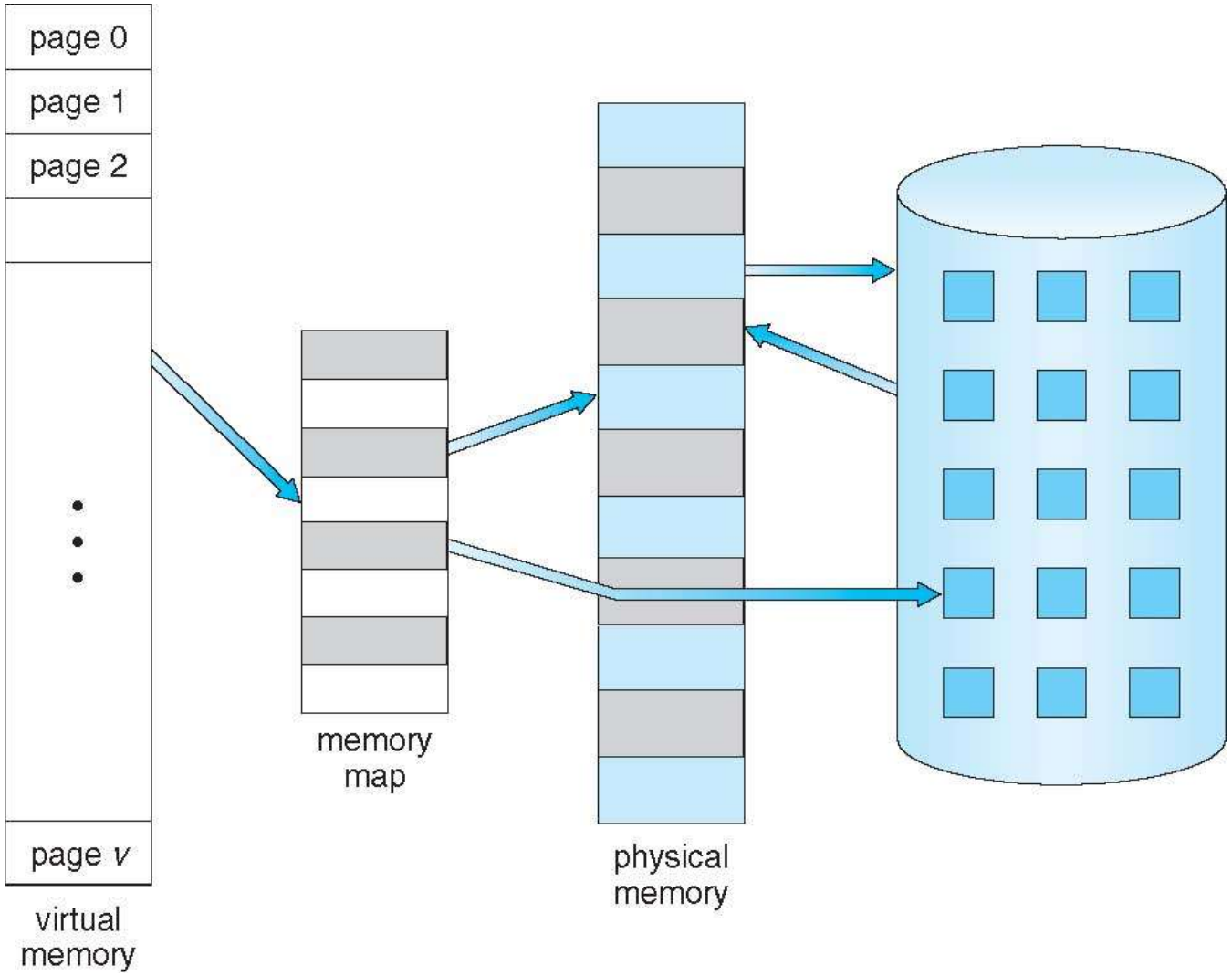
# Chap. 10) Virtual Memory

경희대학교 컴퓨터공학과

조 진 성

# Virtual Memory That is Larger Than Physical Memory



# Demand Paging

A paging system with (page-level) swapping

Bring a page into memory only when it is needed

- ✓ Cf) swapping: entire process is moved

OS uses main memory as a (page) cache of all of the data allocated by processes in the system

- ✓ Initially, pages are allocated from physical memory frames
- ✓ When physical memory fills up, allocating a page requires some other page to be evicted from its physical memory frame

Evicted pages go to disk (only need to write if they are dirty)

- ✓ To a swap file
- ✓ Movement of pages between memory/disks is done by the OS
- ✓ Transparent to the application



# Demand Paging

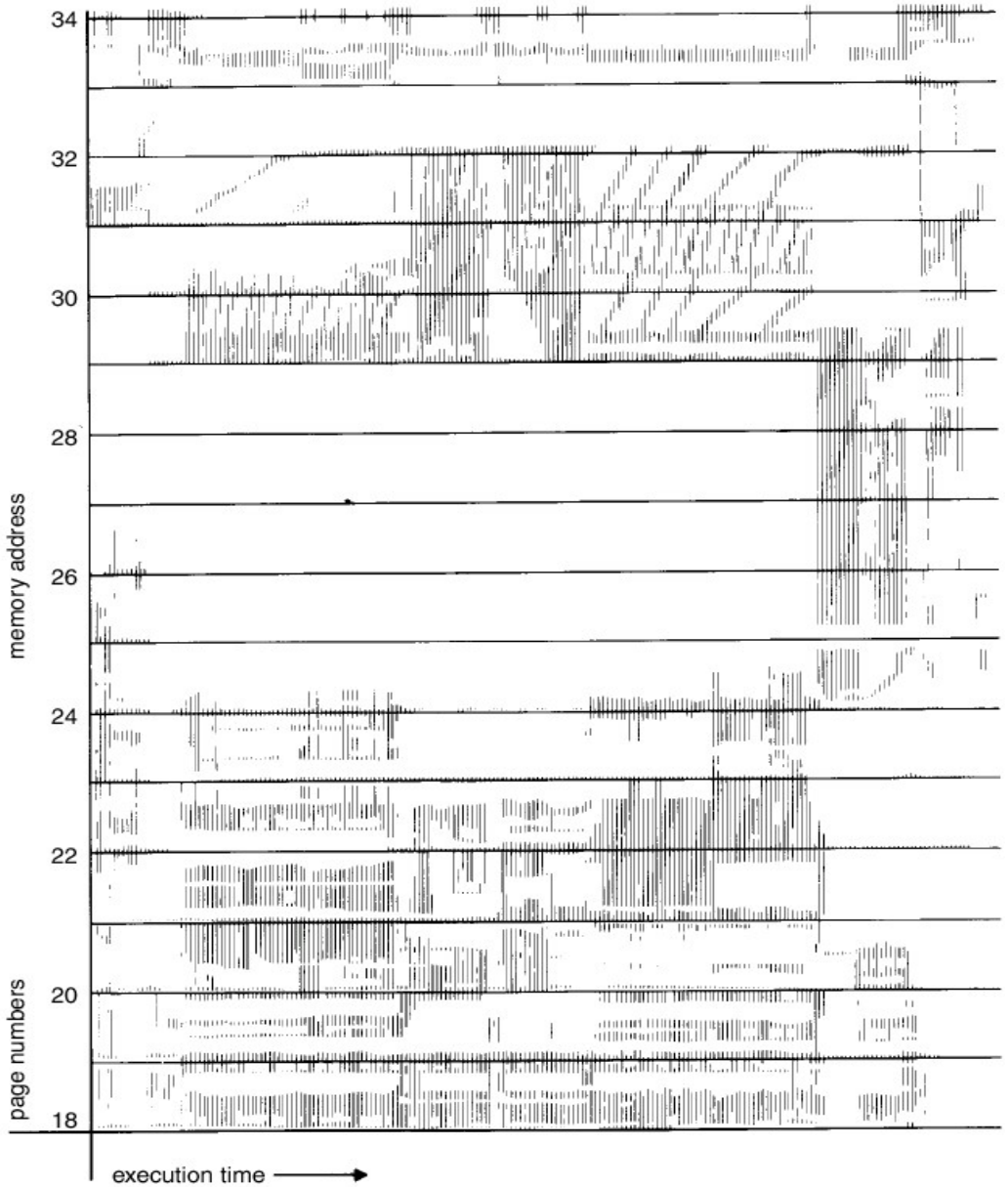
## Why does this work? → Locality

- ✓ Temporal locality: locations referenced recently tend to be referenced again soon
- ✓ Spatial locality: locations near recently referenced locations are likely to be referenced soon

## Locality means paging can be infrequent

- ✓ Once you've paged something in, it will be used many times
- ✓ On average, you use things that are paged in
- ✓ But this depends on many things:
  - Degree of locality in application
  - Page replacement policy
  - Amount of physical memory
  - Application's reference pattern and memory footprint

# Locality in a Memory-Reference Pattern



# Demand Paging

## Why is this “demand” paging?

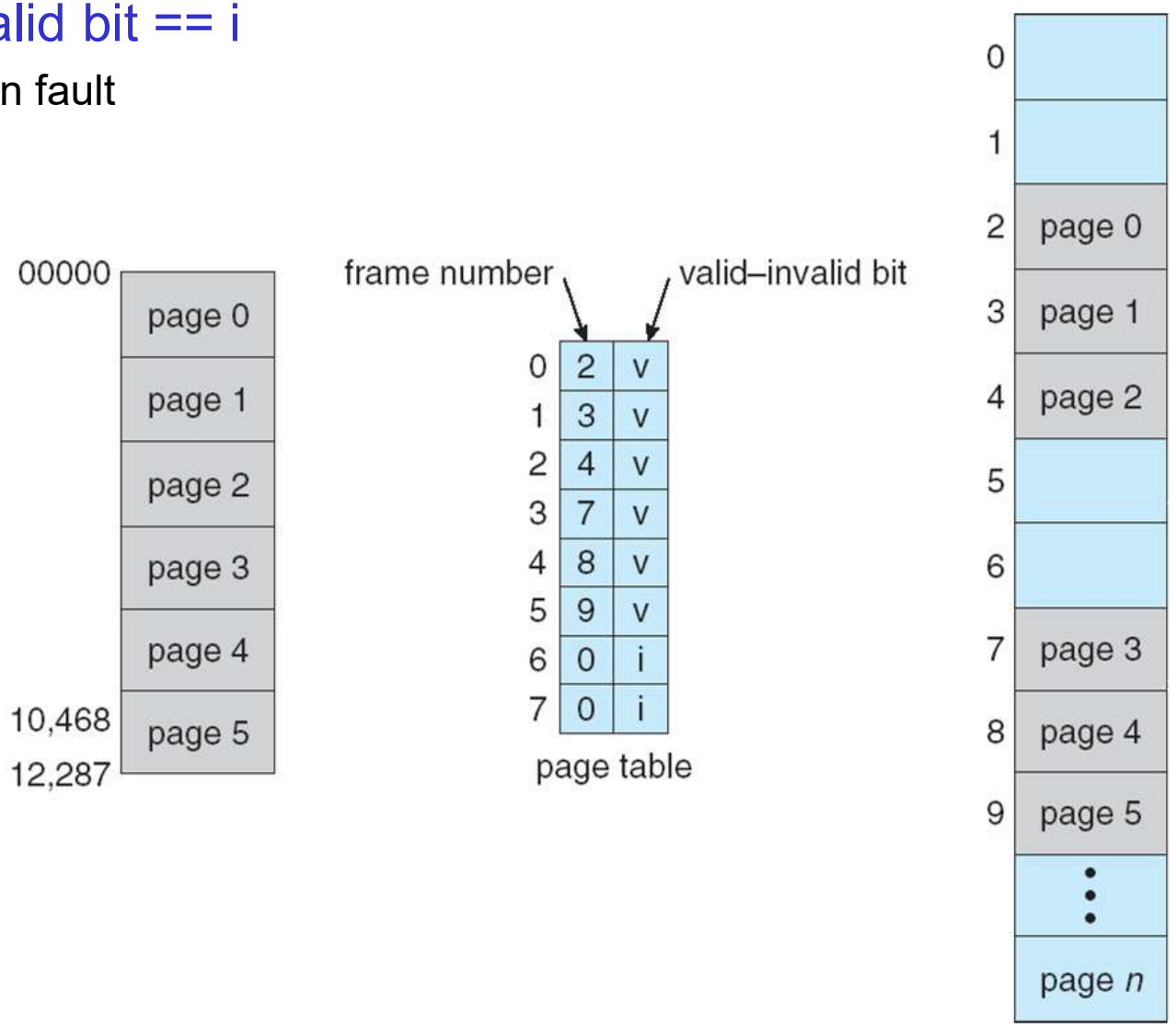
- ✓ When a process first starts up, it has a brand new page table, with all PTE valid bits “false”
  - No pages are yet mapped to physical memory
- ✓ When the process starts executing:
  - Instructions immediately fault on both code and data pages
  - Faults stop when all necessary code/data pages are in memory
  - Only the code/data that is needed (demanded!!) by process needs to be loaded
  - What is needed changes over time, of course…



# Revisited: Memory Protection in Paging

if valid-invalid bit == i

- ✓ Protection fault



# Valid-Invalid Bit in Demand Paging

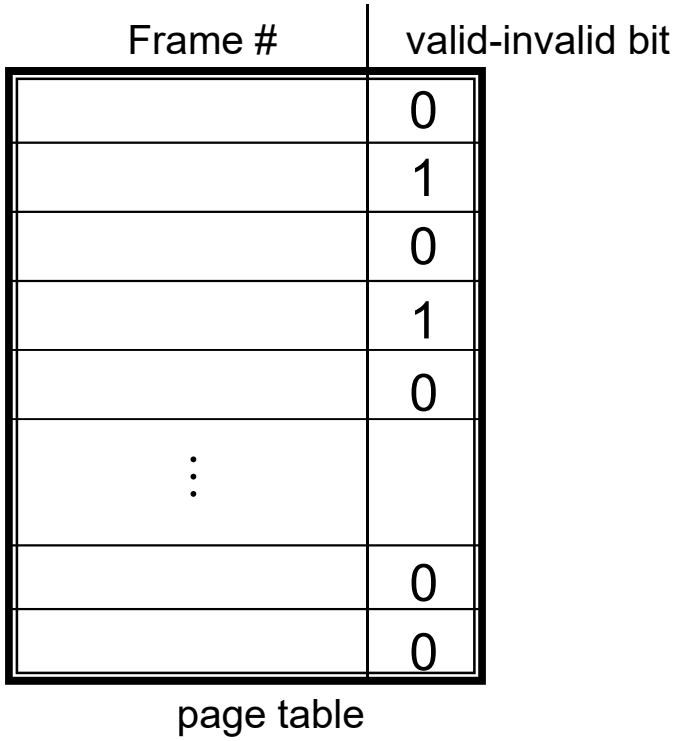
if valid-invalid bit == v

- ✓ in-memory

if valid-invalid bit == i

- ✓ not-in-memory
- ✓ Page fault

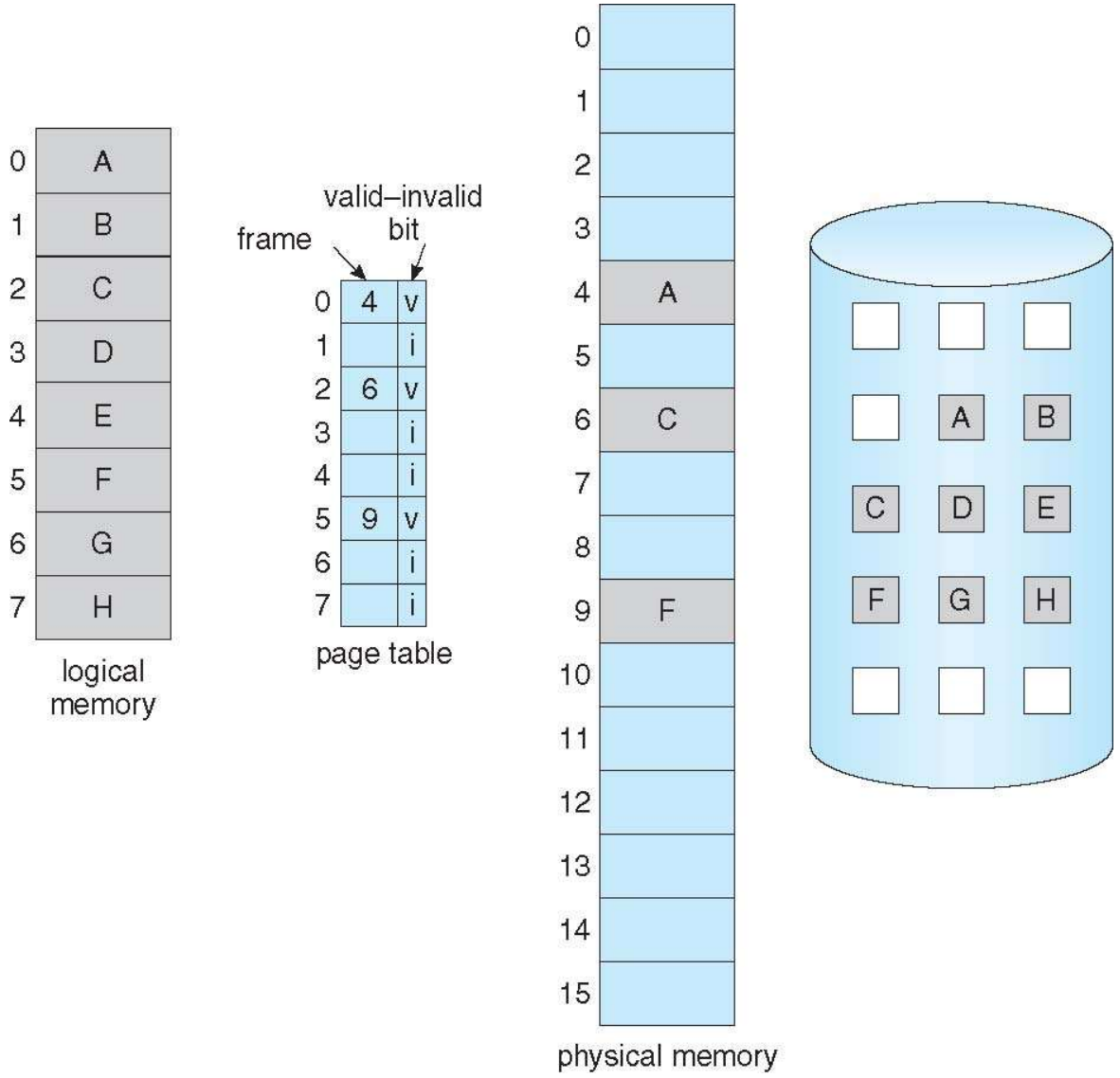
Example of a page table snapshot




Initially, valid–invalid bits are set to invalid on all entries

- ✓ Demand paging

# Valid-Invalid Bit in Demand Paging



# Page Fault

What happens to a process that references a virtual address in a page that has been evicted?

- ✓ When the page was evicted, the OS sets the PTE as invalid and stores (in PTE) the location of the page in the swap file
- ✓ When a process accesses the page, the invalid PTE will cause an exception to be thrown

The OS will run the page fault handler in response

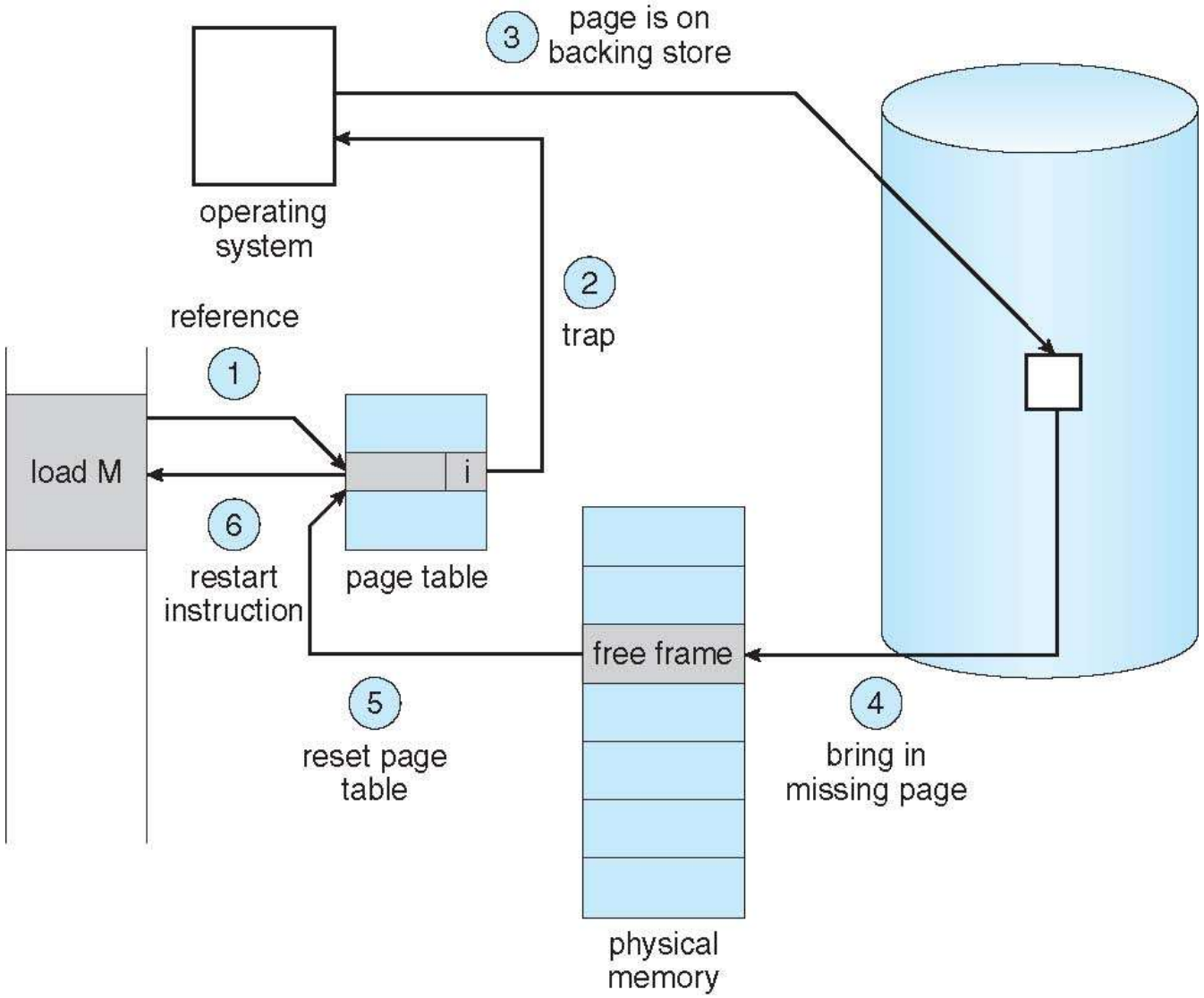
- ✓ Handler uses invalid PTE to locate page in swap file
- ✓ Handler reads page into a physical frame, updates PTE to point to it and to be valid
- ✓ Handler restarts the faulted process

Where does the page that's read in go?

- ✓ Have to evict something else (page replacement algorithm)
- ✓ OS typically tries to keep a pool of free pages around so that allocations don't inevitably cause evictions



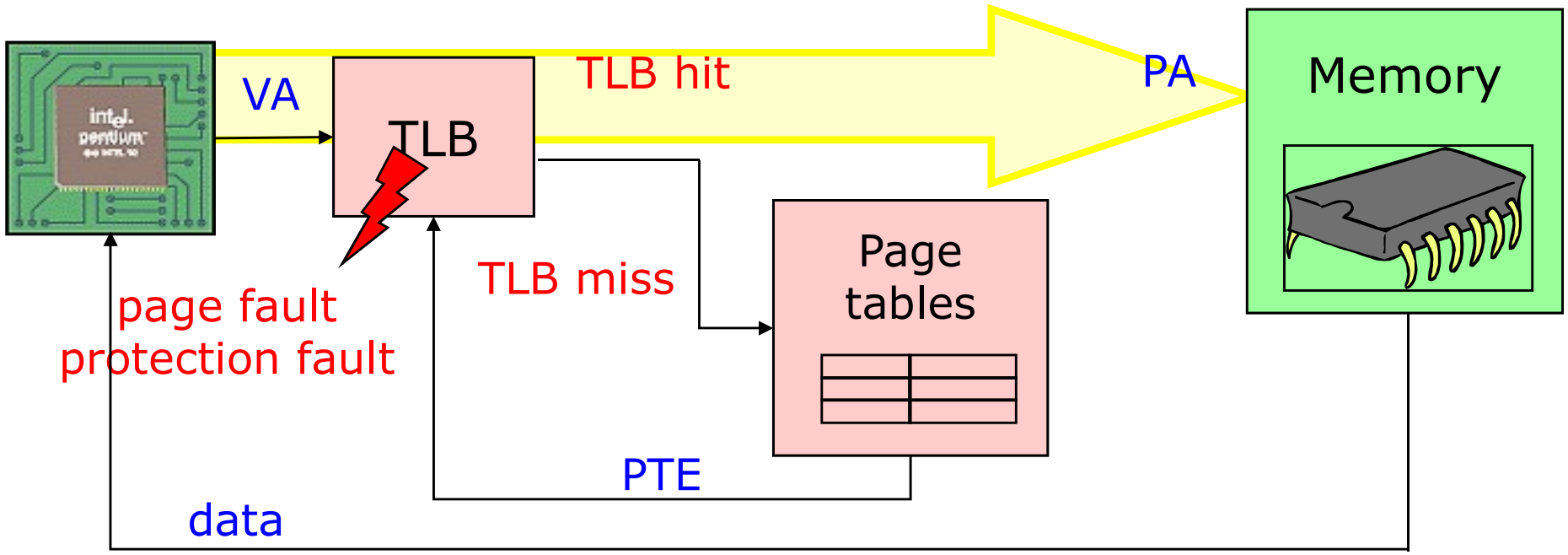
# Steps in Handling a Page Fault



# Memory Reference

## Situation

- ✓ Process is executing on the CPU, and it issues a read to a (virtual) address



# Memory Reference

## The common case

- ✓ The read goes to the TLB in the MMU
- ✓ TLB does a lookup using the page number of the address
- ✓ The page number matches, returning a PTE
- ✓ TLB validates that the PTE protection allows reads
- ✓ PTE specifies which physical frame holds the page
- ✓ MMU combines the physical frame and offset into a physical address
- ✓ MMU then reads from that physical address, returns value to CPU



# Memory Reference

## TLB misses: two possibilities

- ✓ (1) MMU loads PTE from page table in memory
  - Hardware managed TLB, OS not involved in this step
  - OS has already set up the page tables so that the hardware can access it directly
- ✓ (2) Trap to the OS
  - Software managed TLB, OS intervenes at this point
  - OS does lookup in page tables, loads PTE into TLB
  - OS returns from exception, TLB continues

- ✓ At this point, there is a valid PTE for the address in the TLB



# Memory Reference

## TLB misses

- ✓ Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out
  - Assuming page tables are in OS virtual address space
  - Not a problem if tables are in physical memory
- ✓ When TLB has PTE, it restarts translation
  - Common case is that the PTE refers to a valid page in memory
  - Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)

# Memory Reference

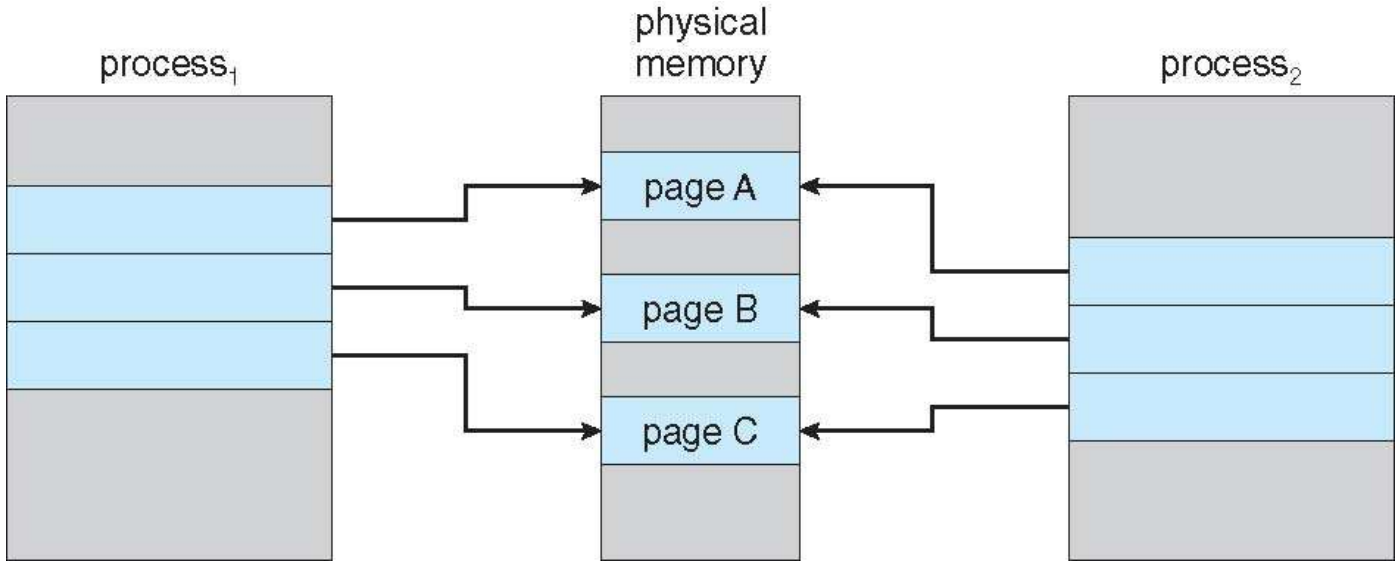
## Page faults

- ✓ PTE can indicate a protection fault
  - Read/Write/Execute – operation not permitted on page
  - Invalid – virtual page not allocated, or page not in physical memory
- ✓ TLB traps to the OS (software takes over)
  - Read/Write/Execute – OS usually will send fault back to the process, or might be playing tricks (e.g., copy on write, mapped files)
  - Invalid (Not allocated) – OS sends fault to the process (e.g., segmentation fault)
  - Invalid (Not in physical memory) – OS allocates a frame, reads from disk, and maps PTE to physical frame

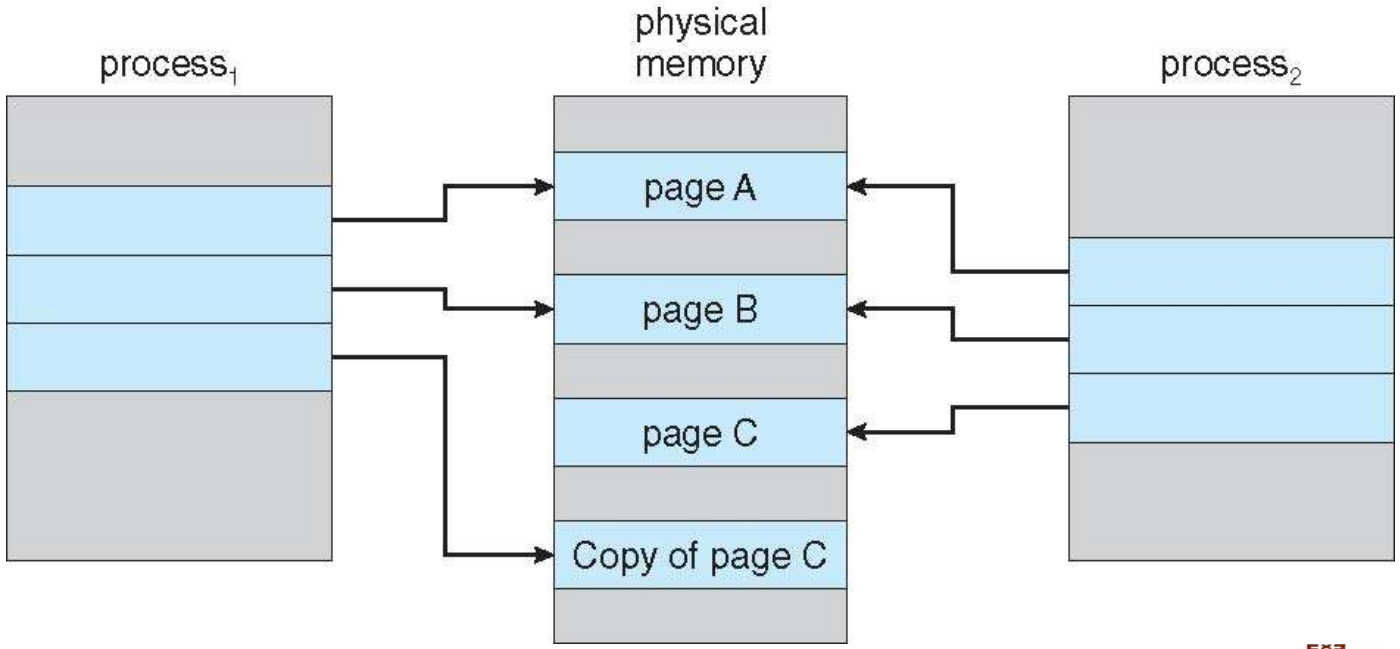


# Copy-on-Write

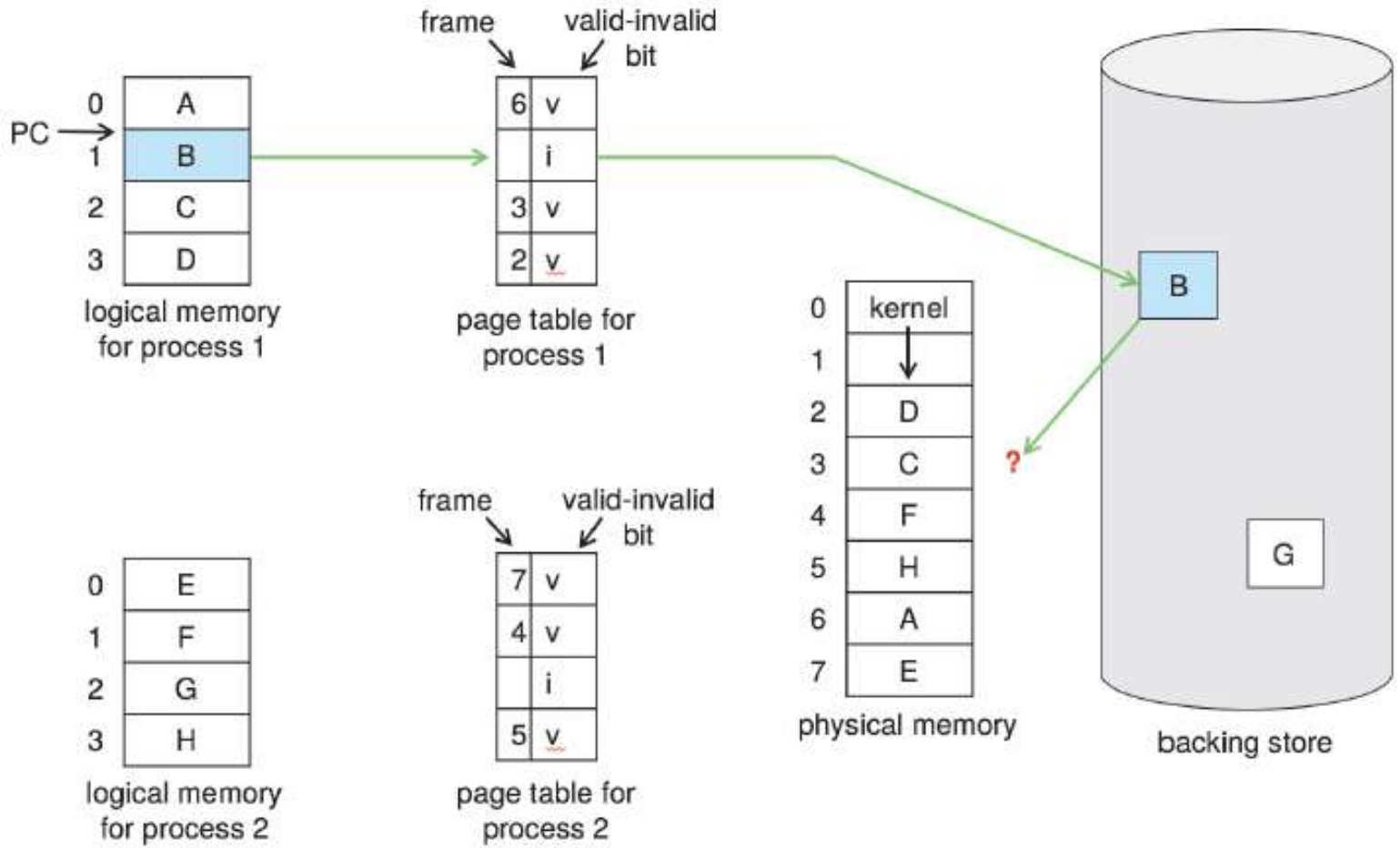
Right after fork( )




When process 1 modifies page C



# What happens if there is no free frame?



# Page Replacement

When a page fault occurs, the OS loads the faulted page from disk into a page frame of memory

At some point, the process has used all of the page frames it is allowed to use

When this happens, the OS must replace a page for each page faulted in

- ✓ It must evict a page to free up a page frame

The page replacement algorithm determines how this is done



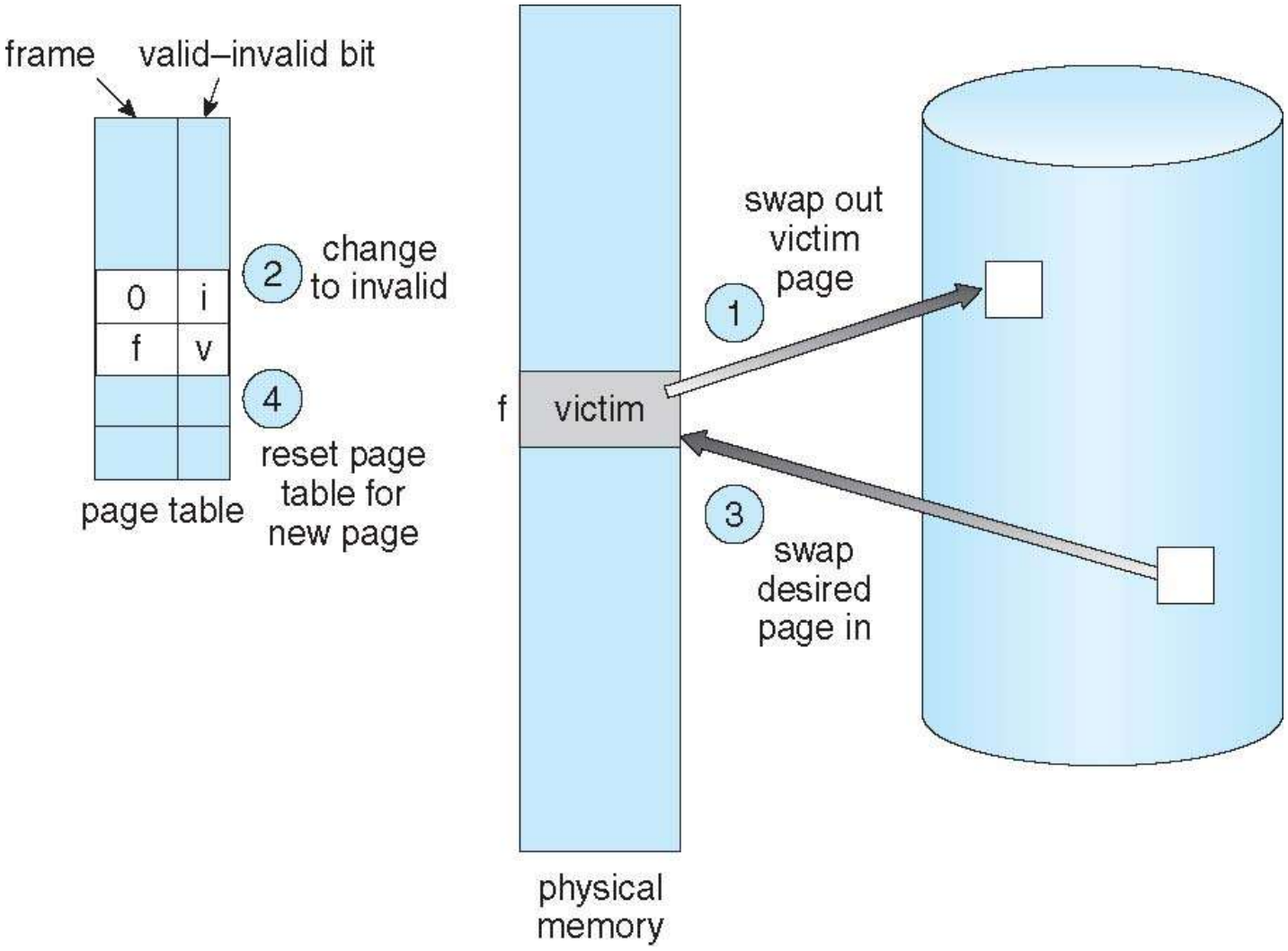
# Page Replacement

## Evicting the best page

- ✓ The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove
- ✓ The best page to evict is the one never touched again
  - as process will never again fault on it
- ✓ "Never" is a long time, so picking the page closest to "never" is the next best thing
  - Belady's proof: Evicting the page that won't be used for the longest period of time minimizes the number of page faults



# Page Replacement



# Performance of Demand Paging

Page Fault Rate $p$, $0 \le p \le 1$

Effective Access Time (EAT)

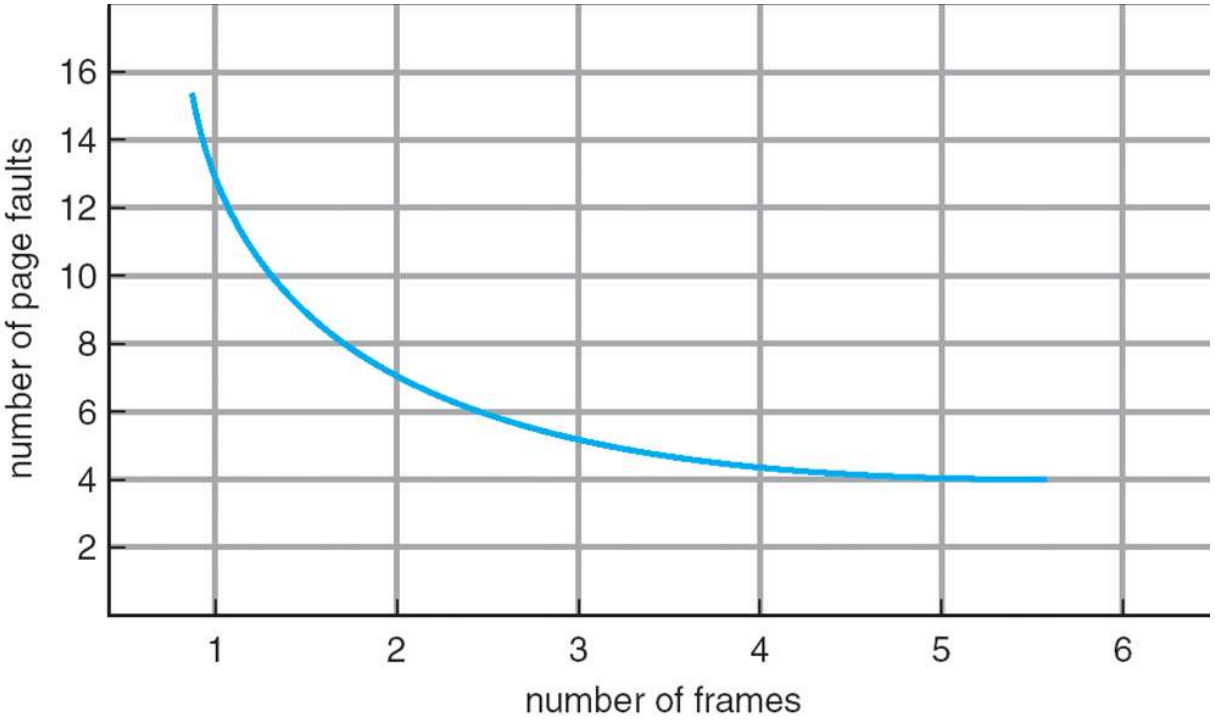
EAT = (1 – $p$) x memory access
+ $p$ x (page fault overhead
+ [swap page out ]
+ swap page in
+ restart overhead)

# Page Replacement Algorithms

## Goal: lowest page-fault rate

- ✓ Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- ✓ In all our examples, the reference string is "1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5"

## Page faults vs. Number of frames



# FIFO

## Obvious and simple to implement

- ✓ Maintain a list of pages in order they were paged in
- ✓ On replacement, evict the one brought in longest time ago

## Why might this be good?

- ✓ Maybe the one brought in the longest ago is not being used

## Why might this be bad?

- ✓ Maybe, it's not the case
- ✓ We don't have any information either way

## FIFO suffers from "Belady's Anomaly"

- ✓ The fault rate might increase when the algorithm is given more memory



# FIFO

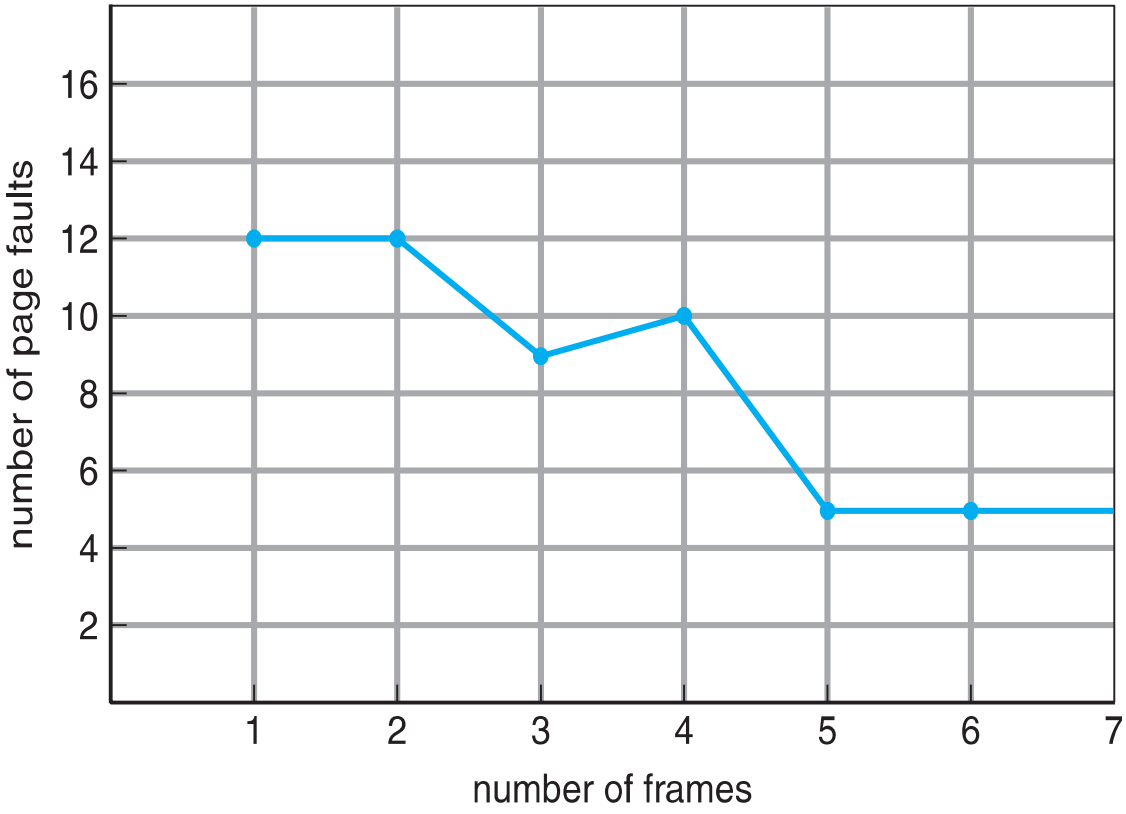
## Example: Belady's anomaly

- ✓ Reference string: 1,2,3,4,1,2,5,1,2,3,4,5
- ✓ 3 frames: 9 faults

| | | |
|---|---|---|
| 1 | 4 | 5 |
| 2 | 1 | 3 |
| 3 | 2 | 4 |

- ✓ 4 frames: 10 faults

| | | |
|---|---|---|
| 1 | 5 | 4 |
| 2 | 1 | 5 |
| 3 | 2 | |
| 4 | 3 | |

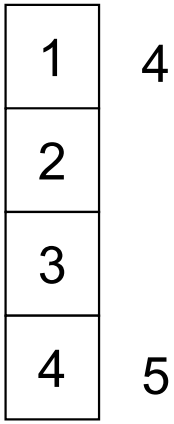


# Optimal Algorithm

Replace page that will not be used for longest period of time

4 frames example

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




6 page faults

How do you know this?

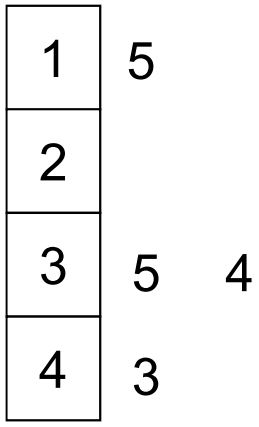
Used for measuring how well your algorithm performs

# Least Recently Used (LRU) Algorithm

LRU uses reference information to make a more informed replacement decision

- ✓ Idea: past experience gives us a guess of future behavior
- ✓ On replacement, evict the page that has not been used for the longest time in the past
- ✓ LRU looks at the past, Belady's wants to look at future

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



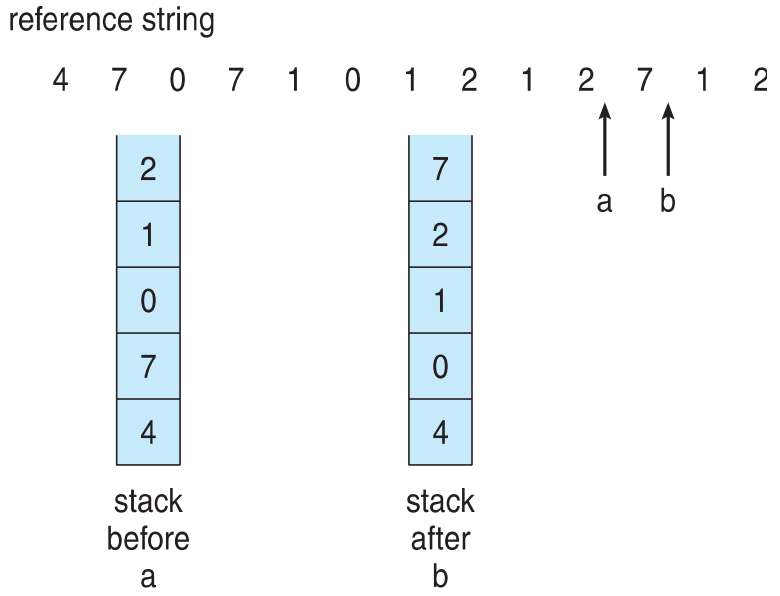
# Implementation of LRU Algorithm

## Timestamp implementation

- ✓ Every page entry has 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 determine which are to change

## Stack implementation

- ✓ Keep a stack of page numbers
- ✓ Page referenced:
  - move it to the top
  - requires 6 pointers to be changed
- ✓ No search for replacement




## Approximation

- ✓ To be perfect, need to timestamp every reference and put it in the PTE (or maintain a stack) – too expensive
- ✓ So, we need an approximation

# LRU Approximation Algorithms

## Reference bit

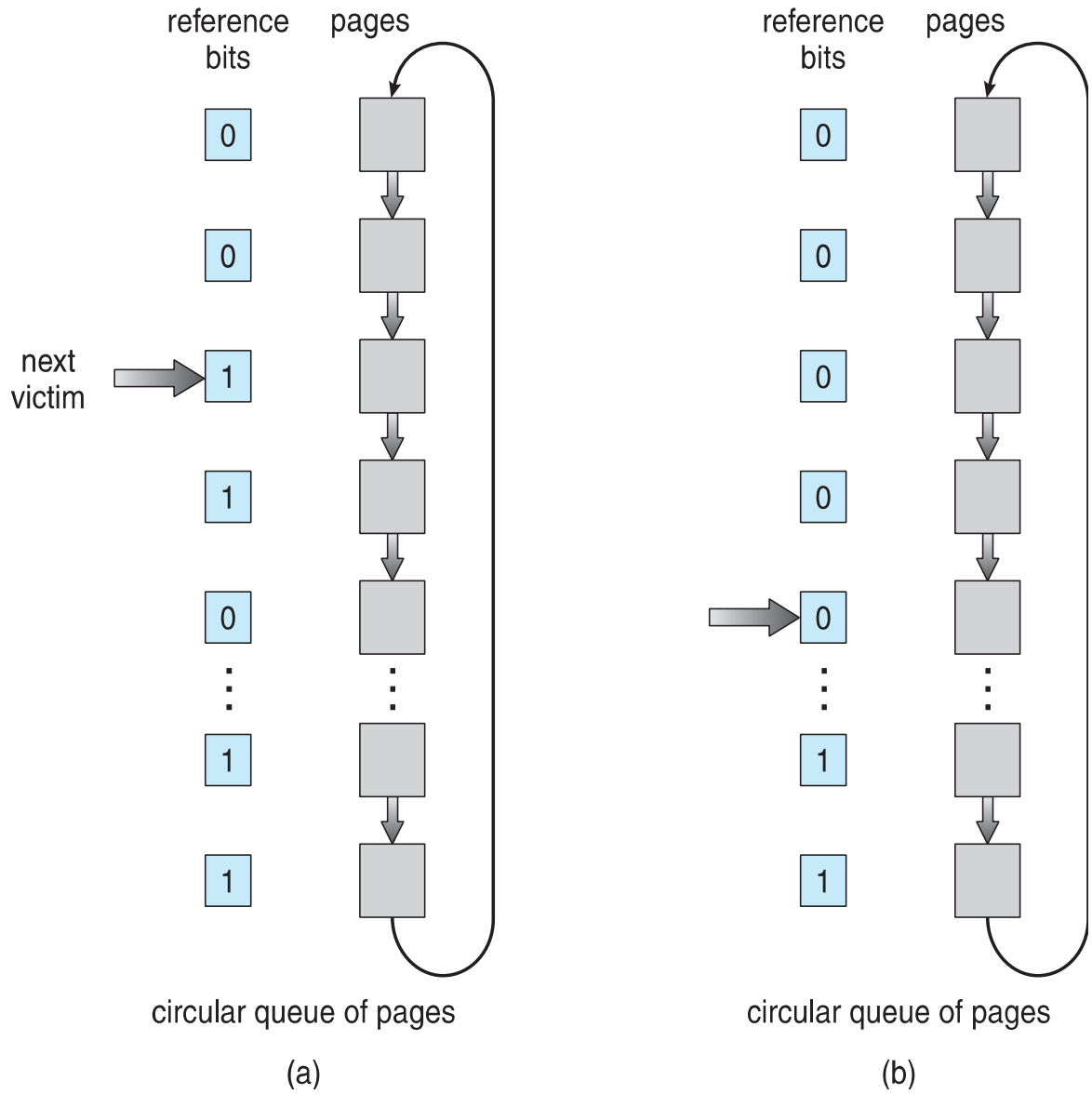
- ✓ With each page associate a bit, initially = 0
- ✓ When page is referenced bit set to 1
- ✓ Replace the one which is 0 (if one exists). We do not know the order, however

## Second chance (or LRU clock)

- ✓ 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
  - replace next page (in clock order), subject to same rules



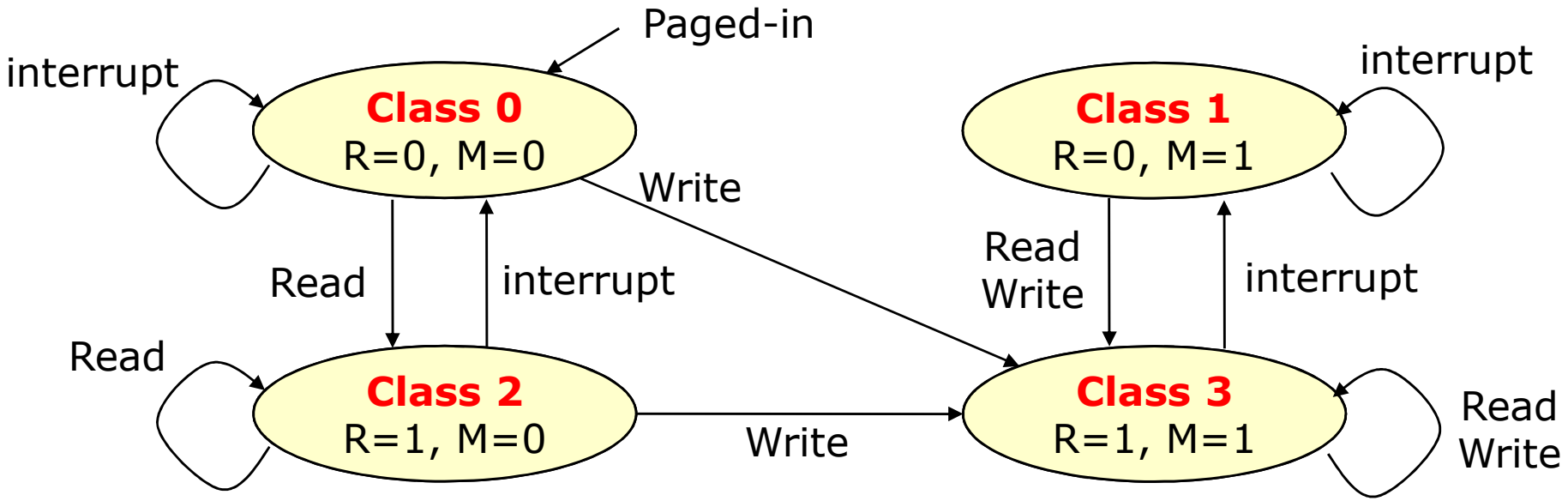
# Second Chance (LRU Clock)



# Not Recently Used (NRU)

## NRU or enhanced second chance

- ✓ Use R (reference) and M (modify) bits
  - Periodically, (e.g., on each clock interrupt), R is cleared, to distinguish pages that have not been referenced recently from those that have been



# Not Recently Used

## Algorithm

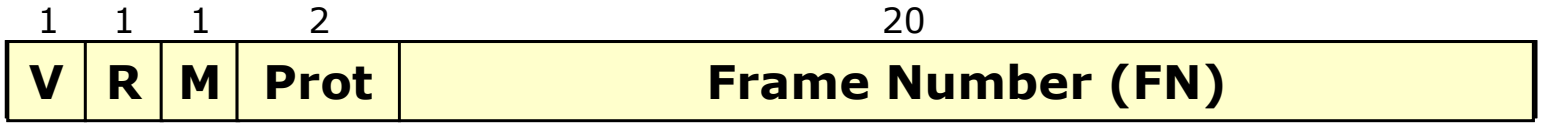
- ✓ Removes a page at random from the lowest numbered nonempty class
- ✓ It is better to remove a modified page that has not been referenced in at least one clock tick than a clean page that is in heavy use

## Advantages

- ✓ Easy to understand
- ✓ Moderately efficient to implement
- ✓ Gives a performance that, while certainly not optimal, may be adequate



# Revisited: Page Table Entries (PTEs)




Valid bit (V) says whether or not the PTE can be used

- ✓ It is checked each time a virtual address is used

Reference bit (R) says whether the page has been accessed

- ✓ It is set when a read or write to the page occurs

Modify bit (M) says whether or not the page is dirty

- ✓ It is set when a write to the page occurs

Protection bits (Prot) control which operations are allowed on the page

- ✓ Read, Write, Execute, etc.

Frame number (FN) determines physical page

# Least Frequently Used (LFU)

## Counting-based page replacement

- ✓ A software counter is associated with each page
- ✓ At each clock interrupt, for each page, the R bit is added to the counter
  - The counters denote how often each page has been referenced

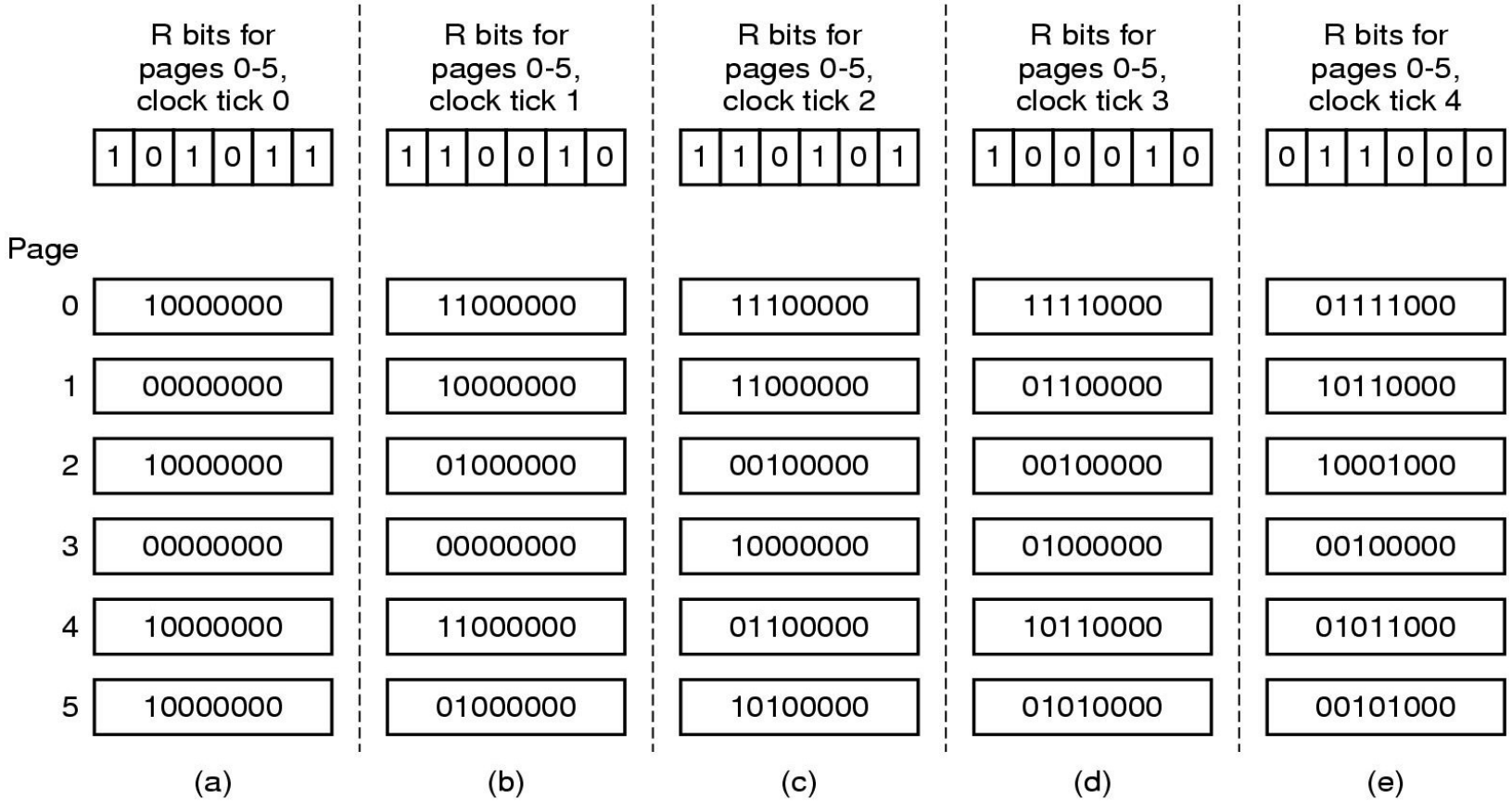
## Least Frequently Used (LFU)

- ✓ The page with the smallest count will be replaced
- ✓ Cf) Most frequently used (MFU) page replacement
  - The page with the largest count will be replaced
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- ✓ It never forgets anything
  - A page may be heavily used during the initial phase of a process, but then is never used again

# Least Frequently Used (LFU)

## Aging

- ✓ The counters are shifted right by 1 bit before the R bit is added to the leftmost



# Allocation of Frames

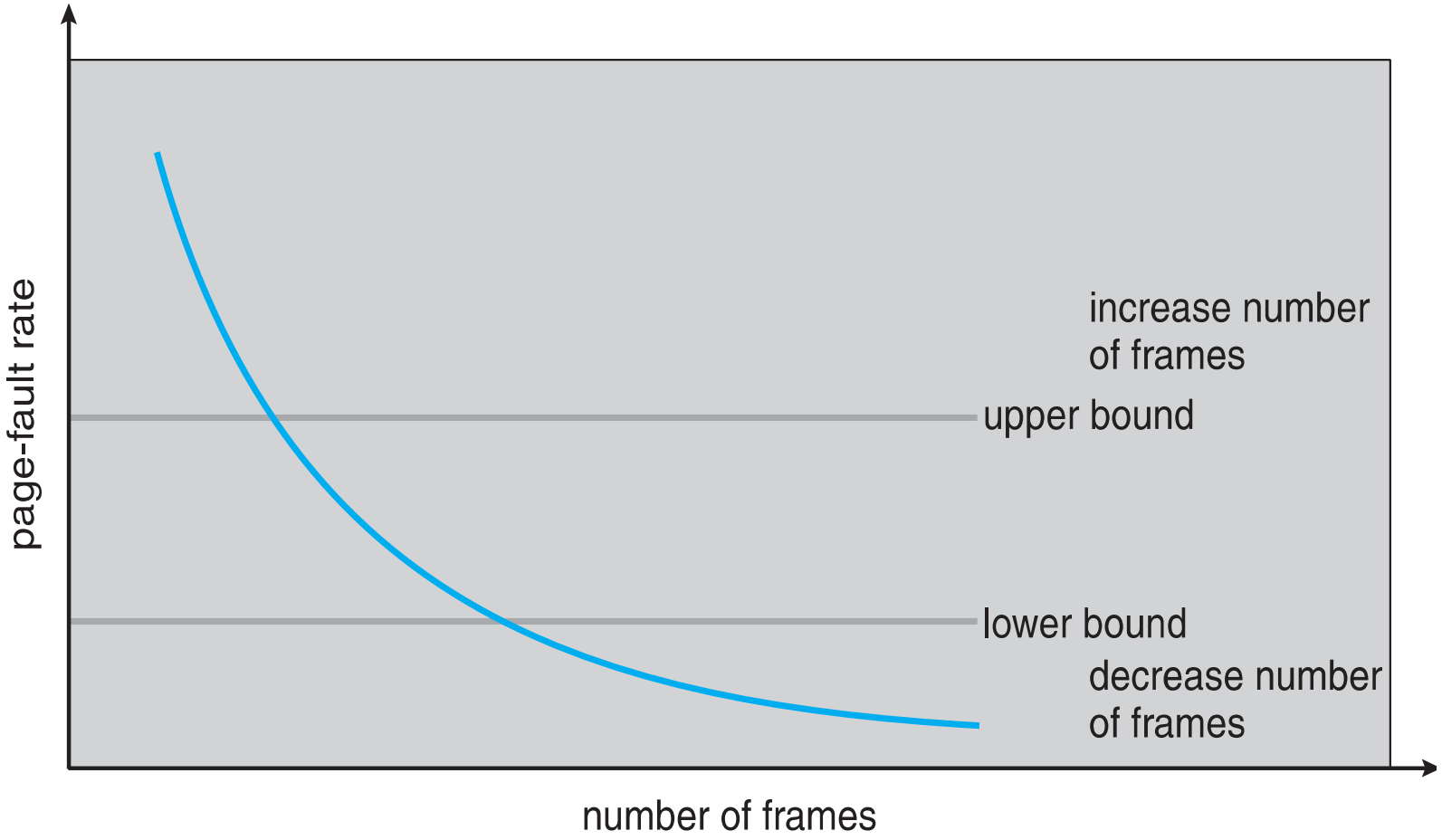
## Allocation algorithms

- ✓ Equal allocation
- ✓ Proportional allocation
  - Allocate according to the size of process
- ✓ Priority-based allocation

## Page replacement

- ✓ Global replacement
- ✓ Local replacement

# Page-Fault Frequency Scheme




Establish “acceptable” page-fault rate

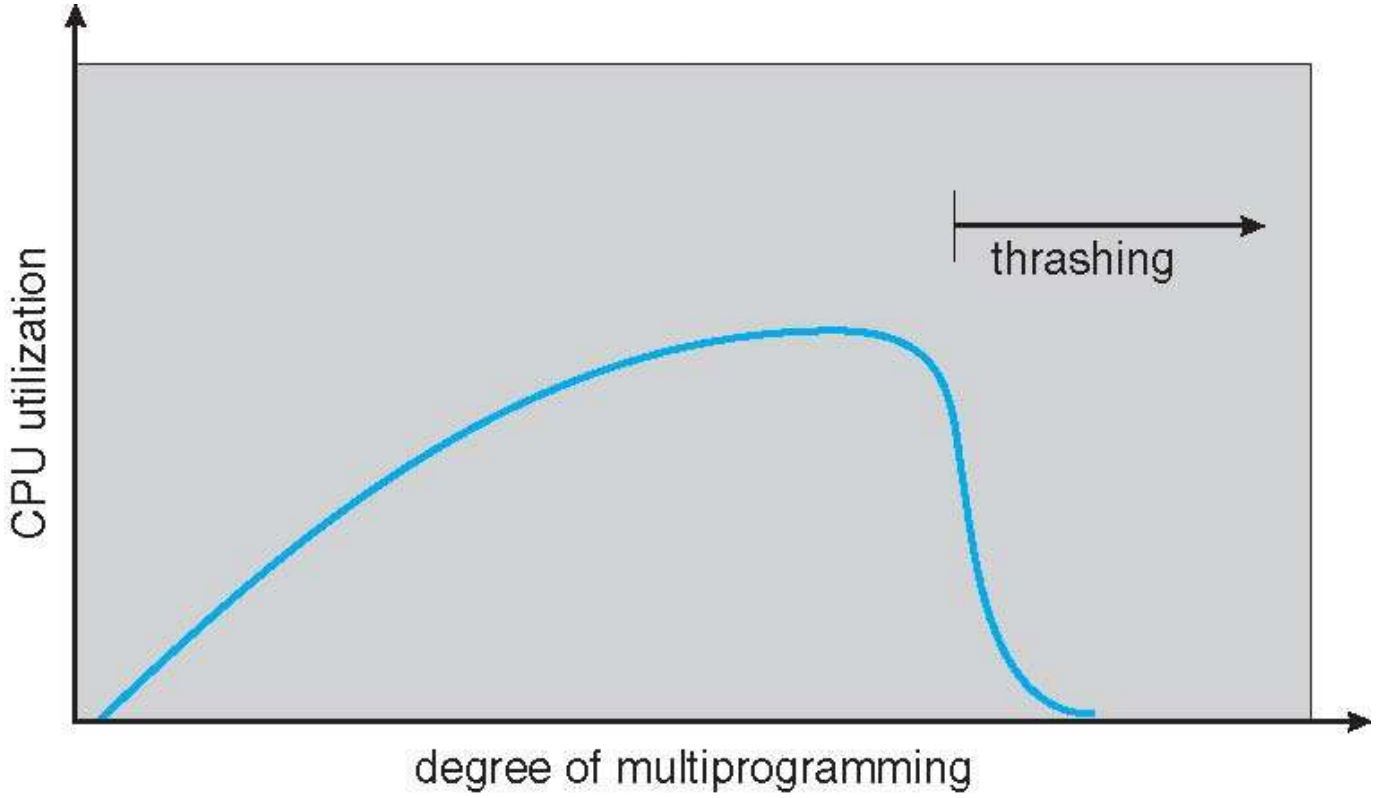
- ✓ If actual rate too low, process loses frame
- ✓ If actual rate too high, process gains frame

# Thrashing

**Thrashing** ≡ a process is busy swapping pages in and out

Why does thrashing occur?

- ✓ Σ size of locality > total memory size

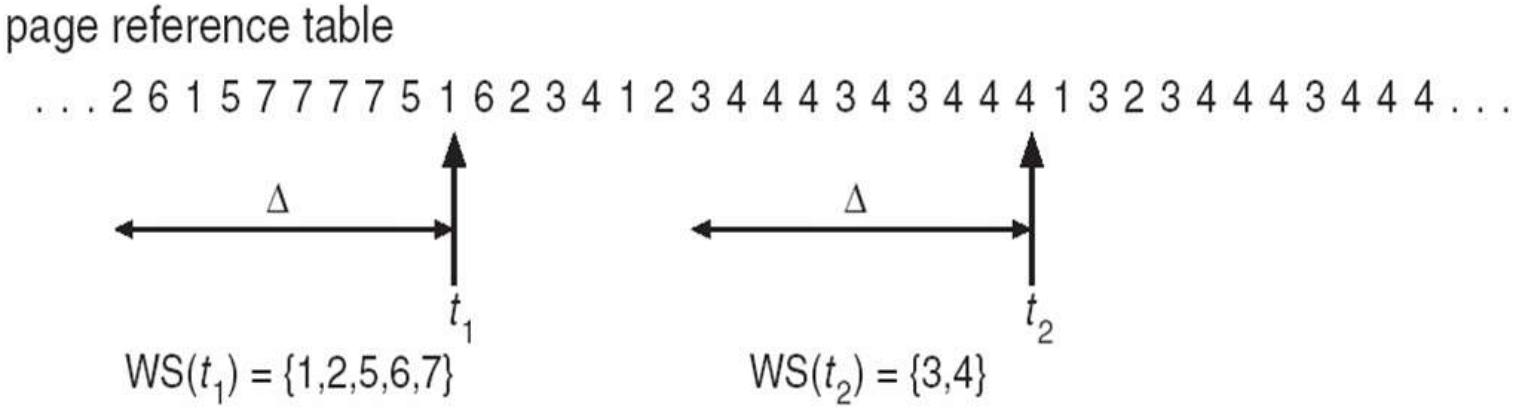


# Working-Set Model

Locality $D = \Sigma\ WSS_i \equiv$ total demand frames

$WSS_i$ (Working Set Size of Process $P_i$) =
total number of pages referenced in the most recent $\Delta$ (varies in time)

- ✓ $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references




- ✓ if $\Delta$ too small will not encompass entire locality
- ✓ if $\Delta$ too large will encompass several localities
- ✓ if $\Delta = \infty \Rightarrow$ will encompass entire program

if $D > m \Rightarrow$ Thrashing

# Other Considerations

## Prepaging

- ✓ Spatial locality

## Page size selection

- ✓ Fragmentation
- ✓ Page table size
- ✓ I/O overhead
- ✓ Locality

## TLB Reach

- ✓ The amount of memory accessible from the TLB
- ✓ TLB Reach = (TLB Size) X (Page Size)
- ✓ Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- ✓ Increase TLB size or Multiple page size

# Other Considerations

## Program structure

- ✓ `int A[1024][1024];`
- ✓ Each row is stored in one page

- ✓ Program 1

```
for (j = 0; j < 1024; j++)
   for (i = 0; i < 1024; i++)
         A[i][j] = 0;
```

1024 x 1024 page faults

- ✓ Program 2

```
for (i = 0; i < 1024; i++)
   for (j = 0; j < 1024; j++)
         A[i][j] = 0;
```

1024 page faults

# Other Considerations

## I/O Interlock

- ✓ Pages must sometimes be locked into memory

## Consider I/O

- ✓ Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm