Introduction To MicroC/OS-II

Chi-Yang Hung

Home Automation, Networking and Entertainment Lab.
Dept. of Computer Science and Information Engineering
hanel.csie.ncku.edu.tw

HANEL

OUTLINES

- Introduction
- Real-Time Systems Concepts
- Kernel Structure
- Porting MicroC/OS-II

Introduction

Introduction to RTOS

- What is a Real-Time System?
- What is a Embedded System?
- What is a Real-Time Operating System?

What is a Real-Time System?

- A system enforcing timing constraints, e.g. Avionics, Missile Control, …
  - The correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced.
  - Real-Time vs. High Performance
- What is a timing constraint?
  - A constraint of timing requirements, e.g. deadline, ready time, response time …

Real-Time System Examples

- Industrial and automation system
- Computer networking system
- Gaming and multimedia
- Medical instrument and devices
- Financial transaction applications
- Military defense system
- Security monitoring and response system
- Data acquisition system
- Machine vision/translation system
- …

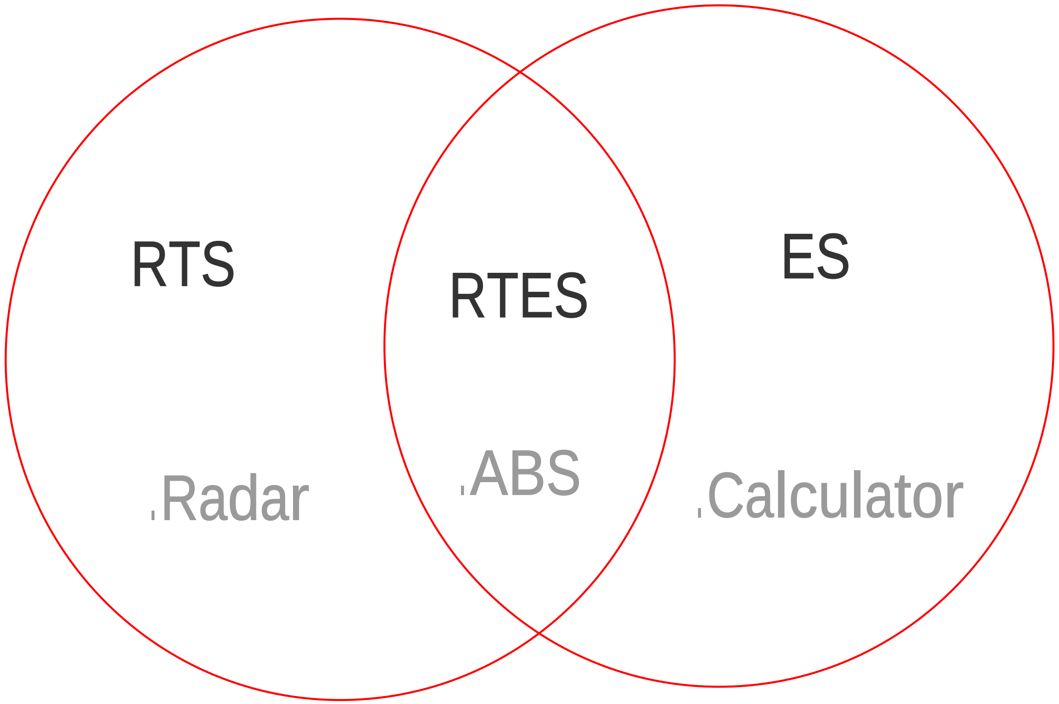
What is a Embedded System?

- A system designed to perform a specific function, e.g. eBook, PDA, eWatch, …
  - A combination of computer hardware and software, and perhaps additional mechanical or other parts.
  - Embedded vs. General-Purpose
- What is a specific function?
  - Is there a limitation?
- Robust, Low-power, Small, …

Embedded System Examples

- Computer peripherals
  - Keyboard, Mouse, …
- Information Appliances
  - Set-Up Boxes, WebTV, …
- Monitors and Sensors
  - Fire Alarm, Heartbeat Detector, …
- Controllers in Electronics
  - Refrigerator, Air Conditioner, …
- Communication Devices
  - Hub, Router, …
- …

Real-Time Systems vs. Embedded Systems



What is a Real-Time Operating System?

- What is a real-time operating system?
  - An operating system enforcing timing constraints, Lynx, pSOS, VxWorks, eCOS, uCLinux, LynxOS, RTLinux, KURT, uC/OS-II, …

Real-Time Systems Concepts

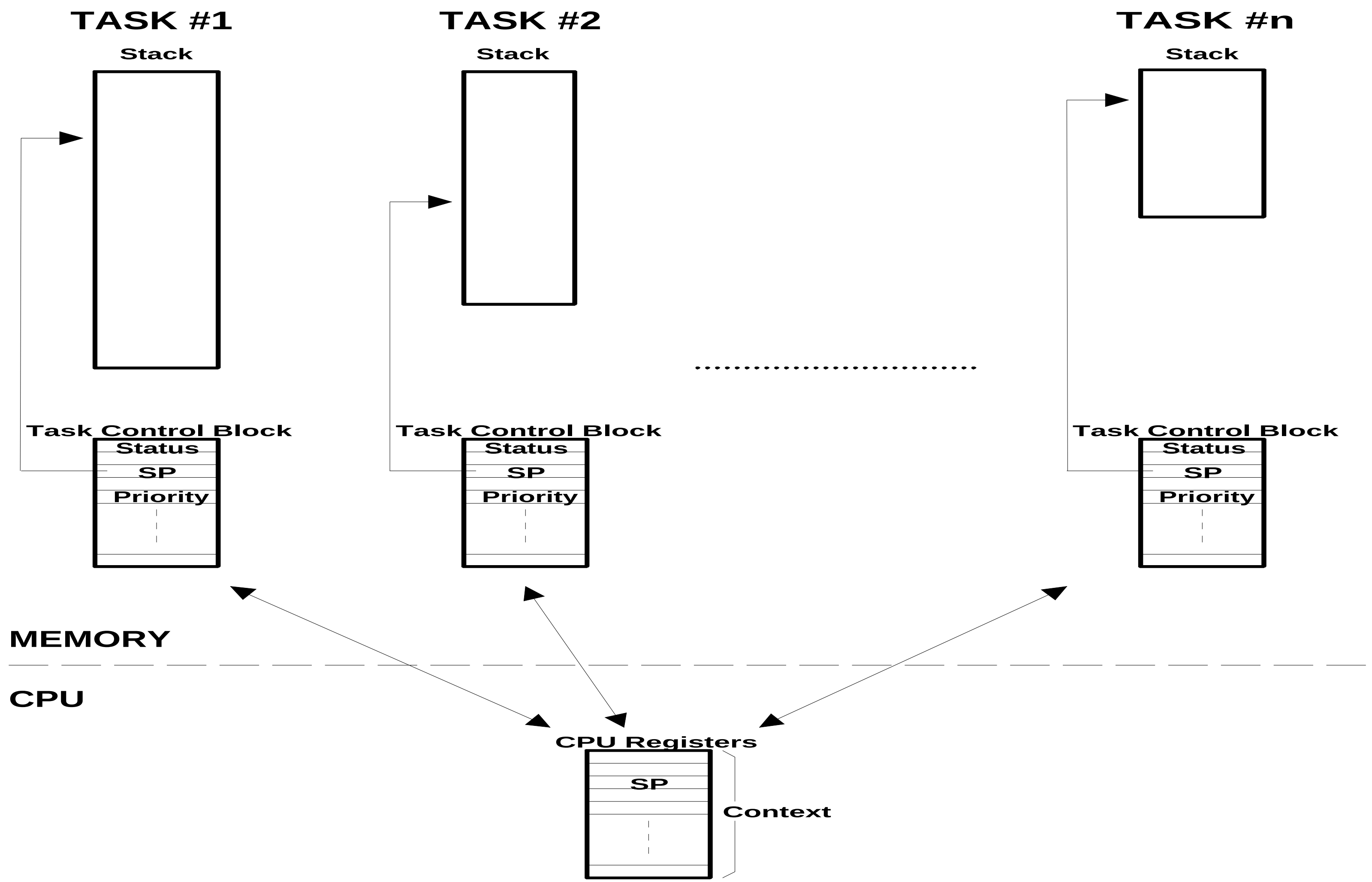
Real-Time Systems Concepts

- Multitasking
- Kernel
- Scheduling
- Mutual Exclusion
- Synchronization
- Interrupt

Multitasking

- Multitasking
  - A process of scheduling and switching CPU between several tasks.
- Related issues
  - Context Switch (Task Switch)
  - Resource Sharing
  - Critical Section

Multitasking(Cont.)



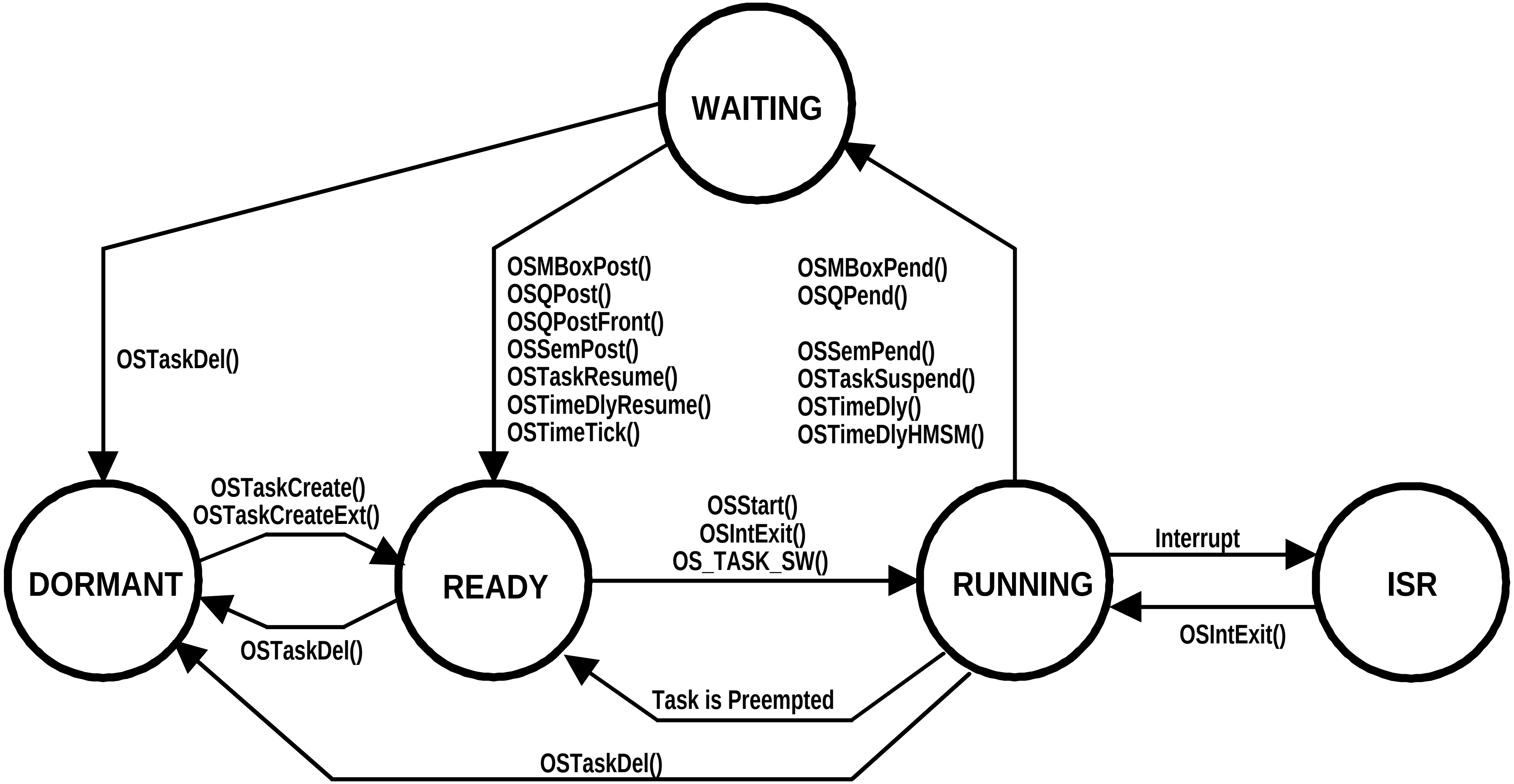
Tasks

- A task, or called a thread, a simple program that thinks it has the CPU all to itself.
- Each task is assigned a priority, its own set of CPU registers, and its own stack area .
- Each task typically is an infinite loop that can be in any one of five states.
  - Ready, Running, Waiting, ISR, Dormant

Tasks(Cont.)

- The DORMANT state corresponds to a task which resides in memory but has not been made available to the multitasking kernel.
- A task is READY when it can execute but its priority is less than the currently running task.
- A task is RUNNING when it has control of the CPU.
- A task is WAITING when it requires the occurrence of an event (waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur, time to expire etc.).

Task States


WAITING
OSTaskDel()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

Kernel

- Kernel is the part of a multitasking system responsible for the management of tasks.
- Context switching is the fundamental service of a kernel.

Non-Preemptive Kernel

v.s

Preemptive Kernel

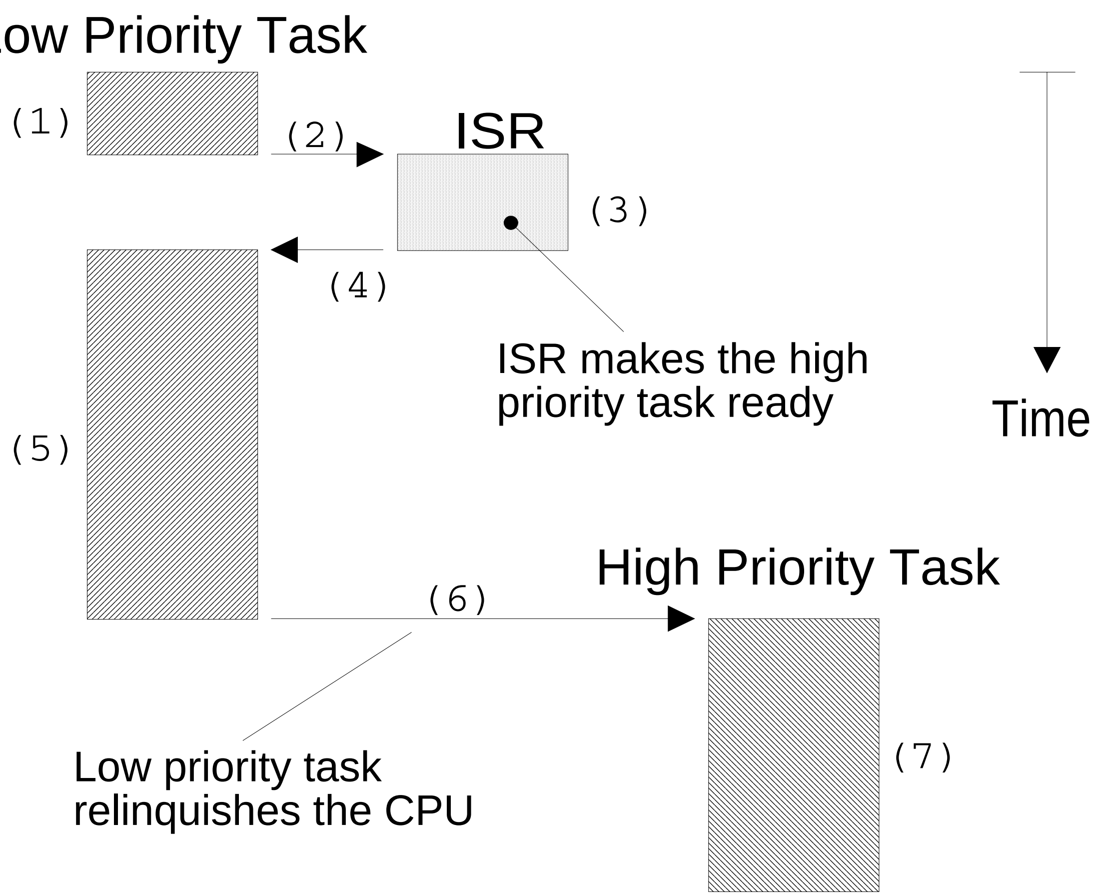
Non-Preemptive Kernel

- A non-preemptive kernel allows each task to run until it voluntarily gives up control of the CPU.
- An ISR can make a higher priority task ready to run, but the ISR always returns to the interrupted task.
- One of the advantages of a non-preemptive kernel is that interrupt latency is typically low.
- Linux 2.4 is non-preemptive.
- Linux 2.5 is to be preemptive.

Non-Preemptive Kernel(Cont.)

- At the task level, non-preemptive kernels can also use non-reentrant functions.
- Another advantage of non-preemptive kernels is the lesser need to guard shared data through the use of semaphores.
- The most important drawback of a non-preemptive kernel is responsiveness.
- Very few commercial kernels are non-preemptive.

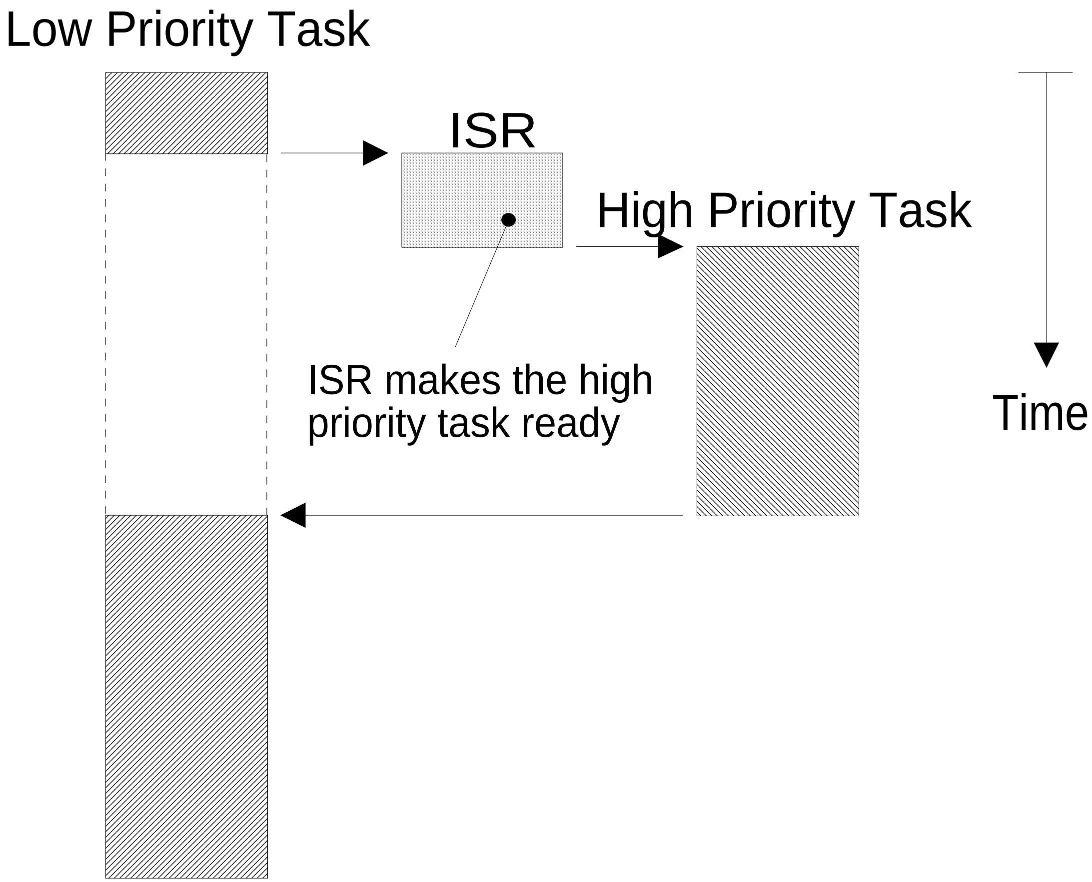
Non-Preemptive Kernel(Cont.)



Preemptive Kernel

- A preemptive kernel is used when system responsiveness is important.
- A preemptive kernel always executes the highest priority task that is ready to run.
- µC/OS-II and most commercial real-time kernels are preemptive.
- Much better response time.
- Should not use non-reentrant functions, unless the functions are mutual exclusive.

Preemptive Kernel(Cont.)



Function Reentrancy

- A reentrant function is a function that can be used by more than one task without fear of data corruption.
- Reentrant functions either use local variables or protected global variables.
  - OS_ENTER_CRITICAL()
  - OS_EXIT_CRITICAL()

Function Reentrancy(Cont.)

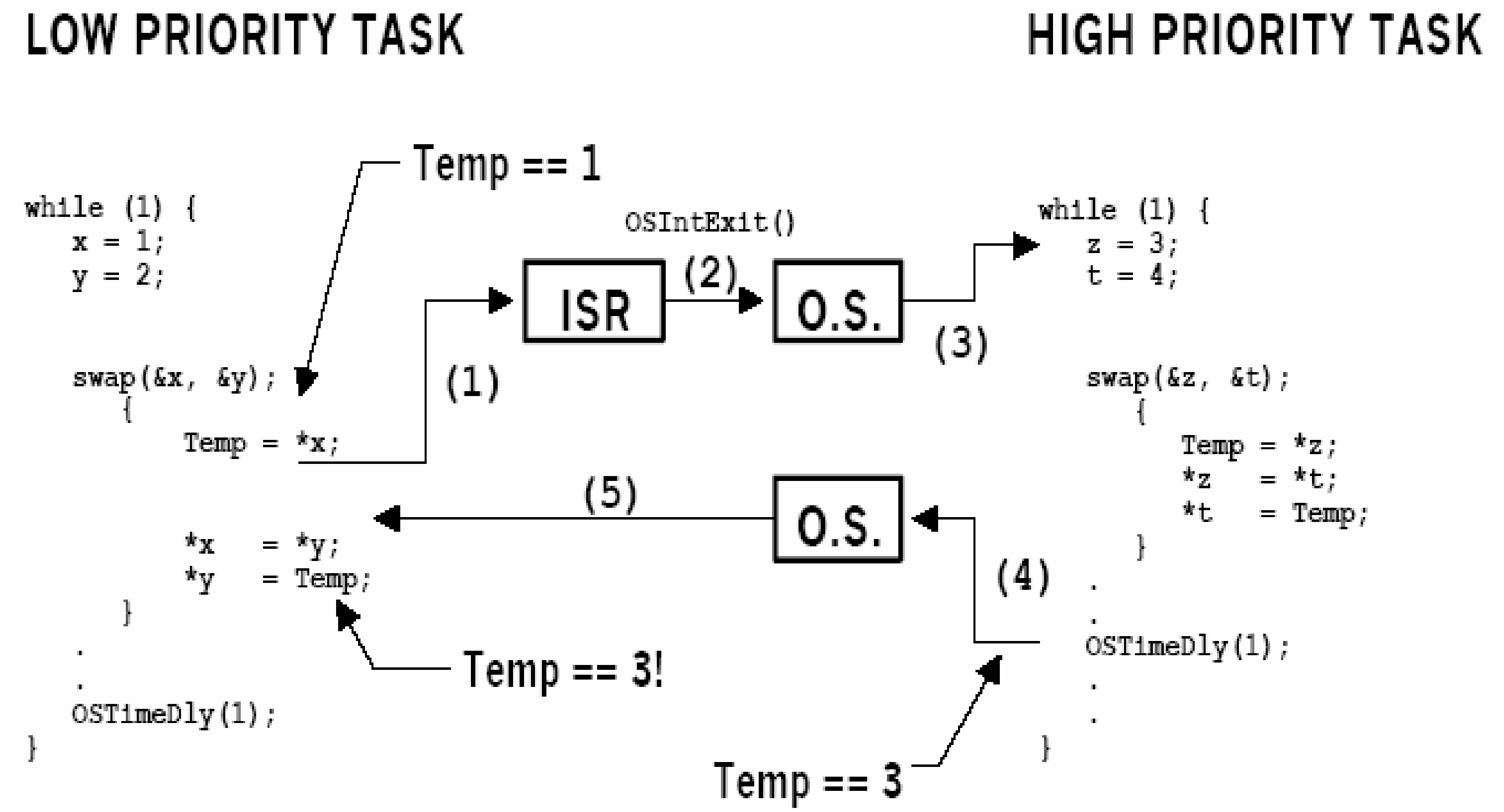
Non-Reentrant Function

```
static int Temp;

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

Reentrant Function

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

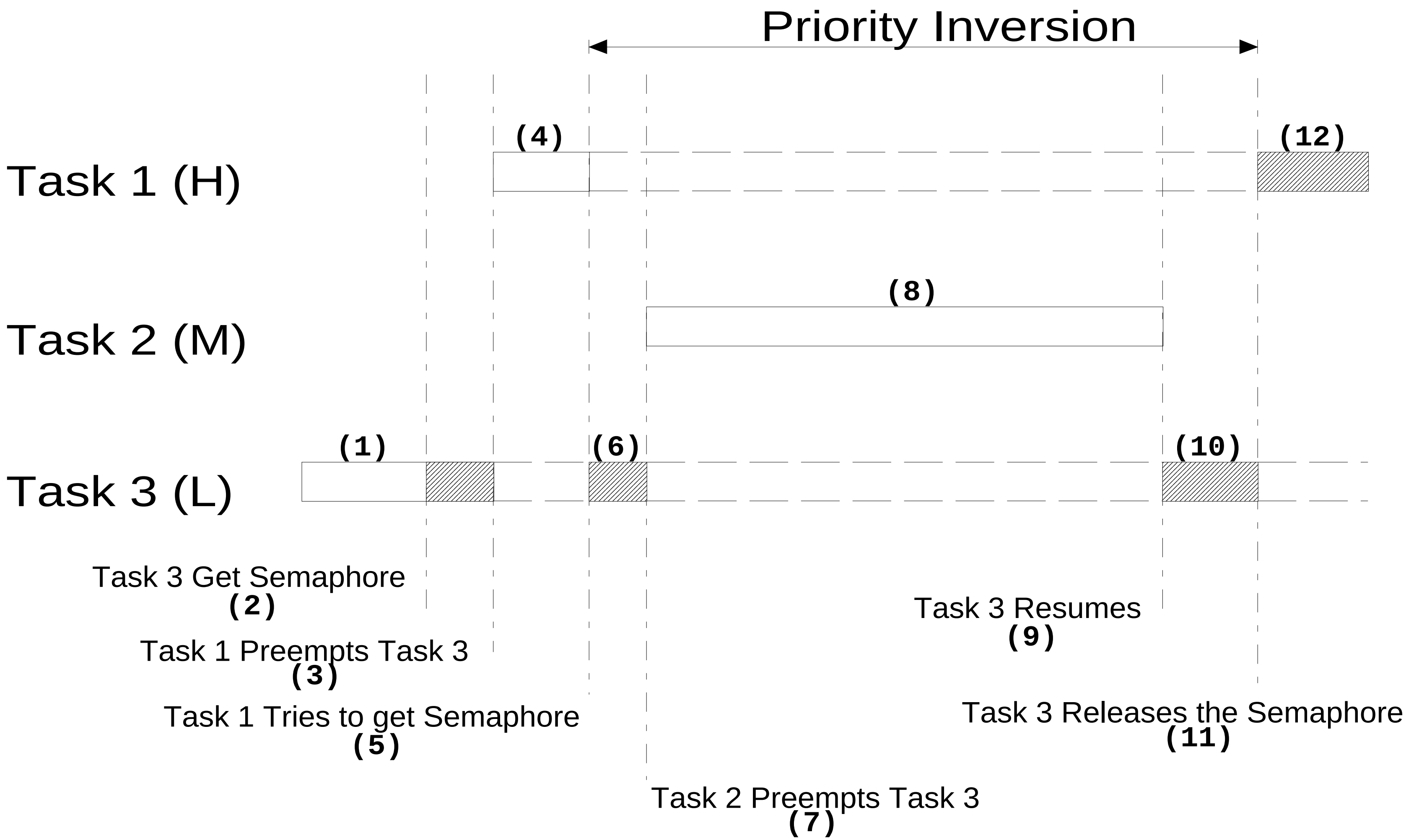
Function Reentrancy(Cont.)



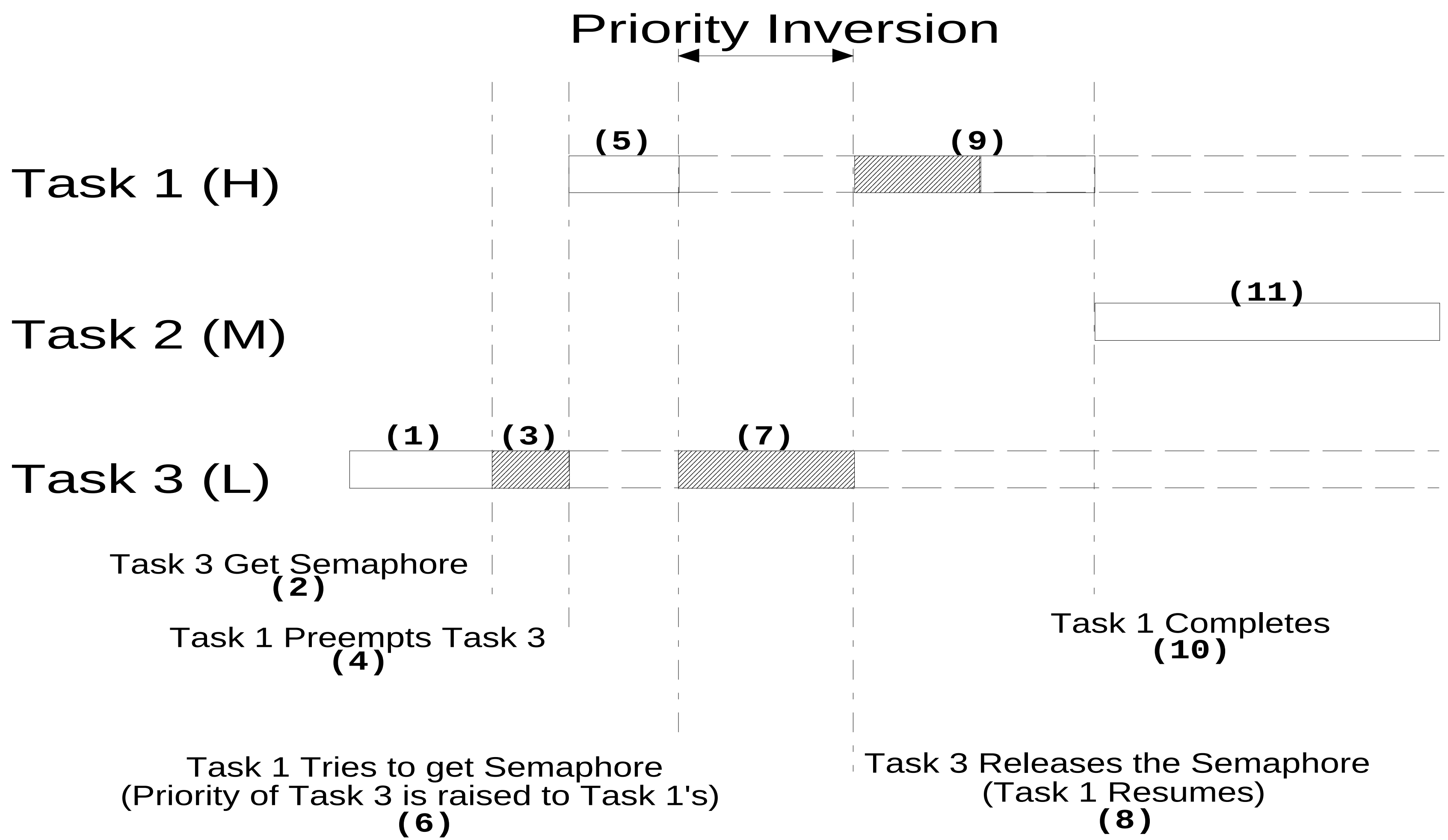
Scheduling

- Task Priority Assignment
  - Static priority
    - Rate Monotonic (RM)
  - Dynamic priority
    - Earliest-Deadline First (EDF)

Priority Inversion Problem



Priority Inheritance



Mutual Exclusion

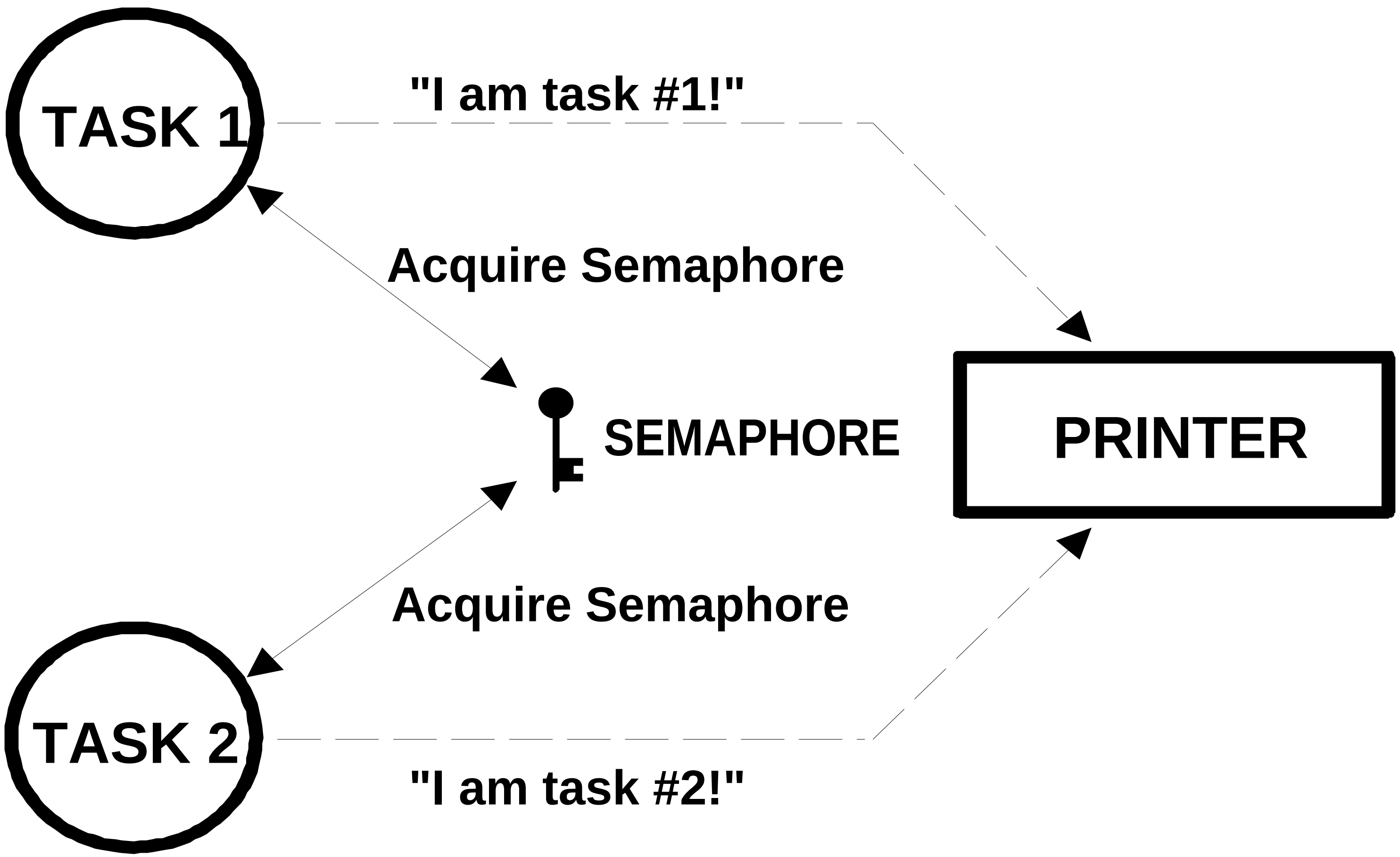
- Protected shared data of processes.
- Exclusive access implementation
  - Disabling and enabling interrupts
  - Test-and-Set
  - Disabling and enabling scheduler
  - Semaphores
    - Binary Semaphore
    - Counting Semaphore
- Deadlock – set timeout for a semaphore

Semaphore

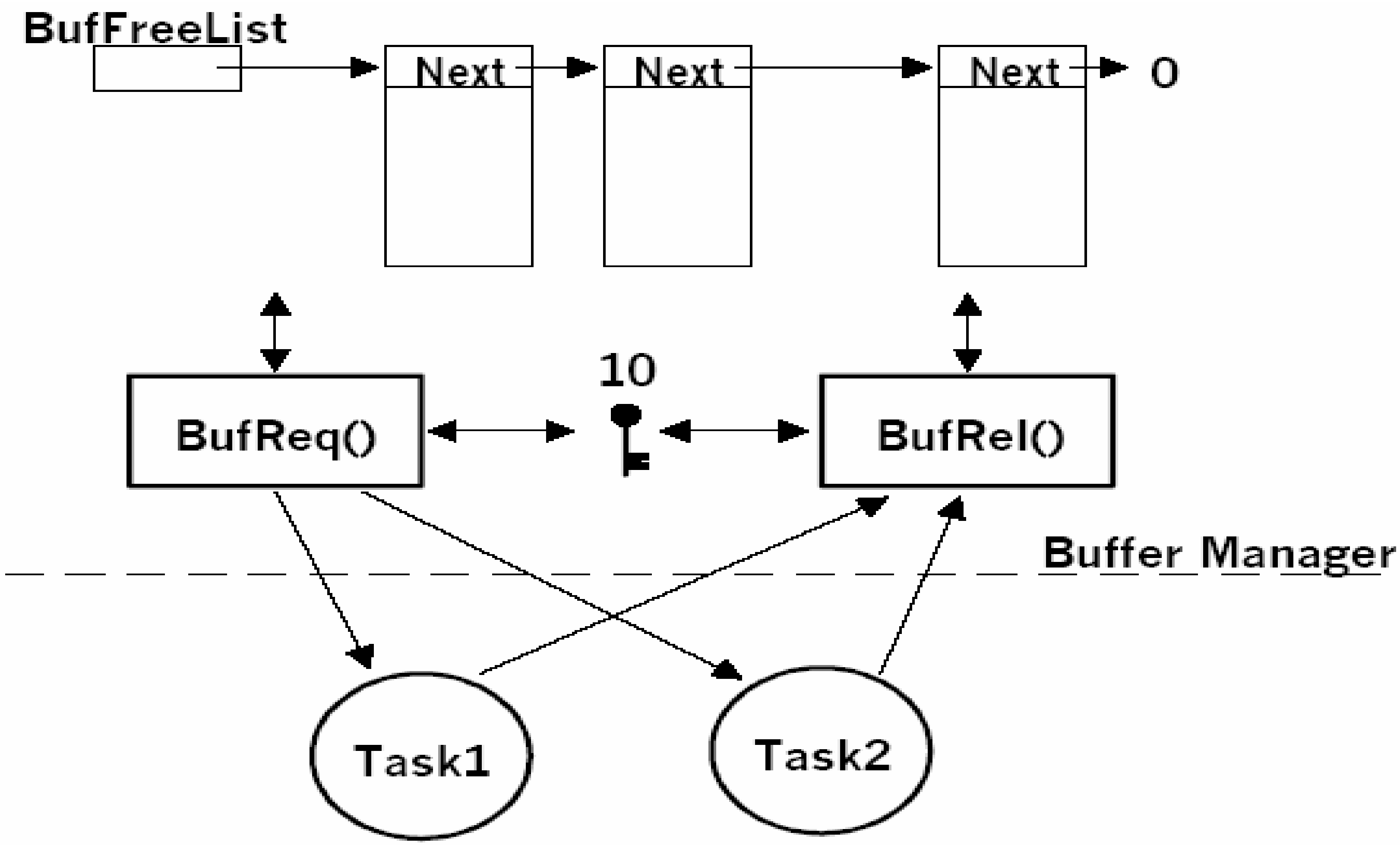
- There are generally only three operations that can be performed on a semaphore: INITIALIZE (also called *CREATE*), WAIT (also called *PEND*), and SIGNAL (also called *POST*).

```
OS_EVENT *SharedDataSem;


void Function (void)
{
    INT8U err;


    OSSemPend(SharedDataSem, 0, &err);
    .
    .      /* You can access shared data in here (interrupts are recognized) */
    .
    OSSemPost(SharedDataSem);
}
```

Using Binary Semaphore



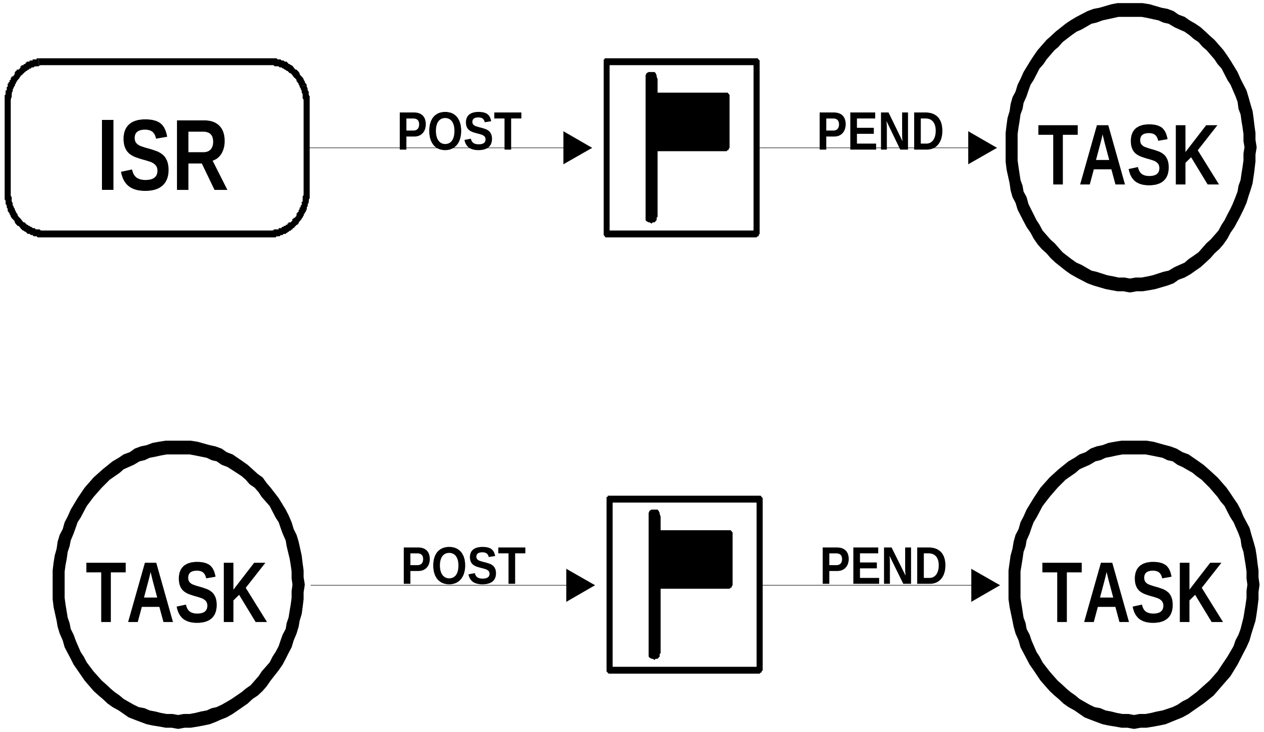
Using Counting Semaphore



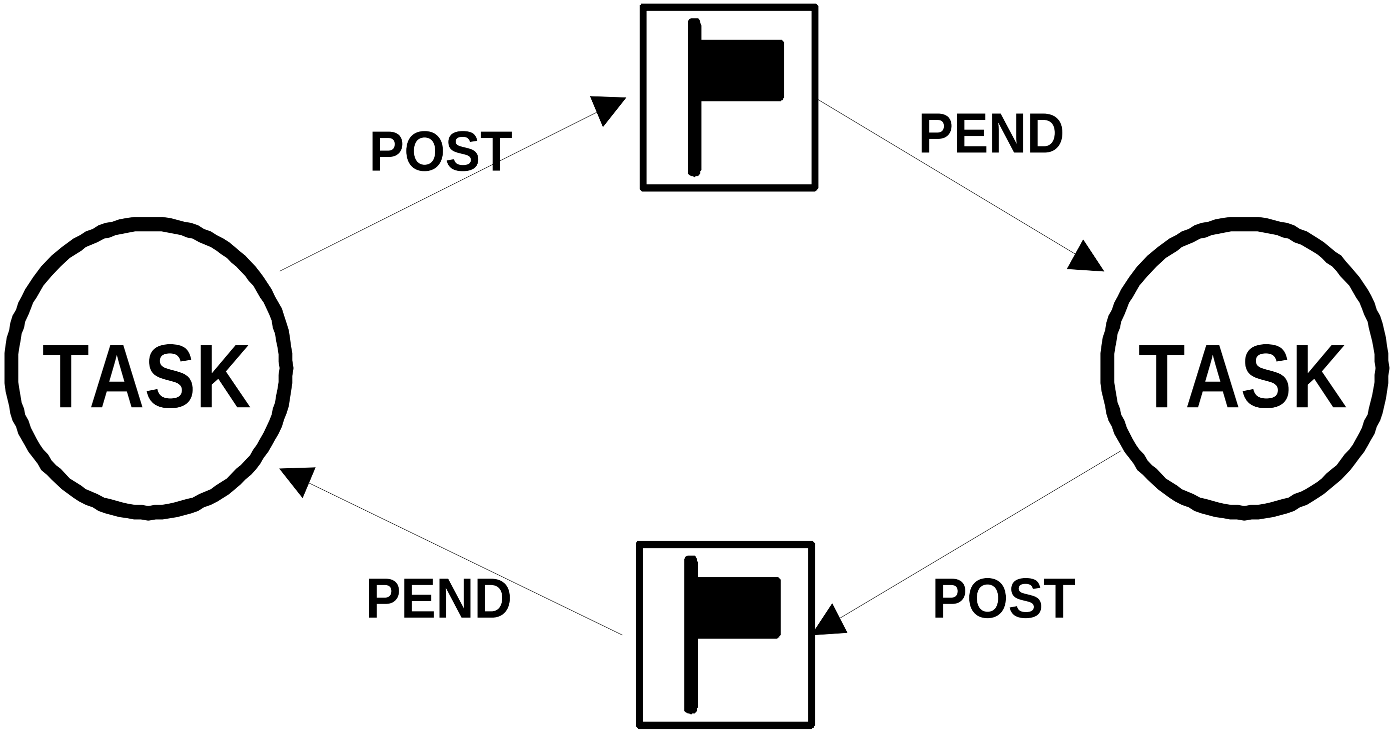
Synchronization

- Synchronization mechanism is used between tasks or task to ISR.
- Unilateral rendezvous
  - A task can be synchronized with an ISR, or another task when no data is being exchanged, by using a semaphore.
- Bilateral rendezvous
  - Two tasks can synchronize their activities by using two semaphores.

Unilateral rendezvous



Bilateral rendezvous

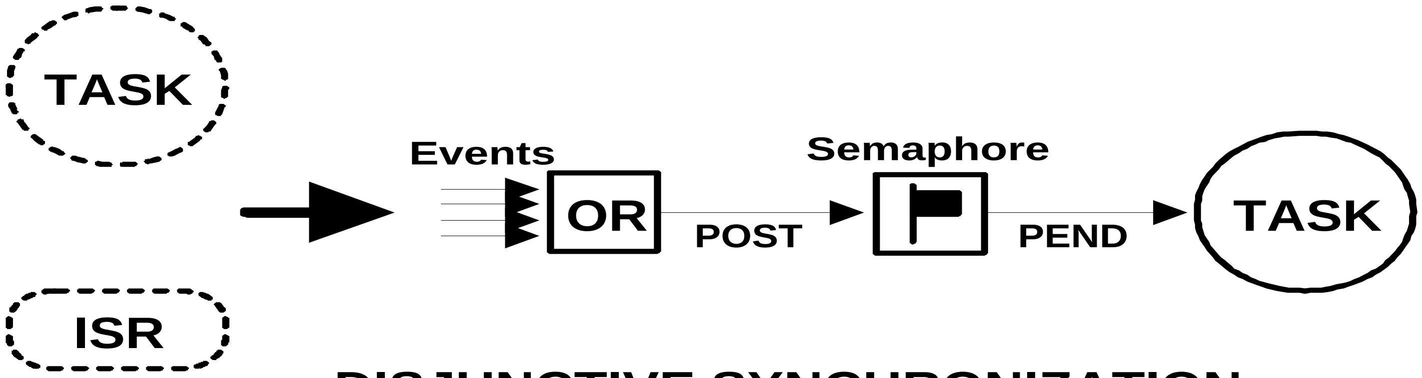


Event Flags

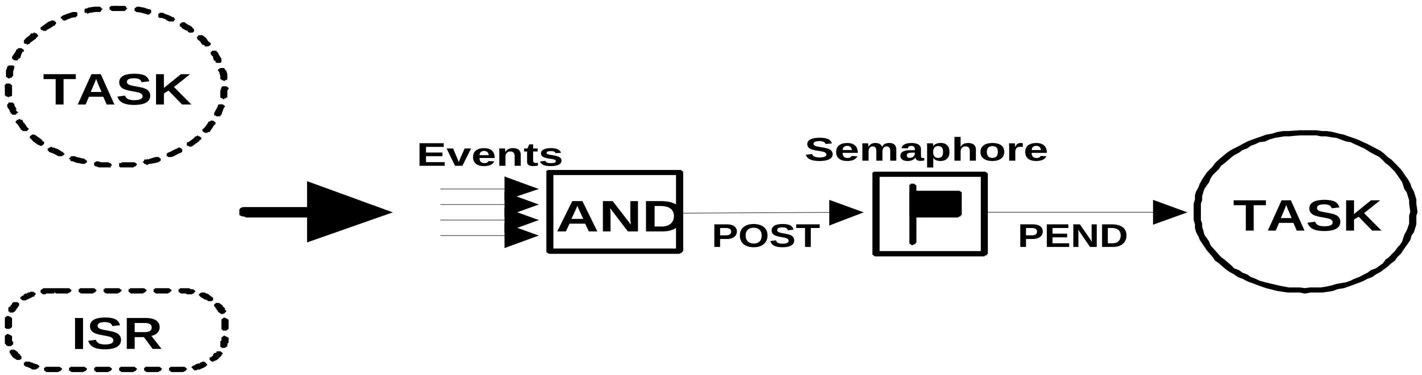
- The task can be synchronized when any of the events have occurred. This is called *disjunctive synchronization* (logical OR).

- A task can also be synchronized when all events have occurred. This is called *conjunctive synchronization* (logical AND).

Event Flags(Cont.)

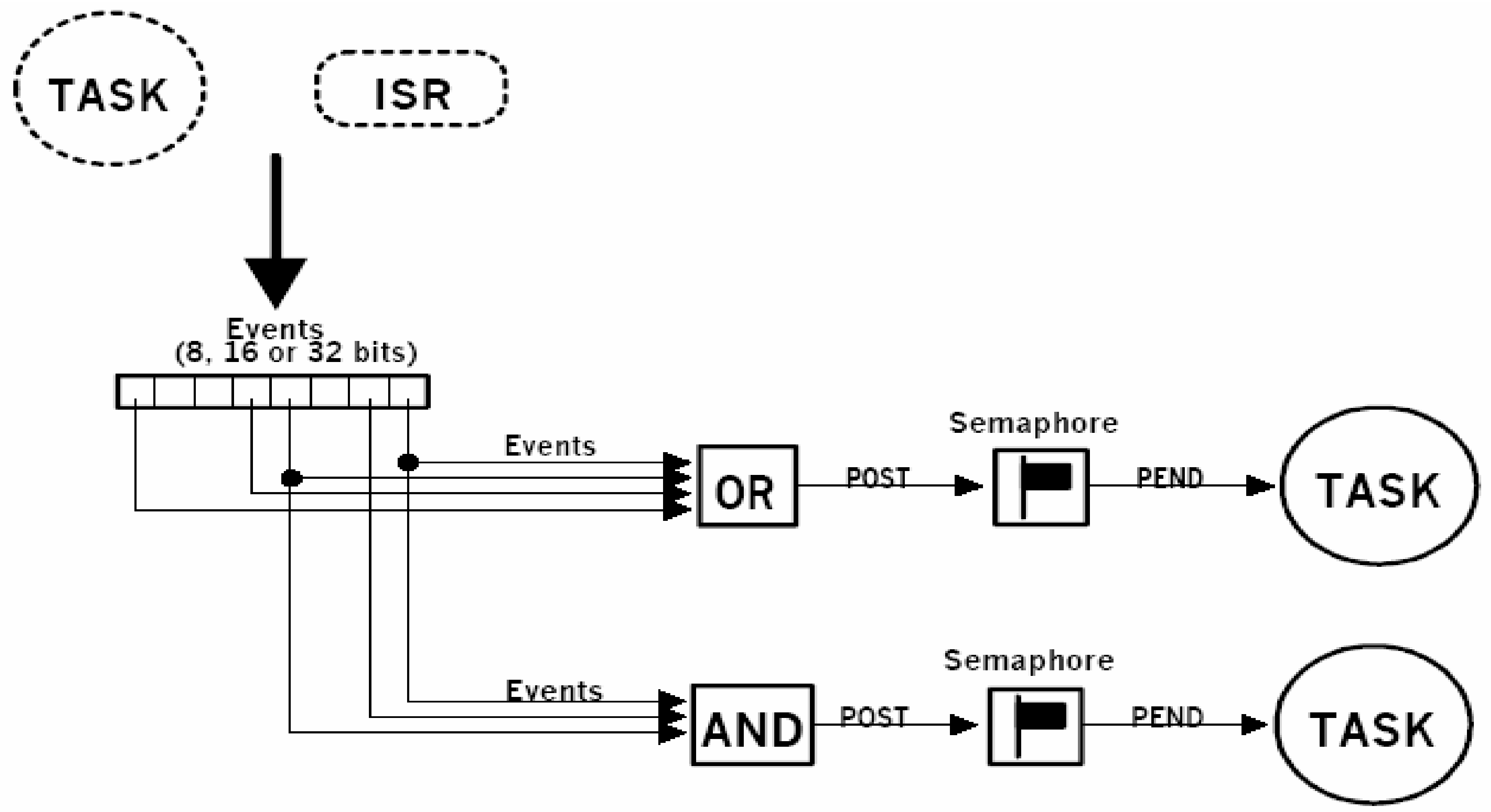


DISJUNCTIVE SYNCHRONIZATION



CONJUNCTIVE SYNCHRONIZATION

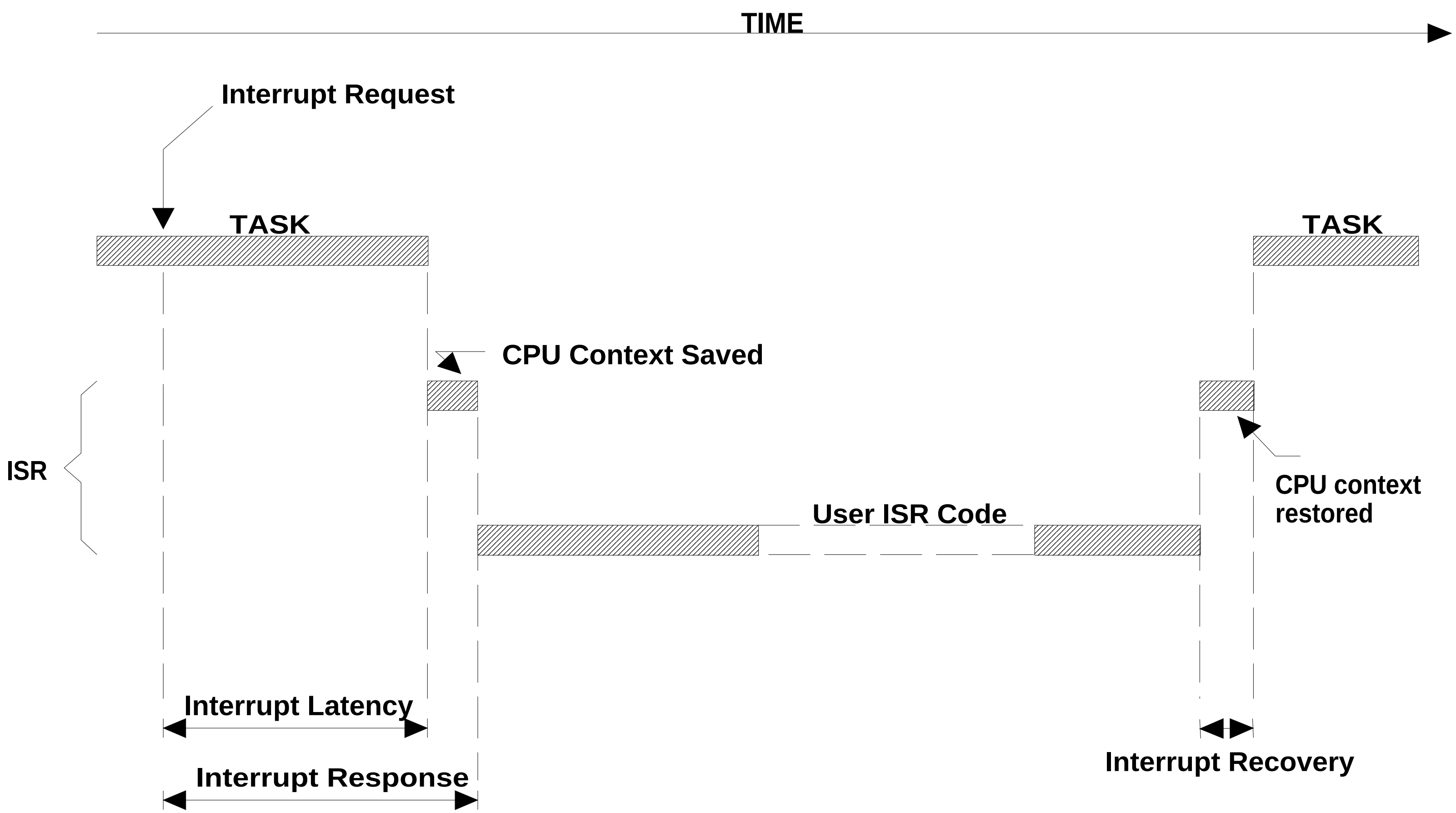
Event Flags(Cont.)



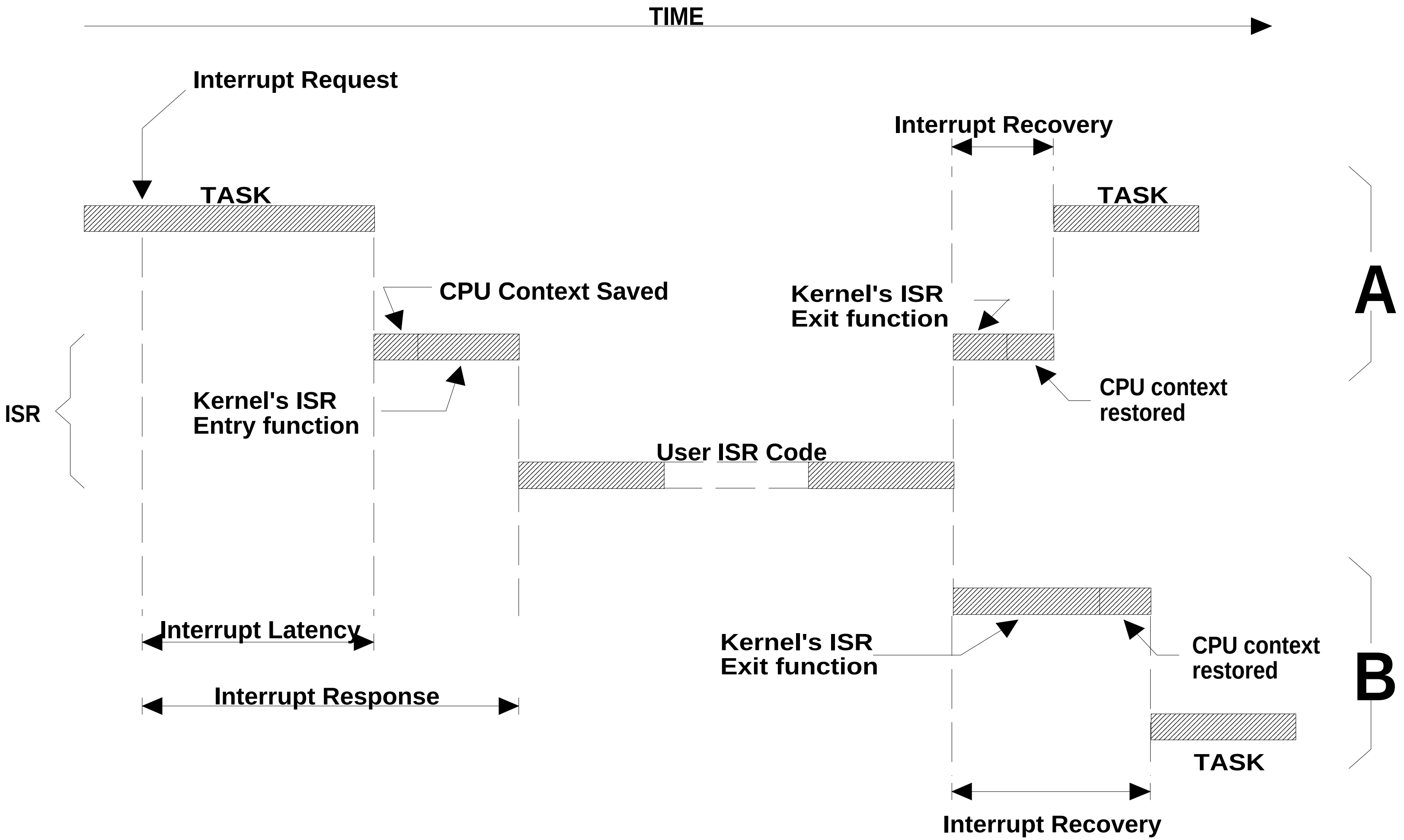
Interrupt

- An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred.
- Interrupt Latency
- Interrupt Response
- Interrupt Recovery
- Interrupt Nesting
- Non-Maskable Interrupts (NMIs)

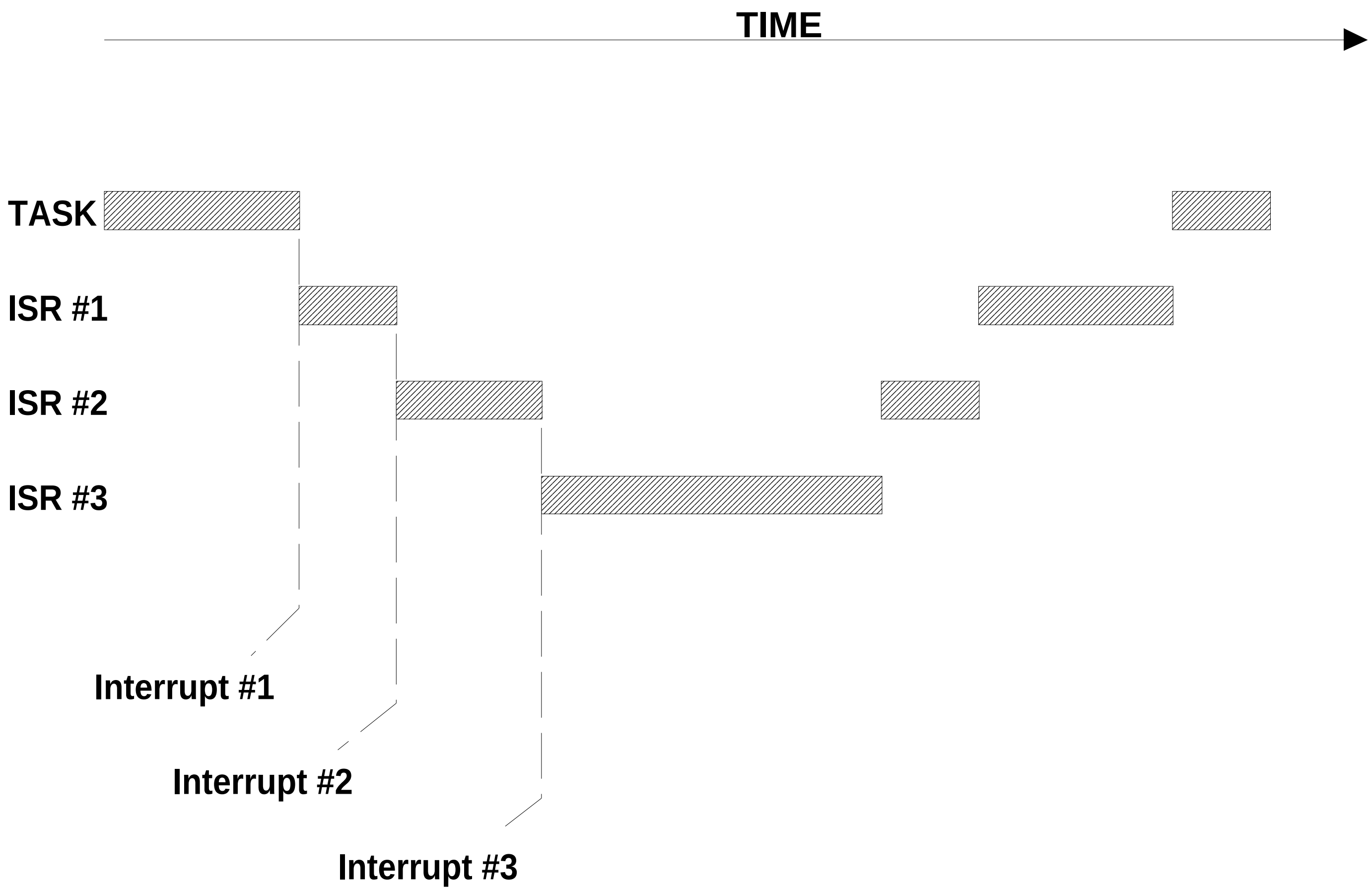
Non-preemptive kernel



Preemptive kernel



Interrupt Nesting



Non-Maskable Interrupts (NMIs)

- Service the most important time-critical ISR
  - Can not be disabled
- Interrupt latency = Time to execution the longest instruction + Time to start execution the NMI ISR
- Interrupt response = Interrupt latency + Time to save CPU context
- Interrupt recovery = Time to restore CPU context + time of executing return-from-interrupt

Clock Tick

- A periodical interrupt
  - Be viewed as heartbeat
  - Application specific and is generally between 10ms and 200ms
  - Faster timer causes higher overhead
  - Delay problem should be considered in real-time kernel.

Real-Time Kernels

- Real-time OS allows real-time application to be designed and expanded easily.
- The use of an RTOS simplifies the design process by splitting the application into separate tasks.
- Real-time kernel requires more ROM/RAM and 2 to 4 percent overhead.
  - Kernel request extra code(ROM)
  - Application code(RAM)
  - All kernel require extra RAM to maintain internal variables, data structure, queue, etc

Kernel Structure

Kernel Structure

- Critical Sections
- Tasks
- Task Control Blocks
- Ready List
- Idle Task
- Statistic Task
- µC/OS-II Initialization

Critical Sections

- OS_CRITICAL_METHOD == 1
  - The first and simplest way to implement thus two macro is to invoke the processor instruction to disable interrupts for OS_ENTER_CRITICAL() and to enable interrupts instruction for OS_EXIT_CRITICAL().
- OS_CRITICAL_METHOD == 2
  - The second way to implement OS_ENTER_CRITICAL() is to save the interrupt disable status onto the stack and then disable interrupt. OS_EXIT_CRITICAL() is implement by restoring the interrupt status from stack.

Critical Sections(Cont.)

- OS_CRITICAL_METHOD == 3
  - Some compiler assume you with extensions that you to obtain the current value of Processor Status Word (PSW) and save it into a local variable within a C function.

Tasks

- Up to 64 tasks.
- Two tasks for system use (idle and statistic).
- Priorities 0, 1, 2, 3, OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1, OS_LOWEST_PRIO for future use.
- Each task must be assigned a unique priority level from 0 to OS_LOWEST_PRIO-2.
- The task priority is also the task identifier.

Multitasking

- Multitasking is started by calling OSStart().
- OSStart() runs the highest priority task that is READY to run.

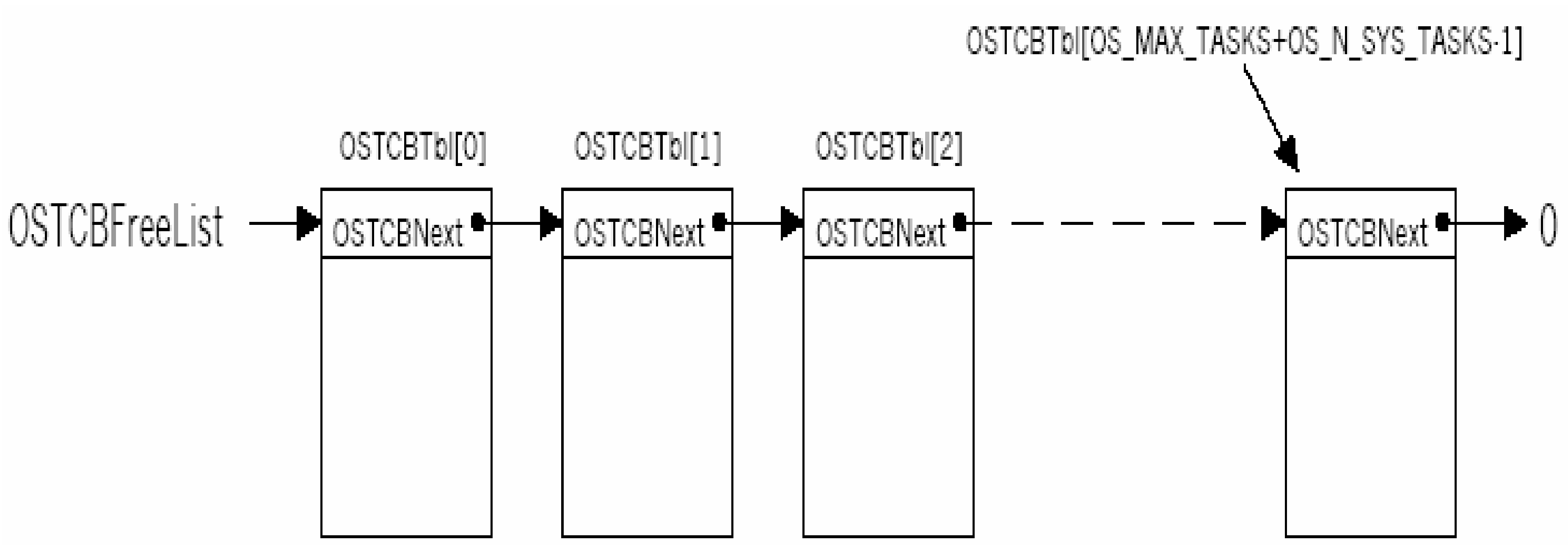
Task Control Blocks (Cont.)

- Used to accelerate the process of making a task ready to run, or to make a task wait for an event (to avoid computing these values at runtime).
- The values for these fields are computed when the task is created or when the task's priority is changed.

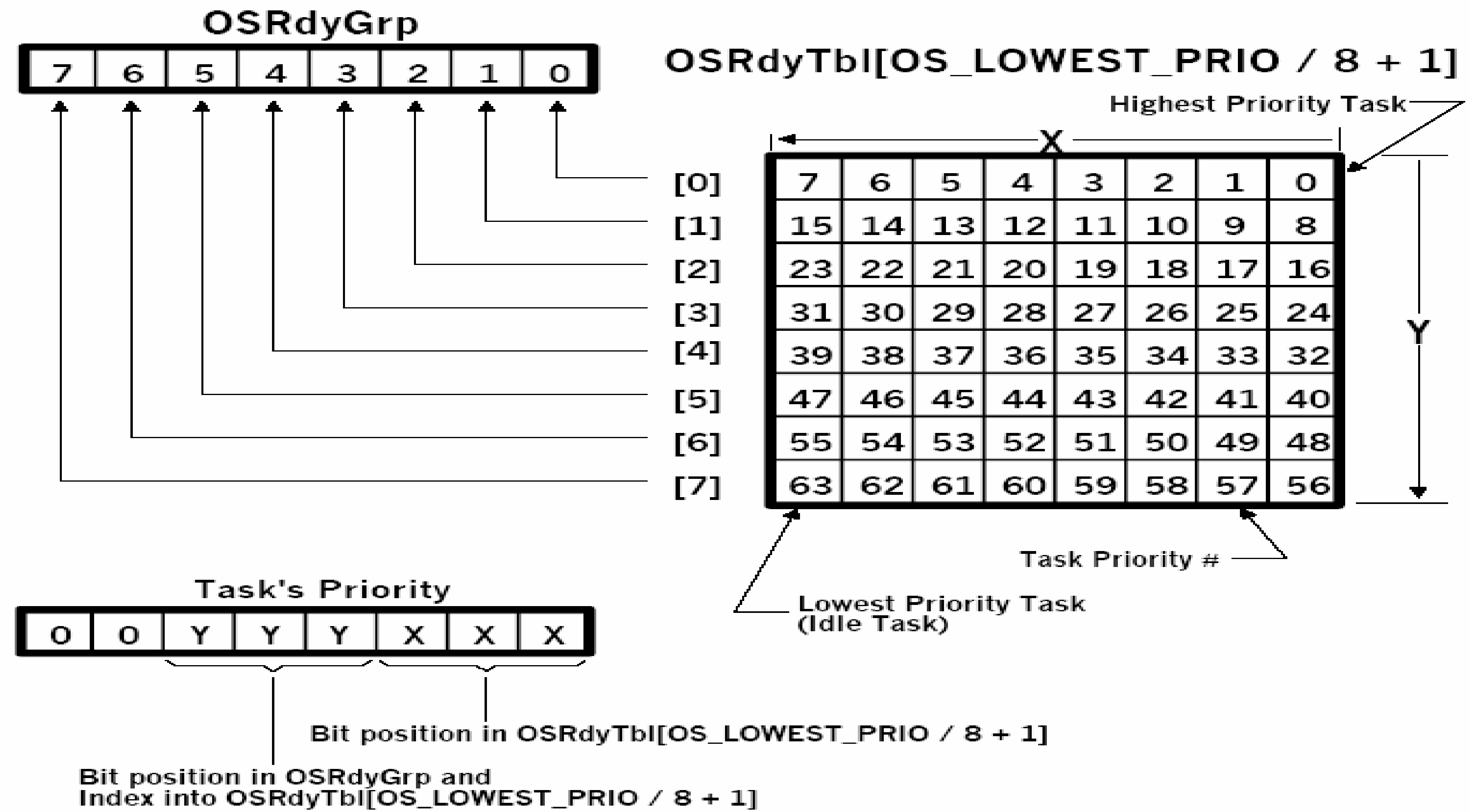
```
OSTCBY      = priority >> 3;
OSTCBBitY   = OSMapTbl[priority >> 3];
OSTCBX      = priority & 0x07;
OSTCBBitX   = OSMapTbl[priority & 0x07];
```

OS_TCB Lists

- TCBs store in OSTCBTbl[].
- All TCBs are initialized and linked when uC/OS-II is initialized.



Ready List



OSRdyGrp
7 6 5 4 3 2 1 0
OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Highest Priority Task
X
Y
[0] [1] [2] [3] [4] [5] [6] [7]
7 6 5 4 3 2 1 0
15 14 13 12 11 10 9 8
23 22 21 20 19 18 17 16
31 30 29 28 27 26 25 24
39 38 37 36 35 34 33 32
47 46 45 44 43 42 41 40
55 54 53 52 51 50 49 48
63 62 61 60 59 58 57 56
Task Priority #
Lowest Priority Task
(Idle Task)
Task's Priority
0 0 Y Y Y X X X
Bit position in OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]
Bit position in OSRdyGrp and
Index into OSRdyTbl[OS_LOWEST_PRIO / 8 + 1]

Ready List(Cont.)

- Each task that is ready to run is placed in a ready list consisting of two variables, OSRdyGrp and OSRdyTbl[].
- Task priorities are grouped (8 tasks per group) in OSRdyGrp.
- Each bit in OSRdyGrp is used to indicate whenever any task in a group is ready to run.
- When a task is ready to run it also sets its corresponding bit in the ready table, OSRdyTbl[].

OSRdyGrp and OSRdyTbl[]

- To determine which priority (and thus which task) will run next, the scheduler determines the lowest priority number that has its bit set in OSRdyTbl[].

Bit 0 in OSRdyGrp is 1 when any bit in OSRdyTbl[0] is 1.
Bit 1 in OSRdyGrp is 1 when any bit in OSRdyTbl[1] is 1.
Bit 2 in OSRdyGrp is 1 when any bit in OSRdyTbl[2] is 1.
Bit 3 in OSRdyGrp is 1 when any bit in OSRdyTbl[3] is 1.
Bit 4 in OSRdyGrp is 1 when any bit in OSRdyTbl[4] is 1.
Bit 5 in OSRdyGrp is 1 when any bit in OSRdyTbl[5] is 1.
Bit 6 in OSRdyGrp is 1 when any bit in OSRdyTbl[6] is 1.
Bit 7 in OSRdyGrp is 1 when any bit in OSRdyTbl[7] is 1.

Making a task ready to run

```
OSRdyGrp            |= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

Index	Bit Mask (Binary)
0	0000000**1**
1	000000**1**0
2	00000**1**00
3	0000**1**000
4	000**1**0000
5	00**1**00000
6	0**1**000000
7	**1**0000000

Task's Priority							
0	0	Y	Y	Y	X	X	X

Removing a task from the ready list

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

Finding the highest priority task

```
y    = OSUnMapTbl[OSRdyGrp];
x    = OSUnMapTbl[OSRdyTbl[y]];
prio = (y << 3) + x;
```

```
INT8U  const  OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,  4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};
```

76543210	OSUnMapTbl[]
00000000	0
00000001	0
00000010	1
00000011	0
00000100	2
00000101	0
00000110	1
00000111	0
00001000	3
00001001	0
00001010	1
00001011	0
00001100	2
00001101	0
00001110	1
00001111	0
…	

Idle Task

- µC/OS-II always creates a task (a.k.a. the *Idle Task*) which is executed when none of the other tasks is ready to run.
- The idle task (OSTaskIdle()) is always set to the lowest priority, i.e. OS_LOWEST_PRIO. OSTaskIdle() does nothing but increment a 32-bit counter called OSIdleCtr.
- Interrupts are disabled then enabled around the increment because on 8-bit and most 16-bit processors, a 32-bit increment requires multiple instructions which must be protected from being accessed by higher priority tasks or an ISR.

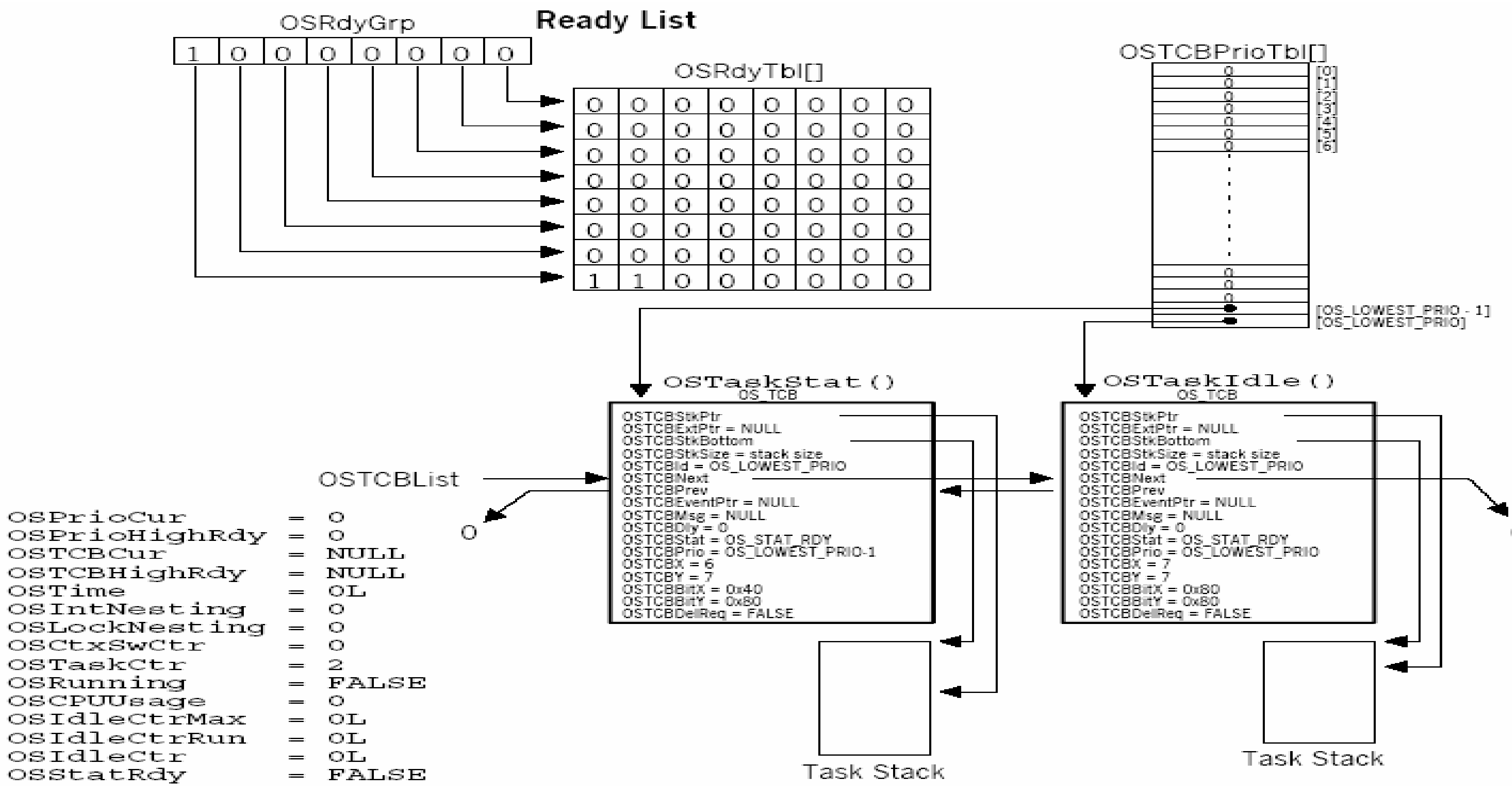
Statistics Task

- µC/OS-II contains a task that provides run-time statistics.
- OSTaskStat() executes every second and computes the percentage of CPU usage.
- This value is placed in the variable OSCPUUsage which is a signed 8-bit integer.
- You MUST call OSStatInit() from the first and only task created in your application during initialization.

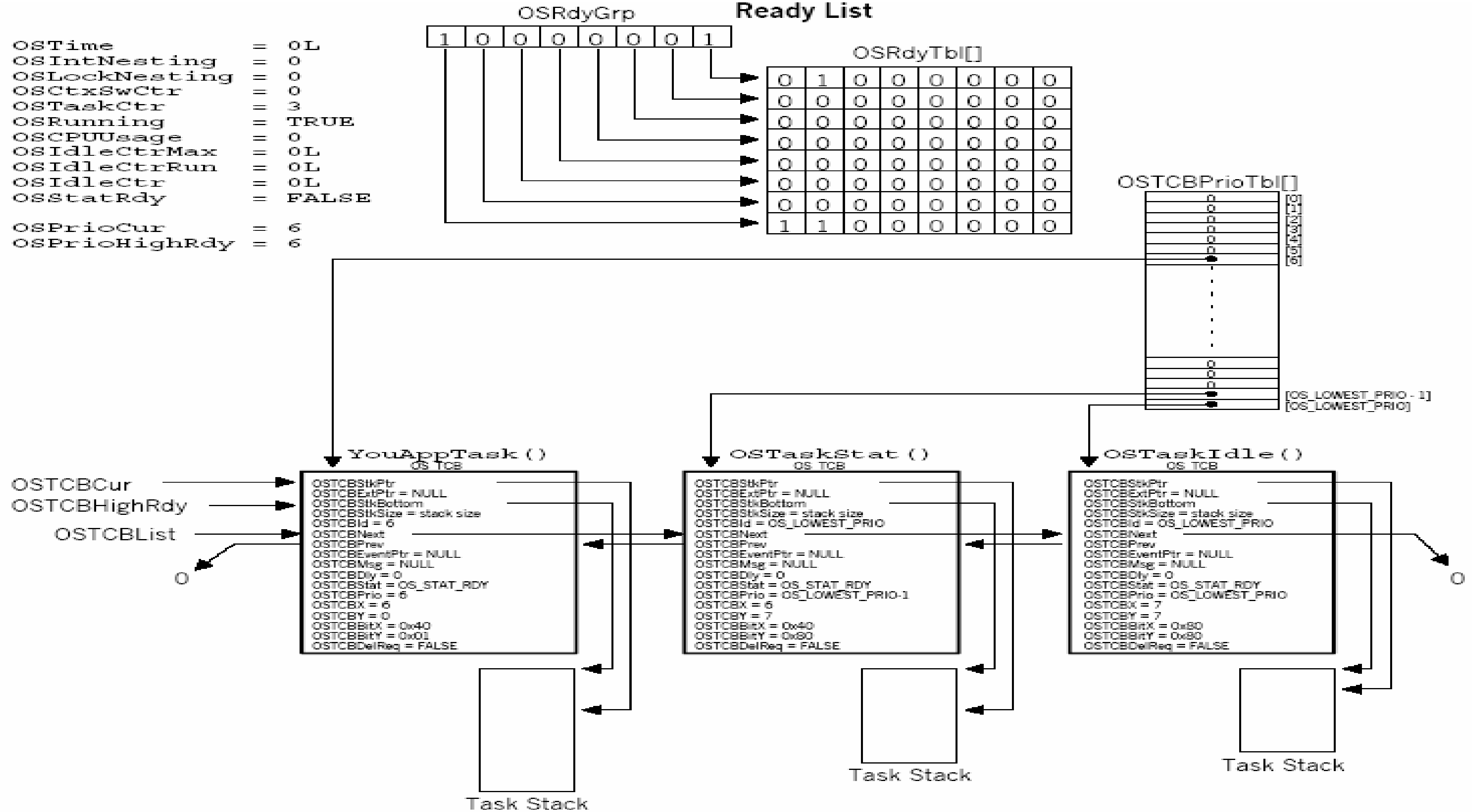
uC/OS-II Initialization

- A requirement of µC/OS-II is that you call OSInit() before you call any of its other services.
- OSInit() initializes all of µC/OS-II's variables and data structures.
- Create the idle task OSTaskIdle()
- Create the statistic task OSTaskStat()
- The Task Control Blocks (OS_TCBs) of these two tasks are chained together in a doubly-linked list.

Data structures after calling OSInit()



Variables and Data Structures after calling OSStart()



Porting MicroC/OS-II

Porting MicroC/OS-II

- Porting MicroC/OS-II to the x86 protected mode
- Loading the MicroC/OS-II
- Initializing the hardware
- Converting to a 32-bits, flat memory model
- Building the application

Porting MicroC/OS-II to the x86 protected mode

- Porting MicroC/OS-II to a protected mode, 32-bit, flat memory model of the 80386, 80486, Pentium and Pentium II CPUs.
- MicroC/OS-II is a portable, ROMable, preemptive, real-time, multitasking kernel for microprocessors.
- Originally written in C and x86 (real mode) assembly languages, it has been ported to a variety of microprocessors.

Loading the MicroC/OS-II

- The MicroC/OS-II can be loaded in various ways:
  a) It can be loaded from DOS. But switching into protected mode and fully bypassing DOS and the BIOS may create some inconsistencies inside DOS, preventing a normal return to it. Thus, the PC would have to be rebooted in order to get back to DOS. Also, the application would have to be built as a DOS application, requiring a 16-bit compiler.

Loading the MicroC/OS-II(Cont.)

b) It can be loaded from a floppy disk upon boot-up. By providing a bootstrap loader, the application would be the first thing loaded and would be in full control. The PC also has to be rebooted back to DOS (or Windows) if required.

c) It can be burned into PROM, into a stand-alone system.

Bootstrap Loader

- The bootstrap loader (BootSctr.img) can easily be installed on the very first sector of a floppy disk (e.g. the boot sector) by using the DEBUG program, distributed with DOS and Windows.
- The DEBUG's write command has the following format:
  - -w *offset disk track sector_count*
- By executing the following commands:
  - C:\MyTask>debug bootstrap.img
  - -w 0100 0 0 1
  - -q

Bootstrap Loader(Cont.)

- When loaded, this bootstrap loader:
  a) Loads the first 64k of the floppy disk (the first file in fact) at the physical address 1000h.
  b) Disables the interrupts.
  c) Jumps at 1000h to start executing the application.

Initializing the Hardware(Cont.)

- Once in *main()*, a call is done to *OsCpuInit()*, in os_cpu_c.c, in order to perform the following:
  - Enable the address line 20, normally disabled for some real mode considerations. The line is enabled by sending a few commands to the Intel 8042 keyboard controller. See *InitA20()* for details (os_cpu_c.c).
  - Relocate the IRQ interrupts, since the overlap the CPU interrupts and exceptions (for instance, it is not possible to know if the interrupt 0 has been triggered by the clock or a division by zero). The IRQ are relocated in the range 20h-2Fh by sending a few commands to the Intel 8259 interrupt controllers. See *InitPIC()* for details (os_cpu_c.c).

Initializing the Hardware(Cont.)

- The interrupt table is initialized by using *SetIntVector()* and *SetIDTGate()*. The 64 entries are set to point to a default interrupt handler (*DefIntHandler()*, in os_cpu_a.asm), which simply performs an interrupt return.
- The clock handler (*OsTickISR()*, in os_cpu_a.asm) is installed as the interrupt 20h handler.
- The MicroC/OS context switch handler (*OSCtxSw()*, in os_cpu_a.asm), is installed as the interrupt *uCOS* handler (*uCOS* is defined as 0x30 in os_cpu.h).