

CPU Scheduling

Objectives

After completing this module, you will be able to understand the following:

- Scheduling Objectives
- Levels of Scheduling
- Scheduling Criteria
- Scheduling Algorithms
 - ✓ **FCFS,**
 - ✓ **Shortest Job First,**
 - ✓ **Priority,**
 - ✓ **Round Robin**

Scheduling Objectives

- Enforcement of fairness
 - in allocating resources to processes
- Enforcement of priorities
- Make best use of available system resources
- Give preference to processes holding key resources.
- Give preference to processes exhibiting good behavior.
- Degrade gracefully under heavy loads.

Levels of Scheduling

- High Level Scheduling or Job Scheduling
 - Selects jobs allowed to compete for CPU and other system resources.
- Intermediate Level Scheduling or Medium Term Scheduling
 - Selects which jobs to temporarily suspend/resume to smooth fluctuations in system load.
- Low Level (CPU) Scheduling or Dispatching
 - Selects the ready process that will be assigned the CPU.
 - Ready Queue contains PCBs of processes.

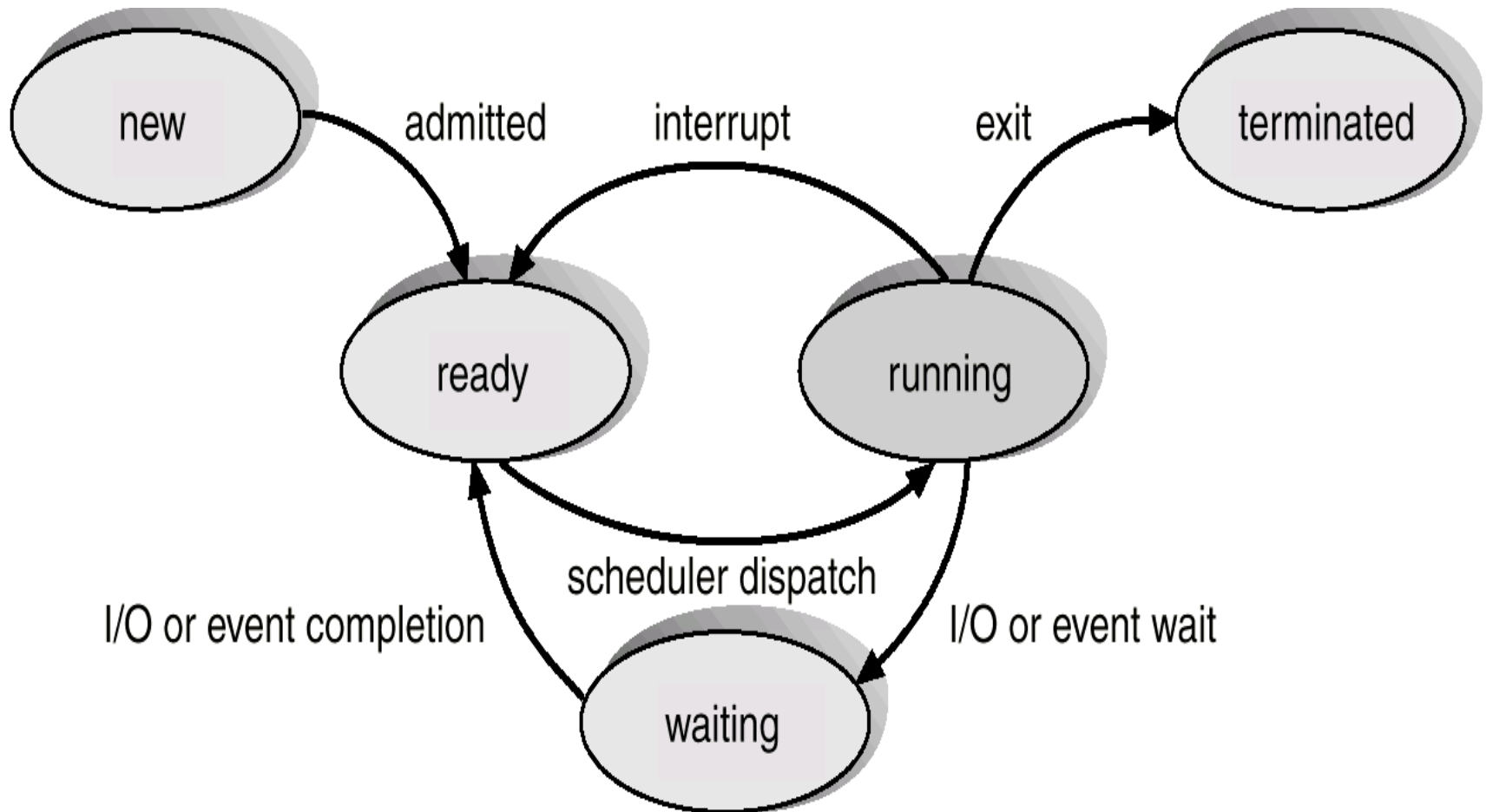
CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
 - **Non-preemptive Scheduling**
 - Once CPU has been allocated to a process, the process keeps the CPU until
 - Process exits OR
 - Process switches to waiting state
 - **Preemptive Scheduling**
 - Process can be interrupted and must release the CPU.
 - Need to coordinate access to shared data

CPU Scheduling Decisions

- CPU scheduling decisions may take place when a process:
 1. **switches from running state to waiting state**
 2. **switches from running state to ready state**
 3. **switches from waiting to ready**
 4. **Terminates**
- Scheduling under 1 and 4 is non-preemptive.
- All other scheduling is preemptive.

CPU scheduling decisions



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.
 - Dispatcher must be fast.

Scheduling Criteria

- CPU Utilization

- Keep the CPU and other resources as busy as possible

- Throughput

- # of processes that complete their execution per time unit.

- Turnaround time

- amount of time to execute a particular process from its entry time.

- Waiting time

- amount of time a process has been waiting in the ready queue.

- Response Time (in a time-sharing environment)

- amount of time it takes from when a request was submitted until the first response is produced, NOT output.

Optimization Criteria

- Maximize CPU Utilization
- Maximize Throughput
- Minimize Turnaround time
- Minimize Waiting time
- Minimize response time

Observations: Scheduling Criteria

- Throughput vs. response time
 - Throughput related to response time, but not identical:
 - Minimizing response time will lead to more context switching than if you only maximized throughput
 - Two parts to maximizing throughput
 - Minimize overhead (for example, context-switching)
 - Efficient use of resources (CPU, disk, memory, etc)
- Fairness vs. response time
 - Share CPU among users in some equitable way
 - Fairness is not minimizing average response time:
 - Better *average* response time by making system *less* fair

Scheduling Policies

- First-Come First-Serve (FCFS)
- Shortest Job First (SJF)
 - Non-preemptive
 - Pre-emptive
- Priority
- Round-Robin

First Come First Serve (FCFS) Scheduling

- Policy: Process that requests the CPU *FIRST* is allocated the CPU *FIRST*.
 - FCFS is a non-preemptive algorithm.
- Implementation - using FIFO queues
 - incoming process is added to the tail of the queue.
 - Process selected for execution is taken from head of queue.
- Performance metric - Average waiting time in queue.
- Gantt Charts are used to visualize schedules.

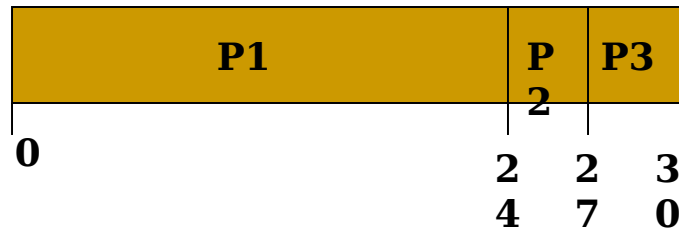
First-Come, First-Served(FCFS)

Scheduling

Example

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule



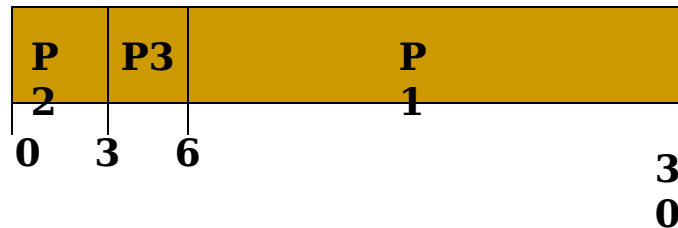
- Suppose the arrival order for the processes is
 - P1, P2, P3**
- Waiting time
 - P1 = 0;**
 - P2 = 24;**
 - P3 = 27;**
- Average waiting time
 - $(0+24+27)/3 = 17$**
- Average completion time
 - $(24+27+30)/3 = 27$**

FCFS Scheduling (cont.)

■ Example

Process	Burst Time
P 1	24
P 2	3
P 3	3

Gantt Chart for Schedule



- Suppose the arrival order for the processes is
 - **P2, P3, P1**
- Waiting time
 - **P1 = 6; P2 = 0; P3 = 3;**
- Average waiting time
 - **$(6+0+3)/3 = 3$, better..**
- Average completion time
 - **$(3+6+30)/3 = 13$, better..**
- *Convoy Effect:*
 - **short process behind long process, e.g. 1 CPU bound process, many I/O bound processes.**

Shortest-Job-First(SJF)

Scheduling

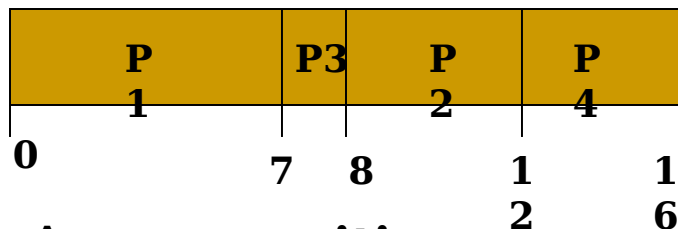
- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time.
- Two Schemes:
 - Scheme 1: Non-preemptive
 - Once CPU is given to the process it cannot be preempted until it completes its CPU burst.
 - Scheme 2: Preemptive
 - If a new CPU process arrives with CPU burst length less than remaining time of current executing process, preempt. **Also called Shortest-Remaining-Time-First (SRTF)..**

SJF and SRTF (Example)

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Non-Preemptive SJF Scheduling

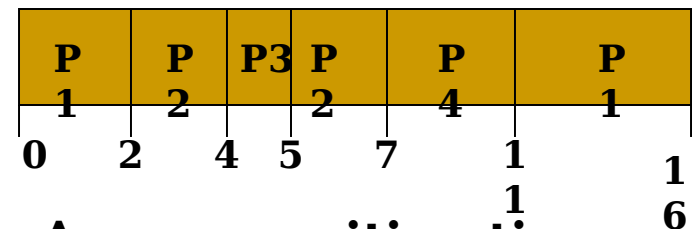
Gantt Chart for Schedule



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

Preemptive SJF Scheduling

Gantt Chart for Schedule



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

SJF/SRTF Discussion

- SJF/SRTF are the best you can do at minimizing average response time
 - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
 - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
 - What if all jobs the same length?
 - SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
 - What if jobs have varying length?
 - SRTF (and RR): short jobs not stuck behind long ones
- Starvation
 - SRTF can lead to starvation if many small jobs!
 - Large jobs never get to run

SRTF Further discussion

- Somehow need to predict future
 - How can we do this?
 - Some systems ask the user
 - When you submit a job, have to say how long it will take
 - To stop cheating, system kills job if takes too long
 - But: Even non-malicious users have trouble predicting runtime of their jobs
- Bottom line, can't really know how long job will take
 - However, can use SRTF as a yardstick for measuring other policies
 - Optimal, so can't do any better
- SRTF Pros & Cons
 - Optimal (average response time) (+)
 - Hard to predict future (-)
 - Unfair (-)



Determining Length of Next CPU Burst

- One can only estimate the length of burst.
- Use the length of previous CPU bursts and perform exponential averaging.
 - t_n = actual length of nth burst
 - τ_{n+1} = predicted value for the next CPU burst
 - $\alpha = 0, 0 \leq \alpha \leq 1$
 - Define
 - $\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$

Exponential Averaging(cont.)

- $\alpha = 0$

- $\tau_{n+1} = \tau_n$; Recent history does not count

- $\alpha = 1$

- $\tau_{n+1} = t_n$; Only the actual last CPU burst counts.

- Similarly, expanding the formula:

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

- Each successive term has less weight than its predecessor.

Priority Scheduling

- A priority value (integer) is associated with each process.
- CPU is allocated to process with the highest priority.
 - Preemptive
 - Nonpreemptive

Priority Scheduling

- As an example, consider the following set of processes, assumed to have arrived at time 0 in the order $P_1, P_2, \dots, P_5$, with the length of the CPU burst given in milliseconds:

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

- Using priority scheduling, we would schedule these processes according to the following Gantt chart:



- The average waiting time is 8.2 milliseconds.

Round Robin (RR)

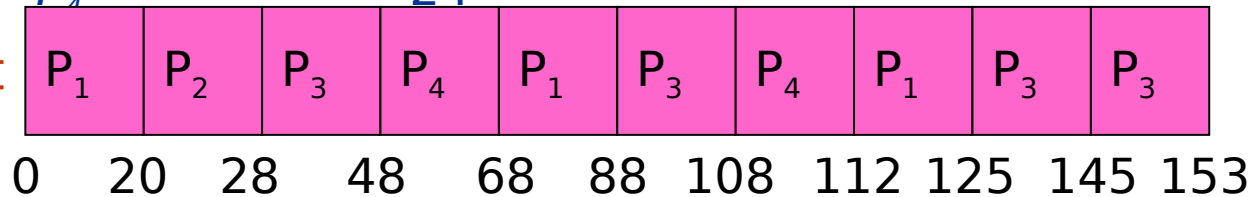
- Each process gets a small unit of CPU time
 - Time quantum usually 10-100 milliseconds.
 - After this time has elapsed, the process is preempted and added to the end of the ready queue.
- n processes, time quantum = q
 - Each process gets $1/n$ CPU time in chunks of at most q time units at a time.
 - No process waits more than $(n-1)q$ time units.
 - Performance
 - Time slice q too large – response time poor
 - Time slice $(\infty)?$ -- reduces to FIFO behavior
 - Time slice q too small - Overhead of context switch is too expensive. Throughput poor

Example of RR with Time Quantum = 20

Example:

<u>Process</u>	<u>Burst Time</u>
P_1	53
P_2	8
P_3	68
P_4	24

□ The Gantt chart



□ Waiting time

- $P_1 = (68 - 20) + (112 - 88) = 72$
- $P_2 = (20 - 0) = 20$
- $P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$
- $P_4 = (48 - 0) + (108 - 68) = 88$

□ Average waiting time = $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$

□ Average completion time = $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$

■ Thus, Round-Robin Pros and Cons:

- Better for short jobs, Fair (+)
- Context-switching time adds up for long jobs (-)

Round Robin (Effect of time quantum tq , with $tq = \min.$, and $tq = \max.$)

- ✓ The performance of the RR algorithm depends heavily on the size of the time quantum.
- ✓ If the time quantum is extremely large, the RR policy is the same as the FCFS policy.
- ✓ If the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches.
- ✓ If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
- ✓ If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.
- ✓ If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly

