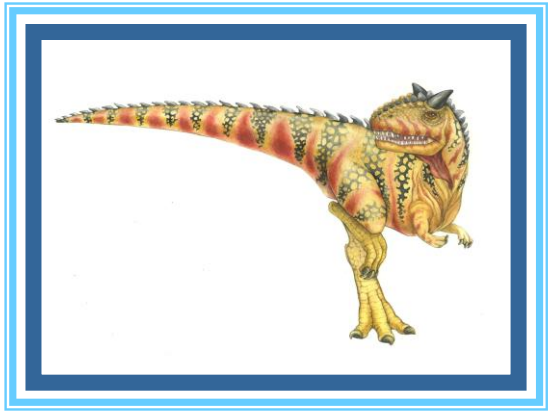
Chapter 5: CPU Scheduling



Chapter 5: CPU Scheduling

Basic Concepts

Scheduling Criteria

Scheduling Algorithms

Thread Scheduling

Multi-Processor Scheduling

Real-Time CPU Scheduling

Operating Systems Examples

Algorithm Evaluation

Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems
- Apply modeling and simulations to evaluate CPU scheduling algorithms

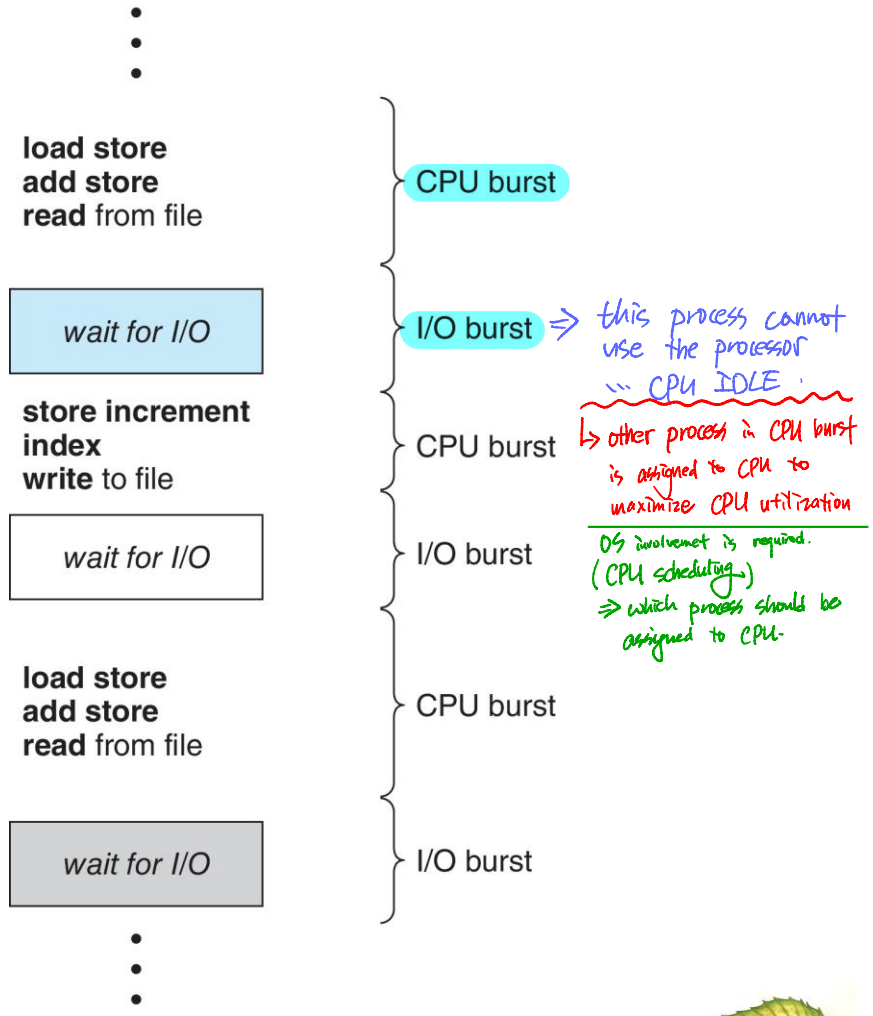
Basic Concepts

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

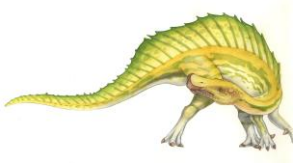
CPU burst distribution is of main concern



code ← CPU execution only : CPU burst
I/O execution needed : I/O burst

CPU scheduling algorithm
- determine when process is deprived to assign new process.
- determine which new process gets CPU resources

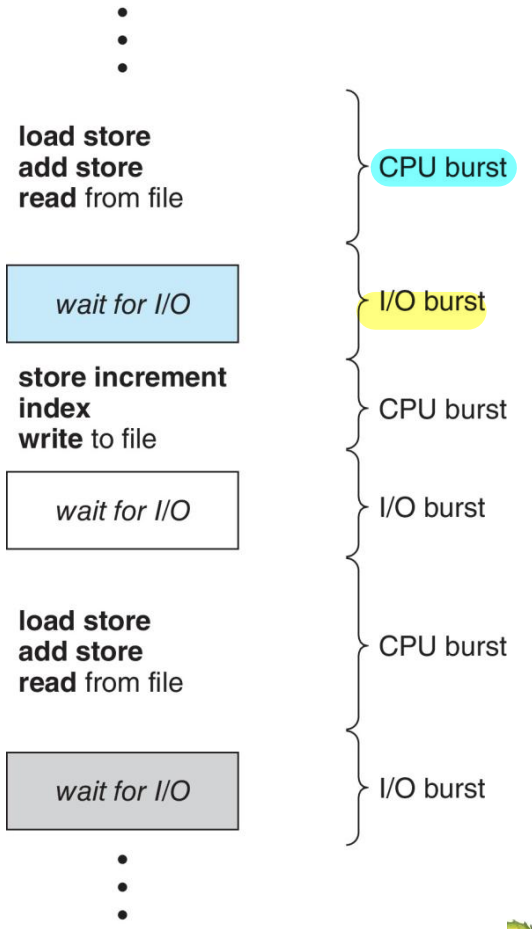
I/O burst ⇒ this process cannot use the processor ... CPU IDLE
↳ other process in CPU burst is assigned to CPU to maximize CPU utilization
OS involvemet is required. (CPU scheduling)
⇒ which process should be assigned to CPU.



Basic Concepts (Cont'd)

An example program

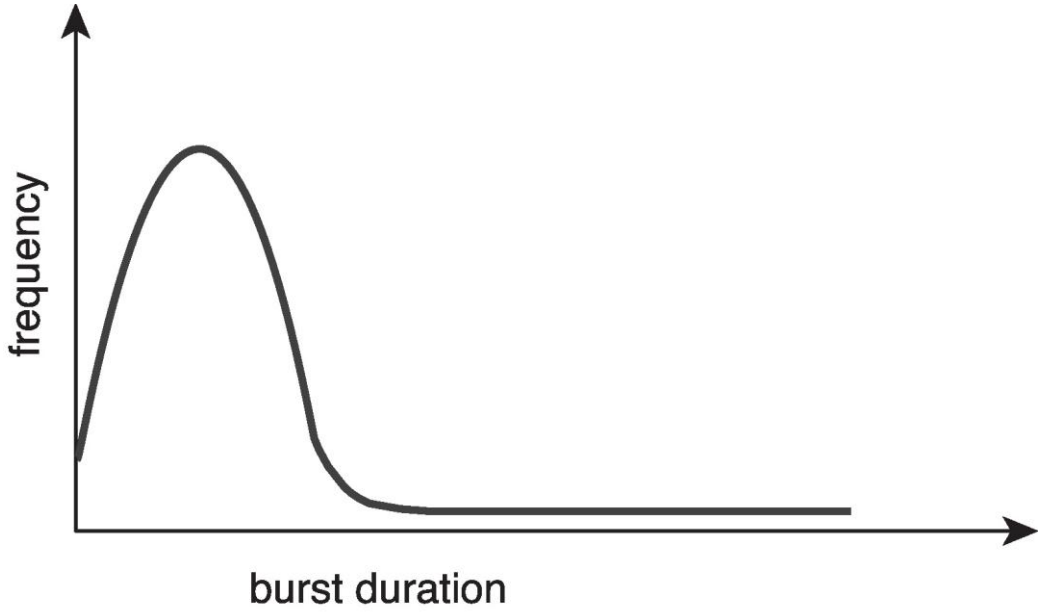
```
public static void main(Strings[] args) {
    int sum,input=1;
    Scanner scan = new Scanner(System.in);
    while(input>0) {
            sum=0;
            input = scan.nextInt();
            for(int i=0;i<input;i++) {
                    sum=sum+i;
            }
            System.out.print("The output is ");
            System.out.println(sum);
    }
}
```



Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts



CPU becomes idle frequently unless the OS assigns a new process to run on it! ⇒ CPU scheduling is necessary

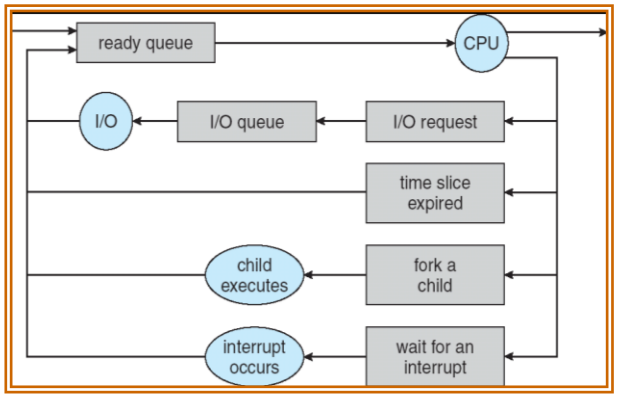
CPU Scheduler

The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them

Queue may be ordered in various ways — head of queue might be selected

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



No OS involvement — processor voluntarily release CPU resource

Scheduling under 1 and 4 is **nonpreemptive**

All other scheduling is **preemptive** — OS deprives resources from the process for assigning CPU to other process.

Consider access to shared data

Consider preemption while in kernel mode

Consider interrupts occurring during crucial OS activities

Dispatcher

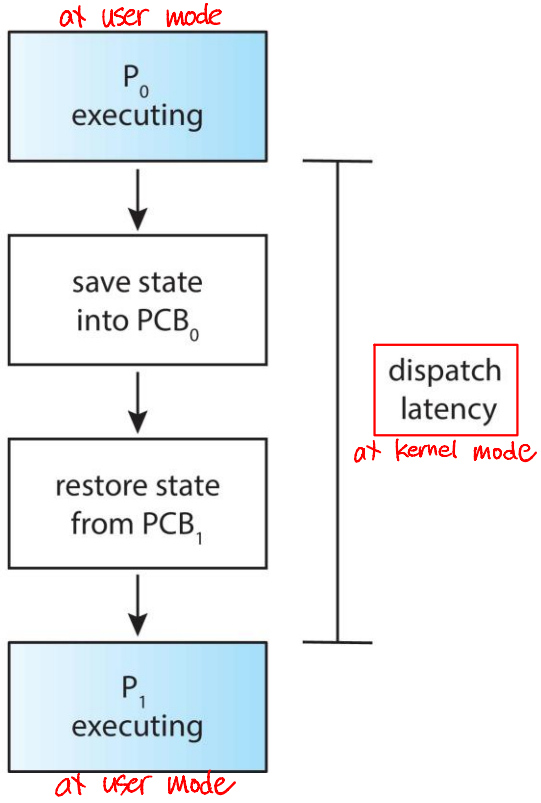
essential module to perform CPU scheduling implementing in OS

Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:

executed by dispatcher in kernel

- 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 Criteria

focusing on system performance

- **CPU utilization** – keep the CPU as busy as possible (100 % is ideal. amount of time the CPU is in use.)
- **Throughput** – # of processes that complete their execution per time unit (increase throughput ⇒ increase performance of system)
- **Turnaround time** – amount of time to execute a particular process (as small as possible)
- **Waiting time** – amount of time a process has been waiting in the ready queue (as small as possible)
- **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) (focusing on "user" interactiveness convenience)

time: resource request ~ assigning resource — as small as possible

Scheduling Algorithm Optimization Criteria

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

First- Come, First-Served (FCFS) Scheduling

Non-preemptive

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.)

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 ⇒ entire waiting time increases.

Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

Non-preemptive

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

CPU burst

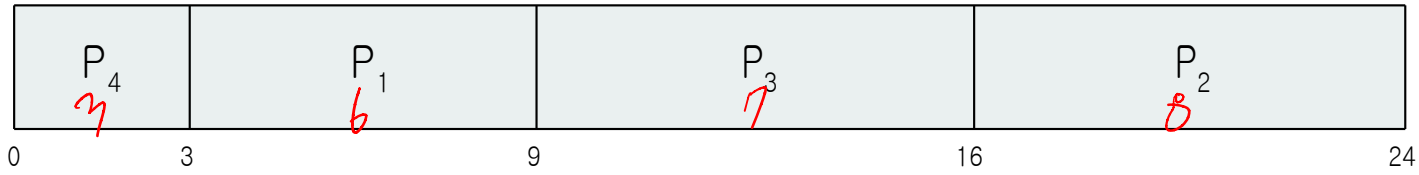
Could ask the user

↳ inconvenience

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

Determining Length of Next CPU Burst

Can only estimate the length – should be similar to the previous one (assumption)

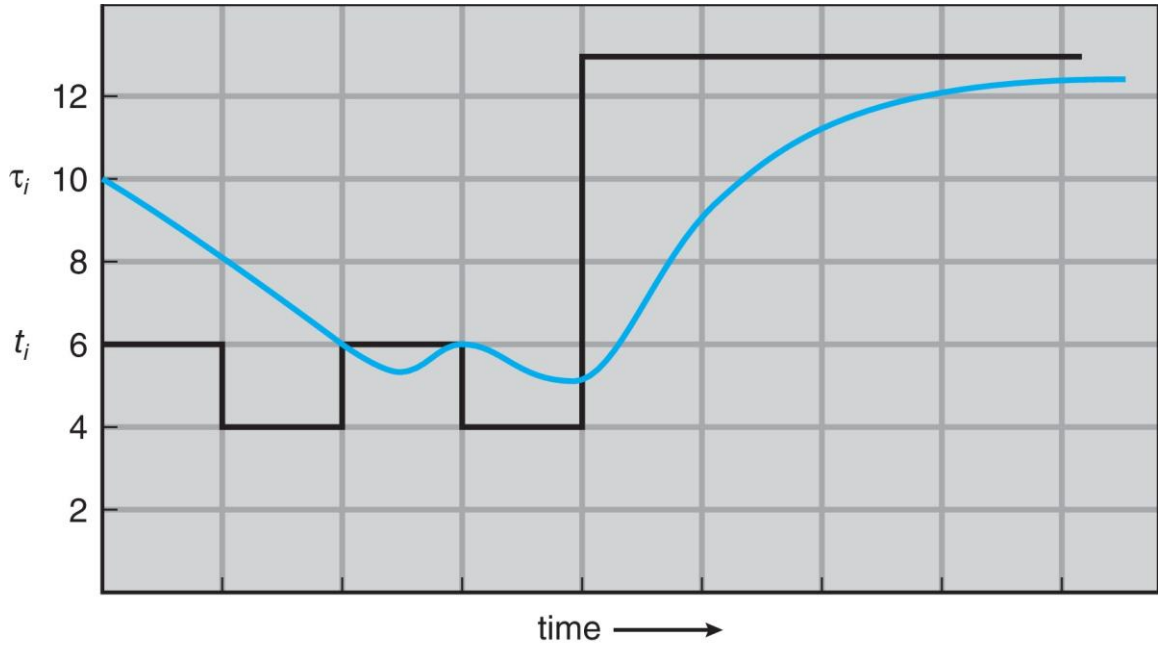
Then pick process with shortest predicted next CPU burst

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. τ_{n+1} = predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
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) $\alpha = 1/2$	10	8	6	6	5	9	11	12	...

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

Examples of Exponential Averaging

$\alpha = 0$

$\tau_{n+1} = \tau_n$

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:

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

Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

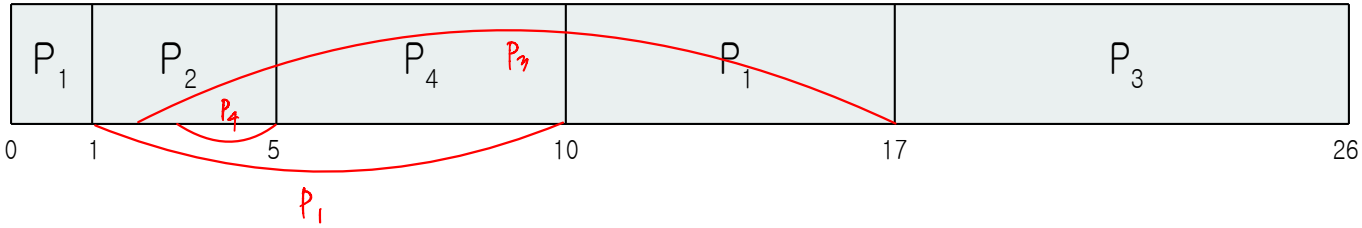
Shortest-remaining-time-first

Preemptive version of SJF 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)

P1 P2 P3 P4

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. No process waits more than $(n\text{-}1)q$ time units.

Timer interrupts every quantum to schedule next process

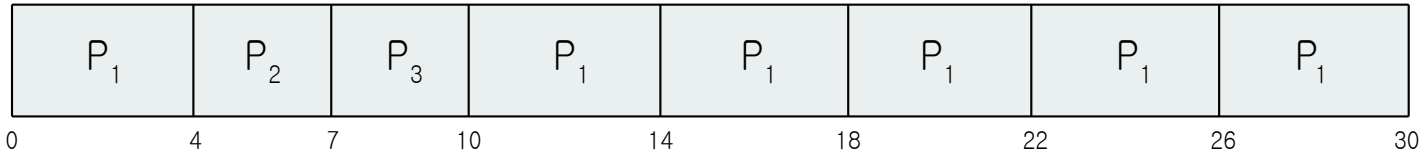
Performance

- q large $\Rightarrow$ FIFO (FCFS)
- q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high
  ∵ frequent context switching

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

The Gantt chart is: $q=4$

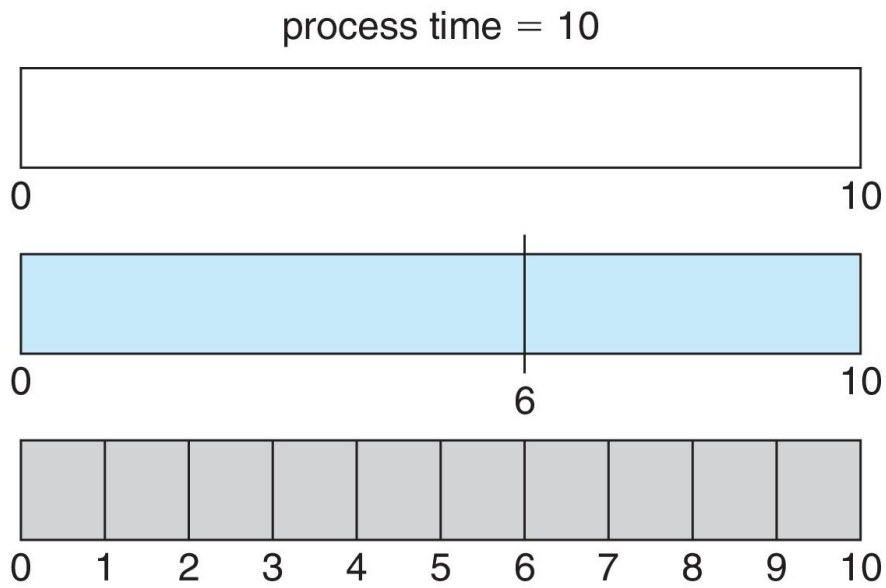


Typically, higher average turnaround than SJF, but better ***response*** $(n-1)q$

q should be large compared to context switch time

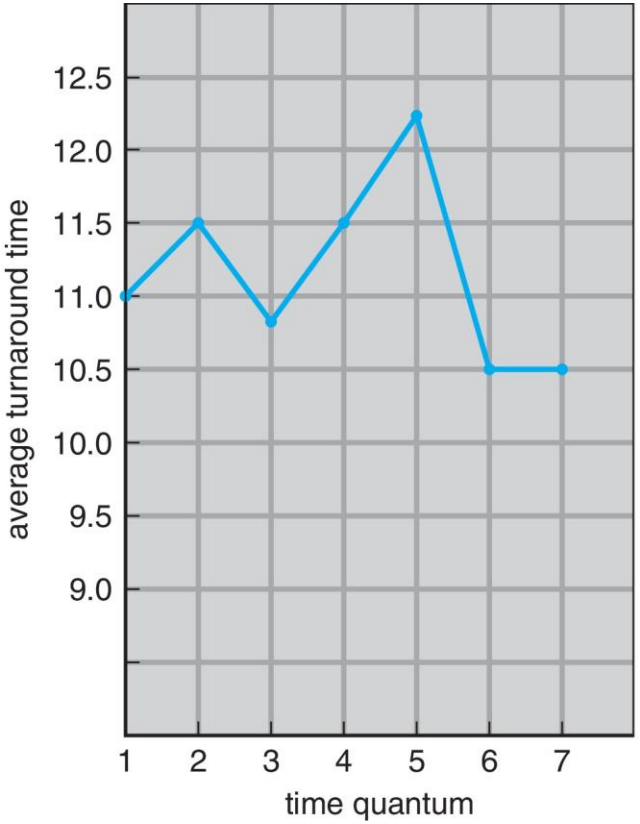
q usually 10ms to 100ms, context switch < 10 usec

Time Quantum and Context Switch Time



quantum	context switches
12	0
6	1
1	9

Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

80% of CPU bursts
should be shorter than q

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 : flexible, better performance
- Nonpreemptive

SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

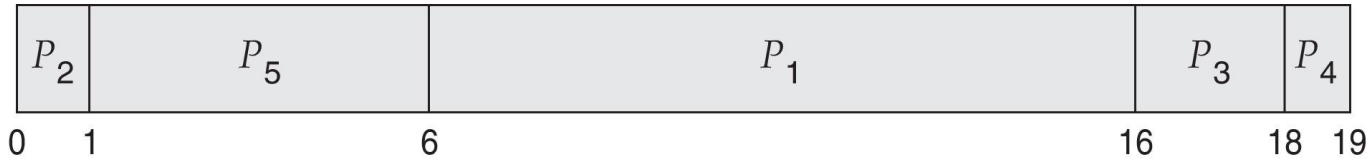
Problem ≡ **Starvation** – low priority processes may never execute

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



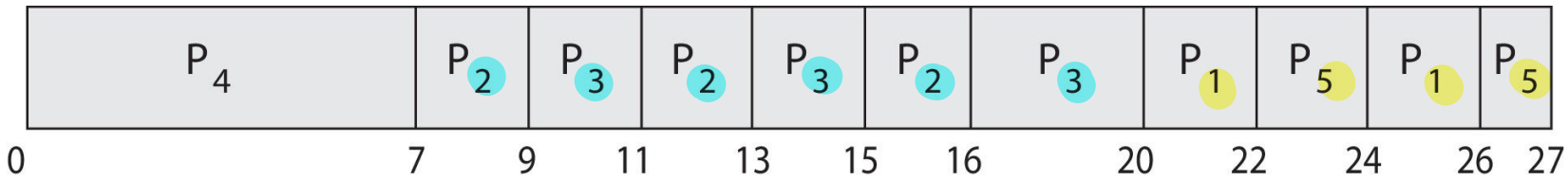
Average waiting time = 8.2 msec

Priority Scheduling w/ Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin

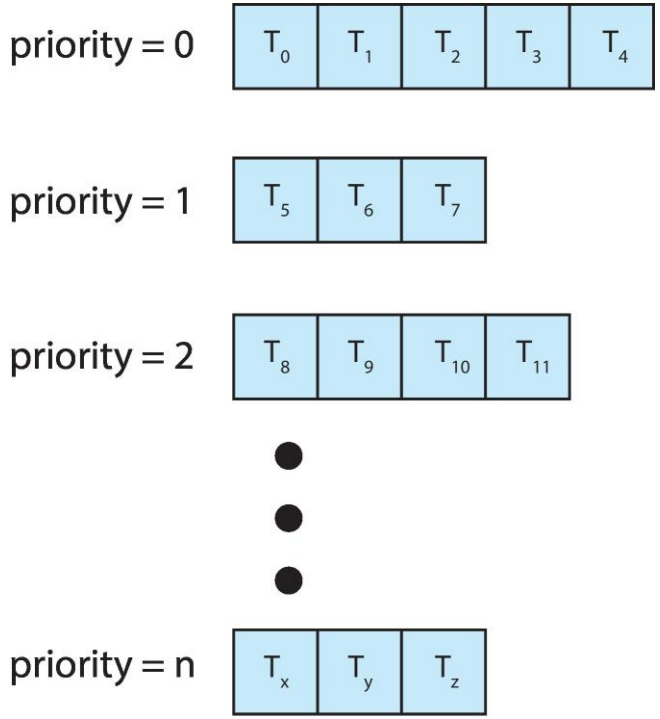
  Gantt Chart with 2 ms time quantum



Multilevel Queue

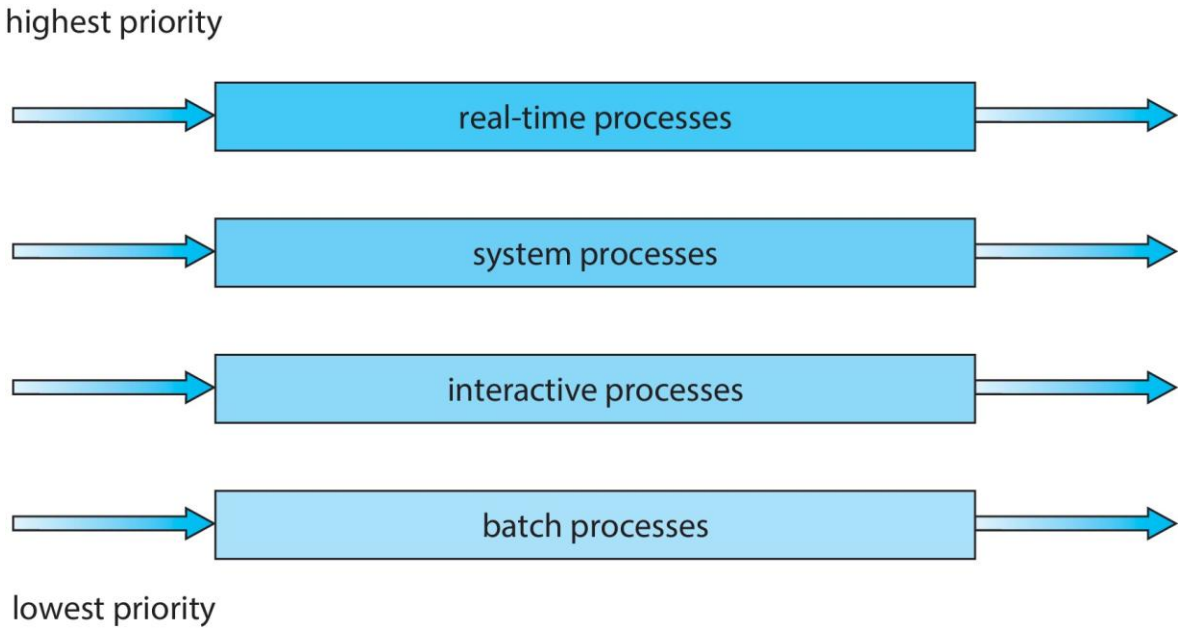
With priority scheduling, have separate queues for each priority.

Schedule the process in the highest-priority queue!



Multilevel Queue

Prioritization based upon process type



Multilevel Feedback Queue

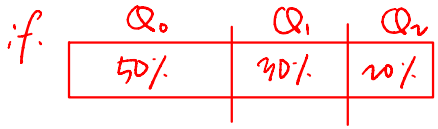
unlike multiple queue

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

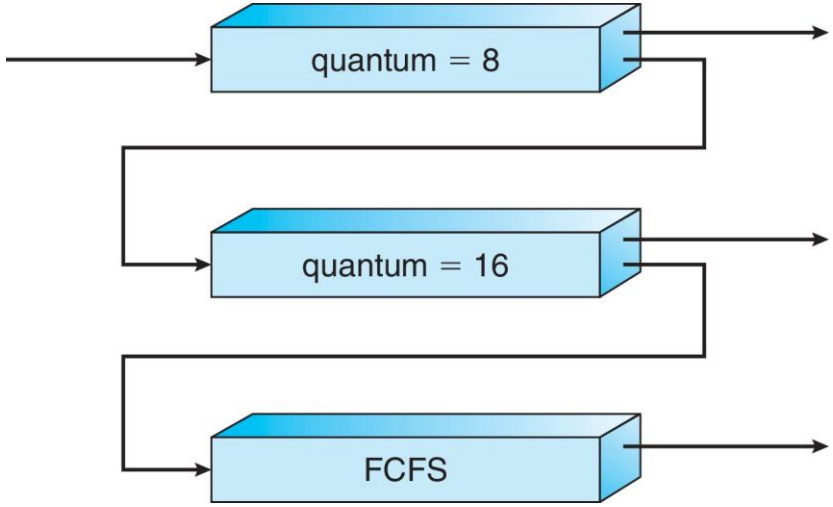
Example of Multilevel Feedback Queue



Three queues:

priority

- #1 Q_0 – RR with time quantum 8 milliseconds
- #2 Q_1 – RR time quantum 16 milliseconds
- #3 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

Thread Scheduling (1/4)

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 (Light Weight Process)

- Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Typically done via priority set by programmer

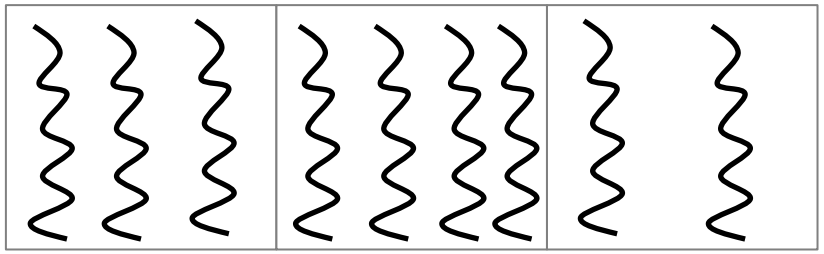
Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

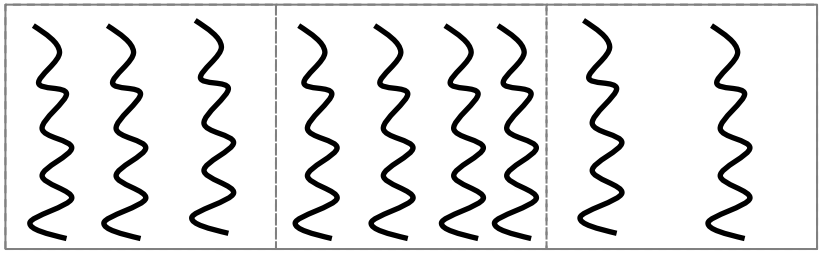
API allows specifying either PCS or SCS during thread creation

- PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
- PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling

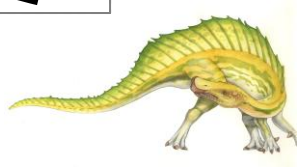
Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM



PCS (Threads compete separately in each process)



SCS (Threads compete all together)



Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* 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)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Multiple-Processor Scheduling (1/8)

CPU scheduling more complex when multiple CPUs are available

Multiprocess may be any one of the following architectures:

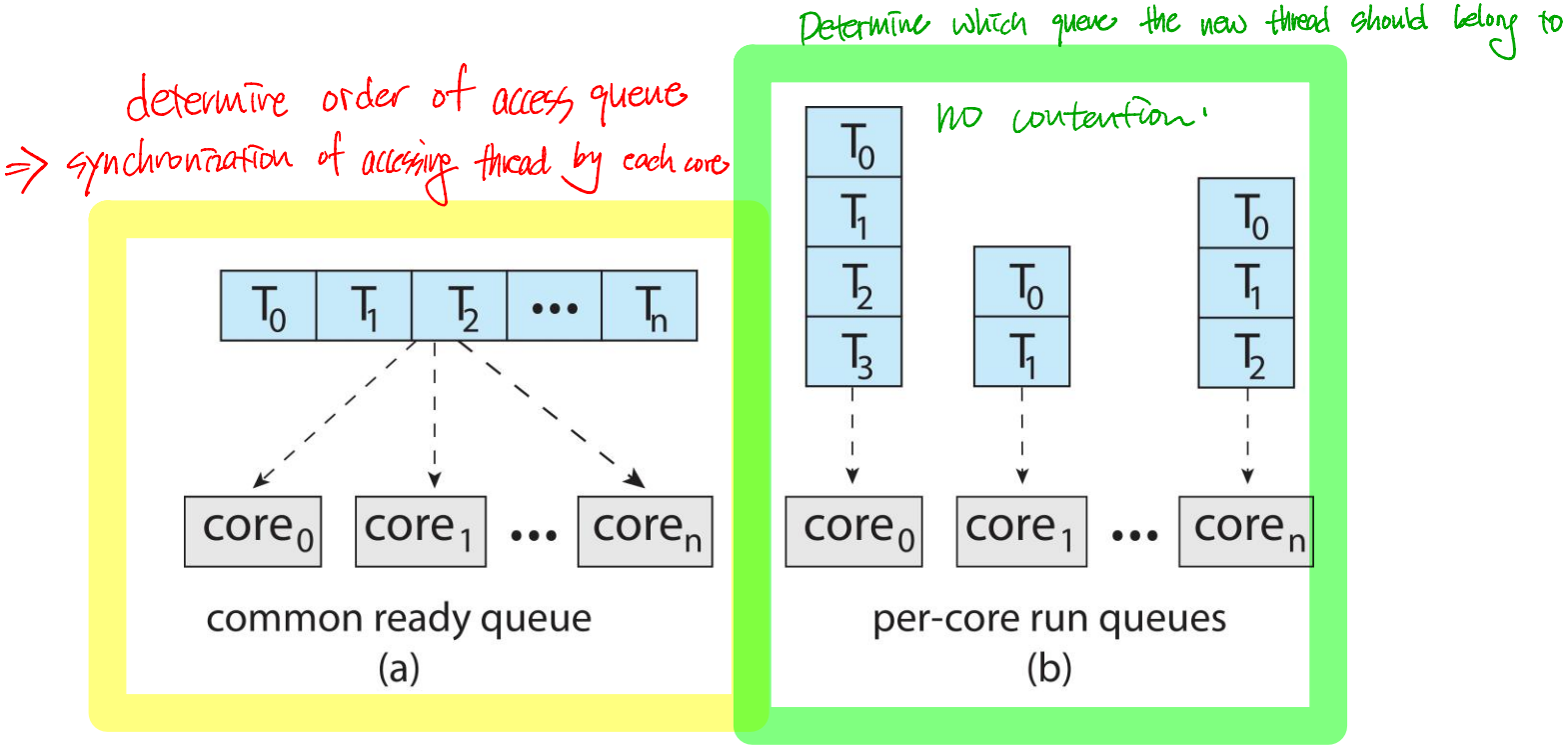
- Multicore CPUs
- Multithreaded cores
- NUMA systems
- Heterogeneous multiprocessing

Multiple-Processor Scheduling (2/8)

Symmetric multiprocessing (SMP) is where each processor is self scheduling.

All threads may be in a common ready queue (a)

Each processor may have its own private queue of threads (b)



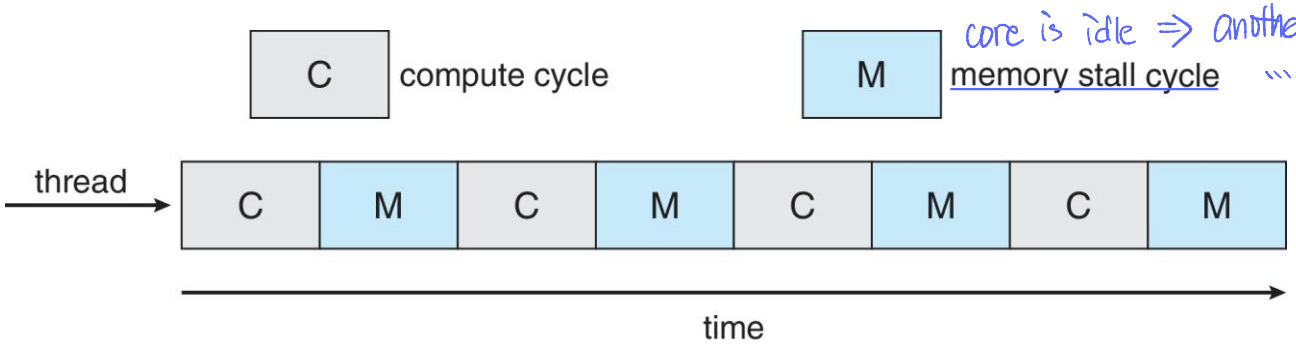
Multicore Processors

Recent trend to place multiple processor cores on same physical chip

Faster and consumes less power

Multiple threads per core also growing

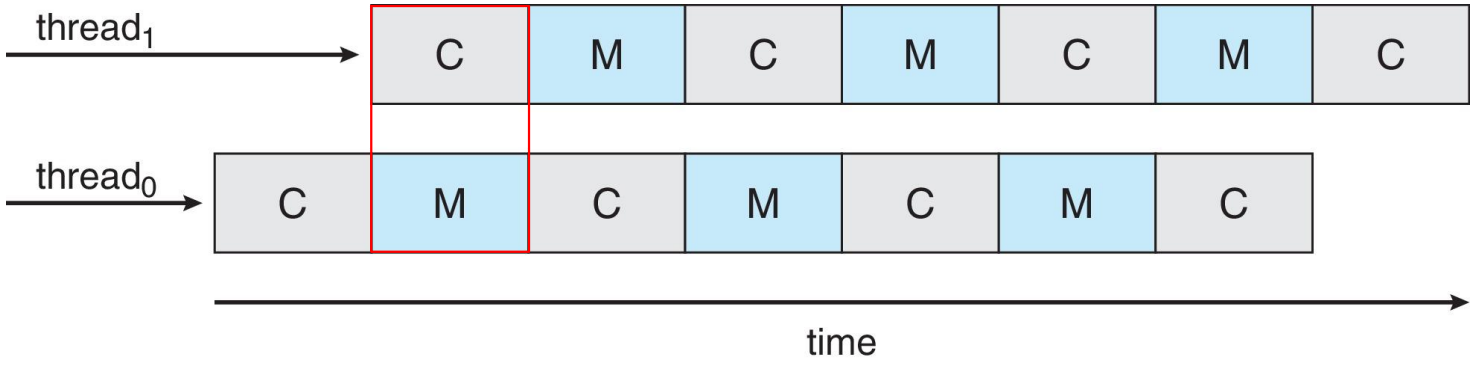
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens



Multithreaded Multicore System

Each core has > 1 hardware threads.

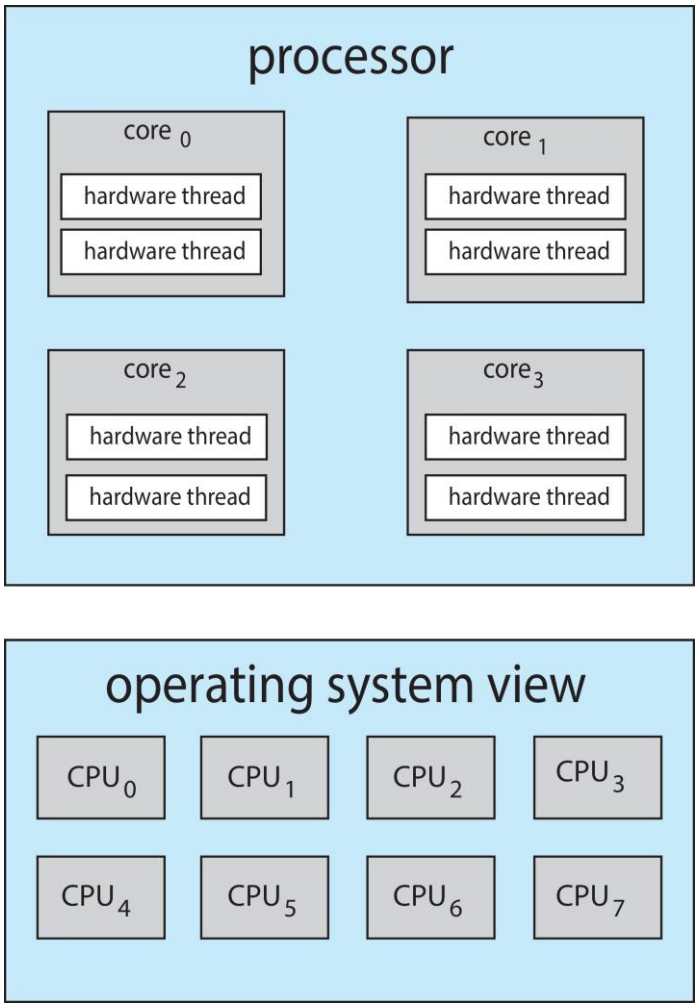
If one thread has a memory stall, switch to another thread!

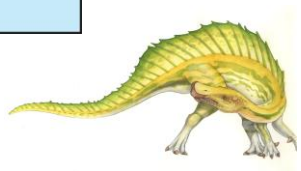


Multithreaded Multicore System

Chip-multithreading (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)

On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



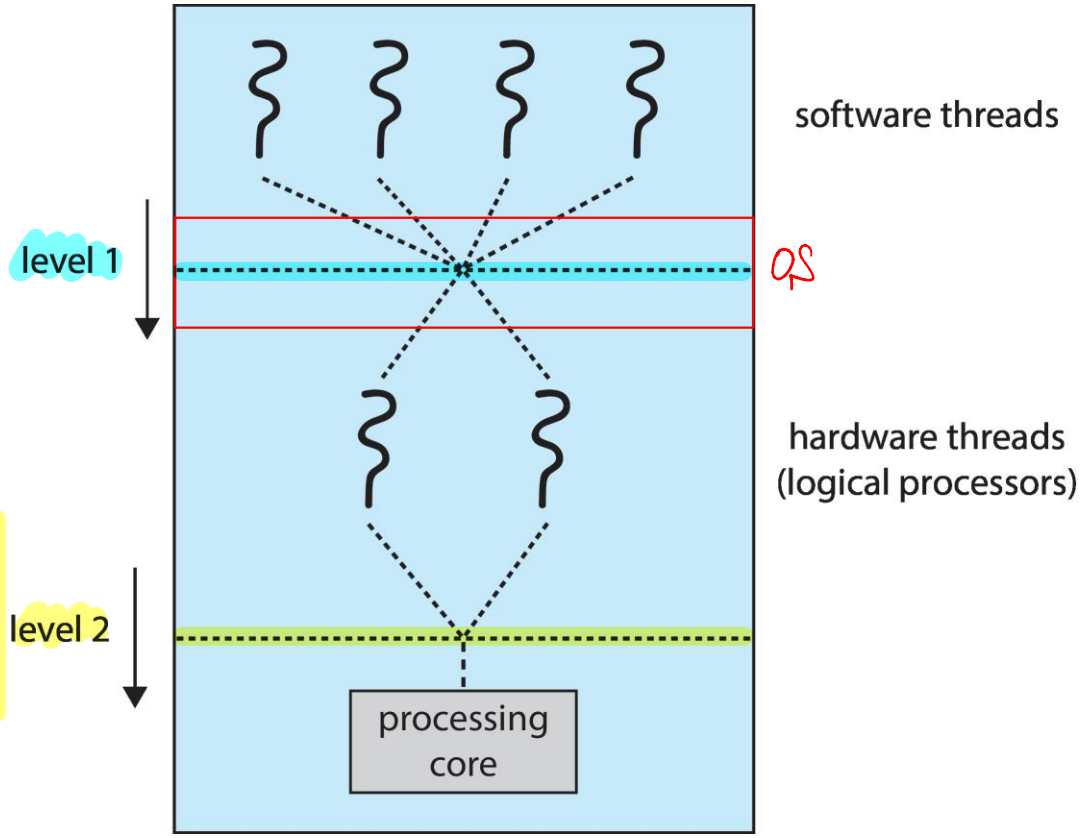


Multithreaded Multicore System

Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU

2. How each core decides which hardware thread to run on the physical core.



Multiple-Processor Scheduling – Load Balancing

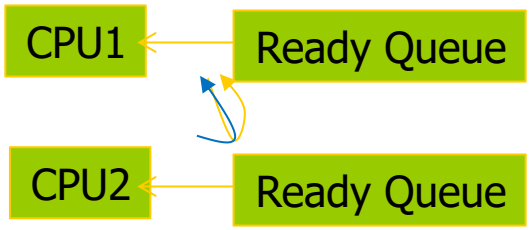
If SMP, need to keep all CPUs loaded for efficiency

Load balancing attempts to keep workload evenly distributed

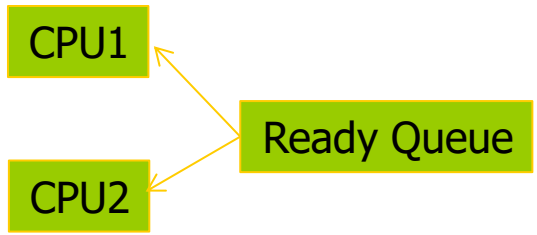
Push migration – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs by overload CPU

Pull migration – idle processors pulls waiting task from busy processor by Idle CPU

* Private ready queue with pull migration



* Shared ready queue



Multiple-Processor Scheduling – Processor Affinity

thread is executed only on one processor, remained data in cache make performance improve.

When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.

We refer to this as a thread having affinity for a processor (i.e. "processor affinity")

due to migration

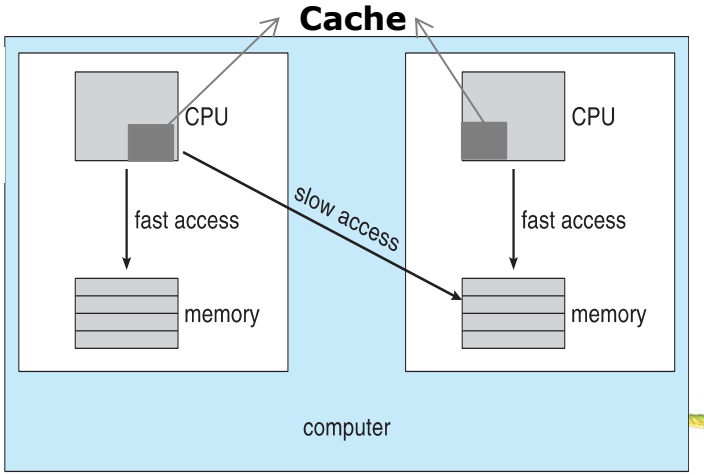
Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.

Soft affinity – the operating system attempts to keep a thread running on the same processor, but no guarantees.

Hard affinity – allows a process to specify a set of processors it may run on.

NUMA (Non-Uniform Memory Access) architecture and CPU scheduling

time it takes to access memory is all different.
... migration have heavy overhead.
So. process affinity is important.



Real-Time CPU Scheduling (1/8)

RTOS (Real Time Operating Systems)

- Tasks should be run in a deadline (constant)
- Uses Real-Time CPU Scheduling

Can present obvious challenges

- **Soft real-time systems** – Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline

Real-Time CPU Scheduling (2/8)

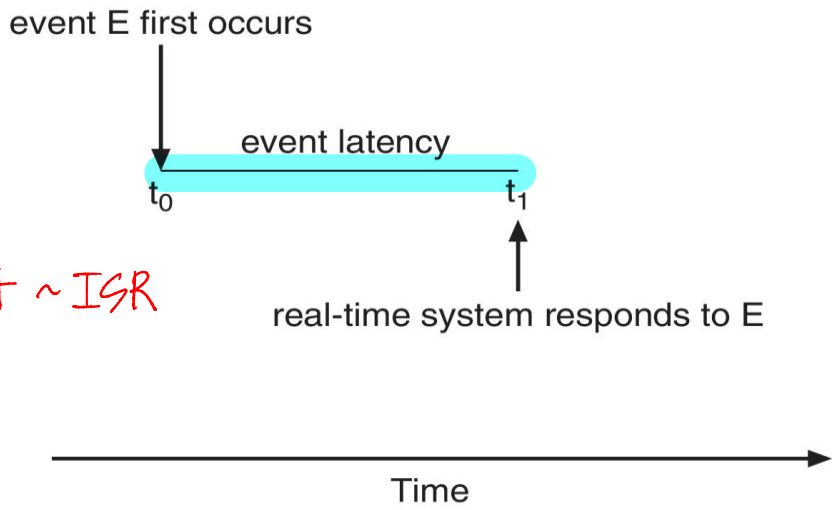
Event latency – the amount of time that elapses from when an event occurs to when it is serviced.

Two types of latencies affect performance

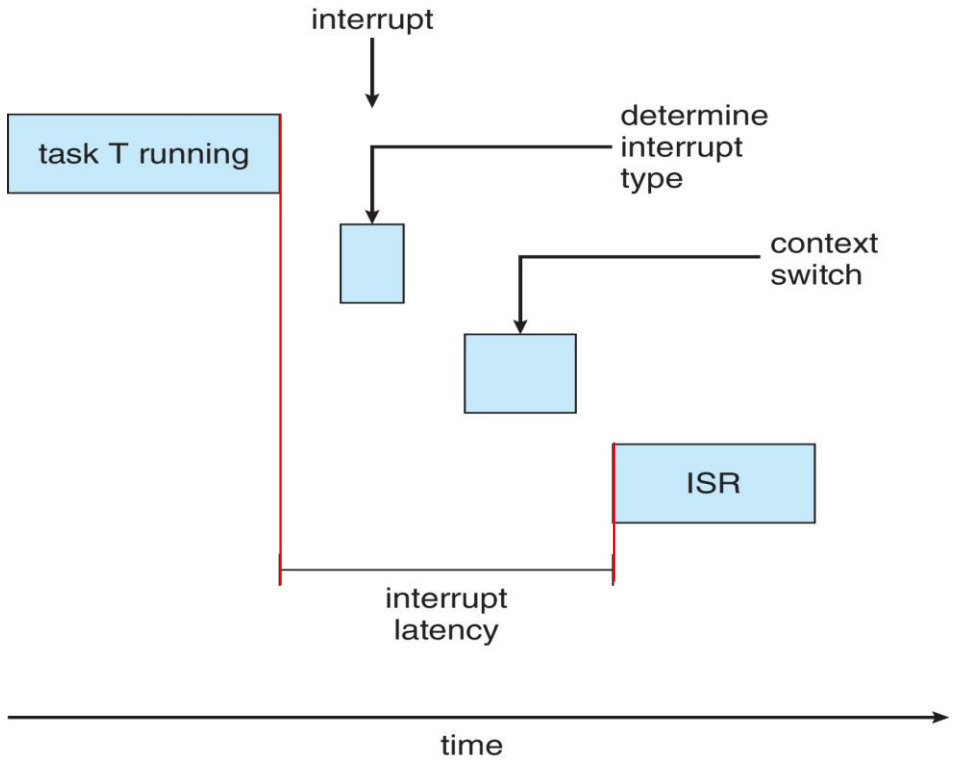
1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
2. **Dispatch latency** – time for schedule to take current process off CPU and switch to another

Interrupt ~ ISR

ISR — Real Time Task



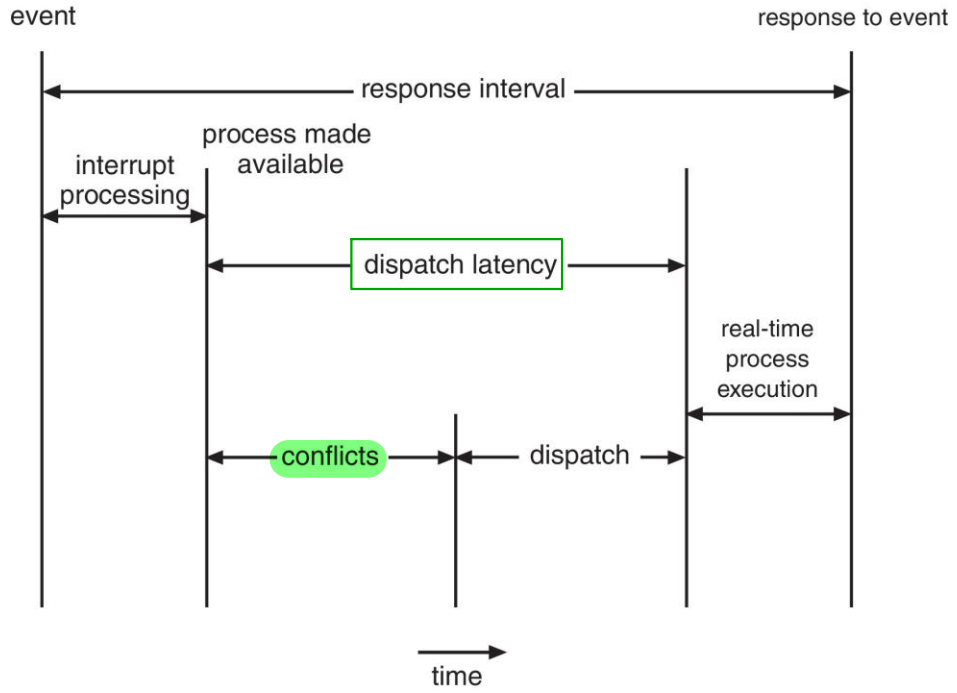
Interrupt Latency



Dispatch Latency

Conflict phase of dispatch latency:

1. Preemption of any process running in kernel mode
2. Release by low-priority process of resources needed by high-priority processes



Priority-based Scheduling

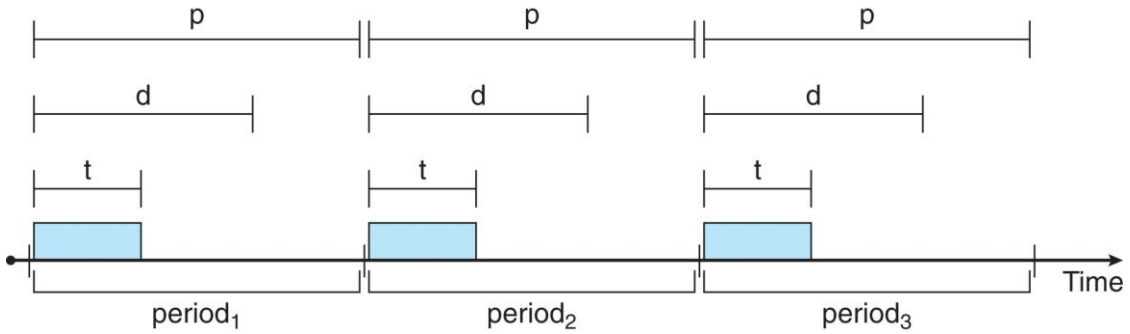
For real-time scheduling, scheduler must support preemptive, priority-based scheduling

- But 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*
- $0 \le t \le d \le p$
- **Rate** of periodic task is $1/p$



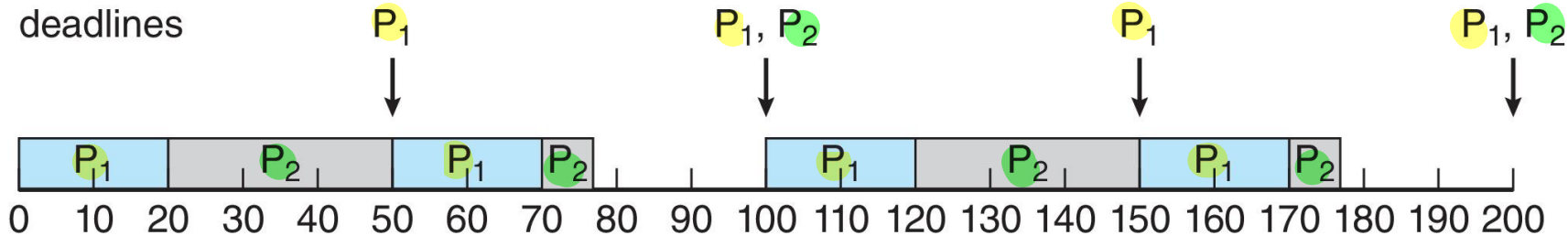
Rate Montonic Scheduling

A priority is assigned based on the inverse of its period

Shorter periods = higher priority;

Longer periods = lower priority

P_1 is assigned a higher priority than P_2.

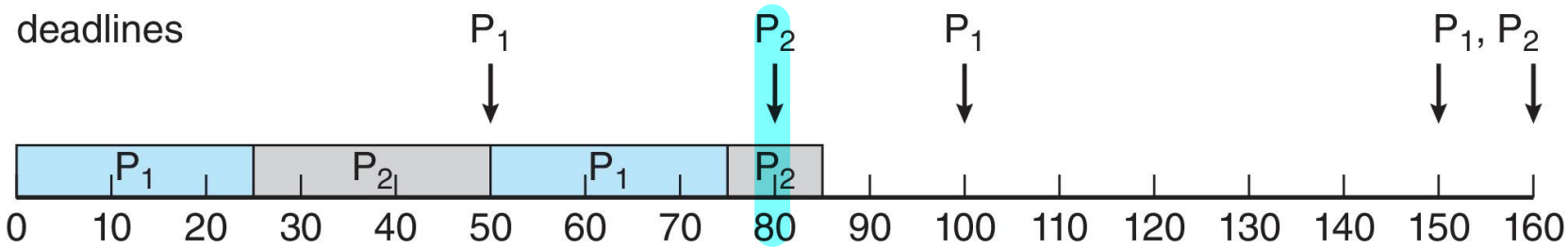


P_1: $p=50$, $t=20$

P_2: $p=100$ $t=35$

Missed Deadlines with Rate Monotonic Scheduling

Process P2 misses finishing its deadline at time 80



P_1: $P = 50$ $t = 25$ $25/50 = 50\%$

P_2: $P = 80$ $t = 35$ $35/80 = 43.75\%$

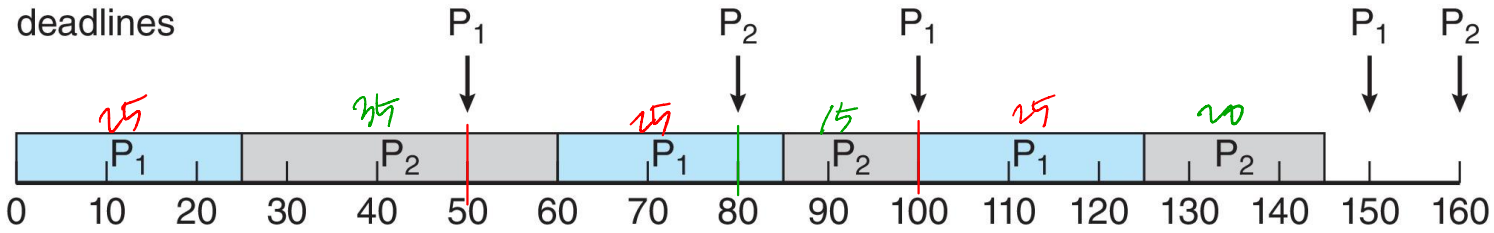
⊕ < 100% ⇒ possible to complete until deadline

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



P_1: $P = 50$ $t = 25$ $25/50 = 50\%$

P_2: $P = 80$ $t = 35$ $35/80 = 43.75\%$

Operating System Examples

Linux scheduling

Windows scheduling

Solaris scheduling

Linux Scheduling Through Version 2.5

Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm

Version 2.5 moved to 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 **nice** value from 100 to 140
- Map into global priority with numerically lower values indicating higher priority
- Higher priority gets larger q
- Task run-able as long as time left in time slice (**active**)
- If no time left (**expired**), not run-able until all other tasks use their slices
- All run-able tasks tracked in per-CPU **runqueue** data structure
  - Two priority arrays (active, expired)
  - Tasks indexed by priority
  - When no more active, arrays are exchanged
- Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

Completely Fair Scheduler (CFS)

Scheduling classes

- Each has specific priority
- Scheduler picks highest priority task in highest scheduling class
- Rather than quantum based on fixed time allotments, based on proportion of CPU time
- 2 scheduling classes included, others can be added
  1. default
  2. real-time

Quantum calculated based on **nice value** from -20 to +19

- Lower value is higher priority
- Calculates **target latency** – interval of time during which task should run at least once
- Target latency can increase if say number of active tasks increases

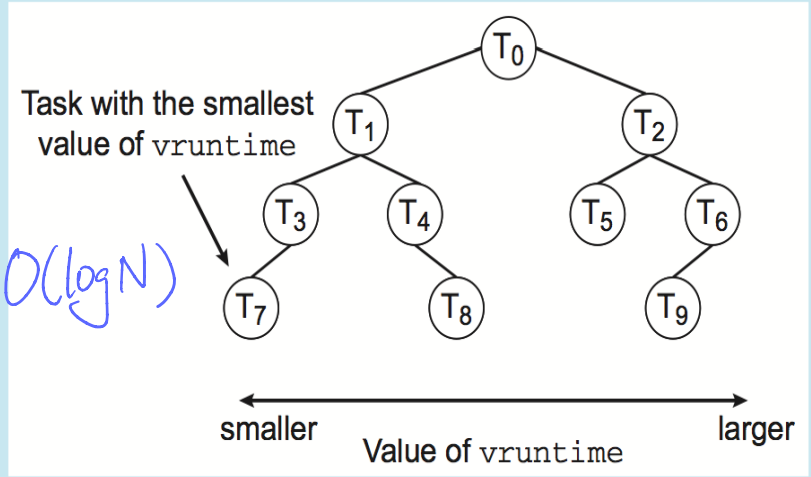
CFS scheduler maintains per task **virtual run time** in variable **`vruntime`**

- 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

To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

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.)

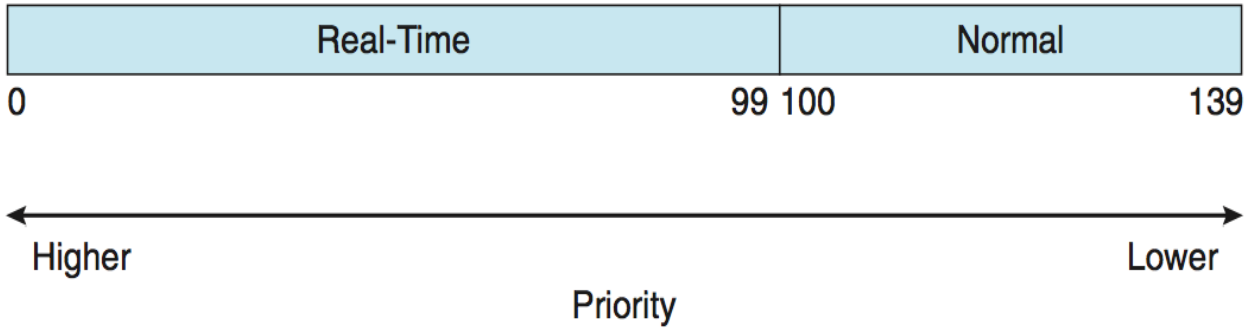
Real-time scheduling according to POSIX.1b

- 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



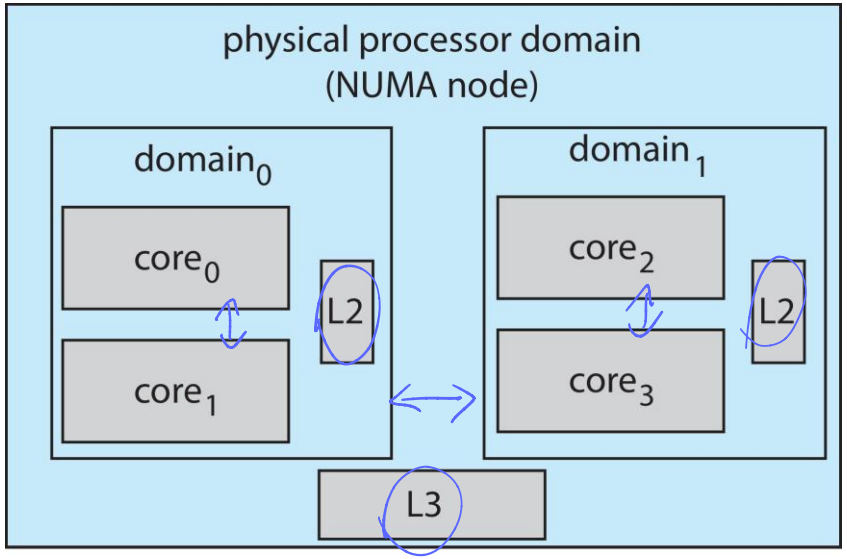
Linux Scheduling (Cont.)

Linux supports load balancing, but is also NUMA-aware. (try to keep affinity)

Scheduling domain is a set of CPU cores that can be balanced against one another.

Domains are organized by what they share (i.e. cache memory.) Goal is to keep threads from migrating between domains.

Load balancing in the same domain, breaking processor affinity is minimized.



Windows Scheduling

Windows uses priority-based preemptive scheduling

Highest-priority thread runs next

Dispatcher is scheduler

Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread

Real-time threads can preempt non-real-time

32-level priority scheme

Variable class is 1-15, **real-time class** is 16-31

Priority 0 is memory-management thread

Queue for each priority

If no run-able thread, runs **idle thread**

Windows Priority Classes

Win32 API identifies several priority classes to which a process can belong

- REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
- All are variable except REALTIME

A thread within a given priority class has a relative priority

- TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE

Priority class and relative priority combine to give numeric priority

Base priority is NORMAL within the class

If quantum expires, priority lowered, but never below base

Windows Priority Classes (Cont.)

If wait occurs, priority boosted depending on what was waited for

Foreground window given 3x priority boost

Windows 7 added **user-mode scheduling** (**UMS**)

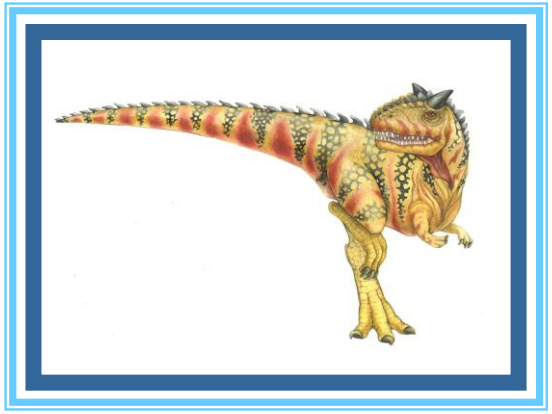
- Applications create and manage threads independent of kernel
- For large number of threads, much more efficient
- UMS schedulers come from programming language libraries like C++ **Concurrent Runtime** (ConcRT) framework

Windows Priorities

Priority classes

Relative 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

End of Chapter 5



may increase. This algorithm is called *scheduling*.

5.4 Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

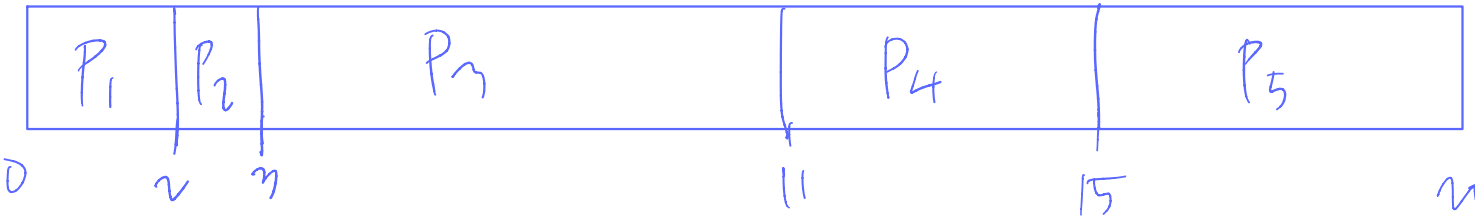
a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a larger priority number implies a higher priority), and RR (quantum = 2).

b. What is the turnaround time of each process for each of the scheduling algorithms in part a?

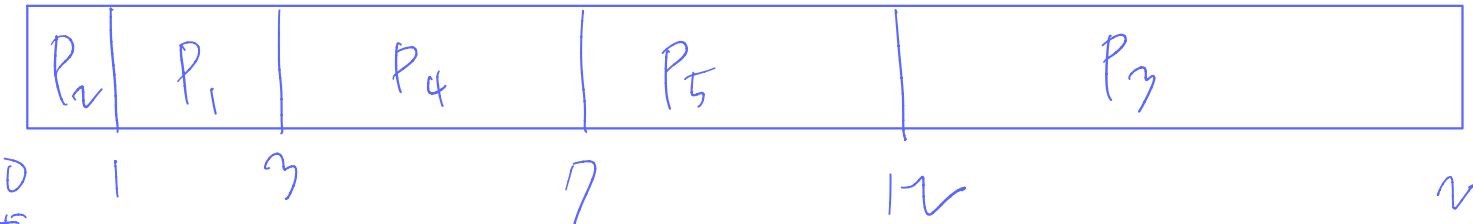
c. What is the waiting time of each process for each of these scheduling algorithms?

d. Which of the algorithms results in the minimum average waiting time (over all processes)?

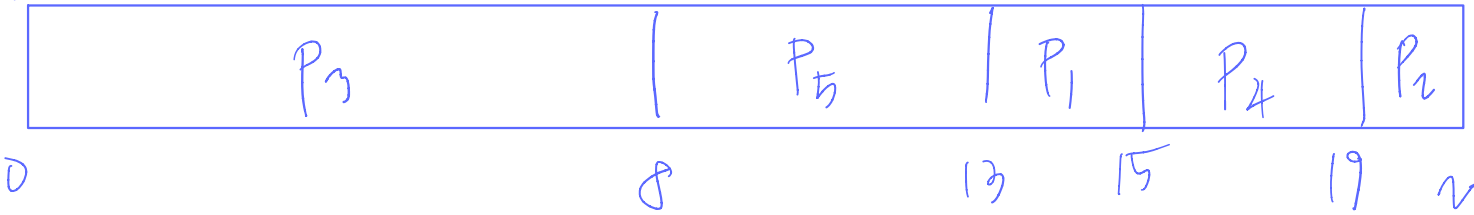
a) FCFS



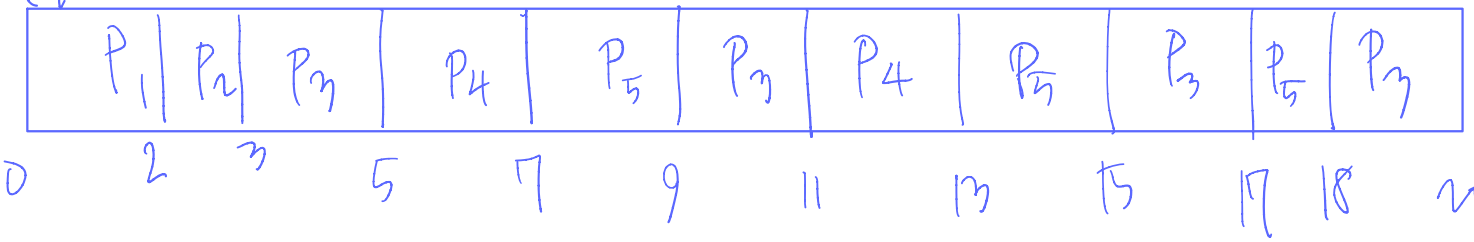
SJF



non-preemptive priority



RR (q=2)



b. turnaround time

	P_1	P_2	P_3	P_4	P_5
FCFS :	2	7	11	15	20
SJF :	3	1	20	7	12
nonpreemptive priority :	15	20	8	9	13
RR (q=2) :	2	7	20	13	18

c. waiting time

	P_1	P_2	P_3	P_4	P_5	avg
FCFS :	0	2	3	11	15	6.2
SJF :	1	0	12	3	7	4.6
nonpreemptive priority :	13	19	0	15	8	11
RR (q=2) :	0	2	12	9	13	7.2

d. SJF.