

# Chapter 5: CPU Scheduling

Prof. Li-Pin Chang  
CS@NYCU

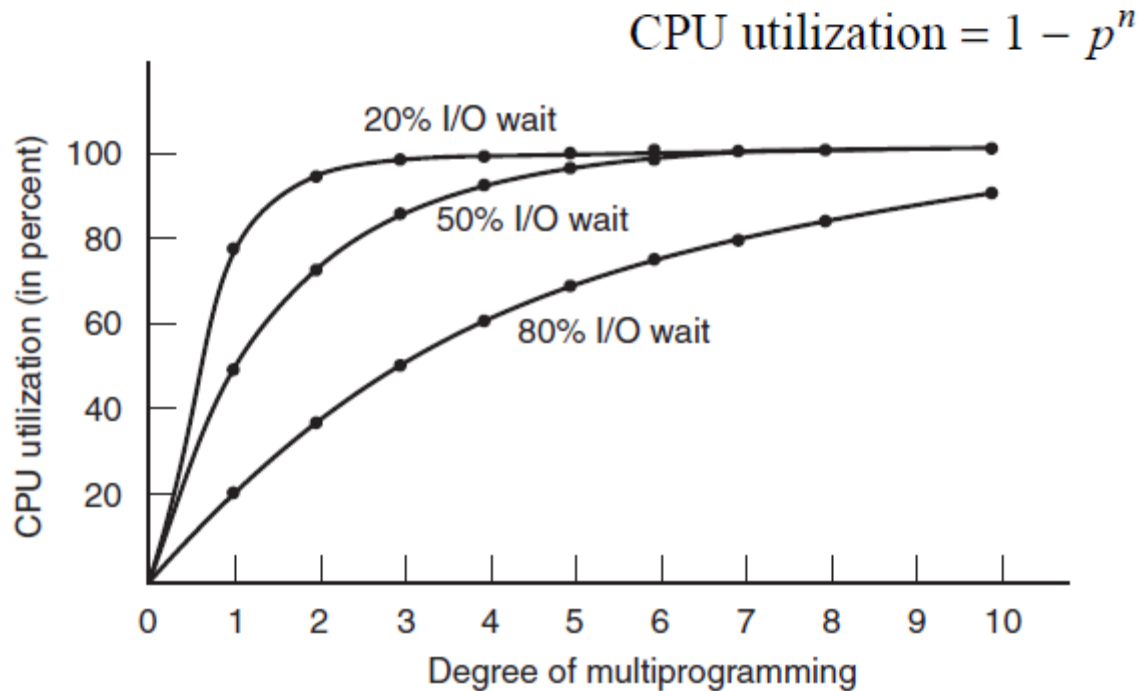
# Chapter 5: Process Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multiple-Processor Scheduling
- Thread Scheduling
- Operating Systems Examples
- Algorithm Evaluation

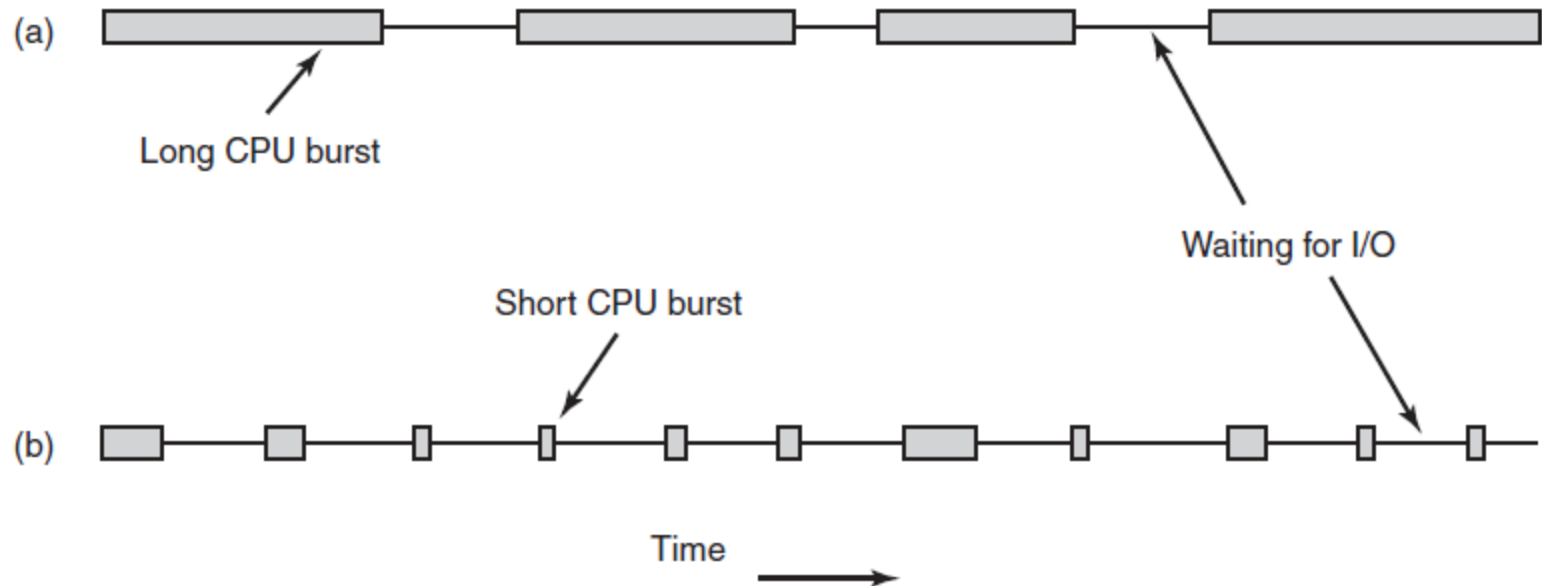
# BASIC CONCEPTS

# 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

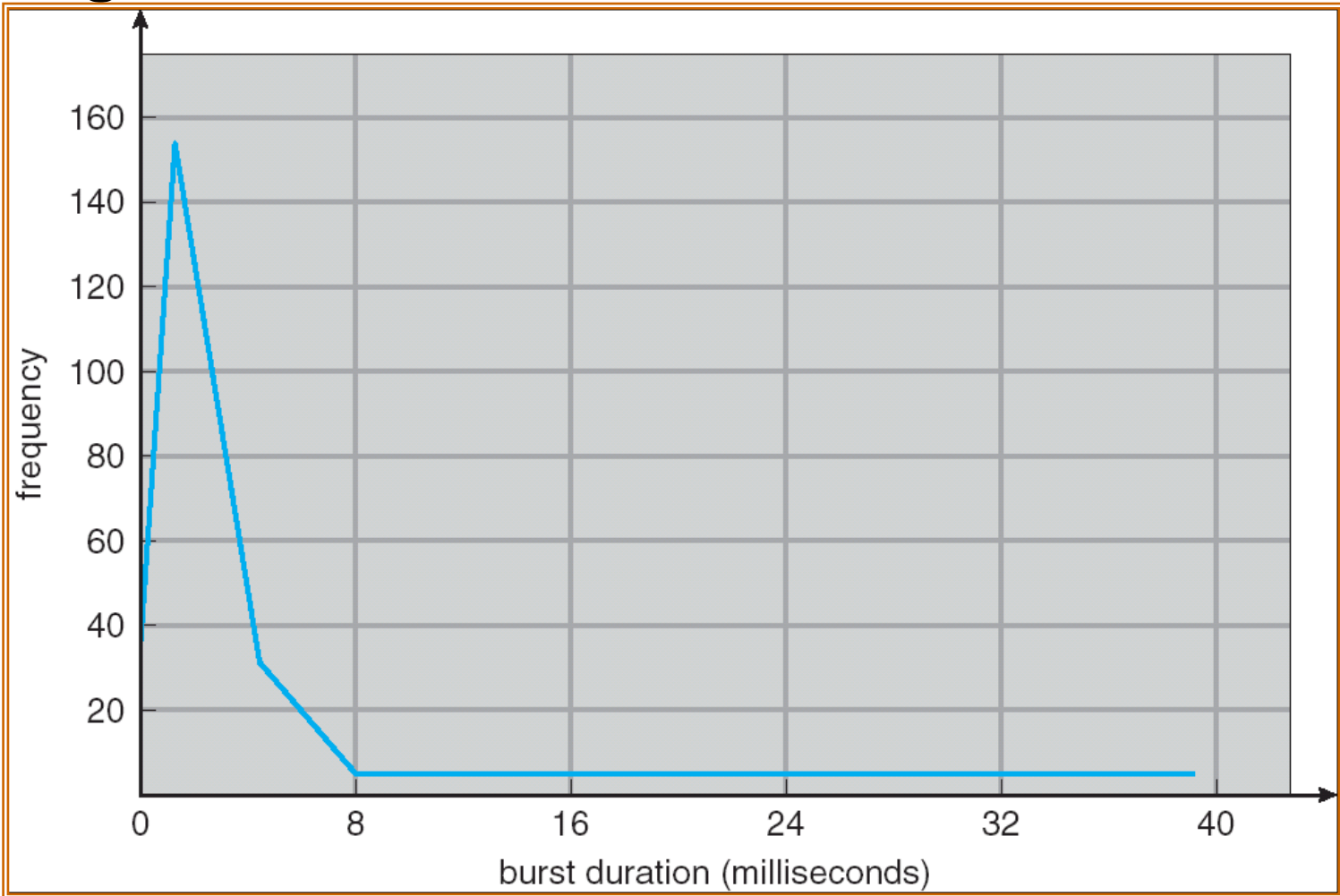


# Process Behavior



Bursts of CPU usage alternate with periods of waiting for I/O.  
(a) A CPU-bound process. (b) An I/O-bound process.

# Histogram of CPU-burst Times



A timesharing system has many short CPU bursts

# CPU 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 **only** 1 and 4 is non-preemptive or cooperative
- All other scheduling is **preemptive**

# Cooperative scheduling

- Easy to implement and requires no extra hardware (e.g., the timer)
- A process may voluntarily gives up the CPU
  - Call Sleep(0) or yield()
  - A blocking call also causes a context switch
- An ill-behaved process can take over the entire system
- Examples
  - Windows 3.1
  - Old versions of Mac OS
  - Sensor OS (e.g., TinyOS 1.0)



# Scheduler Preemptivity

- Preemptive scheduling
  - Higher responsiveness
  - Higher cxtsw overheads
  - Must deal with race conditions
- Cooperative scheduling
  - Easy to implement
  - Poor responsiveness
  - Ill-behaved processes will take over the system

# 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 (the context-switch overhead) : the time it takes for the dispatcher to stop one process and start another running
- Dispatching is a mechanism, while scheduling often refers to a policy

# SCHEDULING CRITERIA

# 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
- (-) Responsiveness (interactivity) – amount of time it takes from when a user sends an input until the system delivers a feedback

# Optimization Criteria

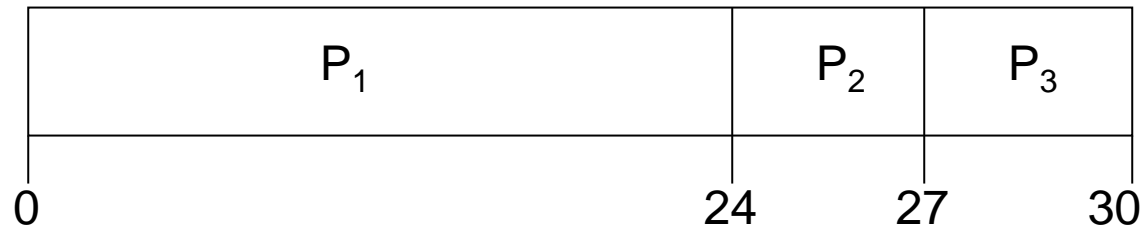
- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time
- There are conflicts among the objectives
  - Throughput vs. waiting time
  - Average performance vs. worst-case performance

# SCHEDULING ALGORITHMS

# First-Come, First-Served (FCFS) Scheduling

<u>Process</u>	<u>Burst Time</u>
$P_1$	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:



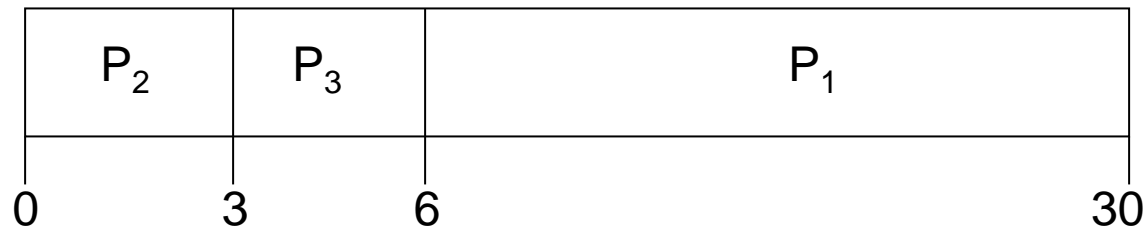
- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

# 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 the previous case



- *Convoy effect* : short process behind long process
  - FCFS is non-preemptible -> poor average waiting time
  - Harmful to I/O-bound processes and result in poor I/O utilization!! (why?)
- Question: Which one(s) of the following could be the performance issue of FCFS?
  - Poor I/O utilization
  - Low CPU utilization
  - Poor responsiveness/interactivity

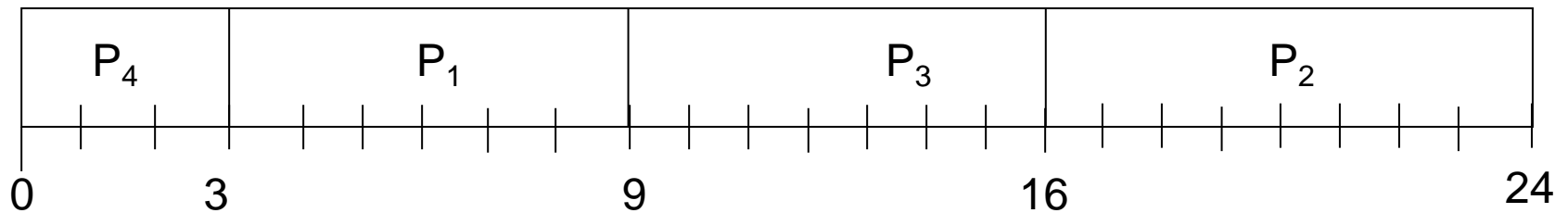
# Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - Non-preemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - Preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, 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 known processes (all ready at time 0)

SJF: All ready at time 0

Process	Arrival Time	Burst Time
P1	0.0	6
P2	0.0	8
P3	0.0	7
P4	0.0	3

SJF (non-preemptive)

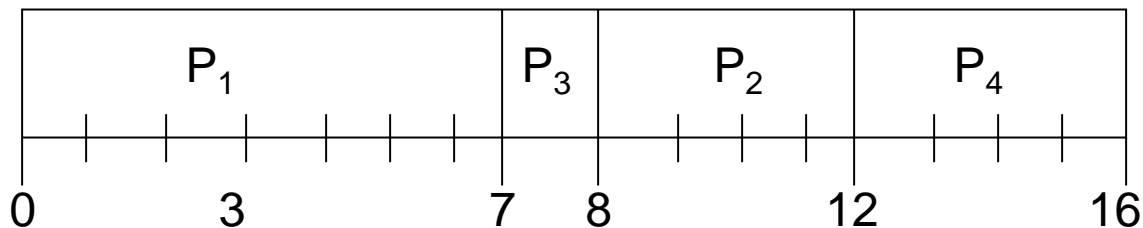


- Average waiting time =  $(0 + 3 + 9 + 16)/4 = 7$
- Compared to FCFS?  $(0+6+14+21)/4=10.25$

# Non-Preemptive SJF with Arbitrary Arrival Times

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (non-preemptive)

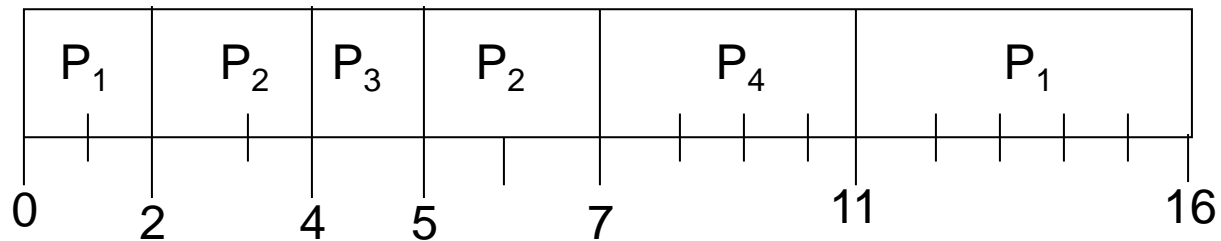


- Average waiting time =  $(0 + 6 + 3 + 7)/4 = 4$

# Preemptive SJF with Arbitrary Arrival Times

Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

- SJF (preemptive)=SRTF



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

- Question: Which one(s) describe SJF?
  - Short average waiting time
  - Small waiting time variation

# Fairness vs. Efficiency

- With SJF, long jobs may be indefinitely delayed if short jobs keep arriving => starvation
- SJF is thus not a fair scheduling algorithm
  - Processes have long CPU bursts may starve
  - By contrast, with FCFS, a process's waiting time is always bounded
- The long-standing dilemma: efficiency vs. fairness
- Some sort of “aging” policy is necessary to prevent starvation in SJF, to be discussed later.

# Determining Length of Next CPU Burst (SJF)

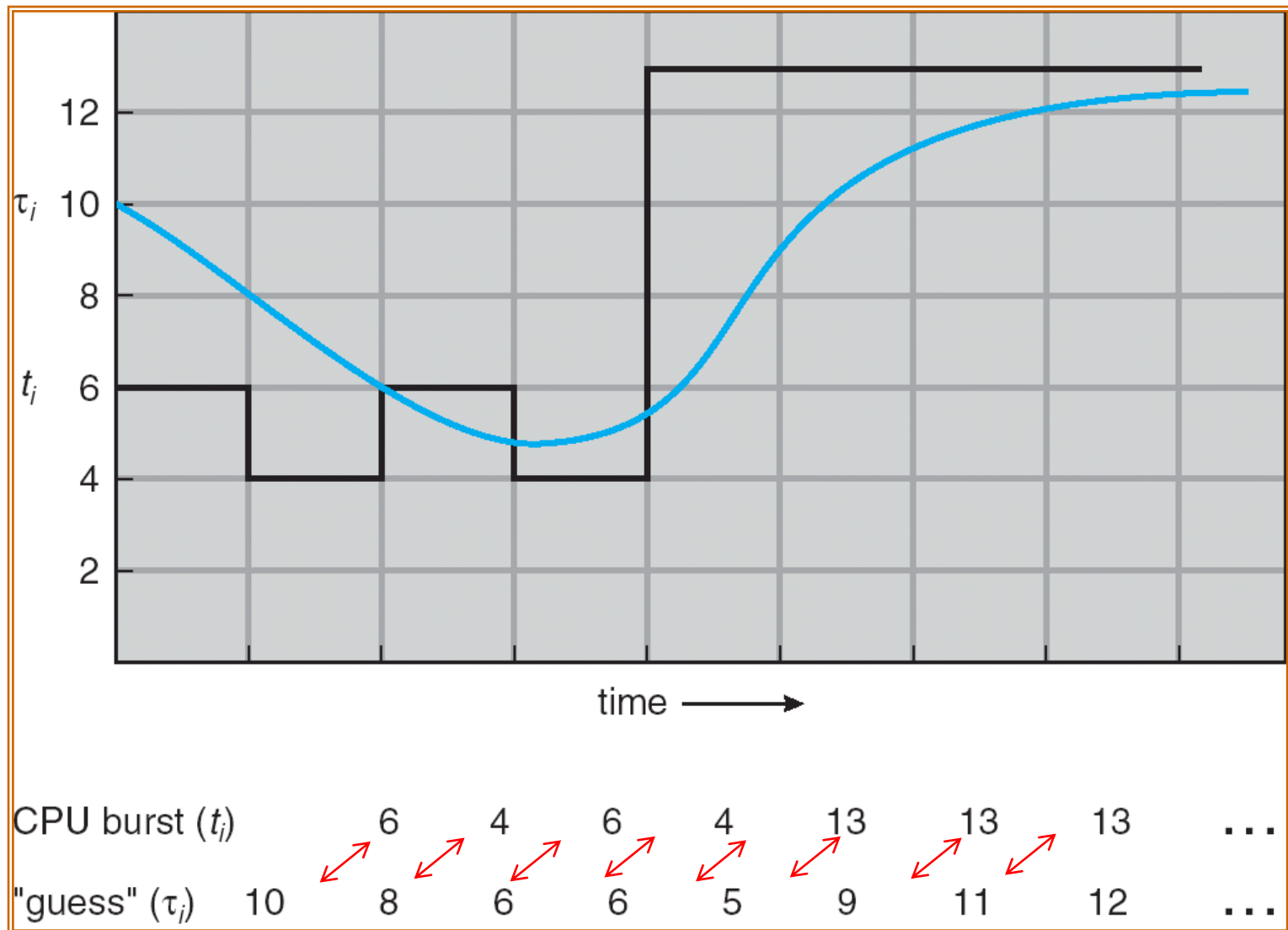
- It is nearly impossible to know the actual job execution time in advance
- Using the exponential moving average method to predict the length of the next CPU burst based on the lengths of previous CPU bursts

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 :

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



# Prediction of the Length of the Next CPU Burst



Alpha=0.5

# Examples of Exponential Moving Average

- $\alpha = 0$ 
  - $\tau_{n+1} = \tau_n$
  - 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:

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

- Since both  $\alpha$  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 or non-preemptive
- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the process (applies to SJF as well)

# Example of Priority Scheduling

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	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

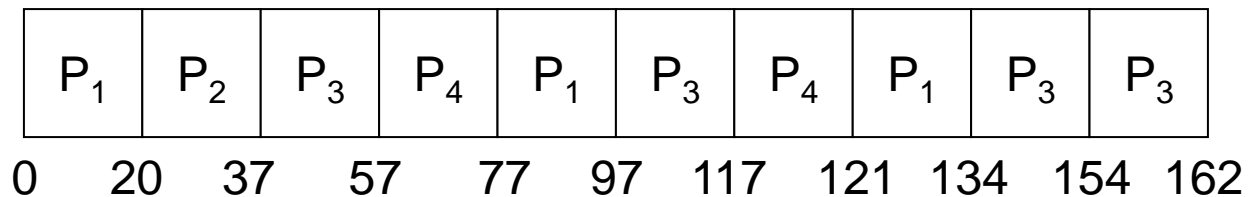
# Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum**), 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 of RR with Time Quantum = 20

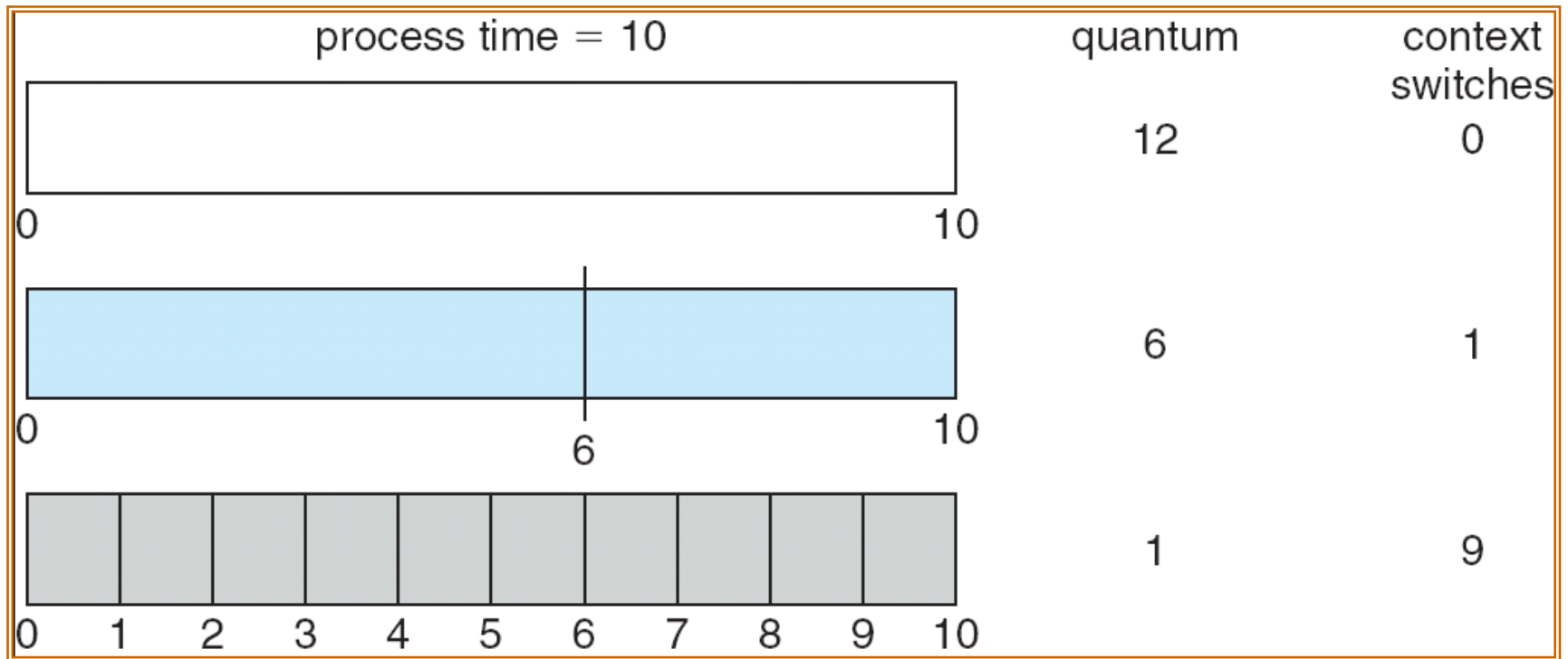
<u>Process</u>	<u>Burst Time</u>
$P_1$	53
$P_2$	17
$P_3$	68
$P_4$	24

- The Gantt chart is:

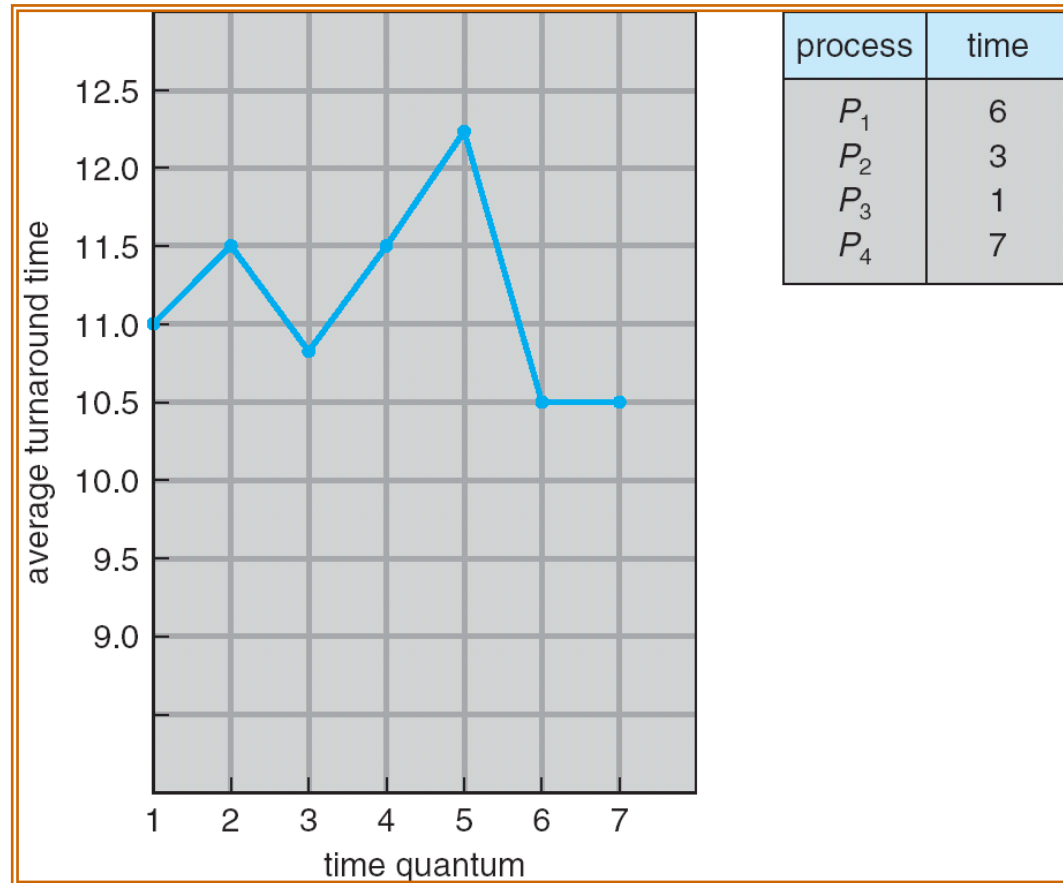


- Processes with RR have longer turnaround time but better response

# Time Quantum and Context Switch Time



# Turnaround Time Varies With The Time Quantum



Large quantum: better turnaround  
Small quantum: better interactivity



## RR vs. FCFS

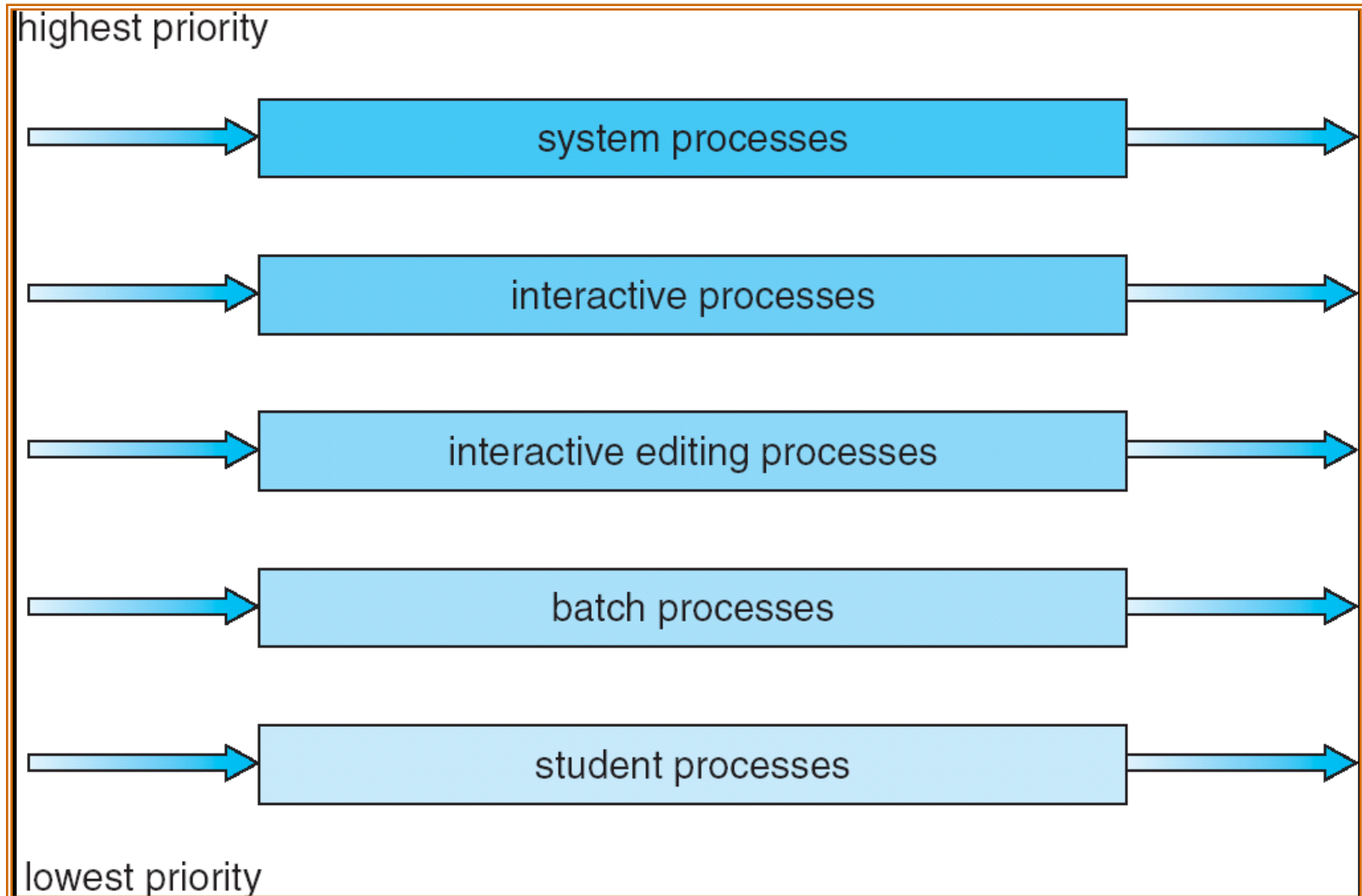
- Processes have better response (interactivity) with RR
  - Timesharing
- Processes (often) have shorter turnaround times with FCFS
  - Consider the case that the time quantum is extremely small. With RR, two processes of the same burst length will complete at the same time

# Multilevel Queue

- 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 and 20% to background in FCFS

(Rumor has it that, when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

# Multilevel Queue Scheduling

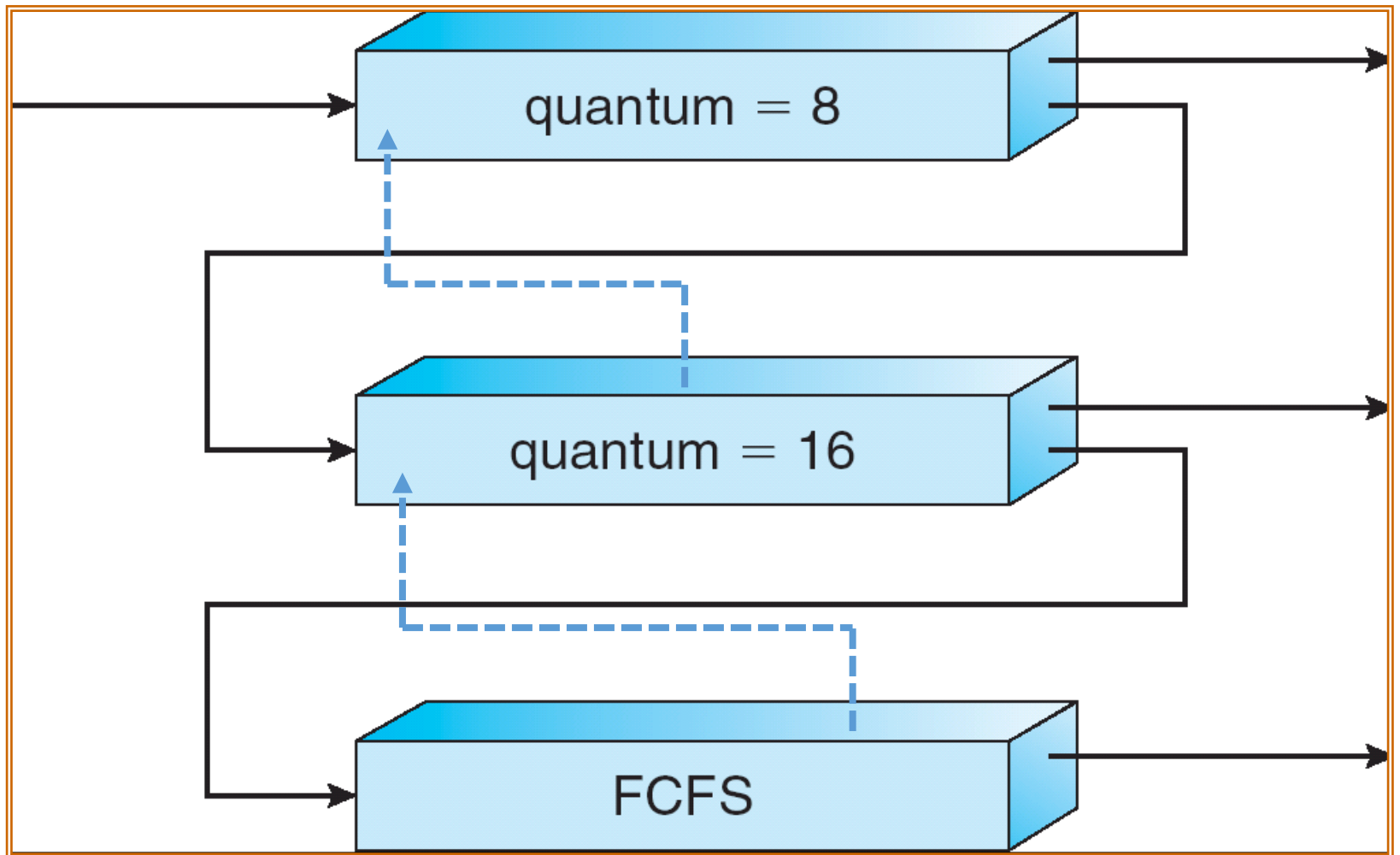


# Multilevel Feedback Queue

- 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 **promote** 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:
  - Q0 – RR with time quantum 8 milliseconds
  - Q1 – RR time quantum 16 milliseconds
  - Q2 – FCFS
- Scheduling
  - A new job enters queue Q0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not release the CPU in 8 milliseconds, the job is preempted and moved to queue Q1
  - At Q1 job is again served FCFS and receives 16 additional milliseconds. If it still does not release the CPU, it is preempted and moved to queue Q2
  - If a job release the CPU before the quantum expires, it is promoted to the next higher-level queue



# Multilevel Feedback Queues

- I/O bound processes release the CPU quite soon, while CPU bound processes often uses up the time quantum
- Why do we promote I/O-bound processes?
  - To improve I/O utilization (avoid the convoy effect)
  - To favor interactive processes
- Why do we demote CPU-bound processes?
  - Assign a large time quantum
  - To improve turnaround and throughput

# MULTIPLE PROCESSOR SCHEDULING

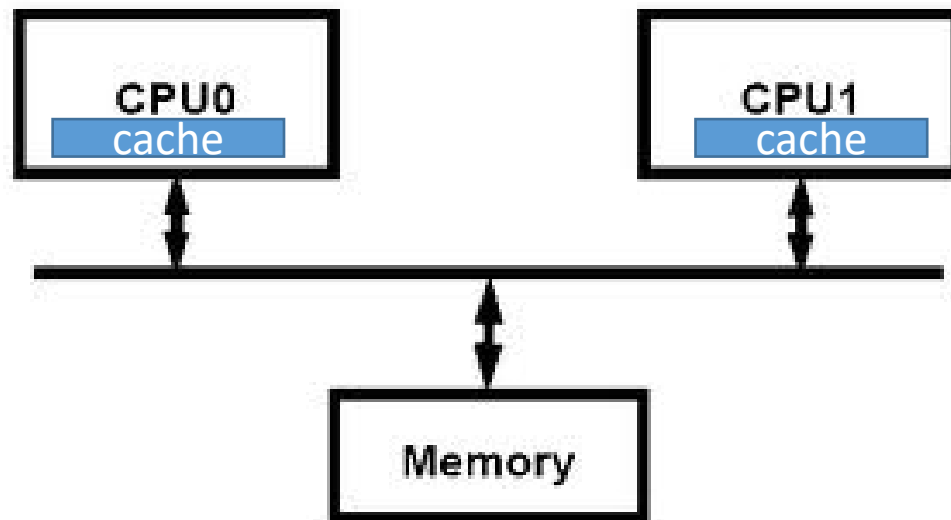


# Multiprocessor Architecture

- Homogeneous architecture
  - All CPUs share global memory (**SMP**)
- Heterogeneous architecture
  - CPUs have local memory and memory reference is either local or remote (**NUMA**)

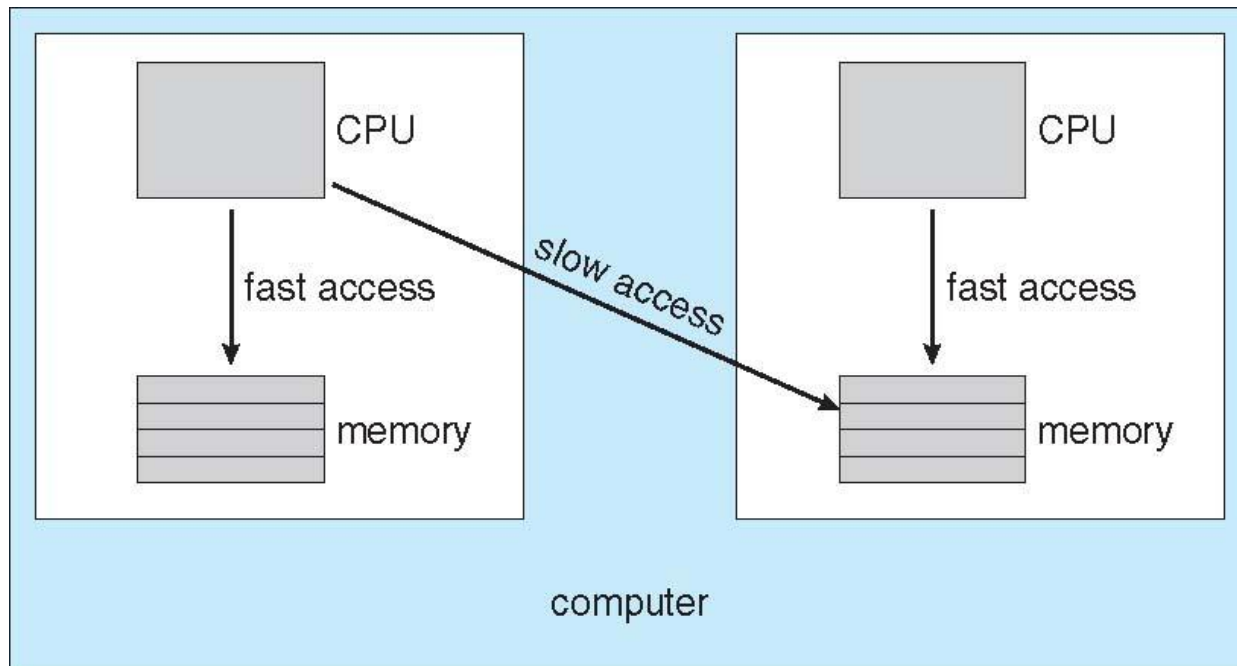
# Symmetric Multiprocessing (SMP)

- Multiple CPUs share the same main memory and contend for memory cycles via the common bus
- Good performance with a few CPUs; high costs with many CPUs due to bus contention and cache coherence operations



# Non-Uniform Memory Access (NUMA)

- Memory access time depends on the location of memory
- More scalable than SMP
- Common topology: ring, mesh, hypercube\*, ad-hoc

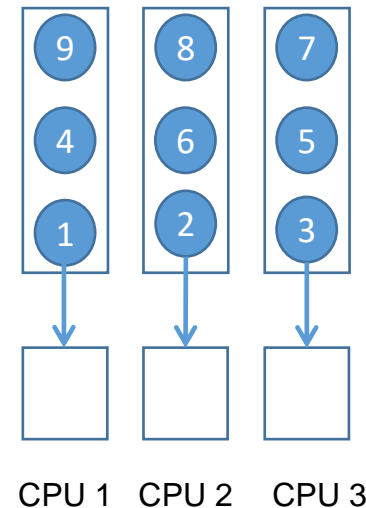


\* <https://lwn.net/Articles/254445/>

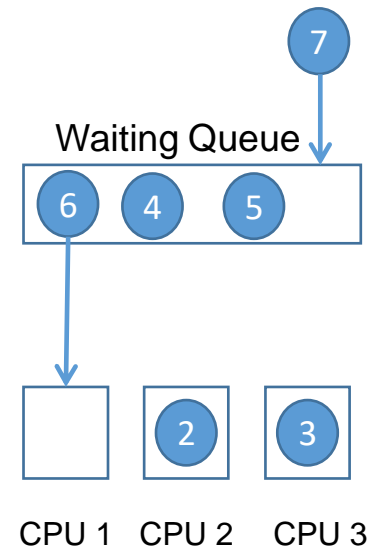
# Multiple-Processor Scheduling Algorithm

- Partitioned scheduling:  
Each processor is self-scheduling and has its own private queue of ready processes (e.g., Linux)
- Global scheduling:  
Only one processor accesses the system data structures and acts as the dispatcher

Partitioned Scheduling



Global Scheduling



# Multiple-Processor Scheduling

- **Processor affinity**

- The cost of process migration: cache re-population, pipeline re-start, and process data transfer (NUMA)
- Sometimes it's better to stick processes with processors
- Soft affinity, hard affinity

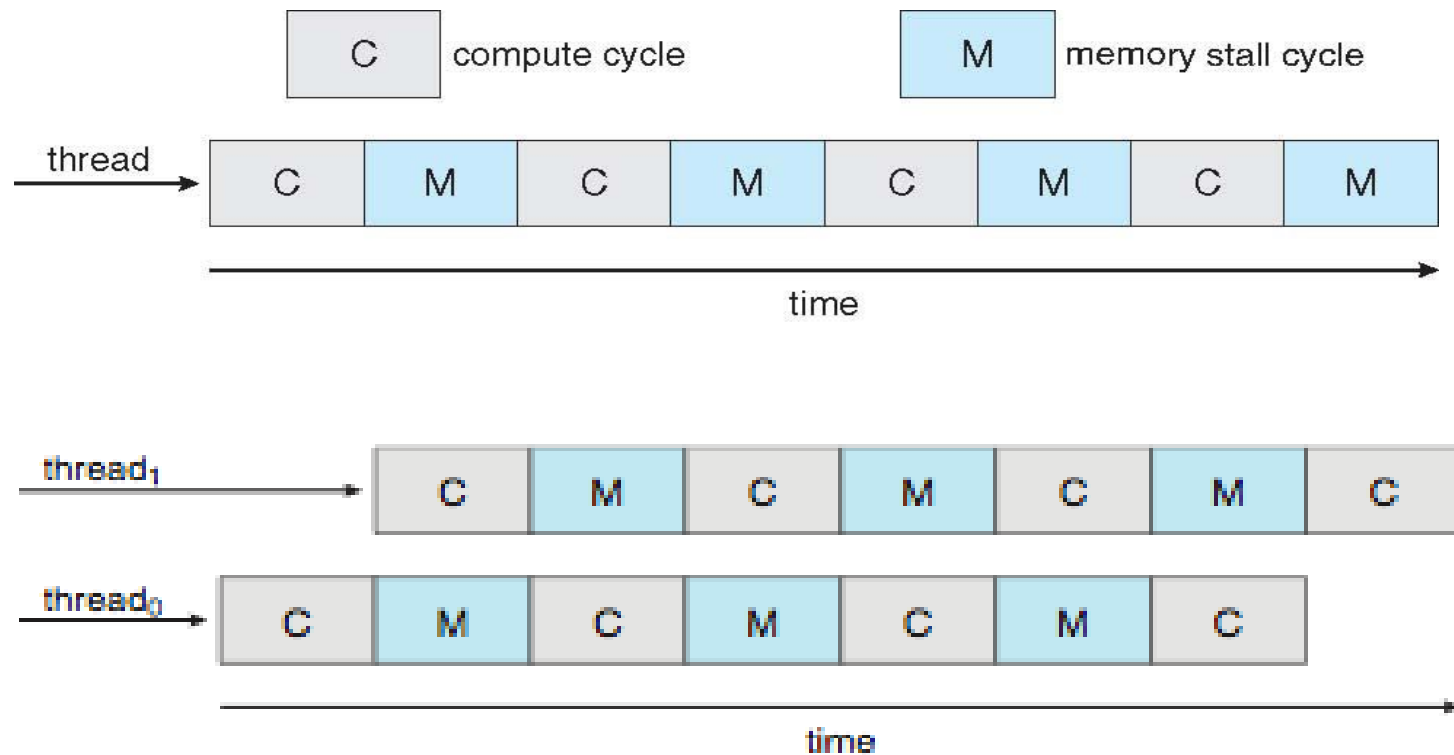
- **Load balancing**

- Balance the utilizations of processors
- Linux – runs push migration periodically and runs pull migration whenever a processor queue is empty
  - Push – pushes task from overloaded CPU to other CPUs
  - Pull – idle processors pulls waiting task from busy processor
- The two goals **conflict** with each other!

# Hardware Multithreading

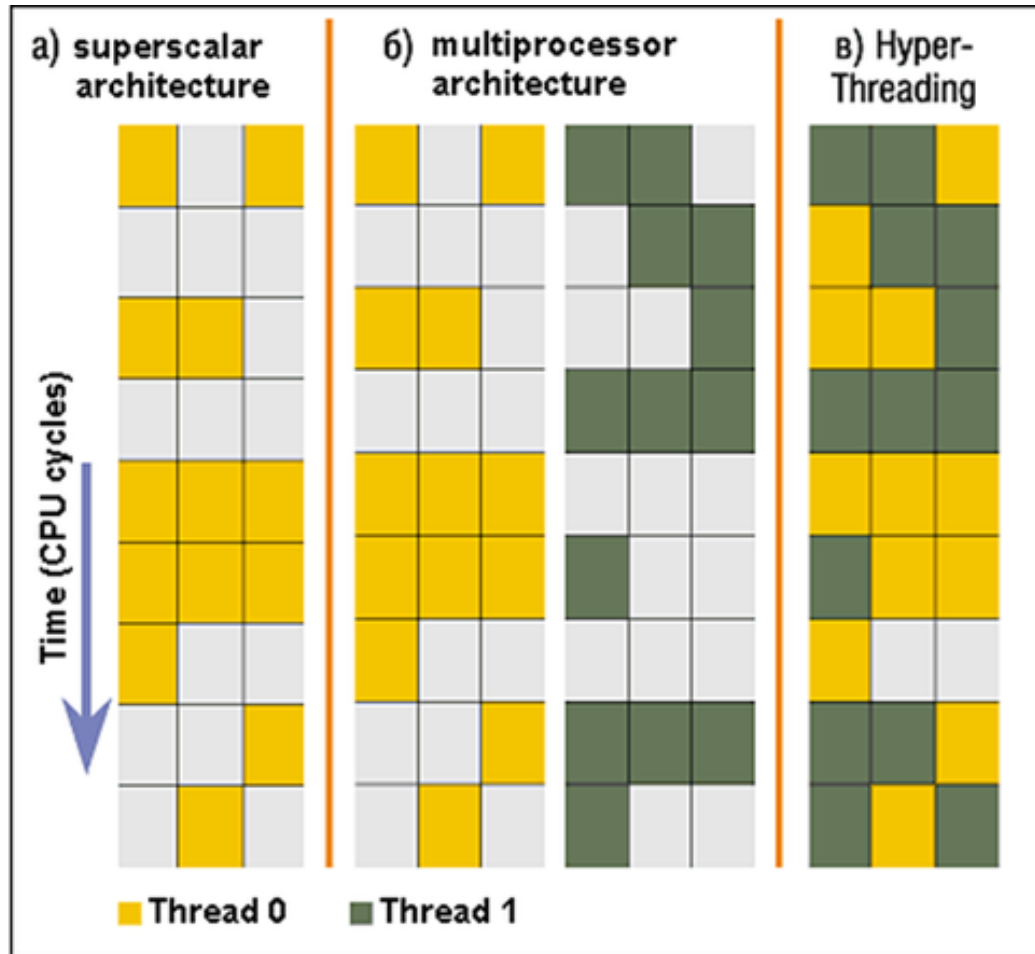
- A thread represents a **logical processor**, emulated by an **independent set of registers**
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens
- A thread here is hardware-based, **different** from “threads” in programming

# Improving CPU and Memory Cycle Utilizations



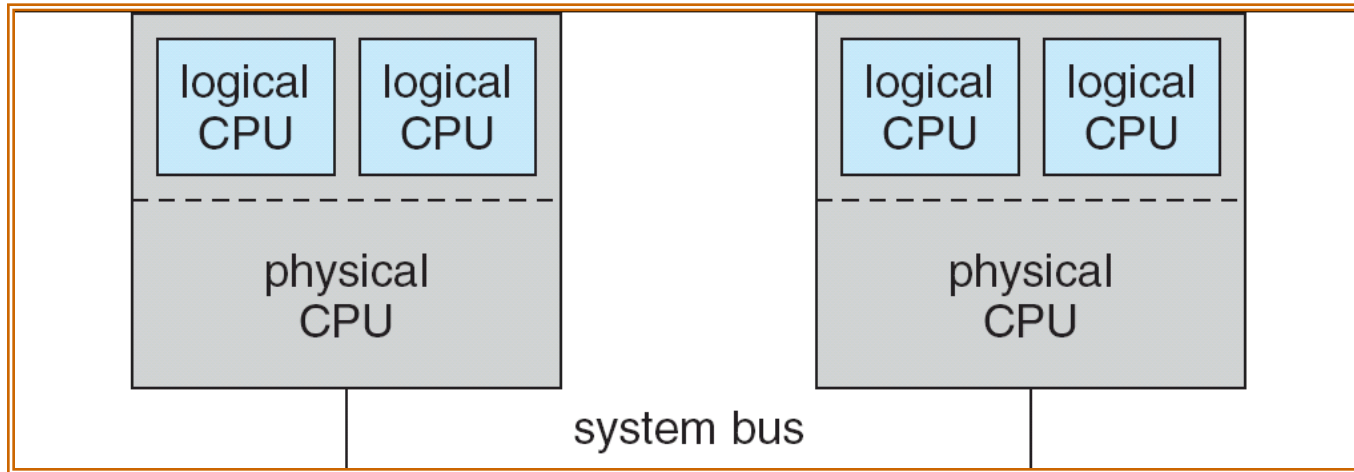
Higher utilizations of both CPU and memory

# Better Utilization of Processing Units in Superscalar (Multi-Issue) Processors





# Hierarchical Scheduling Domains



Assume: 2 PP, each has 2 LP.

$[1\ 1][1\ 1]$  is better than  $[2\ 0][2\ 0]$

$[1\ 0][1\ 0]$  is better than  $[1\ 1][0\ 0]$

Linux: **hierarchical** scheduling domains for load balancing

Firstly, evenly distribute process among physical CPUs

In each physical CPU, evenly distribute processes among logical CPUs

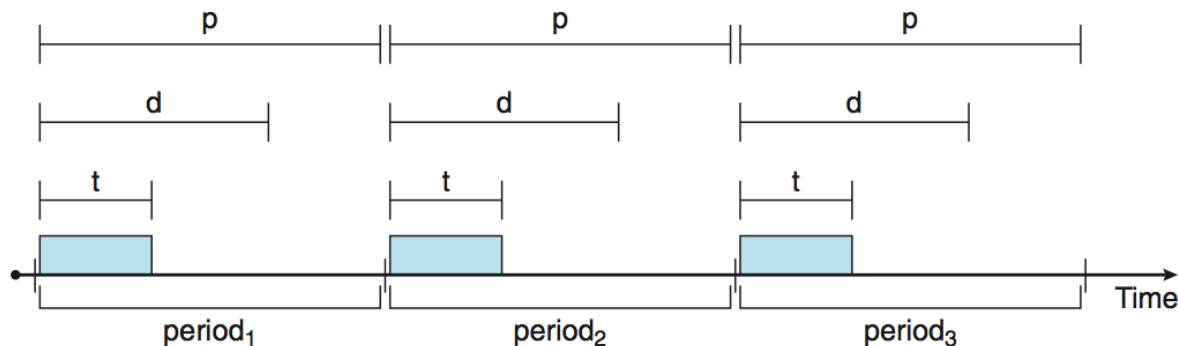
# Real-Time Scheduling

# Real-Time CPU Scheduling

- IEEE definition of real time systems:
  - “A real-time system is a system whose correctness includes its response time as well as its functional correctness.”
- **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

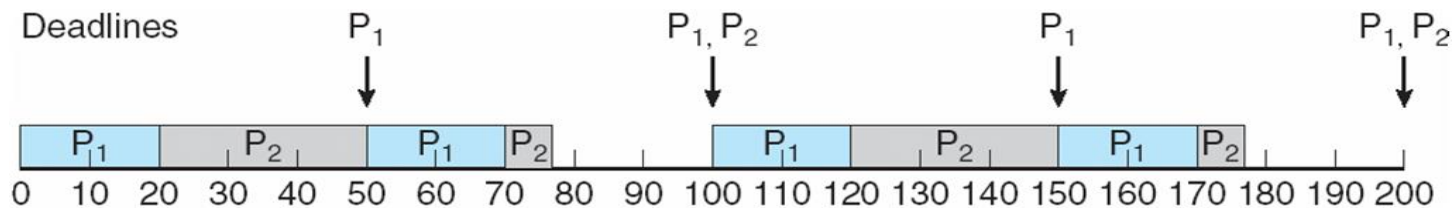
# Priority-based Scheduling

- The kernel supports preemptive, priority-based scheduling
  - This only guarantees soft real-time
  - For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time  $t$ , deadline  $d$ , period  $p$
  - $0 \leq t \leq d \leq p$
  - **Rate** of periodic task is  $1/p$



# Rate Monotonic Scheduling

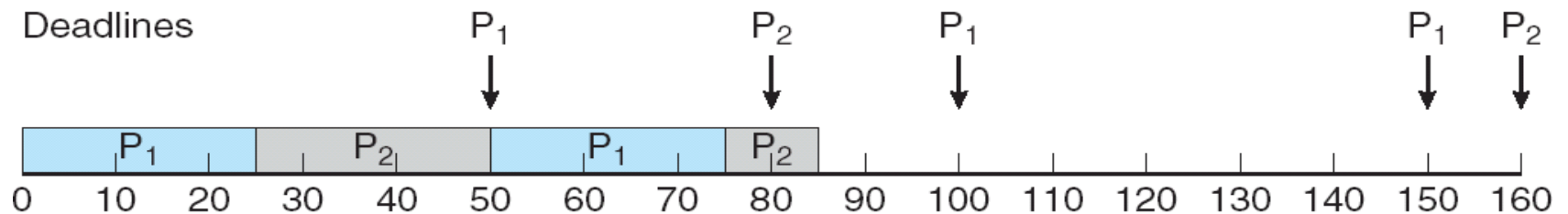
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P1 is assigned a higher priority than P2.



Computation:  $c_1=20$ ,  $c_2=35$

Periods: P1=50, P2=100

# Missed Deadlines with Rate Monotonic Scheduling

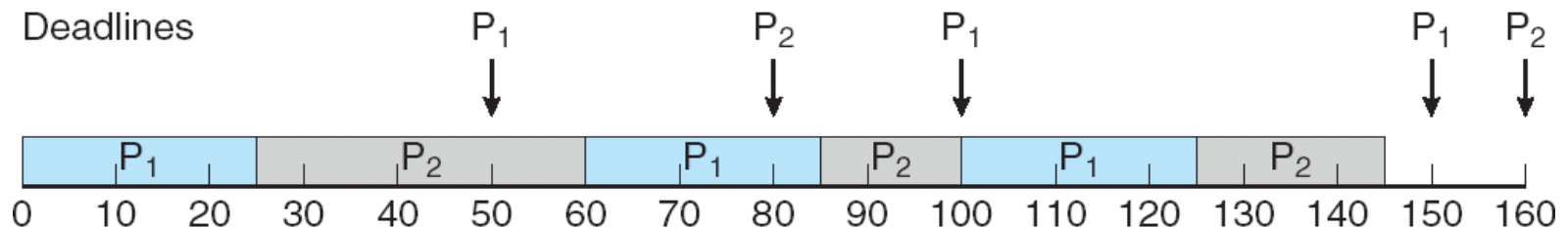


Computation:  $c_1=25$ ,  $c_2=35$   
Periods:  $P_1=50$ ,  $P_2=80$

# Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:

the earlier the deadline, the higher the priority; the later the deadline, the lower the priority



Computation:  $c_1=25$ ,  $c_2=35$

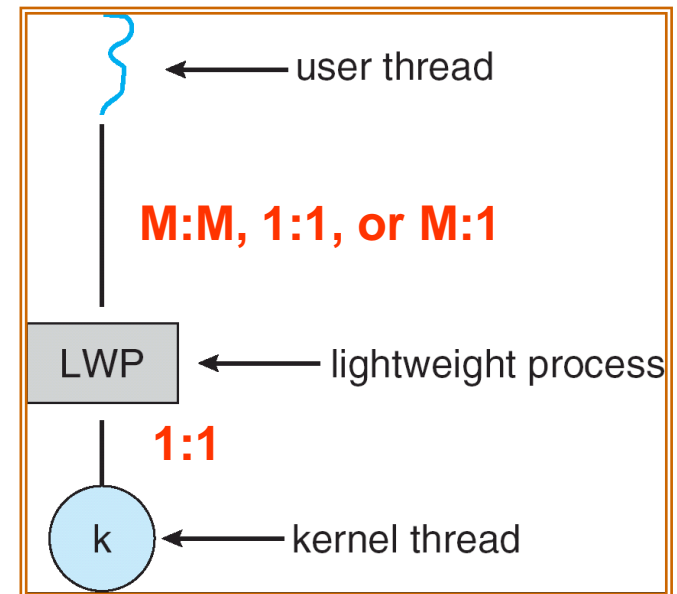
Periods:  $P_1=50$ ,  $P_2=80$

# THREAD SCHEDULING



# Light-Weight Process

- LWP is an **optional** abstraction for mapping of user threads to kernel threads
- An LWP is like a virtual processor on which user threads are scheduled
- Basically the mapping of LWPs to kernel threads is 1-1
- The mapping of user threads to LWP is 1-1, M-1, or M-M

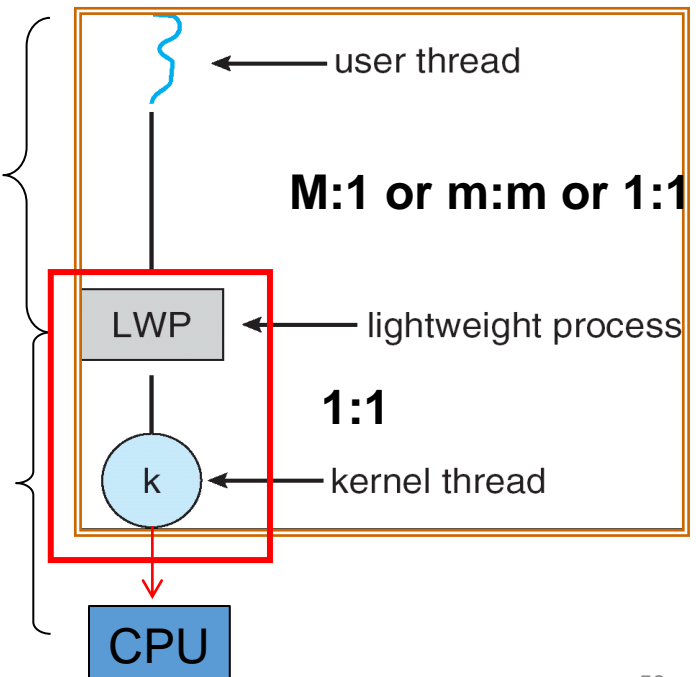


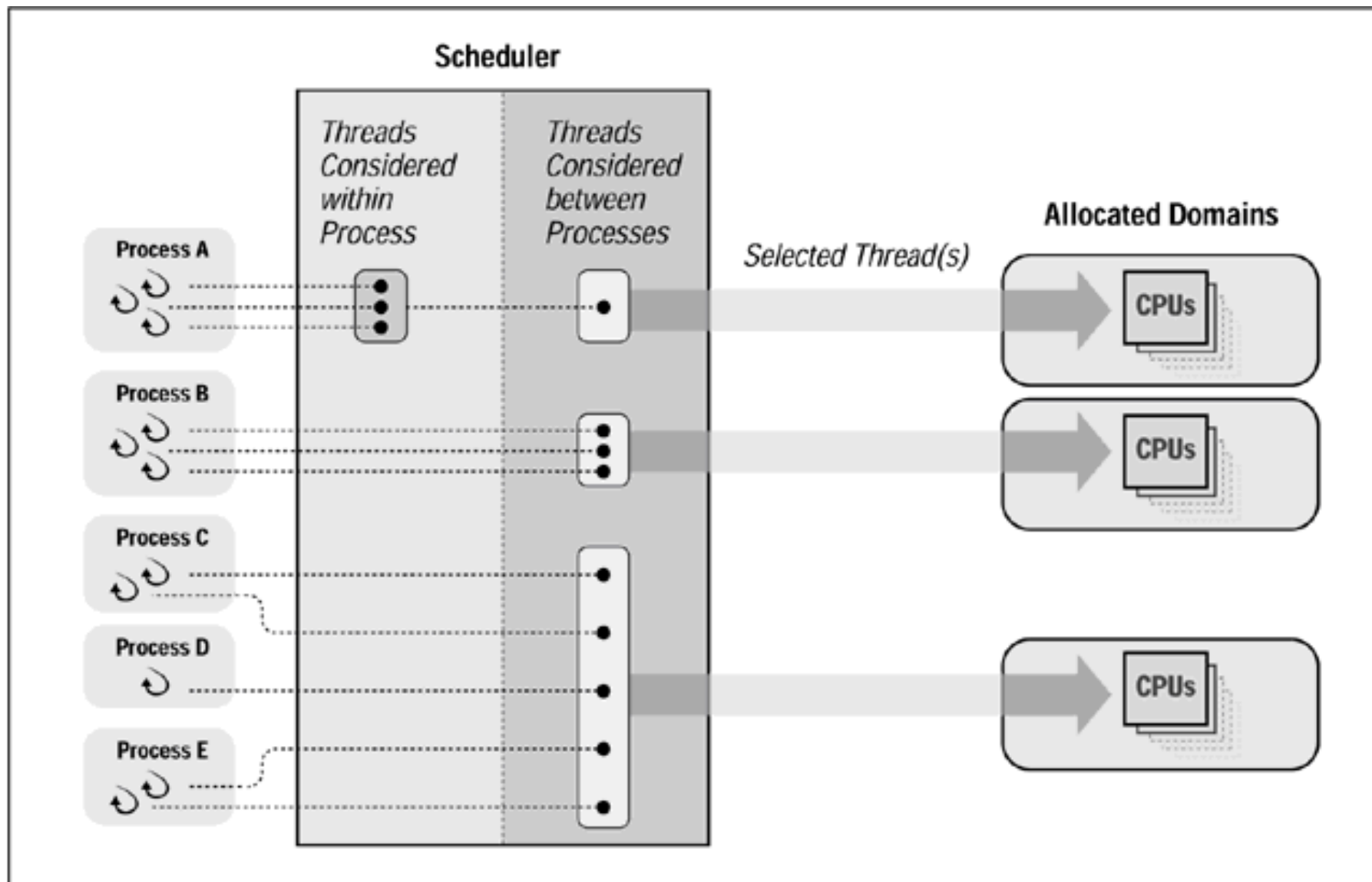
# Thread Scheduling

- Local Scheduling – How the threads library decides which thread to put onto an available LWP
  - Process contention scope (PCS)
- Global Scheduling – How the kernel decides which kernel thread to run next
  - System contention scope (SCS)

PCS (thread lib)

SCS(OS kernel)





```

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
    int i;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_t init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_t setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RR, or OTHER */
    pthread_attr_t setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread_exit(0);
}

```

# Pthread Scheduling API

Linux native thread model is 1-1, so in Linux pthread, **only** PTHREAD\_SCOPE\_SYSTEM is supported.

Changing the scheduling policy for system contention scope effective changes the thread scheduling policy

PTHREAD\_SCOPE\_SYSTEM  
PTHREAD\_SCOPE\_PROCESS

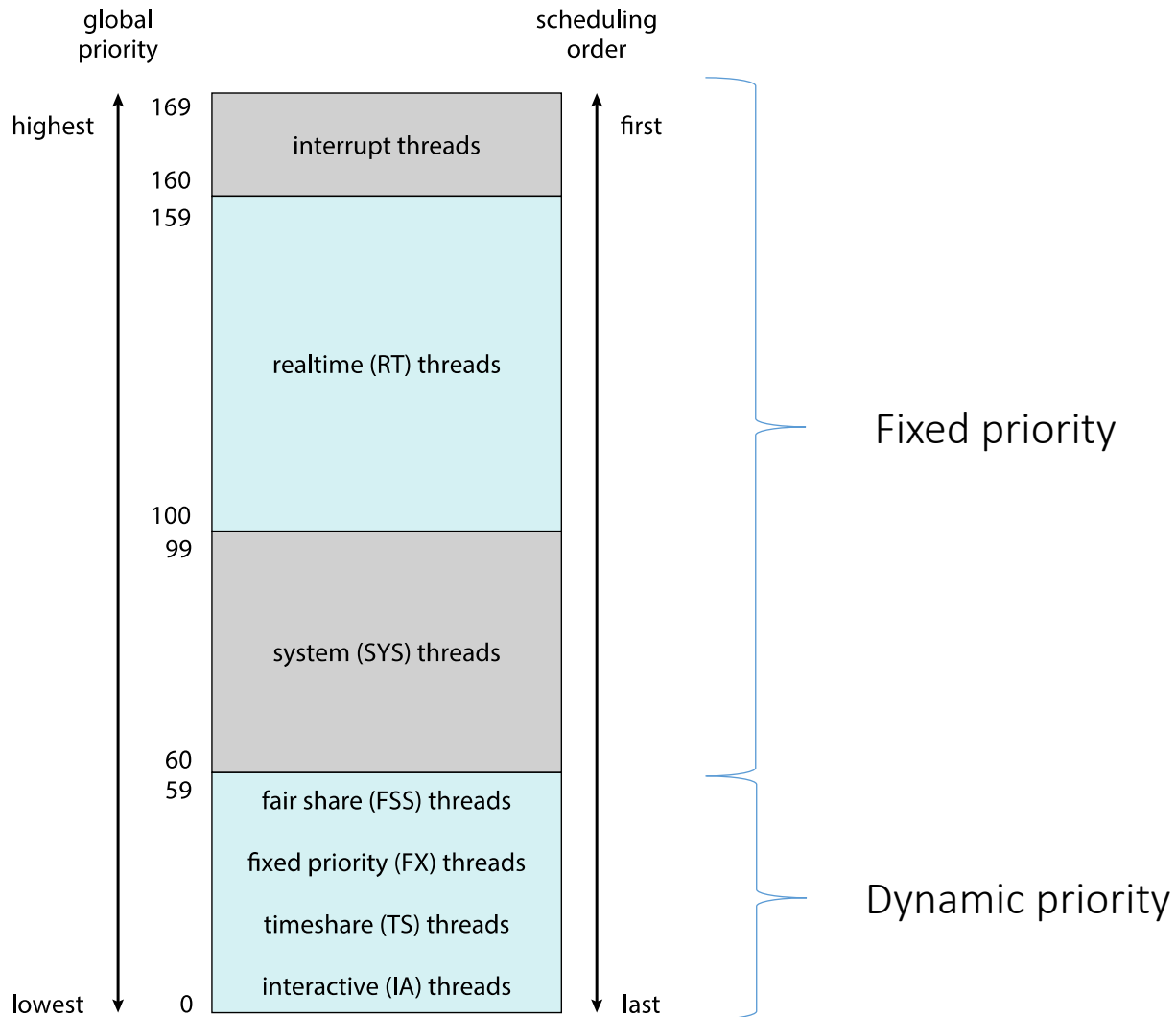
SCHED\_FIFO  
SCHED\_RR  
SCHED\_DEADLINE  
SCHED\_SPORADIC  
SCHED\_OTHER → in Linux, it's CFS

# OPERATING-SYSTEM EXAMPLES

# Solaris Scheduling

- Priority-based scheduling
  - RR on threads of the same priority
- Six classes available
  - Time sharing (default) (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair Share (FSS)
  - Fixed priority (FP)
- Given thread can be in one class at a time
- Each class has its own scheduling algorithm
- Time sharing is multi-level feedback queue
  - Loadable table configurable by sysadmin

# Solaris Scheduling



# Solaris Dispatch Table

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

- Priority + → time quantum –
- Processes run out their time quantum are demoted
- Processes return from I/O operations are promoted

High/important



# Linux CFS Scheduling (2.6.23+)

- Virtual runtime (vruntime)
  - Increases as a process executes on the CPU
  - The process of the smallest vruntime is selected for running
  - As known as the virtual clock algorithm
- Completely Fair Scheduler (CFS)
  - CFS favors IO-bound processes over CPU-bound processes
  - Assigning an adjustment to processes according to their I/O tendency
  - IO-bound processes increase their vruntime slower than CPU-bound processes

# CFS: Linux CPU Scheduling (2.6.23+)

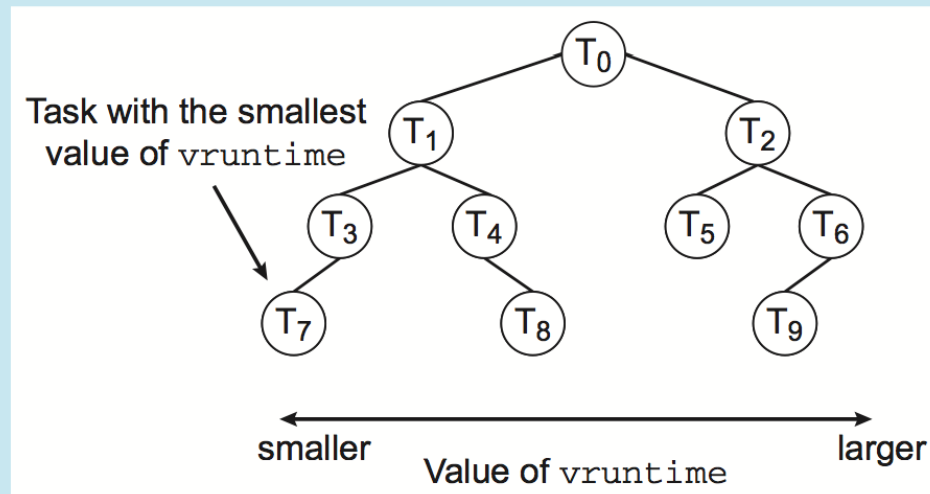
- Nice value
  - -20~+19 (high priority ~ low priority)
  - The increasing rate of vruntime
    - E.g., with -20, after a process runs 200ms, its vruntime increases less than 200ms
    - with +19, after a process runs 200ms, its vruntime increases larger than 200ms
  - Processes with small nice values increase their vruntime slower, and thus, receive larger portions of CPU time
- CFS determines the increasing rate of vruntime for a process based on its **nice value** and its **recent execution history** (IO- or CPU-bound)

# Virtual Runtime (Virtual Clock Algorithm)

- Smallest vruntime first
- Virtual clock 1: increasing rate = 1
- Virtual clock 2: increasing rate = 0.5
- VC1:  $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$
- VC2:  $0 \rightarrow 0.5 \rightarrow 1 \rightarrow 1.5 \rightarrow 2 \rightarrow 3.5 \rightarrow 4$
- Out of every 3 units of time, VC1 receives 1 unit of time, while VC2 receives 2 units of time
  - $1/1$  and  $1/0.5 = 1:2$
- A slower clock receives more CPU attention

# CFS vruntime Mechanism

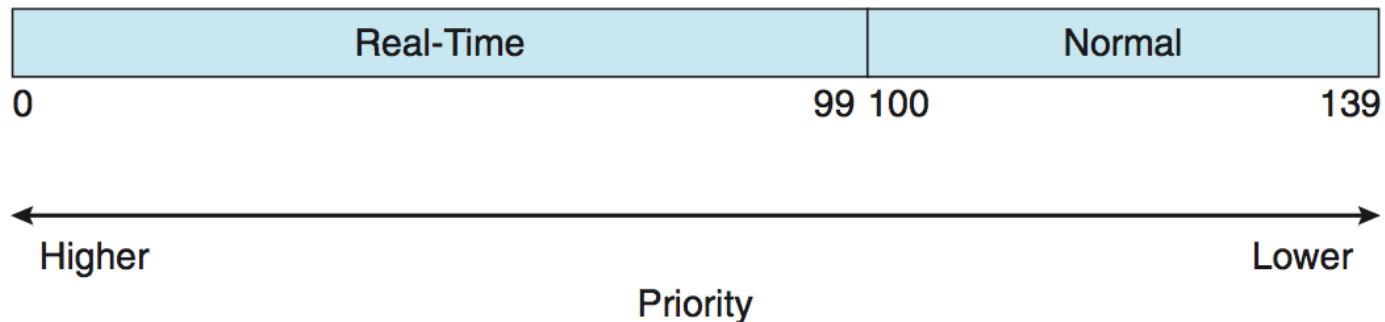
The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:



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 (Cont.)

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



End of Chapter 5

# Review Questions

1. Compare FCFS and SJF in terms of efficiency and fairness
2. What is the trade-off of using a large time slice and a small one for RR scheduling?
3. Why MLFBQ algorithm favors I/O-bound processes?
4. Why NUMA becomes increasingly popular compared with SMP?
5. Compare the pros and cons of global and dispatch scheduling for multiprocessor systems
6. Why Linux uses a hierarchical approach to load balancing? Why not use a flat model?
7. Discuss the difference of the two contention scopes (process & system) in Pthread scheduling
8. Design a case for the virtual clock algorithm, in which two processes receives 40% and 60% of the CPU time
9. In practice, rate-monotonic scheduling is more common than earliest-deadline first scheduling. Please comment on this
10. Comprehend Little's Law described in the next page

# Little's Formula

- $n$  = average queue length
- $W$  = average waiting time in queue
- $\lambda$  = 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