
 - Time after which the pattern of tasks execution starts to repeat.
- If the system can be scheduled for one hyper-period, it can be scheduled for all, given no aperiodic or sporadic tasks, and no resource constraints
- Can be verified by exhaustive simulation.



REAL-TIME OPERATING SYSTEMS CONCEPTS

Clock Tick



Clock Tick

- Scheduling may be triggered by the **tasks** by means of an RTOS service call.
 - μC/OS: OSTimeDlyHMSM(...), ...
 - OSEK: TerminateTask(),
 - FreeRTOS: vTaskSuspend(...), ...
- Scheduling may be triggered by the RTOS **kernel** itself by means of a periodic interrupt
 - Clock Tick

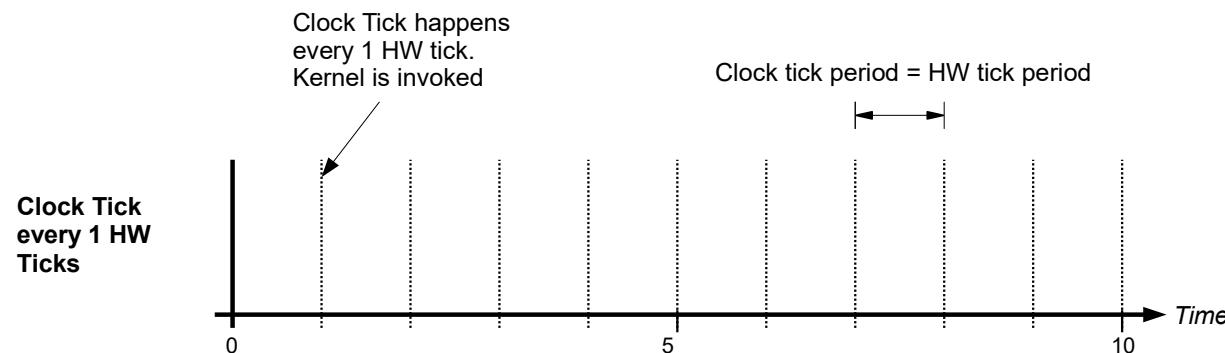
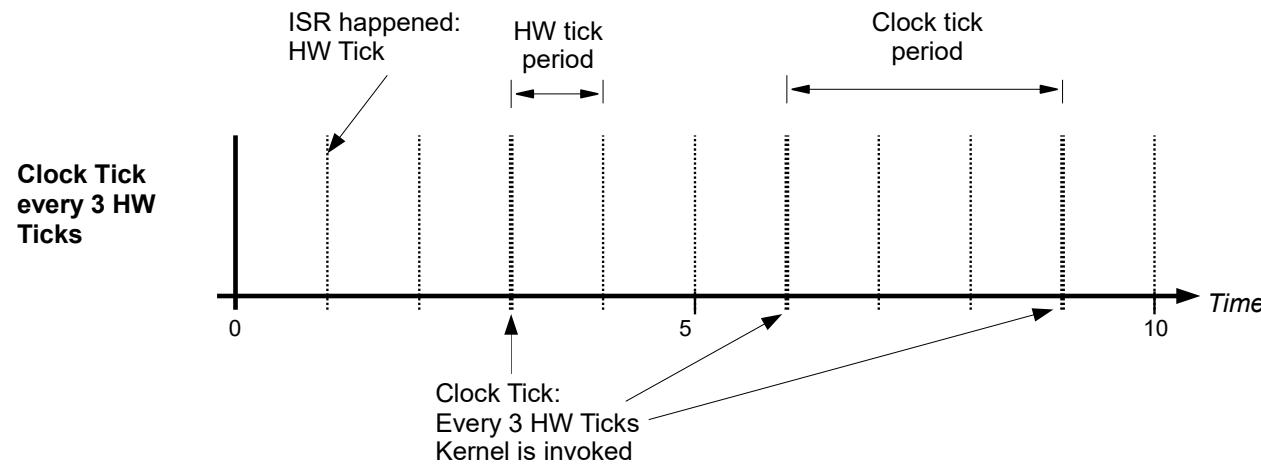
REAL-TIME OPERATING SYSTEMS CONCEPTS

...Clock Tick

- A periodic interrupt is configured in the target MCU.
- The clock tick can be triggered everytime the ISR occurs or every n times the ISR occurs depending on the design.
 - Each time the ISR occurs it is said that a HW tick occurred.
 - Every n HW ticks, a Clock Tick happens. The value of n can be 1 or more..

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Clock Tick



REAL-TIME OPERATING SYSTEMS CONCEPTS

...Clock Tick

- Everytime the **clock tick** occurs, the kernel is invoked to do, among others, the following actions:
 - Keep track of time: update a tick counter variable.
 - Run the scheduling algorithm: check the ready list and possibly select a task for execution, etc.
 - Keep track of events, etc..

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Periodic interrupt in target MCU
 - Preferably select a MCU timer peripheral which doesn't have an input capture/output compare functionality (do not want to waste those resources).
 - E.g., “SysTick” in ARM Cortex or similar when available
- Clock Tick:
 - Shall provide accurate time
 - Keeps the system “alive”
 - Selection of clock tick period is important
 - Too small → High overhead
 - Too big → Slow response time
 - Common clock tick periods: 2.5 ms, 5 ms, 10 ms

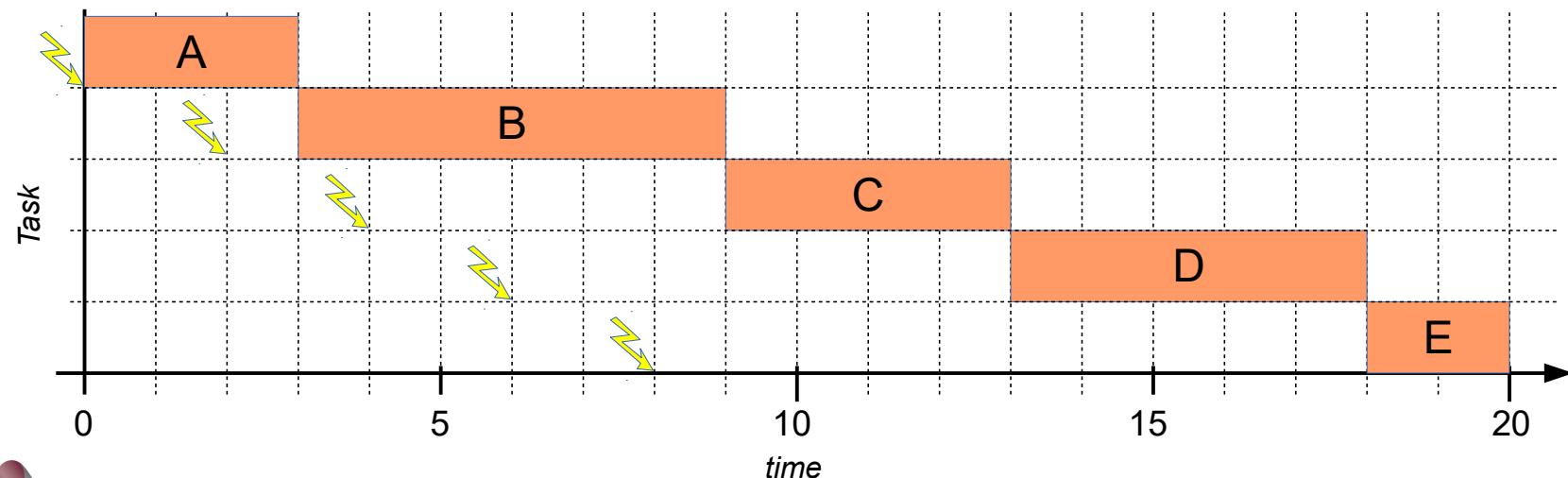
Scheduling Algorithms

REAL-TIME OPERATING SYSTEMS CONCEPTS

FCFS: First-Come First-Served

- Decision Mode: Non-Preemptive
- Selection Function: Process which has been longer in the **ready** queue
- The ready queue is indeed a FIFO
- Priority handing: Tasks priorities given indirectly by the ready queue FIFO.

Task	Arrival Time	WCET
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



REAL-TIME OPERATING SYSTEMS CONCEPTS

FCFS: First-Come First-Served

- Difficult to guarantee deadlines
- Time arrivals may not necessarily occur in the application priority order

REFERENCES

- William Stallings, “Operating Systems - Internals and Design Principles”, Seventh Edition, Prentice Hall, 2012
- Jean J Labrosse. “uC/OS-III, The Real-Time Kernel”, Third Edition, Micrium Press, 2009
- Jean J Labrosse. “uC/OS-II, The Real-Time Kernel”, Second Edition, CMP Books, 2002
- Richard Barry, “Using the FreeRTOS Real-Time Kernel – A practical Guide”, version 1.0.5, FreeRTOS.org, 2009
- Allen B. Downey, “The Little Book of Semaphores”, Second Edition, Green Tea Press, 2008
- OSEK/VDX steering committee, “OSEK/VDX Operating System Specification 2.2.3”, 2005
- Rieko, “Inside TOPPERS/ASP A Real-Time Operating System for Embedded Systems”, http://www.nces.is.nagoya-u.ac.jp/NEXCESS/blog_en/index.php?catid=4&blogid=4.
- Gerhard Fohler, “Operating Systems - 2011/2012 lecture slides”, Technische Universität Kaiserslautern
- Gerhard Fohler, “Real-Time Systems I - 2011 lecture slides”, Technische Universität Kaiserslautern
- Raphael Guerra, Gerhard Fohler, “Real-Time Systems II – 2011/2012 lecture slides”, Technische Universität Kaiserslautern



REAL-TIME OPERATING SYSTEMS

MAESTRÍA EN SISTEMAS INTELIGENTES MULTIMEDIA

INSTRUCTOR: CARLOS ENRIQUE DIAZ GUERRERO



AUTHOR: CARLOS ENRIQUE DIAZ GUERRERO

7/14/2018

1

OVERVIEW

- Real-Time Systems Concepts
- Real-Time Operating Systems Concepts
- Kernel Structure and Task Management
- Time Management
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- References

REAL-TIME OPERATING SYSTEMS CONCEPTS

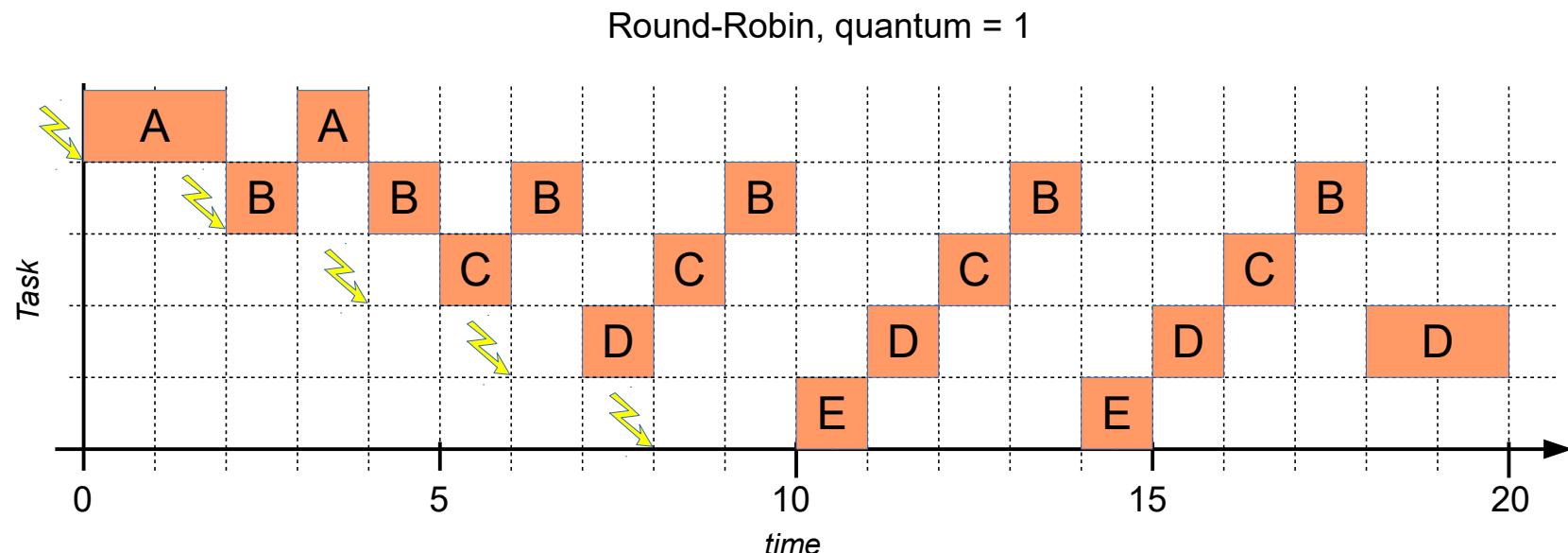
RR: Round-Robin

- Decision Mode: Preemptive
- There is a “**clock tick**” triggering the scheduling algorithm periodically. The period of this “clock tick” is known as the *quantum*.
- Selection Function: Every “clock tick” the current task is preempted by the task which has been longer in the **ready** queue
 - If two or more task have been the same “longer time” in the ready queue, then an additional criteria needs to be specified to select only one of them for execution.
- The ready queue is a FIFO (similar than FCFS)
- Priority handling: Priority of tasks is pretty much the same. At most, priority is given by the entrance to the ready queue.

REAL-TIME OPERATING SYSTEMS CONCEPTS

RR: Round-Robin

Task	Arrival Time	WCET
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



REAL-TIME OPERATING SYSTEMS CONCEPTS

RR: Round-Robin

- Design issue: difficult to select the proper *quantum*.
- Long quantum:
 - Probable long idle times (bad CPU utilization)
 - Slow system reaction
- Short quantum:
 - Short process will be processed quickly
 - Short quantum creates high overhead! (not a good thing to do)
- RR is good for **CPU-intensive** applications
- RR is bad for **I/O-intensive** applications

REAL-TIME OPERATING SYSTEMS CONCEPTS

CPU-intensive vs. I/O-intensive applications????

REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

- Decision Mode: Preemptive
- Selection Function: Shortest task period → highest priority
- Priority handling: **Fixed priority**. Priorities are decided “offline” according to the tasks periods: shorter period means higher priority.
 - **RMS** is an example of a Fixed Priority Scheduling (FPS) algorithm.

REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

- Assumptions
 - Tasks are periodic.
 - Tasks deadlines are equal to their periods ($D_i = T_i$)
 - Arrival time of tasks is the period start time.
 - Tasks do not suspend themselves
 - Tasks have bounded execution time (bounded C_i : $C_i \leq D_i$)
 - Tasks are independent
 - Scheduling overhead negligible

REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

- Example:

Task	Ti	Ci
A	13	3
B	15	2
C	9	1
D	16	2
E	10	2

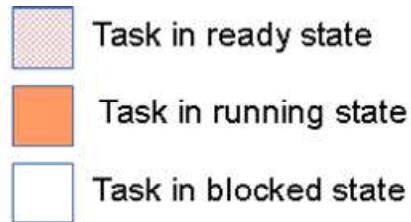
- Priorities are calculated based on tasks periods. The lower the number the higher the priority:

Task	Ti	Ci	Priority
A	13	3	3
B	15	2	4
C	9	1	1
D	16	2	5
E	10	2	2

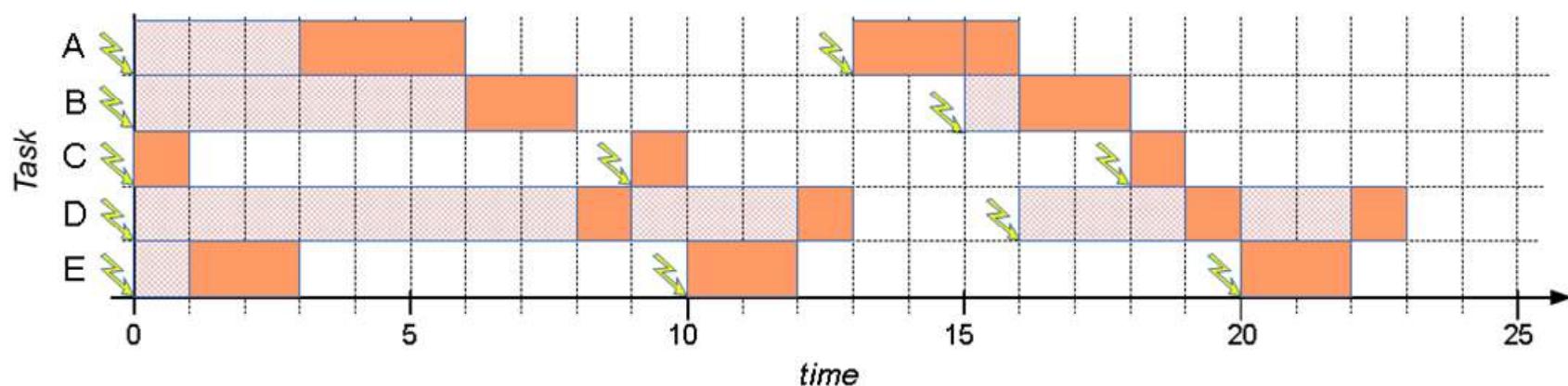
REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

- ... Example:



Task	Ti	Ci	Priority
A	13	3	3
B	15	2	4
C	9	1	1
D	16	2	5
E	10	2	2



REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

- **Sufficient** schedulability test for **RMS**:

$$\sum_{i=1}^n \frac{C_i}{T_i} \leq n \times (2^{1/n} - 1)$$

$$U \leq n \times (2^{1/n} - 1)$$

- **Necessary** schedulability test for **RMS**:

$$U \leq 1$$

REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

$$U \leq n \times (2^{1/n} - 1)$$

n	$n \times ((2^1/n) - 1)$
1	1
2	0.82843
3	0.77976
4	0.75683
...	...
∞	0.69315

- In RMS we ensure schedulability if utilization is less or equal than 69.31%

REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

- The given **sufficient** schedulability test for **RMS** is too conservative
 - Low CPU usage.
 - Previous example will fail this schedulability test but it is still schedulable!
 - **Critical Instance** analysis proves that it is schedulable.

REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

- Example 1: Is the following set schedulable under RMS?

Task	Ti	Ci	Priority
A	3	1	1
B	6	1	3
C	5	1	2
D	9	2	4

$$U = \frac{1}{3} + \frac{1}{6} + \frac{1}{5} + \frac{2}{8} = 0.95$$

$$U_{sufficient} = 0.76$$

$$0.95 \leq 0.76 \text{ ??} \quad \leftarrow \text{NO!!!}$$

Critical Instance analysis fails as well!

REAL-TIME OPERATING SYSTEMS CONCEPTS

RMS: Rate Monotonic Scheduling

■ Advantages:

- RMS is an **optimal** fixed priority scheduling policy:
 - If there exist another fixed priority assignment that makes a task set schedulable, RMS will do it too.
- Easy to implement
- Small overhead

■ Drawbacks:

- Artificial correlation: “period” and “priority”
- Low CPU utilization

REAL-TIME OPERATING SYSTEMS CONCEPTS

EDF: Earliest Deadline First

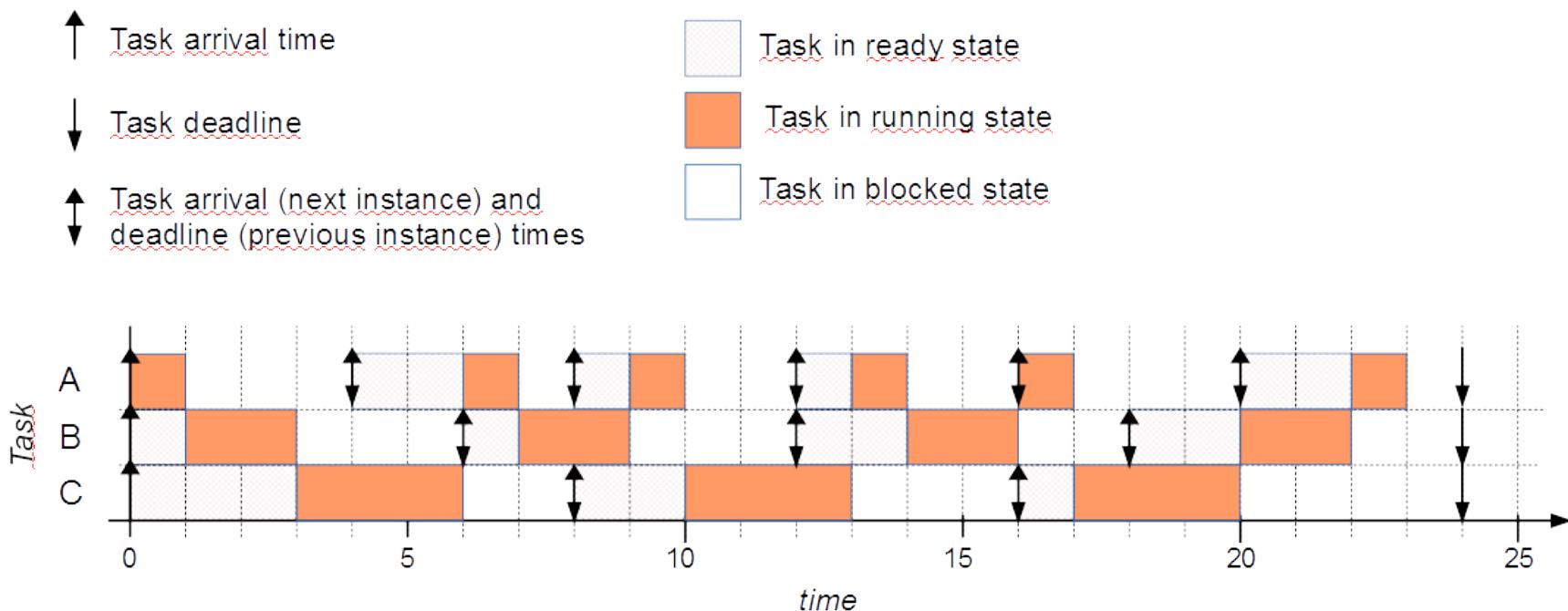
- Decision Mode: Preemptive
- Selection Function: Task with earliest absolute deadline first
- Priority handling: **Dynamic priority**. Priorities are decided at run time according to the tasks deadlines: The task with the closest deadline gets the higher priority.
 - Closest deadline → highest priority
 - Many tasks with same deadline? → Next randomly chosen (preferably chosen in a way that context switching is minimized).
- Tasks assumptions: same than RMS.

REAL-TIME OPERATING SYSTEMS CONCEPTS

EDF: Earliest Deadline First

- Example

Task	T _i	C _i
A	4	1
B	6	2
C	8	3



REAL-TIME OPERATING SYSTEMS CONCEPTS

EDF: Earliest Deadline First

- **Necessary and sufficient** schedulability condition for **EDF**:

$$U \leq 1$$

- Utilization calculation for the EDF example:

$$U = \frac{1}{4} + \frac{2}{6} + \frac{3}{8} = 0.96$$

$$0.96 \leq 1$$

- In general EDF provides higher utilization than RMS
- EDF is optimal if utilization is less than 1; performs badly (worst than RMS) with overload.

REAL-TIME OPERATING SYSTEMS CONCEPTS

EDF: Earliest Deadline First

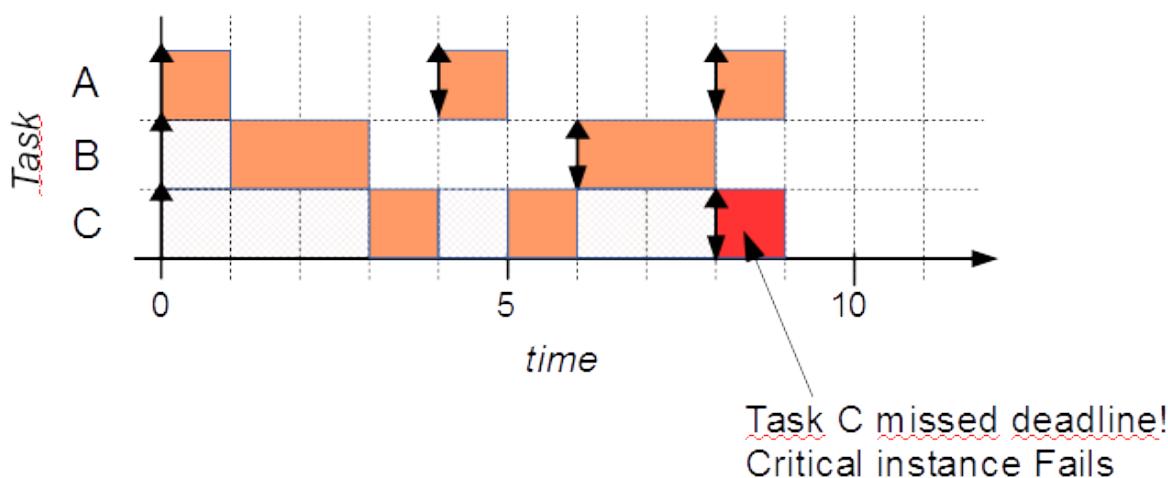
- **EDF** advantages:
 - Higher utilization than RMS (less idle times, etc.)
 - Optimal for utilization smaller than 1
- **EDF** Drawbacks
 - **EDF** performs worst than RMS under overload ($U \geq 1$)
 - High overhead due to the dynamic prioritization (sorting ready list, etc.)

REAL-TIME OPERATING SYSTEMS CONCEPTS

EDF vs RMS

- Example of EDF now for RMS:
 - Sufficient test fails: $0.96 \geq 0.78$
 - Critical instance???

Task	Ti	Ci	Priority
A	4	1	1
B	6	2	2
C	8	3	3





REAL-TIME OPERATING SYSTEMS CONCEPTS

Tasks/ISRs Interactions



REAL-TIME OPERATING SYSTEMS CONCEPTS

- Recalling → Task can be considered as “**independent**” threads of execution.
- However, since multiple tasks are indeed serving a higher-level purpose, the job to be done, often they need to interact with each other.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Multiple **tasks** can **share**:
 - A. Code** → functions
 - B. Data** → variables, constants, ...
 - C. Resources** → I/Os, HW peripherals, etc.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Multiple tasks may need to **synchronize** with each other
- Multiple tasks may need to **communicate** with each other
- Tasks may need to interact with **ISRs**
- Everything applicable for **inter-tasks** interactions also applies for **tasks-ISRs** interactions!!

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Artifacts to overcome **code sharing** issues:
 - Re-entrant functions
- Artifacts to overcome **data sharing** issues:
 - Critical sections
 - Semaphores
 - Mutexes

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Artifacts for **synchronization/communication**:
 - Semaphores
 - Mutexes
 - Event queues
 - Message queues

REAL-TIME OPERATING SYSTEMS CONCEPTS

A. Code sharing

- Multiple Tasks and ISRs may need to call the same piece of code (the same function)
 - → Only Possible when such functions are **reentrant**
- A function is **reentrant** if it can be interrupted at the middle of its execution and then *safely* called again before its previous invocations complete execution.
- *Safely* in this context means that there is no data corruption among the multiple calls.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Sharing code by itself is a simple thing
 - Just by “moving” the program counter we are re-using code!...
- However, the difficulty resides in the data handled by such code.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Rules for writing **reentrant** code are:
 1. Reentrant code may not hold any static (or global) variable data.
 - If so, atomic operations need to be used and special care of the “meaning” of such data.
 2. Reentrant function may no modify its code
 - Possible in non real-time OS but in RTOS this is not common anyways (program is in ROM, indeed read-only and not “modifiable” like in the RAM of a computer).
 3. Reentrant code may not call non-reentrant code

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Example of **non-reentrant** function

```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- In this example just for simplicity we will assume that **each C sentence executes atomically**. Later we will see that in reality this is actually NOT true.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

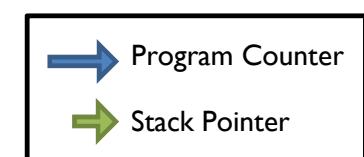
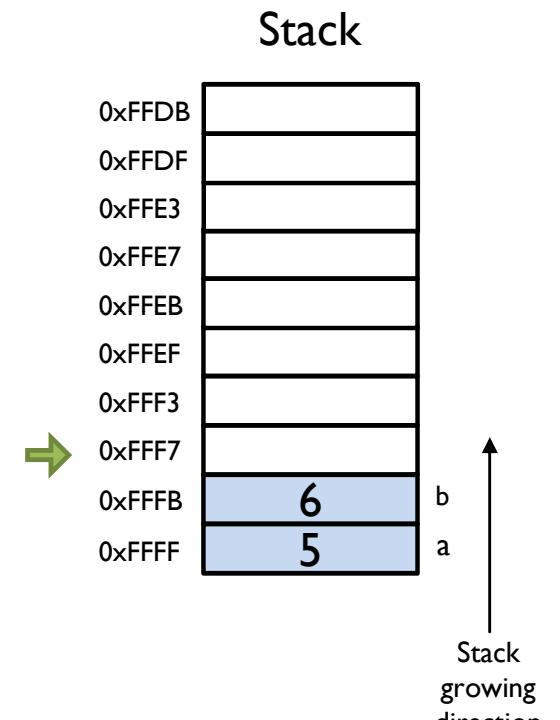
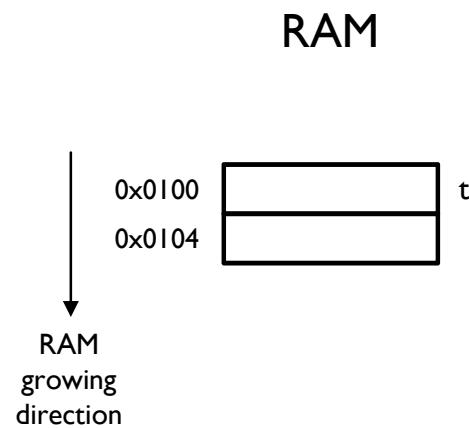
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

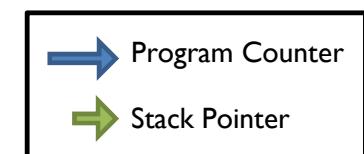
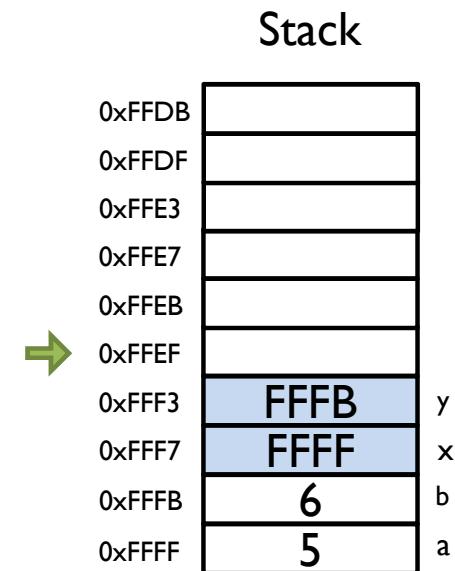
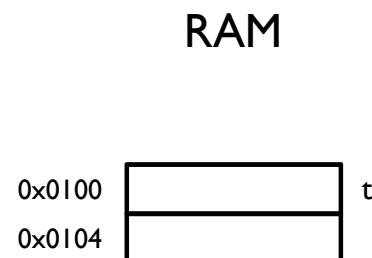
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

```
int t;

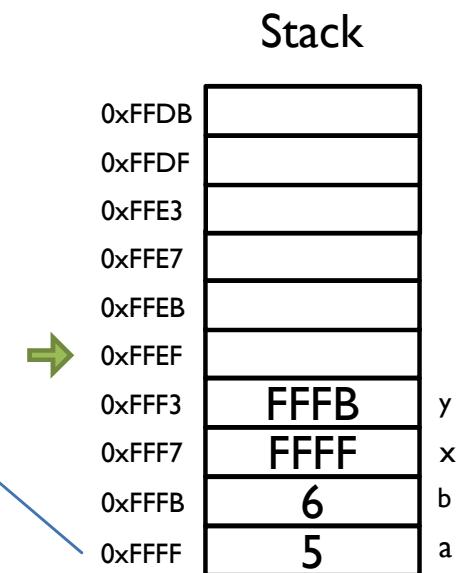
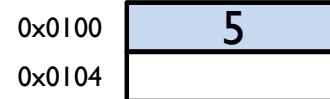
void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```

RAM



→ Program Counter
→ Stack Pointer

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

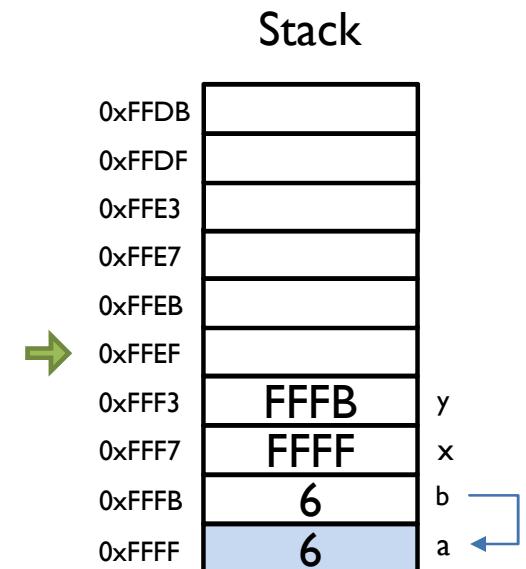
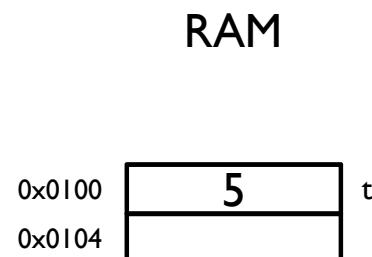
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



→ Program Counter
→ Stack Pointer

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

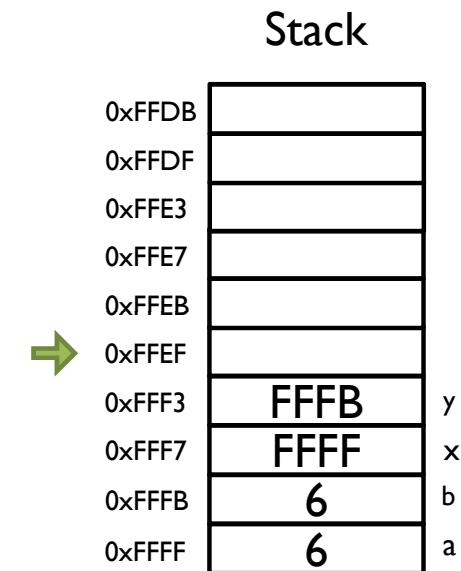
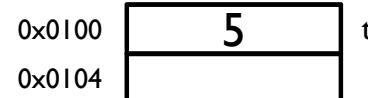
void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



Interrupt occurred!

RAM



→ Program Counter
→ Stack Pointer

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

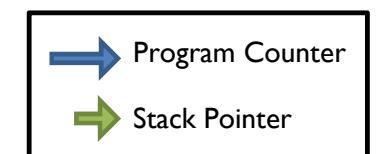
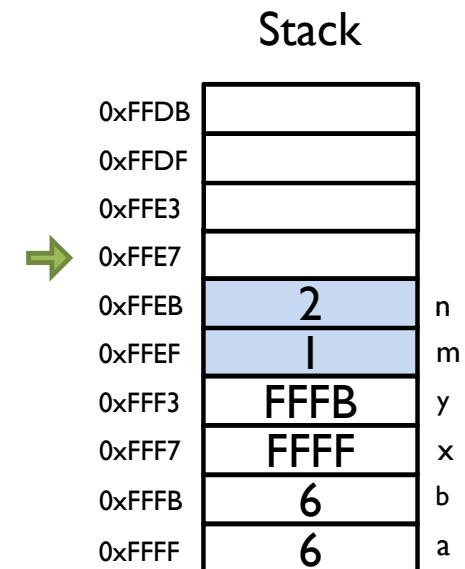
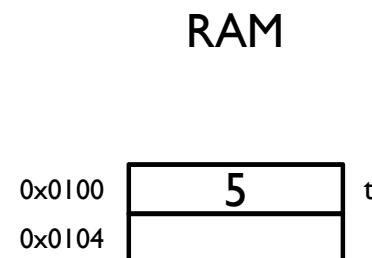
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

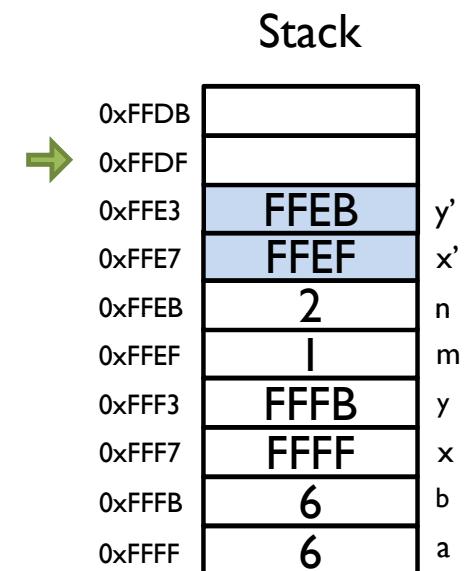
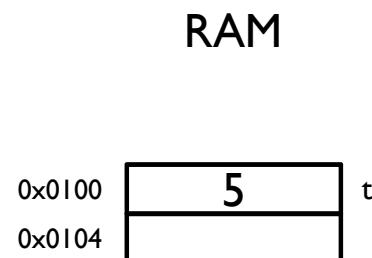
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

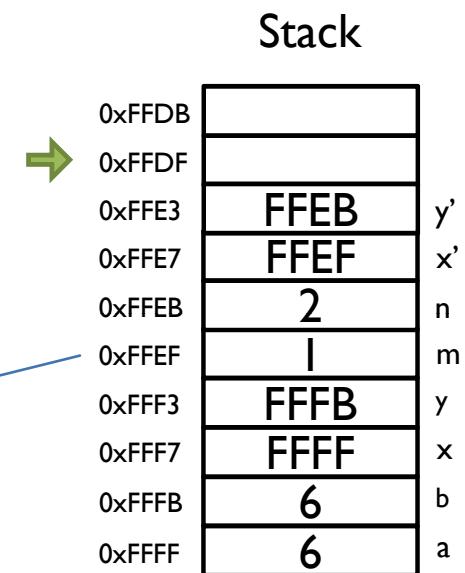
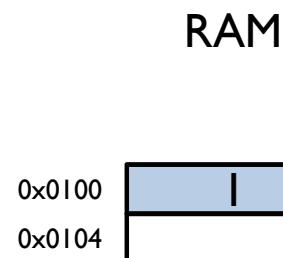
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



→ Program Counter
→ Stack Pointer

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

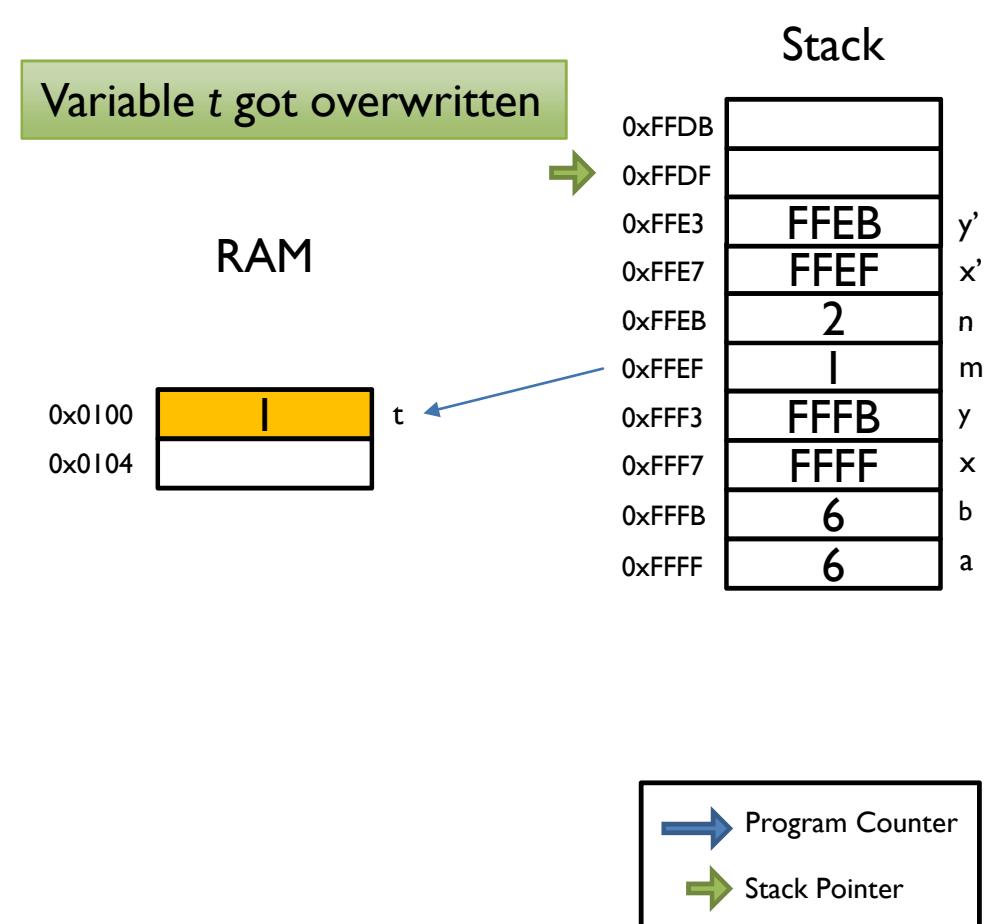
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

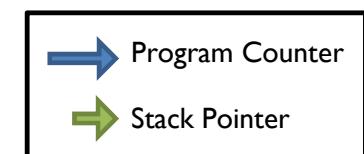
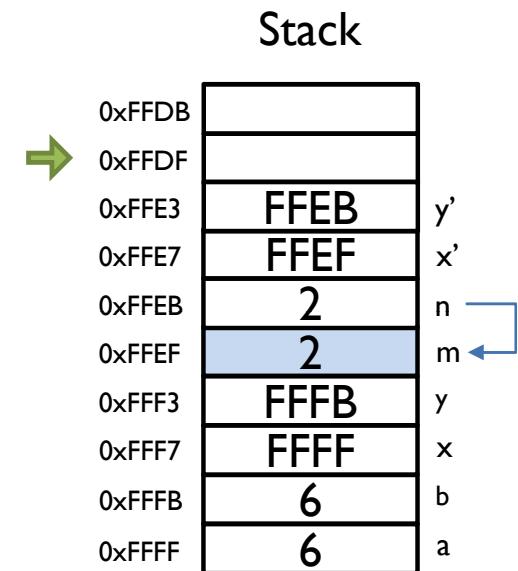
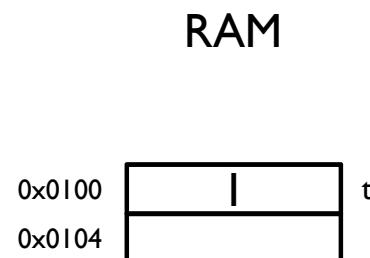
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

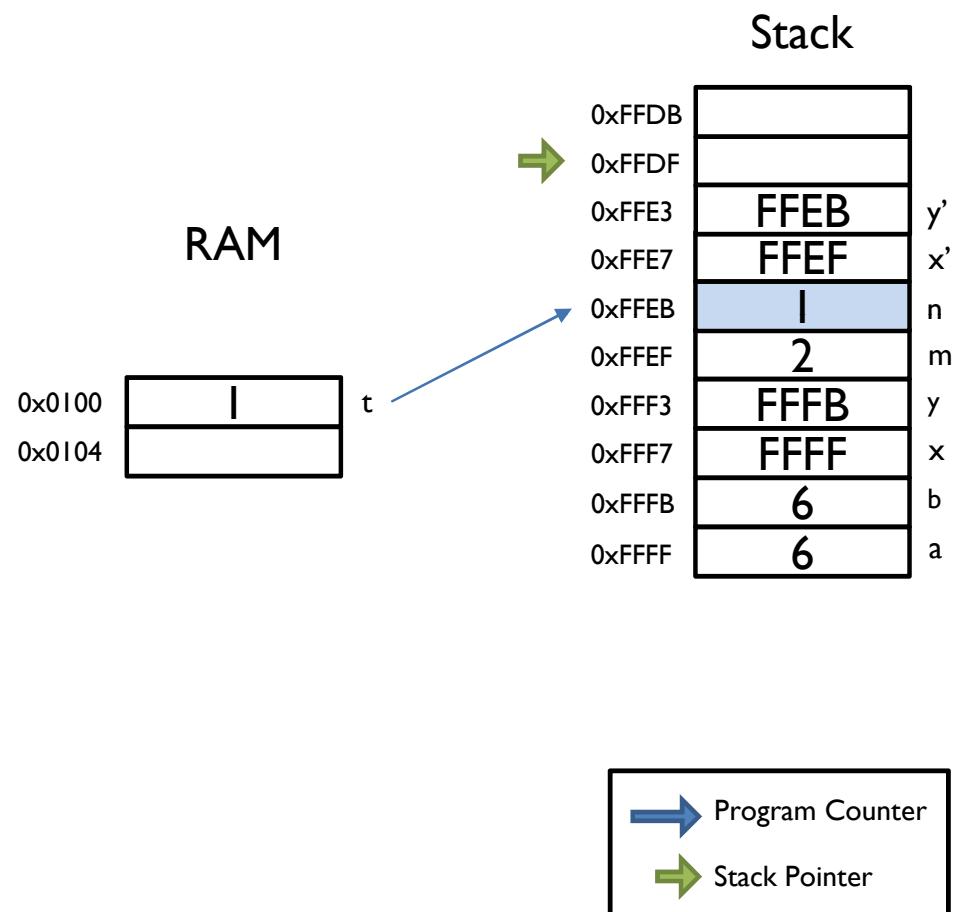
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

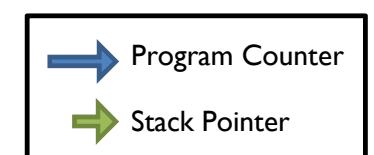
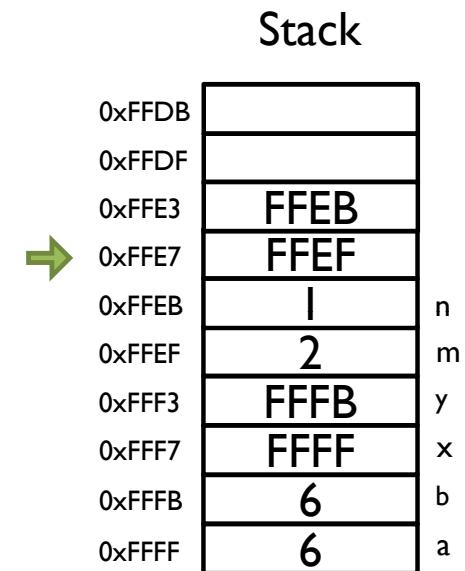
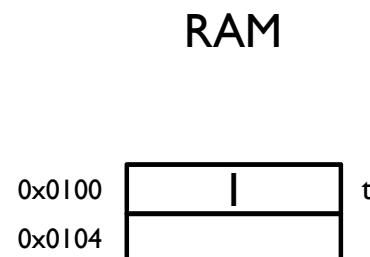
```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

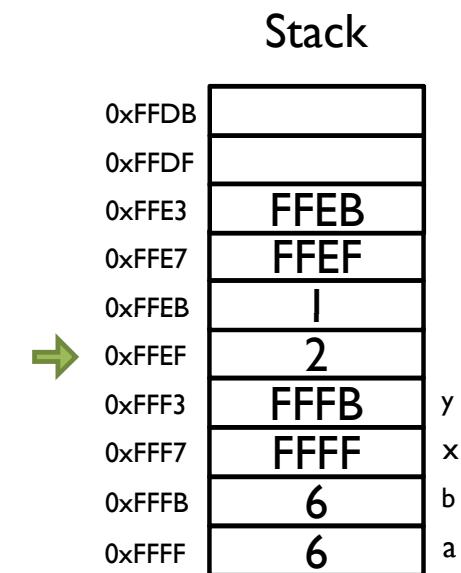
    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```

Return from interrupt

RAM



Program Counter
Stack Pointer

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

```
int t;

void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

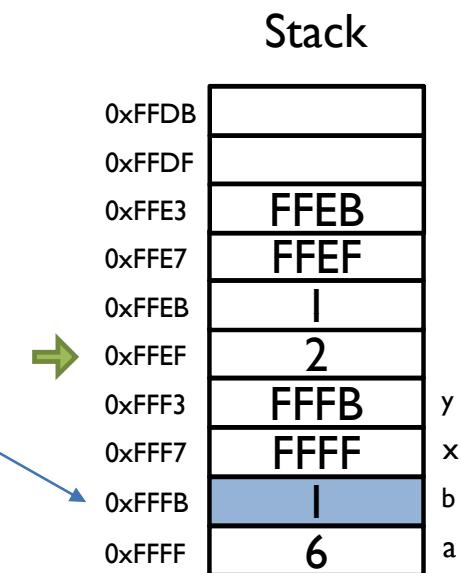
    // isr() may be invoked here!
    *y = t;

    → }

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```

RAM



→ Program Counter
→ Stack Pointer

REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Example of **non-reentrant** function ...

```
int t;

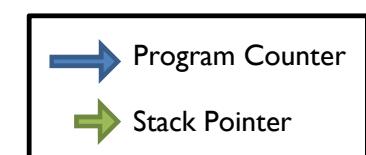
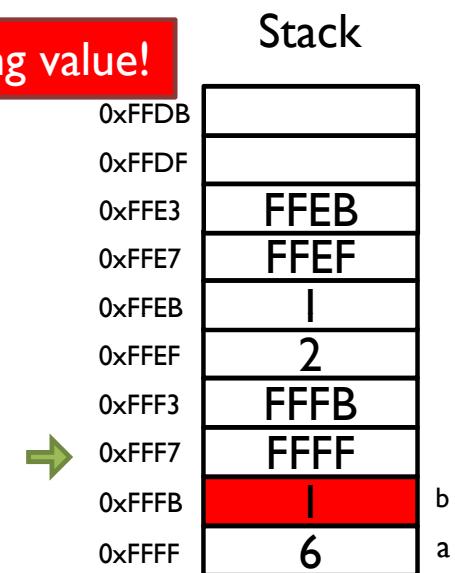
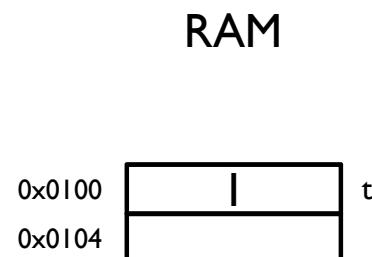
void swap(int *x, int *y)
{
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
    }
```

Variable “b” got the wrong value!



REAL-TIME OPERATING SYSTEMS CONCEPTS

- Re-writing the Example to make it a **reentrant** function

```
int t;

void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

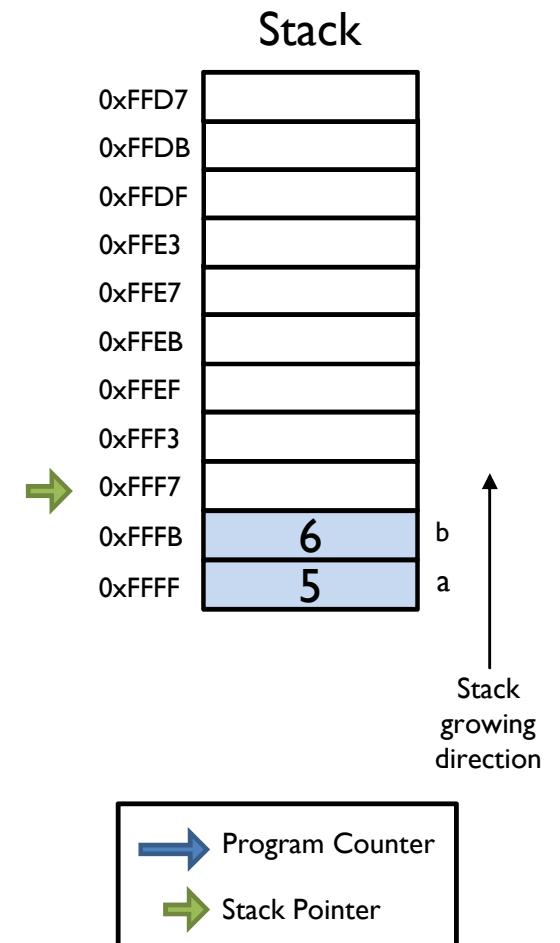
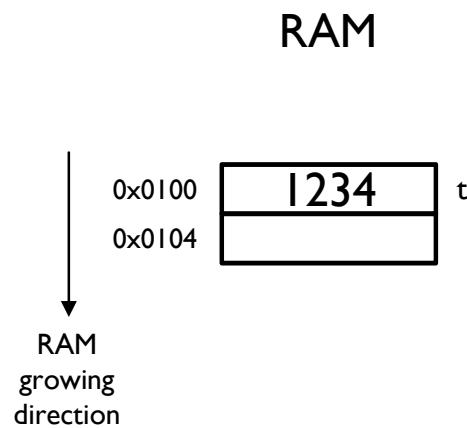
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

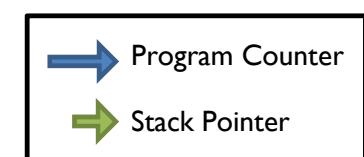
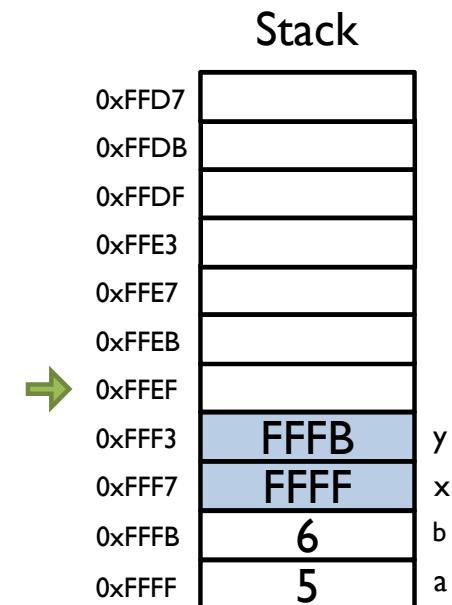
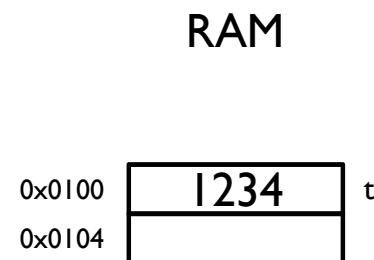
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

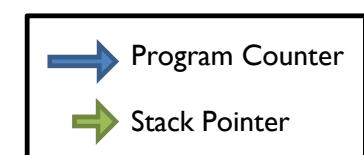
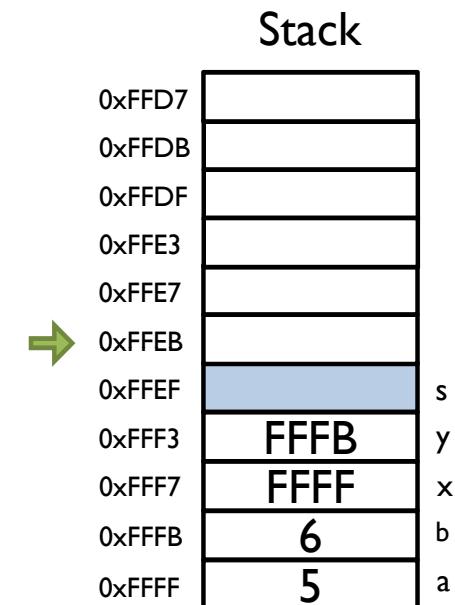
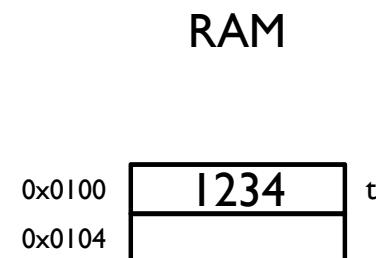
void swap(int *x, int *y)
{
    int s;

    → s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

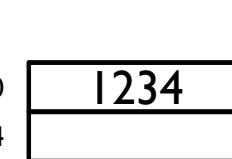
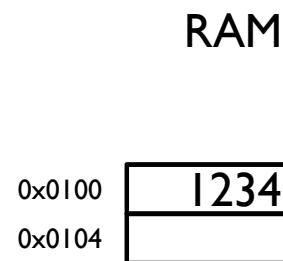
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

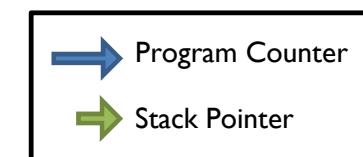
    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



t



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

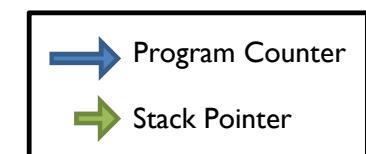
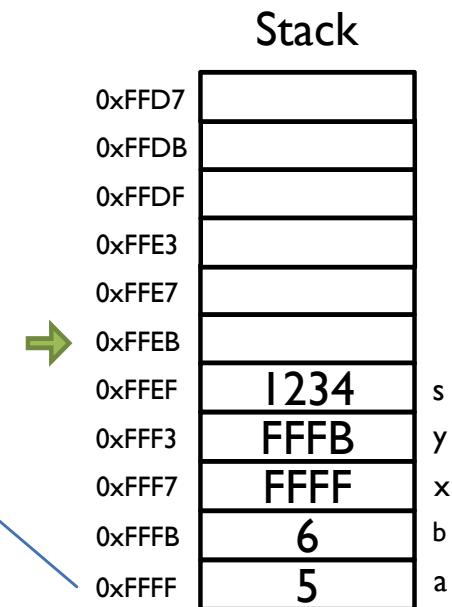
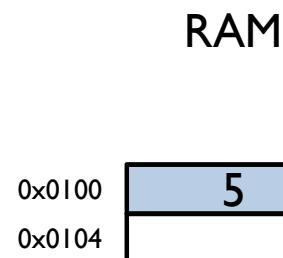
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

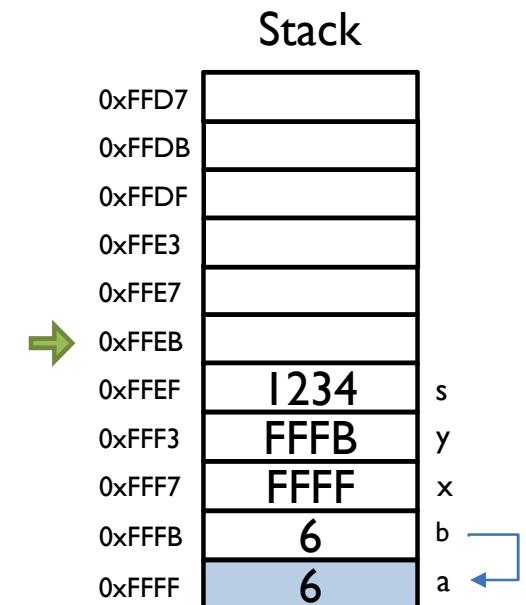
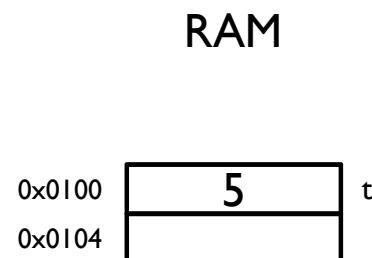
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

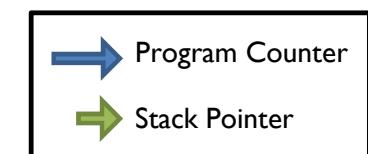
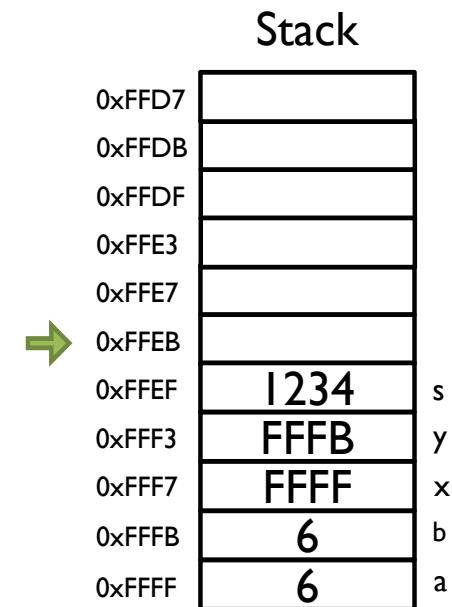
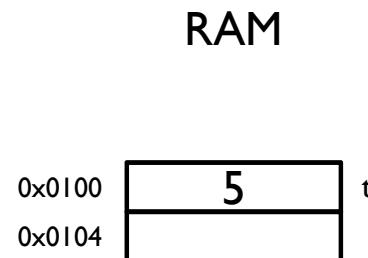
    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



Interrupt occurred!



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

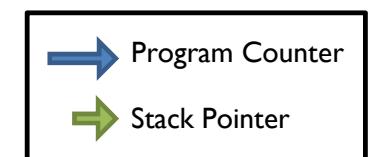
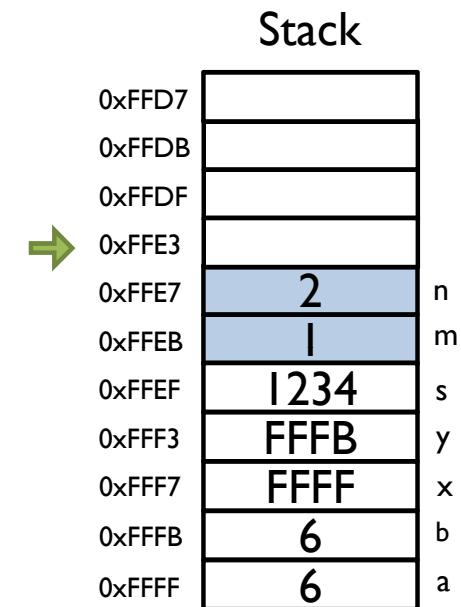
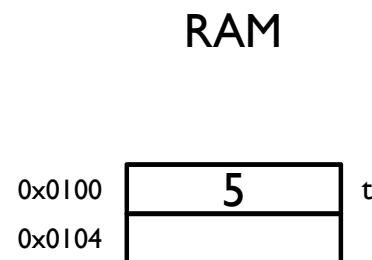
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

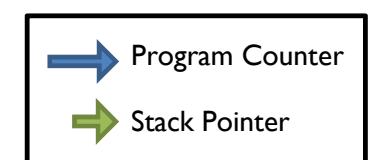
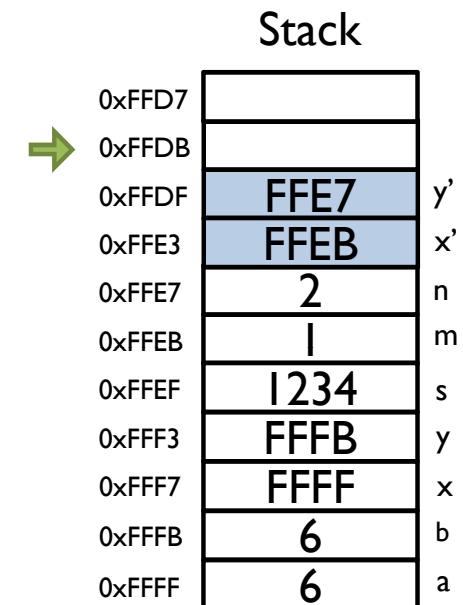
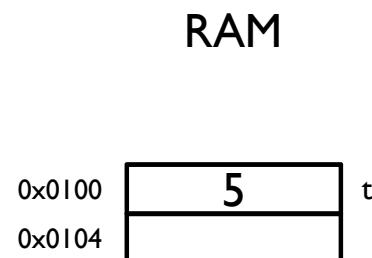
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

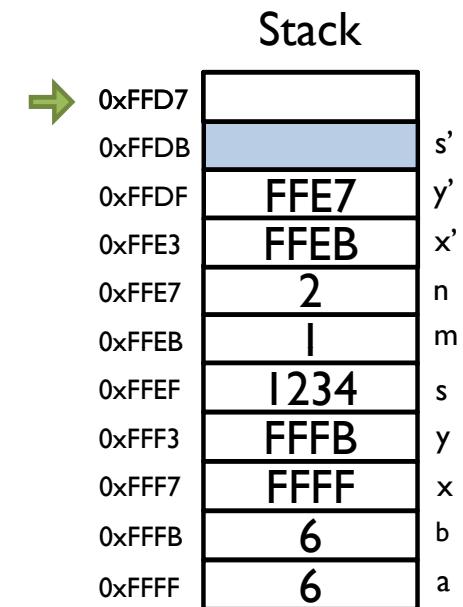
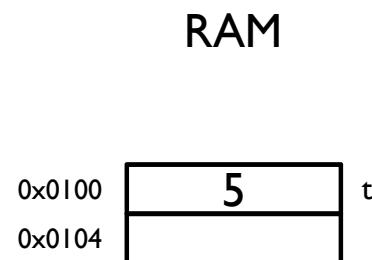
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

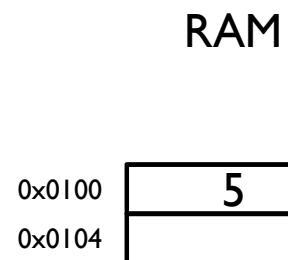
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

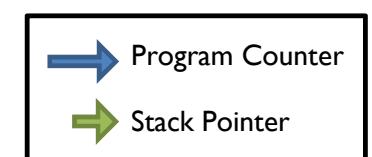
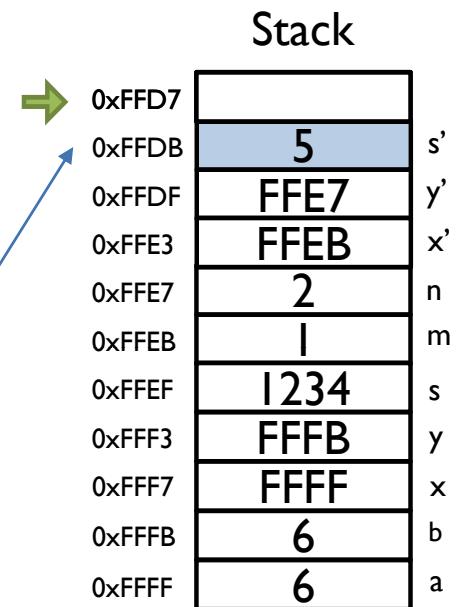
    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



t



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

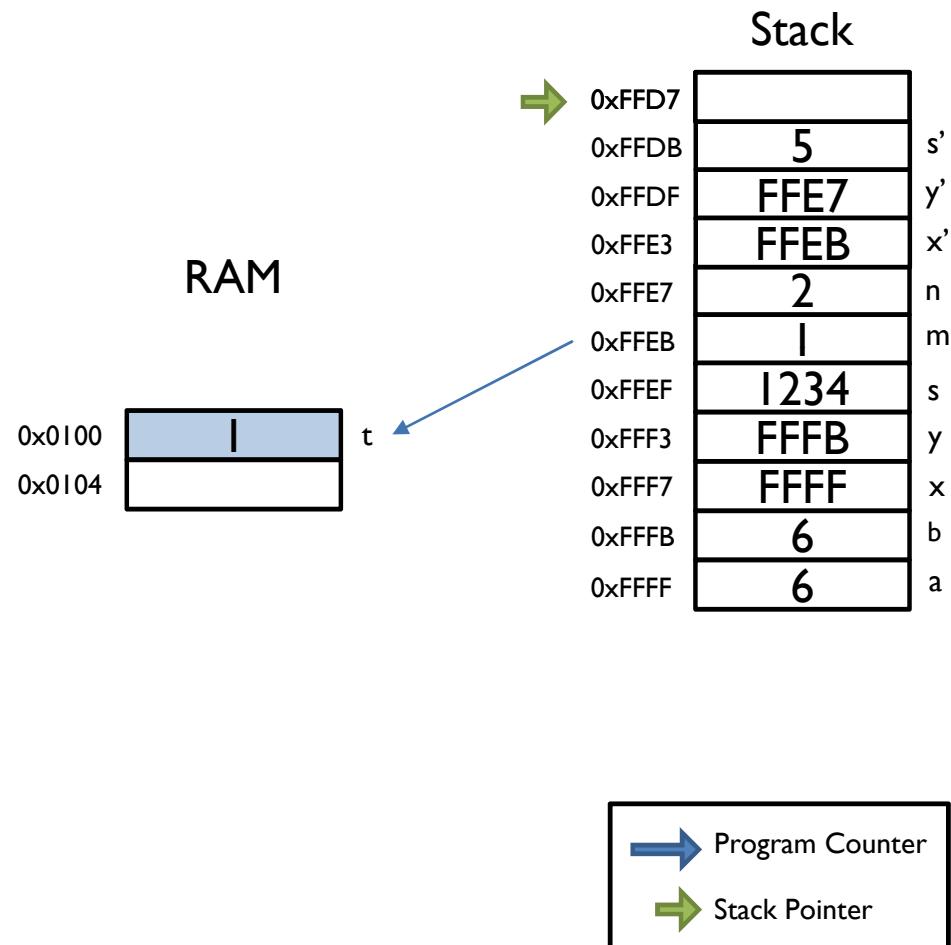
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

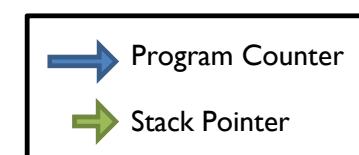
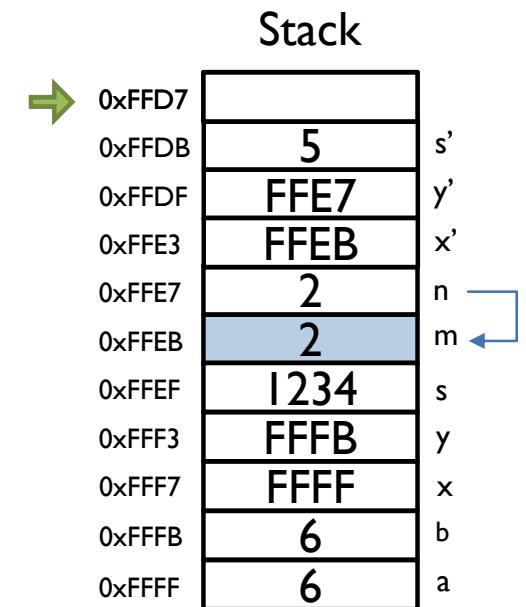
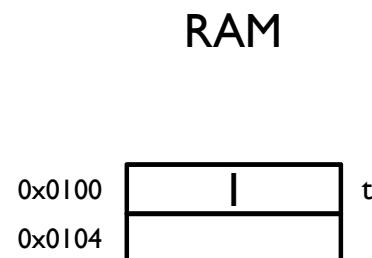
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

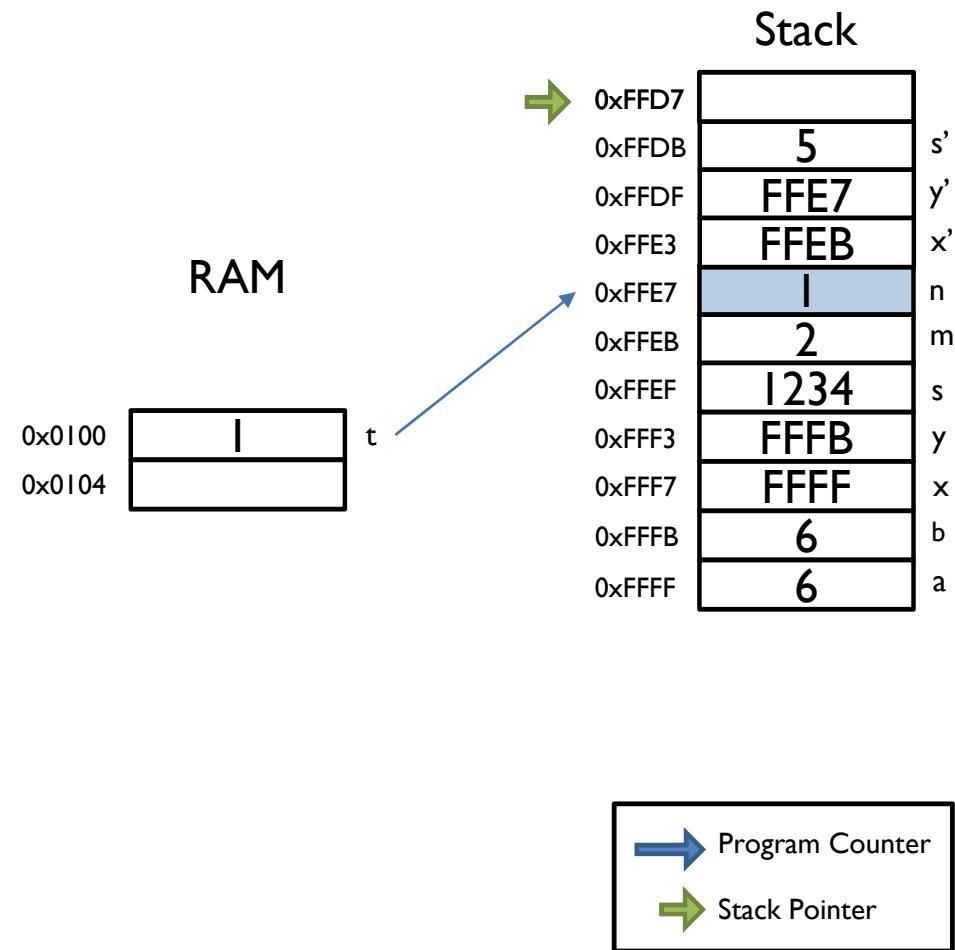
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

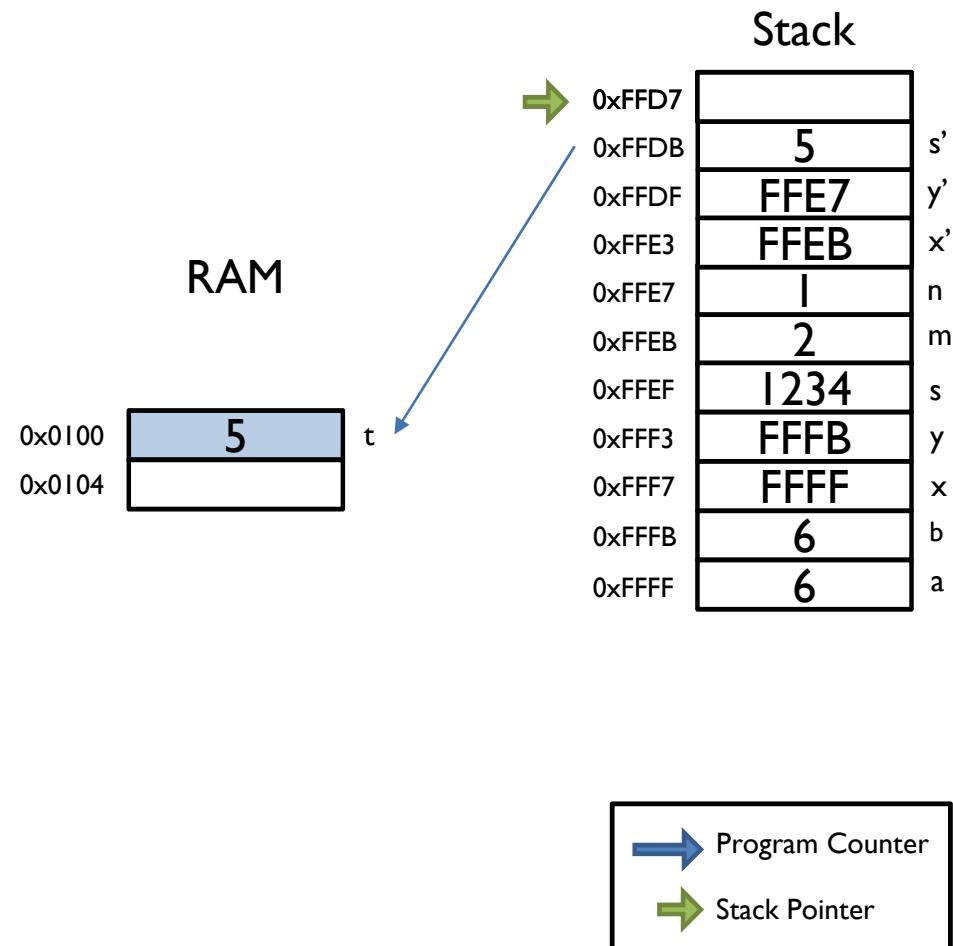
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

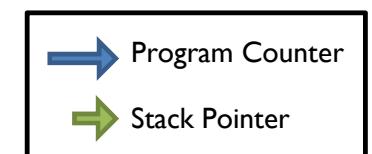
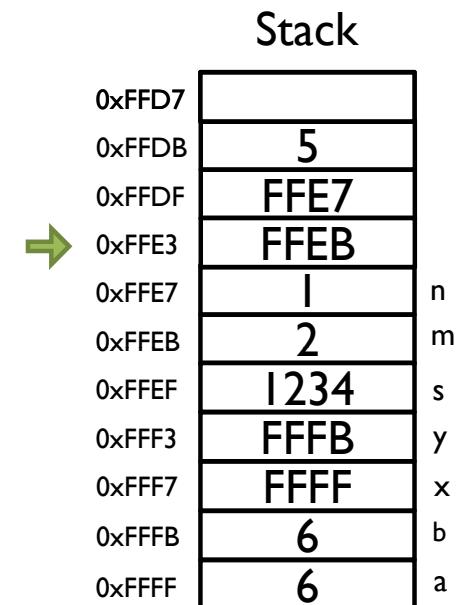
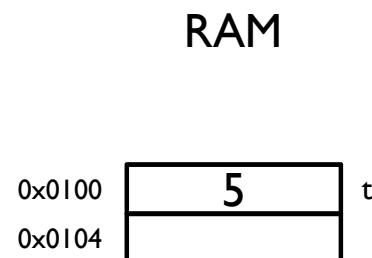
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

void swap(int *x, int *y)
{
    int s;

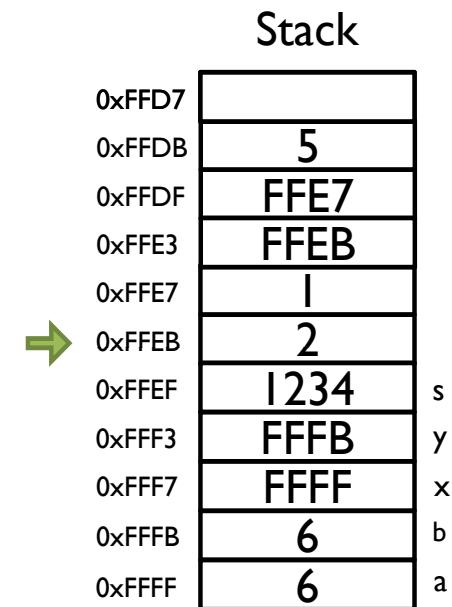
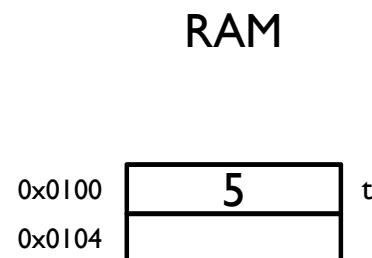
    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```

Return from interrupt



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

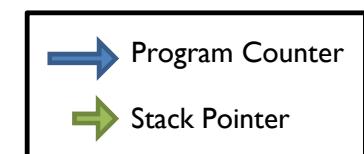
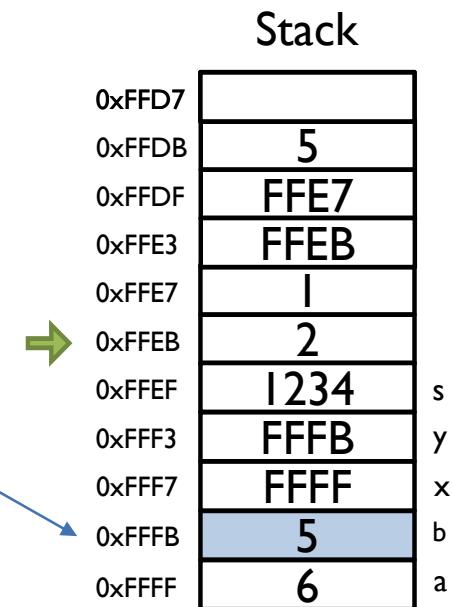
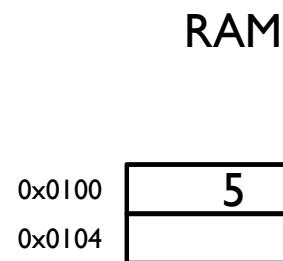
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

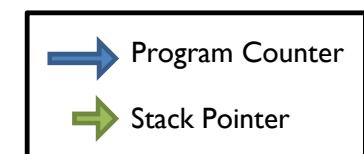
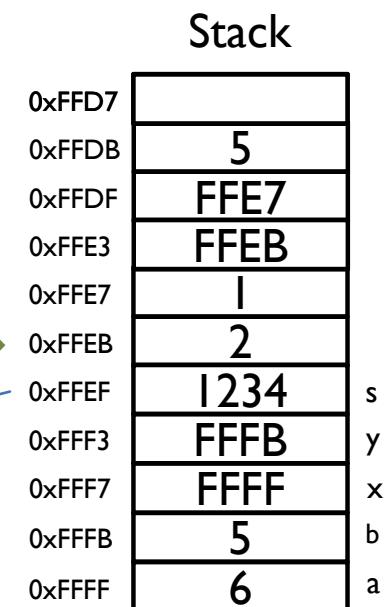
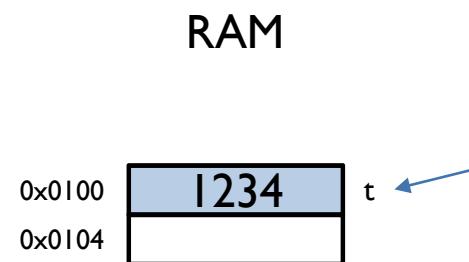
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

- ... Re-writing as **reentrant** function ...

```
int t;

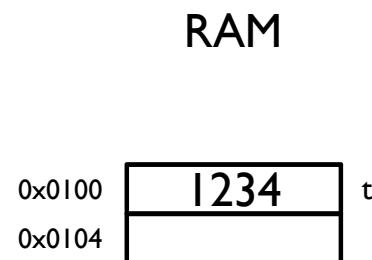
void swap(int *x, int *y)
{
    int s;

    s = t; //save global variable
    t = *x;
    *x = *y;

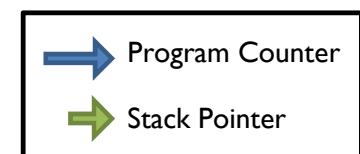
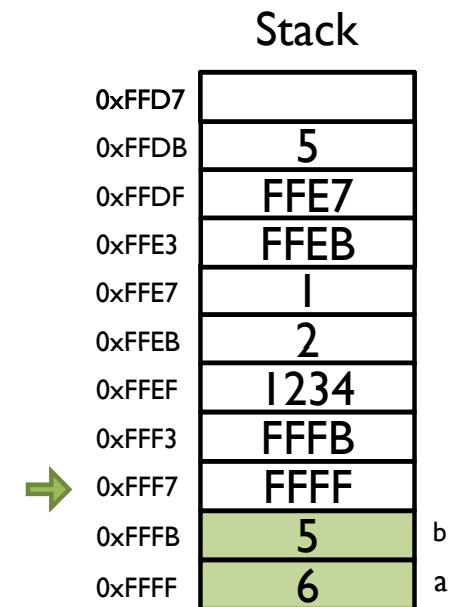
    // isr() may be invoked here!
    *y = t;
    t = s; //restore global variable
}

void isr()
{
    int m = 1, n = 2;
    swap(&m, &n);
}

void main()
{
    int a = 5, b = 6;
    swap(&a, &b);
}
```



Variables a and b swapped correctly!



REAL-TIME OPERATING SYSTEMS CONCEPTS

B. Data sharing

- Multiple Tasks and ISRs often need to access (write/read) shared **data**
→ Global Variables
- Data corruption may occur if no special care is taken!

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Consider that:
 - a) Accessing global data (in RAM) normally takes more than one MCU instruction to complete.
 - And normally CPU registers are “re-used”
 - b) By means of the RTOS scheduler, a *ready* Task may interrupt a *running* Task **at any time!**
 - c) RAM data (as opposed to local data) is NOT part of the Task context (this will be explained later) and hence nothing is done by the RTOS itself to protect accesses to it.
 - Application developer needs to take care of it
- Because of these things is easy to “**corrupt**” global data when working with RTOS if no special care is taken!

REAL-TIME OPERATING SYSTEMS CONCEPTS

Data **corruption** example:

- Let's assume two simple tasks (**Task1_task** and **Task2_task**) wanting to write to a shared global variable (**myGlobalVariable**).

```
/* Global variable */
volatile long long int myGlobalVariable = 0;

/* Task 1 */
void Task1_task(uint32_t task_init_data)
{
    while(1) {
        myGlobalVariable = 10;

        _time_delay_ticks(200);
        /* Write your code here ... */
    }
}

/* Task 2 */
void Task2_task(uint32_t task_init_data)
{
    while(1) {
        myGlobalVariable = 20;

        _time_delay_ticks(100);
        /* Write your code here ... */
    }
}
```

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

- In this example:
 - “long long int” data type is used for the global variable to accentuate the need of multiple instructions for the data access.
 - “volatile” is used to avoid compiler optimizations
 - Using Freescale FRDM-KL25Z board with an ARM Cortex m0+ MCU.
 - Using ARM GCC compiler
 - Using Freescale MQXLite RTOS

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

- The **disassembly output** for the global data access is as follows:

```
/* Task 1 */
void Task1_task(uint32_t task_init_data)
{
    while(1) {
        myGlobalVariable = 10;
        _time_delay_ticks(200);
        /* Write your code here ... */
    }
}
```

```
myGlobalVariable = 10;
2:4e04 ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a movs r4, #10
6:2500 movs r5, #0
8:6034 str r4, [r6, #0]
a:6075 str r5, [r6, #4]
```

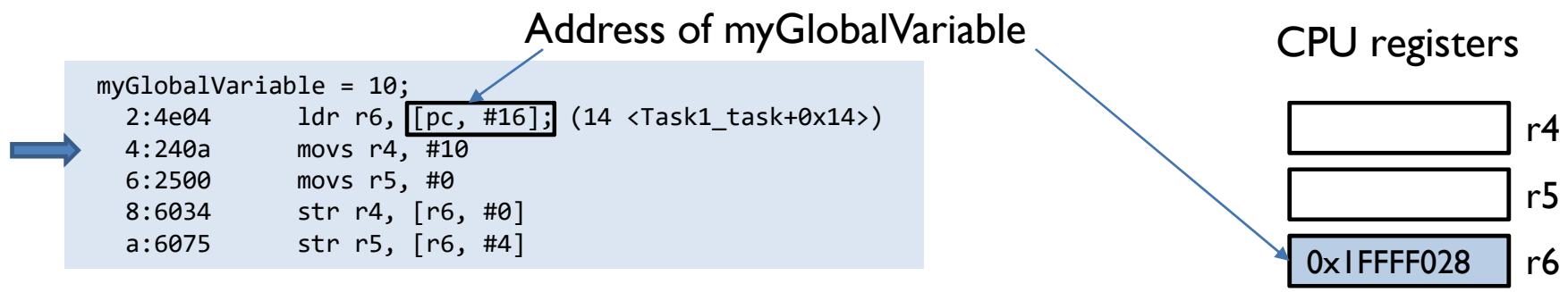
```
/* Task 2 */
void Task2_task(uint32_t task_init_data)
{
    while(1) {
        myGlobalVariable = 20;
        _time_delay_ticks(100);
        /* Write your code here ... */
    }
}
```

```
myGlobalVariable = 20;
2:4e04 ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414 movs r4, #20
6:2500 movs r5, #0
8:6034 str r4, [r6, #0]
a:6075 str r5, [r6, #4]
```



REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...



```
myGlobalVariable = 20;
2:4e04  ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414  movs r4, #20
6:2500  movs r5, #0
8:6034  str r4, [r6, #0]
a:6075  str r5, [r6, #4]
```

RAM

myGlobalVariable
0 0xFFFFF028

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```



CPU registers

10	r4
	r5
0x1FFFF028	r6

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

RAM

myGlobalVariable	0	0x1FFFF028
------------------	---	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;  
2:4e04    ldr r6, [pc, #16]; (14 <Task2_task+0x14>)  
4:240a    movs r4, #10  
6:2500    movs r5, #0  
8:6034    str r4, [r6, #0]  
a:6075    str r5, [r6, #4]
```

RTOS switches execution
to Task 2



CPU registers

10	r4
	r5
0x1FFFF028	r6

```
myGlobalVariable = 20;  
2:4e04    ldr r6, [pc, #16]; (14 <Task2_task+0x14>)  
4:2414    movs r4, #20  
6:2500    movs r5, #0  
8:6034    str r4, [r6, #0]  
a:6075    str r5, [r6, #4]
```

RAM

myGlobalVariable	0	0x1FFFF028
------------------	---	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

CPU registers

10	r4
	r5
0x1FFFF028	r6

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

RAM

myGlobalVariable	0	0x1FFFF028
------------------	---	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

CPU registers

20	r4
	r5
0x1FFFF028	r6

myGlobalVariable = 20;
2:4e04 ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414 movs r4, #20
6:2500 movs r5, #0
8:6034 str r4, [r6, #0]
a:6075 str r5, [r6, #4]



RAM

myGlobalVariable	0	0x1FFFF028
------------------	---	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

CPU registers

20	r4
0	r5
0x1FFFF028	r6

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```



RAM

myGlobalVariable	0	0x1FFFF028
------------------	---	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```



CPU registers

20	r4
0	r5
0xFFFFF028	r6

RAM

myGlobalVariable	20	0xFFFFF028
------------------	----	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

CPU registers

20	r4
0	r5
0xFFFFF028	r6

RAM

myGlobalVariable	0020	0xFFFFF028
------------------	------	------------



REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

Rest of Task 2 code runs and
RTOS gives back execution
to Task 1

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

CPU registers

20	r4
0	r5
0xFFFFF028	r6

RAM

0020	myGlobalVariable	0xFFFFF028
------	------------------	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```



CPU registers

20	r4
0	r5
0xFFFFF028	r6

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

RAM

myGlobalVariable	0020	0xFFFFF028
------------------	------	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```



```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

CPU registers

20	r4
0	r5
0xFFFFF028	r6

RAM

myGlobalVariable	0020	0xFFFFF028
------------------	------	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)
4:240a      movs r4, #10
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

```
myGlobalVariable = 20;
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)
4:2414      movs r4, #20
6:2500      movs r5, #0
8:6034      str r4, [r6, #0]
a:6075      str r5, [r6, #4]
```

CPU registers

20	r4
0	r5
0xFFFFF028	r6

RAM

myGlobalVariable	0020	0xFFFFF028
------------------	------	------------

REAL-TIME OPERATING SYSTEMS CONCEPTS

...Data **corruption** example...

```
myGlobalVariable = 10;  
2:4e04      ldr r6, [pc, #16]; (14 <Task1_task+0x14>)  
4:240a      movs r4, #10  
6:2500      movs r5, #0  
8:6034      str r4, [r6, #0]  
a:6075      str r5, [r6, #4]
```

Task 1 intended to write
myGlobalVariable with 10 but it
will actually read it as 20!!!
Data Corruption!!

```
myGlobalVariable = 20;  
2:4e04      ldr r6, [pc, #16]; (14 <Task2_task+0x14>)  
4:2414      movs r4, #20  
6:2500      movs r5, #0  
8:6034      str r4, [r6, #0]  
a:6075      str r5, [r6, #4]
```

CPU registers

20	r4
0	r5
0xFFFFF028	r6

RAM

myGlobalVariable
0020 0xFFFFF028

REAL-TIME OPERATING SYSTEMS CONCEPTS

- **Q:** How do we protect shared data accesses against data corruption??
- **A:** We need to make such accesses in an **atomic** way!

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Atomic means “indivisibly”
- We shall **NOT** allow the code for accessing shared data to be interrupted by any other Task or ISR before it finishes!
- Code instructions prevented from “interruption” are said to be placed within a **critical section**.
- **Critical section** of code is a set of code instructions which are executed indivisibly, this means they are protected from being interrupted.

REAL-TIME OPERATING SYSTEMS CONCEPTS

- The most common way to implement critical sections is by **disabling interrupts** while the protected code gets executed and then **enabling interrupts** afterwards.

```
/* Unprotected (divisible) code ... */  
  
CriticalSection {  
    DISABLE_GLOBAL_INTERRUPTS(); /* Enter Critical Section */  
  
    /* Indivisible code goes here */  
  
    ENABLE_GLOBAL_INTERRUPTS(); /* Exit Critical Section */  
  
    /* Unprotected (divisible) code ... */
```

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Previous example now **without** data corruption!

```
/* Global variable */  
volatile long long int myGlobalVariable = 0;  
  
/* Task 1 */  
void Task1_task(uint32_t task_init_data)  
{  
    while(1) {  
  
        DISABLE_GLOBAL_INTERRUPTS();  
        myGlobalVariable = 10;  
        ENABLE_GLOBAL_INTERRUPTS();  
  
        _time_delay_ticks(200);  
        /* Write your code here ... */  
    }  
}  
  
/* Task 2 */  
void Task2_task(uint32_t task_init_data)  
{  
    while(1) {  
  
        DISABLE_GLOBAL_INTERRUPTS();  
        myGlobalVariable = 20;  
        ENABLE_GLOBAL_INTERRUPTS();  
  
        _time_delay_ticks(100);  
        /* Write your code here ... */  
    }  
}
```

REAL-TIME OPERATING SYSTEMS CONCEPTS

- A more proper implementation of a **critical zone** will:
 1. **Save** the current Enable/Disable state of interrupts
 2. **Disable** Interrupts
 3. **Execute** code within the critical section
 4. **Restore** the saved state of interrupts
- RTOS normally provide APIs to enter and exit critical sections
 - μC/OS: OS_CRITICAL_ENTER(), OS_CRITICAL_EXIT(), OS_CRITICAL_EXIT_NO_SCHED()
 - FreeRTOS: taskENTER_CRITICAL(), taskEXIT_CRITICAL()
 - OSEK: SuspendAllInterrupts() / ResumeAllInterrupts()

REAL-TIME OPERATING SYSTEMS CONCEPTS

- Critical Sections shall be **small!**
 - Long **interrupt suppression times** are not good! → missing events
- Alternatives for creating critical sections:
 - If data is **only** shared among **tasks** (but not with any **ISR**) we can just “**lock**” the scheduler (prevent task switching) and still allow interrupts.
 - Use of semaphores and mutexes (explained later).

OVERVIEW

- Real-Time Systems Concepts
- Real-Time Operating Systems Concepts
- **Kernel Structure and Task Management**
- Time Management
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- References

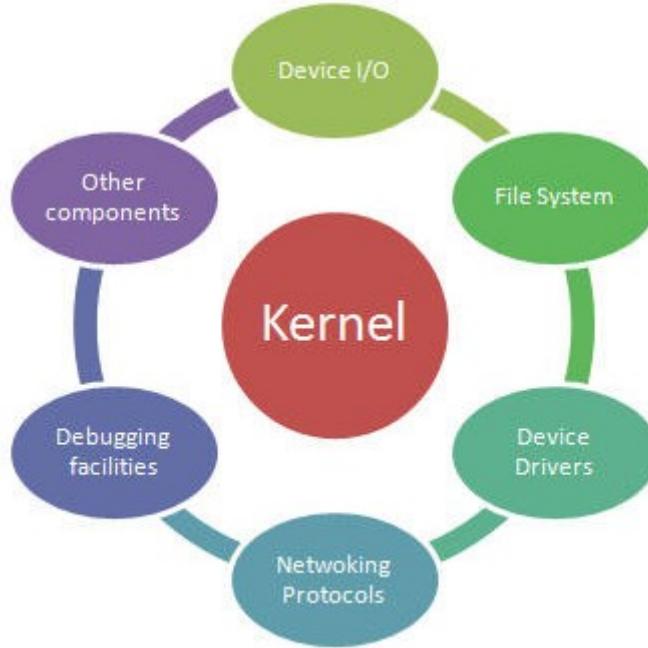


KERNEL STRUCTURE AND TASK MANAGEMENT

RTOS Kernel

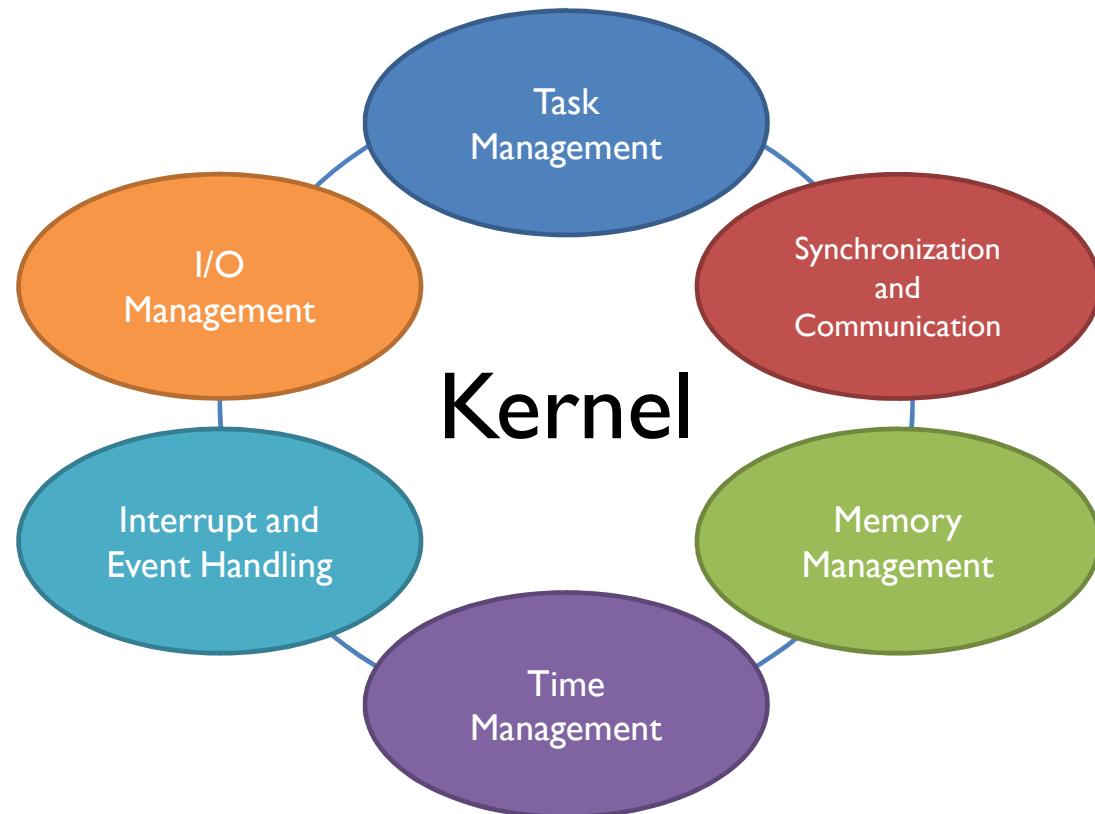
KERNEL STRUCTURE AND TASK MANAGEMENT

- The central indispensable component of an RTOS is the **kernel**
- Other possible components depend on the “size” of the RTOS



KERNEL STRUCTURE AND TASK MANAGEMENT

Main **Kernel** services



KERNEL STRUCTURE AND TASK MANAGEMENT

Main **Kernel** services

- Task Management
 - Tasks scheduler, dispatcher, ...
- Synchronization and communication
 - Semaphores, mutexes, ...
- Memory Management
 - Privileged / User level memory management (for MCUs with MMU units)
- Time Management
- Interrupt and Event handling
- I/O Management



KERNEL STRUCTURE AND TASK MANAGEMENT

Kernel: Task Management

KERNEL STRUCTURE AND TASK MANAGEMENT

- **Task Management** is the main activity that a kernel does. It consists in the following steps:
 - Tasks **Scheduling**
 - Task **Dispatching**
- **Tasks Scheduling**
 - Performed by the **scheduler**: decides which task to select next for execution:
 - **Updates ready list** based on time, event occurrence, etc.
 - Runs the **scheduling algorithm** (algorithms already discussed in previous sections)
- **Tasks Dispatching**
 - **Task dispatcher**: Prepares and deploys for execution the task selected by the scheduler
 - Performs the so-called **context switching**



KERNEL STRUCTURE AND TASK MANAGEMENT

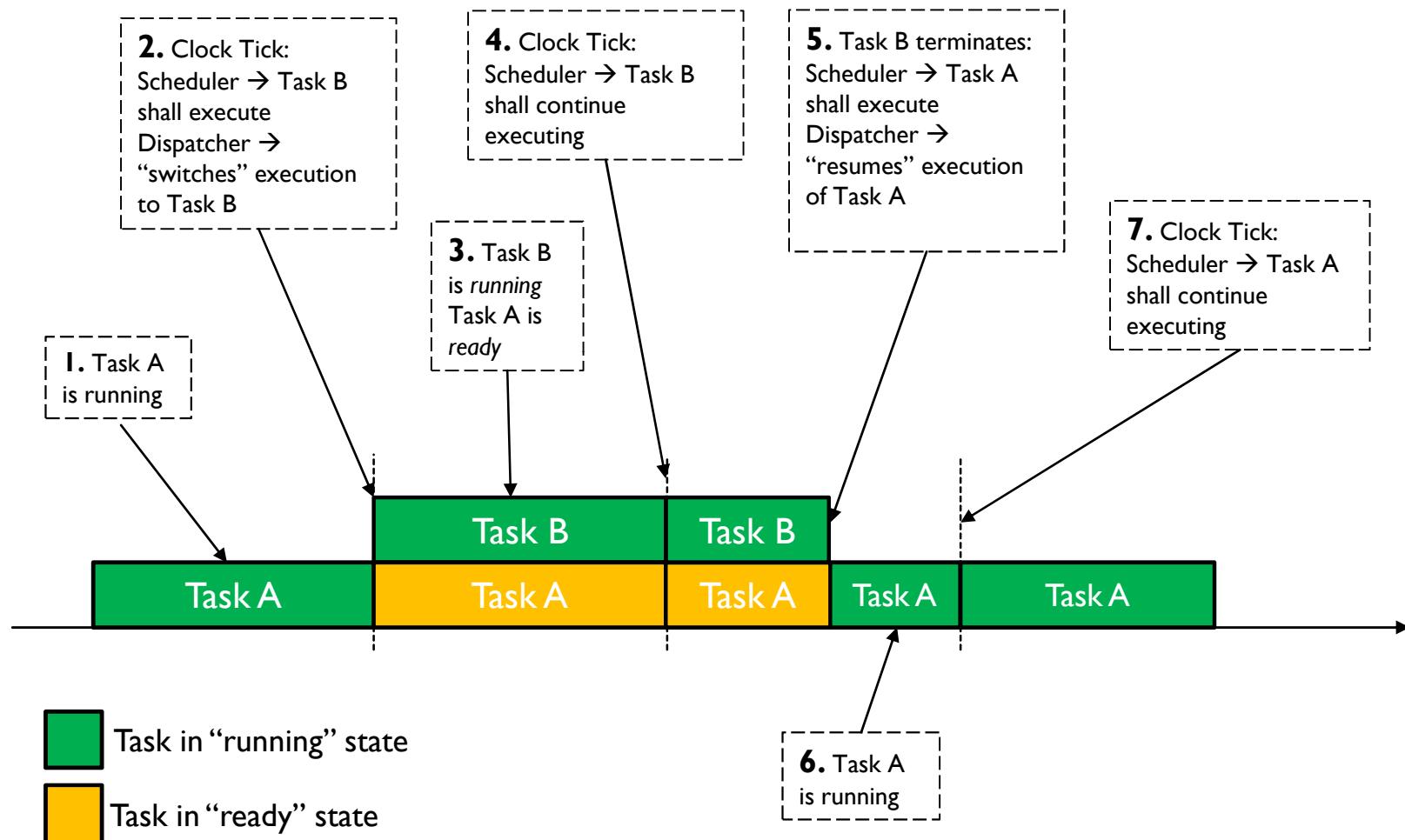
Context Switching



KERNEL STRUCTURE AND TASK MANAGEMENT

- Consider the scenario in following slide...

KERNEL STRUCTURE AND TASK MANAGEMENT



KERNEL STRUCTURE AND TASK MANAGEMENT

- What does it happen at 2. ?
 - RTOS scheduler decides that Task B is to be executed:
 - Task A is set to “ready” state
 - Task B is set to “running” state
 - RTOS **dispatcher** switches execution from Task A to Task B this operation is called **context switching**.

KERNEL STRUCTURE AND TASK MANAGEMENT

- What is the **context** of a Task?
 - Is the set of **all** data used by a task which needs to be backed-up upon a context switching.
- Which **data** is included in the task **context**?
 - A. Current state of the processor
 - PC, SP, Status Register, Index Registers, General Purpose Registers, ...
 - B. Automatic data (C terminology)
 - Data located in the stack. E.g., local variables, function's parameters, ...

KERNEL STRUCTURE AND TASK MANAGEMENT

- Such data is prone to unintended modifications if a different “task” starts suddenly executing
 - Remember that “tasks” can be seen as normal functions.

KERNEL STRUCTURE AND TASK MANAGEMENT

A. Current State of Processor. During execution of any program:

- **PC** (program counter) is continuously modified to point to the current instruction being executed by the CPU
- **SP** (stack pointer) is continuously modified when allocating local data (function's parameters, local variables, etc.)
- **Status Register** is continuously modified when performing logical/arithmetical operations, etc. Typical flags of a status register are:
 - zero flag (Z)
 - carry flag (C)
 - sign flag (S) or negative flag (N)
 - overflow flag, etc..
- **Index registers** modified when branching, handling arrays, etc..
- **General purpose registers**, continuously modified to hold the operand data for the processor's instructions being executed.

KERNEL STRUCTURE AND TASK MANAGEMENT

B. Local Data: stored in the stack

- Stack is a RAM area reserved for “automatic” variables (C terminology) → local data
- Stack space is highly re-used upon function calls
- Remember again → tasks can be seen as functions!
- RAM variables are not included in the context because:
 - RAM variables have their own fixed addresses
 - Non-static RAM variables (global variables) are intended for “sharing”. It is expected that any task can modify them at any time (with cautions of course).
 - Each task refers to the same data and not to an internal instance of it.
 - Static RAM variables (static local variables) are “private” to each task and hence no other task should be able to modify them.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Normally, the **processor state** data is actually pushed/pulled into/from the **stack** by means of special assembly instructions
- E.g., in a Freescale HCS12 MCU:

Stack Pointer Instructions		
CPS	Compare SP to memory	$(SP) - (M : M + 1)$
DES	Decrement SP	$(SP) - 1 \Rightarrow SP$
INS	Increment SP	$(SP) + 1 \Rightarrow SP$
LDS	Load SP	$(M : M + 1) \Rightarrow SP$
LEAS	Load effective address into SP	Effective address $\Rightarrow SP$
STS	Store SP	$(SP) \Rightarrow M : M + 1$
TSX	Transfer SP to X	$(SP) \Rightarrow X$
TSY	Transfer SP to Y	$(SP) \Rightarrow Y$
TXS	Transfer X to SP	$(X) \Rightarrow SP$
TYS	Transfer Y to SP	$(Y) \Rightarrow SP$

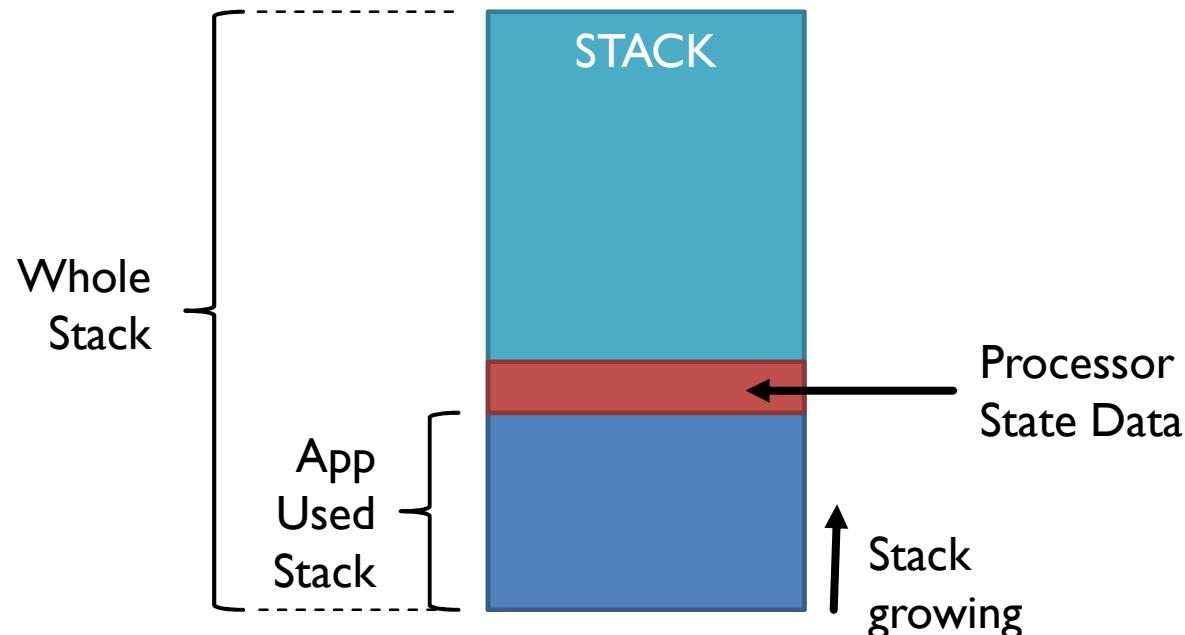
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... in a Freescale HCS12 MCU ...

Stack Operation Instructions		
PSHA	Push A	$(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$
PSHB	Push B	$(SP) - 1 \Rightarrow SP; (B) \Rightarrow M_{(SP)}$
PSHC	Push CCR	$(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$
PSHD	Push D	$(SP) - 2 \Rightarrow SP; (A : B) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PSHX	Push X	$(SP) - 2 \Rightarrow SP; (X) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PSHY	Push Y	$(SP) - 2 \Rightarrow SP; (Y) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PULA	Pull A	$(M_{(SP)}) \Rightarrow A; (SP) + 1 \Rightarrow SP$
PULB	Pull B	$(M_{(SP)}) \Rightarrow B; (SP) + 1 \Rightarrow SP$
PULC	Pull CCR	$(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$
PULD	Pull D	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B; (SP) + 2 \Rightarrow SP$
PULX	Pull X	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X; (SP) + 2 \Rightarrow SP$
PULY	Pull Y	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y; (SP) + 2 \Rightarrow SP$

KERNEL STRUCTURE AND TASK MANAGEMENT

- This means that an RTOS normally will “backup up” the **processor state into the stack** first and then “backup up” the whole stack.
- After saving the processor state data into the stack it will look like:



KERNEL STRUCTURE AND TASK MANAGEMENT

- Where is the **context** of each task “backed up”??
 - In the **Task Control Block (TCB)**
 - TCB also known as the Task Descriptor (TD)



KERNEL STRUCTURE AND TASK MANAGEMENT

Task Control Block (TCB)



KERNEL STRUCTURE AND TASK MANAGEMENT

- The **Task Control Block** is a data structure that contains all information needed to handle the task (by the Task Manager).
- The TCB normally contains:
 - A. **Task Control** Data:
 - Current **State** of the task (ready, running, blocked, ...).
 - Pointer to the task **code** (pointer to the function associated to the task)
 - Task identifier (name or unique number assigned to the task, ...).
 - Task Priority (if fixed priority scheduling is used).
 - Other RTOS implementation-specific required information (linked lists indexes, task queueing, etc.).
 - B. **Task Context** Data
 - Described Previously
- There shall be one TCB for each task of the system

KERNEL STRUCTURE AND TASK MANAGEMENT

■ Simplistic example of a **Task TCB**

I. Defined task **states**

```
/* Definition of task states */
typedef enum{
    OS_TASK_STATE_READY,
    OS_TASK_STATE_RUNNING,
    OS_TASK_STATE_BLOCKED
}OS_TASK_STATE;
```

KERNEL STRUCTURE AND TASK MANAGEMENT

- ...Simplistic example of a **Task TCB**...

2. Define data structure for the **TCB**...

KERNEL STRUCTURE AND TASK MANAGEMENT

- ...Simplistic example of a **Task TCB**...

```
#define OS_TASK_ID_MAX_CHARS 10/* Maximum allowed characters for the task ID */
#define OS_CPU_STACK_SIZE 200 /* CPU Stack Size */
typedef void (*OS_TASK_CODE_PTR)(void); /* Assuming tasks receive no parameters */

typedef struct OS_TASK_TAG_TCB{
    /* ---- Pointers for task queueing ---- */
    OS_TASK_TAG_TCB * p_next_tcb; /* Pointer to the next entry */
    OS_TASK_TAG_TCB * p_previous_tcb; /* Pointer to the previous entry */

    /* ---- Task Control Data ---- */
    char task_ID[OS_TASK_ID_MAX_CHARS]; /* String identifying the Task */
    OS_TASK_STATE task_state; /* Current state of the task */
    OS_TASK_CODE_PTR task_code_ptr; /* pointer to the task code */
    char task_priority; /* Task priority */
    OS_TASK_WAIT_QUEUE task_timing_queue; /* Task timing queue linking list */
    /* Place here any other queueing variables */

    /* ---- Context Data ---- */
    int stack_pointer; /* Sometimes stack-pointer is kept separated outside the stack */
    int program_counter; /* Sometimes program counter is kept separated outside the stack */
    /* Task stack can be same or smaller than CPU stack. In this example it is same */
    char task_stack[OS_CPU_STACK_SIZE];
}OS_TASK_TCB;
```



KERNEL STRUCTURE AND TASK MANAGEMENT

- Simplistic example of a **Task definition**

- I. To create a task we need to create its TCB and define its code.

```
/* Create Task A TCB */
OS_TASK_TCB os_task_tcb_taskA;

/* Define Task A code function */
void TaskA( void ){
    /* Task code ... */
}
```

KERNEL STRUCTURE AND TASK MANAGEMENT

- ... Simplistic example of a **Task definition** ...

2. We need to initialize the TCB of our Task A!

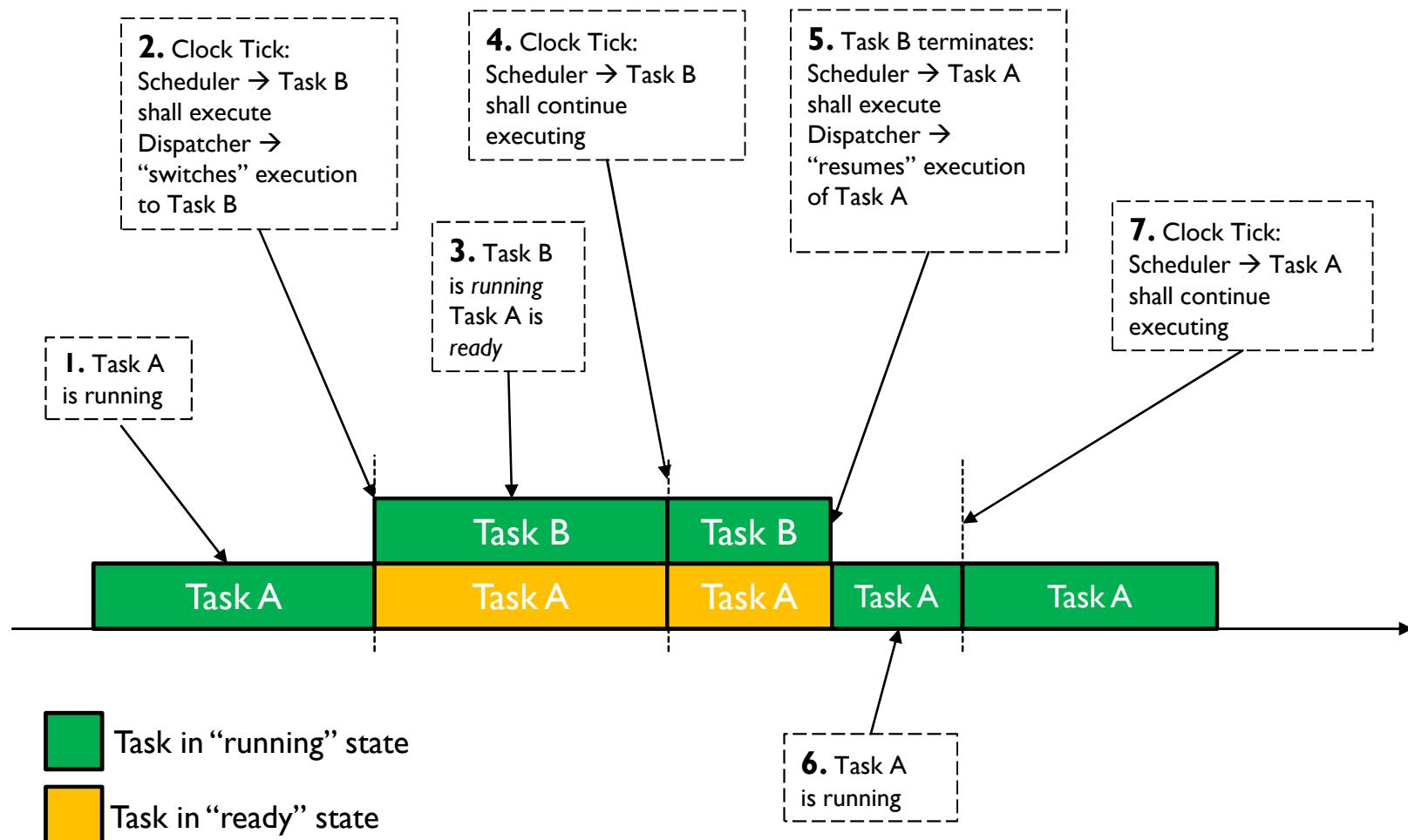
- The control information needs to be initialized
- The processor state may be initialized (may not, depends on the implementation)
- The Stack data (other than that of the processor state) doesn't need to be initialized. It will get meaningful data the very first time the Task A gets preempted by the scheduler.
- Linking task to the ready list and other lists needs to be done (omitted in this example).

```
/* Initialize Task A TCB control data */
os_task_tcb_taskA.task_ID = "Task A";
os_task_tcb_taskA.task_state = OS_TASK_STATE_READY;
os_task_tcb_taskA.task_code_ptr = TaskA;
os_task_tcb_taskA.task_priority = 8;
```

KERNEL STRUCTURE AND TASK MANAGEMENT

- ... back to our example: What does it happen at 2?

KERNEL STRUCTURE AND TASK MANAGEMENT



KERNEL STRUCTURE AND TASK MANAGEMENT

- **RTOS Dispatcher:**

- A. Context of currently running task (Task A) is copied (backed-up) into TCB of Task A.
 - The backed-up PC shall point to the last-plus-one instruction executed of Task A!!
- B. Context of new task to execute (Task B) is restored, i.e., copied from the TCB of Task B to the CPU
 - The last thing to do is to “restore” the program counter (PC) since once this gets written, the CPU will start executing the code of Task B.

KERNEL STRUCTURE AND TASK MANAGEMENT

- The **processor state** data and the **stack** data can be seen conceptually as independent items
 - Some times only one item and not the other is required to be handled (performance optimization)
- The context items shall be backed-up/restored at the appropriate time instants
- Special care is needed for the **processor state** data
 - E.g., How does SP behaves? Does it point to the last, current or to the next modified data?

KERNEL STRUCTURE AND TASK MANAGEMENT

- Recall that the “Clock Tick” is implemented using an MCU **interrupt**.
- The interrupt mechanism of the CPU does part of the job for us!
 - Upon a normal interrupt, before calling the ISR, the CPU normally copies the processor state into the stack!!
 - This is part of what we require for the action A) of the task dispatcher.
- Example: Following table shows what happens upon an interrupt for the Freescale HCS12 CPU.

Table 7-2. Stacking Order on Entry to Interrupts

Memory Location	CPU Registers
SP + 7	RTN _H : RTN _L
SP + 5	Y _H : Y _L
SP + 3	X _H : X _L
SP + 1	B : A
SP	CCR

KERNEL STRUCTURE AND TASK MANAGEMENT

- We still have things to do to complete action A) of the Task dispatcher:
 - Copy the whole stack to the TCB
 - For performance → do not copy part of the stack which has not been used (this information can be deducted from the stored SP)...
- What about action B) of Task dispatcher, i.e., context **restoring**?
 - First we copy the task local stack from TCB to the CPU stack
 - To restore the CPU processor state we can use the “return from interrupt” assembly instruction
 - E.g., This instruction is called RTI in Freescale HCS12 MCUs:

Table 5-22. Interrupt Instructions

Mnemonic	Function	Operation
RTI	Return from interrupt	$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$

KERNEL STRUCTURE AND TASK MANAGEMENT

- Real implementations do more actions, specially for proper handling of the SP and the PC.
- Considering that the scheduler and task dispatcher also modify the processor state! (They have “code” as any other function). We need to consider this situation as well!

KERNEL STRUCTURE AND TASK MANAGEMENT

- **Task dispatcher** Summary:

- A. Context Saving

- i. Save processor state (done by the ISR mechanism)
 - ii. Save stack into TCB (memcpy)

- B. Context Restoring

- i. Restore stack from TCB (memcpy)
 - ii. Restore processor state (done by “return from interrupt” instruction)

KERNEL STRUCTURE AND TASK MANAGEMENT

- Task **Scheduling** Summary:
 - A. Update task ready list
 - B. Execute scheduling policy (scheduling algorithm)

KERNEL STRUCTURE AND TASK MANAGEMENT

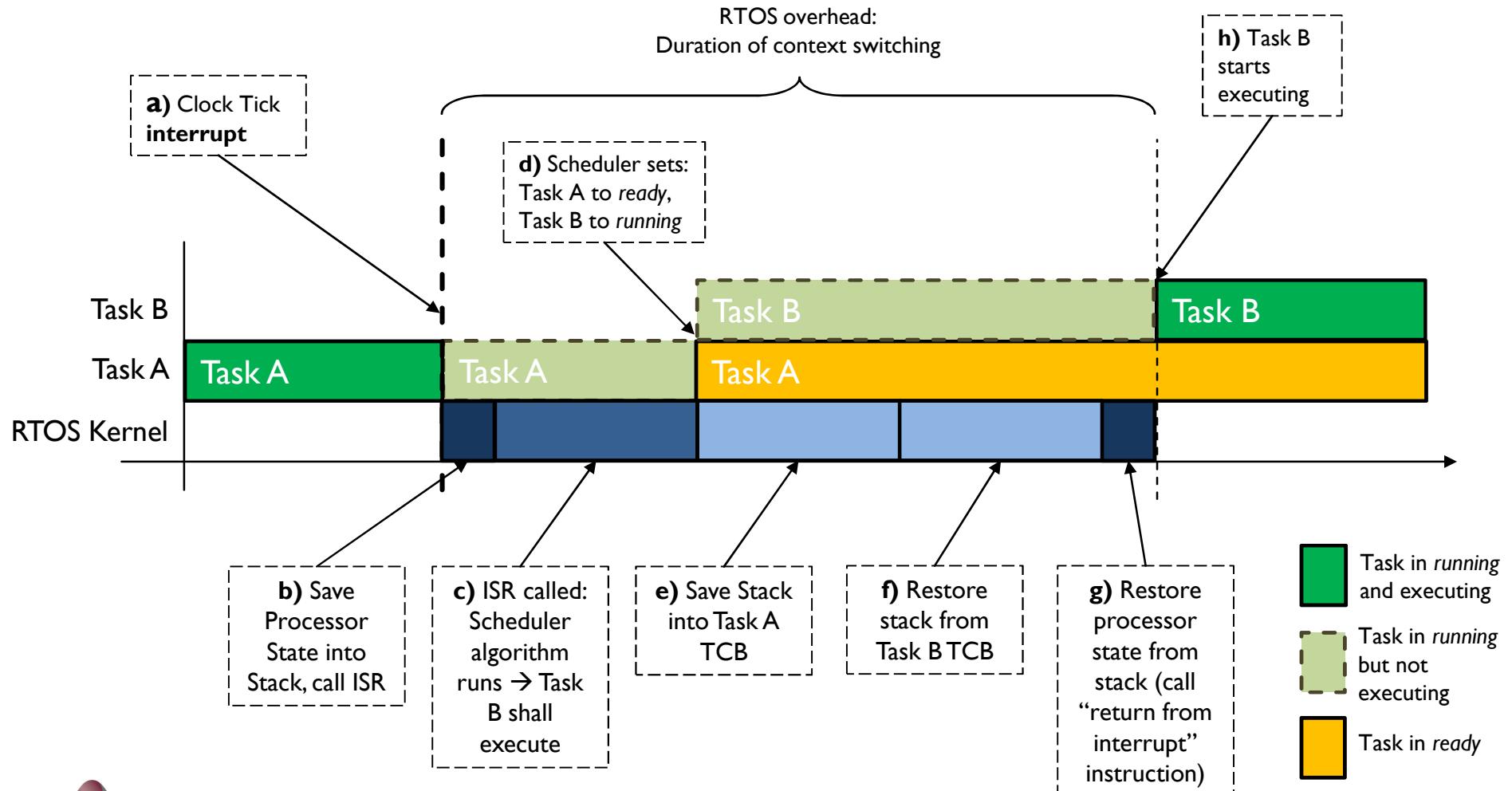
- Copying data items for context switching takes time!
 - Copy **processor state** data usually takes little time (only few bytes/instructions)
 - Copy **stack** data usually takes long time (can be hundreds of bytes)

KERNEL STRUCTURE AND TASK MANAGEMENT

- So, again but more in detail → What does it happen at 2?
- Now let's consider the RTOS kernel layer...

KERNEL STRUCTURE AND TASK MANAGEMENT

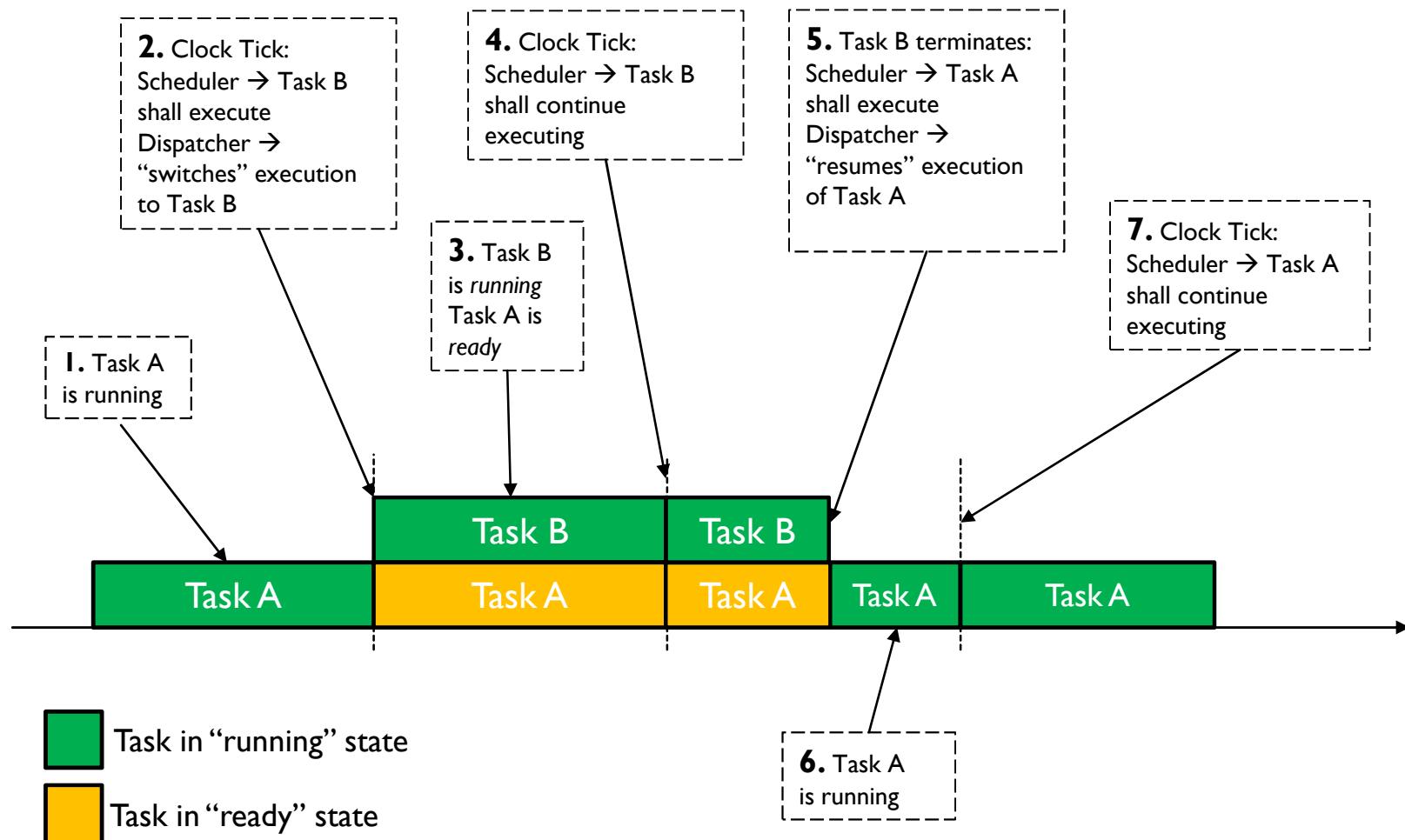
Figure: Context Switching



KERNEL STRUCTURE AND TASK MANAGEMENT

- Let's continue with the example...
- What does it happen at 4?

KERNEL STRUCTURE AND TASK MANAGEMENT



KERNEL STRUCTURE AND TASK MANAGEMENT

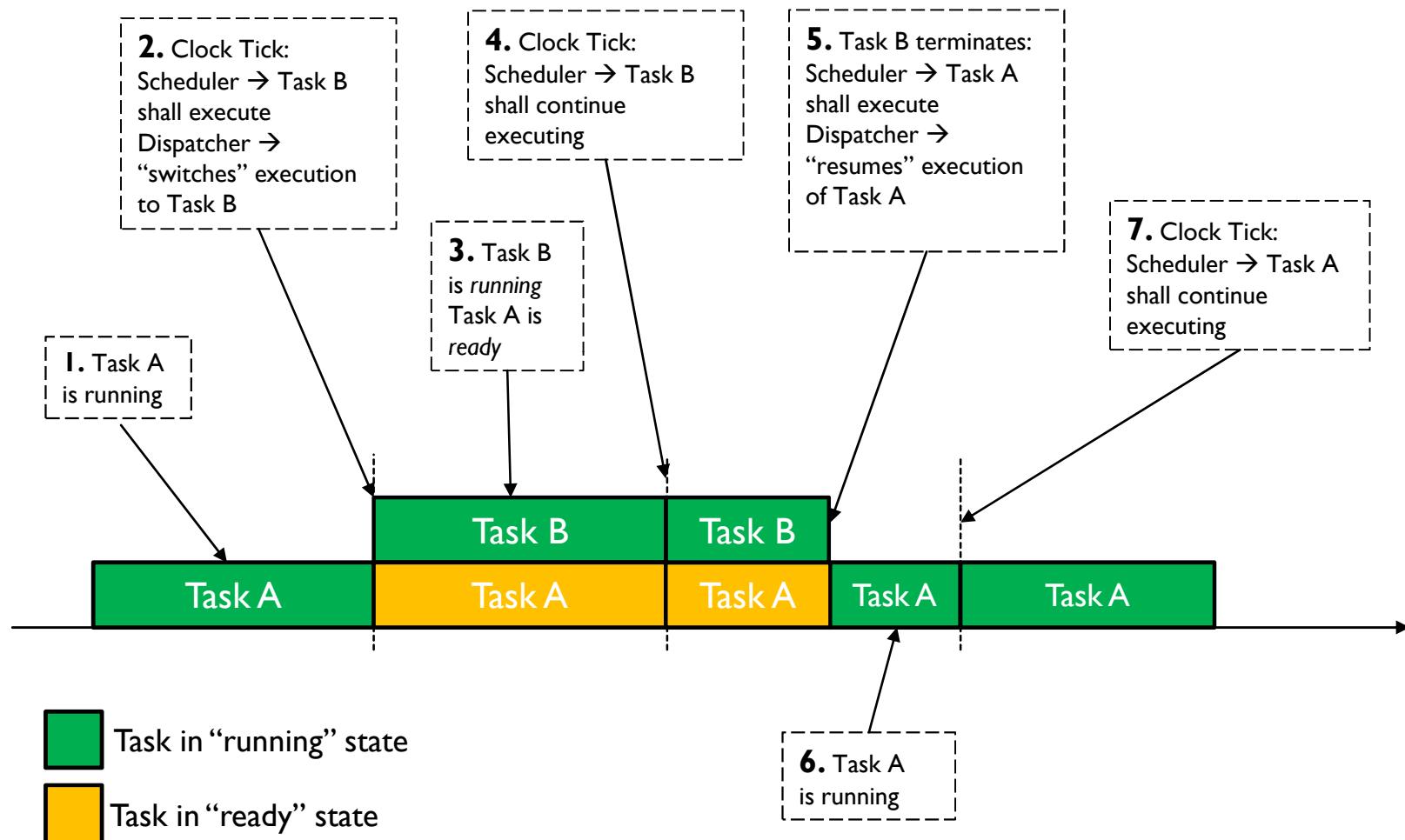
- At 4, a clock tick occurs however Task B continues running.
- There is no context switching
- Still some code of the kernel is executed
 - From “Figure: Context Switching”, only steps a), b), c) and g) will occur.
 - This takes “little” time → No stack copy/restore is performed



KERNEL STRUCTURE AND TASK MANAGEMENT

- What does it happen at 5?

KERNEL STRUCTURE AND TASK MANAGEMENT



KERNEL STRUCTURE AND TASK MANAGEMENT

- Task B terminates execution and calls the kernel.
 - This will be more efficient than waiting for the next clock tick to occur
→ Make productive use of the slack time (laxity)!
 - RTOS shall provide a “task terminate” API.
 - Task Terminate APIs are normally provided:
 - For periodic Tasks: when the RTOS (not the tasks) is the one implementing the periodicity, i.e., the task doesn't have an endless-loop and instead is normal function getting called cyclically by the RTOS.
 - For aperiodic Tasks
- The kernel decides that Task A shall resume execution.

KERNEL STRUCTURE AND TASK MANAGEMENT

- This situation at **5** is slightly different than the one at **2**..
 - There is no ISR being called (no clock tick).
 - We are indeed not interested in saving any context of Task B → It has terminated
 - We are interesting only in restoring the context of Task A
 - From “Figure: Context Switching”, only steps c), f) and g) will occur
 - “Medium” time: One stack copy is involved.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Step 7 is conceptually equal to step 4
- We have finished the analysis of our example 😊

KERNEL STRUCTURE AND TASK MANAGEMENT

- Now let's consider the following scenario:
- A Task X suspend **itself before termination** to wait for a certain event to occur
 - This is achieved by calling an RTOS API (a function) to suspend the task.

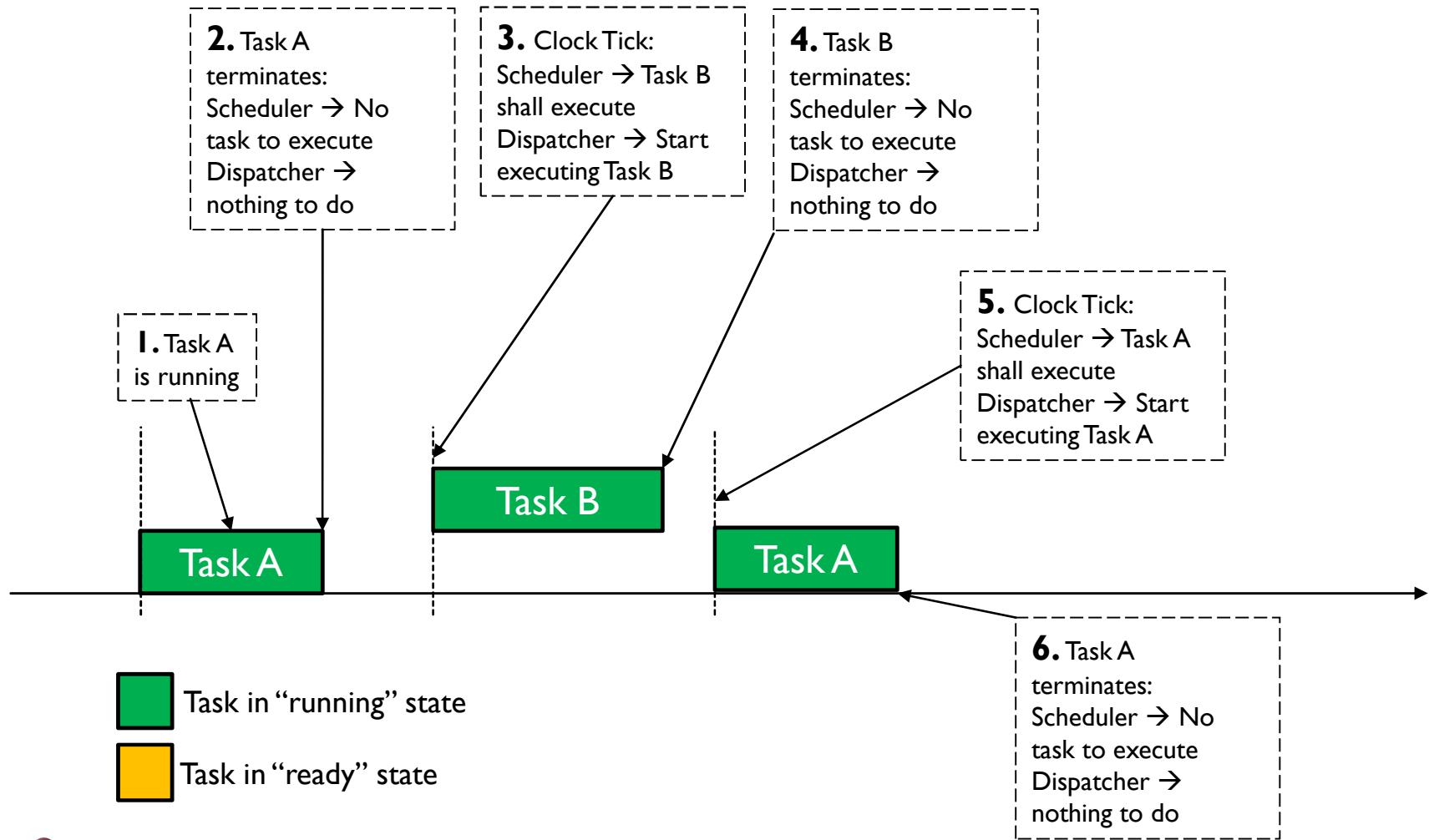
KERNEL STRUCTURE AND TASK MANAGEMENT

- We still need to save the Task X context (task has not terminate)
- There is no ISR involved. Rather we have a normal function call → the RTOS API for suspending the task
- We need to “mimic” the actions that happen when an interrupt occurs for saving the processor state!
 - We need to manually save SP, PC, GP register, index registers, etc. in the stack in the same order as if it was performed by the MCU’s interrupt controller.
 - We want later to make use of the “return from interrupt” instruction.
 - Special care needs to be taken to “ignore” the stack modifications made by the RTOS kernel itself.
- Some MCUs provide a “software interrupt” instruction which can be used for this purpose (E.g., SWI in Freescale HCS12)

KERNEL STRUCTURE AND TASK MANAGEMENT

- Lets now consider another scenario (just one more)...

KERNEL STRUCTURE AND TASK MANAGEMENT





KERNEL STRUCTURE AND TASK MANAGEMENT

- What does it happen at points 2, 4 and 6?

KERNEL STRUCTURE AND TASK MANAGEMENT

- Real-time applications run most of their functionality in periodic tasks.
- Most likely there will be time intervals where there is no periodic task ready to run (unless continuous CPU utilization is 100% which is definitively not good).
- In such time intervals, the CPU is “free” to execute:
 - ISRs
 - Aperiodic tasks
 - Background Tasks

KERNEL STRUCTURE AND TASK MANAGEMENT

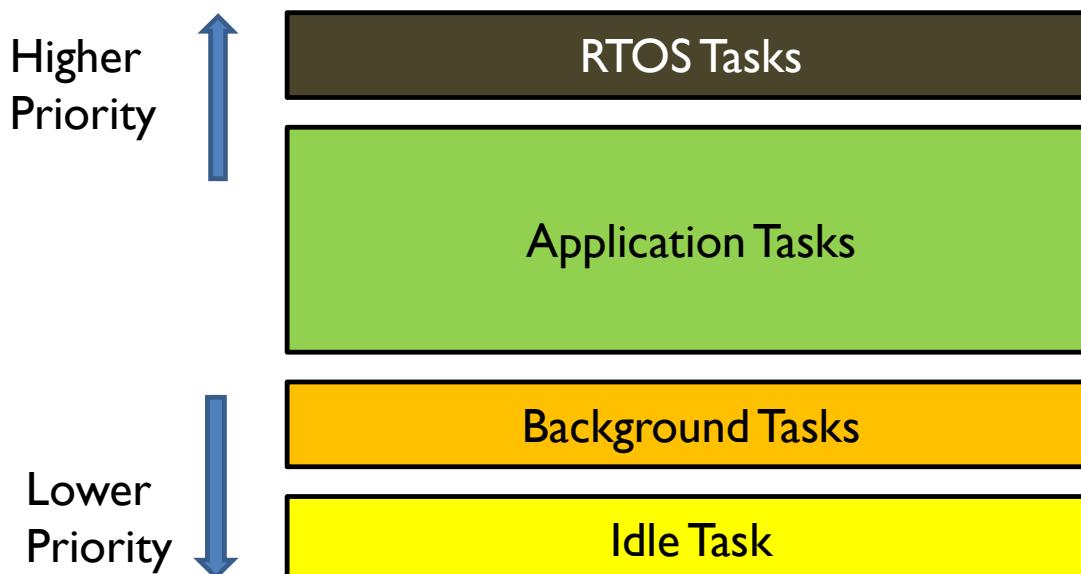
- If there are no task at all to be run then the only thing running will be the RTOS kernel itself → By means of the **idle task**
 - Kernel shall still be continuously checking if any task gets ready for execution (updating ready list):
 - Event which a task is waiting for occurs,
 - Time for which a task is waiting for elapses, etc...
 - Idle task keeps the system “alive”.
 - Idle task may contain code for CPU usage logging, etc.

KERNEL STRUCTURE AND TASK MANAGEMENT

RTOS Tasks

KERNEL STRUCTURE AND TASK MANAGEMENT

- Lot of RTOS services are indeed implemented using Tasks
 - Timers
 - Message passing
 - ...
- Those RTOS-Tasks are normally set with higher priority than any of the application tasks.



KERNEL STRUCTURE AND TASK MANAGEMENT

- So far we have mostly talked about **tasks** but..
- What about interrupts **interactions** with tasks??



KERNEL STRUCTURE AND TASK MANAGEMENT

Interrupts Handling

KERNEL STRUCTURE AND TASK MANAGEMENT

- Recall that real-time systems are tightly coupled with the **environment**
 - I/O devices are the interface with the “outside” world
 - The CPU needs to have the ability to interact with the I/O devices
 - Need a mechanism for an I/O device to gain CPU’s attention.
- Brake in normal execution flow is often required
 - Error handling
 - Kernel invocation
- **Interrupts** provide a way of doing this!

KERNEL STRUCTURE AND TASK MANAGEMENT

- What is an **interrupt**?
 - An event that causes a change in the normal flow of instruction execution.

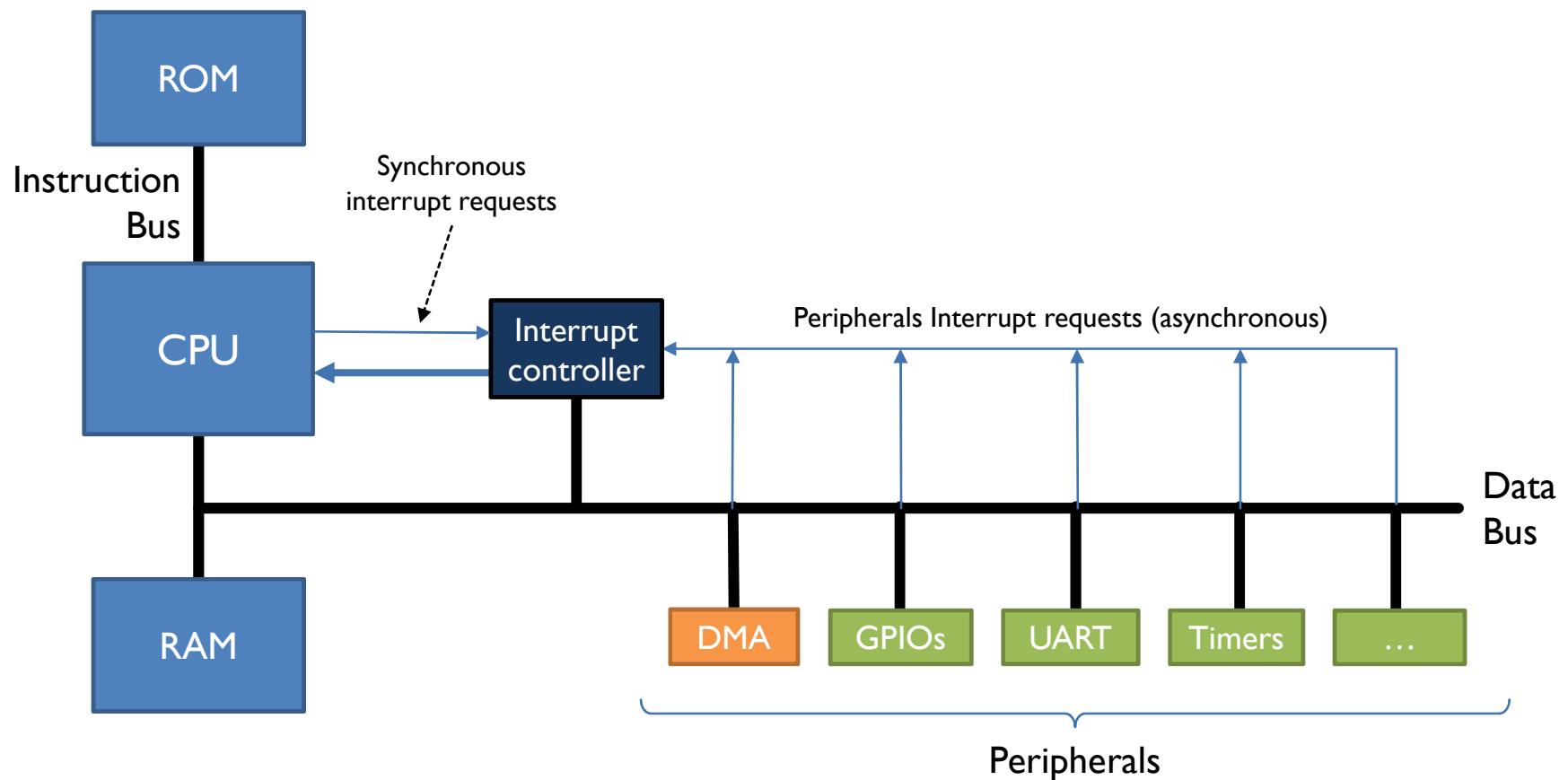
KERNEL STRUCTURE AND TASK MANAGEMENT

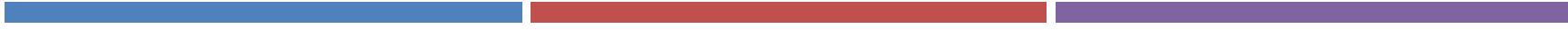
Types of interrupts

- **Synchronous (a.k.a. exceptions)**
 - **Processor-detected exceptions**
 - Commonly related to the instruction being executed
 - Fault Exceptions
 - E.g., divide by zero, wrong memory access, invalid opcode, etc.
 - **Programmed exceptions**
 - Requested by the code
 - E.g., by means of a “software interrupt” instruction.
- **Asynchronous (often simply referred as interrupts)**
 - From an external source (external to the CPU) such as an I/O device
 - Not related to the instruction being currently executed

KERNEL STRUCTURE AND TASK MANAGEMENT

- Simplified diagram of a MCU architecture showing the interrupt controller

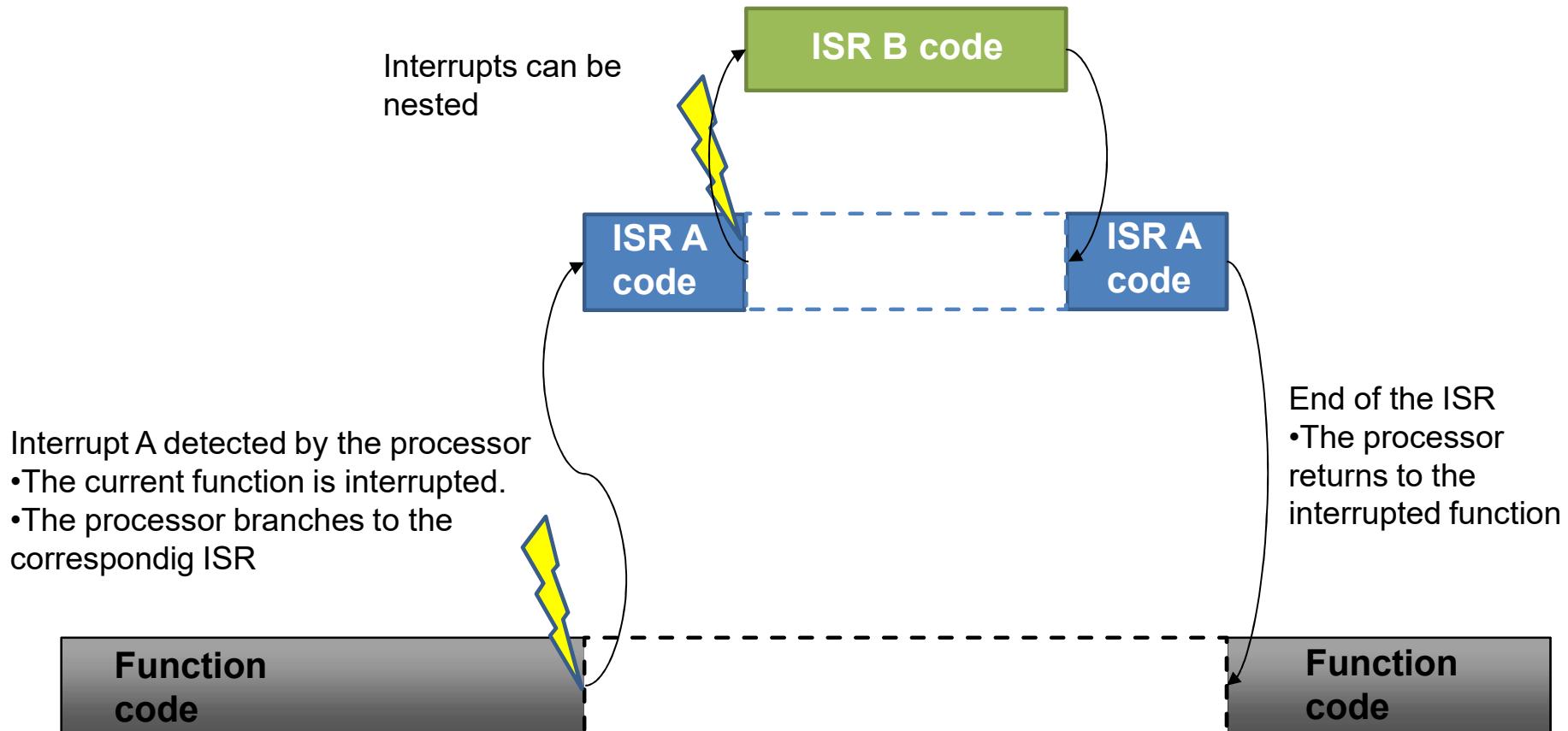




KERNEL STRUCTURE AND TASK MANAGEMENT

- How does an interrupt execution look like?

KERNEL STRUCTURE AND TASK MANAGEMENT



KERNEL STRUCTURE AND TASK MANAGEMENT

- More in detail....

KERNEL STRUCTURE AND TASK MANAGEMENT

Interrupt handling by the MCU

- When an interrupt event occurs and is presented to the processor following steps are normally performed by the MCU...
- **“Non-Kernel Aware” ISR steps:**
 - a) Current instruction being executed is finished
 - b) Disable further interrupts
 - c) Save processor state into the stack (return address -PC-, status register, gp registers, etc.)
 - d) Select and call the ISR from the interrupt vector table
 - e) **ISR gets executed** (interrupts may be re-enabled inside the ISR if nesting is allowed)
 - f) Re-enable interrupts
 - g) Restore saved processor state
 - h) Code being executed before the interrupt continues its execution

KERNEL STRUCTURE AND TASK MANAGEMENT

- Previous steps are for “**non-kernel aware**” interrupts.
 - Normally when executing the ISR the global interrupts are kept disabled.
- Later we will discussed how the “**kernel aware**” interrupts handling looks like.

KERNEL STRUCTURE AND TASK MANAGEMENT

- The **Interrupt Service Routine (ISR)** is a *callback* function associated to a certain event or events causing an interrupt and which purpose is to service such event.
- An **interrupt vector** is a memory location holding the starting address (pointer) of the ISR associated to a certain interrupt source.
- The **interrupt vector table** is an array of interrupt vectors, i.e., an array of pointers to ISRs.
 - There can be implemented by SW or by HW depending on the MCU architecture

KERNEL STRUCTURE AND TASK MANAGEMENT

- Different **MCU architectures** implement interrupt handling in different manners.
- **Single interrupt vector for all interrupt sources**
 - Some MCUs provide only a single interrupt vector for all interrupt sources.
 - It is responsibility of the **ISR** to look for the interrupt source and call the corresponding function for servicing the triggering event.
- **Multiple interrupt vectors for the different interrupt sources**
 - Most MCUs provide an interrupt vector for each interrupt source.
 - If an interrupt vector is associated for each *peripheral* then the ISR will need to find the particular “reason” of the interrupt.
 - E.g., If a single vector for UART0 is provided, then the ISR needs to find if the interrupt reason was “reception OK”, “transmission OK”, “Framing Error”, etc.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Interrupts can be **nested**
 - An interrupt event happens while another previous interrupt event is being serviced. This new event service will “preempt” the previous one.
- Multiple interrupts can be **prioritized**
 - If multiple interrupt events are “pending” at the same time, attend first the one with higher priority
 - If a “higher” priority interrupt event occurs while running another ISR, execute this new nested ISR.
- Most MCUs provide HW utilities for configuring interrupt priorities
 - What if they don't? Could we still have this functionality (nesting, priorities, etc.)?

KERNEL STRUCTURE AND TASK MANAGEMENT

- Some Interrupts can be **masked**
 - When masked, interrupts are prevented from occurring.
 - Events are still saved as “pending” normally by means of flags in the peripheral registers however, the CPU never “notices” those events. The “interrupt controller” doesn’t tell the CPU about such events.
 - Most of the asynchronous interrupt sources associated to peripherals can be masked.
- Some interrupts can **NOT** be masked
 - The so called **Non-Maskable Interrupts (NMI)**
 - NMI can be associated to GPIO pins (asynchronous interrupts)
 - NMI can be associated to fault exceptions, etc. (synchronous interrupts)

KERNEL STRUCTURE AND TASK MANAGEMENT

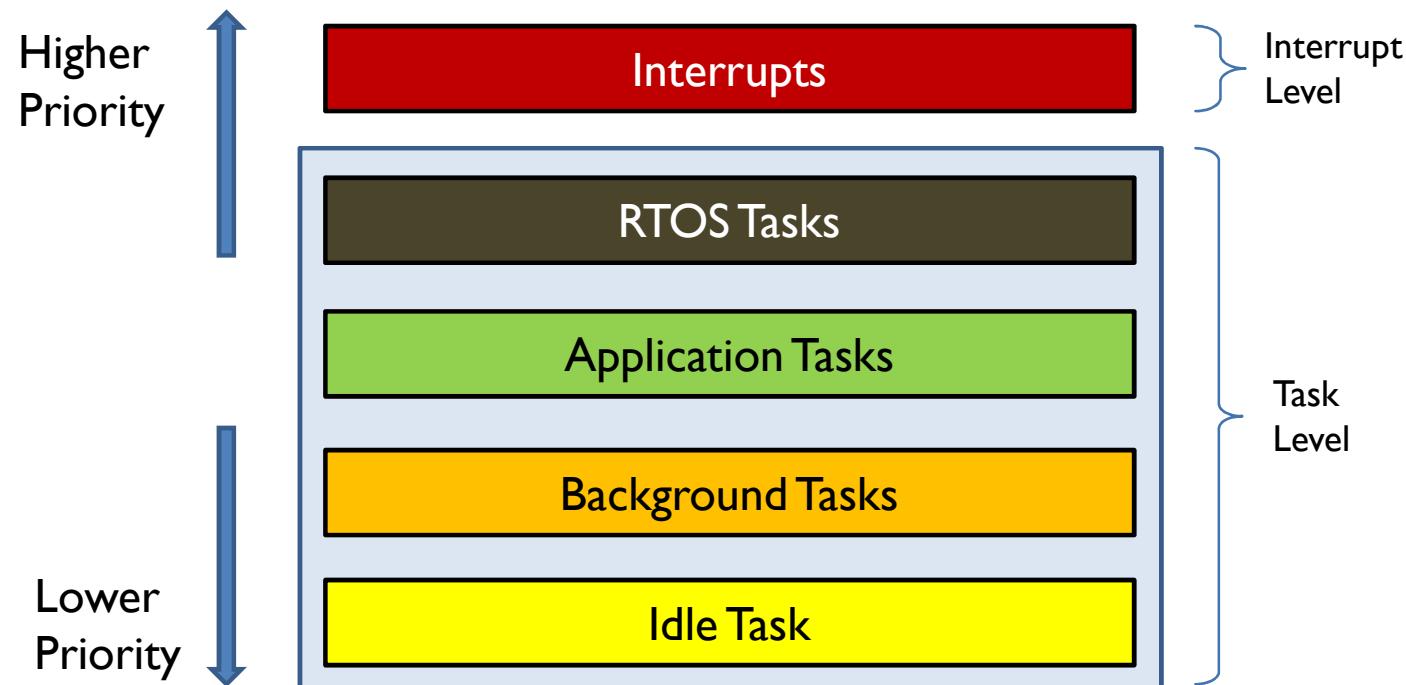
- The MCU's **interrupt controller**
 - Detects and masks the interrupt sources
 - Prioritize the active interrupt events
 - Selects the higher priority interrupt and *interrupts* the CPU
 - If HW has a single vector for all interrupt sources:
 - Provide a mean for the SW to “get” the interrupt source
 - E.g., By passing an interrupt request number (IRQ) which can be used as index to a SW-implemented interrupt vector.
 - If HW has multiple vectors for the different interrupt sources:
 - Normally HW automatically selects the specific vector from a HW-implemented interrupt vector table and give the associated ISR function address to the CPU

KERNEL STRUCTURE AND TASK MANAGEMENT

- General **ISRs** implementation philosophy
 - Do as **little** as possible inside the ISRs
 - ISRs normally run with global interrupts disabled (interrupt suppression time)
 - Long ISRs may brake the schedulability of periodic tasks.
 - Defer non-critical actions for **later**
 - That “later” can be attended at task level.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Interrupts have inherently higher priority than any task!
 - The HW “interrupt controller” is the one taking control of the CPU



KERNEL STRUCTURE AND TASK MANAGEMENT

- RTOS can help to improve the usage of interrupts in the system:
- Commonize the interrupt handling regardless of the MCU architecture
 - Offer prioritization in MCUs that do not provide this feature
 - Extends prioritization policies in MCUs with existing prioritization feature
- Improve ISRs interaction with Tasks
 - Provide means to “signal” or activate tasks within the ISR
 - Scheduling may happen immediately after the ISR maybe giving execution to a higher-priority task activated within the ISR instead of returning to the interrupted one.
 - Context switching optimization

KERNEL STRUCTURE AND TASK MANAGEMENT

- ...RTOS can help to improve the usage of interrupts in the system...
- Improve stack usage by creating a exclusive stack for ISRs (**ISRs stack**)
 - RTOS can provide a Stack area to be used only by ISRs separated from the one used for tasks
 - Task stacks will not need to give additional room for ISRs execution → Task stack can be keep smaller (faster context switching)
- Provide ISR logging/debugging functions
 - Services for keeping track of ISRs execution times, etc.

KERNEL STRUCTURE AND TASK MANAGEMENT

- How does the RTOS provide all those advantages for ISRs handling?
 - By means of an interrupt “**prologue**” and “**epilogue**” surrounding the ISR execution.
- The execution of an interrupt handler when it is managed by the RTOS will then look as follows:
 1. Interrupt event occurs and is caught by the CPU
 2. The CPU jumps to the corresponding interrupt vector
 3. Interrupt “**prologue**” is executed
 4. Associated **ISR** is executed
 5. Interrupt “**epilogue**” is executed
 6. Execution is returned to the interrupted code.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Interrupts handled this way are known as “**kernel-aware**” interrupts.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Interrupt “**prologue**”
 - Is code written in assembly language that “complements” the actions performed by the MCU when starting to attend an interrupt.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Common “**prologue**” activities:

1. Disable further interrupts (done by MCU)
2. Save processor state into the stack (done by MCU)
3. Start some ISR logging/debugging features (optional).
4. Increment a “nesting counter” (nesting counter is initially 0)
5. If (nesting counter == 1) save current SP and then set it to point to the ISRs stack.
 - Nesting counter == 1 means that this interrupt is the first one being processed, i.e. is not a nested interrupt.
 - When nesting counter > 1 means that we are processing a nested ISR and we are already using the ISRs stack so no need to do anything here related to the SP.
6. If MCU doesn't provide interrupt priorities then code for prioritizing ISRs is executed here.
7. Clear interrupt flag
8. Re-Enable interrupts (optional, performed if nested interrupts are desired)

KERNEL STRUCTURE AND TASK MANAGEMENT

- After “**prologue**” is executed then the ISR is called.
 - ISR can run with interrupts enabled!
 - ISR can safely call certain RTOS services
- Once **ISR** finishes its execution then the interrupt “**epilogue**” gets called.
 - In this case, the ISR does not perform a “return from interrupt” but rather a normal “function return”.

KERNEL STRUCTURE AND TASK MANAGEMENT

- **Interrupt “epilogue”**
 - Is code written in assembly language that “complements” the actions performed after an ISR is executed and before returning to the interrupted function.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Common “epilogue” activities:
 1. Stop ISR logging/debugging features (optional).
 2. Decrement nesting counter
 3. If (nesting counter == 0) then switch back from the ISRs Stack to the Tasks Stack and restored the saved SP.
 4. Execute Kernel scheduler
 5. If a different task with higher priority than the original *interrupted* one is to be executed then switch execution to this new task (context switching).

Otherwise it means that the original *interrupted* task is the one to continue executing and then:

6. Re-enable interrupts
7. Return execution to the *interrupted* task (return from interrupt)

KERNEL STRUCTURE AND TASK MANAGEMENT

- Note that steps 6 and 7 of the epilogue are not executed if a context switching happens at step 5.
- Putting all together for “**kernel-aware**” interrupt handling....

KERNEL STRUCTURE AND TASK MANAGEMENT

■ Putting all together: “Kernel-aware” interrupt handling:

- Prologue**
 - 1. Disable further interrupts (done by MCU)
 - 2. Save processor state into the stack (done by MCU)
 - 3. Start some ISR logging/debugging features (optional).
 - 4. Increment a “nesting counter” (nesting counter is initially 0)
 - 5. If (nesting counter == 1) save current SP and then set it to point to the ISRs stack.
 - 6. If MCU doesn’t provide interrupt priorities then code for prioritizing ISRs is executed here.
 - 7. Clear interrupt flag
 - 8. Re-Enable interrupts (optional, performed if nested interrupts are desired)
 - 9. Execute **ISR**
- Epilogue**
 - 10. Stop ISR logging/debugging features (optional).
 - 11. Decrement nesting counter
 - 12. If (nesting counter == 0) then switch back from the “ISRs” Stack to the Tasks Stack and restored the saved SP.
 - 13. Execute Kernel scheduler
 - 14. If a different task with higher priority than the original *interrupted* one is to be executed then switch execution to this new task (context switching).
Otherwise it means that the original *interrupted* task is the one to continue executing and then:
 - 15. Re-enable interrupts
 - 16. Return execution to the *interrupted* task (return from interrupt)

KERNEL STRUCTURE AND TASK MANAGEMENT

- Additional steps can be performed by the “prologue” and “epilogue” as per the needs and implementation of the particular RTOS.
- In MCUs with a single interrupt vector for all interrupt sources:
 - A single prologue and epilogue is defined for all ISRs
 - The RTOS shall “query” the interrupt source before calling the corresponding ISR.
- In MCUs with multiple interrupt vectors for the different interrupt sources:
 - A “copy” of the epilogue and prologue is placed on each of the interrupt vectors.
 - Of course in between the epilogue and prologue, the corresponding ISR call shall be placed to service the specific interrupt vector.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Normally RTOS Scheduling is blocked while performing an ISR.
 - We do not want a context switching to occur at the middle of an ISR execution otherwise the ISR will get serious delays.
- RTOS normally give the user the option to configure a certain interrupt vector as a “**non-kernel aware**” or as a “**kernel-aware**” interrupt.
 - Non-kernel aware interrupts are often referred as “direct” interrupts.

KERNEL STRUCTURE AND TASK MANAGEMENT

- When to use “**Kernel Aware**” interrupts?
 - When the interrupt has a tight operation with a task.
 - When the interrupt may activate a task (e.g., trigger an aperiodic task)
 - When the interrupt needs to use some RTOS APIs:
 - Posting semaphores, posting events
 - Message passing to a task, etc.
 - When easy debugging of such interrupt is desired:
 - Measuring its execution time, etc.

KERNEL STRUCTURE AND TASK MANAGEMENT

- When to use “**Non-Kernel Aware**” interrupts?
 - When speed of the ISR execution is critical
 - Kernel aware interrupts add overhead for the “prologue” and “epilogue”
 - When the ISR does not directly interact with a task
 - No task activation is required within the ISR
 - No RTOS API usage is required within the ISR: semaphores, events, messages, etc.

KERNEL STRUCTURE AND TASK MANAGEMENT

- RTOS normally provide different **types** of “**kernel aware**” interrupts varying the things that are performed by the “prologue” and “epilogue”.
 - E.g., A simple “kernel-aware” category just with debugging capabilities and another category with full nesting, priorities, scheduling capabilities.
 - E.g., OSEK Category I and Category II interrupts, etc.
- RTOS normally provide additional interrupt handlers:
 - Default handlers to manage “**unused**” interrupts.
 - Handlers to manage **fault exception** events, etc..



REAL-TIME OPERATING SYSTEMS

MAESTRÍA EN SISTEMAS INTELIGENTES MULTIMEDIA

INSTRUCTOR: CARLOS ENRIQUE DIAZ GUERRERO



AUTHOR: CARLOS ENRIQUE DIAZ GUERRERO

8/3/2018

1

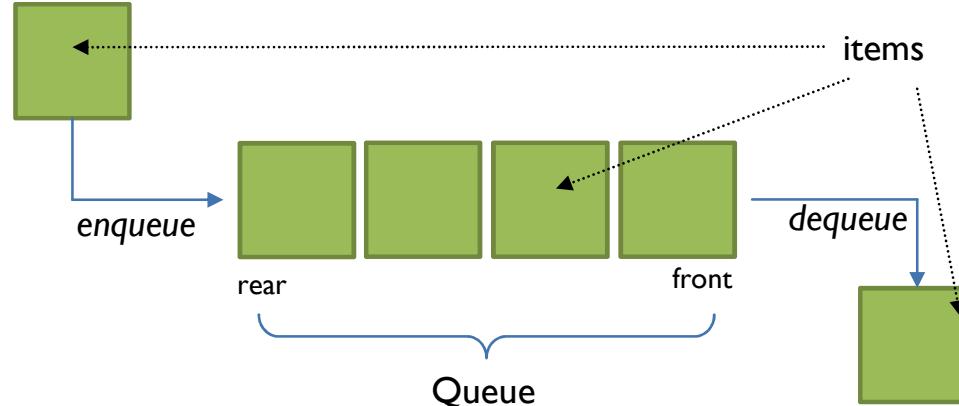


KERNEL STRUCTURE AND TASK MANAGEMENT

Queuing Model

KERNEL STRUCTURE AND TASK MANAGEMENT

- Most important RTOS features regarding Task Management can be implemented using **queues**
- In general a **queue** is a data structure where the data items are kept adjacently and are handled in FIFO order.
- Operations over a queue are:
 - Enqueue: **add** an item at the **rear** of the queue
 - Dequeue: **remove** an item from the **front** of the queue

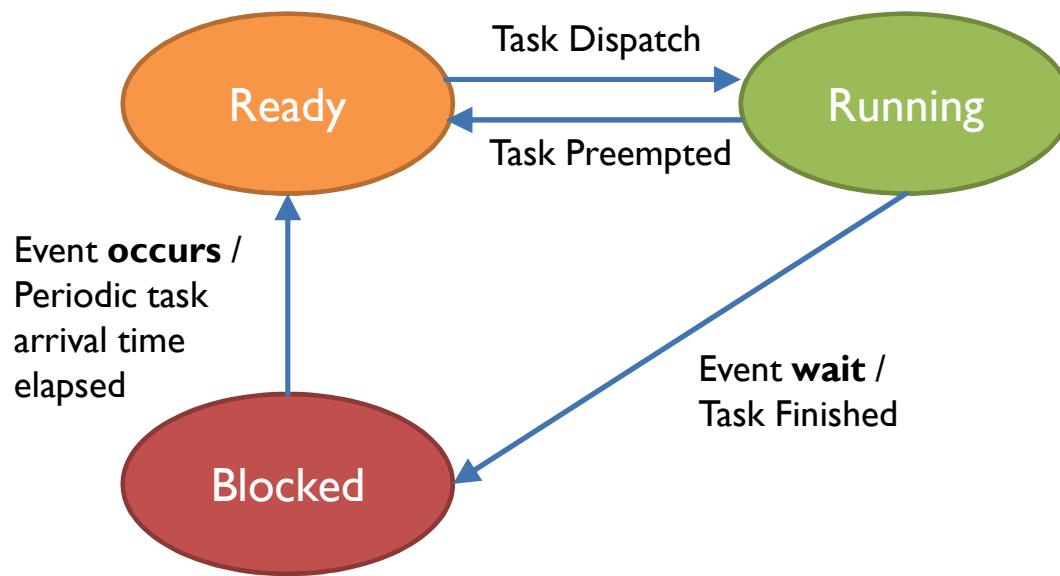


KERNEL STRUCTURE AND TASK MANAGEMENT

- Examples of queues within an RTOS
 - Ready list → Ready queues
 - Event queues
 - Message queues
 - ...

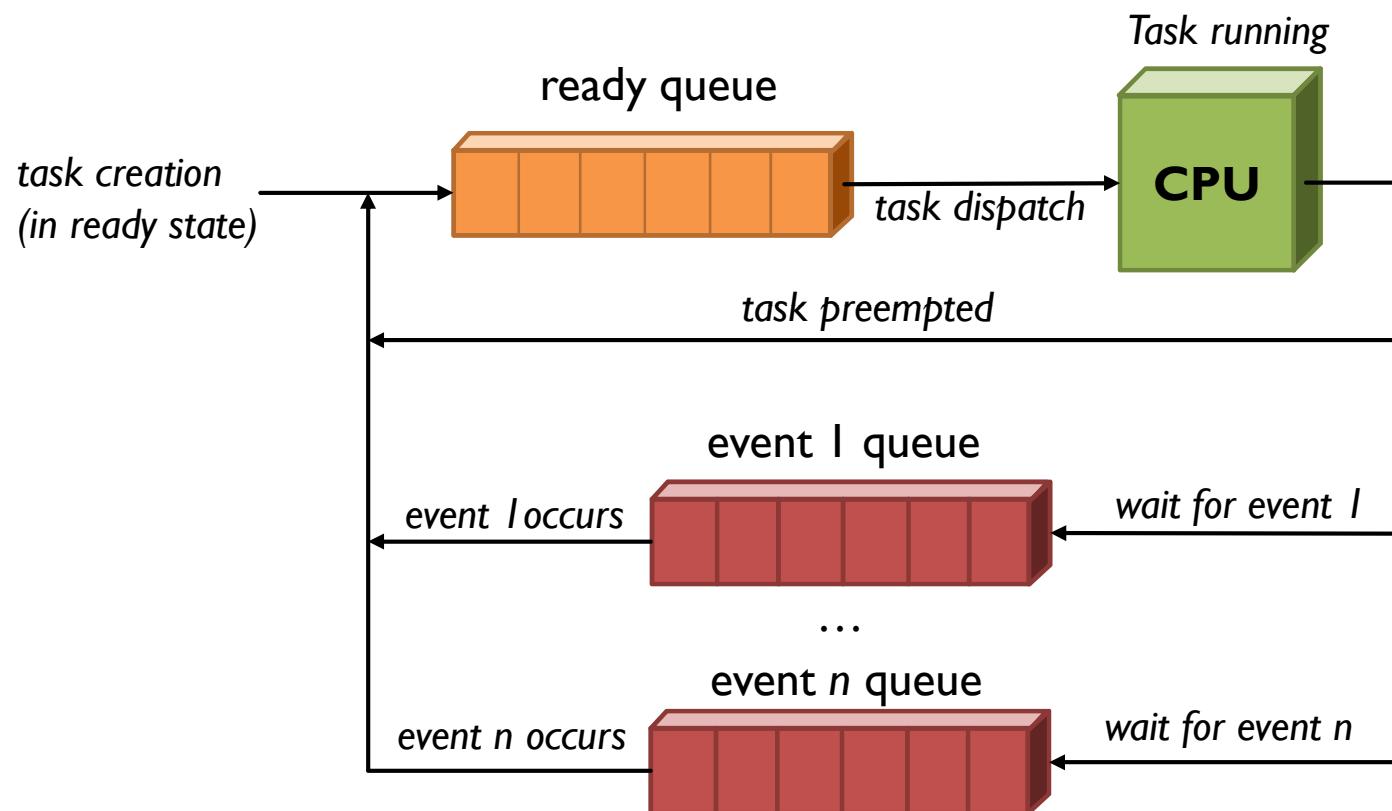
KERNEL STRUCTURE AND TASK MANAGEMENT

- Let's recall the 3 states Task model:



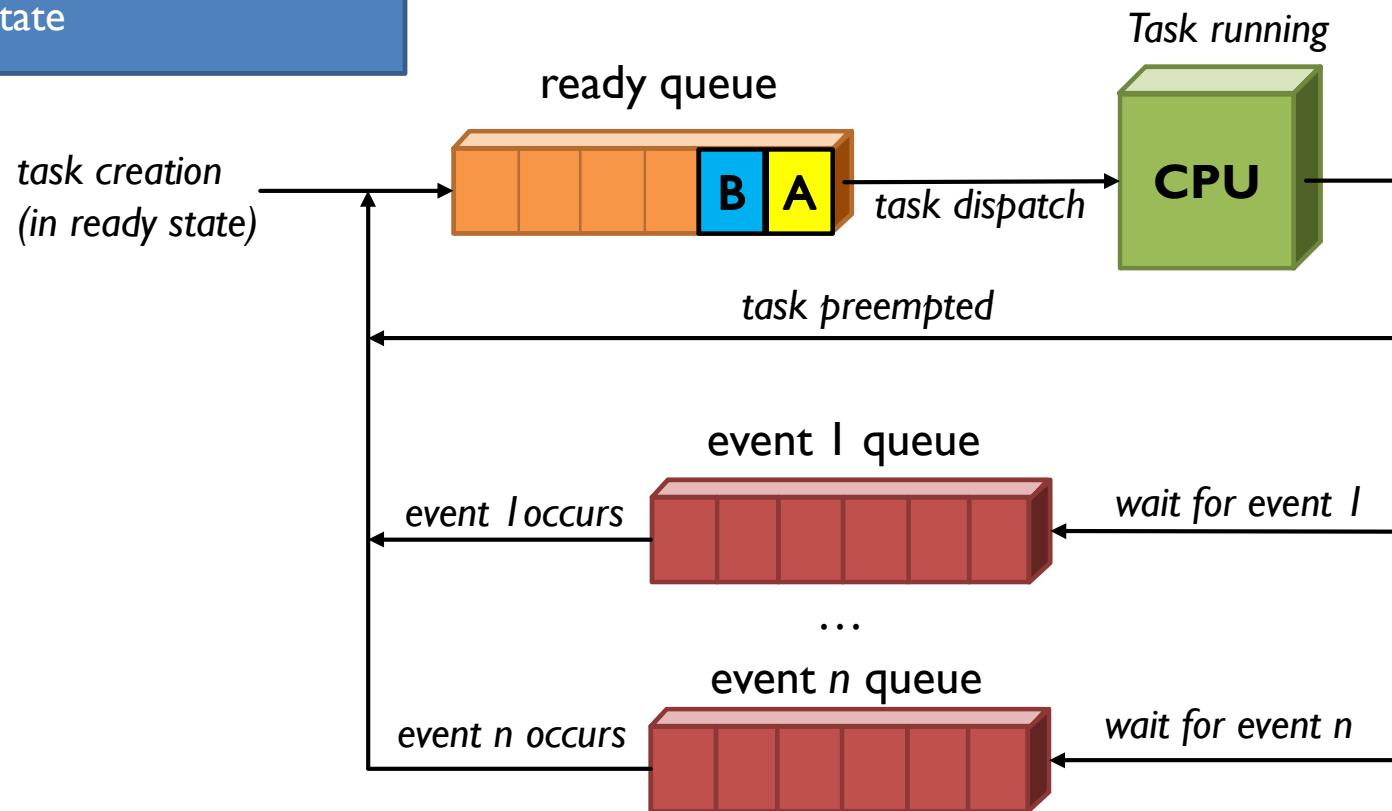
KERNEL STRUCTURE AND TASK MANAGEMENT

- Using queuing model, task state transitions will look as follows:

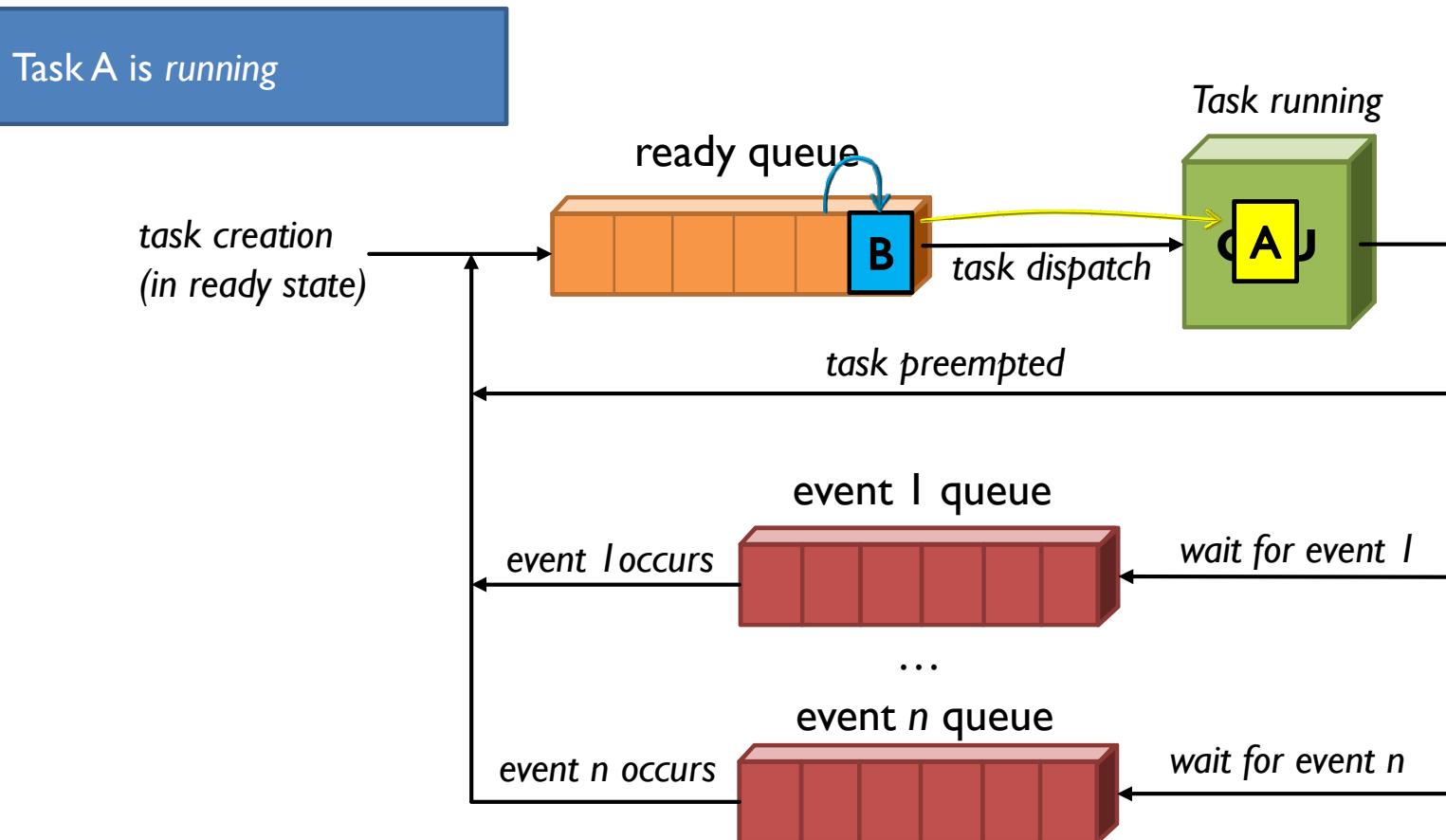


KERNEL STRUCTURE AND TASK MANAGEMENT

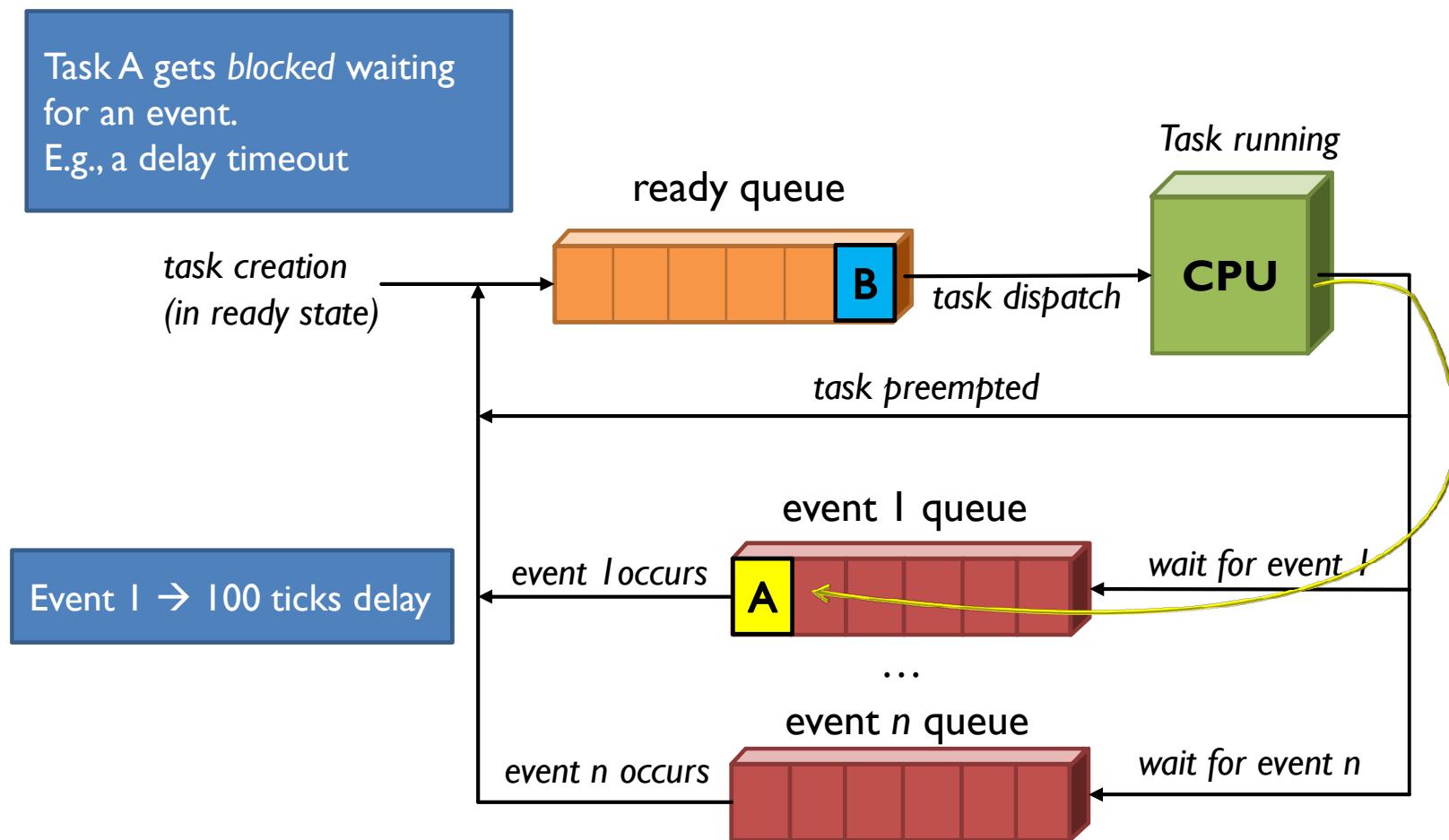
Task A and B are created in ready state



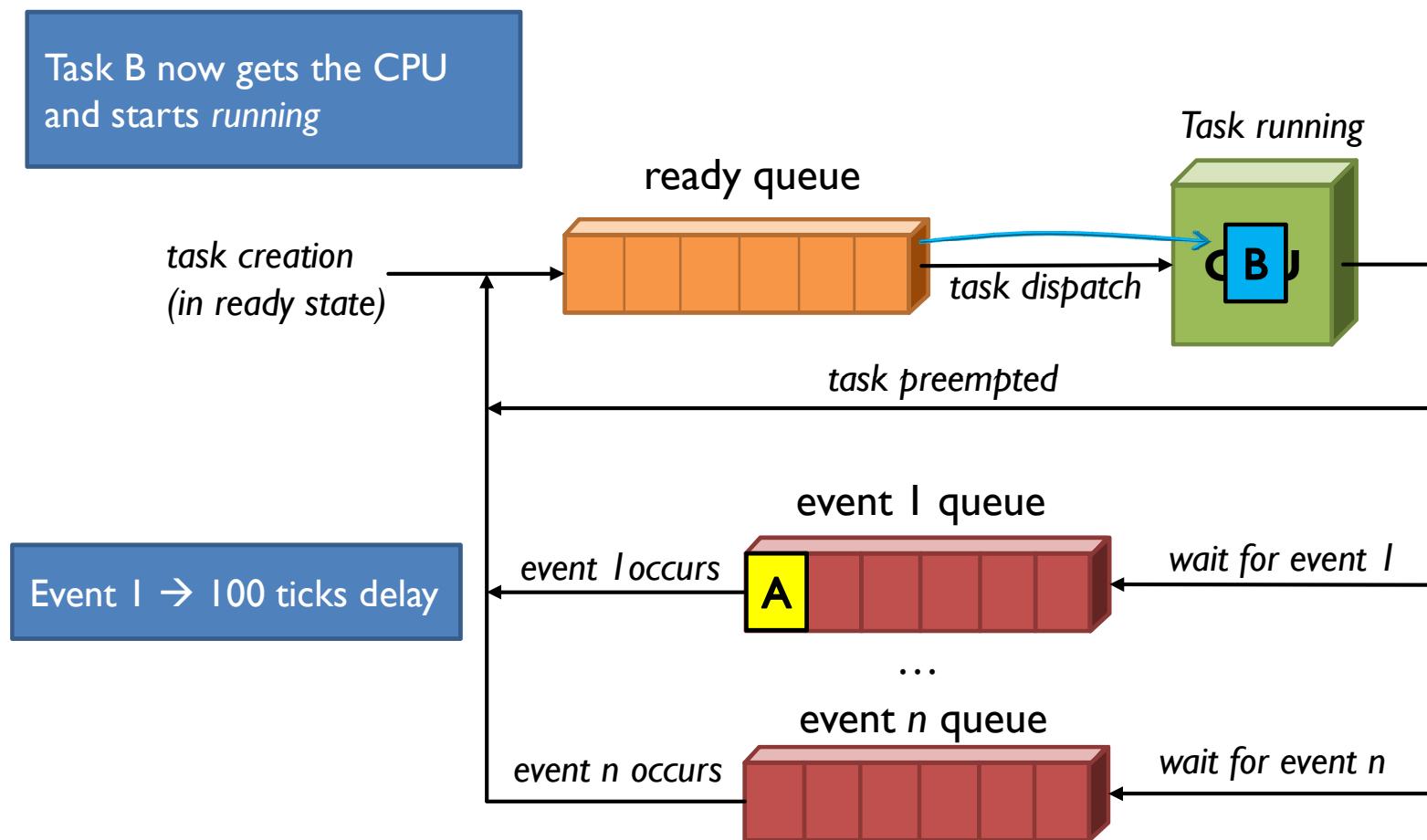
KERNEL STRUCTURE AND TASK MANAGEMENT



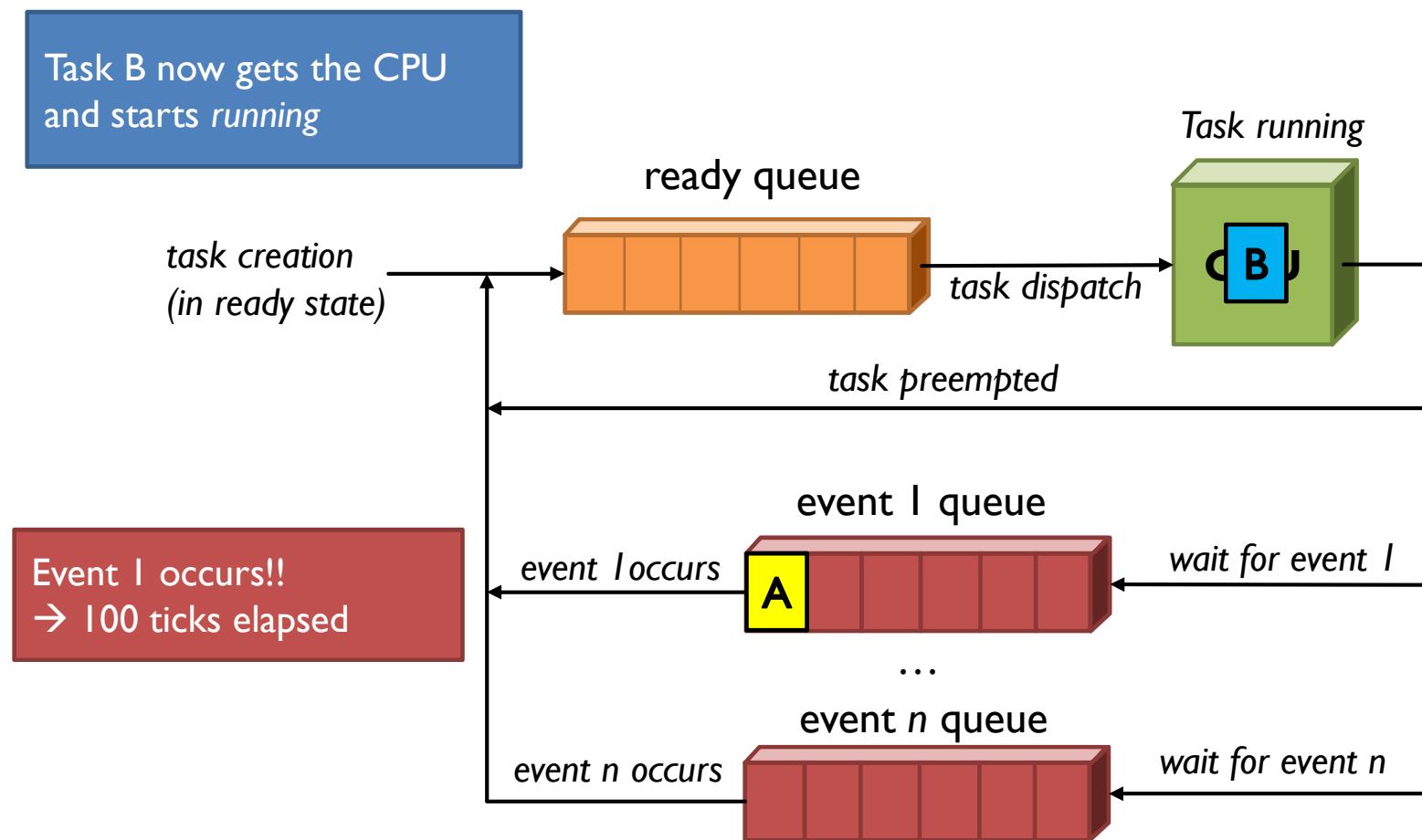
KERNEL STRUCTURE AND TASK MANAGEMENT



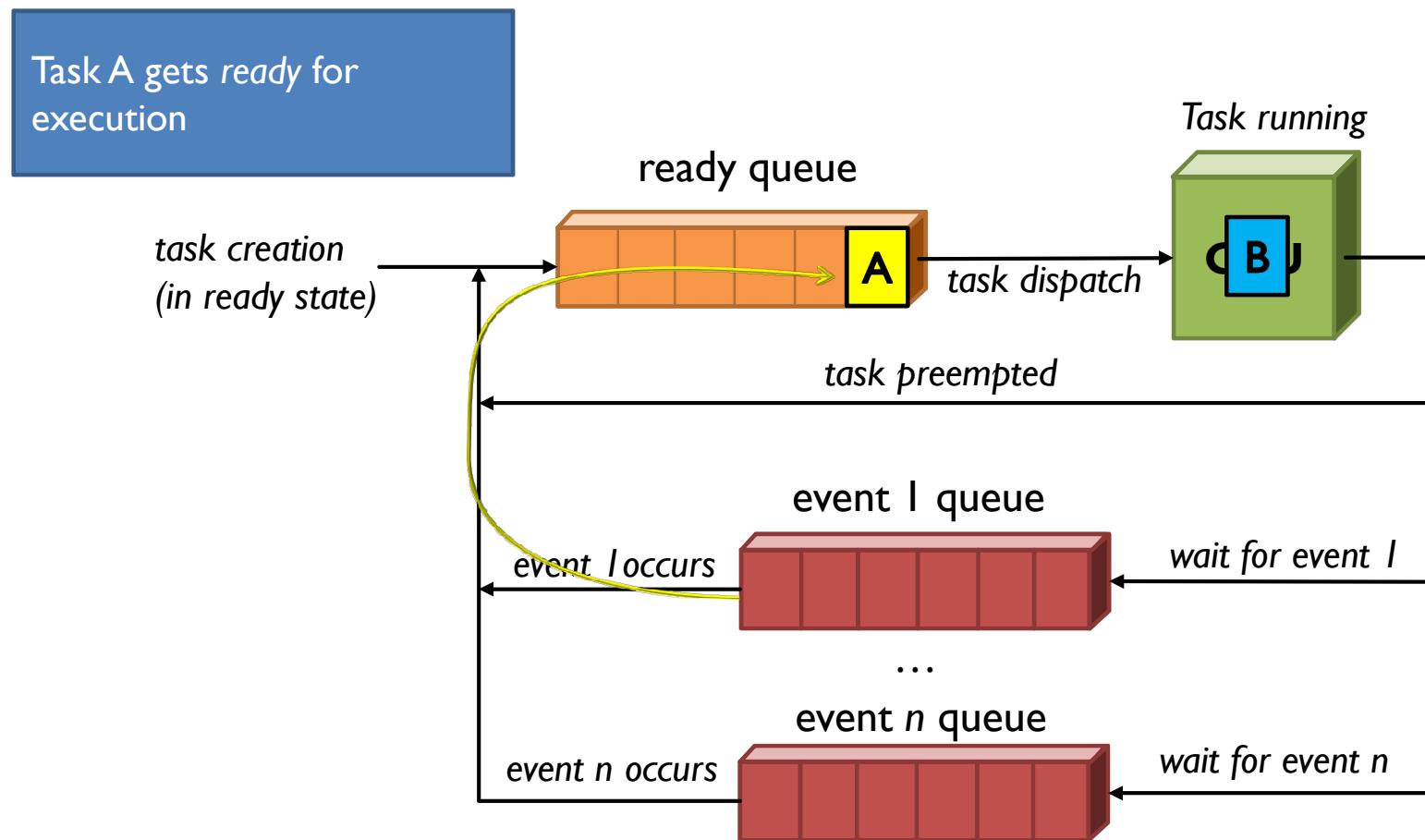
KERNEL STRUCTURE AND TASK MANAGEMENT



KERNEL STRUCTURE AND TASK MANAGEMENT

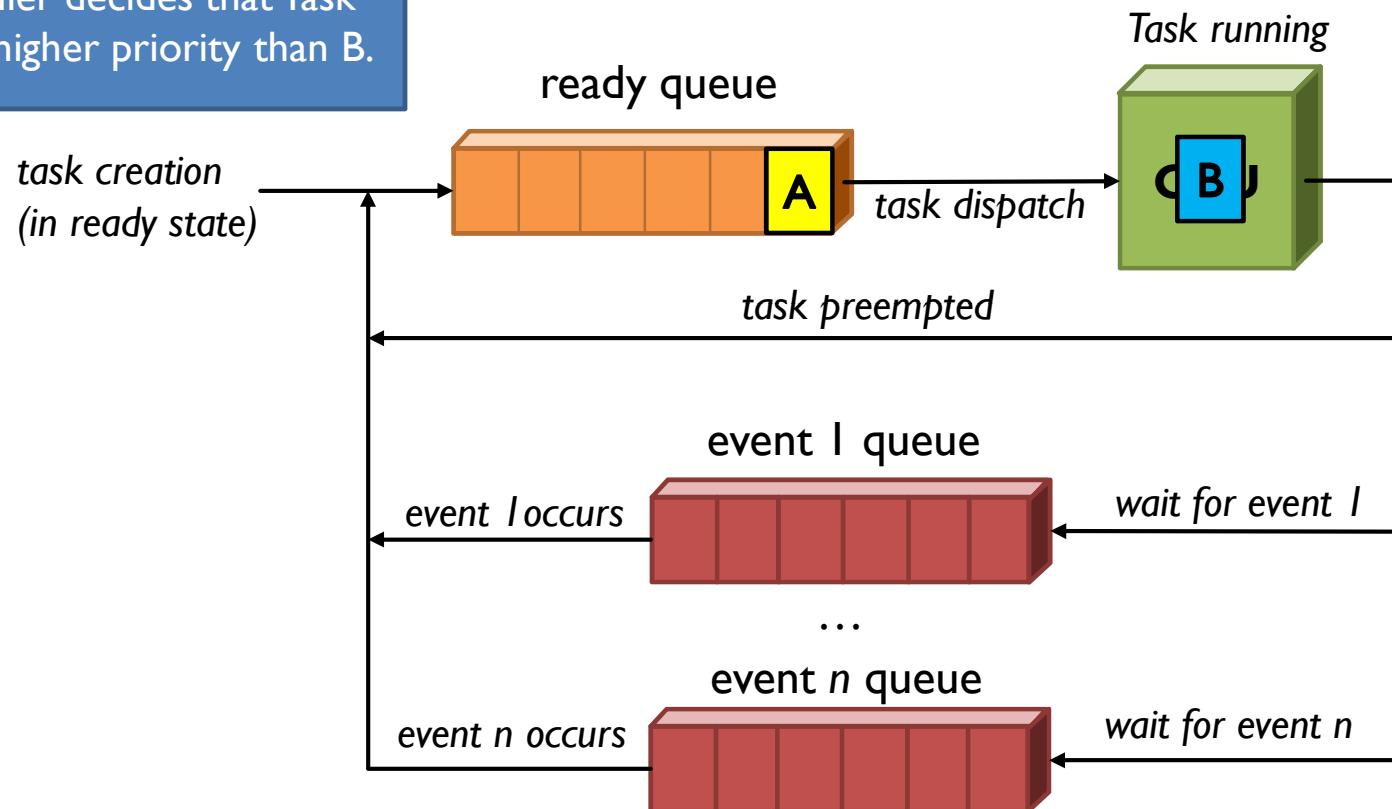


KERNEL STRUCTURE AND TASK MANAGEMENT



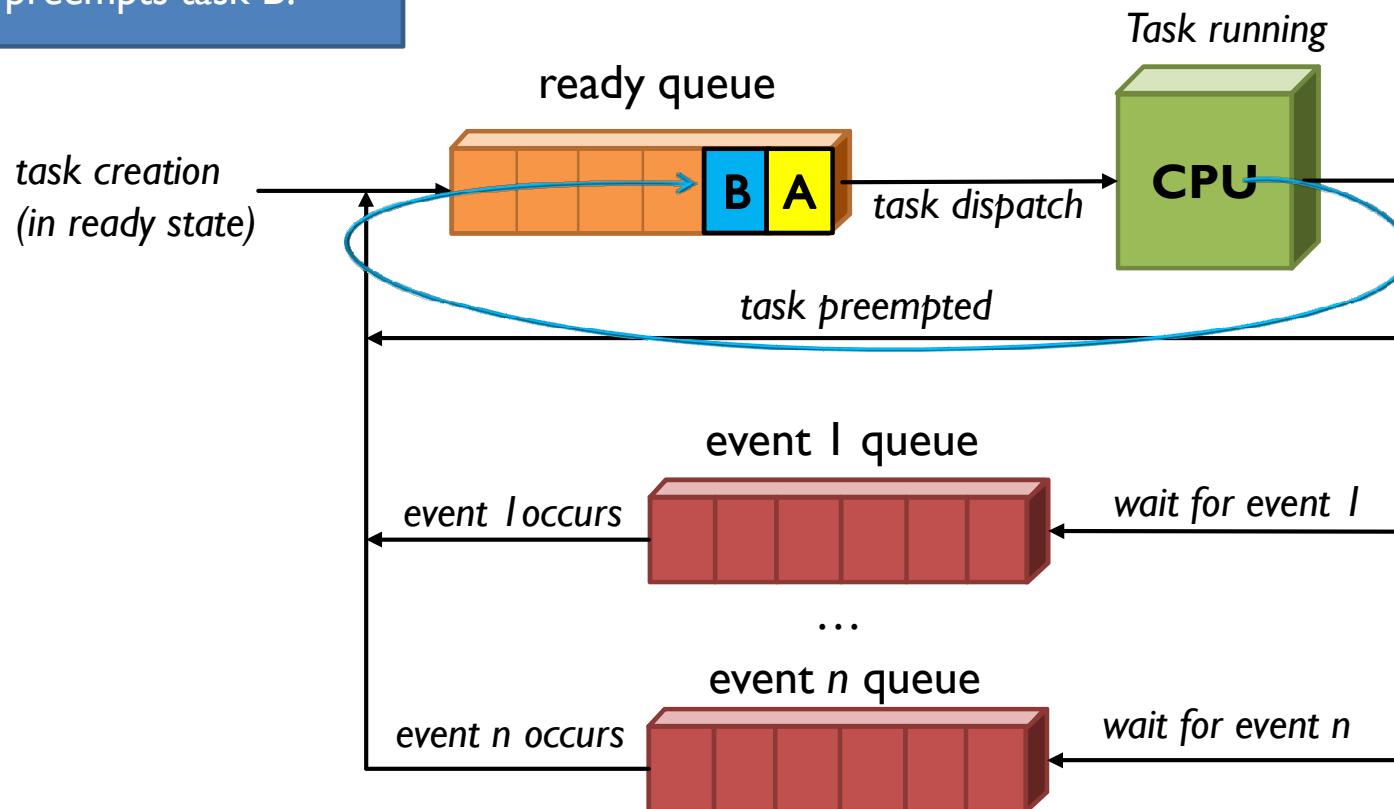
KERNEL STRUCTURE AND TASK MANAGEMENT

Scheduler decides that Task A has higher priority than B.

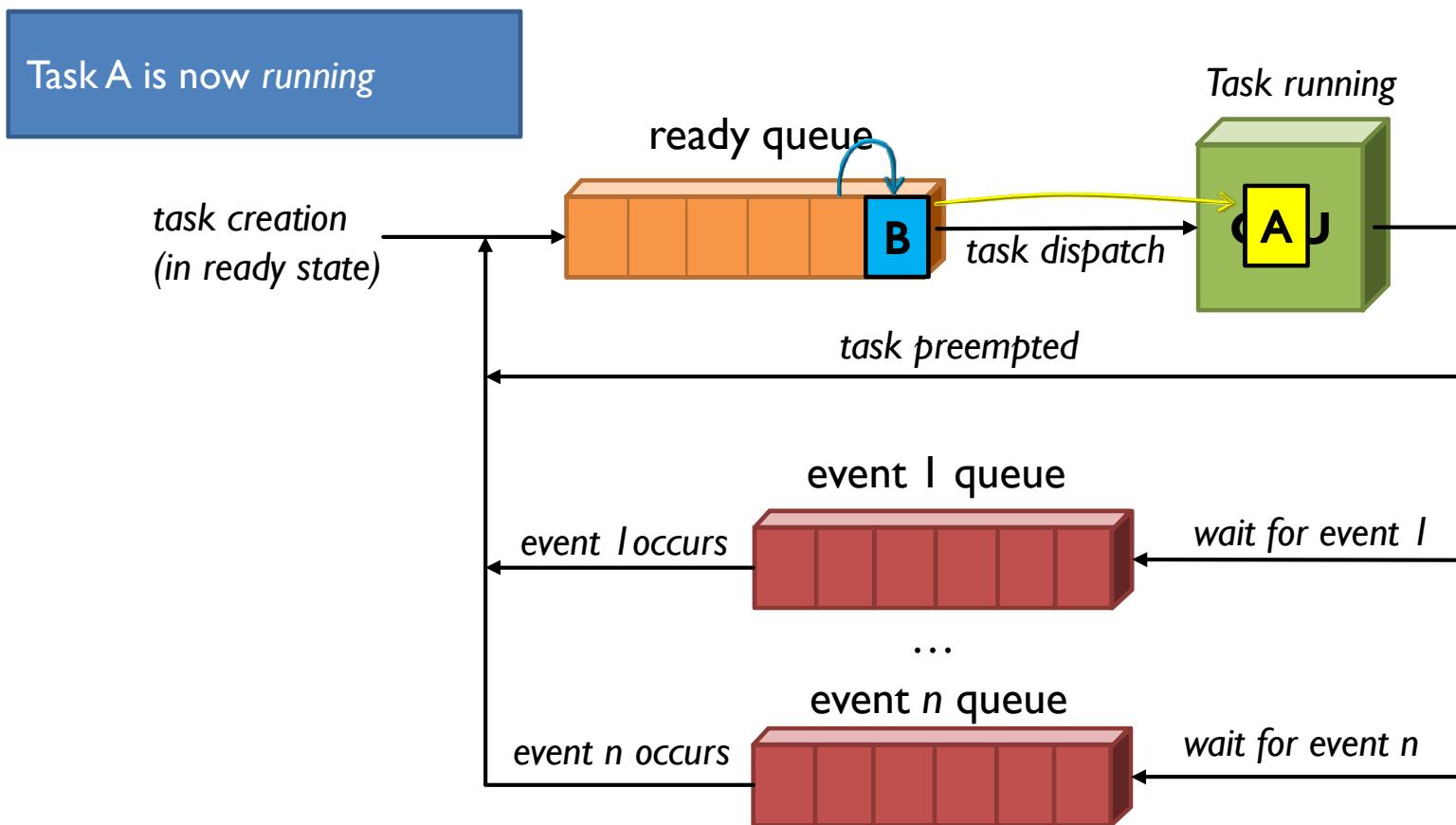


KERNEL STRUCTURE AND TASK MANAGEMENT

Task A preempts task B!



KERNEL STRUCTURE AND TASK MANAGEMENT



KERNEL STRUCTURE AND TASK MANAGEMENT

- Normally the “items” of the queues are pointers to the tasks TCBs..
- Dereferencing such pointers the RTOS can have access to the tasks parameters:
 - Read/Modify the task state
 - Call the task function
 - Handle the context of the task
 - ...

KERNEL STRUCTURE AND TASK MANAGEMENT

- Queues can be implemented as **arrays** (not exactly the best way of doing it)
 - Arrays have **fixed** size
 - For the ready queue
 - → We need to have a slot for each of the existing tasks in the system.
 - For **each** of the “event” queues
 - → We need to have a slot for each of the existing tasks in the system assuming that **all** may be waiting for the same event to happen
 - → An alternative is to only create a certain amount of slots for each event queue. This imposes **limitation** of how many tasks can be waiting for the same event to happen.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Operating over queues implemented with **ARRAYS...**
 - If insertion of items is needed, we need to check first if there is an **available** slot for such item → may not be the case!
 - For inserting data at the rear of the queue we need a **index** variable telling us where the last item is located at.
 - For removing items from the front of the queue:
 - Option A: Having a separate **index** telling us where is the “front” item located within the array.
 - Option B: Always take the first element of the array. Once this is done, all items need to be **shifted!**
 - Doing additional operations like inserting elements at any position between the front and the rear of the queue → lot of shifting involved.

KERNEL STRUCTURE AND TASK MANAGEMENT

- In summary arrays are NOT the best way of implementing the required queues for an RTOS (may be OK for other kind of applications)
- **Linked lists** are indeed a best way of implementing the required queues for the RTOS.
- A **Linked list** is a **dynamic** data structure whose length can be increased or decreased at **run time**.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Linked lists are normally related to *dynamic memory allocation* for the creation of new nodes of the list.
- In many critical real-time applications *dynamic memory allocation* is NOT allowed:
 - Raises the risk of memory overflows and other related issues at runtime.
 - Runtime issues are very hard to predict and correct
 - Lot of standards for example for automotive industry prohibit the use of dynamic memory allocation.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Are then linked lists still useful with no *dynamic memory allocation*?
 - **Yes!.**
- Without dynamic memory allocation we will need to create every “item” that can be in a queue at compile time.
 - All task’s TCBs
 - All events-related items, etc.
- Still and even without dynamic memory allocation the **length** of the queues can be changed **dynamically!**

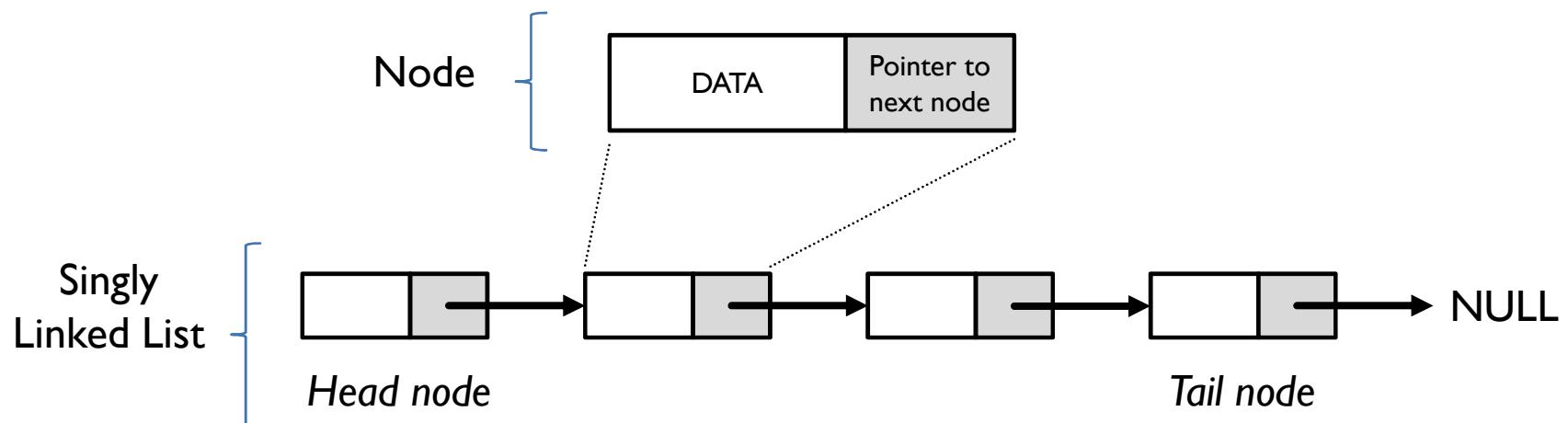
KERNEL STRUCTURE AND TASK MANAGEMENT

- Disadvantages of linked lists over arrays
 - Code is more complex since lot of **pointers** logic is involved
 - Nodes can not be accessed in a random way. E.g., we can not access node “5” of a linked list without first passing through 1,2,3 and 4.
- Random access to the nodes of the queues is normally not required in an RTOS so still linked lists are the best way to go for!
- Two common types of linked lists:
 - **Singly** linked lists.
 - **Doubly** linked lists.

KERNEL STRUCTURE AND TASK MANAGEMENT

■ Singly linked list

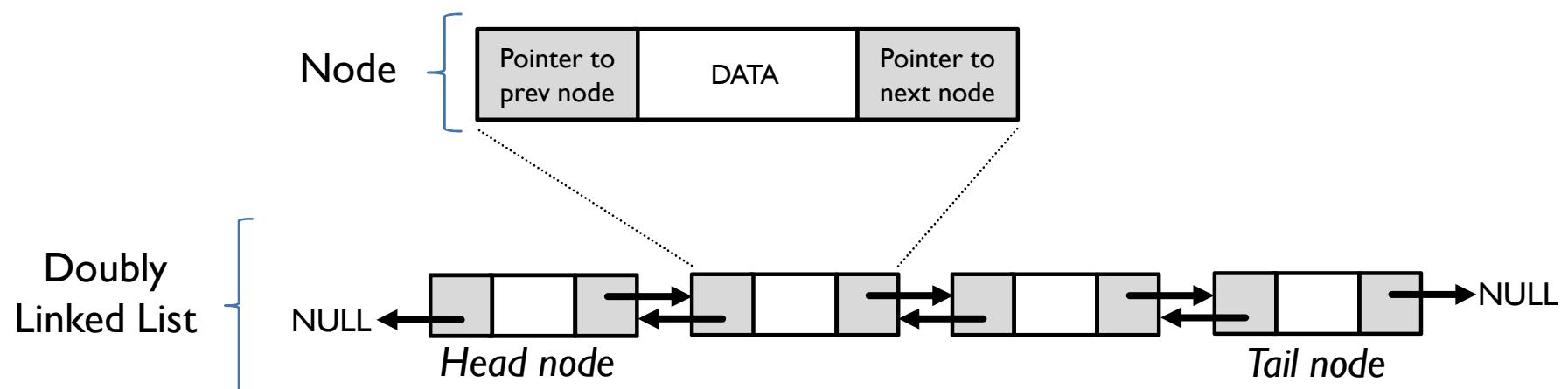
- Each node of the list has a “data” item and a **pointer** to the **next** element in the list.
- The list has an initial node called the “**head**” node.
- The last node called the “**tail**” node will have its “pointer to the next element” set to **NULL**



KERNEL STRUCTURE AND TASK MANAGEMENT

■ Doubly linked list

- Each node of the list has a “data” item, a pointer to the **previous** element and a pointer to the **next** element in the list.
- The list has an initial node called the “**head**” node. The “pointer to the previous element” of the head node is set to **NULL**.
- The last node called the “**tail**” node will have its “pointer to the next element” set to **NULL**

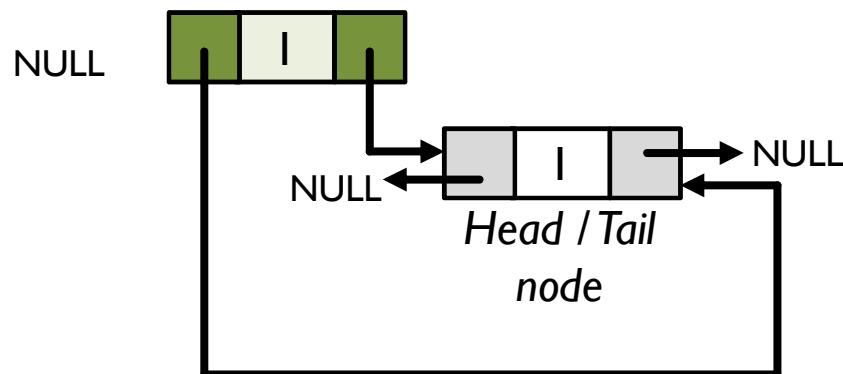


KERNEL STRUCTURE AND TASK MANAGEMENT

- **Doubly linked list example**
- Creating a doubly linked “list”

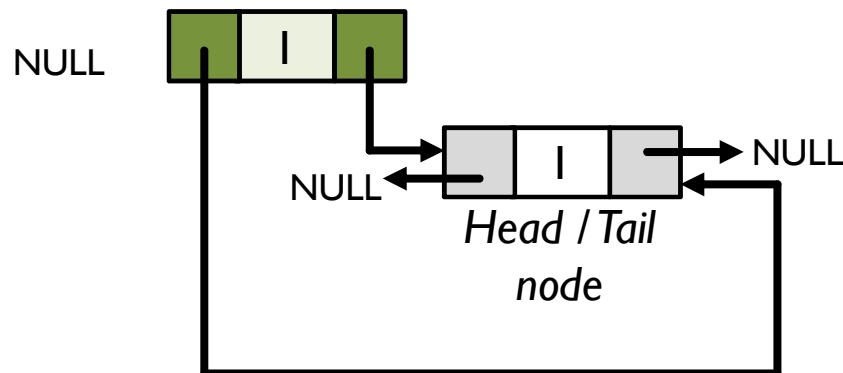


- **Adding the first element to the list.** This first element will be at the same time the *head* and the *tail* of the link.



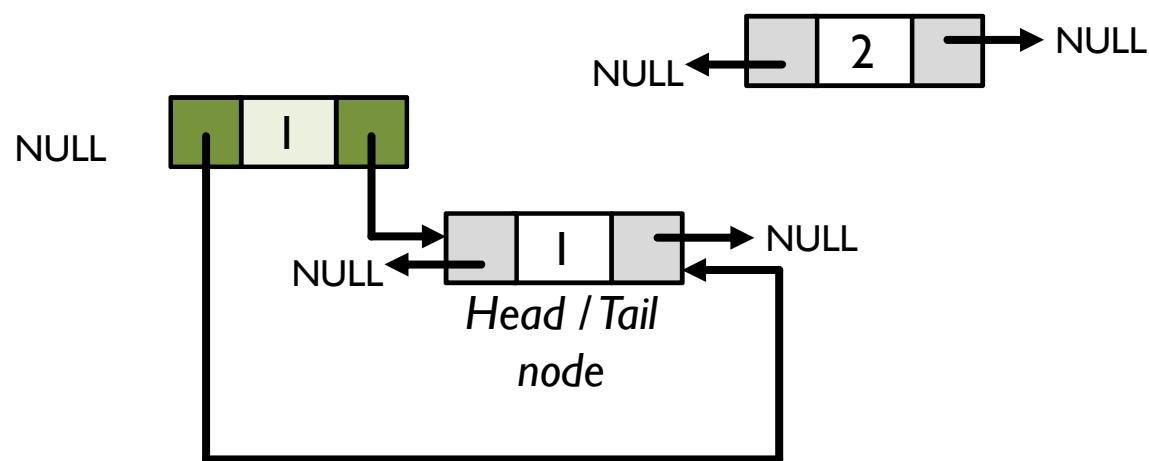
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list example...**
- **Adding the first element to the list.** This first element will be at the same time the *head* and the *tail* of the link.
 - a) We now it is the first element by reading the “Node counter” as 0.
 - b) “Head” of the linked list is set to point to the new node
 - c) “Tail” of the linked list is set to point to the new node
 - d) Node Counter is set to 1



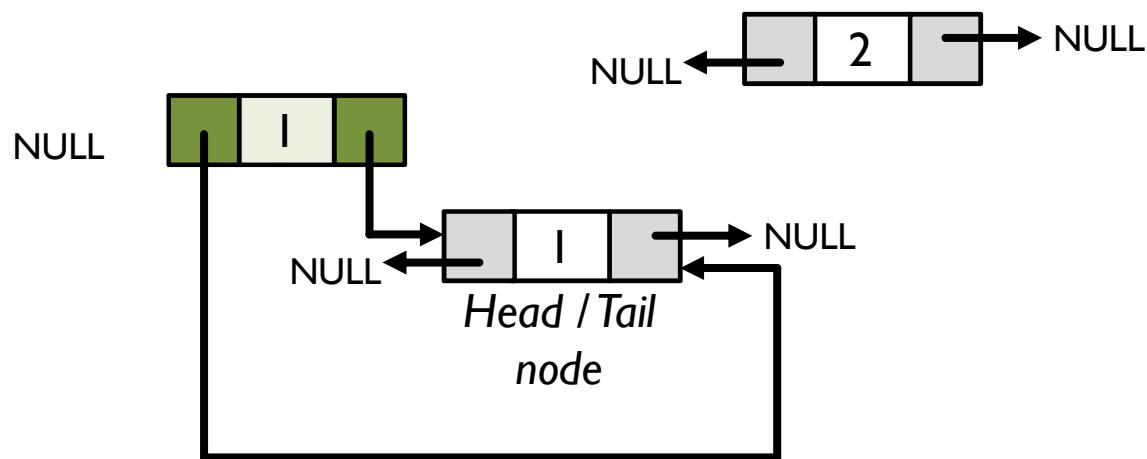
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- **Adding** a second element to the tail of the list (**pushing** a new element)



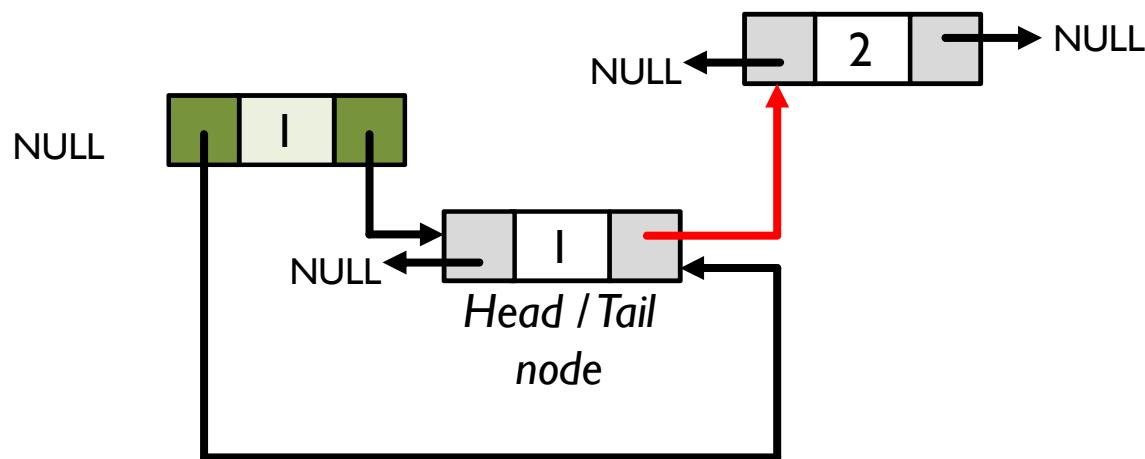
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- Adding a second element to the tail of the list (pushing a new element)
 - a) We know this new node is not the first one because Node Counter > 0



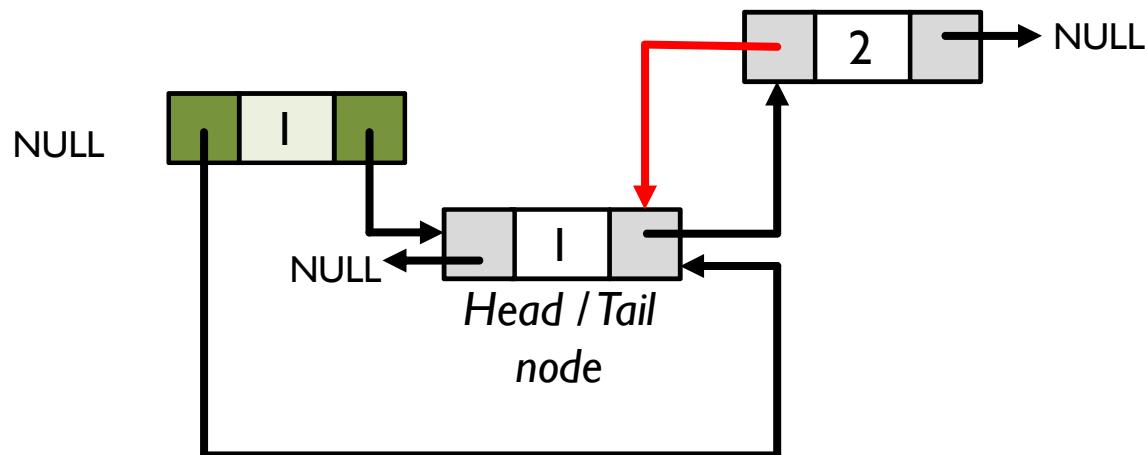
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- Adding a second element to the tail of the list (pushing a new element)
 - a) We know this new node is not the first one because Node Counter > 0
 - b) “Next” of the *tail* element is set to point to the new element



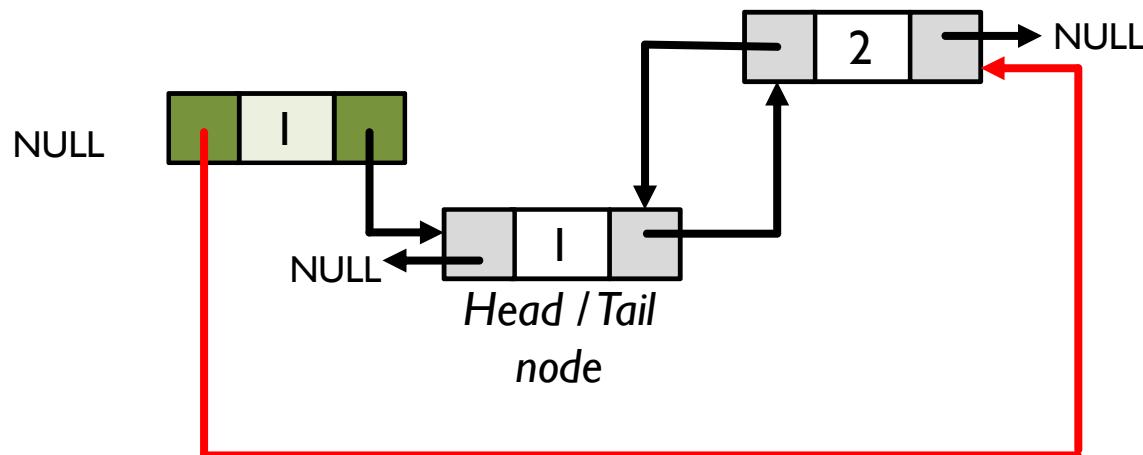
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- Adding a second element to the tail of the list (pushing a new element)
 - a) We know this new node is not the first one because Node Counter > 0
 - b) “Next” of the *tail* element is set to point to the new element
 - c) “Prev” of new element is set to point to current *tail* element (the first element we inserted)



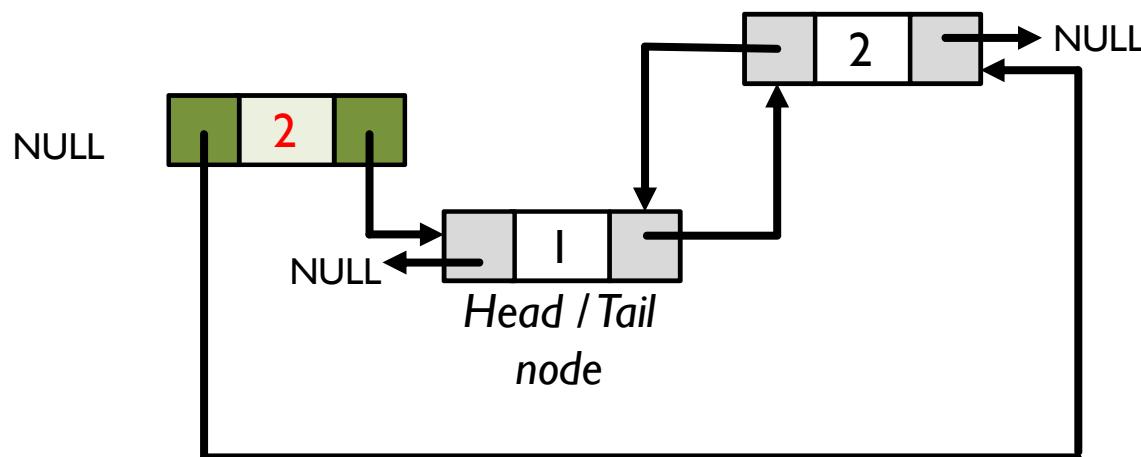
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- Adding a second element to the tail of the list (pushing a new element)
 - We know this new node is not the first one because Node Counter > 0
 - “Next” of the *tail* element is set to point to the new element
 - “Prev” of new element is set to point to current *tail* element (the first element we inserted)
 - “Tail” of *linked list* is set to point to the new element



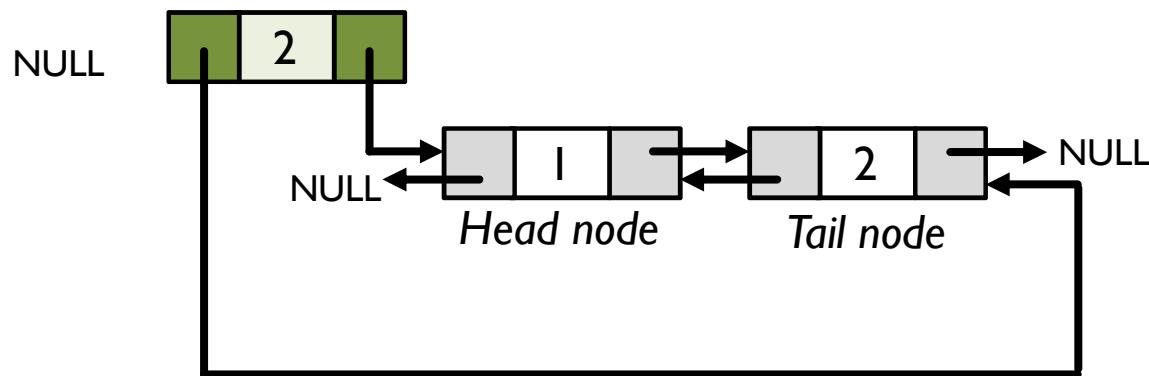
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- Adding a second element to the tail of the list (pushing a new element)
 - We know this new node is not the first one because Node Counter > 0
 - “Next” of the *tail* element is set to point to the new element
 - “Prev” of new element is set to point to current *tail* element (the first element we inserted)
 - “Tail” of *linked list* is set to point to the new element
 - Node counter of *linked list* is incremented



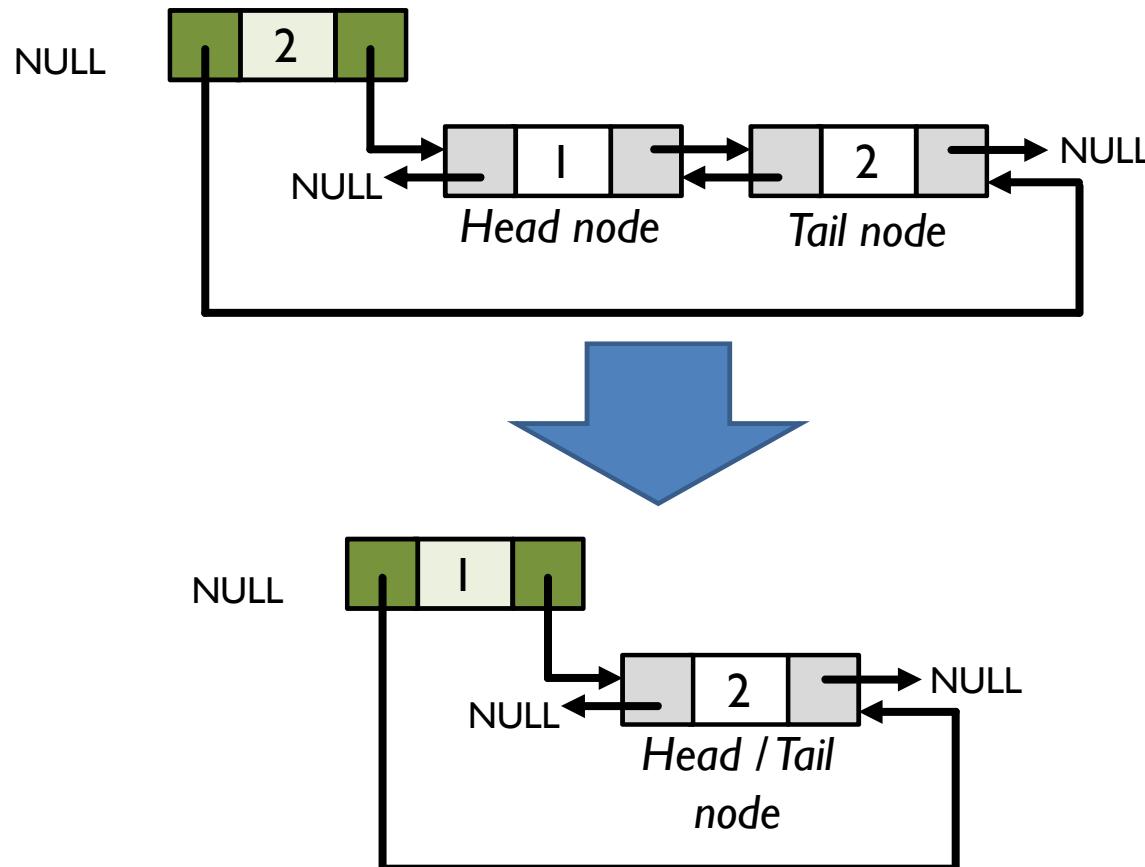
KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- Adding a second element to the tail of the list (pushing a new element)
 - We know this new node is not the first one because Node Counter > 0
 - “Next” of the *tail* element is set to point to the new element
 - “Prev” of new element is set to point to current *tail* element (the first element we inserted)
 - “Tail” of *linked list* is set to point to the new element
 - Node counter of *linked list* is incremented



KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- **Get and Remove** the *head* element from the list (**poping** the first element)



KERNEL STRUCTURE AND TASK MANAGEMENT

- ... **Doubly linked list** ...
- ...**Get** and **Remove** the *head* element from the list (**poping** the first element)...
 - a) Check that the list is not empty (Node Counter > 0)
 - b) Data of the *head* node is retrieved

If this node being popped is the only one in the list then:

- b) “Head” and “Tail” of the *linked list* are set to NULL
- c) Node counter of *linked list* is decremented

If the node being popped is not the only one in the list then:

- b) “Prev” of the node pointed by the “Next” of the *head* node is set to NULL
- c) “Head” of the *linked list* is set to point to the node pointed by the “Next” of the *head* node
- d) Node counter of *linked list* is decremented

KERNEL STRUCTURE AND TASK MANAGEMENT

Example: C implementation of a doubly linked list

- Defining types for a *node* and for a *list*

```
typedef struct ListNode {
    struct ListNode *next;
    struct ListNode *prev;
    int data;
} ListNode;

typedef struct List {
    int count;
    ListNode *head;
    ListNode *tail;
} List;
```

KERNEL STRUCTURE AND TASK MANAGEMENT

Example: C implementation of a doubly linked list

- Statically create the *nodes* and the *list*

```
/* Creating a list */
List myDoublyLinkedList;

/* Creating nodes (they do not belong to the
list yet) */
ListNode node1;
ListNode node2;
ListNode node3;
```

KERNEL STRUCTURE AND TASK MANAGEMENT

Example: C implementation of a doubly linked list

- Initialize the nodes and the list

```
/* Initializing the empty list */
myDoublyLinkedList.count = 0;
myDoublyLinkedList.head = NULL;
myDoublyLinkedList.tail = NULL;

/* Initializing the nodes (not yet linked)*/
node1.value = 1;
node1.next = NULL;
node1.prev = NULL;

node2.value = 2;
node2.next = NULL;
node2.prev = NULL;

node3.value = 3;
node3.next = NULL;
node3.prev = NULL;
```

KERNEL STRUCTURE AND TASK MANAGEMENT

Example: C implementation of a doubly linked list

- Function for inserting a node at the *tail* -last- node of the list (push a node)

```
void List_pushToTail(List *list, ListNode *newNode)
{
    /* Check if the list is empty */
    if(!list->count) {
        /* list is empty, this will be the head node */
        list->head = newNode;
        list->tail = newNode;
    } else {
        /* list is NOT empty, insert node at the tail */
        list->tail->next = newNode;
        newNode->prev = list->tail;
        list->tail = newNode;
    }
    list->count++;
}
```

KERNEL STRUCTURE AND TASK MANAGEMENT

Example: C implementation of a doubly linked list

- Function for retrieving and removing the *head* -first- node of the list (pop the head node)

```
ListNode * List_popHead(List *list)
{
    ListNode * returnNode = NULL;

    if(!list->count){
        /* Nothing to return, list is empty */
    } else
    {
        /* Retrieve Head Node */
        returnNode = list->head;
        /* Remove head node from list */
        if(list->count == 1){
            list->head = NULL;
            list->tail = NULL;
        } else if(list->count > 1) {
            list->head->next->prev = NULL;
            list->head = list->head->next;
        }
        list->count--;
    }

    return returnNode;
}
```



KERNEL STRUCTURE AND TASK MANAGEMENT

- Let's analyse an example of an implementation of the ready list as a set of **ready queues**.
- The example will be using **double linked lists** for implementing the queues.



KERNEL STRUCTURE AND TASK MANAGEMENT

Implementing the Ready List

KERNEL STRUCTURE AND TASK MANAGEMENT

- A **ready list** can be implemented as a **queue** where each of the nodes is indeed a Task Control Block (TCB).

KERNEL STRUCTURE AND TASK MANAGEMENT

- In a fixed priority system, tasks do not necessarily have **unique** priorities for each one.
- Indeed, more than one task can **share** the same priority!

KERNEL STRUCTURE AND TASK MANAGEMENT

- If the RTOS has the limitation that each task should have its unique priority, then a **single** ready queue can be enough.
 - At the time a task becomes ready, it gets “inserted” in the ready queue (more precisely, its TCB gets inserted in the ready queue).
 - The kernel can do “ordered” insertions. This means that every time a new ready task needs to be inserted, the kernel first check its priority and inserts such task in the correspondin queue position.
 - → The list needs to always kept ordered and hence, for example, the “head” of the queue will always hold the task with the higher priority.

KERNEL STRUCTURE AND TASK MANAGEMENT

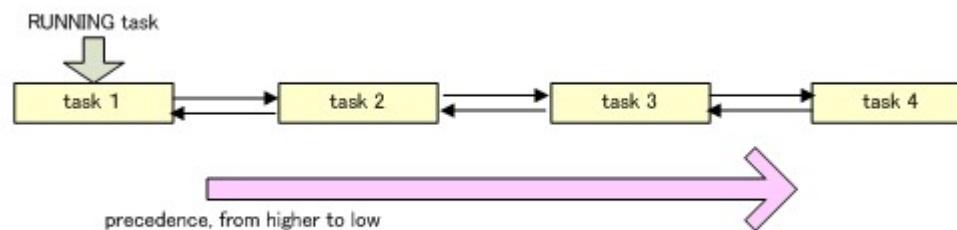
- If the RTOS allows two or more tasks to have the same priority:
 - An implementation with several **ready queues**, one for each **priority** results very efficient.
 - Each of those ready queues are in fact FIFOs

KERNEL STRUCTURE AND TASK MANAGEMENT

- Based on this, some commonly required **ready queue** operations are:
 - To find the task that has the highest precedence
 - To append a task to a queue of tasks that share the same priority
 - To delete a task from the queue
 - To search the task that has the highest precedence among the ones with a specified priority

KERNEL STRUCTURE AND TASK MANAGEMENT

- Almost all the RTOS service calls will operate the ready queues!
- A structure with **high efficiency** is especially important, or execution of these service-calls will lead to low performance.
- A ready queue can be viewed as tasks being arranged in order of **precedency** from high to low



- Such a ready queue can be implemented by a double linked list.

KERNEL STRUCTURE AND TASK MANAGEMENT

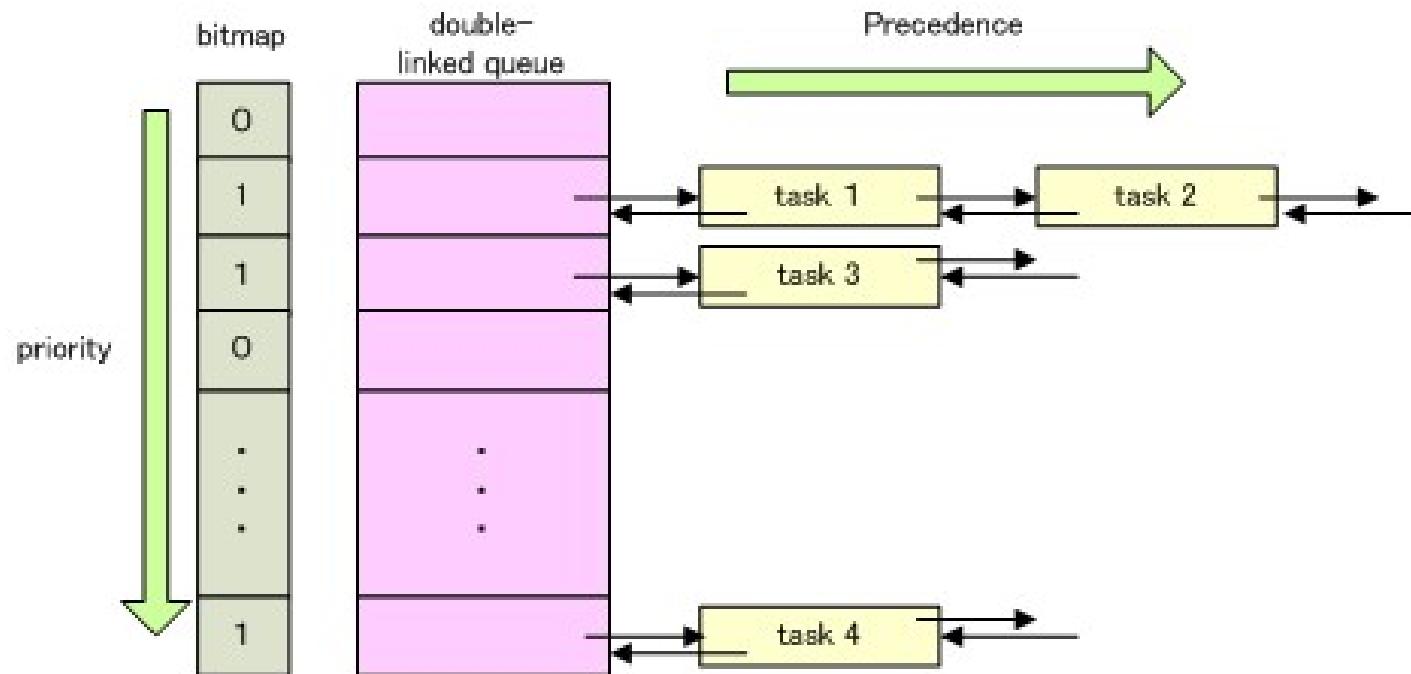
- Then, a double linked list (a ready queue) for **each** priority is required.
 - I. We will have an “array” of linked lists / ready queues. The array length is determined by the number of allowed priorities.
 - Why don’t we use a “linked list” of “linked lists” in this case? → Not efficient...
 - II. Each array “index” corresponds to a given priority
 - III. We need to “navigate” the array searching for “ready tasks”
 - IV. In order to navigate such array of linked list we may read each of the “linked list” elements to get the number of nodes that such linked list has:
 - a) Node counter = 0 → No task is ready for the given priority (list is empty)
 - b) Node counter > 0 → At least one task is ready for the given priority (list is not empty)
 - V. Once a non-empty linked list is found, we get the *head* element that indeed is a task TCB and switch context to such task.

KERNEL STRUCTURE AND TASK MANAGEMENT

- Previous steps, specially step IV is still not efficient enough.
- Instead of “reading” each of the linked lists to determine if they are empty or not, we can use an additional construct:
 - A **bit-map**
- Such a **bit-map** will reserve 1 bit for each priority.
 - If the bit is set to 0 → ready queue for such priority is empty
 - If the bit is set to 1 → ready queue for such priority is NOT empty
- The order of the bits are indeed aligned with the indexes of the array (and hence with the priorities)
 - Bit 0 corresponds to index 0 of the array of ready queues,
 - Bit 1 corresponds to index 1 of the array of ready queues,
 - ...

KERNEL STRUCTURE AND TASK MANAGEMENT

- Ready list implemented with several **ready queues** (double linked lists) and a **bitmap**:



KERNEL STRUCTURE AND TASK MANAGEMENT

■ Priority and Precedence

- Priority is A PARAMETER which is specified by a application, while Precedence is A CONCEPT used to make clear the execution order.
- For example, when several tasks are holding the same priority, it is their precedence the one that determines their execution order.

KERNEL STRUCTURE AND TASK MANAGEMENT

- How did we gain efficiency by adding this **bitmap**?
 - Some processors have a “Count Leading Zeros” instruction which makes this enquiry to the bitmap very fast.
 - E.g. ARM9 and above MCUs have a CLZ that does as follow:
 - “The *CLZ* instruction counts the number of leading zeros in the value in *Rm* and returns the result in *Rd*. The result value is 32 if no bits are set in the source register, and zero if bit 31 is set.”
 - If processor does not have such instruction, still evaluating bits is faster than accessing the “nodes counter” variable of each of the ready queues.

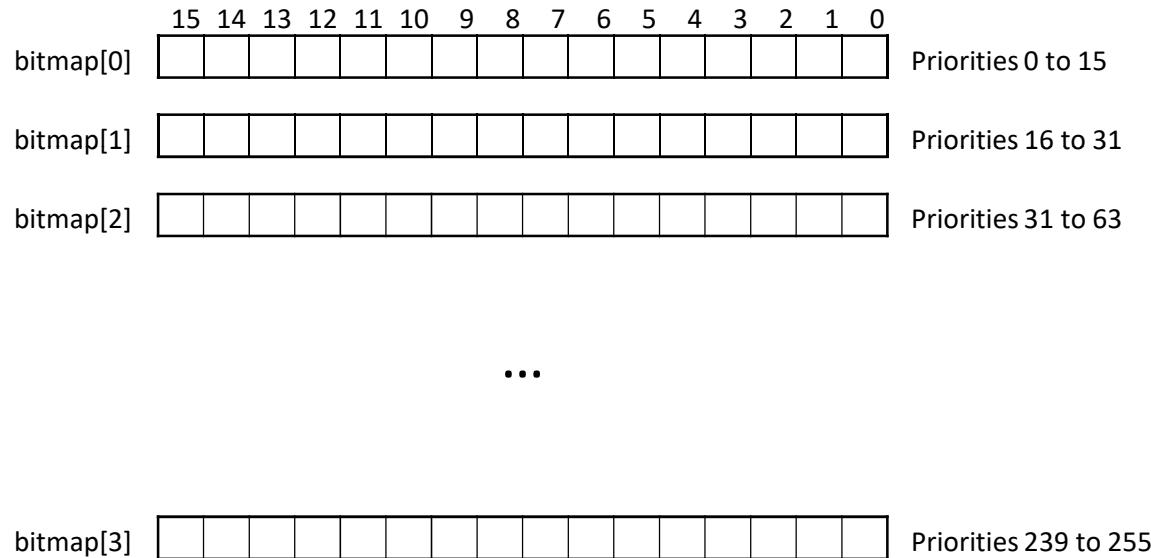
KERNEL STRUCTURE AND TASK MANAGEMENT

■ **Bitmap** Implementation

- Normally several “word” sized variables are used to create the bitmap.
- In this context a “word” means the data type which corresponds to the MCU architecture (will correspond to the *int* data type size for the specific MCU):
 - E.g., 16-bits MCUs → a word is of 16-bits
 - E.g., 32-bits MCUs → a word is of 32-bits
- If 256 different priorities are allowed for the tasks and the MCU is of 32 bits then **8 *int*** variables will be required for the **bitmap**.
 - To fully navigate the bitmap, the CLZ instruction will be used 8 times together with few other instructions to evaluate the result of the CLZ. This is actually very efficient!

KERNEL STRUCTURE AND TASK MANAGEMENT

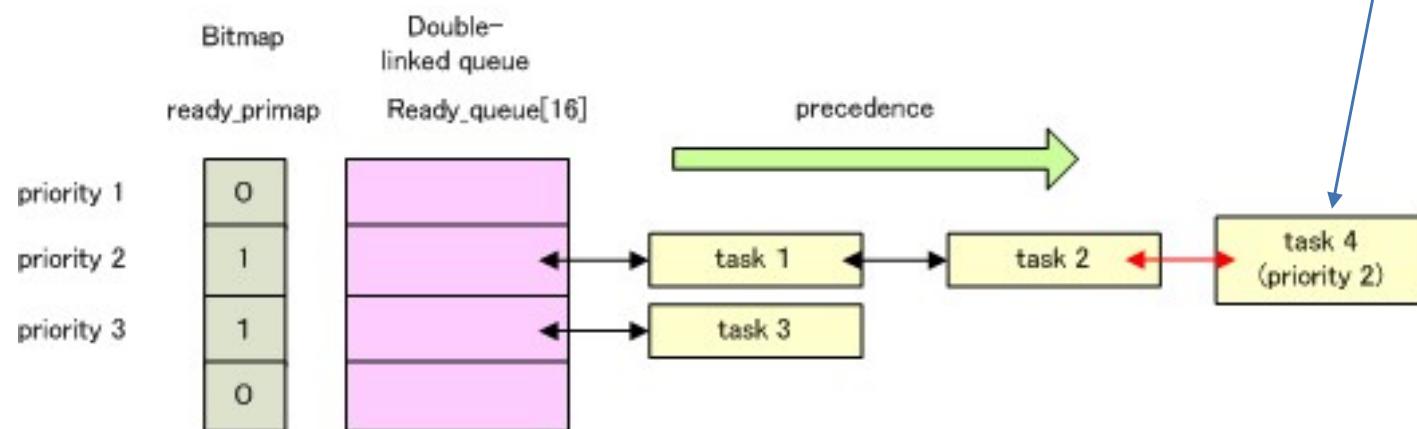
- **Bitmap Implementation...**
- **256 priorities, word size = 16 bits...**



KERNEL STRUCTURE AND TASK MANAGEMENT

- ... Back to the **ready list**...
- How to put a task into the ready list?
- Example: Task 4 becomes ready. It has a priority = 2.

New ready task (Task 4) gets inserted into the “priority 2” ready queue



KERNEL STRUCTURE AND TASK MANAGEMENT

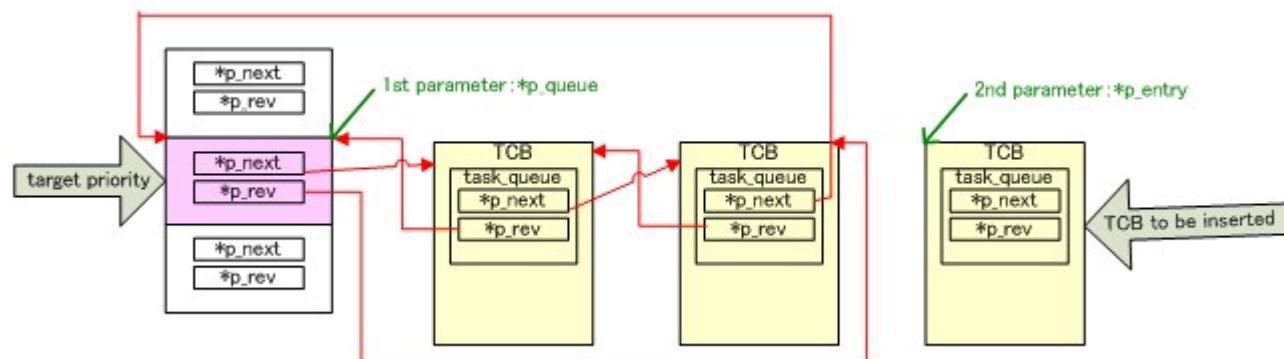
- **Example:** Function to **insert** a task to the ready list

```
1  inline void queue_insert_prev(QUEUE *p_queue, QUEUE *p_entry)
2  {
3      p_entry->p_prev = p_queue->p_prev;
4      p_entry->p_next = p_queue;
5      p_queue->p_prev->p_next = p_entry;
6      p_queue->p_prev = p_entry;
7  }
```

- 1st parameter: the ready queue's address of the involved priority.
- 2nd parameter: the address of <task queue> , a member of the target task's TCB.

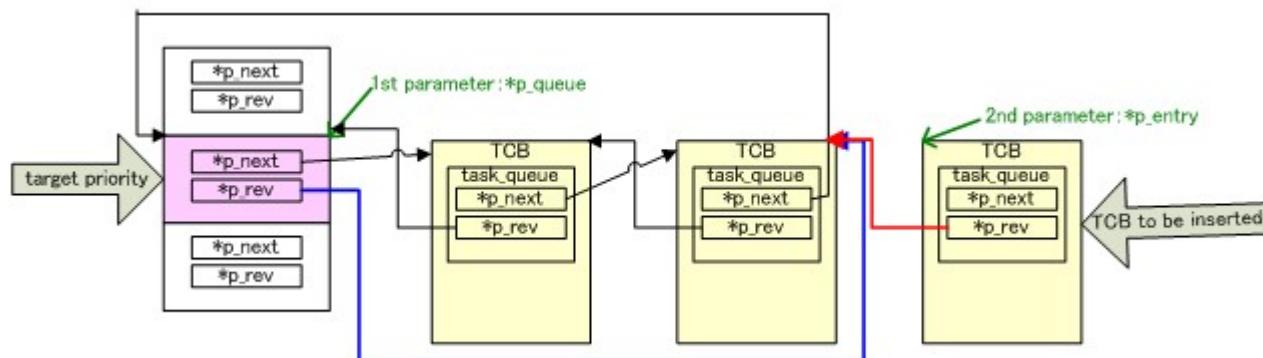
KERNEL STRUCTURE AND TASK MANAGEMENT

- ...**Example:** Function to **insert** a task to the ready list...
- Original state of the ready queue



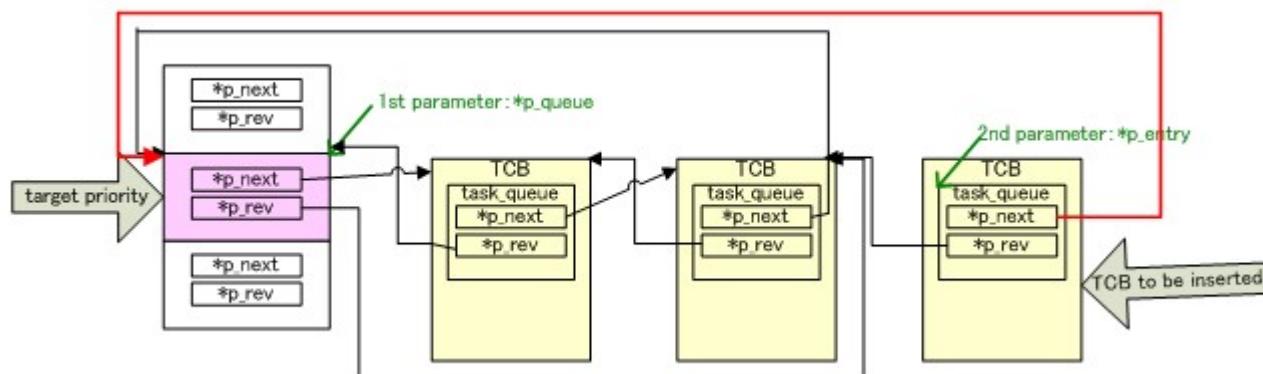
KERNEL STRUCTURE AND TASK MANAGEMENT

- ...**Example:** Function to **insert** a task to the ready list...
- Line 3: `p_entry->p_prev = p_queue->p_prev;`
Makes the target TCB's `p_prev` point to another TCB, to which `p_prev` of the current double-linked queue top is pointing.



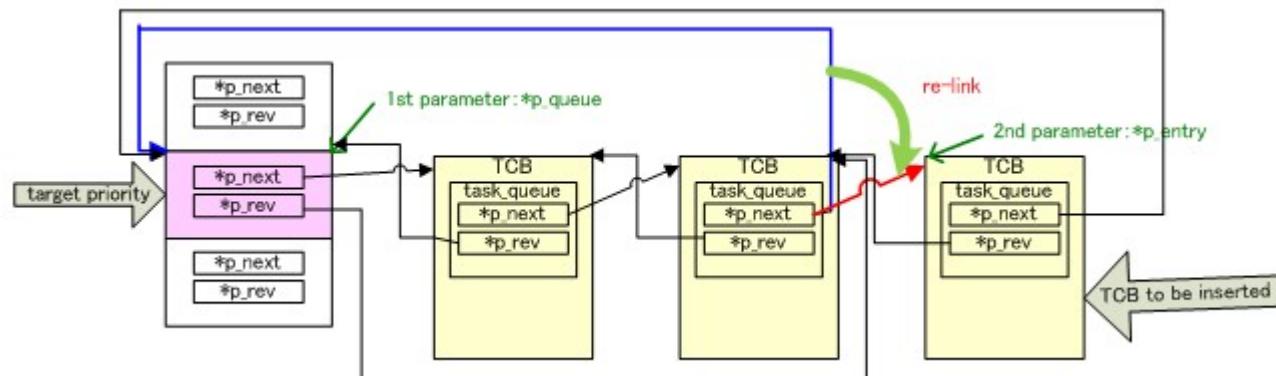
KERNEL STRUCTURE AND TASK MANAGEMENT

- ...**Example:** Function to **insert** a task to the ready list...
- Line 4: `p_entry->p_next = p_queue;`
Forces the target TCB's `p_next` point to the double-linked queue top.



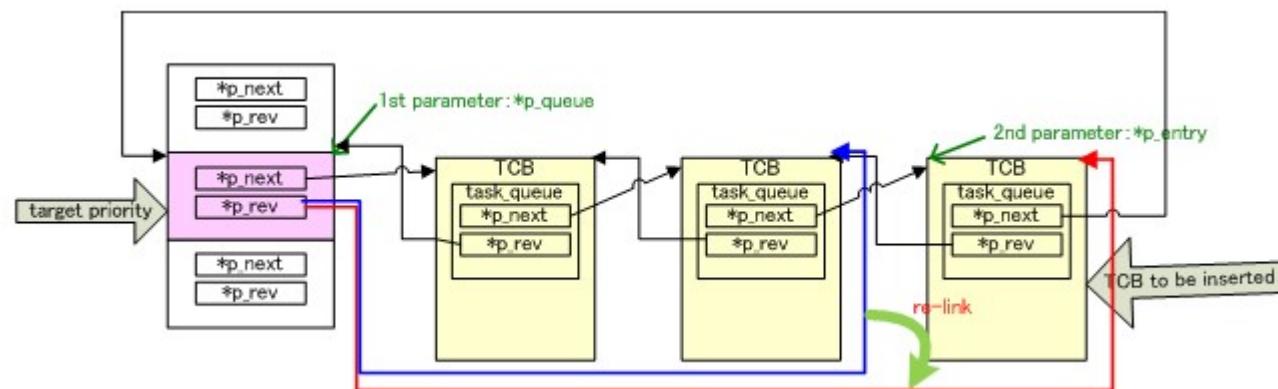
KERNEL STRUCTURE AND TASK MANAGEMENT

- ...**Example:** Function to **insert** a task to the ready list...
- Line 5: `p_queue->p_prev->p_next = p_entry;`
Makes `p_queue->p_prev->p_next`, the double-linked queue top's `p_prev`, also `p_next` of the TCB at the tail, point to `p_entry`.



KERNEL STRUCTURE AND TASK MANAGEMENT

- ...**Example:** Function to **insert** a task to the ready list...
- Line 6: `p_queue->p_prev = p_entry;`
Makes the double-linked queue top's `p_prev` point to `p_entry`.



OVERVIEW

- Real-Time Systems Concepts
- Real-Time Operating Systems Concepts
- Kernel Structure and Task Management
- **Time Management**
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- References



TIME MANAGEMENT

Time Management



TIME MANAGEMENT

- Time is a central concern within an RTOS
- Times related to tasks management
 - Task can be waiting due to it has called a “delay” function.
 - Task can be waiting for an event for a maximum specified time (*timeout*)
 - Timing base for periodic tasks, etc.
- Timer services to be used by the application
 - Timers
 - One shot
 - Auto-reload, etc.

TIME MANAGEMENT

- APIs that may cause a Task to go to a **blocking** state can be clasified in two types:
- **Blocking APIs**
 - If the condition or event for which the task is waiting for is not present when the function (API) is called then the Task is blocked and the function does **NOT** return.
 - Once the condition or event is met then the Task is resumed and the function returns with an status code.
- **Non-blocking APIs**
 - If the condition or event for which the task is waiting for is not present when the function is called then it does return without blocking the task but returning a special status code telling that the condition or event was not present.
 - Application is responsible for leading the task flow safely upon the “status codes”.
 - This kind of APIs are normally used when “polling” for an event.



TIME MANAGEMENT

Time for Tasks Management

TIME MANAGEMENT

Delaying a Task

- RTOS usually provide an API to **delay** a task for a given time.
 - While the task is waiting, other task can be running.
 - Time can be specified in different time *units*
 - Clock Ticks
 - HW Ticks
 - HMSM (Hours, Minutes, Seconds, Milliseconds)
- Additionally, RTOS also provide an API to **resume** a task that has been delayed.

TIME MANAGEMENT

- Delay APIs examples:
- μC/OS II
 - OSTimeDelay(),
 - OSTimeDelayHMSM(),
 - OSTimeDelay(),
 - OSTimeDelay()
 - ...
- MQXLite
 - _time_delay_for()
 - _time_delay_until()
 - _time_delay_ticks()
 - _time_diff_ticks()
 - _time_get_elapsed_ticks()
 - _time_get_hw_ticks()
 - ...

TIME MANAGEMENT

Timeout for events waiting

- Task can be waiting for an event to occur in order to continue its execution
 - Semaphore
 - Event Flag,
 - Message, etc.
- It is a good idea to provide a **timeout** for such event(s) to occur.
- If the timeout expires **before** the event occurs, then the Task is resumed.
 - The blocking API returns a special error code telling that the task was resumed because the **timeout elapsed** and not because the event happened.



TIME MANAGEMENT

- When a Task is “waiting” for a given time to expire it is removed from the **ready list** and changed to a blocked state.



TIME MANAGEMENT

Timer Services



TIME MANAGEMENT

- RTOS normally provide services for creating “timers” to be used by the application code.
- Typical kind of **timers**
 - One-Shot timers
 - One-Shot timers with initial delay
 - Auto-reload timers
 - Auto-reload timers with initial delay



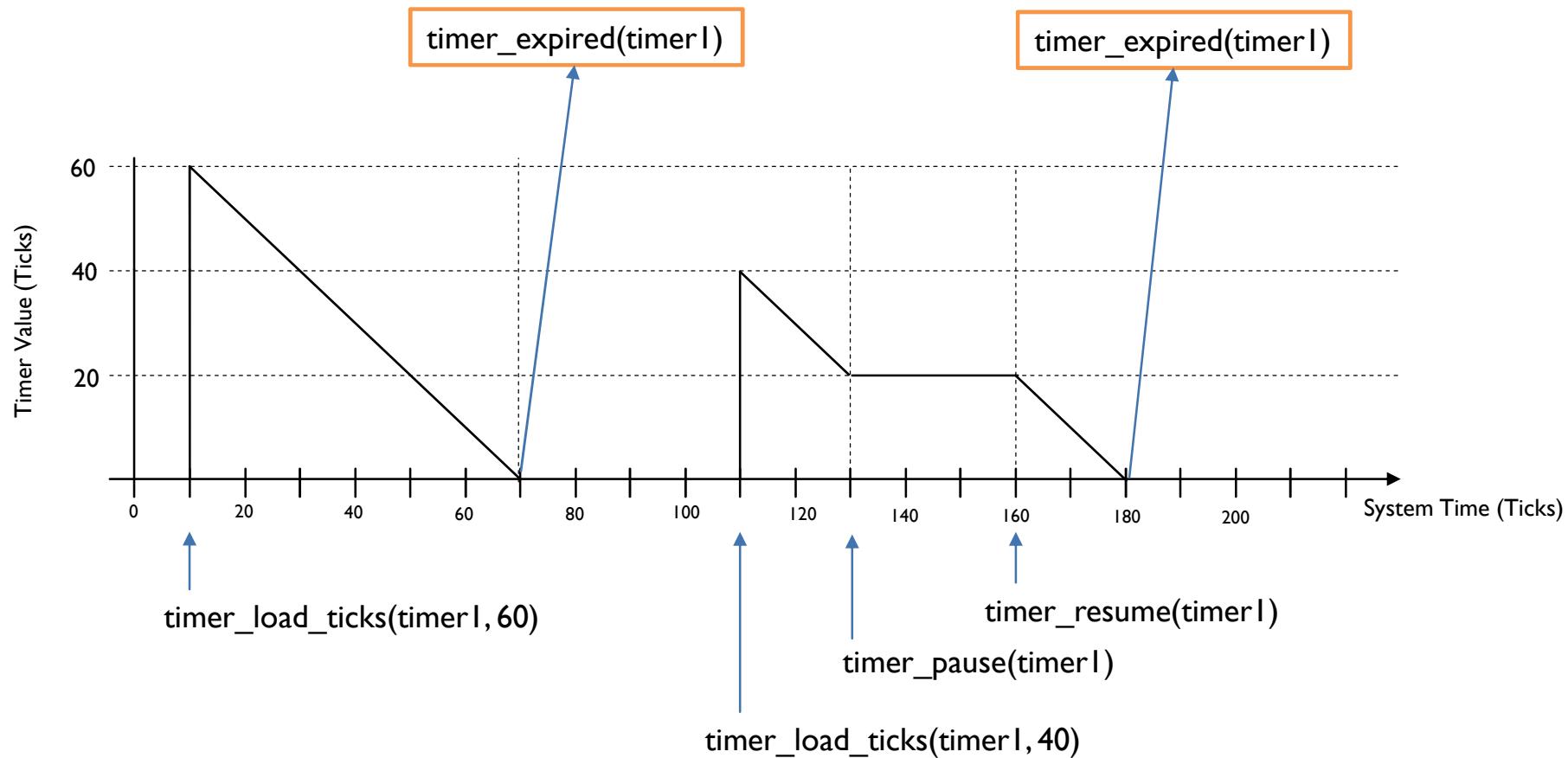
TIME MANAGEMENT

- RTOS normally provide services for creating “timers” to be used by the application code.
- The resolution of those timers are based in the clock tick interrupt.
- Typical kind of **timers**
 - One-Shot timers
 - Auto-reload timers
 - Auto-reload timers with initial delay

TIME MANAGEMENT

■ One-Shot timers

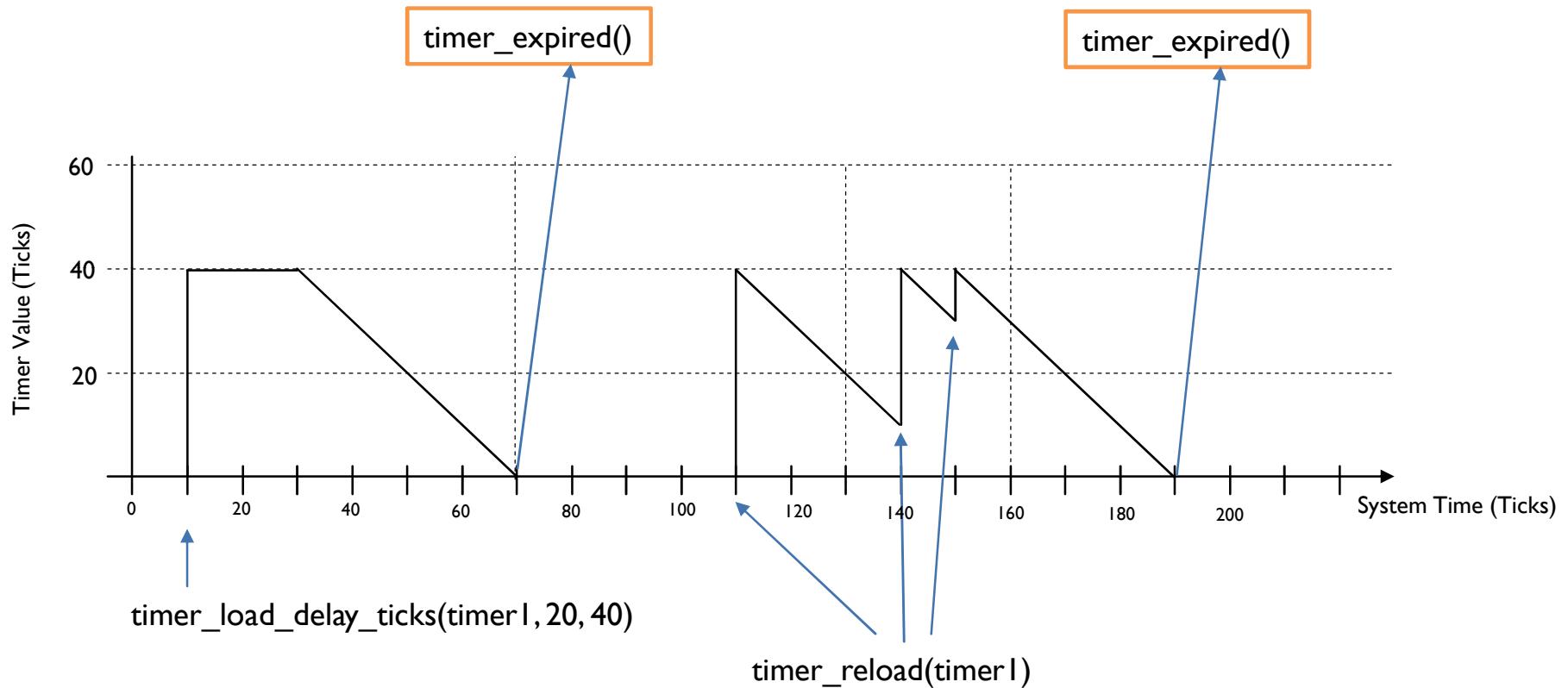
`timer_load_ticks(timer, timeout)`



TIME MANAGEMENT

- **One-Shot** timers with **initial** delay

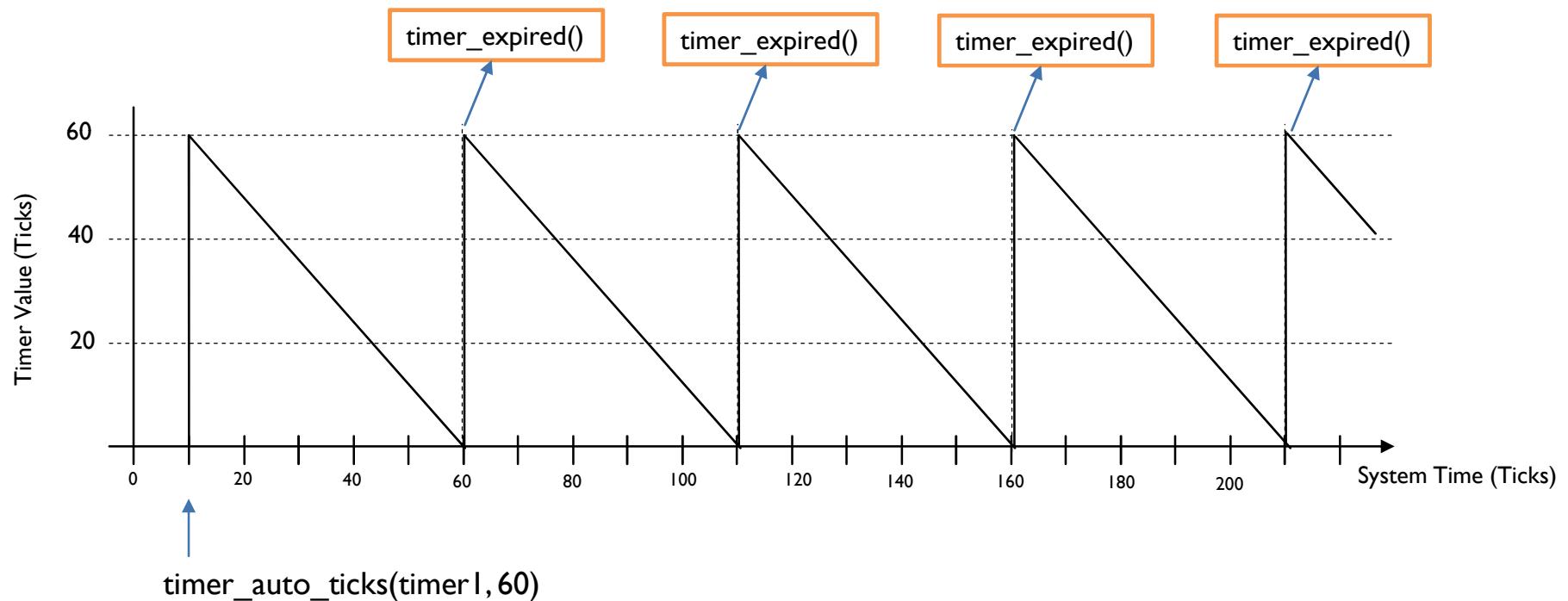
`timer_load_delay_ticks(timer, initialDelay, timeout)`



TIME MANAGEMENT

- **Auto-reload timers**

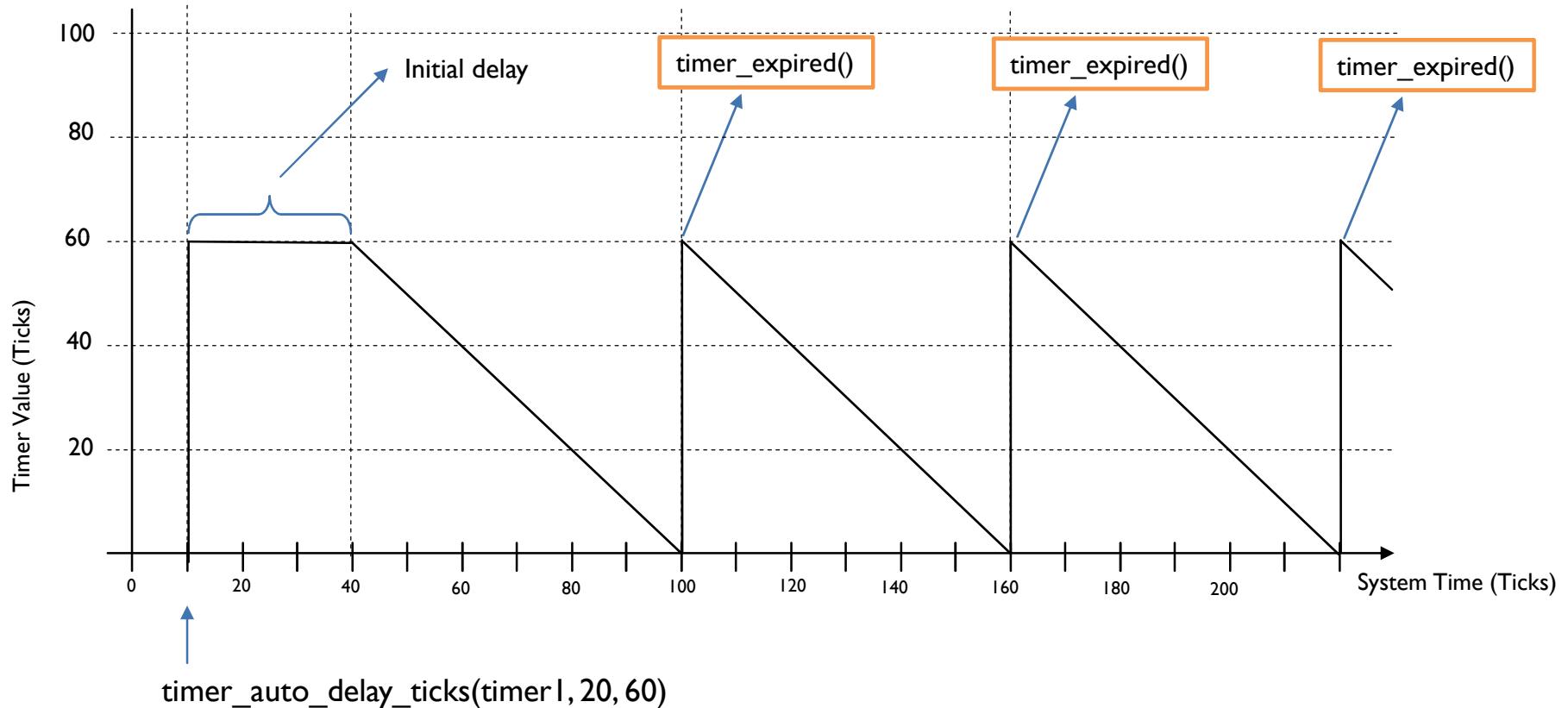
`timer_auto_ticks(timer, period)`



TIME MANAGEMENT

- **Auto-reload** timers with initial delay

```
timer_auto_delay_ticks(timer, initialDelay, period)
```



TIME MANAGEMENT

- Normally timers can be:
 - Initialized
 - Started (or re-started)
 - Paused
 - Resumed
 - Canceled
- When timers **expire**
 - A **callback** may get called
 - An **event** can be activated within such callback
 - A **flag** can be set when timer expires (to be consumed by polling)

TIME MANAGEMENT

- Some RTOS use the same basis of auto-reload timers for activating periodic tasks:
 - E.g., Alarms concept in OSEK.

TIME MANAGEMENT

- The RTOS implements the event groups by means of **Event Control Blocks** (ECB)
- ECBs contain:
 - Event Flags
 - Control Information
 - Timestamp of when a given event occurred, etc..



TIME MANAGEMENT

Time Management Task

TIME MANAGEMENT

- When a Task is “waiting” for a given time to expire it is removed from the **ready list** and changed to a blocked state.
- Normally RTOS provide timing services by means of a Time Management Task (**TMT**):
 - The Task is based on the clock tick.
 - Within the TMT, the “waiting time” for the tasks are updated in order to notice when they have expired and the associated tasks shall become *ready*.
 - Within the TMT, the values of the timers are updated (down counting).



TIME MANAGEMENT

- We will analyze the internal mechanism of the TMT for µC/OS III.
- Normally an RTOS will need to update hundreds of timers
 - Timers used to implement Tasks delays/timeouts.
 - Timers created by the application code.
- Need to update all those timers in an efficient way.

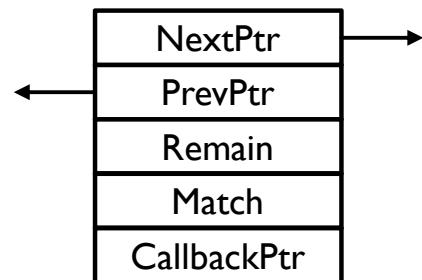
TIME MANAGEMENT

Timer Objects

- Data structure for each of the timers created.
- They contain information about:
 - Expiration time (**Match** Value)
 - Type of timer
 - Internal used for task delay/timeout
 - One-shot
 - Auto-reload, etc.
 - Pointer to the callback to be called upon time expiration
 - If the timer is used for task delay/timeout, the callback is indeed implemented by the RTOS and aimed to re-activate the waiting task.

TIME MANAGEMENT

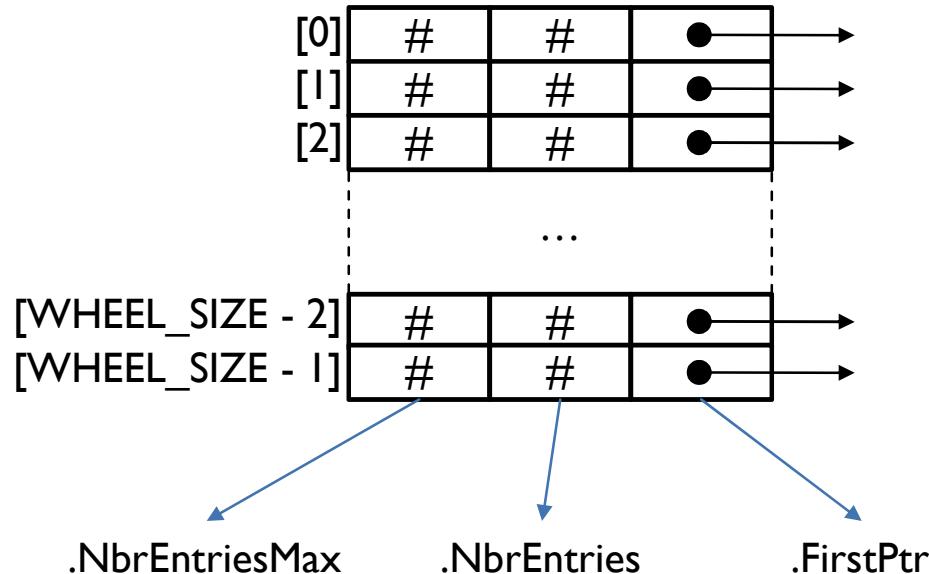
...Timer **Objects**...



TIME MANAGEMENT

Timer **Wheel**

- Data structure used to **optimize** the handling of several timer objects by the **TMT**.
 - Decrementing every timer on each occurrence of the TMT is not efficient.
- Is an array with following structure:



TIME MANAGEMENT

...Timer **Wheel**...

- **WHEEL_SIZE** is a statically configurable value
 - Recommendations → A prime number, not a multiple of the TMT frequency.
- **.FirstPtr** → Pointer to the first Time Object if any (singly linked list).
- **.NbrEntries** → Number of elements in the Time Object List
- **.NbrEntriesMax** → Max number of elements ever in the Time Object List.
- Together with the Time Wheel, the TMT also needs a global timer tick counter (**tmrTickCntr**) which will tell the current system time (in ticks of the TMT).

TIME MANAGEMENT

...Timer **Wheel**...

- APIs for starting a timer service will look like:

```
tmrStart(timer, delay)
```

- timer → the timer to be started
- delay → the amount of time to be loaded in the timer

TIME MANAGEMENT

...Timer **Wheel**...

- Timer Objects are inserted in the Timer Wheel according to following equations:
 - Match Value (**Match**): the time value (in ticks) when the timer its considered expired. When this time is reached, the callback gets called.

$$\mathbf{Match} = \mathbf{tmrTickCntr} + \mathbf{delay}$$

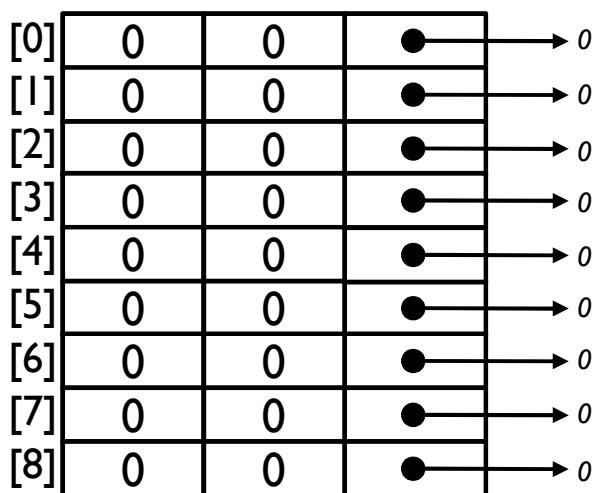
- Index (**Idx**) in the Timer Wheel where the Time Object is to be queued.

$$\mathbf{Idx} = \mathbf{Match \% WHEEL_SIZE}$$

TIME MANAGEMENT

Timer **Wheel** example:

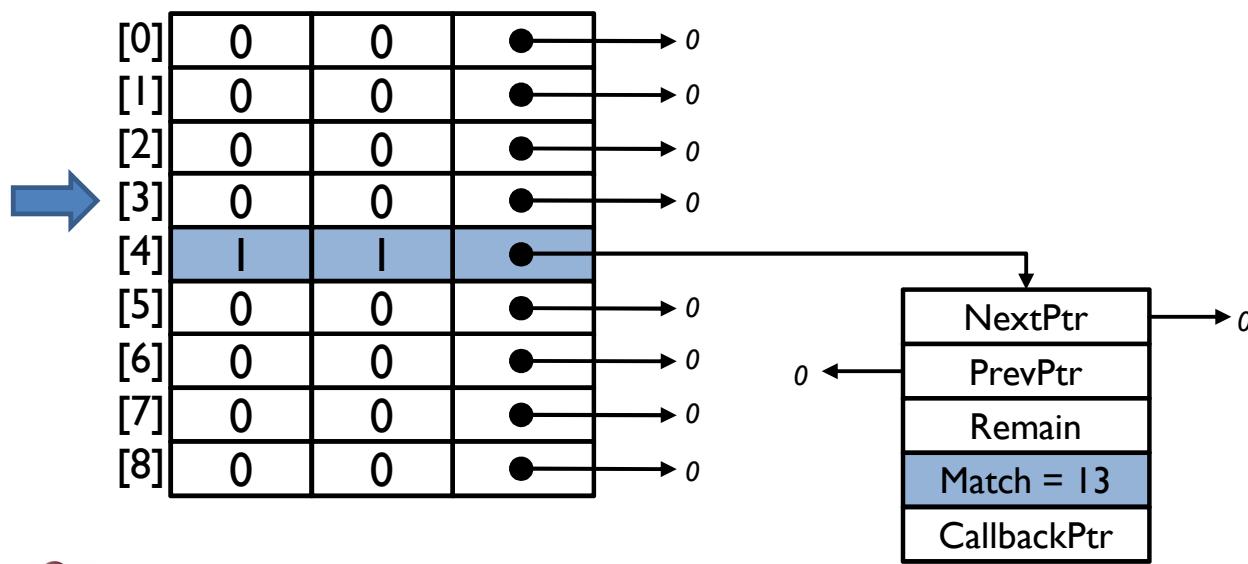
- WHEEL_SIZE = 9



TIME MANAGEMENT

Timer **Wheel** example:

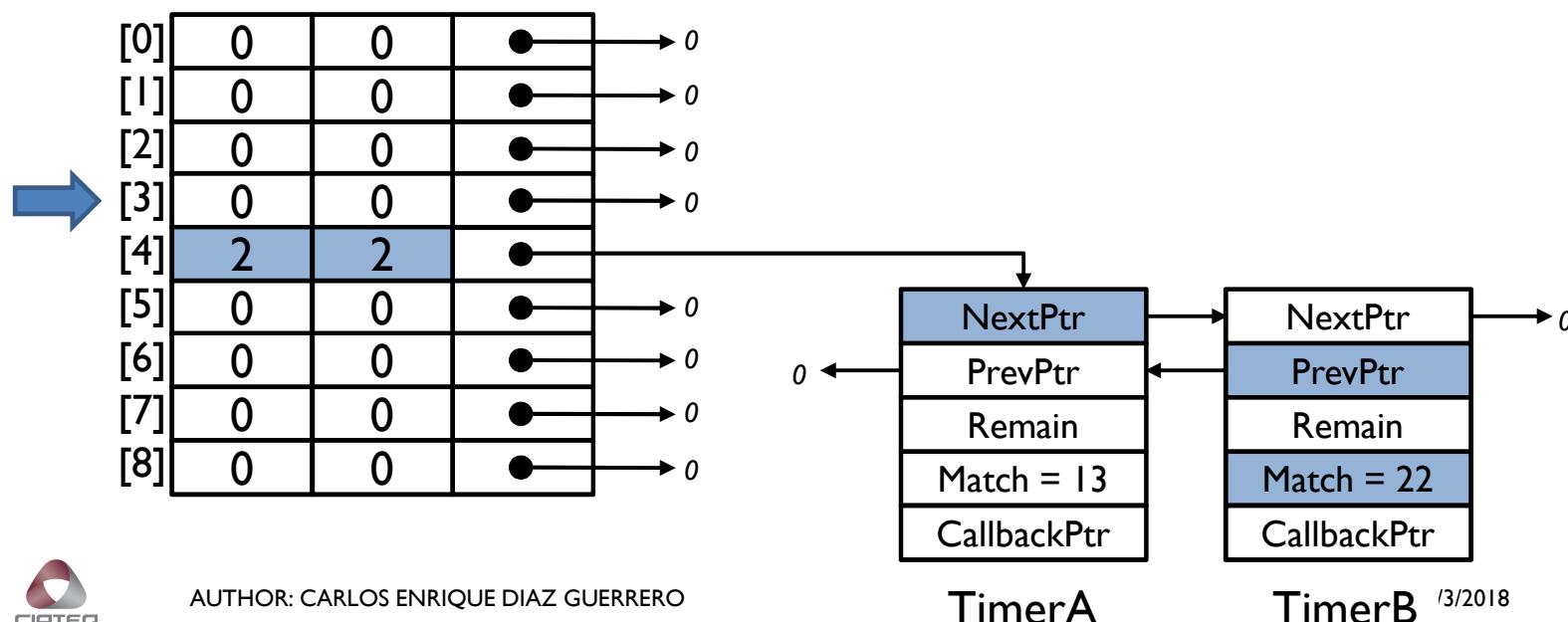
- Current time is 12 \rightarrow tmrTickCntr = 12
- Starting a timer with delay = 1 \rightarrow tmrStart(TimerA, 1);
 - Match = tmrTickCntr + delay = 12 + 1 = 13
 - Idx = Match % WHEEL_SIZE = 13 % 9 = 4



TIME MANAGEMENT

...Timer **Wheel** example...

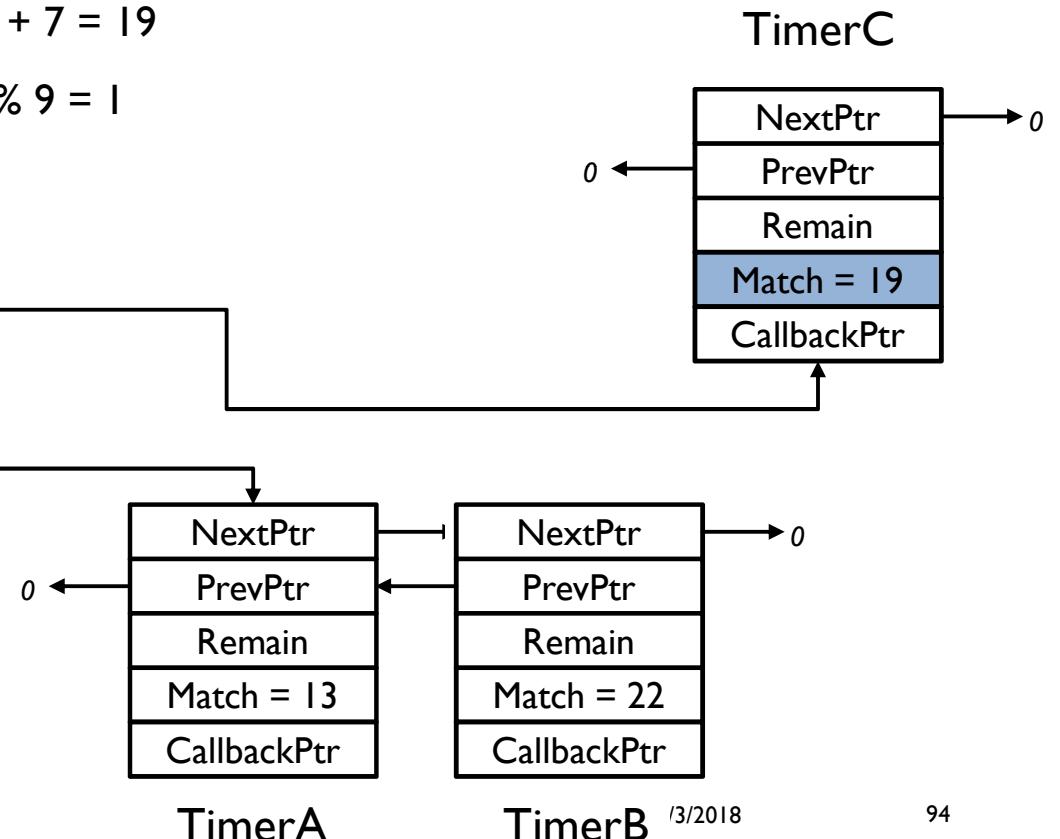
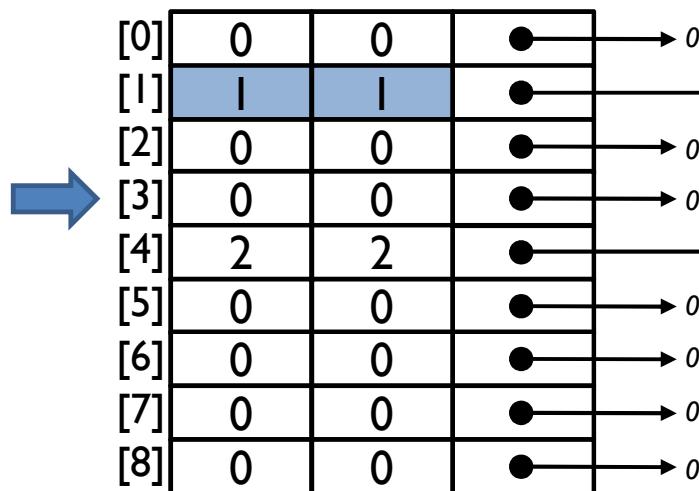
- Current time is 12 \rightarrow tmrTickCntr = 12
- Starting another timer with delay = 10 \rightarrow tmrStart(TimerB, 10);
 - Match = tmrTickCntr + delay = 12 + 10 = 22
 - Idx = Match % WHEEL_SIZE = 22 % 9 = 4



TIME MANAGEMENT

...Timer **Wheel** example...

- Current time is 12 \rightarrow tmrTickCntr = 12
- Starting yet another timer with delay = 7 \rightarrow tmrStart(TimerC, 7);
 - Match = tmrTickCntr + delay = 12 + 7 = 19
 - Idx = Match % WHEEL_SIZE = 19 % 9 = 1



TIME MANAGEMENT

...Timer **Wheel** example...

- The timer objects are always inserted in the list **sorted** by increasing Match value.
 - The timer object with the smaller Match value will always be at the “front” of the list.
- Every tick of the TMT, the tmrTickCntr gets increased by one.
- The index to be analyzed within the time wheel (currTimIdx) is determined by:

$$\text{currTimIdx} = \text{tmrTickCntr \% WHEEL_SIZE}$$

TIME MANAGEMENT

...Timer **Wheel** example...

- Every TMT tick, **currTimIdx** is analyzed:
 - If **NbrEntries** is greater than zero then the Time Object List is navigated.
 - The navigation ends immediately when finding the first Time Object whose Match value is different than tmrTickCntr.
 - Because of this, the navigation should not take much time!
- Ideally, the Time Objects will be uniformly distributed across the Time Wheel

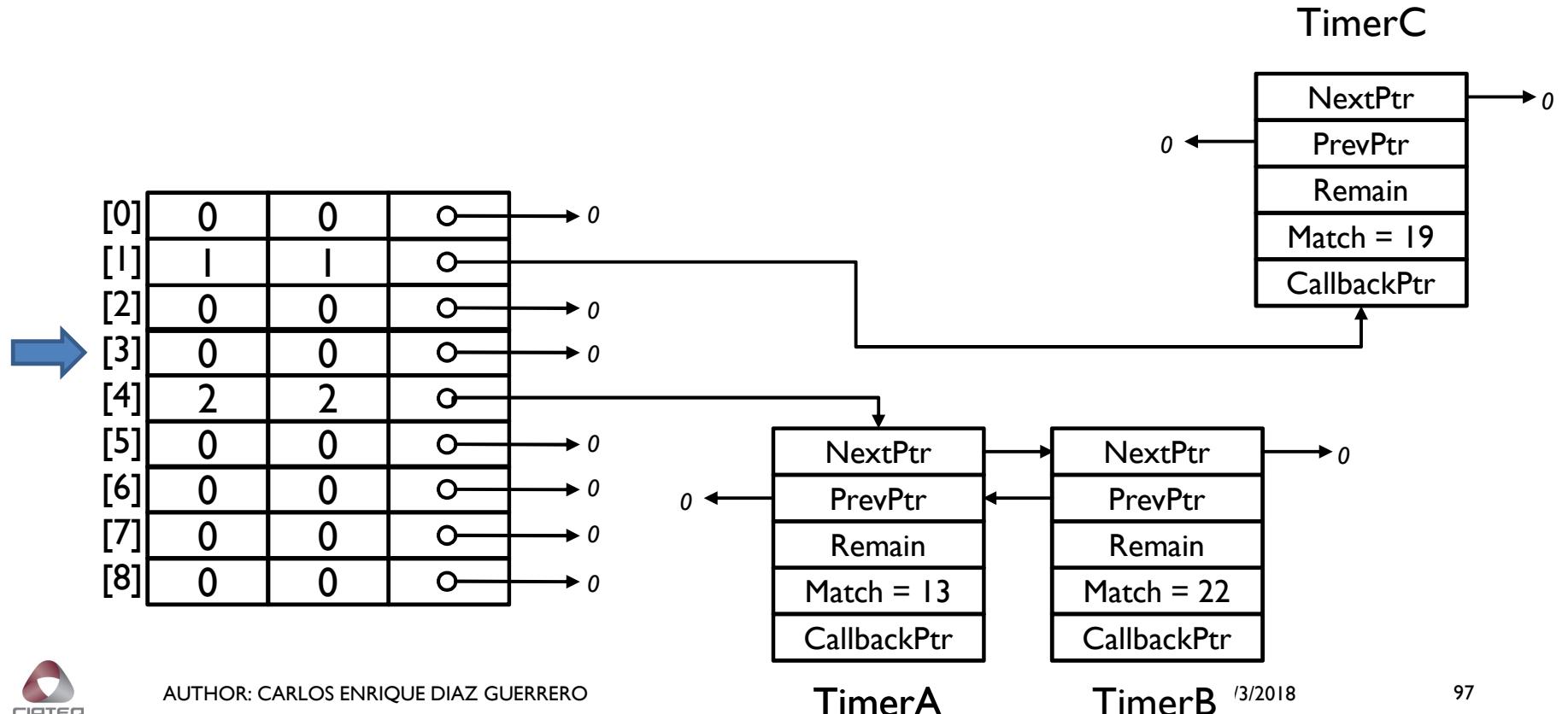
TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...

tmrTickCntr = 12

currTimIdx = 3



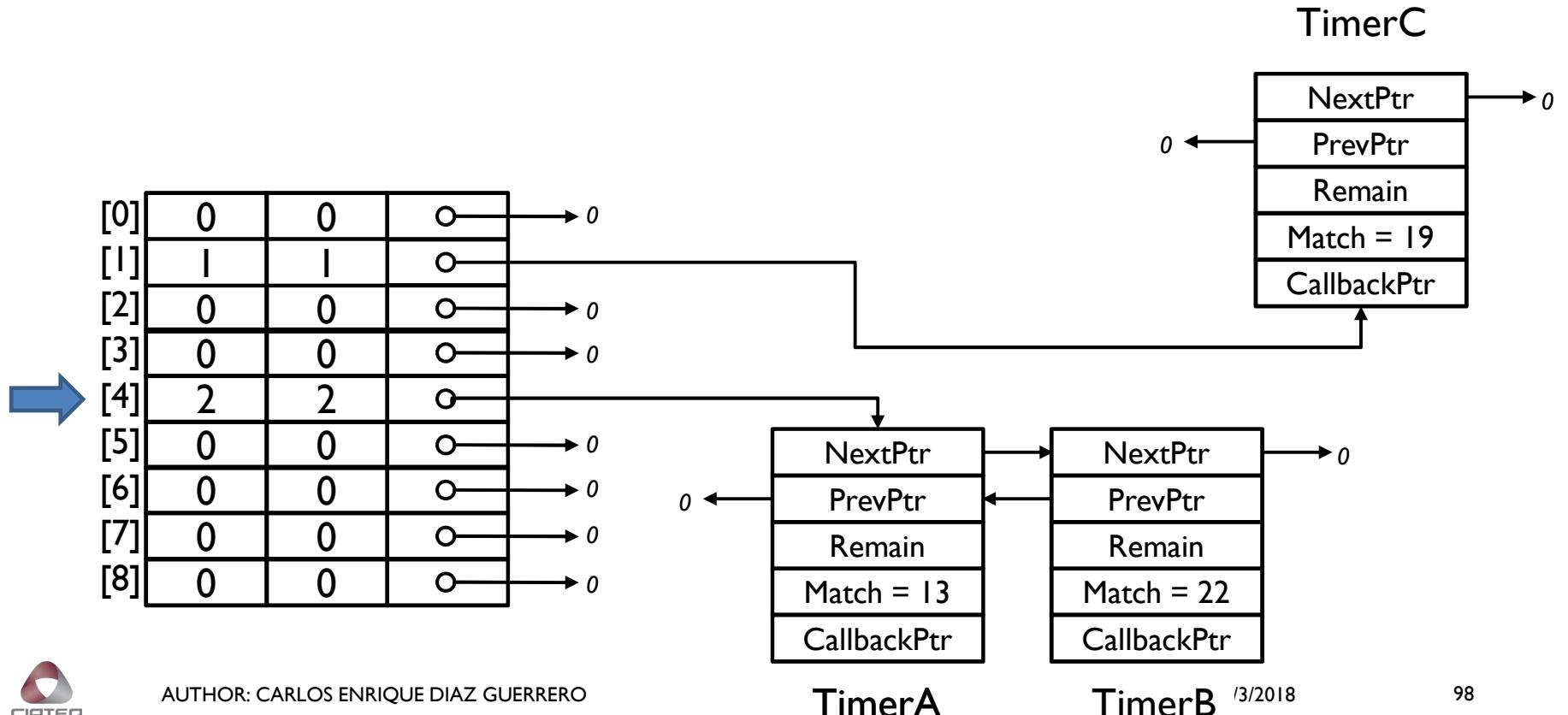
TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...

tmrTickCntr = 13

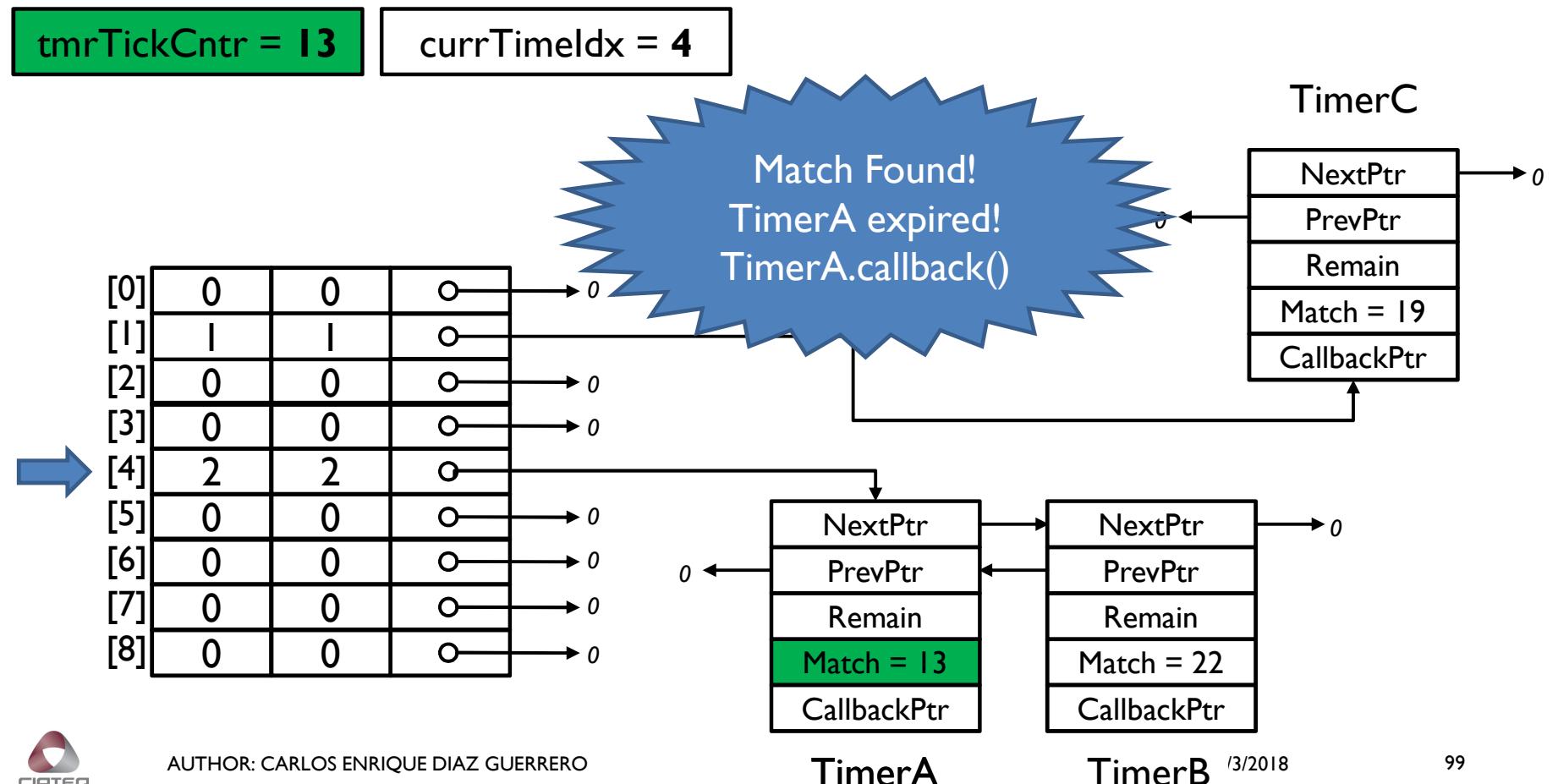
currTimIdx = 4



TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...



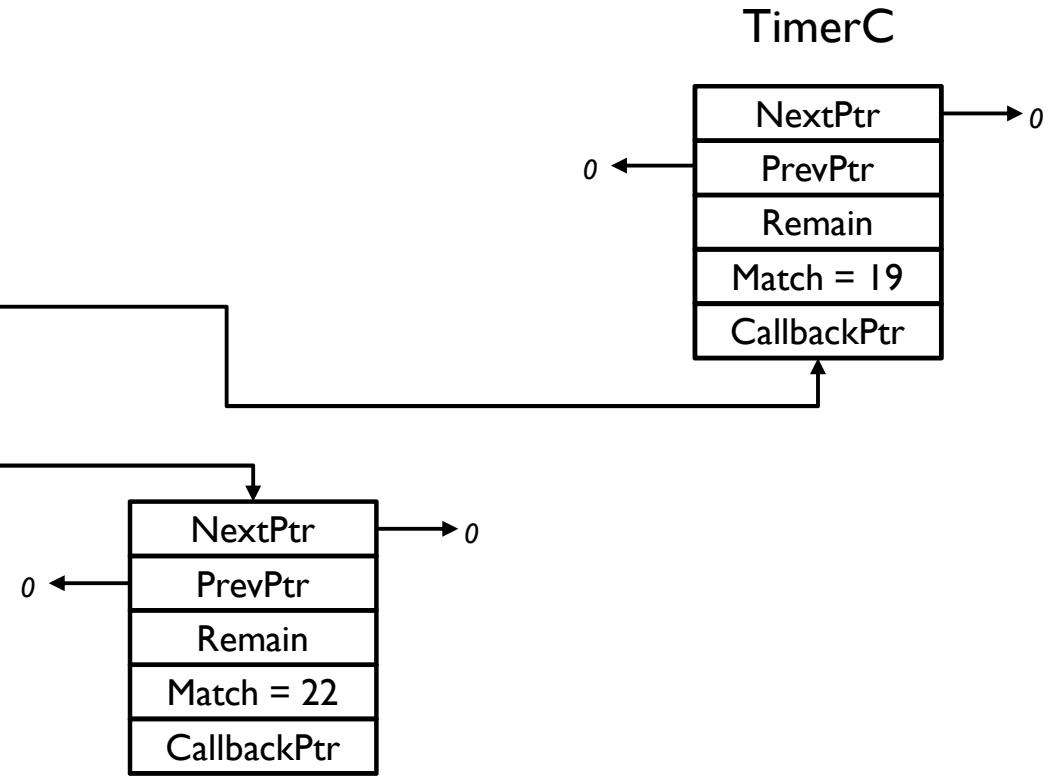
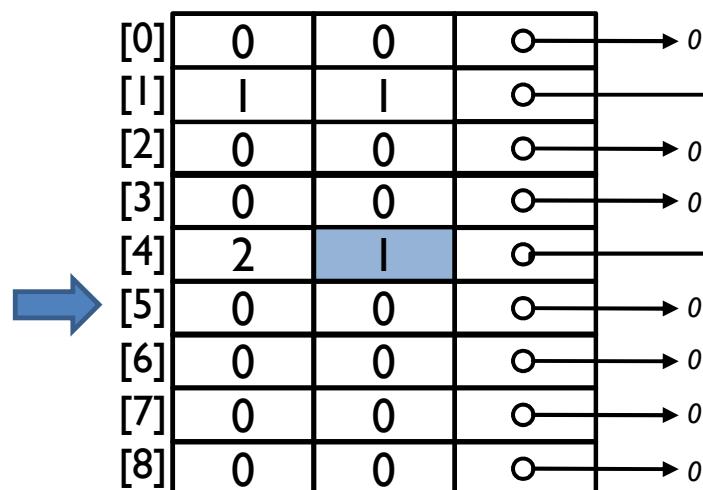
TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...

tmrTickCntr = 14

currTimIdx = 5



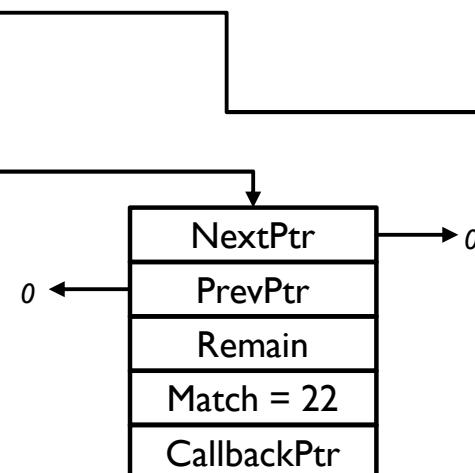
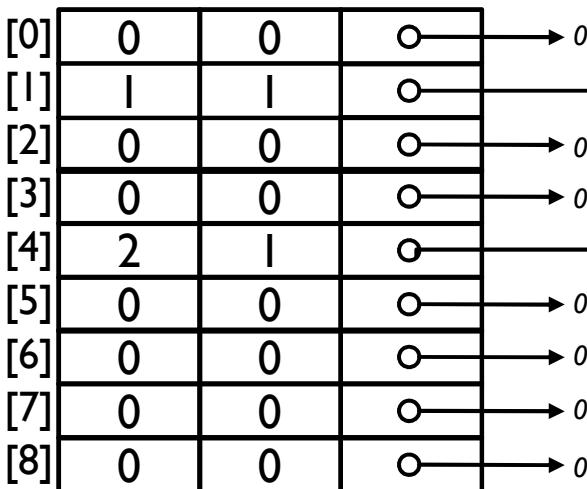
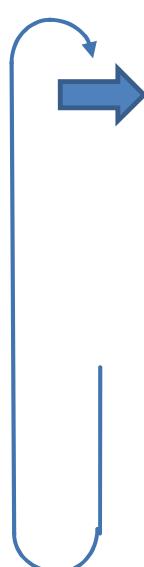
TIME MANAGEMENT

...Timer **Wheel** example...

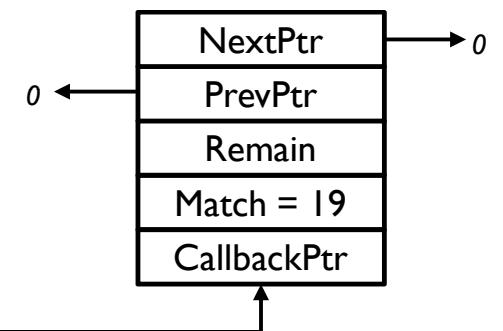
- Lets advance tmrTickCntr...

tmrTickCntr = 18

currTimIdx = 0



TimerC

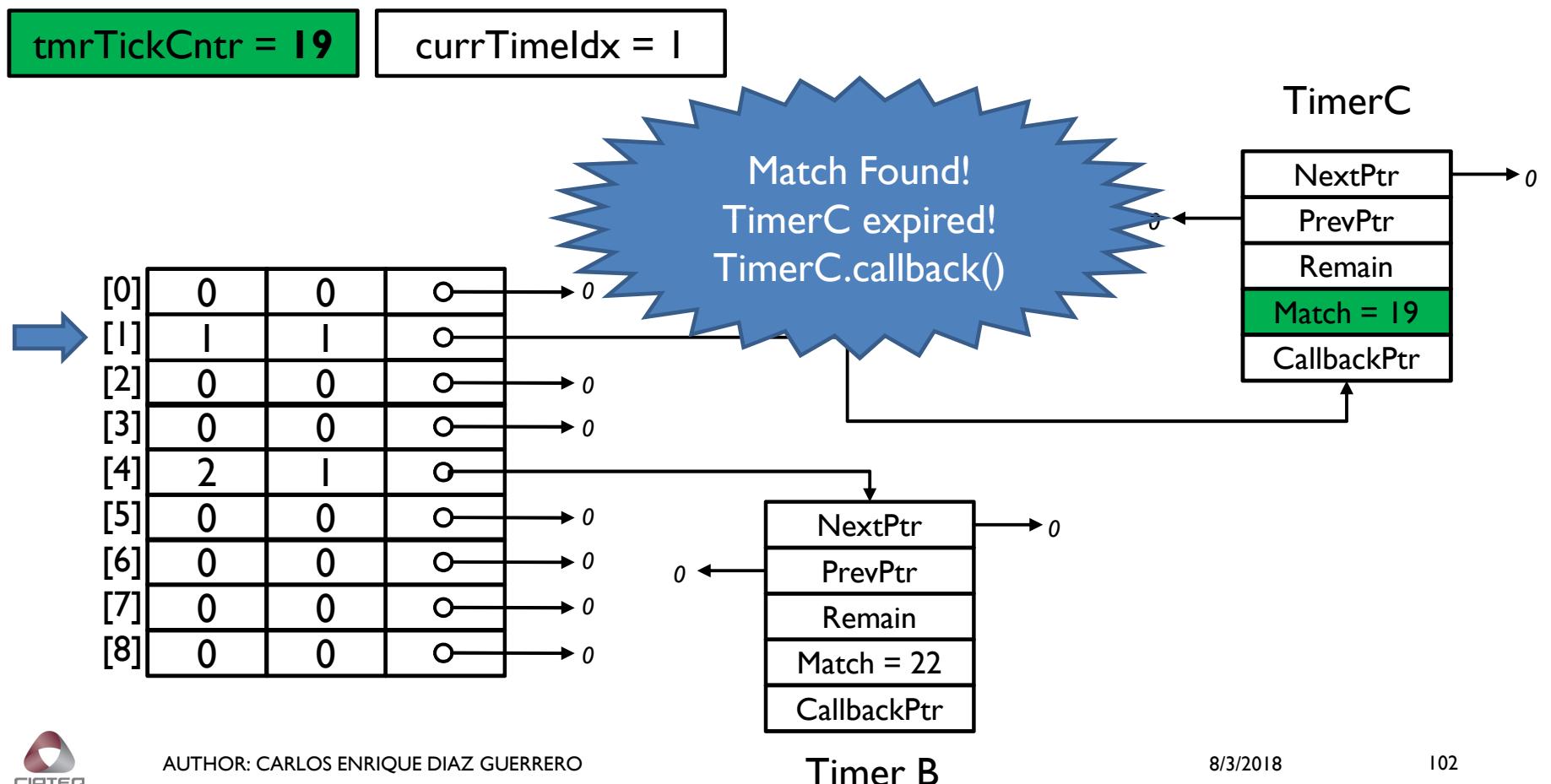


Timer B

TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...

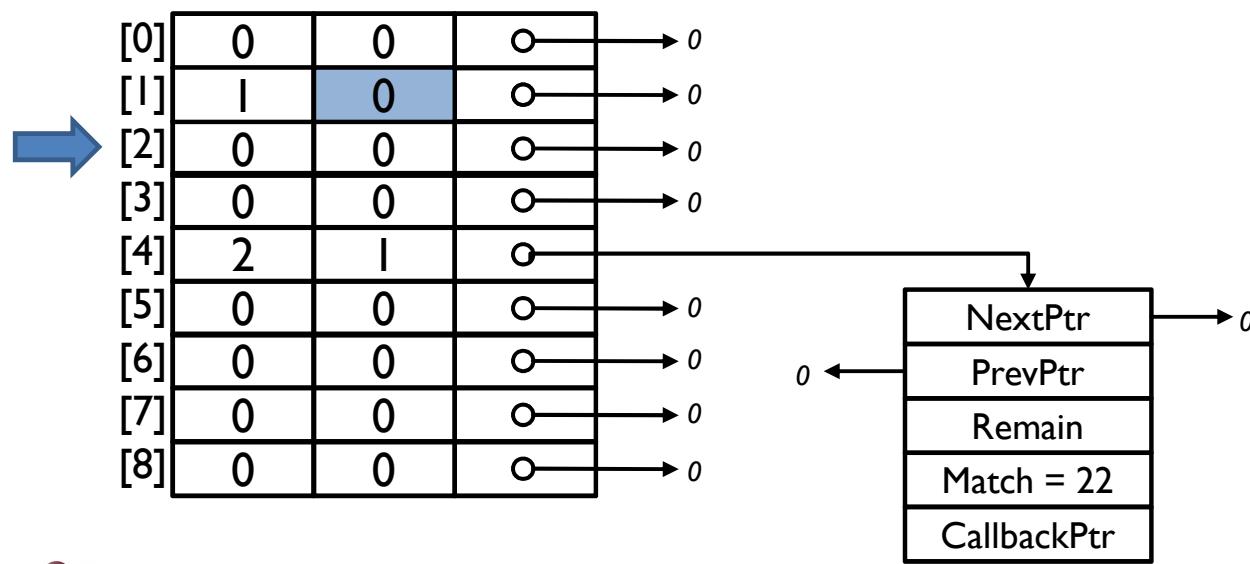


TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...

tmrTickCntr = 20 currTimIdx = 2

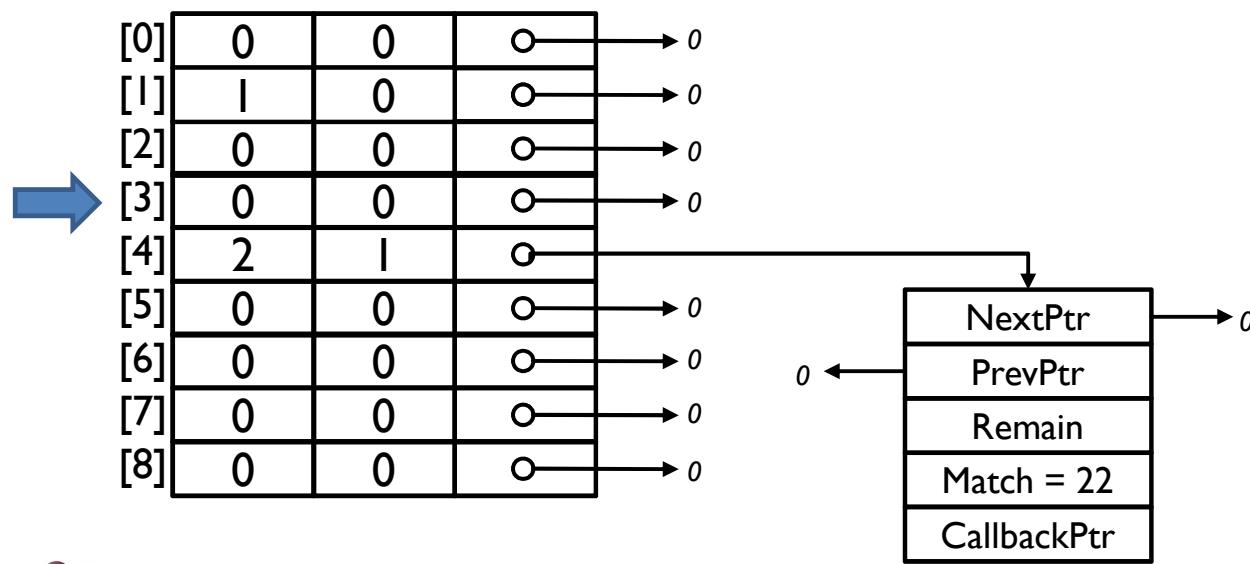


TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...

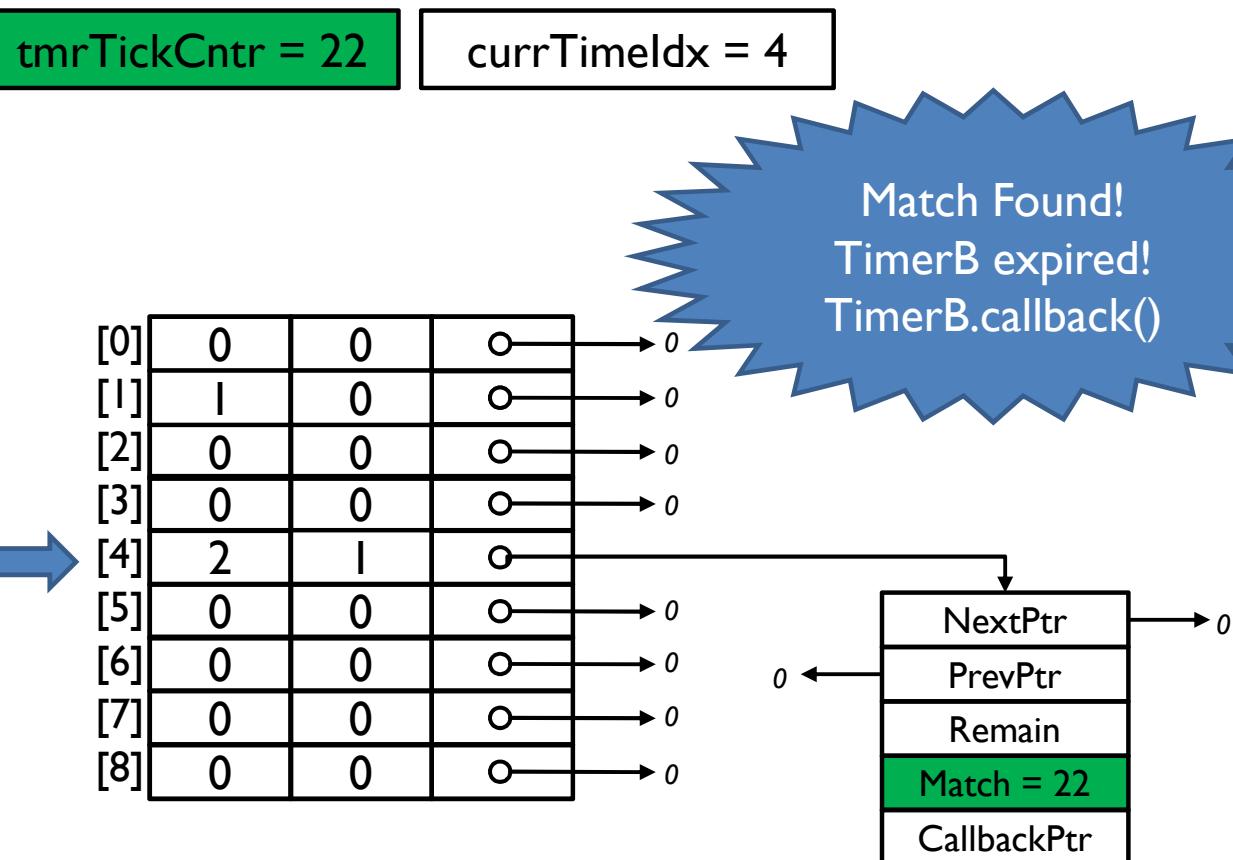
tmrTickCntr = 21 currTimIdx = 3



TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...



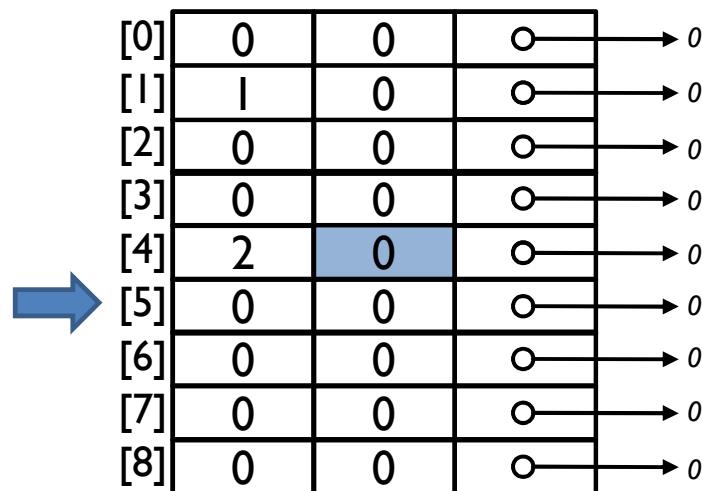
TIME MANAGEMENT

...Timer **Wheel** example...

- Lets advance tmrTickCntr...

tmrTickCntr = 23

currTimeldx = 5



OVERVIEW

- Real-Time Systems Concepts
- Real-Time Operating Systems Concepts
- Kernel Structure and Task Management
- Time Management
- **Semaphores and Mutual Exclusion**
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- References



SEMAPHORES AND MUTUAL EXCLUSION

Sharing Resources and Task Synchronization

SEMAPHORES AND MUTUAL EXCLUSION

- Recall from previous sections...
- Multiple tasks can **share**:
 - Code** → re-entrant functions
 - Data** → variables, ...
 - Resources** → I/Os, HW peripherals, etc.
- Multiple tasks may need to **synchronize** with each other
- Multiple tasks may need to **communicate** with each other

SEMAPHORES AND MUTUAL EXCLUSION

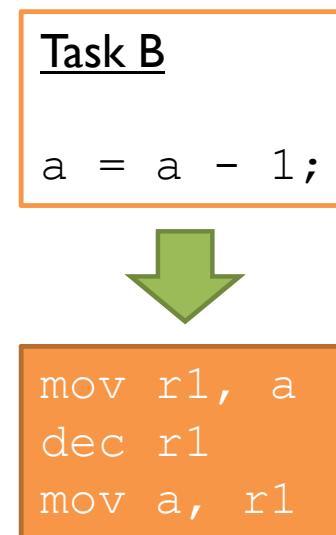
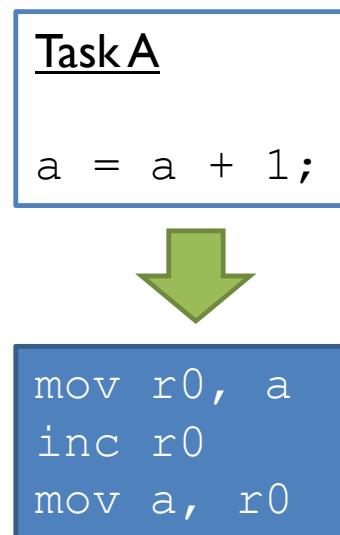
- We have already discussed the data corruption problem.
- We have already presented the critical section as a solution for such problem.
 - Disabling interrupts have been presented as a way of creating critical sections
- In this chapter we will see two useful artifacts that help us to:
 - Deal with **resource sharing** problems → an alternative to disabling interrupts for creating critical sections.
 - Deal with task **synchronization**
- Such artifacts are **mutual exclusion** and **semaphores**.

SEMAPHORES AND MUTUAL EXCLUSION

- When tasks have close interactions with each other, often:
 - They need to **share** resources
 - Probability of data corruption
 - **Race conditions**
 - They need to **coordinate** the access to such resources
 - Order of “operations” needs to follow certain sequence

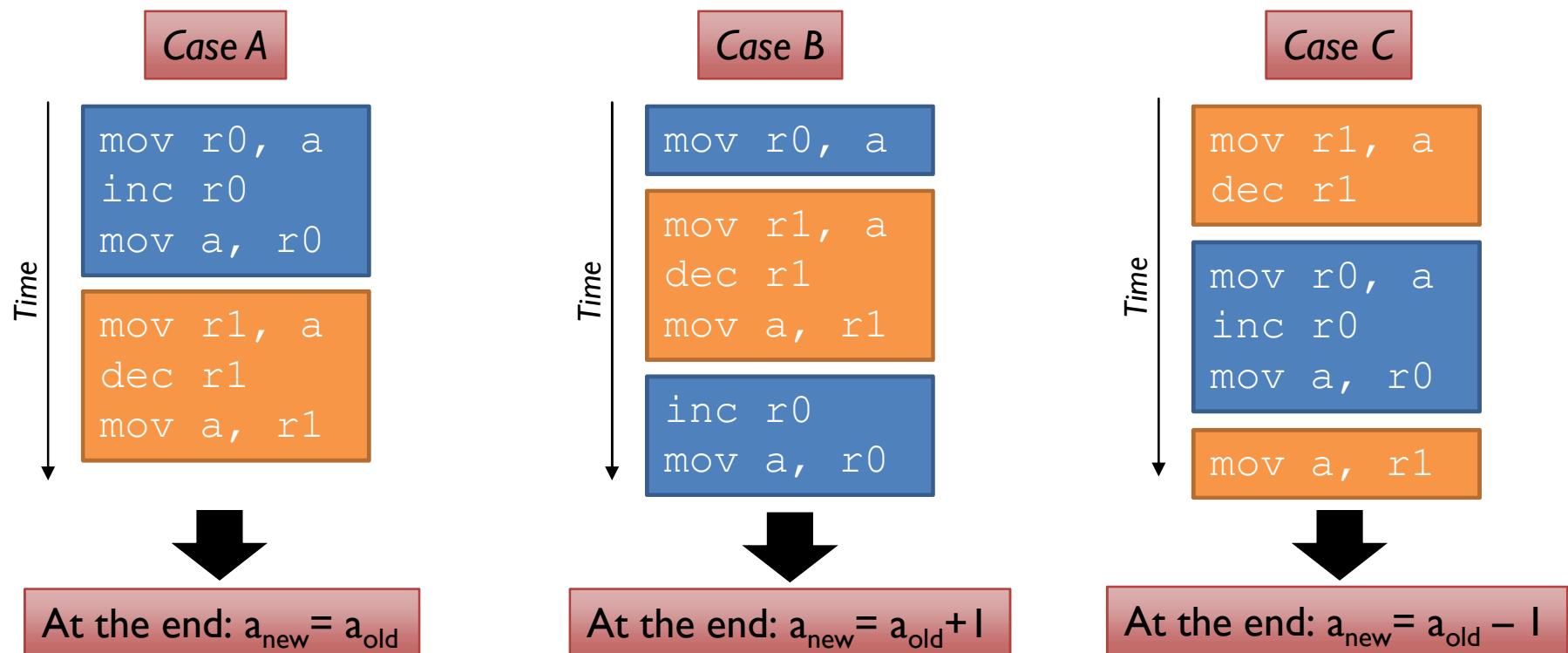
SEMAPHORES AND MUTUAL EXCLUSION

- **Race Condition:** a problematic situation when multiple tasks read and write to a shared data item and the final result depends on the **order of execution** of the instructions in the tasks.
- **Race Condition** example:



SEMAPHORES AND MUTUAL EXCLUSION

- ...**Race Condition** example...
- “Concurrent” execution of Tasks A and B: they are ‘racing’ for the access to variable **a**.
 - a. The order of the instructions execution indeed determines the final value of **a**.



SEMAPHORES AND MUTUAL EXCLUSION

- In the case of “competing” tasks, some problems need to be faced:
 - **Mutual Exclusion**
 - Deadlocks / Livelocks
 - Starvation
 - **Synchronization**

SEMAPHORES AND MUTUAL EXCLUSION

A. Mutual Exclusion

- Prevents Simultaneous access to shared resources
- **Goal** → Assure consistency of data

B. Tasks Synchronization

- Task shall wait for certain condition(s) to occur in order to continue execution
- **Goal** → Define an order of operations



SEMAPHORES AND MUTUAL EXCLUSION

The Mutual Exclusion Problem

SEMAPHORES AND MUTUAL EXCLUSION

■ Mutual Exclusion

- The requirement that when one task is in a **critical section** that accesses a shared resource (**critical resource**), no other task can be in a critical section that accesses such shared resource.
- When implementing mutual exclusion, new problems can be created:
 - Deadlocks / Livelocks
 - Starvation

SEMAPHORES AND MUTUAL EXCLUSION

■ Deadlock

- A situation in which two or more tasks are unable to proceed because each is waiting for one of the others to do something.
- Example:
 - Two Tasks A and B are the only two tasks sharing a resource R.
 - At a given time Task A is waiting for Task B to release the resource R
 - At the same time Task B is inversely waiting for Task A to release R
 - None of the Task can indeed release resource R
 - Both task will be waiting endlessly for each other.

SEMAPHORES AND MUTUAL EXCLUSION

■ Livelock

- A situation in which two or more tasks are continuously changing their states in response to other task's actions without doing any useful work.
- Known also as a “mutual courtesy”
- Example:
 - Two people attempting to pass each other in a corridor
 - Person A moves to his left to let person B pass, while B moves to his right to let A pass.
 - Seeing that they are still blocking each other, A moves to his right, while B moves to his left.
 - They're still blocking each other.
 - This may go forever, each person is given “courtesy” to the other one but none can pass.

SEMAPHORES AND MUTUAL EXCLUSION

■ Starvation

- A situation in which a ready task is *overlooked* indefinitely by the scheduler; although it is able to proceed, it is never chosen for execution.
- Example:
 - Three Tasks A, B and C require periodic access to a shared resource R.
 - Task A gets the resource and accesses it.
 - Then, the scheduler may decide that tasks B or C can get it;
 - Task B is chosen by the scheduler.
 - Before task B releases R, task A requests again the resource.
 - When task B releases R, options for the scheduler are task A or task C. Task A is chosen to get it.
 - This situation continues endlessly alternating the access to R between tasks A and B but task C indeed never gets access to the resource!
 - Task C “dies” of starvation.

SEMAPHORES AND MUTUAL EXCLUSION

- A solution to the **mutual exclusion** problem shall:
 - Ensure the **mutual exclusion**
 - Ensure execution **progress** (liveness)
 - If no task is in its critical section and another task wish to enter its critical section, then the decision of which task can enter its critical section next cannot be delayed infinitely
 - Free of deadlocks/livelocks
 - Ensure **bounded waiting** (fairness)
 - There is a time limit on how long a task may be forced to wait for entering its critical section.
 - Free of starvation

SEMAPHORES AND MUTUAL EXCLUSION

- Types of solutions to the **mutual exclusion** problem:
 - **Software** solutions
 - Assumption: atomicity of read or write operation in memory
 - **Hardware** supported solutions
 - Require special MCU instructions
 - Higher **language** “**constructs**”
 - Functions and data structures provided to the programmer

SEMAPHORES AND MUTUAL EXCLUSION

- **Software** Solutions:
 - Dekker-Algorithm
 - Peterson-Algorithm
- **Hardware** supported solutions
 - Disable/Enable interrupts
 - “Compare and Swap” instruction
 - “Test and Set” instruction
- Higher language “constructs”
 - Semaphores
 - Monitors
 - Message passing, etc..

SEMAPHORES AND MUTUAL EXCLUSION

- **Software** Solutions:
 - Dekker-Algorithm
 - Peterson-Algorithm
- **Hardware** supported solutions
 - Disable/Enable interrupts
 - “Compare and Swap” instruction
 - “Test and Set” instruction
- Higher language “constructs”
 - **Semaphores**
 - Monitors
 - Message passing, etc..





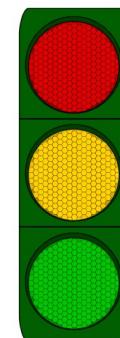
SEMAPHORES AND MUTUAL EXCLUSION

Semaphores for Mutual Exclusion

SEMAPHORES AND MUTUAL EXCLUSION

■ **Semaphore**

- In real life → a system of signals used to visually coordinate traffic (streets, rails, etc. are shared resources).
- In computer science → Data structure for solving data sharing and synchronization problems.
 - Invented by Edsger Dijkstra a Dutch computer scientist.



SEMAPHORES AND MUTUAL EXCLUSION

- A **Semaphore** is like an integer but with following differences:
 - **Semaphore initialization**
 - When creating the semaphore, its value can be **initialized** to any integer.
 - After initialization the only allowed operations are **increment** (increase by one) and **decrement** (decrease by one).
 - It should not be possible to read the current value of the semaphore.
 - When a task **decrements** the semaphore, if the result is negative, the task blocks itself and cannot continue until another task increments the semaphore.
 - When a task **increments** the semaphore, if there are other tasks waiting, one of the waiting tasks gets unblocked.

SEMAPHORES AND MUTUAL EXCLUSION

- Semaphores are data structures normally provided by the RTOS.
- The Semaphore **Data Structure** normally contains:
 - Its **integer** value
 - A **queue** of tasks waiting for such semaphore.
 - Queue is normally implemented using linked lists.

```
typedef OS_SEM{  
    int value;  
    OS_PEND_LIST queue;  
};
```

SEMAPHORES AND MUTUAL EXCLUSION

- The implementation of semaphores within an RTOS normally implies having a **Semaphore Control Block**
 - Will contain the semaphore data structure
 - Value
 - Waiting queue
 - Other control information for the semaphore:
 - Timestamp
 - Task “owning” the semaphore in case of a binary semaphore, etc.

SEMAPHORES AND MUTUAL EXCLUSION

- Valid operations over a **Semaphore**:
 - Semaphore **initialization**
 - **Wait** on the semaphore
 - **Signal** the semaphore

SEMAPHORES AND MUTUAL EXCLUSION

- **Semaphore initialization**

- Semaphores should be initialized to a **non-negative** integer value (0,1,2,...)
- Pseudo code (S = semaphore to operate with, val = initial value):

```
init(S, val):  
    S.value = val;  
    queue = empty list;
```

SEMAPHORES AND MUTUAL EXCLUSION

- **Wait** on semaphore

- This operation **decrements** the semaphore and then checks whether the calling task is allowed to continue or not.
- If the task is not allowed to continue, its state is changed to “blocked” and the task is “appended” into the waiting queue of the semaphore.

```
wait(S) :
```

```
    S.value = S.value - 1;  
    if S.value < 0  
    then add the calling task to S.queue  
          and block it;
```

- This operation is also referred as P(S) or pend(S)
- P comes from dutch word “Proberen” that means “test”.

SEMAPHORES AND MUTUAL EXCLUSION

■ **Signal** the semaphore

- This operation **increments** the semaphore and then checks whether a “waiting” task shall become **ready** for execution.
- If a waiting task shall become ready then the task is removed from the waiting queue of the semaphore and its state is set to “ready”.

```
signal(S) :
```

```
    S.value = S.value + 1;  
    if S.value <= 0  
        then remove a task from S.queue and place it  
             in the RTOS ready queue;
```

- This operation is also referred as $V(S)$ or $post(S)$
- V comes from dutch word “Verhogen” that means “increment”.

SEMAPHORES AND MUTUAL EXCLUSION

- **Remarks** on semaphore operations:
 - All semaphore operations shall be executed **atomically**
 - $S.value \geq 0$: number of tasks which can execute the *wait* operation in sequence without actually having to wait.
 - When initializing $S.value$ with a value of n : Then n tasks can enter their critical sections at the same time
 - $S.value < 0$: The absolute value $|S.value|$ tells how many tasks are blocked in the queue of S

SEMAPHORES AND MUTUAL EXCLUSION

- ...**Remarks** on semaphore operations...
 - The order on which of the waiting Tasks becomes active by the “signal” operation is not standardized.
 - If FIFO order is used then the semaphore is known as a **strong semaphore**
 - If no order is defined then the semaphore is known as a **weak semaphore**.

SEMAPHORES AND MUTUAL EXCLUSION

- The previously discussed semaphore is known as “**counting semaphore**” or “general semaphore”.
- There is another type of semaphore known as “**binary semaphore**”
- **Types** of semaphores:
 - **Binary** semaphore: its value can be **only** 0 or 1.
 - **Counting** semaphore: can have arbitrary integer values.
- The operations of a binary semaphore **slightly change** as compared to those of the general semaphore.

SEMAPHORES AND MUTUAL EXCLUSION

- **Binary Semaphore initialization**

- Binary semaphores can be initialized **only** as 0 or 1
- Pseudo code (S = semaphore to operate with, val = initial value):

```
init(S, val):  
    S.value = (val) ? 1 : 0;  
    queue = empty list;
```

SEMAPHORES AND MUTUAL EXCLUSION

- **Wait** on **binary** semaphore

- This operation first checks the current value of the semaphore.
 - If value is **zero**, then the task executing the operation gets blocked.
 - If the value is **one**, then the value is decremented to zero and the Task can continue execution.
- If the task is not allowed to continue, its state is changed to “blocked” and the task is “appended” into the waiting queue of the binary semaphore.

```
wait(S) :  
    if S.value == 1  
        then S.value = 0  
    else add the calling task to S.queue  
        and block it;
```

SEMAPHORES AND MUTUAL EXCLUSION

■ **Signal** the **binary** semaphore

- This operation first check the blocked queue of the semaphore.
 - If there are task in the semaphore's blocking queue, then one of them is unblocked.
 - If no task is blocked then the binary semaphore is **incremented to one**.
- If a waiting task becomes unblocked, then the task is removed from the waiting queue of the semaphore and its state is set to "ready".

```
signal(S) :  
    if S.queue == empty  
        then S.value = 1;  
    else remove a task from S.queue and place it  
        in the RTOS ready queue;
```

SEMAPHORES AND MUTUAL EXCLUSION

- Semaphores can be used for:
 - Resource sharing → mutual exclusion, critical sections
 - Tasks/ISRs synchronization

SEMAPHORES AND MUTUAL EXCLUSION

■ Mutual exclusion with semaphores

- Normally achieved with **Binary** semaphores (only **one** task can be inside its critical section at a time).
- Critical sections can be created with such binary semaphores

Semaphore initialization



init(S, I)

Critical section with
semaphores



Some non-critical code
wait(S)
critical section code
signal(S)
Some more non-critical code

SEMAPHORES AND MUTUAL EXCLUSION

- When semaphores are used for **mutually exclusive** resource sharing:
 - One **binary semaphore** is created for each shared resource
 - Each semaphore is initialized with 1 indicating that initially the resource is available.

```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
localVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
localVarA = a;
signal(semVarA)
/* do something with localVarA */
```

- In the example above, only one resource is shared (variable a) and hence one binary semaphore is created → semVarA

SEMAPHORES AND MUTUAL EXCLUSION

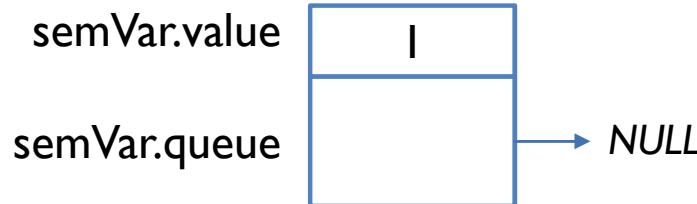
init(semVarA,1)

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

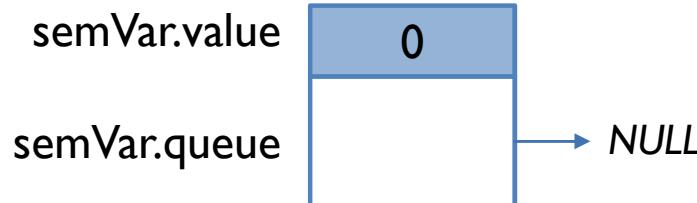
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

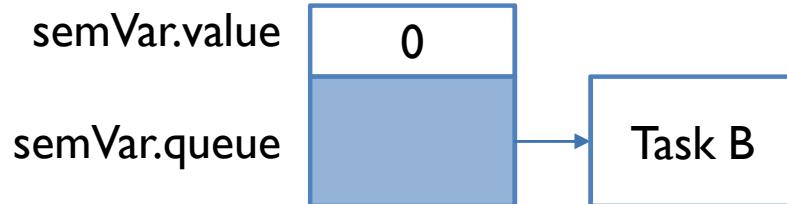
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

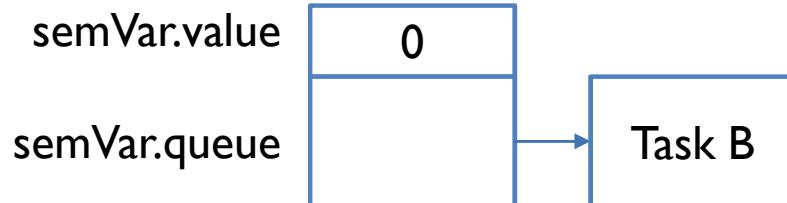
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

init(semVarA,1)

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

semVar.value

0

semVar.queue

→ NULL

SEMAPHORES AND MUTUAL EXCLUSION

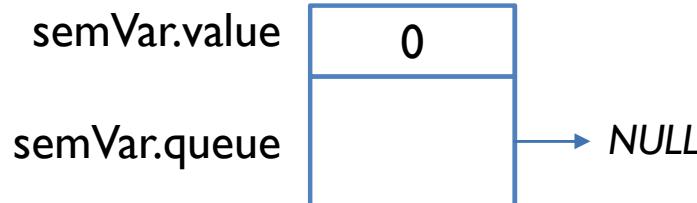
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

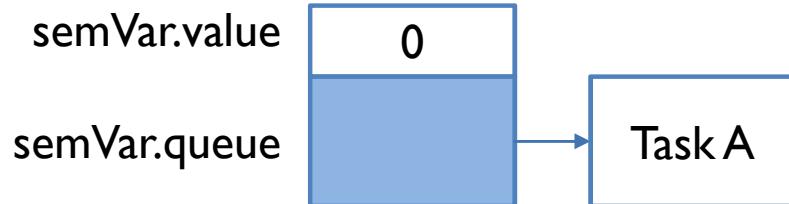
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

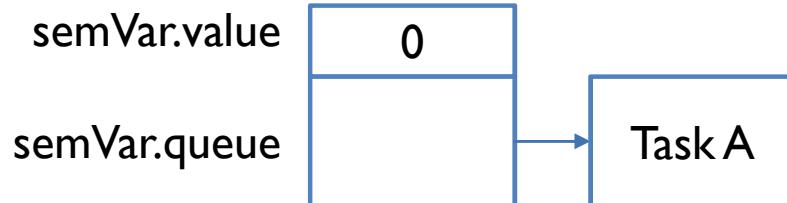
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

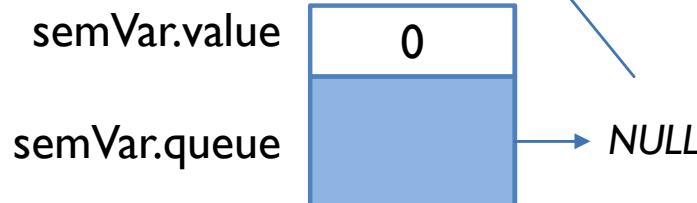
init(semVarA,1)

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

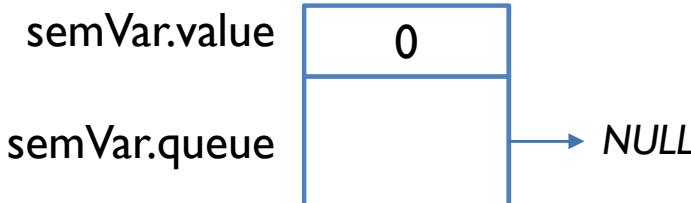
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

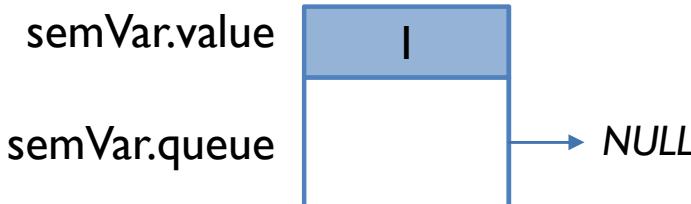
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

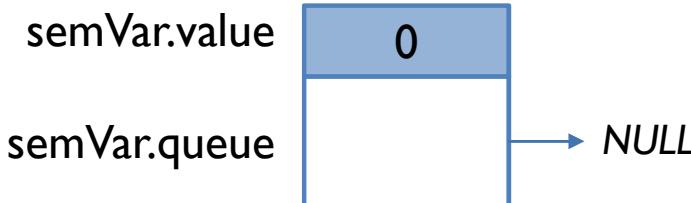
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

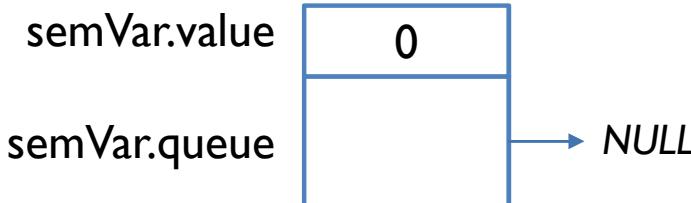
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

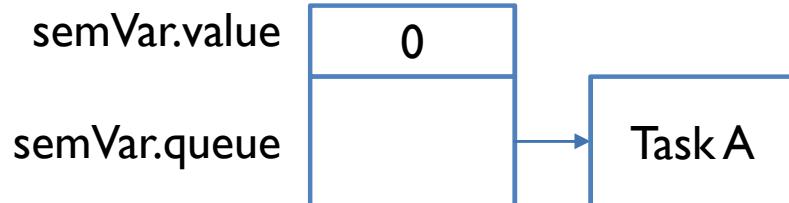
```
init(semVarA,1)
```

Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

```
init(semVarA,1)
```

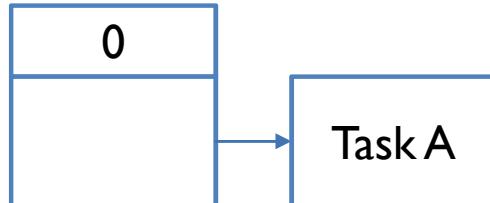
Task A

```
wait(semVarA)
a = a + 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

Task B

```
wait(semVarA)
a = a - 1;
LocalVarA = a;
signal(semVarA)
/* do something with localVarA */
```

semVar.value



semVar.queue

Analysis continues like this...

SEMAPHORES AND MUTUAL EXCLUSION

- The example can be expanded to any number of tasks sharing the same resource.

Task A

```
wait(semVarA)  
a = a + 1;  
LocalVarA = a;  
signal(semVarA)  
/* do something  
with localVarA */
```

Task B

```
wait(semVarA)  
a = a - 1;  
LocalVarA = a;  
signal(semVarA)  
/* do something  
with localVarA */
```

...

Task n

```
wait(semVarA)  
a = a >> 2;  
LocalVarA = a;  
signal(semVarA)  
/* do something  
with localVarA */
```



SEMAPHORES AND MUTUAL EXCLUSION

- Maximum **one** task can be in its critical section at a time
 - → Mutual Exclusion is achieved
- Order of tasks entering their critical sections is still undetermined.

SEMAPHORES AND MUTUAL EXCLUSION

- If a shared resource can be taken by more than 1 task at a time then a **counting** semaphore can be used instead of a binary one.
 - Counting semaphore shall be initialized to the number of allowed tasks sharing the same **resource** at the same time.
 - This situation is known as **multiplex**.
- **Multiplex** synchronization pattern
 - Generalize the **mutual exclusion** solution so that it allows multiple tasks to run in the critical section at the same time, but it enforces an upper limit on the number of concurrent tasks.
 - No more than n tasks can run in the critical section at the same time.



REAL-TIME OPERATING SYSTEMS

MAESTRÍA EN SISTEMAS INTELIGENTES MULTIMEDIA

INSTRUCTOR: CARLOS ENRIQUE DIAZ GUERRERO



AUTHOR: CARLOS ENRIQUE DIAZ GUERRERO

8/3/2018

1



SEMAPHORES AND MUTUAL EXCLUSION

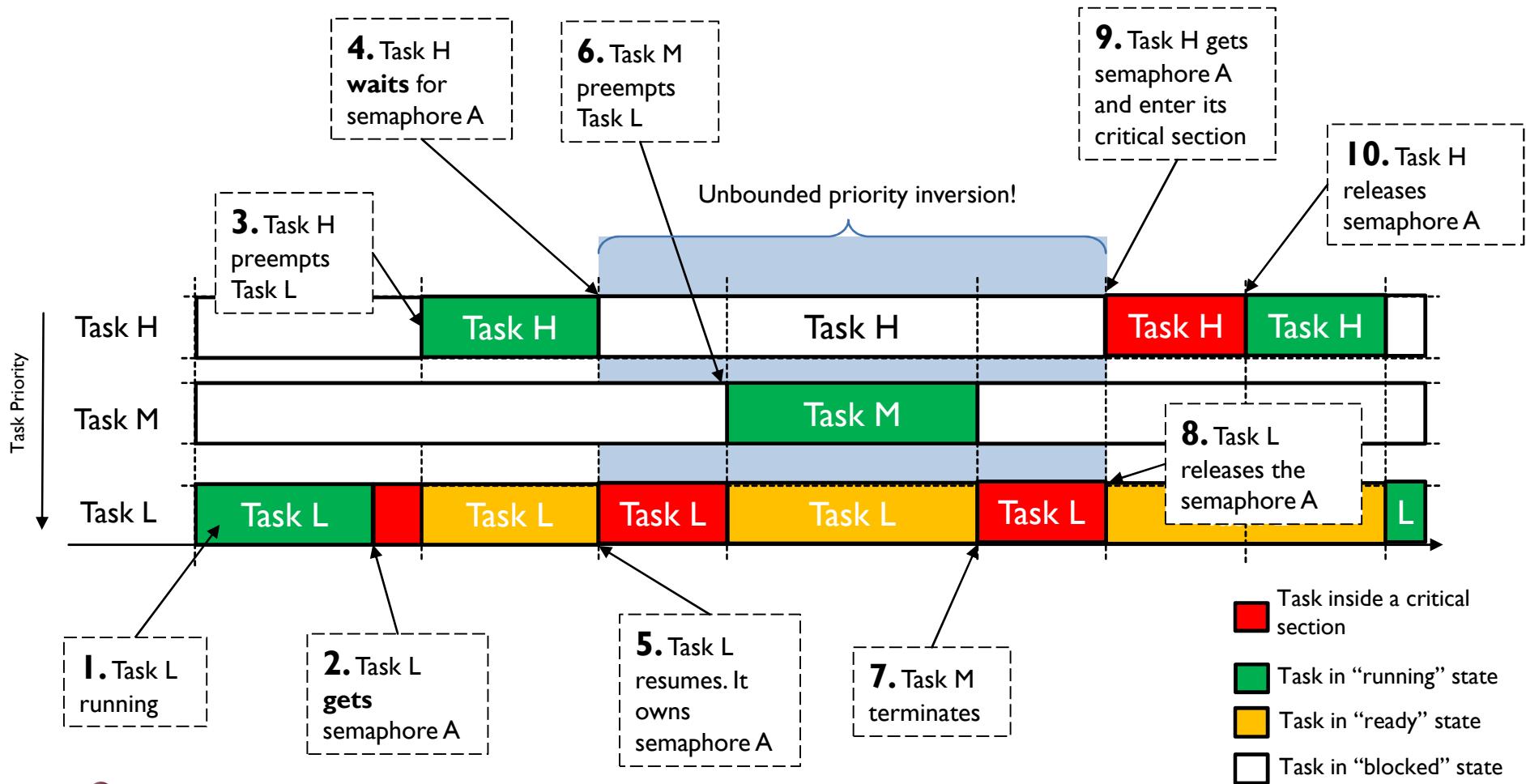
Unbounded Priority Inversion

SEMAPHORES AND MUTUAL EXCLUSION

- There is one **problem** when using semaphores for mutual exclusion within a **fixed priority** RTOS.
 - → **Priority inversion!**
- **Priority inversion:** a problematic scenario in scheduling in which a **high** priority task is indirectly preempted by a lower priority task effectively "inverting" the relative priorities of the two tasks.
 - Violates the priority model → High priority tasks can only be prevented by higher priority tasks.

SEMAPHORES AND MUTUAL EXCLUSION

- Consider the following scenario...



SEMAPHORES AND MUTUAL EXCLUSION

- The priority inversion problem is ***unbounded*** because:
 - Any *medium* priority Task can extend the time that Task H needs to wait for the resource.
- How can this problem be solved?
 - → **Priority ceiling protocol!** (a.k.a. priority inheritance)

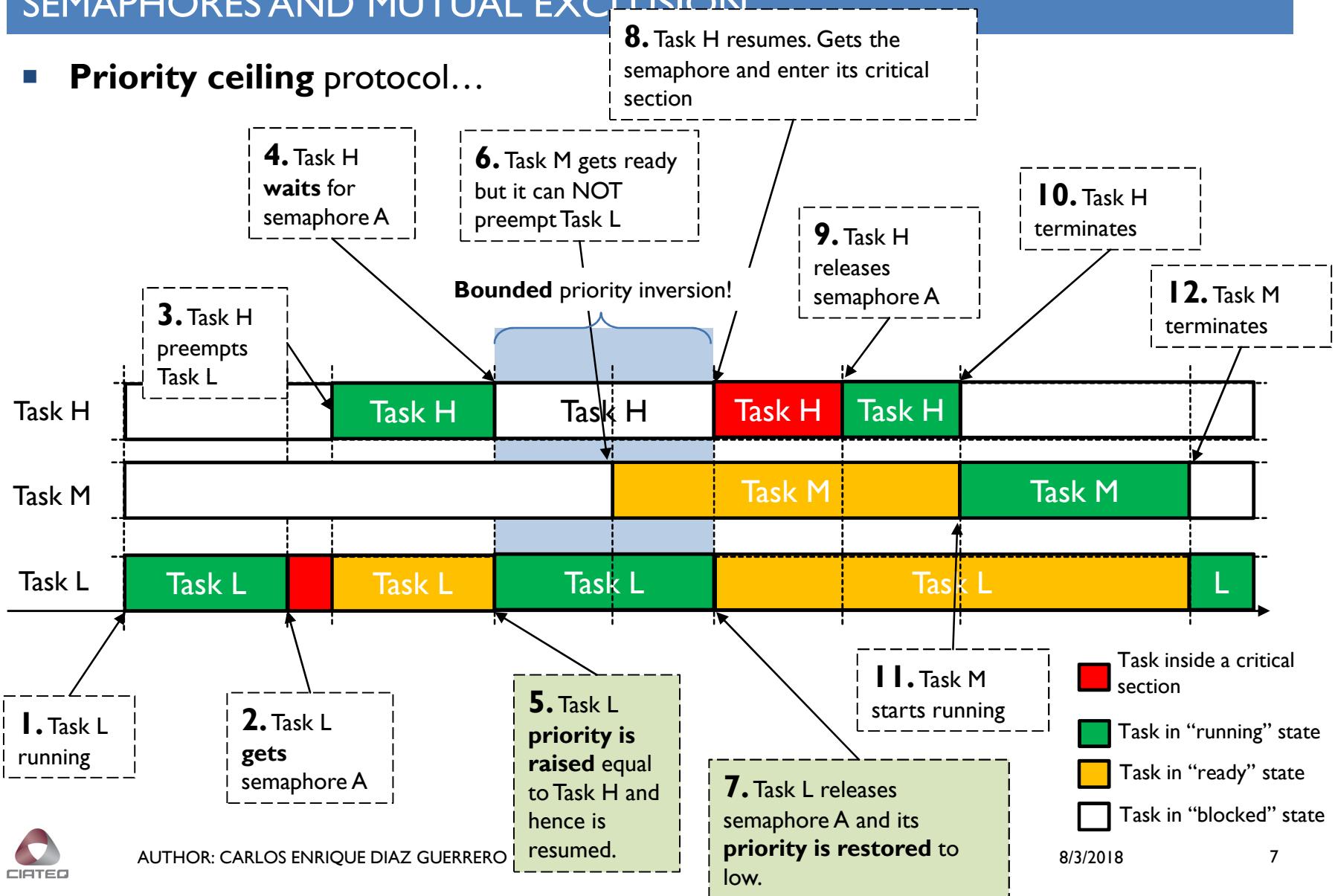
SEMAPHORES AND MUTUAL EXCLUSION

■ Priority ceiling protocol

- The problem can be avoided by temporarily raising the priority of Task L.
- In the example:
 - When Task H waits for the semaphore A, the RTOS detects that the semaphore is owned by a lower priority Task L. The priority of Task L is raised to be same as Task H.
 - By doing so, Task L will now run with high priority, will terminate its critical section code and will release the semaphore as soon as possible.
 - Once Task L releases the semaphore, its priority is restored back to its original one.
 - Now Task H preempts Task L and can get the semaphore and enter its critical section.
 - While Task L is in its critical section, no preemption can occur by medium priority tasks so the **unbounded** priority is solved → becomes a **bounded** and *short* priority inversion.

SEMAPHORES AND MUTUAL EXCLUSION

■ Priority ceiling protocol...



SEMAPHORES AND MUTUAL EXCLUSION

- Normally RTOS provide an special object implementing a binary semaphore with priority ceiling protocol → **mutex** object.
- Differences between a mutex and a binary semaphore:
 - Mutex normally implement protocol ceiling
 - The **same** task that “gets” a Mutex shall be the one that “signals” such Mutex.

SEMAPHORES AND MUTUAL EXCLUSION

- When to use **semaphores** and when a **mutex**?
 - Consider that ceiling protocol adds some overhead to the semaphore operations.
 - Use semaphores if none of the task competing for the shared resource have deadline.
 - If deadlines are to be met, mutex shall be used instead.

SEMAPHORES AND MUTUAL EXCLUSION

- **Deadlocks** when using semaphores.
- Example:
 - Two tasks, Task A and Task B need to access two shared resources R1 and R2 in a mutual exclusive way.

Task A

```
void TaskA(void){  
  
    while(TRUE){  
        Access R1;  
        Access R2;  
    }  
}
```

Task B

```
void TaskB(void){  
  
    while(TRUE){  
        Access R1;  
        Access R2;  
    }  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

- ...**Deadlock** example...
- A probable solution using mutexes:
 - Two mutexes M1 and M2 are created associated to R1 and R2 respectively.
 - Mutexes are both initialized with 1.

Task A

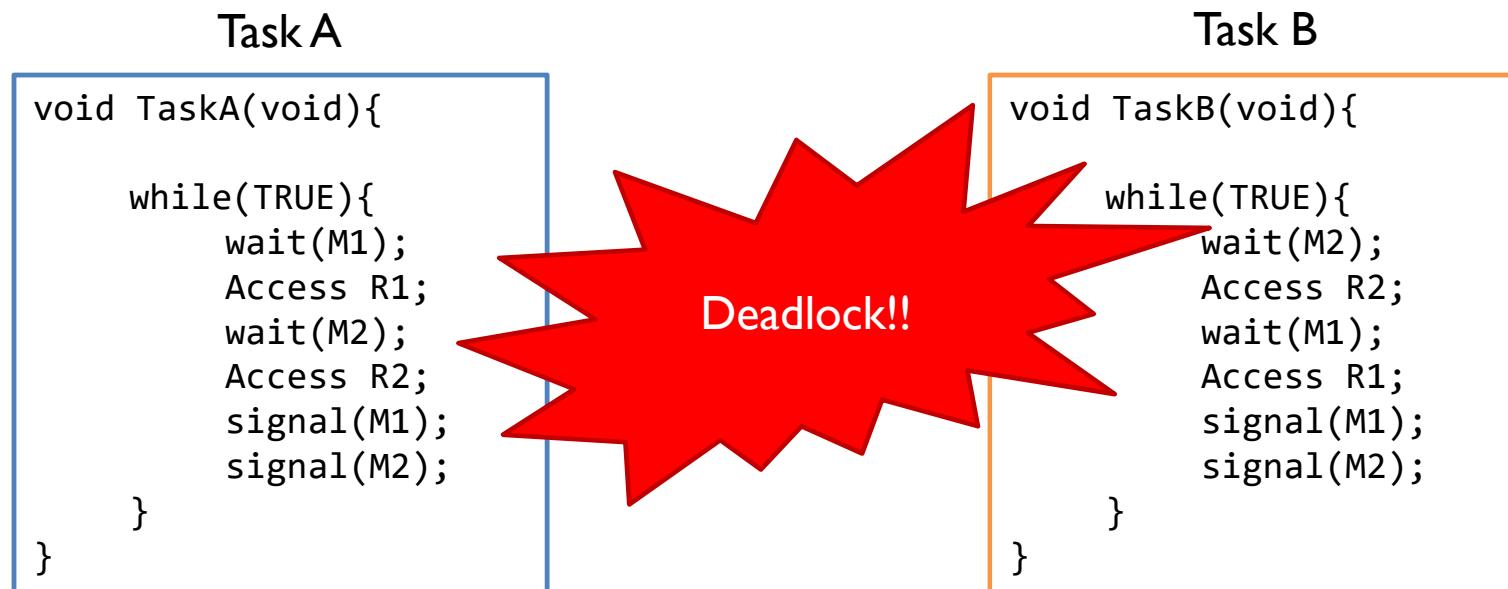
```
void TaskA(void){  
  
    while(TRUE){  
        wait(M1);  
        Access R1;  
        wait(M2);  
        Access R2;  
        signal(M1);  
        signal(M2);  
    }  
}
```

Task B

```
void TaskB(void){  
  
    while(TRUE){  
        wait(M2);  
        Access R2;  
        wait(M1);  
        Access R1;  
        signal(M1);  
        signal(M2);  
    }  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

- ...**Deadlock** example...
- A probable solution using mutexes:
 - Two mutexes M1 and M2 are created associated to R1 and R2 respectively.
 - Mutexes are both initialized with 1.



SEMAPHORES AND MUTUAL EXCLUSION

- ...**Deadlock** example...
- A probable solution option 2:

Task A

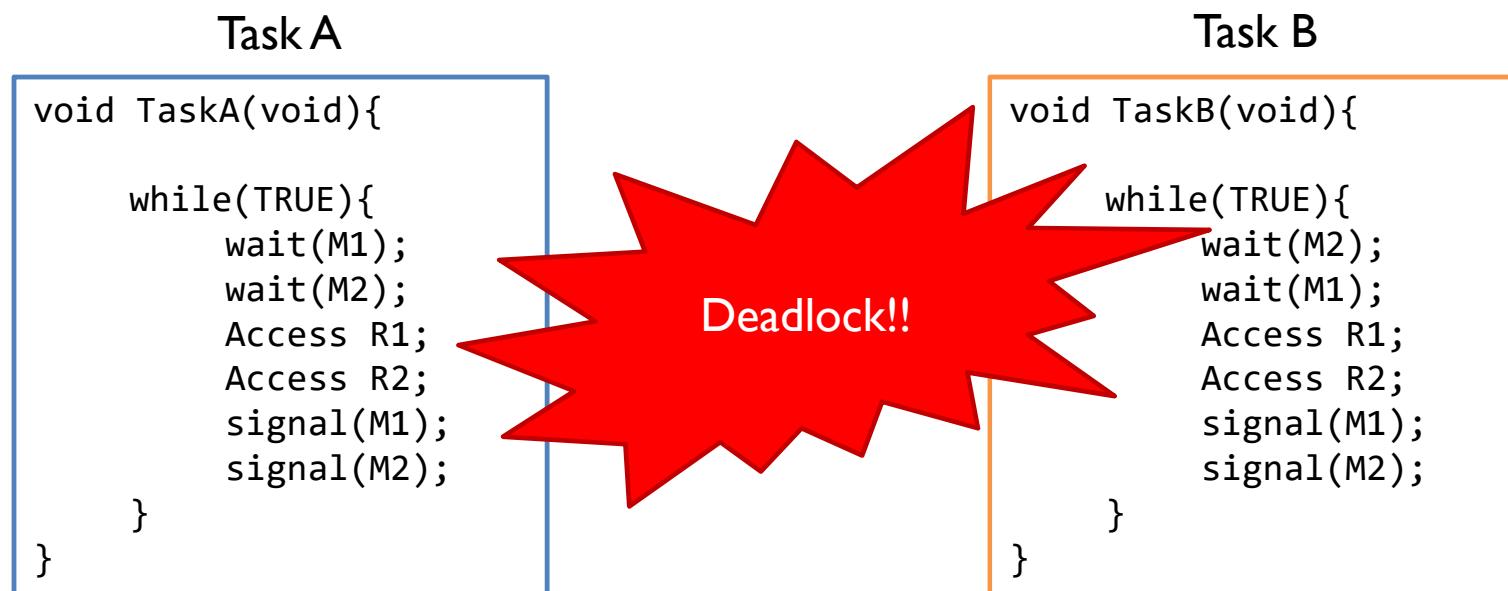
```
void TaskA(void){  
  
    while(TRUE){  
        wait(M1);  
        wait(M2);  
        Access R1;  
        Access R2;  
        signal(M1);  
        signal(M2);  
    }  
}
```

Task B

```
void TaskB(void){  
  
    while(TRUE){  
        wait(M2);  
        wait(M1);  
        Access R1;  
        Access R2;  
        signal(M1);  
        signal(M2);  
    }  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

- ...**Deadlock** example...
- A probable solution option 2:



SEMAPHORES AND MUTUAL EXCLUSION

- ...**Deadlock** example...
- Techniques to avoid deadlocks:
 - A. Acquire all resources before proceeding
 - B. Always acquire resources in the same order
 - C. Use timeouts for waiting calls

SEMAPHORES AND MUTUAL EXCLUSION

- ...**Deadlock** example...

- Correct solution!

Task A

```
void TaskA(void){  
  
    while(TRUE){  
        wait(M1);  
        wait(M2);  
        Access R1;  
        Access R2;  
        signal(M1);  
        signal(M2);  
    }  
}
```

Task B

```
void TaskB(void){  
  
    while(TRUE){  
        wait(M1);  
        wait(M2);  
        Access R1;  
        Access R2;  
        signal(M1);  
        signal(M2);  
    }  
}
```



SEMAPHORES AND MUTUAL EXCLUSION

Semaphores for Synchronization

SEMAPHORES AND MUTUAL EXCLUSION

- Semaphores are also widely used for Tasks **synchronization**
- There are several common **synchronization patterns** that can be required by an application:
 - Signaling
 - Unilateral Rendevouz
 - Bilateral Rendevouz
 - Credit tracking
 - Barrier
 - ...

SEMAPHORES AND MUTUAL EXCLUSION

■ **Signaling**

- Possibly the simplest use for a semaphore
- One task sends a signal to another task to indicate that something has happened.
- Signaling makes it possible to guarantee that a section of code in one task will run before a section of code in another task
- Solves the serialization problem.

SEMAPHORES AND MUTUAL EXCLUSION

- **Signaling** example
 - E.g. for following tasks statements:

Task A

statement a1;

Task B

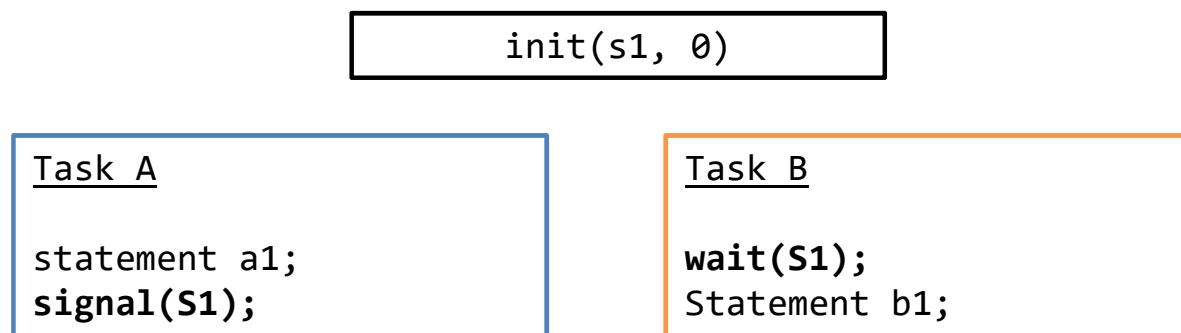
Statement b1;

- We want a1 to occur **before** b1.

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Signaling example...**

- We can use one semaphore `s1` for Task A to signal Task B that it has executed statement `a1`.



- Statement `a1` will always be executed **before** statement `b1`.

SEMAPHORES AND MUTUAL EXCLUSION

- **Unilateral rendez-vous** example

- E.g., a Task makes an I/O reading request and waits for an ISR to flag that the I/O operation has finished.

Task A

```
request I/O read data;  
Do something with the I/O data;
```

ISR

```
retrieve I/O data;
```

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Unilateral rendez-vous** example ...
 - One semaphore can be used for the ISR to signal the I/O operation completion to the Task.

```
init(sI0done, 0)
```

Task A

```
request I/O data;  
wait(sI0done);  
Do something with the I/O data
```

ISR

```
retrieve I/O data;  
signal(sI0done);
```

SEMAPHORES AND MUTUAL EXCLUSION

■ **Bilateral rendez-vous**

- Two tasks may need to synchronize their activities with one another.
- Two semaphores can be used for this situation.
- This pattern only works for **two** tasks.

SEMAPHORES AND MUTUAL EXCLUSION

- **Bilateral rendez-vous** example
 - For following tasks:

Task A

Statement a1;
Statement a2;

Task B

Statement b1;
Statement b2;

- We want to guarantee that a1 happens before b2 and b1 happens before a2.

SEMAPHORES AND MUTUAL EXCLUSION

- ...**Bilateral rendez-vous** example...
 - Solution: Two semaphores are created aArrived and bArrived to indicate when certain statement of a given task has been reached.

```
init(aArrived, 0);  
init(bArrived, 0);
```

Task A

```
Statement a1;  
signal(aArrived);  
wait(bArrived);  
Statement a2;
```

Task B

```
Statement b1;  
signal(bArrived);  
wait(aArrived);  
Statement b2;
```

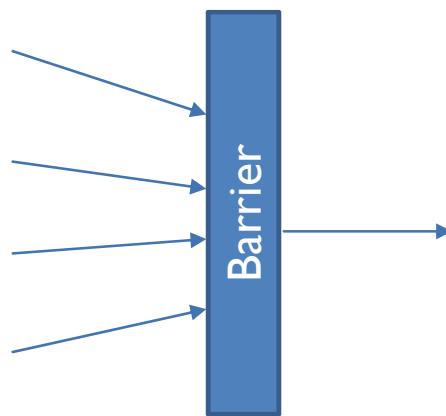
- Probable execution outcomes:
 - $a1, b1, a2, b2, \dots$ \rightarrow Still $a1 < b2$ and $b1 < a2$
 - $b1, a1, b2, a2, \dots$ \rightarrow Still $a1 < b2$ and $b1 < a2$
 - $a1, b1, a2, a1, b2, b1, a2, b2, \dots$ \rightarrow Still $a1 < b2$ and $b1 < a2$
 - \dots \rightarrow Always $a1 < b2$ and $b1 < a2$!!!



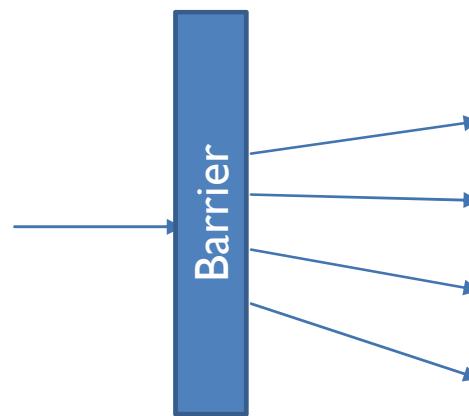
SEMAPHORES AND MUTUAL EXCLUSION

■ Barrier

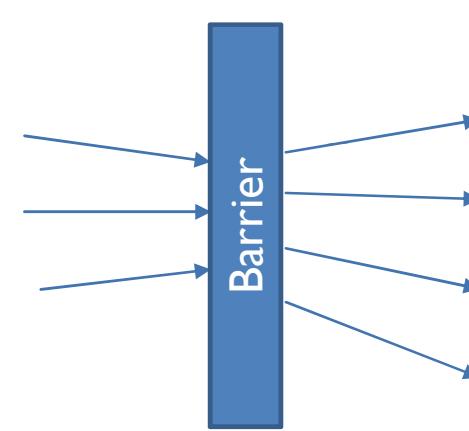
- Generalizes the “rendesvouz” pattern to any number of tasks.
- In order for a Task to proceed, all other tasks shall reach certain point.



Many to One:
Many Tasks enabled a
single Task



One to Many:
Single Task enables
Many Tasks



Many to Many:
X number of Tasks
enable Y number of
Tasks

SEMAPHORES AND MUTUAL EXCLUSION

- **Barrier example**
 - We have n Tasks that look as follows:

Task i

```
rendevouz statement;  
Critical code;
```

- Requirement: no task shall executes its critical code until after all tasks have executed rendezvous statement.
- There are n tasks and that this value is stored in a variable, n , that is accessible from all tasks.
- When the first $n - 1$ tasks arrive they should block until the nth task arrives, at which point all the tasks may proceed.

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Barrier** example ...

- Solution:

- Following objects are needed:
 - Variable n : number of tasks
 - Variable $count$: keeps track of how many tasks have arrived to the rendezvous point.
 - Mutex $mCount$: used to provide mutually exclusive access to variable $count$.
 - A semaphore $sBarrier$: Semaphore to lock or unlock the barrier and allow Tasks to continue execution of their critical codes.
 - Objects initial values:
 - n = number of tasks
 - $count = 0$: initially, no task has reached the rendezvous point.
 - $mCount = 1$: initially access to variable $count$ is allowed.
 - $sBarrier = 0$: Semaphore is initially locked until all tasks arrive to their rendezvous point.
 - Assumptions:
 - Writing $count$ variable takes more than one CPU instruction but reading it takes only one.

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Barrier** example ...

- Solution:

initialization

```
Count = 0;  
init(mCount, 1);  
init(sBarrier, 0);
```

Task *i*

```
rendevouz statement;  
  
wait(mCount);  
    count = count + 1;  
signal(mCount);  
  
if count == n  
then signal(sBarrier);  
  
wait(sBarrier);  
signal(sBarrier);  
  
critical code;
```

- Such solution has still some caveats → Barrier is not re-usable

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Barrier** example ...
 - We may have think also in the following probable solution:

```
initialization  
  
Count = 0;  
init(mCount, 1);  
init(sBarrier, 0);
```

```
Task i  
  
rendevouz statement;  
  
wait(mCount);  
    count = count + 1;  
signal(mCount);  
  
if count == n  
then signal(sBarrier);  
  
wait(sBarrier);  
  
critical code;
```

- Is there a **problem** with this solution???

SEMAPHORES AND MUTUAL EXCLUSION

- Solving synchronization/concurrency problems:
 - Experience required, if not,
 - Formal methods are available:
 - Petri Nets, etc.

SEMAPHORES AND MUTUAL EXCLUSION

- Classical **synchronization problems**:
 - **Producer-Consumer** problem
 - Infinite buffer
 - Finite buffer
 - **Readers-Writers** problem
 - **Dinning philosophers** problem
 - **Cigarette Smokers** problem



SEMAPHORES AND MUTUAL EXCLUSION

Producer-Consumer: Infinite Buffer

SEMAPHORES AND MUTUAL EXCLUSION

- **Producer-Consumer** problem: **Infinite** buffer
 - **Producers** create items of some kind and add them to a data structure (a buffer).
 - **Consumers** remove the items from the buffer and process them.
 - We will assume the buffer is of **infinite** size.
- **Constraints:**
 - Adding/removing items operations shall be executed atomically → Tasks must have exclusive access to the buffer.
 - If a consumer Task arrives while the buffer is empty, it blocks until a producer Task adds a new item.

SEMAPHORES AND MUTUAL EXCLUSION

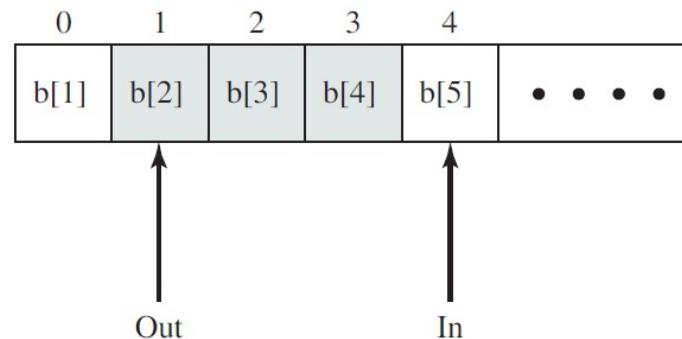
- ... **Producer-Consumer** problem: Infinite buffer ...
 - Tasks will look like as follows:

```
producer:           consumer:  
while (true) {      while (true) {  
    /* produce item v */;  
    b[in] = v;  
    in++;  
}  
                    while (in <= out)  
                    /* do nothing */;  
w = b[out];  
out++;  
/* consume item w */;  
}
```

- Variable “in” is a pointer to the buffer slot where the next item is to be added by a producer.
- Variable “out” is a pointer to the buffer slot where a consumer will take an item from.
- Array “b[]” is the buffer → it can be seen as a FIFO

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- The infinite buffer will look as follows:



SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...

Infinite Buffer → Solution using **binary** semaphores

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- Solution using **binary** semaphores
- Additional variables and objects needed:
 - An auxiliary variable *n* that will tell the number of items in the buffer
 - A **binary** semaphore M1 for accesing the buffer in a mutually exclusive way
 - A **binary** semaphore S1 to put a consumer in blocked state if the buffer is empty.

```
init(M1, 1);
init(S1, 0);
n = 0;
in = 0;
out = 0;
```

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- **Probable** Solution to the problem: producer/consumer code

```
producer:  
  
void producer(){  
    while(true){  
        producedItem = produce();  
        wait(M1);  
        apendItem(producedItem);  
        n++;  
        if(n == 1) signal(S1);  
        signal(M1);  
    }  
}  
  
void apendItem(item){  
    b[in] = item;  
    in++;  
}
```

```
consumer:  
  
void consumer(){  
    wait(S1);  
    while(true){  
        wait(M1);  
        localItem = takeItem();  
        n--;  
        signal(M1);  
        consume(LocalItem);  
        if(n==0) wait(S1);  
    }  
}  
  
item takeItem(){  
    item = b[out];  
    out++;  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- **Probable** Solution to the problem: producer/consumer code

```
producer:
```

```
void producer(){
    while(true){
        producedItem = produce();
        wait(M1);
        apendItem(producedItem);
        n++;
        if(n == 1) signal(S1);
        signal(M1);
    }
}

void apendItem(item){
    b[in] = item;
    in++;
}
```

```
consumer:
```

```
void consumer(){
    wait(S1);
    while(true){
        wait(M1);
        localItem = takeItem();
        n--;
        signal(M1);
        consume(LocalItem);
        if(n==0) wait(S1);
    }
}
```

Problem: Consumer
may “take” an item
from an empty
buffer!

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- **Correct** Solution to the problem: producer/consumer code

```
producer:
```

```
void producer(){
    while(true){
        producedItem = produce();
        wait(M1);
        apendItem(producedItem);
        n++;
        if(n == 1) signal(S1);
        signal(M1);
    }
}

void apendItem(item){
    b[in] = item;
    in++;
}
```

```
consumer:
```

```
void consumer(){
    int m; /* local variable */
    wait(S1);
    while(true){
        wait(M1);
        localItem = takeItem();
        n--;
        m = n; /* backup value of n */
        signal(M1);
        consume(LocalItem);
        if(m==0) wait(S1);
    }
}

item takeItem(){
    item = b[out];
    out++;
}
```



SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...

Infinite Buffer → Solution using **counting** semaphores

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- Solution using **counting** semaphores
- Additional variables and objects needed:
 - A **counting** semaphore M1 used to have mutually exclusive access to the buffer
 - A **counting** semaphore N which actually tells how many items are in the buffer.

```
init(M1, 1);
init(N, 0);
in = 0;
out = 0;
```

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- Solution to the problem: producer/consumer code (**counting semaphores**)

```
producer:  
  
void producer(){  
    while(true){  
        producedItem = produce();  
        wait(M1);  
        appendItem(producedItem);  
        signal(M1);  
        signal(N);  
    }  
}  
  
void appendItem(item){  
    b[in] = item;  
    in++;  
}
```

```
consumer:  
  
void consumer(){  
    while(true){  
        wait(N);  
        wait(M1);  
        localItem = takeItem();  
        signal(M1);  
        consume(LocalItem);  
    }  
}  
  
item takeItem(){  
    item = b[out];  
    out++;  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Infinite buffer ...
- More clean solution using counting semaphores rather than binary semaphores!.



SEMAPHORES AND MUTUAL EXCLUSION

Producer-Consumer: Finite Buffer

SEMAPHORES AND MUTUAL EXCLUSION

- **Producer-Consumer** problem: **Finite** buffer
 - **Producers** create items of some kind and add them to a data structure (a buffer).
 - **Consumers** remove the items from the buffer and process them.
 - Realistic case: buffer is of **finite** size.
 - Assuming circular buffer
- **Constraints:**
 - Adding/removing items operations shall be executed atomically → Tasks must have exclusive access to the buffer.
 - If a consumer Task arrives while the buffer is empty, it blocks until a producer Task adds a new item.
 - If the buffer is full, producer Task blocks until a consumer Task takes an item.

SEMAPHORES AND MUTUAL EXCLUSION

- **Producer-Consumer** problem: **Finite** buffer

- Tasks will look like as follows:

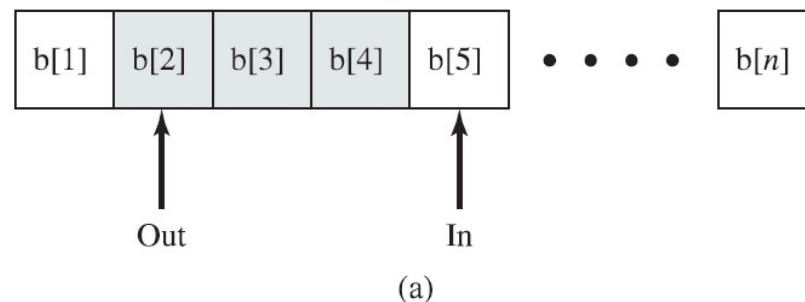
```
producer:  
while (true) {  
    /* produce item v */  
    while ((in + 1) % n == out)  
        /* do nothing */;  
    b[in] = v;  
    in = (in + 1) % n;  
}
```

```
consumer:  
while (true) {  
    while (in == out)  
        /* do nothing */;  
    w = b[out];  
    out = (out + 1) % n;  
    /* consume item w */;  
}
```

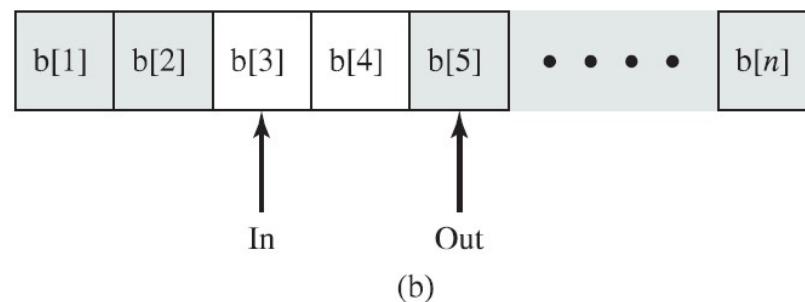
- Variable “n” is the size of the buffer
- Array “b[]” is the circular buffer

SEMAPHORES AND MUTUAL EXCLUSION

- **Producer-Consumer** problem: **Finite** buffer



(a)



(b)

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Finite buffer ...

Finite Buffer → Solution using **counting** semaphores

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Finite buffer ...
- Solution using **counting** semaphores
- Additional variables and objects needed:
 - A **counting** semaphore M1 used to have mutually exclusive access to the buffer
 - A **counting** semaphore N which actually tells how many items are in the buffer.
 - A **counting** semaphore E which tells the size “n” of the buffer

```
init(M1, 1);
init(N, 0);
init(E, n);
```

SEMAPHORES AND MUTUAL EXCLUSION

- ... **Producer-Consumer** problem: Finite buffer ...
- Solution to the problem: producer/consumer code (**counting semaphores**)

```
producer:
```

```
void producer(){
    while(true){
        producedItem = produce();
        wait(E);
        wait(M1);
        appendItem(producedItem);
        signal(M1);
        signal(N);
    }
}
```

```
consumer:
```

```
void consumer(){
    while(true){
        wait(N);
        wait(M1);
        localItem = takeItem();
        signal(M1);
        signal(E);
        consume(LocalItem);
    }
}
```



SEMAPHORES AND MUTUAL EXCLUSION

Dinning Philosophers

SEMAPHORES AND MUTUAL EXCLUSION

Dinning Philosophers problem

- Proposed by Dijkstra in 1965
- Scenario:
 - A table with 5 plates.
 - Five forks (or chopsticks) in the table, each next to a plate.
 - A big bowl of spaghetti at the center of the table
- Actors:
 - 5 philosophers sitting around the table.
 - Every philosopher **thinks** and **eats**
 - To be able to eat, each philosopher needs to take **two** forks, the one at his left and the one at his right.
 - Once the philosopher finishes eating, he puts the two forks back in the table and starts thinking...

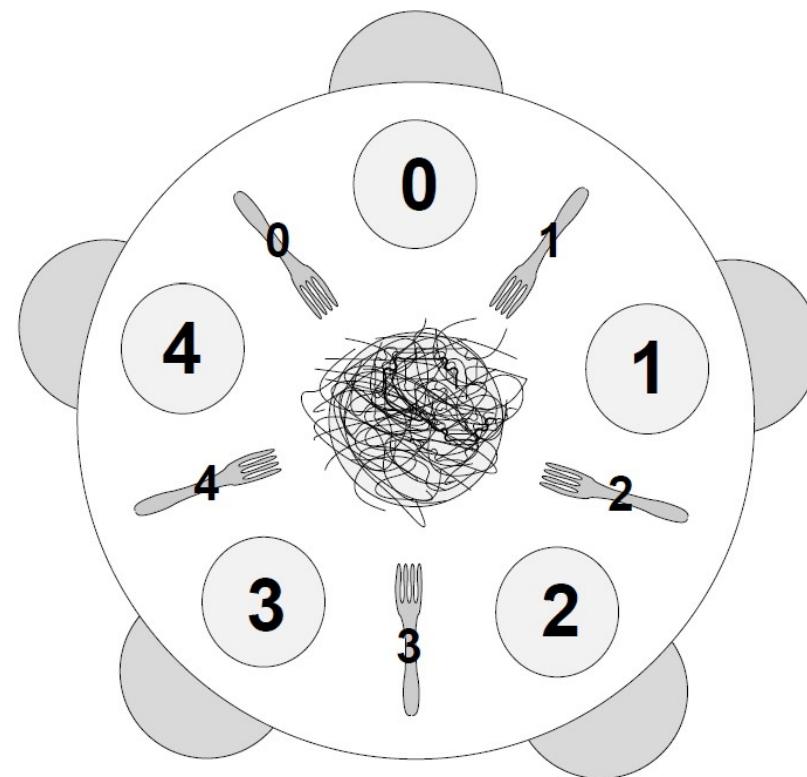
SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...

- Constraints:
 - Only one philosopher can hold a fork at a time.
 - It must be impossible for a deadlock to occur.
 - It must be impossible for a philosopher to starve waiting for a fork.
 - It must be possible for more than one philosopher to eat at the same time.
 - The greater the number of philosophers allowed to eat at the same time, the better (higher concurrency).

SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...



SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...

- Every philosopher can be considered a Task
- Main actions of each philosopher Task are:
 - Thinking → No shared resource needed
 - Eating → Forks are needed shared resources
- Every fork is a shared resource to be able to **eat**.

SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...

- At a glance...

Philosopher *i* Task:

```
void philosopher(){  
    while(true){  
        think();  
        eat();  
    }  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...

- For eating each philosopher i needs to take two forks the one at his left and the one at his right.

```
int leftFork(int i){  
    return(philNumber);  
}  
  
int rightFork(int i){  
    return((i+ 1) % 5);  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...

- An array of 5 mutexes can be created to allow exclusive access to each of the 5 forks.

```
mutex forks[5];
```

- A non-solution to the problem...

SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...

- A non-solution to the problem (deadlock)...

```
Philosopher i Task:  
  
void philosopher(int i){  
  
    while(true){  
        think();  
        /* try to get both forks */  
        wait(forks[leftFork(i)]);  
        wait(forks[rightFork(i)]);  
  
        eat();  
  
        /* release both forks */  
        signal(forks[leftFork(i)]);  
        signal(forks[rightFork(i)]);  
    }  
}
```

SEMAPHORES AND MUTUAL EXCLUSION

... Dinning Philosophers problem ...

- Solutions????
- ...
- ...



REAL-TIME OPERATING SYSTEMS

MAESTRÍA EN SISTEMAS INTELIGENTES MULTIMEDIA

INSTRUCTOR: CARLOS ENRIQUE DIAZ GUERRERO



AUTHOR: CARLOS ENRIQUE DIAZ GUERRERO

7/29/2019

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

Events Management

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- An **event** is essentially a Boolean **flag** that tasks or ISRs can set or reset and that other tasks can wait for.
- Events are used for **Task synchronization / Signaling**.
- ISRs can **only post** events but they can NOT **wait** or pend for an event.
- Tasks can both post and wait for events

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Main **event** operations:
 - **Create** the event → allocate and initialize the necessary data structures for handling such event
 - Generate the event or “**post**” the event → set the corresponding event flag. Normally this will re-activate a task or tasks that may be waiting for it.
 - Wait for the event or “**pend**” on the event → Task can be set to be blocked until certain event or set of events are occur (until events are posted).
 - While waiting for an event, Task can define a maximum time to wait for (timeout).
 - **Consume** event → clear the corresponding event flag once the event has been processed so it is not unintendedly re-triggered.
 - Similar than clearing an interrupt flag.
 - **Delete** the event → deallocate the data structure for the event

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Events (**event flags**) are normally organized on **event groups**
- Since each event can be associated to one bit, normally RTOS use a “word” for creating groups of events.
 - E.g., in a 32 bit MCU, the event groups will be of 32 event flags (an “int” variable can be used for each group).
 - E.g., in a 16 bit MCU, the event groups will be of 16 event flags (an “int” variable can be used for each group).

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Task can use **masks** to wait for certain subset of event flags within an event group.
 - Such events will be the ones **associated** to a given task.
- Task can be synchronized upon events when:
 - **Any** of the events associated to such task has occurred
 - Disjunctive synchronization (logical **OR**)
 - **All** of the events associated to such task have occurred
 - Conjunctive synchronization (logical **AND**)

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Example of APIs for **event** management in µC/OS III:
 - `OSFlagCreate()` → Create an event flag group
 - `OSFlagDel()` → Delete an event flag group
 - `OSFlagPend()` → Pend (i.e., wait for) on an event flag group
 - `OSFlagPendAbort()` → Abort waiting on an event flag group
 - `OSFlagPendGetFlagsRdy` → Get the flags that caused a task to become ready
 - `OSFlagPost()` → Post flags to an event flag group

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

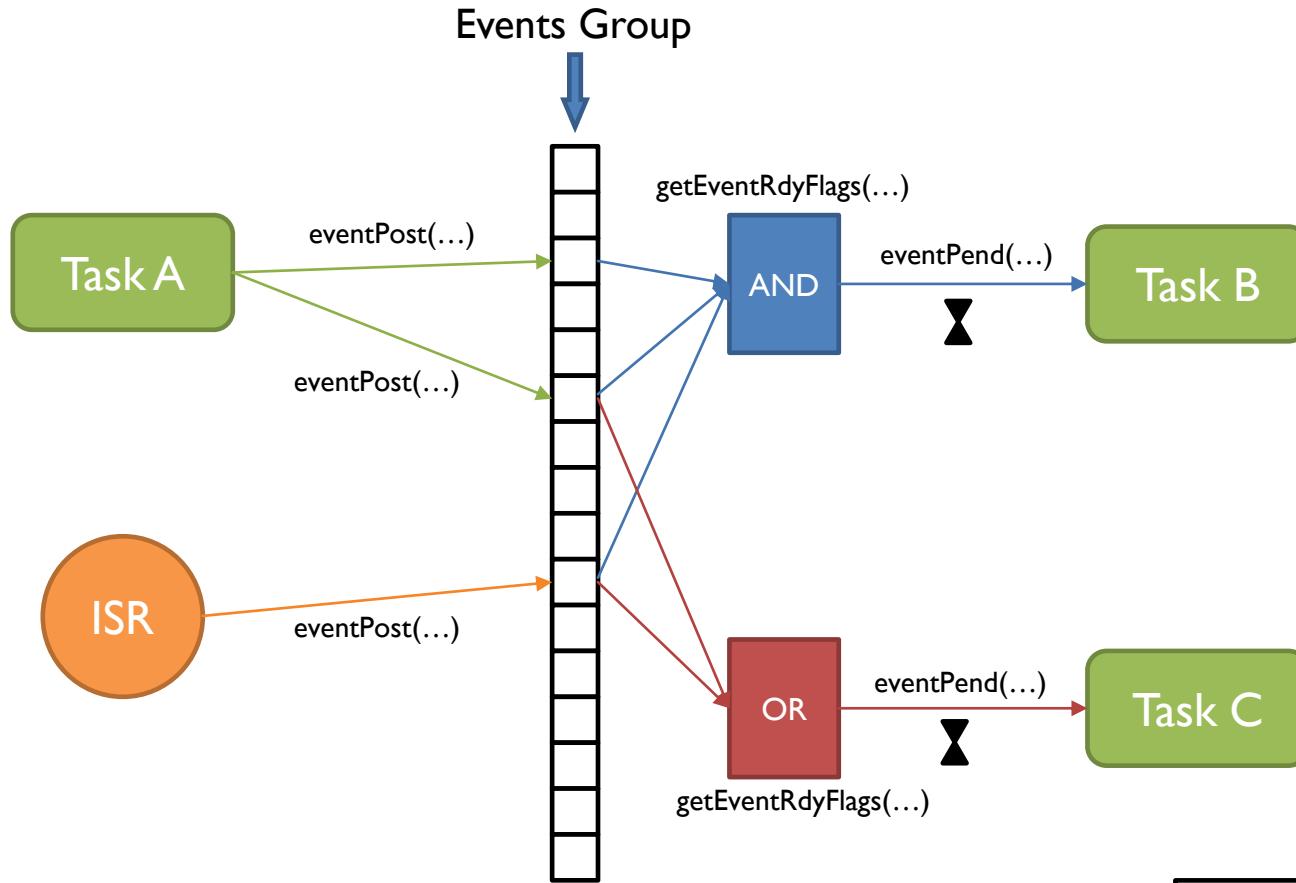
- The application shall give the meaning to each of the **event flags**. Examples:
 - An event that indicates that a button has been pressed
 - An event associated to a temperature sensor and that gets active when temperature reaches certain threshold
 - An event telling that a timer has expired
 - ...
- Normally RTOS provide means of configuring how the event flags are to be consumed:
 - Automatically when activating tasks.
 - Not done by the RTOS → the application should do it.

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Many **synchronization patterns** can be achieved using **events...**

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

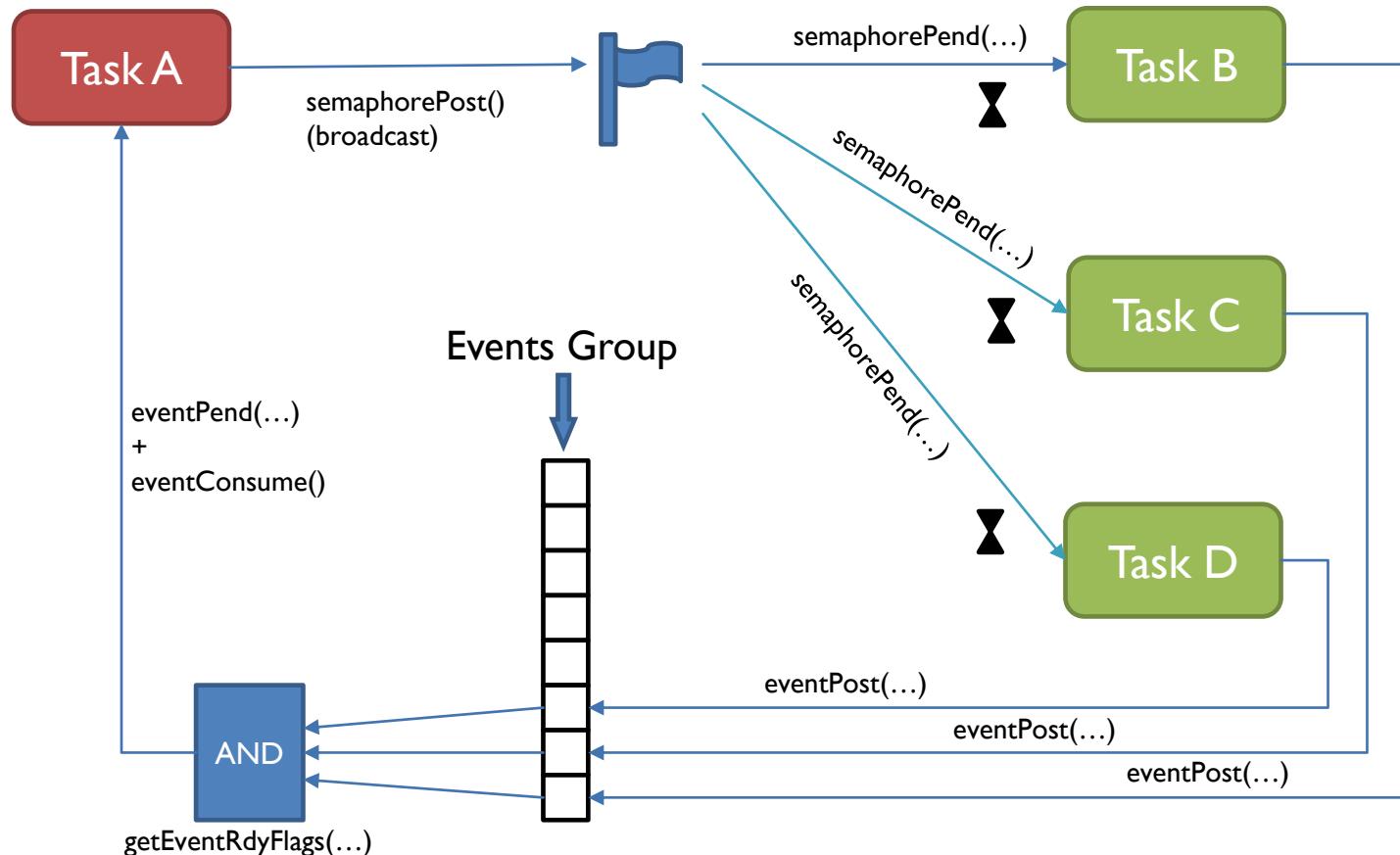
- **Signaling...**



⌚ = Timeout

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Multiple tasks **Rendezvous (Barrier)**...



Intertask Communication

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Tasks must be able to communicate with one another to coordinate their activities or to share data.
- Common means of communication:
 - Queues
 - Mailboxes
 - Pipes (Out of scope, more common in non real-time OS)

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Communication among tasks can be performed by means of **messages**
- A **message** is basically a data item which is exchanged between tasks.
 - Sometimes the **data item** is actually a pointer to the beginning of the data structure to be “transferred”.
- In a RTOS, tasks generally have direct access to a common memory space, and the fastest way to share data is by sharing memory.
 - In ordinary OS's, tasks are usually prevented from accessing another task's memory.
- Communication can be **synchronous** or **asynchronous**

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

■ **Synchronous** Communication

- Exchange of message is an **atomic** reaction that requires participation of the sender task, and the receiver task.
 - If the sender is ready to send, but the receiver is not ready to receive, the **sender is blocked**.
 - If the receiver is ready to receive, but the sender is not ready to send, the **receiver is blocked**.
-
- Example → A telephone call.

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

■ **Asynchronous** Communication

- There is **no temporal dependence** between the sending of the message and the receiving of a message.
 - The sender can send a message and continue independent of the state of the receiver.
 - The receiver can receive a message and continue independent of the state of the sender.
-
- Example → An e-mail.

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

■ **Synchronous vs Asynchronous** Communication

- Both synchronous and asynchronous communication have their advantages.
- Asynchronous communication does not block sender or receiver, if there counterpart is not ready.
- Asynchronous communication requires a buffer to store messages that are sent, but not yet received.

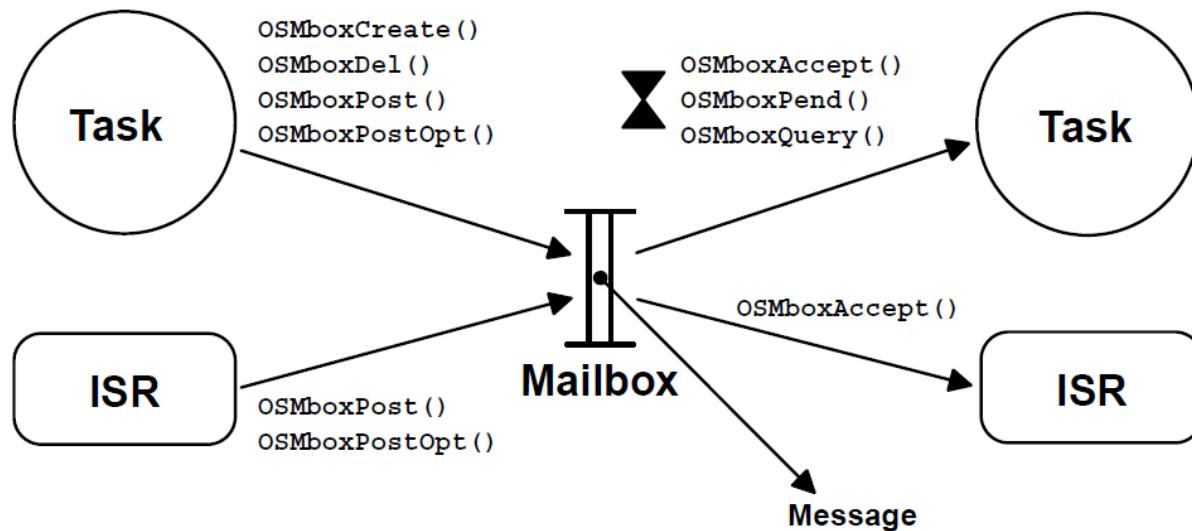
Message Mailbox

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- A **message mailbox** (or simply a mailbox) is a placeholder for a **single** message to be transferred between an ISR and a Task or between two Tasks.
- The RTOS normally “*transfers*” the message by means of:
 - A **pointer** to the application-specific data structure containing the message data.
 - A variable telling the **size** of the message data

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Example: Mailbox APIs in μ C/OS II:



EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- **Common operations** over a mailbox:
 - **Create** → allocate and initialize the necessary data structures for the mailbox.
 - **Waiting** for a message or “**pend**” on the message → The calling Task will get blocked until a message arrives to the mailbox.
 - **Sending** a message to the mailbox. “**Posting**” the message → Deposits a message into a mailbox
 - Posting to a single mailbox → A waiting task will be re-activated.
 - Posting to several mailboxes (broadcasting) → All waiting tasks of the individual mailboxes will be re-activated.
 - **Access** a message → Retrieves the data of an existing message in the mailbox.
 - **Querying** the mailbox → Check whether or not the mailbox is empty.
 - **Delete** the mailbox → de-allocate the data structure for the mailbox.

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- RTOS normally provide Non-Blocking versions of the mailbox “waiting” APIs.
- RTOS normally provide an option to specify a waiting **timeout** when using the Blocking APIs for getting a message from a mailbox.
- Mailboxes can be used as **binary semaphores** (with data transfer).
 - Pend on a message → equivalent to the wait() operation of a semaphore.
 - Post a message → equivalent to the signal() operation of a semaphore.
 - Using mailboxes, additionally a data item will be transferred!

Message Queues

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

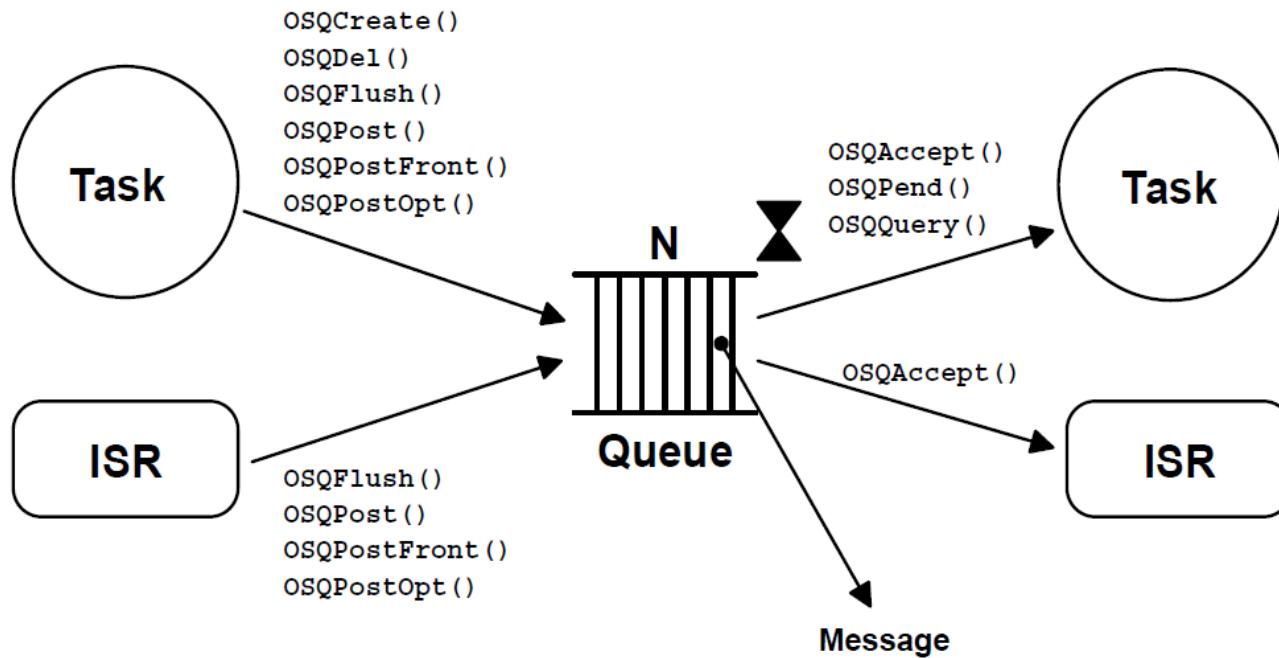
- **Message queues** generalize the mailboxes but for “*n*” number of messages.
- “*n*” is indeed the size of the queue.
- As in the case of the mailboxes, for message queues, the RTOS normally “*transfers*” the message by means of:
 - A **pointer** to the application-specific data structure containing the message data.
 - A variable telling the **size** of the message data

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Common **operations** over a message queue:
 - **Create** the message queue → allocate and initialize the necessary data structures for the queue.
 - **Sending** a message to the mailbox. “**Posting**” the message → Deposits a message into the queue. If the queue is full, the calling Task gets blocked.
 - Normally message is posted at the back of the queue but some RTOS allow a message to be also posted at the front of the queue.
 - **Getting** a message. “**Pend**” on a message → Try to get a message from the queue. If the queue is empty, the calling Task gets blocked otherwise it does NOT block.
 - Normally the message is retrieved from the front of the queue
 - **Access** a message → Retrieves the data of an existing message in the message queue.
 - **Querying** the message queue → Check whether or not the mailbox is empty.
 - **Flush** the message queue → Remove all messages from the queue. After this the queue will be empty.
 - **Delete** the message queue → de-allocate the data structure for the mailbox.

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Example: Message Queue APIs in μ C/OS II:



EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- RTOS normally provide Non-Blocking versions of the message queue “waiting” APIs.
- RTOS normally provide an option to specify a waiting **timeout** when using the Blocking APIs for getting a message from the message queue.
- Message Queues can be used as **counting semaphores** (with data transfer).
 - Pend on a message → equivalent to the `wait()` operation of a semaphore.
 - Post a message → equivalent to the `signal()` operation of a semaphore.
 - Number of messages in the queue → equivalent to the value of the semaphore.
 - Using mailboxes, additionally a data item will be transferred.

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- **Mailboxes** → Can be used for **synchronous communication**
 - The producer and consumer will be operating at the same rate
- **Message queues** → Can be used for **asynchronous communication**:
 - The producer and the consumer can be operating at different rates

Waiting Queues

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Recalling → Tasks may be **waiting** for:
 - A counting semaphore to be positive
 - A mutex to be released
 - An event flag(s) to happen
 - A message to arrive
 - ...
- Task that are blocked while waiting are placed into a **Waiting Queue** (a.k.a. Pend Queue)
 - Again, linked lists are the best candidate for implementing such queues

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- We will analyze an example implementation of a waiting queue as done by µC/OS III...

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- A **waiting queue** is similar to a ready queue, except that instead of keeping track of Tasks ready to be executed, it keeps track of Tasks **waiting** for an **object** to be posted.



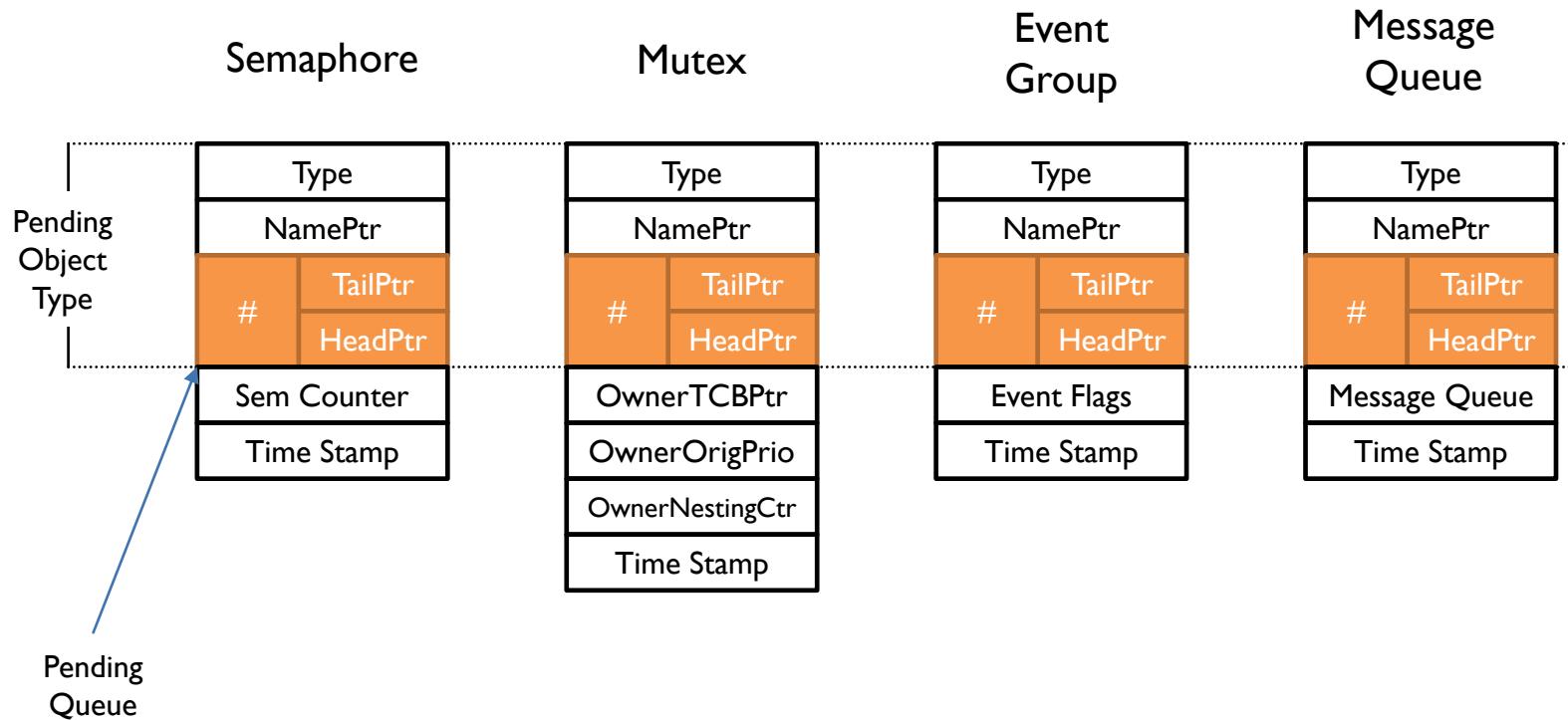
- The **items** of such waiting queue normally are data structures (**pending Data**) that will point to “**pending objects**” and to **TCBs** of the Task(s) waiting for such object.

EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- “**Pending Objects**” are indeed of any of the following types:
 - Semaphores
 - Mutex
 - Event Group
 - Message queue
- A Task can be waiting for **more than one** object at a time:
 - E.g., in μ C/OS III only possible to be pending for multiple semaphores and/or message queues → The **first** semaphore or message to be posted will set the waiting Task as **ready**.

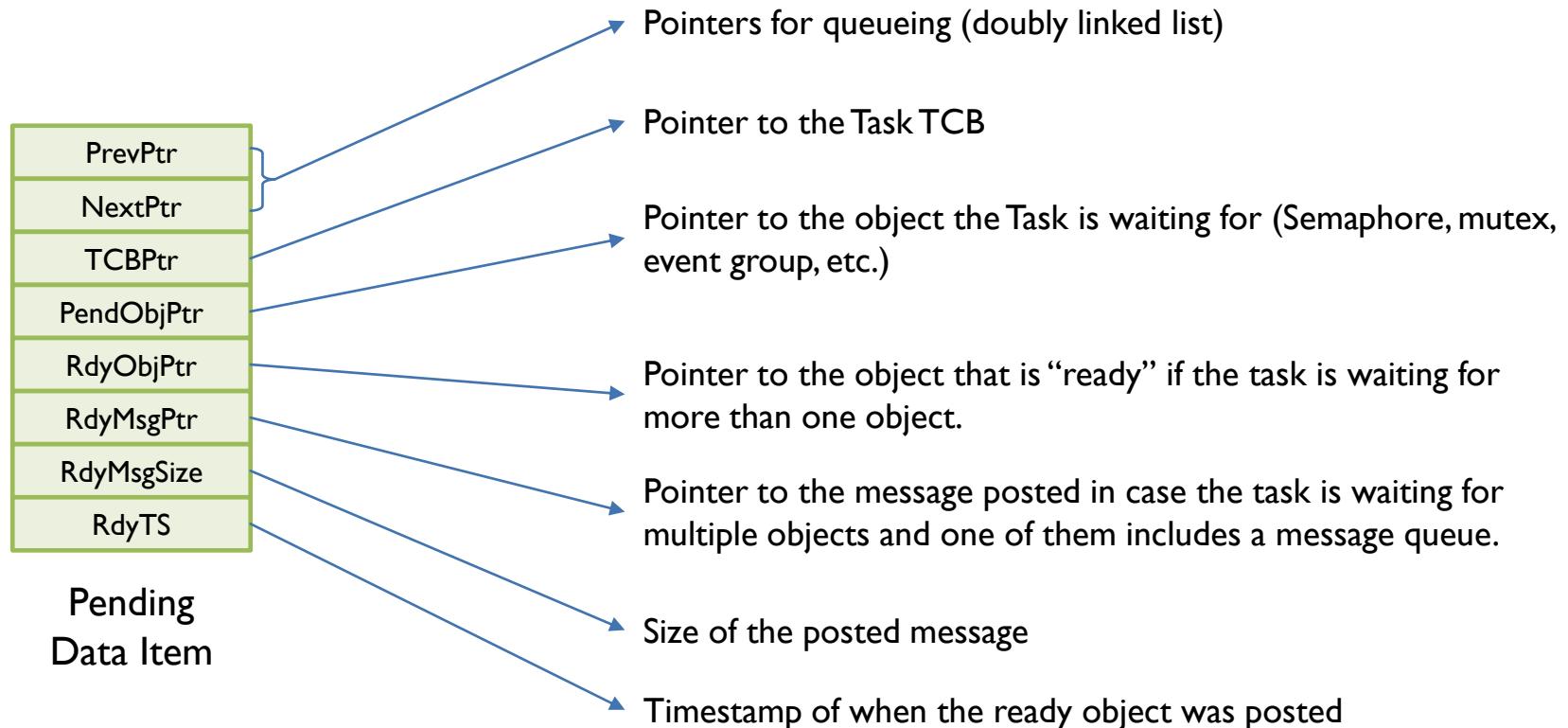
EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Implementation of the data structures for the different “**Pending Objects**” (μ C/OS III):



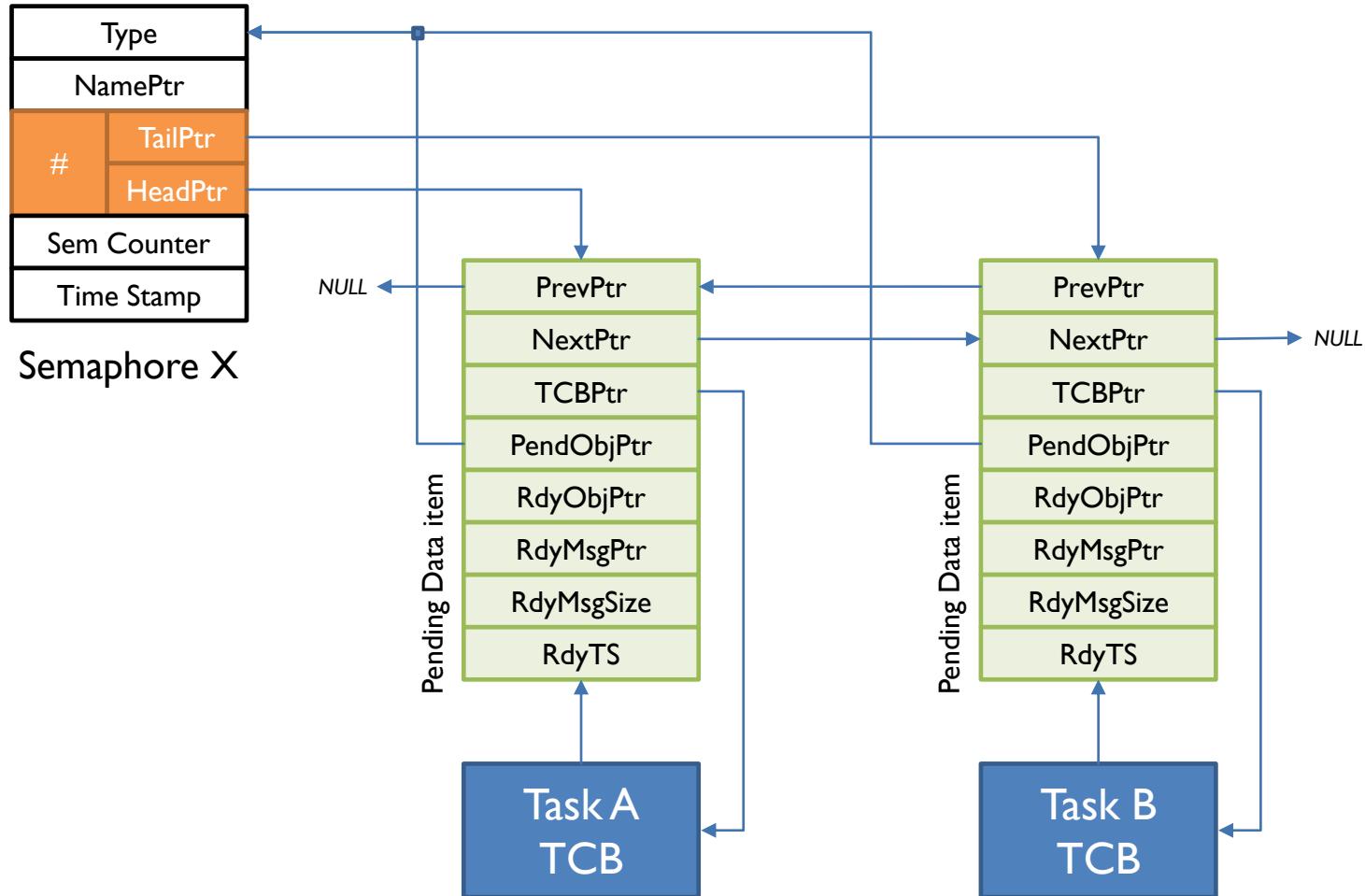
EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- The “items” of the pending queue is indeed a “pending data” structure that allows a task to wait for multiple objects (semaphores and/or messages only):



EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- **Example:** Two Tasks (Task A and B) waiting for a **semaphore X**

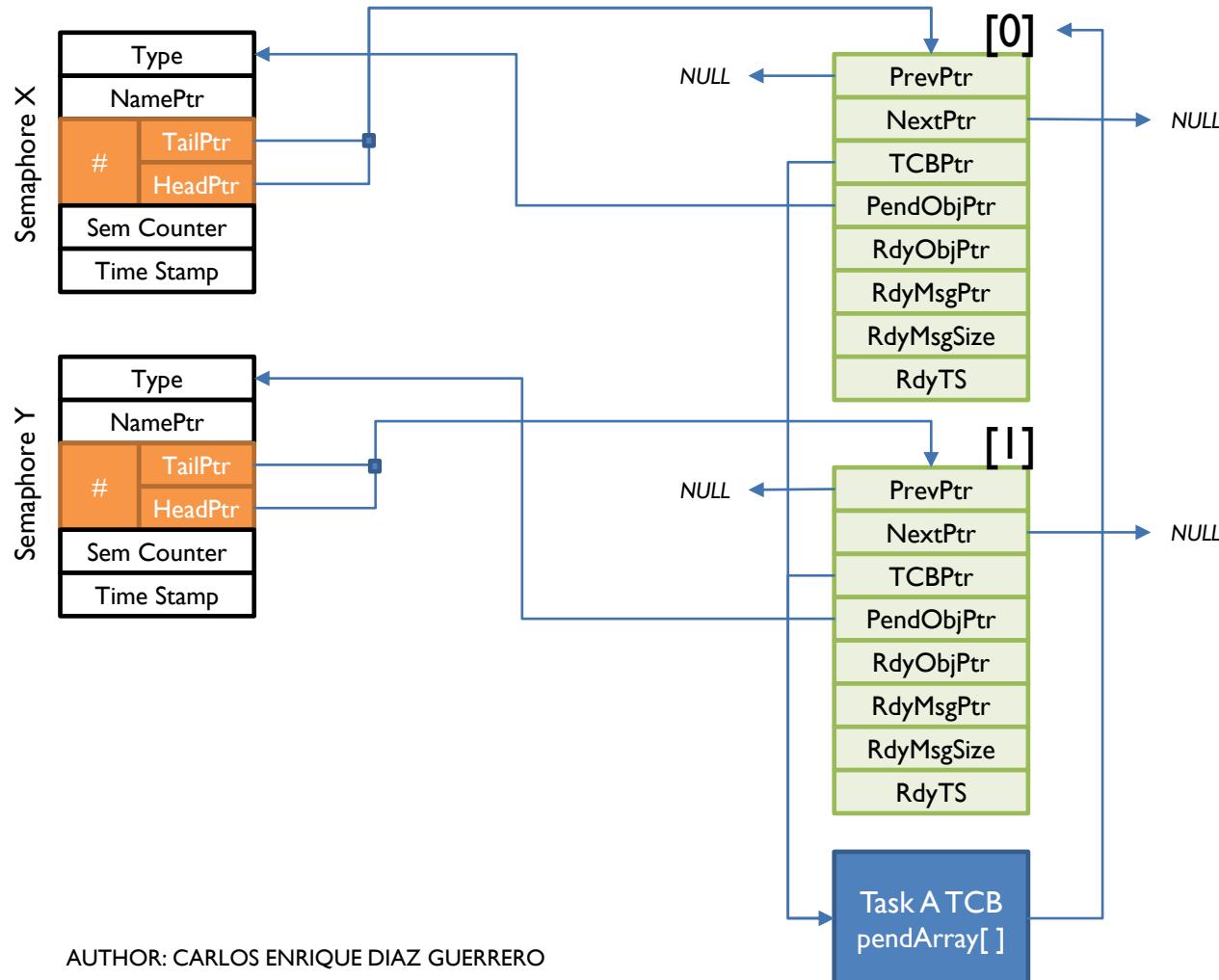


EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- When waiting for **multiple objects**:
 - An array of “**pending data**” items needs to be created.
 - The size of the array is the number of objects the task is waiting for
 - The array is created within the Task
 - An special API is used to link the “**pending data**” items to “**pending objects**”.
 - μC/OS III → OSPendMulti(...)

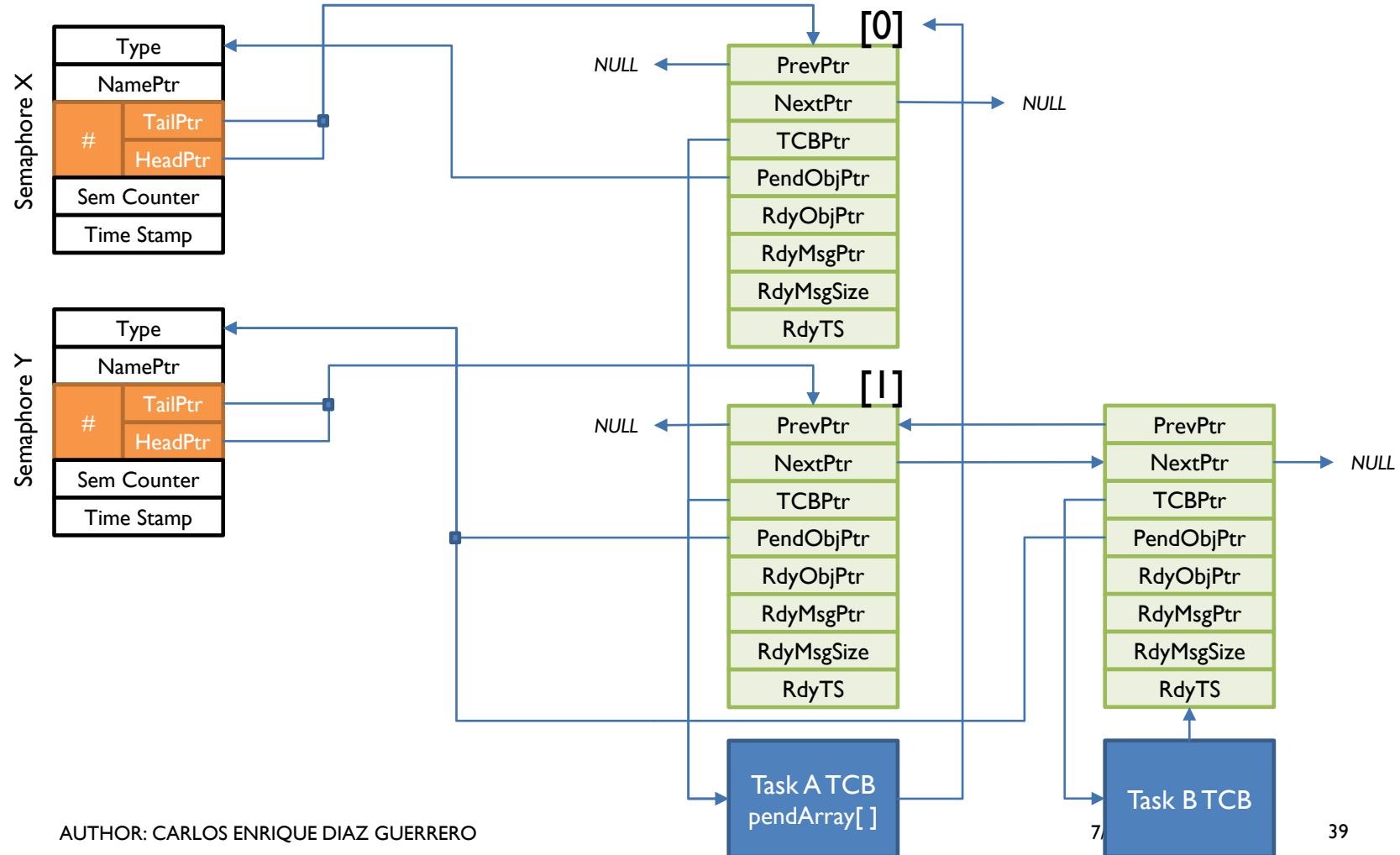
EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- **Example:** One Tasks (Task A) waiting for two **semaphore** (X and Y)



EVENT MANAGEMENT, MAILBOXES, MESSAGE QUEUES

- Example: What does this do??



OVERVIEW

- Real-Time Systems Concepts
- Real-Time Operating Systems Concepts
- Kernel Structure and Task Management
- Time Management
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- **RTOS Porting**
- References

RTOS PORTING

RTOS PORTING

- As any other software module, RTOS should follow good SW design practices:
 - RTOS code layered:
 - Increase maintainability
 - Improve portability
 - RTOS code focused on efficiency
 - Some code is directly written in assembly language
 - RTOS code shall reliable
 - Follow recommendations, standards, etc. E.g., OSEK following MISRA.

RTOS PORTING

- RTOS portability is indeed an important topic to be considered when structuring its code.
- An RTOS “**port**” is a version of the RTOS which has been adapted to work for an specific **compiler** and **hardware platform** (MCU).
- Most RTOS provide a manual with instructions to know which functions shall be adapted if a new port is to be created:
 - Which source files / functions have direct HW dependencies
 - Which source files / functions have direct compiler dependencies
 - Which source files / functions have direct board dependencies

RTOS PORTING

- RTOS have certain **MCU constraints** that need to be meet so the RTOS can be implemented over such MCU.
- Examples:
 - ANSI C compiler support for the given MCU
 - A peripheral able to provide a periodic interrupt (for clock tick, etc.)
 - Ability to enable/disable interrupts by SW
 - Facilities for having an Stack (Stack pointer, etc.)
 - Instructions to save and restore CPU registers (SP, PC, index, GPs etc.)
 - Sufficient amount of RAM/ROM to allocate the RTOS code

RTOS PORTING

- Normally the code of the RTOS provides a file structure that eases the creation of new ports.
 - A folder containing source code with CPU specific functions
 - A folder containing source code that configures the RTOS
 - A folder containing source code that provides the **Board Support Package (BSP)**

RTOS PORTING

- Example of CPU-dependant code
 - Symbols to determine the CPU characteristics
 - Endianess
 - Word-width of the CPU
 - CPU registers, ...
 - Interrupt controller characteristics
 - Some functions that can be provided by the MCU:
 - Count-Leading Zeros
 - Double SP to ease the change from Tasks Stack to Interrupt Stack, ...

RTOS PORTING

- Example of Compiler-dependant code
 - Specific compiler #pragma's
 - Handling of assembly instructions within the C code
 - Compiler optimizations
 - Compiler handling of interrupts code, etc.

RTOS PORTING

- Example of RTOS configurable items
 - Enabling/Disabling RTOS features
 - Semaphores
 - Events
 - Message queues / mailboxes
 - Statically allocated Tasks:
 - Tasks handlers
 - Tasks types
 - Tasks priorities, etc.
 - Clock Tick configuration → Clock Tick period, etc.
 - Interrupt handling configuration
 - Which interrupts to use as “kernel-aware” interrupts

RTOS PORTING

- Example of Board Support Package code
 - Handling of components in the “Board” or development kit
 - LEDs
 - Switches
 - Accelerometers, etc.
- BSP code is provided when the RTOS extends the hardware abstraction layer to the board level.
 - Eases the usage of the components included in the given board.

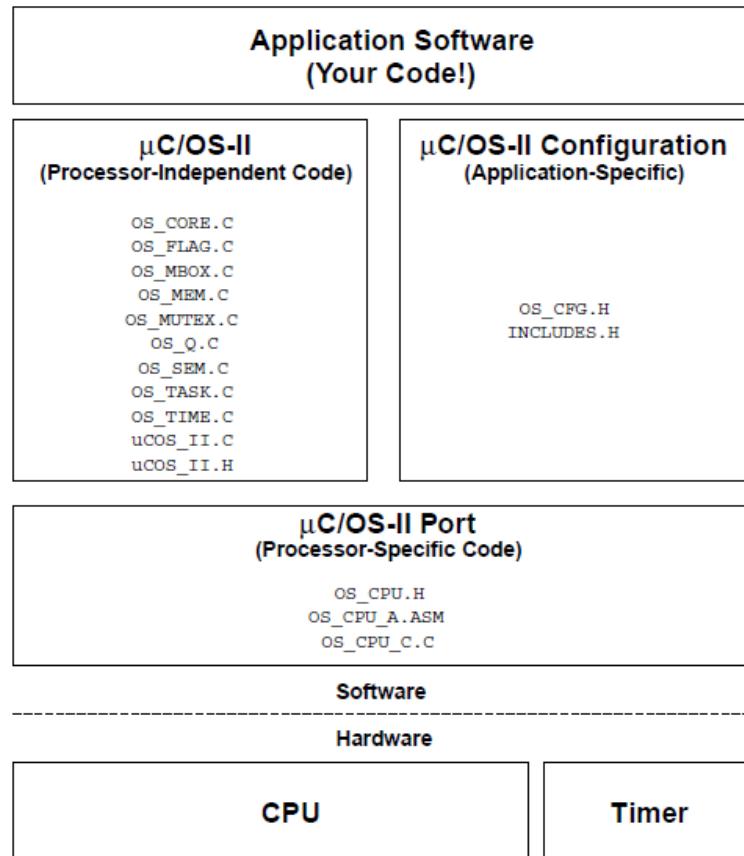
RTOS PORTING

- Folder structure example for FreeRTOS:

```
FreeRTOS
  |
  +-Source      The core FreeRTOS kernel files
    |
    +-include    The core FreeRTOS kernel header files
    |
    +-Portable   Processor specific code.
      |
      +-Compiler x  All the ports supported for compiler x
      +-Compiler y  All the ports supported for compiler y
      +-MemMang     The sample heap implementations
```

RTOS PORTING

- File structure example for μ C/OS II:



OVERVIEW

- Real-Time Systems Concepts
- Real-Time Operating Systems Concepts
- Kernel Structure and Task Management
- Time Management
- Semaphores and Mutual Exclusion
- Event Management, Mailboxes, Message Queues
- RTOS Porting
- **References**

REFERENCES

- William Stallings, “Operating Systems - Internals and Design Principles”, Seventh Edition, Prentice Hall, 2012
- Jean J Labrosse. “uC/OS-III, The Real-Time Kernel”, Third Edition, Micrium Press, 2009
- Jean J Labrosse. “uC/OS-II, The Real-Time Kernel”, Second Edition, CMP Books, 2002
- Richard Barry, “Using the FreeRTOS Real-Time Kernel – A practical Guide”, version 1.0.5, FreeRTOS.org, 2009
- Allen B. Downey, “The Little Book of Semaphores”, Second Edition, Green Tea Press, 2008
- OSEK/VDX steering committee, “OSEK/VDX Operating System Specification 2.2.3”, 2005
- Rieko, “Inside TOPPERS/ASP A Real-Time Operating System for Embedded Systems”, http://www.nces.is.nagoya-u.ac.jp/NEXCESS/blog_en/index.php?catid=4&blogid=4.
- Gerhard Fohler, “Operating Systems - 2011/2012 lecture slides”, Technische Universität Kaiserslautern
- Gerhard Fohler, “Real-Time Systems I - 2011 lecture slides”, Technische Universität Kaiserslautern
- Raphael Guerra, Gerhard Fohler, “Real-Time Systems II – 2011/2012 lecture slides”, Technische Universität Kaiserslautern