


# 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

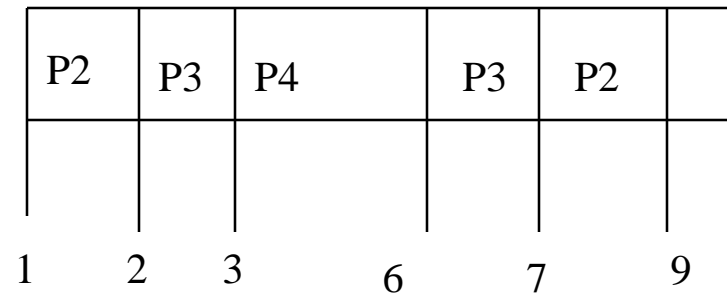
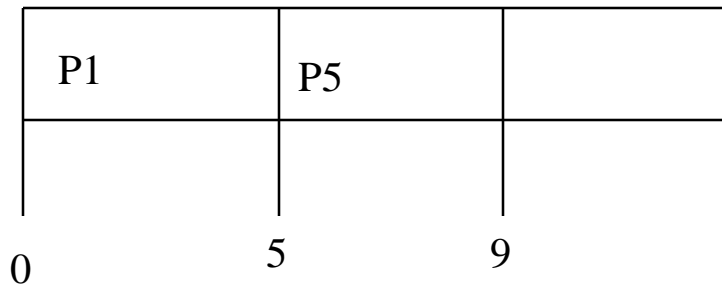
Consider a SMP computer composed of two symmetric processors. A certain OS is run on the computer. With the OS, these two processors share a common set of process queues. Suppose following processes are submitted to the computer:

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	0	5	1
$P_2$	1	3	10
$P_3$	2	2	4
$P_4$	3	3	2
$P_5$	4	4	5

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

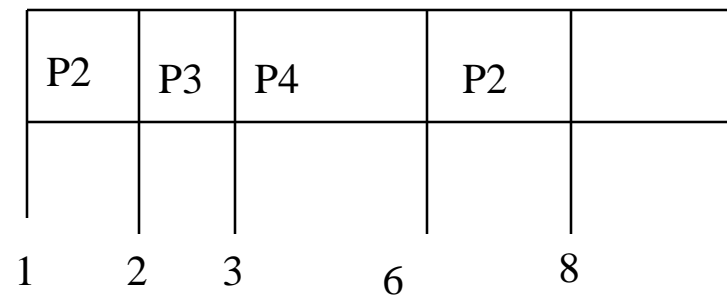
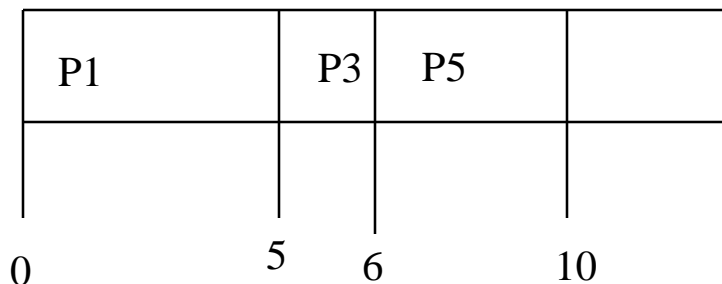
# Example: 2 Processors; Pre-emptive Priority Scheduling; Hard Affinity

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	0	5	1
$P_2$	1	3	10
$P_3$	2	2	4
$P_4$	3	3	2
$P_5$	4	4	5

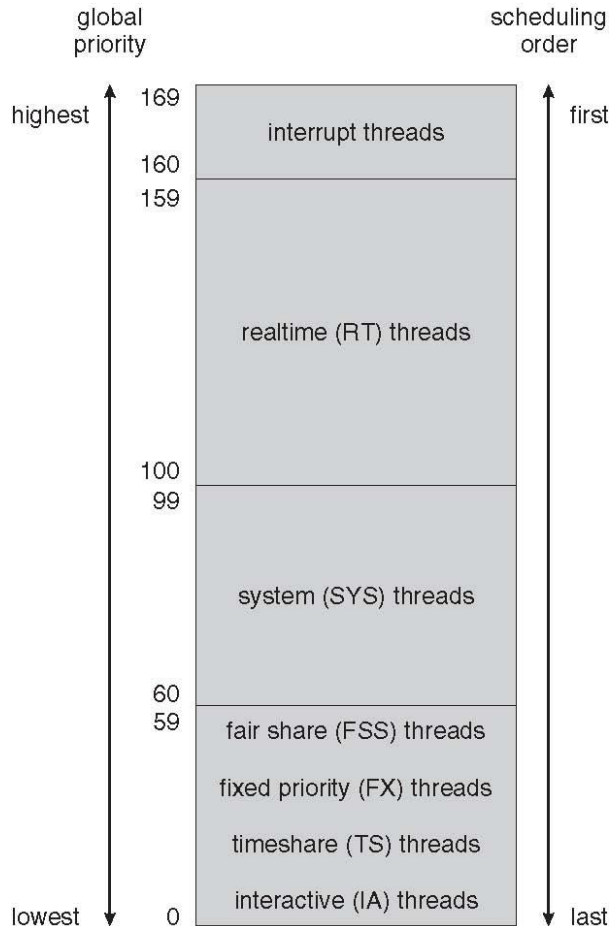


# Example: 2 Processors; Pre-emptive Priority Scheduling; Soft Affinity

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>	<u>Priority</u>
$P_1$	0	5	1
$P_2$	1	3	10
$P_3$	2	2	4
$P_4$	3	3	2
$P_5$	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. )

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

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



# Windows XP Scheduling

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

## Priority Classes

	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

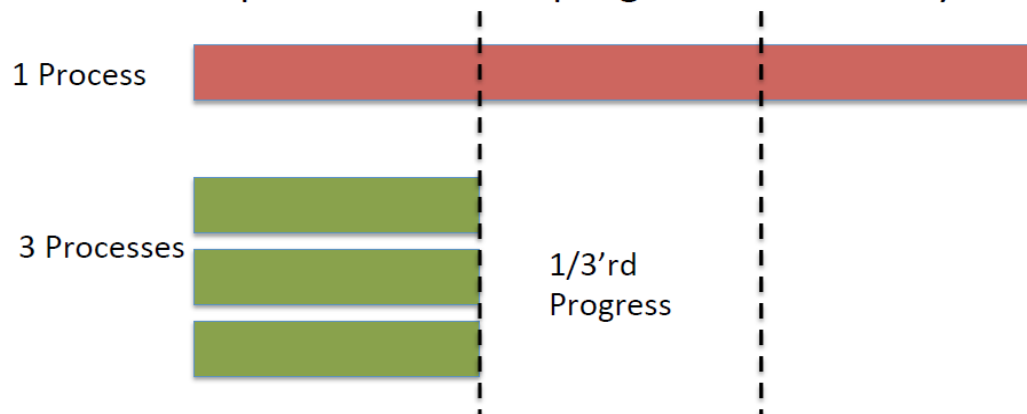
# Skeleton of the Hierarchical Scheduler

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.

# 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.