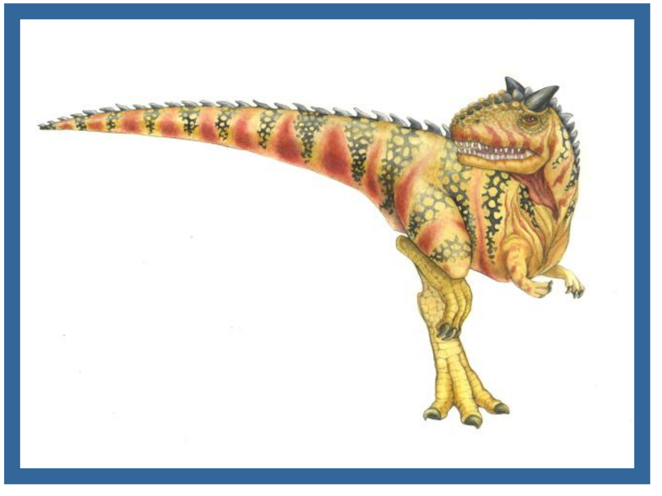
Chapter 6: CPU Scheduling



Chapter 6: CPU Scheduling

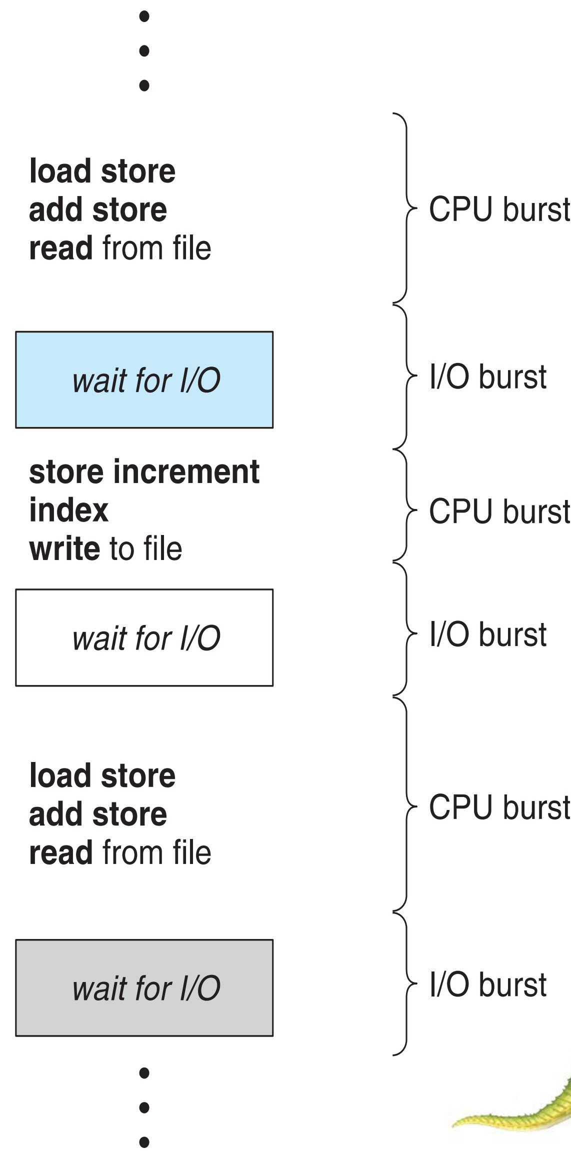
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

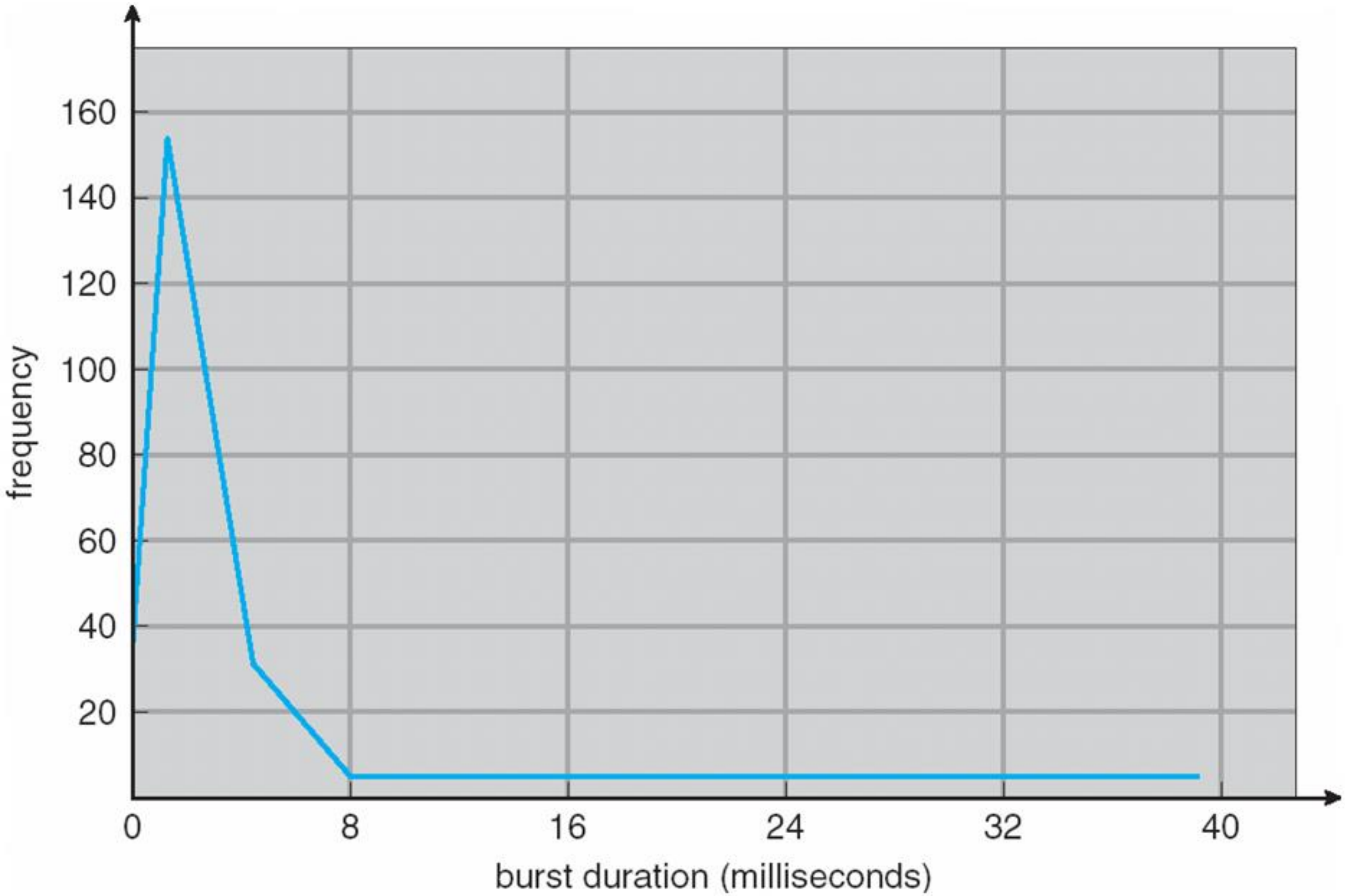
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern

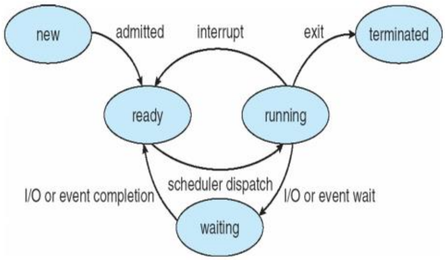


Histogram of CPU-burst Times



CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Ready queue may be ordered in various ways
- CPU scheduling decisions may 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



- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data, while in kernel mode, and interrupts occurring during critical OS activities

Dispatcher

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

Scheduling Criteria

- **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)

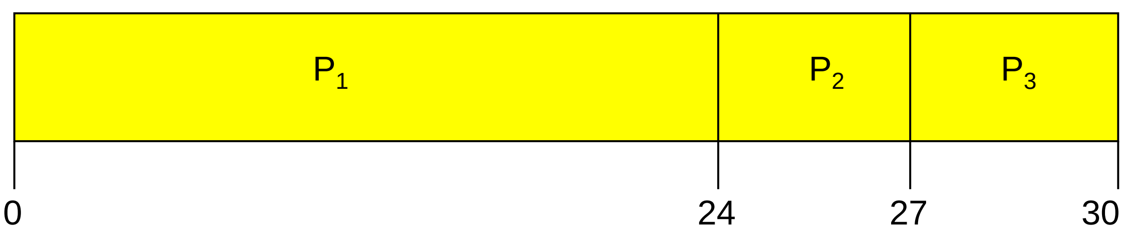
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

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:



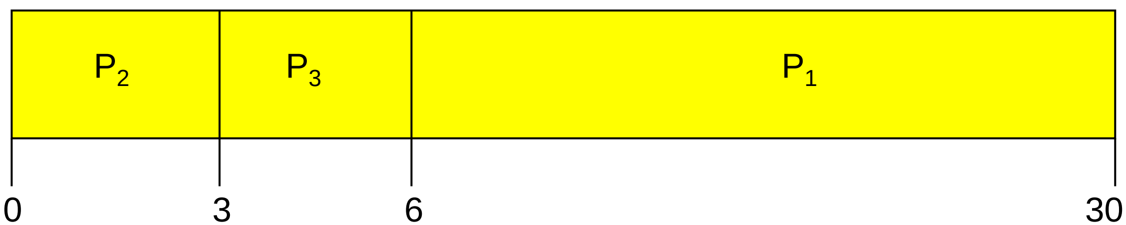
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

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
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

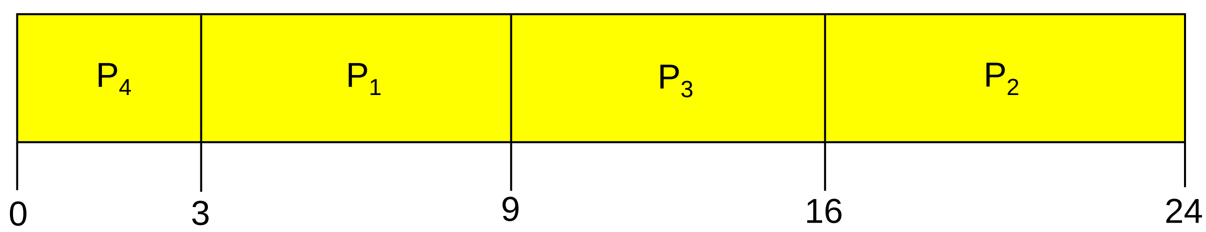
Shortest-Job-First (SJF) Scheduling

- 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
  - Could ask the user

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

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging

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

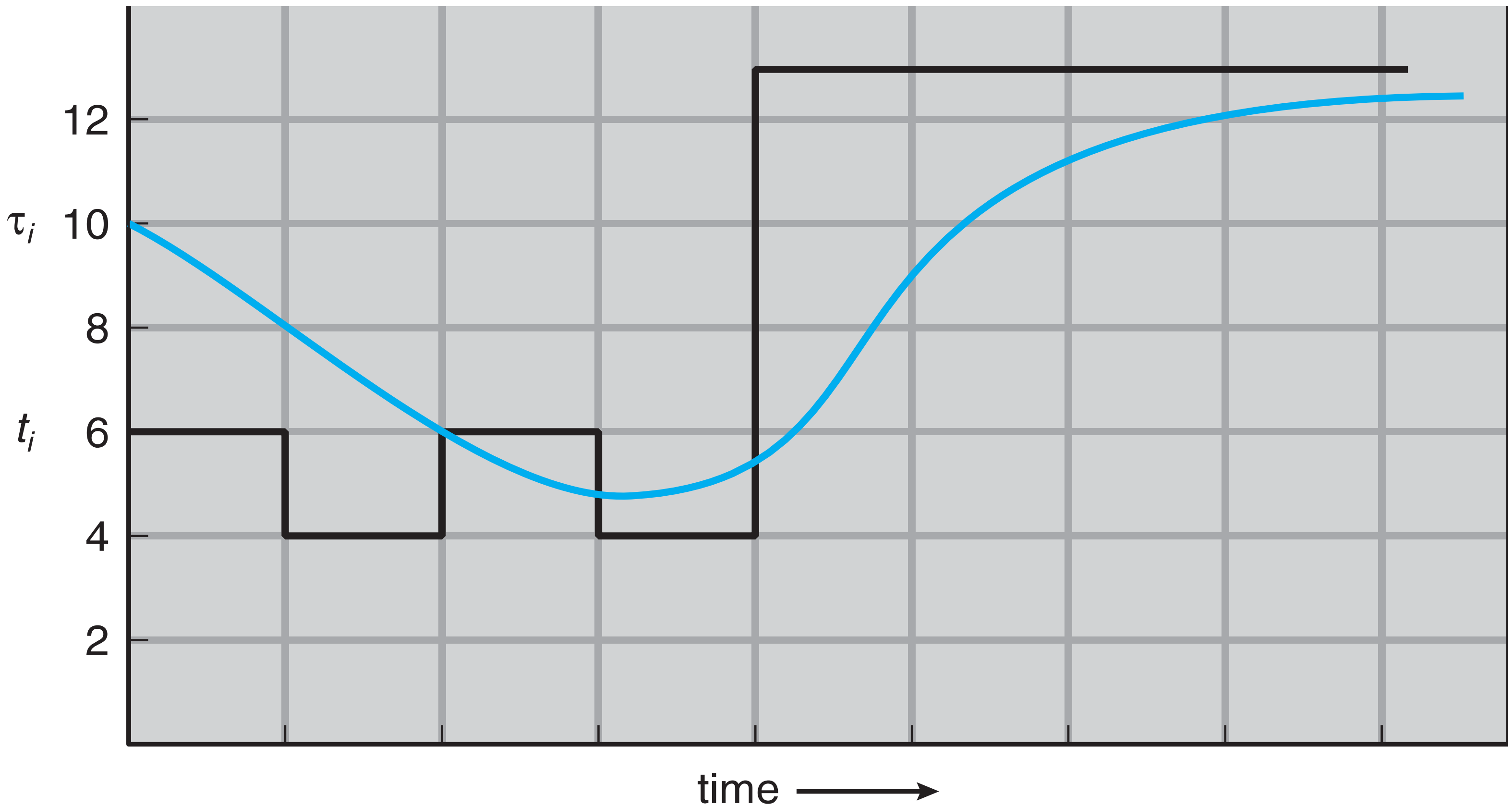
and τ_{n+1} = predicted value for the next CPU burst

Then for $\alpha, 0 \le \alpha \le 1$, define:

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

  - Commonly, α set to ½

Prediction of the Length of the Next CPU Burst



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

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

Shortest-Job-First (SJF) Scheduling

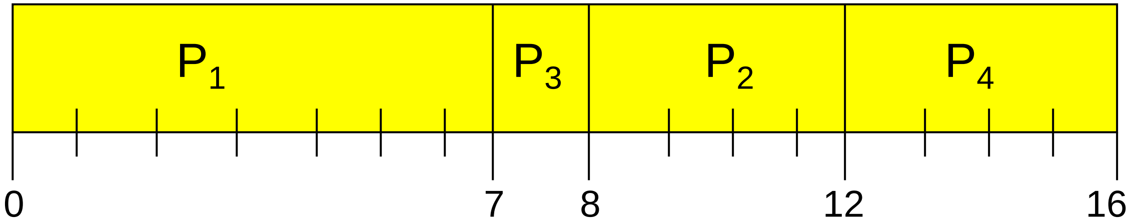
- Associate with each process the length of its next CPU burst and use these lengths to schedule the process with the shortest time.
- SJF is optimal – gives minimum average waiting time for a given set of processes.
- Two schemes:
  - **Nonpreemptive** – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - **Preemptive** – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the **Shortest-Remaining-Time-First** (SRTF).

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Now we add the concepts of varying arrival times

- Non-preemptive SJF Gantt Chart

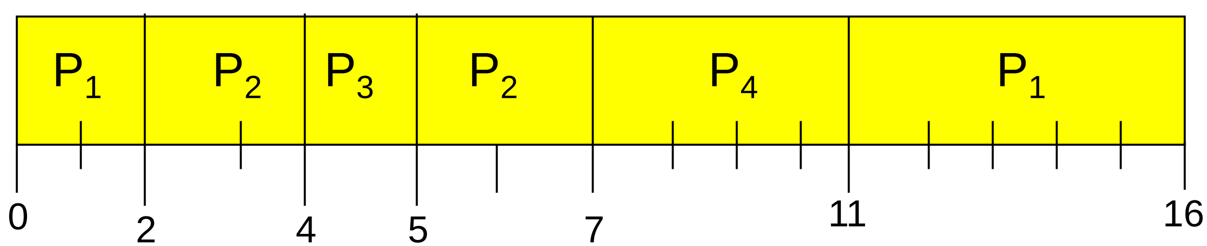


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF (SRTF)

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Preemptive SJF Gantt Chart



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Example of SRTF

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

- SRTF Gantt Chart

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

- Average waiting time = (9+0+15+2)/4 = 6.5

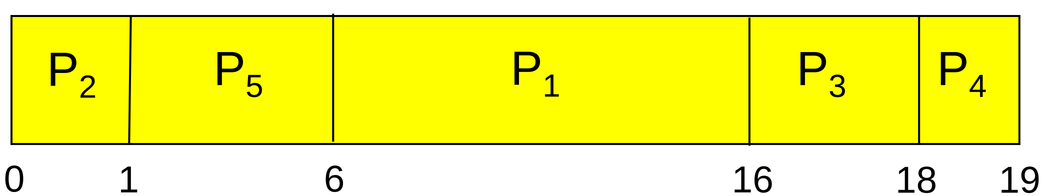
Priority Scheduling

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

Example of Priority Scheduling

Process	Burst Time	Priority
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 = (6+0+16+18+1)/5 = 8.2

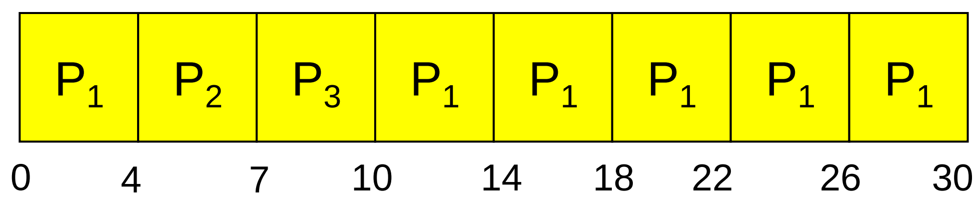
Round Robin (RR)

- 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 *q* to schedule next process
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high

Example of RR (Time Quantum = 4)

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Average waiting time = (6+4+7)/3 = 5.66
- Typically, higher average turnaround than SJF, but better ***response***

Example of RR (Time Quantum = 20)

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

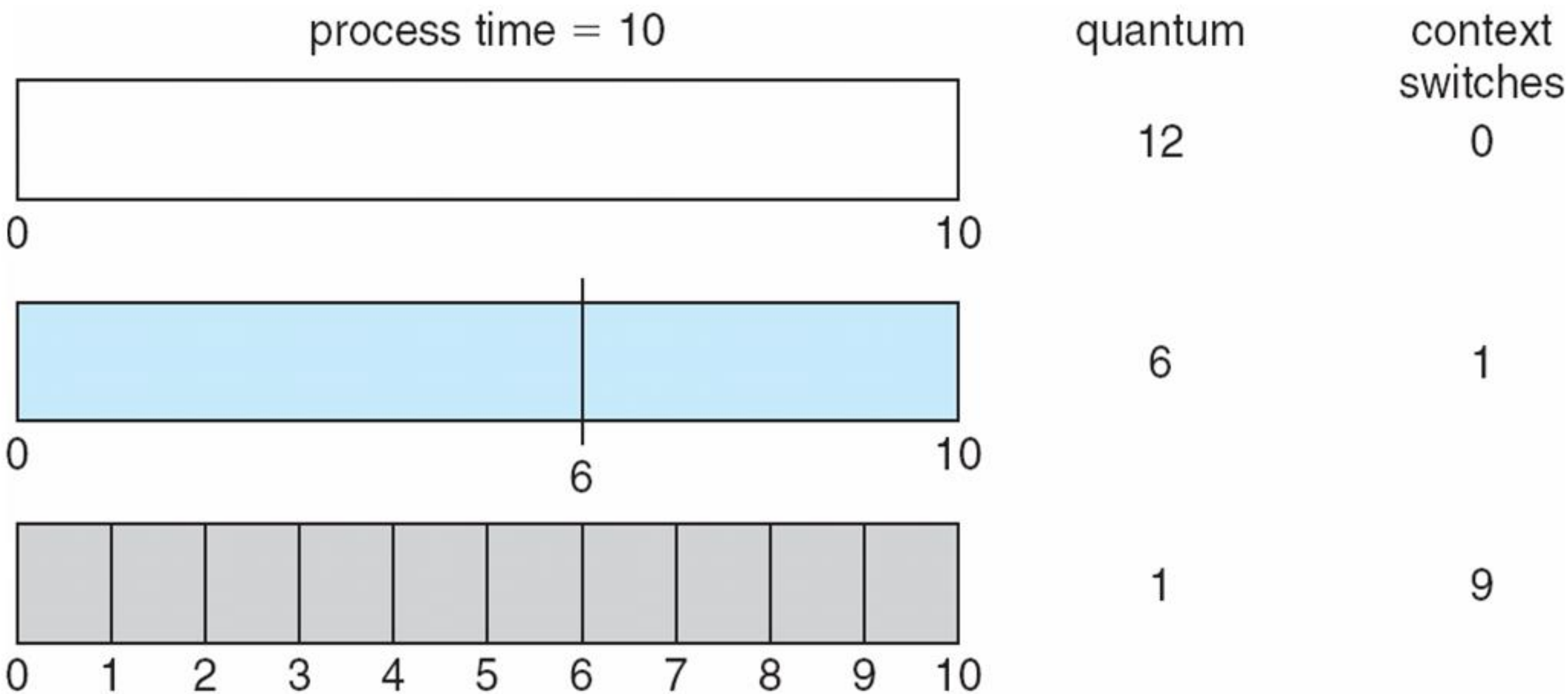
- The Gantt chart is:

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

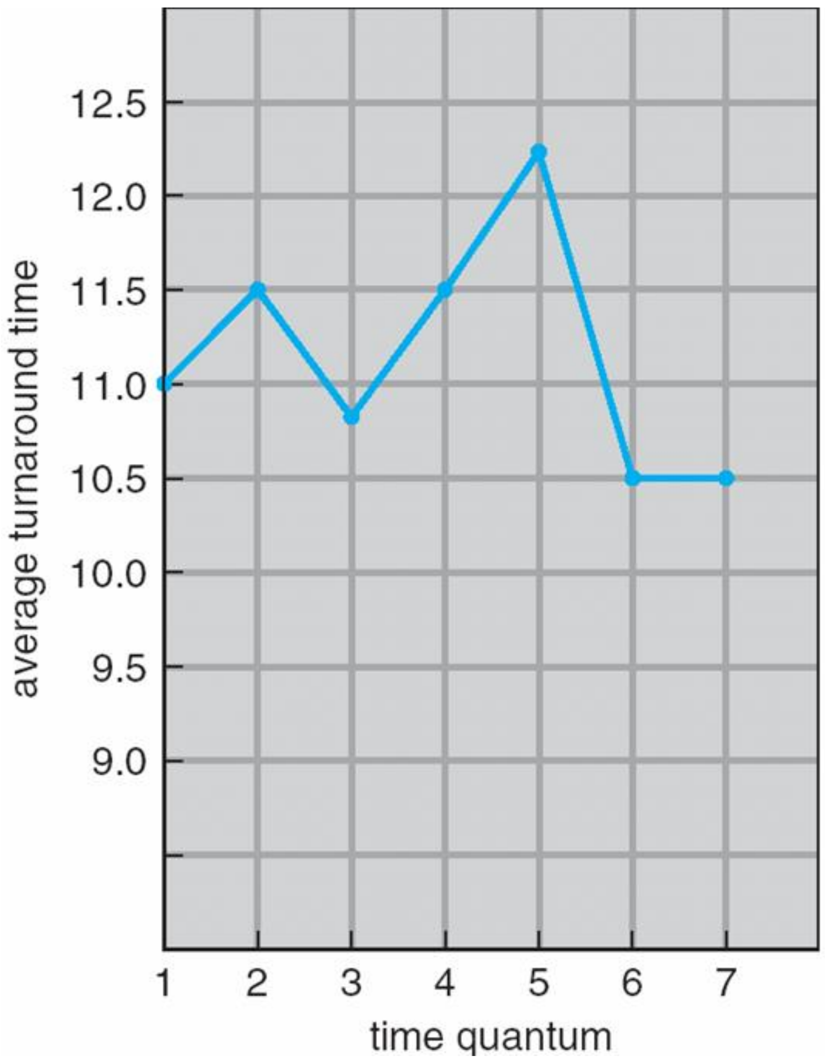
0	20	37	57	77	97	117	121	134	154	162

- Average waiting time = (81+20+86+97)/4 = 71

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum



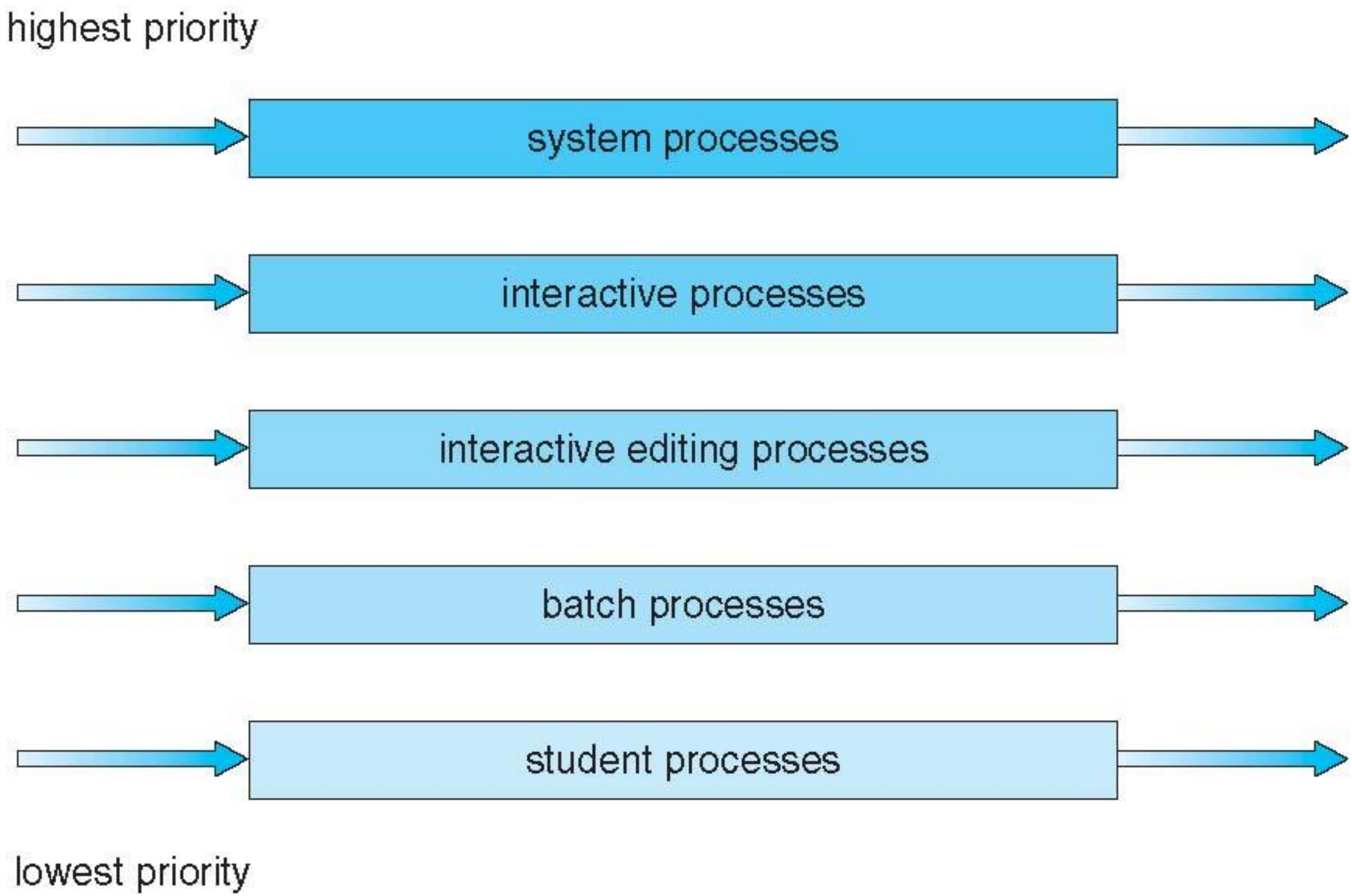
process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts should be shorter than q

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

Multilevel Queue Scheduling



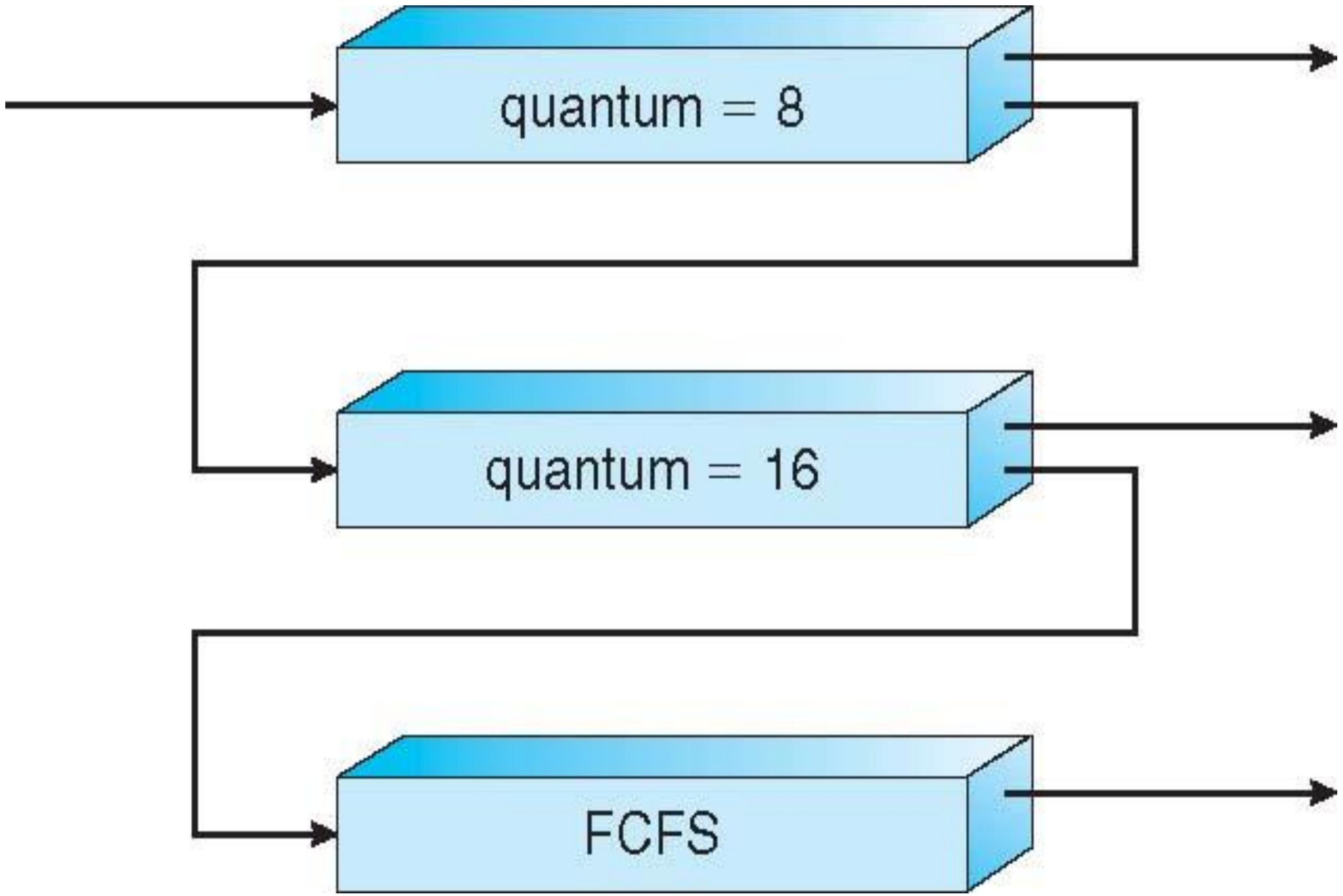
Multilevel Feedback Queue

- 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 upgrade 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 FCFS
    - 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 FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2

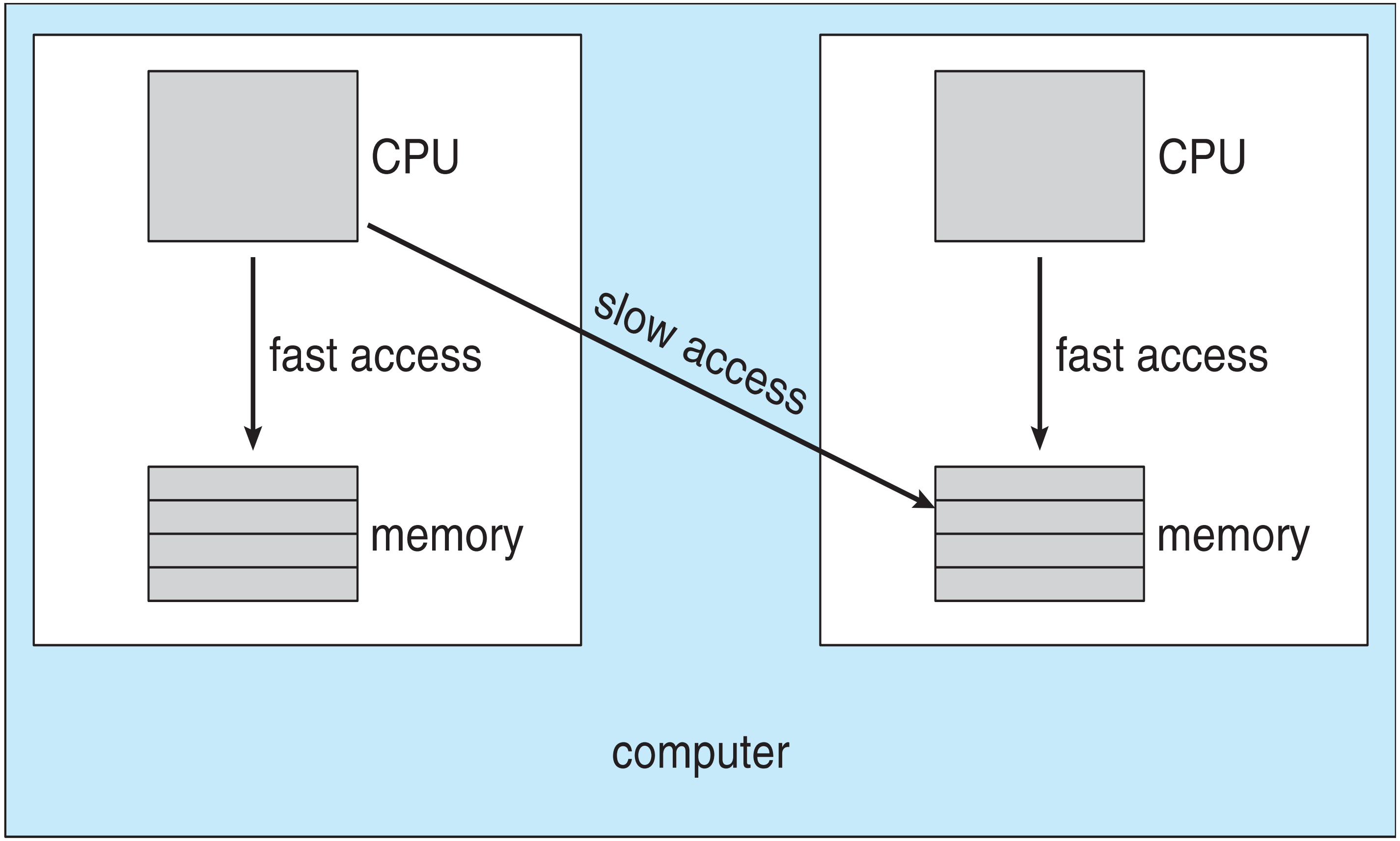
Example of Multilevel Feedback Queue



Multiple-Processor Scheduling

- With multiple CPUs, **load sharing** becomes possible but CPU scheduling more complex
  - **Homogeneous processors**
    - Can use any processor to run any process in the ready queue
  - **Asymmetric multiprocessing**
    - Only one master processor accesses system data structures and other processors execute only user code
  - **Symmetric multiprocessing (SMP)**
    - Each processor is self-scheduling, all processes in common ready queue, or each has its own private ready queue
    - Currently, most common
  - **Processor affinity** – process has affinity for processor on which it is currently running
    - **soft affinity** or **hard affinity**
    - Variations including **processor sets**

NUMA and CPU Scheduling



Note that memory-placement algorithms can also consider affinity

Multiple-Processor Scheduling – Load Balancing

- On SMP, need to keep workload balanced among all CPUs to fully utilize benefits of multiprocessors
- **Load balancing** attempts to keep workload evenly distributed across all processors
  - Necessary only when each processor has its own private queue of ready processes
- **Push migration** – a task periodically checks load on each processor, and if found imbalance pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

Algorithm Evaluation

- How to select CPU-scheduling algorithm for a particular OS?
  - there are many scheduling algorithms, each with its own parameters.
  - As a result, selecting an algorithm can be difficult.
- Determine criteria, then evaluate algorithms
- Various evaluation methods we can use:
  - **Deterministic modeling**
  - **Queueing models**
  - **Simulations**
  - **Implementation**

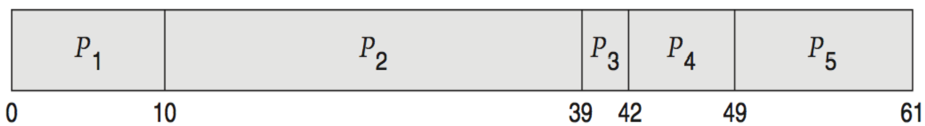
Deterministic Evaluation

- Type of **analytic evaluation**
- Takes a particular predetermined workload and defines the performance of each algorithm for that workload
  - Consider 5 processes arriving at time 0:

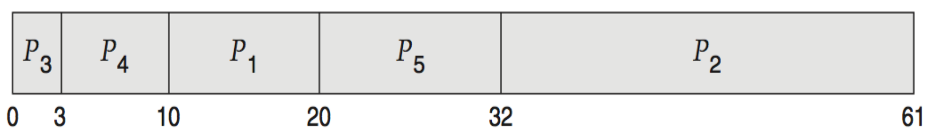
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

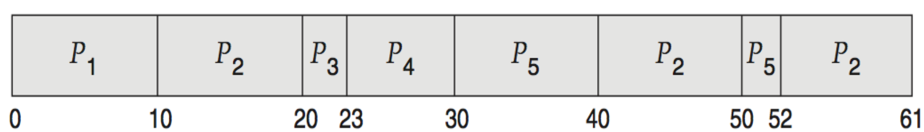
- For each algorithm, calculate minimum average waiting time
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



- Simple and fast, but requires exact numbers for input, and its answers apply only to those cases

Queueing Models

- Normally, there is no static set of processes to use for deterministic modeling
- However, the distribution of CPU and I/O bursts can be measured and then simply estimated
  - Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes verage throughput, utilization, waiting time
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc

Queueing Models

- Little's Formula:
  - n = average queue length
  - W = average waiting time in queue
  - λ = average arrival rate into queue
- Little's law – in steady state, processes leaving queue must equal processes arriving, thus:

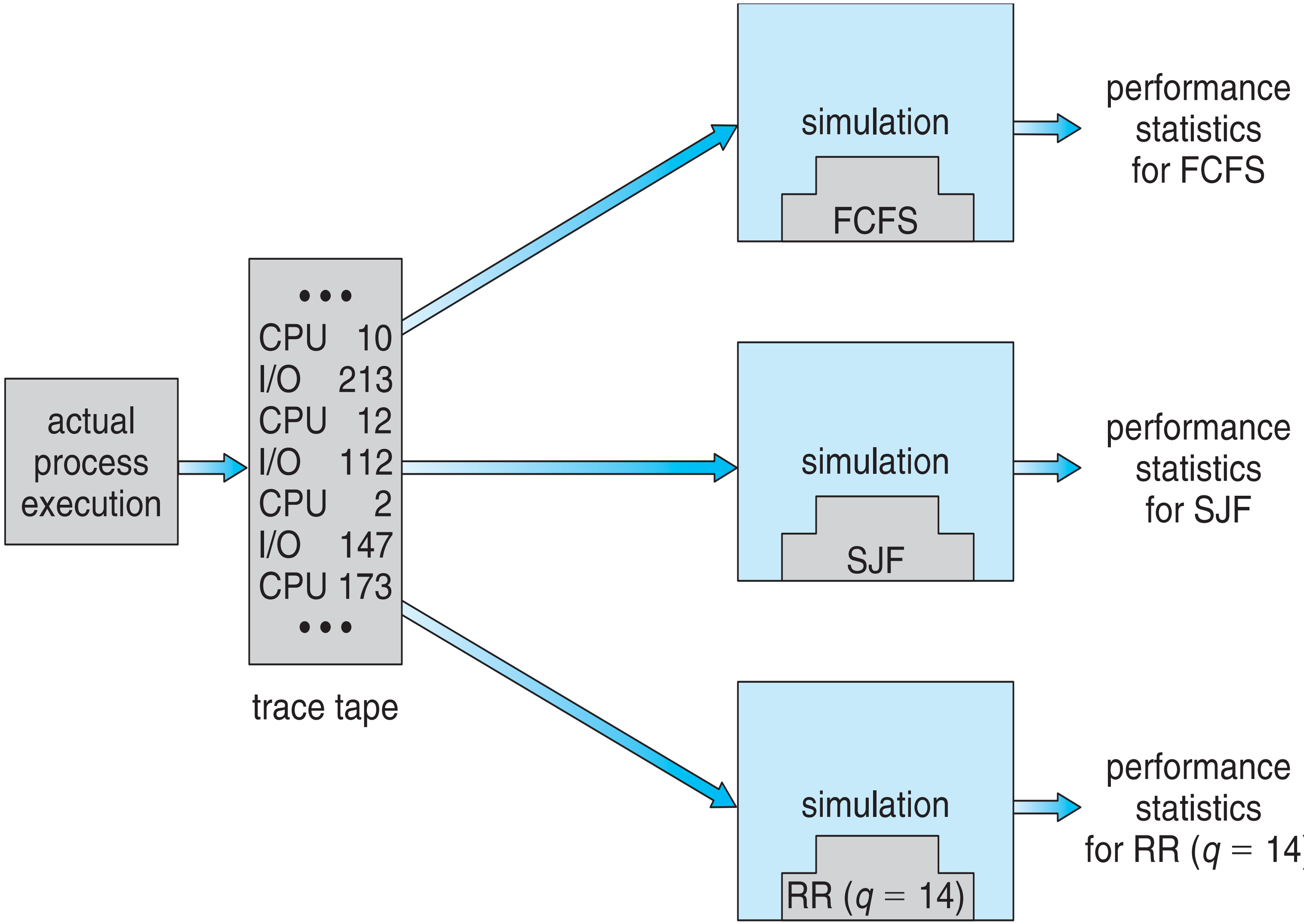
$$n = \lambda \times W$$

  - Valid for any scheduling algorithm and arrival distribution
  - For example, if on average 7 processes arrive per second, and normally 14 processes in queue, then average wait time per process = 2 seconds

Simulations

- Queueing analysis is useful but has limitations
  - Mathematics of complex algorithms and distributions can be difficult to work with
  - Thus, arrival and service distributions are defined in mathematically tractable ways
- Simulations more accurate
  - Programmed model of computer system
  - Data structures represent components of the system
    - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation



Implementation

- Simulation can be expensive and limited accuracy
  - A more detailed simulation provides more accurate results, but it also takes more computer time.
- Only completely accurate way to implement new scheduler and test in real systems
  - Cost of coding the algorithm and risk of users reaction
  - Changing the environment in which algorithm is used
- Most flexible scheduling algorithms that can be altered by system managers and tuned for a specific set of applications
  - Use APIs that can modify priority of a process or thread
  - But again environments vary

End of Chapter 6