

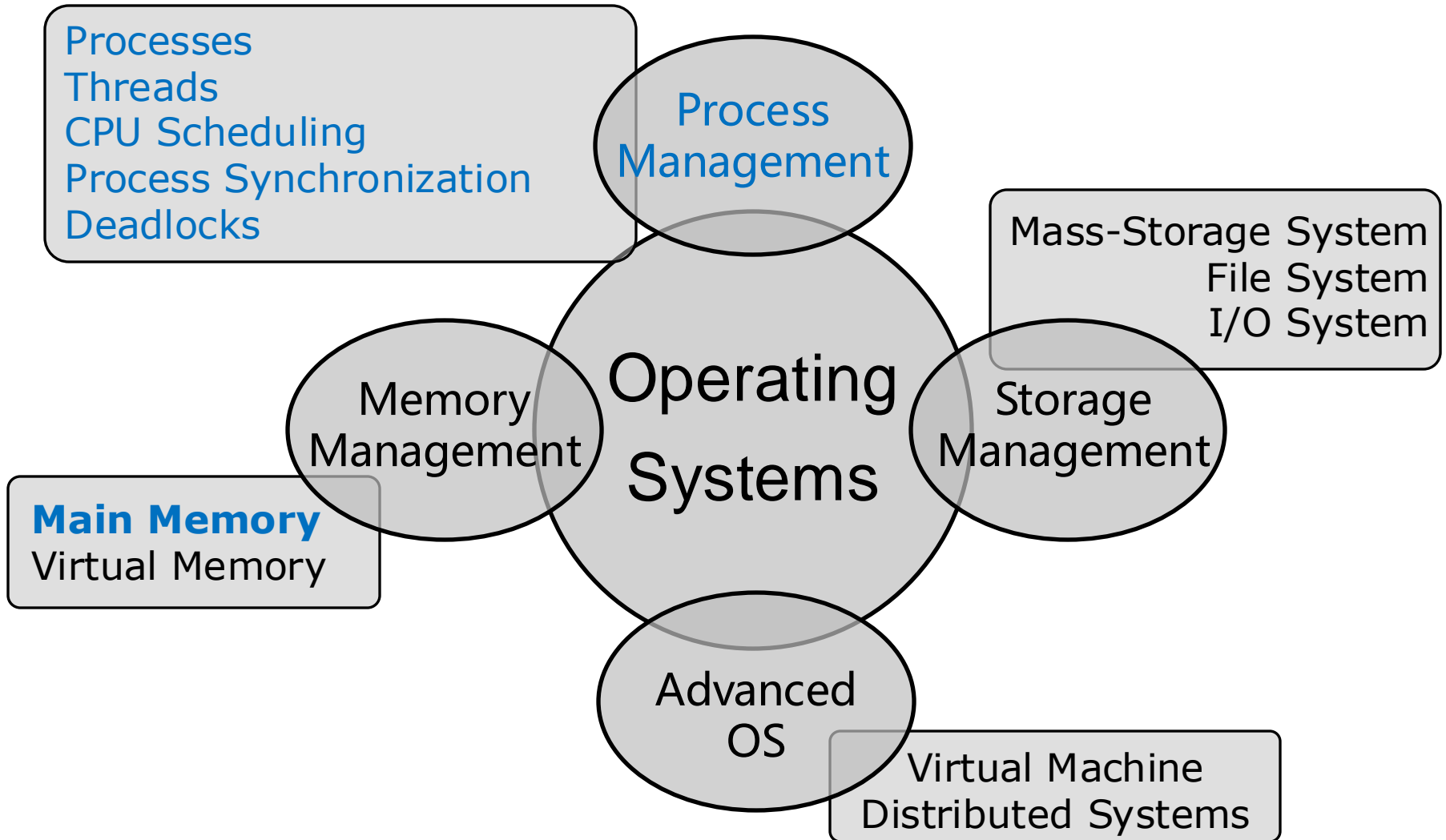
Main Memory

Shengzhong Liu

Department of Computer Science and Engineering

Shanghai Jiao Tong University

Operating System Topics



Outline

- Background
- Contiguous Memory Allocation
- Non-contiguous Memory Allocation
 - Paging
 - Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures

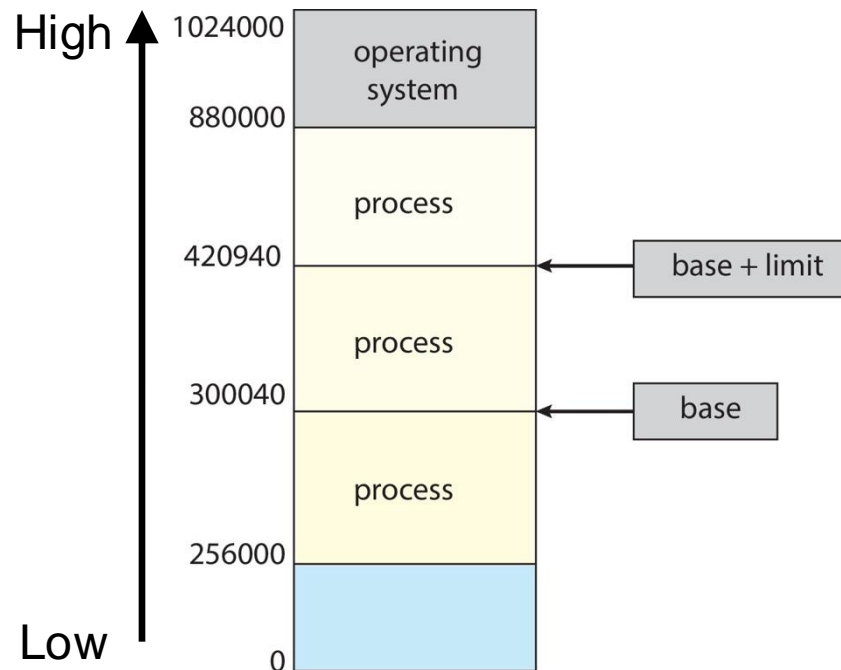
Background

Background

- ❑ Program must be **brought from disk into memory** and placed within a process to run
- ❑ **Main memory** and **registers** are only storage CPU can access directly
- ❑ Memory unit only sees a stream of:
 - ❑ address + **read requests**
 - ❑ address + data + **write requests**
- ❑ Speed:
 - ❑ Register access is done in one CPU clock (or less)
 - ❑ Main memory can take many cycles, causing a **memory stall**
 - ❑ **Cache** sits between main memory and CPU registers
- ❑ Protection of memory required to ensure correct operation

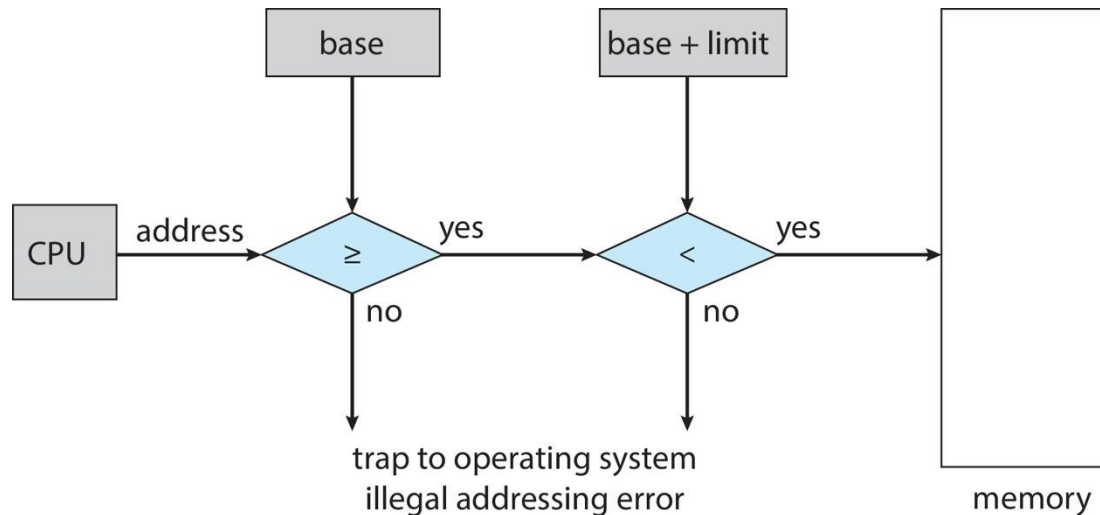
Protection

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** to define the logical address space of a process



Hardware Address Protection

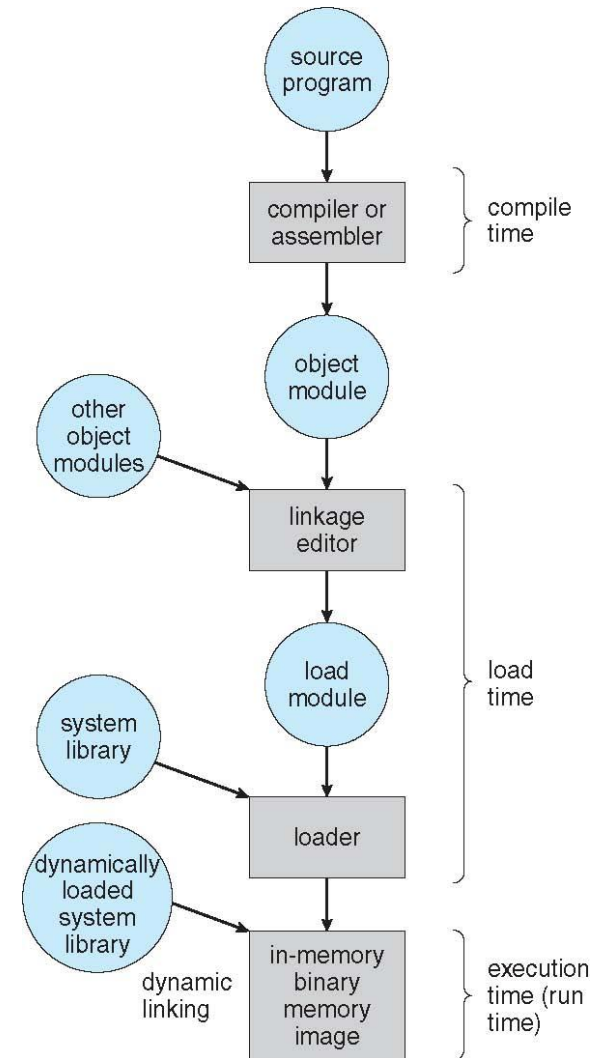
- CPU must check every memory access generated in user mode to be sure it is **between base and limit for that user**



- The instructions to load the base and limit registers are privileged

Address Binding

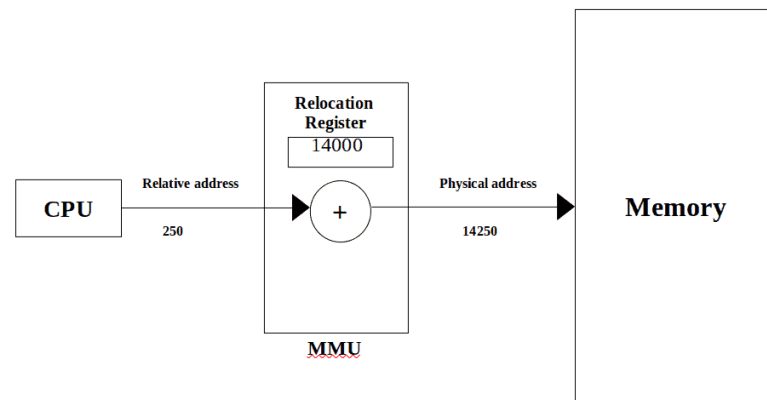
- Programs on disk, ready to be brought into memory to execute form an **input queue**
 - Without support, must be loaded into address 0000
- Addresses represented in different ways at different stages of a program's life
 - **Source code** addresses usually symbolic
 - **Compiled code** addresses bind to relocatable addresses
 - ▶ e.g., “14 bytes from beginning of this module”
 - **Linker or loader** binds relocatable addresses to absolute addresses
 - ▶ e.g., 74014
 - Each binding maps one address space to another



Processing User Program

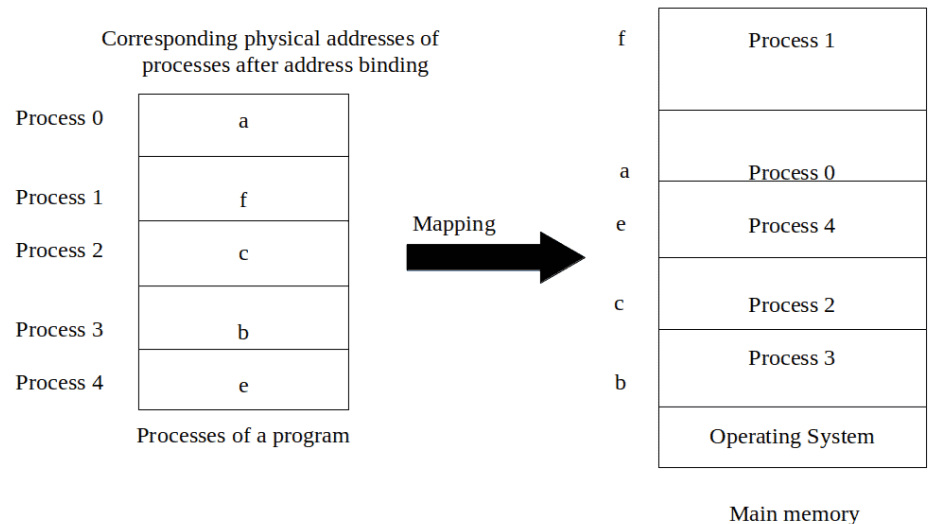
Binding of Instructions and Data to Memory

- When address binding of instructions and data to memory addresses happen?
 - **Compile time:** If memory location is known, **absolute code** can be generated;
 - ▶ Must recompile code if starting location changes
 - **Load time:** The compiler generates **relocatable code** if memory location is not known at compile time
 - **Execution time:** Binding delayed until runtime if the process can be moved during its execution
 - ▶ Need hardware support for address maps (e.g., base and limit registers)
 - ▶ Most operating systems use this method



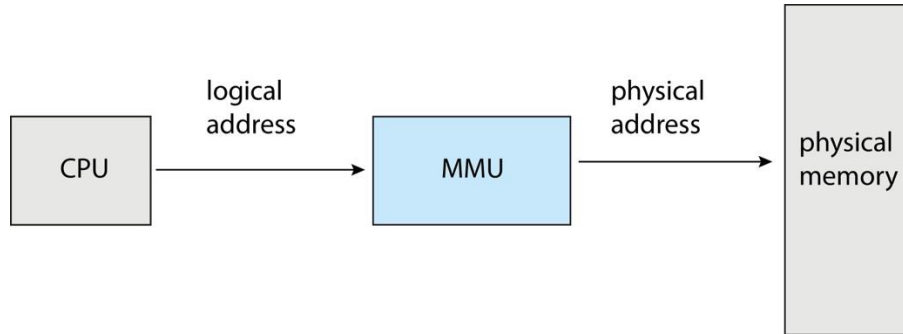
Logical vs. Physical Address Space

- A logical address space is bound to a separate **physical address space** in memory management
 - **Logical address (virtual address)** – generated by the CPU;
 - ▶ Logical address space: Set of all logical addresses generated by a program
 - ▶ **Range**: 0 to max
 - **Physical address** – address seen by the memory unit
 - ▶ Physical address space: Set of all physical addresses generated by a program
 - ▶ **Range**: base + 0 to base + max
- Relations: Logical and physical addresses are
 - the same in **compile-time** and **load-time** address-binding schemes;
 - different in **execution-time** address-binding scheme



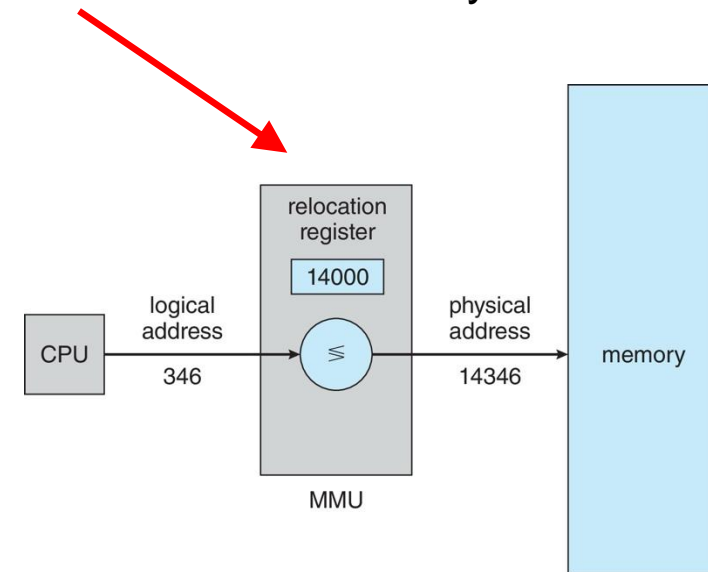
Memory-Management Unit (MMU)

- Hardware device that at run time maps logical to physical address



- Consider a simple scheme, where the value in the relocation register is added to every address generated by a user process when it is sent to memory
 - Base register now called **relocation register**

- The user program deals with **logical addresses**; it never sees the **real physical addresses**
 - Execution-time binding occurs when reference is made to location in memory
 - Logical address bound to physical addresses



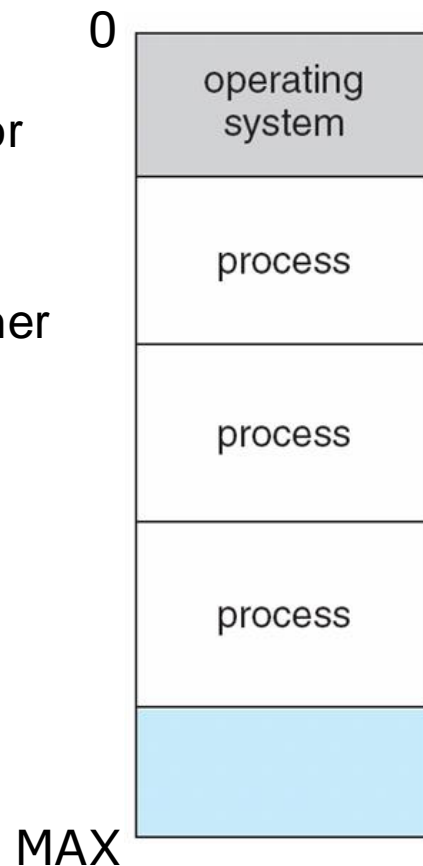
Memory Management

- Contiguous memory allocation
- Non-contiguous memory allocation
 - Paging

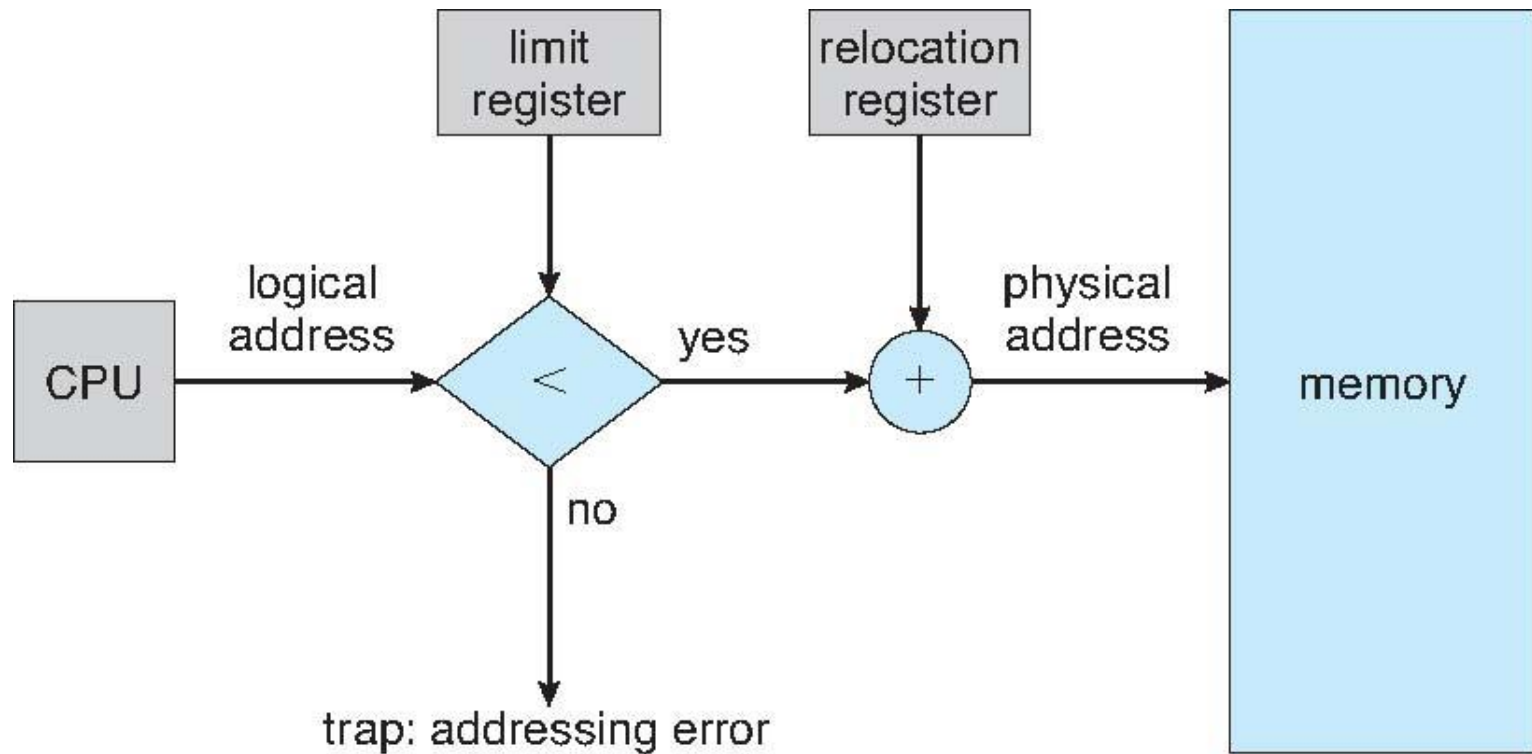
Contiguous Memory Allocation

Contiguous Allocation (连续分配)

- ❑ Each process is contained in a contiguous section of memory
- ❑ Main memory usually contains two partitions:
 - ❑ **Resident OS:** held in low memory with interrupt vector
 - ❑ **User processes:** held in high memory
- ❑ Relocation registers protect user processes from each other and from changing OS code and data
 - ❑ **Base register:** smallest physical address
 - ❑ **Limit register:** range of logical addresses
 - ❑ Each logical address must $<$ the limit register
 - ❑ **MMU** maps logical addresses *dynamically*
- ❑ Can allow actions like
 - ❑ kernel code being transient
 - ❑ kernel changing size

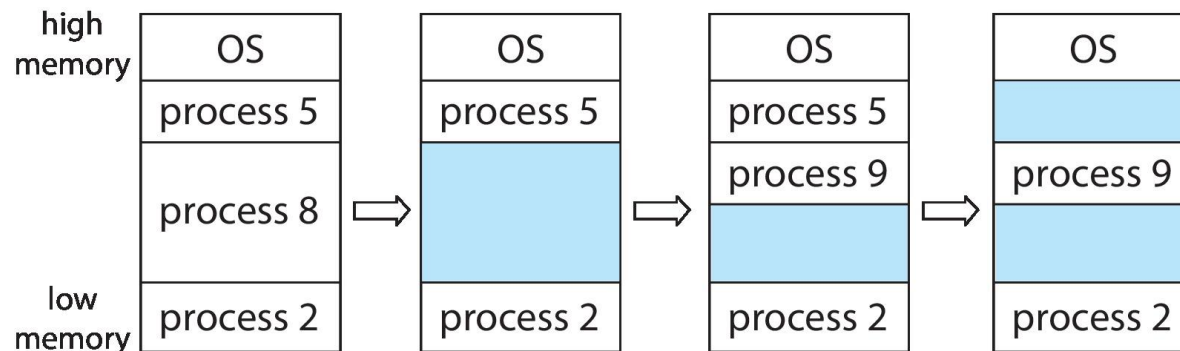


Hardware Support for Relocation and Limit Registers



Variable Partition

- ❑ Multiple-partition allocation
 - ❑ Degree of multiprogramming limited by number of partitions
 - ❑ **Variable-partition** sizes for efficiency (sized to a given process' needs)
 - ❑ **Hole** – 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
- ❑ Exiting process frees its partition, and combines with adjacent free partitions
- ❑ OS maintains information about: a) allocated partitions; b) free partitions (hole)



Dynamic Storage-Allocation Problem

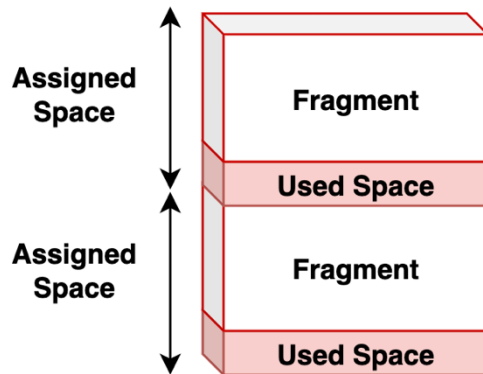
How to satisfy a request of size n from a list of free holes?

- **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 hole
- **Worst-fit**: Allocate the **largest** hole; must also search entire list
 - Produces the largest leftover hole

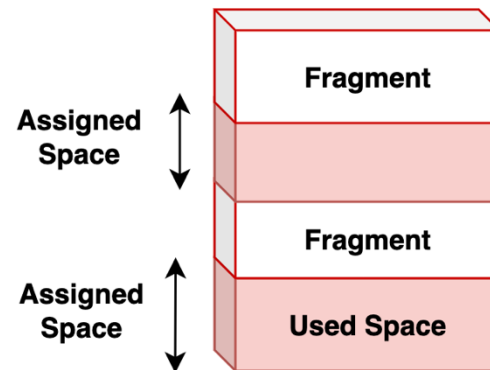
First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation (碎片化)

- ❑ **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- ❑ **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- ❑ First fit analysis reveals that given N blocks allocated, another $0.5N$ blocks lost to fragmentation
 - ❑ $1/3$ may be unusable -> **50-percent rule**



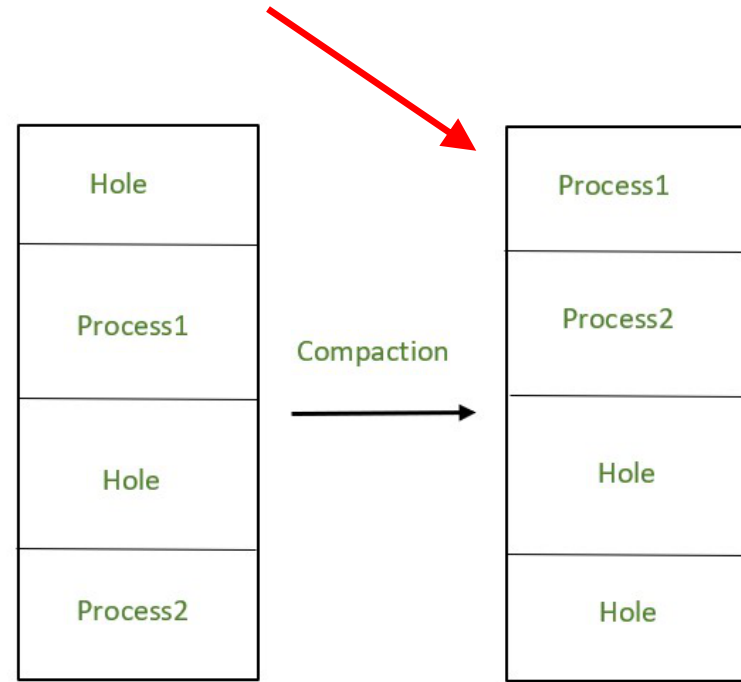
Internal Fragmentation



External Fragmentation


Fragmentation (Cont.)

- ❑ Reduce external fragmentation by **compaction (压缩)**
 - ❑ Shuffle memory contents to place all free memory together in one large block
 - ❑ Compaction is possible only if relocation is dynamic and done at execution time
 - ❑ Simple compaction strategy:
 - ▶ Move all processes toward one end of memory; all holes move in the other direction
 - ▶ **Expensive!**
- ❑ Another solution to permit the logical address space of the processes to be **noncontiguous**
 - ❑ **paging**
 - ❑ **segmentation**

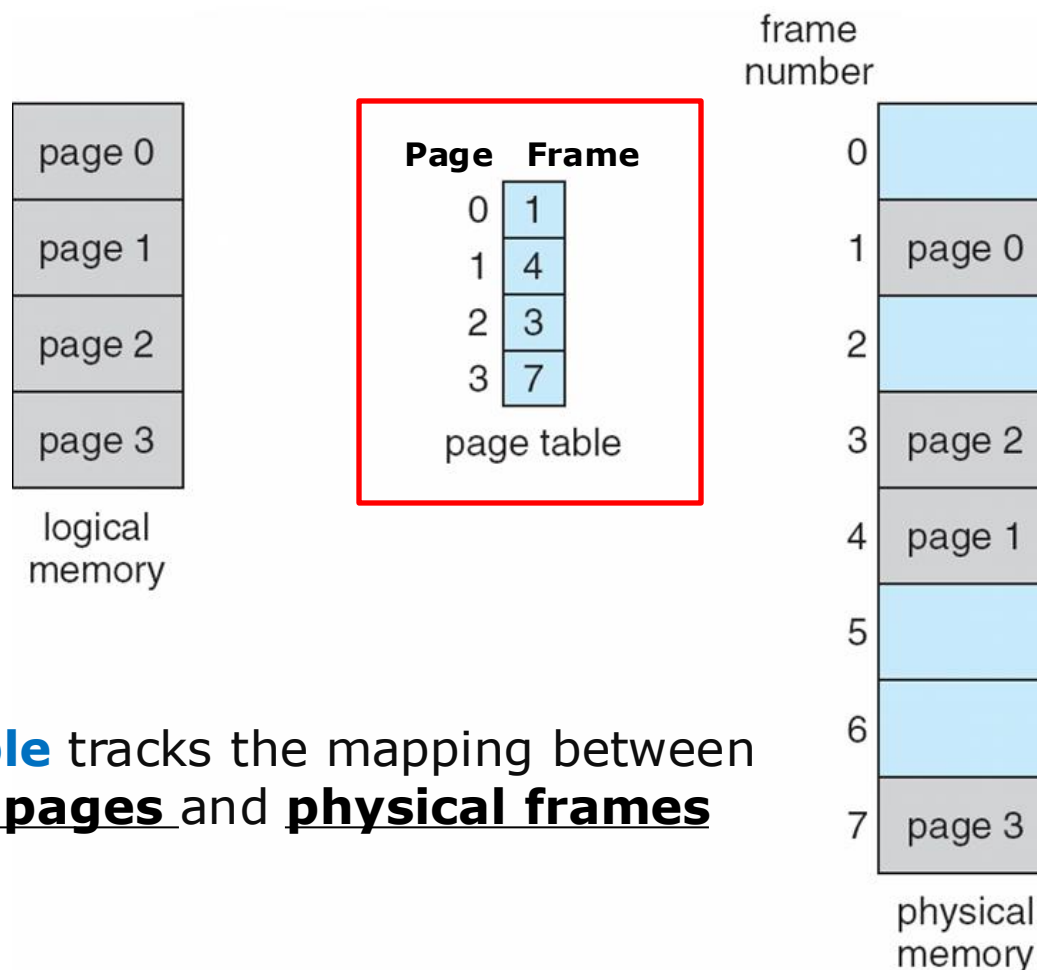


Paging

Paging

- ❑ Physical address space of a process can be **noncontiguous**;
 - ❑ Process is allocated physical memory whenever available
 - ❑ Avoids external fragmentation, but still has internal fragmentation
 - ❑ Avoids problem of varying-sized memory chunks
 - ❑ Divide **physical memory** into fixed-size blocks called **frames**
 - ❑ Size is power of 2, between 512 bytes and 16 Mbytes
 - ❑ Keep track of all free frames
 - ❑ Divide **logical memory** into blocks of the same size called **pages**
 - ❑ To run a program of **N** pages, need to find **N** free frames
 - ❑ Backing store similarly split into pages
 - ❑ Set up a **page table** to translate logical to physical addresses
- 

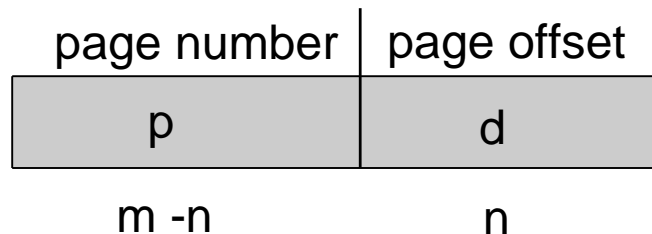
Paging Model of Logical and Physical Memory



Page table tracks the mapping between virtual pages and physical frames

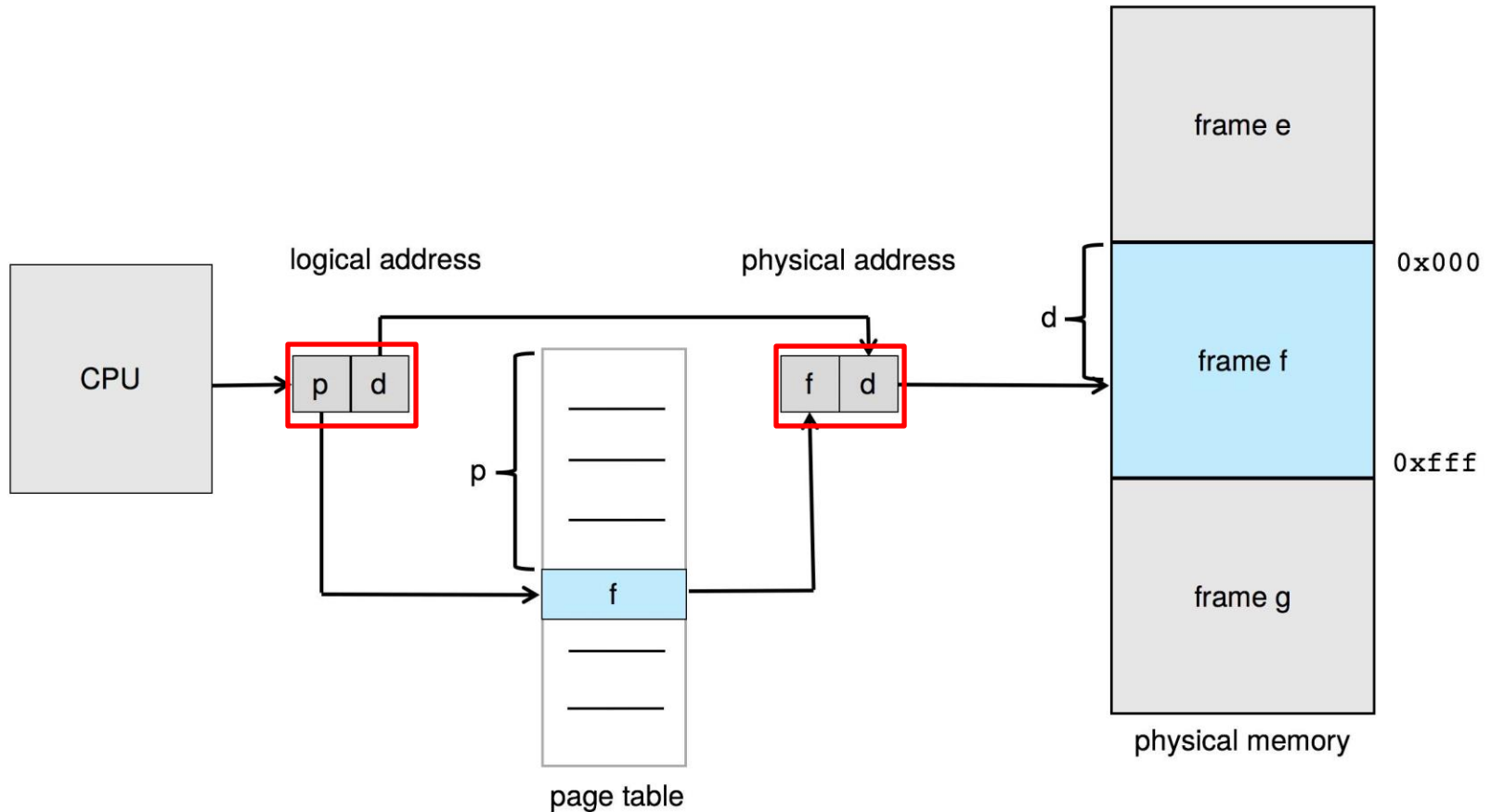
Address Translation Scheme

- Address generated by CPU is divided into:
 - **Page number (p)** – an index into a page table which contains base address of each page in physical memory
 - ▶ **Page table** tracks the mapping between virtual pages and physical frames
 - **Page offset (d)** – combined with base address to define the physical memory address for the memory unit
 - ▶ **Page offset remains the same** between virtual page and physical frame.



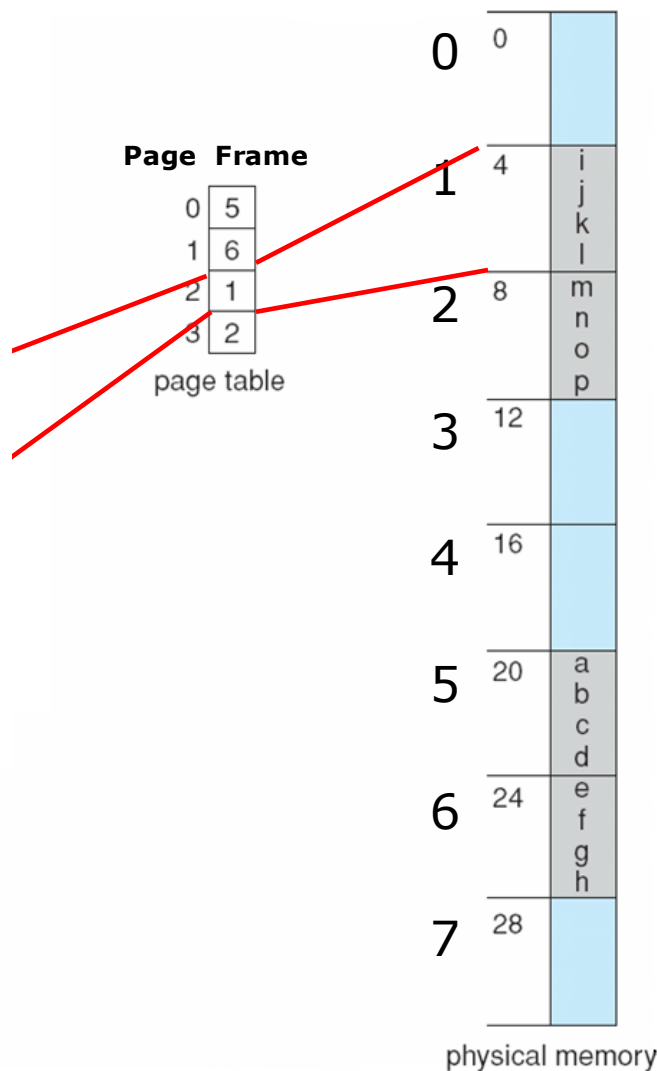
- For given logical address space 2^m and page size 2^n
 - ▶ Page number takes $m-n$ bits
 - ▶ Page offset takes n bits

Paging Hardware



MMU translates virtual page number into physical frame number

Paging Example



What character does the following logical addresses map to?

(A) 10 = 10 | 10 = **k**

(B) 5 = 01 | 01 = **f**

(C) 14 = 11 | 10 = **o**

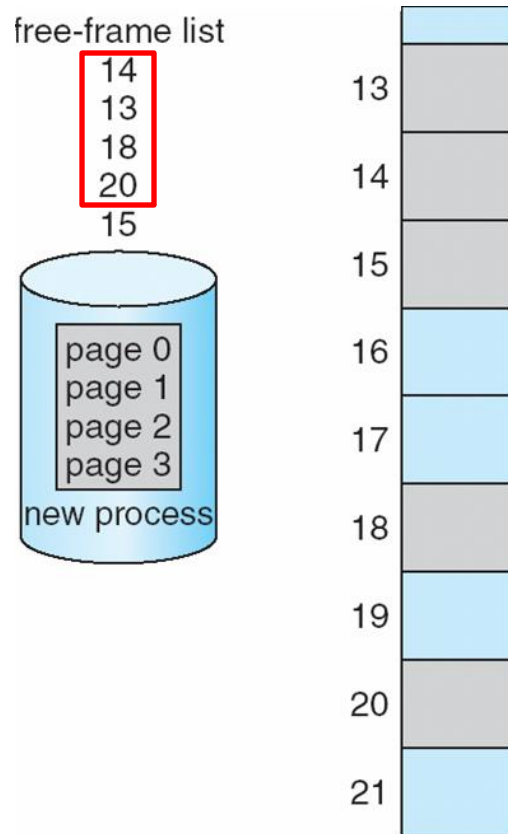
Page Number | Page Offset

$n=2$ and $m=4$ 32-byte memory and 4-byte pages

Paging (Cont.)

- Calculating internal fragmentation
 - Page size = 2,048 bytes
 - Process size = 72,766 bytes
 - 35 pages + 1,086 bytes
 - Internal fragmentation of 2,048 - 1,086 = **962 bytes**
- Internal fragmentation
 - Worst case fragmentation = frame size – 1 byte
 - On average fragmentation = 1/2 frame size
- Calculate the page numbers and offsets for the following address, when page size is 1KB:
 - $2375 = 1024 * \mathbf{2} + \mathbf{327}$
 - $19366 = 1024 * \mathbf{18} + \mathbf{934}$

Free Frames



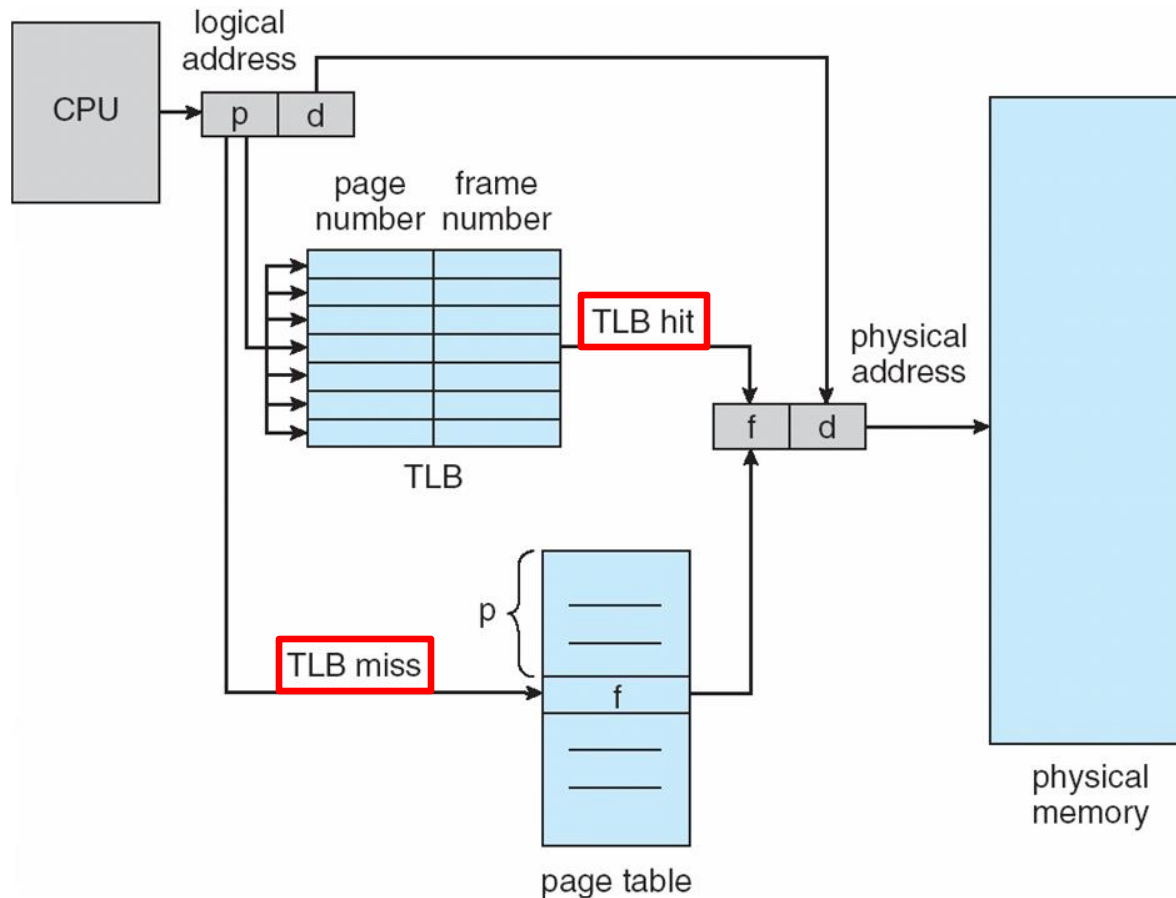
(a)

Before allocation

Implementation of Page Table

- Page table is kept in main memory
 - **Page-table base register (PTBR)** points to the page table
 - **Page-table length register (PTLR)** indicates size of the page table
 - Every data/instruction access requires two memory accesses
 - ▶ First access for the page table
 - ▶ Second access for actual data / instruction
- The double-access problem can be solved by a special fast-lookup hardware cache called **translation look-aside buffers (TLB)**
 - TLBs typically small (64 to 1,024 entries)
 - On a TLB miss, the value is loaded into the TLB for faster access next time
 - ▶ Replacement policies must be considered
 - ▶ Some entries can be **wired down** for permanent fast access
 - They can not be removed from the TLB.

Paging Hardware With TLB



Effective Access Time with TLB

- **Hit ratio** – percentage of times that a page number is found in the TLB
 - 80% hit ratio means we find the page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
 - If we find the desired page in TLB then a mapped-memory access take 10 ns
 - Otherwise we need two memory access so it is 20 ns

- **Effective Access Time (EAT)**

$$\text{EAT} = 0.80 \times 10 + 0.20 \times 20 = 12 \text{ nanoseconds}$$

implying **20% slowdown** in access time

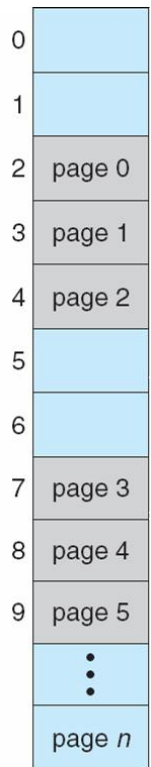
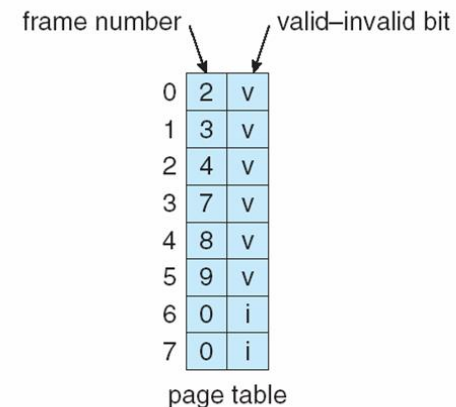
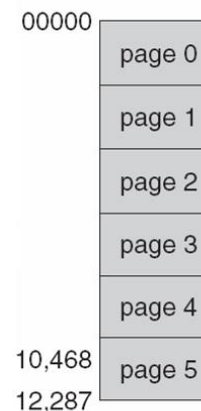
- Consider a more realistic hit ratio of 99%,

$$\text{EAT} = 0.99 \times 10 + 0.01 \times 20 = 10.1\text{ns}$$

implying only **1% slowdown** in access time

Memory Protection

- ❑ Memory protection implemented by associating **protection bit** with each frame to indicate if **read-only** or **read-write access** is allowed
 - ❑ Can also add more bits to indicate page execute-only, and so on
- ❑ **Valid-invalid** bit attached to each entry in the page table:
 - ❑ “**valid**”: the page is in the process logical address space, and is thus a legal page
 - ❑ “**invalid**”: the page is not in the process logical address space
- ❑ Any violations result in a trap to the kernel !



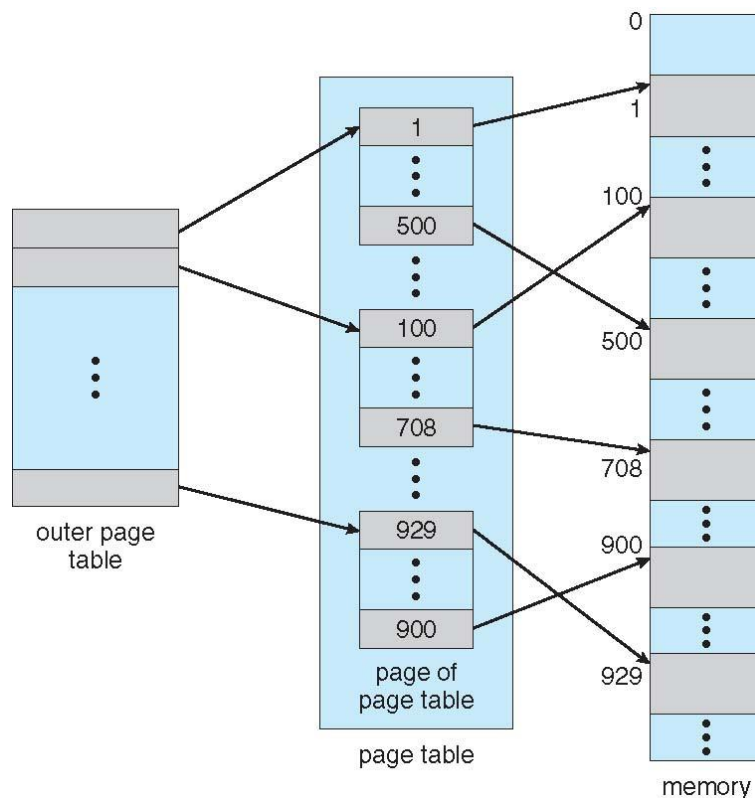
Structure of Page Table

Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
 - Consider a 32-bit logical address space as on modern computers
 - ▶ Page size of 4 KB (2^{12})
 - ▶ Page table would have **1 million entries** ($2^{32} / 2^{12}$)
 - If each entry is 4 bytes → each process 4 MB of physical address space for the page table alone
 - ▶ **Do not want to allocate that space contiguously in the main memory**
 - One simple solution is to divide the page table into smaller units
 - ▶ Hierarchical Paging
 - ▶ Hashed Page Tables
 - ▶ Inverted Page Tables

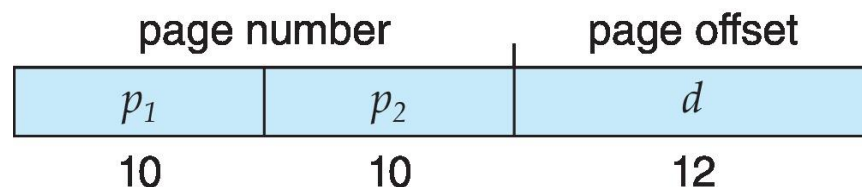
Hierarchical Page Tables

- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then **page the page table**



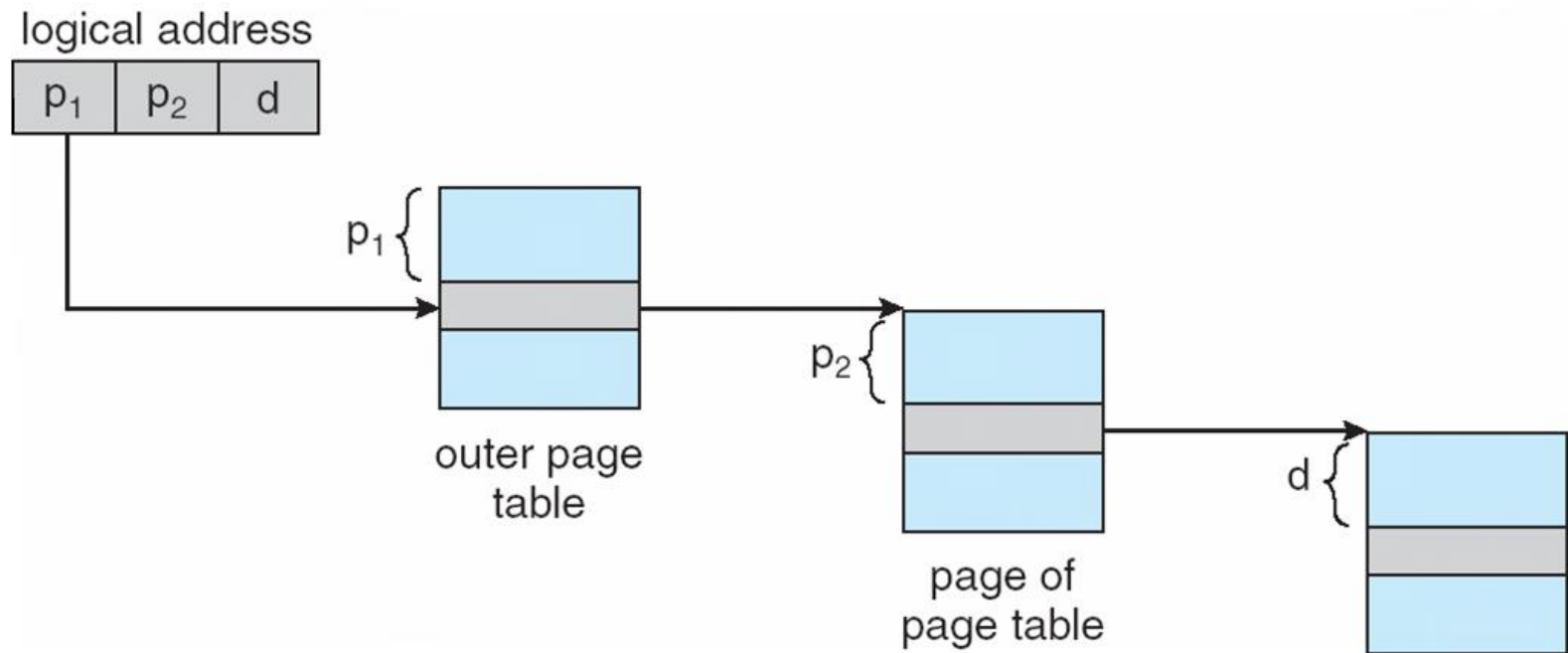
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
 - a page number consisting of 20 bits
 - 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:



- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
 - Then page table has 2^{52} entries
 - If two-level scheme, inner page tables could be 2^{10} 4-byte entries
 - Address would look like

outer page	inner page	offset
p_1	p_2	d
42	10	12

- Outer page table has 2^{42} entries or 2^{44} bytes
- One solution is to add a 2nd outer page table
- But in the following example the 2nd outer page table is still 2^{34} bytes in size
 - ▶ And **possibly 4 memory accesses to get to one physical memory location**

Three-level Paging Scheme

outer page	inner page	offset
p_1	p_2	d
42	10	12

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

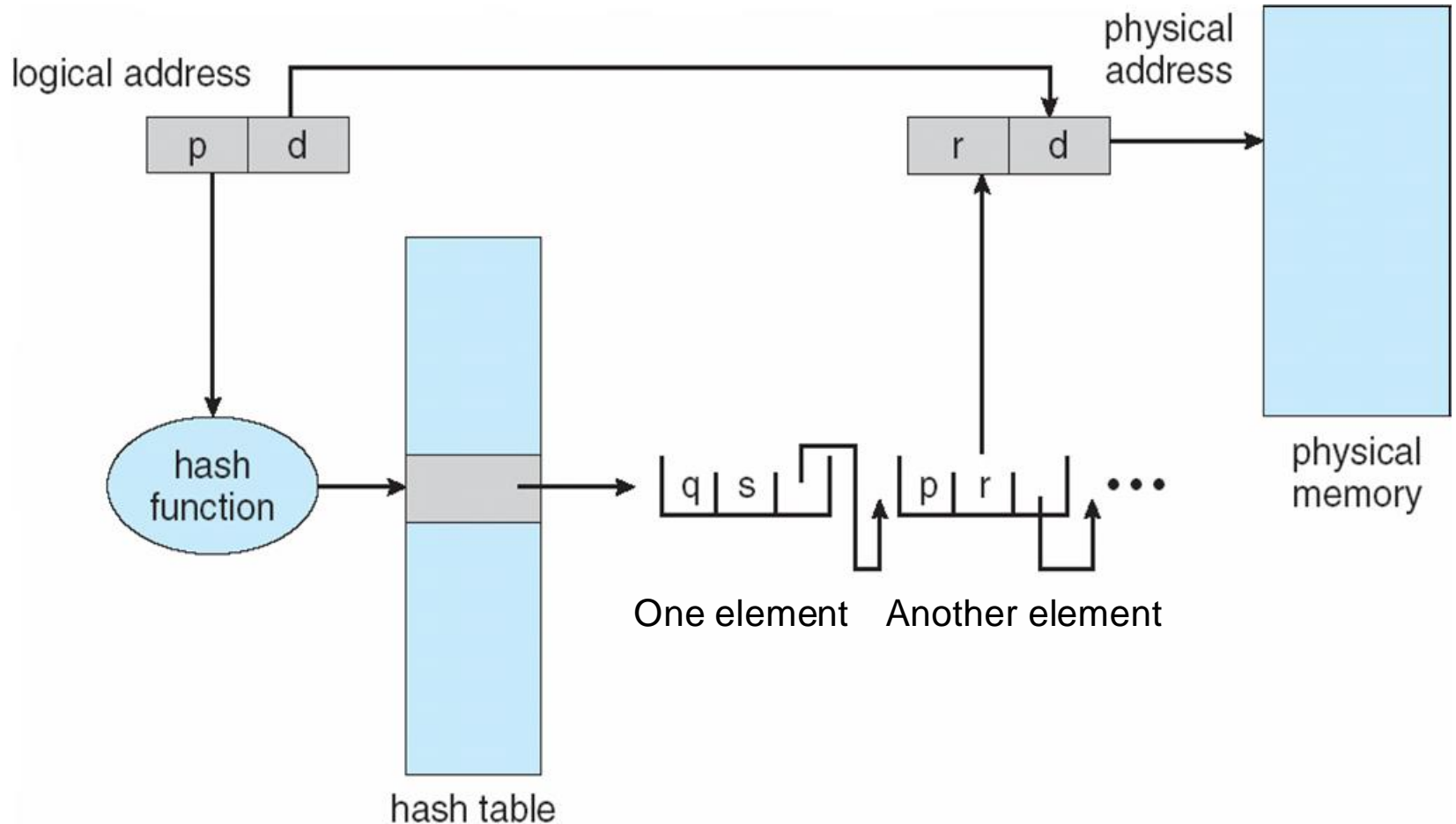
Hierarchical Page Table

Benefits	Drawbacks
Reduced Memory Overhead: Only portions of the address space that are actively used need to be mapped	Increased Memory Access Latency: Traversing multiple levels of page tables to translate a virtual address to a physical address can incur overhead
Scalability: Accommodate the increased address size without a proportional increase in memory overhead	Increased Page Table Size: In systems with very large address spaces, consuming significant amounts of memory to store page table entries
Improved Locality: The system can exploit spatial locality, as neighboring virtual pages often map to nearby physical frames	Complexity: More complex to implement and manage compared to flat page tables
Flexibility: Depending on the system requirements and memory characteristics, different levels of hierarchy can be used	Fragmentation: Hierarchical page tables can suffer from fragmentation, especially in systems with sparse address spaces

Hashed Page Tables

- ❑ Common in address spaces > 32 bits
- ❑ The virtual page number is hashed into a page table
 - ❑ This page table contains a **chain of elements** hashing to the same location
 - ❑ Each element contains (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element
- ❑ Virtual page numbers are compared in this chain searching for a match
 - ❑ If a match is found, the corresponding physical frame is extracted
- ❑ Variation for 64-bit addresses is **clustered page tables**
 - ❑ Similar to hashed but each entry refers to several pages (such as 16) rather than 1
 - ❑ Especially useful for sparse address spaces
 - ▶ Memory references are non-contiguous and scattered

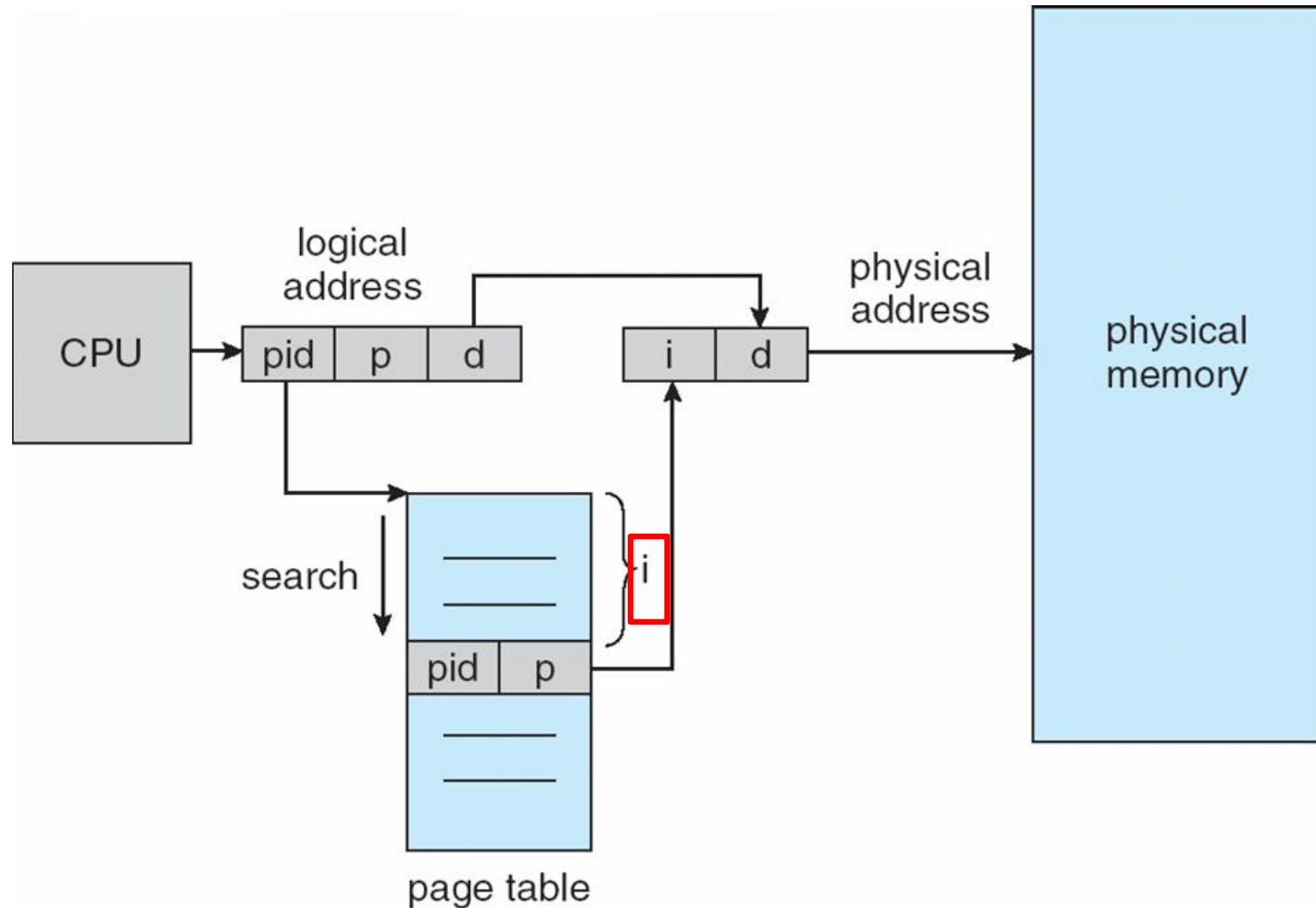
Hashed Page Table



Inverted Page Table

- ❑ Rather than each process having a page table and keeping track of all possible logical pages, **track all physical frames**
 - ❑ One entry for each real frame of memory
- ❑ Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- ❑ Reduces used memory for page table, but **increases time to search the table for page reference**
- ❑ Can use TLB to accelerate access

Inverted Page Table Architecture

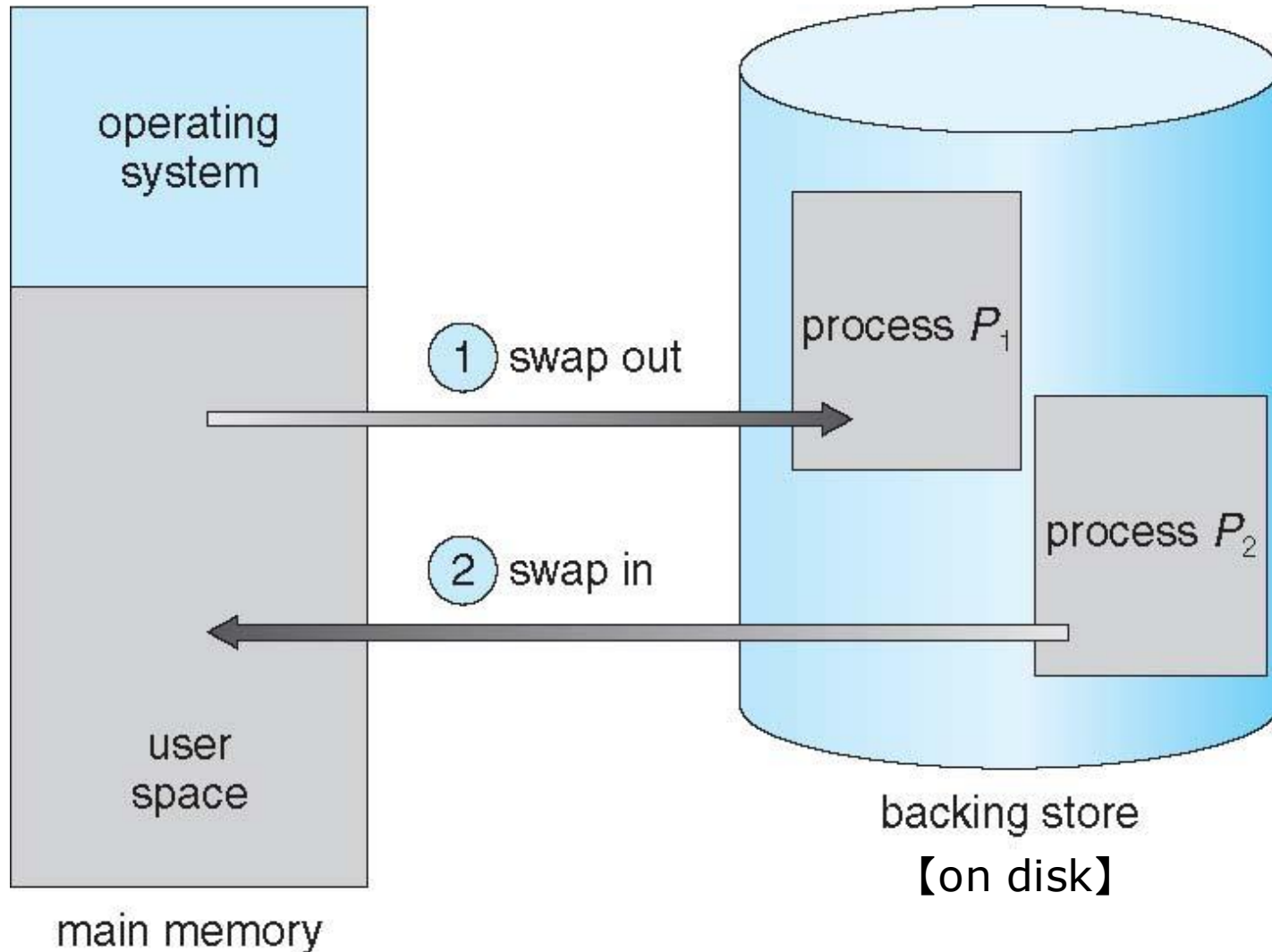


Swapping

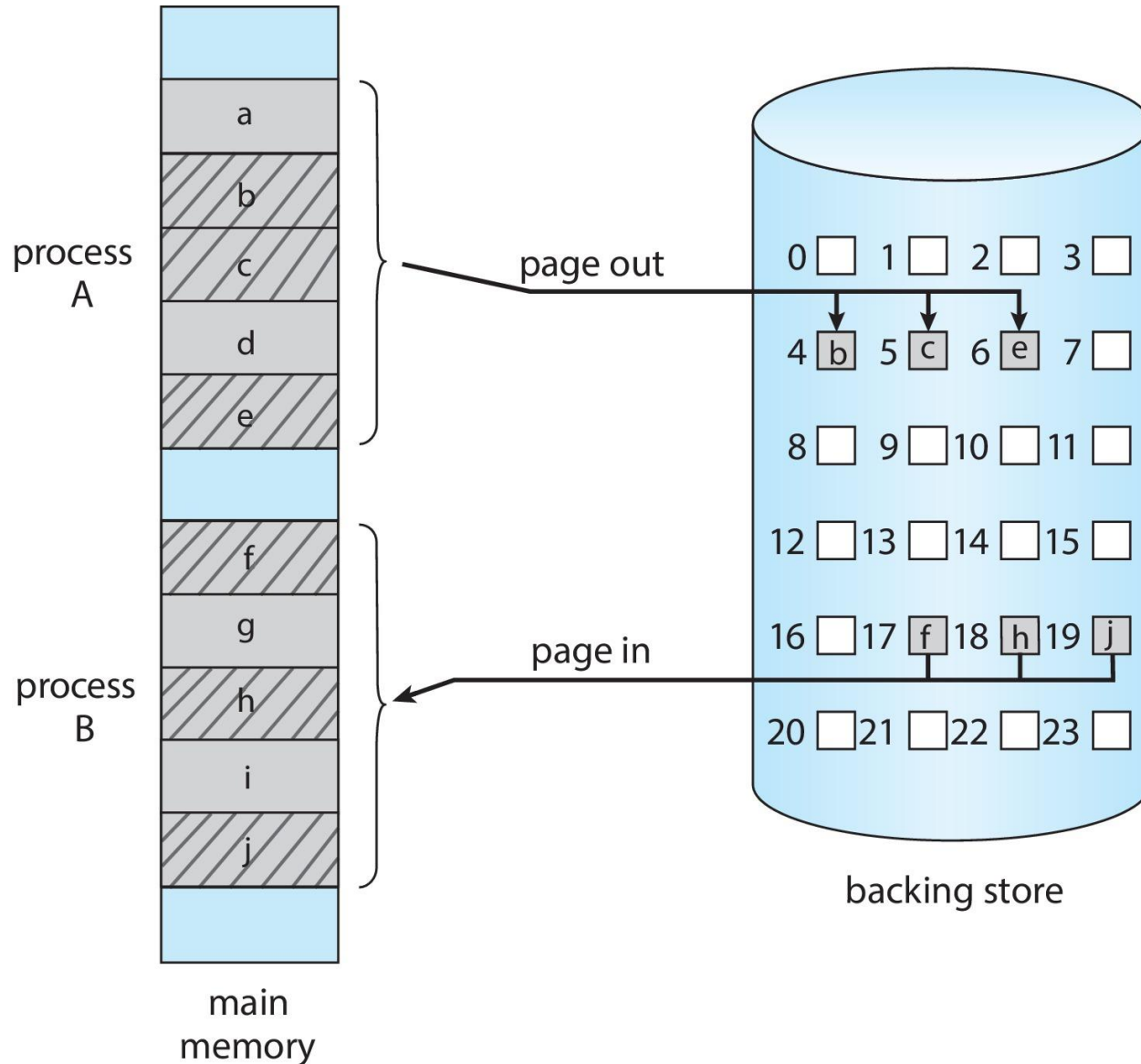
Swapping

- ❑ A process can be **swapped** temporarily out of memory to a **backing store**, and then brought back into memory for continued execution
 - ❑ Total physical memory space of processes can exceed physical memory
- ❑ **Backing store** – use **fast disk** to save copies of all user memory images;
 - ❑ must provide direct access to these memory images
- ❑ In priority scheduling, lower-priority process is swapped out so higher-priority process can be loaded and executed
- ❑ Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- ❑ System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



Swapping with Paging



Context Switch Time including Swapping

- ❑ If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
 - ❑ Context switch time can then be very high
- ❑ 100 MB process swapping to hard disk with transfer rate of 50 MB/sec
 - ❑ Swap out time of 2 seconds
 - ❑ Plus swap in of the equal-size process in 2 seconds
 - ❑ Total context switch swapping time is **4 seconds**
- ❑ It can save time if reducing the size of memory swapped –
 - ❑ Know how much memory is really being used
 - ❑ System calls to inform OS of memory use via `request_memory()` and `release_memory()`

Summary

- ❑ Memory consists of a large array of bytes, each with its own address
- ❑ Process address space can be allocated with **base** and **limit registers**.
- ❑ CPU generates a logical address, which the MMU translates to a physical address in memory.
- ❑ Contiguous memory allocation: (1) first fit, (2) best fit, and (3) worst fit.
- ❑ **Paging**: Physical memory is divided into fixed-sized blocks called **frames**, and logical memory is divided into blocks of the same size called **pages**.
 - ❑ Page number + page offset
- ❑ TLB is a hardware cache of the page table.
- ❑ Hierarchical paging involves dividing a logical address into multiple parts, each referring to different levels of page tables.
 - ❑ Avoid a too-large hierarchy: hashed page tables and inverted page tables.
- ❑ Swapping allows the system to move process pages to disk to increase the degree of multiprogramming.

Homework

- Reading
 - Chapter 9