



# 操作系统原理及应用

---

李 伟

xchlw@seu.edu.cn

计算机科学与工程学院、软件学院  
江苏省网络与信息安全重点实验室



# **Chapter 8 Memory Management**

---



# Outline

---

- **Background**
- **Contiguous Allocation**
- **Paging**
- **Structure of Page Table**
- **Segmentation**



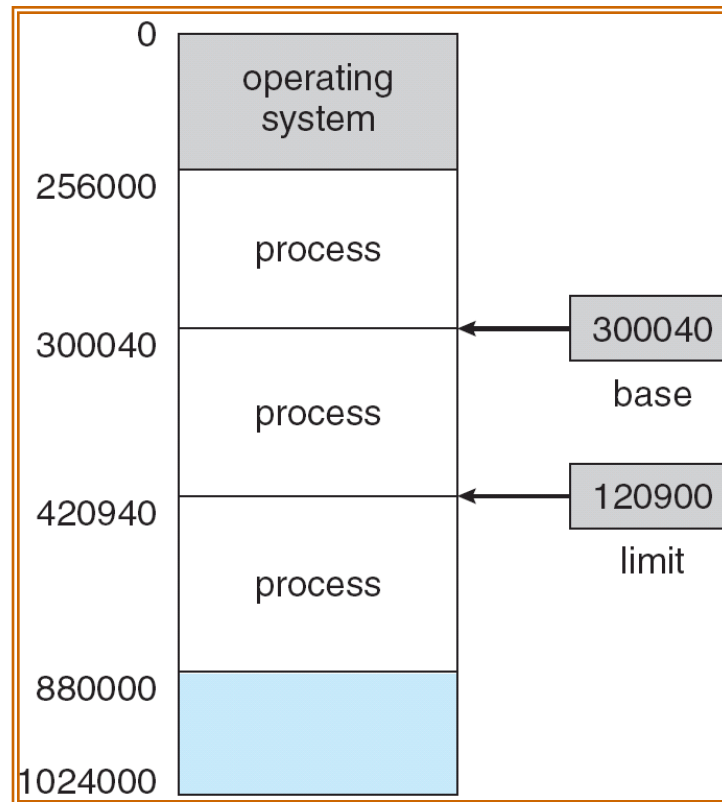
# Background

---

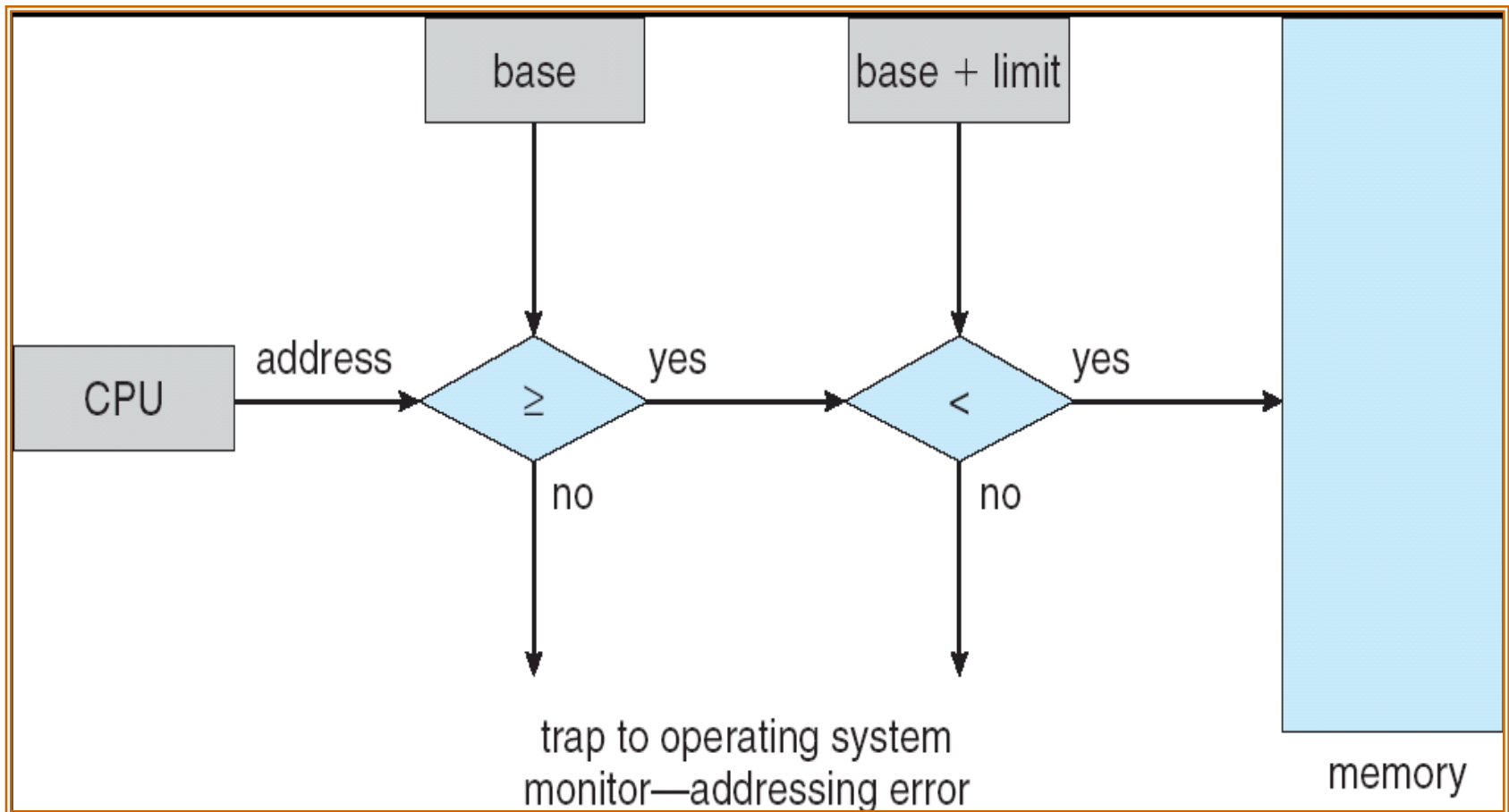
- Program must be brought into memory and placed within a process for it to be run.
- **Main memory** and **registers** are only storage CPU can access directly
  - Register access in one CPU clock (or less)
  - Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

# Base and Limit Registers

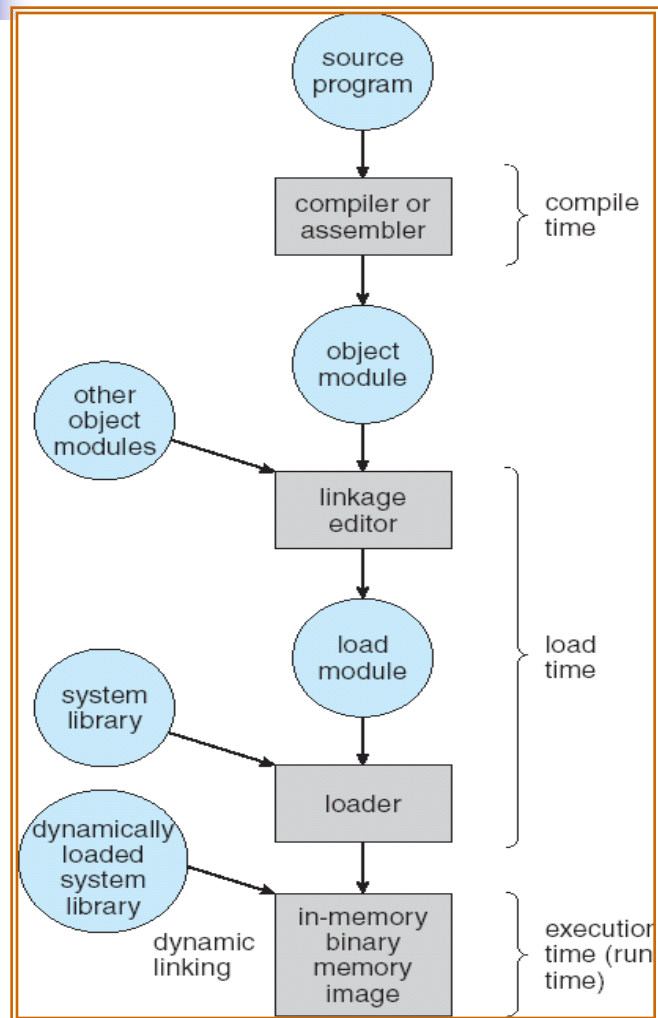
- A pair of **base** and **limit** registers define the legal address space



# Hardware Address Protection

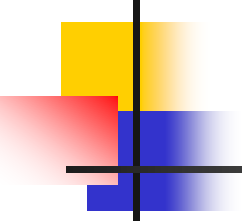


# Multistep Processing of a User Program



## ■ Representation of Addresses

- **Symbolic address (Source Program)**
- **Relocatable address (Object Program)**
- **Absolute address**

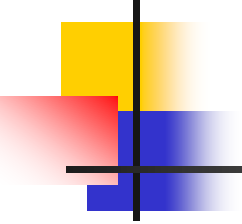


# Binding of Instructions and Data to Memory Addresses

---

- **Compile time:** If memory location known a priori, *absolute code* can be generated; code must be recompiled if starting location changes.
- **Load time:** Must generate *relocatable code* and binding delayed until load time if memory location is not known at compile time.
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps





# Logical vs. Physical Address Space

---

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to proper memory management.
  - *Logical address* – generated by the CPU; also referred to 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 *differ* in execution-time address-binding scheme.

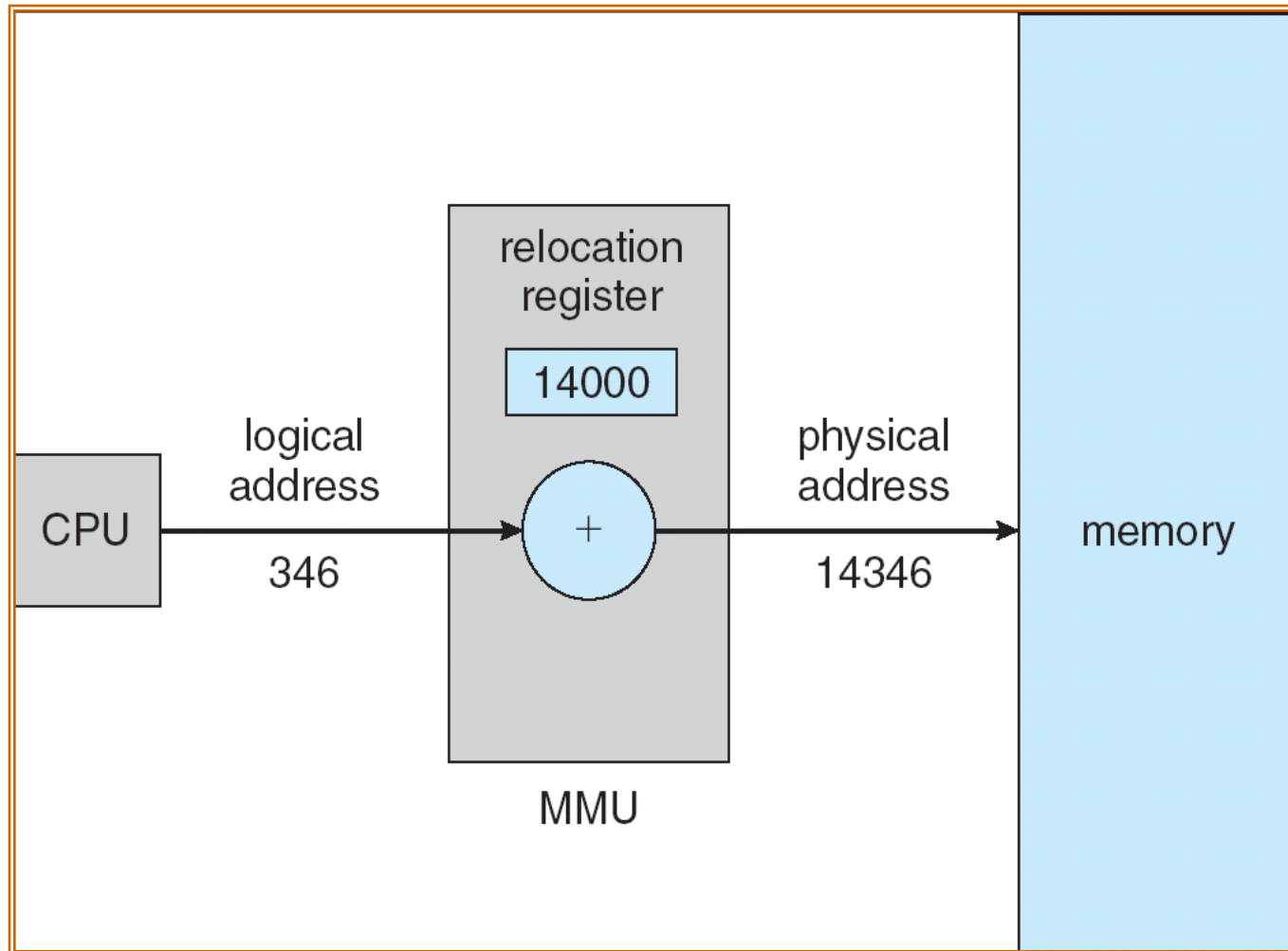


# Memory-Management Unit (MMU)

---

- Hardware device that maps logical address to physical address is the memory-management unit (MMU).
- In MMU scheme, the value in **the relocation (base) register** is added to every address generated by a user process at the time it is sent to memory.
- The user program deals with *logical* addresses; it never sees the *real physical* addresses.

# Dynamic relocation using a relocation register





# Dynamic Loading

---

- The size of a process is limited to the size of physical memory.
- Routine is not loaded until it is called
- To use **dynamic loading**, all routines must be in a *relocatable* format.
- The main program is loaded and executes.
- When a routine **A** calls **B**, **A** checks to see if **B** is loaded. If **B** is not loaded, the *relocatable linking loader* is called to load **B** and updates the address table. Then, control is passed to **B**.



# Dynamic Loading

---

- **Better memory-space utilization; unused routine is never loaded.**
- **Useful when large amounts of code are needed to handle infrequently occurring cases.**



# Dynamic Linking

---

- Both **Linking** and **loading** postponed until execution time.
- A ***stub*** is added to each reference of library routine. A stub is a small piece of code that indicates how to locate and load the routine if it is not loaded.
- When a routine is called, its stub is executed. The routine is loaded, the address of that routine replaces the stub, and executes the routine.



# 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** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.



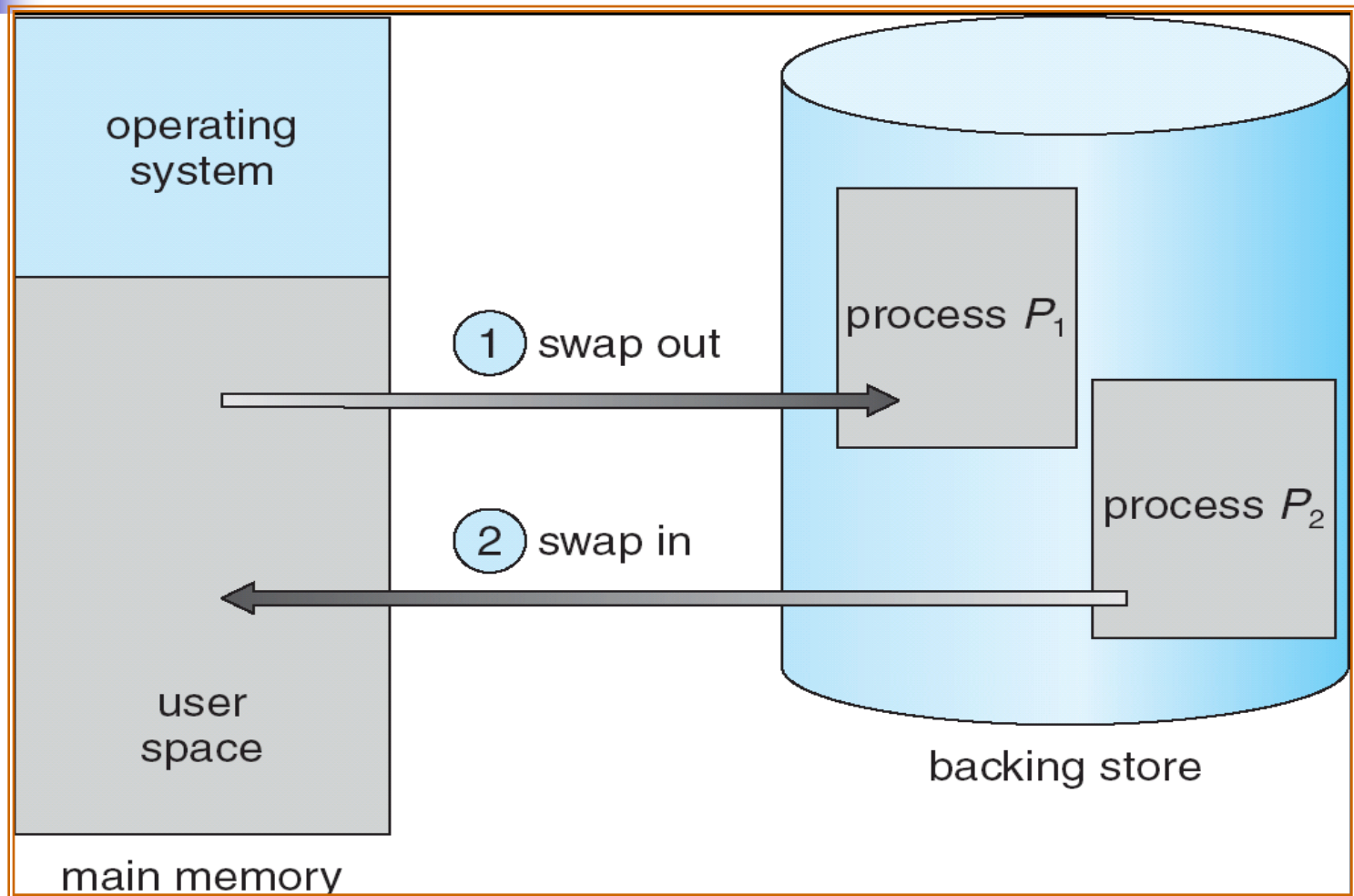
# Swapping

---

- **Ready queue** is consisting of all processes *on the backing store or in memory* and are ready to run
- Major part of swap time is transfer time; total transfer time is directly proportional to the *amount* of memory swapped.



# Schematic View of Swapping





# Outline

---

- Background
- **Contiguous Allocation**
- Paging
- Structure of Page Table
- Segmentation

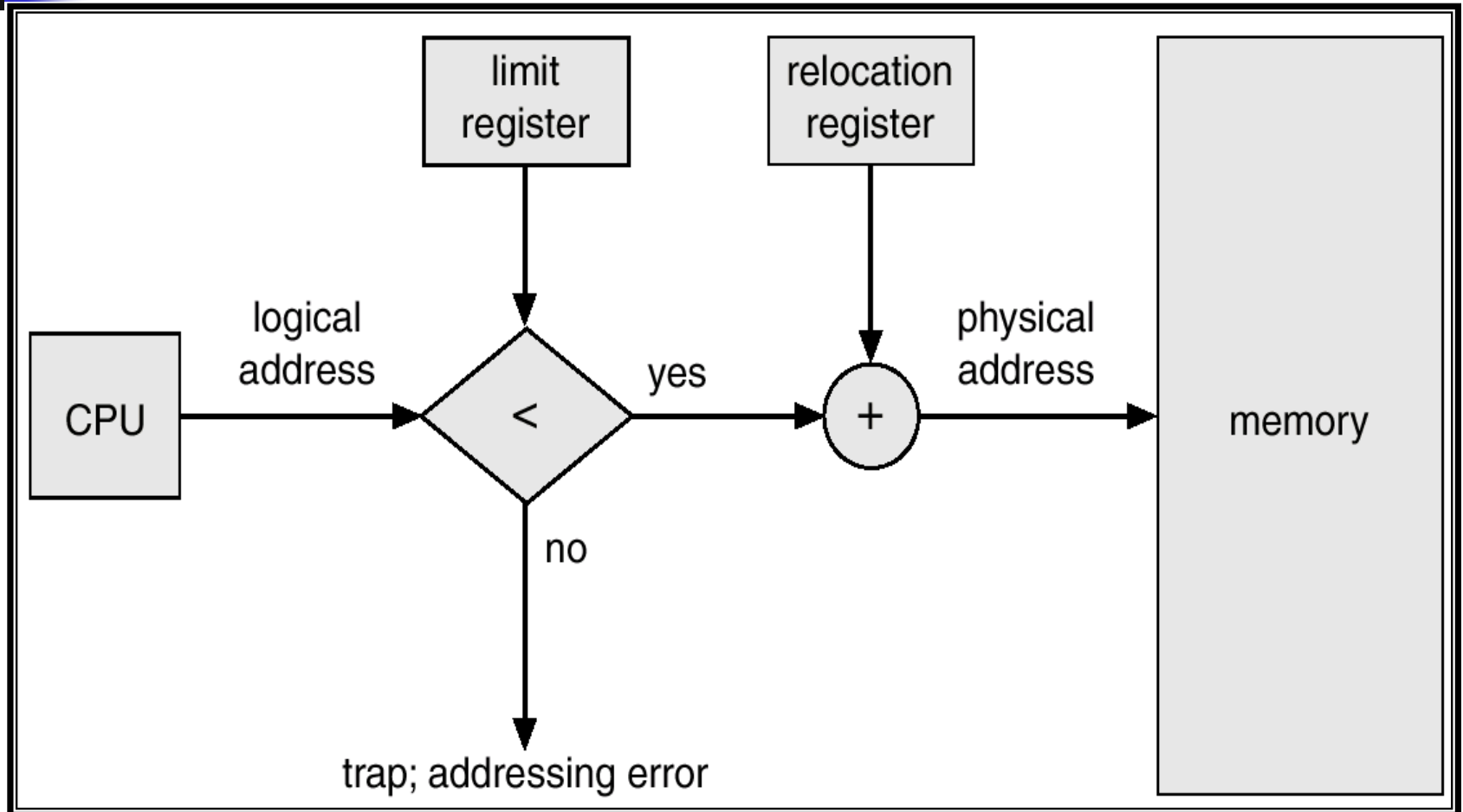


# Contiguous Allocation

---

- The main memory is usually divided into two partitions
  - **One for the resident operating system**, usually held in low memory with interrupt vector.
  - **One for user processes** then held in high memory.
- **Relocation register scheme** used to protect user processes from each other, and from changing operating-system code and data.

# Hardware Support for Relocation and Limit Registers





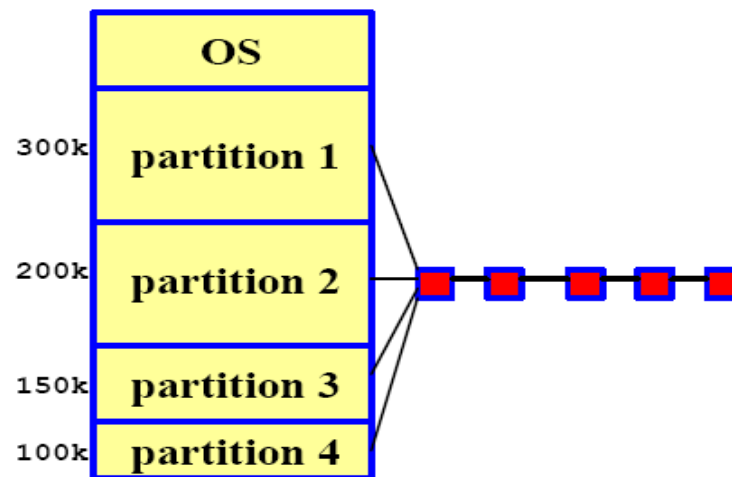
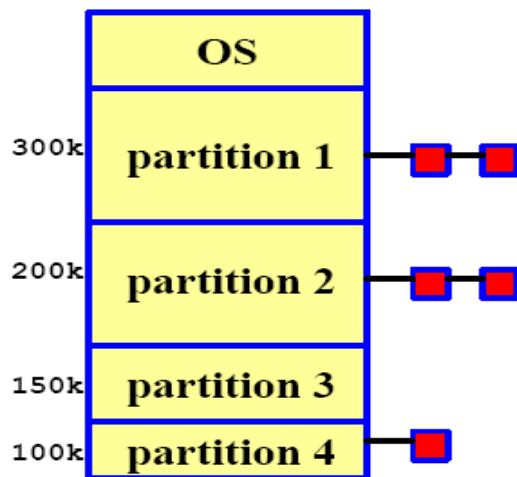
# Contiguous Allocation

---

- **Fixed partitions**
- **Variable partitions**

# Fixed partitions

- Memory is divided into  $n$  partitions at the startup time and altered later on.
- Each partition may contain exactly one process.
- Each partition may have a job queue. Or, all partitions share the same job queue.





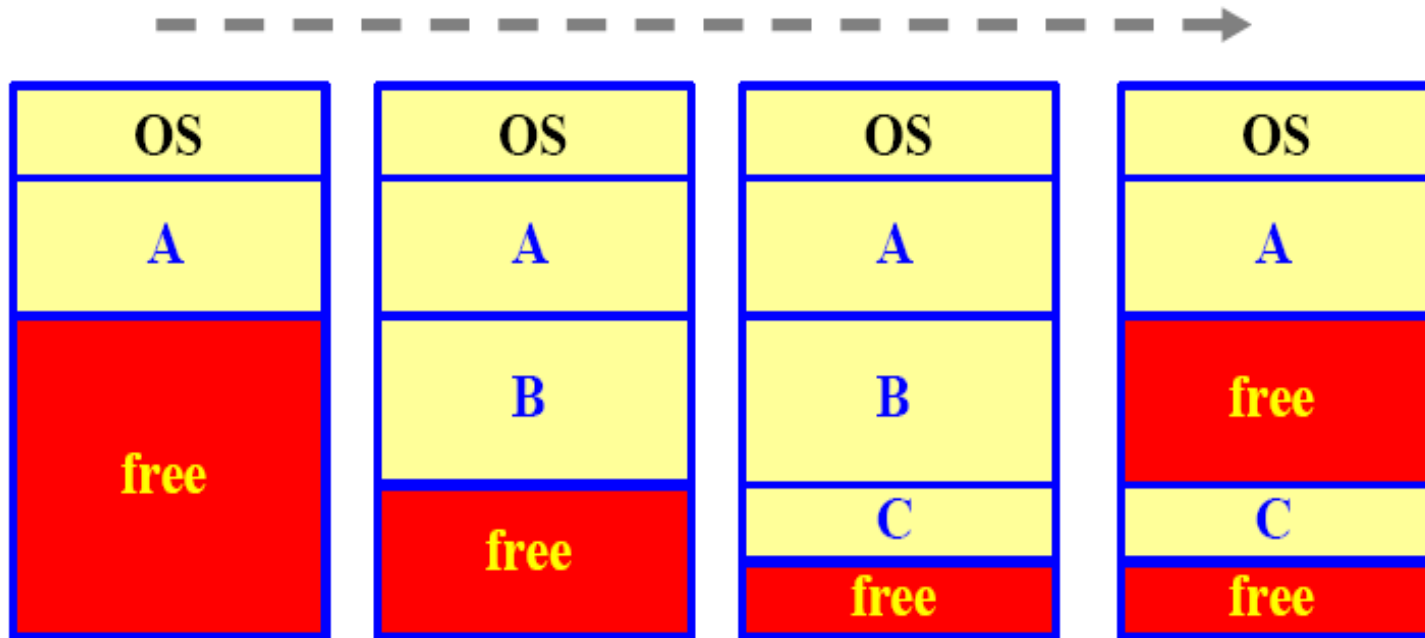
# Variable Partitions

---

- **Hole ( $\mathcal{H}$ )** – **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.
- Thus, partition sizes are not fixed, The number of partitions also varies.
- Operating system maintains information about
  - allocated partitions
  - free partitions (hole)

# Variable Partitions

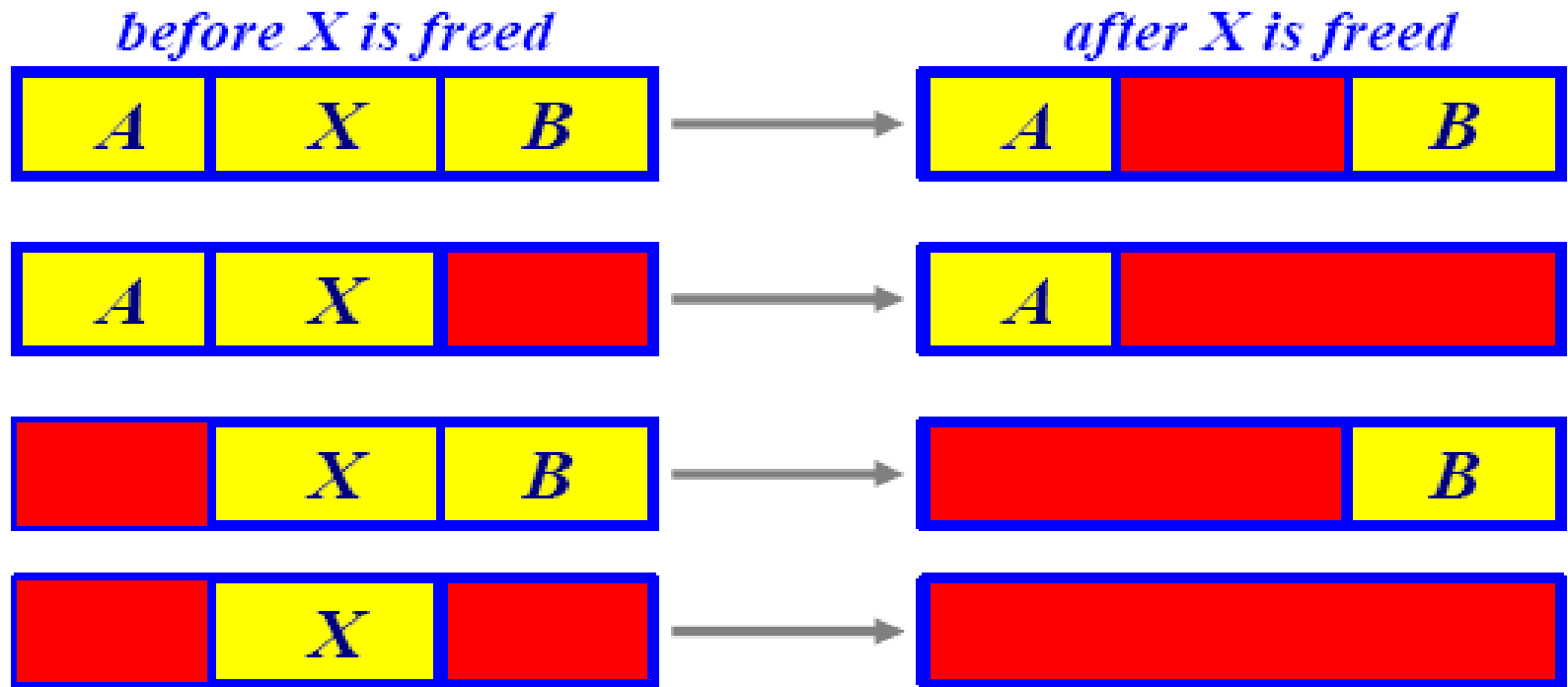
- If the hole is larger than the requested size, it is cut into two. The one of the requested size is given to the process, the remaining one becomes a *new* hole.





# Variable Partitions

- When a process returns a memory block, it becomes a hole and must be combined with its neighbors.





# Solutions to Dynamic Storage-Allocation Problem

---

- **First-fit(首次适配):** Allocate the *first* hole that is big enough.
- **Best-fit(最佳适配):** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size. Produces the smallest leftover hole.
- **Worst-fit(最差适配):** Allocate the *largest* hole; must also search entire list. Produces the largest leftover hole.



# Solutions to Dynamic Storage-Allocation Problem

---

- Simulations have shown that
  - both **first fit** and **best fit** are better than **worst fit** in terms of **decreasing time and storage utilization**
  - Neither **first fit** nor **best fit** is clearly better than the other in terms of **storage utilization**, but **first fit** is generally faster.



# Exercise1

---

- 某基于动态分区存储管理的计算机，其主存容量为55MB（初始为空闲），采用最佳适配算法，分配和释放的顺序为分配15MB、分配30MB、释放15MB、分配8MB、分配6MB，此时主存中最大空闲分区的大小是多少？

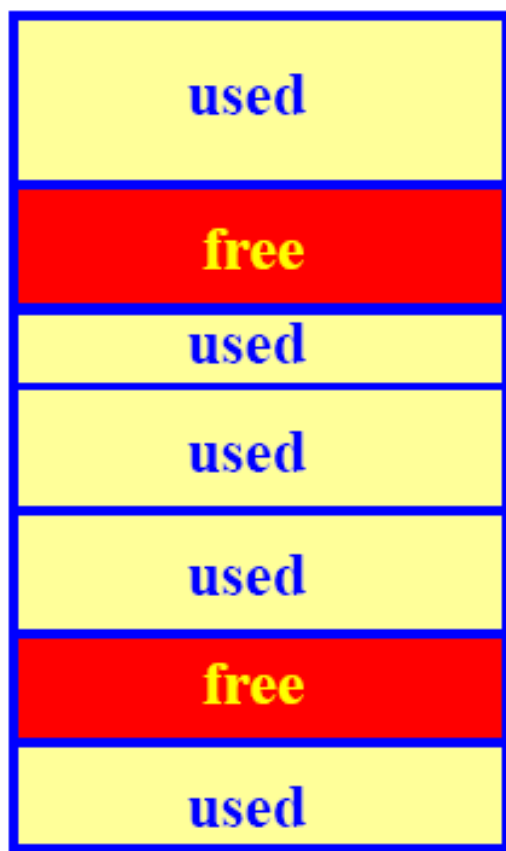


# Fragmentation(碎片)

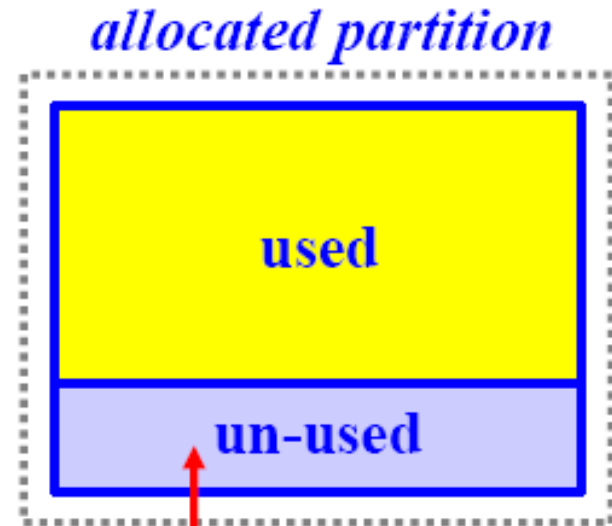
---

- Processes are loaded and removed from memory, eventually the memory will be cut into small holes that are not large enough to run any incoming process.
- Free memory holes between allocated ones are called ***external fragmentation***.
- It is unwise to allocate exactly the requested amount of memory to a process, **because of the minimum requirement for memory management**.
- Thus, memory that is allocated to a partition, but is not used, are called ***internal fragmentation***.

# Fragmentation



*external  
fragmentation*



*internal fragmentation*



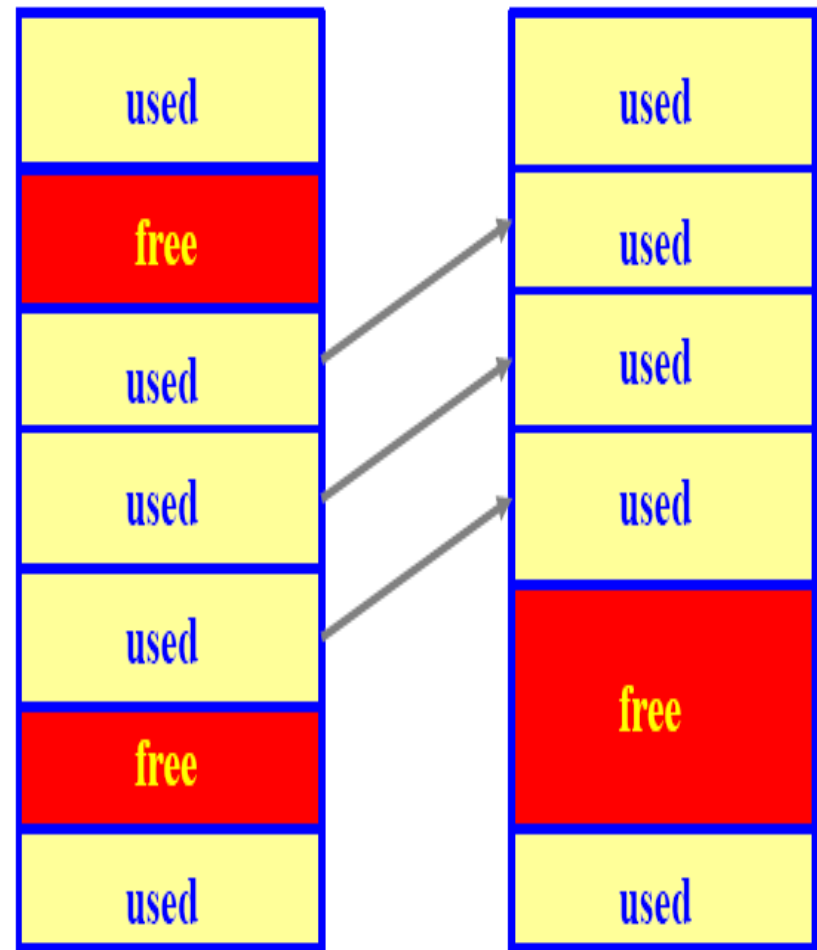
# External Fragmentation Problem

---

- **External fragmentation problem** exists when there is enough total free memory space to satisfy a request, but the available spaces are not contiguous.
- **Solutions**
  - **Compaction**
  - Permitting the physical address space of the processes to **be noncontiguous** — Paging and Segmentation

# Compaction for External Fragmentation

- 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.
- compaction scheme can be expensive







# Outline

---

- Background
- Contiguous Allocation
- **Paging**
- Structure of Page Table
- Segmentation



# Paging

---

- Physical address space of a process can be **noncontiguous**; process is allocated physical memory whenever the latter is available.
- Divide physical memory into fixed-sized blocks called **frames(帧)** (size is power of 2, between 512 bytes and 16M bytes).
- Divide logical memory into blocks of same size called **pages(页)**.



# Paging

---

- Keep track of all free frames.
- To run a program of size  $n$  pages, need to find  $n$  free frames and load program.
- Set up a **page table** to translate logical to physical addresses.
- Still have Internal Fragmentation.



# 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.



# Address Translation Scheme

---

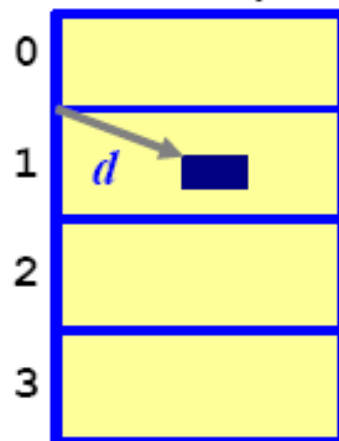
- For given logical address space  $2^m$  and page size  $2^n$

page number	page offset
$p$	$d$
$m - n$	$n$



$p$        $d$   
page #      offset within the page

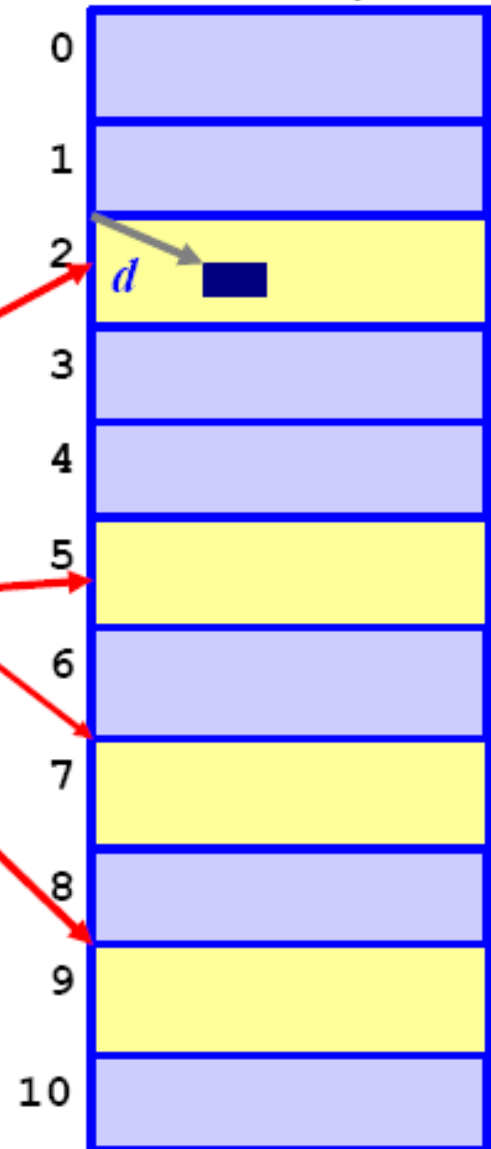
logical  
memory



page table

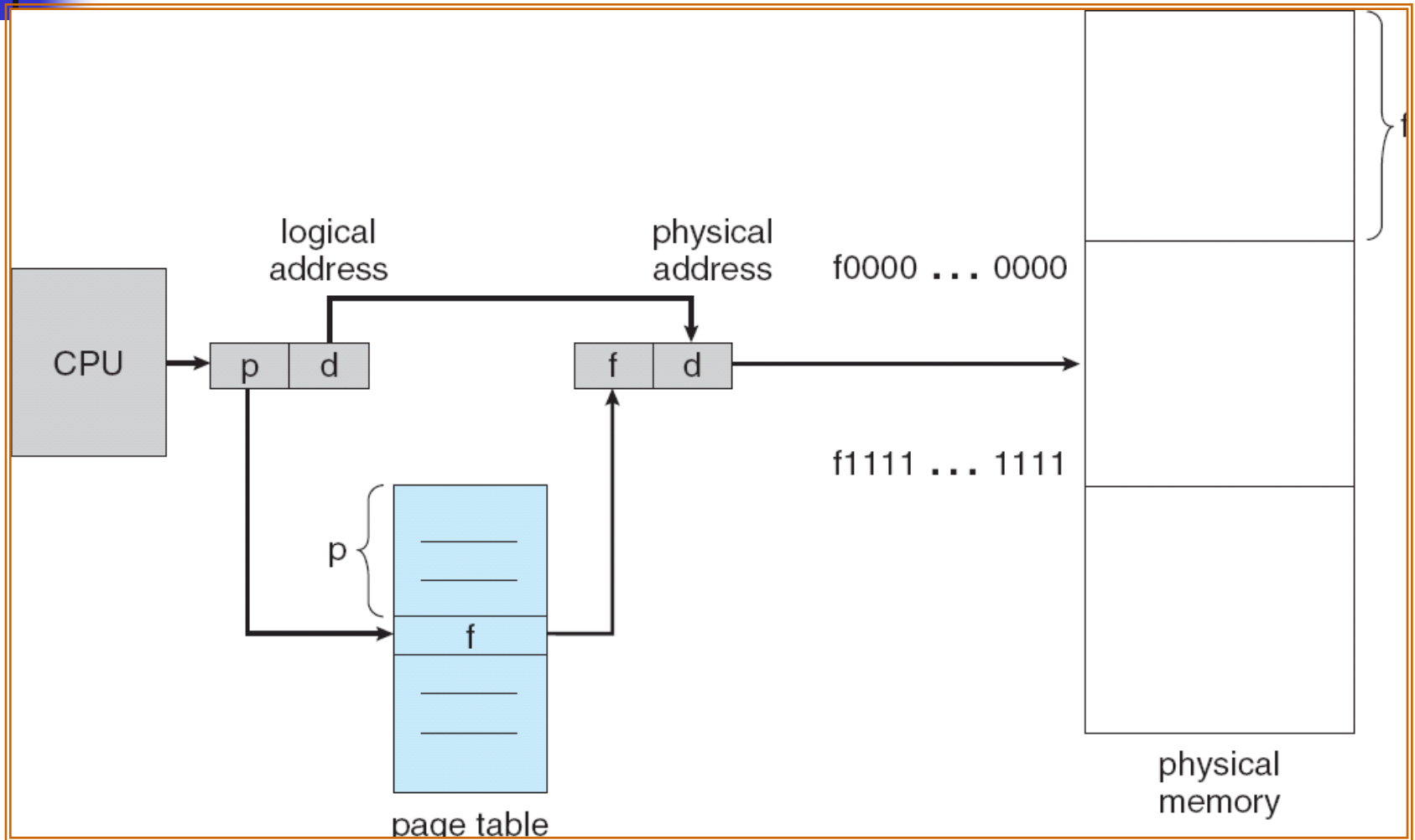
0	7
1	2
2	9
3	5

memory

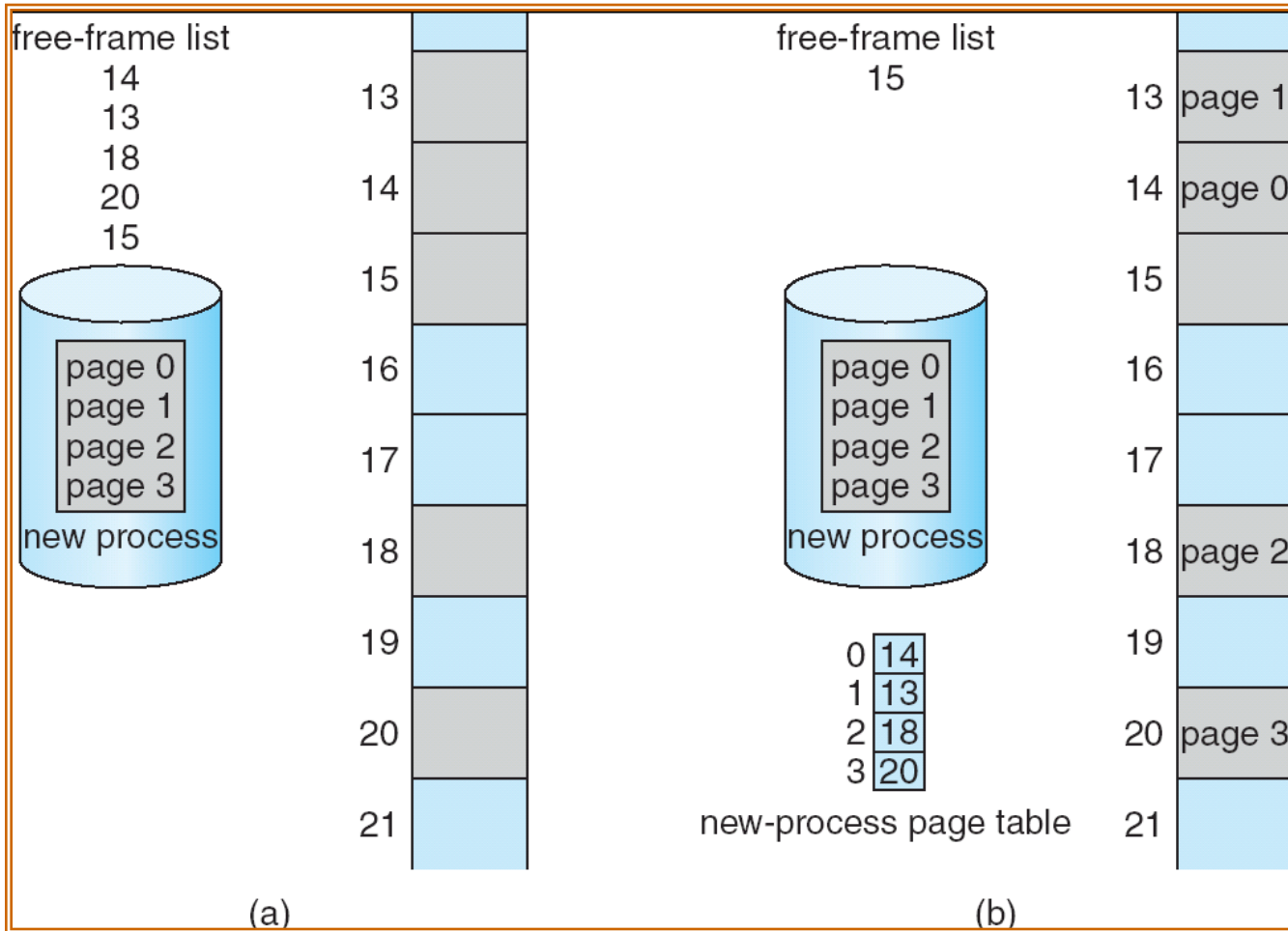


logical address  $\langle 1, d \rangle$  translates to  
physical address  $\langle 2, d \rangle$

# Paging Hardware Support



# Paging Example







## Exercise2

---

- 如果帧大小为4KB，那么具有4B大小页表条目的系统可以访问的最大物理内存空间为多少？

*如果固定页表大小为一页呢？*



# Implementation of Page Table

---

- Most allocate a page table for each process.
- In the simplest case, the page table is implemented as **a set of dedicated registers**.
- The use of registers is fit for **the small page tables** (for example, 256 entries) and not is fit for the large page tables (for example, 1 million entries) .



# Implementation of Page Table

---

- Page table is kept in main memory.
- *Page-table base register (PTBR)* points to the page table.
- In this scheme every data/instruction access requires **two memory accesses**. One for the page table and one for the data/instruction. Memory access is slowed by a factor of 2.



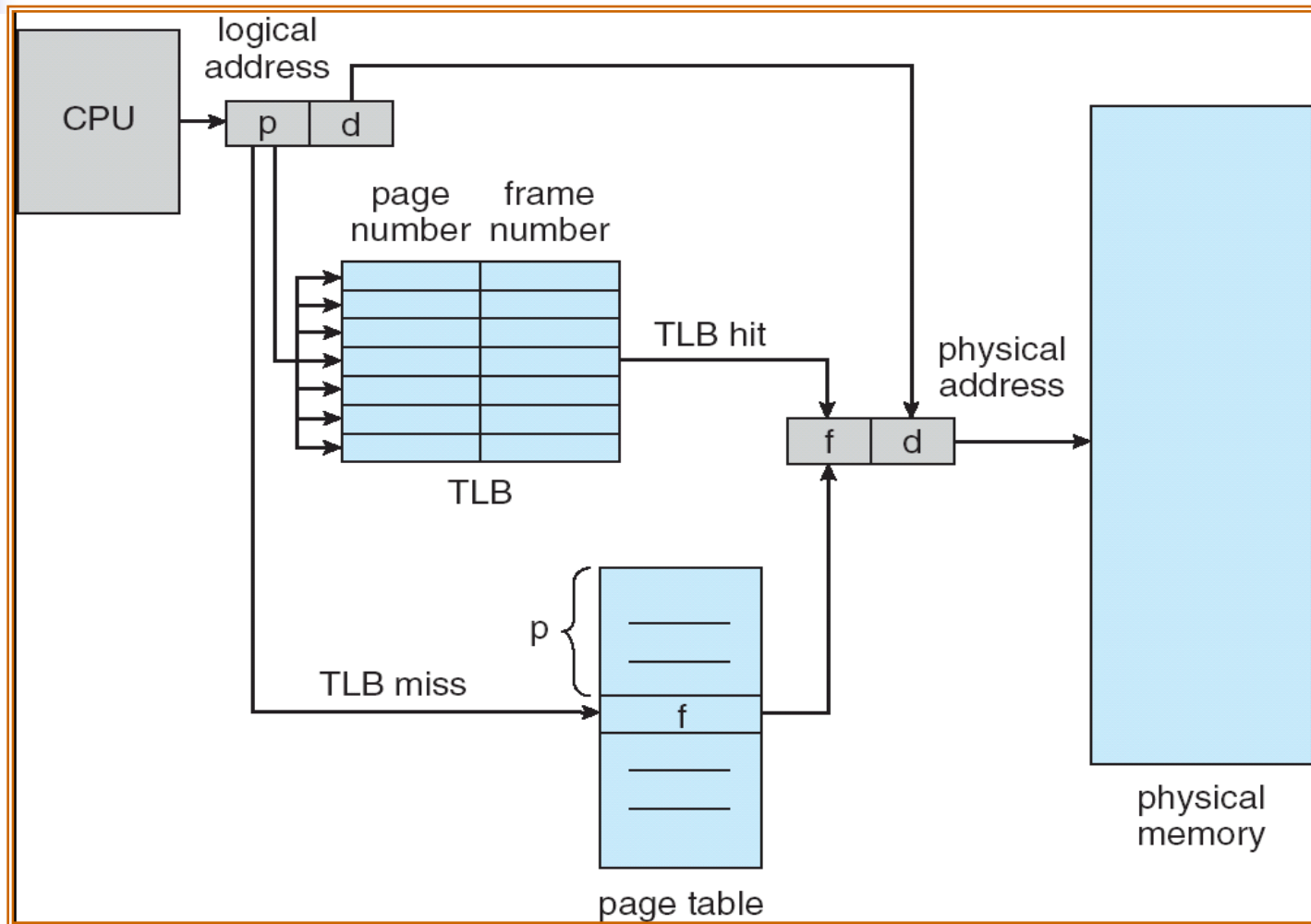
# Implementation of Page Table

- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called *associative memory* or *translation look-aside buffer (TLB)*

Page #	Frame #

If page # is in associative register, get frame # out.  
Otherwise get frame # from page table in memory

# Paging Hardware With TLB





# Implementation of Page Table

---

- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process.
- ASID allows the TLB to contain entries for several different processes simultaneously. Otherwise it need to be flushed at every context switch.



# Effective Access Time

---

- **Hit ratio (命中率)** – percentage of times that a page number is found in TLB.
- Hit ratio =  $\alpha$
- TLB Lookup =  $\varepsilon$  time unit
- Assume memory cycle time is 1 microsecond
- Effective Access Time (EAT)

$$\begin{aligned} \text{EAT} &= (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\ &= 2 + \varepsilon - \alpha \end{aligned}$$



## Exercise3

---

- 考虑一个分页系统，其页表存放在内存：
  - (1) 如果内存读写周期为 $2\ \mu\text{s}$ ，则CPU从内存取一条指令或一个操作数需要时间为多少？
  - (2) 如果设立一个可存放8个页表项的TLB，70%的地址变换可通过TLB完成，内存平均存取时间为多少？（假设TLB的访问时间可以忽略不计）



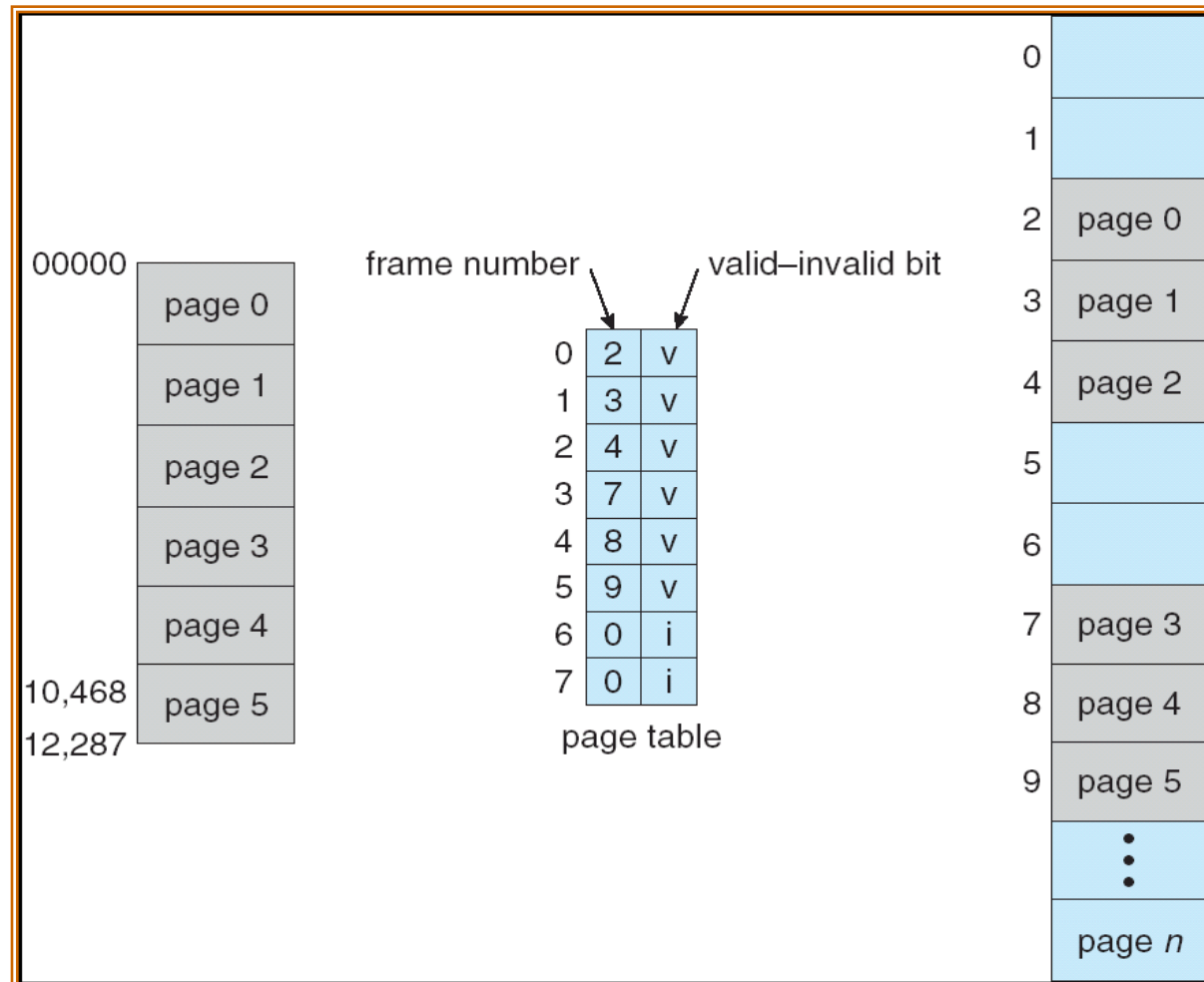


# Memory Protection

---

- Memory protection implemented by **associating protection bit** with each frame in **a paged environment**.
- ***Valid-invalid bit*** attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’s logical address space, and is thus a legal page.
  - “invalid” indicates that the page is not in the process’s logical address space.

# Valid (v) or Invalid (i) Bit In A Page Table





# Memory Protection

---

- we can use a *page table length register (PTLR)* that stores the length of a process's page table. In this way, a process cannot access the memory beyond its region.
- We can also add read-only, read-write, or execute bits in page table to enforce **r-w-e** permission.

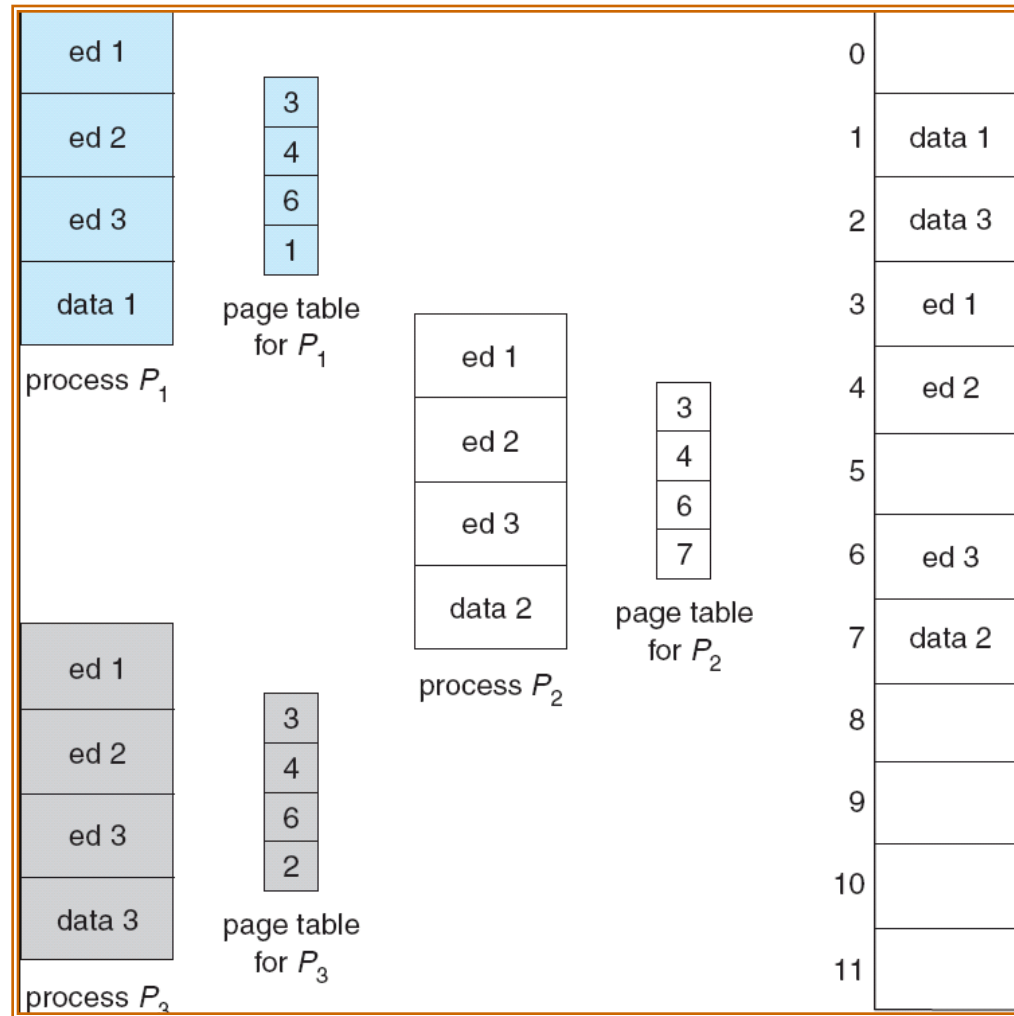


# Shared Pages

---

- An advantage of paging is the possibility of sharing common code.
- Shared code: One copy of **read-only (reentrant) code** shared among processes (i.e., text editors, compilers, window systems).
- Each process keeps a separate copy of the private code and data

# Shared Pages Example





# Outline

---

- **Background**
- **Contiguous Allocation**
- **Paging**
- **Structure of Page Table**
- **Segmentation**



# Structure of Page Table

---

- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**



# Hierarchical Paging

---

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





# Two-Level Paging Example

---

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



# Two-Level Paging Example

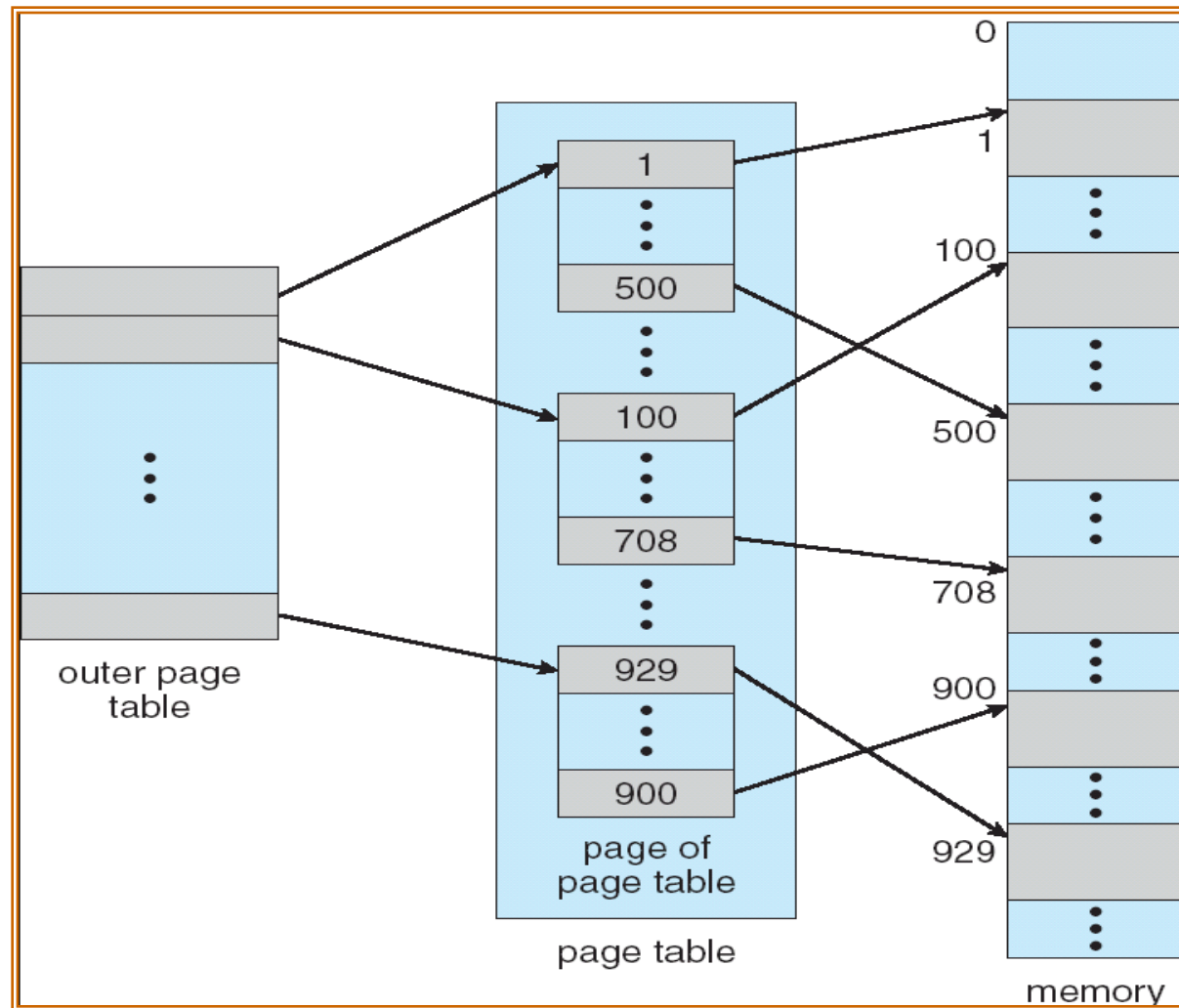
---

- Thus, a logical address is as follows:

page number		page offset
$p_1$	$p_2$	$d$
10	10	12

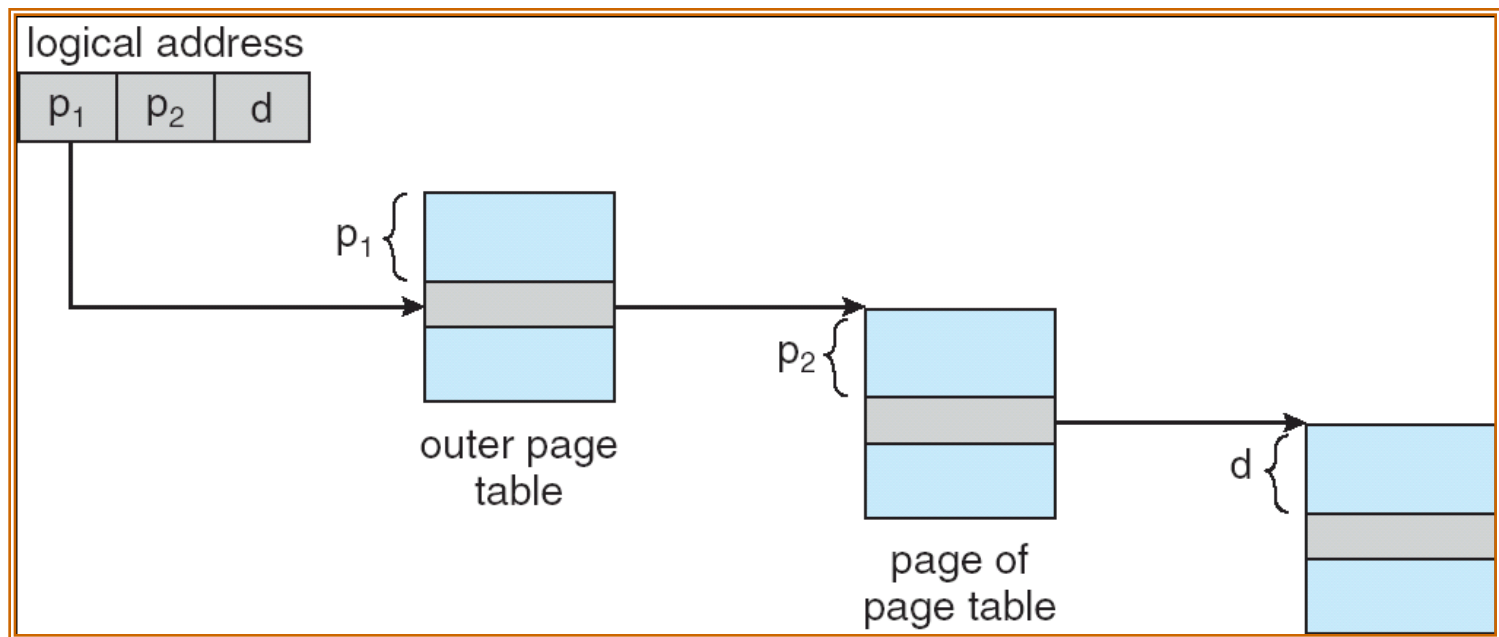
where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement within the page of the **outer page table** (页目录表) .

# Two-Level Page-Table Scheme



# Address-Translation Scheme

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





# 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



## Exercise4

- 某计算机采用二级页表的分页存储管理方式，按字节编址，页大小为 $2^{10}\text{B}$ ，页表项大小为 $2\text{B}$ ，逻辑地址结构为（页目录号，页号（页表偏移量），页内偏移量），逻辑地址空间大小为 $2^{16}$ 页，则表示整个逻辑地址空间的页目录表中包含表项的个数至少是多少？

整个逻辑地址结构中，每一部分分别需要占用多少个bit？

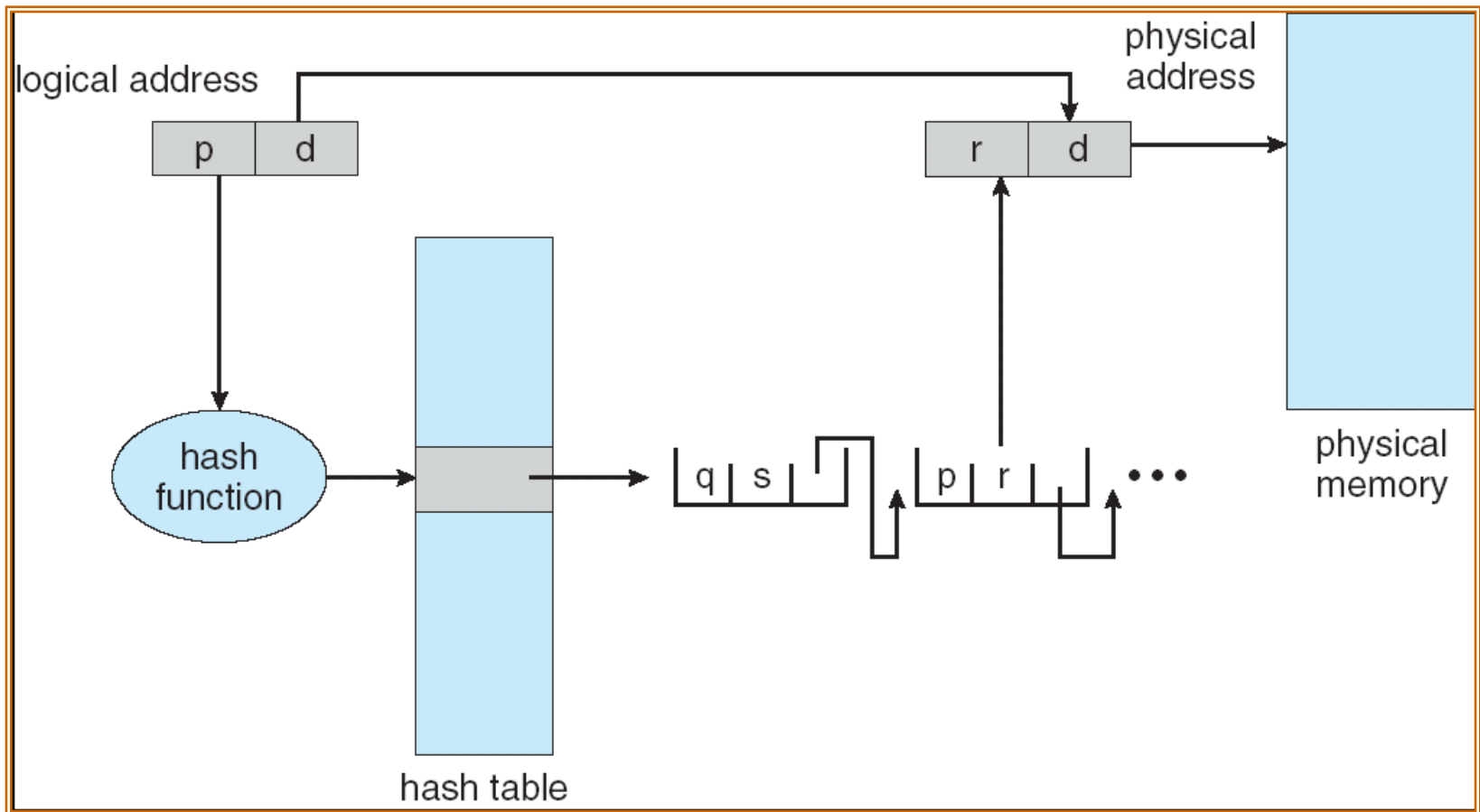


# Hashed Page Tables

---

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

# Hashed Page Table





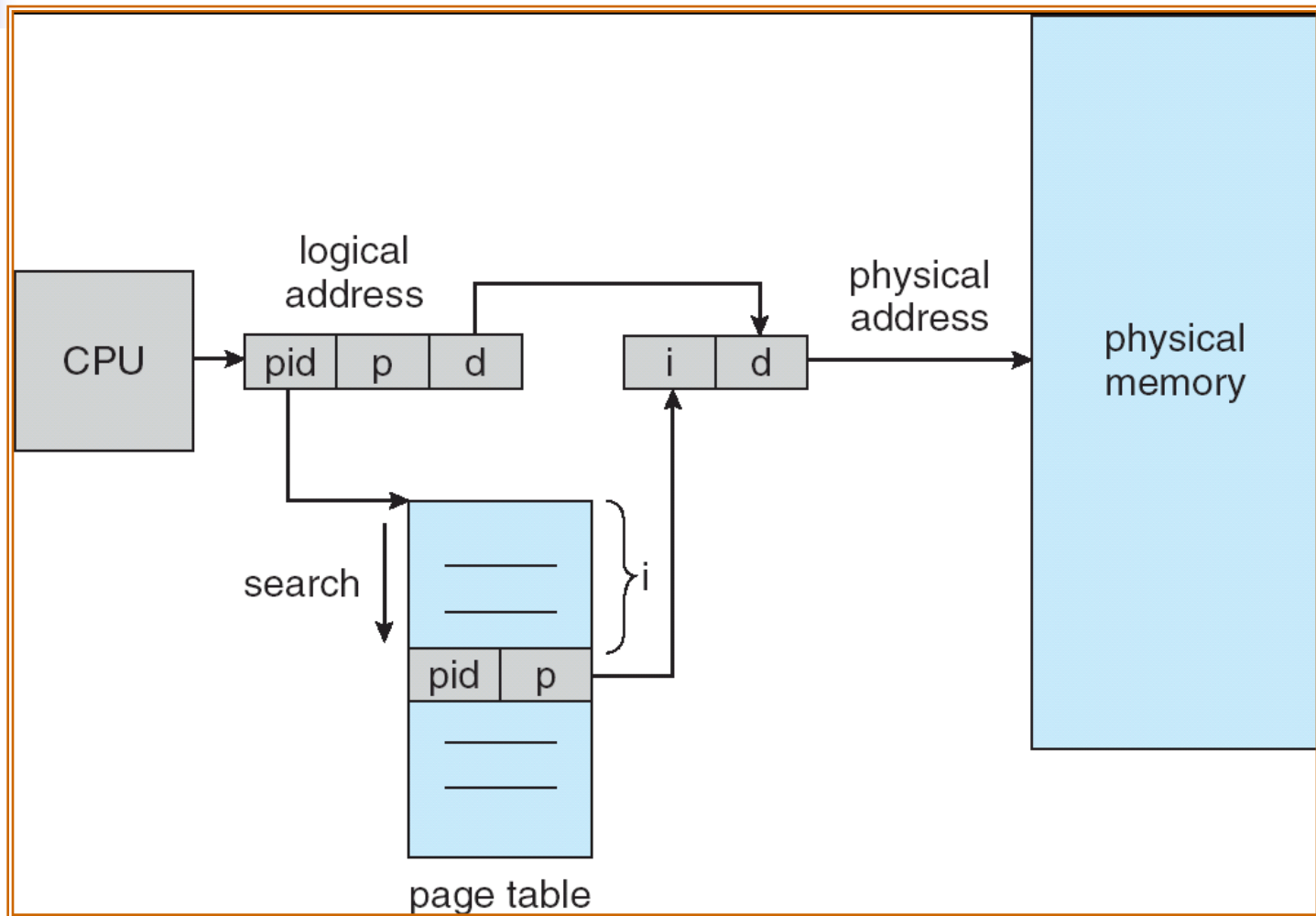


# Inverted Page Table

---

- One entry for each real frame of memory.
- Entry consists of **the virtual address of the page** stored in that real memory location, with **information about the process** that owns that page.
- **Decreases memory** needed not to store each page table, but **increases time** needed to search the table when a page reference occurs.
- Use hash table to limit the search to one or at most a few page-table entries.

# Inverted Page Table Architecture





# Outline

---

- **Background**
- **Contiguous Allocation**
- **Paging**
- **Structure of Page Table**
- **Segmentation**

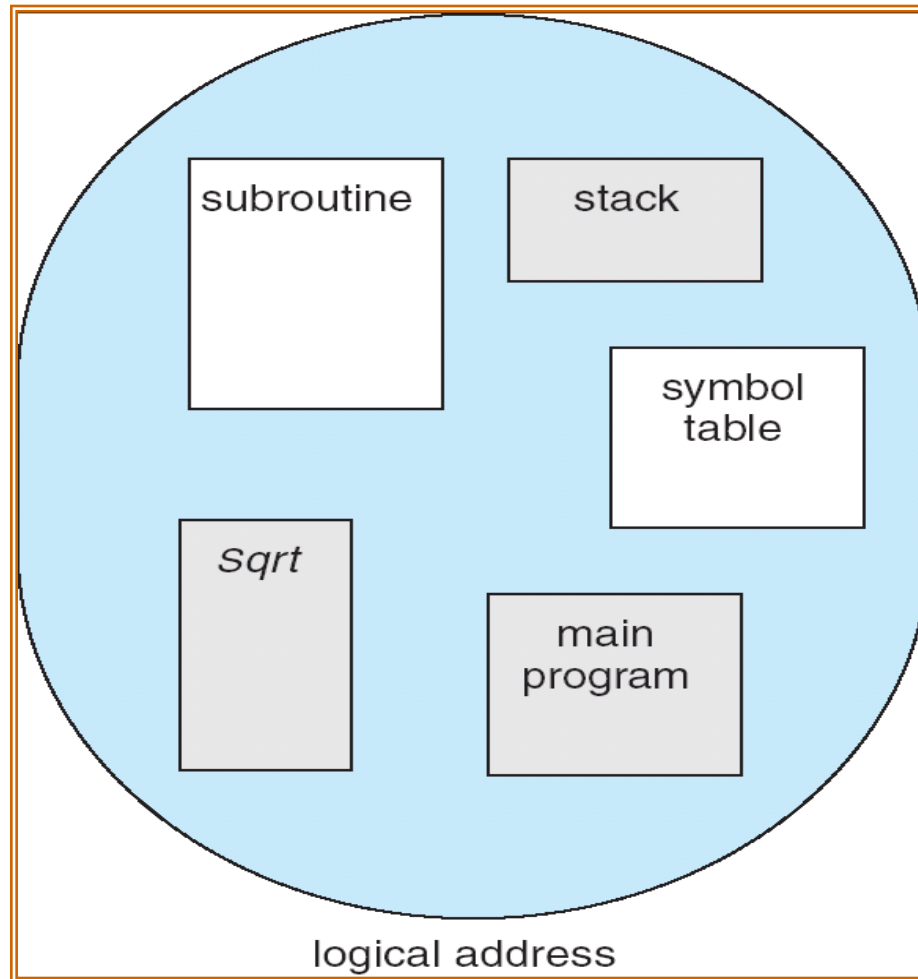


# Segmentation

---

- **Memory-management scheme that supports user view of memory.**
- **A program is a collection of segments. A segment is a logical unit such as:**
  - main program, procedure, function, method,**
  - object, local variables, global variables,**
  - common block, stack, symbol table, arrays**

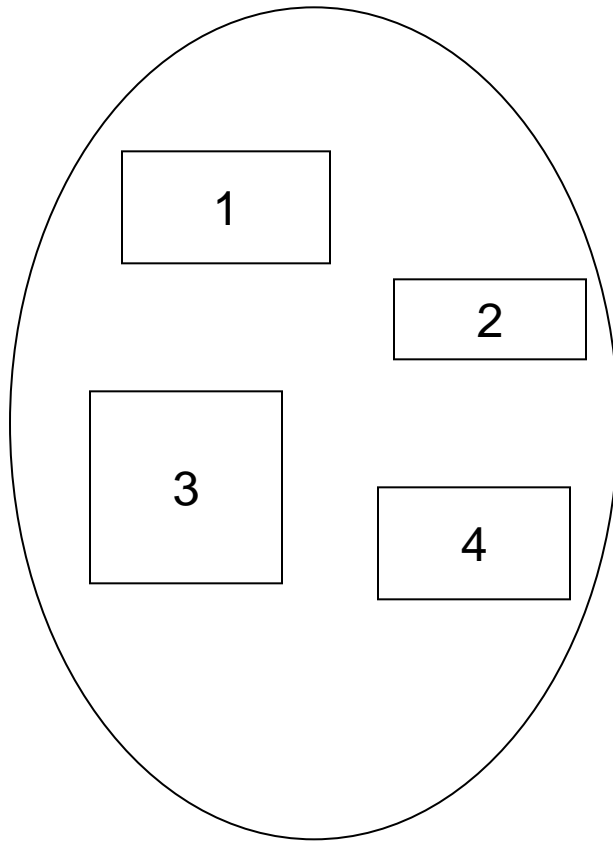
# User's View of a Program



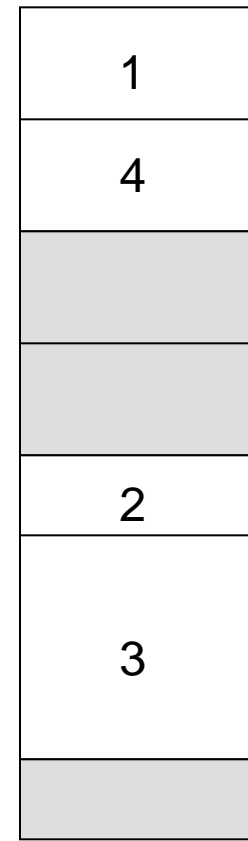


# Logical View of Segmentation

---



user space



physical memory space

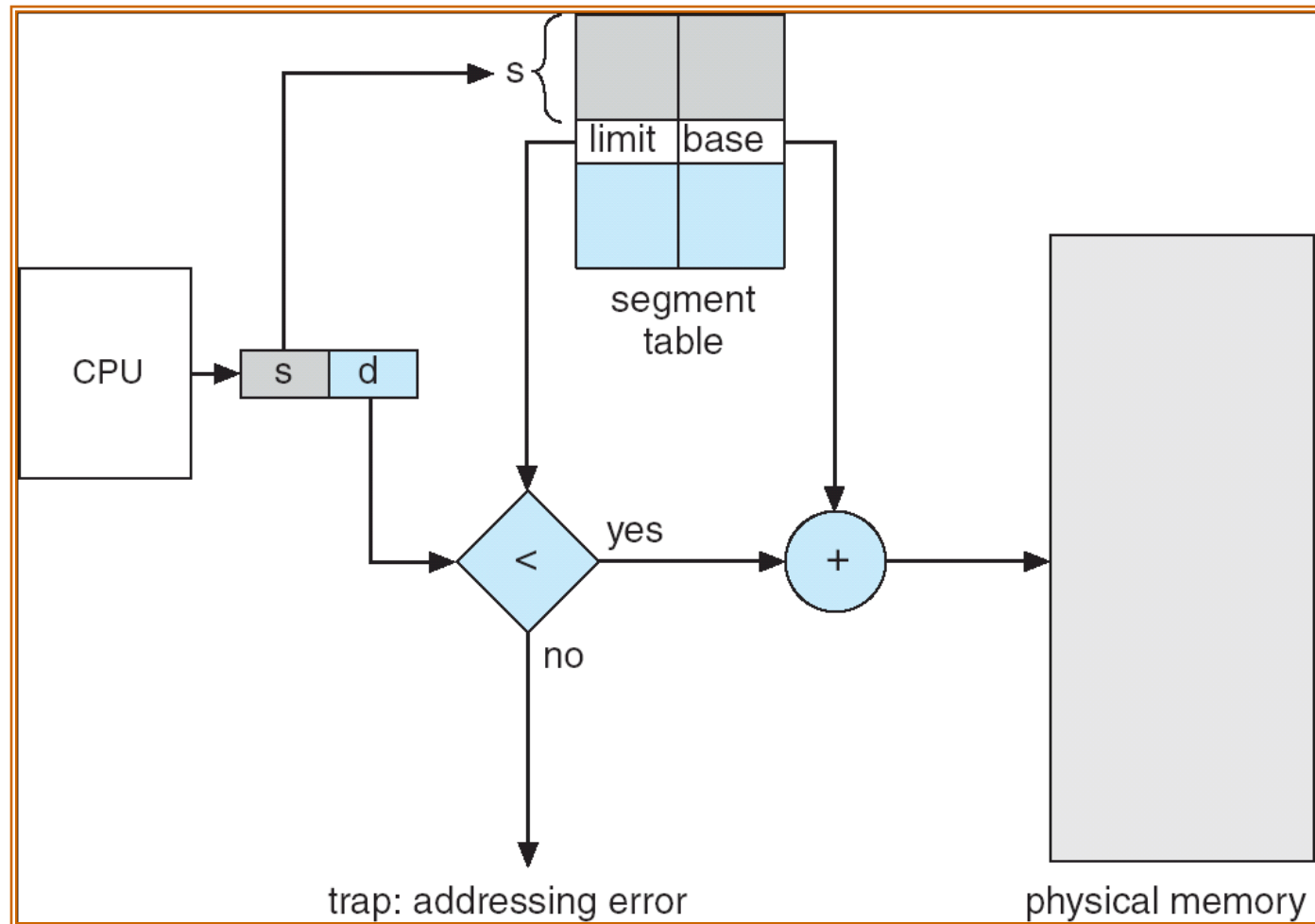


# Segmentation Architecture

---

- Logical address consists of a two tuple:  
    <segment-number, offset>,
- **Segment table** – maps two-dimensional to 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.

# Segmentation Hardware





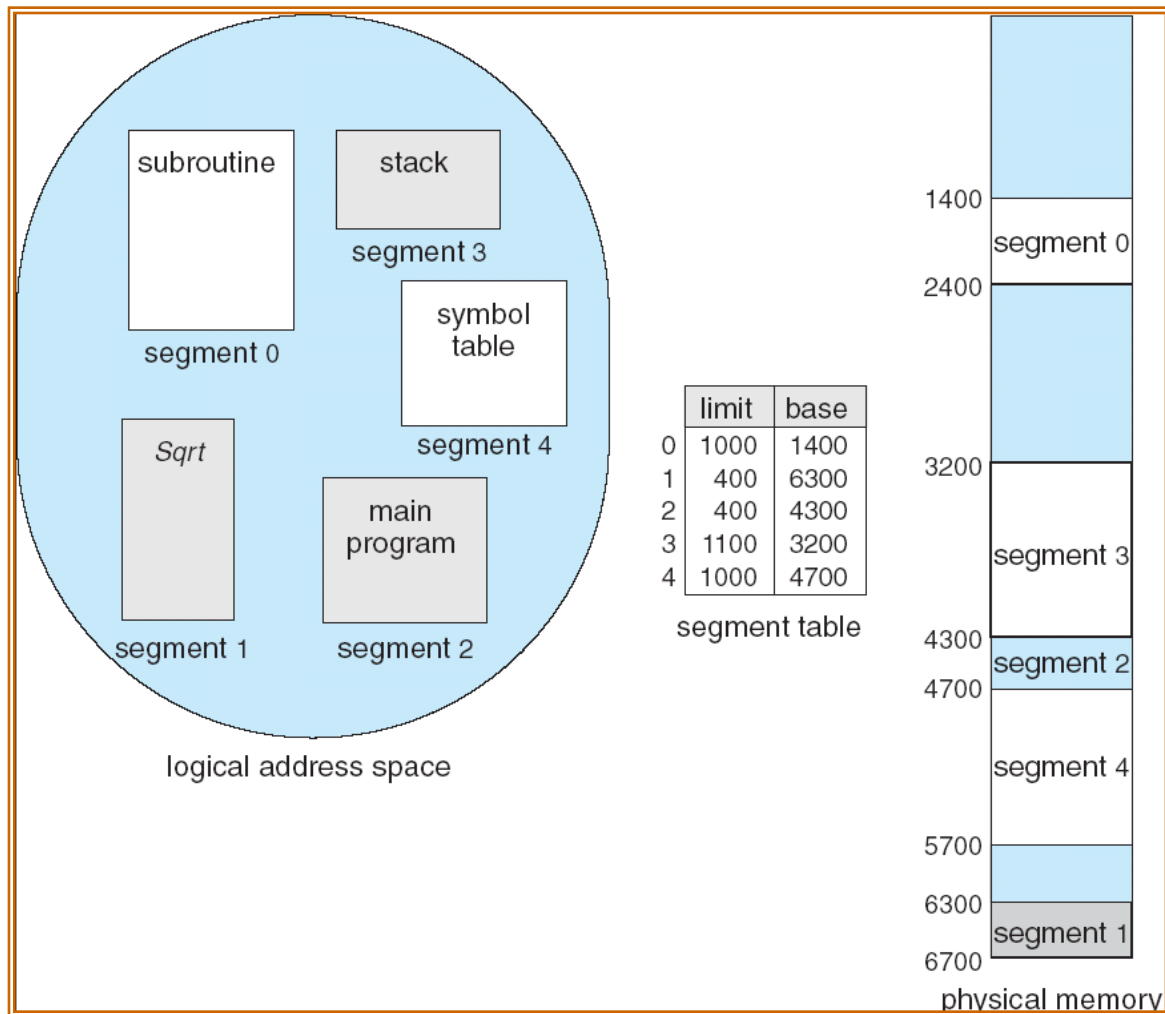


# Segmentation Architecture

---

- **Protection:** With each entry in segment table associate:
  - validation bit = 0  $\Rightarrow$  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**.
- A segmentation example is shown in the following diagram.

# Example of Segmentation



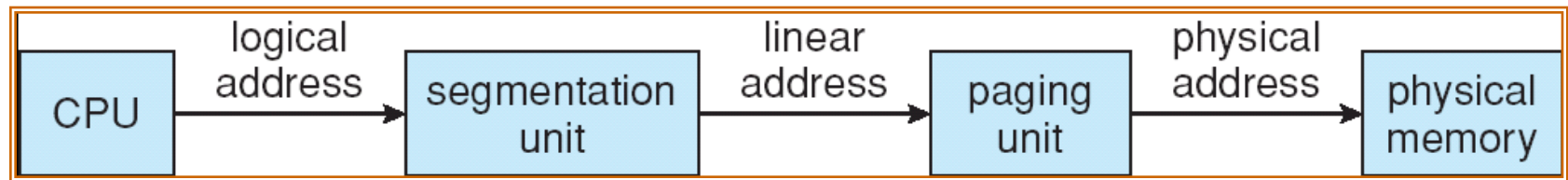


# Example: The Intel Pentium

---

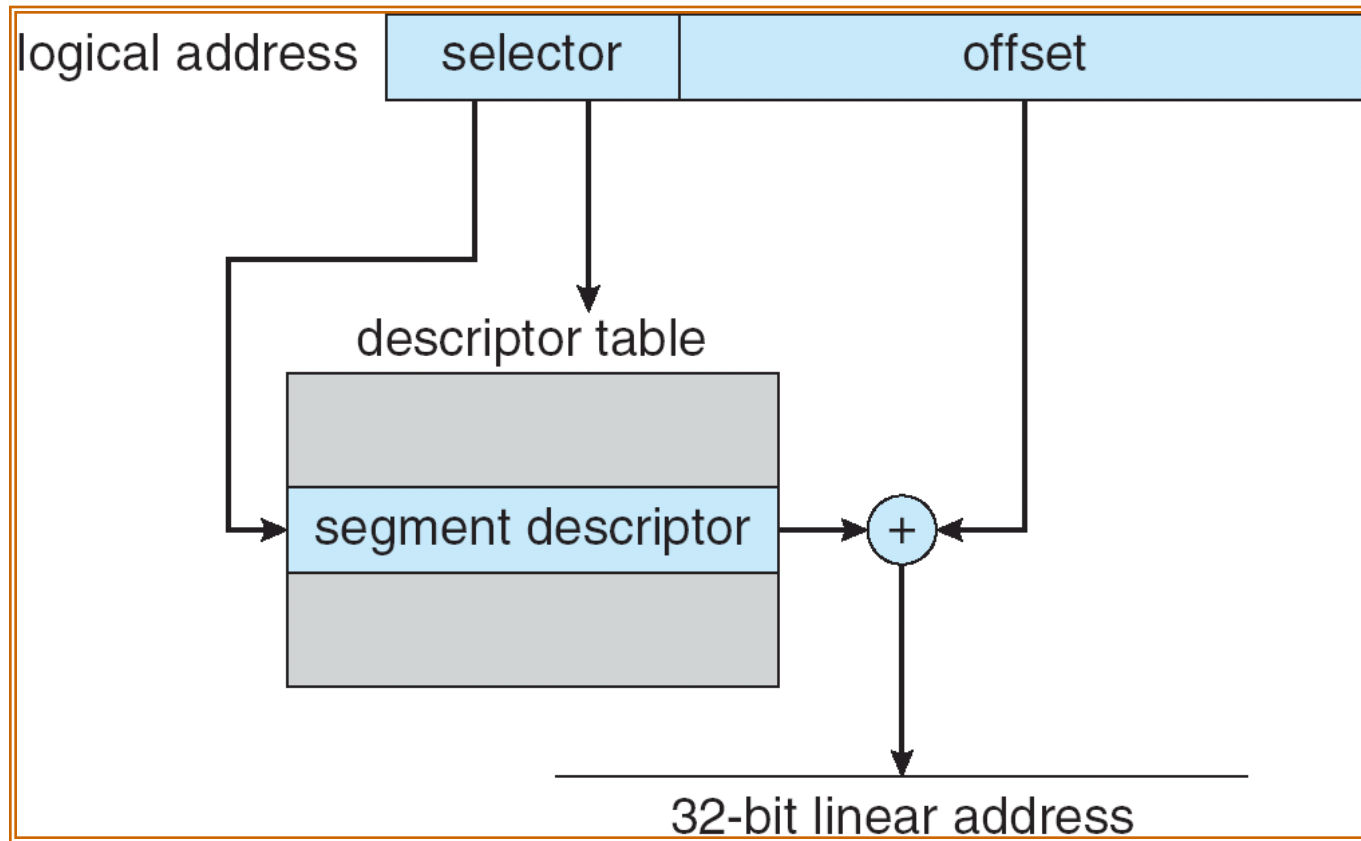
- **Supports both segmentation and segmentation with paging**
- **CPU generates logical address**
  - **Given to segmentation unit**
    - **Which produces linear addresses**
  - **Linear address given to paging unit**
    - **Which generates physical address in main memory**
    - **Paging units form equivalent of MMU**

# Logical to Physical Address Translation in Pentium

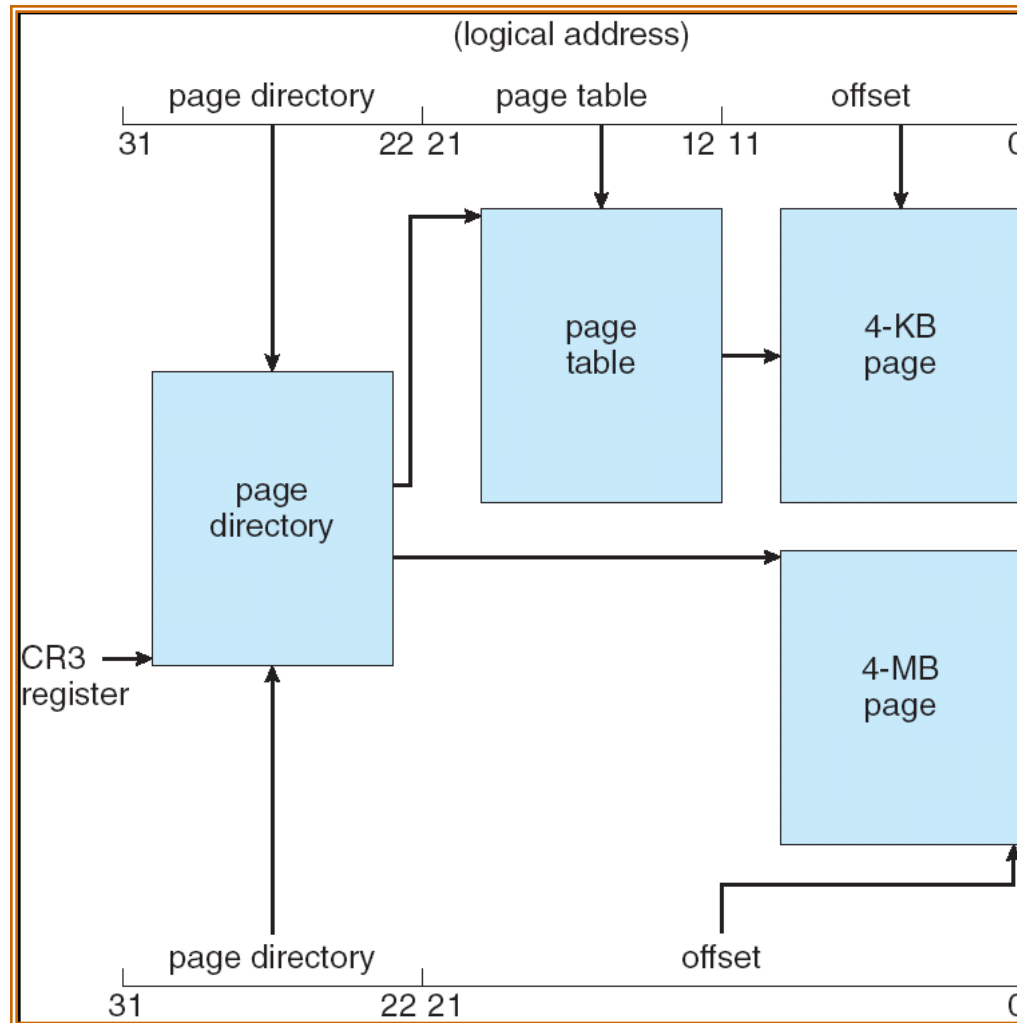


page number		page offset
$p_1$	$p_2$	$d$
10	10	12

# Intel Pentium Segmentation



# Pentium Paging Architecture





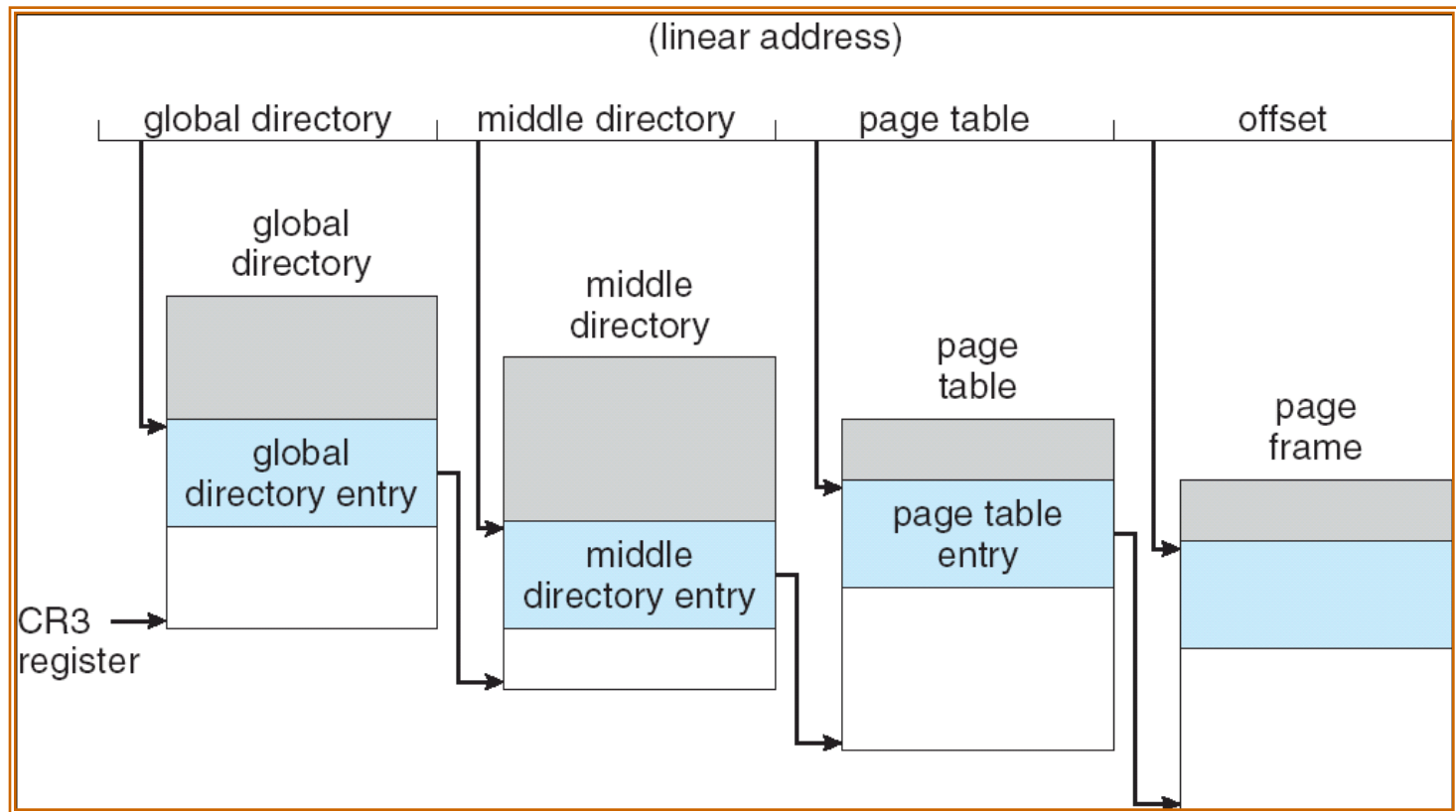
# Linear Address in Linux

---

**Broken into four parts:**

global directory	middle directory	page table	offset
---------------------	---------------------	---------------	--------

# Three-level Paging in Linux







# 作业1

---

- 试比较内部碎片与外部碎片的异同。



## 作业2

---

- 如果有内存块100KB、500KB、200KB、300KB和600KB（按顺序），首次适应算法、最佳适应算法、最差适应算法各自将怎样放置大小分别为212KB、417KB、112KB和426KB（按顺序）的进程？哪一种算法的内存利用率最高？