# Le noyau Temps Réel µC/OS II

email : kadionik@enseirb-matmeca.fr
web : kadionik.enseirb-matmeca.fr

**Patrice KADIONIK**
**ENSEIRB-MATMECA**

# Contents

# Introduction

- µC/OS II: acronym for Micro-Controller Operating Systems Version 2: it's a very small real-time kernel.
  - Memory footprint is about 20KB for a fully functional kernel
  - Source code is about 5,500 lines
  - mostly in ANSI C
- II is a highly portable, ROMable, very scalable, preemptive real-time, deterministic, multitasking kernel
- It can manage up to 64 tasks (56 user tasks available)

# Introduction

- It has connectivity with µC/GUI and µC/FS (GUI and File Systems for µC/OS II)
- It is ported to more than 100 microprocessors and microcontrollers
- It is simple to use and simple to implement but very effective compared to the price/performance ratio
- It supports all type of processors from 8-bit to 64-bit

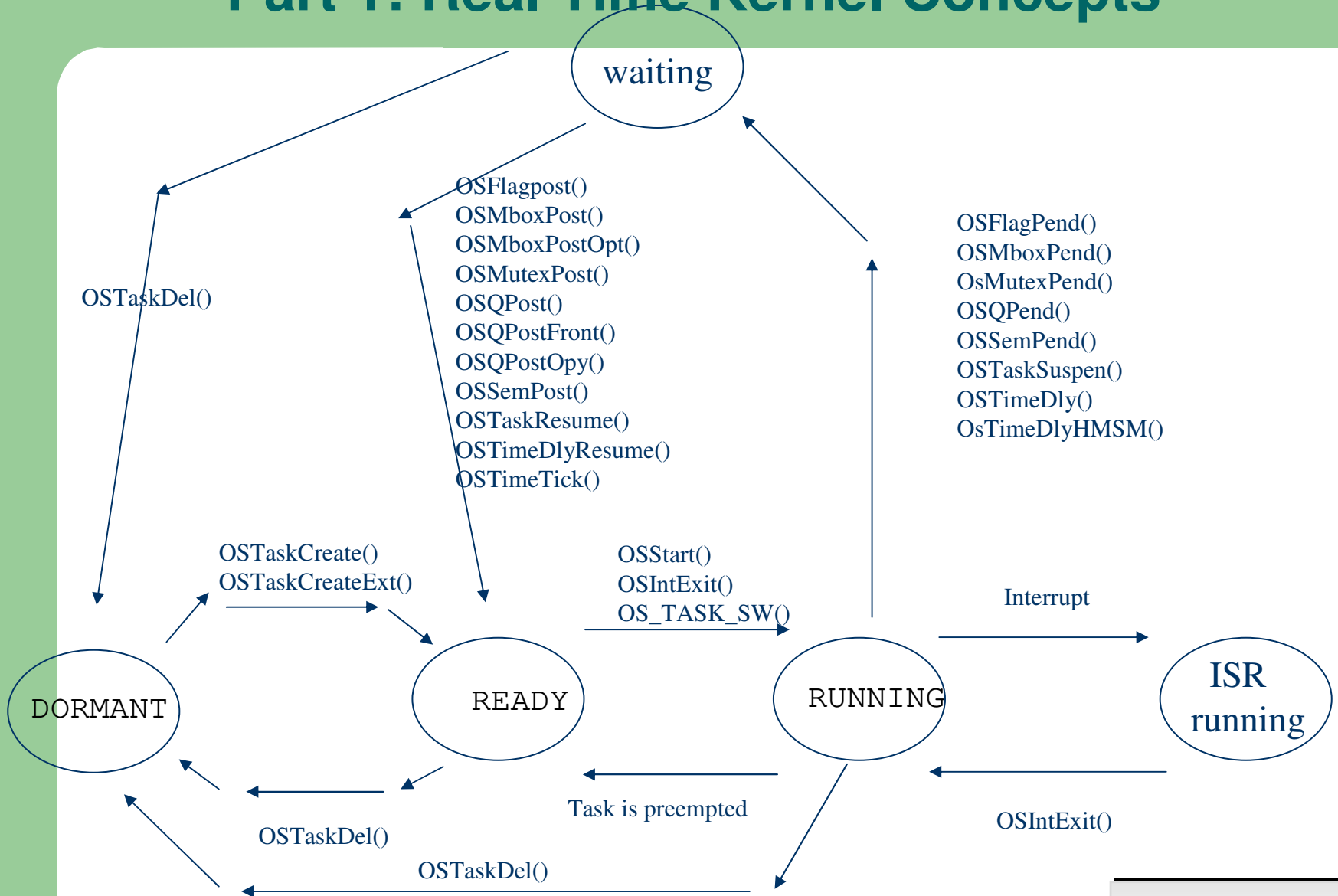
# Introduction

- Different ports from the official µC/OS-II
- Web site at http://www.ucos-ii.com
- Neither freeware nor open source code (freeware for academic purpose)
- Reference Book:
  - Jean J. Labrosse, “MicroC/OS-II: The Real-Time Kernel”
  - CMP Book, ISBN: 1-57820-103-9

# Part 1: Real Time Kernel Concepts

- Task (Thread) States
  - Task is a simple program that thinks it has the CPU all to itself. Each task has its priority, its own set of CPU registers and its own stack area.
  - States:
    - DORMANT
    - READY
    - RUNNING
    - DELAYED
    - PENDING
    - INTERRUPTED

# Part 1: Real Time Kernel Concepts



## Foreground/background systems ←→ Kernel

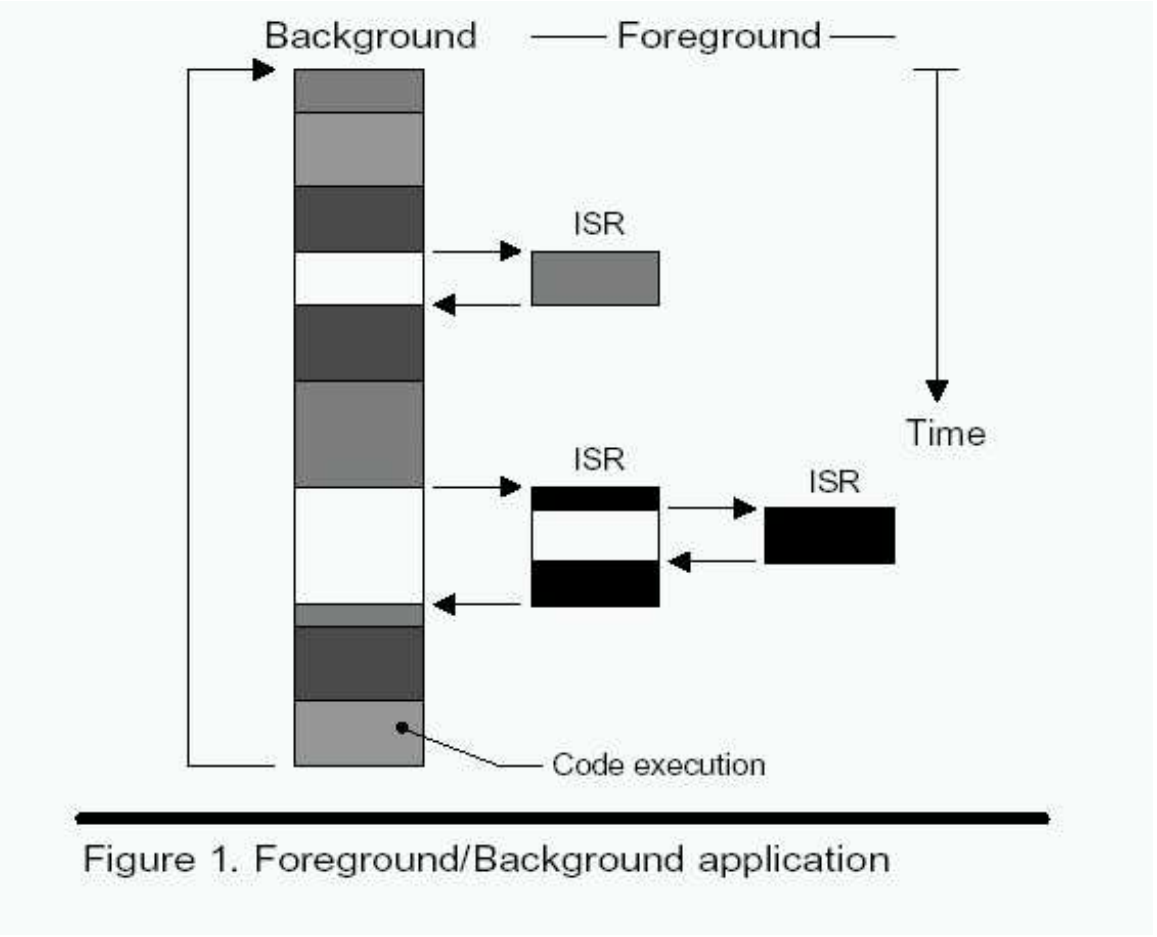
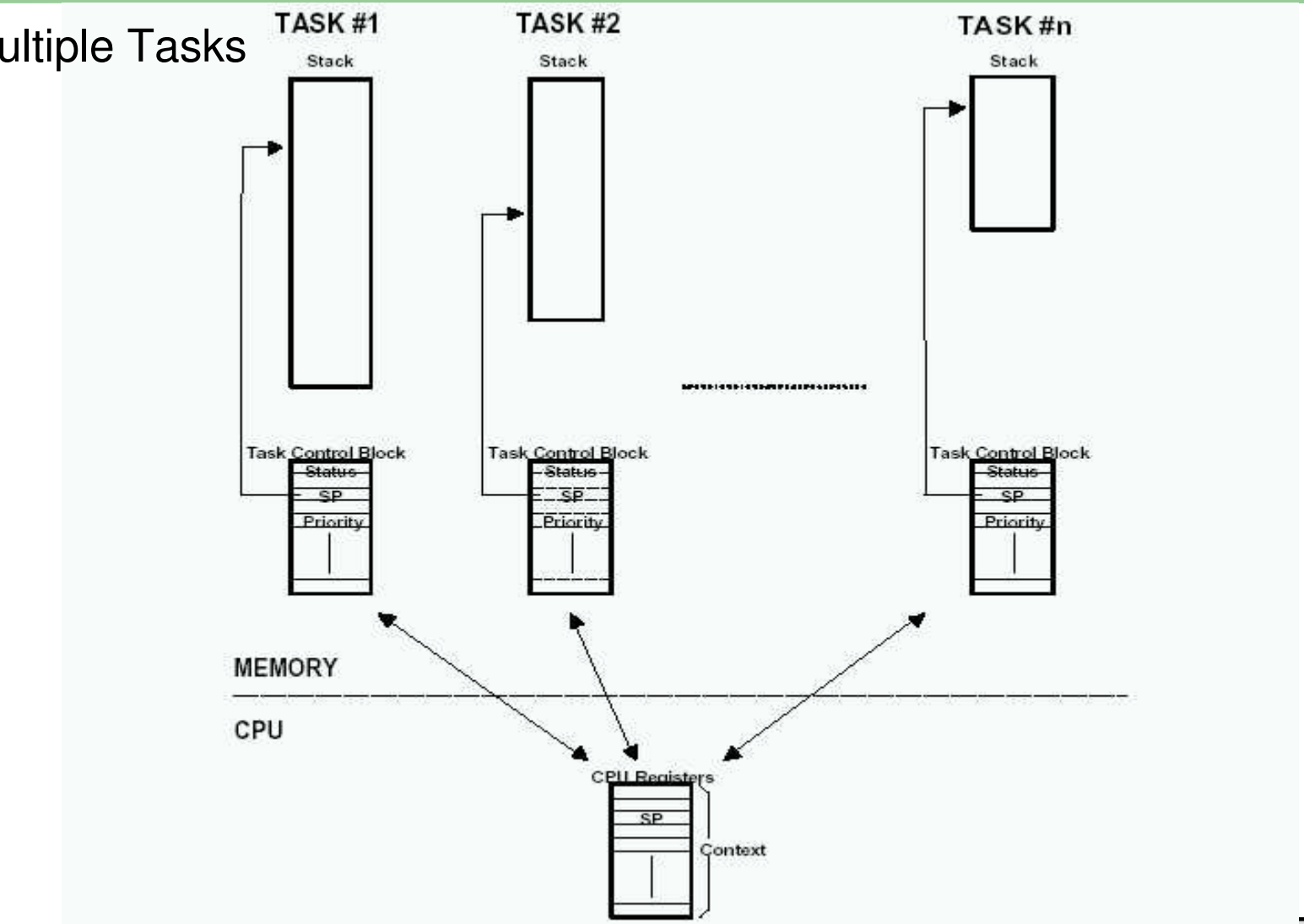



Figure 1. Foreground/Background application

# Part 1: Real Time Kernel Concepts

- Multiple Tasks



# Part 1: Real Time Kernel Concepts

- Resources:
  - I/O devices: printer,keyboard,display
  - memory: variable,structure,array
- Shared resources: mutual exclusion
- Task: ”small” process

- Kernel for RTOS: context switching
  - overhead 2%-5%
- Scheduler (dispatcher):
  - non-preemptivec= cooperative multitasking
  - preemptive

# Part 1: Real Time Kernel Concepts

- Preemptive Kernel vs. Non-Preemptive Kernel
  - Preemptive Kernel is used when system responsiveness is important. The highest priority task ready to run is always given control of the CPU. When a task makes a higher task ready to run, the current task is preempted (suspended and the higher priority task is immediately given control of the CPU. If a ISR makes a higher priority task ready, when the ISR completes, the interrupted task is suspended and the new higher priority task is resumed. )
  - Non-Preemptive kernel require that each task does something to explicitly give up control of CPU. It is also called cooperative multitasking. The kernel is much simpler to design than preemptive kernels. One of the advantages of a non-preemptive kernel is that the interrupt latency is typically low. Non-reentrant functions can be used by each task without fear of corruption by another task.

# Preemptive Kernel vs. Non-Preemptive Kernel

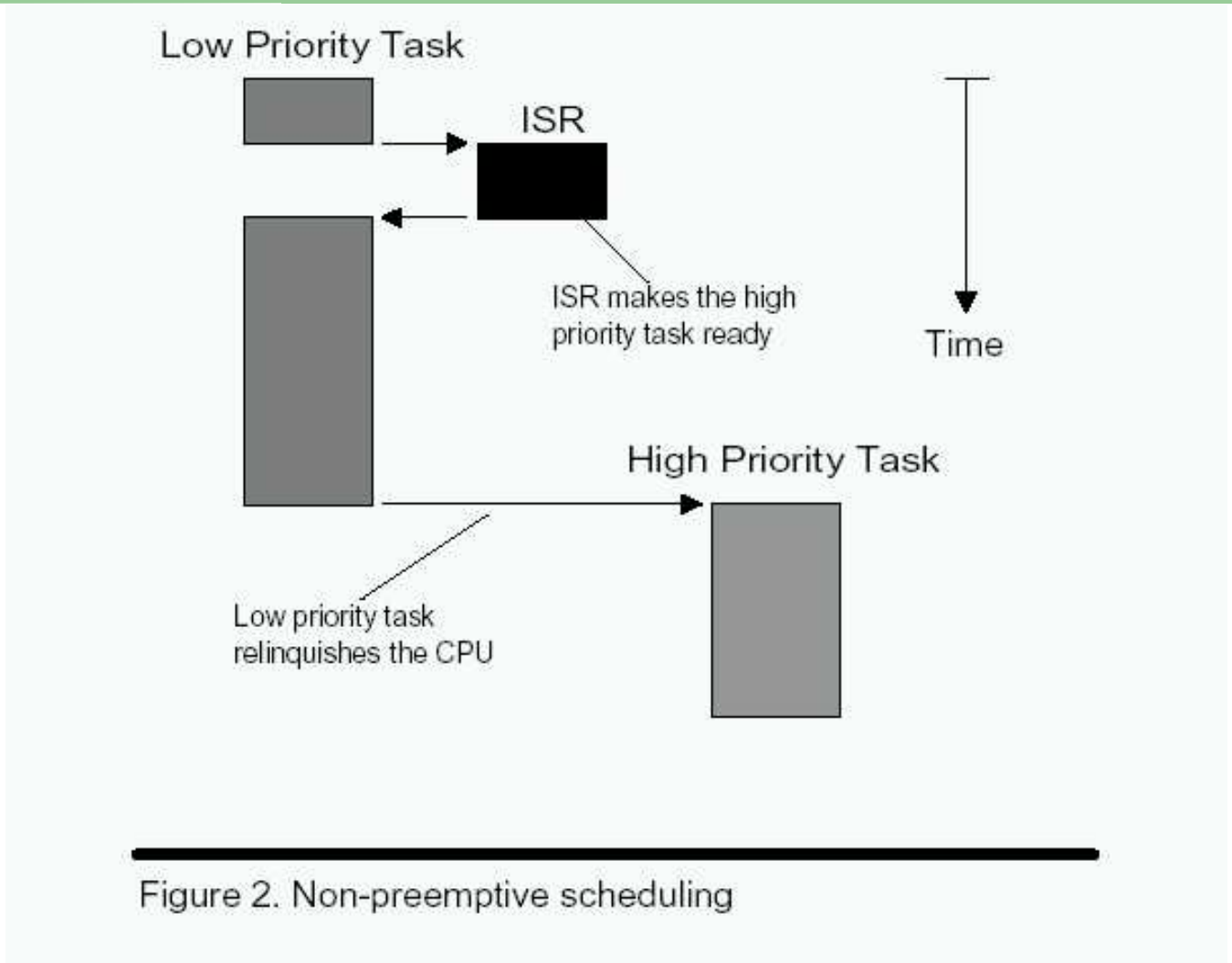



Figure 2. Non-preemptive scheduling

# Preemptive Kernel vs. Non-Preemptive Kernel

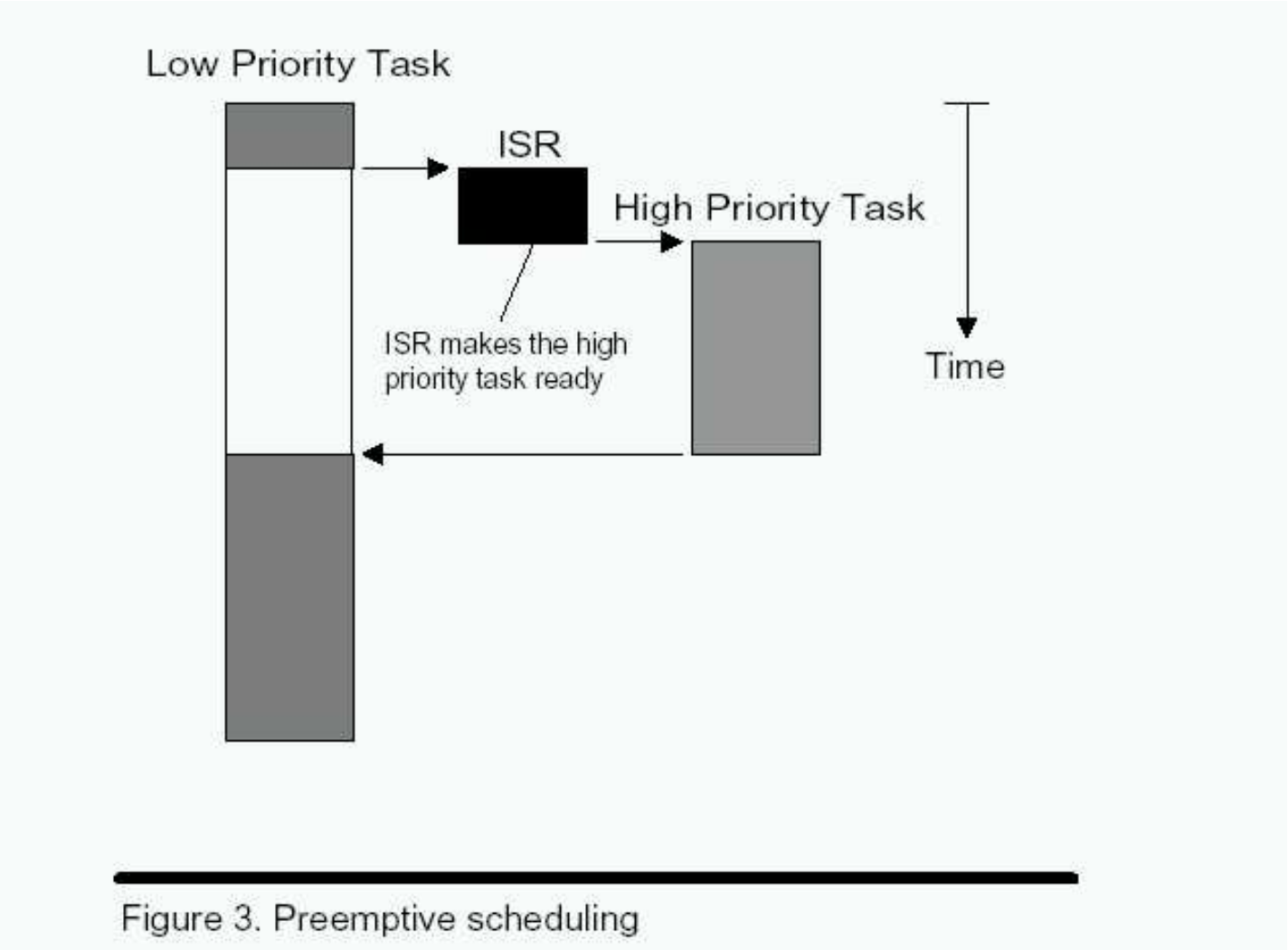
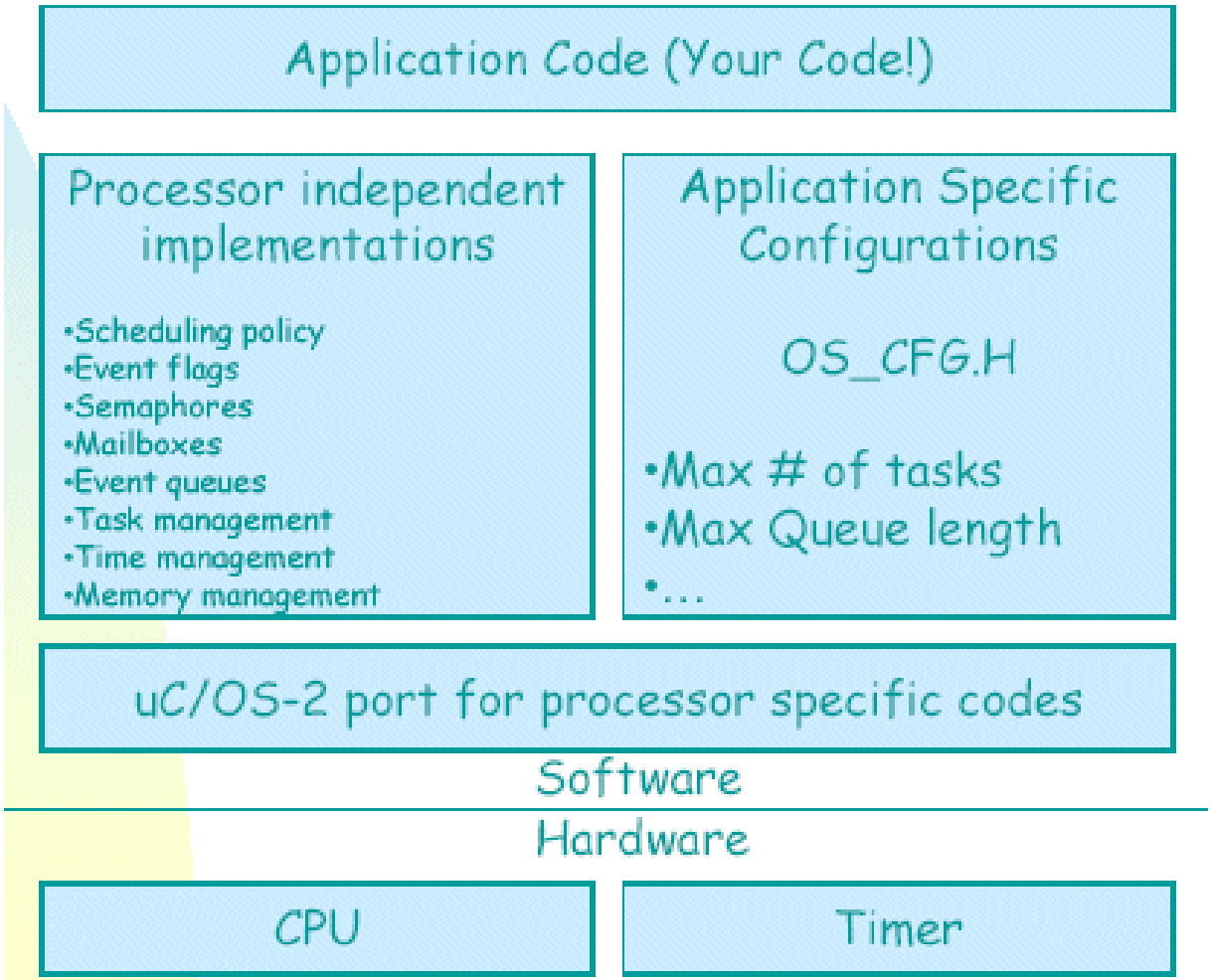



Figure 3. Preemptive scheduling

# Part 1: Real Time Kernel Concepts

- Round-Robin Scheduling: if 2 tasks have the same priority
- Static priorities: do not change during execution of task
- In Rate Monotonic Scheduling (RM) tasks with the highest rate of execution are given the highest priority

# Part 2: Kernel Structure



# Part 3: Interrupt Processing

- Interrupt Service Routine
- Interrupt Latency, Response and Recovery
- Clock Tick

# Interrupt Latency

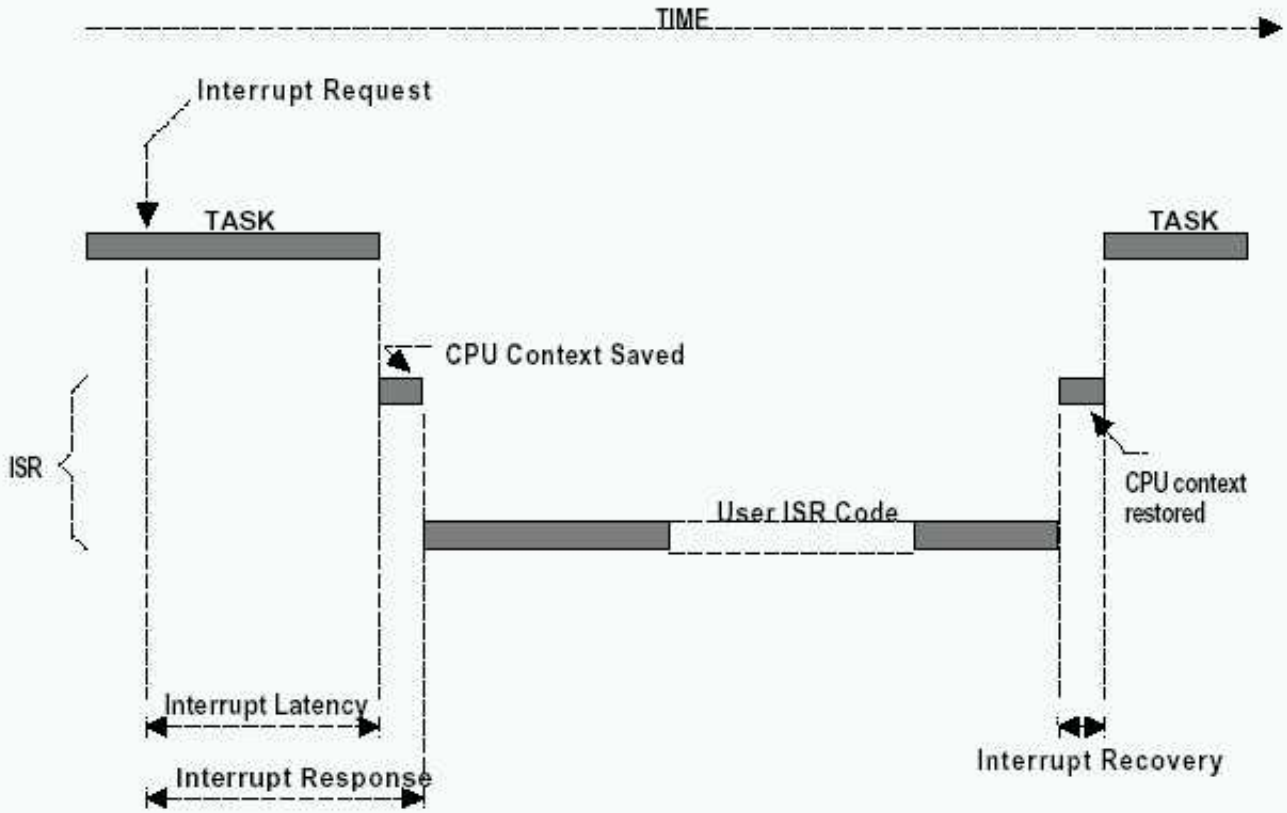



**Figure 2-20, Interrupt latency, response, and recovery (Non-preemptive kernel)**

# Interrupt Latency

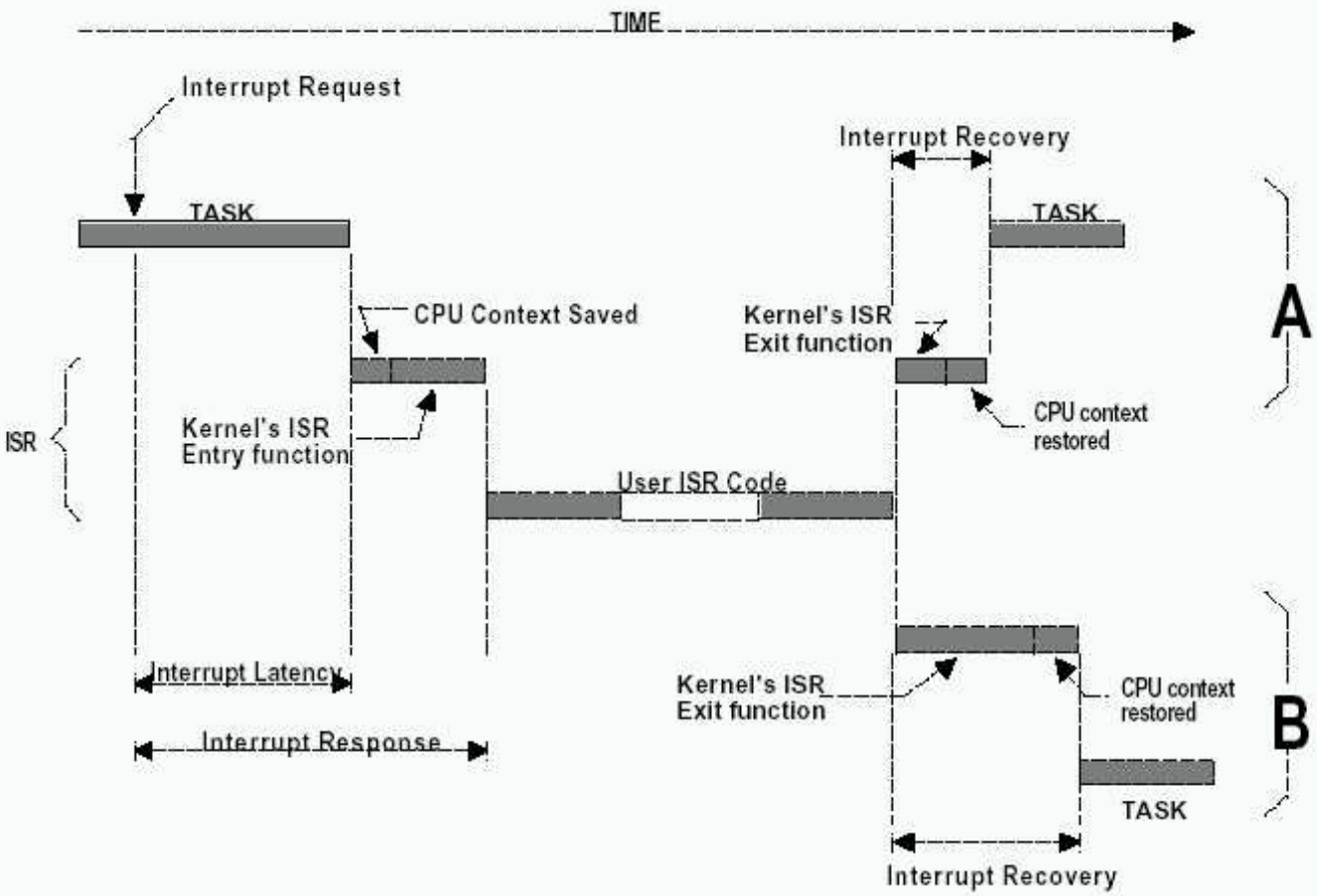
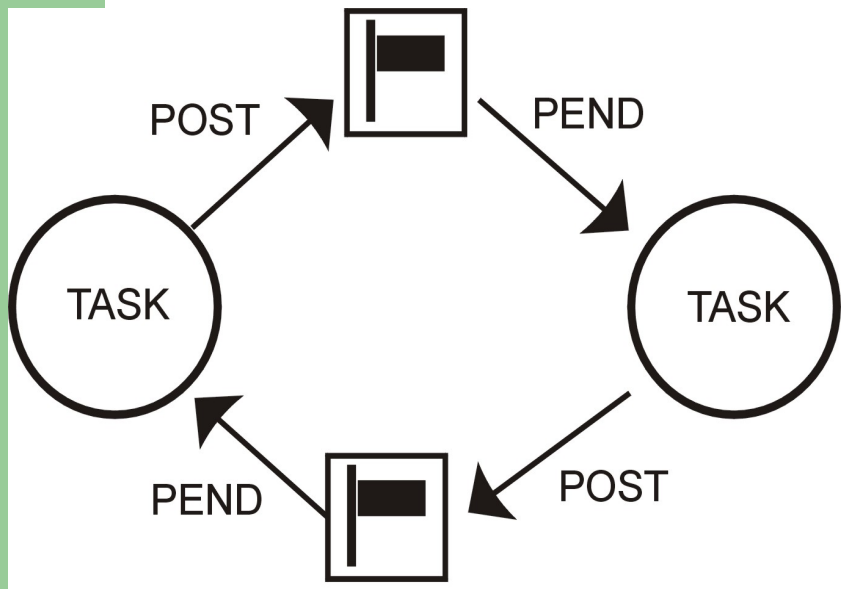



Figure 2-21, Interrupt latency, response, and recovery
(Preemptive kernel)

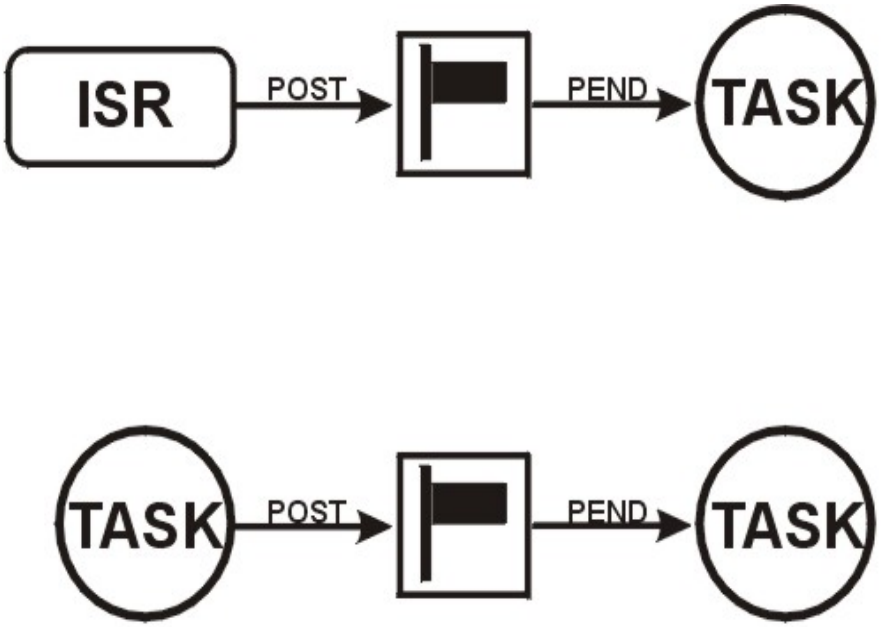
# Part 4: Communication

- Semaphores
- Message Mailboxes
- Message Queues

# Semaphores



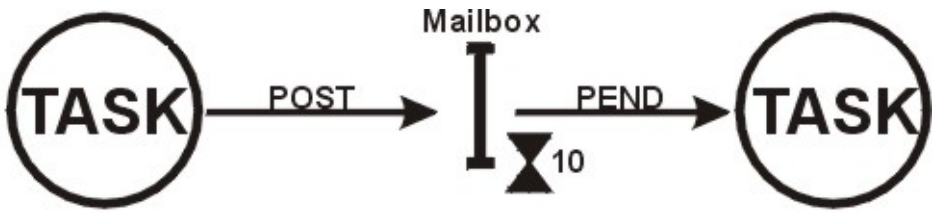

Synchronizing two Tasks




**Signaling Events through Semaphores**

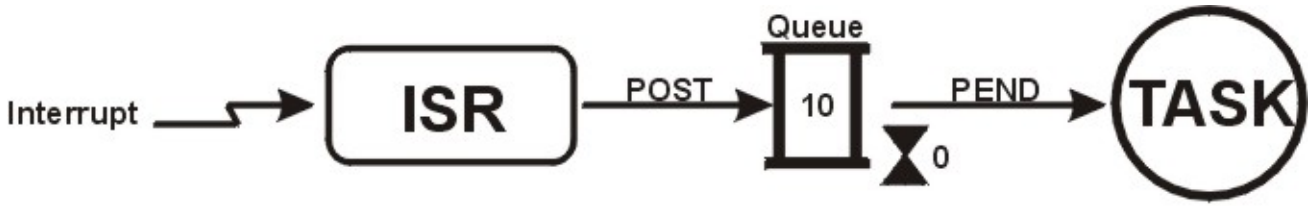
- Task synchronisation: the Rendez Vous!

# Mailbox and Queue




Note: POST deposites a pointer size variable in the mailbox

**Message Mailbox**




Note: POST deposites a pointer size variable in the queue

**Message Queue**

# Part 5: Init and Config

- Init and Config
  - OS_IDLE_TASK_STK_SIZE
  - OS_MAX_TASKS
  - OS_MAX_QS
  - OSInit(), OSStart()

# Part 6: Function Lists and Examples

- System Functions Lists (primitives)
  - OSInit() OSStart() OSStartHighRdy()
  - OSTimeDly() OSTimeTick() OSTickISR()
  - OSIntEnter() OSIntExit()
  - OSCtxSw() OSIntCtxSw()
  - OSSched() OSChangePrio()
  - OSTaskCreate() OSTaskDelete()
  - OSLock() OSUnlock()
  - OSSemInit() OSSemPost() OSSemPend()
  - OSMboxInit() OSMboxPost() OSMboxPend()
  - OSQInit() OSQPost() OSQPend()
  - OSTCBGetFree() OSTCBPutFree()

# Task Management

- Tasks may not return a value → type is always void
- Tasks must be:
  - Endless loop or
  - Delete themselves

```
void YourTask (void *pdata)
{
   for (;;) {
      /* USER CODE */
      Call one of uC/OS-II's services:
      OSMboxPend();
      OSQPend();
      OSSemPend();
      OSTaskDel(OS_PRIO_SELF);
      OSTaskSuspend(OS_PRIO_SELF);
      OSTimeDly();
      OSTimeDlyHMSM();
      /* USER CODE */
   }
}
```

# Task Management

- Maximally 64 tasks possible in µC/OS II
- Priority is a number between 0 and OS_LOWEST_PRIORITY (63)
- Priority is also task id
- Lowest priority has highest number: the idle task!
- 4 lowest levels and 4 highest levels reserved by uCOS
- For OS_LOWEST_PRIORITY see OS_CFG.H
- There is always one task present: the idle task

```
void YourTask (void *pdata)
{
   /* USER CODE */
   OSTaskDel(OS_PRIO_SELF);
}
```

# Task Management

- OSTaskCreate(void(*task) (void *pd), void *pdata, OS_STK *ptos,INT8U prio)
  - void(*task) (void *pd):
    - pointer to code task
    - Litteraly: task is a pointer to a function with argument *pd and returns void
  - Ptos is a pointer to the stack (see below) of the task
  - Prio is the priority
- Task Stacks

```
OS_STK  *pstk;

-
pstk = (OS_STK *)malloc(stack_size);
if (pstk != (OS_STK *)0) {           /* Make sure malloc() had enough space */
    Create the task;
}
```

# Task Management

- Stacks grow
  - from top to bottom or
  - bottom to top

When `OS_STK_GROWTH` is set to 1 in `OS_CPU.H`, you need to pass the highest memory location of the stack to the task create function as shown in listing 4.8.

```
OS_STK  TaskStack[TASK_STACK_SIZE];

OSTaskCreate(task, pdata, &TaskStack[TASK_STACK_SIZE-1], prio);
```

**Listing 4.8, Stack grows from HIGH memory to LOW memory**

```
void YourTask (void *pdata)
{
   /* USER CODE */
   OSTaskDel(OS_PRIO_SELF);
}
```

# Task Management

- Deleting task: goes to state dormant

- INT8U OSTaskDel(INT8U prio)
  - Prio is priority and task id
  - Returns OS_TASK_NOT_EXIST if task prio not found

# Task Management

- Freeing resources before deletion:

```
void RequestorTask (void *pdata)
{
    INT8U err;


    pdata = pdata;
    for (;;) {
        /* Application code */
        if ('TaskToBeDeleted()' needs to be deleted) {                    (1)
            while (OSTaskDelReq(TASK TO DEL PRIO) != OS TASK NOT EXIST) {  (2)
                OSTimeDly(1);                                             (3)
            }
        }
        /* Application code */                                            (4)
    }
}
```

# Task Management

```
void TaskToBeDeleted (void *pdata)
{
    INT8U err;


    pdata = pdata;
    for (;;) {
        /* Application code */
        if (OSTaskDelReq(OS PRIO SELF) == OS TASK DEL REQ) {                (1)
            Release any owned resources;                                    (2)
            De-allocate any dynamic memory;
            OSTaskDel(OS_PRIO_SELF);                                        (3)
        } else {
            /* Application code */
        }
    }
}
```

# Task Management

```
#define STACKSIZE 1024

OS_STK stack[STACKSIZE];

INT8U OS_result;


OS_result = OSTaskCreateExt(task1,
                            (void *)NULL,
                            &stack1[STACKSIZE-1],
                            TASK1_PRIO,
                            TASK1_PRIO,
                            &stack1[0],
                            STACKSIZE,
                            (void *)0,
                            0 );
```

# Task Management

```
void task1(void *arg) {
INT8U OS_result;

OS_result = 0;
counter1 = 0;

while(1) {
        counter1++;

        OSTimeDly(10);
}
}
```

# Critical Sections

- Critical section: the task must not be preempted. All Interrupts are disabled and the tick timer too!

- Use a µC/OS II primitive or a macro like cli()/sti() under Linux!

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .     /* You can access shared data in here */
    .
    OS_EXIT_CRITICAL();
}
```

```
void Function (void)
{
    OSSchedLock();
    .
    .     /* You can access shared data in here (interrupts are recoqnized) */
    .
    OSSchedUnlock();
}
```

# Semaphores

```
OS_EVENT *SharedDataSem;


void Function (void)
{
    INT8U err;


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

# Semaphores

- OSSemCreate(): creates new semaphore
- OSSemDel(): use with care!
- OSSemPend(),OSSemPost: see Practical Exercices

# Semaphores

```
OS_EVENT *sem;

INT8U OS_result;

sem  = OSSemCreate(0);

OSSemPend(sem, 0, &OS_result);

OSSemPost(sem);

OSSemPend(sem, 10, &OS_result); // Timeout of 10 ticks
      if (OS_result == OS_TIMEOUT) {
            ...
            goto fin;
      }
```

# Event Flags

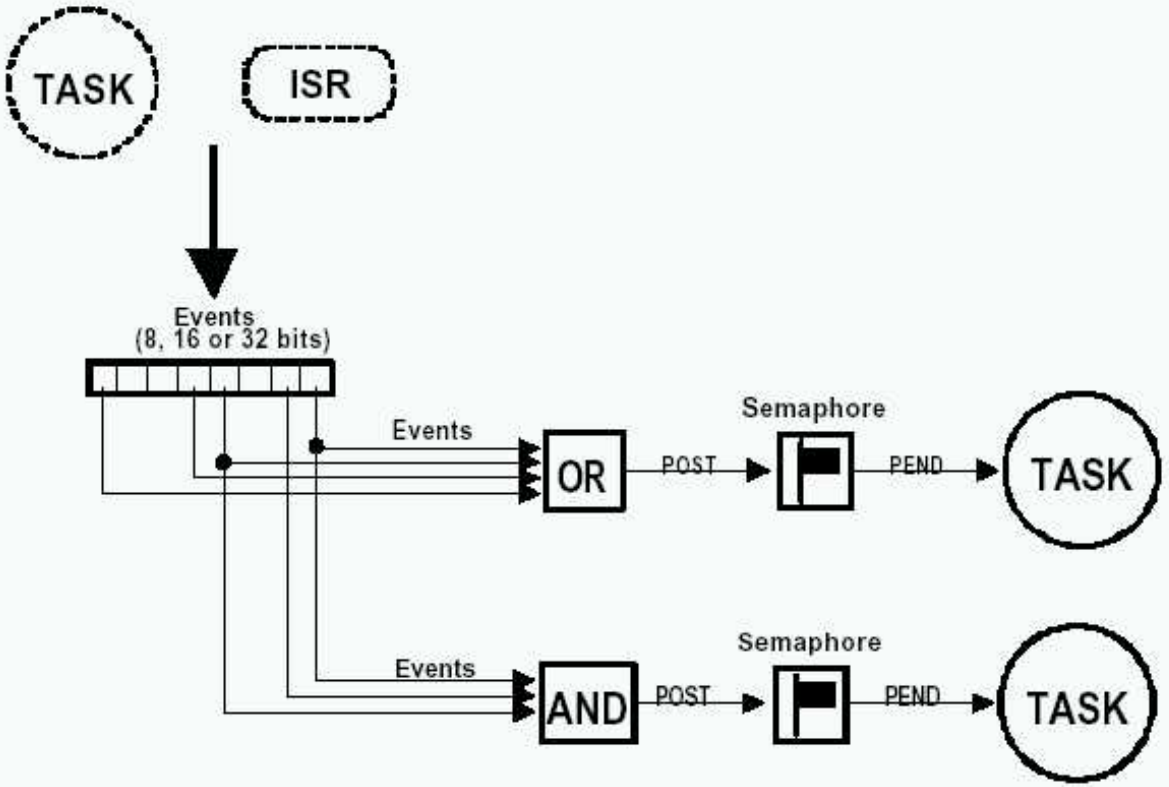



Figure 2-15, Event flags.

# Event Flags

- OSFlagCreate(): creates an Event Flag (a bit from a memory location)
- OSFlagDel()
- OSFlagPend(),OSFlagPost(): a task waits for a Event Flag or posts an Event Flag

# Mailbox

- OSMboxCreate(): creates a Mailbox
- OSMboxDel()
- OSMboxPend(),OSMboxPost(): a pointer (address) to the message (buffer) is used as argument

# Mailbox

```
OS_EVENT *mbox;

INT8U OS_result

char etat[2];
char *data;


mbox = OSMboxCreate((void *)0);

etat[0]=1; etat[1]=0;
OSMboxPost(mbox, etat);

data = (char *)OSMboxPend(mbox, 0, &OS_result);
if(data[1] == 1)
        …;
else
        …;
```

# Memory Management

- The Memory management includes:
  - Initializing the Memory Manager
  - Creating a Memory Partition
  - Obtaining Status of a Memory Partition
  - Obtaining a Memory Block
  - Returning a Memory Block
  - Waiting for Memory Blocks from a Memory Partition

# Memory Management

- Each memory partition consists of several fixed-sized memory blocks
- A task obtains memory blocks from the memory partition
- A task must create a memory partition before it can be used
- Allocation and de-allocation of these fixed-sized memory blocks is done in constant time and is deterministic
- Multiple memory partitions can exist, so a task can obtain memory blocks of different sizes
- A specific memory block should be returned to its memory partition from which it came

# Memory Management

- Problem with malloc() and free(): it can run out of memory
- Even if you calculate the memory, fragmentation can occur.
- Use memory blocks:
  - OSMemCreate()
  - OSMemGet()
  - OSMemPut()

# Time Management

- Clock tick: A clock tick is a periodic time source to keep track of time delays and time outs.
  - Tick intervals: 10 ~ 100 ms.
- The faster the tick rate, the higher the overhead imposed on the system
- When ever a clock tick occurs µC/OS-II increments a 32-bit counter
- The counter starts at zero, and rolls over to 4,294,967,295 (2^32-1) ticks.
- A task can be delayed and a delayed task can also be resumed

# Time Management

- Five services:
  - OSTimeDly(): delaying a task according a number of clock ticks
  - OSTimeDlyHMSM(): hours(H), minutes(M), seconds(S), milliseconds(m) Maximum Delay 256hours (11days) OSTimeDlyHMSM( 0, 0, 1, 500);
  - OSTimeDlyResume(): resuming a Delayed Task
  - OSTimeGet(): getting the current 32-bit counter
  - OSTimeSet()

# References


- µC/OS II: The Real-Time Kernel. J. Labrosse. CMP Editions
- The Real-Time Operating System uCOS-II. Anonymous
- INF4600 Systèmes temps réel. Ecole polytechnique de Montréal