Embedded System

Real Time Operating System (RTOS)

Real Time Operating System Concept

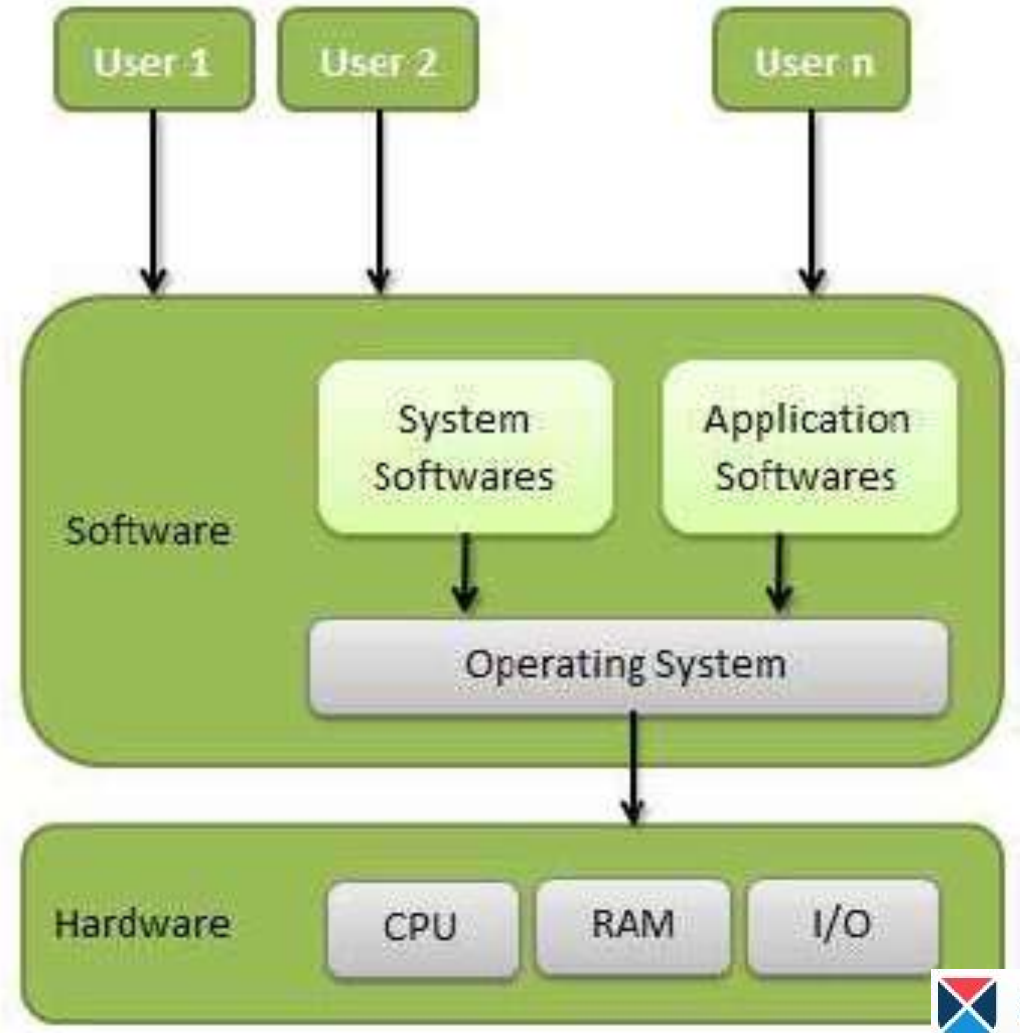
Architecture of kernel
Task scheduler
ISR
Semaphores
Mailbox
Message queues
Pipes
Events
Timers
Memory management

Introduction to Ucos II RTOS
Study of kernel structure of Ucos II
Synchronization in Ucos II
Inter-task communication in Ucos II
Memory management in Ucos II
Porting of RTOS.

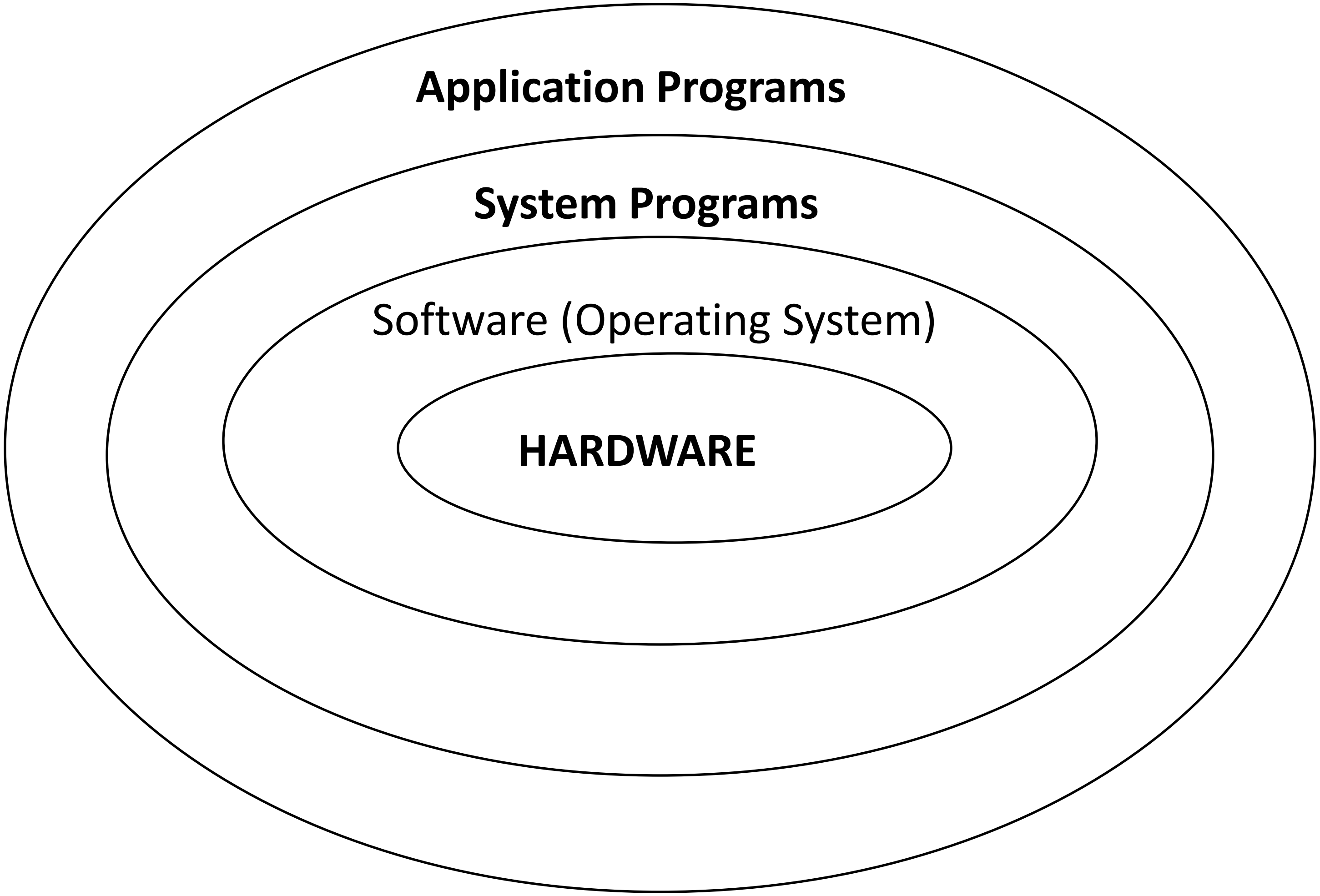
RTOS services in contrast with traditional OS

What is Operating System ?

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



Structure of Operating System:



Structure of Operating System:

The structure of OS consists of 4 layers:

1. **Hardware**

   Hardware consists of CPU, Main memory, I/O Devices, etc,

2. **Software (Operating System)**

   Software includes process management routines, memory management routines, I/O control routines, file management routines.

3. **System programs**

   This layer consists of compilers, Assemblers, linker etc.

4. **Application programs**

   This is dependent on users need.

Ex. Railway reservation system, Bank database manageme

Goals / Function of Operating system

Following are some of important functions of an operating System.

- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.
- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address. Main memory provides a fast storage that can be access directly by the CPU. So for a program to be executed, it must in the main memory.

Operating System does the following activities for memory management.

•Keeps tracks of primary memory i.e. what part of it are in use by whom, what part are not in use.

•In multiprogramming, OS decides which process will get memory when and how much.

•Allocates the memory when the process requests it to do so.

•De-allocates the memory when the process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, OS decides which process gets the processor when and how much time. This function is called process scheduling.

Operating System does the following activities for processor management.

- Keeps tracks of processor and status of process.
  Program responsible for this task is known as traffic controller.

- Allocates the processor (CPU) to a process.

- De-allocates processor when processor is no longer required.

Device Management

OS manages device communication via their respective drivers.

Operating System does the following activities for device management.

- Keeps tracks of all devices.
  Program responsible for this task is known as the I/O controller.

- Decides which process gets the device when and for how much time.

- Allocates the device in the efficient way.

- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

Operating System does the following activities for file management.

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Other Important Activities

Following are some of the important activities that Operating System does.

•**Security --** By means of password and similar other techniques, preventing unauthorized access to programs and data.

•**Control over system performance --** Recording delays between request for a service and response from the system.

•**Job accounting --** Keeping track of time and resources used by various jobs and users.

•**Error detecting aids --** Production of dumps, traces, error messages and other debugging and error detecting aids.

•**Coordination between other software and users --** Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Why Operating system is needed?

- ➢To run one single program is easy
- ➢To run Two, or three program is called fine handling,
- ➢but more than five would be difficult
- ➢you need to take care
  - ✓memory areas
  - ✓Program counters
  - ✓Scheduling, run one instruction each?
  - ✓....
  - ✓Communication/synchronization
  - ✓Device drivers

➢OS is nothing but a program offering the functions needed in all applications

Operating System handles

- Memory Addressing & Management
- Interrupt & Exception Handling
- Process & Task Management
- File System
- Timing
- Process Scheduling & Synchronization

Examples of Operating Systems

- RTOS – Real-Time Operating System
- Single-user, Single-task: example PalmOS
- Single-user, Multi-task: MS Windows and MacOS
- Multi-user, Multi-task: UNIX

What OS provides?

Kernel with Hardware Abstraction Layer + I/O Management	Hardware Abstraction Application developers need not know about the underlying hardware
Scheduling	Proper Scheduling ensures an efficient usage of resources
Process Management	User Programs can be protected from each other
Inter Process Communication	Application can communicate with each other with ease
File Management	Faster Storage and Retrieval of Data
POSIX and other compliance Subsystem	Portability

"Portable Operating System Interface for uni-X"

Types of Operating Systems

Types of Operating Systems

1. Batch operating system

2. Time-sharing operating systems

3. Distributed operating System

4. Network operating System

5. Real Time operating System

Batch operating system

1. The users of batch operating system do not interact with the computer directly.
2. Each user prepares his job on an off-line device like punch cards and submits it to the computer operator.
3. To speed up processing, jobs with similar needs are batched together and run as a group. Thus, the programmers left their programs with the operator.
4. The operator then sorts programs into batches with similar requirements.

The problems with Batch Systems are following

1. Lack of interaction between the user and job.
2. CPU is often idle, because the speeds of the mechanical I/O devices are slower than CPU.
3. Difficult to provide the desired priority.

Time-sharing operating systems

1. Time sharing is a technique which enables many people, located at various terminals, to use a particular computer system at the same time.

2. Time-sharing or multitasking is a logical extension of multiprogramming.

3. Processor's time which is shared among multiple users simultaneously is termed as time-sharing.

4. The main difference between Multiprogrammed Batch Systems and Time-Sharing Systems is that in case of multiprogrammed batch systems, objective is to maximize processor use, whereas in Time-Sharing Systems objective is to minimize response time.

1. Multiple jobs are executed by the CPU by switching between them, but the switches occur so frequently. Thus, the user can receive an immediate response.
2. For example, in a transaction processing, processor execute each user program in a short burst or quantum of computation.

3. That is if n users are present, each user can get time quantum. When the user submits the command, the response time is in few seconds at most.

4. Operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time.

5. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.

Advantages of Timesharing operating systems

1. Provide advantage of quick response.
2. Avoids duplication of software.
3. Reduces CPU idle time.

Disadvantages of Timesharing operating systems

1. Problem of reliability.
2. Question of security and integrity of user programs and data.
3. Problem of data communication.

Distributed operating System

1. Distributed systems use **multiple central processors** to serve multiple real time application and multiple users.

2. Data processing jobs are distributed among the processors accordingly to which one can perform each job most efficiently.

3. The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines).

4. These are referred as loosely coupled systems or distributed systems.

5. Processors in a distributed system may vary in size and function.

6. These processors are referred as sites, nodes, and computers and so on.

The advantages of distributed systems

1. With resource sharing facility user at one site may be able to use the resources available at another.

2. Speedup the exchange of data with one another via electronic mail.

3. If one site fails in a distributed system, the remaining sites can potentially continue operating.

4. Better service to the customers.

5. Reduction of the load on the host computer.

6. Reduction of delays in data processing.

Network operating System

1. Network Operating System runs on a **server** and and provides server the capability to manage data, users, groups, security, applications, and other networking functions.

2. The primary purpose of the network operating system is to allow **shared file and printer access** among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.

Examples of network operating systems are

1. Microsoft Windows Server 2003
2. Microsoft Windows Server 2008
3. UNIX
4. Linux
5. Mac OS X
6. Novell NetWare, and BSD.

The advantages of network operating systems

1. Centralized servers are highly stable.
2. Security is server managed.
3. Upgrades to new technologies and hardware can be easily integrated into the system.
4. Remote access to servers is possible from different locations and types of systems.

The disadvantages of network operating systems

1. High cost of buying and running a server.
2. Dependency on a central location for most operations.
3. Regular maintenance and updates are required.

Real Time operating System

Those systems in which the **correctness** of the system depends **not only** on the **logical result** of computation, **but also** on the **time** at which the results are produced.

A real time system is a system/that ensures the exact time requirements for a job.

1. Real time system is defines as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment.

2. Real time processing is always on line whereas on line system need not be real time.

3. The time taken by the system to respond to an input and display of required updated information is termed as response time. So in this method response time is very less as compared to the online processing.

Uses of Real Time operating System

Real-time systems are used

1. when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application.
2. Real-time operating system has well-defined, fixed time constraints otherwise system will fail.

For example

1. Scientific Experiments
2. Medical Imaging Systems
3. Industrial Control Systems
4. Weapon Systems
5. Robots
6. Home-appliance Controllers
7. Air Traffic Control System Etc.
8. Industrial control systems
9. Food processing
10. Engine Controls

FAX machines
Copiers
Printers
Scanners
Routers
Robots

Compare Desktop and Real Time operating System

S N	Desktop	RTOS
1	The operating system takes control of the machine as soon as it is turned on and then lets you to start your applications. You compile and link your applications separately from the operating system.	You usually link your application and RTOS. At boot-up time your application usually gets control first, and then starts the RTOS.
2	Usually strong memory control and no recovery in the emergency case	RTOS usually do not control the memory but the whole system must recover anyway
3	Standard configuration	Sharp configuration possibilities and also the possibility to choose the limited number of services because of the limit in the memory usage

Operating systems come in two flavors, real-time versus non real-time

The difference between the two is characterized by the consequences which result if functional correctness and timing parameters are not met in the case of real-time operating systems

1. The main difference between them is the task manager, which is composed by the Dispatcher and the Scheduler.

2. Like a non-RTOS, a RTOS provides support for multi-tasking with multiple threads and inter-task communication and synchronization mechanisms such as semaphores, shared memory, pipes, mailboxes, etc…

3. In RTOS synchronization is of an even greater importance in order to avoid blocking of shared resources and to guarantee that the tasks are preformed in the correct order when necessary.

Characteristics of a Embedded Real Time system:

- ❖Deadline driven.
- ❖Process the events in a deterministic time.
- ❖Have limited resources.

Soft versus Hard Real-Time

In a *soft real-time system, tasks are* completed as fast as possible without having to be completed within a specified timeframe.

In a *hard real-time operating system* however, not only must tasks be completed within the specified timeframe, but they must also be completed correctly.

Hard real time

1. Hard real time means strict about adherence to each task deadline.

2. When an event occurs, it should be serviced within the predictable time at all times in a given hard real time system.

3. The preemption period for the hard real time task in worst case should be less than a Few micro seconds.

4. A hard RT RTOS is one, which has predictable performance with no deadline miss, even in case of sporadic tasks (sudden bursts of occurrence of events requiring attention).

5. Automobile engine control system and antilock brake are the examples of hard real time systems

Hard real time system design

1. Disabling of all other interrupts of lower priority when running the hard real time tasks

2. Preemption of higher priority task by lower priority tasks

3. Some critical code in assembly to meet the real time constraint (deadline) fast

4. Task running in kernel space, [This saves the time required to first check whether access is outside the memory space allocated to the kernel functions.]

5. Provision of asynchronous iOS (iPhone OS)

6. Provision of locks or spin locks

1. Predictions of interrupt latencies and context switching latencies of the tasks

2. Predictability is achieved by writing all functions which execute always take the same predefined time intervals in case of varying rates of occurrences of the events.

3. Response in all the time slots for the given events in the system and thus providing the guaranteed task deadlines even in case of sporadic and aperiodic tasks.

4. Sporadic tasks means tasks executed on the sudden-bursts of the corresponding events at high rates, and

5. Aperiodic tasks mean task having no definite period of event occurrence.

Example of hard real time system

Video transmission, each picture frame and audio must be transferred at fixed rate

Soft real time system

1. One in which deadlines are mostly met.

2. Soft real time means that only the precedence and sequence for the task operations are defined, interrupt latencies and context switching latencies are small but there can be few deviations between expected latencies of the tasks and observed time constraints and a few deadline misses are accepted

Soft real time task

1. The preemption period for the soft real time task in worst case may be about a few ms.

2. Mobile phone, digital cameras and orchestra playing robots are examples of soft real time systems.

www.getmyuni.com

OPEN SOURCE RTOS / Embedded OS

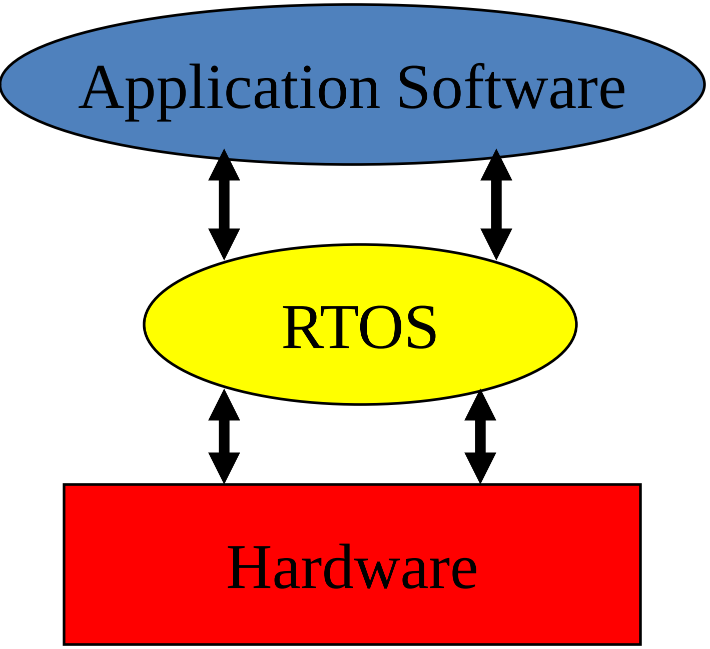
NAME RTOS	DESRIPTION
Linux	This is the primary site for the Linux kernel source
eCos	eCos is provided as an open source runtime system supported by the GNU open source development tools. Currently eCos supports ten different target architectures (ARM, Hitachi H8300, Intel x86, MIPS, Matsushita AM3x, Motorola 68k, PowerPC, SuperH, SPARC and NEC V8xx) including many of the popular variants of these architectures and evaluation boards.
uClinux	MMU Less version of Linux. Support ARM, Coldfire etc processor.
FreeRTOS	FreeRTOS.org TM is a portable, open source, mini Real Time Kernel - a free to download and royalty free RTOS that can be used in commercial applications Support 17 architectures. ARM, AVR32, MSP430 etc.
RTAI	The RealTime Application Interface for Linux. It support following architecture … x86, PowerPC, ARM
coscox	An embedded real-time multi-task OS specially for ARM Cortex M4, M3 and M0 chipset.

PROPERITARY RTOS / Embedded OS

NAME RTOS	DESRIPTION
QNX	QNX is real time RTOS, which support ARM, MIPS, PowerPC, SH and X86 Processor family.
VxWorks	Vxworks is also real time rtos, it support wide range of processor architectures like ARM, PowerPC, ColdFire, MIPS etc.
INTEGRITY	INTEGRITY is hard real time rtos. It support ARM, PowerPC, Coldfire,x86 etc processor architectures.
ThreadX	ThreadX is Express Logic's advanced Real-Time Operating System (RTOS) designed specifically for deeply embedded applications. It support ARM, PowerPC,X86 etc.
MicroC/OS2	µC/OS-II, The Real-Time Kernel is a highly portable, ROMable, very scalable, preemptive real-time, multitasking kernel (RTOS) for microprocessors and microcontrollers. It support ARM, PowerPC,X86 etc.
embOS	embOS is a priority-controlled multitasking system, designed to be used as an embedded operating system for the development of real time applications for a variety of microcontrollers.
SafeRTOS	Real Time OS for mission critical applications based on the FreeRTOS scheduling algorithm, certified for IEC61508 SIL3, FDA 510K or DO-178B.

RTOS Kernel

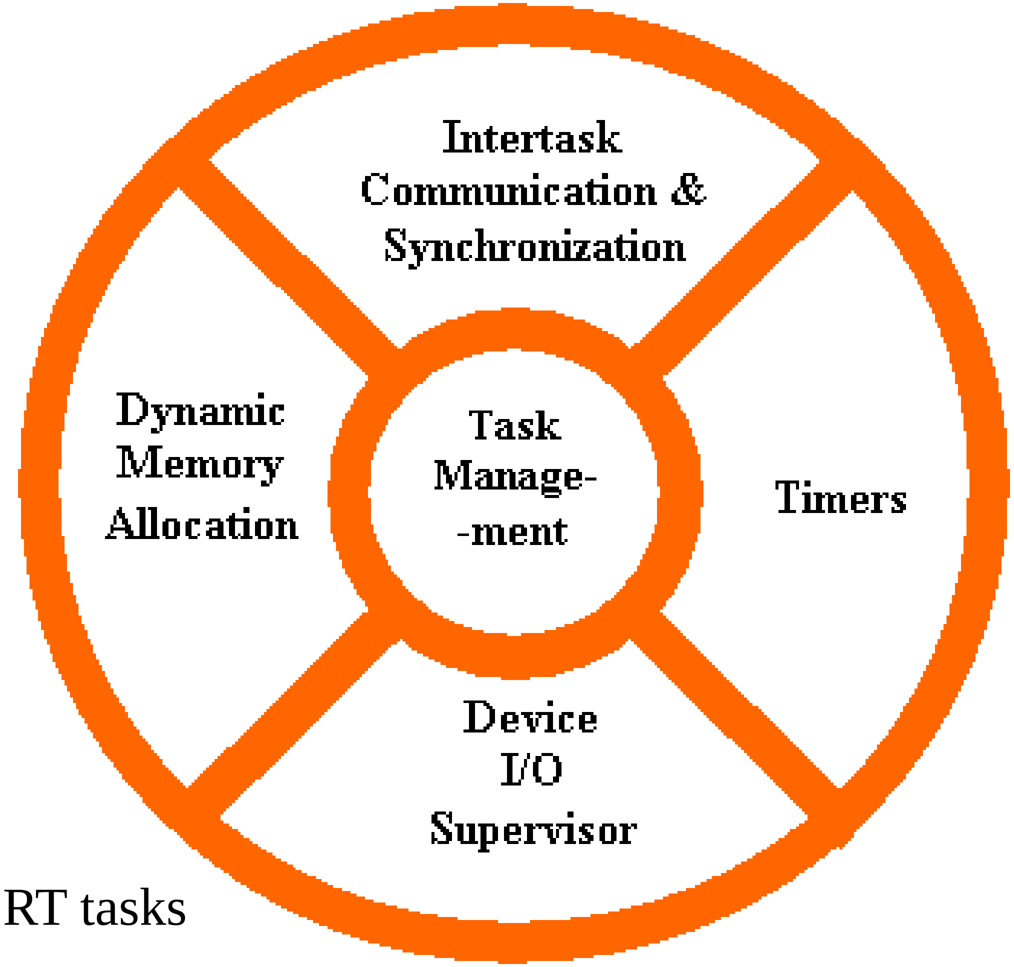
RTOS Kernel provides an Abstraction layer that hides from application software the hardware details of the processor / set of processors upon which the application software shall run.



1. The central component of most Operating systems is called Kernel.
2. The kernel manages system's resources and the *communication*
3. The kernel provides the most basic interface between the *computer* itself and the rest of the operating system.
4. The kernel is responsible for the management of the central processor.
5. The kernel includes the dispatcher to allocate the central processor, to determine the cause of an interrupt and initiate its processing, and some provision for *communication* among the various system and user tasks currently active in the system.
6. Kernel is the core of an operating

Basic functions of RTOS kernel

1. Task management
2. Task synchronization
   - Avoid priority inversion
3. Task scheduling
4. Interrupt handling
5. Memory management
   - no virtual memory for hard RT tasks
6. Exception handling (important)


Intertask Communication & Synchronization
Dynamic Memory Allocation
Task Manage- -ment
Timers
Device I/O Supervisor

Task

1. A task (also called a thread) is a program on a computer which can be executed and run.

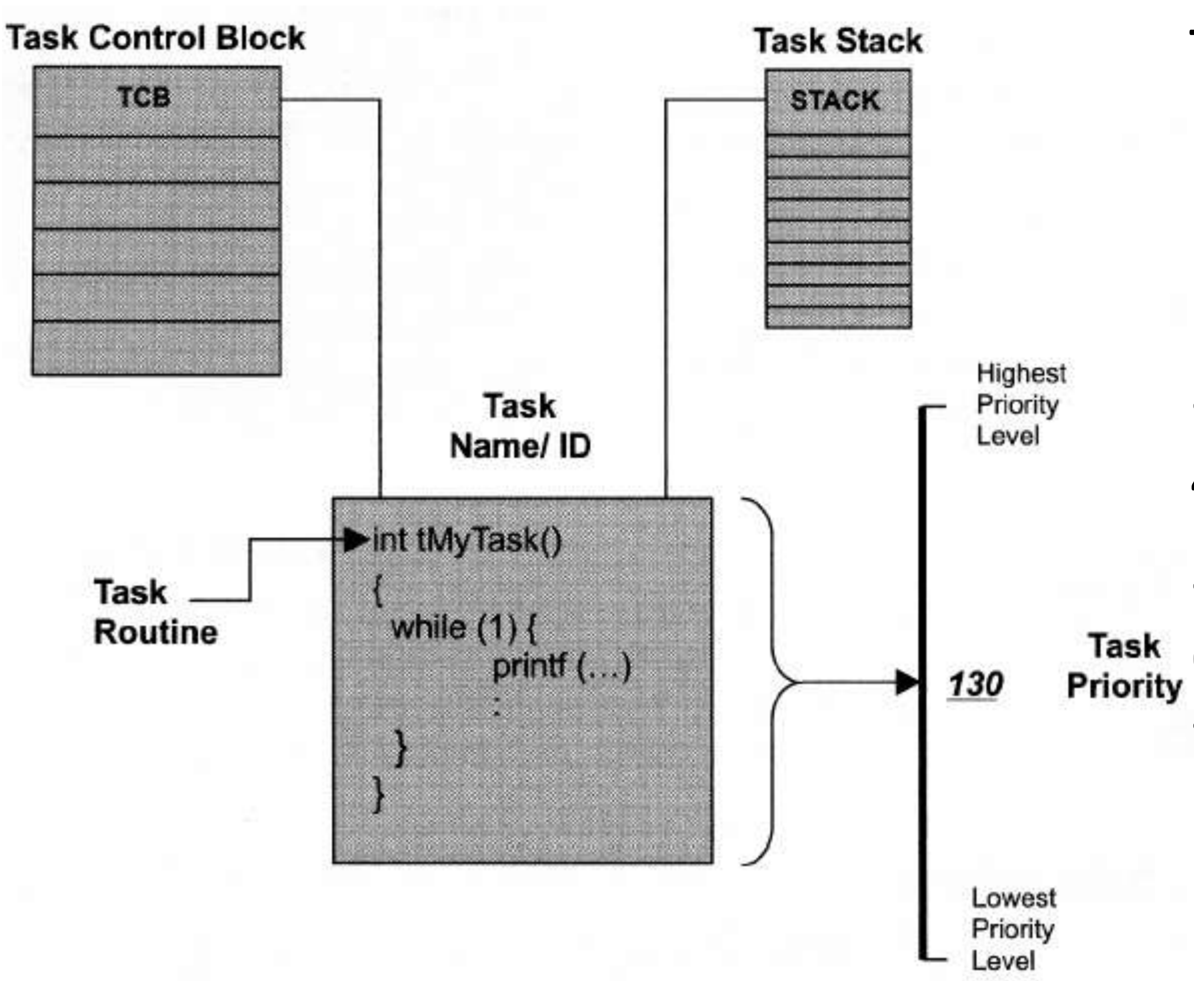
2. A task is an independent thread of execution that can compete with other concurrent tasks for processor execution time.

3. A task is schedulable.

4. The design process for a real-time application involves splitting the work to be done into tasks which are responsible for a portion of the problem.

5. Each task is assigned a priority, its own set of CPU registers, and its own stack area

A task can be in any of one of states



```
int tMyTask()
{
  while (1) {
        printf (...)
        :
  }
}
```

– Task States:

1. DORMANT
2. READY
3. RUNNING
4. DELAYED
5. PENDING
6. BLOCKED
7. INTERRUPTED

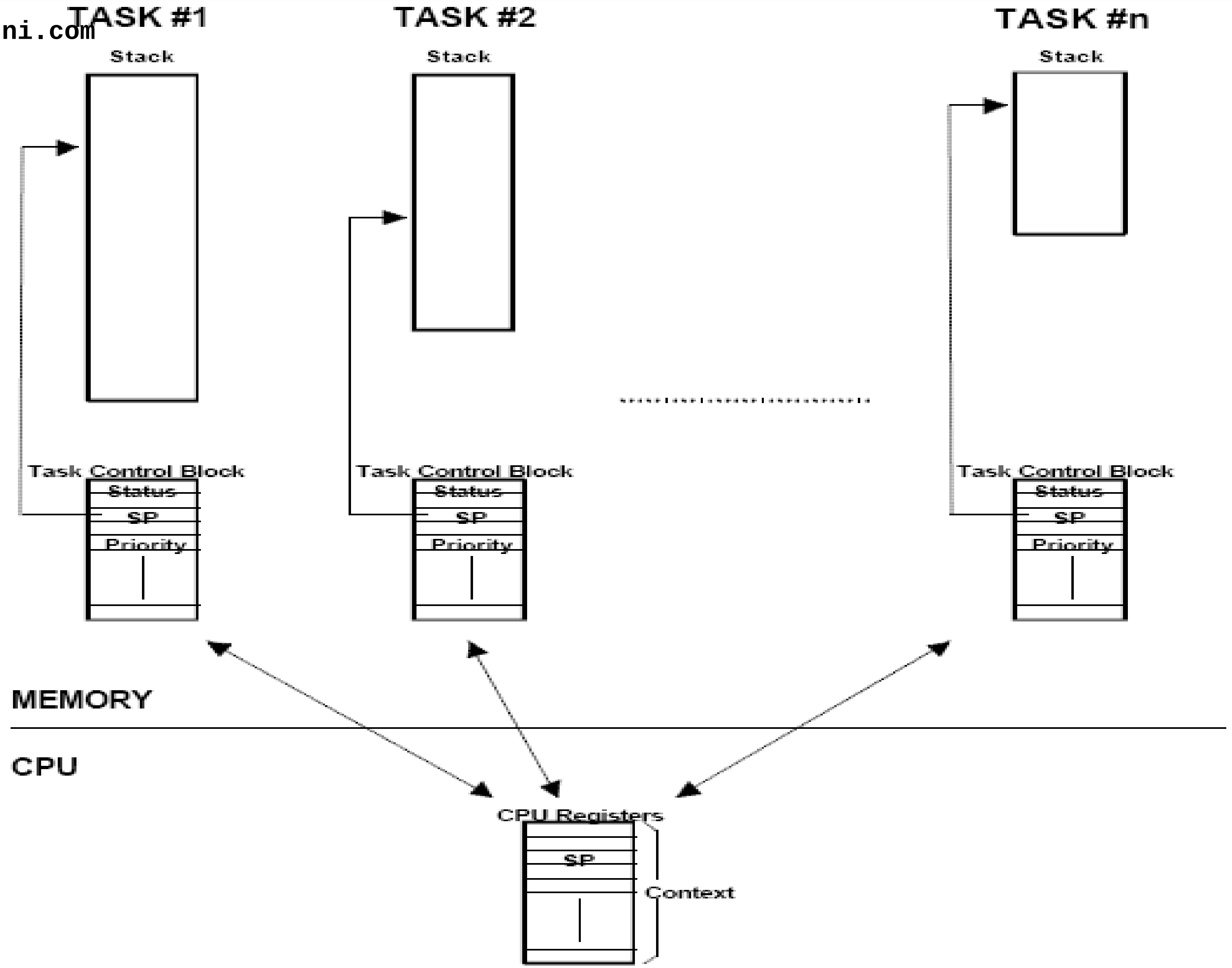


Figure 2-2, Multiple tasks.

DORMANT-The DORMANT state corresponds to a task that resides in memory but has not been made available to the multitasking kernel.

READY-A task is READY when it can be executed but its priority is less than that of the task currently being run.

In this state, the task actively competes with all other ready tasks for the processor’s execution time.

The kernel’s scheduler uses the priority of each task to determine which task to move to the running state.

RUNNING-A task is RUNNING when it has control of the CPU and it's currently being executed.

On a single-processor system, only one task can run at a time.

- When a task is preempted by a higher priority task, it moves to the ready state.
- It also can move to the blocked state.

–Making a call that requests an unavailable resource

–Making a call that requests to wait for an event to occur

–Making a call to delay the task for some duration

WAITING-A task is WAITING when it requires the occurrence of an event (waiting for an I/O operation to complete, a shared resource to be available, a timing pulse to occur, time to expire, etc.).

BLOCKED

CPU starvation occurs when higher priority tasks use all of the CPU execution time and lower priority tasks do not get to run.

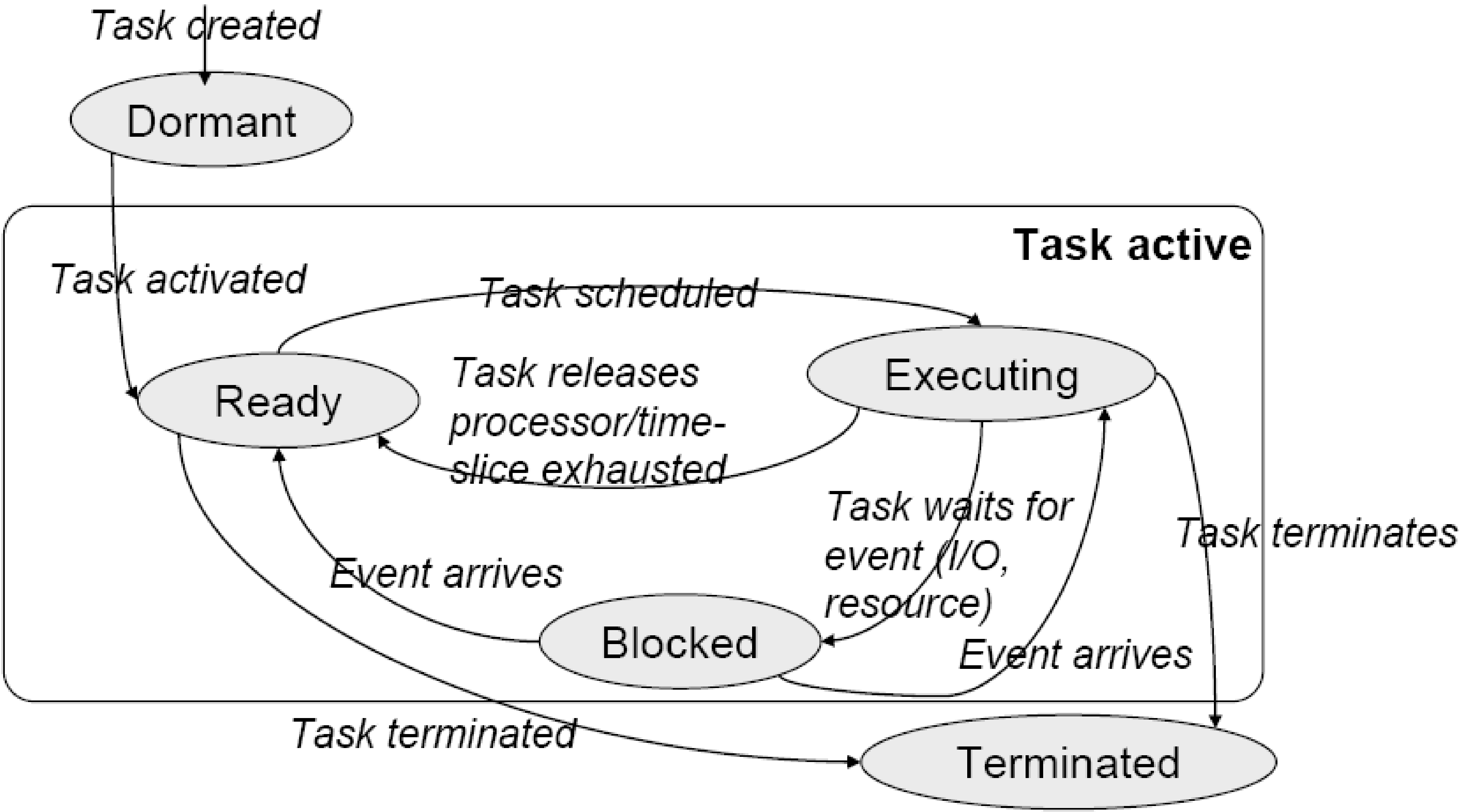
The cases when blocking conditions are met

–A semaphore token for which a task is waiting is released

–A message, on which the task is waiting, arrives in a message queue

–A time delay imposed on the task expires

ISR(Interrupt Service Routine)-A task is in ISR state when an interrupt has occurred and the CPU is in the process of servicing the interrupt.


Task created
Dormant
Task active
Task activated
Task scheduled
Ready
Task releases processor/time-slice exhausted
Executing
Task waits for event (I/O, resource)
Task terminates
Event arrives
Blocked
Event arrives
Task terminated
Terminated

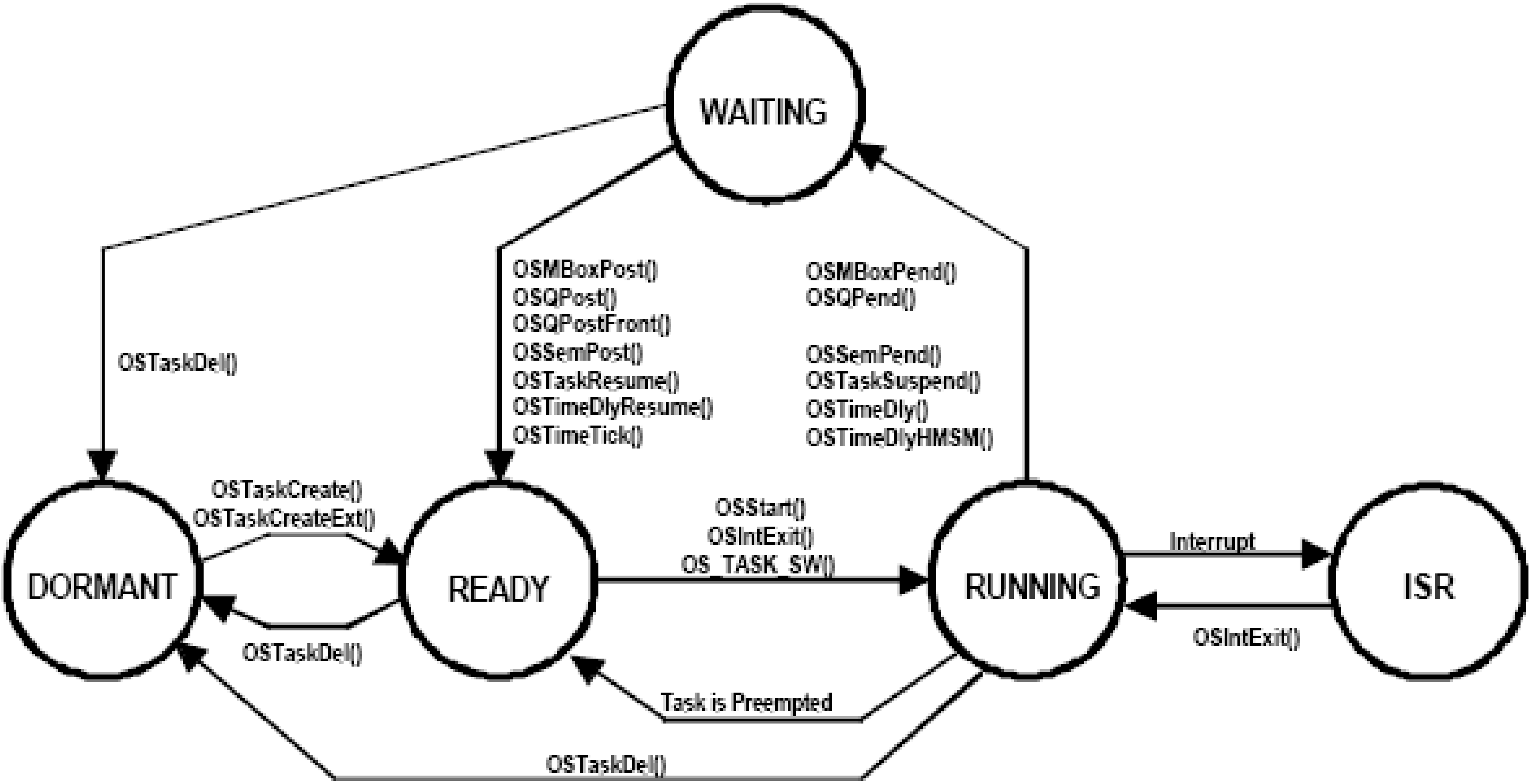
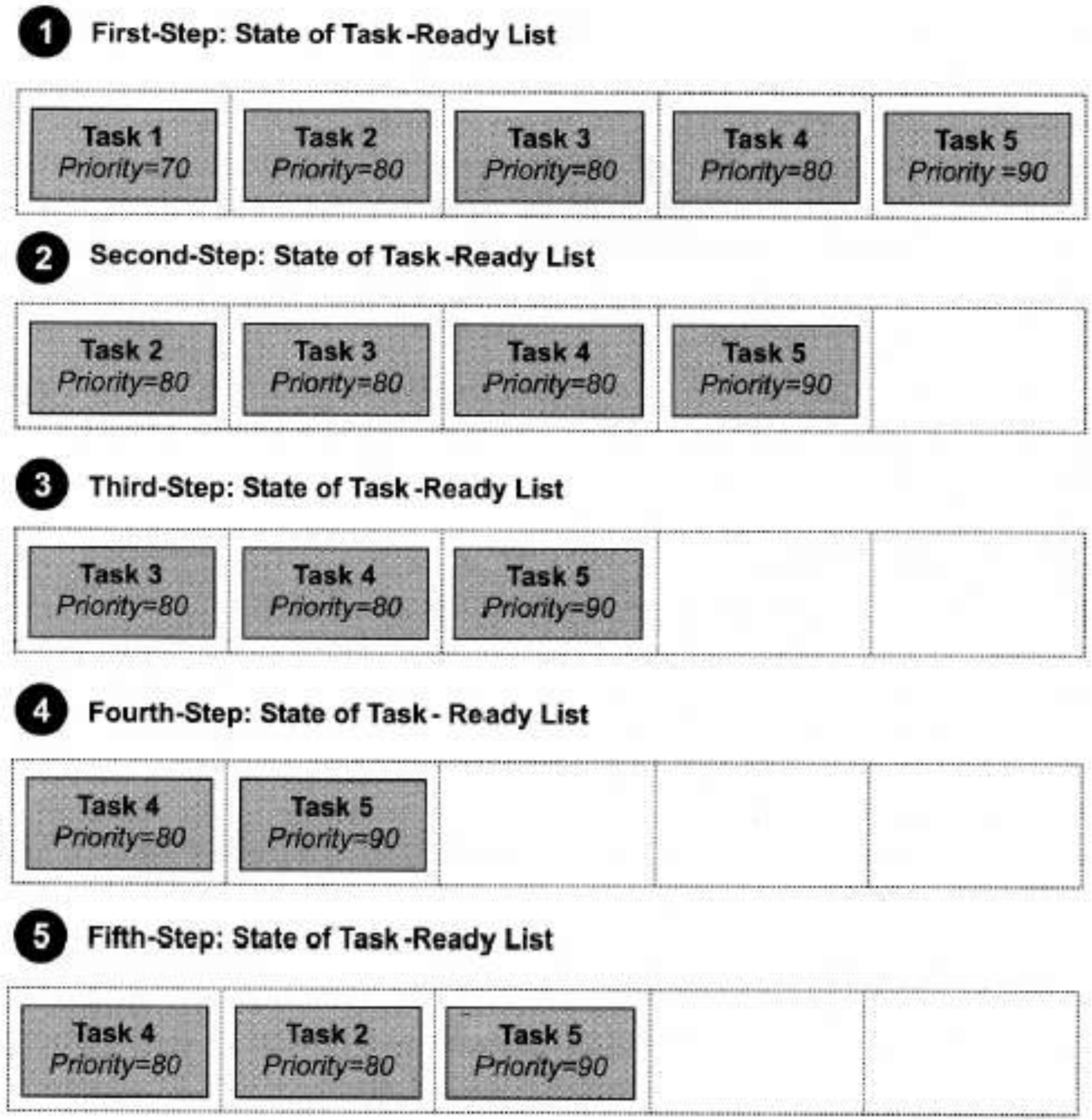


Figure 2-3, Task states

An example (255=lowest, 0=highest)



1 First-Step: State of Task -Ready List

Task 1 Priority=70	Task 2 Priority=80	Task 3 Priority=80	Task 4 Priority=80	Task 5 Priority =90

2 Second-Step: State of Task -Ready List

Task 2 Priority=80	Task 3 Priority=80	Task 4 Priority=80	Task 5 Priority=90	

3 Third-Step: State of Task -Ready List

Task 3 Priority=80	Task 4 Priority=80	Task 5 Priority=90		

4 Fourth-Step: State of Task - Ready List

Task 4 Priority=80	Task 5 Priority=90			

5 Fifth-Step: State of Task -Ready List

Task 4 Priority=80	Task 2 Priority=80	Task 5 Priority=90		

Task Management

1. Set of services used to allow application software developers to design their software as a number of separate chunks of software each handling a distinct topic, a distinct goal, and sometimes its own real-time deadline.

2. Main service offered is Task Scheduling
   1. controls the execution of application software tasks
   2. can make them run in a very timely and responsive fashion.

Scheduling

1. **Scheduling** is the process of deciding how to commit resources between a variety of possible tasks. Time can be specified (scheduling a flight to leave at 8:00) or floating as part of a sequence of events.
2. Scheduling is a key concept in computer multitasking, multiprocessing operating system and real-time operating system designs.
3. Scheduling refers to the way processes are assigned to run on the available CPUs, since there are typically many more processes running than there are available CPUs. This assignment is carried out by software's known as a scheduler and dispatcher.

The scheduler is concerned mainly with

CPU utilization - to keep the CPU as busy as possible.

Throughput - number of processes that complete their execution per time unit.

Turnaround - total time between submission of a process and its completion.

Waiting time - amount of time a process has been waiting in the ready queue.

Response time - amount of time it takes from when a request was submitted until the first response is produced.

Fairness - Equal CPU time to each thread.

Task Scheduling

1. Schedulers are parts of the kernel responsible for determining which task runs next
2. Most real-time kernels use priority-based scheduling
   1. Each task is assigned a priority based on its importance
   2. The priority is application-specific
3. The scheduling can be handled automatically.
4. Many kernels also provide a set of API calls that allows developers to control the state changes.
5. Manual scheduling
6. Non Real -time systems usually use Non-preemptive Scheduling
   - Once a task starts executing, it completes its full execution
7. Most RTOS perform priority-based preemptive task scheduling.
8. Basic rules for priority based preemptive task scheduling
   - The Highest Priority Task that is Ready to Run, will be the Task that Must be Running.

Why Scheduling is necessary in RTOS

- More information about the tasks are known
  - Number of tasks
  - Resource Requirements
  - Execution time
  - Deadlines
- Being a more deterministic system better scheduling algorithms can be devised.

Different Approach used in Scheduling Algorithms

- Clock Driven Scheduling

- Weighted Round Robin Scheduling

- Priority Scheduling

Scheduling Algorithms in RTOS (*cont.)*

- Clock Driven
  - All parameters about jobs (execution time/deadline) known in advance.
  - Schedule can be computed offline or at some regular time instances.
  - Minimal runtime overhead.
  - Not suitable for many applications.

Scheduling Algorithms in RTOS (*cont.*)

- Weighted Round Robin
  - Jobs scheduled in FIFO manner
  - Time quantum given to jobs is proportional to it's weight
  - Example use : High speed switching network
    - QOS guarantee.
  - Not suitable for precedence constrained jobs.
    - Job A can run only after Job B. No point in giving time quantum to Job B before Job A.

Scheduling Algorithms in RTOS (*cont.*)

- Priority Scheduling
  - Processor never left idle when there are ready tasks
  - Processor allocated to processes according to priorities
  - Priorities
    - Static - at design time
    - Dynamic - at runtime

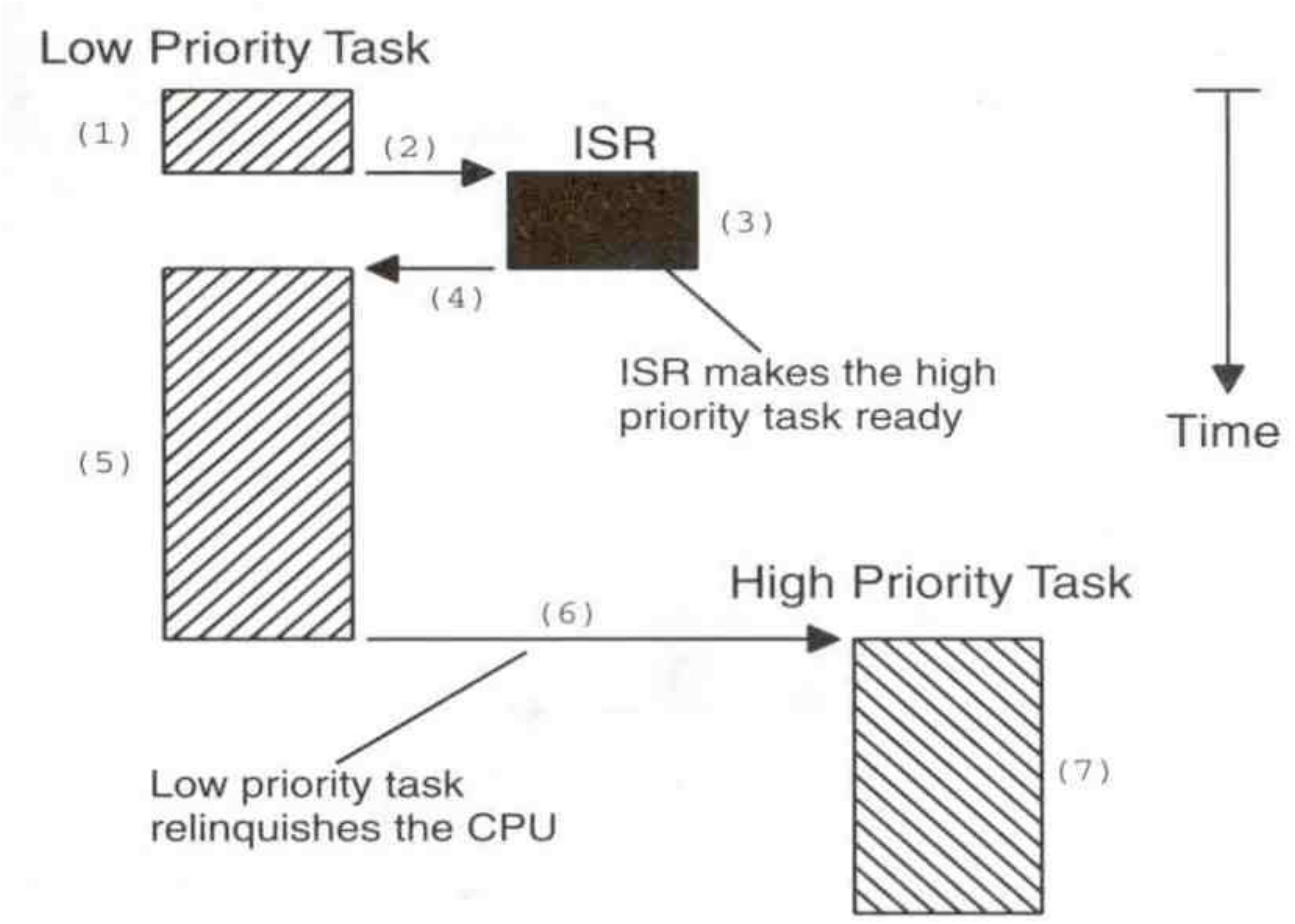
Priority-Based Kernels

- There are two types
  - Non-preemptive
  - Preemptive

Non-Preemptive Kernels

- Perform “cooperative multitasking”
  - Each task must explicitly give up control of the CPU
  - This must be done frequently to maintain the illusion of concurrency
- Asynchronous events are still handled by ISRs
  - ISRs can make a higher-priority task ready to run
  - But ISRs always return to the interrupted tasks

Non-Preemptive Kernels


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)

Advantages of Non-Preemptive Kernels

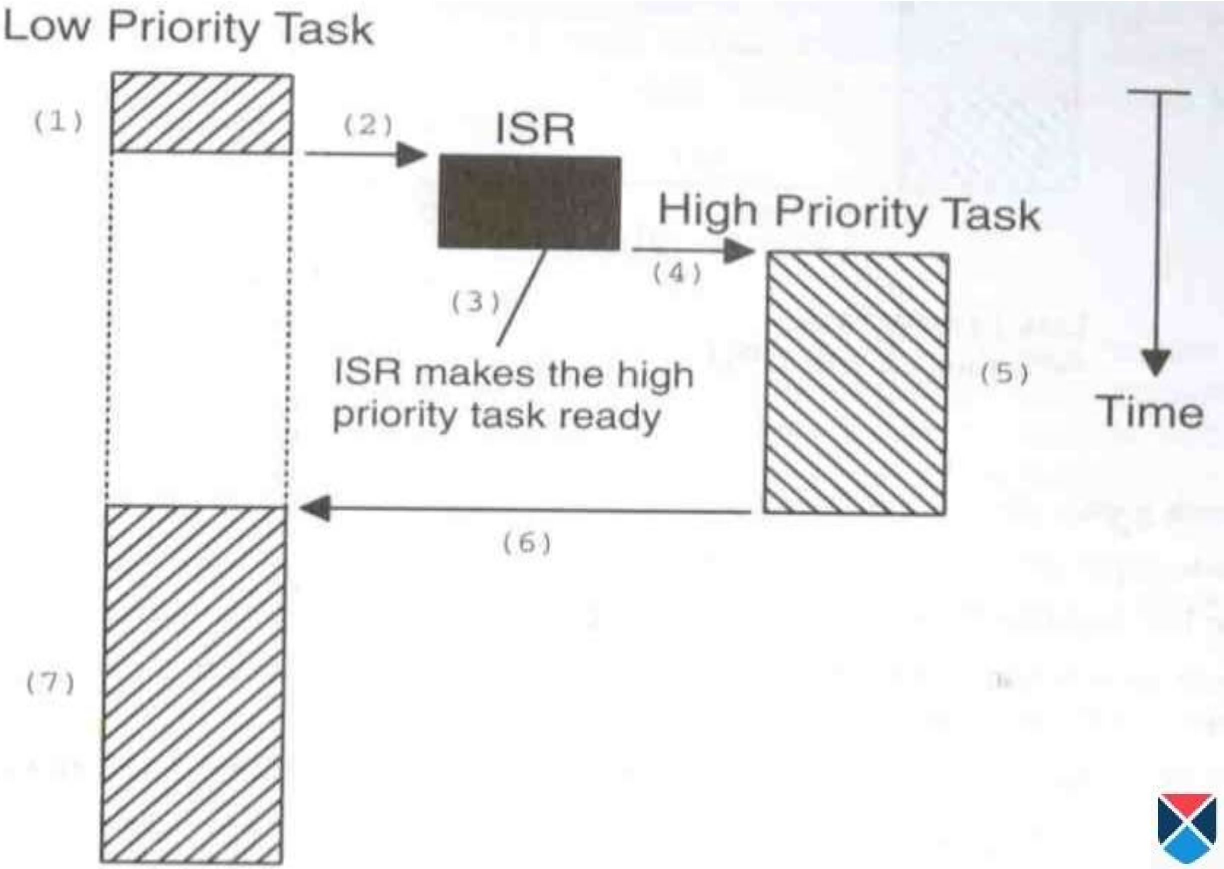
- Interrupt latency is typically low
- Can use non-reentrant functions without fear of corruption by another task
  - Because each task can run to completion before it relinquishes the CPU
  - However, non-reentrant functions should not be allowed to give up control of the CPU
- Task-response is now given by the time of the longest task
  - much lower than with F/B systems
- Less need to guard shared data through the use of semaphores
  - However, this rule is not absolute
  - Shared I/O devices can still require the use of mutual exclusion semaphores
  - A task might still need exclusive access to a printer

Disadvantages of Non-Preemptive Kernels

- Responsiveness
  - A higher priority task might have to wait for a long time
  - Response time is nondeterministic
- Very few commercial kernels are non-preemptive

Preemptive Kernels

- The highest-priority task ready to run is always given control of the CPU
  - If an ISR makes a higher-priority task ready, the higher-priority task is resumed (instead of the interrupted task)
- Most commercial real-time kernels are preemptive



Advantages of Preemptive Kernels

- Execution of the highest-priority task is deterministic
- Task-level response time is minimized

Disadvantages of Preemptive Kernels

- Should not use non-reentrant functions unless exclusive access to these functions is ensured

Context Switch

- Context switch
  - Also called task switch or process switch
  - Occurred when a scheduler switches from one task to another

- Although each process can have its own address space, all processes have to share the CPU registers
  - Kernel ensure that each such register is loaded with the value it had when the process was suspended

Each task has its own context

–The state of the CPU registers required for tasks*'running*

–When a task running, its context is highly dynamic

–The context of a task is stored in its process descriptor

Operations

–Save the context of the current process

–Load the context of new process

–If have page table, update the page table entries

–Flush those TLB entries that belonged to the old process

Scheduling Algorithms

First In First Out / First come First Serve

Round-robin scheduling

Round-robin with priority scheduling

Shortest- Job- First

Fixed priority pre-emptive scheduling

pre-emptive multitasking scheduling

First In First Out Algorithm

Also known as First- Come, First -Served (FCFS).

It is the simplest scheduling algorithm.

FIFO simply queues processes in the order that they arrive in the ready queue.

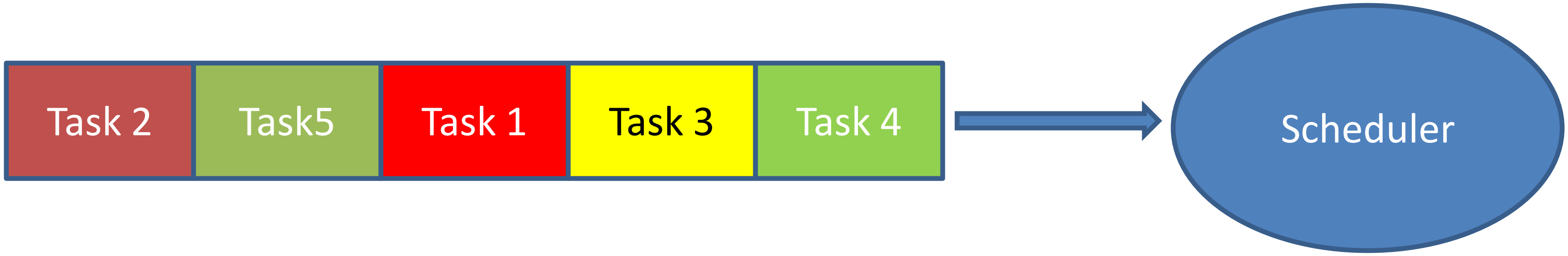
Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.

Throughput can be low, since long processes can hog the CPU Turnaround time, waiting time and response time can be low for the same reasons above No prioritization occurs, thus this system has trouble meeting process deadlines.

The lack of prioritization does permit every process to eventually complete, hence no starvation.

starvation is a multitasking-related problem, where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task
Starvation is similar in effect to deadlock

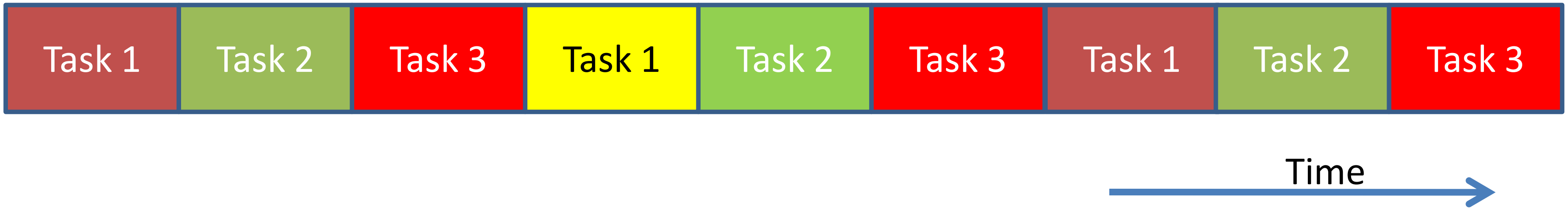
First In First Out Algorithm



Executed Task
Task 4
Task 3
Task 1
Task 5
Task 2

1. In F-I-F-O algorithm the task which are in Ready to Run are kept in a queue.
2. CPU serves the task on First come first basis
3. It is very simple to implement. But not suited for large application
4. This is considered as a very good algorithm when few small tasks all with small execution times

Round-robin scheduling Algorithm



Task 1 | Task 2 | Task 3 | Task 1 | Task 2 | Task 3 | Task 1 | Task 2 | Task 3

Time

1. The scheduler assigns **a fixed time unit per process, and cycles through them**.
2. RR scheduling involves extensive overhead, especially with a small time unit.
3. Balanced throughput between FCFS and SJN, shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJN.
4. Fastest average response time, **waiting time is dependent on number of processes**, and not average process length.
5. Because of high waiting times, deadlines are rarely met in a pure RR system.
6. Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

Round-robin with priority scheduling

The Round Robin Algorithm is slightly modified by assigning priority to all tasks.

Highest priority task can interrupt the CPU so it can be executed.

Shortest- Job- First Algorithm

Also known as ***S**hortest- **J**ob- **F**irst* (SJF).

1. **With this strategy the scheduler arranges processes with the least estimated processing time remaining to be next in the queue.**

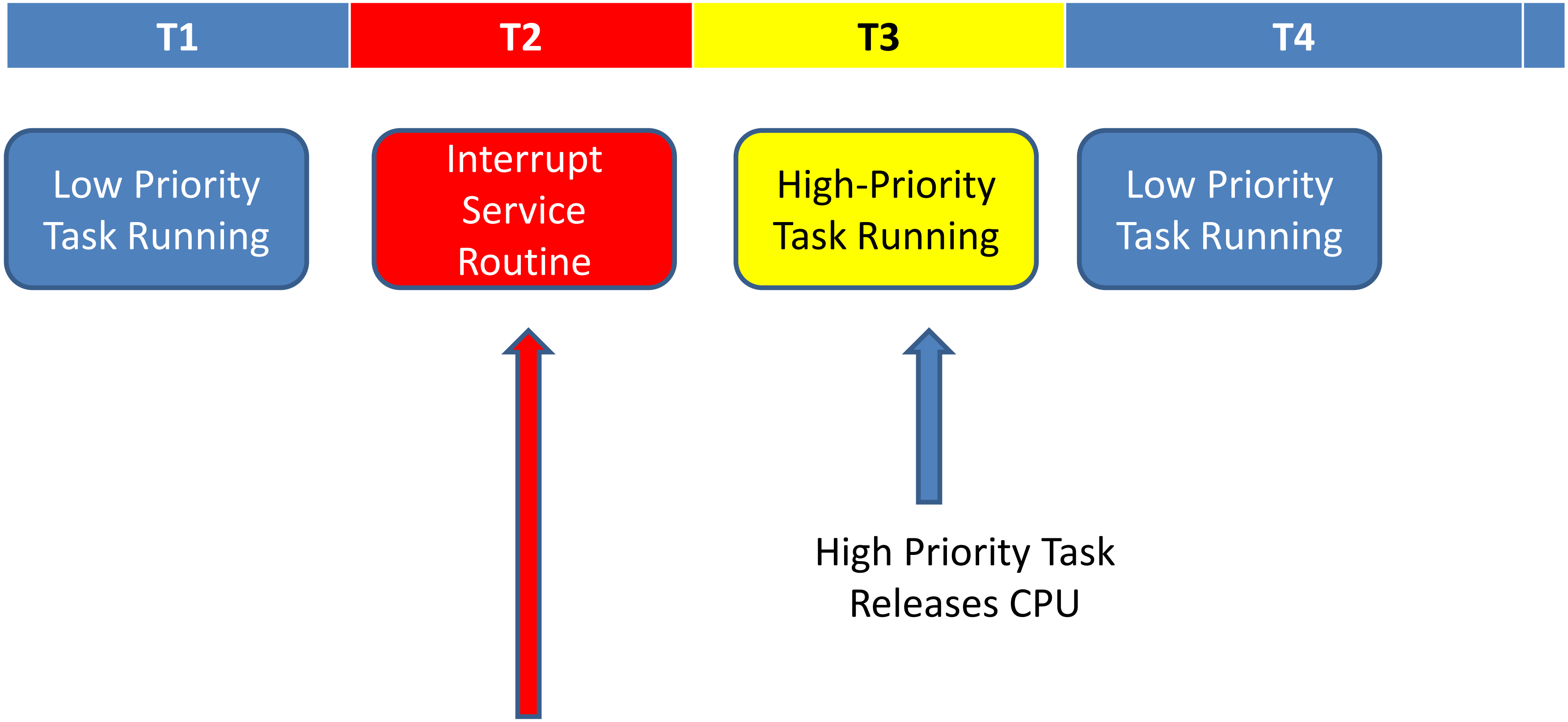
2. This requires advanced knowledge or estimations about the time required for a process to complete.

3. **If a shorter process arrives during another process' execution, the currently running process may be interrupted,** dividing that process into two separate computing blocks. This creates excess overhead through additional context switching. The scheduler must also place each incoming process into a specific place in the queue, creating additional overhead.

4. This algorithm is designed for maximum throughput in most scenarios.
5. **Waiting time and response time increase as the process' computational requirements increase.** Since turnaround time is based on waiting time plus processing time, longer processes are significantly affected by this. Overall waiting time is smaller than FIFO, however since no process has to wait for the termination of the longest process.
6. **No particular attention is given to deadlines**, the programmer can only attempt to make processes with deadlines as short as possible.
7. **Starvation is possible**, especially in a busy system with many small processes being run.

Fixed priority pre-emptive scheduling Algorithm

1. The O/S assigns a fixed priority rank to every process, and the scheduler arranges the processes in the ready queue in order of their priority.
2. Lower priority processes get interrupted by incoming higher priority processes.
3. Overhead is not minimal, nor is it significant.
4. This scheduling has no particular advantage in terms of throughput over FIFO scheduling.
5. Waiting time and response time depend on the priority of the process.
6. Higher priority processes have smaller waiting and response times.
7. Deadlines can be met by giving processes with deadlines a higher priority.
8. Starvation of lower priority processes is possible with large amounts of high priority processes queuing for CPU ti



T1

T2

T3

T4

Low Priority Task Running

Interrupt Service Routine

High-Priority Task Running

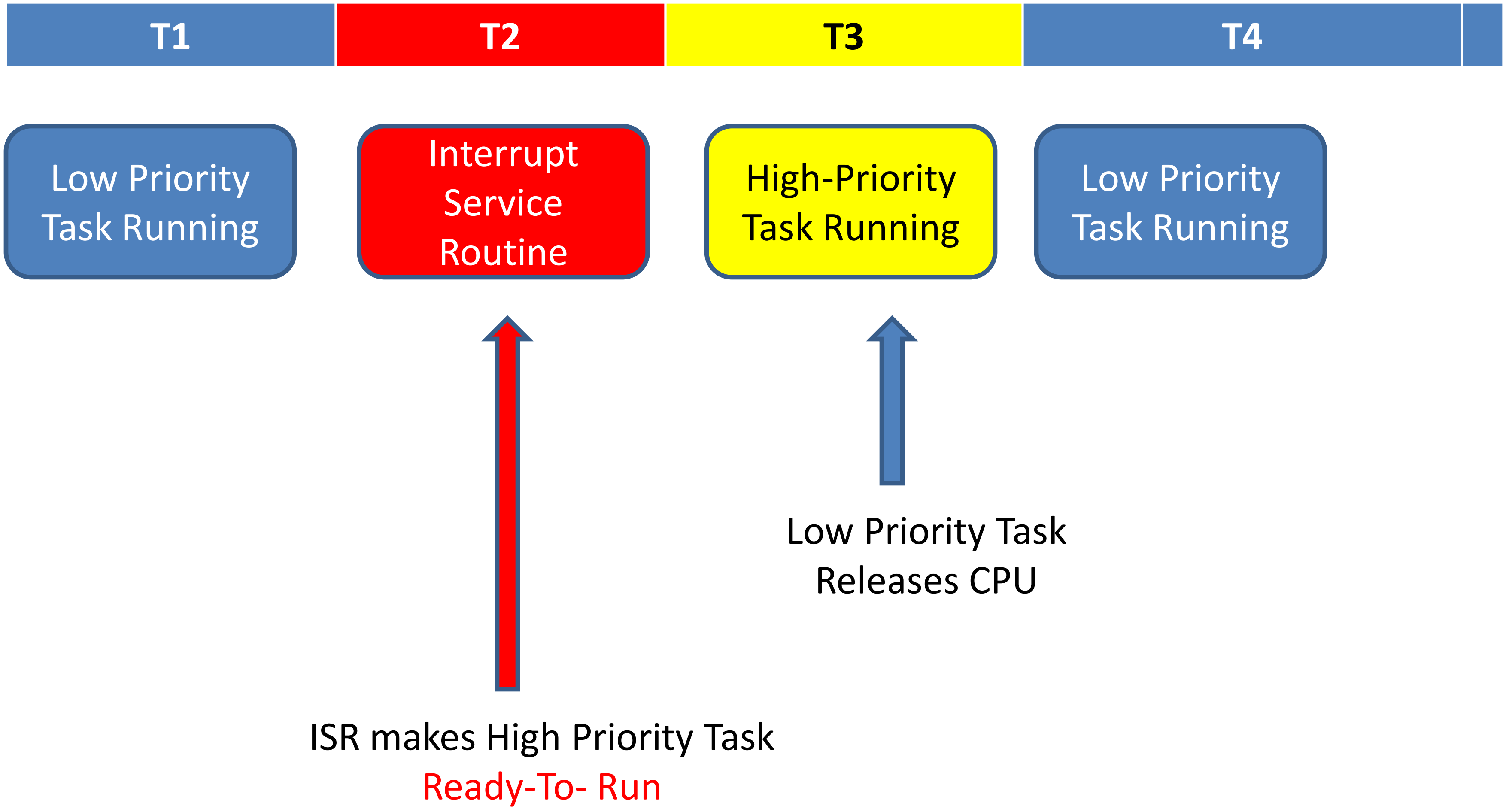
Low Priority Task Running

High Priority Task Releases CPU

ISR makes High Priority Task
Ready-To- Run

- Consider Two task are to be executed
- One High Priority and one Low Priority Task.
- Assume High Priority task waits for an external events then the
- Then low priority task will get chance to run.
- So low priority task is putted in running state.
- After some time if high priority task is ready.
- Then using ISR the high priority task is moved to Ready-to-Run state.
- After ISR execution the low priority task will continue to be execute.
- After some time, low priority task releases the CPU and then the
- high priority task get executed.

pre-emptive multitasking scheduling Algorithm



T1

T2

T3

T4

Low Priority Task Running

Interrupt Service Routine

High-Priority Task Running

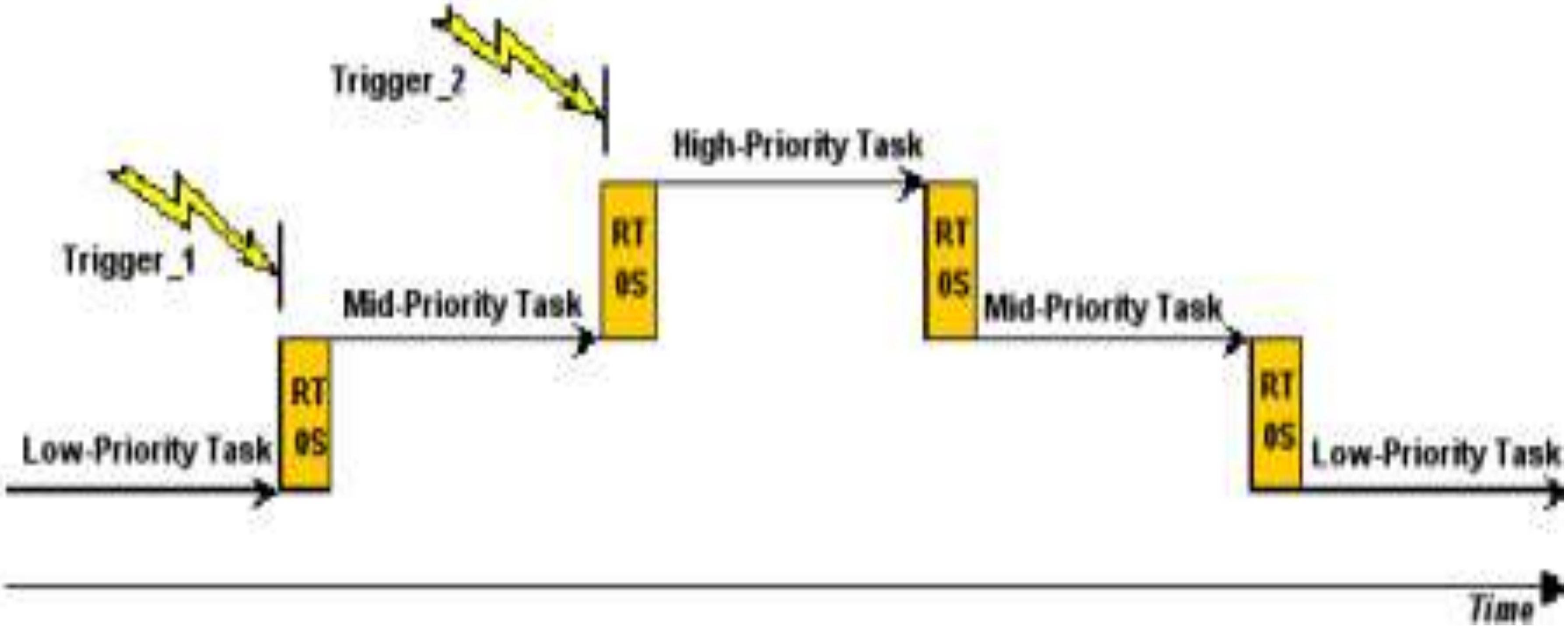
Low Priority Task Running

ISR makes High Priority Task Ready-To- Run

Low Priority Task Releases CPU

- Consider Two task are to be executed
- One High Priority and one Low Priority Task.
- Assume High Priority task waits for an external events then the
- Then low priority task will get chance to run.
- So low priority task is putted in running state.
- After some time if high priority task is ready.
- Then using ISR the high priority task is moved to Ready-to-Run state.
- After ISR execution the High priority task will start to execute and low priority task is putted in wait state.
- After some time, High priority task releases the CPU and then the
- low priority task get resumed.

Nested Preemption



Timeline for Priority-based Preemptive Scheduling

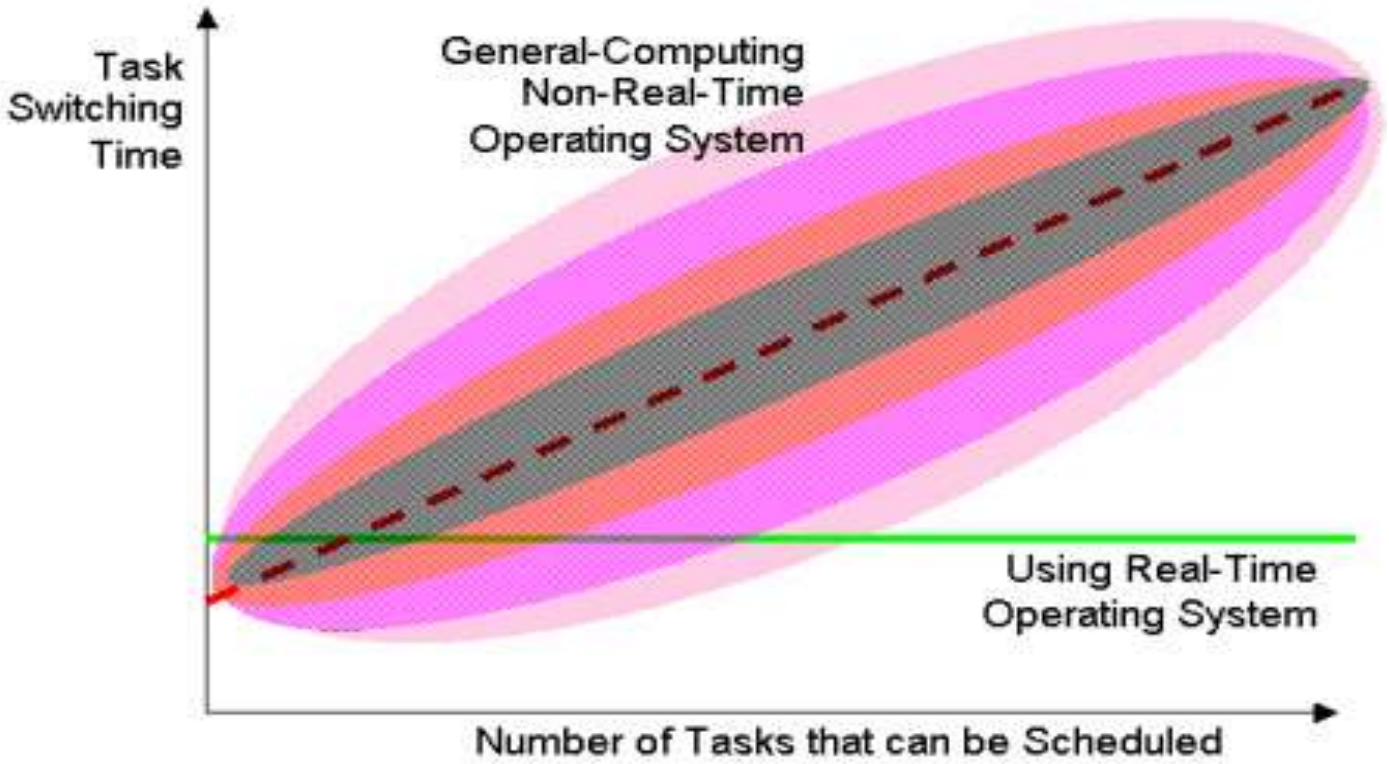
Task Switch

- Each time the priority-based preemptive scheduler is alerted by an External world trigger / Software trigger it shall go through the following steps that constitute a **Task Switch**:
  - Determine whether the currently running task should continue to run.
  - Determine which task should run next.
  - Save the environment of the task that was stopped (so it can continue later).
  - Set up the running environment of the task that will run next.
  - Allow the selected task to run.

Task Switch

- A Non Real time operating system might do task switching only at timer tick times.
- Even with preemptive schedulers a large array of tasks is searched before a task switch.
- A Real time OS shall use Incrementally arranged tables to save on time.

Task Switch



Task Switching Timing

Interrupts

Interrupts can suspend the execution of a task and, if a higher priority task is awakened as a result of the interrupt, the highest priority task will run as soon as all nested interrupts complete.

Interrupts can be nested up to 255 levels deep.

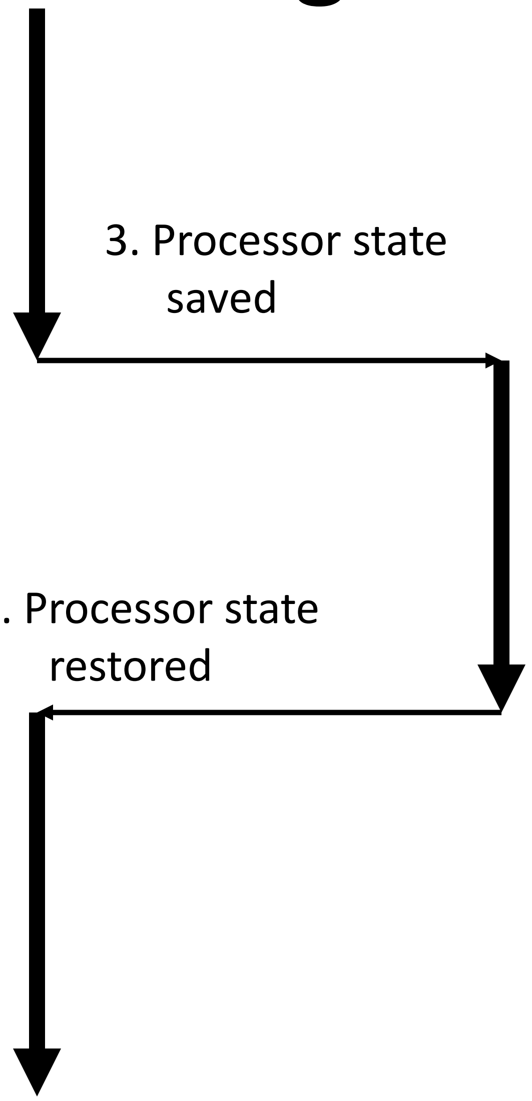
An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred. When an interrupt is recognized, the CPU saves part (or all) of its context (i.e. registers) and jumps to a special subroutine called an Interrupt Service Routine, or ISR. The ISR processes the event and upon completion of the ISR, the program returns to:

a) The background for a foreground/background system.
b) The interrupted task for a non-preemptive kernel.
c) The highest priority task ready-to-run for a preemptive kernel.

Interrupt Service Routines

- Most interrupt routines:
- Copy peripheral data into a buffer
- Indicate to other code that data has arrived
- Acknowledge the interrupt (tell hardware)
- Longer reaction to interrupt performed outside interrupt routine
- E.g., causes a process to start or resume running

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

Interrupt latency

In real-time operating systems, interrupt latency 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.

The interrupt latency is the interval of time measured from the instant an interrupt is asserted until the corresponding ISR begins to execute. The worst-case latency for any given interrupt is a sum of many things, from longest to shortest:
The longest period global interrupt recognition is inhibited
The time it would take to execute all higher priority interrupts if they occurred simultaneously The time it takes the specific ISR to service all of its interrupt requests (if multiple are possible)
The time it takes to finish the program instructions in progress and save the current program state and begin the ISR

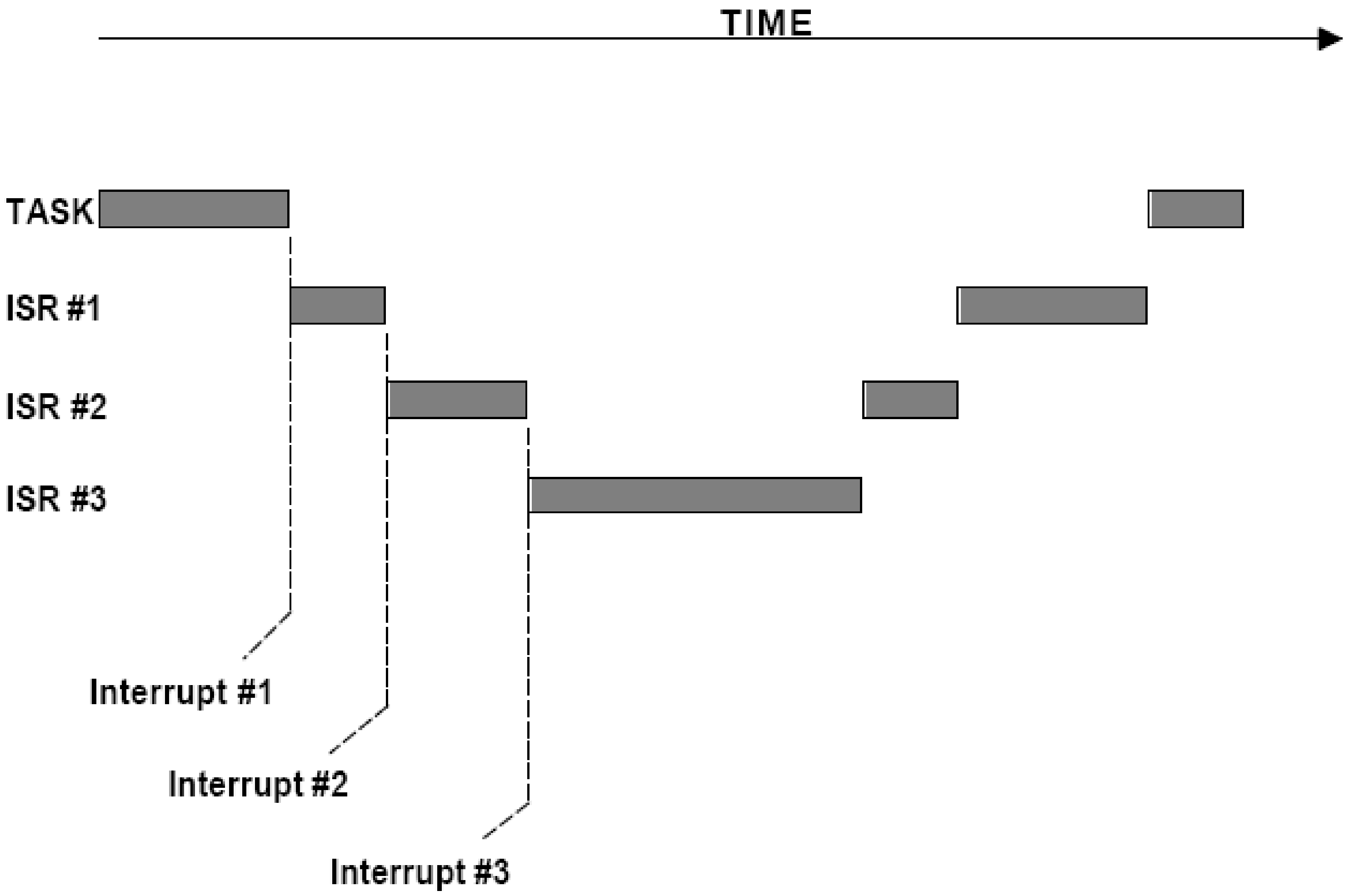


Figure 2-18, Interrupt nesting.

Interrupt Latency

Probably the most important specification of a real-time kernel is the amount of time interrupts are disabled. All real-time systems disable interrupts to manipulate critical sections of code and re -enable interrupts when the critical section has executed. The longer interrupts are disabled, the higher the *interrupt latency*.

Interrupt latency is given by:

Maximum amount of time interrupts are disabled +
Time to start executing the first instruction in the ISR

Interrupt 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 interrupt response time accounts for all the overhead involved in handling an interrupt.

Typically, the processor's context (CPU registers) is saved on the stack before the user code is executed. For a foreground/background system, the user ISR code is executed immediately after saving the processor's context.

The response time is given by:

Interrupt latency + Time to save the CPU's context

For a **non-preemptive kernel**, the user ISR code is executed immediately after the processor's context is saved. The response time to an interrupt for a non-preemptive kernel is given by:

Interrupt latency + Time to save the CPU's context

For a **preemptive kernel**, a special function provided by the kernel needs to be called. This function notifies the kernel that an ISR is in progress and allows the kernel to keep track of interrupt nesting. For µC/OS-II, this function is called **OSIntEnter()**.

The response time to an interrupt for a preemptive kernel is given by:

Interrupt latency +

Time to save the CPU's context +

Execution time of the kernel ISR entry function

Interrupt Recovery

Interrupt recovery is defined as the time required for the processor to return to the interrupted code.

Interrupt recovery in a foreground/background system simply involves restoring the processor's context and returning to the interrupted task. Interrupt recovery is given by:

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

As with a foreground/background system, interrupt recovery with a **non-preemptive kernel** simply involves restoring the processor's context and returning to the interrupted task. Interrupt recovery is thus:

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

For a **preemptive kernel**, interrupt recovery is more complex. Typically, a function provided by the kernel is called at the end of the ISR. For µC/OS-II, this function is called **OSIntExit()** and allows the kernel to determine if all interrupts have nested. If all interrupts have nested (i.e. a return from interrupt will return to task level code), the kernel will determine if a higher priority task has been made ready-to-run as a result of the ISR. If a higher priority task is ready-to-run as a result of the ISR, this task is resumed. Note that, in this case, the interrupted task will be resumed
only when it again becomes the highest priority task ready-to-run. For a preemptive ke rnel, interrupt recovery is given by:

Time to determine if a higher priority task is ready +
Time to restore the CPU's context of the highest priority task +
Time to execute the return from interrupt instruction

Interrupt Latency, Response, and Recovery

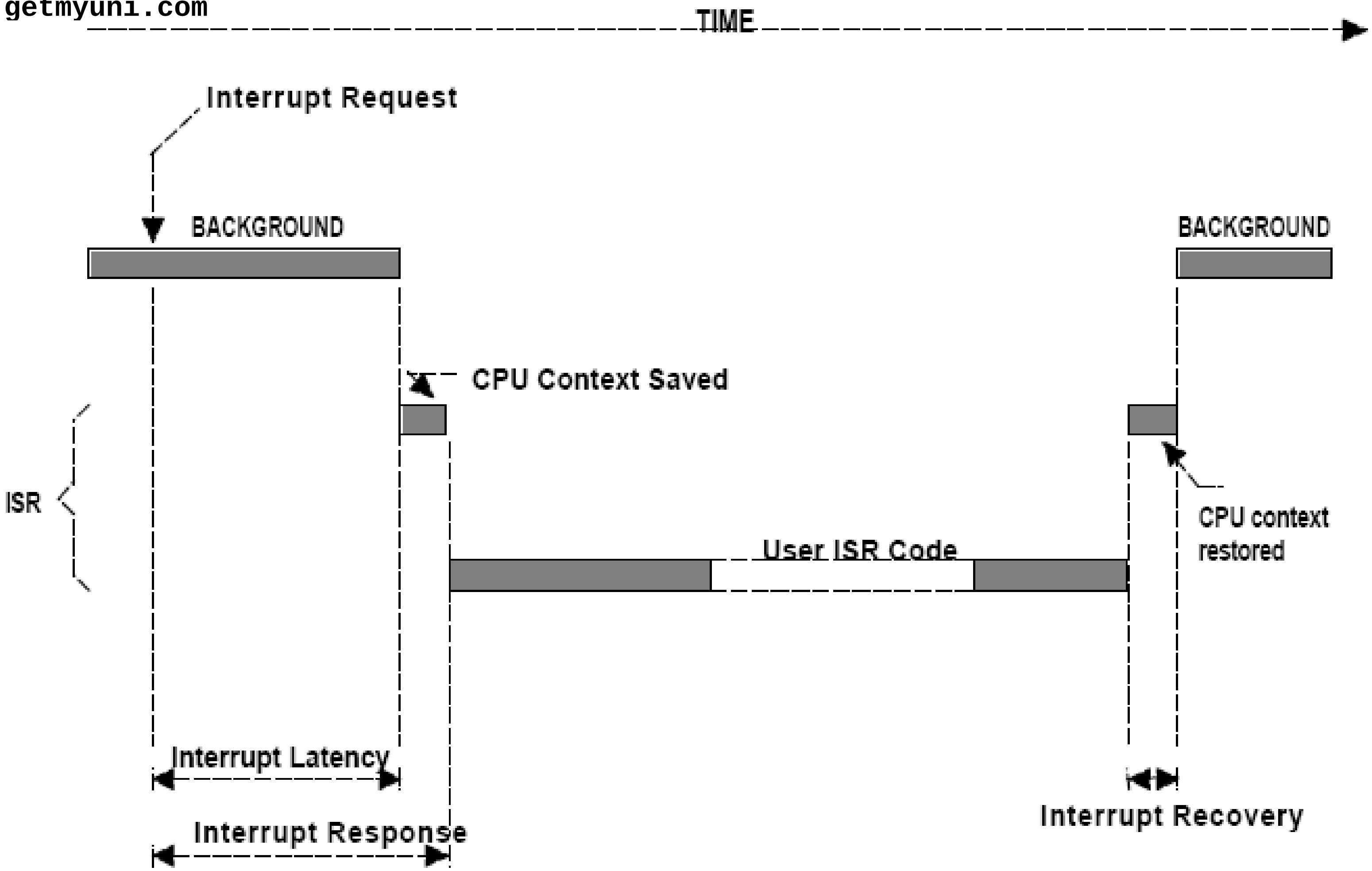


Figure 2-19, Interrupt latency, response, and recovery (Foreground/Background)

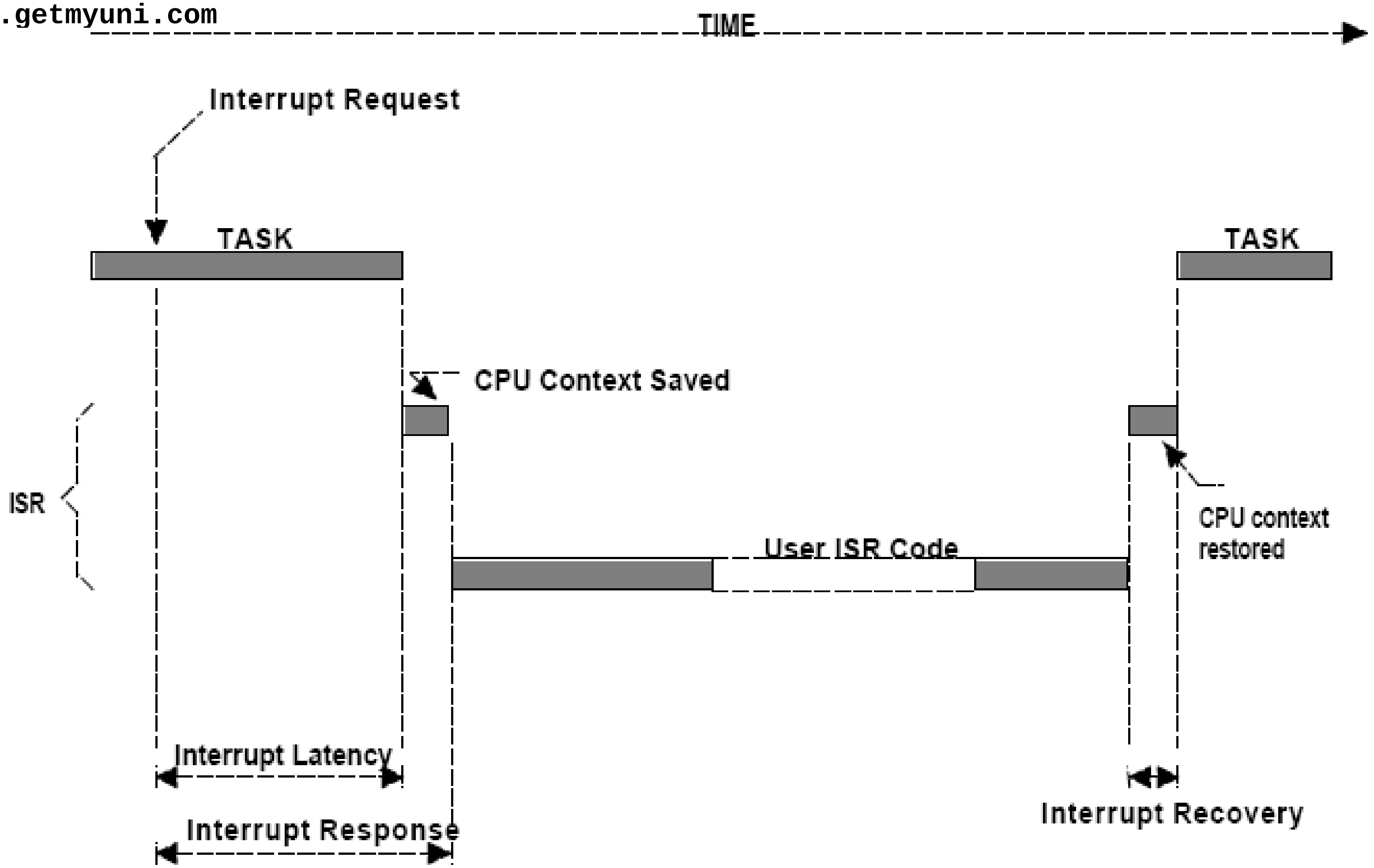


Figure 2-20, Interrupt latency, response, and recovery (Non-preemptive kernel)

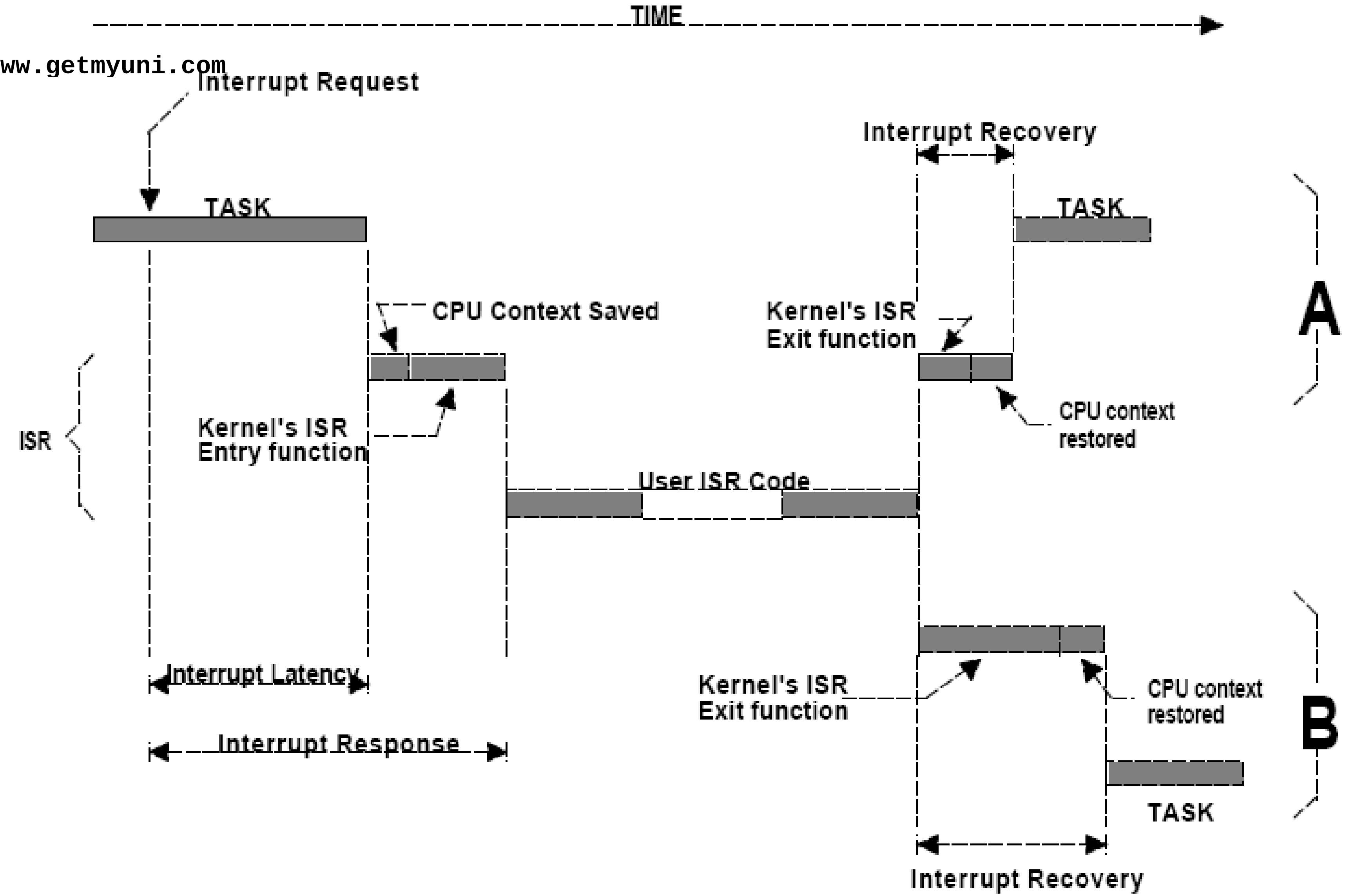


Figure 2-21, Interrupt latency, response, and recovery (Preemptive kernel)

Semaphore

The semaphore was invented by Edgser Dijkstra in the mid 1960s.

A semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores are used to:

a) control access to a shared resource (mutual exclusion);
b) signal the occurrence of an event;
c) allow two tasks to synchronize their activities.

A semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment.

A semaphore generally takes one of two forms: **binary and counting.**

A **binary semaphore** is a simple "true/false" (locked/unlocked) flag that controls access to a single resource.

As its name implies, a binary semaphore can only take two values: **0** or **1**.

A **counting semaphore** is a counter for a set of available resources. Either semaphore type may be employed to prevent a race condition.

A counting semaphore allows values between **0** and **255**, **65535** or **4294967295**, depending on whether the semaphore mechanism is implemented using 8, 16 or 32 bits, respectively.

The actual size depends on the kernel used. Along with the semaphore's value, the kernel also needs to keep track of tasks waiting for the semaphore's availability.

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

The initial value of the semaphore must be provided when the semaphore is initialized. The waiting list of tasks is always initially empty.

Semaphores are especially useful when tasks are sharing I/O devices.

Imagine what would happen if two tasks were allowed to send characters to a printer at the same time.

The printer would contain interleaved data from each task.

For instance, if task #1 tried to print "I am task #1!" and
task #2 tried to print "I am task #2!"

then the printout could look like this:
I Ia amm t tasask k#1 #!2!

In this case, we can use a semaphore and initialize it to **1**

(i.e. a binary semaphore).

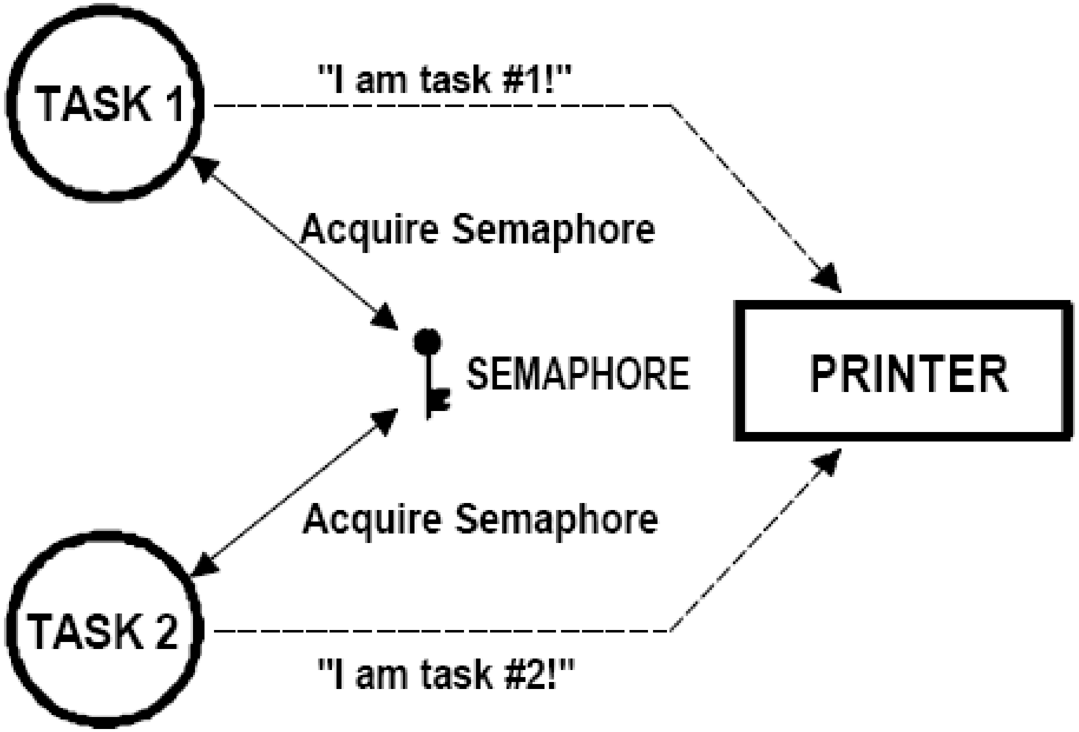


Figure 2-9, Using a semaphore to get permission to access a printer.

Counting semaphore analogy

Assume the "resources" are tables in a restaurant and
the "processes" are the restaurant's customers.

The "semaphore" is represented by the MAIN_WAITER,
who has sole responsibility and authority in granting access to the tables.

The MAIN_WAITER mentally maintains a count of unoccupied tables (utables) and always knows who is first to be seated when a table becomes available.

He or she is also very focused and can never be interrupted while performing his or her duties.

When the restaurant opens for business, it will initially be empty.

In other words, the "value" of the "semaphore" will be equal to the number of tables (tables) in the restaurant—that is, utables=tables.

When someone arrives, the MAIN_WAITER will seat him or her, and will mentally reduce the count of available tables, that is, utables=utables-1.

Now, the "value" of the "semaphore" will be equal to the number of tables in the restaurant minus one.

If several diners simultaneously arrive, the MAIN_WAITER will seat them in the order of arrival, assuming there are sufficient tables for all (utables > 0). Arriving diners with reservations may be seated ahead of others who do not have reservations, the former having priority over the latter.

For each table taken, the MAIN_WAITER will again mentally compute utables=utables-1.

If all tables are occupied, that is, utables=0, new arrivals will have to wait their turn to be seated.

As each diner leaves, the MAIN_WAITER will mentally compute utables=utables+1.

If another diner is waiting and at least one unoccupied table is available, the MAIN_WAITER will seat him or her, and again mentally compute utables=utables-1.

Eventually, all diners will have left and the MAIN_WAITER utables=utables+1 mental computation will result in utables=tables—an empty restaurant.

Task Priority

A priority is assigned to each task. The more important the task, the higher the priority given to it.

Static Priorities

Task priorities are said to be *static* when the priority of each task does not change during the application's execution.
Each task is thus given a fixed priority at compile time. All the tasks and their timing constraints are known at compile time in a system where priorities are static.

Dynamic Priorities

Task priorities are said to be dynamic if the priority of tasks can be changed during the application's execution; each task can change its priority at run-time. This is a desirable feature to have in a real-time kernel to avoid priority
inversions.

Priority Inversions

Priority inversion is a problem in real-time systems and occurs mostly when you use a real-time kernel.

Task#1 has a higher priority than Task#2 which in turn has a higher priority than Task#3.

If Task#1 and Task#2 are both waiting for an event to occur and thus, Task#3 is executing **(1)**

At some point, Task#3 acquires a semaphore that it needs before it can access a shared resource **(2)**

Task#3 performs some operations on the acquired resource **(3)**

it gets preempted by the high priority task, Task#1 **(4).**

Task#1 executes for a while until it also wants to access the resource **(5)**

Because Task#3 owns the resource, Task#1 will have to wait until Task#3 releases the semaphore.

As Task#1 tries to get the semaphore, the kernel notices that the semaphore is already owned and thus, Task#1 gets suspended and Task#3 is resumed **(6).**

Task#3 continues execution until it gets preempted by Task#2 because the event that Task#2 was waiting for occurred **(7).**

Task #2 handles the event **(8)** and when it's done,

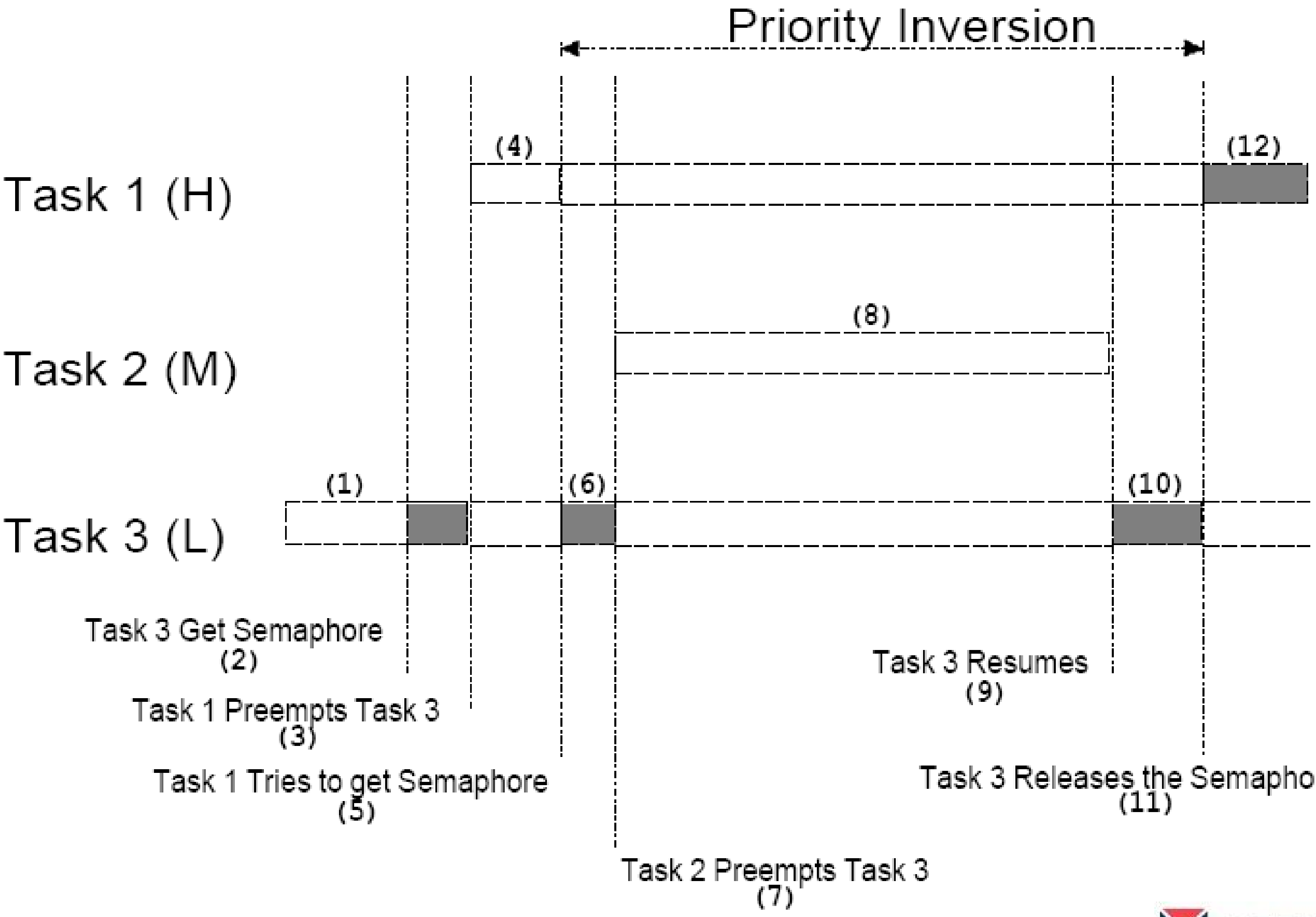
Task#2 relinquishes the CPU back to Task#3 **(9)**.

Task#3 finishes working with the resource **(10)**

and thus, releases the semaphore **(11).**

At this point, the kernel knows that a higher priority task is waiting for the semaphore and, a context switch is done to resume Task#1.
At this point, Task#1 has the semaphore and can thus access the shared resource **(12)**

The priority of Task#1 has been virtually reduced to that of Task#3's because it was waiting for the resource that Task#3 owned. The situation got aggravated when Task#2 preempted Task#3 which further delayed the execution of Task#1.



Mutual Exclusion

Resource

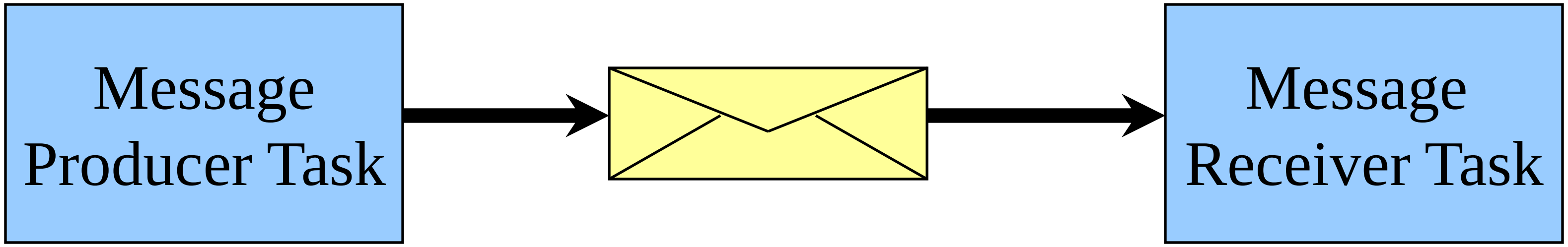
A resource is any entity used by a task. A resource can thus be an I/O device such as a printer, a keyboard, a display, etc. or a variable, a structure, an array, etc.

Shared Resource

A shared resource is a resource that can be used by more than one task. Each task should gain exclusive access to the shared resource to prevent data corruption. This is called *Mutual Exclusion.*

Inter-Task communication & Synchronization

- These services makes it possible to pass information from one task to another without information ever being damaged.
- Makes it possible for tasks to coordinate & productively cooperate with each other.
- The most important communication between tasks in an OS is the passing of data from one task to another.
- If messages are sent more quickly than they can be handled, the OS provides message queues for holding the messages until they can be processed.



It is sometimes necessary for a task or an ISR to communicate information to another task. This information transfer is called *intertask communication*. Information may be communicated between tasks in two ways: through global data or by sending messages.

When using global variables, each task or ISR must ensure that it has exclusive access to the variables.

If an ISR is involved, the only way to ensure exclusive access to the common variables is to disable interrupts.

If two tasks are sharing data each can gain exclusive access to the variables by using either disabling/enabling interrupts or through a semaphore (as we have seen). Note that a task can only communicate information to an ISR by using global variables.

A task is not aware when a global variable is changed by an ISR unless the ISR signals the task by using a semaphore or by having the task regularly poll the contents of the variable.

To correct this situation, you should consider using either a *message mailbox* or a *message queue*.

Message Mailboxes

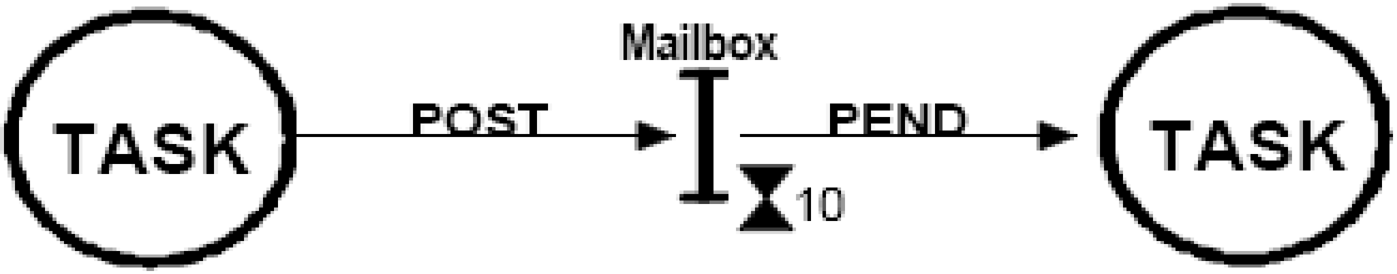


Figure 2-16, Message mailbox.

Messages can be sent to a task through kernel services.

A Message Mailbox, also called a message exchange, is typically a pointer size variable.

Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into this mailbox. Similarly, one or more tasks can receive messages through a service provided by the kernel.

Both the sending task and receiving task will agree as to what the pointer is actually pointing to.

Kernel services are typically provided to:

a) Initialize the contents of a mailbox. The mailbox may or may not initially contain a message.
b) Deposit a message into the mailbox (POST).
c) Wait for a message to be deposited into the mailbox (PEND).
d) Get a message from a mailbox, if one is present, but not suspend the caller if the mailbox is empty (ACCEPT). If the mailbox contains a message, the message is extracted from the mailbox.

A return code is used to notify the caller about the outcome of the call.

µC/OS-II provides five services to access mailboxes:
OSMboxCreate(),
OSMboxPend(),
OSMboxPost(),
OSMboxAccept() and
OSMboxQuery().

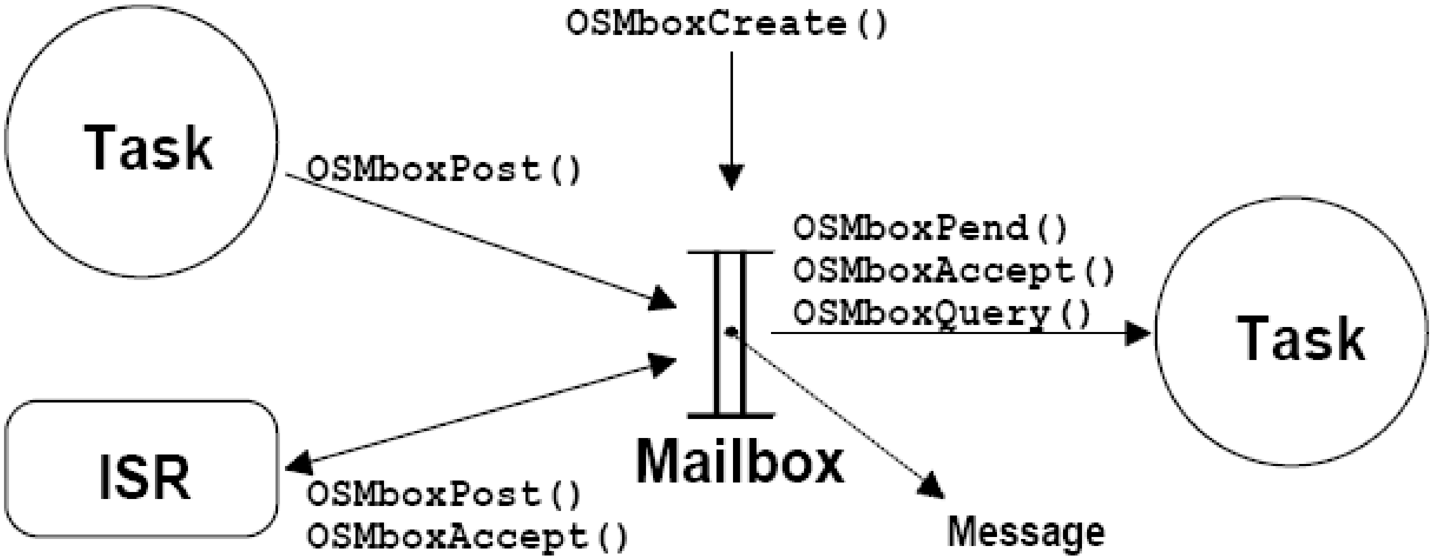


Figure 6-6, Relationship between tasks, ISRs and a message mailbox.

Message queue

message queues and mailboxes are software-engineering components used for inter process communication, or for inter-thread communication within the same process. They use a queue for messaging – the passing of control or of content. Group communication systems provide similar kinds of functionality.

A message queue is used to send one or more messages to a task.

A message queue is basically an array of mailboxes.

Through a service provided by the kernel, a task or an ISR can deposit a message (the pointer) into a message queue.

one or more tasks can receive messages through a service provided by the kernel.

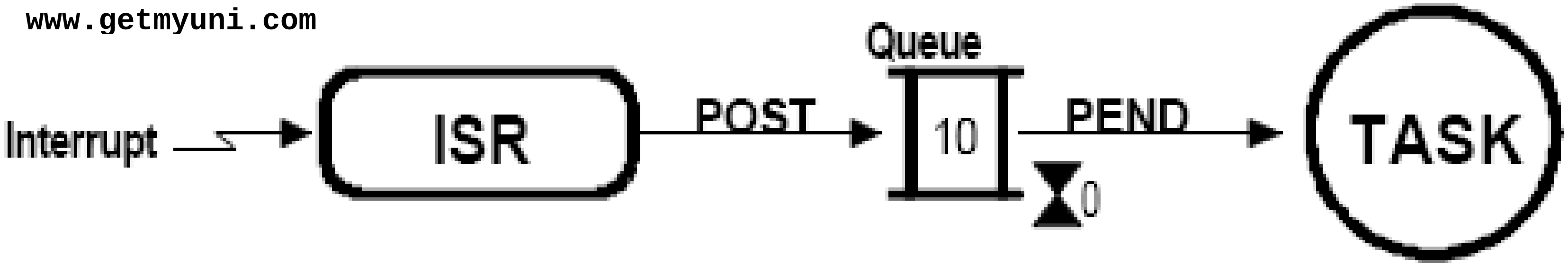


Figure 2-17, Message queue.

Kernel services are typically provided to:
a) Initialize the queue. The queue is always assumed to be empty after initialization.
b) Deposit a message into the queue (POST).
c) Wait for a message to be deposited into the queue (PEND).
d) Get a message from a queue, if one is present, but not suspend the caller if the queue is empty (ACCEPT).

If the queue contained a message, the message is extracted from the queue. A return code is used to notify the caller about the outcome of the call.

Both the sending task and receiving task will agree as to what the pointer is actually pointing to.

Generally, the first message inserted in the queue will be the first message extracted from the queue (FIFO).

In addition to extract messages in a FIFO fashion, µC/OS-II allows a task to get messages *Last-In-First-Out* (LIFO).

As with the mailbox, a waiting list is associated with each message queue in case more than one task is to receive messages through the queue.

A task desiring to receive a message from an empty queue will be suspended and placed on the waiting list until a message is received.

The kernel will allow the task waiting for a message to specify a timeout. If a message is not received before the timeout expires, the requesting task is made ready-to-run and an error code (indicating a timeout occurred) is returned to it. When a message is deposited into the queue, either the highest priority task or the first task to wait for the message will be given the message.

µC/OS-II provides seven services to access message queues:
OSQCreate(),
OSQPend(),
OSQPost(),
OSQPostFront(),
OSQAccept(),
OSQFlush() and
OSQQuery().

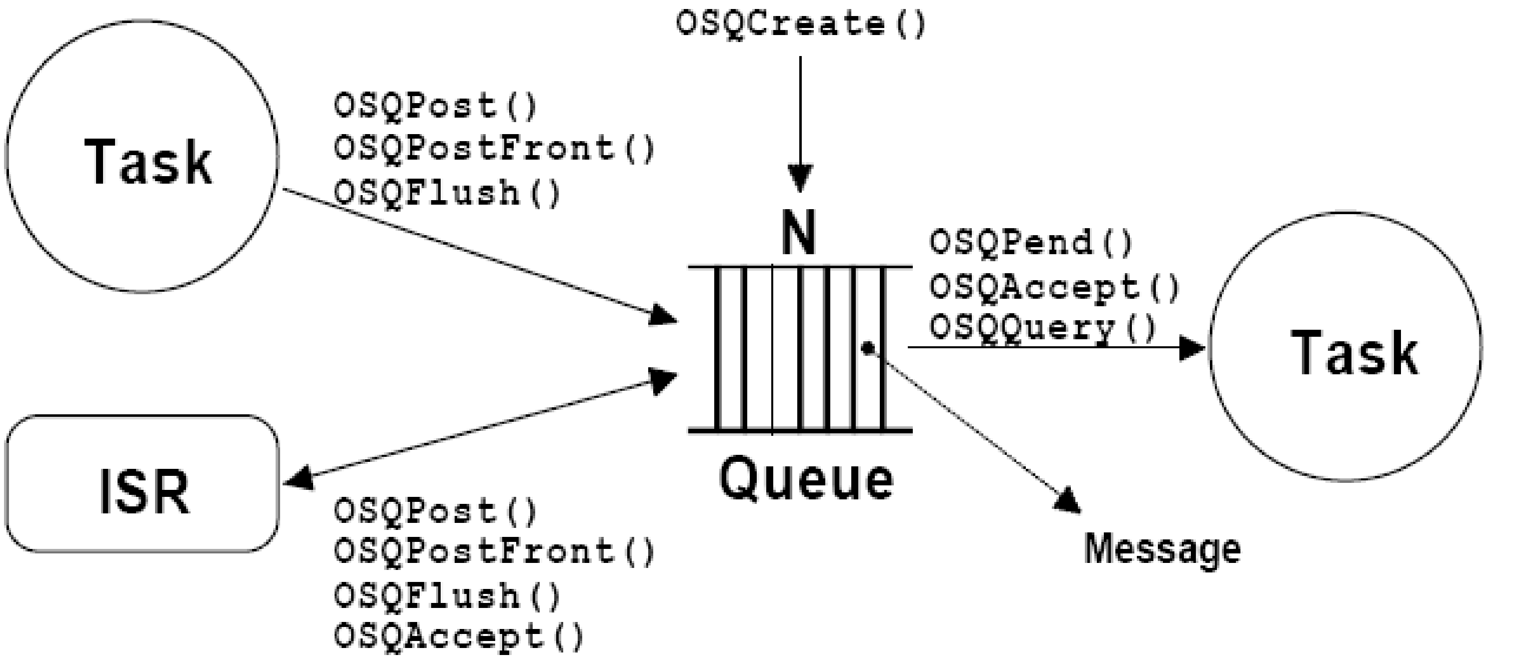


Figure 6-7, Relationship between tasks, ISRs and a message queue.