Distributed & Embedded Real-Time Systems (0640751)

Lecture (12)

Real-Time Operating Systems

Prof. Kasim M. Al-Aubidy
Philadelphia University

Lecture Outline:

- Components of a simple operating system.
- Types of operating systems.
- RTOS for real-time systems.
- What an RTOS does? How it works?
- Benefits and drawbacks of an RTOS.
- Soft and hard tasks.
- Task states.
- Task Scheduling algorithms.
- Open-Source RTOS

What Operating System is?

An **operating system (OS)** is a software program that manages the hardware and software resources of a computer.

- The OS performs basic tasks, such as controlling and allocating memory, prioritizing the processing of instructions, controlling input and output devices, facilitating networking, and managing files.
- Operating system goals:
  - Execute user programs and make solving user problems easier.
  - Make the computer system convenient to use.
  - Use the computer hardware in an efficient manner.

Kernel: the lowest level of any OS,

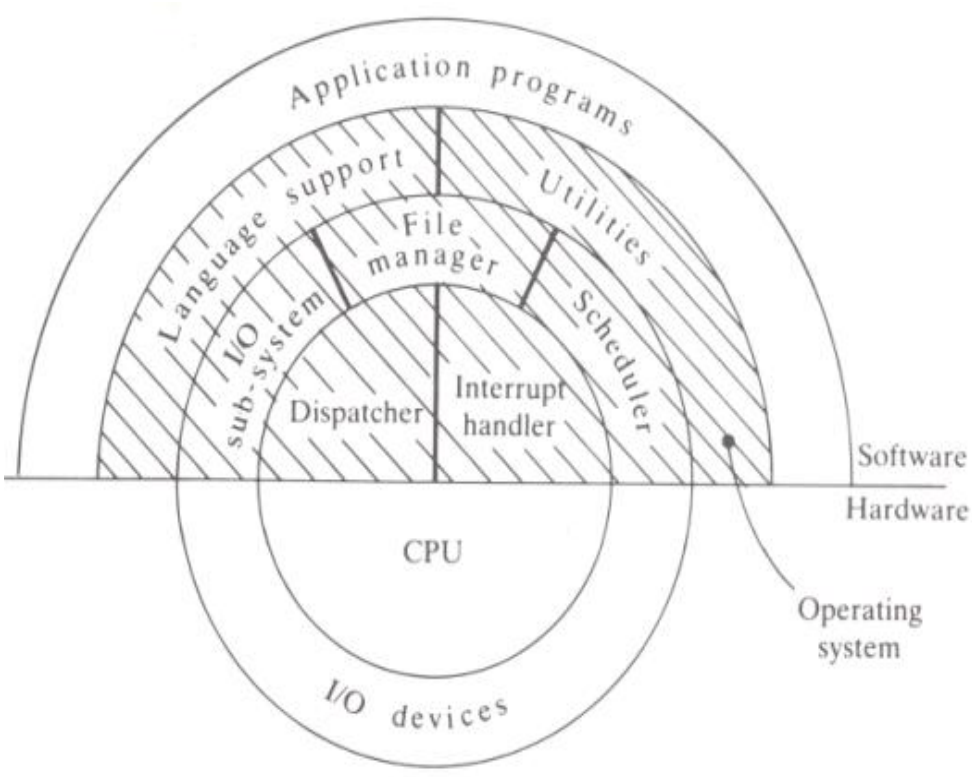
- It is the 1st layer of software loaded into memory when a system boots or starts up.
- It provides access to various services to all other system and application programs. These services include: disk access, memory management, task scheduling, and access to other hardware devices.

General Purpose Operating System:

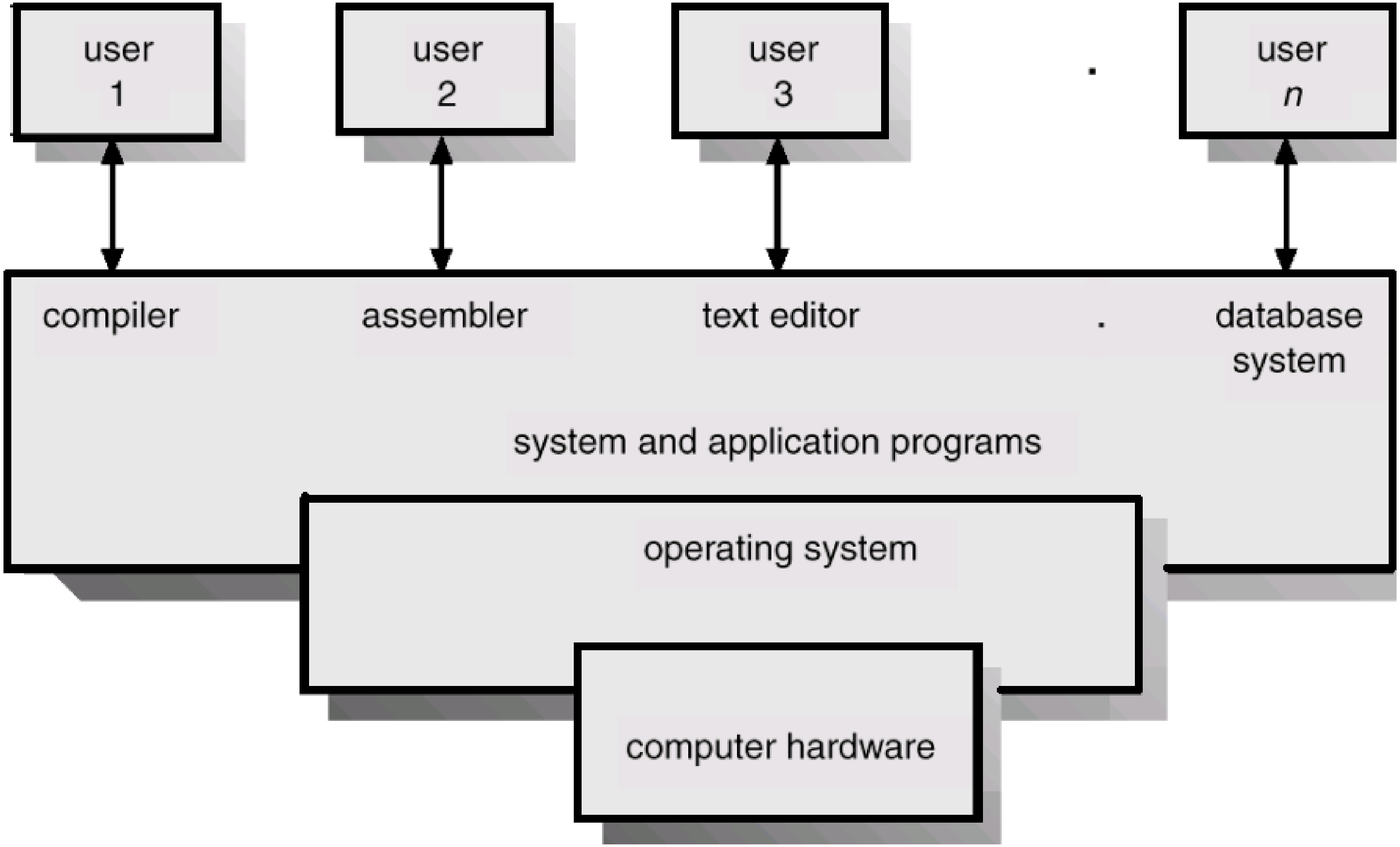
- OSs are used on most computer systems. The simplest computers, such as embedded systems did not have OSs. Instead, they rely on the application programs to manage the devices and with the help of libraries developed for this purpose.

Computer System Components:

1. **Hardware**; CPU, memory, I/O devices.
2. **OS**; controls and coordinates the use of the hardware among the various application programs for the various users.
3. **Applications Programs**; define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, programs)
4. Users; people, machines, others ….

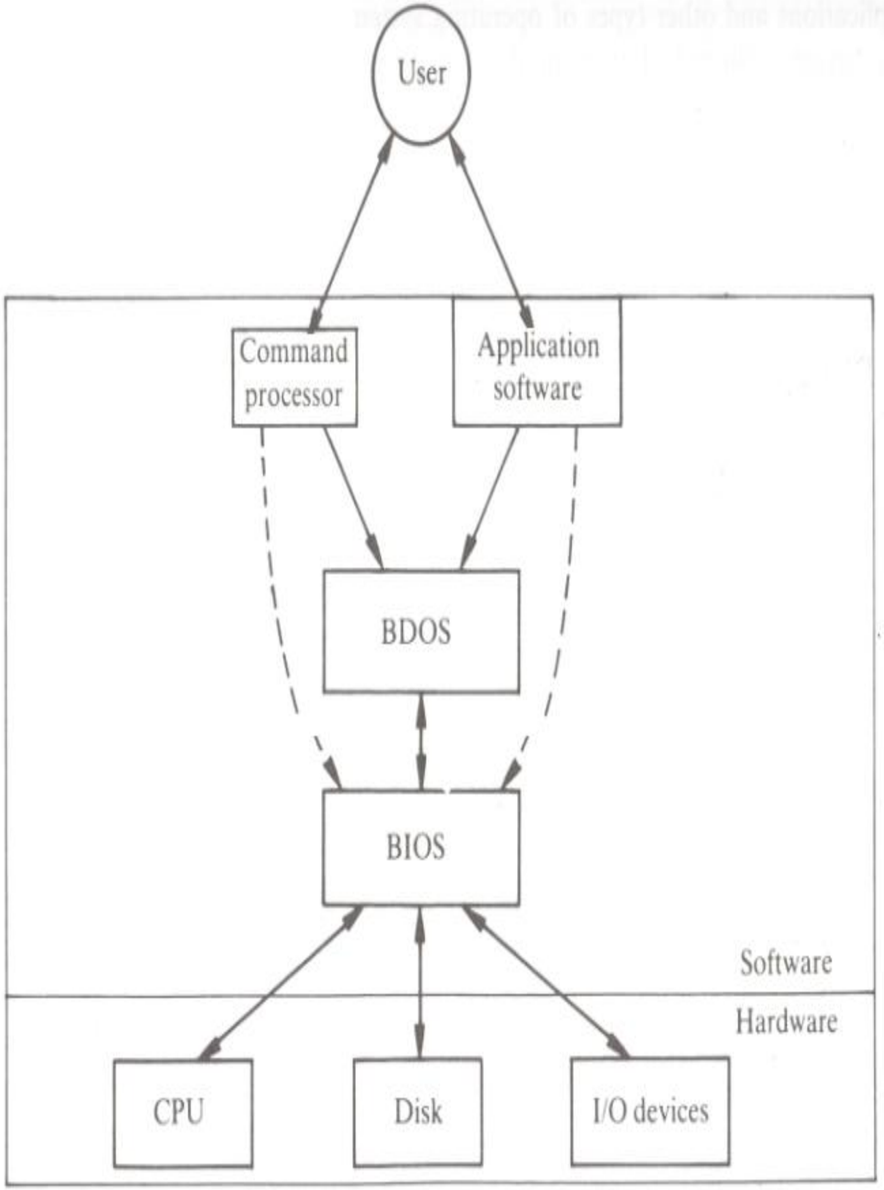


General purpose operating system


user
1
user
2
user
3
.
user
n
compiler
assembler
text editor
.
database
system
system and application programs
operating system
computer hardware

General Structure of a Simple OS:

- **Command Processor:** provides a means to communicate with the OS.
- **BDOS:** functions;
    - ➢ executes actual processing of the user commands.
    - ➢ handles the I/O and the file operations on the disks.
    - ➢ makes the actual management of the file and I/O operations transparent to the user.
- **Application Programs:** communicate with the hardware through *system calls* which are processed by the BDOS.
- **BIOS:** contains the various device drivers which manipulate the physical devices and OS.



General structure of a simple operating system

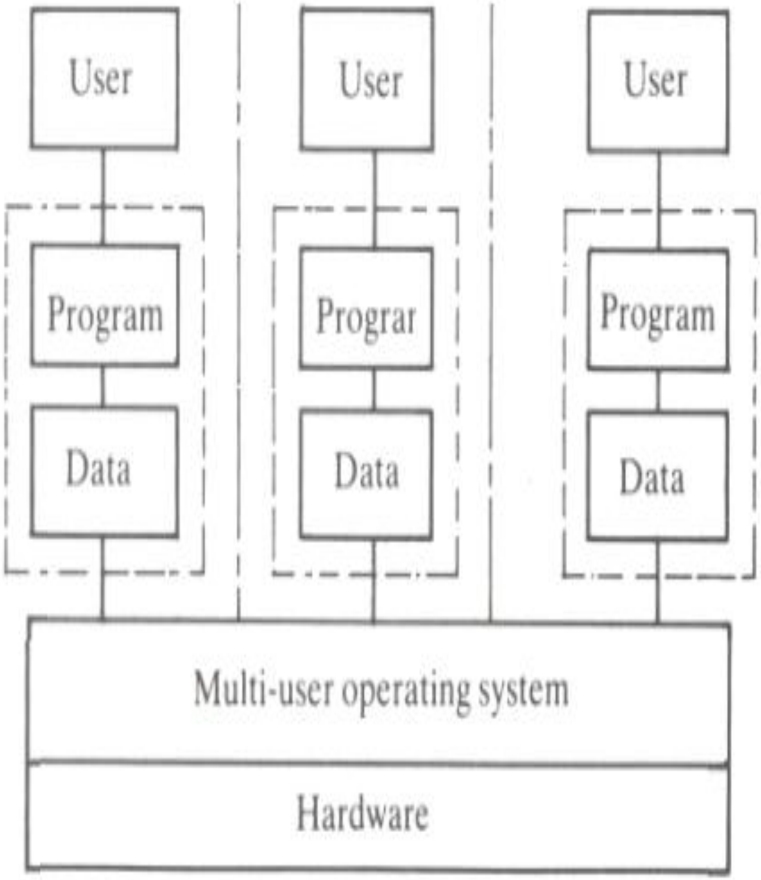
Types of Operating Systems:

There are different types of OSs:

- Single-user & Multi-user operating systems.
- Single-task & Multi-tasking operating systems.
- Real-time operating systems.

Multi-User Operating Systems:

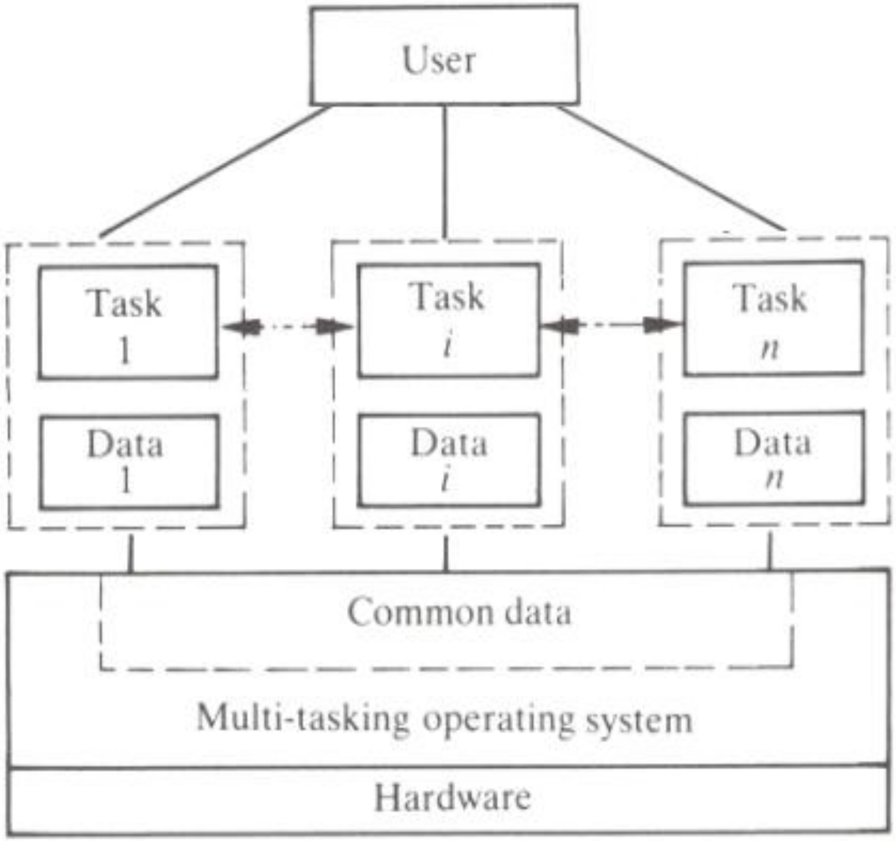
- The OS ensures that each user can run a single program as if the had the whole computer system.
- At any given instance, it is not possible to predict which user will have the use of the CPU.
- The OS ensures that one user program cannot interfere with the operation of another user program. Each user program runs in its own protected environment.



Multi-user operating system

Multi-Tasking Operating Systems:

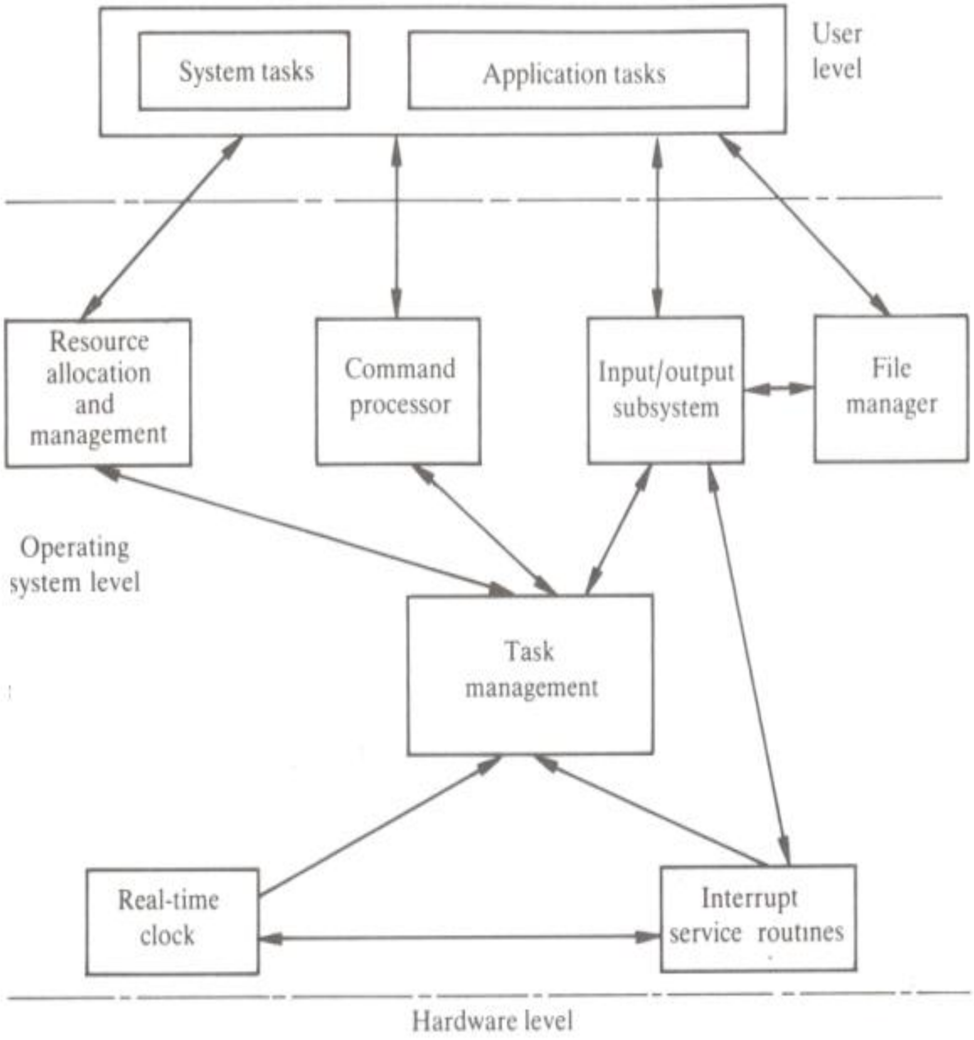
- In a multi-tasking operating system, it is assumed that there is a single user and that the various tasks co-operate to serve the requirements of the user.
- Co-operation requires that all tasks communicate with each other and share common data.
- Task communication and data sharing will be regulated so that the OS is able to prevent inadvertent communication or data access, and hence protect data which is private to a task.



Multi-tasking operating system

Multi-Tasking Operating Systems:

- A real-time multi-tasking operating system has to support the resource sharing and the timing requirements of the tasks. Functions of an OS can be divided as follows:
  - ➢ **Task Management:** the allocation of memory and processor time (scheduling) to tasks.
  - ➢ **Memory Management:** control of memory allocation.
  - ➢ **Intertask Communication & Synchronization:** provision of support mechanisms to provide safe communication between tasks and to enable tasks to synchronies their activities.



Typical structure of a real-time operating system

What is an RTOS?

An RTOS is a class of operating systems that are intended for real time-applications.

What is a real time application?

A real time application is an application that guarantees both correctness of result and the added constraint of meeting a deadline.

A **soft real-time system** is one where the response time is normally specified as an average value. This time is normally dictated by the business or market.

Ex: Airline reservation system.

A **hard real-time system** is one where the response time is specified as an absolute value. This time is normally dictated by the environment.

- A system is called a hard real-time if tasks always must finish execution before their deadlines or if message always can be delivered within a specified time interval.

The RTOS allows access to sensitive resources with defined response times:

- ➢ Maximum response times are good for hard real-time.
- ➢ Average response times are ok for soft real-time.

Any system that provides the above can be classified as a real-time system.

What makes an RTOS special?

- An RTOS will provide facilities to guarantee deadlines will be met.
- An RTOS will provide scheduling algorithms in order to enable deterministic behavior in the system.
- An RTOS is valued more for predictability than throughput.

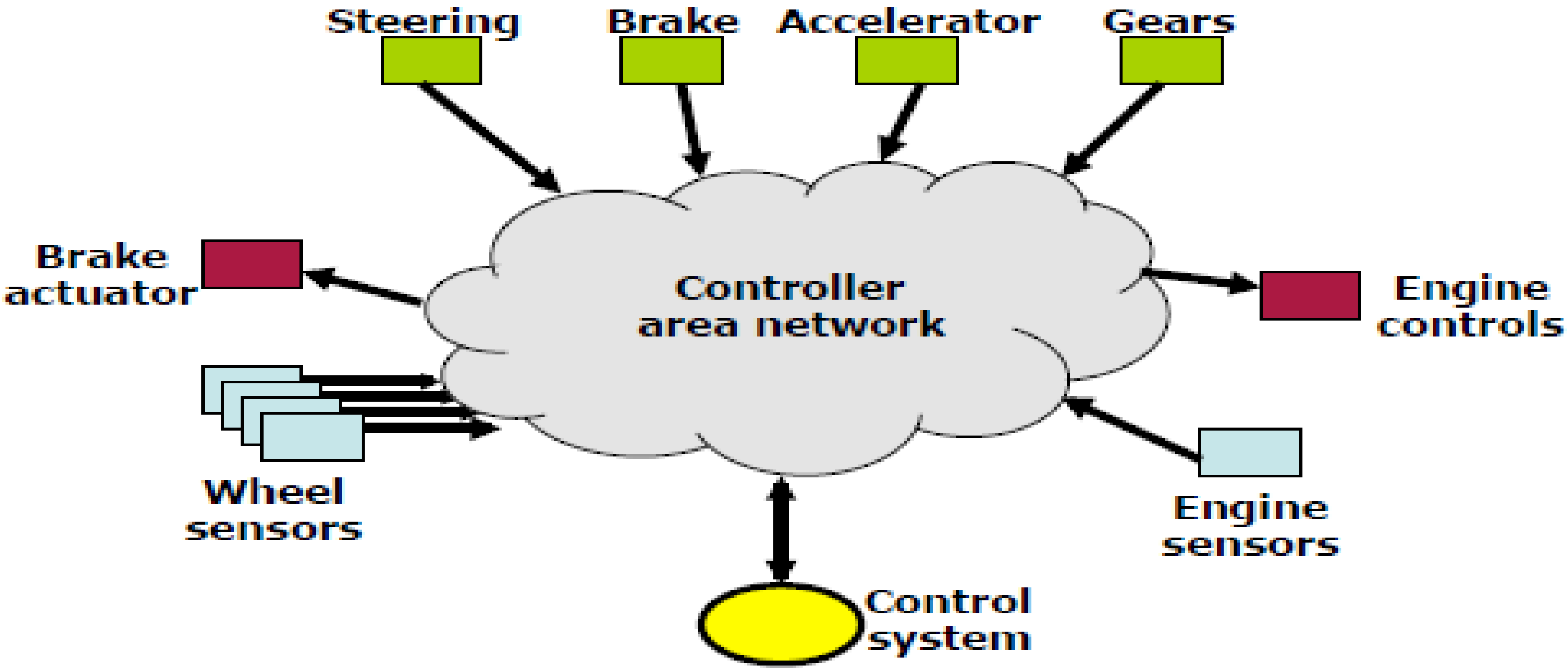
Design Philosophies of an RTOS?

Some of the design philosophies of an RTOS are with respect to:

- Scheduling
- Memory allocation
- Inter task communication
- Interrupt handlers

Example:

- ❑ All data must be delivered reliably
  – Bad if you turn the steering wheel, and nothing happens
- ❑ Commands from control system have highest priority, then sensors & actuators, then control inputs
  – Anti-lock brakes have a faster response time than the driver, so priorities to ensure the car doesn’t skid
- ❑ Network must schedule and priorities communications



Implementation Considerations:

Some real-time embedded systems are complex, implemented on high-performance hardware, such as;

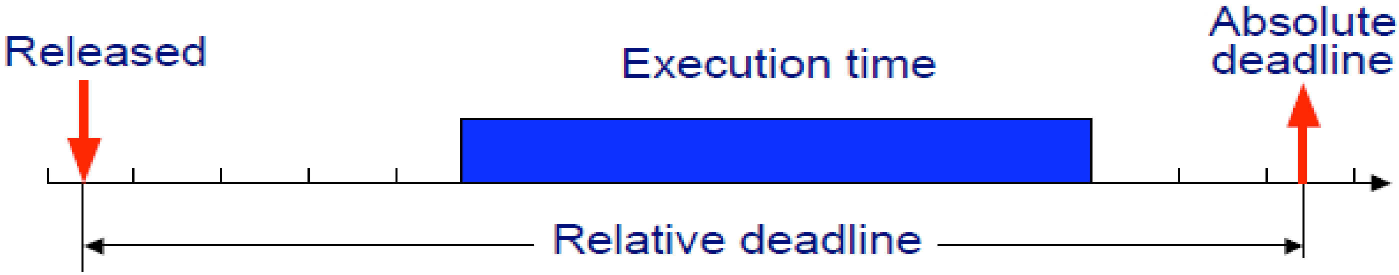
- Industrial plant control
- Civilian flight control

Many must be implemented on hardware chosen to be low cost, low power, light-weight and robust; with performance a distant concern

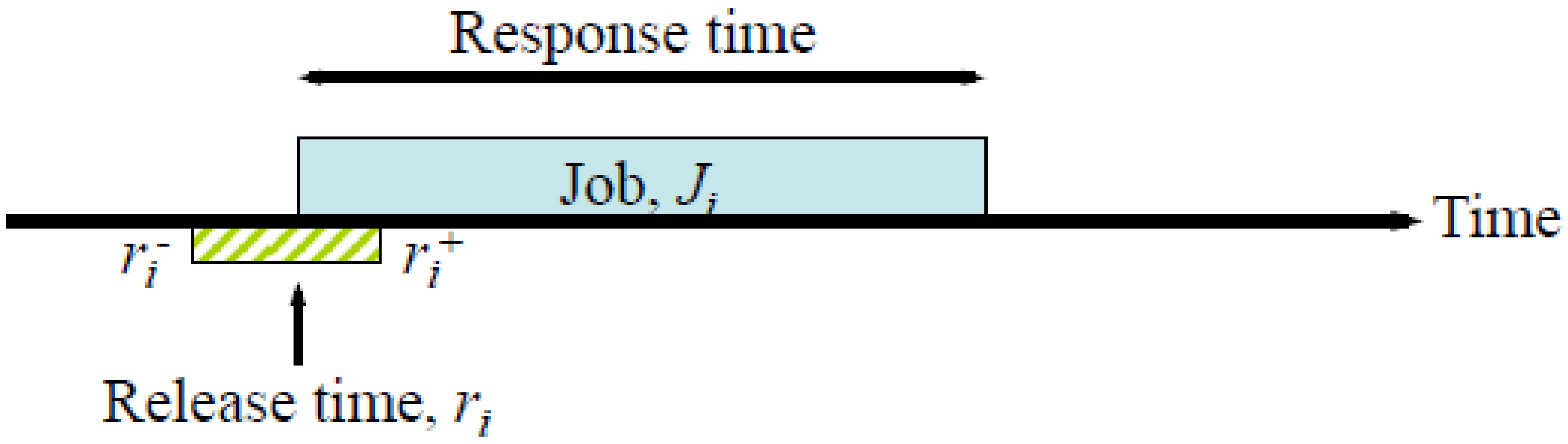
- Military flight control, space craft control
- Consumer goods

Execution Time:

- ➢ A job Ji will execute for time ei.
- ➢ This is the amount of time required to complete the execution of Ji when it executes alone and has all the resources it needs.
- ➢ Job execution time value (ei) depends upon complexity of the job and speed of the processor on which it is scheduled. This value may change for a variety of reasons:
    - Conditional branches.
    - Cache memories and/or pipelines.
- ➢ Execution times fall into an interval [ei-, ei+]; assume that we know this interval for every hard real-time job, but not necessarily the actual ei.



Release and Response Time:



Release time: the instant in time when a job becomes available for execution.

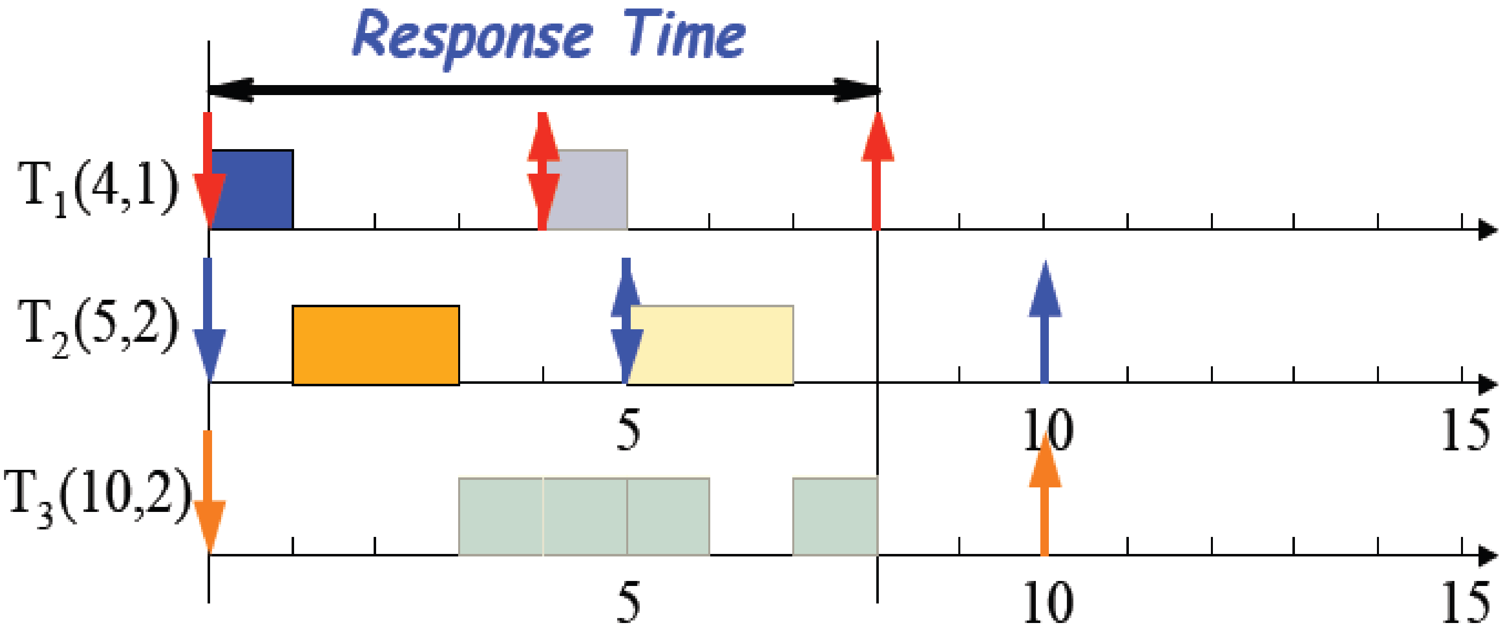
➢ May not be exact: Release time jitter so ri is in the interval [ri-, ri+]

➢ A job can be scheduled and executed at any time at, or after, its release time, provided its resource dependency conditions are met.

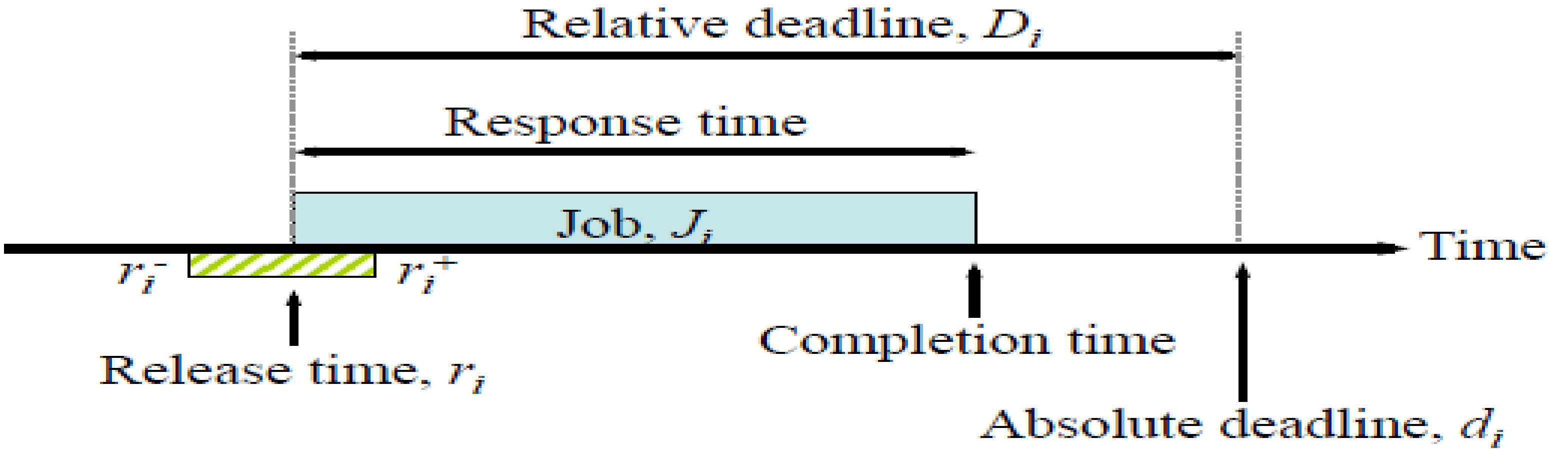
Response time: the length of time from the release time of the job to the time instant when it completes.

Response time = Duration from released time to finish time

➢ Not the same as execution time, since may not execute continually.



Deadlines and Timing Constraints:



Completion time: the instant at which a job completes execution.
Relative deadline: the maximum allowable job response time.
Absolute deadline: the instant of time by which a job is required to be completed (often called simply the deadline).

absolute deadline = release time + relative deadline

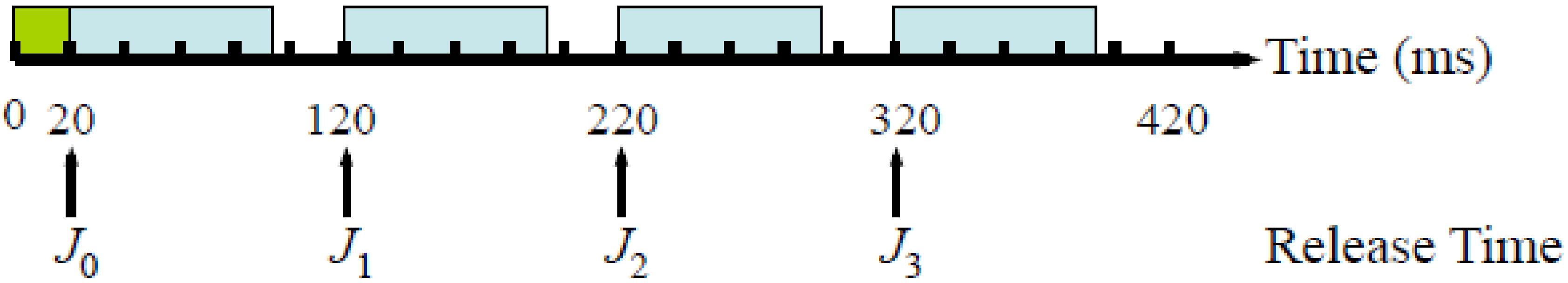
- Feasible interval for a job Ji is the interval (ri, di)
- Deadlines are examples of timing constraints.

Example: A system to monitor and control a heating furnace.

- The system takes 20ms to initialize when turned on.
- After initialization, every 100 ms, the system:
  - Samples and reads the temperature sensor
  - Computes the control-law for the furnace to process. temperature readings, determine the correct flow rates of fuel, air and coolant.
  - Adjusts flow rates to match computed values.

The periodic computations can be stated in terms of release times of the jobs computing the control-law: $J_0, J_1, \ldots, J_k, \ldots$

– The release time of J_k *is* 20 + (k × 100) ms



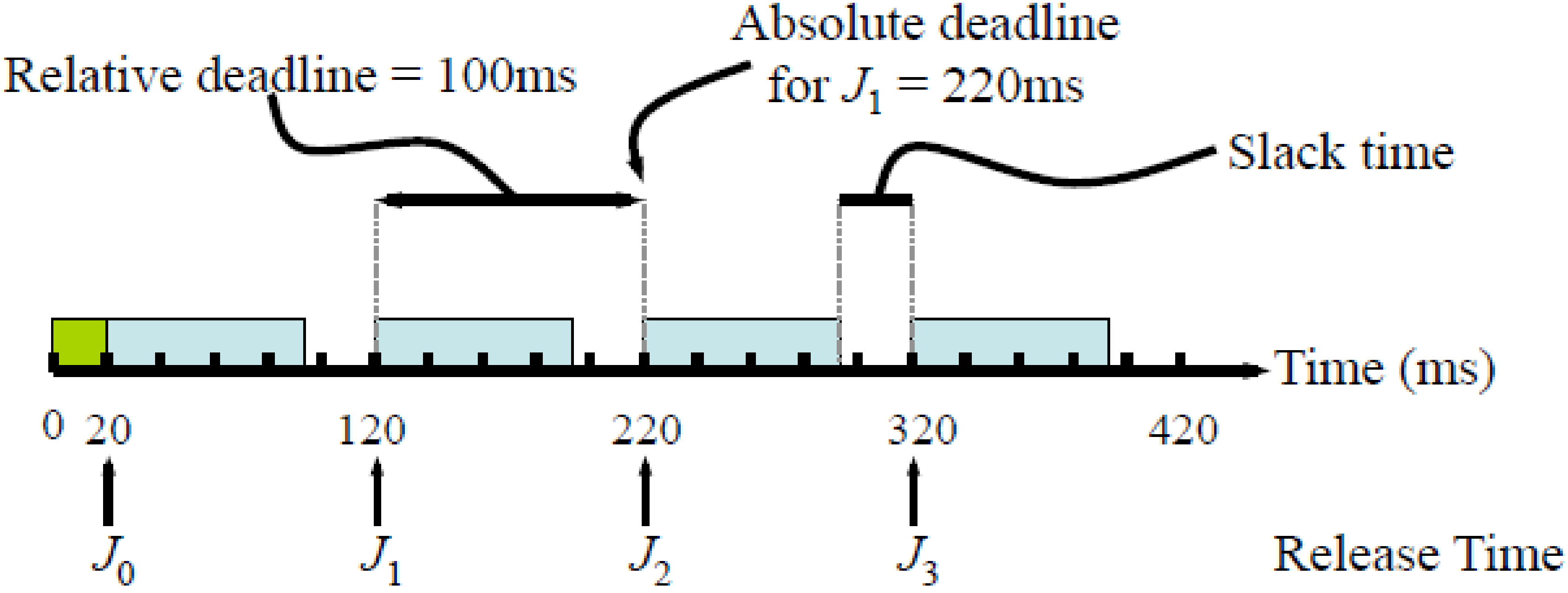
Example:

Suppose each job must complete before the release of the next job:

– Jk's relative deadline is 100 ms.

– Jk's absolute deadline is 20 + ((k + 1) × 100) ms.

Alternatively, each control-law computation may be required to finish sooner, i.e. the relative deadline is smaller than the time between jobs, allowing some slack time for other jobs.



Scheduling Strategies:

There are two basic strategies for the scheduling of time allocation on a single CPU;

1. Cyclic Strategy:

- ➢ The task uses the CPU for as long as it wishes.
- ➢ It is a very simple strategy which is highly efficient, it minimizes the time lost in switching between tasks.
- ➢ It is an efficient strategy for small embedded systems for which the execution times for each task run are carefully calculated and for which the software is carefully divided into appropriate task segments.
- ➢ This approach is too restrictive since it requires that the task units have similar execution times. It is difficult to deal with random events using this approach.

2. Pre-emptive Strategies:

- ➢ There are many pre-emptive strategies, all involve the possibility that a task will be interrupted before it has completed a particular invocation.
- ➢ The simplest form of pre-emptive scheduling is to use a time slicing approach.

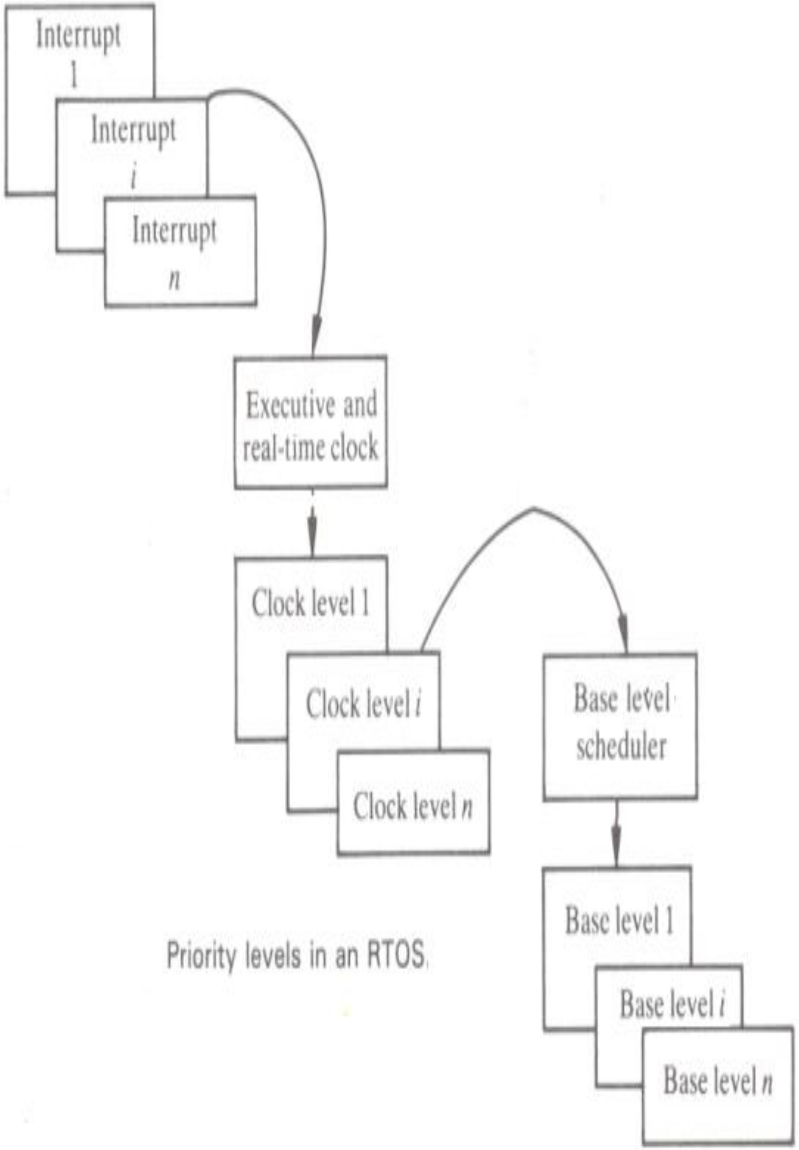
Priority Scheduling Mechanism:

- Tasks are allocated a priority level and at the end of a predetermined time slice, the task with highest priority of those ready to run is chosen and is given control of the CPU.
- Task priorities may be fixed (static priority system) or may be changed during system execution (dynamic priority system).
- Dynamic priority schemes can increase the flexibility of the system.
- Changing priorities is risky as it makes it much harder to predict and test the behavior of the system.
- The task management system has to deal with the handling of interrupts. These may be hardware interrupts caused by external events, or software interrupts generated by a running task.

Priority Structures:

- Assign priorities to the tasks in the system.
- The priority will depend on how quickly a task will have to respond to a particular event.
- In most RTOSs, tasks can be divided into three board levels:

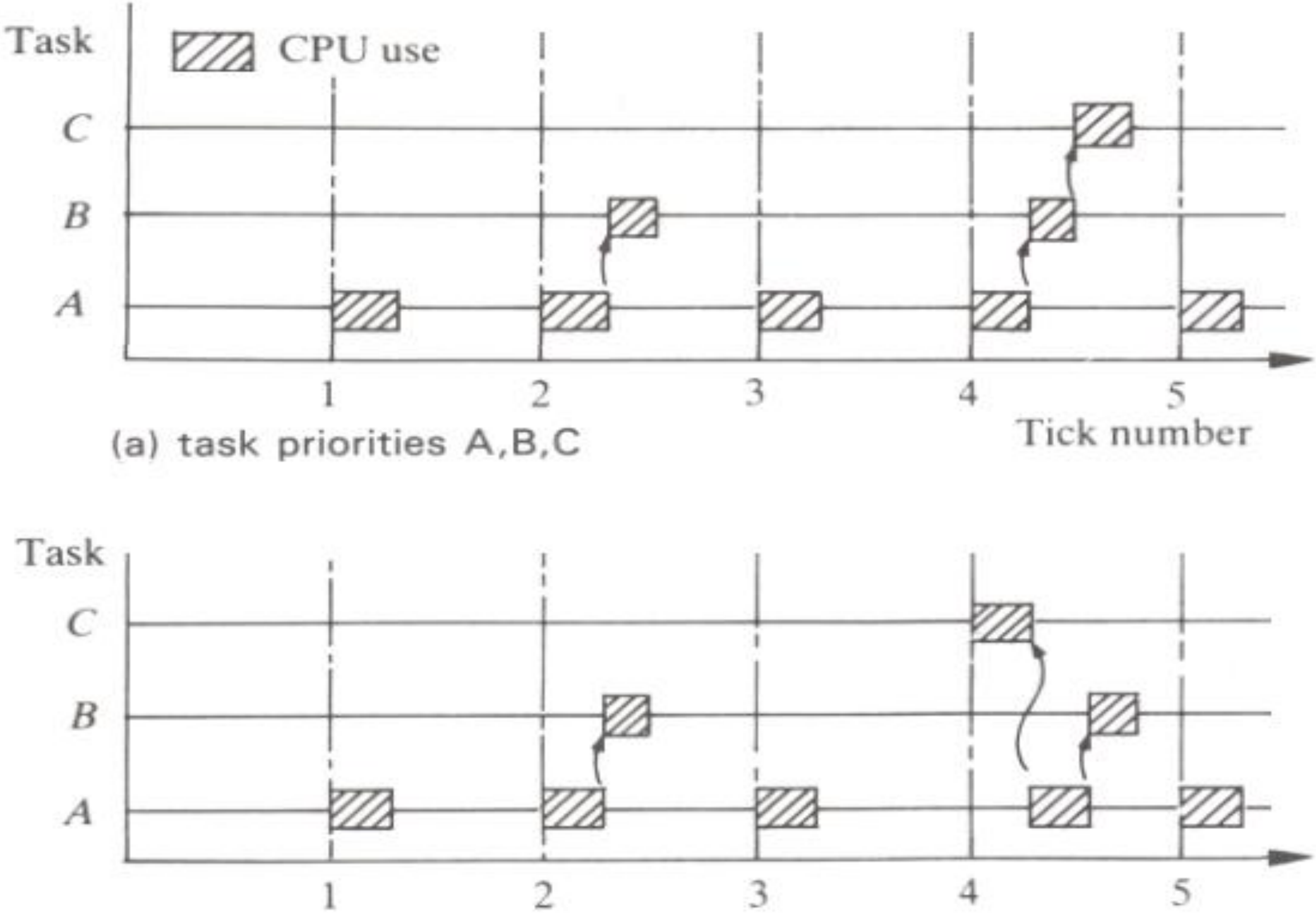
1. **Interrupt Level:** Service routines for the tasks and devices which require very fast response (measured in msec.) Example: real-time clock task.
2. **Clock Level:** Tasks with accurate timing and repetitive processing, such as the sampling and control tasks.
3. **Base Level:** tasks with low priority and either have no deadlines to meet or are allowed a wide margin of error in their timing. Tasks at this level may be allocated priorities or may all run at a single priority level.



Priority levels in an RTOS.

Clock Level:

- One interrupt level task will be the real-time clock.
- Typical values 1-200 msec.
- Each clock interrupt is known as a tick and represents the smallest time interval in the system.
- The function of the clock interrupt handling routine is to update the time of day clock in the system and to transfer control to dispatcher.
- The scheduler selects which task is to run at a particular clock rate.
- Clock level tasks divided into two categories;
  - **Cyclic:** these are tasks which require accurate synchronization with outside world.
  - **Delay:** these tasks simply wish to have a fixed delay between successive repetitions or to delay their activities for a given period of time.



(a) task priorities A,B,C

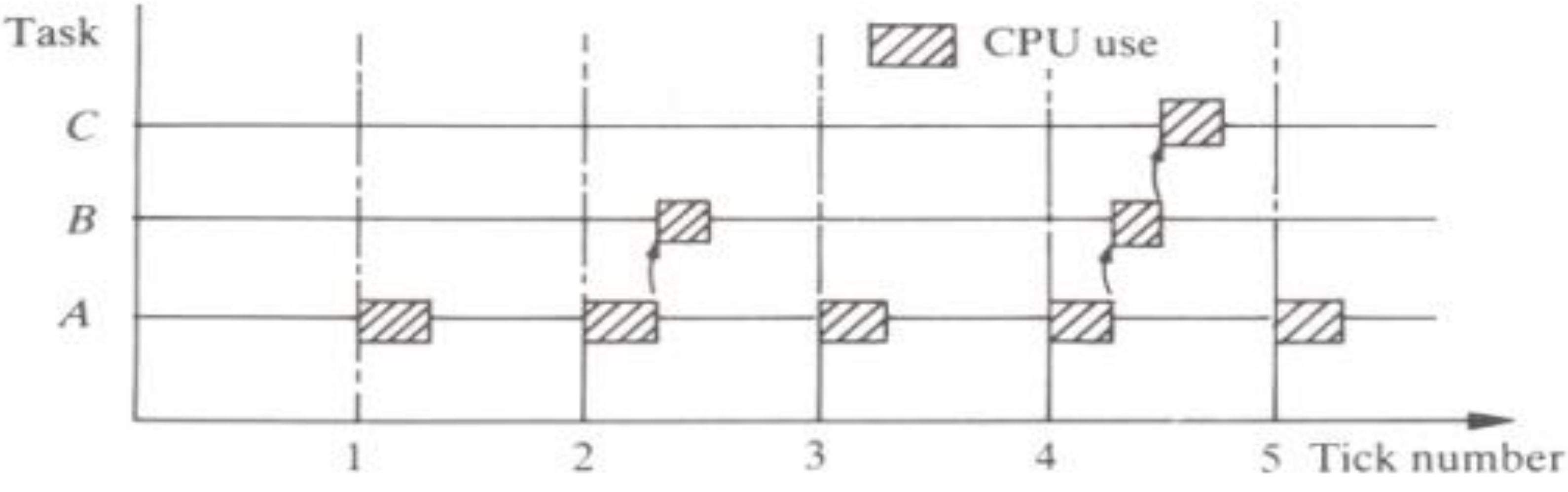
(b) task priorities C,A,B

Cyclic Tasks:

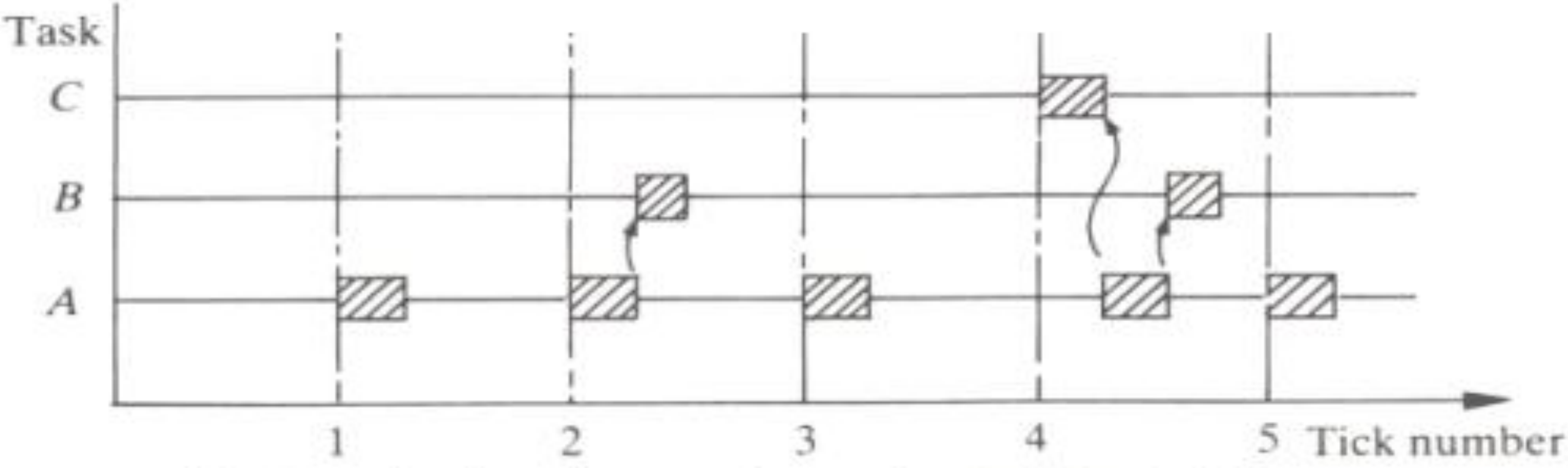
- Cyclic tasks are ordered in a priority which reflects the accuracy of timing required for the task, those which require high accuracy being given the highest priority
- Tasks of lower priority within clock level will have some jitter since they will have to await completion of the higher-level tasks.

Example:

- Three tasks A, B, and C are required to run at 20 msec, 40 msec and 80 msec intervals. If the clock interrupt rate is set at 20 msec, and the task priority order is set as A, B, and C with A as the highest priority. Then, thee following slid shows task activation diagram for this example in two cases;
  - ➢ Case (a): Task priorities are: A, B, then C.
  - ➢ Case (b): Task priorities are: C, A, then B.



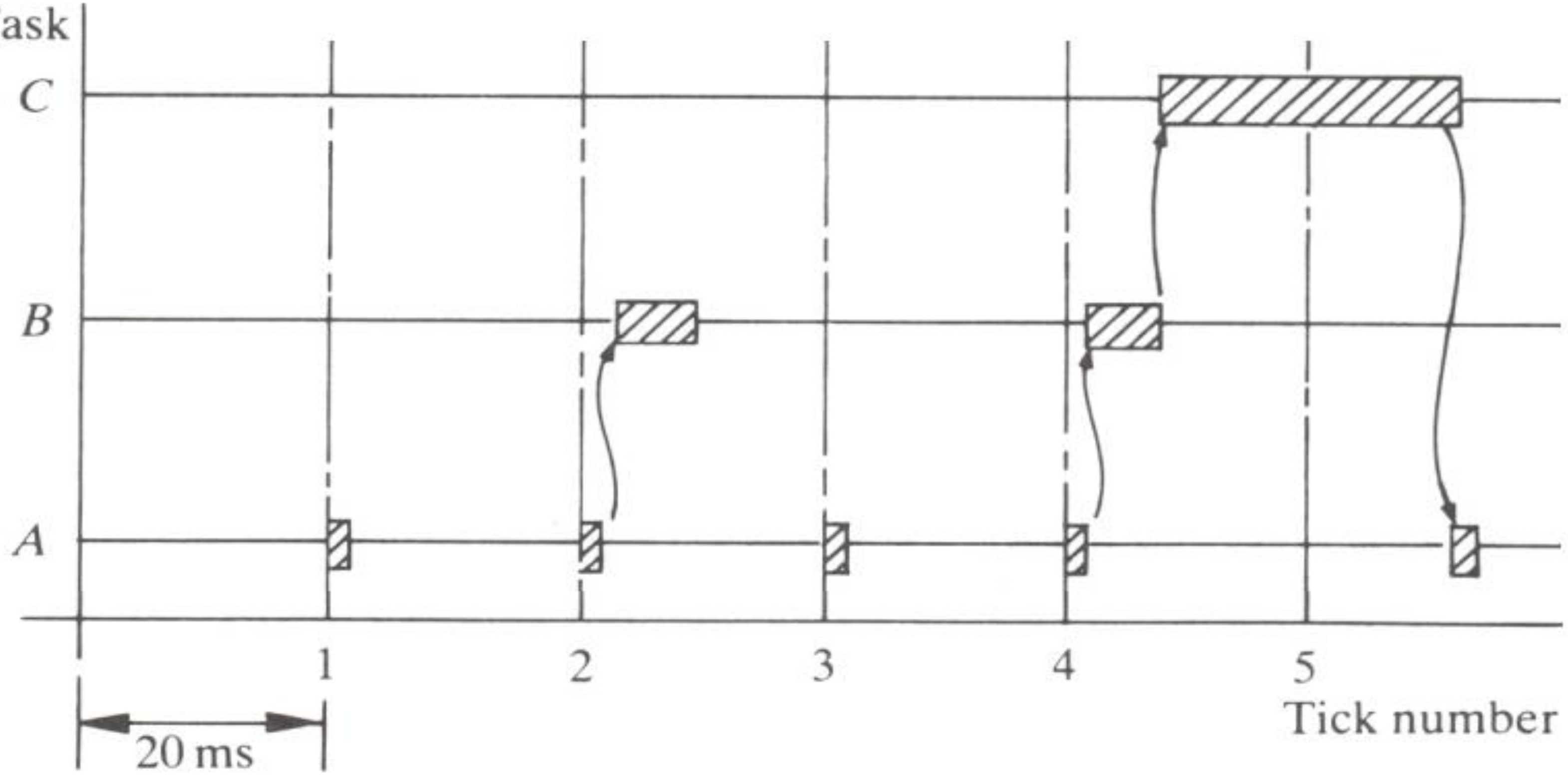
Task activation diagram for task priorities A,B,C·



Task activation diagram for task priorities C,A,B.

Example:

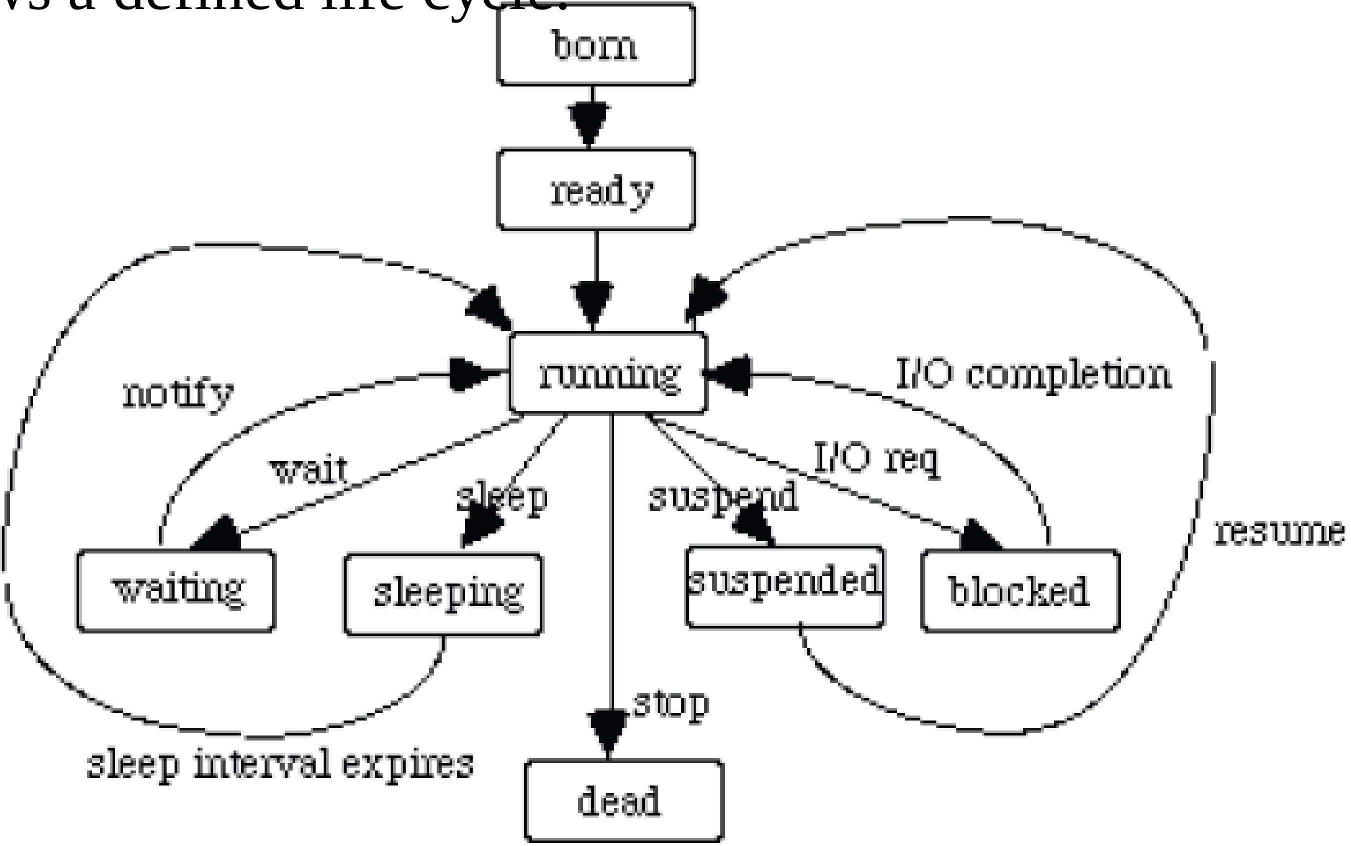
- Now assume that task C takes 25 msec to complete, task A takes 1 msec and task B takes 6 msec. if task C is allowed to run until completion then the activity diagram is given bellow.
- Task A will be delayed by 11 msec at every fourth invocation.



What is a Task?

Task is a sequence of similar jobs. Task is also called thread, is a user application.

- Shares the CPU and resources with other tasks.
- Follows a defined life cycle.

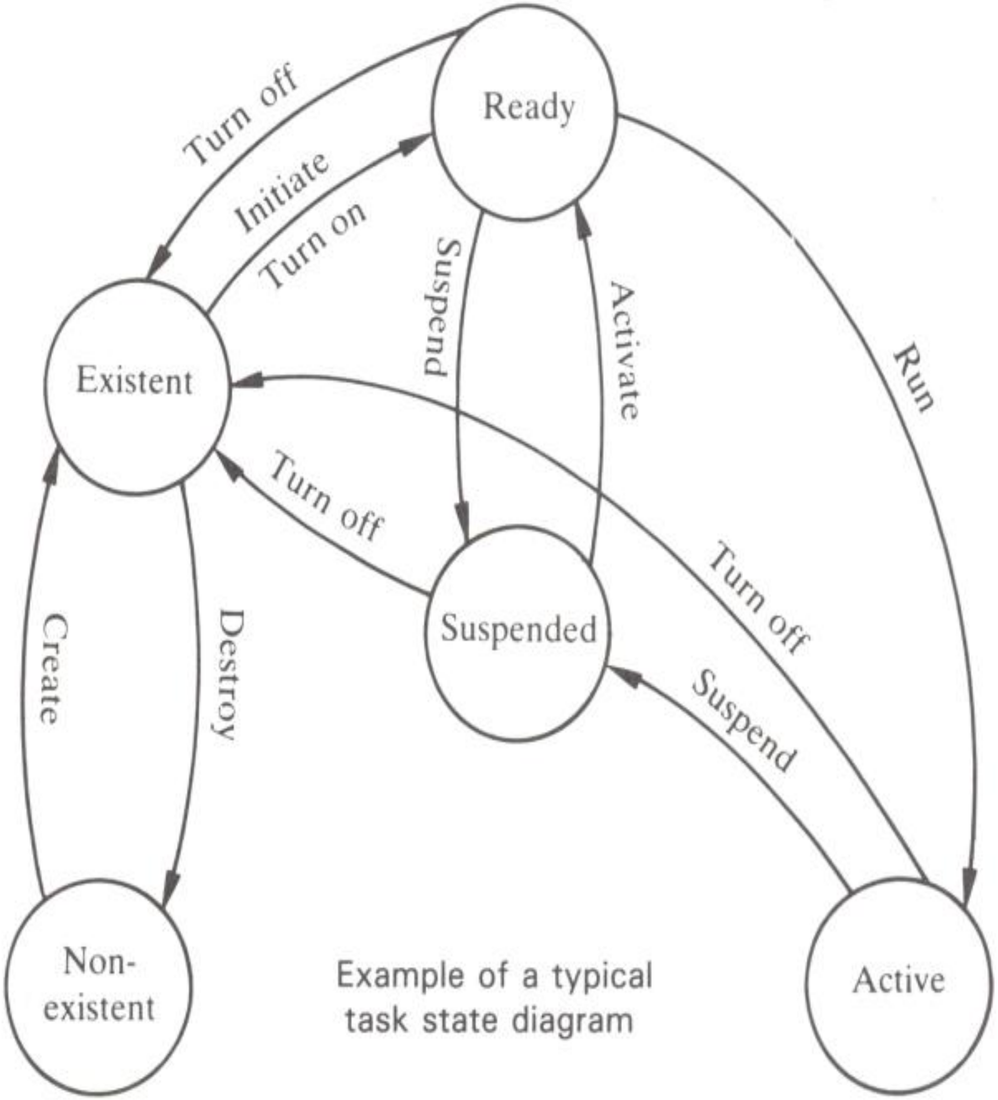


Task States:

Tasks are in one of four states:

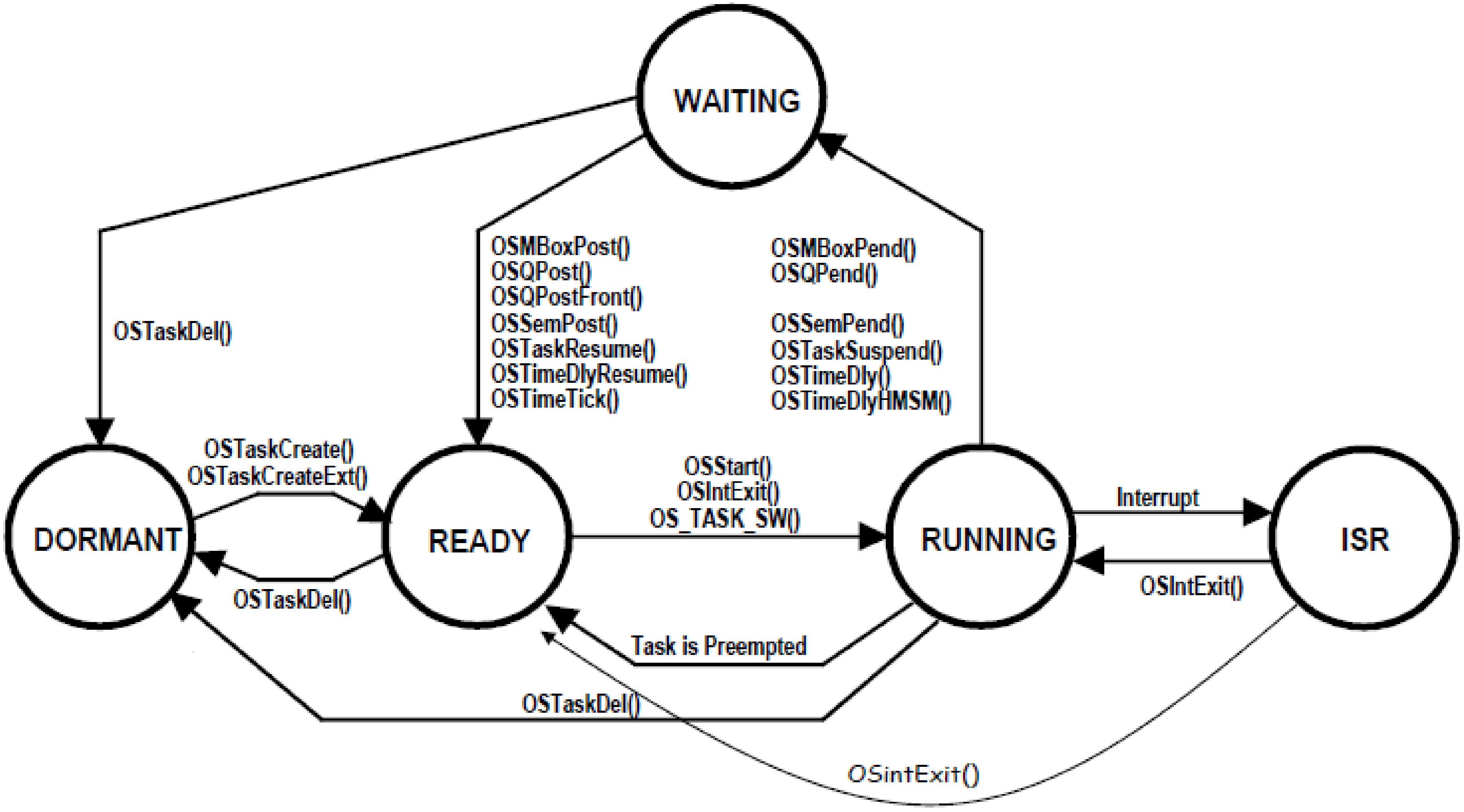
1. **Running**: a ready task is scheduled to run on the CPU.
2. **Ready**: task is neither delayed nor waiting for any event to occur.
3. **Waiting:** task is waiting for certain events to occur.
4. **Inactive:** procedures residing on RAM/ROM is not an task unless creating or calling to execute.

- Only one task can be Running at a time (unless "multi-core" CPU is used).



Example of a typical task state diagram

Task States:


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

Task Descriptor:

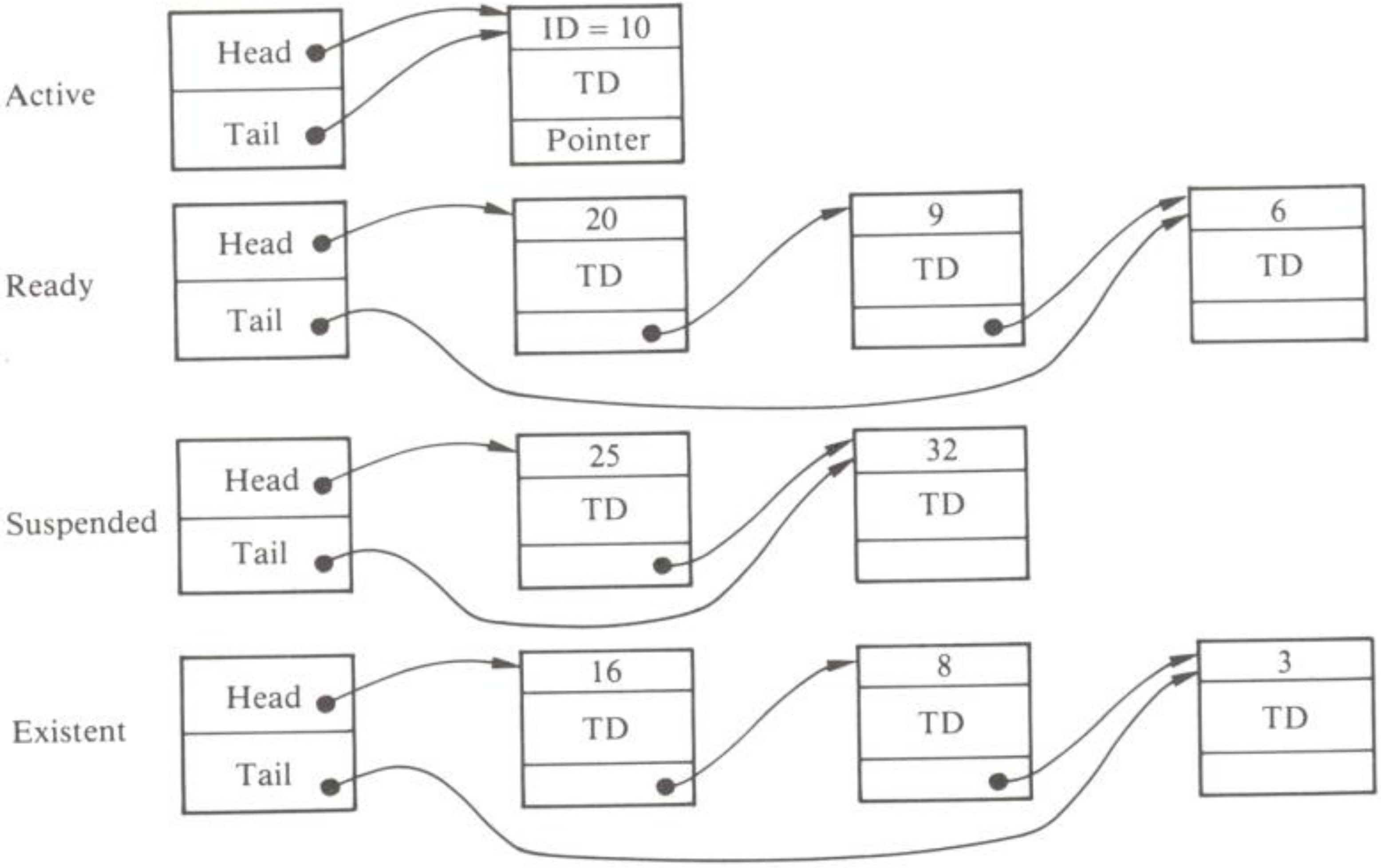
- Information about the status of each task is held in a block of memory by the RTOS. This block is called Task Descriptor (TD), or Task Control Block (TCB) or Task Data Control (TDC).
- The information is held in the TD will vary from system to system, but will typically consist of the following:
  - Task Identification.
  - Current state of task.
  - Task Priority.
  - Pointer to next task in list.
  - Area to store volatile environment (or a pointer to an area for storing the volatile environment).

Example:

The next slide shows list structure for holding task state information:

- There is one active task (task ID=10).
- There are 3 tasks ready to run (ID=20, ID=9 and ID=6). The entry held in the executive for the ready queue head points to task 20, which in tern points to task 9 and so on.
- The advantage of the list structure is that the actual TD can be located anywhere in the memory and hence the OS is not restricted to a fixed number of tasks as the case in older OSs which used fixed length tables to hold task state information.

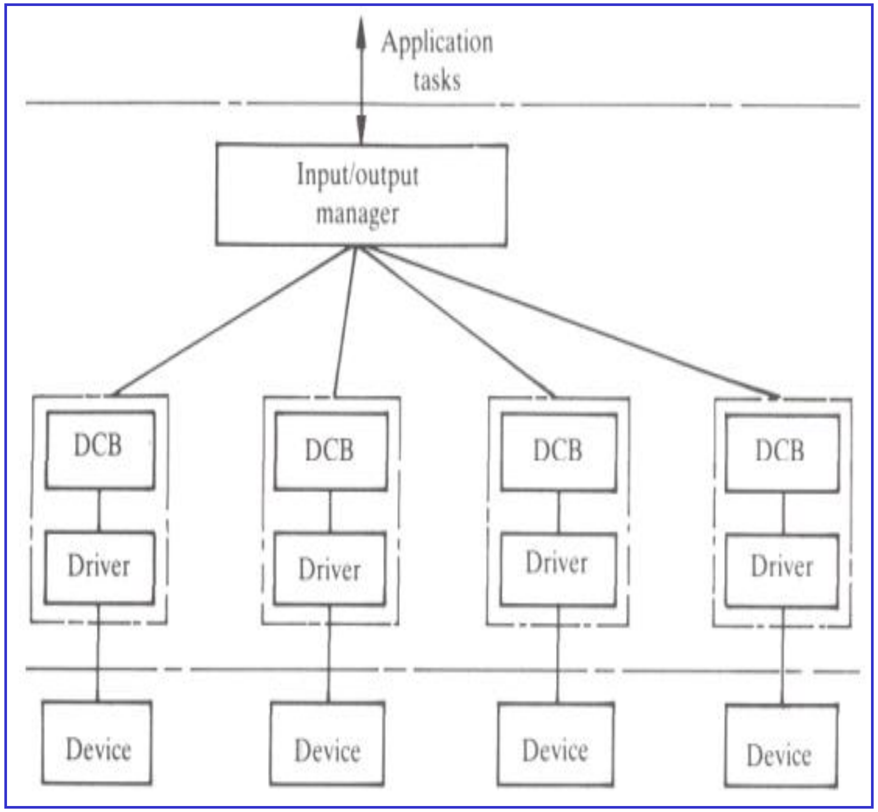
List structure for holding task state information:



List structure for holding task state information

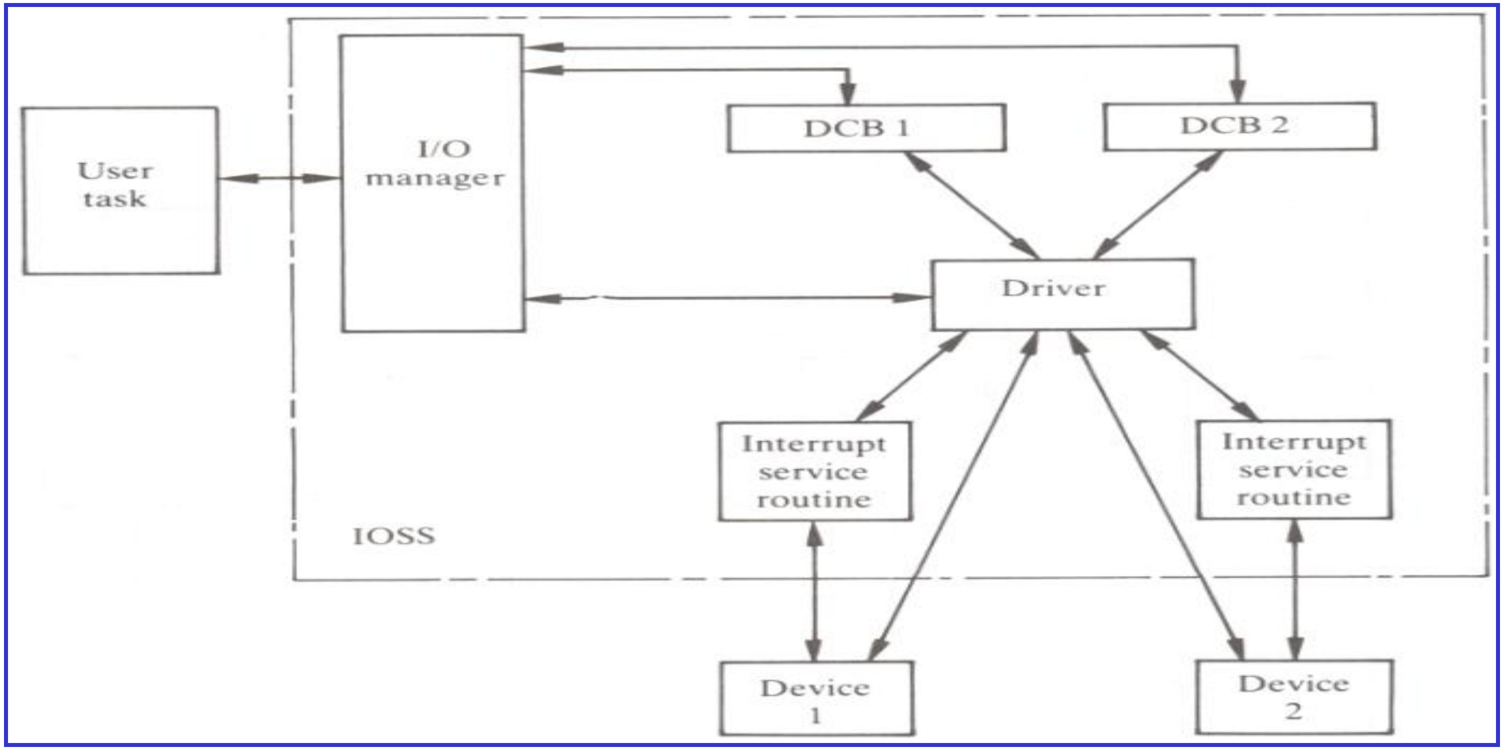
Resource Control:

- The information transfer with external devices is not an easy task in programming.
- The availability of an I/O subsystem (IOSS) in an OS is essential for efficient programming. This enables programmer to perform I/O by means of system calls. The IOSS handles all the details of the devices.
- A typical IOSS will be divided into two levels; I/O Manager, and DCB.
- The **I/O manager** accepts the system calls from the user tasks and transfers its information to the **Device Control Block (DCB)** for the particular device.
- The information supplied in the call by the user task will be;
  - ➢ the location of a buffer area in which the data to be transferred is stored .
  - ➢ the amount of data to be transferred.
  - ➢ type of data.
  - ➢ direction of transfer, and
  - ➢ the device to be used.



Detailed Arrangement of IOSS:

1. The actual transfer of the data between the user task and the device will be carried out by the device driver and this segment of code will make use of other information stored in the DCB.
2. A separate device driver may be provided for each device.
3. A single driver may be shared between several devices, however, each device will require its own DCB.
4. The OS will normally be supplied with DCBs for the more common devices.



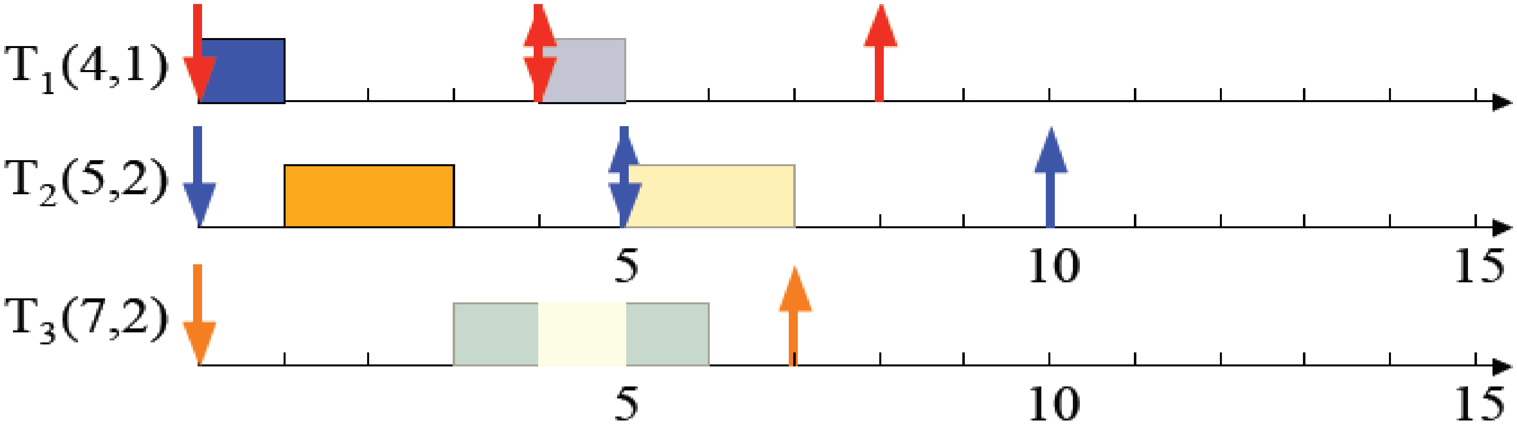
Task Scheduling:

Static scheduling: a fixed schedule is determined statically.

Static-priority scheduling:

– Assign fixed priorities to tasks.

– A scheduler only needs to know about priorities.



Dynamic-priority scheduling:

– Assign priorities based on current state of the system.

– For example: Least Completion Time (LCT), Earliest Deadline First (EDF), Least Slack Time (LST)

Real-Time Scheduling Algorithms:

Different classes of scheduling algorithm used in real-time systems:

1. **Clock-driven Scheduling:** used for hard real-time systems where all properties of all jobs are known at design time, such that offline scheduling techniques can be used
2. **Weighted Round-robin Scheduling:** used for scheduling real-time traffic in high-speed, switched networks
3. **Priority-driven Scheduling:** used for more dynamic real-time systems with a mix of clock-based and event-based activities, where the system must adapt to changing conditions and events

1. Clock-driven Scheduling:

- Specific time instants are chosen before the system begins execution.
- Usually implemented using a periodic timer interrupt.
- Scheduler awakes after each interrupt, schedules the job to execute for the next period, then blocks itself until the next interrupt.
- Typically in clock-driven systems:
  - All parameters of the real-time jobs are fixed and known.
  - A schedule of the jobs is computed off-line and is stored for use at runtime.
  - Simple and straight-forward, not flexible.

2. Weighted Round-Robin Scheduling:

Regular round-robin scheduling is used for scheduling time-shared applications.

- Every job joins a FIFO queue when it is ready for execution
- When the scheduler runs, it schedules the job at the head of the queue to execute for at most one time slice.
- If the job has not completed by the end of its quantum, it is preempted and placed at the end of the queue.
- When there are n ready jobs in the queue, each job gets one slice every n time slices (n time slices is called a round).
- Only limited use in real-time systems.

2. Weighted Round-Robin Scheduling:

In weighted round robin;

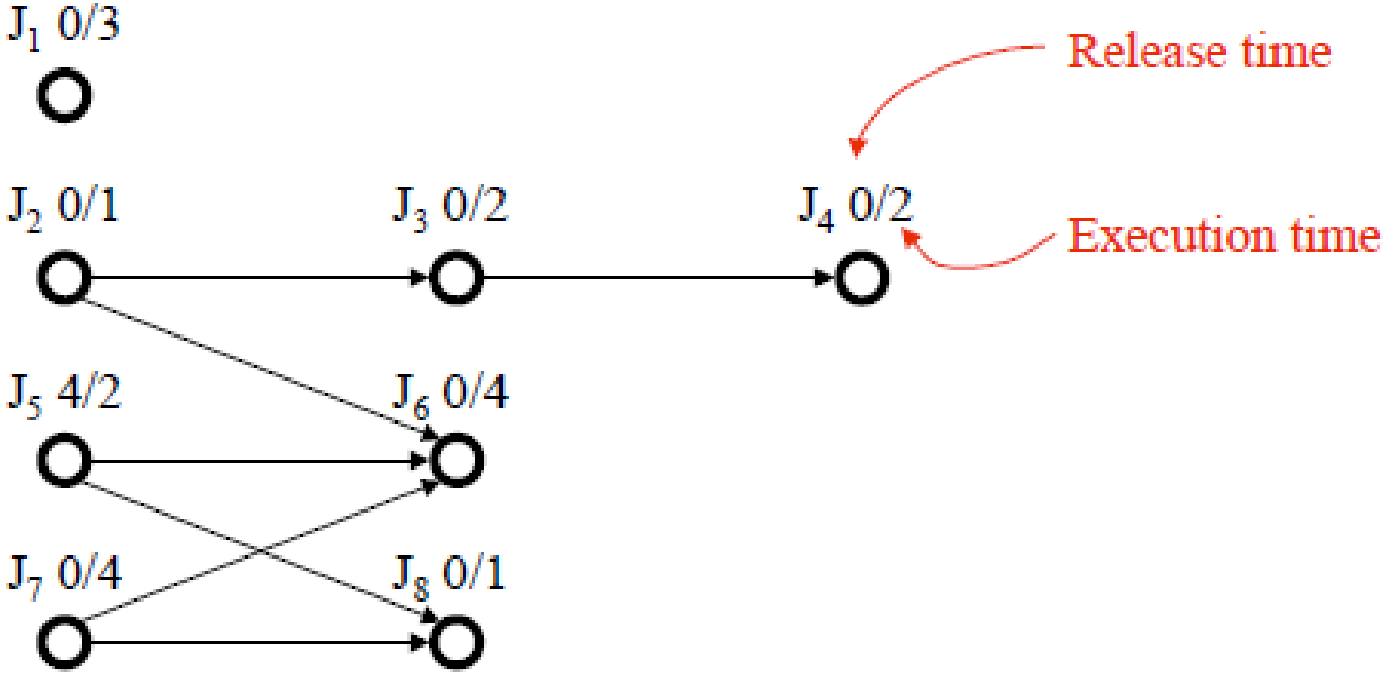
- Each job J_i is assigned a weight w_i.
- The job will receive w_i consecutive time slices each round, and the duration of a round is $\sum w_i$.
- Equivalent to regular round robin if all weights equal 1.
- Simple to implement, since it doesn’t require a sorted priority queue.
- Offers throughput guarantees, where each job makes a certain amount of progress each round.
- By giving each job a fixed fraction of the processor time, a round robin scheduler may delay the completion of every job.
- Weighted round-robin is primarily used for real-time networking.

3. Priority-Driven Scheduling:

- Assign priorities to jobs, based on some algorithm.
- Make scheduling decisions based on the priorities, when events such as releases and job completions occur.
- Priority scheduling algorithms are event-driven.
- Jobs are placed in one or more queues; at each event, the ready job with the highest priority is executed.
- The assignment of jobs to priority queues, along with rules such a whether preemption is allowed, completely defines a priority scheduling algorithm.
- Priority-driven algorithms make locally optimal decisions about which job to run.

Example: Consider the following task which has eight jobs:

- Jobs J1, J2, …, J8, where Ji had higher priority than Jk if i < k.
- Jobs are scheduled on two processors (P1 and P2).
- Jobs communicate via shared memory, so communication cost is negligible.
- The schedulers keep one common priority queue of ready jobs.
- All jobs are preemptable; scheduling decisions are made whenever some job becomes ready for execution or a job completes.



Priority-Driven Scheduling:

Most scheduling algorithms used in non real-time systems are priority-driven;

- ➢Assign priority based on release time, such as:
  - – First-In-First-Out
  - – Last-In-First-Out
- ➢Assign priority based on execution time, such as:
  - – Shortest-Execution-Time-First
  - – Longest-Execution-Time-First

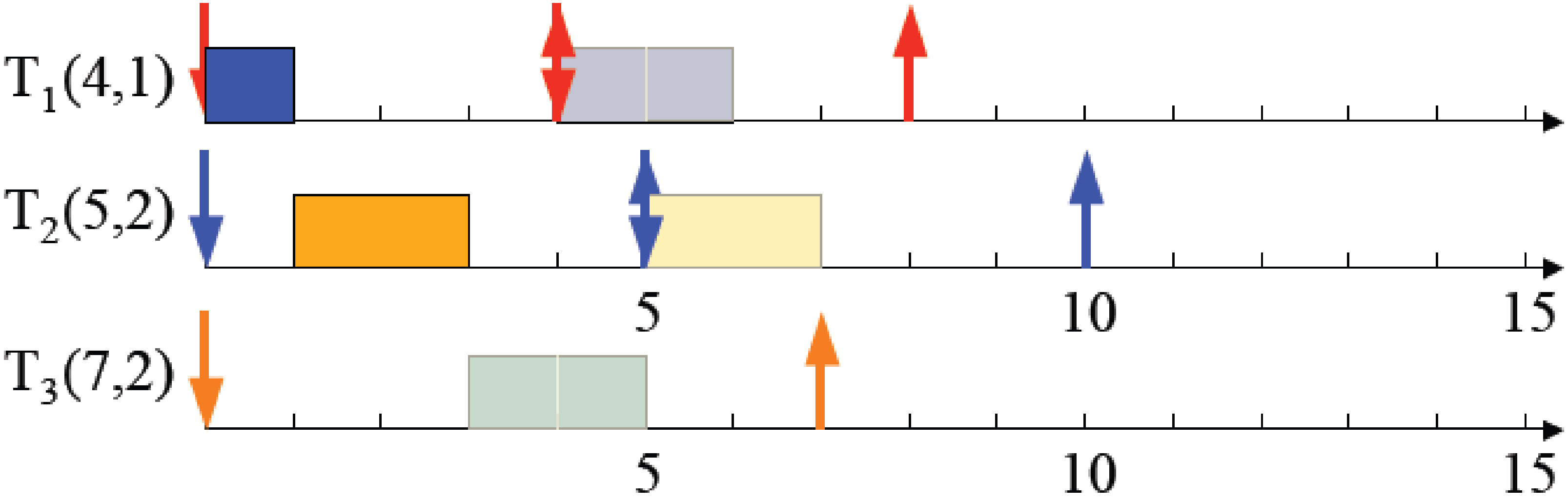
Real-time priority scheduling assigns priorities based on deadline or some other timing constraint:

- – Earliest deadline first
- – Least slack time first

Priority Scheduling Based on Deadlines:

1. Earliest Deadline First (EDF):

- Optimal dynamic priority scheduling
- A task with a shorter deadline has a higher priority
- Executes a job with the earliest deadline



Priority Scheduling Based on Deadlines:

2. Least Slack Time First (LST)

- A job J_i has deadline d_i, execution time e_i, and was released at time r_i.
- At time $t < d_i$:
  - Remaining execution time $t_{rem} = e_i - (t - r_i)$
  - Slack time $t_{slack} = d_i - t - t_{rem}$
- Assign priority to jobs based on slack time, t_{slack}.
- The smaller the slack time, the higher the priority.
- More complex, requires knowledge of execution times and deadlines.

Open-Source RTOS:

There are many RTOSs available for use today, both commercial and experimental, and new ones are still being developed. This is mainly due to:

- There is much research in the field of embedded systems,
- Real-time embedded applications are inherently less homogeneous than general purpose applications, and
- The computing power and the overall hardware architecture of embedded systems are much more varied than those of PCs.

Open-source OSs are especially promising, for two main reasons:

1. The source code of an open-source RTOS can be used both to develop real-world applications, and for study and experimentation.
2. Open-source RTOSs have no purchasing cost, so their adoption can cut down the costs of an application.

For more information:

1. R. Zurawski (ed). *The Industrial Communication Systems Handbook. CRC Press, 2005.*
2. J. Liu. *Real-Time Systems. Prentice-Hall, 2000.*
3. Kopetz H.. *Real-Time Systems: Design Principles for Distributed Embedded Applications. Kluwer Academic,* Publishers, 1997.
4. http://www.cs.ou.edu/~fagg/umass/classes/377f02/lectures.html
5. http://www.cs.umbc.edu/~younis/Real-Time/CMSC691S.html