# Operating Systems
# COMP2006

## CPU Scheduling
## Lecture 4

# CPU Scheduling

**References:** Silberschatz, Galvin, and Gagne, *Operating System Concepts*, Chapter 6

**Topics:**

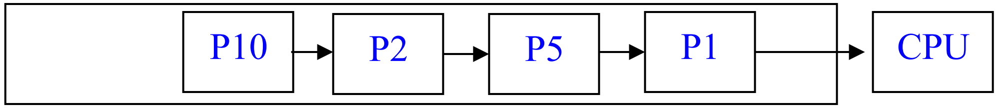
- Scheduling concepts.
- CPU scheduling algorithms.
- CPU scheduling evaluations.

# CPU Scheduling

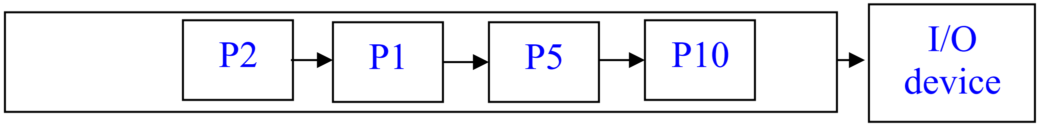
***Why schedule the CPU?***

- The objective of multiprogramming is to maximize CPU utilization.
  - Select and run one process in ready queue when CPU is available.
- Process execution consists of a cycle of CPU execution and I/O wait; *i.e.*, CPU and I/O burst cycle.
  - Processes alternate between these two activities
  - A process terminates after the last CPU burst
- Processes can be:
  - CPU-bound process: very long CPU burst.
  - I/O-bound process: short CPU burst.

**Ready Queue:**




**Device Queue:**



# Why (cont.)

.

.

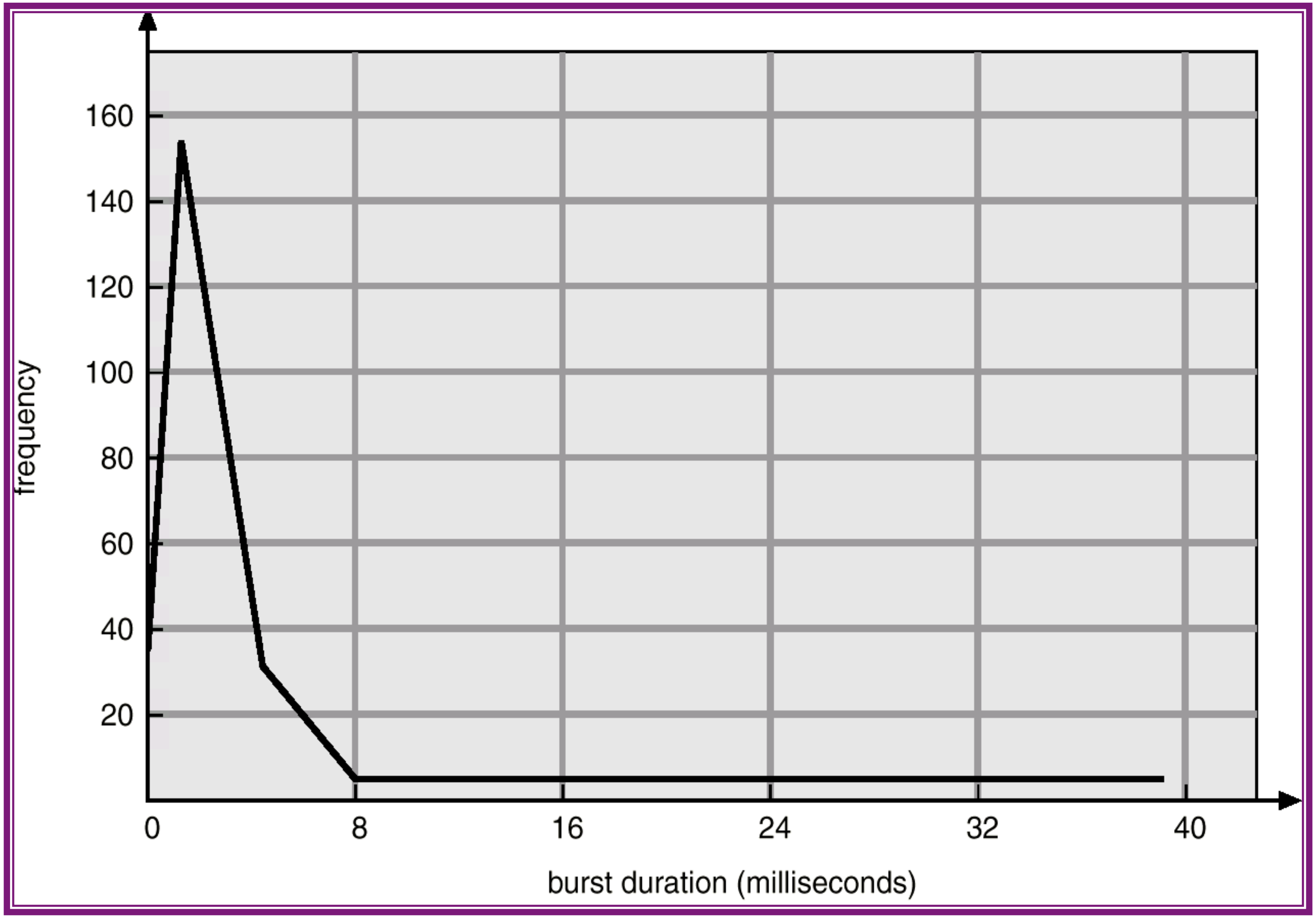
**Alternating sequence of CPU and I/O bursts**

| | |
|---|---|
| x := 0;<br>read from file | CPU burst |
| Wait for I/O | I/O burst |
| x := x + y;<br>write to file | CPU burst |
| Wait for I/O | I/O burst |
| x := x + z;<br>write to file | CPU burst |

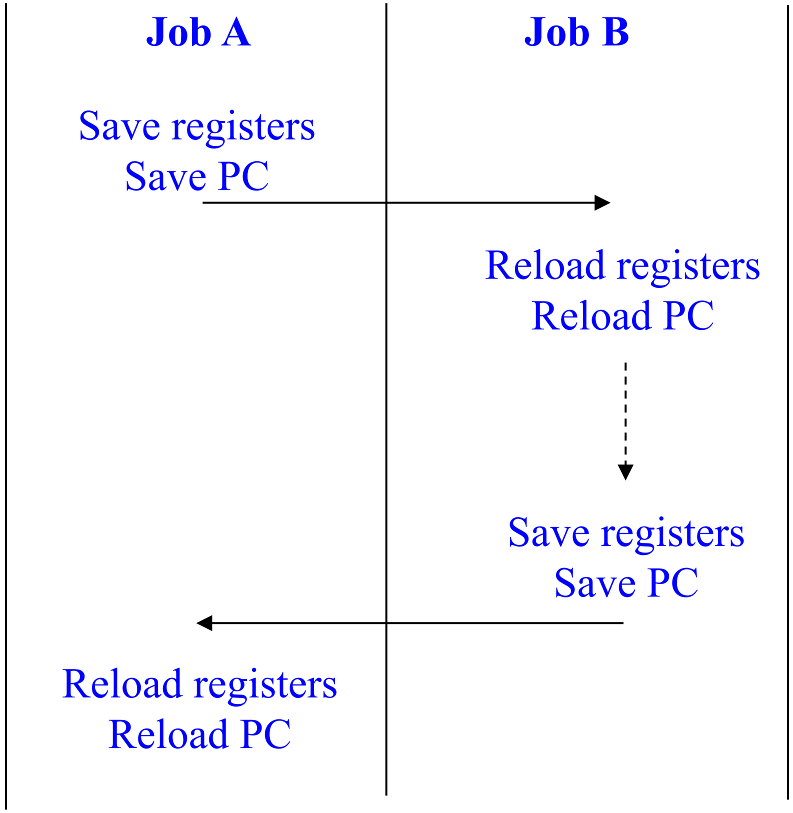
.

.

# Histogram of CPU-burst Times



# How and when do we switch CPU?

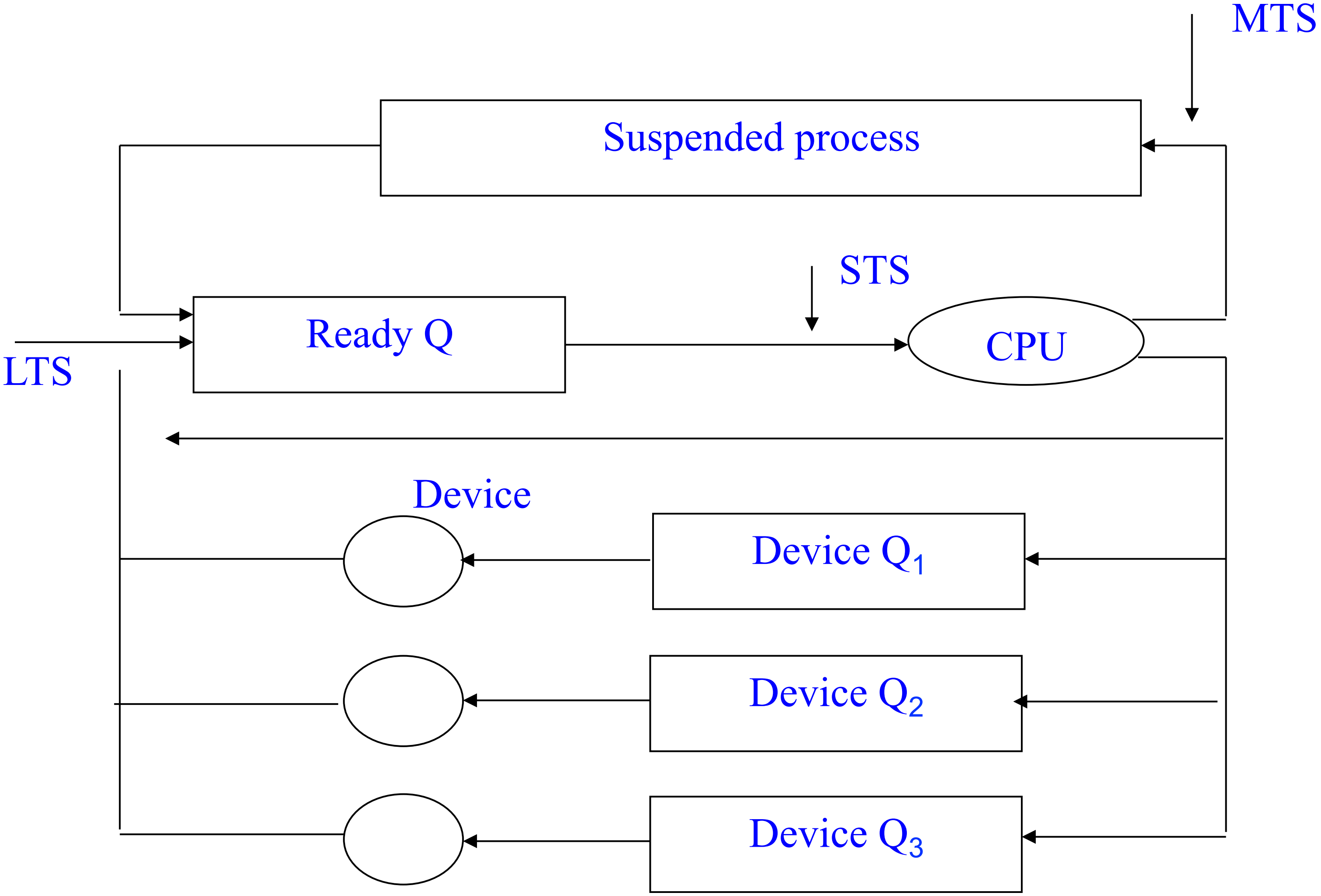



- From running to waiting state (*e.g.*, I/O request, or calling wait()).
- From running to ready state (*e.g.*, timer off).
- Process terminates.

# CPU schedulers

- CPU scheduler selects one of the processes in memory ready for execution, and allocates the CPU to it.
  - also called as Short Term Scheduler (STS)
  - Ready queue can be a FIFO, priority, a tree, or unordered linked list.
  - The content of the queue: PCBs of the processes.
  - This scheduler must be quick. Why?
- Other schedulers:
  - Medium term scheduler (MTS) → Swaps jobs in and out of memory to reduce contention for the CPU.
  - Long term scheduler (LTS) → Determines which jobs are admitted
    - LTS is executed less frequently than STS
      - It is invoked only when a job finishes.
    - LTS controls the degree of multiprogramming
      - It must select carefully between CPU bound and I/O bound jobs.
    - LTS may be absent on time-sharing systems
      - MTS is added in this case.

# Queueing diagram



# Preemption

- Scheduling can be *preemptive* or *non-preemptive*.
- Non-preemptive: once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the waiting state.
  - Example: Windows 3.x
- Preemptive: otherwise.
  - Example: Windows 95 on ward, Mac OS X, Linux.
- Preemptive scheduling needs special hardware (*e.g*., timer).
- Preemptive scheduling can result in a race condition.
  - What happens when one process updating a shared data is preempted, and the second process is accessing the data?
- Be careful when pre-empting a kernel process
  - What happens if the kernel is in the middle of changing important kernel data involving the process? See page 186.

# Dispatcher

- Dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.
- This function involves:
  - Switching context.
  - Switching to user mode.
  - Jumping to proper location in the user program to restart that program.
- Dispatch latency: the time it takes for the dispatcher to stop one process and start running another
  - must be short because every context switch invokes dispatcher.

# Scheduling Criteria

- Criteria are used for comparing the scheduling algorithms to determine which algorithm is better.
- Some scheduling criteria include:
  - **CPU utilisation**: percent usage of CPU → the higher the better.
  - **Throughput**: #processes that complete their execution per time unit → the higher the better.
  - **Turnaround time**: amount of time to execute a particular process, *i.e.*, sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O → the shorter the better.
  - **Waiting time**: the sum of the periods a process spent waiting in the ready queue → the shorter the better.
  - **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.

# Criteria (cont.)

- For interactive systems:
  - Minimise variance in response time.
    - A system with predictable response is more useful than a system that is faster on average.
  - In general: minimise the maximum response time *or* average response time.

- Should we also include *fairness* in the criteria?
  - Make sure that each process gets its fair share of the CPU time.

# Scheduling Algorithms

## (a) First Come First Served (FCFS) Scheduling

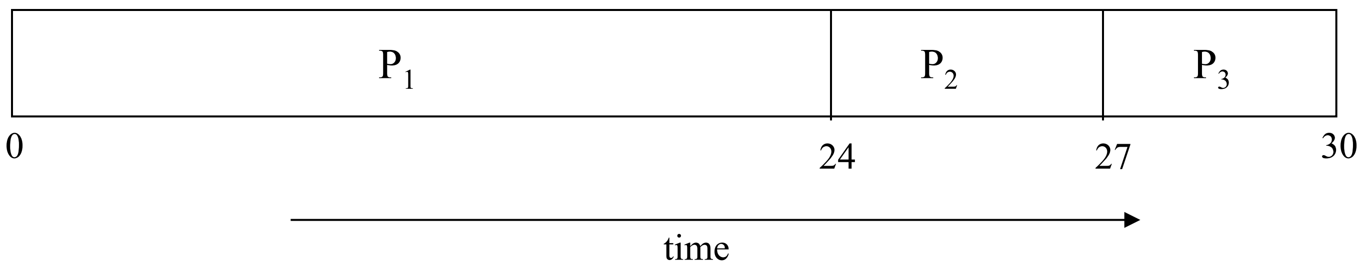
- A process that requests CPU first is allocated the CPU first.
- Easy implementation with a FIFO queue.
- Non-preemptive.
- Performance: poor waiting time, turnaround time and response time.

| Process | Burst time |
|---|---|
| 1 | 24 |
| 2 | 3 |
| 3 | 3 |

# Examples

Case (i): Processes arrive in the order P1, P2, P3

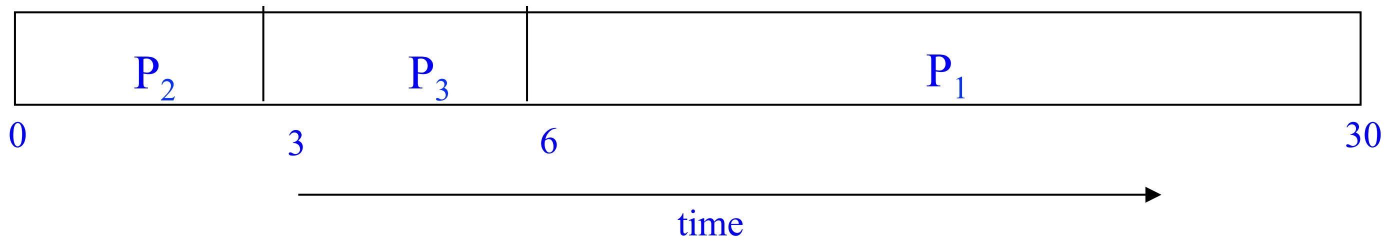
The Gantt chart for the schedule is:




- Waiting time for P1 = 0; P2 = 24; P3 = 27.
- Average waiting time: $\dfrac{0+24+27}{3} = \dfrac{51}{3} = 17$
- Average turnaround time: $\dfrac{24+27+30}{3} = \dfrac{81}{3} = 27$
- Waiting time = turnaround time – burst time

# Examples (cont.)

Case (ii) Processes arrive in the order P2, P3, P1




Waiting time for P1 = 6; P2 = 0; P3 = 3.

Average waiting time = $\frac{6+0+3}{3} = \frac{9}{3} = 3$ → better than (i).

Average turnaround time: $\frac{3+6+30}{3} = \frac{39}{3} = 13$

**Convoy effect:** short processes behind long process.
Consider one CPU bound job and several I/O bound jobs → lower CPU utilisation and device utilisation.

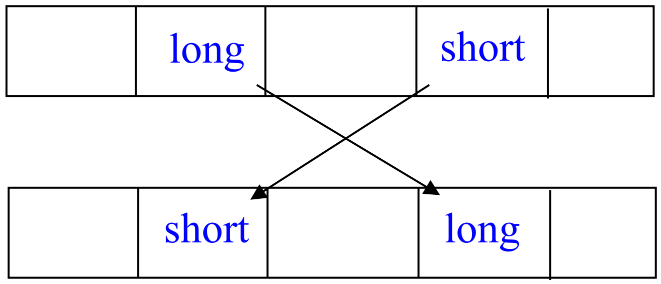
# Algorithms (cont.)

(b) Shortest Job First (SJF) Scheduling

- Associate with each process the length of its next CPU burst, and use this length to schedule the process with shortest time.
- Two schemes:
  - Non-preemptive – once the CPU is given to the process, it cannot be pre-empted until it completes its CPU burst.
  - Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt
    - Also known as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time, turnaround time, and response time **for a given set of processes.**

# SJF (cont.)

- Moving a short job before a long one decreases the waiting time of the short job more than it increases the waiting time of the long job → therefore the average waiting time decreases.




**Simple proof**

Consider the case of four processes with run times of $a$, $b$, $c$, and $d$. The first process finishes at time $a$, the second process finishes at time $a + b$, *etc*. The average turn around time is $\frac{4a + 3b + 2c + d}{4}$.

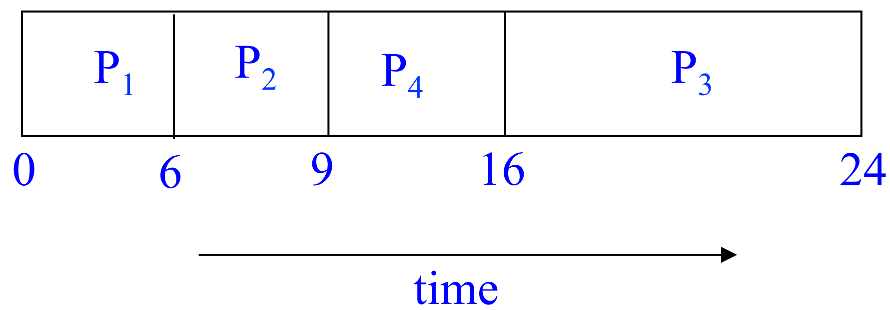
It is clear that $a$ contributes more to the average than the other times, so it should be the shortest process, with $b$ next, then $c$, and finally $d$ as the longest since it affects only its own turn around time.

**Problem:** How to know which of the currently run-able processes have the shortest CPU burst?

# Example (non-preemptive SJF)

| Process | Arrival time | Burst time |
|---|---|---|
| 1 | 0 | 6 |
| 2 | 1 | 3 |
| 3 | 2 | 8 |
| 4 | 3 | 7. |

The Gantt chart for the schedule:




Waiting time for $P_1 = 0$; $P_2 = 6\text{-}1$; $P_3 = 16\text{-}2$; $P_4 = 9\text{-}3$.

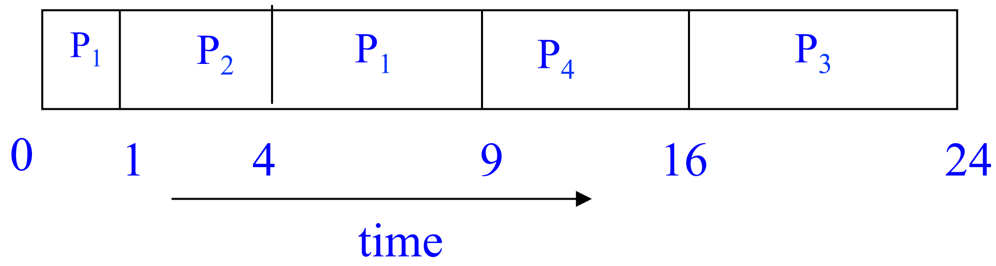
Average waiting time = $\frac{0+5+14+6}{4} = \frac{25}{4} = 6.25$

Average turnaround time: $\frac{(6-0)+(9-1)+(16-3)+(24-2)}{4} = \frac{49}{4} = 12.25$

# Example (preemptive SJF)

| Process | Arrival time | Burst time |
|---|---|---|
| 1 | 0 | 6 |
| 2 | 1 | 3 |
| 3 | 2 | 8 |
| 4 | 3 | 7 |

The Gantt chart for the schedule:




Waiting time for P1 = 4-1; P2 = 1-1; P3 = 16-2; P4 = 9-3

Average waiting time = $\frac{3+0+14+6}{4} = \frac{23}{4} = 5.75$

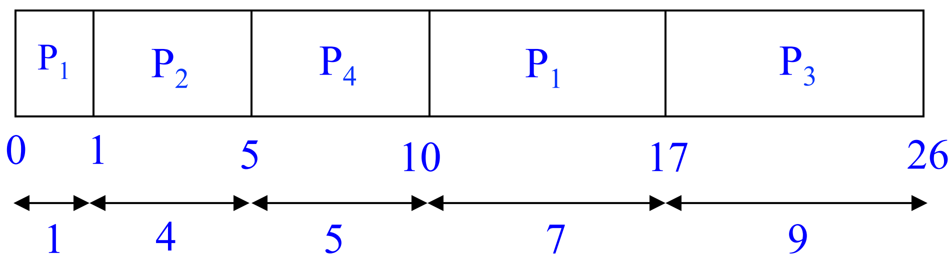
Average turnaround time: $\frac{(9-0)+(4-1)+(24-2)+(16-3)}{4} = \frac{47}{4} = 11.75$

**Note:** waiting time of a process is its turnaround time *minus* its burst time.

# Other example (SJF – preemptive)

| Process | Arrival time | Burst time |
|---|---|---|
| 1 | 0 | 8 |
| 2 | 1 | 4 |
| 3 | 2 | 9 |
| 4 | 3 | 5 |

The Gantt chart for the schedule:




Average turnaround time: $$\frac{(17-0)+(5-1)+(26-2)+(10-3)}{4}=13$$

Average turnaround time for non-pre-emptive SJF: 14.25 seconds.

# Determining Length of next CPU Burst

**<u>Problems</u>:** How do we know the length of the job?

- Can only estimate the length
- Can be done using the length of previous CPU burst, using exponential averaging

$$\tau_{n+1} = \alpha . t_n + (1 - \alpha) . \tau_n$$

$t_n =$ actual length of $n^{th}$ CPU burst.

$\tau_{n+1}$ = predicted value for the next burst.

$\alpha : 0 \leq \alpha \leq 1$ controls the weight of the recent *vs.* past history.

- More commonly $\alpha = \frac{1}{2}$, so recent history and past history are equally weighted; The initial $\tau_0$ can be defined as a constant or as an overall system average.

# Examples of Exponential Averaging

$\alpha = 0 \rightarrow \tau_{n+1} = \tau_n$; recent history does not count

$\alpha = 1 \rightarrow \tau_{n+1} = t_n$; only the actual last CPU burst counts

If we expand the formula, by repeatedly substituting $\tau_n = \alpha t_{n-1} + (1 - \alpha)\, \tau_{n-1}$,

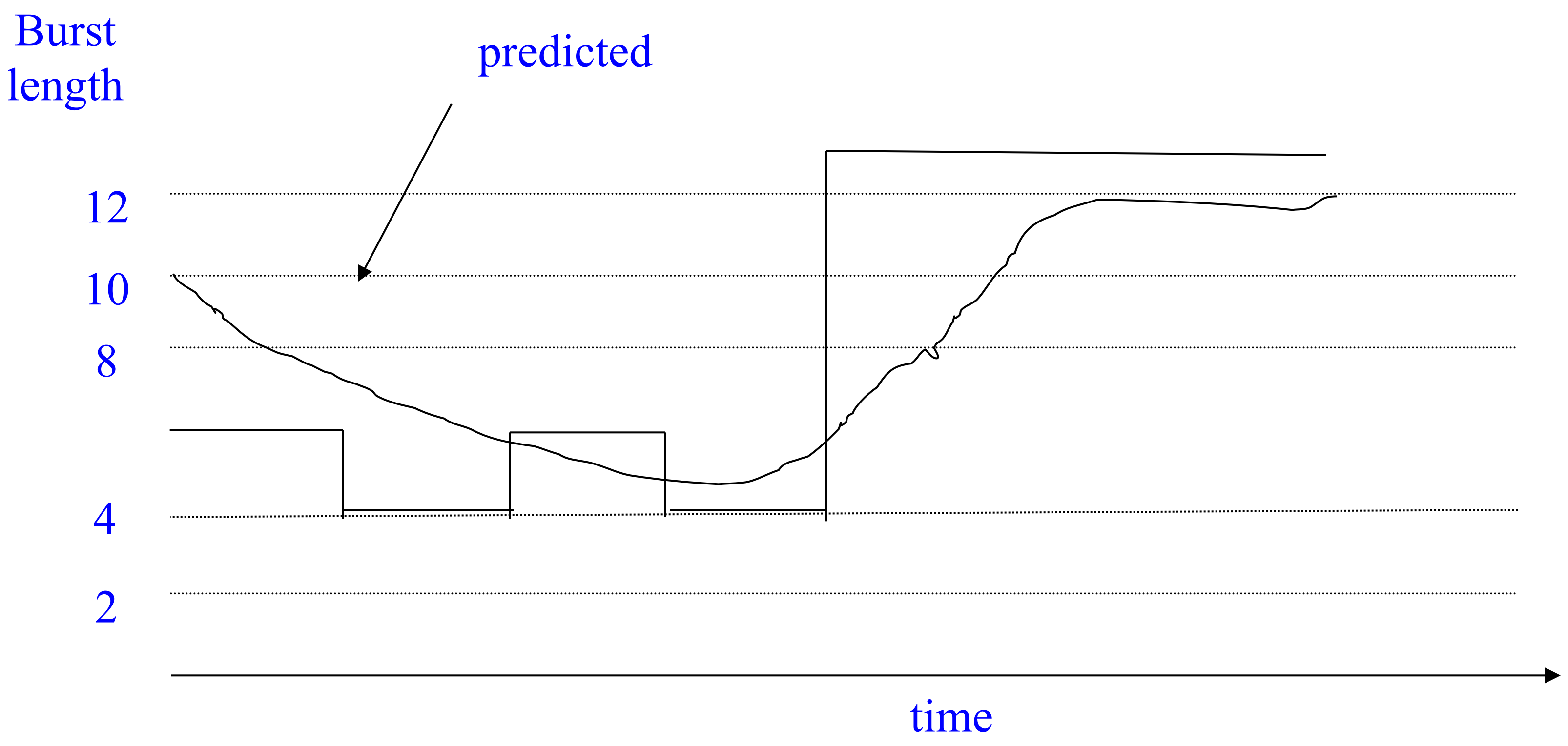
we get:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\, \alpha t_{n-1} + \ldots + (1 - \alpha)^j\, \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$$

Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

# Example: $\alpha = ½$ and $\tau_0 = 10$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| CPU | | 6 | 4 | 6 | 4 | 13 | 13 | 13 |
| Guess | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 |



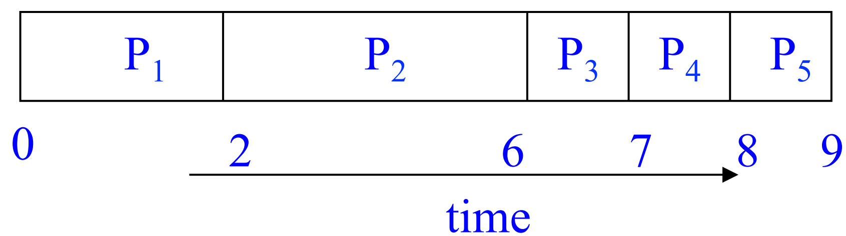
# SJF (cont.)

- SJF is only optimal when all jobs are available simultaneously.

| Process | Arrival time | Burst time |
|---|---|---|
| 1 | 0 | 2 |
| 2 | 0 | 4 |
| 3 | 3 | 1 |
| 4 | 3 | 1 |
| 5 | 3 | 1 |

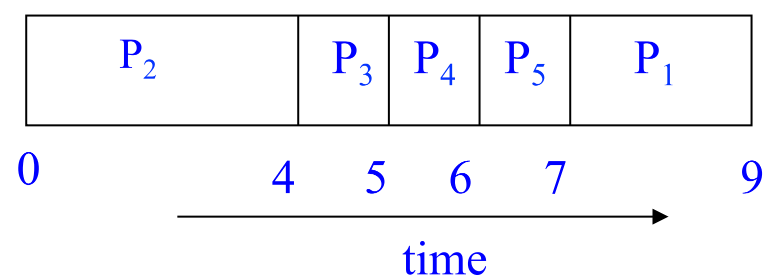
If we run them in the order 1, 2, 3, 4, 5, average waiting time =

$$\frac{(0)+(2-0)+(6-3)+(7-3)+(8-3)}{5} = 2.8$$




If we run them in the order 2, 3, 4, 5, 1 (*non- SJF*), we get a *better* average waiting time =

$$\frac{(0)+(4-3)+(5-3)+(6-3)+(7-0)}{5} = 2.6$$



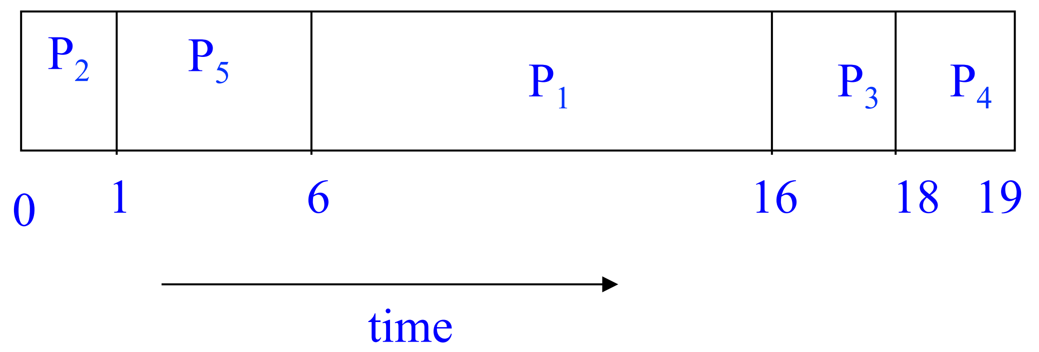
# Priority Scheduling

- A priority number (integer) is associated with each process.
- The CPU is allocated to the process with the highest priority (low number → high priority).
  - Preemptive → preempt the CPU if the priority of the arrived process is higher than the priority of the currently running process.
  - Non-preemptive.
- Internally defined priority - based on some measurable quantity:
  - Time limits, memory requirements, number of open files, ratio of average I/O to CPU bursts.
- Externally defined priority:
  - Type and amount of funds, department, politics.
- Problem:  starvation → low priority processes may never execute.
  - Solution: aging → as time progresses increase the priority of the process.
- Shortest Job First is a priority scheduling
  - priority is the predicted next CPU burst time.

# Example (non-premptive priority scheduling)

| Process | Burst time | Priority |
|---|---|---|
| 1 | 10 | 3 |
| 2 | 1 | 1 |
| 3 | 2 | 3 |
| 4 | 1 | 4 |
| 5 | 5 | 2 |

The Gantt chart for the schedule




Waiting time for $P_1 = 6$; $P_2 = 0$ $P_3 = 16$; $P_4 = 18$; $P_5 = 1$.

Average waiting time = $$\frac{6+0+16+18+1}{5} = \frac{41}{5} = 8.2$$

# Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (time quantum), usually 10 to 100 ms.
  - After this time has elapsed, the process is pre-empted and added to the end of the ready queue.
- If the ready queue has $n$ processes and the time quantum is $q$,
  - 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 unit.
- Performance of RR system depends on the size of $q$.
  - Large $q$ → FCFS.
  - Small $q$ → $q$ must be large with respect to context switch,
    - Otherwise overhead is too high.
    - In practice context switch is less than 10μs.
  - Turn around time improves if most processes finish their next CPU burst within $q$.
  - Rule of thumb: 80% of CPU burst is at most $q$.
- Typically, higher average turnaround than SRTF, but better response time.

# Example (time quantum=20)

| Process | Burst time |
|---|---|
| $P_1$ | 53 |
| $P_2$ | 17 |
| $P_3$ | 68 |
| $P_4$ | 24 |

The Gantt chart for the schedule is:

| $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_4$ | $P_1$ | $P_3$ | $P_3$ |
|---|---|---|---|---|---|---|---|---|---|

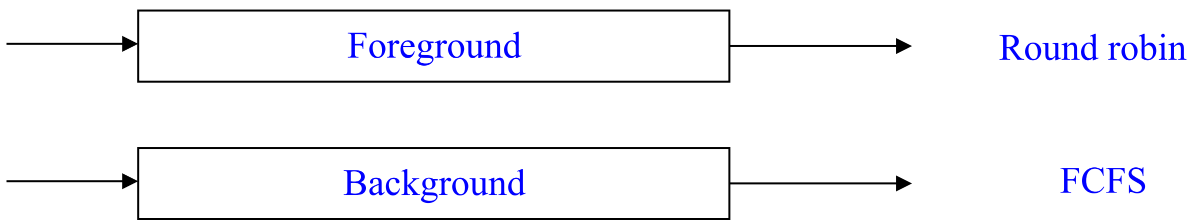
0 20 37 57 77 97 117 121 134 154 162

time

# Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues; e.g., foreground (interactive), background (batch).
- Each queue has its own scheduling algorithm.
- 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 and 20% to background in FCFS.

Example:

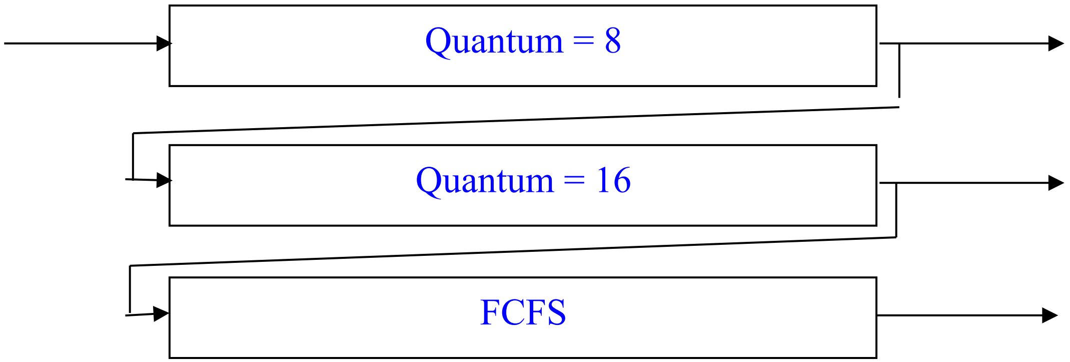


# Multilevel Feedback Queue Scheduling

- A process can move between the various queues
  - one form of aging mechanism to prevent starvation in multilevel queue scheduling.
- Multi-level feedback queue scheduler is defined by the following parameters:
  - Number of queues.
  - Scheduling algorithm for each Queue.
  - Method used to determine when to upgrade a process to higher Queue.
  - Method used to determine when to demote a process to lower Queue.
  - Method for determining which Queue a process will enter when that process needs service.

# Example

- Three queues, $Q_0$ with time quantum = 8 ms, $Q_1$ with time quantum = 16 ms, and $Q_2$ with FCFS.
- Scheduling
  - A new process enters $Q_0$ which is served FCFS. When it gains CPU, process receives 8 ms. If it does not finish in 8 ms, process is moved to $Q_1$.
  - At $Q_1$, process is again served FCFS and receives 16 additional ms. If it still does not complete, it is pre-empted and moved to $Q_2$.



# Guaranteed Scheduling

- Make real promises to the user about performance, and then live up to them.
- Example: for $n$ users, the promise can be $1/n$ CPU time for each user.
- System must keep track of how much CPU time a user has had for all his processes since login, and how long the user has logged in.

# Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available.
- *Homogeneous processors* within a multiprocessor → all processors are identical.
- *Load sharing*:
  - A queue for each processor → load *not* balanced.
  - A single ready queue for all processors; two approaches:
    - Each process is self-scheduling → need mutual exclusion on accessing the ready queue; symmetric multiprocessing.
    - Master-slave structure → one processor as the scheduler; asymmetric multiprocessing.
- *Asymmetric multiprocessing* – only one processor accesses the system data structures, alleviating the need for data sharing.
- Asymmetric multiprocessing is simpler.

# Real-time Scheduling

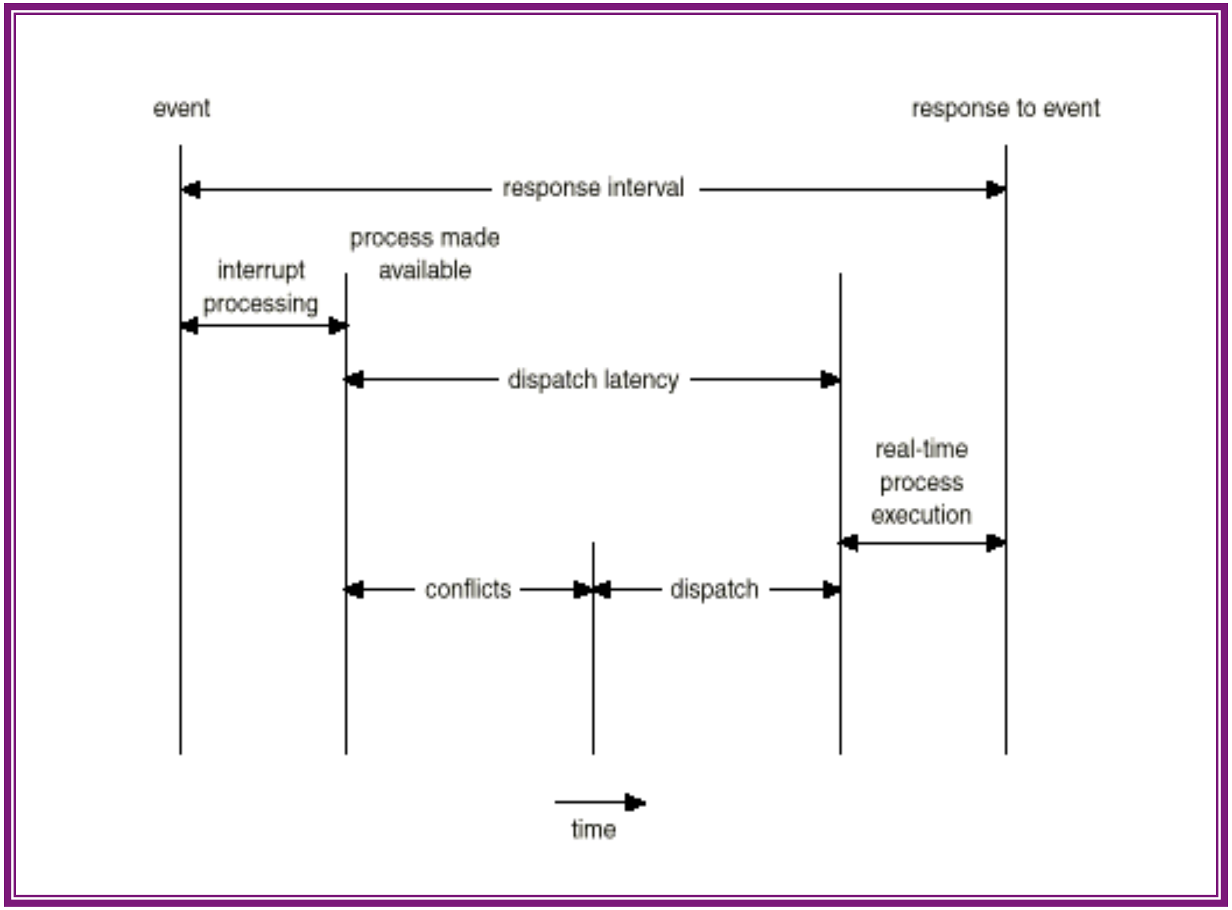
- *Hard real-time* systems require to complete a critical task within a guaranteed amount of time.
  - Each submitted process has a statement of the amount of time needed to complete or perform I/O.
  - The scheduler:
    - Admits the process if it can guarantee that the process will complete on time.
    - Rejects the process if impossible.
  - The scheduler should know how long each type of OS function takes to perform → impossible in a system with secondary storage or virtual memory.
  - Hard real-time systems are composed of special purpose software running on hardware dedicated to their critical process.

# Real-time Scheduling (cont.)

- *Soft real-time* computing requires critical processes to have priority over others.
  - General-purpose system supporting multimedia, high- speed interactive graphics, *etc*.
  - Requirements:
    - The system must have priority scheduling → real time processes always must have the highest priorities.
    - Dispatch latency must be small; ways to achieve this goal:
      - Allow system calls to be preemptible.
      - Make the entire kernel preemptible → Create priority inversion: the high priority process waits for lower priority one to finish.
      - Solution: use priority inheritance protocol where a low priority task that is using resources needed by the higher priority task inherits its priority until completing the resources.

# Real-time Scheduling (cont.)

- Two components of conflict phase of dispatch latency:
  1. Preemption of a kernel process.
  2. Release of low-priority process resources that are needed by a high-priority process



# Thread Scheduling

**User-level thread scheduling**

- Thread library schedules user level threads to run on an available kernel level thread
  - Known as process-contention scope (PCS):
  - for many-to-one and many-to-many models
- Priority scheduling is commonly used.
- The thread is not actually running in CPU yet.
  - Kernel does not know the existence of a user-level thread.
  - Context switch (kernel) occurs when the time quantum for the process is up.

**Kernel-level thread scheduling**

- kernel schedules which thread gets CPU
  - Known as system-contention scope (SCS)
  - For one-to-one model uses only SCS → Linux, Windows XP
  - The threads can be from the same or different processes.
- Each thread is given time quantum, and context switch is for each thread.
- Context switch in kernel thread is much more expensive than for user-level thread.
- Context switch between threads from the same process is faster than from different processes.

# Algorithm Evaluation

(a) **Deterministic model**

- Takes a particular predetermined workload and defines the performance of each algorithm for that workload.
- Simple, fast
- Requires exact number for input.
- Results are indicative only for this input.
- Too specific to be of general use.

| Job | Burst time |
|---|---|
| 1 | 10 |
| 2 | 29 |
| 3 | 3 |
| 4 | 7 |
| 5 | 12 |

# Deterministic model Example

FCFS:

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 10 | 39 | 42 | 49 | 61 |

SJF(NP):

| 3 | 4 | 1 | 5 | 2 |
|---|---|---|---|---|
| 3 | 10 | 20 | 32 | 61 |

RR(Q=10):

| 1 | 2 | 3 | 4 | 5 | 2 | 5 | 2 |
|---|---|---|---|---|---|---|---|
| 10 | 20 | 23 | 30 | 40 | 50 | 52 | 61 |

Waiting Time Table:

| Process | FCFS | SJF | RR |
|---|---|---|---|
| 1 | 0 | 10 | 0 |
| 2 | 10 | 32 | 32 |
| 3 | 39 | 0 | 20 |
| 4 | 42 | 3 | 23 |
| 5 | 49 | 20 | 40 |
| | 140 | 65 | 115 |

# Algorithm Evaluation (cont.)

**(b) Queueing model**

- Average queue length: $n$.
- Average waiting time in queue: $w$.
- Average arrival rate to the queue: $\lambda$.
- System steady state → the number of processes leaving the queue must be equal to the number of processes that arrive, thus:

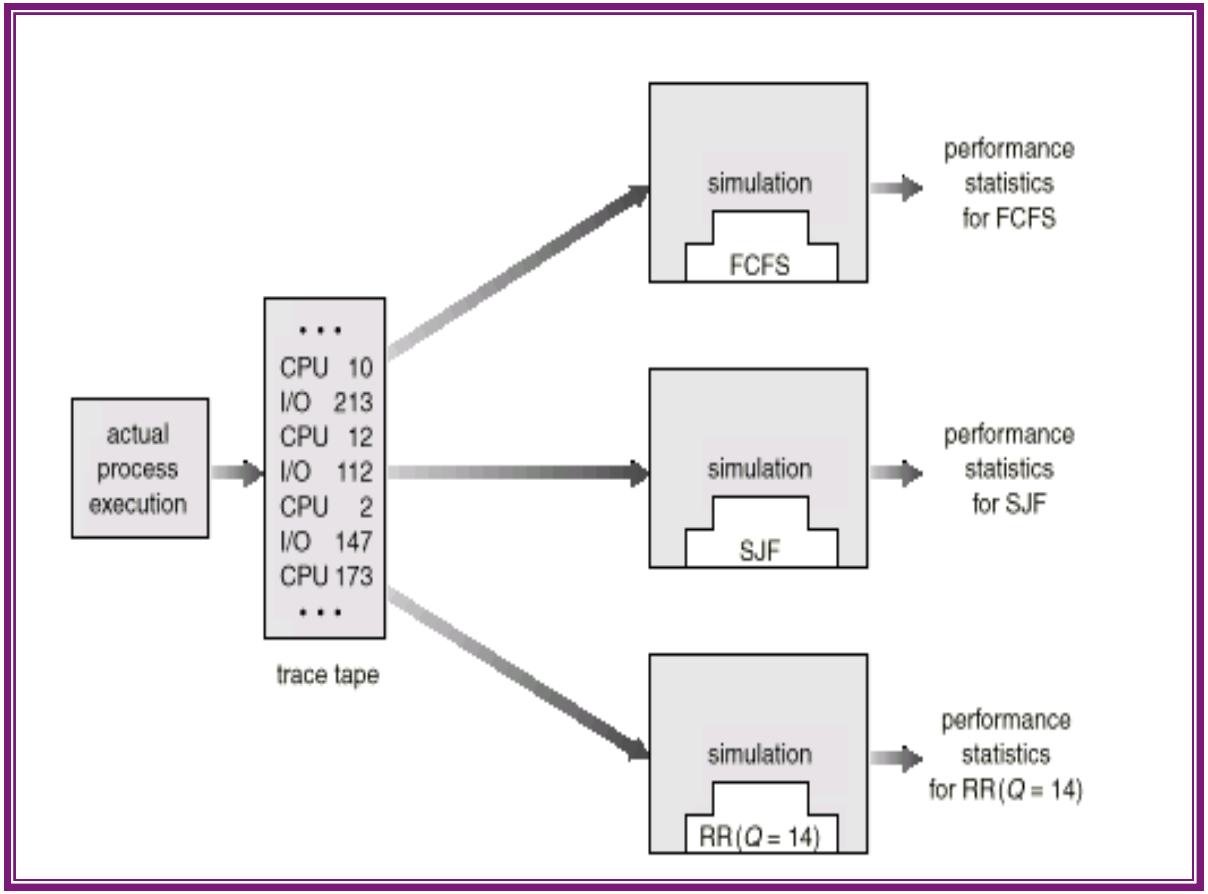
  Little's formula: $n = \lambda * w$
- Queueing analysis is useful:
  - In comparing algorithms.
  - Only as good as the distributions (often difficult and unrealistic to make the problem mathematically tractable).

# Algorithm Evaluation (cont.)

## (c) Simulations.

- Involve programming a model of the computer system.
- Software data structures represent major components.
- Variable representing the clock.
- Data can be generated by:
  - Random number generator.
  - Trace tapes: created by monitoring a real system.
  - Simulation can be very expensive:
    - Requiring hours of CPU time, large amount of storage (for trace tapes), and needs a lot time in designing, coding, and debugging the simulator.

# Simulations (cont)



# Algorithm Evaluation (cont.)

**(d) Implementation**.

- The only completely accurate way to evaluate a scheduling algorithm → code it, put it in the OS, see how it works.
- Difficulties:
  - The cost of this approach.
  - The environment in which the algorithm is used may change.

# An Example: Linux 2.2

- Provides two separate process-scheduling algorithms:
  - Timesharing: fair preemptive
  - Real time: absolute priorities
- Linux 2.2 allows only user-mode processes to be pre-empted
- Each process has a scheduling class → a prioritised, credit-based algorithm
  - The first is for timesharing processes
- Prioritised, credit-based
  - Each process has a certain number of scheduling credits → the largest means the highest priority.
  - The running process' credit is decremented by one each time a timer interrupt occurs → the process is suspended when its credit becomes 0
  - If there is no runnable processes, do re-crediting to every process in the system:

    $$\text{credits} = \text{credits}/2 + \text{priority}.$$

    - Requires $O(n)$ for this step, where $n$ is the total number of processes.
  - Give high priority to interactive or I/O-bound processes.
- Real-time scheduling: FCFS, RR

# Example $O(1)$ Scheduling

- Scheduling problem with Linux 2.2:
  - Not adequately support SMP systems.
  - Getting slower when the total number of tasks $n$ increases; $O(n)$
- Scheduling algorithm in Linux 2.5 is $O(1)$, and provides better support for SMS systems.
  - The algorithm runs in constant time, irrespective of the total number of processes $n$
- Preemptive and priority based
- Two separate priority range (lower value has higher priority):
  - For real time: 0 to 99
  - Nice: 100 to 140.
- Higher priority task is assigned with higher time quantum, *i.e.*,
  - A process with priority 0 → $q$ = 200ms.
  - A process with priority 140 → $q$ = 10ms.

# *O*(1) Scheduling (cont. )

- Kernel maintains a list of runnable tasks in a runqueue that contains two priority arrays

  **Active:** contains all tasks with time remaining in their time slices.

  **Expired:** contains all tasks with no remaining time in their time slices.

- Scheduler selects the highest priority task in Active array, indexed by priority.
- For SMP, each processor maintains its own runqueue and schedules itself independently
- Recalculate new priority of each process with exhausted time and move it to **Expired**
- Calculate its new priority:
  - Real time task is set with fix priority.
  - All other tasks have dynamic priorities (calculated before going to **Expired**):

    $$\textit{nice} \text{ value} + d; \text{ where } -5 \le d \le +5$$

  - The value for $d$ is determined by the task interactivity: How long the process has been sleeping waiting for I/O.
    - Longer sleep time → more interactive → $d$ is set close to -5.
  - CPU bound process → gets lower priority.
- When Active queue is empty → swap Active and Expired.

# Completely Fair Scheduler (CFS)

- $O(1)$ has poor response time for interactive processes
- Kernel release 2.6.23 makes CFS as the default Linux scheduler
- Scheduling in Linux is based on scheduling classes
  - Each class has a specific priority
  - The next task to run is the task with the highest priority in the highest priority class.
- Standard Linux implements two scheduling classes (A new scheduling class can be added
  - A default class using CFS scheduler
  - A real time scheduling class
- CFS assigns a proportion of CPU time to each task
  - The proportion is calculated based on the *nice* value (-20 to +19) assigned to each task
    - Lower value means higher priority → receives higher proportion of CPU time
    - The default *nice* value is 0

# CFS (cont.)

- CFS uses *targeted latency*: an interval of time during which every runnable task should run at least once.
  - Targeted latency has default and minimum values
  - Targeted latency increases when the total number of tasks in the system increases above a threshold value
  - Proportions of CPU time are allocated from the value of targeted latency
- CFS keeps virtual run time (vruntime – how long a task has run) for each task
- Vruntime is associated with decay factor based on priority of the task
  - Lower priority has higher decay factor
  - Normal task (*nice* value 0) has vruntime equal to physical runtime
    - A normal task that runs for for 200 milliseconds has vruntime = 200 milliseconds
    - A lower priority task that runs for 200 milliseconds has vruntime > 200 milliseconds
    - A higher priority task that runs for 200 milliseconds has vruntime < 200 milliseconds
  - CFS selects the task with the smallest vruntime to run next
    - A higher priority task can preempt a lower priority task