Chapter 8: Memory Management

Prof. Li-Pin Chang

National Chiao Tung University

Chapter 8: Memory Management

- Address Binding
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Segmentation
- Example: The Intel Pentium

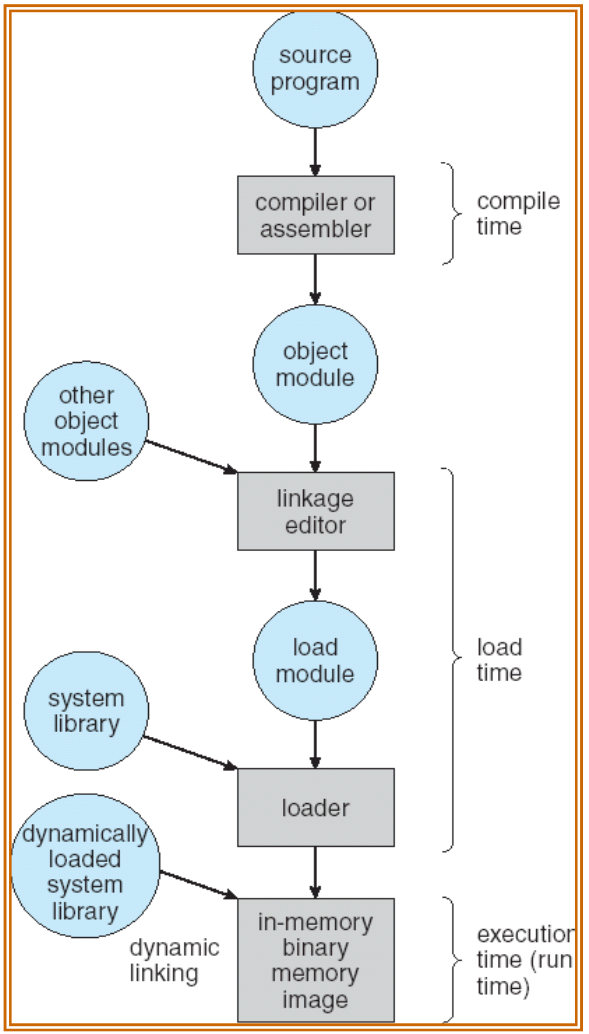
Address Binding

Address Binding

- Program must be brought into memory and placed within a process for it to be run
- Input queue – collection of processes on the disk that are waiting to be brought into memory to run the program
- User programs go through several steps before running

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - Compile time
  - Load time
  - Execution time



Binding of Instructions and Data to Memory

如果能預先知道程式會被放在MM地哪裡，就能產生絕對位置
但是這樣如果位置改變，就必須整個重新compile

- Compile time: If memory location known a priori, absolute code can be generated; must recompile code if the starting location changes

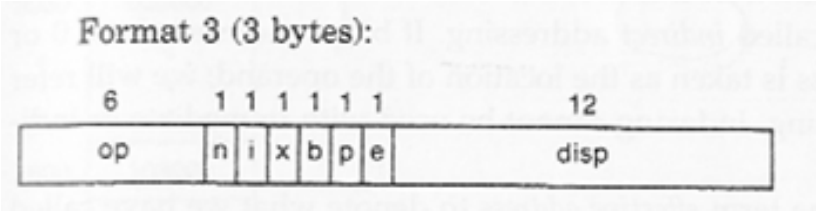
Binding of Instructions and Data to Memory

如果不知道執行時的位置，compiler就必須產生
相對的記憶體位置，在Loading時才決定絕對位置

- Load time: Must generate relocatable code if memory location is not known at compile time
  - Relocatable instructions
  - Non-relocatable instructions

  - Re-locatable instructions of the *SIC* CPU

0017 J CLOOP 3F2FEC



FEC=-14
Target addr=(PC)-14

Binding of Instructions and Data to Memory

- Load time:
  - PC-relative addressing (relocatable) in x86 CPU

```
Target:   xchg eax, 0
          test eax,eax
          jnz target
```

Let each instruction be 4 bytes.
The target address of “jnz” is -12 bytes relative to PC

  - Non-relocatable instructions
    - Objective code patching
    - Once a program is loaded, it cannot change its memory location

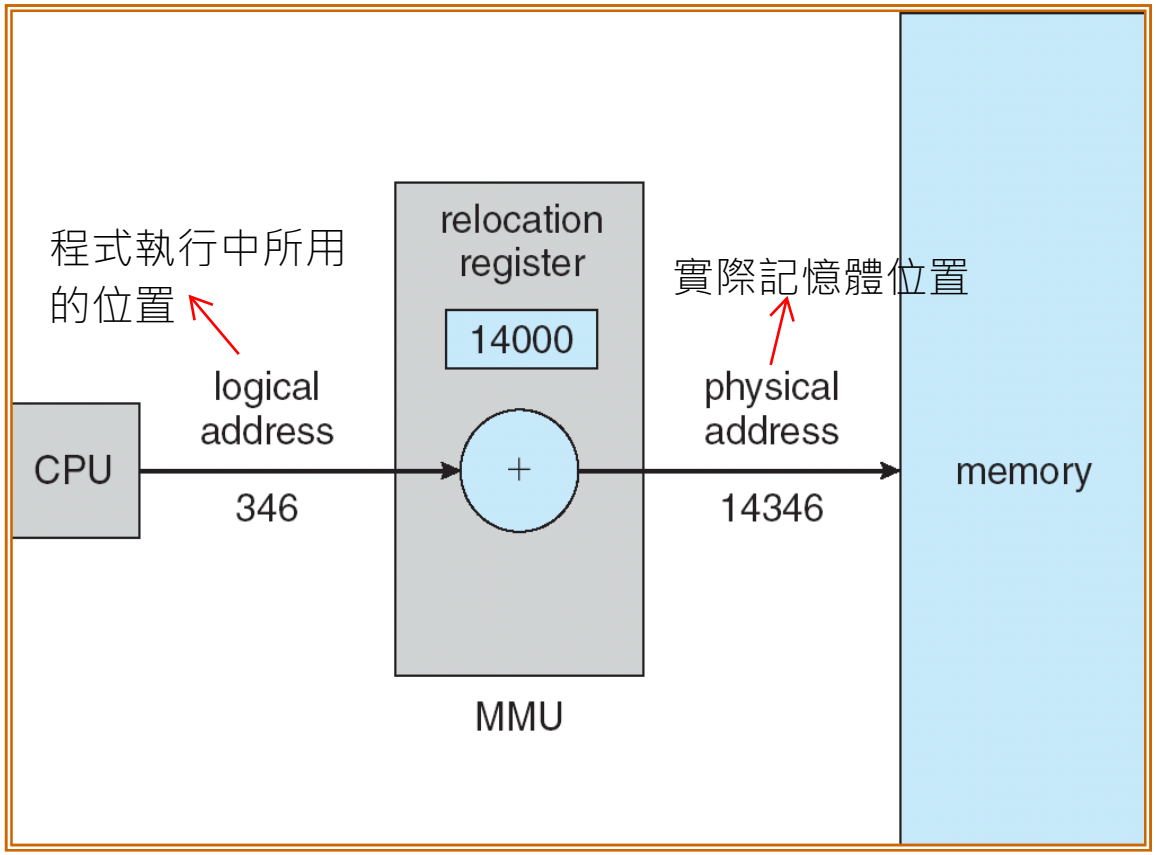
Binding of Instructions and Data to Memory

- Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another.

Execution Time Binding

- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
  - Logical address – generated by the CPU; also referred to as virtual address
  - Physical address – address seen by the memory unit
- The user program deals with logical addresses; it never sees the real physical addresses
- Need hardware support to translate logical addresses into physical addresses during runtime
  - Relocation registers (base/limit)

Execution Time Binding/Relocating using a Relocation Register



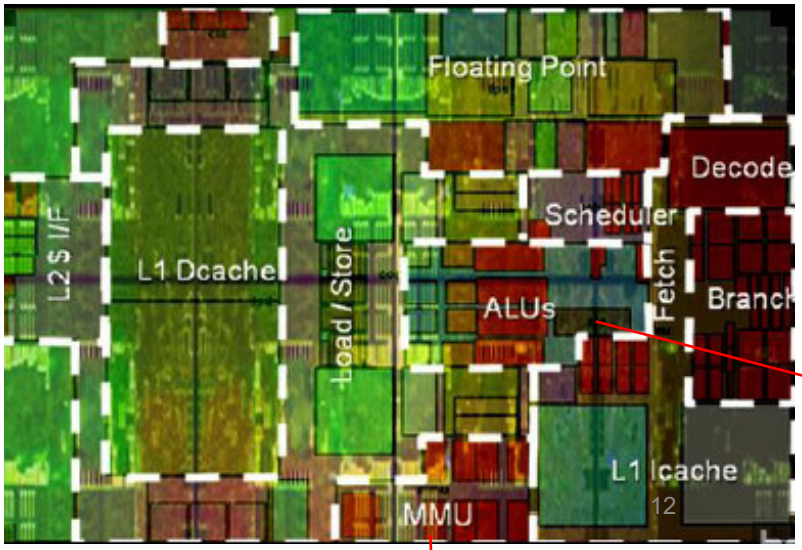
compiler在計算位置時都是從零開始算。
而MMU會記錄程式開頭在哪裡→邏輯位置加上紀錄就是實際位置

Memory-Management Unit (MMU)

- A hardware component in the CPU that translates logical addresses into physical addresses

- Paging
  - Address mapping
  - Virtual memory
- Segmentation
  - Memory protection
- To be explained later

Project Denver 64-bit CPU core



計算單元

記憶體管理單元

Dynamic Linking Loading (Execution Time Binding) .dll檔

如果程式有用到才會載入.dll，且多個process都能共用

- Both loading and linking are postponed until execution time
- Small piece of code, stub, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries

- By using dynamic linking, program modules have no information about where the needed routines are. The required routines may even not be anywhere (i.e., does not exist).
- Link is established by the names of procedures.
- For example, Windows DLLs

Dynamic Linking Loading: Windows DLL

- A program can call the procedures in a DLL file
  - Upon the first request, the corresponding DLL file is loaded to a runtime-determined memory address
  - Use address tables to handle dynamic linking
  - Subsequent calls use the address table to locate the DLL memory address

```
#include <windows.h>

// Export this function
extern "C" __declspec(dllexport) double AddNumbers(double a, double b);

// DLL initialization function
BOOL APIENTRY
DllMain(HANDLE hModule, DWORD dwReason, LPVOID lpReserved)
{
        return TRUE;
}

// Function that adds two numbers
double AddNumbers(double a, double b)
{
        return a + b;
}
```

如同main

The DLL source file

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

// DLL function signature
typedef double (*importFunction)(double, double);

int main(int argc, char **argv)
{
        importFunction addNumbers;
        double result;

        // Load DLL file
        HINSTANCE hinstLib = LoadLibrary("Example.dll");
        if (hinstLib == NULL) {
                printf("ERROR: unable to load DLL\n");
                return 1;
        }

        // Get function pointer
        addNumbers = (importFunction)GetProcAddress(hinstLib, "AddNumbers");
        if (addNumbers == NULL) {
                printf("ERROR: unable to find DLL function\n");
                return 1;
        }

        // Call function.
        result = addNumbers(1, 2);

        // Unload DLL file
        FreeLibrary(hinstLib);

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

        return 0;
}
```

載入dll

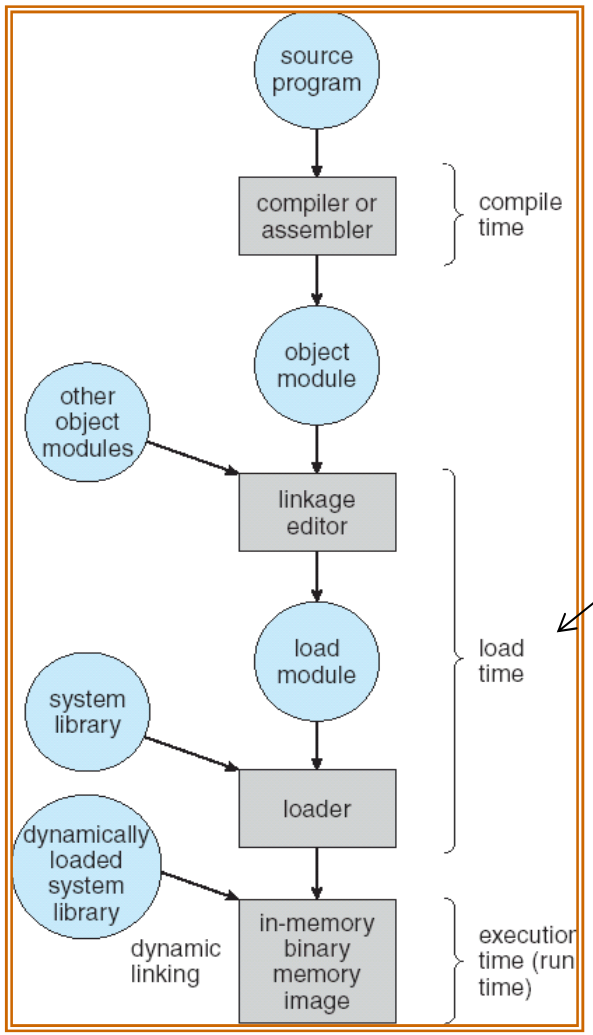
回傳function in dll 的address

此時就能像一般function一樣使用

程式會跑到dll裡面找function用

A program calls the DLL

Review: Address Binding Timing

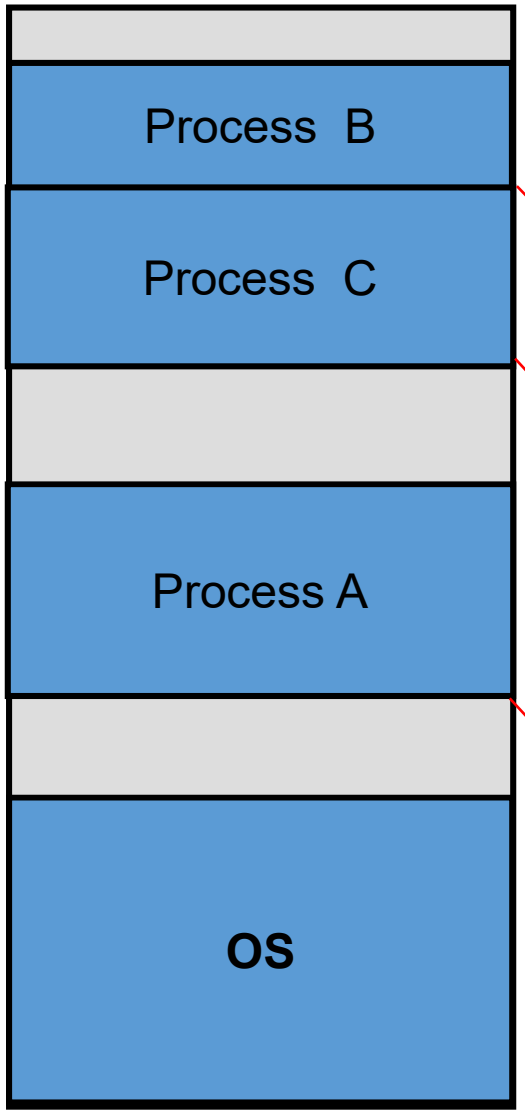


Absolute addressing

Relative addressing and object code patching

Dynamic linking loading

CONTIGOUS MEMORY ALLOCATION



每個process要的記憶體量都不一樣

- Process are allocated to contiguous memory space 因為程式都是由上而下執行
- Processes can be loaded into any contiguous and sufficiently large memory space
- As processes arrive and leave, free space may be fragmented into many pieces

Contiguous Allocation

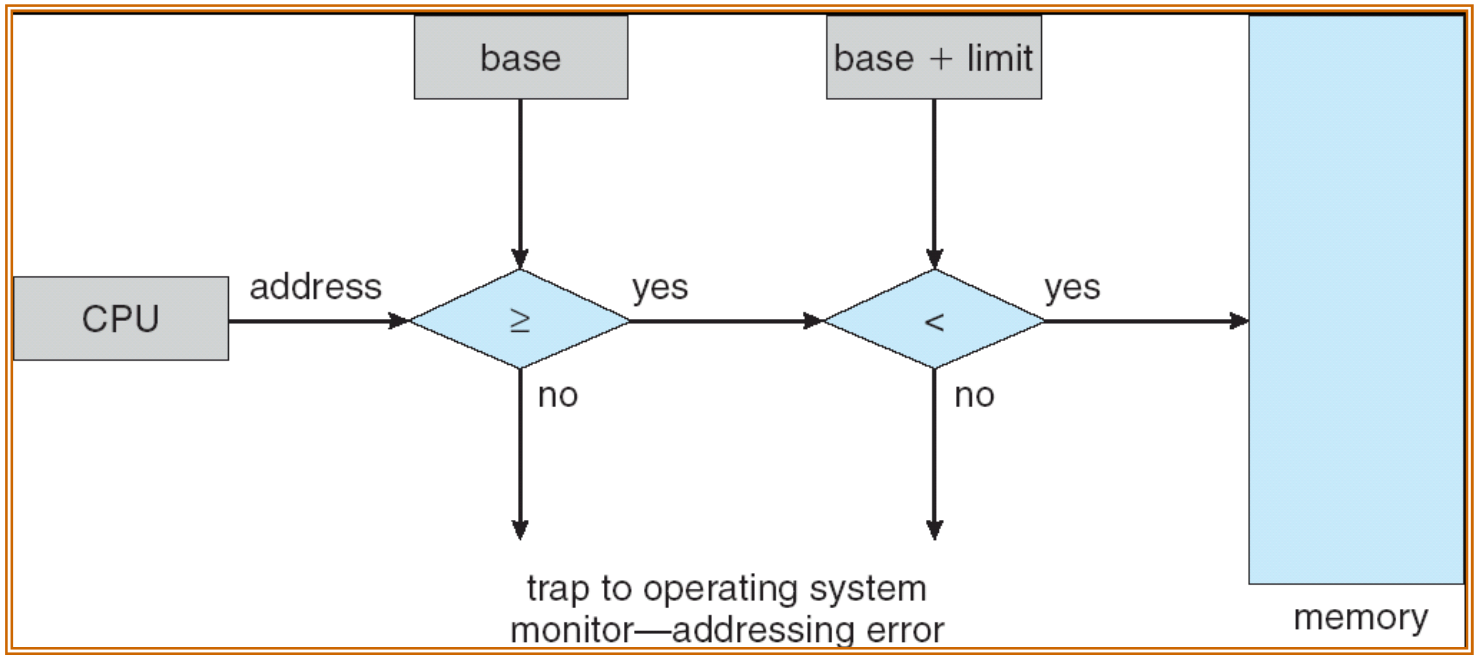
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory

- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address; limit register contains range of logical addresses – each logical address must be less than the limit register

Relocation register可以在OS要動態卸掉不需要的device時控制OS所使用的記憶體大小

limit register紀載了邏輯位址的範圍，來確保程式不會用到不屬於自己的記憶體

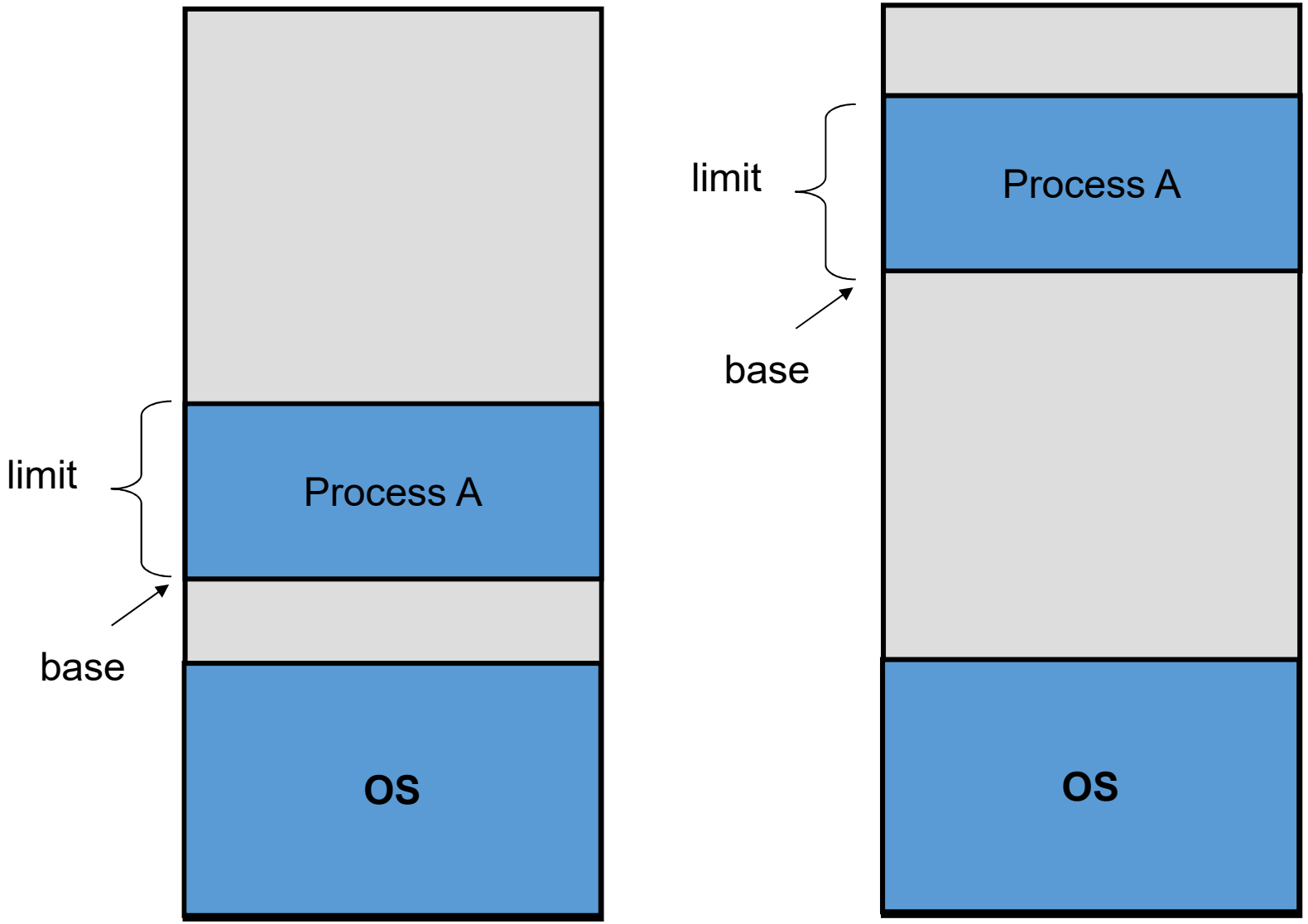
Memory Mapping and Protection



>=

CPU -(LA)->Compare_with_Limit -(yes)-> Plus base

-(no)-> illegal address:Trap


limit
Process A
base
OS
limit
Process A
base
OS

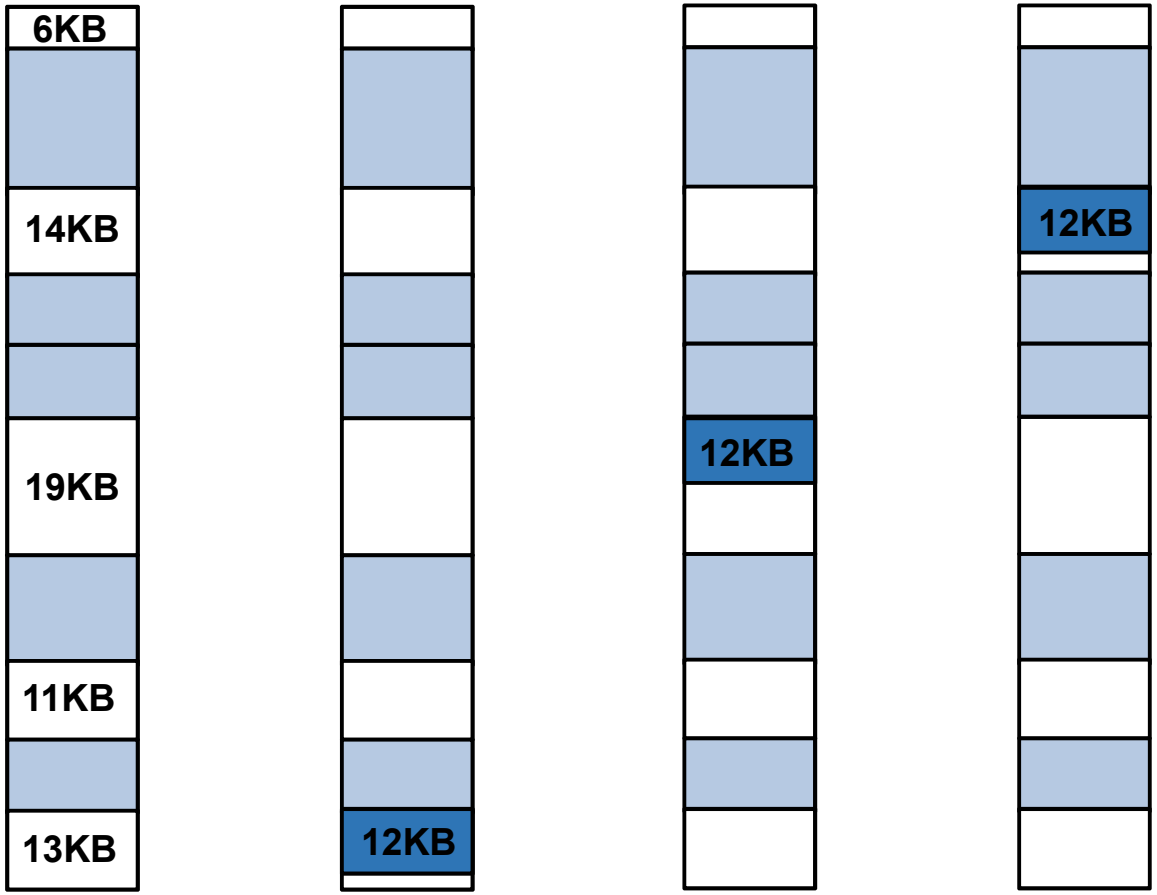
Memory Allocation 給process的記憶體空間一定要連續!!

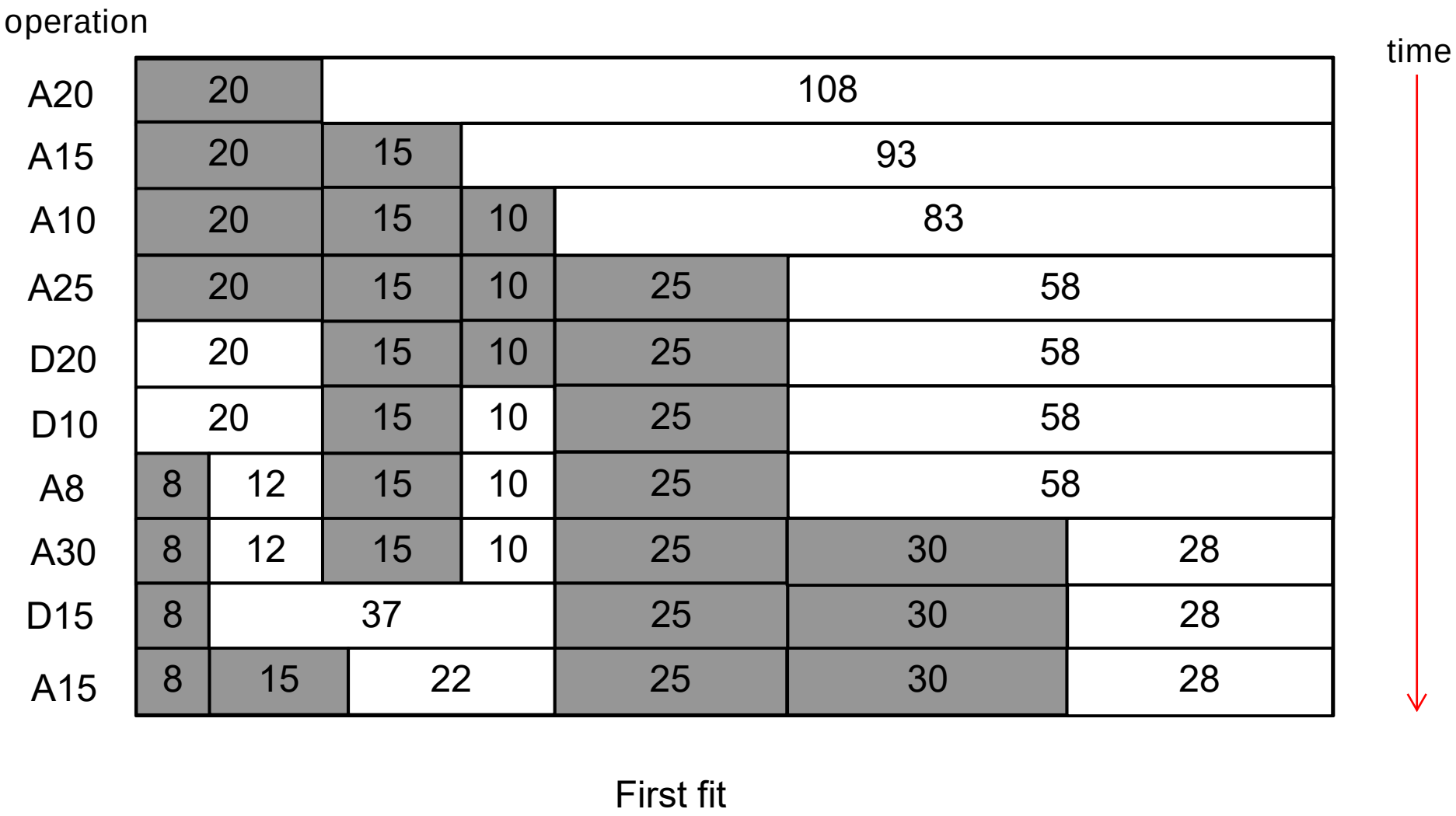
How to satisfy a request of size *n* from a list of free holes

有時候因為記憶體空間破碎，會導致明明有空間，卻找不到合適的地方

- First-fit: Allocate the *first* hole that is big enough
- Best-fit: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- Worst-fit: Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.

First-fit and best-fit better than worst-fit in terms of speed and storage utilization


6KB
14KB
19KB
11KB
13KB
12KB
12KB
12KB
Best fit
Worst fit
First fit


operation
time
A20
A15
A10
A25
D20
D10
A8
A30
D15
A15
20
108
20
15
93
20
15
10
83
20
15
10
25
58
20
15
10
25
58
20
15
10
25
58
8
12
15
10
25
58
8
12
15
10
25
30
28
8
37
25
30
28
8
15
22
25
30
28

First fit

盡量維持一個大的連續記憶體，但是allocate之後的碎片都很小(很難再利用)

D10	20		15		10		25	58	
A8	20		15		8	2	25	58	
A30	20		15		8	2	25	30	28
D15	35				8	2	25	30	28
A15	35				8	2	25	30	15 13

Best fit

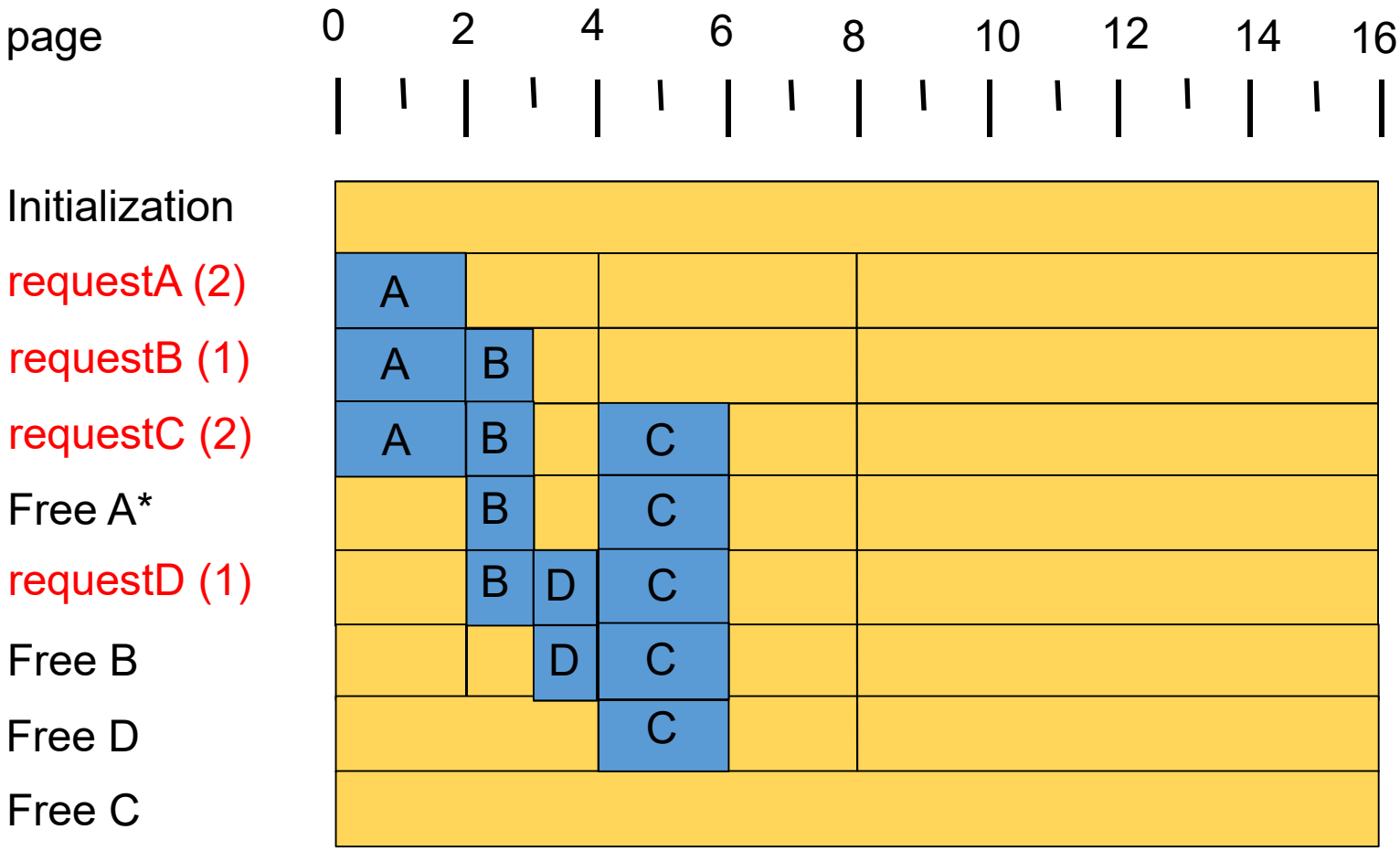
D10	20		15		10		25	58	
A8	20		15		10		25	8	50
A30	20		15		10		25	8	30 20
D15	45						25	8	30 20
A15	15		30				25	8	30 20

盡量讓allocate之後的碎片大一點，通常最後會留下很多大小差不多的洞

Worst fit

整個Memory Allocate的問題是NP complete

Buddy System



速度很快，而且破碎化不見得比前面的多

都以2^n給記憶體

Fragmentation

只要不是完全連續的狀況，都稱為有External Fragmentation

- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used 像buddy system就可能給超過，但worst, first, best都沒有這種狀況

PAGING

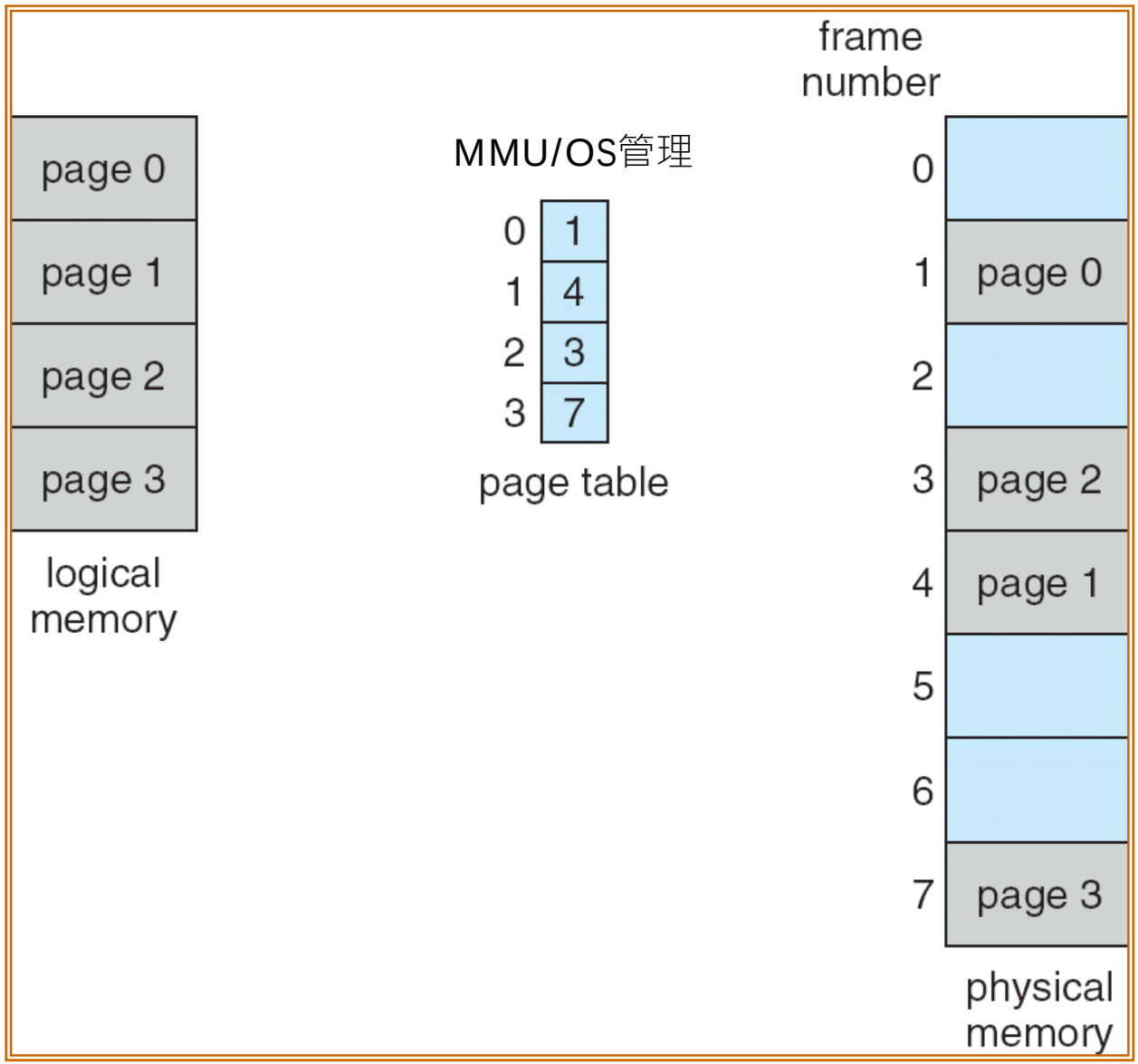
External Fragmentation

- Compaction 現在大多不採用
  - Migrate allocated memory chunks together to make free space contiguous
  - Will relocate programs– need execution time binding
  - Occasionally slows down the system
- A better approach: What if memory can be logically continuous but not physically continuous?

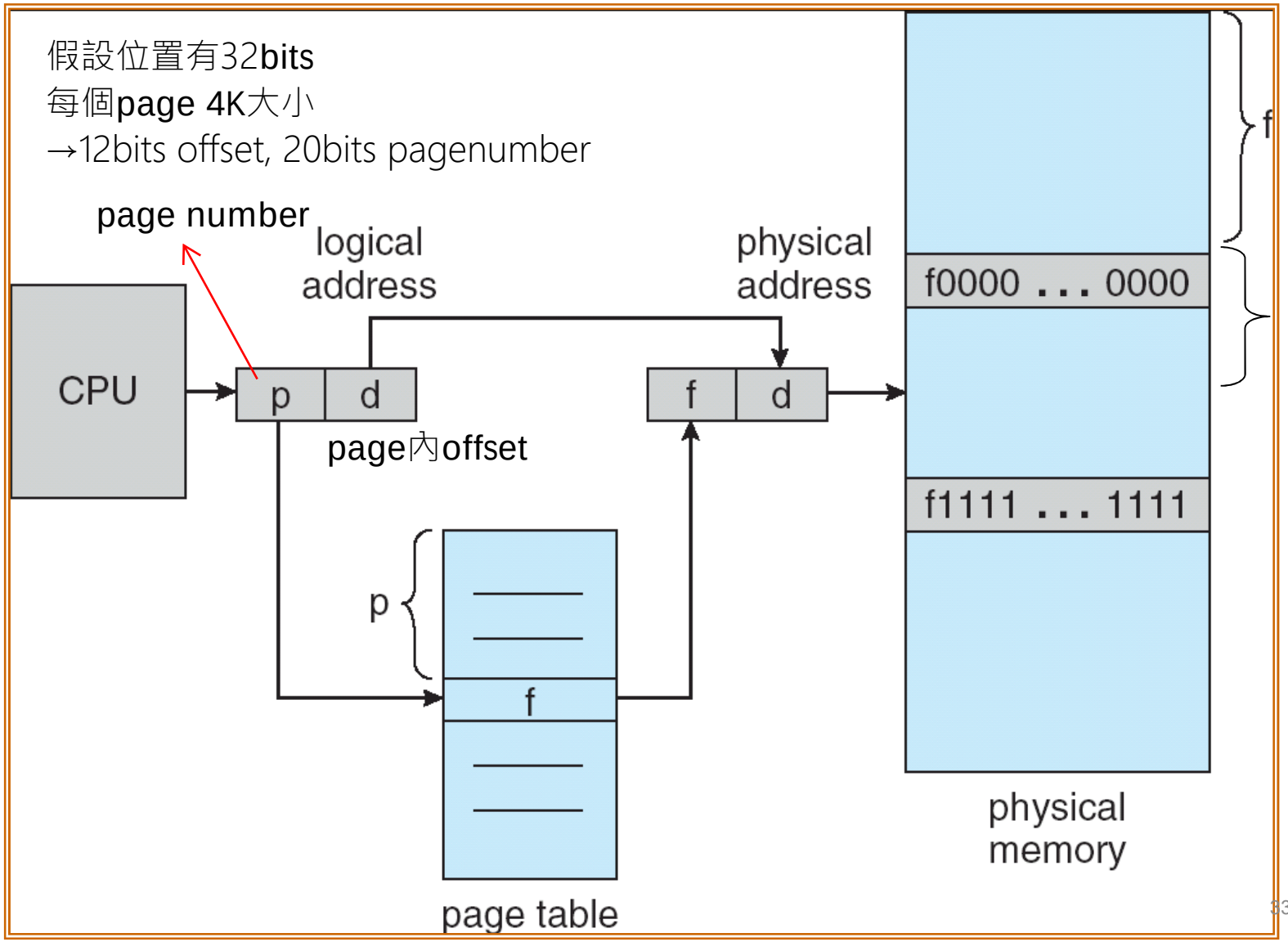
Paging

- Divide physical memory into fixed-sized blocks called frames (size is power of 2, between 512 bytes and 8192 bytes) 通常4K最多
- Divide logical memory into blocks of same size called pages. pages與frames大小會一致→比較好mapping
- Keep track of all free frames
- To run a program of size *n* pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses

Paging Model of logical and physical memory

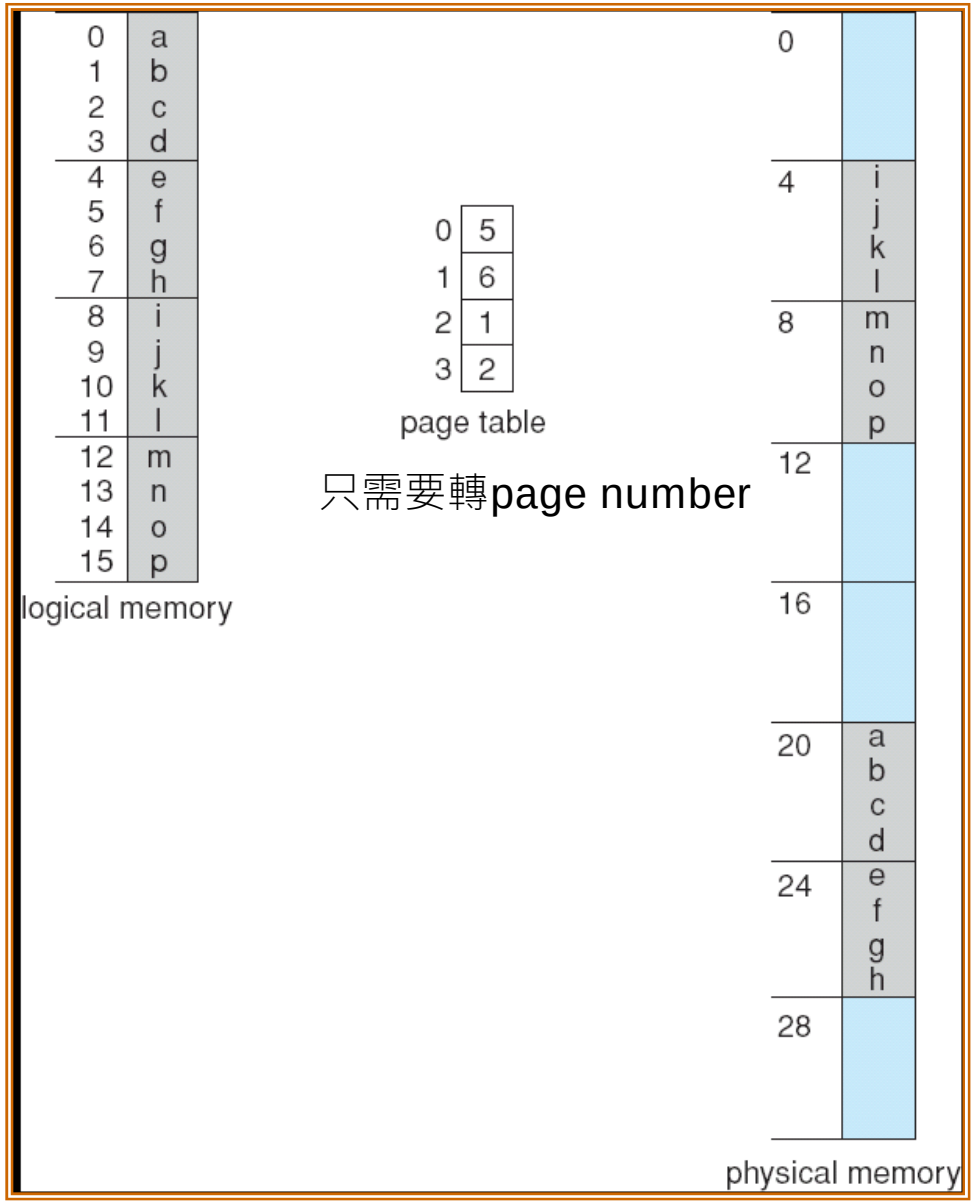


Paging hardware



OS會設定好table/MMU負責轉換

Paging Example

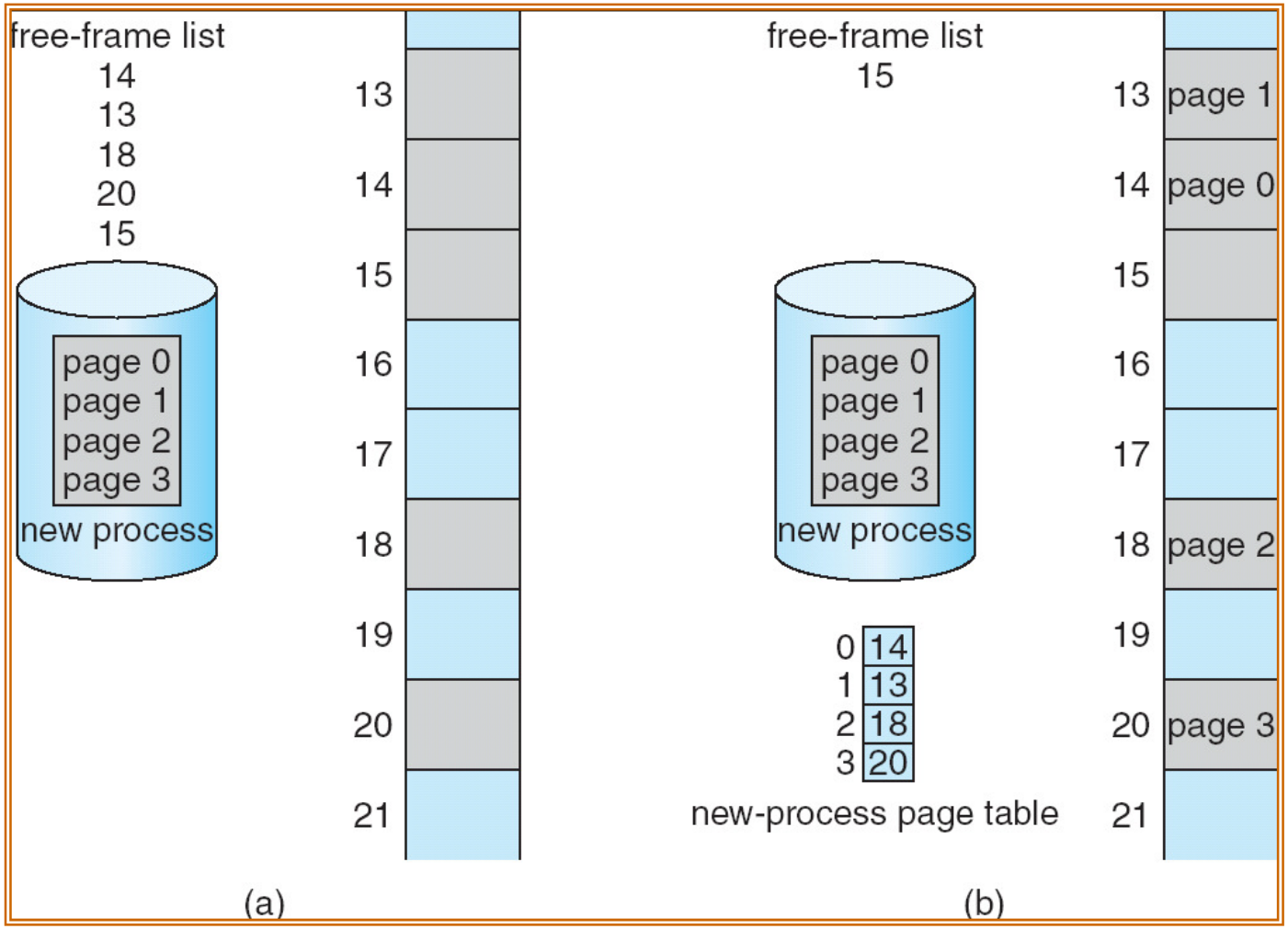


Q: how many bits are required to represent

1. The page number 2
2. The frame number 3
3. The displacement 2 (offset)
4. A logical address 4 page#+displace
5. A physical address 5 frame#+displa
6. A page-table entry? 3 frame#

比較常見的實作是
20bits page number
12bits offset
整個page table有2^20 * (20+function bits)

Free Frames



Before allocation

After allocation

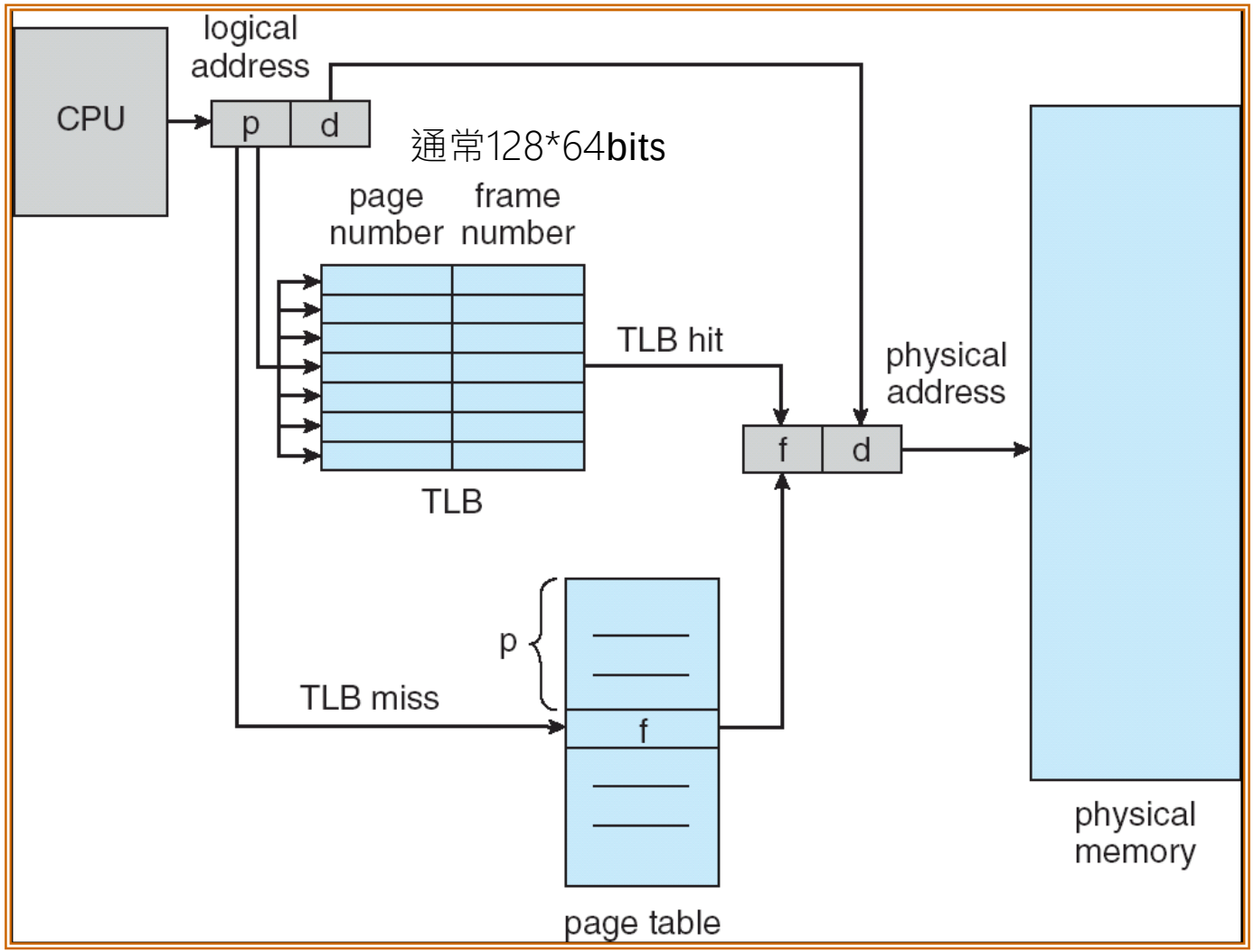
Implementation of Page Table

每個processes都有自己的Page Table

- Page table is kept in main memory
- A table per process
  - Page-table base register (PTBR) points to the page table
  - Page-table length register (PRLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)

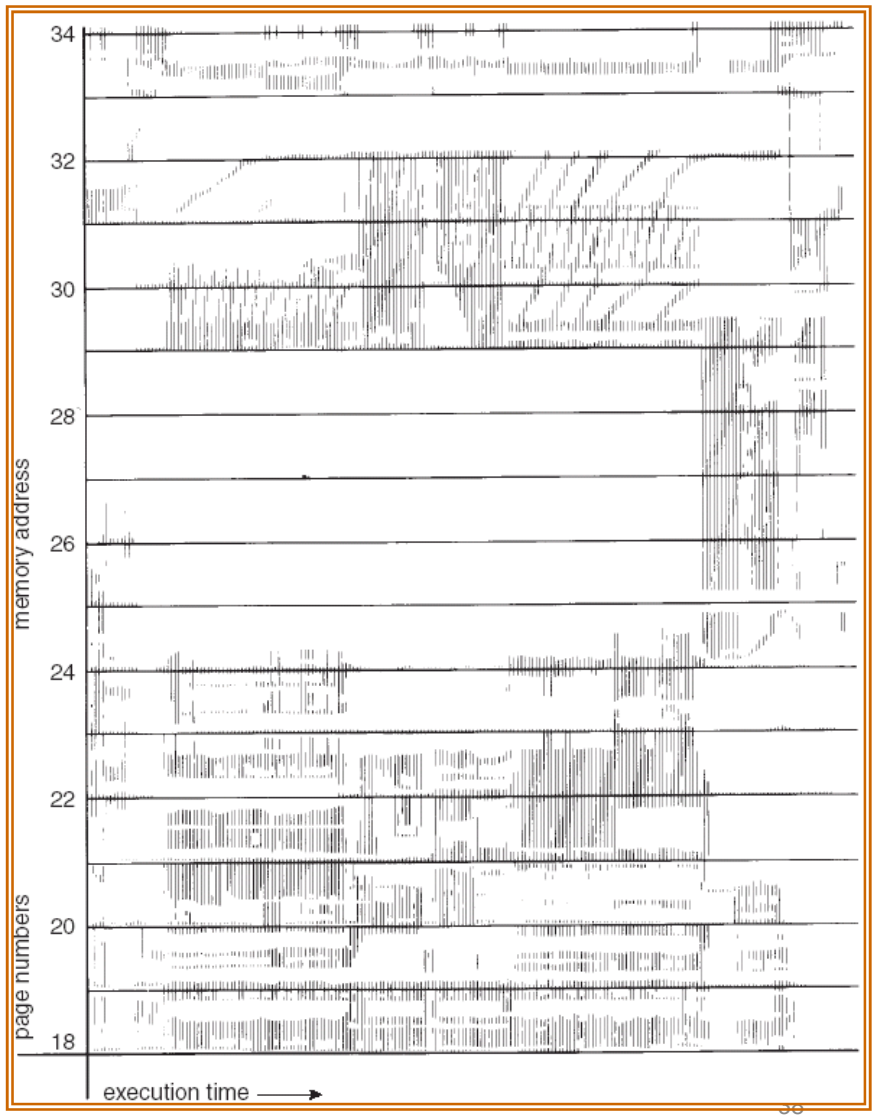
TLBs是page table的快取，是一塊特殊的cache(full-association)。page table則是放在main memory中

Paging Hardware With TLB



TLB hit ratio vs. Locality of Reference

- TLB is small, usually holds 64~1024 entries
  - A replacement policy should be used.
  - E.g., random policy or LRU
  - Page references have temporal localities and spatial localities
    時間與空間區域性→MM access 會有一定趨勢
- Important entries can be wired down (nailed)
  - E.g., kernel code



Effective Access Time

- Associative Lookup = ε time unit 查TLB要多久
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- Effective Access Time (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$

↘ 很小

The overhead of updating page table is ignored

Let the TLB hit ratio be 98%

20ns to lookup the TLB
100 ns to access memory

TLB hit: 20+100

TLB miss: 20 + 100 (page table) + 100 (target address)

EAT= 0.98*120+0.02*220=122 ns

其實很差
實際上能達到0.9999...

The overhead of updating page table is ignored here

STRUCTURE OF THE PAGE TABLE

Page table得是連續的，不然會為了查page table再去查page table...

Structure of Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

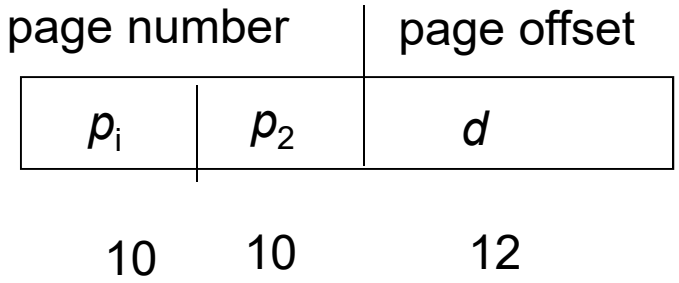
Hierarchical Page Tables

好處：整張page table不用連續(小張的連續就好了)
沒有真的用到的地方就不用先配好(節省空間)

- A per-process page table could be very large and sparse
- Allocating Large and contiguous page tables for processes is inconvenient and may under-utilize memory space (of page tables)
- Breaking up the logical address space into multiple page tables
  - Page table can be fragmented
  - Page table pieces are allocated on demand
- A simple technique is a two-level page table
  - Example: Intel 80386

Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit outer-page number
  - a 10-bit displacement of the outer page
- Thus, a logical address is as follows:

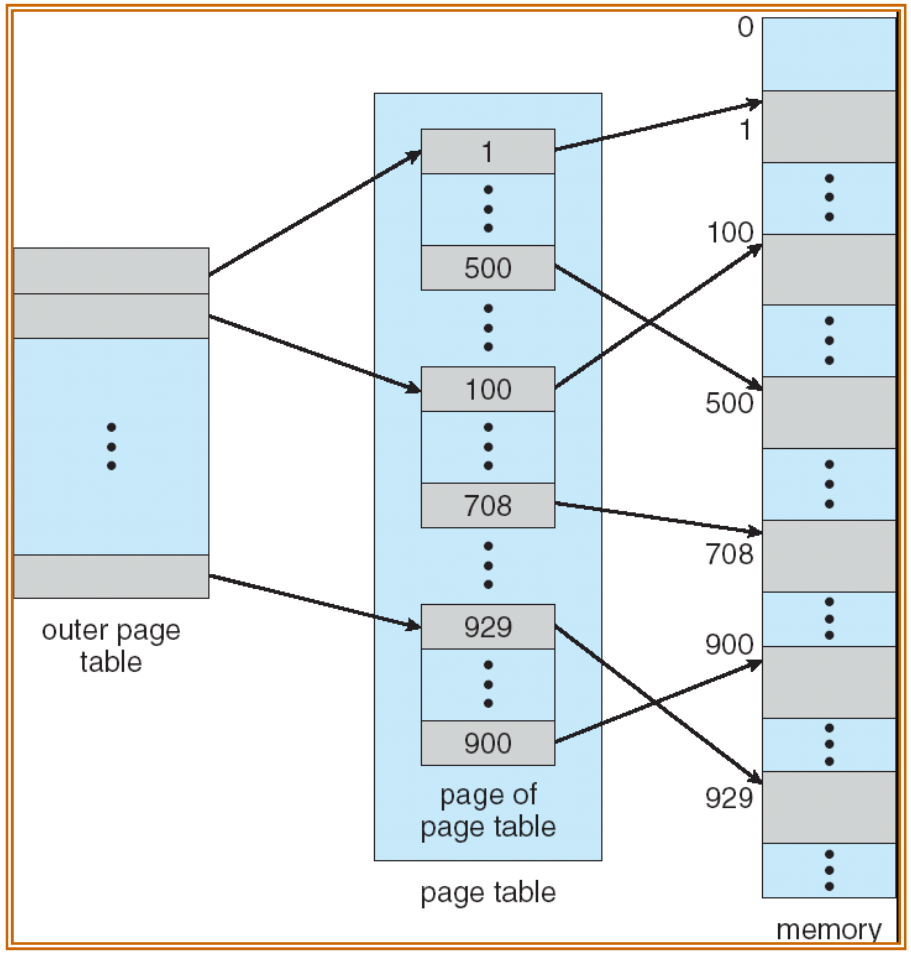


2^{10}個小page table，每個小page table有2^{10}個entry

where pi is an index into the outer page table, and p2 is the displacement within the page of the outer page table

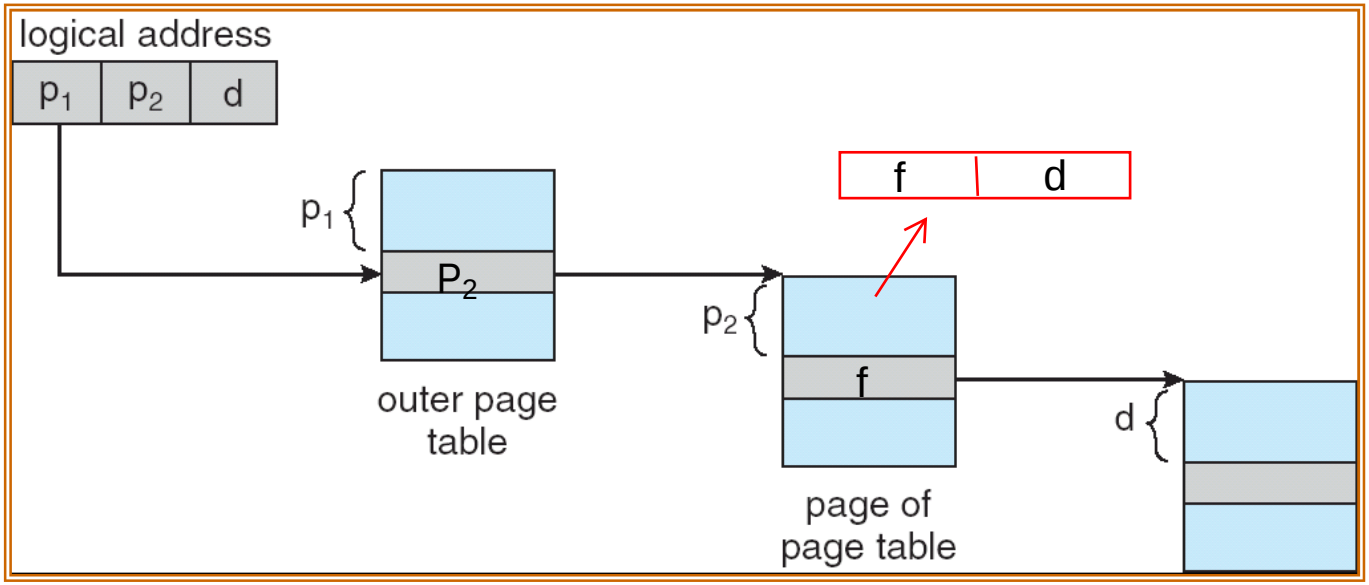
Two-Level Page-Table Scheme

[p1][p2][d]



Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

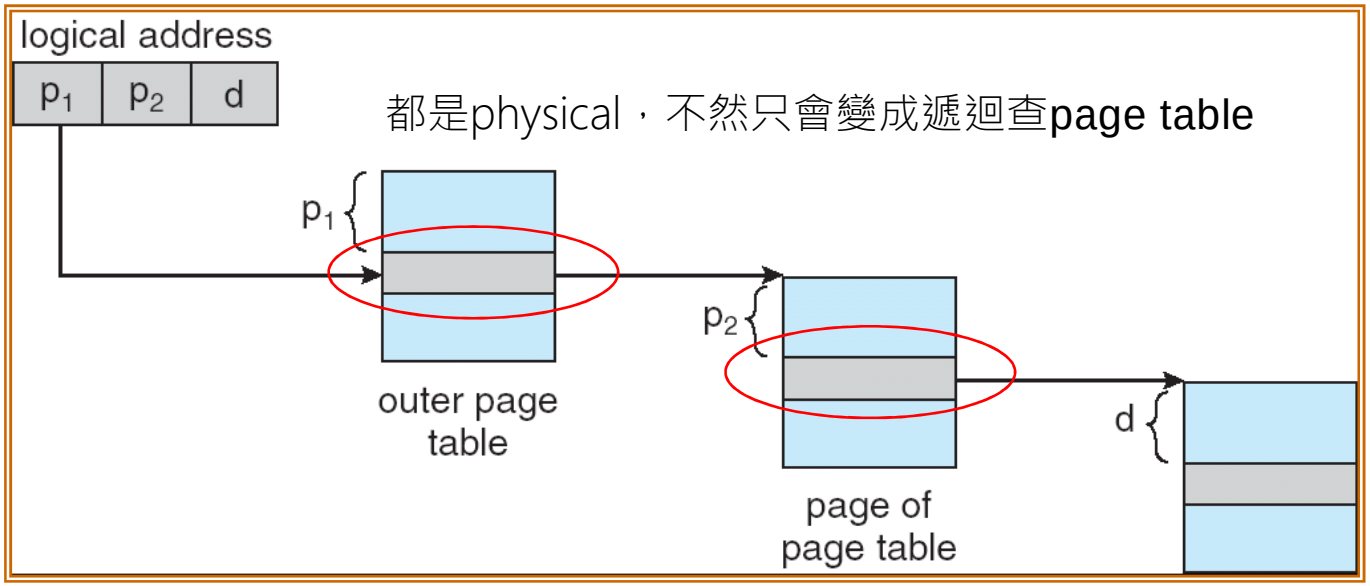


Pros: page tables need not to be contiguous, and need not all present in memory
Cons: multiple memory accesses on TLB miss. 7-level paging in UltraSparc 64

因為邏輯空間實在太大了
→TLB hit ratio一定要很高，不然很可怕

Think about it…

- Logical address or physical address?

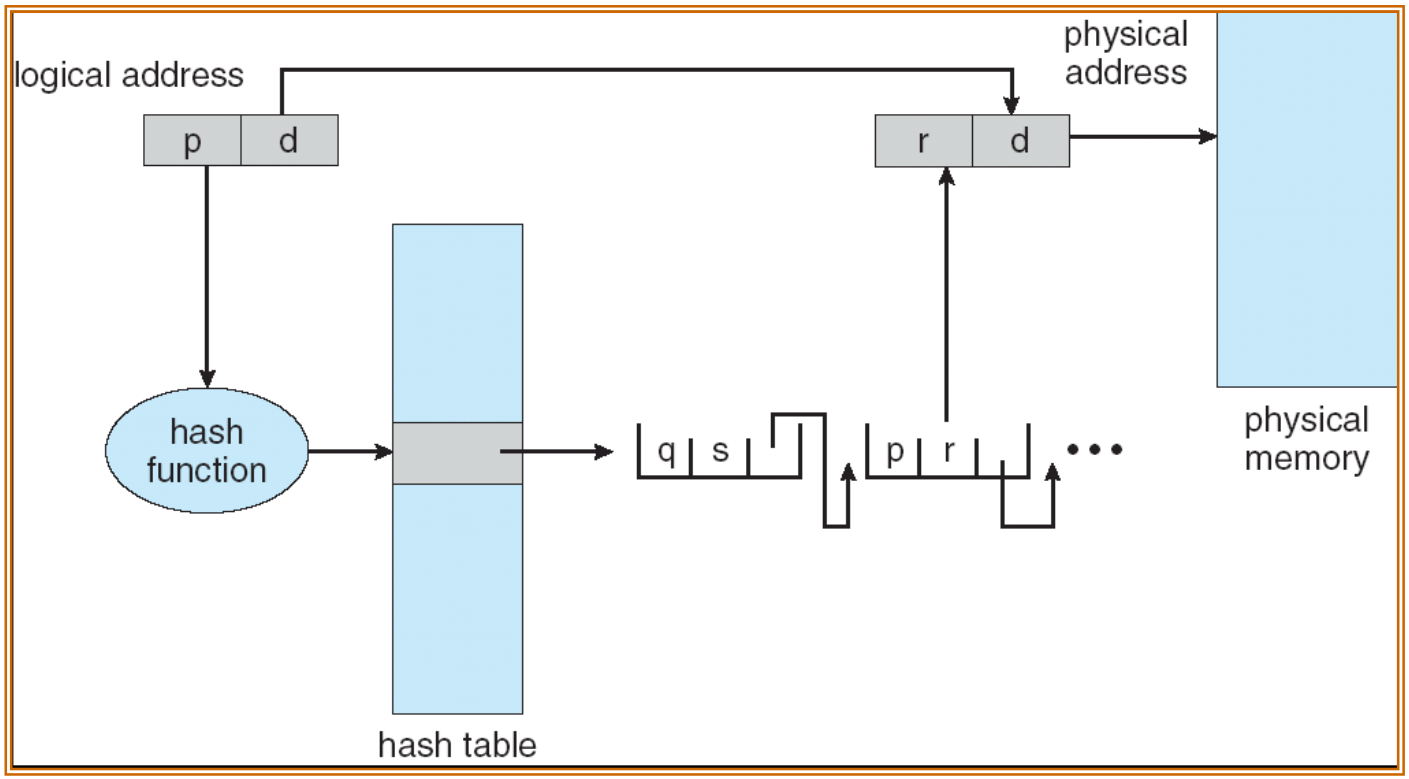


Hashed Page Tables

如果記憶體使用很稀疏，就很適合用這種實作
尤其是對64位元處理器(邏輯空間實在太大)

- Replace the radix-based multi-level table with a hash table
- Common in address spaces > 32 bits
  - The logical address footprint of a process << the logical address space 真正使用的意義體遠小於邏輯記憶體空間
- The page number is hashed into a page table. This page table contains a chain of elements hashing to the same location
- Page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted
- Example: 64-bit PowerPC CPUs

Hashed Page Table



How many extra memory references are needed on TLB miss is related to the quality of the hash function

Inverted Page Table

- Consider that there can be up to 16K processes, and the logical addressing space of each process can be of 220 pages. If each entry is of 4B, the total size of page tables is: …… very large
- The size of an inverted page table is bounded by the size of physical memory and is irrelevant to # of processes

無關的

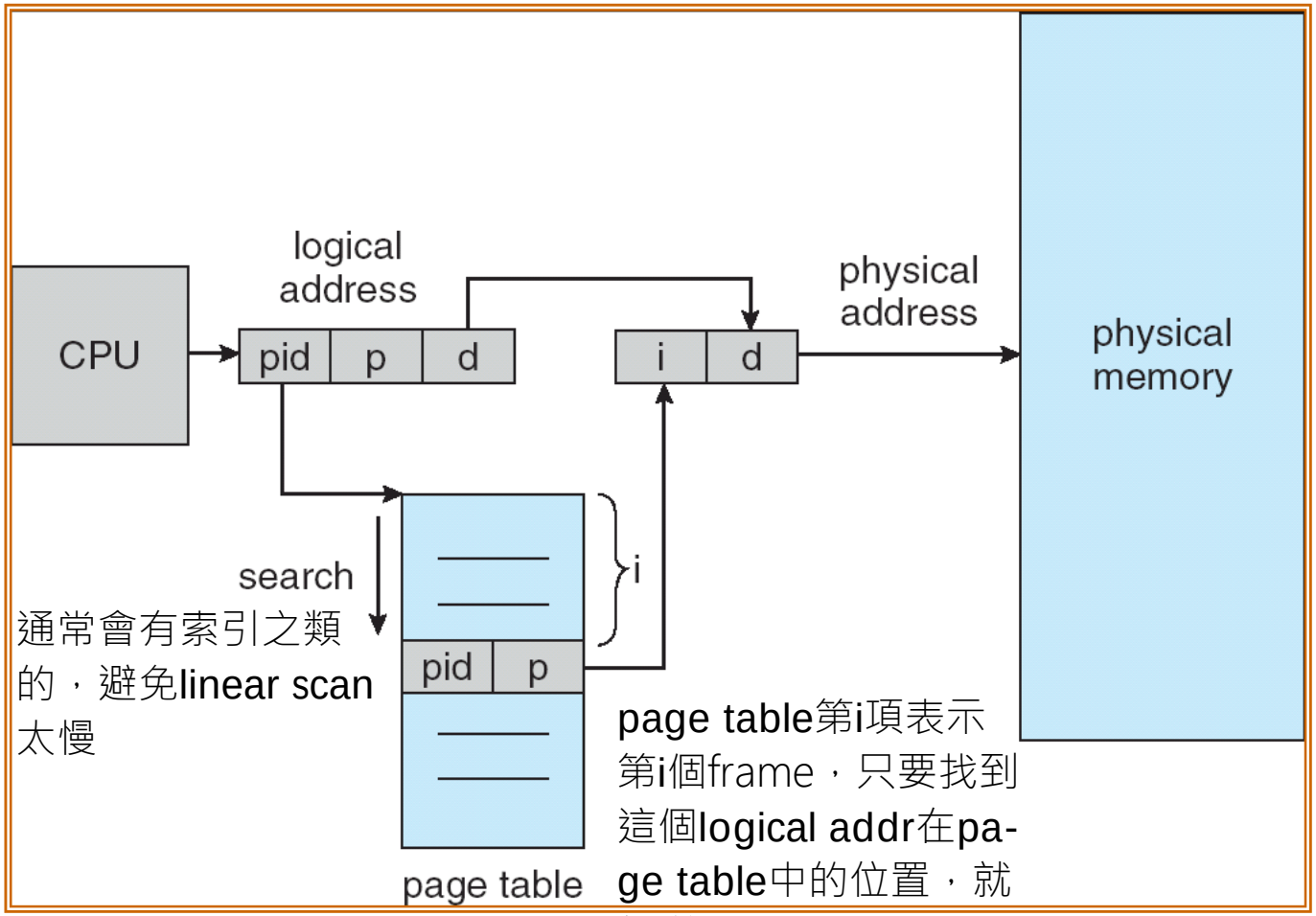
邏輯空間的總量大小會隨著processes數量而改變
但實體空間還是一樣

inverted page table的大小與實際記憶體大小有關

Inverted Page Table

- One entry for each real page (frame) of memory
  - Entry index number is the frame number
- Each table entry stores a page number and the process ID owns that owns the page
- One global page table shared among all processes
- On memory reference, find the table entry that stores the current process ID and the referenced page number
  - Use hash table to limit the search to one — or at most a few — page-table entries
- Example: PowerPC 603

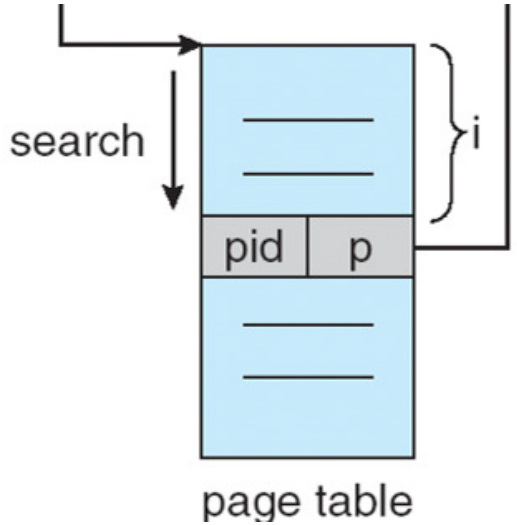
Inverted Page Table Architecture



Think about it…

- Why pid is necessary in inverted page tables?

不同的process間可能有相同的page number



- Isn’t an inverted page table like a TLB outside of the CPU?

Design Considerations of Paging

- Increased memory references
  - Solution: Use TLB
- Space requirement of the page table (in main memory)
- Multilevel page tables
  - Break the entire table into small pieces, allocate on demand
- Hash page tables 使用的space << logical space時適合
  - Small hash table with overflow handling (linked lists)
- Inverted page tables
  - Table size is bounded by the physical memory size and is independent of the total number of active processes

SEGMENTATION

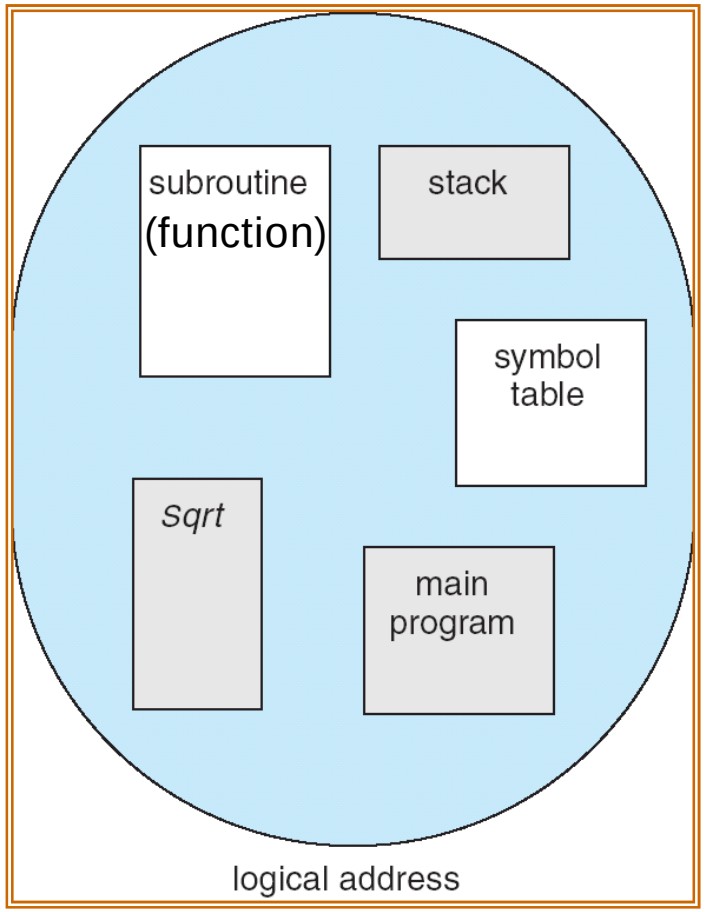
Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:

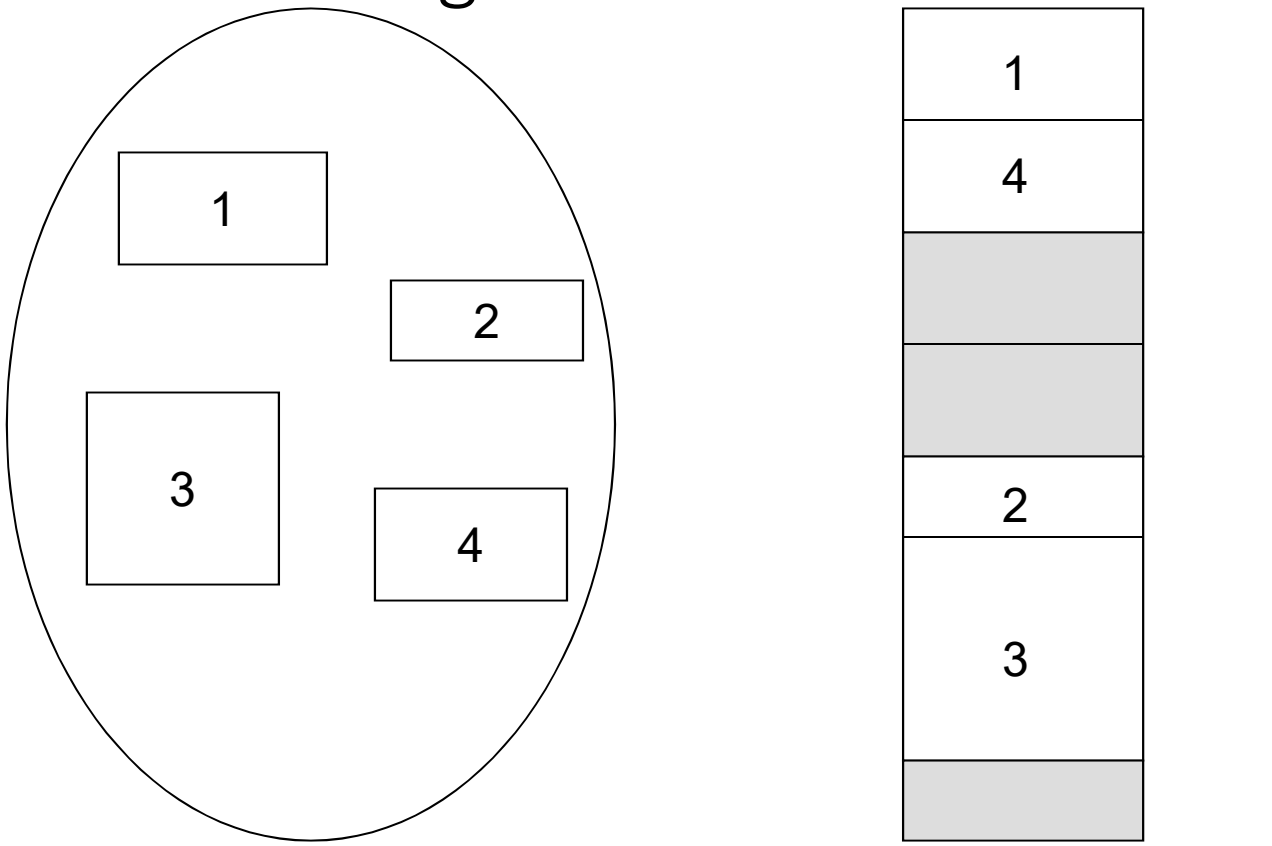
從人的視角去區分記憶體

main program,
procedure,
function,
method,
object,
local variables, global variables,
common block,
stack,
symbol table, arrays

User's View of a Program



Logical View of Segmentation



User programs do not know where in physical memory a segment is placed.

User programs不需要擔心在實體記憶體中的位置分配

Hardware

- Logical address consists of a two tuple:

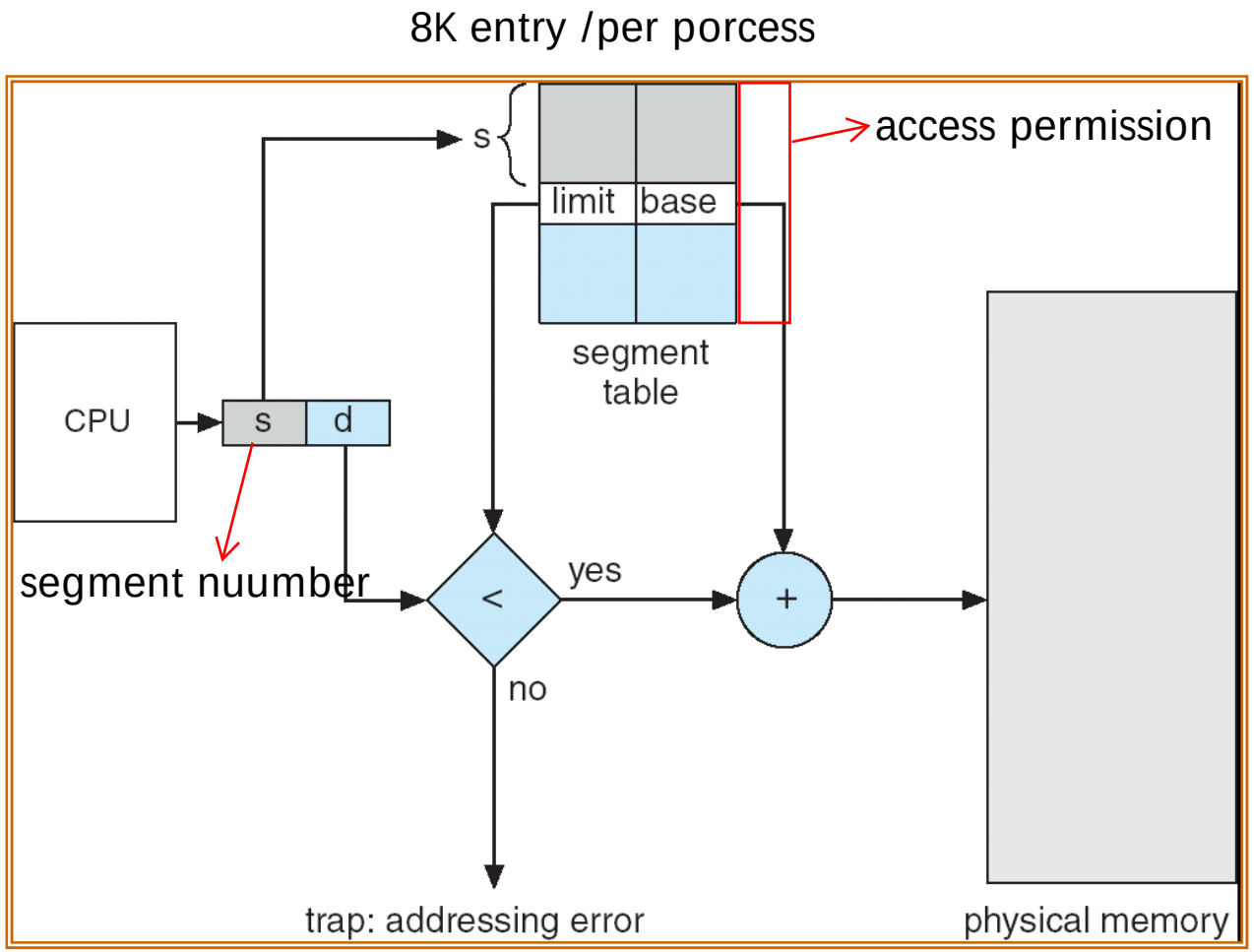
  <segment-number, offset>,

- Segment table – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - limit – specifies the length of the segment
- Segment-table base register (STBR) points to the segment table's location in memory
- Segment-table length register (STLR) indicates number of segments used by a program;

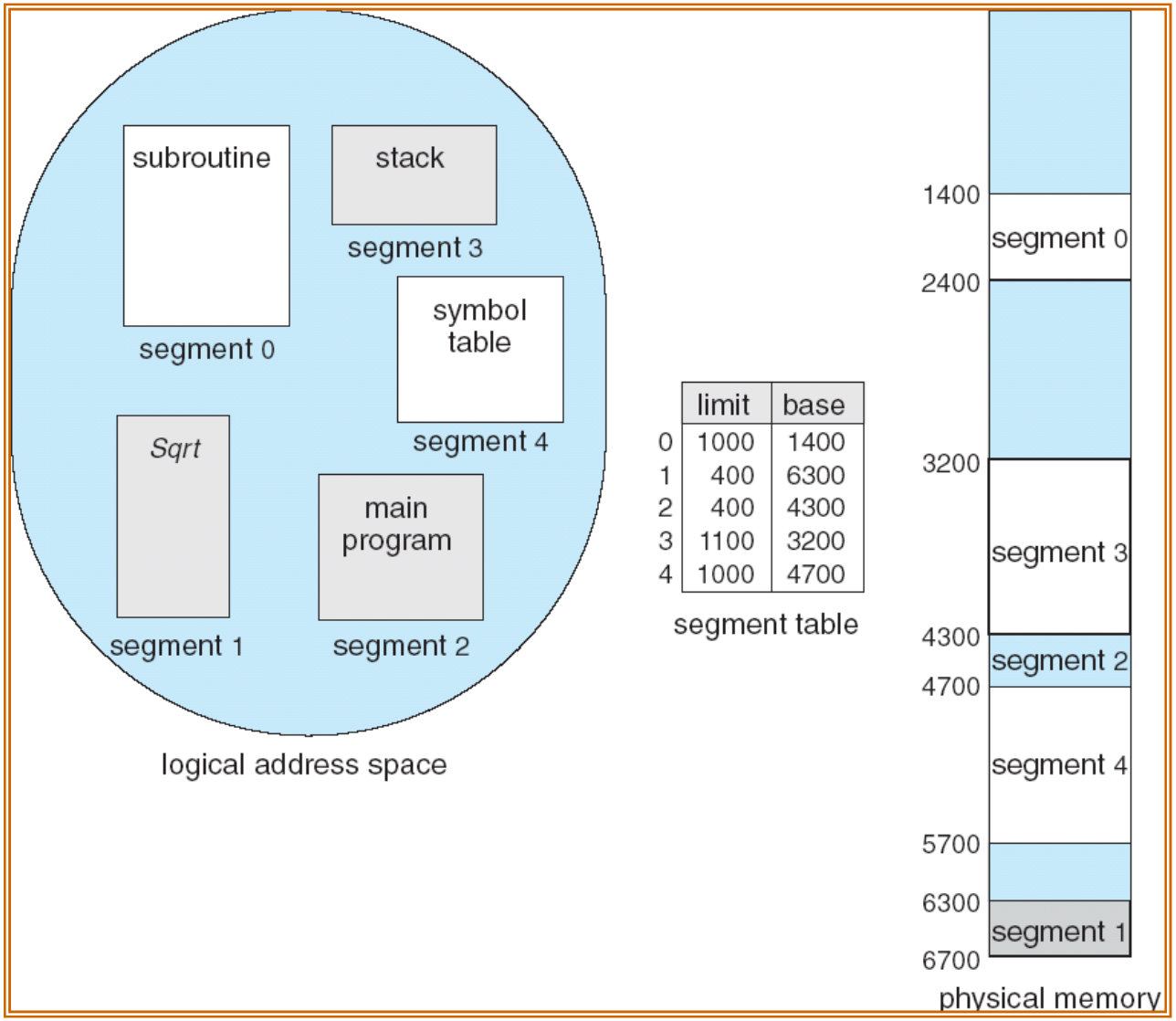
  segment number s is legal if s < STLR

Segmentation hardware

也會有類似TLB的東西來cache segmentation table



Example of Segmentation



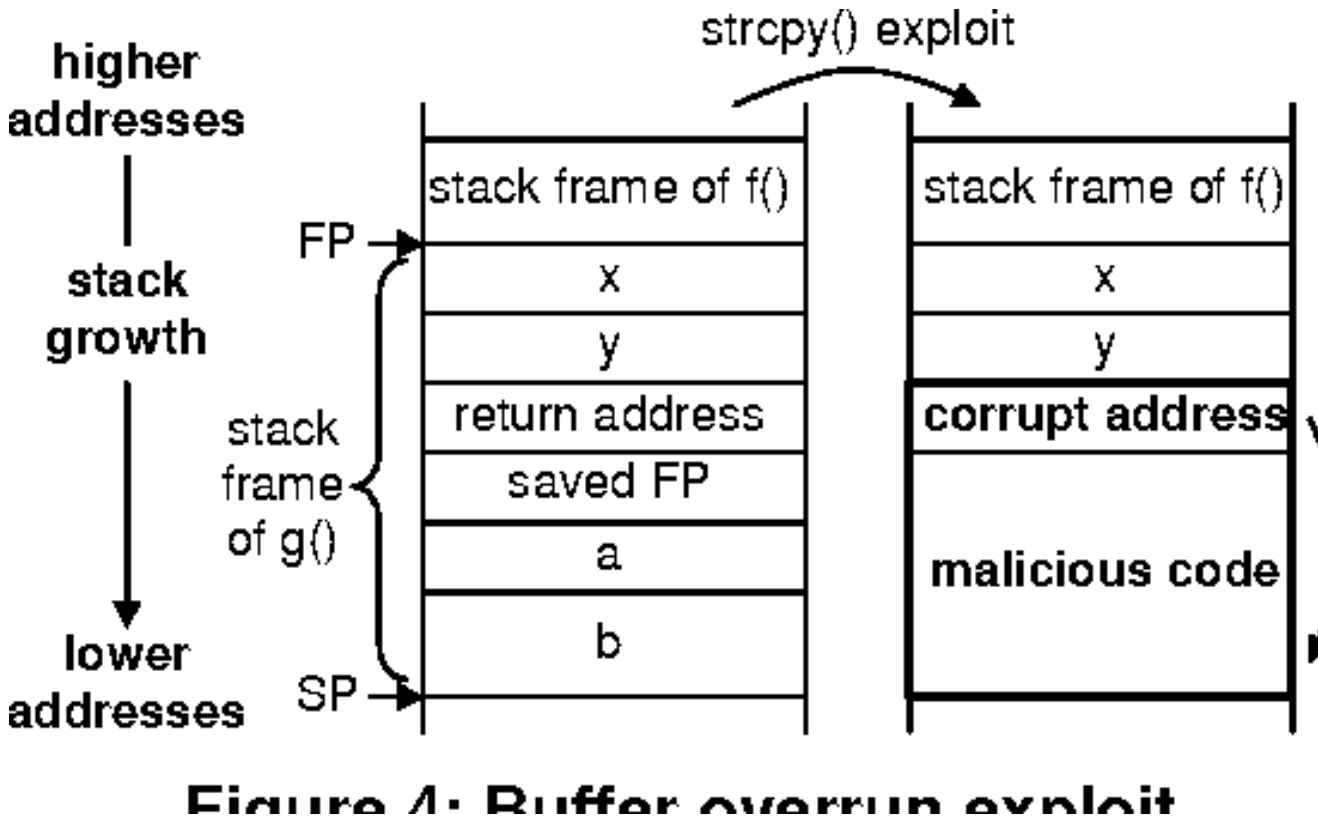
Segment Protection

基本上正經軟體都不會自我修改，除了少數例外

- Segments storing execution code normally do not allow modification
  - Allow read, execute but not write
  - Avoid self-modifying malicious code, such as polymorphic viruses
- Stack segments normally do not allow execution
  - Allow read, write but not execute
  - Avoid malicious code injection

```
g(x, y){
  a, b
}
```

Stack Overrun Attack



可以拿到y的位置
那把y的位置之後的東西改寫，reutrn address被改成跑到惡意程式碼那邊。
又由於stack可執行，惡意碼就被執行了。

Figure 4: Buffer overrun exploit

Fiskiran, A. Murat and Ruby B. Lee. "Runtime execution monitoring (REM) to detect and prevent malicious code execution." *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings.* (2004): 452-457.

Hardware Support for Segmentation

- Provided by MMU
- Segment Limit
  - Segment base register
  - Segment limit register
- Segment Protection
  - With each entry in segment table associate:
  - Permissions of read, write, and/or execute
  - Access privileges of user mode or kernel mode

Comparison

- Paging solves the external *fragmentation* problem at the process level
- Segmentation provides memory *protection* according to the purposes (r,w,x) of memory regions

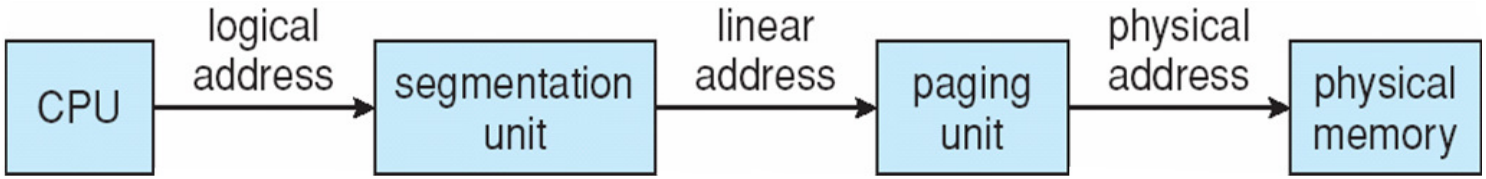
- Segmented paging (more common)
- Paged segmentation

EXAMPLE: THE INTEL 80386+

Segmented Paging

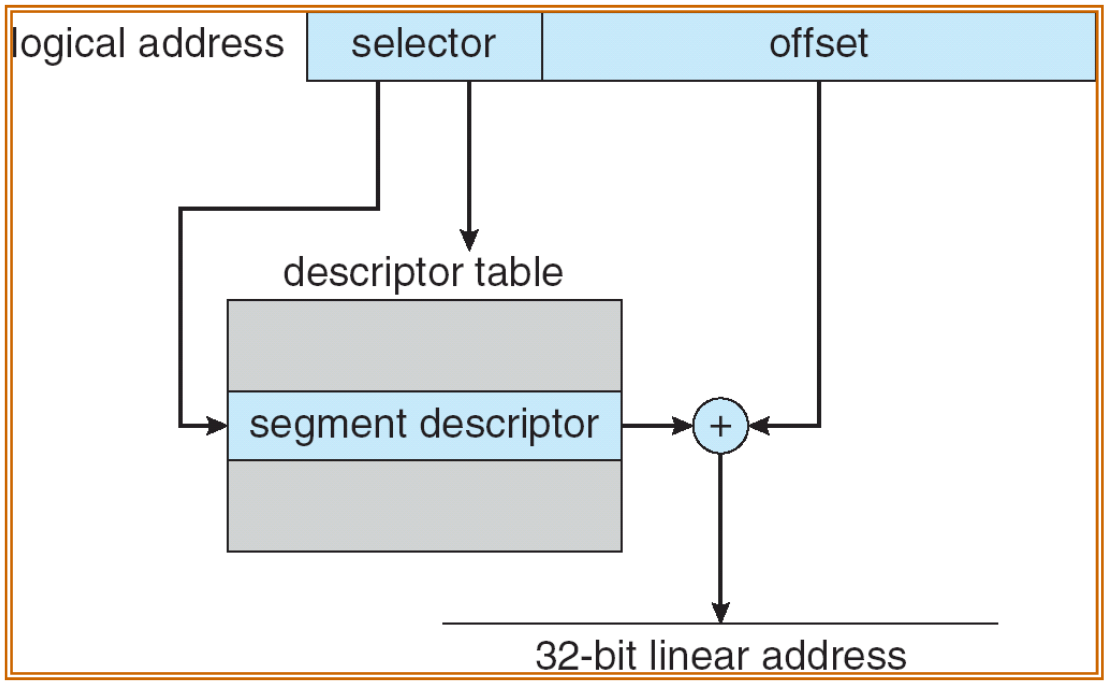
- Intel 80386
  - Segmentation
  - 2-level paging

The CPU generates logical addresses, which are given to the segmentation unit. The segmentation unit produces a linear address for each logical address. The linear address is then given to the paging unit, which in turn generates the physical address in main memory.



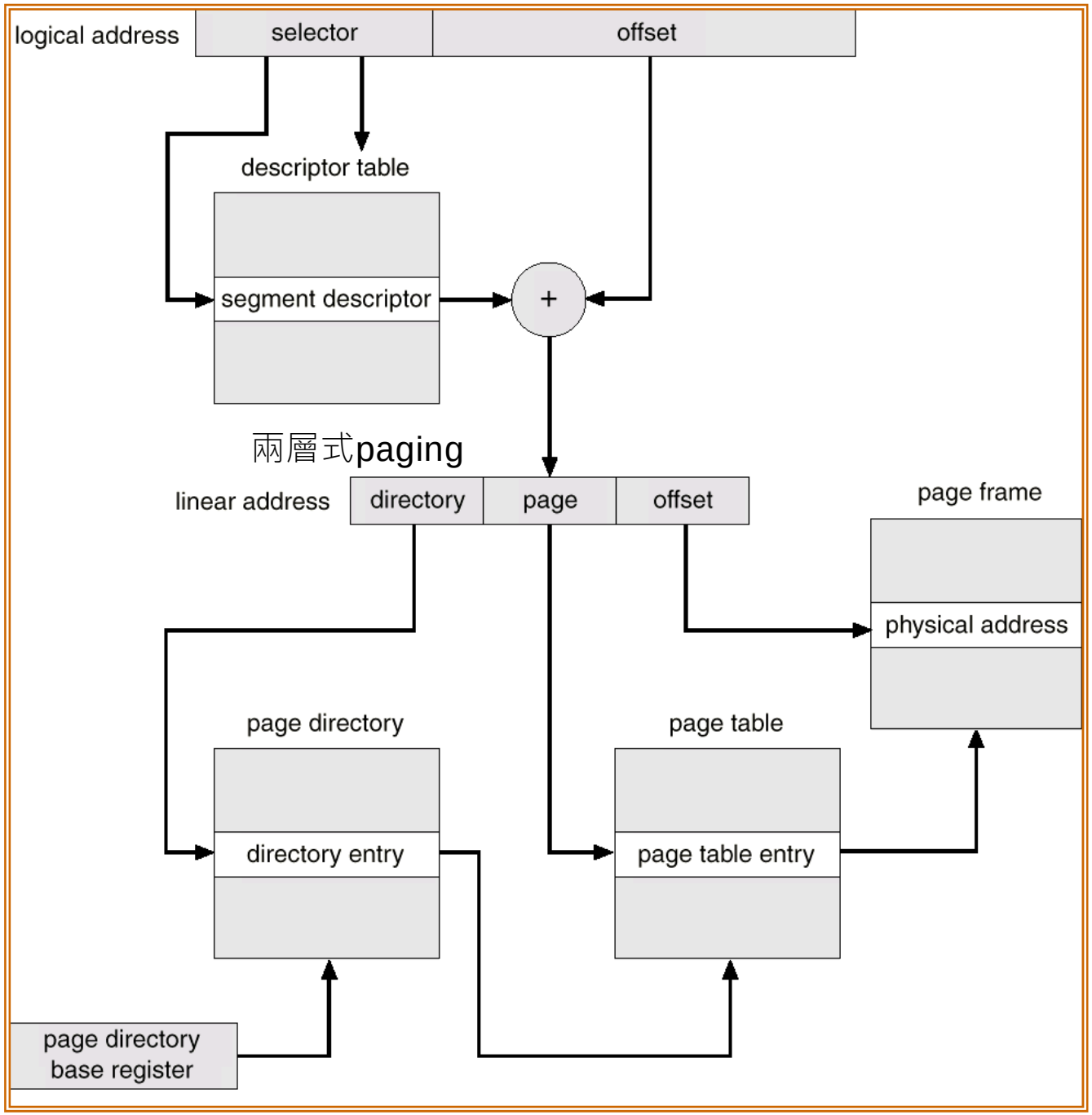
Logical Address to Linear Address

- Logical address format
  - [Selector:16]:[offset:32]
  - Selector format is [s:13][g:1][p:2]



Linear Address to Physical Address

- Once logical address is translated into linear address, it is handled by paging
- Linear address format
  - [p1:10][p2:10][d:12]


logical address
selector
offset
descriptor table
segment descriptor
+
兩層式paging
linear address
directory
page
offset
page frame
physical address
page directory
page table
directory entry
page table entry
page directory
base register

Linux on Intel 80386

- Uses minimal segmentation to keep memory management implementation more portable
- Uses 6 global segments: (80386 has many)
  - Kernel code
  - Kernel data
  - User code (shared by all user processes)
  - User data (shared by all user processes)
  - Task-state (per-CPU hardware context)
  - An LDT (shared by all processes)
- Uses 2 protection levels: (80386 has 4)
  - Kernel mode
  - User mode
- Use three-level paging (80386 has only 2)
  - The mid-level paging is disabled

End of Chapter 8