Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

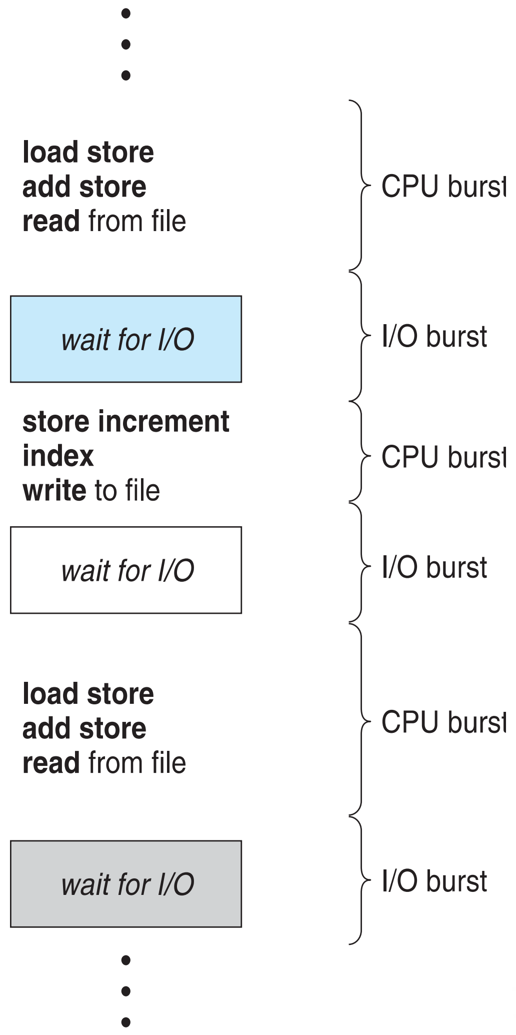
Zeinab Zali

CPU Scheduling

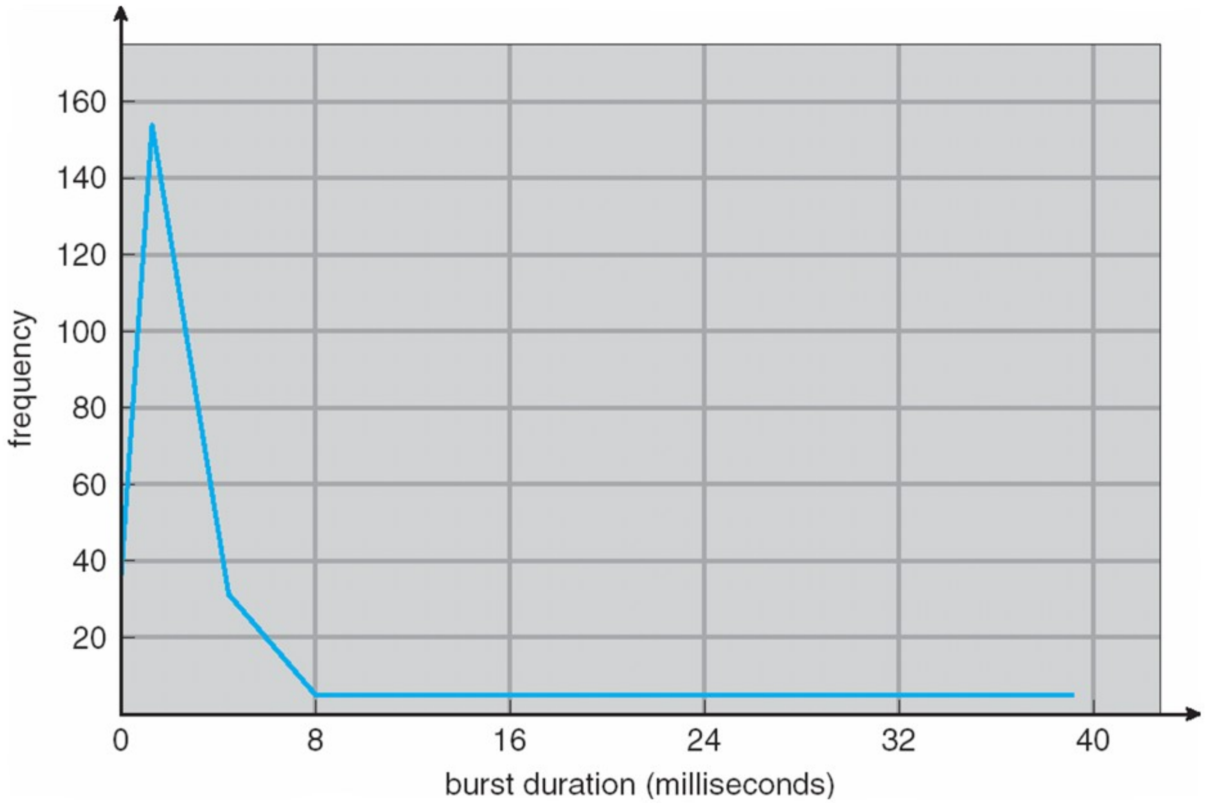
Basic Concepts

- 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

IO-bound and CPU-bound



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**
  - **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** قبضه نشدنی یا انحصاری
- All other scheduling is **preemptive** قبضه شدنی یا غیرانحصاری
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial 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

```
vmstat 1 3
```

```
cat /proc/pid/status
```

Scheduling Criteria

- **CPU utilization** : keep the CPU as busy as possible درصد مصرف پردازنده
- **Throughput:** number of processes that complete their execution per time unit بازده
- **Turnaround time**: amount of time to execute a particular process زمان برگشت
- **Waiting time**: amount of time a process has been waiting in the ready queue زمان انتظار
- **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) زمان پاسخ
- **Fairness**: give each process a fair share of CPU عدالت
- **Overhead:** computation resource required for implementing the scheduling algorithm سربار

Scheduling Algorithm Optimization Criteria

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

First- Come, First-Served (FCFS) Scheduling

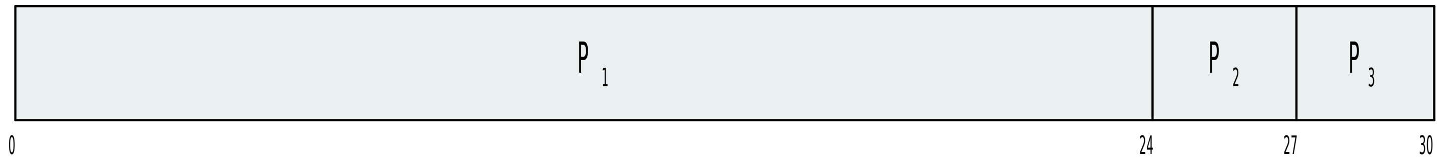
Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3

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

FCFS Scheduling (Cont.)

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order:

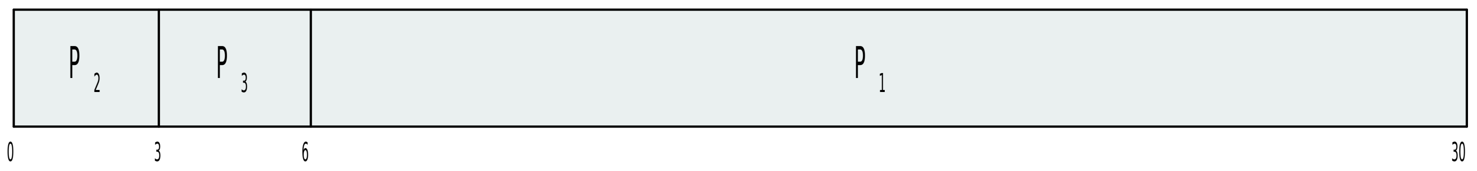
P_2 , P_3 , P_1

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

FCFS Scheduling (Example)

پروسس	زمان ورود	زمان پردازش
A	0	3
B	1	3
C	4	3
D	6	2

Waiting_time=(0+2+2+3)/4=7/4

FCFS Properties

- Non-preemptive
- No starvation
- More suitable for short jobs compared to long ones
- The least overhead

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

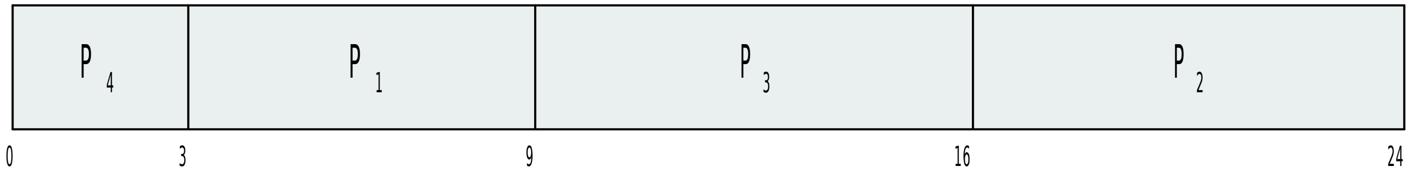
Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

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

SJF Properties

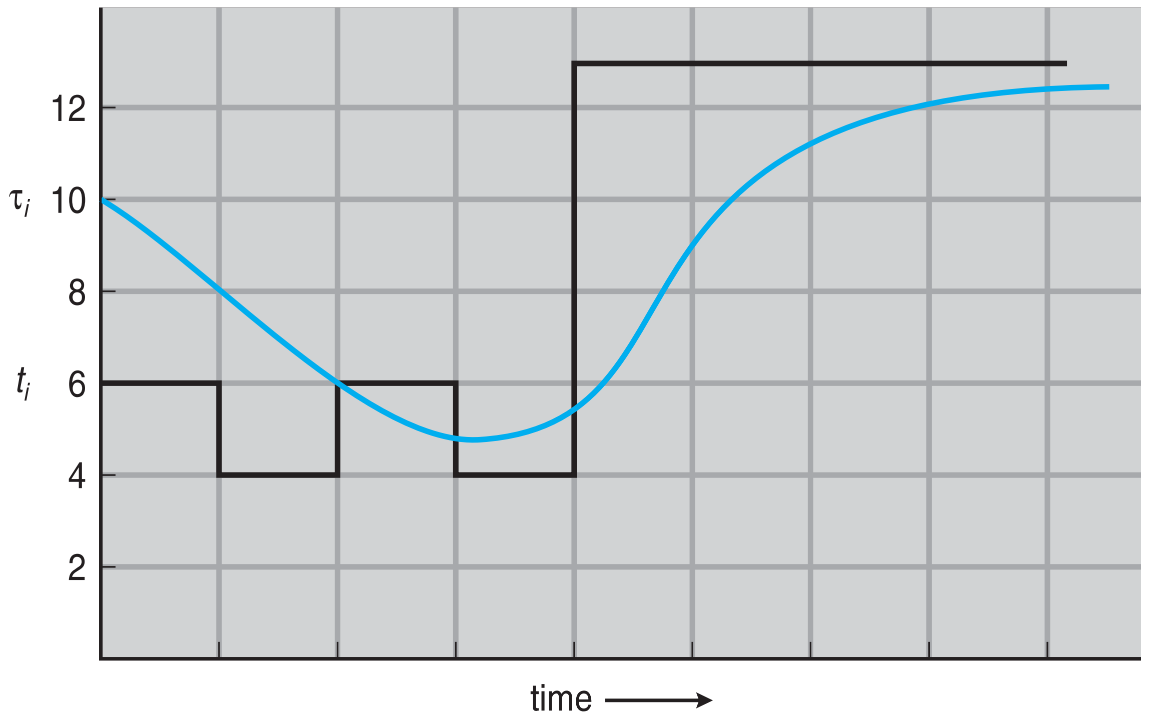
- Non preemptive
- Maybe starvation for long processes
- The least average waiting time

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

  1. $t_n =$ actual length of n^{th} 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

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	. . .

Example of Shortest-remaining-time-first

- Preemptive version of SJF is called **shortest-remaining-time-first**
- Now we add the concepts of varying arrival times and preemption to the analysis

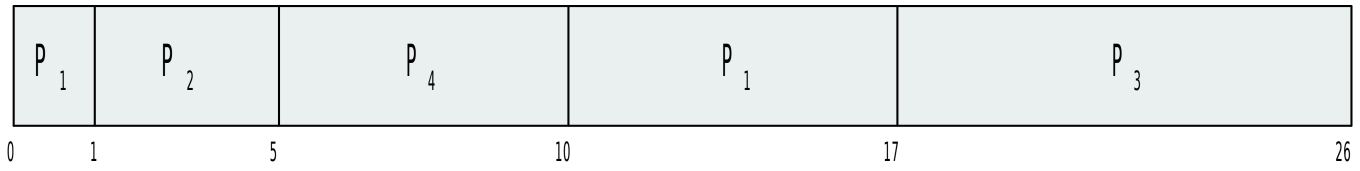
Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Example of Shortest-remaining-time-first

- Preemptive version of SJF is called **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

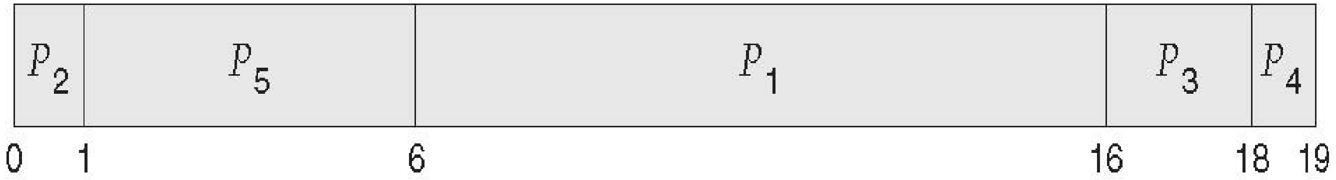
Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem: **Starvation** – low priority processes may never execute
  - When shut downing IBM 7094 at MIT in 1973, there is a waiting job from 1967!
- 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

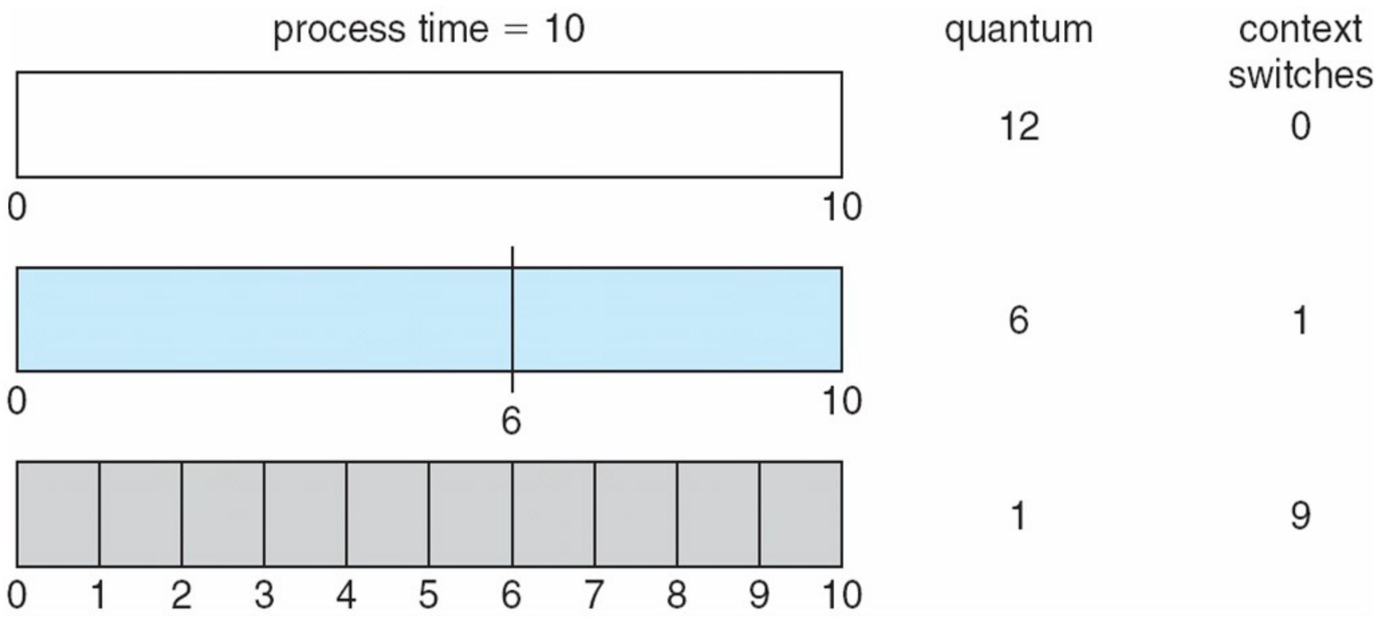


- Average waiting time = 8.2 msec

Round Robin (RR)

- 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.
- **What is the maximum response time?**
  - **No process waits more than (*n*-1)*q* time units.**
- Timer interrupts every quantum to schedule next process
- Performance
  - Large q: FIFO
  - Small q: *q* must be large with respect to context switch, otherwise overhead is too high

Time Quantum and Context Switch Time

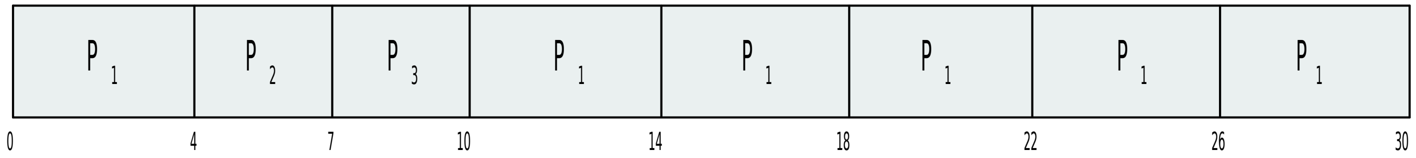


q usually 10ms to 100ms, context switch < 10 usec

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



- Compare average waiting time, turnaround time and response time with SJF
  - Typically, higher average waiting time and turnaround than SJF, **but better *response***

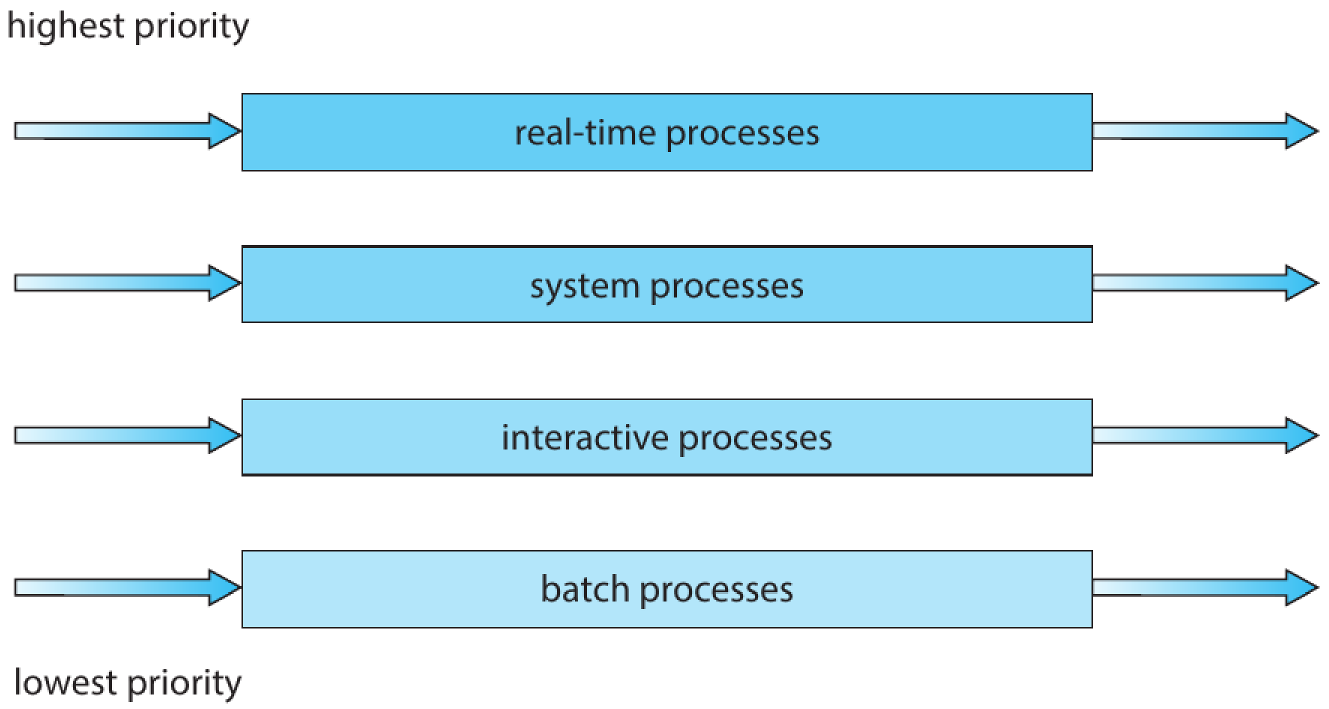
RR Properties

- Preemptive
- No starvation
- Suitable for time sharing systems
- Low throughput with short quantum
- Same as FCFS with a quantum larger than CPU bursts

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

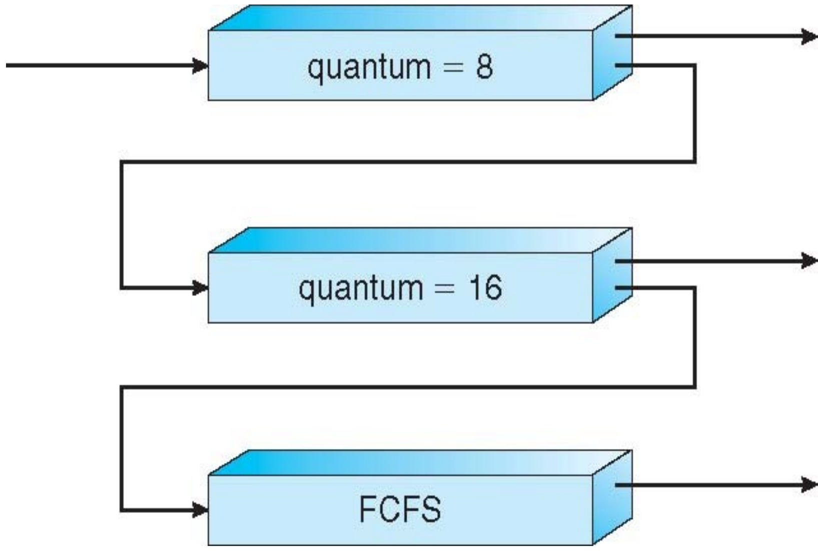


Multilevel Feedback Queue

- 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



Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Real-time characteristics:
  - Periodic (p value)
  - Processing time (t)
  - Deadline (d)
- Some algorithms:
  - Priority-based

Linux command to check process priority and scheduling
chrt

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- **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**
  - **hard affinity**

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
  - **Load balancing** attempts to keep workload evenly distributed
  - **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
  - **Pull migration** – idle processors pulls waiting task from busy processor
- What is the problem of load balancing according to affinity?

Linux command to check process affinity
taskset

Thread Scheduling

- Distinction between user-level and kernel-level threads
- **When threads supported, threads scheduled, not processes**
- 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

Pthread Scheduling

- 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
- Scheduling policies
  - FIFO
  - RR
  - OTHER

Linux scheduling

- pthread_attr_init(&attr)
- Config Scope
  - pthread_attr_getscope(&attr,&scope)

    pthread_attr_setscope(&attr,PTHREAD_SCOPE_PROCESS)
- Config scheduling algorithm
  - pthread_attr_getschedpolicy(&attr,&policy)

    pthread_attr_setschedpolicy(&attr,SCHED_FIFO)
- Shell commands
  - chrt

Linux Scheduling in Version 2.6.23 +

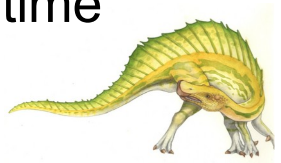
- Features:
  - constant order *O*(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
    - Real-time range from 0 to 99 and time sharing range from 100 to 140
  - **Higher priority gets larger q**

Linux Scheduling in Version 2.6.23 +

- Scheduling in the Linux system is based on scheduling classes
  - Each class is assigned a specific priority
  - To decide which task to run next, the scheduler selects the highest-priority task belonging to the highest-priority scheduling class
- By using different scheduling classes, the kernel can accommodate different scheduling algorithms based on the needs of the system and its processes.
  - Ex. The scheduling criteria for a Linux server, may be different from those for a mobile device running Linux.

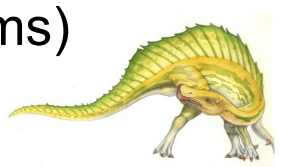
Completely Fair Scheduler (CFS)

- Standard Linux kernels implement two scheduling classes
  - a default scheduling class using the CFS scheduling algorithm
  - a real-time scheduling class
  - New scheduling classes can, of course, be added
- ***Completely Fair Scheduler*** (CFS)
  - assigns a fair proportion of CPU processing time to each task (the quantum value)
  - **Scheduling decision:** CFS will pick the process with the lowest vruntime to run next
  - **Scheduling quantum (Time Slice):** Tasks with lower nice values receive a higher proportion of CPU processing time



CFS: Nice Value

- Lower nice value indicates a higher relative priority
- Tasks with lower nice values receive a higher proportion of CPU processing time than tasks with higher nice values.
  - If a task increases its nice value from, say, 0 to +10, it is being nice to other tasks in the system by lowering its relative priority (allow other tasks to scheduled earlier and for longer quantum)
- sched_latency: (Targeted latency) : an interval of time during which every runnable task should run at least once.
  - Proportions of CPU time are allocated from the value of sched latency
  - But what if there are “too many” processes running? Wouldn’t that lead to too small of a time slice, and thus too many context switches?
    - Yes! Solution is considering min granularity (Ex: 6ms)



CFS: Virtual runtime

- CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
  - For runtime= 200, what is **virtual runtime** for normal, high, low priority?
- To decide next task to run, scheduler picks task with lowest virtual run time
- Which one has higher priority?
  - IO bound or CPU bound process?

Calculating time slice and vruntime

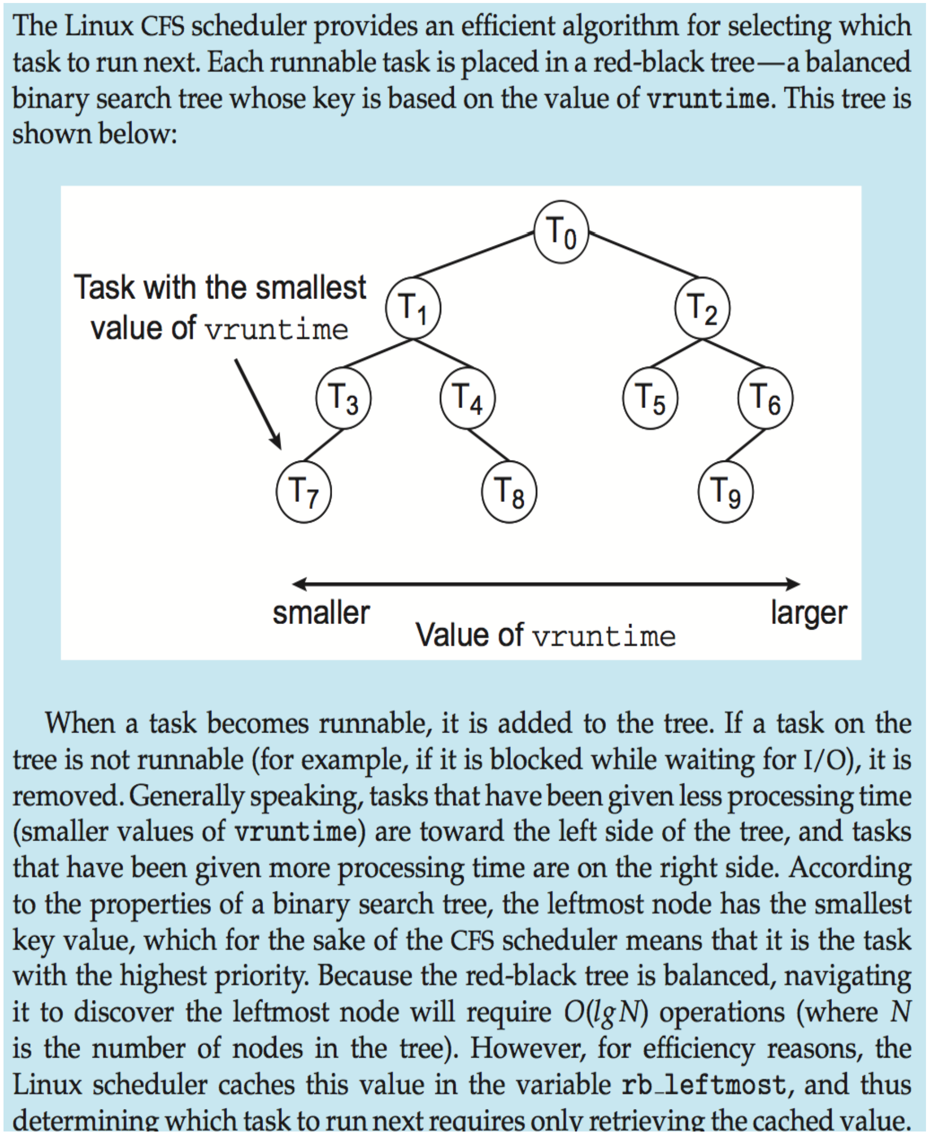
- CFS maps the nice value of each process to a weigh

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

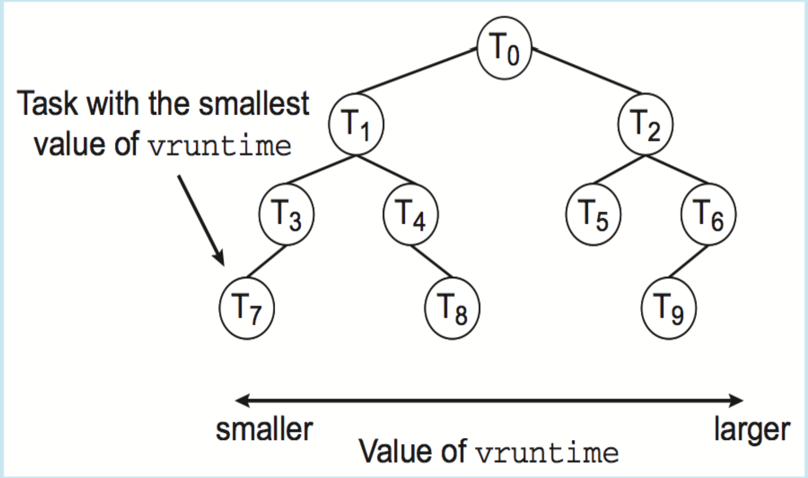
$$\mathtt{time\_slice}_k = \frac{\mathtt{weight}_k}{\sum_{n=0}^{n-1} \mathtt{weight}_i} \cdot \mathtt{sched\_latency}$$

$$\mathtt{vruntime}_i = \mathtt{vruntime}_i + \frac{\mathtt{weight}_0}{\mathtt{weight}_i} \cdot \mathtt{runtime}_i$$

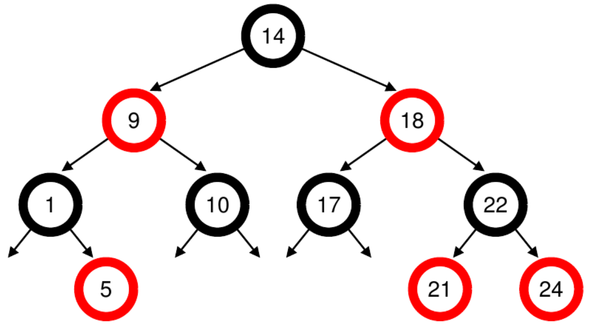
CFS: red-black tree



The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

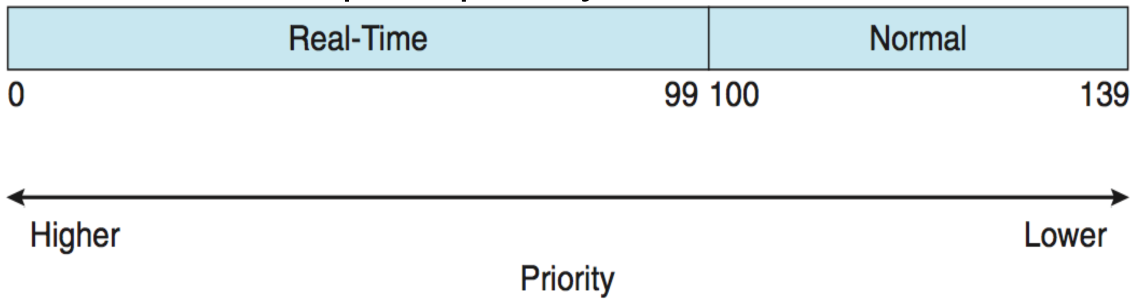


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.



Linux Scheduling: realtime

- Linux also implements real-time scheduling using the POSIX standard
  - SCHED FIFO
  - SCHED RR
- real-time policy runs at a higher priority than normal (non-realtime)
- Real-time plus normal map into global priority scheme
- Nice value of -20 maps to global priority 100
- Nice value of +19 maps to priority 139



Top command represents PR and NI values for processes
for normal processes (sched_other): PR = 20 + NI
(NI is nice and ranges from -20 to 19)
for real time processes (sched_fifo or sched_rr): PR = - 1 – real_time_priority
(real_time_priority ranges from 1 to 99)
Try chrt -m

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
- **Queueing Models**
- **Simulations**
- **Implementation**