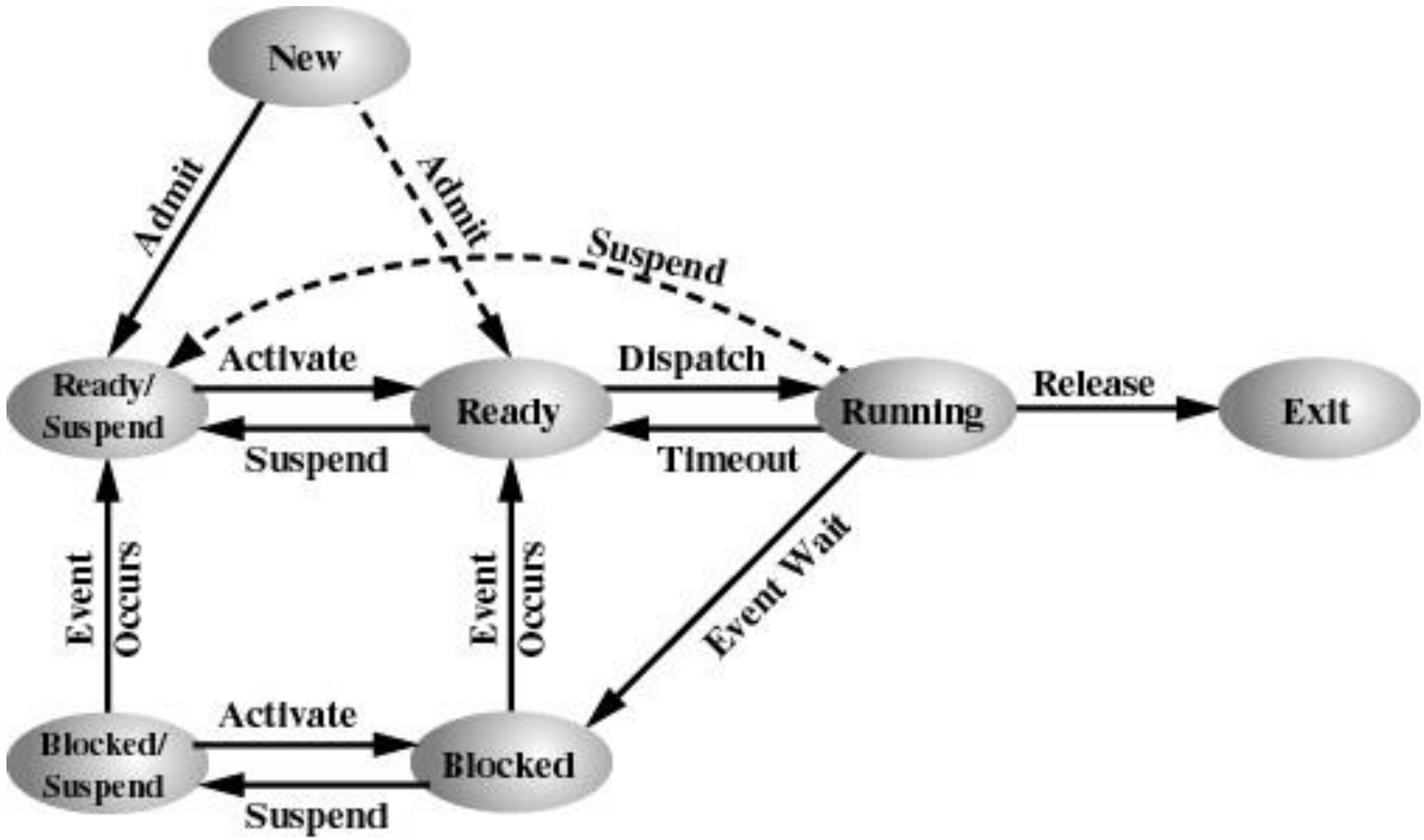
PROCESS SCHEDULING

Scheduling

- Processes in different state maintain Queues.
- The different queues are maintained for different purpose eg.
  - Ready Queue : Processes waiting for CPU
  - Blocked : processes waiting for I/O to complete
- Transition from a state where queue is maintained to next state involves decision making such as
  - When to move process from one state to another
  - Which process to move
- When transitions occur, OS may be required to carry out some house keeping activity such as context switch, Mode switch etc. These activities are considered as overhead and must be carried out in efficient manner

- Scheduling is matter of managing queues to minimize queuing delay and to optimize performance in queuing environment
- Scheduling affects the performance of the system because it determines which process will wait and which will progress

7 States



(b) With Two Suspend States

Types of Scheduling

Long-term scheduling	The decision to add to the pool of processes to be executed
Medium-term scheduling	The decision to add to the number of processes that are partially or fully in main memory
Short-term scheduling	The decision as to which available process will be executed by the processor
I/O scheduling	The decision as to which process's pending I/O request shall be handled by an available I/O device

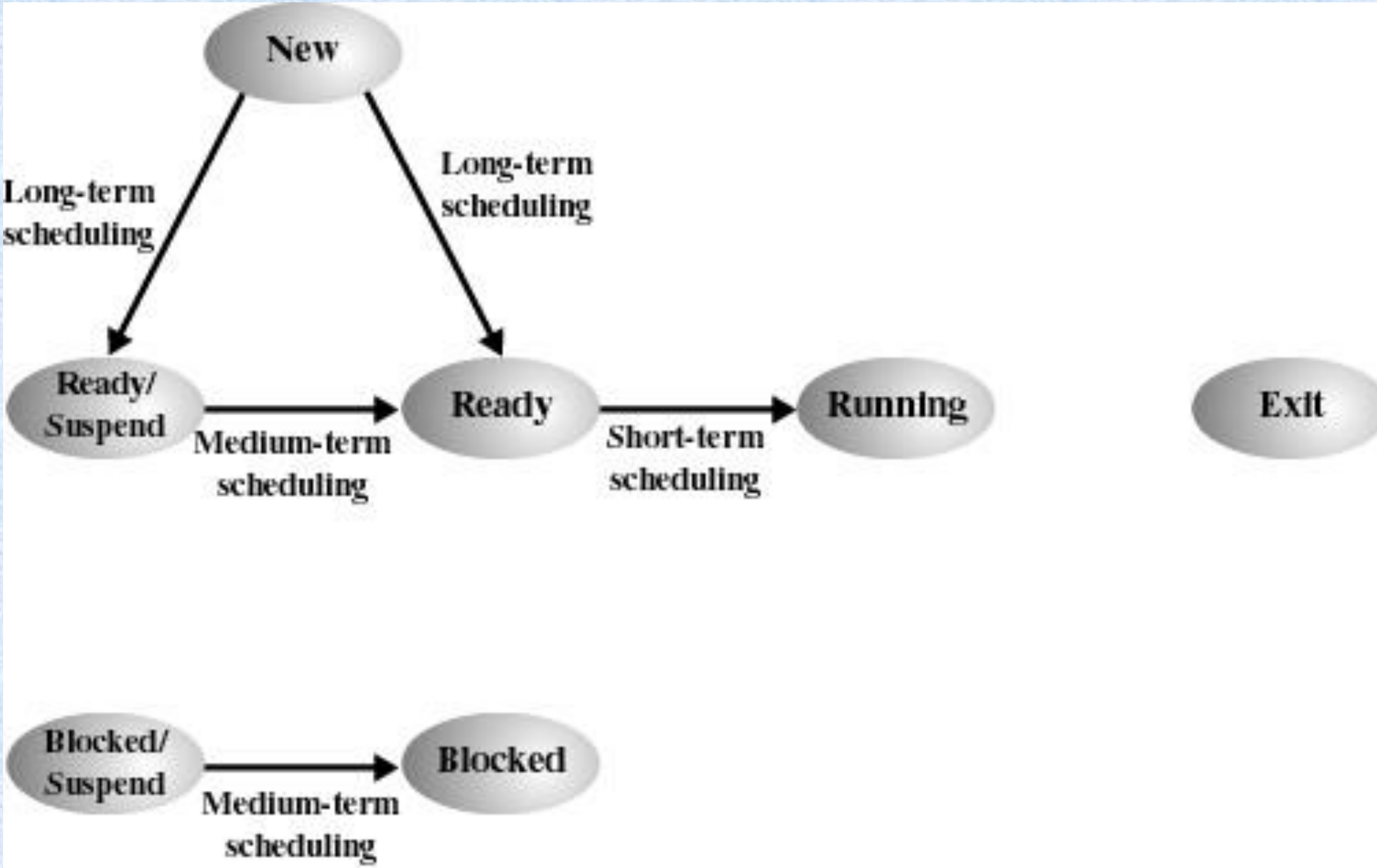


Figure 9.1 Scheduling and Process State Transitions

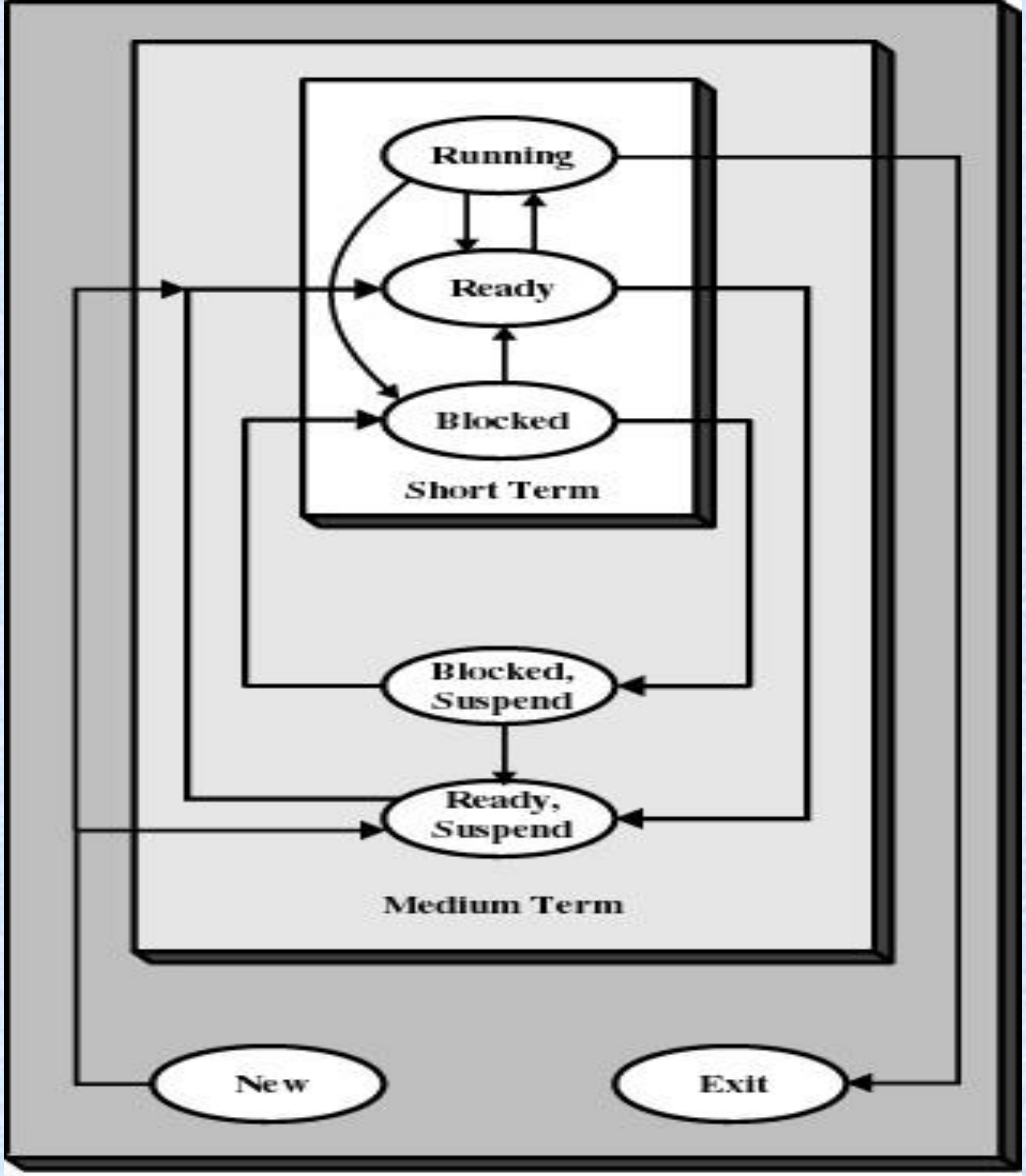


Figure 9.2 Levels of Scheduling

Long-term Scheduling

- Long-term scheduler is invoked very infrequently (seconds, minutes) $\Rightarrow$ **(may be slow).** Invoked to move a new process to ready or ready suspend queue
- Determines which programs are admitted to the system for processing
- Controls the degree of multiprogramming

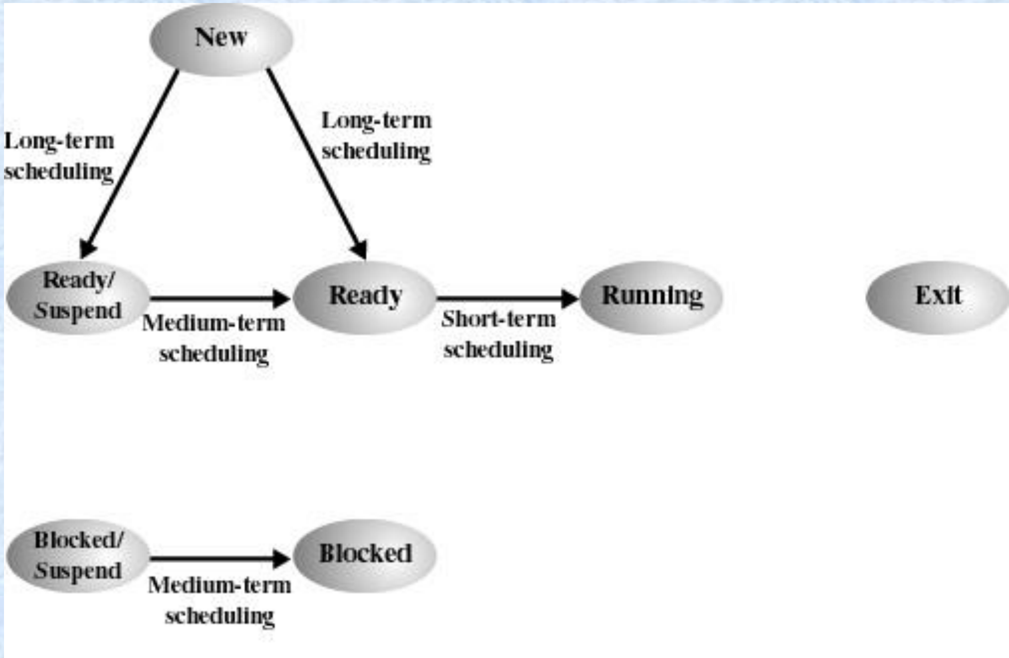


Figure 9.1 Scheduling and Process State Transitions

Long-term Scheduling

- More processes=> smaller percentage of time each process is executed
- Processes can be described as either:
  - I/O-*bound process* – spends more time doing I/O than computations; very short CPU bursts.
  - CPU-*bound process* – spends more time doing computations; very long CPU bursts.

The long term scheduler must select a good process mix of I/O bound and CPU bound processes

Medium-term Scheduling

- Part of the swapping function
- Based on the need to manage the degree of multiprogramming and provide a good process mix
- Done to free up some memory when required
- Invoked to move a process from ready suspend to ready or block to block suspend and vice-versa

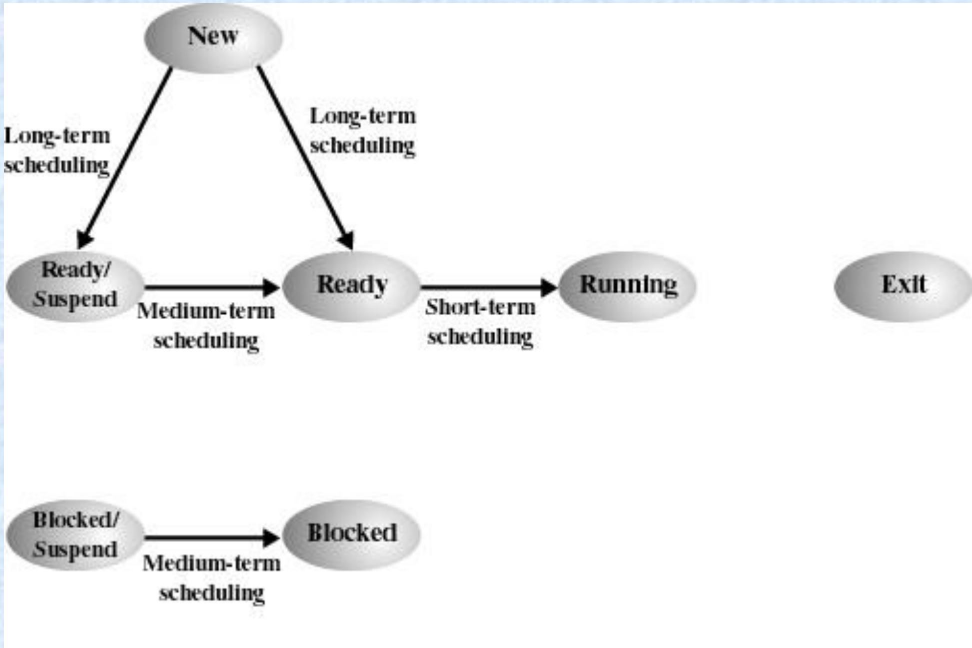


Figure 9.1 Scheduling and Process State Transitions

CPU(short Term) Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
- 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.*

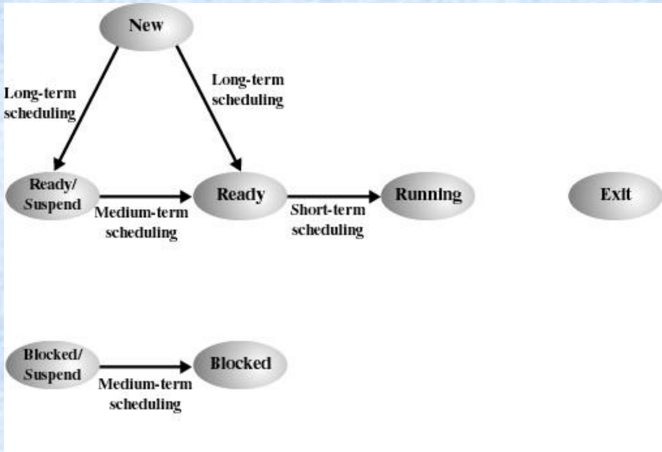


Figure 9.1 Scheduling and Process State Transitions

Decision Mode

- Nonpreemptive
  - Once a process is in the running state, it will continue until it terminates or blocks itself for I/O.
- Preemptive
  - Currently running process may be interrupted and moved to the Ready state by the operating system
  - Allows for better service since any one process cannot monopolize the processor for very long.Unix uses this strategy but the Kernel is nonpreemptive.

Short-term Scheduling

- Short-term scheduler is invoked very frequently (once in every 100 milliseconds or so)
  $\Rightarrow$ **must be fast**.
- Includes the dispatcher module
- Executes most frequently
- Invoked when an event occurs
  - Clock interrupts
  - I/O interrupts
  - Operating system calls

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.

Short-term Scheduling Criteria

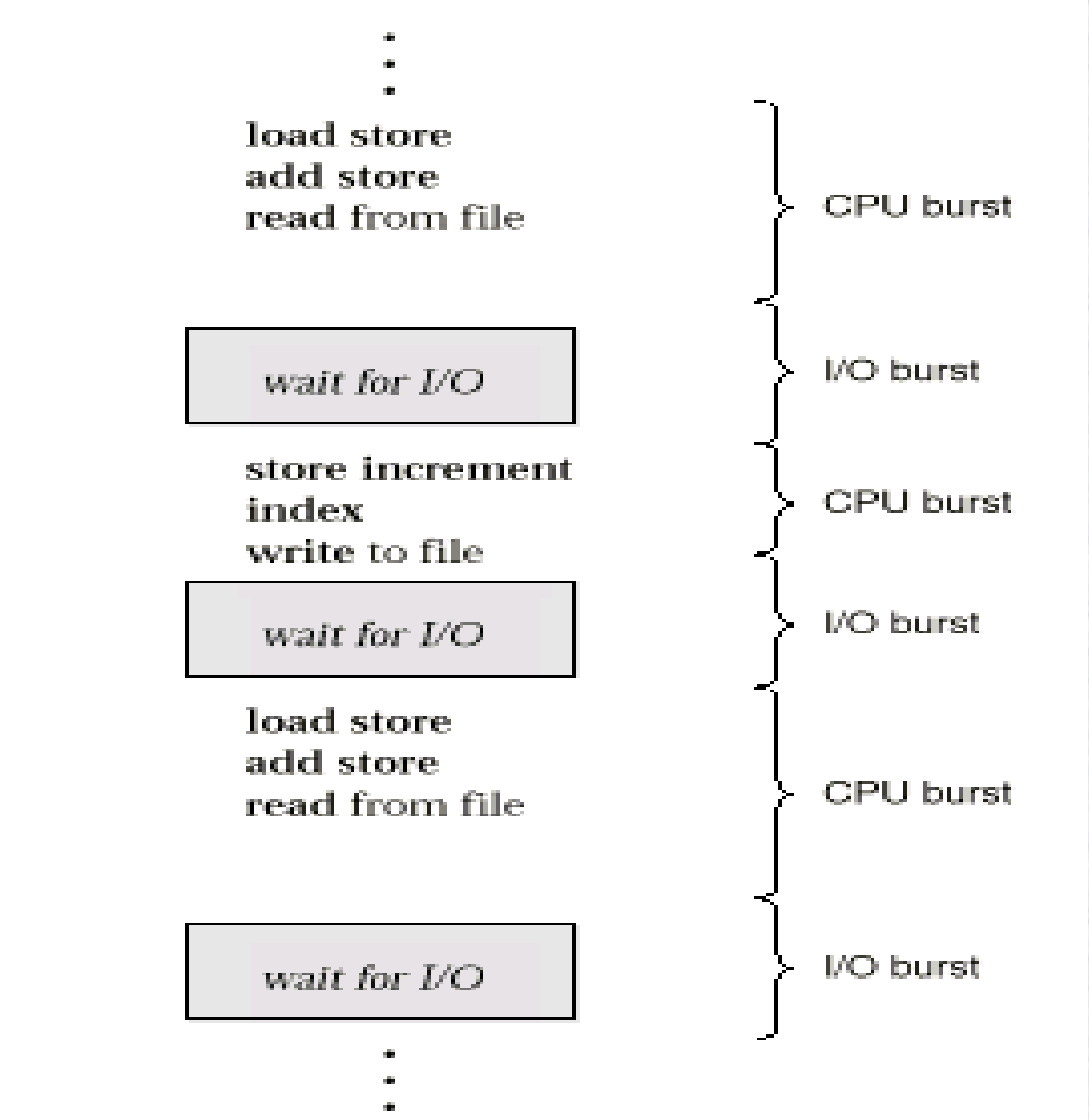
- User-oriented
  - Response Time
    - *Elapsed time between the submission of a request until there is output(or the CPU is allocated).*
  - Predictability
- System-oriented
  - Effective and efficient utilization of the processor
    - *throughput*

Uniprocessor Scheduling

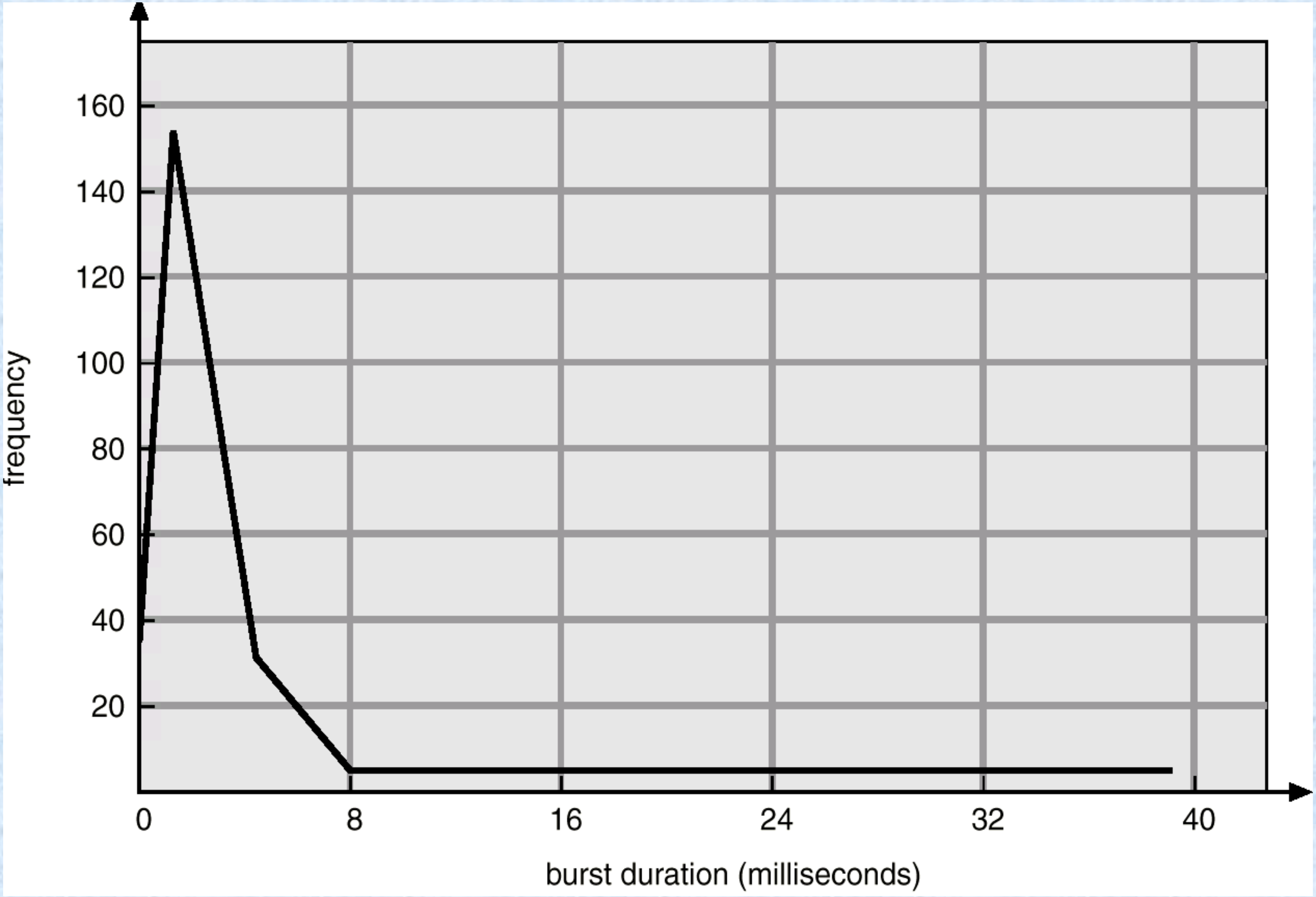
Basic Concepts

- Maximum CPU Utilization Obtained With Multiprogramming
- Process Execution Consists of a *Cycle* of CPU Execution and I/O Wait. => CPU–I/O Burst Cycle
- CPU Burst Distribution – Processes Mostly Have Short CPU Bursts (~5ms)

Alternating Sequence of CPU And I/O Bursts



Histogram of CPU-burst Times



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 taken to complete a particular process from its submission to its termination. Also known as residence time T_r.
  - The actual CPU time taken by a process is called Service time, T_s. The Normalized turnaround time is then defined as T_r/T_s. Minimum possible value is 1, which means immediate service.
- Priorities- can be fixed in many ways

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** output (for time-sharing environment)
- Predictability – keep the variance of response time low under different load conditions
- Fairness- Provide equal share of CPU time to all workgroups

Optimization Criteria

In short:

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

First-come, First-served (FCFS) Scheduling

- Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

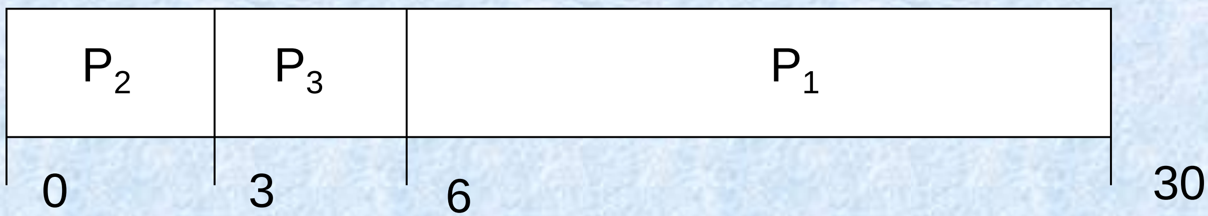
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:

- 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 => FCFS penalizes short processes

First-come-first-served (FCFS)

- A short process may have to wait a very long time before it can execute
- Favors CPU-bound processes
  - I/O processes have to wait until CPU-bound process completes(FCFS is non preemptive)

Shortest-job-first (SJF) Scheduling

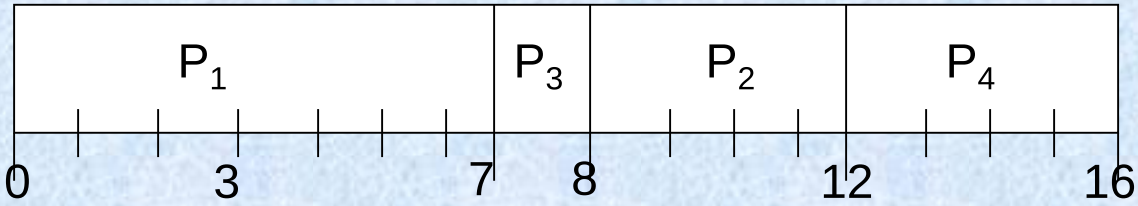
- Associated with each process is the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - Nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.

    Also known as Shortest Process Next (SPN)
  - Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, then preempt. This scheme is known as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Example of Non-preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)



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

Shortest Process Next (SJF/SPN)

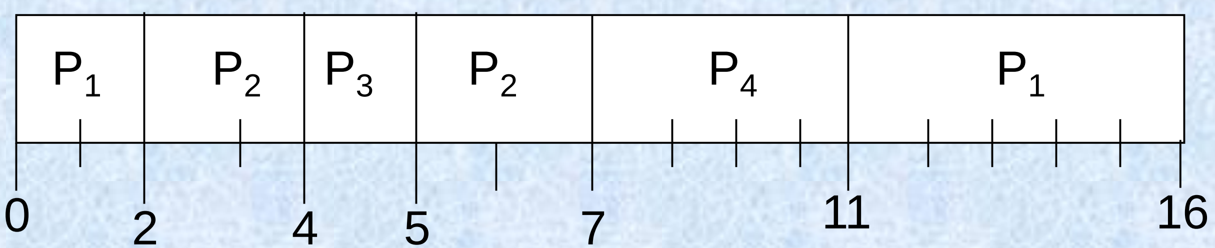
Disadvantages:

- Predictability of longer processes is reduced
- If estimated time for process not correct, the operating system may abort it (in case where the estimate is required to be given by the user)
- Possibility of starvation for longer processes

Example of Preemptive SJF/SRTF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



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

Determining Length of Next CPU Burst

- Can only estimate the length.
- Overhead of storing the last burst info
- Can be done by using the length of previous CPU bursts, using exponential averaging:

1. $t_n =$ actual lenght of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

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

Examples of Exponential Averaging

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

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots + (1 - \alpha)^j \alpha\, t_{n-j} + \ldots + (1 - \alpha)^n \tau_1$$

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

Priority Scheduling

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

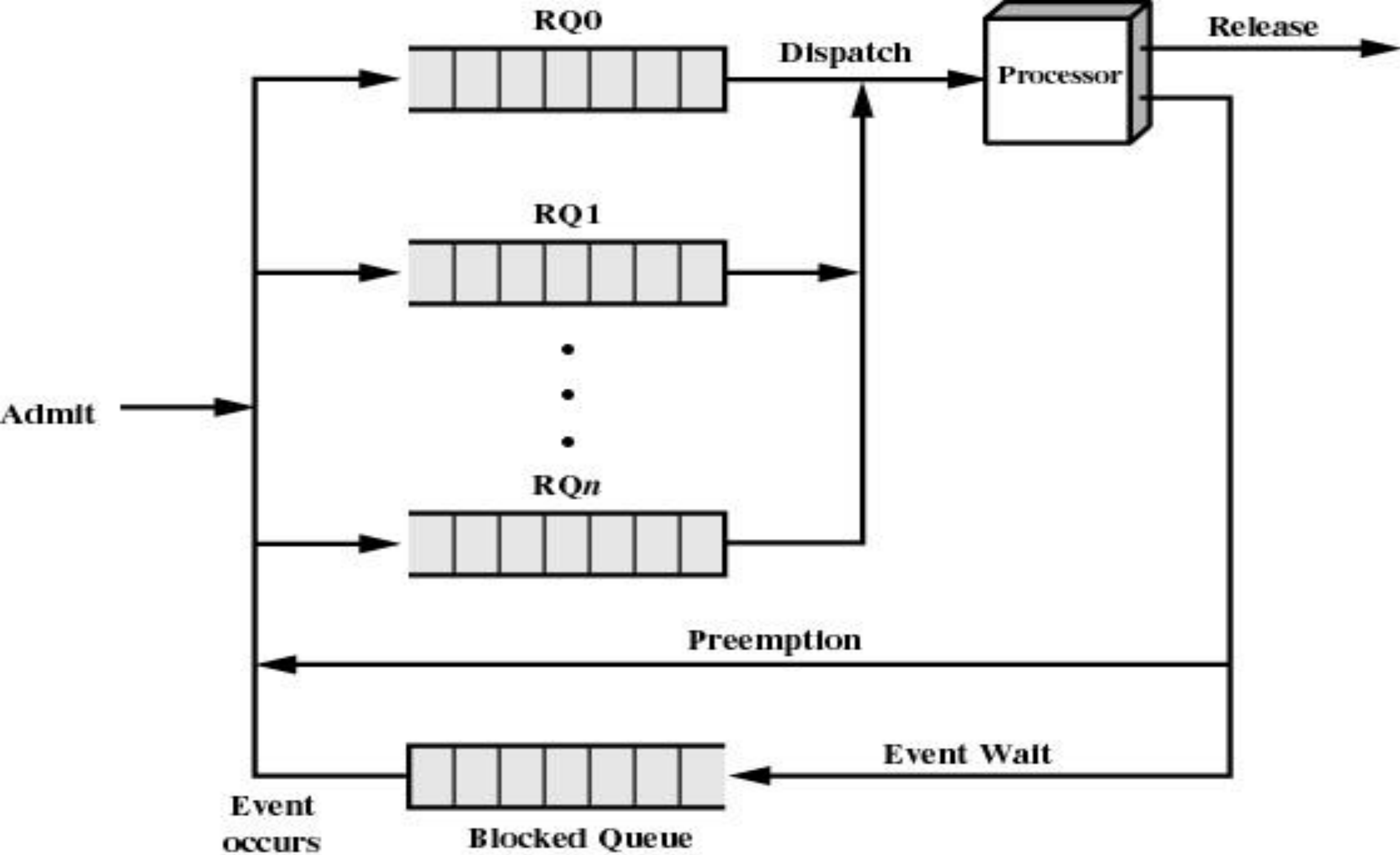


Figure 9.4 Priority Queuing

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

Example: RR With Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

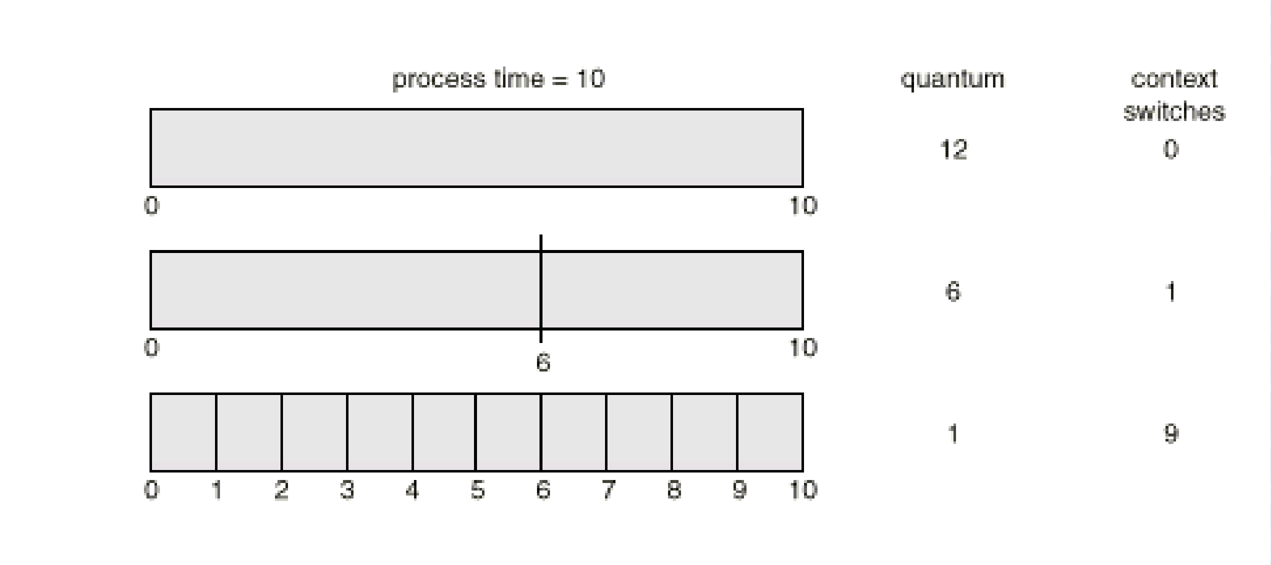
- The Gantt chart is:

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

- Typically, higher average turnaround than SJF, but better *response*.

Smaller Time Quantum Increases Context Switches



- Rule of thumb: time quantum should be slightly larger than the time required for a typical interaction. also the time quantum should be about 10 times larger than the context switch time
- RR is unfair to I/O bound processes=> Use Virtual RR(VRR) with an auxiliary queue

Virtual Round-Robin

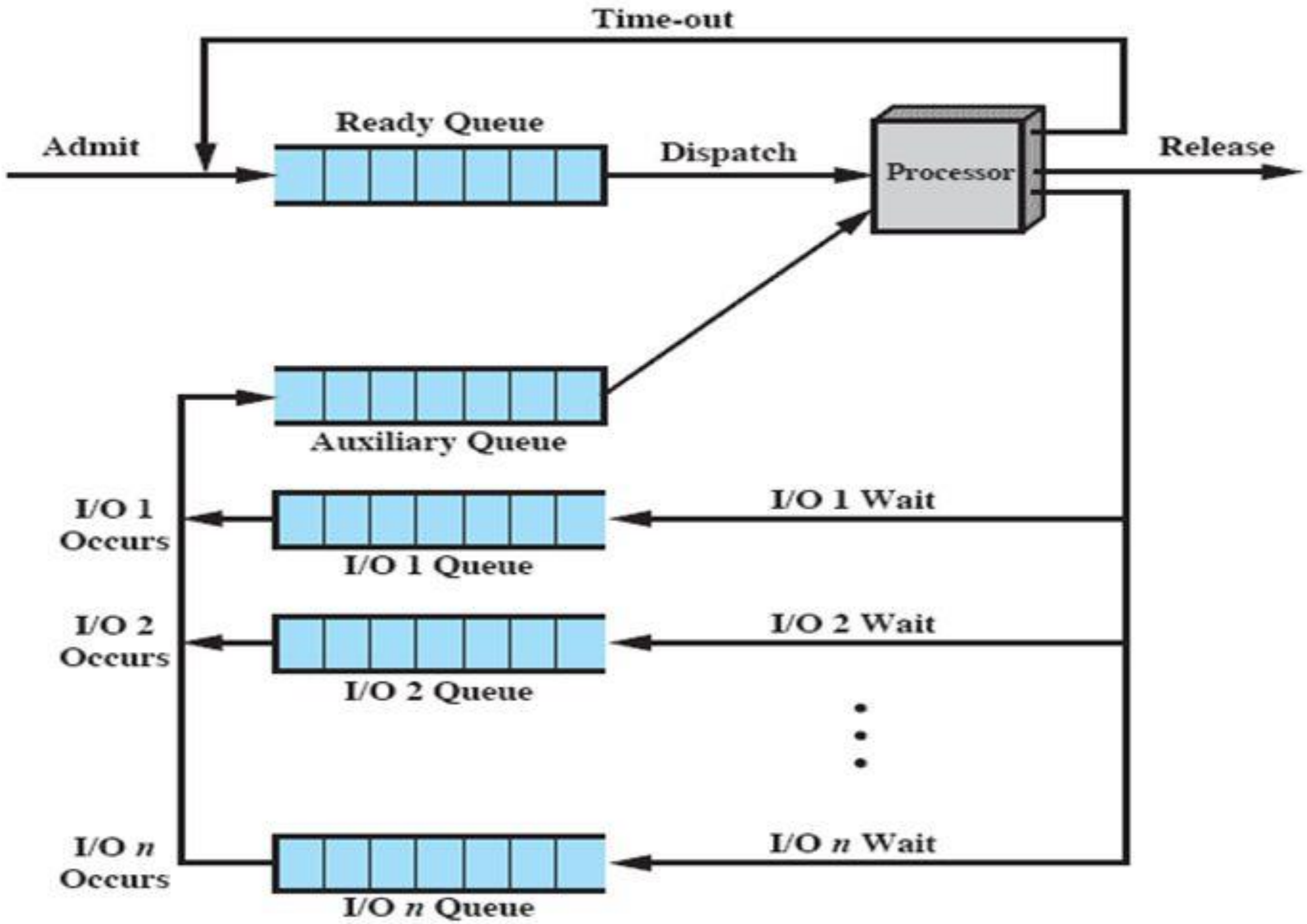
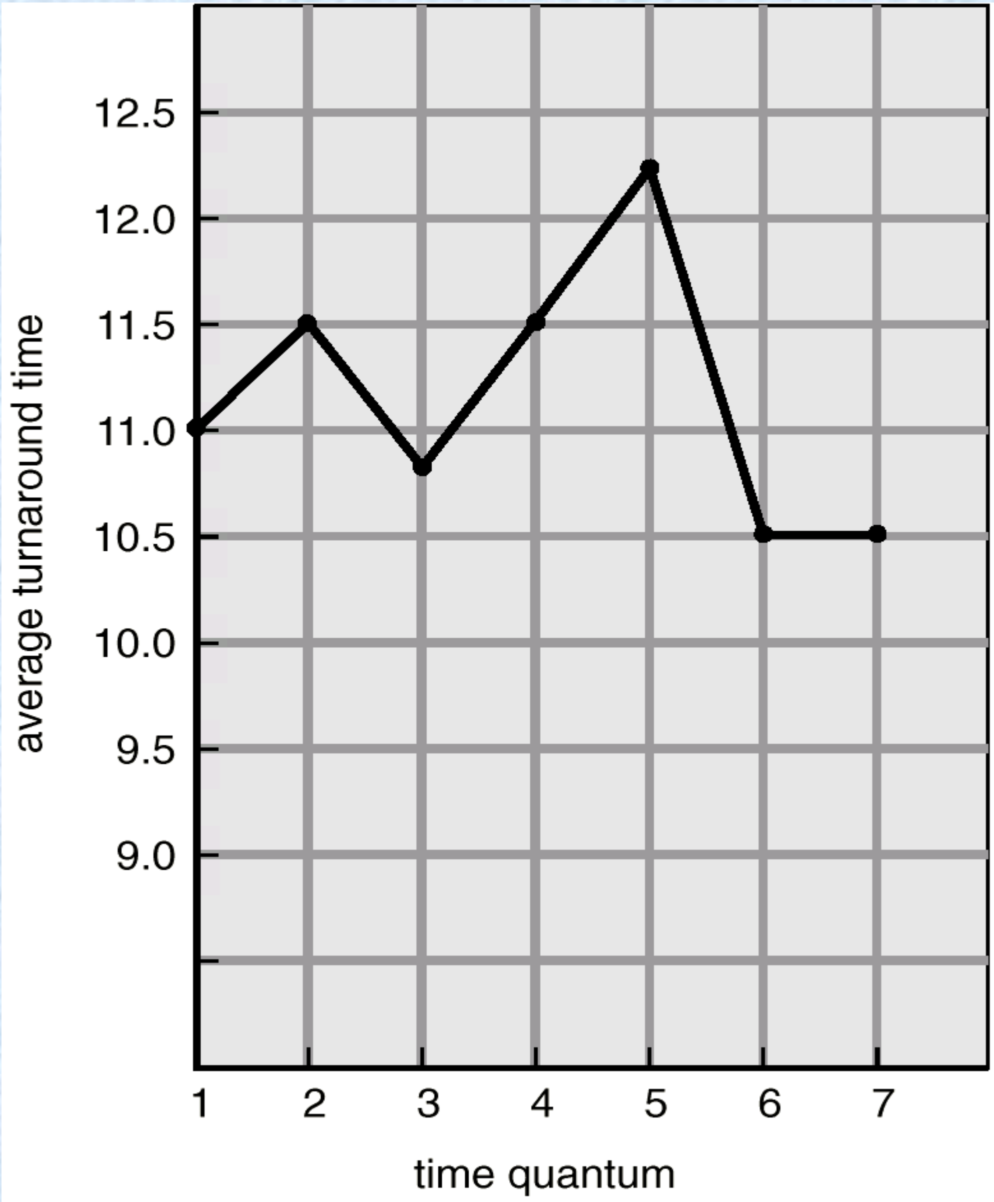


Figure 9.7 Queuing Diagram for Virtual Round-Robin Scheduler

Turnaround Time Varies With The Time Quantum

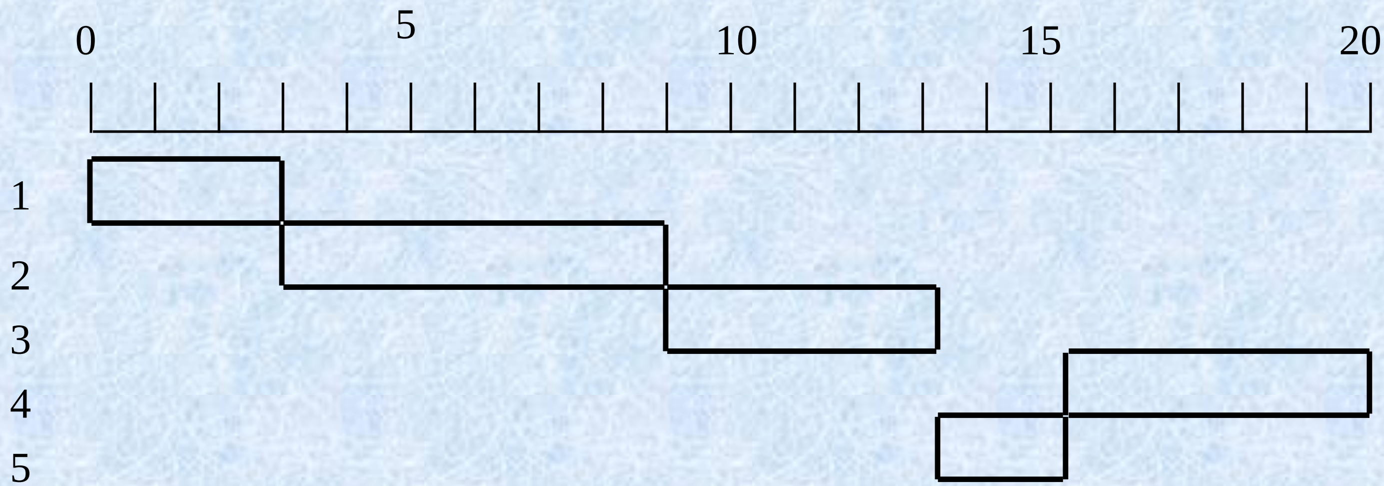


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Process Scheduling Example for HRRN

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Highest Response Ratio Next (HRRN)



- Choose next process with the highest ratio (normalized turnaround time)

$$\frac{\textbf{time spent waiting + expected service time}}{\textbf{expected service time}}$$

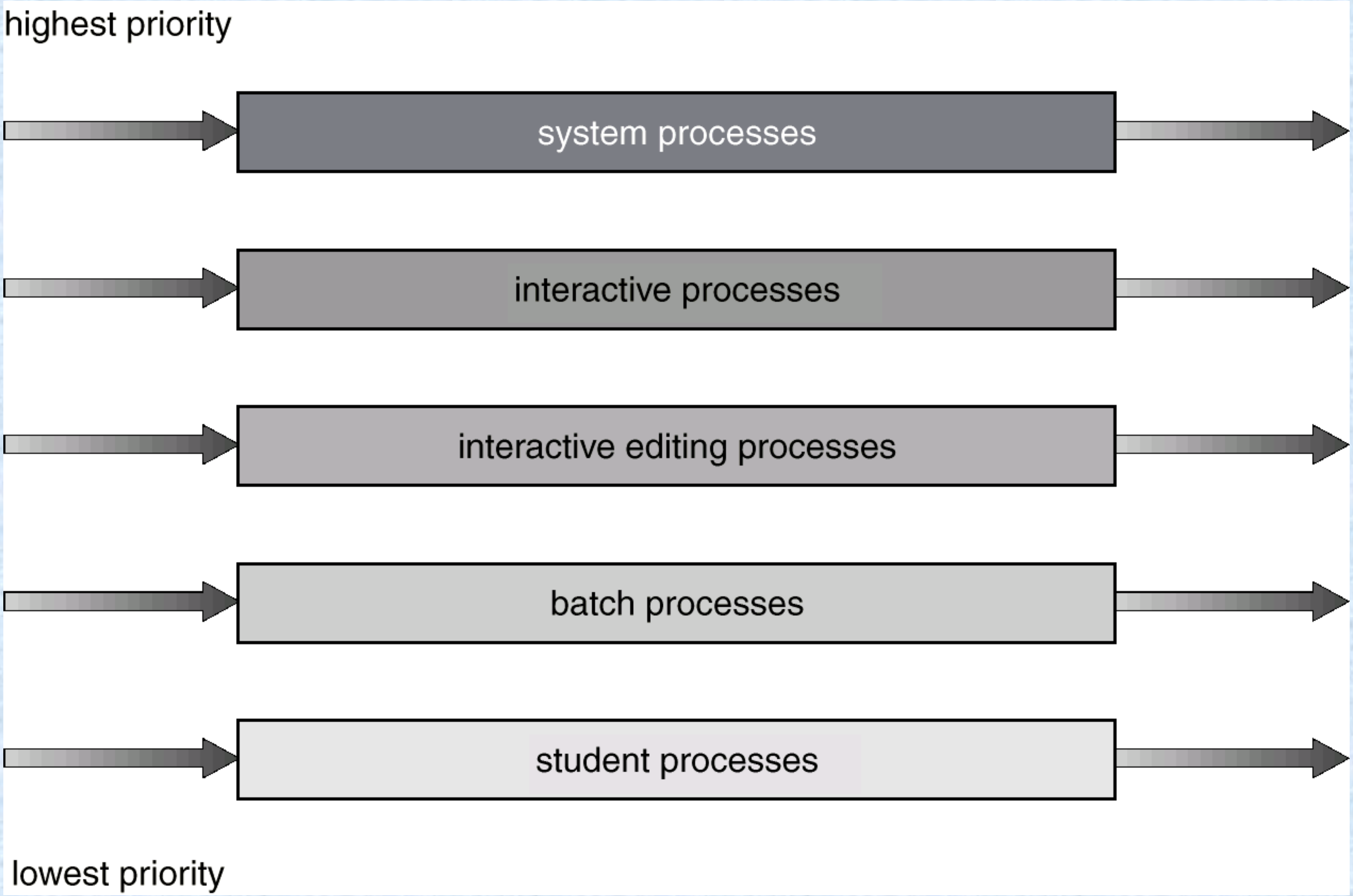
•This is a Nonpreemptive strategy

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Multilevel Queue

- Ready queue is partitioned into separate queues: example:
  foreground (interactive)
  background (batch)
- 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 and 20% to background in FCFS

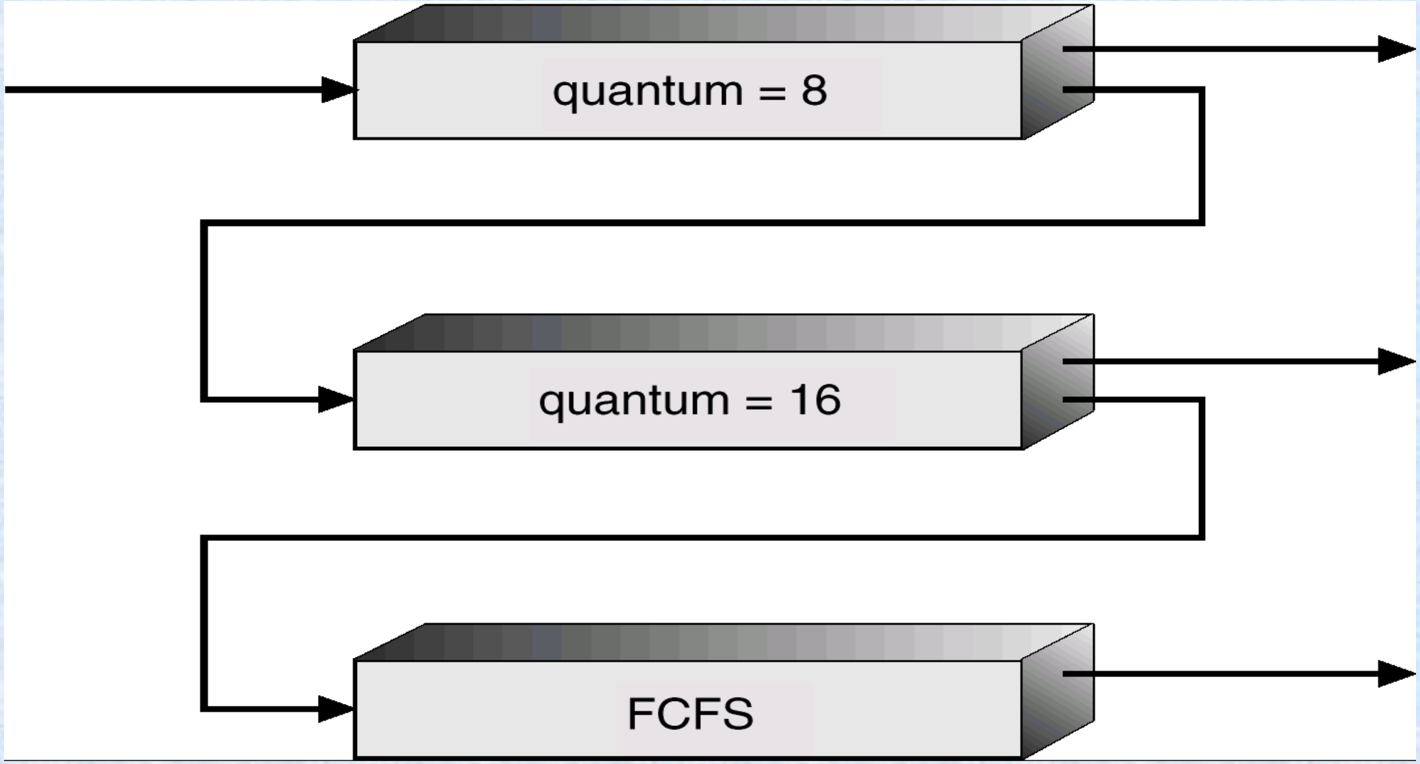
Multilevel Queue Scheduling



Multilevel Feedback Queue

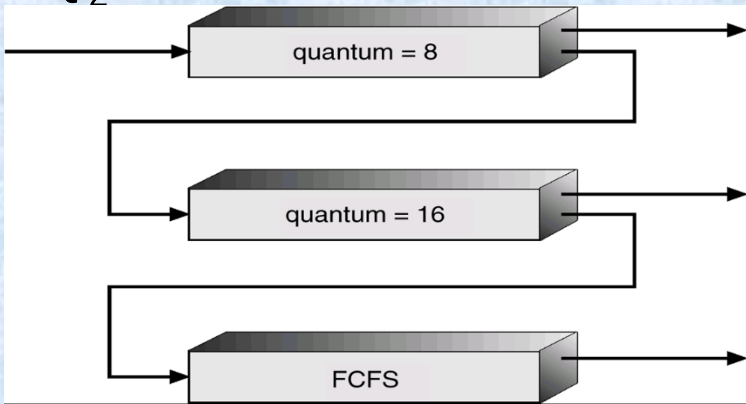
- Separate different processes according to their CPU burst characteristics. Preemption employed
- A process can move between the various queues; aging can be implemented this way.
- Multilevel-feedback-queue scheduler is defined by the following parameters:
  - number of queues
  - scheduling algorithms 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

Multilevel Feedback Queues



Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – time quantum 8 milliseconds
  - Q_1 – 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.



Fair-share Scheduling

- User’s application runs as a collection of processes (sets)
- User is concerned about the performance of the application made up of a set of processes
- Need to make scheduling decisions based on process sets(groups)
- Think of processes as part of a group
- Each group has a specified share of the machine time it is allowed to use
- Priority is based on the time this processes is active, and the time the other processes in the group have been active

Fair Share Scheduling

- Values defined
  - $P_j(i)$ =Priority of process *j* at beginning of ith interval
  - $U_j(i)$ =Processor use by process *j* during ith interval
  - $GU_k(i)$ =Processor use by group *k* during ith interval
  - $CPU_j(i)$ =Exponentially weighted average for process *j* from beginning to the start of ith interval
  - $GCPU_k(i)$ =Exponentially weighted average for group *k* from beginning to the start of ith interval
  - W_k =Weight assigned to group *k*, $0 \le W_k \le 1$, $\Sigma_k W_k = 1$

=> $CPU_j(1)=0$, $GCPU_k(1)=0$, i=1,2,3,….

Fair Share Scheduling

- Calculations (done each second):
  - $P_j(i) = Base_j + CPU_j(i)/2 + GCPU_k(i)/(4*W_k)$
  - $CPU_j(i) = U_j(i-1)/2 + CPU_j(i-1)/2$
  - $GCPU_k(i) = GU_k(i-1)/2 + GCPU_k(i-1)/2$

- Values defined
  - $P_j(i)$ =Priority of process *j* at beginning of ith interval
  - $U_j(i)$ =Processor use by process *j* during ith interval
  - $GU_k(i)$ =Processor use by group *k* during ith interval
  - $CPU_j(i)$ =Exponentially weighted average for process *j* from beginning to the start of ith interval
  - $GCPU_k(i)$ =Exponentially weighted average for group *k* from beginning to the start of ith interval
  - W_k =Weight assigned to group *k*, $0 \leq W_k \leq 1$, $\Sigma_k W_k = 1$

=> $CPU_j(1)=0$, $GCPU_k(1)=0$, $i=1,2,3,\ldots$.

Fair Share Example

- Three processes A, B, C; B,C are in one group; A is by itself
- Both groups get 50% weighting

	Process A			Process B			Process C		
	Priority	Process	Group	Priority	Process	Group	Priority	Process	Group
t=0	60	0	0	60	0	0	60	0	0
A		+60	+60						
t=1	90	30	30	60	0	0	60	0	0
B					+60	+60			+60
t=2	74	15	15	90	30	30	75	0	30
A		+60	+60						
t=3	96	37	37	74	15	15	67	0	15
C						+60		+60	+60
t=4	78	18	18	81	7	37	93	30	37
A		+60	+60						
t=5	98	39	39	70	3	18	76	15	18
B					+60	+60			+60
t=6	78	19	19	94	31	39	82	7	39
A		+60	+60						
t=7	98	39	39	76	15	19	70	3	19
C						+60		+60	+60
t=8	78	19	19	82	7	39	94	31	39
A		+60	+60						
t=9	98	39	39	70	3	19	76	15	19
B					+60	+60			+60
t=10	78	19	19	94	31	39	82	7	39
A		+60	+60						
t=11	98	39	39	76	15	19	70	3	19
C						+60		+60	+60
t=12	78	19	19	82	7	39	94	31	39

Traditional UNIX Scheduling

- Priorities are recomputed once per second
- Base priority divides all processes into fixed bands of priority levels
- Adjustment factor used to keep process in its assigned band (called *nice*)

Bands

- Decreasing order of priority
  - Swapper
  - Block I/O device control
  - File manipulation
  - Character I/O device control
  - User processes
- Values
  - ❄ $P_j(i)$ =*Priority of process j at start of ith interval*
  - ❄ $U_j(i)$ =*Processor use by j during the ith interval*
  - Calculations (done each second):
    - ❄ $CPU_j = U_j(i-1)/2 + CPU_j(i-1)/2$
    - ❄ $P_j = Base_j + CPU_j/2 + nice_j$