CS222
Operating Systems

Lecture 07
Process Scheduling

(Section 100001)

ผศ. ดร. กษิดิศ ชาญเชี่ยว

ckasidit@tu.ac.th

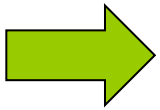
Tom Duff's Device

- ในปี 1983 Tom Duff ได้เขียน Email ส่งไปที่ Usenet (เป็น bulletin board สำหรับโพสข้อความและแลกเปลี่ยนความคิดเห็นในสมัยนั้น)
- เขาเปลี่ยนโค๊ด ใน printer driver แบบหนึ่ง **จาก A เป็น B** ข้างล่าง **นศ คิดว่า โค๊ดนี้ทำอะไร**
- มีตัวอย่างโปรแกรมอยู่ที่

https://github.com/kasidit/system-programming/blob/master/src/duff/duff.c

```
send(to, from, count)
register short *to, *from;
register count;
{
        do
                *to = *from++;
        while(--count>0);
}
```

A



```
send(to, from, count)
register short *to, *from;
register count;
{
        register n=(count+7)/8;
        switch(count%8){
        case 0: do{     *to = *from++;
        case 7:         *to = *from++;
        case 6:         *to = *from++;
        case 5:         *to = *from++;
        case 4:         *to = *from++;
        case 3:         *to = *from++;
        case 2:         *to = *from++;
        case 1:         *to = *from++;
                }while(--n>0);
        }
}
```

B

Tom Duff's Device

- https://github.com/kasidit/system-programming/blob/master/src/duff/duff.c

```
void send1(register char *to, register
        do
                *to++ = *from++;
        while(--count>0);
}
```

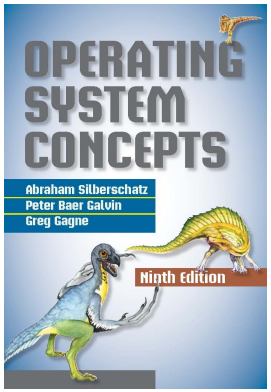
A

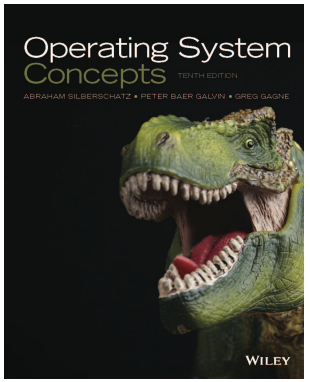
```
void send2(register char *to, register char
        register int n=(count+7)/8;

        switch(count%8){
        case 0: do{     *to++ = *from++;
        case 7:         *to++ = *from++;
        case 6:         *to++ = *from++;
        case 5:         *to++ = *from++;
        case 4:         *to++ = *from++;
        case 3:         *to++ = *from++;
        case 2:         *to++ = *from++;
        case 1:         *to++ = *from++;
                }while(--n>0);
        }
}
```

B

Textbook

- Avi Silberschatz, Peter B. Galvin and Greg Gagne; Operating System Concepts, 9th Edition; John Wiley & Sons, Inc; 2012; ISBN 978-1118063330



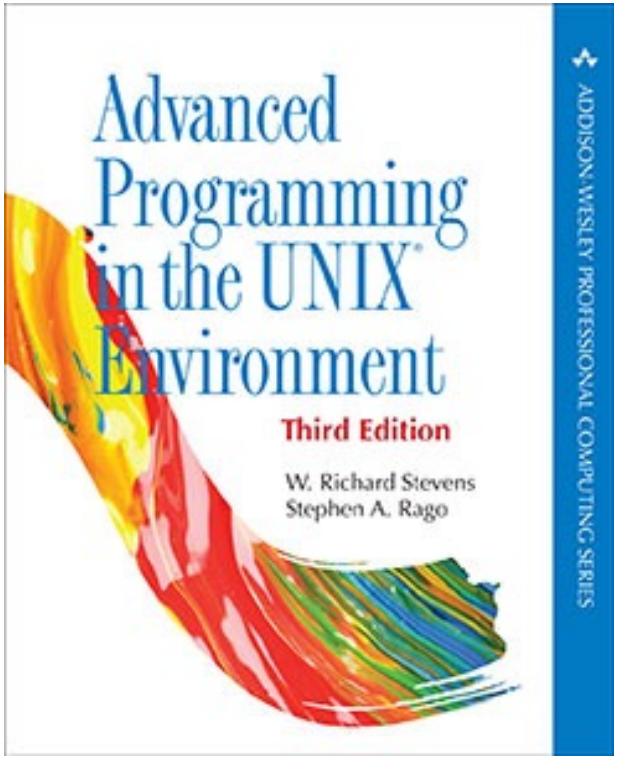


- Chapter 5

- Original Slides
- https://www.os-book.com/OS9/slide-dir/index.html
- https://www.os-book.com/OS10/slide-dir/index.html

Textbook

- Advanced Programming in the UNIX Environments
- http://www.apuebook.com/



Chapter 5

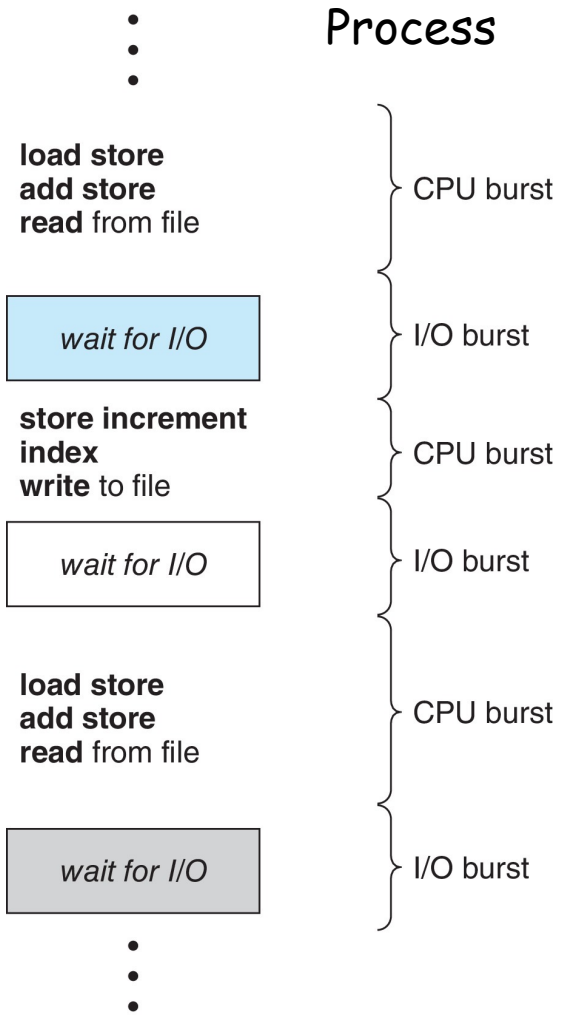
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Multi-Processor Scheduling
- Operating Systems Examples

Objectives

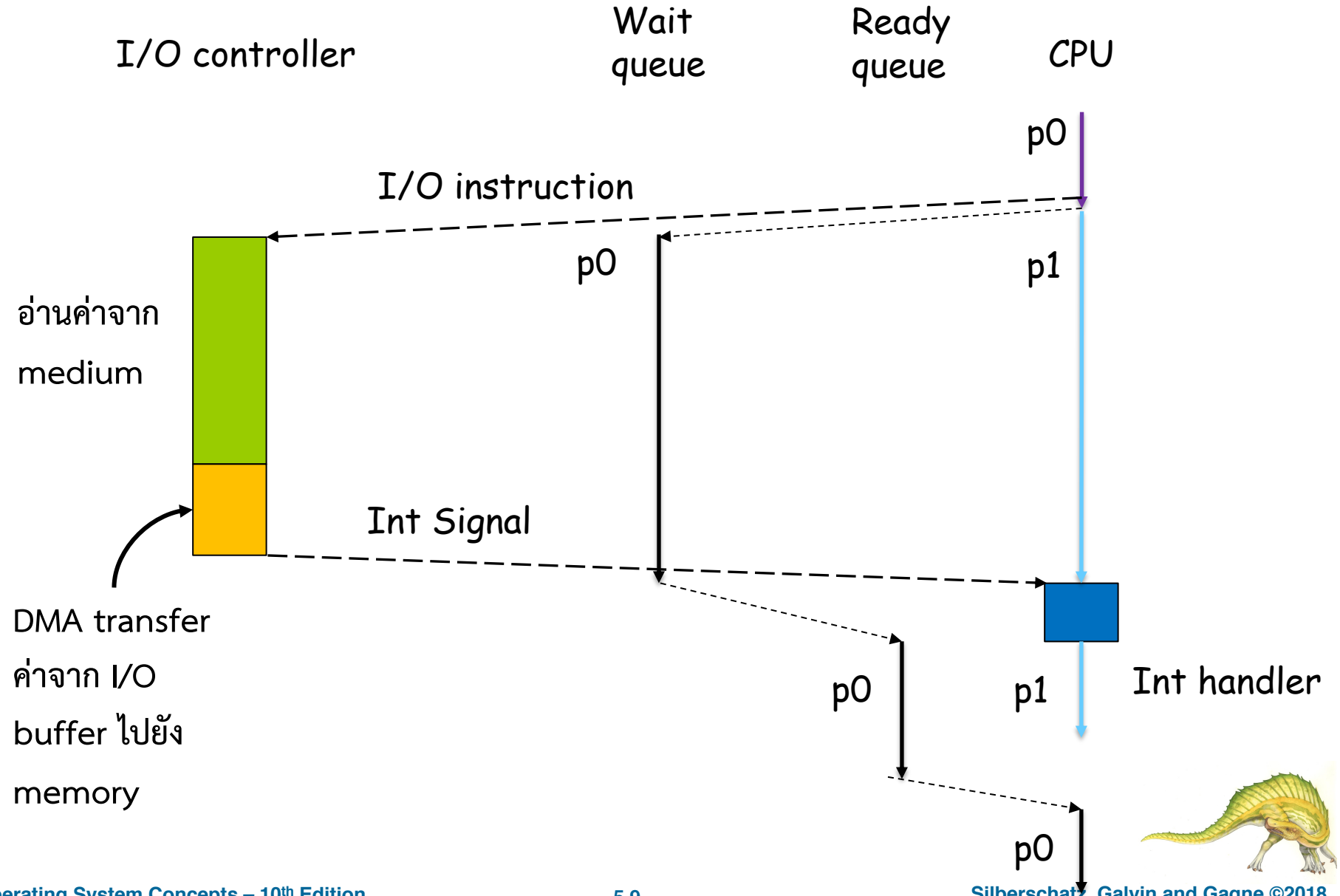
- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe the scheduling algorithms used in the Windows, Linux, and Solaris operating systems

Basic Concepts

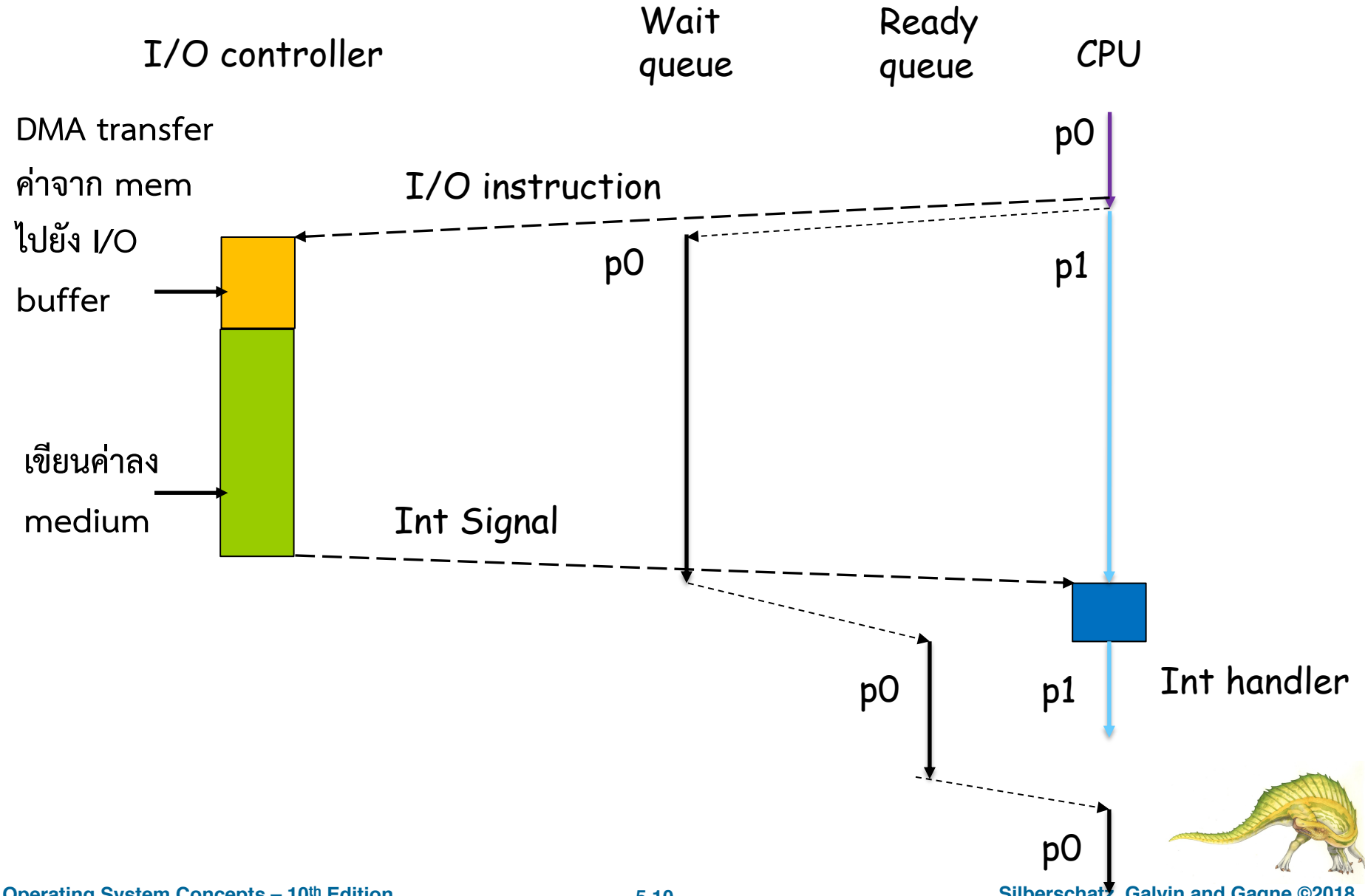
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



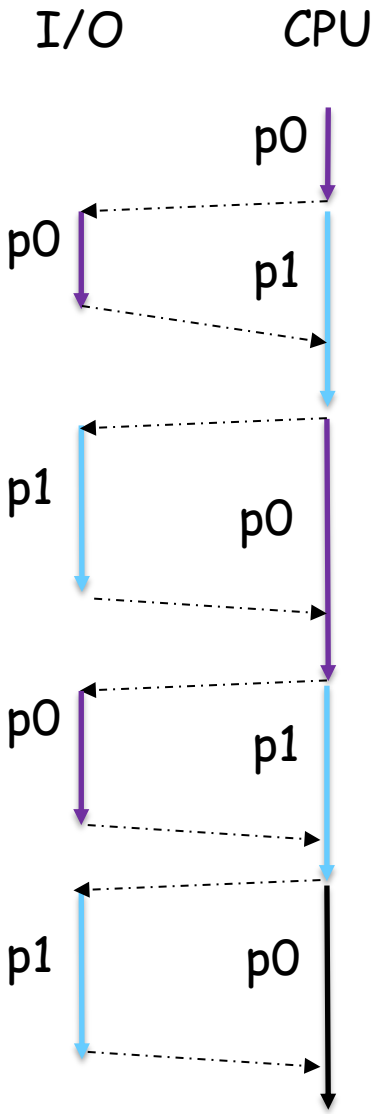
I/O read


I/O controller
Wait queue
Ready queue
CPU
p0
I/O instruction
p0
p1
อ่านค่าจาก medium
Int Signal
DMA transfer ค่าจาก I/O buffer ไปยัง memory
p0
p1
Int handler
p0

I/O write


I/O controller
Wait queue
Ready queue
CPU
DMA transfer
ค่าจาก mem
ไปยัง I/O
buffer
เขียนค่าลง
medium
I/O instruction
Int Signal
p0
p1
p0
p1
p0
p0
Int handler

CPU burst/IO burst of 2 processes



ถ้ามีมากกว่า 2 processes ต้องทำอย่างไร

Context switching ของระบบคอมพิวเตอร์

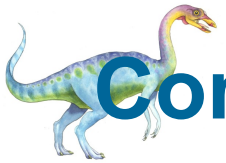
ให้ vmstat รายงานข้อมูลสถิติ ทุก 1 วินาที เป็นจำนวน 3 รายการ

จำนวน context switching
ต่อ 1 วินาที ตั้งแต่ boot เครื่อง

```
kasidit@cs22201:~$ man vmstat
kasidit@cs22201:~$
kasidit@cs22201:~$ vmstat 1 3
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 24342204  54704 4039724    0    0     0     1    1    1  0  0 100  0  0
 0  0      0 24342228  54704 4039724    0    0     0     0   43   46  0  0 100  0  0
 0  0      0 24342228  54704 4039724    0    0     0     0   35   45  0  0 100  0  0
kasidit@cs22201:~$ _
```

จำนวน context switching ในวินาทีที่ 1 ตั้งแต่ออกคำสั่ง

จำนวน context switching ในวินาทีที่ 2 ตั้งแต่ออกคำสั่ง



Context switching ของ Process (eg 242978)

```
kasidit@cs22201:~$ ls /proc/242978
arch_status  cmdline          exe      loginuid   mountstats  oom_score_adj
attr         comm             fd       map_files  net         pagemap
autogroup    coredump_filter  fdinfo   maps       ns          patch_state
auxv         cpuset           gid_map  mem        numa_maps   personality
cgroup       cwd              io       mountinfo  oom_adj     projid_map
clear_refs   environ          limits   mounts     oom_score   root
kasidit@cs22201:~$ cat /proc/242978/status | grep ctxt
voluntary_ctxt_switches:        24
nonvoluntary_ctxt_switches:     0
kasidit@cs22201:~$
kasidit@cs22201:~$
```

จำนวน context switching เช่นเกิดจาก I/O wait

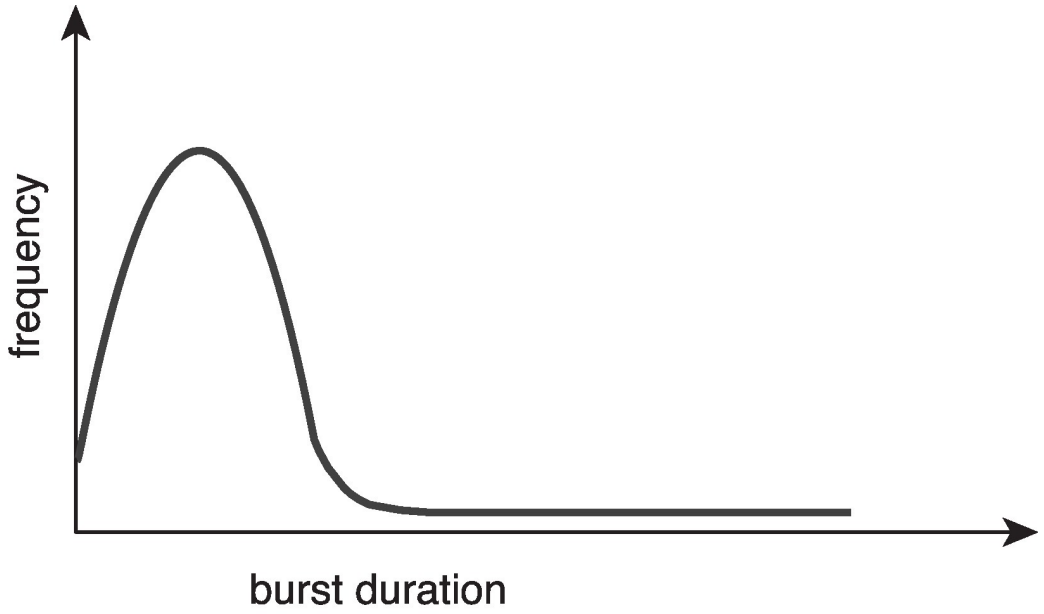
จำนวน context switching เช่นเกิดจาก Timer Interrupt

```
kasidit@cs22201:~$
kasidit@cs22201:~$
kasidit@cs22201:~$
kasidit@cs22201:~$ ps
    PID TTY          TIME CMD
 242978 pts/2    00:00:00 bash
 242989 pts/2    00:00:00 ps
kasidit@cs22201:~$
kasidit@cs22201:~$
```

Histogram of CPU-burst Times

Large number of short bursts

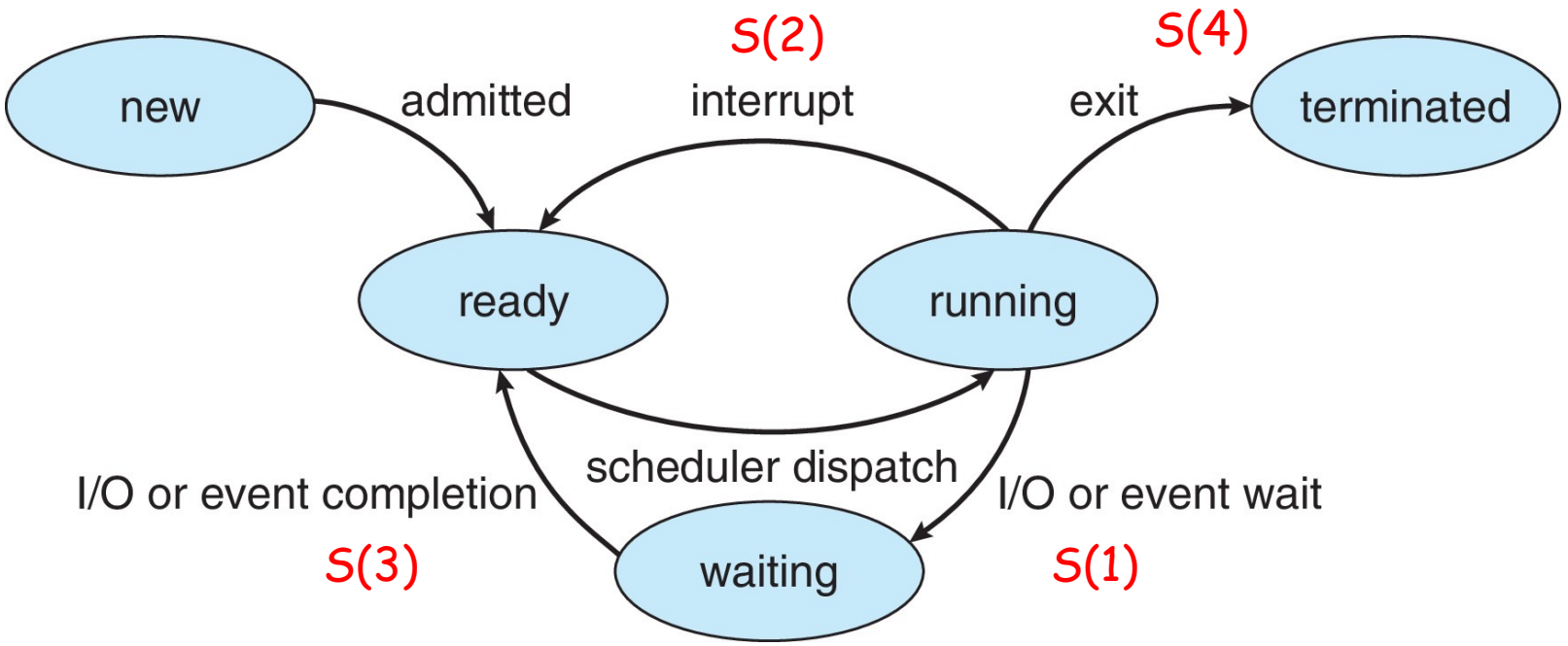
Small number of longer bursts



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
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state *S(1)*
  2. Switches from running to ready state *S(2)*
  3. Switches from waiting to ready *S(3)*
  4. Terminates *S(4)*
- For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution.
- For situations 2 and 3, however, there is a choice.

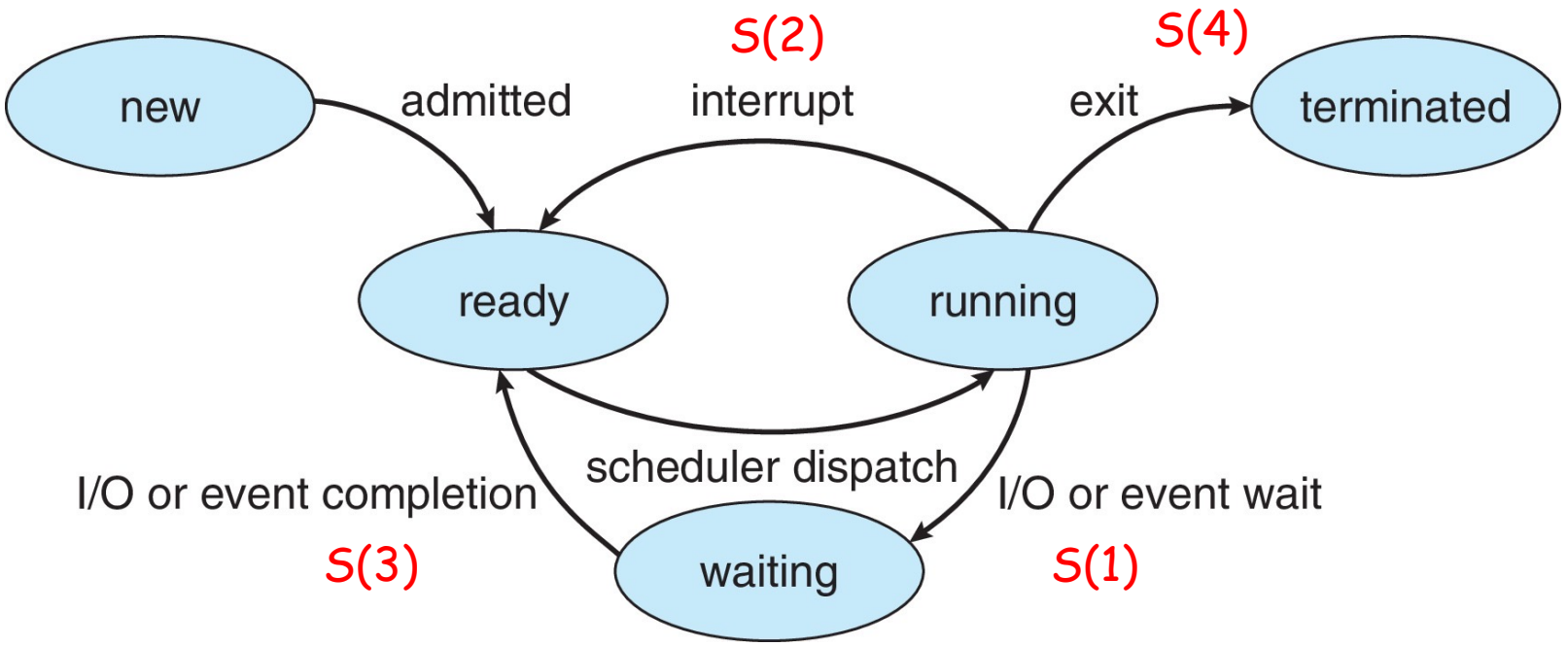
Diagram of Process State



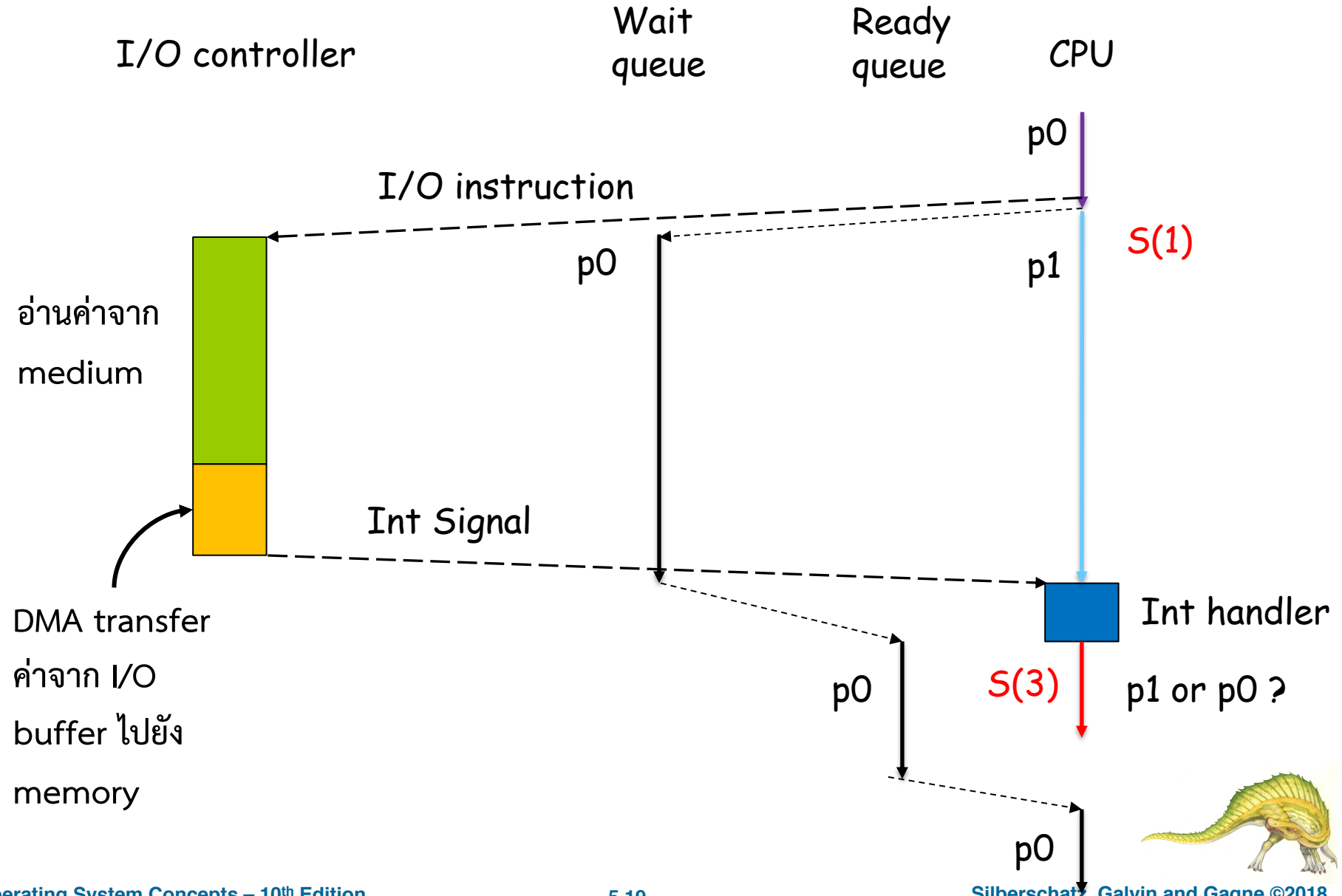
Preemptive and Nonpreemptive Scheduling

- When scheduling takes place only under circumstances 1 and 4, the scheduling scheme is **nonpreemptive**.
- Otherwise, it is **preemptive**.
- Under Nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
- Virtually all modern operating systems including Windows, MacOS, Linux, and UNIX use preemptive scheduling algorithms.
- **nonpreemptive**: เกิดจากการประมวลผลของ process เอง
- **preemptive**: เกิดจากการขัดจังหวะจากภายนอก process (เช่น จาก hardware หรือ OS หรือ ผู้ใช้)

Diagram of Process State



I/O read (s3 preemption)

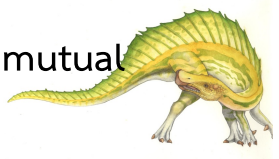


Preemptive Scheduling and Race Conditions

- Preemptive scheduling can result in race conditions when data are shared among several processes.
- Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.
- This issue will be explored in detail in Chapter 6.

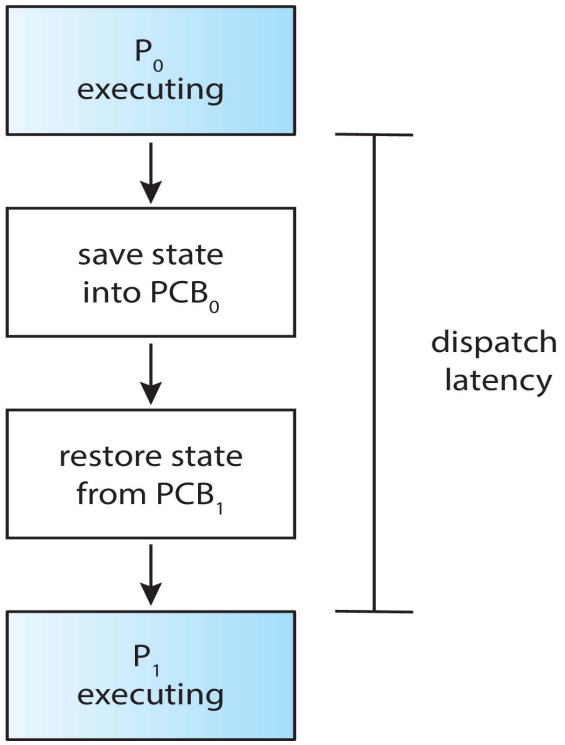
p0	p1
R1 ← A [10]	
	R2 ← A [10]
	R2++
	R2 → A [11]
R1++	
R1 → A [11] (ควรเป็น 12)	

- ในกรณีที่โพรเซส สองโพรเซส สามารถอ่านและเขียนข้อมูลบนพื้นที่ใดพื้นที่หนึ่งในหน่วยความจำได้เหมือนกัน อาจทำให้เกิดปัญหา race condition ได้
- ปกติแล้วโพรเซสจะแยก Isolate พื้นที่หน่วยความจำกัน แต่ (1) โพรเซสอาจขอให้ kernel สร้างพื้นที่สำหรับตัวแปรร่วม หรือ (2) โพรเซสมีหลายเทรดที่ใช้ตัวแปรร่วมกัน
- ปัญหา Race Condition (แย่ง) คือการที่โพรเซส 2 โพรเซสใดๆสามารถเข้าถึงทรัพยากรได้พร้อมกัน โดยที่โพรเซสหนึ่งอาจเข้าใช้ทรัพยากรนั้นในขณะมที่โพรเซสอื่นกำลังใช้งานทรัพยากรนั้นอยู่ (เหมือนแย่งใช้งาน โพรเซสไม่ยึดถือกติการใดในการเข้าใช้งานทรัพยากร)
- กรณีของ Preemptive Scheduling โพรเซส A ถูกขัดจังหวะ และโพรเซส B ถูกนำมารันใน CPU แต่ในขณะนั้น A อาจกำลังใช้งานตัวแปรร่วมอยู่ ดังนั้น B เข้ามาแทรกและแย่งใช้ค่าตัวแปรร่วมนั้นได้ ทำให้ผิดพลาด
- ต้องสร้างกติกาในการใช้ตัวแปรร่วม เช่น mutual exclusion



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
- เวลาที่ Dispatcher ใช้เพื่อหยุดโพรเซสที่กำลังใช้งานซีพียูอยู่จนถึงเวลาที่รันโพรเซสถัดไปที่มันเลือกบนซีพียู



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution 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.

Scheduling Algorithm Optimization Criteria

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

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

we will use a **Gantt Chart** to demonstrate the order in which processes are selected for scheduling according to the FCFS scheduling algorithm

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

process P1 is selected first as it is the first process to arrive; it begins running at time 0

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

process P1 runs to completion after which process P2 is selected to run at time 7

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

process P3 is selected next and begins running at time 10

Process	Burst Time
P1	~~7~~
P2	~~3~~
P3	~~12~~
P4	5
P5	9

Gantt Chart

at time 22, process P4 is selected to run

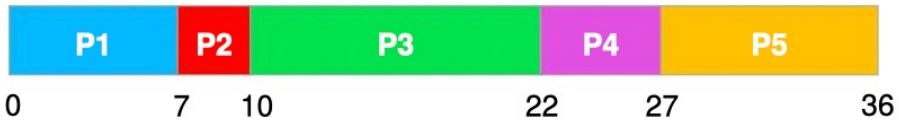
Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

because process P5 arrived last, it is the last process selected by the FCFS scheduling algorithm and begins running at time 27

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart



at time 36, process P5 finishes running

First- Come, First-Served (FCFS) Scheduling

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

Shortest-Job-First (SJF) Scheduling

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart

we will use a **Gantt Chart** to demonstrate the order in which processes are selected for scheduling according to the SJF scheduling algorithm

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

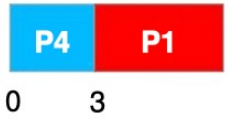
Gantt Chart

0

process P4 is selected first as it is the process with the shortest CPU burst; it begins running at time 0

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

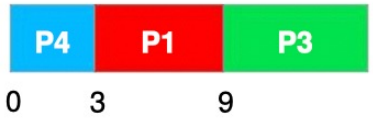
Gantt Chart



process P4 runs to completion, after which process P1 is selected as it has the next shortest CPU burst; process P1 begins to run at time 3

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart



process P1 runs to completion, and process P3 is selected as it has the next shortest CPU burst; process P3 begins to run at time 9

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart

as process P2 has the longest CPU burst, it is scheduled last and begins to run at time 16

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Gantt Chart

process P2 finishes executing at time 24

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
- 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
  - Could ask the user

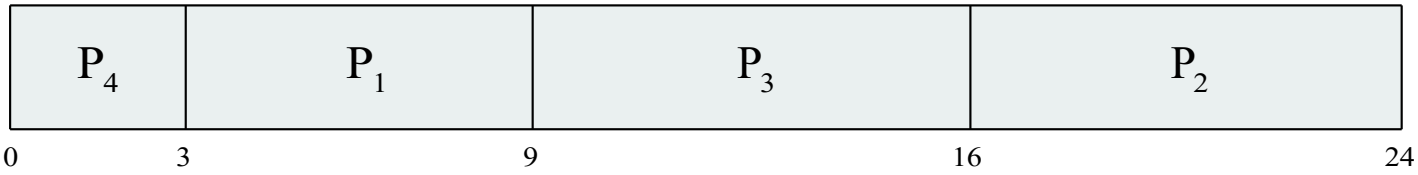
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- Preemptive version called **shortest-remaining-time-first**
- How do we determine the length of the next CPU burst?
  - Could ask the user
  - Estimate

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
  - 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 :

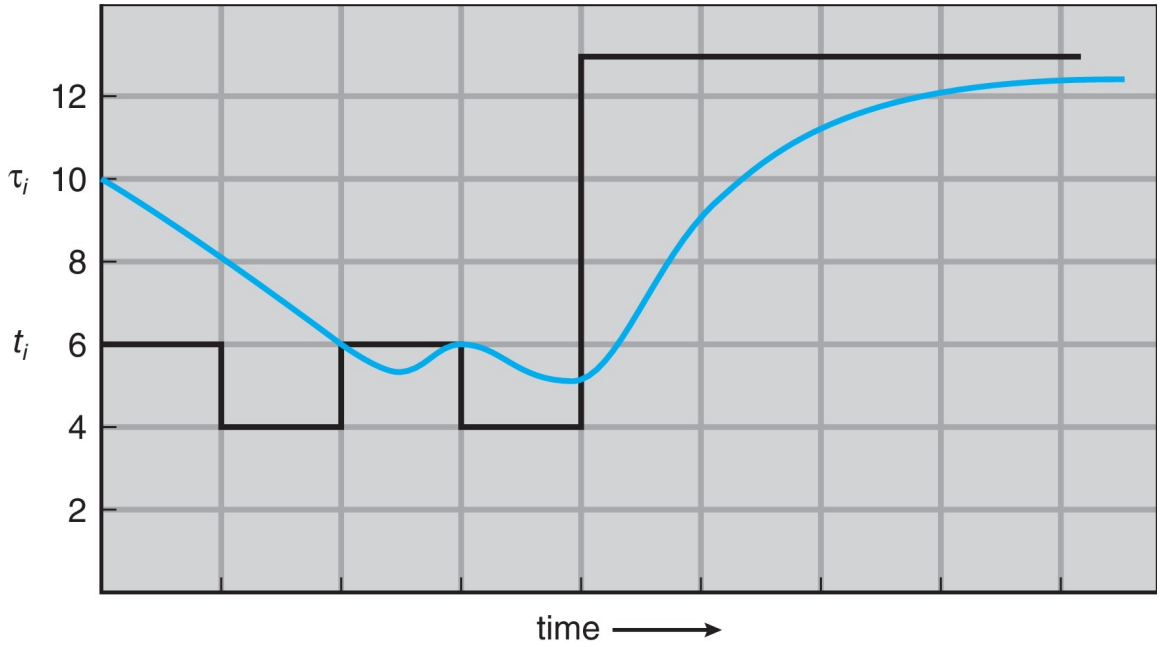
- Commonly, α set $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$
- สมมุติว่าสูตรนี้เป็นสูตร สำหรับคำณวนค่า อันดับตัวเต็งในการแข่งขันกีฬา
- ค่าอับดับตัวเต็งในรอบ n คือ τ_n
- ผลการแข่งจริงในรอบ n คือ t_n

เปรียบเทียบ: สูตรหาค่ามือวาง ของนักเทนนิส

- ให้ α เป็นความสำคัญของผลล่าสุด (ปัจจุบัน) = ½
- แสดงว่าเราให้ความสำคัญกับผลในอดีต = $1 - \alpha$ = ½
- ทัวร์นาเม้น 0: ก่อนแข่ง ให้ เป็นมือวางอันดับ 10 : $\tau_0 = 10$
- นักกีฬาได้สิทธิประโยชน์และค่าตัวตามอันดับมือวาง
- สูตรคำณวนค่าอันดับมือวา $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\tau_n.$

ครั้งที่	อันดับจากการแข่งจริง	อันดับมือวางใหม่
0		10
1	6	0.5x6 + 0.5x10 = 8
2	4	0.5x4 + 0.5x8 = 6
3	6	0.5x6 + 0.5x6 = 6
4	4	0.5x4 + 0.5x6 = 5
5	13	0.5x13 + 0.5x5 = 9
6	13	0.5x13 + 0.5x9 = 11

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	...

พิจารณา

อาจเป็นการบ้าน

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:

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

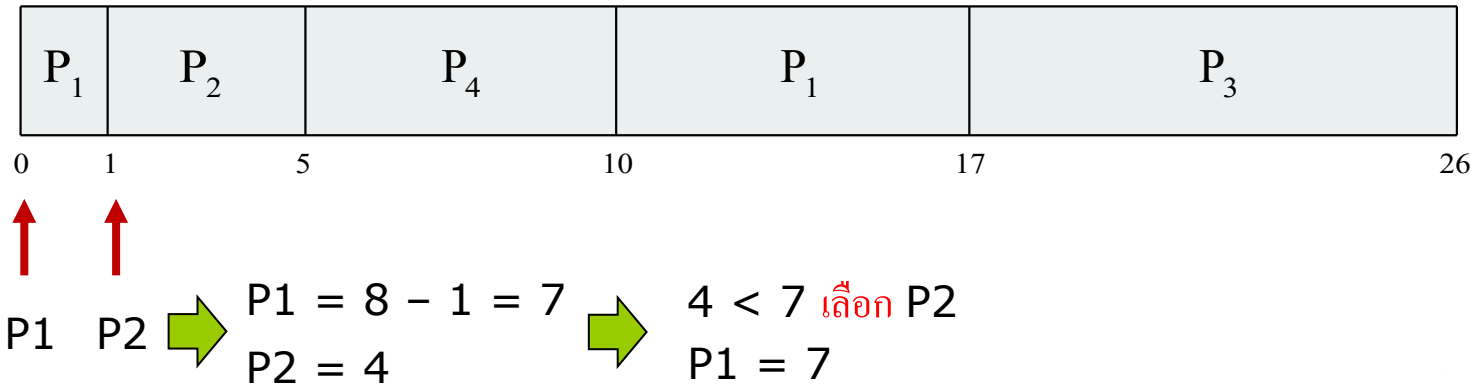
- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Example of 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

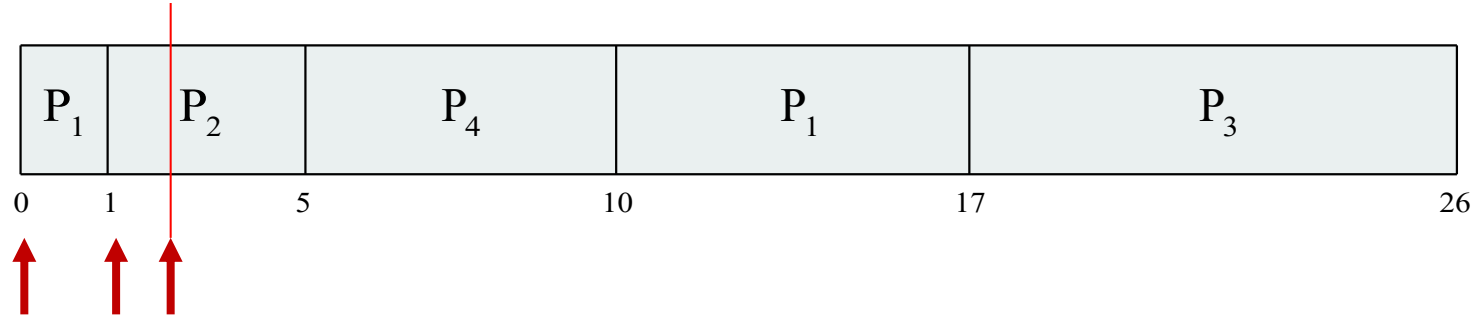


Example of Shortest-remaining-time-first

- 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

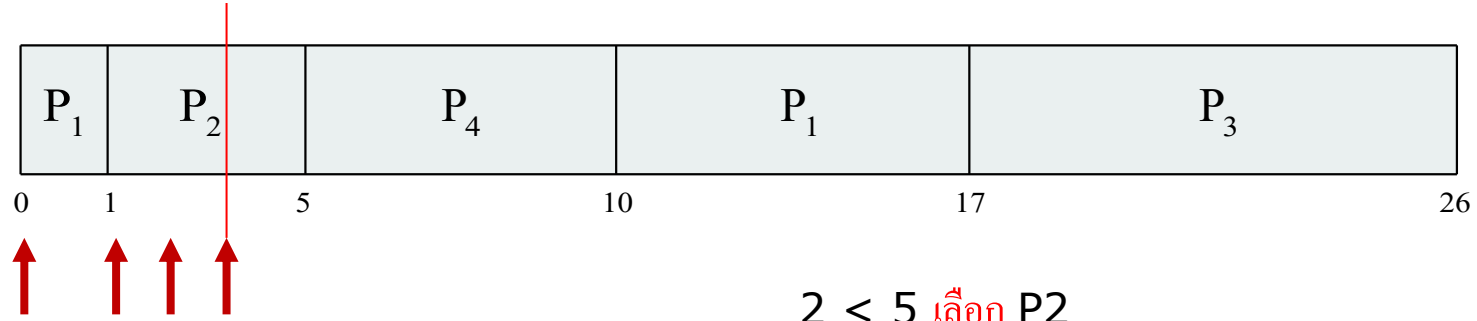


Example of Shortest-remaining-time-first

- 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

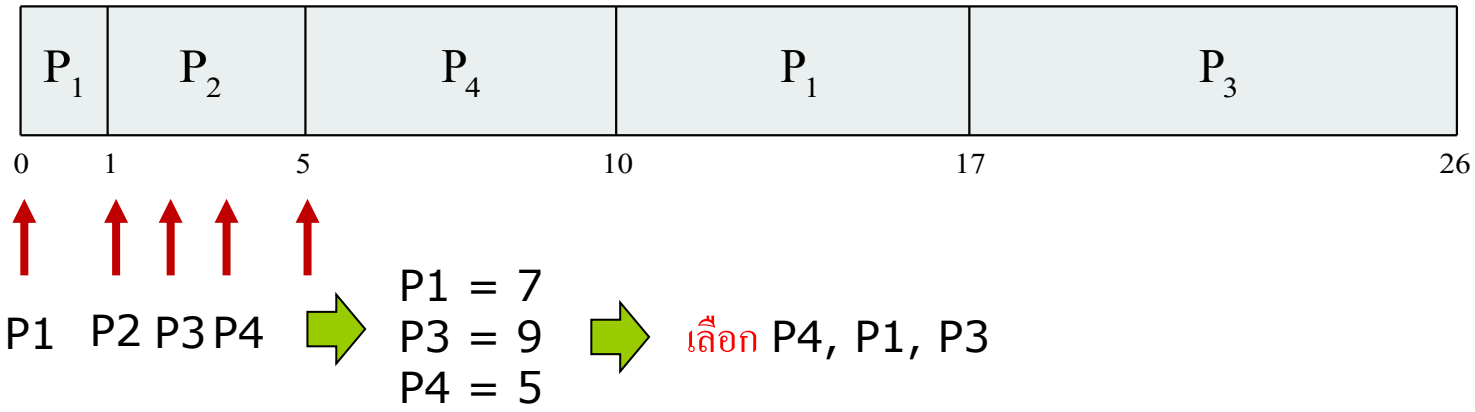


Example of Shortest-remaining-time-first

- 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

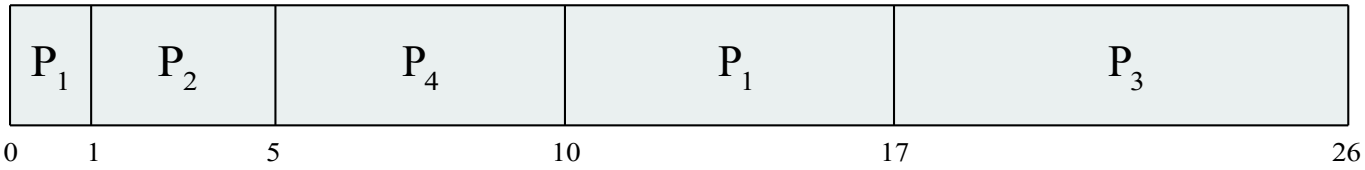


Example of Shortest-remaining-time-first

- 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

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*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large $\Rightarrow$ FIFO
  - *q* small $\Rightarrow$ *q* must be large with respect to context switch, otherwise overhead is too high

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

Gantt Chart

we will use a **Gantt Chart** to demonstrate the order in which processes are selected for scheduling according to the RR scheduling algorithm

Time Quantum

the **time quantum** is the amount of time a process can run before it is preempted and the CPU is assigned to another process

the time quantum for this example is three units of time

Process	Burst Time
P1	7
P2	3
P3	12
P4	5
P5	9

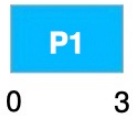
Gantt Chart

process P1 is selected first as it is the first process to arrive; it begins running at time 0 where it will run for the duration of a time quantum – three units

Process	Burst Time
P1	~~7~~ 4
P2	3
P3	12
P4	5
P5	9

process P1 has 4 units of time remaining in its CPU burst

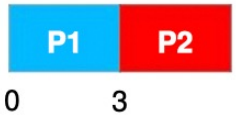
Gantt Chart



at time 3, process P1 is preempted as its time quantum has expired

Process	Burst Time
P1	~~7~~ 4
P2	3
P3	12
P4	5
P5	9

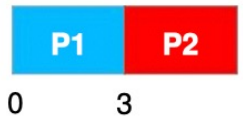
Gantt Chart



process P2 will run next, beginning at time 3

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	12
P4	5
P5	9

Gantt Chart

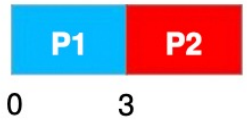


process P2 runs for the duration of a time quantum

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	12
P4	5
P5	9

process P2 completes its CPU burst

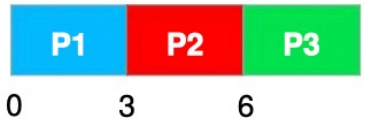
Gantt Chart



process P2 runs for the duration of a time quantum

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	12
P4	5
P5	9

Gantt Chart



process P3 is selected next

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	~~12~~ 9
P4	5
P5	9

Gantt Chart

followed by process P4 at time 9

Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	~~12~~ 9
P4	~~5~~ 2
P5	9

Gantt Chart

process P5 begins running at time 12

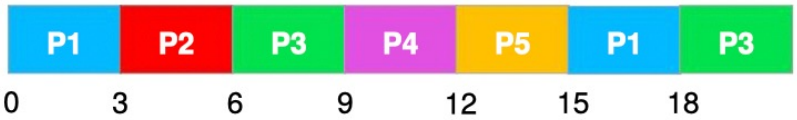
Process	Burst Time
P1	~~7~~ 4
P2	~~3~~ 0
P3	~~12~~ 9
P4	~~5~~ 2
P5	~~9~~ 6

Gantt Chart

process P1 runs for a time quantum beginning at time 15

Process	Burst Time
P1	~~7~~ ~~4~~ 1
P2	~~3~~ 0
P3	~~12~~ 9
P4	~~5~~ 2
P5	~~9~~ 6

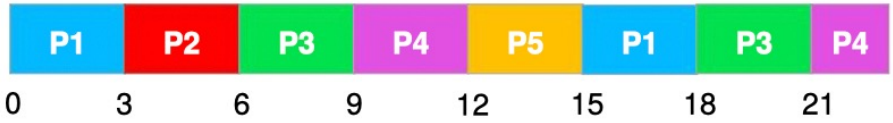
Gantt Chart



process P3 runs for a time quantum beginning at time 18

Process	Burst Time
P1	~~7~~ 4 1
P2	3 0
P3	~~12~~ 9 6
P4	5 2
P5	9 6

Gantt Chart

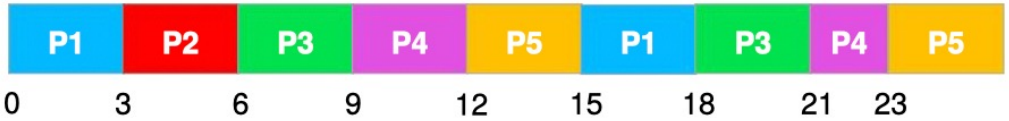


at time 21, process P4 begins to run

Process	Burst Time
P1	~~7~~ 4 1
P2	3 0
P3	~~12~~ 9 6
P4	~~5~~ 2 0
P5	9 6

process P4 only runs for the duration of its CPU burst

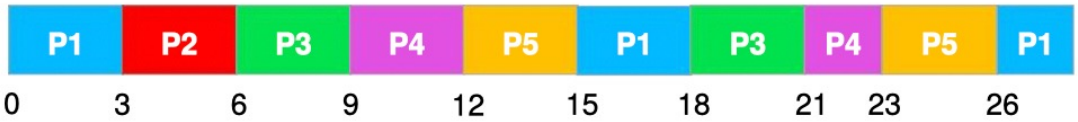
Gantt Chart



at time 23, process P5 runs

Process	Burst Time
P1	~~7~~ 4 1
P2	3 0
P3	~~12~~ 9 6
P4	~~5~~ 2 0
P5	9 6 3

Gantt Chart



at time 26, process P1 runs

Process	Burst Time
P1	~~7~~ ~~4~~ ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ ~~9~~ 6
P4	~~5~~ ~~2~~ 0
P5	~~9~~ ~~6~~ 3

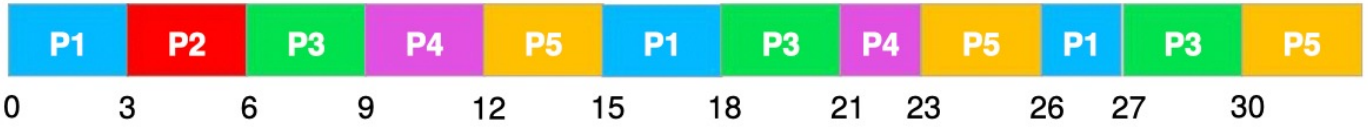
process P1 only runs for one time unit

Gantt Chart

at time 27, process P3 runs

Process	Burst Time
P1	~~7~~ 4 ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ 9 6 3
P4	~~5~~ ~~2~~ 0
P5	9 6 3

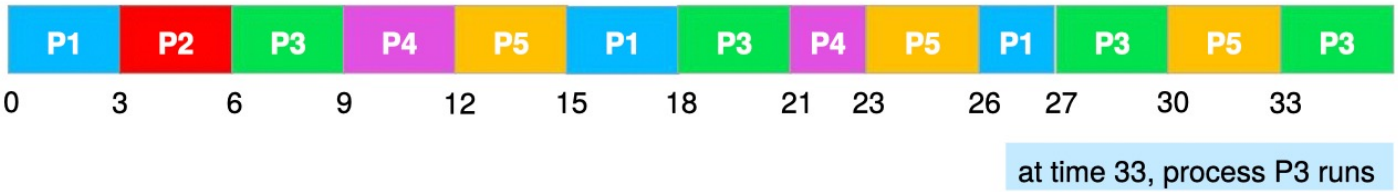
Gantt Chart



at time 30, process P5 runs

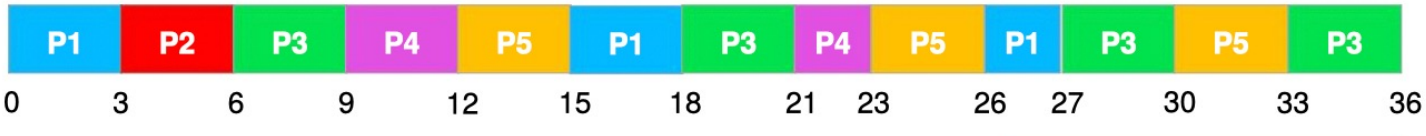
Process	Burst Time
P1	~~7~~ 4 ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ 9 6 3
P4	~~5~~ 2 0
P5	~~9~~ 6 3 0

Gantt Chart



Process	Burst Time
P1	~~7~~ ~~4~~ ~~1~~ 0
P2	~~3~~ 0
P3	~~12~~ ~~9~~ ~~6~~ ~~3~~ 0
P4	~~5~~ ~~2~~ 0
P5	~~9~~ ~~6~~ ~~3~~ 0

Gantt Chart

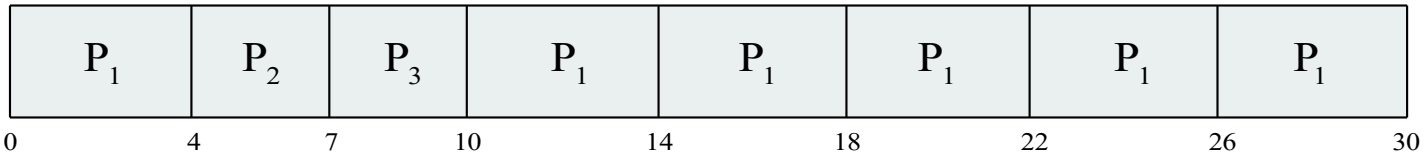


process P3 finishes at time 36

Example of RR with Time Quantum = 4

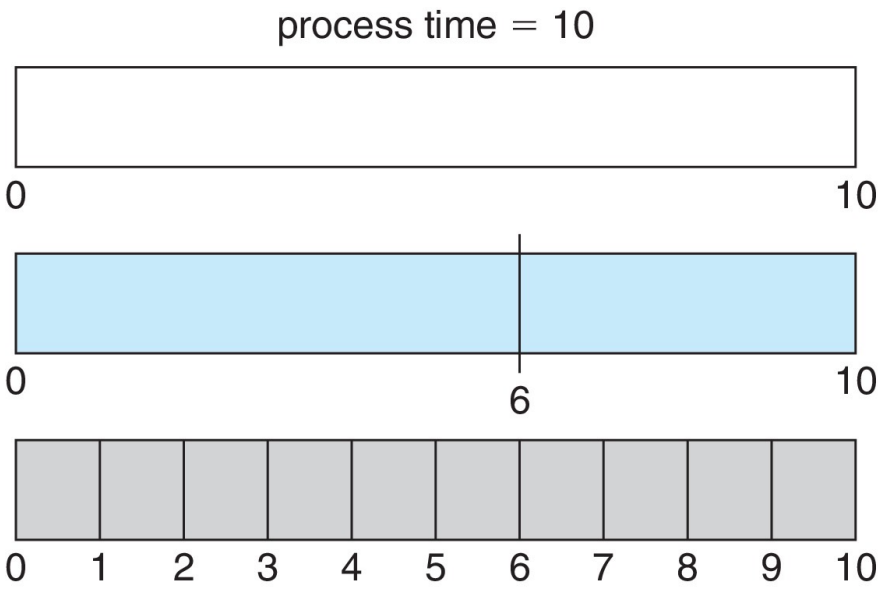
Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:



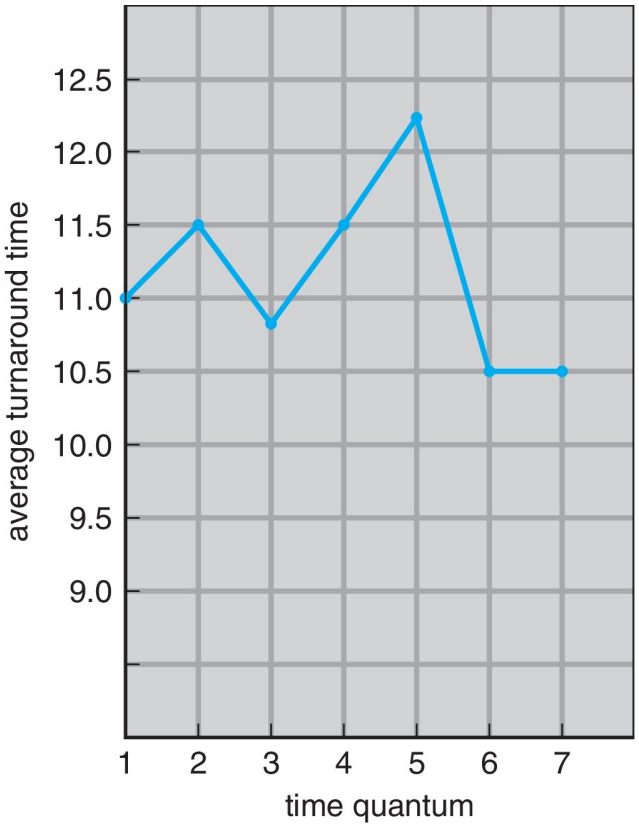
- Typically, higher average turnaround than SJF, but better ***response***
- q should be large compared to context switch time
  - q usually 10 milliseconds to 100 milliseconds,
  - Context switch < 10 microseconds

Time Quantum and Context Switch Time



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

พิจารณา

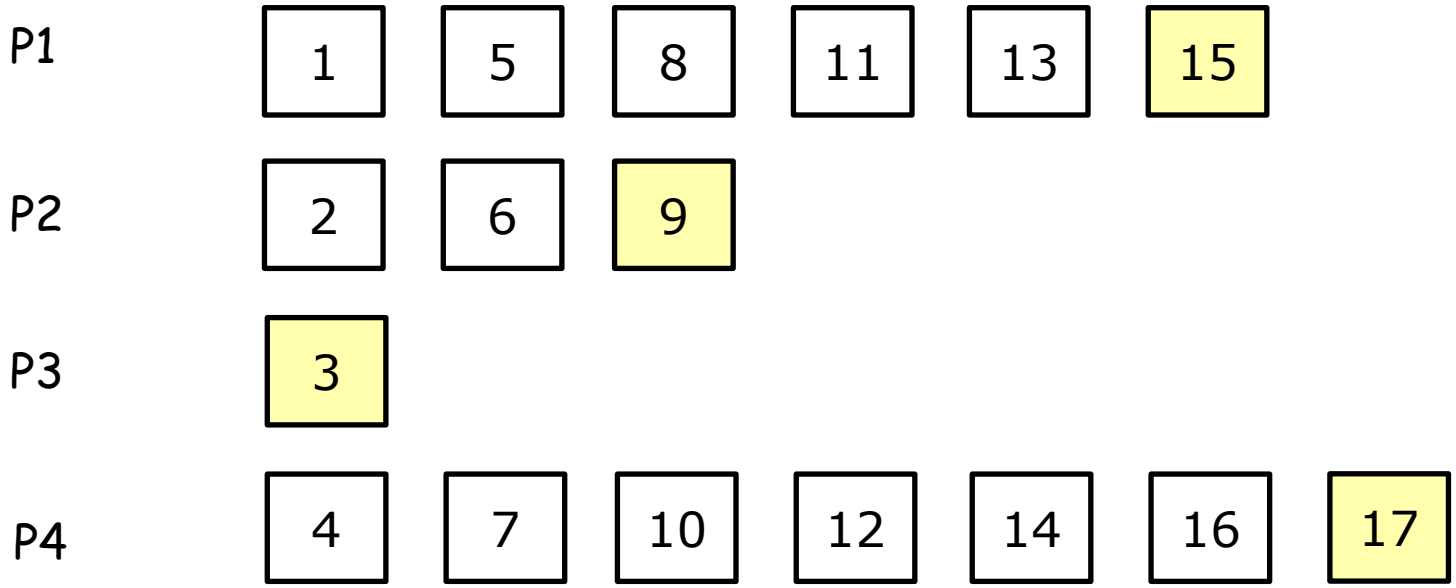
อาจเป็นการบ้าน

คำณวน avg turnaround time

Waiting time

Quantum = 1

process	time
P_1	6
P_2	3
P_3	1
P_4	7



Avg.Turnaround = (Trnd(P3)+Trnd(P2)+Trnd(P1)+Trnd(P4))/4
= (3 + 9 + 15 + 17)/4
= 44/4 = 11

คำณวน avg turnaround time
เมื่อ quantum = 2, 3, 4, etc.

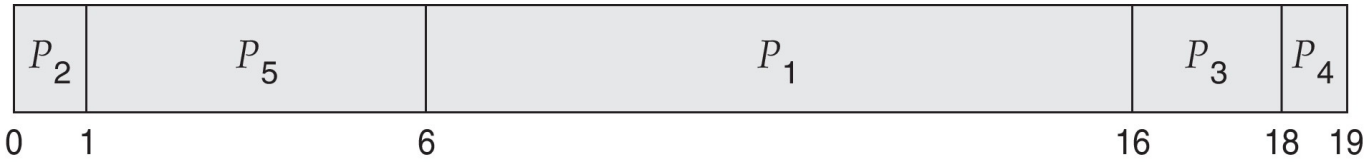
Priority Scheduling

- 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 priority scheduling where priority is the inverse of predicted next CPU burst time
  - ในกรณีนี้ เรามอง ค่าไพออริตี้ต่ำ คือค่า index ที่เป็นค่าต่ำ
- ต่อไปนี้เราใช้ข้อตกลงว่า ค่า index ต่ำหมายถึง Priority สูง และ index สูงหมายถึง Priority ต่ำ
- 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

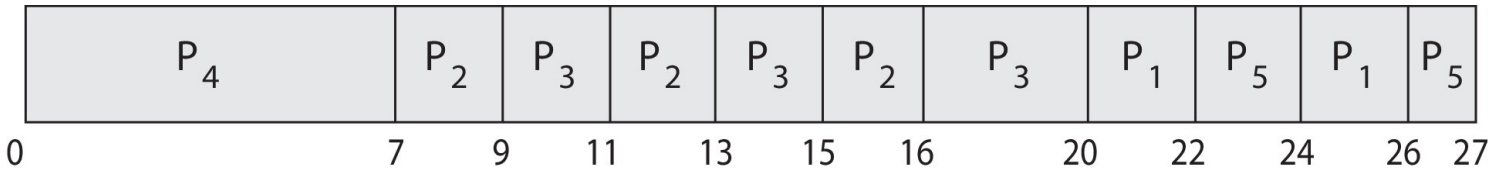


- Average waiting time = 8.2 = (6+0+16+18+1)/4

Priority Scheduling w/ Round-Robin

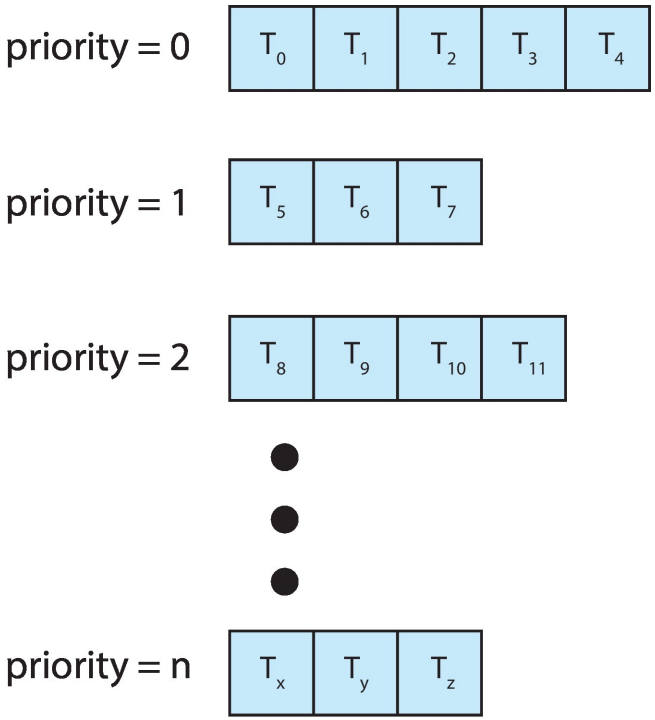
Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin
- Gantt Chart with time quantum = 2



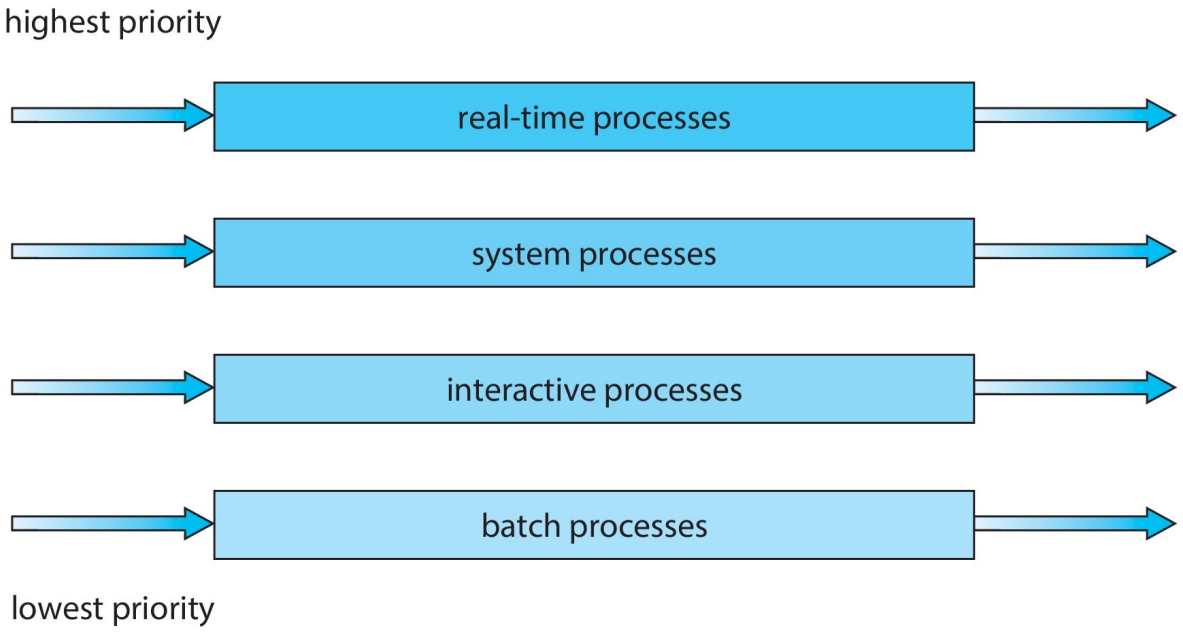
Multilevel Queue

- With priority scheduling, have separate queues for each priority.
- Schedule the process in the highest-priority queue!



Multilevel Queue

- Prioritization based upon process type

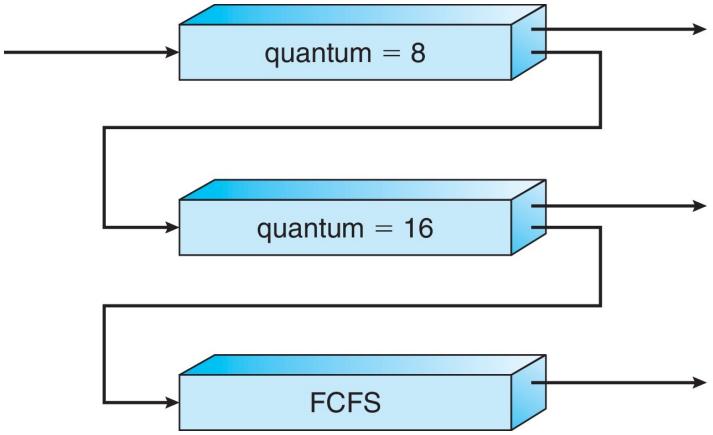


Multilevel Feedback Queue

- 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 to a higher priority queue
  - Method used to determine when to demote a process to a lower priority queue
  - 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
- 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

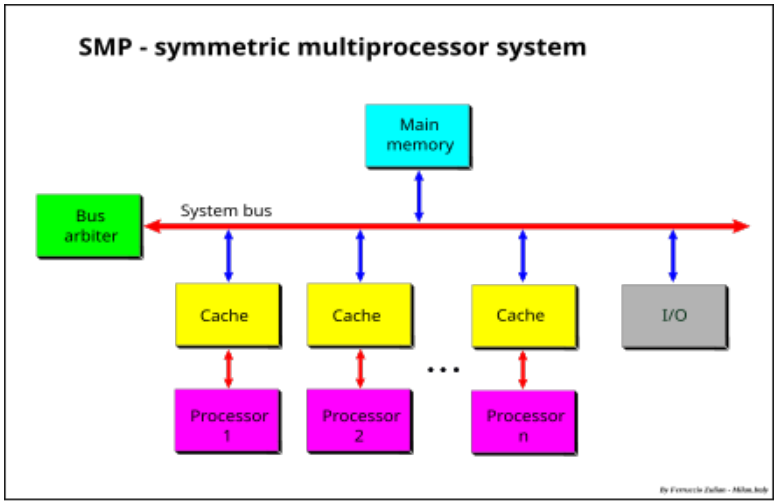


Multiple-Processor Scheduling

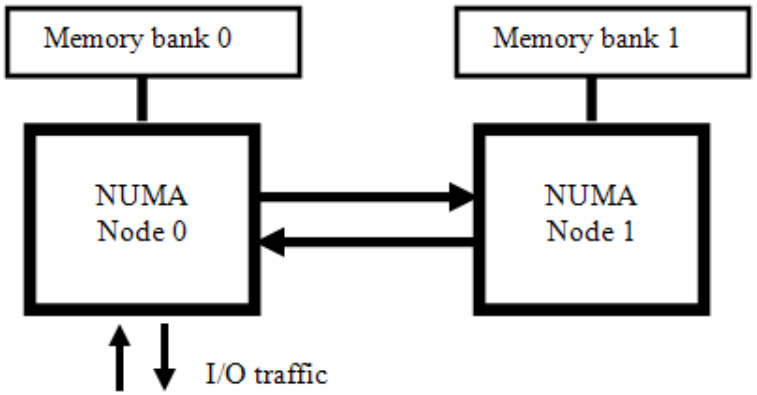
- CPU scheduling more complex when multiple CPUs are available
- Multiprocessors may be any one of the following architectures:
  - Multicore CPUs
  - Multithreaded cores
  - NUMA systems
  - Heterogeneous multiprocessing

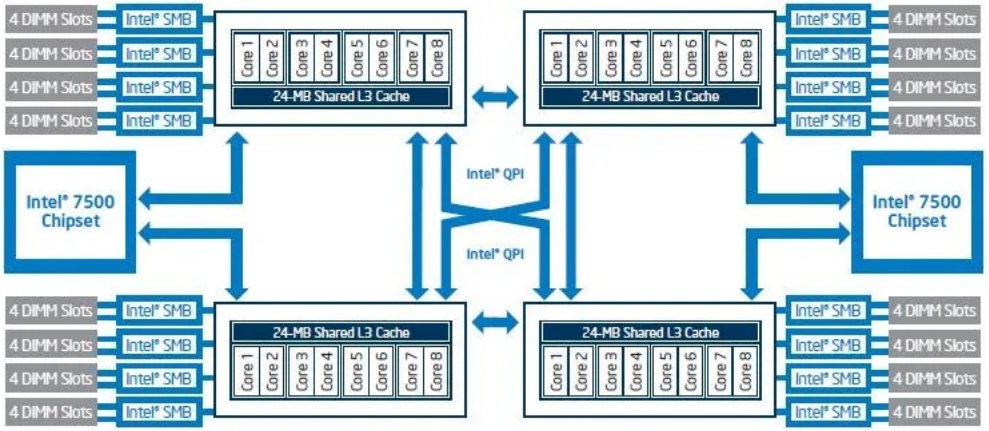
Types of MultiProcessors

- Multiprocessors หมายถึงระบบคอมพิวเตอร์มี CPU หลาย CPU
- ถ้า CPU เหล่านั้นใช้ Memory ร่วมกันและใช้เวลาเข้าถึง Memory เท่ากัน เราเรียกว่า Symmetric Multiprocessors (SMP)
- ถ้า CPU เหล่านั้นใช้เวลาเข้าถึง Memory ต่างกันเราเรียกมันว่า Non-Uniform Memory Access (NUMA)



https://en.wikipedia.org/wiki/Symmetric_multiprocessing

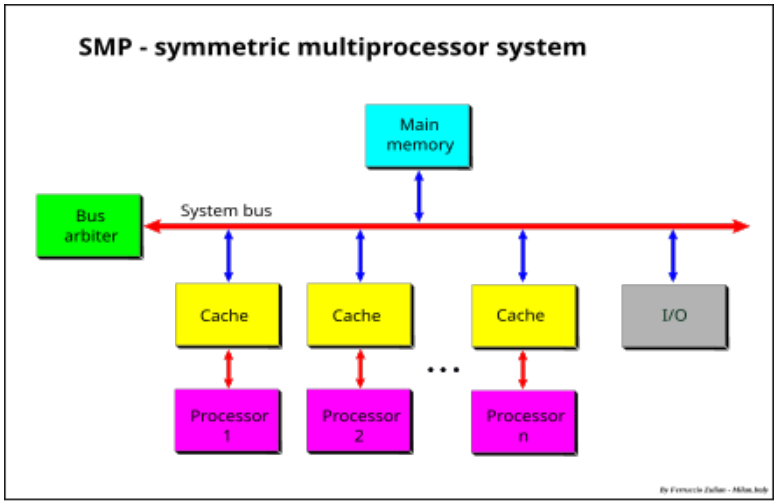




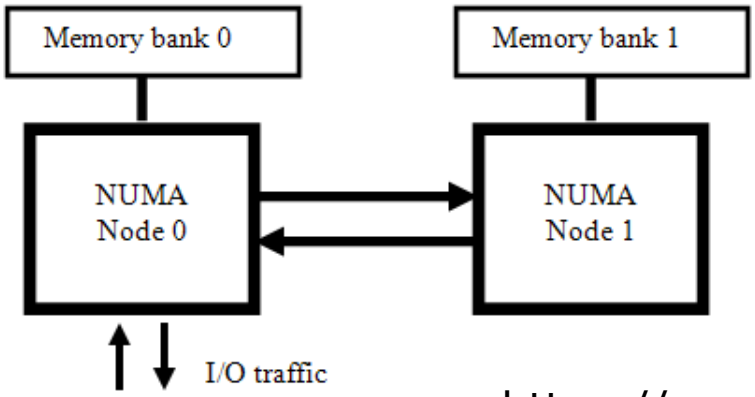
https://en.namu.wiki/w/NUMA

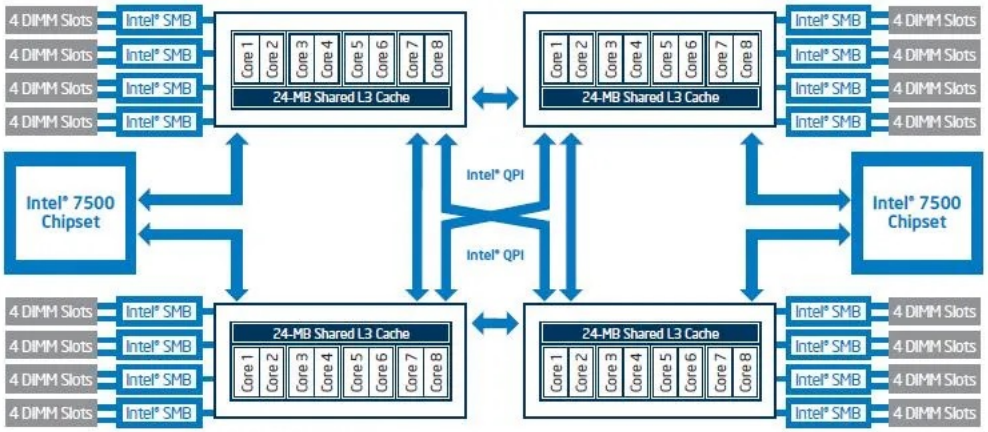
UMA vs. NUMA

- Multiprocessors หมายถึงระบบคอมพิวเตอร์มี CPU หลาย CPU
- ถ้า CPU เหล่านั้นใช้ Memory ร่วมกันและใช้เวลาเข้าถึง Memory เท่ากัน เราเรียกว่า **Symmetric Multiprocessors (SMP)** หรือ **Uniform Memory Access (UMA)**
- ถ้า CPU เหล่านั้นใช้เวลาเข้าถึง Memory ต่างกันเราเรียกมันว่า **Non-Uniform Memory Access (NUMA)**



https://en.wikipedia.org/wiki/Symmetric_multiprocessing

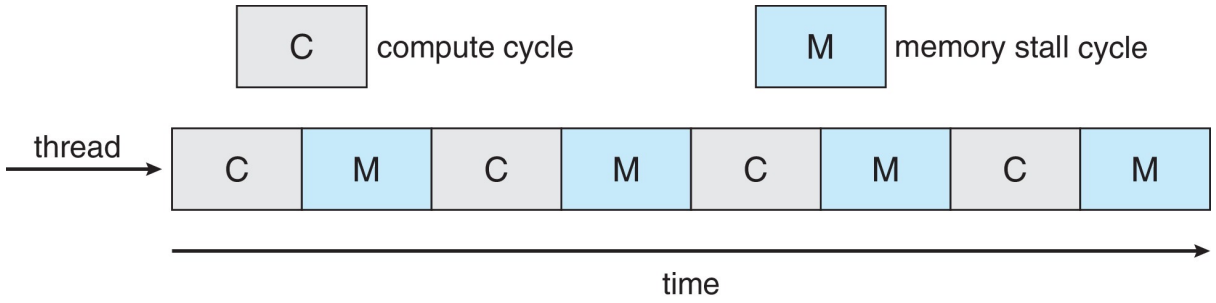




https://en.namu.wiki/w/NUMA

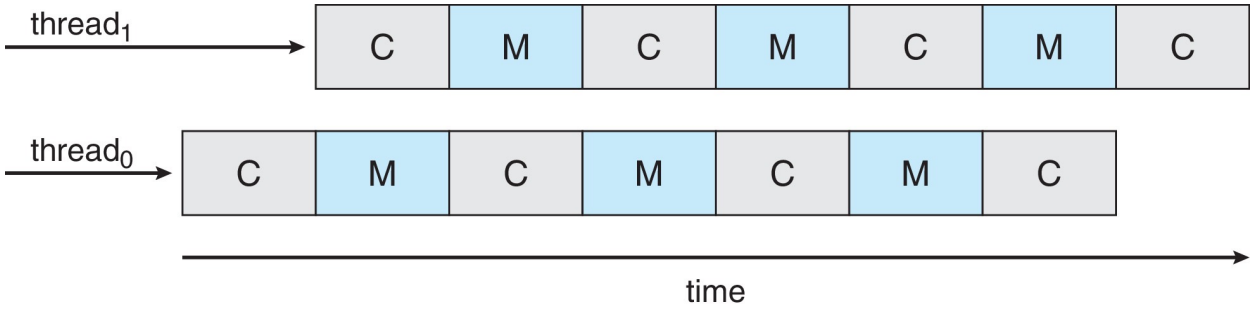
Multicore Processors

- 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



Multithreaded Multicore System

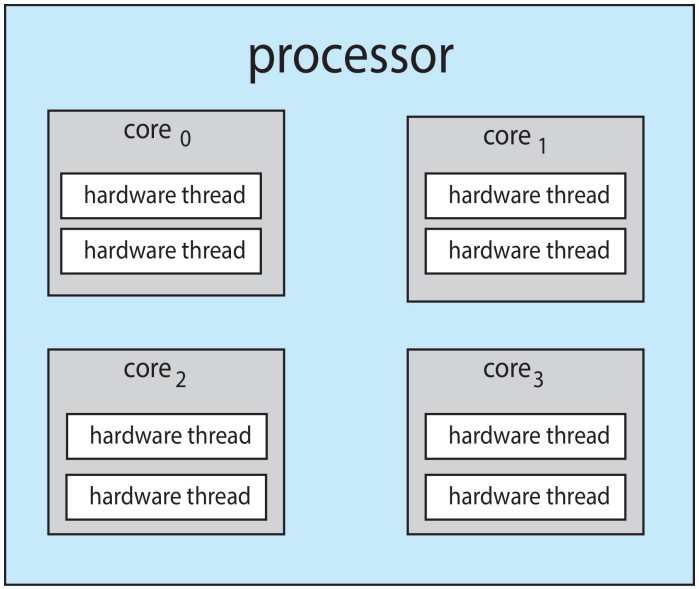
- Each core has > 1 hardware threads.
- If one thread has a memory stall, switch to another thread!
- Figure

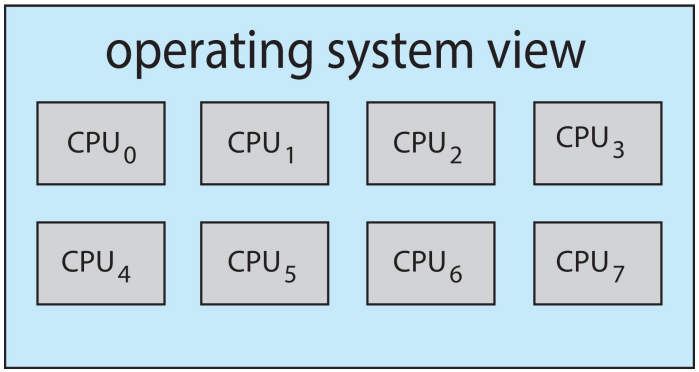


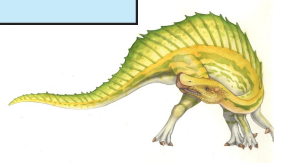
Multithreaded Multicore System

- **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.**

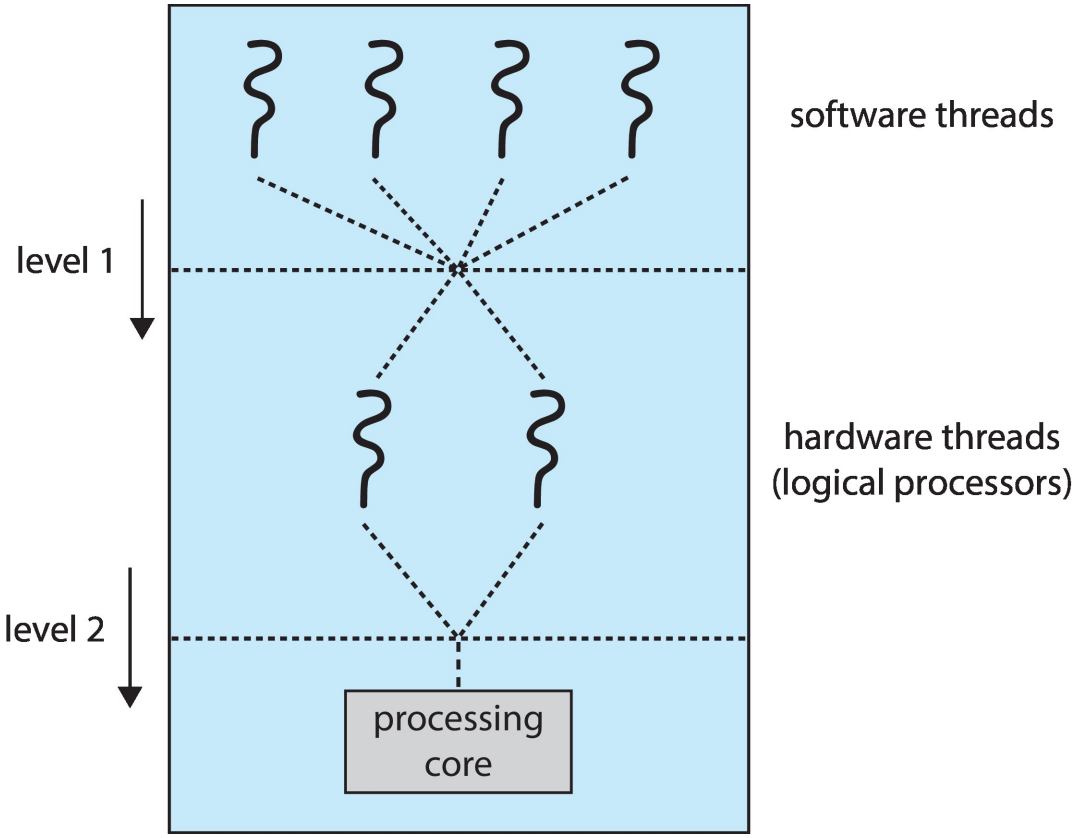






Multithreaded Multicore System

- 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.

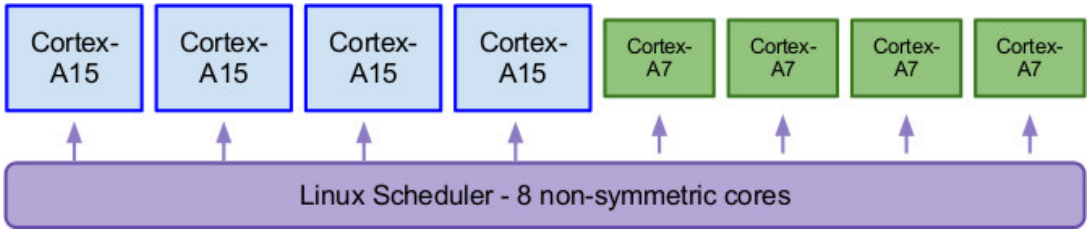


Multicore VS Mutithreaded cores

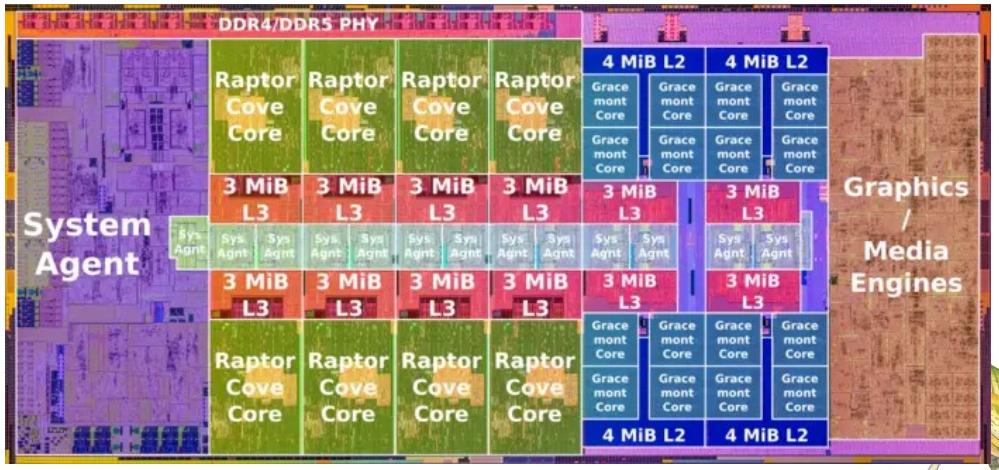
- Multicore CPU คือ CPU ที่มี หลาย core อยู่ใน CPU เดียวกัน
- Multithreaded cores คือ การที่ผู้ผลิต CPU ผลิต CPU แบบที่มีหลาย Physical Cores แต่ออกแบบให้แต่ละ Physical Core รองรับการประมวลผลเป็น 2 Threads
- Technology นี้คิดค้นโดย Sun Microsystem
- Intel เรียกมันว่า Hyper-Threading Technology
  - มีทั้งใน Xeon CPU และ Core i3 to Core i9 CPU
- AMD เรียกมันว่า Simultaneous Mutithreading Technology (SMT)
  - มีทั้งใน EPYC CPU และ Ryzen 3, 5, 9 และ Ryzen Threadripper

Heterogeneous Multiprocessors

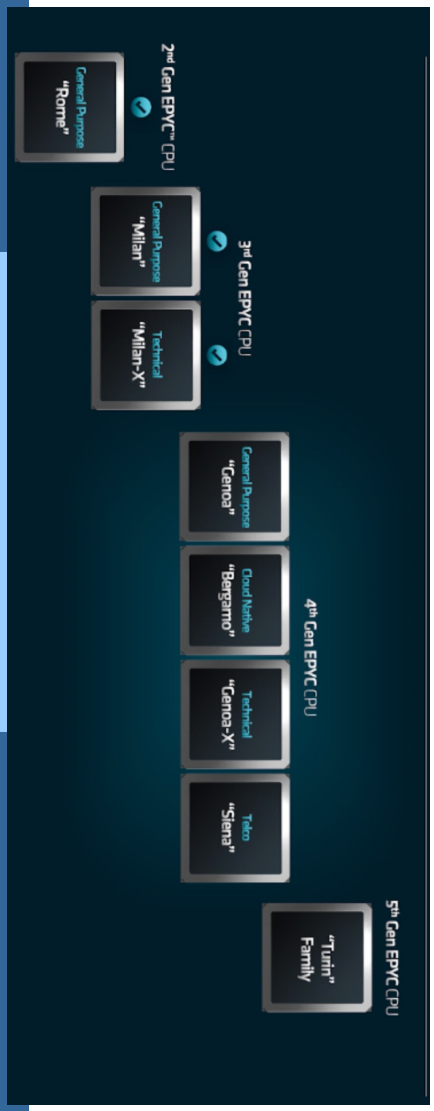
- Heterogeneous Multiprocessors (HMP) คือระบบ Multicore processor CPU ที่ประกอบไปด้วย cores ที่ทุก core ประมวลผลชุดคำสั่ง(ISA) แบบเดียวกัน แต่มี clock speed ไม่เท่ากัน
- ARM big.LITTLE
  - ในมือถือเช่น sumsung



- Intel P-core/E-core
  - 13th gen
  - Raptor Lake
  - ใน notebook/desktop



AMD Epyc uses SMT



AMD Epyc CPU generations[15][16][17][18][19]

Gen	Year	Codename	Product line	Cores	Socket	Memory
Server						ECC
1st	2017	Naples	7001 series	32 × Zen	SP3 (LGA)	DDR4
2nd	2019	Rome	7002 series	64 × Zen 2		
3rd	2021	Milan	7003 series	64 × Zen 3		
	2022	Milan-X				
4th	2022	Genoa	9004 series	96 × Zen 4	SP5 (LGA)	DDR5
	2023	Genoa-X				
		Bergamo		128 × Zen 4c		
		Siena	8004 series	64 × Zen 4c	SP6 (LGA)	
5th	2024	Turin	9005 series	192 × Zen 5	SP5 (LGA)	
6th	TBA	Venice	TBA	TBA	SP7 (LGA)	TBA

https://en.wikipedia.org/wiki/Epyc

AMD Ryzen Zen 4 (TSMC's 5 nm)

Branding and model		Cores (threads)	Clock rate (GHz)		L3 cache per CCD	Thermal solution	TDP	Chiplets	Core config[ii]	Release date	MSRP
			Base	Boost							
Ryzen 9	7950X3D	16 (32)	4.2	5.7	32+96 MB[iii]	—	120 W	2 × CCD 1 × I/OD	2 × 8	Feb 28, 2023	US $699
	7950X		4.5		32+32 MB		170 W			Sep 27, 2022	
	7900X3D	12 (24)	4.4	5.6	32+96 MB[iii]		120 W		2 × 6	Feb 28, 2023	US $599
	7900X		4.7		32+32 MB		170 W			Sep 27, 2022	US $549
	7900		3.7	5.4		Wraith Prism	65 W			Jan 10, 2023	US $429[49]
	PRO 7945					Wraith Spire				Jun 13, 2023	OEM
Ryzen 7	7800X3D	8 (16)	4.2	5.0	96 MB	—	120 W	1 × CCD 1 × I/OD	1 × 8	Apr 6, 2023	US $449
	7700X		4.5	5.4	32 MB		105 W			Sep 27, 2022	US $399
	7700		3.8	5.3		Wraith Prism	65 W			Jan 10, 2023	US $329[49]
	PRO 7745					Wraith Spire				Jun 13, 2023	OEM
Ryzen 5	7600X	6 (12)	4.7			—	105 W		1 × 6	Sep 27, 2022	US $299
	7600		3.8	5.1		Wraith Stealth	65 W			Jan 10, 2023	US $229[49]
	PRO 7645					Wraith Spire				Jun 13, 2023	OEM
	7500F		3.7	5.0		Wraith Stealth				Jul 22, 2023	US $179[50]

AMD Threadripper Zen 4 (TSMC's 5 nm)

Branding and Model		Cores (threads)	Clock rate (GHz)		L3 cache (total)	TDP	Chiplets	Core config[i]	Release date	MSRP
			Base	Boost						
Ryzen Threadripper PRO	7995WX [51]	96 (192)	2.5	5.1	384 MB	350 W	12 × CCD 1 × I/OD	12 × 8	November 21, 2023	US $9999
	7985WX [51]	64 (128)	3.2		256 MB		8 × CCD 1 × I/OD	8 × 8		US $7349
	7975WX [51]	32 (64)	4.0	5.3	128 MB		4 × CCD 1 × I/OD	4 × 8		US $3899
	7965WX [51]	24 (48)	4.2					4 × 6		US $2649
	7955WX [51]	16 (32)	4.5		64 MB		2 × CCD 1 × I/OD	2 × 8		US $1899
	7945WX [51]	12 (24)	4.7					2 × 6		US $1399
Ryzen Threadripper	7980X [51]	64 (128)	3.2	5.1	256 MB		8 × CCD 1 × I/OD	8 × 8		US $4999
	7970X [51]	32 (64)	4.0	5.3	128 MB		4 × CCD 1 × I/OD	4 × 8		US $2499
	7960X [51]	24 (48)	4.2					4 × 6		US $1499

Intel Xeon uses Hyperthreading

Intel 5th Gen Emerald Rapids Xeon CPU Specs "Official":

ECC

CPU NAME	CORES / THREADS	CACHE	BASE / BOOST (MAX)	DDR5 SUPPORT	TDP	PRICE (RCP)
Xeon Platinum 8593Q	64/128	320 MB	2.2 / 3.9 GHz	DDR5-5600	385W	$12400
Xeon Platinum 8592+	64/128	320 MB	1.9 / 3.9 GHz	DDR5-5600	350W	$11600
Xeon Platinum 8592V	64/128	320 MB	2.0 / 3.9 GHz	DDR5-4800	330W	$10995

Intel Xeon

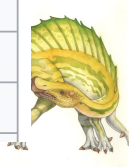
Model number	Cores (Threads)	Base clock	All core turbo boost	Max turbo boost	Smart Cache	TDP	Maximum scalability	Registered DDR5 w. ECC support	UPI links	Release MSRP (USD)
Xeon Platinum (8500)										
8593Q	64 (128)	2.2 GHz	3.0 GHz	3.9 GHz	320 MB	385 W	2S	5600 MT/s	4	$12400
8592+	64 (128)	1.9 GHz	2.9 GHz	3.9 GHz	320 MB	350 W	2S	5600 MT/s	4	$11600
8592V	64 (128)	2.0 GHz	2.9 GHz	3.9 GHz	320 MB	330 W	2S	4800 MT/s	3	$10995
8581V	60 (120)	2.0 GHz	2.6 GHz	3.9 GHz	300 MB	270 W	1S	4800 MT/s	0	$7568
8580	60 (120)	2.0 GHz	2.9 GHz	4.0 GHz	300 MB	350 W	2S	5600 MT/s	4	$10710
Xeon Gold (5500 and 6500)										
6558Q	32 (64)	3.2 GHz	4.1 GHz	4.1 GHz	60 MB	350 W	2S	5200 MT/s	3	$6416
6554S	36 (72)	2.2 GHz	3.0 GHz	4.0 GHz	180 MB	270 W	2S	5200 MT/s	4	$3157
6548Y+	32 (64)	2.5 GHz	3.5 GHz	4.1 GHz	60 MB	250 W	2S	5200 MT/s	3	$3726
6548N	32 (64)	2.8 GHz	3.5 GHz	4.1 GHz	60 MB	250 W	2S	5200 MT/s	3	$3875
6544Y	16 (32)	3.6 GHz	3.6 GHz	4.1 GHz	45 MB	270 W	2S	5200 MT/s	3	$3622

Intel CPU Desktop Hyperthread (P/E cores)

Desktop - Raptor Lake-S Refresh (codenamed "Raptor Lake") (14th Gen) [edit]

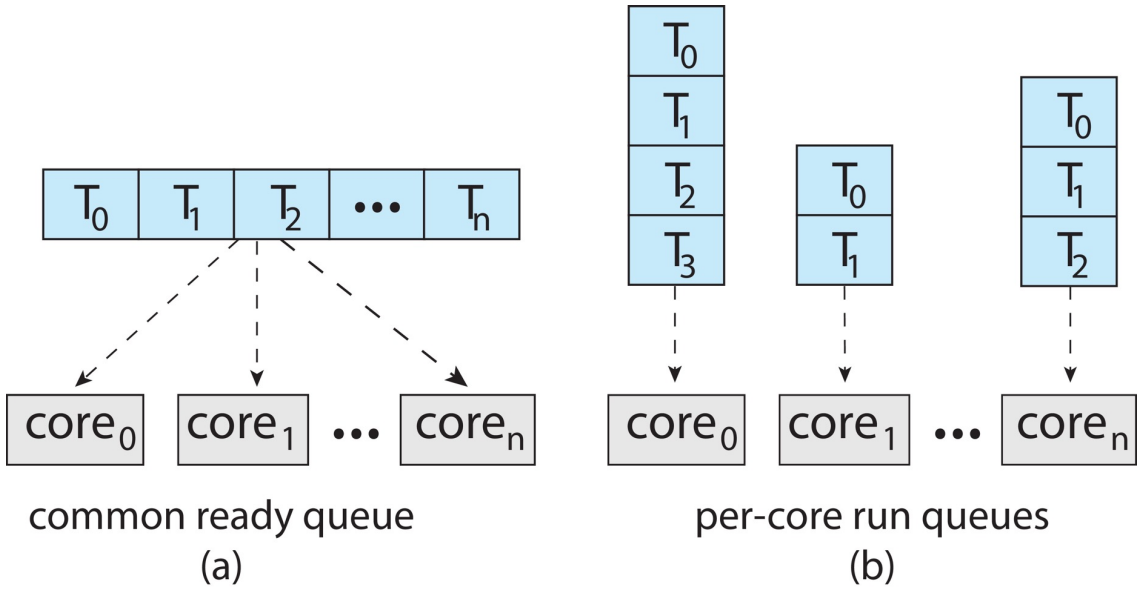
An iterative refresh of Raptor Lake-S desktop processors, called the 14th generation of Intel Core, was launched on October 17, 2023.[1][2]

Processor branding	Model	Cores (threads)		Clock rate (GHz)						GPU		Smart cache	TDP		Price (USD)[a]
				Base		Turbo Boost				Model	Max. freq. (GHz)				
						2.0		3.0	TVB						
		P	E	P	E	P	E	P	P				Base	Turbo	
Core i9	**14900K**	8 (16)	16 (16)	3.2	2.4	5.6	4.4	5.8	6.0	UHD 770	1.65	36 MB	125 W	253 W	$589
	14900KF									N/A					$564
	14900			2.0	1.5	5.4	4.3	5.6	5.8	UHD 770	1.65		65 W	219 W	$549
	14900F									N/A					$524
	14900T			1.1	0.8	5.1	4.0	5.5		UHD 770	1.65		35 W	106 W	$549
Core i7	14790F		8 (8)	2.1	1.5	5.3	4.2	5.4		N/A			65 W	219 W	China only
	14700K		12 (12)	3.4	2.5	5.5	4.3	5.6		UHD 770	1.6	33 MB	125 W	253 W	$409
	14700KF									N/A					$384
	14700			2.1	1.5	5.3	4.2	5.4		UHD 770	1.6		65 W	219 W	
	14700F									N/A					$359
	14700T			1.3	0.9	5.0	3.7	5.2		UHD	1.6		35 W	106 W	$384



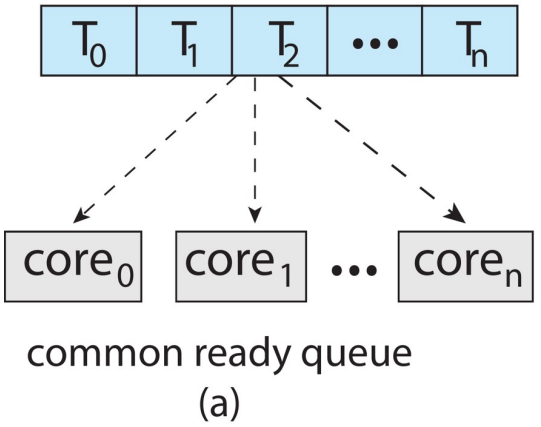
Multiple-Processor Scheduling

- 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)



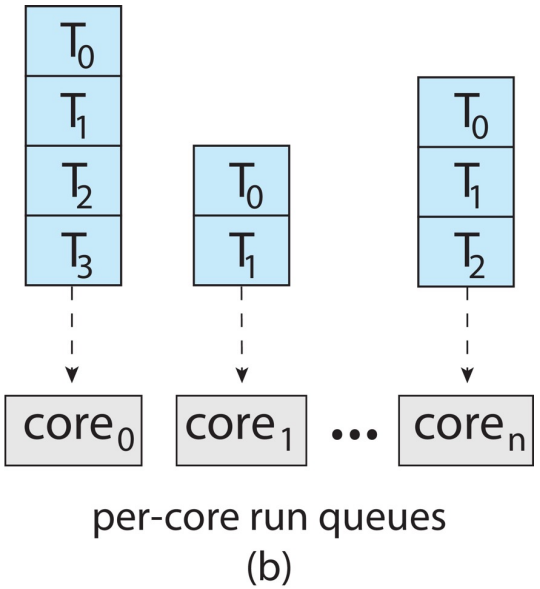
Multiple-Processor Scheduling

- แบบ (a) ต้องมีการป้องกัน race condition เวลา update data structure ที่ใช้เก็บข้อมูลของ Task โดยใช้การ lock หรือ mutual exclusion
- ถ้า core หลาย core หยิบ T ได้พร้อมกัน มันอาจหยิบ T เดียวกันพร้อมกันและเกิดความผิดพลาด (ถ้า T เป็นของมีค่าก็เหมือนกับ T ขาดหรือเสียหาย) = เปรียบเหมือน race condition
- ดังนั้น แต่ละ core จะต้องเปิดประตูเข้าไปเอา T มาประมวลผลได้ทีละ Core เท่านั้น
- ประตูเปรียบเหมือน locking mechanism หรือ mutual exclusion mechanism
- เสียเวลาจัดการ mutual exclusion
- Linux Scheduler โบราณใช้วิธีนี้



Multiple-Processor Scheduling

- แบบ (b) ไม่ต้องกังวลเรื่อง race condition
- แต่ private queue ของแต่ละ core อาจ ไม่สมดุล
- อาจมีการทำ load balancing ด้วยการ migrate task ข้าม core
- Linux Scheduler ปัจจุบันใช้วิธีนี้



per-core run queues
(b)

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
- **Pull migration** – idle processors pulls waiting task from busy processor
- ใน SMP Processors (ใช้เวลาเข้าถึง memory เท่ากัน (UMA)) OS จะพยายามกระจายงานให้ทุก Processor รองรับการประมวลผลของ Processes จำนวนเท่าๆกัน
- แต่เราไม่รู้ว่าแต่ละ Process จะใช้เวลาประมวลผลเท่าไร ดังนั้นเมื่อเวลาผ่านไป บาง Processor จึงอาจมี Process ที่มันต้องประมวลผลมากกว่า Processor อื่น
- ด้วยเหตุนี้ OS จึงอาจจำเป็นต้องทำ Load Balancing โดยย้าย Process จาก Processor หนึ่งข้ามไปอีก Processor หนึ่ง

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
- **Pull migration** – idle processors pulls waiting task from busy processor
- Linux มีวิธีการทำ Load balancing สองแบบ ได้แก่
- Push migration: task ที่เช็ค load ของทุก processor เป็นระยะ (periodically) จะเช็คว่ามี CPU core ไหนที่ Overload และจะแบ่งงานจาก queue ของ CPU นั้นไปให้ CPU อื่น
- Pull migration: CPU ที่ว่างไปดึงงานจาก CPU อื่นที่ busy มาประมวลผล
- Balance (นิยามขึ้นอยู่กับ OS) โดยทั่วไปแล้ว หมายถึงทุก priority queue มีจำนวน Tasks เท่าๆกันและ Tasks มี priority หลากหลายเหมือนๆกัน

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.

Multiple-Processor Scheduling – Processor Affinity

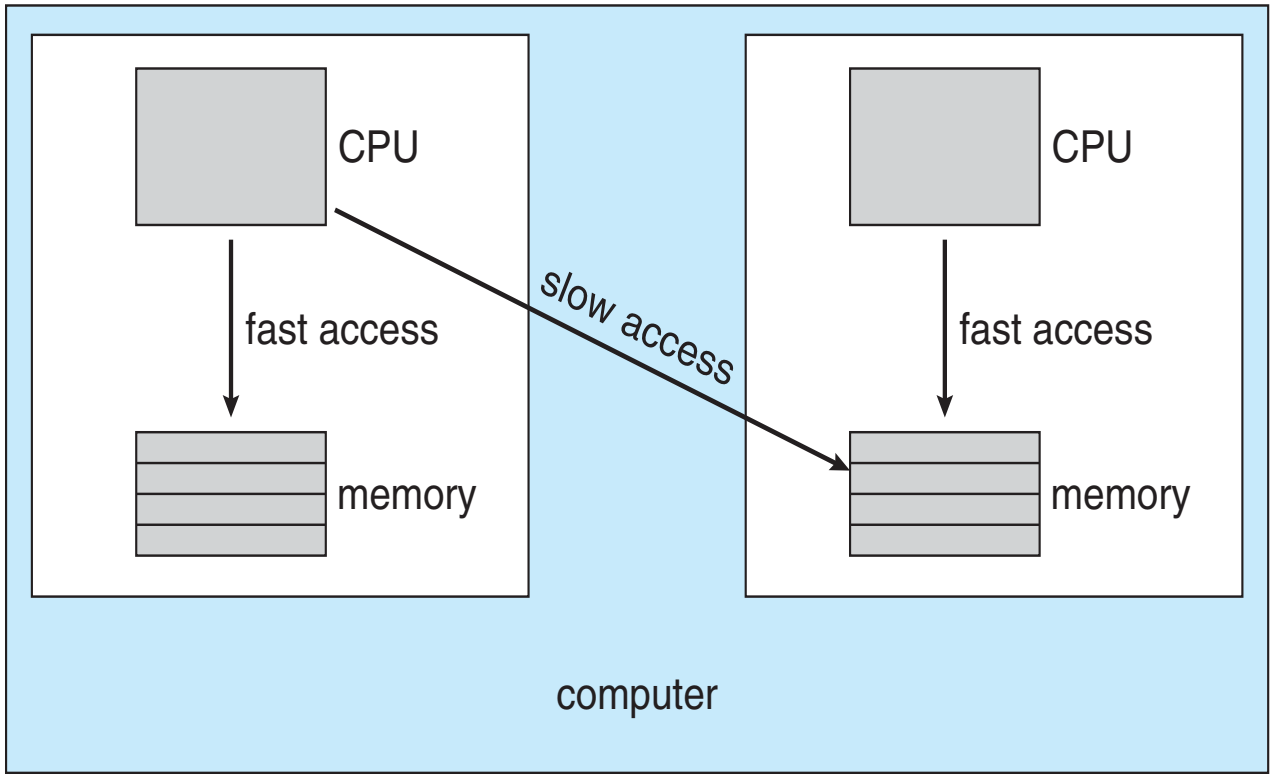
- เมื่อ OS กำหนดให้มีการรัน Thread (ใน Process) บน processors ใดก็ตาม ชุดคำสั่งหรือข้อมูลที่ thread นั้นใช้บ่อยจะถูกเก็บใน Cache ของ processor นั้น
- เราเรียกสถานการณ์นี้ว่า Thread นั้นมี Affinity กับ Processor นั้น (i.e., "processor affinity")
- การย้าย thread ไปรันบน Processor อื่นเพื่อ load balancing อาจทำให้ thread นั้นมีประสิทธิภาพลดลงเนื่องจากต้องสละชุดคำสั่งและข้อมูลใน cache
- Soft affinity คือการที่ OS พยายามจะ schedule thread ให้รันบน Processor เดียวกัน (หรือรันบนกลุ่มของ Processor ที่ share cache ร่วมกัน) แต่ไม่การันตี
- Hard Affinity คือการที่ผู้ใช้กำหนดให้ Thread/Process รันบน set ของ Processor กลุ่มใดกลุ่มหนึ่งเท่านั้น (Thread จะรันบน Processor ใน set เท่านั้น)

NUMA Scheduling

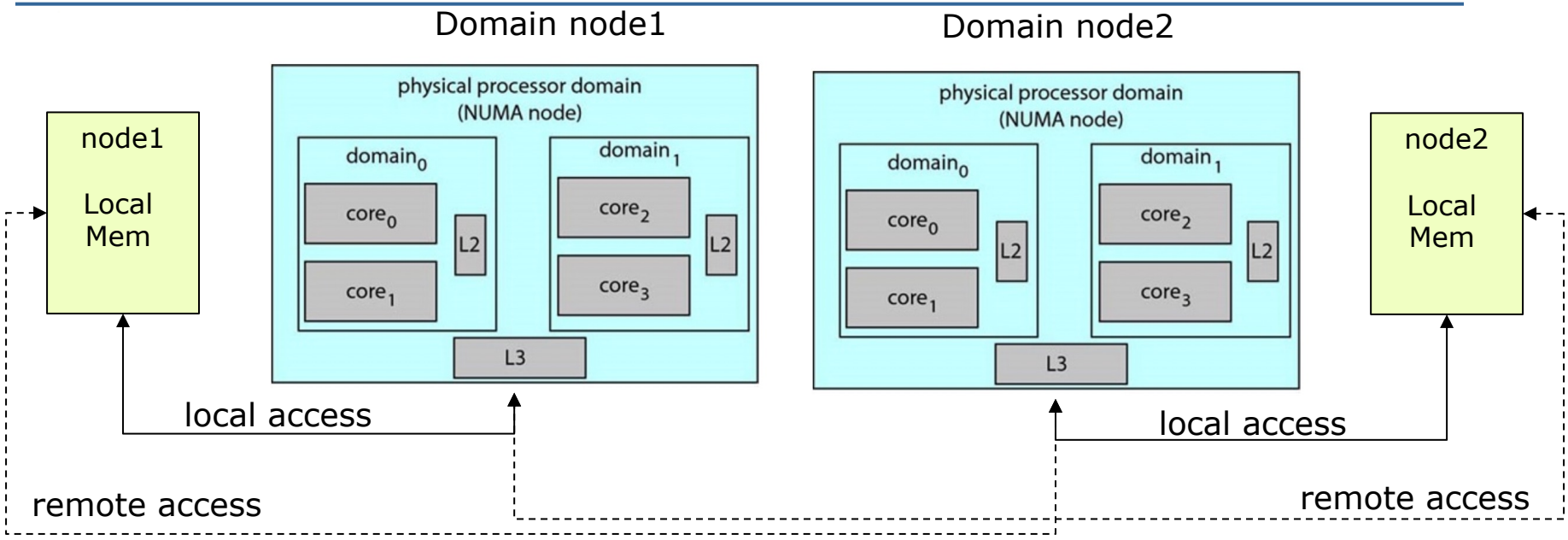
- คอมพิวเตอร์แบบ NUMA ประกอบไปด้วยหลาย CPU node แต่ละ node มี Local memory ของตนเอง
- Process สามารถใช้งาน local และ remote memory ได้แต่ความเร็วในการเข้าถึงข้อมูลจะต่างกัน
- ในระบบคอมพิวเตอร์แบบ NUMA OS จะพยายาม กำหนดให้ Thread ของ Process หนึ่ง รันอยู่บน CPU เดียวกัน
- OS จะพยายาม load ข้อมูลและชุดคำสั่งของ Process เข้าสู่ Memory ของ CPU ที่มัน Schedule ให้ Process นั้นรัน

NUMA and CPU Scheduling

If the operating system is **NUMA-aware**, it will assign memory closes to the CPU the thread is running on.



NUMA Load balancing by Linux CFS



- Linux จัดกลุ่ม cores ที่ share cache ให้อยู่ใน domain เดียวกัน
- Linux CFS จะย้าย Process ระหว่าง Cores ใน Domain เดียวกันก่อน (share L2)
- กำหนด physical processor domain สำหรับแต่ละ NUMA CPU node (mode 1 และ node 2)
- https://www.kernel.org/doc/html/v5.4/scheduler/sched-domains.html

NUMA Load balancing by Linux CFS

- ในกรณีที่ NUMA Processor หนึ่งมี Task มากกว่า Processors อื่นมาก Linux อาจ อพยพ Migrate Process จาก Processor ที่ load มากไปยัง processor ที่มี load น้อย
- เบื้องต้นมันจะย้ายจาก Process ระหว่าง Processor cores ที่อยู่ใน Domain เดียวกัน เช่นภายใน Domain 0 ที่แชร์ L2 cache ร่วมกัน
- OS มันจะไม่ย้าย Process ข้าม Domain (เช่นจาก Domain 0 ไปยัง Domain 1) เพราะจะทำให้ประสิทธิภาพในการเข้าถึงหน่วยความจำลดลง เนื่องจากต้องสละข้อมูลใน L2 cache
- **แต่ถ้าจำเป็น (**เช่น **เมื่อเกิดความ** Imbalance ระหว่าง NUMA node อย่างมาก) **OS อาจ**ย้ายข้าม Domain 0/1 ได้ (แต่ยังคง share L3)

NUMA Load balancing by Linux CFS

- **ในกรณีที่ OS จำเป็นต้องย้าย Process ข้าม NUMA node (เช่นในกรณีที่เกิด Load imbalance อย่างมาก หรือ NUMA node เกิด Processor Partial Errors/hw fault)** ค่าหน่วยความจำของ process จะไม่คงที่เนื่องจาก memory เก่าและใหม่จะอยู่ต่าง node กัน
- OS อาจใช้ **Page migration** เพื่อย้าย memory ตาม process ที่ถูกย้ายข้าม NUMA Node เพื่อให้ mem ทั้งหมดของ process อยู่ใน Local mem ของ node เดียวกัน https://docs.kernel.org/mm/page_migration.html

สรุป

- Scheduling Algorithms
- FCFS, SJF, RR
- Exponential Averaging
- Priority Scheduling, Priority RR
- Multilevel Queue
- Multilevel Feedback Queue
- Multi-processor scheduling
- Loadbalancing, Processor Affinity
- NUMA scheduling