

Chapter 9: Virtual-Memory Management

Prof. Li-Pin Chang
National Chiao Tung University

Chapter 9: Virtual Memory

- Demand Paging
- Copy-on-Write
- Page Replacement
- Thrashing
- Allocation of Frames
- Performance Issues
- Swapping
- Memory-Mapped Files
- Kernel Memory Allocation
- Operating System Examples

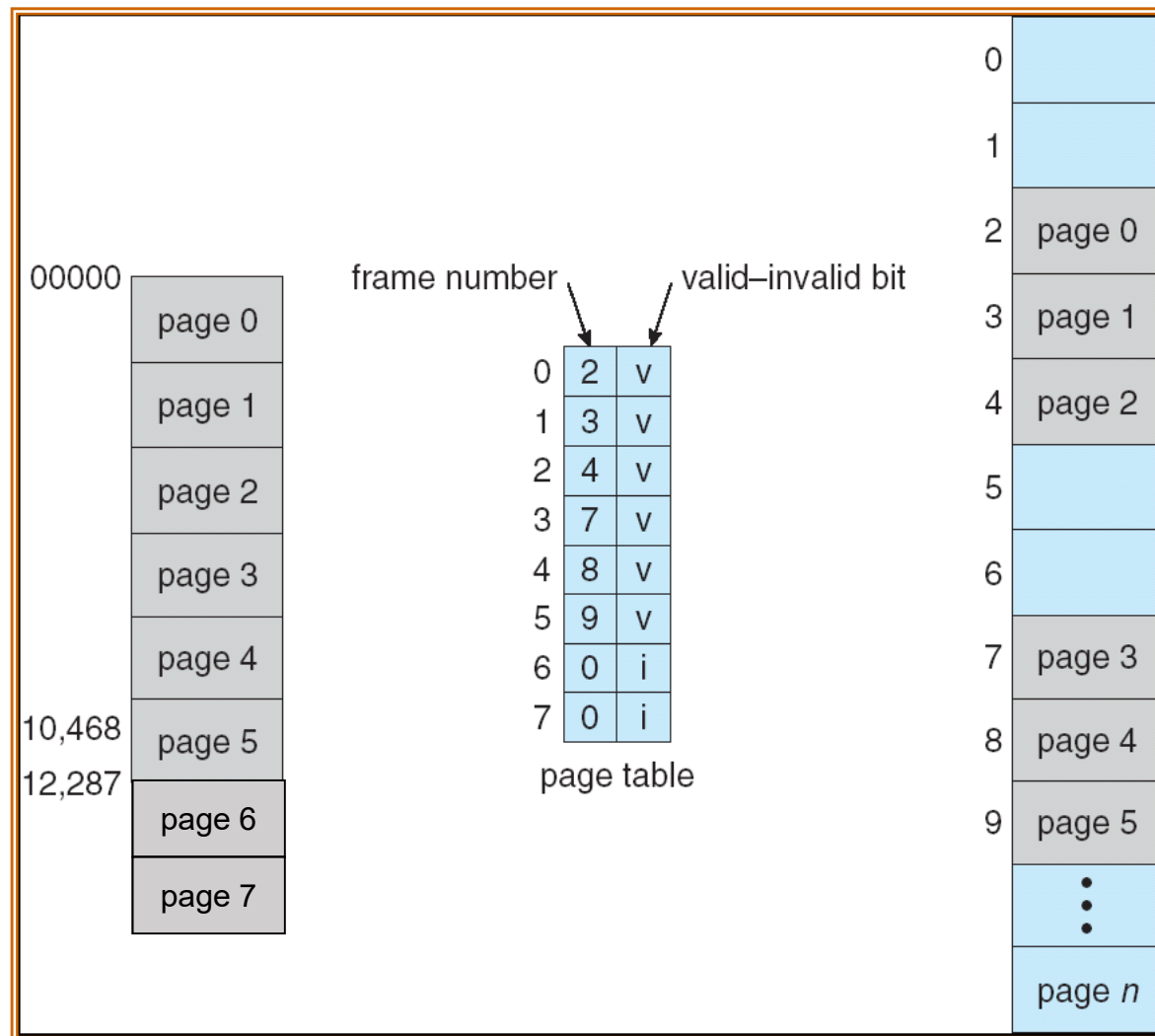
DEMAND PAGING

Memory Protection

- Memory protection implemented by associating protection bit with each page
- **Valid-invalid** bit attached to each entry in the page table:
 - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
 - “invalid” indicates that the page is not in the process’ logical address space
- An invalid page is either an illegal one or a valid one but not referenced yet

表示這東西在 logical space 裡面還沒用到(是空的)

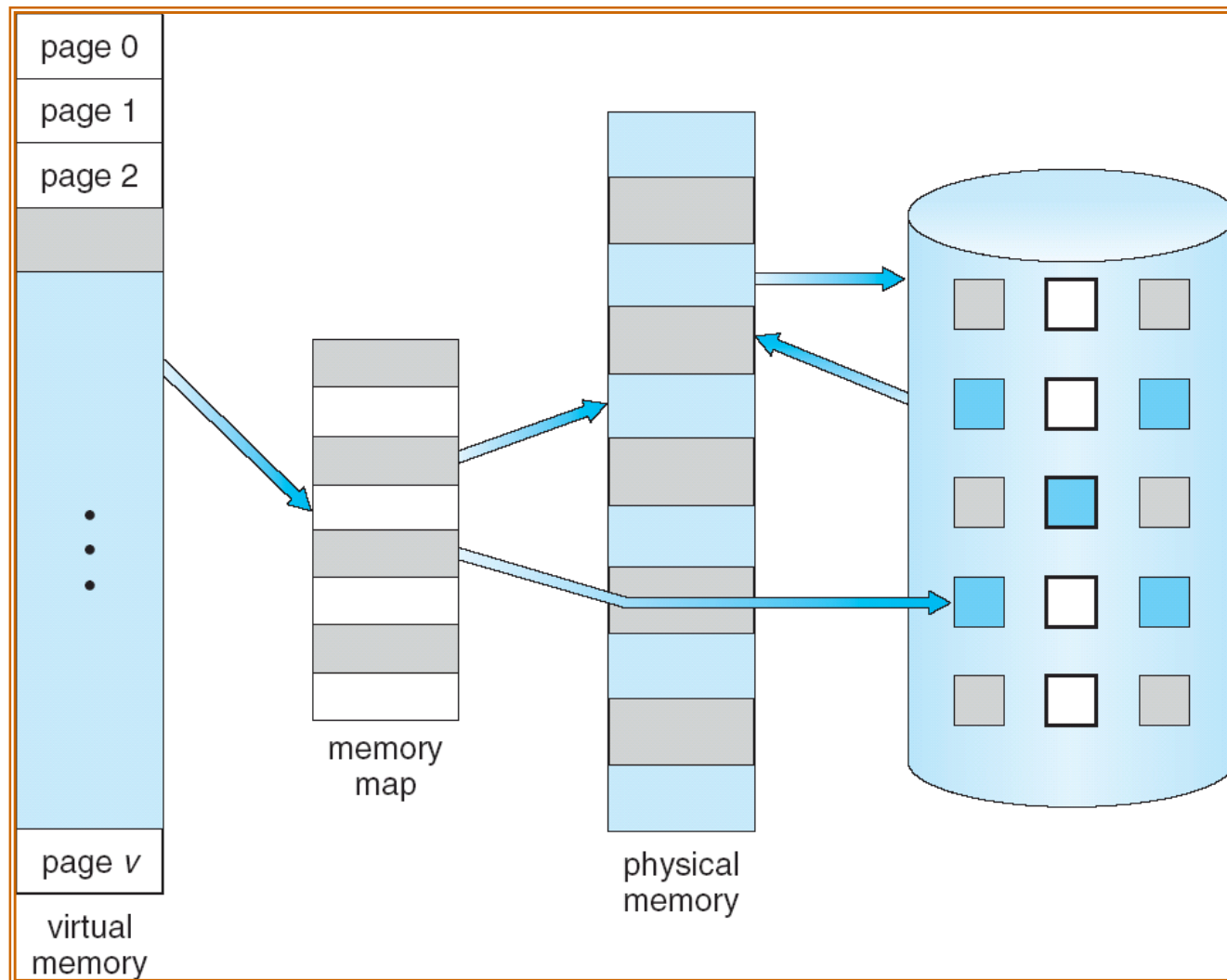
Valid (v) or Invalid (i) Bit In A Page Table



Virtual Memory

- Virtual memory – separation of user logical memory from physical memory
 - Only **part of the program** needs to be in memory for execution
 - Logical address space can therefore be **much larger than physical address space**
 - Allows for more efficient process creation
 - Improves the degree of multiprogramming
 - Allows address spaces to be shared by several processes
- Virtual memory is usually implemented by demanded paging

Virtual Memory That is Larger Than Physical Memory



Demand Paging

- Logical memory space can be larger than physical memory space
- Bring a page into memory only when it is needed
 - Less I/O needed
 - Less memory needed
 - Faster response
 - More users/processes
- Page is needed \Rightarrow reference to it
 - invalid reference \Rightarrow abort
 - not-in-memory \Rightarrow bring to memory

Valid-Invalid Bit, Revisited

- With each page table entry a valid–invalid bit is associated (1 \Rightarrow in-memory, 0 \Rightarrow not-in-memory)
- Initially valid–invalid bit is set to 1 on all entries
- Example of a page table snapshot:

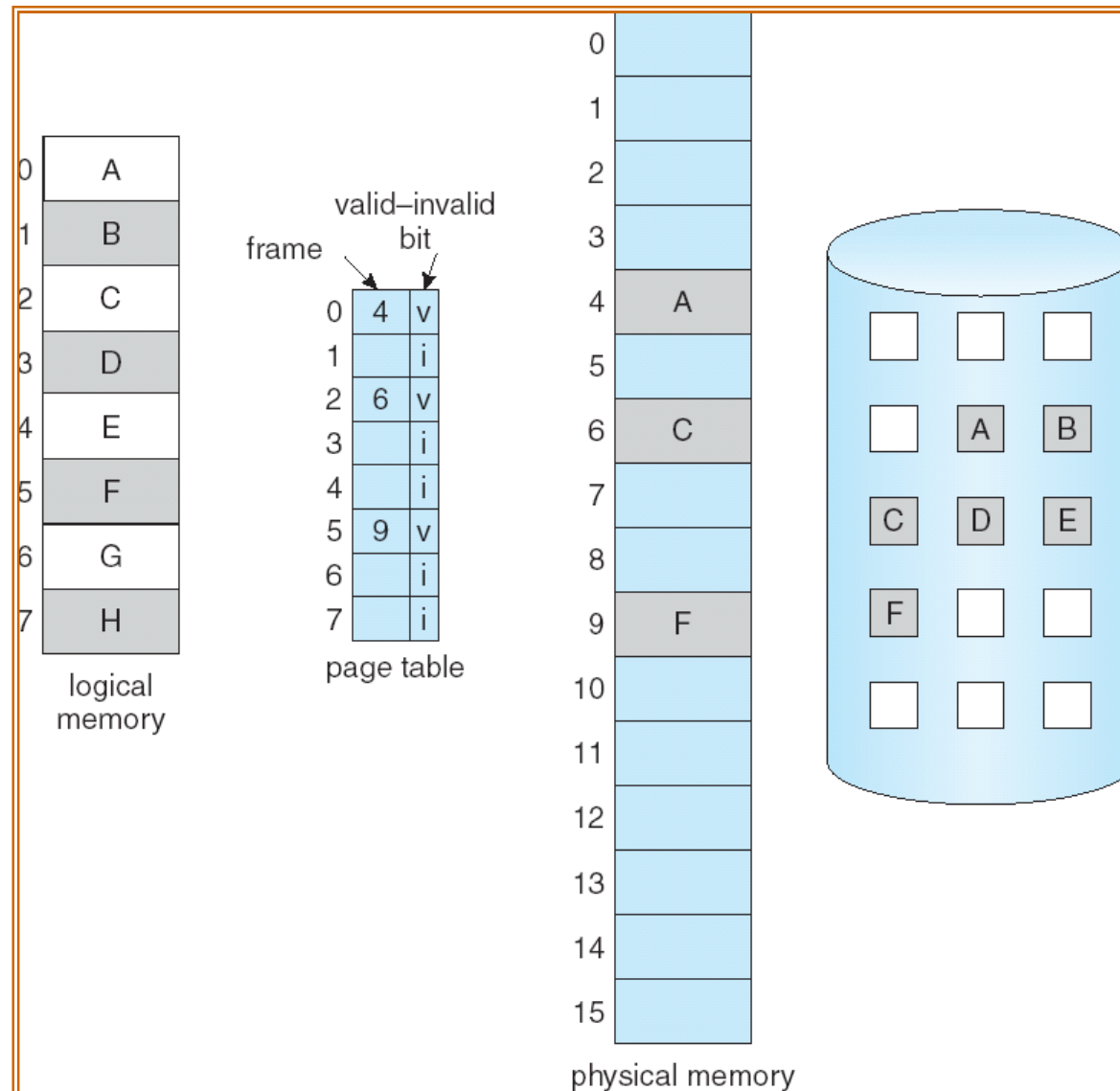
Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

page table

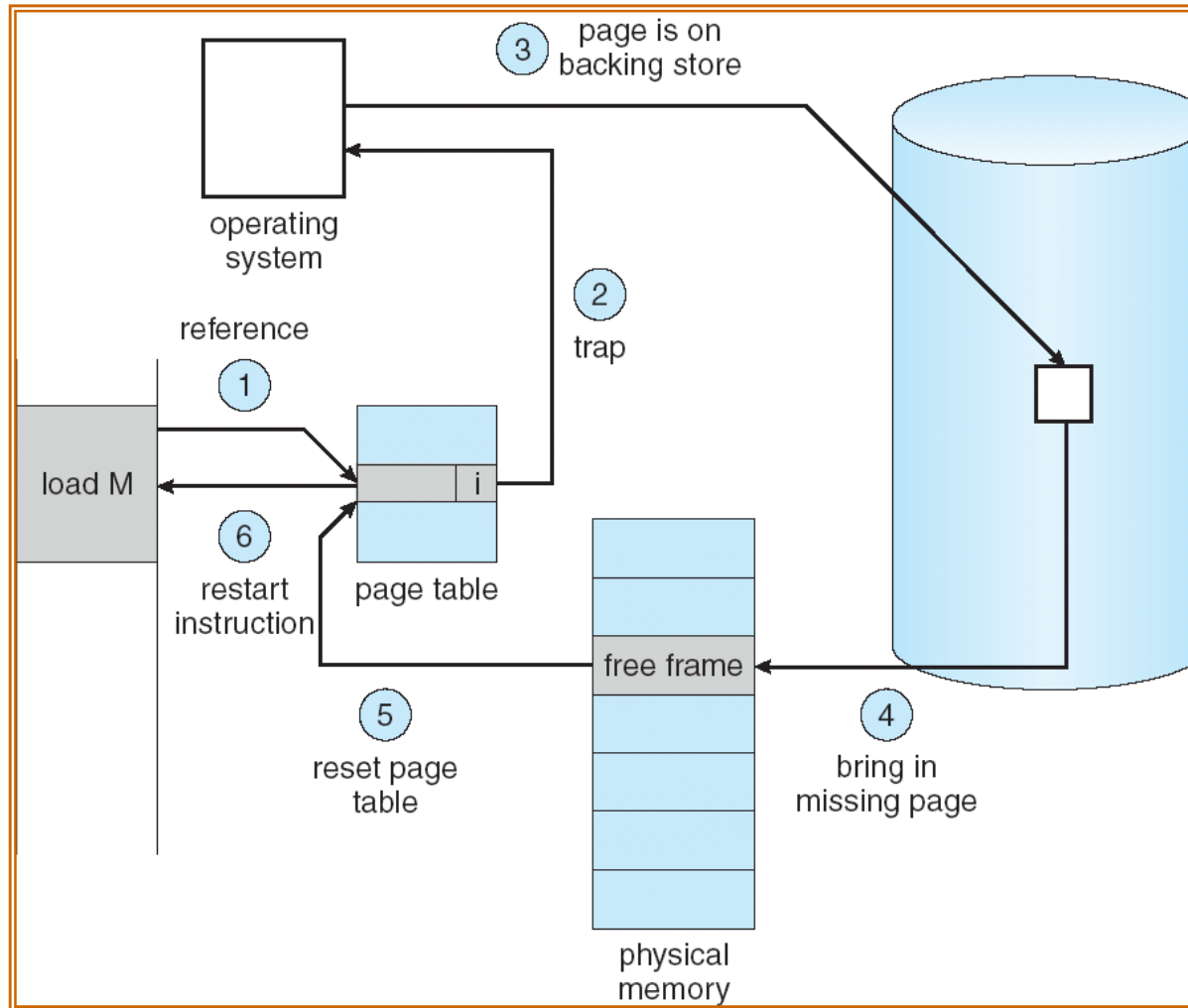
- During address translation, if valid–invalid bit in page table entry is 0 \Rightarrow page fault

page不在memory裡(沒有對應的frame)
要去磁碟撈資料

Page Table When Some Pages Are Not in Main Memory



Steps of Handling a Page Fault



1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
 - a. Wait in a queue for this device until the read request is serviced.
 - b. Wait for the device seek and/or latency time.
 - c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.

- Up to how many page faults will be caused by the following instruction?

mov ax, [edx]

- Total 2
 - The instruction itself once and the access of memory location [edx] the other one
 - Instructions and data are aligned to page boundaries

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)
$$\begin{aligned} \text{EAT} = & (1 - p) \times \text{memory access} \\ & + p (\text{page fault overhead} \\ & + [\text{swap page out}] \\ & + \text{swap page in} \\ & + \text{restart overhead}) \end{aligned}$$

Suppose that the page-fault service time is 8 ms, including disk I/O

A memory access takes 200ns (TLB hit+TLBmiss)

The page-fault ratio is p

The EAT is

$$(1-p)*200+p*8ms \\ =200+7999800p$$

The RHS term dominates the EAT! p should be as low as possible!!

If $p=1/1000$, $EAT = 200+7999 \sim 8.2\mu s$, 40 times slower!!

If the expected slowdown is no larger than 10% compared to 200ns, then

$$220 \geq 200 + 7999800p$$

$$20 \geq 7999800p$$

$P \leq 0.0000025$ in other words, no more than 1 page fault should happen out of 399,990 memory access.

- If TLB hit
 - won't be a page fault
 - TLB access
- If TLB miss
 - If not a page fault
 - page table access + TLB update
 - If a page fault
 - The page is not in memory
 - page table access + page fault handling + TLB update

****Consider a demand-paging system with a paging disk that has an average access and transfer time of 10 milliseconds. Addresses are translated through a page table in main memory, with an access time of 4 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page-table entry is in the associative memory.**

Assume that 87.5% of the accesses are in the associative memory and that, 20% of the remain 12.5% cause page faults. What is the effective memory access time?

$$0.875 \times 4 + 0.125 \times 0.8 \times (4 + 8) + 0.125 \times 0.2 \times (4 + 8 + 10000)$$

PAGE REPLACEMENT

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Data in virtual memory is not necessarily in physical memory

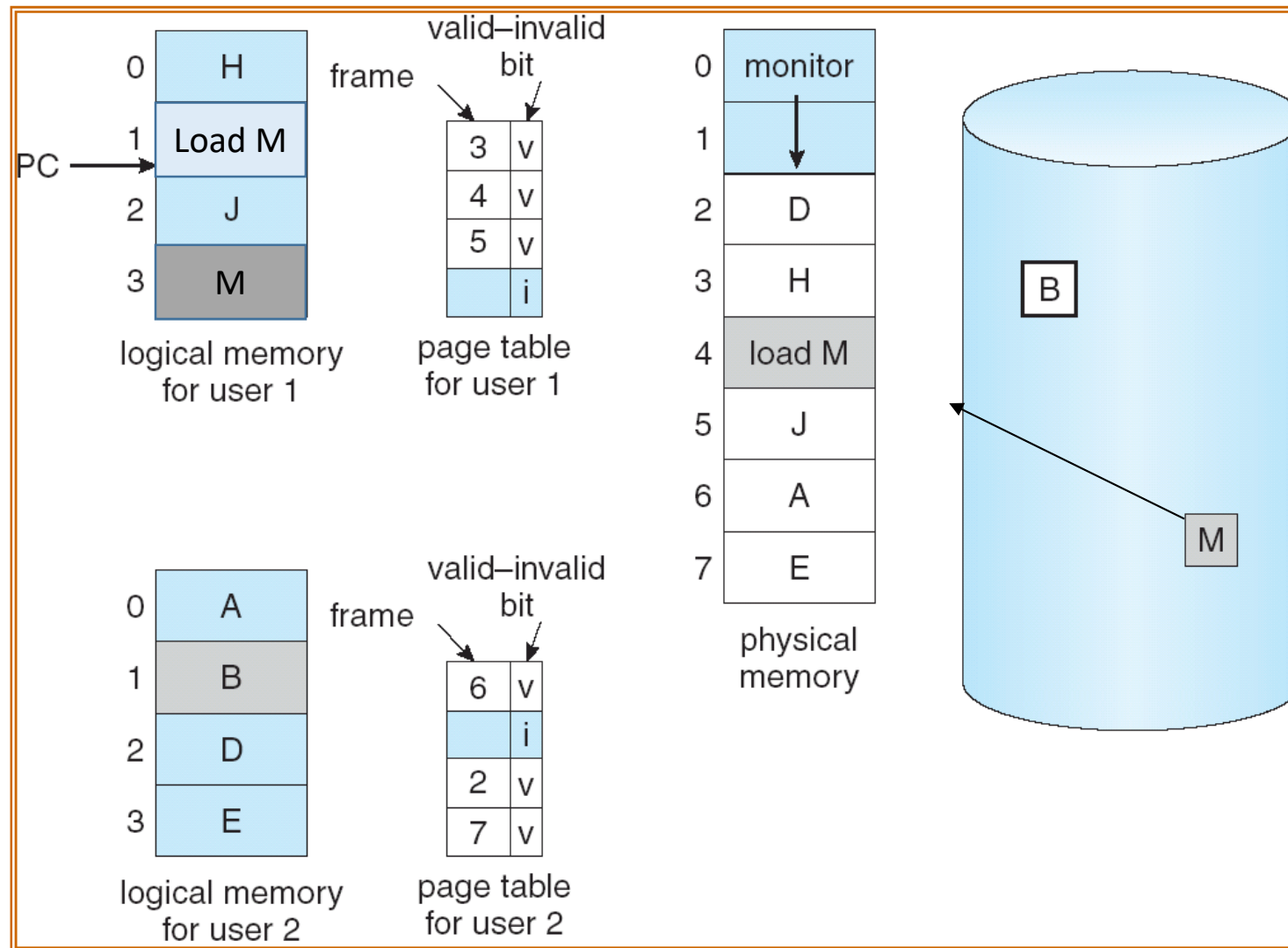
Basic Page Replacement

- Find the location of the desired page on disk
- Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim** frame
- Read the desired page into the (newly) free frame
- Update the page and frame tables
- Restart the process

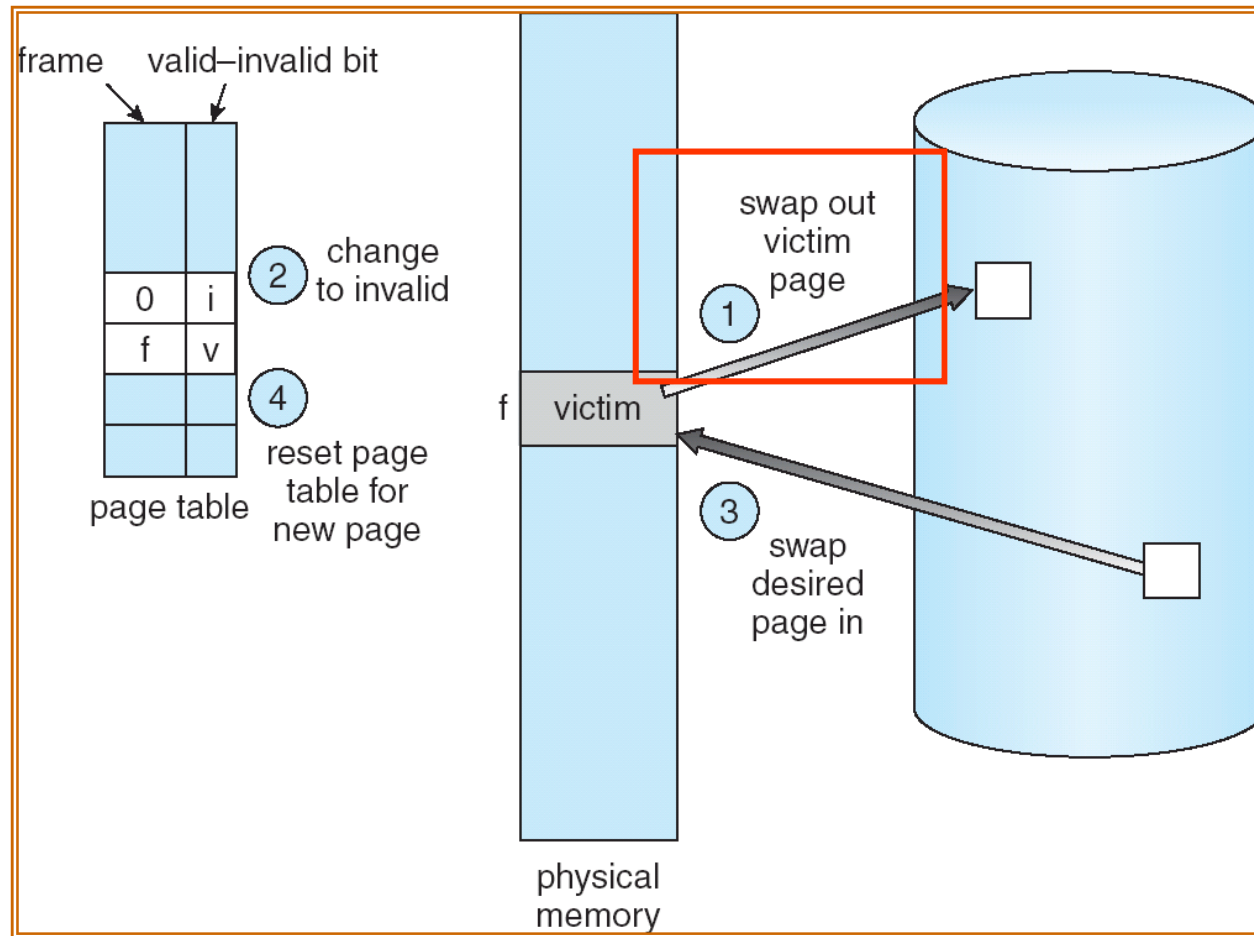
Page Replacement

- To replace a page, the victim page must be **written back** to the backing store. It may double the time of page-fault handling
- Replace a page that **is unlikely to be used in the near future** to reduce the overhead of reading a page from disk
- Use **modify (dirty) bit** to reduce overhead of page transfers – **only modified pages are written to disk**

Need For Page Replacement



Page Replacement



Reference string

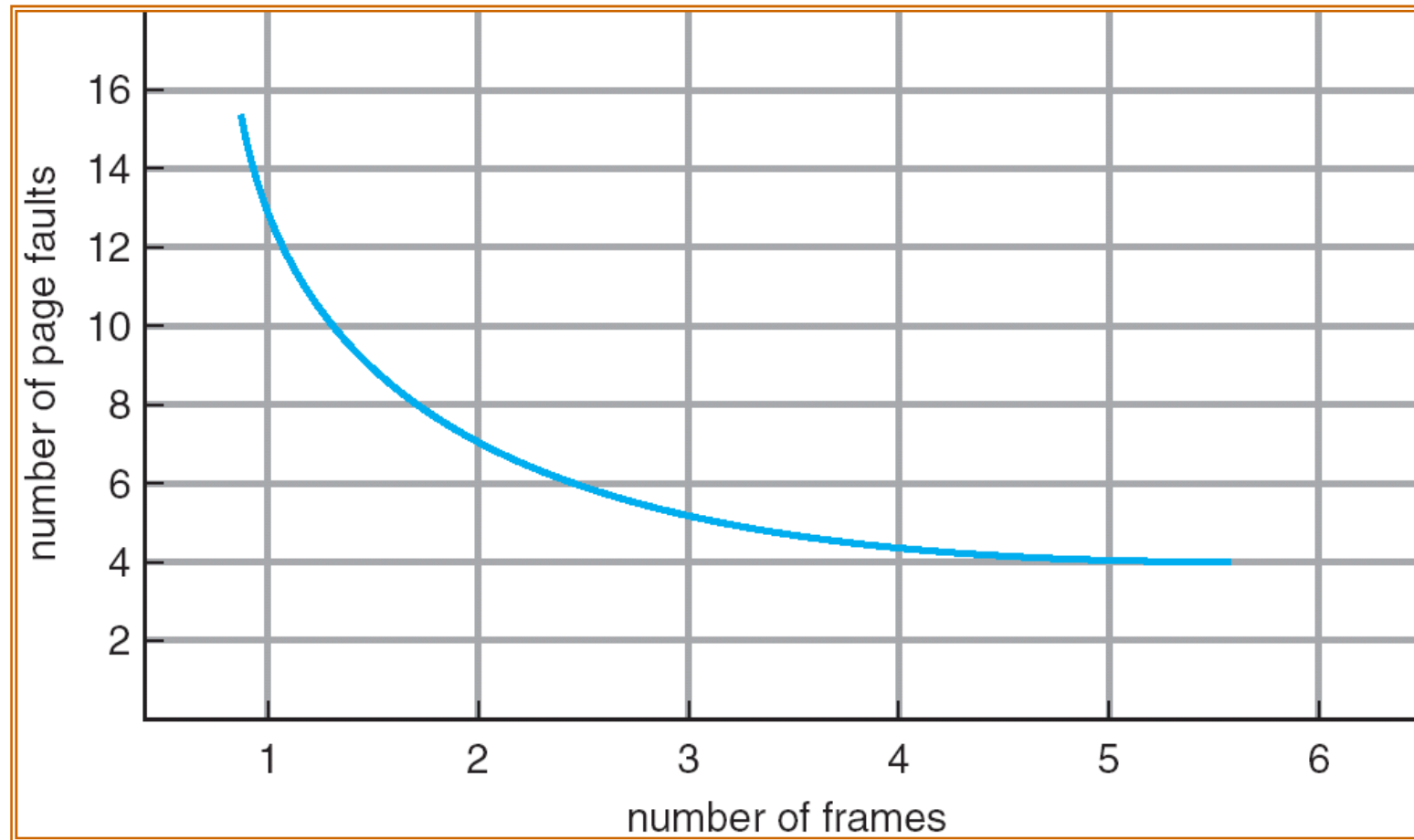
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105



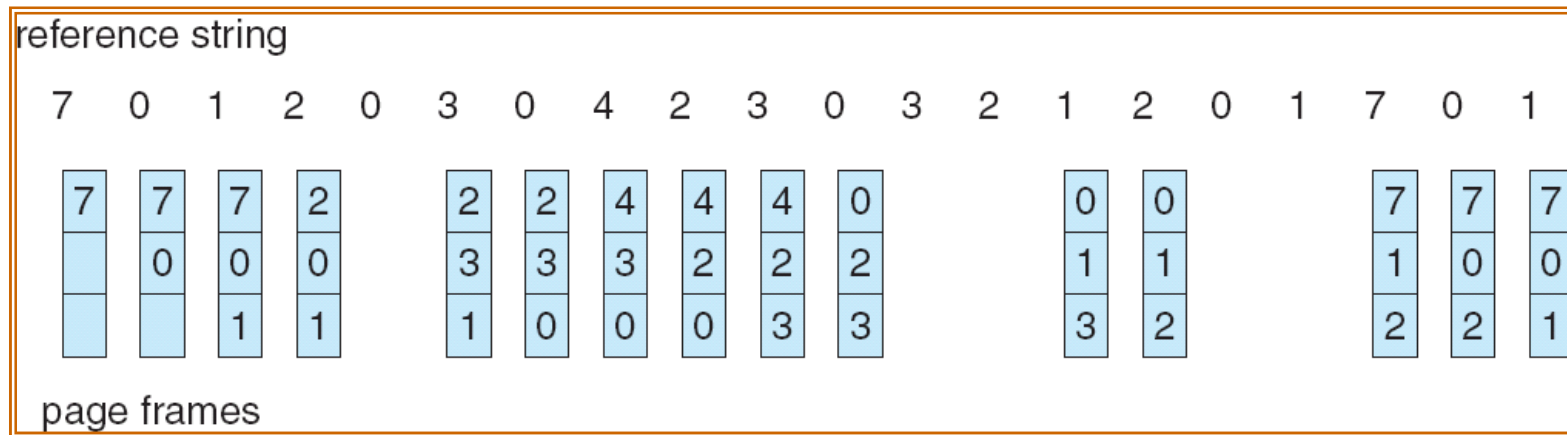
Page size=100B

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Graph of Page Faults Versus The Number of Frames

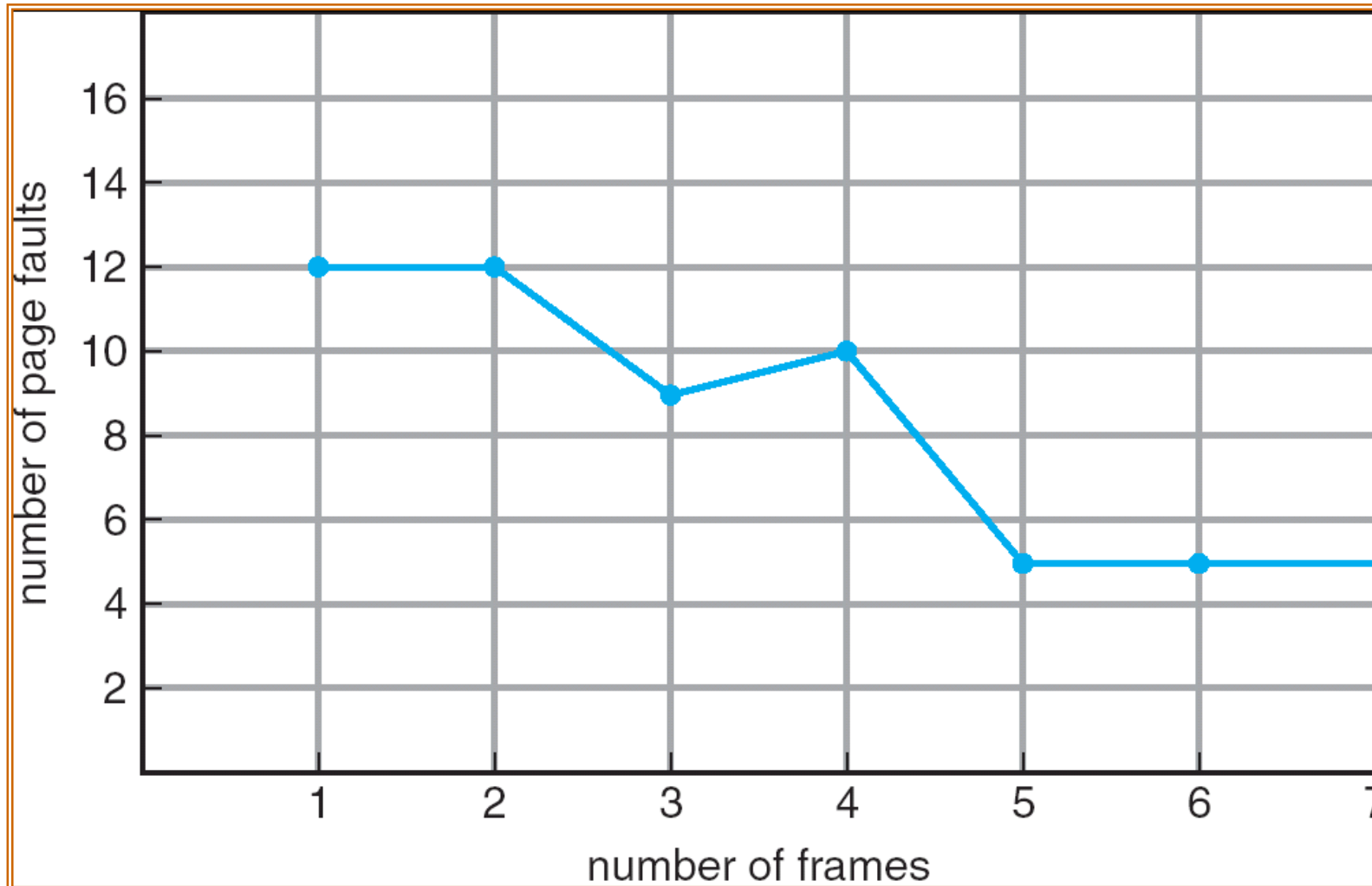


FIFO Page Replacement



15 page faults

FIFO suffers from Belady's Anomaly

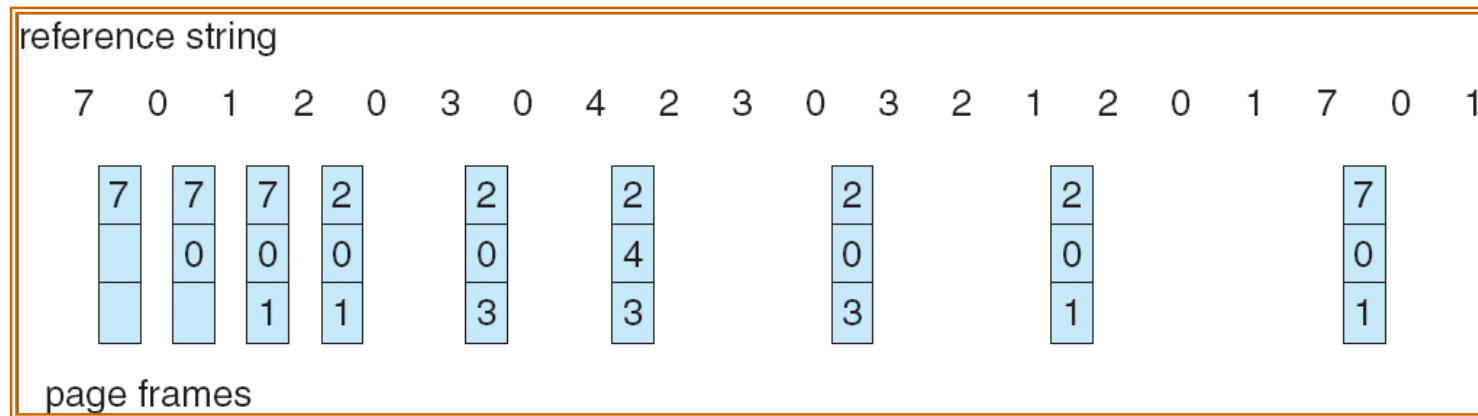


Reproduce Belady's Anomaly

異常

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Optimal Page Replacement (OPT)

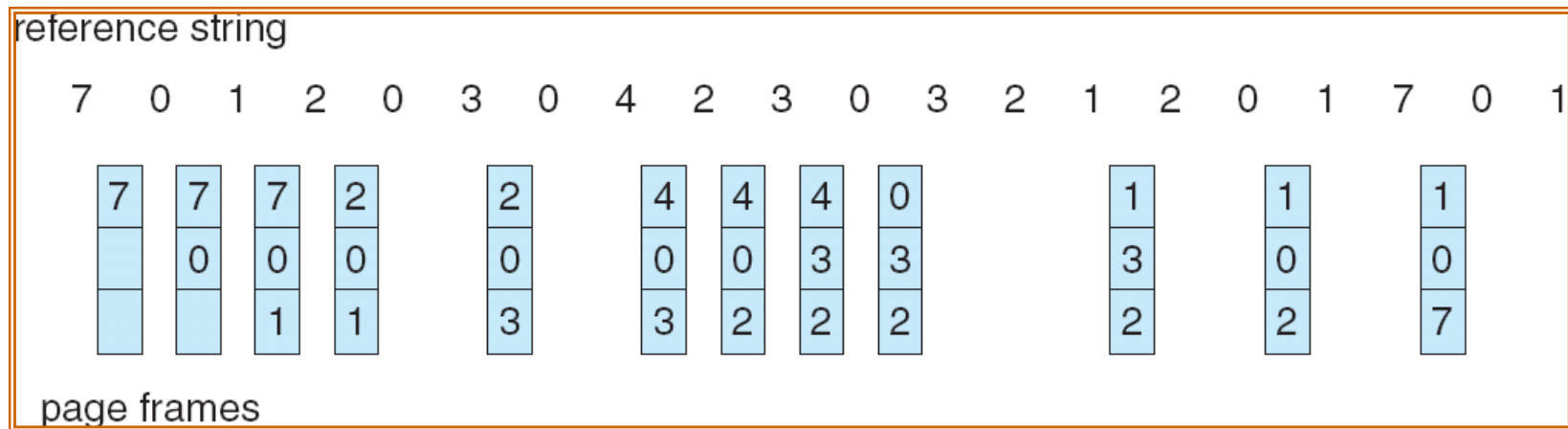


9 page faults.

OPT is not applicable in real systems, however.

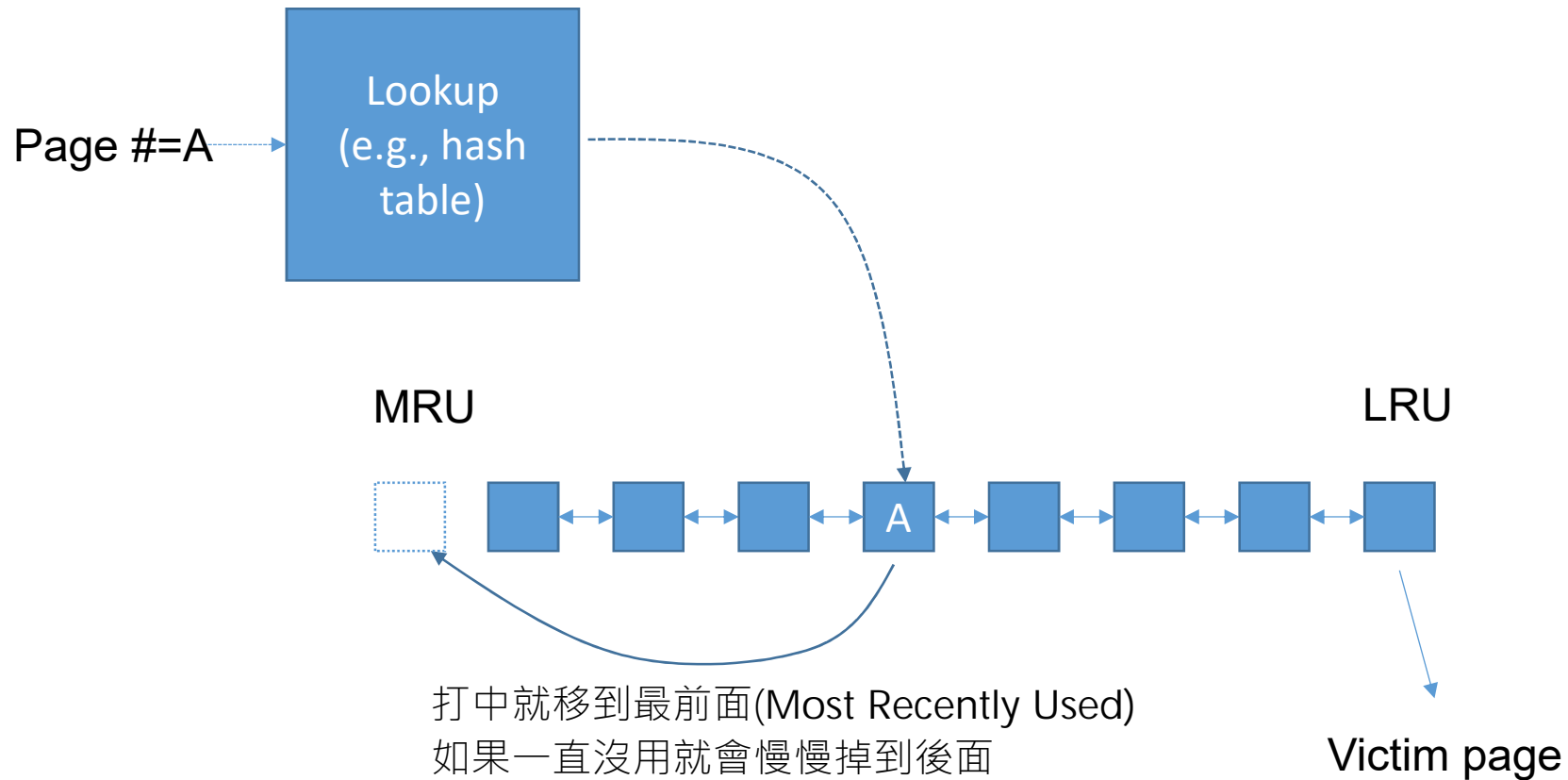
簡單來說就是神預測做不到

Least-Recently Used (LRU)



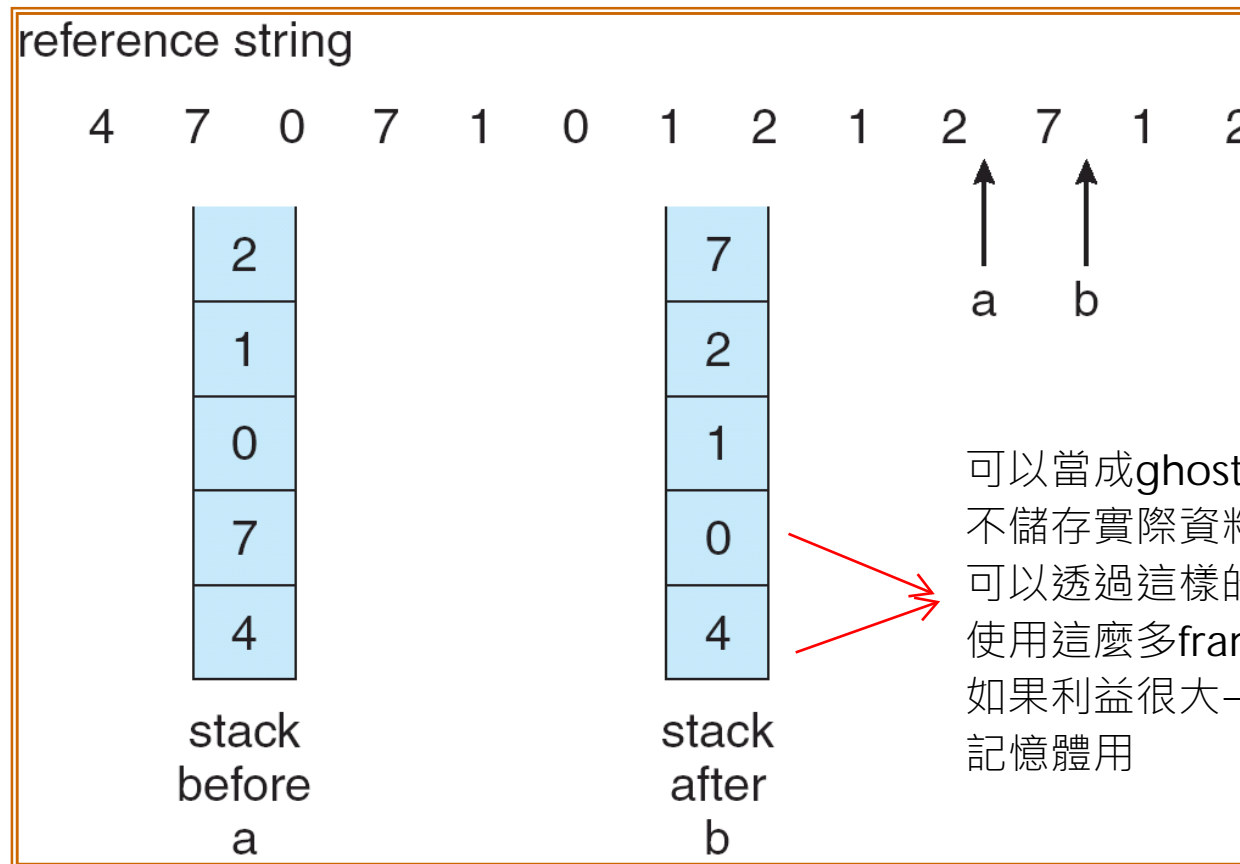
12 page faults.

An LRU Implementation



The “Stack” Property of LRU

用n frames 的結果
是n+1 frames 的結果
→stack property
→用更多frames只會
讓結果更好



See what happens if there are four frames only.

The “Stack” Property of LRU

- The page set of LRU with K pages is a superset of that of LRU with $K-1$ pages
- Useful in cache simulation. Simulating K pages get results of $K, K-1, K-2, \dots$

Belady's Anomaly (revisited)

- LRU and OPT won't suffer from Belady's anomaly
- Observation
 - LRU and OPT are “stack algorithms”.
 - i.e., the page set with n frames is a subset of the set of page set with $n+1$ frames
 - Both OPT and LRU have this property
- Proof
 - By mathematical induction

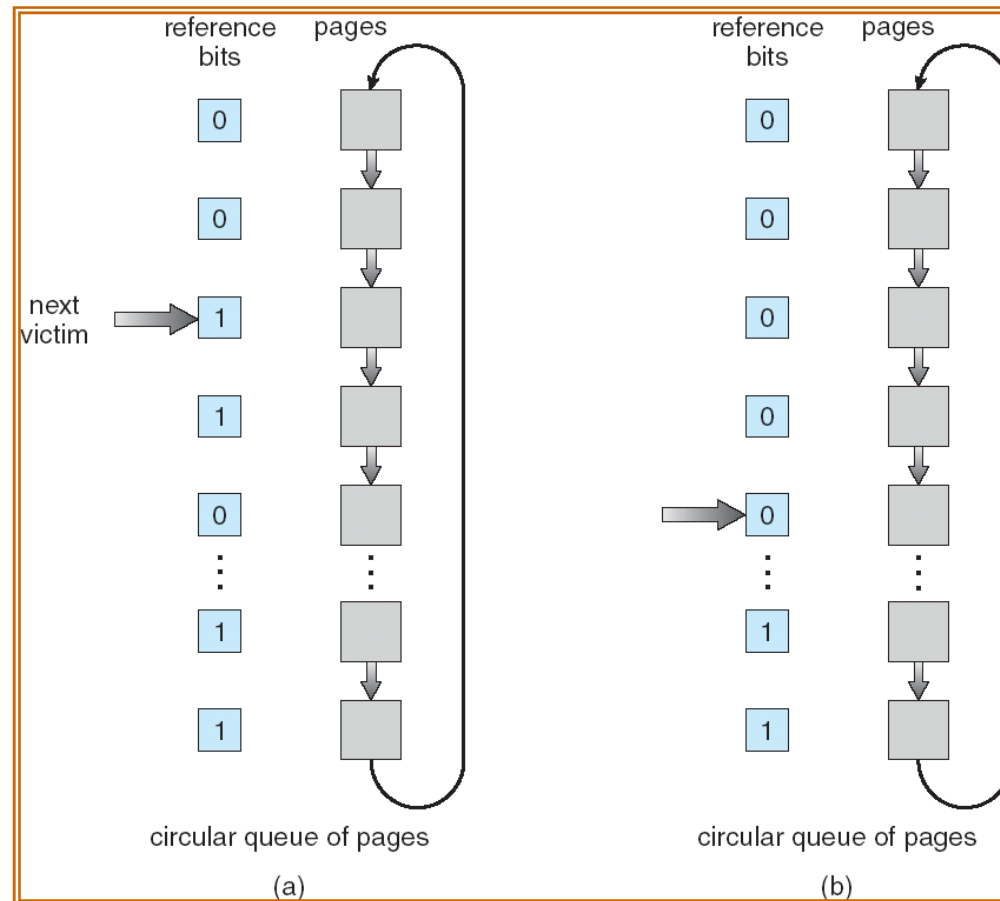
變形實作方式，更簡單，但效果只是"接近"

LRU Approximation Algorithms

- Reference bit
 - With each page associate a bit, initially = 0
 - When page is referenced bit set to 1
 - Replace the one which is 0 (if one exists). We do not know the order, however
- Second chance
 - Need reference bit
 - Clock replacement
 - If page to be replaced (in clock order) has reference bit = 1 then:
 - set reference bit 0
 - leave page in memory (i.e., give a second chance)
 - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm

如果有新的要擠進來
看指到的reference bits
1：從上一次指到它到現在
，它有被用過
→把bit設0，指到下一個
0：從上一次指到它到現在
，它沒被用過
→replace掉這個



It degrades into the FIFO policy in the worst case

Enhanced Second-Chance Algorithm

(ref, dirty)

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon
4. (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

第二、第三要先
踢出哪個？

選二來replace

因為如果踢三，當下會省掉寫回
的時間，但很可能等一下又要把它載回來(更虧)

- The first page encountered at the lowest nonempty class is replaced
- This method considers to reduce I/O costs.

Counting Algorithms

但如果有一個
page曾被用過很多
次就不會被踢
出去

解法：counter有
半衰期(向右shift)

- Keep a counter of the number of references that have been made to each page

- **LFU** Algorithm: replaces page with smallest count

- Periodically bit shift for counters (aging)

每個page都維護一個counter，表示它被
用了幾次(counter也不用太大)

- **MFU** Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

踢出次數最小的

Most Frequently Used

- Preserving pages newly brought in

覺得用的很少的frame是因為才被帶進來
沒多久，應該保留

Recency vs. Frequency

- LRU

- Replace the least recently used page
- Fast response to locality change
- Sequential reference will wash away all cached pages

缺點：碰到sequential access就會死掉
會把原本常用的page全部洗掉，接下來一陣子cache裡面都是沒用的東西

- LFU

- Replace the least frequently used page
- Resistant to sequential reference
- Need time to warm up and cool down a page

因為sequential reference data 的 counter基本上都是1(只會用一次)

缺點：

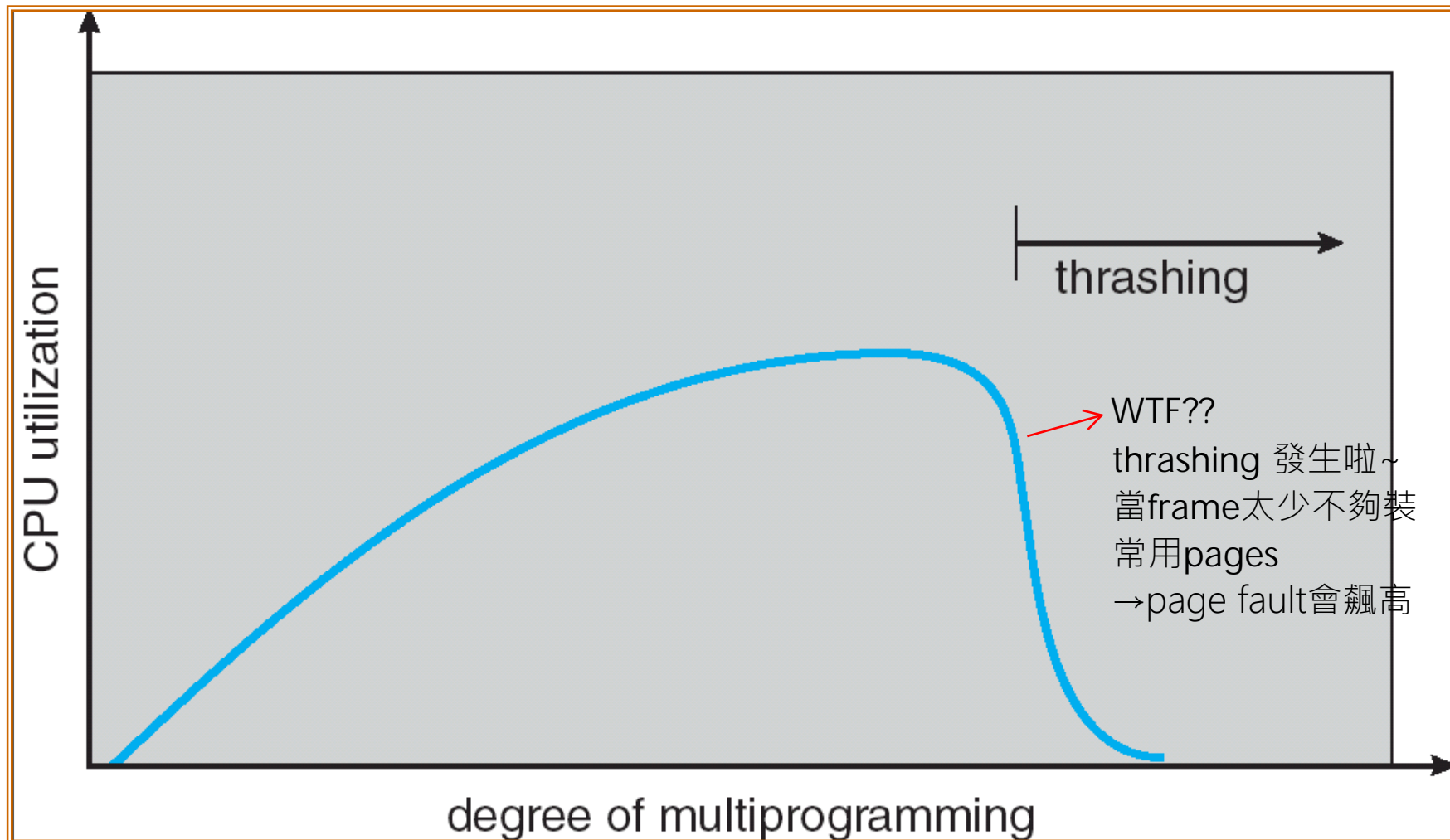
剛進來的page很可能被刷掉
即使它以後會很常被用
沒有長大的機會，幫QQ

THRASHING

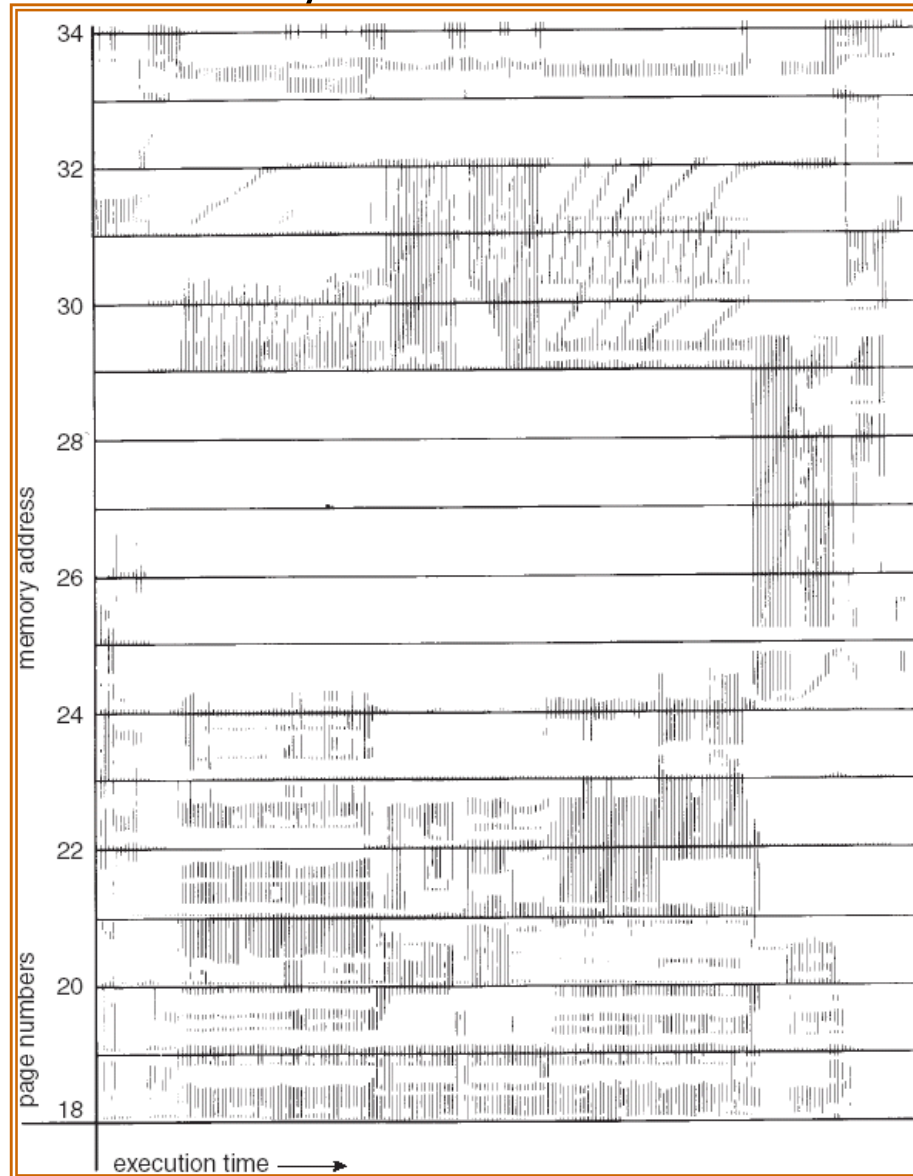
Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
 - low CPU utilization
 - operating system thinks that it needs to increase the degree of multiprogramming
 - another process is added to the system
- **Thrashing** \equiv a process that is busy swapping pages in and out

Thrashing (Cont.)



Locality In A Memory-Reference Pattern



Working-Set Model

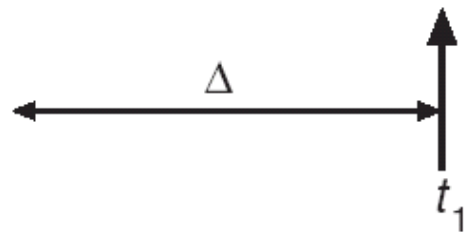
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instruction
- WSSi (working set of Process Pi) =
total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
 - m is the total number of frames
- Policy if $D > m$
 - Swap out some processes
 - De-allocate pages from processes

空出更多的frames來用

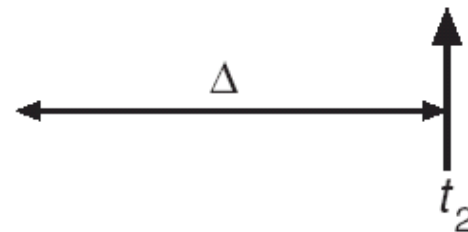
Working-set model

page reference table

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .



$WS(t_1) = \{1, 2, 5, 6, 7\}$

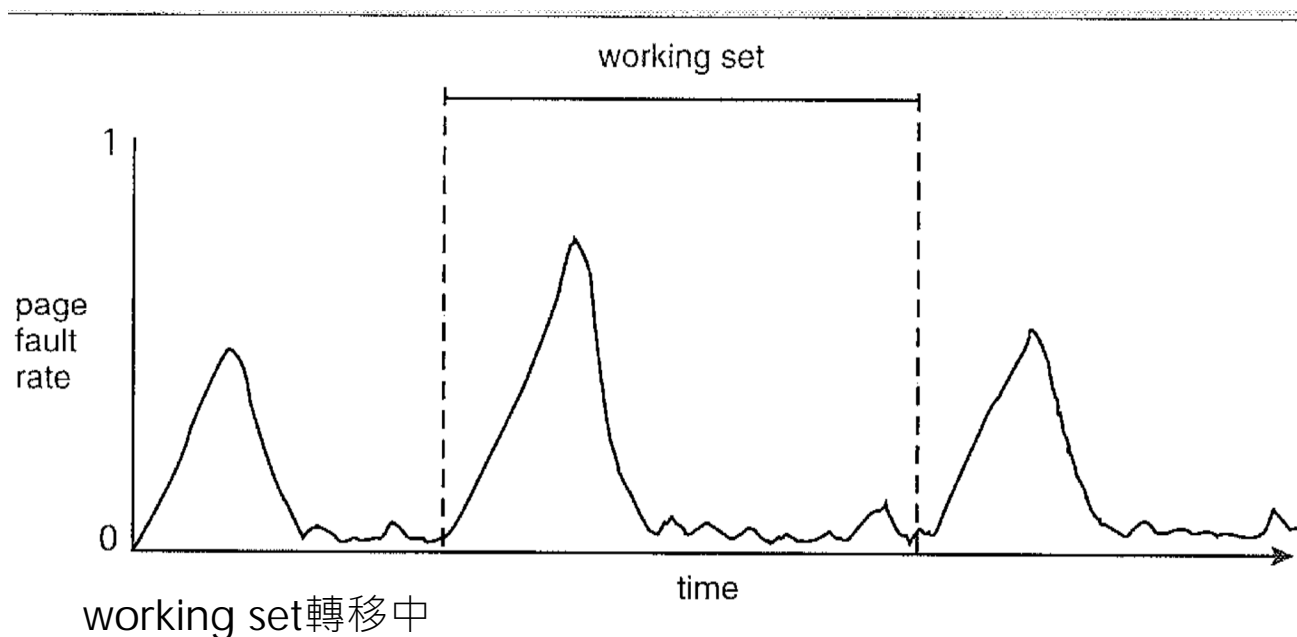


$WS(t_2) = \{3, 4\}$

一定時間內，用到的page集合

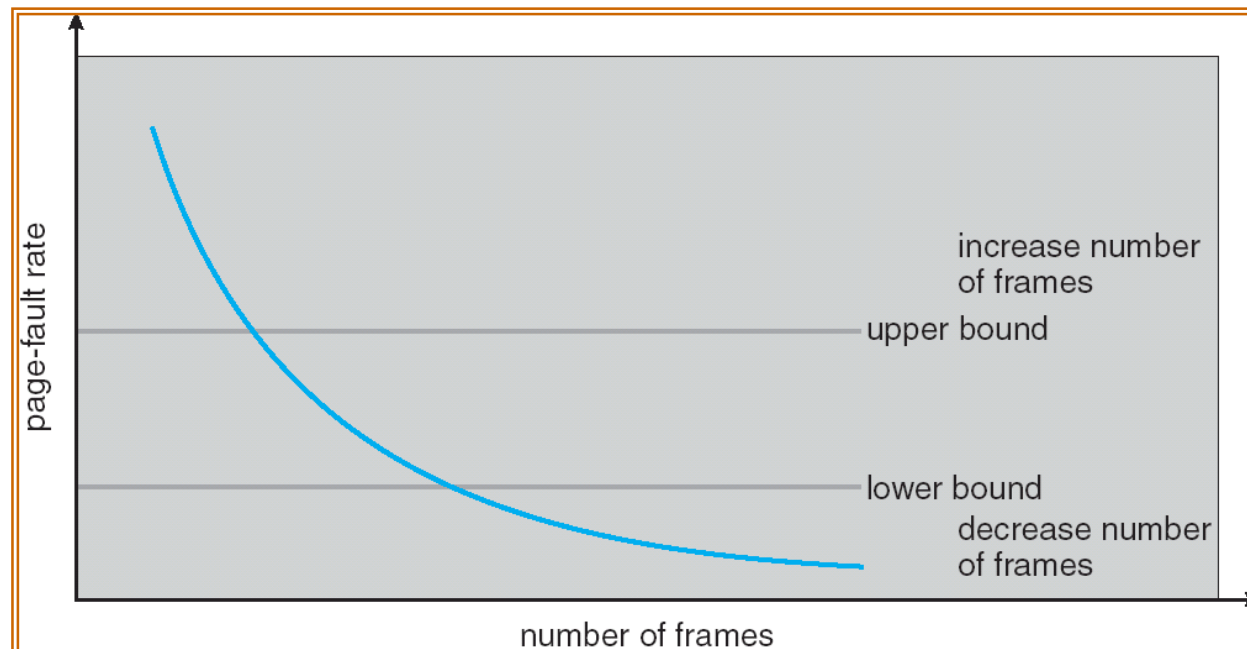
Remark: Migration of Working Sets

- Working set is a time-variant, when a process
轉移 ← migrates from a working set to another, the page
fault rate also increases (not thrashing) 因為要換另一批pages



Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
 - If actual rate too **low**, process **gains** frame
 - Swap in more processes 可以多跑點processes
 - If actual rate too **high**, process **loses** frame
 - Swap out some processes or discard some pages 太多processes了，踢出去一些



PERFORMANCE ISSUES

Performance Issues

- Set aside a number of free frames.
 - A page fault is handled and the process is restarted immediately.
- The OS replaces and swap out pages in **background** to keep the number of free frames adequate 足夠的
 - No need to discard them, just make them “clean”
 - The **OS flushes dirty pages to disks whenever the computer is idle** 反正也沒事做，就確保一下資料完整
 - `kswapd` and `pdflush` in Linux

Performance issues

- **Pre-paging** (similar to disk pre-fetch)
 - To reduce the large number of page faults that occurs at process startup
 - Reducing the count of disk operations by taking advantage of **spatial localities**
 - **Fetch several pages ahead upon a page fault**
 - 128 KB=32 pages in Linux
 - But if pre-paged pages are unused, I/O and memory was wasted 可能載到不需要的
- Assume s pages are pre-paged and α (**a fraction of s**) of the pages is used
 - Is cost of $s * \alpha$ save pages faults $>$ or $<$ than the cost of pre-paging $s * (1 - \alpha)$ unnecessary pages?
 - α near zero \Rightarrow pre-paging loses pre-paged進來的用的很少 \rightarrow 不值得

Program Structure

- Program structure

- `Int[128,128] data;`
- Each row is stored in one page

- Program 1

```
for (j = 0; j < 128; j++)  
  for (i = 0; i < 128; i++)  
    data[i,j] = 0;
```

locality差的寫法...

128 x 128 = 16,384 page faults

- Program 2

```
for (i = 0; i < 128; i++)  
  for (j = 0; j < 128; j++)  
    data[i,j] = 0;
```

128 page faults

Page size=128

Suppose that we have < 128 frames...

TLB Reach

- TLB Reach = (TLB Size) X (Page Size)
 - The amount of memory accessible from the TLB
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of TLB miss 如果working set大於TLB Reach會造成TLB miss急速升高
- Increase the Page Size
 - May have negative performance impacts: internal fragmentation, higher page fault ratios, etc.
- Provide Multiple Page Sizes
 - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.
 - UltraSparc provides multiple page sizes of: 8KB, 64KB, 512KB, 4MB
 - IA-64 supports 4 KB and 4 MB pages 雙層page table很容易就能提供不同種page size

Increase the TLB size

實作上不可行!因為TLB是fully associative，電路面積成長是super linear

會在讀取時容易
讀進很多不要的
資料

Page Size Tradeoff


- Large pages
 - Small page table 因為page number會減小，page entry不增加(因為總byte固定)
 - Large TLB reach
 - High page fault ratio (why?) 因為memory replacement用大塊的page，很容易出現需要常常替換的狀況
- Small pages
 - Large page table
 - Small TLB reach
 - Low page fault ratio (why?)

Page Size Tradeoff (cont'd)

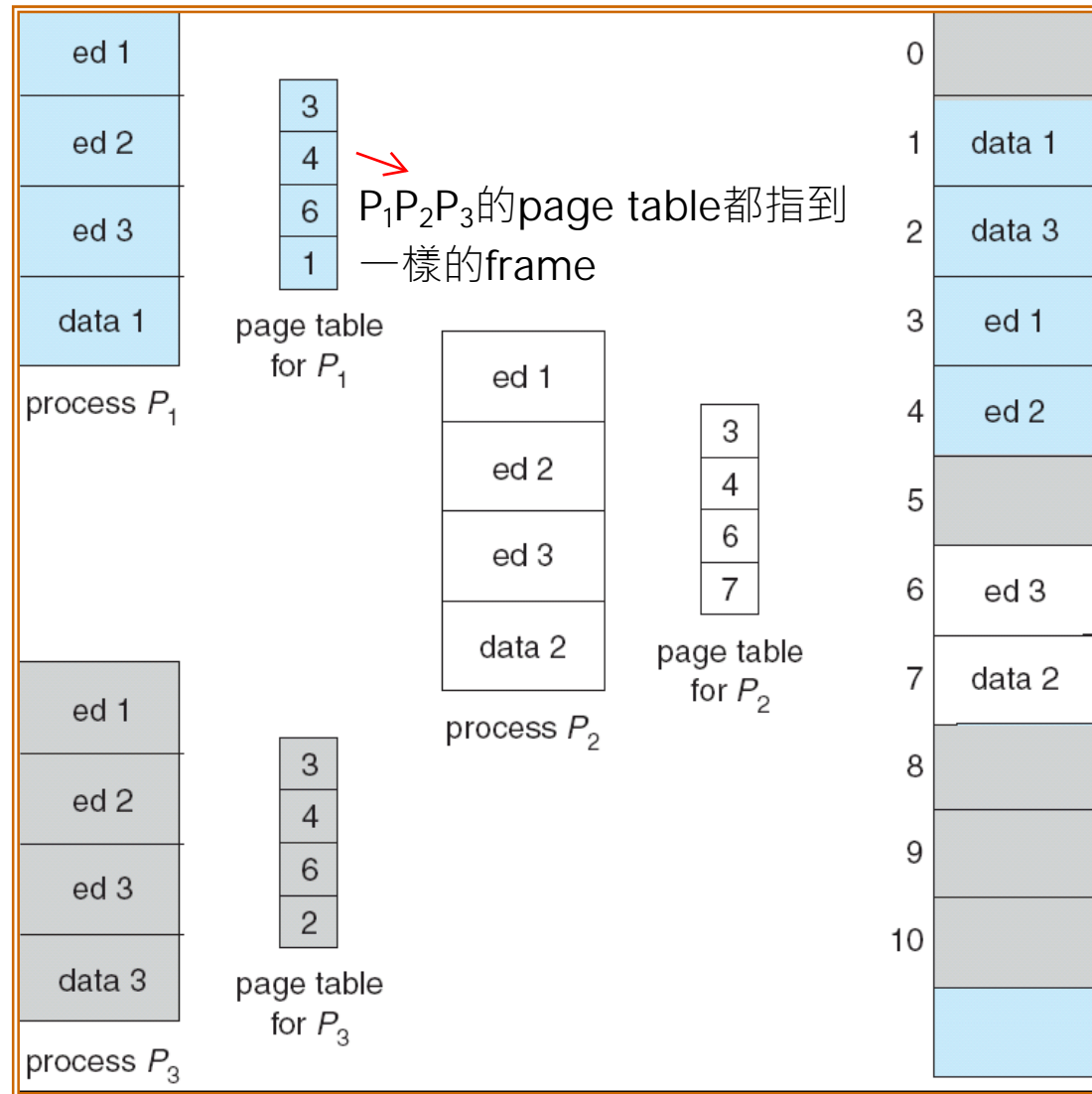
- Page Table size
 - Use large pages for a small page table
- I/O overhead
 - Use reasonably large pages to take advantage of disk sequential access performance
- Locality
 - Use small pages to accurately capture localities

Shared Pages & Copy on Write

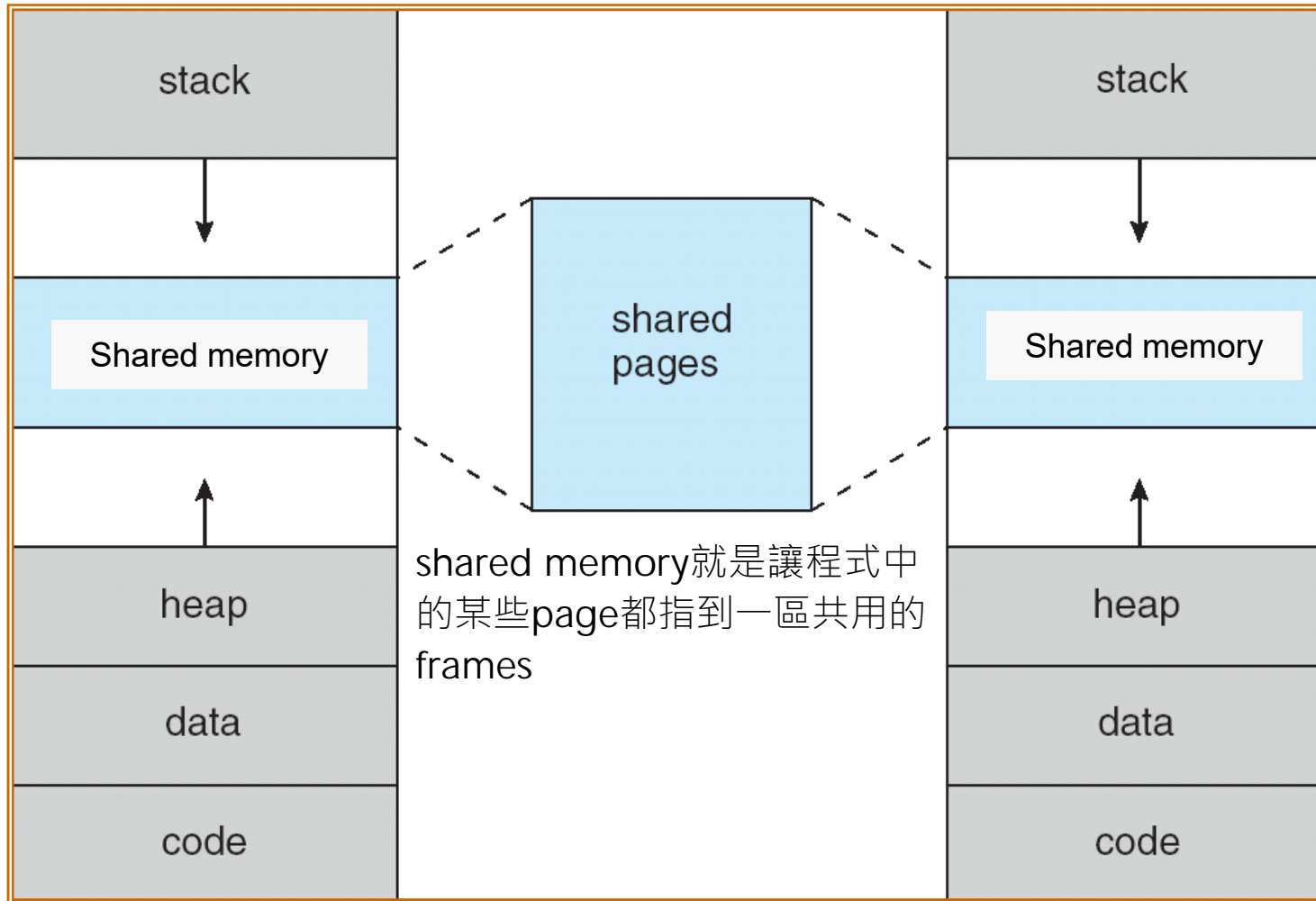
Shared Pages

- Shared code  dll · 放在某些pages · 所有process共用這些pages
 - Dynamic linking-loading libraries, kernel code, etc
 - Shared code must appear in same location in the logical address space of all processes
- Shared memory
 - Created via shmget()

Sharing of code in a paging environment



Shared Memory Using Virtual Memory



Copy-on-Write

child 一開始會完全複製 parent 的 page table

→兩者就看到一樣的東西

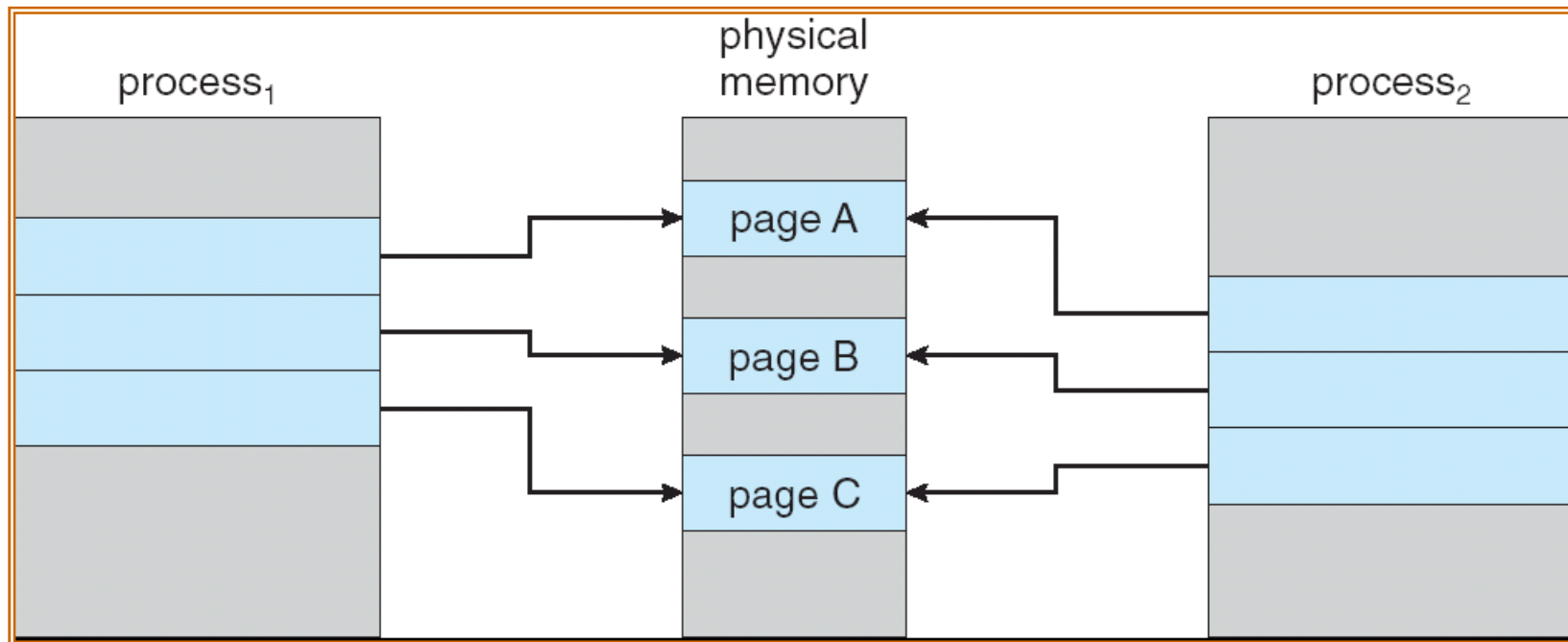
同時對 page readonly

→process 想要寫到 page 時，會觸發trap，另外複製一份該page 來修改

這是一種實作方式

- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
 - If either process modifies a shared page, a copy of the page is then created
 - Write to the page cause an exception
- COW allows more efficient process creation as only modified pages are copied
- A RO/RW bit for each page is necessary to trap writes to shared page

這些都需要MMU(比較貴的microprocessor)才有
page table, Copy-on-Write etc.



fork() and COW

- 想省掉複製整份
parent的時間
- fork() uses copy-on-write
 - vfork() does not use copy-on-write. It suspends the parent process and let the child process use the memory pages of the parent
 - The parent and child share anything but their stacks
 - The child reuses the parent's stack
 - The child calls exec() immediately
 - vfork() is for CPUs without a MMU. Otherwise, fork() with COW is efficient enough

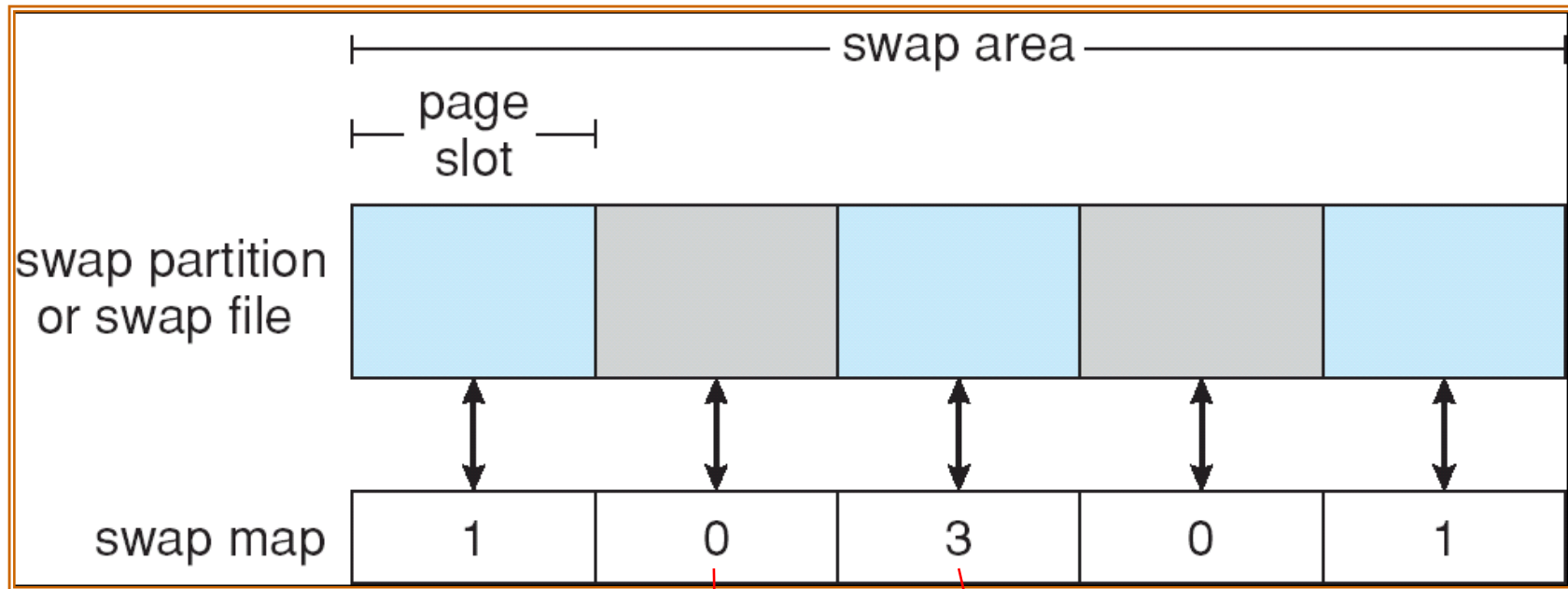
Swap Space Management & Memory-Mapped Files

是給 anonymous memory(runtime創建的array等)使用，讓這些東西能從 frames中被替換到swap-space，等待下一次page hit。不然一般檔案中也沒有他們對應的檔案，無法被replaced掉。可以更好的增進memory的使用。

Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory
- Swap-space can be carved out of the normal file system, or, more commonly, it can be in a separate disk partition 能獨立於一般的檔案系統之外，甚至不同的磁碟分割
- Swap-space management
 - Kernel uses swap maps to track swap-space use
 - 4.3BSD allocates swap space when process starts; holds text segment (the program) and data segment
 - Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created

The Data Structures for Swapping on Linux Systems



0表示沒在用

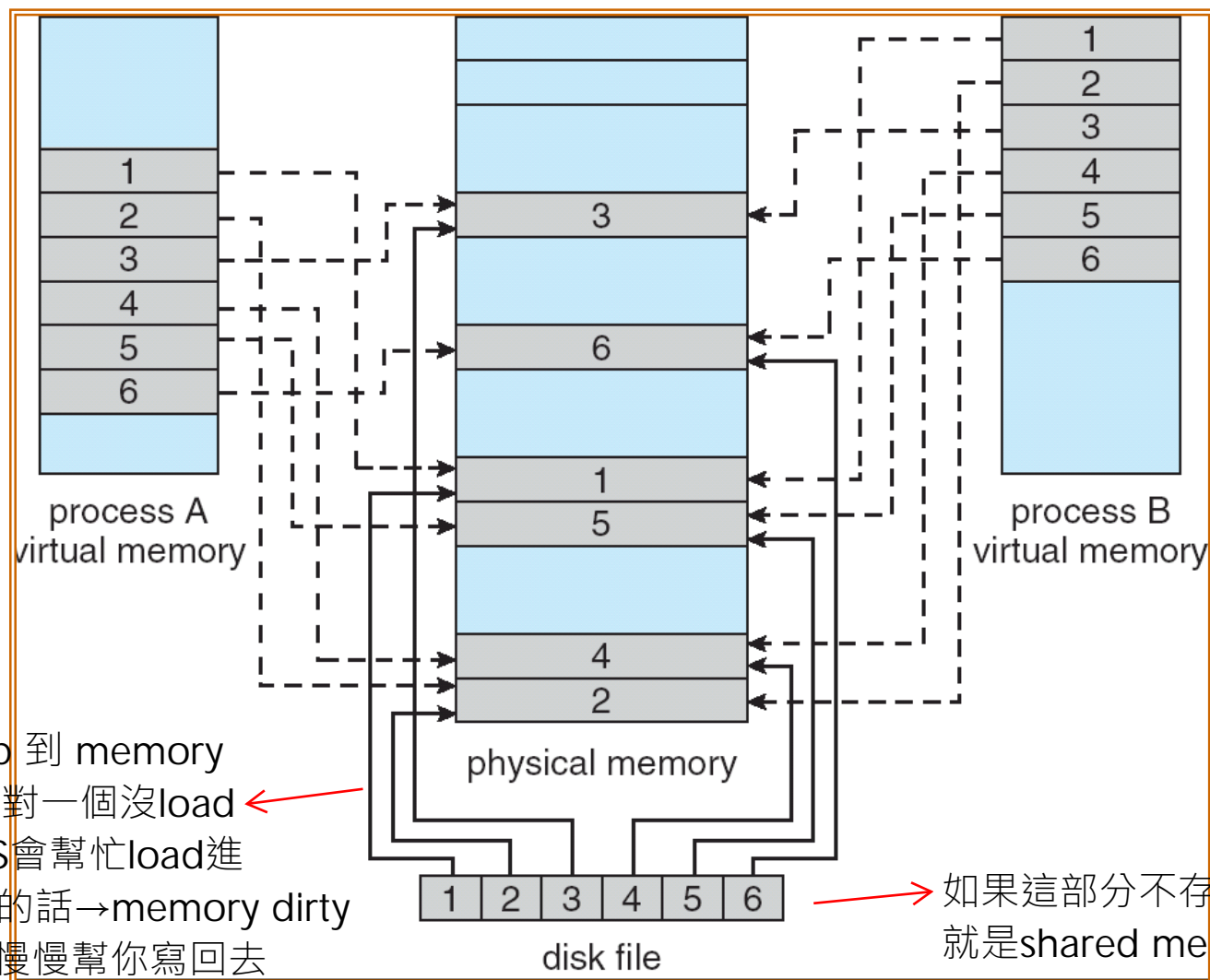
數字表示有幾個
processes在用這塊
page

Memory-Mapped Files

簡單來說就是不需要system call的 file access

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than read() write() system calls
 - Very convenient to implement huge data structures!!
 - No user-kernel mode switch
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Sharing a NULL file equals to SHM

Mapping of pages

- Anonymous pages belong to memory segments in processes, such as data, stack, and code segments, and they are mapped to the swap space
- File mapped pages are directly mapped to files in file system
- A process has both anonymous pages and file-mapped pages
- They work on the same mechanism for paging in and out, but the destination files are different (regular files and swap file)

SWAPPING

Swapping

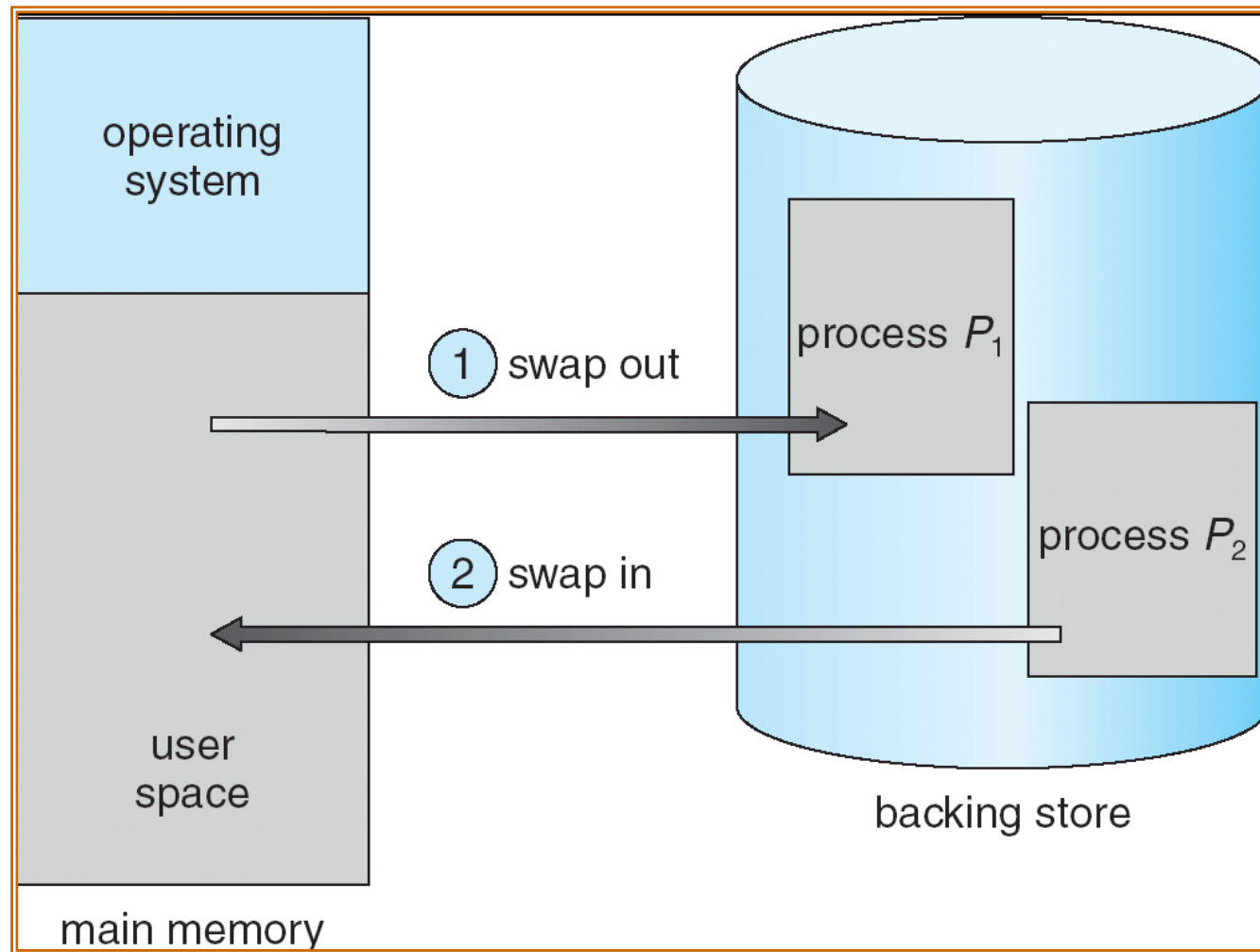
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Swap out, swap in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found in many systems (i.e., UNIX, Linux, and Windows)

成比例的

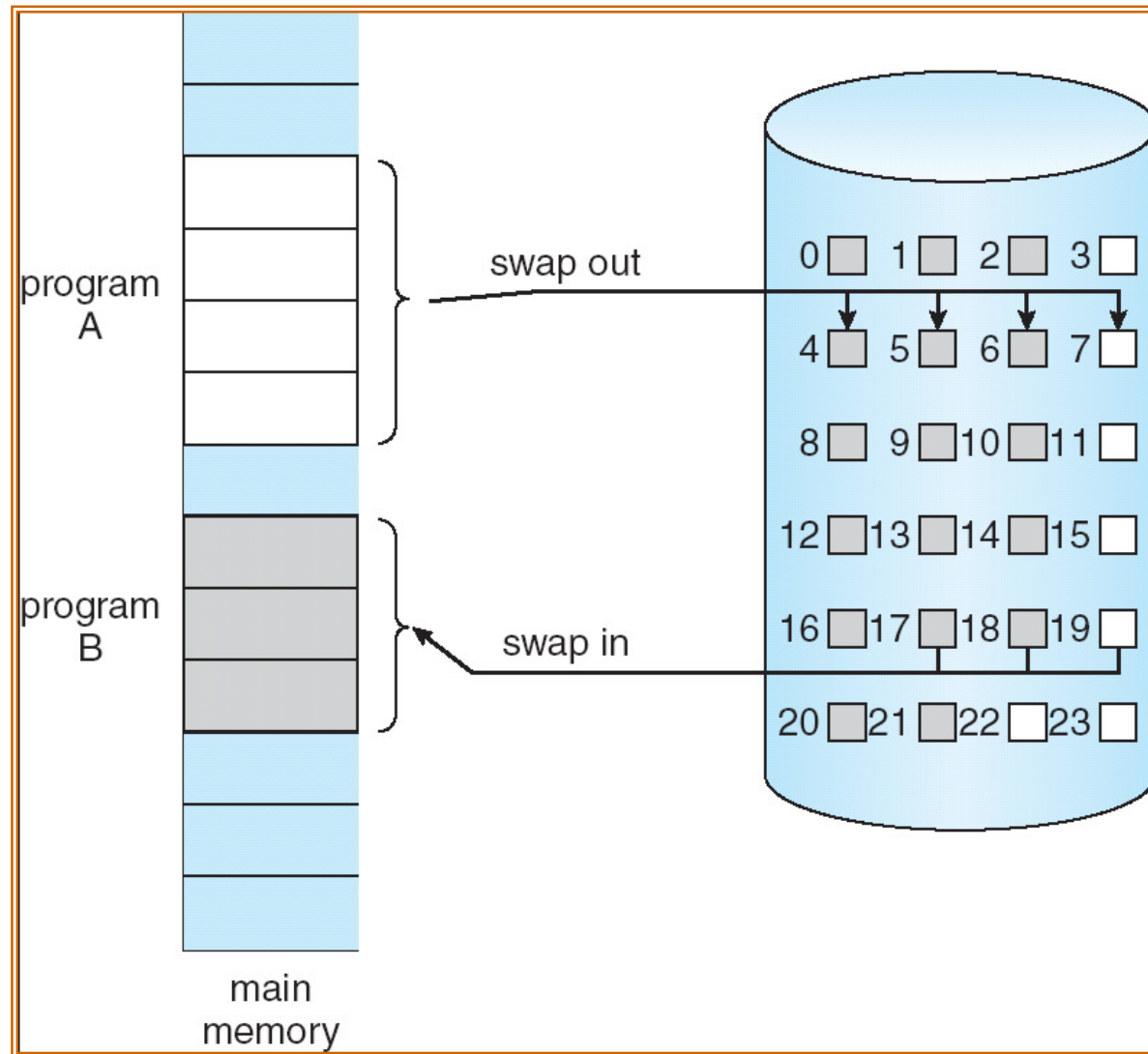


圖解的

Schematic View of Swapping



Transfer of a Paged Memory to Contiguous Disk Space



Swapping

- Consider that the disk transfer rate is 40MB/s, and the average disk seek overhead is 8 ms. To swap out a 10-MB process and then to swap in a 10-MB process require

$$10,000\text{KB}/40,000\text{KB} = 250\text{ms}$$

$$(250\text{ms} + 8) * 2 = 0.516\text{s}$$

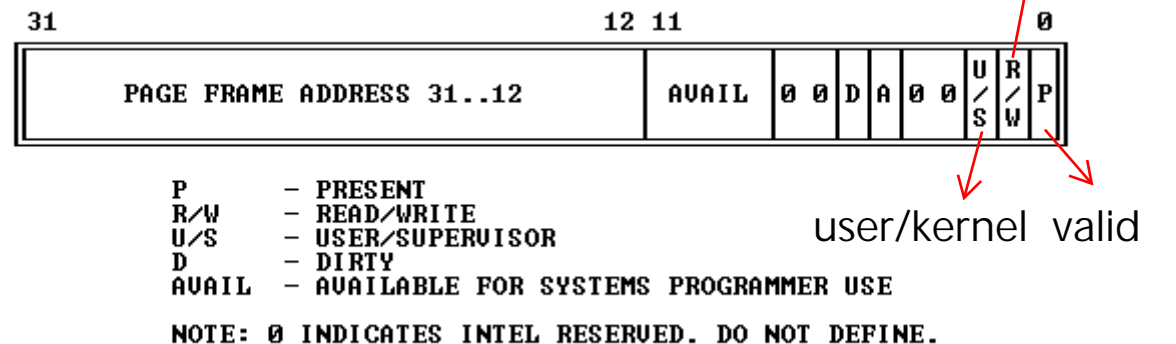
- The time quantum of a typical time-sharing system is 10ms; much smaller than this number
- Avoid swapping out an active process
- A process can be swapped out only if
 - It has been idle for a long time
 - It is not blocked in the kernel
 - Swapping out a process that is waiting for an I/O operation may crash the system
I/O 完成 interrupt 回來可能找不到 process

Operating System Examples

Status Bits Associated with a Page (x86)

- Valid (Present) bit: does the page present in main memory?
- RW bit: is the page read-only or read-write?
- Reference bit: is the page referenced?
- Dirty bit: is a page modified?

Figure 5-10. Format of a Page Table Entry



PG_active : 連續hit兩次就會設起來→把用一次跟兩次以上的區分(隔出sequential)

PG_reference : 最近有用到(適當時機會清掉)

The Linux Page Replacement Algorithm

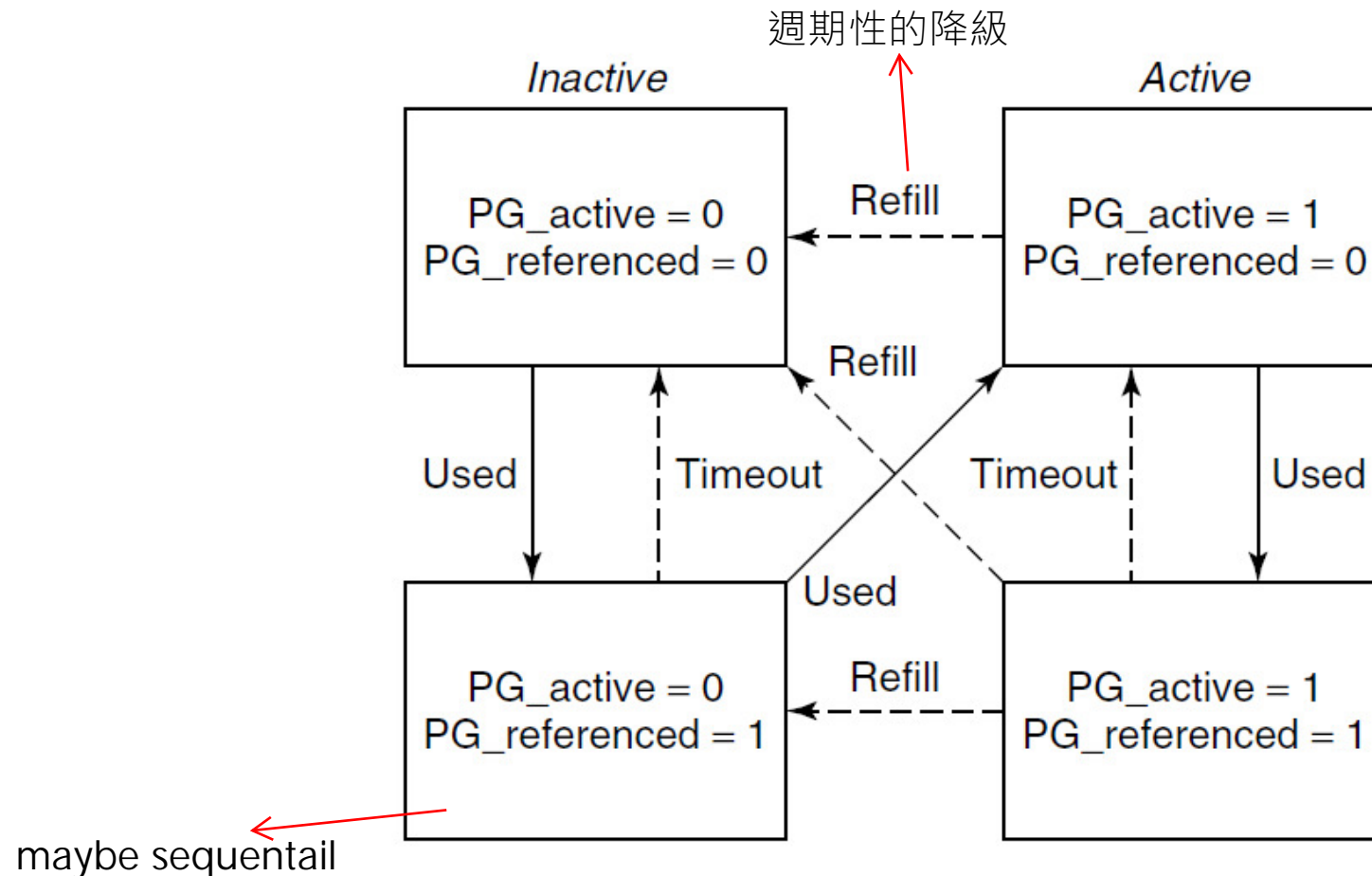



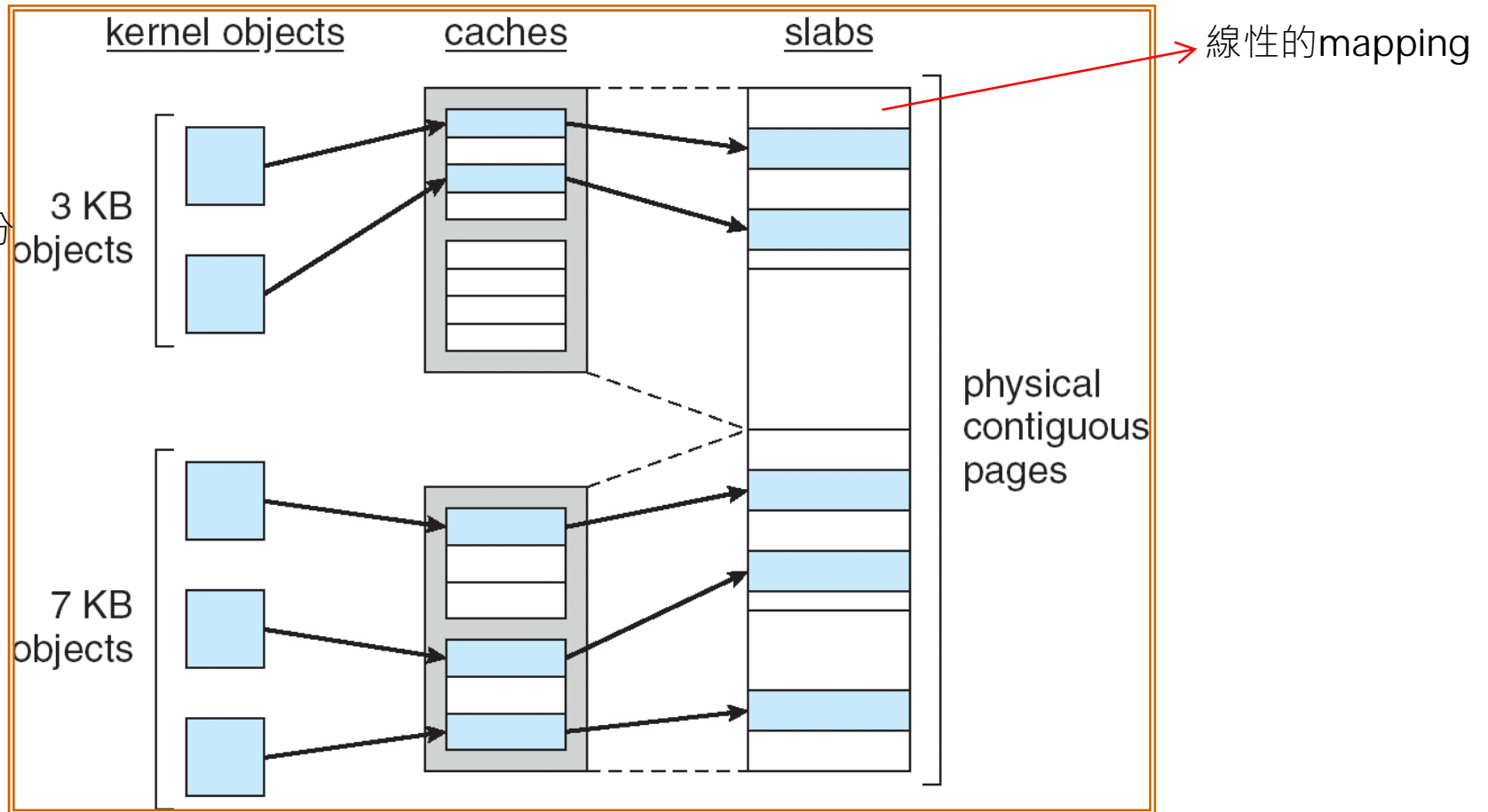
Figure 10-18. Page states considered in the page frame replacement algorithm.

Linux Kernel Memory Allocation

- Characteristics of memory usages in the kernel
 - IO devices cannot refer to information in MMU Direct Memory Access
 - **memory should be physically contiguous** because of I/O is 
not aware of paging kernel裡面配的記憶體很多都是拿來給I/O的→DMA操作，看不懂
 - Small objects are frequently allocated/deallocated page table(因為在CPU外)
→必須物理連續
 - **Fragmentation is a vital issue**
kernel通常裡面有很小的物件，配置/釋放的頻率很高，大小又不一
→很容易造成 Fragmentation

Linux kernel slab cache

會把同樣大小(類型)的物件，用同一pool來分配→解決破碎問題



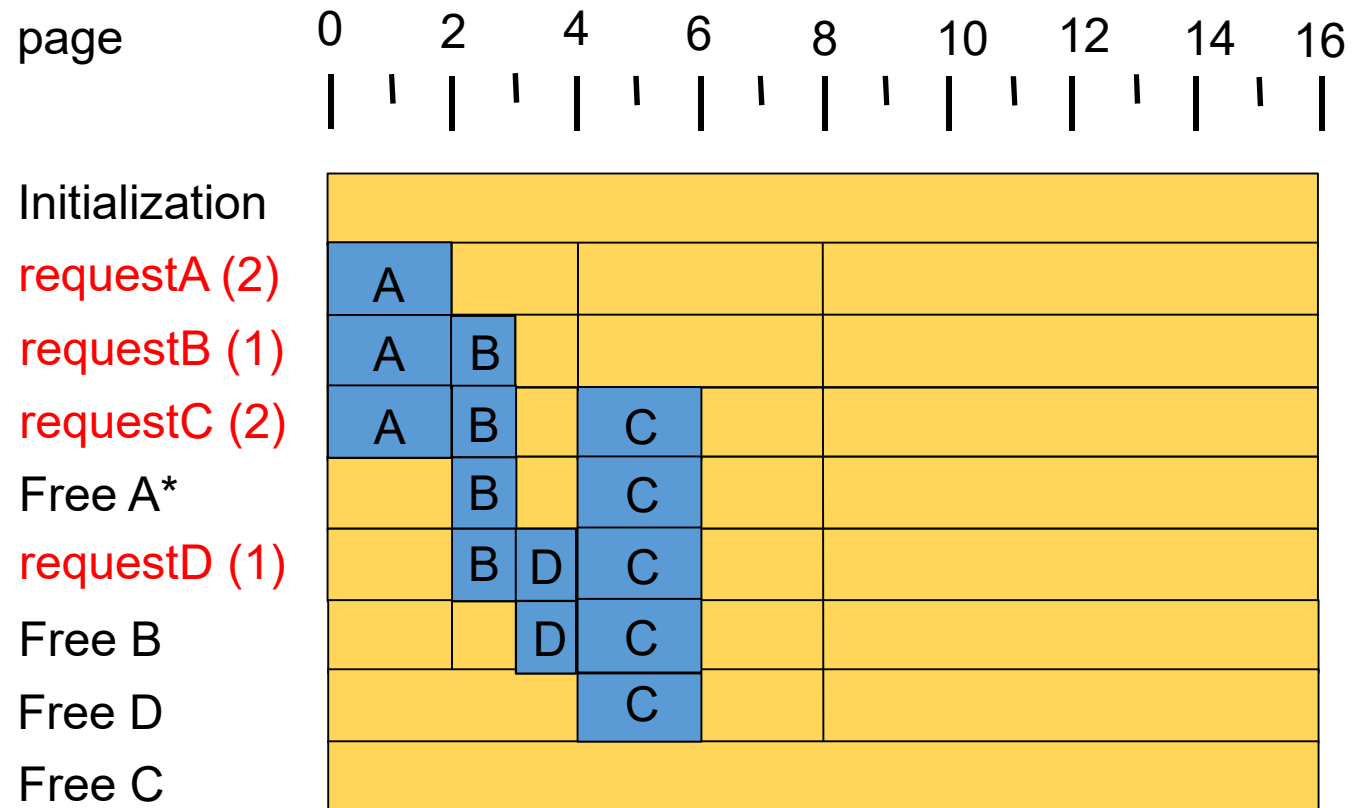
Benefits:

- No fragmentation
- Quick memory allocation
- Objects get physically contiguous memory

[cache[obj]][obj][obj]]
[page][page][page]

Buddy System

因為找東西速度非常快
常用於kernel



Slab Cache and Buddy System

- Both exist in the Linux kernel
- Frequent allocation/deallocation of objects of the same size, use slab cache 同一類型的小東西，就用同一片cache→沒有破碎問題
- Allocation/deallocation of objects of various sizes, use buddy system

End of Chapter 9