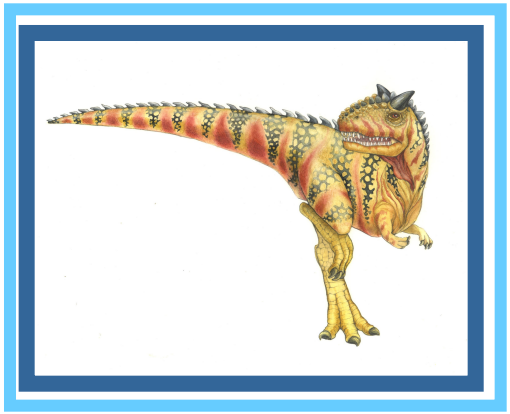
# Chapter 5: CPU Scheduling



# Chapter 5: CPU Scheduling

- Basic Concepts/introduction of CPU scheduling.
- Scheduling Criteria – metrics to optimize.
- Uniprocessor Scheduling Algorithms.
- Real-Time Scheduling.
- Multiple-Processor Scheduling.
- Operating Systems Examples
- Algorithm Evaluation – how to pick one?

# Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems.
- To describe various CPU-scheduling algorithms.
- Explain issues related to multiprocessor scheduling.
- Discuss scheduling in Windows, Solaris and Linux.
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system.
- Program(s) that implement CPU scheduling algorithms.

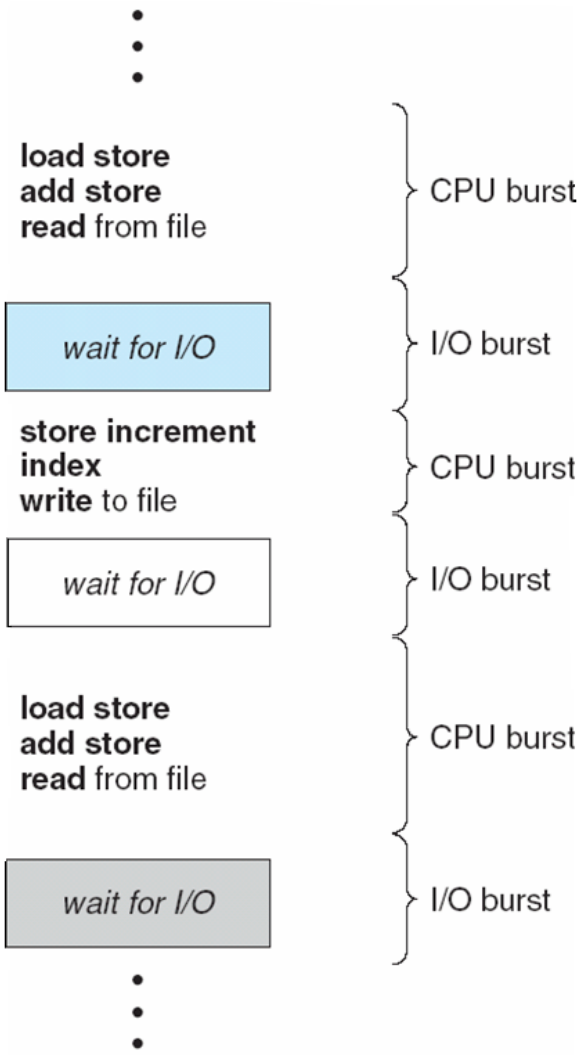
# Basic Concepts

- Scheduling is a very basic problem, not just in operating systems.
- We deal with scheduling decisions every day:
  - Do home work.
  - Eat meals.
  - Go for a run.
  - Chat online with parents.
- Which sequence of events to follow?
- **CPU scheduling is the basis of multiprogrammed operating system.**
  **How?**
- Maximum CPU utilization obtained with multiprogramming.
  - Try to ensure that as long as there is even one process to run, the CPU is never idle.
- Idea is to keep several ready processes in memory (ready queue).
  - When a process waits, CPU is given to a process from that queue.

# CPU and I/O Bursts

- Success of CPU scheduling is based on an important property of processes:
  - **Typically, process execution consists of periods of CPU and I/O bursts.**
- CPU– I/O Burst Cycle: Process execution consists of *cycles* of CPU execution and I/O wait, and so on.
  - To “burst” means to appear or to become visible.
  - Any example of such a process?
- Duration of CPU and I/O bursts vary from one set of processes to the other.
- Have been measured extensively by researchers.
- **A CPU burst** distribution in next slide (frequency of bursts vs. burst duration).
  - Shows a large number of short CPU bursts followed by a small number of large CPU bursts: **how can this information be useful?**
- **An I/O bound process will typically have short CPU bursts.**
- **A CPU bound process will typically have long CPU bursts.**

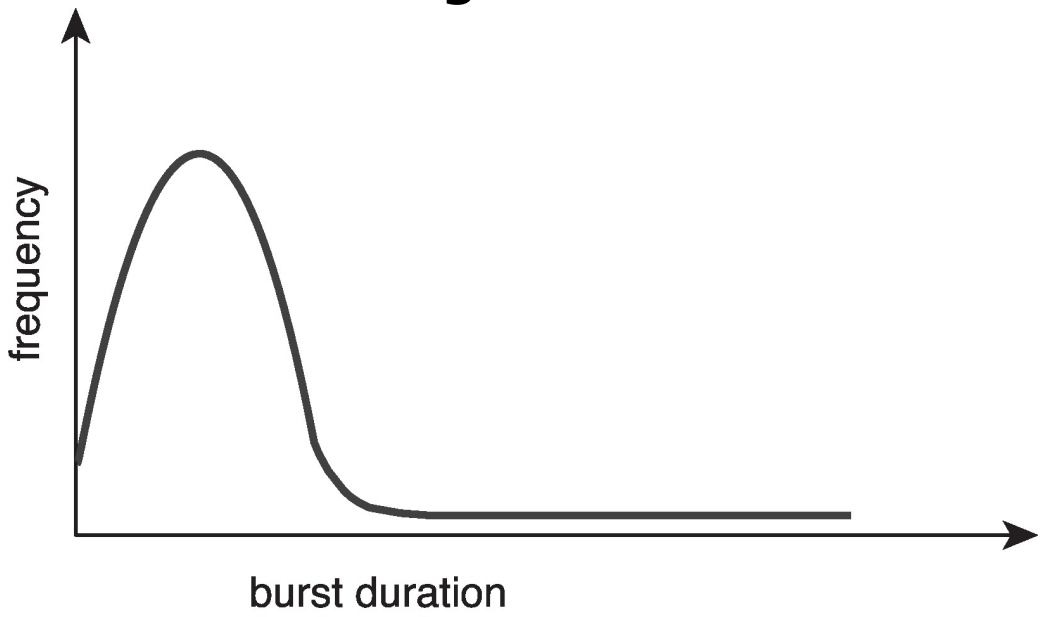
# Alternating Sequence of CPU And I/O Bursts




# Histogram of CPU-burst Times

Large number of short bursts

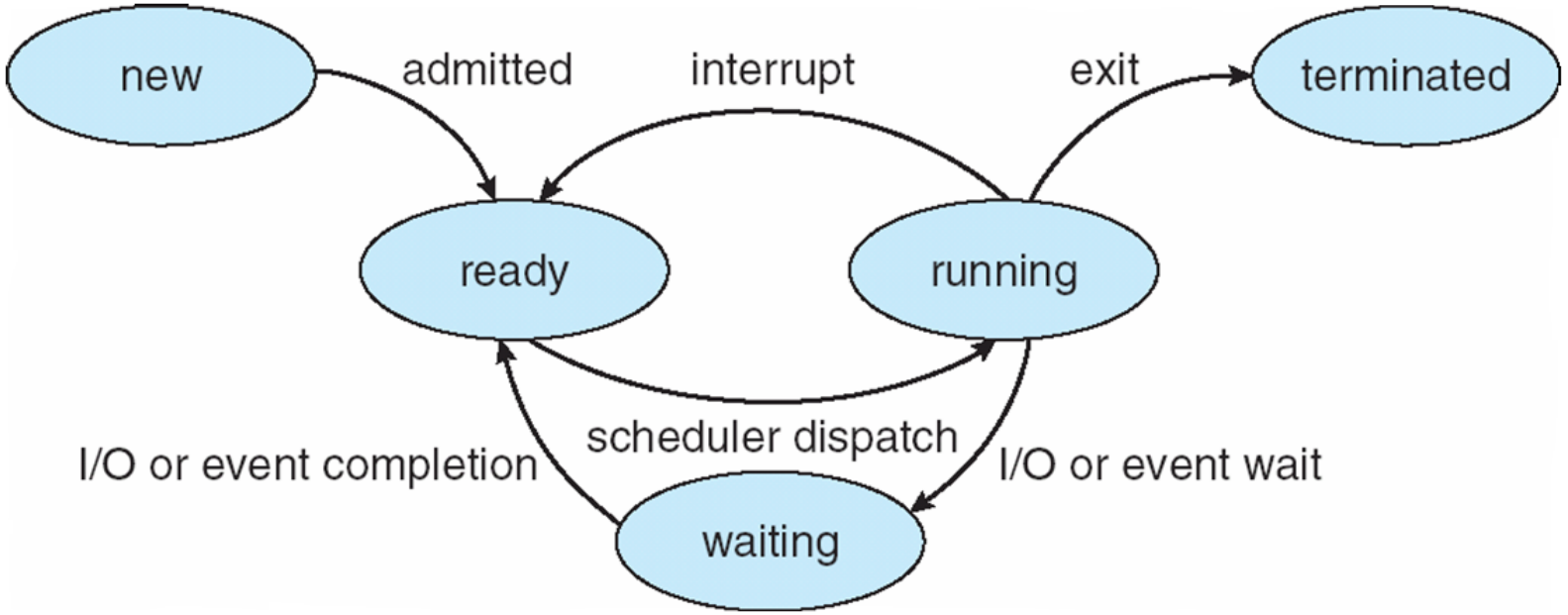
Small number of longer bursts




# CPU Scheduler

- A very important module of any OS.
- Main task of scheduler: **selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.**
- Ready queue: list of processes that are ready to execute on the CPU.
  - PCBs are linked together in the queues.
- Recall the different states a process can be in, from chapter 3.
- Let's look at the picture (next slide).
- CPU scheduling decisions take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.

# Diagram of Process State




# Dispatcher

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

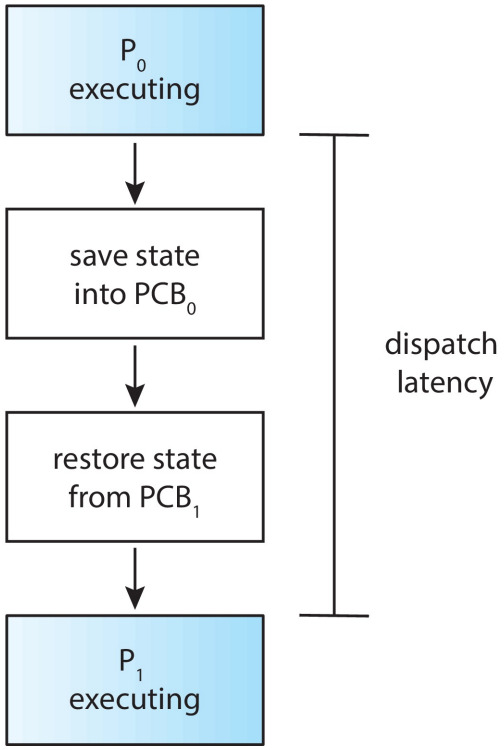
- switching context (one proc. to another).
- switching to user mode.
- jumping to the proper location in the user program to restart that program.

**Dispatch latency** – time it takes for the dispatcher to stop one process and start another running.

Dispatcher should be fast, why?

Peek to see context switches.

- cd /proc/2166
- vi status
- Voluntary and involuntary context switches: difference?




# Preemptive vs. Non Preemptive

- **Scheduling algorithms can broadly be classified into:**
- **Non preemptive scheduling**:
  - Once CPU allocated to process, process keeps CPU till releases it, either by terminating or by switching to waiting state (to do I/O).
- **Preemptive scheduling**:
  - Once CPU is allocated to process and it is running, it may be interrupted (its execution is stopped) to run another process.
  - Another process usually has higher priority.
- **Which is better?**
- Not an easy question to answer.
- We will discuss preemption & non-preemption throughout this chapter.

# Scheduling Criteria

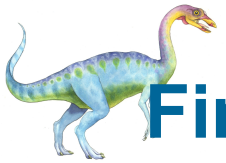
- **CPU utilization** – **% time spent in executing instructions**. Conceptually, CPU utilization varies from 0 to 100. Realistically, a value of 100 may not be achievable. Why?
- **Throughput** – **number of processes that complete their execution per time unit**. For long processes, it could be: one process/sec; for short processes, it could be: 10 processes/sec.
- **Turnaround time** – **amount of time to execute a particular process.** Is the interval from the time of submission of a process to the time of it's completion.
  - Includes time spent:
    - executing on CPU,
    - doing I/O,
    - waiting in ready queue,
    - waiting on disk.

# Scheduling Criteria

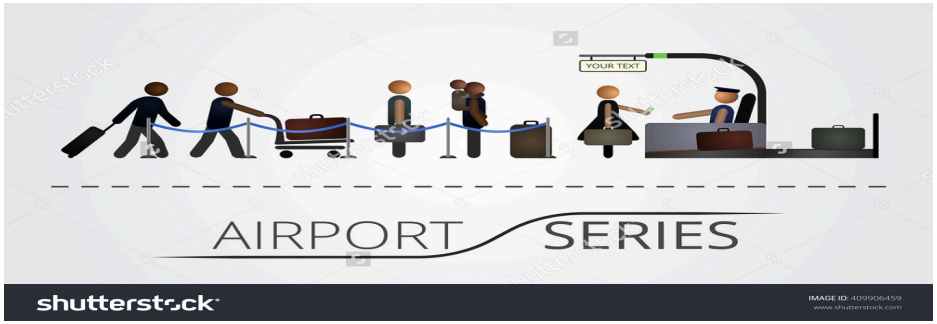
- **Waiting time** – **amount of time a process has been waiting in the ready queue**.
- **Response time** – **amount of time it takes from when a request was submitted until the first response is produced, not the final output**.
  - Used for interactive processes, for example, a video game.
  - Often, a process can continue computing new results while previous results are being output to the user.

# Scheduling Algorithm Optimization Criteria

- **Maximize or minimize these?**
- CPU utilization?
- Throughput?
- Turnaround time?
- Waiting time?
- Response time?
- Can also study variation in response times (for predictability).
- Some criteria are conflicting: **CPU utilization vs. response time, average turnaround time vs. maximum waiting time. How?**
- Scheduling is an NP – hard problem for all but the very simplest of cases (**easy to phrase, challenging to solve**).
- Relaxing some assumptions can make the scheduling problem easier to solve. For example:
  - Independent processes easier than dependent processes.
  - Single CPU easier than multiple CPUs.
  - Homogenous CPUs easier than heterogeneous CPUs.



# First Come First Serve Scheduling (FCFS)

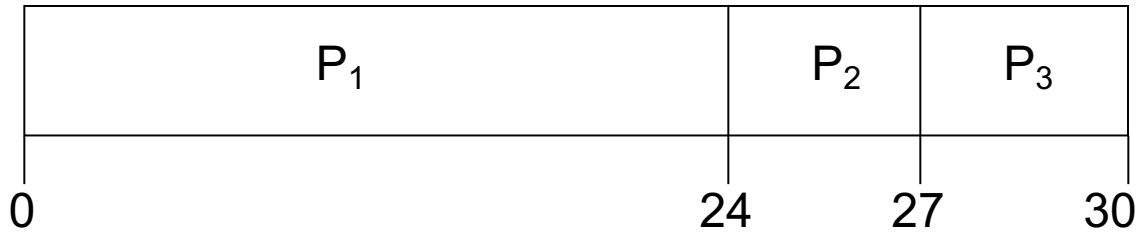



- Image courtesy: www.shutterstock.com.
- Very basic/simple scheduling algorithm.
- **Idea: process that requests CPU first is allocated the CPU first: First in First Out (FIFO).**
- Which process to pick from ready queue? At any given point, always select the process at the head of the ready queue.
- New processes added at the tail of the ready queue.
- Advantage?
  - Easy to understand and implement.
- Disadvantage?
  - Can result in very high wait times for some processes. How?

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

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- Suppose that the processes arrive in the order: $P_1$ , $P_2$ , $P_3$ . The Gantt Chart (*named after Mech. Engg. from 1800s*) for the schedule is:



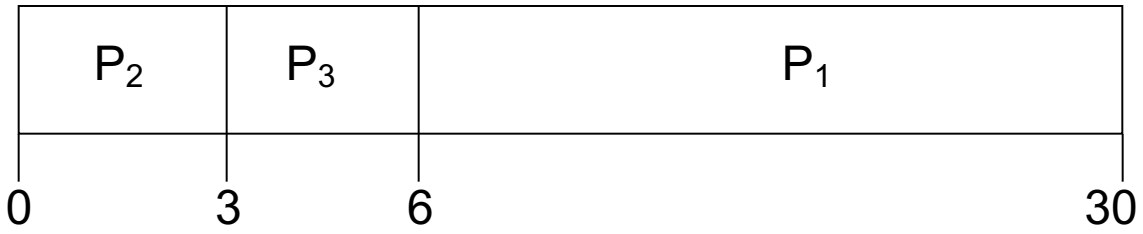

- Waiting time for $P_1$ ? , for $P_2$ ? for $P_3$ ?
- Average waiting time?
- = (0 + 24 + 27)/3 = 17.
- Average completion time?
- = (24 + 27 + 30)/3 = 27.
- **Can we do better in terms of the average waiting time?**

# FCFS Scheduling (Cont)

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.
- Average completion time: (3 + 6 + 30)/3 = 13.
- **Note that the total execution time is the same.**

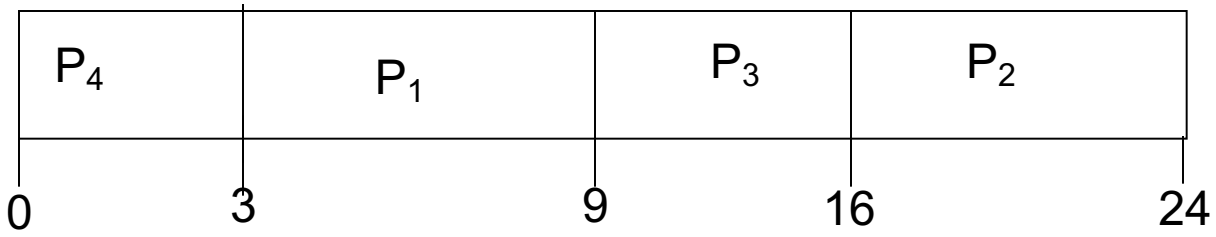
# Shortest Job First (SJF) Scheduling

- **Idea: execute the process with the shortest CPU burst first.**
  - If two processes have same burst, select randomly or use FCFS to break the tie.
- SJF is optimal – gives minimum average waiting time for a given set of processes.
- Why does it work well?
- **Moving a short process before a long one decreases the waiting time for the short process more than it increases the waiting time for the long process.**
- Hence, average waiting time decreases.
- Run shorter processes first. After a specified time, larger number of processes would have finished execution, versus the case when longer processes execute before the shorter ones.

# Example of SJF

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |

- SJF scheduling chart




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

# Make SJF Preemptive

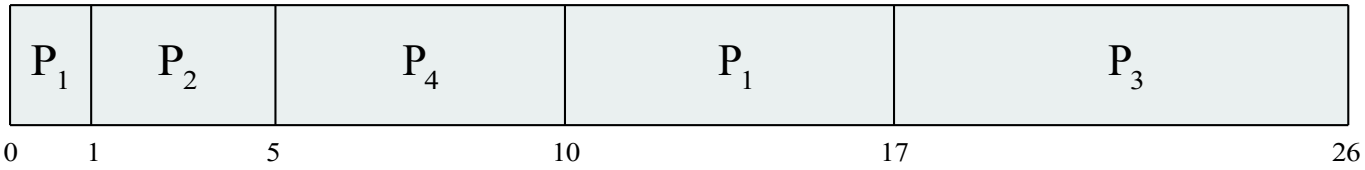
- Two flavours for SJF: non preemptive or preemptive.
- Consider a new process that arrives in the system while a previous process is still executing. What do we do?
- Two options:
  - Let previous running process execute.
  - Take the CPU away from the previous process and give it to the new process, provided its CPU burst time is the lowest.
- Concept leads to the "**Shortest Remaining Time First Algorithm (SRTF)**".
- **Idea: Given a choice, the process with the minimum value of CPU burst time remaining is given the CPU.**
- This is a preemptive algorithm.
- Lets run this on a set of processes.

# Shortest-remaining-time-first (SRTF)

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

| Process | *Arrival* Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |

- *Preemptive* SJF Gantt Chart



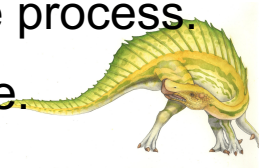

- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3))]/4 = 26/4 = 6.5 msec.

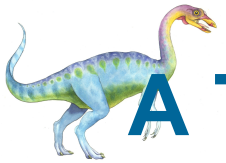
# SJF vs. SRTF

- Last lecture: discussed algorithms for scheduling processes on a CPU – **FCFS**, **SJF & SRTF**.
- Shortest Job First (SJF) – Non preemptive: *process upon getting the CPU, is executed till completion, uninterrupted*.
- Average wait time for same process set?
- Shortest Remaining Time First (SRTF) – Preemptive: *process may be interrupted/preempted*.

# Priority Scheduling

- The Shortest Job First (SJF) and Shortest Remaining Time First (SRTF) are examples of **priority based** scheduling algorithms.
- **Question - why do we need priority based scheduling?**
- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (for example, smallest integer ≡ highest priority).
- SJF is a priority scheduling where priority is?
- Earliest Deadline First (EDF) scheduling assigns priority based on the deadline.
  - Earlier deadline implies higher priority. More on this later.
- A problem with priority scheduling ≡ **Starvation** – low priority processes may never execute.
  - Steady stream of high priority processes arrive in the system.
- Solution ≡ **Aging** – as time progresses increase the priority of the process.
  - Eventually, it will get a chance to run. Let's look at an example.





# A Tutorial Exercise on Process Aging

- A preemptive priority scheduling algorithm – larger number => higher priority.
- Priorities change dynamically over time, say rate of *x/second*.
- Process waiting in ready queue → priority changes at rate α.
- Process running → priority changes at rate β.
- All processes given priority 0 when they enter ready queue.
  - **What happens when: β > α > 0?**
  - **What happens when: α < β < 0?**

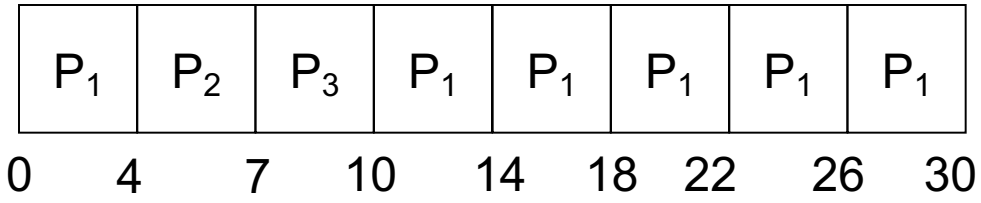
# Round Robin (RR)

- **Observation: priority algorithms tend to be unfair to lower priority processes.**
- A fairer algorithm: – round robin.
- **Similar to FCFS, but with preemption added.**
- Each process gets a unit of CPU time (*time quantum*). After this time has elapsed, the process is preempted and added to the tail of the ready queue.
- What if process finishes before expiry of its time quantum?
- The CPU is given to the next process in the ready queue.
- Theoretically, If there are ***n*** processes in the ready queue and the time quantum is ***q***, then each process can get **1/*n*** of the CPU time in slots of at most ***q*** time units at once.
- Let's look at an example.

# Example of RR with Time Quantum = 4

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |

- The scheduling chart is:



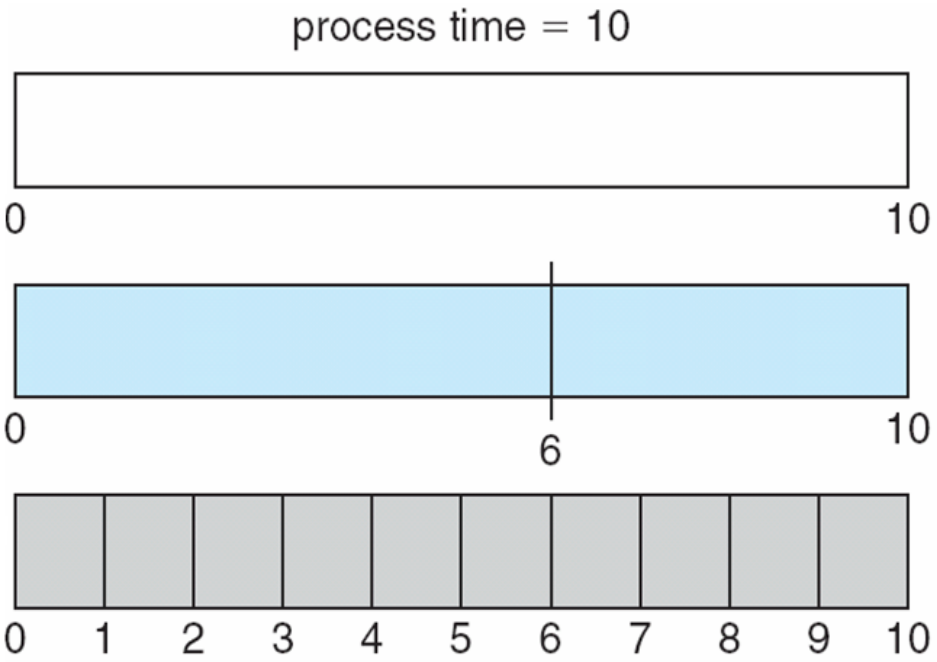

# Round Robin Wait Time Analysis

- **With this technique, what is the maximum wait time for a process?**
- If there are 5 processes and q = 5. In the worst case, how much time does a process have to wait?
- Process 1 → 0 – 5, process 2 → 5 – 10, process 3 → 10 – 15, process 4 → 15 – 20 and process 5 → 20 – 25.
- Hence, the maximum wait time is 20.
- For the general case of ***n*** processes and ***q*** time quantum, maximum wait time is:
- ***(n – 1) * q***.
- In contrast, the waiting time could be high in FCFS or priority based scheduling.
- Goal of round robin: *share the CPU equally among all processes and to reduce the overall waiting time*.

# RR Scheduling

- No process is allocated the CPU for more than 1 time quantum at a stretch.
  - **One exception to this rule?**
- Performance depends on the size of the time quantum.
  - *q* is very large ⇒ **what would happen?**
  - *q is* small ⇒ **what would happen?**
- Ideally, we want *q* to be large with respect to the context switch time.
  - Typical context switch times are in the microseconds range.
  - Time quantum is generally in the milliseconds range.

# Time Quantum and Context Switch Time




| quantum | context switches |
|---|---|
| 12 | 0 |
| 6 | 1 |
| 1 | 9 |

# A Tutorial Exercise – RR Scheduling

- A system runs 10 I/O bound tasks and 1 CPU bound task.
- I/O bound tasks issue an I/O operation once after every millisecond of CPU computing. Each I/O operation takes 10 milliseconds to complete.
- Context switching overhead is 0.1 millisecond.
- All processes are long running.
- What is the CPU utilization for a round-robin scheduler when:
  - The time quantum is 1 millisecond.
  - The time quantum is 10 milliseconds.

# Real-Time Systems Scheduling

- Recall from chapter 1 that real-time systems have strict timing constraints.
- Real-Time does not necessarily mean fast (common misconception)!
- Process needs to finish execution before its deadline, else there can be negative consequences.
  - Loss of human life.
  - Loss of property.
- For example, braking system of an airplane, sensor system on the border that keeps track of enemy movements.
- Scheduling algorithms for real-time systems have one broad goal in mind:
  - **Meet all process deadlines.**
  - **Give guarantee for meeting deadlines: if so and so condition is satisfied, it is guaranteed that all deadlines will be met.**

# Real-Time CPU Scheduling

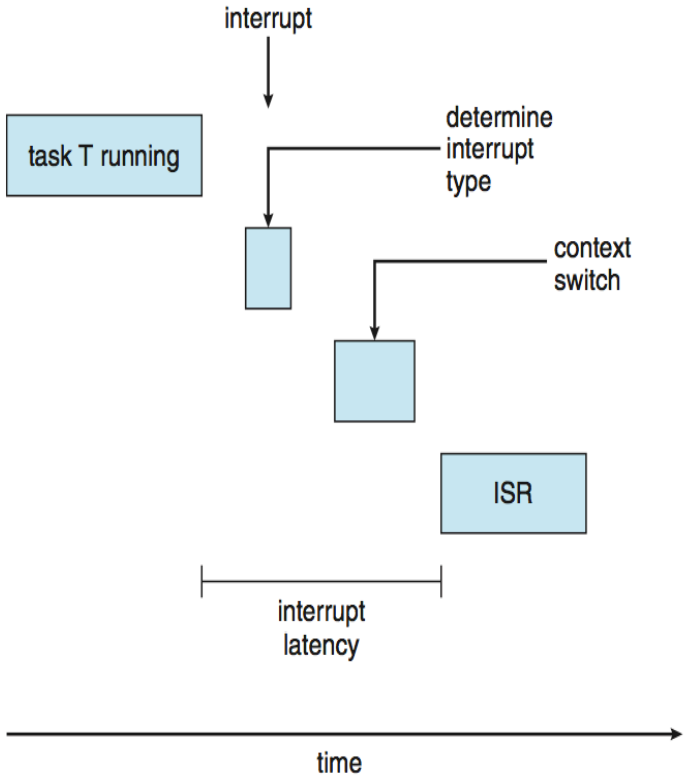
Can present obvious challenges

**Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled, some lateness can be tolerated (e.g. transmission of audio, video frames over the internet).

**Hard real-time systems** – task must be serviced by its deadline (e.g. braking system of an airplane).

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 schedule to take current process off CPU and switch to another.



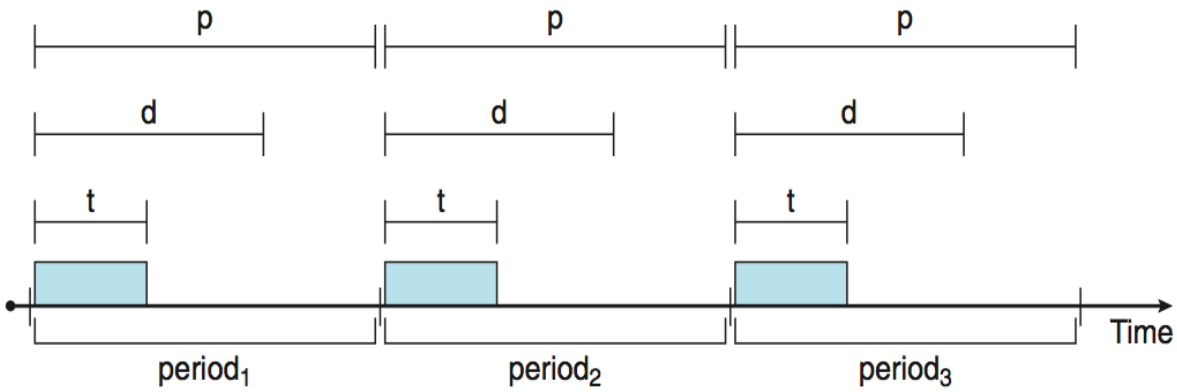

# Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling.
- For hard real-time, must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals

  Has processing time *t*, deadline *d*, period *p*

  $0 \le t \le d \le p$

  **Rate** of periodic task is $1/p$

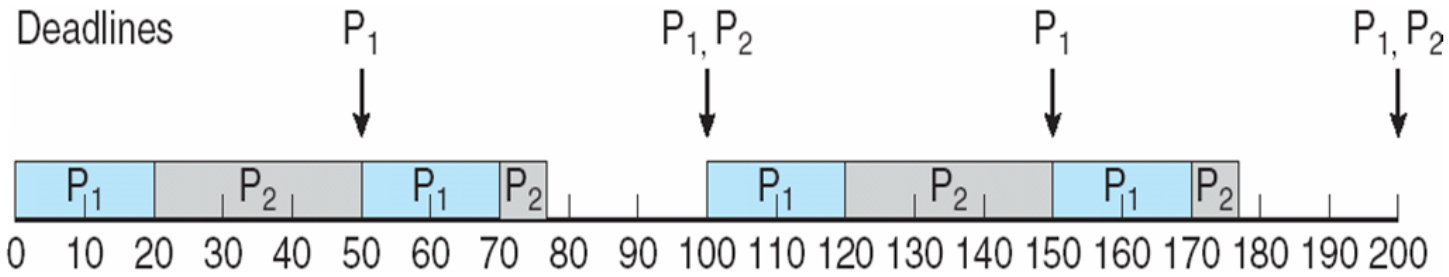



# Rate Monotonic (RM) Scheduling

- Static priority scheduling algorithm for real-time systems.
- Uses preemption.
- **Key concept: priority of a process is inversely proportional to its period.**
- Lower the period, higher will be the priority.
- Explanation:
  - Lower period tasks will request the CPU more often.
  - Give them higher priority.
- Before the schedule is generated, the priorities are evaluated using the period values.
- **Key feature here is that priorities are static: they do not change during the course of execution of the algorithm.**
- Let's look at an example from the book: given on pages 227 – 228 of the book.

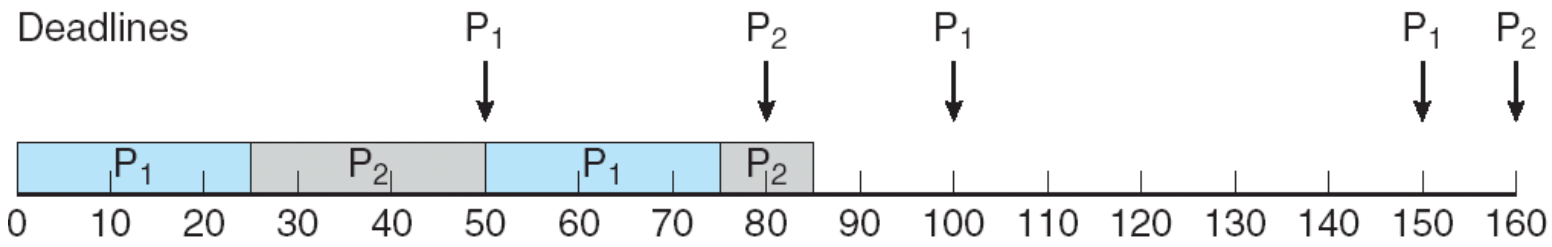
# Rate Monotonic Scheduling: An Example

- A (static) priority is assigned based on the inverse of its period
- Period of P1 = 50, of P2 = 100.
- Execution time for P1 = 20, of P2 = 35.
- Hence, deadline of P1 = 50, of P2 = 100.
- Which task gets a higher priority?
- $P_1$ is assigned a higher priority than $P_2$.




# Missed Deadlines with Rate Monotonic Scheduling

- P1: period = 50, execution time = 25.
- P2: period = 80, execution time = 35.
- Which process has higher priority?
- P1.



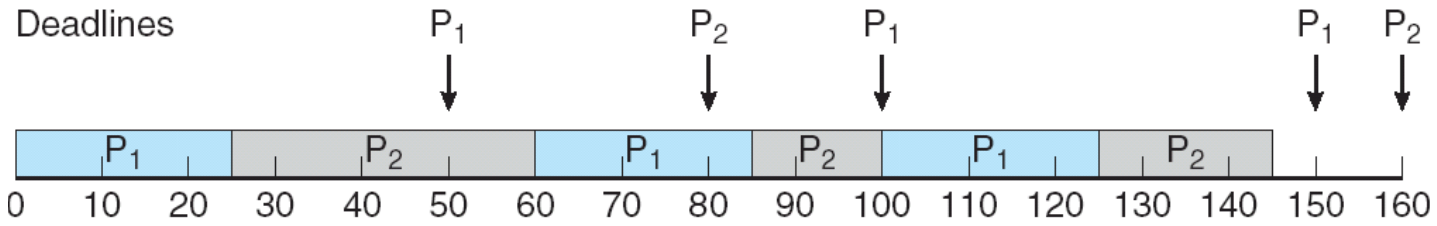

# Earliest Deadline First (EDF) Scheduling

- Another scheduling algorithm for real time systems.
- Processes are periodic.
- Deadline is at the end of the period.
- **Key concept: priorities are assigned based on deadline values.**
  - **Smaller deadline means higher priority.**
- **Dynamic priorities – priority of a process might change during the course of execution.**
- Process preemption is employed.
- Let's go though an example – given on pages 228-229 of the book.

# Earliest Deadline First Scheduling (EDF)

Priorities are assigned according to deadlines:

- the earlier the deadline, the higher the priority;
- the later the deadline, the lower the priority
- P1: period = 50, execution time = 25, deadline = 50.
- P2: period = 80, execution time = 35, deadline = 80.
- Which process has higher priority?
- P1, initially.




# EDF Schedulability Condition

- EDF is an optimal algorithm.
- Guarantees to meet deadlines, even if CPU utilization = 1.
- Utlization (task) = execution cost (task) / period (task).
- $U_{total} <= 1$.
- Where $U_i = e_i / p_i$.
- Note that this value is theoretical, as it ignore context switching costs.

# Tutorial Problem on Real-Time Scheduling.

- A real-time system needs to handle two voice calls that each run every 5 milliseconds and consume 1 millisecond of CPU time per burst, plus one video at 25 frames/sec, with reach frame requiring 20 milliseconds of CPU time. Is the system schedulable, such that all deadlines are met?

# Multiprocessor Scheduling

- Processors are becoming cheaper and faster.
- Makes sense to use their “combined” power to make scheduling more efficient.
- **Idea**: execute processes/sub processes in parallel on multiple processors so that processes can “finish faster”.
- Two processes with execution costs 5 each will finish at time = 10.
- Executing them in parallel on two processors could result in them finishing at time t = 5.
- Turns out that this is not that simple.
- Challenges:
  - Scheduling algorithms become more complex.
  - Communication between processes adds delay.
  - Maintaining balanced load on all processors.
    - Ensure that a popular processor is overloaded, while other processors are idle.

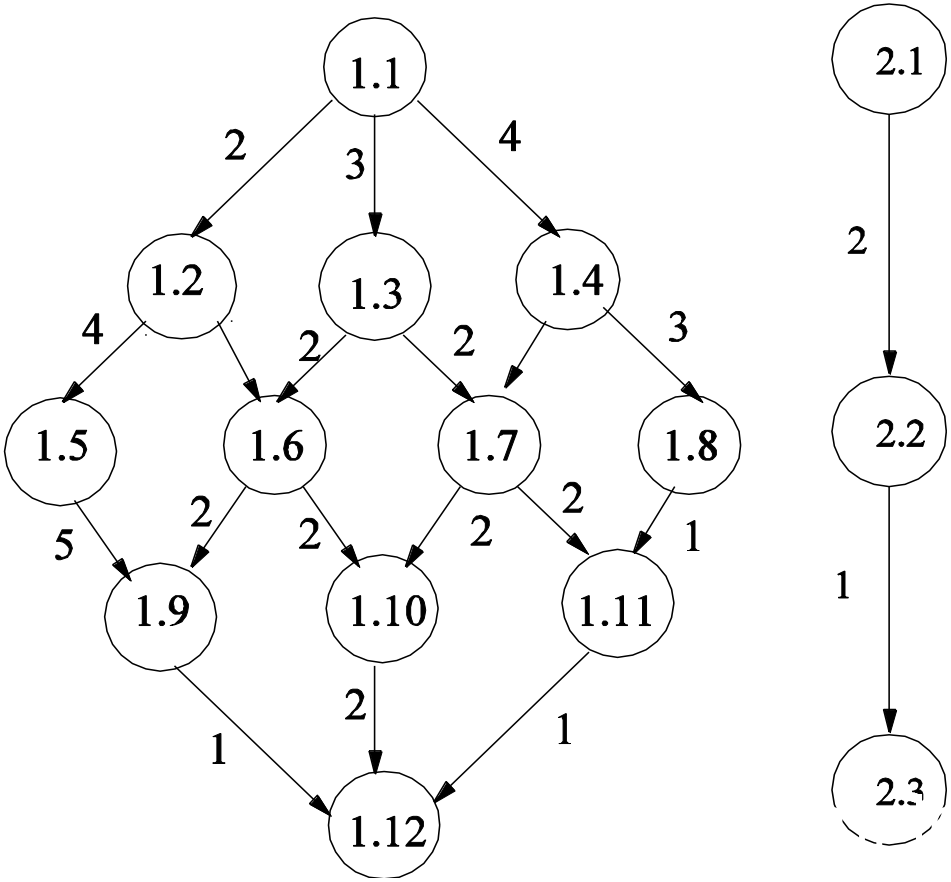
# Various Versions of Problem

- **Independent processes.**
  - All processes execute independently of each other.
  - There is no exchange of data between them.
  - Easier to schedule such processes.
- **Precedence related processes.**
  - One task depends on the other. For example, sense the environment (process 1) and then analyze the reading (process 2).
  - Process 1 has to be scheduled before process 2.
  - Harder to schedule such processes.
- **Homogeneous (identical) processors**.
  - Can assign process to any processor as all are the same.
- **Heterogeneous (different) processors.**
  - Tasks have different execution costs on different processors.
  - Need to select processor carefully.

# Task Model

- Applications can be modeled as Directed Acyclic Graphs (DAGs).
- Each DAG corresponds to a task/process.
- All tasks execute independently.
- Each task consists of a set of subtasks.
  - These subtasks are precedence related.
  - A subtask cannot start until it has received the results from all of its predecessors.
  - For example, task 1.10 has two predecessor tasks: 1.6 and 1.7. In addition, it has one successor task: 1.10.
- Each node will have an execution cost associated with it.
- Edges correspond to communication delays between specific nodes.
  - Message passing occurs between communicating processes.
  - Need to factor in the volume of traffic in the communication channel.

# DAG Representing Task Set




# Example Multiprocessor Algorithm

- Called "Task Duplication Algorithm" (TDS).
  - Duplicates certain tasks so that process finish times are minimized.
- Identical multiprocessors.
- Precedence related tasks.
- Goal is to schedule tasks on processors so that the makespan is minimized.
  - Makespan is the finish time of the exit task in the graph.
- **Algorithm works by calculating some mathematical quantities and then generating clusters (groups) of tasks.**
- **Each cluster is then assigned to a processor.**
- Need to watch out for communication costs.
- Let us go through the algorithm step by step.
- Algorithm originally appeared in the paper:
  - S Darbha and D P Agrawal, "Optimal Scheduling Algorithms for Distributed Memory Machines", IEEE Transactions on Parallel and Distributed Systems, 1998.

# TDS Algorithm

- The TDS algorithm works in the following sequence:
  1. Given the input graph, calculate the various mathematical quantities.
  2. Using the values of these quantities, generate node clusters.
  3. Assign each cluster to a processor.
  4. Generate schedule, paying special attention to the communication costs.

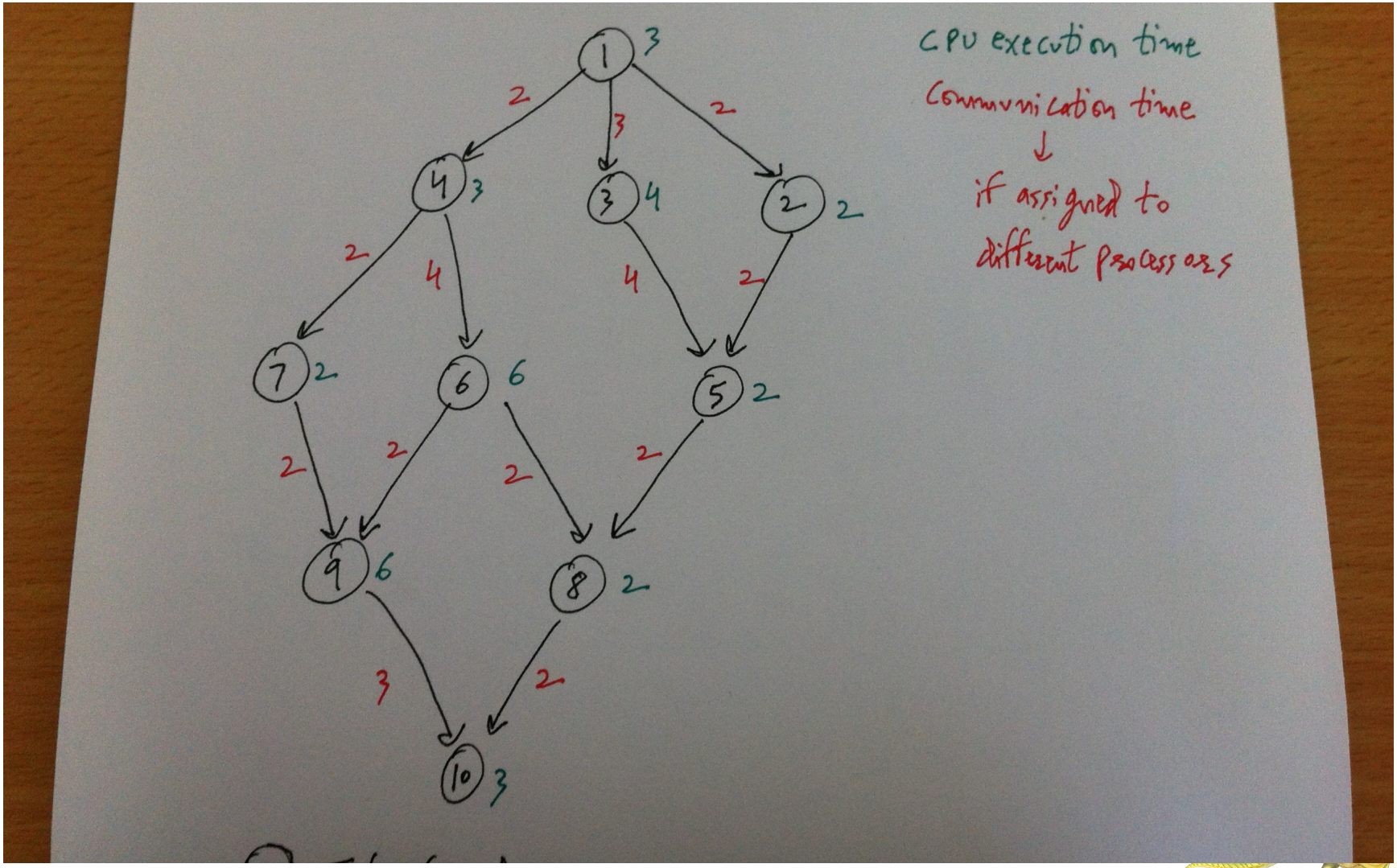
# Mathematical Quantities

- Need to calculate some mathematical quantities for each node in graph.
- Level: length of longest path (CPU costs only) from node to the last node. Communication costs are ignored.
  - Node clusters are assigned in increasing order of level values.
- Est (earliest start time): earliest time that a node can start.
  - Calculated top-down.
- Ect (earliest completion time): earliest time that a node can finish.
  - Calculated top-down.
- Fparent: favorite parent of a node. It is the parent that makes it wait the longest.
  - Assigning a node and its favorite parent to the same cluster will result in minimum completion times.

# Values of Mathematical Quantities

| Node | Level | Est | Ect | Fparent |
|---|---|---|---|---|
| 1 | 21 | 0 | 3 | - |
| 2 | 9 | 3 | 5 | 1 |
| 3 | 11 | 3 | 7 | 1 |
| 4 | 18 | 3 | 6 | 1 |
| 5 | 7 | 7 | 9 | 3 |
| 6 | 15 | 6 | 12 | 4 |
| 7 | 11 | 6 | 8 | 4 |
| 8 | 5 | 12 | 14 | 6 |
| 9 | 9 | 12 | 18 | 6 |
| 10 | 3 | 18 | 21 | 9 |

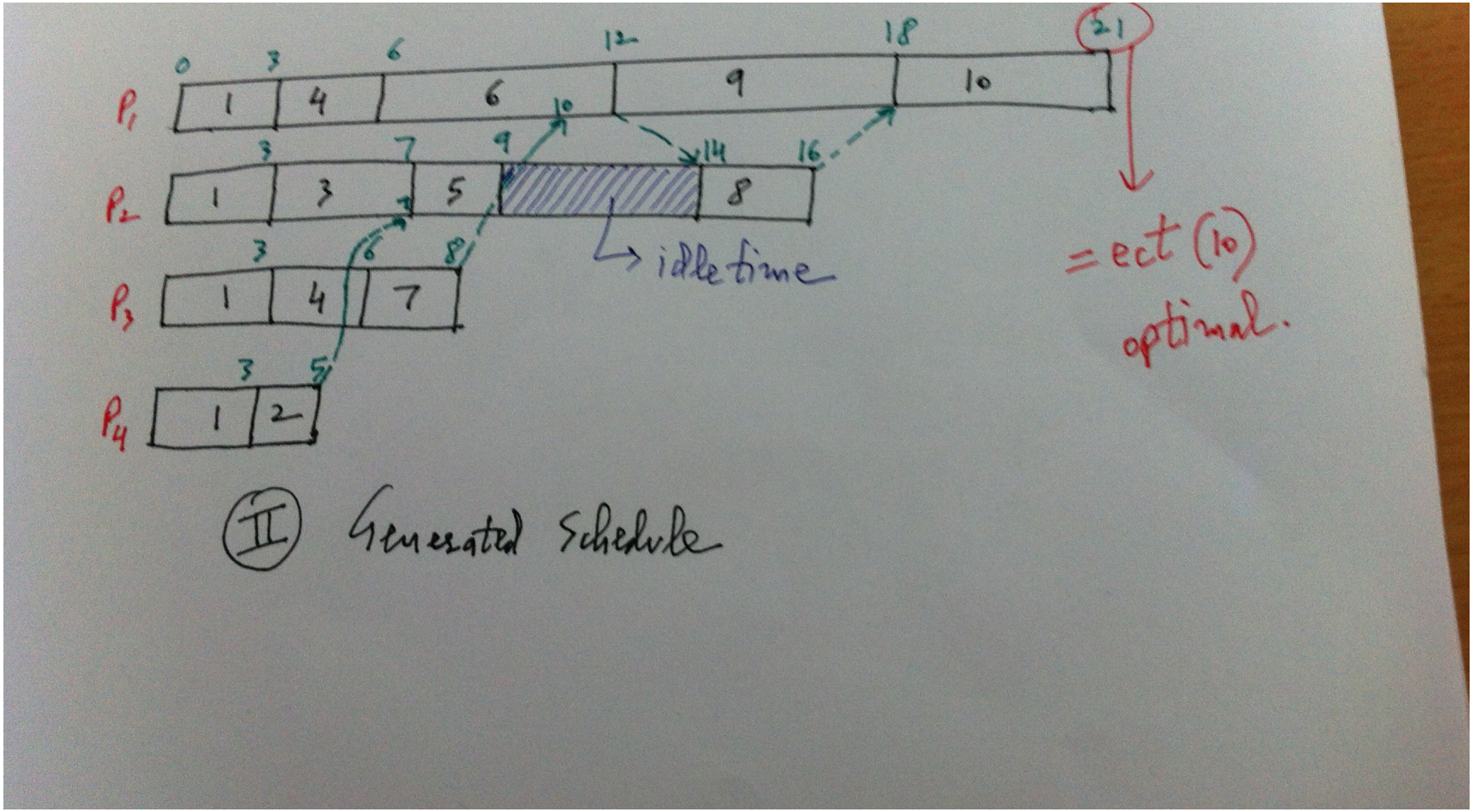
# Input Graph


CPU execution time
Communication time
↓
if assigned to
different processors
1 3
2 3
3 4
4 3
5 2
6 6
7 2
8 2
9 6
10 3
2 2
3 4
2 2
2 4
4 2
2 2
2 2
2 2
3 2

# Cluster Generation

- Clusters are assigned in increasing order of level values.
- The chain of favorite parents is followed, till the top node.
- If the favorite parent has already been assigned, another parent is chosen.
- In the example given, we obtain the following clusters:
  - 10 – 9 – 6 – 4 – 1
  - 8 – 5 – 3 – 1
  - 7 – 4 – 1
  - 2 – 1

# Generated Schedule



# CPU vs. IO Bound Processes

- We have in a previous lecture, discussed CPU bound & I/O bound processes.
- CPU bound: *spends most of the time doing CPU activity*.
- I/O bound: *spends most of the time doing I/O activity*.
  - These are generally interactive processes.
- **Question: Should the scheduler distinguish between CPU & I/O bound processes? How should it schedule them?**
- Better usage of computer's resources if I/O bound get priority over CPU bound. **How?**

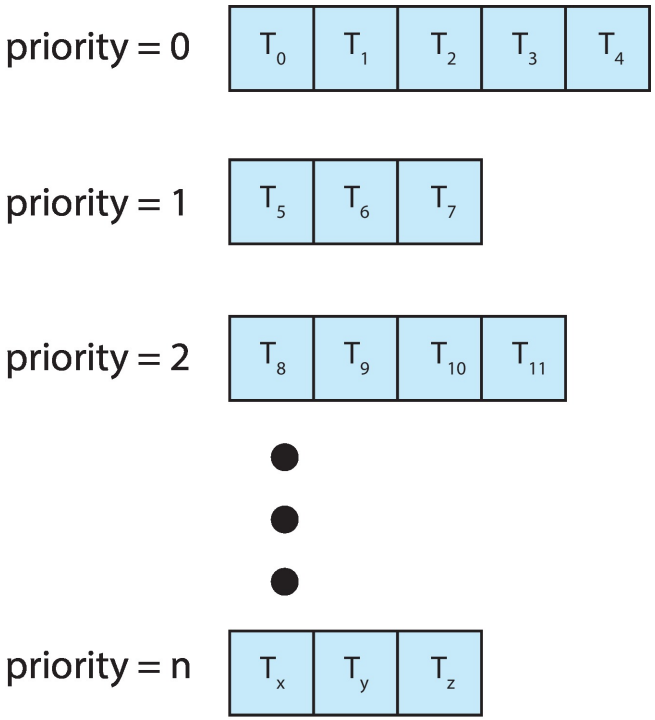
# Multilevel Queue

- So far, we assume that we have 1 ready queue and we pick processes to execute from that queue.
- Sometimes, it makes sense to group processes together (say, regular processes & real-time processes).
- **Key concept: multiple queues, specific scheduling policies for each queue.**
- Create separate process groups.
- For real-time processes, we can use earliest deadline first (EDF).
- For the non real-time, we could use round robin (RR).
- The real-time queue would have absolute precedence over the non real-time queue.
  - A non real-time process would be scheduled only if there is no real-time process in the system.

# Multilevel Queue

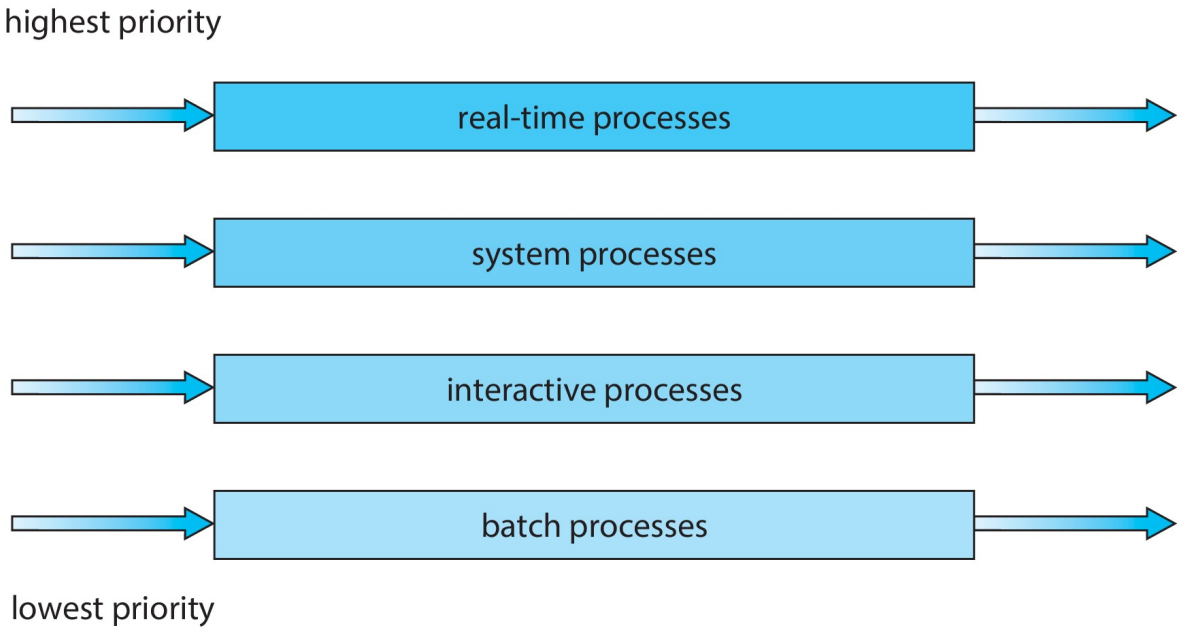
With priority scheduling, have separate queues for each priority.

Schedule the process in the highest-priority queue!




# Multilevel Queue

Prioritization based upon process type




# Multilevel Queue Scheduling

- **How to schedule among different queues?**
- **Option 1:**
  - Assign a fixed priority to each queue (EDF queue always has higher priority than RR queue).
- **Option 2:**
  - Assign a time slice to each queue.
  - Each queue gets it's own time slice, which it can use to schedule it's processes.
  - For example, give 80 % of CPU time to EDF queue and 20 % CPU time to RR queue.
  - Why give more to EDF queue?

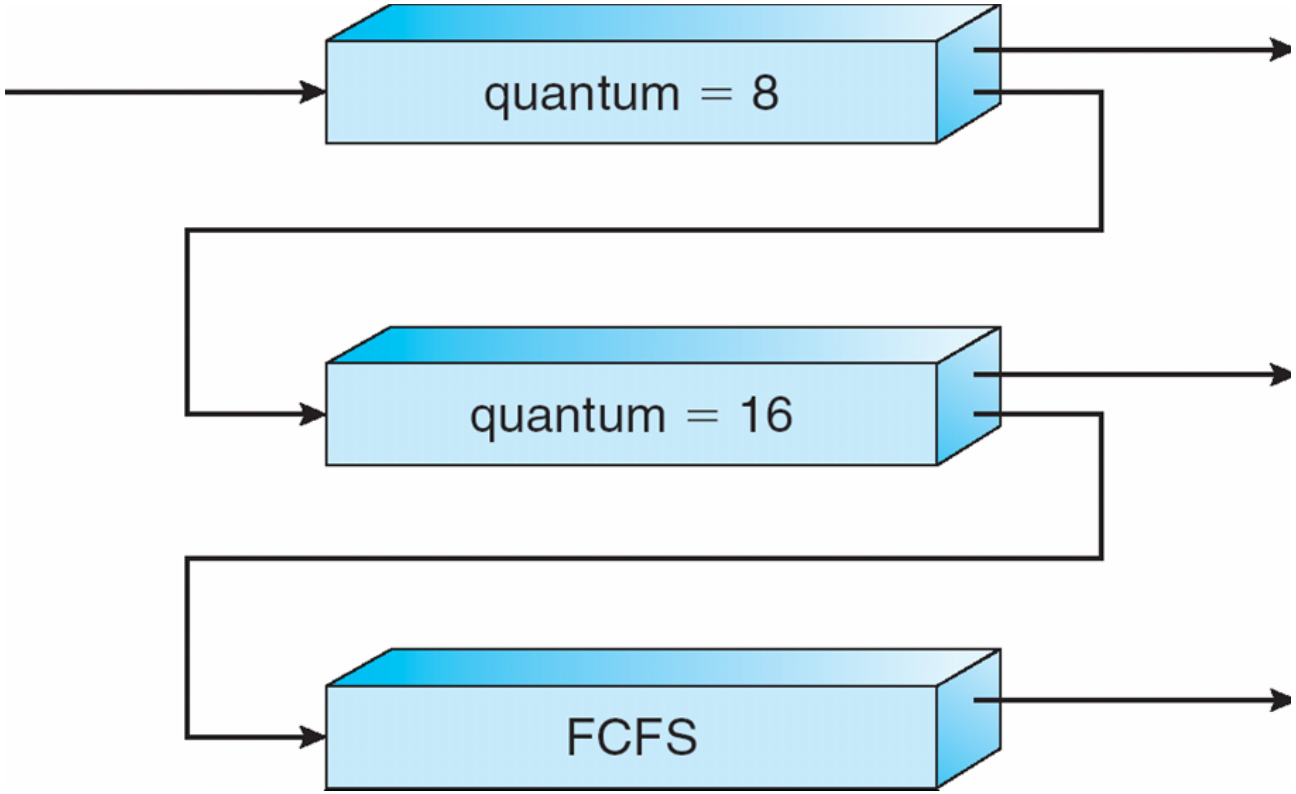
# Multilevel Feedback Queue

- In the previous case of multilevel queue scheduling, processes are fixed to a queue.
- They do not move from one queue to the other because their nature remains the same.
  - For example, a real-time process is a real-time process.
- What if we relax this assumption, and allow processes to move between queues?
  - There is some overhead due to queue movement.
  - But, our algorithm becomes more flexible. Advantage?
- This scheduling queue now becomes a **multilevel feedback queue**.
- Let's see an example.
- We have three queues:
  - $Q_0$ – RR with time quantum of 8 milliseconds (**highest priority**)
  - $Q_1$ – RR with time quantum of 16 milliseconds
  - $Q_2$ – FCFS (**lowest priority**)

# Multilevel Feedback Queue Scheduling

- Scheduling algorithm is as follows:
  - A new process enters queue $Q_0$.
  - When it gains CPU, it receives 8 milliseconds.
  - If it does not finish in 8 milliseconds, process is moved to the tail of queue $Q_1$.
  - At $Q_1$, process receives 16 additional milliseconds.
  - If it still does not complete, it is pre-empted and moved to the tail of queue $Q_2$. In this queue, it is scheduled in FCFS fashion.
  - How are jobs put in the queues?
    - Using FCFS.
- What kind of processes is this algorithm partial to?

# Multilevel Feedback Queues



# Multilevel Feedback Queues

- If queue $Q_0$ is empty, then the processes in $Q_1$ can start execution, starting from the head of the queue.
- Processes in queue $Q_2$ are executed only when $Q_0$ and $Q_1$ are empty.
- This scheduling algorithm gives the highest priority to any process with an execution time of 8 units, or less.
- Such a process will enter $Q_0$, and will eventually run and finish (as it's execution time ≤ 8, the time quantum size).
- The next highest priority is for processes whose execution time is between 9 and 24.
  - They will enter $Q_0$, run partially and then go to $Q_1$, where they will finish execution (time quantum in $Q_1$ = 16).
- Processes with execution times > 24 will eventually end up in $Q_2$, and will run only if $Q_0$ and $Q_1$ are both empty.
- Can there be starvation? How can we prevent it?

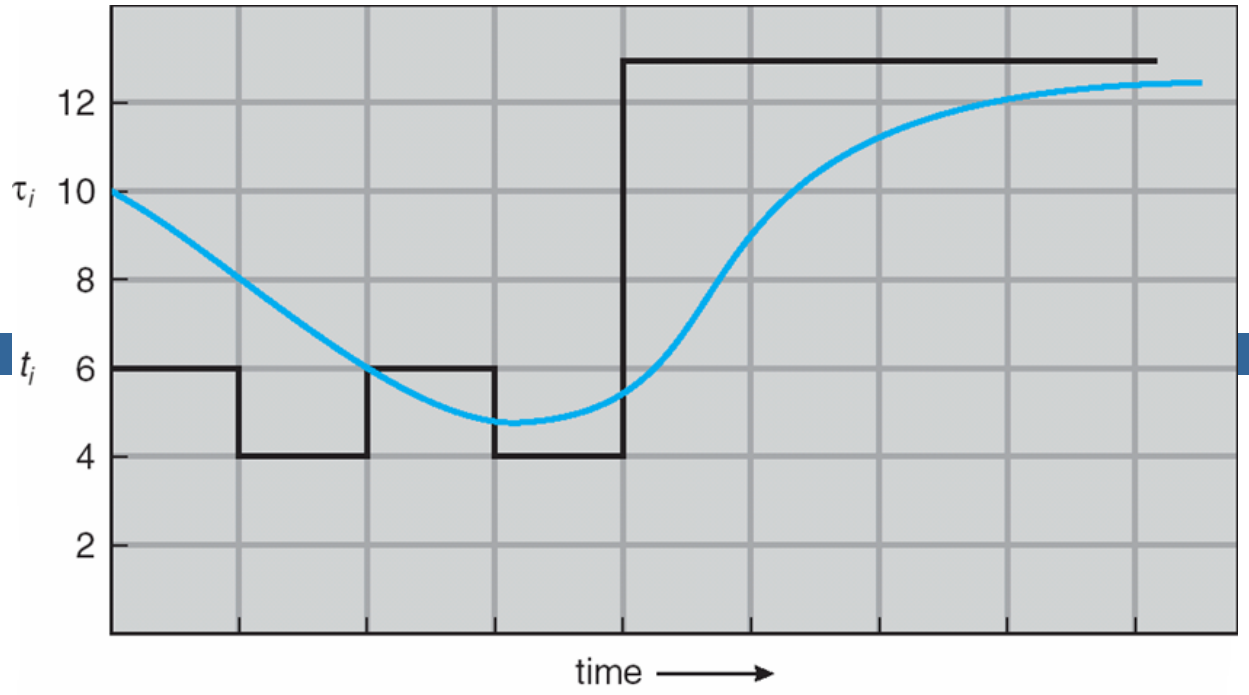
# Predict CPU Bursts: Knowing the Future

- In priority based scheduling (e.g. SJF), scheme is based on assigning priority based on CPU burst times.
  - Processes with smaller next CPU burst times are assigned a higher priority.
- In FCFS and RR, processes are serviced in their arrival order.
- For priority based schemes, the challenge is knowing the length of the process CPU bursts (don't know future CPU bursts of processes).
- One option is use "worst case" execution times.
  - Process execution time can be lower.
- Better approach is to "predict" current CPU burst with the help of past CPU bursts.
  - Expect that the next CPU burst will be similar to the previous CPU bursts.
  - Any examples we can think of?
- Use this predicted CPU burst in scheduling decisions.

# Formula for CPU Burst Prediction

- $T_{n+1} = \alpha * t_n + (1 - \alpha)*T_n$.
- $t_n$ is the actual length of the $n^{th}$ CPU burst, $T_{n+1}$ is the predicted value for the next CPU burst.
- Here, α has a value between 0 and 1.
- $T_n$ is the previous prediction used to schedule most recent ($n^{th}$) CPU burst.
  - This is a weighted value from past iterations.
- We can tweak α to give weightage to last burst or earlier bursts.
- If $\alpha = 0$, $T_{n+1} = T_n$. This means that the most recent CPU burst can be ignored.
- If $\alpha = 1$, $\tau_{n+1} = t_n$. This means that only the most recent CPU burst matters.
- To use both most recent and historical burst values, α value can be 0.5.

# 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 | 5 | 9 | 11 | 12 | ... |

# Practice Problems

- Consider the CPU burst prediction formula just discussed. What are the implications of assigning the following values to the parameters used by this technique?
  - $\alpha = 0$ and $T_0 = 100$ milliseconds.
  - $\alpha = 0.99$ and $T_0 = 10$ milliseconds.
- Which of the following algorithms could result in starvation?
  - FCFS
  - SJF
  - RR
  - Priority

# Round Robin with Priority

| Process | Priority | Burst | Arrival |
|---|---|---|---|
| $P_1$ | 40 | 20 | 0 |
| $P_2$ | 30 | 25 | 25 |
| $P_3$ | 30 | 25 | 30 |
| $P_4$ | 35 | 15 | 60 |
| $P_5$ | 5 | 10 | 100 |
| $P_6$ | 10 | 10 | 105 |

- Time quantum, $q$ = 10.
- Higher number => higher priority.
- Generate schedule chart.
- CPU utilization?
- Wait times?
- Turn round times?

# Operating System Examples

- How do actual operating systems schedule processes?
  - Both user and kernel.
- We now discuss three case studies:
  - Solaris scheduling (UNIX variant by Sun Microsystems).
  - Windows XP scheduling.
  - Linux scheduling.

# Linux Scheduling Through Version 2.5

- Version 2.5 moved to constant order *O*(1) scheduling time:
  - Preemptive, priority based.
  - Two priority ranges: time-sharing and real-time (rt: can see on running *top* command).
  - **Real-time** range from 0 to 99 and **nice** value from 100 to 140.
    - Can have multiple processes with the same priority value.
  - Numerically lower values indicate higher priority.
  - Higher priority gets larger time quantum *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.

# Linux Scheduling in Version 2.6.23 +

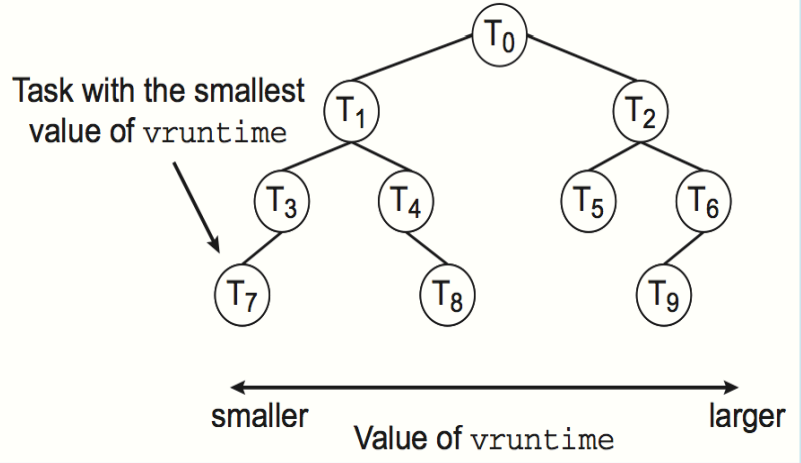
- ***Completely Fair Scheduler*** (CFS)
- **Scheduling classes (multi level scheduling queue)**
  - Each has specific priority.
  - Scheduler picks highest priority task in highest scheduling class.
  - Scheduler assigns portion of CPU (quantum) to each task.
  - 2 scheduling classes included, others can be added:
    - Default (time shared)
    - real-time
- Quantum calculated based on **nice value** from -20 to +19
  - Lower nice value indicates higher priority.
  - Default nice value is 0.
  - Tasks with lower nice values receive a higher proportion of the CPU.
  - Scheduler calculates **target latency** – interval of time during which every run-able task should run at least once.
  - Proportions of CPU are assigned based on the target latency.
  - Target latency can increase if say number of active tasks increases.

# Linux Scheduling

- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`.**
  - Virtual run time is associated with decay factor based on priority of task: lower priority task has a higher decay rate.
  - Normal default priority yields virtual run time = actual run time.
  - Default priority task (nice value = 0) runs for 200 msec, it's vruntime = 200 msec.
  - High priority task (nice < 0) runs for 200 msec, it's vruntime < 200 msec.
  - Low priority task (nice > 100) runs for 200 msec, it's vruntime > 200 msec.
- To decide next task to run, scheduler picks task with lowest virtual run time value.
- As an example, consider two tasks – one CPU bound & one I/O bound. Assume both tasks have the same nice value. **How are they scheduled?**

# CFS Performance

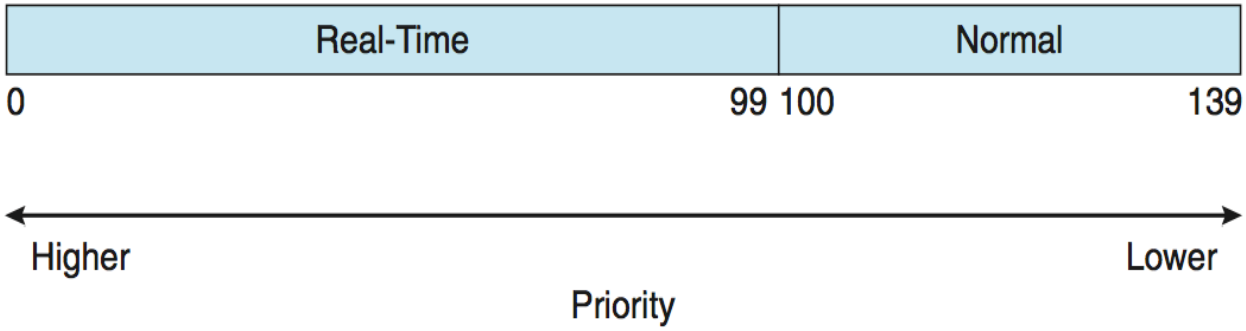
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:




When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where $N$ is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

# Linux Scheduling (Cont.)

- Linux supports real-time scheduling according to POSIX.1b standards.
  - Real-time tasks have static or dynamic priorities.
- Normal tasks assigned priority based on their nice values:
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139




# Windows Scheduling

- Windows uses priority-based preemptive scheduling.
- Highest-priority process runs next.
- **Dispatcher** is the name of the scheduler here.
- Process runs until: (1) it blocks, (2) it uses time slice, (3) it is preempted by higher-priority process.
- Real-time processes can preempt non-real-time processes.
- 32-level priority scheme.
- **Variable class** is 1-15, **real-time class** is 16-31.
- Queue for each priority (multilevel queue).
- If no run-able task, runs special task, called **idle task.**

# Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong:
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS.
  - Priority class of process can be altered using API function `SetPriorityClass ( )`
  - Priorities in all classes are variable (meaning the priority can change), except REALTIME.
- Moreover, a process within a given priority class has a relative priority.
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE.
- Priority class and relative priority combine to give numeric priority.
- Base priority of a process is NORMAL within the class.
  - 24 for real-time, 4 for idle.
- If quantum expires, priority lowered, but never below base.

# Windows Priorities

| | real-time | high | above normal | normal | below normal | idle priority |
|---|---|---|---|---|---|---|
| time-critical | 31 | 15 | 15 | 15 | 15 | 15 |
| highest | 26 | 15 | 12 | 10 | 8 | 6 |
| above normal | 25 | 14 | 11 | 9 | 7 | 5 |
| normal | 24 | 13 | 10 | 8 | 6 | 4 |
| below normal | 23 | 12 | 9 | 7 | 5 | 3 |
| lowest | 22 | 11 | 8 | 6 | 4 | 2 |
| idle | 16 | 1 | 1 | 1 | 1 | 1 |

# Solaris

- Priority-based scheduling.
- Six classes available:
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- A process can be in one class at a time.
- Each class has its own priority scheduling algorithm.
- E.g.: time sharing supports dynamic priorities and time quantum of variable lengths using multi-level feedback queue.
- **Inverse relationship between priority and time quantum.**

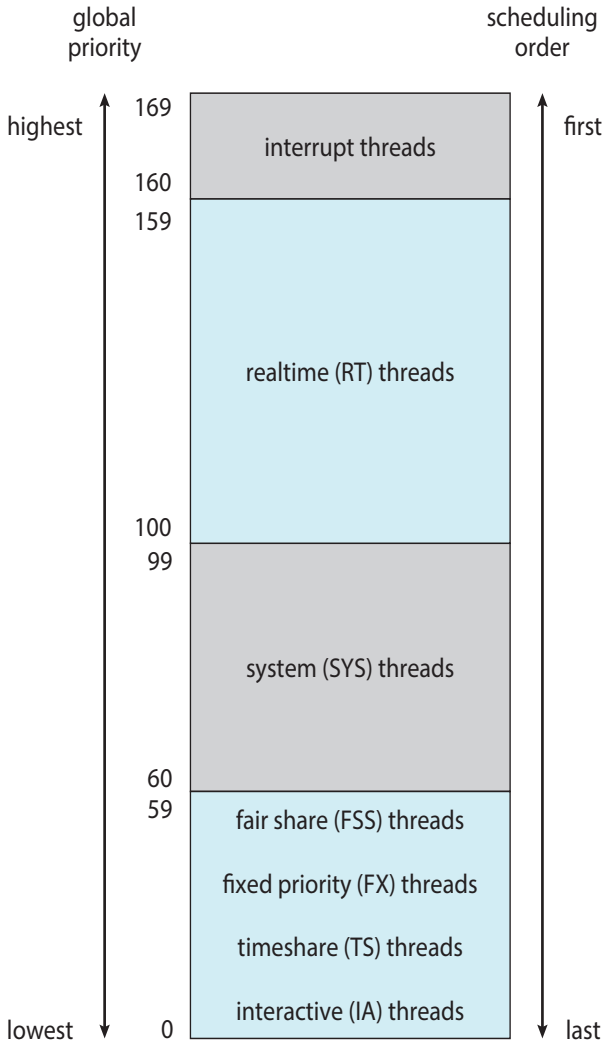
# Solaris

- Higher priority number means higher priority, & vice versa.
- Higher priority process gets smaller time slice, & vice versa.
- Interactive processes (I/O bound) get higher priority.
- CPU bound processes get lower priority.
- Table on next slide shows some priority & time quantum values for time sharing and interactive processes.
- Some observations:
  - Lower priority processes get larger time slices (CPU bound).
  - Higher priority processes get smaller time slices (I/O bound).
  - *Time quantum expired:* new priority of thread once it uses up it's entire quantum without blocking. **Priorities for these lowered.**
  - *Return from sleep*: priority of process that is returning from sleep (I/O wait). **Priorities for these are increased.**

# Solaris Dispatch Table

| priority | time quantum | time quantum expired | return from sleep |
|---|---|---|---|
| 0 | 200 | 0 | 50 |
| 5 | 200 | 0 | 50 |
| 10 | 160 | 0 | 51 |
| 15 | 160 | 5 | 51 |
| 20 | 120 | 10 | 52 |
| 25 | 120 | 15 | 52 |
| 30 | 80 | 20 | 53 |
| 35 | 80 | 25 | 54 |
| 40 | 40 | 30 | 55 |
| 45 | 40 | 35 | 56 |
| 50 | 40 | 40 | 58 |
| 55 | 40 | 45 | 58 |
| 59 | 20 | 49 | 59 |

# Solaris Scheduling




# Solaris Scheduling (Cont.)

- Scheduler converts scheduling class-specific priorities into a per-process global priority.
  - Thread with highest priority runs next
  - Runs until: (1) it blocks, (2) it uses its time slice, (3) it is preempted by higher-priority process.
  - Multiple processes at same priority selected via RR.

# Algorithm Evaluation

- A number of scheduling algorithms – FCFS, SJF, Priority based (RM, EDF), RR.
  - Can be preemptive or non preemptive.
- Which is the best algorithm?
  - Not an easy question to answer.
  - Answer depends on the specific requirements of the processes. For example:
    - Maximizing CPU utilization – non preemptive.
    - Minimizing average waiting time - SJF.
- Two techniques can answer this question:
  - Analytic modeling.
  - Simulation.

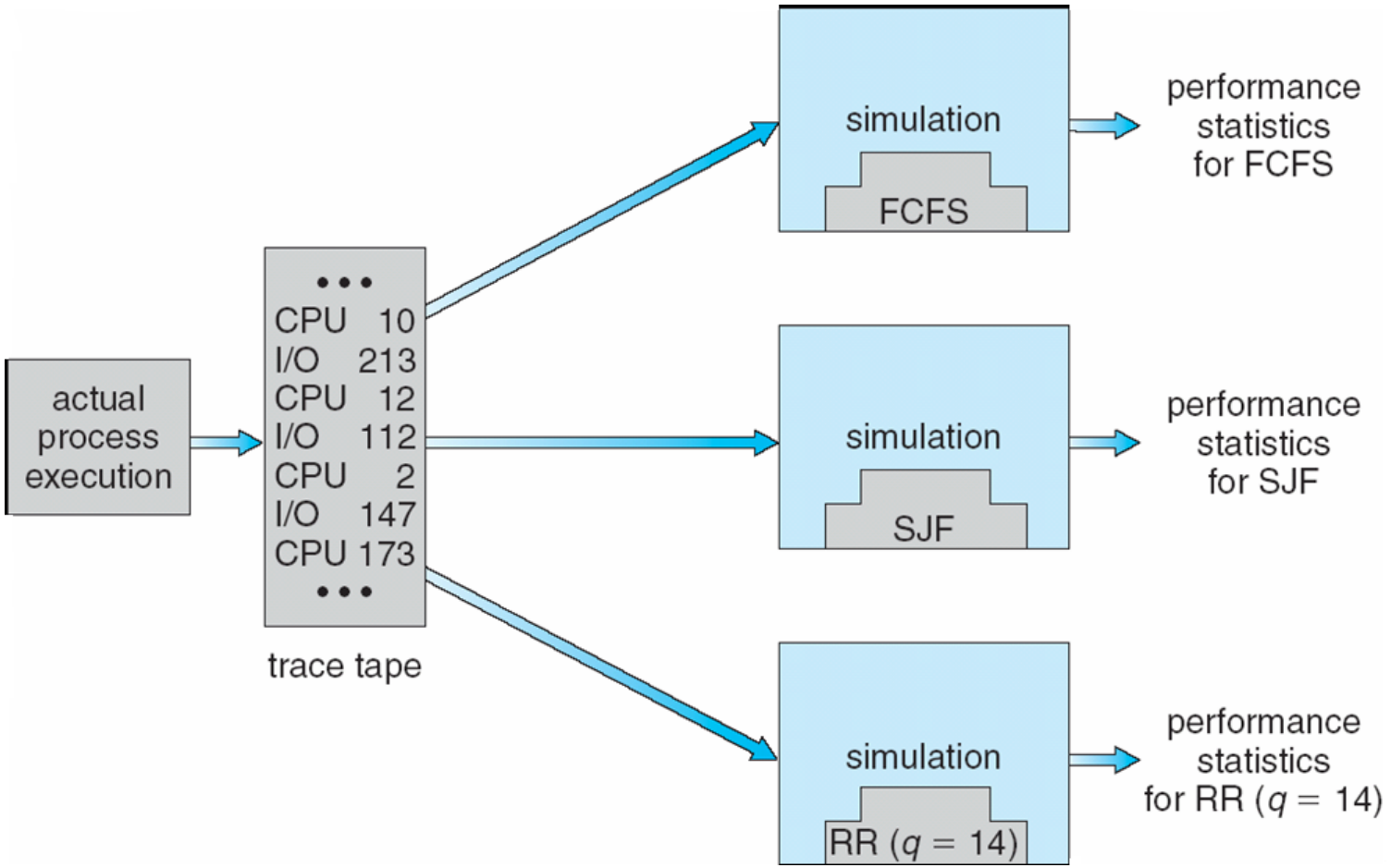
# Analytic Evaluation

- Use the given scheduling algorithm and the system workload to produce a formula or number that evaluates the performance of the algorithm for the workload.
- For example, Earliest Deadline First (EDF) is guaranteed to meet all deadlines if the combined utilization of all processes <=1.
- Run the various algorithms on the same input.
- Compare the results based on performance metrics.
- Over a set of example inputs, this technique may indicate trends that can be analyzed and proven separately.
  - For example, if minimizing the average wait time is the sole criteria, then SJF algorithm will be the best choice.

# Simulations

- Aim is to create a computer program that simulates the scheduling algorithm.
- The simulator is then fed with a variety of inputs.
- The results can then be analyzed.
- If the execution costs are increased, what will happen to the average waiting time?
- If the execution costs are increased, will the processes still meet the deadlines?
  - If algorithm A still meets the deadlines and algorithm B does not, then we could say that A is better.
- The execution costs could be generated using a uniform distribution.
  - Given a range, any number can be picked with equal probability.

# Evaluation of CPU schedulers by Simulation




# End of Chapter 5