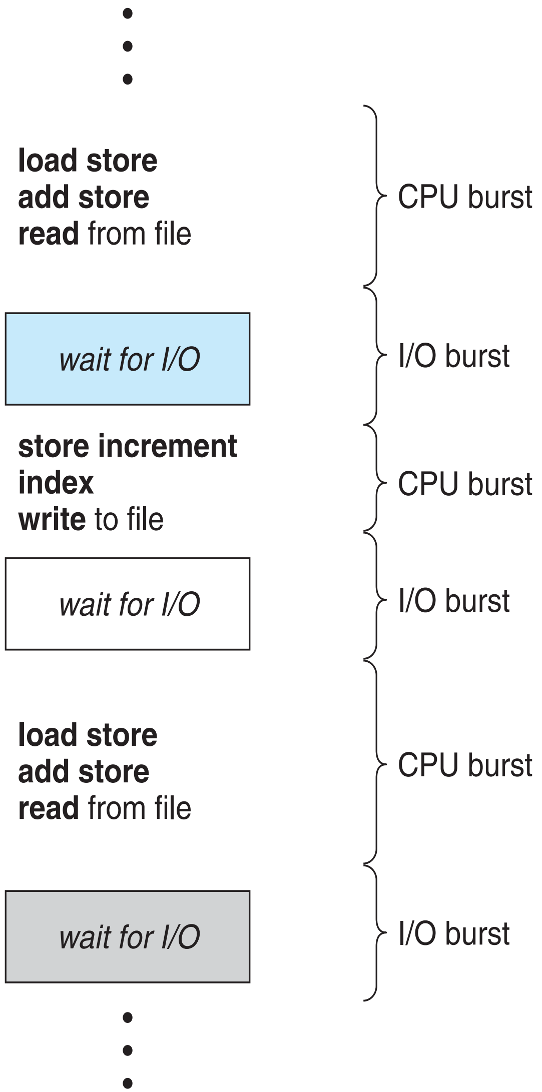
Operating System

Ch05: CPU Scheduling

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

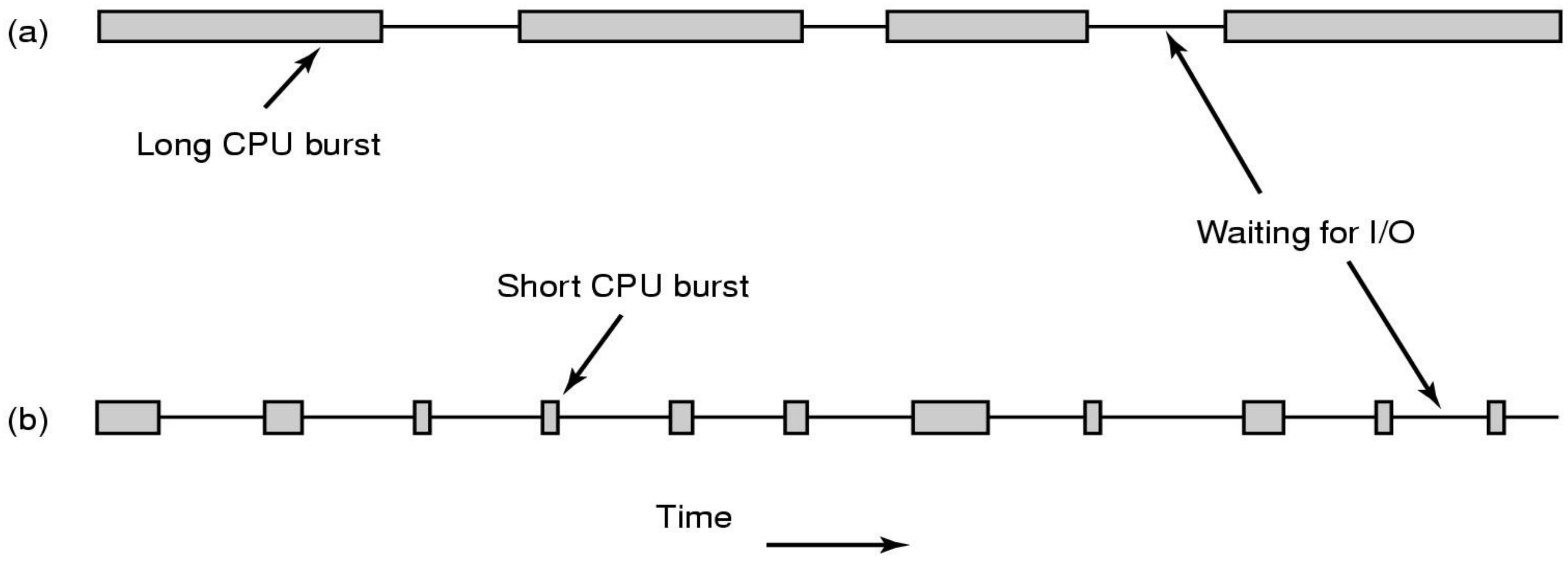
Process Execution

- Alternating sequence of
  - ✓ CPU burst & I/O burst

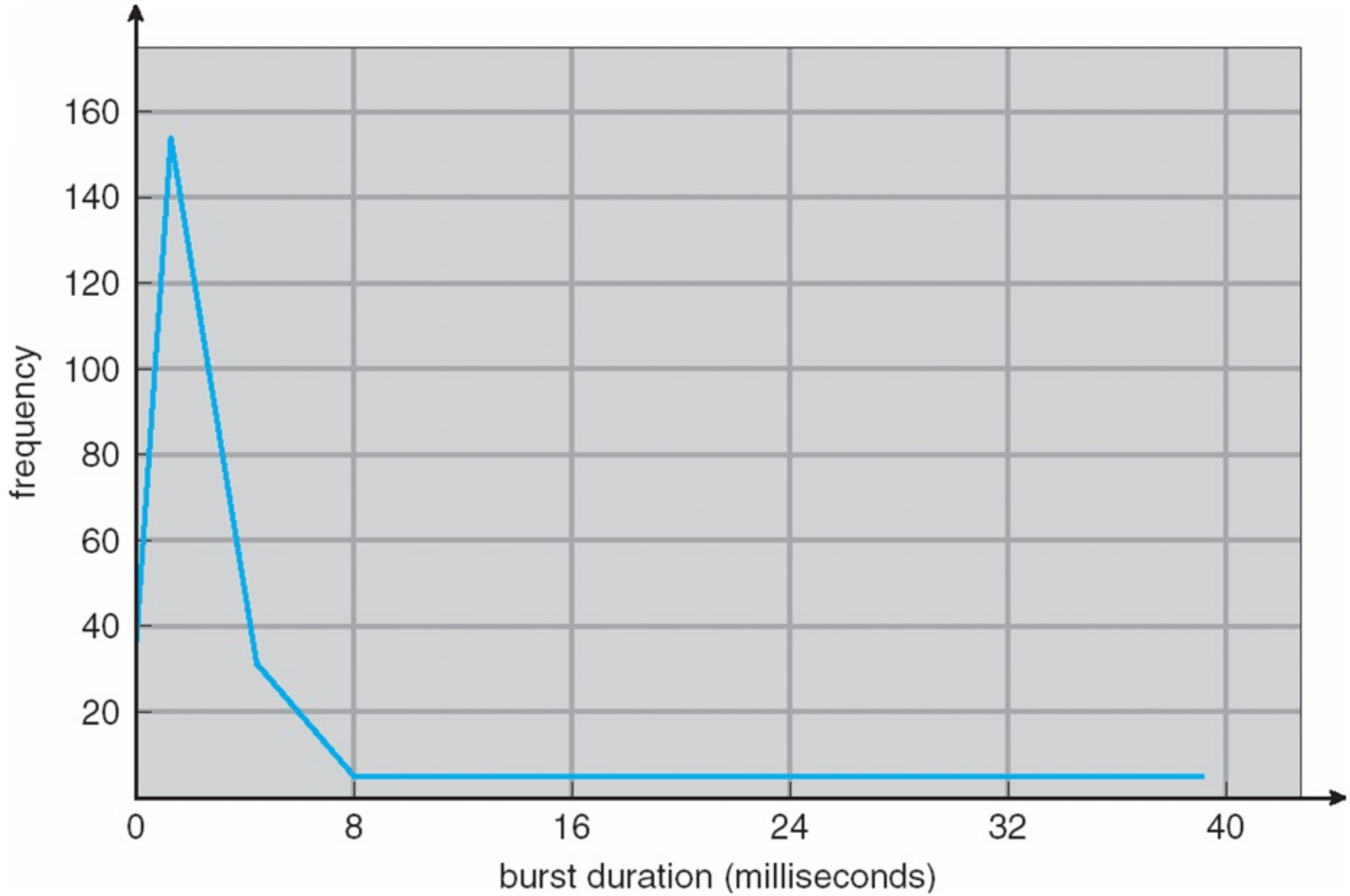


CPU burst vs. I/O burst

- (a) A CPU-bound process
- (b) An I/O-bound process

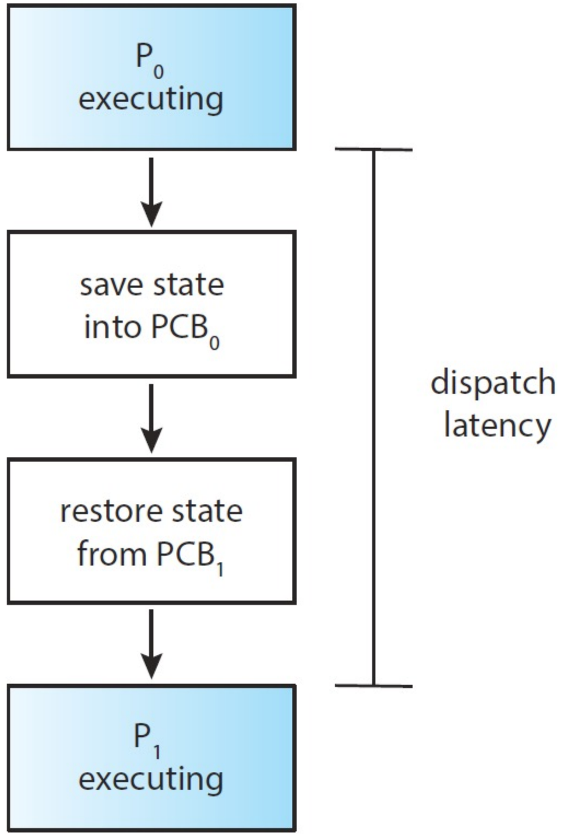


Histogram of CPU-burst Times



Dispatcher

- Dispatch
  - ✓ 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
- Scheduling vs. Dispatch



Preemptive vs. Non-preemptive

- Non-preemptive scheduling
  - ✓ The scheduler waits for the running job to explicitly (voluntarily) block
  - ✓ Scheduling takes place only when
    - ➢ A process switches from running to waiting state
    - ➢ A process terminates

- Preemptive scheduling
  - ✓ The scheduler can interrupt a job and force a context switch
  - ✓ What happens
    - ➢ If a process is preempted in the midst of updating the shared data?
    - ➢ If a process in system call is preempted?

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
- 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)

Optimization Criteria

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

Scheduling Goals

- All systems
  - ✓ Fairness: giving each process a fair share of the CPU
  - ✓ Balance: keeping all parts of the system busy

- Batch systems
  - ✓ Throughput: maximize jobs per hour
  - ✓ Turnaround time: minimize time between submission and termination
  - ✓ CPU utilization: keep the CPU busy all the time

Scheduling Goals

- Interactive systems
  - ✓ Response time: minimize average time spent on ready queue
  - ✓ Waiting time: minimize average time spent on wait queue
  - ✓ Proportionality: meet users' expectations

- Real-time systems
  - ✓ Meeting deadlines: avoid losing data
  - ✓ Predictability: avoid quality degradation in multimedia systems

Scheduling Non-goals

- Starvation
  - ✓ A situation where a process is prevented from making progress because another process has the resource it requires.
    - ➢ Resource could be the CPU or a lock
  - ✓ A poor scheduling policy can cause starvation
    - ➢ If a high-priority process always prevents a low-priority process from running on the CPU
  - ✓ Synchronization can also cause starvation
    - ➢ One thread always beats another when acquiring a lock
    - ➢ Constant supply of readers always blocks out writers

Scheduling Algorithms

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- SRTF (Shortest Remaining Time First)
- Priority
- Round Robin (RR)
- Multi-Level Queue
- Multi-Level Feedback Queue (MLFQ)

FCFS Scheduling (FIFO)

- First-Come, First-Served
  - ✓ Jobs are scheduled in order that they arrive
  - ✓ “Real-world” scheduling of people in lines
    - ➢ e.g. supermarket, bank tellers, McDonalds, etc.
  - ✓ Typically, non-preemptive
  - ✓ Jobs are treated equally: no starvation

- Problems
  - ✓ Average waiting time can be large if small jobs wait behind long ones
    - ➢ Basket vs. cart
  - ✓ May lead to poor overlap of I/O and CPU

FCFS Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Gantt chart for FCFS scheduling

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

FCFS Scheduling

- Suppose that the processes arrive in the order

  P_2 , P_3 , P_1 .

- Gantt chart for FCFS scheduling

  - ✓ Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - ✓ Average waiting time: $(6 + 0 + 3)/3 = 3$

- *Convoy effect*
  - ✓ short process behind long process

SJF Scheduling

- Shortest Job First
  - ✓ Choose the job with the smallest expected CPU burst
  - ✓ Can prove that SJF has optimal min. average waiting time
    - ➢ Only when all jobs are available simultaneously
  - ✓ Non-preemptive

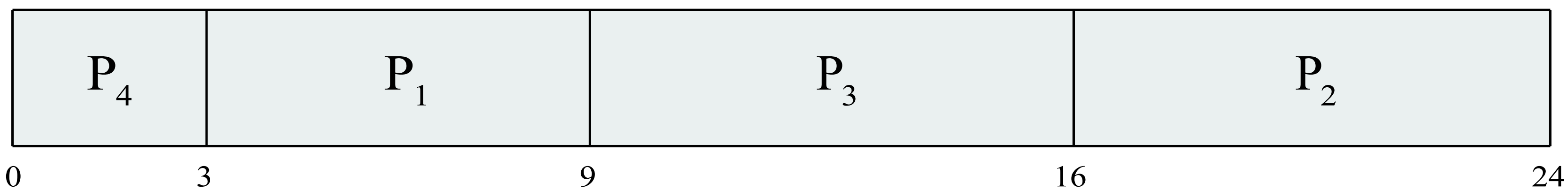
- Problems
  - ✓ Impossible to know size of future CPU burst
  - ✓ Can you make a reasonable guess?
  - ✓ Can potentially starve

- SRTF (Shortest Remaining Time First)
  - ✓ Preemptive
  - ✓ Preemptive SJF

SJF Scheduling

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- Gantt chart for SJF scheduling

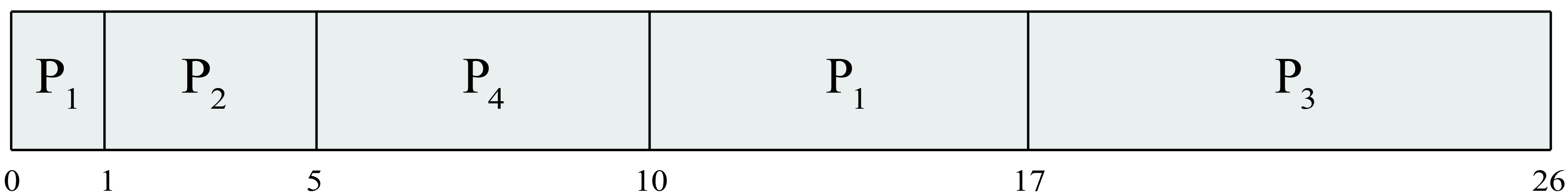


✓ Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

SRTF Scheduling

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

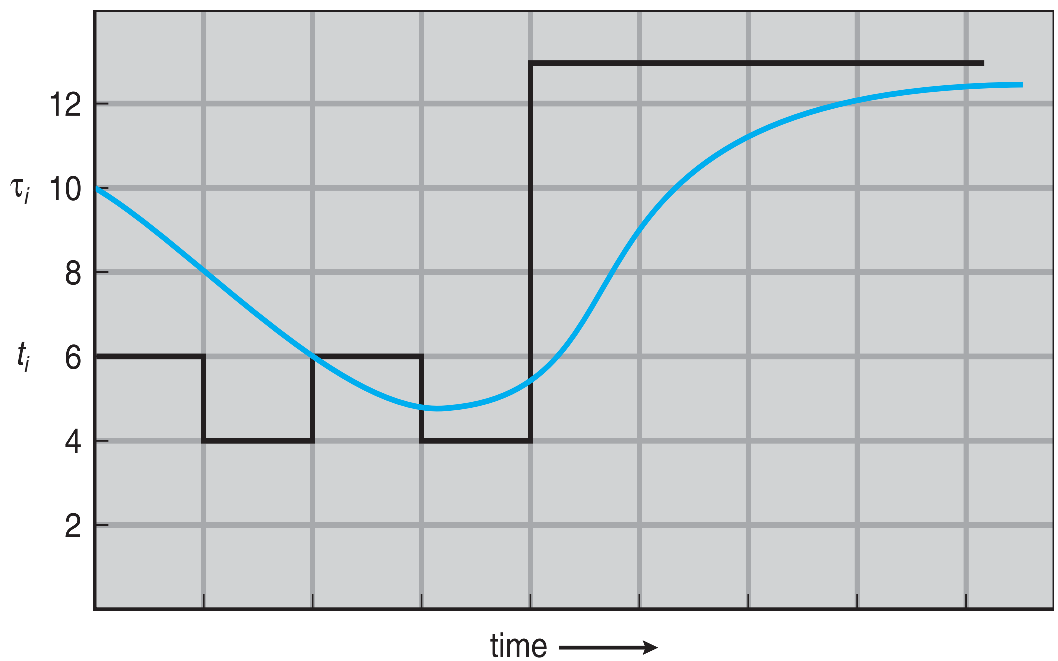
- SRTF (= preemptive SJF)



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

SJF/SRTF Scheduling

- 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	…

Priority Scheduling

- A priority number (integer) is associated with each process

- The CPU is allocated to the process with the highest priority
  - ✓ Preemptive
  - ✓ Non-preemptive

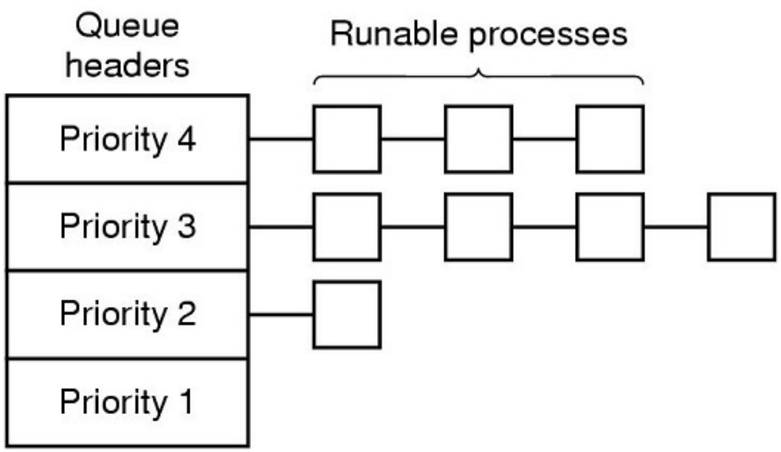
- SJF is a priority scheduling where priority is the predicted next CPU burst time

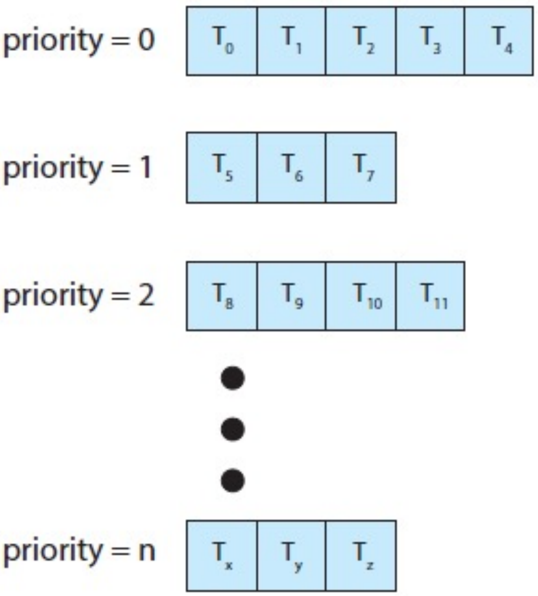
- Problem ≡ Starvation (or Indefinite blocking)
  - ✓ low priority processes may never execute

- Solution ≡ Aging
  - ✓ as time progresses increase the priority of the process

Priority Scheduling

- Abstractly modeled as multiple "priority queues"
  - ✓ Put ready job on Q associated with its priority

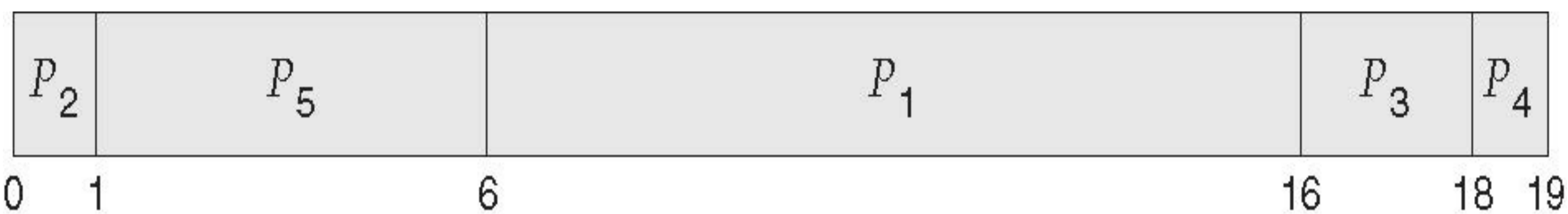




Priority Scheduling

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

- Gantt chart for priority scheduling



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

Round Robin (RR) Scheduling

- Each process gets a small unit of CPU time (*time quantum, q*)
  - ✓ After this time has elapsed, the process is preempted and added to the end of the ready queue

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

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

- q should be large compared to context switch time

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

RR Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Gantt chart for RR scheduling (Time Quantum = 20)

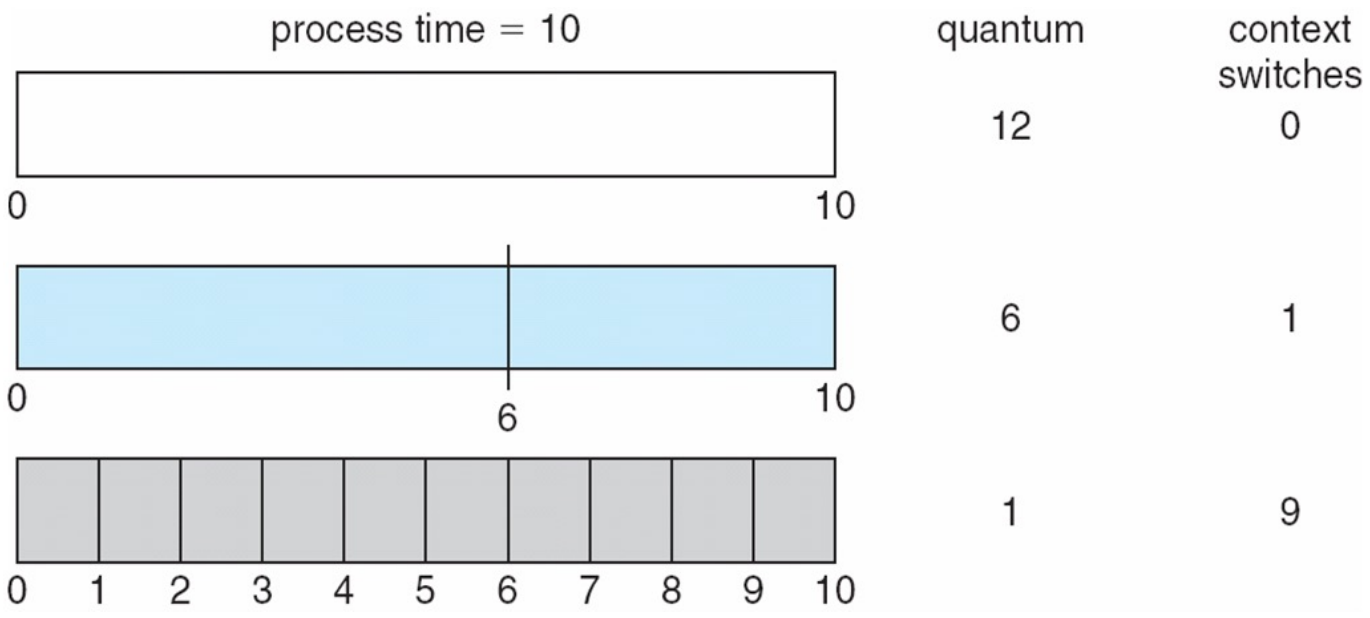
P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

✓ Average waiting time = [(10-4) + 4 + 7] / 3 = 5.67

RR Scheduling

- Time quantum and context switch time



Problems of RR

- What do you set the quantum to be?
  - ✓ quantum $\rightarrow \infty$: FIFO
    quantum $\rightarrow 0$: processor sharing
  - ✓ If small, then context switches are frequent incurring high overhead (CPU utilization drops)
  - ✓ If large, then response time drops
  - ✓ A rule of thumb: 80% of the CPU bursts should be shorter than the time quantum

- Treats all jobs equally
  - ✓ Multiple background jobs?

Combining Algorithms

- Scheduling algorithms can be combined in practice
  - ✓ Have multiple queues
  - ✓ Pick a different algorithm for each queue
  - ✓ Have a mechanism to schedule among queues
  - ✓ And maybe, move processes between queues

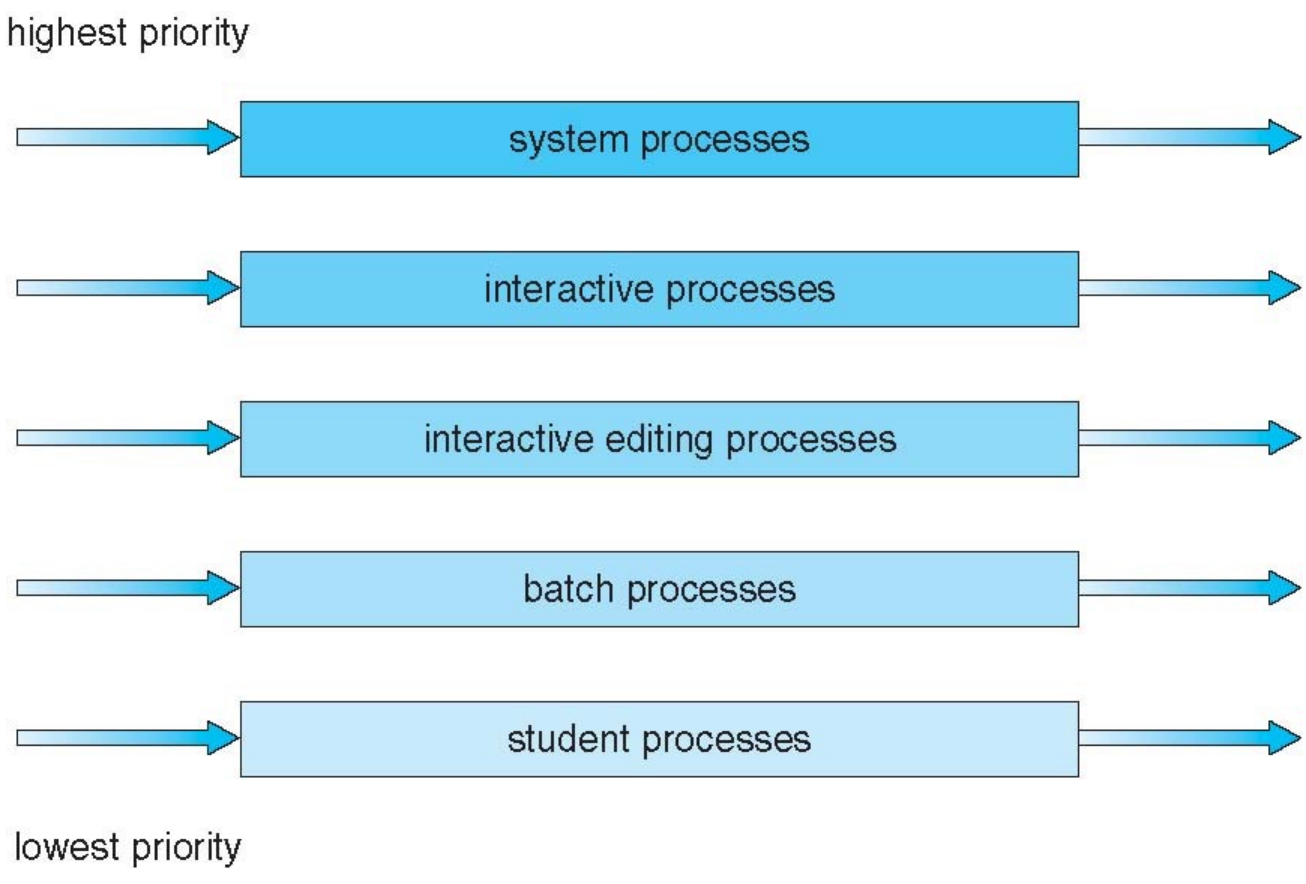
Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues:
  - ✓ foreground (interactive)
  - ✓ background (batch)

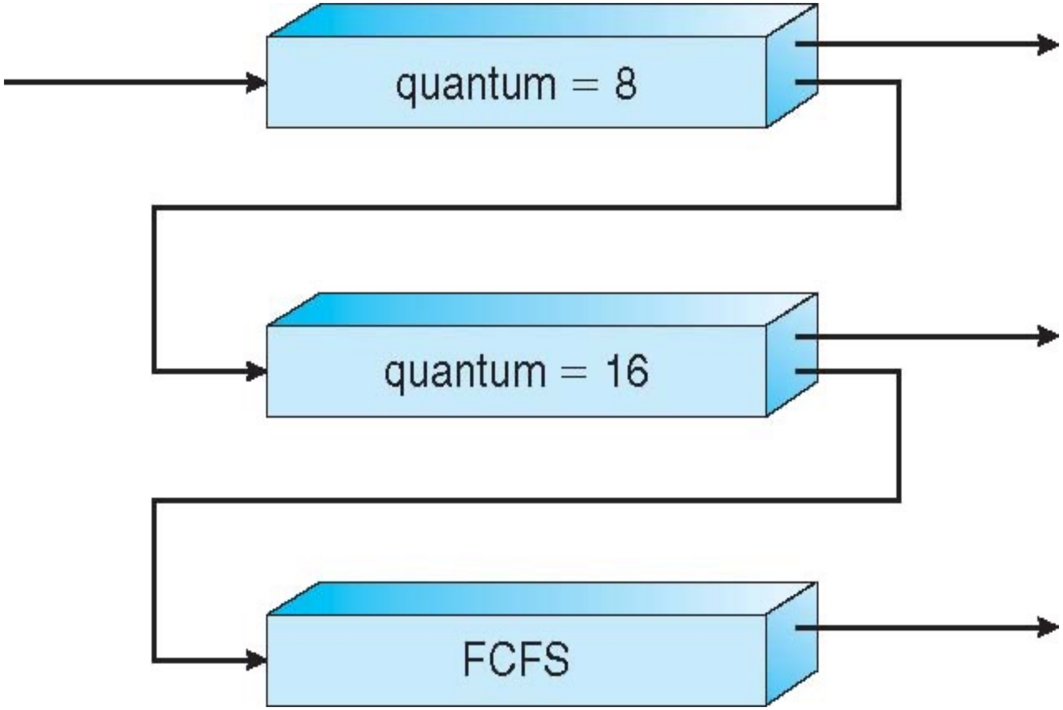
- Each queue has its own scheduling algorithm:
  - ✓ foreground – RR
  - ✓ background – FCFS

- 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 & 20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue Scheduling



Multilevel Feedback Queue Scheduling

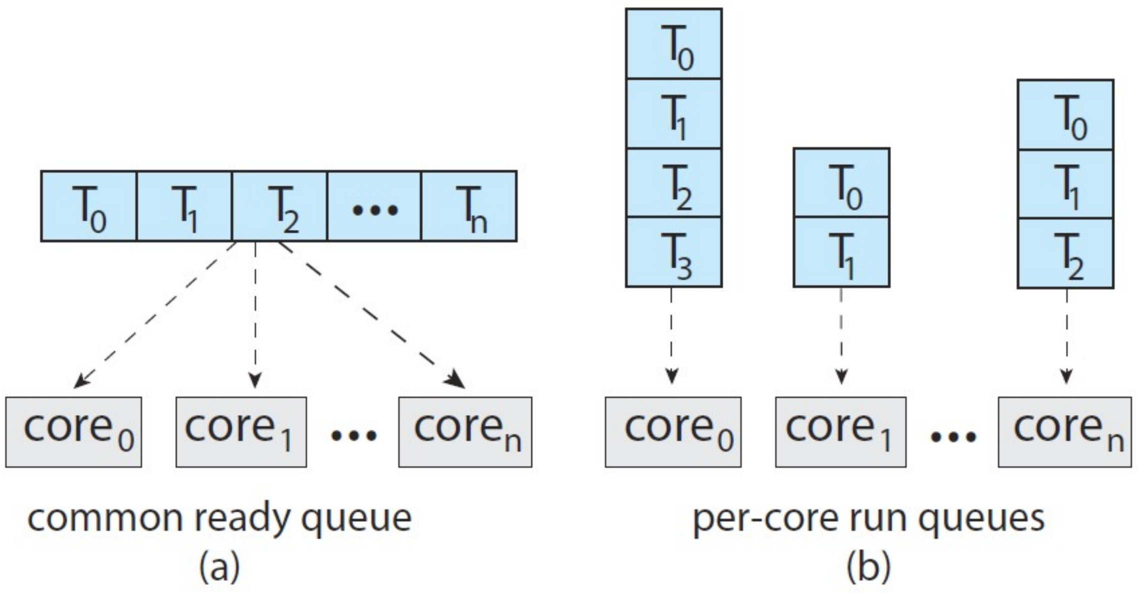
- The canonical UNIX scheduler uses a MLFQ
  - ✓ 3 – 4 classes spanning ~170 priority levels
    - ➢ Timeshare, System, Real-time, Interrupt (Solaris 2)
  - ✓ Priority scheduling across queues, RR within a queue
    - ➢ The process with the highest priority always runs
    - ➢ Processes with the same priority are scheduled RR
  - ✓ Processes dynamically change priority
    - ➢ Increases over time if process blocks before end of quantum
    - ➢ Decreases over time if process uses entire quantum

Multilevel Feedback Queue Scheduling

- Motivation
  - ✓ The idea behind the UNIX scheduler is to reward interactive processes over CPU hogs
  - ✓ Interactive processes typically run using short CPU bursts
    - ➢ They do not finish quantum before waiting for more input
  - ✓ Want to minimize response time
    - ➢ Time from keystroke (putting process on ready queue) to executing the handler (process running)
    - ➢ Don’t want editor to wait until CPU hog finishes quantum
  - ✓ This policy delays execution of CPU-bound jobs

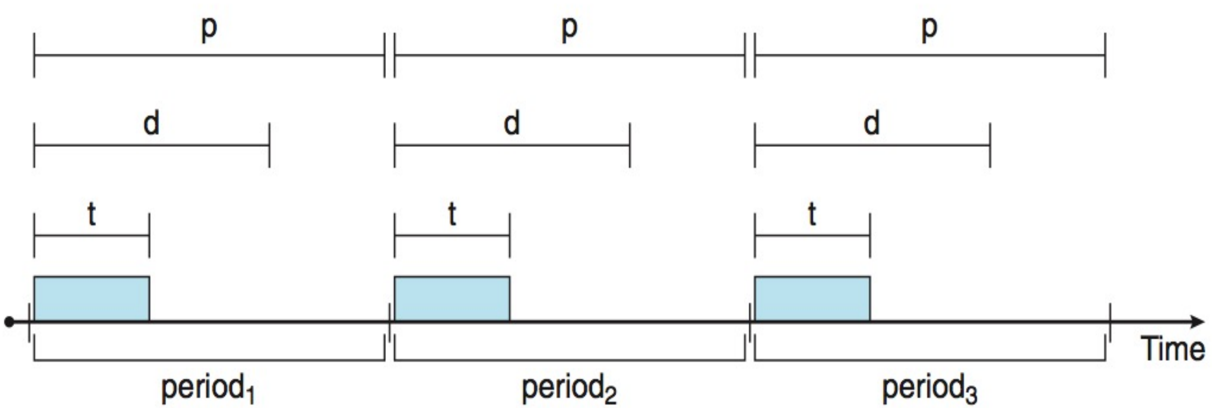
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Symmetric multiprocessing vs. Asymmetric multiprocessing
- Load balancing
  - ✓ Push vs. Pull migration
- Processor affinity
  - ✓ Soft vs. Hard affinity



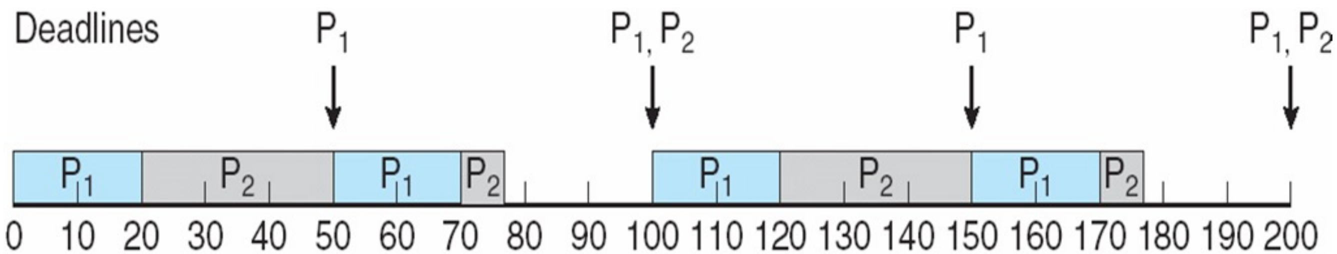
Real-Time Scheduling

- *Hard real-time* systems
  - ✓ required to complete a critical task within a guaranteed amount of time
- *Soft real-time* systems
  - ✓ requires that critical processes receive priority over less fortunate ones
- Periodic task in real-time systems

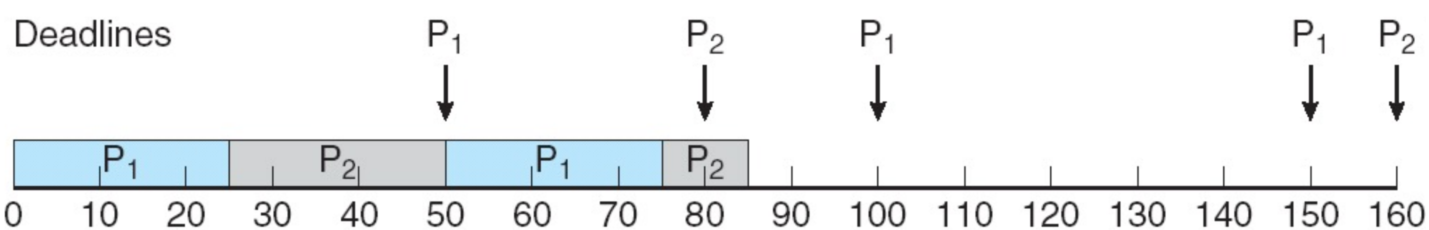


Real-Time Scheduling

- *Static vs. Dynamic priority scheduling*
  - ✓ Static: Rate-Monotonic algorithm
  - ✓ A priority is assigned based on the inverse of its period

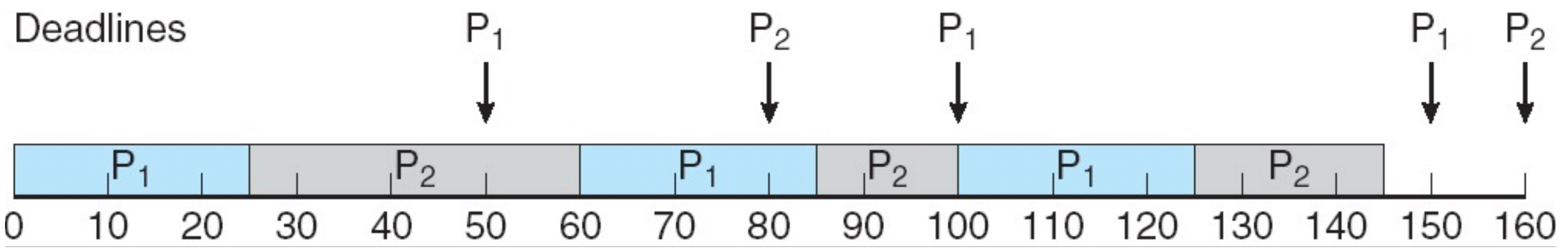


  - ✓ Missed deadline



Real-Time Scheduling

- *Static vs. Dynamic priority scheduling*
  - ✓ Dynamic: EDF (Earliest Deadline First) algorithm
  - ✓ Priorities are assigned according to deadlines:
    - ➢ the earlier the deadline, the higher the priority
    - ➢ the later the deadline, the lower the priority

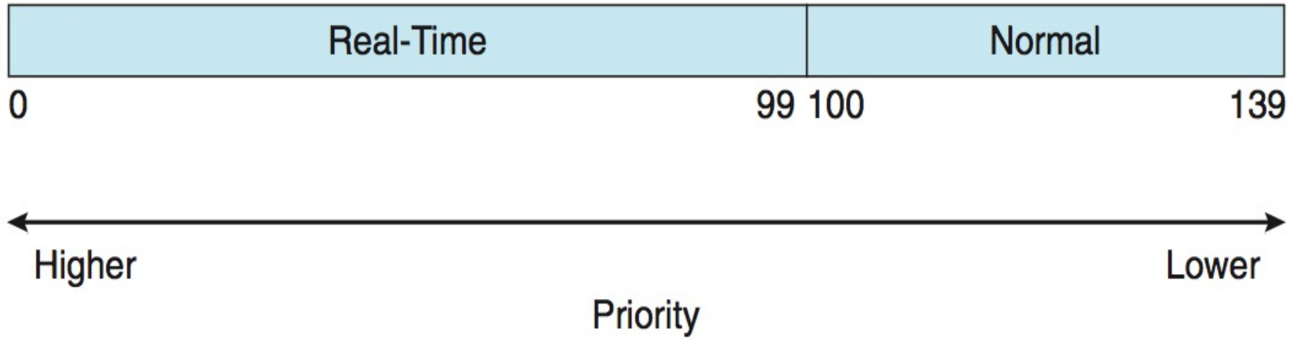


Operating System Examples

- Linux scheduling
- Windows scheduling
- Solaris scheduling

Linux Scheduling

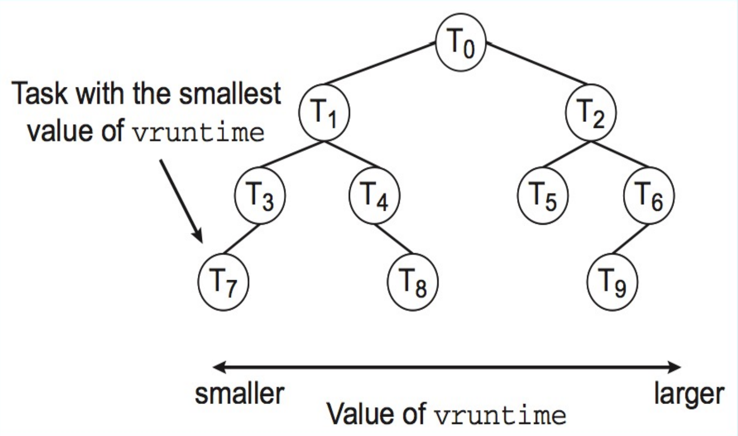
- Real-time scheduling according to POSIX.1b
  - ✓ Real-time tasks have static priorities
- Real-time plus normal map into global priority scheme
  - ✓ Nice value of -20 maps to global priority 100
  - ✓ Nice value of +19 maps to priority 139



Linux Scheduling

- CFS
  - ✓ Completely Fair Scheduling
  - ✓ Quantum calculated based on nice value from -20 to +19
  - ✓ maintains per task virtual run time in variable **`vruntime`**

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:

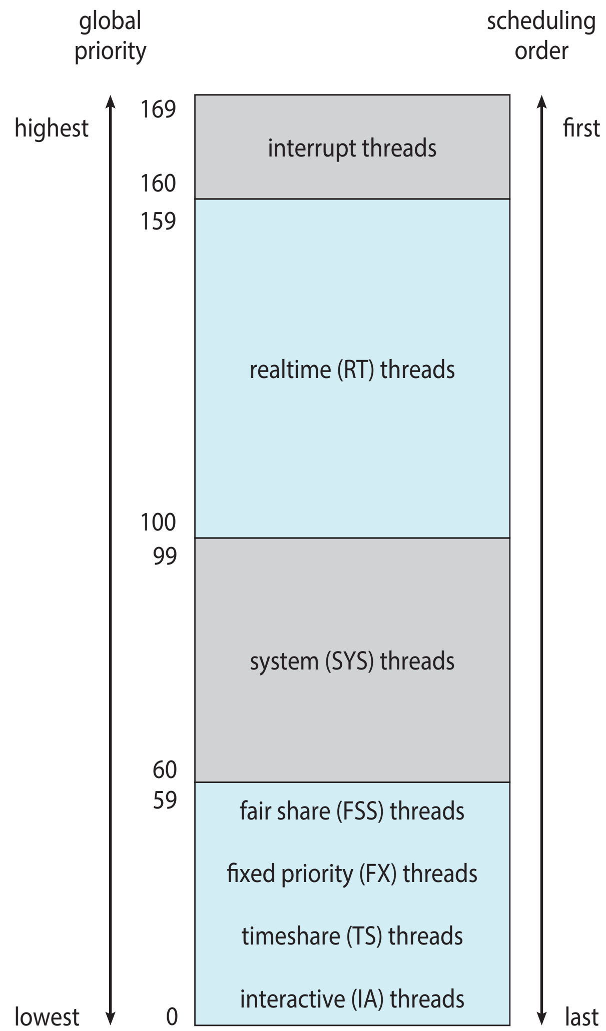


When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and 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 $O(lgN)$ 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.

Windows Scheduling

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Solaris Scheduling

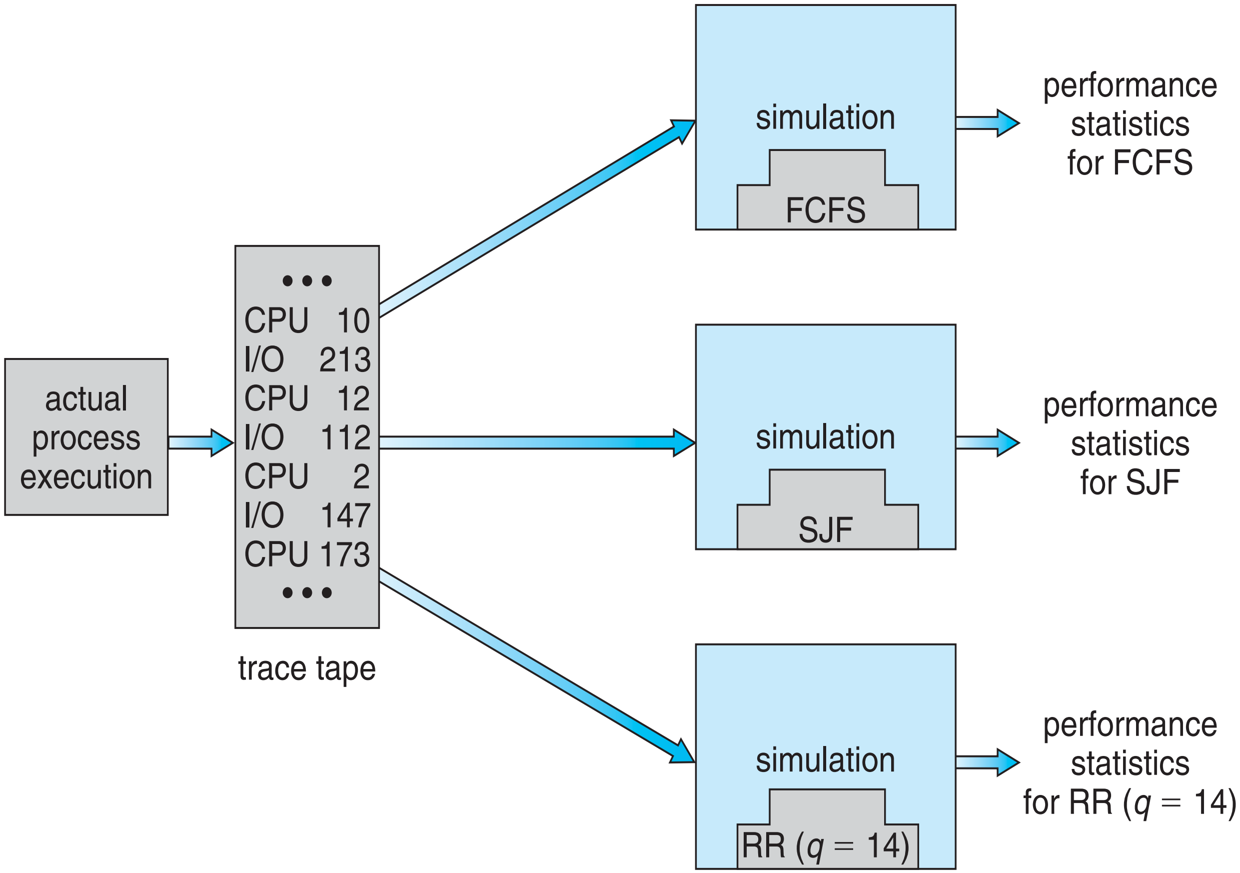


Algorithm Evaluation

- Deterministic modeling
  - ✓ Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Queueing models
  - ✓ Mathematical models used to compute expected system parameters
- Simulation
  - ✓ Algorithmic models which simulate a simplified version of a system using statistical input
  - ✓ Trace tape (or trace data)
  - ✓ *Cf) Emulation*
- Implementation
  - ✓ Direct implementation of the system under test, with appropriate benchmarks

Algorithm Evaluation

- Evaluation of CPU schedulers by simulation



Thank You!
Q&A