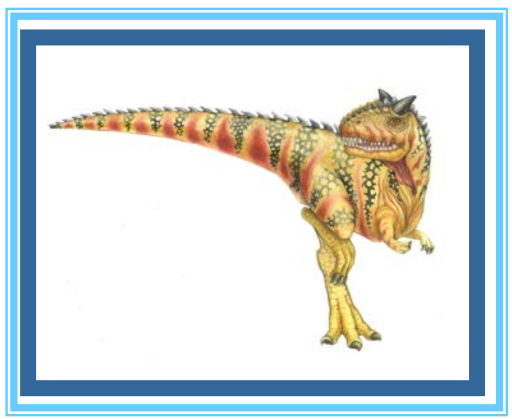
Chapter 5: Process Scheduling

By Worawut Srisukkham   Updated By Dr. Varin Chouvatut



Chapter 5: Process Scheduling

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

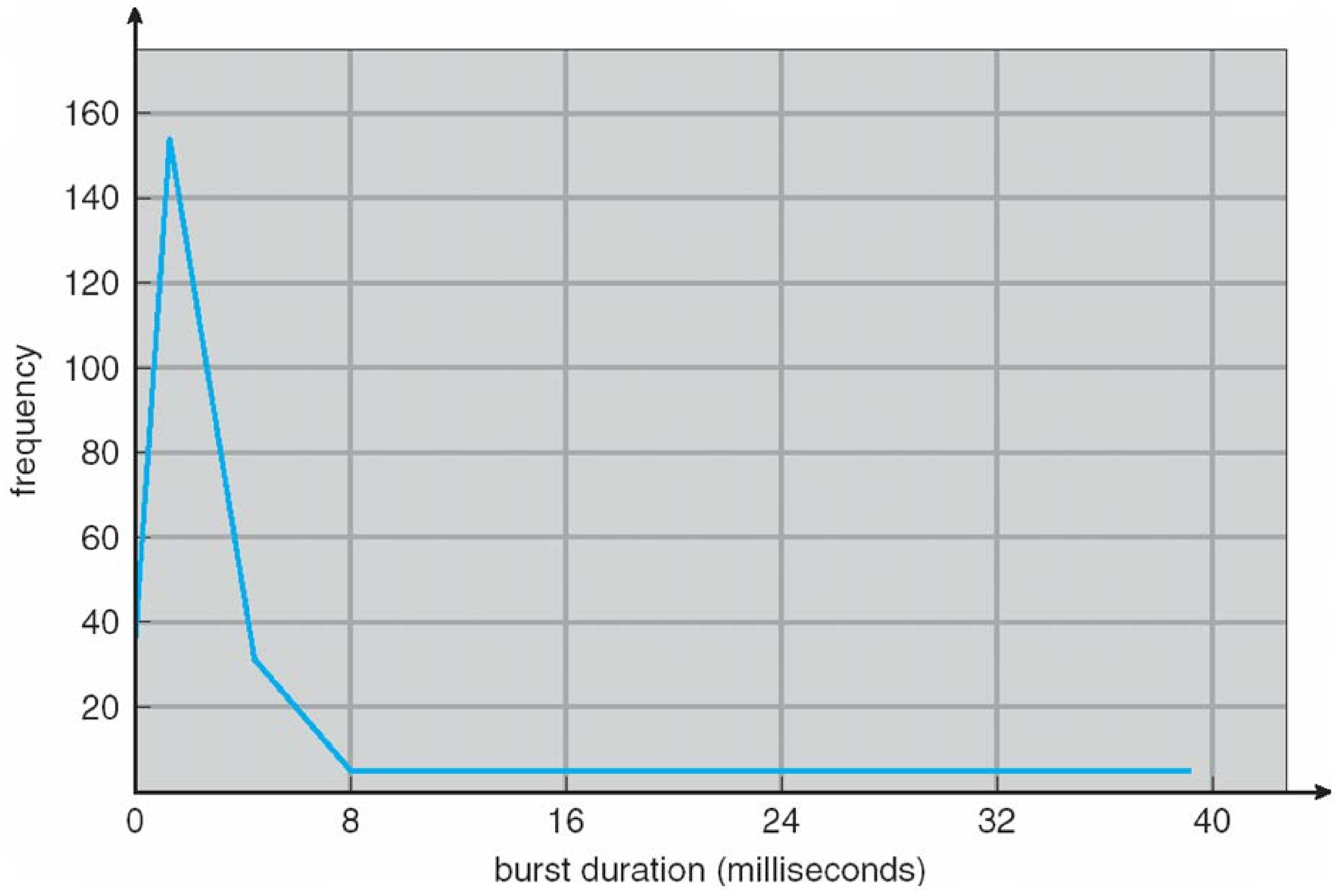
Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system

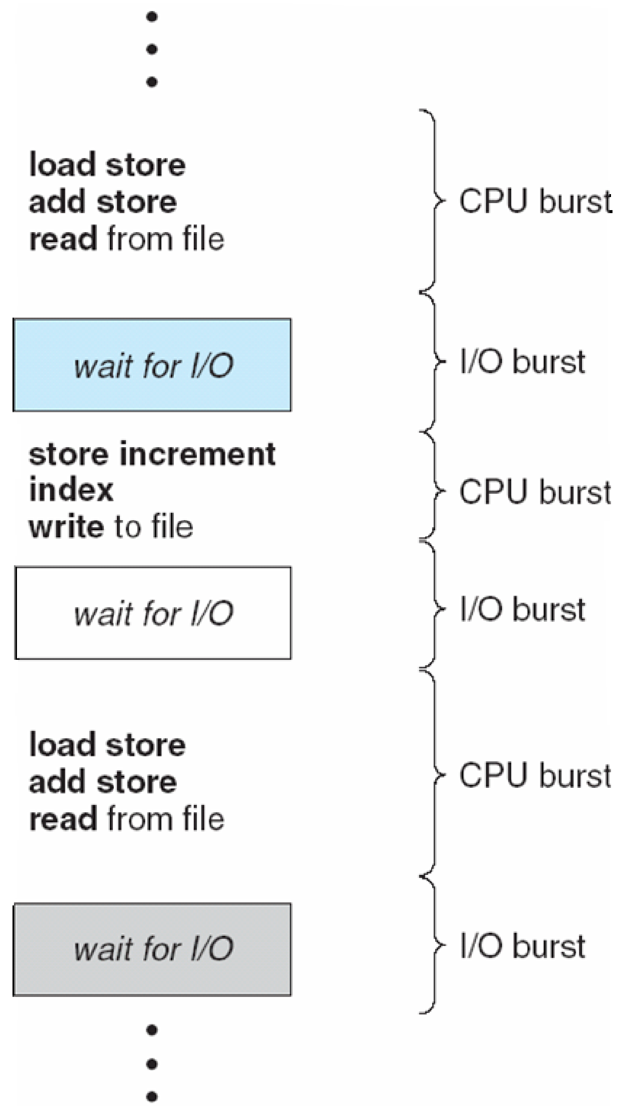
Basic Concepts

- Maximum CPU utilization is obtained with multiprogramming
- CPU-I/O Burst Cycle – Process execution consists of a *cycle* of CPU execution and I/O wait. Processes alternate between these 2 states.
- **CPU-burst** distribution

Histogram of CPU-burst Times



Alternating Sequence of CPU and I/O 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 state
  4. Terminates
- Scheduling schemes under circumstances 1 and 4 are **nonpreemptive**
- All other schemes are **preemptive**

nonpreemptive: ไม่สามารถแทรกการทำงานกลางคันขณะที่ CPU กำลังประมวลผลโปรเซส
preemptive: แทรกการทำงานกลางคันขณะที่ CPU กำลังประมวลผลโปรเซส

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler (or the CPU scheduler); this function involves:
  - Switching context
  - Switching to user mode
  - Jumping to the proper location in the user program to restart that program
- **Dispatch latency** – the time it takes for the dispatcher to stop one process and start another running

Dispatcher: ตัวส่งข่าวสารไปยัง state อื่น , ตัวส่งต่อ

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – the number of processes that are completed 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)

Scheduling Algorithm: Optimization Criteria

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

There are many different CPU-scheduling algorithms:

1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Priority Scheduling
4. Round-Robin Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time (ms)
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:

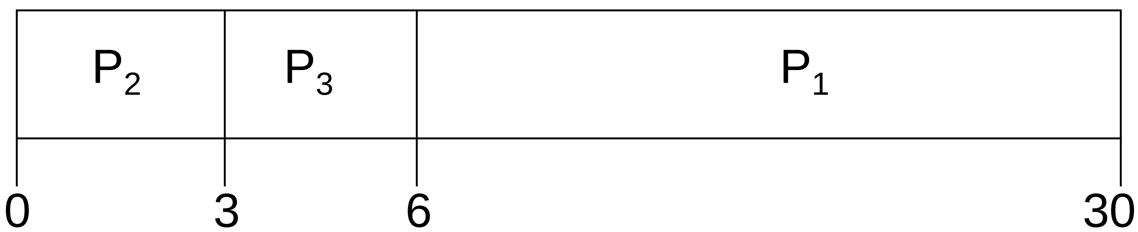
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Turnaround time : P_1 = 24; P_2 = 27; P_3 = 30

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
- Turnaround time : $P_1 = 30$; $P_2 = 3$; $P_3 = 6$
- Much better than previous case
- A ***Convoy effect*** – short processes stand behind a long process

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 first
- Two schemes:
  - *nonpreemptive* – 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 know as the Shortest-Remaining-Time-First (SRTF).
- 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

arrive : มาถึง

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

หมายเหตุ ทุก **Process** มาถึงเวลาเดียวกัน

- SJF scheduling chart

P_4	P_1	P_3	P_2
0–3	3–9	9–16	16–24

P_1 P_2 P_3 P_4

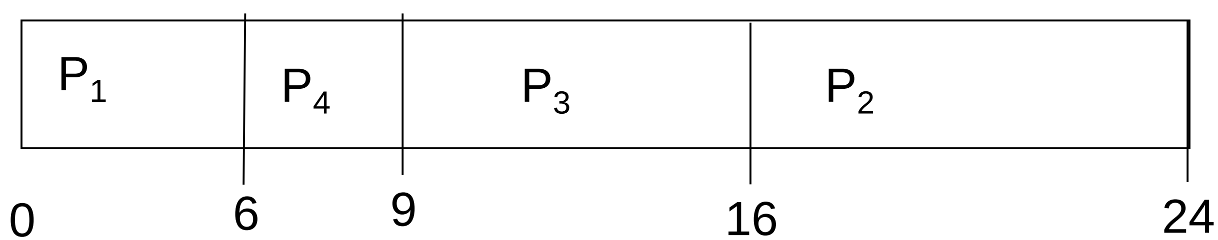
- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7
- Turnaround Time : P_1 = 9; P_2 = 24; P_3 = 16; P_4 = 3;

Example of nonpreemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

หมายเหตุ *เวลามาถึงของแต่ละ* ***Process*** *ไม่เท่ากัน*

- SJF scheduling chart : แบบ nonpreemptive ไม่สามารถแทรกการทำงานกลางคันได้



- Average waiting time = (0 + 14 + 5 + 1) / 4 = 20/4 = 5 ms

(P_1: 0-0, P_2: 16-2, P_3: 9-4, P_4: 6-5)

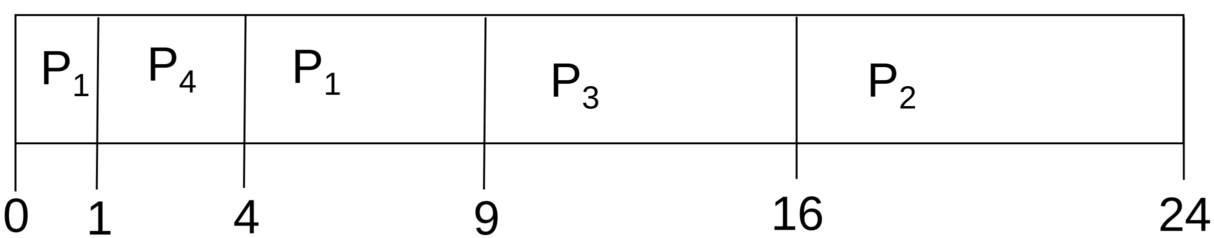
* คิด Arrival time ด้วย

Example of preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	1.0	3

** Process มาถึงเวลาไม่เท่ากัน

- SJF scheduling chart : แบบ preemptive แทรกการทำงานกลางคันได้



- Average waiting time = (3 + 14 + 5 + 0) / 4 = 22/4 = 5.5 ms
  - P_1: 4-1, P_2: 16-2, P_3: 9-4, P_4: 1-1

* คิด Arrival time ด้วย

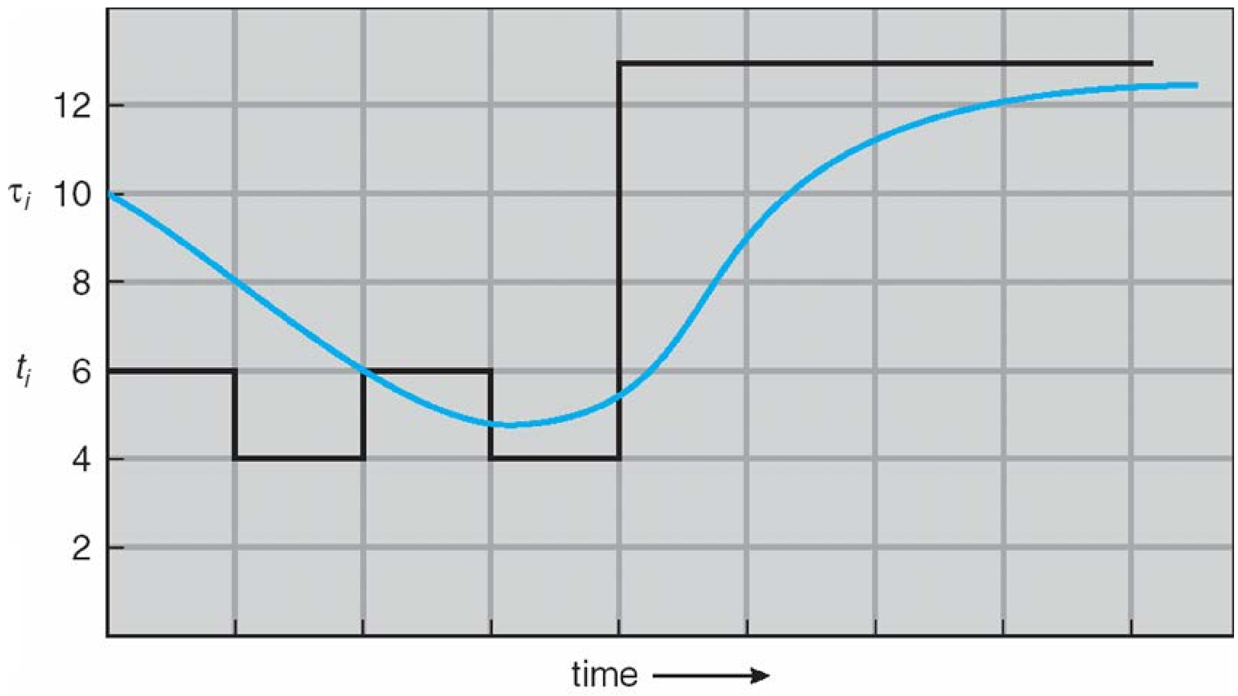
Determining Length of Next CPU Burst

เนื่องจากว่า SJF เหมาะกับการจัด Schedule แบบ Long-Term Scheduling จะไม่สามารถนำมาใช้กับ Short-Term Scheduling เพราะไม่สามารถที่จะรู้ช่วงเวลาถัดไปที่ CPU Burst จึงเกิดวิธีการต่อไปนี้ขึ้น

- Can only estimate the length
- 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. $\alpha, 0 \leq \alpha \leq 1$
  4. Define $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

α is constant or as an overall system average

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	…

$\alpha = 1/2$ and $\tau_0 = 10$

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = t_n$
  - 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

Recent : ผ่านมาเร็วๆ นี้, พึ่งผ่านมา
Predecessor: บรรพบุรุษ, ตัวที่ทำมาก่อน

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority

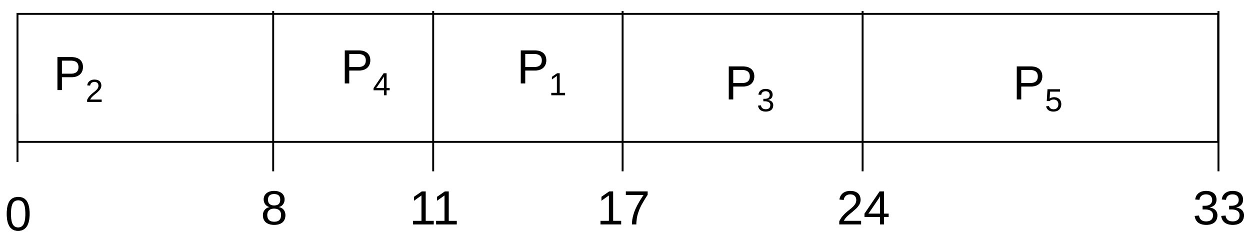
  (**smallest integer ≡ highest priority**)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never be executed
- Solution ≡ **Aging** – as time progresses, increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	6	3
P_2	8	1
P_3	7	4
P_4	3	2
P_5	9	5

หมายเหตุ ทุก **Process** มาถึงเวลาเดียวกัน

- priority scheduling chart



P_1 P_2 P_3 P_4 P_5

- Average waiting time = (11 + 0 + 17 + 8+ 24) / 5 = 12 ms
- Turnaround Time : P_1 = 17; P_2 = 8; P_3 =24; P_4 = 11; P_5 = 33;

Round-Robin (RR) Scheduling

- 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. No processes wait longer than $(n-1) \times q$ time units.
- Performance
  - *q* large $\Rightarrow$ FCFS
  - *q* small $\Rightarrow$ *q* must be large with respect to the context-switch time, otherwise overhead is too high

time quantum: ส่วนแบ่งเวลา

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

หมายเหตุ ทุก **Process** มาถึงเวลาเดียวกัน

- The Gantt chart is:

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

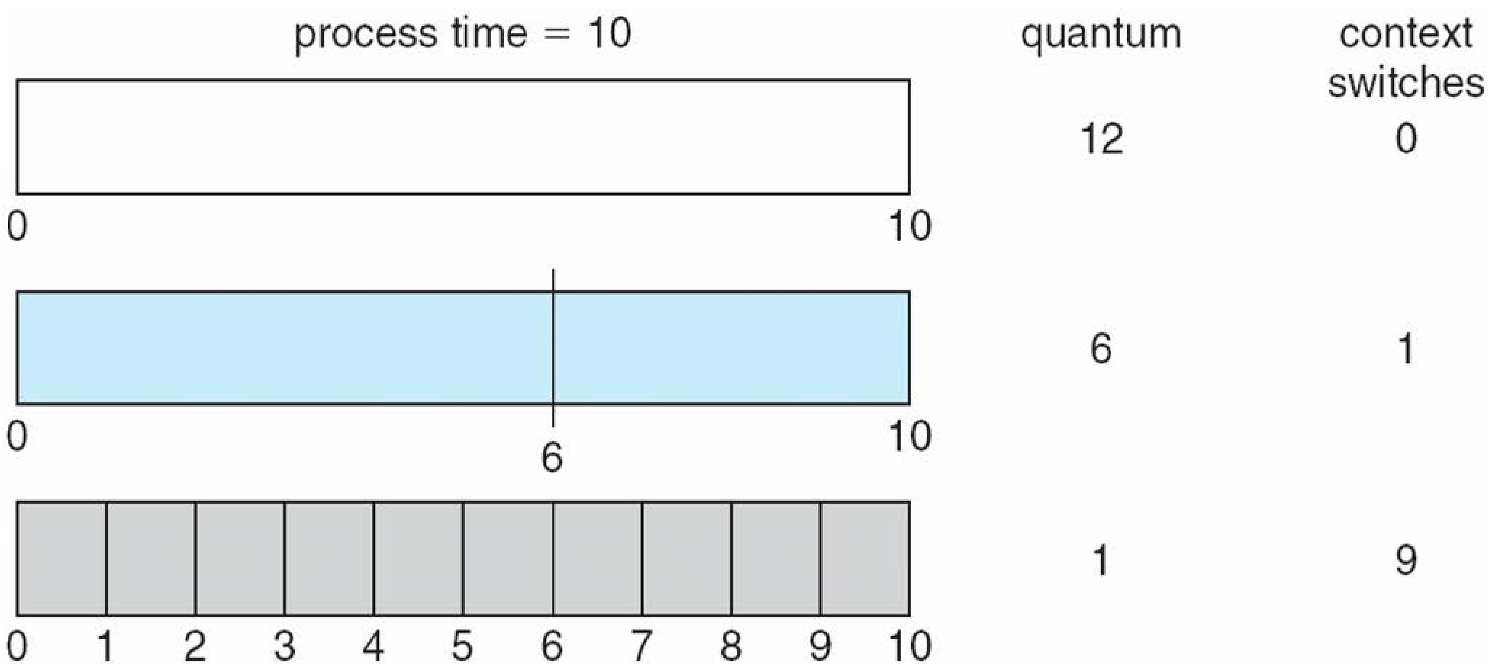
0 4 7 10 14 18 22 26 30

- average waiting time: (6 + 4 + 7) /3 = 5.67 ms

  (P_1: 10-4; P_2: 4; P_3: 7)

- Turnaround time : P1= 30 ; P2 = 7 ; P3= 10 ;
- Typically, higher average turnaround than SJF, but **better *response***

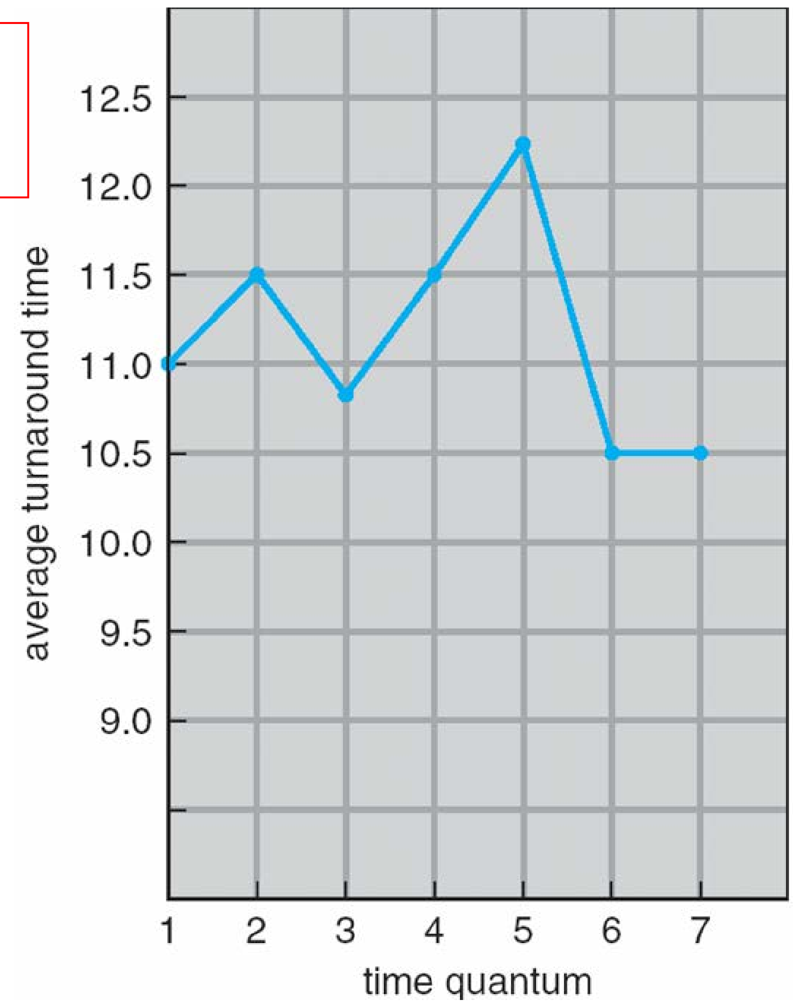
Time Quantum and Context-Switch Time



Showing how a smaller time quantum increases context switches

Turnaround Time Varies with the Time Quantum

Turnaround time also depends on the size of the quantum time

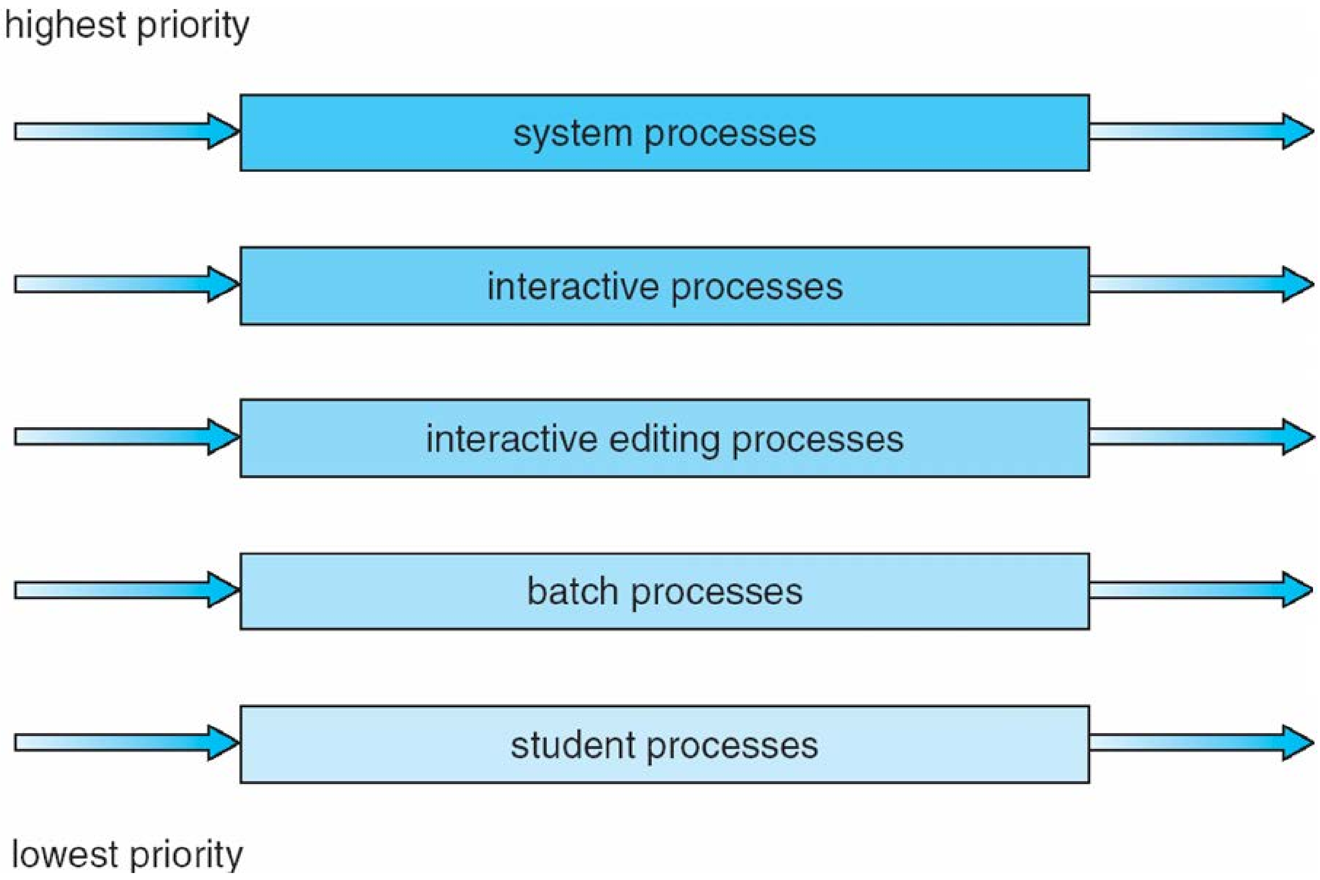


process	time
P_1	6
P_2	3
P_3	1
P_4	7

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
  - Commonly implemented as fixed-priority preemptive 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

Multilevel Queue Scheduling



An example of a multilevel queue scheduling algorithm with 5 queues, listed in order of priority.

Multilevel Feedback Queue Scheduling

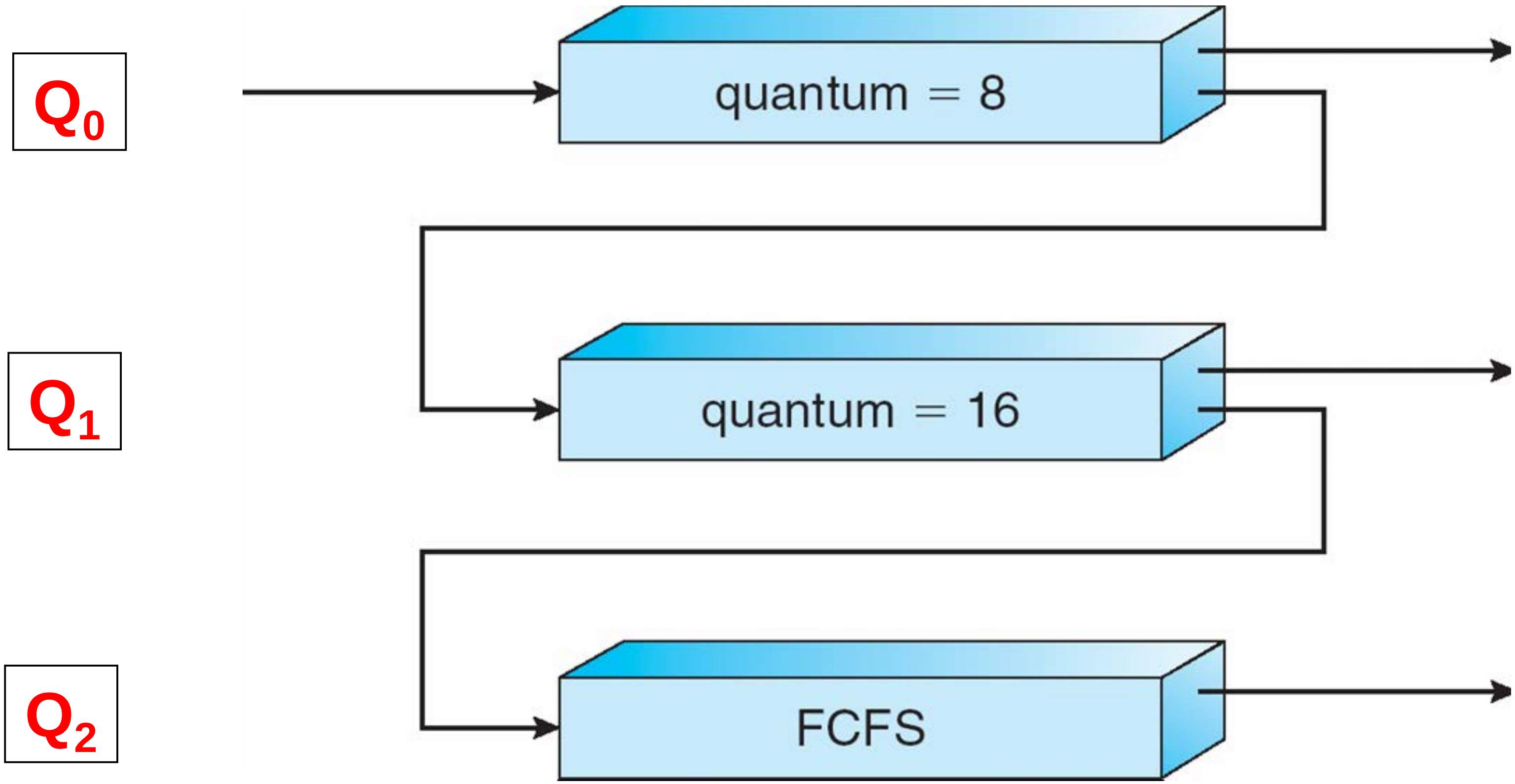
- A process can move between various queues; aging can be implemented this way to prevent starvation.
- Multilevel-feedback-queue scheduler is generally defined by the following parameters:
  - the number of queues
  - the scheduling algorithm for each queue
  - method used to determine when to upgrade 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

aging : เพิ่มศักดิ์ขึ้น
demote: ลดศักดิ์ลง

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum of 8 milliseconds
  - Q_1 – RR with time quantum of 16 milliseconds
  - Q_2 – FCFS
- Scheduling
  - A new job enters queue Q_0. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to the tail of queue Q_1.
  - Only when queue Q_0 is empty will the scheduler execute processes in queue Q_1. At Q_1 job receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2.
  - Processes in queue Q_2 are run on an FCFS basis but are run only when queues Q_0 and Q_1 are empty.

Multilevel Feedback Queues



Queue ถัดมาไม่สามารถเริ่มทำงานได้ หาก Queue ก่อนหน้าทำงานไม่เสร็จ หรือ ยังไม่ empty หรือ ยังไม่หมดช่วงเวลาที่กำหนดให้ (time quantum)

Thread Scheduling

- Distinction between user-level and kernel-level threads
- Many-to-one and many-to-many models, thread library schedules **user-level threads** to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the same process
- **Kernel thread** scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system
- System using one-to-one models such as Windows XP, Solaris, and Linux schedule thread using only SCS

Pthread Scheduling

- In thread creation with Pthreads, the POSIX Pthread API allows specifying either PCS or SCS during thread creation.
- Pthreads identifies the following contention scope values:
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling.

PCS: process-contention scope
SCS: system-contention scope

Pthread Scheduling API

```
#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_init(&attr);
         /* set the scheduling algorithm to PROCESS (PCS) or SYSTEM  (SCS)*/
      pthread_attr_setscope(&attr, PTHREAD_SCOPE _SYSTEM);
         /* set the scheduling policy - FIFO, RT, or OTHER */
      pthread_attr_setschedpolicy(&attr, SCHED_OTHER);

        /* create the threads */
      for (i = 0; i < NUM_THREADS; i++)
                pthread_create(&tid[i], &attr, runner, NULL);
```

Pthread Scheduling API

```
    /* now join on each thread */
    for (i = 0; i < NUM_THREADS;  i++)
            pthread_join(tid[i], NULL);
} /* end main   */

  /* Each thread will begin control in this function */
void *runner(void *param)
{
       printf("I am a thread\n");
    pthread_exit(0);
}
```

Multiple-Processor Scheduling

- CPU scheduling is more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **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
- **Processor affinity** – a process has an affinity for the processor on which it is currently running
  - **soft affinity : process may migrate between processors**
  - **hard affinity : process must not migrate to other processors**

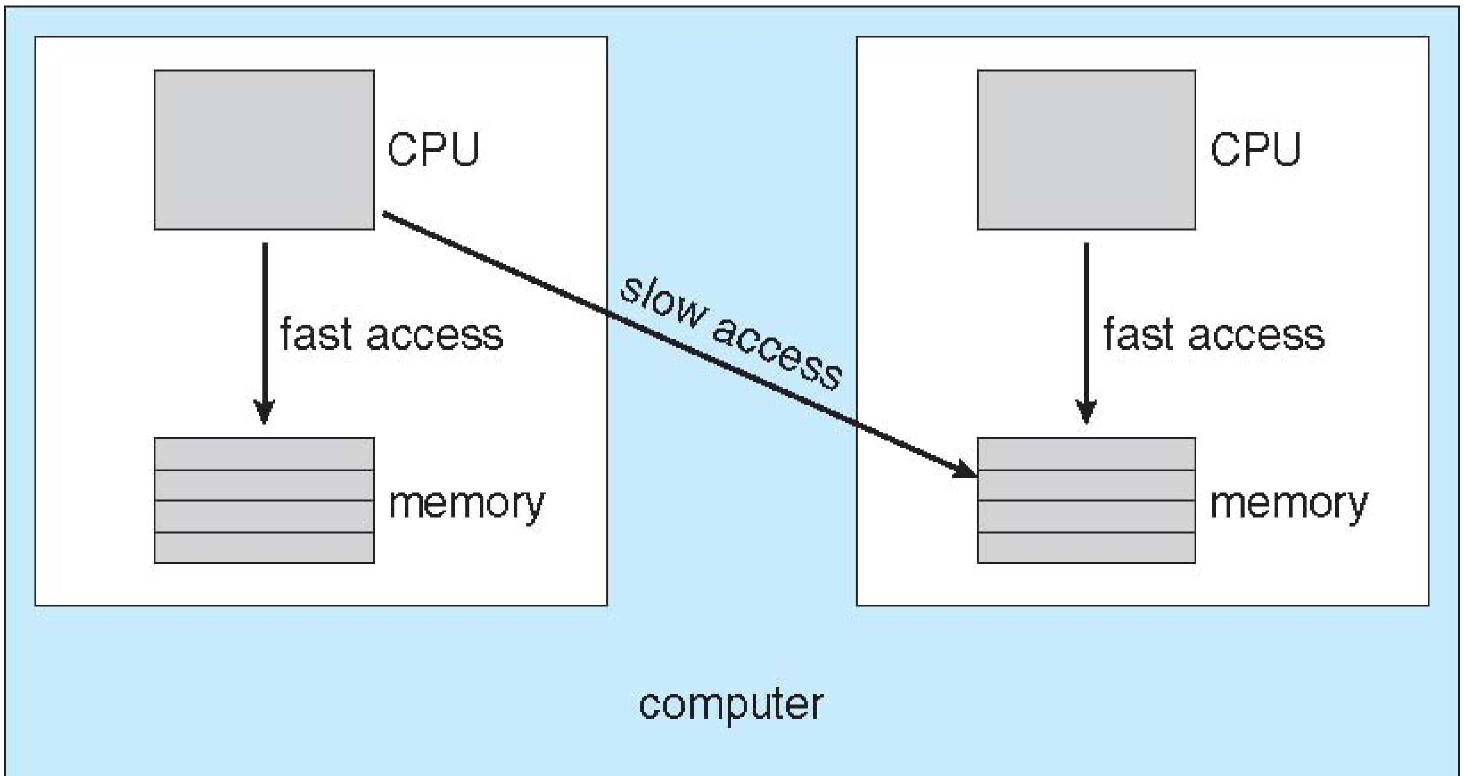
Homogeneous : แบบเดียวกัน เช่น cpu เป็น Intel เหมือนกัน
Heterogeneous : หลายแบบ เช่น cpu เป็น Intel, AMD, ultra spark , power Mac
Alleviating : แบ่งเบาภาระ
migrate : ย้ายการทำงาน
affinity: เกี่ยวพันกัน

NUMA and CPU Scheduling

NUMA : Non-Uniform Memory Access

สถาปัตยกรรมที่มีการใช้ NUMA จะทำให้

A CPU has faster access to some parts of main memory than to other parts.

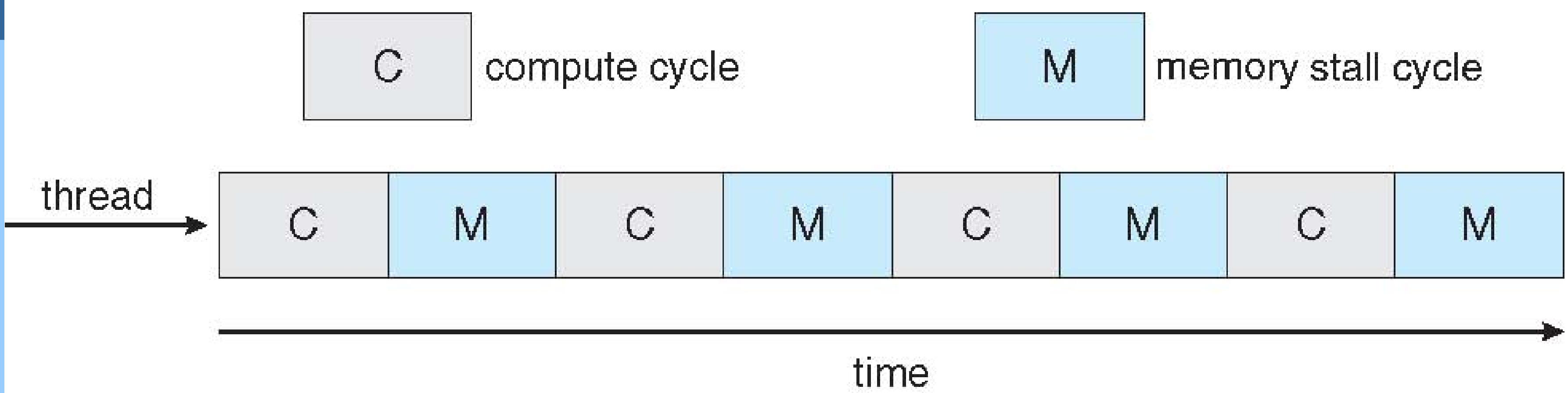


Multicore Processors

- Recent trend is to place multiple processor cores on the same physical chip
- SMP systems that use Multicore processors are Faster and consume Less power than systems in which each processor has its own physical chip
- Multiple threads per core are also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieval happens

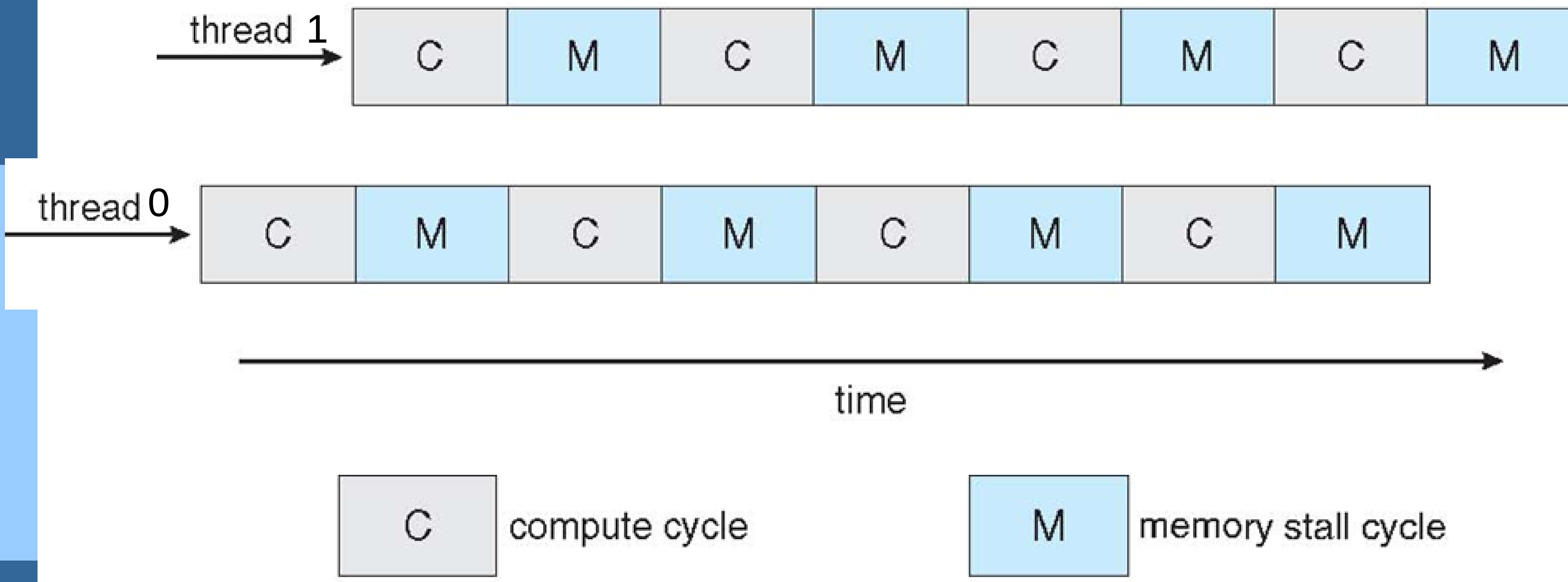
SMP : Symmetric Multiprocessing

Memory Stall



memory stall cycle : ช่วงเวลาที่ cpu ต้องรอการนำข้อมูลที่ไม่ได้อยู่ในหน่วยความจำให้ถูกload มาไว้ในหน่วยความจำ เช่น a cache miss (ข้อมูลที่ต้องการเข้าถึง ไม่ได้อยู่ใน cache memory)

Multithreaded Multicore System



Multithreaded processor cores in which 2 (or more) heardware threads are assigned to each core.
Thus, if one thread stalls while waiting for memory, the core can switch to another thread.

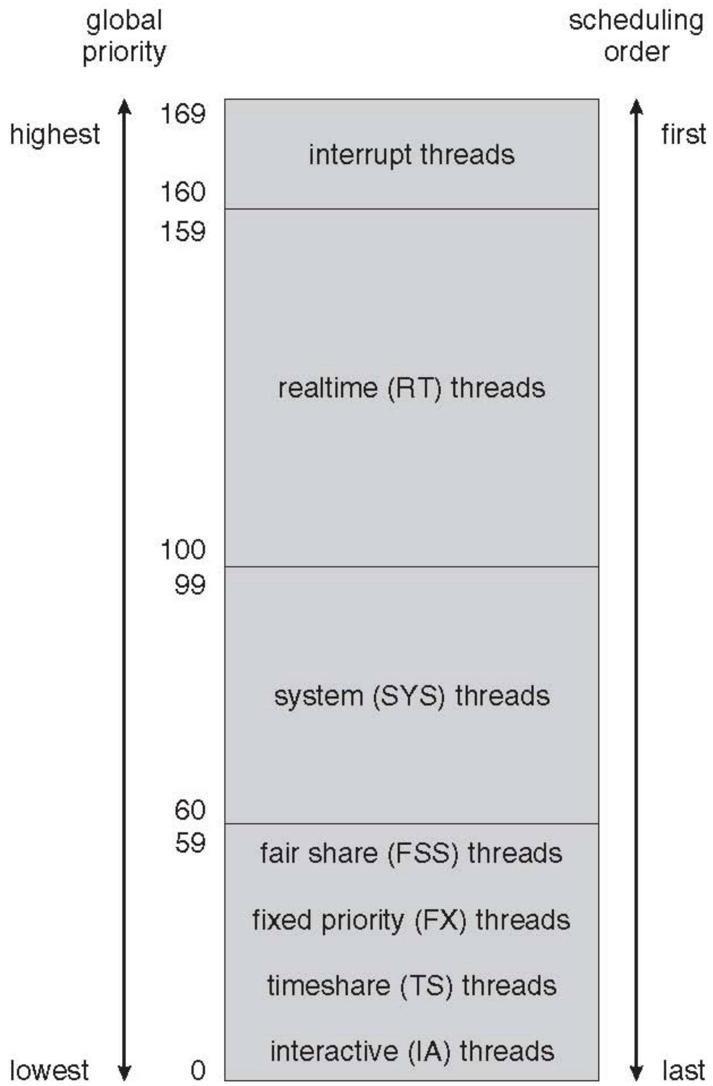
Operating System Examples

- Solaris scheduling
- Windows XP scheduling
- Linux 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

Solaris Scheduling



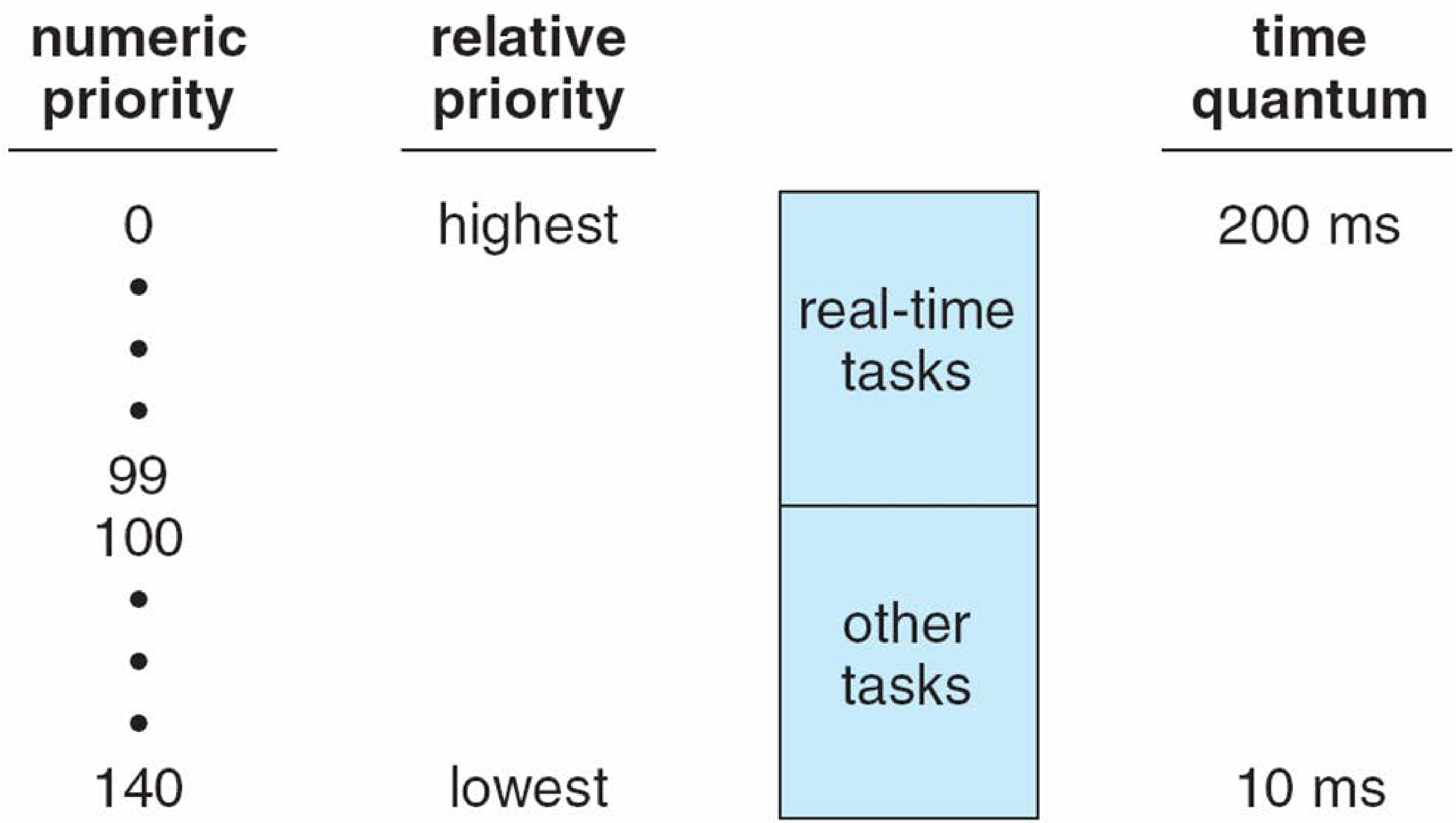
Windows XP Priorities

	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

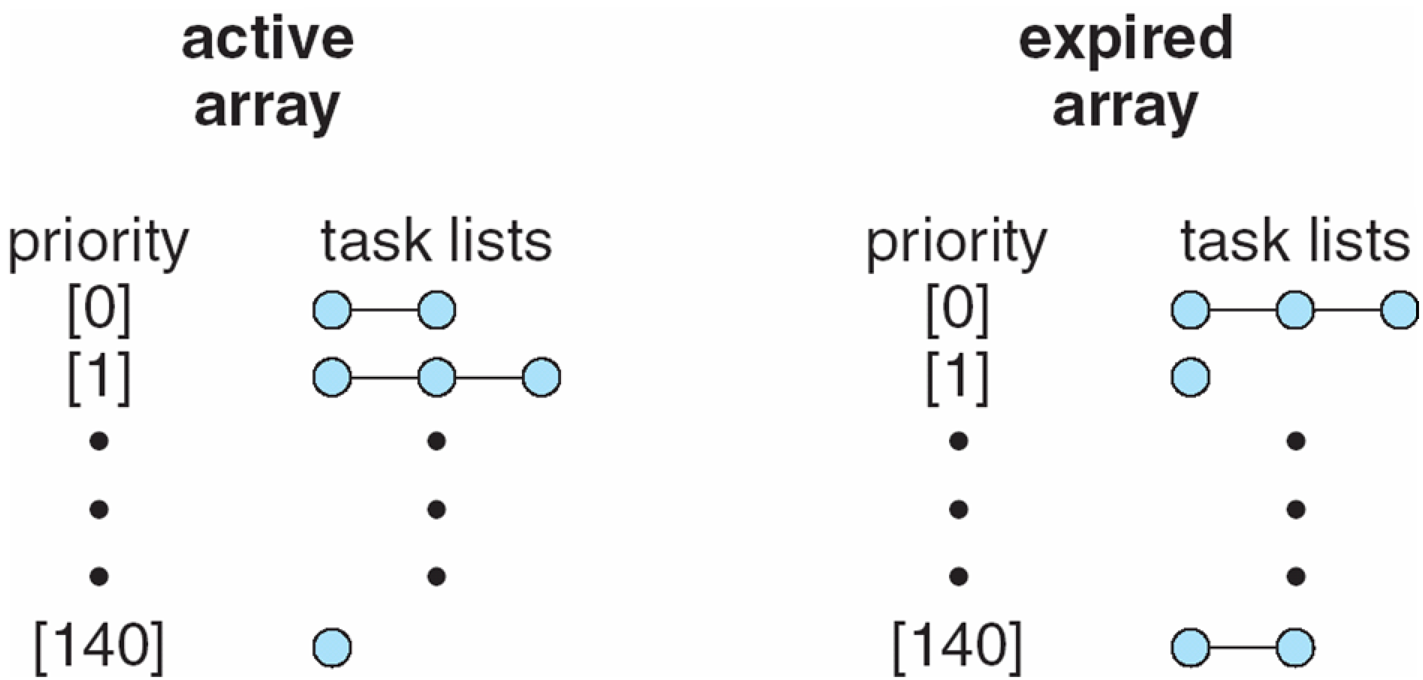
Linux Scheduling

- Constant order *O*(1) scheduling time
- Two priority ranges: time-sharing (or multitasking) and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
- See example picture on next slide

Priorities and Time-slice length



List of Tasks Indexed According to Priorities



active array: เก็บ task ที่ทำงานอยู่
expired array:เก็บ task ที่หมดเวลา

Algorithm Evaluation

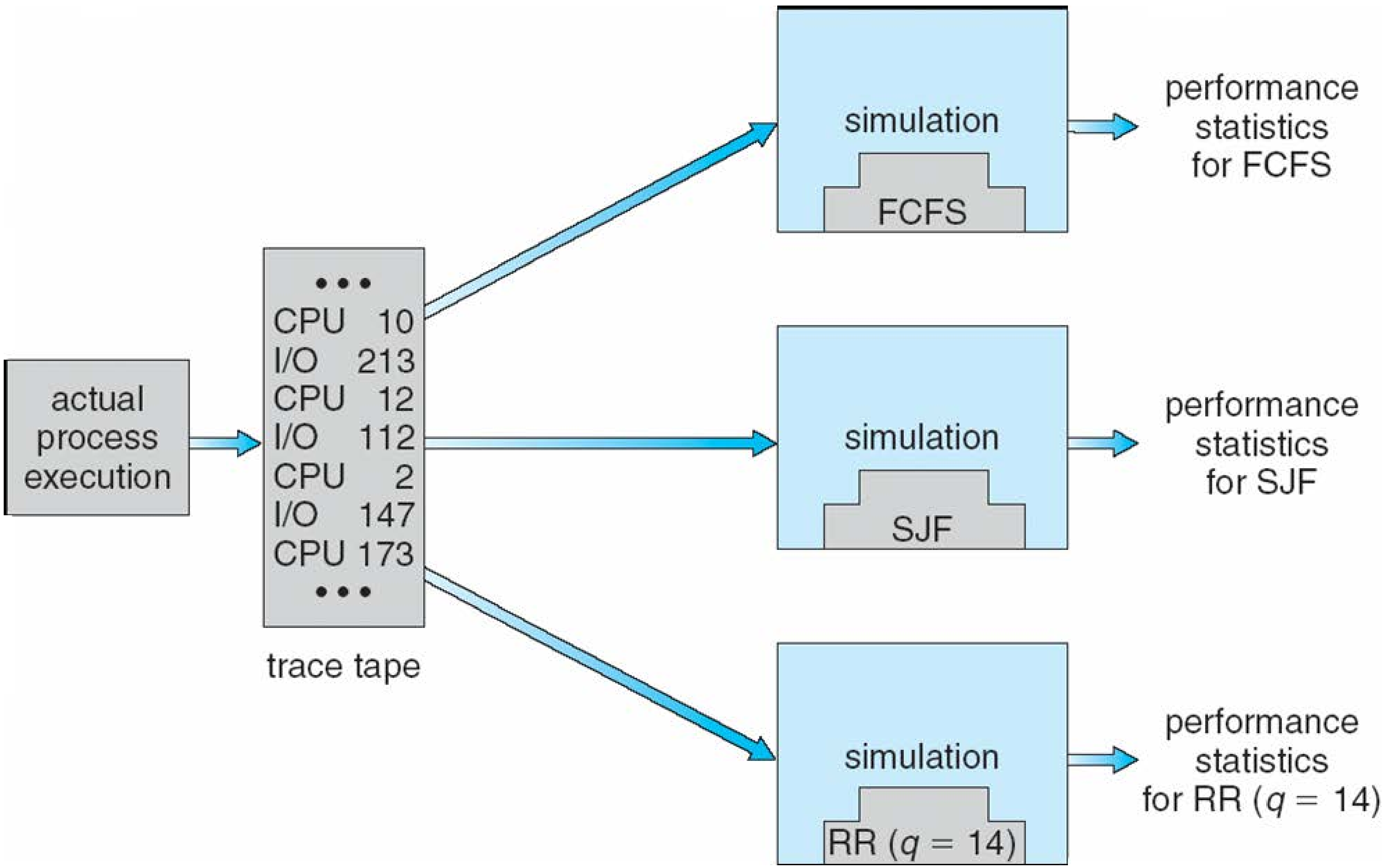
- Deterministic modeling – takes a particular predetermined workload and defines the performance of each algorithm for that workload.

  Ex. Define all processes running in FCFS, SJF, RR and then find out the result of minimum waiting time.

- Queueing models – what can be determined is the distribution of CPU and I/O bursts. Knowing arrival rate and service rates, we can compute utilization, average queue length, average wait time, and so on.

- Implementation – the only completely accurate way to evaluate a scheduling algorithm is to code it up, put it in the OS, and see how it works.

Evaluation of CPU schedulers by Simulation



End of Chapter 5