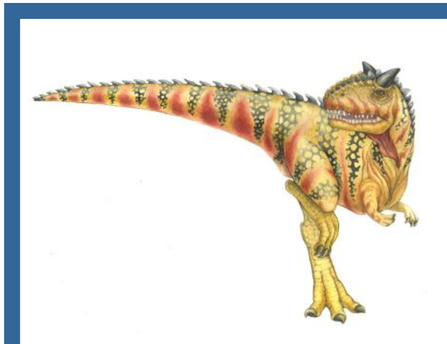
Xi'an Jiaotong-Liverpool University
西交利物浦大學

CPT104 - Operating Systems Concepts

Virtual Memory



Virtual Memory

- Background
- Demand Paging
- Copy-on-Write in Operating System
- Page Replacement
- Frame Allocation. Thrashing

Background

Virtual memory (VM) is a method that manages the exceeded size of larger processes as compared to the available space in the memory.

Virtual memory - separation of user logical memory from physical memory.

- ***Only part of the program needs to be in memory for execution***.
- The components of a process that are present in the memory are known as **resident** set of the process
- ***Need to allow pages/segments to be swapped in and out***.

Swap space / Swap partition

The implementation of a VM system requires both ***hardware*** and ***software*** components.

- The software implementing the VM system is known as *VM handler*.
- The hardware support is the *memory management unit* built into the CPU.

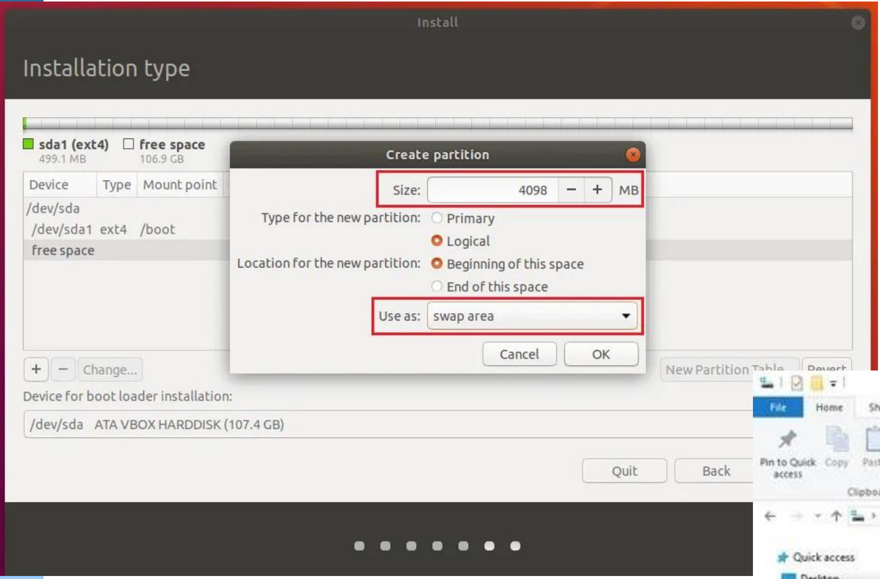
The VM system realizes a huge memory only due to the hard disk.

With the help of the hard disk, the VM system is able to manage larger-size processes or multiple processes in the memory.

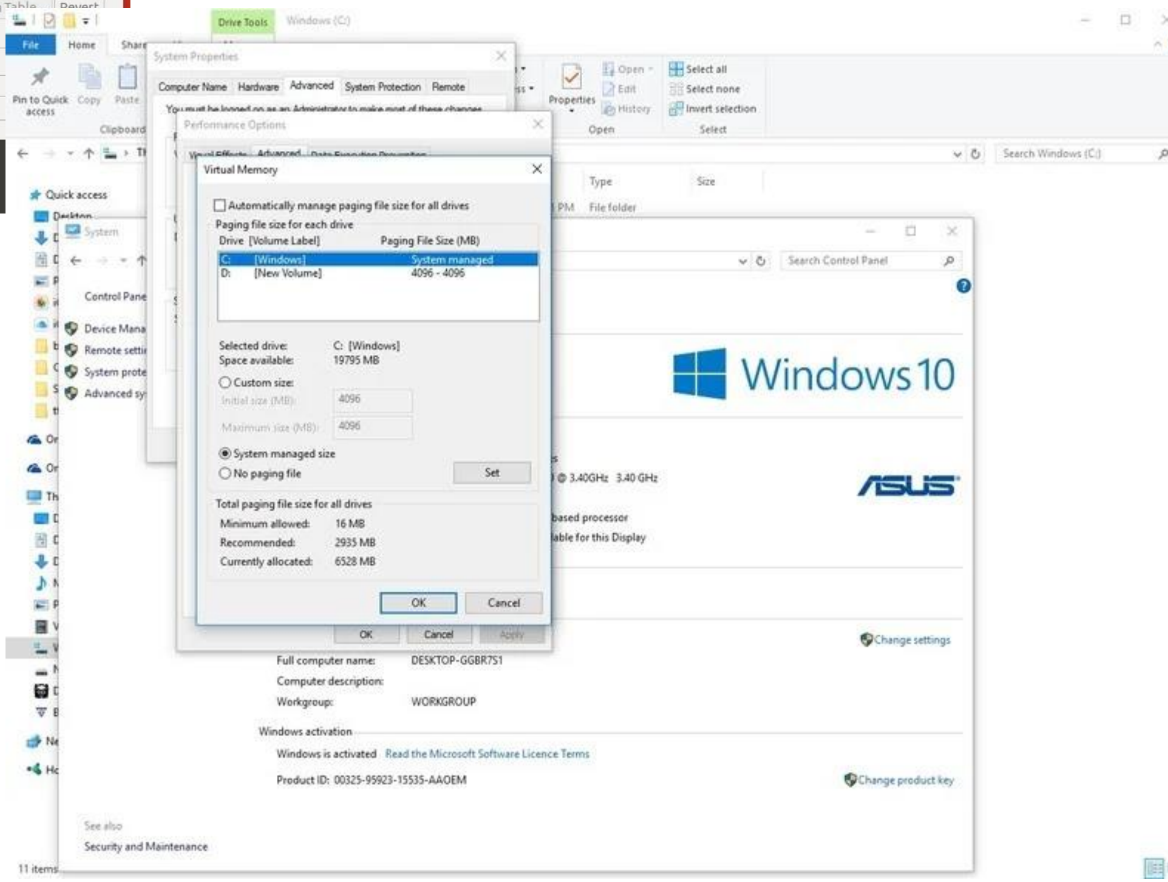
For this purpose, a **separate space** known as **swap space** is reserved in the disk.

Swap space / Swap partition

Create **Swap Partition** – Ubuntu 18.04 LTS



Swap space reserved in the disk.



Windows 10 – **Virtual Memory**

Background

What if there isn't enough physical memory available to load a program?

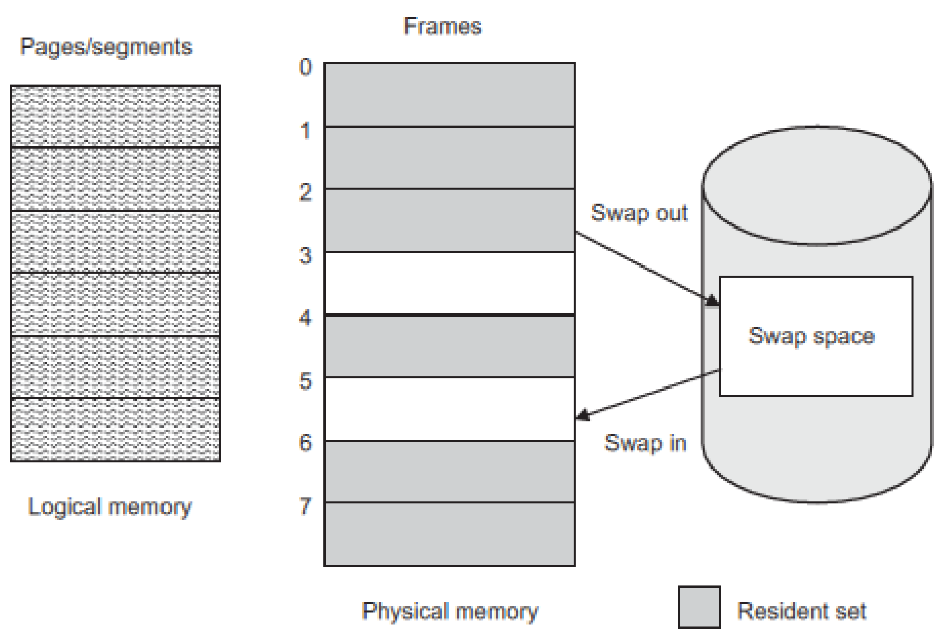
– A large portion of program's code may be unused

– A large portion of program's code may be used infrequently

- **Idea: load a component of a process only if it is needed**

Virtual memory can be implemented via:

- **Demand paging**
- **Demand segmentation**



Demand loading

Demand Paging

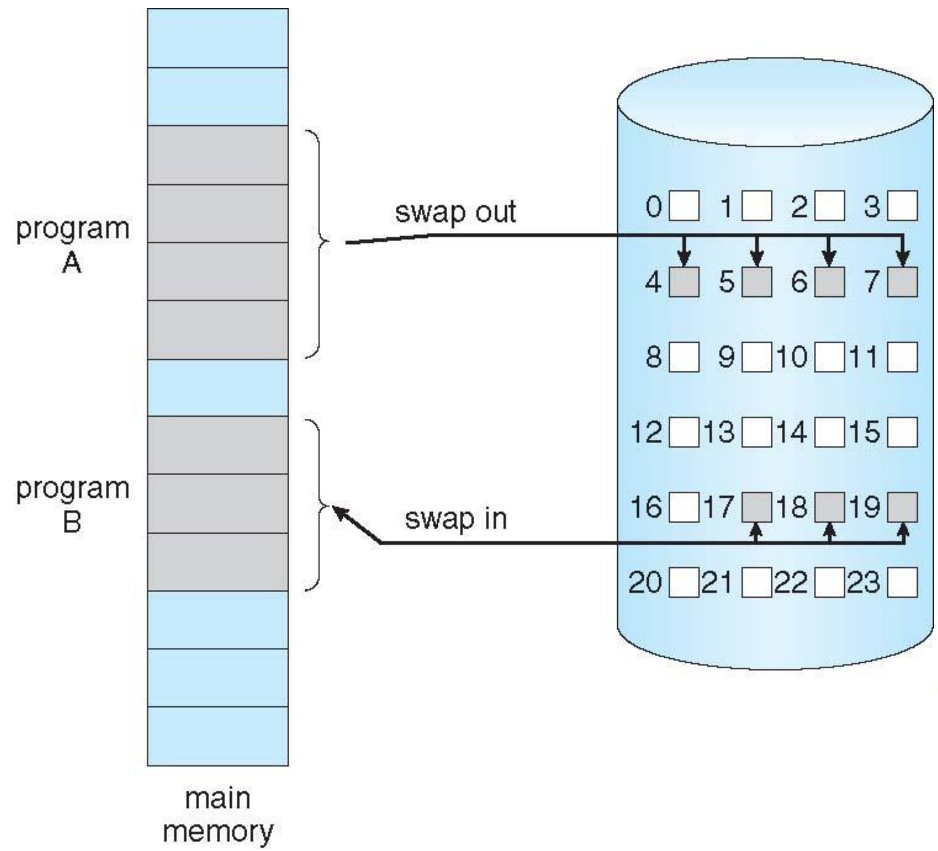
The concept of loading only a part of the process (pages) into memory for processing.

Demand Paging

- when the process begins to run, its pages are brought into memory only as they are needed, and if they're never needed, they're never loaded.

- when a logical address generated by a process points to a page that is not in memory.

- **Lazy swapper** – never swaps a page into memory **unless page will be needed**
  - Swapper that deals with pages is a *pager*



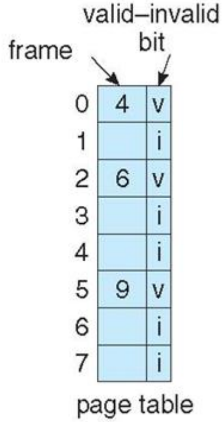
Demand Paging

How to recognize whether a page is present in the memory?

- A **bit** set to "**valid**" → the associated page **is** both **valid (**in the logical address space of the process) and **in memory.**
- A **bit** set to "**invalid**" → the page either is **not valid** (not in the logical address space of the process) or **is valid but is currently on the disk**.

What happens if the process tries to access a page that is not in the memory?

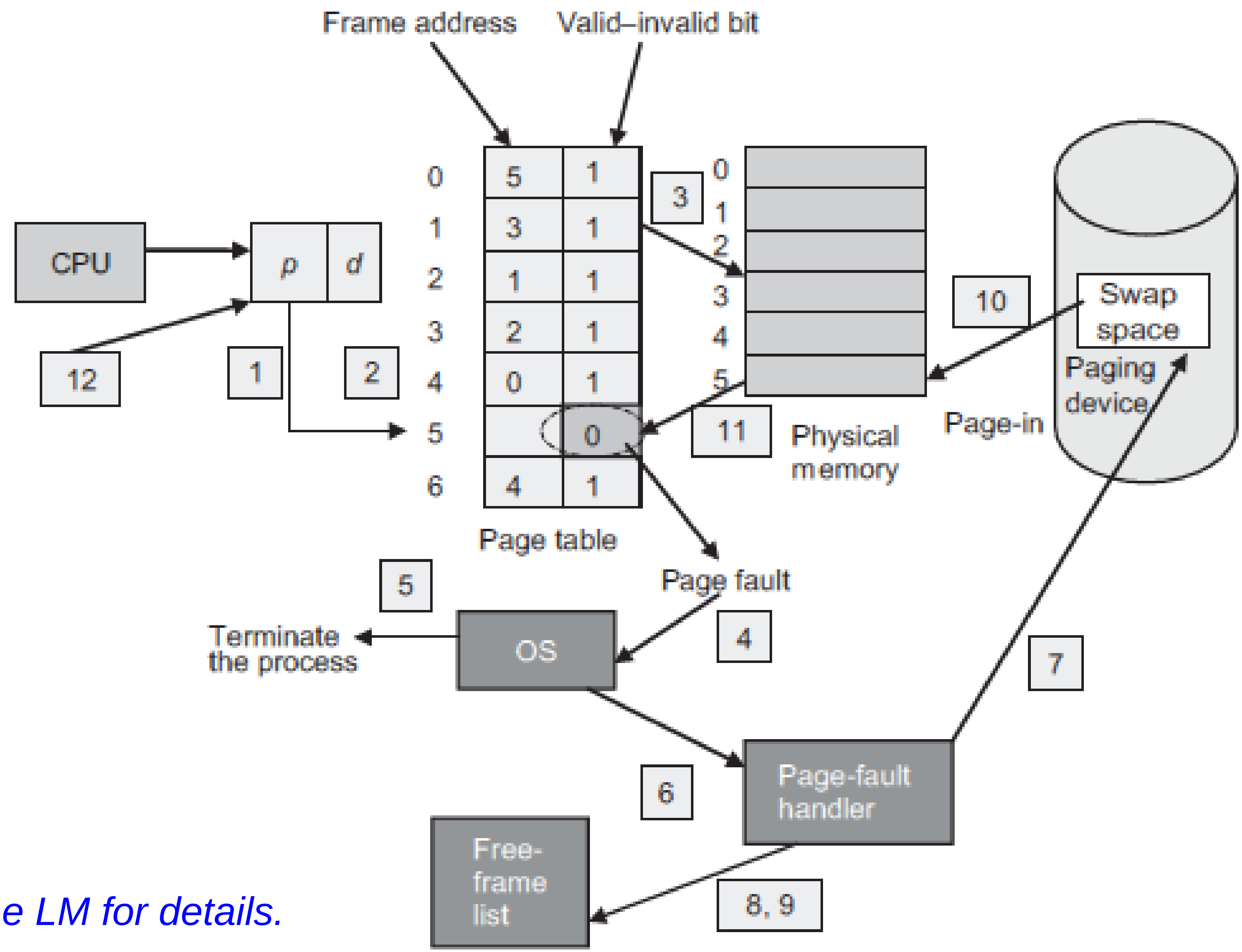
- when the page referenced is not present in the memory → **PAGE FAULT**.
- while translating the address through the page table notices that the page-table entry has an invalid bit. **It causes a trap to the OS** so that a page fault can be noticed.



What happens if there is no free frame in the memory ?

- an existing page in the memory needs to be **paged-out**.
- which page will be replaced? → **page-replacement algorithms**.

Steps in Handling a Page Fault



See the LM for details.

Performance of Demand Paging

NO page fault: the effective access time = the memory access time**.**

Otherwise…

p be **the probability of a PAGE FAULT** $0 \le p \le 1$

- if $p = 0$, no page faults
- if $p = 1$, every reference is a fault

The Effective Access Time (EAT):

$$\textbf{EAT} = (1 - p) \times \textbf{Memory Access Time} + p \times \textbf{Page Fault Time}$$

Major components of the *page-fault (service) time*:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

VM system also uses TLB to reduce the memory accesses and increase the system performance.

Copy-on-Write in Operating System

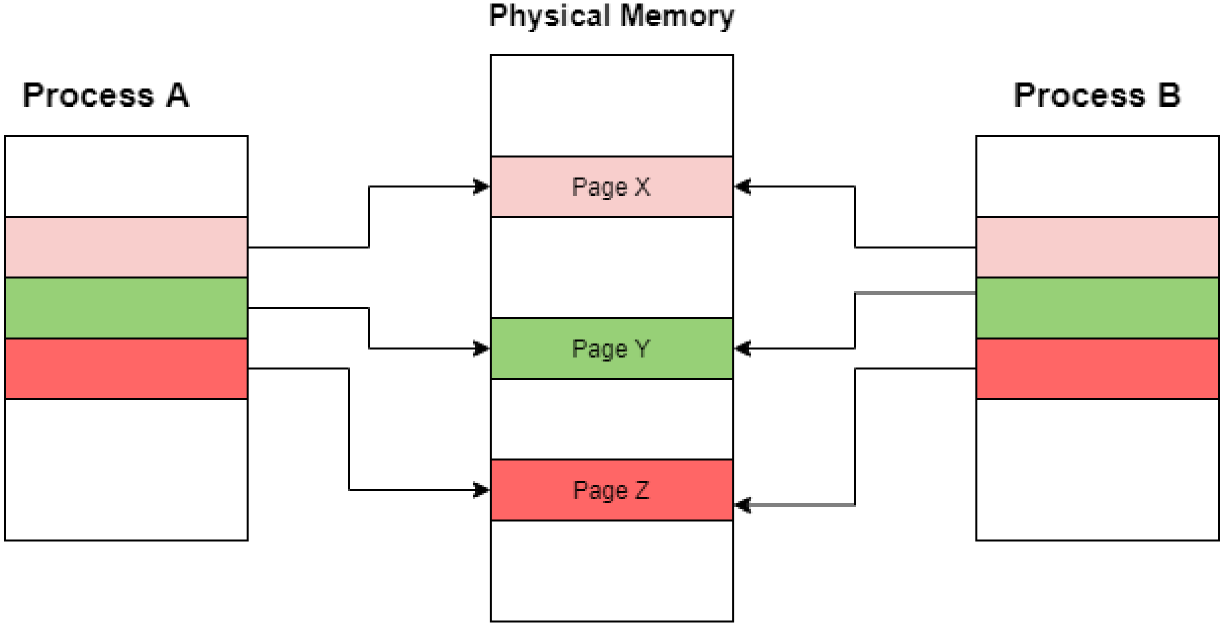
Only pages that are written need to be copied.

Copy-on-Write (COW)

<u>Process creation</u> using the **fork()** system call **may** (*initially*) **bypass the need for demand paging** by using a technique similar to ***page sharing***.

Copy-on-Write = strategy that those pages that are never written need not be copied. Only the pages that are written need be copied.

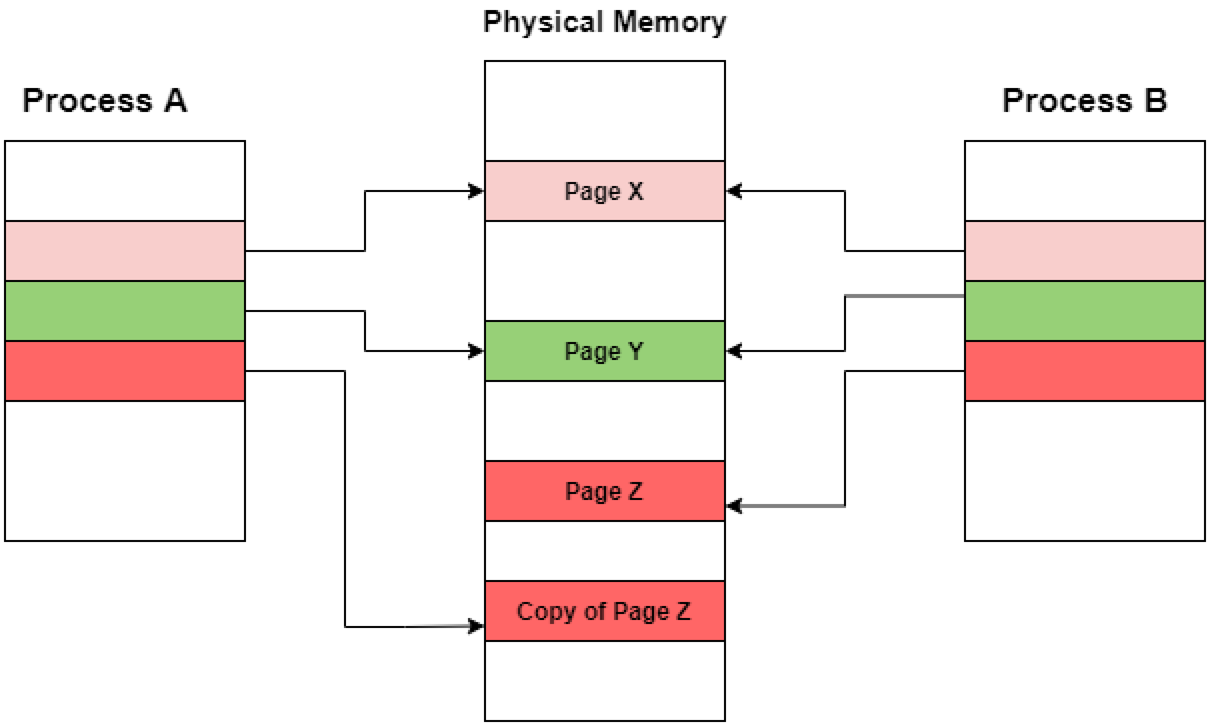
The parent and child process to **share the same pages of the memory** initially.



Process A creates a new process - Process B

Copy-on-Write (COW)

If any process either parent or child modifies the shared page, only then the page is copied.



process A wants to modify a page (Z) in the memory

Page Replacement

When a page fault occurs during the execution of a process, a page needs to be paged into the memory from the disk.

Two major problems to implement demand paging:

What happens if there is no free frame? ⇒ Page replacement

- When a page is to be replaced, which frame shall be the "***victim***"?
- How to select a replacement algorithm? it wants one with the lowest page-fault rate.

How many frames shall be allocated to each process? ⇒ Frame allocation

- When page replacement is required, it must select the frames that are to be replaced.

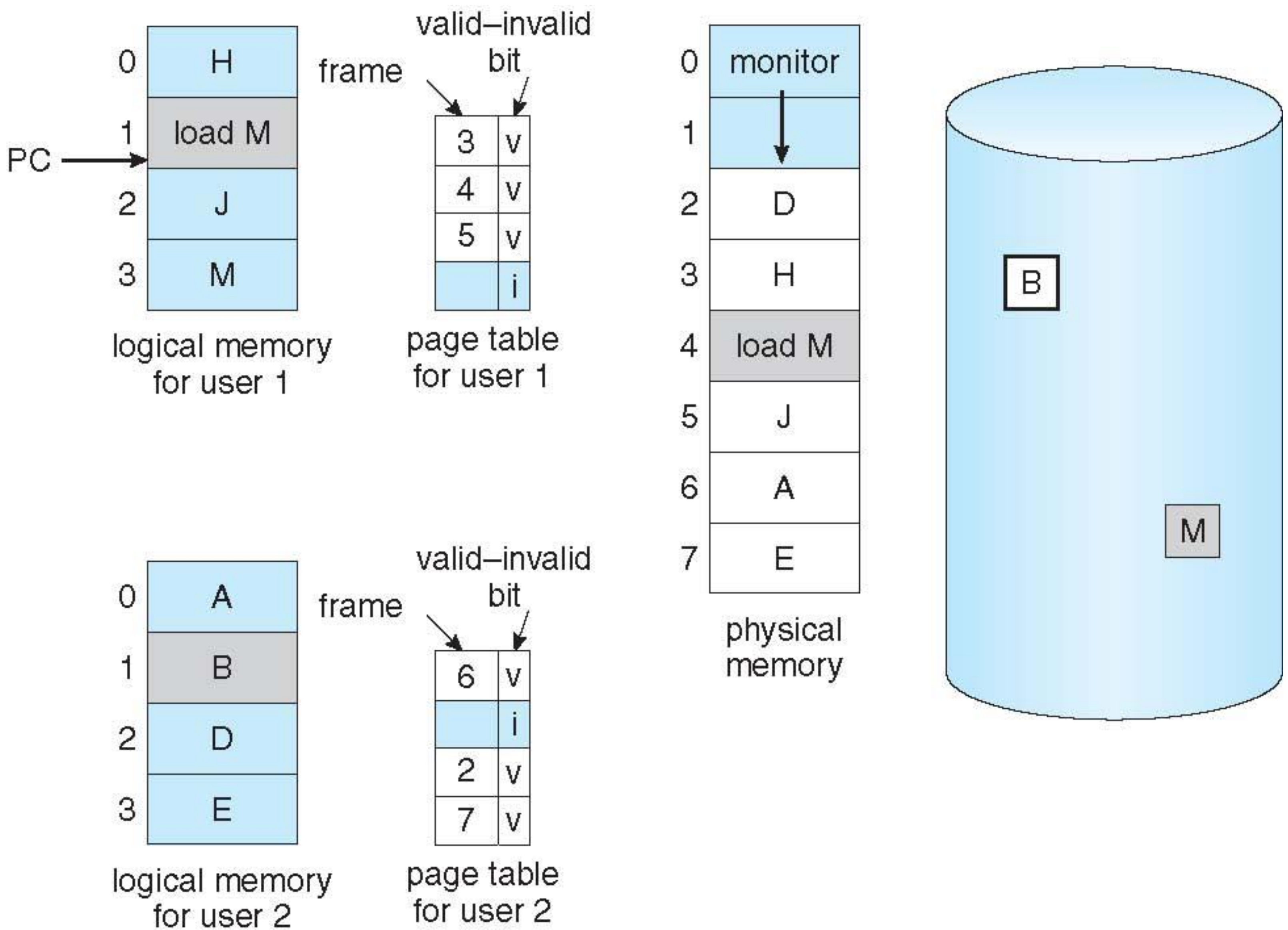
Page Replacement

The degree of multiprogramming increases $\Rightarrow$ over-allocating memory $\Rightarrow$ NO free frames on the free-frame list, all memory is in use.

A good replacement algorithm achieves:

- a *low page fault rate*
  - ensure that heavily used pages stay in memory
  - the replaced page should not be needed for some time
- a *low latency of a page fault*
  - efficient code
  - replace pages that do not need to be written out
  - a special bit called the ***modify*** *(dirty)* ***bit*** can be associated with each page.

Need For Page Replacement



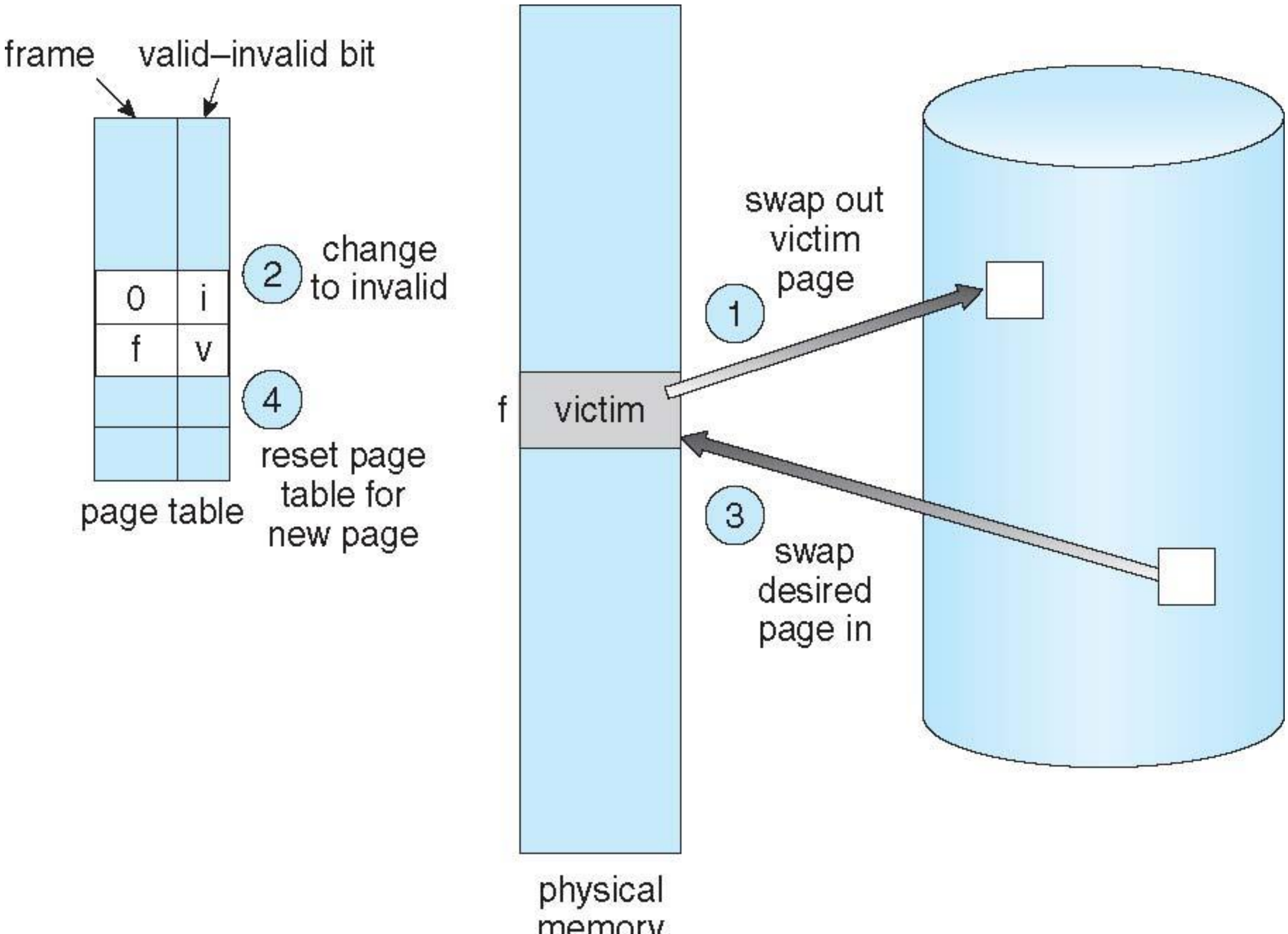
Basic Page Replacement

1. **Find** the **location of** the desired **page on disk.**
2. **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**
   - Check the ***modify (dirty) bit*** *with each page or frame.*

     If the bit is set → the page has been modified.

     If the bit is not set → the page has not been modified. It need not be paged-out for replacement and can be overwritten by another page because its copy is already on the disk. This mechanism reduces the page-fault service time.
3. **Bring** the desired **page into** the (newly) **free frame** and **update** the **page and frame tables**.
4. **Continue** the process by restarting the instruction that caused the trap.

Page Replacement



Page-replacement algorithms

- First-In First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- Second-Chance (Clock) Algorithm
- Counting Algorithms

Algorithm evaluation:

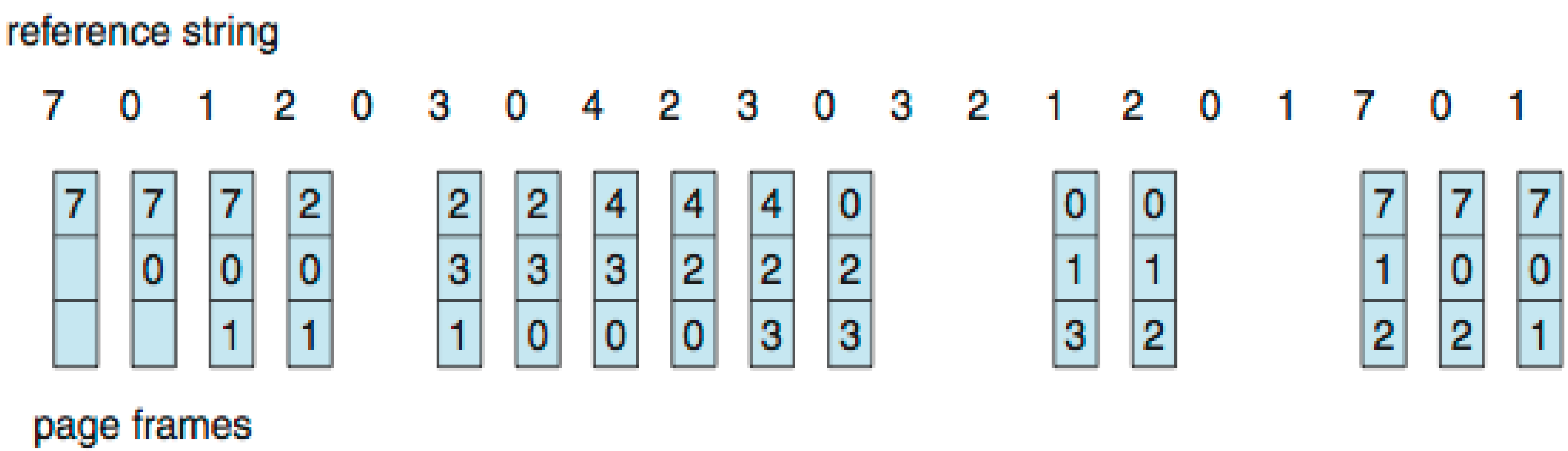
- run a **string of memory references** and **compute** the number of **page faults**.

- **recording** a trace of the **pages accessed by a process**.

Reference string - is the sequence of pages being referenced**.**

First-In First-Out (FIFO) Algorithm

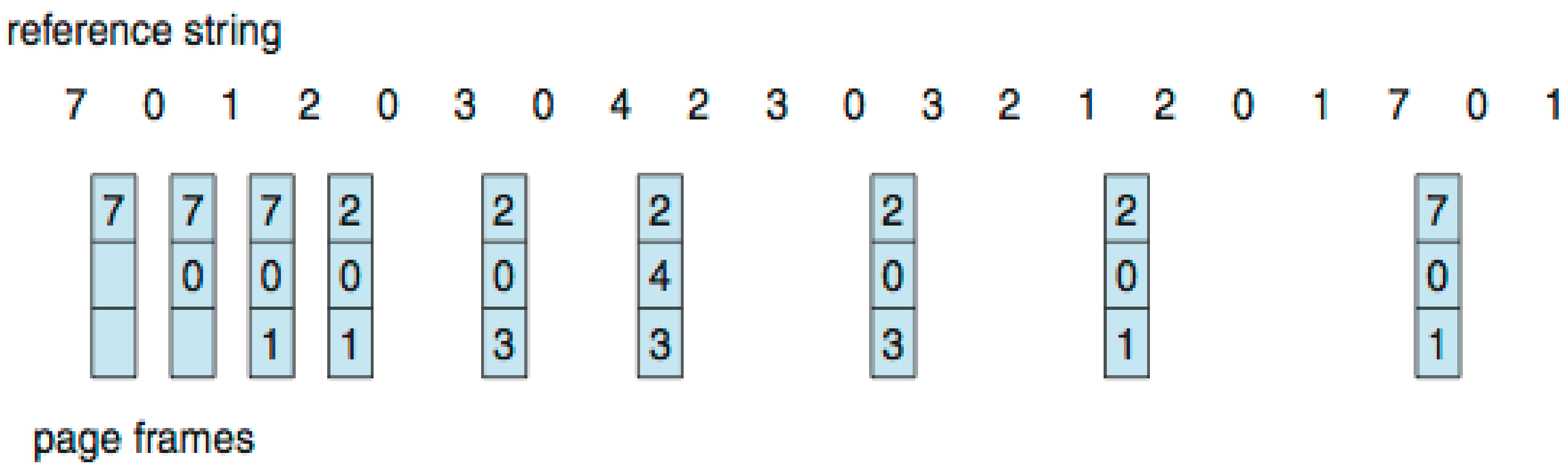
- When a page must be replaced**, the oldest page is chosen**.
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- **3 frames** (3 pages can be in memory at a time per process)



- **15 page-fault**

Optimal Algorithm

- Replace page that **will not be used for longest period of time**
  - **9 page-fault**



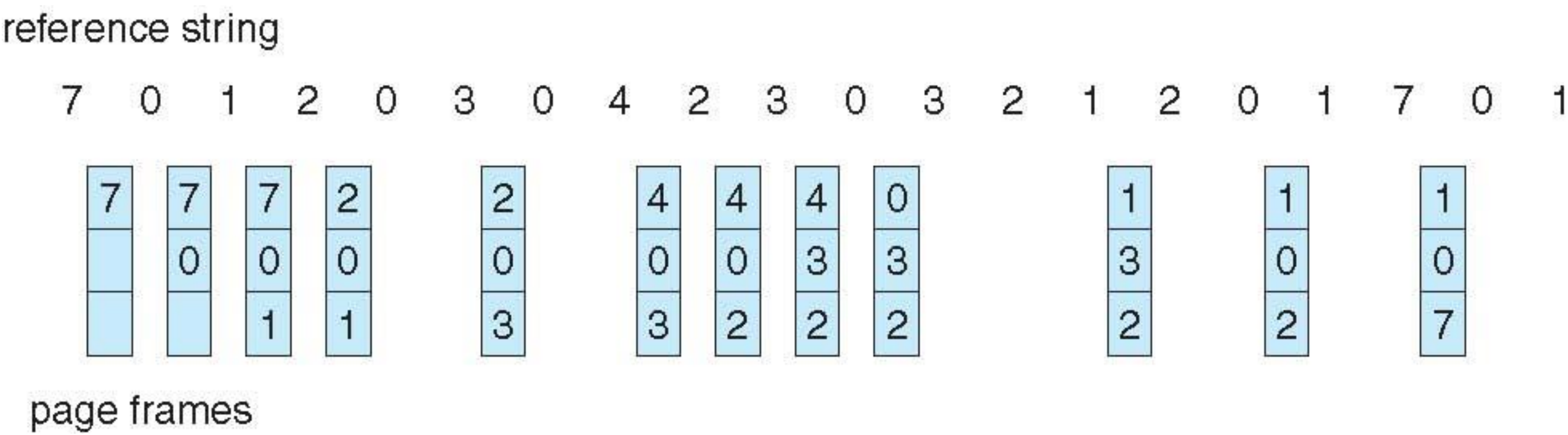
Optimal algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

It cannot be implemented - there is no provision in the OS to know the future memory references.

The idea is to predict future references based on the past data,

Least Recently Used (LRU) Algorithm

- replace **the page that has been unused for the longest time.**



- **12 faults** – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

How to implement LRU replacement?

How to find out a page that has not been used for the longest time?

Two implementations are feasible:

- **Counter** implementation
- **Stack** implementation

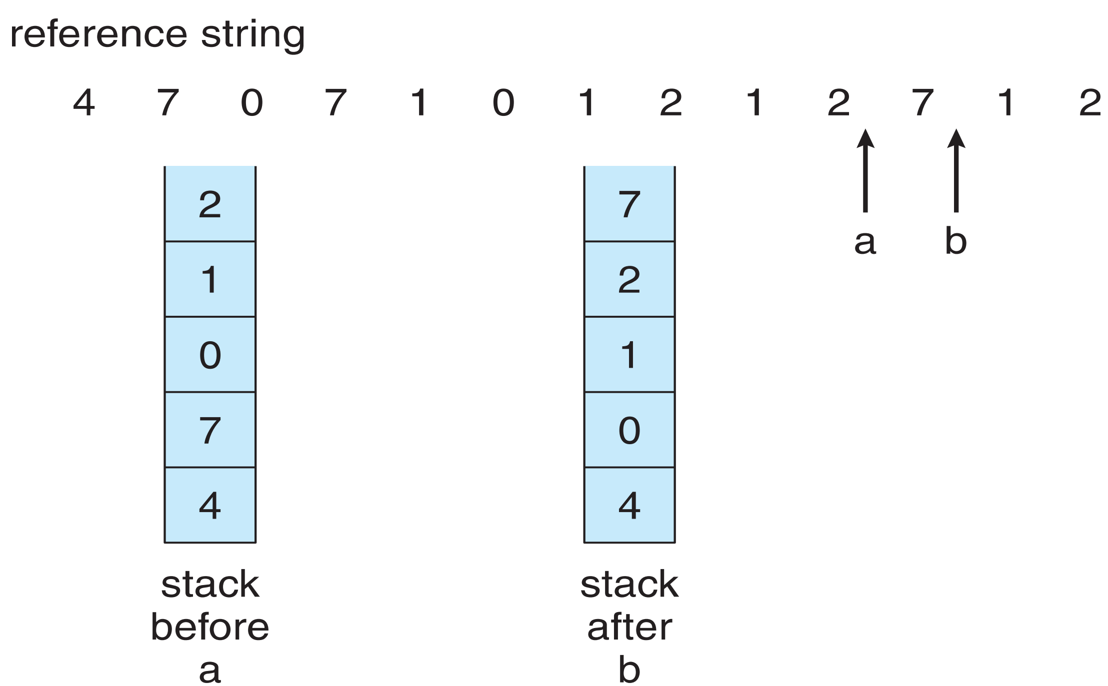
COUNTER implementation

- associate with each page-table entry a **time-of-use field** or a **counter**; every time page is referenced through this entry, copy the clock into the counter
- **When a page needs to be changed, look at the counters to find the smallest value**
- **replace the page with the smallest time value**.

How to implement LRU replacement?

STACK implementation

- **whenever a page is referenced**, it **is removed from the stack and put on the top**.
- the **most recently used page is always at the top of the stack** and the **least recently used page is always at the bottom**



LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference bit -** will say whether **the page has been referred in the last clock cycle or not.**
- **The reference bit** for a page **is set by the hardware** whenever that **page is referenced** (either a read or a write to any byte in the page).
- Reference bits are associated with each entry in the page table.
  - With each page associate a bit, **initially = 0**
  - <u>When **page is referenced, bit set to 1**</u>

Second-Chance (Clock) Page-Replacement Algorithm

- keeps a circular list of pages, with the "**hand**" (iterator) pointing to the last examined page in the list.

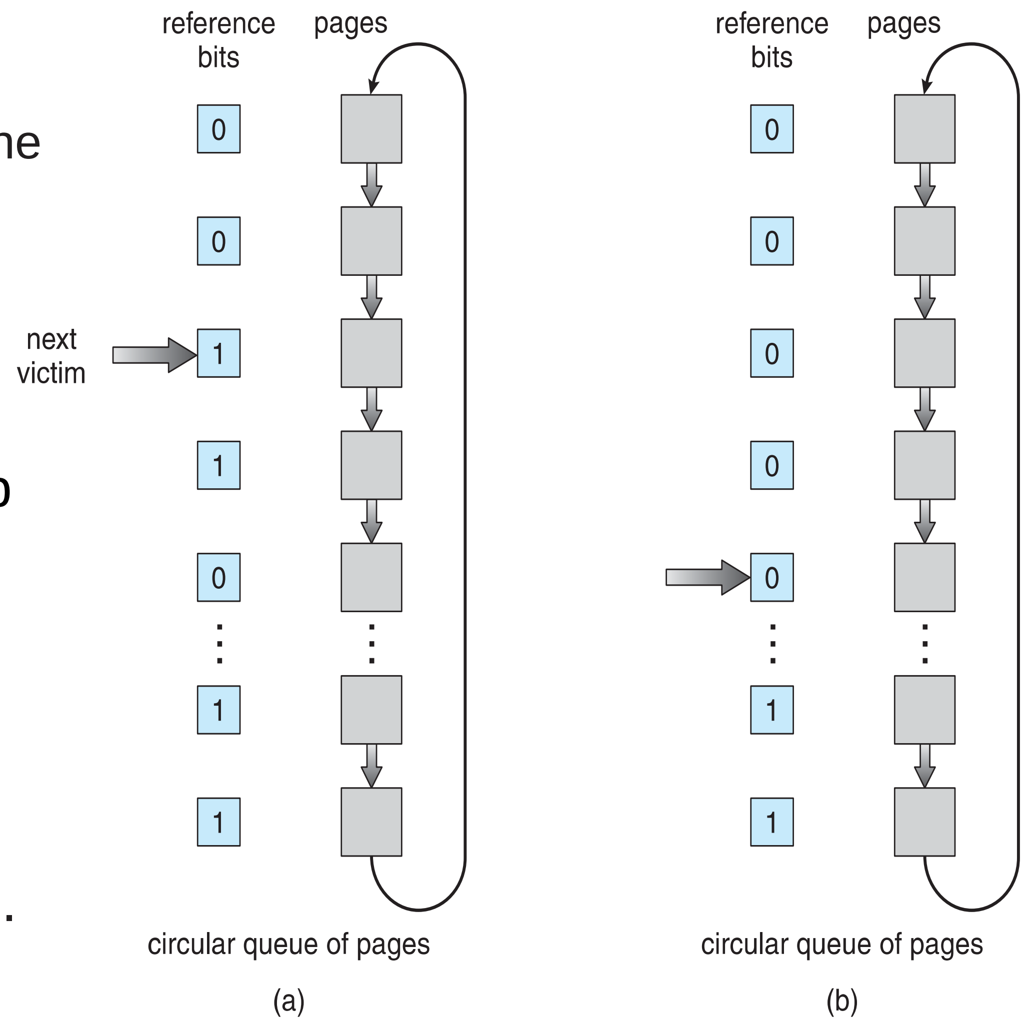
RB = Reference bit or **Use bit**
give information regarding whether the page has been used

Iterator scan:

- IF page's RB = 1, set to 0 & skip
- ELSE if RB = 0, remove

The idea behind:

A page that is being frequently used will not be replaced (RB=1).



Counting-Based Page Replacement

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

- **Least Frequently Used** (**LFU**) **Algorithm** replaces page with smallest count.

- **Most Frequently Used** (**MFU**) **Algorithm** is based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Frame Allocation. Trashing

Frame Allocation Algorithms

The two algorithms commonly used to allocate frames to a process:

- **Equal allocation** - In a system with *x* frames and *y* processes, each process gets equal number of frames
  - *Example*, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
- **Proportional allocation** – Frames are allocated to each process according to the process size.
  - Dynamic as degree of multiprogramming, process sizes change

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

Frame Allocation Algorithms

Proportional allocation - *example*

A system with 62 frames

- P1 = 10KB
- P2 = 127KB

Then

- P1 will get (10 / 137) * 62 = 4 frames
- P2 will get (127 / 137) * 62 = 57 frames.

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 added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out
  - a process spends more time paging then executing

End of Lecture

- **Summary**
  - Background
  - Demand Paging
  - Copy-on-Write in Operating System
  - Page Replacement
  - Frame Allocation. Thrashing

- **Reading**
  - Textbook 9th edition, **chapter 9 of the module textbook**