

政大資科系

作業系統

Operating System

廖峻鋒

cfliao@nccu.edu.tw

Operating System

Main Memory

Chun-Feng Liao

廖峻鋒

Department of Computer Science

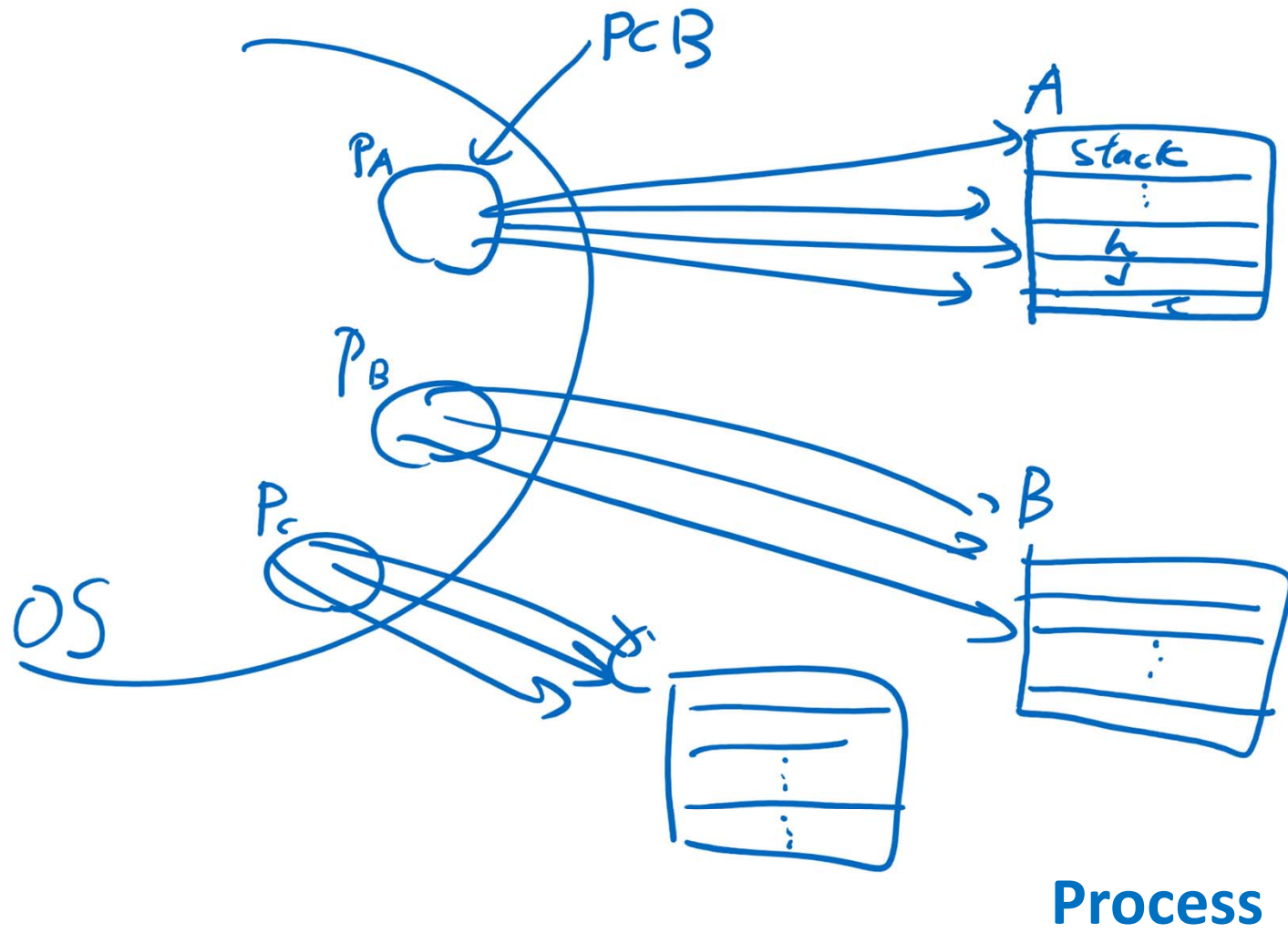
National Chengchi University

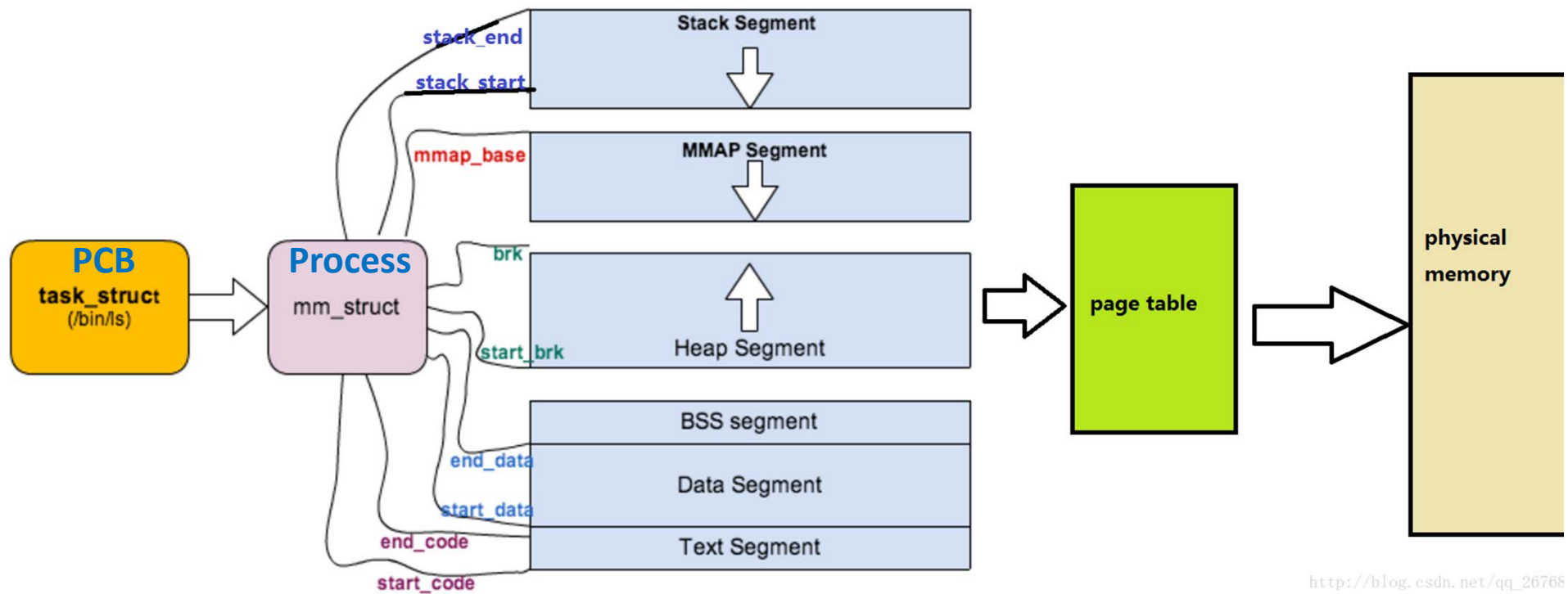
Outline

- Overview
- Memory Allocation
- Paging
- Page Table
- Swapping

Overview

- CPU可直接存取的儲存媒體
 - Registers
 - L1/L2 Cache
 - Main memory
- Speeds of memory hierarchy
 - Register access is done in one CPU clock (or less)
 - Main memory can take many cycles, causing a stall
 - Cache sits between main memory and CPU registers



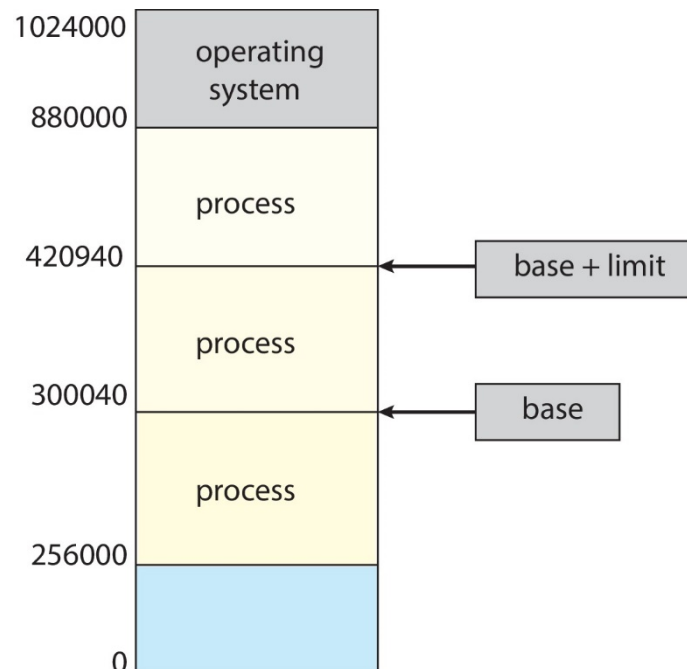


http://blog.csdn.net/qq_26768

https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h#L402

Memory Protection

- Protection is required to ensure correct operation
 - Using **base and limit** registers to define the logical address space of a process



$$300040 \leq \text{addr} < 300040 + 120900$$

$$\text{base} \leq \text{addr} < \text{base} + \text{limit}$$

Base: 限制了實體位址的存取下限

Base+Limit: 限制了實體位址的存取上限

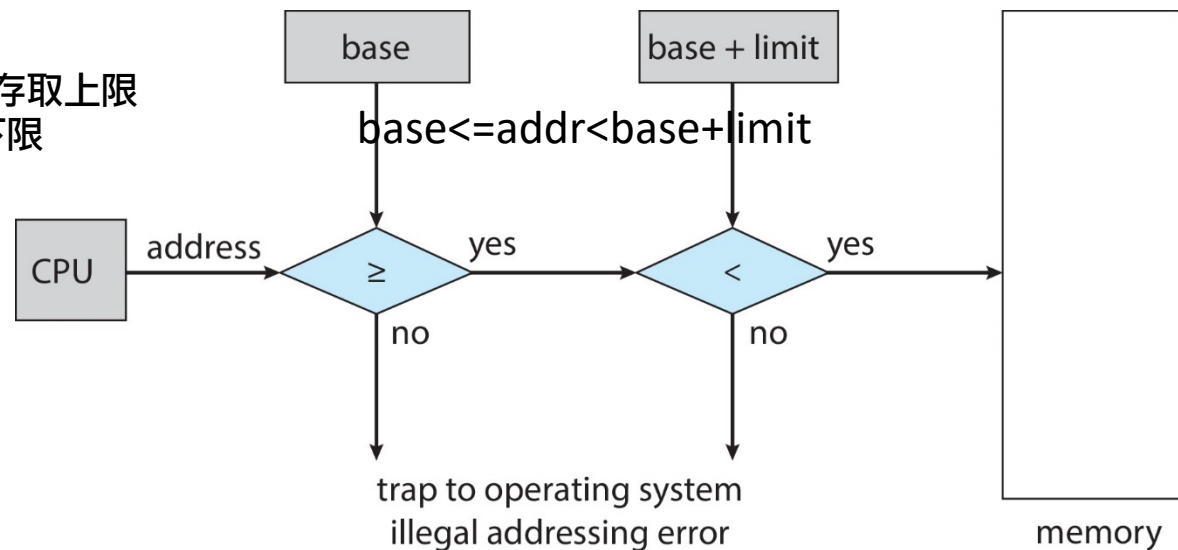
Memory Protection

- CPU must check every access generated in **user mode** to be sure it is between base and limit for that process
- Loading the base and limit registers are privileged instructions (i.e. can be modified only in kernel mode)

邊界檢查

Base+Limit: 限制了實體位址的存取上限

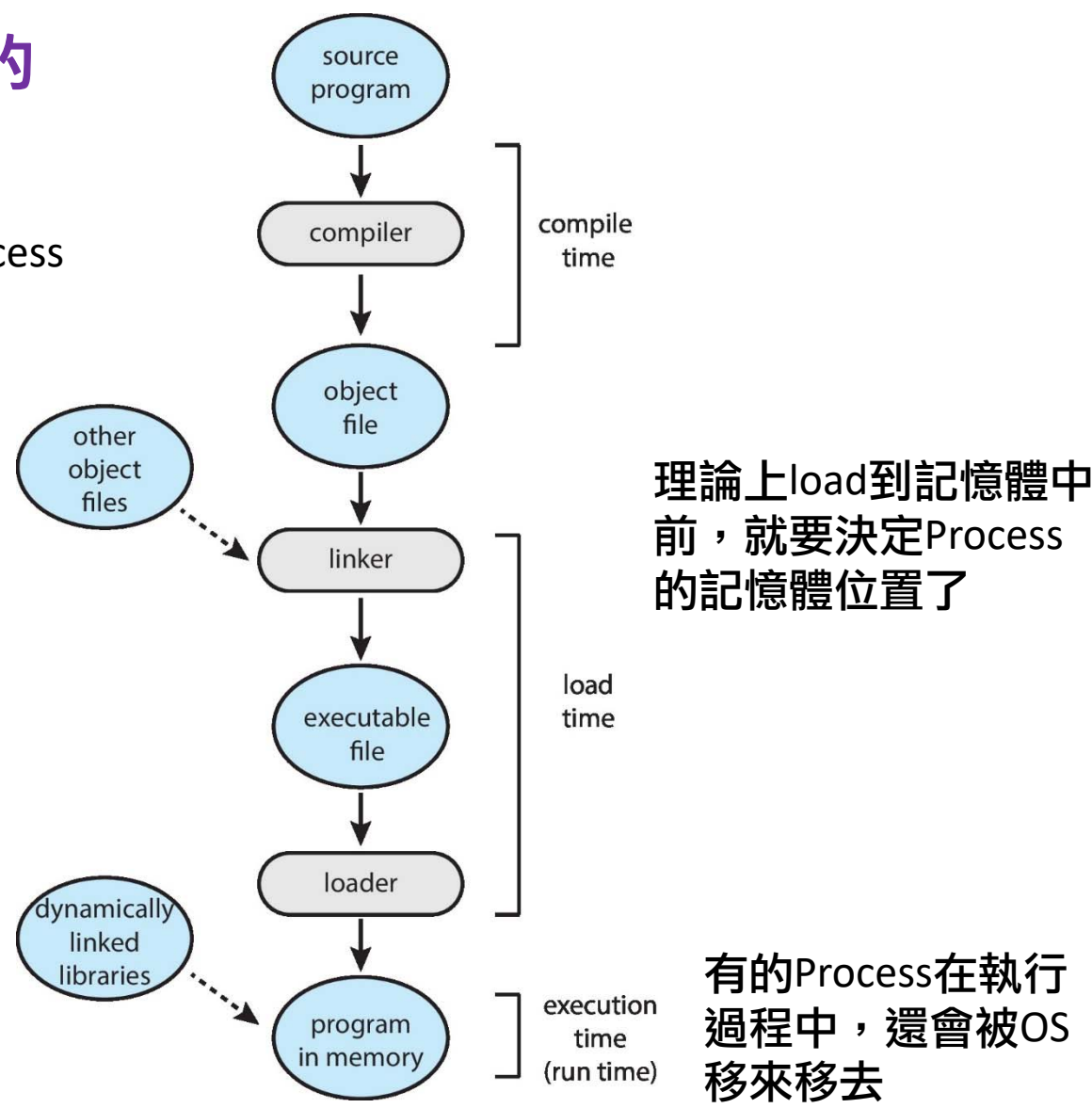
Base: 限制了實體位址的存取下限



確保process只存取自己的memory space

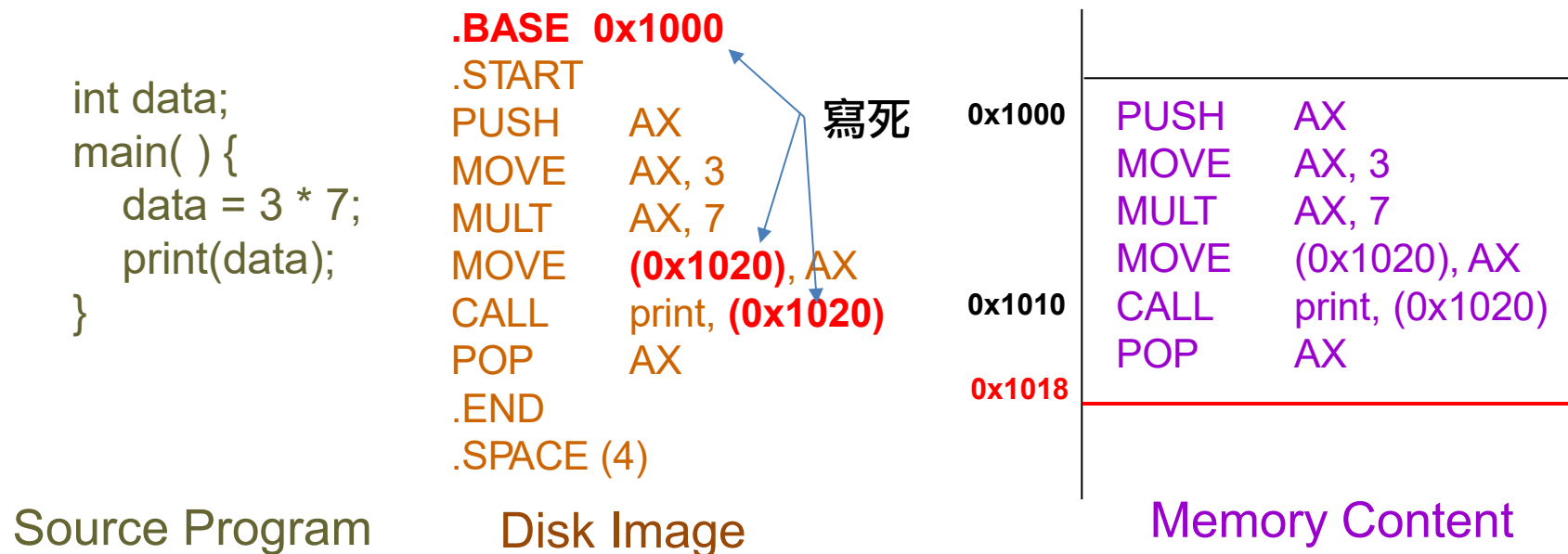
那時候決定Process的Physical Address?

Address binding: 「決定Process的Physical address的時機」



Compile Time Address Binding

- Compiler generates absolute address
- If (memory) location changes → recompile
- Example: MS-DOS .COM format binary



.SPACE(4): every word has four bytes

Load Time Address Binding

- Compiler generates relocatable address representations
- Relocatable code 所有程式內結構的定址，以相對於.BS的位址來表示
 - Can be run from any memory location
 - If starting location changes → reload the code
 - 理由: .BS值load time產生，一旦load完，.BS值就不能再動

Source Program	Disk Image	Memory Content (After Load)
<pre>int data; main() { data = 3 * 7; print(data); }</pre>	<pre>.START PUSH AX MOVE AX, 3 MULT AX, 7 MOVE (.BS+0x20), AX CALL print, (.BS+0x20) POP AX .END .SPACE (4)</pre>	<div><div>.BS 0x2000</div><div>0x2010</div><div>0x2018</div><div>PUSH AX MOVE AX, 3 MULT AX, 7 MOVE (0x2020), AX CALL print, (0x2020) POP AX</div></div>

Execution Time Address Binding

執行時，Process程式不知道自己的實體位址→要靠MMU輔助

- Compiler generates logical address
- Special hardware (MMU) is needed for this scheme
 - MMU transforms logical addr. into physical addr.
 - 執行時，程式可被任意搬動
- Most general-purpose OS use this method

int data;	.START
main() {	PUSH AX
data = 3 * 7;	MOVE AX, 3
print(data);	MULT AX, 7
}	MOVE (0x20), AX
	CALL print, (0x20)
	POP AX
	.END
	.SPACE (4)

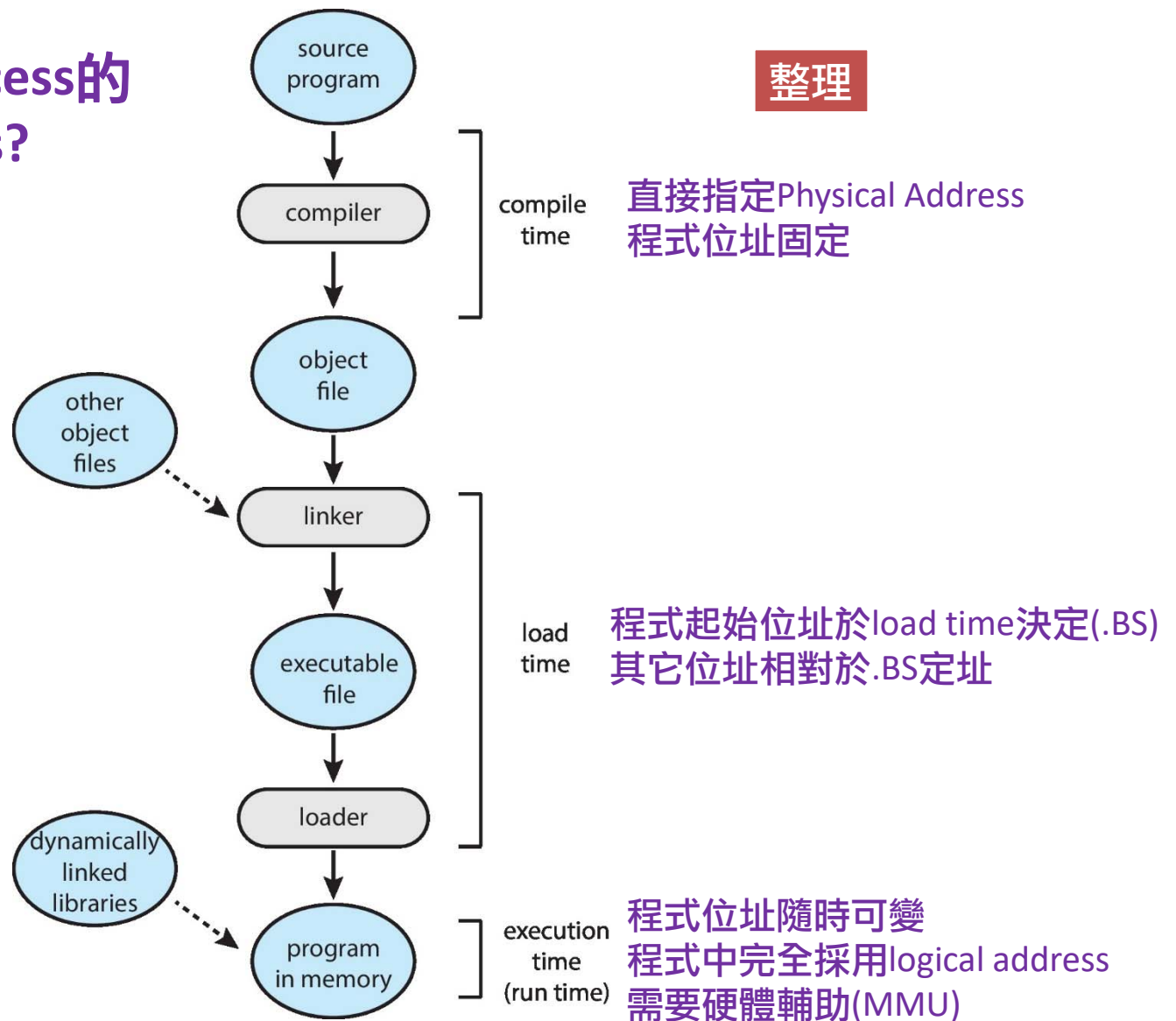
Source Program

Disk Image

0x2000	PUSH AX
	MOVE AX, 3
	MULT AX, 7
	MOVE (0x20), AX
0x2010	CALL print, (0x20)
	POP AX
0x2018	

Memory Content

那時候決定Process的Physical Address?



Logical vs. Physical Address

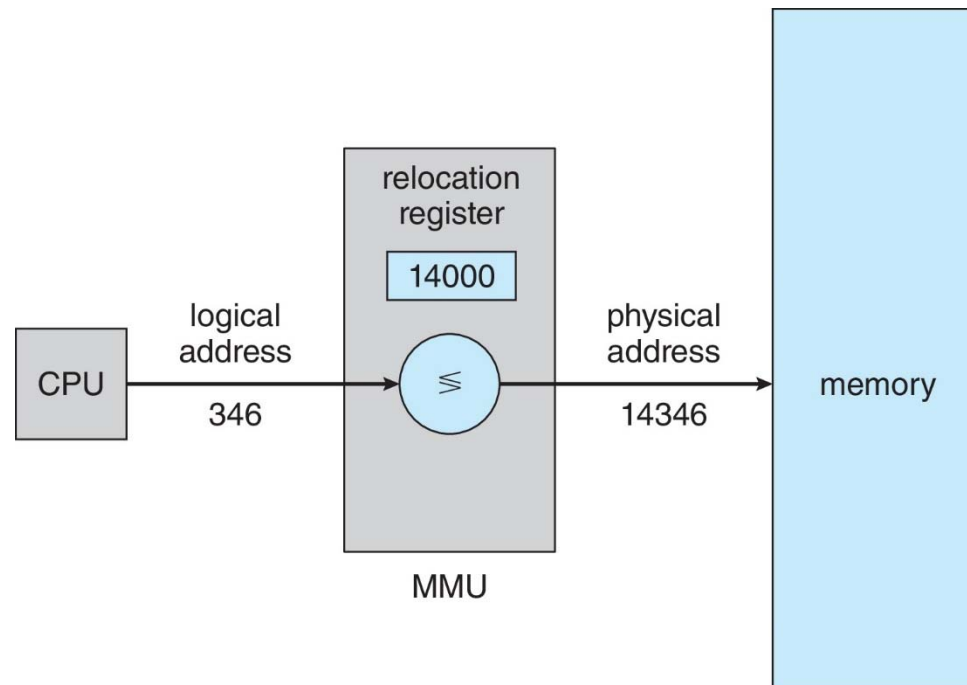
- Compile-time & load-time address binding
 - (Code in memory) logical addr = physical addr
- Execution-time address binding (DLL)
 - logical addr \neq physical addr; must be assisted by MMU
 - The user program is only aware of logical addresses
 - Never sees the physical addresses

“執行時期”記憶體空間配置與管理

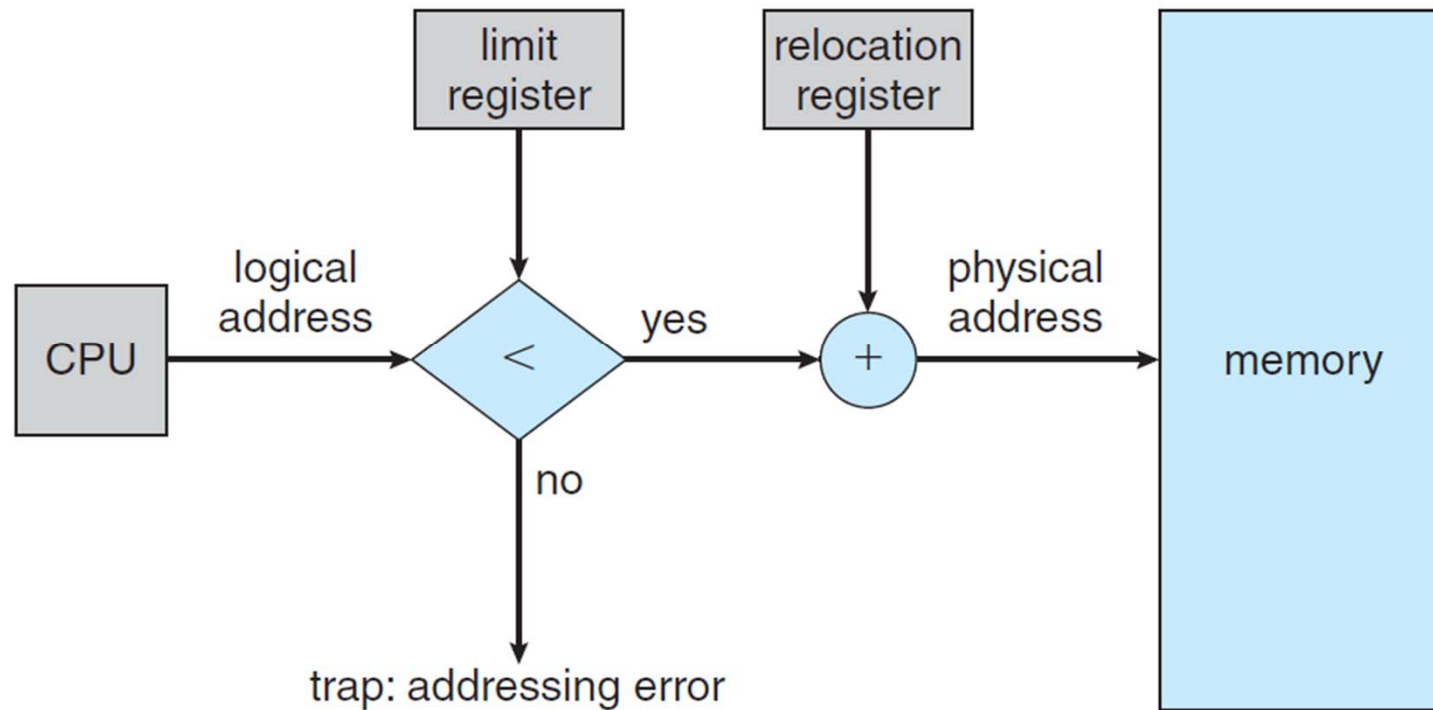
- Protecting the address space of a process
 - Developer writing programs using logical address
 - Starting from 0
- Address binding (Logical-Physical address mapping)
 - 使用硬體專責管理記憶體配置 (位址映射)
 - 標示每個程式的“0”是從那一個(實體)位址起算
 - MMU: Memory Management Unit
 - Handles logical-physical address mapping

Memory-Management Unit (MMU)

- Hardware device that maps logical to physical address
- MMU中包含多個relocation registers
 - The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



加上Boundary Check



Dynamic Loading

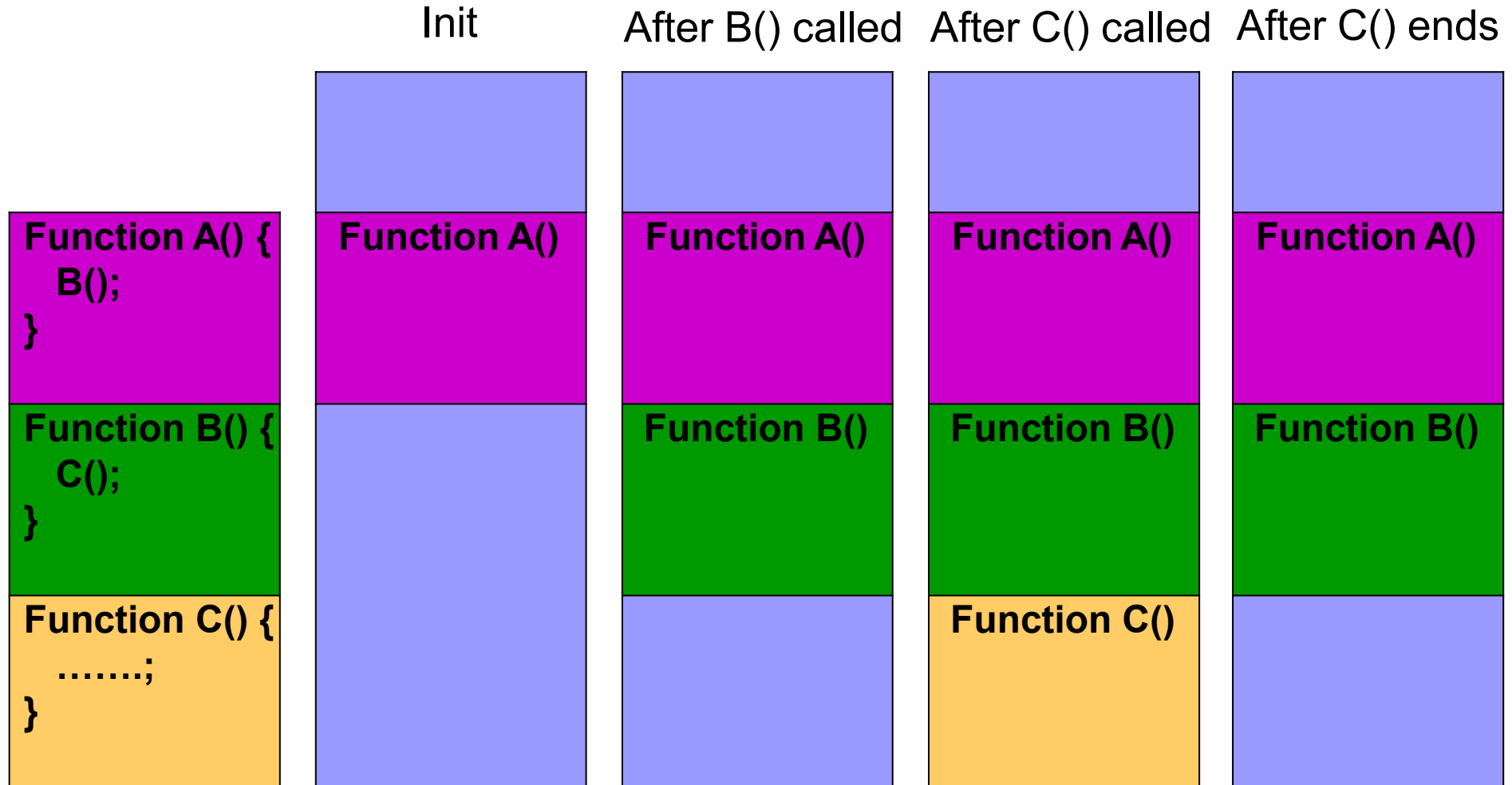
- Definition
 - A routine is loaded into memory “only” when it is called
- Better memory-space utilization
 - Un-used routine is never loaded
 - Particularly useful when large amounts of code are infrequently used (e.g., error handling code)
- No special support from OS
 - Users must implement themselves
 - OS may provide help via library or API calls
 - `plugin = dlopen(file_name, RTLD_NOW);`

同一支程式，不同function

Dynamic Loading

Disk image

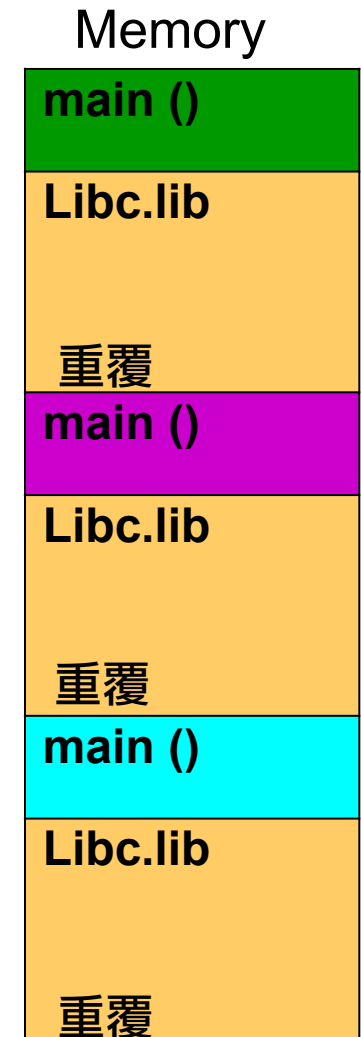
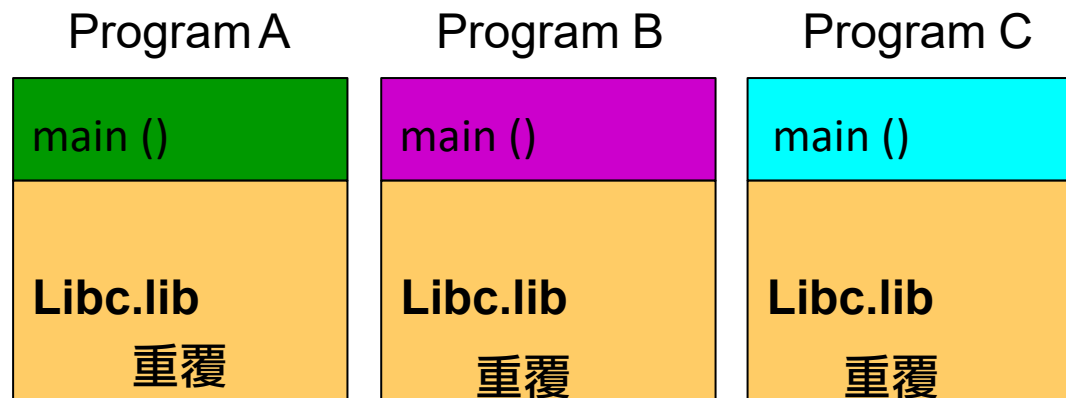
Memory content



不同程式，同一個library

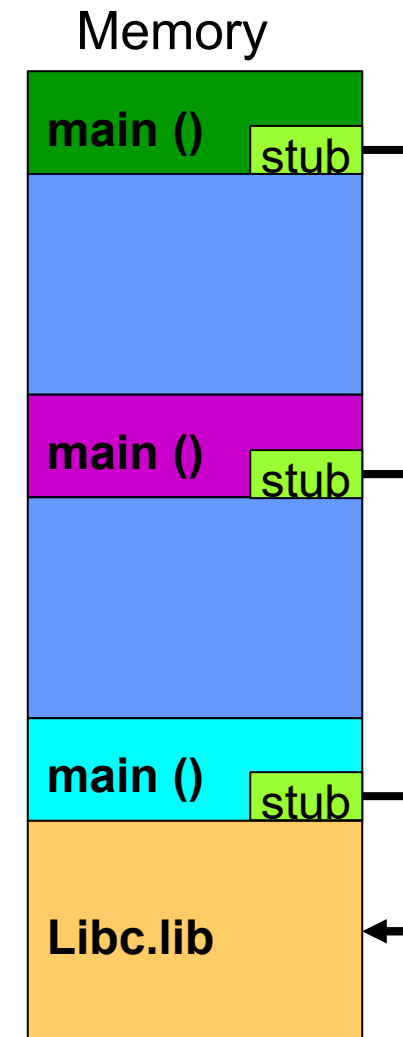
Static Linking

- Static linking: libraries are combined by the loader into the program (in-memory image)
 - Waste memory: duplicated code
 - Faster during execution time
- Static linking + Dynamic loading
 - Still can't prevent duplicated code
 - Dynamic loading是針對一支程式中的多個function
 - 二個並用，整個系統多支程式還是會有重覆code問題



Dynamic Linking

- Dynamic linking: Linking postponed until execution time
 - Only one shared code copy in memory
 - A **stub** is included in the program in-memory image for each lib reference
 - Stub call → check if the referred lib is in memory → if not, load the lib → execute the lib
- Ex: DLL (Dynamic link library) on Windows



Example

```
#include <windows.h>
```

```
#include <stdio.h>
```

```
// Import function that adds two numbers
```

```
extern "C" __declspec(dllimport) double AddNumbers(double a, double b);
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    double result = AddNumbers(1, 2);
```

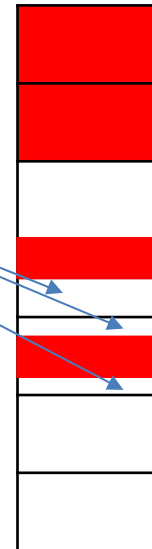
```
    printf("The result was: %f\n", result);
```

```
    return 0;
```

```
}
```

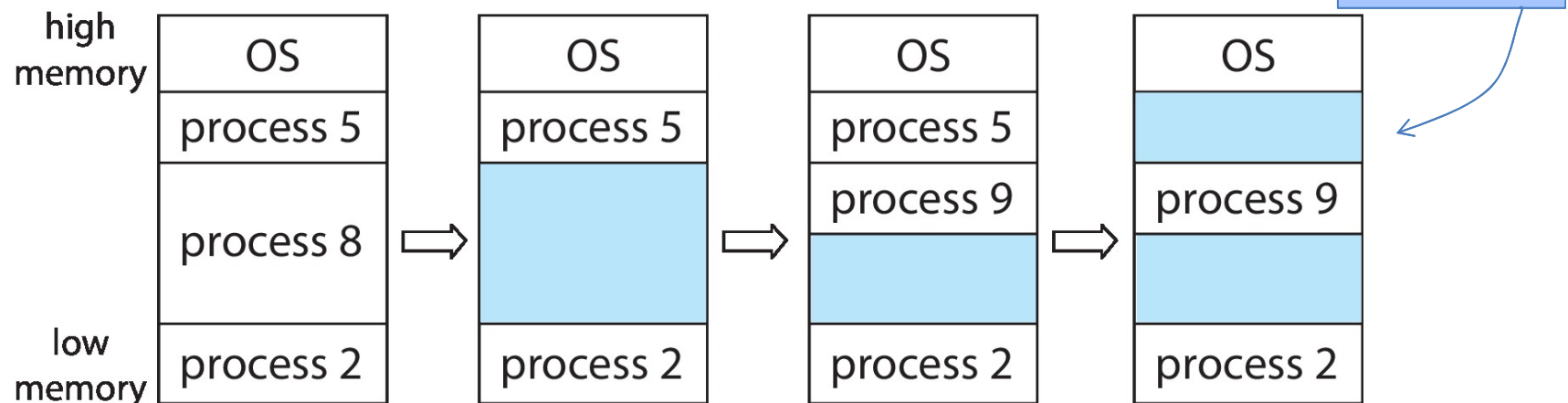
Memory Allocation

- Fixed-partition allocation (paging):
 - Each process loads into one partition of fixed-size
 - Degree of multi-programming is bounded by the number of partitions
 - May have internal fragmentation
- Variable-size partition
 - May have external fragmentation
 - Hole: block of contiguous free memory
 - Holes of various size are scattered in memory



Multiple Partition (Variable-Size) Method

- When a process arrives, it is allocated a hole large enough to accommodate it
- The OS maintains info. on each in-use and free hole
- A freed hole can be merged with another hole to form a larger hole



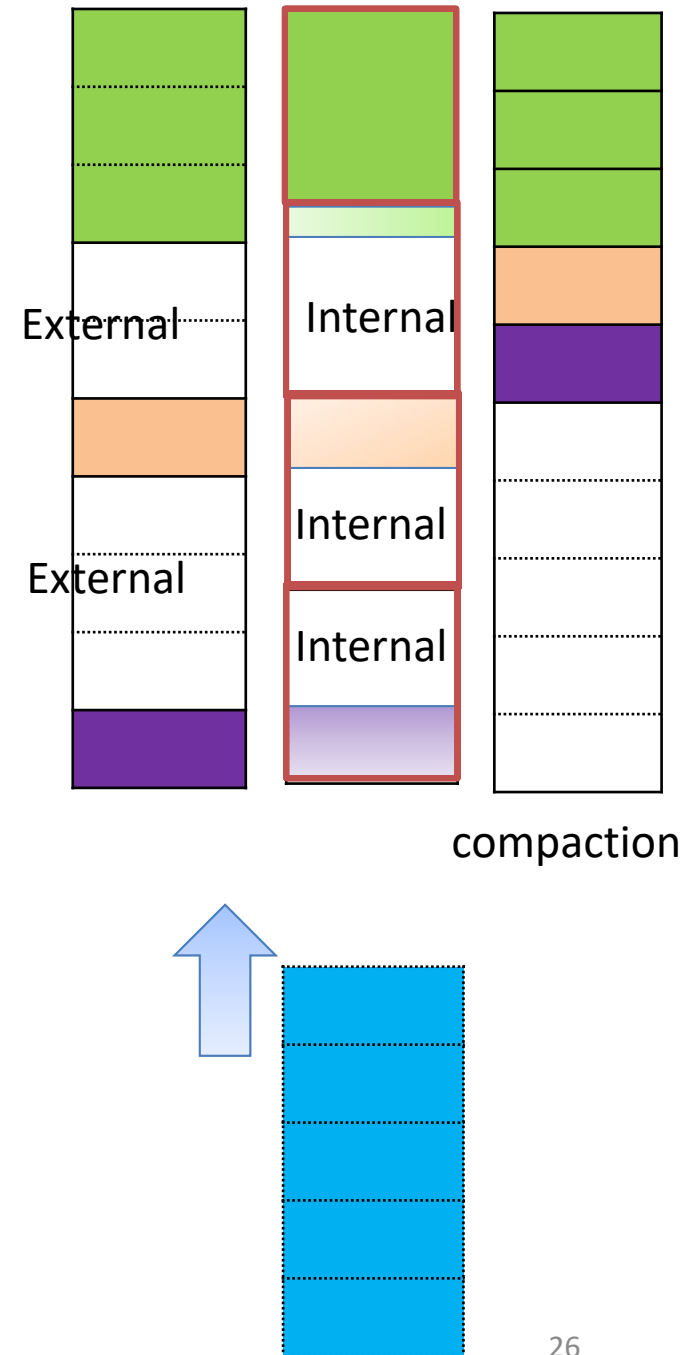
Problem of Dynamic Storage Allocation

- How to satisfy a request of size n from a list of free holes
 - First-fit – allocate the 1st hole that fits
 - Best-fit – allocate the smallest hole that fits
 - Must search through the whole list
 - Worst-fit – allocate the largest hole
 - Must also search through the whole list
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

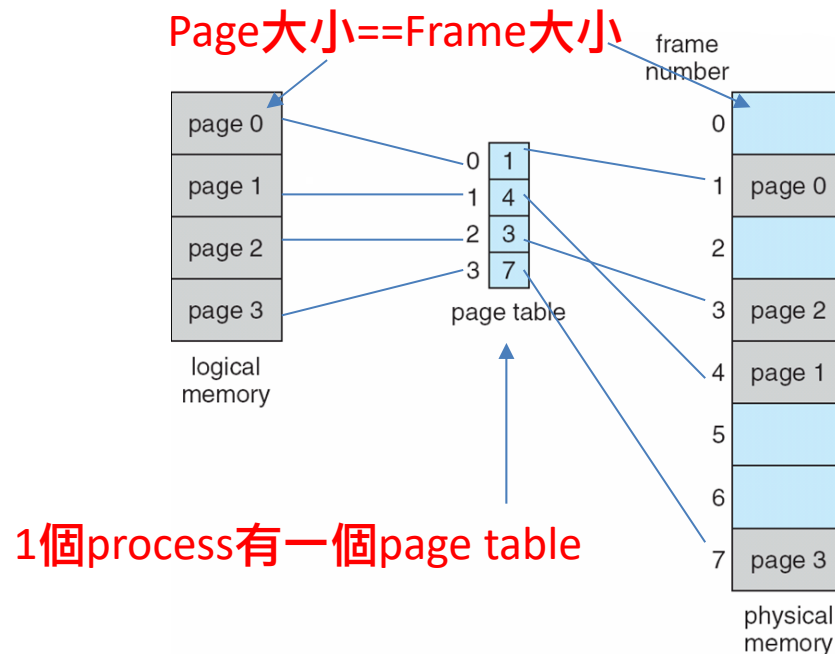
- External fragmentation
 - Total free memory space is big enough to satisfy a request, but is not contiguous
 - Occur in variable-size allocation
- Internal fragmentation
 - Memory that is internal to a partition but is not being used
 - Occur in fixed-partition allocation
- Solution: compaction
 - Move the memory contents to place all free memory together in one large block at execution time
 - Valid only if binding is done at **execution time** (why?)

Load time binding → .BS決定後就不能再改!



Paging

- Method
 - Divide physical memory into fixed-sized blocks called frames
 - Divide logical address space into blocks of the same size called pages
 - A program of **n** pages, need **n** free frames and load the program
 - A page table (for each process) to translate logical to physical addresses



Paging

- Benefit:
 - Allow the physical-address space of a process to be **non-contiguous** 整個process分成好幾塊來放
 - Avoid external fragmentation (frame size == page size)
 - Limited internal fragmentation (within size)
- Note
 - Page size == Frame size
 - (Process) Page 數量 未必 == (全系統) Frame 數量
 - Page number > Frame number → virtual memory
 - Frame number > Page number → multi-process

Example

- How many pages (frames) are needed?
 - Process size = 72766 bytes
 - Page size = 2048 bytes

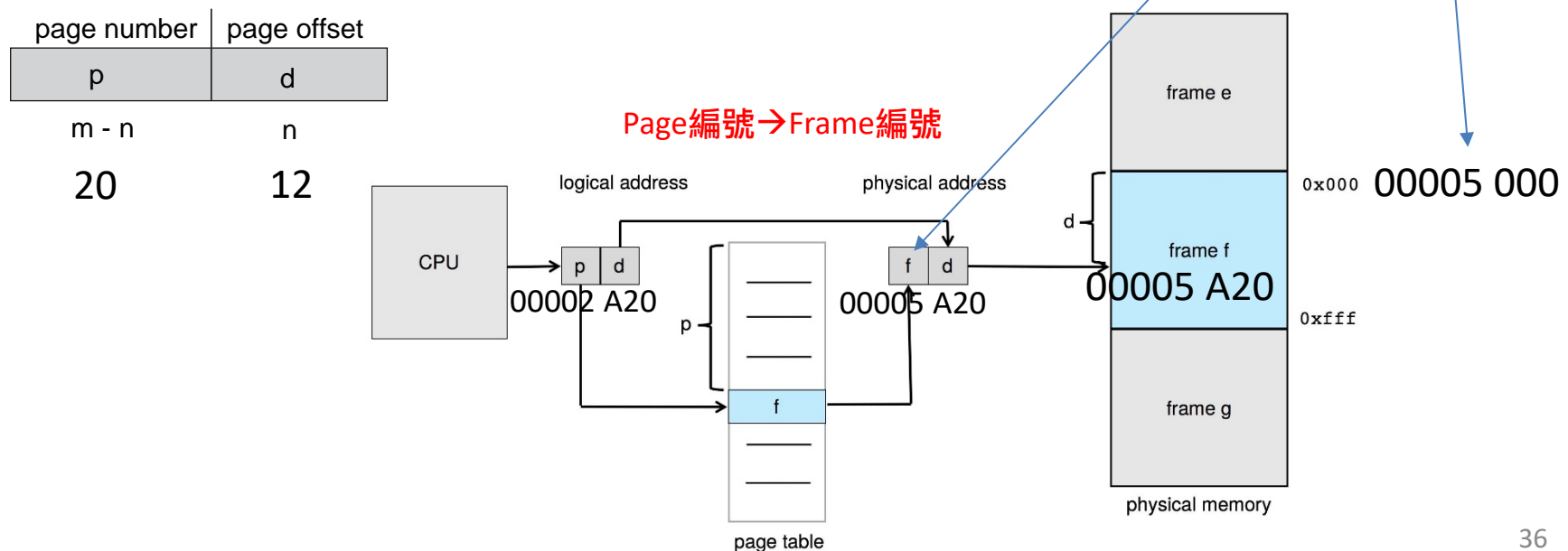
$$\left\lceil \frac{72766}{2048} \right\rceil = 36$$

Address Translation Scheme

- | page number | page offset |
|-------------|-------------|
| p | d |
- Address is divided into two parts:
 - Page number (p)
 - Used as an index into a page table
 - $m-n$ bits means a process can allocate at most 2^{m-n} pages (有幾個pages)
 - page number (2^{m-n}) \times page size (2^n) = logical memory size (2^m)
 - Page offset (d)
 - combined with (page) base address to define the physical memory address that is sent to the memory unit
 - n bits means the page size is 2^n (1個page的大小; 通常故意設成2的n次方, why?)

Page Table

- Entry
 - Key: page number; Value: frame number
 - Frame number is the base address of a page in physical memory d補0
- A structure maintained by OS for each process
 - Page table includes only pages owned by the process
 - A process cannot access memory outside its space



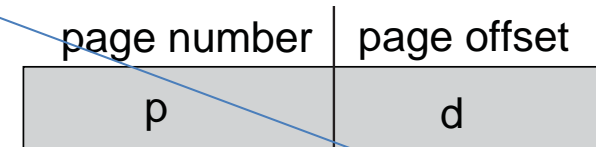
Address Translation Scheme

- 假定: Page size:1KB(2^{10}) & Page 2 對到 frame 5

page size $2^{10} \Rightarrow n = 10$

- Given 13 bits logical and physical address and $d=20$, what is physical address?

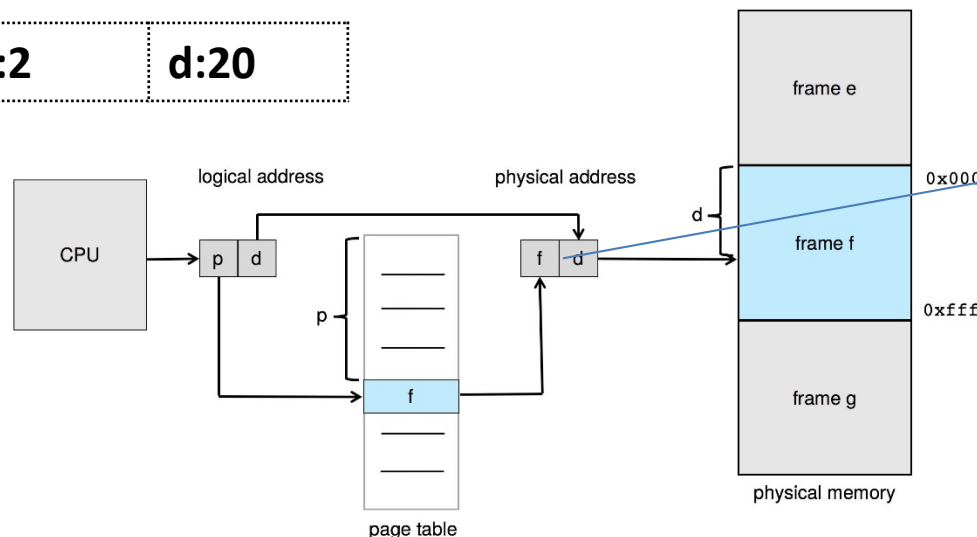
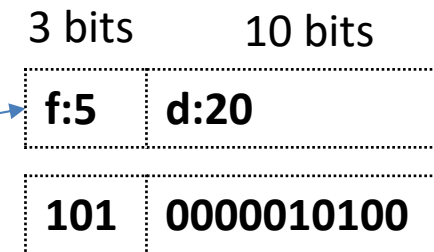
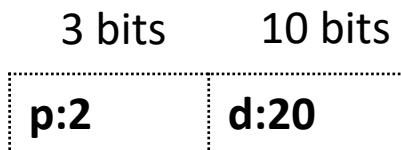
13 bits logical address: $m=13 \Rightarrow m-n=3$



值
位數

$m - n$
3

n
10



Example

logical: 1 page 4 bytes $\Rightarrow n = 2$;

Page共4個 \Rightarrow 只需2bits $\Rightarrow m-n=2$

\Rightarrow logical位址長度=2+2=4bits

Frame共8個(2^3)

page size=frame size=4bytes; ($n=2 \Rightarrow 2^2=4$)

physical memory=32bytes $\Rightarrow \frac{32}{4} = 8$ physical frames

16 bytes

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

0	5
1	6
2	1
3	2

page table

32 bytes

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

Logical address定址能力: $2^4=16$ bytes

p	d
---	---

4-2=2 bits 2 bits

Page的數量
(索引)

Page的大小

page number	page offset
p	d

$m - n$

n

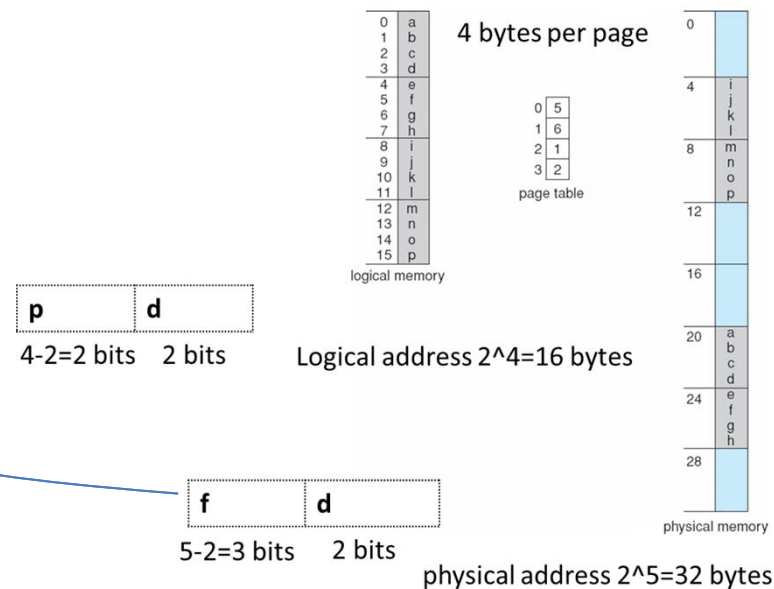
physical address定址能力 $2^5=32$ bytes

f	d
---	---

5-2=3 bits 2 bits

Address Translation

- Total number of pages does not need to be the same as the total number of frames
 - Frame size = page size
 - Total # pages determines the logical memory size
 - Total # frames depending on the physical memory size



Example

- Given 32 bits logical address, 36 bits physical address and 4KB page size



logical: $x + y = 32$

physical: $z + y = 36$

page size: $4KB = 2^2 \times 2^{10} = 2^{12} \Rightarrow y = 12$

$x = 32 - 12 = 20$

$z = 36 - 12 = 24$

Page的數量
(索引)

Page的大小

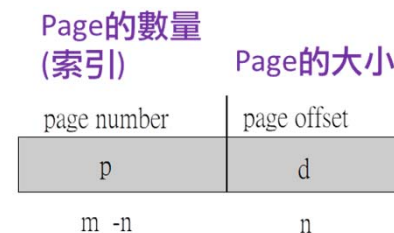
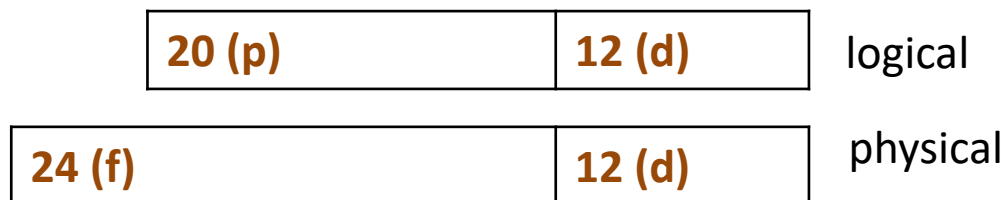
page number	page offset
p	d

m - n

n

Example

- Given 32 bits logical address, 36 bits physical address and 4KB (2^{12}) page size
 - Page table entries $32 - 12 = 20 \Rightarrow 2^{20}$ entries
 - Max program (logical) memory 2^{32} bytes
 - Total physical memory 2^{36} bytes
 - Number of bits for page number (p) 20
 - Number of bits for frame number (f) 24
 - Number of bits for page offset (d) 12



Free-Frame List

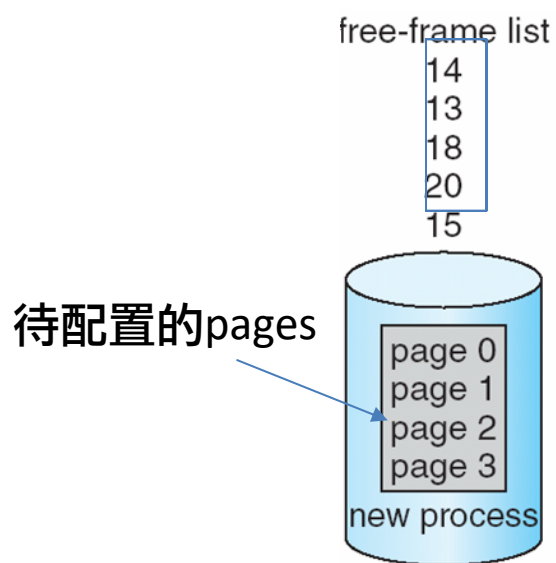
- When a page fault occurs, the OS brings the desired page from disk into memory
- Free-frame list
 - a pool of free frames for satisfying such requests.

head → 7 → 97 → 15 → 126 ... → 75

- OS typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being allocated
- When a system starts up, all available memory is placed on the free-frame list

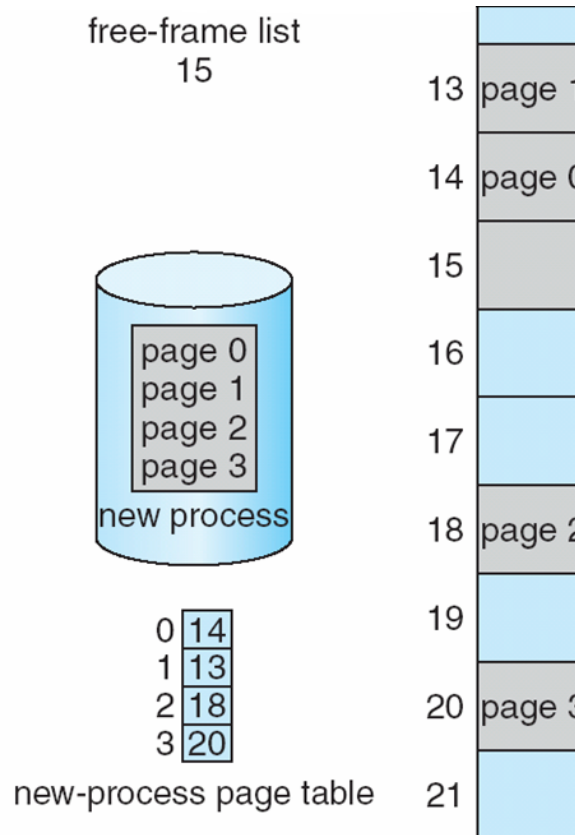
Page的配置: Free-frame List

灰色是原本free的部份



(a)

Before allocation



(b)

After allocation

Page (Frame) Size

- The page (frame) size is defined by hardware
 - Typically a power of 2
 - Ranging from 512 bytes to 16MB / page
 - 4KB / 8KB page size is commonly used $1K = 1 \times 1024 = 1 \times 2^{10}$
- Internal fragmentation issue
 - Larger page size → More space waste
- In practice, page size **has grown over time**
 - Memory, process, data sets have become larger
 - Better I/O performance (during page fault)
 - Make page table smaller (page table is a pure overhead)
 - Page size愈大，切出來的pages就愈少→page table愈小

Page Table Summary

- Address abstraction
 - Paging helps separate user's (process's) view of memory and the actual physical memory
 - User view's memory: one single contiguous space
 - Actually, user's memory is scatter out in physical memory
- OS maintains
 - One page table for each process
 - One frame table for managing physical memory
 - One entry for each physical frame
 - Indicate whether a frame is free or allocated
 - If allocated, to which page of which process or processes

Implementation of Page Table

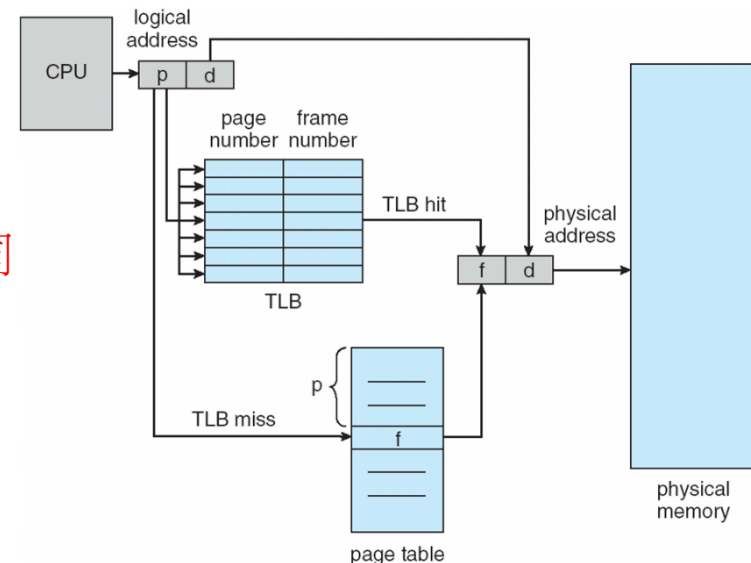
- Memory: stores the page tables
- 如何找到Page Table: page-table base register (PTBR) (方便CPU直接找到Page table)
 - Stores the physical address of the page table
 - 直接指向現正處理process的page table
 - The value is stored in PCB (Process Control Block)
 - Changing the value of PTBR during context-switch
- 1 memory reference == 2 memory reads
 - One for the page table and one for the real address
 - Can be enhanced by using TLB (Translation Lookaside Buffer)
 - Implemented with fast associative memory (HW)
 - Key: page number; value: frame number
 - TLB == Page Table 的cache

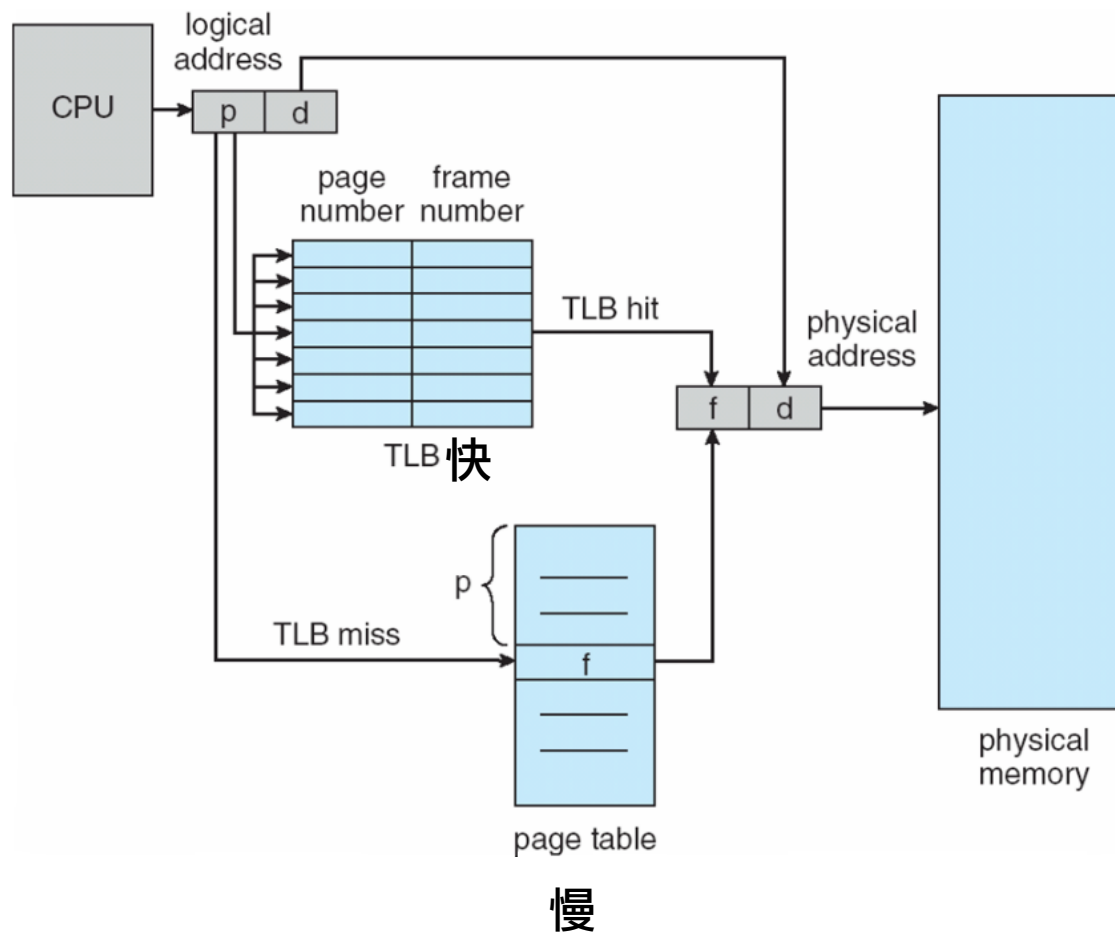
Translation Look-aside Buffer (TLB)

P.365-P.367

- A cache for page table shared by all processes
 - 先找cache，miss才查memory上的page table
 - Issue: page numbers are specific to each process
- TLB must be flushed after a context switch
 - Otherwise, TLB entry must have a PID field (address-space identifiers (ASIDs))

每個不同的process有相同的page number:
Process A的page 2和Process B的page 2內容不同
(指向不同的physical address!)





Effective Memory-Access Time

- Assume
 - 20 ns for TLB search
 - 100 ns for memory access
- Effective Memory-Access Time (EMAT)
 - 70% TLB hit-ratio:

No TLB: 200ns

$$\text{EMAT} = 0.70 \times \overset{\text{TLB}}{\underset{\text{Hit}}{20}} + \overset{\text{Mem}}{100} + (1-0.70) * \overset{\text{TLB}}{\underset{\text{Miss}}{20}} + \overset{\text{PT}}{100} + \overset{\text{Mem}}{100} = 150 \text{ ns}$$
 - 98% TLB hit-ratio
$$\text{EMAT} = 0.98 \times \underset{\text{Hit}}{120} + 0.02 \times \underset{\text{Miss}}{220} = 122 \text{ ns}$$

Memory Protection

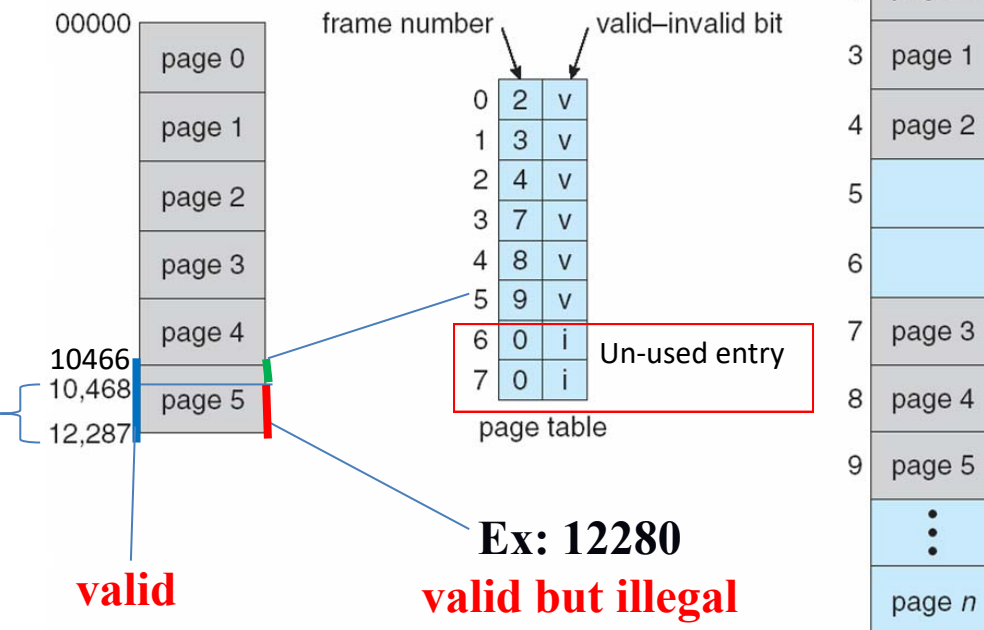
- Each page is associated with a set of **protection bit** in the page table
 - E.g., a bit to define read/write/execution permission
- Common use: valid-invalid bit
 - Valid: the page/frame is **in** the process' logical address space, and is thus a legal page
 - Invalid: the page/frame is **not in** the process' logical address space
 - 結合Virtual Memory使用時，有不同定義!
 - invalid代表in logical address space but not in memory

Valid-Invalid Bit

- Potential issues:
 - Unused PT entry: Un-used entry causes memory waste → use page table length register (PTLR) 計算Page Table長度
 - Internal overflow: Process memory may NOT be on the boundary of a page → memory limit register is still needed

在右邊的例子中，page 5邏輯位址最多到12287
10466-12287都是valid

程式只佔到10468; 所以
10469-12287都是無意義資料

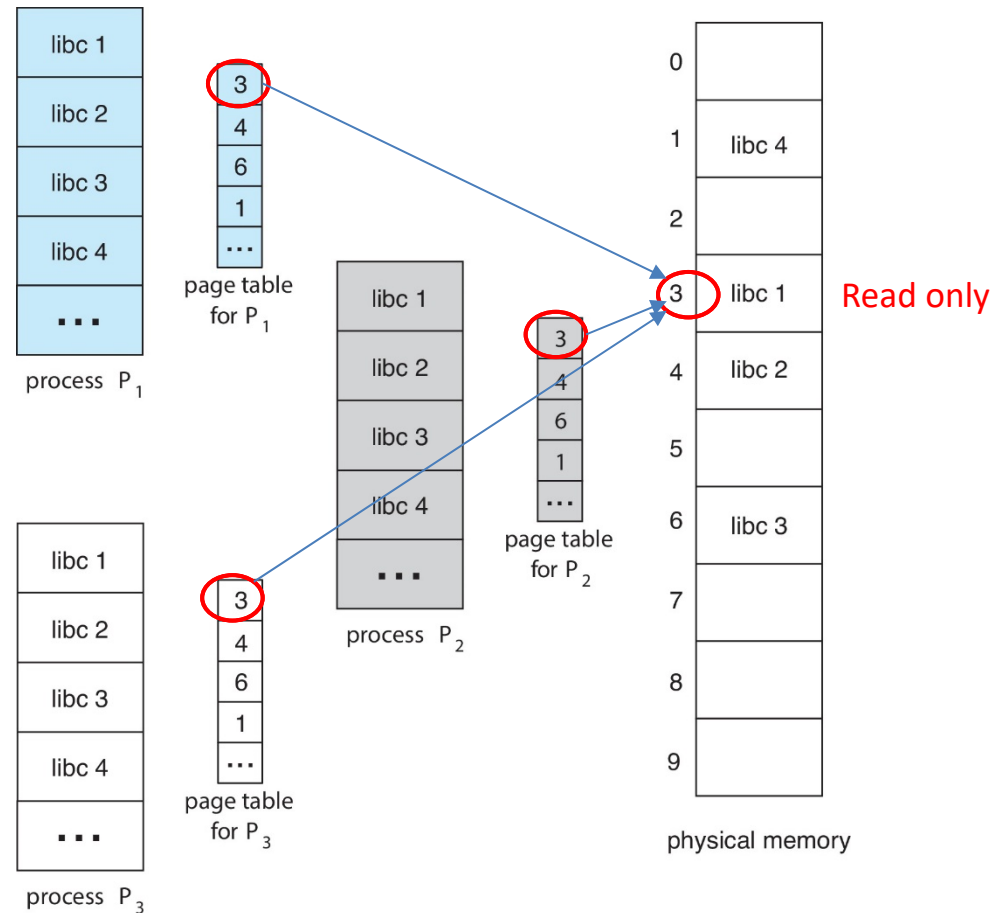


Shared Pages

- Paging allows processes share common code, which must be reentrant
 - The shared code never change during execution (immutable)
 - One copy of the code is kept in physical memory
 - Two (or more) virtual addresses are mapped to one physical address
- Process keeps a copy of its own private data and code

Shared Pages by Page Table

不同process的page table entry指向同一個physical frame



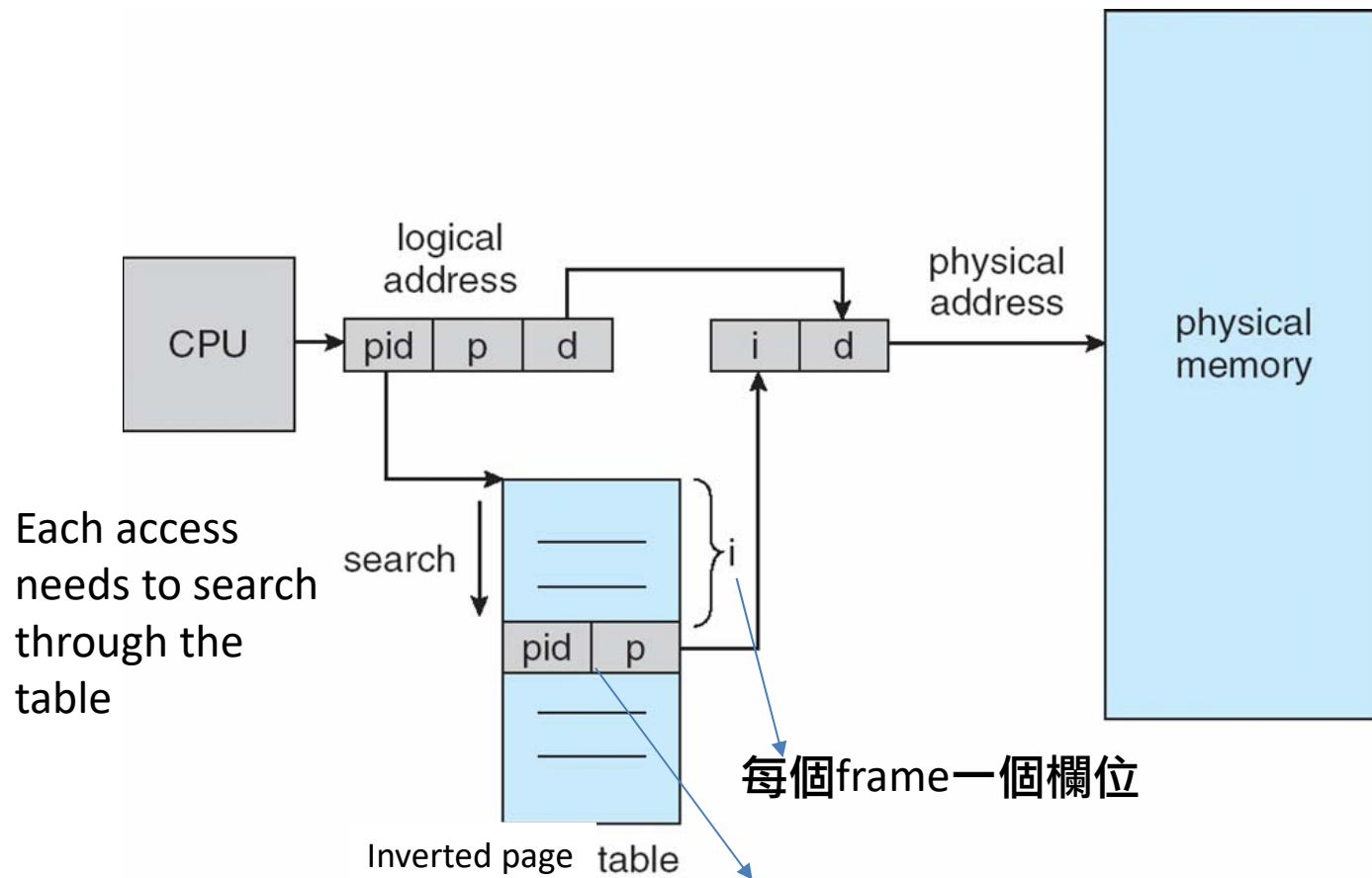
Page Table 太大的問題

- Page table could be huge and difficult to be loaded
 - 4GB (2^{32}) logical address space with 4KB (2^{12}) page
 - 1 million (2^{20}) page table entry
 - Assume each entry need 4 bytes (32bits)
 - Total size=4MB
 - 狀況: Page Table (4M)遠比1個Page (4K)大 → 不划算; 且page table要連續!
 - Need to break it into several smaller page tables, better within a single page size (i.e. 4KB) 最好每個Page Table片段比一個Page小
 - Or reduce the total size of page table
- Solutions:
 - Inverted Page Table (frame table) → Hard to support shared paging
 - Hierarchical Paging → Additional memory reference、64bit定址不適用
 - Hash Page Table → Additional memory reference

Inverted Page Table

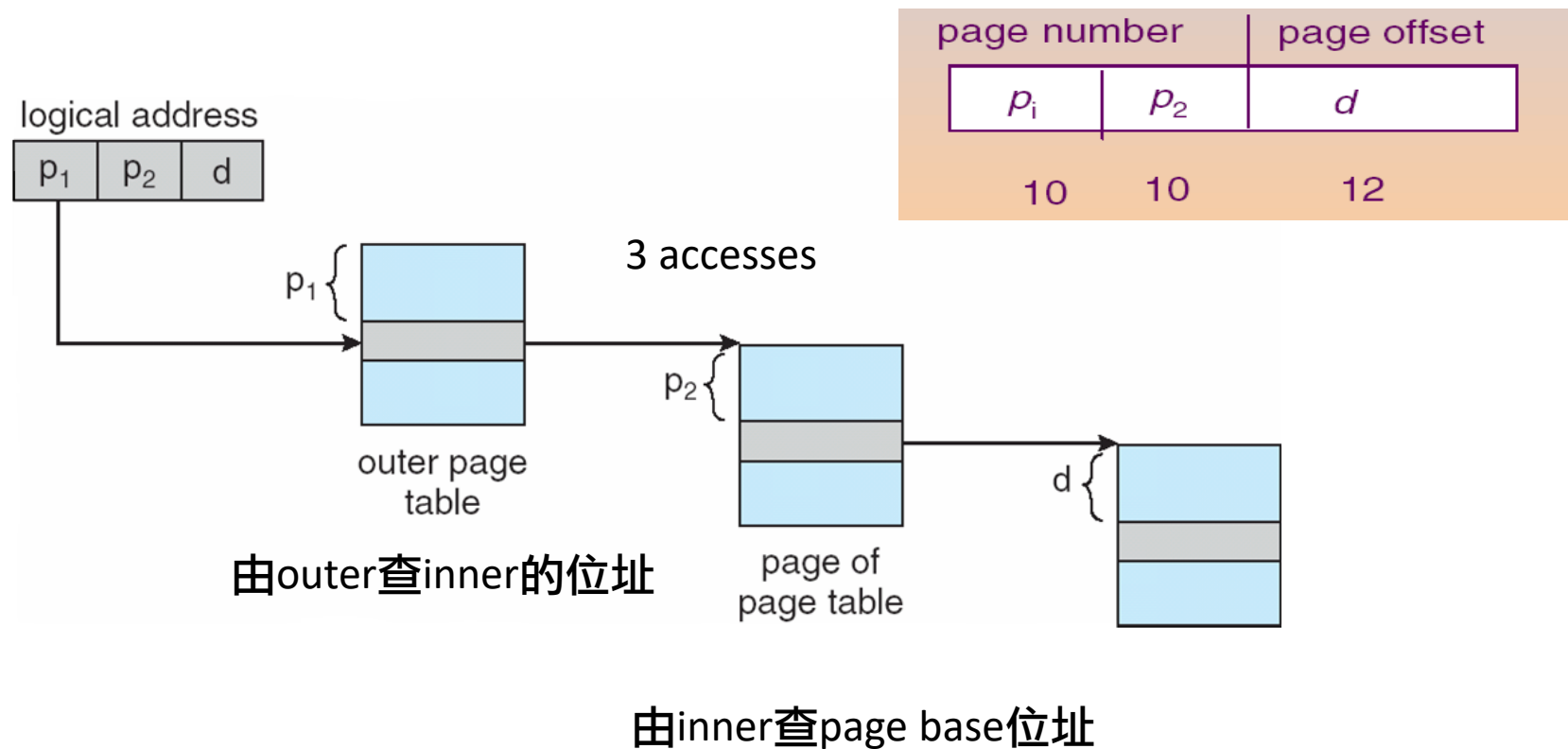
- Maintains **an inverted page table (frame table)** for the whole memory
 - One entry for each real frame of memory
 - Maintains NO page table for each process
- Each entry in the inverted page table has
 - (PID, Page Number)
- Eliminate the memory needed for page tables but **increase memory access time**
 - Each access needs to search through the table (linearly)
 - Solution: use hashing
- Hard to support shared page/memory
 - Why?
因為一個frame只能與一個(PID, Page Number)對應

Inverted Page Table



如果要多個共用一個frame, 這裡勢必要能存多個(pid,p)值

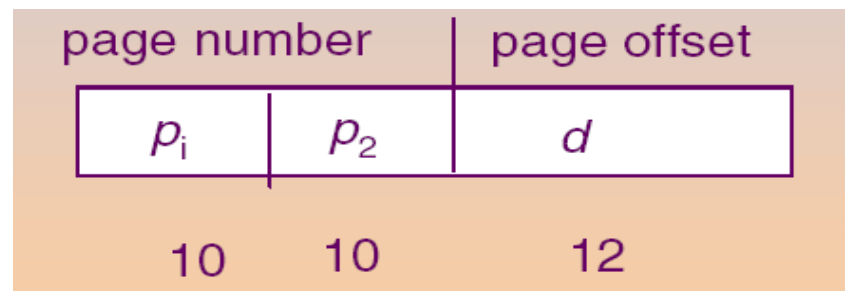
Paging the page table



和原來有何不同: 原來:所有page table entries連續不間斷地放在一起，要「整體」共同進出記憶體
階層式: 因為page table本身也分頁了，所以可以頁為單位進出記憶體 (結合demand paging)

Hierarchical Paging

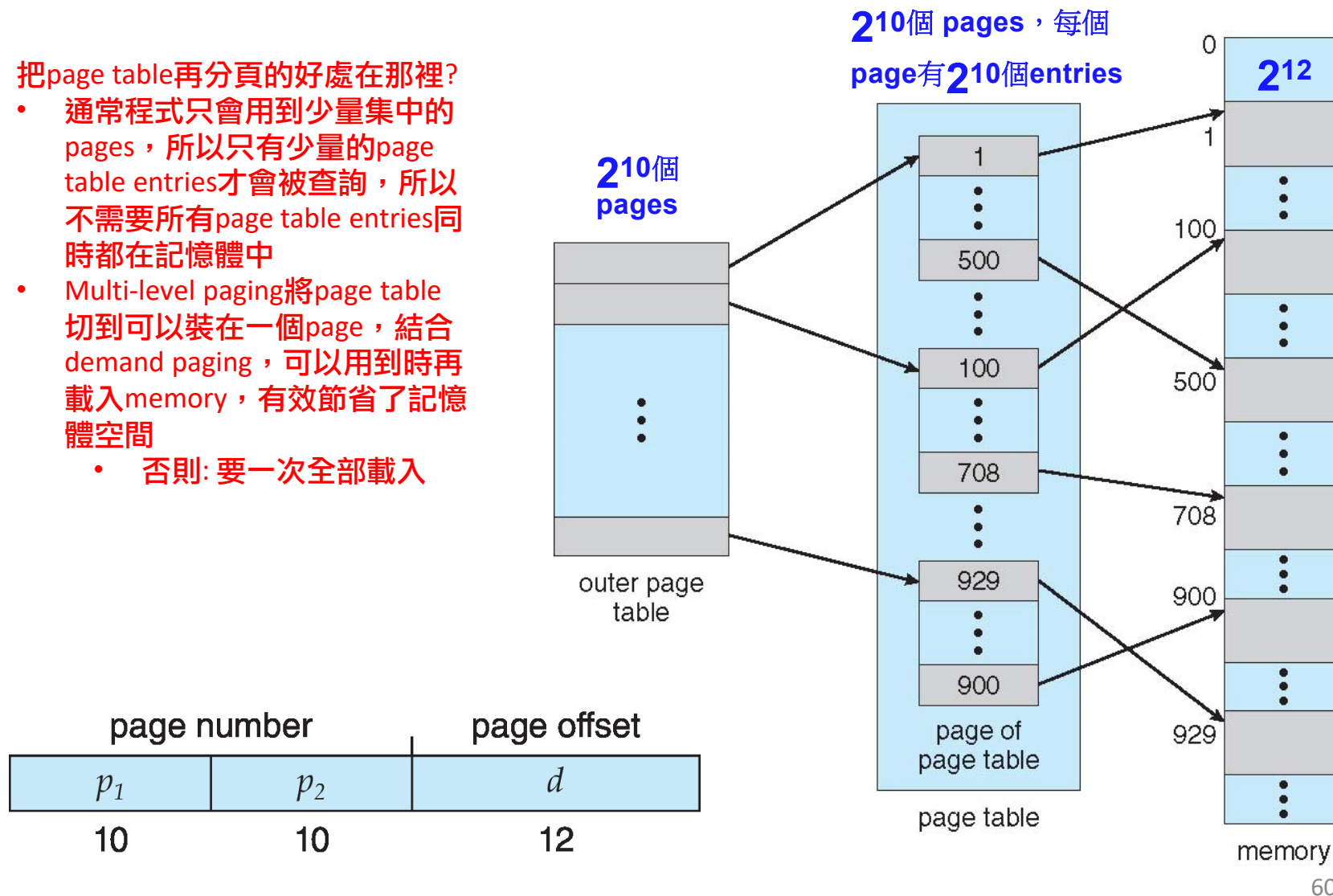
- Break up the logical address space into multiple page tables
 - Paging the page table
 - i.e. n-level page table
- Ex: Two-level paging (32-bit address with 4KB (2^{12}) page size)
 - 12-bit offset (d) \rightarrow 4KB (2^{12}) page size
 - 10-bit outer page number \rightarrow 1K (2^{10}) page table entries
 - 10-bit inner page number \rightarrow 1K (2^{10}) page table entries
 - 3 memory accesses
 - PT 2次+M1次



Two-Level Page Table Example

把page table再分頁的好處在那裡?

- 通常程式只會用到少量集中的pages，所以只有少量的page table entries才會被查詢，所以不需要所有page table entries同時都在記憶體中
- Multi-level paging將page table切到可以裝在一個page，結合demand paging，可以用到時再載入memory，有效節省了記憶體空間
 - 否則: 要一次全部載入

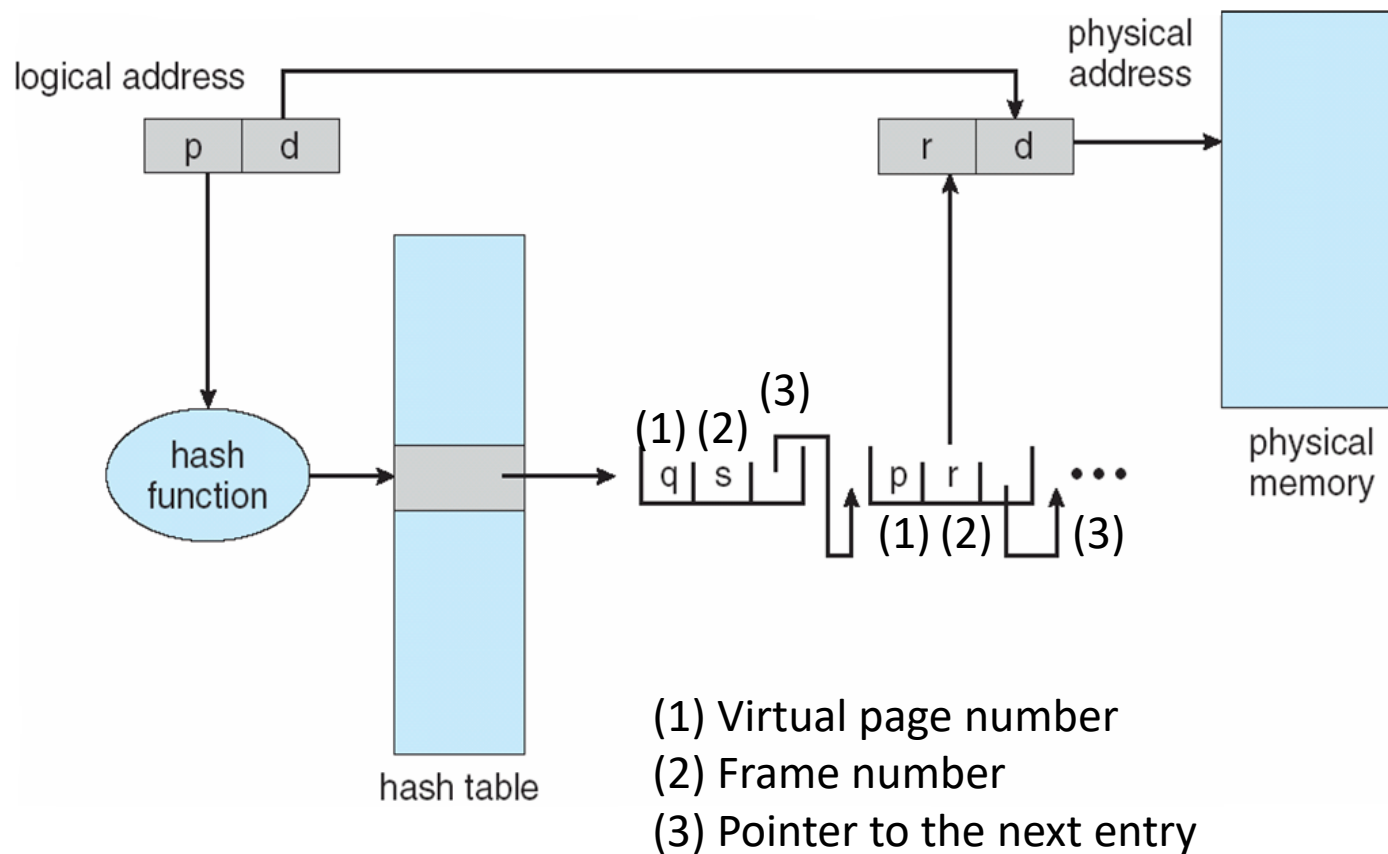


64-bit Logical Address Space

- How about 64-bit address?
 - assume each entry needs 4 bytes
 - $42 (p1) + 10 (p2) + 12 (\text{offset})$
 - outer table requires $2^{42} \times 4B = 16TB$ contiguous memory!!!
 - $12 (p1) + 10 (p2) + 10 (p3) + 10 (p4) + 10 (p5) + 12 (\text{offset})$
 - outer table requires $2^{12} \times 4B = 16KB$ contiguous memory
 - 6 memory accesses!
- For 64-bit computers, hierarchical page tables are generally considered **inappropriate**

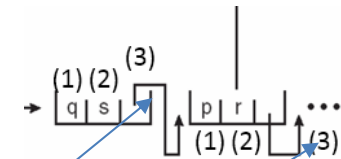
Hashed Page Table

核心想法: 由hash值找出list所在，再逐一比對(1)，match(1)後再由(2)+d找出physical



Hashed Page Table

- Commonly-used for address > 32 bits
- 做法
 - Virtual page number is hashed into a hash table
 - Each entry in the hashed table contains
(Virtual Page Number, Frame Number, Next Pointer)
- Hash table size
 - Larger hash table → smaller chains in each entry
- Problems
 - Pointers waste memory Ex: 64位元中一個pointer需8 bytes (64bits=8bytes)
 - Traverse linked list waste time & cause additional memory references

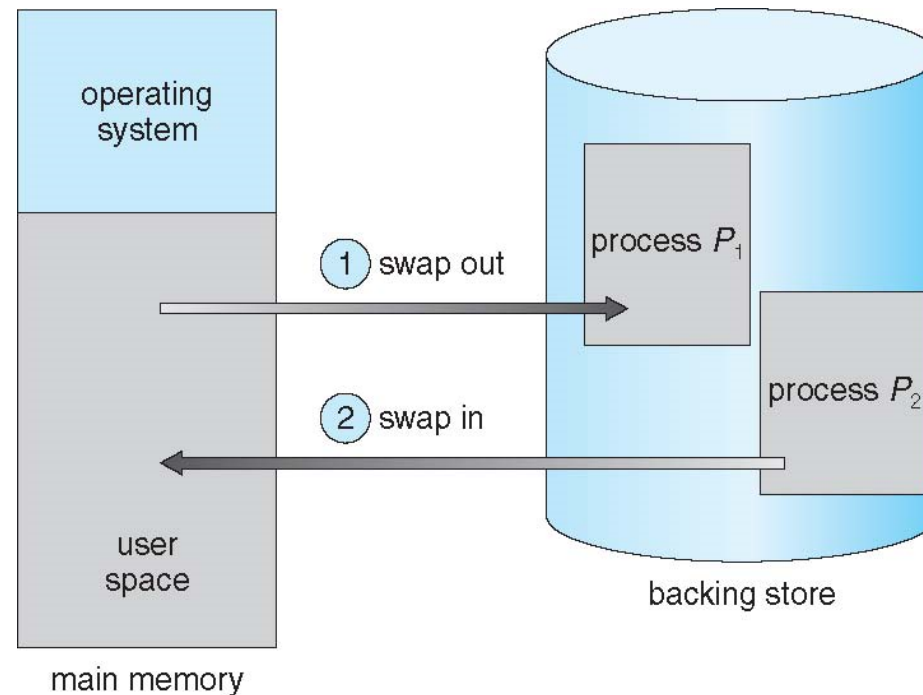


Swapping

- Definition
 - A process can be brought out of memory to a backing store, and later brought back into memory for continuous execution
 - 和paging的不同: swapping是process為單位共同進出記憶體; (demand) paging是以頁為單位進出記憶體
- Backing store
 - A chunk of disk, separated from file system, to provide direct access to these memory images
- Why Swap?
 - Free up memory
 - Roll out, roll in: swap lower-priority process with a higher one

Process Swapping to Backing Store

- Major part of swap time is transfer time
- Total transfer time is directly proportional to the amount of memory swapped



Swapping

- Swap back memory location
 - If binding is done at compile/load time
 - swap back memory address must be the same
 - If binding is done at execution time
 - swap back memory address can be different
- A process to be swapped == must be idle (No CPU direct access)
 - Imagine a process that is waiting for I/O is swapped
 - Solutions:
 - Never swap a process with pending I/O
 - I/O operations are done through OS buffers (i.e. a memory space not belongs to any user processes)

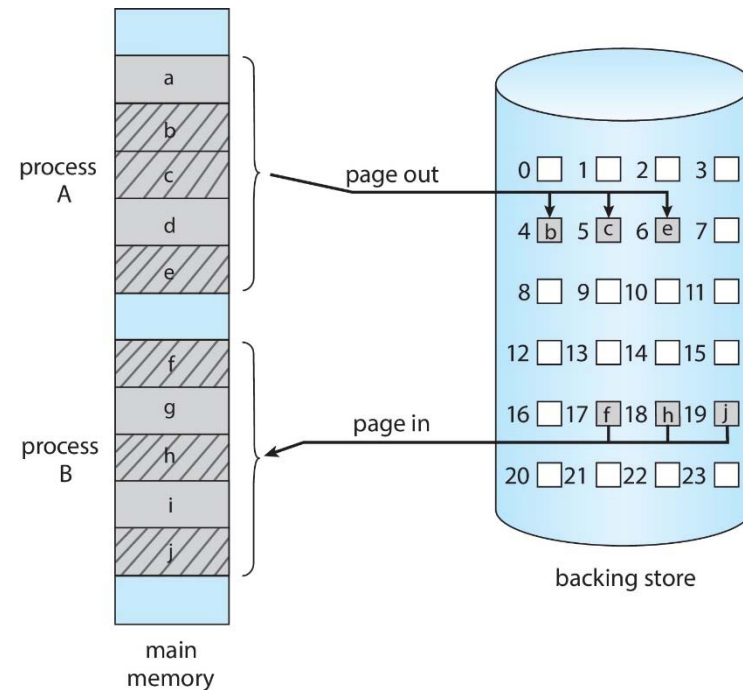
因為CPU此時不能直接存取memory，所以I/O的資料無法存入user process管的memory area

Swapping on Mobile Systems

- Usually does not support
 - Flash memory based
 - Small amount of space
 - Limited number of write cycles
 - Poor throughput between flash memory and CPU on mobile platform
- Use other methods to free memory if low
 - iOS *asks* apps to voluntarily relinquish allocated memory
 - Read-only data thrown out and reloaded from flash if needed
 - Failure to free can result in termination
 - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
 - Both OSes support paging as discussed below

Swapping with Paging = Demand Paging

- Swap only part of a process
 - Some pages belonging to the process
 - Typically combined with virtual memory



Q & A