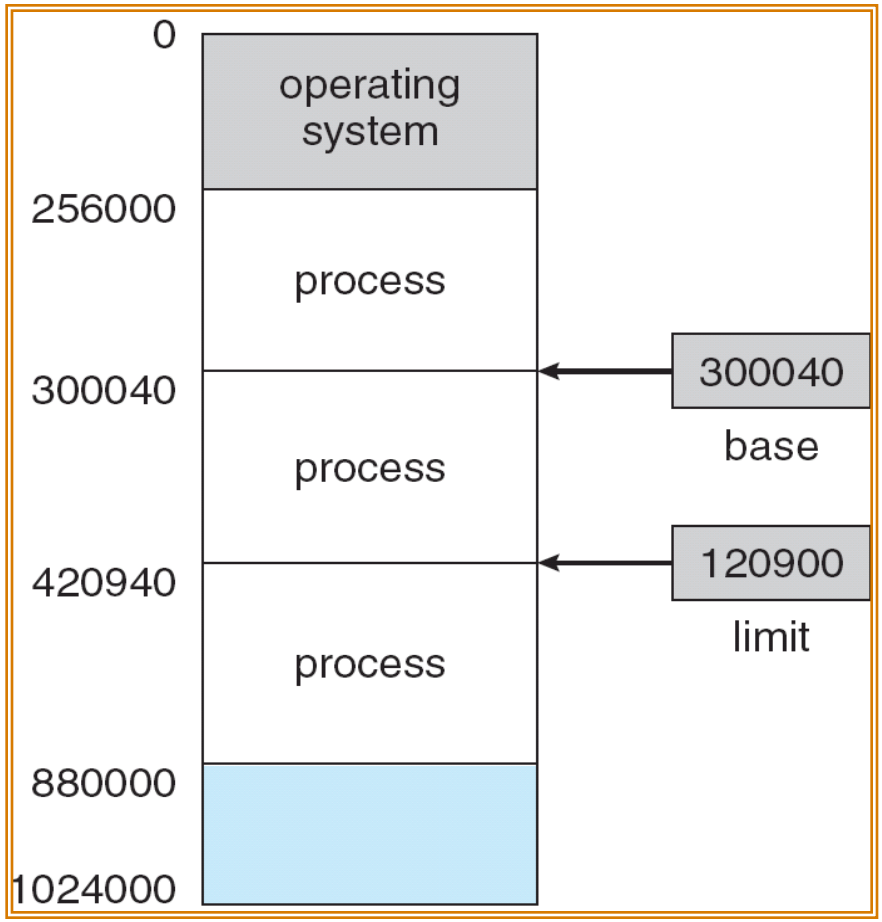
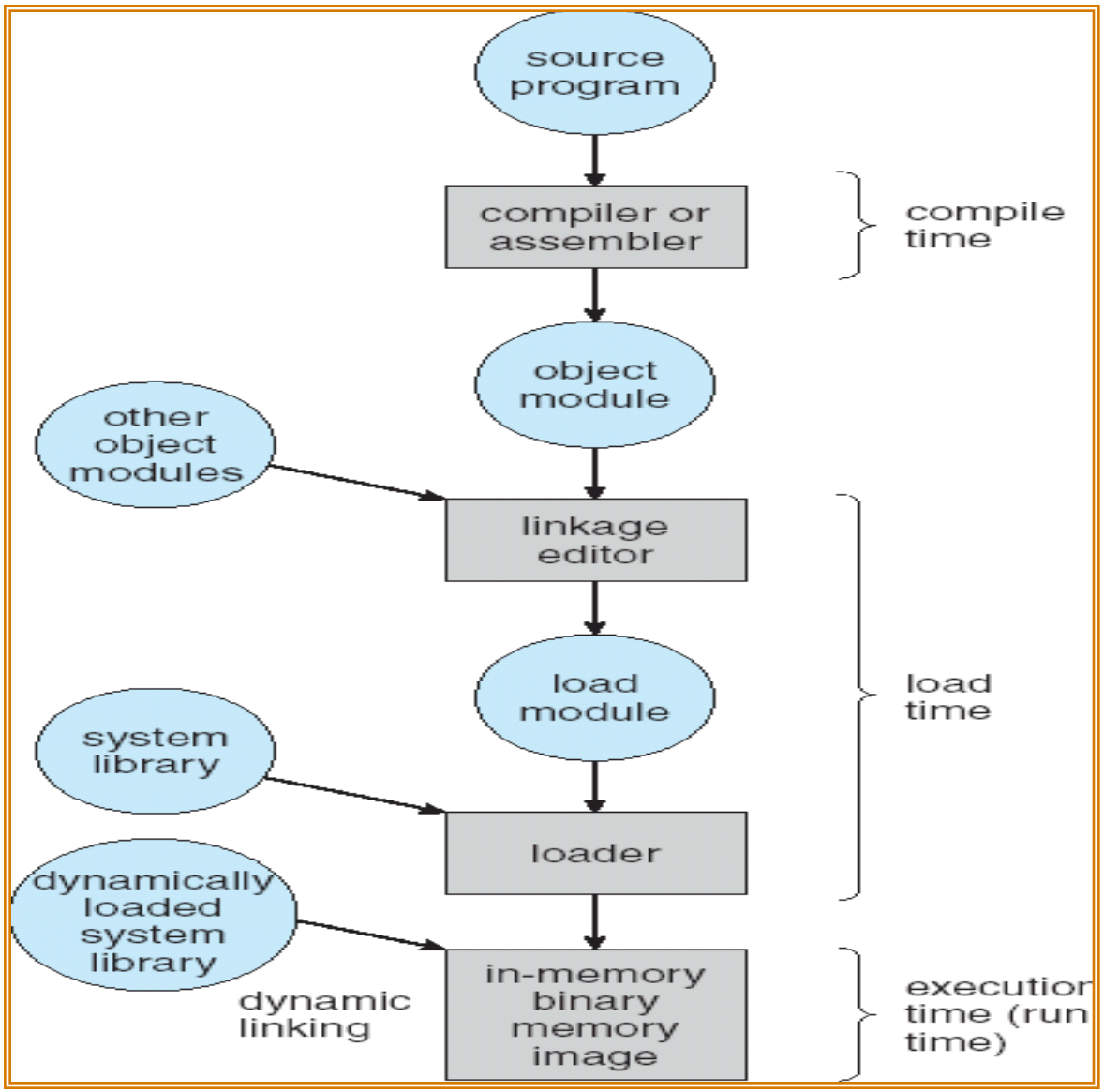
Memory Management

Base and Limit Registers

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



Multistep Processing of a User Program



Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** 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 (MMU) for address maps (e.g., base and limit registers)

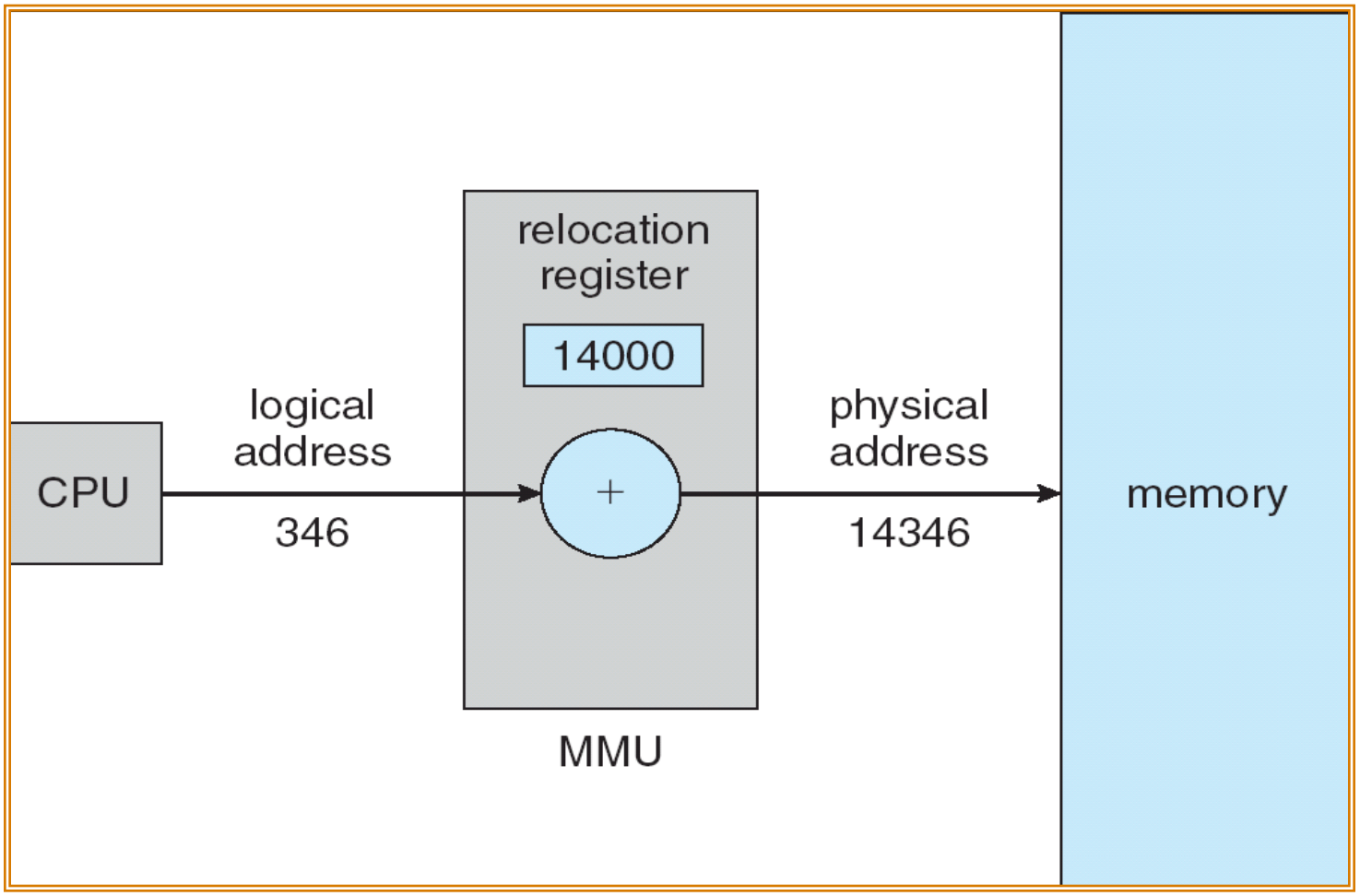
Logical vs. Physical Address Space

- The concept of a logical address space that is bound to a separate **physical address space** is 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 (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the **relocation 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

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required

Dynamic Linking

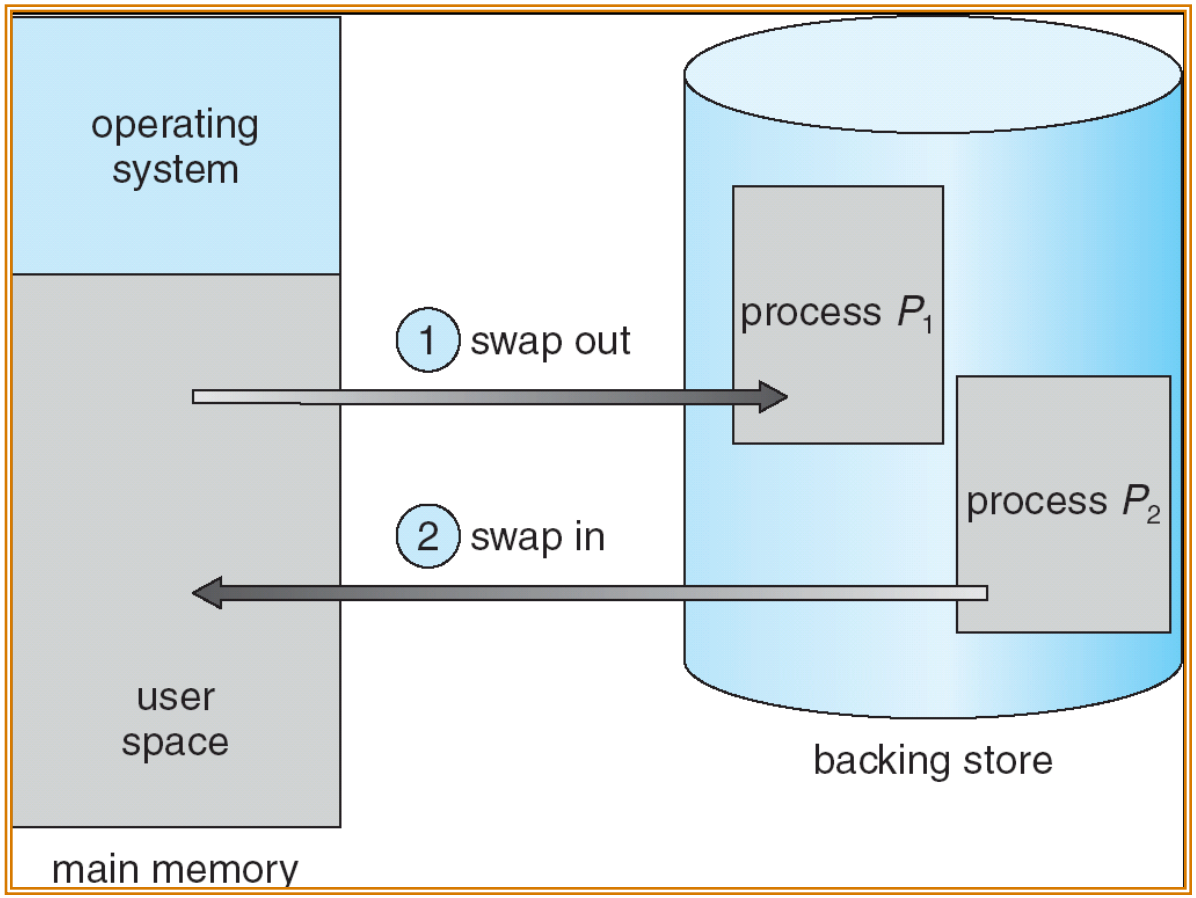
- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Dynamic linking is particularly useful for libraries

Swapping

- A process can be swapped temporarily out of memory to backing store (e.g., disk), 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

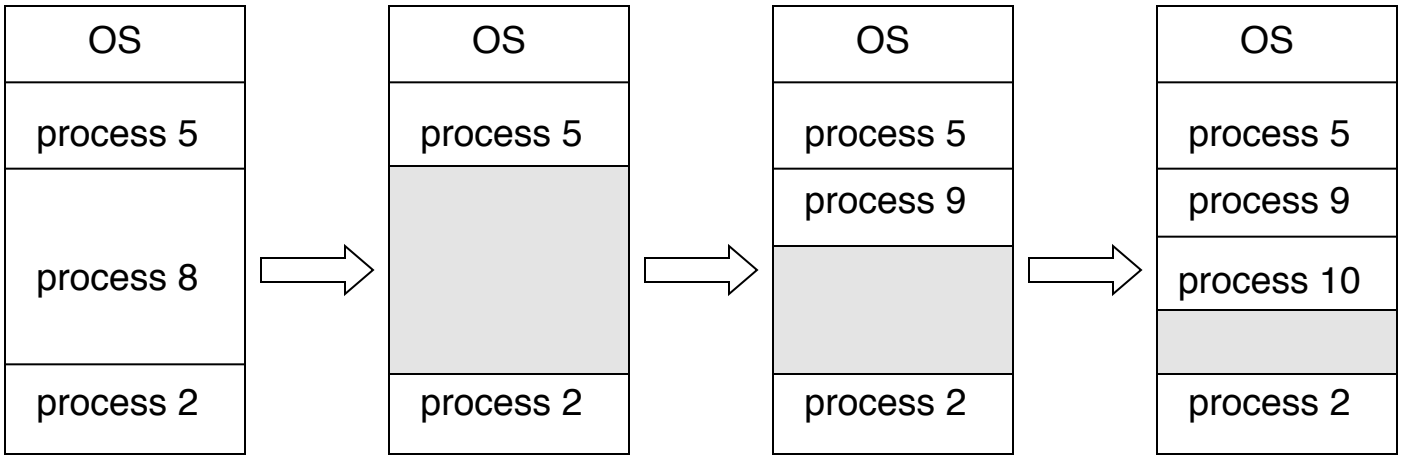
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

Schematic View of Swapping



Contiguous Allocation

- Multiple-partition allocation
  - 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
  - Operating system maintains information about: a) allocated partitions b) free partitions (hole)

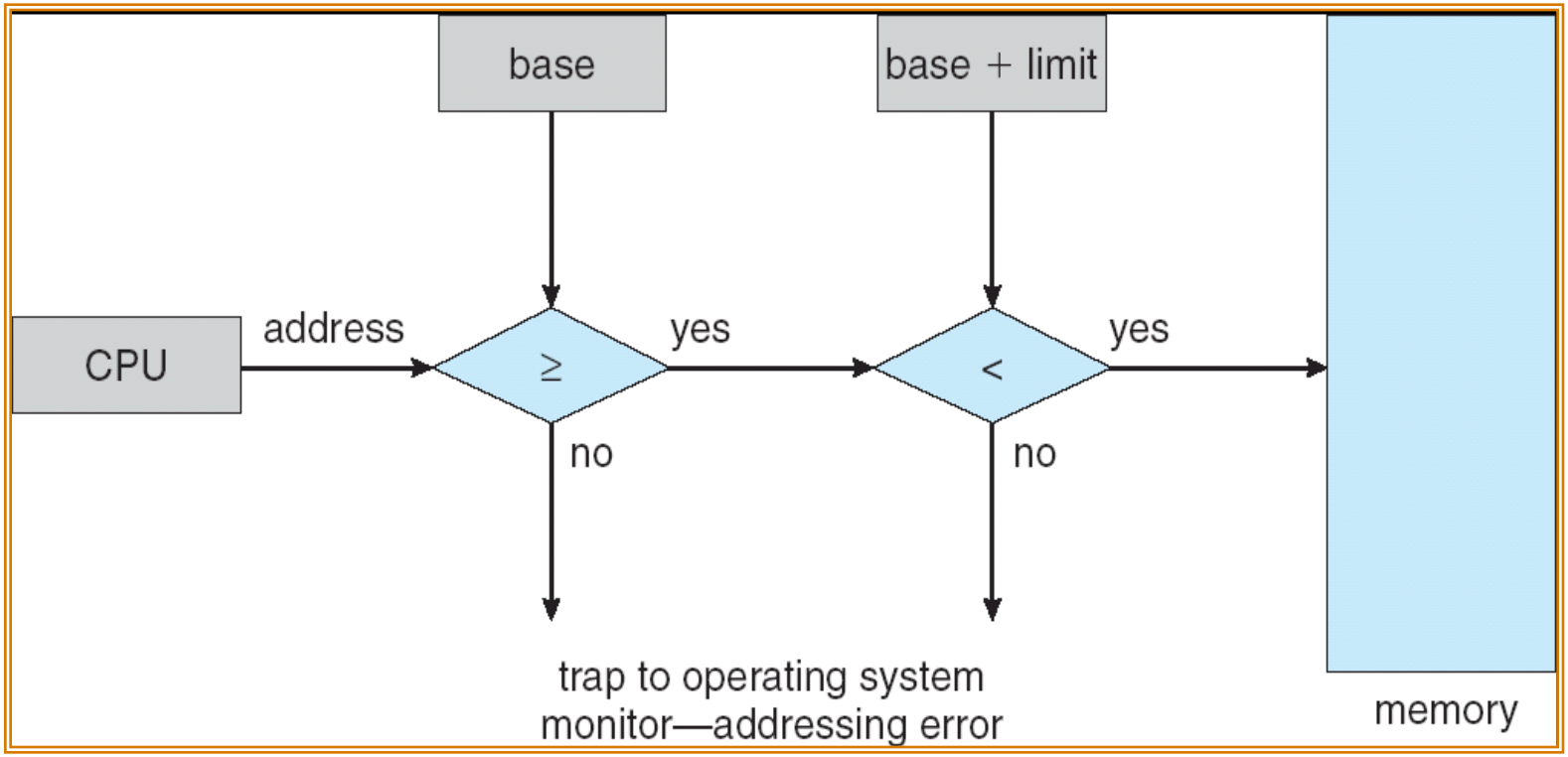


Contiguous Allocation (cont.)

- Main memory is usually divided into two partitions:
  - *Resident operating system*: usually held in low memory with interrupt vector
  - *User processes*: held in high memory

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - *Base register*: contains value of smallest physical address
  - *Limit register*: contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*

HW address protection with base and limit registers



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

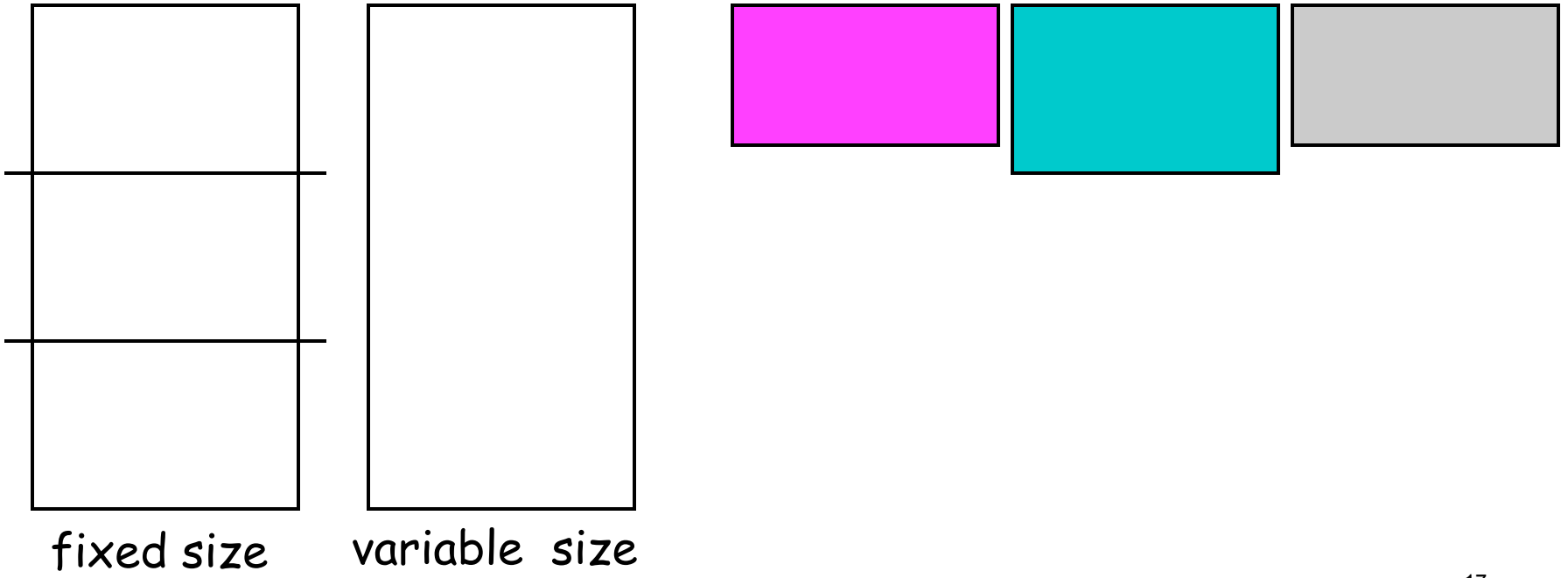
Question:
What are the tradeoffs among those algorithms?

Fragmentation

- **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
- 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

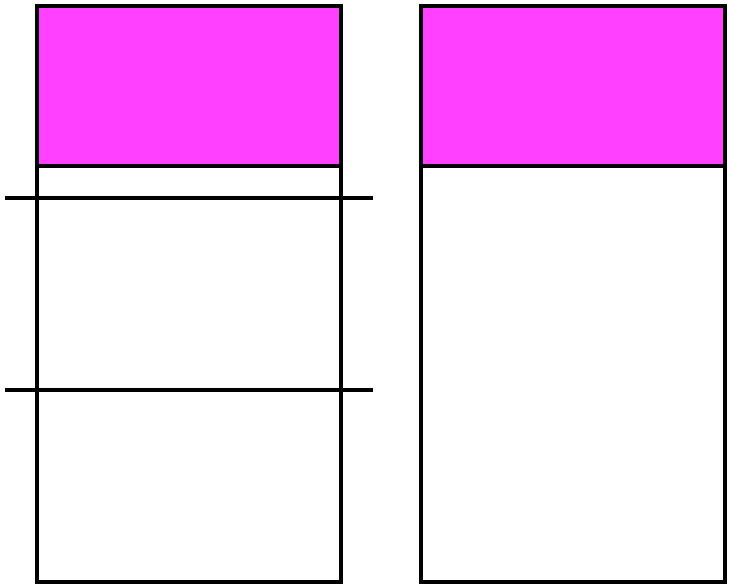
Memory Partitioning

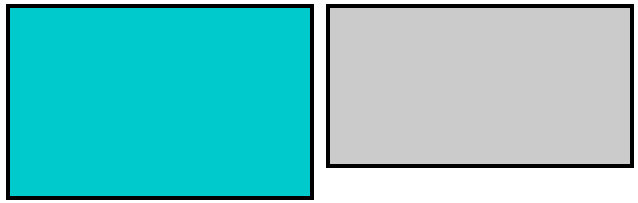
- Internal Fragmentation
  - Fixed size, some unused (static)
- External fragmentation
  - Variable size, gaps between



Memory Partitioning

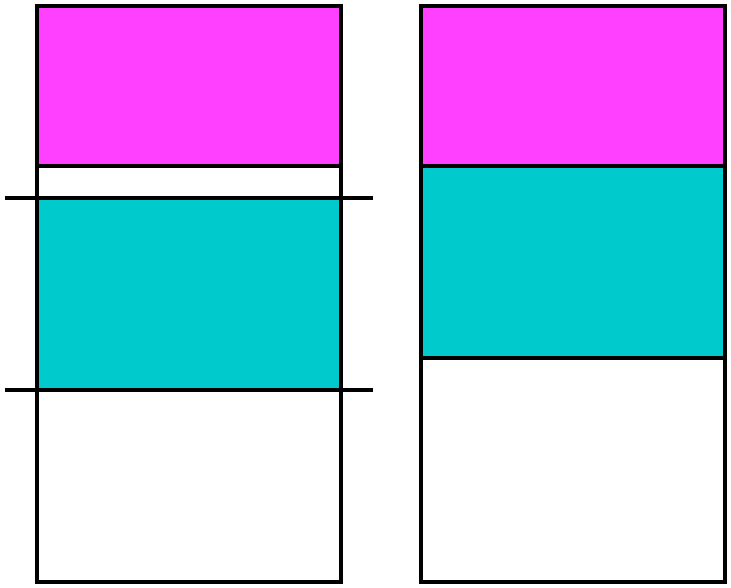
- Internal Fragmentation
  - Fixed size, some unused (static)
- External fragmentation
  - Variable size, gaps between





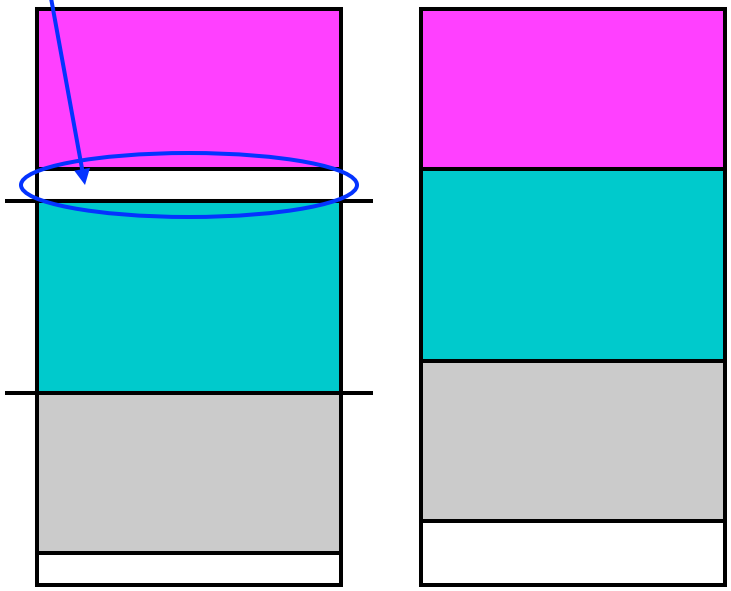
Memory Partitioning

- Internal Fragmentation
  - Fixed size, some unused (static)
- External fragmentation
  - Variable size, gaps between



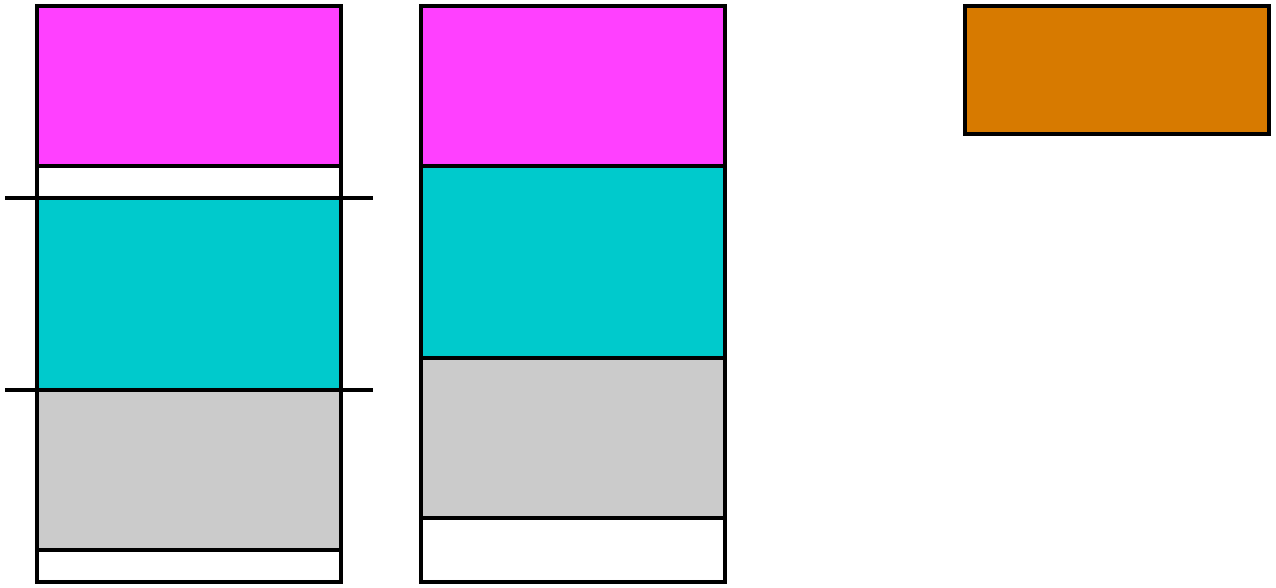
Memory Partitioning

- Internal Fragmentation
  - Fixed size, some unused (static)
- External fragmentation
  - Variable size, gaps between



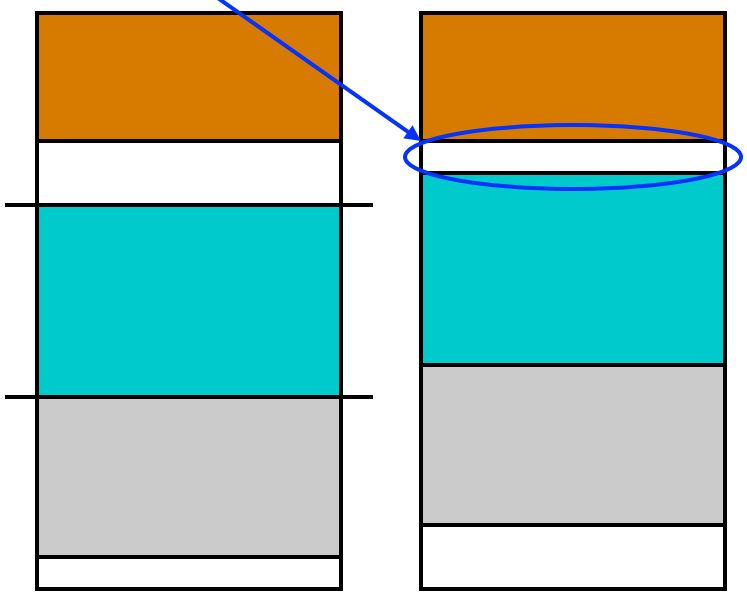
Memory Partitioning

- Internal Fragmentation
  - Fixed size, some unused (static)
- External fragmentation
  - Variable size, gaps between



Memory Partitioning

- Internal Fragmentation
  - Fixed size, some unused (static)
- External fragmentation
  - Variable size, gaps between

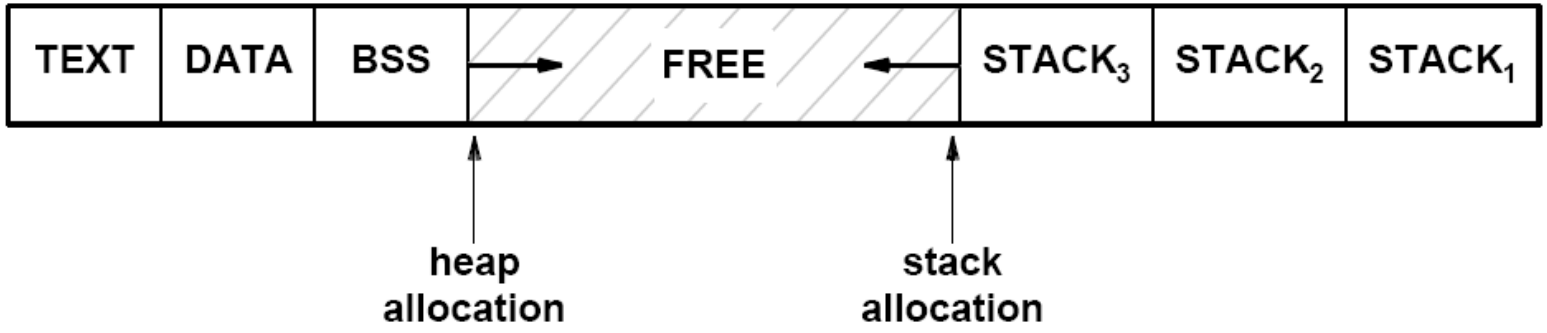


Example Implementation in Xinu

- Single free list
  - Ordered by increasing address
  - Singly-linked
  - Initialized at system startup to all free memory

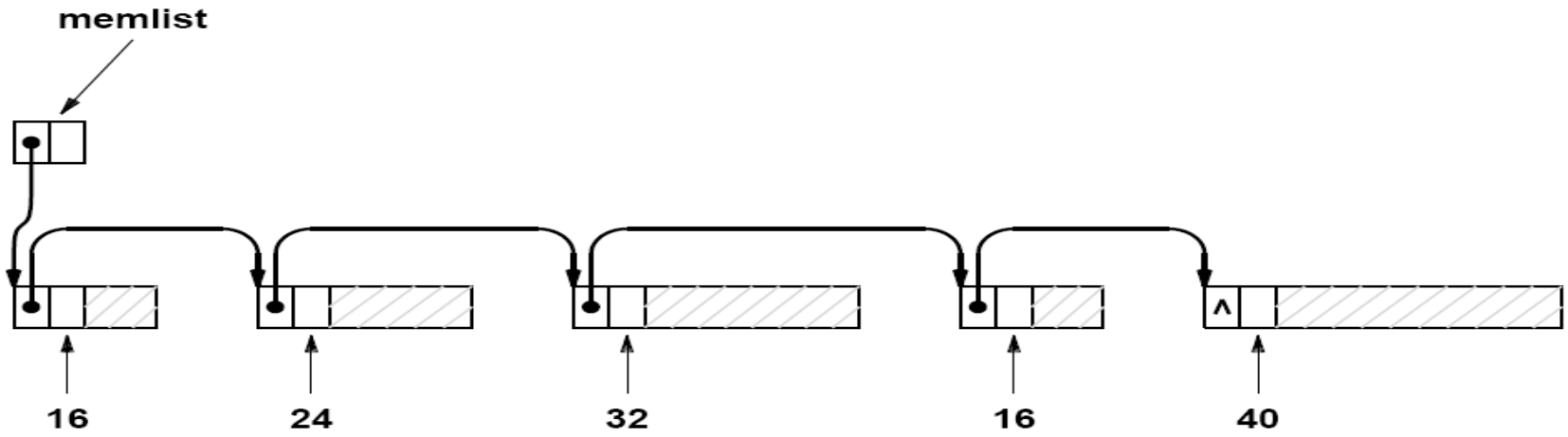
- Allocation policy
  - Heap: first-fit
  - Stack: last-fit

Result of Xinu Allocation Policy



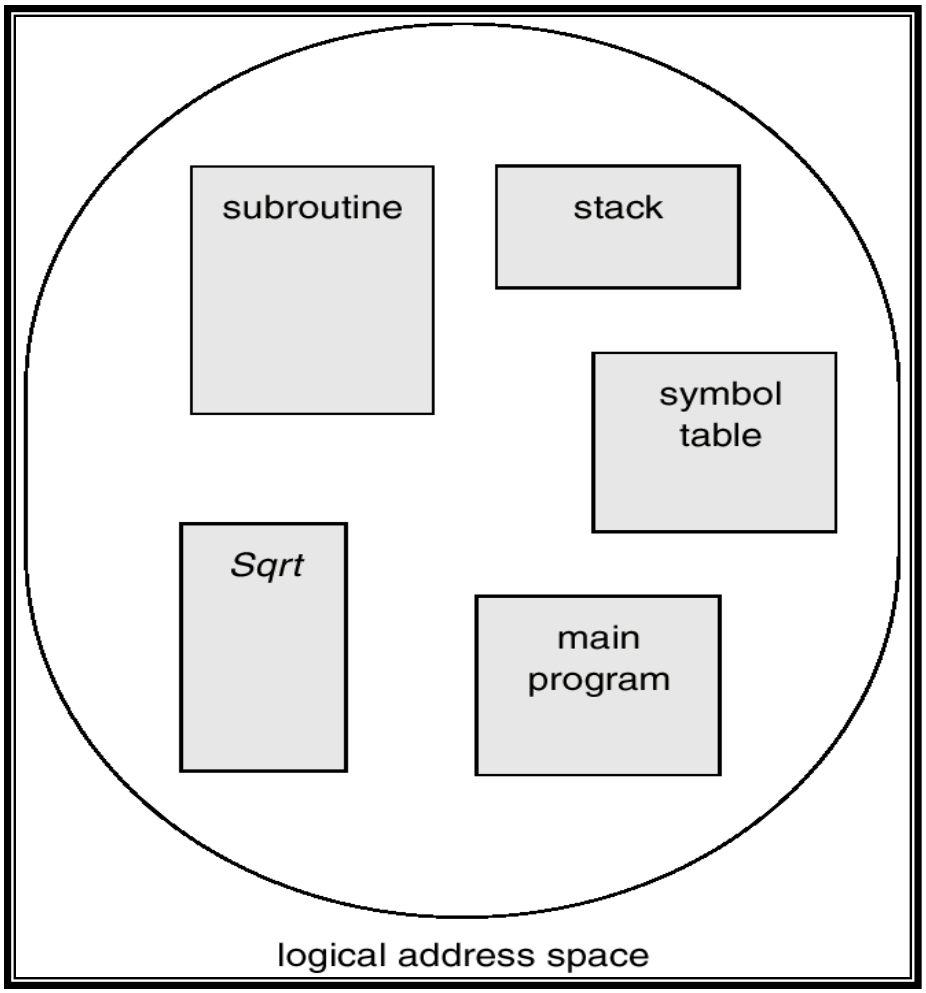
- Stack allocation from highest free memory
- Heap allocation from lowest free memory

Illustration of Xinu Free List

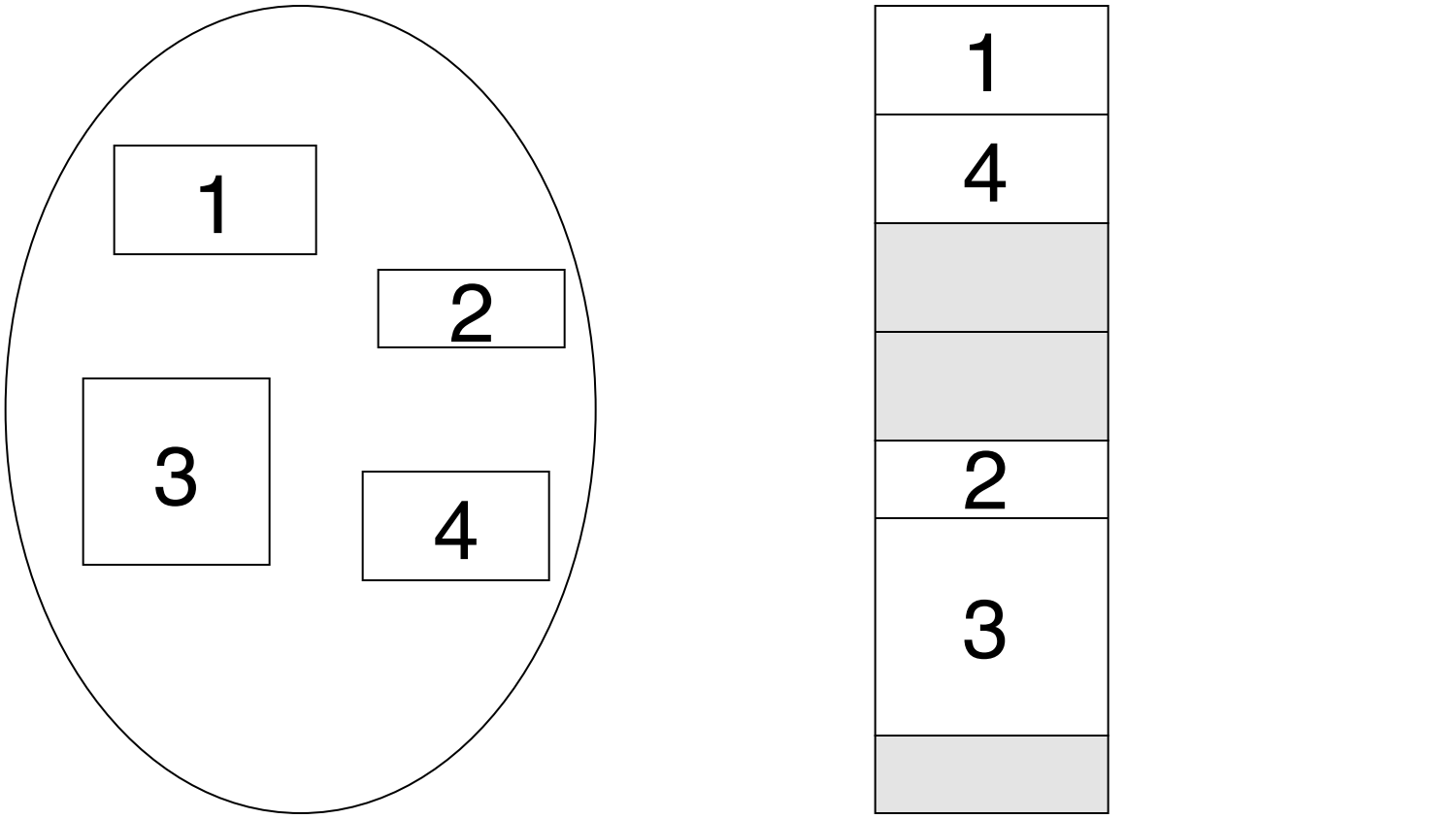


- List in order of address
- Related Files*: h/mem.h sys/getmem.c sys/freemem.c*

User's View of a Program



Logical View of Segmentation



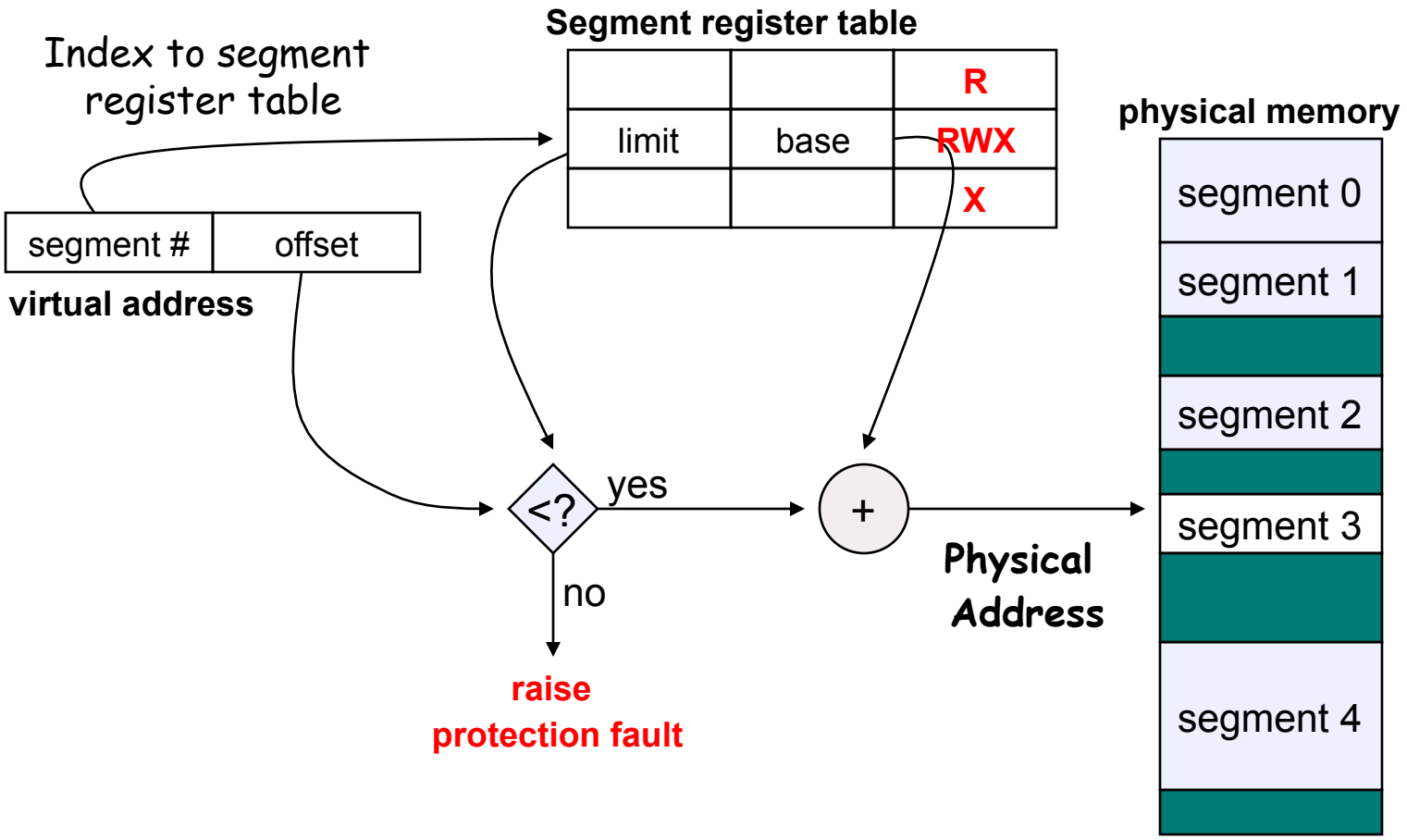
Segmentation

- Logical address consists of a pair:

  <segment-number, offset>

- Segment table where each entry has:
  - Base: contains the starting physical address where the segments reside in memory.
  - Limit: specifies the length of the segment.

Segment Lookup



Segmentation

- Common in early minicomputers
  - Small amount of additional hardware – 4 or 8 segments
  - Used effectively in Unix
- Good idea that has persisted and supported in current hardware and OSs
  - X86 supports segments
  - Linux supports segments

Question:

Do we still have external fragmentation problem? If yes, can we further improve it?

Paging

- ***Noncontiguous*** memory allocation
  - Physical address space of a process can be noncontiguous
  - Process is allocated physical memory whenever the latter is available
- Frames:
  - Divide **physical memory** into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Pages:
  - Divide **logical memory** into blocks of same size called **pages**

Paging (cont.)

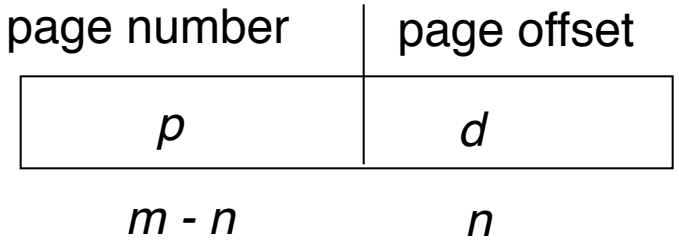
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses

Question:

Do we still have external fragmentation problem? Do we still have internal fragmentation problem?

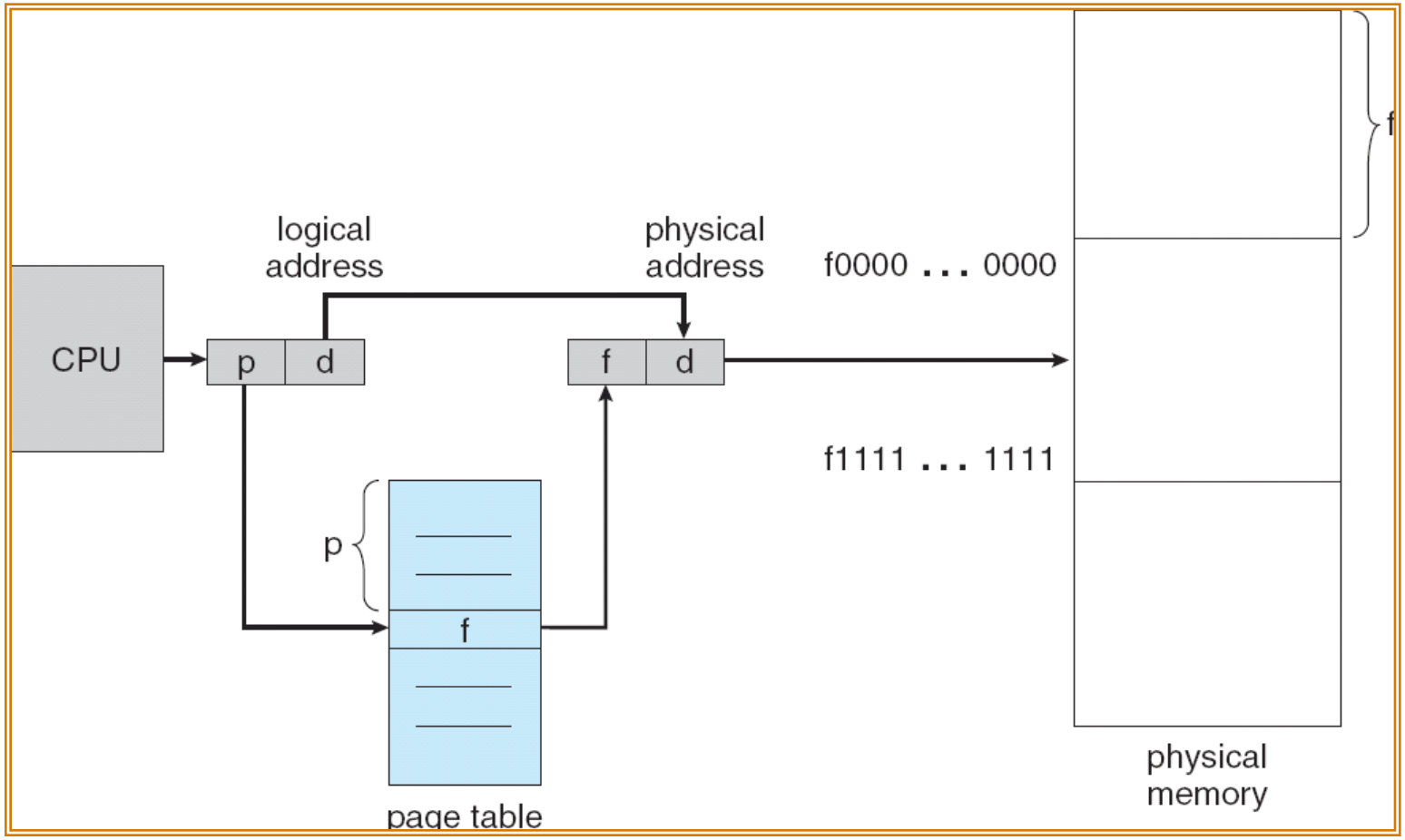
Address Translation Scheme

- 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

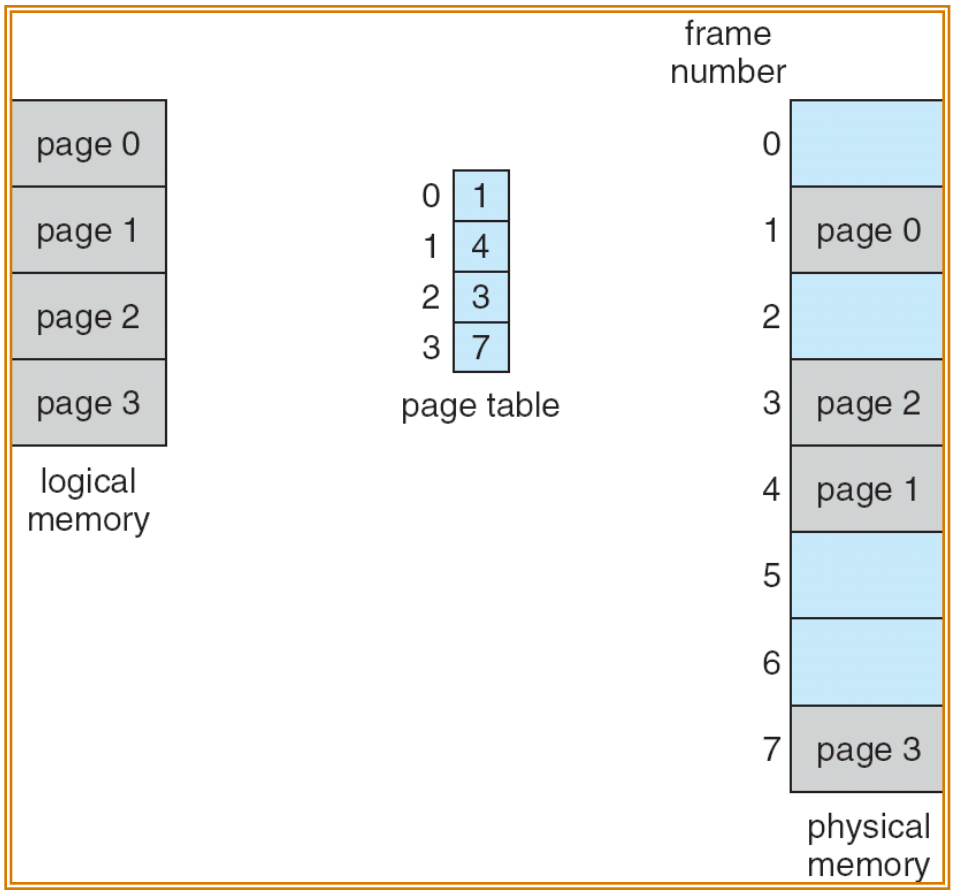


  - For given logical address space 2^m *and page size* 2^n

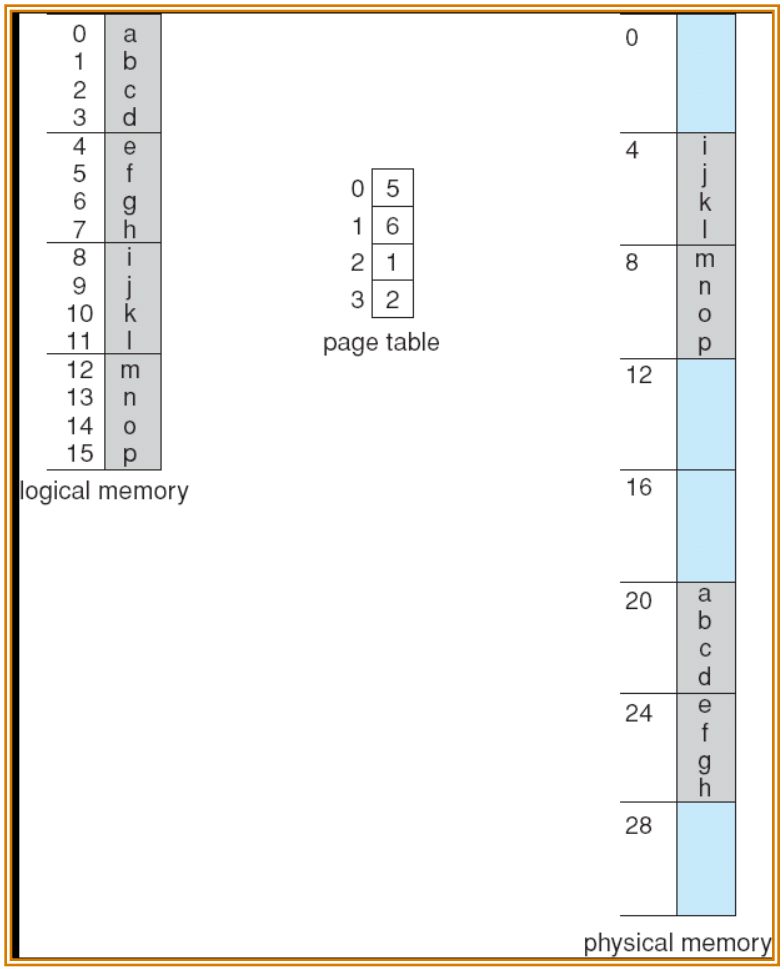
Paging Hardware



Paging Model of Logical and Physical Memory

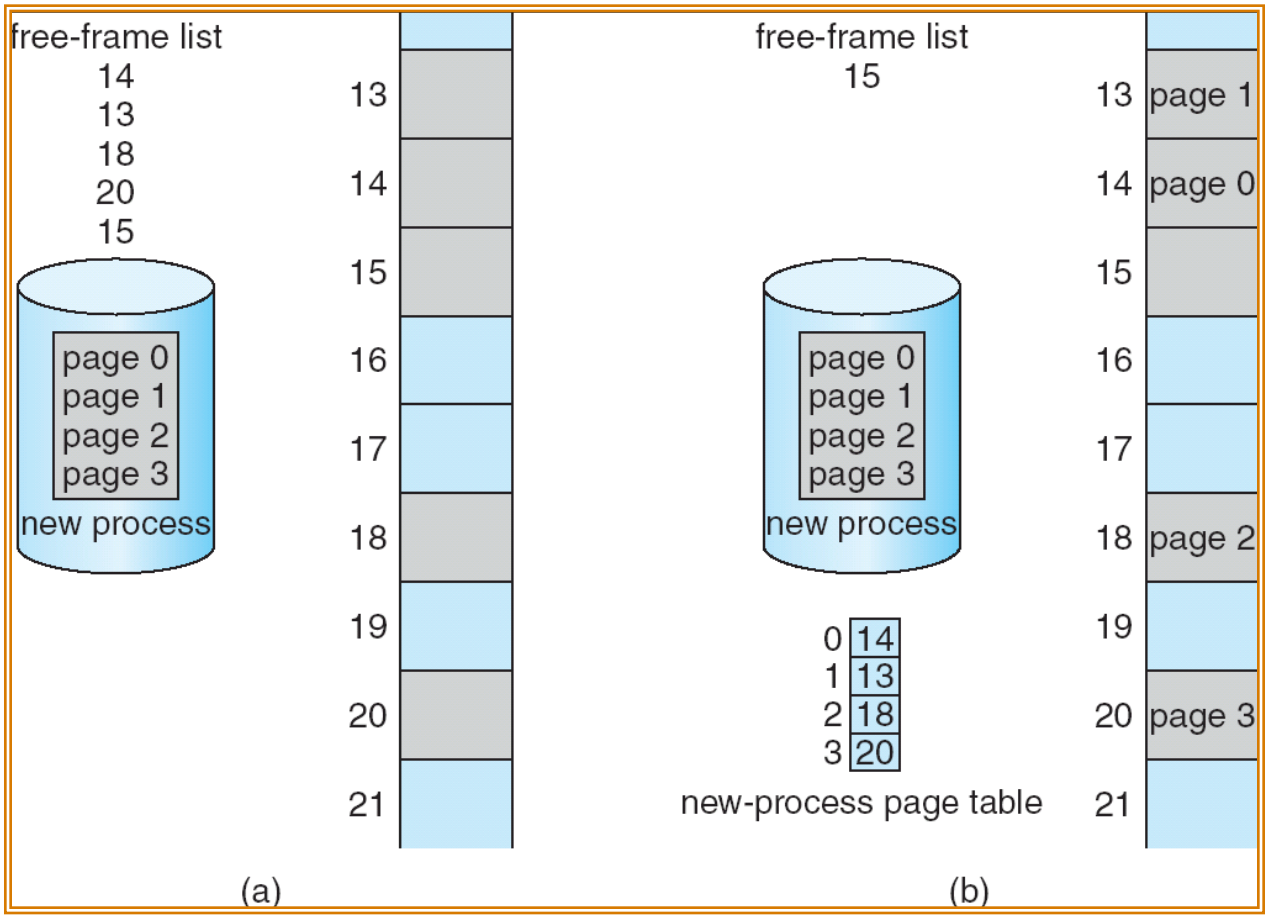


Paging Example



32-byte memory and 4-byte pages

Free Frames



Before allocation

After allocation

Question!

- Where should we keep the page table?
- How do we find it?

Implementation of Page Table

- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the base address of the page table
- **Page-table length register (PRLR)** indicates size of the page table
- In this scheme every data/instruction access requires ***two memory accesses*.**
  - One for the page table
  - One for the data/instruction

Reduce Memory Access

- 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 buffers (TLBs)**
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - The ID of current running process must match the ASID in the TLB entry
  - ASID mismatch is considered as TLB miss

Associative Memory (Hardware)

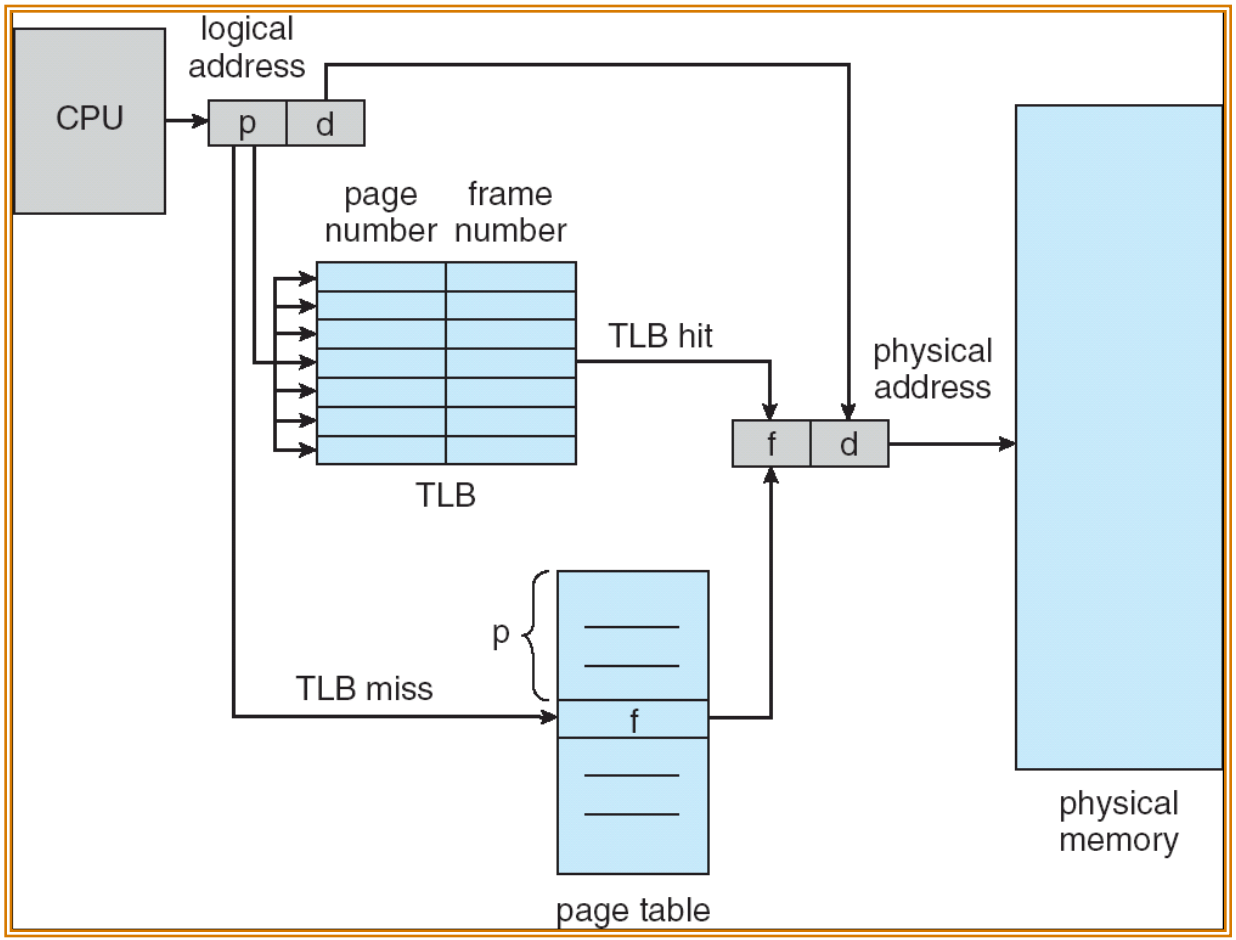
- Associative memory – parallel search

Page #	Frame #

Address translation (p, d)

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

Paging Hardware With TLB



Effective Access Time

- TLB Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

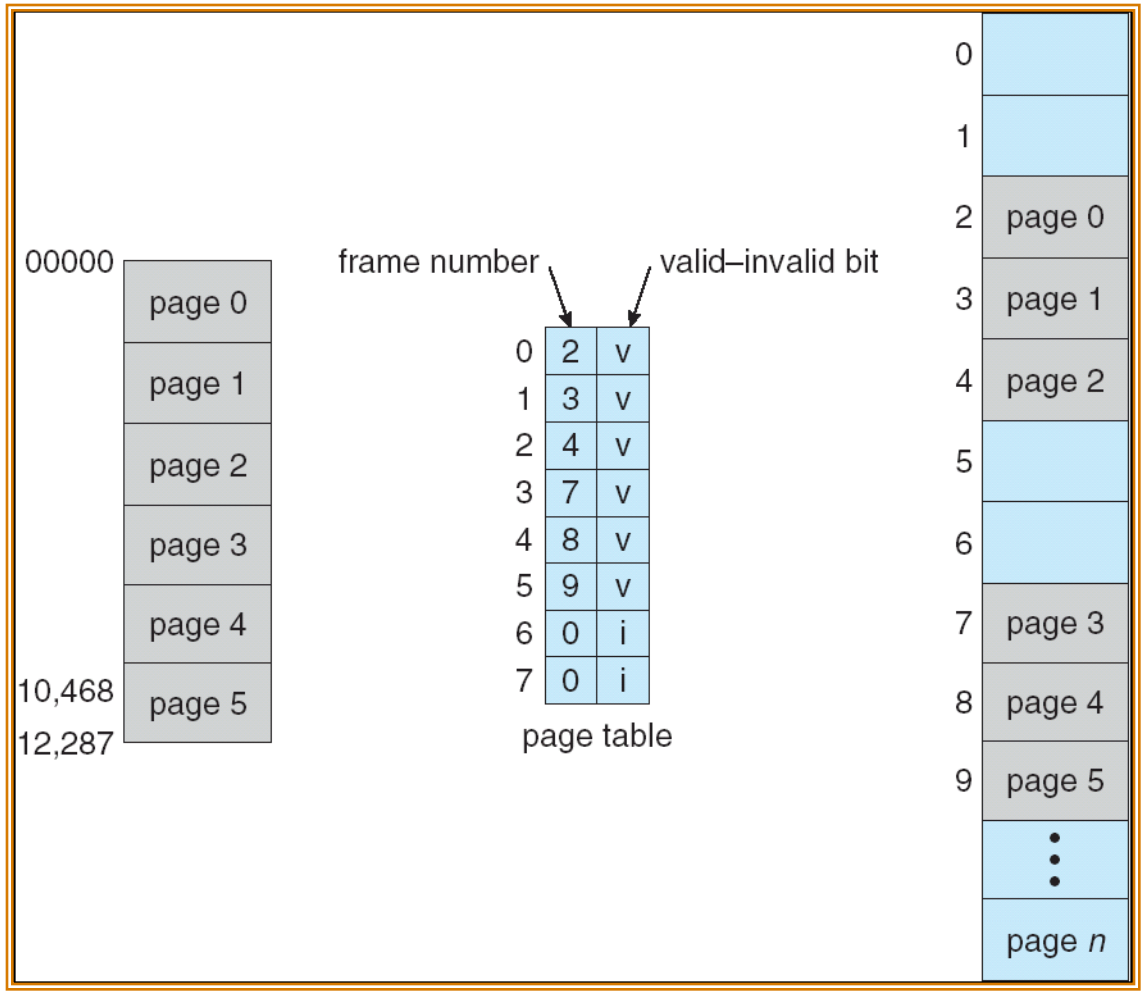
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

Memory Protection

- Memory protection implemented by associating protection bit with each frame

- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

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

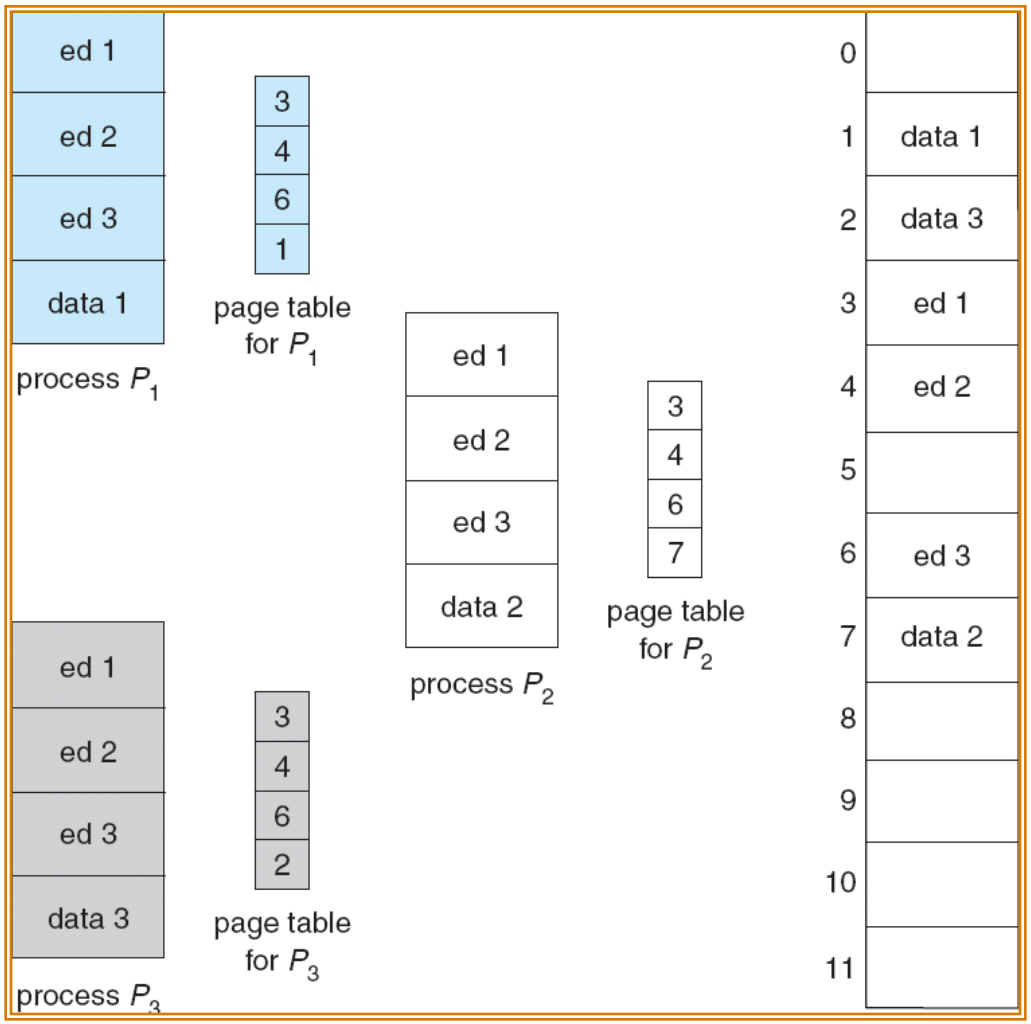


Shared Pages

- **Shared code**
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes

- **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



Structure of the Page Table

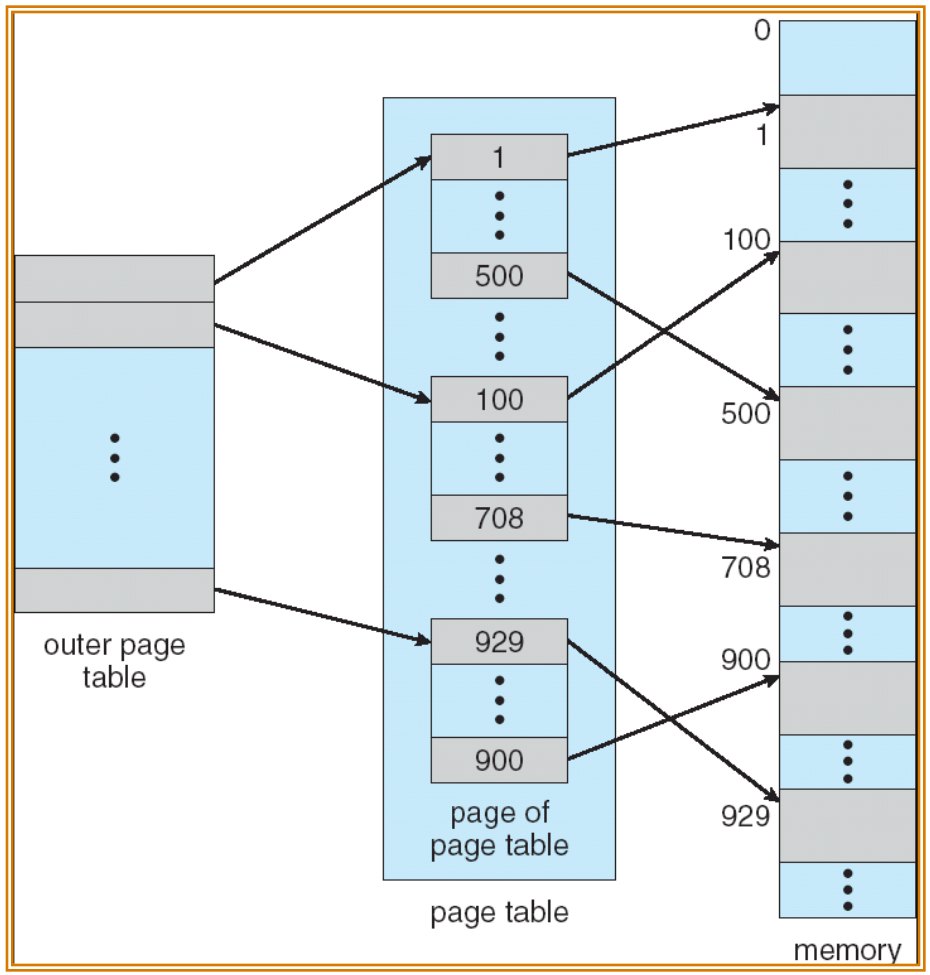
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

- Break up the logical address space into multiple page tables

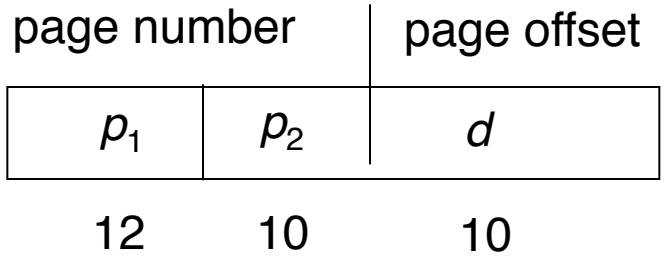
- A simple technique is a two-level page table

Two-Level Page-Table Scheme



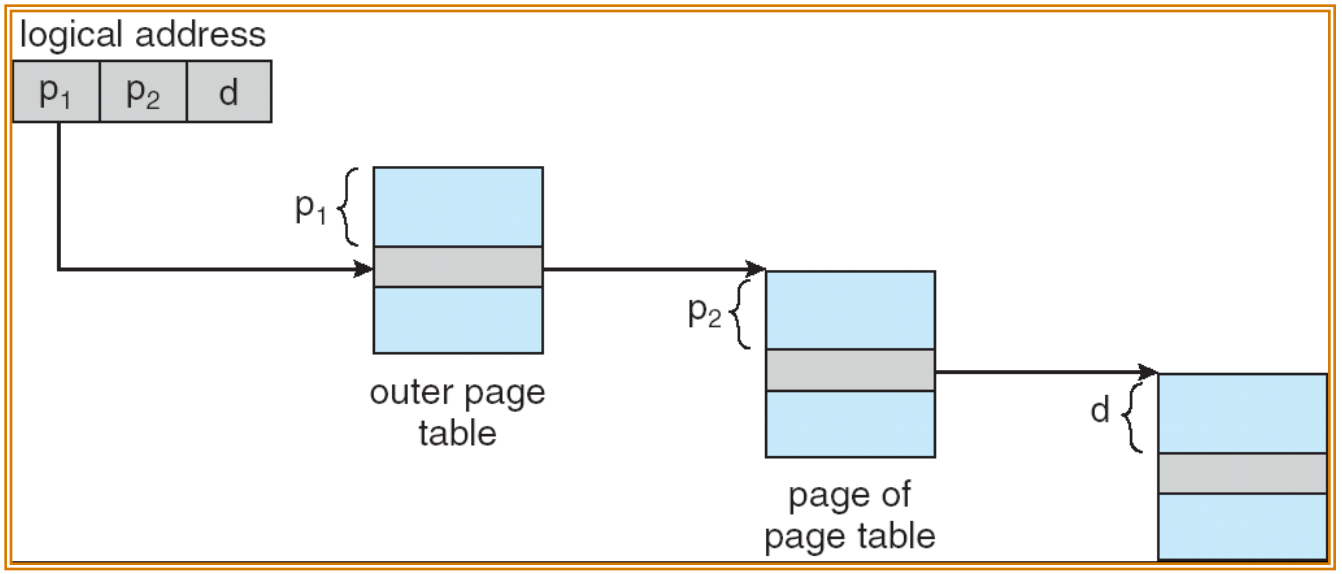
Two-Level Paging Example

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-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

Address-Translation Scheme



Three-level Paging Scheme (64 bit)

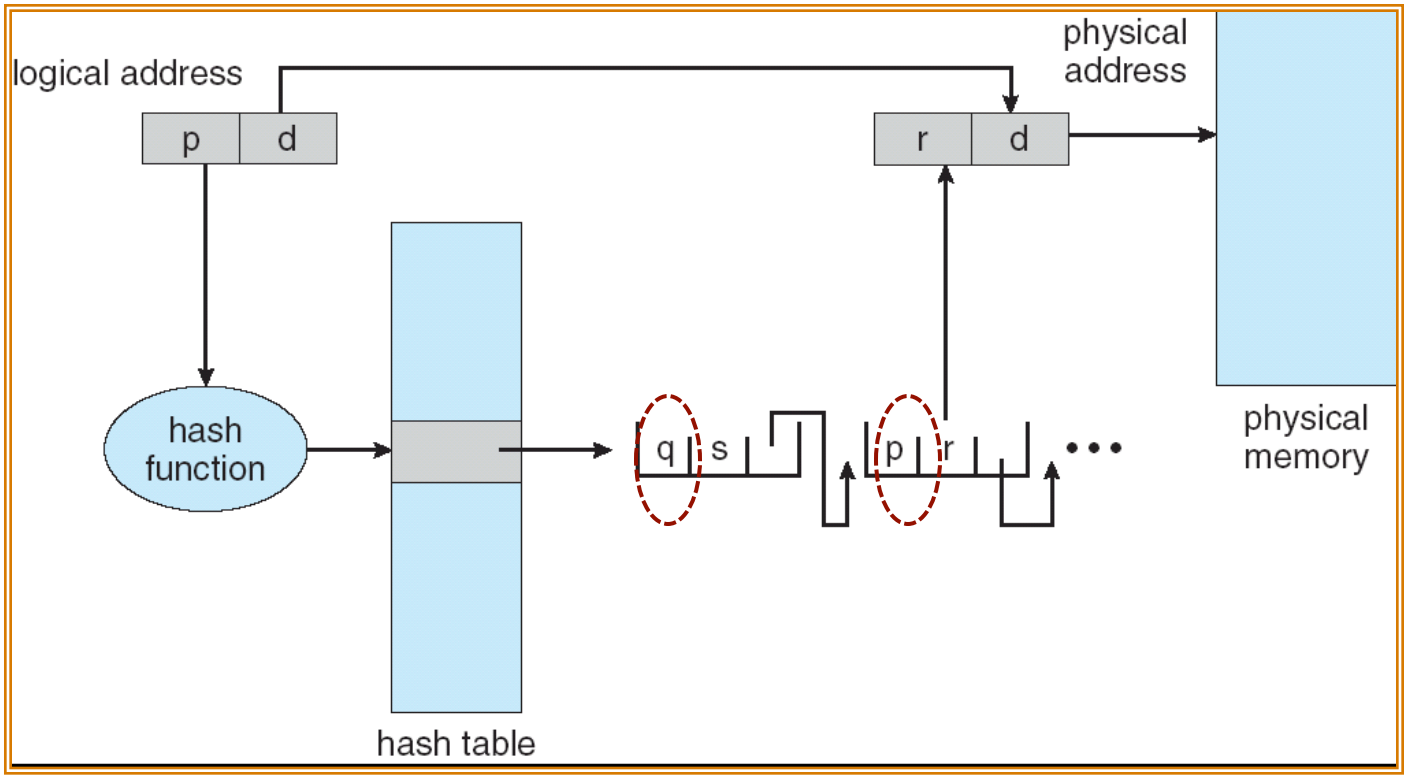
outer page	inner page	offset
p_1	p_2	d
42	10	12

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

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.
- 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 page 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 to store each page table, but increases time needed to search the table when a page reference occurs

Inverted Page Table Architecture