COMMONWEALTH OF AUSTRALIA
Copyright Regulation 1969

WARNING

This material has been copied and communicated to you by or on behalf of **Curtin University of Technology** pursuant to Part VB of the *Copyright Act 1968* (**the Act**)

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

Operating Systems
COMP2006

Memory Management
Lecture 6 and 7

Memory Management

References:

Silberschatz, Galvin, and Gagne, *Operating System Concepts*, Chapter 8 and Chapter 9

Topics:

- Logical and Physical Address Space, and Swapping.
- Contiguous Allocation, Paging, and Segmentation
- Virtual Memory and its complexity and cost for demand paging.
- Page replacement algorithms.
- Frame allocation and algorithms.
- Thrashing.
- Other considerations for memory management

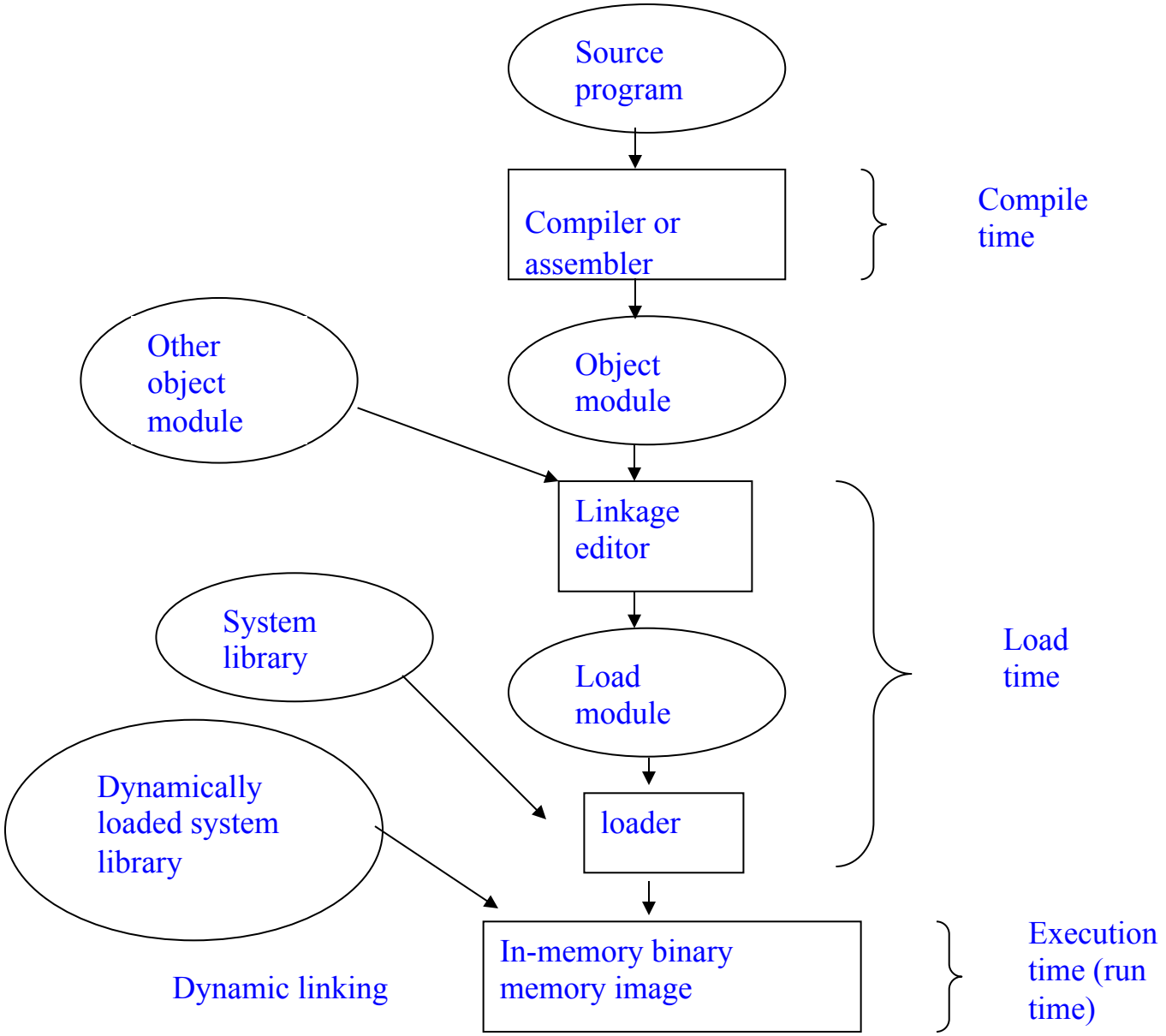
Main Memory – Chapter 8

- Memory is a large array of words or bytes, each with its own address
  - It is a repository of quickly accessible data shared by CPU and I/O devices.
  - Memory and registers are the only storage that CPU can directly access
- Main memory is a volatile storage device
  - It loses its contents in the case of system failure.
- A program must be mapped to absolute addresses and loaded into memory.
- Selection of a memory management scheme for a specific system depends on many factors, especially on the hardware design of the system.
- The OS is responsible for the following activities:
  - Keep track of which parts of memory are being used and by whom.
  - Decide which processes to load next when memory space becomes available.
  - Allocate and deallocate memory.

MEMORY MANAGEMENT (cont.)

- A program that resides on disk as binary executable file must be brought into the memory within a process to be executed.
- To improve both the CPU utilization and speed, the computer must keep several processes in memory.
  - We must share the memory, and hence we need memory management scheme.
  - Most systems allow a user process to reside in any part of physical memory.
    - Each process has a separate memory area to protect one process from the other
- Addresses in the source program are generally symbolic, such as a variable *count*.
  - A compiler will *bind* these symbolic addresses to re-locatable addresses
    - e.g., 14 locations from the start of this module
  - A loader/linkage editor binds each relocatable address to an absolute address
    - e.g., the start of this module is mapped to absolute address 10000, 14 locations from it is to 10014, etc
- *Binding* is a mapping from one address space to another.

Binding of instructions and data to memory addresses



Several types of binding time

- **Compile time:** if memory location is known a priori, *absolute* code can be generated
  - Must recompile code if starting location changes.
  - Process must be loaded at the same place in memory.
- **Load time**: binding is delayed until code loaded into memory.
  - Must generate *relocatable* code if memory location is not known at compile time.
  - Process must be reloaded if the starting address changes
- **Execution time**: binding is delayed until run time.
  - The process can be moved during its execution from one memory segment to another.
  - Needs special hardware support for address mapping (e.g., *base* and *limit* registers).

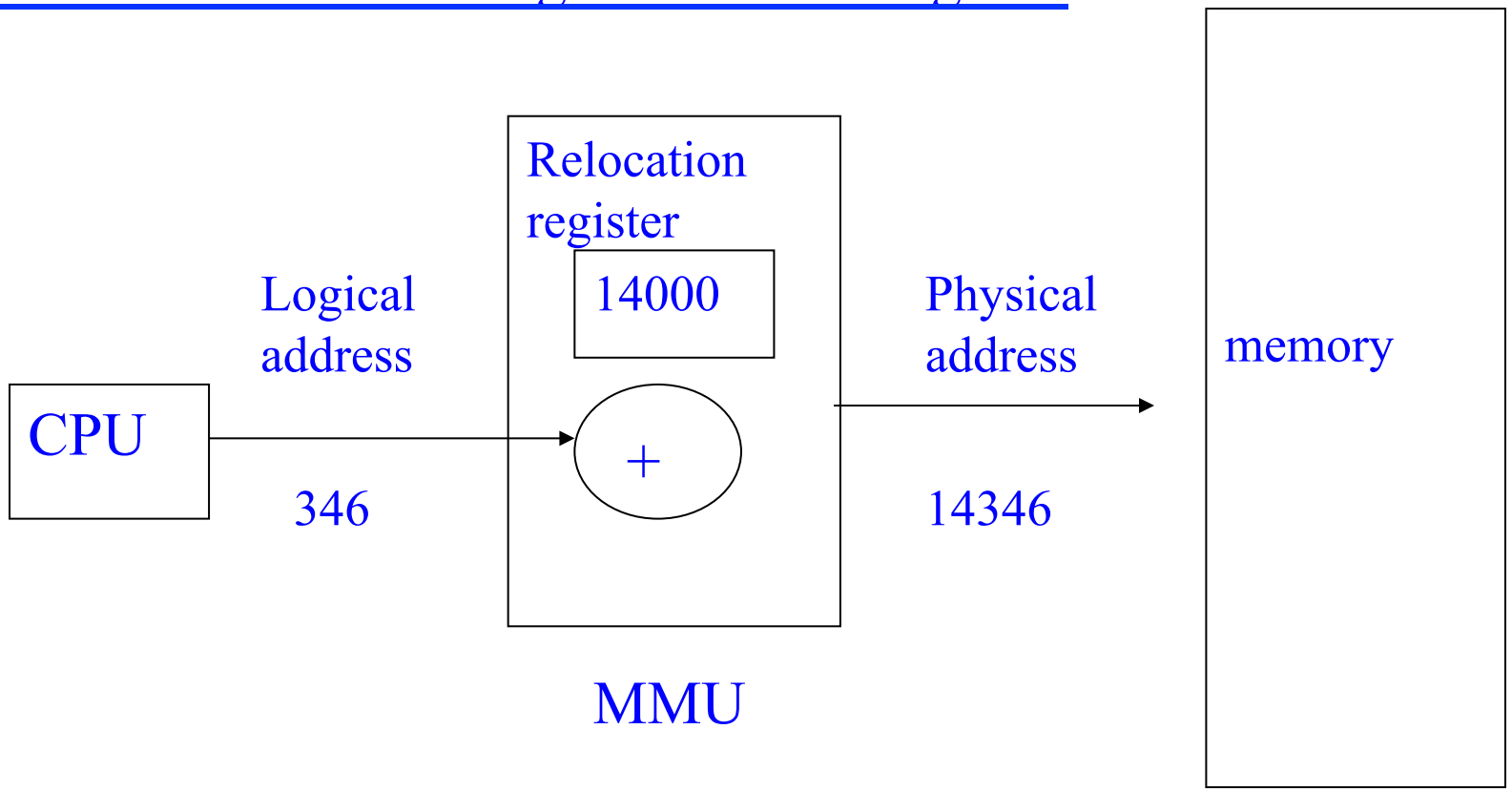
Logical vs. Physical Address Space

- The concept of a *logical address space* that is bound to a separate *physical address space* is critical in memory management.
  - *Logical address* – generated by the CPU.
    - also called as *virtual address.*
  - *Physical address* – address seen by the memory unit.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes.
- Logical and physical addresses are different in execution-time address binding scheme.
  - The run-time mapping from virtual to physical addresses is done by the memory management unit (MMU) – a hardware device.

Memory-management unit (MMU)

- In MMU, the value in the **relocation register** is added to every address generated by a user process when it is sent to memory.
  - The user program deals with logical addresses.

Dynamic relocation using a relocation register



Dynamic loading and linking

Dynamic loading: Routine is not loaded until it is called.

- To have a better memory-space utilization.
  - unused routine is never loaded; it is kept in disk in a relocatable format.
- Useful when large amounts of code are needed to handle infrequently occurring cases.
- No special support from the OS is required.
  - implementation through program design, but OS provides library routines to use dynamic loading.

Dynamic linking: Linking is delayed until execution time.

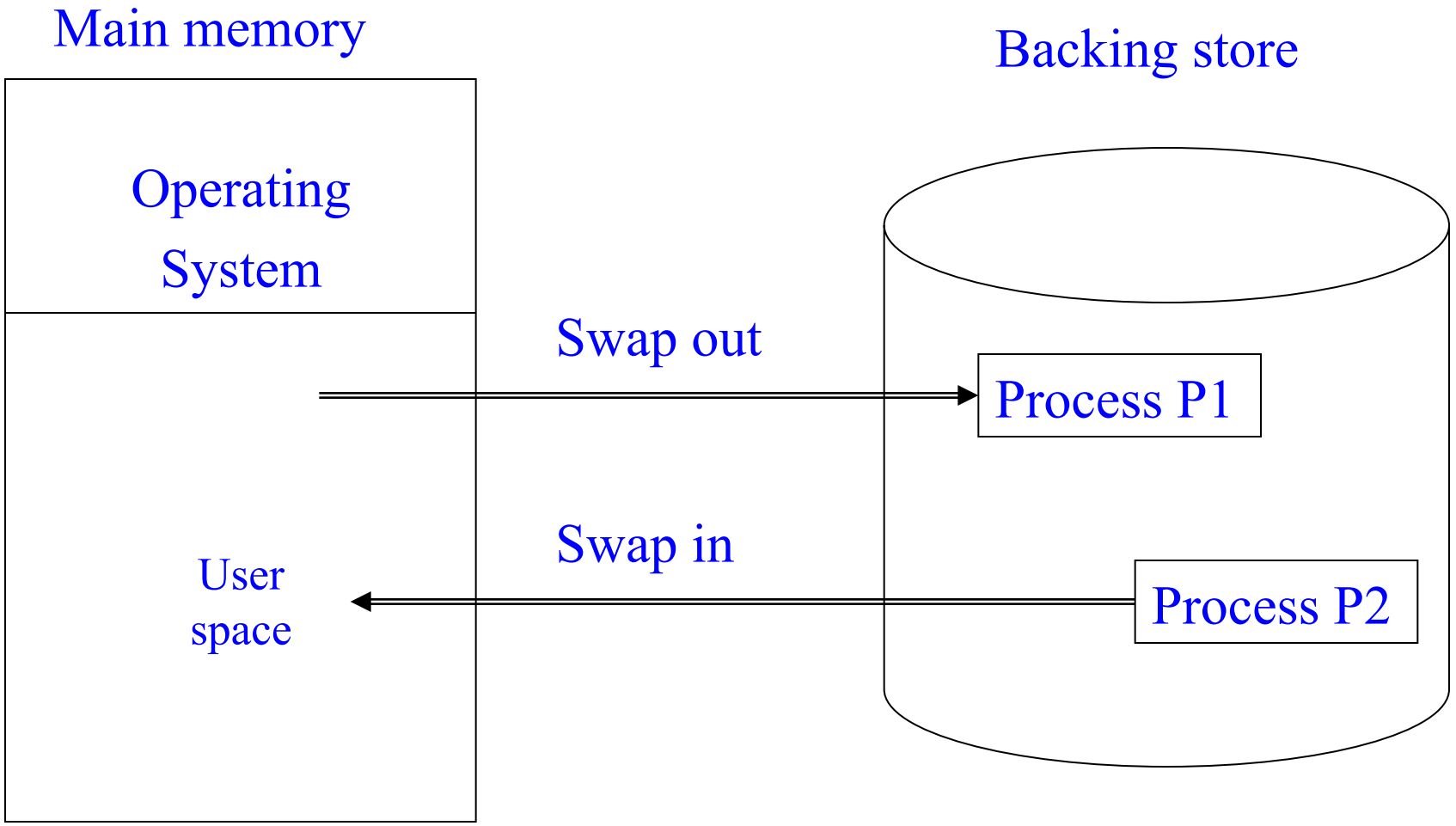
- Used for system libraries
  - Without this feature, each user program must include a copy of system libraries
- Small piece of code, *Stub*, is used to locate the appropriate memory-resident library routine.
  - Stub replaces itself with the address of the routine, and executes the routine.
- OS needs to check if the routine is in process' memory address.

Swapping

- A process can be swapped temporarily out of memory to a *backing store*, and then brought back into memory for continued execution
  - Backing store – a fast, large disk to store copies of **all** memory images for all users
  - Must provide direct access to these memory images
- When CPU scheduler selects a process to run, it calls dispatcher which checks if the process is in memory
  - If not, if there is not sufficient free memory area, the dispatcher swaps out one or more processes from memory, and use the space for the process.
- Major part of swap time is transfer time
  - Total transfer time is directly proportional to the amount of memory swapped
  - E.g., to swap 100MB to or from disk with transfer rate of 50MB/s takes 2 seconds.
- In the compile or load time binding, a process that is swapped out will be swapped back into the same memory space that it used before
- Potential problem may occur if the swapped out process is in the middle of doing I/O

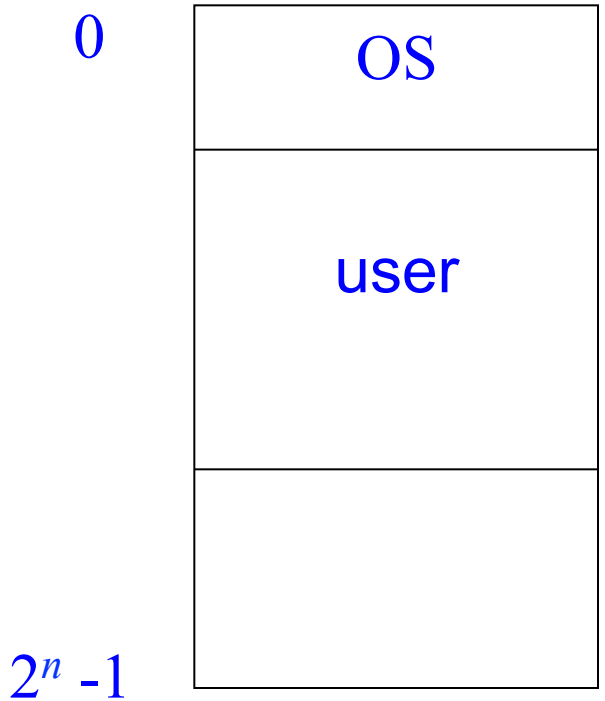
  Solution:
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

Schematic View of Swapping



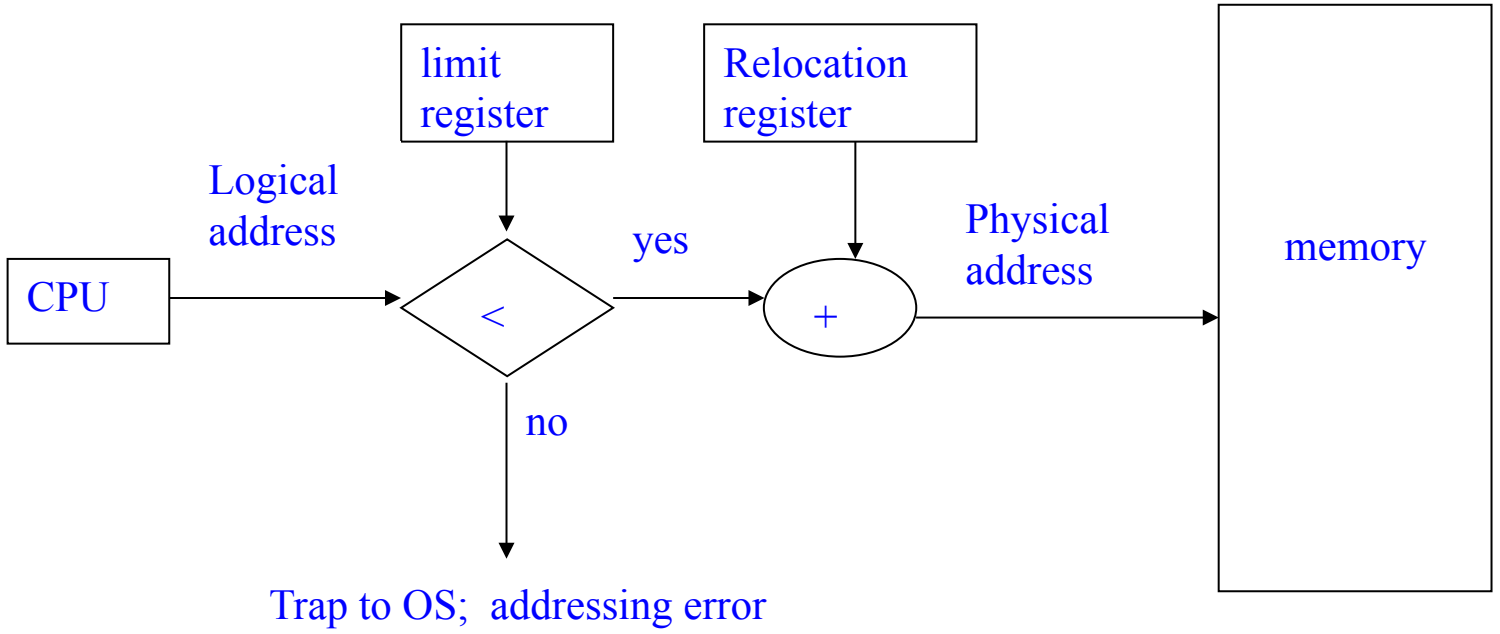
Contiguous Allocation

- Main memory is usually divided into two partitions:
  - Resident Operating System, usually is held in low memory with interrupt vector
  - One or more user processes, are held in high memory
- Contiguous allocation: each process is in one contiguous memory location
  - Used in early/old system



Memory Mapping and Protection

- Mapping and protection can be done using relocation and limit registers
  - Relocation register contains the value of the smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - The relocation-register scheme is used to protect user processes from each other, and from accessing OS memory
  - The registers are loaded by dispatcher as part of context switch



Multiple-partition allocation

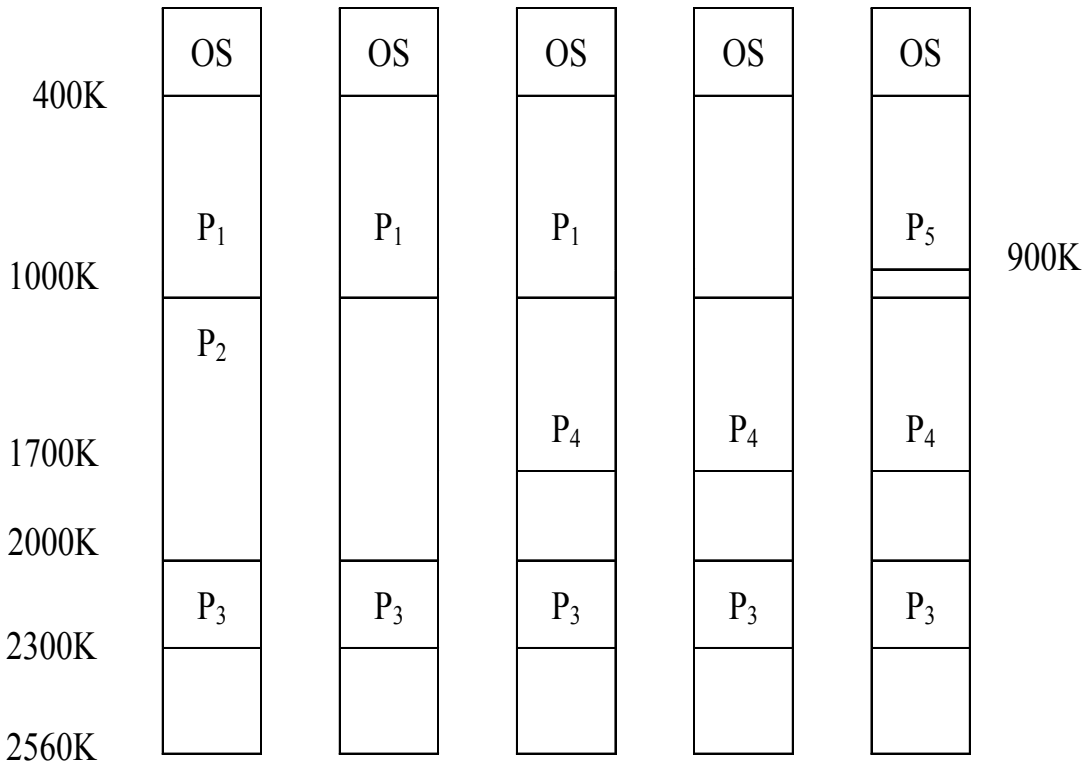
- Memory can be divided into some fixed-sized partitions.
  - One partition is used for each process; no longer used.
- Memory can also be divided into variable partitions.
  - OS maintains information about Allocated and Free partitions (holes) in a table.
    - A hole is a block of available memory
    - holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it.

Example

Scheduling example:

Job queue		
Process	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

Memory allocation (FCFS)



- There are three available holes of size 100K, 300K, and 260K
  - A process of size 400K cannot run although the system has a total of 660K available memory
  - Called external fragmentation.

Dynamic storage-allocation problem

- How to satisfy a request 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.
  - **Worst-fit:** Allocate the largest hole.
    - must also search entire list. Produces the largest leftover hole.
- Simulation has shown that first-fit and best-fit are better than worst-fit in terms of speed and storage utilization.

Fragmentation

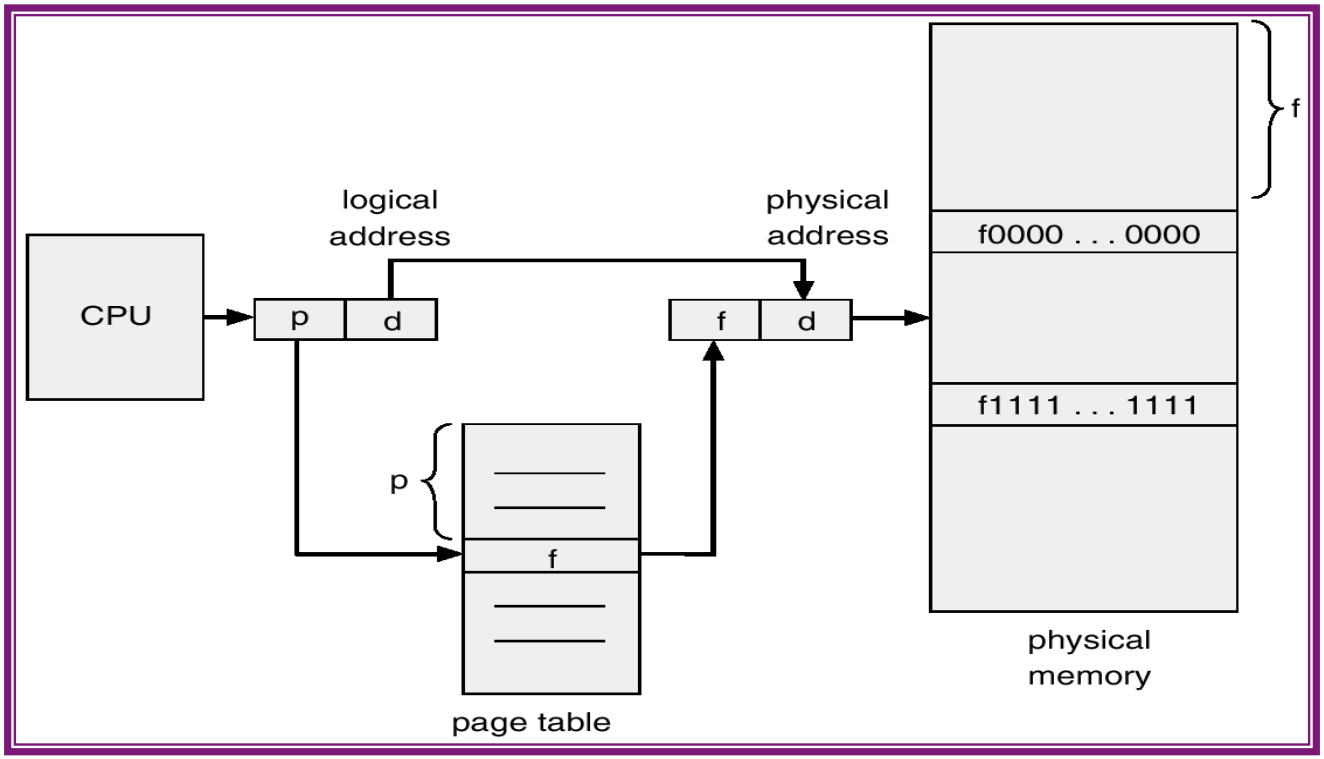
- The storage-allocation strategies suffer from external fragmentation.
  - External fragmentation – total memory space exists to satisfy a request, but it is not contiguous.
- The fragmentation can also be Internal fragmentation – the allocated memory may be slightly larger than requested memory.
  - 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 relocation is dynamic, and is done at execution time.
  - When compaction is possible, we must determine the cost → the size of memory contents that have to be moved; the less the better.
  - Selecting an optimal compaction strategy is quite difficult.

Paging

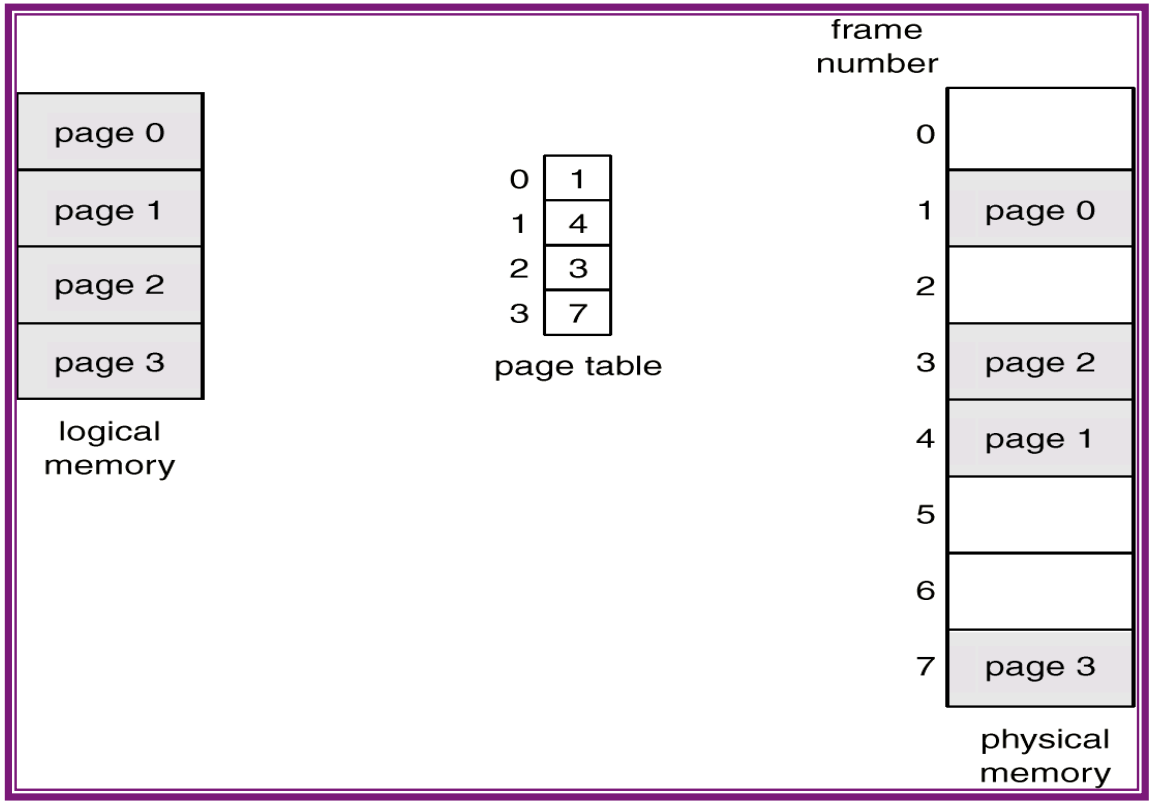
- Another solution to the external fragmentation is by using **paging**.
  - Paging system: NO *external* fragmentation,
    - but may create *internal* fragmentation.
  - Commonly used in many OS, including Unix, Linux, Windows.
- In the paging system:
  - The logical address space of a process can be noncontiguous.
    - A process is allocated physical memory when it is available.
  - The physical memory is divided into fixed-sized blocks called **frames**.
  - The logical memory is divided into fixed-sized blocks called **pages**.
    - Page size = Frame size
  - Frame size is a **power of 2**, e.g., 512 bytes and 1 GB; defined by hardware
    - This makes translation of logical address into **page #** and **page offset** simple
    - OS keeps track of all free frames.
    - The backing store is divided into fixed sized block the same size as clusters of frames
- To run a program of size *n* pages,
  - OS finds *n* free frames and load the program.
  - Set up a **page table** to translate logical to physical addresses.

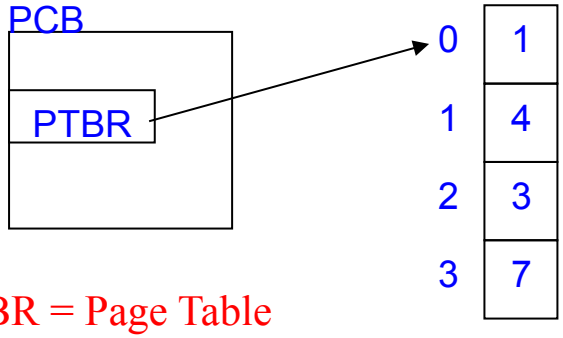
Address translation scheme

- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset (*d*)** – combined with base address to define the physical memory address that is sent to the memory unit
  - For 2^m logical address space and page size of 2^d, $p = m - d$



Paging example-1





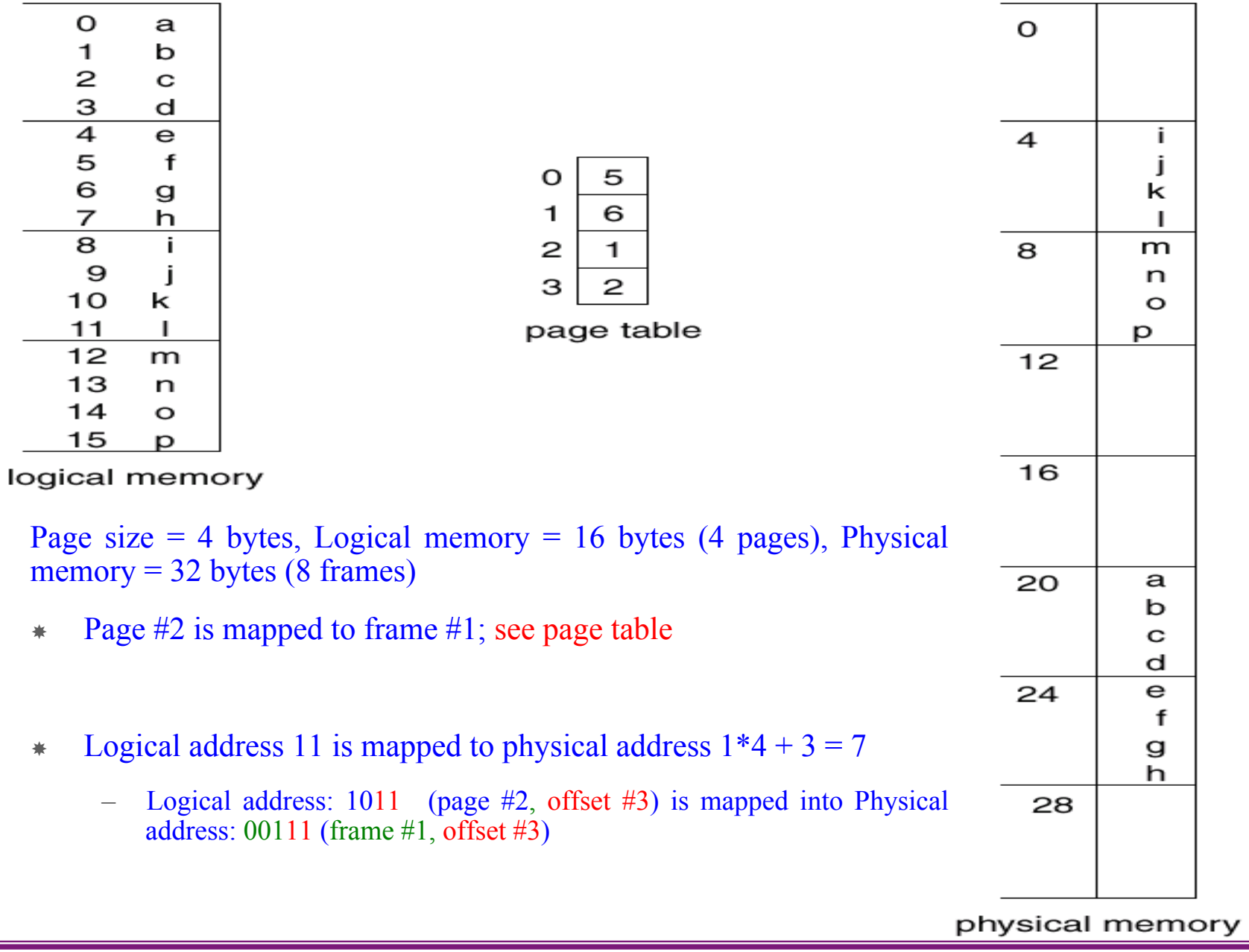
PTBR = Page Table Base Register

Page table

Logical address space = 16 = 2^4 bytes → m = 4 bits
Page size = 4 = 2^2 bytes → d = 2 bits
p = 4 - 2 = 2 bits → there are 2^2 = 4 pages

Similarly: Physical address space = 32 bytes → n = 5 bits
Frame size = Page size = 4 bytes → d = 2 bits
f = 5 - 2 = 3 bits → there are 2^3 = 8 frames

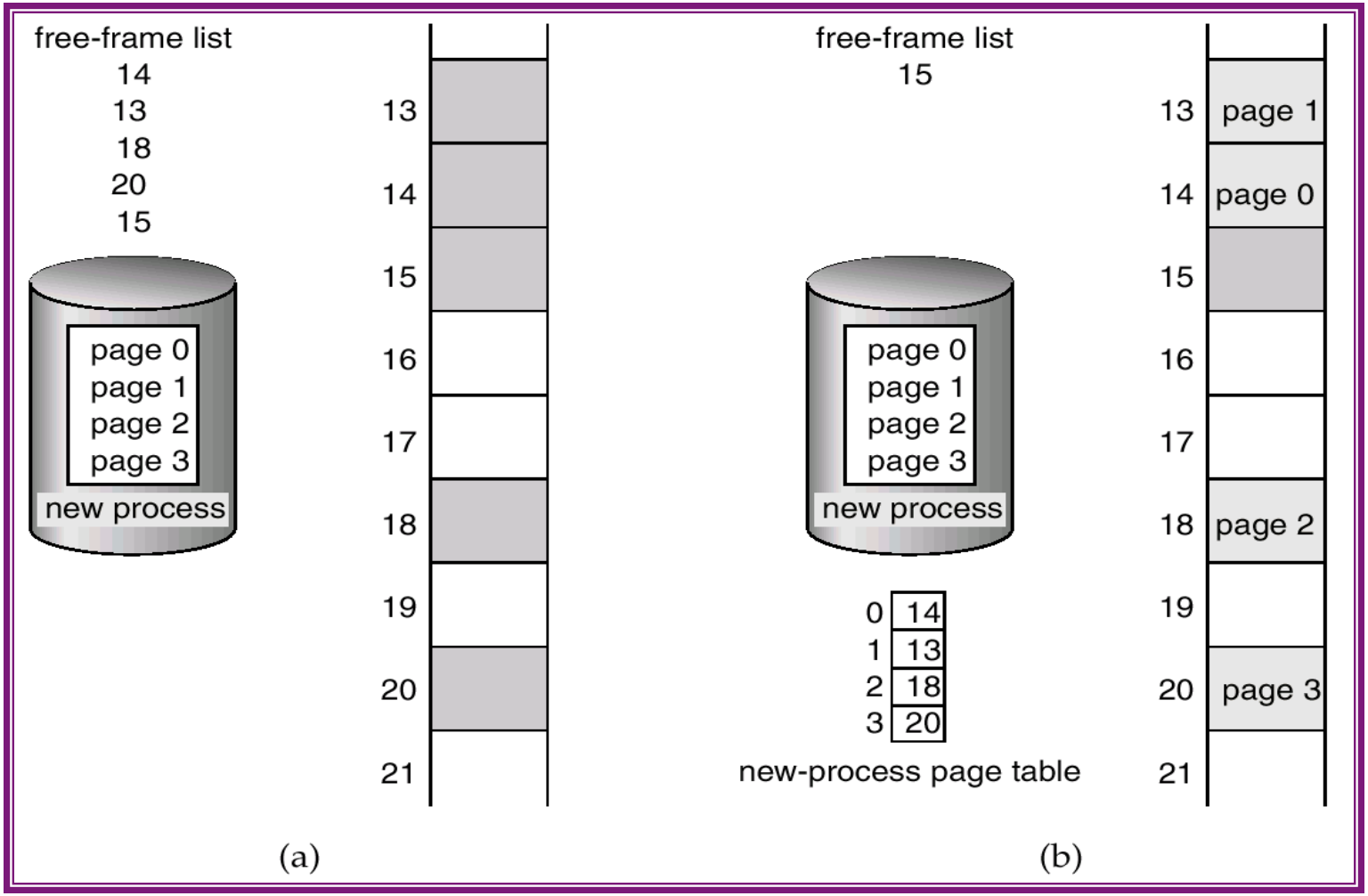
Paging example-2



Page size = 4 bytes, Logical memory = 16 bytes (4 pages), Physical memory = 32 bytes (8 frames)

- Page #2 is mapped to frame #1; see page table
- Logical address 11 is mapped to physical address 1*4 + 3 = 7
  - Logical address: 1011 (page #2, offset #3) is mapped into Physical address: 00111 (frame #1, offset #3)

Free Frames



Before allocation

After allocation

Implementation of page table

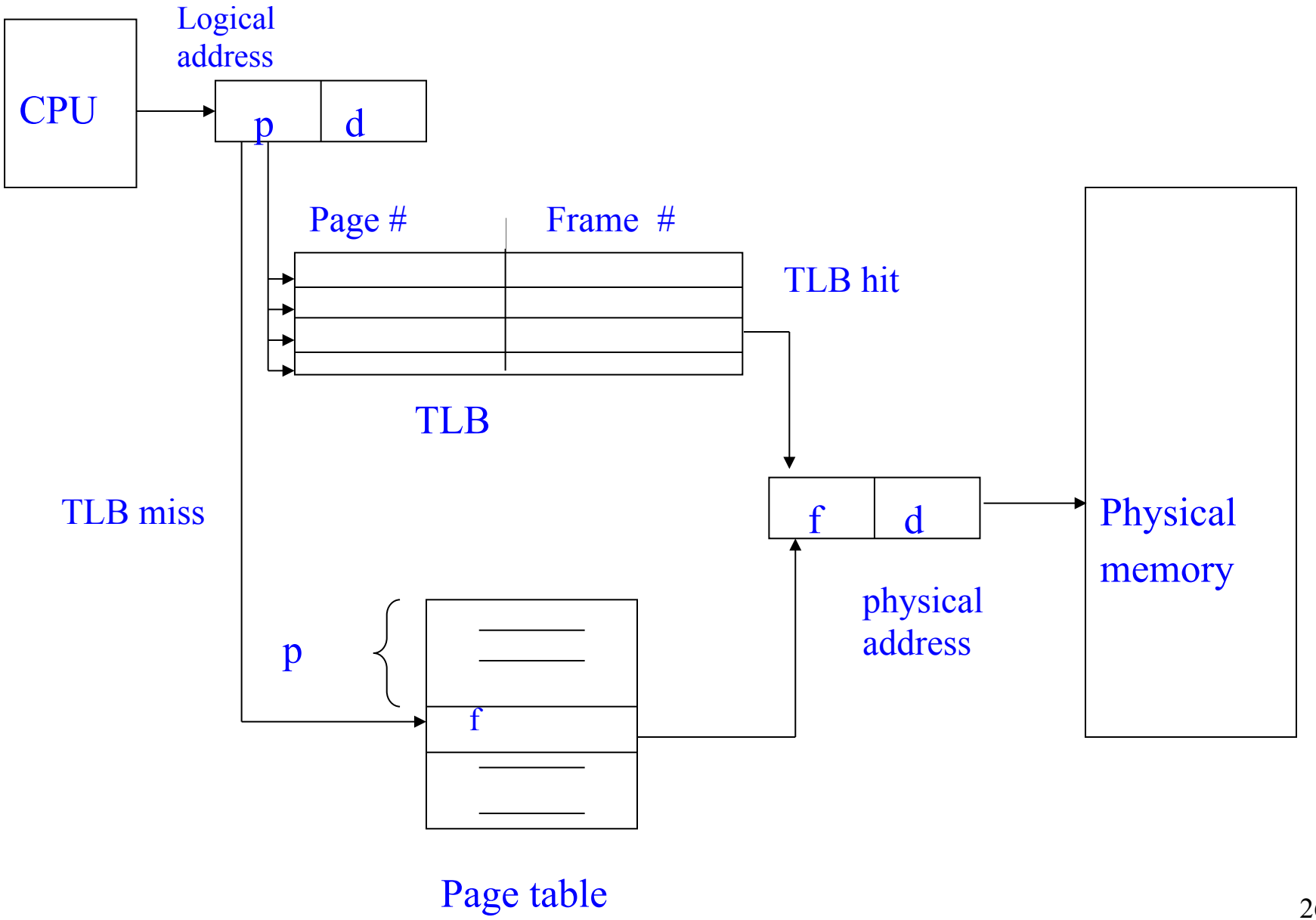
- Each page table is kept in main memory.
  - Page table base register (PTBR) points to the page table.
  - Page table length register (PTLR) shows size of the page table.
- In this scheme every data/instruction access requires **two** memory accesses:
  - **one** for page table access, and **one** for data/instruction access.
- The two memory access problem can be solved using a special fast-lookup hardware cache called ***associative registers***.
  - Also called ***translation look-aside buffers*** **(TLBs)**.
- TLB is expensive, so typical number of entries: 32 – 1024
  - Can have one TLB for data and another one for instruction
    - Can also have multi-level TLBs, e.g., Intel Core i7 has L1 and L2 TLBs as well as instruction and data TLBs.

Associative Register

- Address translation (A, A')
  - **parallel search;** very fast!
  - if A is in TLB, get frame #
  - Otherwise get frame# from page table in memory
    - Replace one entry in TLB with the new one → use algorithm, e.g., LRU
    - TLB entries for kernel code can be wired down → cannot be replaced
  - The step is done as part of instruction pipeline in CPU → adding no search time penalty
- Some TLBs include Address Space Identifiers (ASIDs) in each TLB entry
  - An ASID identifies a process
  - ASID for a running process must match the ASID in its TLB entry
    - For address space protection for that process
    - Allow TLB to contains entries for different processes at the same time.
  - Without using ASID, TLB must be **flushed** when there is a context switch

Page #	Frame #

Associative Register (cont.)



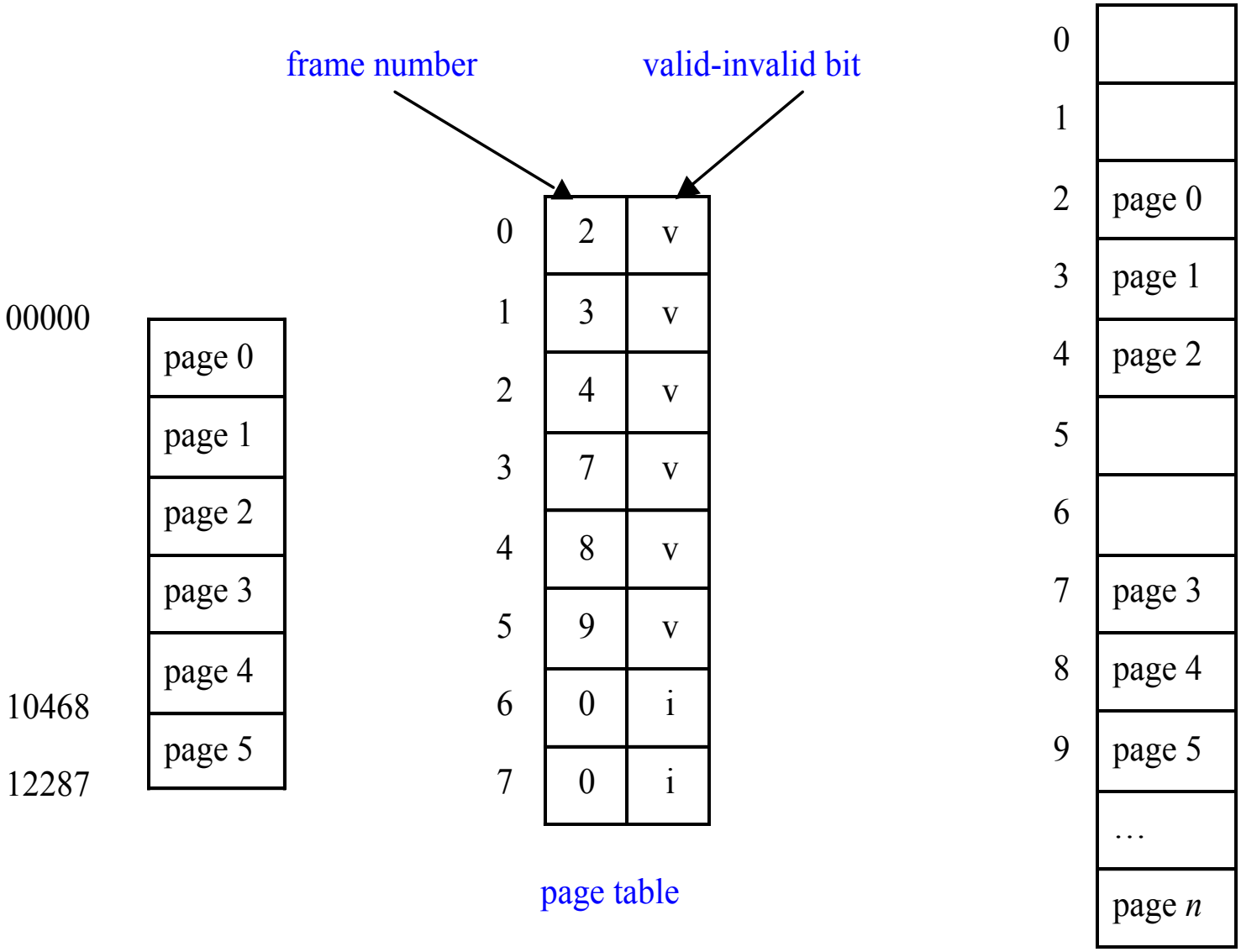
Effective access time

- **Notation:**
  - Associative lookup = ε time unit.
  - Assume memory cycle time is *t* time unit.
  - Hit ratio = α is the percentage of times that a page number is found in the associative registers.
- Effective access time (EAT):
  - If the page number is in TLB: time = ε + *t*.
  - If the page number is NOT in TLB: time = ε + *t* + *t*.
  - EAT = (*t* + ε) α + (2 *t* + ε)(1 - α) .
- **Example:** α = 0.8, ε = 20 ns, *t* = 100 ns;
  - EAT = 0.8 x 120 + 0.2 x 220 = 140 ns → 40% slowdown in memory access time.
  - For α = 0.98 → EAT= 122 ns.
- Hit ratio is related to the number of associative registers.

Memory protection

- Memory protection is implemented by associating a protection bit with each frame.
  - A **valid-invalid** bit is attached to each entry in the page table.
  - 'valid' shows the associated page is in the process' logical address space, and is thus a legal page.
  - 'invalid' shows the page is not in the process' logical address space.
- In addition, one bit can be used to set a page read-write or read-only access
  - Additional bits for other protection can be included
- Each attempt to access a page is checked against the protection bits
  - Any illegal attempt traps the OS

Memory protection (cont.)



Page Table Structure

- Hierarchical Paging.
- Hashed Page Tables.
- Inverted Page Tables.

Hierarchical Paging

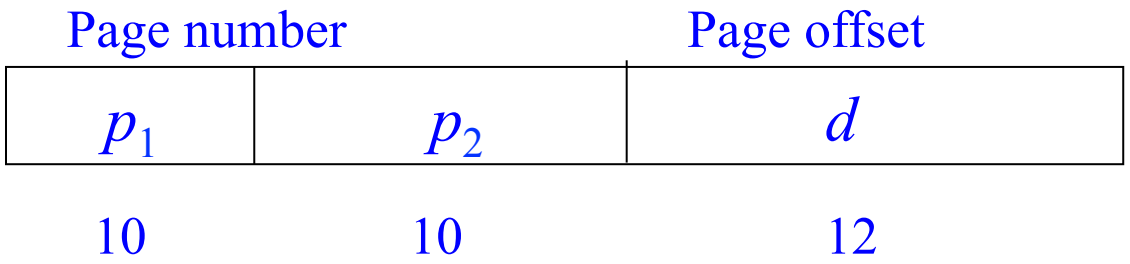
- Most modern computer systems support a very large logical address, e.g., 2^{32} to 2^{64} → page table becomes very large.

  Example: For a system with 32 bit logical address, and page size = 4KB (12 bits offset)

  - page table = 1 Million entries (32-12 = 20 bits) → 4 MB of physical address spaces for page table; 2^{20} * 4 bytes/entry
- One solution is to divide the page table into smaller pieces
  - For a two-level paging scheme, the page table itself is also paged.

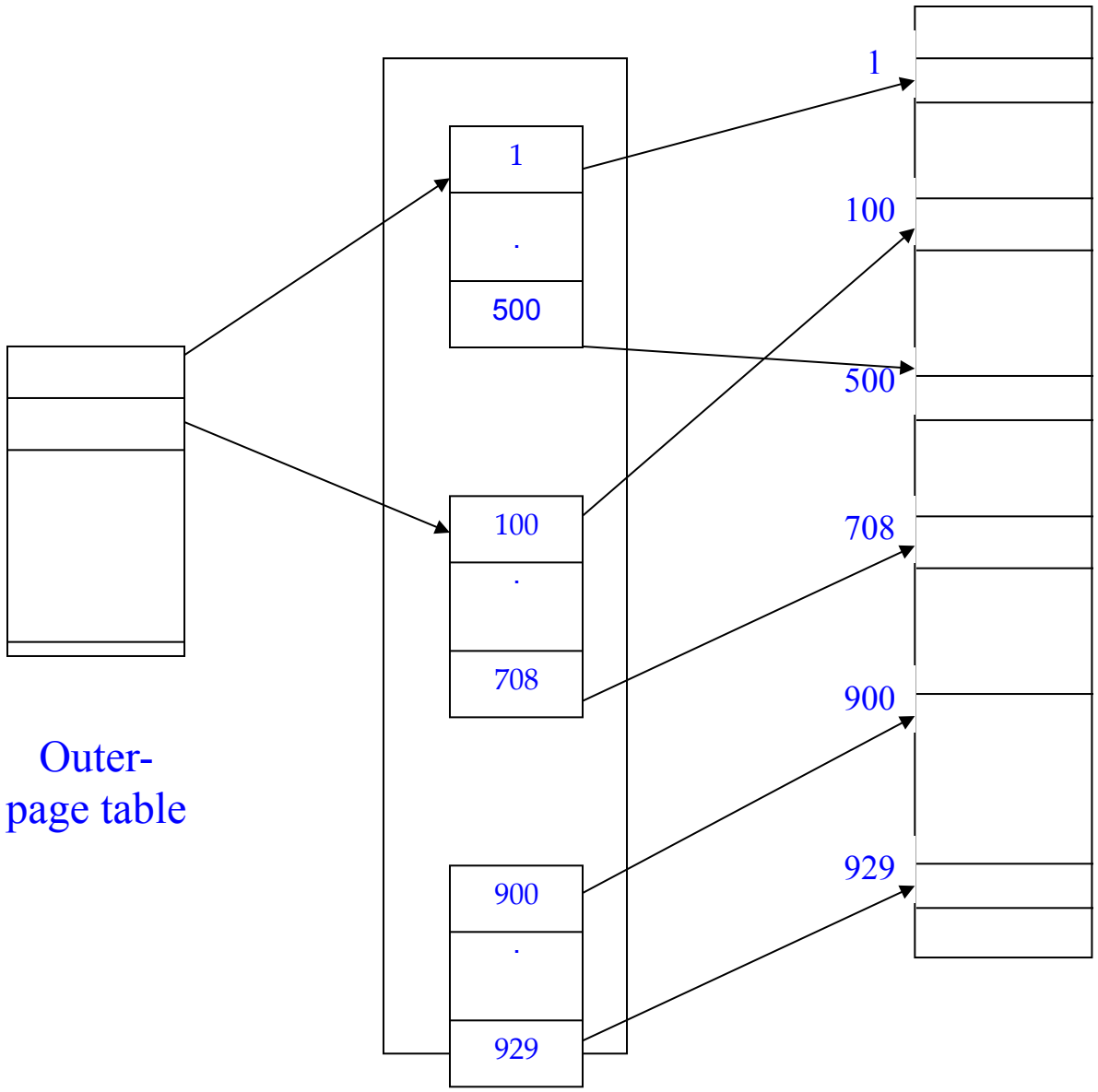
Two-level paging example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- Since the page table is paged, the page number is further divided into:
  - a 10 bit page number.
  - a 10 bit page offset.
- Thus, a logical address is as follows:



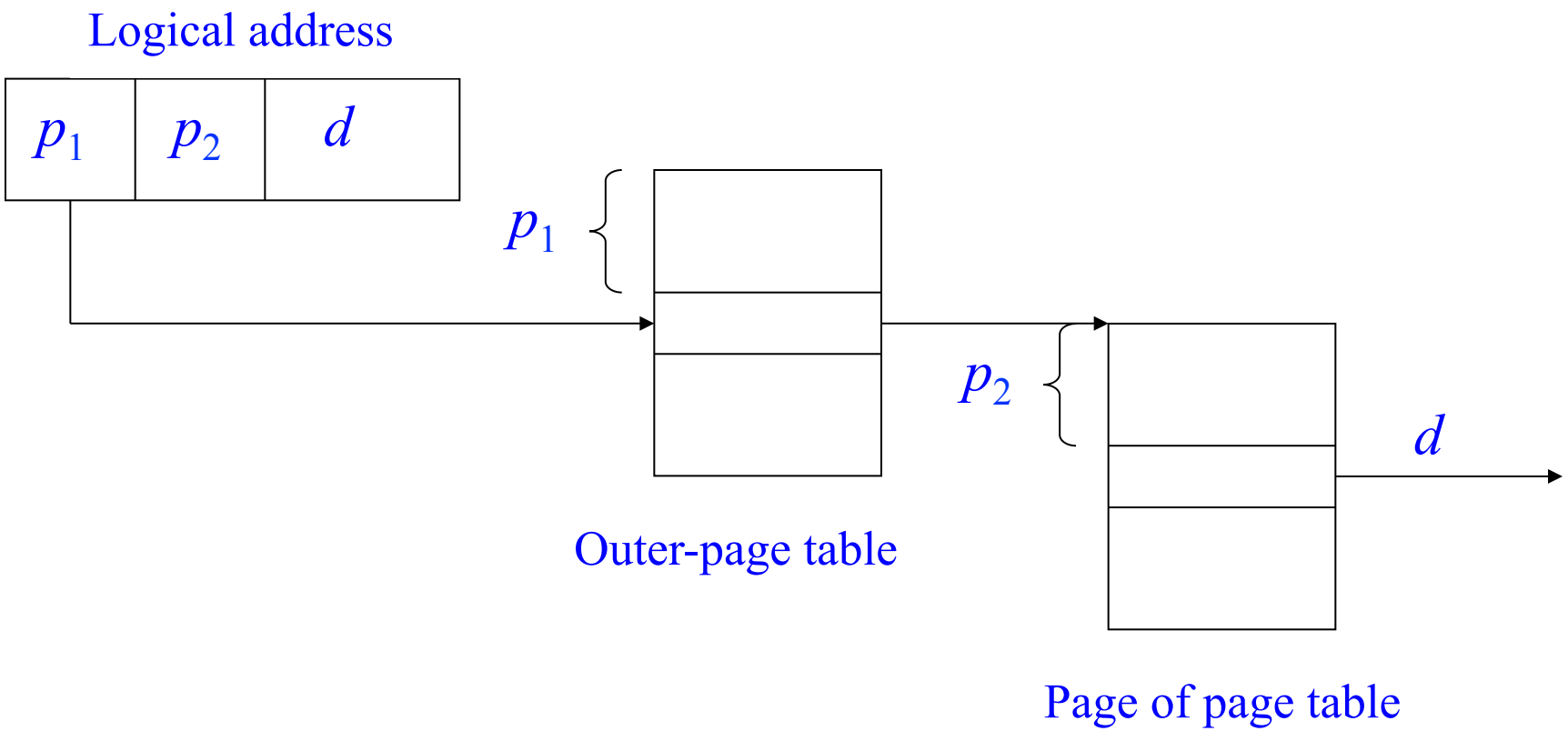
- p_1 is an index into the *outer page table*, and p_2 is the displacement within the page of the *inner page table*
  - Translation starts from the outer page table

Two-level paging example



Address-translation scheme

Two-level 32-bit paging architecture



Hierarchical paging and performance

- Since each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory accesses (for four-level paging system).
- Even though time needed for one memory access is quintupled, caching permits performance to remain reasonable.
- Cache hit rate of 98 percent yields:

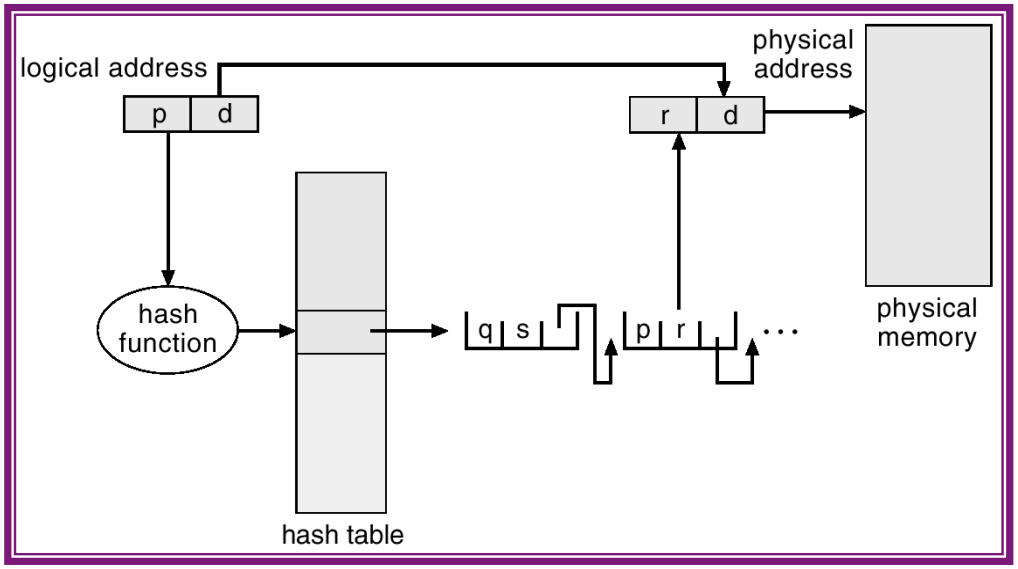
  EAT = 0.98 * 120 + 0.02 x 520 = 128 nanoseconds

  → only a 28 percent slowdown in memory access time.

  Assume: ε = 20 ns, t = 100 ns

Hashed Page Tables

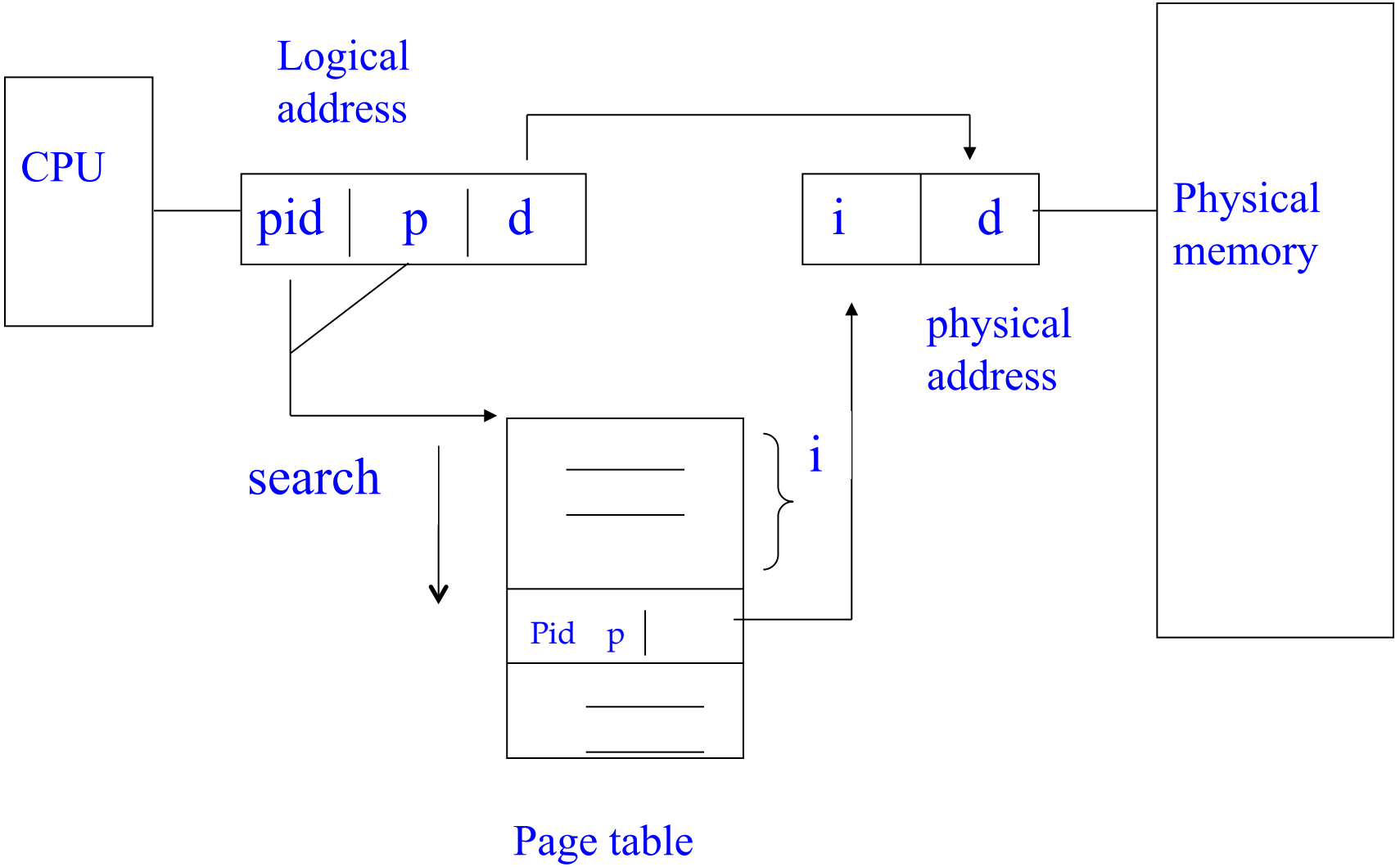
- Is commonly used when address spaces > 32 bits.
- The virtual page number is hashed into a page table.
  - This page table contains a chain of elements hashing to the same location.
  - Each element contains: virtual page number, the value of the mapped frame, pointer to the next element
- Virtual page number is compared in this chain searching for a match.
  - If a match is found, the corresponding physical frame is extracted.
- For address spaces > 64 bits, use Clustered page tables.
  - Similar to hash page table except each entry in page table refers to several pages (e.g., 16).



Inverted Page Table

- In the paging system, each process has a page table associated with it.
  - Drawback: each page table may consist of millions of entries → page tables consume large amount of physical memory.
- To solve this problem, use Inverted Page Table.
  - It uses ONLY one table that has one entry for each real page of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process (e.g., PID) that owns that page.
- This scheme *decreases* memory needed to store each page table, but *increases* time needed to search the table when a page reference occurs.
  - It may use hash table to limit the search to one – or at most a few page table entry.

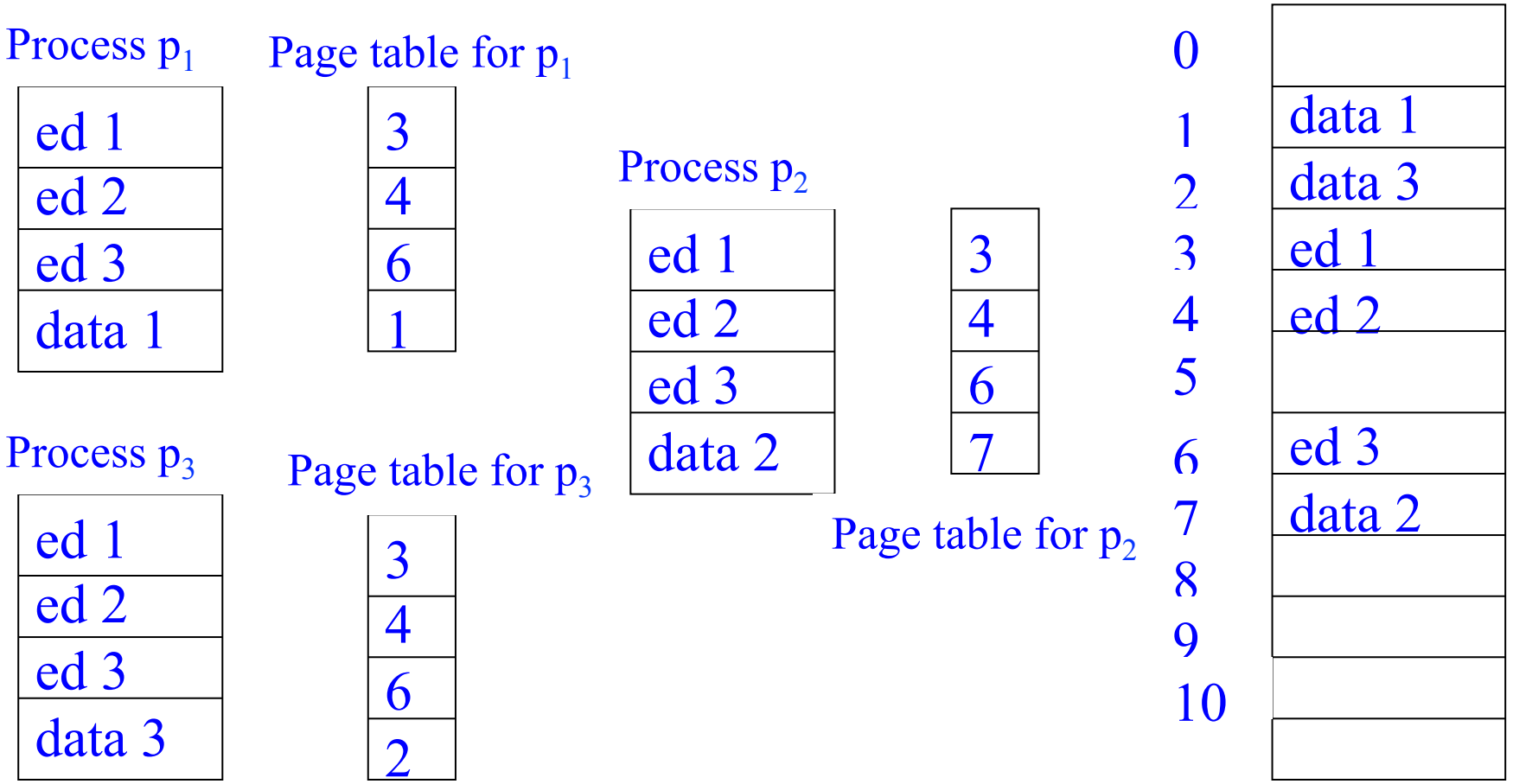
Inverted page table architecture



Shared pages

- Other advantage of paging is the possibility of *sharing* common code (reentrant code also called pure code).
- Reentrant code is non-self-modifying code → its code never change during execution.
- Shared code.
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- 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.

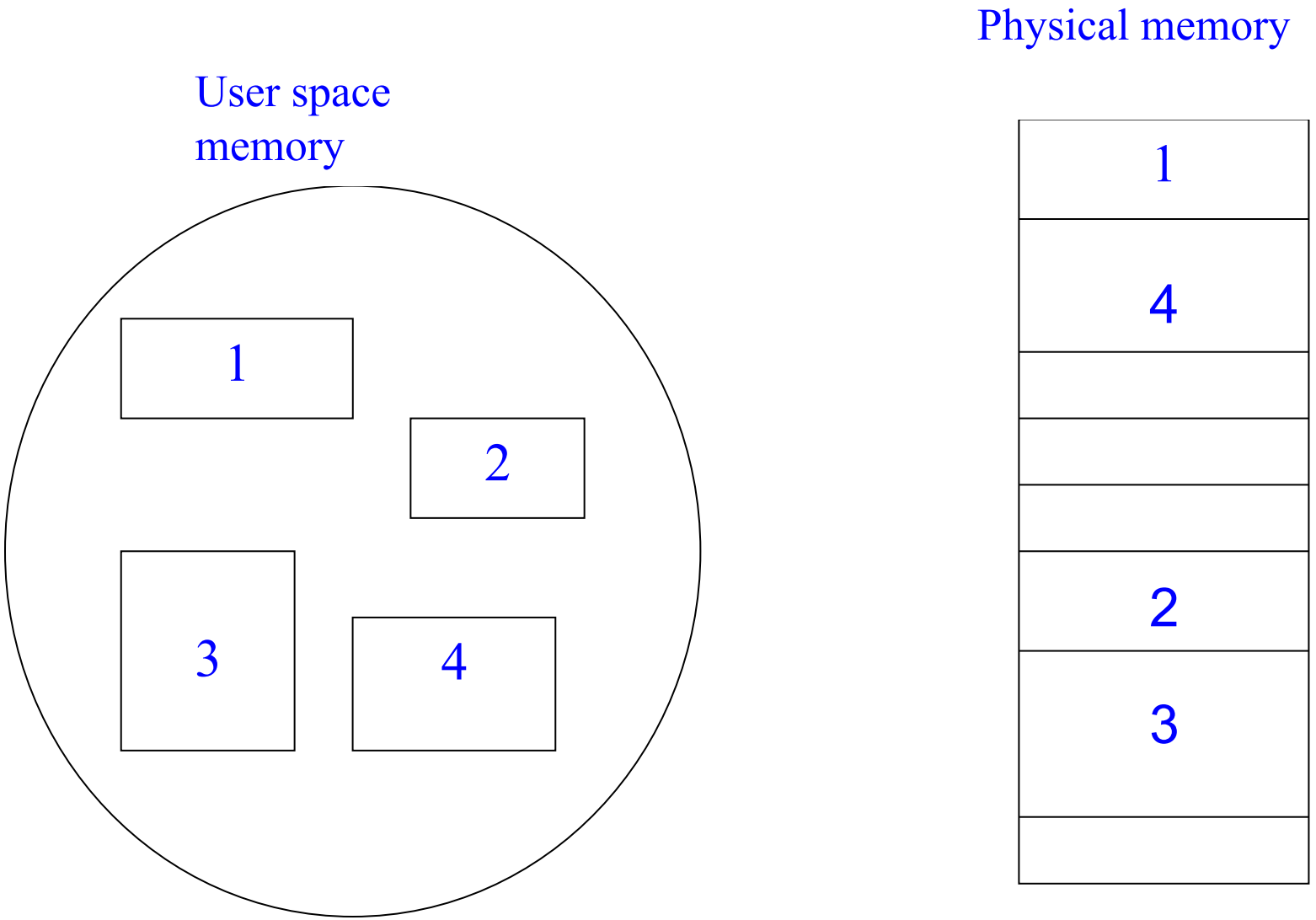
Shared pages example



Segmentation

- Memory management with paging makes separation between user view of memory and the actual physical memory.
- Segmentation is memory-management scheme that supports user view of memory.
- From user's view, a program is a collection of segments, and a segment is a logical unit such as:
  - Main program.
  - Procedure / Function.
  - Local variables, global variables.
  - Common block.
  - Stack.
  - Symbol table, arrays.
- Each segment has a name and length; the addresses specify both the segment name and an offset.
- Program specifies each address by two quantities: a segment name, and an offset (within the segment).

Logical view of segmentation



Segmentation architecture

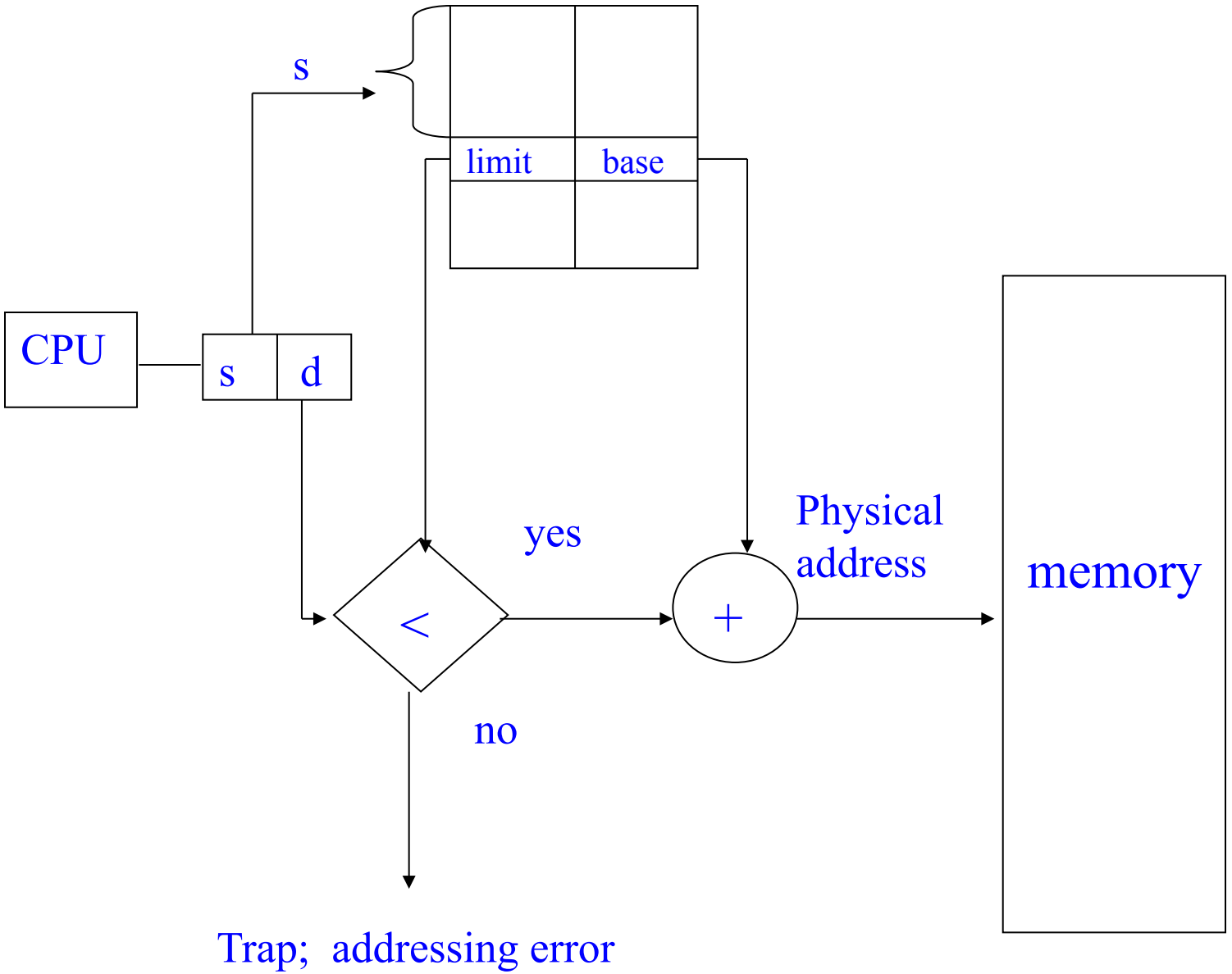
- Logical address consists of a two tuple: <segment-number, offset>.
- *Segment table* – maps two-dimensional user-defined addresses into one-dimensional physical addresses.
  - Each table entry has:
    - *Base* – contains the starting physical address where the segments reside in memory.
    - *Limit* – 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 the number of segments used by a program;
  - segment number s is legal if $s < \text{STLR}$.

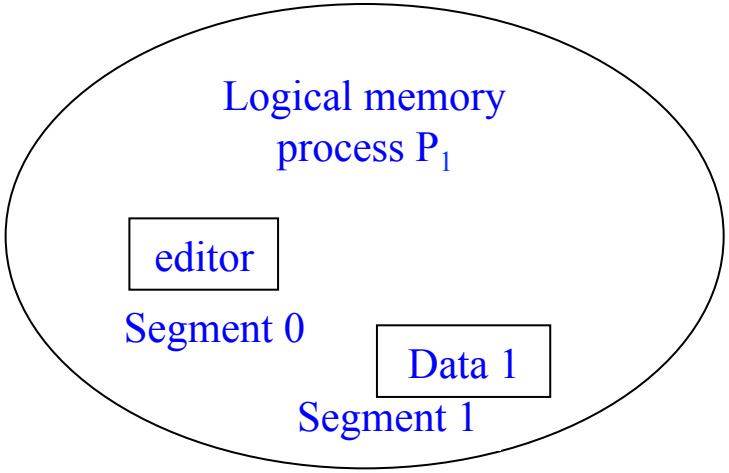
Segmentation architecture (cont.)

- Sharing.
  - Shared segments.
  - Same segment number.
- Allocation
  - First fit/best fit.
  - External fragmentation.
- Protection

  With each entry in segment table associate:
  - Validation bit = 0 → illegal segment.
  - Read/write/execute privileges.
- Protection bits associated with segments; code sharing occurs at segment level.
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem.

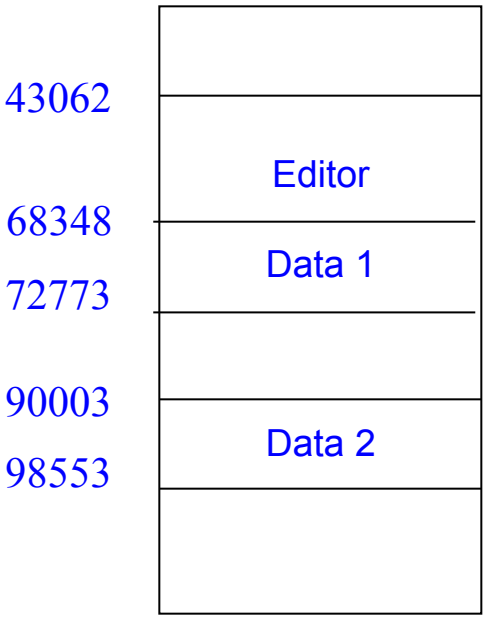
Segmentation Hardware



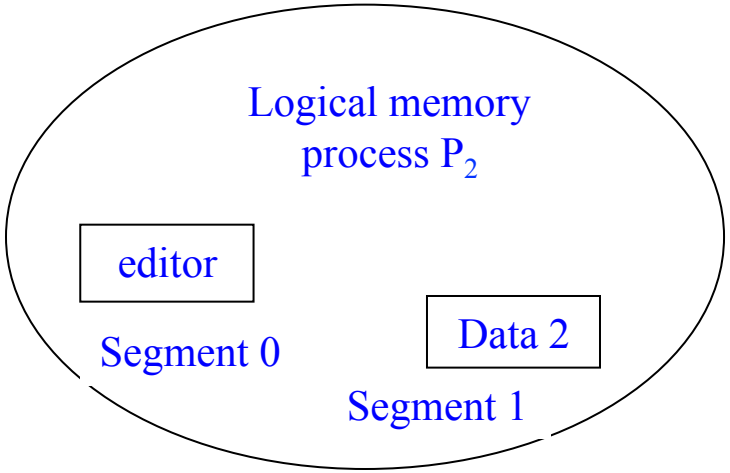


	limit	base
0	25286	43062
1	4425	68348

Segment table
process P_1



Physical memory



	limit	base
0	25286	43062
1	8850	90003

Segment table
process P_2

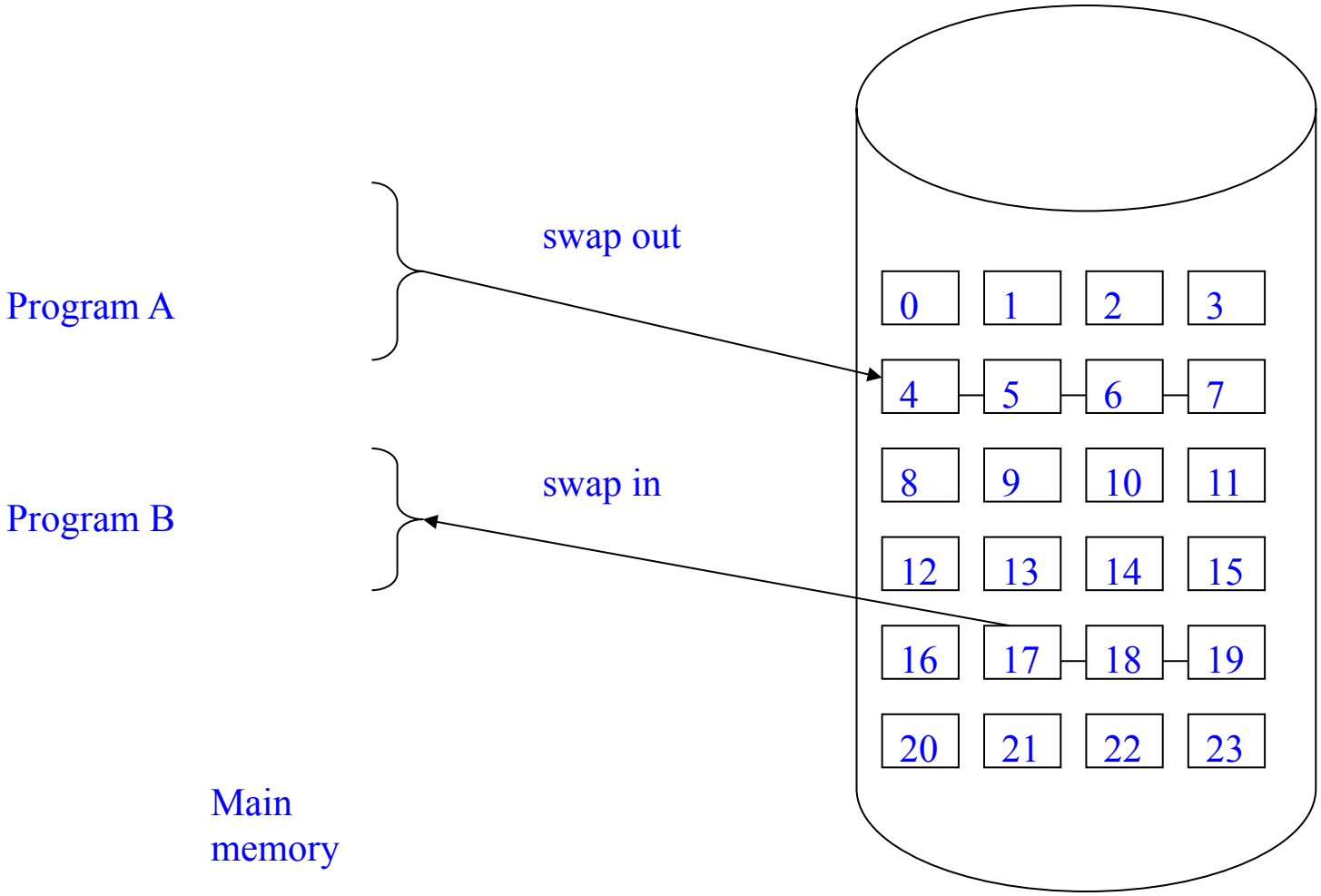
Virtual Memory – Chapter 9

- **Facts:**
  - Programs may have code that is almost never executed; e.g., error handler which is needed only when the program is in error
  - Not all parts of the program are needed at the same time
- Virtual memory – separation of user logical memory from physical memory
  - Non-virtual memory limits the size of program to the size of physical memory
  - Virtual memory allows the execution of program that is only partially in memory; advantages:
    - A program is not constrained by the amount of available physical memory.
    - With the available physical memory, more programs can be run at the same time.
    - Less I/O would be needed to load/swap user program into memory → program runs faster.
- Virtual memory can be implemented via:
  - Demand paging.
  - Demand segmentation.

Demand Paging

- Processes reside on secondary memory (usually disk).
- Bring a page (not entire process) into memory only when it is needed; advantages:
  - Less I/O needed.
  - Less memory needed.
  - Faster response.
  - More users.
- Needs hardware support to distinguish between those pages that are in memory and those pages that are on disk.

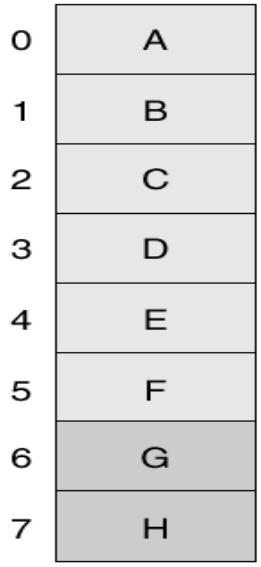
Demand Paging (cont.)



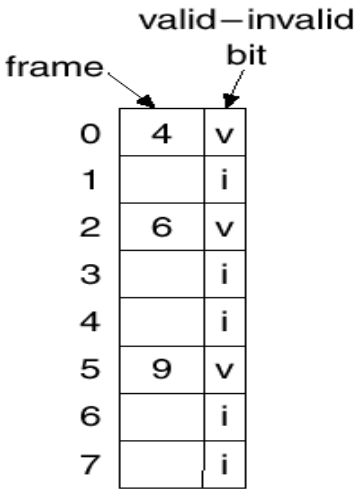
Demand Paging (cont.)

- Hardware requirements:
  - Swap space or backing store (usually high-speed disk).
    - It holds those pages that are not present in the main memory.
  - Page tables to mark an entry valid/invalid:
    - Valid bit = 1 → legal page, and in memory.
    - Valid bit = 0 → page on disk (valid) **or** not valid.
- Needs page replacement algorithm and frame-allocation algorithm.
- When a process needs a page:
  - If the page is in memory, no need to do anything → continue as before.
  - If the page is not in memory (invalid bit is set)→ generate **page-fault** trap to OS,
    - OS brings in the page to memory.

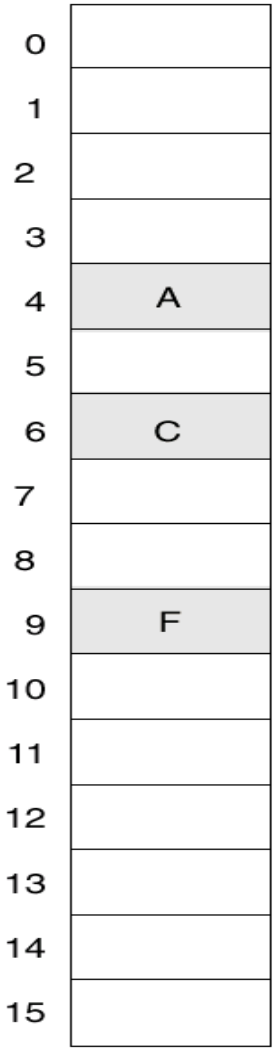
Page Table When Some Pages Are Not in Main Memory



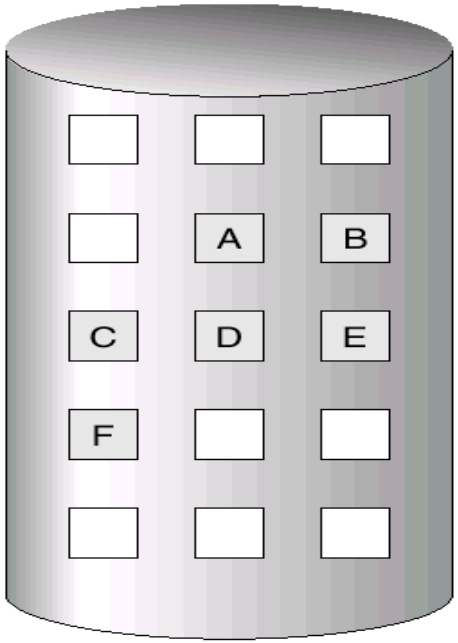
logical memory



page table



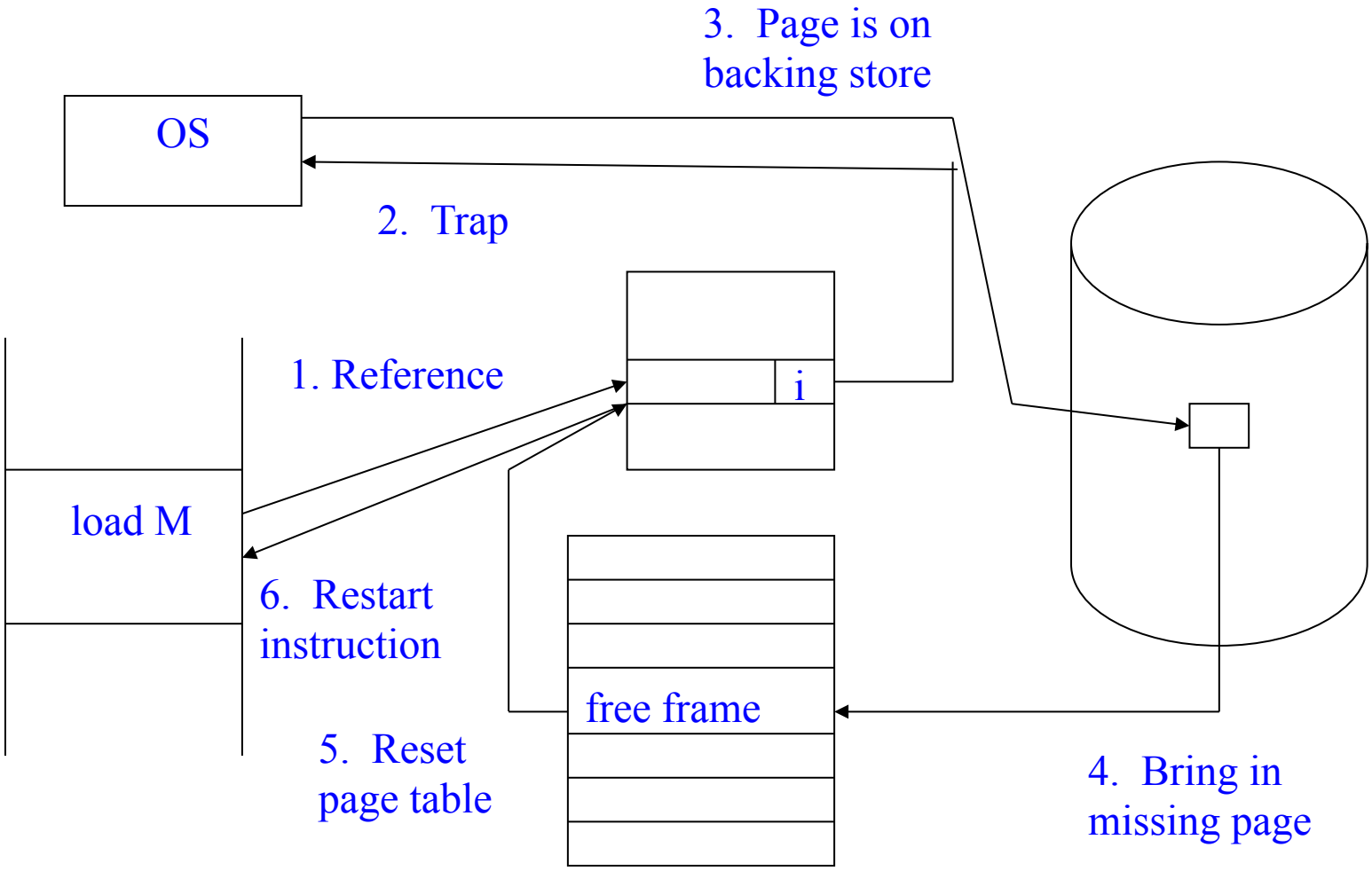
physical memory



Demand Paging (cont.)

- Procedure for handling page fault trap:
  1. Check if reference address is valid (from internal table in PCB).
  2. If invalid, terminate with an error.
  3. Else, find a free frame in memory (use the free frame list).
  4. Schedule a disk read operation to read the page into allocated frame.
  5. After reading, modify the page table.
  6. Restart instruction that was interrupted by the trap.
- Extreme cases: pure demand paging - never bring in a page into memory until it is required.
  - Start execution with **no** pages in memory.
  - Page fault occurs until every page that is needed is in memory.

Steps in handling a page fault



Demand Paging (cont.)

- Need to restart any instruction after a page fault since a page fault can occur at any memory reference:
  - On instruction fetch: restart by fetching the instruction again.
  - On operand fetch: restart by re-fetch instruction, re-decode, and fetch operand.
- Other problems involving:
  - Instruction that modify several different locations such as MVC instruction (IBM 360/370) that can move up to 256 bytes from one location to another (possibly overlapping) location → two different solutions.
  - Machines with special addressing modes, such as, auto increment/ decrement → solution use status register that stores the information about which register has changed and its old content so that the system can undo the changes.

What happen if there is no free frame?

- Do page replacement – find some page in memory, but not really in use, swap it out.
  - Need page replacement algorithm.
  - Performance – want an algorithm with minimum number of page faults.
- During program execution, same page may be brought into memory several times.

Performance of demand paging

- Demand paging can have significant effect on the performance of computer system
- Memory access time (*ma*) = 10 to 200 ns.
- Probability of page fault (page fault rate): $0 \le p \le 1$.
  - If $p = 0$ → no page faults.
  - If $p = 1$ → every reference is a fault.
- Effective Access Time (EAT):
  EAT = $(1 - p)$ * memory access + p * (page fault time).

Page fault time include the time for:

- Trap to operating system.
- Save user registers and process status.
- Determine that interrupt is a page fault.
- Check page reference was legal and determine location of page on disk.
- Issue a read from disk.
  - Wait in queue for this read to be serviced.
  - Wait for device seek and latency time.
  - Begin to transfer page to a free frame.
- Allocate CPU to another process while waiting.
- I/O from disk completed (interrupt occurs).
- Save registers and process state for the other process.
- Determine that I/O interrupt was from disk.
- Correct page table to reflect the new state.
- Wait for the CPU to be allocated to that process again.
- Restore page table, process state and user registers for this process and resume instruction.

Three major components of page fault service time:

- Service page fault interrupt (1-100μs).
- Read in page (8ms): average latency = 3ms, seek time = 5ms, transfer time = 0.05ms
- Restart process (1-100μs).

Demand paging example

- Memory access time = 200 ns.
- Page fault Time = 8ms.
- EAT = (1 – p) * (200) + p (8,000,000) ns
  = (200 + 7,999,800 * p) ns.
- EAT is directly proportional to the page fault rate → if we want to have less than 10% performance degradation, we need p < 0.0000025.
- It is important to keep the page fault rate low in a demand paging system, otherwise the EAT increases → slowing process execution dramatically.
- To decrease inefficiency:
  - Use special swap disk/partition (bigger blocks, no file structure) → faster access.
  - Copy whole process to swap disk before execution → demand page from swap space.
  - For re-entrant code, do not swap out; just overwrite.
  - Demand pages from file system when first required. Write to swap when need to replace (used in BSD UNIX).

Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files

Copy-on-Write

- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory.
  - If either process modifies a shared page, only then the page is copied.
  - COW allows more efficient process creation as only modified pages are copied.
  - Used in Linux
- Free frames are allocated from a *pool* of zeroed-out frames.

Note: virtual memory fork - vfork() is different from fork with COW

- vfork does not use COW
  - The child process runs and parent process is suspended
  - Child process can change the parent's address space, which will be visible from the parent when it resumes execution
  - vfork is good when the child process calls exec() immediately after it is created
    - Used to implement Linux command line shell

COW (cont.)

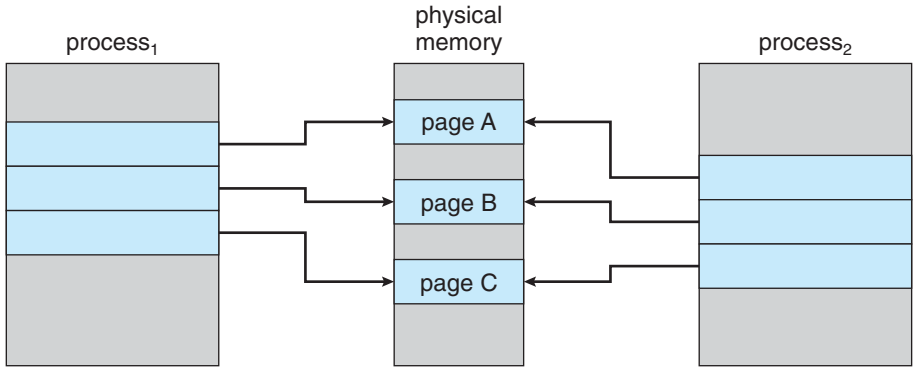


Fig. 9.7 (textbook)

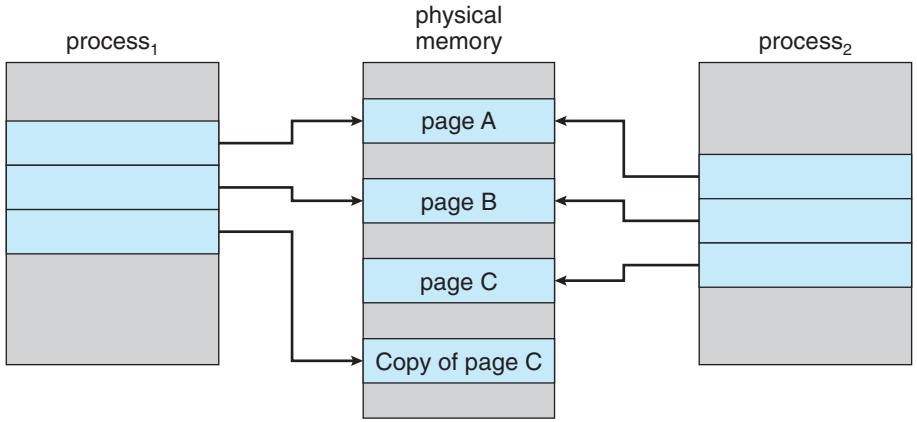
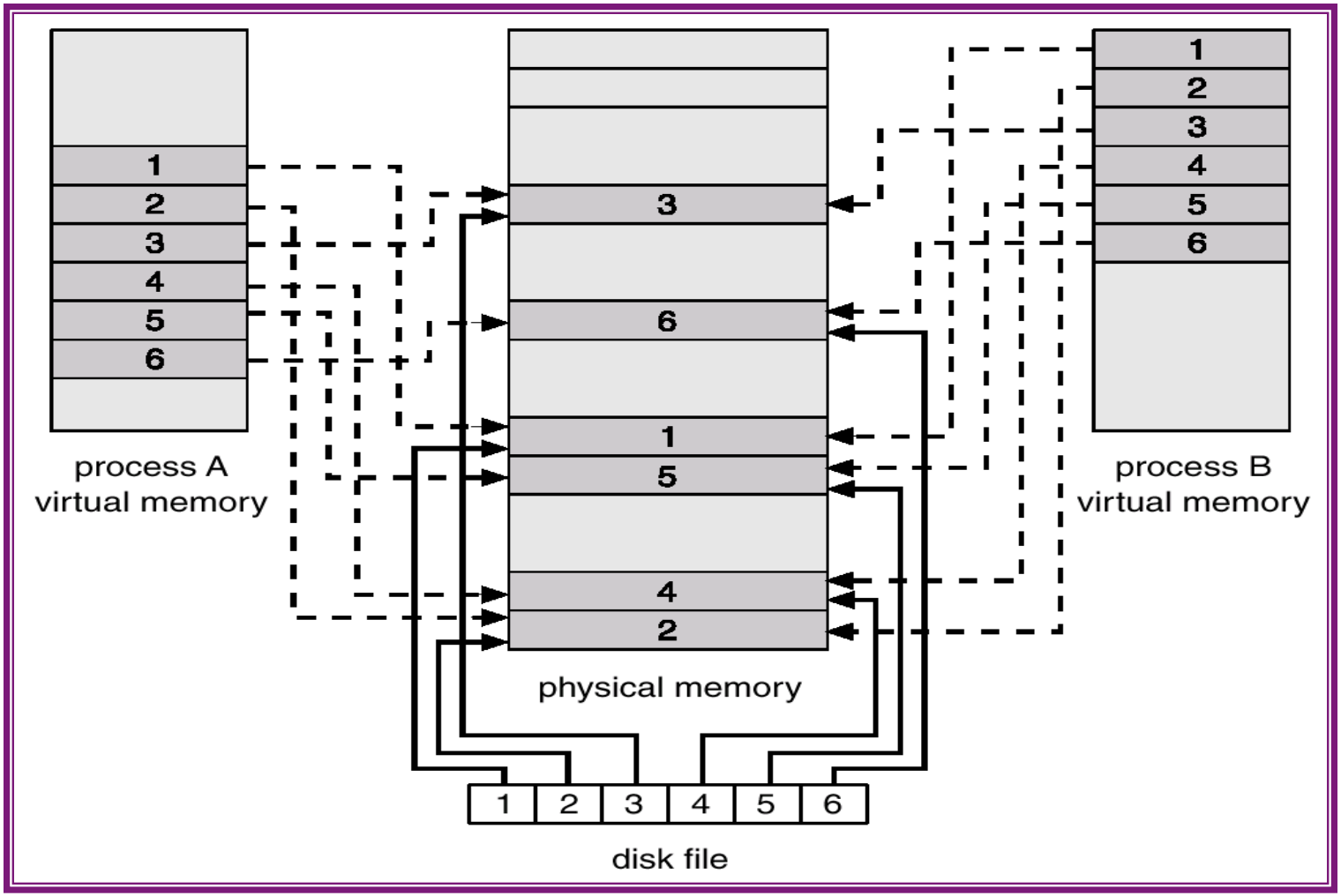


Fig. 9.8 (textbook)

Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory.
- A file is initially read using demand paging.
  - A page-sized portion of the file is read from the file system into a physical page.
  - Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **read()** **write()** system calls.
- Also allows several processes to map the same file allowing the pages in memory to be shared.
- Writing to the memory mapped file does not require writing to the physical file on disk immediately
  - Some OS periodically updates the physical file

Memory-Mapped Files



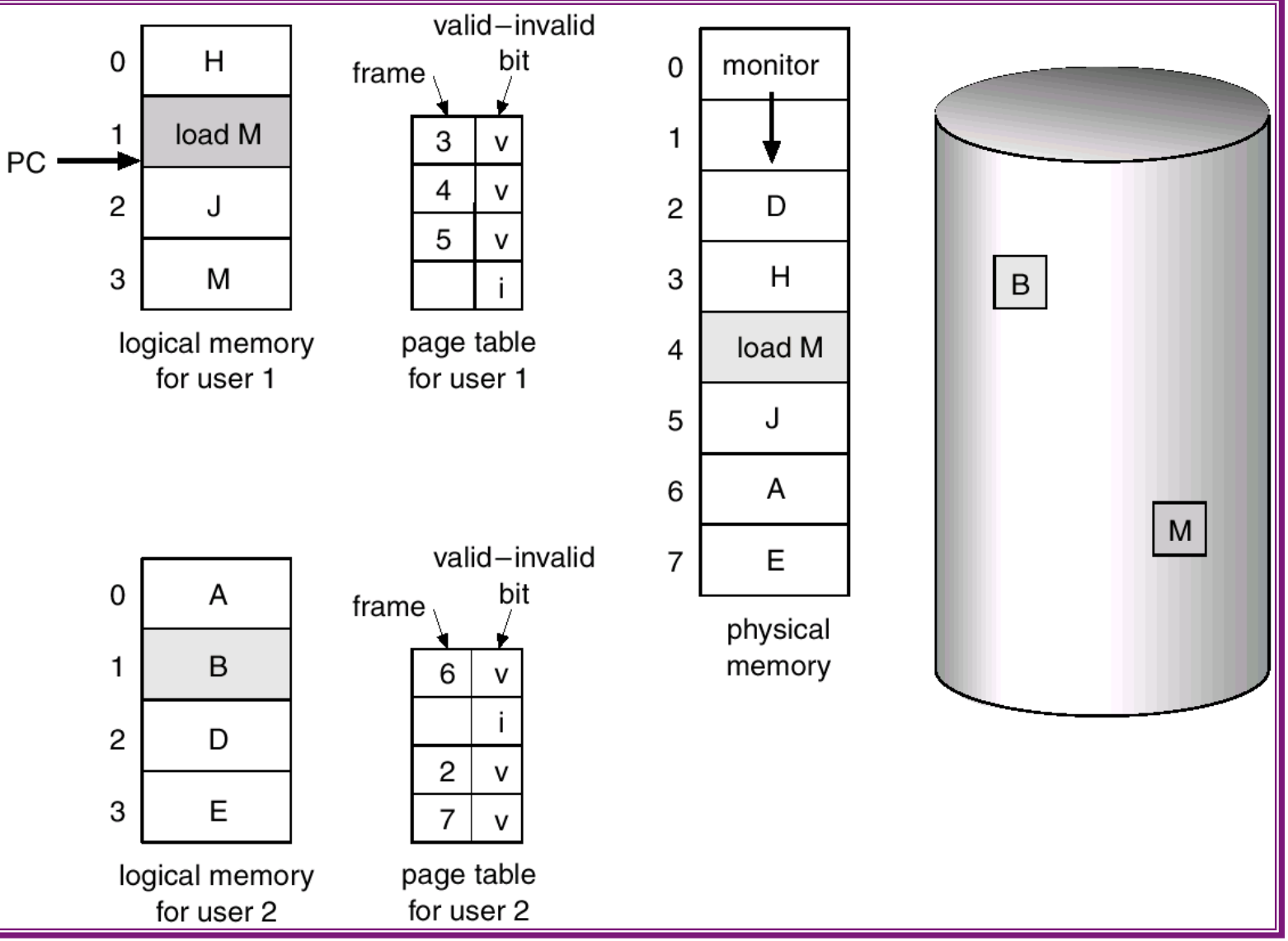
Page replacement

- Processes may need more pages than available frames in physical memory.
  - If we increase the degree of multiprogramming → *over-allocating* memory.
- What should be done if a process generates a page fault, but there is no free frame for requested page?
  - Terminate the process? NOT good!
  - Use a page replacement algorithm!
- Page replacement completes separation between logical memory and physical memory
  - Large virtual memory can be provided on a smaller physical memory
- Demand paging also needs a frame-allocation algorithm
  - Decide how many frames to be allocated to each process

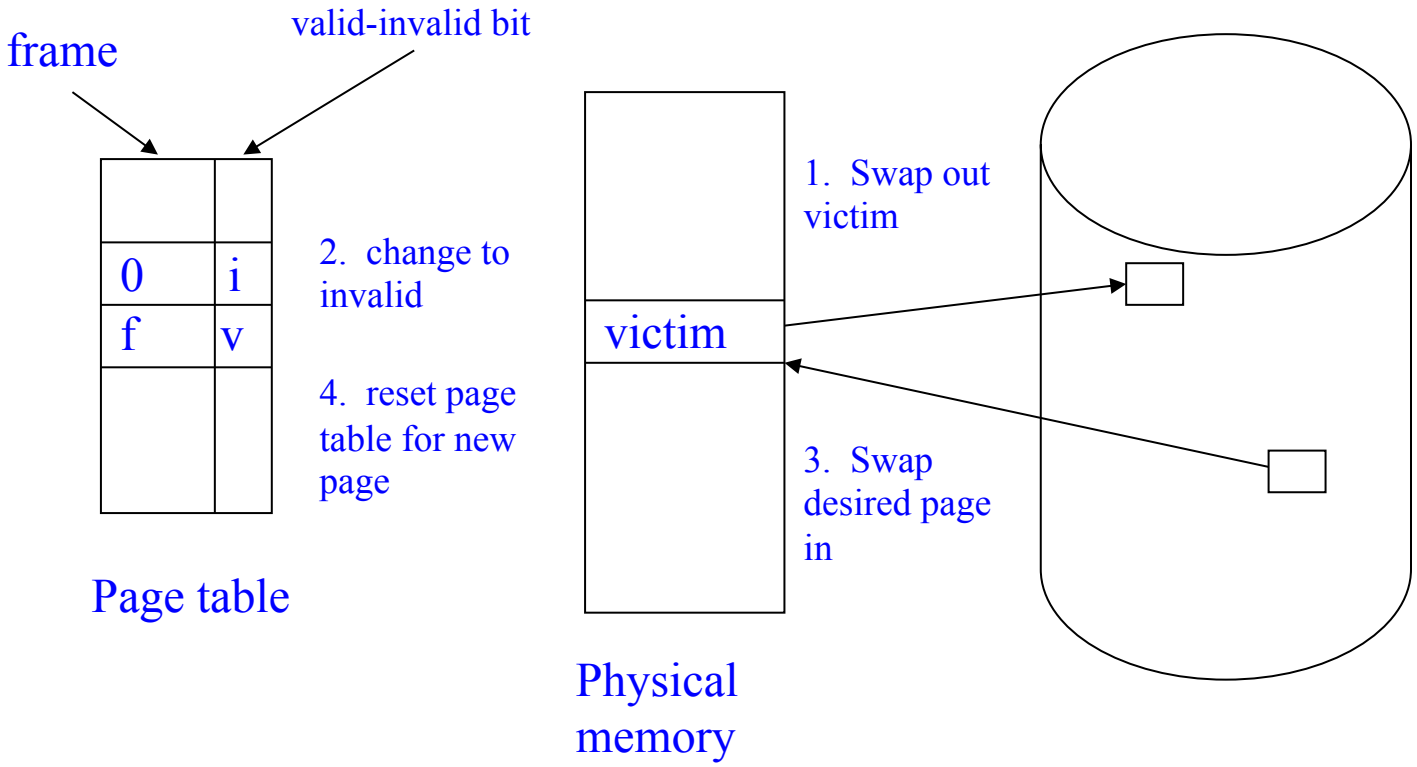
Page replacement (cont.)

- Over-allocation of memory is prevented by modifying page-fault service routine to include a page replacement function:
  - Find a location of the desired page on the disk.
  - Find a free frame.
    - If there is a free frame, use it.
    - Otherwise, use a page-replacement algorithm to select a *victim* frame.
    - Write the victim page on disk
      - Update the page and frame tables accordingly
    - Read the desired page into the (new) free frame
      - Update the page and frame tables
    - Restart the user process.
- If no free frame, two page transfers are required!
  - Use *modify* (*dirty*) bit to reduce overhead of page transfer
    - Each page is associated with a dirty bit
      - When a page is modified, set its dirty bit
      - Only modified pages are written back to disk

Need For Page Replacement



Page Replacement



Page-replacement algorithms

- There are many page replacement algorithms
  - We want an algorithm that results in the lowest *page-fault rate.*
- Evaluate algorithm by running it on a particular string of memory references (reference string) and compute the number of page faults on that string.
- Reference string can be generated
  - by random number generator (artificial) or
  - by tracing a given system and recording the address of each memory reference.

Example:

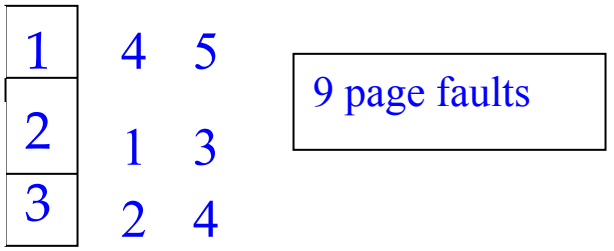
- Consider the following memory address sequence:

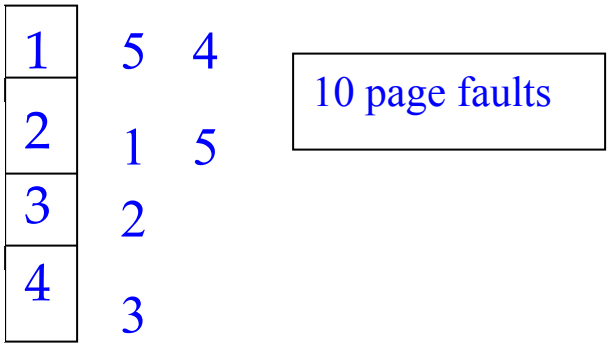
  0100, 0432, 0101, 0612, **0102, 0103, 0104, 0101**, 0611, **0102, 0103, 0104, 0101**, 0610, **0102, 0103, 0104, 0101**, 0609, **0102, 0105**.

- If system's page size = 100 bytes, we have the following page reference string: 1, 4, 1, 6, **1**, 6, **1**, 6, **1**, 6, **1**
- To determine the number of page faults for a particular reference string and page-replacement algorithm, we need to know the total number of frames available.

First-in-first-out (FIFO) algorithm

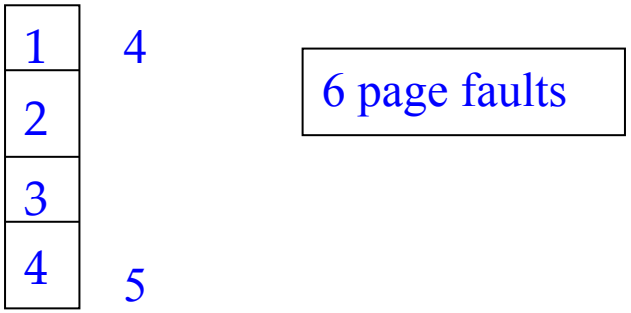
- The simplest page-replacement algorithm.
  - Use FIFO queue: replace the page at the head of the queue, and insert a new page at the tail.
- FIFO replacement suffers from Belady's anomaly
  - More frames result in more page faults – an anomaly!
- **Examples:** reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 or 4 frames (3 or 4 pages can be in memory at a time per process)





Optimal page replacement algorithm (OPT)

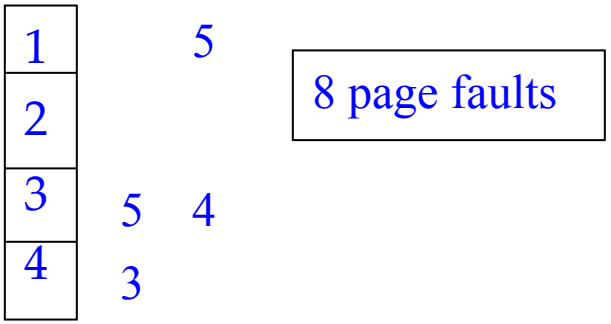
- OPT replaces page that will not be used for the longest period of time
- For a fixed number of frames, OPT has the lowest page fault rate of all algorithms
  - It also never suffers from Belady's anomaly.
- 4 frames example: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- **Problem:** It is difficult to implement
  - It requires future knowledge of the reference string!
- This algorithm is used mainly to measure how well a new algorithm performs

Least recently used (LRU) algorithm

- LRU replaces the page that has not been used for the longest period of time.
  - Associate each page with its last use
  - It has good performance, and is often used
  - It does not suffer from Belady's anomaly
- A class of algorithms that do no suffer from Belady's anomaly are called *stack algorithms*.
  - In a stack algorithm, the set of pages in memory for *n* frames is always a subset of the set of pages that would be in memory with *n*+1 frames.
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

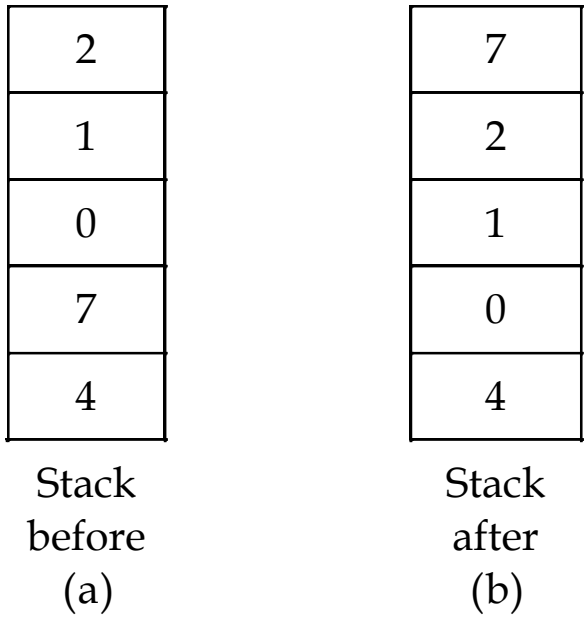


LRU Implementation

- Require hardware assistance
  - Otherwise slow down the system (because interrupt for every memory reference and software data structure update is inefficient and slow).
- Implementation using counter.
  - Every page entry has a time-of-use field, and CPU has a logical clock or counter (incremented for every memory reference).
    - Every time a page is referenced, copy the clock into its time field.
    - Replace a page with the smallest time value.
- Implementation using stack.
  - Keep a stack of page numbers in a double link list with head and tail pointers
  - Page referenced: move it to the top; requires 6 pointers to be changed because entries may be removed from the middle of the list.
  - No search for replacement, but update is expensive.
  - Updating must be done every memory reference.

Example

- Reference string: 4 7 0 7 1 0 1 2 1 2 (a) 7 (b) 1 2



LRU approximation algorithms

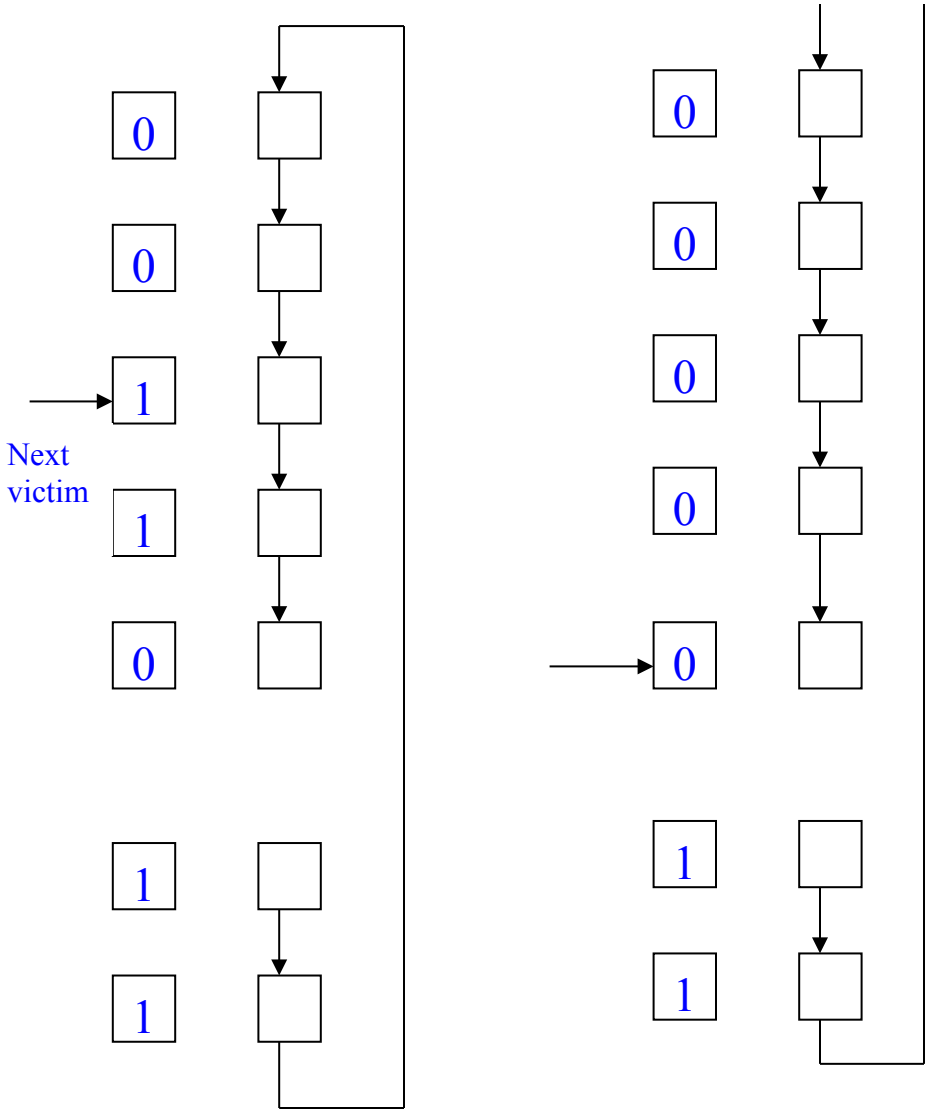
- If the hardware provides assistance with *reference bit*, use LRU approximation.
- Reference bit algorithm.
  - With each page associate a bit, initially = 0 (by OS)
- When a page is referenced, its bit is set to 1 (by hardware)
- Replace a page with bit = 0 (if one exists).
  - However, we do not know the order → lead to many page replacement algorithms that approximate LRU.

Additional-reference-bits algorithm

- Keep 8-bit byte record for each page.
- Record the reference bit at regular interval in a table in memory (say every 100 ms, by OS).
- OS shifts the reference bit of each page into MSB of the 8-bit byte shift register, shifting the other bits, discarding LSB.
- Replace page which has smallest 8-bit byte value.

Second chance algorithm

- Basic algorithm is FIFO.
- Need the reference bit.
- If a page to be replaced (clockwise) has reference bit = 1:
  - Set reference bit to 0.
  - Leave page in memory.
  - Replace next page (in clock order), subject to same rule.



Enhanced second-chance algorithm

- Include a *modify* bit to reduce the number of I/O required.
- Use the *modify* and *reference* bits as a pair (four page classes).
  - (0,0) neither used or modified.
  - (0,1) not used recently but modified.
  - (1,0) used recently but not modified.
  - (1,1) used recently and modified.
- Replace (0,0) first.
- Use FIFO if all pages have the same pair of values.

Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- Least Frequently Used (LFU) Algorithm: replaces page with smallest count.
- Most Frequently Used (MFU) Algorithm: the page with the smallest count was probably just brought in and has yet to be used.
- Implementation of these algorithms is expensive.

Page Buffering Algorithm

- An algorithm that is used together with any page replacement algorithm.
- OS keeps a pool of free frames.
- When a page fault occurs,
  - Select victim frame as before
  - Write the desired page into a free frame from the pool
    - before the victim is written out (so that process can restart sooner).
  - Write out the victim frame to the disk
    - Add the frame to free frame pool.

Allocation of Frames

- How do we allocate the fixed amount of free memory frames among processes?
- **Example:** Single user system with Memory = 128k.
  - Page size = 1k ∴ there are 128 frames.
  - Operating system = 35 k (35 frames).
  - ∴ there are 93 frames available for the user.
  - Use demand paging, all frames on free frame list.
  - Running a process - 93 page faults satisfied by available frames, then a page replacement algorithm is invoked.
  - When finished, all 93 frames returned to the free frame list.

Variation:

- Operating system and user share memory → allocate operating system buffer and table space from free frame list.
- Basic strategy - user process is allocated with any free frame.

Multi-process system:

- More than one process in memory.
- Minimum number of frames:
  - Cannot allocate more than the total frames available (unless have page sharing).
  - Usually there is a minimum number of frames that must be allocated (defined by instruction set architecture, i.e., the number of pages an instruction can reference).
  - If too few then page fault frequency increases.
- Two major allocation schemes:
  - Fixed allocation.
  - Priority allocation.

Fixed Allocation

Equal allocation:

- m frames, n processes. $\therefore$ m/n frames per process.
- Put the remainder in free frame pool.

Example: 93 frames, 5 processes → 18 / process, and 3 in free frame pool.

Proportional allocation: allocate according to the size of process.

- s_i = virtual memory required for process p_i
- $s = \Sigma s_i$
- m = total number of frames
- a_i = allocation for p_i → $a_i = \frac{s_i}{s} \times m$

- a_i must be larger than minimum number of frames required by instruction set
- Adjust for all to get required allocation and $\sum a_i \leq m$
- $m = 62$, $s_1 = 10$, $s_2 = 127$, $a_1 = 10/137 * 62 \approx 4$, $a_2 = 127/137 * 62 \approx 57$

Priority Allocation

- Use proportional allocation scheme using priorities rather than size
- If process p_i generates a page fault:
  - Select for replacement one of its frames, or
  - 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
    - Thus, one process can take a frame from another.
  - Disadvantage: a process cannot control its own page-fault rate.
  - Generally results in greater system throughput
    - Global is more commonly used method.
- Local replacement
  - Each process selects from only its own set of allocated frames.
  - The number of frames allocated to a process does not change.

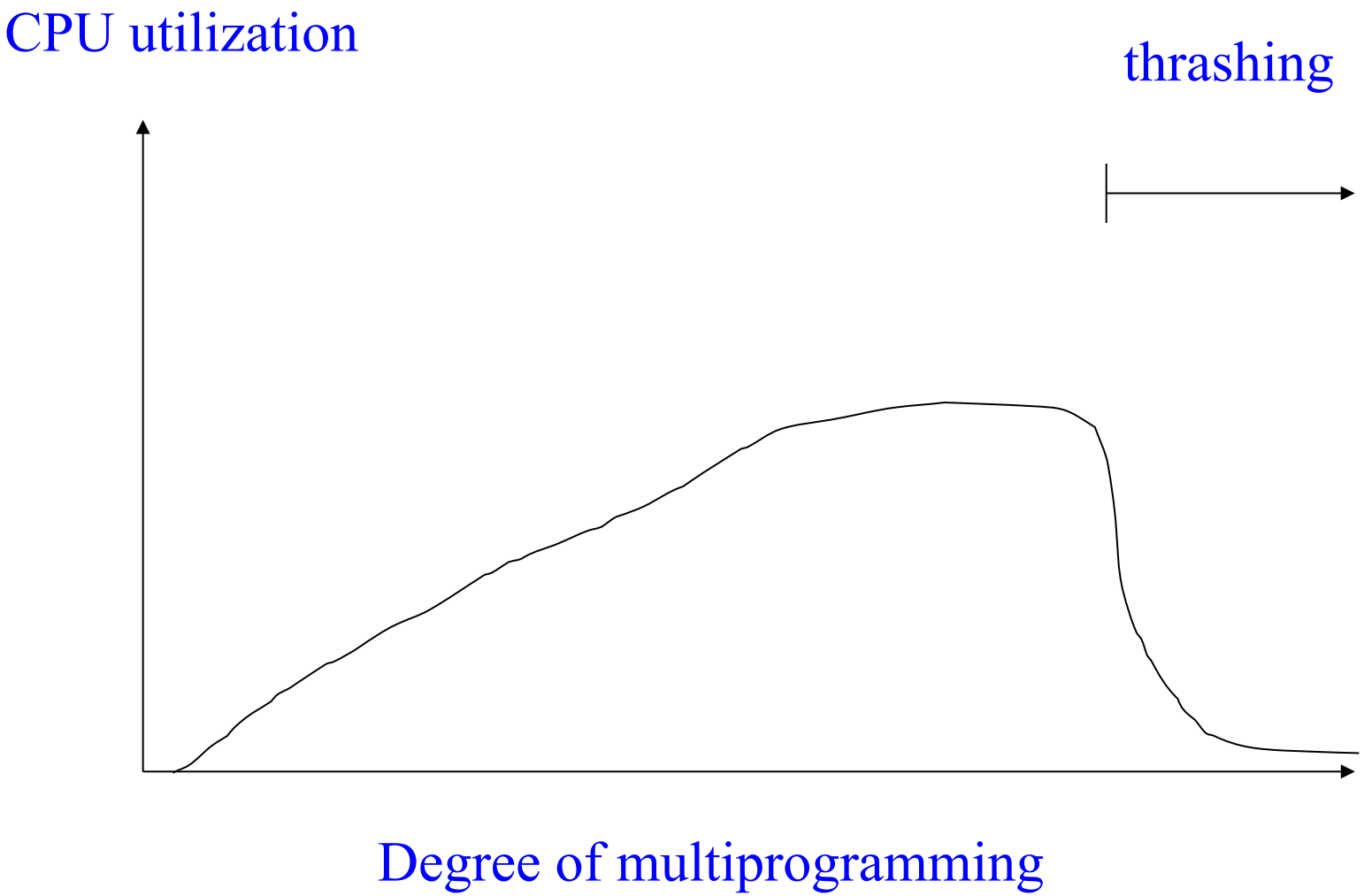
Thrashing

- If a process does not have *enough* frames, the page-fault rate is very high. This leads to:
  - Low CPU utilization
  - OS thinks that it needs to increase the degree of multiprogramming
- Thrashing ≡ each process is busy swapping pages in and out
  - Thus, spending more time paging than executing
- Thrashing results in severe performance problems

How thrashing occurs (Scenario)

1) System monitors CPU utilisation, and tries to maximise the CPU utilisation.
2) If CPU utilisation is low, increase degree of multiprogramming.
   - Introduce another process to the system.
   - Use a global page replacement algorithm (replace any pages regardless of process).
3) A process now needs more frames.
   - Starts page faulting.
   - Steals frames from other processes (these pages are actively being used by other processes!).
4) Other processes start to page fault to retrieve their active pages.
5) Processes get queued up to use the paging device; so, stop using the CPU.
6) CPU utilisation is detected to be low; so, go back to 1).
7) After a few iterations:
   - CPU utilisation has reduced dramatically.
   - Page faulting is high.
   - Virtually there is no productivity.

Thrashing diagram



How to reduce the effect of thrashing?

- Decrease the degree of multiprogramming.
- Use local (or priority) replacement algorithm.

Local:

- Only replace frames belonging to this process.
- Only slows this process down dramatically
  - Slows others as well, but not as much
- Stops other processes thrashing.

Priority:

- Only high priority process can steal frames from low priority processes.
- Low priority processes can be thrashing
  - Low priority processes decrease speed.

How to prevent thrashing?

- Provide a process with as many frames as it needs
- How do we know how many frames a process needs?
- Define *locality model* of process execution
  - Process migrates from one locality to another
  - A program is generally composed of several different localities, which may overlap

  Example:
  - Subroutine
    - Locality: Instructions, local variables, some global variables
    - Active when in the subroutine, else inactive
    - Need to allocate enough frames for a particular locality to prevent thrashing
- Why does thrashing occur?
- Σ size of locality > total memory size

Working set model

- Try to prevent thrashing.
- Working-set model is based on the assumption of locality.
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references.

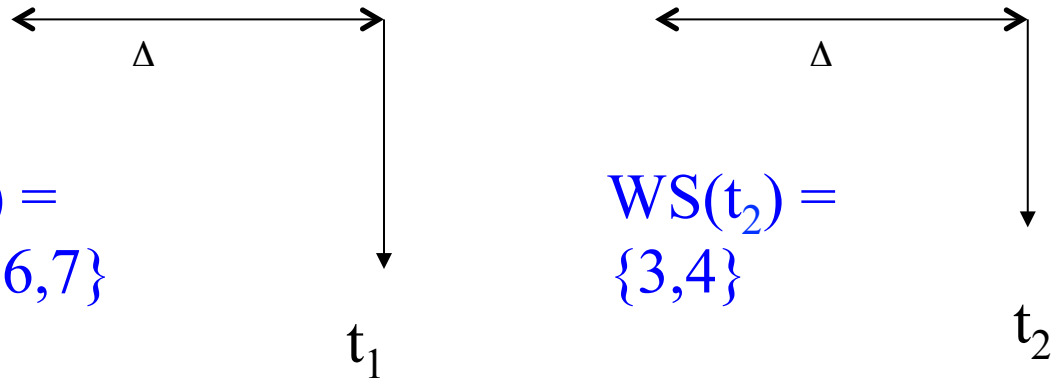
Example: 10,000 instructions.

- WS_i (working set of process P_i) = Total number of pages referenced in the most recent Δ (varies in time) → accuracy depends on Δ.
  - If Δ is too small → locality is not completely represented.
  - If Δ is too large → will encompass several localities.
  - If $\Delta = \infty$ → will encompass entire program.
- $D = \Sigma\ WS_i \equiv$ total demand frames.
- When $D > m$ (i.e., total number of available frames) → thrashing.

Example

Page reference table:

… 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4



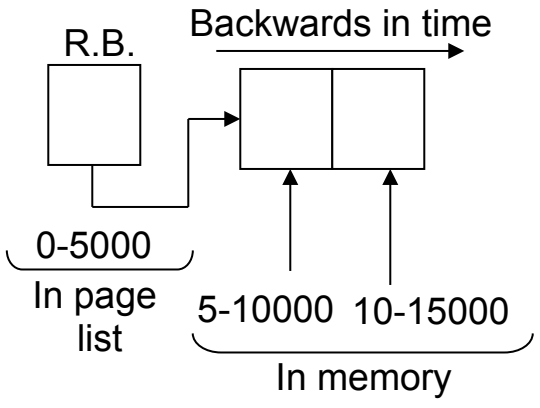
Use of Working set model

- OS monitors the working set of each process and allocates the process enough frames.
- If there are enough extra frames (in free frame pool) → increase degree of multiprogramming.
- If $D > m$, then suspend one of the processes and return frames.
- **Problem:** How to keep track of working set window?

Keeping track of the working set

- Approximate with interval timer and a reference bit (R.B).
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units.
  - Keep in memory 2 bits (B1 and B2) for each page
  - Whenever a timer interrupts, OS copies B1 to B2, R.B to B1, and clears the values of all R.Bs of each other page to 0.
  - If page fault occurs, examine current reference bit, and 2 in memory bits.
  - If one of the bits = 1 → page in working set.

For each page:



- If R.B. or 'in memory" bit = 1 this page was used in the last 15000 references.
- Why is this not completely accurate?
  - It is not known where where in the interval 5000 the reference has occurred.
- Improvement: use 10 bits and interrupt every 1000 time units → cost is high.

Page-Faults Frequency Scheme

- Other thrashing solution.
- Thrashing generates a high page-fault rate.
- Establish an upper and lower bounds of ‘acceptable’ page-fault rate for each process.
  - If actual rate goes below the lower bound, process may have too many frames → remove frames from that process.
  - If actual rate exceeds the upper limit, process needs more frames → allocate that process with more frames.

Other consideration (for paging system)

Page size selection (how big is best?)

- Ranging from 4KB bytes to 4 MB.
- Smaller size → needs larger sized page table
  - Larger size → waste more on internal fragmentation.
- From I/O point of view:
  - Larger size: minimize I/O time per byte and minimize number of page faults.
  - Smaller size: fits in more for locality of reference
    - Have better *resolution* → less total I/O and less total memory allocated.
- Historical trend upwards:
  - 1990 - 4 Kbytes maximum; 80386: 4 Kbytes,
  - 68030: 256 → 32kbytes.
  - Probably because CPU and memory speeds increases faster than disk speeds
    - page faults become more expensive now
    - reduce page fault frequency by increasing page size;

Other consideration (cont.)

Prepaging

- When reloading a suspended process, load all the working set of pages to prevent many page faults on restart.
- **Problem:** for some cases, many of the pages brought into memory by prepaging are not used.

TLB Reach

- The amount of memory accessible from the TLB.
- TLB Reach = (TLB Size) * (Page Size)
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of referencing page table.

How to increase the size of TLB reach?

- **Increase the Page Size**. May increase internal fragmentation as not all applications require a large page size.
- **Provide Multiple Page Sizes**. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.

Other consideration (cont.)

Program structure

- User has control over performance.
- Stack: Good locality, always accessed at the top.
- Hash table, Dynamic array etc: Poor locality, scattered references.
- Example: in C

  var A: array[1..128] of array[1..128] of integer;

  Row major: A[1][1], A[1][2],...,A[1][128],...,A[128][128].

  Page size = 128 words → one row per page.

  Access *i* most frequently in A[*i*][*j*] will get many page faults (poor locality).

Program 1 (128 x 128 = 16,384 page faults)

```
For j := 1 to 128 do
    For i := 1 to 128 do
        A[i, j] := 0;
```

Program 2 (128 page faults)

```
For i := 1 to 128 do
    For j := 1 to 128 do
        A[i, j] := 0;
```

I/O interlock and addressing

Problem of a process requesting I/O if the I/O is done to/from **user** virtual memory.

Example:

- Process issue an I/O request → put in queue to access device.
  - What if its buffer space (for I/O) is swapped out by page replacement algorithm?
  - When I/O runs, it overwrites buffer space (now used by another process)!

Solutions:

(i) Buffer in Operating System space

- Time consuming

I/O … (cont.)

Solution (cont.)

(ii) Lock buffer pages in memory

- A lock bit is associated with every frame → if its lock bit is set, the frame cannot be selected for replacement.
- To write a block on a tape → lock the page containing the block, unlock the page when I/O completes.

Other use of lock bit

Consider a low priority process that has a newly brought page waiting for CPU. High priority process steals pages as low priority process hasn't used them for a long time.

- Use lock bit to prevent replacing a newly brought in page until it can be used at least once.

Problem with lock bit: Process crashes while page is locked - not released.

Kernel Memory Allocation

- Efficient memory management is important in OS
  - There are many dynamic / short live data structures / objects used in the kernel and for user applications that should be done quickly
    - e.g., each time a thread / process is created, OS allocates memory (stacks, data structures, etc) and de-allocates the memory when it terminates
- Kernel memory is allocated from a free memory pool different than the pool for user processes; two main reasons:
  - Kernel requests memory in various sizes that might be less than one page
    - Need to reduce the possible wastes
    - Many OS do not use the paging system for their kernel code and data
  - Some hardware devices may require contiguous memory location
    - The hardware may interact directly with the physical memory
    - It is different from the virtual memory used in user processes in which pages are stored in non contiguous physical memory
- How to manage free memory for kernel processes?
  - Buddy system
  - Slab allocation

Buddy system

- Allocate memory from a fixed sized segment consisting of physically contiguous pages
  - Linux before ver 2.2 used buddy system
  - The size of each allocated memory is always in a power-of-two
    - E.g., 4 KB (= 2^{12} bytes), 8 KB, etc → called a **power-of-2** allocator
    - Round up to the next power-of-2 larger than requested memory, e.g., E.g., a request for 21 KB of memory is allocated with memory of size 32 KB
- A segment is sequentially divided into two **buddies** (of the same size) until it finds a buddy that is closely satisfy the request
  - E.g., a system has a block of size 256 KB and a request for 21 KB of memory.
    - As shown, keep divide the block until there is C_L = 32 KB that is sufficiently large for the request (= 21 KB)

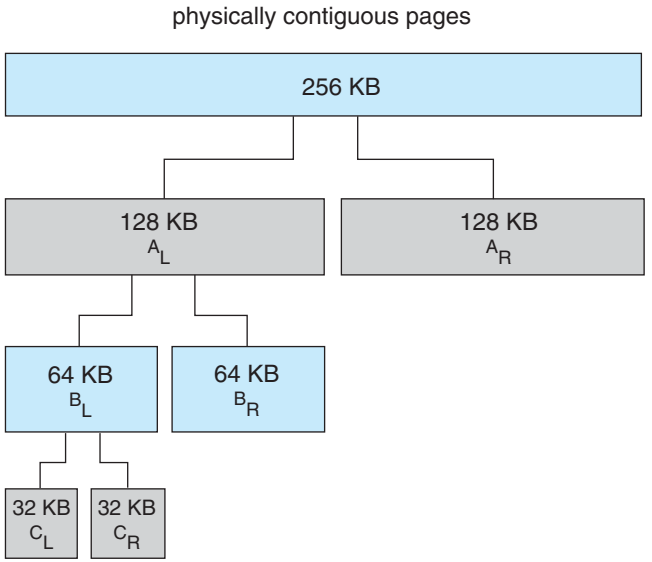


Figure 9.26 (textbook)

Buddy system (cont.)

- Advantages of buddy system:
  - Finding a free memory block is simple
  - Coalescing adjacent buddies into one large segment is fast
    - E.g., assume the memory C_L (32 KB) is no longer used, i.e., the kernel process releases it
    - C_L is coalesced with C_R to form a larger segment B_L of size 64 KB
    - B_L is coalesced with B_R to form segment A_L of size 128 KB
    - A_L is coalesced with A_R to form the original segment segment A of size 256 KB
- Disadvantages:
  - Memory waste due to internal fragmentation
  - Memory waste close to 50% is possible

Slab allocation

- Many type of objects in the kernel are quickly allocated and de-allocated
  - Allocation includes finding sufficient memory block, initializing data structures, pointers, values, etc.
  - De-allocation includes freeing the allocated memory for others to use
  - There is time overhead in memory allocation / de-allocation, in addition to memory waste (e.g., internal fragmentation)
- There is a single **cache** to store each unique type of kernel data structure / object
  - The objects can be data structures for PCB, file structures, device structures, semaphore, etc.
- A cache consists of one or more **slabs**
  - A **slab** is composed of one or more physically contiguous pages
  - A **constructor** is used to allocate a slab and initialize the memory area to hold the objects
  - When a cache is created, some objects (marked free) are allocated to the cache
  - The number of objects in the cache depend on the size of its slabs and its object size, e.g., a slab with five pages (of 4 KB / page) for a cache of 2 KB objects can store ten objects
  - When kernel requests for an object for its data structure, **slab allocator** assigns one **free** object for the request, and marks it **used**

Slab allocation (cont.)

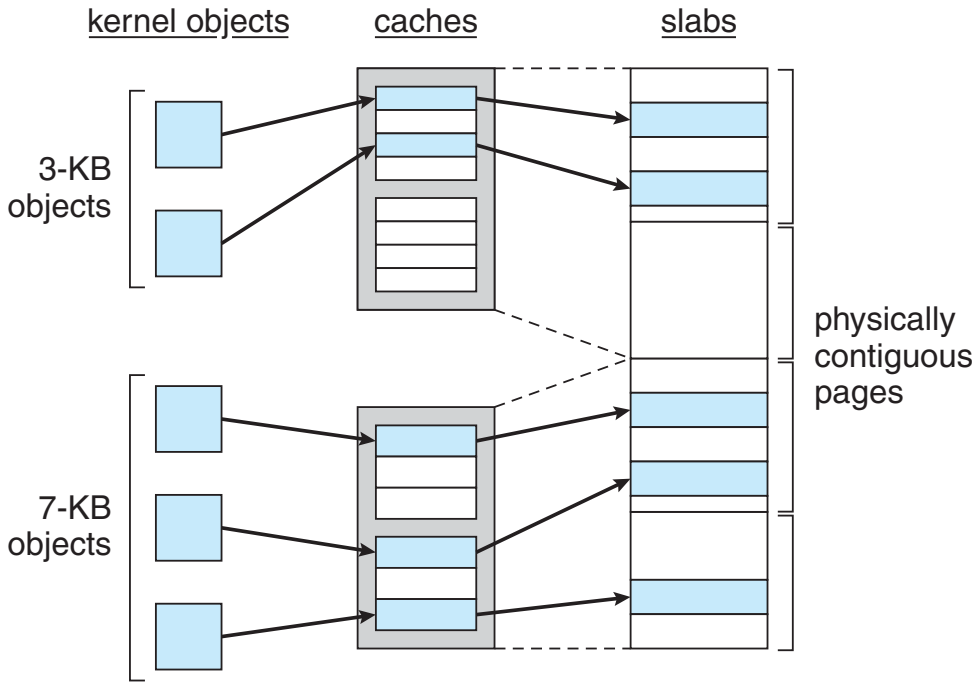


Figure 9.27 (textbook)

Slab allocation

- There are three possible states of a slab in Linux:
  - **Full** – all objects in the slab are "used"
  - **Empty** – all objects in the slab are "free"
  - **Partial** – some objects in the slab are "free" some others are "used"
- Use free object from the partial slab first
  - No partial slab? Use objects from the empty slab
  - No empty slab? Create a new slab for the cache from contiguous physical pages
  - All slabs are empty? OS can de-allocate the slabs if it needs memory for other use.
- Advantage
  - No internal fragmentation
    - Memory request for each object is allocated with memory exactly the size of the object
  - Fast memory allocation / de-allocation
    - Objects are created in advance, and thus can be allocated immediately
    - Release the memory by setting it "free" which can be used immediately by any later request
- **Slab** allocator was used starting Linux 2.2
  - **Slob** allocator is used for system with limited memory (e.g., embedded system)
  - **Slub** allocator is used starting Linux 2.6.24