Module 4 :

Real Time Systems Concepts with µCOSII

Contents

Introduction

- When an Embedded Software Development project begins, one question needs to be answered: “Whether this project needs an OS?”
- If the answer is yes, then there are 2 issues related :

  – Does the system have enough resources to accommodate an OS?

  – Is OS really needed for this system?

Writing program for one single task is more easy!

- two or three is fine, but more than five tasks simultaneously is difficult without help.
- You need to take care of
  - – Memory management
  - – Maintain task context (register contents, SP, PC, etc.)
  - – Scheduling
  - – Device drivers Etc.
- OS will take care of all these and you can concentrate more on application.

Difference between RTOS & Other operating system

1. All the devices have to respond user/ Environment within specified time.
2. In other operating system, OS takes control of machine first, then applications are executed. In RTOS, applications are linked and executed first, then OS gets executed.
3. Many RTOS do not protect themselves as carefully as other OS does. E.g. Passing a pointer is valid or not is checked in other OS but not in RTOS.
4. RTOS only includes those functions which are needed for execution.

RTOS issues

- Interrupt Latency should be very small
  - – Kernel has to respond to real time events.
  - – Interrupts should be disabled for minimum possible time.
- For embedded applications Kernel Size should be small
  - – Should fit in ROM.
- Sophisticated features can be removed
  - – No Virtual Memory
  - – No Protection

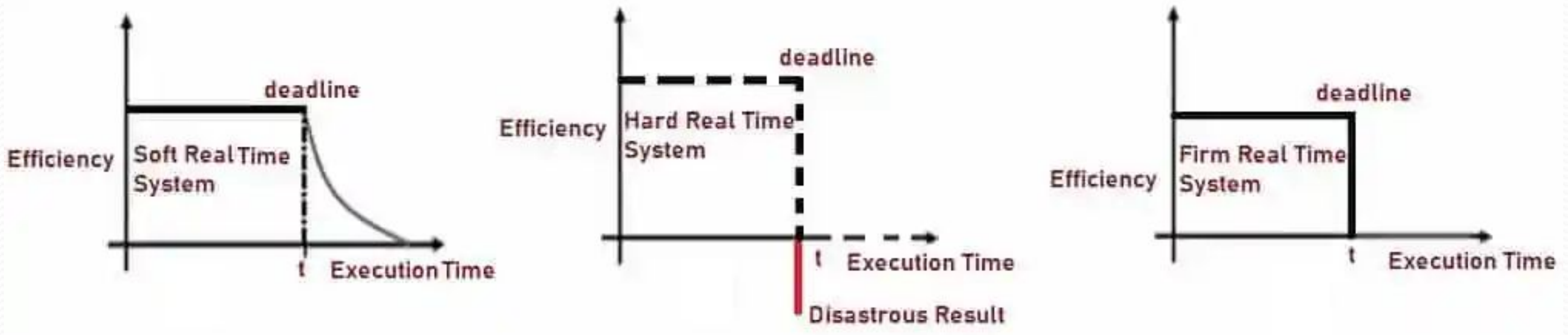
RTOS issues

Thumb-rule: If your application is hard real-time, you should generally use an RTOS.

- RTOS is structured so that only the needed components are included in the RTOS image. Other features are added as necessary.
- Scalability makes RTOS widely applicable to small, single-processor applications and to large, distributed ones.
- Critical operations must be performed by ISR, so that they are dealt with timely

Types of RTOS

- Soft Real Time Operating System
- Hard Real Time Operating System
- Firm Real Time Operating System



RTOS Types

- In Soft real time operating system, tasks are performed as quickly as possible, but they don’t have to be finished by specific time.
- In hard real time systems, not only they have to be performed quickly but also in specific time. Otherwise there is a major failure.

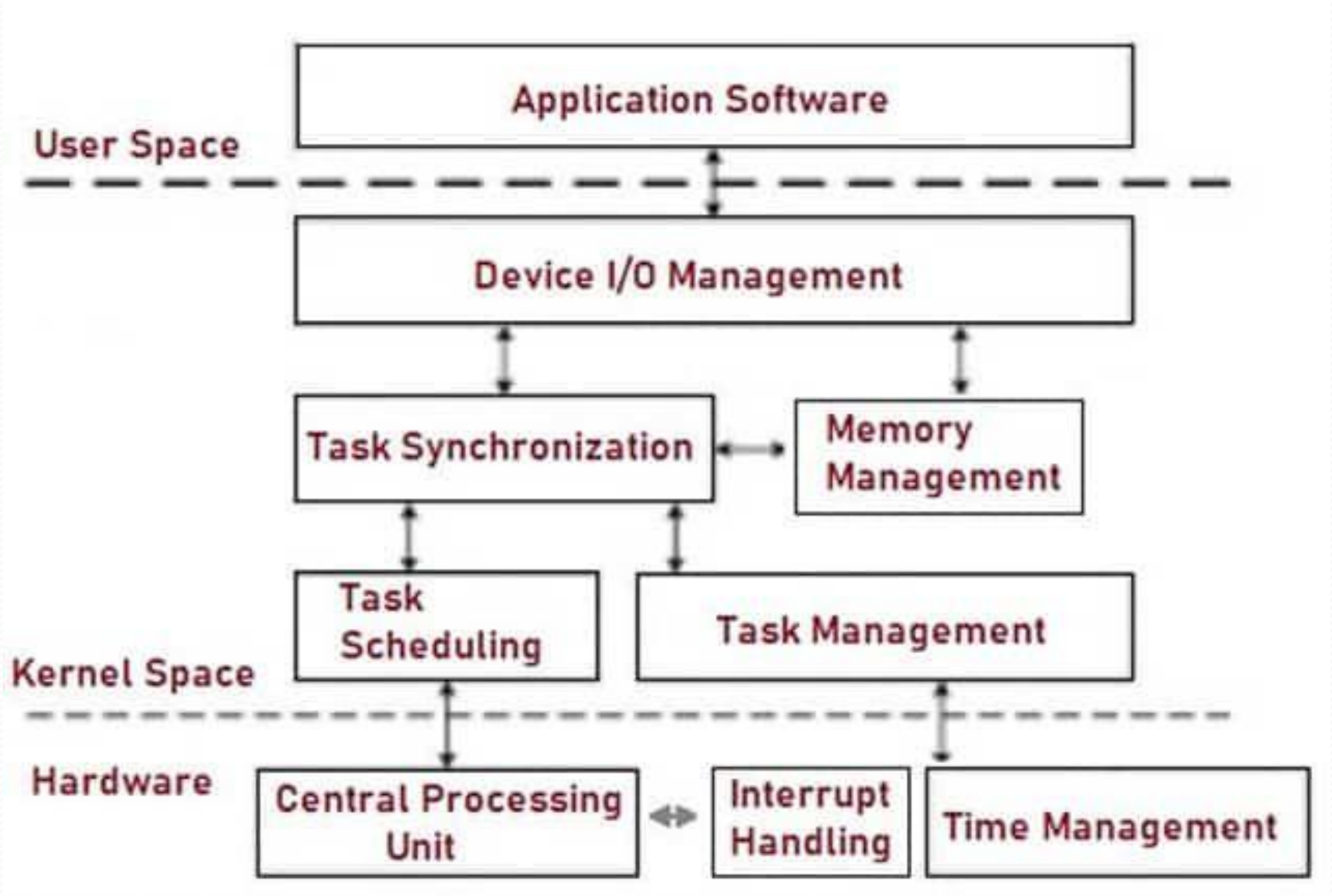
Which RTOS

- Performance
  - – Worst case interrupt latency
  - – Minimum context switch time
- Reliability: is it used for safety critical applications – such as avionics, life support medical systems…
- Footprint and data-memory requirement: is it small enough to fit in available resources?

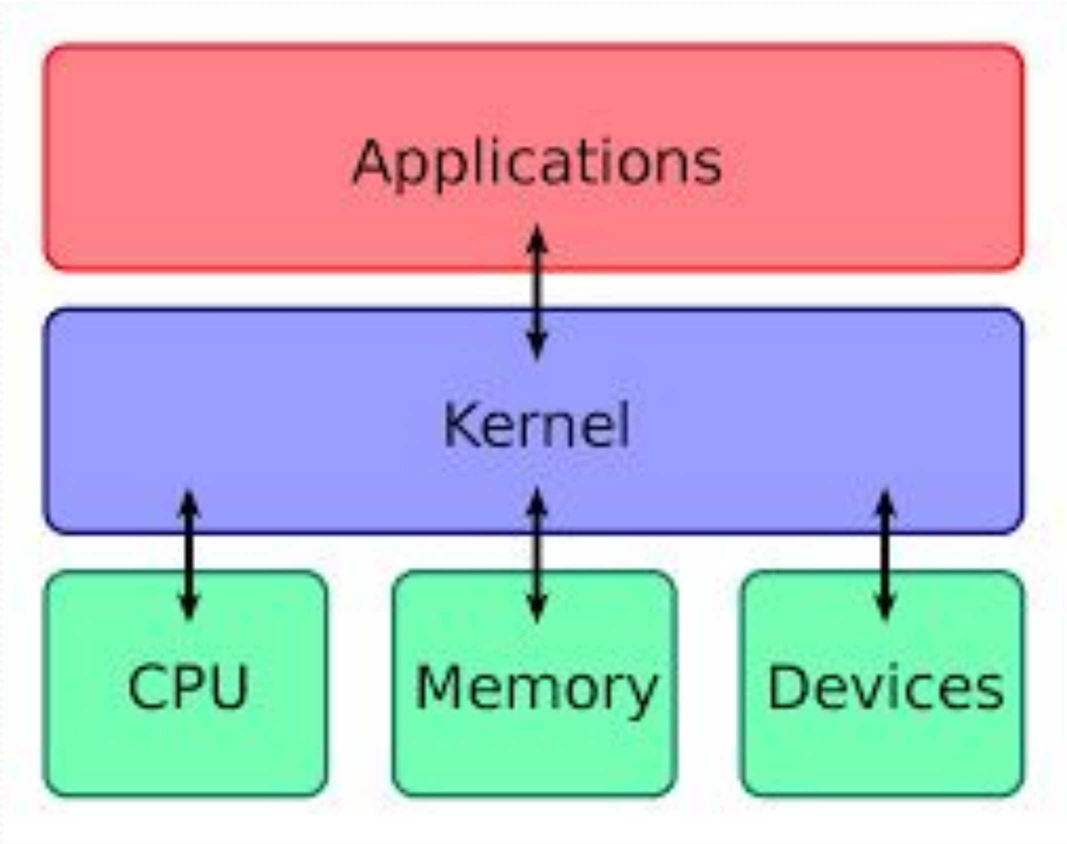
- Does it support the target processor?
- Are good development tools (like Compiler) available? At what cost?
- Is the RTOS available in the form of source code? Or object code?
- What is the basic cost? And royalty?
- Is it compatible with standards?

- Are the needed device drivers available? If not, we will have to write those.
- Are other needed components available? e.g. TCP/IP stack or GUI?
- Other:
  - What is the reputation of vendor?
  - Is the RTOS matured? Since when is it used?
  - Is technical support available? At what cost?

RTOS Architecture



RTOS Architecture



Basic RTOS Functions

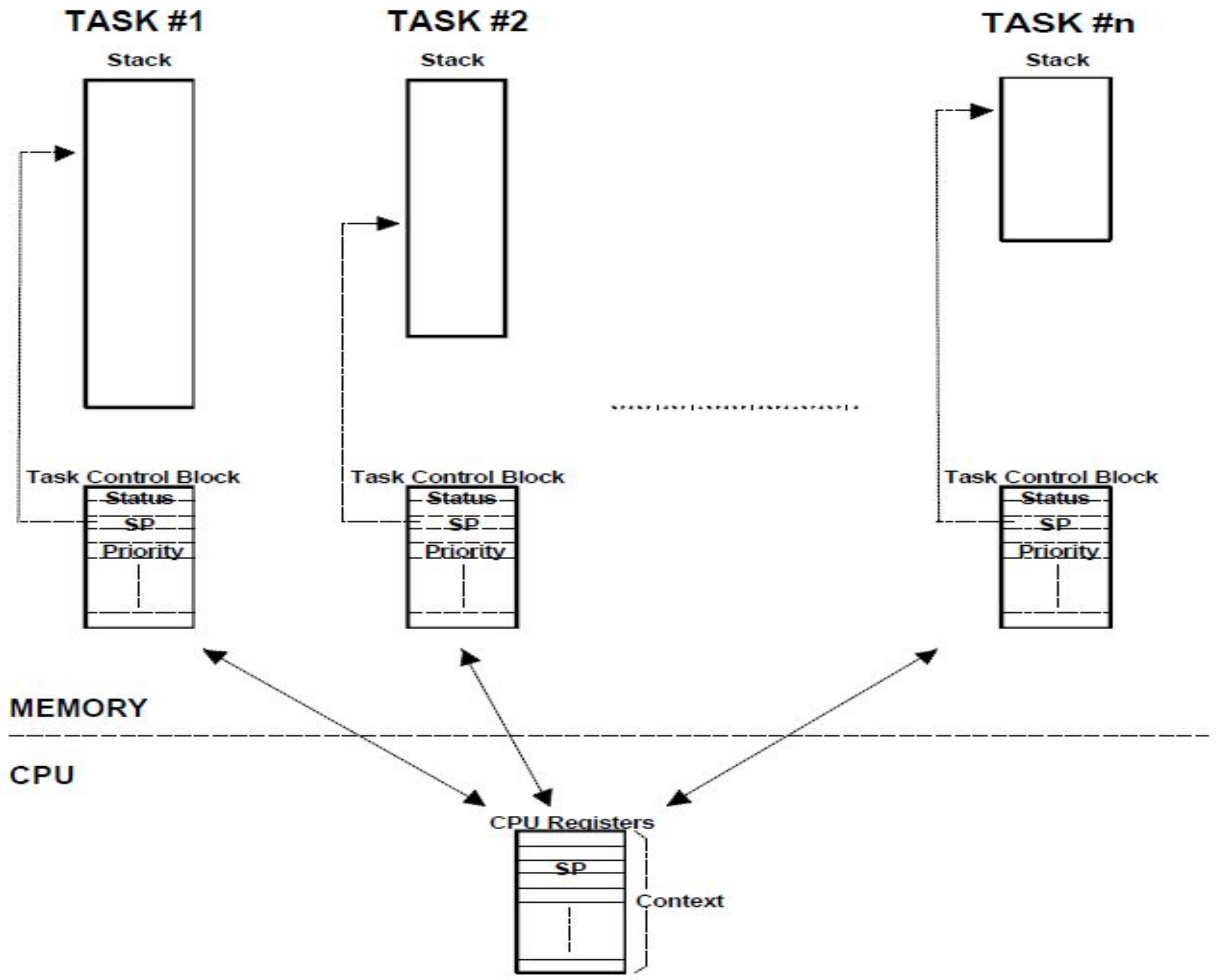
- Task management
- Memory management
- Interrupt handling
- Exception handling
- Task synchronization (IPC)
- Task scheduling
- Scalable architecture
- Resource management (Memory)

Real Time Systems Concept

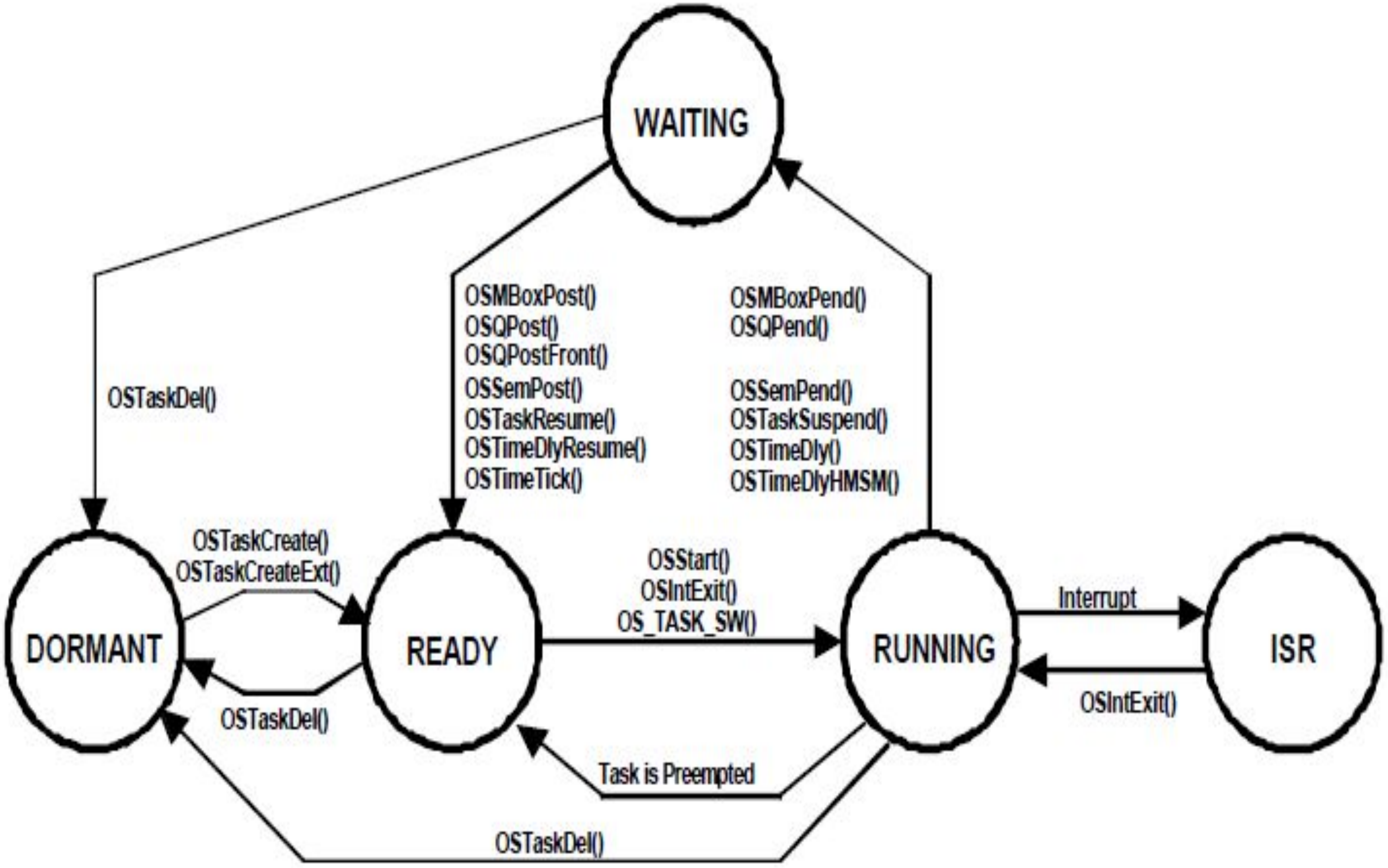
- Task
- Critical Section of code
- Foreground/background systems
- Resources
- Shared Resources
- Multitasking
- Context Switching
- Reentrant Function
- Kernels
- Interrupts
- Scheduler
- Mutual Exclusion

TASK

- Basic notion in RTOS
- Task or thread is lightweight process.
- Task is a simple sequential program that thinks it has the CPU all to itself.
- Real-time application design splits problem into different tasks. It may use system resources.
- A TCB (Task Control Block) contains task stack pointer, priority number, delay time value, task state, and some useful information for this task.


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

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()

Task States

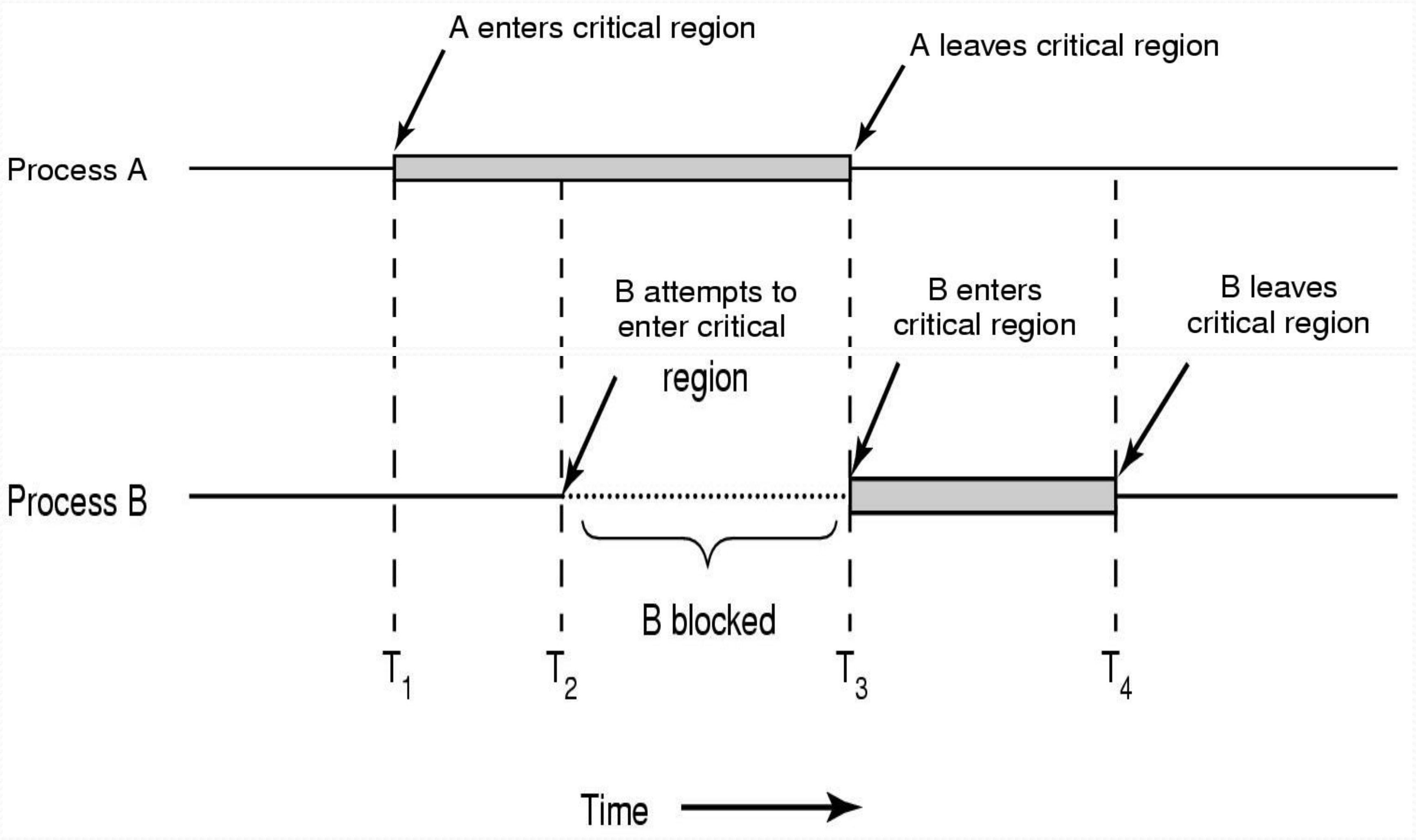
- The DORMANT state corresponds to a task which resides in memory but has not been made available to the multitasking kernel.
- A task is READY when it can execute but its priority is less than the currently running task.
- A task is RUNNING when it has control of the CPU.
- A task is WAITING for an event when it requires the occurrence of an event (waiting for an I/O operation).
- Finally, a task is INTERRUPTED when an interrupt has occurred and the CPU is in the process of servicing the interrupt.

Critical section of code

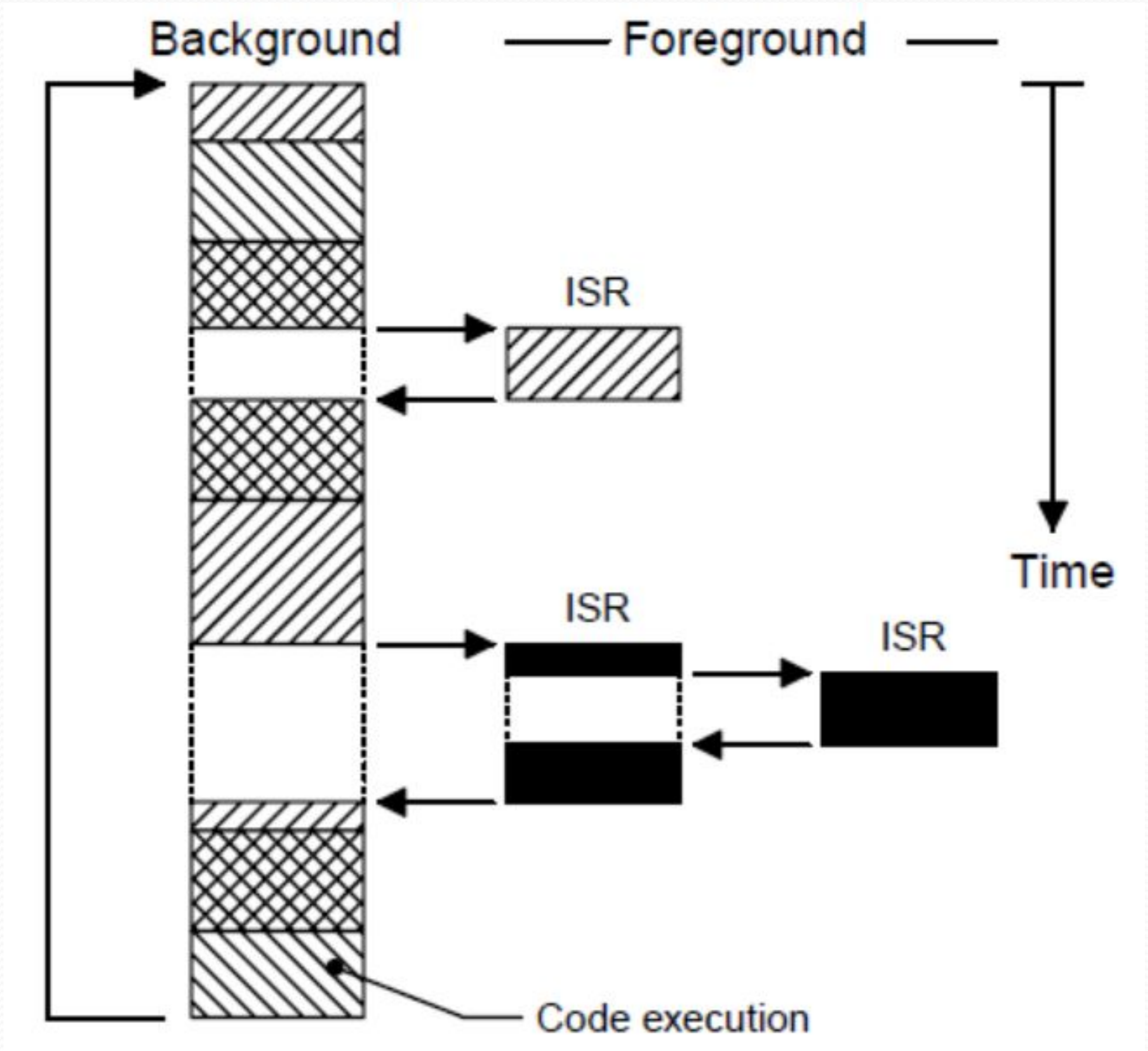
- A critical section of code also called ‘Critical region’, is the code that needs to be executed indivisibly. Once the section of the code starts executing, it must not be interrupted.

- To ensure this, interrupts are typically disabled, before the critical code is executed and enabled when the critical code is finished.

What we would like to happen



Foreground/background systems



Resources

- Resource can be any entity used by a task. For e.g. It can be printer, a display or a variable, array etc.
- Shared Resource: A shared resource is that which can be used by more than one task.
- Each task should gain exclusive access to the resource to avoid data corruption. This is called mutual exclusion.

Shared Resources

- Each task should gain exclusive access to prevent data corruption. Since each task exist in single address space, all tasks can reference global variables, pointers, buffers, etc.
- A common requirement on a real-time program with shared resources is that only one task at a time is allowed access to a shared resource.
- This can be formulated as a wish to obtain mutual exclusion between the tasks that need access to a shared resource.
- The techniques used for mutual exclusion are:
  - Disabling interrupt.
  - Disabling scheduling
  - Semaphores

Multitasking

- Multitasking is a technique used to handle the execution of multiple task.
- For a system to be real-time, there must be a guaranteed time frame where within a task must execute.
- In a multitasking RTOS the task scheduling, switching and execution elements are key to the effectiveness of the OS in achieving its real-time requirements.
- Thus, multitasking is the process of scheduling and switching the CPU between several tasks i.e. backgrounds.
- Multitasking maximizes the utilization of the CPU. It also provides modular construction of the application.
- It allows the application programmer to manage complexity in real time applications.

Context Switch

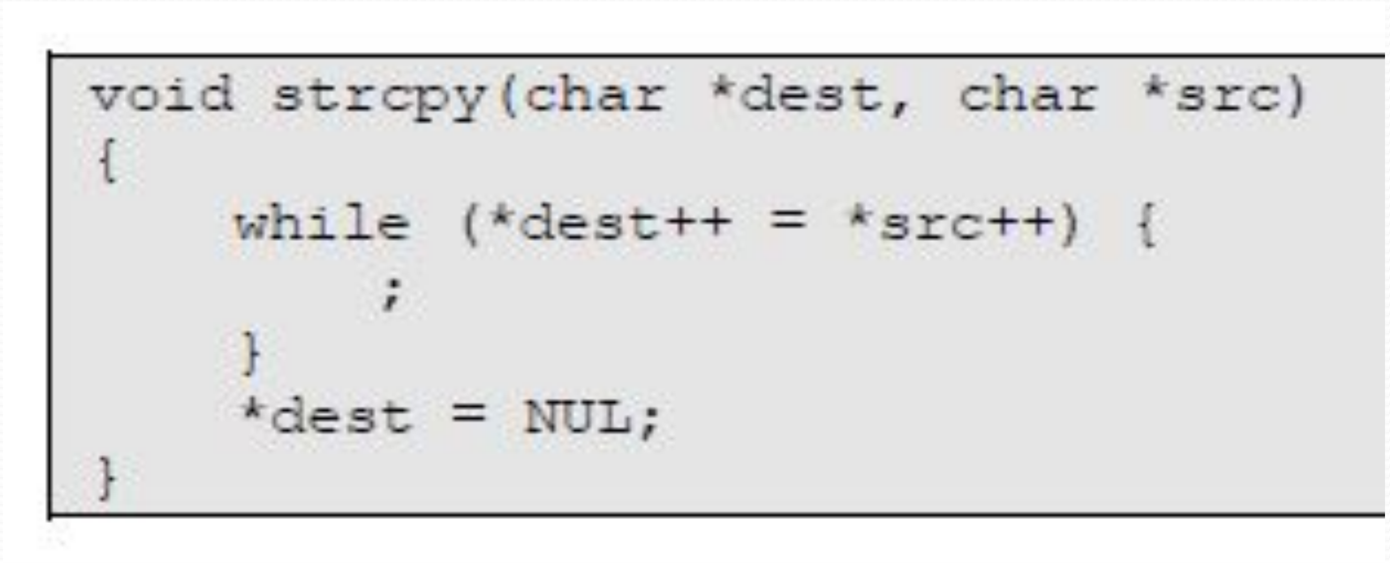
- Store current tasks context (CPU Registers) on its stack and load new tasks context from its stack and resume execution of new task.
- Overhead to the application.
- The more registers the CPU has, the higher is the overhead.
- So context should not be too long or too small, so that some information may get lost.

Reentrant Function

- A reentrant function is a function that can be used by more than one task without the fear of data corruption. A reentrant function can be reentered at any time and resumed at any time without the loss of data.
- Reentrant function either uses local variable (i.e. CPU registers or variables on stack) or protect data when the global variables are used.
- The same function can be invoked multiple times.

  An example of reentrant function is as follows

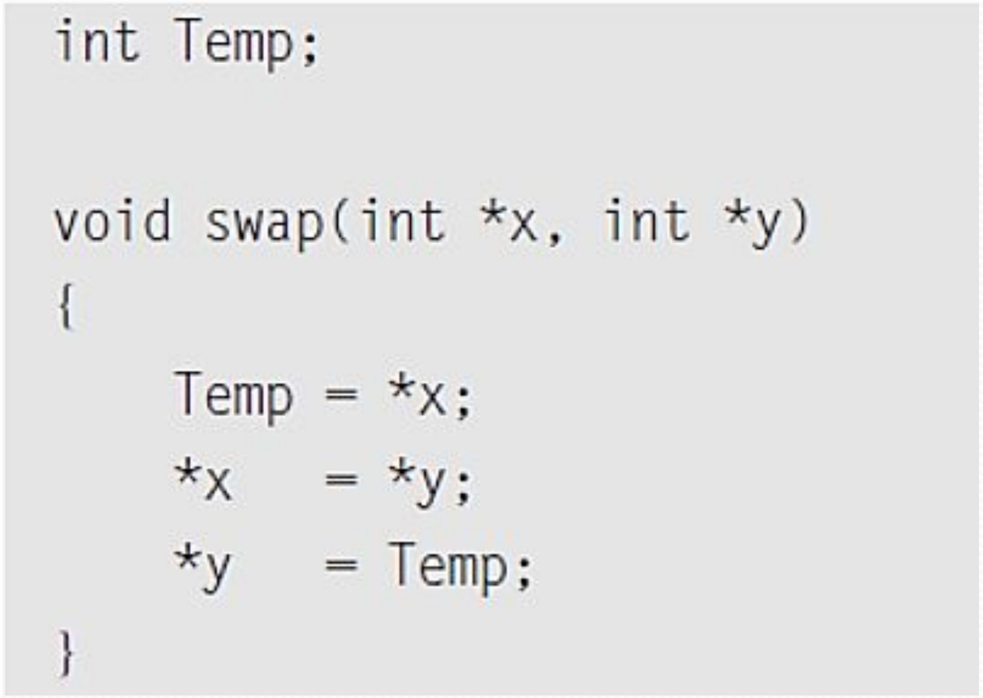
Ex



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

Non Reentrant Function

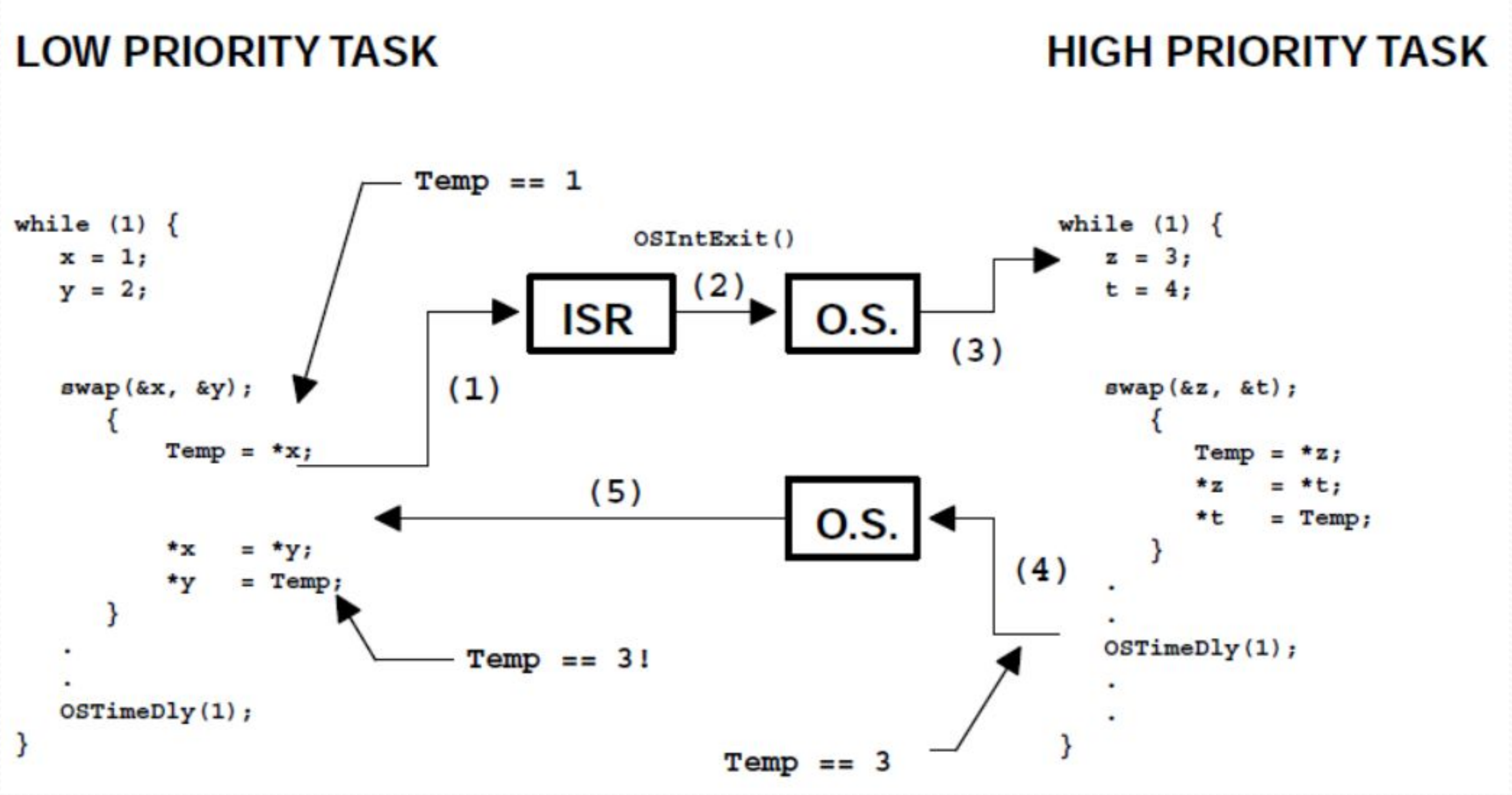
A **non-reentrant** function is one that cannot be shared by more than one task **unless mutual exclusion to the function is ensured** either by using a semaphore or by disabling interrupts during critical sections of code.



```
int Temp;

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

Non Reentrant Function



```
while (1) {
   x = 1;
   y = 2;


   swap(&x, &y);
      {
         Temp = *x;


         *x    = *y;
         *y    = Temp;
      }
   .
   .
   OSTimeDly(1);
}
```

```
while (1) {
   z = 3;
   t = 4;


   swap(&z, &t);
      {
         Temp = *z;
         *z   = *t;
         *t   = Temp;
      }
   .
   .
   OSTimeDly(1);
   .
   .
}
```

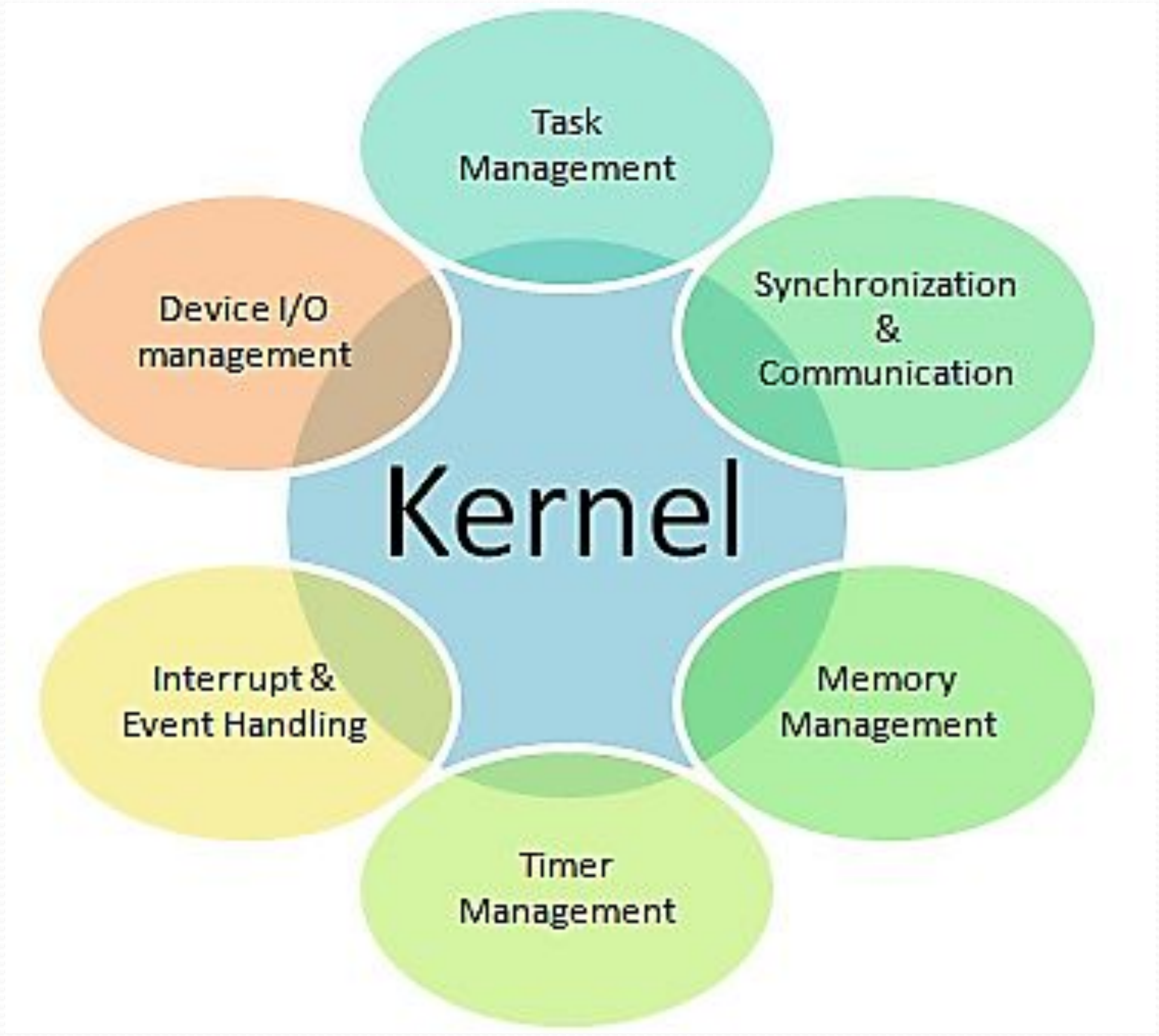
Techniques to make swap() reentrant

- Declare Temp local to swap().
- Disable interrupts before the operation and enable them afterwards.
- Use a Semaphore.

Kernel

- The Kernel is the part of multitasking system responsible for the management of the tasks.(i.e. managing CPU's time) and communication between tasks.
- Overhead to the system
- Requires extra ROM (code)
- Requires extra RAM for kernel data structures
- Each task requires its own stack (RAM) space
- Consumes CPU time (between 2 to 5%)

Kernel Architecture



KERNAL

- Once the system is designed using real-time kernel you don’t have to worry about…
  - – Task management (creation, termination, change priority)
  - – Interrupt handling
  - – Memory management
  - – Exception handling (timeouts, deadlocks,…)
  - – Task scheduling
  - – Time management
- There are two types of kernel, **non-preemptive** and **preemptive kernel**

Non-preemptive kernel

- Each task does something to give-up control of CPU. To maintain the illusion of concurrency, this process must be done frequently.
- Non-preemptive scheduling is also called cooperative multitasking; tasks cooperate with each other to share the CPU.
- Asynchronous events are handled by ISRs. ISR always returns to interrupted task.
- The new higher priority task will gain control of the CPU only when the current task gives up the CPU.

Non-preemptive kernel

- Advantage:

  – One of the advantages of a non-preemptive kernel is that interrupt latency is typically low.

  – simple task design since no fear that the task will get preempted (switched).

  -Non-reentrant functions can be used by each task.

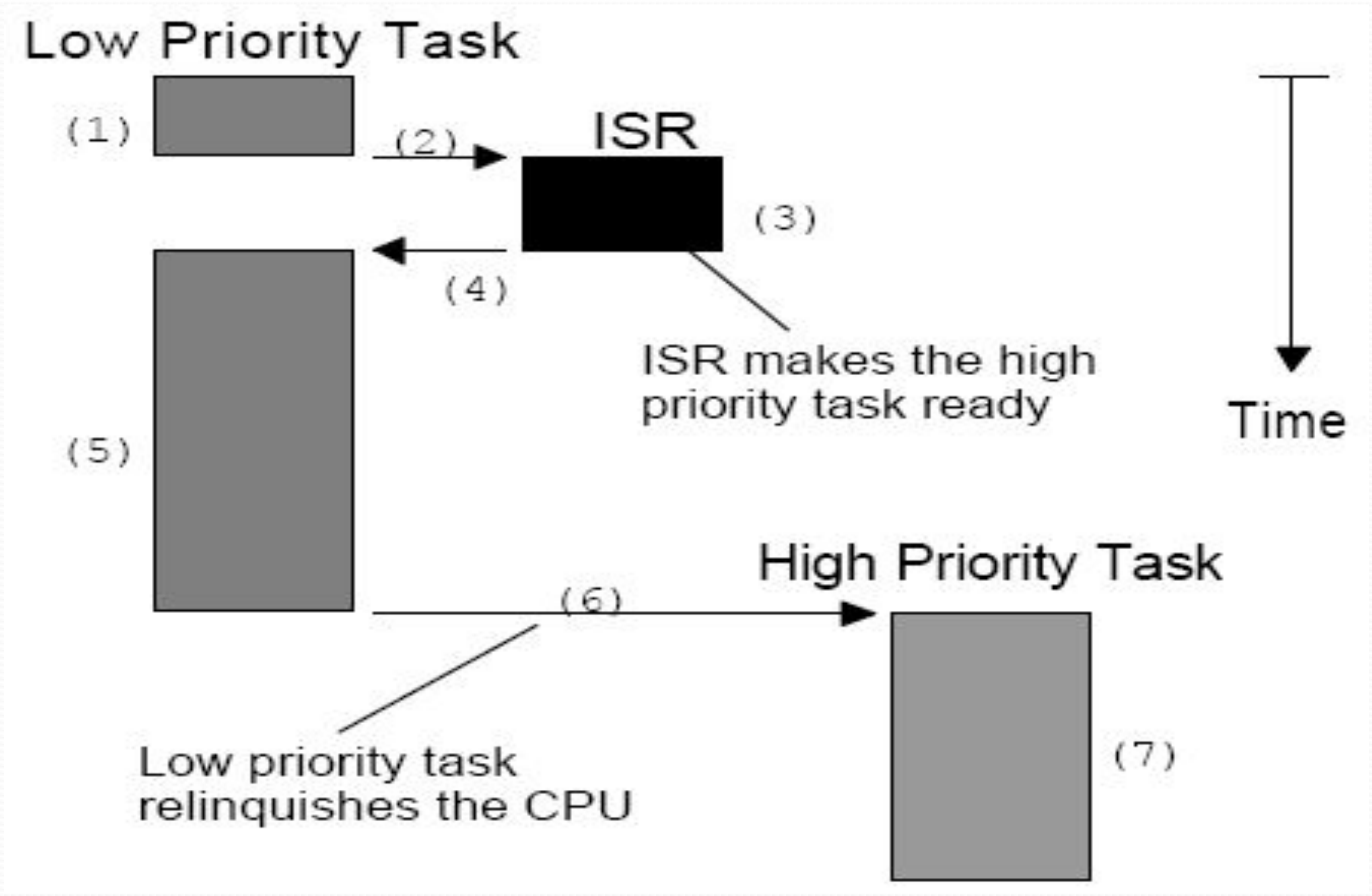
  -Another advantage is the lesser need to guard shared data through the use of semaphores.

- Disadvantage:

  –Biggest disadvantage is task responsiveness, a high priority task have to wait until the low-priority task gives up the CPU

  – Very few commercial kernels are non-pre emptive.

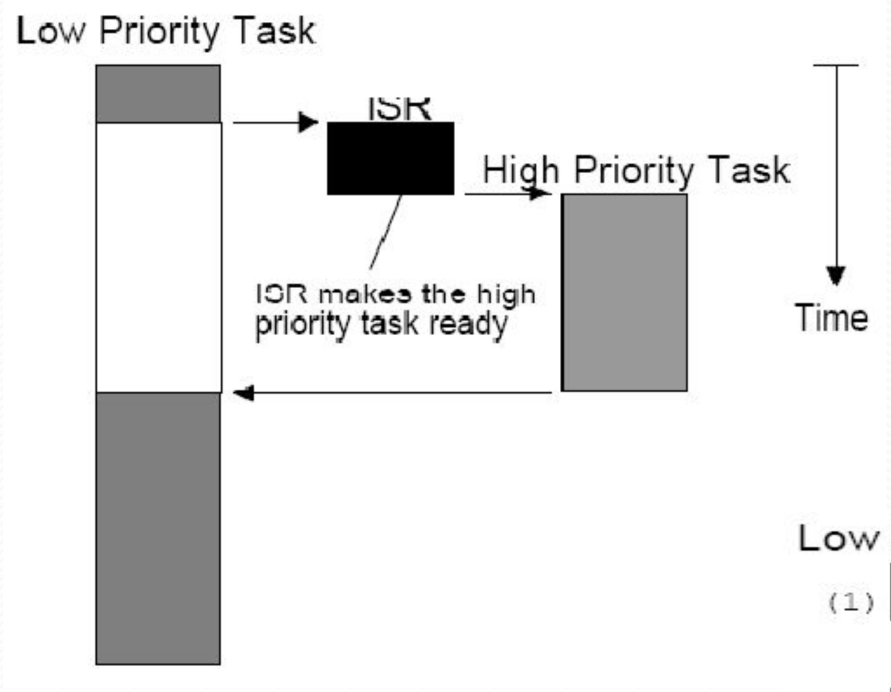
Non-preemptive kernel

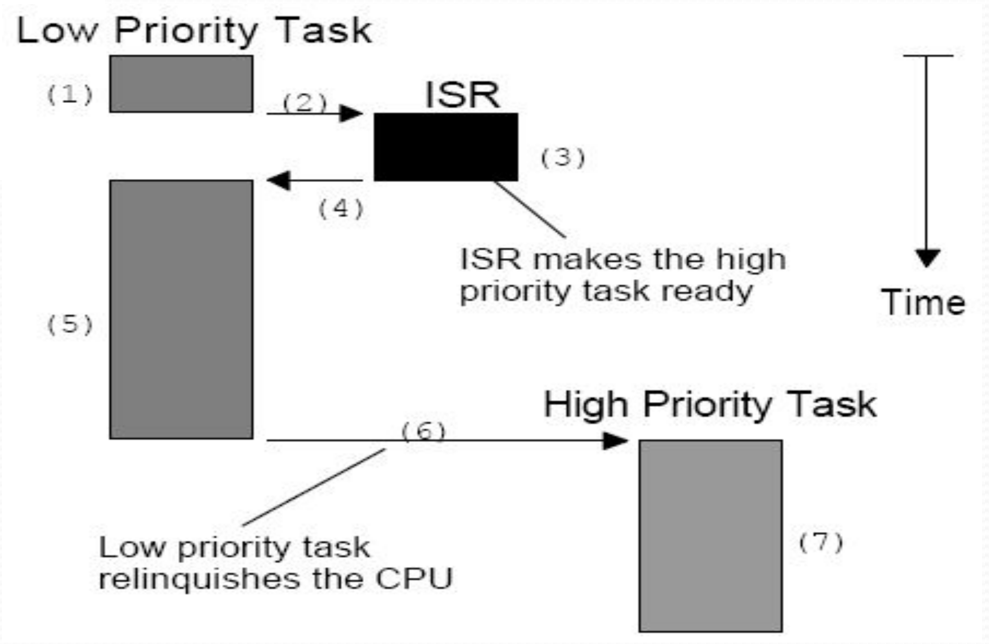

Low Priority Task
(1)
(2)
ISR
(3)
(4)
ISR makes the high priority task ready
Time
(5)
High Priority Task
(6)
Low priority task relinquishes the CPU
(7)

Preemptive kernel

- Always execute first the high priority task ready to run
- ISR returns to high priority task
- Advantage:
  – Task responsiveness is faster and deterministic
  – uCOS-II and most commercial kernels are preemptive
- Disadvantage:
  – Corruption of data may happen
  – Could not call non-preemptive function unless exclusive access provided using semaphore/mutex

Preemptive kernel




Low Priority Task
(1)
(2)
ISR
(3)
(4)
ISR makes the high priority task ready
Time
(5)
High Priority Task
(6)
Low priority task relinquishes the CPU
(7)

Non Preemptive kernel

Advantages and disadvantages of real time Kernel

- A real time operating system, allows real time application to be easily designed and added, functions can be easily added without requiring major changes in software.

- The application can be split into separate tasks. With a preemptive task all the task are handled effectively and efficiently.

- RTOS makes better use of resources by providing facilities such as semaphores, mailboxes, queues, time delays and timeouts.

Advantages and disadvantages of real time Kernel

- Only your application should afford extra requirements, extra ROM/ RAM, extra overhead of CPU.
- The one factor is the cost associated with the use of a real-time kernel. In some applications, cost is everything and would preclude you from even considering an RTOS.
- Some of these packages include also an input/output manager, windowing systems (display), a file system, networking, language interface libraries, debuggers, and cross-platform compilers.
- RTOS vendors may require extra royalties as per target system basis. It may also consider maintenance cost.

Interrupt Latency

- Probably the most important specification of a real time system is ‘Interrupt Latency’. All real time systems disable interrupt to enter in critical section of code and re-enable when the critical section has been executed.
- The interrupts should be disabled for shortest amount of time. The longer time it is disabled, the higher is interrupt latency.
- It is defined as

Interrupt Latency = Max. time interrupt is disabled +
time to execute first line of ISR.

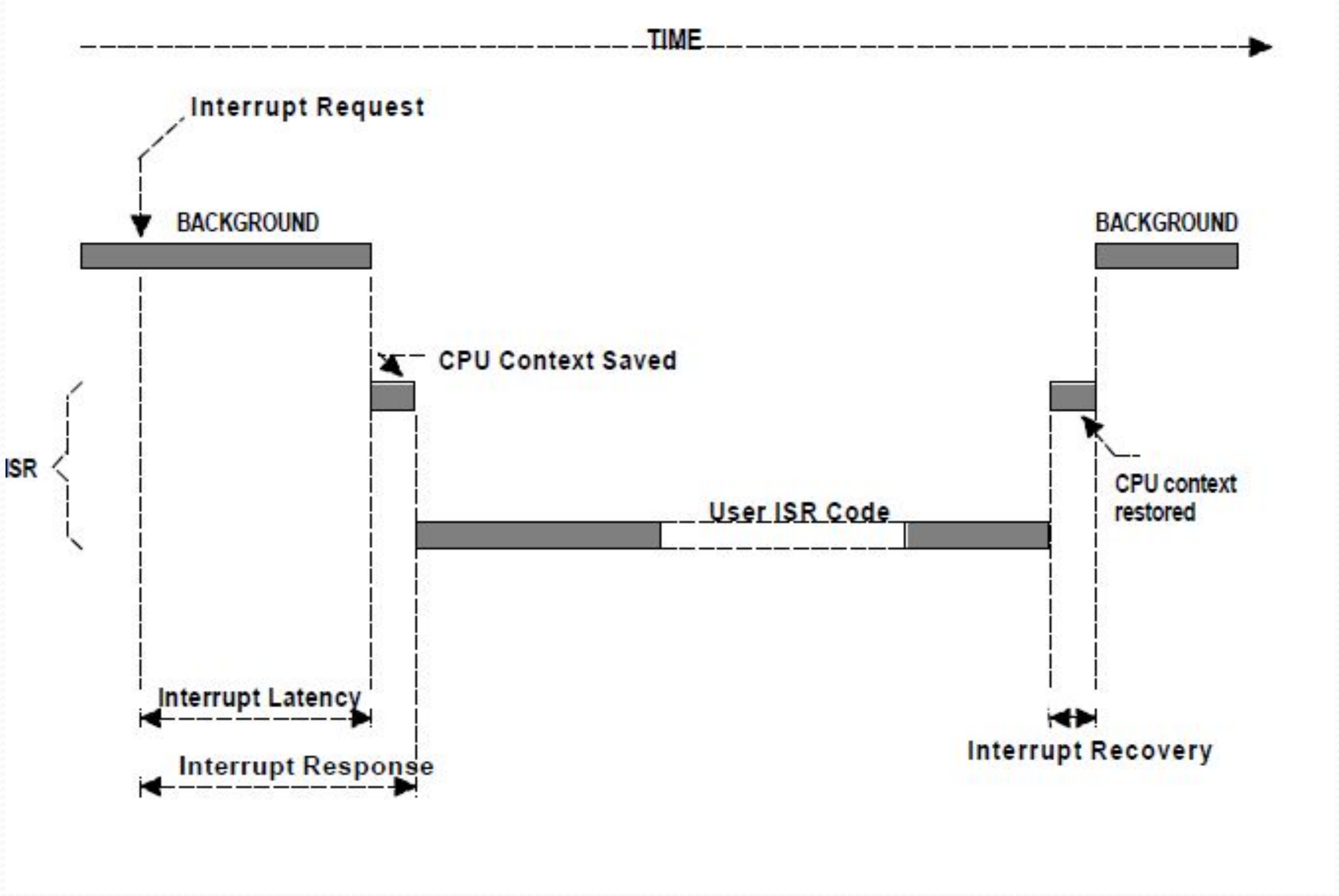
Interrupt Recovery, Response

- Interrupt response is defined as the time between the reception of the interrupt and the start of the user code which will handle the interrupt.
- The response time is given by:

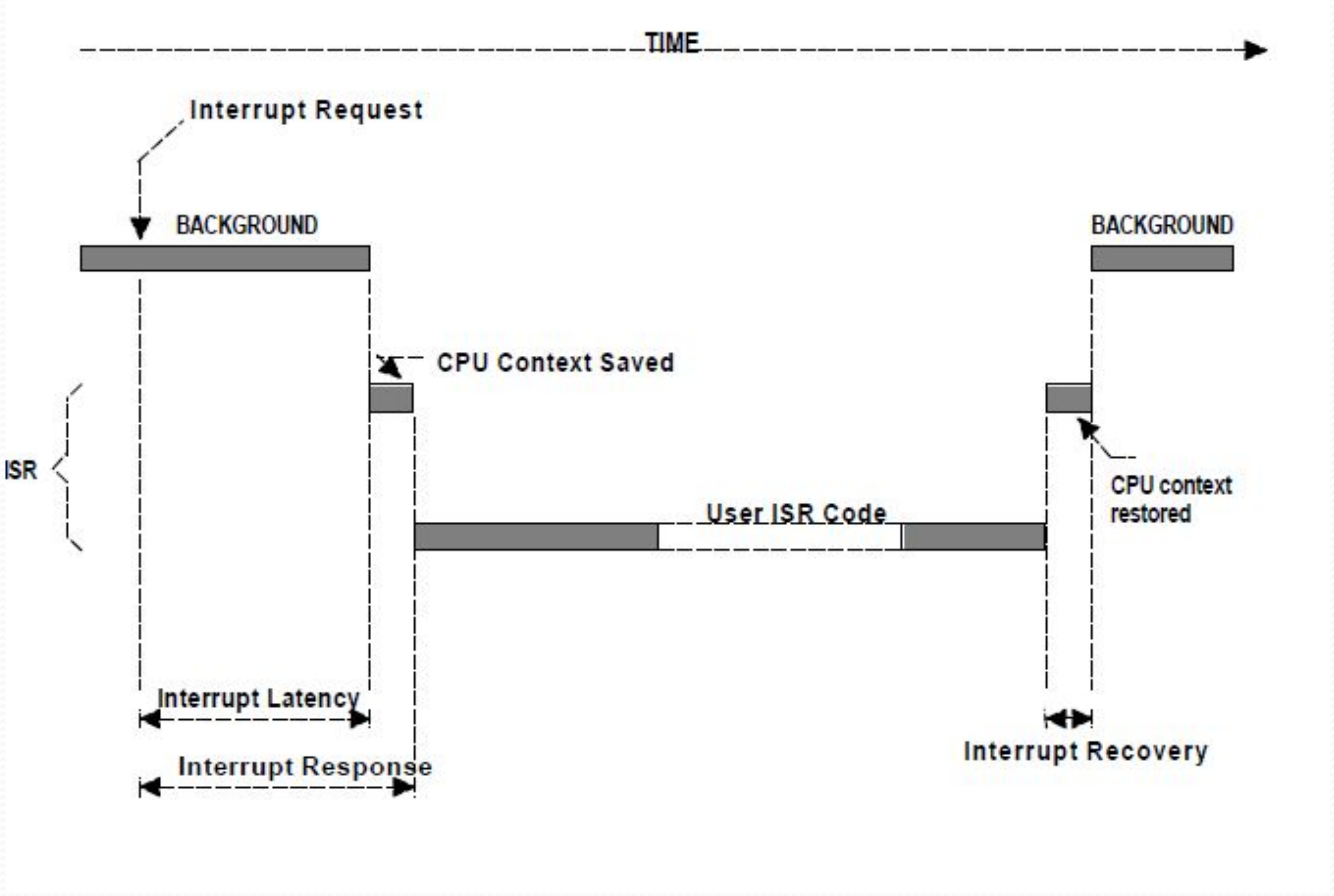
  = Interrupt latency + Time to save the CPU's context.
- Interrupt Recovery: Interrupt recovery is defined as the time required for the processor to return to the interrupted code.
- Interrupt recovery is given by:

  = Time to restore the CPU's context + Time to execute the return from interrupt instruction

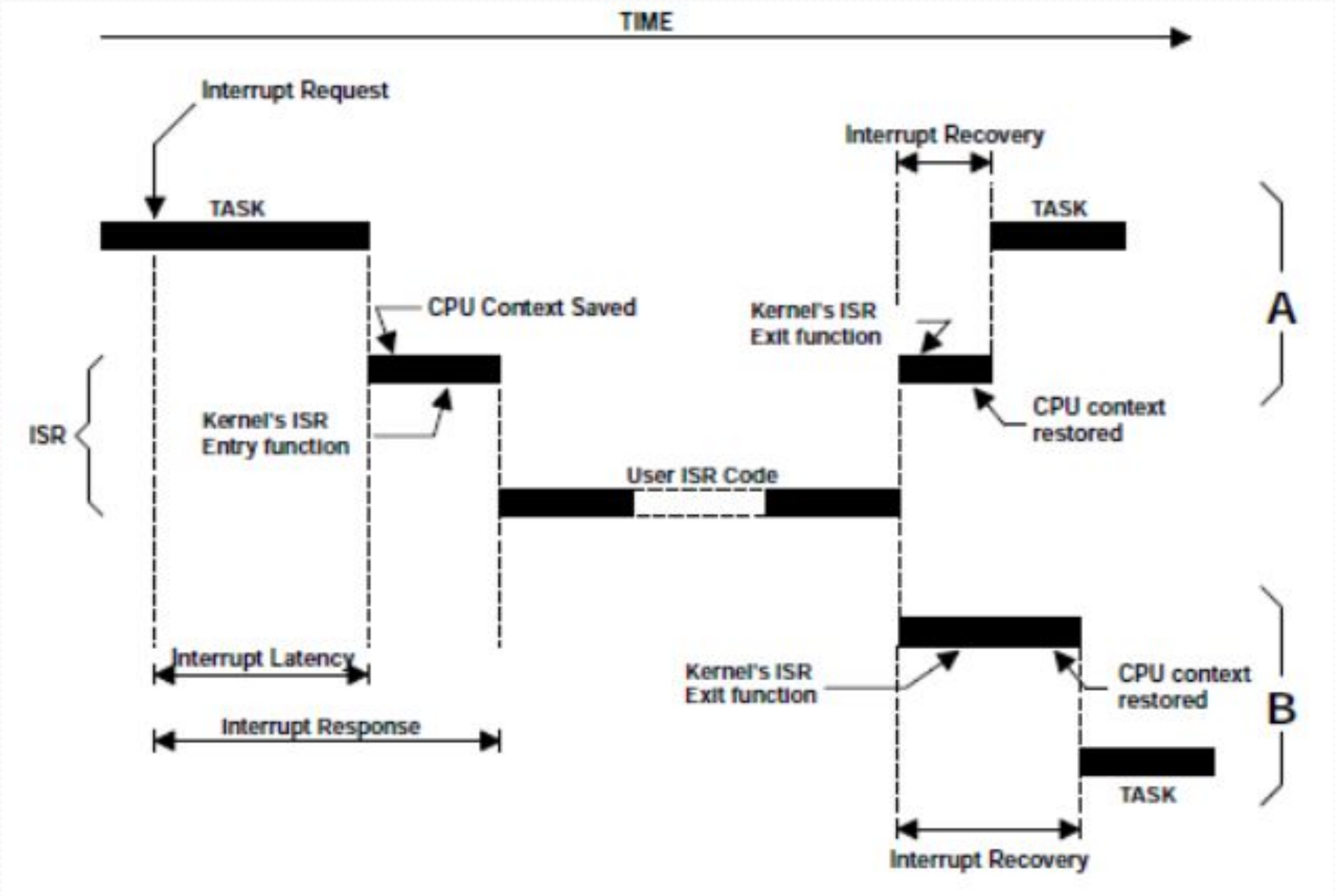
Interrupt Latency, Response and Recovery – Foreground/Background System



Interrupt Latency, Response and Recovery – Non Preemptive Kernel



Interrupt Latency, Response and Recovery – Preemptive Kernel



Task Scheduler

- A part of the RTOS called the **scheduler keeps track of the** state of each task and decides which one task should go into the running state.

- The schedulers in most RTOSs are about which task should get the processor: they look at priorities you assign to the tasks and among the tasks which are not in blocked state, the one with the highest priority runs, and the rest of them wait.

- The lower-priority tasks just have to wait; the scheduler assumes that you knew what you were doing when you set the task priorities.

Round Robin Scheduling

- When two or more tasks are at same priority, the kernel will allow the one task to run for a predetermined time, called ‘Quantum’ and then selects other tasks. This is also called time slicing.
- The kernel gives the control to another task, only if ;

  1) The current task doesn’t have any work to do during the time slice.

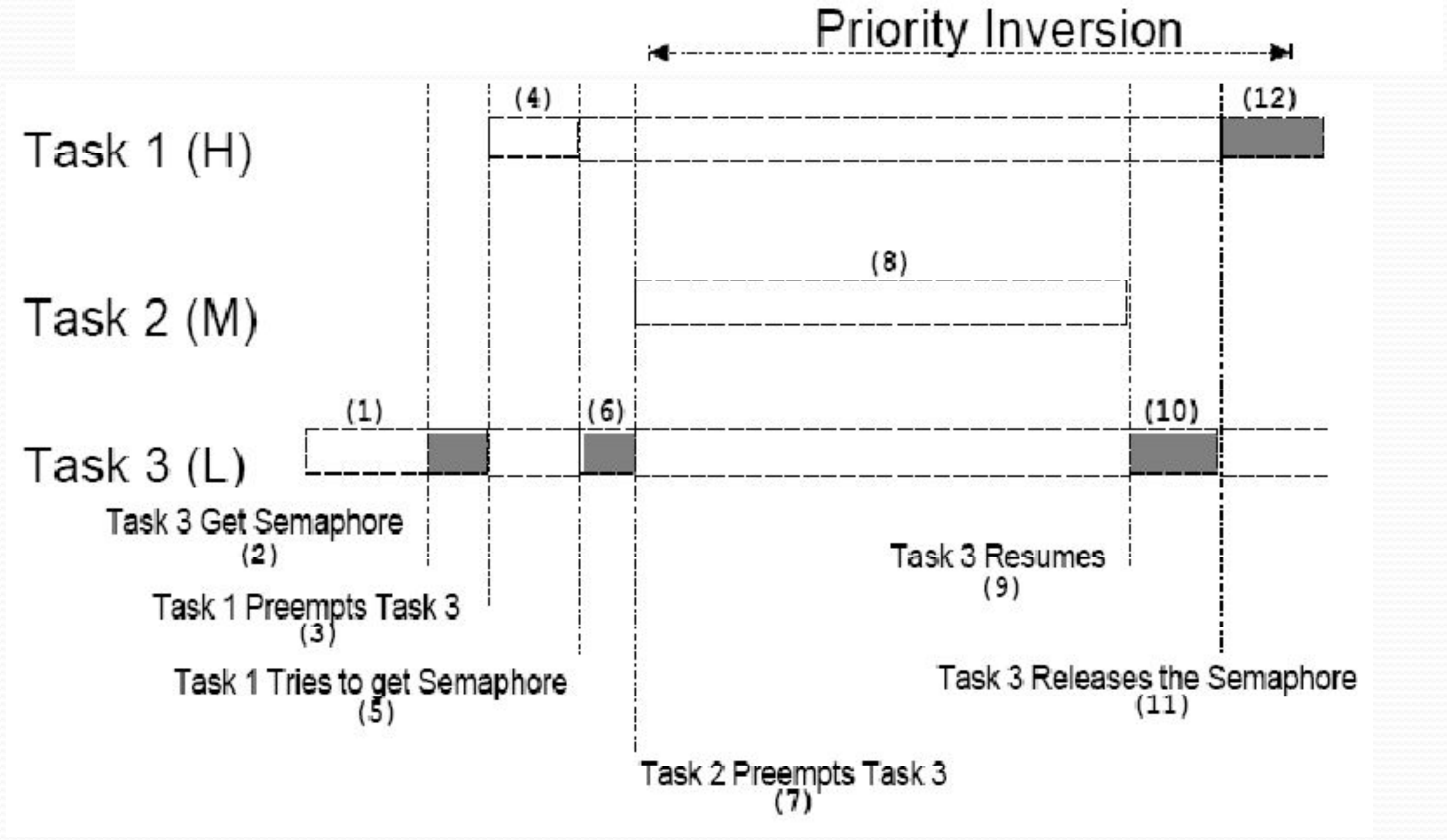
  2) Or, the current task completes before the current time slice.

  3) The time slice ends.
- Generally, uCOSII does not support Round robin scheduling, because no two tasks have the same priority.

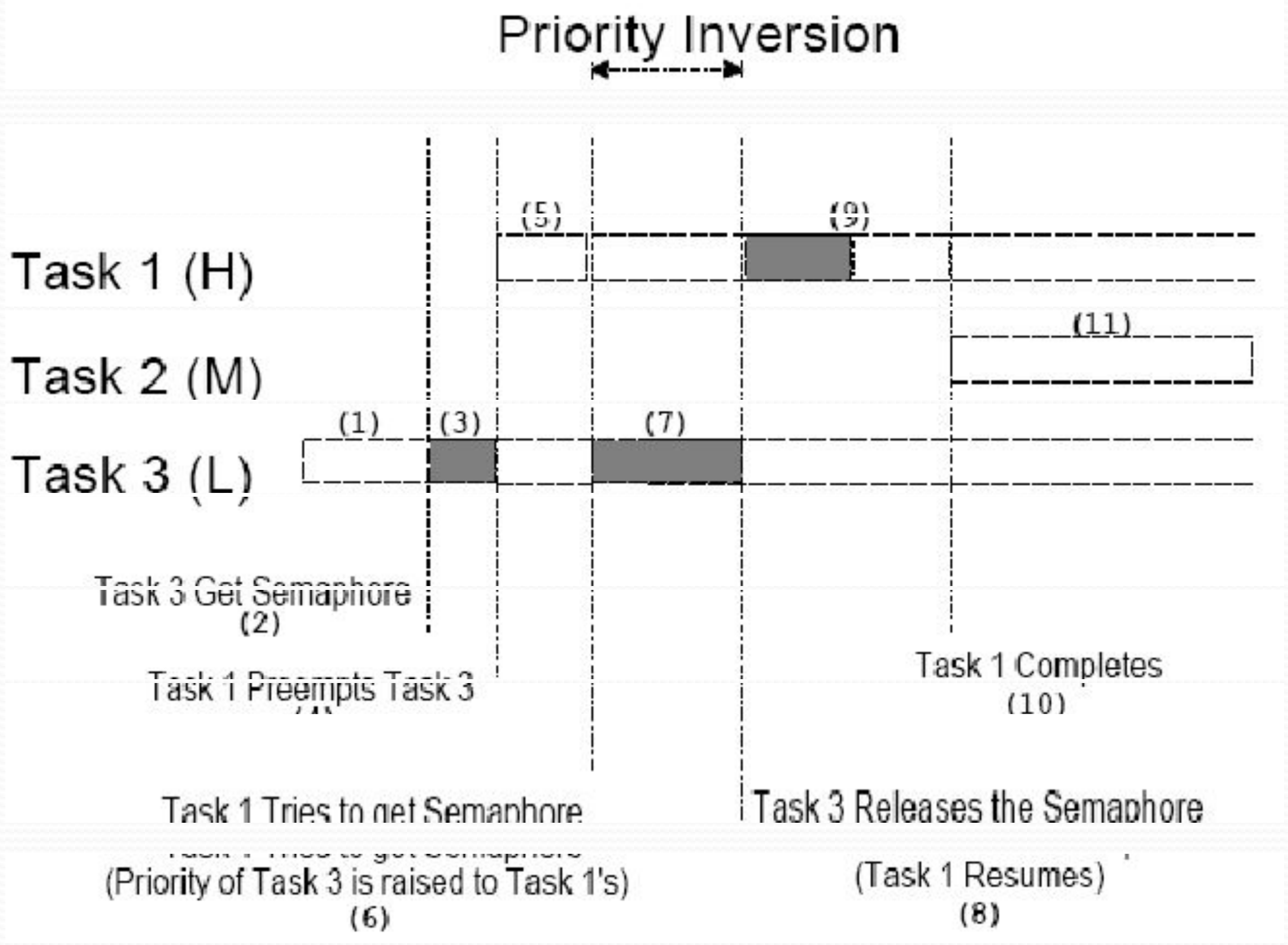
Task Priorities

- Each task is assigned a unique priority, with configurable lowest priority.
- The lower the value of the priority, the higher the priority of the task. The task priority number also serves as the task identifier.
- Static priority: priority can not be changed during execution. All the tasks and their timing constraints are known at compile time.
- Dynamic priority: priority can be changed during execution, required feature to avoid **priority inversion problem.**

Priority Inversion : A high priority task waits for resource acquired by low priority task



Solution- Priority Inversion



Assigning Task Priorities

- Assigning a task priority, is not a trivial undertaking because of the complex nature of Real time operating system.
- In most of the system, not all tasks are considered as critical tasks. Non critical tasks should always be given lower priorities.
- Most real time systems have combination of Soft and Hard real time systems.

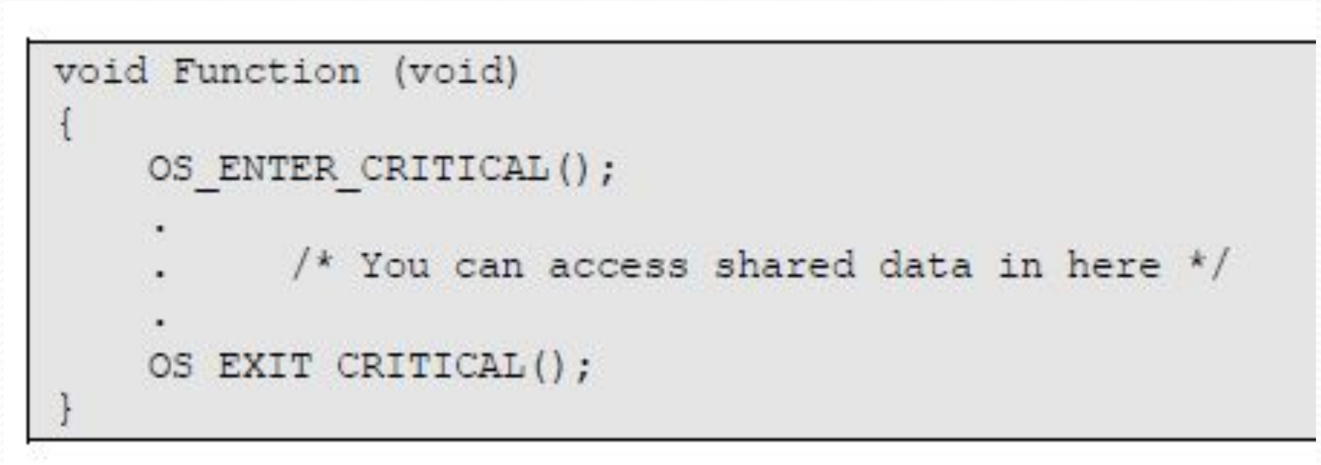
Mutual Exclusion

- The easiest way for tasks to communicate with each other is Data structures. Tasks thus can reference global variables, pointers, buffers, linked lists etc.
- While sharing, we must ensure that each task has exclusive access to the data to avoid contention and data corruption.
- The most common methods are:

1) Disabling Interrupts
2) Performing Test and Set operations
3) Disabling Scheduling
4) Semaphores

Easy Approach – Disable Interrupts

- Disable Interrupts before accessing critical section and then re-enable afterwards
  - Ensures the process will not be interrupted while accessing the shared memory
  - Can be problematic if the process does not re-enable interrupts.
  - uCOSII allows to use two macros for that

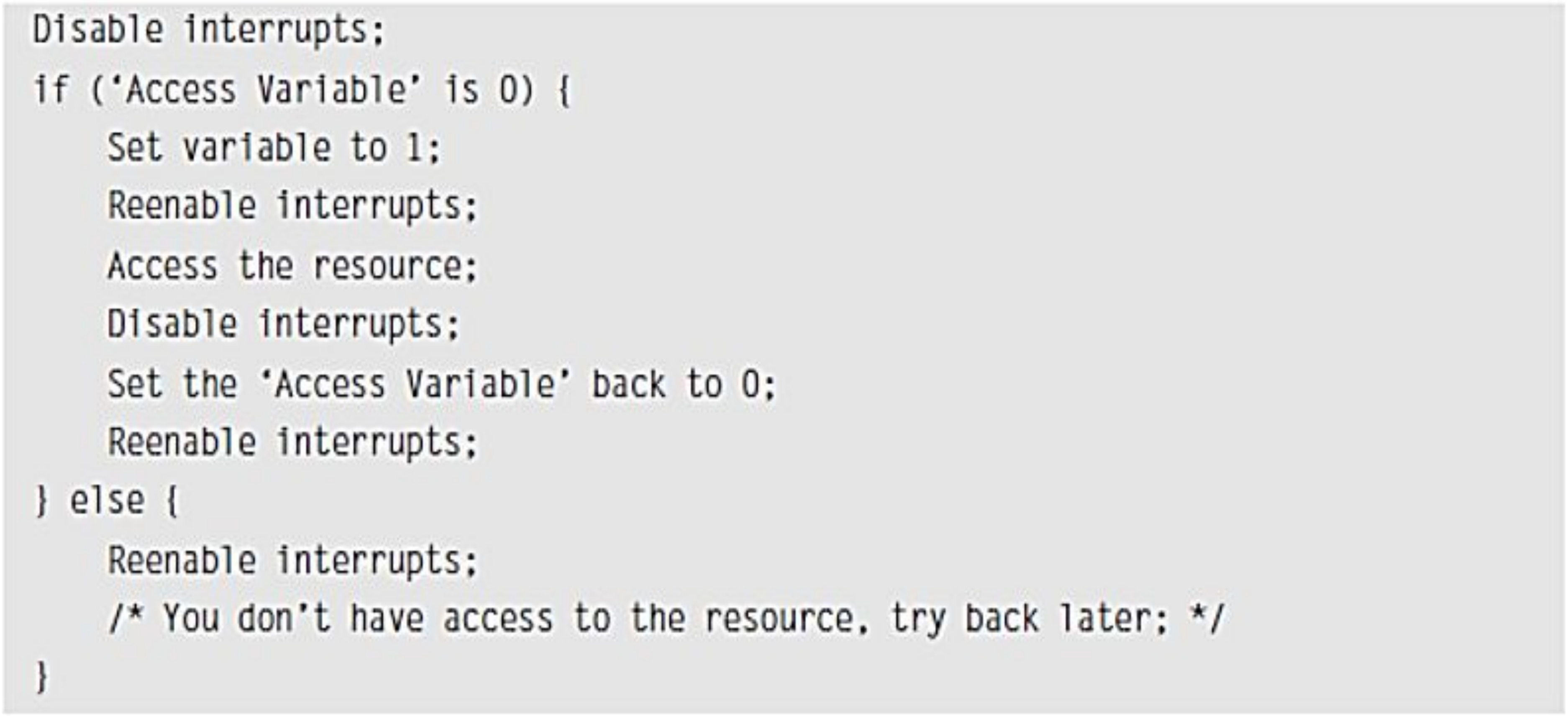


```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .      /* You can access shared data in here */
    .
    OS EXIT CRITICAL();
}
```

Test and Set operations

- In case a kernel is not used, two functions could agree that to access a resource, they must check a global variable and if the variable is ‘0’, the function has access to the resource.
- To prevent the other function from accessing the resource, however the first function who gets access to the resource sets the variable to 1, which is called a test and set operation.

Test and set operations



```
Disable interrupts;
if ('Access Variable' is 0) {
    Set variable to 1;
    Reenable interrupts;
    Access the resource;
    Disable interrupts;
    Set the 'Access Variable' back to 0;
    Reenable interrupts;
} else {
    Reenable interrupts;
    /* You don't have access to the resource, try back later; */
}
```

Disabling and Enabling the Scheduler

- If the task is not sharing variables or data structures with an ISR, Disabling and Enabling the Scheduler can be used.

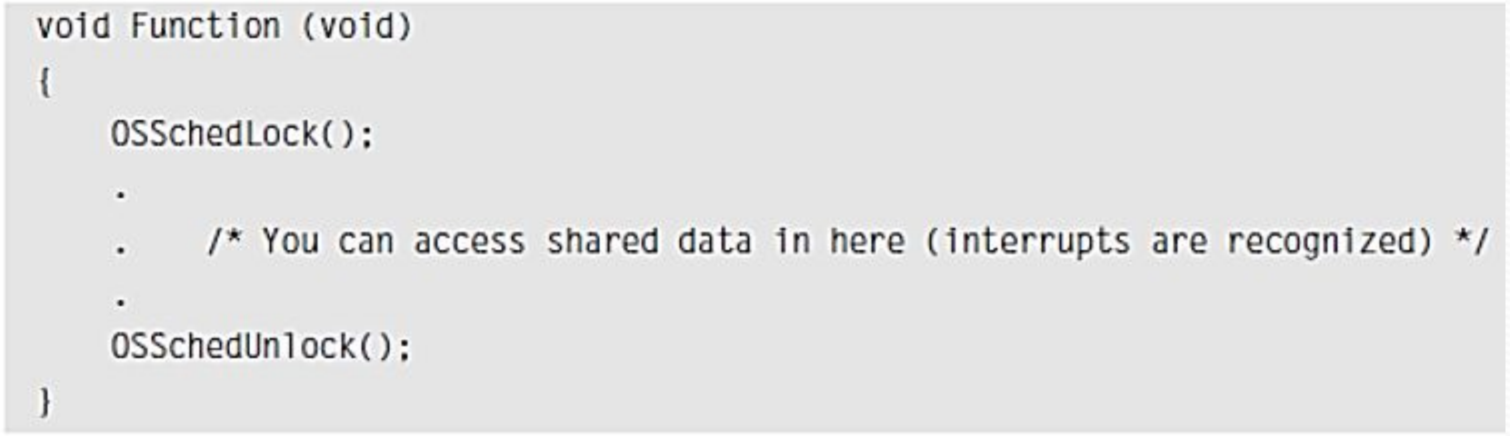
- In this case, two or more tasks can share data without the possibility of contention.

- While the schedule is locked, the interrupts are enabled, and, if an interrupt occurs in the critical section of the code, the ISR is immediately executed.

- At the end of the ISR, the kernel always returns to the interrupted task, even if the ISR has made a high priority task ready to run. Same as in case of Non-Preemptive Kernel

Disabling and Enabling the Scheduler

- The scheduler is invoked when OSSchedUnlock() is called to see if a higher priority task has been ready to run by the task or an ISR. A context switch results if a higher priority task is ready to run.



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

- However, this method should be avoided as it defeats the purpose of having a kernel

Semaphore

- Is a **key to access shared resource**
- Operation performed on semaphore
  – Wait for semaphore
  – Release semaphore
- It will also solve problem of deadlock, where two task are unknowingly waiting for resources held by other task.
- Is a 8, 16 or 32 bit number

Semaphores

- The Semaphore was invented by Dijkstra in the mid 1960s. It is a protocol mechanism offered by most multitasking kernel. Semaphores are used to:

  – Control access to shared resource

  – Signal the occurrence of an event

  – Allows two tasks to synchronize their activities

- Operation performed on semaphore

  – Initialize (CREATE): OSSemCreate()

  – Wait (PEND): OSSemPend()

  - Signal (Post): OSSemPost()

- A Semaphore is a key that the code acquires to continue its execution.
- If the semaphore is in use, the requesting task is suspended until the semaphore is released by the current task.

Semaphores

- There are two types of Semaphores:

  - Binary Semaphore

  - Counting Semaphore

- Binary Semaphore can take only two values 0 or 1
- Counting Semaphore can take values between 0 and 255, 65536 and 4,294,967,295 depending on whether the semaphore mechanism is implemented using 8, 16 and 32 respectively.
- The actual size depends on the kernel used.
- Operations performed on Semaphore:

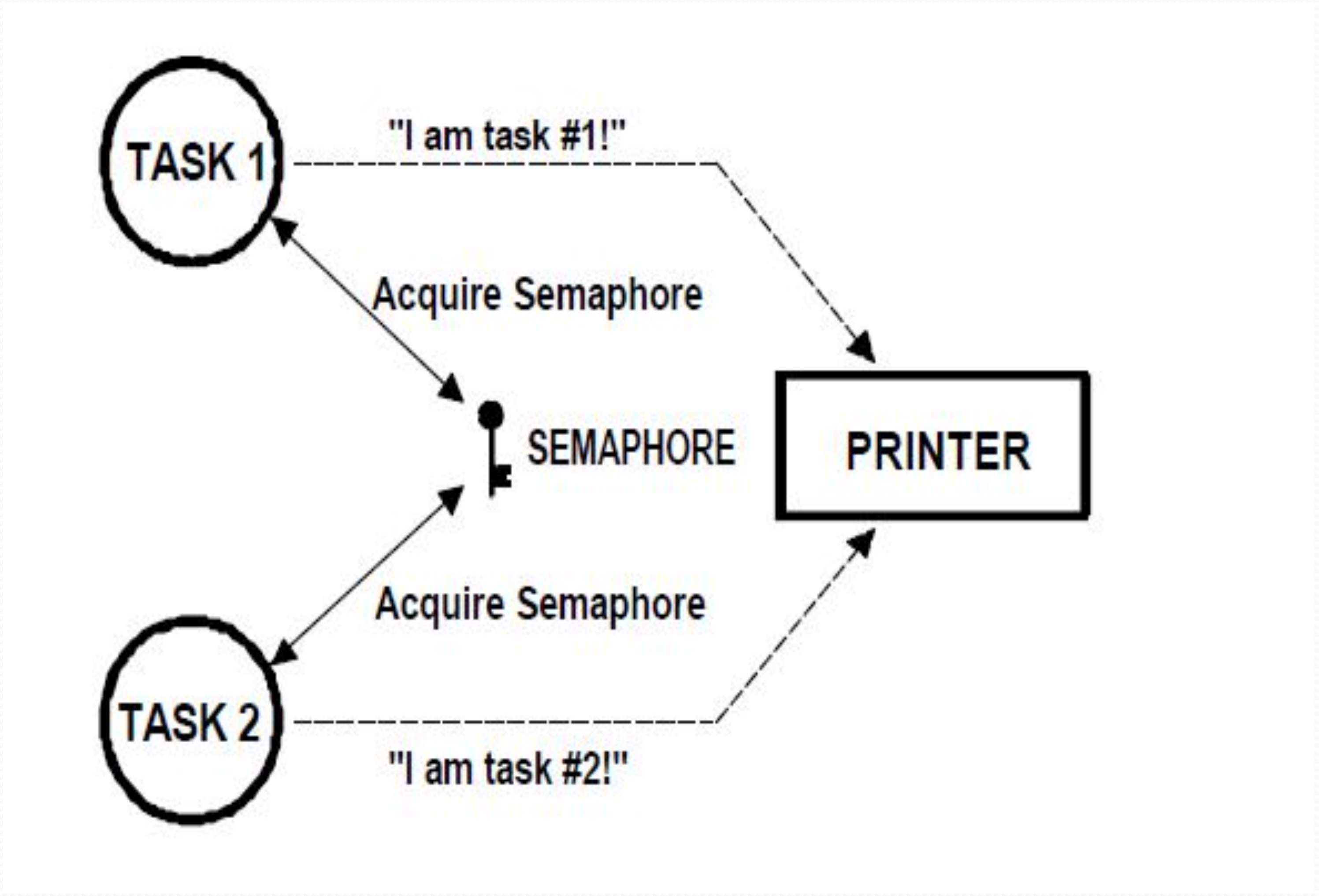
  - INITIALIZE (also called CREATE)

  - WAIT (also called PEND)

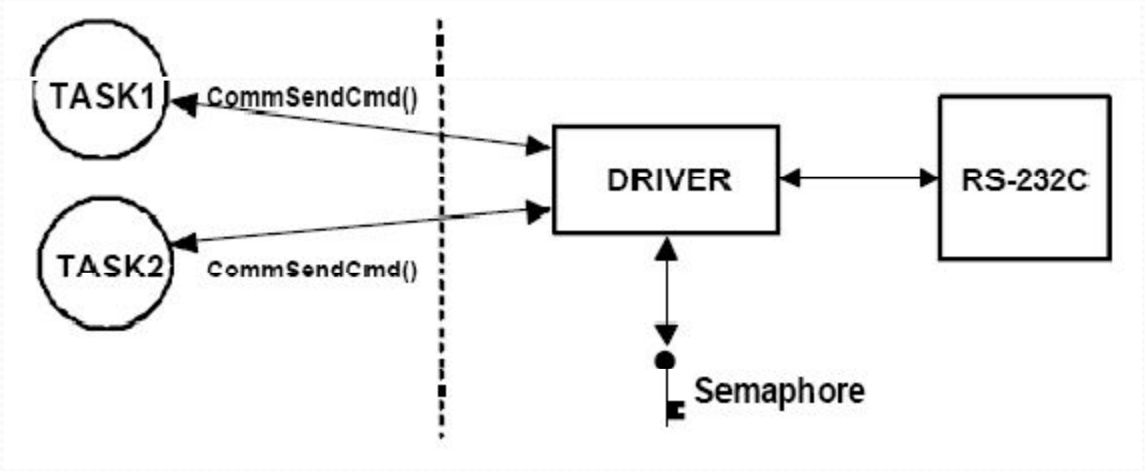
  - SIGNAL (also called POST)

- The initial value of the semaphore should be provided when the semaphore is initialized whereas the waiting list of task is always initially empty.

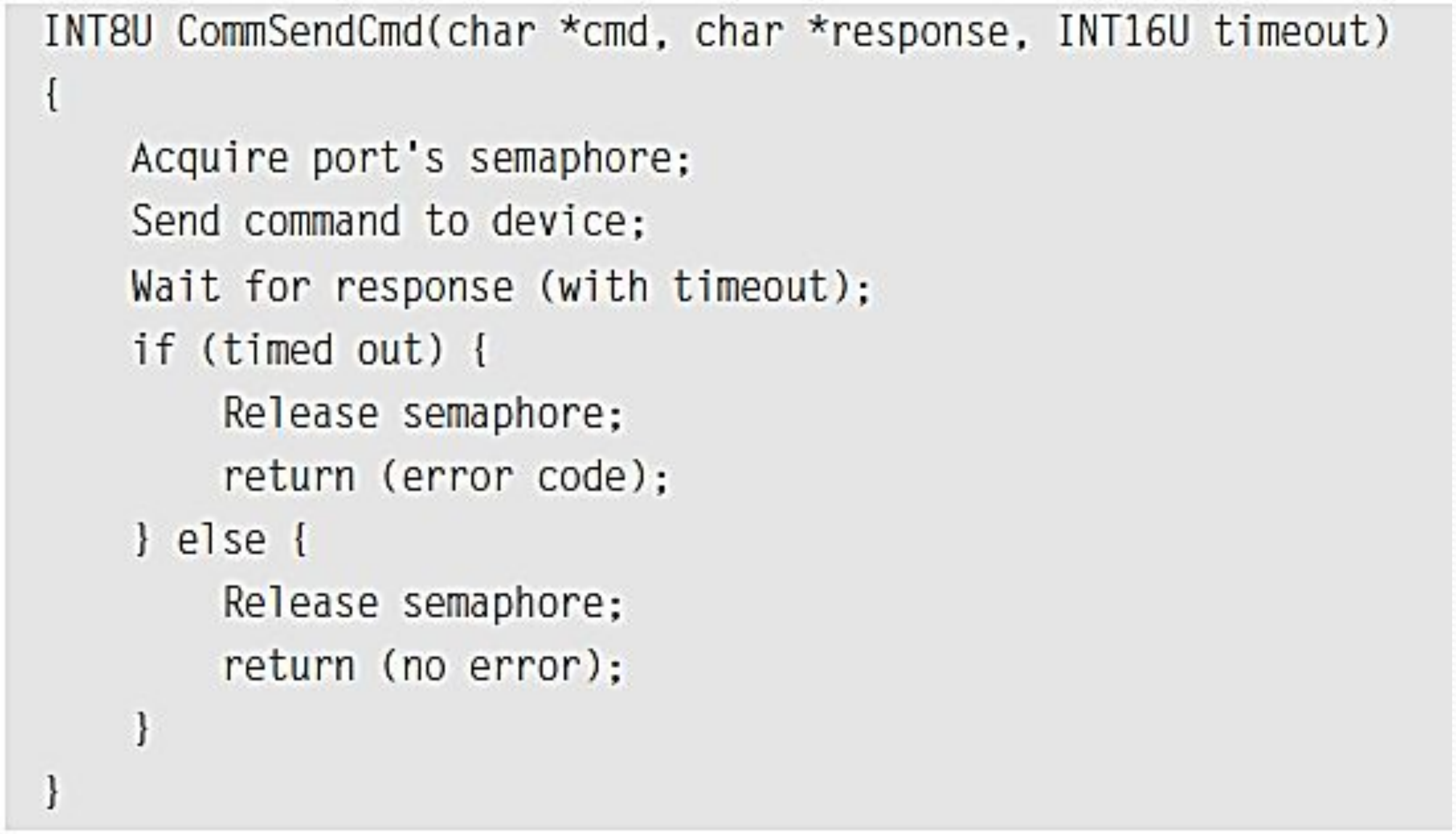
Binary Semaphores



Binary Semaphores

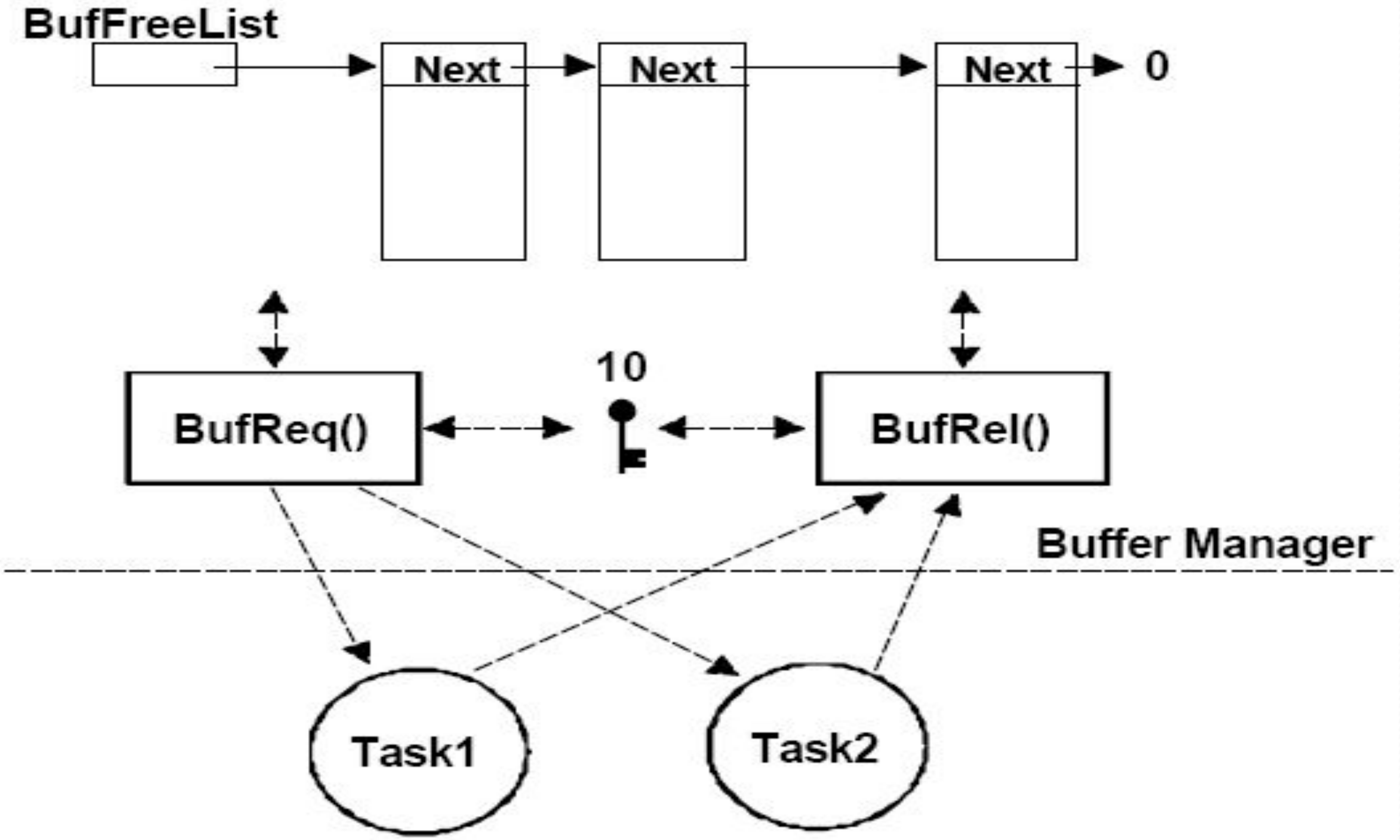


Encapsulating a semaphore



```
INT8U CommSendCmd(char *cmd, char *response, INT16U timeout)
{
    Acquire port's semaphore;
    Send command to device;
    Wait for response (with timeout);
    if (timed out) {
        Release semaphore;
        return (error code);
    } else {
        Release semaphore;
        return (no error);
    }
}
```

Counting Semaphores



Deadlock

- A Deadlock, is a situation in which two tasks are unknowingly waiting for resources held by each other. If T1 has exclusive access of Resource R1 and T2 has exclusive access of Resource R2, then consider T1 wants access of R2 and T2 wants access of R1.
- Solution-

  1) Acquire all resources before proceeding

  2) Acquire the resources in same order.

  3) Release the resources in reverse order

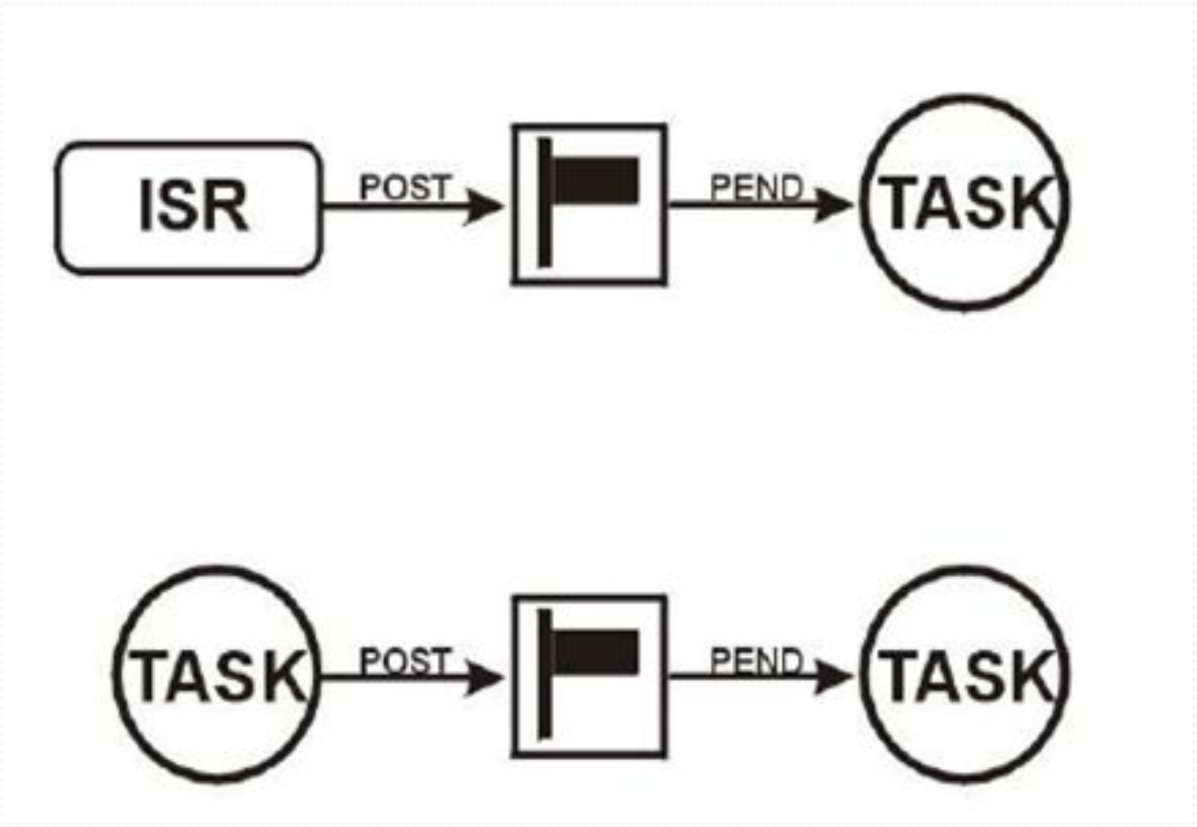
Deadlock

- Most kernels allow you to specify a timeout when acquiring a semaphore. This feature allows a deadlock to be broken.
- If the semaphore is not available within a certain amount of time, the task requesting the resource will resume execution.
- Some form of error code must be returned to the task to notify it that a timeout has occurred.
- A return error code prevents the task from thinking it has obtained the resource.

Synchronizing task with ISR / task

- A task can be synchronized with an ISR (or other task, when no data is to be exchanged) by using a semaphore.
- In this case, semaphore is drawn as flag, and that it is used to signal the occurrence of an event.
- When used as a synchronization mechanism, the semaphore is initialized to 0.
- A task initiates an I/O operation and then waits for the semaphore.
- When an I/O operation is complete, ISR / other task signals the semaphore and the task is resumed.
- This is unilateral operation.

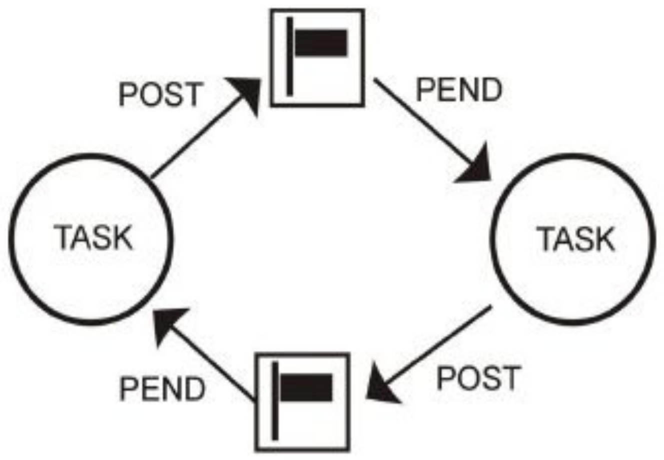
Signaling Events through Semaphores



Synchronizing two tasks

- 2 tasks can synchronize their activities by using 2 semaphores which is called bilateral operation.
- Bilateral operation can not be performed on a task and ISR as ISR can not wait on a semaphore.
- When first task reaches a certain point it signals the second task and waits for a return signal.
- Similarly when second task reaches a certain point it signals the first task and waits for a return signal.
- At this point both tasks are synchronized.

Task Synchronizing – Bilateral Operations



Synchronizing two Tasks

EVENT FLAGS

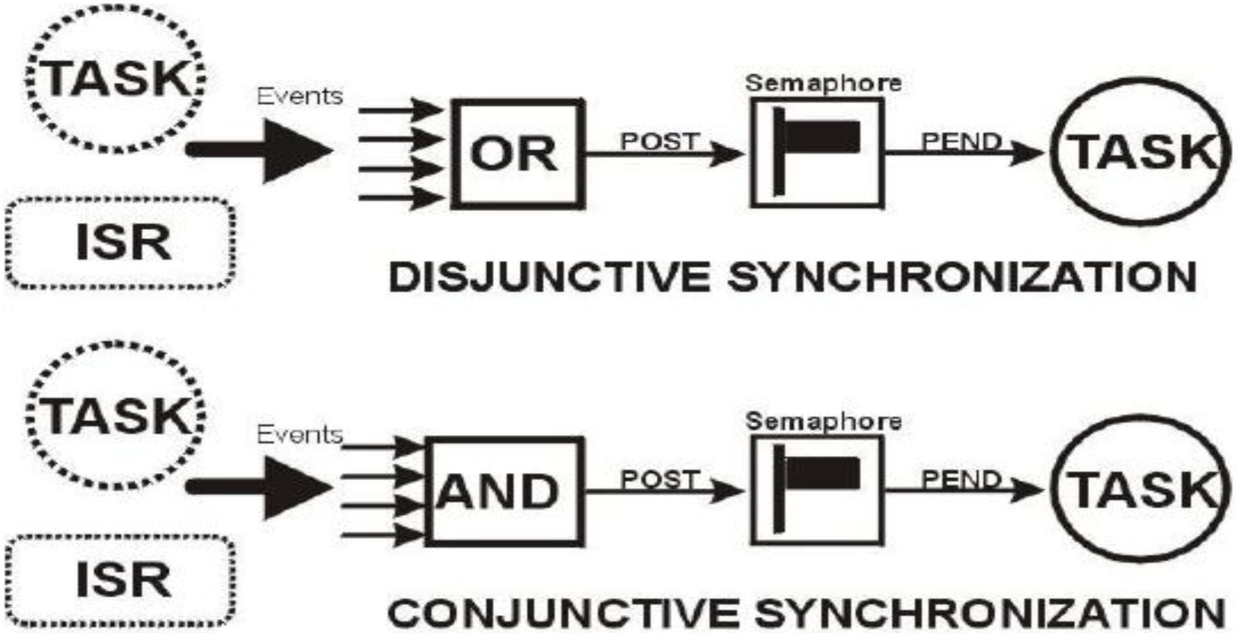
- Event flags are used when a task needs to be synchronized with multiple events.
  - Disjunctive synchronization (OR):

    Task can be synchronized when any of the multiple events have occurred
  - Conjunctive synchronization (AND):

    Task can be synchronized when all events have occurred
- Events are typically grouped (8 / 16/ 32 events per group).
- Tasks / ISRs can set or clear any event in a group.
- Task is resumed when all the events it requires are satisfied.
- MUCOS II offers: SET / CLEAR event flags / WAIT for event flags (OR / AND)

EVENT FLAGS

Semaphores



Disjunctive and Conjunctive Synchronization

Semaphore Problems

- When first reading about semaphores, it is very tempting to conclude that they represent the solutions to all of our shared-data problems. This is not true.
- The problem is that semaphores work only if you use them perfectly, and there are no guarantees that you (or your coworkers) will do that.
- *Forgetting to take the semaphore. Semaphores only* work if every task that accesses the shared data, for read or for write, uses the semaphore.
- *Forgetting to release the semaphore. If any task fails to* release the semaphore, then every other task that ever uses the semaphore will sooner or later block waiting to take that semaphore.

- *Taking the wrong semaphore. If you are* using multiple semaphores, then taking the wrong one is as bad as forgetting to take one.
- *Holding a semaphore for too long*. Whenever one task takes a semaphore, every other task that subsequently wants that semaphore has to wait until the semaphore is released.

THANK YOU!