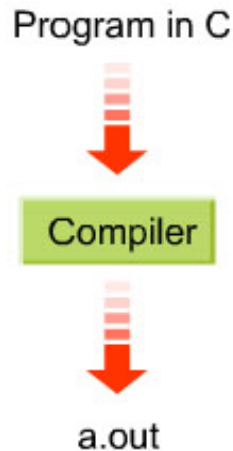


# CPU Scheduling

# WHAT IS A PROCESS?

- a process is a *program in execution*.

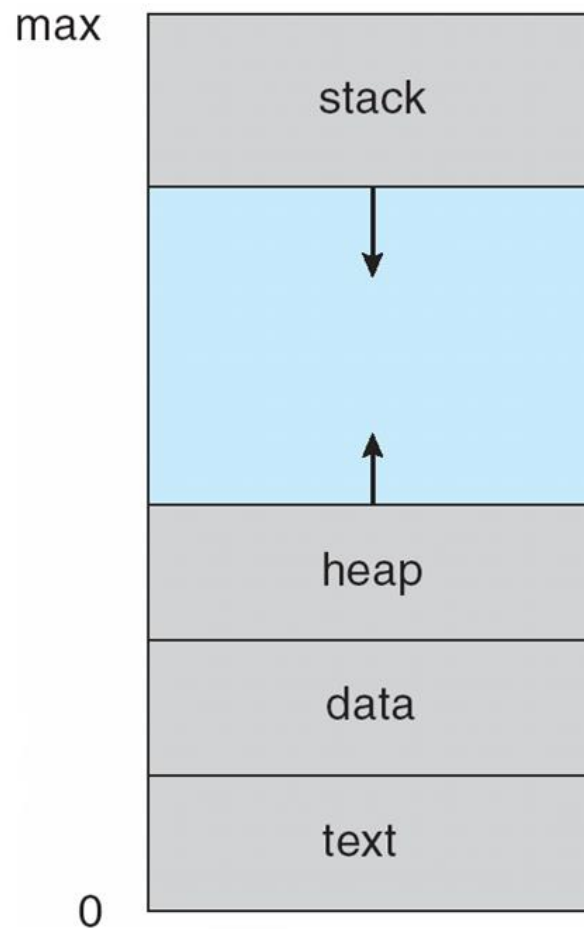


- Recipe of some menu => program
- Baking oven is processor, spices => resources
- food in preparation => process in execution

# Process in Memory

- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

# Process in Memory



# Process Control Block (PCB)

Information associated with each process  
(also called **task control block**)

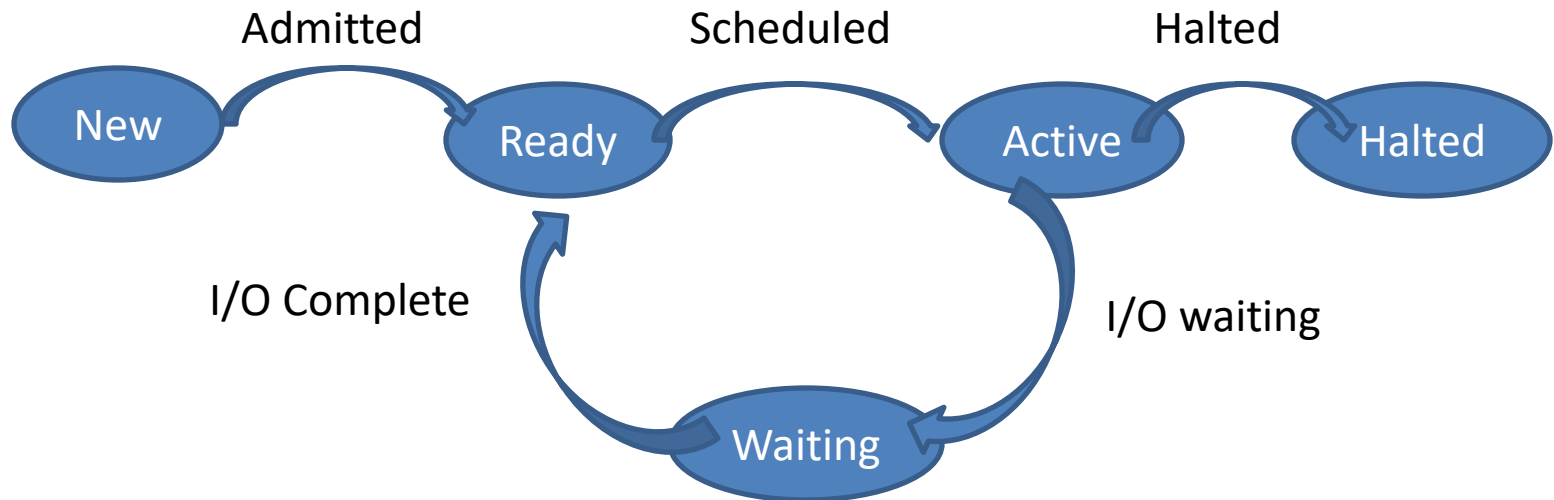
- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files

Process ID
State
Pointer
Priority
Program counter
CPU registers
I/O information
Accounting information
etc....

# Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **Active/running**: Instructions are being executed
  - **Halted/terminated**: The process has finished execution
  - **waiting**: The process is waiting for some event to occur

# Process State

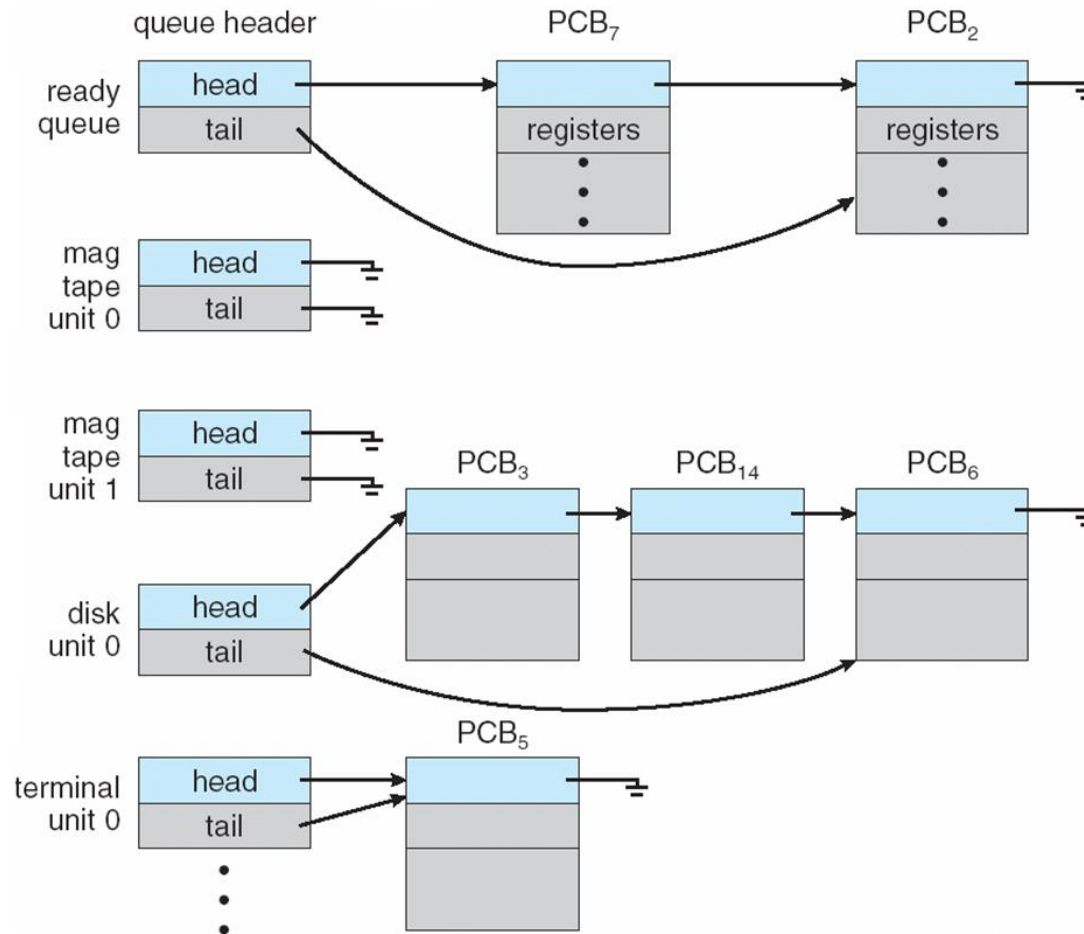


# Different Queues in Process Scheduling

- OS maintains various **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues



# Ready Queue And Various I/O Device Queues



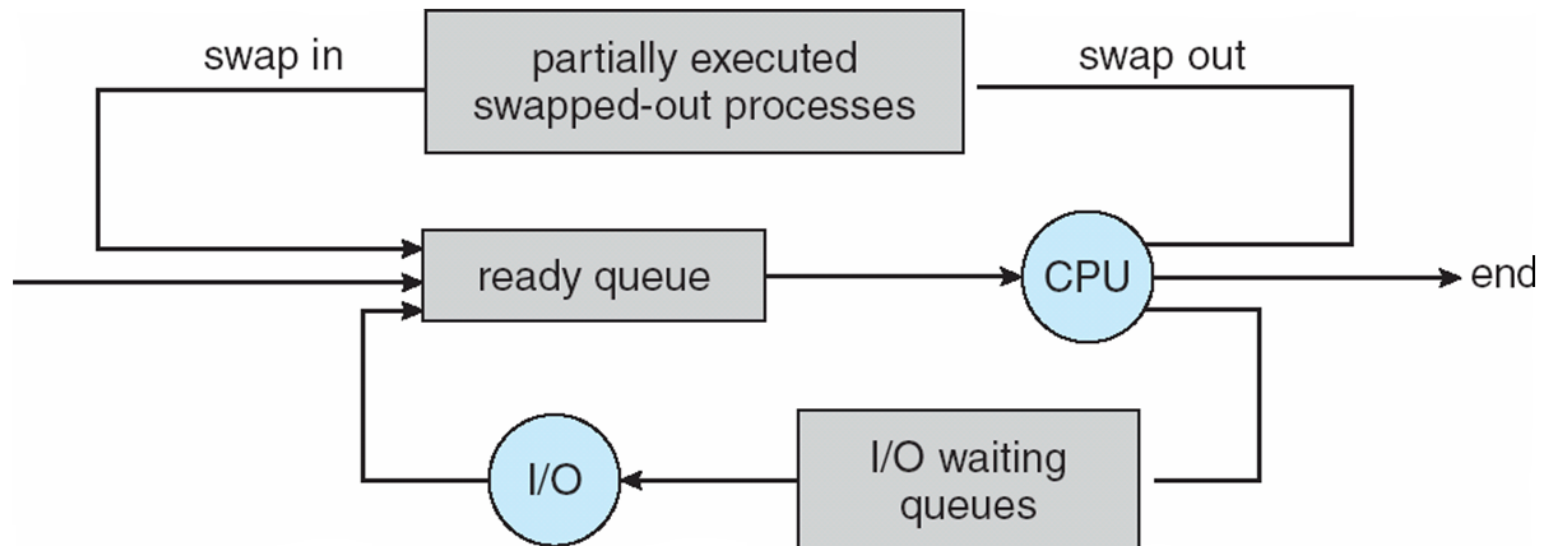
# Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds)  $\Rightarrow$  (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes)  $\Rightarrow$  (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

CPU Burst I/O Burst CPU Burst I/O Burst CPU Burst I/O Burst

# Medium Term Scheduling

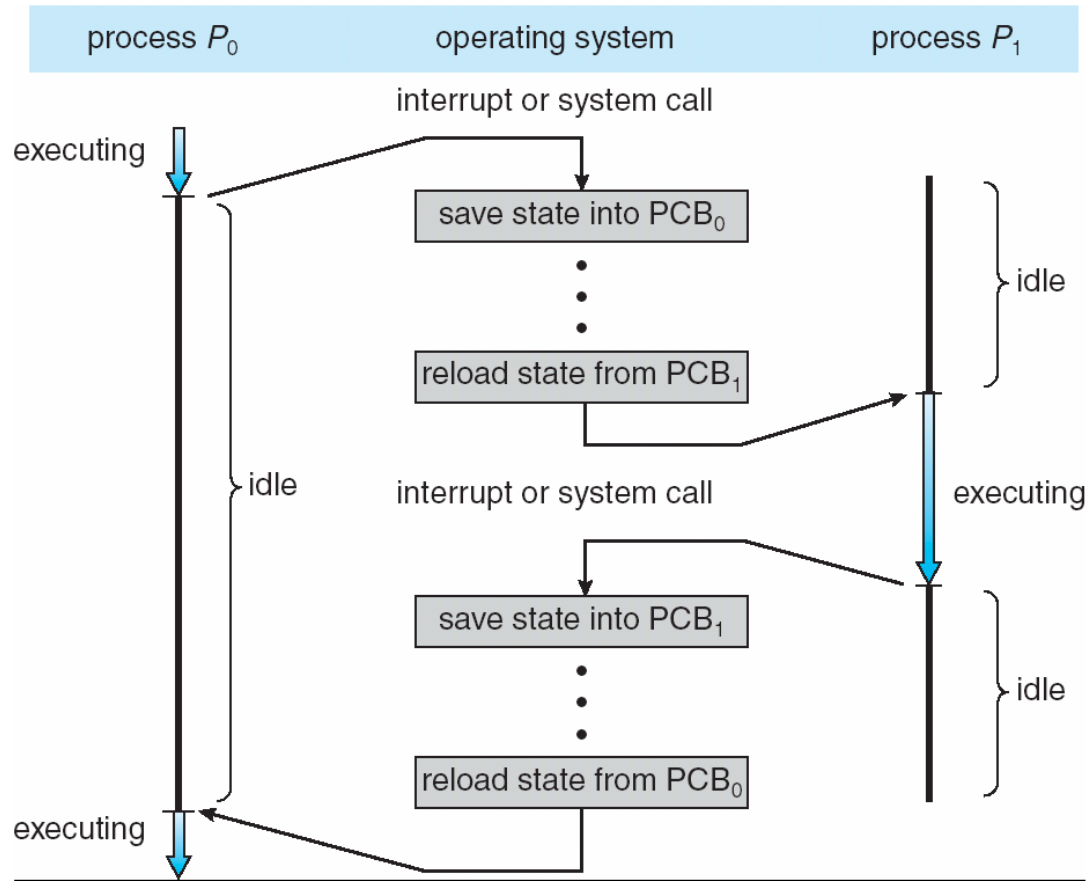
- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



# Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch

# Context Switch



# Scheduler Triggers

■ CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state
2. Switches from running to ready state
2. Switches from waiting to ready
3. Terminates

# Performance Metrics

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **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, not output (for time-sharing environment)
- The fairness of allocation (includes starvation).

# CPU Scheduling Algorithms



# First- Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:



- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

Suppose that the processes arrive in the order:

$$P_2, P_3, P_1$$

- The Gantt chart for the schedule is:



- Waiting time for  $P_1 = 6$ ;  $P_2 = 0$ ;  $P_3 = 3$
- Average waiting time:  $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

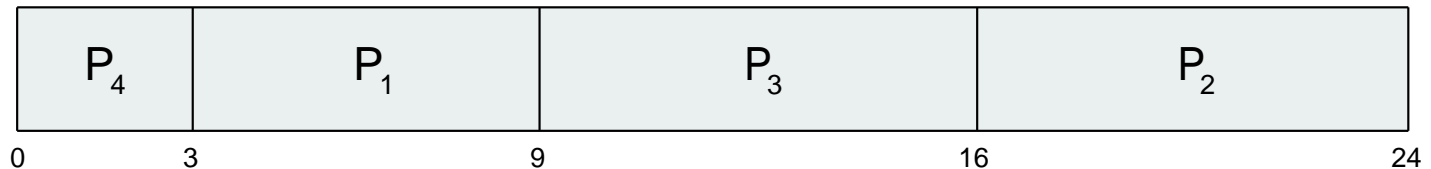
# Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request

# Example of SJF

<u>Process</u>	<u>Burst Time</u>
$P_1$	6
$P_2$	8
$P_3$	7
$P_4$	3

- SJF scheduling chart



- Average waiting time =  $(3 + 16 + 9 + 0) / 4 = 7$

# Determining Length of Next CPU Burst

- Can be done by using the length of previous CPU bursts, using exponential averaging

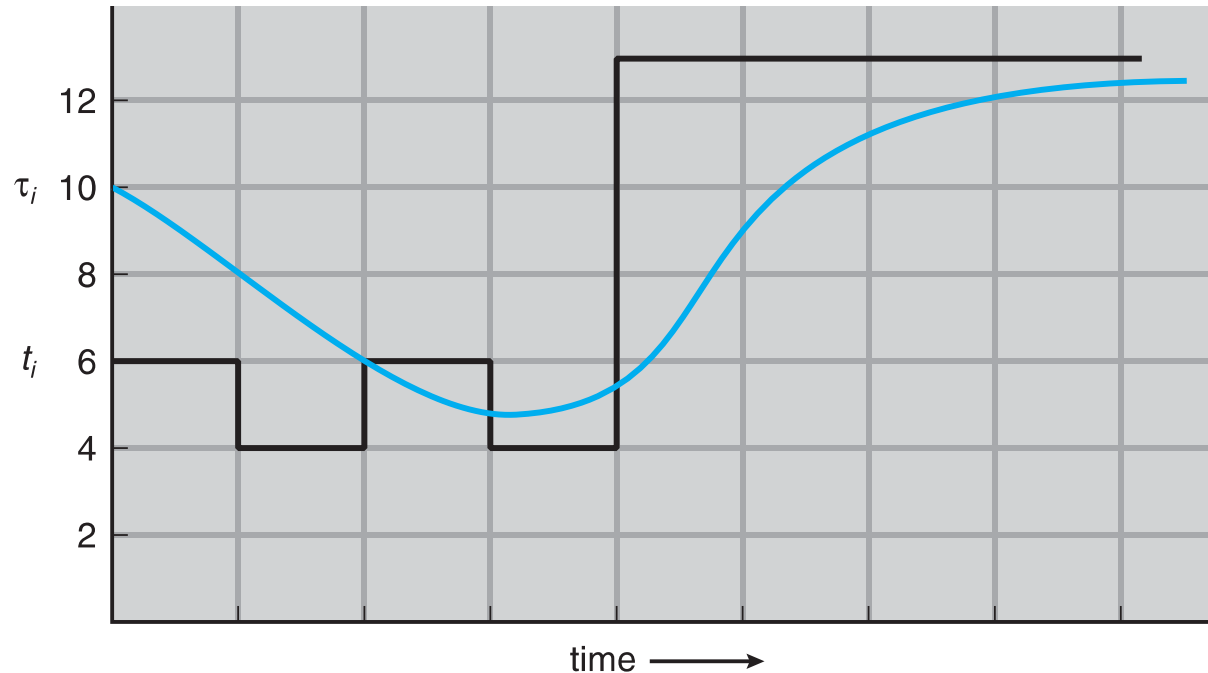
1.  $t_n$  = actual length of  $n^{th}$  CPU burst
2.  $\tau_{n+1}$  = predicted value for the next CPU burst
3.  $\alpha, 0 \leq \alpha \leq 1$
4. Define:  $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$ .

- If we expand the formula, we get:

$$\begin{aligned} \tau_{n+1} = & \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ & + (1 - \alpha)^j \alpha t_{n-j} + \dots \\ & + (1 - \alpha)^{n+1} \tau_0 \end{aligned}$$

- Commonly,  $\alpha$  set to  $\frac{1}{2}$

# Prediction of the Length of the Next CPU Burst



CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...
"guess" ( $\tau_i$ )	10	8	6	6	9	11	12	...

# Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer  $\equiv$  highest priority)
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem  $\equiv$  **Starvation** – low priority processes may never execute
- Solution  $\equiv$  **Aging** – as time progresses increase the priority of the process

# Priority

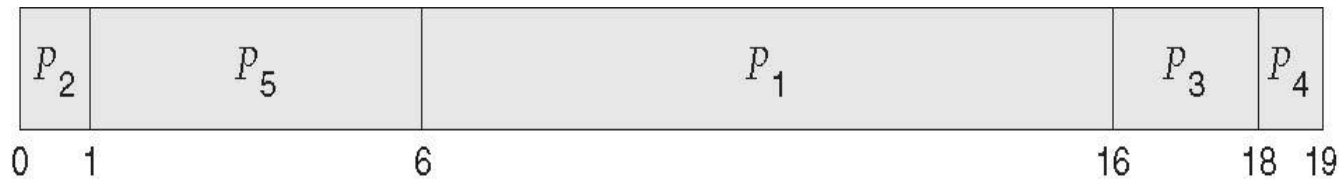
- Priorities are assigned by scheduling algorithms
- Most algorithms are classified as fixed priority, dynamic priority, or mixed priority.
- A fixed-priority algorithm assigns all priorities at design time, and those priorities remain constant for the lifetime of the task.
- A dynamic-priority algorithm assigns priorities at runtime, based on execution parameters of tasks, such as upcoming deadlines.
- A mixed-priority algorithm has both static and dynamic components.



# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	10	3
$P_2$	1	1
$P_3$	2	4
$P_4$	1	5
$P_5$	5	2

- Priority scheduling Gantt Chart

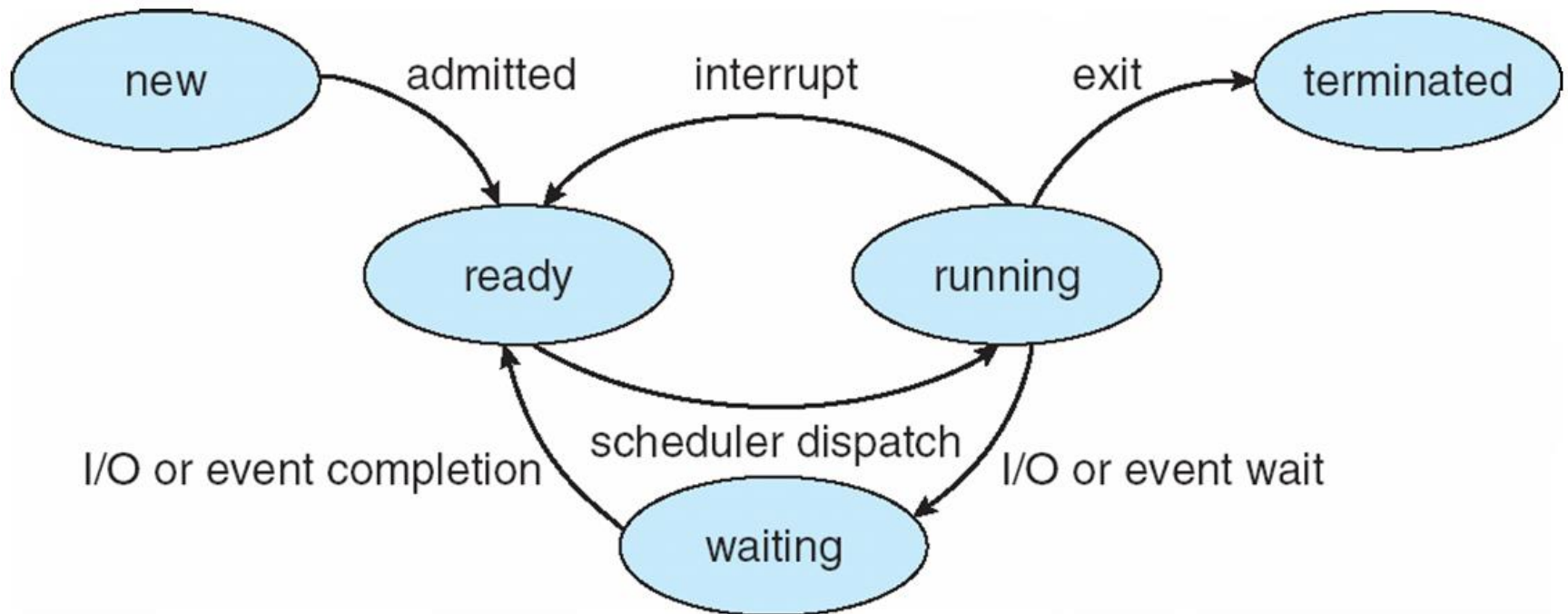


- Average waiting time = 8.2 msec

# Types of Scheduling Algorithm

- Non-preemptive – process release CPU only after its current CPU burst is over (all algorithms discussed till now)
- Preemptive – Process may have to release CPU even before finishing the current CPU burst

# Modified Diagram of Process State

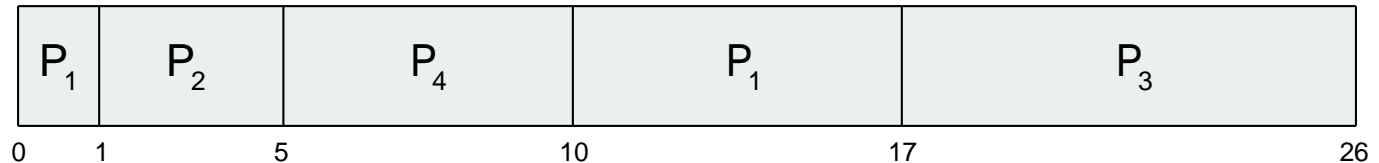


# Example of Shortest-remaining-time-first

- Now we add the concepts of varying arrival times and preemption to the analysis

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
$P_1$	0	8
$P_2$	1	4
$P_3$	2	9
$P_4$	3	5

- Preemptive SJF* Gantt Chart

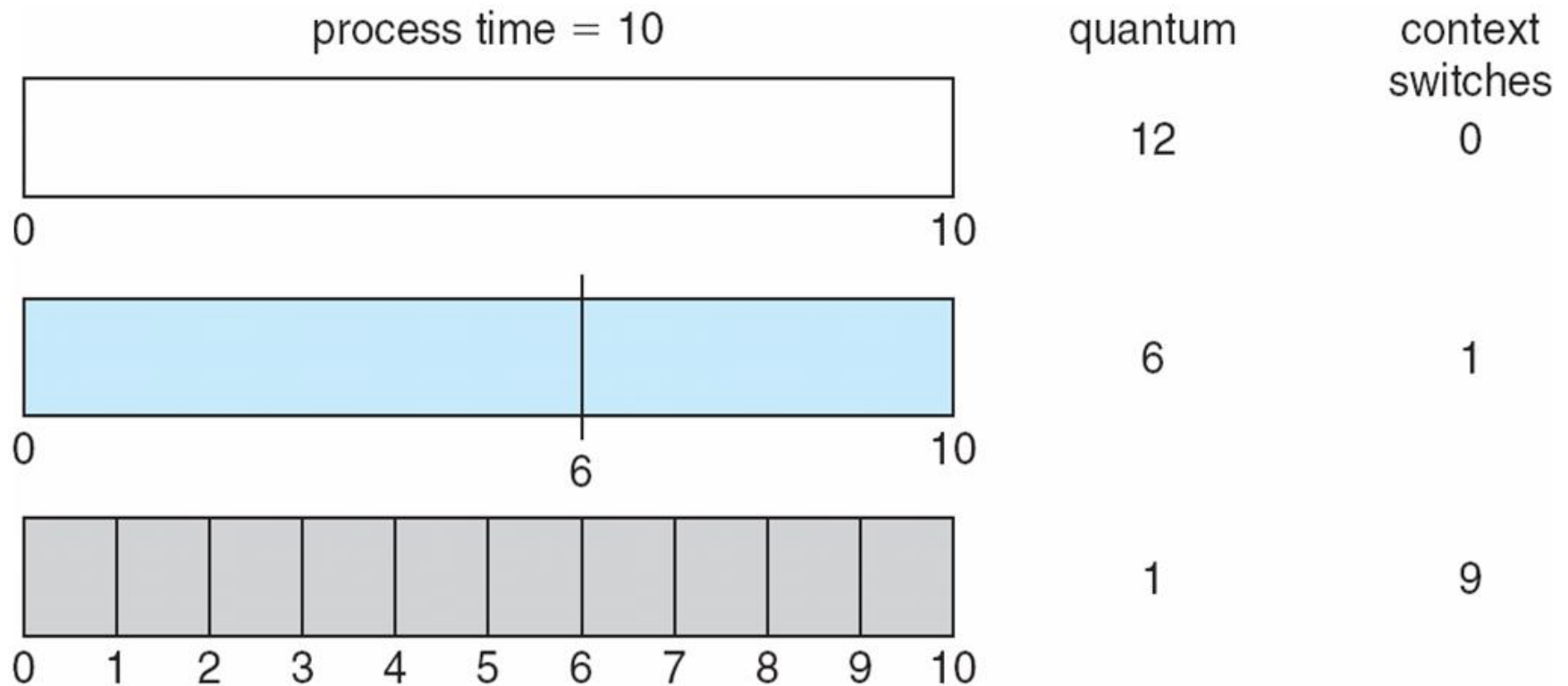


- Average waiting time =  $[(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5$  msec

# Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum  $q$** ), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are  $n$  processes in the ready queue and the time quantum is  $q$ , then each process gets  $1/n$  of the CPU time in chunks of at most  $q$  time units at once. No process waits more than  $(n-1)q$  time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - $q$  large  $\Rightarrow$  FCFS
  - $q$  small  $\Rightarrow q$  must be large with respect to context switch, otherwise overhead is too high

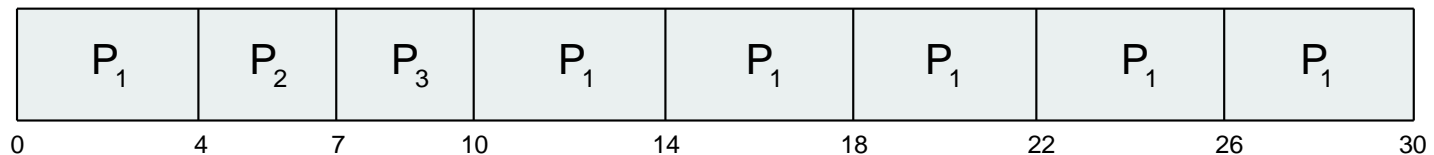
# Time Quantum and Context Switch Time



# Example of RR with Time Quantum = 4

<u>Process</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:



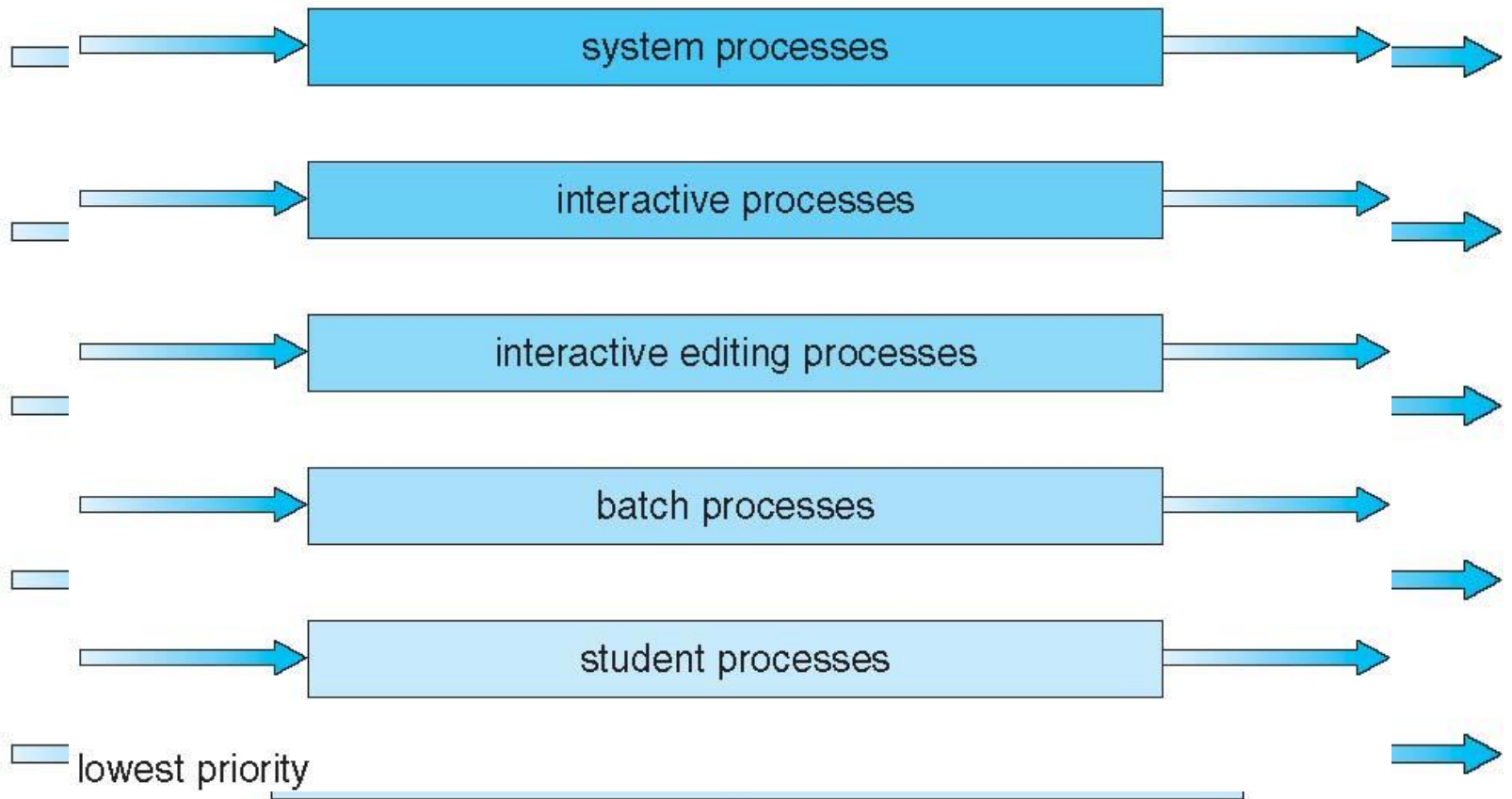
- Typically, higher average turnaround than SJF, but better **response**
- $q$  should be large compared to context switch time
- $q$  usually 10ms to 100ms, context switch < 10 usec

# Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS



highest priority



lowest priority

# Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to promote a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

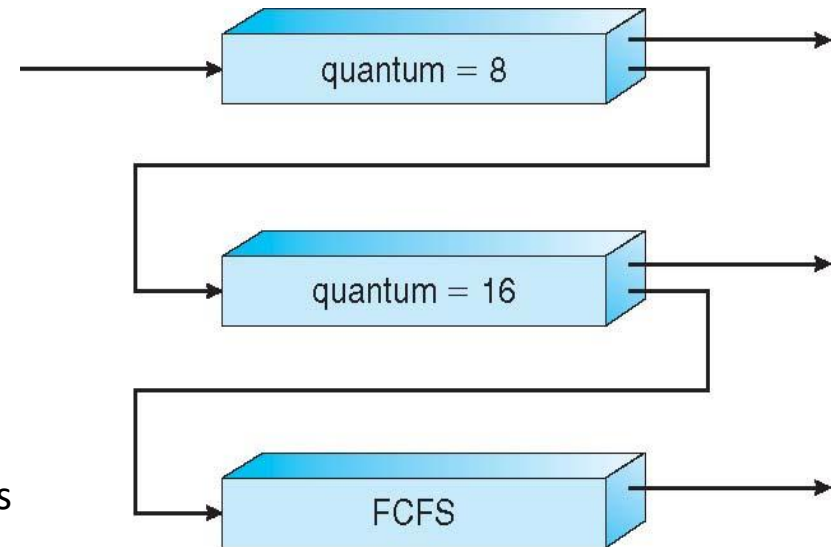
# Example of Multilevel Feedback Queue

- Three queues:

- $Q_0$  – RR with time quantum 8 milliseconds
- $Q_1$  – RR time quantum 16 milliseconds
- $Q_2$  – FCFS

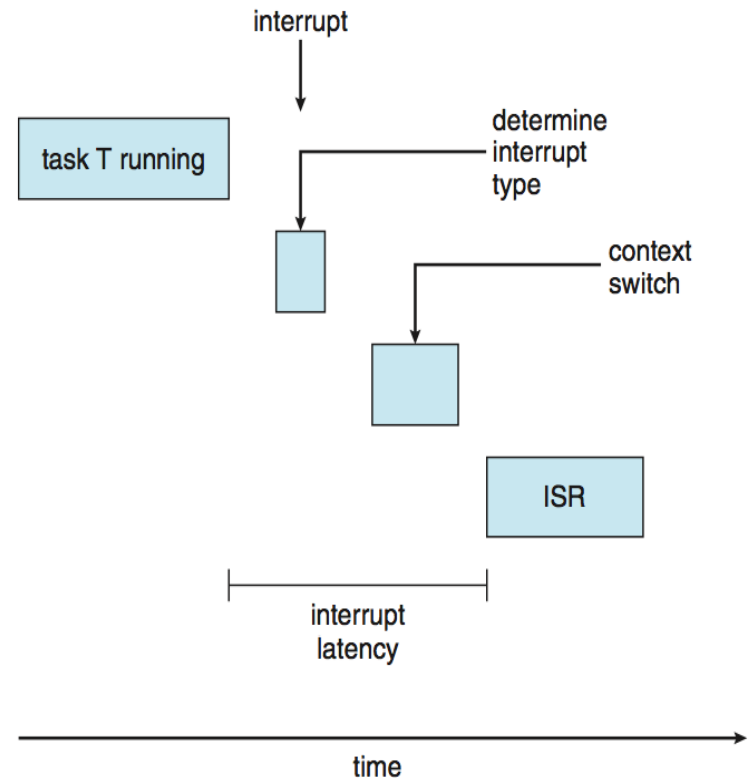
- Scheduling

- A new job enters queue  $Q_0$  which is served RR
  - When it gains CPU, job receives 8 milliseconds
  - If it does not finish in 8 milliseconds, job is moved to queue  $Q_1$
- At  $Q_1$  job is again served RR and receives 16 additional milliseconds
  - If it still does not complete, it is preempted and moved to queue  $Q_2$



# Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – tasks must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for scheduler to take current process off CPU and switch to another

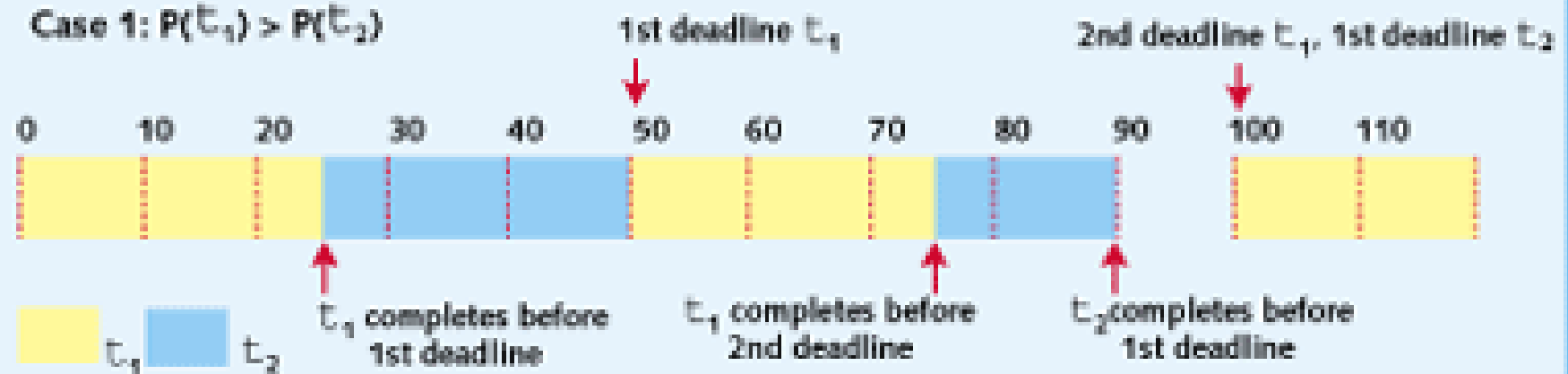


# Rate Monotonic Scheduling

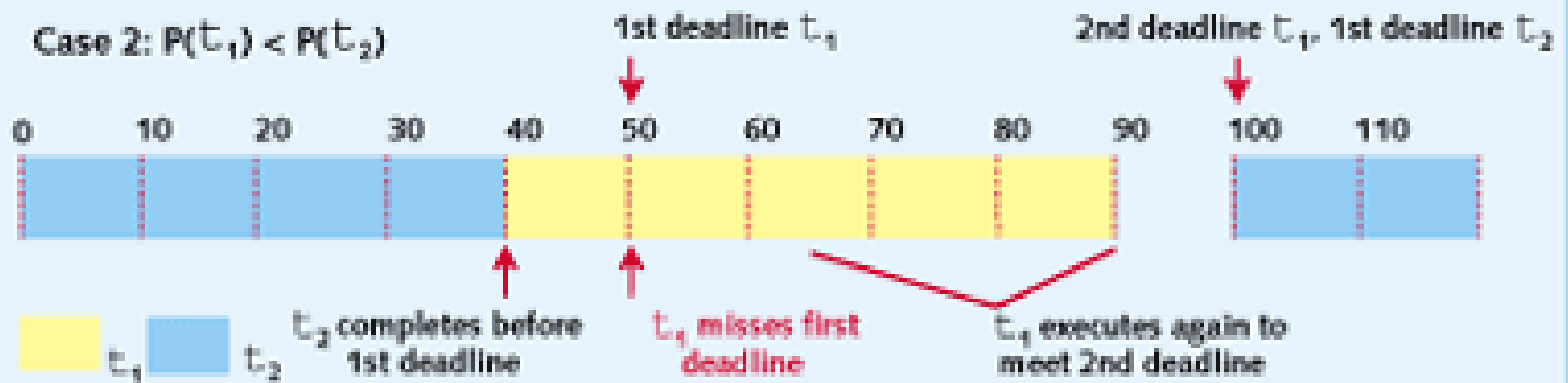
- The rate monotonic algorithm (RMA) is a procedure for assigning fixed priorities to tasks to maximize their schedulability.
- A task set is considered schedulable if all tasks meet all deadlines all the time.
- Processes have new characteristics: **periodic** ones require CPU at constant intervals

- **Assign the priority of each task according to its period, so that the shorter the period the higher the priority.**
- **Example:**
  - **Task1(t1): execution time = 25, Period = 50**
  - **Task2(t2): execution time= 40, period = 100**

Case 1:  $P(\tau_1) > P(\tau_2)$

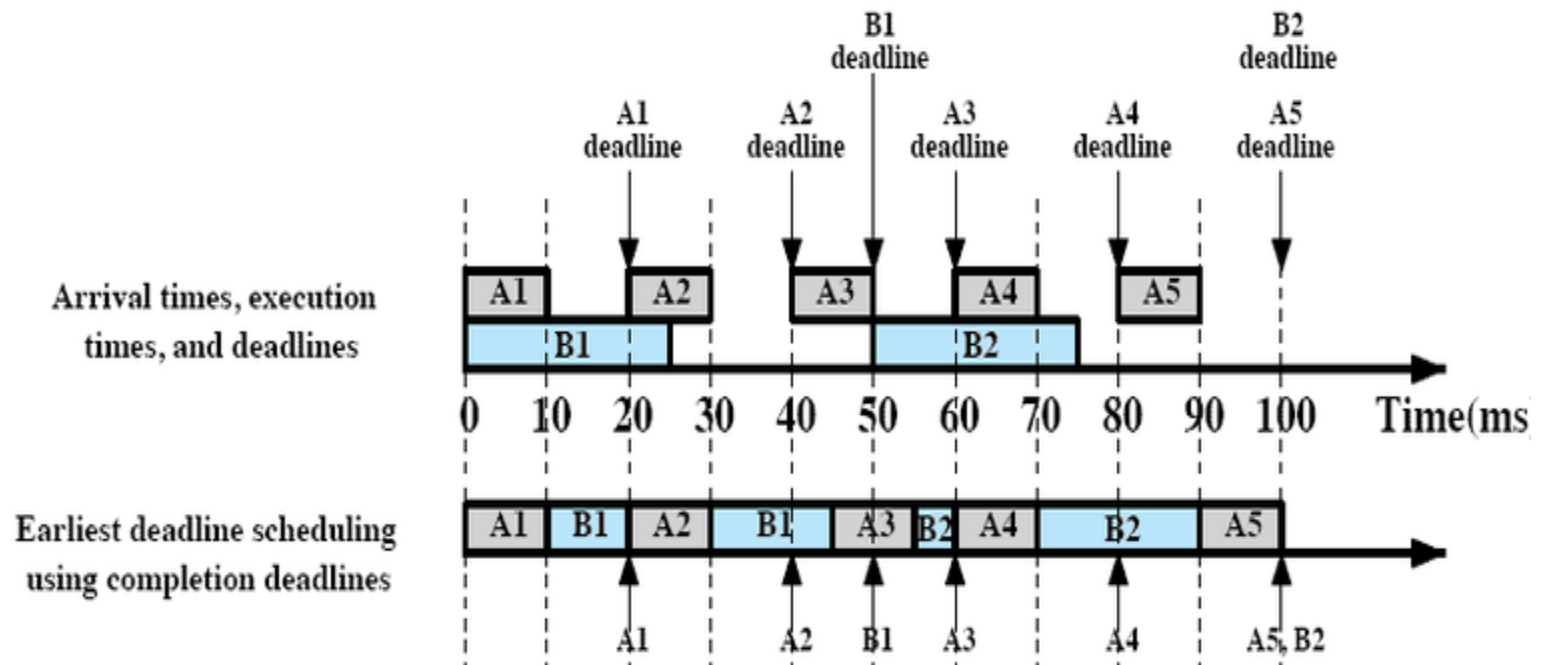


Case 2:  $P(\tau_1) < P(\tau_2)$



# Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines: the earlier the deadline, the higher the priority;

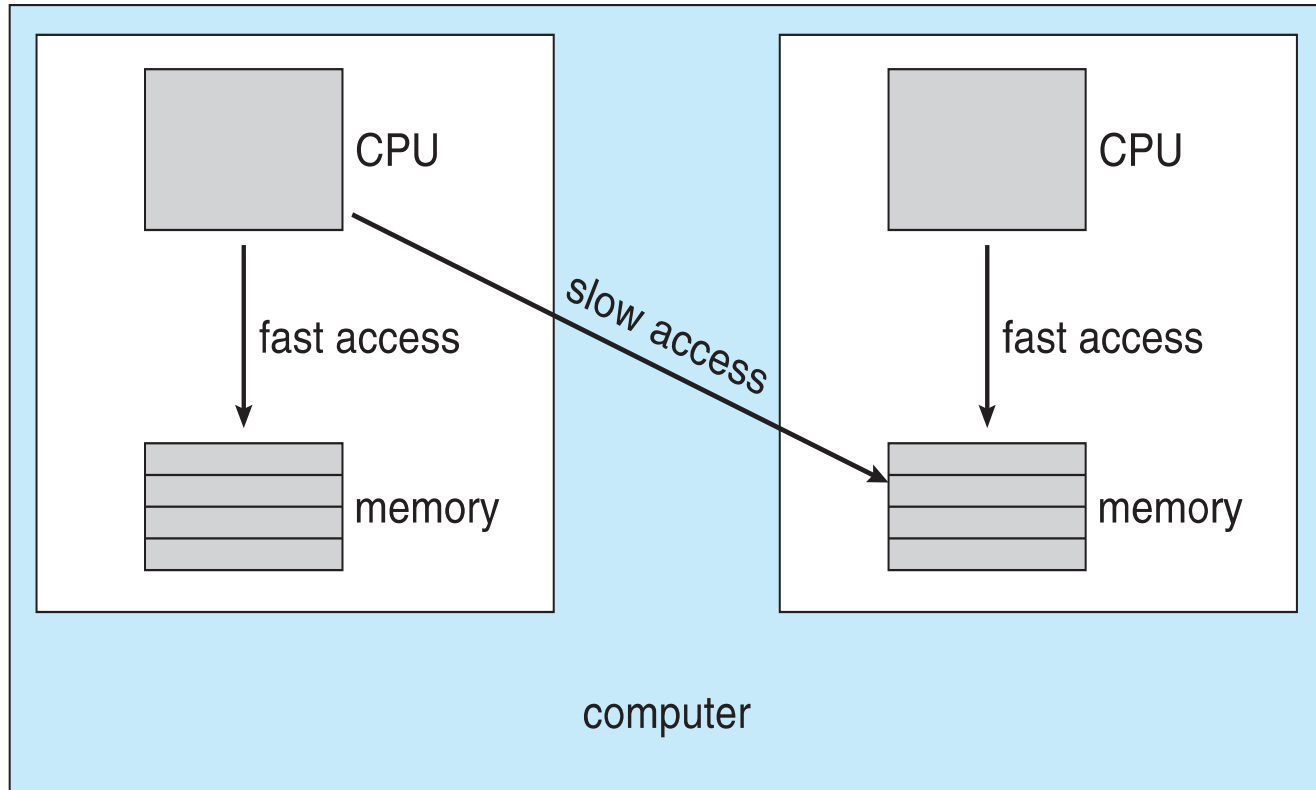




# Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity**
  - **hard affinity**
  - Variations including **processor sets**

# NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

## Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

# Linux Scheduling Through Version 2.5

- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order  $O(1)$  scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger  $q$
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
    - Two priority arrays (active, expired)
    - Tasks indexed by priority
    - When no more active, arrays are exchanged
  - Worked well, but poor response times for interactive processes