SE 292: High Performance Computing [3:0][Aug:2014]

Memory Organization *Redux*

Yogesh Simmhan

Adapted from "Memory Organization and Process Management", Sathish Vadhiyar, SE292 (Aug:2013) & "Operating Systems Concepts", Silberschatz, Galvin & Gagne, 2005

Preliminaries

- Continuity
  - From Prof. Govindarajan earlier in the semester
  - Previous editions of course taught by Prof. Vadhiyar
- Lectures: Tue & Thu 8-930AM
  - Office Hours: Thu 930-1030AM
- Grading
  - Assignments & Projects : 25%
  - Midterms : 11/Sep, 14/Oct, 11/Nov : 30%
  - Final Exam : Dec ??? : 45%

Preliminaries

- Classes are interactive. Stop & ask questions.
- Use the mailing list. Use the office hours.
- Academic Integrity
  - Students must uphold IISc's Academic Integrity guidelines. Review them. Failure to follow them will lead to penalties, including a reduced or failing grade in the course. Severe violations will be reported to the Institute and may lead to an expulsion.
  - http://www.iisc.ernet.in/students-corner/existingstudents-academicintegrity.php
- Call me *Yogesh*
  - Not “Sir” or “Prof” or “Dude” ...
  - You’re grad students and soon to be peers. Be courteous but not obsequious.

Preliminaries: Syllabus

- **Introduction to computer systems:** Processors, Memory, I/O Devices; Cost, timing and scale (size) models; Program Execution: Machine level view of a program; typical RISC instruction set and execution; Pipelining. Performance issues and Techniques; Caching and Virtual Memory. Temporal and spatial locality. Typical compiler optimizations. Identifying program bottlenecks – profiling, tracing. Simple high level language optimizations – locality enhancement, memory disambiguation. Choosing appropriate computing platforms: benchmarking, cost-performance issues. [6 weeks]
- **OS Concepts:** Process Management, System Calls, Memory Management, and Disk & I/O Management. **[~3 weeks]**
- **Parallel Computing:** Introduction to parallel architectures and interconnection networks, communication latencies. Program parallelization; task partitioning and mapping; data distribution; message passing; synchronization and deadlocks. Distributed memory programming using MPI. Shared memory parallel programming. Multithreading. **[~6 weeks]**

Preliminaries: Reference Material

- **Operating Systems Concepts**

Silberschatz, Galvin & Gagne, Wiley, 7th Edition (2005) or later

- **Computer Systems: A Programmer's Perspective**

Bryant & O'Hallaron, Pearson, 2nd Edition (2011) or later

- **Introduction to Parallel Computing**

Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Addison Wesley, 2nd Edition (2003) or later

Preliminaries: Holidays!

✗ Thu 2 Oct	Mahatma Gandhi's Jayanthi.
✓ Sat 11 Oct	Substitute Class @ 1030AM?
✗ Thu 23 Oct	Deepavali.
✓ Sat 1 Nov	Substitute Class @ 1030AM?
✗ Tue 4 Nov	Official Holiday. Muharram
~~✓ Wed 5 Nov~~	~~Substitute Class @ 1030AM?~~
✗ Thu 6 Nov	Official Holiday. Guru Nanak's Birthday
✗ Sat 8 Nov	Substitute Class @ 1030AM?

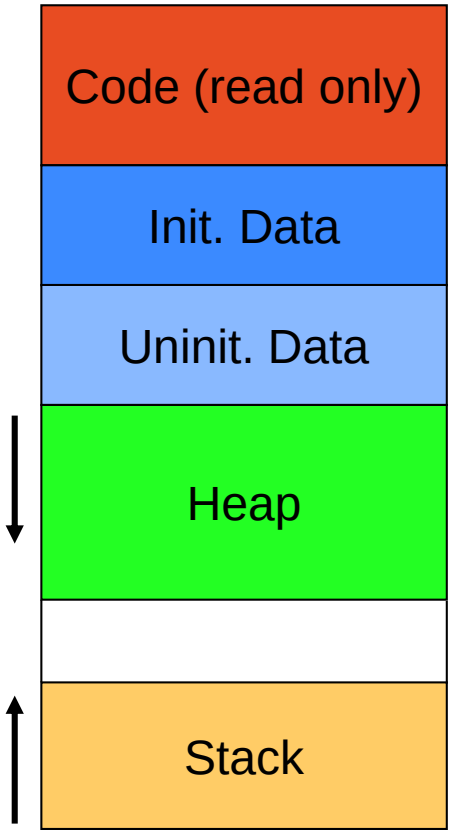
32-bit Architecture? 64-bit?

- What does it mean?
- 32-bit Arch
  - Integers and data registers are ~32-bits wide, etc.
  - 2^32 memory addresses (4GibiByte of addressable space)
  - Gibi vs Giga?
- 64-bit Arch
  - Integers and data registers are ~64-bits wide, etc.
  - 2^64 memory addresses (16 Exbibytes of ***address space***)
    - kilo < mega < giga < tera < peta < **exa** < zetta < yotta
- Do we have so much memory?

Virtual Memory & Addressing

Memory Image of a Process

- Process: A program in execution, containing
  - code
  - data (initialized and uninitialized)
  - heap (for dynamically allocated data)
  - stack (for local variables, function parameters, …)



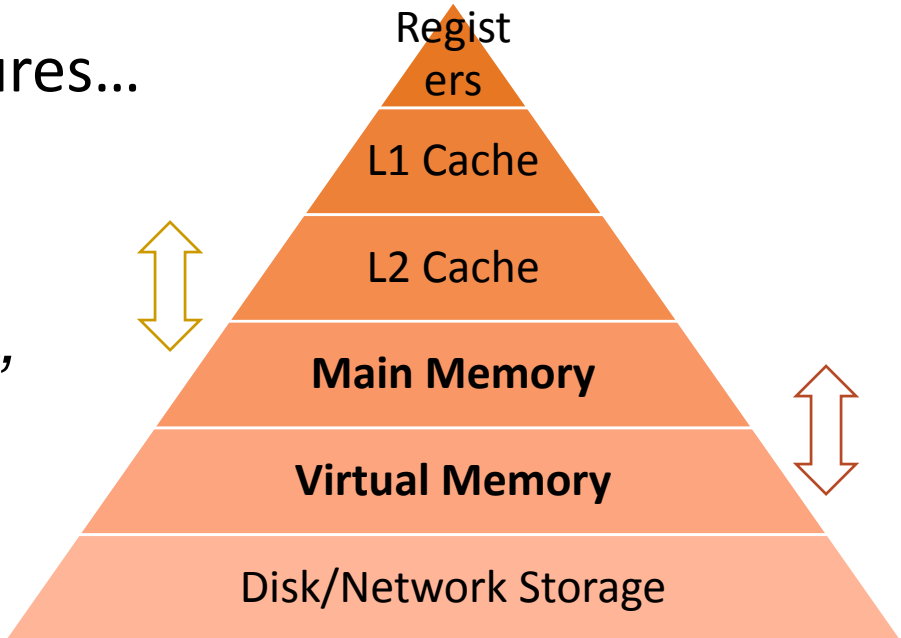
Memory Abstraction

- Above scheme requires entire process to be in memory
- What if process space is larger than physical memory?

Virtual Memory

- Provides an abstraction to physical memory
- Allows execution of processes not completely in memory
- Offers many other features…

Main memory is a "cache" for the virtual memory



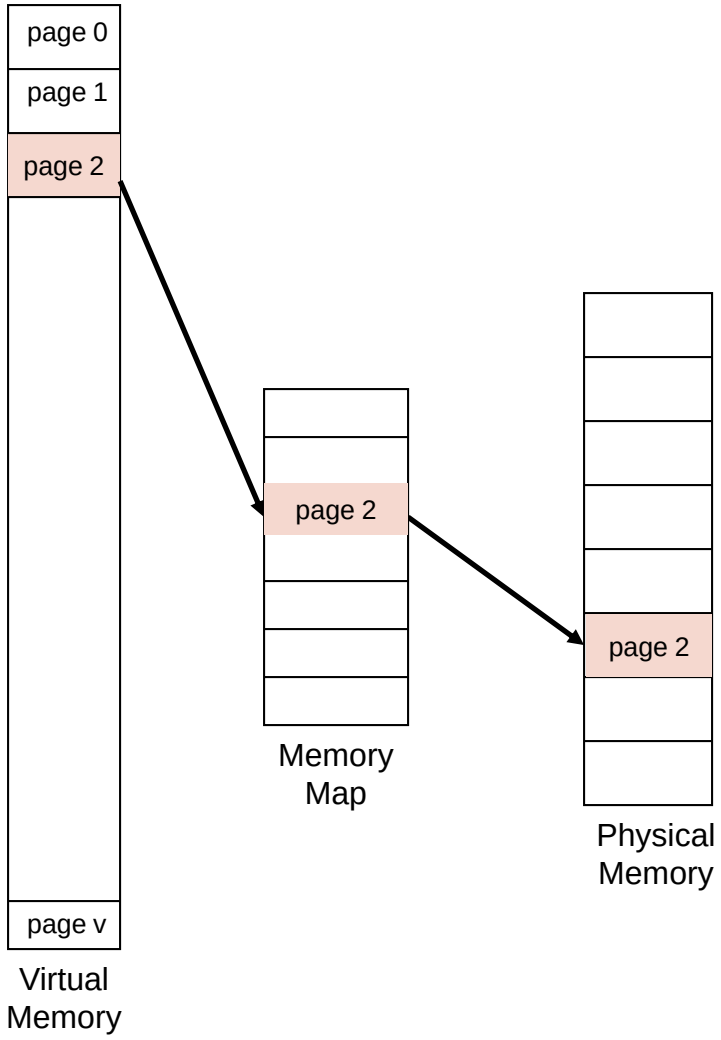
Virtual Memory Concepts

- Instructions must be in physical memory to be executed
- But not all of the program need to be in physical memory

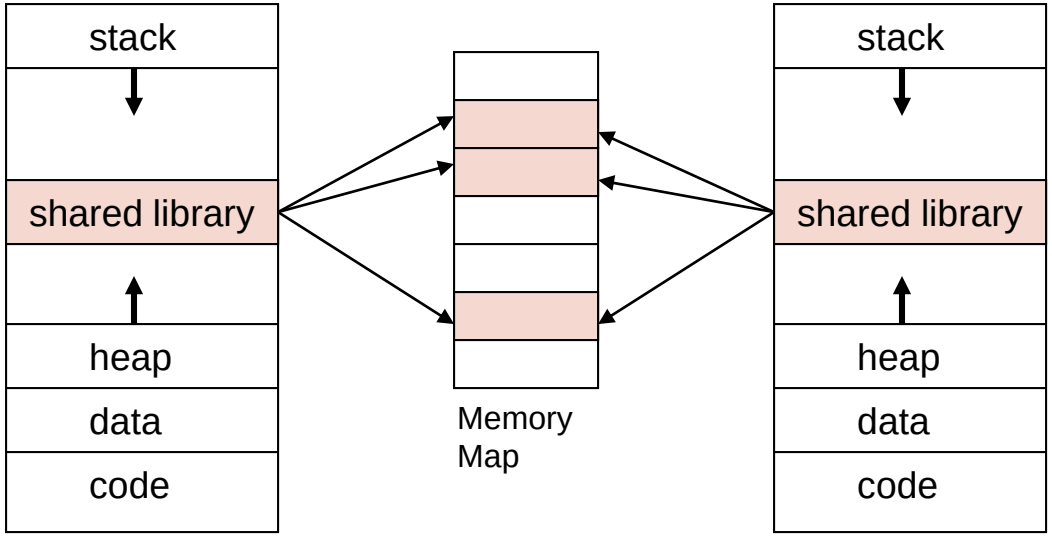
- Each program could take less memory; more programs can reside simultaneously in physical memory
- Need for memory protection
  - One program should not be able to access the variables of the other
  - Typically done through address translation (more later)

Virtual Memory Concepts – Large size

- Granularity of virtual memory management is a *Virtual Page* of 2^p bytes in size
- Fixed size unit of contiguous memory locations
- Corresponds to a *Physical Page* or *Page Frame* of same size
- Virtual memory can be extremely large when compared to a smaller physical memory



Virtual Memory Concepts - Sharing



- Virtual memory allows programs to share pages and libraries
  - Different Virtual Memory pages holding shared libraries point to the same physical memory page

Address Translation

- Virtual and Physical Addresses
- Virtual memory addresses translated to physical memory addresses
- Memory Management Unit (MMU) provides the mapping
  - MMU – a dedicated hardware in the CPU chips
  - Uses a lookup table

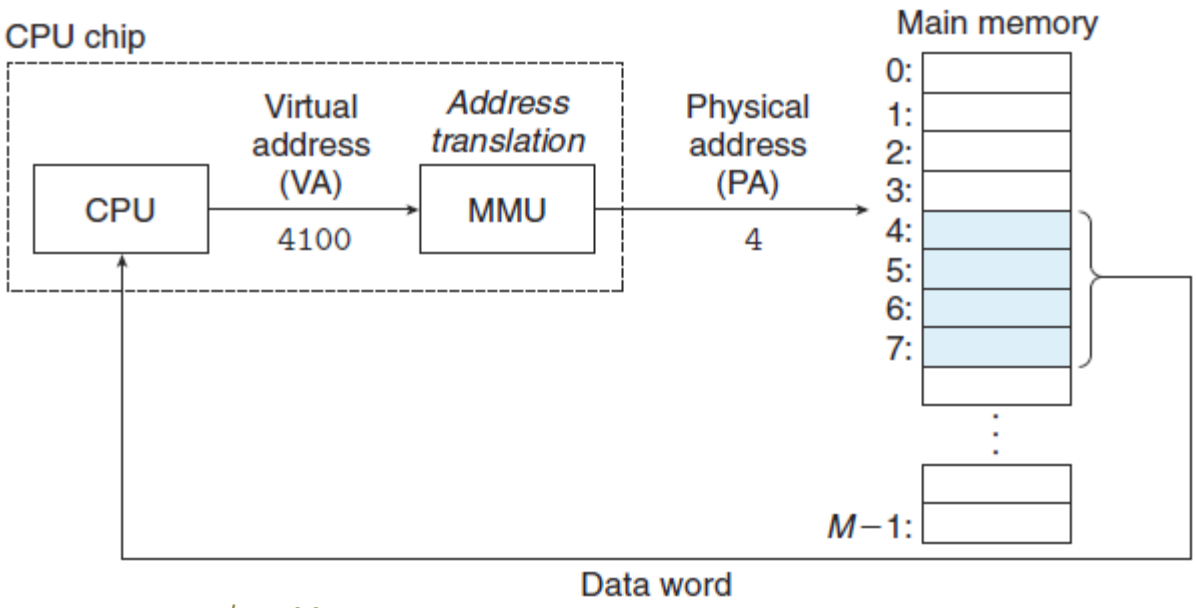


Fig 9.2 (Pg. 778, Bryant, 2nd Ed.)

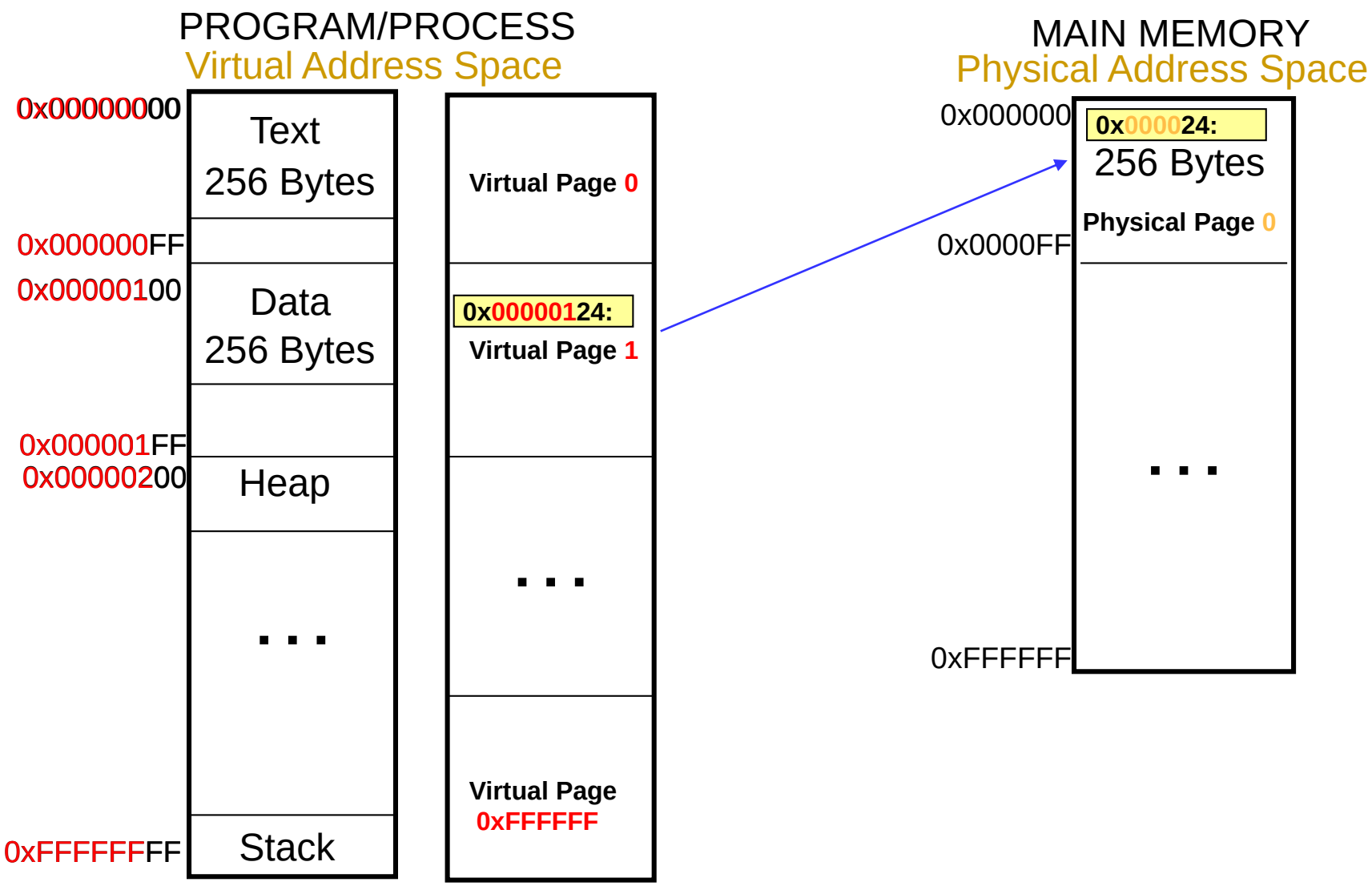
Address Translation

- To translate a virtual address to the corresponding physical address, a table of translation information is needed
- Minimize size of table by not managing translations on byte basis but a larger granularity

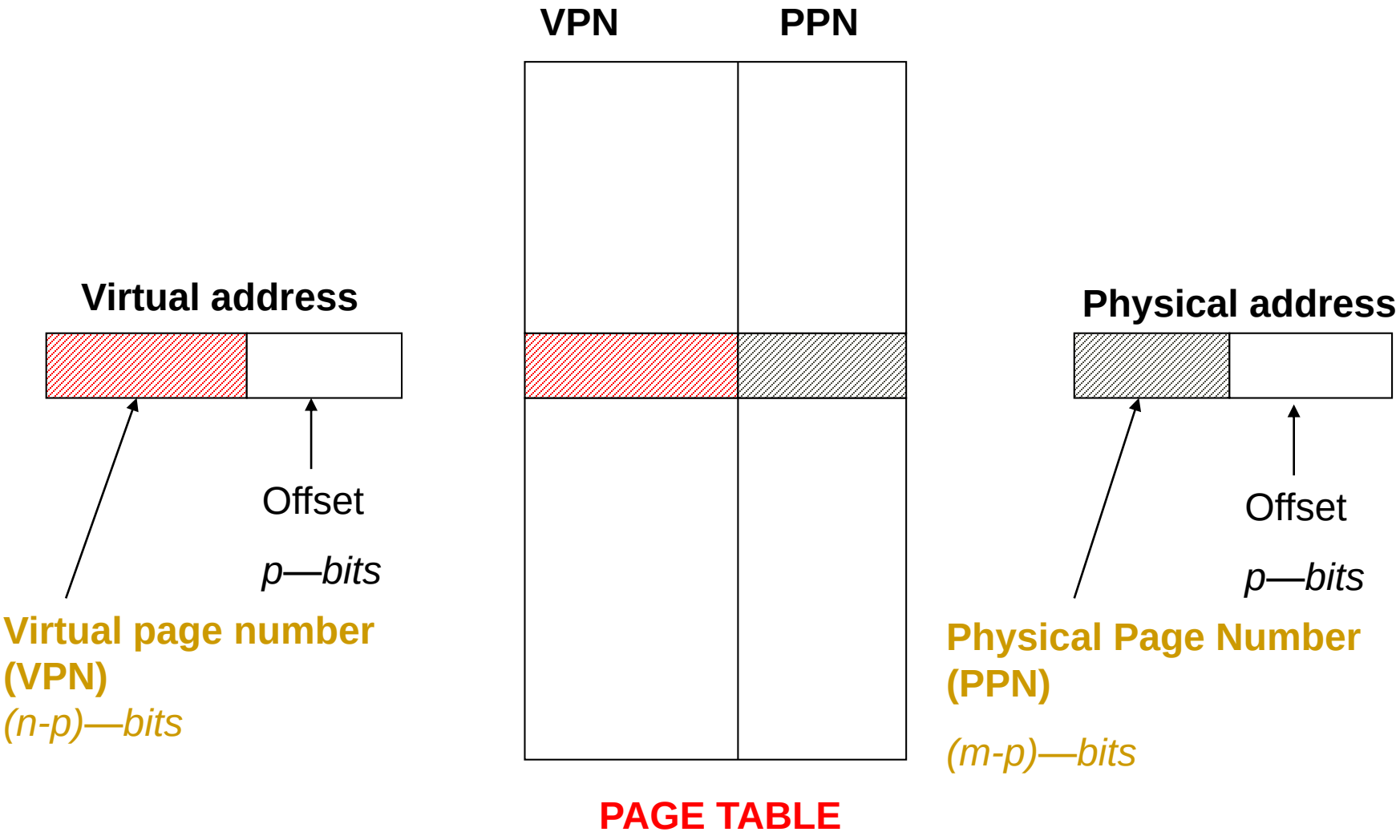
- **Page** is the unit of translation information is maintained
- The resulting table is called **page table**
- The contents of the page table managed by OS

- Thus, (address translation logic in MMU + OS + page table) used for address translation

Address Translation



Address Translation



Size of Page Table

$$\frac{2^{32}}{2^{14}}$$

- Big Problem: Page Table size
  - Example: 32b virtual address @ 16KB page size
  - 256K virtual pages → MB page table size per process
  - Has to be stored in memory
- Solution – Multi-level page table (more later)

Address Translation

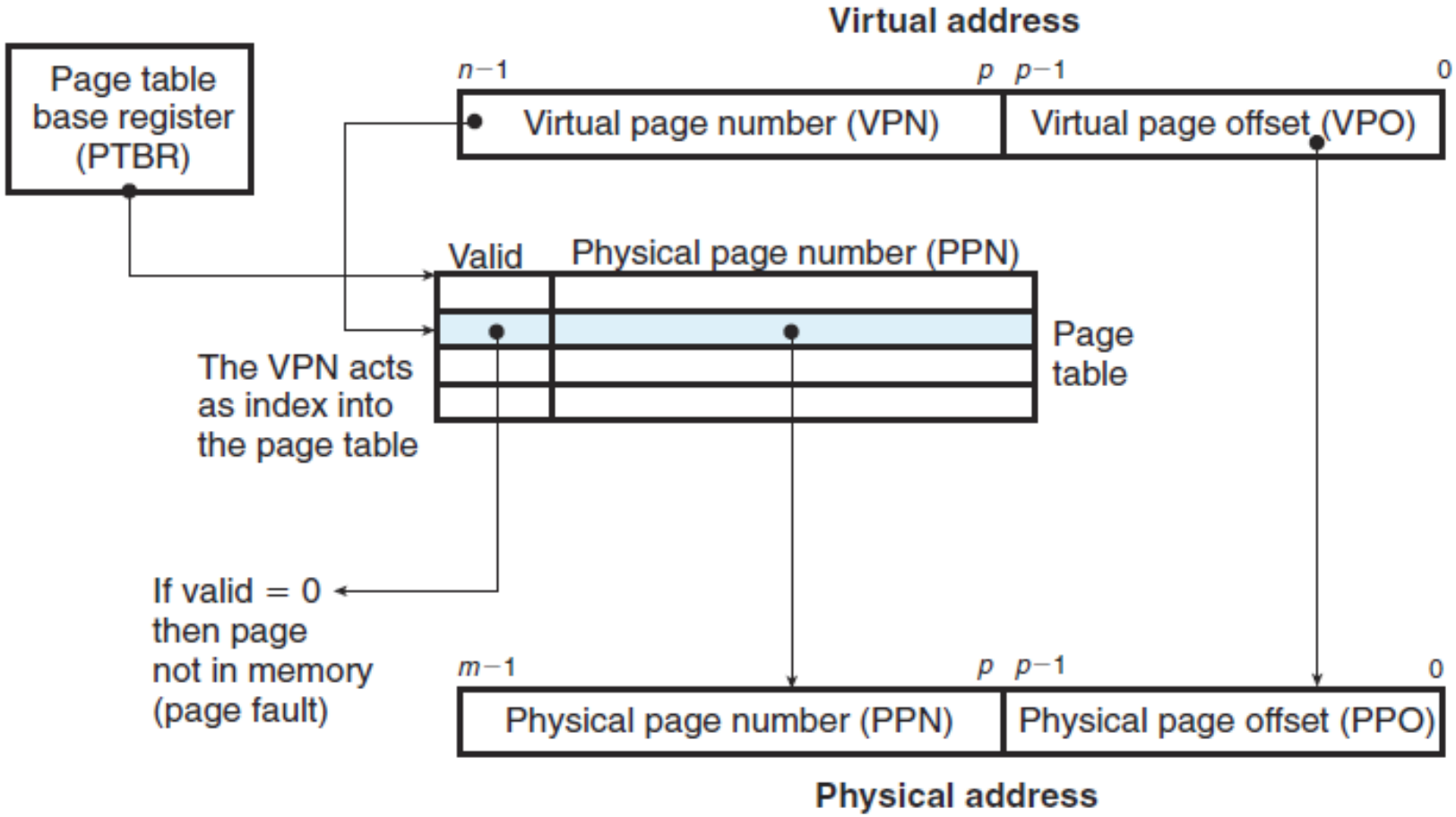
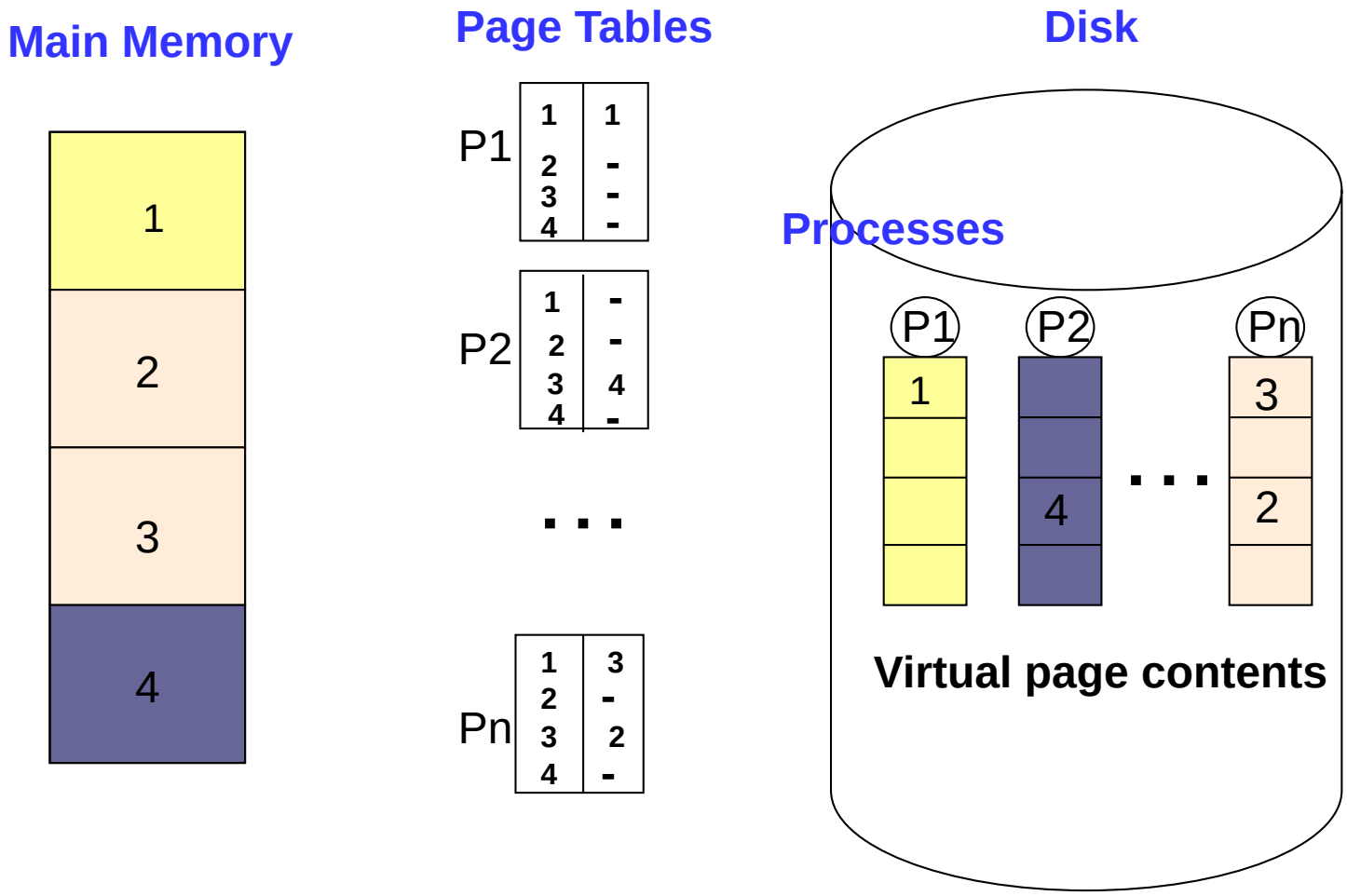


Fig 9.12 (Pg. 789, Bryant, 2nd Ed.)

What's happening...



Recommended Reading

- Chapter 9: Virtual Memory, Bryant 2nd Ed.

Page Faults & Replacements

Demand Paging

- When a program refers a page, the page table is looked up to obtain the corresponding physical page
- If the physical frame is not present, page fault occurs
- The page is then brought from the disk
- Pages are swapped in from the disk to the physical memory only when needed – *demand paging*

- For this, the page table has valid and invalid bits
  - Valid – present in memory
  - Invalid – present in disk

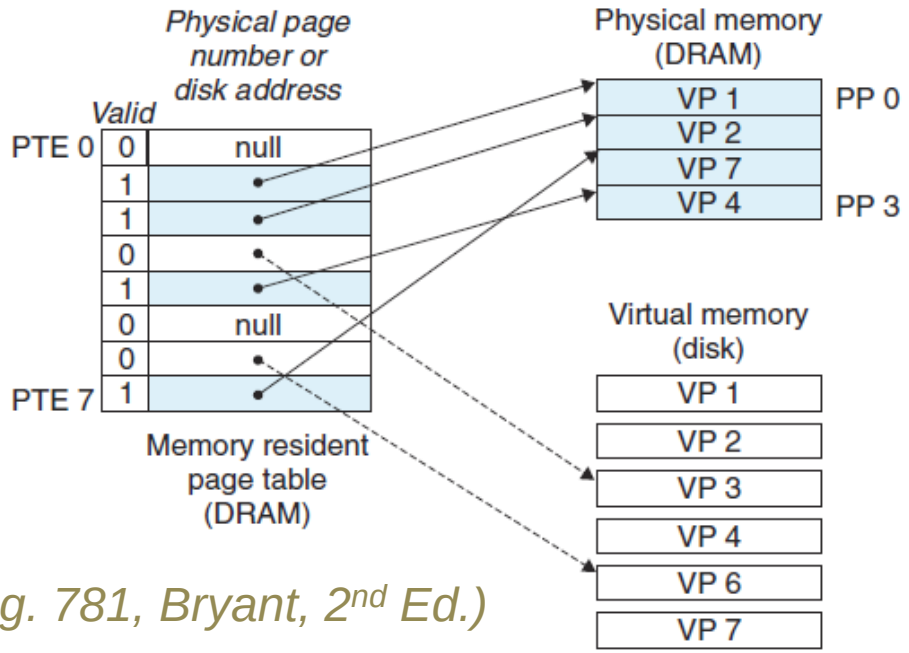


Fig 9.4 (Pg. 781, Bryant, 2nd Ed.)

Page Fault

- Virtual address generated by processor is not available in main memory
- Detected on attempt to translate address
  - Page Table Entry is invalid
- Must be `handled' by *operating system*
  - Identify slot in main memory to be used
  - Get page contents from secondary memory
    - **Part of disk can be used for this purpose**
  - Update page table entry
- Data can now be provided to the processor

Demand Paging w/ valid Bit

Page Hit

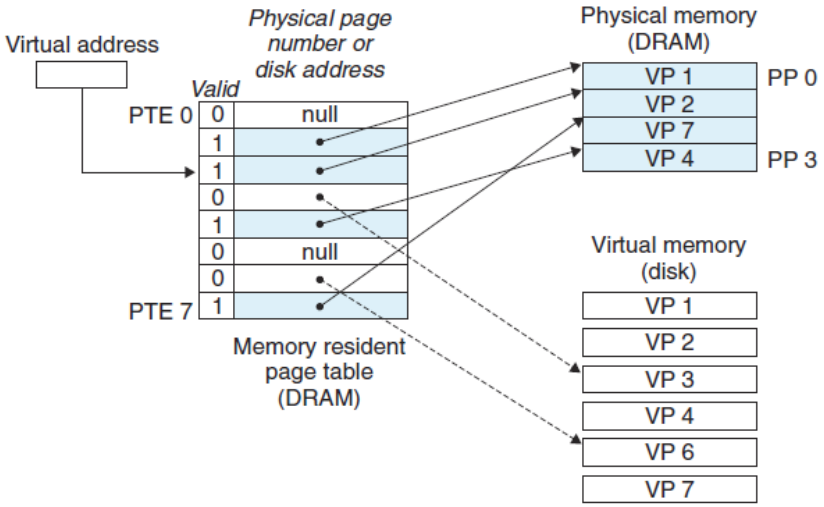
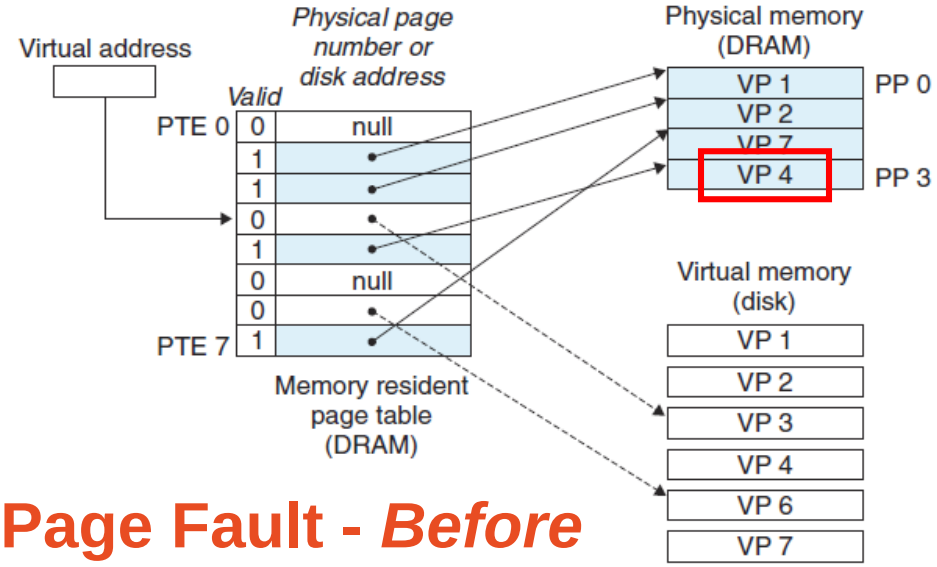
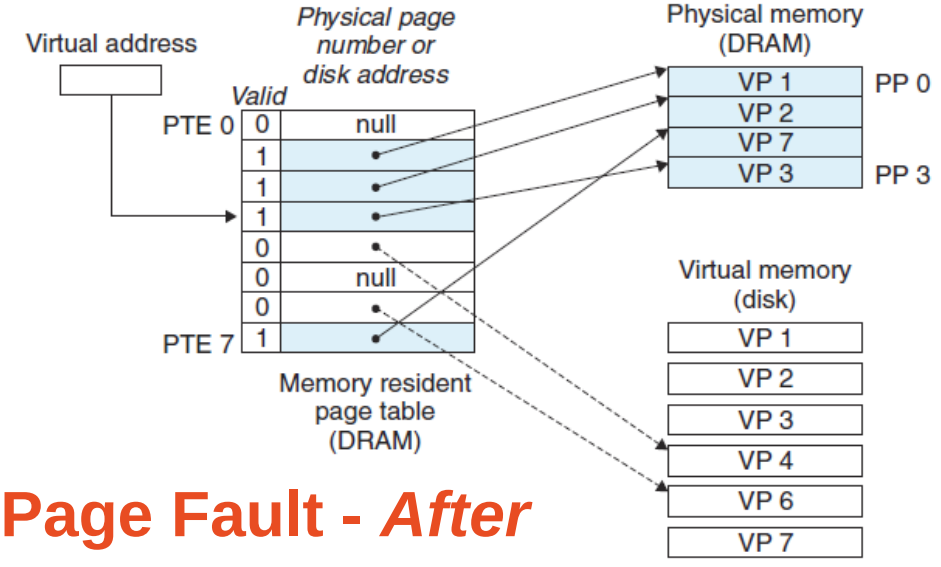


Fig 9.5-9.7 (Pg. 782-783, Bryant, 2nd Ed.)



Page Fault - *Before*



Page Fault - *After*

Page Fault Handling Steps

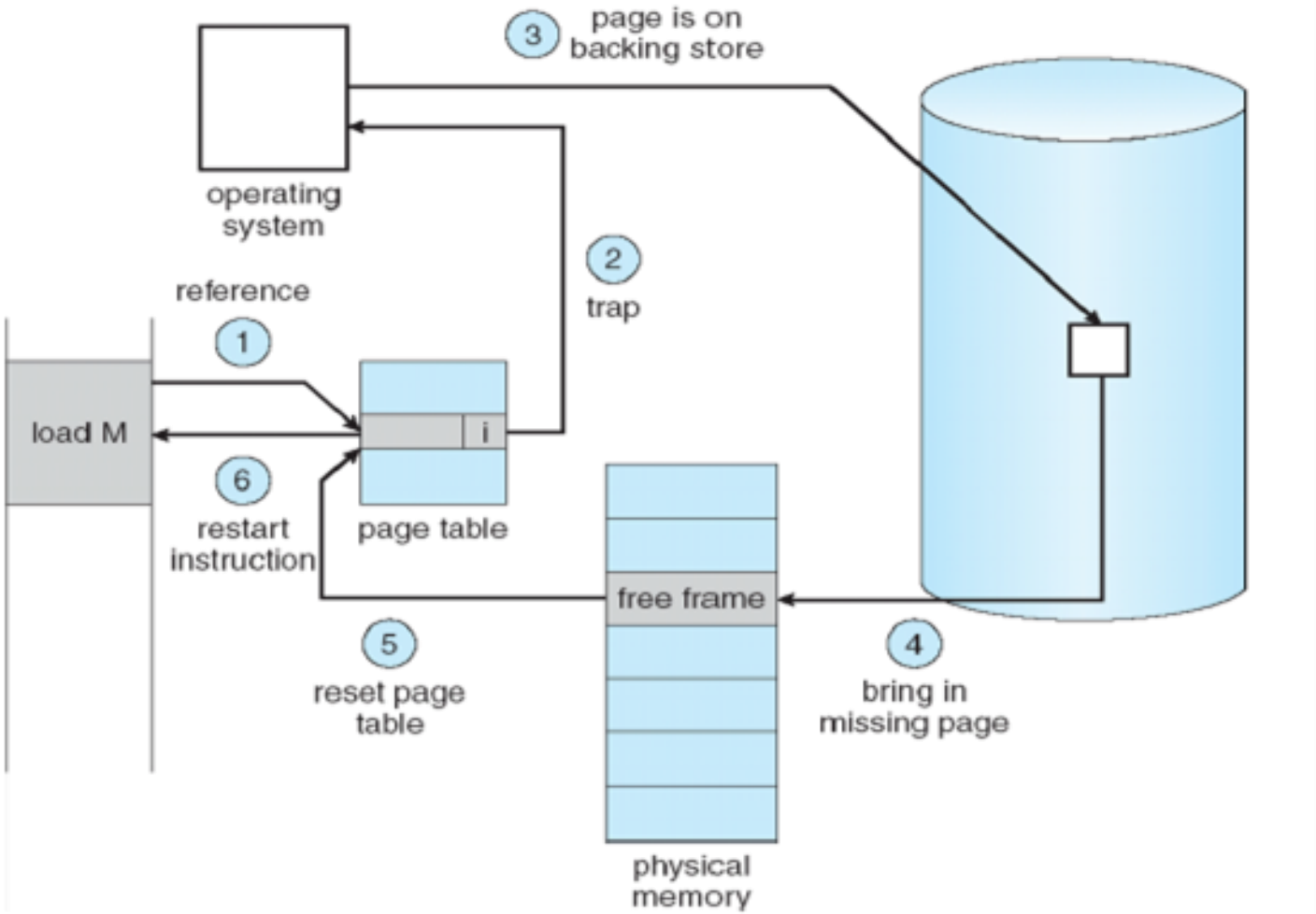
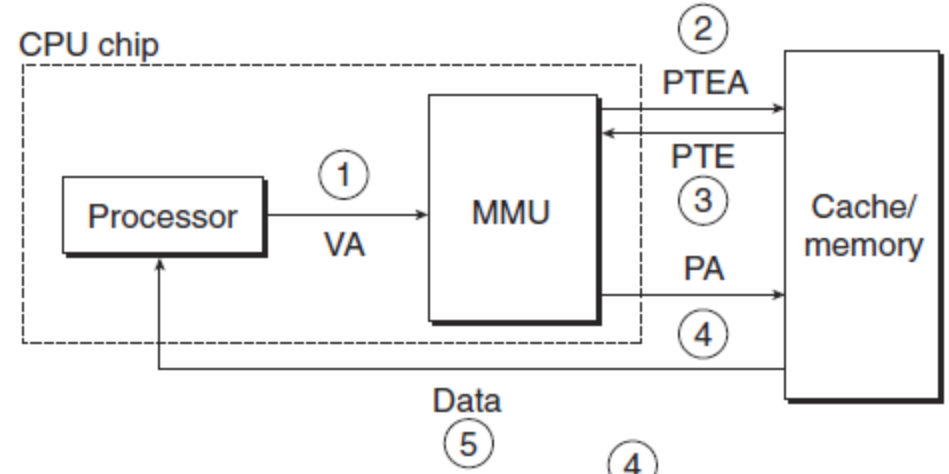


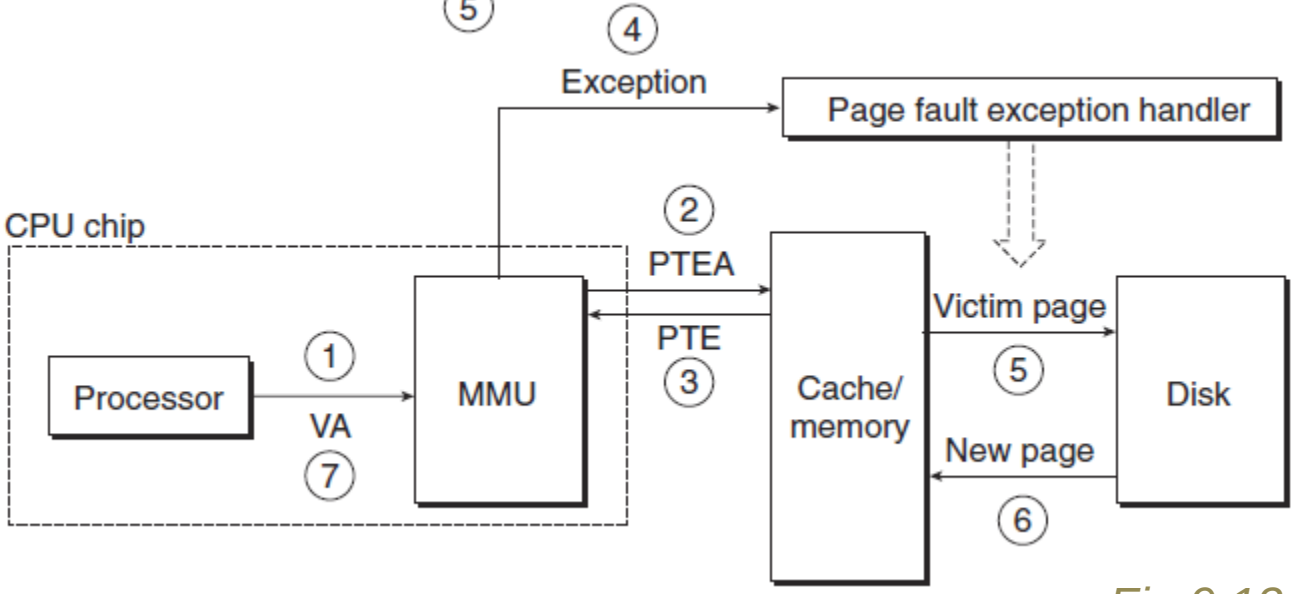
Fig. 9.6 (© Silberschatz, 7th Ed., Pg. 321)

Page Fault Handling – *A different perspective*



Page Hit

- Page Hit handled fully in hardware.
- Page Fault required OS intervention.



Page Fault

(5) Write dirty victim page back to disk

Fig 9.13 (Pg. 790, Bryant, 2nd Ed.)

Page Fault Handling Steps

1. Check page table to see if reference is valid or invalid
2. If invalid, a trap into the operating system
3. Find a free page frame, locate the virtual page on the disk
4. *Swap in* the virtual page from the disk
    1. Replace (*Swap out*) an existing page (page replacement)
5. Modify page table
6. Restart instruction

Performance Impact of Page Faults

- **p** – probability of page fault
- **ma** – memory access time
- Effective Access Time

$$\mathbf{EAT} = (1-p) \times ma + p \times page\_fault\_time$$

- E.g. EAT $= (1-p) \times (200\ nanosecs) + p \times (8\ millisecs)$

$$= 200 + 7{,}999{,}800 \times p\ nanosecs$$

➲ Dominated by page fault time

Where is the time spent?

1. Trap to OS, Save registers and process state, Determine location of page on disk
2. Issue read from the disk
    1. major time consuming step – 3 milliseconds latency, 5 milliseconds seek
3. Receive interrupt from the disk subsystem, Save registers of other program
4. Restore registers and process state of this program

- To reduce EAT due to page faults (< 10%), only 1 in few 100k memory access should page fault
    - *i.e., most memory references should be pages hits*
- But how do we manage this? *Locality!*

Page Replacement Policies

- Question: How does the page fault handler decide which main memory page to replace when there is a page fault?
- **Principle of Locality of Reference**
  - A commonly seen property of programs
  - If memory address ***A*** is referenced at time ***t***, then **it** and **its neigbhouring** memory locations are likely to be referenced in the **near future**
  - Suggests that a Least Recently Used (LRU) replacement policy would be advantageous

Spatial Locality

Temporal Locality

Locality of Reference

- Based on your experience, why do you expect that programs will display locality of reference?

	Same address (temporal)	**Neighbours (spatial)**
Program	Loop Function	Sequential code Loop
Data	Local Loop index	Stepping through array

Page Replacement Algorithms

- **FIFO** – Performance depends on if the initial pages are actively used
- **Optimal page replacement** – Replace the page that *will not be used for the longest time*
  - Difficult to implement
  - Some ideas?
- **LRU**
  - Least Recently Used is most commonly used
  - Implemented using counters
  - But LRU might be too expensive to implement

Page Replacement Algorithms

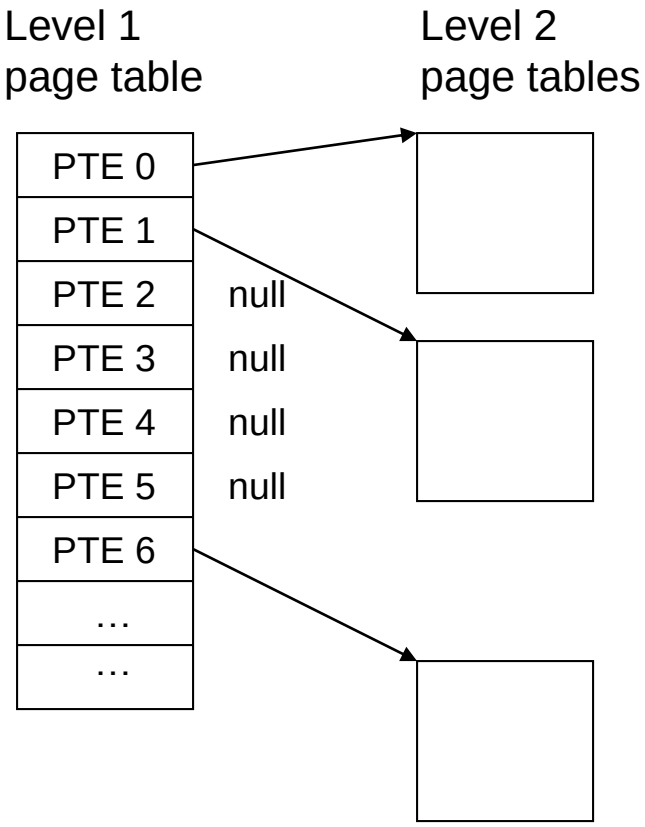
- **LRU Approximation** – Second-Chance or Clock algorithm
  - Similar to FIFO, but when a page's reference bit is 1, the page is given a second chance
  - Implemented using a circular queue
- **Counting-Based Page Replacement**
  - LFU, MFU
- Page replacement Performance depends on applications
  - See Silberschatz, Section 9.4.8

Thrashing

- Global page replacement algorithms
  - Multiple processes keep alternatively swapping out each others' frequently used pages
- Increased multi-programming
  - Low CPU usage causes more processes to be scheduled and more frames to be used actively (vicious cycle)
- Program performance drops dramatically!
  - Key aspect of performance profiling

Multi Level Page Tables

- A page table can occupy significant amount of memory
- Multi level page tables to reduce page table size
- If a PTE in Level 1 PT is null, the corresponding Level 2 PT does not exist
- Only Level 1 PT needs to be in memory. Other PTs can be swapped in on-demand.



Multi Level Page Tables (k)

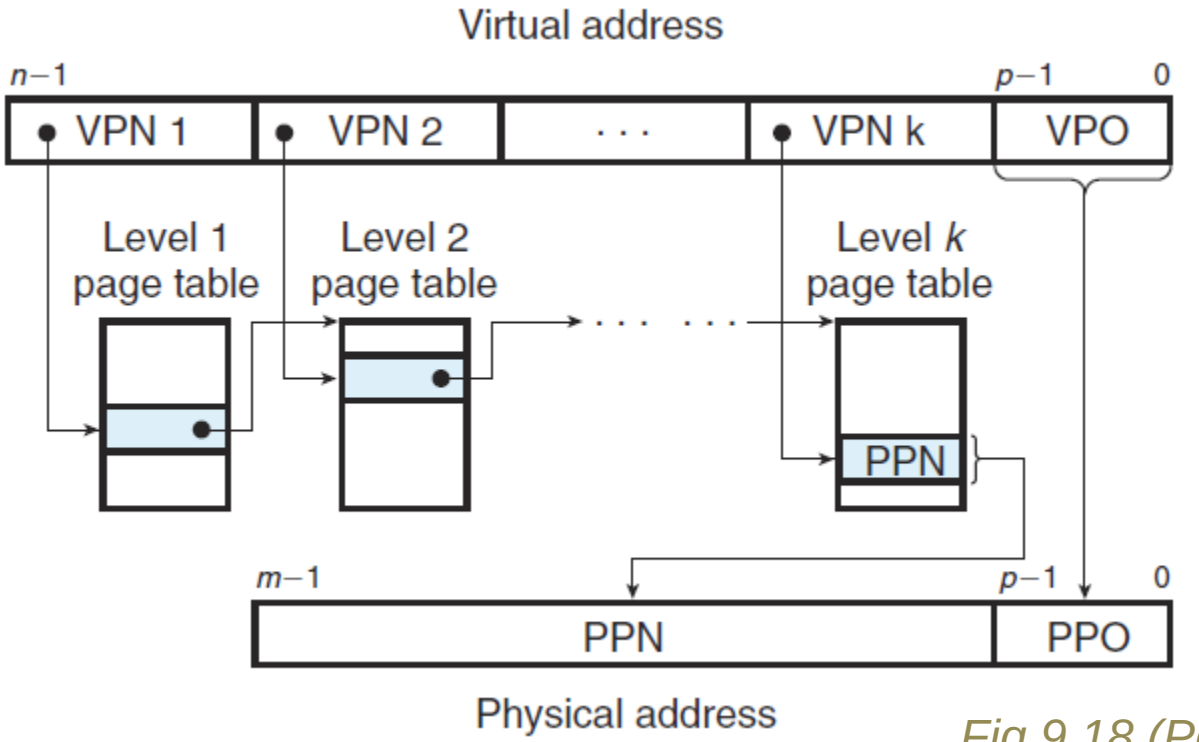


Fig 9.18 (Pg. 795, Bryant, 2nd Ed.)

- Each VPN *i* is an index into a PT at Level *i*
- Each PTE in a level j PT points to the base of some PT at level (j+1)

Virtual Memory and Fork

- During fork, a parent process creates a child process
- Initially the pages of the parent process is shared with the child process
- But the pages are marked as copy-on-write.
- When a parent or child process writes to a page, a copy of a page is created.
  - Copy only pages that are written to
  - Copy just in time

Before Process 1 Modifies Page C

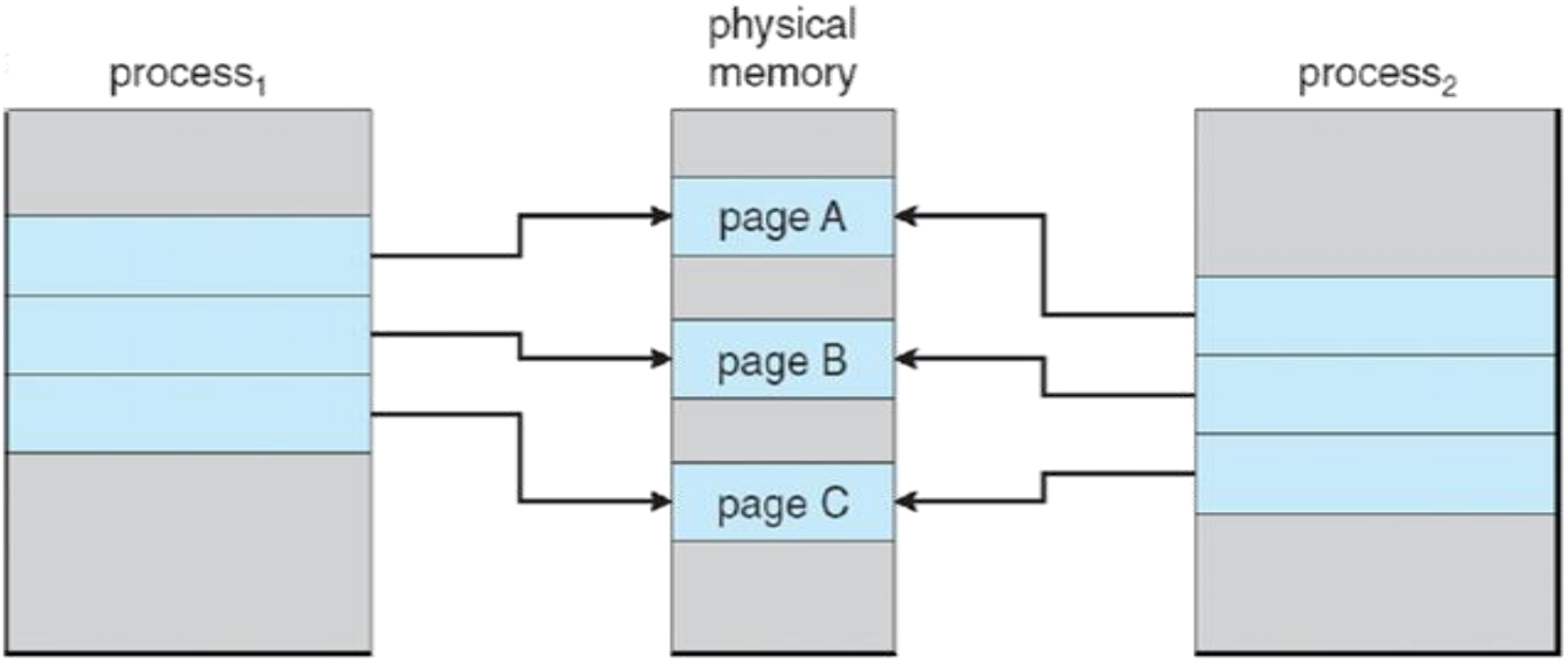


Fig. 9.7 (© Silberschatz, 7th Ed., Pg. 326)

After Process 1 Modifies Page C

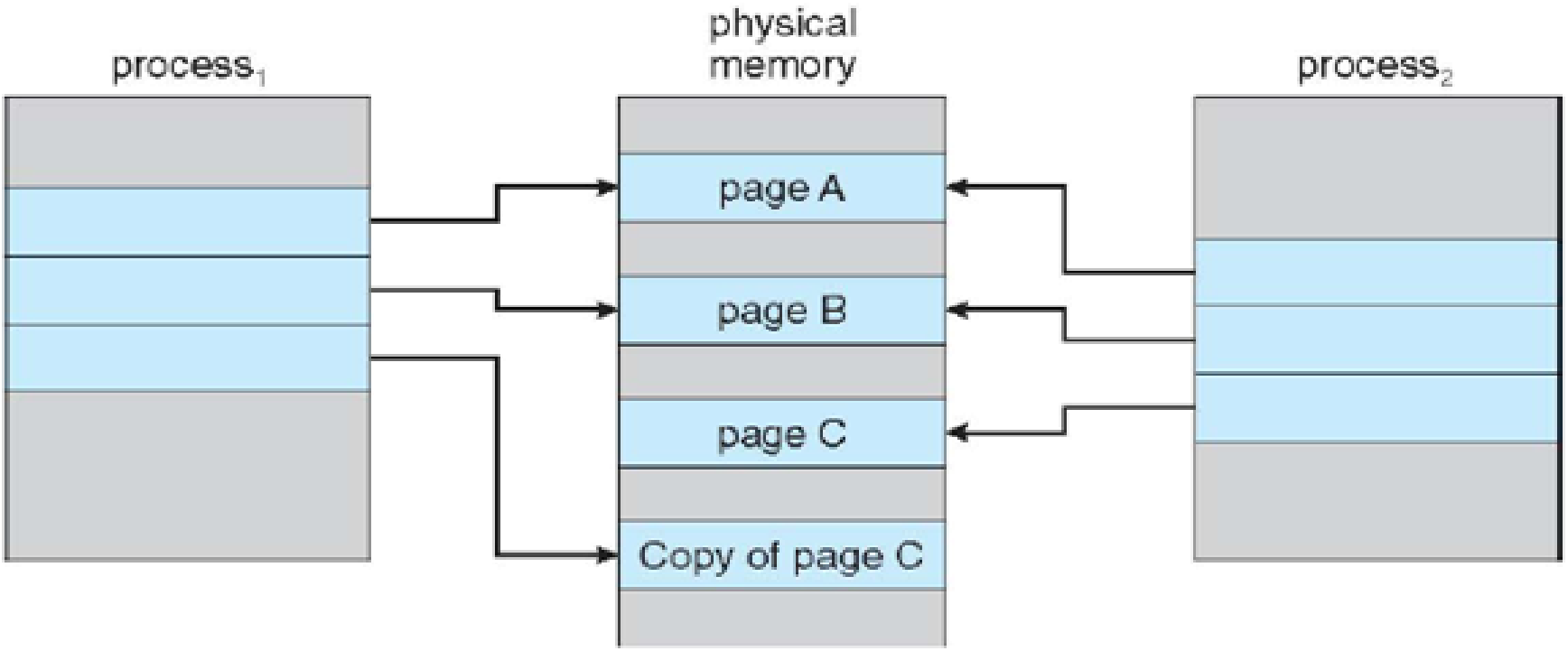


Fig. 9.8 (© Silberschatz, 7th Ed., Pg. 326)