政大資科系

作業系統

Operating System

廖峻鋒

cfliao@nccu.edu.tw

Operating System

CPU Scheduling

Chun-Feng Liao

廖峻鋒

Department of Computer Science

National Chengchi University

提要

- Basic concepts
  - Terms: burst, bound, scheduler, dispatcher, preemptive
- CPU Scheduling
  - **排程的根據**: CPU utilization, throughput, …
  - **排程機制**: FCFS, SJF, …
  - Multi-processor scheduling
    - Hardware thread (hyper-threading)
    - Affinity

CPU Scheduling

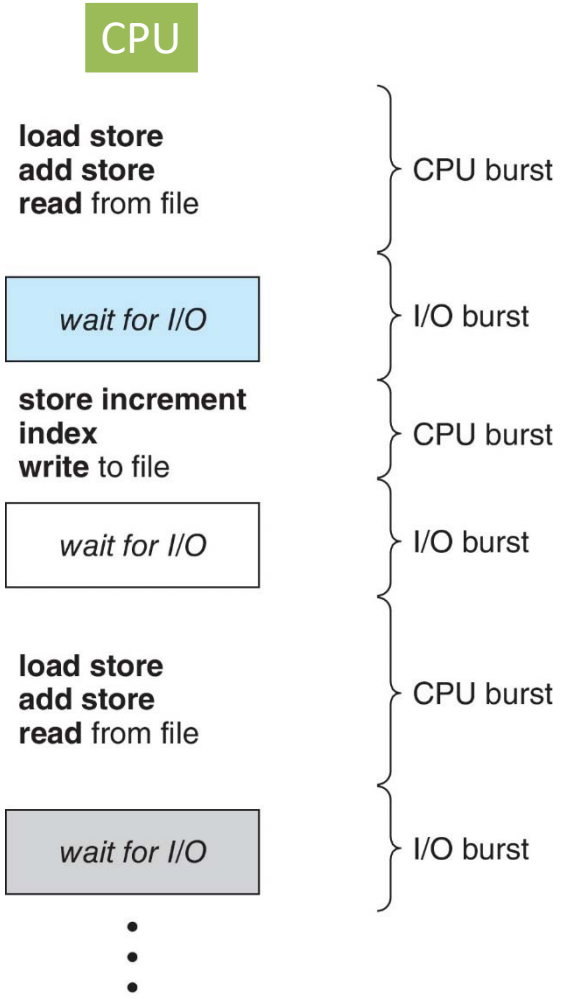
- 排程的功能
  - 無法降低個別工作本來就要花的時間
  - 降低「浪費」的時間
- 當代作業系統通常就kernel-level threads (而非Process)進行排程
  - The terms process and thread scheduling are often used interchangeably
  - In the textbook, process scheduling refers a general concept; thread scheduling refers to thread-specific ideas

為何要排程

- Multiprogramming: CPU多個工作一起處理
  - Keep several processes in memory at the same time
  - Theoretically, CPU can run one process at a time
    - Wait for I/O is a waste of time
  - Improve CPU utilization
    - Every time one process (thread) has to wait (for I/O), another process (thread) takes over the use of the CPU

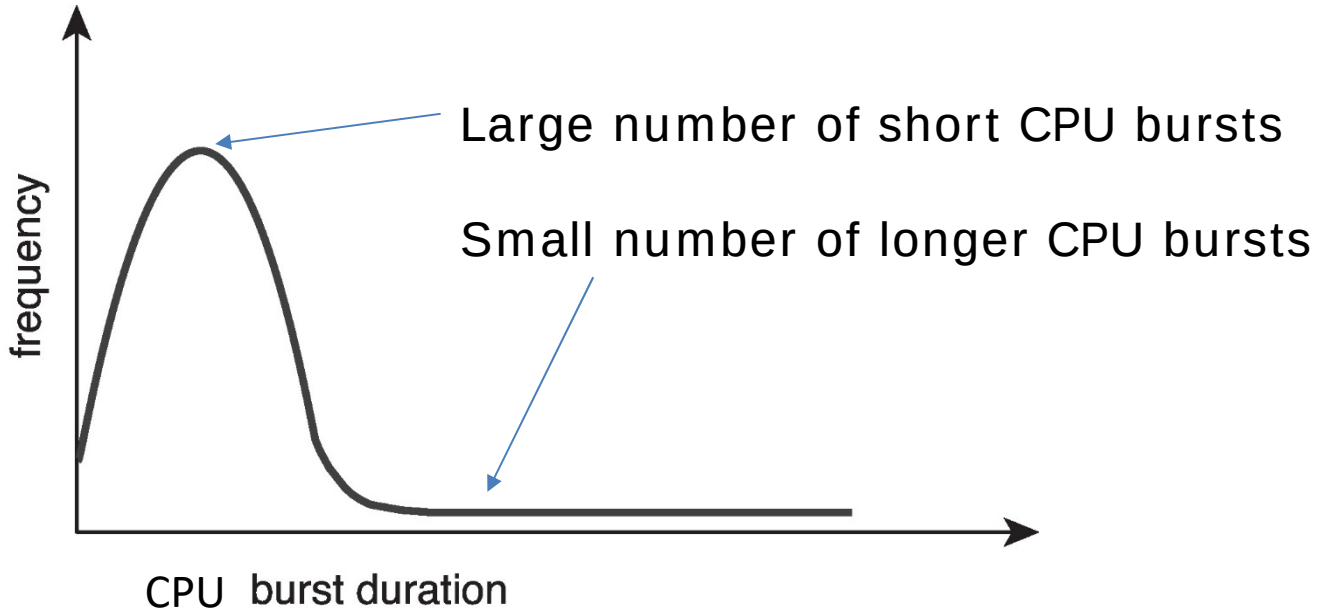
CPU – I/O Burst Cycles

- Process execution
  - CPU burst / I/O burst
- 程式特性
  - I/O-bound
    - has many very short CPU bursts
    - I/O多: 大部份程式是這種類型
  - CPU-bound
    - has a few long CPU bursts
    - 計算多: 少部份程式是這種類型



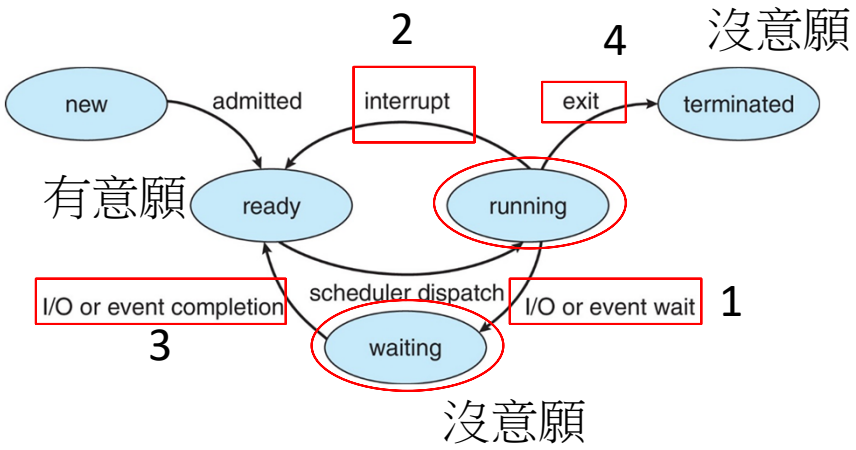
Histogram of CPU-burst Times

(Histogram: 次數累計圖)



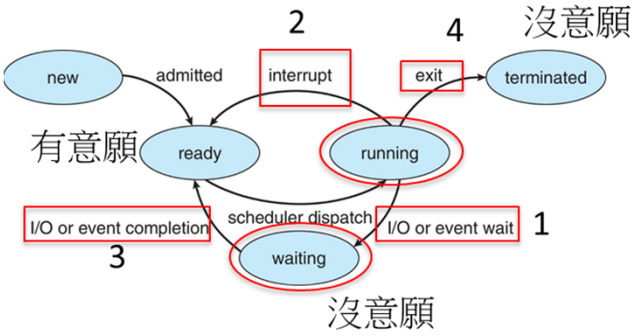
Preemptive vs. Non-preemptive

- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates



Preemptive vs. Non-preemptive

- Non-preemptive scheduling:
  - Scheduling only under 1 (running→waiting) and 4 (terminated)
  - 沒意願才換人
- Preemptive scheduling:
  - Scheduling under all cases (1-4)
  - Virtually all modern OSs use preemptive scheduling algorithms
  - May result in race conditions
    - Preempt→running state的processes可能被拉出來→造成二個processes指令交錯→race condition

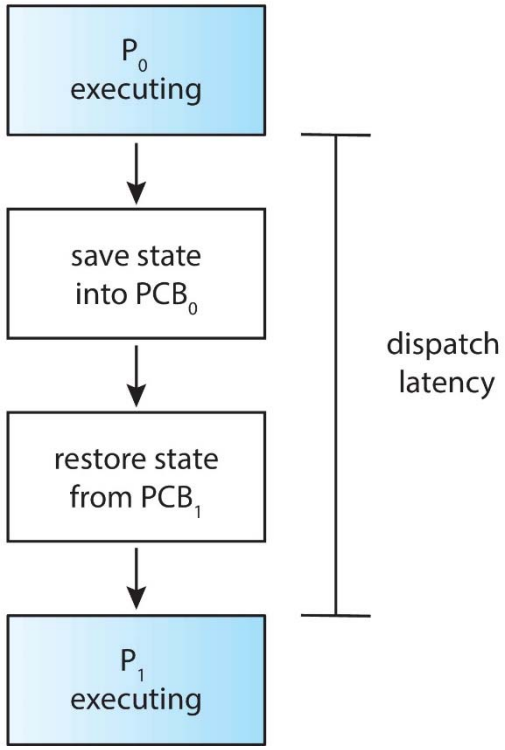


Preemptive Issues

- Inconsistent state of shared data
  - Require synchronization (Chap6-7)
- Affect the design of the OS kernel
  - 被preempted後造成資料混亂
    - Ex: user process在(kernel mode)修改重要系統結構(如I/O queue)到一半時，被kernel preempted，而kernel也對該結構進行修改
    - Modern kernel uses synchronization tools introduced in Ch.6-7

Dispatcher

- Dispatcher module 將CPU控制權交給Scheduler選定的Process/Thread
  - Switching the context
  - Switching to user mode (if is user code)
  - Jumping to the proper location in the selected program (依據PC)
- Dispatch latency
  - time for the dispatcher to stop one process and start another running



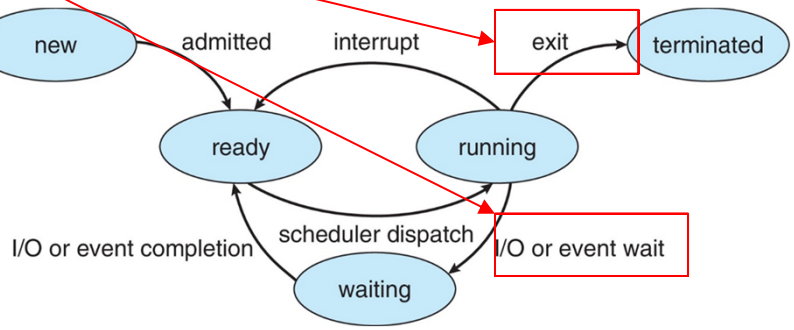
Demo: Context Switch

- vmstat 1 3
  - 3 lines of sampling output over a 1-second delay

```
root@try-VirtualBox:~# 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
 3  0      0 546912 117748 2738960    0    0  4151  7362 1238 6228 57 14  4 26  0
 1  0      0 516944 117900 2762552    0    0   812  1852  593 7876 56 44  0  0  0
 1  0      0 468888 118220 2810084    0    0     0     0  502 6774 30 70  0  0  0
```

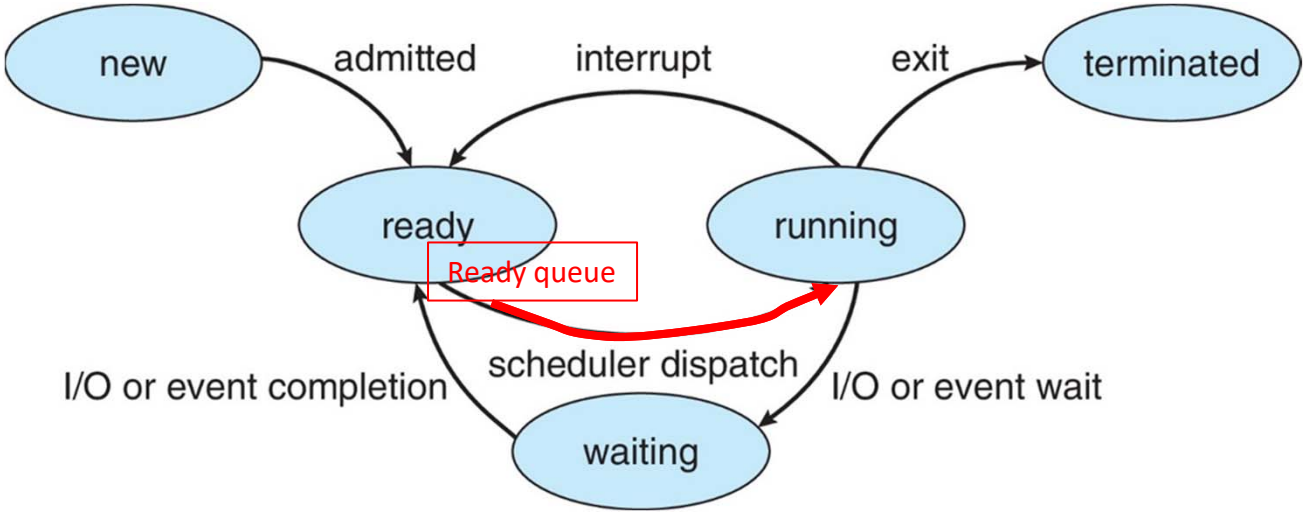
- cat /proc/{pid}/status

```
voluntary_ctxt_switches:        2
nonvoluntary_ctxt_switches:     0
```



CPU Scheduler

- 主要任務
  - Selects from ready queue to execute (i.e. allocates a CPU for the selected process)



Scheduling 考量點

Process排程只會影響Process在queue等待時間 ，不會縮短個別Task所需執行時間

- CPU utilization
  - theoretically: 0%~100%
  - real systems: 40% (light)~90% (heavy)
- Throughput
  - number of completed processes per time unit
- Turnaround time 完成一件工作總時間
  - submission ~ completion
- Waiting time 在ready queue的時間完全是浪費←排程可以縮短
  - total waiting time in the ready queue
- Response time 第一次回應時間(工作不一定完成)
  - submission ~ the first response is produced

Algorithms

- First-Come, First-Served (FCFS) scheduling
- Shortest-Job-First (SJF) scheduling
- Priority scheduling
- Round-Robin scheduling
- Multilevel queue scheduling
- Multilevel feedback queue scheduling

First- Come, First-Served (FCFS) Scheduling

現在時間點ready queue內容

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Process in arriving order:
  - Non-preemptive
- The Gantt Chart of the schedule

- Waiting time: P1 = 0, P2 = 24, P3 = 27
- Average Waiting Time (AWT): (0+24+27) / 3 = 17

FCFS Scheduling

- Suppose that the processes arrive in the order:

  P2 , P3 , P1
- The Gantt chart for the schedule is:

- Waiting time: P1 = 6, P2 = 0, P3 = 3
- Average Waiting Time (AWT): (6+0+3) / 3 = 3
  - Much better than previous case

多個短process 在等待長process

- Convoy effect - short process behind long process
  - Consider: one CPU-bound → many I/O-bound processes

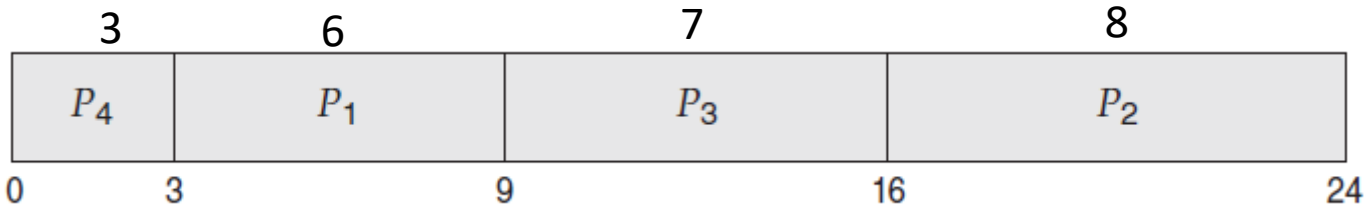
Shortest-Job-First (SJF) Scheduling

- 方法
  - Associate with each process the length of its next CPU burst
  - A process with shortest burst length gets the CPU first
  - SJF provides the minimum average waiting time (optimal!)
- Two schemes (排好後執行時)
  - Non-preemptive – once CPU given to a process, it cannot be preempted until its completion
  - Preemptive – if a new process arrives with shorter burst length, preemption happens (新來的若短就先超車)

通常不可能達成
因為不見得總是能估計下一個CPU burst length
SJF通常用來當baseline

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3



Average waiting time = (3 + 16 + 9 + 0) / 4 = 7ms
比較: FCFS AWT: 10.52ms

Approximate Shortest-Job-First (SJF)

- SJF difficulty
  - no way to know length of the next CPU burst
- Approximate SJF
  - The next burst can be predicted as an exponential average of the measured length of previous CPU bursts

直覺意義: 現在和過去都以一定的比例(alpha，1-alpha)影響未來

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

history 根據過去歷史的猜測值

new one 最近CPU burst執行時間

Commonly,

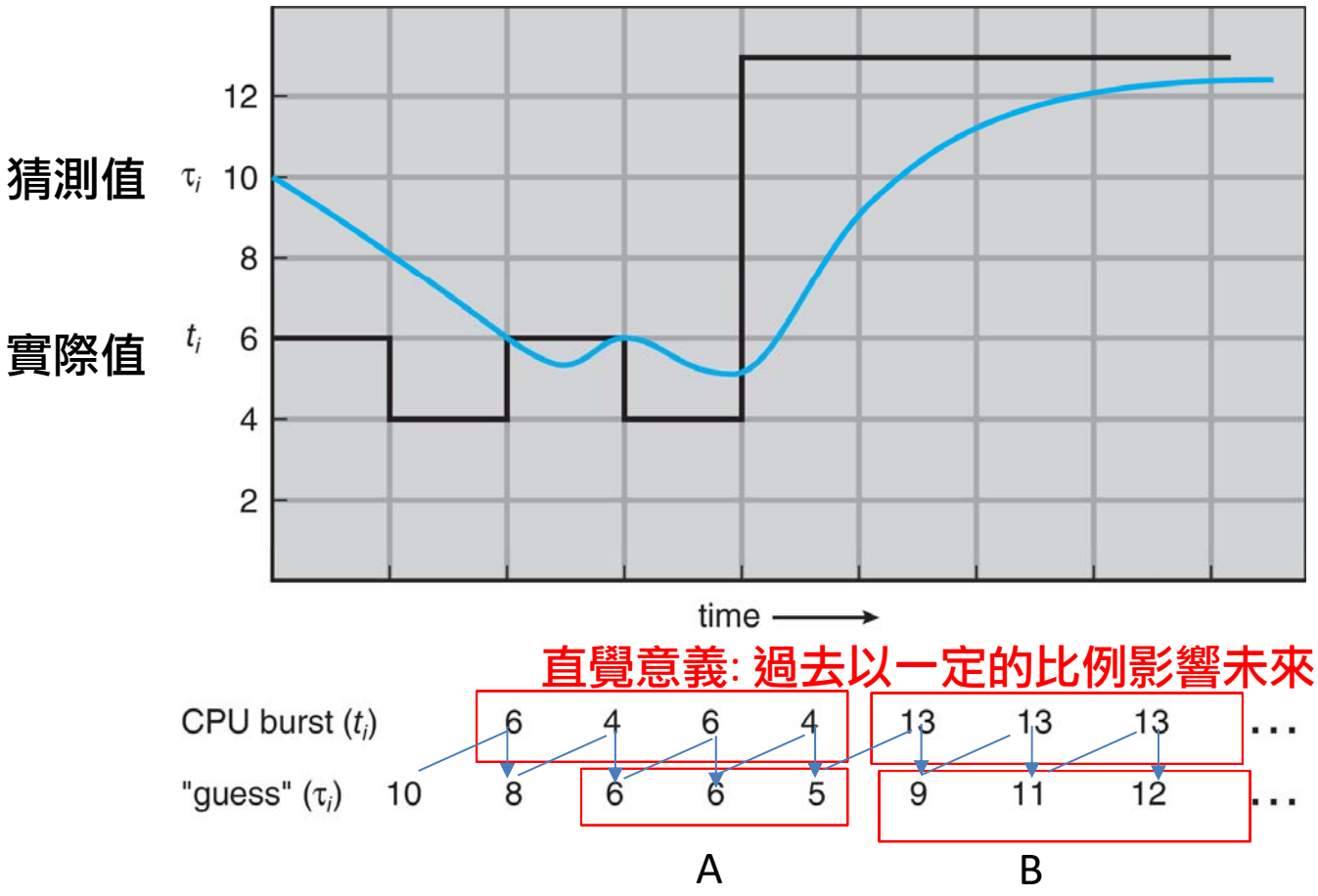
$$= \alpha t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \ldots$$

$\alpha = 1/2$ →

$$= \left(\frac{1}{2}\right) t_n + \left(\frac{1}{2}\right)^2 t_{n-1} + \left(\frac{1}{2}\right)^3 t_{n-2} + \ldots$$

$t_{n-1}, t_{n-2}, \cdots$ →過去的CPU burst time(歷史)

Prediction of the Length of the Next CPU Burst



Shortest-remaining-time-first

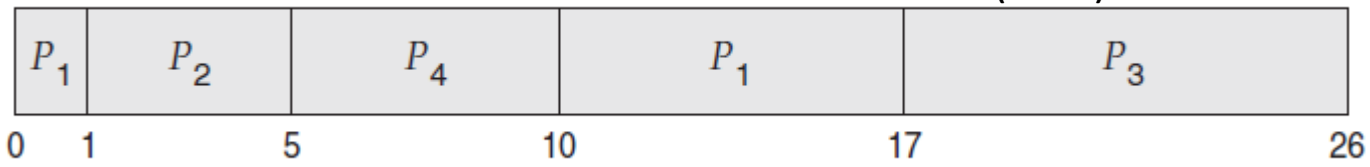
- Now we add the concepts of varying arrival times and preemption to the analysis

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

選剩下時間最短的

Time 0 (P1:8)
Time 1 (P1:7, P2:4) — Preempt!
Time 2 (P1:7, P2:3, P3:9)
Time 3 (P1:7, P2:2, P3:9, P4:5)
Time 5 (P1:7, P3:9, P4:5)
Time 10 (P1:7 , P3:9)
Time 17 (P3:9)

- Preemptive SJF Gantt Chart



P1 P2 P3 P4

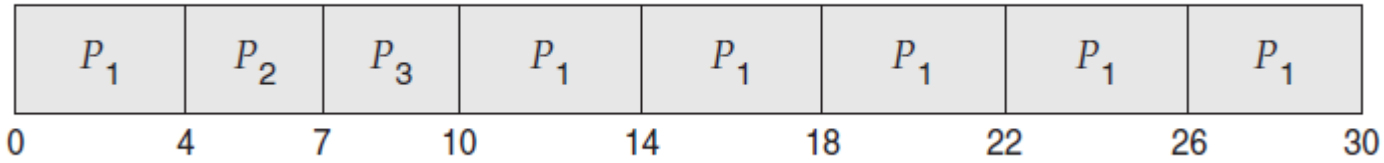
- AWT = [(10-1)+(1-1)+(17-2)+(5-3)]/4 = 26/4 = 6.5 ms

Round-Robin (RR) Scheduling

- Each process gets a small unit of CPU time
  - Time quantum, TQ
  - TQ usually 10~100 ms
- After TQ elapsed
  - Process is preempted
  - Added to the end of the ready queue
- TQ affects performance
  - TQ too large → FIFO
  - TQ too small → (context switch) overhead increases

Example of RR with Time Quantum = 4

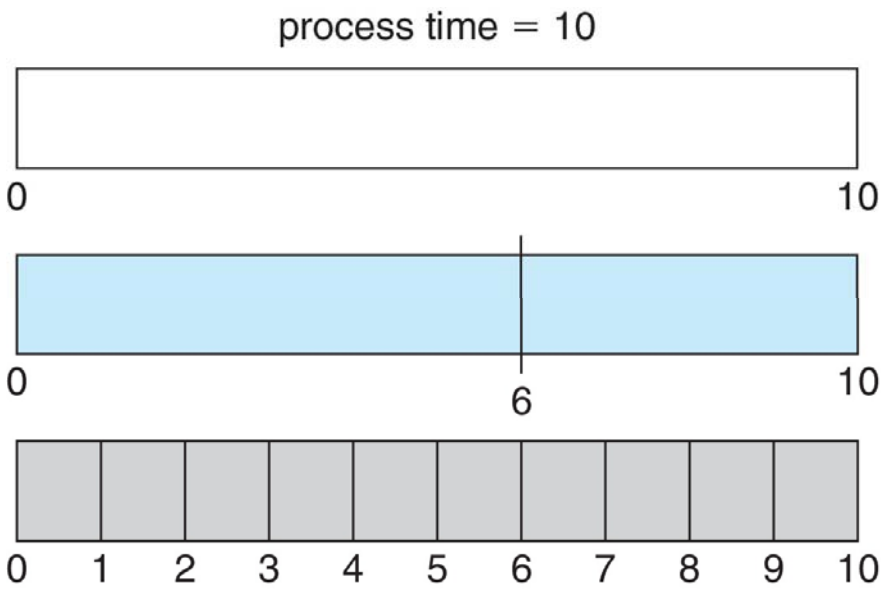
Process	Burst Time
P_1	24
P_2	3
P_3	3



- 效果: when compared to SJF
  - (Worse) Higher average turnaround time
  - Better response (適用於互動多的系統)
- TQ should be large compared to context switch time
  - TQ usually 10ms to 100ms, context switch < 10 mu-sec
  - TQ愈小代表ctx switch頻率愈高，ctx switch是pure overhead
  - 試想: TQ=ctx switch time時代表CPU耗損了1/2時間在switch (淨損失)

Turnaround time = submission ~ completion

Time Quantum and Context Switch Time

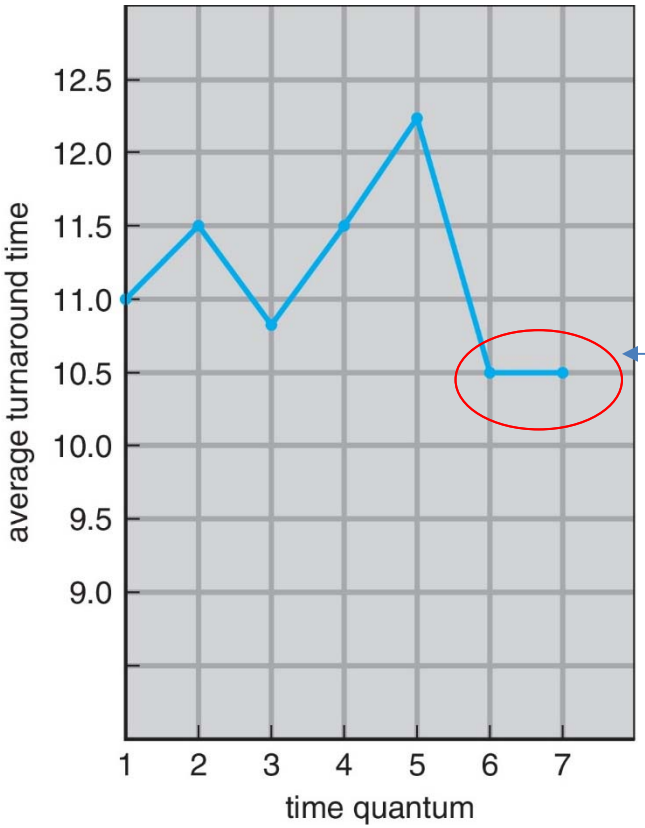


quantum	context switches
12	0
6	1
1	9

TQ愈小，ctx switch次數愈多，淨損耗相對愈大

Turnaround Time Varies With Time Quantum

P.211



process	time
P_1	6
P_2	3
P_3	1
P_4	7

如果大部份工作能在一次TQ內做完, ctx switch time 影響較少。

根據經驗, TQ應大於80% of CPU bursts (time)
換句話說只有20%的task會需要switch in/out

TQ也不能太大, 太大就變成FCFS了

Turnaround time: submission ~ completion

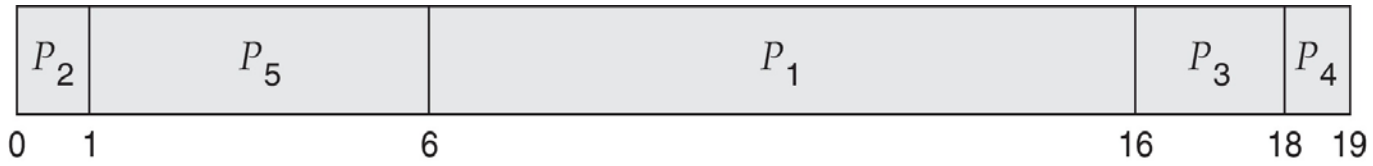
Priority Scheduling

- A priority number is associated with each process
  - SJF is a priority scheduling: priority is set based on next CPU burst time
- The CPU is allocated to the highest priority process (號碼小)
  - Preemptive: 立即切換
  - Non-preemptive: 放到ready queue
- Problem
  - Starvation – low priority processes may never execute
    - Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run
- Solution
  - Aging – as time progresses increase the priority of the process
  - Ex: increase priority by 1 every 15 minutes

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



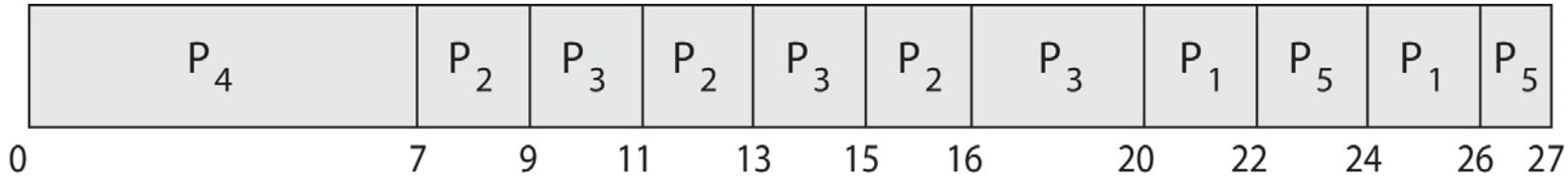
- Average waiting time = 8.2 msec

Priority Scheduling +RR

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

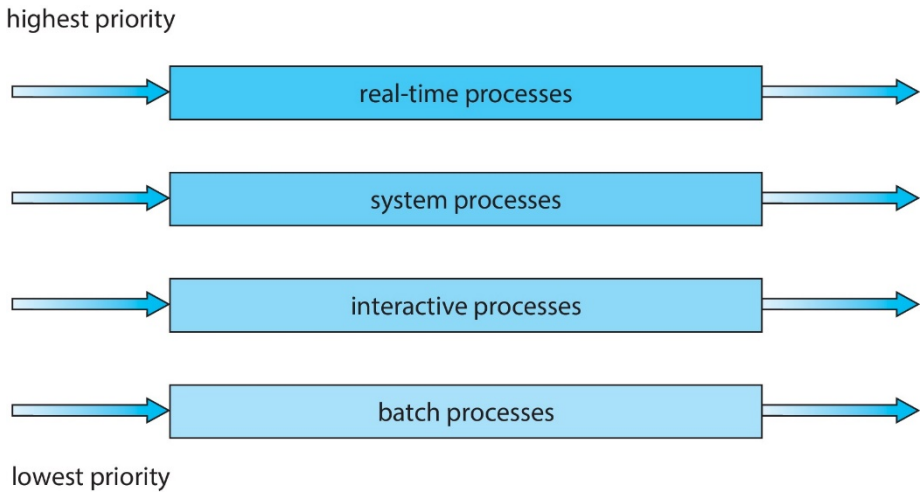
RR

- Run the process with the highest priority
  - Processes with the same priority run RR
- Gantt Chart with TQ=2



Multilevel Queue Scheduling

- Ready queue is partitioned into separate sub-queues
  - Each sub-queue has its own scheduling algorithm
- Scheduling must be done between queues Queue之間有先後次序
  - Fixed priority scheduling: possibility of starvation 固定優選權:一個queue的工作都做完才做next priorit
  - Time slice – each queue gets a certain amount of CPU time (e.g. 80%, 20%) 比例: 每個queue依一定比例分配工作時間



Multilevel Feedback Queue Scheduling

- A process can move between the various queues
  - The most general CPU scheduling algorithm
  - Aging can be implemented (等太久的process移到高優先)
- Idea
  - Separate processes according to their CPU bursts
  - High priority: I/O-bound and interactive processes
    - Short CPU burst
  - Lower priority: CPU-bound processes
    - Long CPU burst

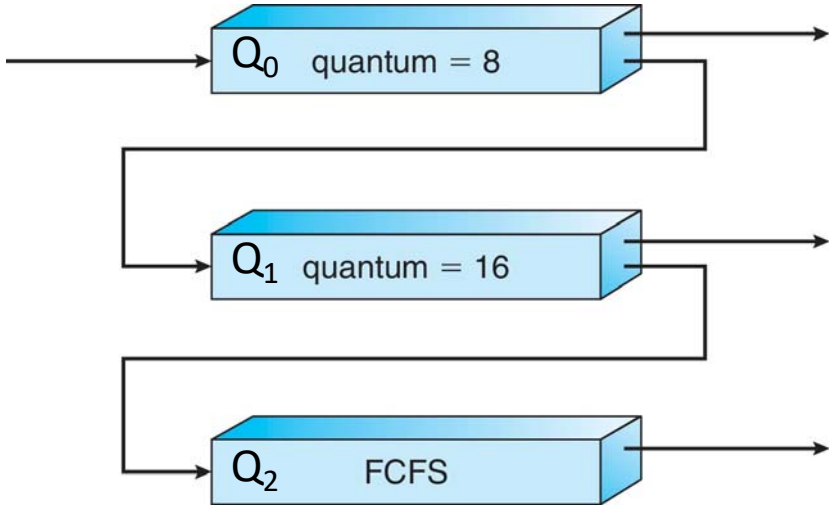
Multilevel Feedback Queue

- In general, multilevel feedback queue scheduler is defined by the following parameters:
  - Number of queues
  - Scheduling algorithm for each queue
  - Method used to determine when to upgrade a process
  - Method used to determine when to demote a process
    降級

- **課後閱讀**. P.236-P.238 Linux CFS/ red-black tree → vruntime

Example

- A new job enters Q_0.
  - Algorithm: RR + demote(8)
  - Demote if not finished in 8 ms
- At Q_1 is again served
  - RR + demote(16)
  - Demote if not finished in 16 ms
- Execution
  - Q_i only gets executed if Q_0 ~Q_{i-1} is empty
    - Ex: Q0>Q1>Q2
  - Starvation is possible if no aging policy



- Q0 – RR with time quantum 8 milliseconds
- Q1 – RR time quantum 16 milliseconds
- Q2 – FCFS

Multiple-Processor Scheduling

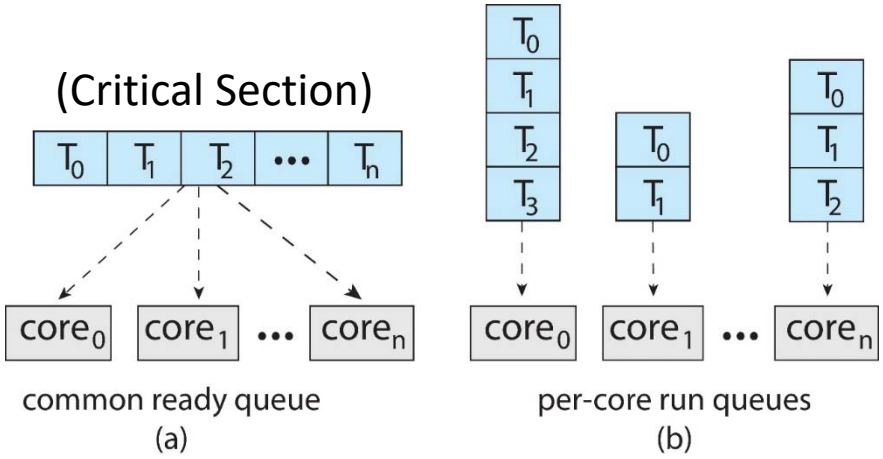
- Scheduling is more complex in multiprocessor systems
- Multiprocessor may be any one of the following architectures:
  - Multicore CPUs (分享同一記憶體)
  - NUMA systems (各自保有local記憶體)
  - Heterogeneous (異質) multiprocessing
    - 同時使用運算能力強弱不同的CPU
    - Ex. ARM big.LITTLE
      - big for short period tasks
      - LITTLE for long period tasks

Multi-Processor Scheduling

- Symmetric multiprocessing (SMP):
  - Each processor is self-scheduling
  - Share the same memory→Need synchronization mechanism
- Asymmetric multiprocessing:
  - Main processor (system)
    - Handles all system activities, can be a performance bottleneck
    - Only one processor access kernel data, reduce the need for data sharing
  - Other processor (application)
    - User code (allocated by the main processor)
  - Far simple than SMP

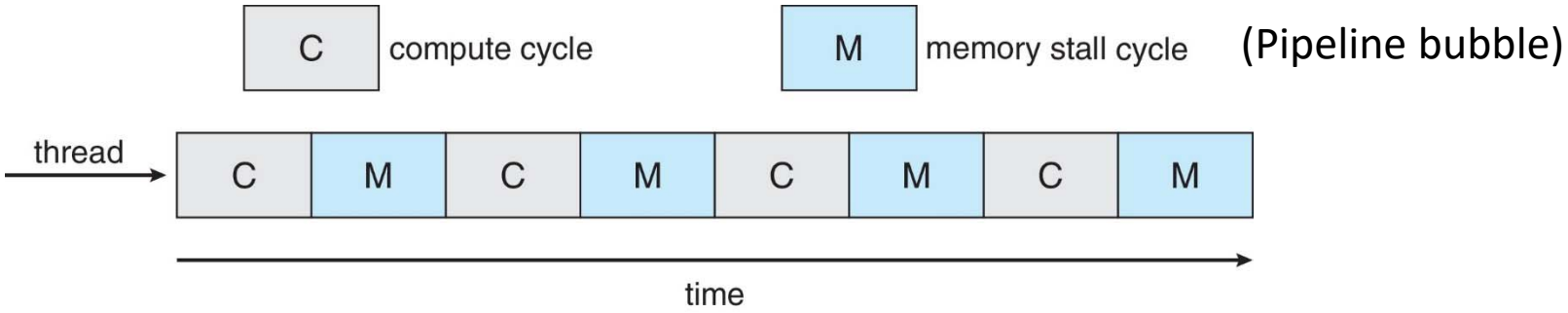
Multiple-Processor Scheduling

- SMP 架構選擇
  - All threads may be in a common ready queue (a)
    - Possibly race condition; must be made as a critical section
    - CPU可以容易takeover彼此工作
  - Each processor may have its own private queue of threads (b)
    - The queue is free from race condition; more common
    - 需要額外load balancing機制 (section 5.5.3)



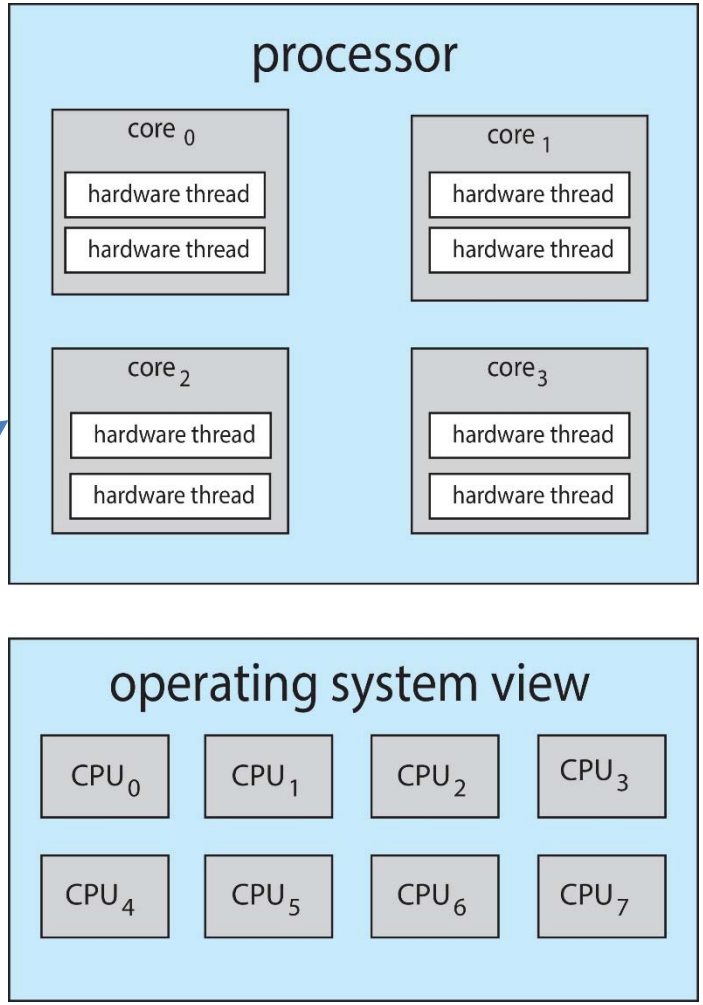
Multicore Processors

- Recent trend to place multiple processor cores on same physical chip (Multi-core)
  - Faster and consumes less power
  - Memory stall
    - When access memory, it spends a significant amount of time waiting for the data become available. (e.g. cache miss)
  - Solution: hardware threading (see next page)
    - 等待時還可以做另一件事



Hardware Threading

- **Chip-multithreading** (CMT)
  - Assigns each core multiple hardware threads
    - Intel **hyper-threading**

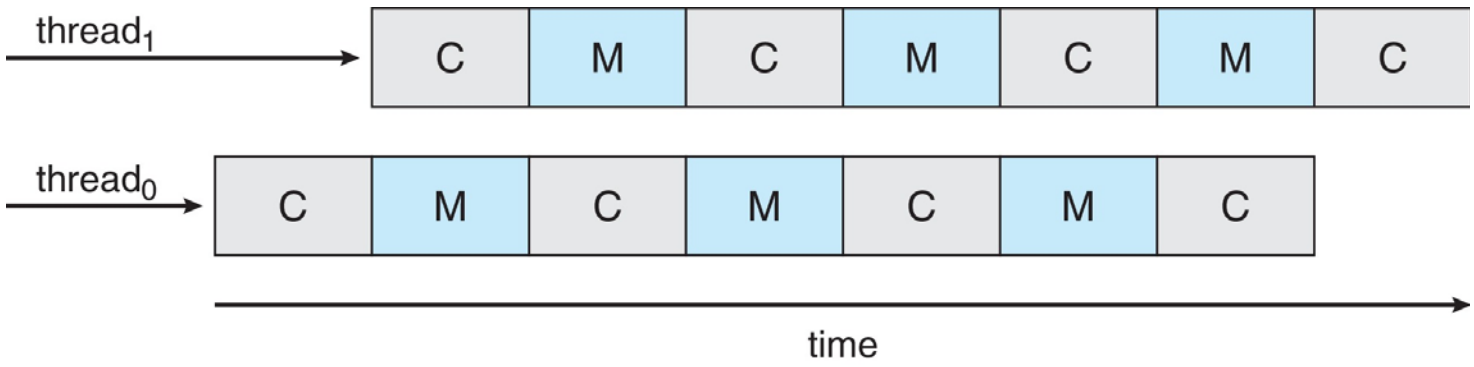


- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors

Multithreaded Multicore System

- Two (or more) hardware threads are assigned to a core (i.e. Intel Hyper-threading)
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens

C: compute; M: memory stall



為什麼一個core要有2 (or more threads)? 如果1個threads memory stall時，另一個就可以做事!

Multi-core Processor Scheduling

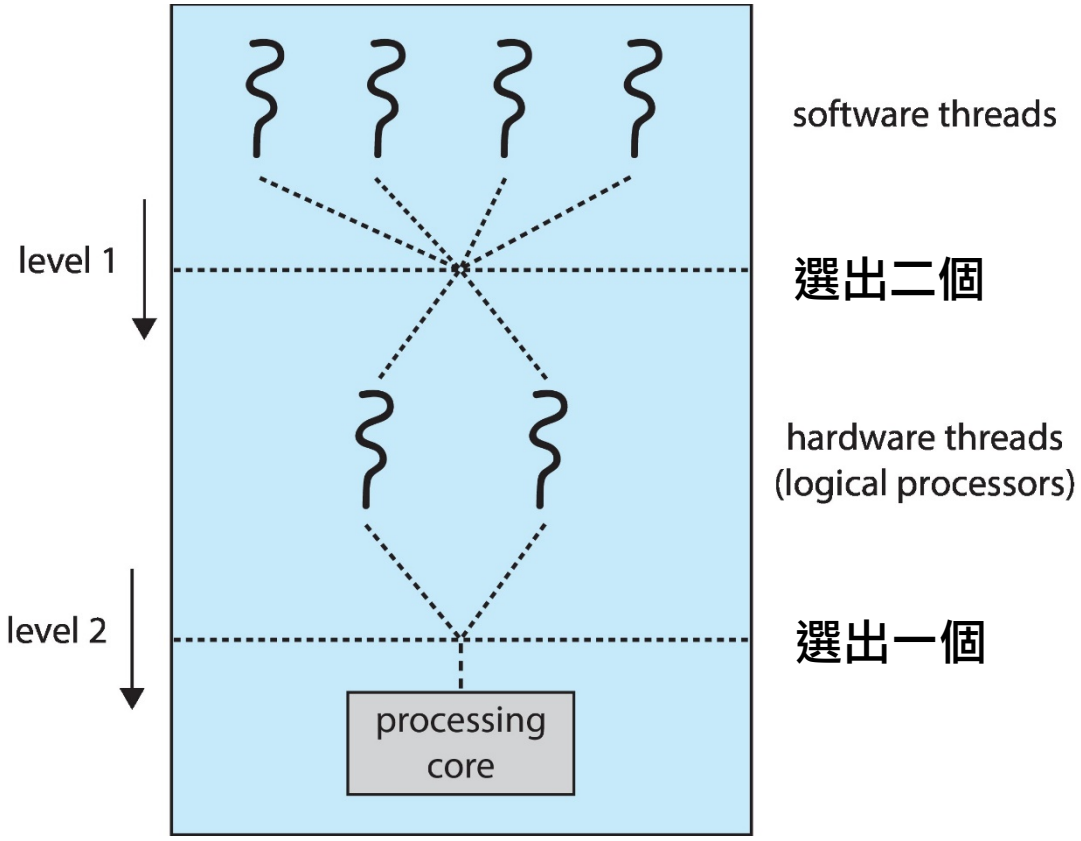
- Two ways to multithread a processor: Pipeline: 有warm-up time
  - Coarse-grained: switch to another thread when a memory stall occurs. The cost is high as the instruction pipeline must be flushed Why?
  - Fine-grained (interleaved): switch between threads at the boundary of an instruction cycle. The architecture design includes logic for thread switching – time cost is low. 實作複雜但效能高
- Scheduling for Multi-threaded multi-core systems
  - 1st level: Choose which software thread to run on each hardware thread (logical processor)
  - 2nd level: How each core decides which hardware thread to run

Multithreaded Multicore System

- Two levels of scheduling:

1. OS decides which (software) thread to run on a logical CPU

2. CPU decides which (hardware) thread to run on the physical core.



P.225, 5.5.4

Processor affinity

傾向、吸引

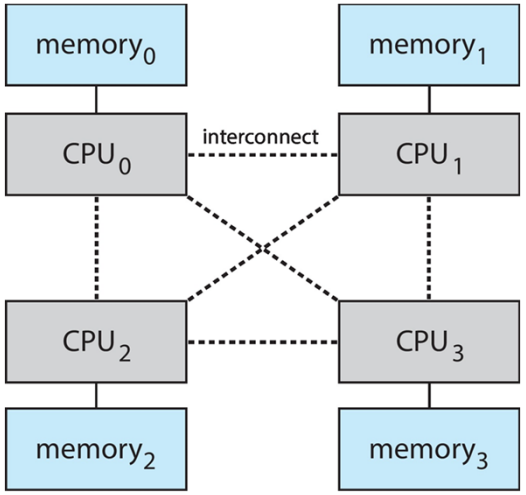
- Processor affinity: a process has an affinity for the processor on which it is currently running
  - A process populates its recent used data in cache memory of its running processor cache通常和CPU用一個chip (i.e. L1/L2 cache)
  - Cache invalidation and repopulation has high cost
- Solution
  - Soft affinity: (process) possible to migrate between processors
  - Hard affinity: (process) not to migrate to other processor

Load-balancing

- Keep the workload evenly distributed across all processors
  - Only necessary on systems where each processor has separate ready queue (若common ready queue就不需要)
- Two strategies:
  - Push migration: move (push) processes from overloaded to idle or less-busy processor (負擔重的將工作推出)
  - Pull migration: idle processor pulls a waiting task from a busy processor (負擔小的拉工作進來)
  - Often implemented in parallel
- Load balancing often counteracts the benefits of processor affinity
  - 因為工作不平均時，Process會被帶離開它原來run的地方
  - 二者的需求拉扯導致分散式排程非常複雜

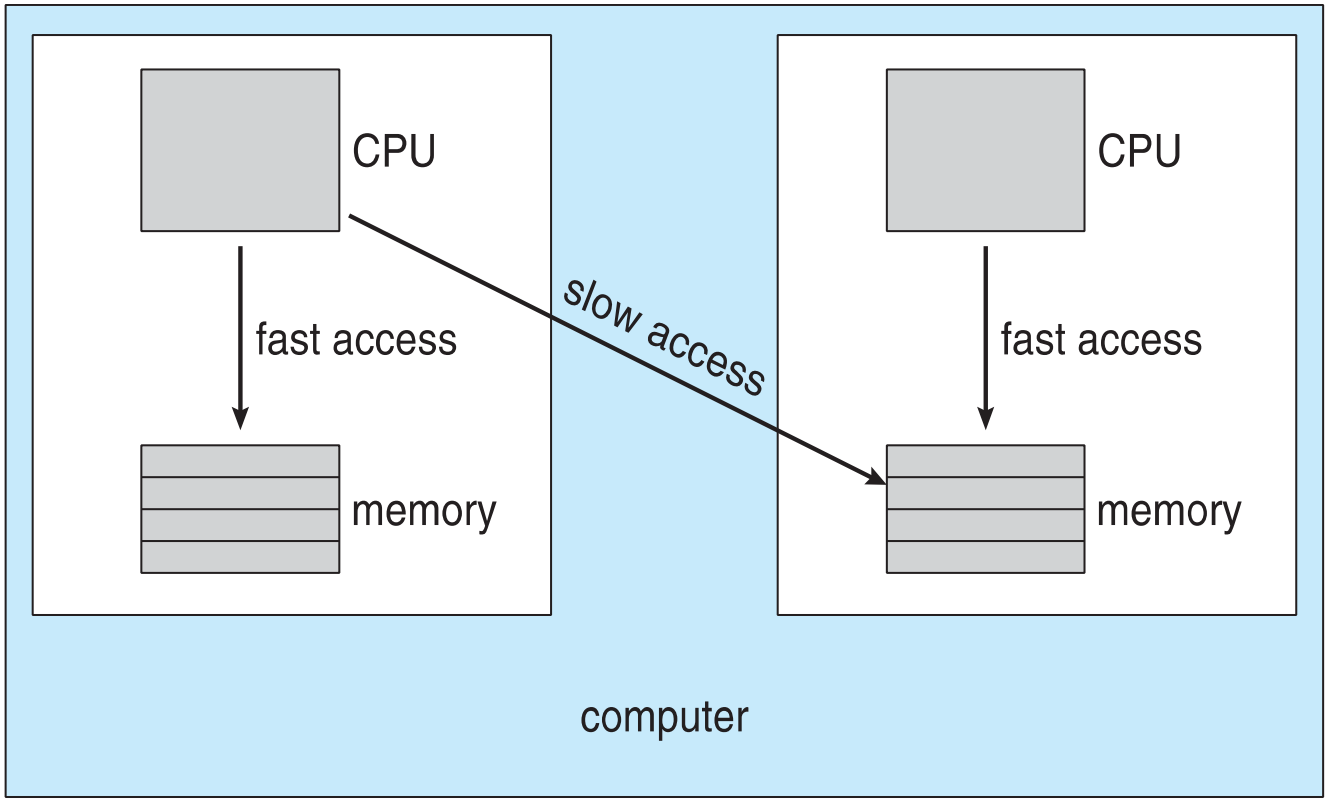
NUMA and CPU Scheduling

- NUMA (non-uniform memory access):
  - Occurs in systems containing combined CPU and memory boards
  - CPU scheduler and memory-placement works together
  - A process (assigned affinity to a CPU) can be allocated memory on the board where that CPU resides



NUMA and CPU Scheduling

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



Q & A