Chapter 2: Process Management

xep lich cac tien trinh de dam bao multi programing

2.3. CPU Scheduling lap lich

operating system

GV: Nguyễn Thị Thanh Vân

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

1

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms

Scheduling in OS

lap lich dai han
- Long term scheduler - also known as job scheduler.
  - It chooses the processes from the pool (secondary memory) and keeps them in the ready queue maintained in the primary memory.
  - It mainly controls the degree of Multiprogramming.
  - to choose a perfect mix of IO bound and CPU bound processes among the jobs present in the pool.

lap lich ngan han
- Short term scheduler - also known as CPU scheduler.
  - It selects one of the Jobs from the ready queue and dispatch to the CPU for the execution.
  - A scheduling algorithm is used to select which job is going to be dispatched for the execution.

lap lich trung han
- Medium term scheduler - takes care of the swapped out processes.
  - If the running state processes needs some IO time for the completion then there is a need to change its state from running to waiting.
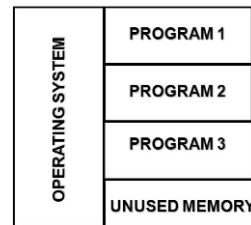
Basic Concepts

- Single program: A process is executed until it must wait for some I/O request.
  - => the CPU then just sits idle.
  - waiting time is wasted; no useful work is accomplished
- Multiprogramming: to maximize CPU utilization.
  - Processes are kept in memory at one time.
  - When 1 process has to <u>wait,</u> the OS takes the CPU away from that process and <u>gives the CPU to another process</u>.
  - => keeping the CPU busy is extended to all processing cores on the system.
- **CPU scheduling:** central to OS design
  - Almost all computer resources are scheduled
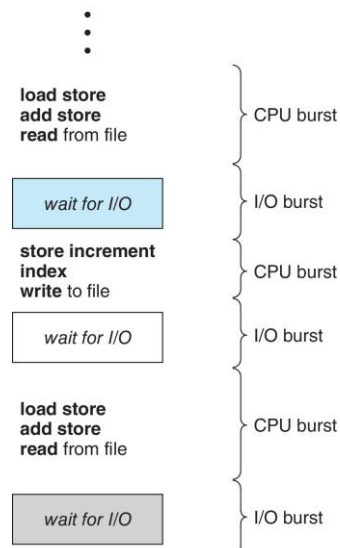 - CPU is one of the primary
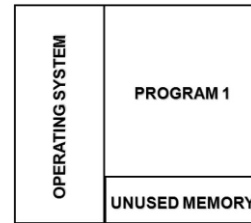
don chuong

da chuong

# Basic Concepts - CPU–I/O Burst Cycle

- **CPU scheduling**: Process execution consists of
  - a **cycle** of CPU execution and I/O wait
- Processes have 2 states (alternate): in fig
  - **CPU burst** followed by **I/O burst**
- Excution
  - begins with a CPU burst.,
  - I/O burst
  - …
  - the final CPU burst ends with a system request to terminate execution
- The durations of CPU bursts have been measured extensively

load store
add store
read from file — CPU burst

wait for I/O — I/O burst

store increment
index
write to file — CPU burst

wait for I/O — I/O burst

load store
add store
read from file — CPU burst

wait for I/O — I/O burst

3

Histogram of CPU-burst Times

- CPU burst distribution is of main concern
  - An I/O-bound program typically has many short CPU bursts.
  - A CPU-bound program might have a few long CPU bursts.
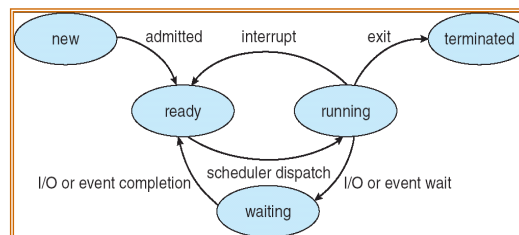

Long CPU burst

Short CPU burst

Waiting for I/O

Time
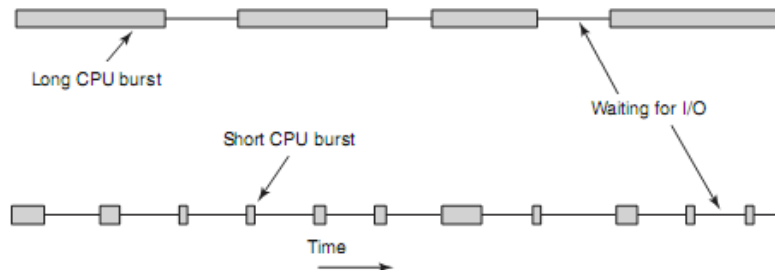
- A large number of short CPU bursts and a small number of long CPU bursts.
- This distribution can be important when implementing a CPU-scheduling

CPU Scheduler

- The **CPU scheduler** selects from among the processes in ready queue, and allocates a CPU core to one of them
  - The ready queue may be ordered in various ways: FIFO, FCFS, SJF
  - The records in the queues are generally process control blocks (PCBs) of the processes
- CPU scheduling decisions may take place when a process:

  trang thai
  1. Switches from running to waiting state   => need schedule
  2. Switches from running to ready state   => a choice (an option; a decision)   bi cuop quyen
  3. Switches from waiting to ready state   => a choice   duoc quyen
  4. Terminates   => need schedule

new   admitted   interrupt   exit   terminated

ready   running

I/O or event completion   scheduler dispatch   I/O or event wait

waiting

dispatch : tuoc quyen

4

Preemptive and Nonpreemptive Scheduling

- For situations 1 and 4, there is no choice in terms of scheduling. A new process (in the ready queue) must be **selected** for execution:
  - nonpreemptive or cooperative.
- For situations 2 and 3, however, there is a **choice** . (an option; a decision)
  - Preemptive
- **Nonpreemptive (không ưu tiên trước - Điều phối độc quyền)** doc quyen su dung cpu
  - is used when a process <u>terminates</u>, or switches from running to the <u>waiting</u>.
  - once the CPU cycles are allocated to a process, the process holds the CPU till it gets terminated or reaches a waiting state.
- **Preemptive (có ưu tiên - Điều phối không độc quyền)** ;khong doc quyen su dung cpu
  - is used when a process switches from running / waiting to <u>ready</u>.
  - The CPU cycles are allocated to the process for a limited amount of time and then taken away, and the process is again placed back in the ready queue to get its next chance to execute.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.

Preemptive and Nonpreemptive Scheduling

Para-meter	Preemptive scheduling	non-preemptive scheduling
Basic	In this resources (CPU Cycle) are allocated to a process for a limited time.	Once resources (CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	duoc ngat Process can be interrupted in between.	Process can not be interrupted until it terminates itself or its time is up. khong duoc ngat
Starvation	If a process having high priority frequently arrives in the ready queue, a low priority process may starve.	If a process with a long burst time is running CPU, then later coming process with less CPU burst time may starve.
qua tai Overhead	It has overheads of scheduling the processes.	It does not have overheads.
Flexibility	flexible	Rigid (inflexible)
Cost	cost associated	no cost associated
CPU Utilization	CPU utilization is high.	It is low in non preemptive scheduling.
Examples	Round Robin and Shortest Remaining Time First.	First Come First Serve and Shortest Job First.

duoc cap tai nguyen den khi tien trinh xong thi thoi

Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in <u>race conditions</u> when data are <u>shared among several processes</u>.
- Consider the case of two processes that share data (Race Conditions).
  - While one process is <u>updating</u> the data, it is **preempted** so that the second process can <u>run</u>.
  - The second process then tries to <u>read</u> the data, which are in an **inconsistent** state.
- We saw this in the bounded buffer example
- This issue will be explored in detail in Chapter 6.

Dispatcher

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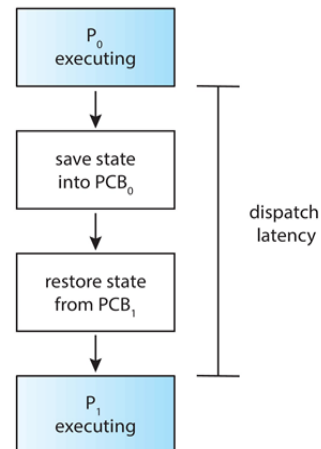
Scheduling Criteria

tieu chuan

Max ▪ **CPU utilization** – keep the CPU as busy as possible

Max ▪ so luong tien trinh se duoc hoan thanh tron 1 khoang thoi gian
Throughput – # of processes that complete their execution per time unit

Min ▪ tong thoi gian hoan thanh tien trinh
Turnaround time – amount of time to execute a particular process

Min ▪ thoi gian de 1 tien trinh cho
Waiting time – amount of time a process has been waiting in the ready queue

Min ▪ thoi gian ke tu luc su goi di sau do co su tra loi
Response time – amount of time it takes from when a request was submitted until the first response is produced.

Optimization Criteria for Scheduling Algorithms

Calculating the times

Calculating *CT* , *TAT* , *WT* :

Let

Completion Time (*CT*) be = 🟩 for each *ms* = Finishing Time – Starting Time

Turnaround Time (*TAT*) be = 🟨 for each *ms* = Completion Time – Arrival Time

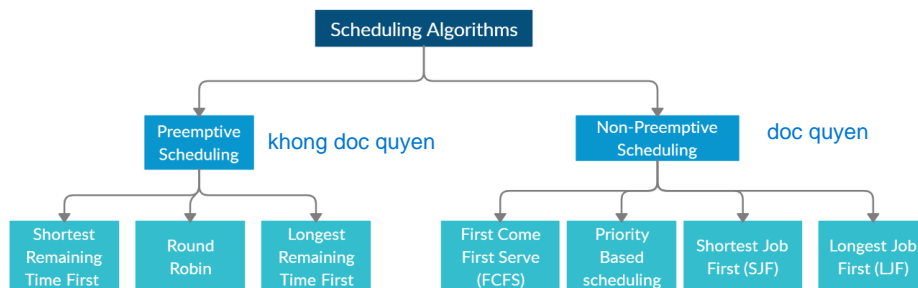Waiting Time (*WT*) be = 🟦 for each = Turnaround Time – Burst Time

Outline

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms

Scheduling Algorithms

Scheduling Algorithms

Preemptive Scheduling — khong doc quyen

Non-Preemptive Scheduling — doc quyen

- Shortest Remaining Time First
- Round Robin
- Longest Remaining Time First
- First Come First Serve (FCFS)
- Priority Based scheduling
- Shortest Job First (SJF)
- Longest Job First (LJF)

1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Round-Robin Scheduling
4. Priority Scheduling (version Preemptive + Non- Preemptive)
5. Multilevel Queue Scheduling
6. Multilevel Feedback Queue Scheduling

First- Come, First-Served (FCFS) Scheduling

- FCFS: the process that requests the CPU first is allocated the CPU first
- The implementation of the FCFS is easily managed with a FIFO queue.
  - When a process enters the ready queue,
  - its PCB is linked onto the tail of the queue.
  - When the CPU is free, it is allocated to the process at the head of the queue.
  - The running process is then removed from the queue.
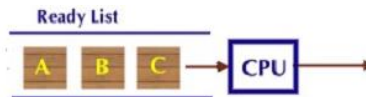
- The average waiting time under the FCFS policy is often quite long.

tien trinh nao toi truoc thi dc phuc vu truoc

First- Come, First-Served (FCFS) Scheduling

- Example with 3 processes

Process	Burst Time thoi gian thuc thi
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 above schedule is:

P_1	P_2	P_3

0   24   27   30

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27   P1 thuc thi xong moi toi P2 => doc quyen
- Average waiting time: (0 + 24 + 27)/3 = 17
- Operation: Once the CPU has been allocated to a process => it keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.
- The FCFS particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals.
  - FCFS: Nonpreemptive khong uu tien - doc quyen

Scheduling Algorithms

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

---

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order: P_2 , P_3 , P_1 . The Gantt chart :

P$_2$	P$_3$	P$_1$
0 3	6	30

- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$ thoi gian cho nghia la , thoi gian cho cho toi khi trien trinh do duoc thuc thi
- Average waiting time: (6 + 0 + 3)/3 = 3 => Much better than previous case
- FCFS may suffer from the **convoy effect**
  - if the burst time of the first job is the highest among all then the processes of lower burst time may get blocked
    - they may never get the CPU if the job in the execution has a very high burst time.

Longer job   **Shorter jobs**

10
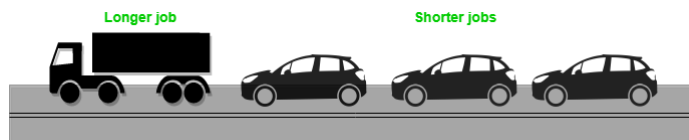
FCFS Scheduling (Cont.)

- Ex, Consider one CPU-bound and many I/O-bound processes
  - The CPU-bound process will get and hold the CPU, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU.
    => I/O devices are idle.
  - the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, execute quickly and move back to the I/O queues
    => CPU sits idle.
- Hence in Convoy Effect, one slow process slows down the performance of the entire set of processes, and leads to wastage of CPU time and other devices.
- To avoid Convoy Effect, preemptive scheduling algorithms like Round Robin Scheduling can be used
  - as the smaller processes don't have to wait much for CPU time – making their execution faster and leading to less resources sitting idle.

---

tien trinh co thoi gian thuc thi ngan nhat se duoc cpu phuc vu truoc

Shortest-Job-First (SJF) Scheduling

- SJF associates with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
  - If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie
- SJF is optimal – gives minimum average waiting time for a given set of processes:
  - Moving a short price before a long one
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

Shortest-Job-First (SJF) Scheduling

thuat toan nay co 2 kieu

- The SJF algorithm can be either preemptive or nonpreemptive.
  - Non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.
  - A preemptive SJF algorithm will preempt the currently executing process,

  (a new process arrives with less work than the remaining time of currently executing proc: => will run)

- Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

thoi gian cho
P2 : 0
P0 : 6-3 = 3
P3 : 8-1 = 7
P1 : 12-2 = 0

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

P2	P0	P3	P1	
0	6	8	12	16

Non-Preemtive Scheduling

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

P2	P3	P0	P1	P2	
0	1	5	7	11	16

Preemptive Scheduling

tien trinh hien tai phai thuc hien xong moi den thang khac

Detail, later =>

Shortest Remaining Time First (SRTF)

- Preemptive SJF: Shortest-remaining-time-first scheduling (SRTF):
  - Whenever a new process arrives in the ready queue, the decision on which process to schedule next is redone using the SJN (nonpreemtive SJF)
  - Is SRTF more "optimal" than SJN in terms of the minimum average waiting time for a given set of processes?

Process	Arrival Time	CPU Burst Time (in millisec.)
P0	3	2
P1	2	4
P2	0	6
P3	1	4

P2	P3	P0	P1	P2	
0	1	5	7	11	16

Preemptive Scheduling

- **Ưu tiên**: arri sớm + CPU time nhỏ
- 0: P2 arr&run
- 1: P3 arr, CPU time: P3(4)<P2(5) => P3 run trc
- 2: P1 arr, CPU time: P3(3)<P1(4), P2(5) => P3 run tiếp
- 3: P0 arr, CPU time: P3(2)=P0(2), P2(5), P1(4) => P3 run tiếp vì arr trc (sameP1)
- 5: P3 done. P0(2)< P2(5), P1(4) => P0 run trc
- 7: P0 done. P1(4)<P2(5) -> P1 run trc
- 11: P1 done. P2 run
- 16: P2 done
- AWT?

12

Example of SRTF (Preemption SJF)

- Preemption SJF to the analysis - arri early + CPU time small

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

turnaround time = completion time - arrival time
waiting time = turnaround time - burst time

- Analysis: Trace

time	0	1	2	3	5	10	17	26
P1	**8**	7	7	7	7	**7**	**0**	0
P2		**4**	3	2	**0**	0	0	0
P3			9	9	9	9	**9**	**0**
P4				5	**4**	**0**	0	0

- Gantt Chart -

P_1	P_2	P_4	P_1	P_3

0 1 5 10 17 26

- Average waiting time = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5ms

---

Example of NonPreemptive SJF

- Non Preemption to the analysis – no break

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

- Analysis:

time	0	1	2	3	5	8	12	17	26
P1	**8**	7	6	5	3	**0**	0	0	0
P2		4	4	4	4	**4**	**0**	0	0
P3			9	9	9	9	9	**9**	**0**
P4				5	5	5	**5**	**0**	**0**

- Gantt Chart:

P1	P2	P4	P3

0   8   12   17   26
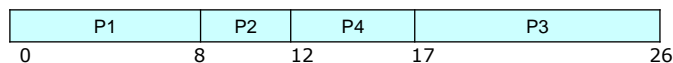
- Average waiting time of 7.75 milliseconds = (0+ (8-1)+(12-3)+(17-2))/4
- **FCFS** result in an average waiting time 8.75= (0+7+10+18)/4
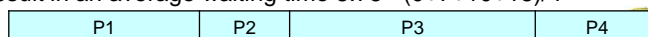
P1	P2	P3	P4

Preemp uu tien (khong doc quyen) :Shorted-Job-First Scheduling
Non-preemp khong uu tien (doc quyen) : FCFS thuuat toan khong uu tien , Shorted-Job-First Scheduling

13

doc quyen : chay cho xong moi thoi
khong doc quyen : neu 1 tien trinh co thoi gian ngan hon tien trinh cpu dang xu li , cpu dung
xu li tien trinh dang chay chuyen sang giai quyet tien trinh tham gia

Determining Length of Next CPU Burst

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
- Equation:

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

- τ_{n+1} = giá trị dự đoán cho thời gian sử dụng CPU tiếp sau
- t_n = thời gian thực tế của sự sử dụng CPU thứ n
- α, $0 \le \alpha \le 1$
- τ_0 là một hằng số

Prediction of the Length of the Next CPU Burst
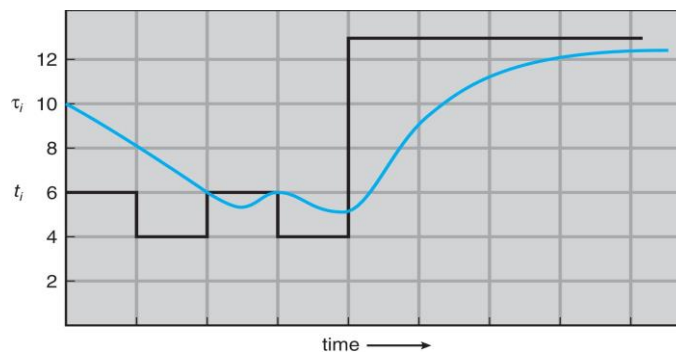
| CPU burst (t_i) | 6 | 4 | 6 | 4 | 13 | 13 | 13 | ... |
| "guess" (τ_i) | 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | ... |

Figure shows an exponential average with $\alpha = \frac{1}{2}$ and $\tau 0 = 10$

14

Examples of Exponential Averaging

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

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n = \tau_0$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha \, t_n = t_n$
  - 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 α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

---

Scheduling Algorithms

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

Round Robin (RR)

uu tien

- RR: similar to FCFS scheduling, but preemption is added to enable the system to switch between processes
- Each process gets a small unit of CPU time (**time quantum** q - *định lượng thời gian*),
  - 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

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

cu sau 1 khoan thoi gian = 4 thi thay doi Process

- The Gantt chart is: order P1,P2,P3

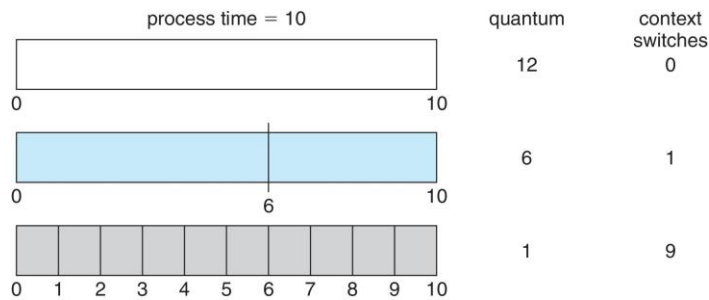| P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0   4   7   10   14   18   22   26   30
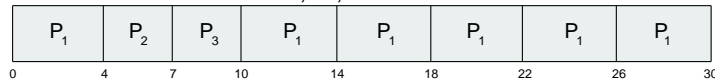
- Waiting time for
  - P1: 3(p2)+3(p3)=6
  - P2: 4(p1) =4
  - P3: 4(p1)+3(p2)=7
- average waiting time =(6+4+7)/3 = 5.67ms
- Typically, higher average turnaround than SJF, but better *response*
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds
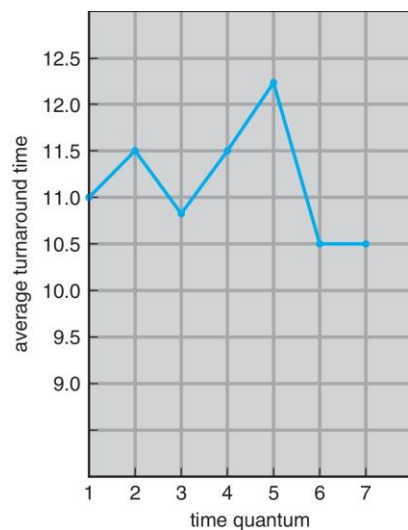
Time Quantum and Context Switch Time

process time = 10	quantum	context switches

process time = 10
```
0                    10
```
| | 12 | 0 |

```
0        6           10
```
| | 6 | 1 |

```
0 1 2 3 4 5 6 7 8 9 10
```
| | 1 | 9 |

Performance
- q large $\Rightarrow$ FIFO (same FCFS)
- q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high,
- => performance decrease   hieu suat bi giam

Turnaround Time Varies With The Time Quantum

process	time
P_1	6
P_2	3
P_3	1
P_4	7

Rule:
80% of CPU bursts should be shorter than q

17

Scheduling Algorithms

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

Priority Scheduling

uu tien scheduling : chuong trinh
quy dinh do uu tien cho cac tien trinhf

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (usually, smallest integer ≡ highest priority)
- Two schemes:
  - Preemptive
  - Nonpreemptive
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process
- Note: SJF is priority scheduling where priority is the inverse of predicted next CPU burst time

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

P_2	P_5	P_1	P_3	P_4
0 1	6	16	18	19

- Average waiting time (0 + 1 + 6 + 16 + 18)/5 = 8.2

Priority Scheduling w/ Round-Robin

- Run the process with the highest priority. Processes with the same priority run round-robin   hai tien trinh co cung do uu tien thi su dung round-robin
- Example:

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

- Gantt Chart with time quantum = 2

P_4	P_2	P_3	P_2	P_3	P_2	P_3	P_1	P_5	P_1	P_5
0	7	9	11	13	15	16	20	22	24	26 27
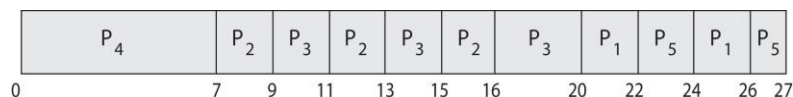
P2 chi thuc thi trong 2s , sau do chuyen sang Process co cung do uu tien voi P2 la P3

Waiting time p1 = 22s

19

Scheduling Algorithms

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

Multilevel Queue

nhieu hang doi , moi hang doi co nhung process

- The ready queue consists of multiple queues
- Example:
  - Priority scheduling, where each priority has its separate queue.
    moi priority co mot queue cua rieng no
  - Schedule the process in the highest-priority queue!   xu li process trong queue co priority cao nhat

priority = 0: T_0, T_1, T_2, T_3, T_4

priority = 1: T_5, T_6, T_7

priority = 2: T_8, T_9, T_{10}, T_{11}

...

priority = n: T_x, T_y, T_z

20

Multilevel Queue

- Prioritization based upon process type

highest priority
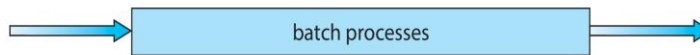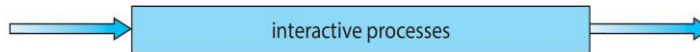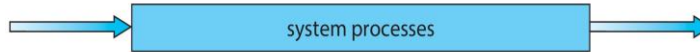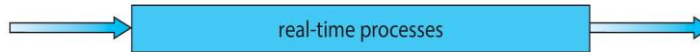
lowest priority   dong goi de thuc thi

Scheduling Algorithms

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

Multilevel Feedback Queue

tien trinh co the di chuyen giua cac hang doi

- A process can move between the various queues.
- 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
- Aging can be implemented using multilevel feedback queue

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS
  - Q1 has higher priority than both Q2 and Q3 (Q1>Q2>Q3)
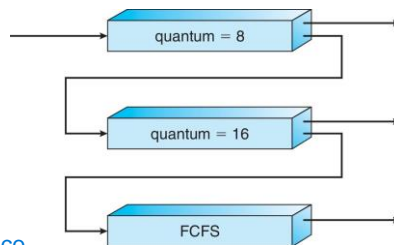
mot tien trinh di vao , neu trong cac queue chua co
process de xu li thi , tien trinh di vao queue co do uu tien cao nhat

- Scheduling:
  - A new process enters queue Q_0 which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue Q_1
  - At Q_1 job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is **preempted** and moved to queue Q_2

ex

Process	Arrival Time	Burst Time
P1	0	36
P2	16	20
P3	20	12

quantum = 8 / quantum = 16 / FCFS
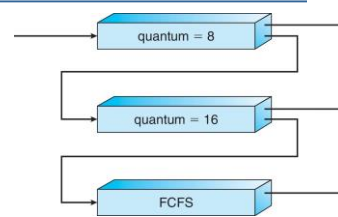
	0	8	16	20	24	32	48	60	64	68
Q1 8	P1(36)		P2(20)	P3(12) P2(16)	P3(12)					
Q2 16		P1(24)	P1(20)	P1(20)	P1(20) P2(12)	P1(20) P2(12) P3(4)	P2(12) P3(4)	P2(0) P3(4)	P3(0)	
Q3							P1(4)	P1(4)	P1(4)	P1(0)

tai 20 : P3 toi Q1 nhung P2 chua run xong . luc nay P2 , P3 cung trong Q1 , P2 toi truoc nen P2 duoc run xong het q8

Expl

The Gantt chart is:

P1	(32-16)+(64-48)= 32
P2	(48-24) = 24
P3	(24-20)+(60-32) = 32

Average Waiting Time

= (32+24+32)/3
= 29.33

- Tại 0: P1 vào Q1, run với q=8, not done, move Q2. Q1: null
- Tại 8: Do trong Q1 null, nên P1 trong Q2 run. Q2: P1
- Tại 16: P2 vào Q1 vì Q1 có ưu tiên cao hơn Q2 nên P2 được run trước, P1 bị ngắt, P1 còn 20 (36-8-8), P2 run với q=8 đến time 24 sẽ còn 12 (20-8). Q1:P2, Q2:P1. Nhưng
- Tại 20: P3 tới Q1 nhưng P2 chưa run xong. Lúc này P2, P3 **cùng trong Q1** nên P2 đc run xong với q=8, còn 12 (20-8) nên move Q2. lúc này Q1: P3, Q2: P1,P2
- Tại 24: P3 ở Q1, do Q1 có ưu tiên cao hơn Q2 nên P3 được run trước, sau q=8 thì ngắt, P3 còn 4 (12-8). P3 move Q2, lúc này Q1: null, Q2: P1,2,3.
- Tại 32: Theo RR, P1 trong Q2 sẽ run trước với q=16, P1 còn 4 (20-16). P1 move to Q3. trong Q2: P2,3
- Tại 48: theo RR, P2 trong Q2 run vì được ưu tiên hơn Q3, sau 12m thì P2 done
- Tại 60: P2 done, P3 run sau 4m còn lại thì done tại 64. Trong Q2 null, đến Q3 run
- Tại 64: P1 trong Q3 được run sau 4m còn lại thì done tại 68

23

Chapter 2: Process Management

Advanced CPU Scheduling

operating system

Outline

- 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

Thread Scheduling

- 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

Thread Scheduling

- Lập lịch cục bộ (Local Scheduling):
  - Bằng cách nào Thư viện luồng quyết định chọn luồng nào để đặt vào một CPU ảo khả dụng:
    - Thường chọn luồng có mức ưu tiên cao nhất
  - Sự cạnh tranh CPU diễn ra giữa các luồng của cùng một tiến trình.
  - Trong các HĐH sử dụng mô hình Many-to-one, Many-to-many.

- Lập lịch toàn cục (Global Scheduling)
  - Bằng cách nào kernel quyết định kernel thread nào để lập lịch CPU chạy tiếp.
  - Sự cạnh tranh CPU diễn ra giữa tất cả các luồng trong hệ thống.
  - Trong các HĐH sử dụng mô hình One-to-one (Windows XP, Linux, Solaris 9)

Operating System Concepts – 10th Edition

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

Pthread Scheduling API

```c
#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

```c
    /* 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);
}
```

Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation
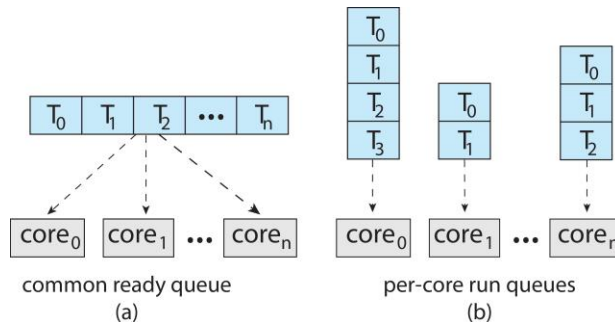
Multiple-Processor Scheduling

- 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
 - Homogeneous

Multiple-Processor Scheduling

- 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)

common ready queue (a)   per-core run queues (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
- Figure

C compute cycle   M memory stall cycle

thread → C M C M C M C M

time

29
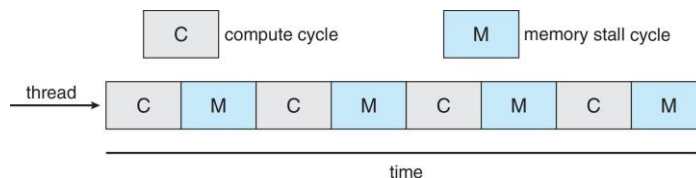
Multithreaded Multicore System

- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure
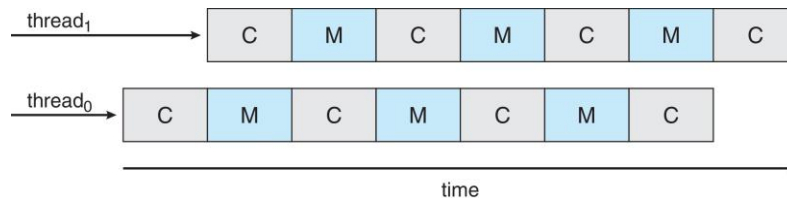
Multithreaded Multicore System

- **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.
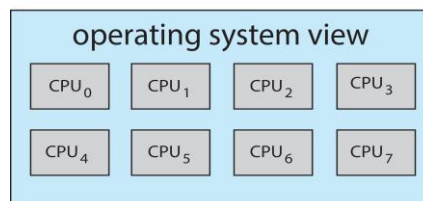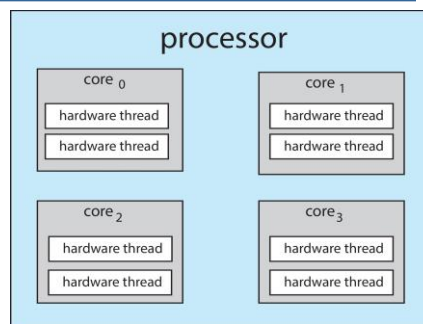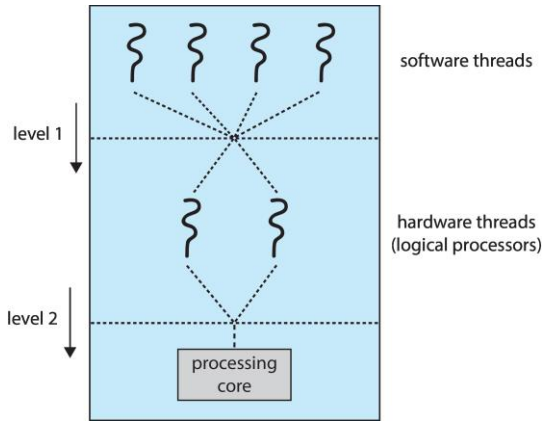
Multithreaded Multicore System

- 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.

level 1

level 2

software threads

hardware threads
(logical processors)

processing
core

Multiple-Processor Scheduling – Load Balancing

- If SMP (Symmetric multiprocessing), 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
- **Pull migration** – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

- 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")
- 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 and CPU Scheduling

- NUMA **(**non-uniform memory access), is a method of configuring a cluster of microprocessor in a multiprocessing system so that they can share memory locally, improving performance and the ability of the system to be expanded.
- NUMA is used in a symmetric multiprocessing (SMP) system.
- If the operating system is **NUMA-aware,** it will assign memory closes to the CPU the thread is running on.
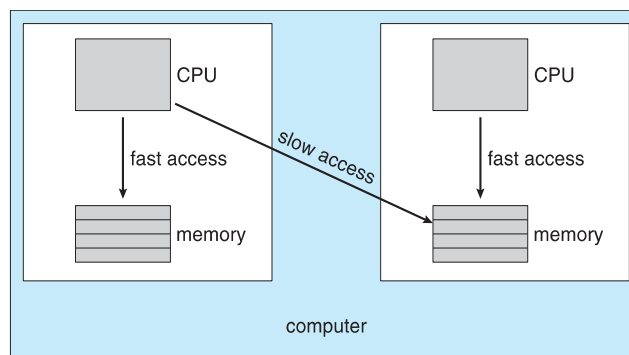
Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Real-Time CPU Scheduling

- Can present obvious challenges
- **Hard real-time systems –**
  - **task must be serviced by its deadline**
- **Soft real-time systems** –
  - Critical real-time tasks have the highest priority, but no guarantee as to when tasks will be scheduled
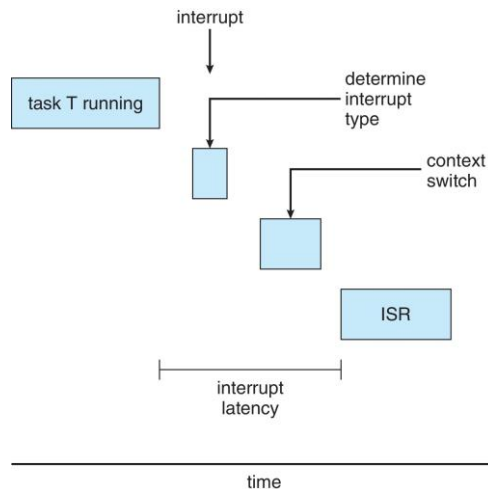
Real-Time CPU Scheduling

- 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

event E first occurs

event latency

t_0 t_1

real-time system responds to E

Time
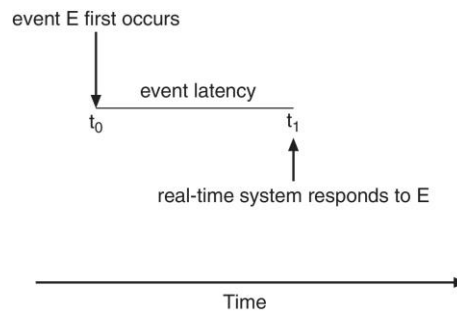
Interrupt Latency

The interrupt latency refers to the delay between the start of an Interrupt Request (IRQ) and the start of the respective Interrupt Service Routine (ISR). The interrupt latency is expressed in core clock cycles.

interrupt

task T running

determine interrupt type

context switch

ISR

interrupt latency

time

34

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

---
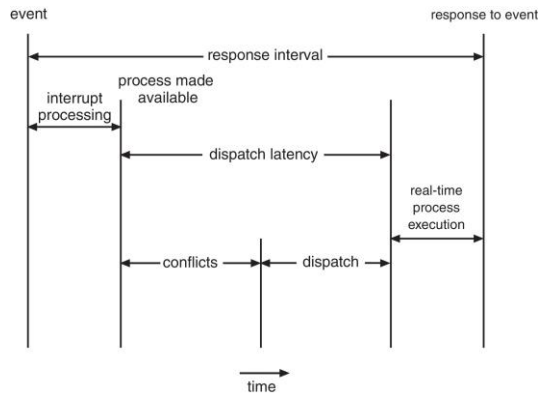
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, ex: 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
    - $0 \le t \le d \le p$
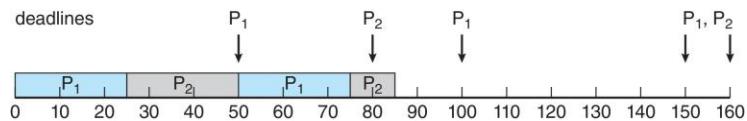   - **Rate** of periodic task is $1/p$
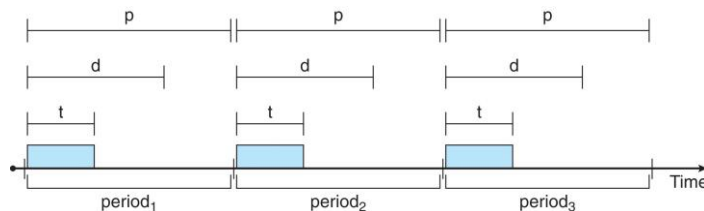
Rate Monotonic Scheduling

- 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.
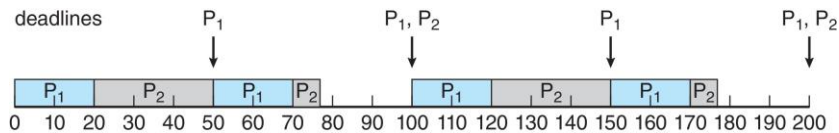
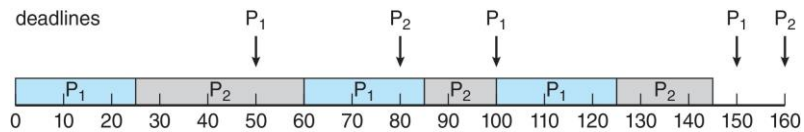# Missed Deadlines with Rate Monotonic Scheduling

- Process P_2 misses finishing its deadline at time 80
- Figure

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
- Figure

Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares where $N < T$
- This ensures each application will receive **N / T** of the total processor time

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:
    1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
    2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority
- Defines two functions for getting and setting scheduling policy:
    1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
    2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

POSIX Real-Time Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t_tid[NUM_THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER) printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR) printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO) printf("SCHED_FIFO\n");
   }
```

```
    /* set the scheduling policy - FIFO, RR, or OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
        fprintf(stderr, "Unable to set policy.\n");
    /* 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);
}
```

Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

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
  - Two scheduling classes included, others can be added
    1. default
    2. real-time

Linux Scheduling in Version 2.6.23 + (Cont.)

- 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:
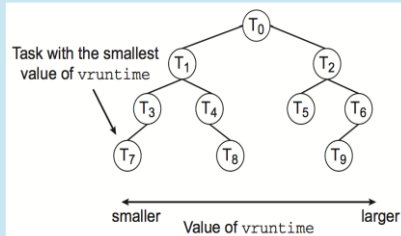

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and 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
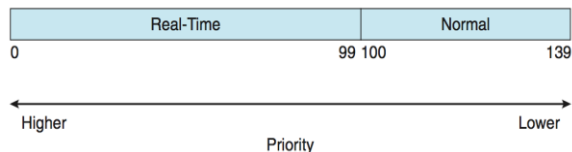
Linux Scheduling (Cont.)

- Linux supports load balancing, but is also NUMA-aware.
- **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.
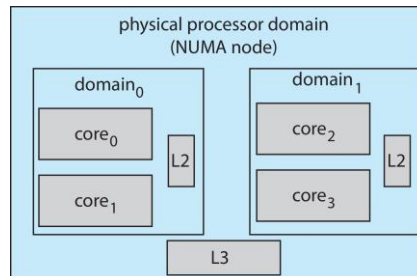
Windows Scheduling

- 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

	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

Solaris

- Priority-based scheduling
- 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
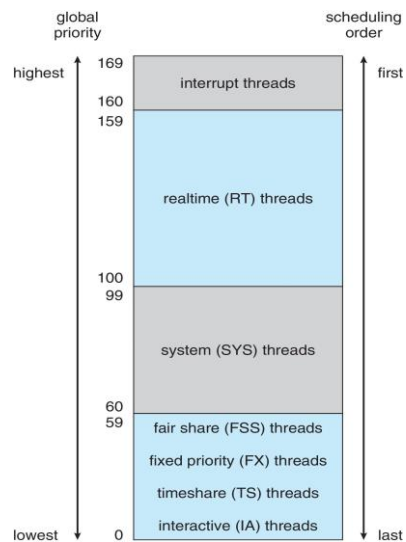 - Loadable table configurable by sysadmin

Solaris 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

46

Solaris Scheduling (Cont.)

- Scheduler converts class-specific priorities into a per-thread global priority
  - Thread with highest priority runs next
  - Runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
  - Multiple threads at same priority selected via RR

Outline

- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Algorithm Evaluation

- How to select CPU-scheduling algorithm for an OS?
- Determine criteria, then evaluate algorithms
- **Deterministic modeling**
  - Type of **analytic evaluation**
  - Takes a particular predetermined workload and defines the performance of each algorithm for that workload
- Consider 5 processes arriving at time 0:

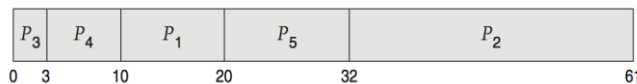| Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

Deterministic Evaluation

- For each algorithm, calculate <u>minimum average waiting</u> time
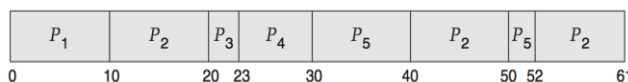- Simple and fast, but requires exact numbers for input, applies only to those inputs

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- FCS is 28ms:

P_1	P_2	P_3	P_4	P_5

0   10   39 42   49   61

- Non-preemptive SFJ is 13ms:

P_3	P_4	P_1	P_5	P_2

0 3   10   20   32   61

- RR is 23ms (q=10):

P_1	P_2	P_3	P_4	P_5	P_2	P_5	P_2

0   10   20 23   30   40   50 52   61

Queueing Models

- Describes the arrival of processes, and CPU and I/O bursts probabilistically
  - Commonly exponential, and described by mean
  - Computes average throughput, utilization, waiting time, etc.
- Computer system described as network of servers, each with queue of waiting processes
  - Knowing arrival rates and service rates
  - Computes utilization, average queue length, average wait time, etc.

Little's Formula

- n = average queue length
- W = average waiting time in queue
- λ = 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
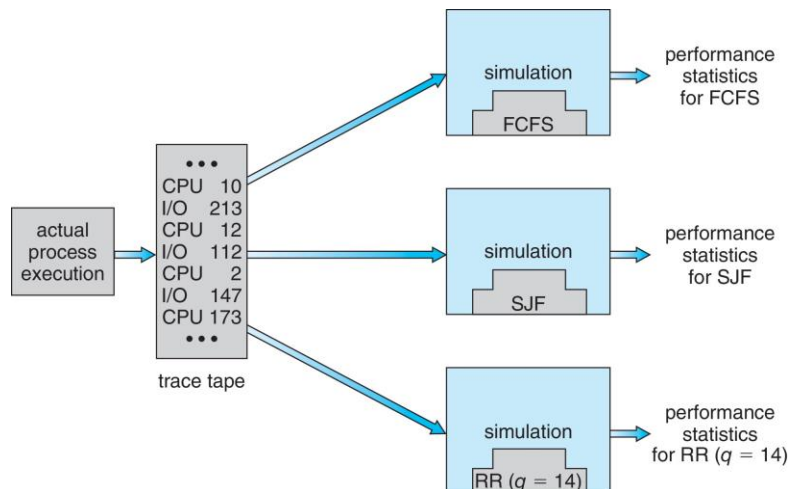
Simulations

- Queueing models limited
- **Simulations** more accurate
  - Programmed model of computer system
  - Clock is a variable
  - Gather statistics indicating algorithm performance
  - Data to drive simulation gathered via
    - Random number generator according to probabilities
    - Distributions defined mathematically or empirically
    - Trace tapes record sequences of real events in real systems

Evaluation of CPU Schedulers by Simulation

Implementation

- Even simulations have limited accuracy
- Just implement new scheduler and test in real systems
  - High cost, high risk
  - Environments vary
- Most flexible schedulers can be modified per-site or per-system
- Or APIs to modify priorities
- But again environments vary

End of Chapter 5