# CSE308 Operating Systems

## Page Replacement Algorithms

**S.Rajarajan**
**SASTRA**

- If a **process of ten pages** actually uses **only half** of them, then **demand paging save**s the **I/O** necessary to **load the five pages** that are **never used.**
- We could also **increase our degree of multiprogramming** by running twice as many processes.
- Thus, if we had **forty frames**, we could **run eight processes**, rather than the **four.**
- If we **run six processes**, each of which is **ten pages in size** but actually **uses only five pages**, we have **higher CPU utilization** and **throughput.**
- we are **over-allocating** memory.

- Consider that **system memory** is not used only for **holding program pages.**
- **Buffers for I/O** also consume a considerable amount of memory.
- Deciding how much memory to allocate to I/O and how much to program pages is a significant challenge.
- Some systems **allocate a fixed percentage of memory for I/O buffers**, whereas others allow both user **processes and the I/O subsystem to compete**

# Need of Page replacement

- While a user process is executing, a page fault occurs.
- The operating system determines the desired page is residing on the disk but finds that there are ***no free frames*** on the free-frame list.
- The operating system has several options at this point.
  - It could **terminate the user process**. Users should not be aware that their processes are running on a paged system—paging should be logically transparent to the user. So this option is **not the best choice**

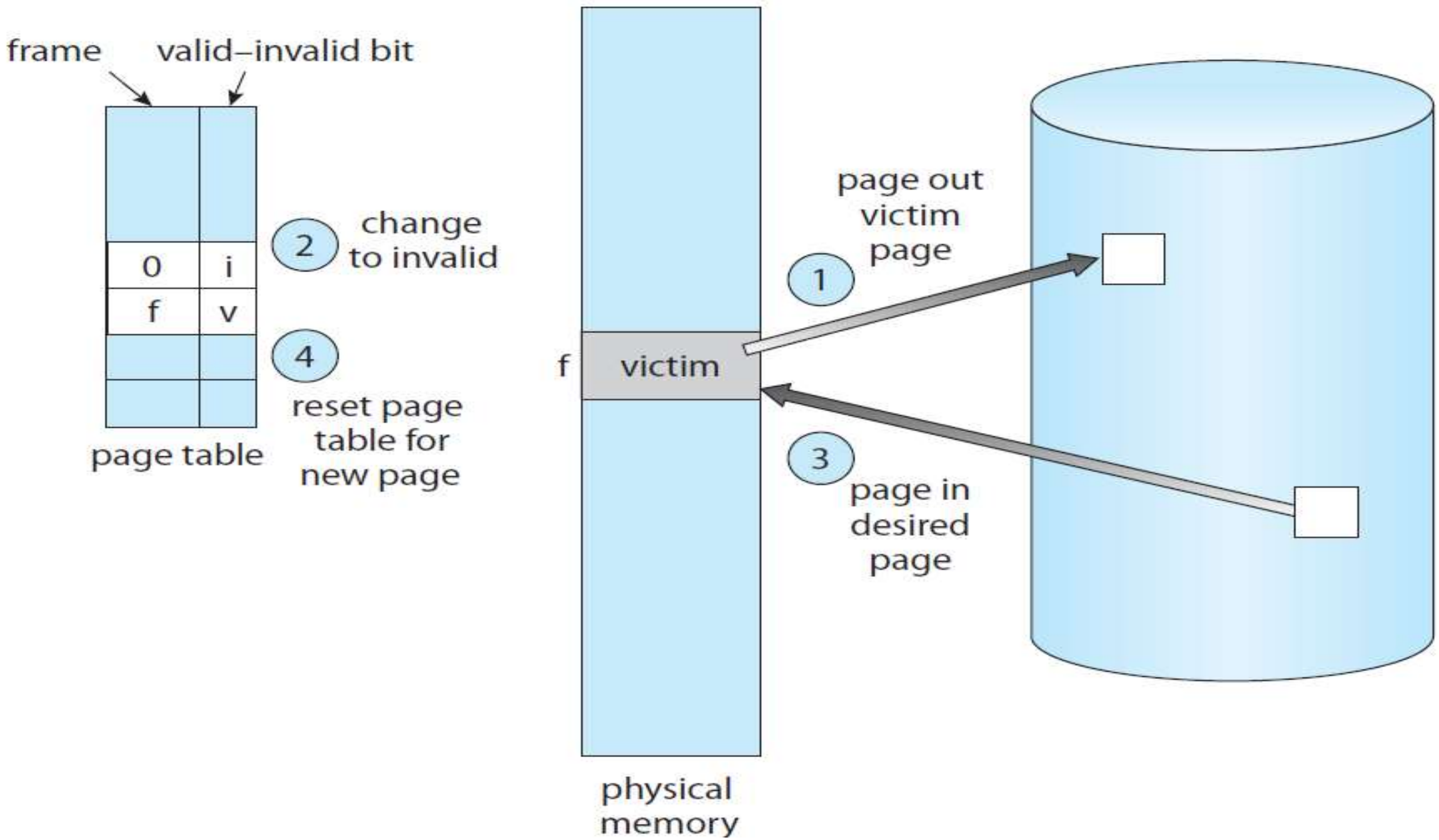
- Operating system could instead **swap out a process**, freeing all its frames and reducing the level of multiprogramming. This option is a good one in certain circumstances

# Basic Page Replacement

- Page replacement takes the following approach.
- If no frame is free, we find one that is not currently being used and free it.
- We can free a frame by **writing its contents to swap space(disk)** and changing the **page table** (and all other tables) to indicate that the **page is no longer in memory.**
- We can now **use the freed frame** to hold the page for which the process faulted.

- We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
   a. If there is a free frame, use it.
   b. If there is no free frame, use a page-replacement algorithm to select a **victim frame**.
   c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred.


frame
valid–invalid bit
0
i
f
v
2
change to invalid
4
reset page table for new page
page table
f
victim
physical memory
page out victim page
1
3
page in desired page

# Role of Modify bit

- Notice that, if no frames are free, ***two page transfers (one out and one in)*** are required.
- This situation effectively **doubles the page-fault service time** and increases the effective access time accordingly.
- We can reduce this overhead by using a **modify bit (or dirty bit).**
- When this scheme is used, each page or frame has a modify bit associated with it in the hardware.
- The **modify bit** for a page is **set by the hardware whenever any byte** in the page is **written into,** indicating that the **page has been modified**

- When we select a page for replacement, we **examine its modify bit**.
- If the **bit is set**, we know that the **page has been modified** since it was read in from the disk.
- In this case, we **must write the page to the di**sk.
- If the **modify bit is not set**, however, the page has **not been modified** since it was read into memory.
- In this case, we **need not write** the memory page **to the disk**: it is already there.
- This technique also applies to **read-only pages.** Such pages cannot be modified; thus, they may be  discarded when desired.

- We must solve two major problems to implement demand paging :
  - we must develop a **frame-allocation algorithm**
  - and a **page-replacement algorithm.**
- That is, if we have multiple processes in memory, we must decide **how many frames to allocate to each process**;
- And when page replacement is required, we must **select the frames** that are **to be replaced**.
- Designing appropriate algorithms to solve these problems is an important task, because disk I/O is so expensive

# Page Reference String

- We evaluate an algorithm by running it on a particular string of memory references and computing the **number of page faults.**
- **<u>Page references</u>:** 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
- To determine the number of page faults for a particular reference string and page-replacement algorithm, we also need to know the **number of page frames available**.
- Obviously, as the **number of frames available increases**, the number of **page faults decreases**

- We next illustrate several page-replacement algorithms. In doing so, we use the reference string
- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.

# FIFO

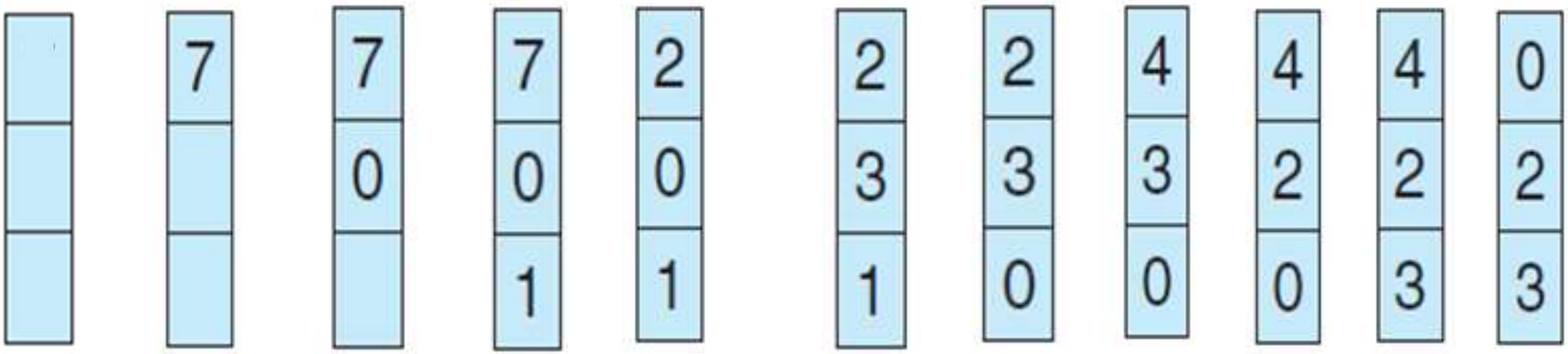
- When a page must be replaced, the **oldest page** is chosen.
- Notice that it is not strictly necessary to record the time when a page is brought in.
- We can create **a FIFO queue** to hold all pages in memory.
- We replace the page at the head of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.

- The FIFO page-replacement algorithm is **easy to understand and program.**
- However, its **performance is not always good**.
- The **oldest page** could contain could contain a **heavily used variable** that was initialized early and is in constant use.
- **Belady's anomaly:** 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.
  - Number of faults **for four frames (10)** is *greater than the number of faults* **for three frames (9).** *This most* unexpected result is known as Belady's anomaly.

# FIFO

10
Page Faults

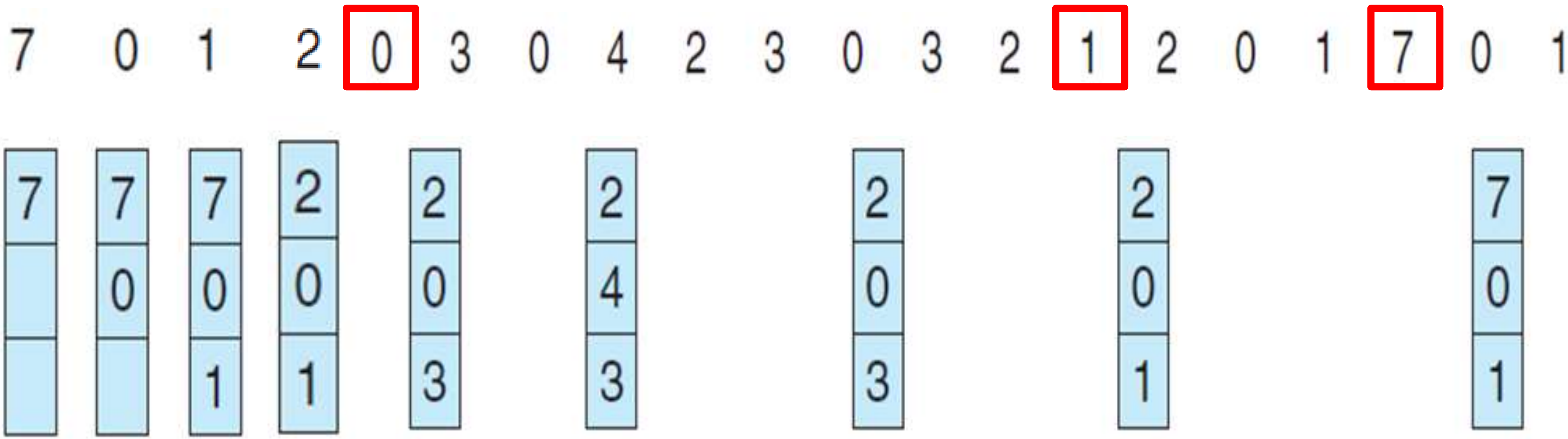
7 0 1 2 0 3 0 4 2 3 0 3



# Optimal Page Replacement

- Algorithm that has the lowest page-fault rate of all algorithms and **will never** suffer from **Belady’s anomaly**.
- Replace the page **that will not be used** for the longest period of time.
- Use of this page-replacement algorithm guarantees the **lowest possible page fault** rate for a fixed number of frames.
- Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string

# Optimal



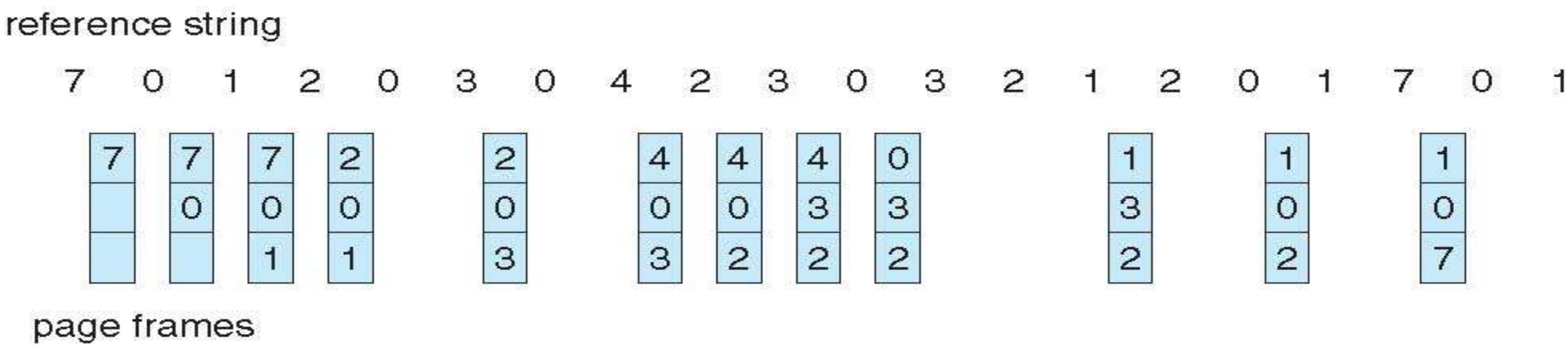

**No of page faults : 9**

# LRU Page Replacement

- If the optimal algorithm is not feasible, perhaps an approximation of the optimal algorithm is possible.
- If we use the recent past as an approximation of the near future, then we can replace the page that has **not been used for the longest period of time.**
- LRU replacement associates with each page the **time of that page's last use.**
- When a page must be replaced, **LRU chooses the page that has not been used for the longest period** of time.
- We can think of this strategy as the **optimal page-replacement** algorithm **looking backward in time**, **rather than forward**.

# Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page




- **12 faults** – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

- The major problem is ***how to implement LRU*** replacement.
- An LRU page-replacement algorithm may require substantial hardware assistance.
- The problem is to determine an order for the frames defined by the time of last use.
- Two implementations are feasible:
- **Counters.** In the simplest case, we associate with each page-table entry **a time-of-use field** and add to the **CPU a logical clock**.
- The **clock is incremented** for every memory reference. Whenever a reference to a page is made, the **contents of the clock register** are **copied to the time-of-use field** in the page-table entry for that page.
- We **replace** the **page with** the **smallest time value**.

- Overflow of the clock must be considered.
- **Stack.** Another approach to implementing LRU replacement is to **keep a stack** of page numbers.
- Whenever **a page is referenced**, it is **removed from the stack** and **put on the top**.
- In this way, the **most recently used page** is **always at the top of the stack** and the **least recently used page** is always **at the bottom.**

- Because entries **must be removed** from the **middle of the stack**, it is best to implement this approach by **using a doubly linked list** with a **head pointer** and a **tail pointer.**
- Like optimal replacement, LRU replacement does not suffer from Belady’s anomaly.
- Both belong to a class of page-replacement algorithms, called **stack algorithms.**
- A **stack algorithm** is an algorithm for which it can be shown that the set of pages in memory for $n$ frames is always a ***subset of the set of pages that would be in memory with n*** + 1 frames.

# LRU-Approximation Page Replacement

- Few computer systems provide **sufficient hardware support** for **true LRU** page replacement.
- In fact, some systems provide **no hardware support**, and other page-replacement algorithms (such as a **FIFO algorithm**) must be used.
- Many systems provide some help, however, in the form of a **reference bit.**
- 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**.
- Initially, **all bits are cleared (to 0)** by the operating system.
- As a user **process executes**, the bit associated with each page referenced is **set (to 1)** by the hardware.
- After some time, we can determine which pages have been used and which have not been  know the ***order of use.***
- This information is the **basis for many page-replacement algorithms that approximate LRU** replacement.

# Additional-Reference-Bits Algorithm

- We can gain additional ordering information by recording the reference bits at regular intervals.
- We can keep an **8-bit byte** for each page in a table in memory.
- At **regular intervals** (say, **every 100 milliseconds**), a **timer interrupt** transfers control to the operating system.
- The operating system **shifts the reference bit** for each page into the **high-order bit of its 8-bit byte**, **shifting the other bits right by 1 bit** and **discarding the low-order bit**

- These 8-bit shift registers contain the history of page use for the last eight time periods
- Assume that Page 5 is referenced every 100 milliseconds

**Ref Bit** | 0

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **Page 5** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- If the page is never referenced for 8 consecutive intervals then its reference byte will be:
- 00000000

- A page with a history register value of **11000100** has been **used more recently** than one with a value of **01110111**.
- If we interpret these 8-bit bytes as unsigned integers, the **page with the lowest number** is the LRU page, and it can be **replaced**.
- Notice that the numbers are **not guaranteed to be unique,** however.
- We can **either replace (swap out) all pages** with the smallest value or use **the FIFO method** to choose among them.
- The **number of bits** of history included in the shift register **can be varied**.

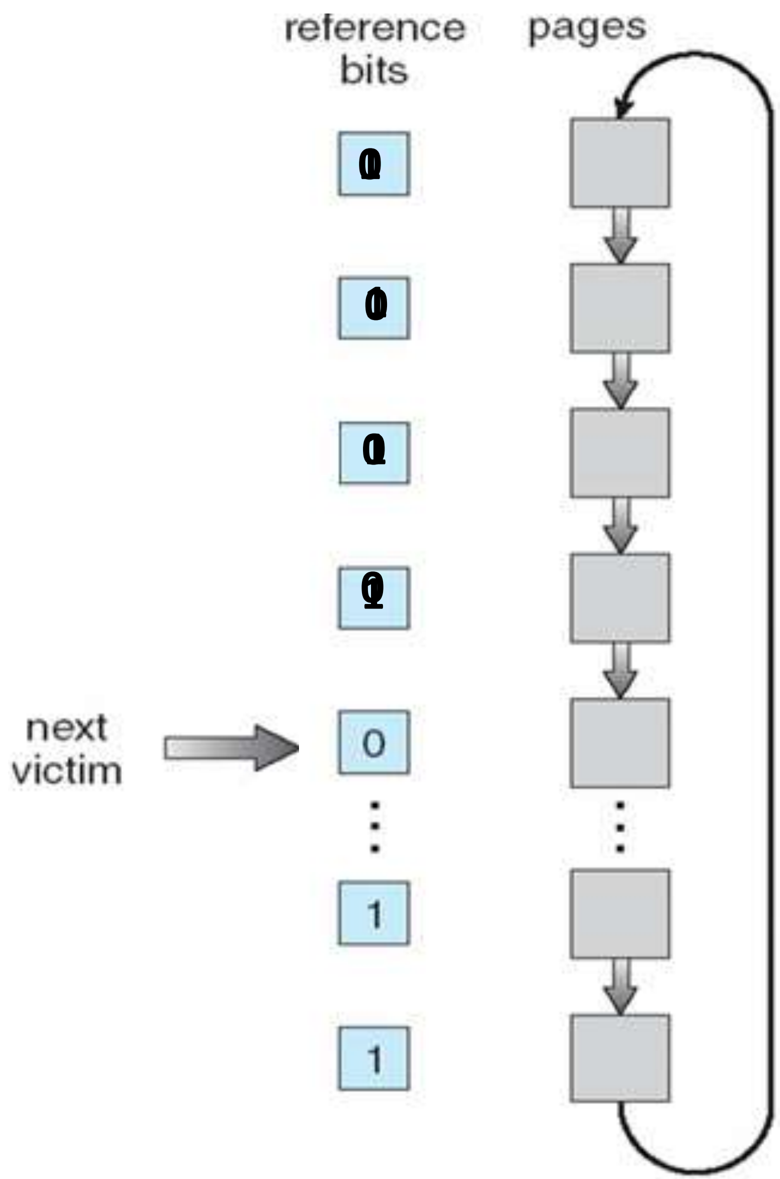
# Second-Chance Algorithm

- The basic algorithm of second-chance replacement is a **FIFO** replacement algorithm
- When a page has been selected, however, we inspect its reference bit.
- If the **value is 0**, we proceed to **replace this page.**
- But if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page.
- When a page **gets a second chance**, its **reference bit is cleared**, and its **arrival time is reset** to the **current time.**

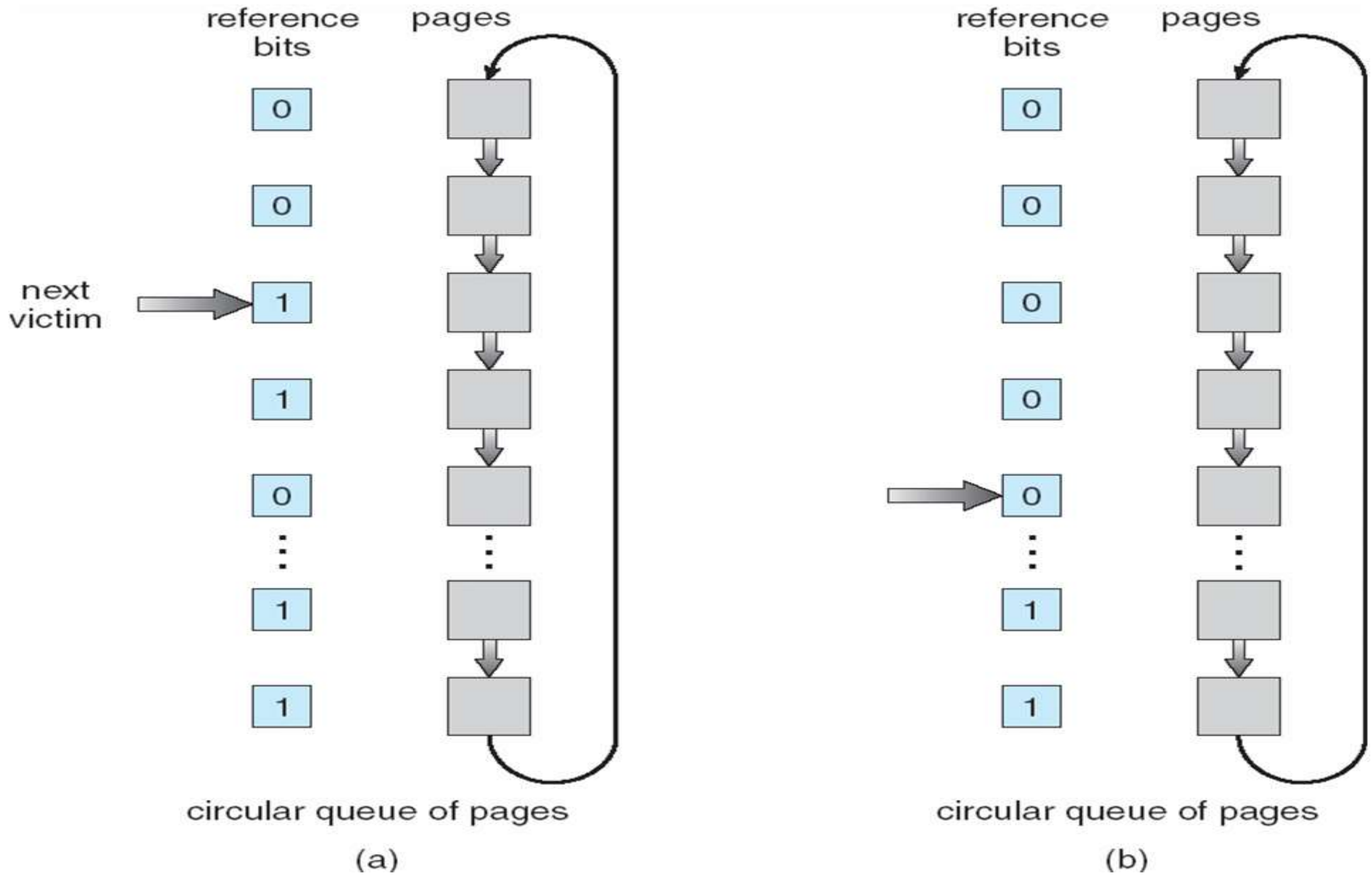
- Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances).
- In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.
- One way to implement the second-chance algorithm (sometimes referred to as the **clock algorithm) is as a circular queue**

- A **pointer** (that is, a **hand on the clock**) indicates **which page is to be replaced next**.
- When a frame is needed, the **pointer advances until it finds a page with a 0** reference bit.
- As it advances, it clears the reference bits.
- Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.
- Notice that, **in the worst case**, when all bits are set, the pointer cycles through the whole queue, **giving each page a second** chance.
- It **clears all the reference bits** before selecting the **first page** for replacement.
- Then Second-chance replacement **degenerates to FIFO** replacement if all bits are set.

# Second Chance Algorithm



Second-Chance (clock) Page-Replacement Algorithm



# Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering the **reference bit and the modify bit.**
- With these two bits, we have the following four possible classes:
- **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
- Each page is in one of these four classes.
- When page replacement is called for, we examine the class to which that page belongs. we replace the **first page encountered** in the **lowest nonempty** class.

# Counting-Based Page Replacement

- There are many other algorithms that can be used for page replacement.
- For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes:
- **least frequently used (LFU) -** the page with the smallest count be replaced
- **most frequently used (MFU)** – the page with the largest count be replaced

# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

- **Thrashing** $\equiv$ a process is busy swapping pages in and out

**9.8** Consider the following page reference string:

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

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

- LRU replacement
- FIFO replacement
- Optimal replacement