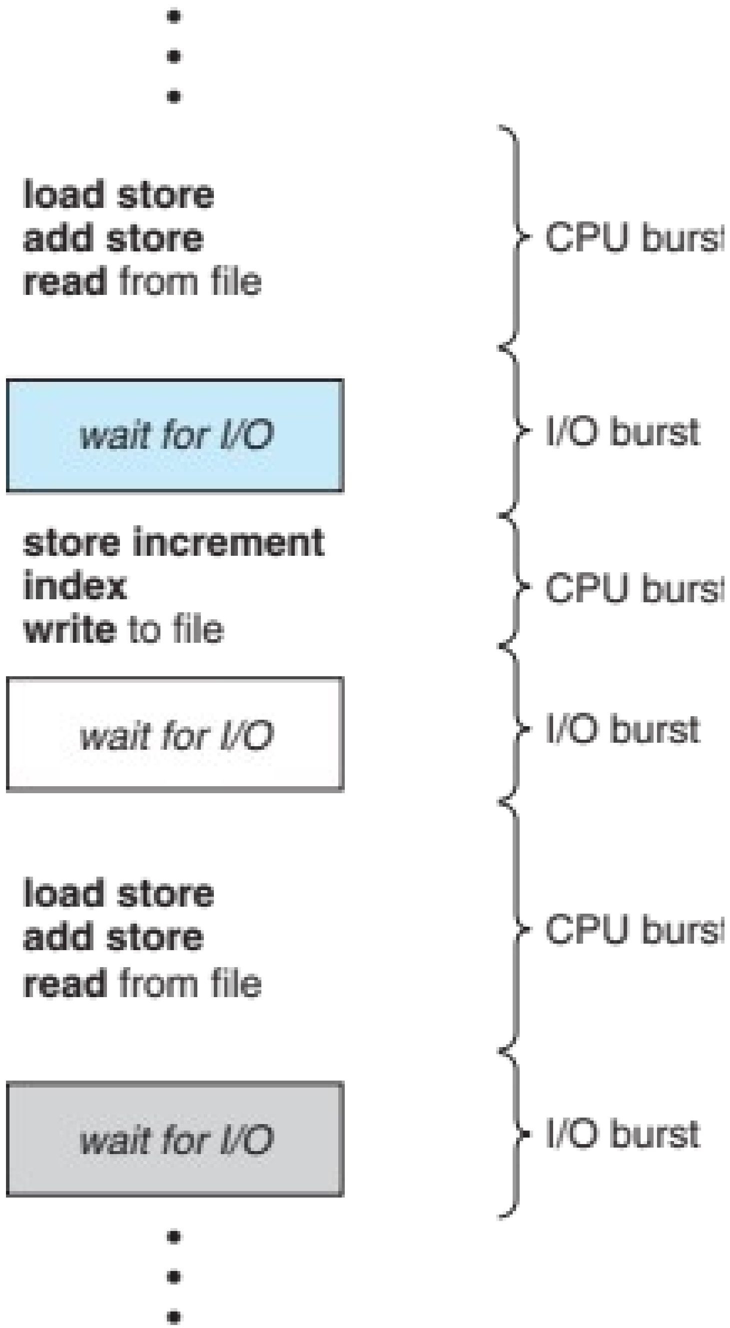
CECS 326
Operating Systems

CPU Scheduling

(Reference: Chapter 5. Operating system Concepts by Silberschatz, Galvin and Gagne)

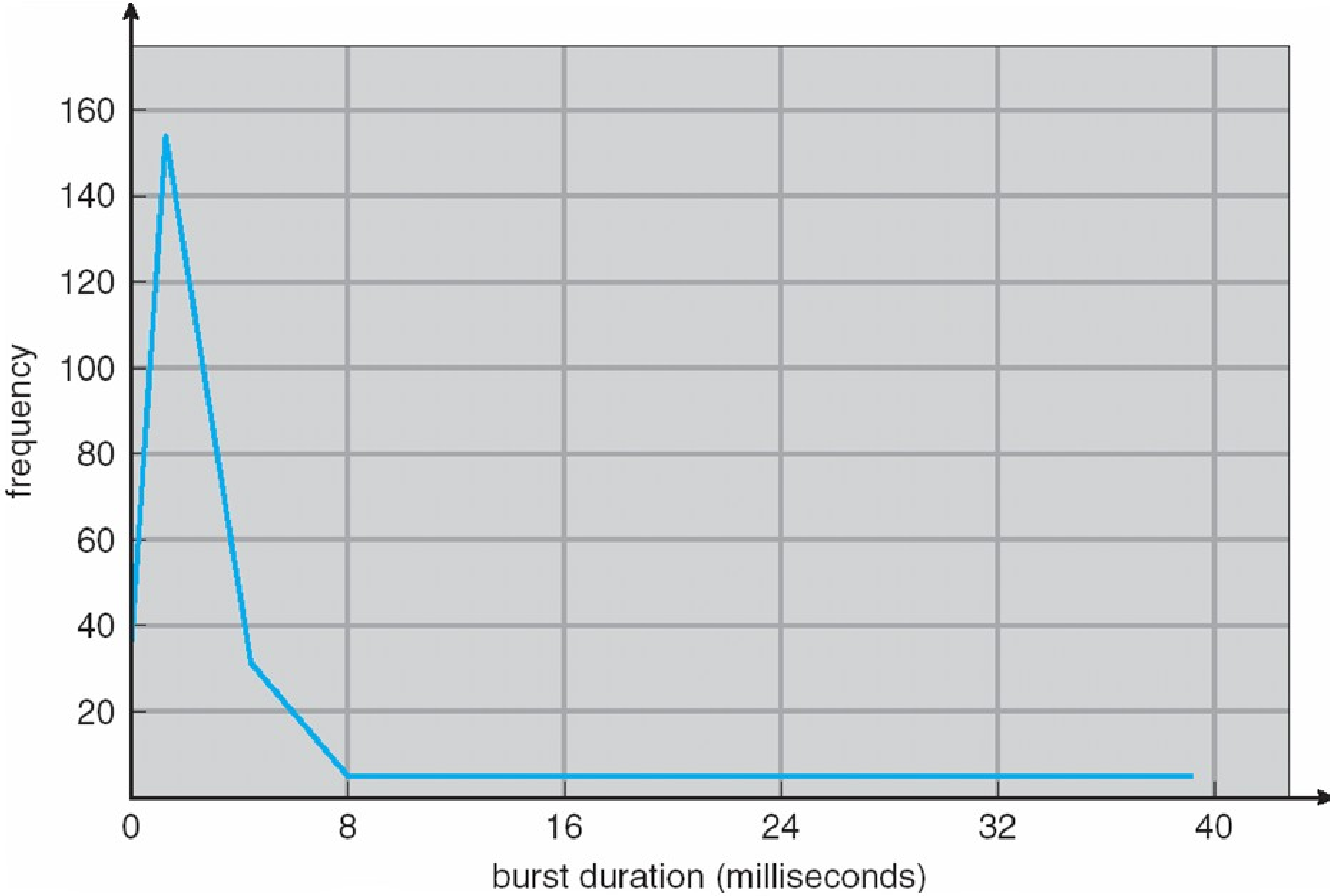
Basic Concepts

- 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
- May increase CPU utilization with multiprogramming



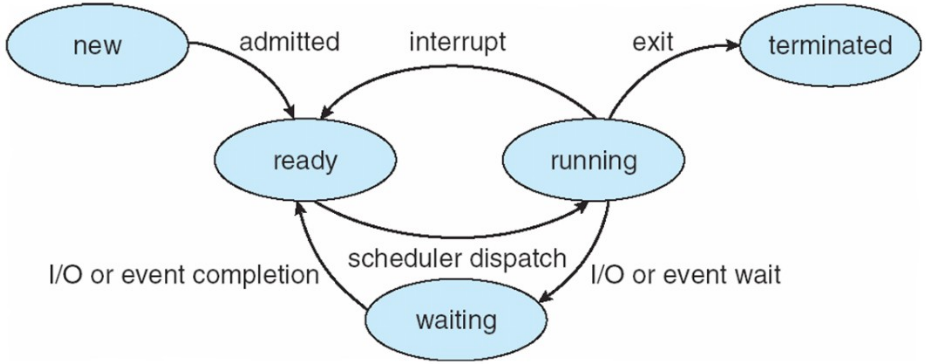
Histogram of CPU-burst Times

- Generally characterized as an exponential or hyperexponential distribution, with a large number of short CPU bursts and a small number of long CPU bursts.
- I/O-bound programs typically have many short CPU bursts, while CPU-bound programs might have a few long CPU bursts.



CPU Scheduler

- When the CPU becomes idle, the **short-term scheduler** selects one of the processes from the ready queue to be executed
- 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
- Scheduling under 2 and 3 is preemptive
  - Need to be aware of race conditions when data are shared among kernel processes

Dispatcher

- The 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 program
- Dispatch latency – time it takes for the dispatcher to stop one process and start another running, which should be as fast as possible since it is invoked in every process switch

Performance Measures of Scheduling Algorithms

- **CPU utilization** – keep the CPU as busy as possible (Maximize)
- **Throughput** – # of processes that complete their execution per time unit (Maximize)
- **Turnaround time** – amount of time to execute a particular process (Minimize)
- **Waiting time** – amount of time a process has been waiting in the ready queue (Minimize)
- **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) (Minimize)

First-Come, First-Served (FCFS) Scheduling

Given Process Burst Time

P_1 24

P_2 3

P_3 3

- Suppose three processes arrive at time 0, in the order P_1 , P_2 , P_3
- Gantt chart for the FCFS schedule is:

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

	Arrival Time	Start Time	Waiting Time
P_1	0	0	0
P_2	0	24	24
P_3	0	27	27

FCFS Scheduling (cont.)

- Suppose the same three processes arrive in the order P_2 , P_3 , P_1
- Then Gantt chart for the schedule is

	Arrival Time	Start Time	Waiting Time
P_1	0	6	6
P_2	0	0	0
P_3	0	3	3

- 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
  - E.g., consider one CPU-bound and many I/O-bound processes
- The preemptive version of SJF is called shortest-remaining-time-first (SRTF)

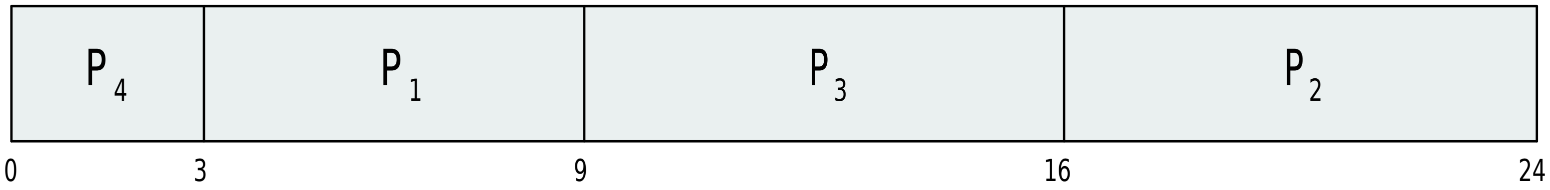
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst, and use these lengths to schedule the process with the shortest time
- SJF schedule is optimal – it gives the minimum waiting time for a given set of processes
  - Difficulty is knowing the length of the next CPU request

Example:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time
  - With SJF = (3 + 16 + 9 + 0) / 4 = 7
  - With FCFS = (0+6+14+21)/4=10.25

  (Assume order of arrival: P_1, P_2, P_3, P_4)

	Arrival Time	SJF		FCFS	
		Start Time	Waiting Time	Start Time	Waiting Time
P_1	0	3	3	0	0
P_2	0	16	16	6	6
P_3	0	9	9	14	14
P_4	0	0	0	21	21

Determining Length of Next CPU Burst

- Can only estimate – 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 with exponential averaging

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

- Commonly, α set to ½

Prediction of the Length of the Next CPU Burst

- Prediction () 10 8 6 6 5 9 11 12
- Actual () 6 4 6 4 13 13 13 13

Examples of Exponential Averaging

- α =0
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- α =1
  - $\tau_{n+1} = \alpha\, t_n = t_n$
  - Only the actual last CPU burst counts
- To see the behavior of the exponential average, we expand the formula yielding:

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

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

Expansion of the Exponential Averaging Formula

$=$

$=$

$=$

$=$

$=$

$= \ldots$

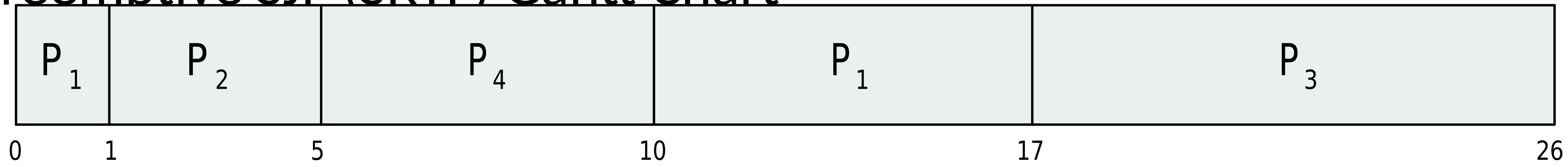
$=$

Shortest-Remaining-Time-First (SRTF) Scheduling

- 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 (SRTF) Gantt Chart



- Average waiting time
  - With SRTF = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5
  - With SJF = [(0+(8-1)+(17-2)+(12-3)]/4 = 31/4 = 7.75

Time	0	1	2	3	5	10	17	26
Processes in System & Their (Remaining) Time	P_1 (8)	P_1 (7)	P_1 (7)	P_1 (7)	P_1 (7)	P_1 (7)	P_1 (0)	
		P_2 (4)	P_2 (3)	P_2 (2)	P_2 (0)			
			P_3 (9)	P_3 (9)	P_3 (9)	P_3 (9)	P_3 (9)	P_3 (0)
				P_4 (5)	P_4 (5)	P_4 (0)		
Process dispatched	P_1 (8)	P_2 (4)	P_2 (3)	P_2 (2)	P_4 (5)	P_1 (7)	P_3 (9)	

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (convention: smallest integer $\equiv$ highest priority). Scheduling may be preemptive or nonpreemptive.
- SJF is priority scheduling where priority is based on the predicted next CPU burst time
  - Problem: starvation (low priority processes may have to wait a long time to get executed
  - Solution: aging (priority of process increases as time progresses)

Example of Priority Scheduling (Nonpreemptive)

- Given the following processes in the system

Process	Burst Time	Priority
P_1	1	1
P_2	5	2
P_3	2	4
P_4	1	5
P_5	10	3

- Priority scheduling Gantt Chart

- Average waiting time = (0+1+16+18+6)/5 = 8.2

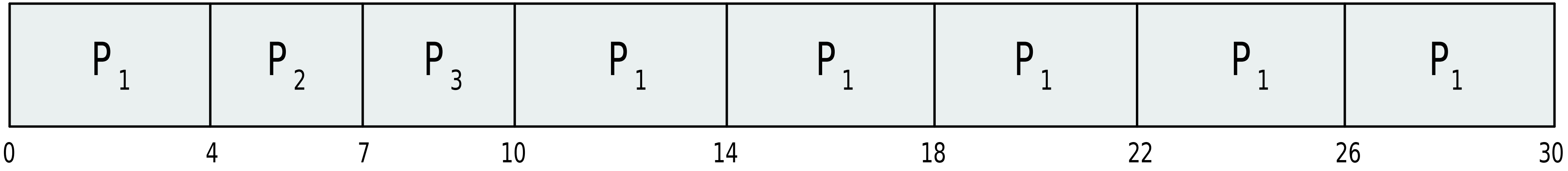
Round Robin (RR) Scheduling

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

Example of RR with Time Quantum q = 4

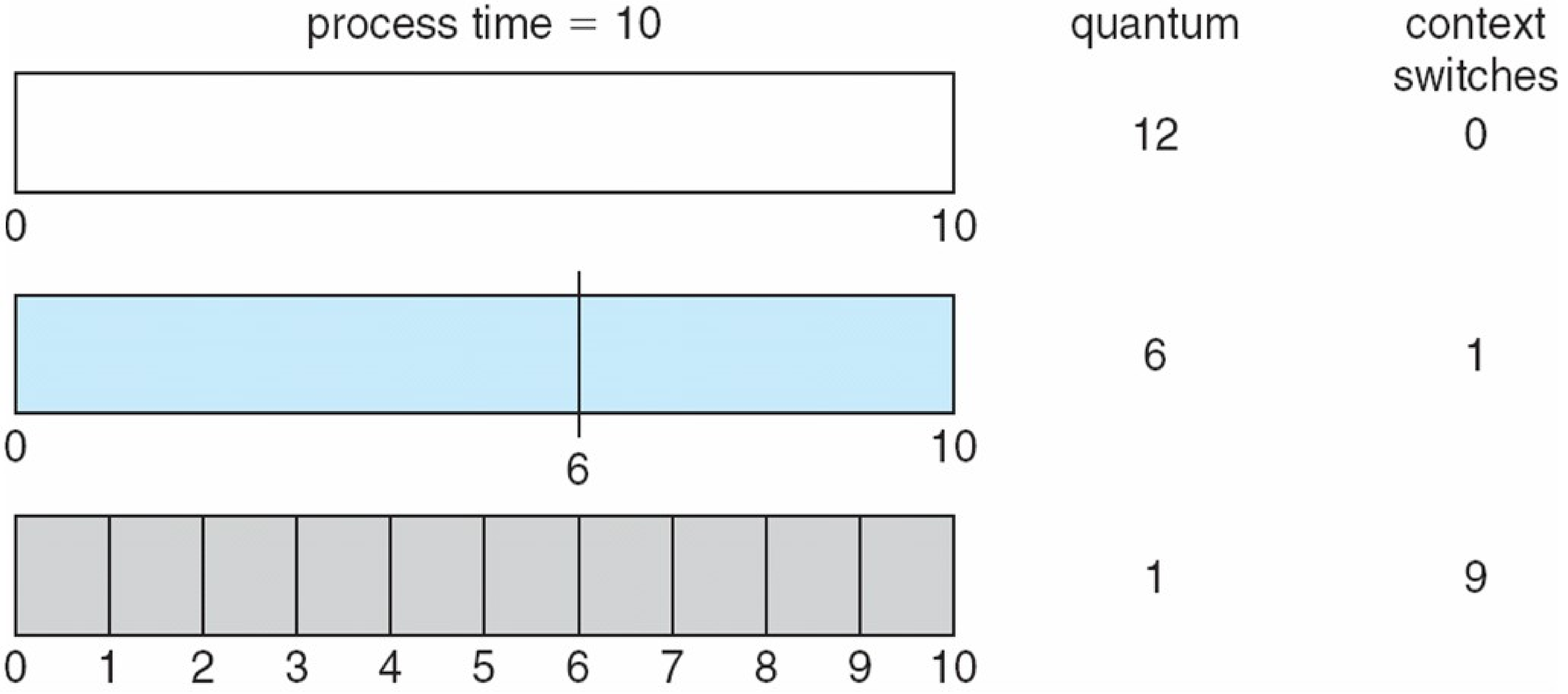
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

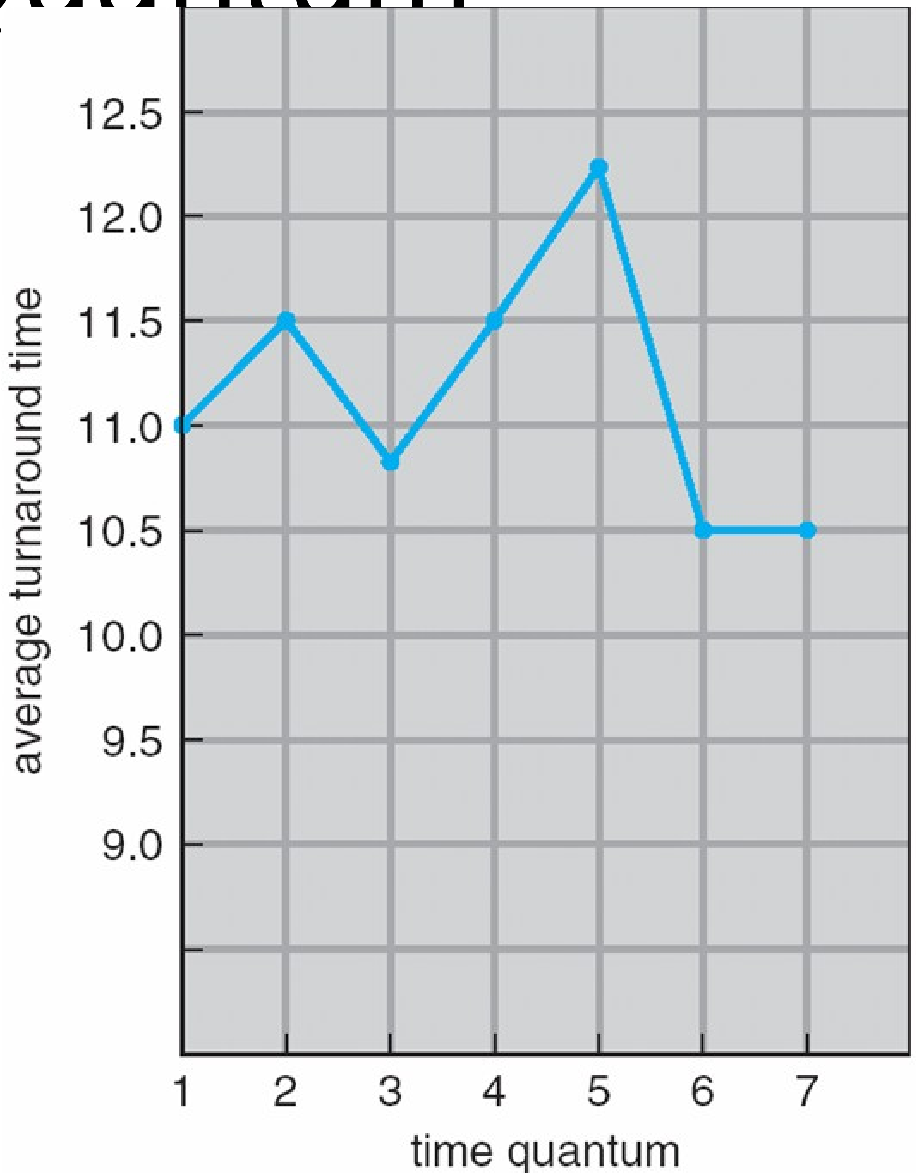


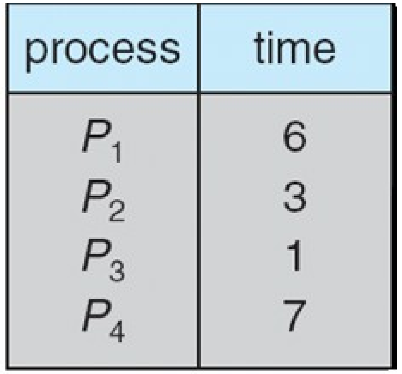
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
- E.g., q may be set to 10ms-100ms,with context switch < 10 μsec

Time Quantum and Context Switch Time



Turnaround Time Varies with 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

For schedule with q=5:

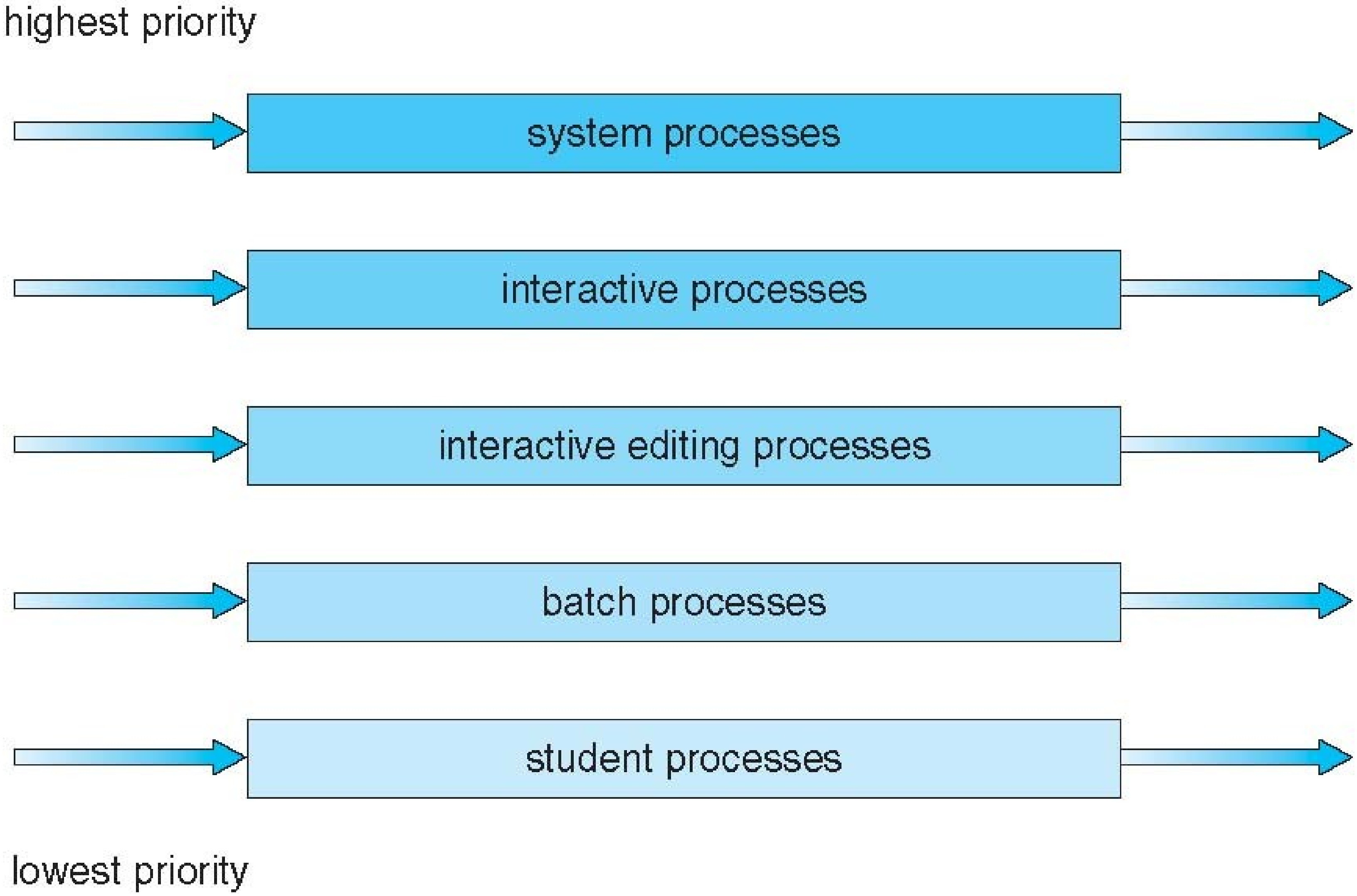
P_1	P_2	P_3	P_4	P_1	P_4	
0	5	8	9	14	15	17

Average turnaround time
=(15+8+9+17)/4=12.25

Multilevel Queues

- In order to enable different policies to be applied to different types of jobs, some systems implement multilevel queues, where the 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; e.g., 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling



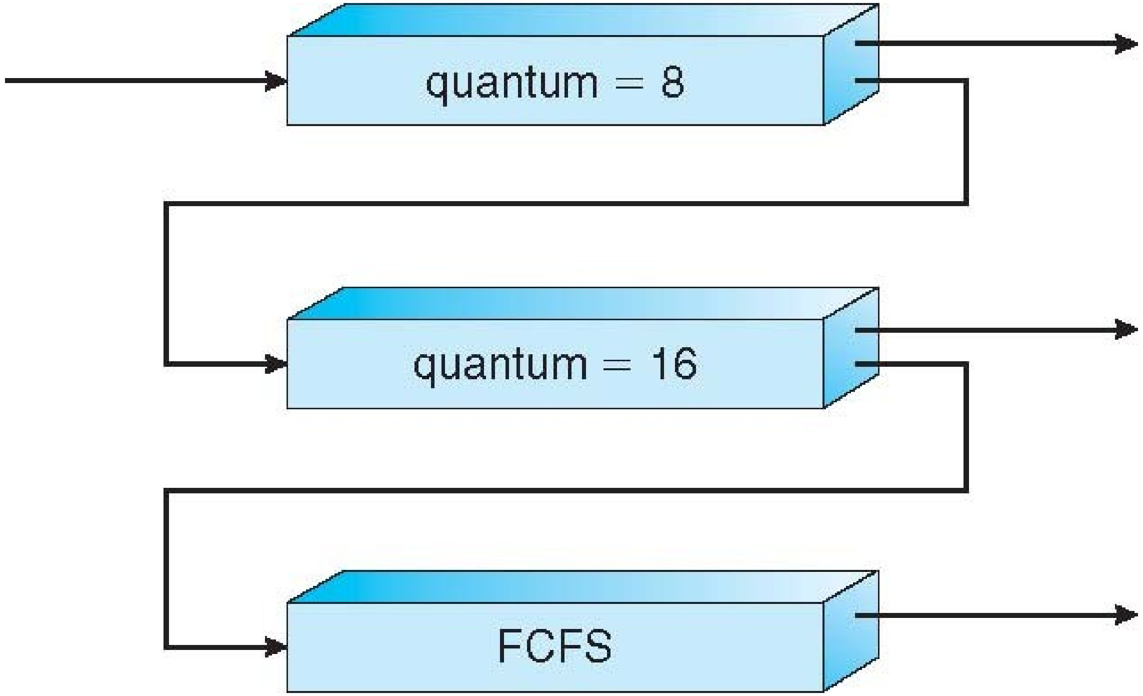
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way to counter the starvation issue with multilevel queues
- 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



Thread Scheduling

- There is distinction between user-level and kernel-level threads
- When threads are supported, scheduling units are threads, not processes
- May use many-to-one or many-to-many model. Thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

- Most applications that run on modern computers as separate processes, each consisting of multiple threads
- Benefits of multithreaded programming include:
  - Responsiveness: one thread can attend to user demands while other threads continue to execute
  - Resource sharing: threads in a process share memory & other resources
  - Economy: no additional resource allocation for creation of new threads
  - Scalability, with multiprocessor architecture
- Pthread, refers to the POSIX standard, defines APIs for thread creation and synchronization
- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Multiple-Processor Scheduling

- CPU scheduling becomes more complex when multiple CPUs are available
- Multiprocessor architecture: homogeneous processors or heterogeneous processors within a multiprocessor
- Approaches to multiple-processor scheduling
  - Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing
  - Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
    - Currently, most common
- Processor affinity – process has affinity for processor on which it is currently running
  - soft affinity – attempt to keep on the same processor, but not guaranteed
  - hard affinity
  - Variations including processor sets

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- Load balancing attempts to keep workload evenly distributed. Needed 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

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Benefits: faster and consumes less power
- The practice of multiple threads per core is also growing (e.g., hyperthreading on Intel)
    - Takes advantage of memory stall to make progress on another thread while memory retrieve happens