

政大資科系

# 作業系統

Operating System

廖峻鋒

cfliao@nccu.edu.tw

Operating System

# Virtual Memory

Chun-Feng Liao

廖峻鋒

Department of Computer Science

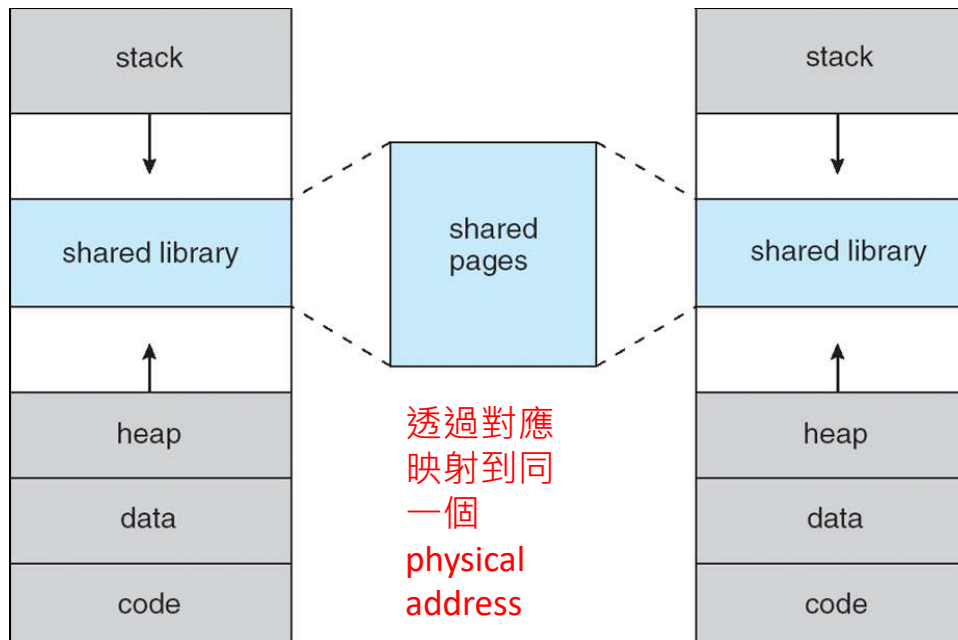
National Chengchi University

# Motivation

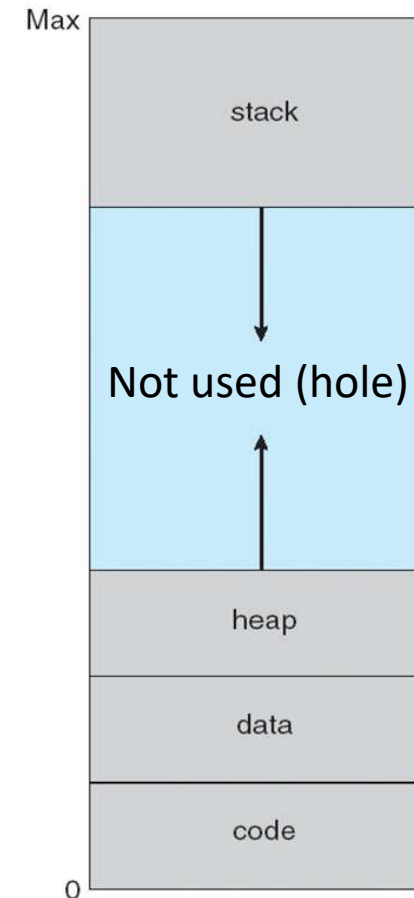
- Entire program rarely used at the same time
  - Many code for handling unusual errors or conditions
  - Certain program routines or features are rarely used
  - Arrays, lists and tables allocated but not used
- Execute partially-loaded program
  - Each program takes less memory → more programs run at the same time → increase utilization

# Motivation

- Use cases for Virtual-Physical address mapping
  - Shared library
  - IPC (shared memory)
  - Fork() copy on write
    - Process的一部份放在shared page中
    - Speeding up process creation



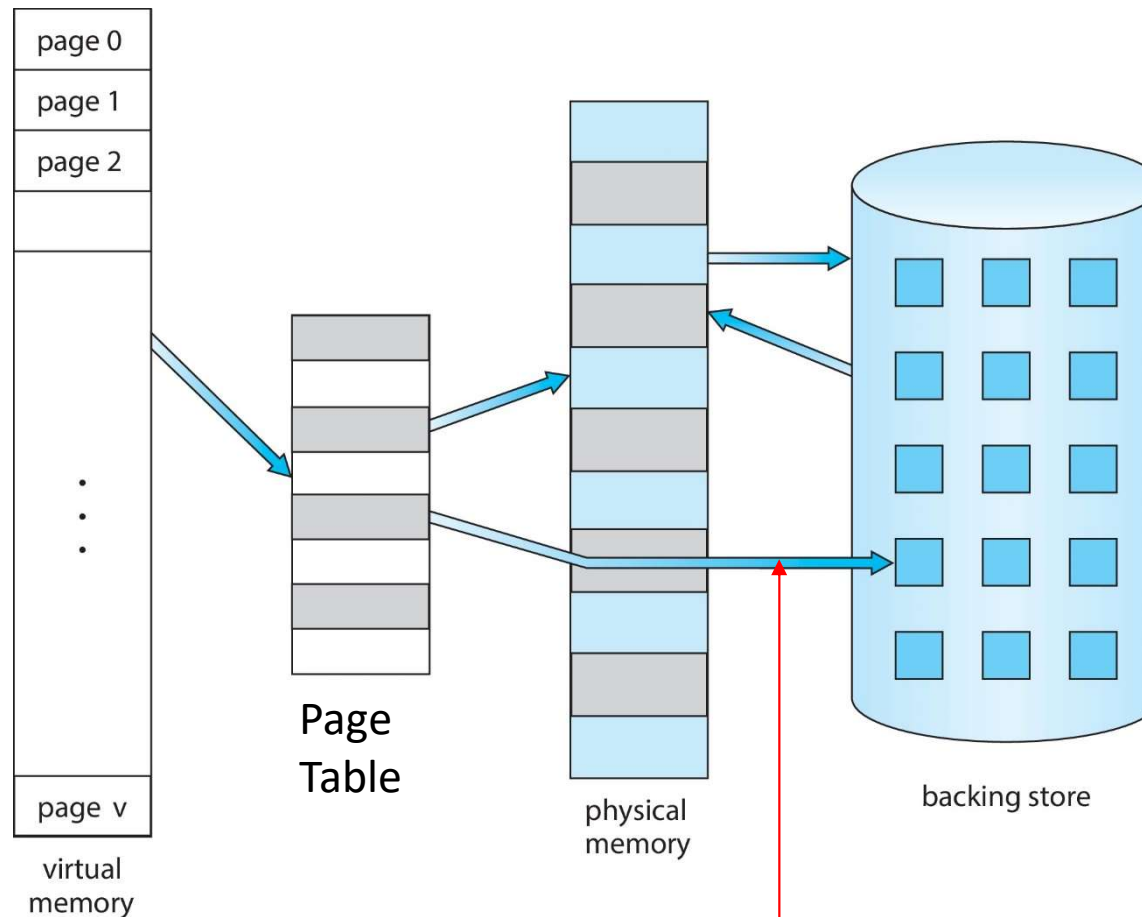
Virtual address of a process



# Virtual Memory

- Separating logical memory and physical memory
  - To run an extremely large process
    - Logical address space can be much larger than physical address space
  - To increase CPU/resources utilization
    - higher multiprogramming degree
  - To simplify programming tasks
    - Free programmer from memory limitation
  - To start programs faster
    - less I/O would be needed to load or swap (因為只載入process的一部份)

# Virtual Memory > Physical Memory



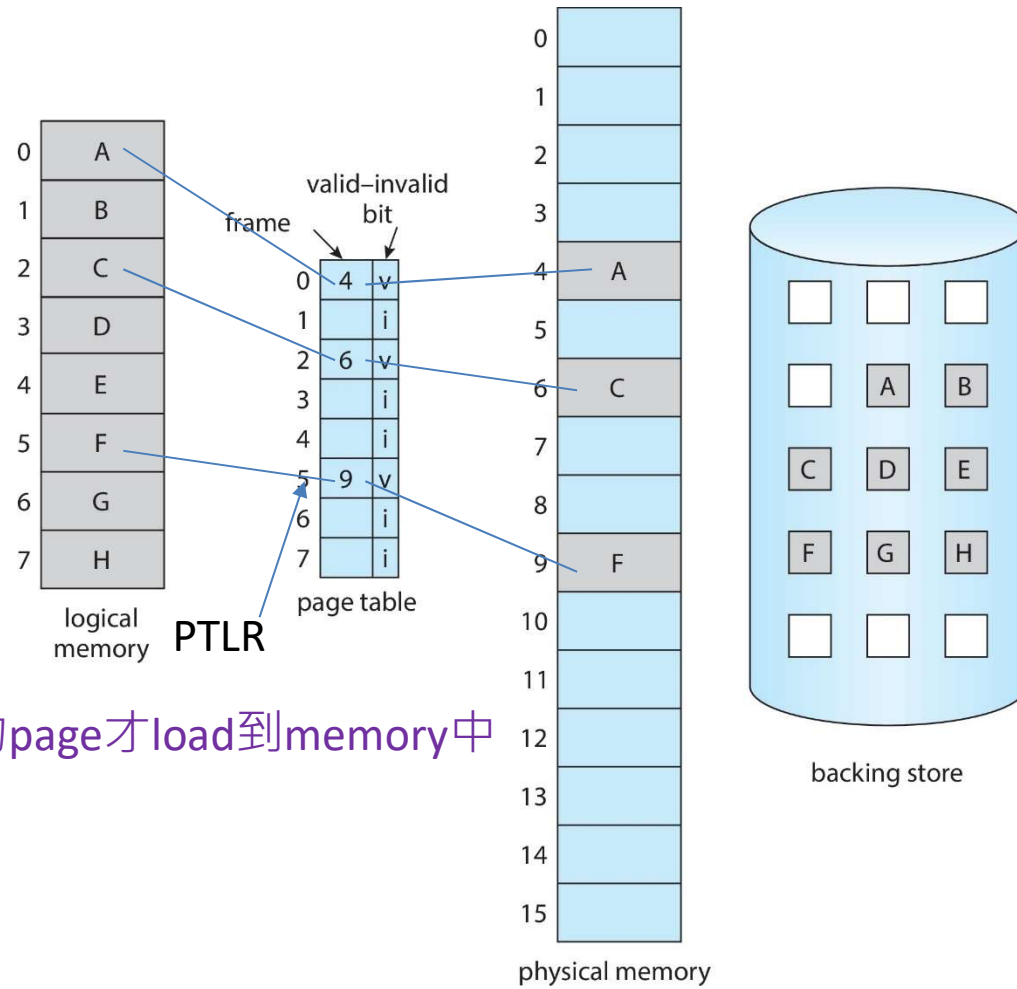
有些virtual memory space事實上存在disk上  
但應用程式沒感覺到

# Demand Paging

Core idea: Page用到時才載入

- A page is brought into memory **only when it is needed**
  - Less I/O needed → Faster response
  - Less memory needed → More users
- Page is needed when there is a reference to the page
  - Illegal reference → abort
    - Invalid and >PTLR
  - Not-in-memory → bring to memory via paging
    - Invalid and ≤PTLR
- Pure demand paging
  - Start a process with no page
  - Never bring a page into memory until it is required

# Demand Paging

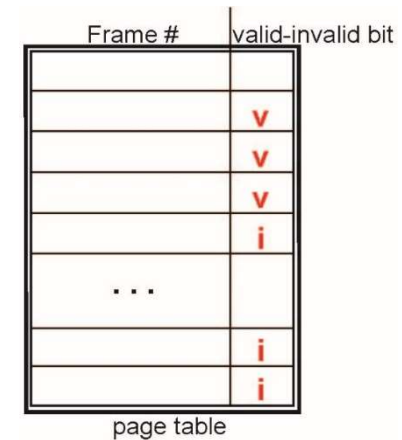


用到的page才load到memory中



# Demand Paging

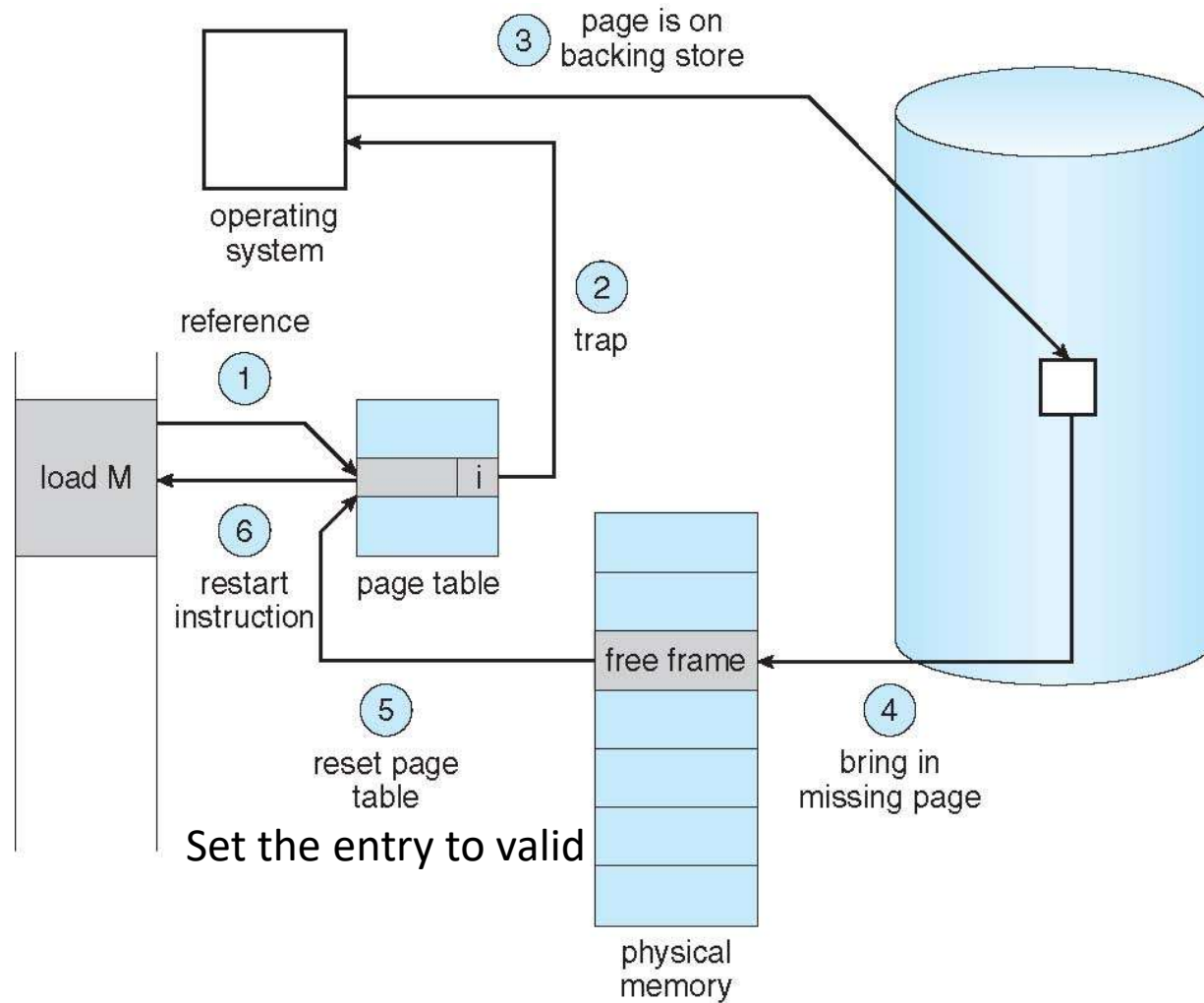
- Demand Paging和Swapping不同
  - Swapper manipulates the entire process
  - Pager is concerned with the individual pages of a process
- Hardware support P.395中間
  - Main Memory (Page Table: a valid-invalid bit)
    - 1 valid → page in memory
    - 0 invalid → page not in memory
      - 可能illegal或只是不在mem; 視PTLR而定
    - Initially, all such bits are set to 0
  - Secondary memory (swap space, backing store):
    - Usually, a high-speed disk (swap device) is used



# Page Fault Process

- First reference to a page will trap to OS
  - page-fault trap
- 1. OS looks at the internal table (in PCB) to decide
  - Illegal → abort (違規存取)
  - just not in memory → continue
- 2. Get an empty frame (from free frame list)
- 3. Read the page from disk into the frame
- 4. Reset the page table entry: set to valid (1)
- 5. Restart instruction

# Handling a Page Fault



# Demand Paging Performance

- Three major task components of the page-fault time
  1. Service the page-fault interrupt
  2. Read in the page (from disk)
  3. Restart the process
- 1 and 3 can be reduced with careful coding
- 2 is the most time consuming part

# Demand Paging Performance

存取記憶體      存取記憶體失敗+Page-in

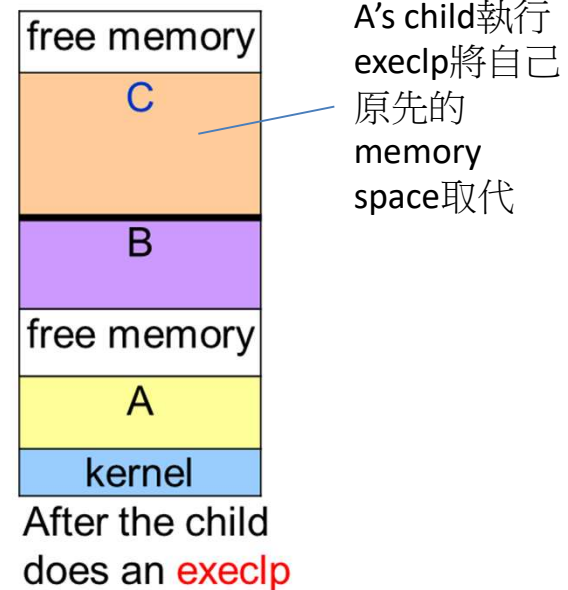
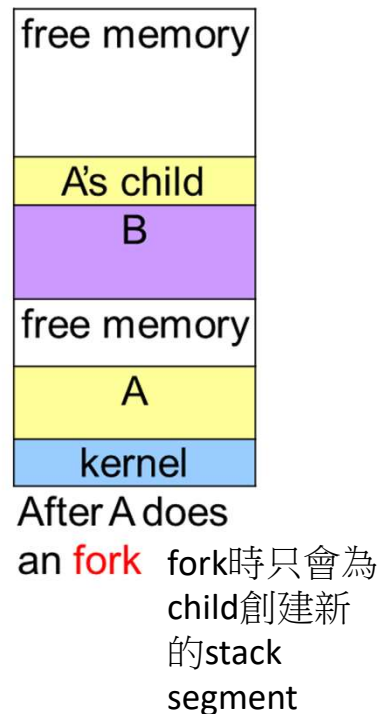
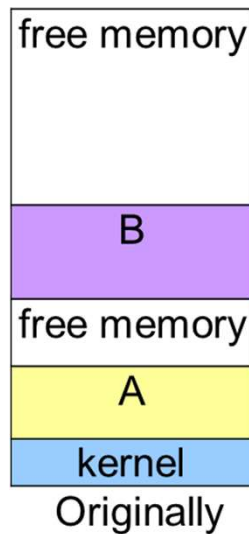
- Effective Access Time (EAT):  $(1 - p) \times ma + p \times pft$ 
  - $p$ : page fault rate,  $ma$ : mem. access time,  $pft$ : page fault time
- Example:  $ma = 200ns$ ,  $pft = 8ms$ 
  - $EAT = (1 - p) * 200ns + p * 8ms$   
 $= 200ns + 7,999,800ns \times p$      $8ms = 8000000 \text{ ns}$
- Access time is proportional to the page fault rate
  - If one out of 1,000 causes a page fault, then  
 $EAT = 8200ns \rightarrow$  slowdown by a factor of 40!     $200ns \text{ vs. } 8200ns$
  - For degradation less than 10% ( $220ns$ ): 確保  $EAT < 220ns$   
 $220 > 200 + 7,999,800 \times p$ ,  $p < 0.0000025$   
one access out of 399,990 to page fault

# Copy-on-Write

- COW allows efficient process creation (e.g., `fork()` )
- Allows both parent and child processes to initially share the same pages in memory
  - Not copy until a modification on the child process

# Copy-on-Write

- Memory space of fork():
  - Old implementation: A's child is an exact copy of parent
  - Current implementation: use **copy-on-write** technique to store differences in A's child address space



# Copy-on-Write

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

```
void main( )
```

```
{
```

```
    int A;
```

```
    /* fork child process */
```

```
    A = fork( );
```

```
    if (A == 0) {
```

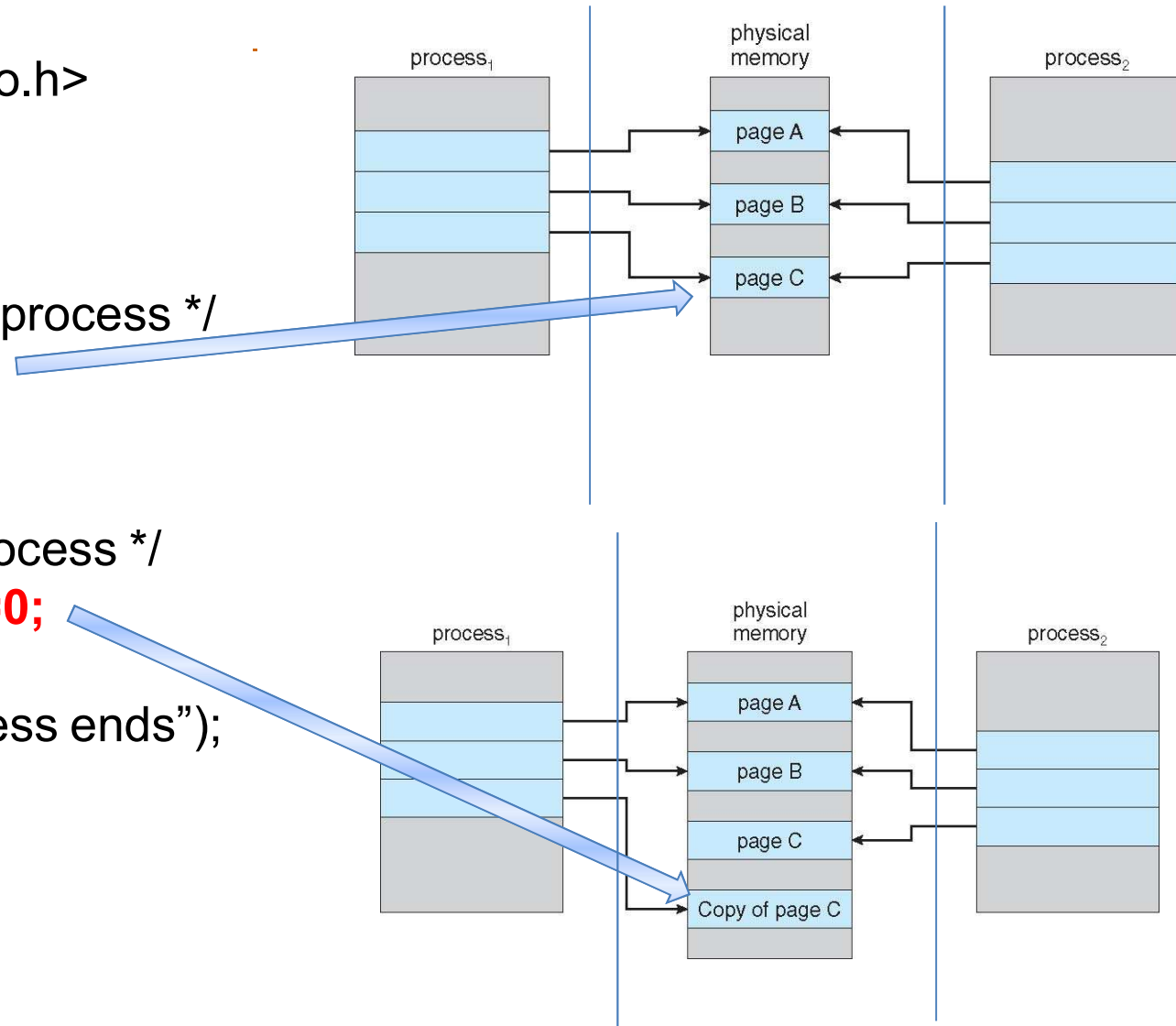
```
        /* child process */
```

```
        int test1=0;
```

```
    }
```

```
    printf("process ends");
```

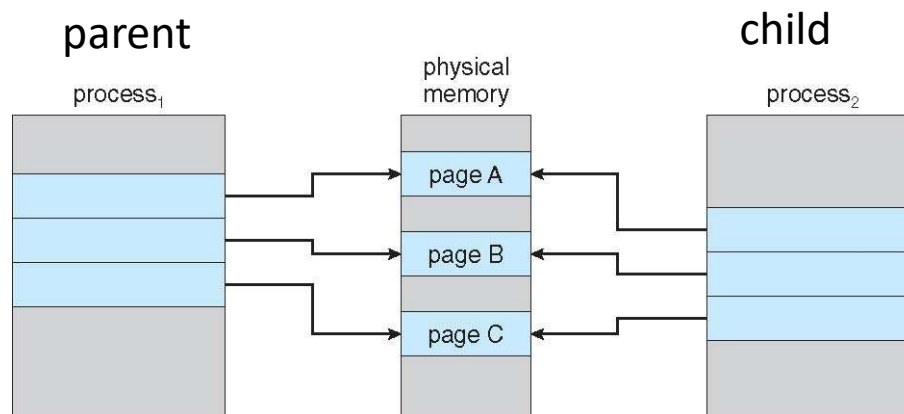
```
}
```





# 另一種做法: vfork()

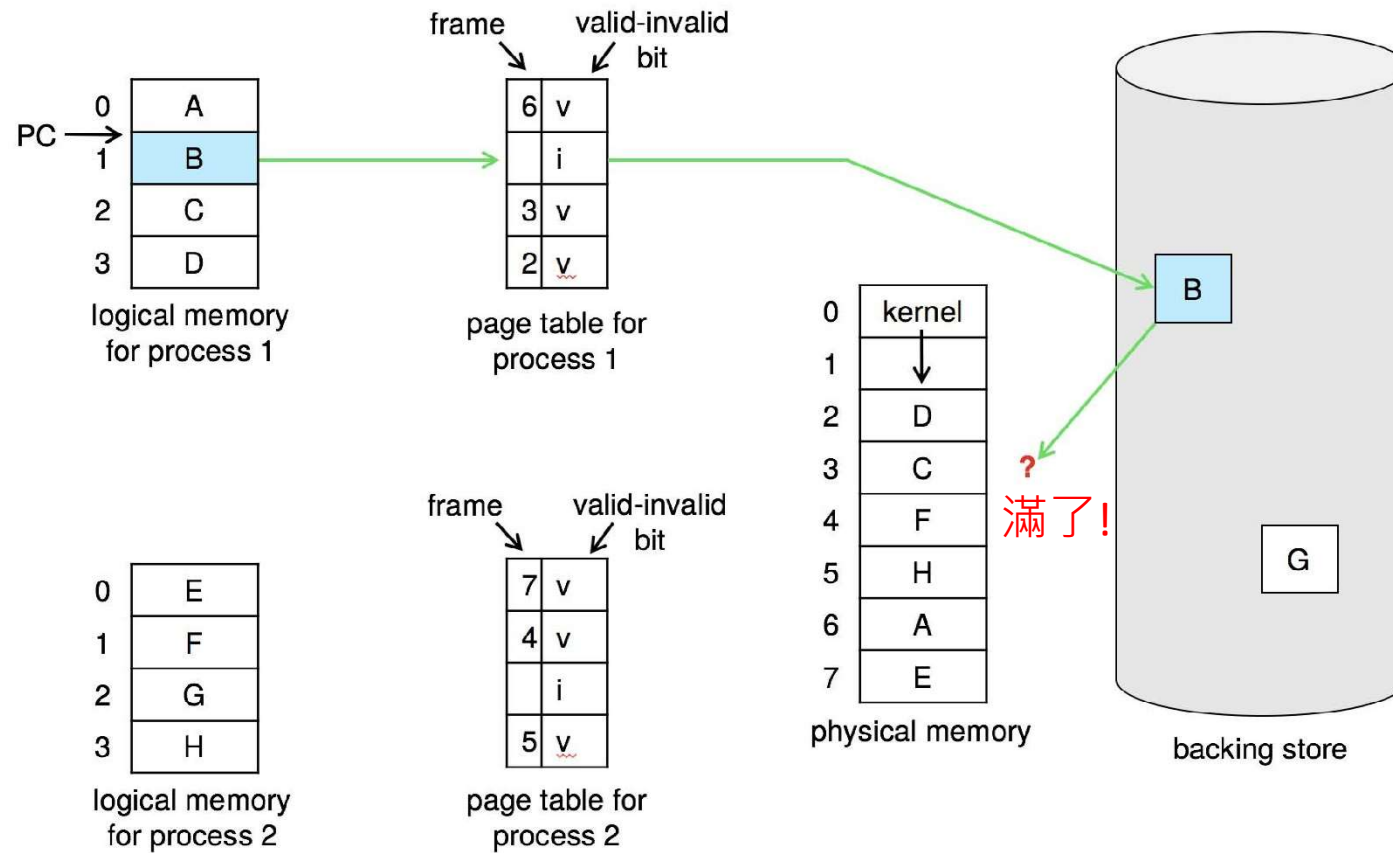
- vfork(): Virtual Memory Fork
  - Child share memory with the parent
  - Parent suspends after vfork()
  - No copy-on-write: **modification visible to the parent**
  - No memory copying, extremely efficient
  - Mainly used to implement shell (why?)



# Case: Windows 10

- Windows 10 implements most of the memory-management features described thus far, including
  - shared libraries, demand paging, and copy-on-write

# Need For Page Replacement



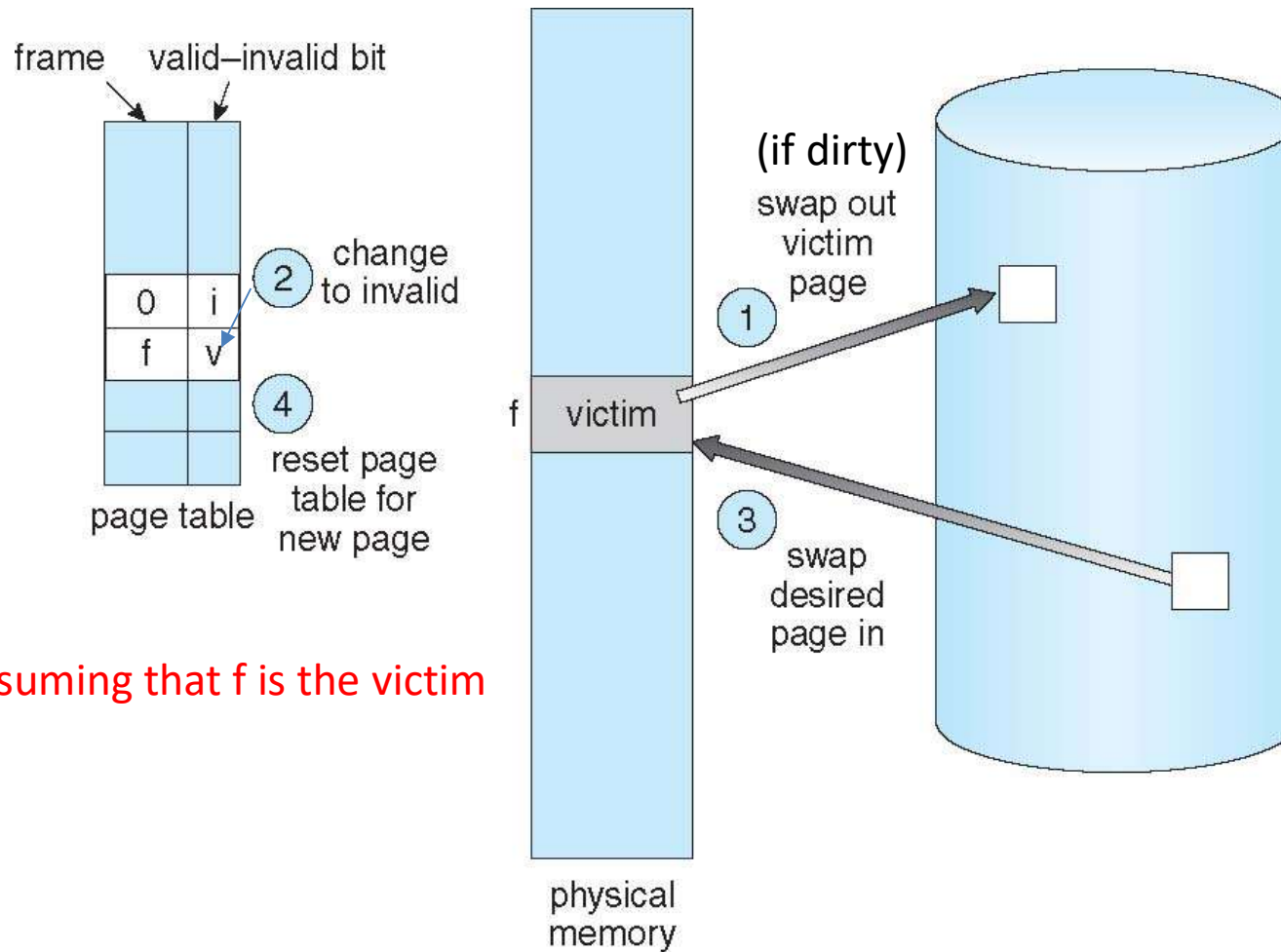
# Page Replacement

- When a page fault occurs and no free frame
  - Swap out a process: freeing all of the process's frames
  - Page replacement: find one not currently used and free it
    - Write back: use dirty bit to reduce overhead of page transfers – only modified pages are written to disk (有被改過的page，才需要回寫)
- Two issues
  - Page-replacement algorithm:
    - Select which frame to be replaced
  - Frame-allocation (section 10.5, P.413):
    - Determine how many frames to be allocated to a process (initially)

# Page Replacement (Page Fault) Steps

1. Find the desired page **on disk**
2. Find a free frame **in memory**
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim** frame
  - Write back the victim frame if dirty
3. Read the desired page into the (newly) free frame
4. Update the page & frame tables
5. Resume the process

# Page Replacement



Assuming that f is the victim

# Page Replacement Algorithms

- Goal
  - Lowest page-fault rate
- Evaluation
  - Test with reference string
  - Compute the number of page faults

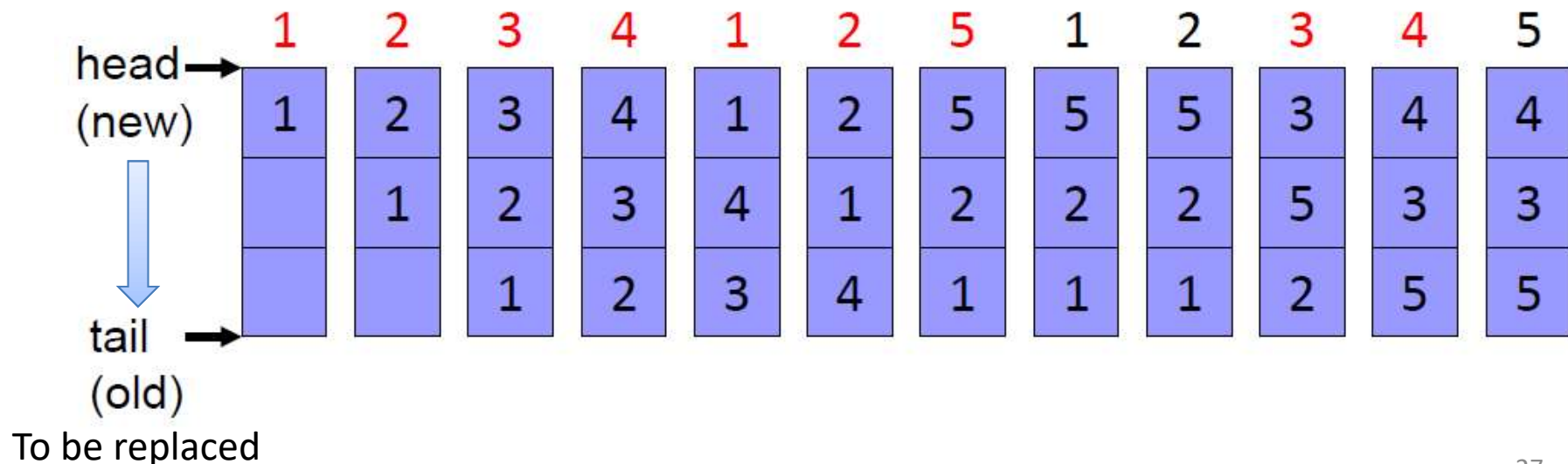
# Replacement Algorithms

- FIFO algorithm
- Optimal algorithm
  - 取代掉未來最久不會用到的page
- LRU algorithm
  - 取代掉最久沒被用的 (根據歷史資料)
- Counting algorithm
  - LFU (Least frequently used) 合計被用到最少次
  - MFU (Most frequently used) 合計被用到最多次

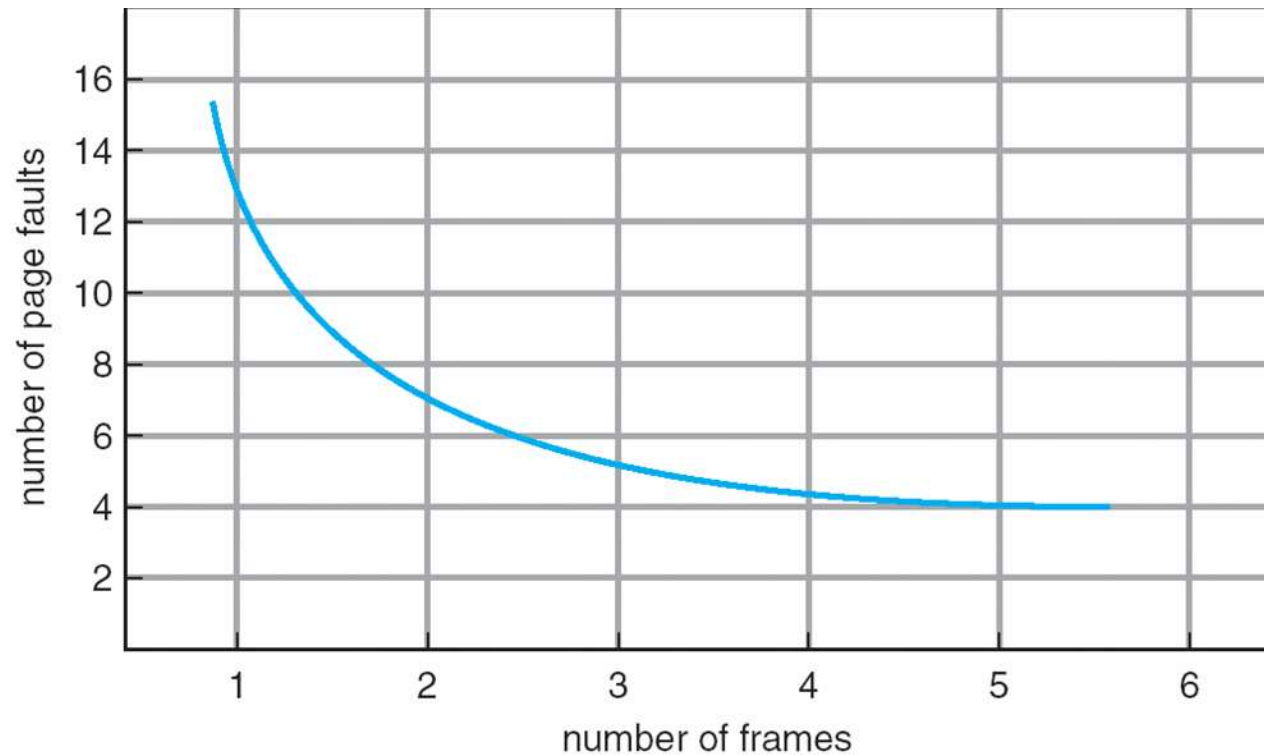


# First-In-First-Out (FIFO) Algorithm

- The oldest page in a FIFO queue is replaced
  - Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
- 3 frames (available memory frames = 3)
  - 9 page faults



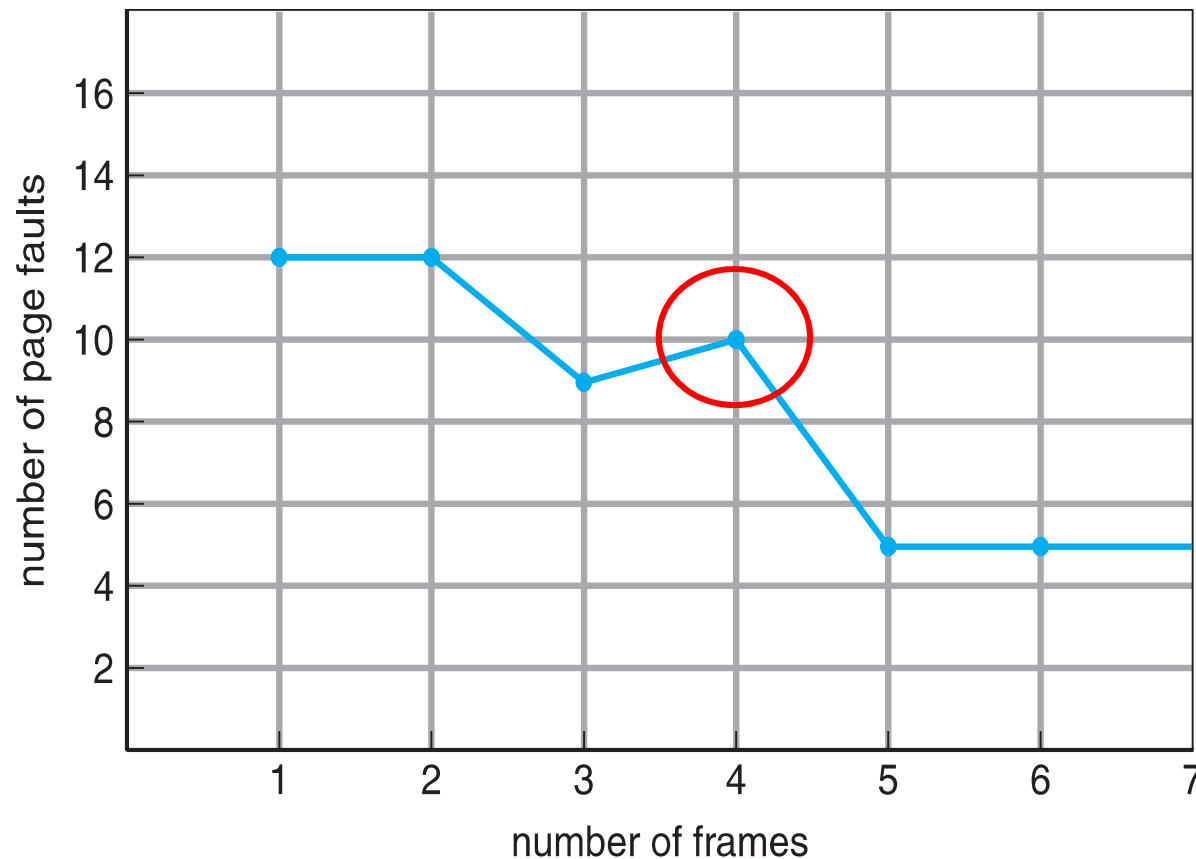
## (理論上的) Graph of Page Faults vs the Number of Frames



實體記憶體愈充足，page fault理論上要愈少

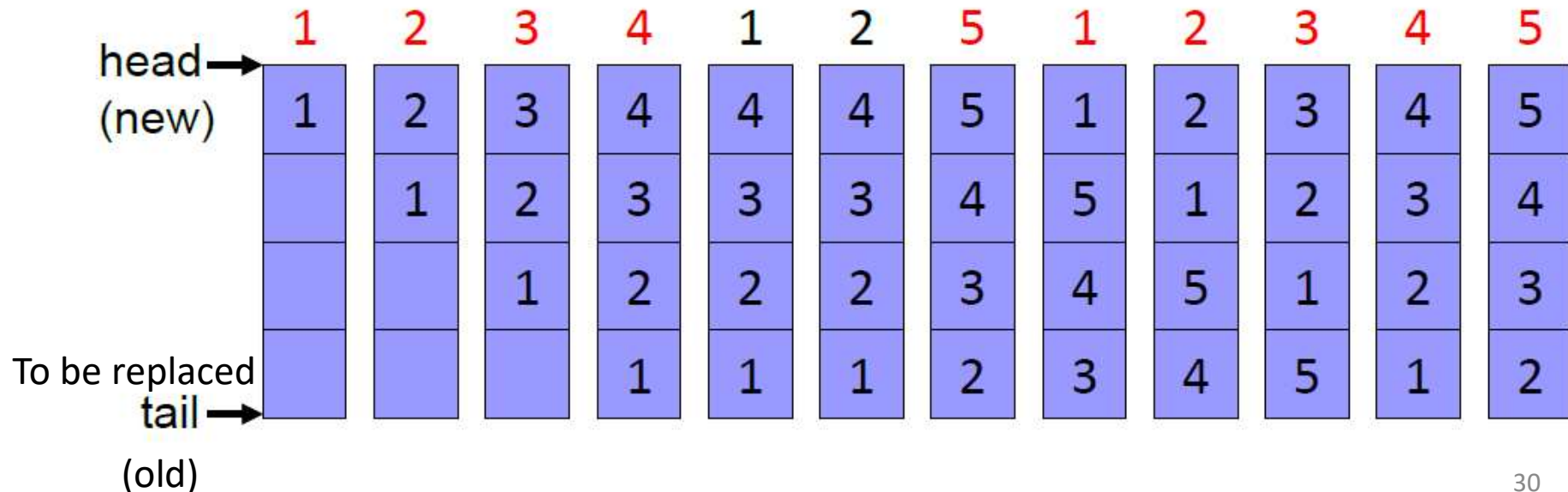
# FIFO Illustrating Belady's Anomaly

想像: cache的entries 愈多，理論上應該hit rate愈高，但FIFO某些情況下，entries多反而會降低hit rate!



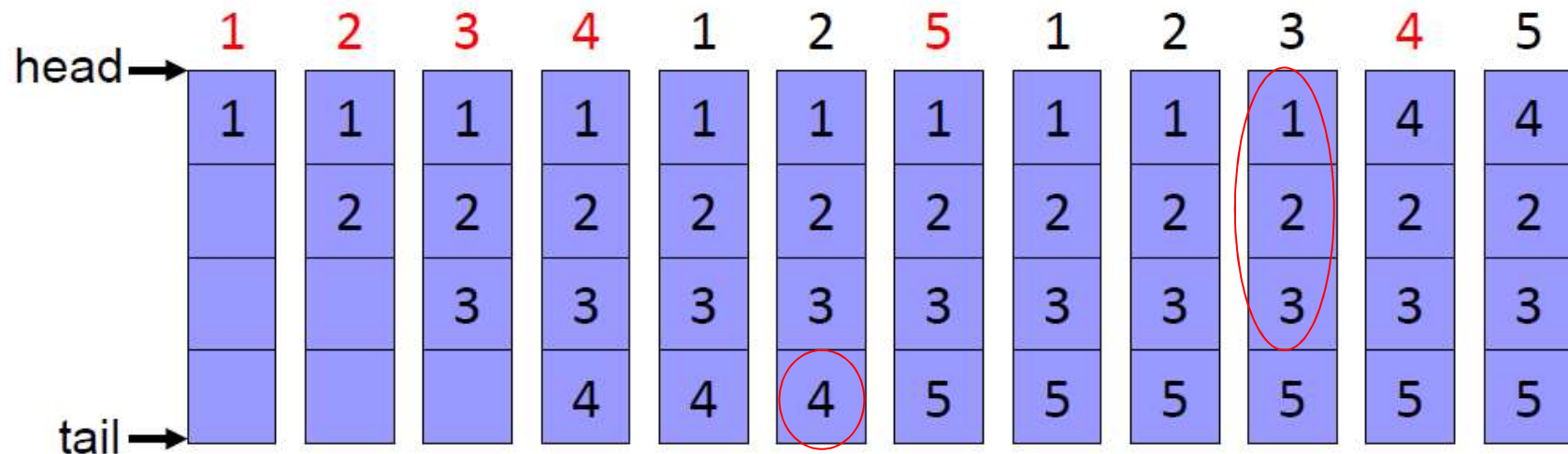
# FIFO Illustrating Belady's Anomaly

- Does more allocated frames guarantee less page fault?
  - Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
  - 4 frames (available memory frames = 4)
  - 10 page faults!
- Belady's anomaly
  - Greater allocated frames → more page fault



# Optimal (Belady) Algorithm

- Replace the page that **will not be used for the longest time**
  - need future knowledge
- 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 → 6 page faults!
- In practice, we don't have future knowledge
  - Only used for reference & comparison 實務上無法實現!



# LRU Algorithm

## (Least Recently Used)

- An approximation of optimal algorithm:
  - Looking backward, rather than forward
    - 直覺意義: 過去反映未來
  - Replaces the page that has not been used for the longest period of time
  - Often used, and is considered as quite good

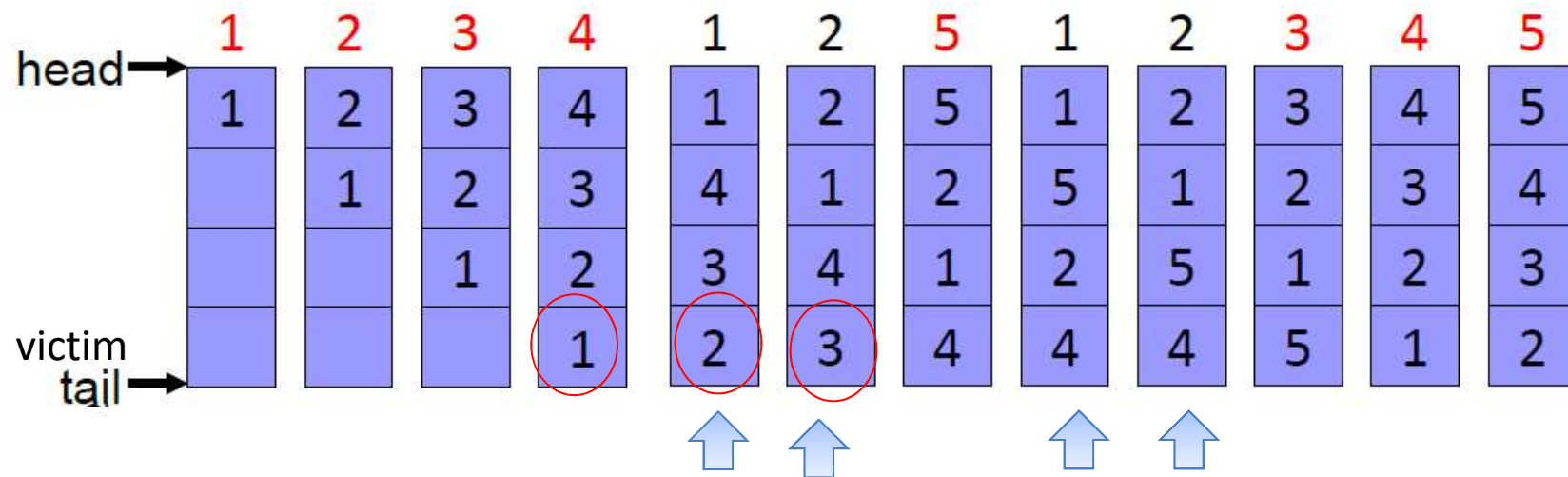
# LRU Implementations

- Counter implementation
  - 紀錄最近被用的時間
  - page referenced: timestamp is copied into the counter
  - replacement: remove the one with oldest counter
    - linear search is required
- Stack implementation
  - page referenced: move to <sup>new</sup> **top** of the double-linked list
  - replacement: remove the page at the **bottom** <sub>old</sub>

# LRU Example

- LRU

– 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 → 8 page faults!

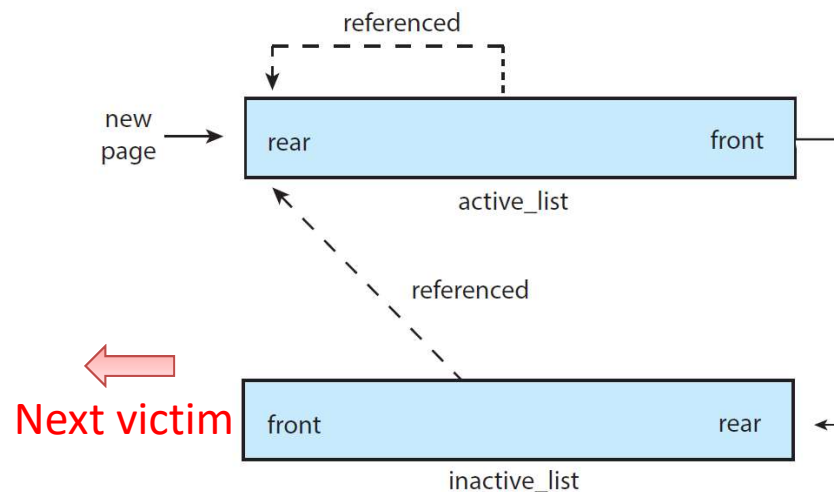


最近有被access的話就提升到queue head



# Case: Linux

- Linux maintains two types of page lists: an active list and an inactive list
  - The front of inactive\_list is the candidate of next victim
  - Initially a page is put to the rear of active\_list
    - Gradually, the page moves toward the front of inactive\_list
  - When referenced, the page is moved to the rear of active\_list



# Allocation of Frames

- 一個Process要配置多少記憶體?
- Fixed allocation
  - Equal allocation 平均分配
    - 100 frames, 5 processes → 20 frames/process
  - Proportional allocation 依需求
    - Allocate according to the size of the process
- Priority allocation
  - using proportional allocation based on priority, instead of size

# Frame Allocation

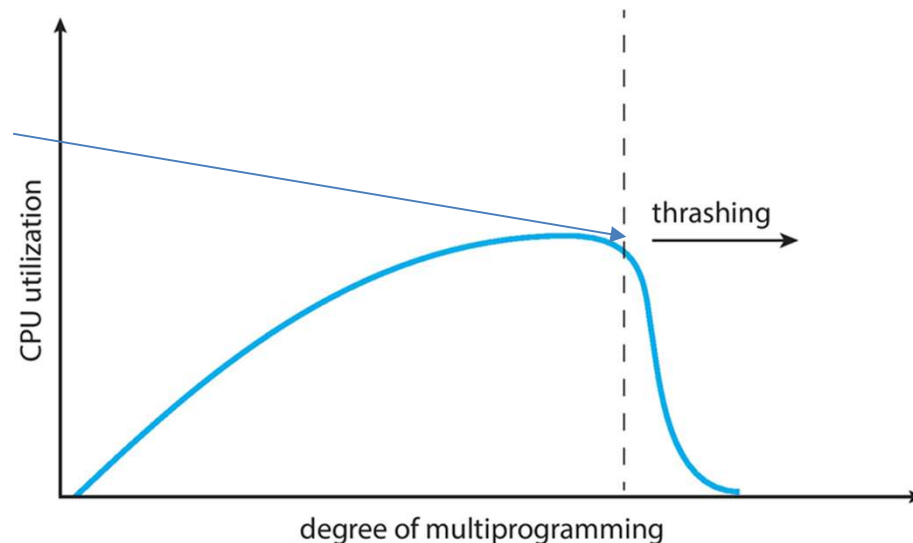
- Local allocation: each process select from its own set of allocated frames (victim只從自己現有的來選)
- Global allocation: process selects a replacement frame from the set of all frames (victim全部系統一起考量來選)
  - One process can take away a frame of another process
    - e.g., allow a high-priority process to take frames from a low-priority process
  - Good system performance and thus is common used
  - A minimum number of frames must be maintained for each process to prevent trashing

# Definition of Thrashing

- If a process does not have “enough” frames
  - the process does not have # frames it needs to support pages in active use
  - Very high paging activity
- A process is thrashing if it is spending more time on paging than executing

一個process所擁有的記憶體少到不足以執行任何有意義的工作

→ 花在放出載入分頁的時間比真正計算時間多



# Thrashing

- Performance problem caused by thrashing (Assume global replacement is used) 不良連鎖反應
  - processes queued for I/O (page fault)
    - low CPU utilization
    - OS increases the degree of multiprogramming
    - new processes take frames from old processes
    - more page faults and thus more I/O
    - CPU utilization drops even further
- To prevent thrashing, must **provide enough frames** for each process:
  - Working-set model, Page-fault frequency

確保一個process所擁有的記憶體至少足以向前執行

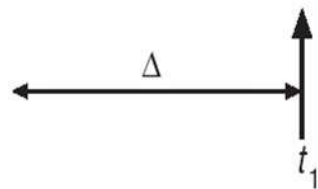
# Working-Set Model

- Locality: a set of pages that are actively used together
- 程式執行時有其固定的locality model
  - program structure (subroutine, loop, stack)
  - data structure (array, table)
- Working-set model (based on locality model)
  - working-set window: a parameter  $\Delta$  (delta)
  - working set: **set of pages** in most recent  $\Delta$  page references (an approximation locality)

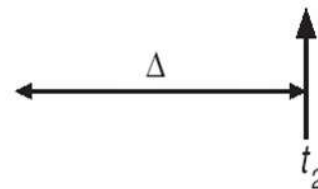
直覺意義: **working set**的size就是某一段時間內, **process**所需要的frames

page reference table  $\Delta=10$

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




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

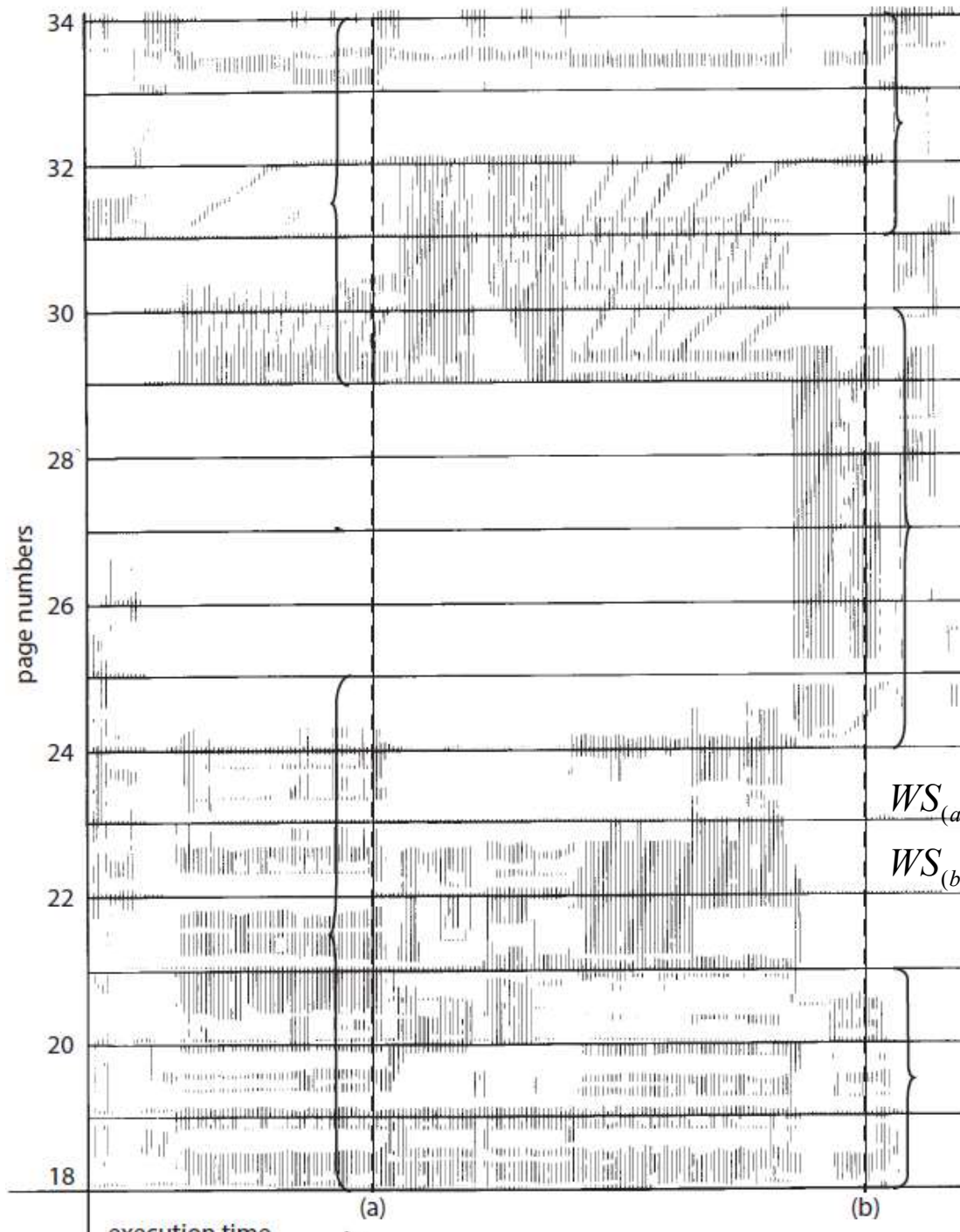


$WS(t_2) = \{3, 4\}$

# Working-Set Model

- Prevent thrashing using the working-set size
  - $WSS_i$  : working-set size for process  $i$   直覺意義: working set的size就是某一段時間內, process所需要的frames
  - $D = \sum WSS_i$  (total demand frames)
  - If  $D > m$  (available frames)  $\rightarrow$  thrashing Available frames不足以應付所有Processes最低限度所需
  - The OS monitors the  $WSS_i$  of each process and allocates to the process enough frames
    - if  $D \ll m$ , 代表available frames還很多, increase degree of MP
    - if  $D > m$ , 代表memory不足, suspend some processes
- Good parts
  - Prevent thrashing while keeping the degree of multiprogramming as high as possible
  - Optimize CPU utilization
- Bad part
  - Tracking WS is too expensive

# Example



$WSS_a = 10$

$WS_{(a)} = \{18, 19, 20, 21, 22, 23, 24, 29, 30, 33\}$

$WS_{(b)} = \{18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33\}$

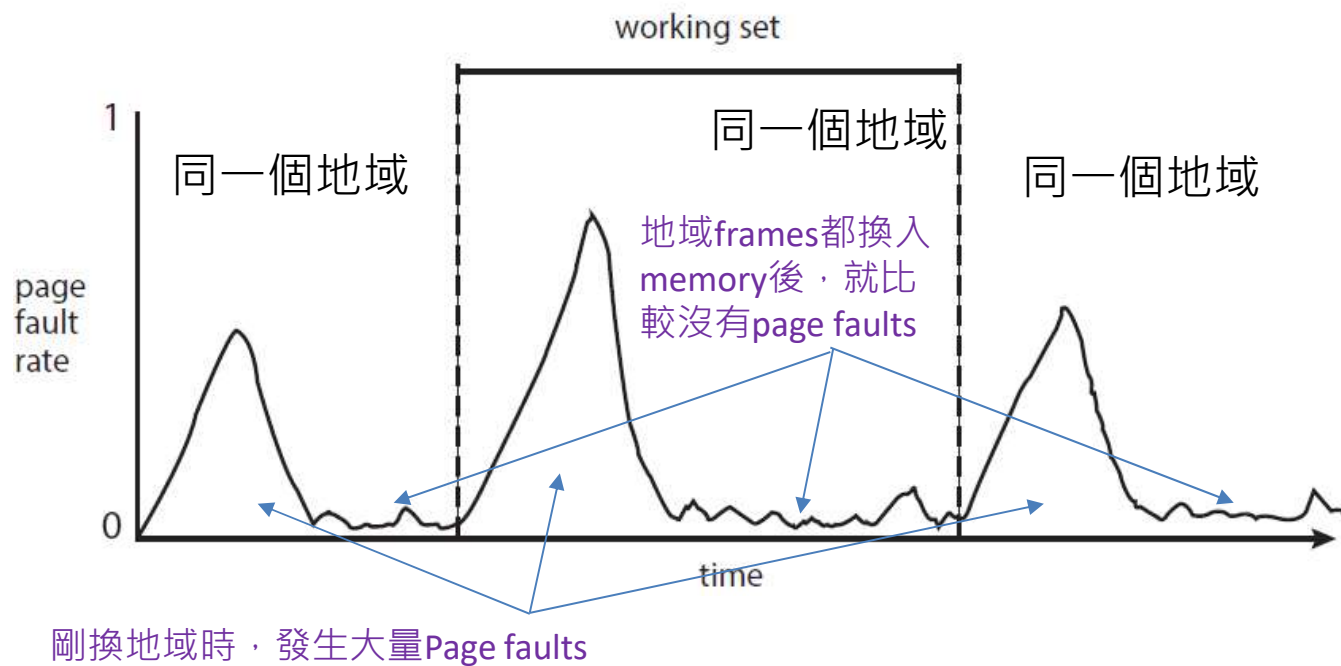
$WSS_b = 12$



# Case: Windows 10

- When a process is created, it is assigned
  - Working set minimum = 50 pages
  - Working set maximum = 345 pages
- When the amount of memory is insufficient
  - Use a global replacement strategy called Automatic working-set trimming
    - If a process has more pages than its working-set minimum, the virtual memory manager removes pages from the working set

# Working Set and Page Fault Rate



# Page Fault Frequency (PFF)

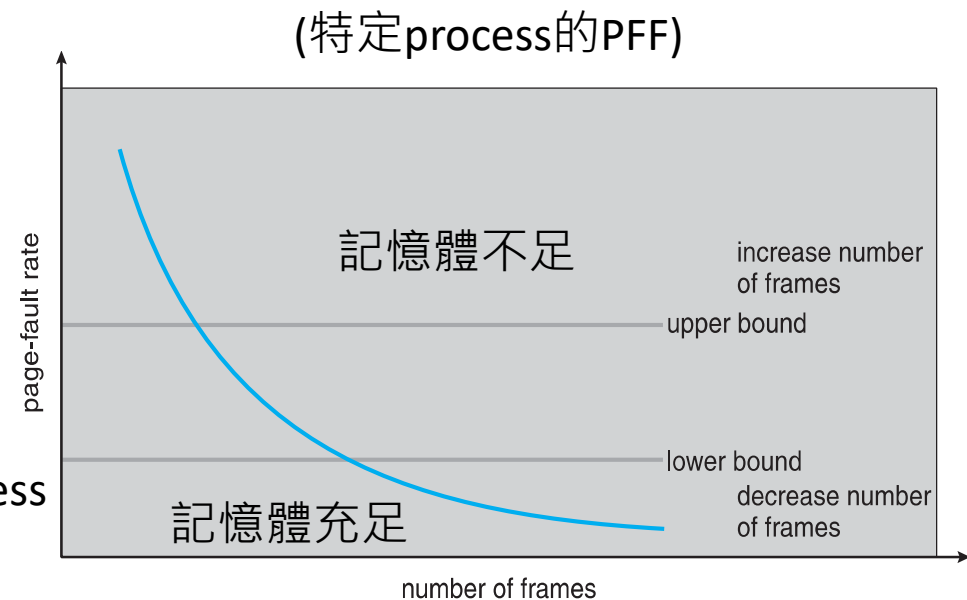
- 追蹤Working Set難度高，PFF是可趨近且較簡單方法
  - 直覺意義: PFF與Thrashing可能性成正比

- 策略

- 直接量測Process的PFF
  - 訂出PFF的上下限，採取因應措施

- 方法

- If PFF exceeds the upper limit
    - allocate another frame to the process
  - If PFF falls below the lower limit
    - remove a frame from the process



# Q & A