Real-time and Embedded Systems Concepts

Da-Wei Chang
CSIE, NCKU

Code and Pictures on the slides are from
MicroC/OS-II, The Real-Time Kernel (2nd Edition), by Jean J. Labrosse

Real-time Systems

- Cares about
  - Logical correctness
  - **Timing** correctness
    - deadline
- Predictable performance
- Two types of real-time (RT) systems
  - Soft vs. Hard *(described in the next slide)*

Soft RT vs. Hard RT

- Soft RT
  - Tasks are performed as fast as possible
- Hard RT
  - Must not miss the deadline
  - Severe consequences if deadline misses…

Embedded Systems

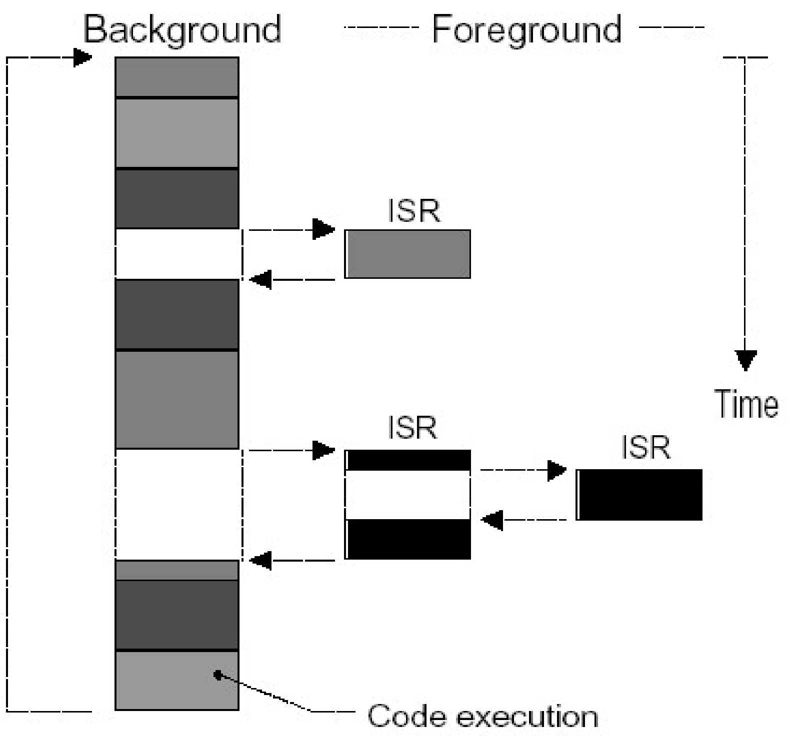
- Embedded system
  - A computer built into a system and not seen as being a computer
- Examples
  - Internet of Things
  - Robots
  - Automotive
  - Smart glasses
  - Drones
  - ….

Embedded Systems

- Three types of embedded software architecture
  - Foreground/background
  - Non-preemptive RTOS (kernel)
  - Preemptive RTOS (kernel)

Foreground/Background System

- Also called as ***super loops***
  - A main loop + *n* ISRs, where *n* >= 0
    - ISR = Interrupt Service Routine
- Used in low complexity, small systems



Tasks

- Tasks
  - Sometimes called as *threads*
  - Programs in execution
    - A sequence of codes in execution
  - Each task thinks that it owns the CPU

Foreground/Background System

- Background
  - Application with an infinite loop
  - Task level, a single task
- Foreground
  - Interrupt service routine (ISR)
  - Interrupt level
  - Handle asynchronous events
  - Used to perform critical operations

No RTOS in such a system!

Foreground/Background System

- The execution time of the application loop
  – Depends on the trigger situations of the interrupts and the ISR execution time
  – **Non-deterministic**

- A lot of micro-controller based systems are foreground/background systems

Non-preemptive or Preemptive RTOS/Kernel

- Responsible for managing tasks
- Ease application design
  - Provide services to allow creating multiple tasks
  - Provide services to allow schedule multiple tasks
  - …
- Has kernel overhead
  - Time overhead
    - Depends on how often you invoke the kernel services
  - Space overhead
    - Depends on what is included in the kernel

RTOS Kernels

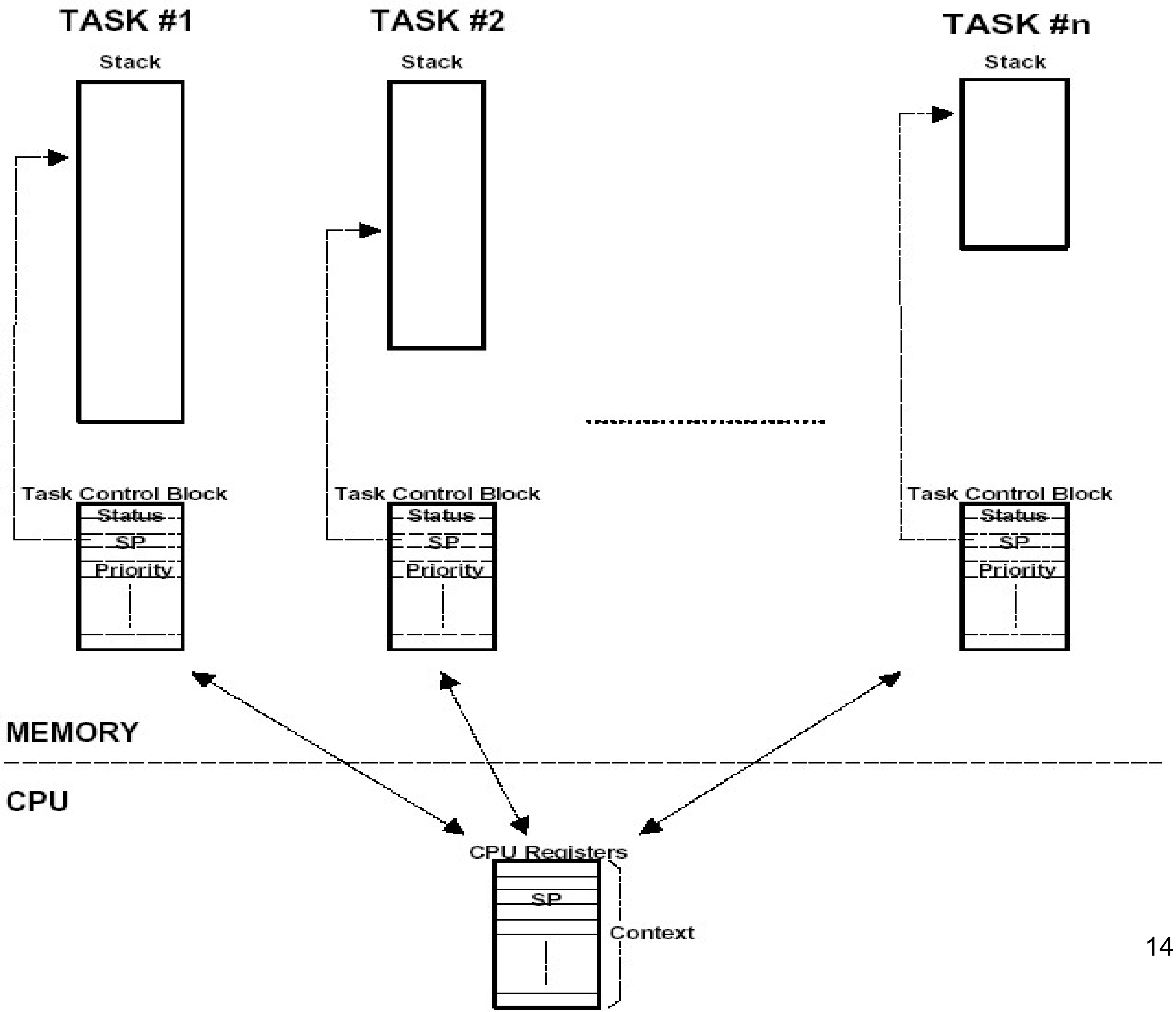
- Real Time and Embedded Kernels
  - FreeRTOS
  - MicroC/OS-II, MicroC/OS-III
  - Cs/OS2, Cs/OS3 (Cesium RTOS)
  - VxWorks
  - QNX
  - RT-Thread
  - ThreadX
  - Zephyr
  - RIOT
  - Mbed
  - …

MultiTasking

- Running multiple tasks concurrently
  - Dynamically switching one task to another
- Benefits
  - Maximize the utilization of the CPU
  - Provide for modular construction of applications
    - Splitting application jobs into multiple tasks➔ applications are easier to design and maintain
- Each task thinks that it owns the CPU

Tasks

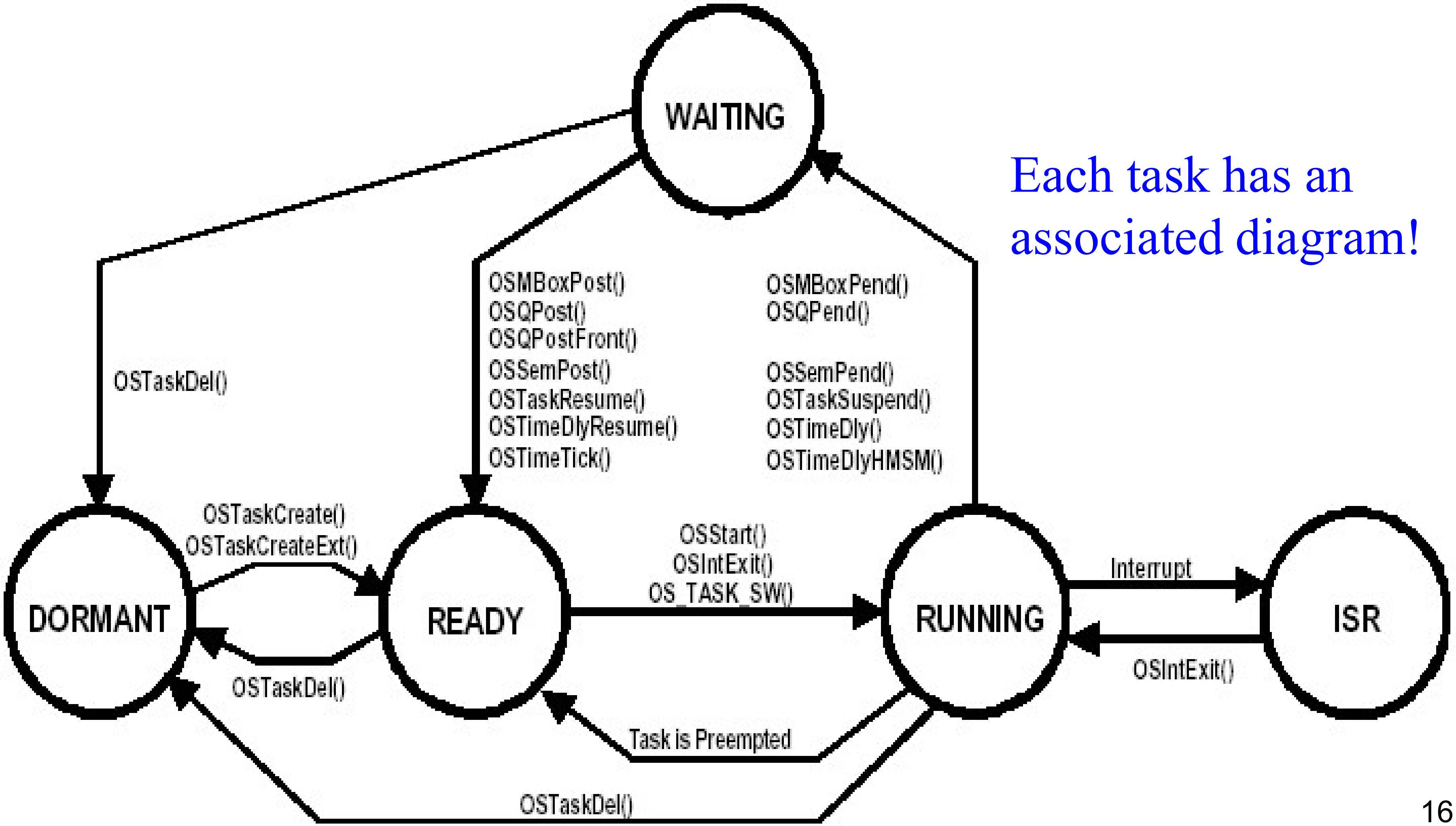
- Each task has the following
  - Priority
  - Stack
  - State
  - Space for storing values of CPU registers
    - For some systems, the space may be in the task's stack


TASK #1
Stack
TASK #2
Stack
TASK #n
Stack
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context

Context Switches

- Also called *task switches*
- Switch the CPU from one task to another
- Context switch jobs (from task A to task B)
  - Save context in A's stack
  - Restore context from B's stack
- Overhead is related to
  - Context size
  - Switching frequency

Task States



Each task has an associated diagram!

Schedulers

- A part of OS kernel
- Determines which task to run next
- Usually based on **priority**
  - The highest-priority task gets the CPU
  - In many cases, ***assigning priority*** is the job of the application designers.
- Priority-based scheduling algorithm
  - static priority vs. dynamic priority
  - For same-priority tasks
    - First-In-First-Out (FIFO), Round-Robin (RR)
- Other scheduling algorithms
  - Rate monotonic (RM), Earliest-Deadline-First (EDF)

Assigning Task Priorities

- In a general priority-based scheduling system, **assigning task priorities may not be an easy job** if you have many tasks
  - Needs to be considered carefully

Schedulers

- When to perform re-scheduling?
  - Different in preemptive and non-preemptive kernels

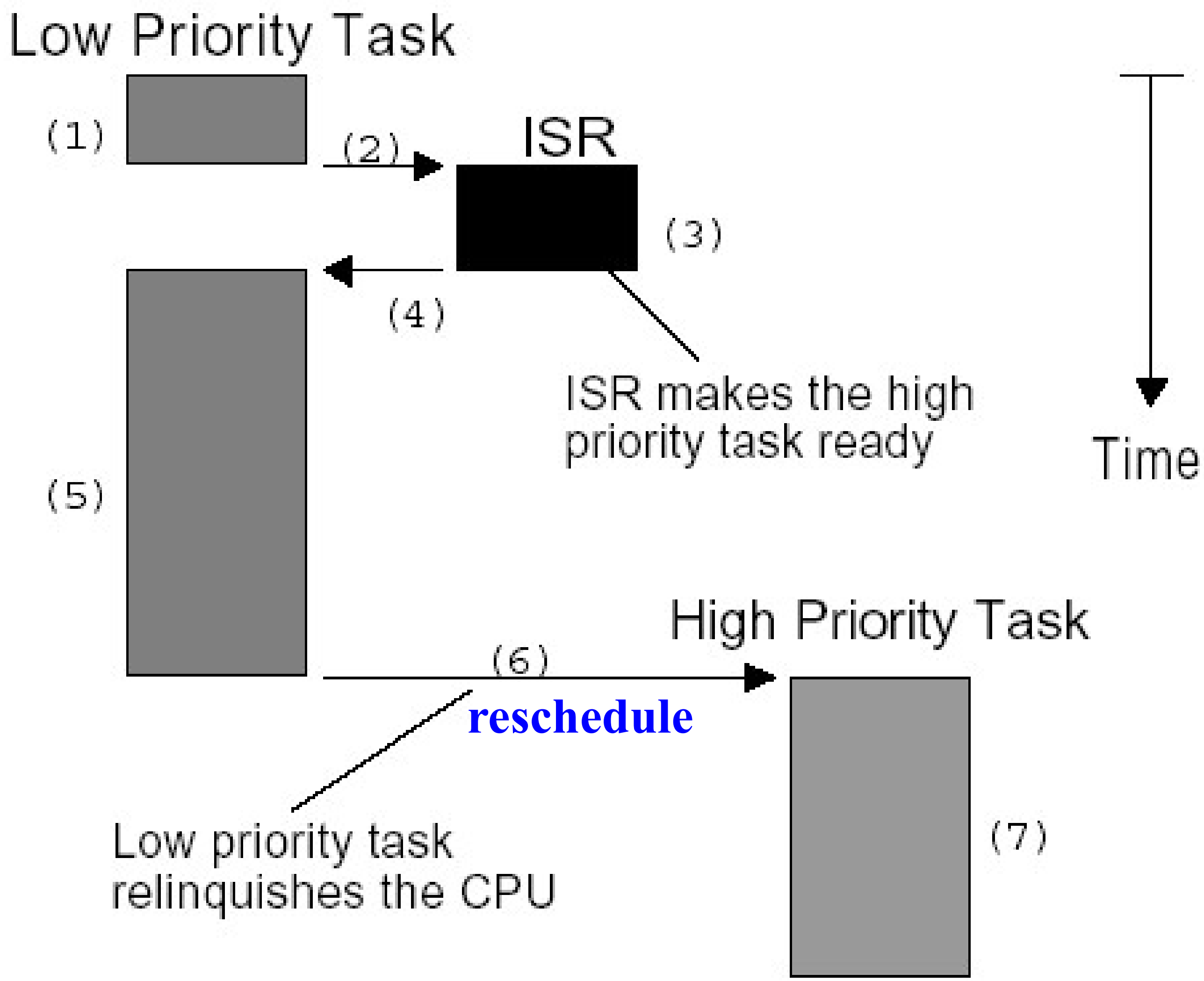
Non-Preemptive Kernels

- Tasks should explicitly give up the CPU to allow rescheduling
  - Cooperative multi-tasking
  - e.g., Win95
- Asynchronous events are still handled by ISRs
  - ISR can preempt current task
  - ISR can make a higher-priority task ready
  - But, ISR still returns to the interrupted task
  - *See the next slide*

\- Less responsive

\+ Less overhead to guard shared resources

Non-Preemptive Kernels



Preemptive Kernels

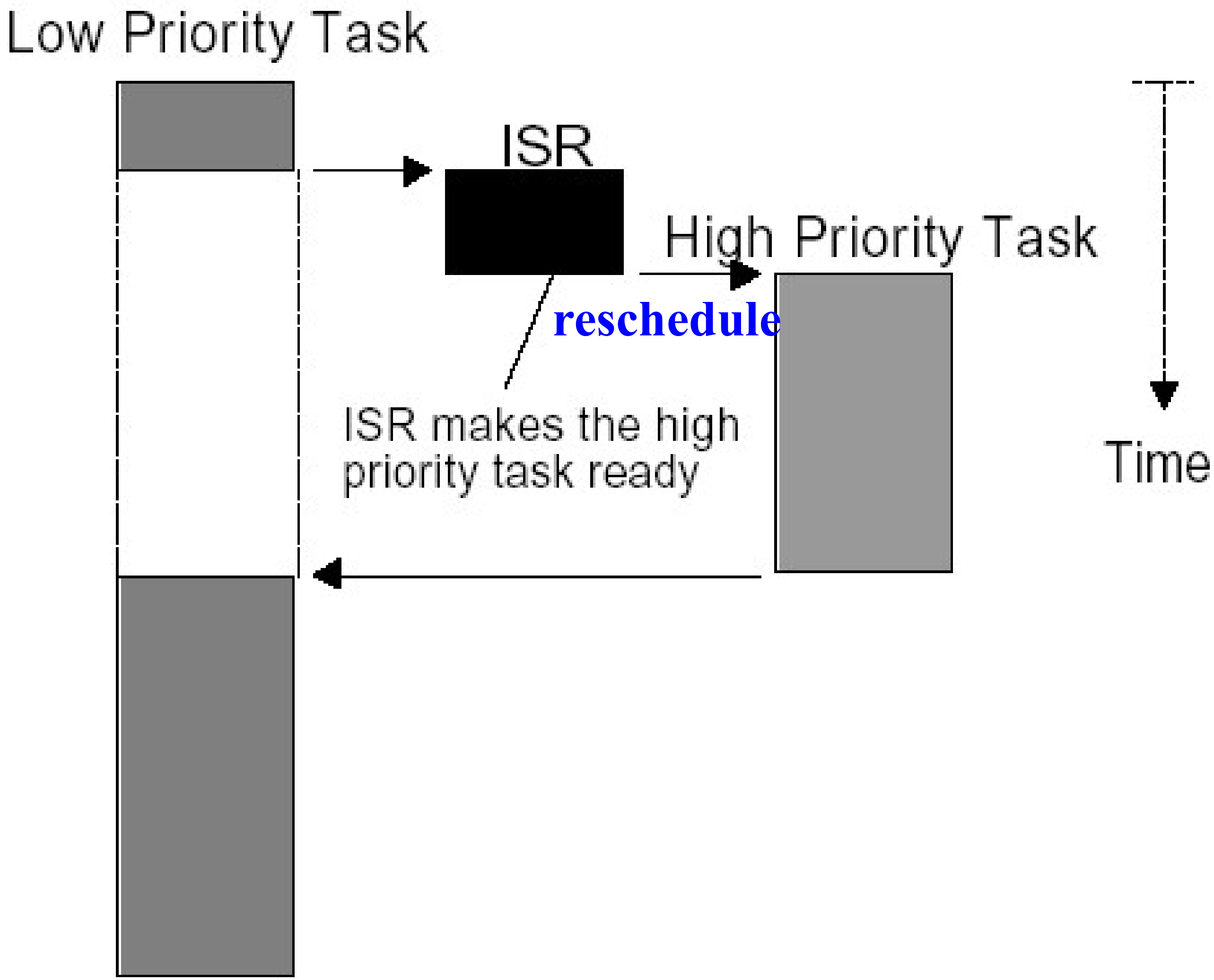
- Tasks are forced to give up CPU when needed
- Whenever a higher priority task is ready, the current task is preempted
  - *See the next slide*

\+ More responsive

\- More overhead to guard shared resources

- Most RTOSs are preemptive kernels

Preemptive Kernels



Preemptive Kernels

- On preemptive kernels
  - it is easier to determine when the highest priority task can get the CPU

Reentrant vs. Non-Reentrant Functions

```
void strcpy(char *dest, char *src)
{
     while (*dest++ = *src++) {
          ;
     }
     *dest = NUL;
}
```

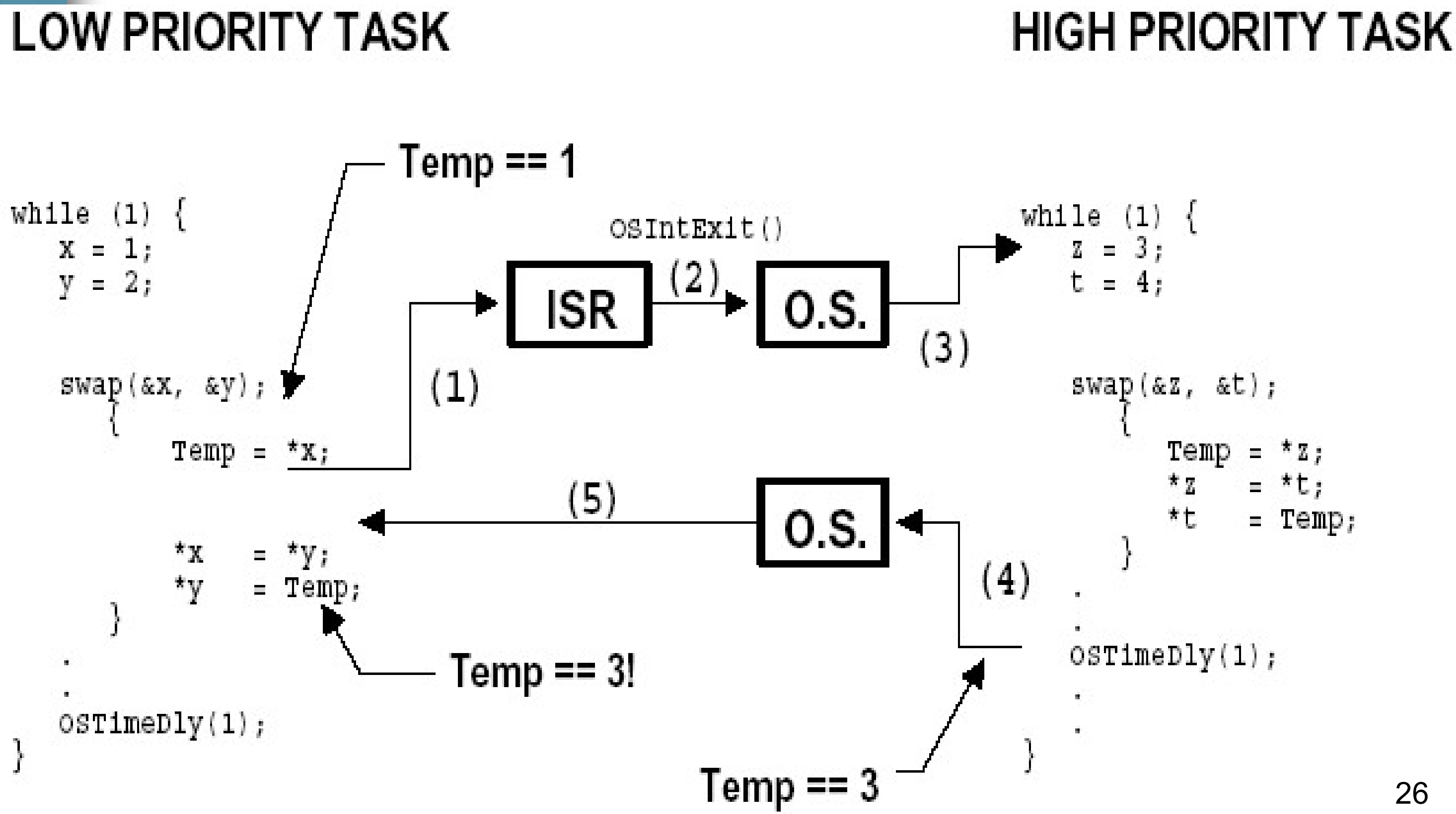
Reentrant

```
int Temp;

void swap(int *x, int *y)
{
     Temp = *x;
     *x   = *y;
     *y   = Temp;
}
```

Non-Reentrant

An Example of Non-Reentrant Function



Making a Function Reentrant?

- Use local variables only

Resources

- Any entity used by a Task/ISR
  - Physical resources
    - e.g., IO devices like printer, keyboard,…
  - Logical resources
    - e.g., data structures like variables, arrays, structures…

Resources

- Shared resources
  - – Resources used/accessed by more than one tasks/ISRs
- **Mutual exclusion**
  - At any given time, only one task can access the (shared) resource

Critical Sections

- A sequence of code that accesses one or more shared resources
- Ensure ***mutual exclusion***

- Approaches for ensuring mutual exclusion
  – Disable interrupts
  – Use test-and-set operations
  – Disable scheduling
  – Use semaphores

Mutual Exclusion

- Disable interrupts
  - OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL() in uC/OS-II
  - Issues
    - Which interrupts?
    - Which interrupt levels?
    - Save interrupt state before disabling?

Mutual Exclusion

- Test-and-set operations
  - Test a global variable
    - 0 ➔ access resource, 1: don’t access
  - Have to Set the variable when testing its value
- Implementation method 1
  - test-and-set or similar hardware instructions
- Implementation method 2
  - Disable interrupt for accessing the variable

Mutual Exclusion

- Disable Scheduler
  - *Less performance impact* than disabling interrupts
  - Rescheduling **does not** occur even if a higher priority task is made ready by an ISR
    - ISR still returns to the interrupted task
    - Similar to non-preemptive kernel
  - Works if the task does not share resources with ISRs
  - In uC/OS-II, the related API are
    - OSSchedLock()/OSSchedUnLock()

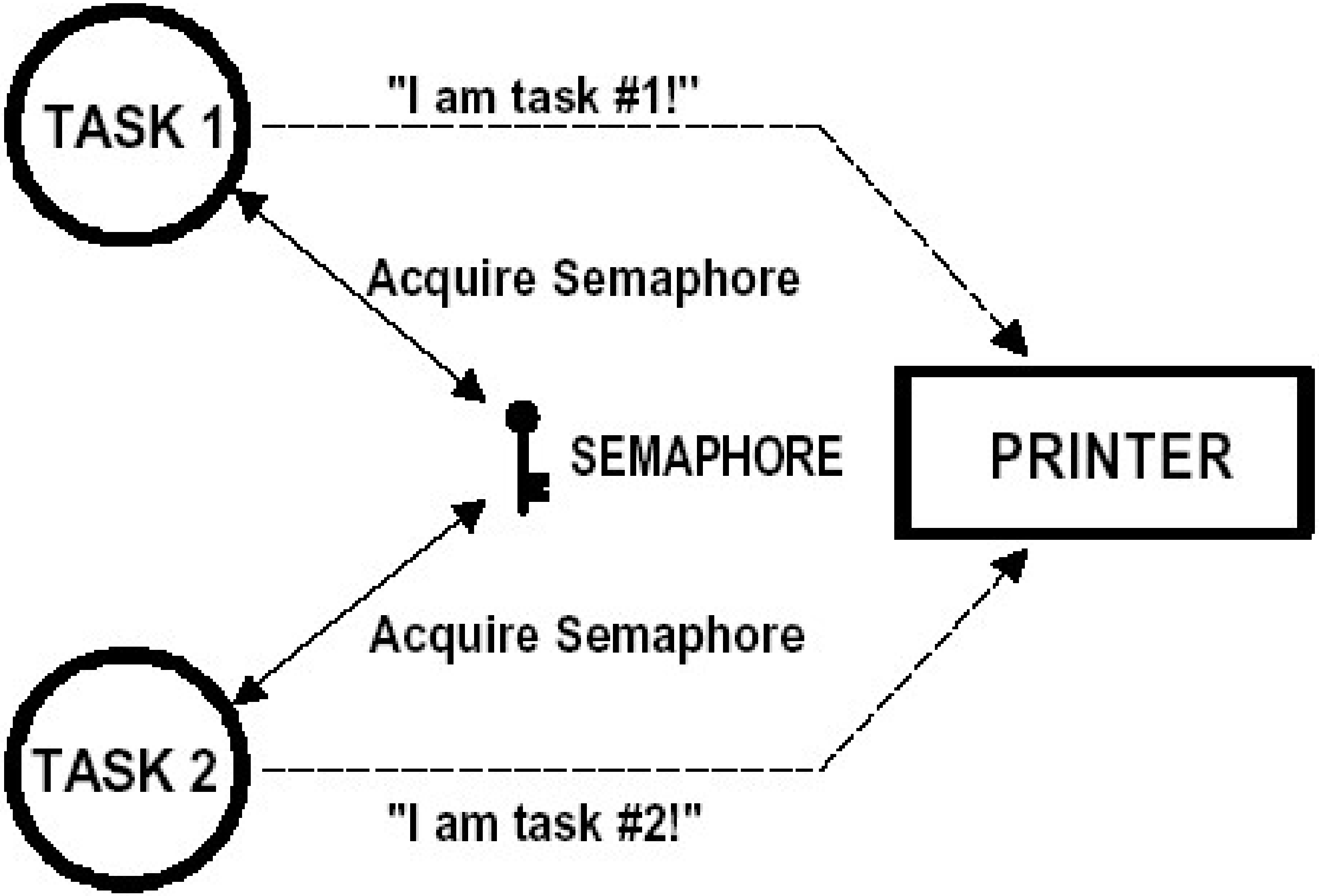
Mutual Exclusion

- Semaphores
  - Used to
    - Control access to a shared resource
    - Signal the occurrence of an event
    - Task synchronization
- Binary vs. Counting
- For Mutual Exclusion
  - Acquire the semaphore (i.e., key) before accessing the resource
  - Release the semaphore after accessing the resource

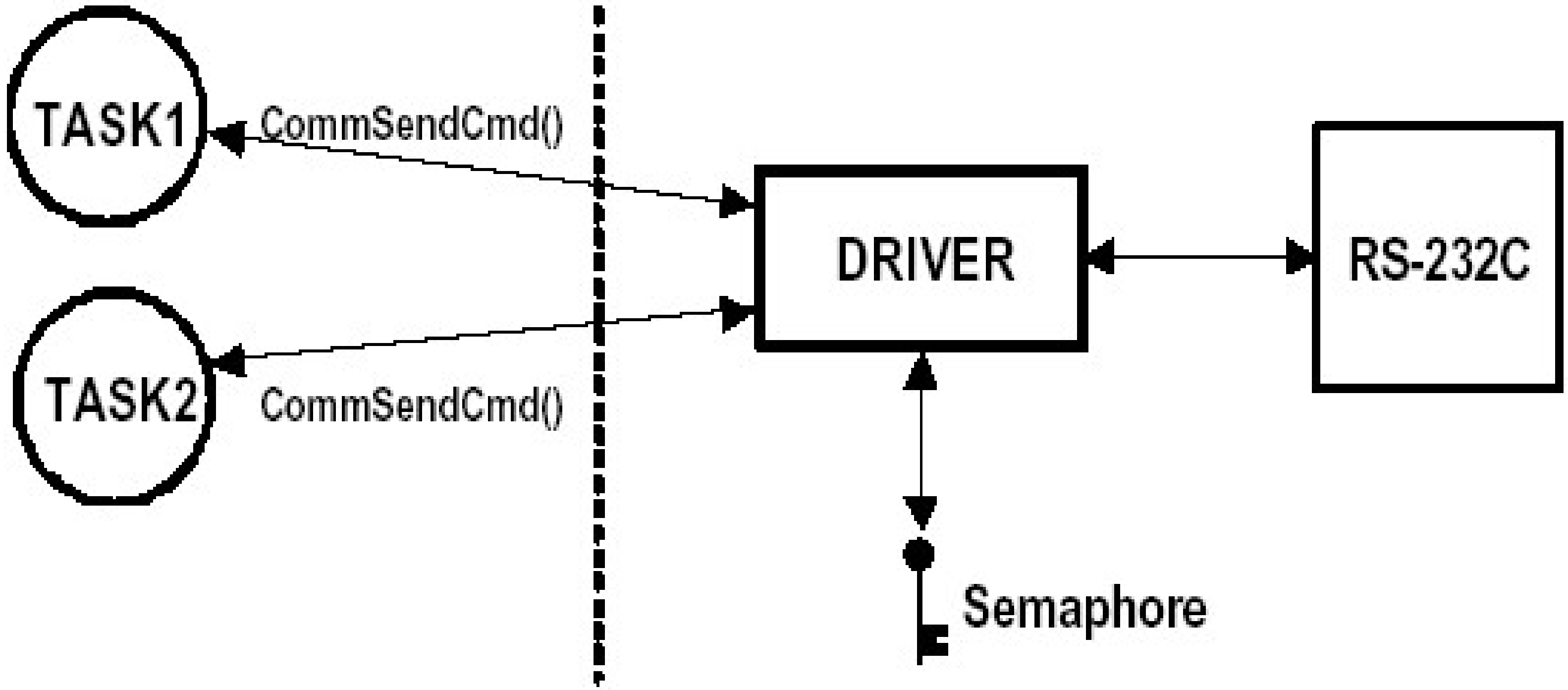
Semaphores

- Operations
  - **Wait** (acquire)
  - **Signal** (release)
- Wait may cause suspension/sleep
  - When waiting a semaphore with value 0
- Signal may cause some tasks to be resumed/wake-up
  - Wake up the first suspended task
  - Wake up the highest-priority task ←uC/OS-II

A Semaphore Example

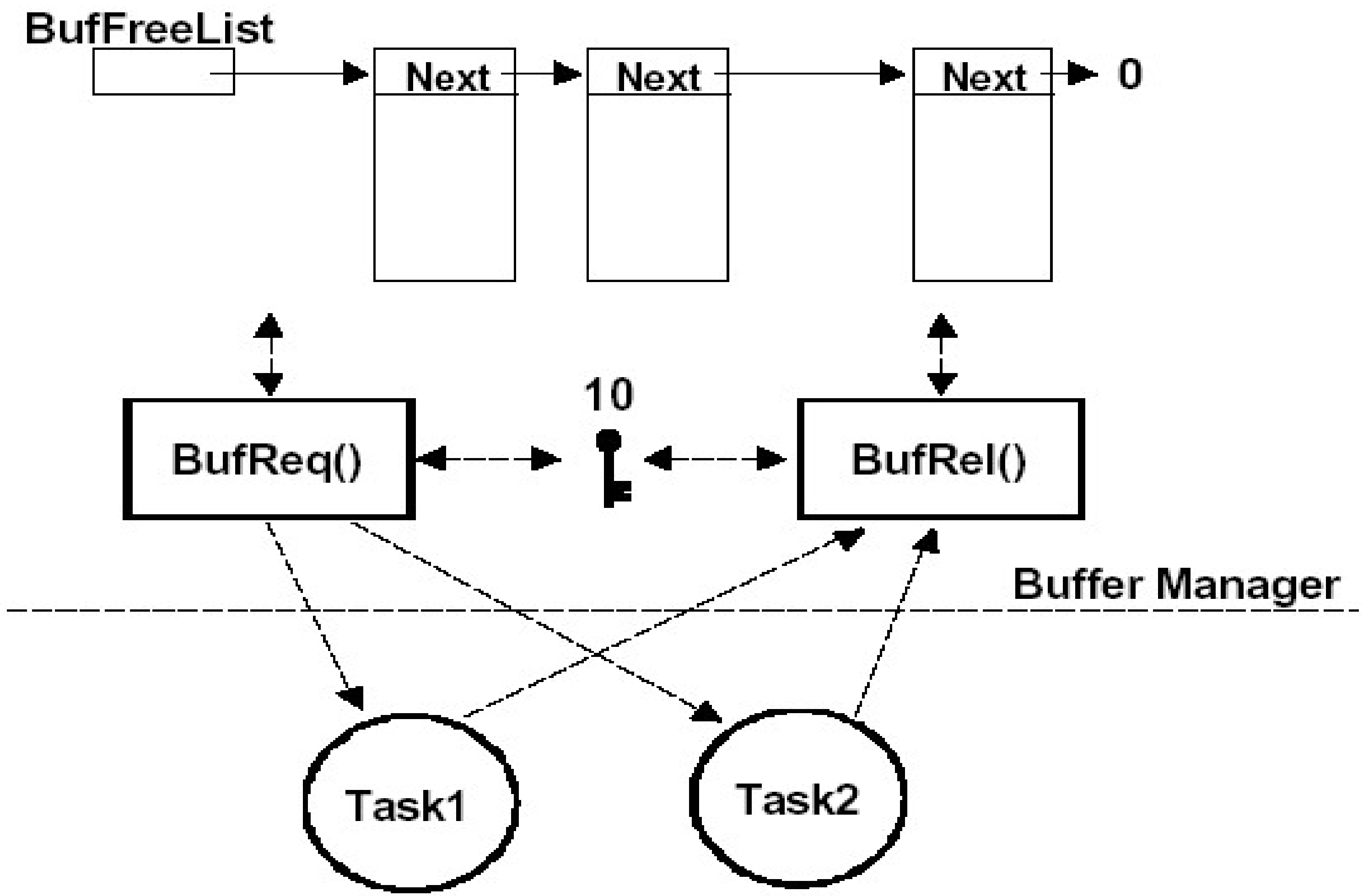


Encapsulating Semaphore Operations



Semaphore operations are encapsulated in CommSendcmd()

Another Semaphore Example



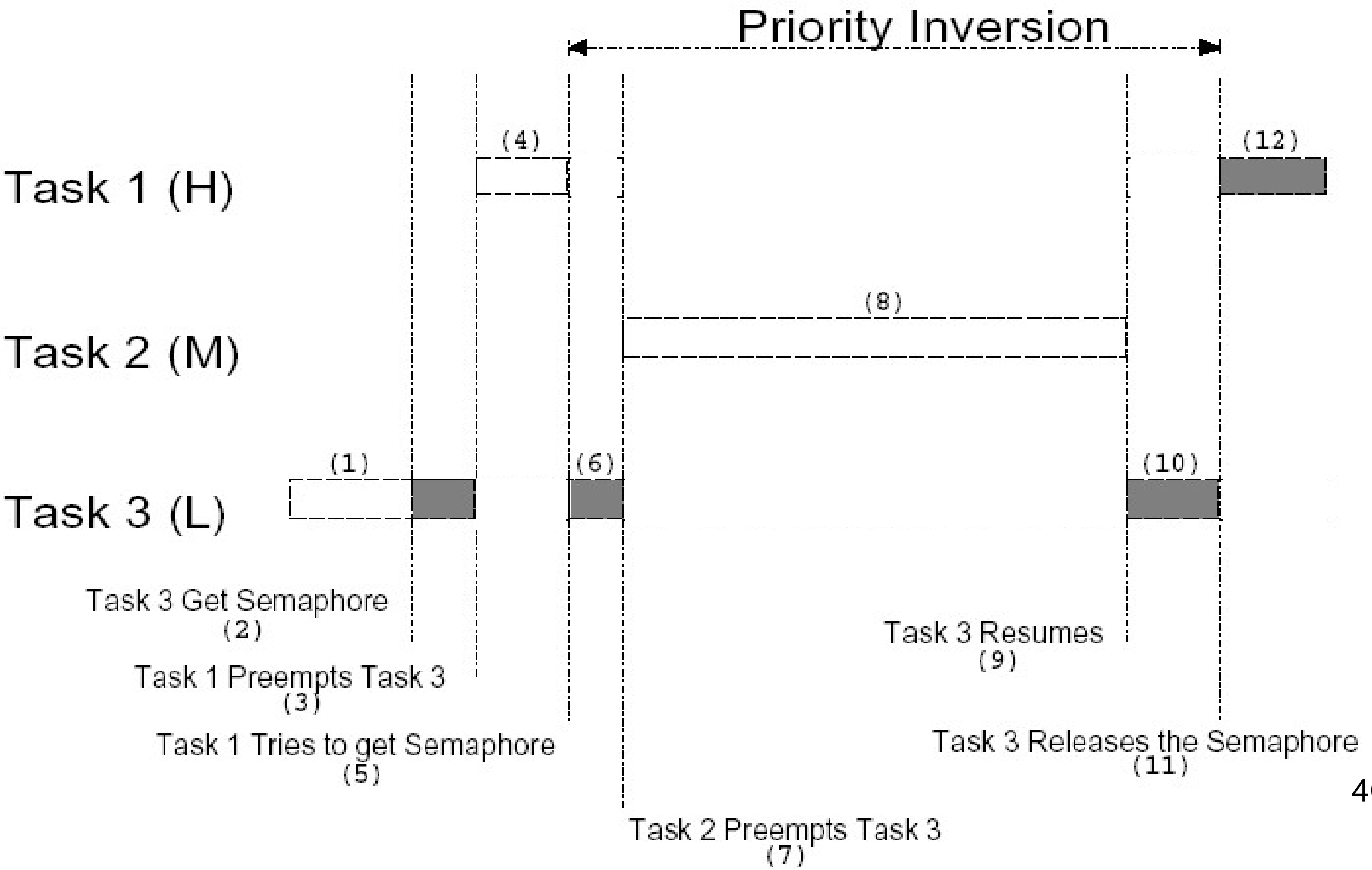
Counting semaphore with value = 10 (i.e., 10 keys)

Deadlock

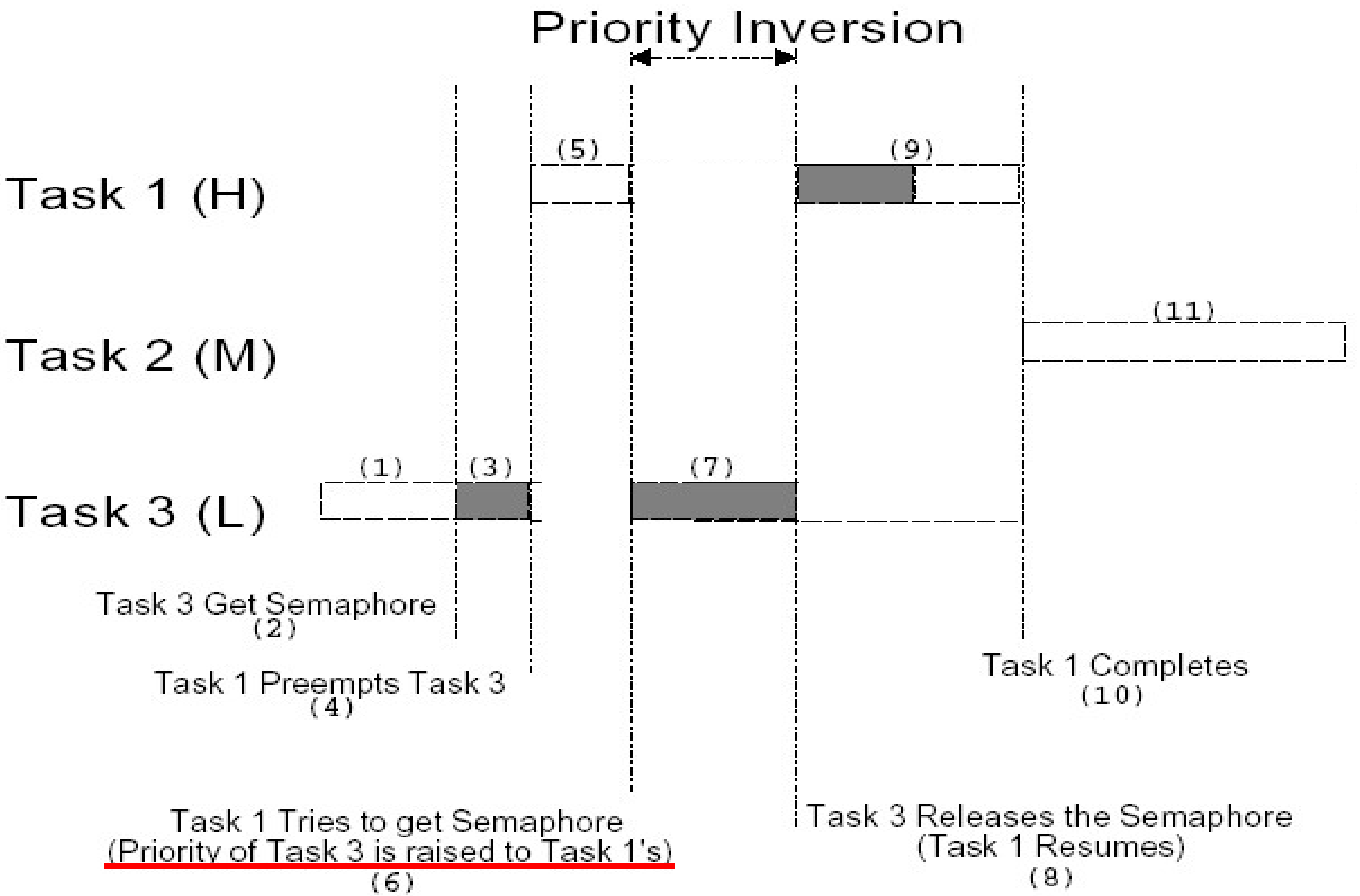
- Two or more tasks wait for resources held by the others
  - Causing system hangs
- Prevent deadlock
  - Acquire all resources before proceeding
  - Acquire resources in the same order
- Breaking deadlock
  - Timeout-based waiting
    - Widely supported in RTOSes!

Priority Inversions

- A problem in RT systems



Priority Inheritance Protocol (PIP)

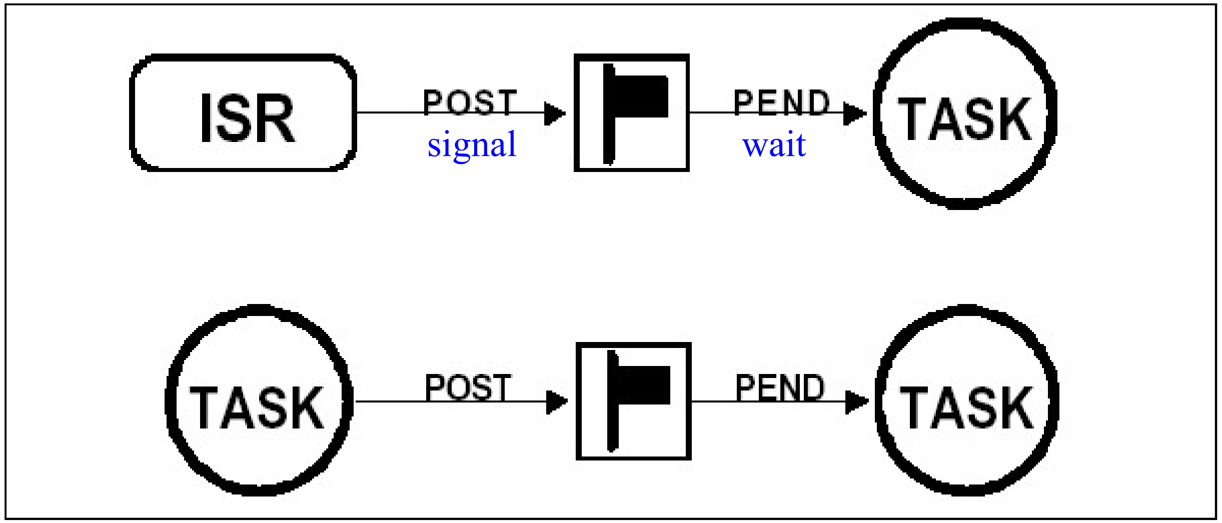


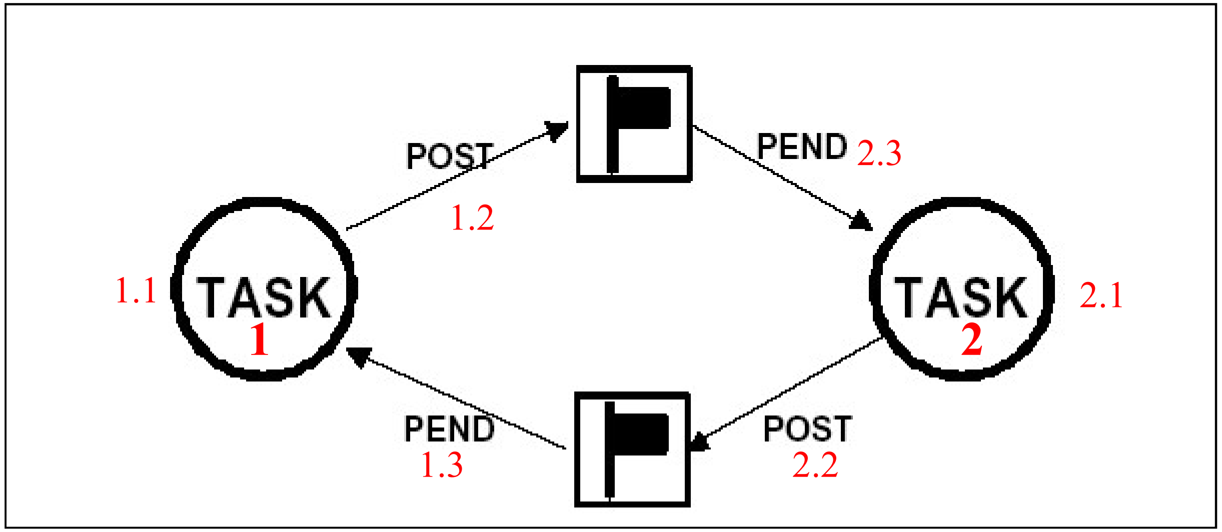
Some level of priority inversion can not be avoided!!

Synchronization

- Task-Task, Task-ISR
- Wait for an event to occur
- Approaches
  - Semaphores
  - Event flags

Synchronization -- Semaphore





Synchronization -- Semaphore

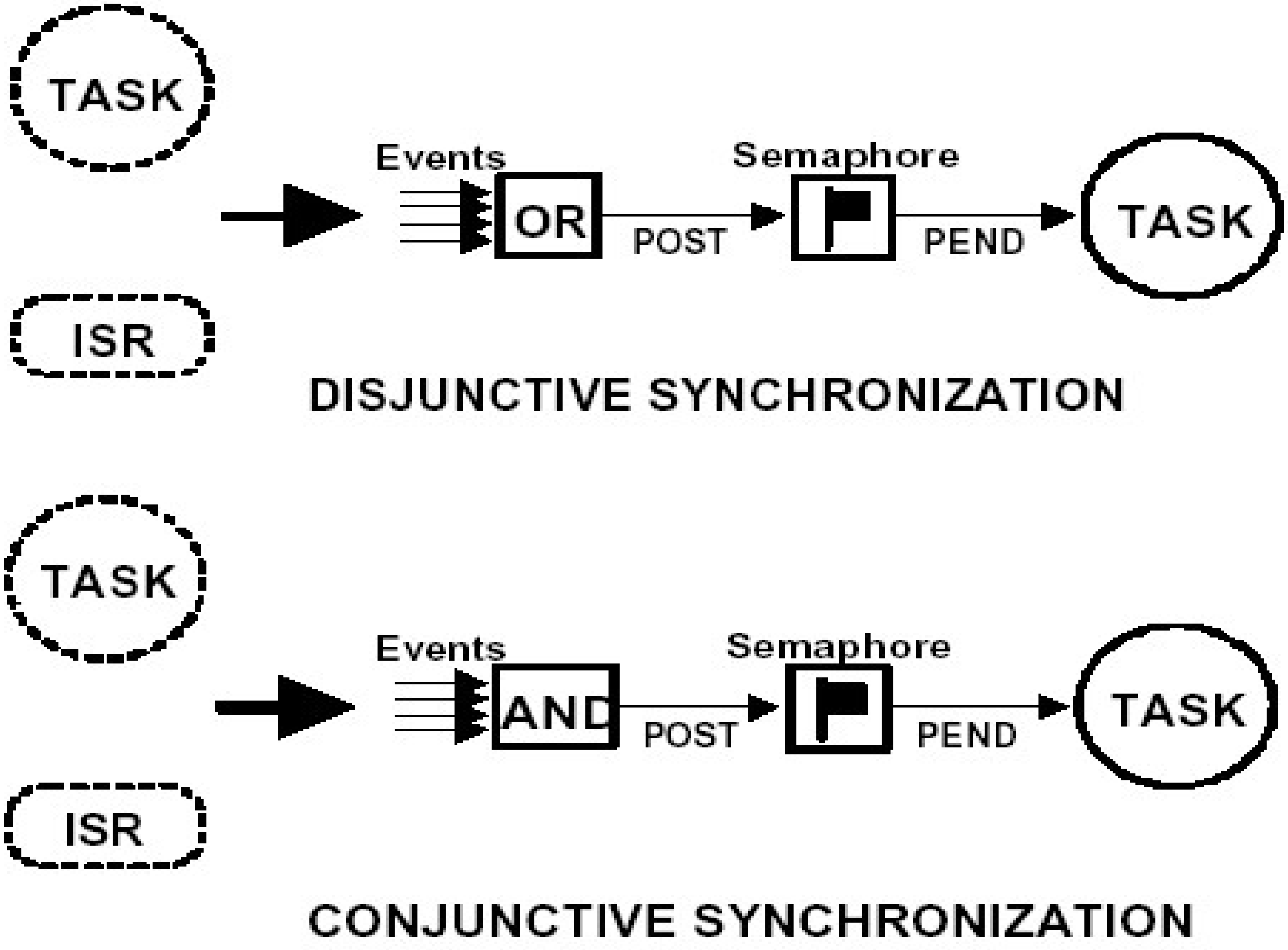
```
Task1()
{
     for (;;) {
        Perform operation;
        Signal task #2;                          (1)
        Wait for signal from task #2;            (2)
        Continue operation;
     }
}


Task2()
{
     for (;;) {
        Perform operation;
        Signal task #1;                          (3)
        Wait for signal from task #1;            (4)
        Continue operation;
     }
}
```

Synchronization – Event Flags

- Synchronize with the occurrence of multiple events
- You can define different eventsTypes
  - Disjunctive (OR)
  - Conjunctive (AND)

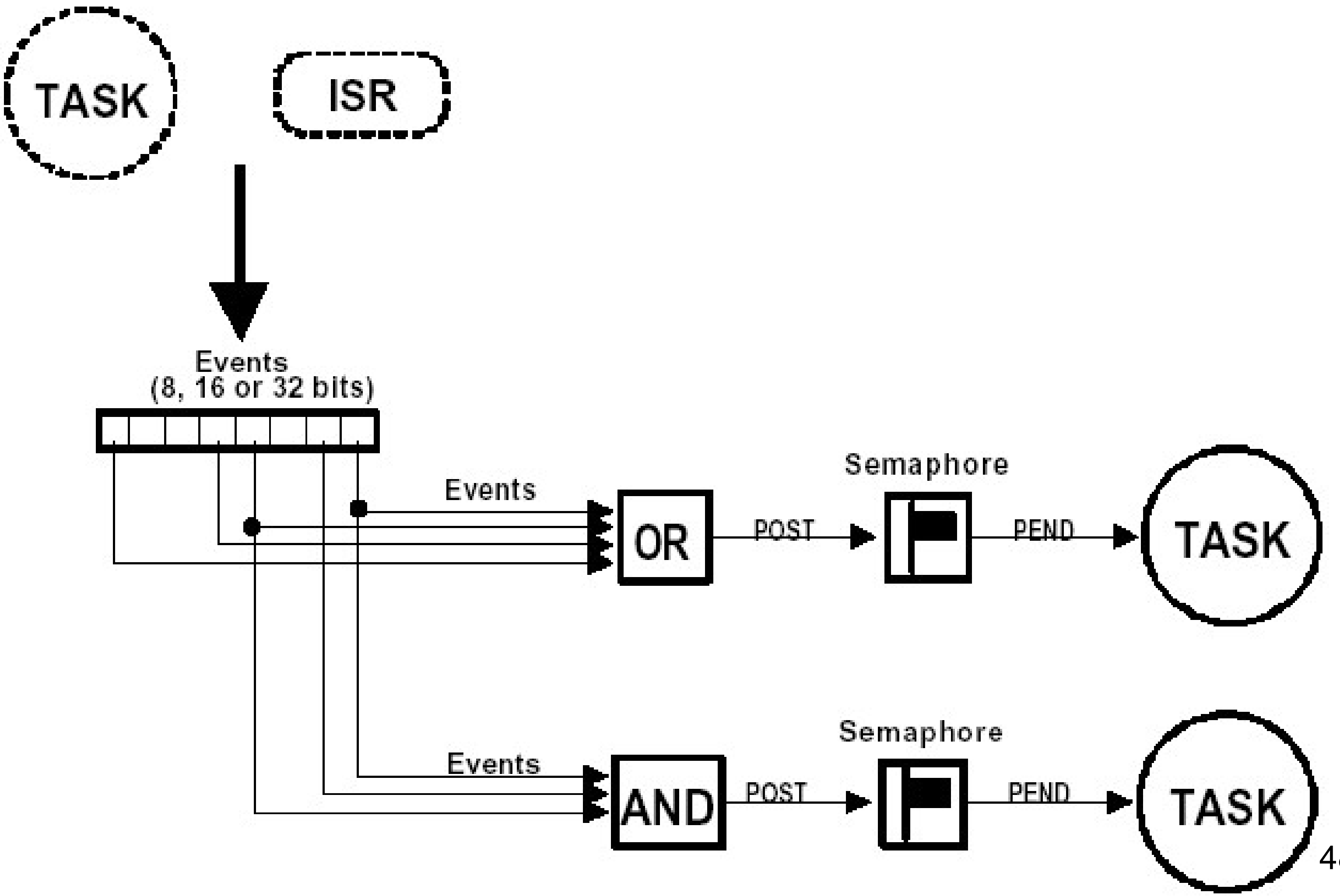
Synchronization – Event Flags



Synchronization – Event Flags

- Events are grouped
  - Usually represented as ***bitmaps***
  - Task/ISR can set or clear events
  - *See next slide*

Synchronization – Event Flags



Inter-Task Communication

- For a task or an ISR to exchange information with another task
- Approaches
  - Global variables
    - Must ensure exclusive access
  - Mailboxes/Message queues
    - The most widely used inter-task communication mechanism in an RTOS

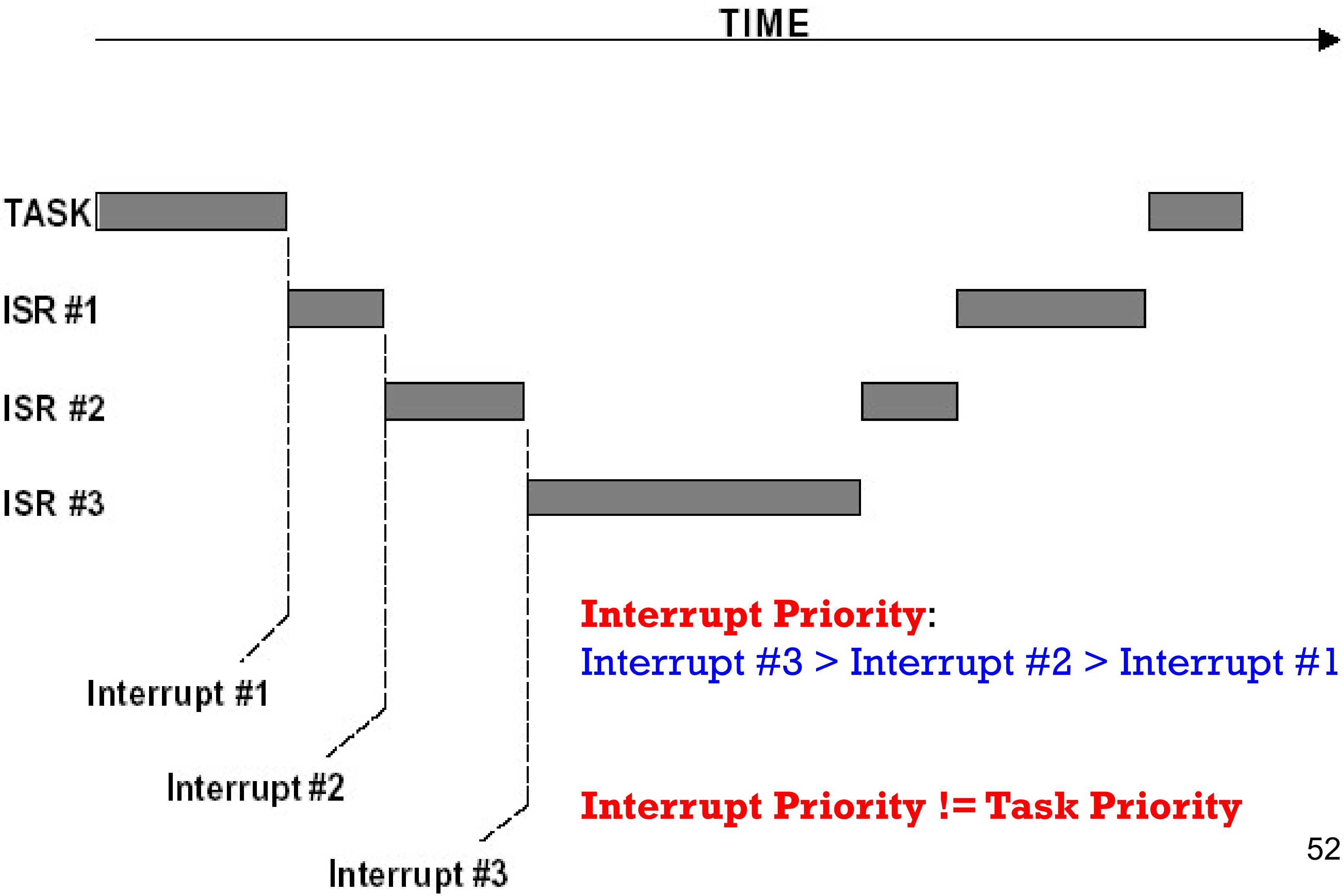
Mailbox/Message Queues

- A message-exchanging facility
- Can contain one or more messages
  - In uC/OS-II,
    - A mailbox can have only one message
    - A message queue can have more than one messages
- One or more tasks can receive on a mailbox
- Send message to a mailbox, not a specific task
  - Indirect communication
- A task may suspend when it wants to receive a message from an empty mailbox
  - Wakeup when a message comes

Interrupts

- Inform the CPU that an asynchronous event has occurred
- Improve CPU efficiency
  - Avoid polling devices
- After recognizing the interrupt, the CPU
  - Save (part of the) context
  - Jump to ISR (Interrupt Service Routine)
- When ISR is returned, control jumps back to
  - The interrupted task (non-preemptive-kernel)
  - The highest-priority task (preemptive-kernel)

Nested Interrupt



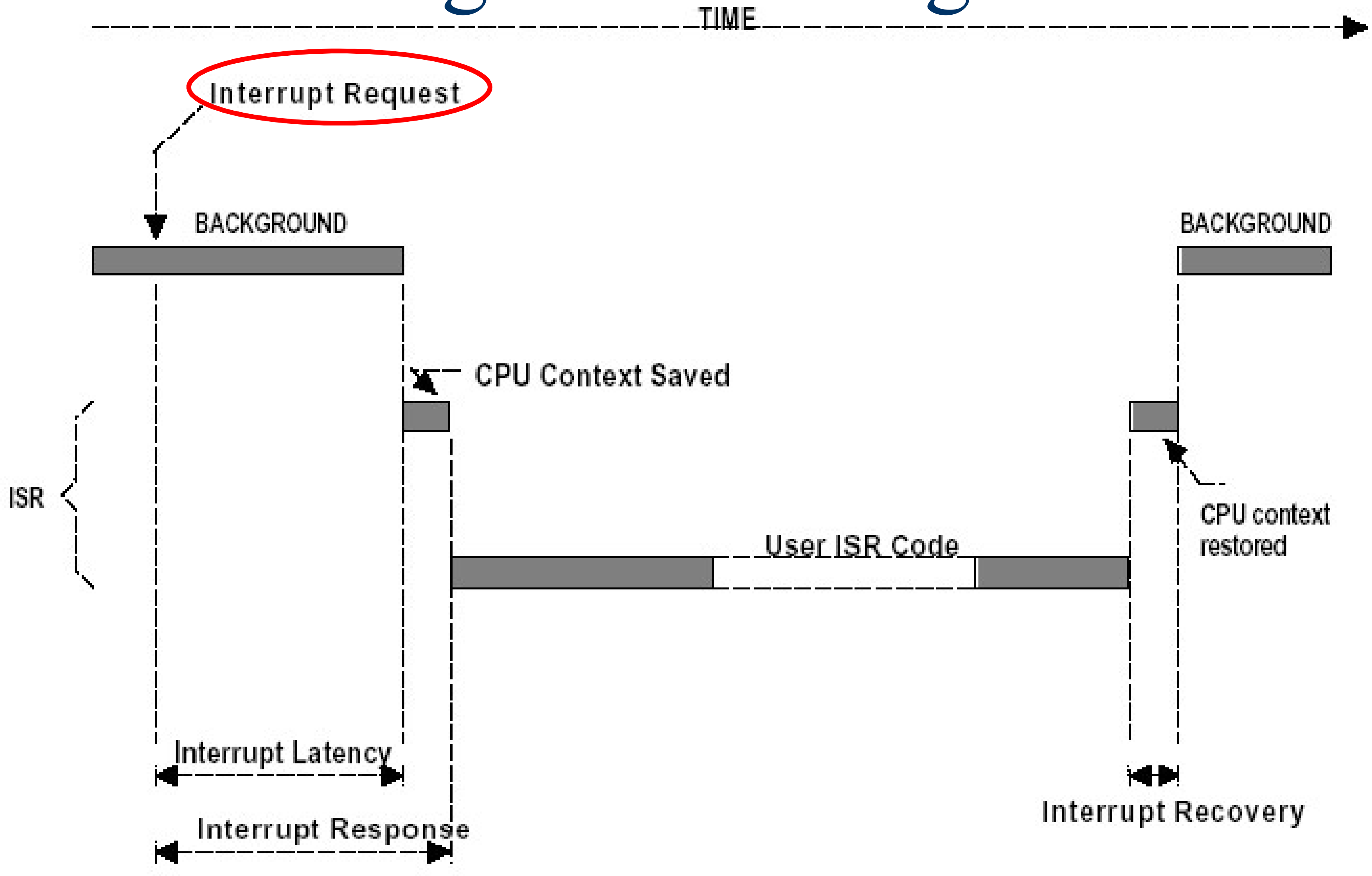
Interrupt Priority:
Interrupt #3 > Interrupt #2 > Interrupt #1

Interrupt Priority != Task Priority

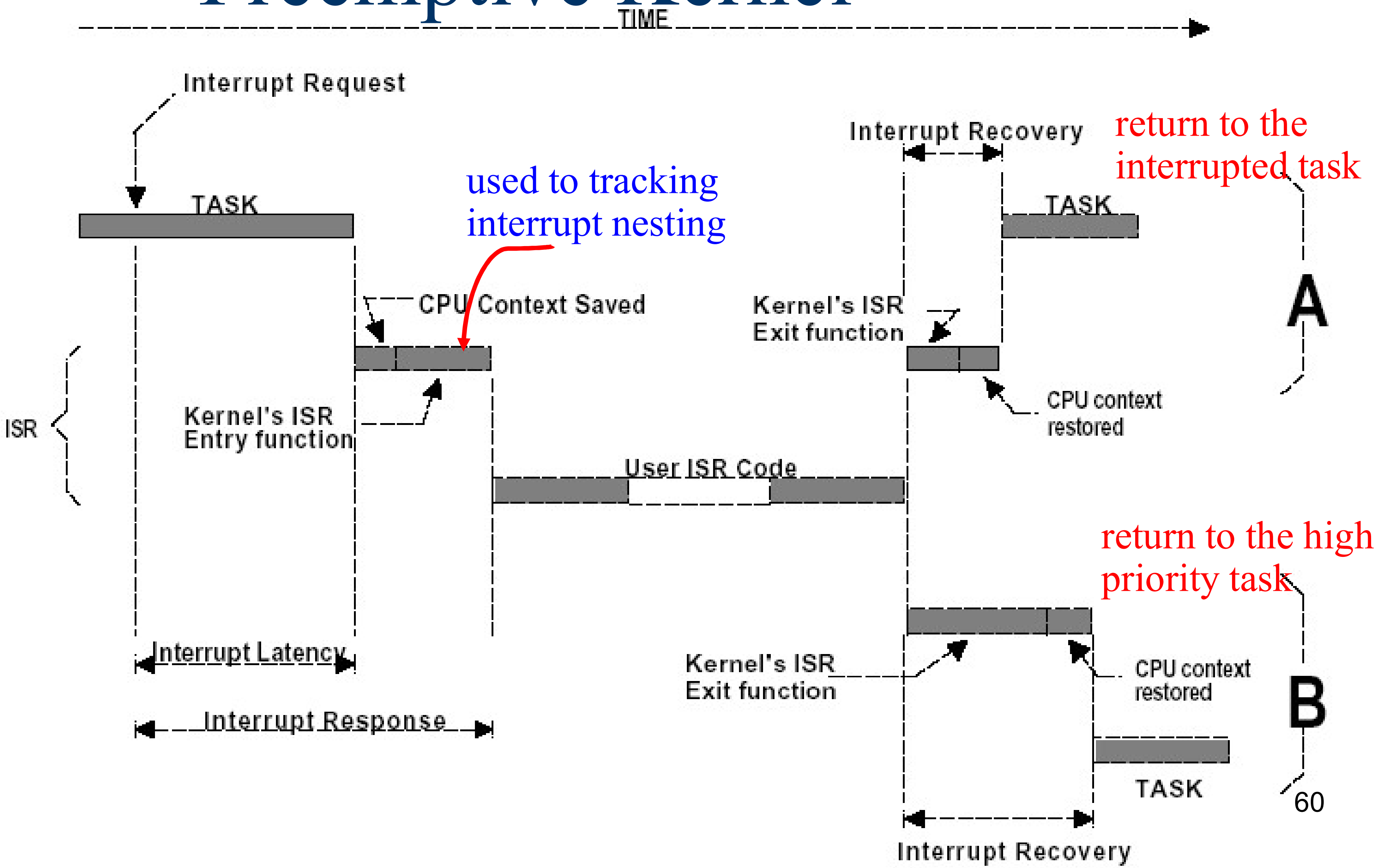
Interrupt Related Time

- Interrupt Latency
  - Between interrupt occurrence and the time to start executing the first ISR instruction
- Interrupt Response
  - Between interrupt occurrence and the start of the **user ISR code** that handle the interrupt
- Interrupt Recovery
  - Time required for the CPU to return to the interrupted/highest-priority task

Interrupt-Related Time -- Foreground/Background



Interrupt-Related Time -- Preemptive Kernel



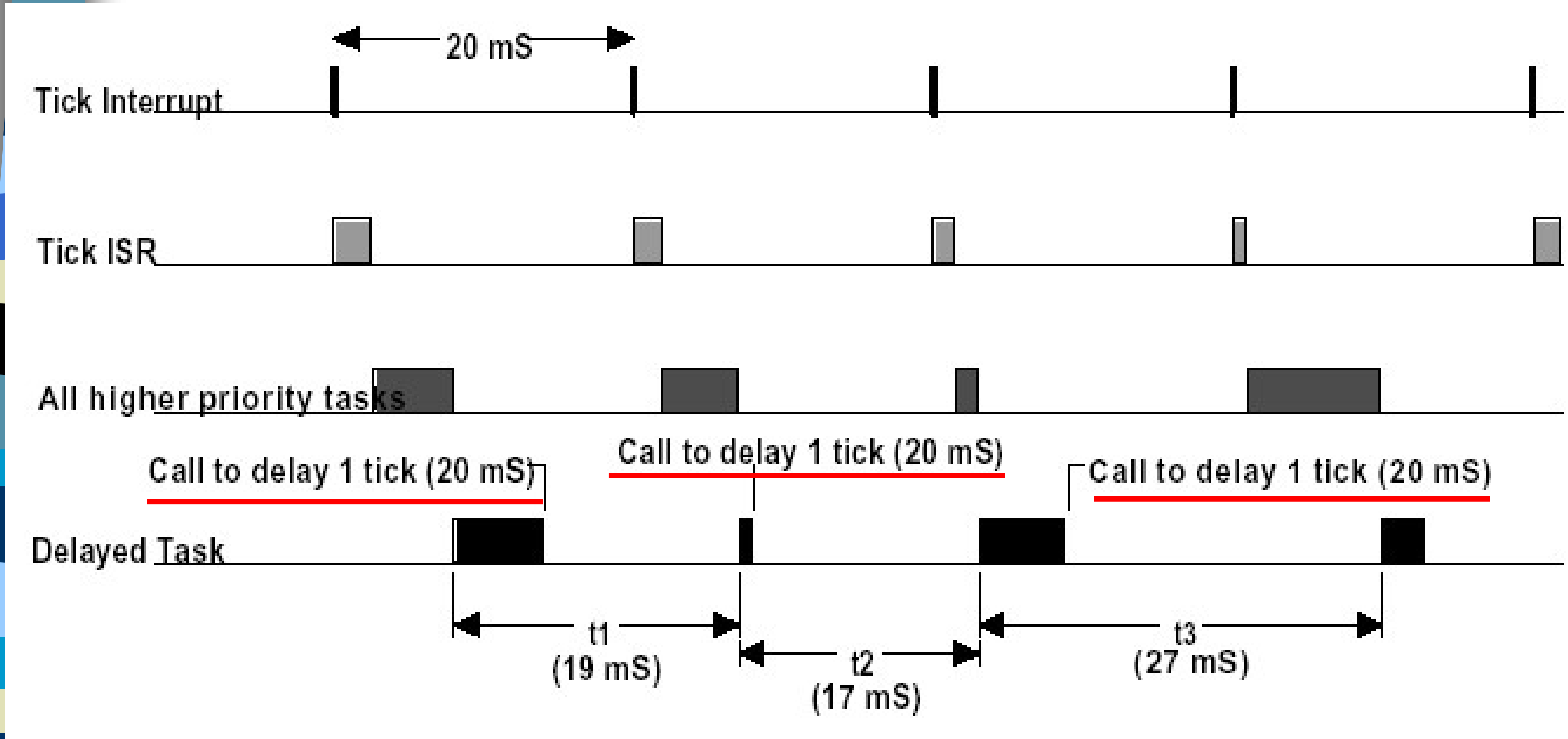
ISR Processing Time

- Interrupt occurs ➔ jobs need to be done
- You can
  - Execute the jobs in the ISR, or
  - Execute the jobs in a task
  - e.g., ISR1/ISR2 in OSEK & AUTOSAR
- It depends!!
  - **Rule:** ISR should not consume too much time

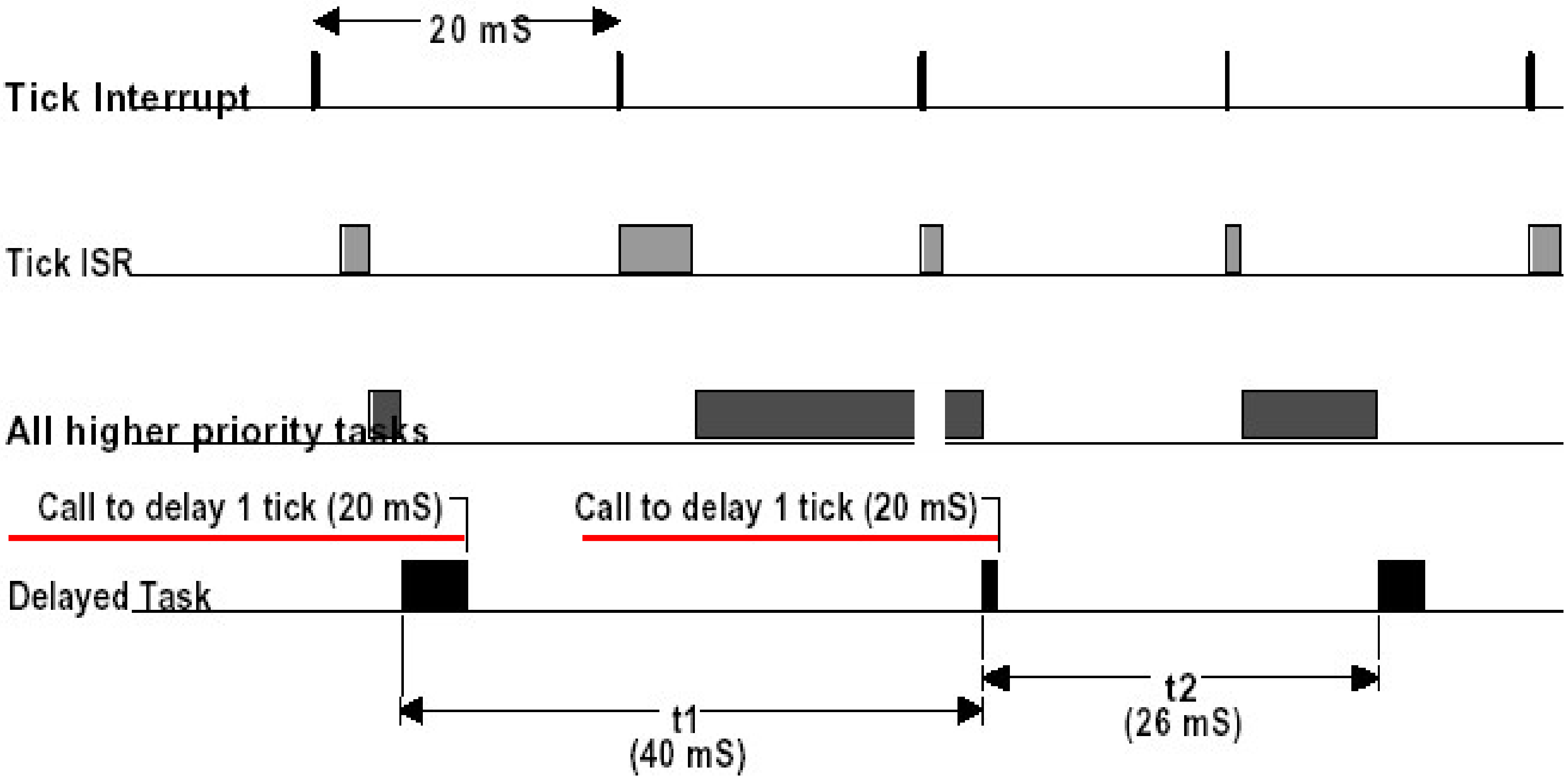
Clock Tick

- Timer interrupt
  - A special interrupt that occurs periodically
  - Can be used to measure time
  - Can also be viewed as the system's heartbeat
- Clock tick
  - Time between successive time interrupts
  - Typically, 1-200ms
  - Allows
    - Task delay, timeout for waiting….
      - The basic delay time unit is a tick….
  - Overhead vs. tick resolution

Delay Jitter – Case 1



Delay Jitter – Case 2



Delay Jitter

- Possible Solutions
  - Rearrange task priorities
  - Write critical code in ISR
- Delay jitter increases as the priority of the task decreases…

Memory Requirement

- *Memory size matters!*
- Memory Size Factors (when a multi-tasking kernel is used)
  - Application data/code size
    - Mainly depends on the application programmer
    - Use a high code-density ISA
    - Utilize compiler optimizations
  - Kernel data/code size
    - Data structures for managing the tasks…
    - Kernel code
      - Depends on the kernel functionality
      - Also depends on ISA and compiler

Memory Requirement

- Memory Size Factors
  - Task stack size
    - Contains local variables, arguments
    - Equal for all tasks?
    - Consider
      - Too many arguments?
      - Use large local arrays, structures….?
      - Too deep function/interrupt nesting?
      - Stack usage of library functions should also be considered

Use a Real Time Kernel or Not?

- Ease application development
  - Multi-task applications
  - Provide basic services
  - Provide/handle time critical event efficiently
- Extra Cost
  - More memory (ROM/RAM) space
  - CPU overhead (about 2% - 4%)
  - More money!!
    - Not all RTOS kernels are free!