# Operating Systems – Memory Management

ECE 344

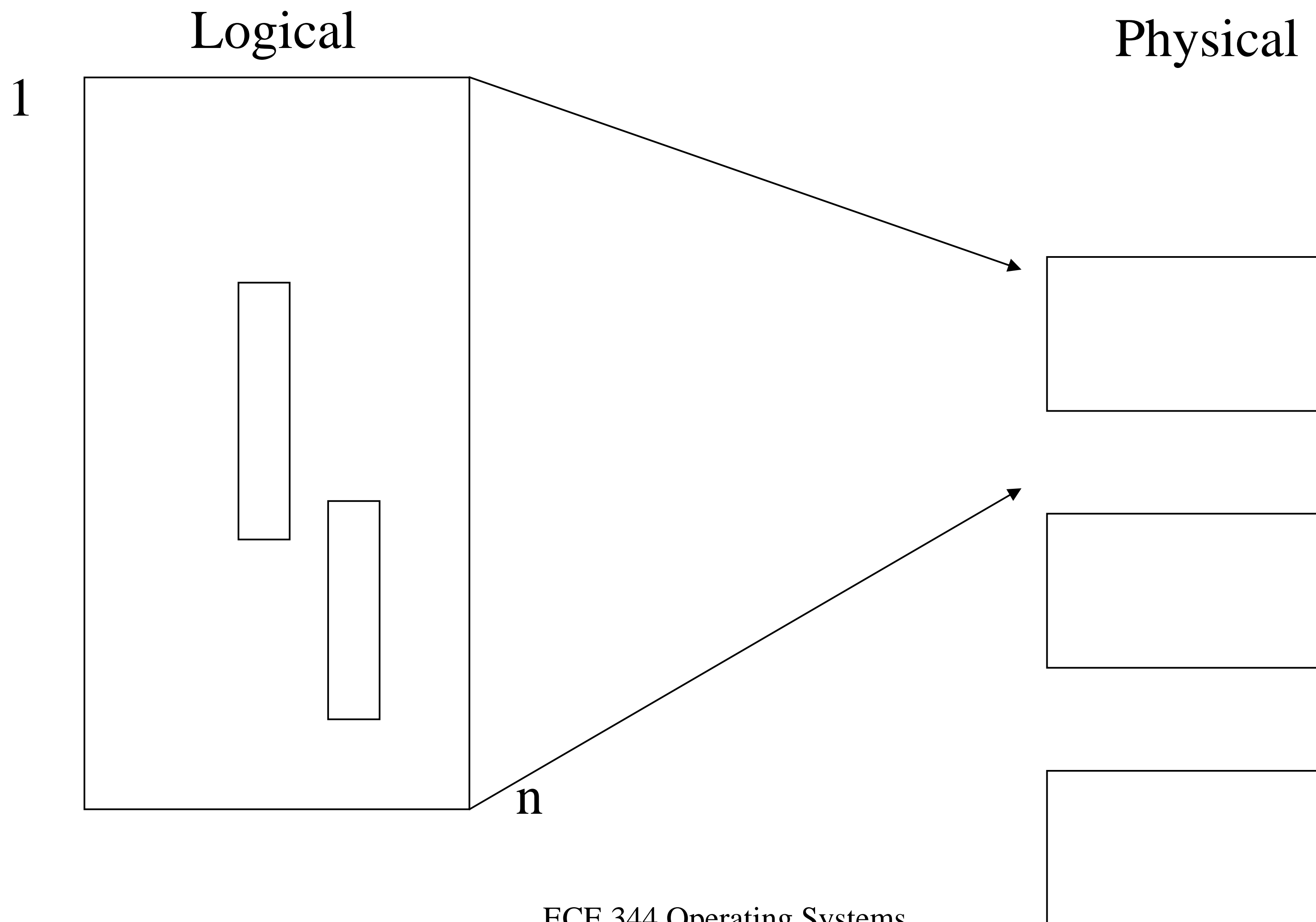
# Memory Management

- Contiguous Memory Allocation
- Paged Memory Management
- Virtual Memory

# Binding of Instructions and Data to Memory

- **Compile time**:
  - known memory location
  - absolute code can be generated
  - must recompile code if starting location changes.
- **Load time**:
  - generate *relocatable* code if memory location is not known at compile time.
- **Execution time**:
  - process can be moved during its execution from one memory segment to another.
  - need hardware support for address mapping

# Logical vs. Physical Address Space



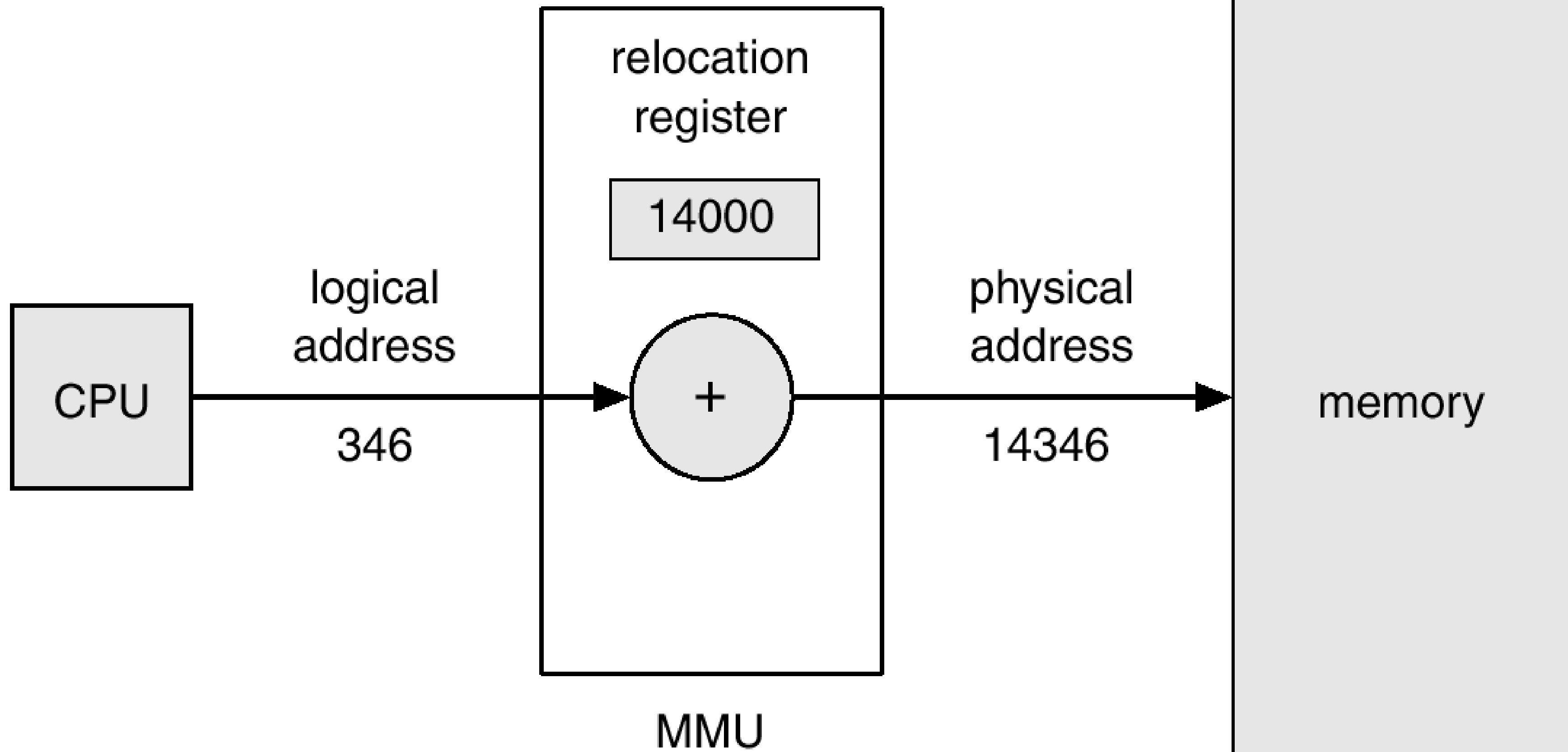
# Logical vs. Physical Address Space

- A **logical *address space*** that is bound to a **separate *physical address space***
  - ***Logical address*** – generated by the CPU; also referred to as ***virtual address*.**
  - ***Physical address*** – address **generated by the memory management unit.**
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.
- Logical (virtual) and physical addresses differ in execution-time address-binding scheme.

# Memory-Management Unit (MMU)

- Hardware device that **maps logical/virtual to physical address.**
- In MMU the value in the **relocation register** is **added to every address generated by a program** at the time the address is sent to memory.
- The **program deals with logical addresses**; it **never sees** the real **physical addresses.**

# Dynamic relocation/binding using a relocation register

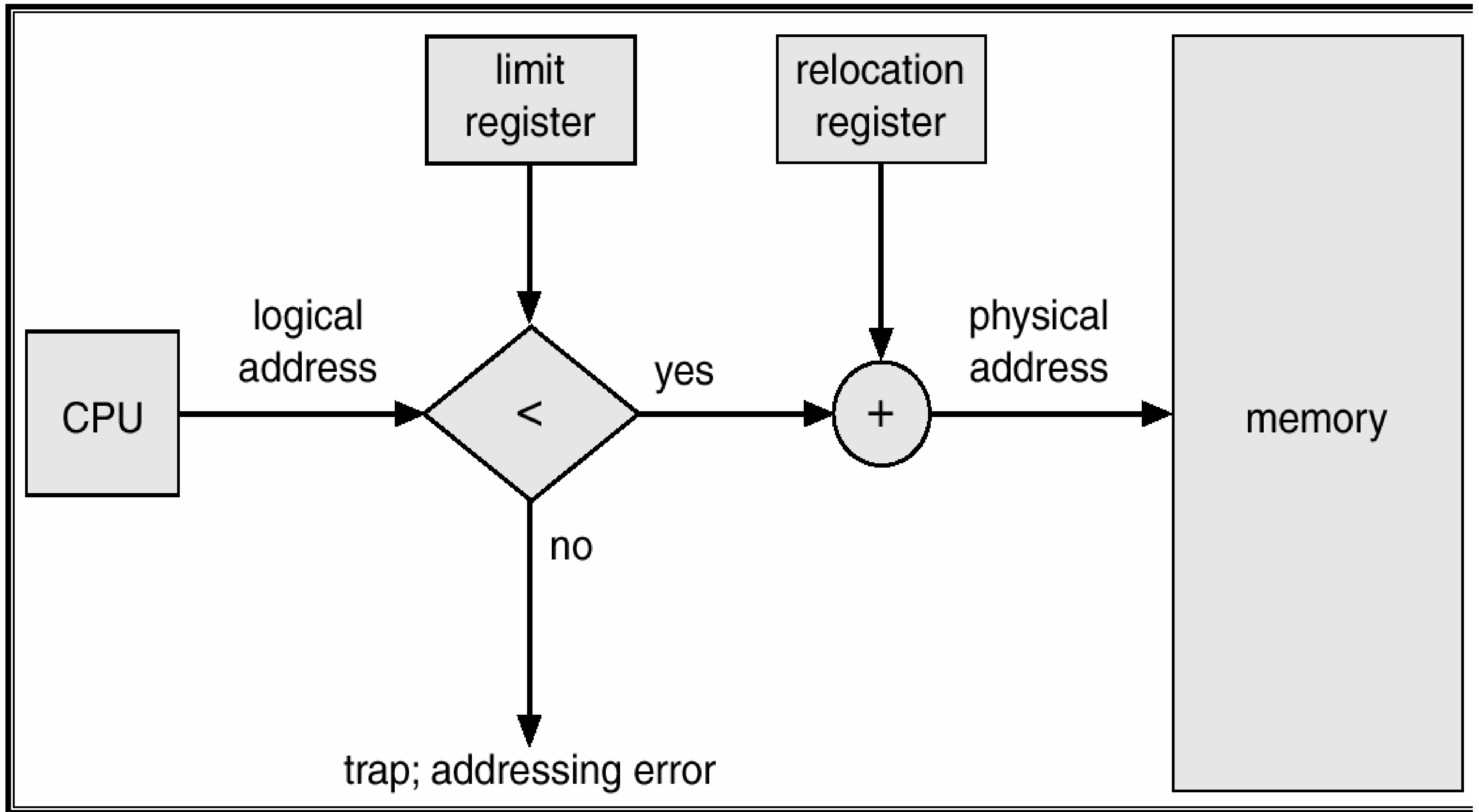


# Memory Allocation

# Contiguous Memory Allocation

- Multiple partitions for multiple processes
- **Relocation register** and **limit registers to protect processes from one another** (and protect OS code)
- Both registers are part of **process context (i.e., PCB)**
- Relocation register contains **value of smallest physical address**
- Limit register contains **range of logical addresses**
- Each logical address must be less than the limit register.

# Hardware Support for Relocation and Limit Registers



# Multi-partition Allocation

- ***Holes*** are blocks of available memory
- Holes of various size are scattered throughout memory.
- When a process arrives, it is allocated memory from a hole large enough to accommodate it.
- Operating system maintains information about:
  – allocated partitions
  – free partitions (i.e., holes)

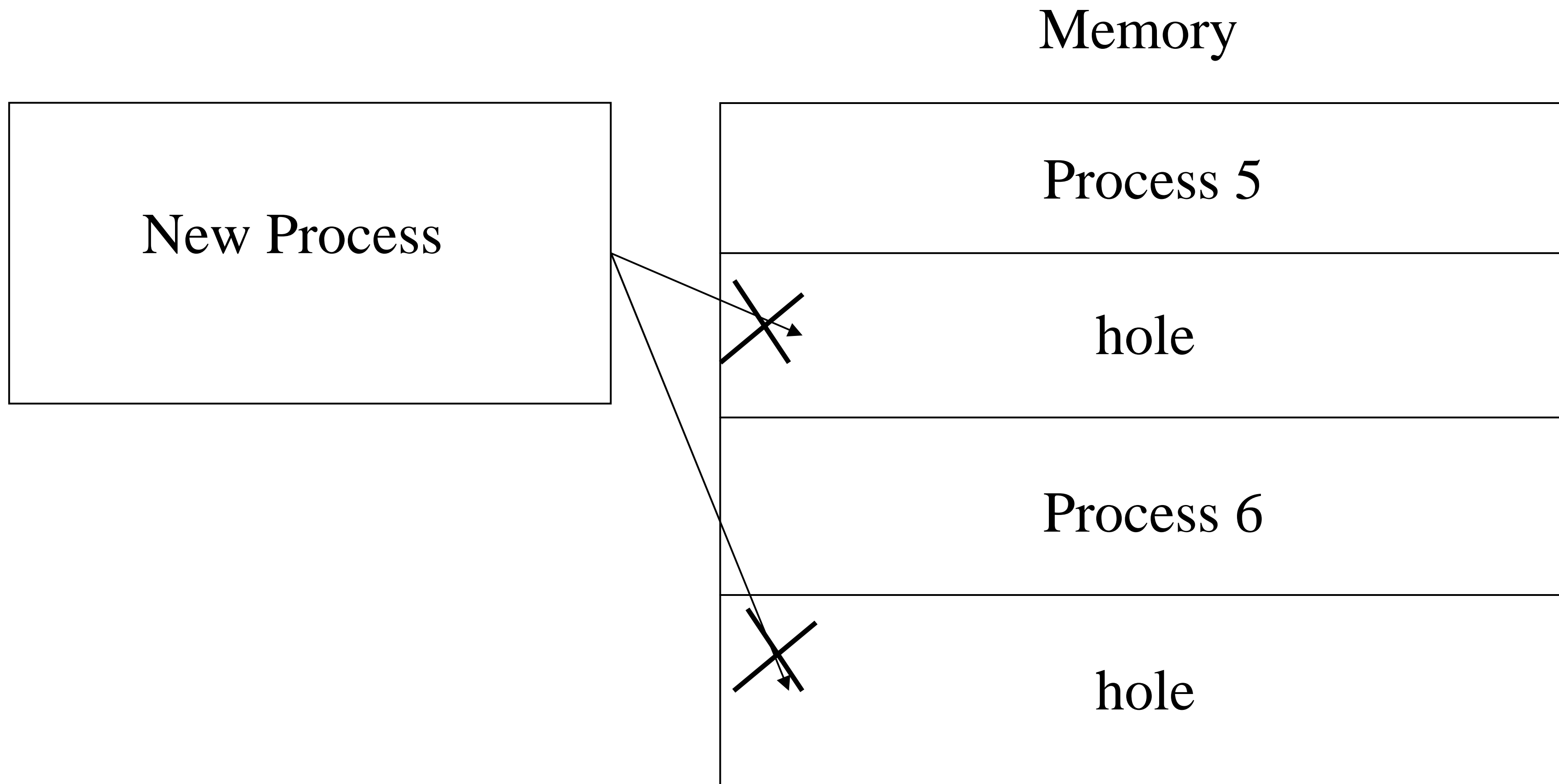
time →

| | | | |
|---|---|---|---|
| P | P | P | P |
| Q | Q | | |
| R | hole | hole | |
| S | S | S | S |
| T | T | T | |

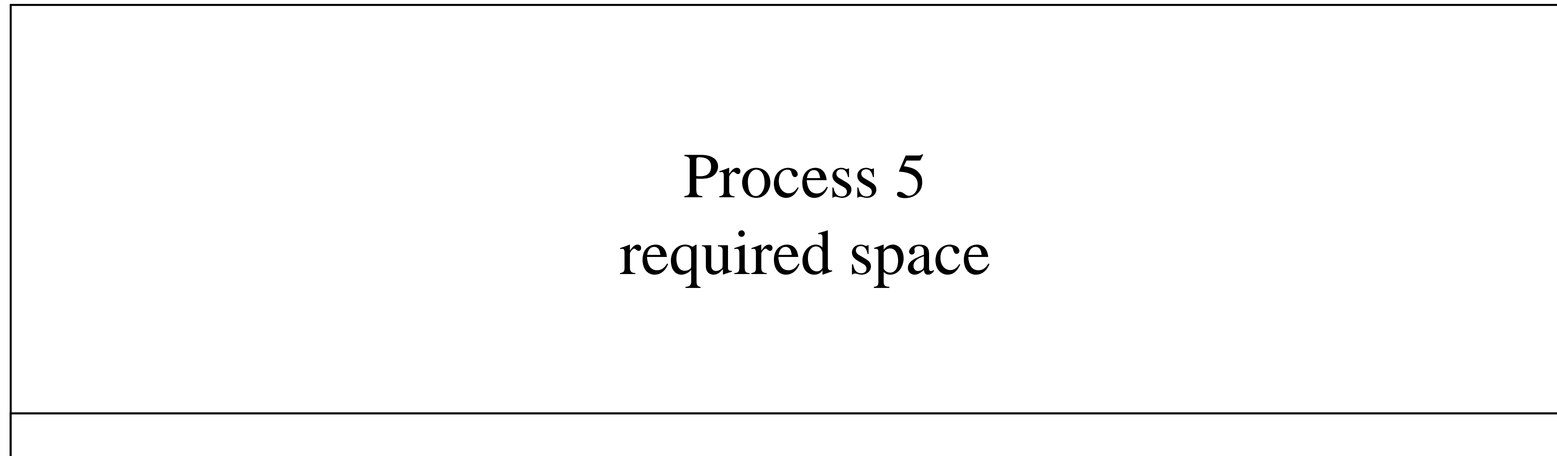
# Dynamic Storage Allocation Problem

- ***How to satisfy a request for memory of size n from a list of free holes*?**
- **First-fit**: Allocate the ***first*** **hole** that is big enough.
- **Best-fit**: Allocate the ***smallest*** **hole** that is big enough; must search entire list, unless ordered by size. **Produces the smallest leftover hole.**
- **Worst-fit**: Allocate the ***largest*** **hole**; must also search entire list. **Produces the largest leftover hole**

# External Fragmentation



# Internal Fragmentation



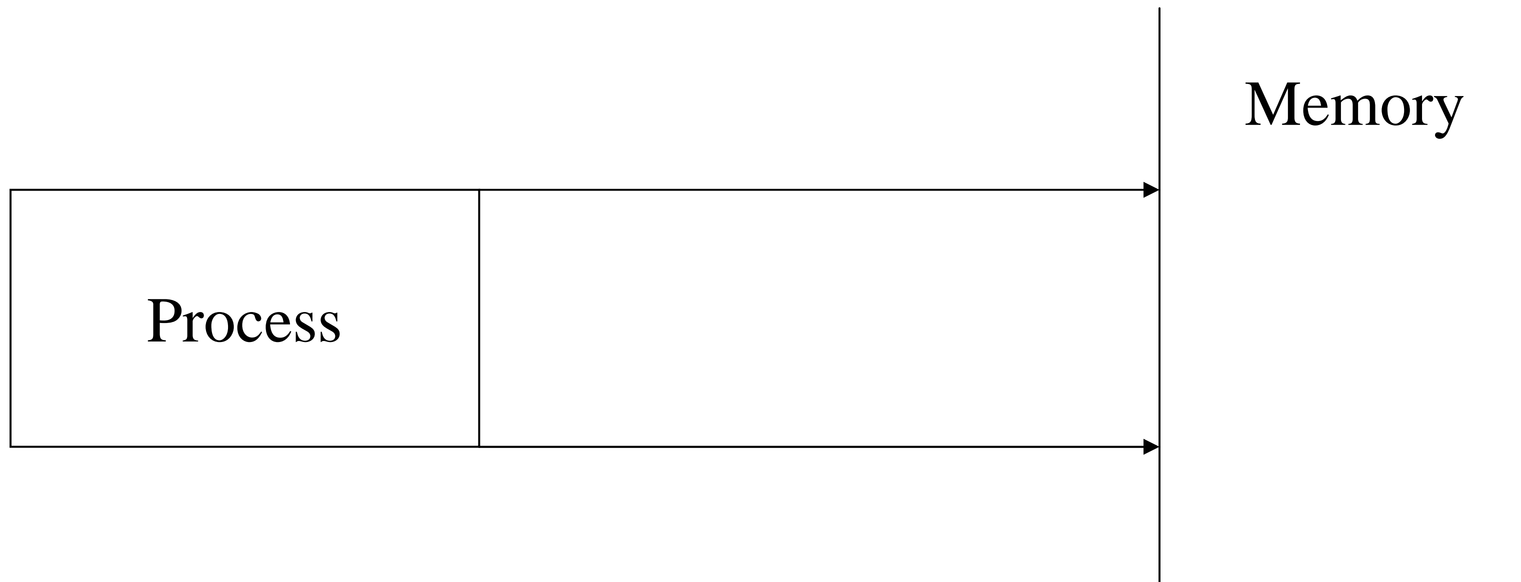

- Memory is allocated in block/partition/junks
- Giving back a small amount of memory to the memory manager is not feasible
- Overhead of managing a few left-over bytes is not worth the effort

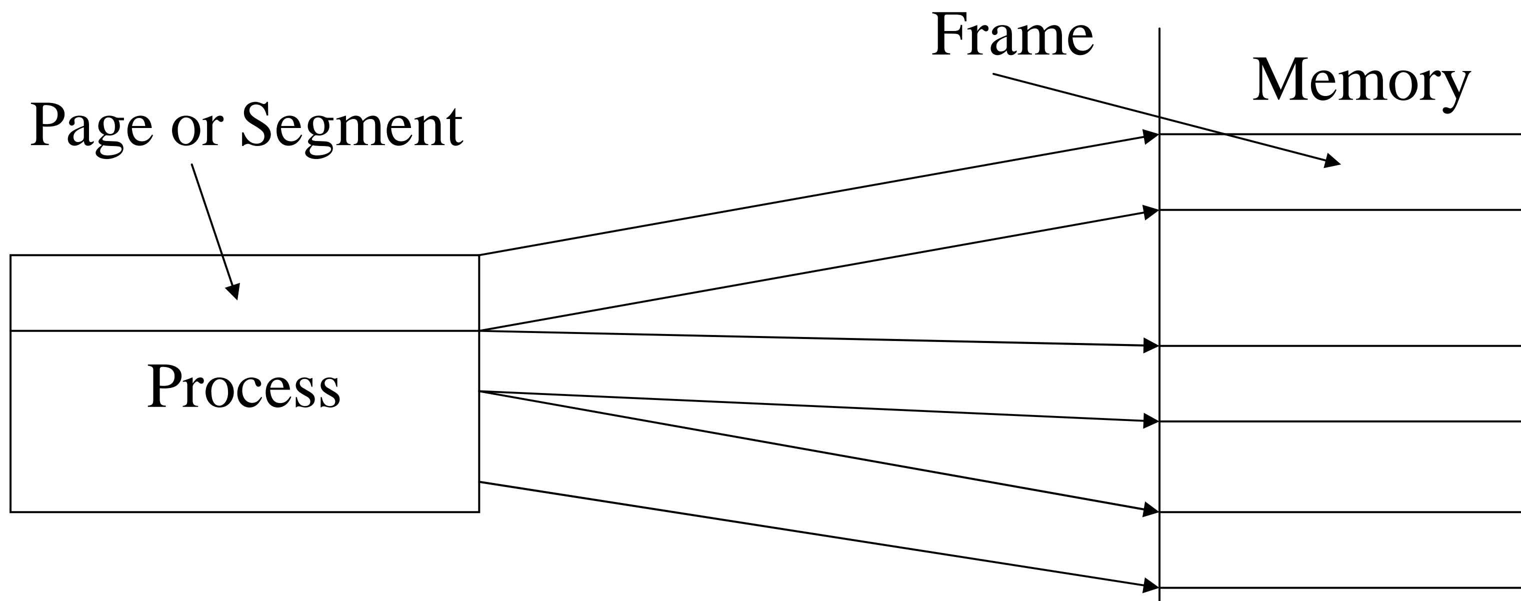
# Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, **but it is not contiguous.**
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible ***only*** **if address binding is dynamic**, and is done at **execution time**.

# Preview

- The problem so far has been that we allocated memory in **contiguous junks**
- What if we could allocate memory in **non-contiguous junks**?
- We will be looking at techniques that **aim at avoiding**
  - External fragmentation
  - (Internal fragmentation)


Memory
Process

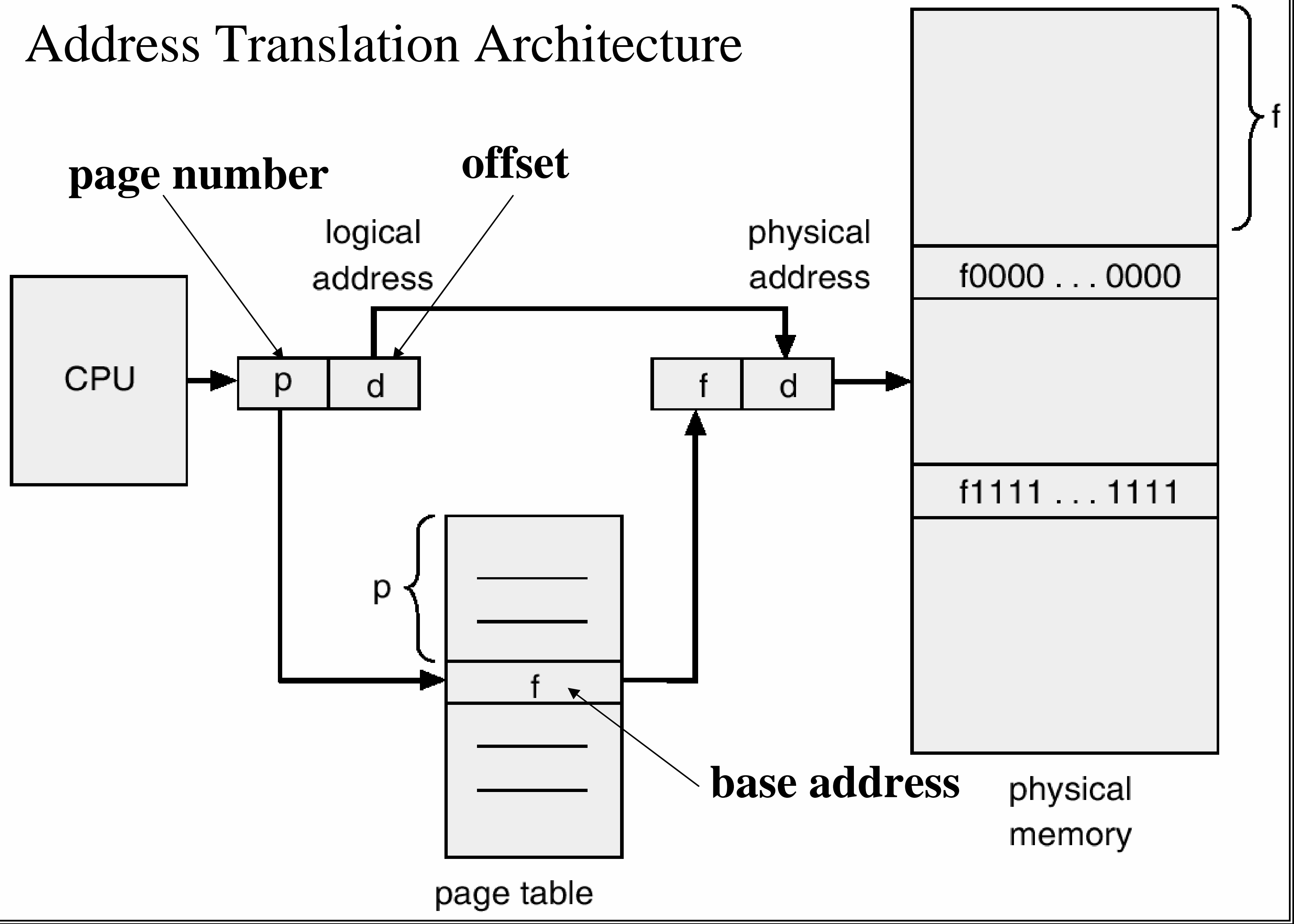

Frame
Memory
Page or Segment
Process

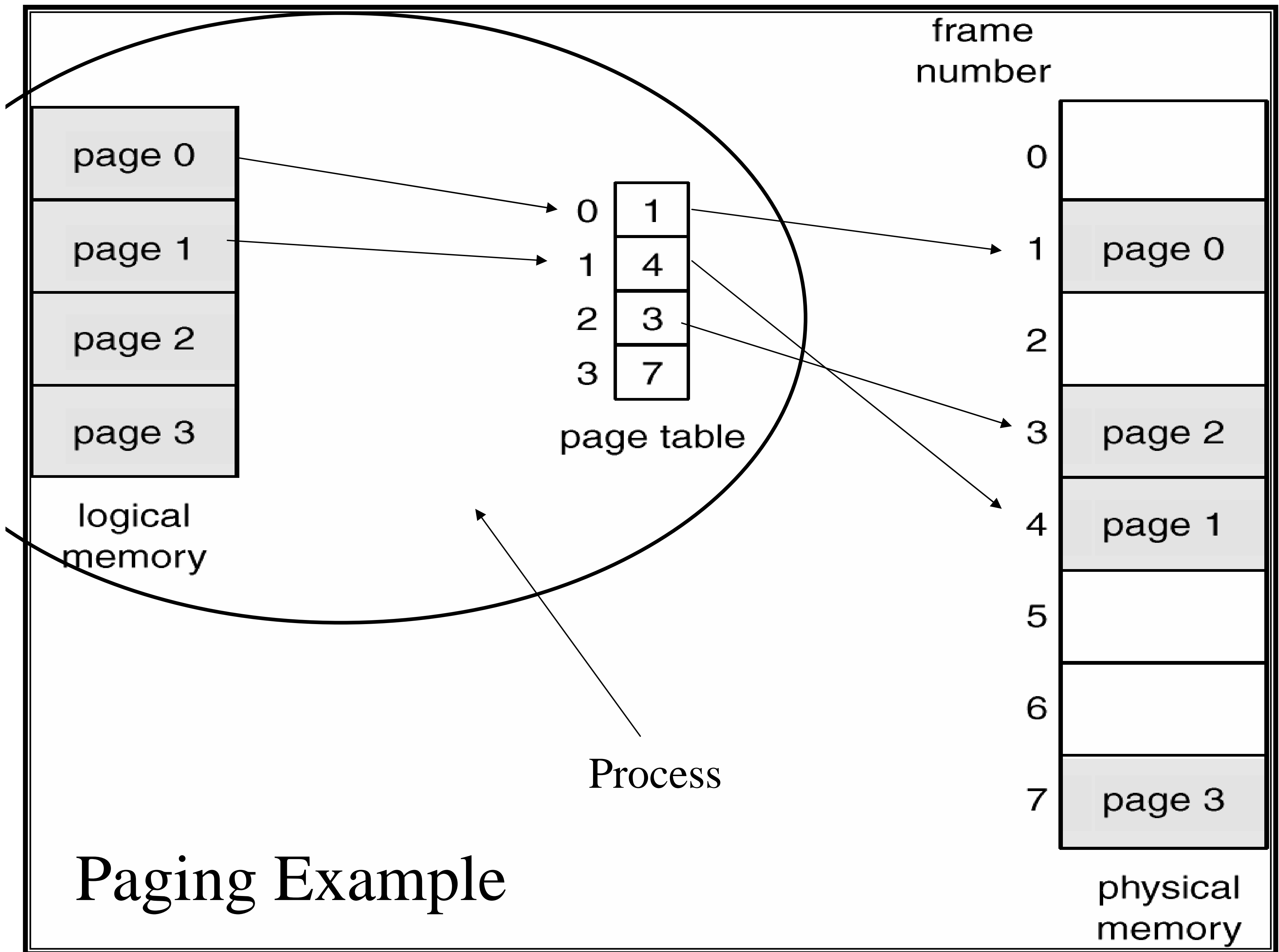
# Paging

- Physical address space of a process can be **non-contiguous**;
- Process is allocated physical memory **whenever the latter is available**.
- **Divide physical memory** into **fixed-sized blocks** called **frames** (size is power of 2, between 512 bytes and 8192 bytes, also larger sizes possible in practice.)
- **Divide logical memory** into blocks of same size called **pages**.
- Keep track of all free frames.
- To run a program of **size *n* pages**, need to **find *n* free frames** and load program.
- Set up a **page table** to translate logical to physical addresses.
- Internal fragmentation, for last page

# Address Translation Scheme

- Address generated by CPU is divided into:
  - ***Page number*** *(p)*
    - Used as an **index** into the page table
    - Page table contains **base address** of each page in physical memory.
  - ***Page offset*** *(d)*
    - **combined with base address** to define the physical memory address sent to the memory unit.


Address Translation Architecture
page number
offset
logical
address
physical
address
CPU
p
d
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
base address
physical
memory
page table


frame number
page 0
page 1
page 2
page 3
logical memory
0 1
1 4
2 3
3 7
page table
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
physical memory
Process
Paging Example

# Paging Example

| | |
|---|---|
| 0 | a |
| 1 | b |
| 2 | c |
| 3 | d |
| 4 | e |
| 5 | f |
| 6 | g |
| 7 | h |
| 8 | i |
| 9 | j |
| 10 | k |
| 11 | l |
| 12 | m |
| 13 | n |
| 14 | o |
| 15 | p |

logical memory

| | |
|---|---|
| 0 | 5 |
| 1 | 6 |
| 2 | 1 |
| 3 | 2 |

page table

| | |
|---|---|
| 0 | |
| 4 | i j k l |
| 8 | m n o p |
| 12 | |
| 16 | |
| 20 | a b c d |
| 24 | e f g h |
| 28 | |

physical memory

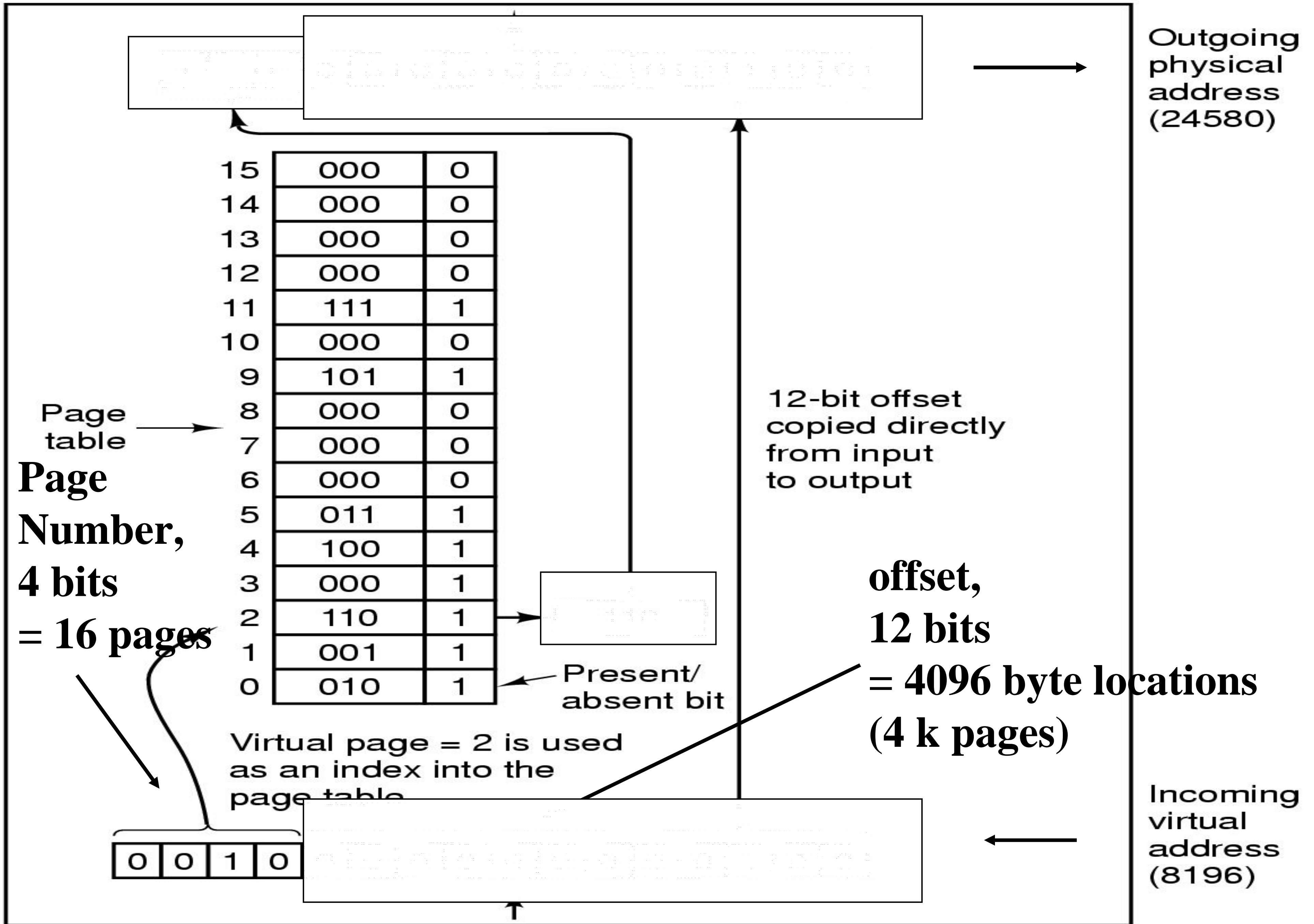
Page size is 4

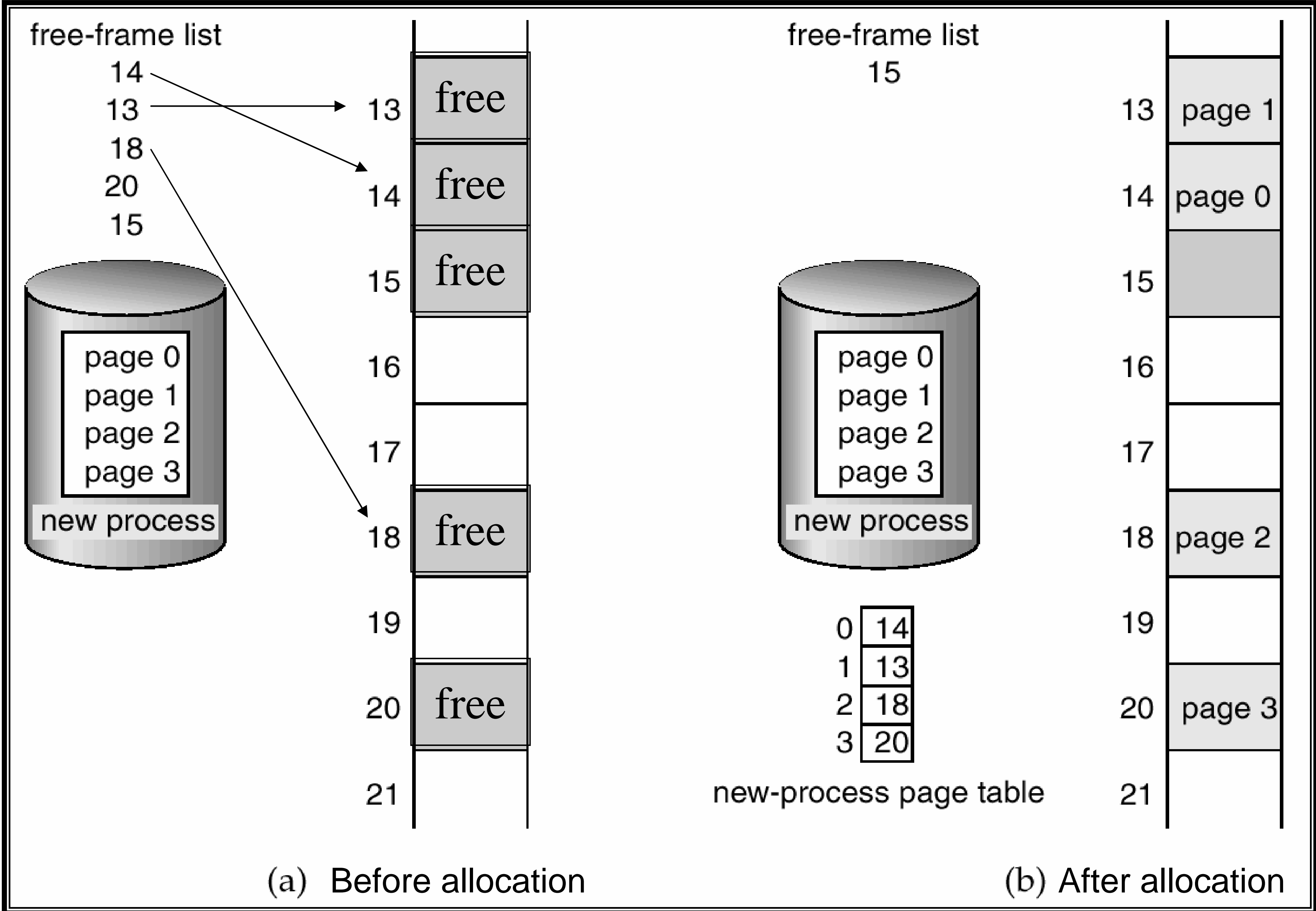
Page 0 is in Frame 5, located at address 20

i.e., 5 x 4 = 20

Logical address (1,3) (=7) is mapped to 27

(6 x 4 + 3 = 27)


Outgoing physical address (24580)
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
Page table
12-bit offset copied directly from input to output
Present/ absent bit
Virtual page = 2 is used as an index into the page table
Incoming virtual address (8196)
0 0 1 0
Page Number, 4 bits = 16 pages
offset, 12 bits = 4096 byte locations (4 k pages)



(a) Before allocation (b) After allocation

# Basic Implementation of a Page Table

- Page table is kept in main memory.
- ***Page-table base register*** *(*PTBR) points to the page table (part of process context).
- ***Page-table length register*** (PRLR) indicates size of the page table (part of process context).

# Implementation of a Page Table

*Efficiency?*

- **Every data and instruction access requires two memory accesses**:
  - one for the page table and
  - one for the data or instruction.
- A typical instruction has 1, 2, … operands, which all typically require memory access (through page table).
- This is pretty inefficient, if done in software.
- **Mapping** has to be **fast**.

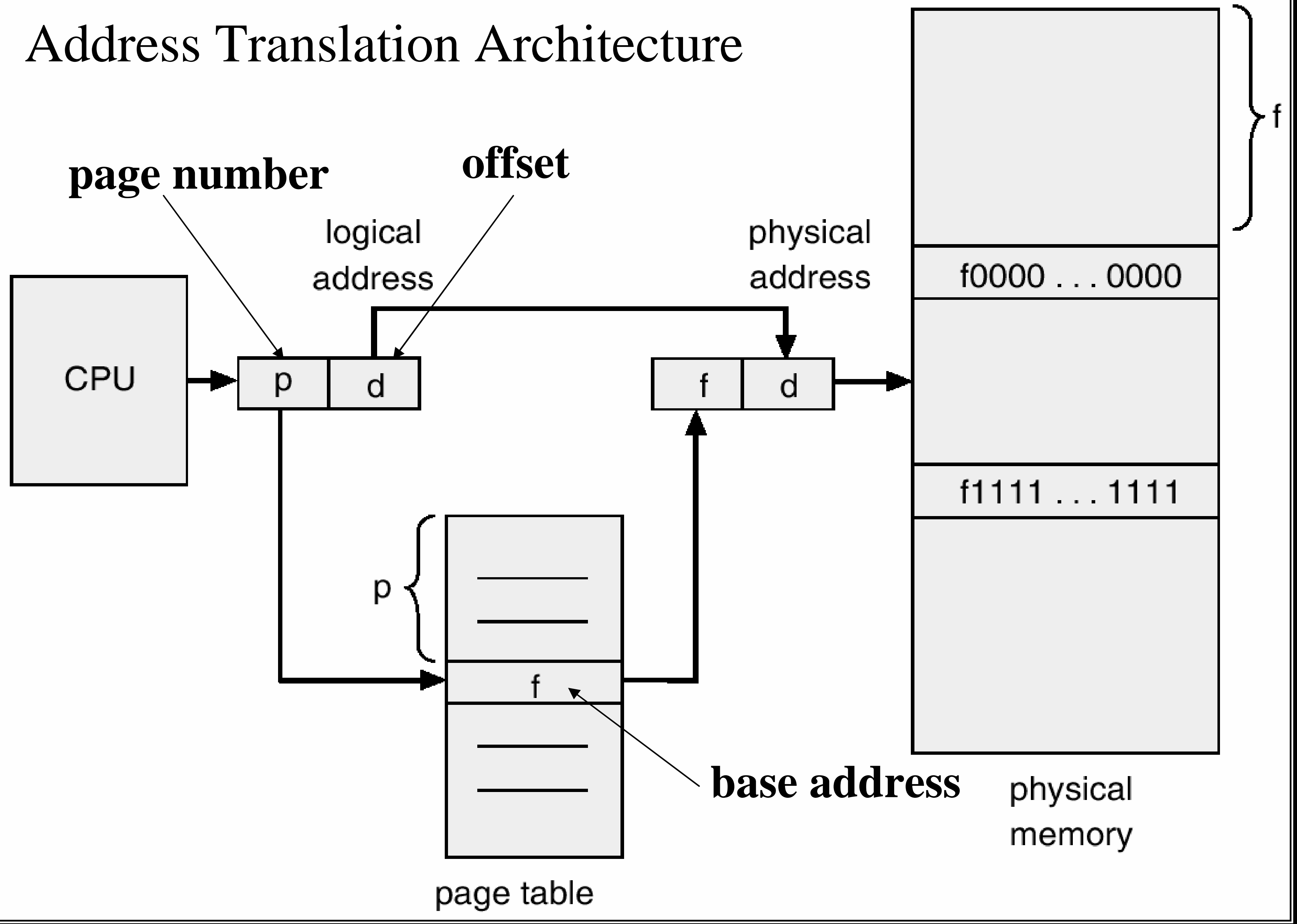
# Implementation of a Page Table

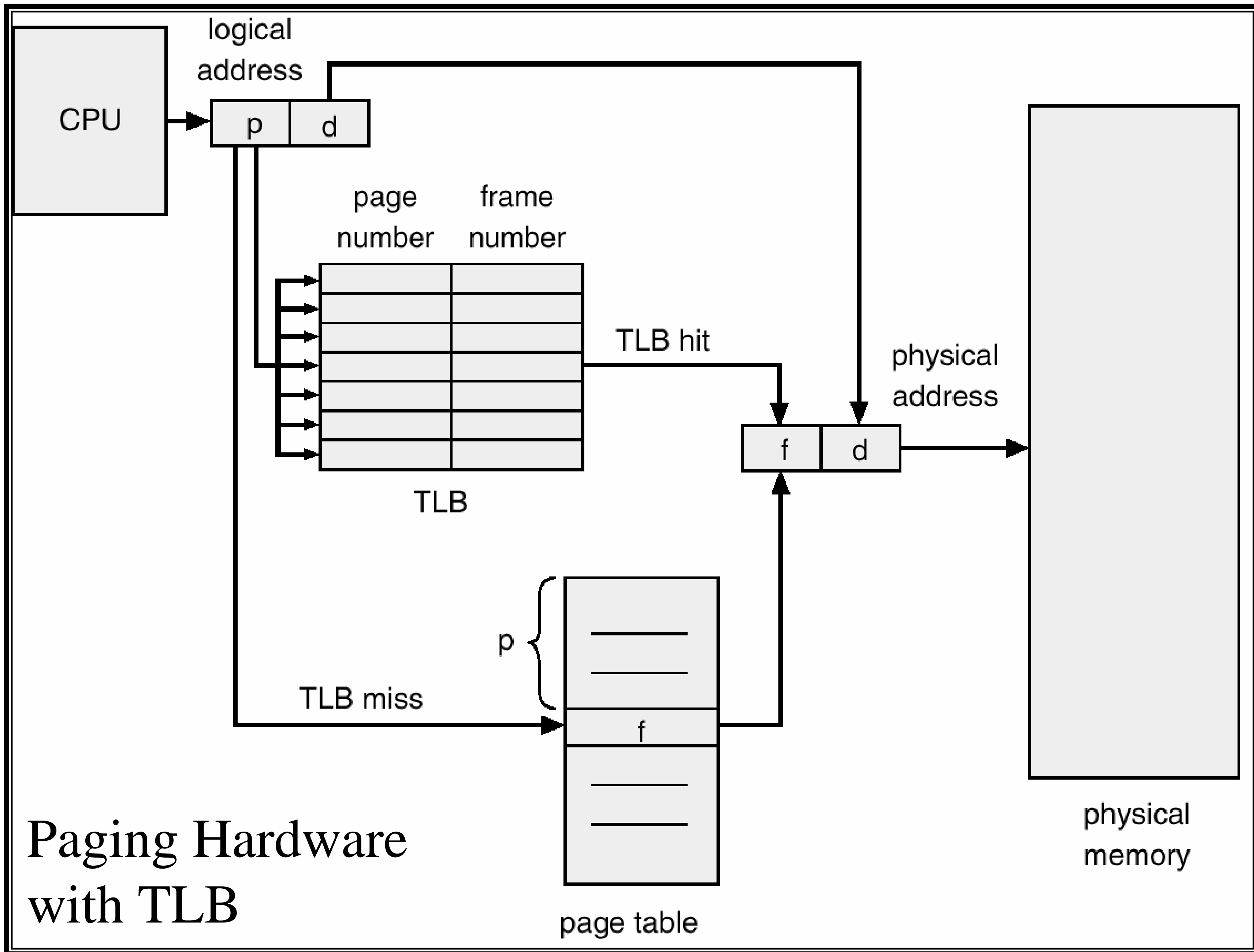
*Size?*

- 32 bit virtual address and 4k page size results in 1 million pages ($2^{32}/2^{12} = 2^{20}$)
- 4 byte page table entry, results in a 4MB table
- Page table requires 1 million entries, each process has its own table
- Page table can be **extremely large.**

# Implementation of a Page Table

- Page table as a set of registers
  - Adds to context switch overhead
  - Page table usually too large
- The two-memory access problem can be solved by the use of a **special fast-lookup hardware cache** called ***associative memory***
- *A.k.a.* a ***translation look-aside buffer*** *(TLB)*


Address Translation Architecture
page number
offset
logical
address
physical
address
CPU
p
d
f
d
f0000 . . . 0000
f1111 . . . 1111
f
p
f
base address
page table
physical
memory




# Paging Hardware with TLB

# Page Table Structure

- Addressing the page table size problem
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables

# Hierarchical Page Tables

- Allocating the page table contiguously in memory is not feasible
- Break up the logical address space into multiple page tables
- Recursively apply the paging scheme to the page table itself

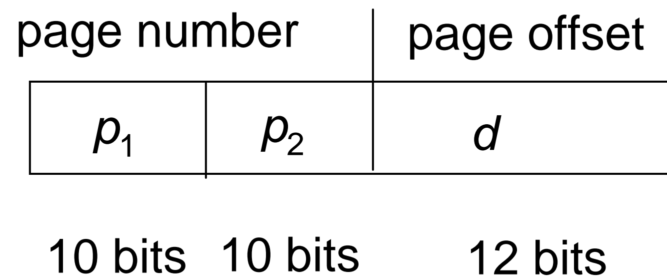
- A simple technique is a two-level page table

# Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - A page number consisting of 20 bits.
  - Possible address space of size $2^{20}$ pages.
  - A page offset consisting of 12 bits.
  - 12 bits can address 4096 bytes (i.e., all bytes in the 4k page).
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.

# Two-level Address

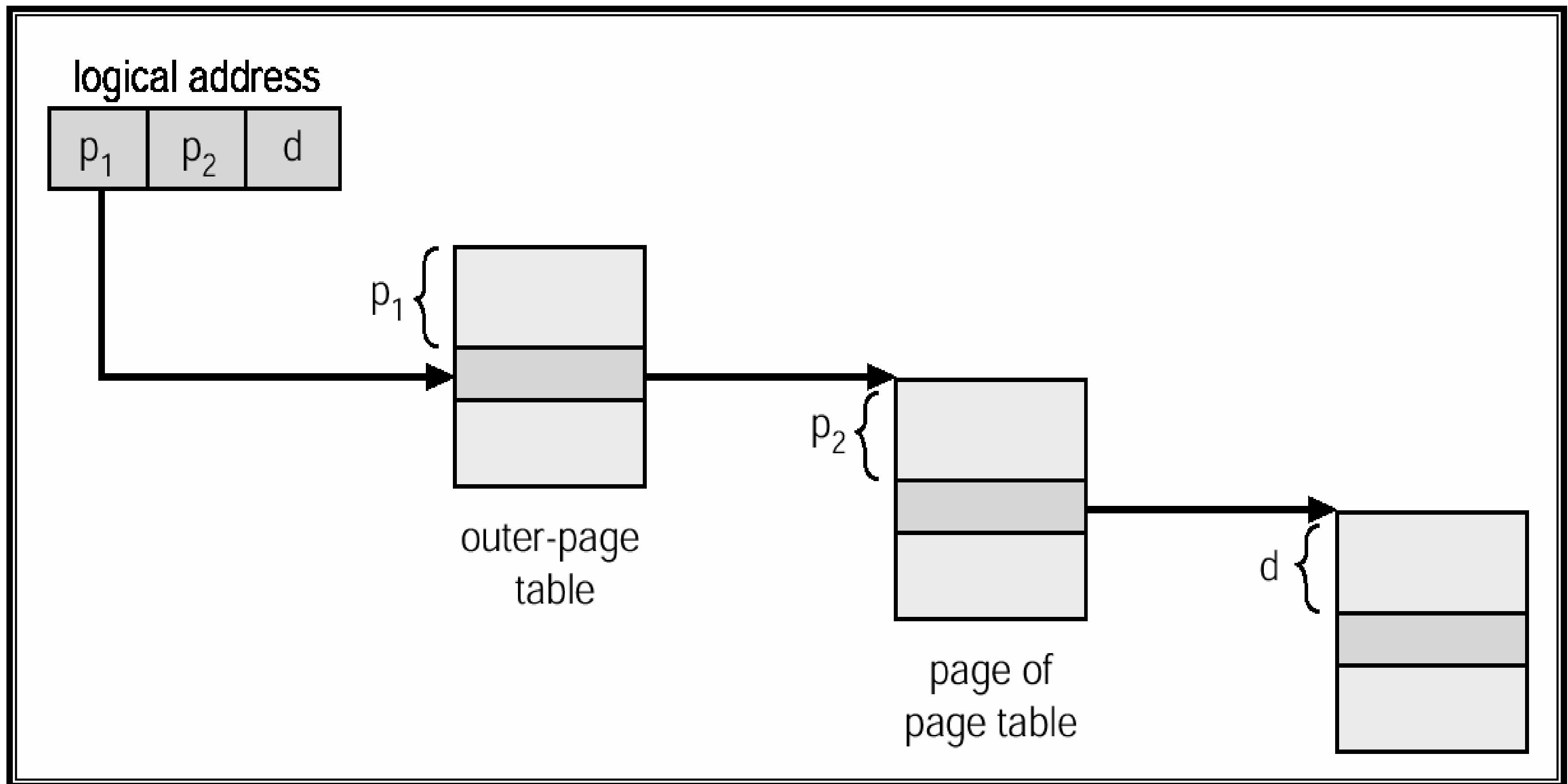
- Thus, a logical address is as follows:

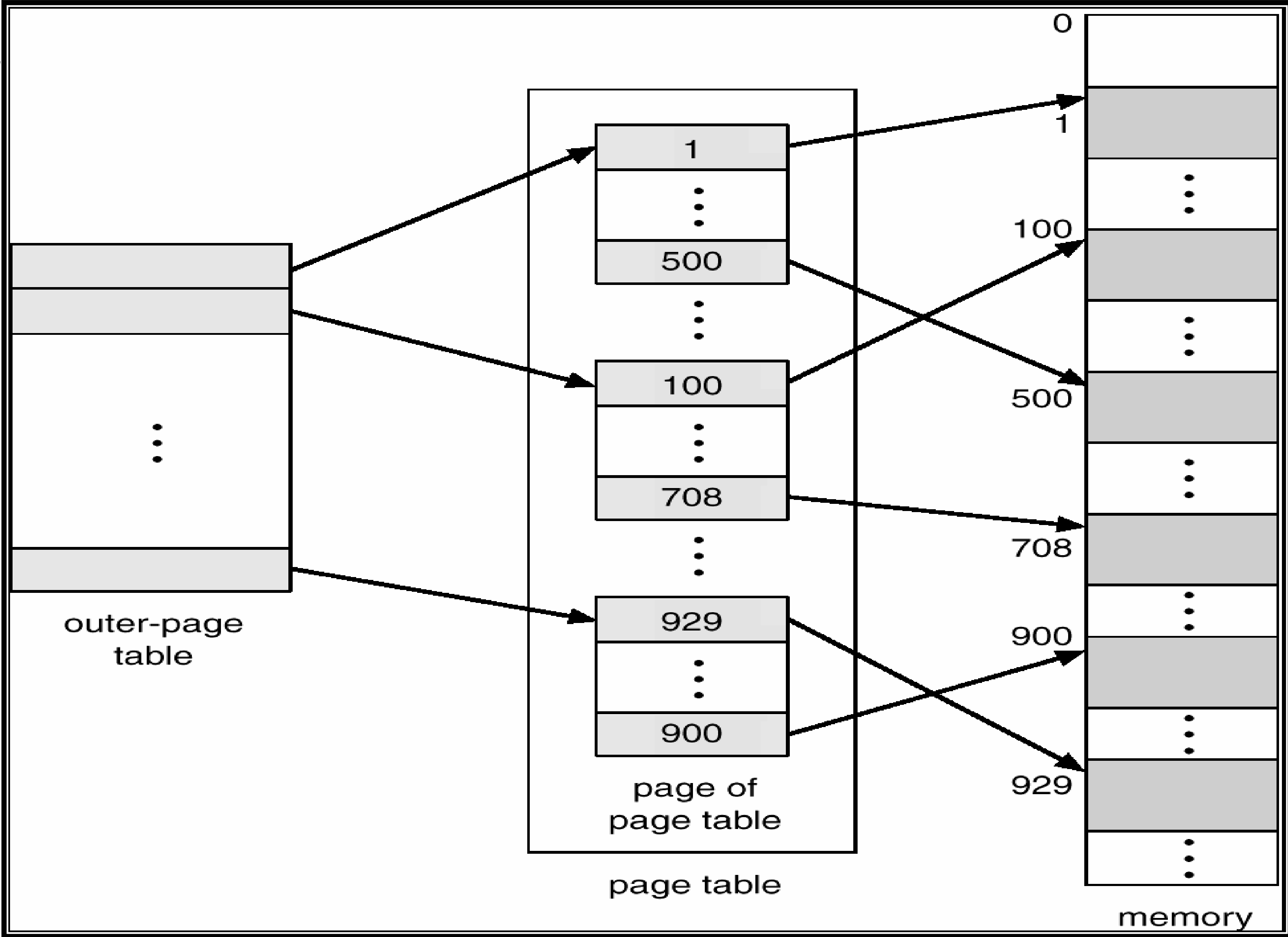



- where $p_1$ is an index into the **outer page table**, and $p_2$ is the displacement within the page of the (**inner**) page table.

# Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

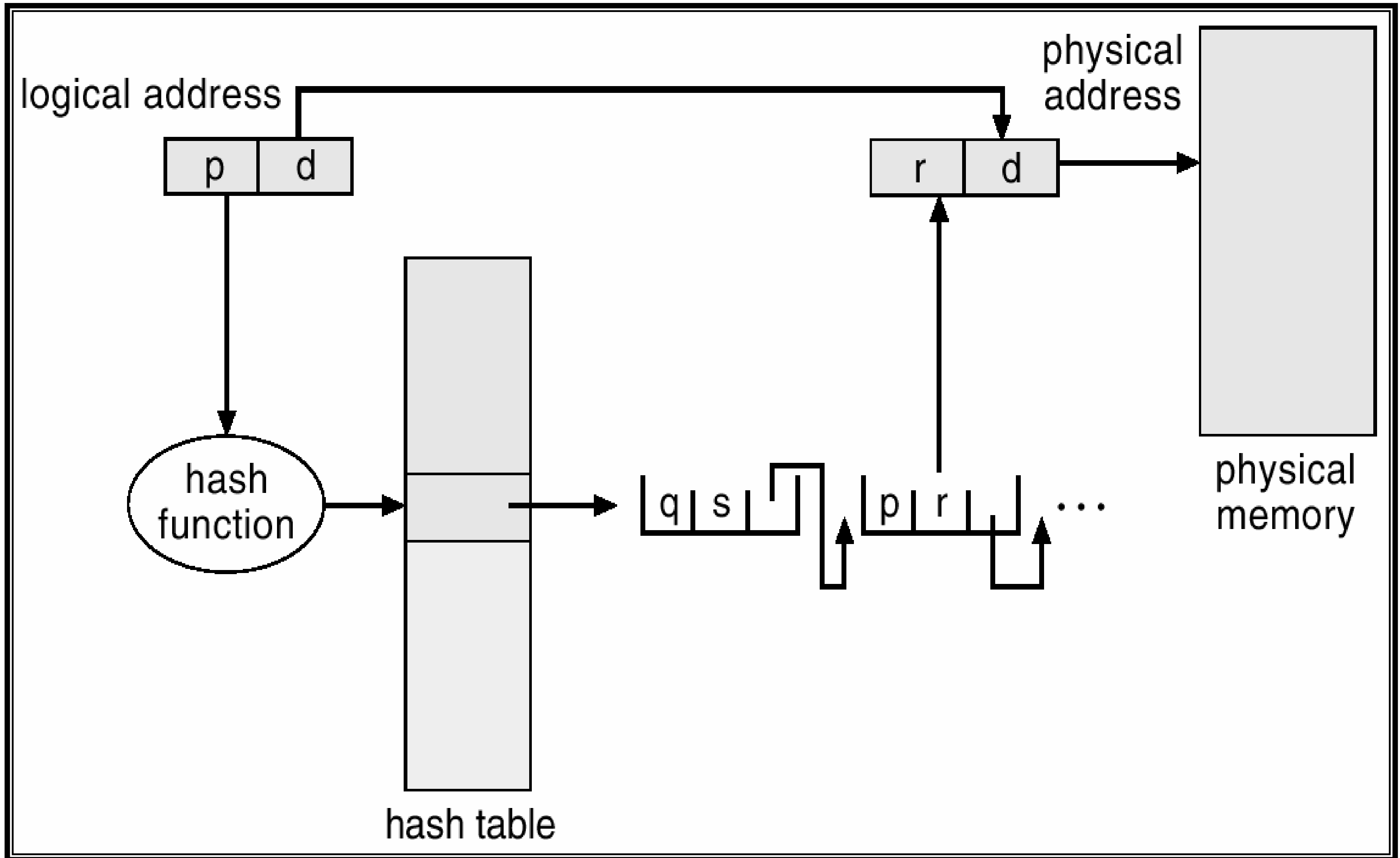



outer-page table
1
500
100
708
929
900
page of page table
page table
0
1
100
500
708
900
929
memory

# Hashed Page Tables

- Common in address spaces > 32 bits.
- The virtual page number is hashed into a page table.
- This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain searching for a match. If a match is found, the corresponding physical frame is extracted.

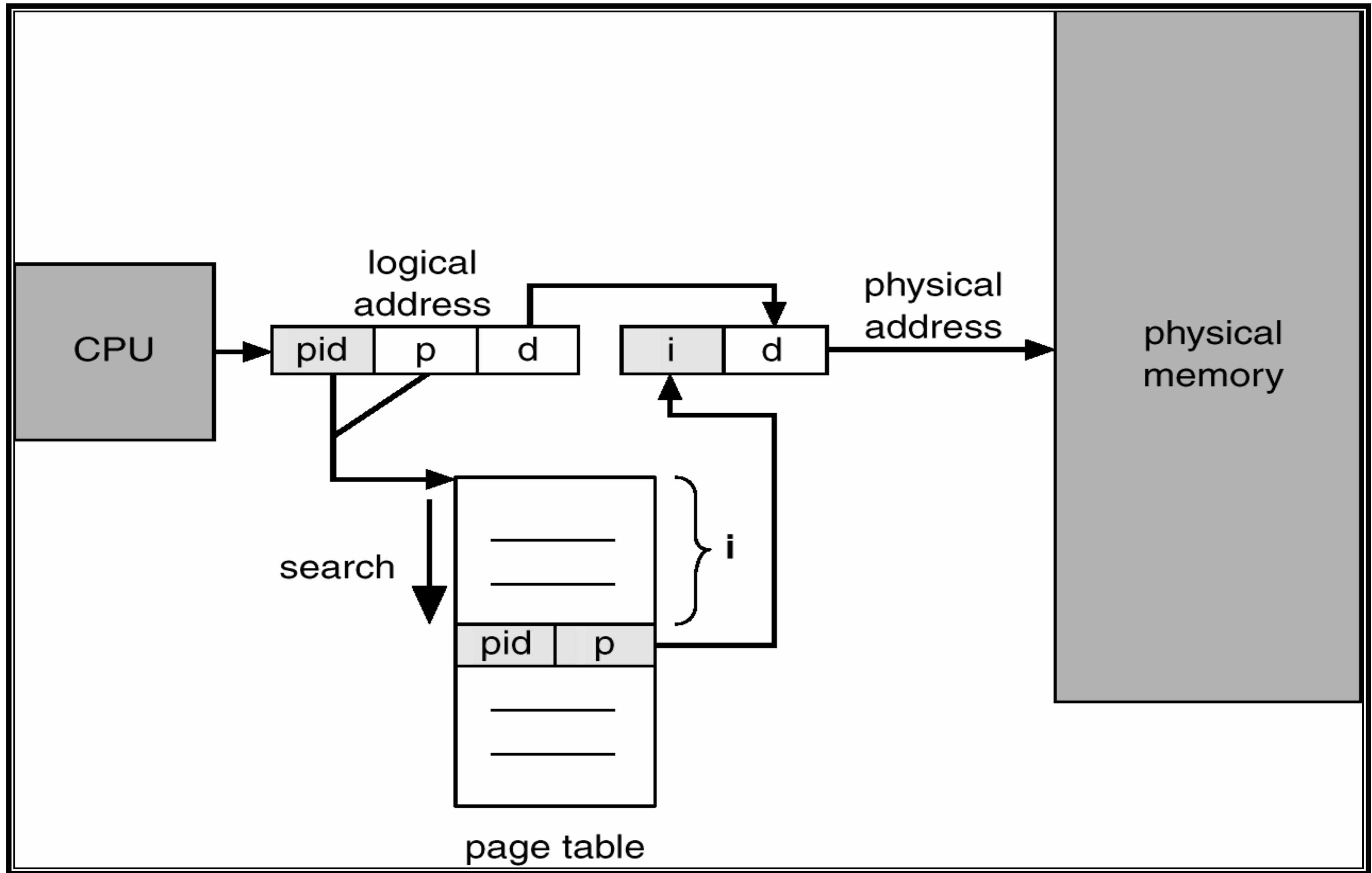
# Hashed Page Table



# Inverted Page Table

- One entry for each real frame of memory.
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- **Decreases memory** needed to store each page table, but **increases time** needed to search the table when a page reference occurs.
- **Use hash table** to limit the search to one — or at most a few — page-table entries.

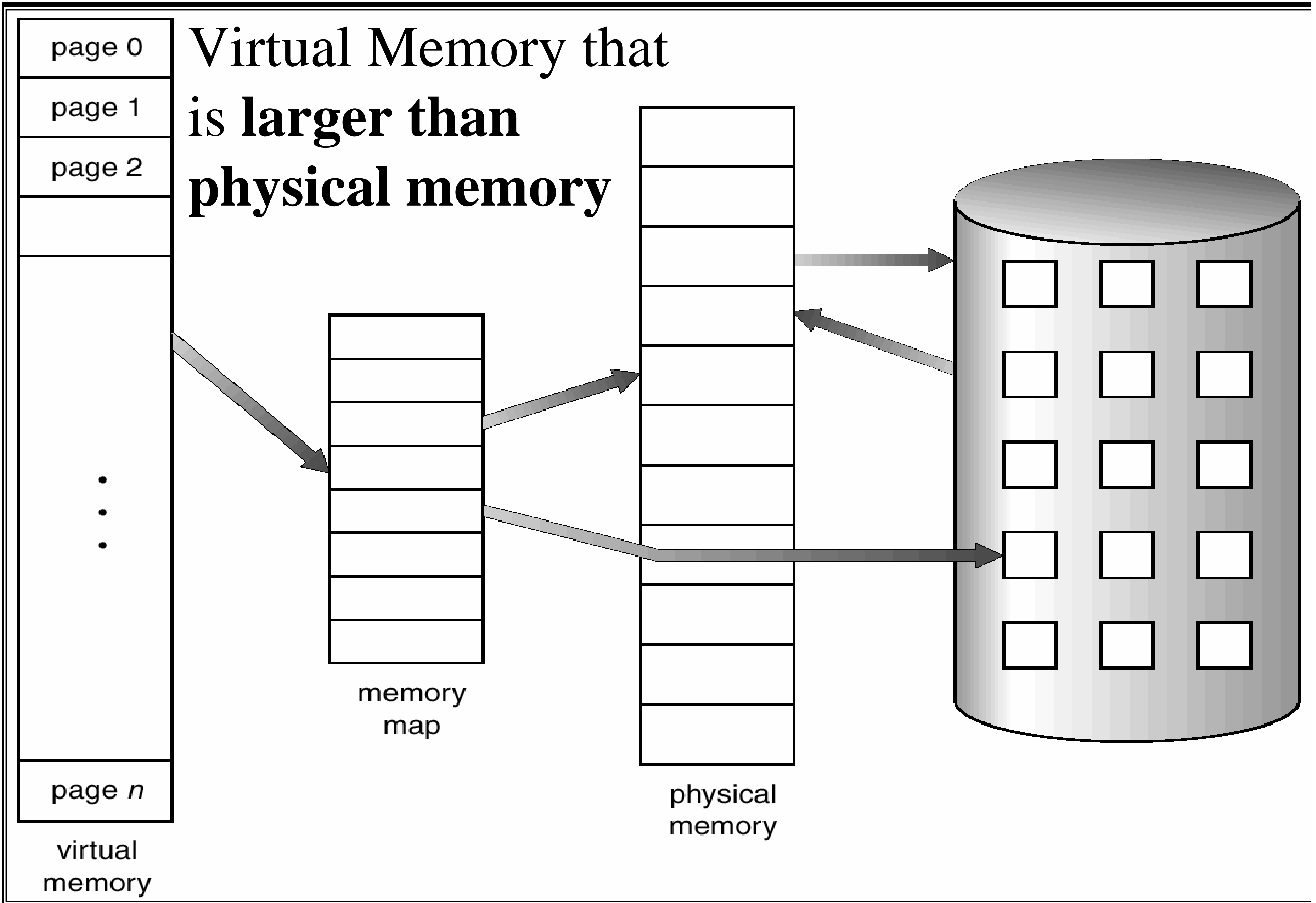
# Inverted Page Table Architecture



# Virtual Memory

# Virtual Memory

- Only **part of the program** needs to be in memory for execution.
- **Logical address space** can therefore be much **larger than physical address space**.
- Physical address spaces can be shared by several processes.
- More **efficient process creation.**
- Virtual memory can be implemented via
  - **Demand paging**
  - Demand segmentation


page 0
page 1
page 2
page n
virtual memory
Virtual Memory that is larger than physical memory
memory map
physical memory

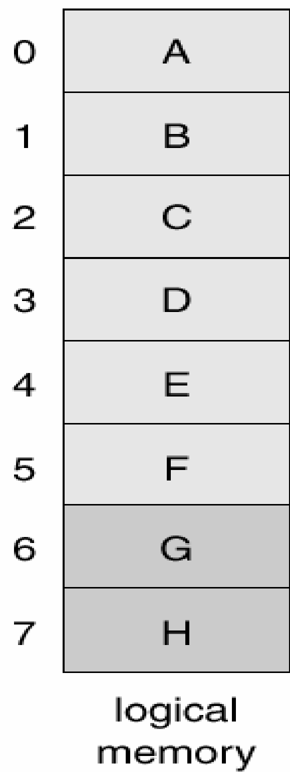
# Demand Paging

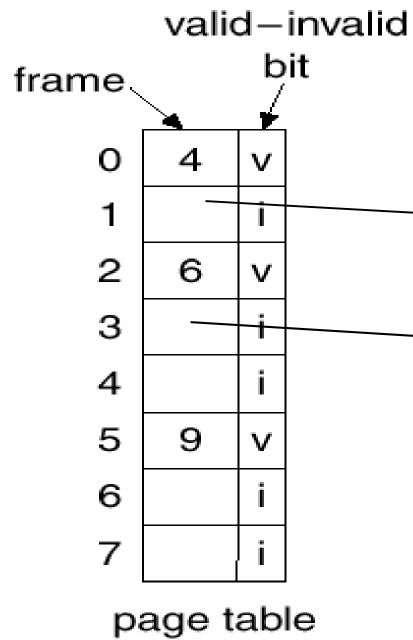
- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

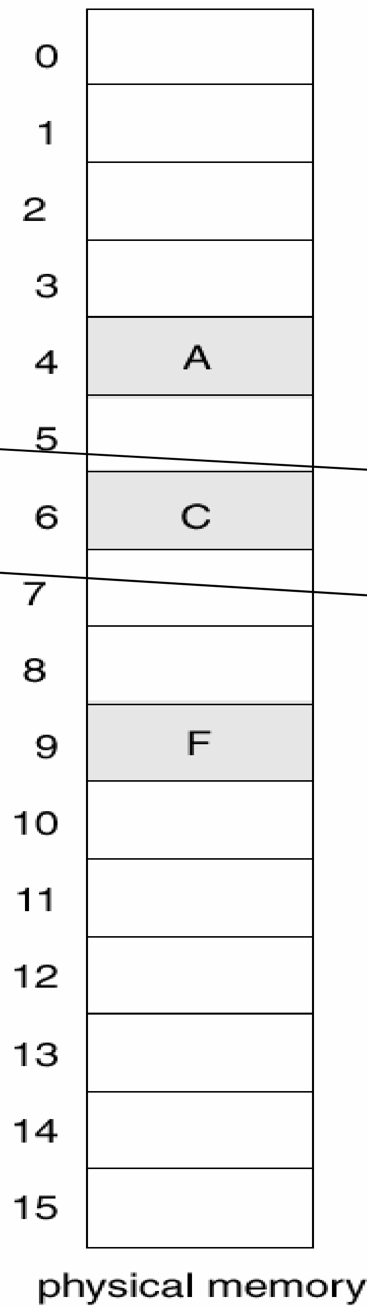
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory

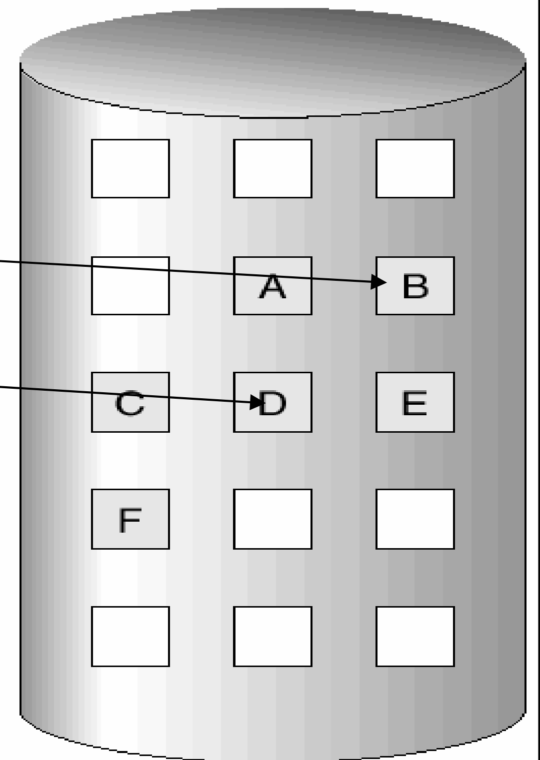
# Valid-Invalid Bit

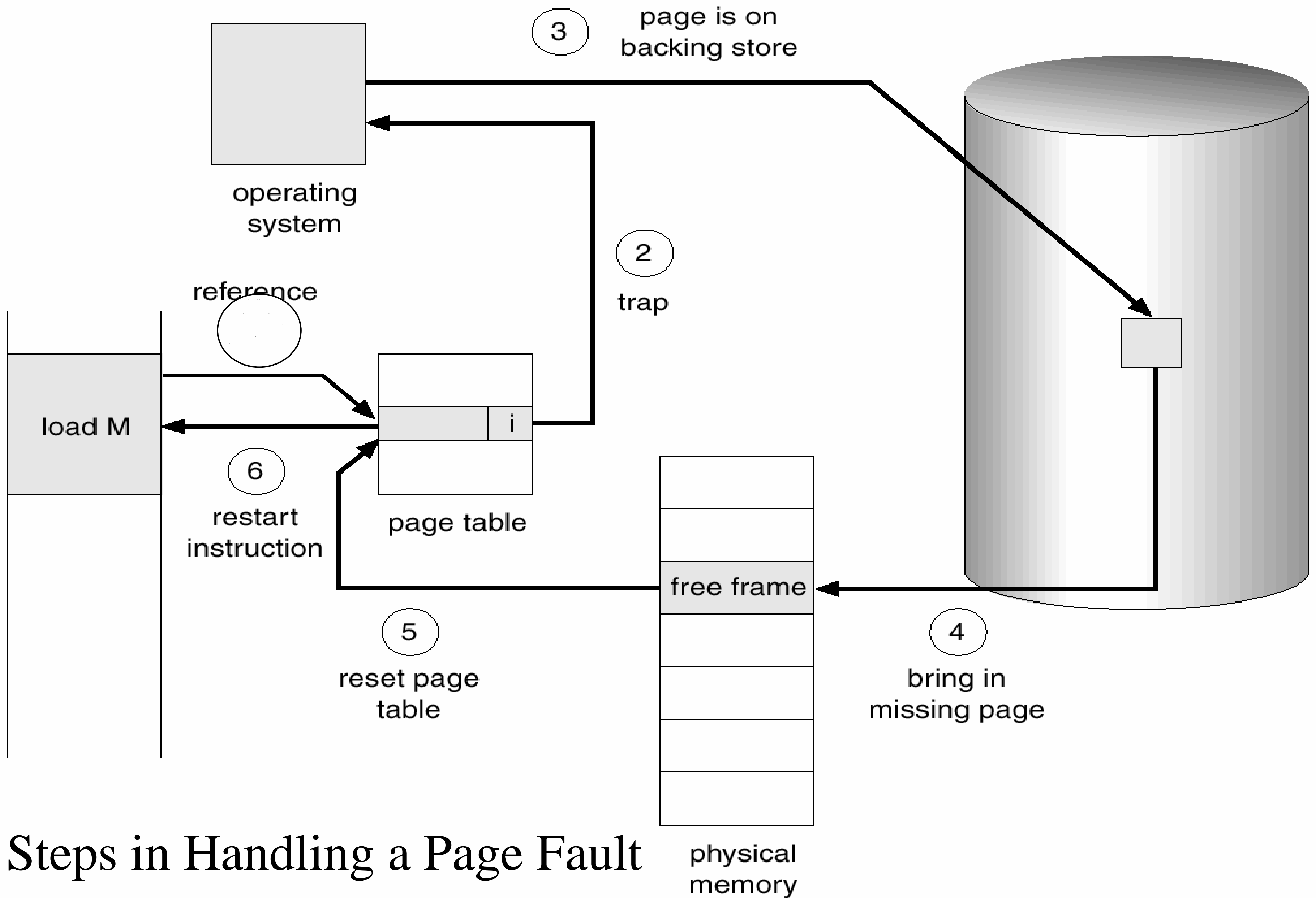
- With each page table entry a **valid–invalid bit** is associated
  (1 $\Rightarrow$ in-memory, 0 $\Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to 0 on all entries
- During address translation, if valid–invalid bit in page table entry is 0 $\Rightarrow$ **page fault**
- **Demand paging** (all bits initially 0)


0
A
1
B
2
C
3
D
4
E
5
F
6
G
7
H
logical memory


valid–invalid bit
frame
0 4 v
1 i
2 6 v
3 i
4 i
5 9 v
6 i
7 i
page table


0
1
2
3
4 A
5
6 C
7
8
9 F
10
11
12
13
14
15
physical memory


A
B
C
D
E
F



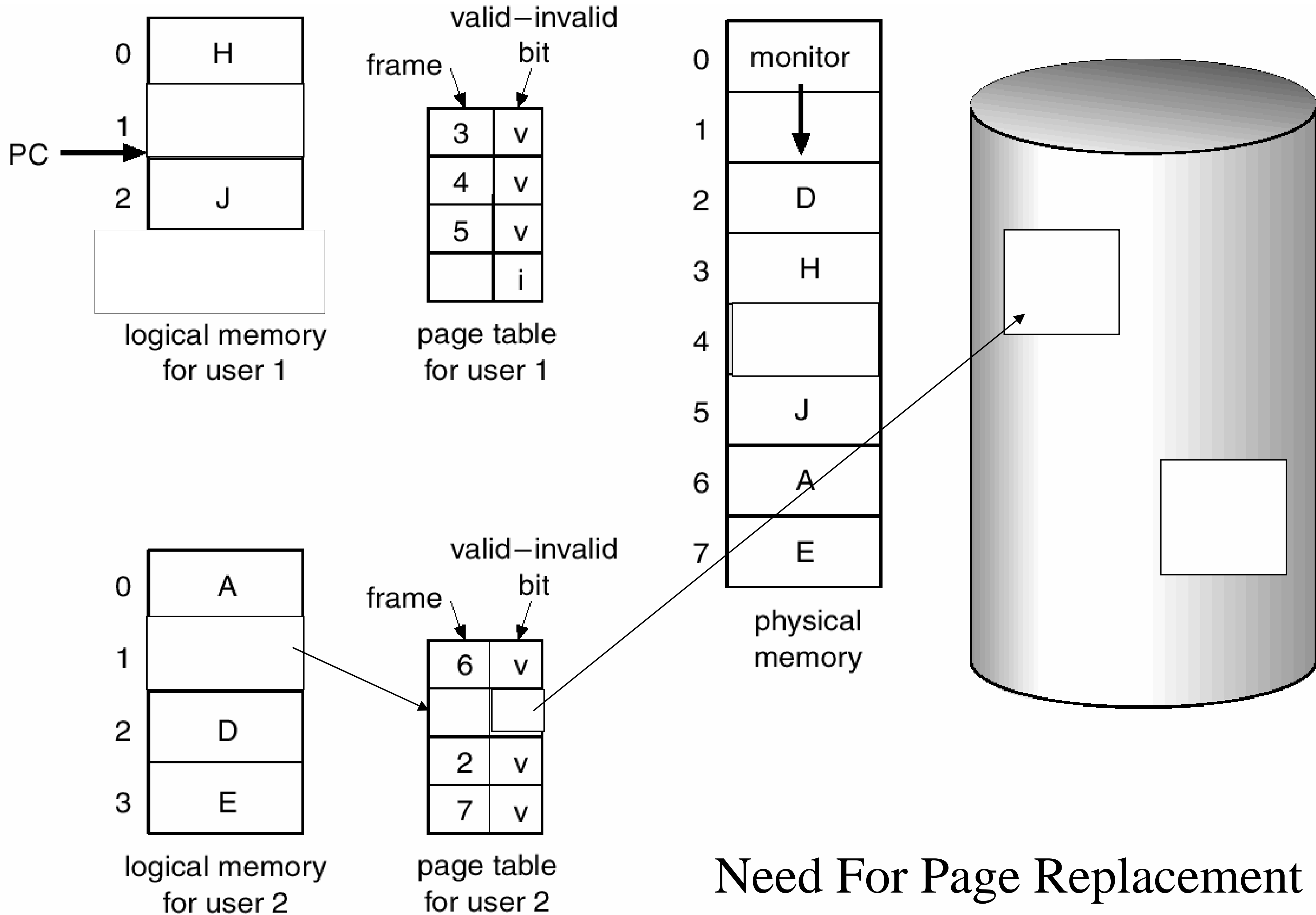

# Steps in Handling a Page Fault

# What happens if there is no free frame?

- **Page replacement** – find some page in memory, *but not really in use*, swap it out.
  - algorithm
  - performance
    - algorithm should result in **minimum number of page faults**
- Same page may be brought into memory several times.

# Page Replacement

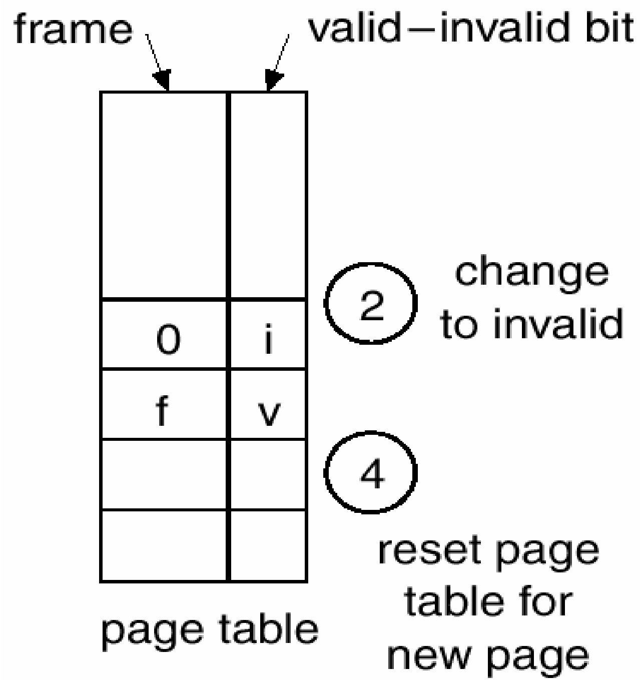
- **Prevent over-allocation of memory** by modifying page-fault service routine to include **page replacement**
- Use *modify* (***dirty***) *bit* to reduce overhead of page transfers
  - only modified pages need to be written to disk
- Page replacement completes separation between logical memory and physical memory
- **Thus large virtual memory can be provided on a smaller physical memory**




# Need For Page Replacement

# Basic Page Replacement

- **Find** the location of the desired **page on disk**
- 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**
- Read the desired page into the (newly) freed frame
- Update the page table
- Restart the process



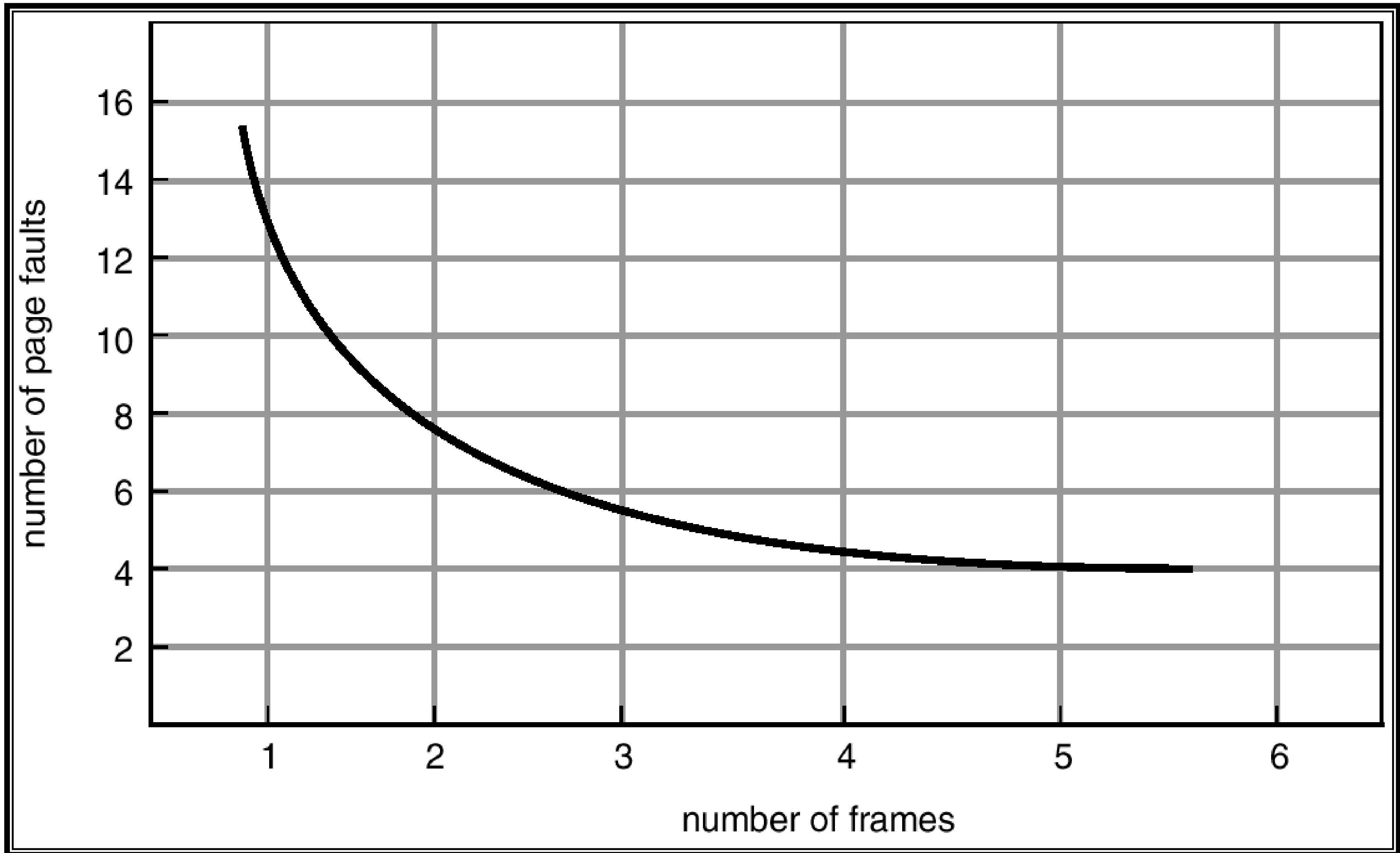

# Page Replacement

# Page Replacement Algorithms

- Want lowest page-fault rate.
- Evaluate algorithm by running it on a particular string of memory references (reference string)
- Compute 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.

# Graph of Page Faults Versus The Number of Frames



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

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- **Replace oldest page**
- 3 (4) frames (3 (4) pages can be in memory at a time per process)
  - initialization code
  - frequently used code

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

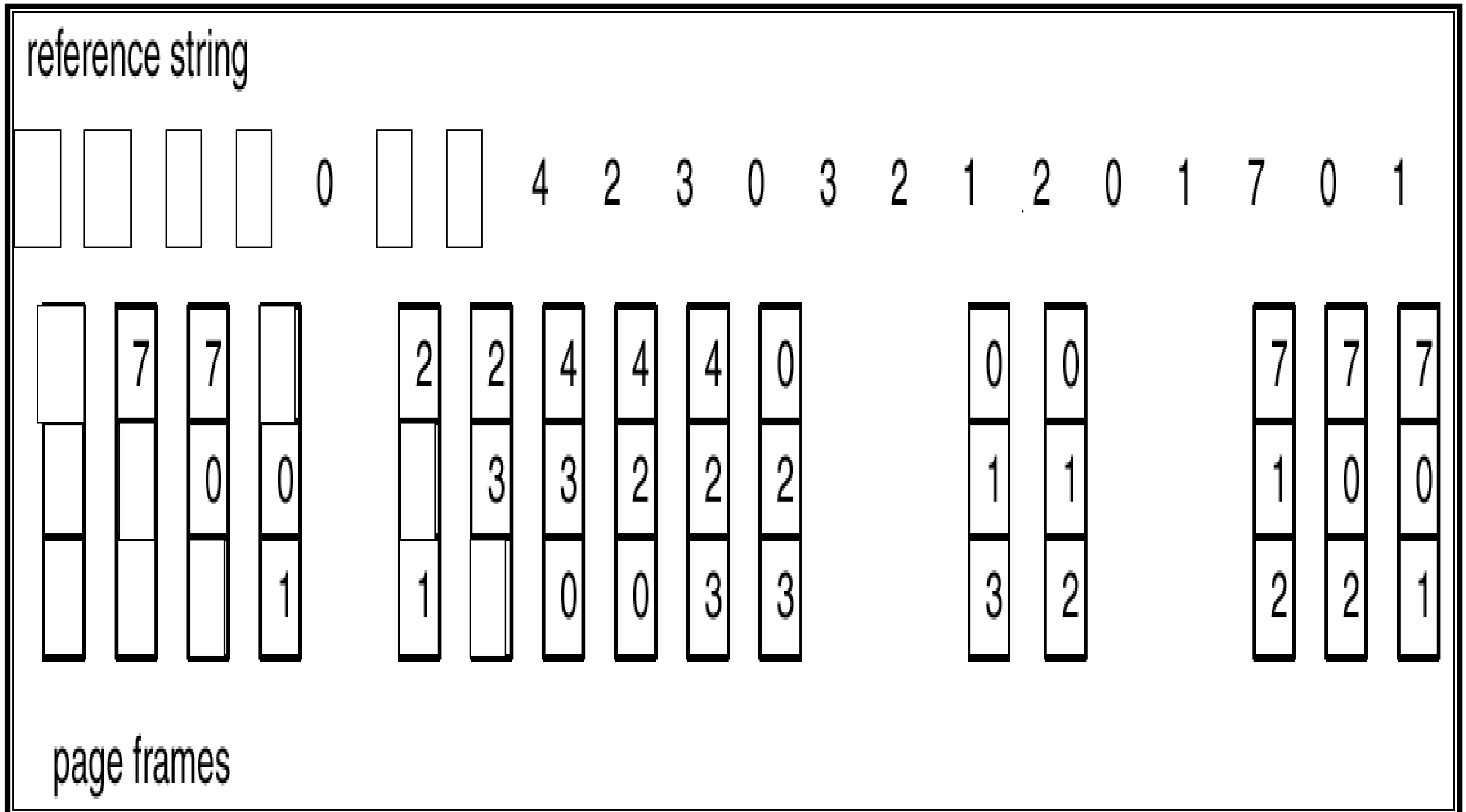
9 page faults

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

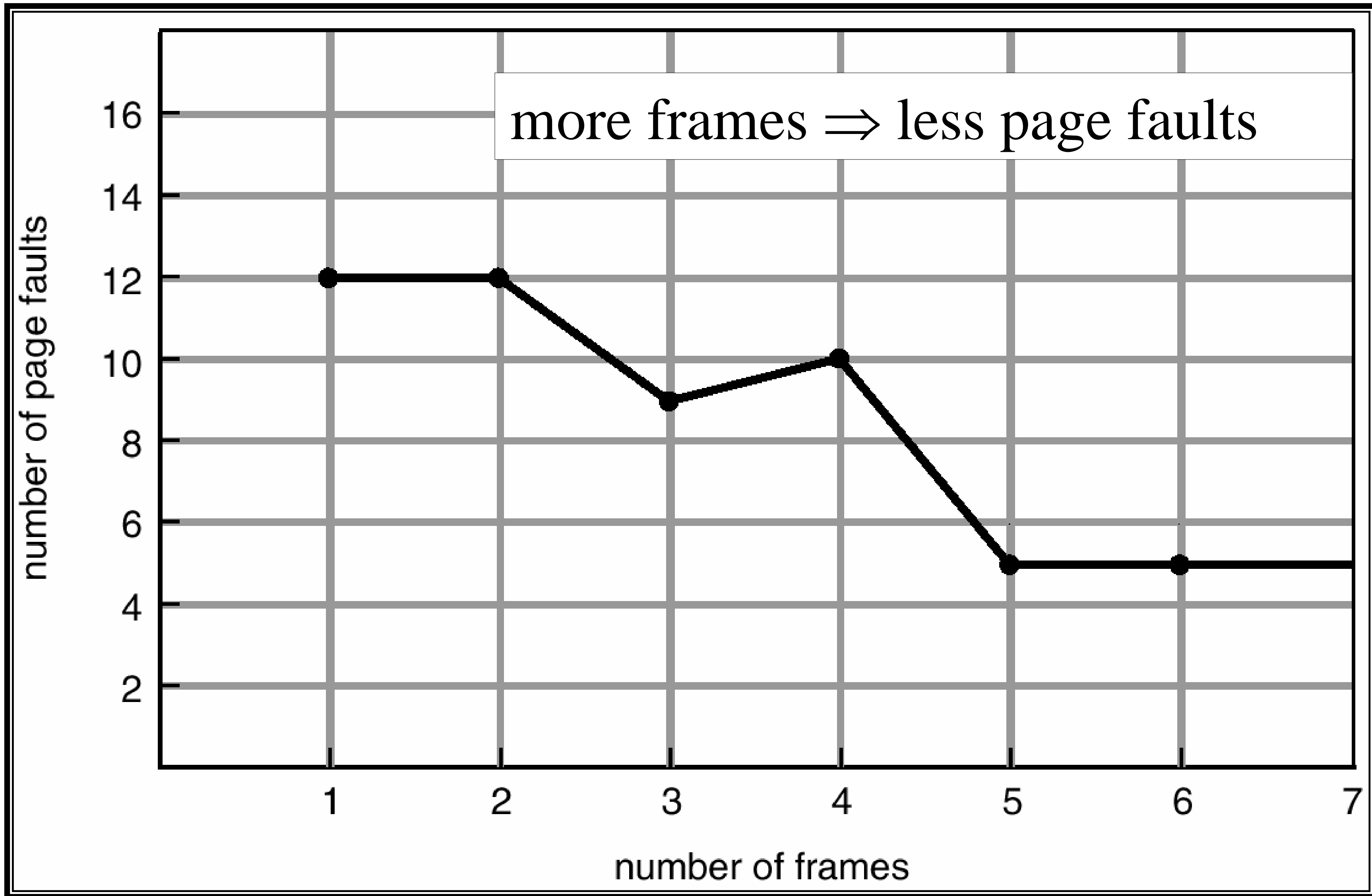
10 page faults

- More frames, more faults )-: !
- Implemented with FIFO-queue

# FIFO Page Replacement



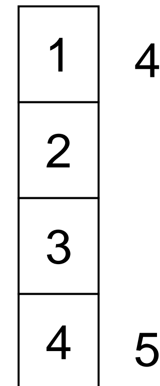
# FIFO Illustrating Belady's Anamoly (1976)



# Optimal Algorithm

- Replace page that will **not be used for longest period of time** (cf. SJF)
- A 4 frames example

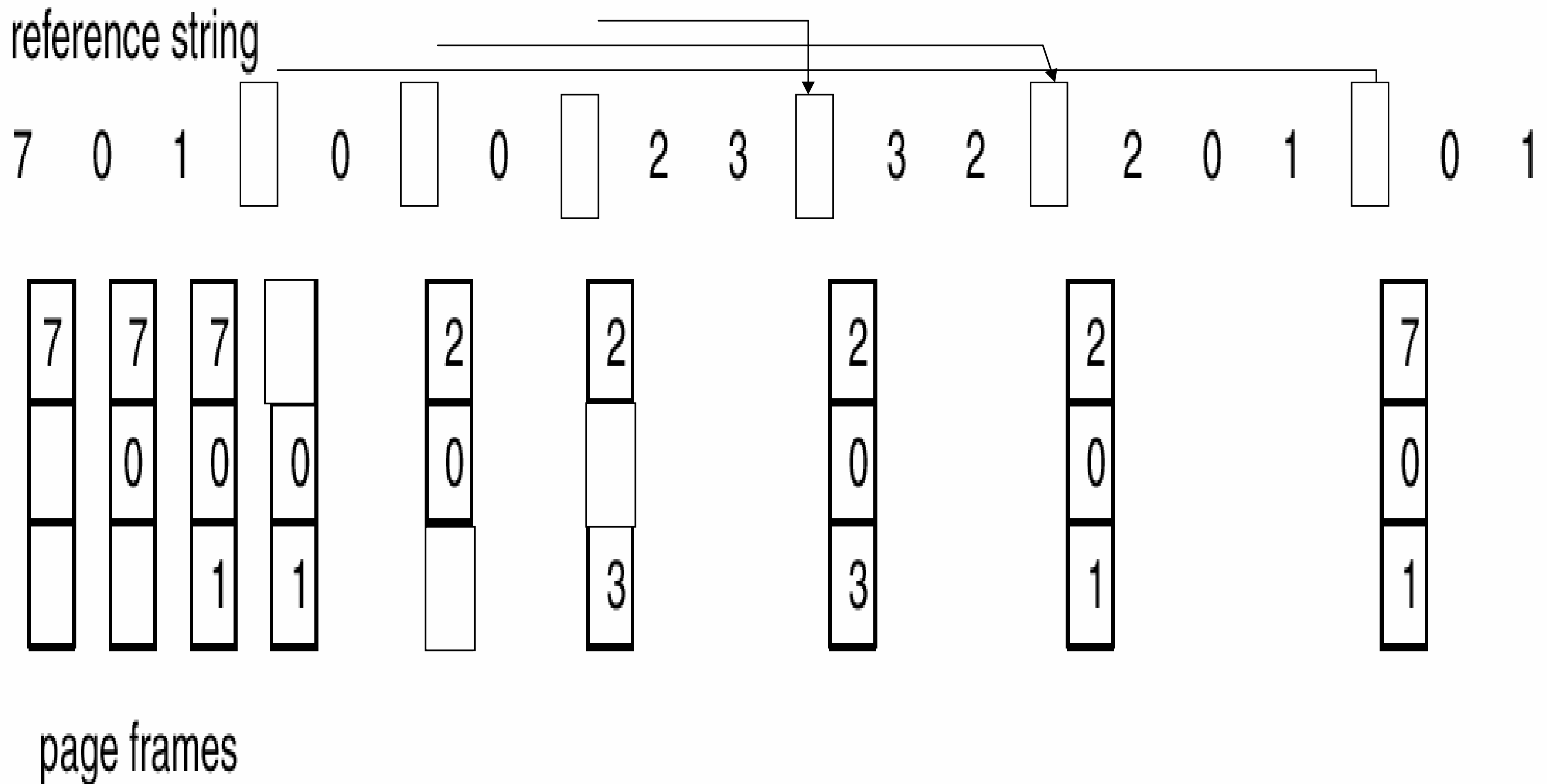
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5




6 page faults

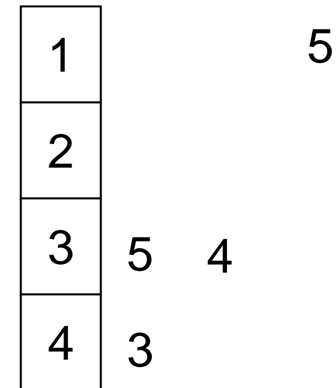
- How do we know this?
- Used for measuring how well an algorithm performs
- A baseline, we can't do better

# Optimal Page Replacement



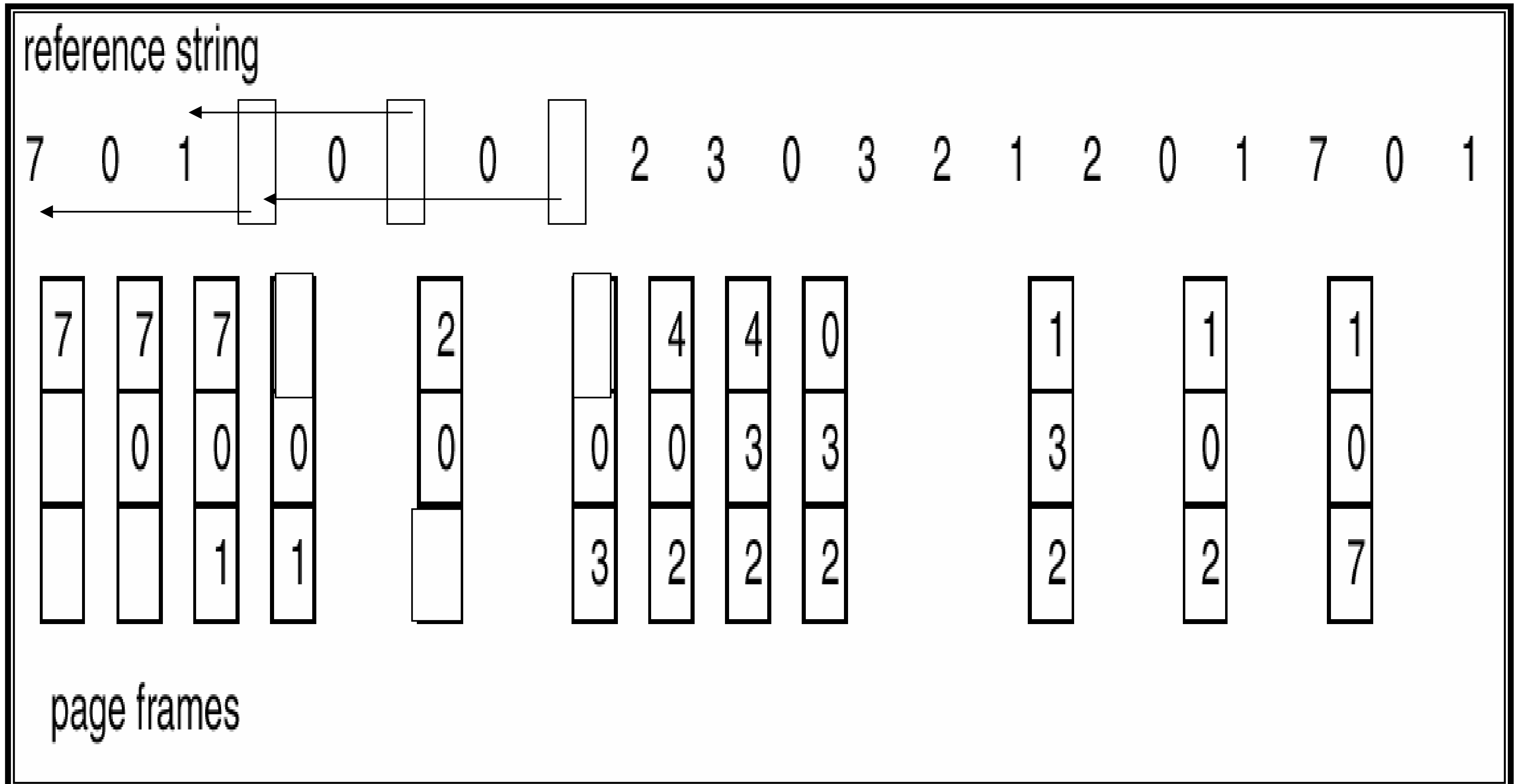
# Least Recently Used (LRU) Algorithm

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




- use recent past as approximation of near future
- Counter implementation
  - Every page table entry has a **counter**; every time page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be replaced, look at the counters to determine least recently used

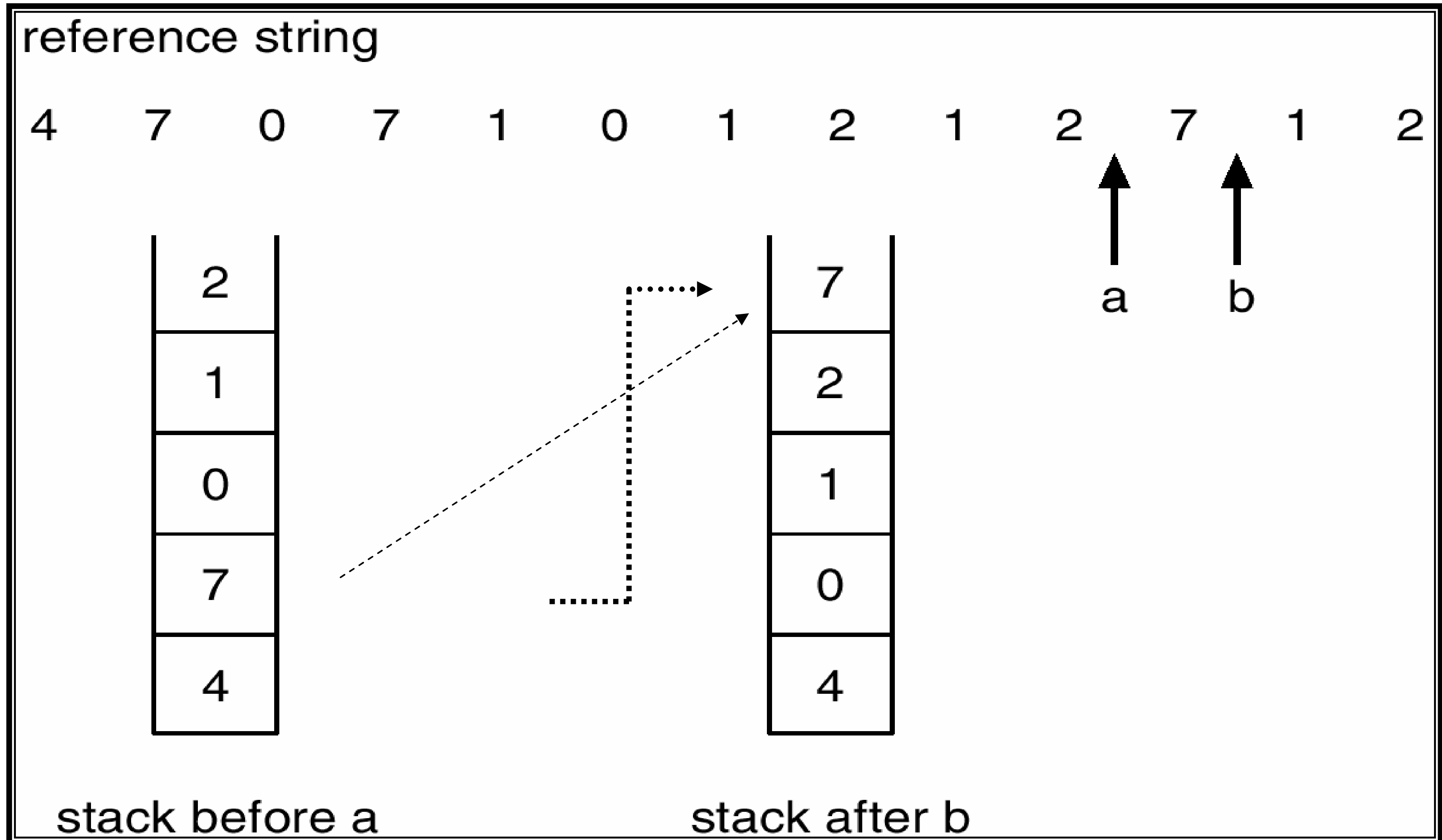
# LRU Page Replacement



# LRU Algorithm (Cont.)

- **Stack implementation** – keep a stack of page numbers in a double link form
- Page referenced
  - move it to the top
  - requires 6 pointers to be changed
  - No search for replacement

# Use Of A Stack to Record The Most Recent Page References



# LRU Approximation Algorithm 1

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

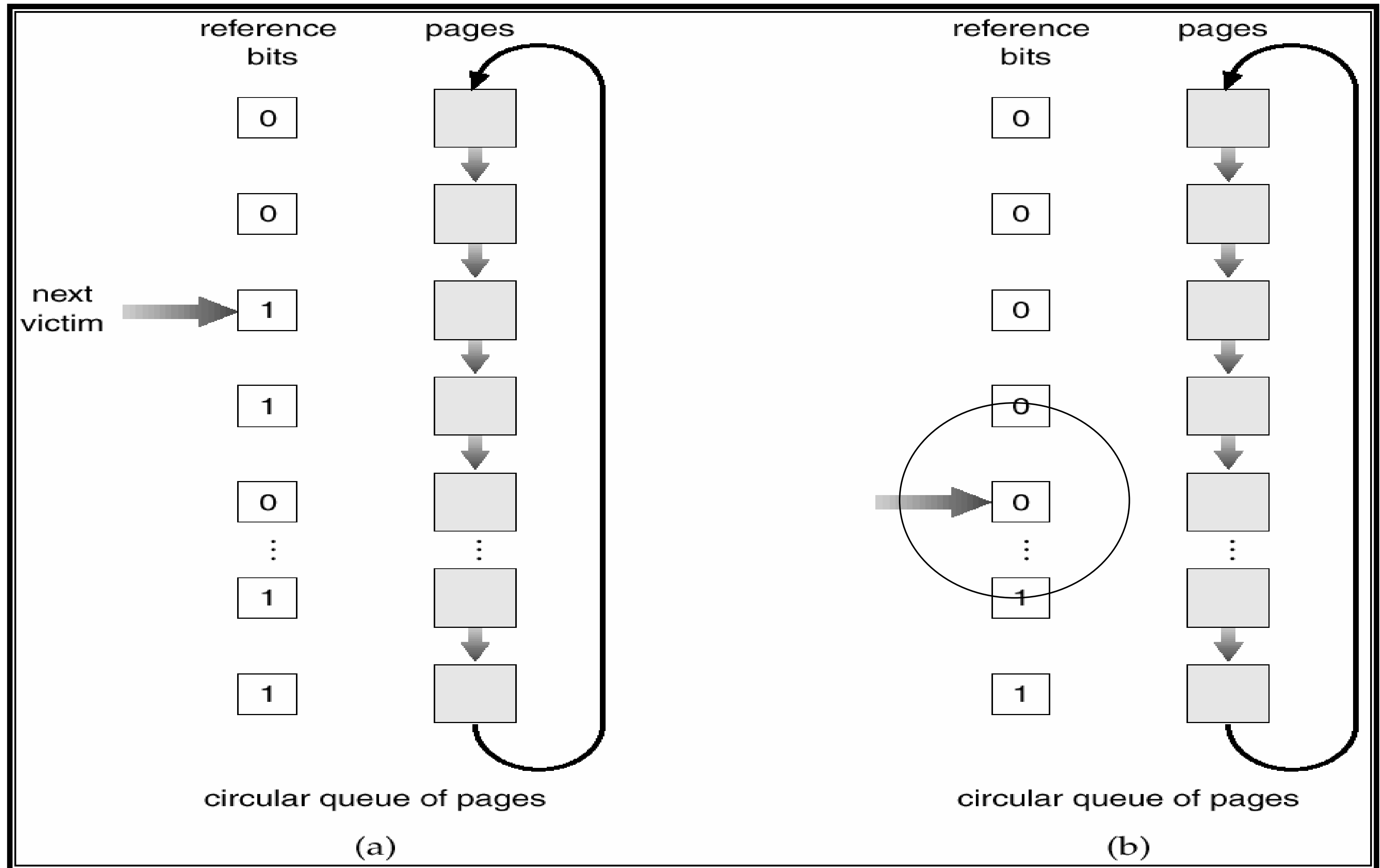
# LRU Approximation Algorithm 2

- Keep several reference bits (e.g., 8 bits) per page
- And keep the reference bit (as before)
- At periodic intervals (timer interrupt, e.g., 100 milliseconds) shift the reference bit of every page into the high-order position of the reference bit
- Right shift the reference bits, dropping low order bit
- 0000 0000 – not been used in past intervals
- 1111 1111 – has been used each in interval
- Interpret as unsigned integers, choose smallest as victim

# LRU Approximation Algorithm 3

- Second chance
  - Need 1 reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit set to 1. then:
    - set reference bit 0.
    - leave page in memory.
    - replace next page (in clock order), subject to same rules.

# Second-Chance (clock) Page-Replacement Algorithm



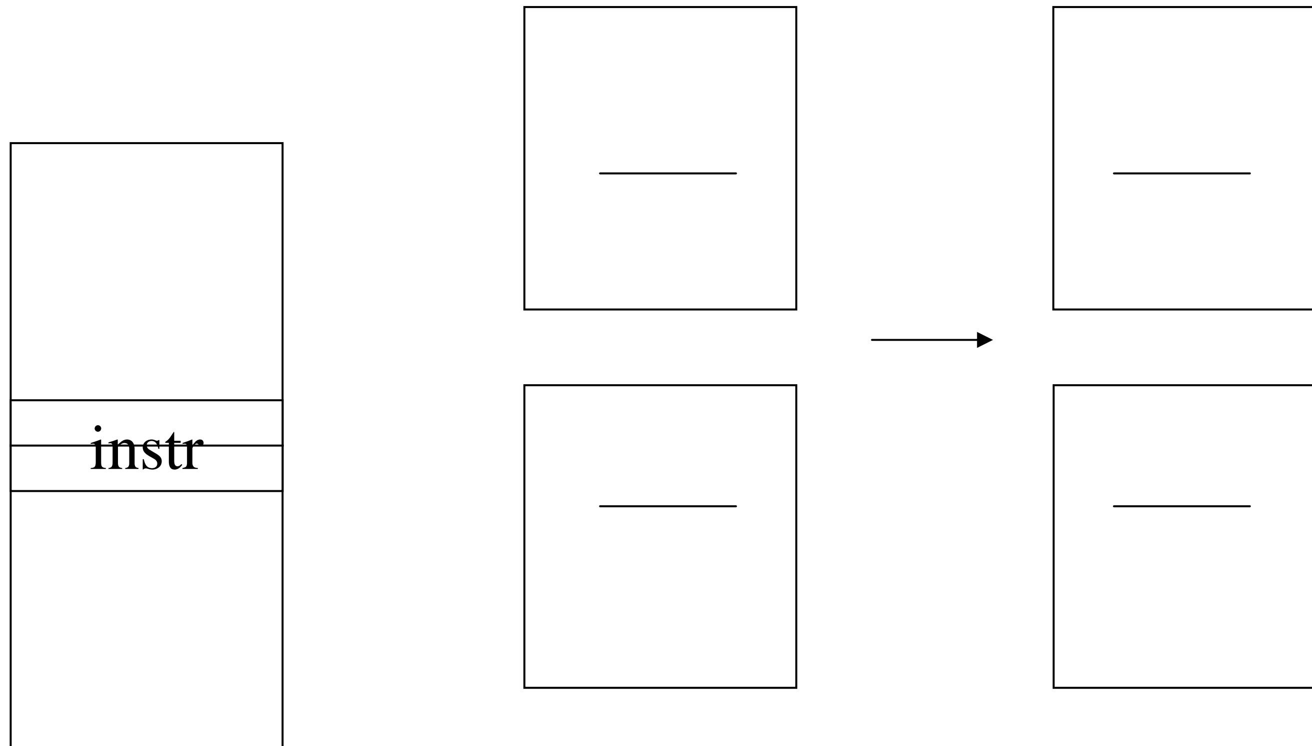
# Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU** Algorithm:  replaces page **with smallest count**
- **MFU** Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

# Allocation of Frames

- Each process needs **minimum** number of pages
- Example: IBM 370 – 6 pages to handle MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle **from**
  - 2 pages to handle **to**

# Minimum number of frames



# Fixed Allocation

- Two major allocation schemes
  - fixed allocation
  - priority allocation

- **Equal allocation** – e.g., if 100 frames and 5 processes, give each 20 pages.
- **Proportional allocation** – Allocate according to the **size** of process.

# Fixed Allocation

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

$S = \sum s_i$

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

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

Example:

$m = 64$

$s_i = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

# Priority Allocation

- Use a proportional allocation scheme using **priorities** rather than size
- If process $P_i$ generates a page fault,
  - select for replacement **one of its frames**
  - select for replacement a frame from a process with **lower priority number**

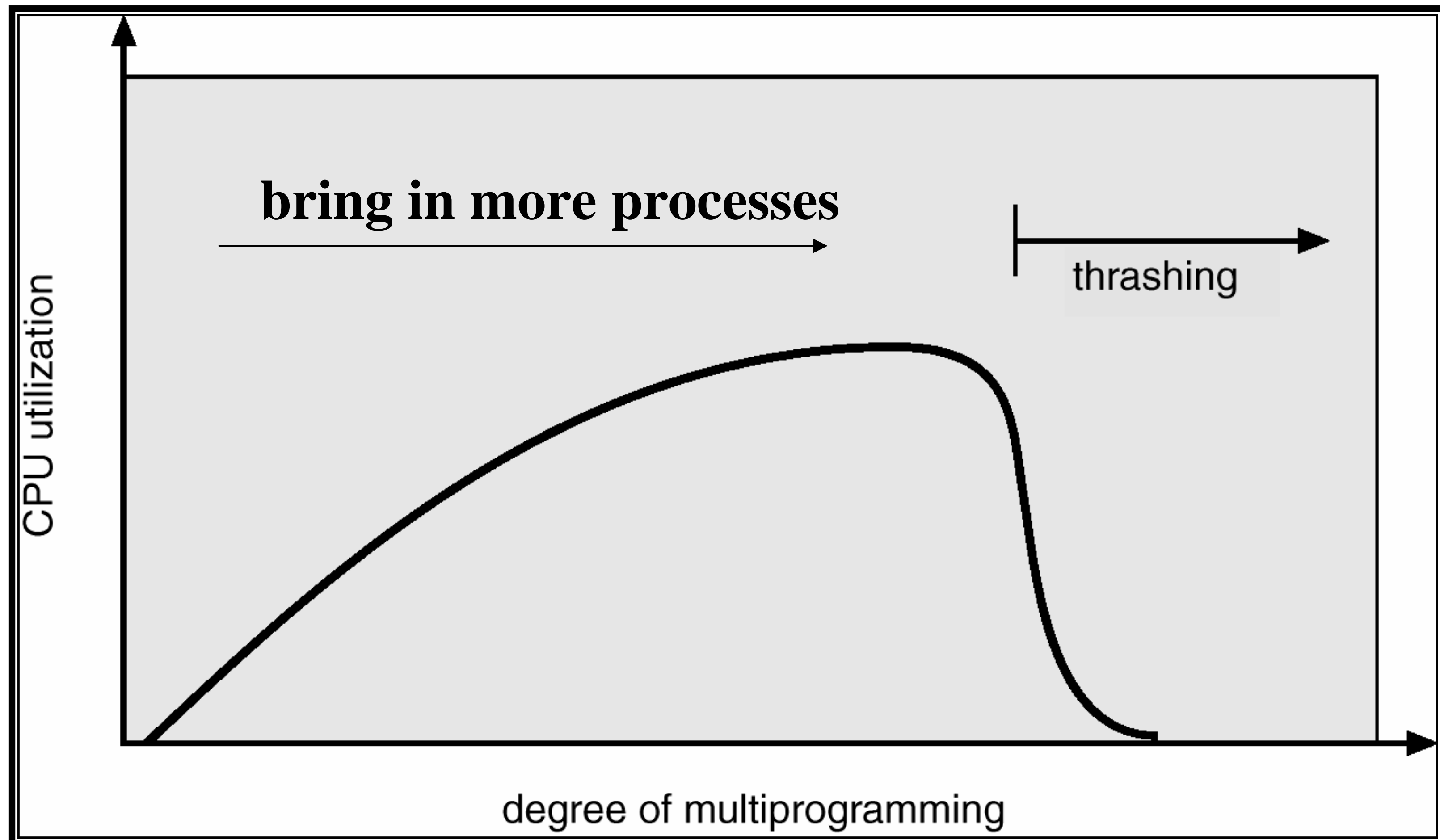
# Global vs. Local Allocation

- **Global** replacement – process selects a replacement frame **from the set of all frames**; one process can take a frame from another
- **Local** replacement – each process selects **from only its own set of allocated frames**.

# Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization (ready queue is empty)
  - operating system (may) think that it needs to increase the degree of multiprogramming
  - another process added to the system
  - this process requires pages to be brought in …
- **Thrashing** ≡ a process is busy **swapping pages in and out** (spends more time paging than executing.)

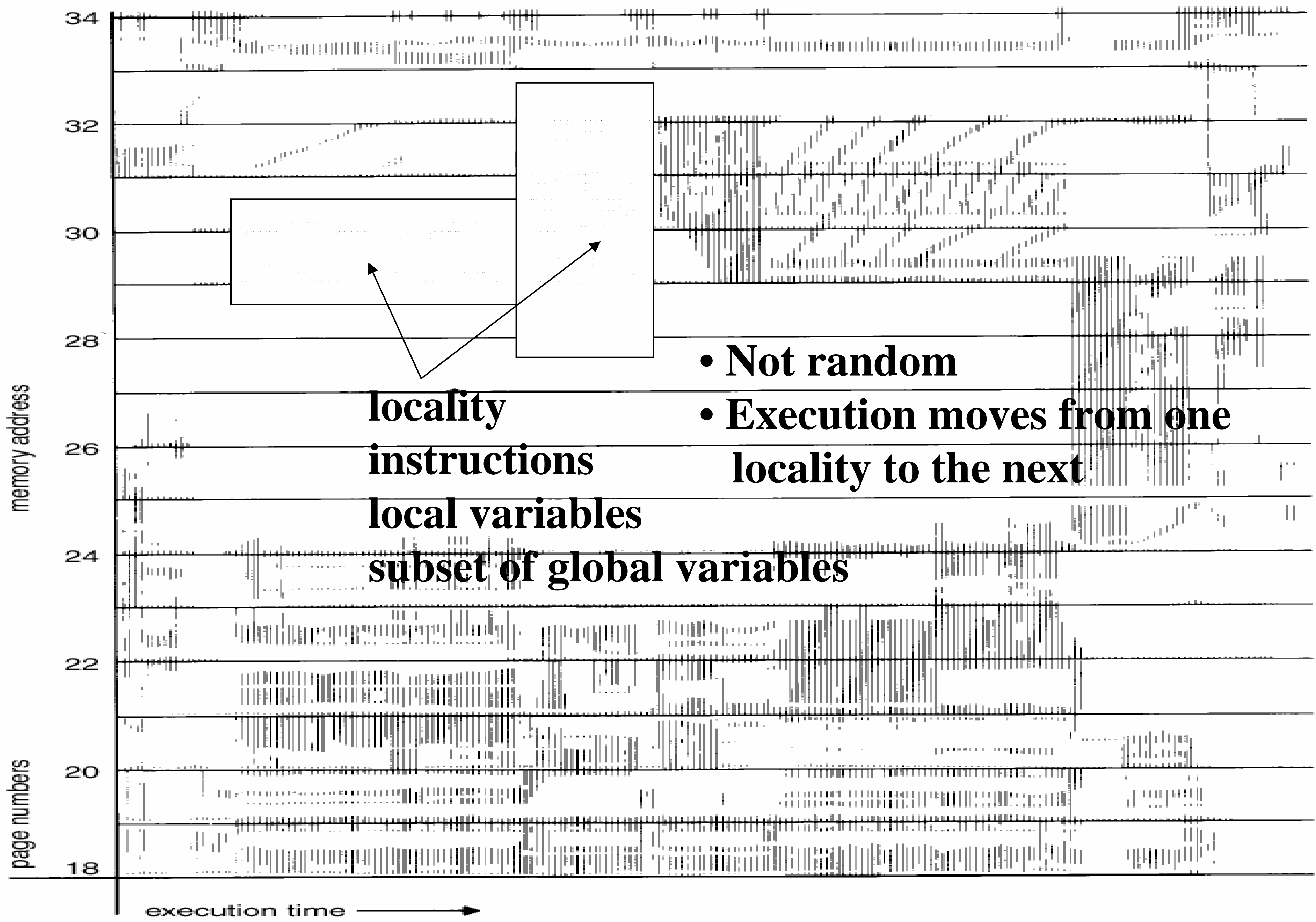
# Thrashing



# Locality

- Why does paging work?
- Due to locality (memory accesses are not random)
- Locality model
  - Process migrates from one locality to another
  - Locality corresponds to a procedure call (local variables, some global variables and instructions of procedure)
  - Localities may overlap
- Why does thrashing occur?

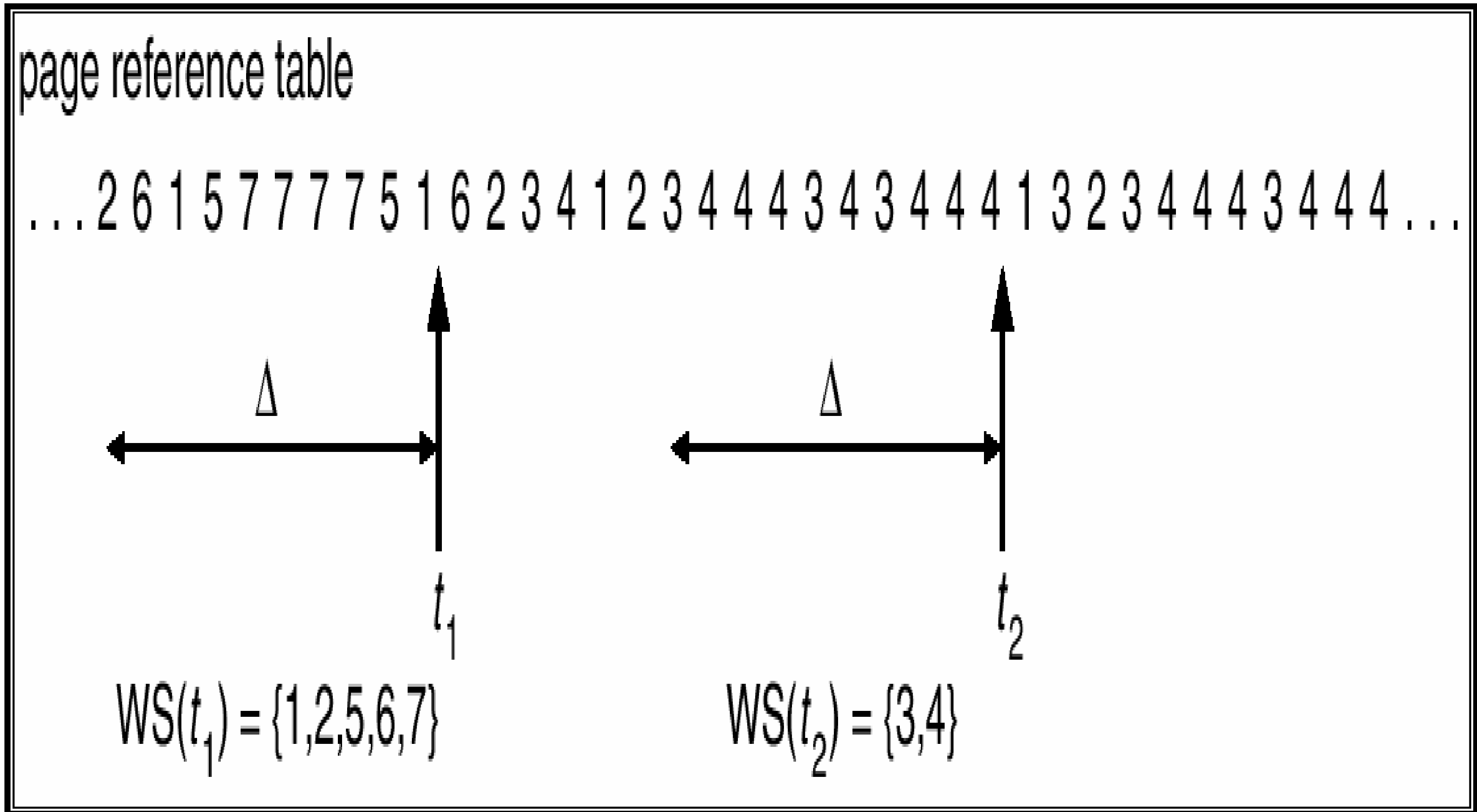
**sum over size of all localities >**
**total physical memory size**


34
32
30
28
26
24
22
20
18
memory address
page numbers
execution time
locality
instructions
local variables
subset of global variables
• Not random
• Execution moves from one locality to the next

# Working-Set Model (approximate locality)

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references. Example: 10,000 instruction
- $WSS_i$ (working set size of Process $P_i$) = total number of pages referenced in the most recent $\Delta$ (varies over time)
  - if $\Delta$ too small will not encompass entire locality.
  - if $\Delta$ too large will encompass several localities.
  - if $\Delta = \infty \Rightarrow$ will encompass entire process.
- $D = \Sigma\ WSS_i \equiv$ total frames demanded
- if $D > m \Rightarrow$ Thrashing (m is total physical memory)
- Policy if $D >$ m, then suspend one of the processes.

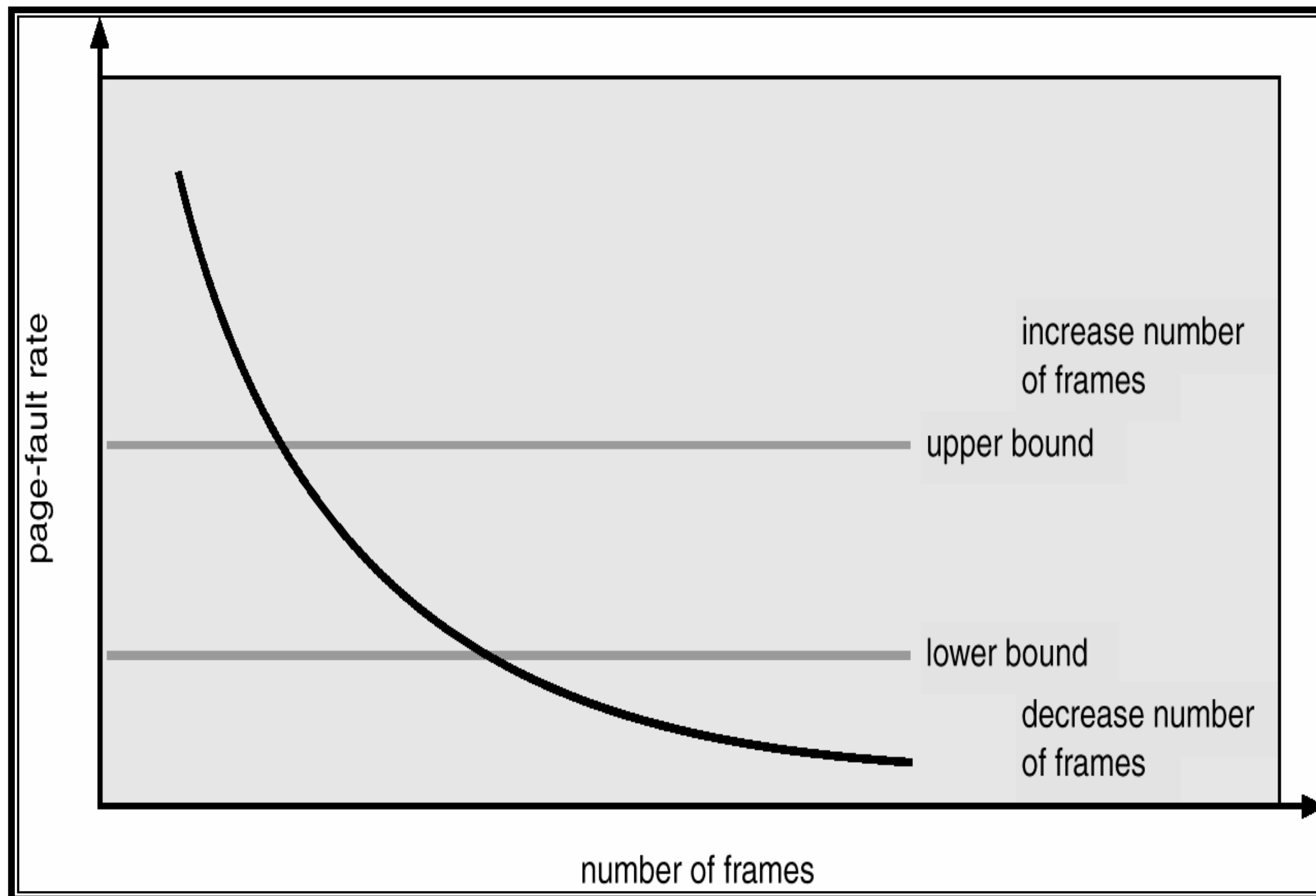
# Working-set model



# Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts copy reference bit to memory bits and sets the values of all reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units (cost of interrupt!).

# Page-Fault Frequency Scheme



# Summary Memory Management

- Contiguous memory management
- Paging and segmentation
- Virtual memory management based on demand paging
- Page replacement algorithm
- Frame allocation strategies
- Thrashing
- Locality and working set model

# OS Lecture

- Concepts and OS hacking
- Processes and Threads
- OS System Structure and Architecture
- Synchronization
  - Software based solutions
  - Hardware based solutions
  - Semaphores, mutexes/locks, CVs, monitors
  - Synchronization problems
- Scheduling algorithm
- Memory management

# Assignments

- **Tools**: CVS, GDB, GCC
- **Adding a delta to** a large and complex **software system**
  - Not much know methodology about how to do this (but see software engineering course)
  - Don't be afraid of the size; **work with a localized understanding** of system; 20K lines of code is nothing compared to the size of real OS, DBs, …
- **Making design decision** which great reach (actually making the decision is difficult)
- Implementation of **synchronization mechanisms**
- **Use** of **synchronization mechanisms**
- **Implementation of system calls** (not just a procedure call)
- Implementation of **scheduling algorithms** and **performance counters**
- OS and Systems is about hacking; that is **building and extending large complex software systems**

# The Final

- Closed book
- Covers entire lecture and assignments
- **Rough** breakdown of final, *don't quote me*
  - 20 – 30 % knowledge questions a la midterm
  - 10 – 20 % about assignments
  - 20 – 30 % synchronization
  - 10 – 20 % memory management
  - Rest other course topics

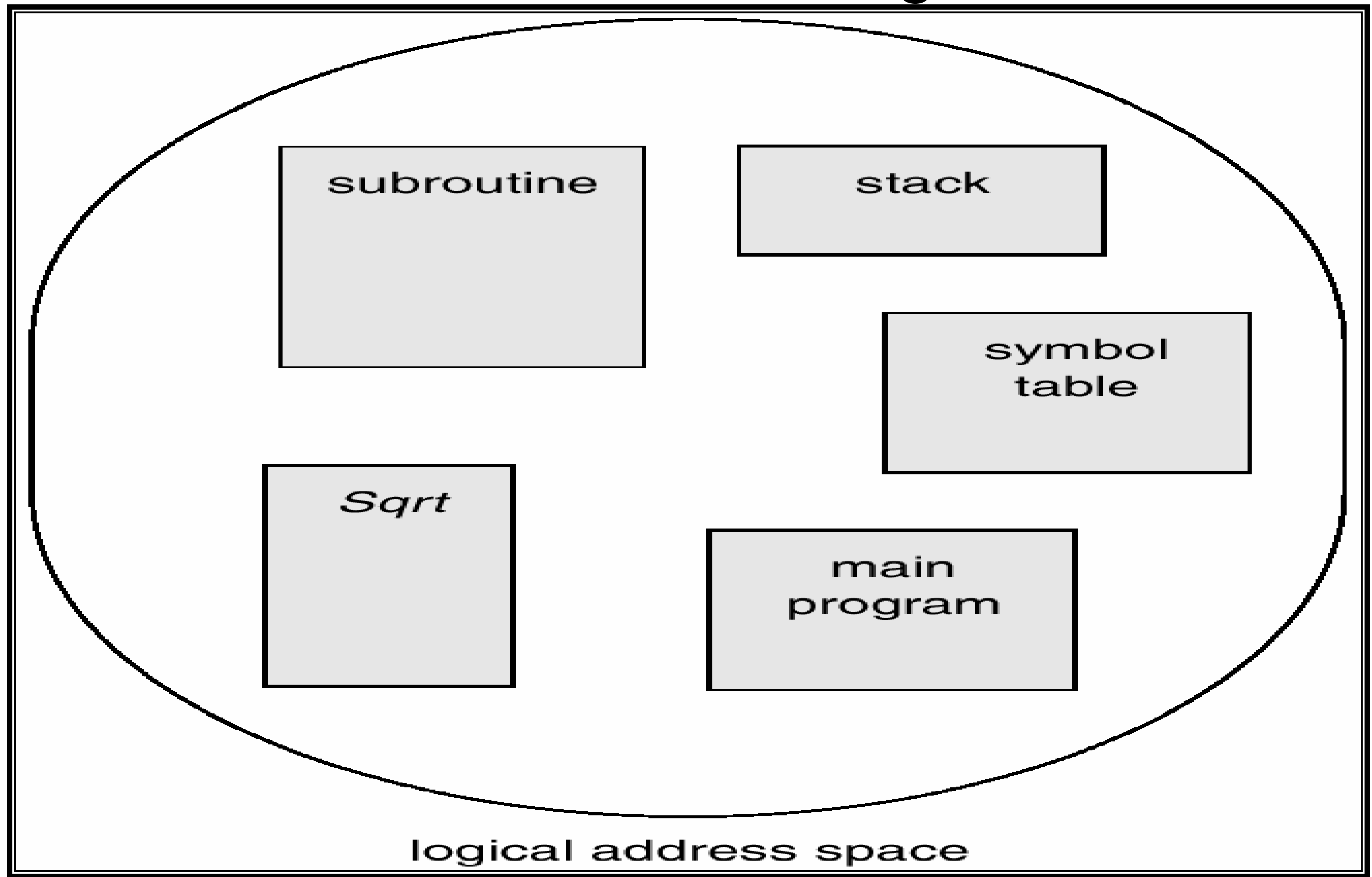
# The End

# Segmentation

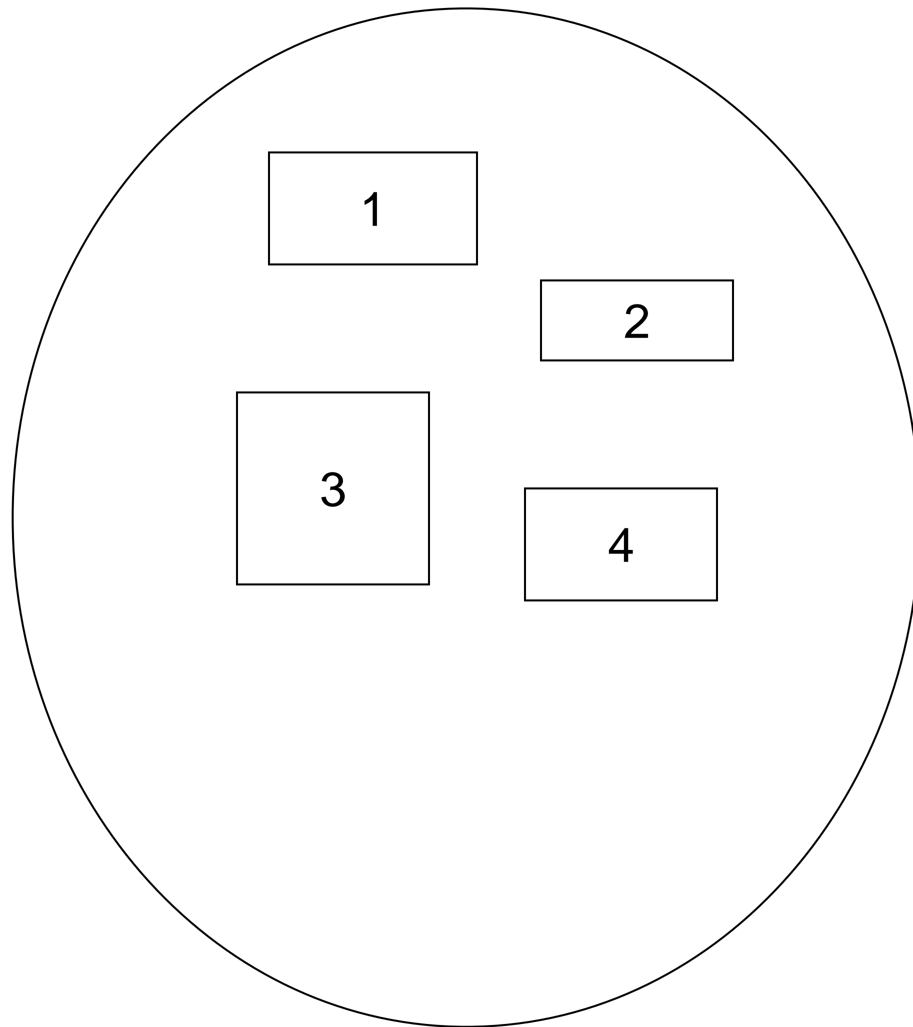
- Memory-management scheme that supports user's view of memory.
- A program is a **collection of segments**. A segment is **a logical unit** such as:

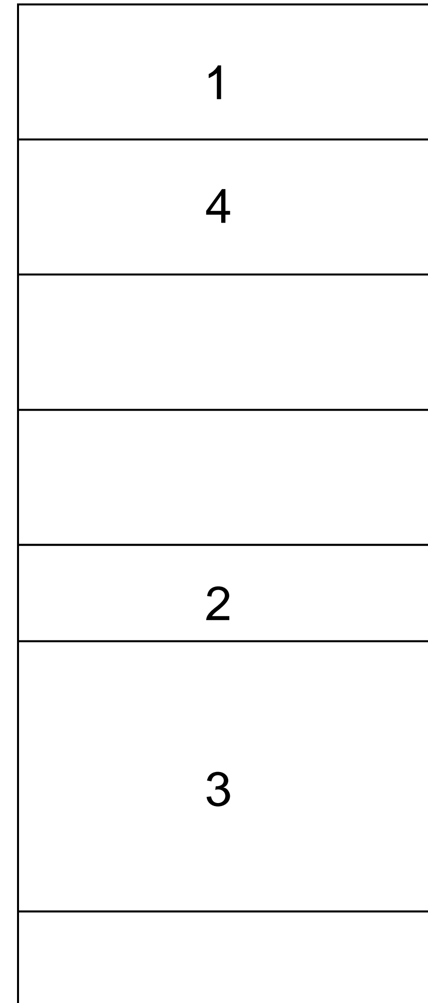
main program,
procedure,
function,
method,
object,
local variables, global variables,
common block,
stack,
symbol table, arrays

# User's View of a Program



# Logical View of Segmentation



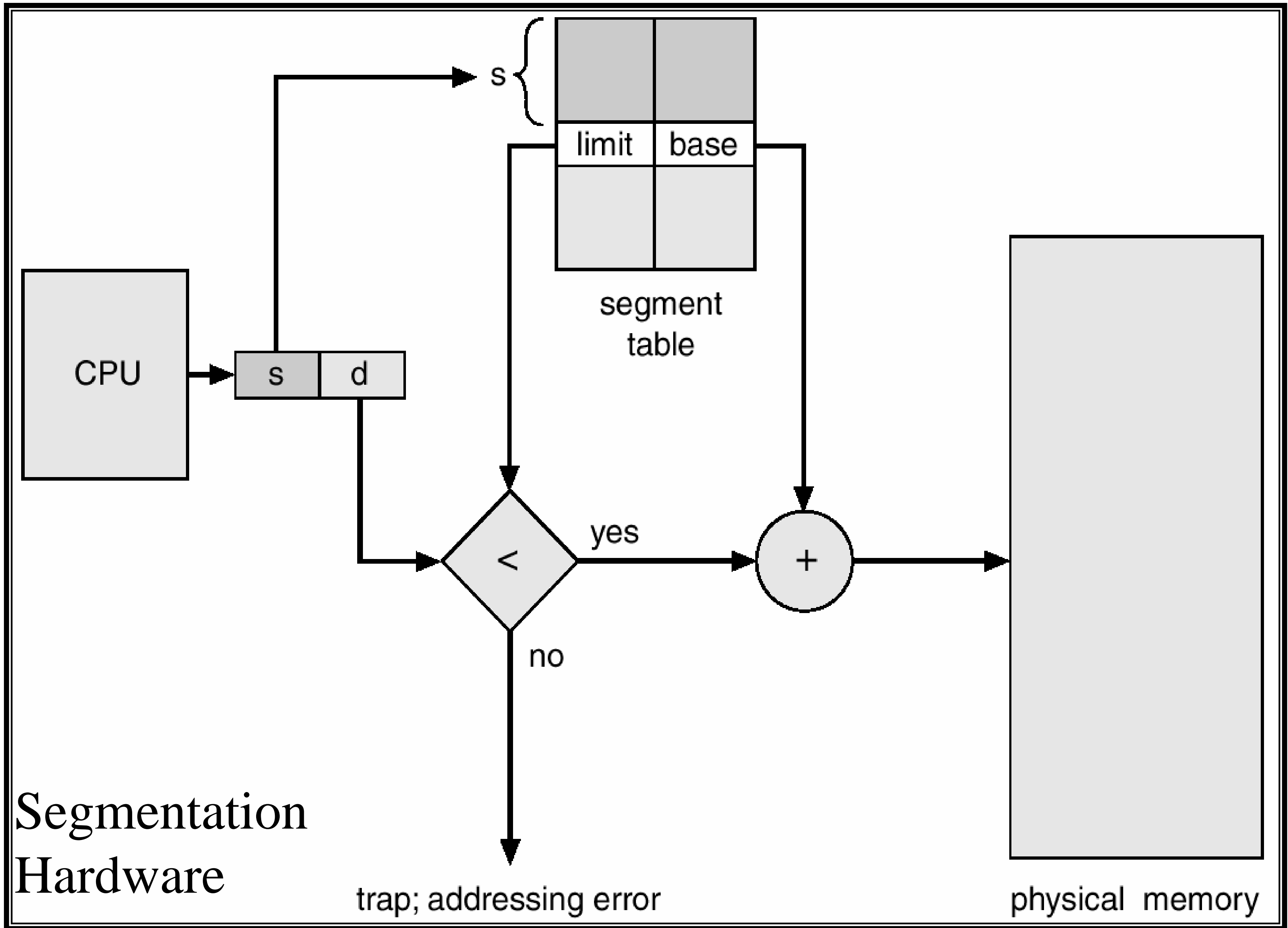

user space

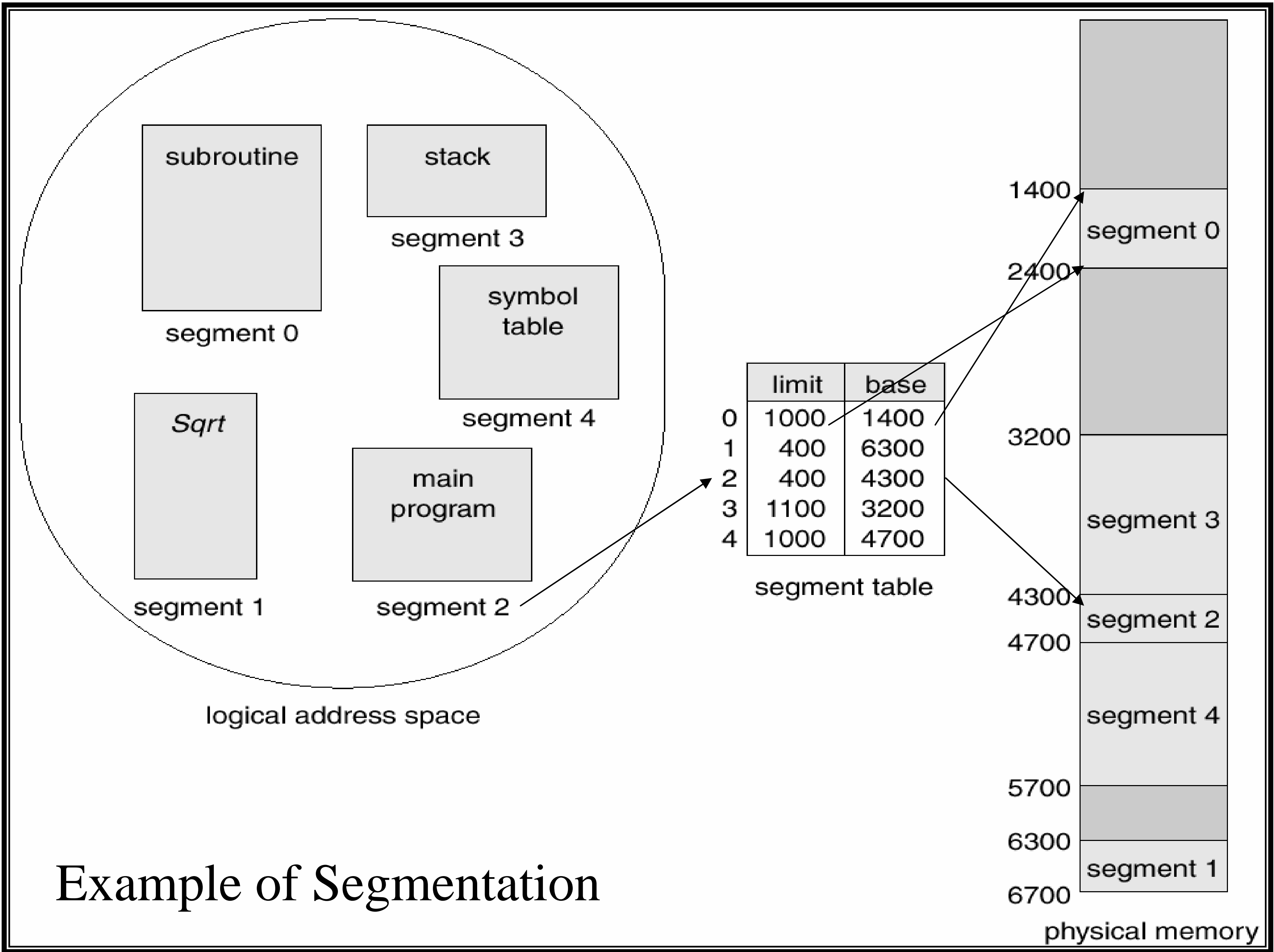
physical memory space

# Segmentation Architecture

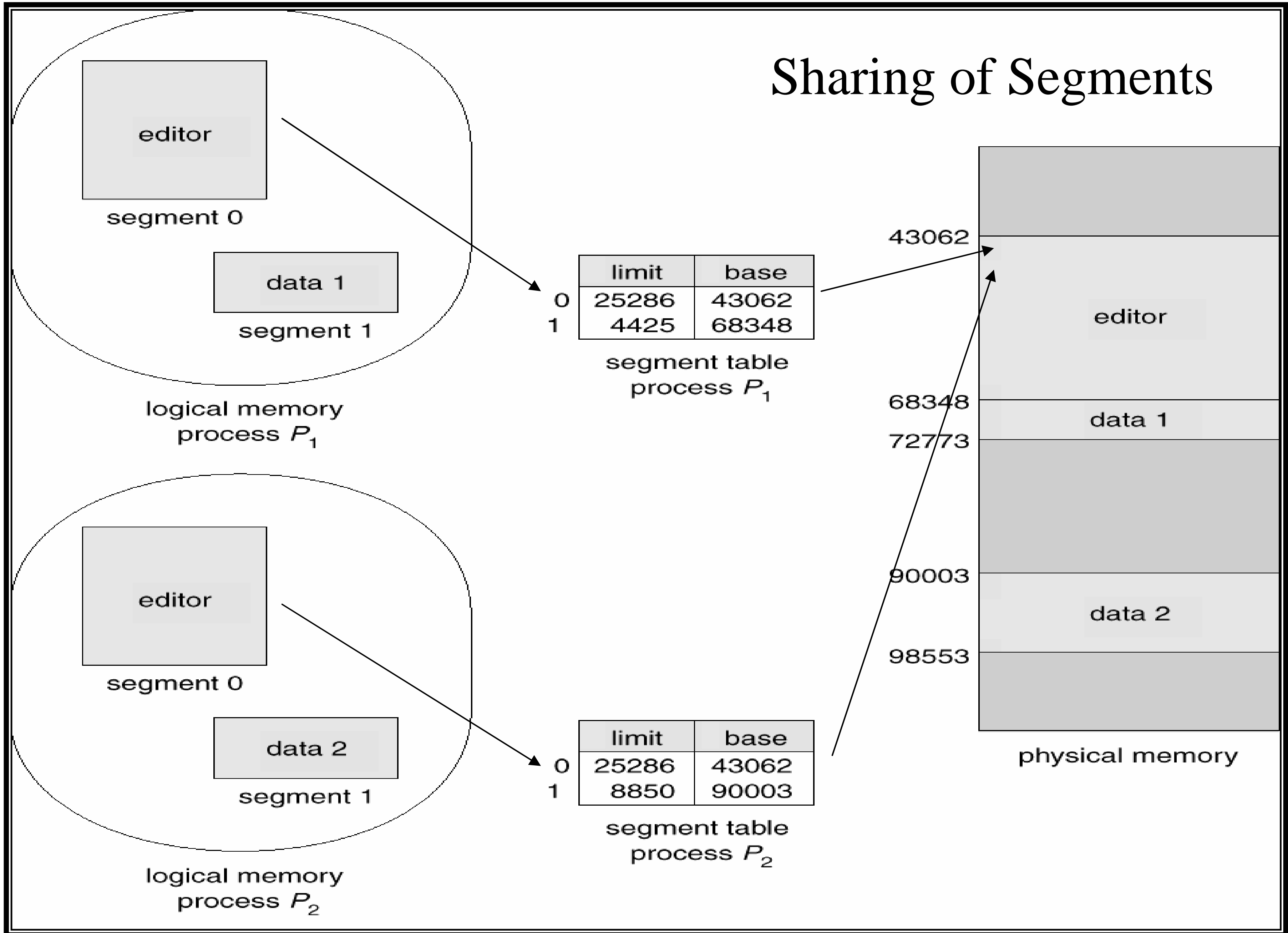
- Logical address consists of a two tuple:

  **<segment-number, offset>,**
- *Segment table* – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory.
  - **limit** – specifies the length of the segment.
- ***Segment-table base register*** *(STBR)* points to the segment table's location in memory.
- ***Segment-table length register*** *(STLR)* indicates **number of segments** used by a program;


s
limit
base
segment
table
CPU
s
d
<
yes
+
no
trap; addressing error
physical memory
Segmentation
Hardware


subroutine
stack
segment 3
symbol
table
segment 0
segment 4
Sqrt
main
program
segment 1
segment 2
logical address space
limit
base
0 1000 1400
1 400 6300
2 400 4300
3 1100 3200
4 1000 4700
segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
physical memory
Example of Segmentation

# Sharing of Segments



# Shared Memory

# Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes.
- Private code and data
  - Each process keeps a separate copy of the code and data.
  - The pages for the private code and data can appear anywhere in the logical address space.

# Shared Pages Example