EDAF35: OPERATING SYSTEMS

# MODULE 6
# MEMORY MANAGEMENT

# CONTENTS

## MEMORY MANAGEMENT

- Background
- Memory Allocation Strategies
- Demand Paging, Copy-on-Write, Page Replacement
- Frame Allocation and Thrashing
- Memory Mapped Files
- Kernel Memory Allocation

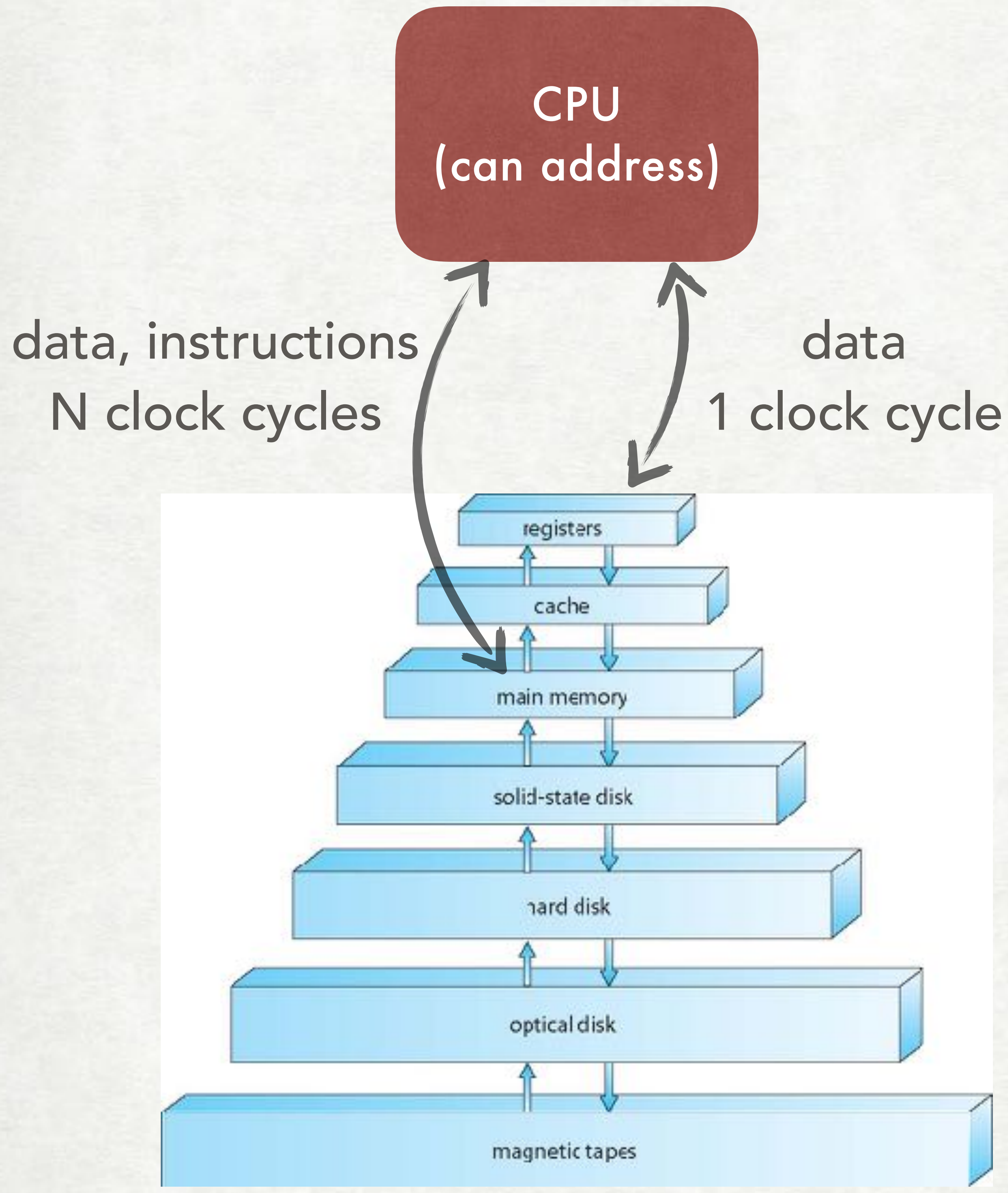
CHAPTER 8
MEMORY-
MANAGEMENT
STRATEGIES

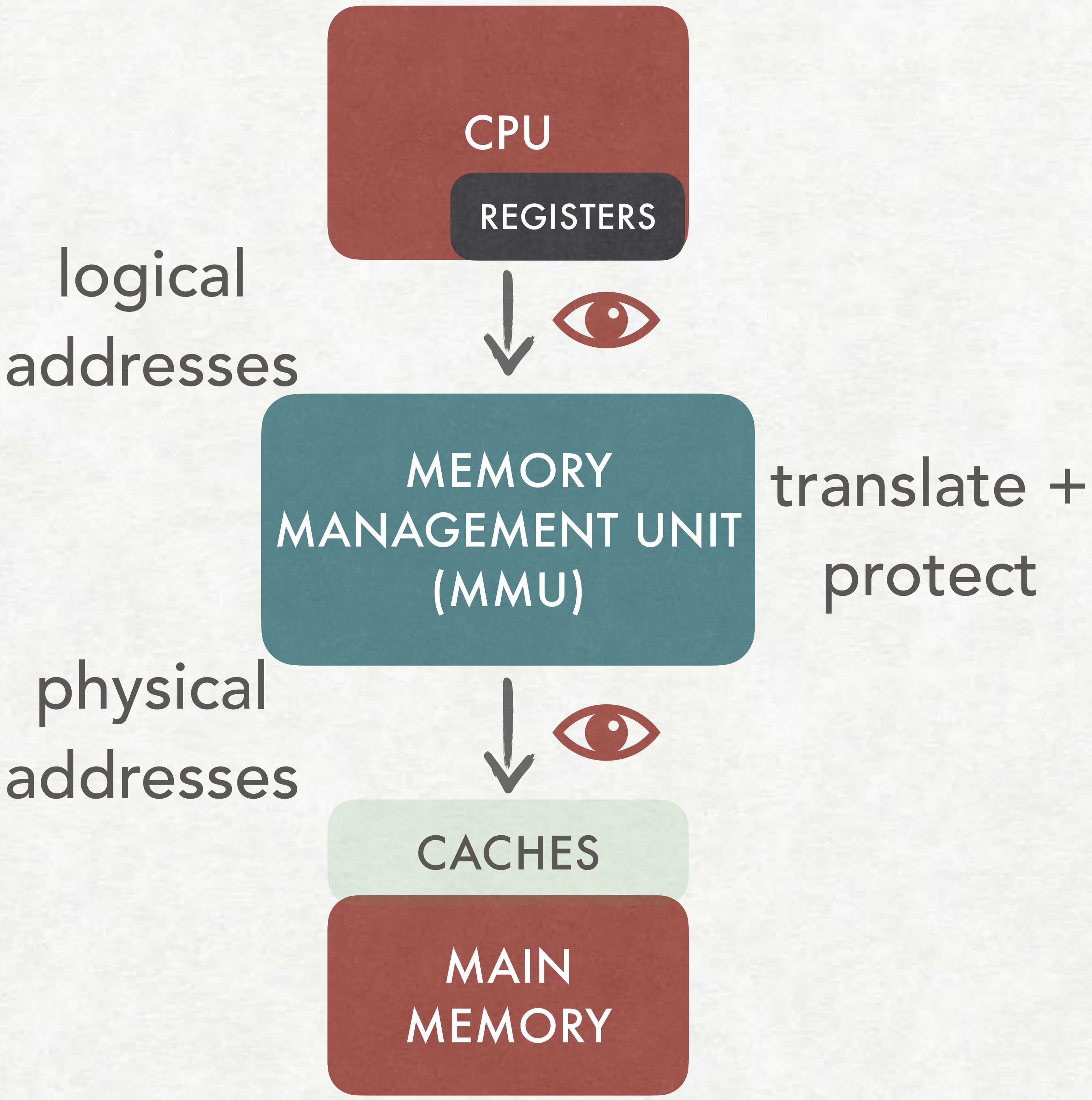
CHAPTER 9
VIRTUAL-MEMORY
MANAGEMENT

Read also: https://www.cs.rutgers.edu/~pxk/416/notes/10-paging.html

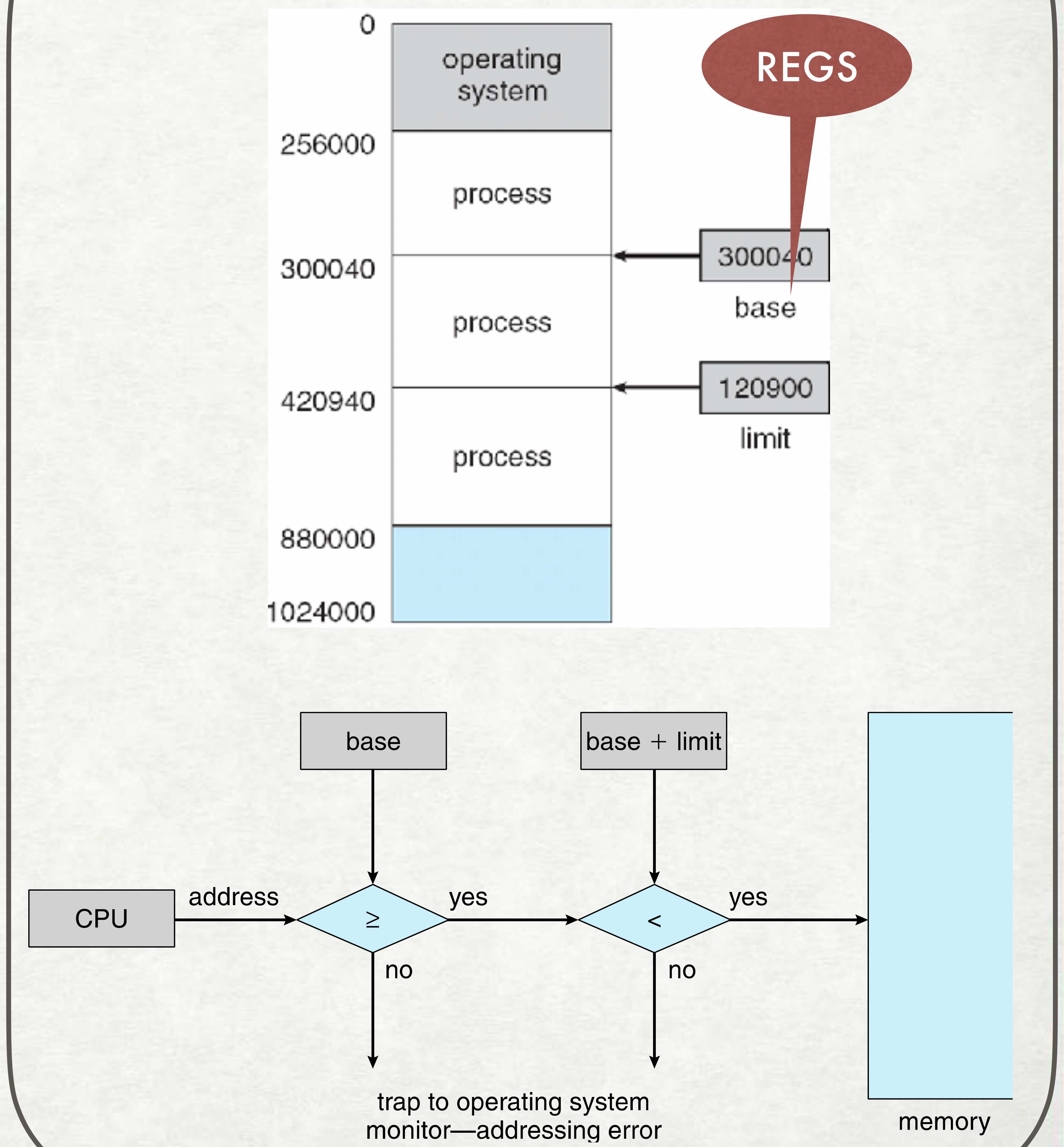
# NOTES ON HARDWARE

## BACKGROUND



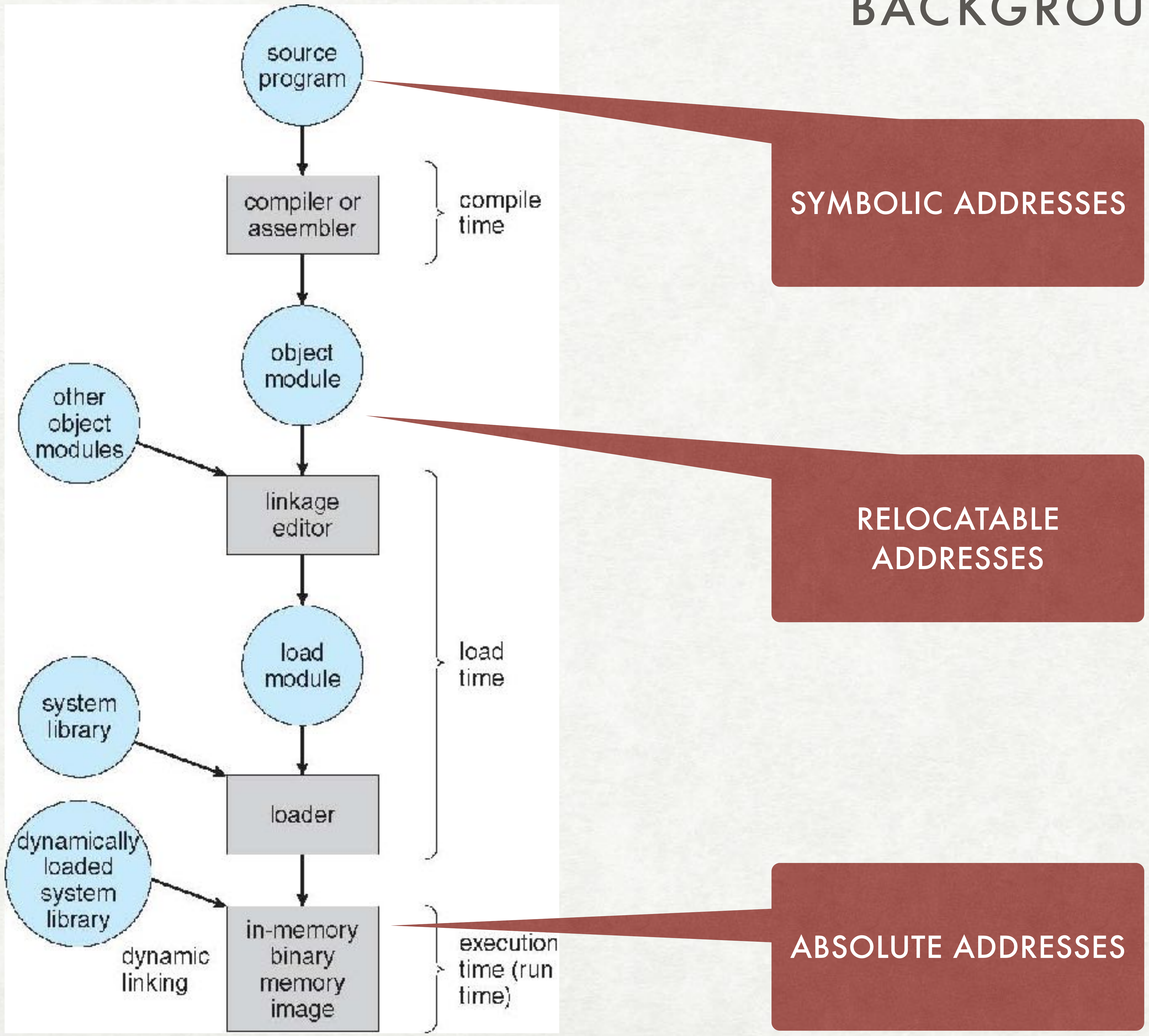




### Example: A simple MMU



# ADDRESS BINDING

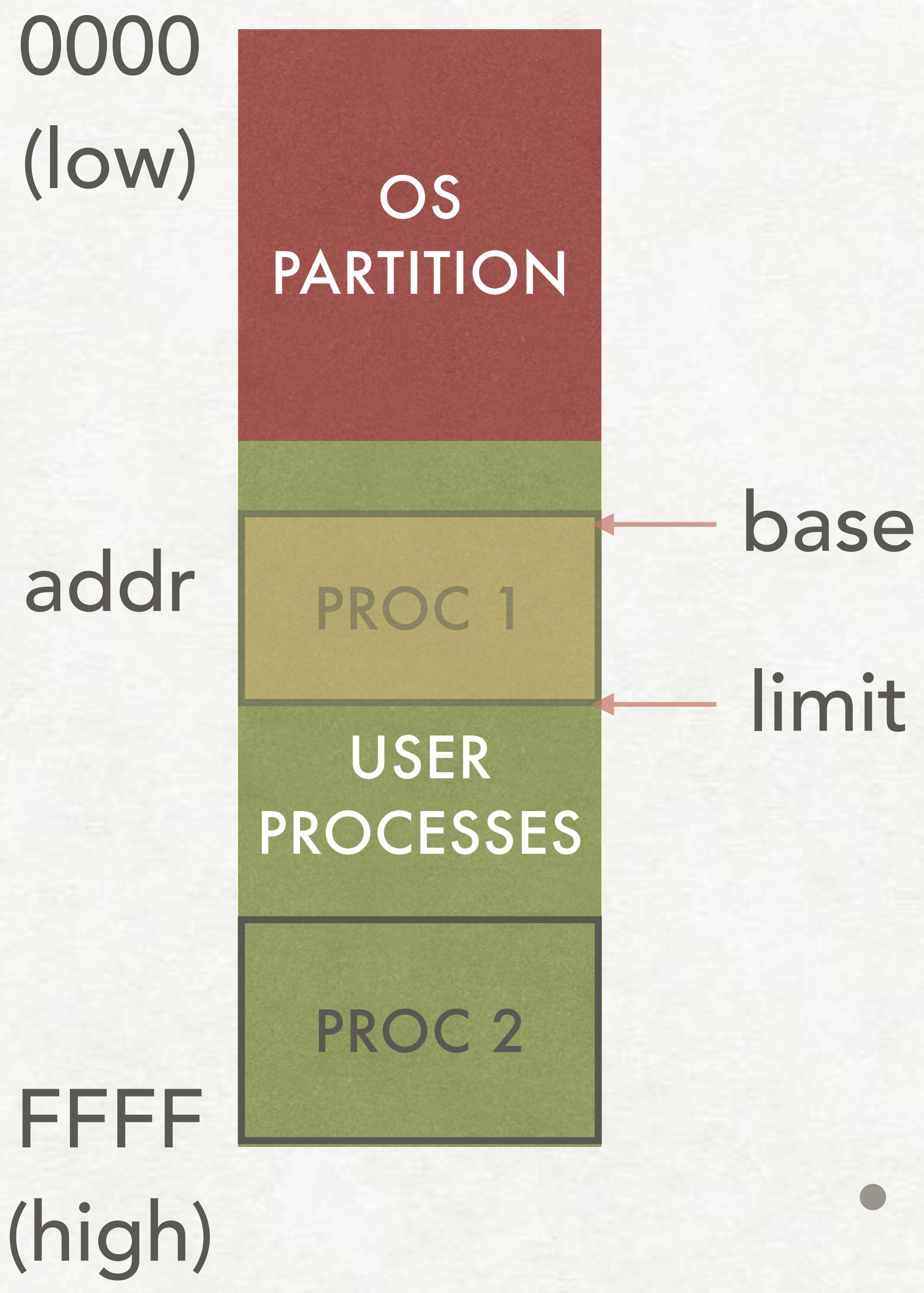
## BACKGROUND

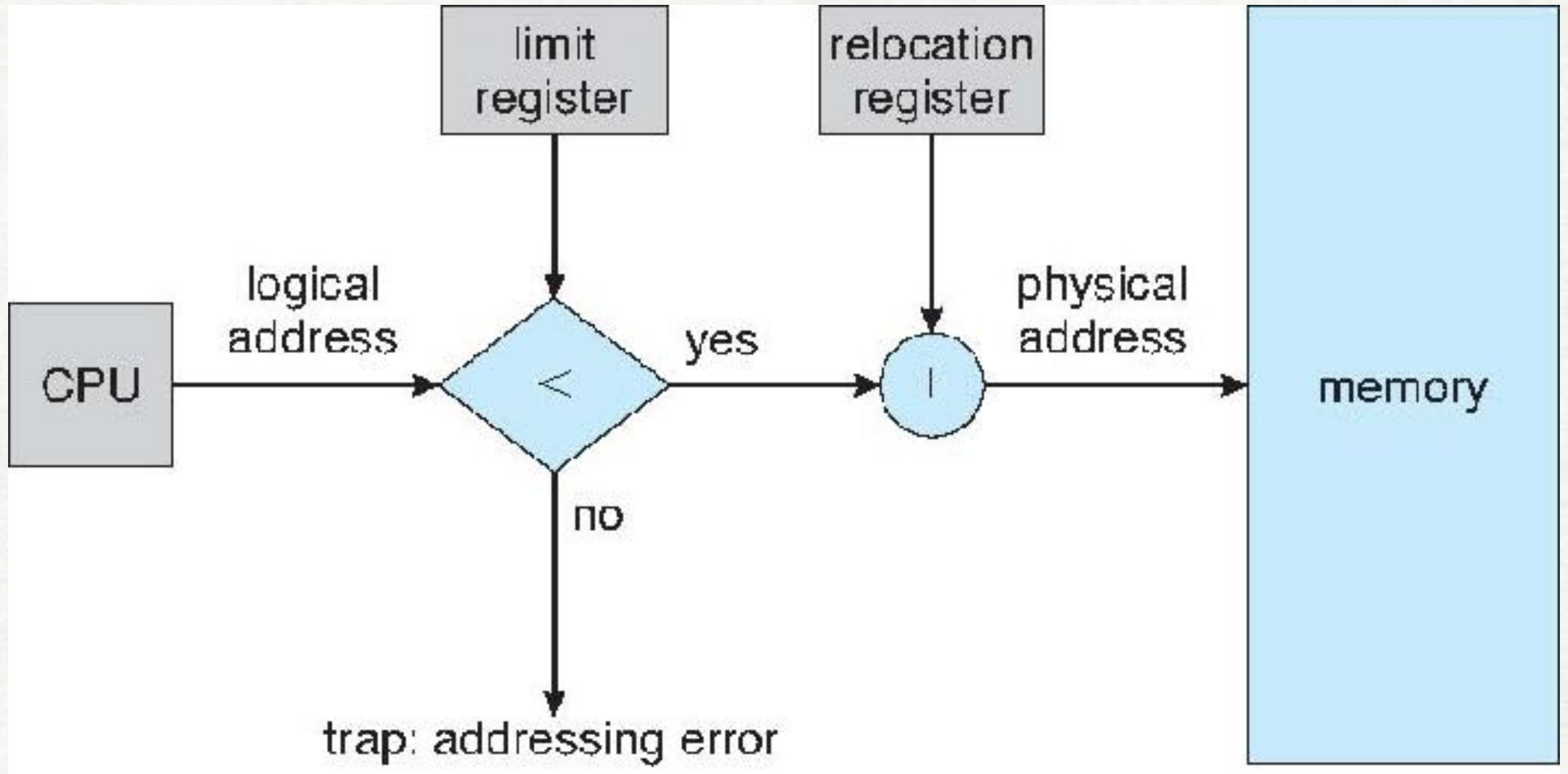



- Symbolic addresses:
  - e.g. variable names — makes it easier to program
- Absolute addresses:
  - logical — usually fixed at runtime
  - physical — may change (move)

# CONTIGUOUS ALLOCATION

## STRATEGIES




protection scheme

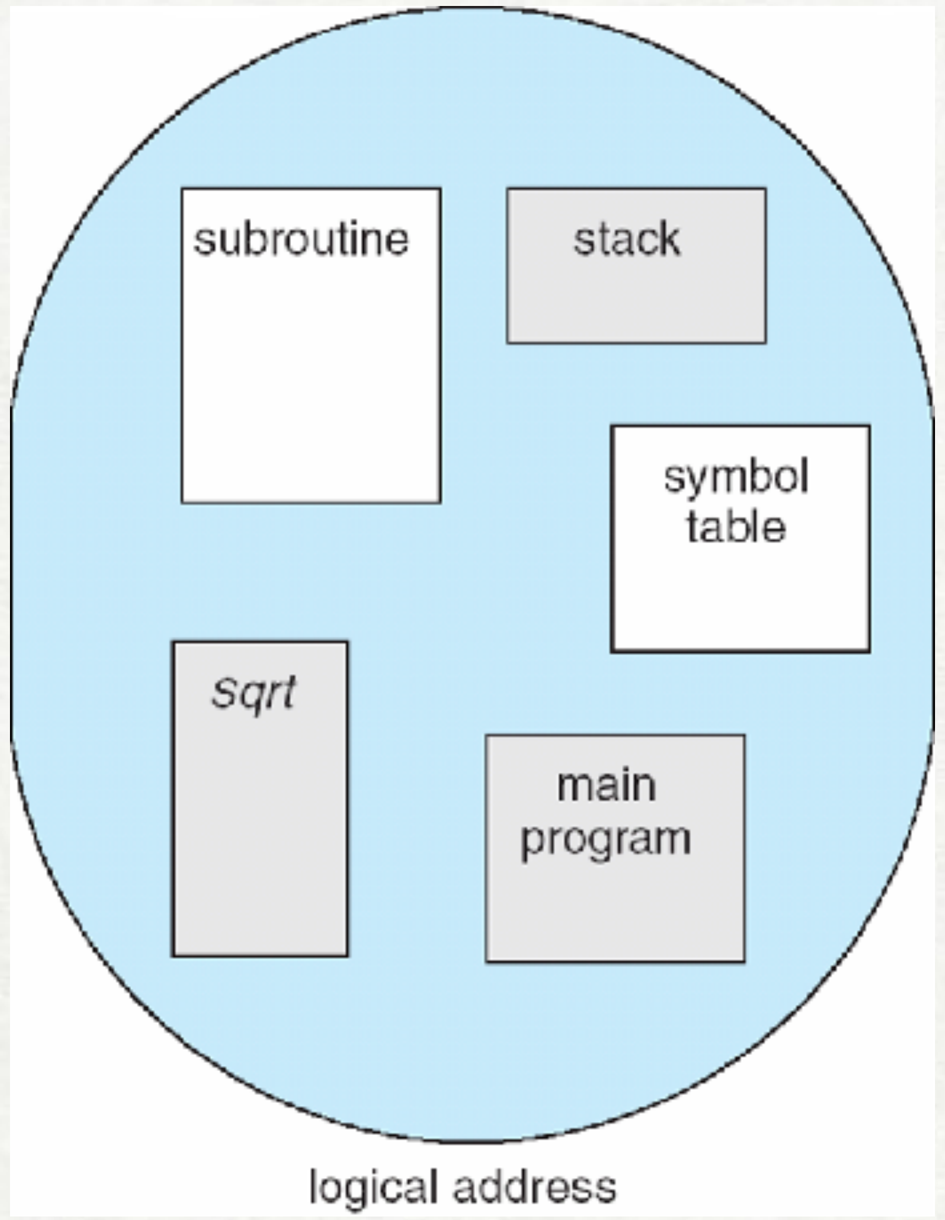
- where to place a new process?
  *first fit, best fit, worst fit*

- *external* fragmentation
  — wasted memory (no process fits there)

# SEGMENTATION

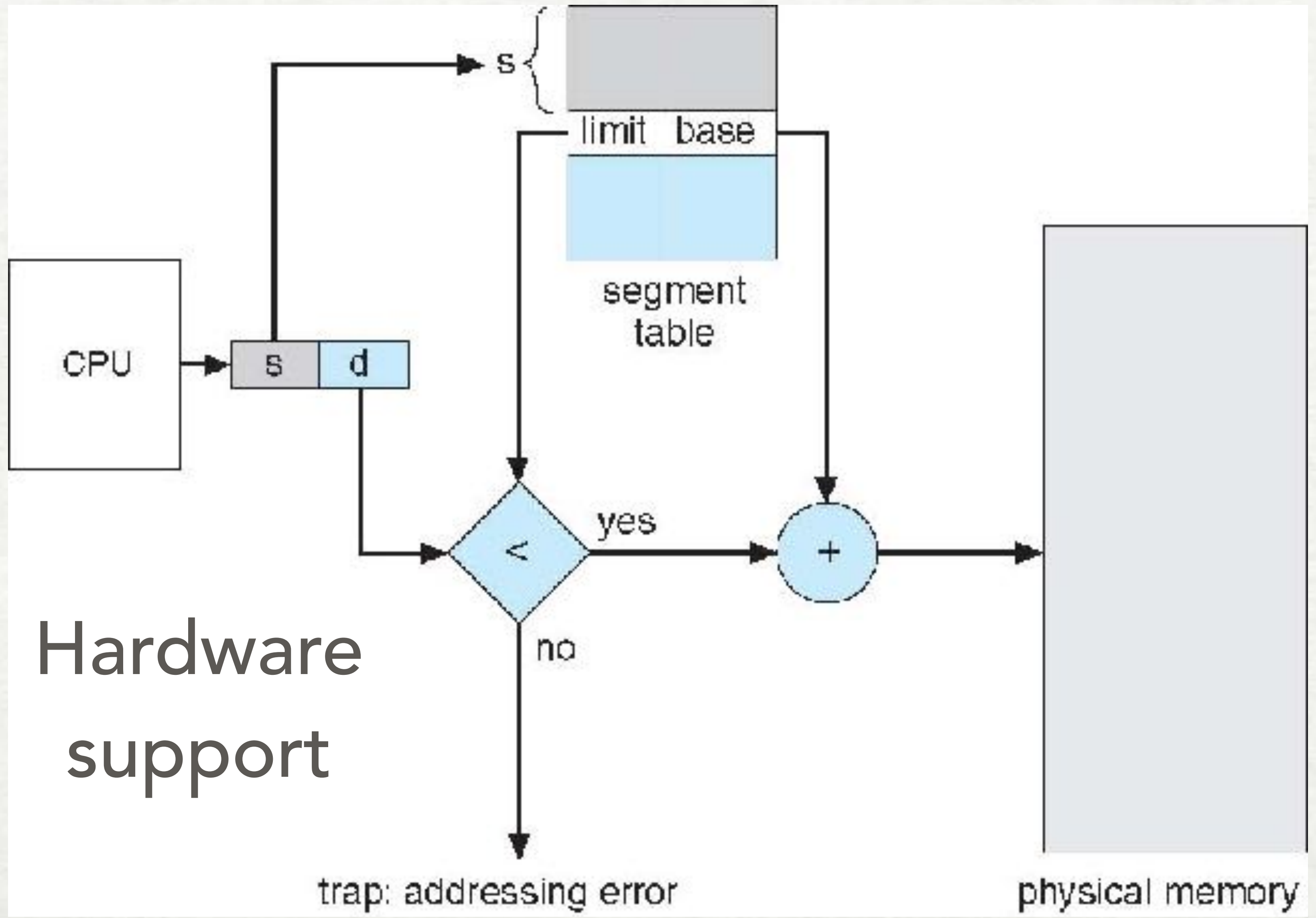
## STRATEGIES

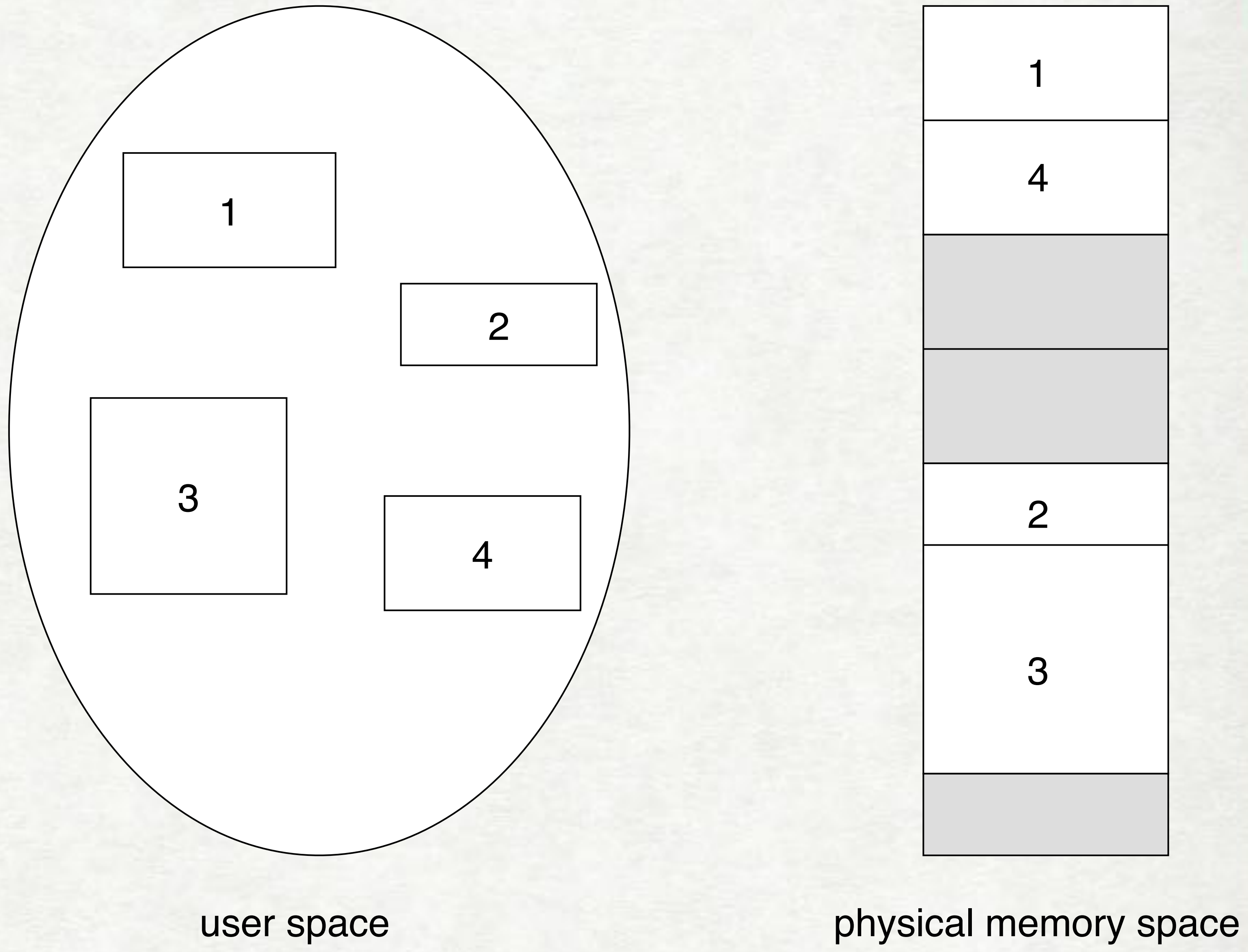
a process




- split a process in segments — logical ranges (also the programmer's view)
- place segments separately in memory

Hardware support



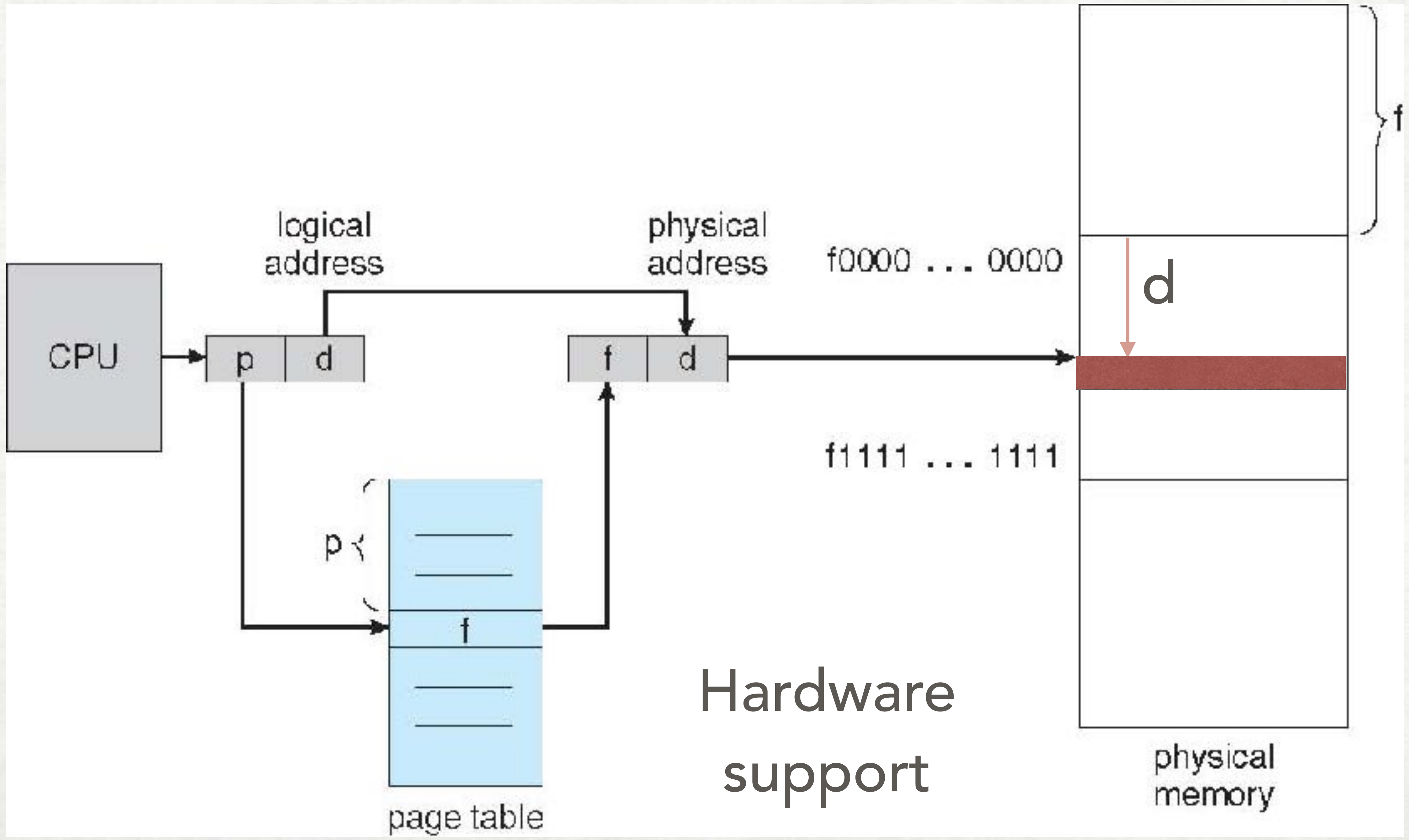



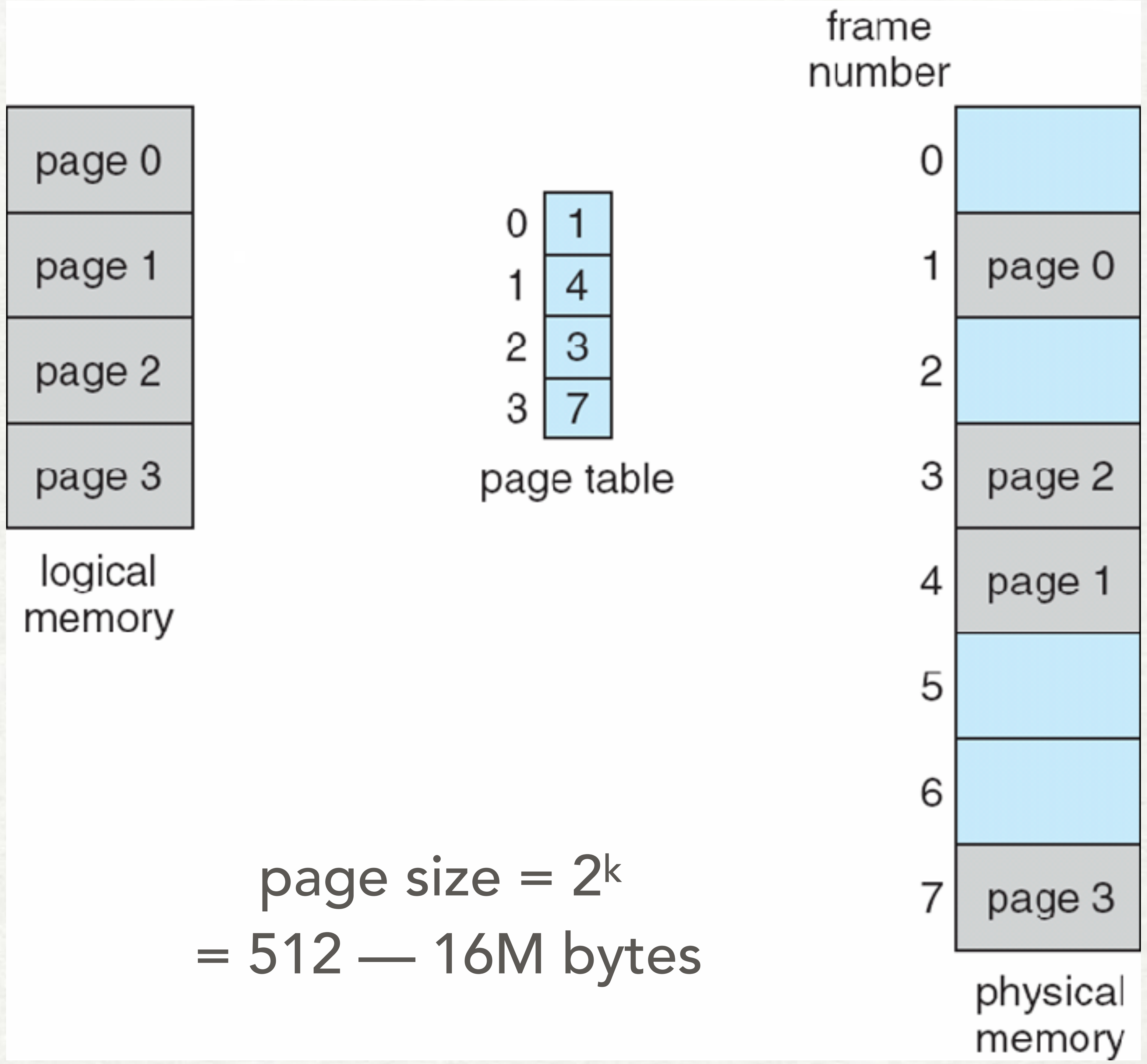

Fragmentation issues?

# PAGING

## STRATEGIES

- splits the memory in equal size pages
- frames (physical) host pages (logical)
- page table/process — for translation



# IMPLEMENTATION OF PAGE TABLES

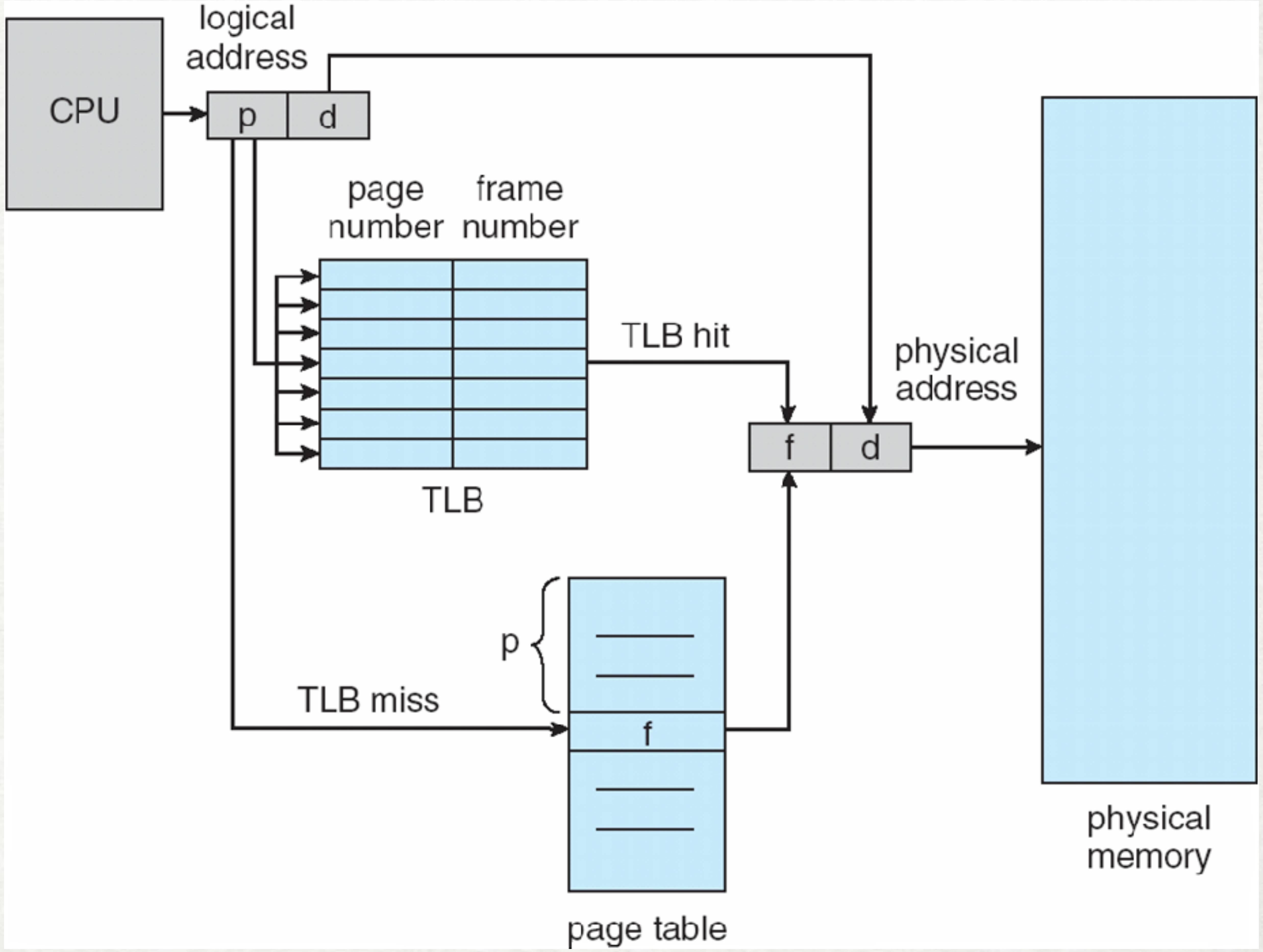
## PAGING

- in memory table
  + base register (PTBR)
  + length register (PTLR)

- **issue**:
  one extra memory access
  (page#-to-frame# translation)

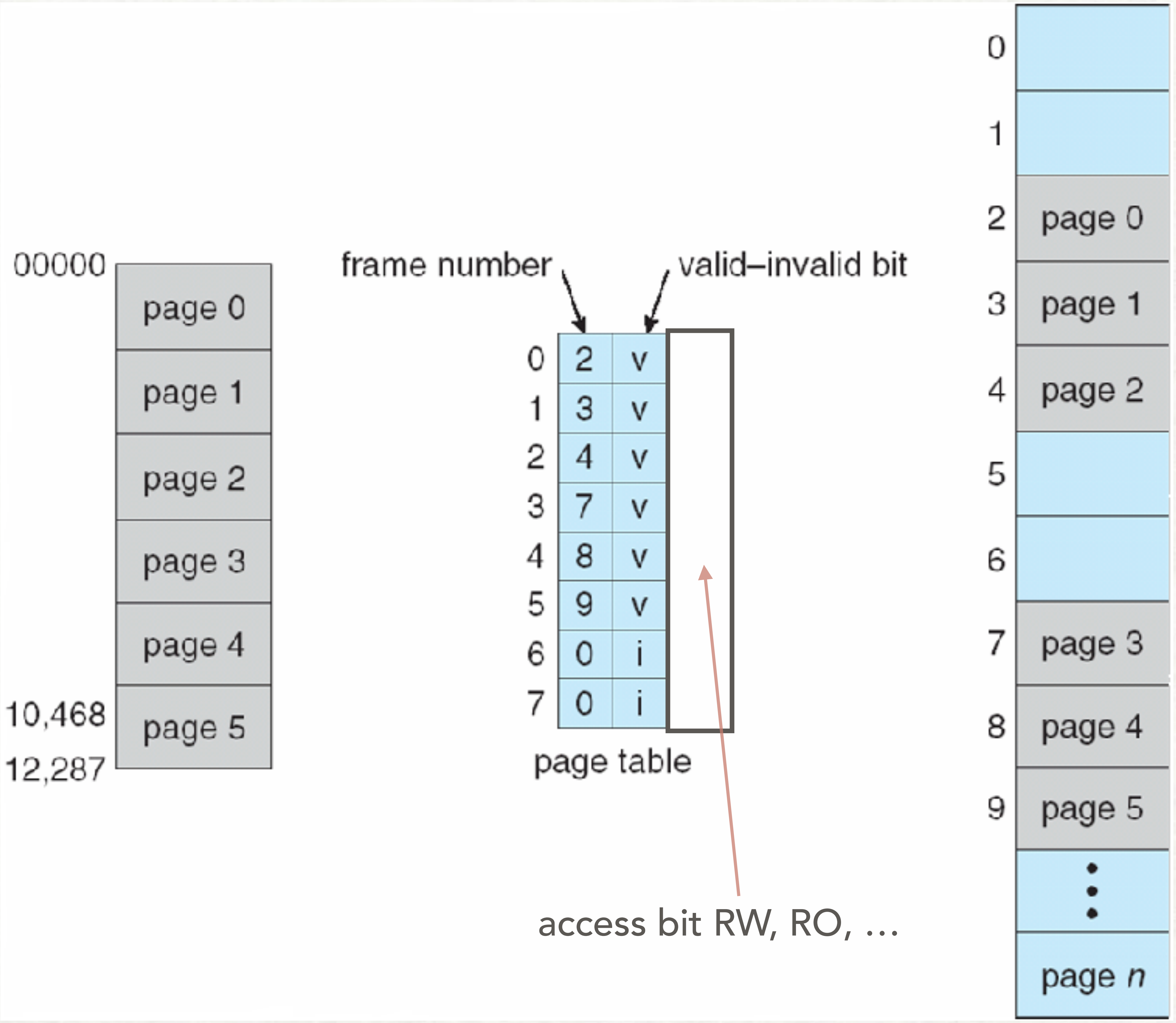
- **solution**: cache?
  translation look-aside buffer (TLB)

- Effective Access Time (textbook)

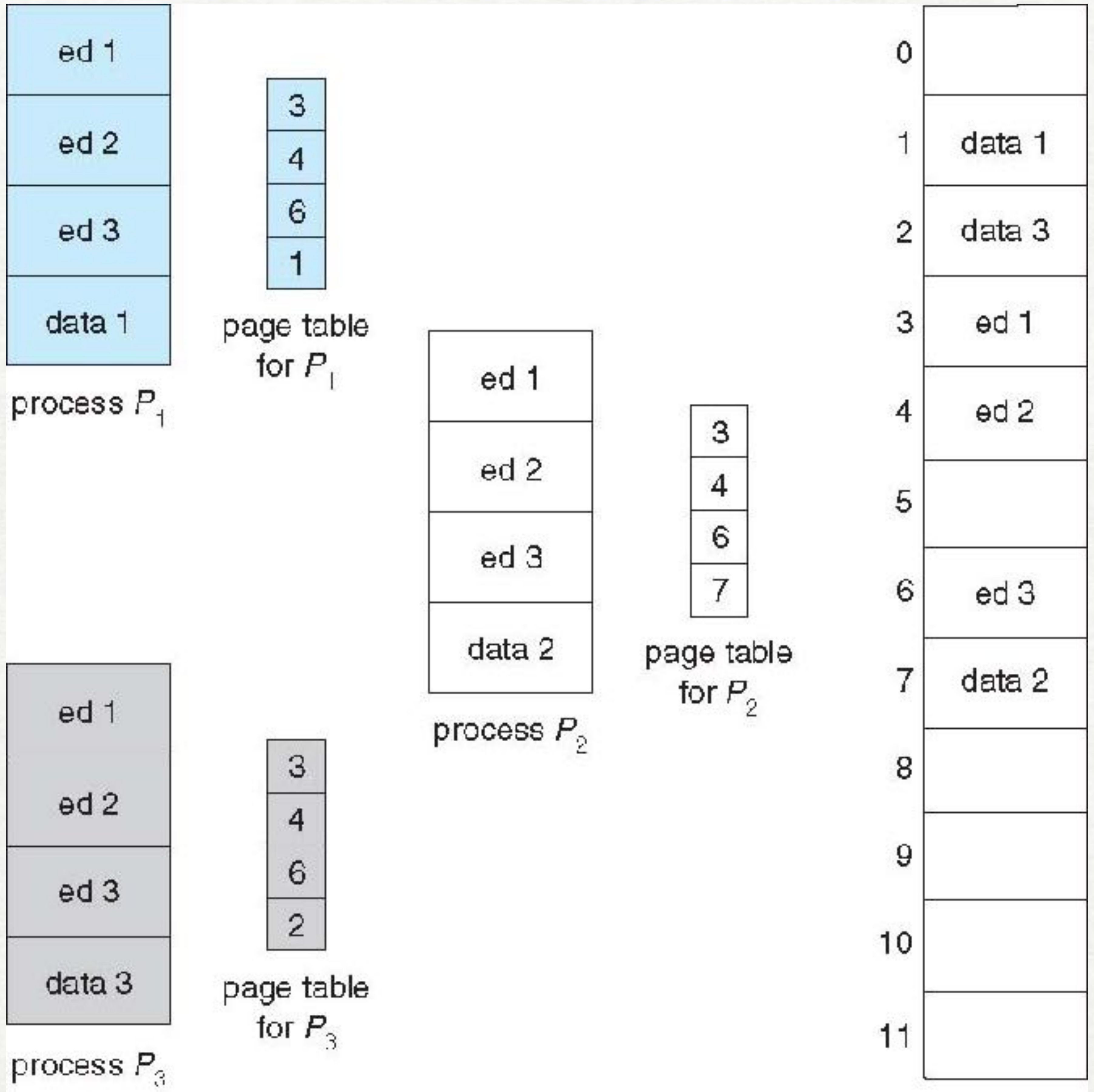


# EXTENSIONS

## PAGING




Memory protection (access bits, valid bit,…)




Sharing pages (code shared, data private)

# PAGE TABLE STRUCTURES

## PAGING

- simple arrays for PT can get huge!

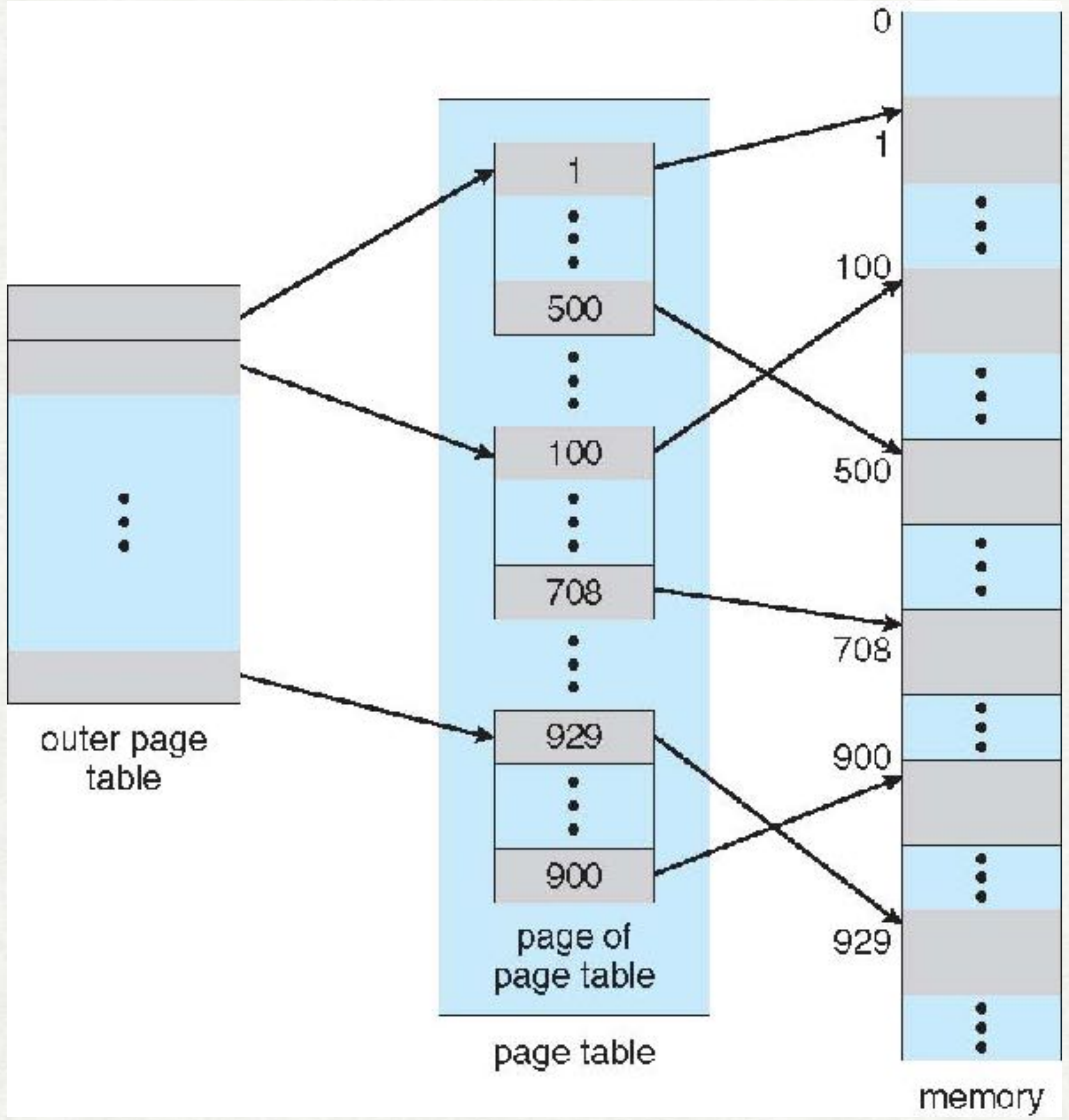
**EXAMPLE**: 32 BIT ADDRESSES, PAGE SIZE = 4KB ($2^{12}$)
– PAGE TABLE =1 MILLION ENTRIES ($2^{20}$), 4MB –
PT COVERS 256 **CONTIGUOUS** PAGES!

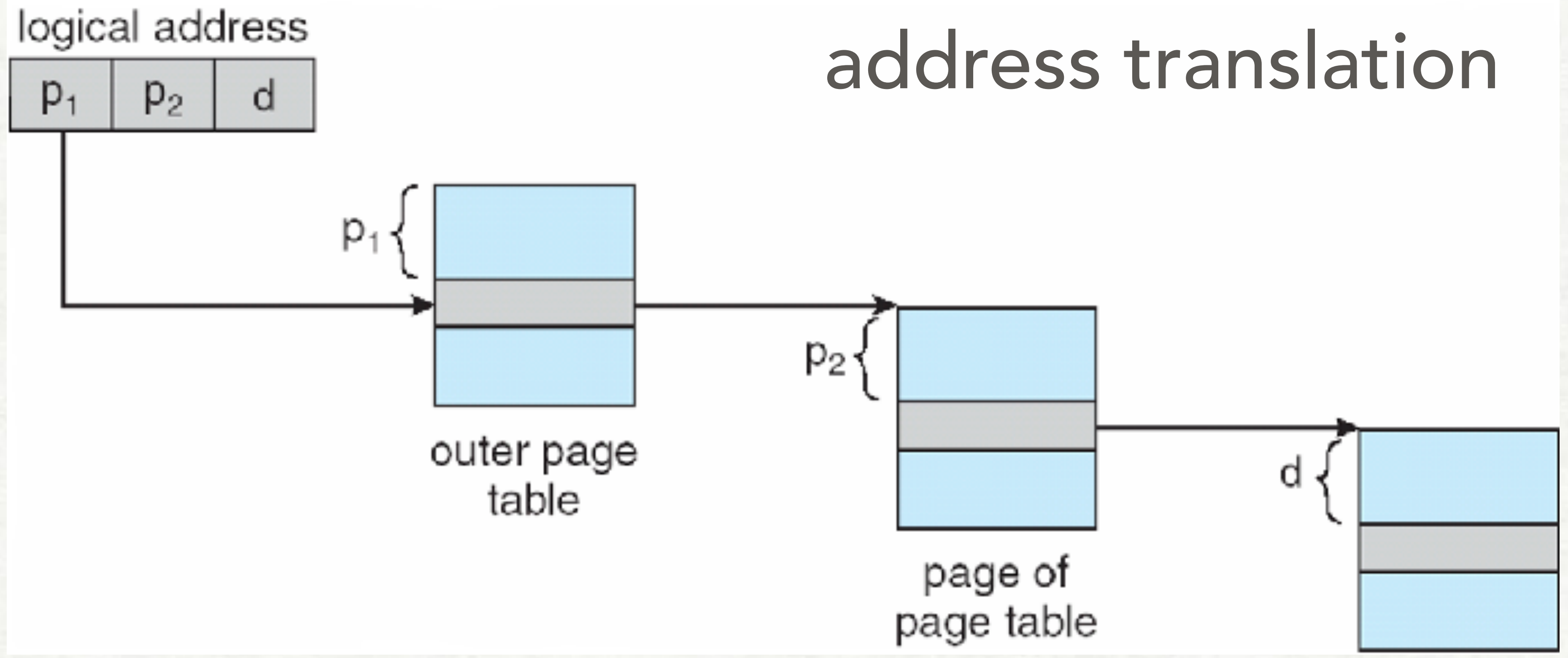
- Better structures needed:
  - Hierarchical page tables
  - Hashed page tables
  - Inverted page tables

# HIERARCHICAL PAGE TABLES

## PAGING

Two-level Page Table

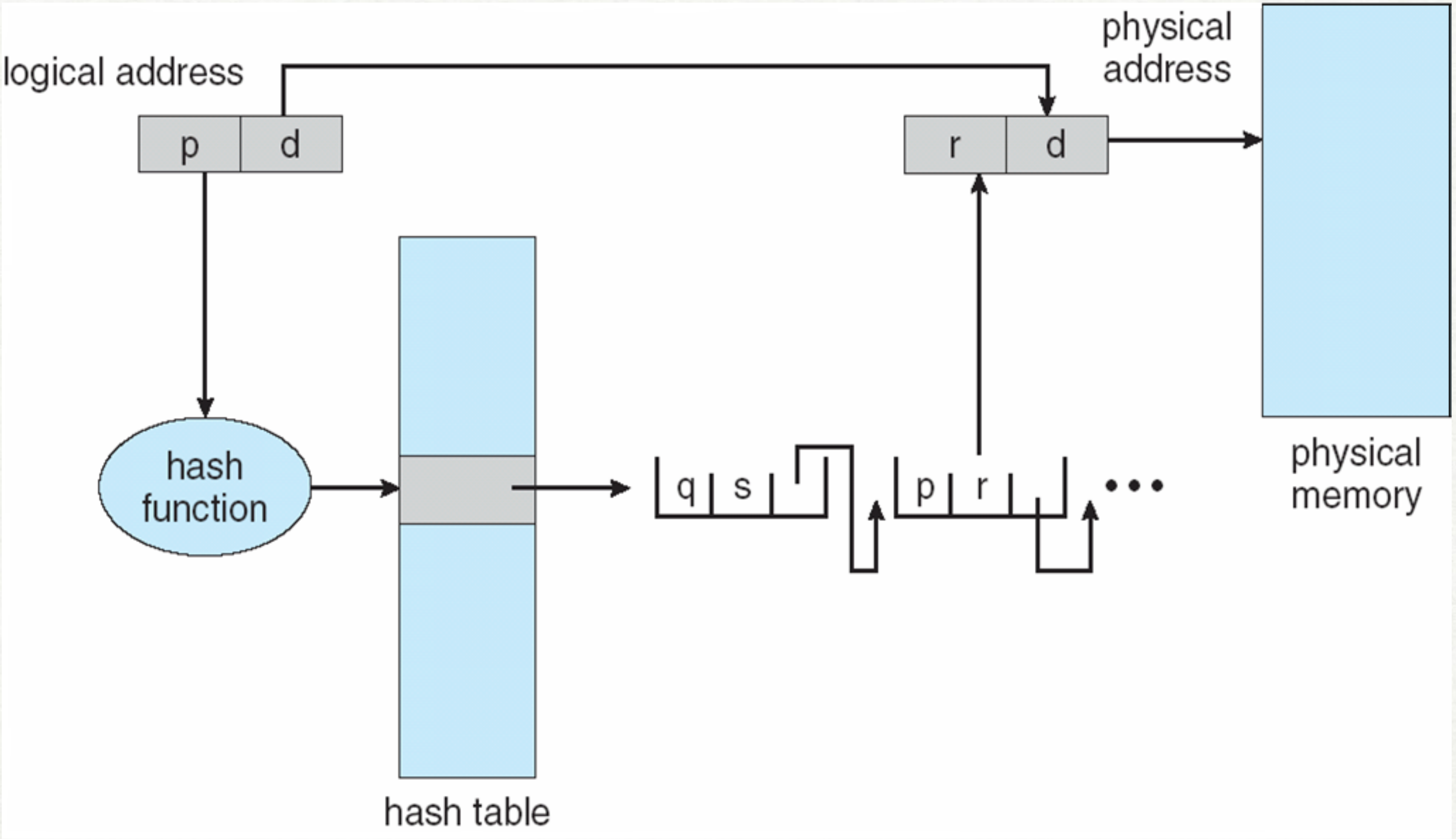



- sparse — occupies only used pages
- increases access time with each extra level
- still huge for 64-bits addresses

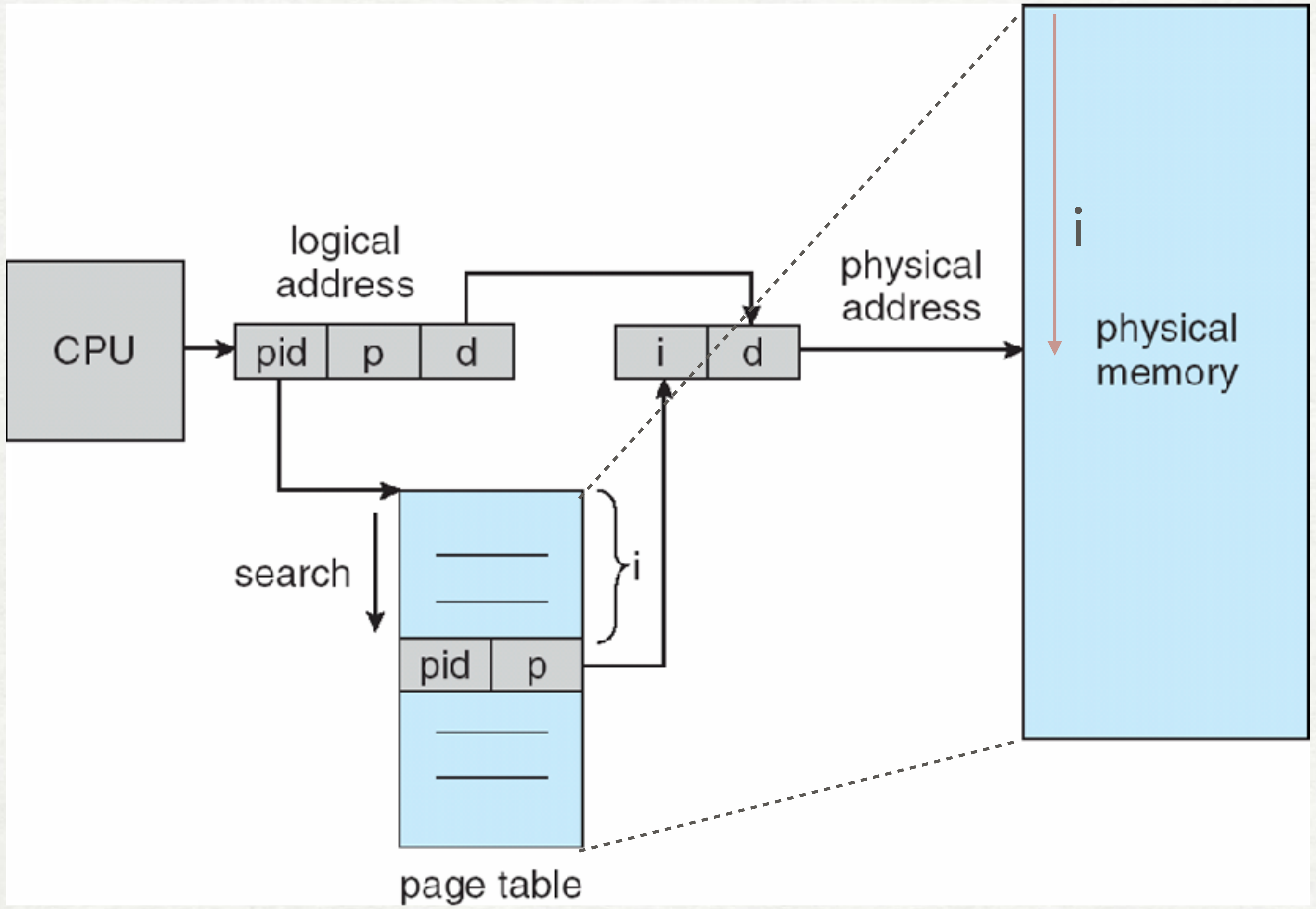
# HASHED PAGE TABLE

## PAGING



# INVERTED PAGE TABLE

## PAGING



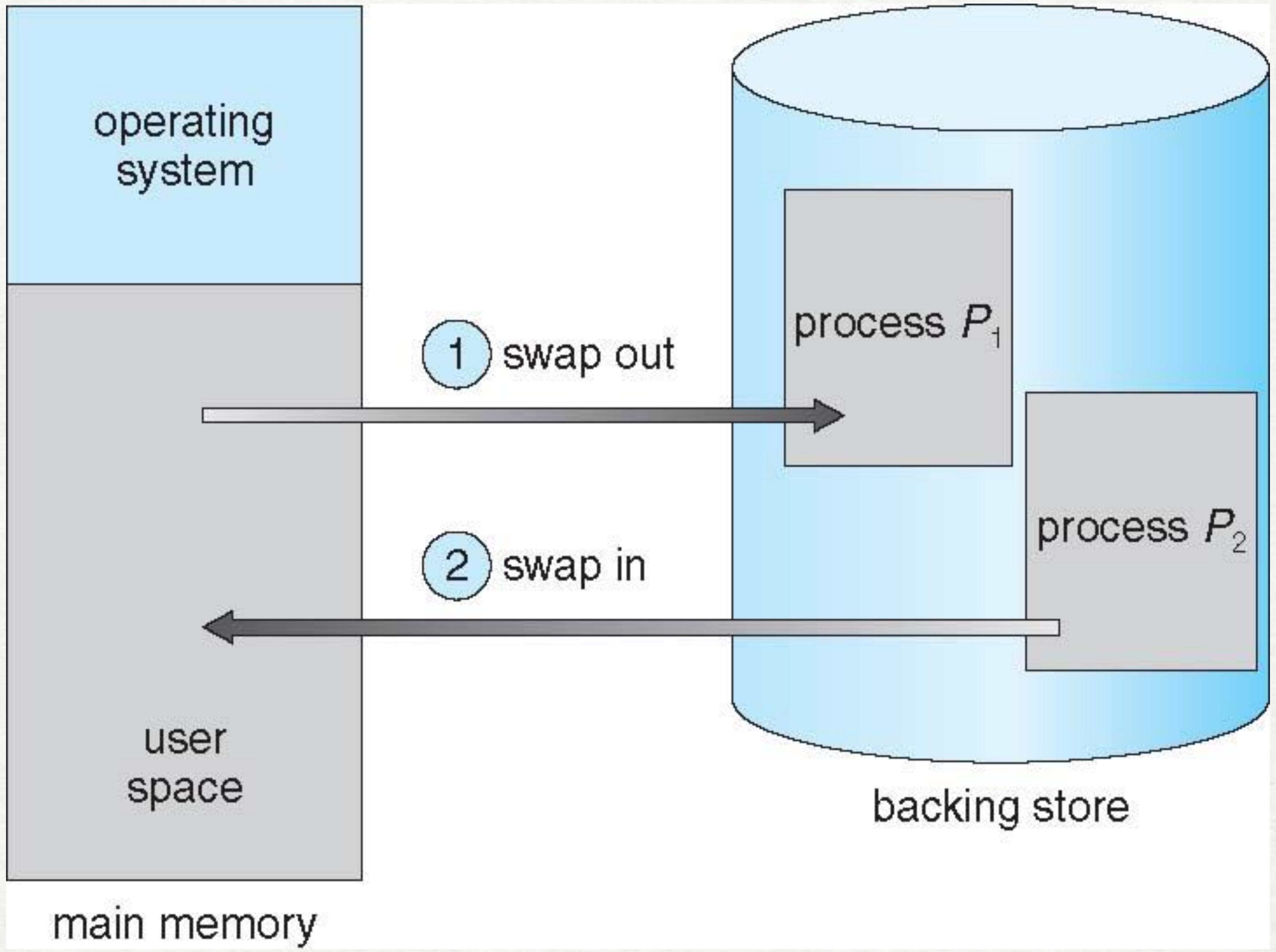

- common global structure (not per process)
- maps a frame# to a process-page# (inverted!)
- limited by the total number of frames: uses less memory
  - **issues:**
    - performance? (hash-table)
    - shared memory? (see book)

# VIRTUAL MEMORY

# SWAPPING

## FOR EXTRA-MEMORY

- save/restore process memory in backing store
- Pros:
  - increase level of multiprogramming
- Cons:
  - large overhead for full process swap
  - not always possible due to pending I/O operations



# BASIC IDEAS

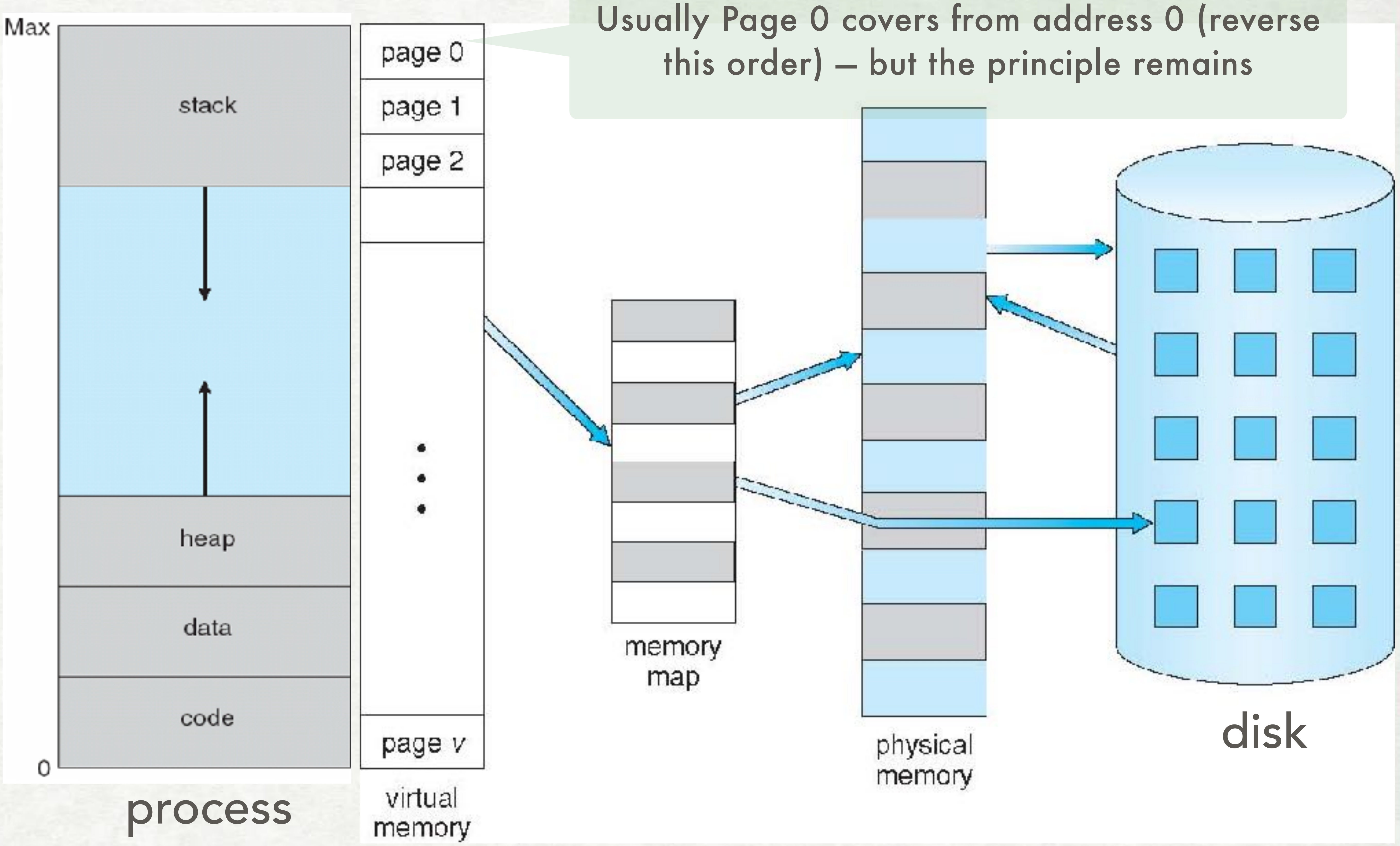
## VIRTUAL MEMORY

- keep in memory only needed code/data, not the whole process (the rest is on the disk)

- decouple logical from physical address spaces

- processes see a larger (virtual) memory than the existing (physical) one

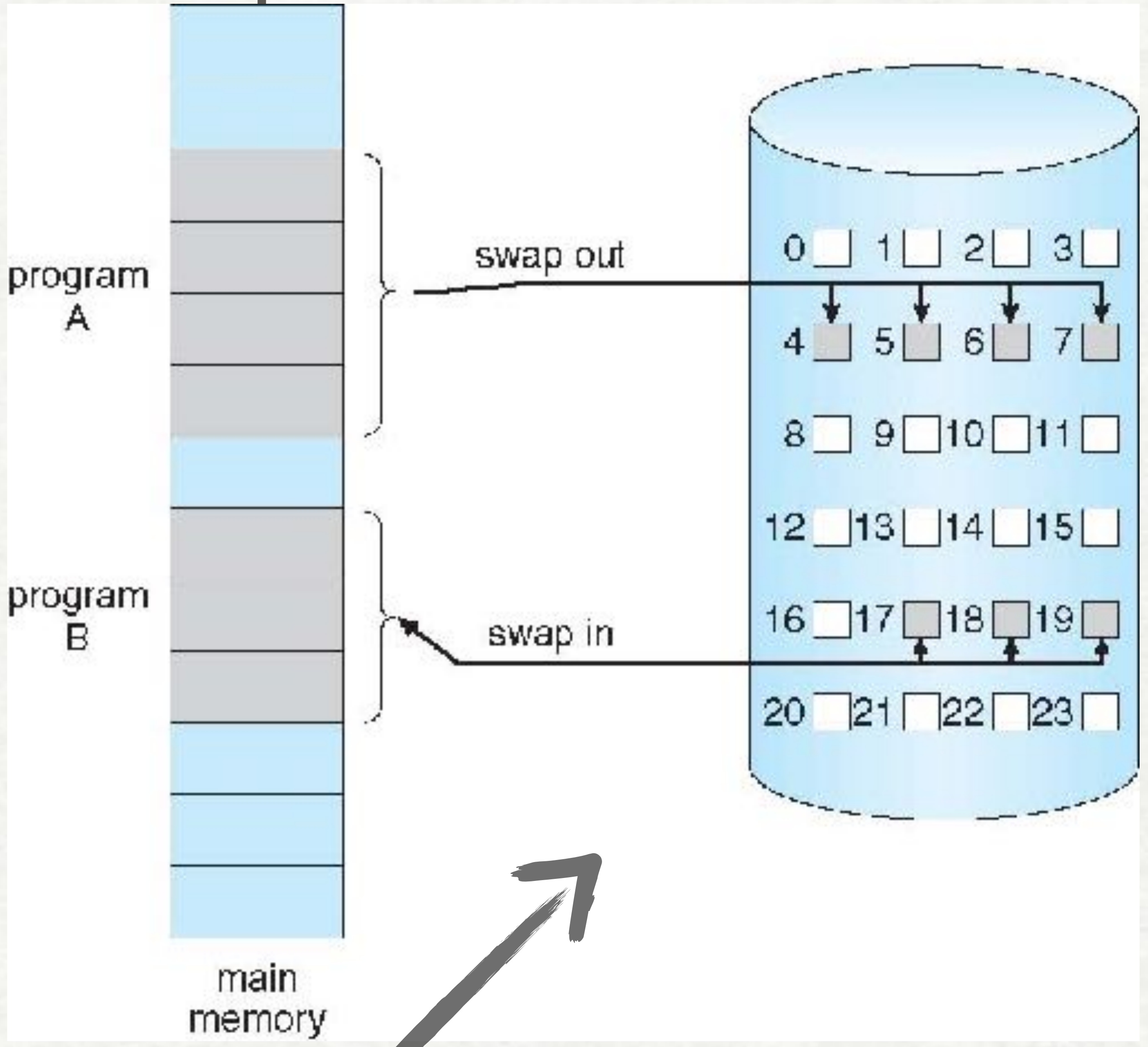
Usually Page 0 covers from address 0 (reverse this order) – but the principle remains

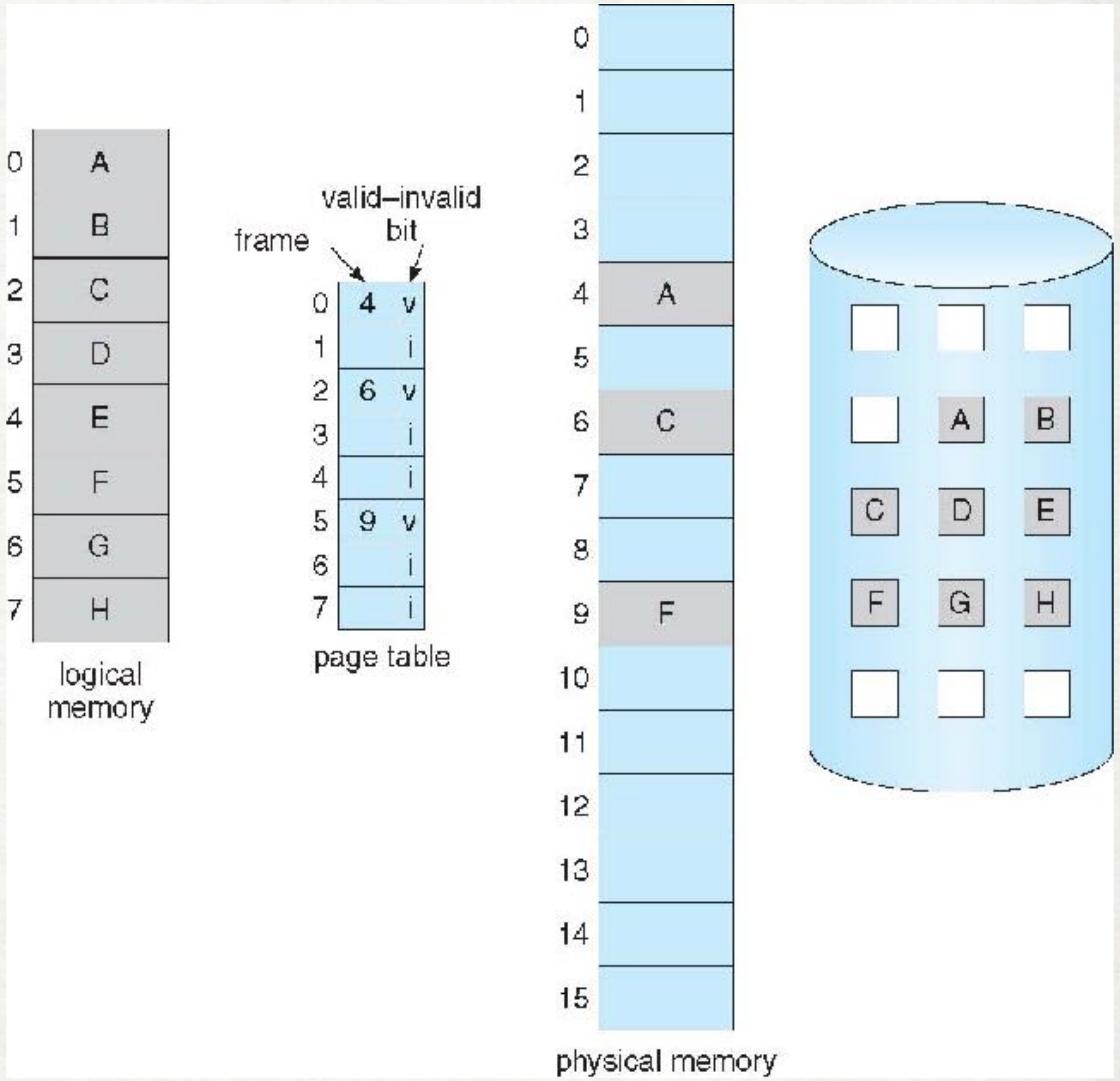


# DEMAND PAGING

- Bring in process pages only when needed (on demand)

- Advantages vs. whole process swap:
  - ✓ faster I/O (one page only)
  - ✓ faster response time
  - ✓ less memory used
  - ✓ more processes supported



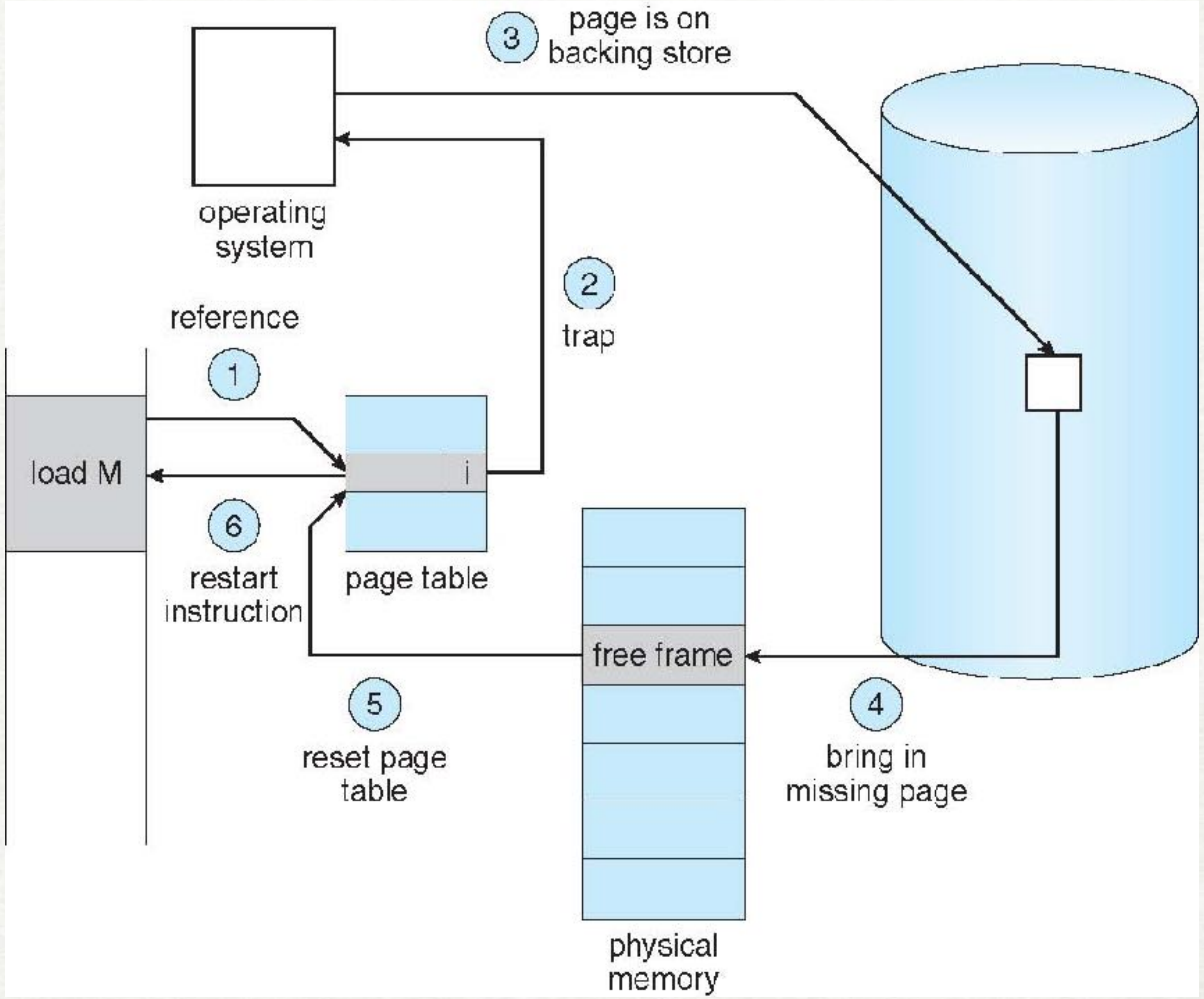

Some pages may remain on the disk!

**Pager**: Like this process **Swapper**… but lazy!

# HANDLING PAGE FAULTS

## DEMAND PAGING

- **page fault** = accessing an invalid address (*va* not present in a *pa*)
- traps into OS
- may bring several pages (for complex instructions)
- Effective Access Time (EAT) = $m * (1 - p) + d * p$ memory (m), disk (d), miss ratio (p)
- all worth it only if *p* is very small!



# COPY-ON-WRITE

## PERKS OF PAGING

PARENT AND CHILD PROCESSES CAN SHARE PAGES UNTIL MODIFIED!

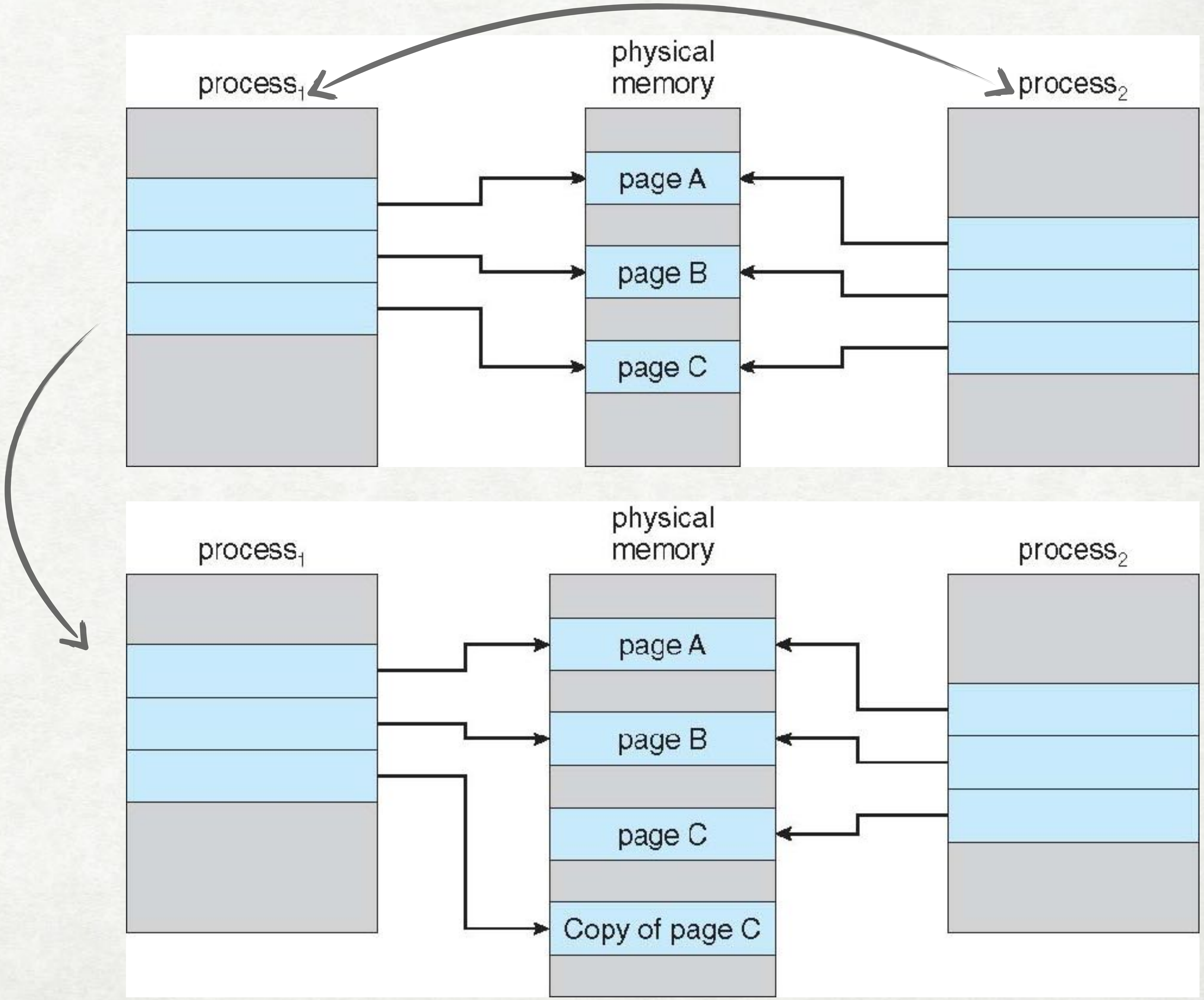
Advantages:

- fast fork (response time)
- less memory used

```
> man fork, vfork, exec
```

say one forked the other

process 1 writes to page C



# FRAME ALLOCATION AND PAGE REPLACEMENT

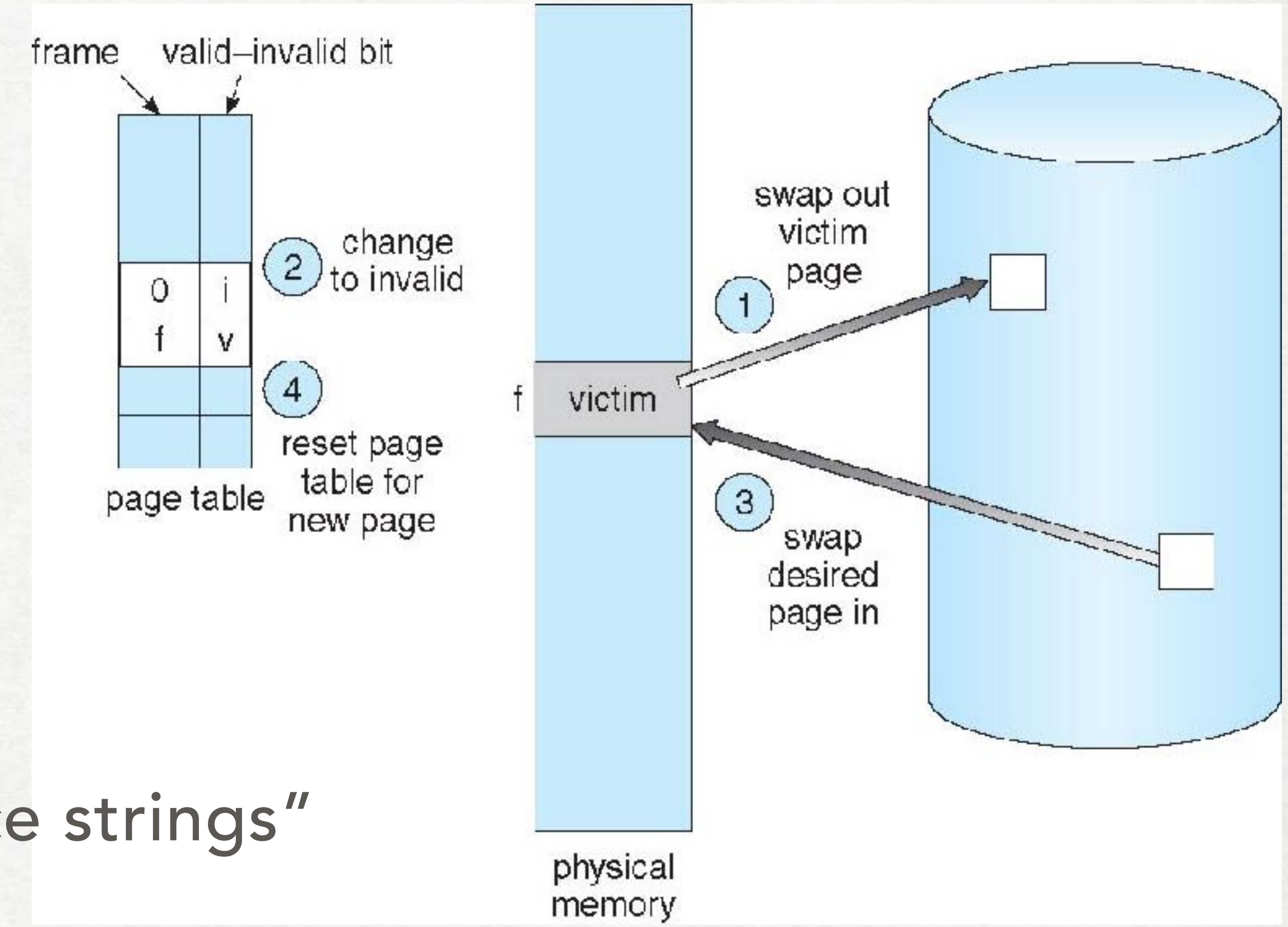
## PAGING

- **frame allocation:**
  how many frames to give to each process?

- **page replacement:**
  make space for a new page =
  swap out/discard the old one, if used

- which page to replace?
  "**goal**: minimize page-faults"

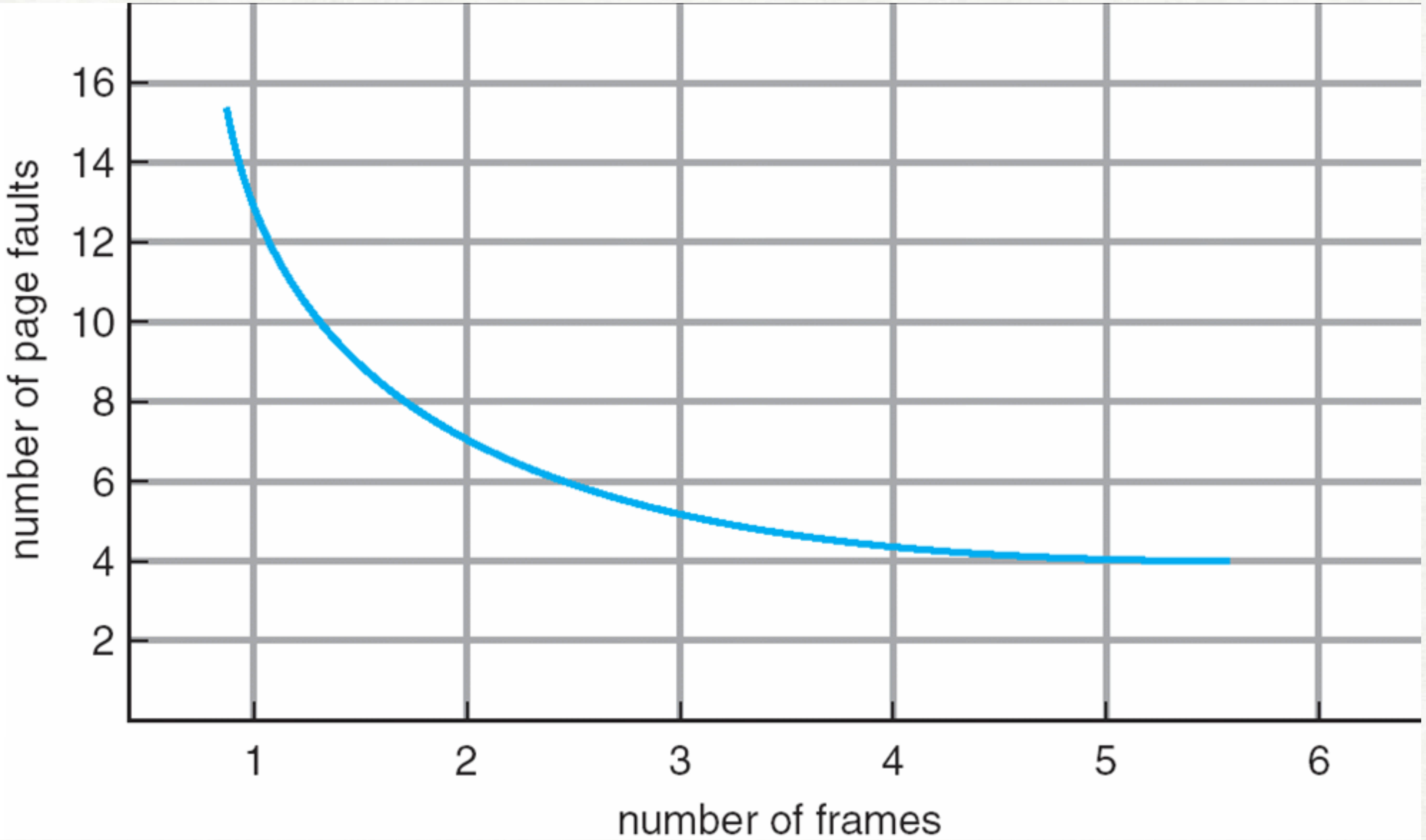
- various algorithms - evaluate them on "reference strings"
  = sequences of addresses (page numbers)




Example of a "reference string": 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

# PAGE FAULTS VS. ALLOCATED FRAMES

## PAGING




Expected shape…

# FIRST-IN FIRST-OUT (FIFO) ALGORITHM

## PAGE REPLACEMENT

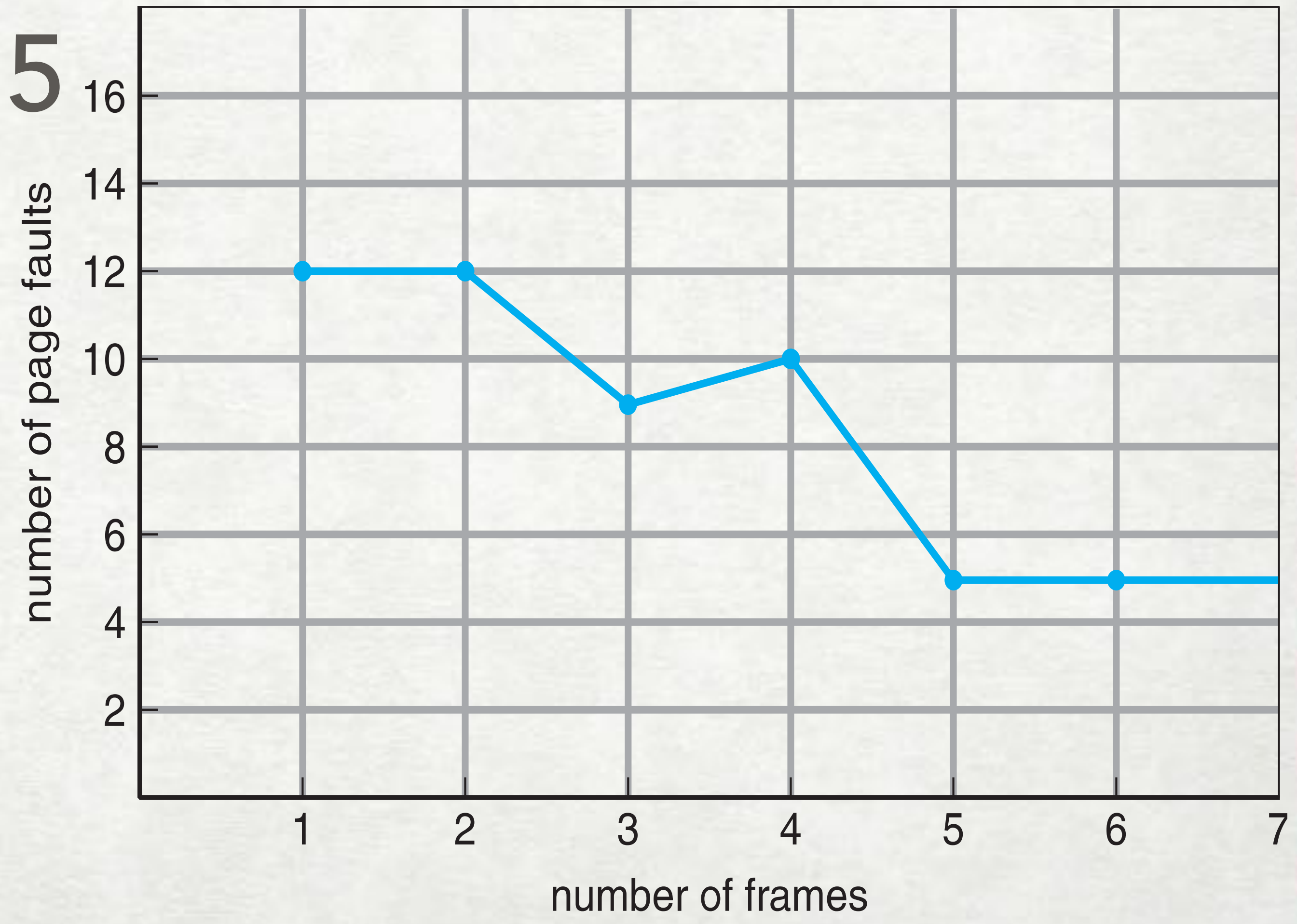
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory per process at a time)

reference string

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

FIFO replacement:

- Result can vary with the reference string: 1,2,3,4,1,2,5,1,2,3,4,5



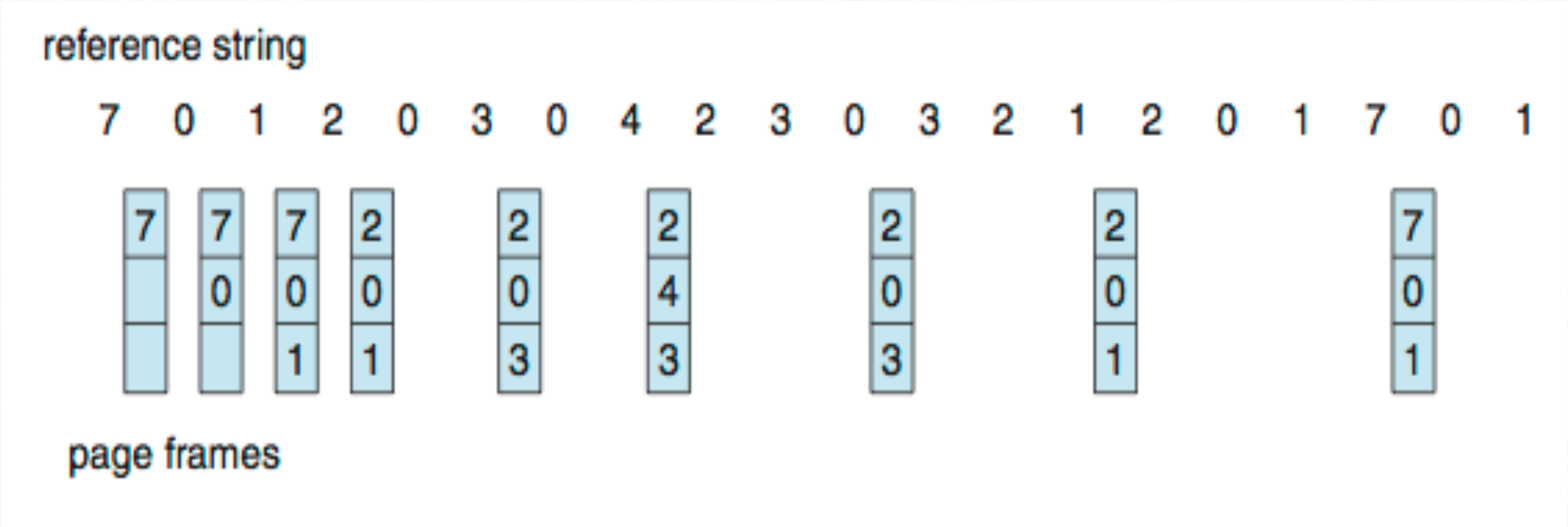

✶Adding frames causes more page faults! <u>**Belady's Anomaly**</u>

- How to track ages of pages? (use a FIFO queue)

# OPTIMAL (OPT) ALGORITHM

## PAGE REPLACEMENT

- *"replace the page that will not be used for the longest time in the future"*
- needs knowledge of the future - **not feasible in practice**
- used as a baseline (to compare to other algorithms)
- practical version: use estimates to predict the future

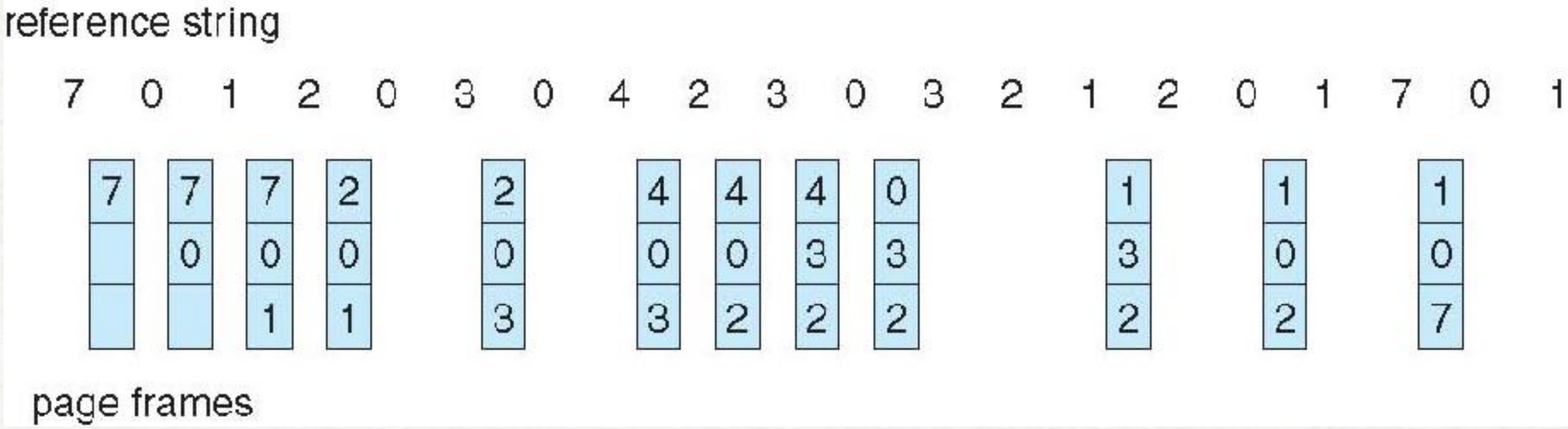



9 page faults

# LEAST RECENTLY USED (LRU) ALGORITHM

## PAGE REPLACEMENT

- estimate the future: history
- *"replace the page not accessed for the longest time in the past"*




OPT < 12 faults < FIFO

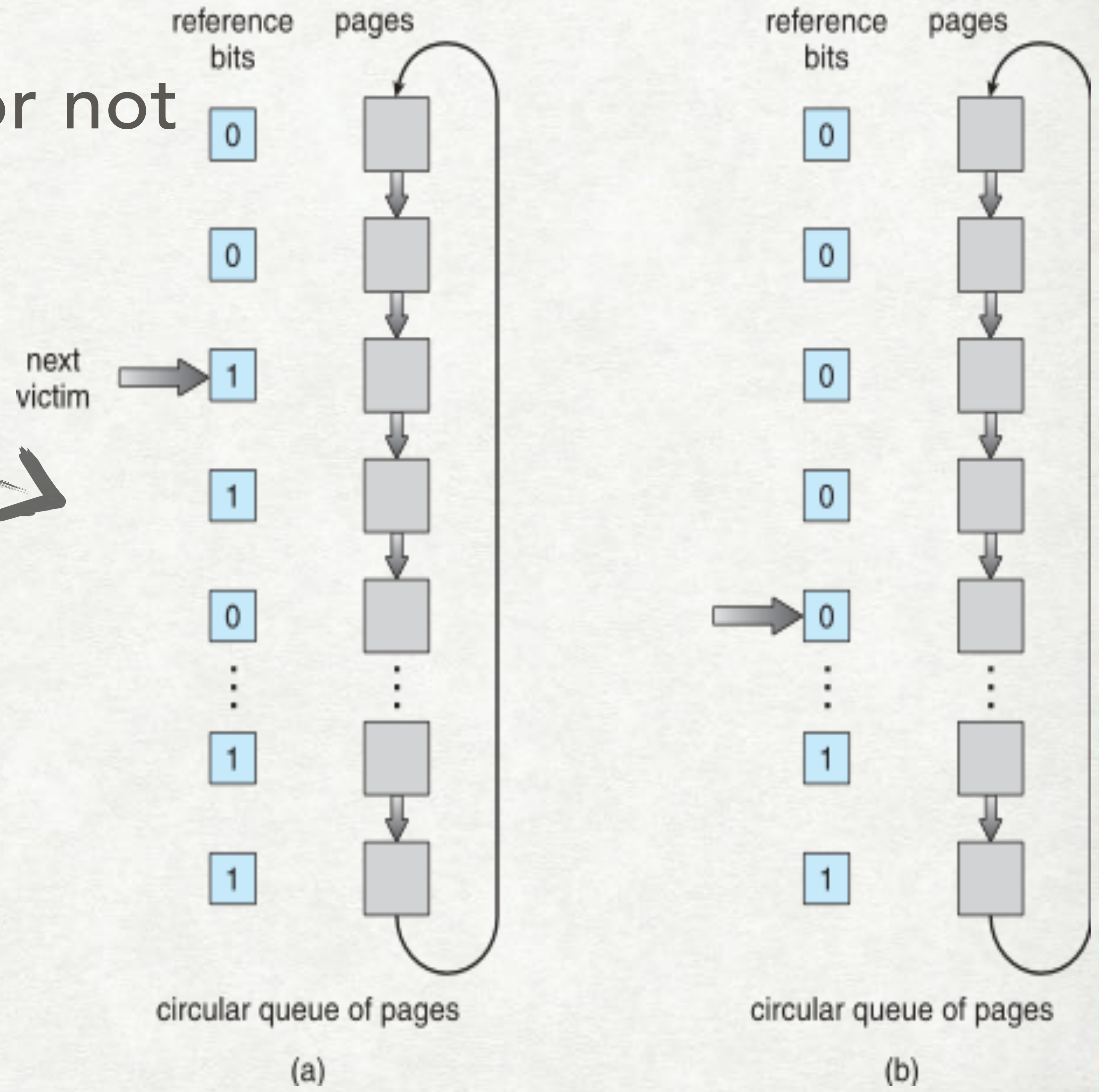
- generally good performance
- implementation… how?

**LRU** AND **OPT** ARE TWO SO CALLED "STACK ALGORITHMS" - DO NOT SUFFER FROM BELADY'S ANOMALY

# MORE ALGORITHMS

## PAGE REPLACEMENT

- LRU approximations:
  - **reference bit** (LRU count is 1-bit): 1 for referenced, 0 for not
  - **second chance**: FIFO plus reference bit
  - …
- other counting algorithms: count accesses
  - **Least Frequently Used** (LFU)
  - **Most Frequently Used** (MFU)



# ALLOCATING FRAMES
## PAGING

- each processes:

  needs a min number of frames
  (max is the total number of frames)

“SS MOVE” INSTRUCTION ON IBM370: 6 PAGES

(6 BYTES) CAN SPAN OVER 2 PAGES
2 PAGES FOR “FROM”
2 PAGES FOR “TO”

- how to distribute between processes?
  - fixed vs. priority
- relation to page replacement?
  - global (all frames) vs. local (own frames)

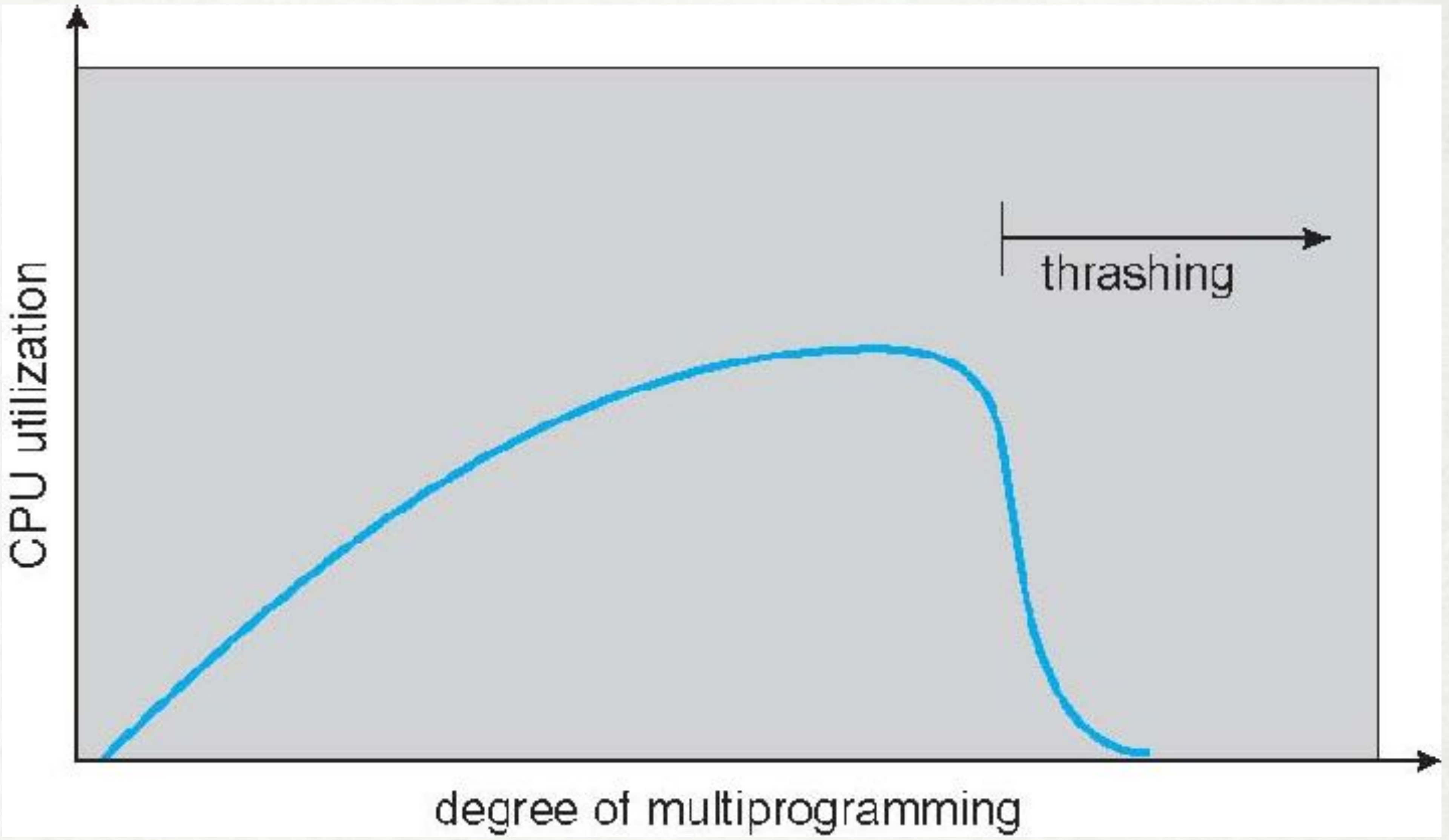
# THRASHING

## PAGE REPLACEMENT

- "busy only swapping pages in and out"

1. needs a page — page fault
2. replaces a page — immediately needs it back
3. mainly waits for I/O — lower CPU utilization
4. OS brings in more processes (increases the degree of multiprogramming)

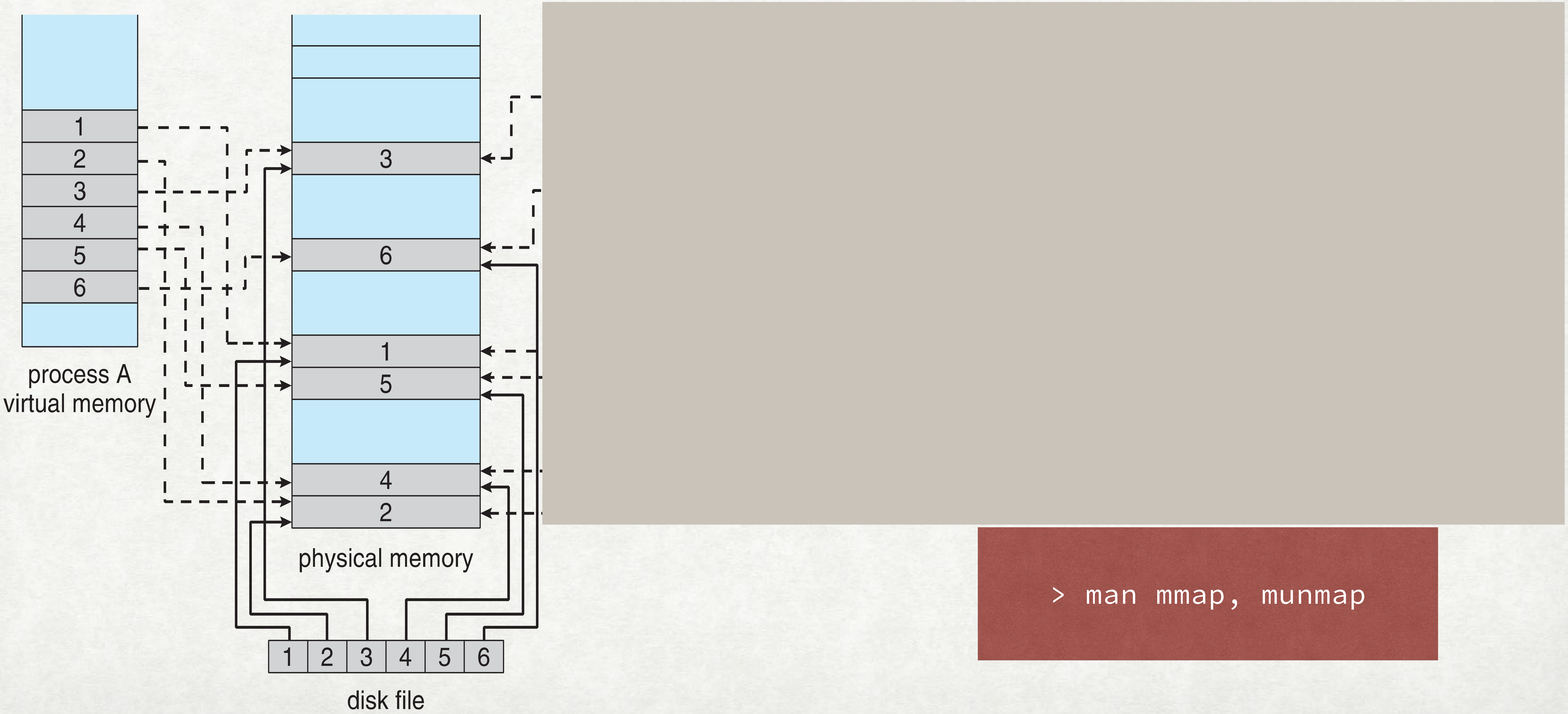
Evil Circle




SEE BOOK FOR MITIGATION STRATEGIES

# MEMORY MAPPED FILES

## PAGING

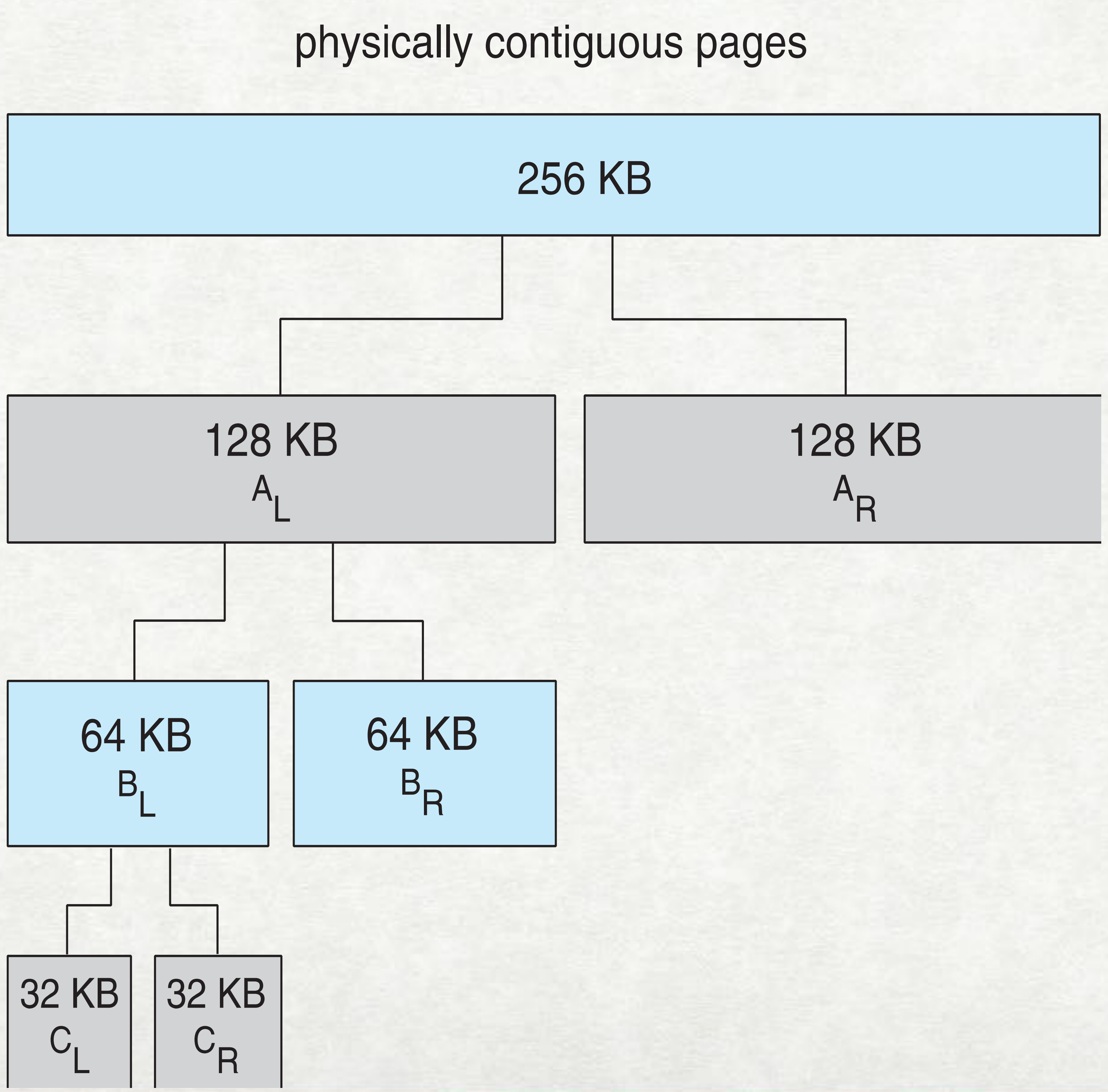



> man mmap, munmap

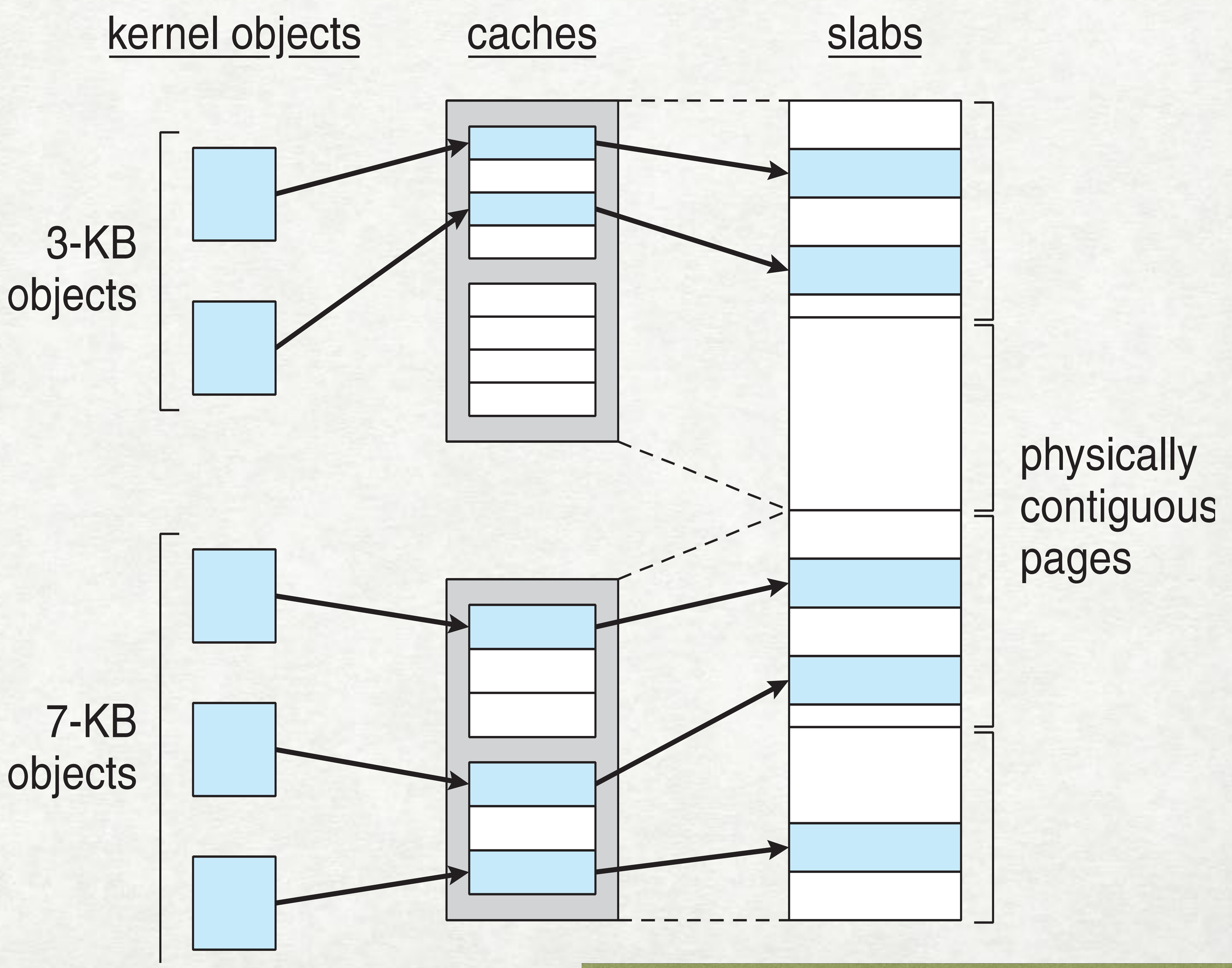
# ALLOCATING KERNEL MEMORY

- treated differently from user memory!

**BUDDY SYSTEM ALLOCATOR**




**SLAB ALLOCATOR**




[LINUX SLAB SLOB SLUB]

# OTHER CONSIDERATIONS

## MEMORY MANAGEMENT

- Pre-paging
- Page size
- TLB Reach
- Program structure
- I/O interlock

128 frames, page size = 128

```
int data[128, 128];
```

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

128x128 = 16,384 page faults

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

128 page faults

# END OF MODULE 6