Unit ~ III

REAL TIME OPERATING SYSTEMS

- Processes and operating systems: Multiple tasks and multiple processes, Multirate systems
- RTOS basics: Architecture of Kernel, Tasks and Task Scheduler, Task States, Content Switching
- Scheduling Algorithms, Rate Monotonic Analysis, Task Management Function Calls
- Interrupt Service Routines, Semaphores, Mutex, Mailboxes, Message queues, Event registers
- Pipes, Signals, Timers, Memory Management, Priority Inversion Problem

Introduction-Multiple Tasks & Multiple Processes

Introduction

The two fundamental abstractions that allow us to build complex applications on microprocessors: the **process** and the **operating system** (OS).

- The process cleanly defines the state of an executing program, while the operating system provides the mechanism for switching execution between the processes. These two mechanisms together let us build applications with more complex functionality and much greater flexibility to satisfy timing requirements.

- Satisfying complex timing tasks can introduce extremely complex control into programs. Using processes to compartmentalize functions and encapsulating in the operating system the control required to switch between processes make it much easier to satisfy timing requirements with relatively clean control within the processes.

- Real-Time Operating Systems (RTOSs), which are operating systems that provide facilities for satisfying real-time requirements. A real-time operating system allocates resources using algorithms that take real time into account. General-purpose operating systems, in contrast, generally allocate resources using other criteria like fairness. Trying to allocate the CPU equally to all processes without regard to time can easily cause processes to miss their deadlines.

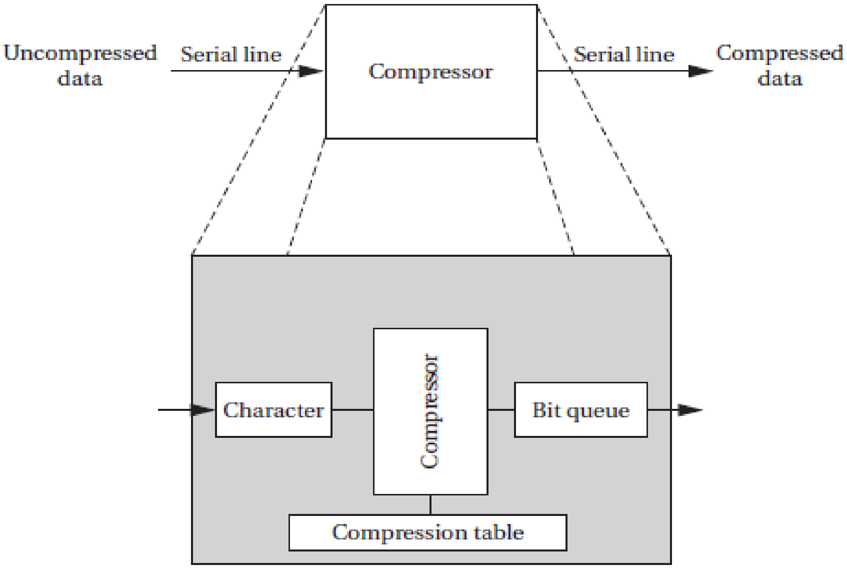
Multiple Tasks and Multiple Processes

The RTOS helps build more complex systems using several programs that run concurrently. Processes and tasks are the building blocks of multitasking systems.

1. Tasks and Processes

- Many embedded computing systems do more than one thing-that is, the environment can cause mode changes that in turn cause the embedded system to behave quite differently.
- A **process** is a single execution of a program. If we run the same program two different times, we have created two different processes. Each process has its own state that includes not only its registers but all of its memory.
- In some operating systems, the memory management unit is used to keep each process in a separate address space. In others, particularly lightweight RTOSs, the processes run in the same address space. Processes that share the same address space are often called **threads.**

This device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line, based on a predefined compression table. Such a box may be used, for example, to compress data being sent to a modem.



Scheduling overhead is paid for at a nonlinear rate.

1. Tasks and Processes...

Variable Data Rates

The need to receive and send data at different rates-for example, the program may emit two bits for the first byte and then seven bits for the second byte. It is possible to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of output bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets.

Asynchronous Input

The text compression box provides a simple example of rate control problems. A control panel on a machine provides an example of a different type of rate control problem, the asynchronous input. The control panel of the compression box may, include a compression mode button that disables or enables compression, so that the input text is passed through unchanged when compression is disabled. Keeping up with the input and output data while checking on the button can introduce some very complex control code into the program.

Sampling the button's state too slowly can cause the machine to miss a button depression entirely, but sampling it too frequently can cause the machine to incorrectly compress data. One solution is to introduce a counter into the main compression loop, so that a subroutine to check the input button is called once every n times the compression loop is executed. But this solution doesn't work when either the compression loop or the button-handling routine has highly variable execution times. We need to be able to keep track of these two different tasks separately, applying different timing requirements to each. such complex control is usually quite difficult to verify for either functional or timing properties.

Multiple Tasks & Multiple Processes.....

1. Tasks and Processes...

TASK

- A task is a functional description of a connected set of operations.
- Task can also mean a collection of processes

PROCESS

- A process is a unique execution of a program.
  - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
  - registers;
  - memory.
- The operating system manages processes.

Multiple Tasks and Processes

- Multiple tasks means multiple processes.
- Processes help with timing complexity:
  - multiple rates
    - multimedia
    - automotive

Multirate Systems

Multirate embedded computing systems are very common, including automobile engines, printers, and cell phones. In all these systems, certain operations must be executed periodically, and each operation is executed at its own rate.

1. Timing requirements on processes

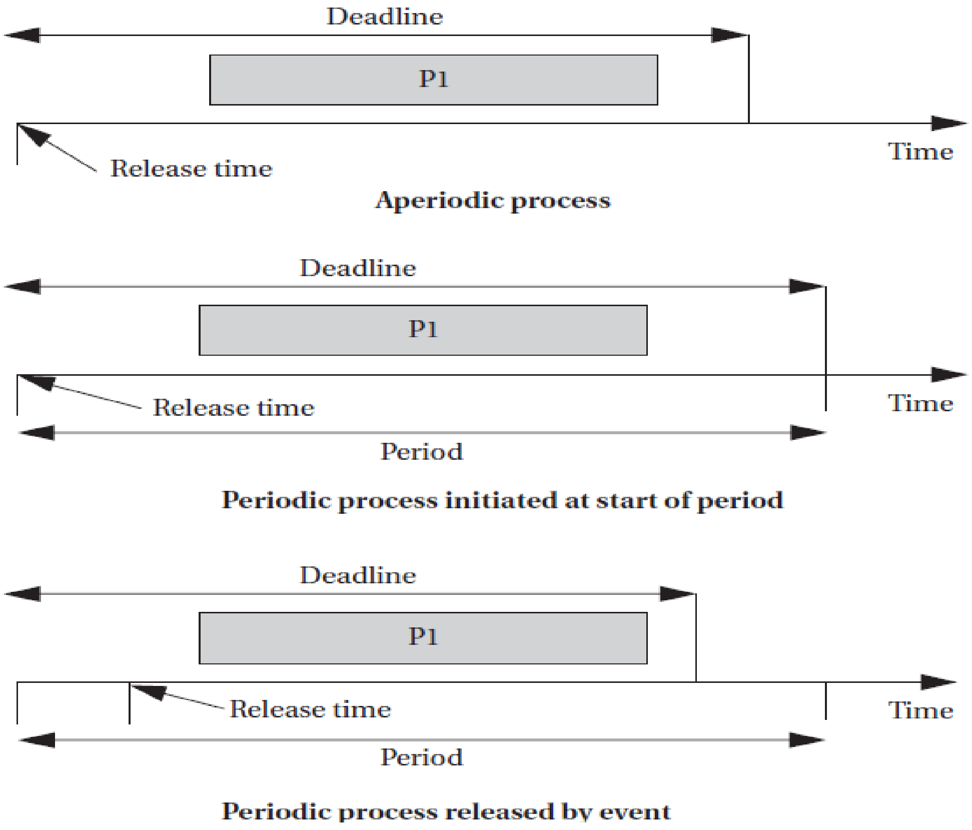
Processes can have several different types of timing requirements imposed on them by the application. The timing requirements on a set of processes strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid.

Two important requirements on processes: initiation time and deadline.

- ➢ The initiation time is the time at which the process goes from the waiting to the ready state.
  - o An aperiodic process is by definition initiated by an event, such as external data arriving or data computed by another process. The initiation time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself.
  - o For a periodically executed process, there are two common possibilities. In simpler systems, the process may become ready at the beginning of the period. More sophisticated systems may set the initiation time at the arrival time of certain data, at a time after the start of the period.

Multirate Systems…..

- A deadline specifies when a computation must be finished.
  - The deadline for an aperiodic process is generally measured from the initiation time because that is the only reasonable time reference.
  - The deadline for a periodic process may in general occur at some time other than the end of the period.



Example definitions of initiation times and deadlines.

Initiation time: time at which the process goes from the waiting to the ready state

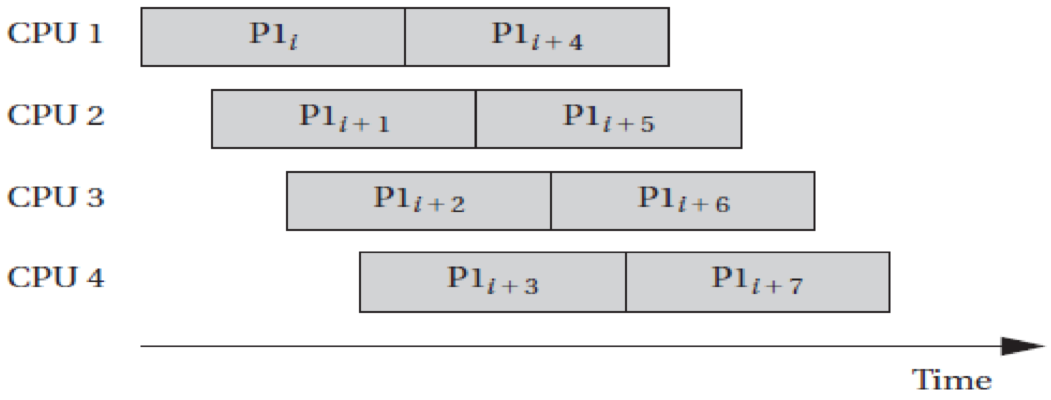
Release time: time at which process becomes ready.

Deadline: time at which process must finish.

Multirate Systems…..

Rate requirements are also fairly common. A rate requirement specifies how quickly processes must be initiated. The period of a process (also called the initiation interval) is the time between successive executions.

In a multirate system, each process executes at its own distinct rate. The most common case for periodic processes is for the initiation interval to be equal to the period. However, pipelined execution of processes allows the initiation interval to be less than the period.



A sequence of processes with a high initiation rate.

Above figure illustrates process execution in a system with four CPUs. The various execution instances of program P1 have been subscripted to distinguish their initiation times. In this case, the initiation interval is equal to one-fourth of the period. It is possible for a process to have an initiation rate less than the period even in single-CPU systems. If the process execution time is significantly less than the period, it may be possible to initiate multiple copies of a program at slightly offset times.

Multirate Systems…..

Jitter

We may also be concerned with the jitter of a task, which is the allowable variation in the completion of the task. Jitter can be important in a variety of applications: in the playback of multimedia data to avoid audio gaps or jerky images; in the control of machines to ensure that the control signal is applied at the right time.

Missing a deadline

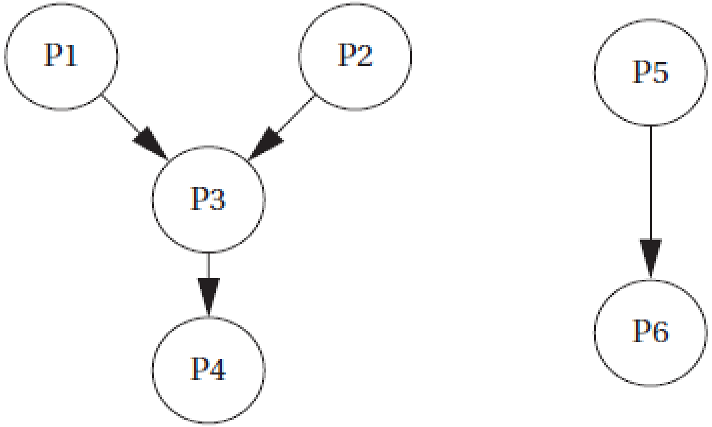
The practical effects of a timing violation depend on the application-the results can be catastrophic in an automotive control system, whereas a missed deadline in a telephone system may cause a temporary silence on the line. The system can be designed to take a variety of actions when a deadline is missed. Safety-critical systems may try to take compensatory measures such as approximating data or switching into a special safety mode. Systems for which safety is not as important may take simple measures to avoid propagating bad data, such as inserting silence in a phone line, or may completely ignore the failure. Even if the modules are functionally correct, their timing behaviour can introduce major execution errors.

Deadline types:

Hard: failure to meet deadline causes system failure.
Soft: failure to meet deadline causes degraded response.
Firm: late response is useless but some late responses can be tolerated.

Multirate Systems…..

The timing constraints between processes may be constrained when the processes pass data among each other.



Data dependencies among processes.

Task: connected set of processes.

Task set: One or more tasks.

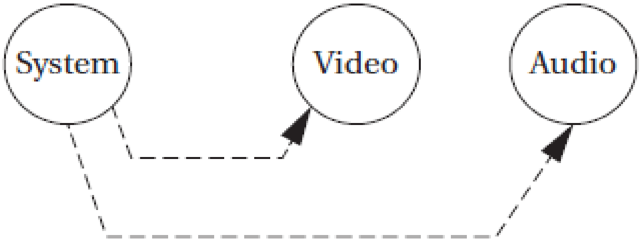
- Above figure shows a set of processes with data dependencies among them. Before a process can become ready, all the processes on which it depends must complete and send their data to it. The data dependencies define a partial ordering on process execution.
- P1 and P2 can execute in any order but must both complete before P3, and P3 must complete before P4. All processes must finish before the end of the period.
- The data dependencies must form a directed acyclic graph (DAG) - a cycle in the data dependencies is difficult to interpret in a periodically executed system.

- A set of processes with data dependencies is known as a **task graph**. A component of the task graph as a task and the complete graph as the task set.

Multirate Systems…..

Communication among processes that run at different rates cannot be represented by data dependencies because there is no one-to-one relationship between data coming out of the source process and going into the destination process. Nevertheless, communication among processes of different rates is very common.

Data come into the decoder in the system format, which multiplexes audio and video data. The system decoder process demultiplexes the audio and video data and distributes it to the appropriate processes.



Communication among processes at different rates.

2. CPU usage metrics

In addition to the application characteristics, need to have a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is utilization:

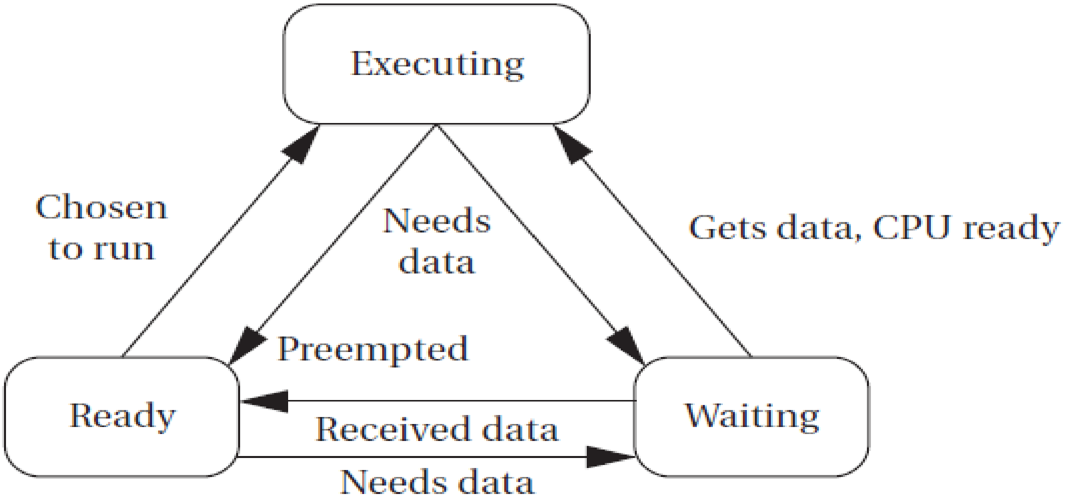
$$U = \frac{\text{CPU time for useful work}}{\text{total available CPU time}}$$

Utilization is the ratio of the CPU time that is being used for useful computations to the total available CPU time. This ratio ranges between 0 and 1, with 1 meaning that all of the available CPU time is being used for system purposes.

Multirate Systems…..

3. Process state and scheduling

The operating system considers a process to be in one of three basic scheduling states: **waiting, ready,** or **executing**. There is at most one process executing on the CPU at any time. Any process that could execute is in the ready state; the operating system chooses among the ready processes to select the next executing process.



Scheduling states of a process.

- ➢ A process goes into the waiting state when it needs data that it has not yet received or when it has finished all its work for the current period.
- ➢ A process goes into the ready state when it receives its required data and when it enters a new period.
- ➢ A process can go into the executing state only when it has all its data, is ready to run, and the scheduler selects the process as the next process to run.

Multirate Systems…..

A scheduling policy defines how processes are selected for promotion from the ready state to the running state. Every multitasking operating system implements some type of scheduling policy.

Scheduling policies vary widely in the generality of the timing requirements they can handle and the efficiency with which they use the CPU. Utilization is one of the key metrics in evaluating a scheduling policy.

In addition to utilization, we must also consider scheduling overhead-the execution time required to choose the next execution process, which is incurred in addition to any context switching overhead. In general, the more sophisticated the scheduling policy, the more CPU time it takes during system operation to implement it.

4. Running periodic processes

To find a programming technique that allows us to run periodic processes, ideally at different rates. For the moment, let's think of a process as a subroutine; we will call them p1( ), p2( ), etc. for simplicity…..

Multirate Systems…..

First step: while loop

Here is a very simple program that runs our process subroutines repeatedly:

```
while (TRUE) {
p1();
p2();
}
```

This program has several problems. First, it doesn’t control the rate at which the processes execute- the loop runs as quickly as possible, starting a new iteration as soon as the previous iteration has finished. Second, all the processes run at the same rate.

A timed loop

A timer is a much more reliable way to control execution of the loop.

The pall() function is called by the timer’s interrupt handler. Then this code will execute each process once after a timer interrupt:

```
void pall() {
p1();
p2();
}
```

But what happens when a process runs too long? The timer’s interrupt will cause the CPU’s interrupt system to mask its interrupts (at least on a reasonable processor), so the interrupt won’t occur until after the pall() routine returns.

Multirate Systems…..

Multiple timers

Our next problem is to execute different processes at different rates. If we have several timers, we can set each timer to a different rate. We could then use a function to collect all the processes that run at that rate:

```
voidpA() {
/* processes that run at rate A */
p1();
p3();
}


voidpB() {
/* processes that run at rate B */
p2();
p4();
p5();
}
...
```

This works, but it does require multiple timers, and we may not have enough timers to support all the rates required by a system.

Multirate Systems…..

Timer plus Counters

An alternative is to use counters to divide the counter rate. If, for example, process p2() must run at 1/3 the rate of p1(), then we can use this code:

```
staticint p2count = 0; /* use this to remember count across timer interrupts */
void pall() {
p1();
if (p2count >= 2) { /* execute p2() and reset count */
p2();
p2count = 0;
}
else p2count++; /* just update count in this case */
}
```

This solution allows us to execute processes at rates that are simple multiples of each other. However, when the rates aren't related by a simple ratio, the counting process becomes more complex and more likely to contain bugs.

RTOS Basics

Operating system (OS) :

An Operating system (OS) is a piece of software that controls the overall operation of the System. It acts as an interface between hardware and application programs.

It facilitates the user to format disks, create, print, copy, delete and display files, read data from files, write data to files, control the I/O operations, allocate memory locations and process the interrupts etc.

In a multiuser system it allows several users to share the CPU time, share the other system resources and also avoids the interference of different users in sharing the resources etc. Hence the OS is also known as a resource manager.

An Operating system (OS) is nothing but a collection of system calls or functions. An OS typically provides multitasking, synchronization, Interrupt and Event Handling, Input/Output, Inter-task Communication, Timers and Clocks and Memory Management.

Popular Operating Systems:

MS-DOS, Windows (Microsoft), Linux(Open source), Unix (Multi user-Bell Labs), MacOS, Android (Mobile).

RTOS Basics .....

REAL TIME SYSTEMS:

Real-time systems are those systems in which the correctness of the system depends not only on the Output, but also on the time at which the results are produced (Time constraints must be strictly followed).

Real time systems are two types:

(i) Soft real time systems (ii) Hard real time systems.

- Soft Real Time system is one in which the performance of the system is only degraded but, not destroyed if the timing deadlines are not met.
  - Ex: Air conditioner, TV remote or music player, Bus reservation, automated teller machine in a bank, Lift, etc.

- Hard Real Time system is one in which the failure to meet the time dead lines may lead to a complete catastrophe or damage to the system.
  - Ex: Air navigation system, Nuclear power plant, Failure of car brakes, Gas leakage system, RADAR operation, etc.

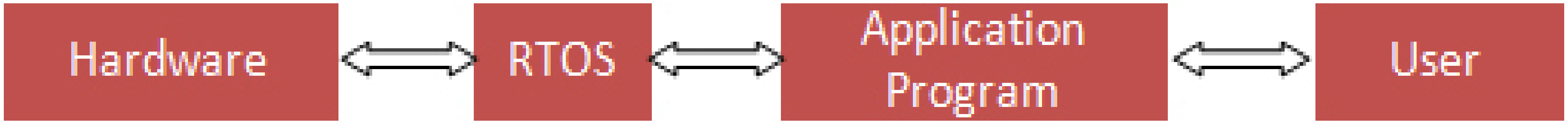
RTOS Basics .....

REAL TIME OPERATING SYSTEM (RTOS):

It is an operating system that supports real-time applications by providing logically correct result within the deadline set by the user. A real time operating system makes the embedded system into a real time embedded system.

A Real-Time Operating System (RTOS) comprises of two components, viz., “Real-Time” and “Operating System”.

Basic Structure is similar to regular OS but, in addition, it provides mechanisms to allow real time scheduling of tasks



Hardware

RTOS

Application Program

User

Fig. Real time embedded system with RTOS

All the embedded systems are not designed with RTOS. Low end application systems do not require the RTOS but only High end application oriented embedded systems which require scheduling alone need the RTOS.

For example an embedded system which measures Temperature or Humidity etc. do not require any operating system.

Where as a Mobile phone, RADAR or Satellite system used for high end applications require an operating system.

RTOS Basics .....

RTOS Classification:

RTOS specifies a known maximum time for each of the operations that it performs. Based upon the degree of tolerance in meeting deadlines, RTOS are classified into following categories

- Hard real-time: Degree of tolerance for missed deadlines is negligible. A missed deadline can result in catastrophic failure of the system
- Firm real-time: Missing a deadly ne might result in an unacceptable quality reduction but may not lead to failure of the complete system
- Soft real-time: Deadlines may be missed occasionally, but system doesn't fail and also, system quality is acceptable

RTOS Features:

- Multithreading and preemptability.
- Thread Priority.
- Inter Task Communication & Synchronization.
- Priority Inheritance.
- Short Latencies.

RTOS Basics .....

Basic functions of RTOS:

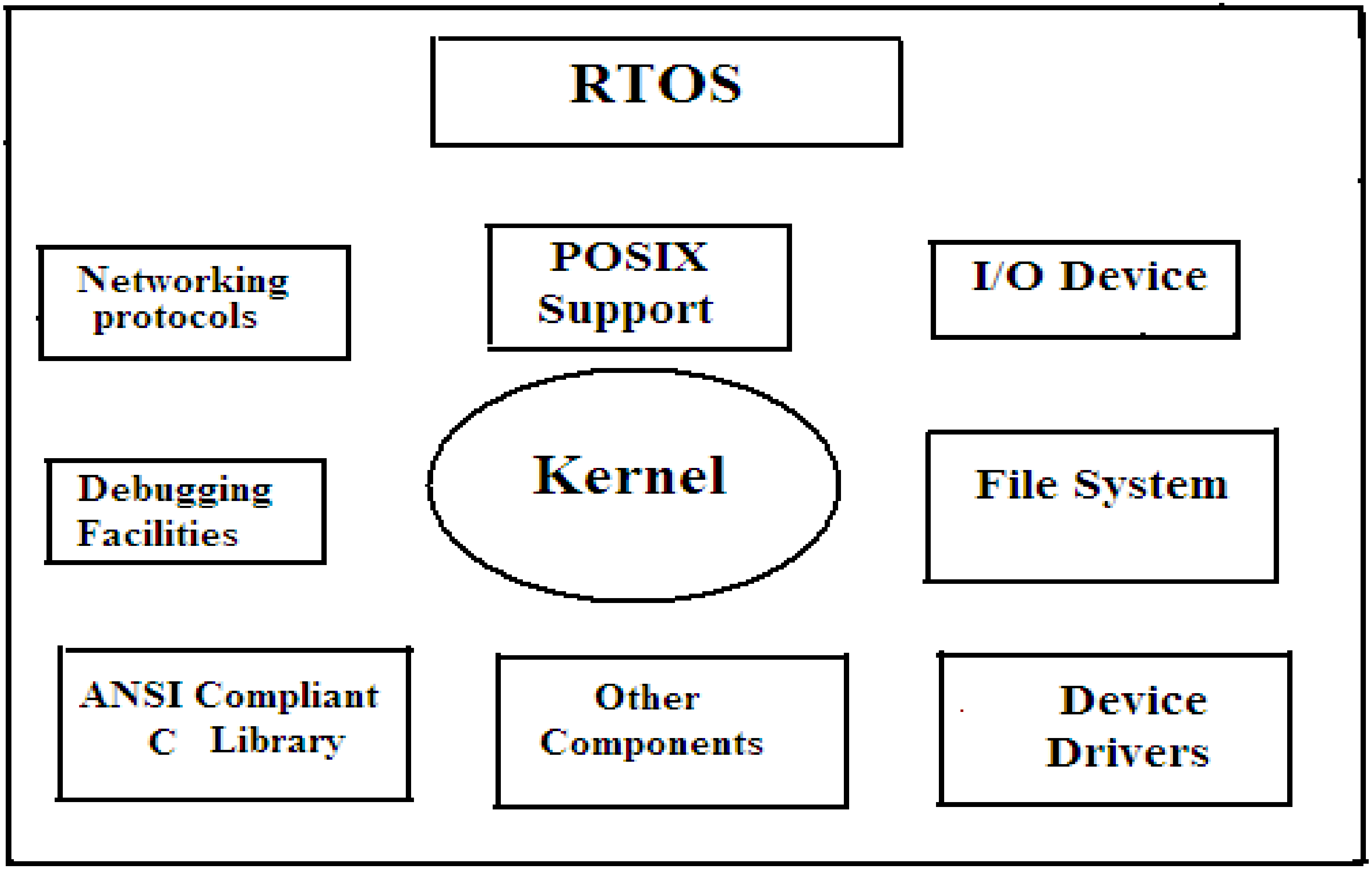
- Time management
  - A high resolution hardware timer is programmed to interrupt the processor at fixed rate (Time interrupt)
  - Each time interrupt is called a system tick (time resolution)
- Task management
  - Task creation: create a newTCB
  - Task termination: remove the TCB
  - Change Priority: modify the TCB
  - State-inquiry: read the TCB
- Interrupt handling
  - ISR
- Memory management
  - No virtual memory for hard RT tasks, Many embedded RTS do not have memory protection
- Exception handling (important)
  - missing deadline, running out of memory, timeouts, deadlocks, divide by zero, etc.
- Task synchronization
  - Semaphore, Mutex, Avoid priority inversion
- Task scheduling
  - Priorities (HPF), Execution times (SCF), Deadlines (EDF), Arrival times (FIFO)

RTOS Basics .....

Requirements on RTOS

- Determinism
  - Deterministic system calls
- Responsiveness (quoted by vendors)
  - Fast process/thread switch
  - Fast interrupt response
- Support for concurrency and real-time
  - Multi-tasking
  - Real-time
  - Synchronization
- User control over OS policies
  - Mainly scheduling, many priority levels
  - Memory support (especially embedded)
    - pages locked in main memory
    - cache partitioning
- Controlled code size

RTOS Basics .....



General Architecture of RTOS

RTOS Basics .....

DIFFERENCES BETWEEN RTOS and GENERAL PURPOSE OS

The basic differences are Determinism, Task scheduling and Preempting

- **Determinism:** The key difference between general-computing operating systems and real-time operating systems is the deterministic timing behavior in the real-time operating systems. "Deterministic" timing means that OS consume only known and expected amounts of time.
- **Task Scheduling:** General purpose operating systems are optimized to run a variety of applications and processes simultaneously, thereby ensuring that all tasks receive at least some processing time. RTOS uses priority-based preemptive scheduling, which allows high-priority threads to meet their deadlines consistently. All system calls are deterministic, implying time bounded operation for all operations and ISRs. The scheduling in RTOS is time based. In case of General purpose OS, scheduling is process based.
- **Pre-Emptive and Non-Pre-Emptive:** In a normal operating system, if a task is running, it will continue to run until its completion. It cannot be stopped by the OS in the middle due to any reason. Such concept is known as non-preemptive. In real time OS, a running task can be stopped due to a high priority task at any time with-out the willing of present running task. This is known as pre-emptiveness.

RTOS Basics .....

REAL TIME OPERATING SYSTEM….. Popular RTOSs:

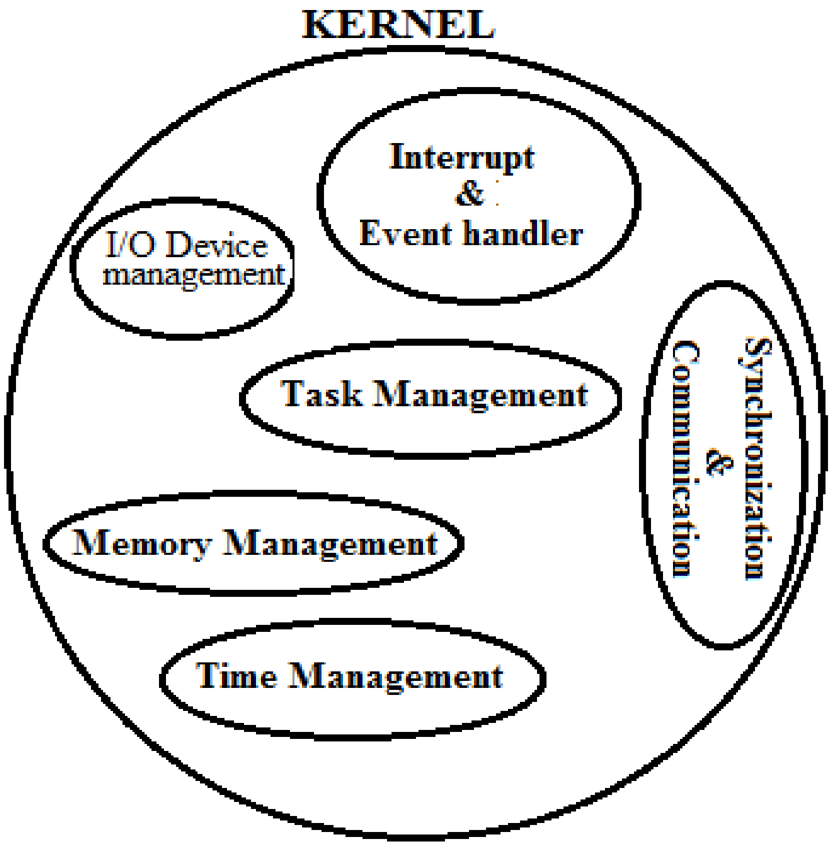
RTOS	Applications/Features
Windows CE	▪ Used small foot print mobile and connected devices ▪ Supported by ARM, MIPS & x86 architectures
LynxOS	o Complex, hard real-time applications o POSIX-compatible, multiprocess, multithreaded OS. o Supported by x86, ARM, PowerPC architectures
VxWorks	▪ Most widely adopted RTOS in the embedded industry. ▪ Used in famous NASA rover robots ▪ Certified by several agencies and for real time systems, reliability and security-critical applications.
µC/OS-II	o Ported to more than a hundred architectures including x86, mainly used in microcontrollers with low resources. o Certified by rigorous standards, such as RTCADO-178B
QNX Neutrino	▪ Most traditional RTOS in the market, Powerful hard RTOS ▪ Microkernel architecture; completely compatible with the POSIX ▪ Certified by FAADO-278 and MIL-STD-1553 standards.
Symbian	o Designed for Smartphones o Supported by ARM, x86 architecture
VRTX	▪ Traditional board based embedded systems and SoC architectures ▪ Supported by ARM, MIPS, PowerPC & other RISC architectures

Architecture of Kernel

The kernel is the core of an operating system. It is a piece of software responsible for providing secure access to the system's hardware and to running the programs.

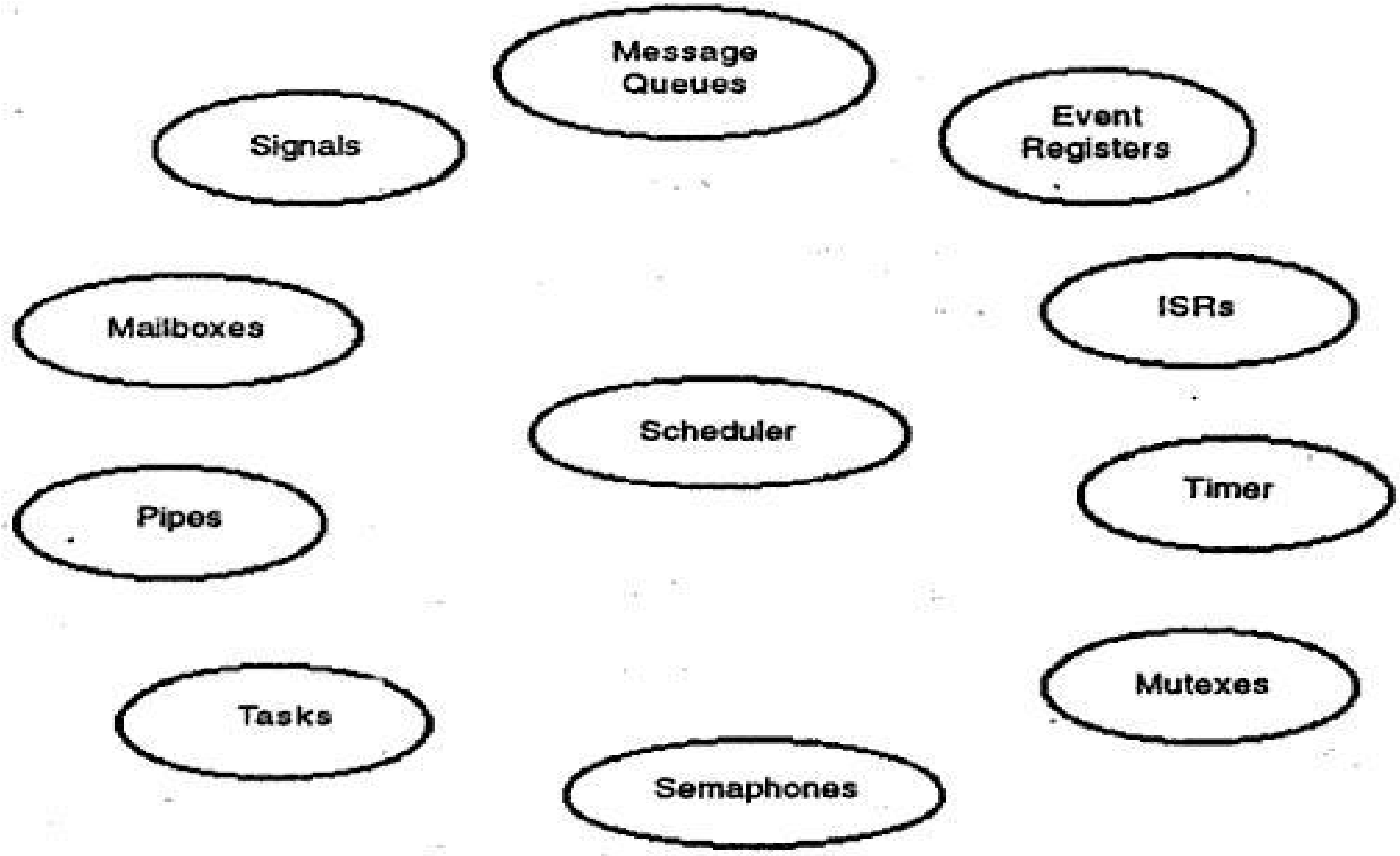
Kernel is common to every operating system either a real time or non-real time. The major difference lies in its architecture.
Since there are many programs, and hardware access is limited, the kernel also decides when and how long a program should run. This is called scheduling.

Kernels has various functions such as file management, data transfer between the file system, hardware management, memory management and also the control of CPU time. The kernel also handles the Interrupts.



Architecture of Kernel..... Kernel Objects

The various kernel objects are Tasks, Task Scheduler, Interrupt Service Routines, Semaphores, Mutexes, Mailboxes, Message Queues, Pipes, Event Registers, Signals and Timers



Kernel Objects

Architecture of Kernel..... Kernel Objects

Real-Time Kernel

The heart of a real-time OS is the kernel. A kernel is the central core of an operating system, and it takes care of all the OS jobs:

- Booting
- Task Scheduling
- Standard Function Libraries

In an embedded system, frequently the kernel will boot the system, initialize the ports and the global data items. Then, it will start the scheduler and instantiate any hardware timers that need to be started.

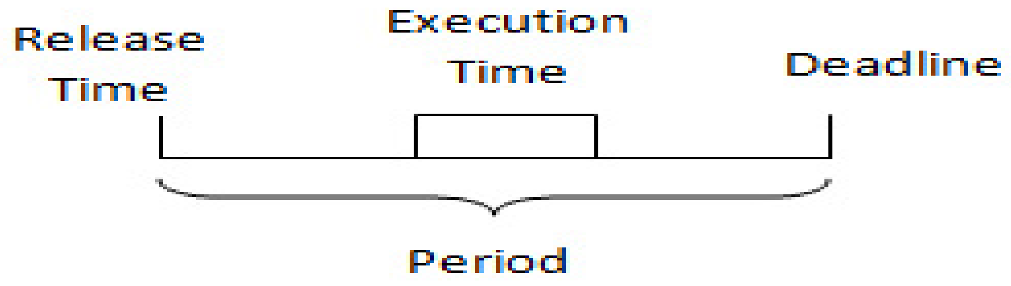
After all that, the Kernel basically gets dumped out of memory (except for the library functions, if any), and the scheduler will start running the child tasks.

Tasks and Task Scheduler
Task States, Content Switching

Task

A task is a basic unit or atomic unit of execution that can be scheduled by an RTOS to use the system resources like CPU, Memory, I/O devices etc. It starts with reading of the input data and of the internal state of the task, and terminates with the production of the results and updating the internal state. The control signal that initiates the execution of a task is provided by the operating system.

In RTOS, The application is decomposed into small, schedulable, and sequential program units known as “Task”, a basic unit of execution and is governed by three time-critical properties; release time, deadline and execution time. Release time refers to the point in time from which the task can be executed. Deadline is the point in time by which the task must complete. Execution time denotes the time the task takes to execute.



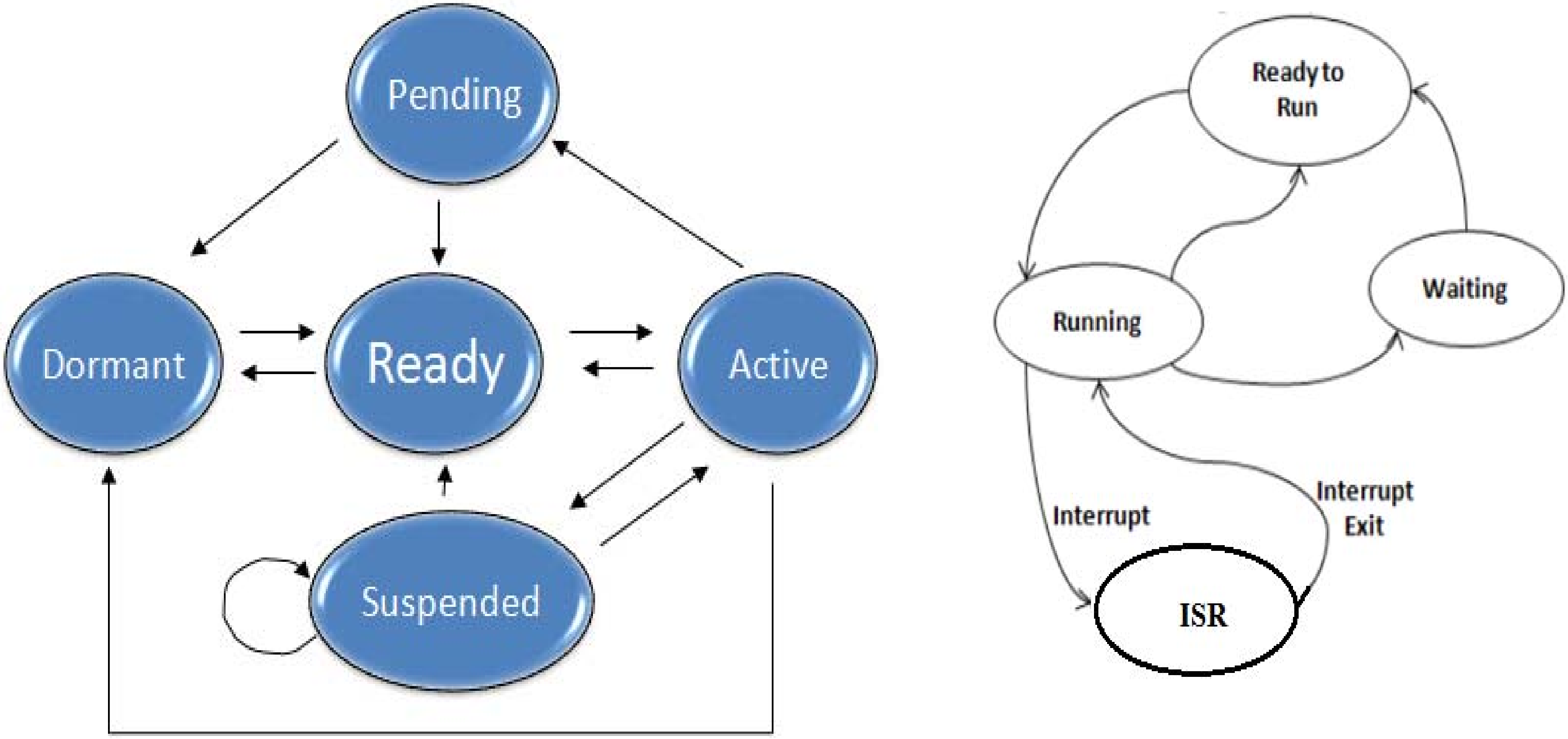
Tasks and Task Scheduler, Task States, Content Switching…..

There are two types of tasks.
(i)Simple Task(S-Task) and (ii) Complex Task(C-Task).

- Simple Task (S-task): A simple task is one which has no synchronization point i. e. , whenever an S-task is started, it continues until its termination point is reached. Because an S-task cannot be blocked within the body of the task the execution time of an S-task is not directly dependent on the progress of the other tasks in the node. S-task is mainly used for single user systems.

- Complex Task (C-Task): A task is called a complex task (C-Task) if it contains a blocking synchronization statement (e. g. , a semaphore operation "wait") within the task body. Such a "wait" operation may be required because the task must wait until a condition outside the task is satisfied, e. g. , until another task has finished updating a common data structure, or until input from a terminal has arrived.

Tasks and Task Scheduler, Task States, Content Switching…..



Task States

Task States

Each task may exist in following states

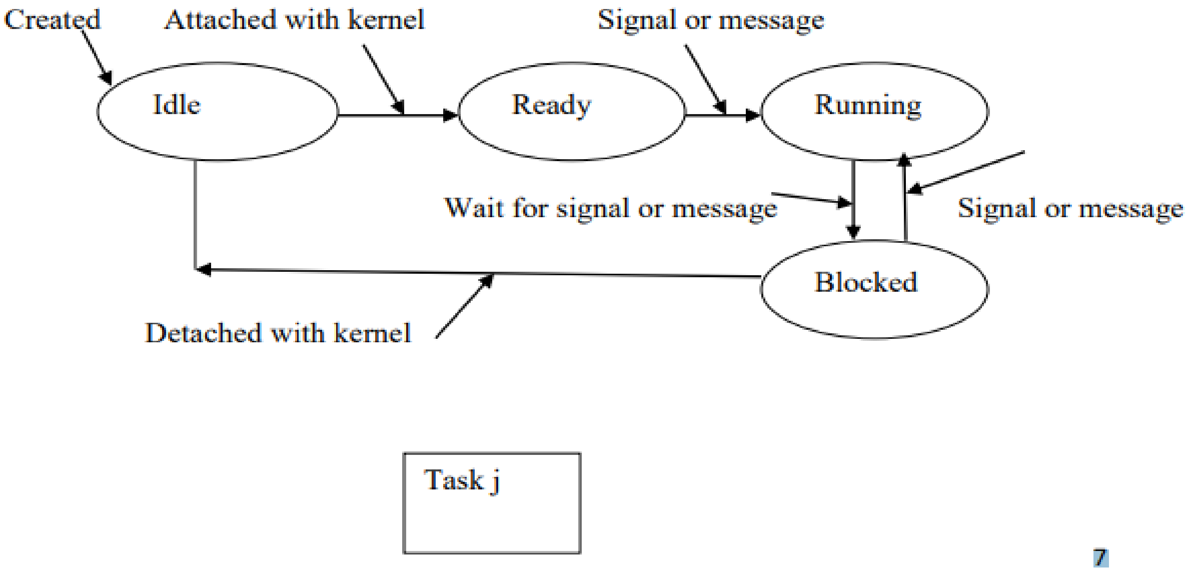
- Dormant : Task doesn’t require computer time ()
- Ready: Task is ready to go active state, waiting processor time
- Active: Task is running
- Suspended: Task put on hold temporarily
- Pending: Task waiting for resource.

Tasks and Task Scheduler, Task States, Content Switching…..

When a task is first created, it is in the dormant task. When it is added to RTOS for scheduling, it is a ready task. If the input or a resource is not available, the task gets blocked.

During the execution of an application program, individual tasks are continuously changing from one state to another. However, only one task is in the running mode (i. e. given CPU control) at any point of the execution. In the process where CPU control is change from one task to another, context of the to-be-suspended task will be saved while context of the to-be-executed task will be retrieved, the process referred to as context switching.

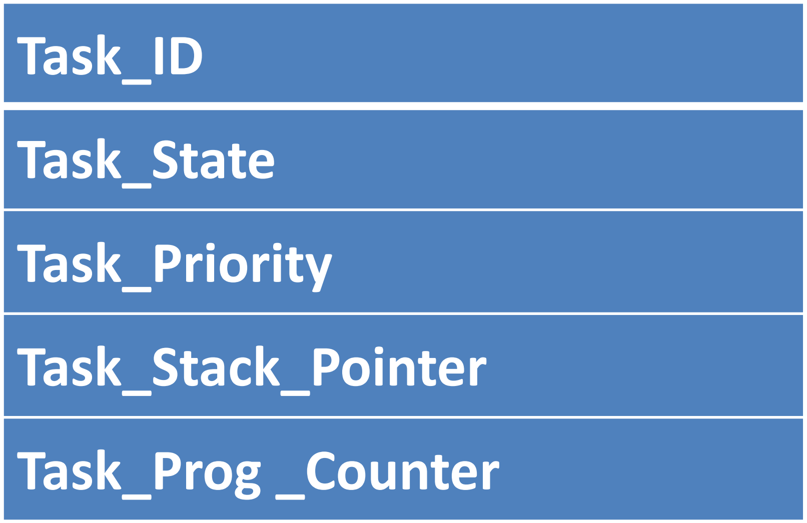
If no task is ready to run and all of the tasks are blocked, the RTOS will usually run the Idle Task. An Idle Task does nothing. The idle task has the lowest priority.



Tasks and Task Scheduler, Task States, Content Switching…..

A task object is defined by the following set of components:

- Task Control block: Task uses TCBs to remember its context. TCBs are data structures residing in RAM, accessible only by RTOS
- Task Stack: These reside in RAM, accessible by stack pointer.
- Task Routine: Program code residing in ROM.



Task_ID

Task_State

Task_Priority

Task_Stack_Pointer

Task_Prog _Counter

Tasks and Task Scheduler, Task States, Content Switching…..

Context Switching

The actual process of changing from one task to another is called a context switch.
The context switch is the mechanism for moving the CPU from one executing process to another.

- Clearly, the context switch must be bug-free-a process that does not look at a real-time clock should not be able to tell that it was stopped and then restarted.
- Cooperative multitasking-the most general form of context switching, preemptive multitasking.
- Preemptive Multitasking-the interrupt is an ideal mechanism on which to build context switching for preemptive multitasking.
- A timer generates periodic interrupts to the CPU.
- The interrupt handler for the timer calls the operating system, which saves the previous process's state in an activation record, selects the next process to execute, and switches the context to that process.
- Processes and Object-Oriented Design-UML often refers to processes as active objects, that is, objects that have independent threads of control.
- We can implement the preemptive context switches using the same basic techniques.

The only difference between the two is the triggering event, voluntary release of the CPU in the case of cooperative and timer interrupt in the case of preemptive.

SCHEDULING ALGORITHMS

In Multitasking system to schedule the various tasks, different scheduling algorithms are used. They are

(a) First in First out

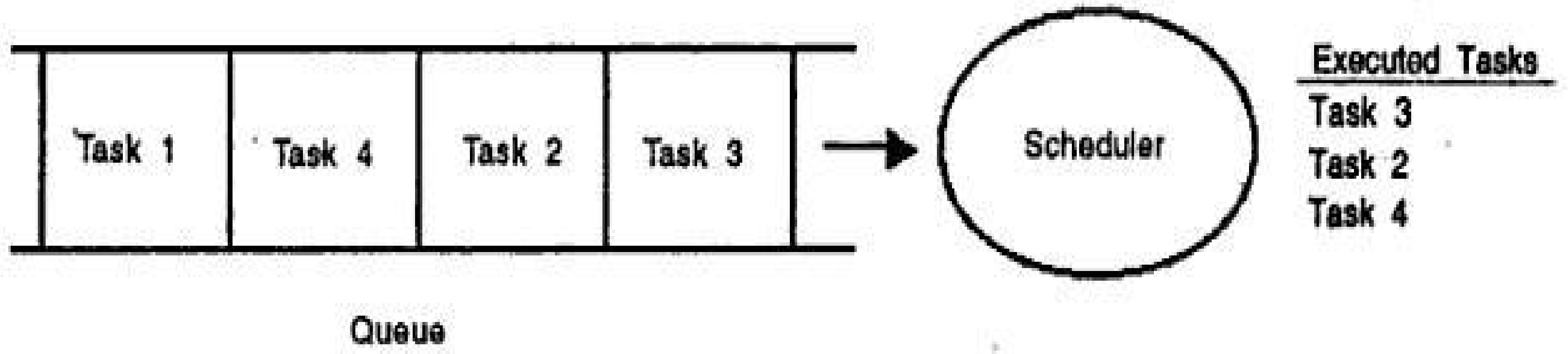
(b) Round Robin algorithm

(c) Round Robin with priority

(d) Non-preemptive

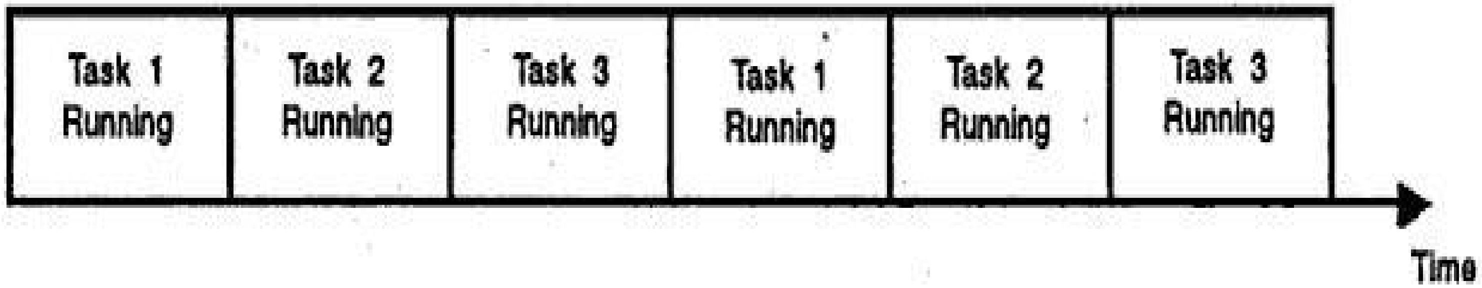
(e) Pre-emptive.

In FIFO scheduling algorithm, the tasks which are ready-to-run are kept in a queue and the CPU serves the tasks on first-come-first served basis.



SCHEDULING ALGORITHMS…..

In **Round-Robin Algorithm** the kernel allocates a certain amount of time for each task waiting in the queue. For example, if three tasks 1, 2 and 3 are waiting in the queue, the CPU first executes task1 then task2 then task3 and then again task1.



Task 1 Running

Task 2 Running

Task 3 Running

Task 1 Running

Task 2 Running

Task 3 Running

Time

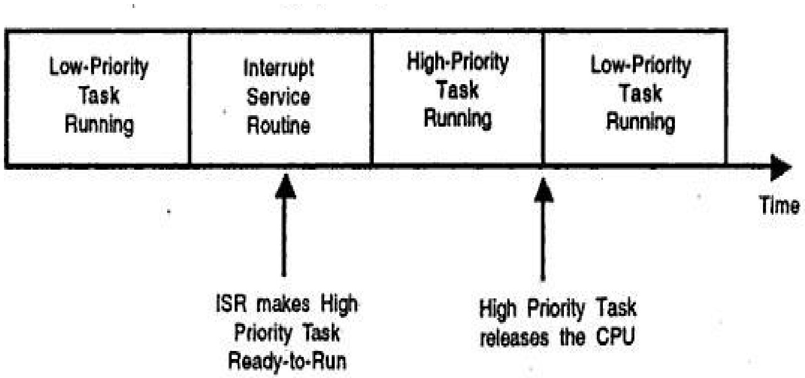
The round-robin algorithm can be slightly modified by assigning priority levels to the tasks. A high priority task can interrupt the CPU so that it can be executed. This scheduling algorithm can meet the desired response time for a high priority task. This is the **Round Robin with priority.**

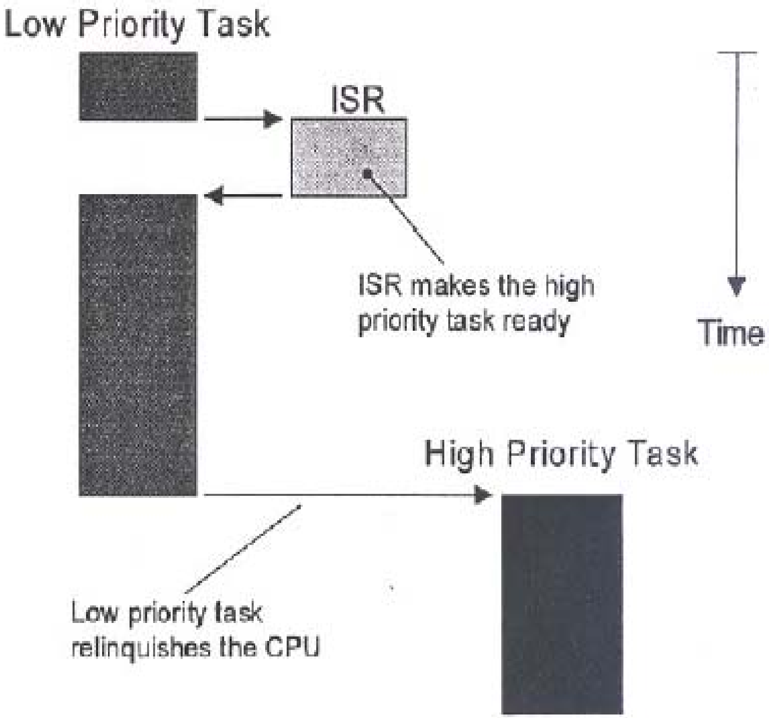
In **Shortest-Job First scheduling algorithm**, the task that will take minimum time to be executed will be given priority. The disadvantage of this is that as this approach satisfies the maximum number of tasks, some tasks may have to wait forever.

SCHEDULING ALGORITHMS…..

Non-preemptive Multitasking

- In Non-preemptive multitasking a task is designed to relinquish control of the CPU to the kernel at regular intervals.
- To the implementation of non-preemptive multitasking is the use of a message queue.
- The kernal manages the message queue which is essentially a queue of message numbers & small amount of data.


Low-Priority Task Running
Interrupt Service Routine
High-Priority Task Running
Low-Priorty Task Running
Time
ISR makes High Priority Task Ready-to-Run
High Priority Task releases the CPU

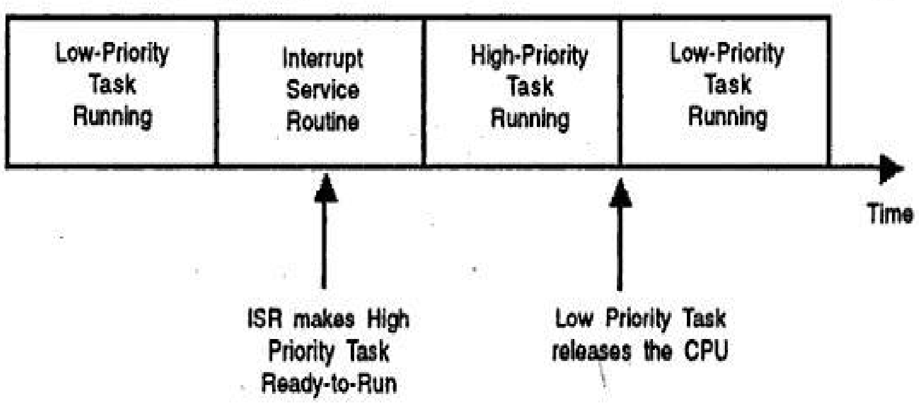

Low Priority Task
ISR
ISR makes the high priority task ready
Time
High Priority Task
Low priority task relinquishes the CPU

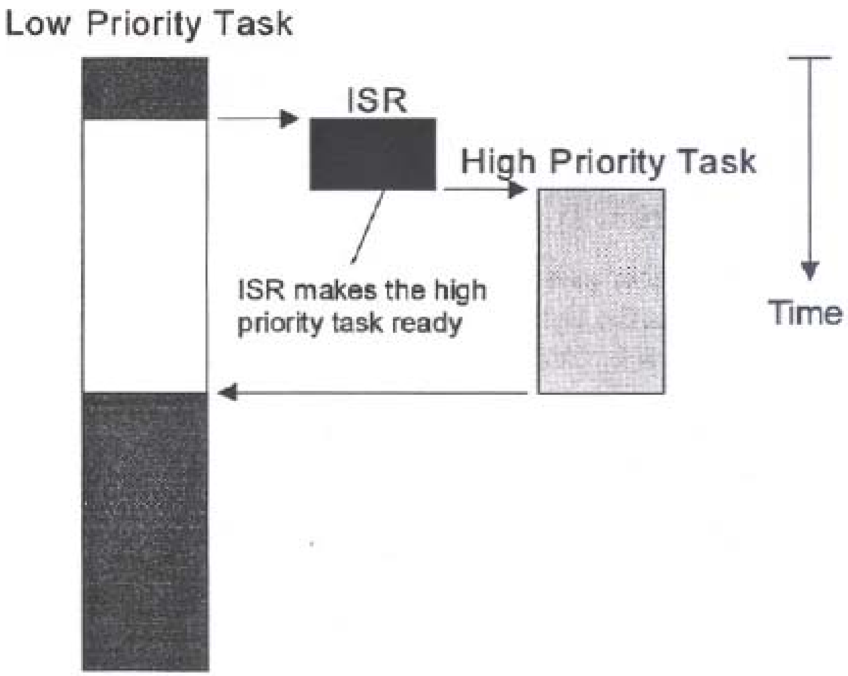
Some mistakes in the text book diagram.
So refer the class notes

SCHEDULING ALGORITHMS…..

In preemptive multitasking, the highest priority task is always executed by the CPU, by preempting the lower priority task. All real-time operating systems implement this scheduling algorithm.

- An external tick interrupt, interrupts task at an indeterminate point and passes control to kernel program.
- The kernel will save the state of the interrupted task ad then determine which task it should run next.
- The Kernel restore the state of task and pass control of the CPU to that task.
- Task will continue to run until it is interrupted by next external tick interrupt or voluntarily gives up allotted time slice.


Low-Priority Task Running
Interrupt Service Routine
High-Priority Task Running
Low-Priority Task Running
Time
ISR makes High Priority Task Ready-to-Run
Low Priority Task releases the CPU



Small mistake in the text book diagram.
So refer the class notes

Scheduling Algorithms…. Rate Monotonic Analysis

Optimal Scheduling Algorithms:

Rate Monotonic (RM)

- Higher rate (1/period) → Higher priority
- Optimal preemptive static priority scheduling algorithm

Earliest Deadline First (EDF)

- Earlier absolute deadline → Higher priority
- Optimal preemptive dynamic priority scheduling algorithm

Assumptions:

- Single processor.
- All tasks are periodic and Tasks are independent.
- Zero context switch time. (Tasks are always released at the start of their periods)
- Relative deadline = period. (Di=Ti)
- No priority inversion.
- RM and EDF have been extended to relax assumptions.

Scheduling Algorithms…. Rate Monotonic Analysis…

A static scheduling algorithm by Lieu and Layland (1973)

Properties of RM:

- RM may not guarantee schedulability even when CPU is not fully utilized.
- Low overhead: when the task set is fixed, the priority of a task never changes.
- Easy to implement on POSIX APIs.

Utilization of a processor:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i}$$

– n: number of tasks on the processor.

- Utilization bound Ub: All tasks are guaranteed to be schedulable if U ≤ Ub.
- No scheduling algorithm can schedule a task set if U>1

Scheduling Algorithms…. Rate Monotonic Analysis…

Example: scheduling using RM

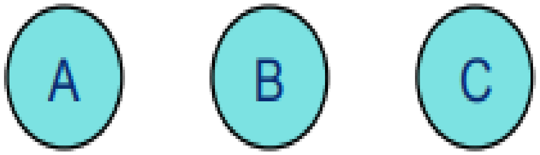
Problem: Assume a system with tasks according to the figure below.

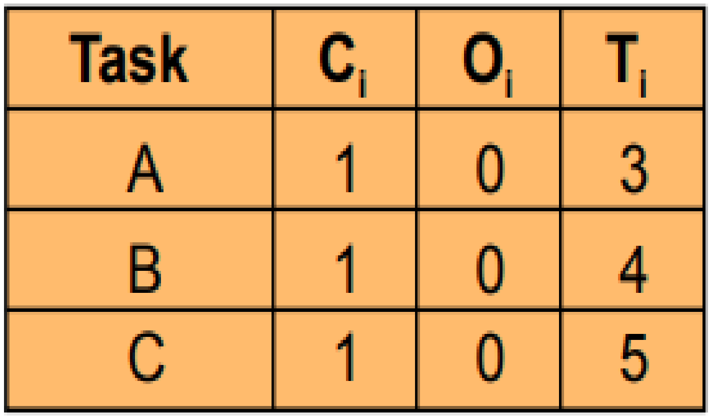
The timing properties of the tasks are given in the table. Schedule the tasks using rate-monotonic scheduling (RM).

a) What is the utilization of the task set?

b) What is the outcome of Liu & Layland’s feasibility test for RM?

c) Show that the tasks are schedulable using RM.





Task	C_i	O_i	T_i
A	1	0	3
B	1	0	4
C	1	0	5

Scheduling Algorithms…. Rate Monotonic Analysis…

a) The utilization of the system is

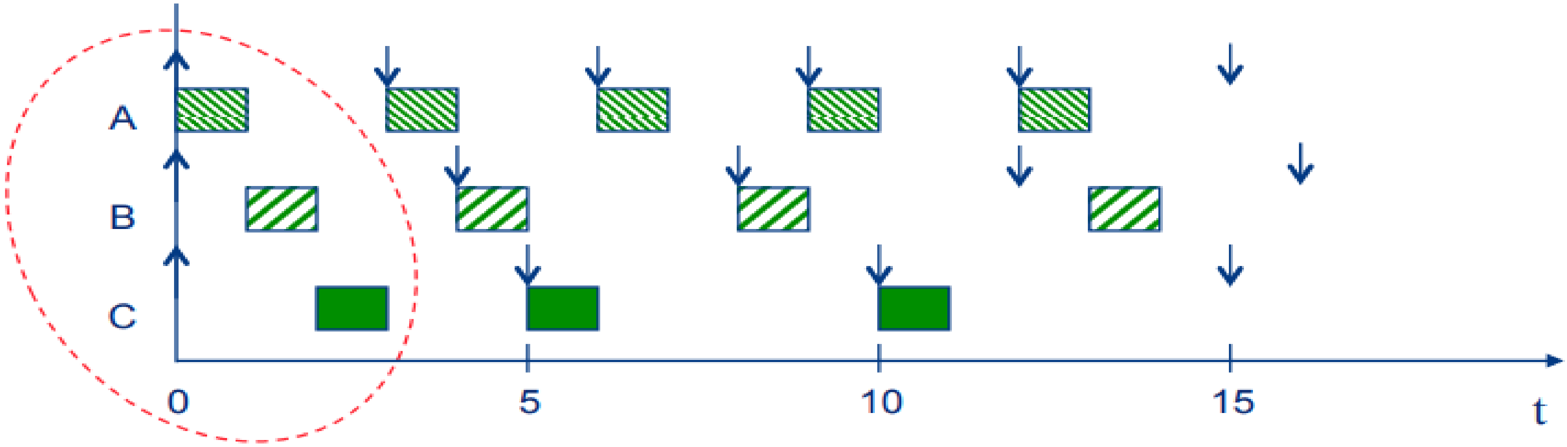
$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} = \frac{1}{3} + \frac{1}{4} + \frac{1}{5} \approx 0.783$$

b) The utilization bound Ulub of the test is

$$U_{lub} = n\left(2^{1/n} - 1\right) = 3\left(2^{1/3} - 1\right) \approx 0.780$$

Since U >Ulub and the test is only a sufficient one, we cannot yet determine whether the task set is schedulable or not.

c) Simulate an execution of the tasks:



Scheduling Algorithms…. Rate Monotonic Analysis…

Liu & Layland have shown that if the task set is schedulable when all tasks arrive at the same time (i.e., at t = 0), then the task set is also schedulable in all other cases. Hence, it is enough to demonstrate that the first instance of each task will meet its deadline.

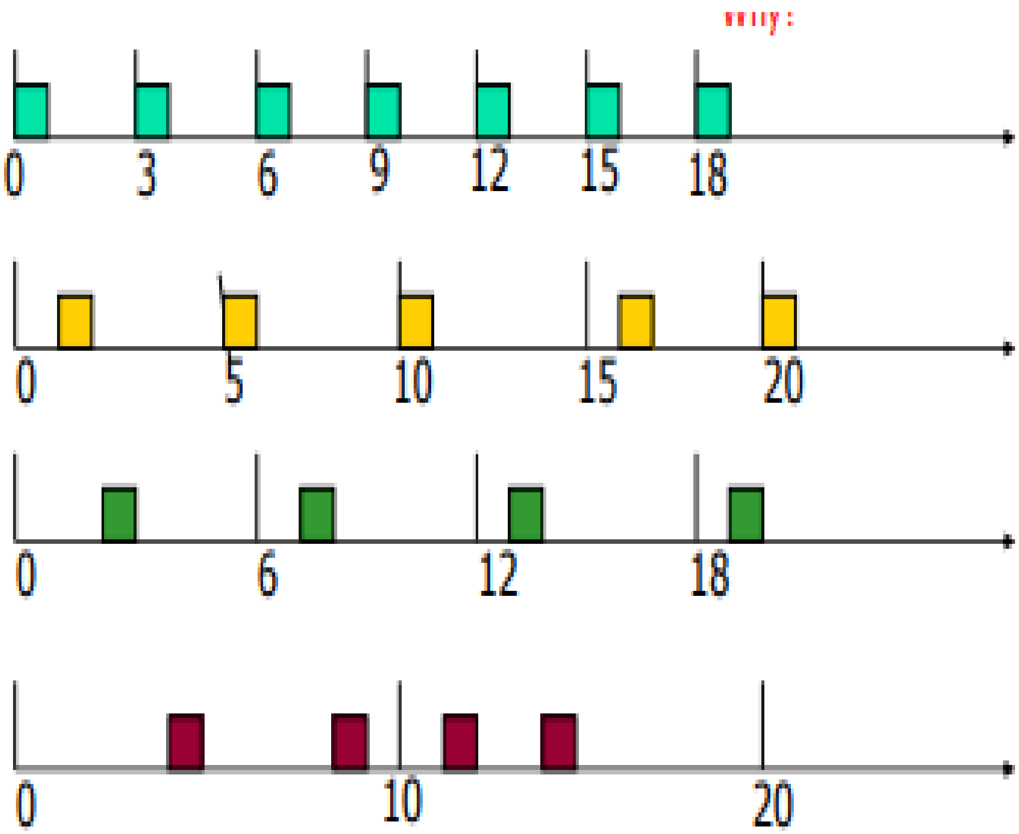
RMS: Summary

1. Task model:
   - periodic, independent, D=T, and a task= (Ci,Ti)
2. Fixed-priority assignment:
   - smaller periods = higher priorities
3. Run time scheduling: Preemptive HPF
4. Sufficient schedulability test: $U <= n*(2^{1/n}-1)$
5. Precise/exact schedulability test exists

Scheduling Algorithms…. Rate Monotonic Analysis…

Ex: 2

- Assume a task set: {(1,3),(1,5),(1,6),(2,10)}
- CPU utilization U= 1/3+1/5+1/6+2/10=0.899
- The utilization bound B(4)=0.756
- The task set fails in the UB test due to U>B(4)
- Question: is the task set schedulable?
- Answer: YES



This is only for the first periods! But we will see that this is enough to tell that the task set is schedulable.

Scheduling Algorithms…. Rate Monotonic Analysis…

Advantages:

- Simple to understand (and remember!)
- Easy to implement (static/fixed priority assignment)
- Stable: though some of the lower priority tasks fail to meet deadlines, others may meet deadlines

Disadvantages:

- “Lower” CPU utilization
- Requires D=T
- Only deal with independent tasks
- Non-precise schedulability analysis

Task Management Function Calls

The various function calls provided by the OS API for task management are

- Create a task → create a new TCB
- Delete a task → remove the TCB
- Suspend a task
- Resume a task
- Change priority of a task → modify the TCB
- Query a task → read the TCB

Task Management Function Calls…..

Challenges for an RTOS (Task Management)

- Creating an RT task, it has to get the memory without delay: this is difficult because memory has to be allocated and a lot of data structures, code segment must be copied/initialized

- The memory blocks for RT tasks must be locked in main memory to avoid access latencies due to swapping

- Changing run-time priorities is dangerous: it may change the runtime behaviour and predictability of the whole system

Task Management Function Calls…..

Challenges for an RTOS (Task Management)

- Creating an RT task, it has to get the memory without delay: this is difficult because memory has to be allocated and a lot of data structures, code segment must be copied/initialized

- The memory blocks for RT tasks must be locked in main memory to avoid access latencies due to swapping

- Changing run-time priorities is dangerous: it may change the runtime behaviour and predictability of the whole system

Task Management Function Calls…..

Task Interaction

Tasks execute asynchronously, i.e., at different speeds, but may need to interact with each other.

Three types of interactions are possible

- ➢ communication
- ➢ synchronization
- ➢ mutual exclusion

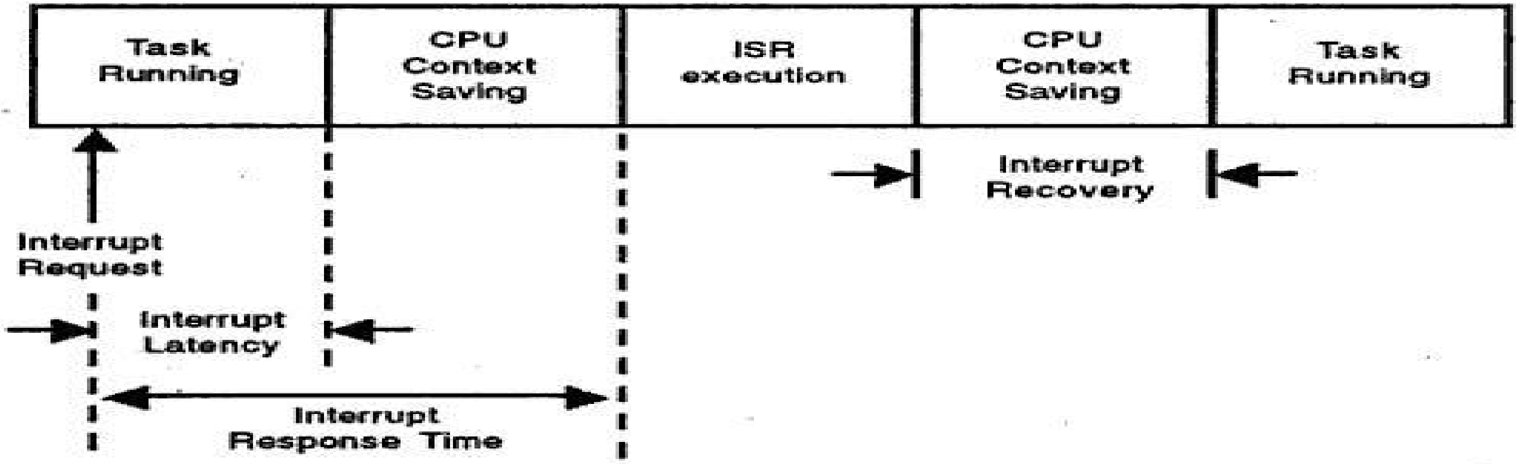
- ▪ Communication is simply used to transfer data between tasks.
- ▪ Synchronization is used to coordinate tasks.
- ▪ Mutual exclusion is used to control access to shared resources.

Interrupt Service Routines

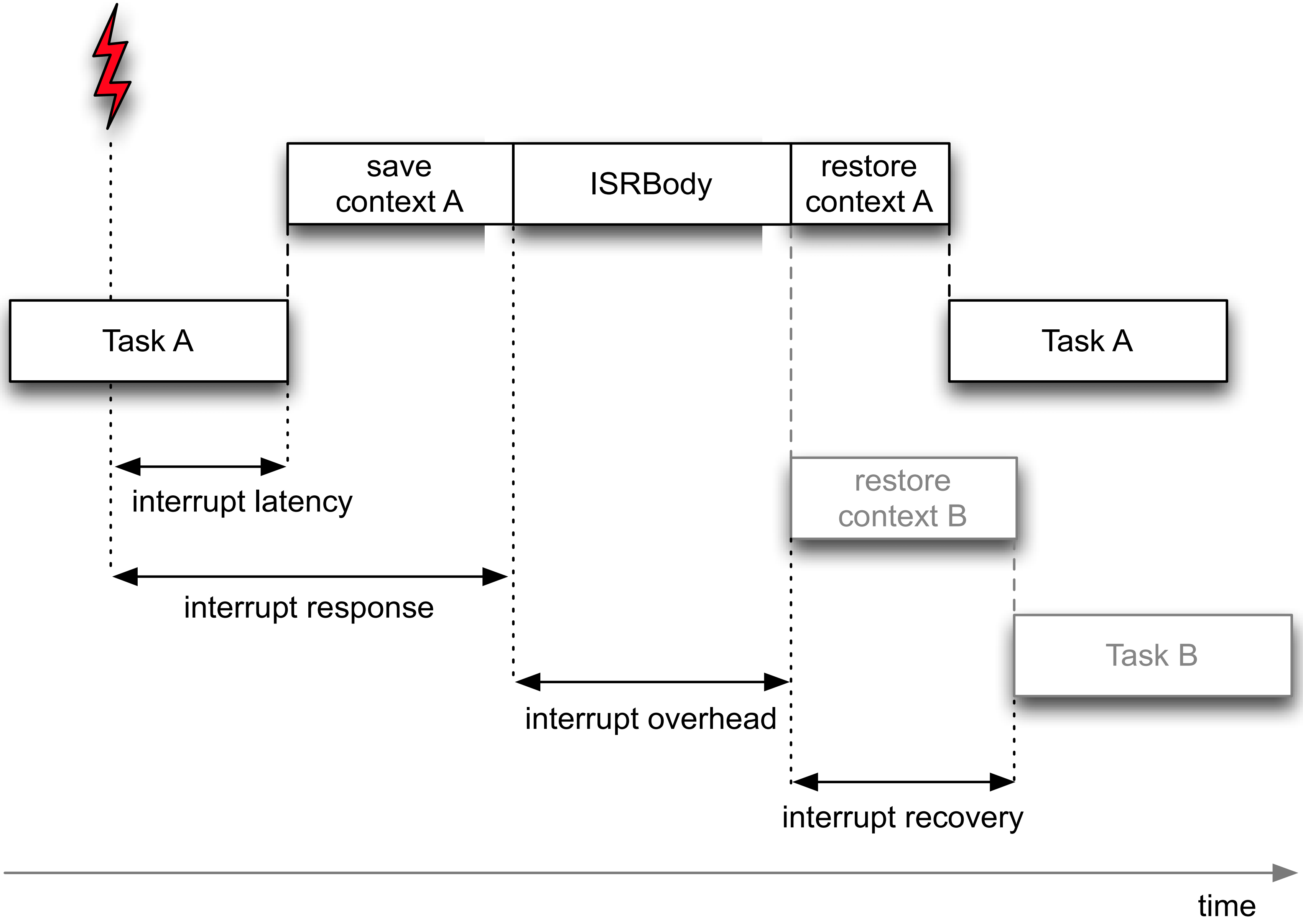
An interrupt service routine (ISR), also known as an interrupt handler, is a call back subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt.

In a real-time embedded system, there are two possible interrupts:
one is the Hardware Interrupt and the other is the software Interrupt.

- Hardware Interrupts are asynchronous interrupts which are triggered by an electric pulse
- Software interrupts are synchronous interrupts and these are triggered by a command or instruction



Interrupt Service Routines…



Interrupt Service Routines…

ISR is a small program, which is executed to develop an interface between the user and the hardware. The CPU will execute the ISR subroutine when it receives either a hardware or software interrupt.

The synchronization mechanism cannot be used in an ISR, because it is not possible in an ISR to wait indefinitely for a resource to be available.

The faster the ISR can do its job, the better the real time performance of the RTOS. Hence the ISR should be always as small as possible.

When the CPU receives either a software or hardware interrupts, it will try to execute the corresponding ISR. Before that all the other interrupt sources are disabled and the interrupts are enabled only after the completion of the ISR. Hence the CPU must execute the ISR as fast as possible and also the ISR must be always as small as possible.

Interrupt Service Routines…

In real-time operating systems, the interrupt latency, interrupt response time and the interrupt recovery time are very important.

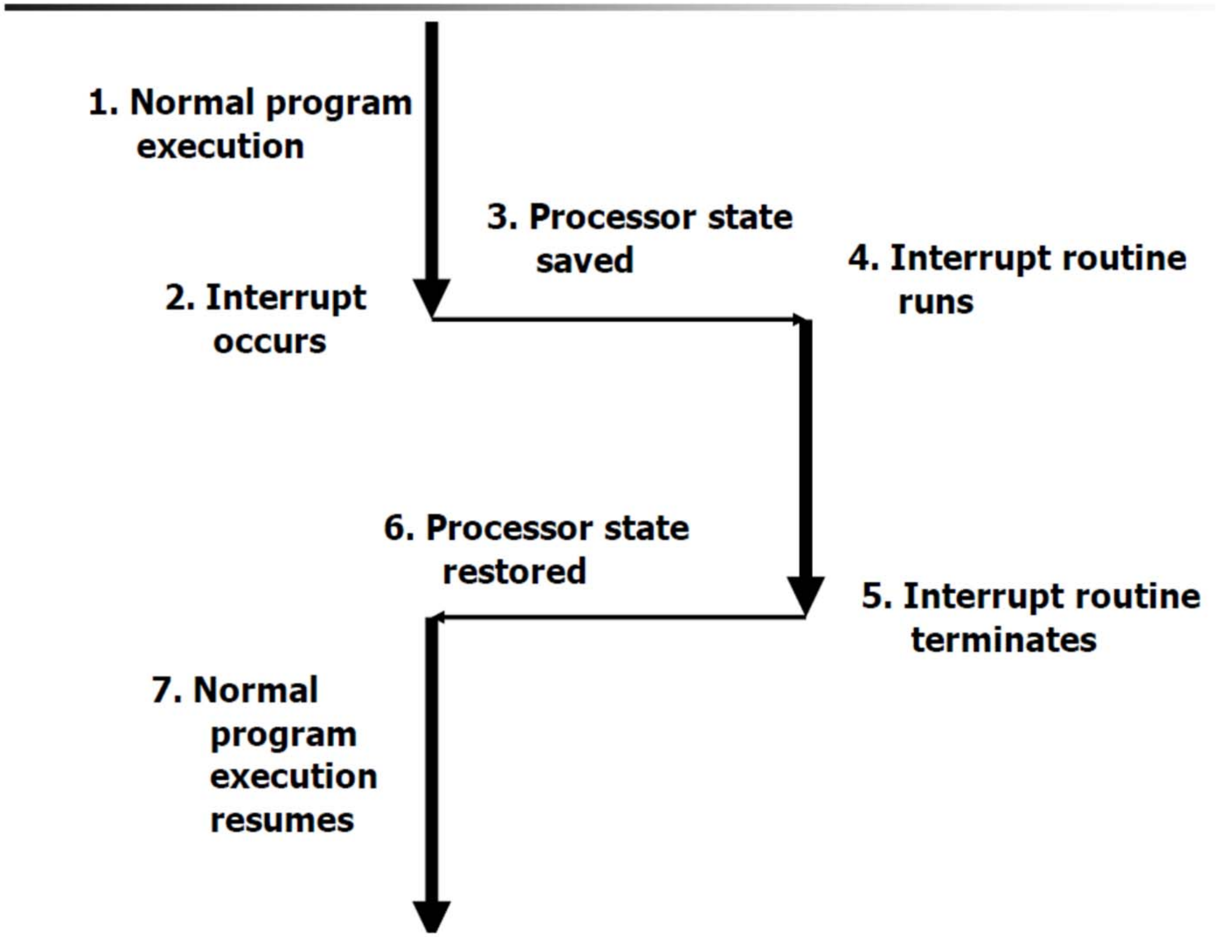
Interrupt Latency: It is the time between the generation of an interrupt by a device and the servicing of the device which generated the interrupt.
For many operating systems, devices are serviced as soon as the device's interrupt handler is executed. Interrupt latency may be affected by interrupt controllers, interrupt masking, and the operating system's (OS) interrupt handling methods.

Interrupt Response Time : Time between receipt of interrupt signal and starting the code that handles the interrupt is called interrupt response time.

Interrupt Recovery Time: Time required for CPU to return to the interrupted code/highest priority task is called interrupt recovery time.

Interrupt Service Routines…

Handling an Interrupt



1. Normal program execution
2. Interrupt occurs
3. Processor state saved
4. Interrupt routine runs
5. Interrupt routine terminates
6. Processor state restored
7. Normal program execution resumes

SEMAPHORES

A semaphore is nothing but a value or variable or data which can control the allocation of a resource among different tasks in a parallel programming environment.

The concept of semaphore was first proposed by the Dutch computer scientist Edsger Dijkstra in the year 1965.
So, Semaphores are a useful tool in the prevention of race conditions and deadlocks; however, their use is by no means a guarantee that a program is free from these problems.

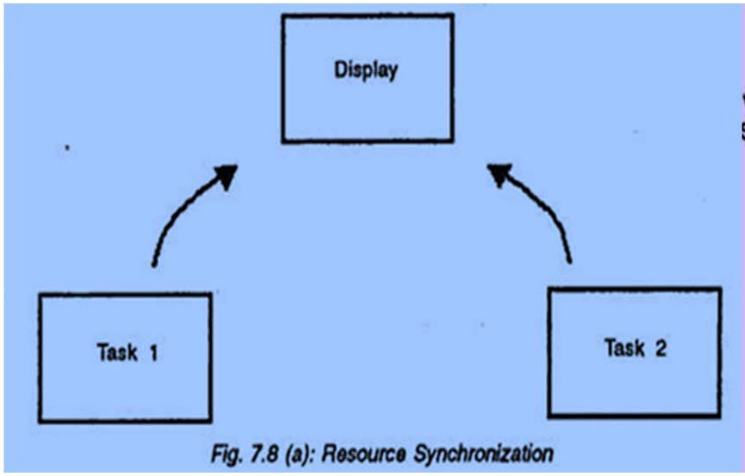


Fig. 7.8 (a): Resource Synchronization

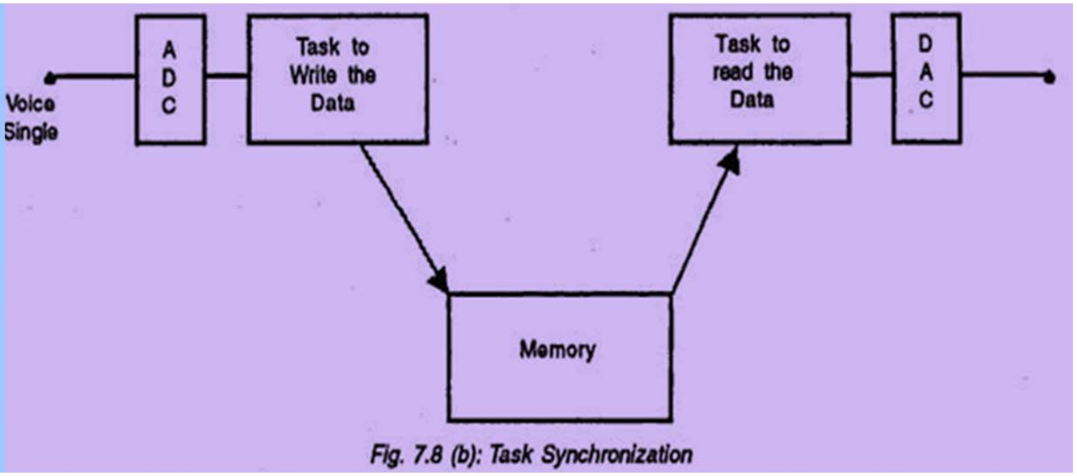


Fig. 7.8 (b): Task Synchronization

Task1 wants to display "Temperature is 50"
Task2 wants to display "Humidity is 40%"

Task1 wants to write the ADC data into memory.
Task2 wants to read the data into memory.

SEMAPHORES…..

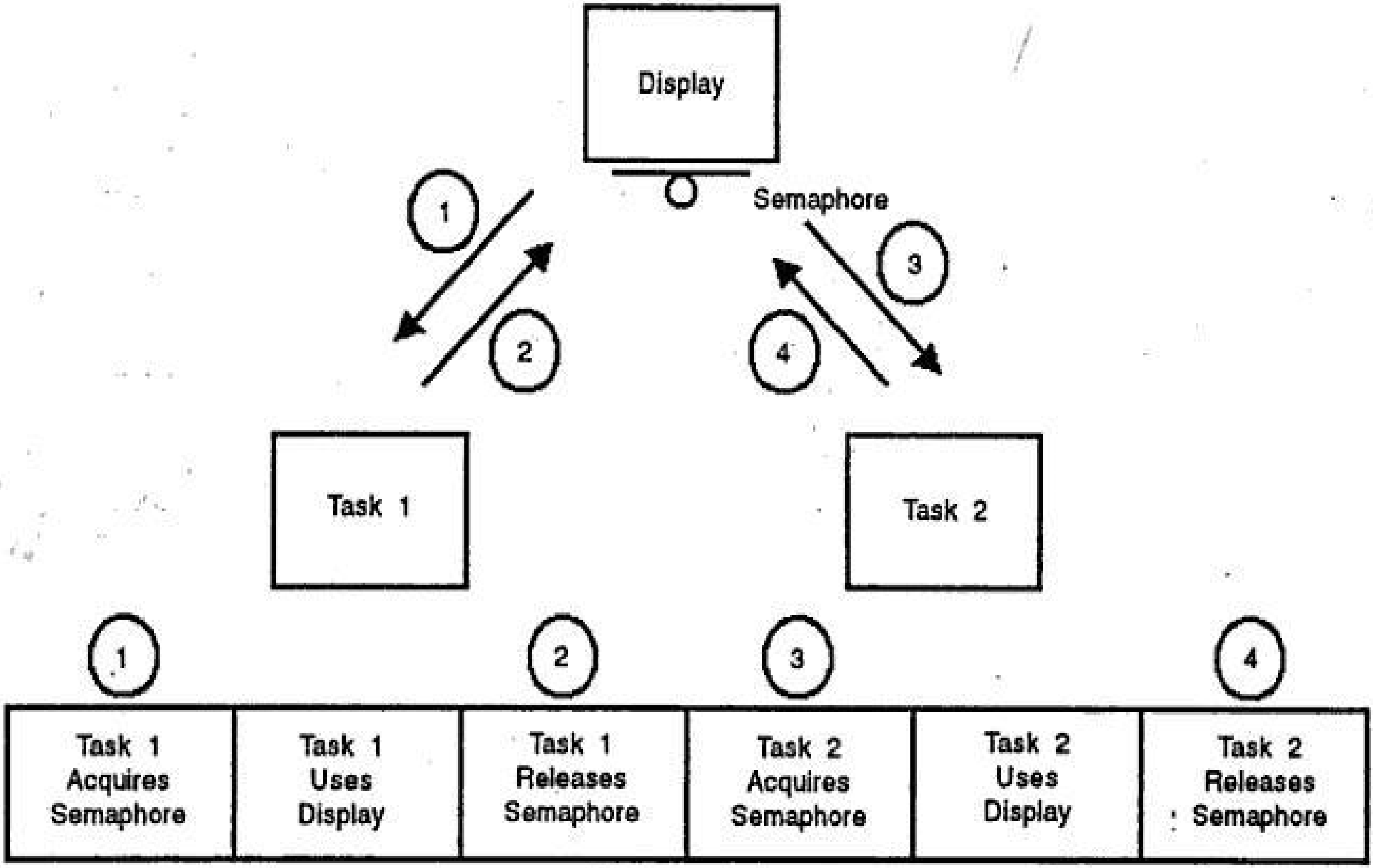

Display
Semaphore
1
2
3
4
Task 1
Task 2
1
2
3
4
Task 1 Acquires Semaphore
Task 1 Uses Display
Task 1 Releases Semaphore
Task 2 Acquires Semaphore
Task 2 Uses Display
Task 2 Releases Semaphore

Fig. 7.9: Display Semaphore

SEMAPHORES…..

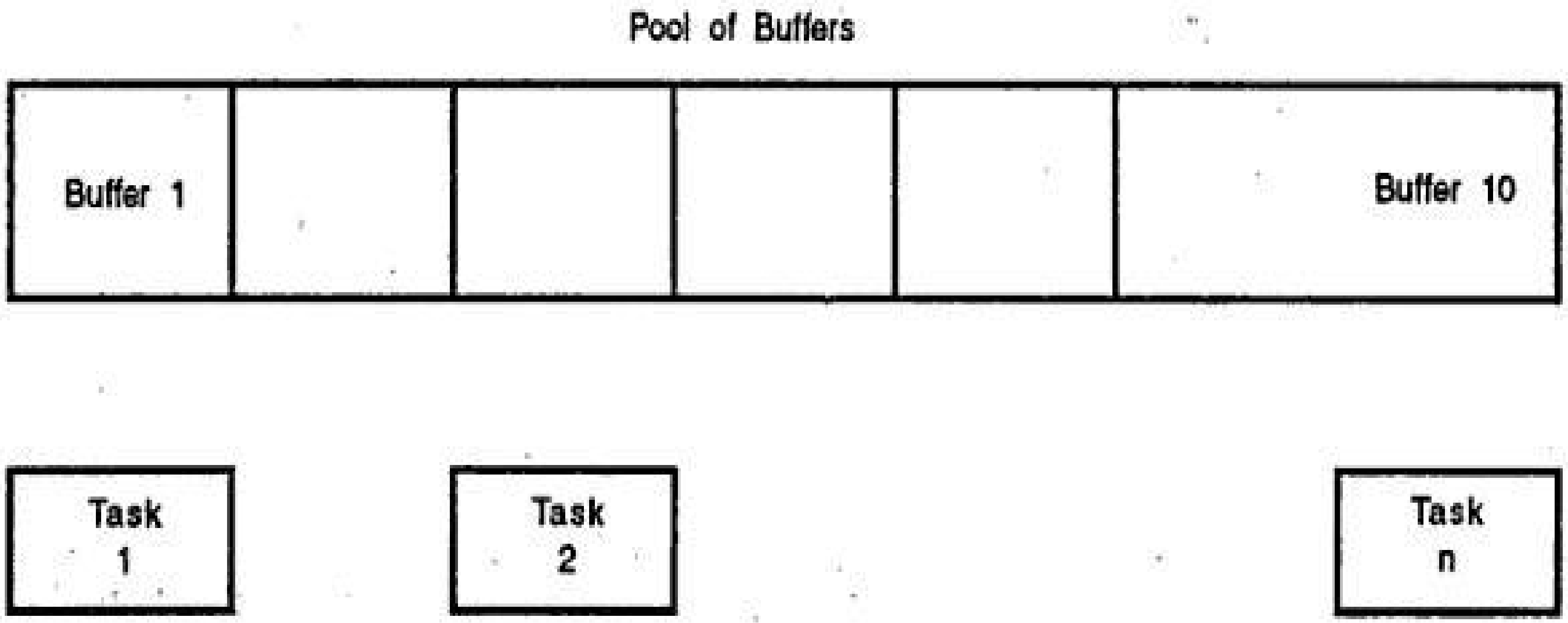


Fig. 7.10: Counting Semaphore

More no. of Tasks wants to access the same resource, then tasks are kept in queue.
Pool of 10 buffers is available, so any task can Write to a buffer.
The initial value is 10, and whenever a task acquires the semaphore the value is decremented by 1, if it reaches 0 then the shared resource is no longer available.
For 8 bit integer semaphore can take value between 0 to 255.

SEMAPHORES…..

Semaphores are system-implemented data structures that are shared between processes.

Features of Semaphores:

- Semaphores are used to synchronize operations when processes access a common, limited, and possibly non-shareable resource.
- Each time a process wants to obtain the resource, the associated semaphore is tested. A positive, non-zero semaphore value indicates the resource is available.

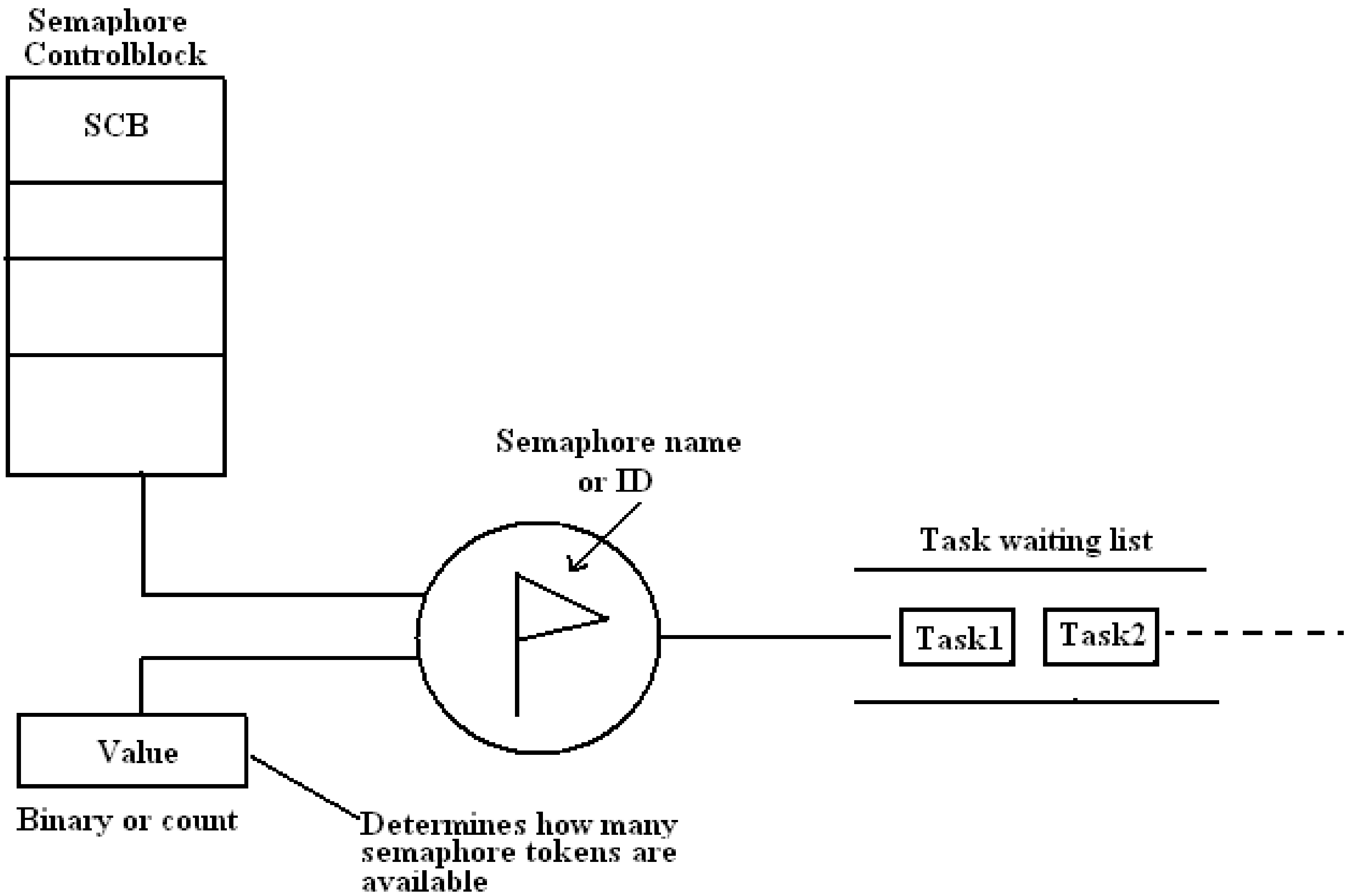
Semaphore Operations:

- wait: P(S) → P from Dutch proberen, to test
  - wait tests semaphore value, and if false, sets to true
  - wait has two parts- test and set, which must occur together atomically
- signal: V(S) → V from Dutch verhogen, to increment
  - sets value to false

Semaphore Control

- get info on a set
- set all semaphore values in a set or get all values
- set one value in a set or get one value
- remove the set

SEMAPHORES…..


Semaphore Controlblock
SCB
Semaphore name or ID
Task waiting list
Task1
Task2
Value
Binary or count
Determines how many semaphore tokens are available

SEMAPHORES…..

Semaphore Operations…

```
wait (aSemaphore)
    if aSemaphore's counter > 0 then
        decrement aSemaphore's counter
    else
        put the caller in aSemaphore's queue
        attempt to transfer control to some ready task
        (if the task ready queue is empty, deadlock occurs)
end

release(aSemaphore)
        if aSemaphore's queue is empty (no task is waiting) then
                increment aSemaphore's counter
        else
                put the calling task in the task ready queue
                transfer control to a task from aSemaphore's queue
end
```

Example of semaphore:

Producer and consumer problem

The producer should stop producing when the warehouse is full. The consumer should stop consuming when the warehouse is empty

Semaphore Management Function Calls:

- Create a semaphore
- Delete a semaphore
- Acquire a semaphore
- Release a semaphore
- Query a semaphore

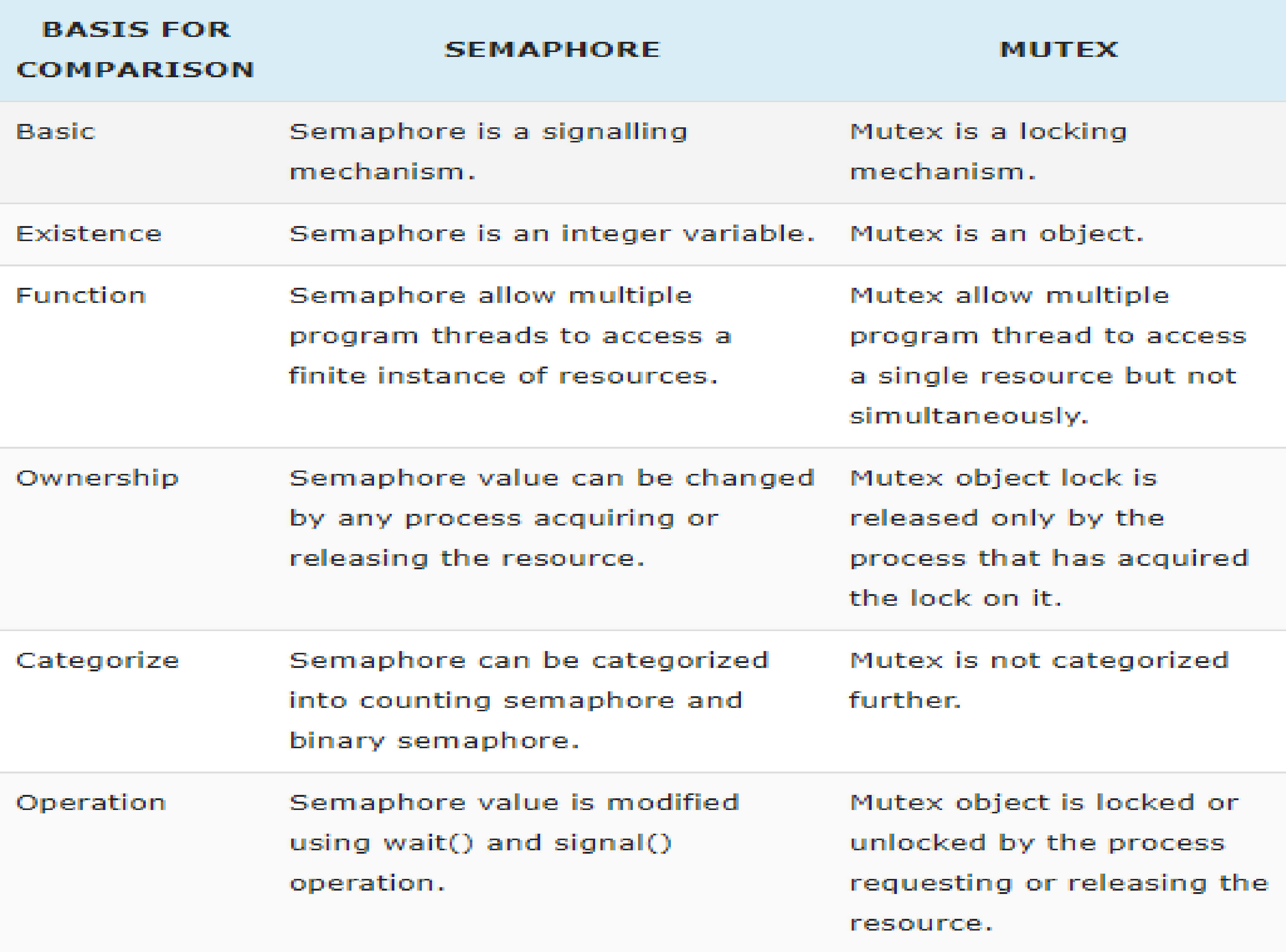
MUTEX (Mutual Exclusion)

Mutex means mutual exclusion. A mutex is a synchronization object that can have only two states. They are not-owned and owned.

Two operations are defined for mutexes.
Lock: This operation attempts to take ownership of a mutex, if the mutex is already owned by another thread then the invoking thread is queued.
Unlock: This operation relinquishes ownership of a mutex. If there are queued threads then a thread is removed from the queue and resumed, ownership is implicitly assigned to the thread.

Mutex is basically a locking mechanism where a process locks a resource using mutex. As long as the process has mutex, no other process can use the same resource. Once process is done with resource, it releases the mutex. Here comes the concept of ownership. Mutex is locked and released by the same process/thread.
Most RTOSs implement this protocol in order to address the Priority Inversion problem. Semaphores can also handle mutual exclusion problems but are best used as a communication mechanism between threads or between ISRs and threads.

MUTEX (Mutual Exclusion)…..



BASIS FOR COMPARISON	SEMAPHORE	MUTEX
Basic	Semaphore is a signalling mechanism.	Mutex is a locking mechanism.
Existence	Semaphore is an integer variable.	Mutex is an object.
Function	Semaphore allow multiple program threads to access a finite instance of resources.	Mutex allow multiple program thread to access a single resource but not simultaneously.
Ownership	Semaphore value can be changed by any process acquiring or releasing the resource.	Mutex object lock is released only by the process that has acquired the lock on it.
Categorize	Semaphore can be categorized into counting semaphore and binary semaphore.	Mutex is not categorized further.
Operation	Semaphore value is modified using wait() and signal() operation.	Mutex object is locked or unlocked by the process requesting or releasing the resource.

Mutex…..

Mutex Management Function Calls:

The OS functions calls provided for mutex management are

- Create a mutex
- Delete a mutex
- Acquire a mutex
- Release a mutex
- Query a mutex
- Wait on a mutex

MAILBOXES

One of the important Kernel services used to sent the Messages to a task is the message mailbox. A Mailbox is basically a pointer size variable. Tasks or ISRs can deposit and receive messages (the pointer) through the mailbox.

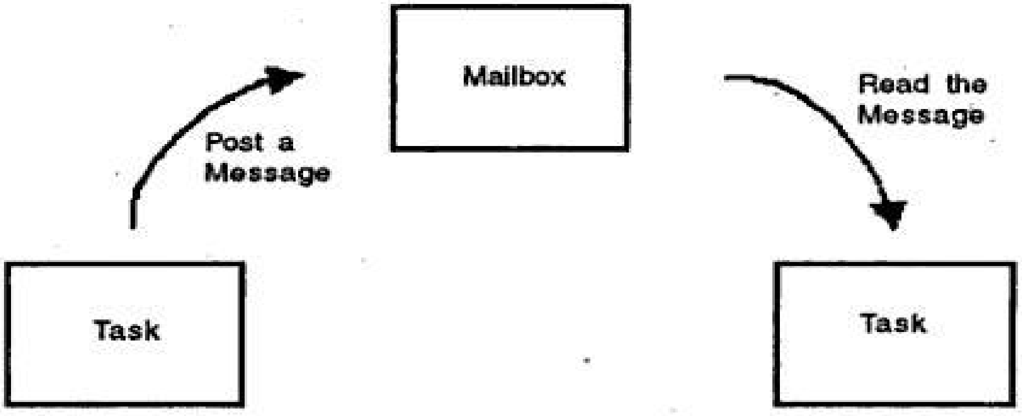


Fig. 7.11 Inter-task Synchronization through Mailbox

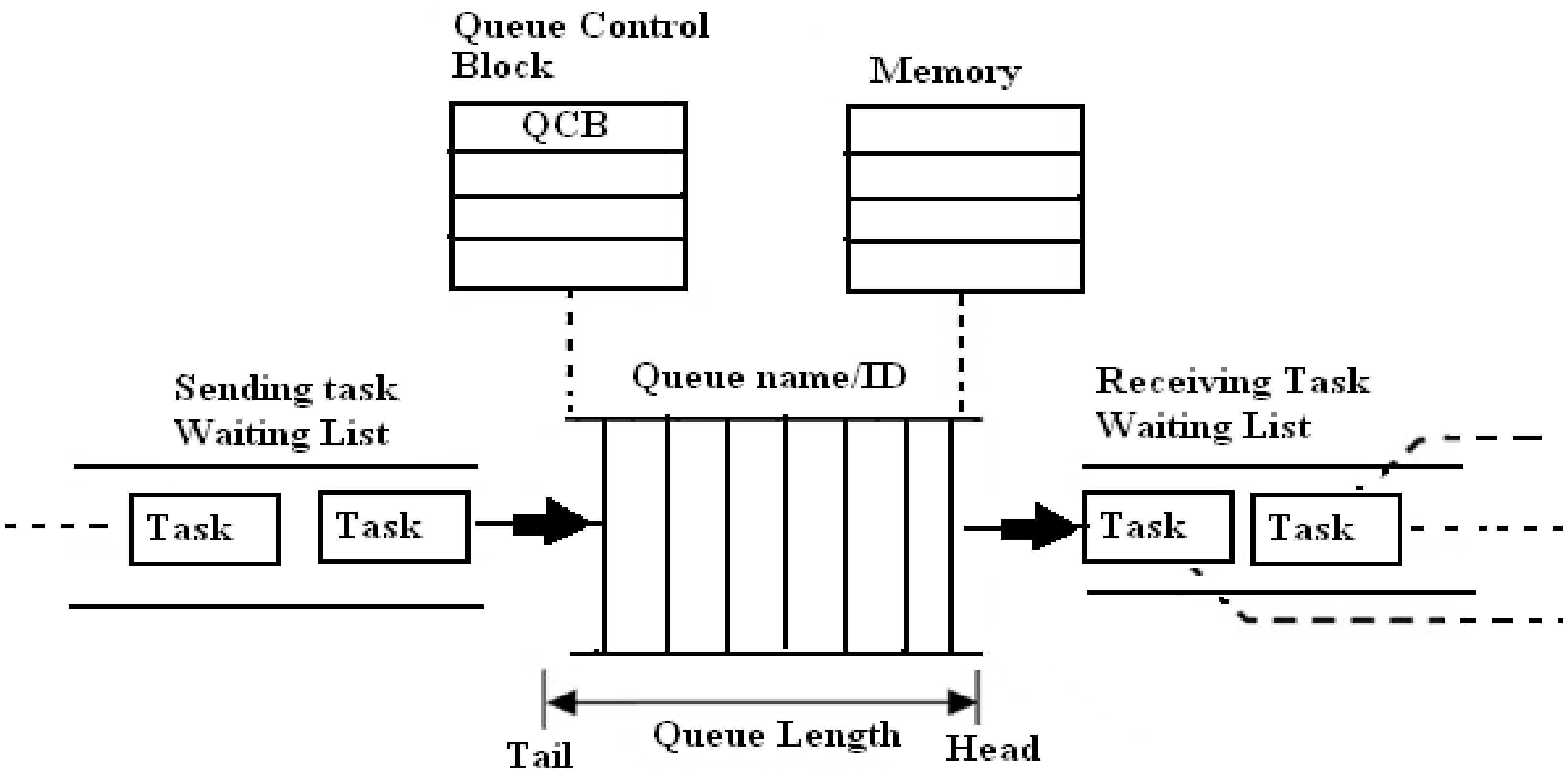
A task looking for a message from an empty mailbox is blocked and placed on waiting list for a time (time out specified by the task) or until a message is received.

When a message is sent to the mail box, the highest priority task waiting for the message is given the message in priority-based mailbox or the first task to request the message is given the message in FIFO based mailbox.

MAILBOXES…

The operation of a mailbox object is similar to our postal mailbox. When someone posts a message in our mailbox, we take out the message.

A task can have a mailbox into which others can post a mail. A task or ISR sends the message to the mailbox.



MAILBOXES…

Mailbox Management Function Calls:

To manage the mailbox object, the following function calls are provided in the OS API:

- Create a mailbox
- Delete a mailbox
- Query a mailbox
- Post a message in a mailbox
- Read a message form a mailbox.

MESSAGE QUEUES

The Message Queues, are used to send one or more messages to a task, i.e., the message queues are used to establish the Inter task communication. Basically Queue is an array of mailboxes. Tasks and ISRs can send and receive messages to the Queue through services provided by the kernel. Extraction of messages from a queue follow FIFO or LIFO structure.

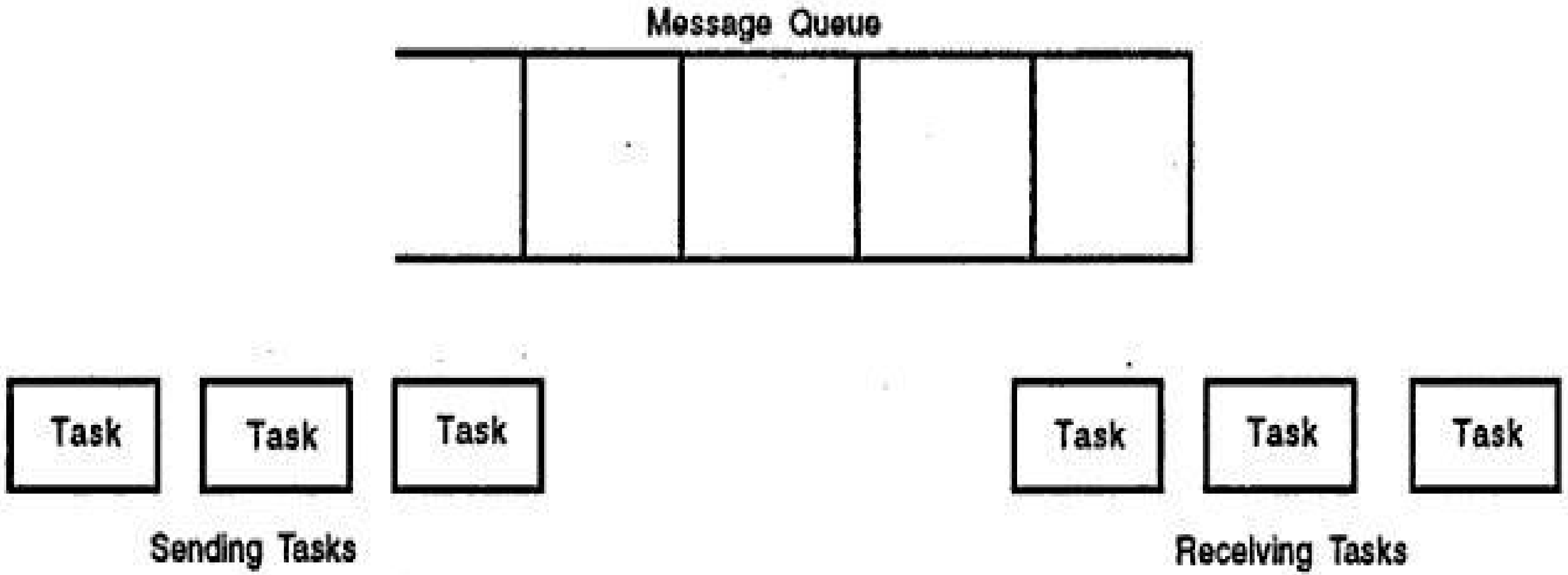


Fig. 7.12: Message Queue

MESSAGE QUEUES…

Applications of message queue are

- ➢ Taking the input from a keyboard
- ➢ To display output
- ➢ Reading voltages from sensors or transducers
- ➢ Data packet transmission in a network

In each of these applications, a task or an ISR deposits the message in the message queue. Other tasks can take the messages. Based on our application, the highest priority task or the first task waiting in the queue can take the message. At the time of creating a queue, the queue is given a name or ID, queue length, sending task waiting list and receiving task waiting list.

To use a message queue, first it must be created.
The creation of a Queue return a queue ID. So, if any task wish to post some message to a task, it should use its queue ID.
qid = queue_create("MyQueue", Queue_options) ; //*Queue name and OS specification options*//
Each queue can be configured as a fixed size/variable size.

MESSAGE QUEUES…

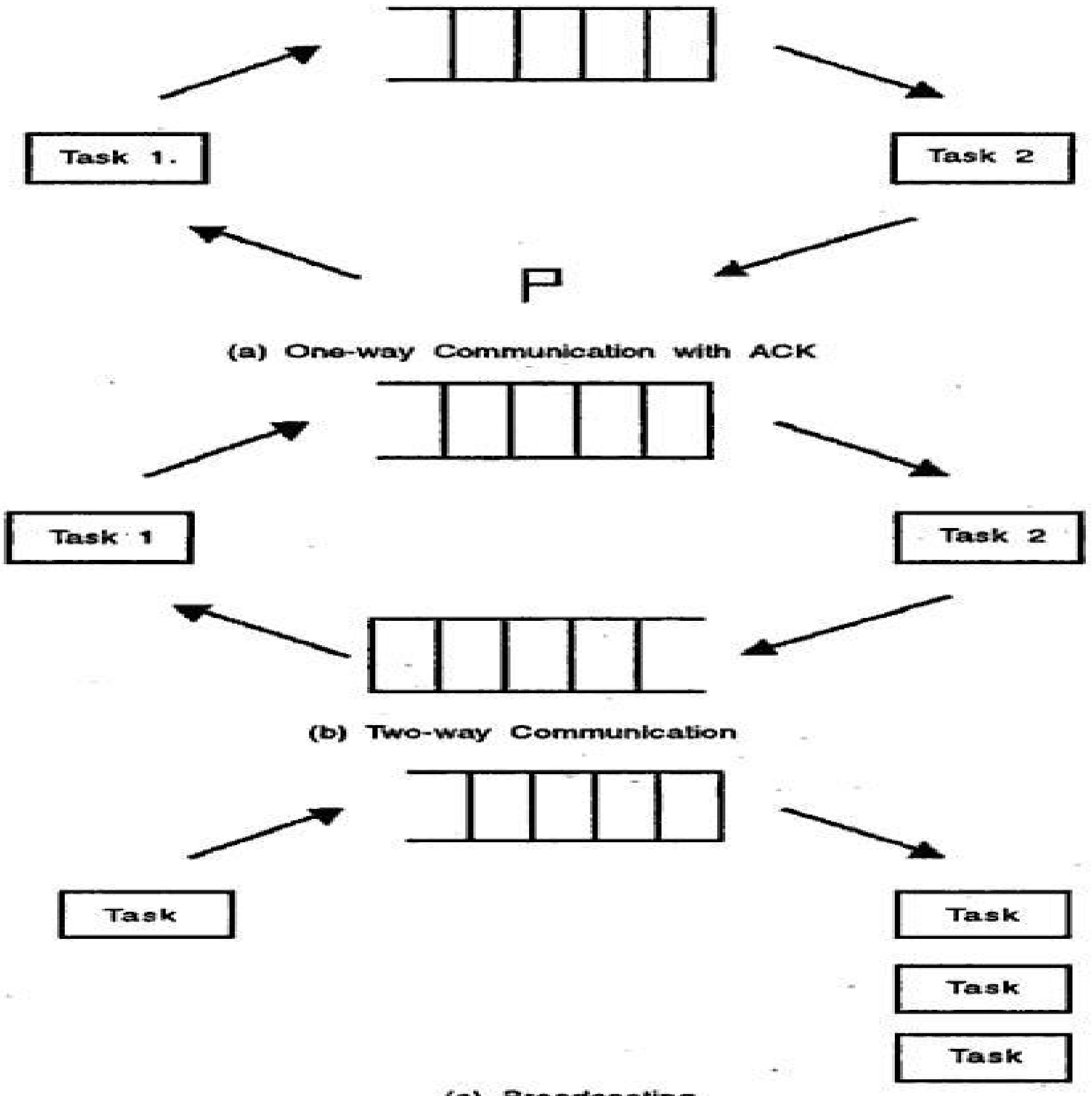


Fig. 7.13: Applications of Message Queues

MESSAGE QUEUES……

Message Queue Management Function Calls:

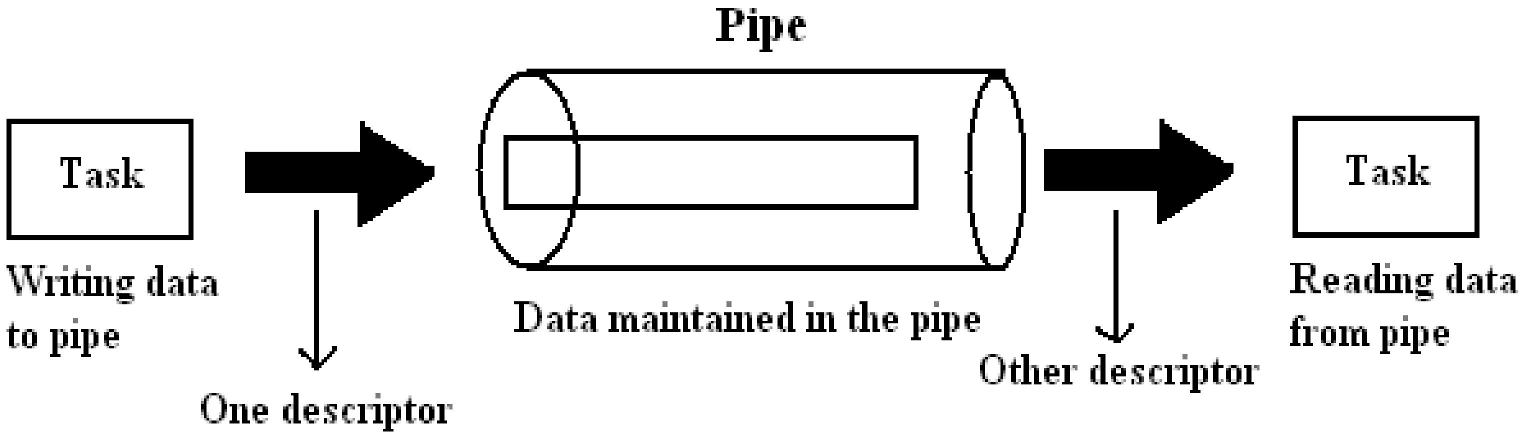
The following function calls are provided to manage message queues

- Create a queue
- Delete a queue
- Flush a queue
- Post a message in queue
- Post a message in front of queue
- Read message from queue
- Broadcast a message
- Show queue information
- Show queue waiting list.

PIPES

Pipes are kernel objects that are used to exchange unstructured data and facilitate synchronization among tasks.

In a traditional implementation, a pipe is a unidirectional data exchange facility, as shown in below Figure.



Two descriptors, one for each end of the pipe (one end for reading and one for writing), are returned when the pipe is created. Data is written via one descriptor and read via the other. The data remains in the pipe as an unstructured byte stream.

PIPES…..

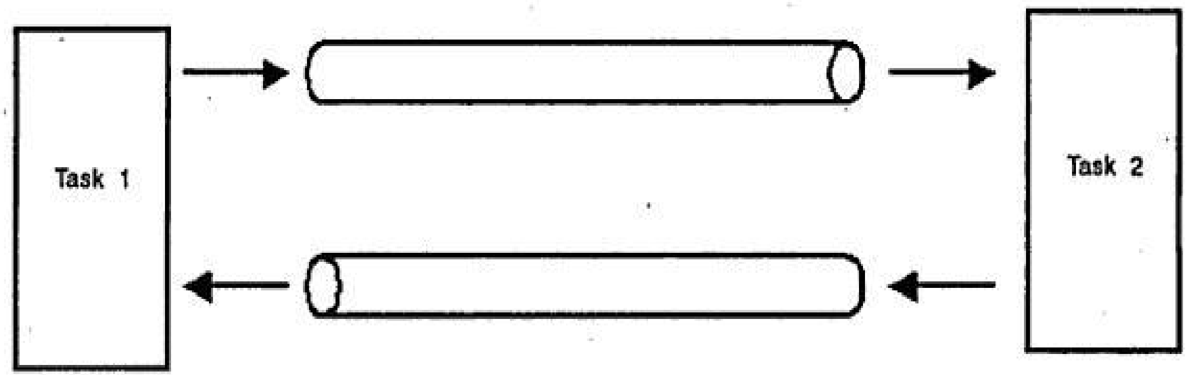
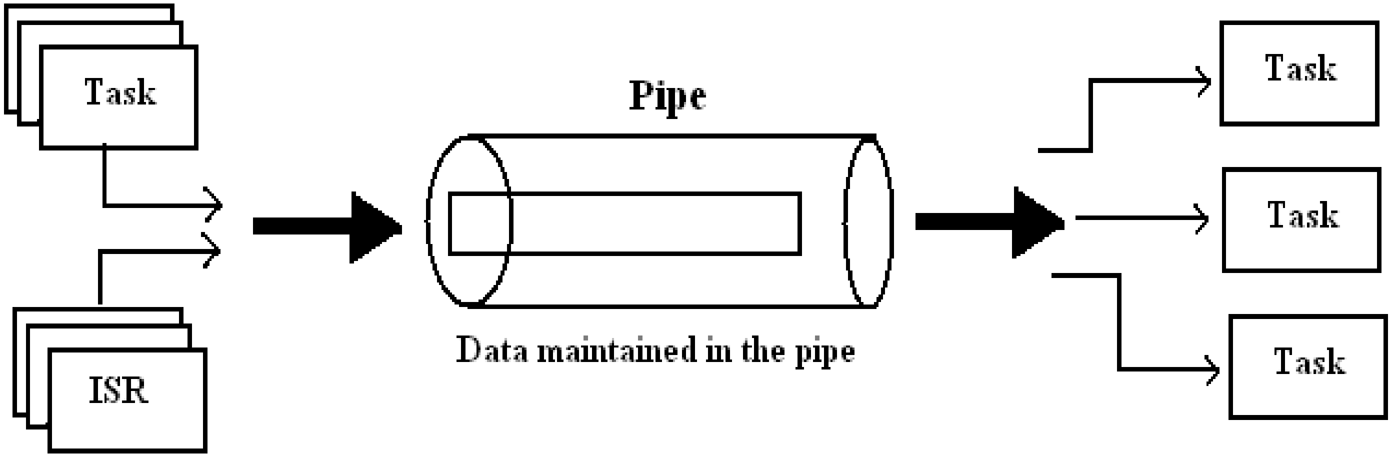


Fig. 7.16 Pipes for Inter-task Communication

Data is read from the pipe in FIFO order. A pipe provides a simple data flow facility so that the reader becomes blocked when the pipe is empty, and the writer becomes blocked when the pipe is full.

Typically, a pipe is used to exchange data between a data-producing task and a data-consuming task, as shown in the below Figure. It is also permissible to have several writers for the pipe with multiple readers on it.



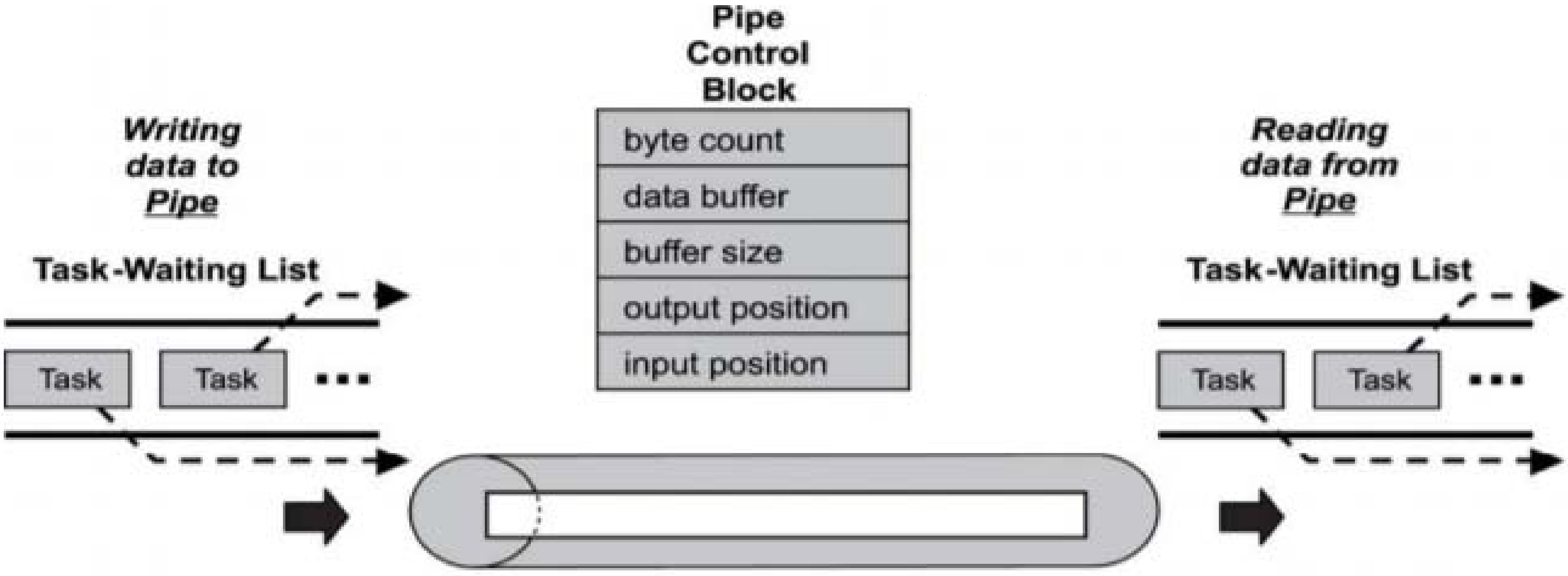
PIPES…..

Pipe versus Message Queue

- A pipe does not store multiple messages,
- Data in a pipe cannot be prioritized; the data flow is strictly first-in, first-out FIFO,
- Pipes support the powerful select operation and message queues do not.

Pipe Control Blocks

- The kernel creates and maintains pipe-specific information in an internal data structure called a pipe control block



PIPES…..

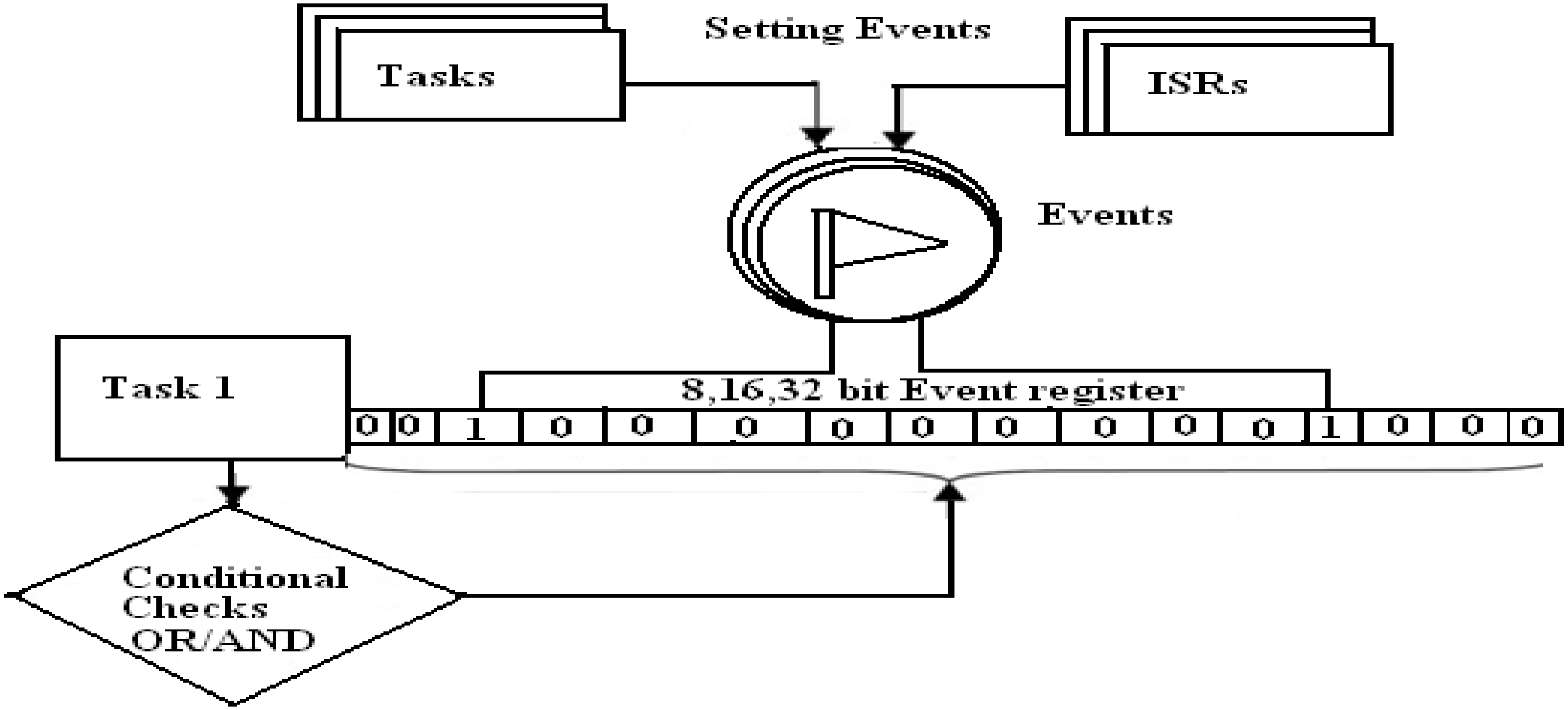
Pipes Management Function Calls:

The function calls in the OS API to manage the pipes are:

- Create a pipe
- Open a pipe
- Close a pipe
- Read from the pipe
- Write to the pipe

EVENT REGISTERS

Some kernels provide a special register as part of each tasks control block. This register, called an event register. It consists of a group of binary event flags used to track the occurrence of specific events. Depending on a given kernel's implementation of this mechanism, an event register can be 8 or 16 or 32 bits wide, may be even more.



EVENT REGISTERS…..

Each bit in the event register treated like a binary flag and can be either set or cleared. Through the event register, a task can check for the presence of particular events that can control its execution. An external source, such as a task or an ISR, can set bits in the event register to inform the task that a particular event has occurred.

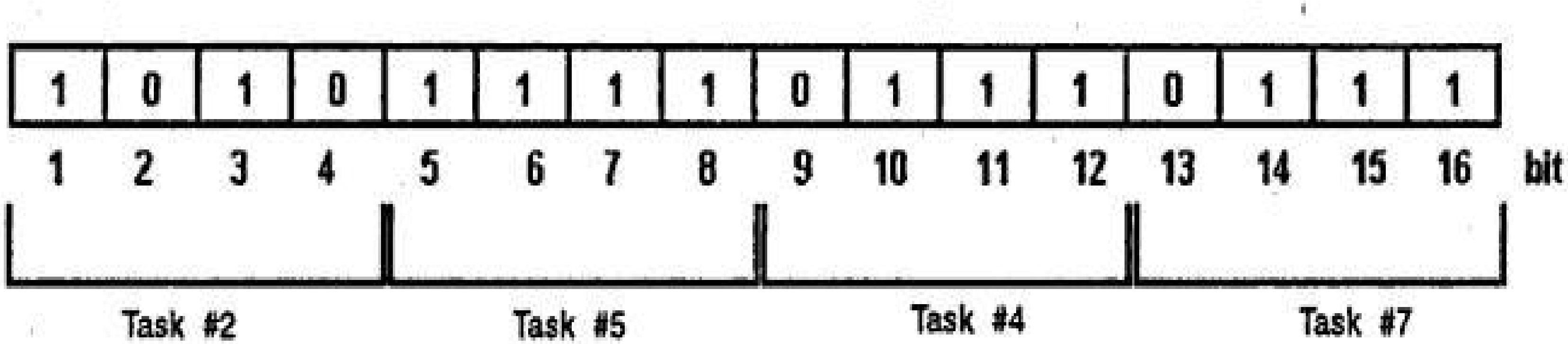


Fig. 7.14: 16-bit Event Register

Event Register ……

Typical Event Register Operations

1. **Send** →The send operation allows an external source, either a task or an ISR, to send events to another task.

2. **Receive** → The receive operation allows the calling task to receive events from external sources

Event registers are typically used for unidirectional activity synchronization.

Event Register ……

Event Registers Management Function Calls:

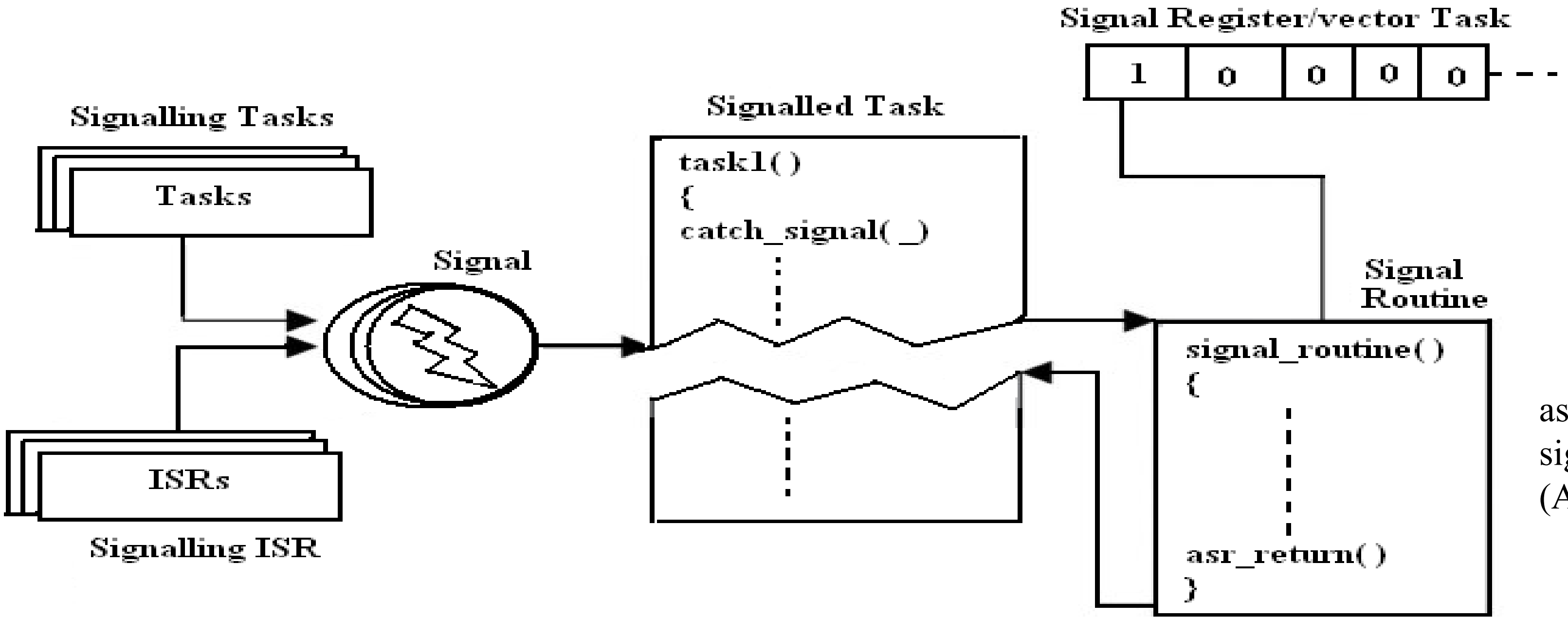
For managing the event registers, the following function calls are provided:

- Create an event register
- Delete an event register
- Query an event register
- Set an event register
- Clear an event flag

SIGNALS

A signal is an event indicator. It is a software interrupt that is generated when an event occurs. It diverts the signal receiver from its normal execution path and triggers the associated asynchronous processing. Mainly the, signals notify tasks of events that occurred during the execution of other tasks or ISRs.

The difference between a signal and a normal interrupt is that signals are so-called software interrupts, which are generated via the execution of some software within the system. By contrast, normal interrupts are usually generated by the arrival of an interrupt signal on one of the CPU's external pins. They are not generated by software within the system but by external devices.



asynchronous signal routine (ASR)

Signal……

The number and type of signals defined is both system-dependent and RTOS-dependent. An easy way to understand signals is to remember that each signal is associated with an event. The event can be either **unintentional**, such as an illegal instruction encountered during program execution, or the event may be **intentional**, such as a notification to one task from another that it is about to terminate. While a task can specify the particular actions to undertake when a signal arrives, the task has no control over when it receives signals. Consequently, the signal arrivals often appear quite random.

When a signal arrives, the task is diverted from its normal execution path, and the corresponding signal routine is invoked. The terms signal routine, signal handler, asynchronous event handler, and asynchronous signal routine are inter-changeable. Each signal is identified by an integer value, which is the signal number or vector number.

Signal……

Signal Management Function Calls:

The function calls to manage a signal are

- Install a signal handler (catch)
- Remove an installed signal handler (release)
- Send a signal to another task (send)
- Ignore a signal (ignore)
- Block a signal from being delivered (block)
- Unblock a blocked signal (unblock)

TIMERS

A timer is the scheduling of an event according to a predefined time value in the future, like setting an alarm clock.

For instance, the kernel has to keep track of different times

- A particular task may need to be executed periodically, say, every 10ms. A timer is used to keep track of this periodicity.
- A task may be waiting in a queue for an event to occur. If the event does not occur for a specified time, it has to take appropriate action.
- A task may be waiting in a queue for a shared resource. If the resource is not available for a specified time, an appropriate action has to be taken.

Timers……

- Most RTOSs implement a function for time delay

- Most time delay functions have a resolution related to the length of the system tick

- Most RTOSs allow the programmer to set a time limit on the wait for semaphores, queues, and mailboxes

- These functions are dangerous and should be used with extreme caution

- Most RTOSs allow the programmer to specify that a function should be called after a certain number of system ticks

- ✓ One tick = 1ms: your system can run 50 days
- ✓ One tick = 20ms: your system can run 1000 days = 2.5 years
- ✓ One tick = 50ms: your system can run 2500 days= 7 years

Timers……

Timers Management Function Calls:

The following function calls are provided to manage the timer:

- Get time
- Set time
- Time delay (in system clock ticks)
- Time delay (in seconds)
- Reset timer

MEMORY MANAGEMENT

It is a service provided by a kernel which allots the memory needed, either static or dynamic for various processes. The manager optimizes the memory needs and memory utilization. The memory manager allocates memory to the processes and manages it with appropriate protection. There may be static and dynamic allocations of memory. The manager optimizes the memory needs and memory utilization. An RTOS may disable the support to the dynamic block allocation, MMU support to the dynamic page allocation and dynamic binding as this increases the latency of servicing the tasks and ISRs.

Hence, the two instructions “Malloc” and “free”, although available in C language, are not used by the embedded engineer, because of the latency problem.

So, an RTOS may or may not support memory protection in order to reduce the latency and memory needs of the processes.

Memory Management ……

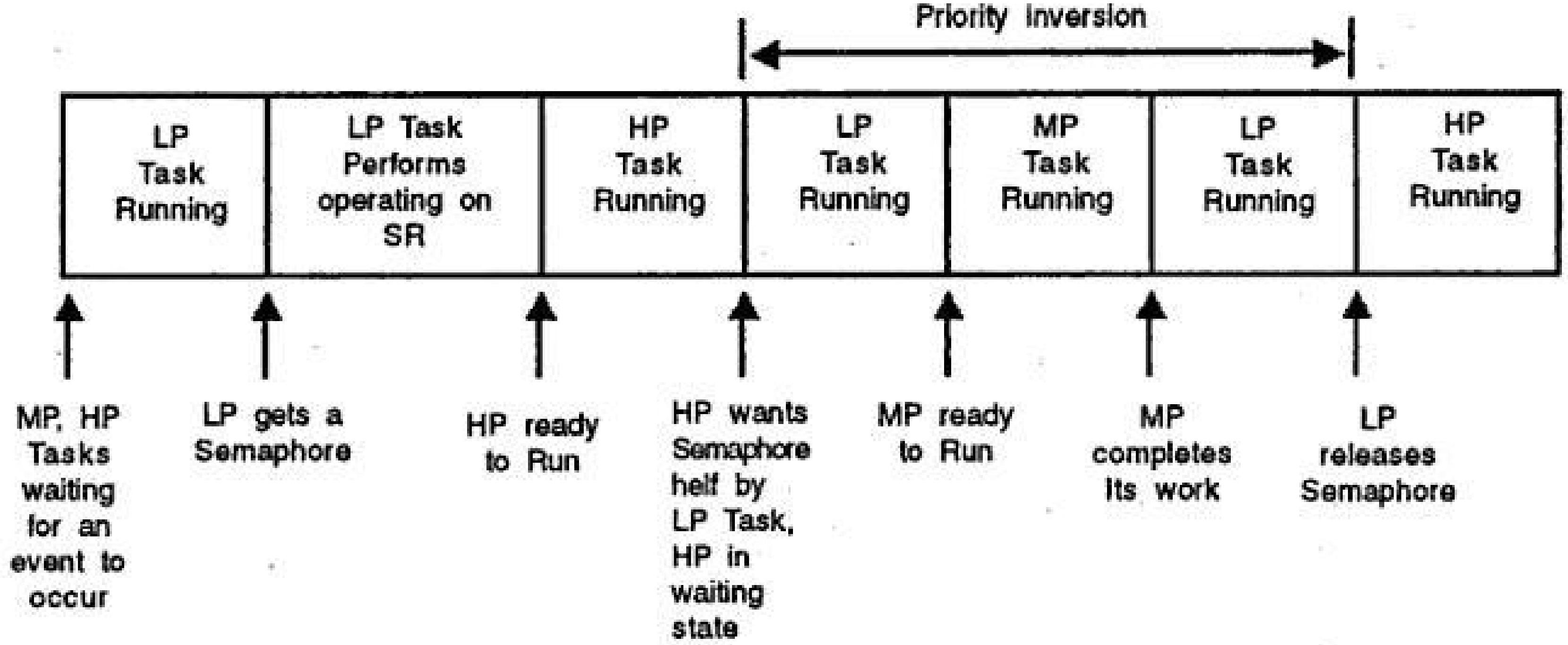
Memory Management Function Calls:

The API provides the following function calls to manage memory

- Create a memory block
- Get data from memory
- Post data in the memory
- Query a memory block
- Free the memory block.

PRIORITY INVERSION PROBLEM

In any real time embedded system, if a high priority task is blocked or waiting and a low priority task is running or under execution, this situation is called Priority Inversion. This priority Inversion is shown in the diagram below.



In Scheduling, priority inversion is the scenario where a low priority Task holds a shared resource that is required by a high priority task. This causes the execution of the high priority task to be blocked until the low priority task releases the resource, effectively “inverting” the relative priorities of the two tasks.

Priority Inversion Problem….

The consequences of the priority Inversion are

(i) Reduce the performance of the system

(ii) May reduce the system responsiveness which leads to the violation of response time guarantees

(iii) Create problems in real-time systems.

There are two types of priority inversions. (i) Bounded and (ii). Unbounded.

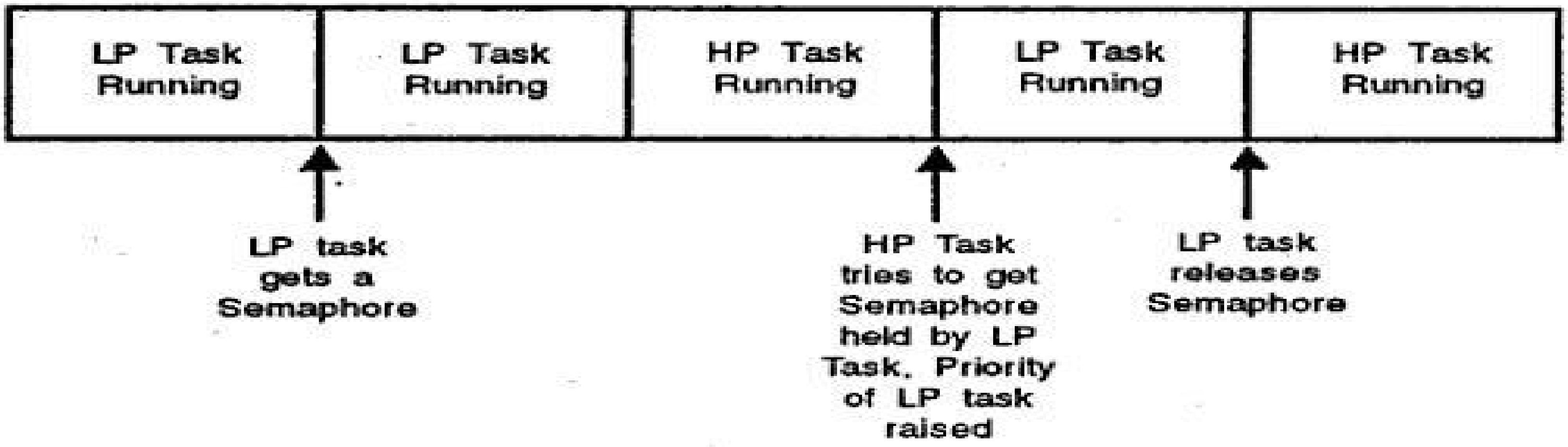
For example let us consider two tasks TA and TB in a real time system. Task A has higher priority than Task B. Initially Task A is under execution. But Task A is blocked after some time due to interruption and Task B is scheduled next. The Task B acquires a mutex corresponding to a resource common to both Task A and Task B. After some time Task A acquire mutex before the completion of Task B. But Task A cannot acquire mutex and it is blocked because already Task B has acquired the mutex. So, the Task A, though has the higher priority, is blocked until Task B releases the mutex for the resource. This is called the bounded Priority inversion.
If the time over which the higher priority is blocked is unknown, then it is called unbounded priority inversion.

Priority Inversion Problem….

The Priority Inversion is avoided by using two protocols, namely
(i) Priority Inheritance Protocol (PIP)
(ii) Priority Ceiling Protocol (PCP).

The Priority Inheritance Protocol is a resource access control protocol that raises the priority of a task, if that task holds a resource being requested by a higher priority task, to the same priority level as the higher priority task.

The priority ceiling protocol is a synchronization protocol for shared resources to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections. In this protocol each resource is assigned a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource.


LP Task Running
LP Task Running
HP Task Running
LP Task Running
HP Task Running
LP task gets a Semaphore
HP Task tries to get Semaphore held by LP Task. Priority of LP task raised
LP task releases Semaphore

REFERENCE BOOKS

- **Marilyn Wolf, "Computers as Components - Principles of Embedded Computing System Design", Third Edition "Morgan Kaufmann Publisher (An imprint from Elsevier), 2012.**
- **K.V.K.K. Prasad, "Embedded Real-Time Systems: Concepts, Design & Programming", Dream Tech Press, 2005.**
- **Raj Kamal. Embedded Systems Architecture, Programming and Design. 2nd Edition, McGraw Hill, 2012.**
- **Arnold S. Berger, An introduction to Processes, Tools and Techniques, CMP books, 2005.**
- **Wang K.C., Embedded and Real-Time Operating Systems, Springer, 2017.**
- **Frank Vahid and Tony Givargis, Embedded System Design: A Unified Hardware/Software Introduction, John Wiley & Sons, Student edition, 2006.**

?

Thank You