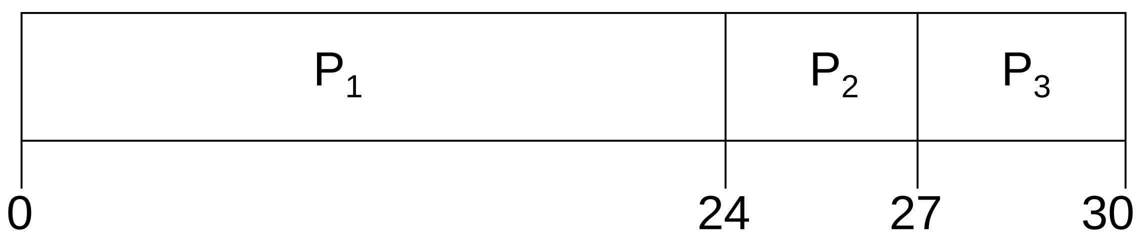
CPU Scheduling (II)

First-Come, First-Served (FCFS) Scheduling

(Ready) Process	(Next) 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 time interval between arrivals is negligible*. The Gantt Chart for the schedule is:



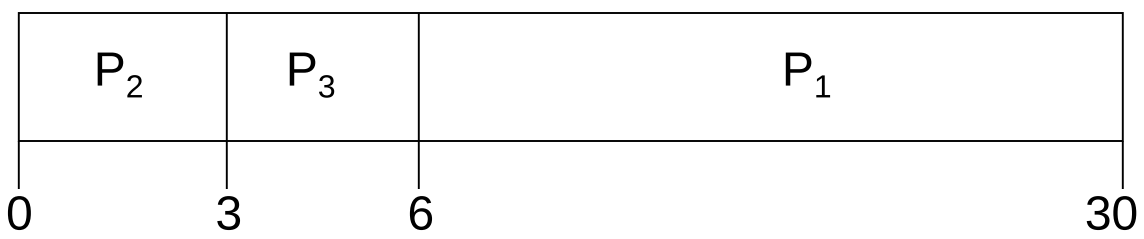
- Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$

FCFS Scheduling (Cont)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:



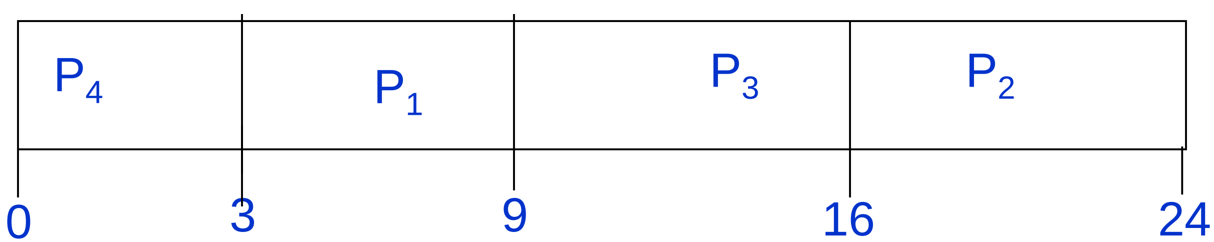
- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- *Convoy effect* when short processes are scheduled behind a long process

Shortest-Job-First (SJF) Scheduling

- Schedule first the process with the shortest burst time

Process	(CPU) 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: Preemptive or Nonpreemptive

SJF can be either preemptive or non-preemptive

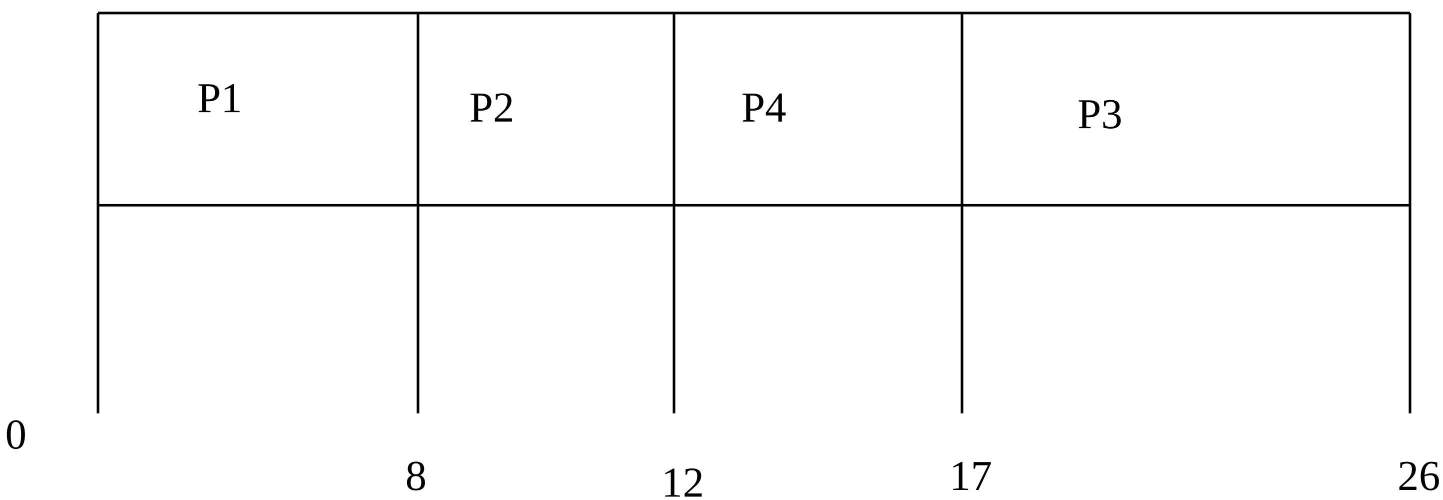
- Preemptive (Shortest-remaining-time-first): the currently running process can be preempted by a newly arriving process that has shorter CPU burst
- Non-preemptive: even a newly arriving process has shorter CPU burst than the currently running one, the running process is not preemptive.

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

What are the Gantt charts for the preemptive SJF and the nonpreemptive SJF?

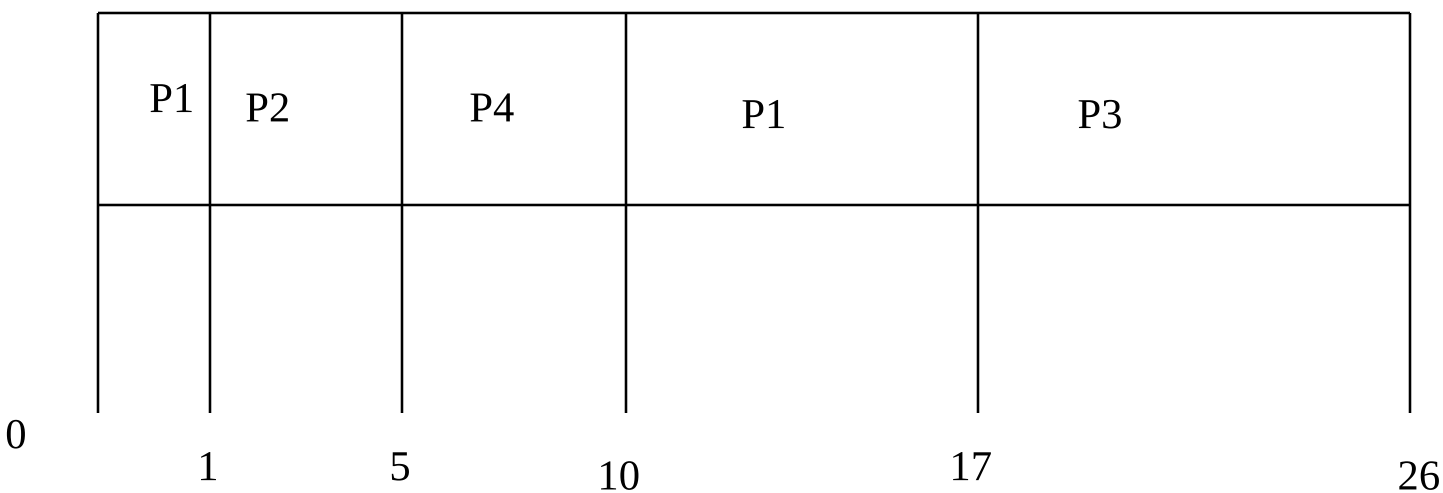
Non-preemptive SJF

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



Preemptive SJF

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



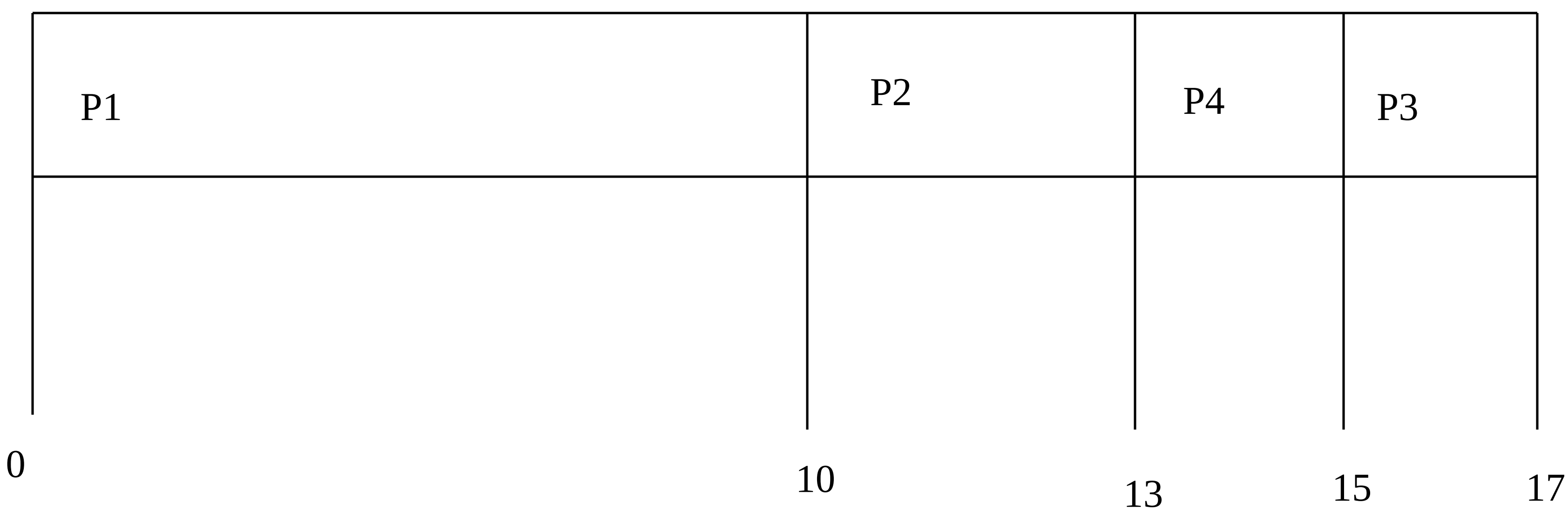
Priority Scheduling

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

Process	Arrival Time	Burst Time	Priority
P_1	0	10	3
P_2	*1*	3	1
P_3	2	2	4
P_4	3	2	2

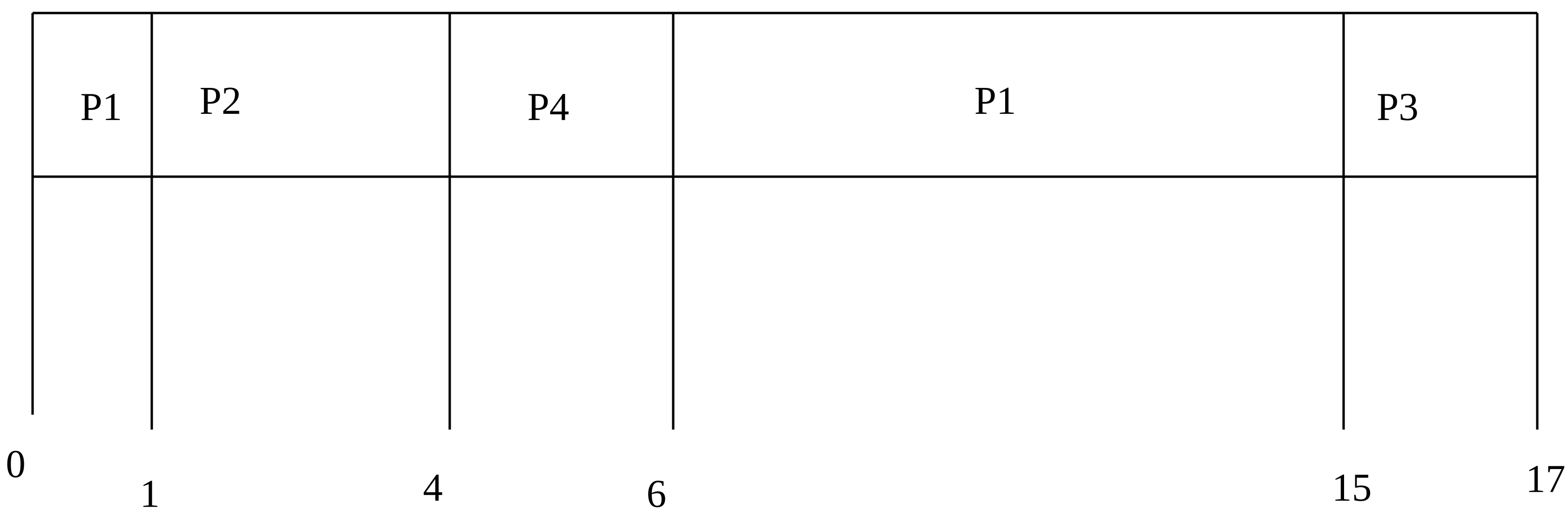
Priority Scheduling: Non-preemptive

Process	Arrival Time	Burst Time	Priority
P_1	0	10	3
P_2	*1*	3	1
P_3	2	2	4
P_4	3	2	2



Priority Scheduling: Preemptive

Process	Arrival Time	Burst Time	Priority
P_1	0	10	3
P_2	1	3	1
P_3	2	2	4
P_4	3	2	2



Priority Scheduling

- Problem: **Starvation** – low priority processes may never execute
- Solution: **Aging** – as time progresses increase the priority of the waiting process

Process	Arrival Time	Burst Time	Priority
P_1	0	10	1
P_2	1	5	10
P_3	2	2	2
P_4	3	2	3
P_5	4	2	3
P_6	5	2	3
P_7	6	1	3
P_8	7	1	2
…	…	…	…

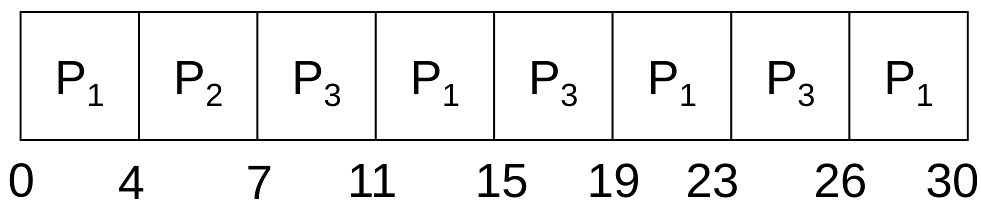
Round Robin (RR)

- 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.

Example of RR with Time Quantum = 4

Process	Arrival Time	Burst Time
P_1	*0*	16
P_2	*1*	3
P_3	*2*	11

- The Gantt chart is:



- Higher average turnaround than SJF, but better *responsiveness*

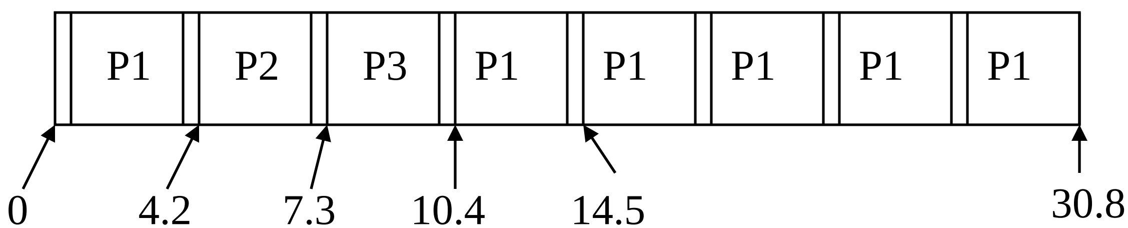
Round Robin (RR)

- 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 $\Rightarrow$ High overhead for context switch
  - q must be large with respect to context switch, otherwise overhead is too high

Practical Issue 1: Context Switch Time

Process	Arrival Time	Burst Time
P_1	*0*	24
P_2	*1*	3
P_3	*2*	3

- The Gantt chart is:



Assume: Quantum = 4; context switch time = 0.1

Practical Issue 2: Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the lengths of previous CPU bursts, using exponential averaging

Multilevel Queue Scheduling

- Applicable when processes are easily classified into different groups
- Ready queue is partitioned into separate queues: For example,
  - foreground (interactive)
  - background (batch)

Multilevel Queue Scheduling

- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done also between the queues
  - Fixed priority scheduling. For example, 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, and 20% to background in FCFS

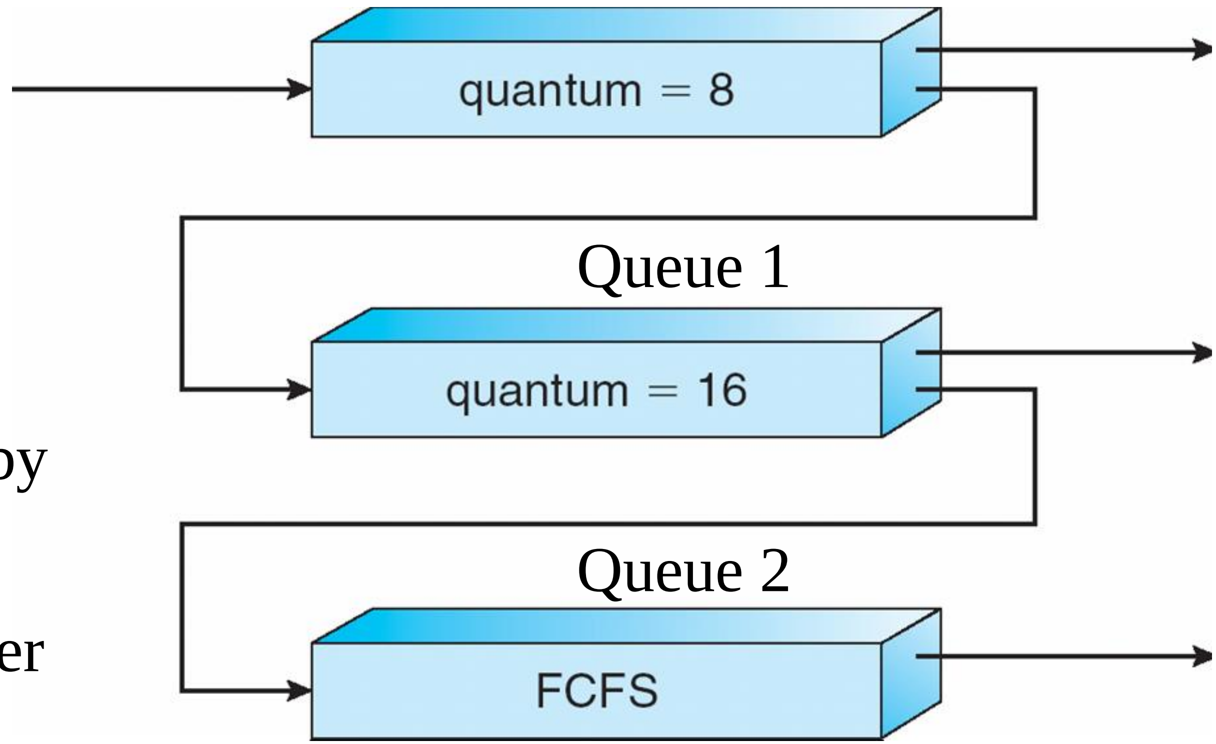
Multilevel Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way

• Queue 1: with lower priority than Queue 0; processes in it are scheduled only when Queue 0 is empty and they can be preempted by new-comer of Queue 0

• Queue 2 has even lower priority than Queue 1.

Queue 0: with the highest priority



Multilevel Feedback Queue Scheduling

- 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

CPU Scheduling (III)

Multiple-Processor Scheduling

- **Assumption:** processors are homogeneous (i.e., identical in functionality)
- **Two approaches for 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

Multiprocessor Scheduling: Processor Affinity

- **Processor affinity** – process has affinity for processor on which it is currently running
  - Why? For example, information caching may become less effective if a process migrates frequently between different processors.
- **Soft affinity**
  - Attempting to keep a process running on the same processor, but not guaranteeing that it will do so
- **Hard affinity**
  - A process does not migrate between different processors
- **Hybrid**
  - A process migrates only among a certain processor set

Example

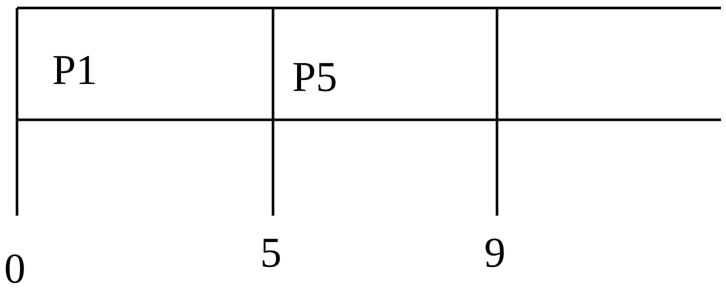
Consider a SMP computer composed of two symmetric processors. A certain OS is run on the computer. With the OS, these two processors share a common set of process queues. Suppose following processes are submitted to the computer:

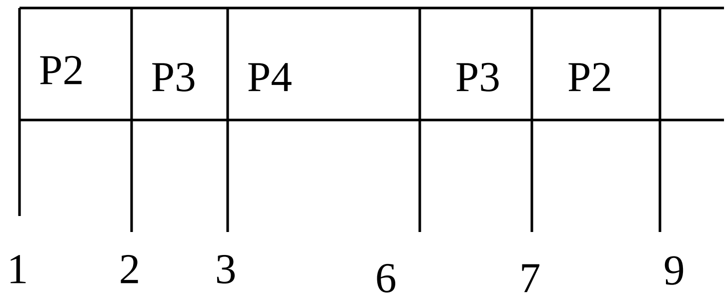
Process	Arrival Time	Burst Time	Priority
P_1	0	5	1
P_2	*1*	3	10
P_3	2	2	4
P_4	3	3	2
P5	*4*	*4*	*5*

How are the processes scheduled when (i) different scheduling algorithms are used and (ii) different affinity settings are used?

Example: 2 Processors; Pre-emptive Priority Scheduling; Hard Affinity

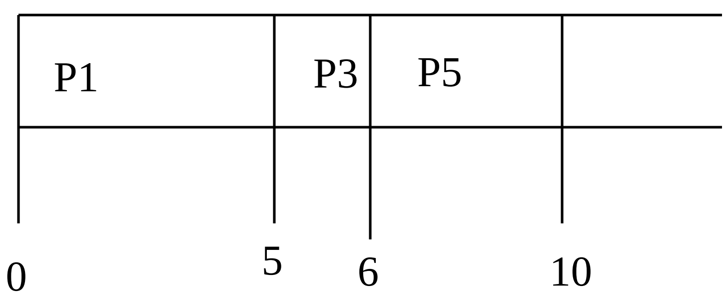
Process	Arrival Time	Burst Time	Priority
P_1	0	5	1
P_2	*1*	3	10
P_3	2	2	4
P_4	3	3	2
P5	*4*	*4*	*5*

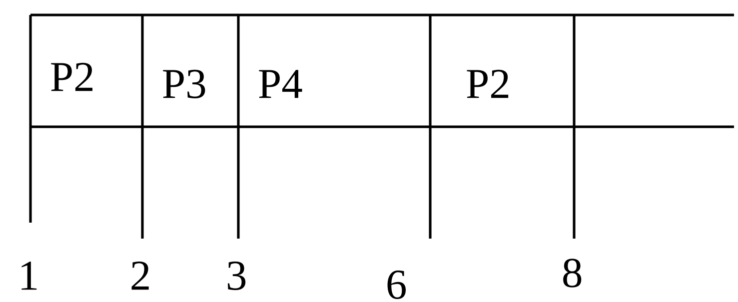




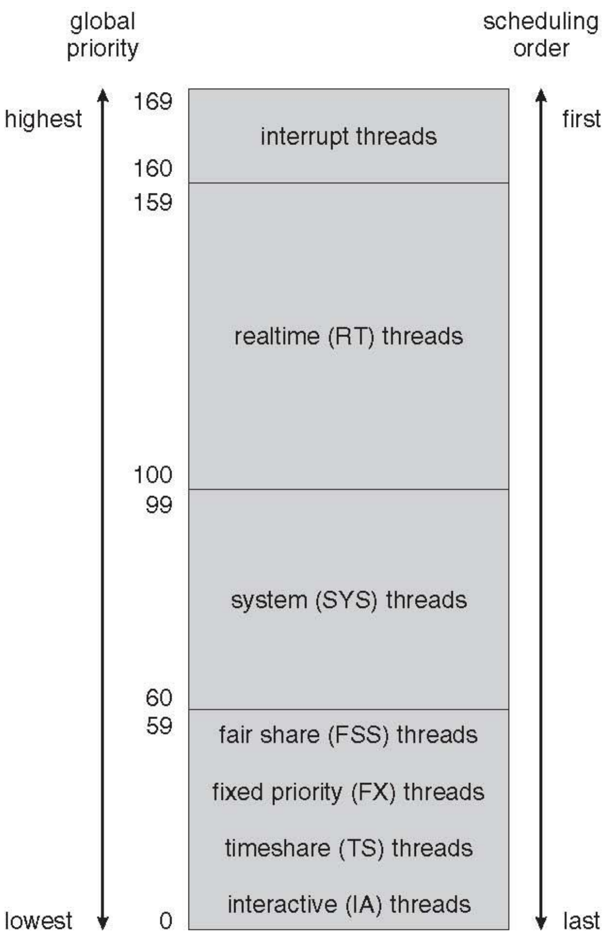
Example: 2 Processors; Pre-emptive Priority Scheduling; Soft Affinity

Process	Arrival Time	Burst Time	Priority
P_1	0	5	1
P_2	*1*	3	10
P_3	2	2	4
P_4	3	3	2
P5	*4*	*4*	*5*





Solaris Scheduling



- Each thread belongs to one of 6 classes
  - Time sharing (TS)
  - Interactive (IA)
  - Real time (RT)
  - System (SYS)
  - Fair share (FSS)
  - Fixed priority (FP)
- Threads belonging to different classes have different priorities.
- Threads in the same class can have different priorities. Scheduler converts the class-specific priorities into global priorities and do scheduling based on global priorities.

Essentially, 170 queues are maintained.

Solaris Scheduling

- Dynamically adjusting priorities and time quanta according to a dispatch table

(Note: the greater the priority number is, the higher the priority is.)

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

- Each queue uses RR scheduling algorithm.
- Policies for migration are defined.

Windows XP Scheduling

- Priority scheduling (each priority is associated with a time quantum)

Priority Classes

Intra-class Priorities

	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

Priority of a thread may be adjusted dynamically: (i) lowered after a quantum ends; (ii) boosted after switching from “waiting” to “ready”

Processor Scheduling in Linux

- Multi-task (kernel thread) scheduling

- Real Time vs. Normal Tasks
  - Task running on Linux can explicitly be classified as real-time or normal tasks.
    - Real time tasks have priorities: 0-99
    - Normal tasks priorities: 100-139

Linux Hierarchical, Modular Scheduler

- Composed of a hierarchy of scheduling classes
- By default, from higher to lower:
  - RT class
    - Applying FCFS and/or RR to run real time tasks
    - Always get priority over non real time tasks
  - CFS class
    - Applying "completely fair scheduling" policy to schedule normal tasks

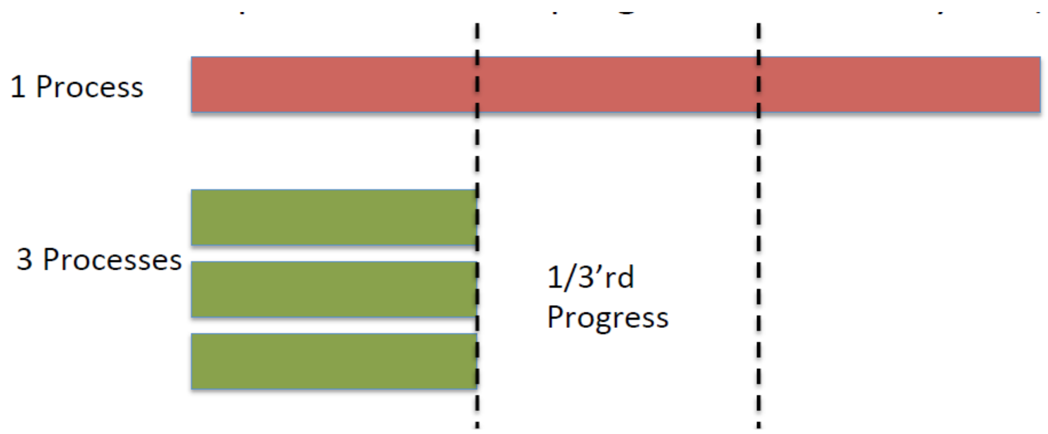
Skeleton of the Hierarchical Scheduler

Code from kernel/sched.c:

```
class = sched_class_highest;
    for ( ; ; ) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
        /*
         * Will never be NULL as the idle class always
         * returns a non-NULL p:
         */
        class = class->next;
    }
```

CFS

- Introduced in kernel 2.6.23
- Models an ideal multitasking CPU
  - Infinitesimally small timeslice
  - n processes: each progresses uniformly at 1/n of the rate
  - Problem: real CPU can't be split into infinitesimally small time slice without excessive overhead



CFS

- Core ideas: dynamic time slice and order
  - Scheduler keeps track of the CPU time consumed by each task.
  - If the current task consumes more-than-a-threshold time than the task consuming the minimal CPU time ➔ scheduling: swap the current task with the min-CPU-time task
  - A minimum reschedule time is set to avoid overly frequent scheduling

CFS

- How to find the min-CPU-time task?
- Tasks are organized as a red-black tree (approximately-balanced binary search tree) based on the CPU time that have consumed
- The min-CPU-time task is the most left element on the tree.
- Operation on the tree: O(log N), where N is the number of tasks.

Exam 1

Coverage

- Overview
- Process
- Thread
- Scheduling

Overview

- Interrupt
  - What are interrupts used for? How does it work? Types, examples?
- Dual mode execution
  - What are privilege instructions? What is kernel mode? What is user mode? When mode switch is needed?
- How to protect memory?
- System call
  - Why system calls are need? how are system calls implemented? Examples of system calls
- Major components of OS

Process

- Structures of process: user space & kernel space
- Process creation: how fork() works
- Process termination: exit(), kill()
- Inter-process communication mechanisms
  - Two basic modes
  - Pipe
  - Shared memory
  - Signal

Thread

- Internal structure of multi-thread process
- Kernel threads: how clone() works
- User threads: pthread library (basic functions)
- Mapping from user threads to kernel threads (deas)

Scheduling

- Internal data structures to support scheduling
- Concept of contexts and context switch
- Basic scheduling algorithms: FCFS, SJF, Priority, RR
- Multi-level queue scheduling
- Multi-processor scheduling
- Quantitative analysis of performance: waiting time, turnaround time, etc.