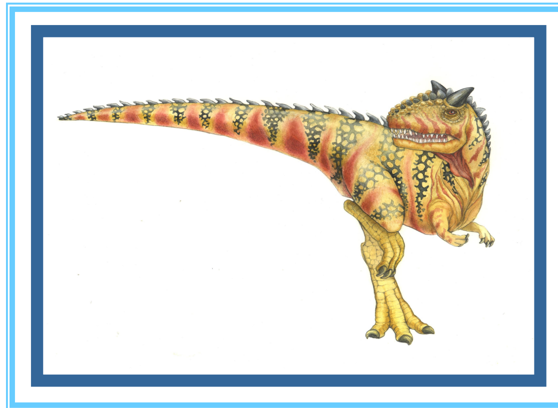
Chapter 5: Process Scheduling



Silberschatz, Galvin and Gagne ©2013

Chapter 5: Process Scheduling

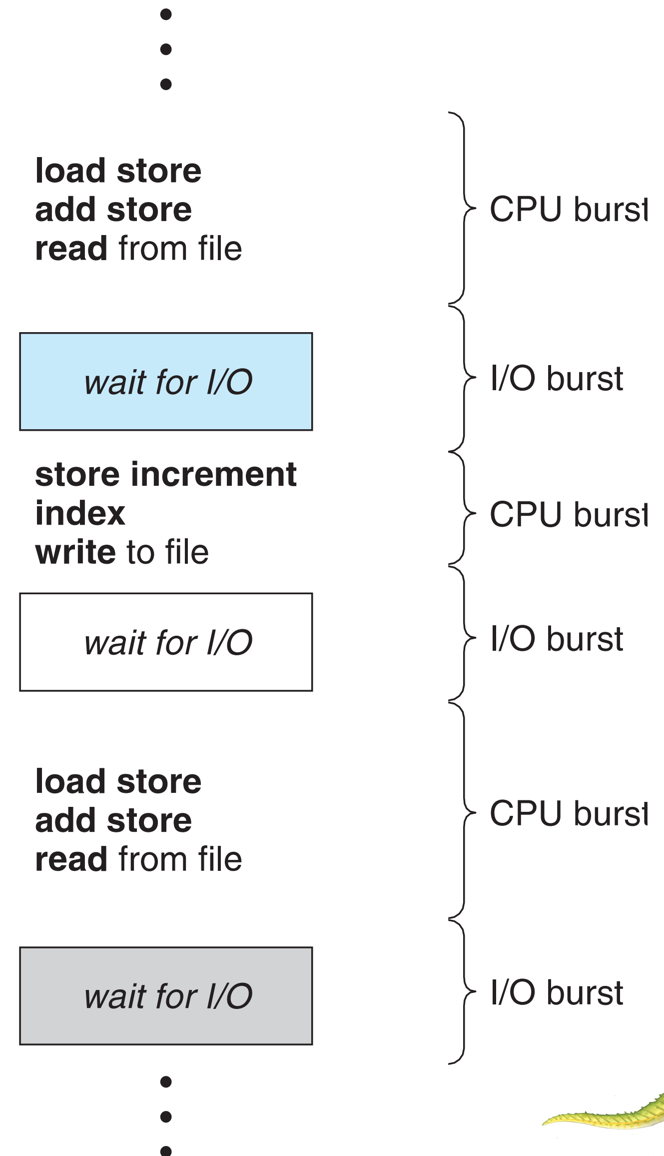
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multiple-Processor Scheduling
- Algorithm Evaluation

Objectives

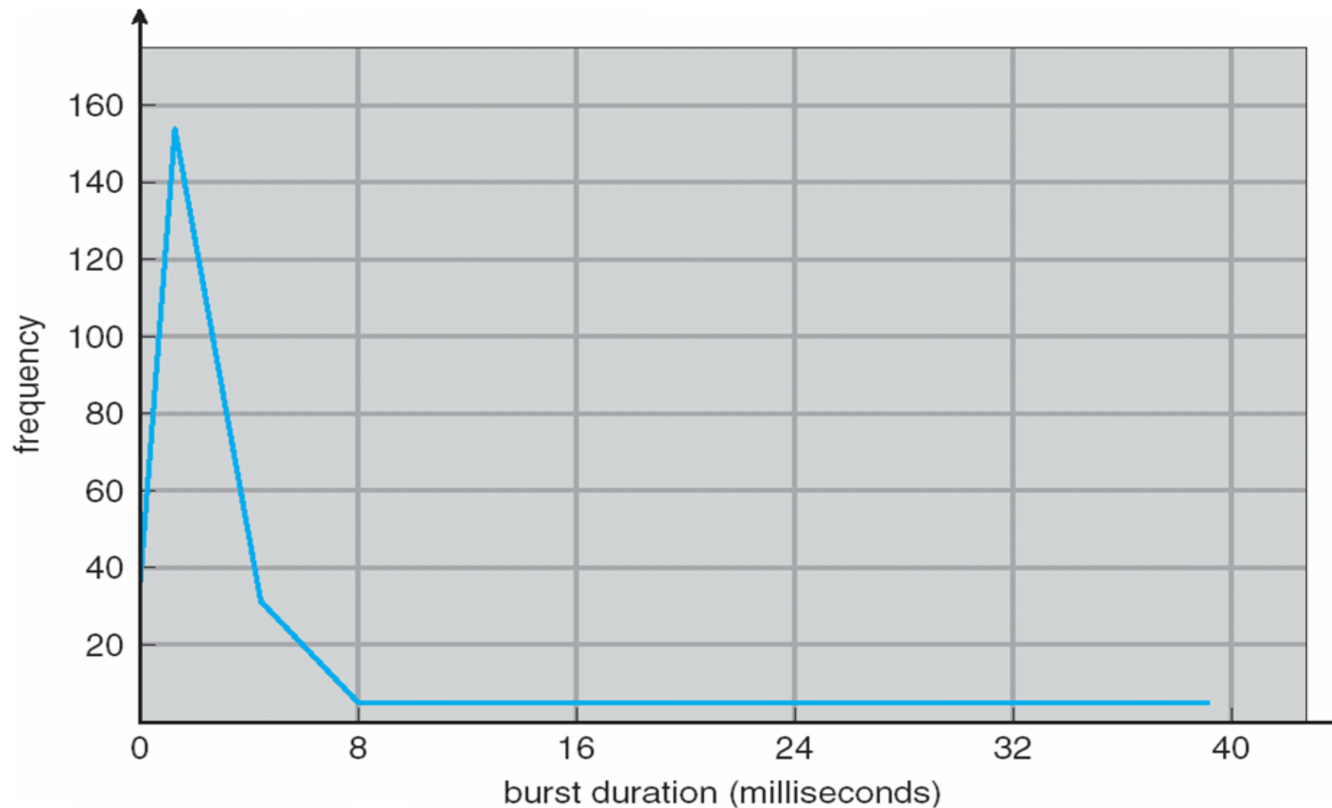
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU burst and I/O burst cycles – 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



- The extensive measurement of CPU bursts shows that the CPU burst duration is consisted of a large number of short CPU bursts and a small number of long CPU bursts.
- This curve is generally characterized as exponential or hyper-exponential

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place in the following four circumstances:
  1. Switches from running to waiting state (I/O request or wait())
  2. Switches from running to ready state (interrupts)
  3. Switches from waiting to ready (the completion of I/O)
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
  - The process releases the CPU voluntarily
- Scheduling under 2 and 3 is **preemptive,** which an result in **race** condition (discussed in Chapter 6)
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher is the module that 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
- The dispatcher should be as fast as possible, since it is invoked during each process switch

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. The interval from the time of submission of a process to the time of completion
- **Waiting time** – the sum of the periods that a process spent waiting in the ready queue
  - Noticing that CPU scheduling algorithm does not affect the total amount of time during which a process executes (on CPU) or does I/O. It affects only the amount of time that a process spends 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, in a time-sharing or interactive environment

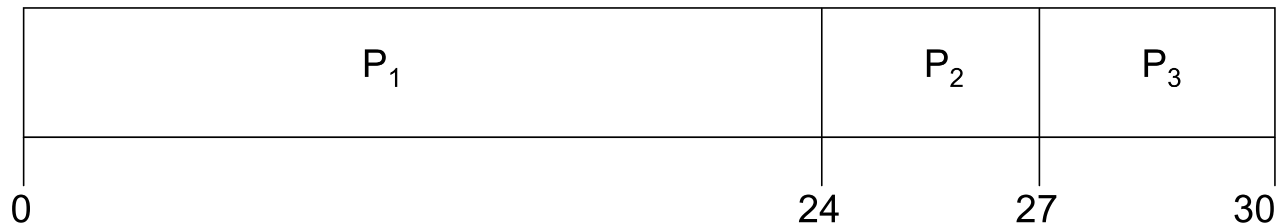
Scheduling Algorithm Optimization Criteria

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

First-Come, First-Served (FCFS) Scheduling

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
- The FIFO scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O

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

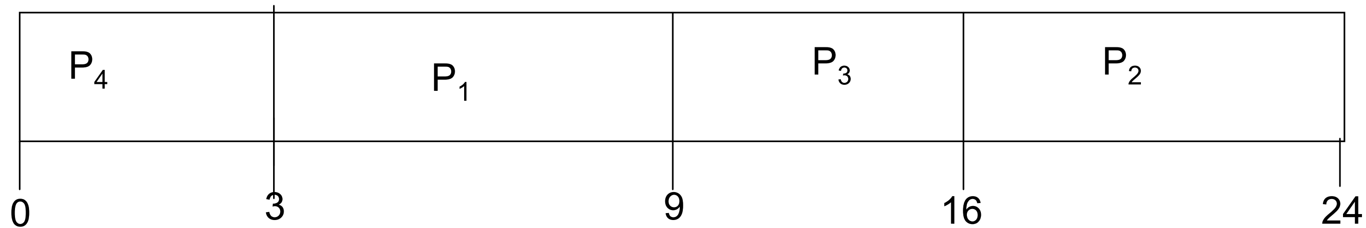
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - Moving a short process before a long process decreases the waiting time of the short process more than it increases the waiting time for the long process. Consequently, the average (or the total) waiting time decreases
  - The difficulty is knowing the **length of the next CPU request**

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

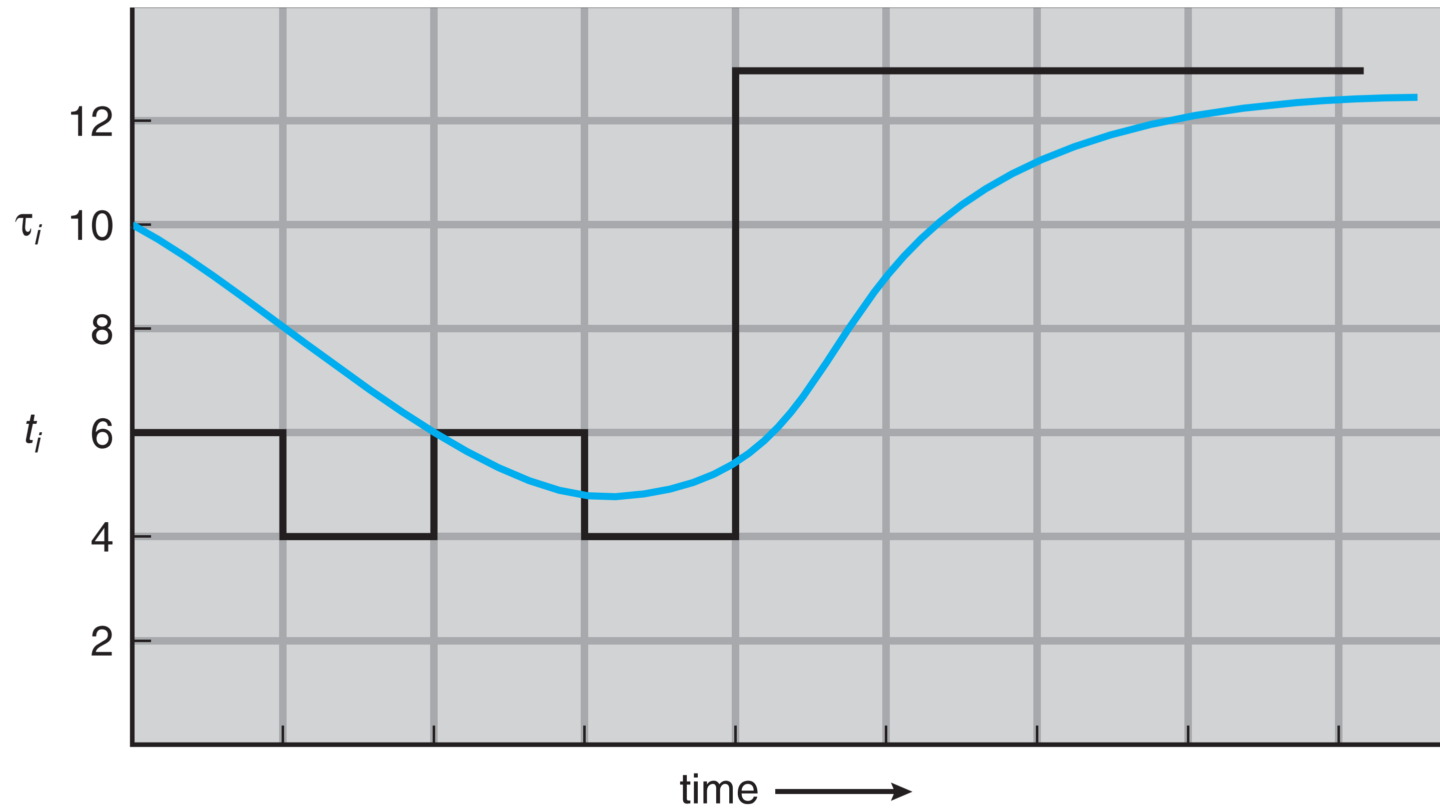
Determining Length of Next CPU Burst

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

1. $t_n =$ actual length of nth CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$
4. Define : $\tau_{n=1} = \alpha\ t_n + (1 - \alpha)\tau_n .$

- Commonly, α set to ½
- Preemptive version called **shortest-remaining-time-first**

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
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\, t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

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

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor, i.e., the impact on the estimate is reduced exponentially.

Preemptive and Nonpreemptive SJF

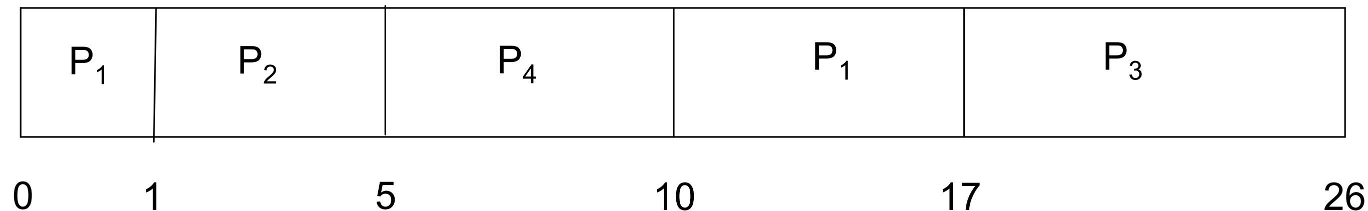
- The SJF can be either nonpreemptive or preemptive
- The choice arises when a process arrives at the ready queue while another process is still executing on the CPU
- The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. If this is the case, the next CPU burst length of this newly arrived process will be shorter than the CPU burst lengths of all processes currently in the ready queue (the SJF determines)
- A preemptive SJF algorithm will pre-empt the currently running process (returning to the ready queue with the remaining CPU time), whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst
- Preemptive SJF scheduling algorithm is sometime called **shortest-remaining-time first** scheduling

Example of Shortest-Remaining-Time-First

- Now we 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 Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

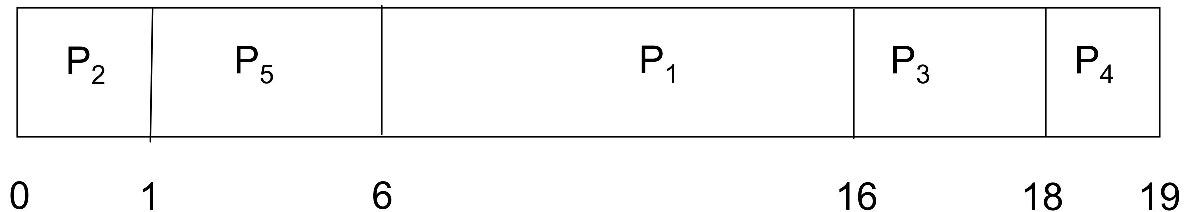
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Priority scheduling can be either preemptive or nonpreemptive
- Major 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 = 8.2 msec

Round Robin (RR)

- A round-robin scheduling is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to restrict the maximum amount of time that a process can occupy the CPU, thus enable the system to switch between processes

- 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 (as a circular 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

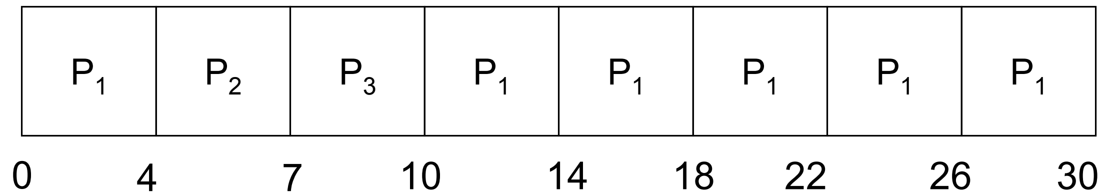
- Timer interrupts every quantum to schedule next process

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

Example of RR with Time Quantum = 4

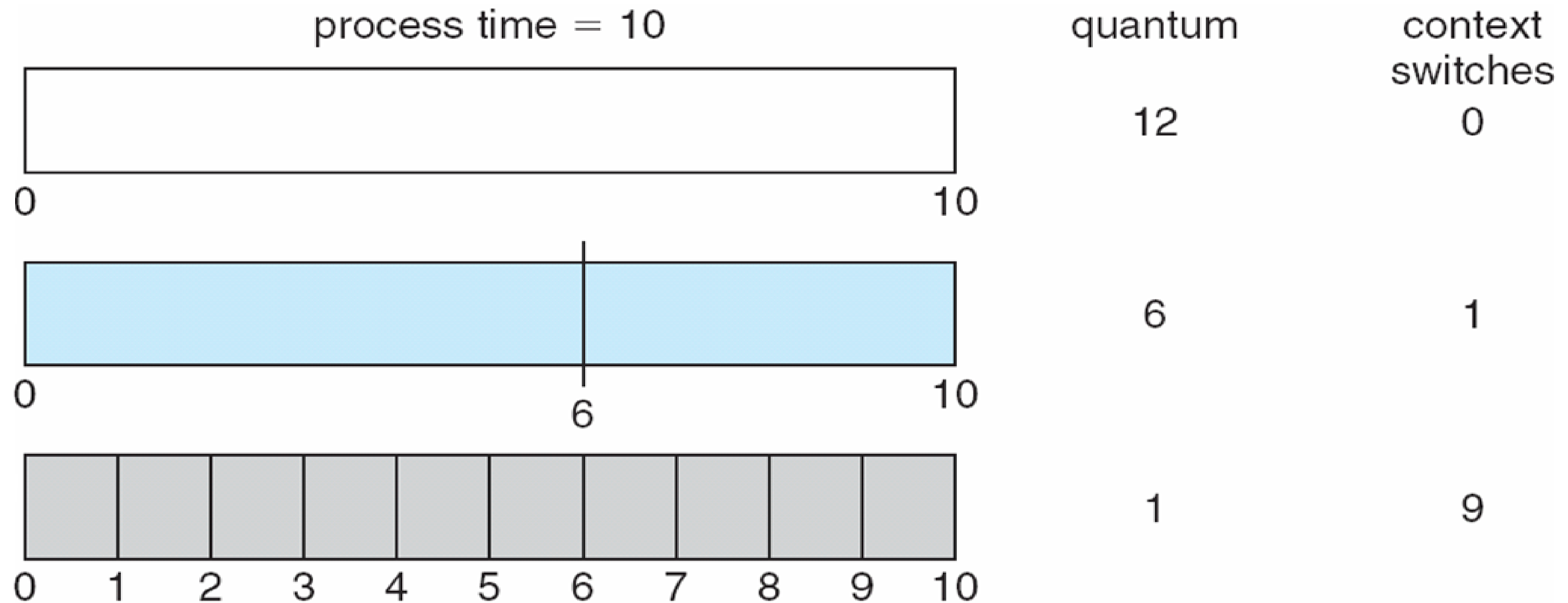
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



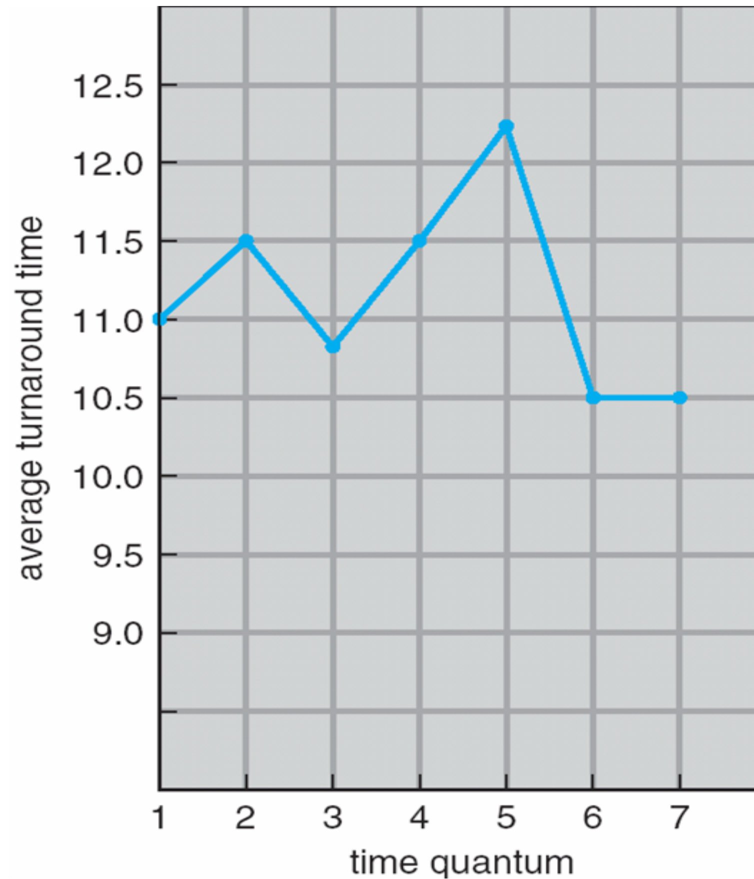
- Typically, higher average turnaround than SJF, but better ***response time***
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 microseconds

Time Quantum and Context Switch Time



How a smaller time quantum increases the number of context switches

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

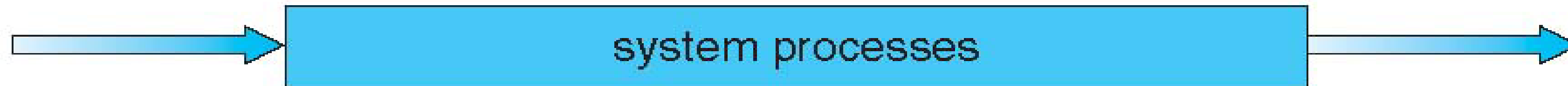
- The average turnaround time does not necessarily improves as the time quantum size increases
- In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single quantum
- The time quantum can not be too big, in which RR degenerates to an FCFS policy
- A rule of thumb: 80% of CPU bursts should be shorter than the time quantum q

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - **foreground** (interactive)
  - **background** (batch)
- Processes are permanently assigned to one queue when they enter the system, based on some property of the process, such as memory size, priority, or process type
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done among the queues”
  - Fixed-priority preemptive 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

highest priority



lowest priority

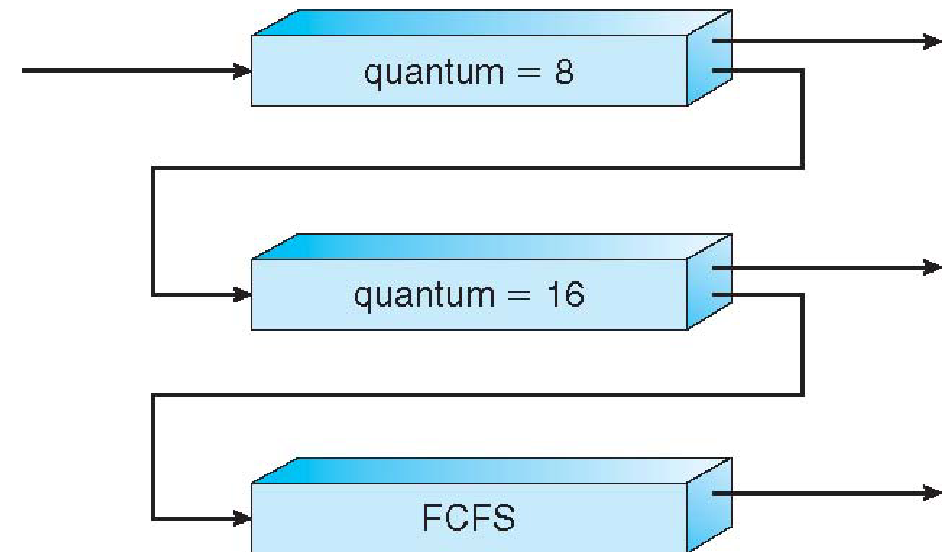
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - the 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

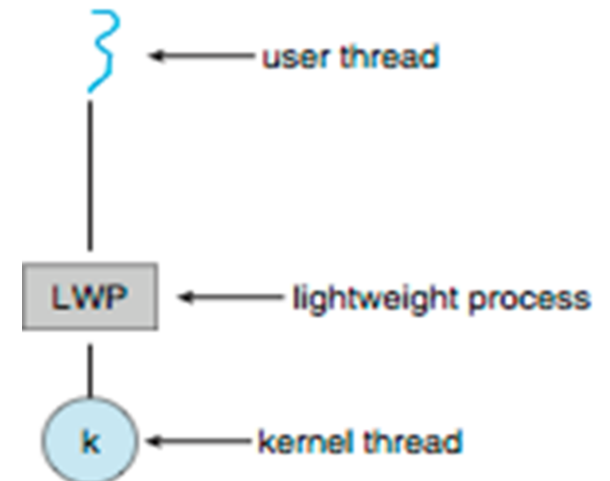
- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling – **preemptive**
  - 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
  - A process in queue 1 or 2 will be preempted by a process arriving for queue 0



Thread Scheduling

- When the OS supports threads, the kernel-level threads are being scheduled, not processes
  - User-level threads are managed by a thread library
- The OS typically uses an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**)
  - Appears to be a virtual processor on which process can schedule user threads to run on it
  - Each LWP attached to kernel thread



- Many-to-one and many-to-many models, 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
  - Systems using one-to-one mapping model, such as Windows, Linux, and Solaris, schedule threads using only SCS

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing. The 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 queue of ready processes
  - Currently, most common
- **Processor affinity** – process has affinity for processor on which it is currently running, esp. the cache content
  - **Soft affinity** – the OS attempt to keep a process running on the same processor, not guaranteeing it
  - **Hard affinity** – allow a process to specify a subset of processors on which it may run

NUMA and CPU Scheduling

- The main-memory architecture of a system can affect processor affinity
- Note that memory-placement algorithms can also consider affinity

Multiple-Processor Scheduling – Load Balancing

- On SMP systems, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – a specific task periodically checks the load on each processor, and if it finds an imbalance, pushes task from overloaded CPU to idle or less-busy CPUs
- **Pull migration** – idle processors pulls waiting task from a busy processor
- Push and pull migration need not to be mutually exclusive and are in fact often implemented in parallel on load-balancing systems

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
- **Memory stall**: a situation when a processor accesses memory, it spends a significant amount of time waiting for the data to become available, due to various reasons such as cache miss
- The scheduling can takes advantage of **memory stall** to make progress on another thread while memory retrieve happens
  - If one thread stalls while waiting for memory, the core can switch to another thread. This becomes a **dual-thread processor core**, or two logical processors
  - **A dual-threaded, dual-core system** presents **four** logical processors to the operating system

Multithreaded Multicore System

C compute cycle

M memory stall cycle

thread

C	M	C	M	C	M	C	M

time

$thread_1$

C	M	C	M	C	M	C

$thread_0$

C	M	C	M	C	M	C

time

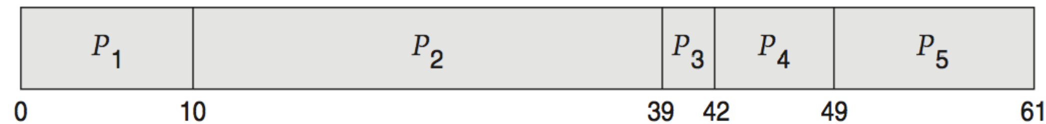
Algorithm Evaluation

- How to select CPU-scheduling algorithm for a particular system - difficult
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular pre-determined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

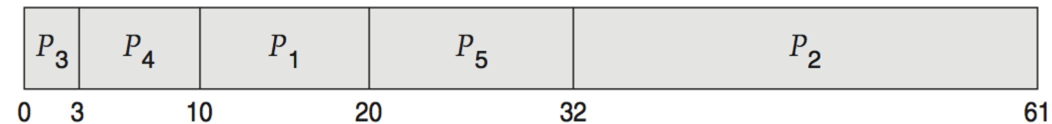
Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

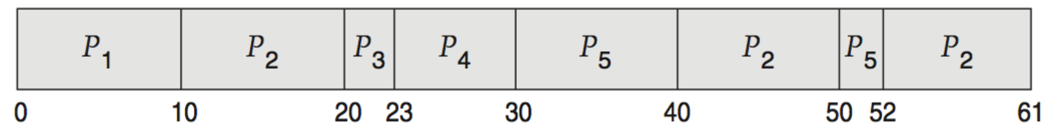
- For each algorithm, calculate minimum average waiting time
- Simple and fast, but requires exact numbers for input, applies only to those inputs
  - FCS is 28ms:



  - Non-preemptive SFJ is 13ms:



  - RR is 23ms:



Queueing Models

- Suppose the distribution pf CPU and I/O bursts are known, which may be measured and then approximated or simply estimated
- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc.
- 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.

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 models limited for a very few known distributions in order to compute the performance mathematically
- **Simulations** more accurate and general
  - Programmed model of computer 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

actual process execution

• • •
CPU 10
I/O 213
CPU 12
I/O 112
CPU 2
I/O 147
CPU 173
• • •

trace tape

simulation
FCFS

performance statistics for FCFS

simulation
SJF

performance statistics for SJF

simulation
RR ($q = 14$)

performance statistics for RR ($q = 14$)

Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary