Chapter 5: Memory Management

Sukomal Pal, CSE, IIT(BHU)

Chapter 5. Memory Management

- *Memory Management: Basic concept, Logical and Physical address map, Memory allocation: Contiguous Memory allocation – Fixed and variable partition– Internal and External fragmentation and Compaction; Paging: Principle of operation – Page allocation – Hardware support for paging, Protection and sharing, Disadvantages of paging.*
- *Virtual Memory: Basics of Virtual Memory – Hardware and control structures – Locality of reference, Page fault, Working Set, Dirty page/Dirty bit – Demand paging, Page Replacement algorithms: Optimal, First in First Out (FIFO), Second Chance (SC), Not recently used (NRU) and Least Recently used (LRU).*

INTRODUCTION

- Memory is the second most important resource after the processor in a computer.
- How many programs can be accommodated in the memory decides the *degree of multiprogramming.*
- Can a process be partially loaded?
- when and where is the remaining portion to be loaded later ?
- Some of the critical questions related to this Memory management:
  1. How many processes are to be loaded in the memory?
  2. Which processes will be allocated space?
  3. How much space will be allocated to a given process and in what way?

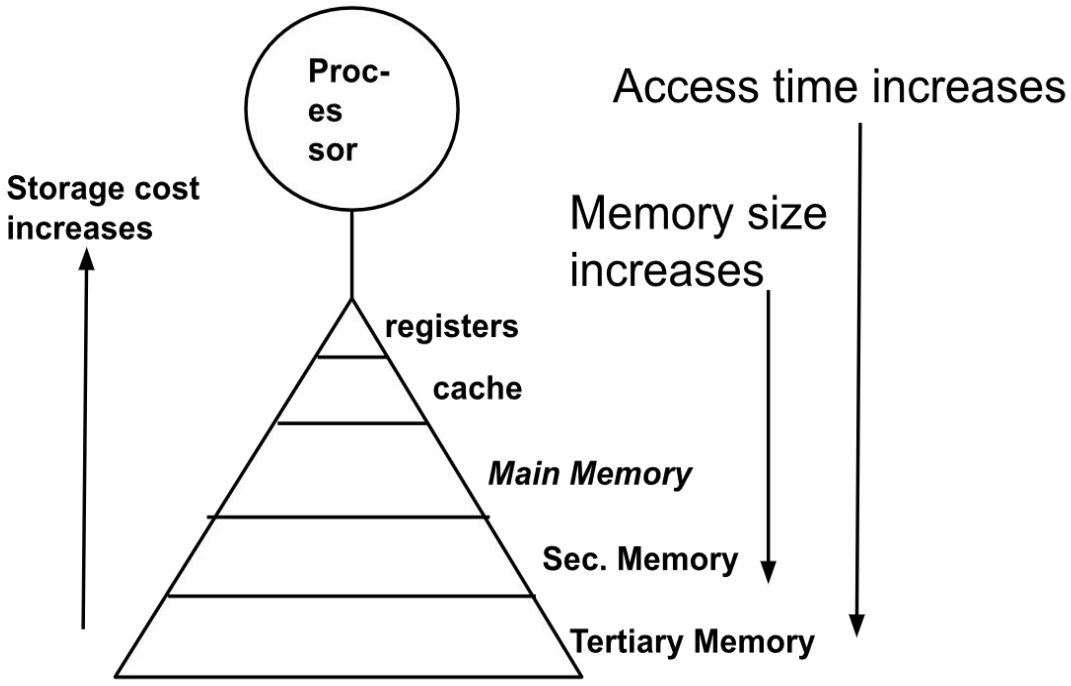


Fig 5.1: Hierarchical Memory organization

Basic hardware and software

- Memory is a linear array of memory elements (bits).
- 8 bits = 1 byte, memory is organized in MBs/GBs.
- Each process has:
  - **Base address** (start of memory segment)
  - **Limit register** (length of memory segment)
- Valid address = base ≤ address ≤ base + limit
- Example:
  - Base address: 1204
  - Limit: 476 → End address = 1204 + 476 = **1680**
  - Access to any location outside 1204 – 1680 is illegal.

Basic hardware and software

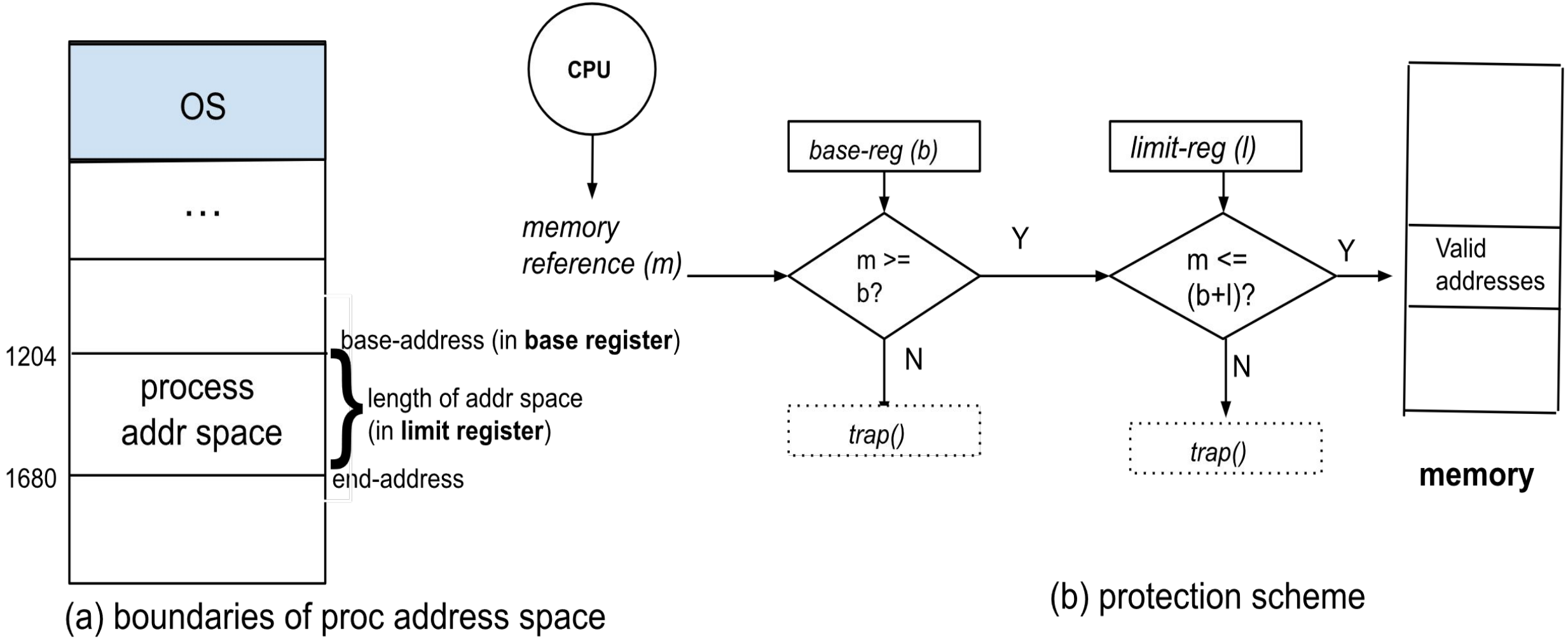


Fig 5.2: Memory access and its protection scheme

Address binding

- CPU does **not** directly generate absolute addresses.
- Machine instructions contain **relative addresses**.
- Real addresses are calculated later through **address binding**.
- Address binding is divided into three categories based on the time of its occurrence:

1. **Compile-time address binding**
2. **Load-time address binding**
3. **Run-time address binding**

Address binding

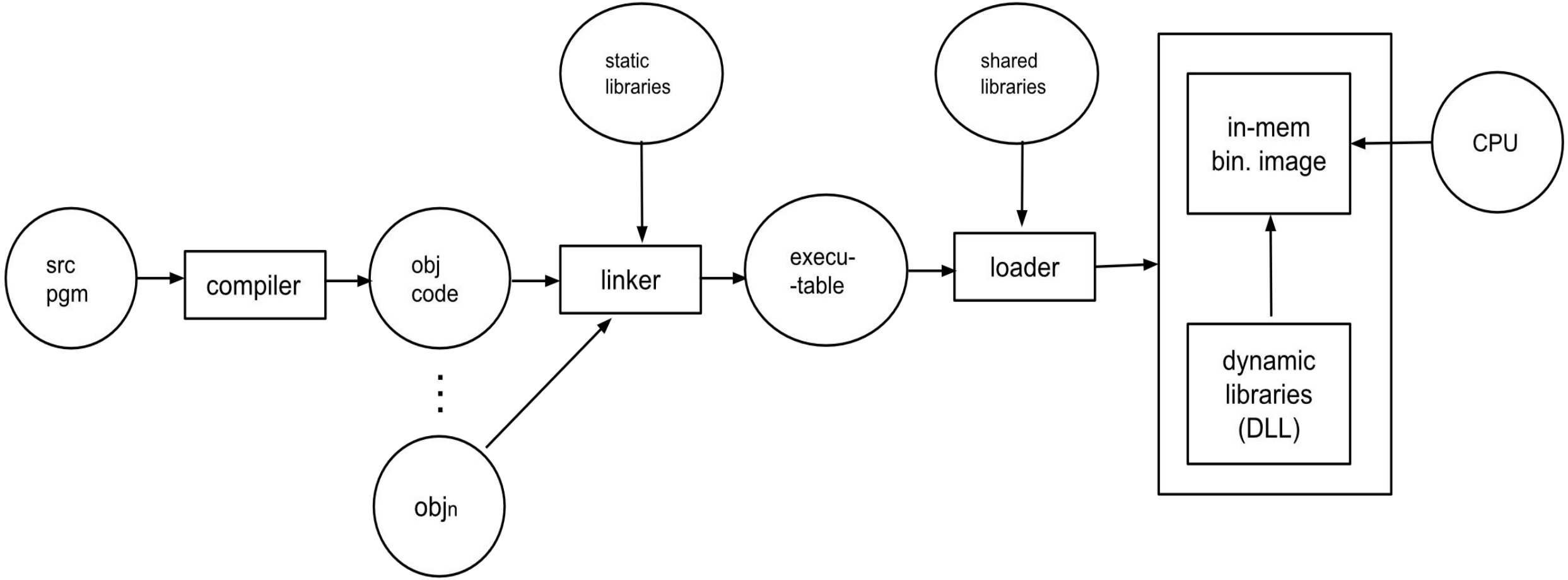


Fig 5.3: Different steps of a user program (from source code to in-memory executable)

Logical vs Physical Addresses

- **Logical Address (Virtual Address):**
  - Generated by the CPU during program execution.
  - Exists in the **logical address space**.
  - Based on **relocatable** or **relative addressing**.
- **Physical Address:**
  - Actual address in main memory (RAM).
  - Used by the memory unit to access data.
- There is a one-to-one mapping between identifiers, their logical addresses and physical addresses.

Logical vs Physical Addresses

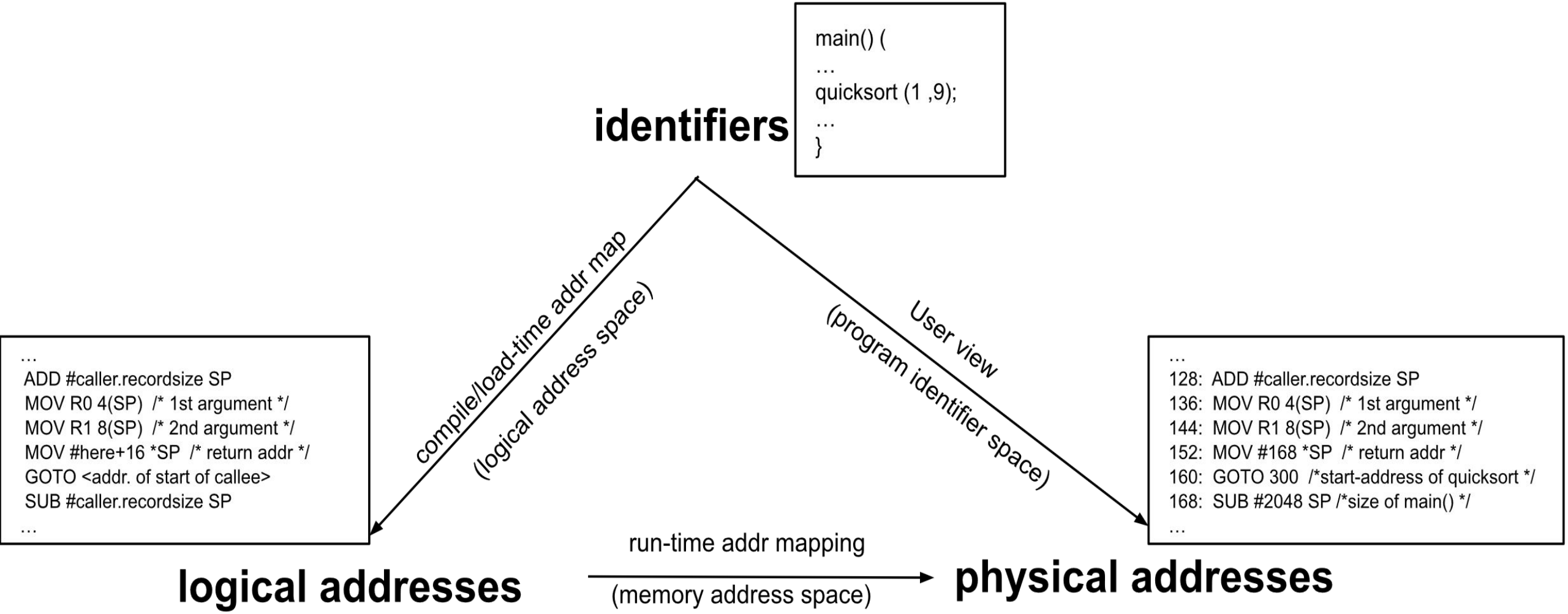


Fig 5.4: Different address spaces and address mapping

Logical vs Physical Addresses

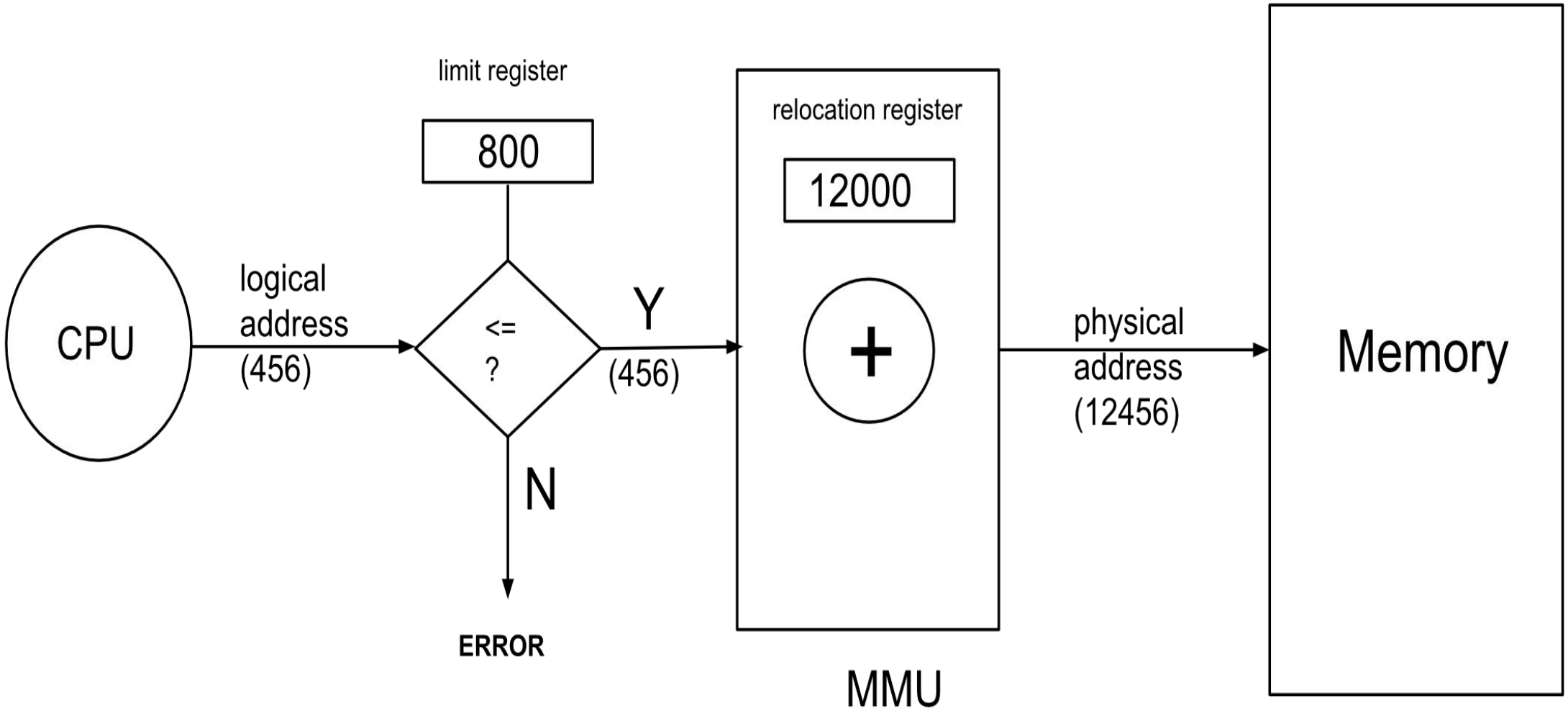


Fig 5.5: Simplest run-time address translation through a MMU

MEMORY ALLOCATION

- Memory allocation is the primary task in memory management.
- Main memory is the largest memory element that CPUs can directly access.
- a large program , requiring more than available memory, cannot be loaded entirely.
- **Overlaying:** When another module needed to be loaded, it used to replace the in-memory counterpart from the secondary memory.

Basic allocation schemes:

1. ***Contiguous Allocation*:** a process in its entirety is allocated contiguous memory space
2. ***Paging*:** a process is divided into a number of equal sized pages; pages are loaded
3. ***Segmentation*:** a process is divided into a number of unequal-sized segments; segments are loaded.

Contiguous Allocation

- A process in its entirety is loaded into the memory at one go.
- The OS occupies a fixed portion of the memory at one of the ends.
- The OS divides the available space into a number of partitions.
- OS allocates at least one partition to each process in a contiguous manner.
- Partitioning can be done using fixed boundaries or adjustable boundaries.
- **Internal fragmentation** occurs when small processes waste partition space.

Fixed Partitioning

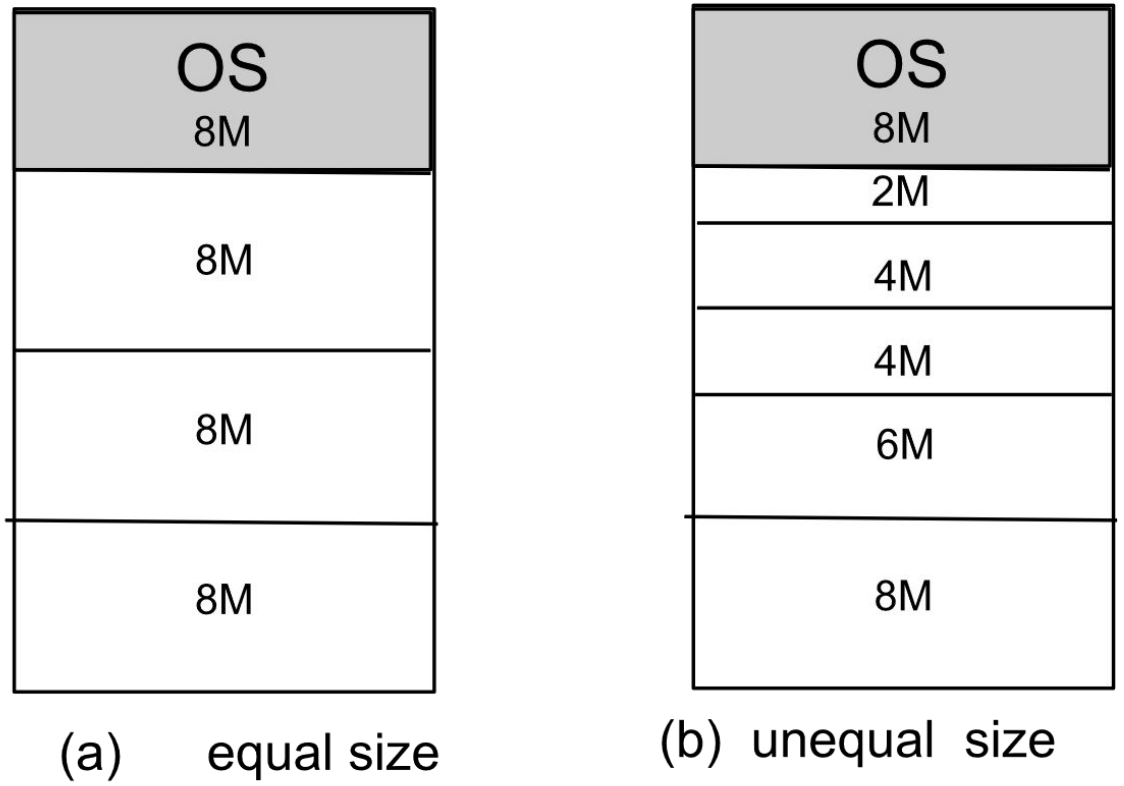


Fig 5.6: Two types of Fixed Partitioning in 32M of memory

Variable Partitioning

- Partitioning done at the time of space allocation.
- process of smaller size replaces an old process a ‘hole’ is created in memory.
- The partition gets smaller, and the fragmentation is outside the partition. This is called **external fragmentation**.

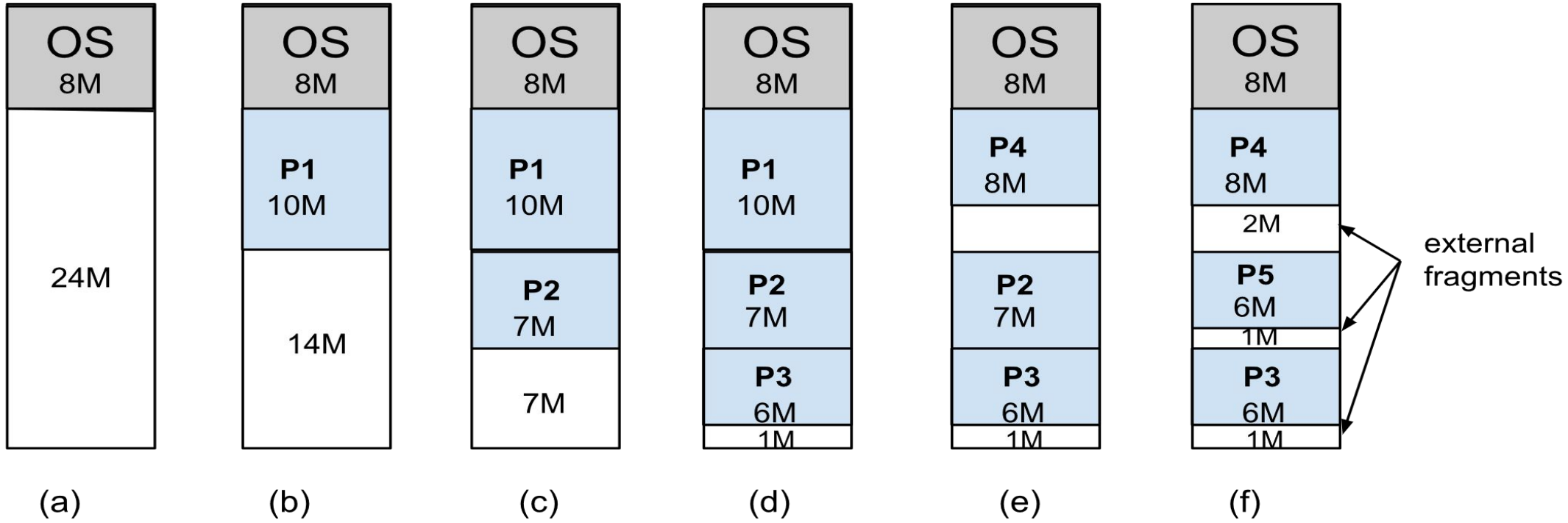


Fig 5.7: Dynamic partitioning and external fragmentation in 32 MB of memory

Handling External Fragmentation

- **Compaction**:
  - Push processes together to form larger contiguous holes.
  - Time-consuming; involves moving live processes.
- **Placement Algorithms**:
  - Optimize allocation at the time of placement.

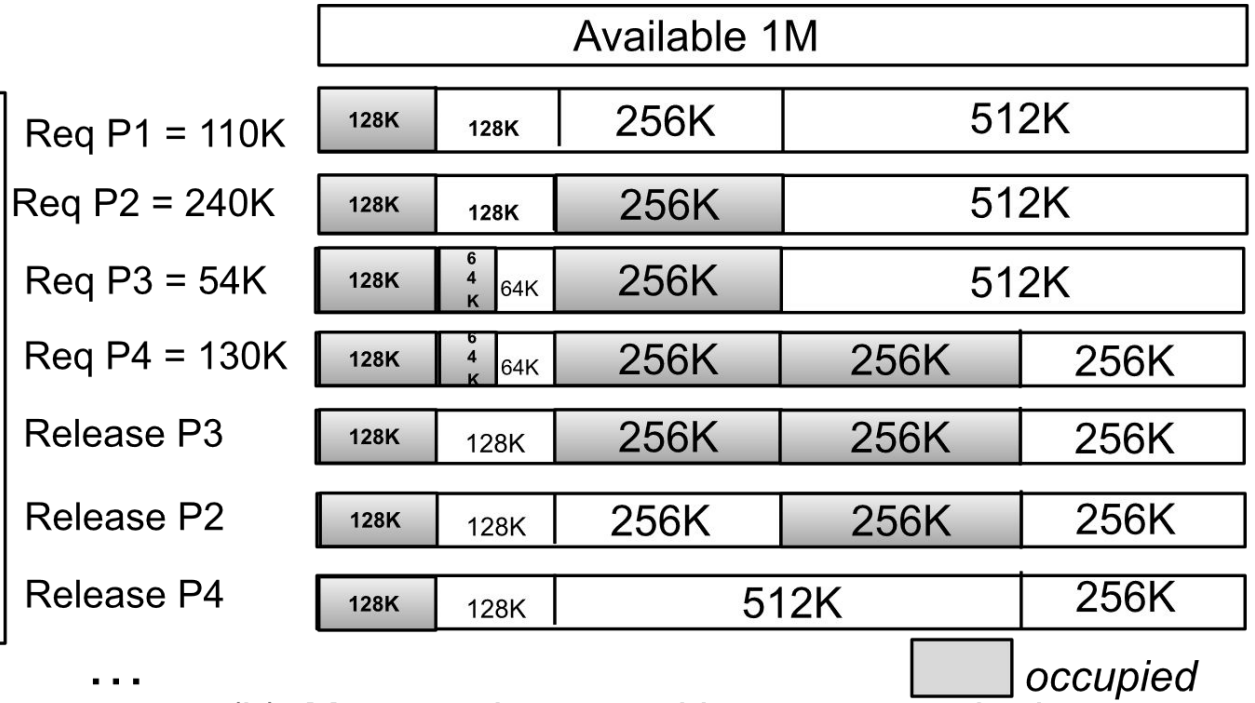
Strategy	Description	Pros	Cons
First-fit	First large enough hole	Fast	May cause early fragmentation
Best-fit	Smallest possible hole	Less wasted space	Time-consuming, often poor overall performance
Worst-fit	Largest hole available	Leaves larger holes	May waste space

Buddy System

- Combines ideas of **fixed and dynamic partitioning**.
- Allocates memory in **powers of 2**:
- Block sizes = 2^i (L ≤ i ≤ U)
- 2^L = smallest block
- 2^U = total memory

```
void alloc_hole(int i){
  if (i > U) error;
  if (list_i == NULL) {
     alloc_hole(i+1);
     <split hole into two>;
     <put buddies on list_i>;
  }
  <take 1 hole from list_i>;
}
```

(a) Algorithm



(b) Memory changes with requests and releases

Fig 5.8: Buddy System

Paging

- Memory is divided into **frames**, and program code/data into **pages**, both of the same size (usually powers of 2).
- Processes are loaded in units of pages into the available frames.
- The frames need not be allocated contiguously in the memory.
- **Logical address :** <page#, offset> , **Physical Address:** <frame#, offset>
- Given a page (p), the page-table returns the frame-id (f) where its stored.
- A *page-table* (PT) maintains a mapping between pages and frames.

Paging

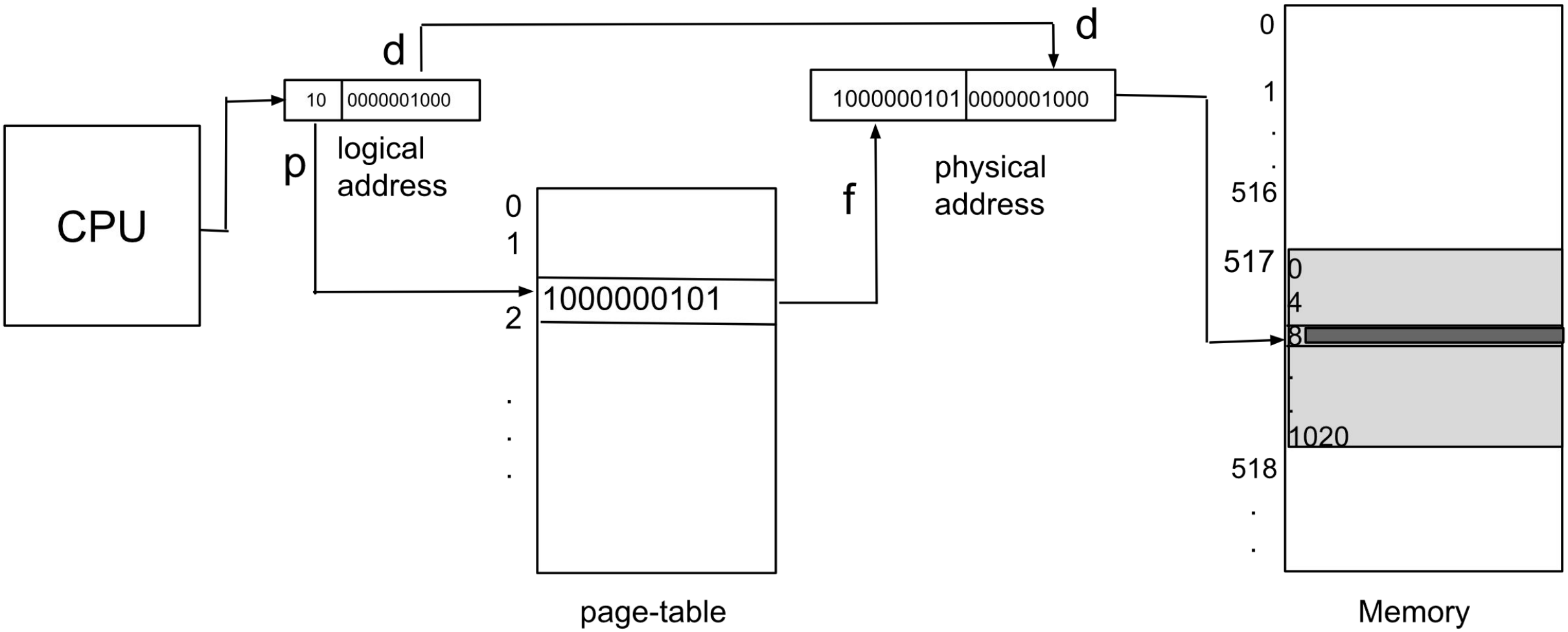


Fig 5.9: Address translation in paging using a page-table

Page Allocation

- **Page Table**: One per process.
- **Frame Table**: Global, tracks which frames are in use.
- When a process is no more active, its page table is also destroyed
- The allocation and release keep on occurring dynamically in a system.
- the pages belonging to a process thus are unlikely to get contiguous frames

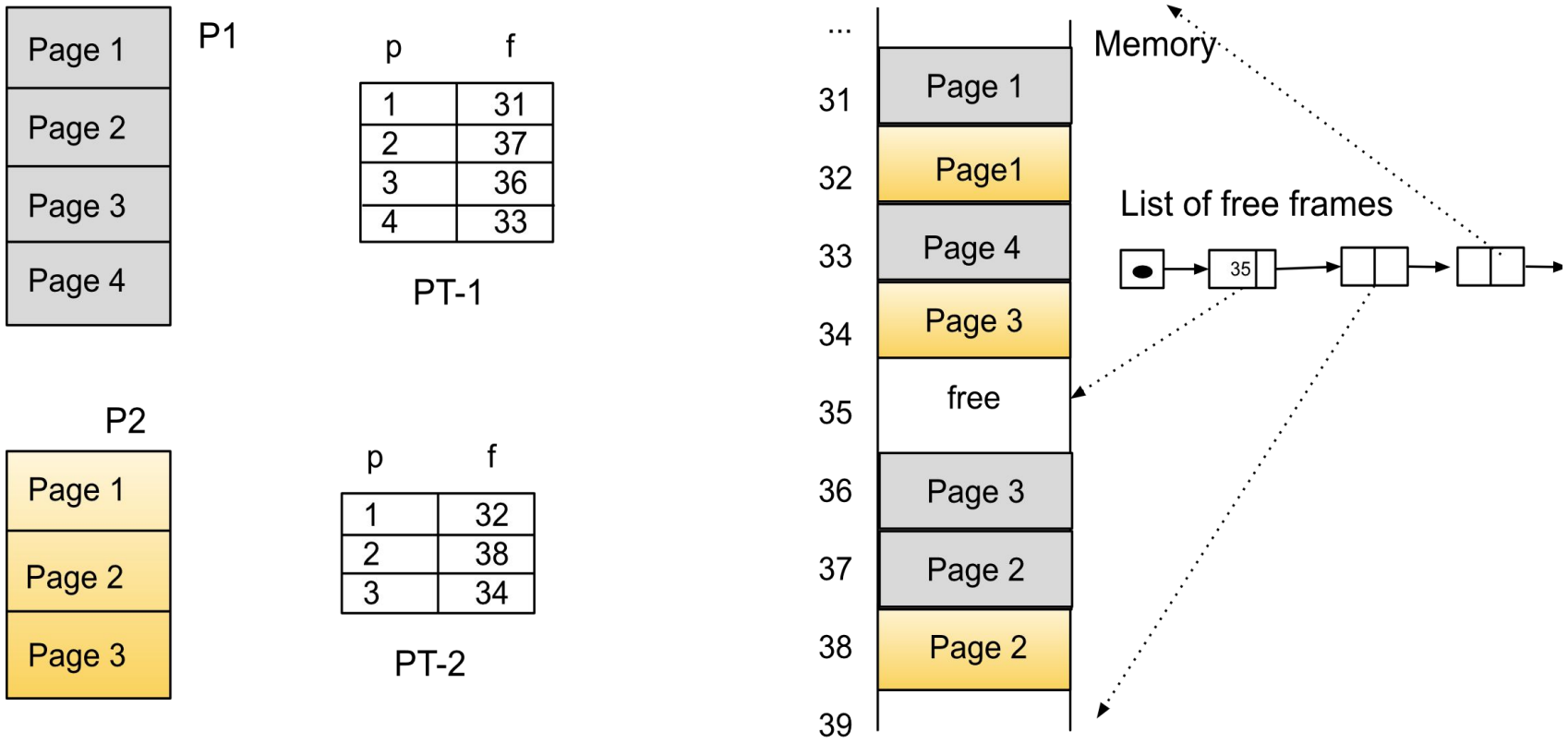


Fig 5.10: Different page table and their allocation

Hardware Support for Paging

- A special register called **Page Table Base Register (PTBR)** points to PT in memory.
- a special, hardware cache **TLB (Translation Lookaside Buffer)**:
  - Fast cache for recently used pages.
  - If TLB hit → fast translation.
  - If TLB miss → memory access needed

Hardware Support for Paging

Average memory access time MAT_{avg} is given by (assuming negligible cache lookup time)

- MAT_{avg} = hit-ratio x mem-access time + (1 - hit-ratio) x 2 x memory-access time.

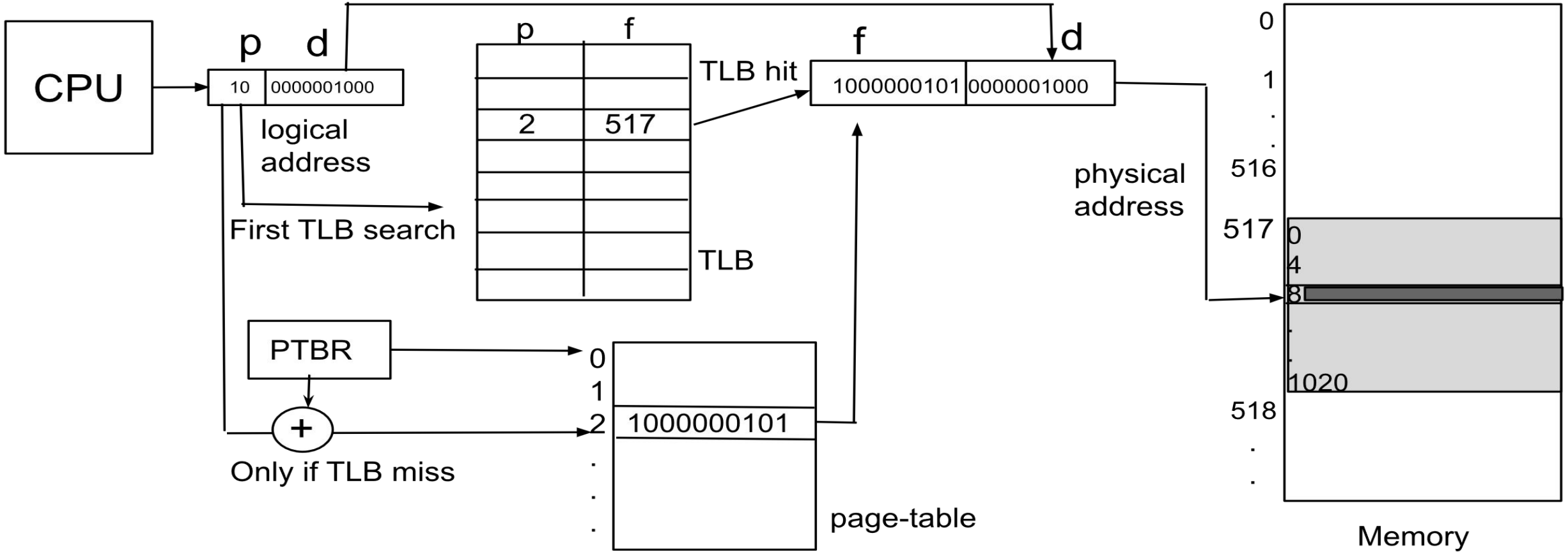


Fig 5.11: Address translation with in-memory page-table and TLB

Page Table Structure

- **Hierarchical Page Table**:
  - Multilevel to handle large address spaces.
  - More levels = more lookup time.
- **Hashed Page Table**:
  - Uses hash function to map pages.
  - Handles collisions via linked lists.
- **Inverted Page Table**:
  - Global table: stores <pid, page#> per frame.
  - Slower due to full table search; hashing used for speed.

Hierarchical Page Table

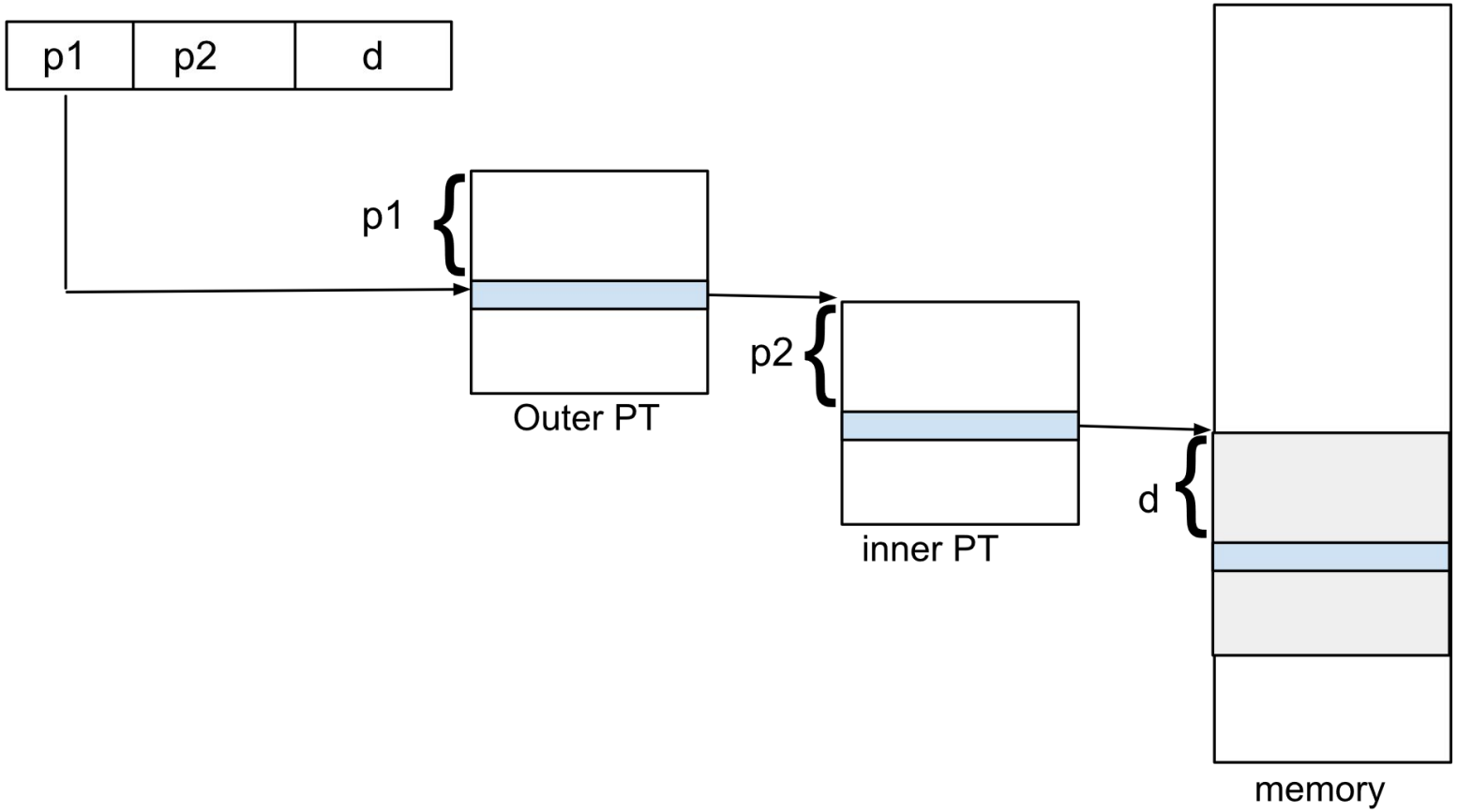


Fig 5.12: Hierarchical page table (32-bit addressing scheme)

Hashed Page Table

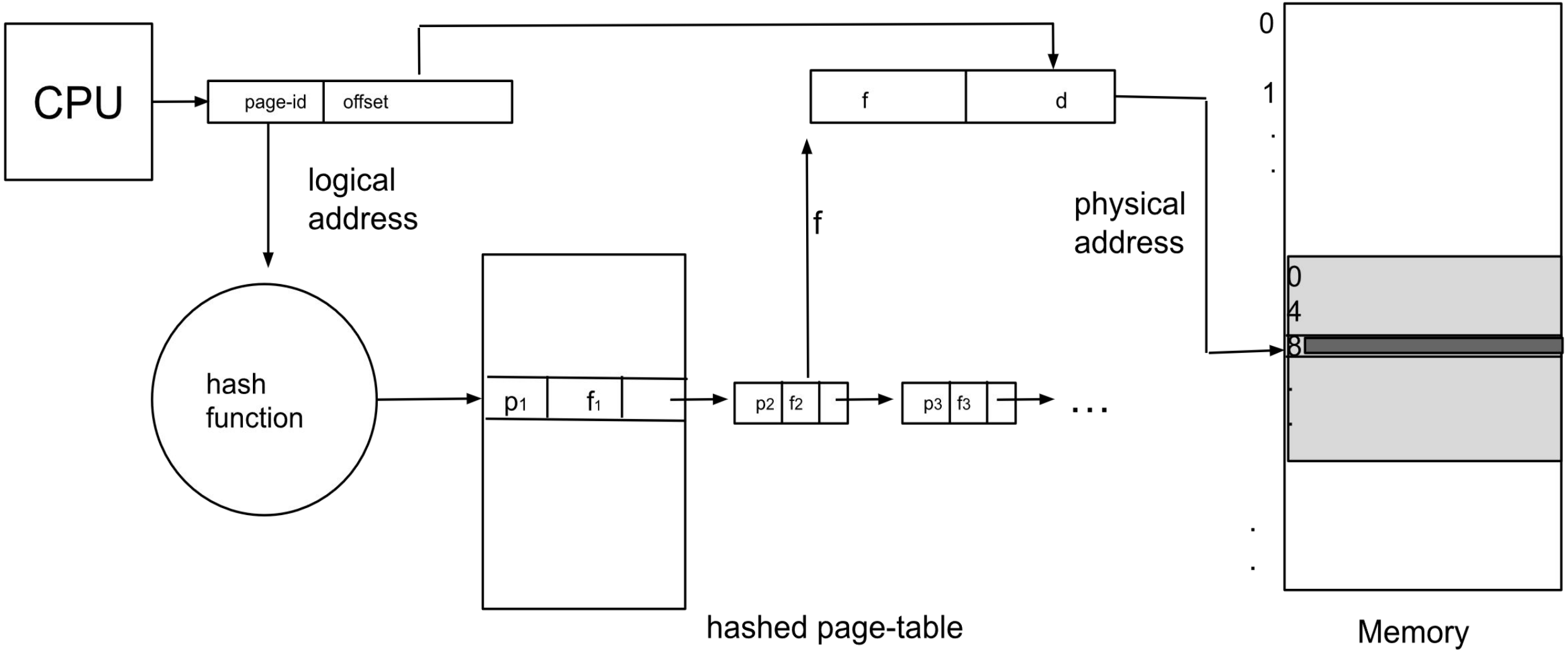


Fig 5.13: Address translation with hashed page-table

Inverted Page Table

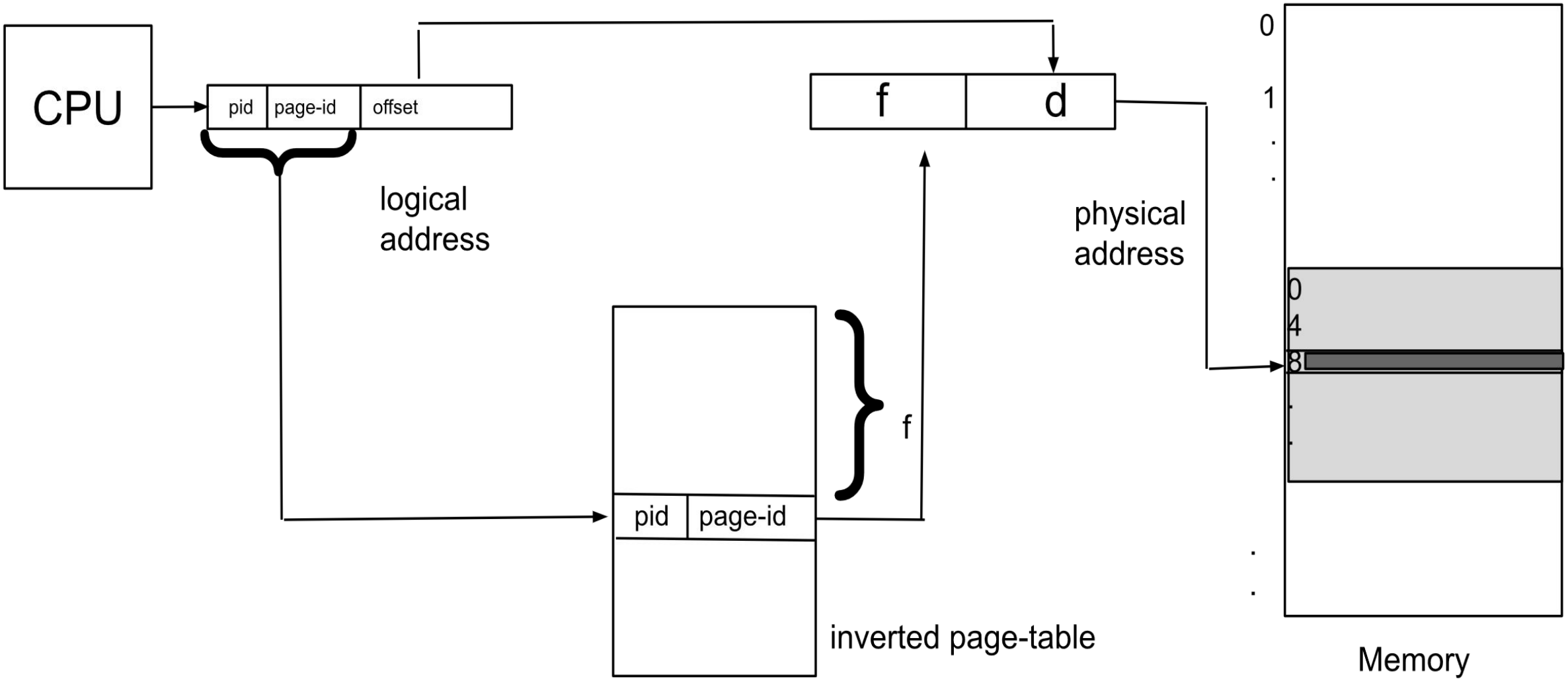


Fig 5.14: Address translation with inverted page-table

Protection in paging

- **Read/Write Protection**: Bit in PT (ro vs rw).
- **Validity Bit**: Ensures reference is to a valid page.
- Traps (interrupts) are generated on invalid access or illegal writes.

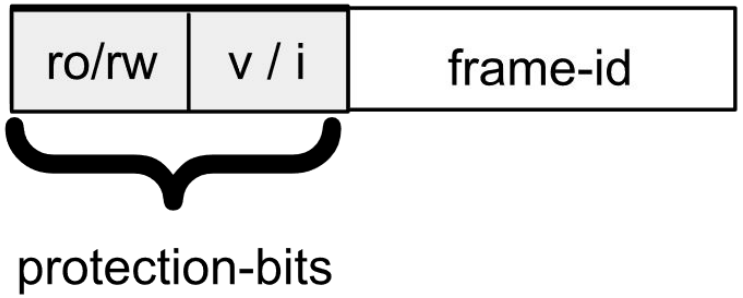


Fig 5.15: An entry in a page table

Page sharing

Example:

- 20 different C programs are running.
- Each is 20MB in size, which includes 2MB for the standard C library (libc).
- Normally, each program would load its own copy of libc.

Without Sharing:

- Total memory = 20 programs × 20MB = 400MB
- libc alone uses = 20 × 2MB = 40MB

With Page Sharing:

- libc is loaded once, and shared across all programs.
- Memory saved = 40MB - 2MB = 38MB

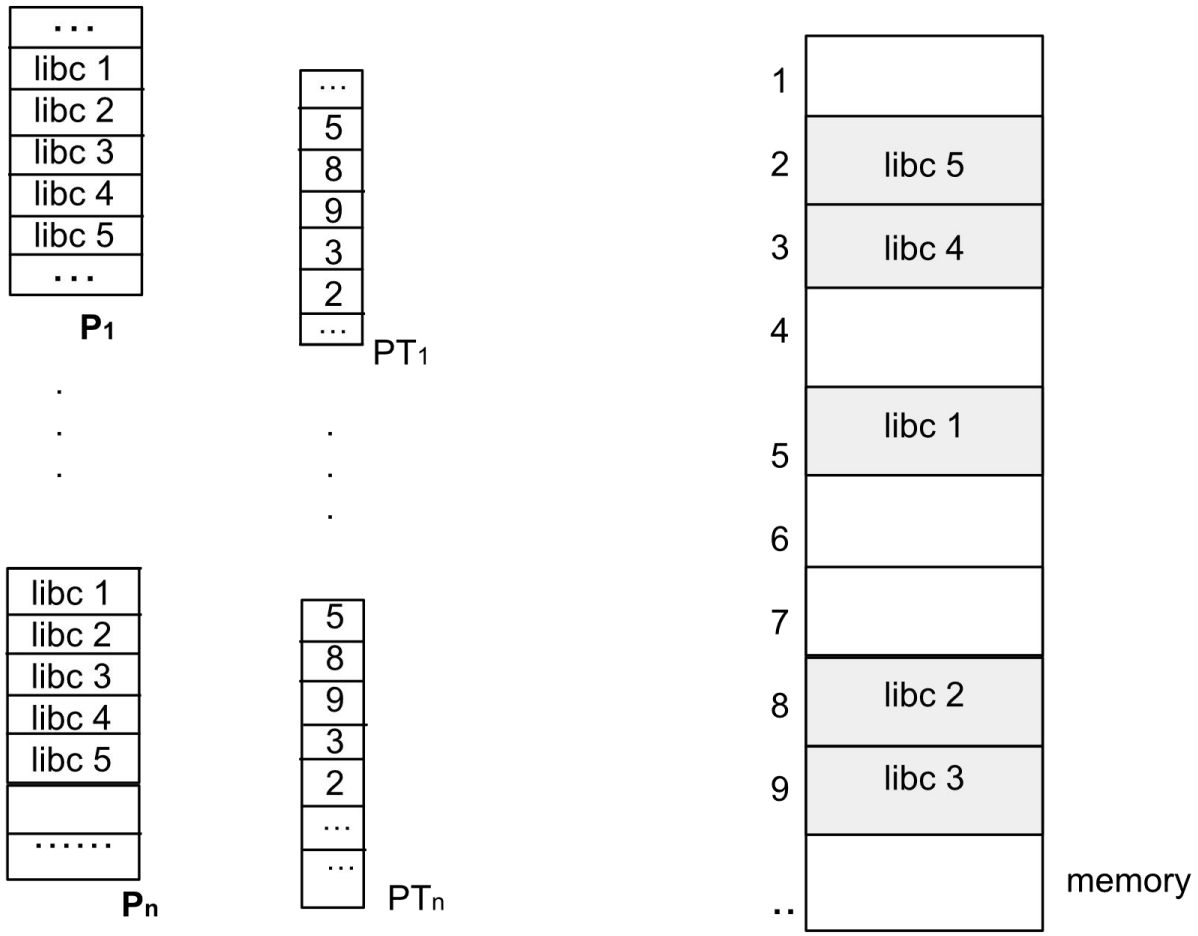


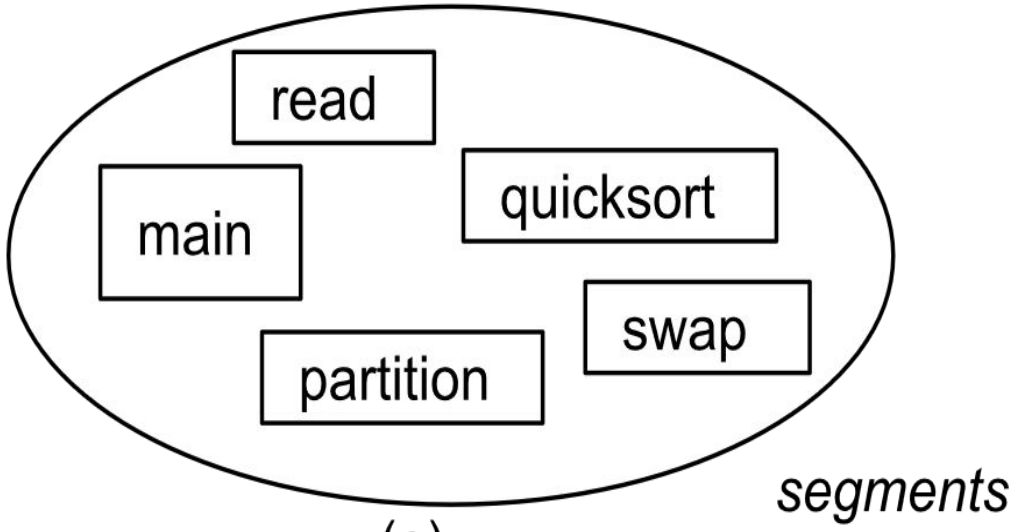
Fig 5.16: Page –sharing among processes

Disadvantages of paging scheme

- **Internal Fragmentation**: Last page might be partially used.
  - E.g., 5200 bytes in 1K pages → 6 pages, 944B wasted.
- **I/O Overhead**: Mismatch between page size and block size can increase I/O operations.
- Trade-off:
  - Smaller pages → less internal fragmentation.
  - Larger pages → better I/O performance.

Segmentation

- Segmentation is a non-contiguous memory allocation technique.
- It divides a program into logical blocks called *segments*.
- Each segment represents a function or data section (e.g., main(), read(), quicksort(), etc.).
- Segments are independently loaded into memory.



(a)

segment table

name	size	address
main (*seg1*)	642	23576
read (*seg2*)	378	32012
partition (*seg3*)	500	76248
quicksort (*seg4*)	476	54500
swap (*seg5*)	188	23204

(b)

Fig 5.17: Segmentation

Segmentation

- Each memory address is represented as <seg-id, offset>.
- seg-id: Identifies the segment.
- offset: Location within the segment.
- Address translation is relative to the segment base address.

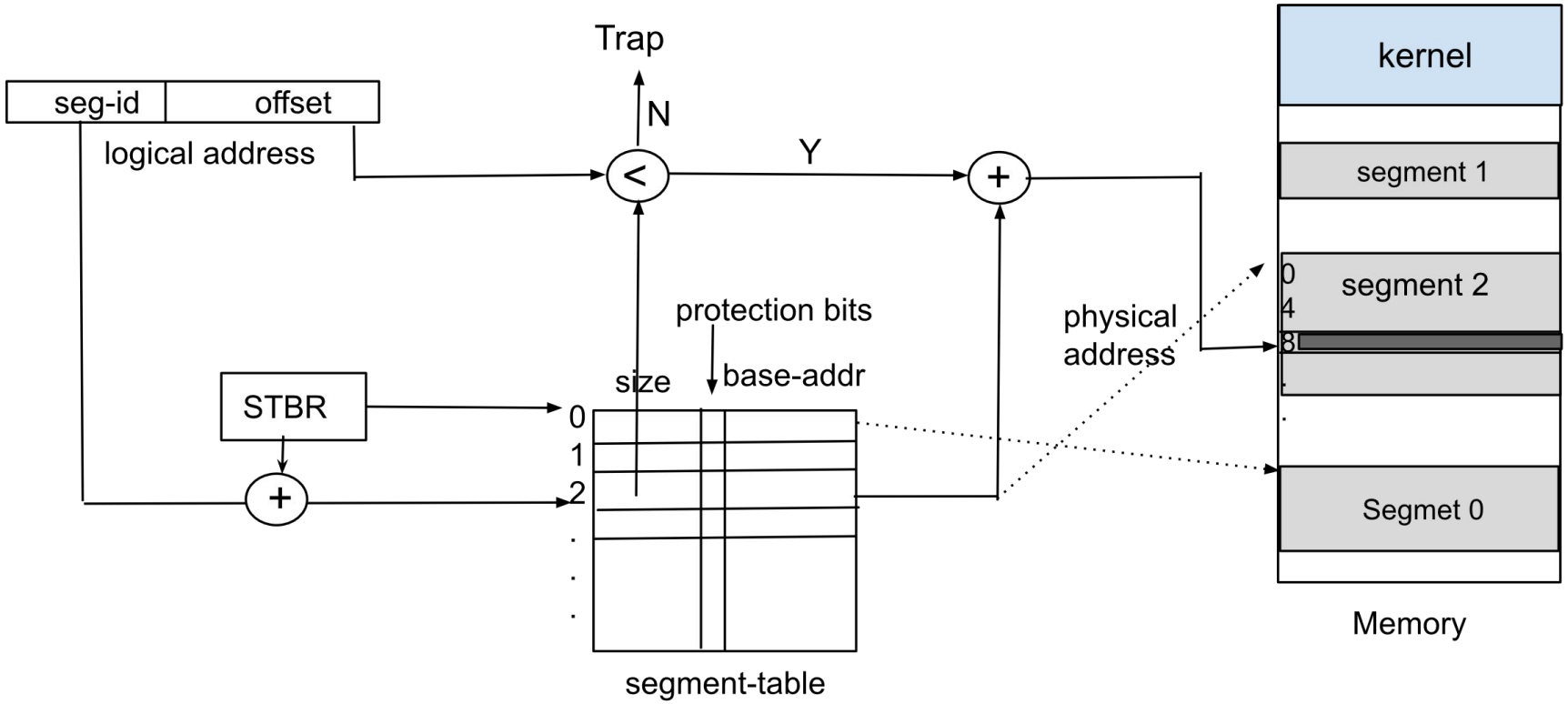


Fig 5.18: Address translation in segmentation (contiguous allocation to segments)

Paged Segmentation

- A combination of segmentation and paging techniques.
- Each segment is divided into pages.
- Segments get a variable number of pages depending on their size.
  - Step 1: Use <seg-id> to identify the segment from the segment table.
  - Step 2: Get base-address of the page table for that segment.
  - Step 3: Use MSBs of offset to find the page within the segment.
  - Step 4: Use LSBs of offset to get the exact location within the page.

Paged Segmentation

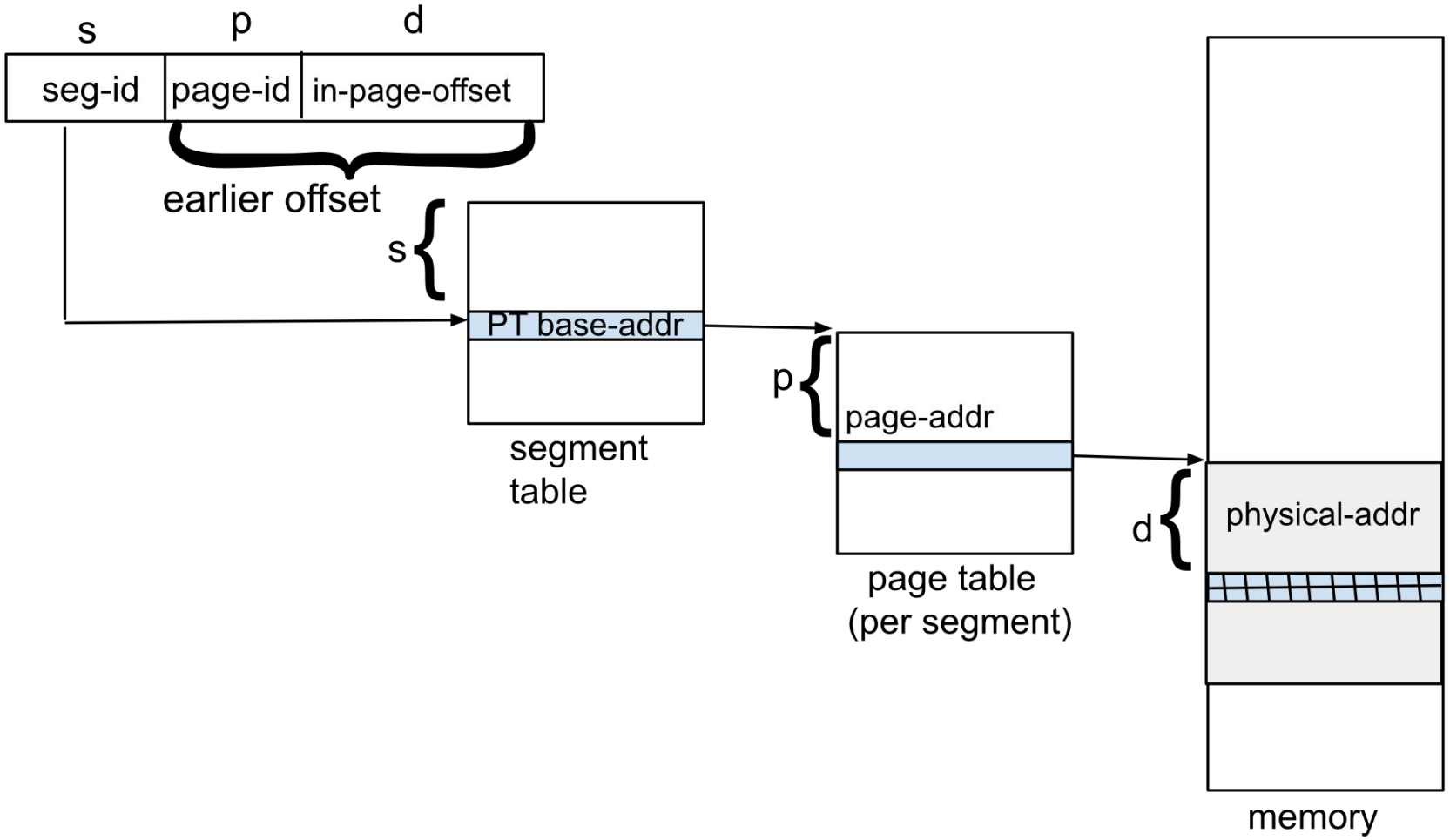


Fig 5.19: Paged segmentation (or, segmentation with paging)

Virtual Memory

- Virtual Memory (VM) = Illusion of a larger main memory
- Achieved by using part of secondary memory (e.g., hard disk)
- Only frequently used parts of a program stay in RAM

Virtual memory is a logical extension of the following facts:

1) *non-contiguous memory allocation* techniques: MMU allows loading and accessing of the code and data of a process scattered all over the memory.
2) every program has a *locality of references*: memory references within a time window are usually made to locations that are spatially close to each other (e.g., accessing successive array elements or nodes in a linked list) and often the same instructions are executed several times repeatedly (within a loop)
3) a program may have a lot of non-essential code to tackle error conditions and/or special cases that are rarely encountered.

Virtual Memory

- Hardware Supports address translation (via MMU, TLB, Page Table)
- Virtual Memory Manager (VMM) handles:
  - When to load code/data (pages/segments)
  - When/how to replace them to/from backing store
  - Implements placement and replacement algorithms

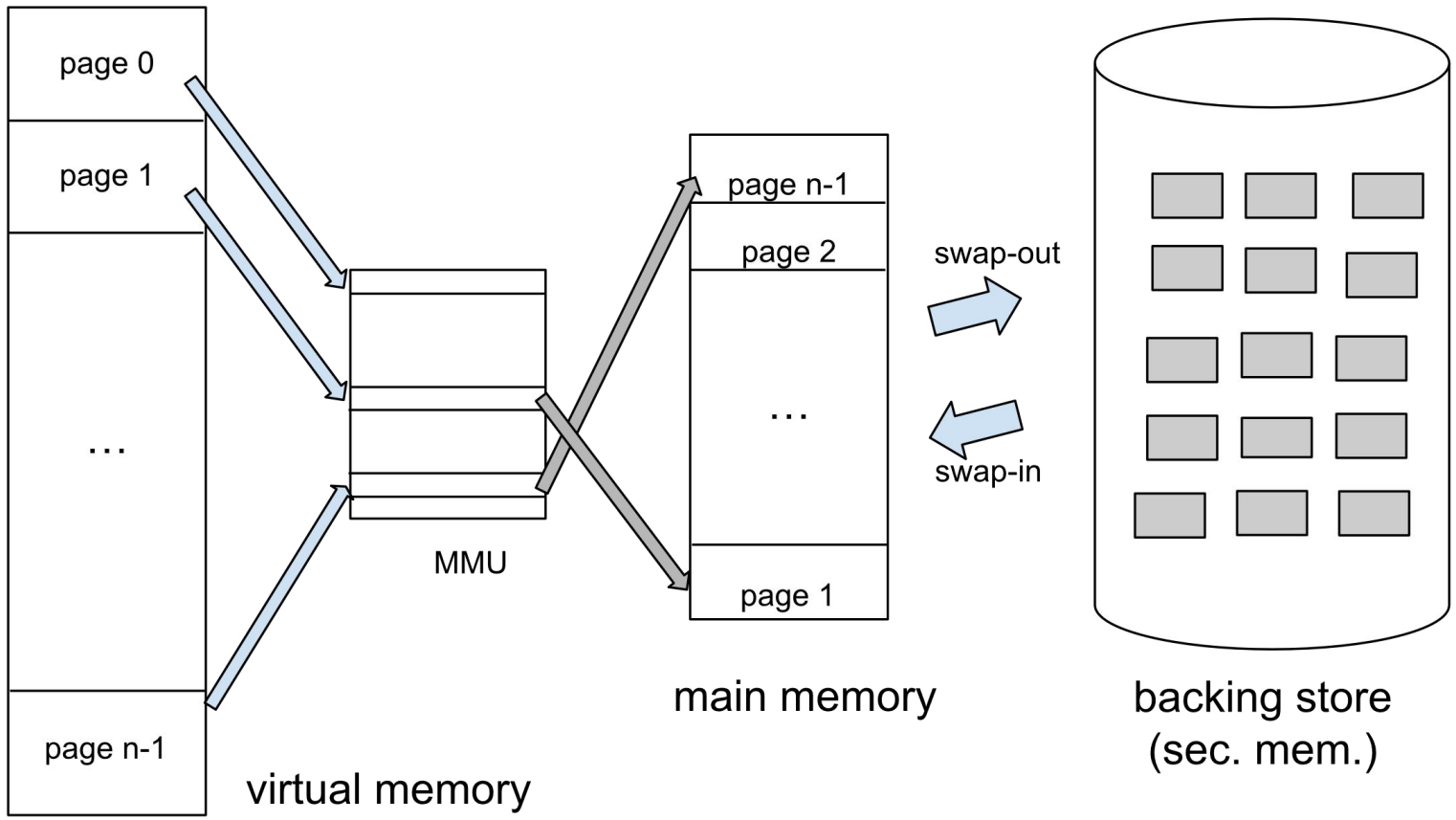


Fig 5.20: A schematic view of virtual memory

Virtual Memory

- **Backing Store:** Portion of secondary memory used to hold non-resident parts of active processes.
- **Page Fault:** Page not in main memory → triggers memory access fault.
- **Steps in Handling**:
  1. Memory reference triggers a trap
  2. OS checks if the reference is valid (legal address)
  3. Page fetched from backing store (disk I/O)
  4. Page table updated with new frame info
  5. Instruction restarts
- Assume that a memory access takes 100 nanoseconds, and disk I/O takes 10 milliseconds then, with page fault rate p:
  - $\textit{effective memory access time} = 100 * (1 - p) + p * 10 * 10^6$ nanoseconds $= (100 + 9{,}999{,}900 * p)$ nanoseconds.

Virtual Memory

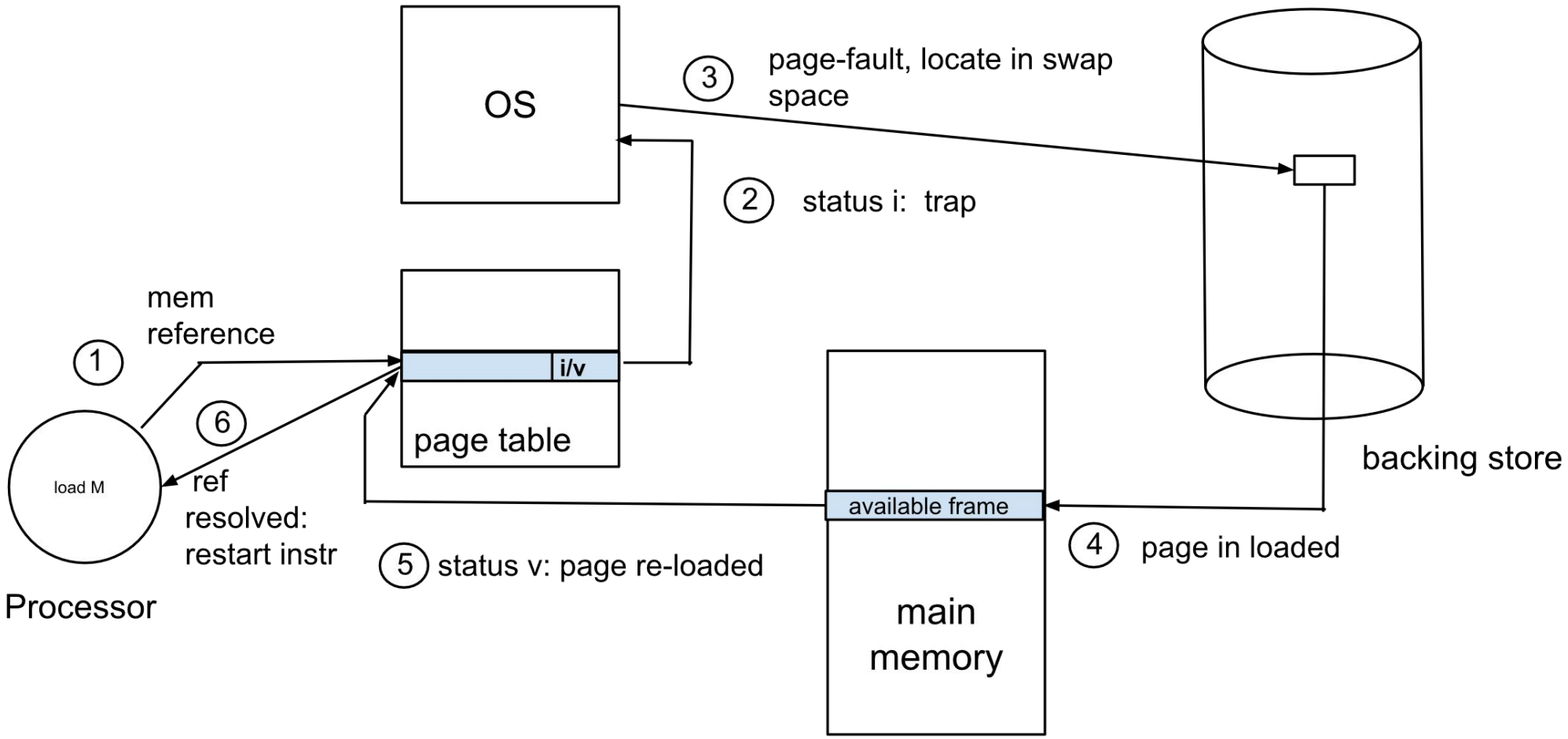


Fig 5.21: Page-fault handling

Virtual Memory

- **Locality of References:**
  - Temporal: Recently used pages are reused
  - Spatial: Nearby memory locations are used together

Virtual Memory

- From the Diagram (Fig. 5.22):
- Programs exhibit Temporal Locality:
  - Same page accessed repeatedly in a short time (e.g., region t)
- Programs exhibit Spatial Locality:
  - Neighboring memory locations accessed closely in time (e.g., regions a–h)
- Regions:
  - a (Pages 18–26), b (Page 31), c (Page 34) are initially active
  - d, e, f, g, h represent changing localities during execution
- Page 32 accessed briefly → shows non-local behavior
- Shifts between regions reflect function calls and returns

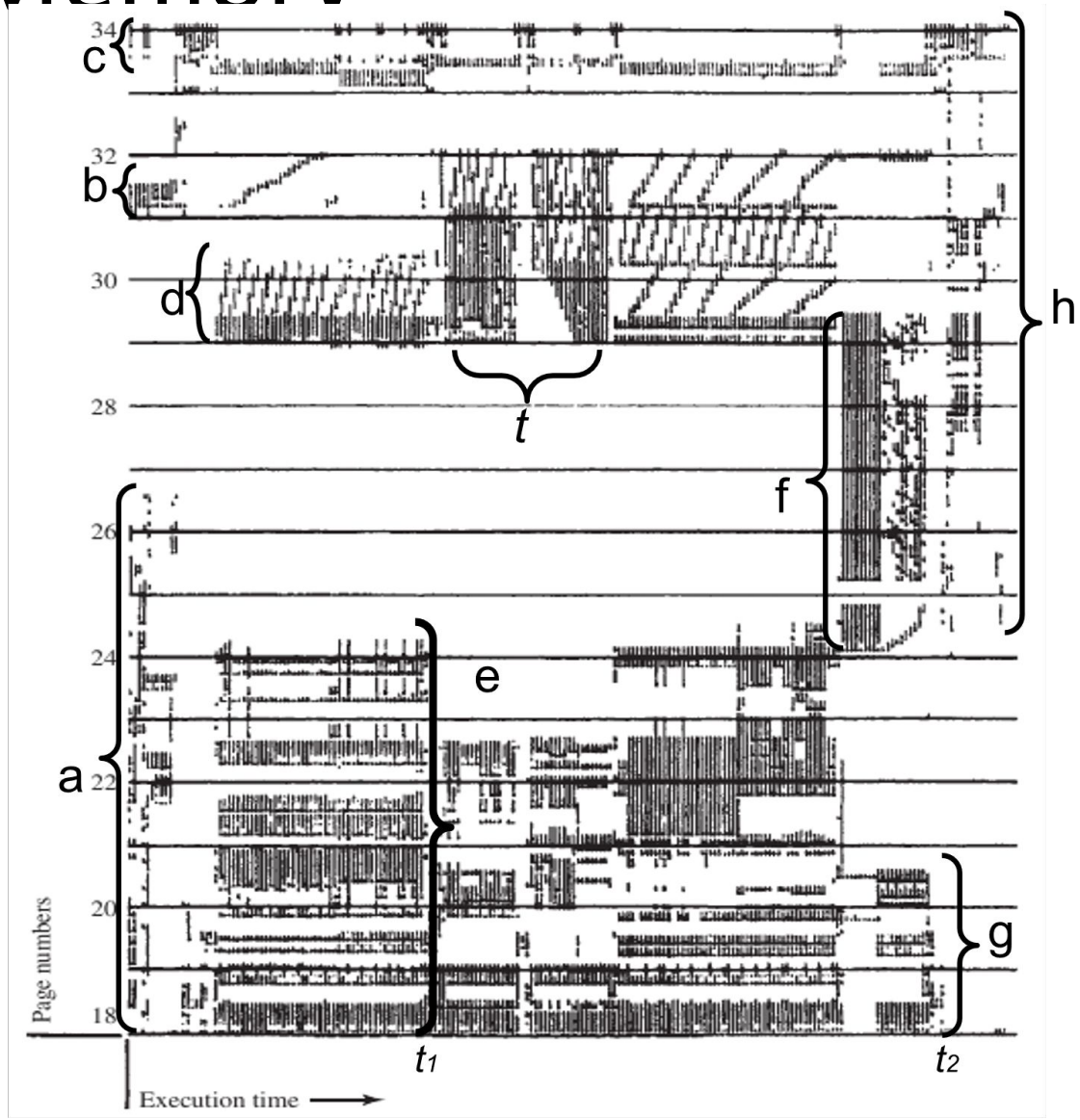


Fig 5.22: Any program execution shows strong locality of references ([Hat72])

Virtual Memory

- **Working Set (WS):** Set of pages used in a time window (Δ).

- If Δ=5, we see the last 5-page references.
- Too small: misses locality
- Too large: includes unnecessary pages

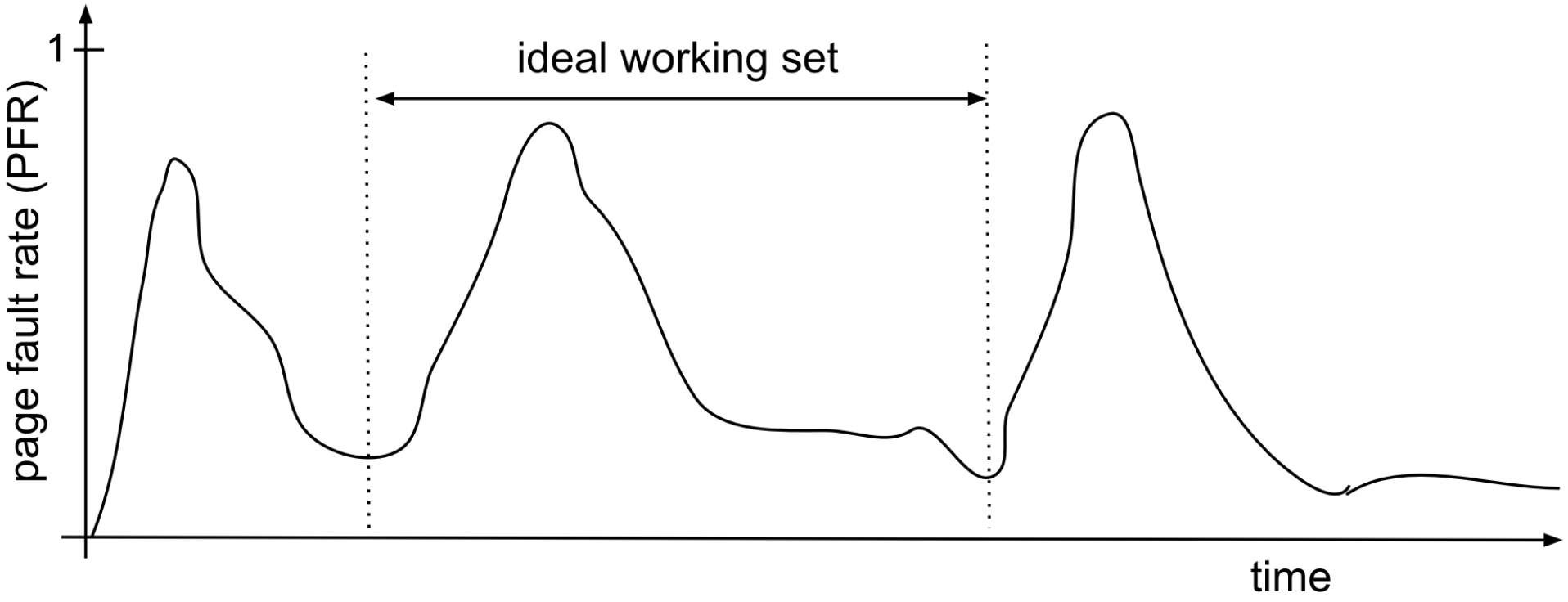


Fig 5.23: Relation between working set and page faults

Page-level Protection in Virtual Memory

- **Valid/Invalid Bit**:
  - Valid: Page is in memory
  - Invalid: Not in memory or not part of the process
- **Read-Only / Read-Write (ro/rw) Bit**:
  - ro: Cannot be modified
  - rw: Can be modified by the process
- **Dirty Bit (Modify Bit)**:
  - Set if a writable page has been modified
  - Used to decide if a page needs to be written to **swap space** before replacement

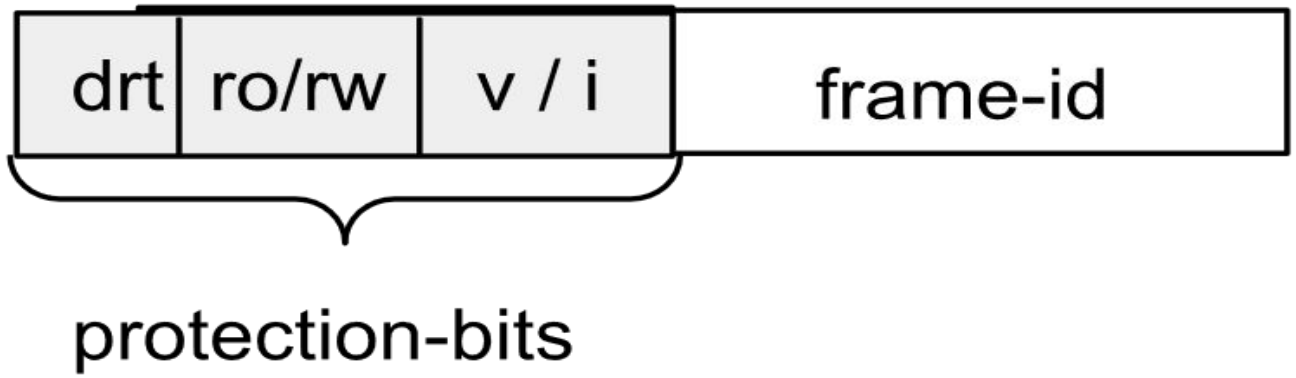


Fig 5.24: An entry in a page table

Operating System Software

- The first two decisions are hardware-driven
- Performance of the VM implementation depends on a few software issues
  1. *Fetch policy*: when to bring pages (or segments) to the memory
  2. *Placement & Replacement policy*: where and how to place the fetched page(s)
  3. *Resident Set Management*: how many frames to be allocated per process
  4. *Load Control*: how many processes to be accommodated in the memory (degree of multiprogramming).
- **Demand Paging**
  - Load pages only when needed (on-demand)
  - Saves memory & I/O but causes page faults
  - Pure demand paging : 100%page faults

Operating System Software

- Page Replacement Algorithm:

  Used when no free frame is available:

  - *OPT (Optimal Replacement)* – Replaces the page that will not be used for the longest time in the future
  - *FIFO (First-In First-Out)* – Replaces the oldest loaded page
  - *SC (Second Chance)* – FIFO with a reference bit check to give pages a "second chance"
  - *NRU (Not Recently Used)* – Uses reference and modified bits to classify and replace pages
  - *LRU (Least Recently Used)* – Replaces the page that was least recently accessed

Optimal Page Replacement Algorithm (OPT)

Assume that 4 frames are allocated to the process Considering 1 KB (1024 bytes) page-size, corresponding page references are:

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

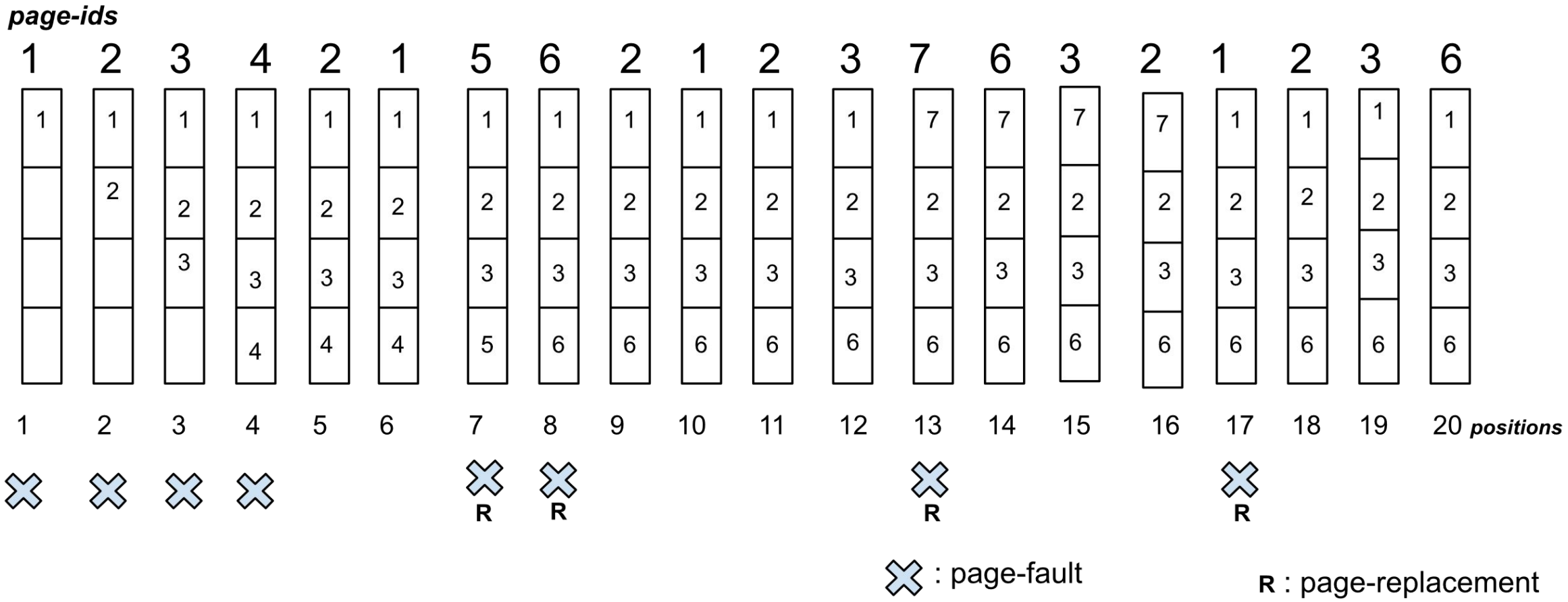


Fig 5.25: Optimal Page Replacement Algorithm

First in First Out (FIFO) Page Replacement Algorithm

Assume that 4 frames are allocated to the process Considering 1 KB (1024 bytes) page-size, corresponding page references are:

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

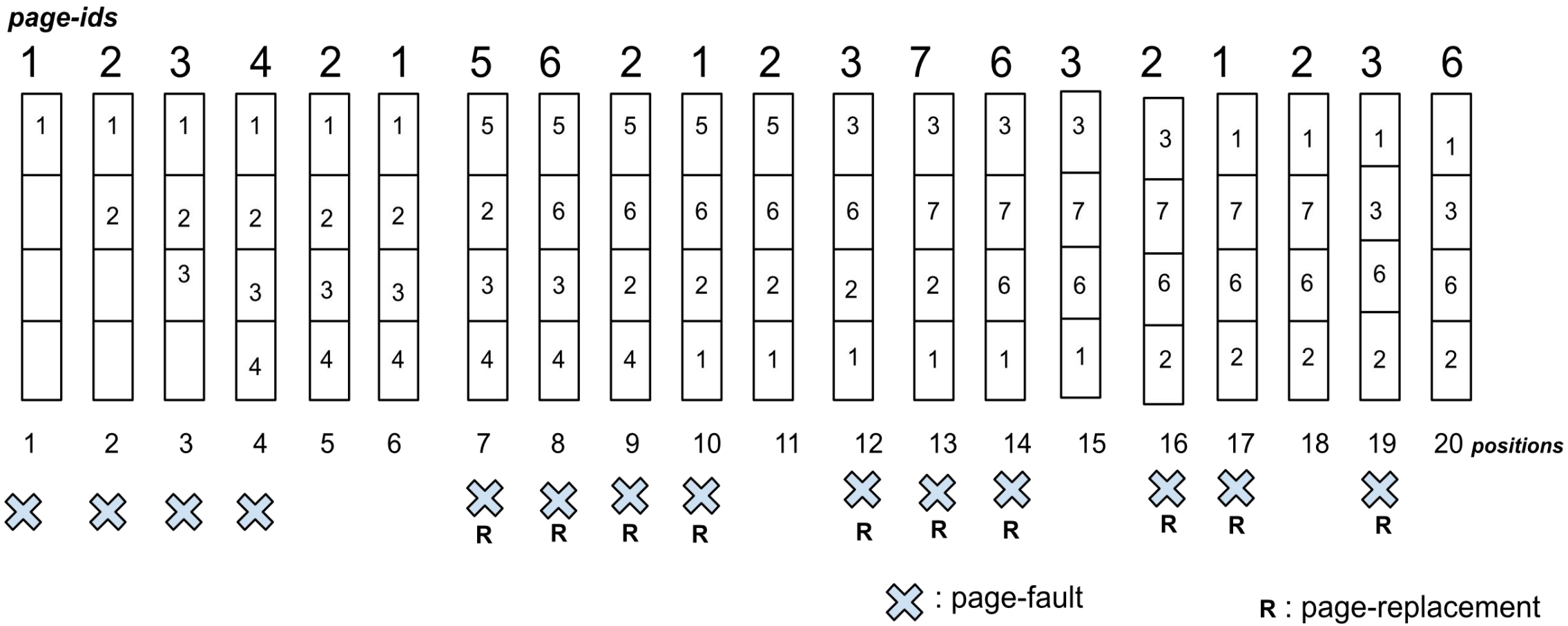


Fig 5.26: FIFO Page Replacement Algorithm

First in First Out (FIFO) Page Replacement Algorithm

- In some of the page replacement algorithms increasing the number of frames sometimes causes an increase in the number of page faults.
- This anomalous phenomenon is called Belady's anomaly

Fig 5.27a: #PFs vs #frames allocated

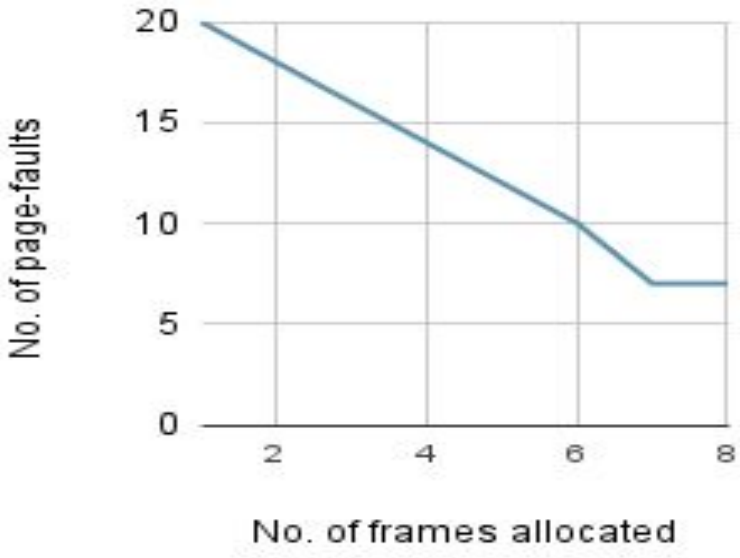
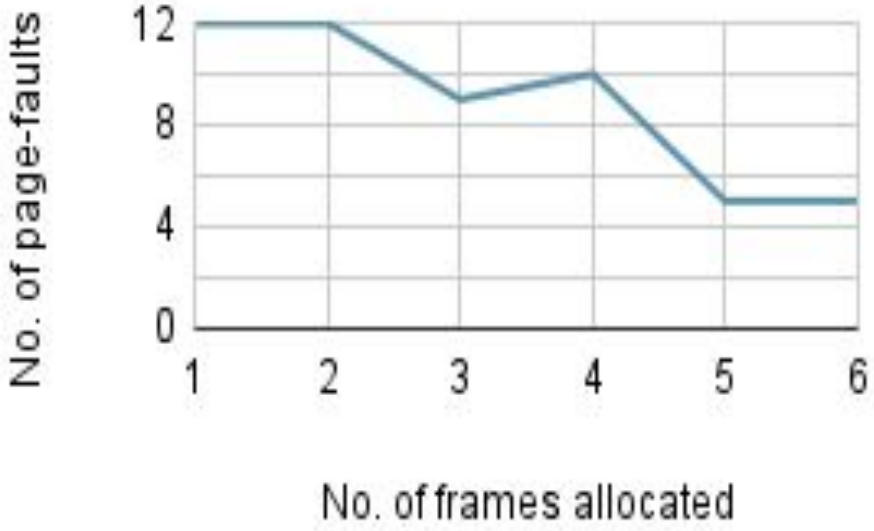


Fig 5.27b: Belady's Anomaly



Second Chance (SC) Page Replacement Algorithm

- Improvement over FIFO using a reference bit (use-bit)
- Each page/frame has a use-bit:
- Set to 1 when referenced
- Initially 0 when loaded
- Pages are organized in a circular buffer
- When replacement is needed:
  1. Check the next frame in order
  2. If use-bit = 1 → reset it to 0

     (give a *second chance*) and skip
  3. If use-bit = 0 → replace the page
  4. If all bits are 1 → on second round, the first 0 is selected

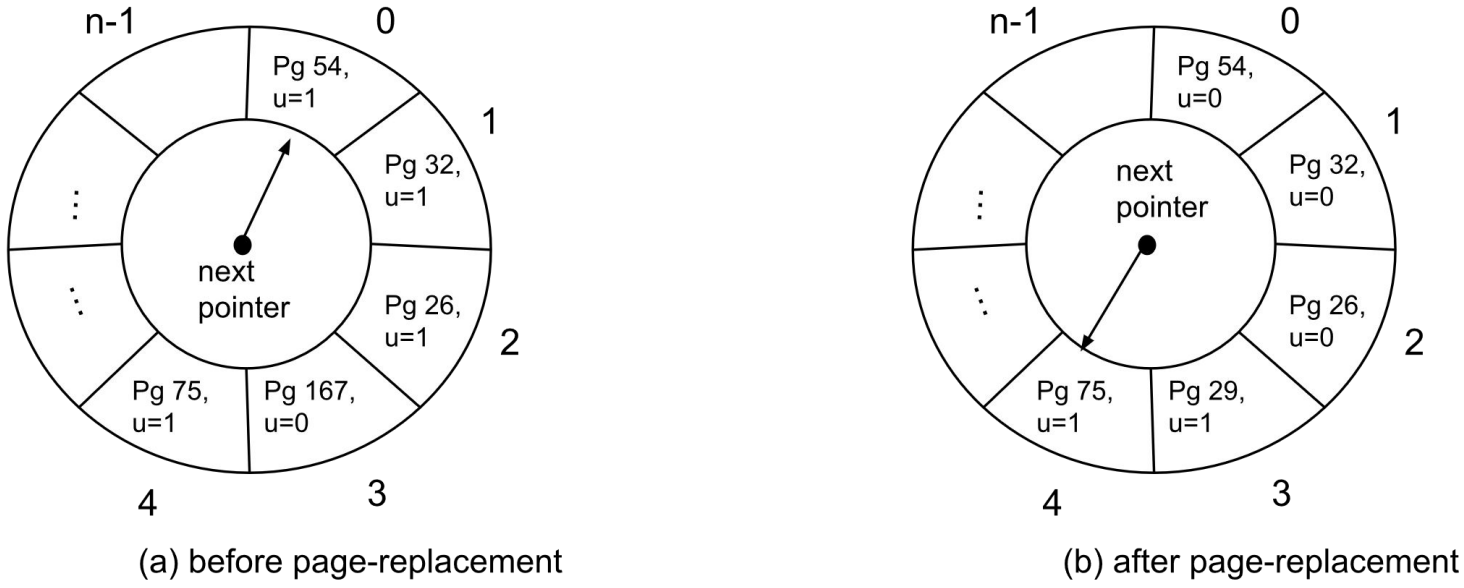


Fig 5.28: Second-Chance (SC) Page Replacement Algorithm

Not Recently Used (NRU) Page Replacement Algorithm

- **Enhanced Second Chance** algorithm using both:
  - **Use-bit (u)**: recently used
  - **Modify-bit (m)**: modified since loaded
- **Frames classified into 4 categories**:

  1. **u=0, m=0** → Not recently used & not modified *(Best candidate)*
  2. **u=0, m=1** → Not recently used but modified
  3. **u=1, m=0** → Recently used but not modified
  4. **u=1, m=1** → Recently used and modified *(Worst candidate)*
- **Algorithm Steps:**

  1. Search circularly for a frame in class 1 and Replace
  2. If such a frame is not found , go for class 2 frame
  3. If Step 2 fails, reset all **use-bits** from 1 to 0 and repeat search
  4. re-run Step 1 or if needed, then Step 2 to find the victim frame. **Modify-bit** is never reset

Least Recently Used (LRU) Page Replacement Algorithm

Assume that 4 frames are allocated to the process Considering 1 KB (1024 bytes) page-size, corresponding page references are:

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

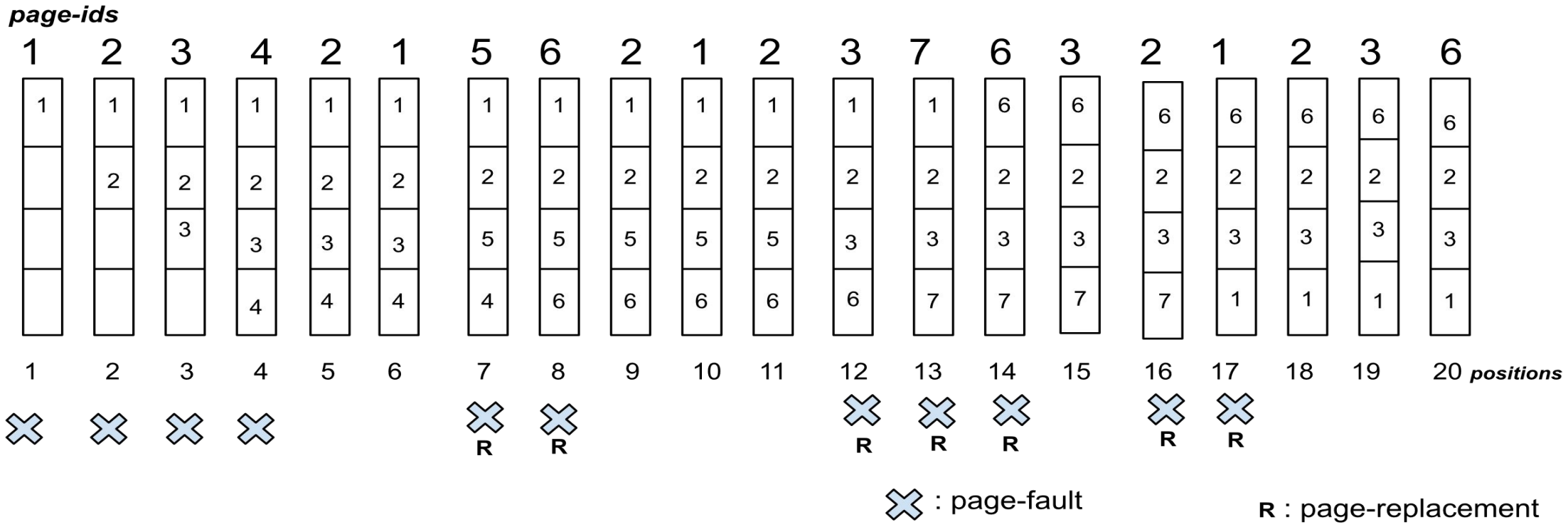


Fig 5.29: LRU Page Replacement Algorithm

Least Recently Used (LRU) Page Replacement Algorithm

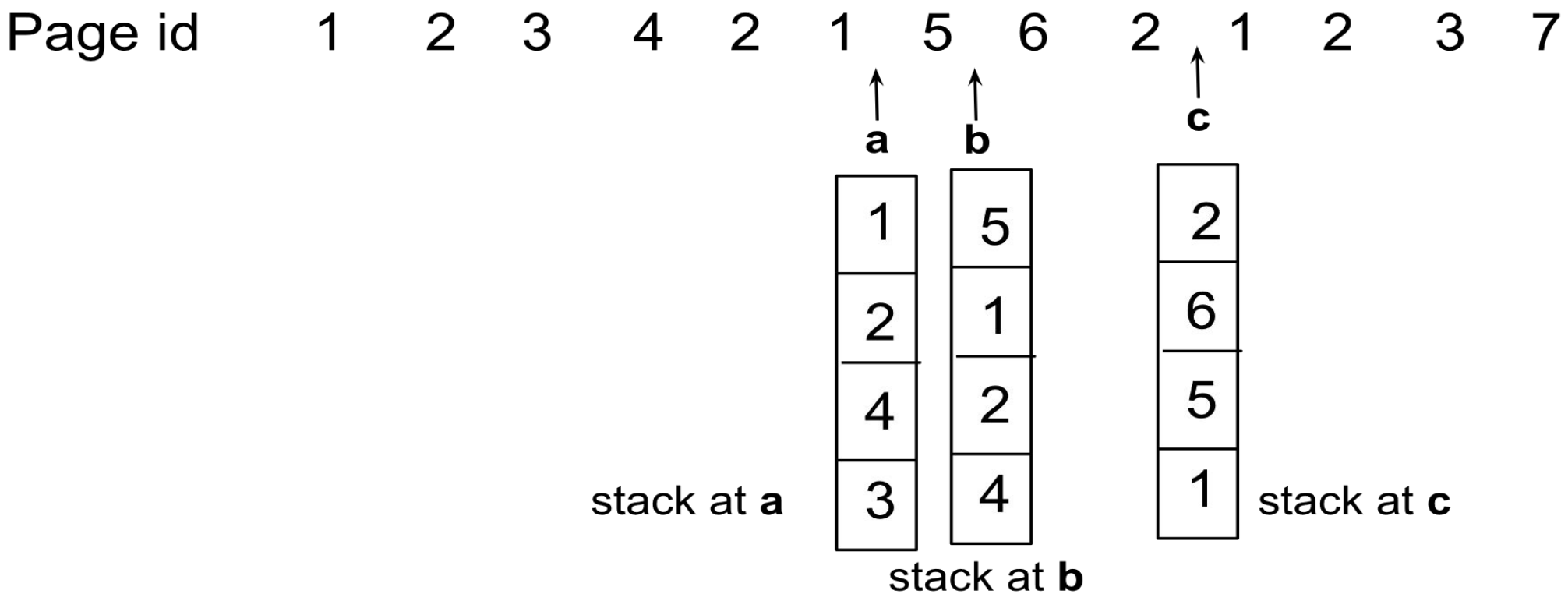


Fig 5.30: Stack-based implementation of LRU

Comparison

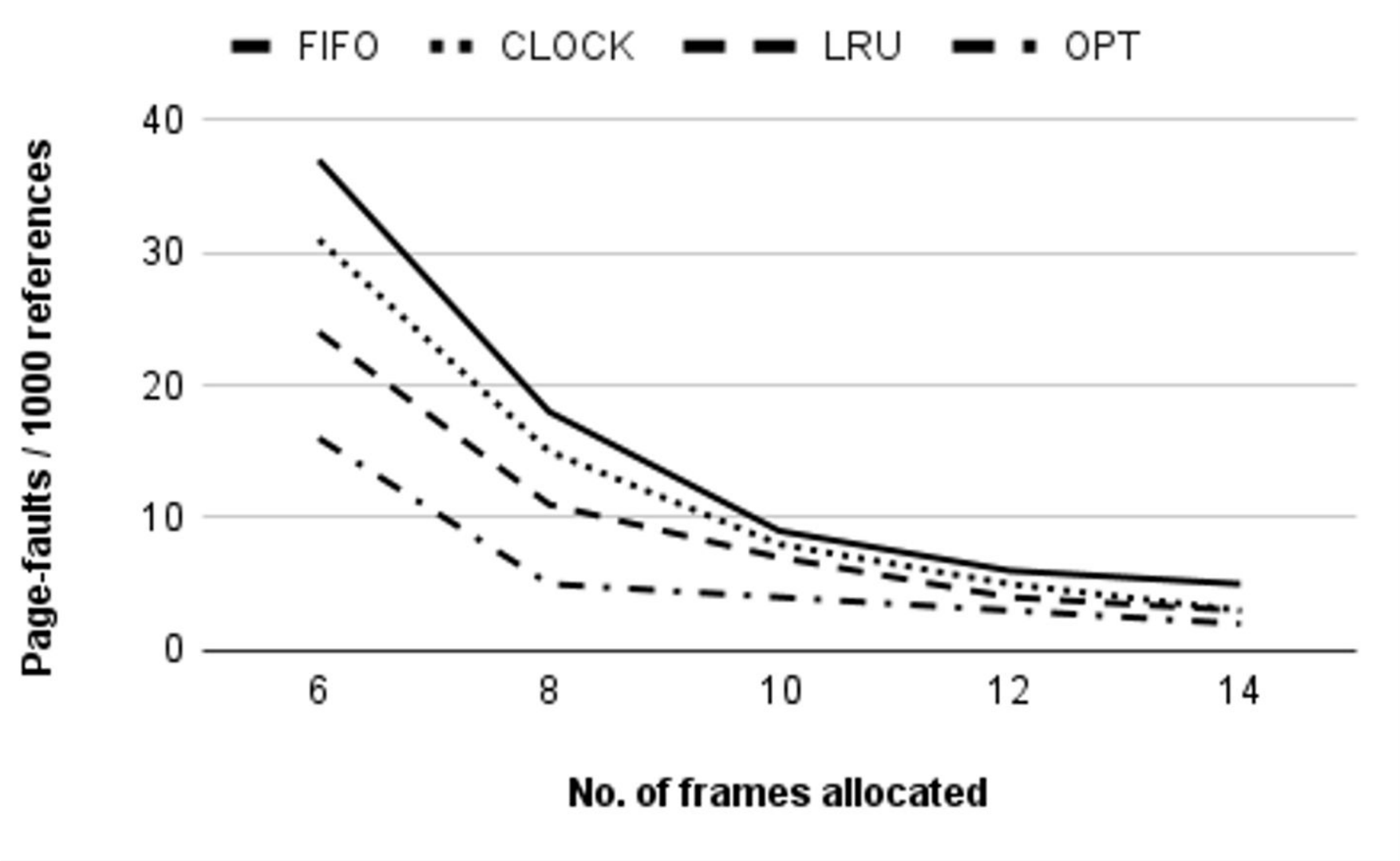


Fig 5.31: Performance Comparison

Operating System Software

- **Resident Set Management:**
  - **Fixed Allocation**: Constant frame count per process
  - **Variable Allocation**: Frame count changes dynamically
  - **Policies**:
    - **Local Replacement**: Within a process’ frames
    - **Global Replacement**: From any process’ frames

Operating System Software

- **Load Control:**
  - Controls degree of multiprogramming
  - Too low : CPU underutilized
  - Too high : High Page Fault Rate -> Thrashing
  - **Thrashing**: CPU spends more time swapping than executing
  - OS monitors PFR & may suspend processes to prevent thrashing
  - Which process(es) need to be suspended - depends on lot many factors.

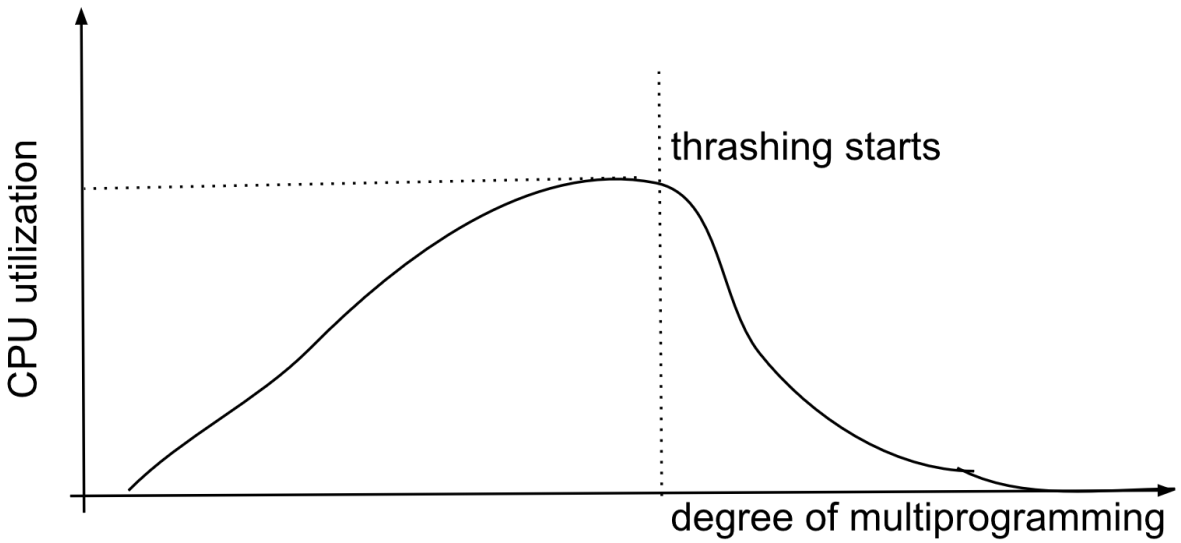


Fig 5.32: Thrashing

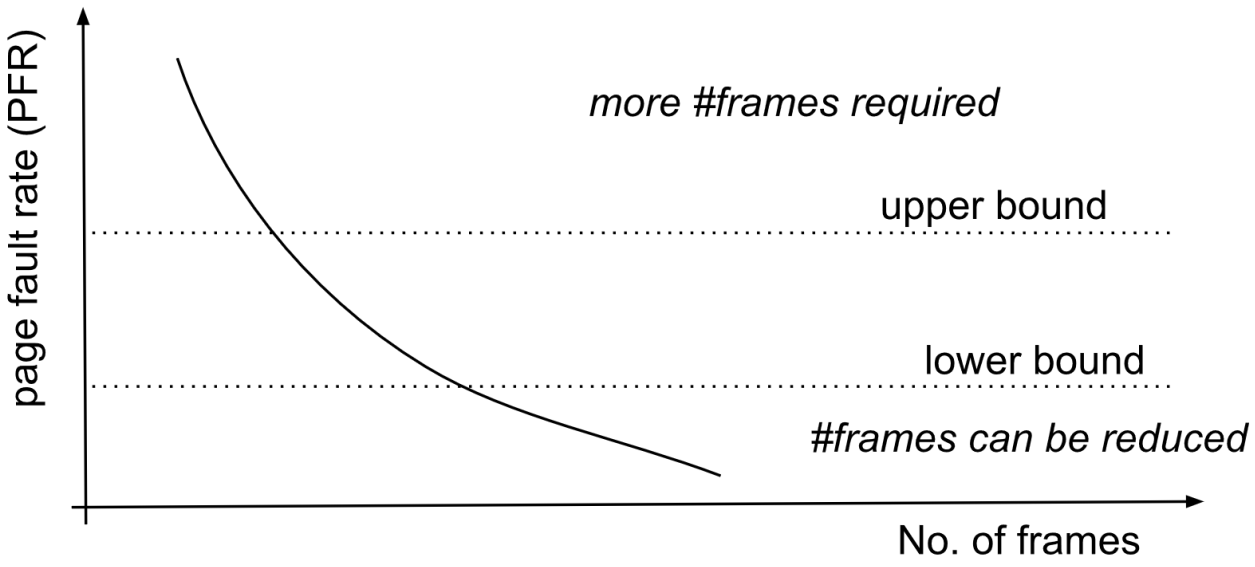


Fig 5.33: Dynamic allocation based on PFR

- Virtual memory implementation with demand paging is summarized in the following flow-diagram.

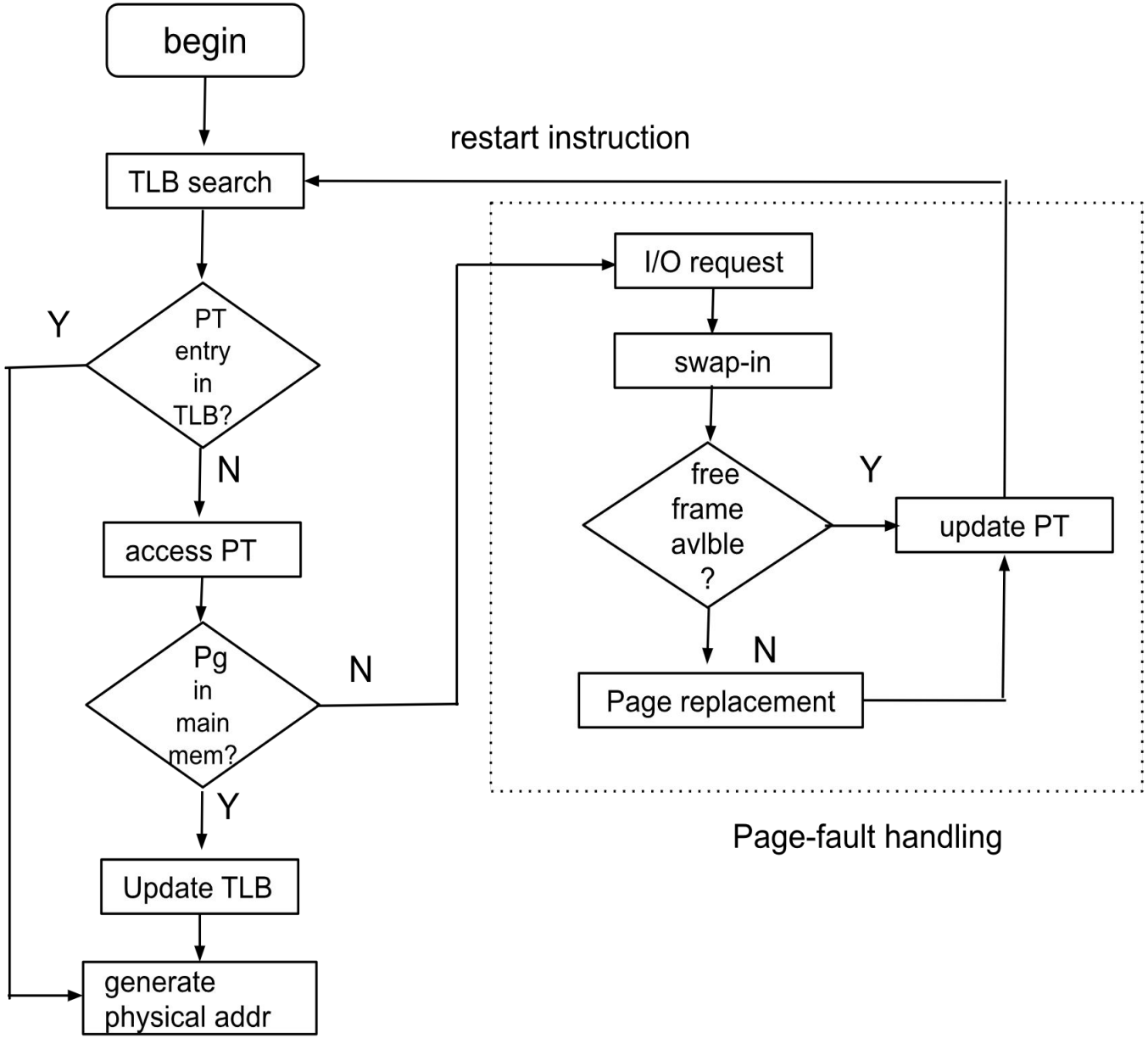


Fig 5.34: Page-fault handling using TLB

Reference

[1] “OPERATING SYSTEMS”, Author: Dr. Sukomal Pal Associate Professor Department of Computer Science & Engineering IIT (BHU), Varanasi, UP