CSE-3211, Operating System

Chapter 8: Memory Management

Md Saidur Rahman Kohinoor
Lecturer, Dept. of Computer Science
Leading University – Sylhet
kohinoor_cse@lus.ac.bd

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation

Contents to be covered:

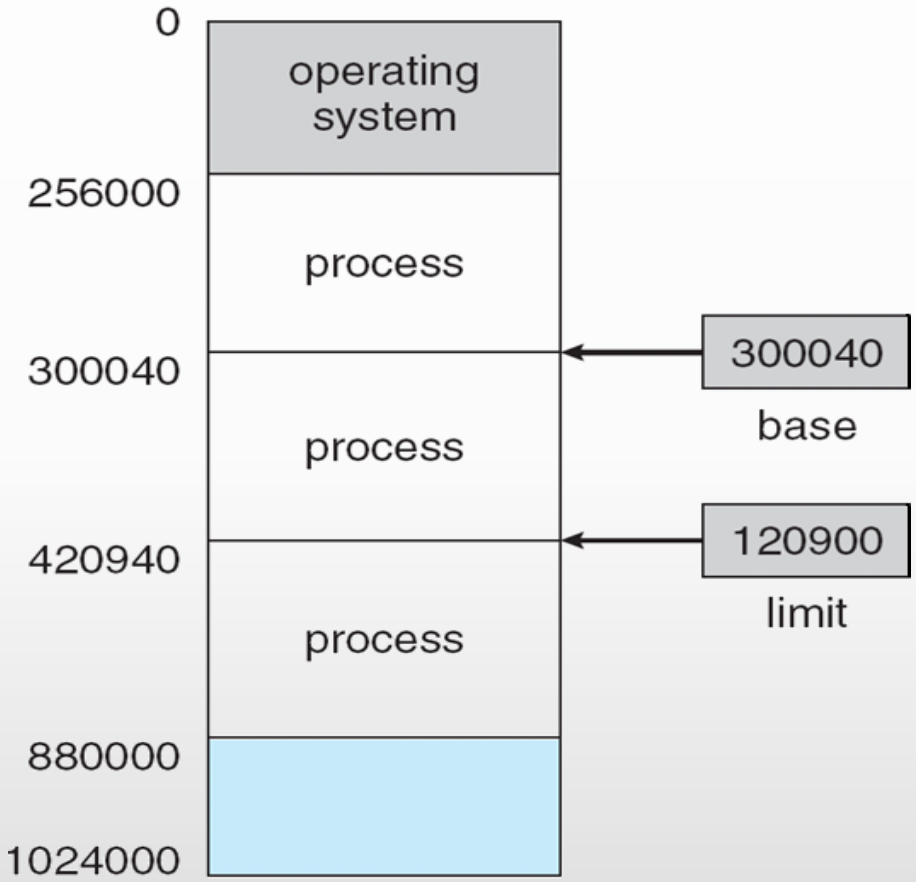
- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of the Page Table

Background

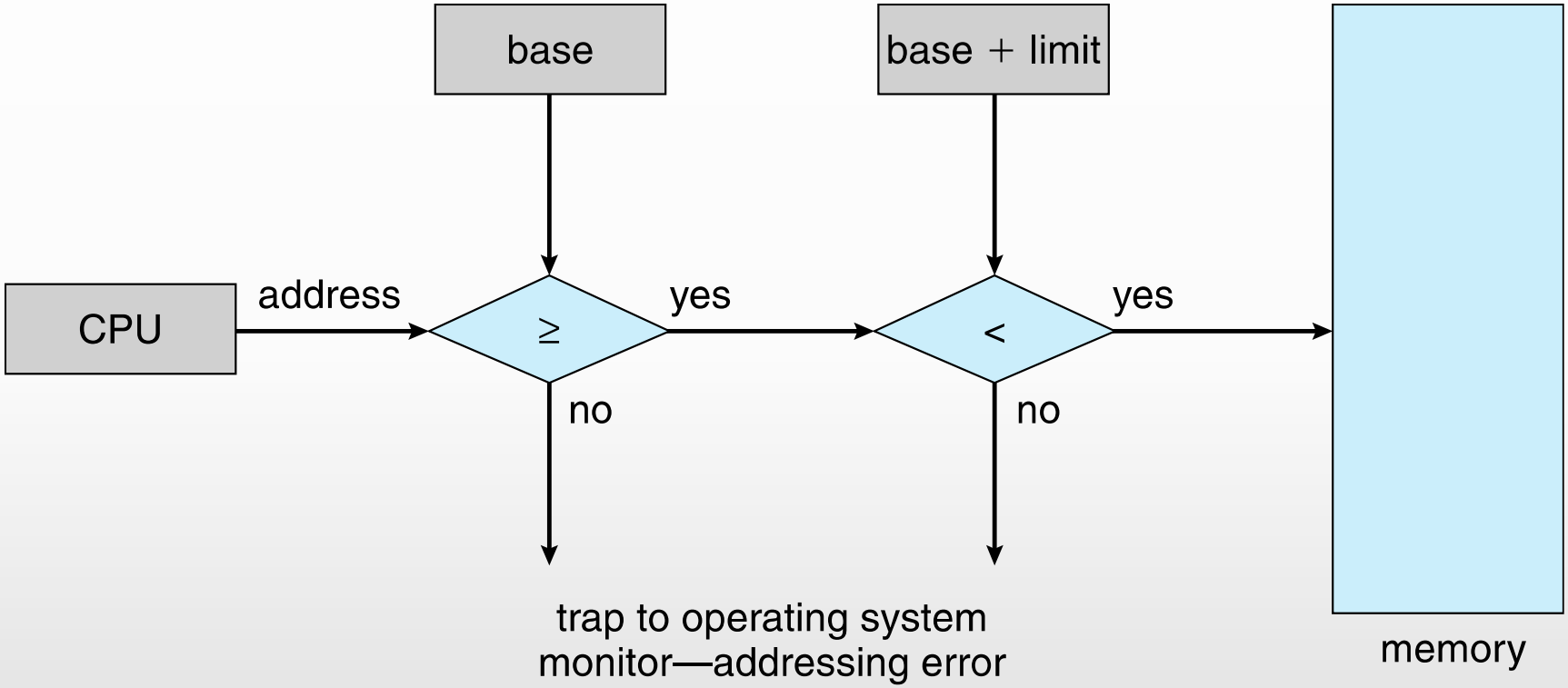
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are the only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock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

Base and Limit Registers

- Need to make sure that each process has a separate memory space.
- Need to determine the range of legal addresses that the process may access.
- This protection can be provided by a pair of **base** and **limit registers** that define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



Hardware Address Protection with Base and Limit Registers



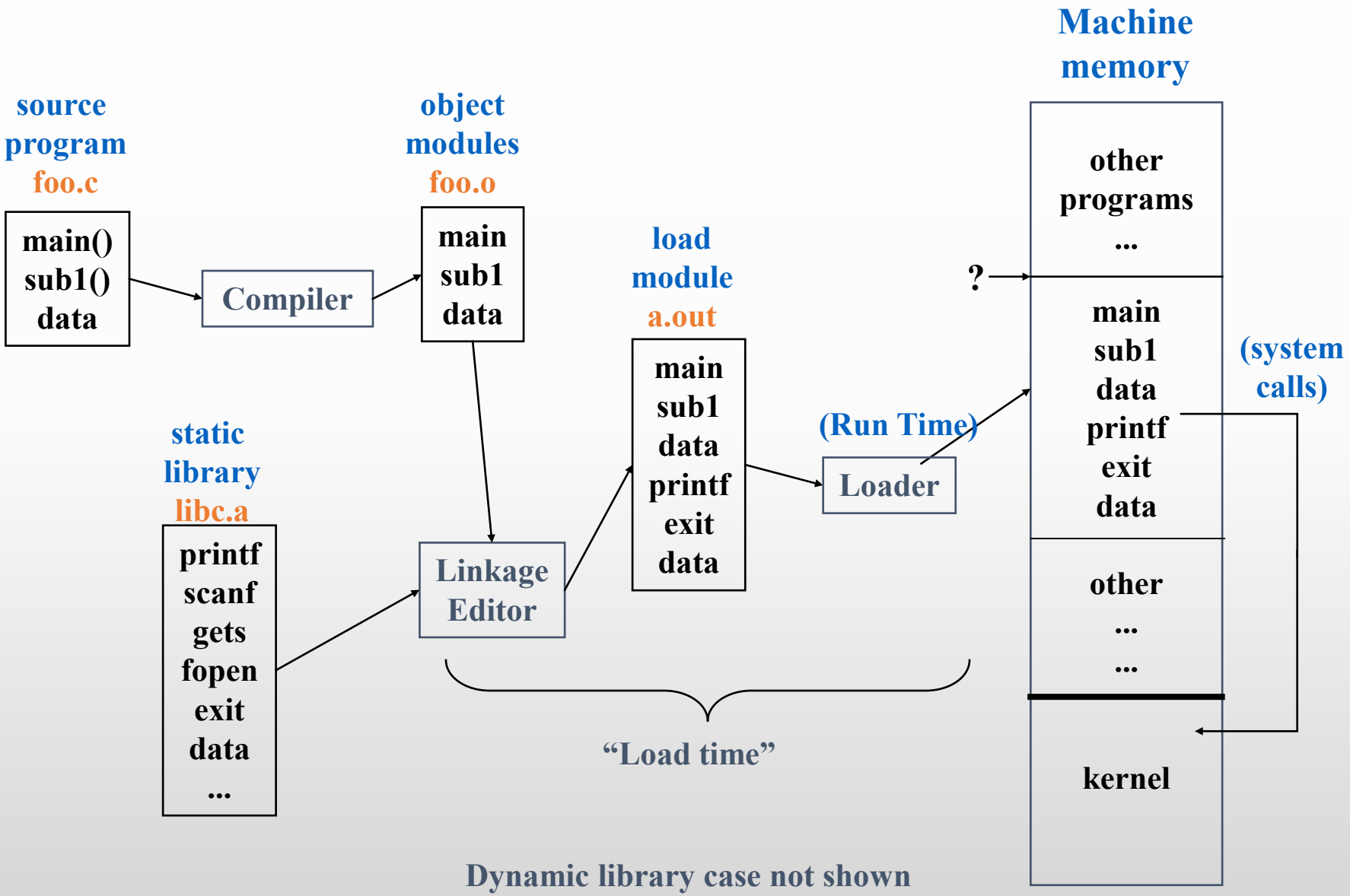
Address Binding

- Programs on disk, ready to be brought into memory to execute from an **input queue**
- Inconvenient to have the first user process physical address always at 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
    - i.e. "14 bytes from the beginning of this module"
  - Linker or loader will bind relocatable addresses to absolute addresses
    - i.e. 74014
  - Each binding maps one address space to another

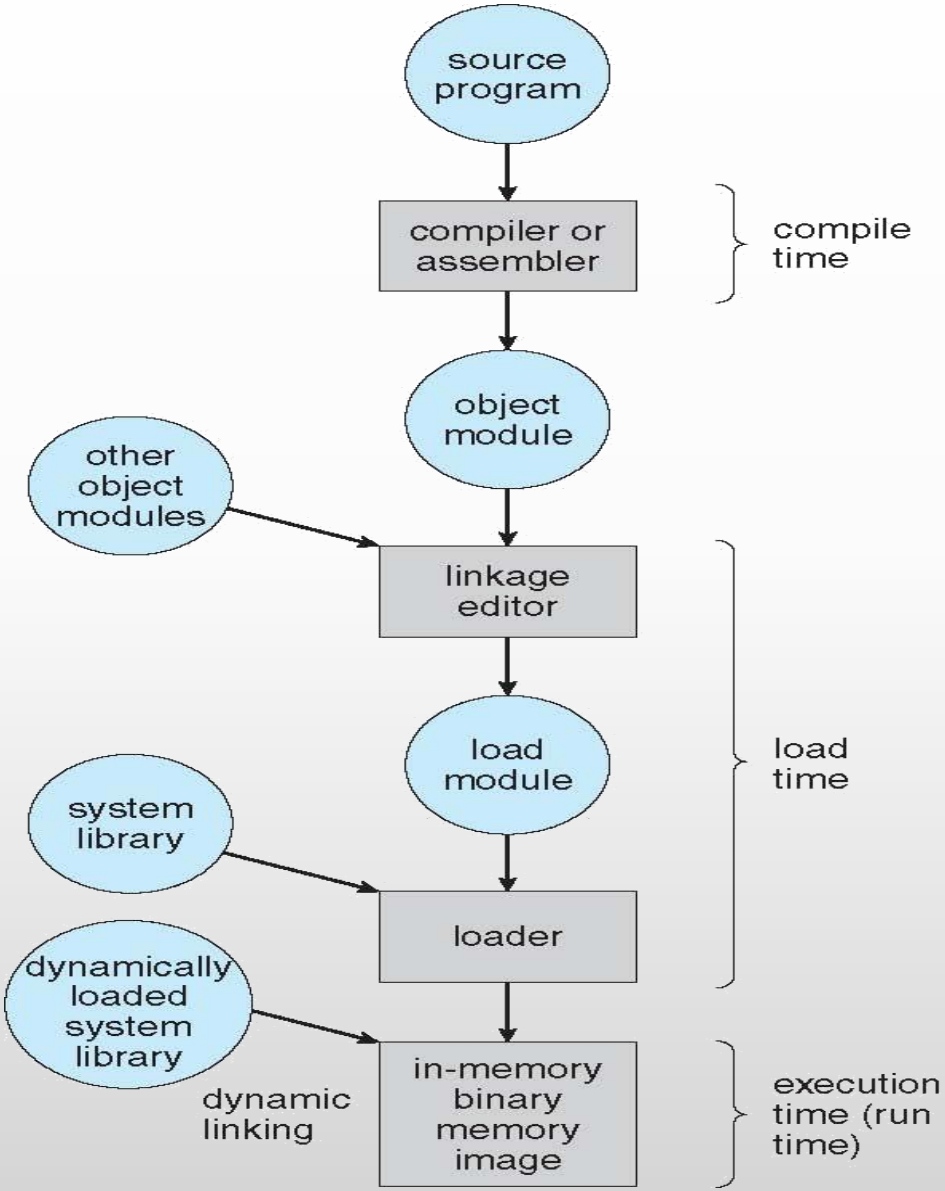
Binding of Instructions and Data to Memory

- 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**: If the memory location is not known at compile time, the compiler must generate relocatable code.
    - The loader knows the final location and binds addresses for that location
  - **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 (e.g., base and limit registers)

From Source to Executable



Multistep Processing of a User Program



Logical vs. Physical Address Space

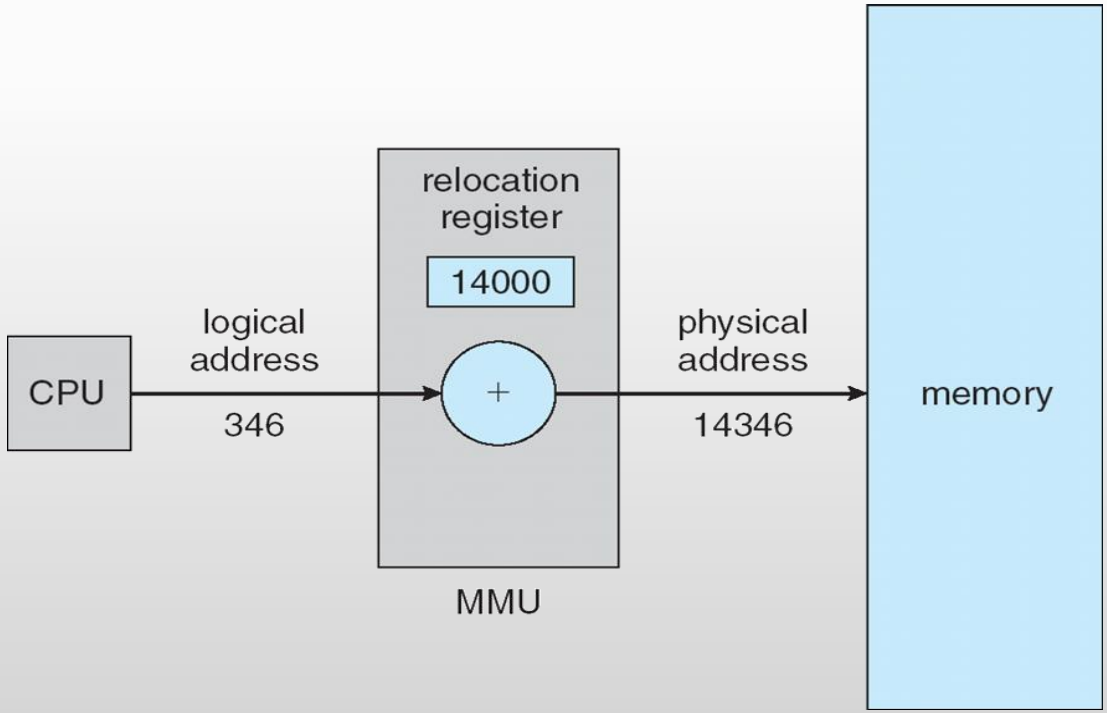
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Physical address** – The actual hardware memory address.
    - 32-bit CPU's physical address 0 ~ 232-1 (00000000 – FFFFFFFF)
    - 128MB's memory address 0 ~ 227-1 (00000000 – 07FFFFF)
  - **Logical address** – generated by the CPU; also referred to as **virtual address.** Each (relocatable) program assumes the starting location is always 0 and the memory space is much larger than the actual memory.
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in the execution-time address-binding scheme
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
- To start, consider a simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to a location in memory
  - Logical address bound to physical addresses

Dynamic relocation using a relocation register

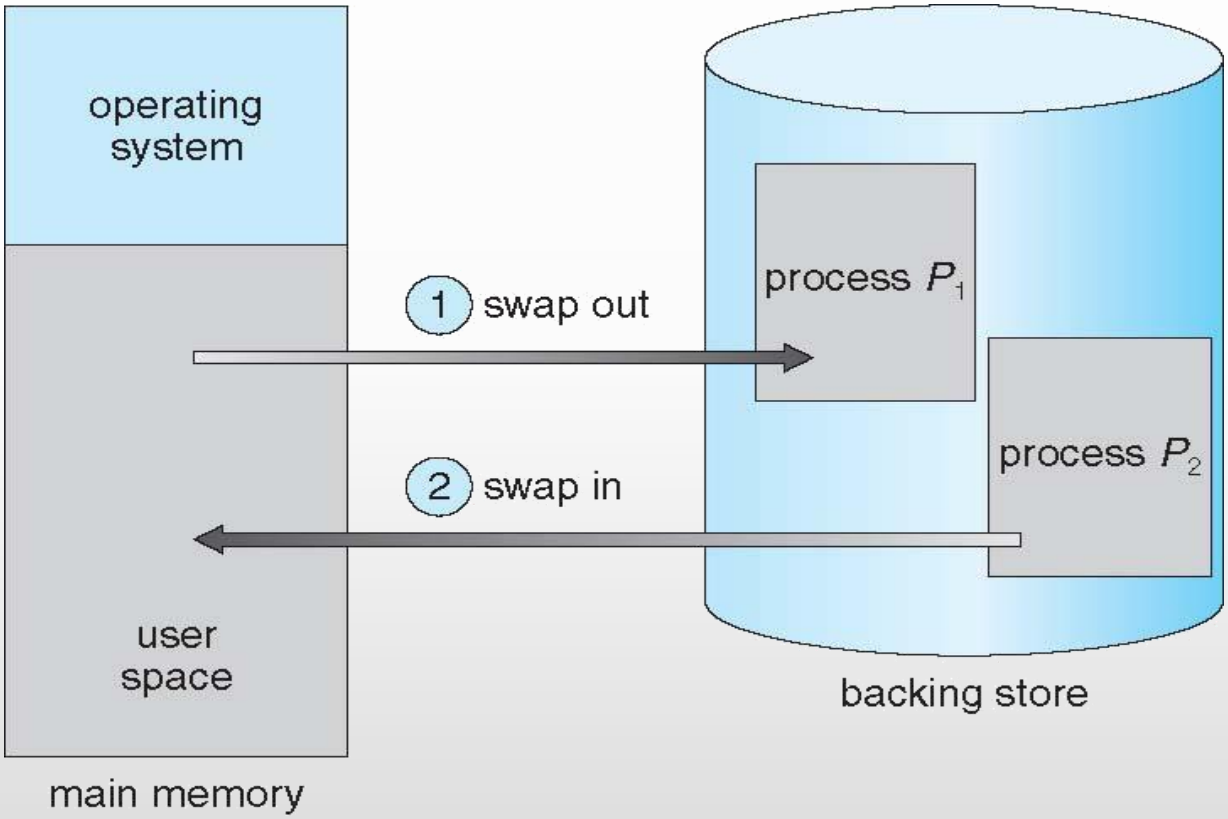
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading



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
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; the lower-priority process is swapped out so the 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



- When a process p1 is blocked so long (for I/O), it is swapped out to the backing store, (swap area in Unix.)
- When a process p2 is (served by I/O and) back to a ready queue, it is swapped in the memory.
- Use the Unix top command to see which processes are swapped out.

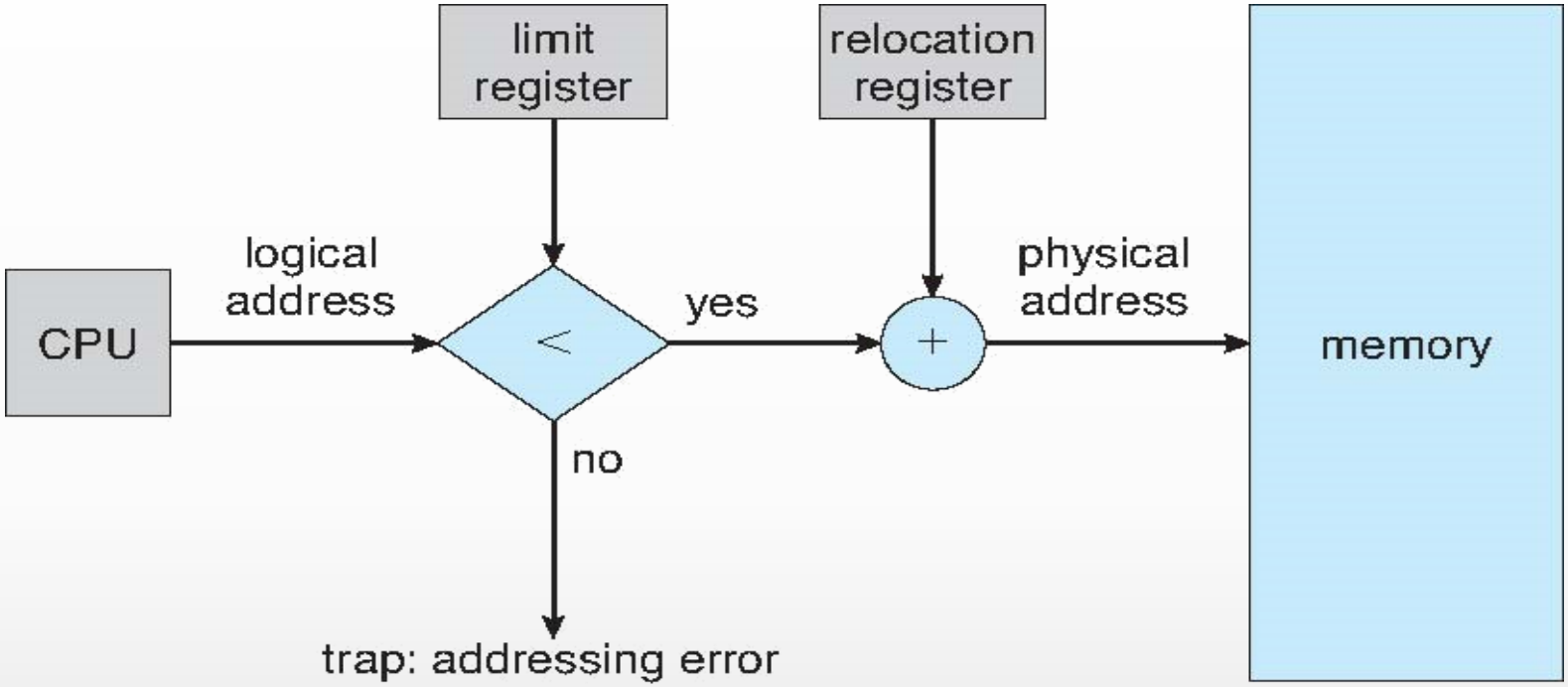
Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resources, must allocate efficiently
- Contiguous allocation is one early method

- The main memory is usually divided into two parts:
  - one for the operating system, usually held in low memory with an **interrupt vector**
  - one for the user processes; held in high memory
  - Each process contained in a single contiguous section of memory

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of the smallest physical address
  - Limit register contains a range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical addresses *dynamically*
  - Can then allow actions such as kernel code being **transient** and kernel changing size

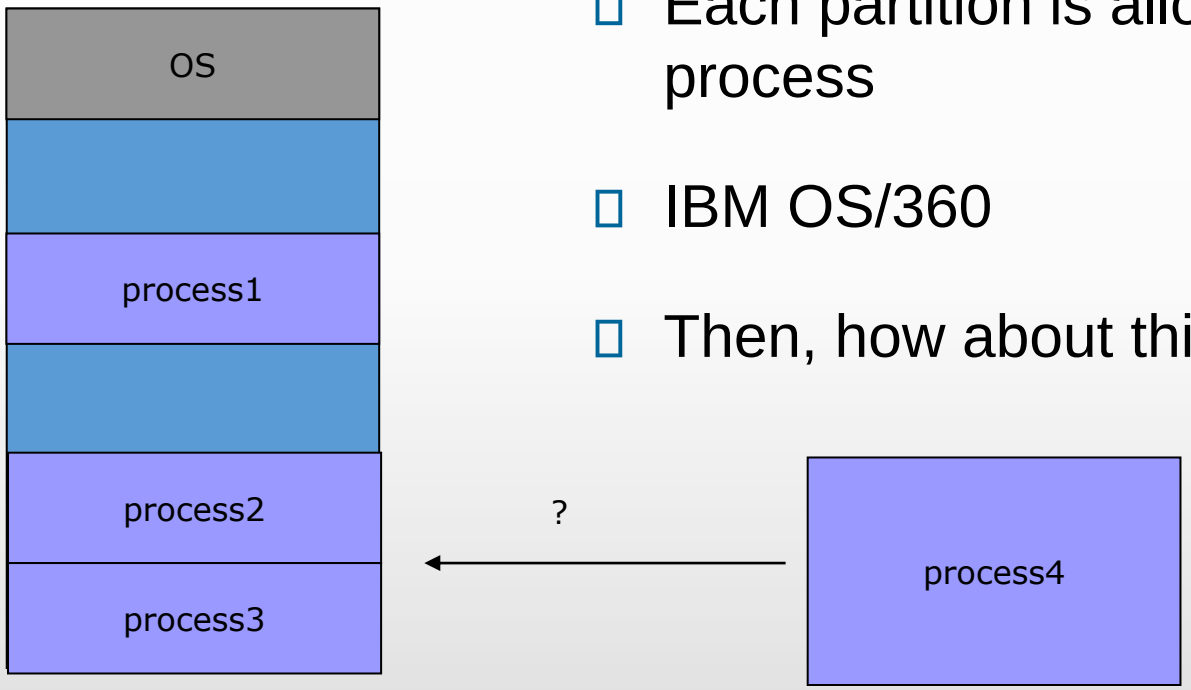
Hardware Support for Relocation and Limit Registers



- For each process
  - Logical space is mapped to a contiguous portion of physical space
  - A relocation and a limit register are prepared
- Relocation register = the value of the smallest physical address
- Limit register = the range of logical address space

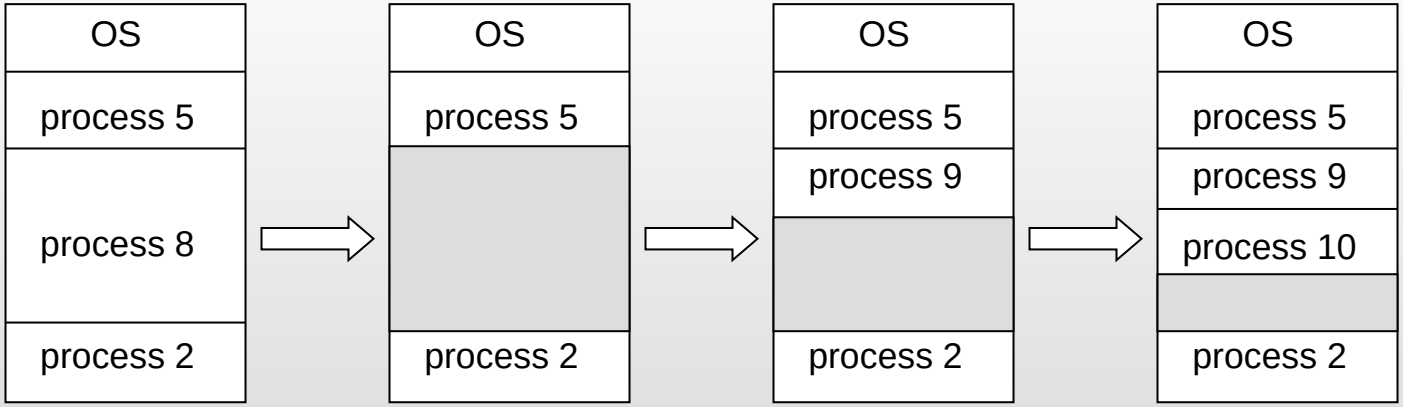
Memory Allocation (Fixed-sized partition)

- Memory is divided to fixed-sized partitions
- Each partition is allocated to a process
- IBM OS/360
- Then, how about this process?



Memory Allocation (Variable-sized partition)

- Multiple-partition allocation, Primarily used in a Batch environment
  - The OS keeps a table indicating which parts of memory are available and which are occupied
  - Initially all memory is available for the user processes and is considered as one large block of available memory, "**HOLE**". When a process arrives, it is allocated memory from a hole large enough to accommodate it. If we find one, we allocate only as much memory as is needed.
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Whenever one of the running processes, (p8) is terminated. Find a ready process whose size is the best fit for the hole, (p9). Allocate it from the top of the hole. If there is still an available hole, repeat the above (for p10).

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 (Fastest search)
  - Advantage: fastest algorithm because it searches as little as possible.
  - Disadvantage: remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for a larger memory requirement cannot be accomplished.
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search the entire list, unless ordered by size
  - Produces the smallest leftover hole (Best memory usage)
    - Advantage: memory utilization is much better than the first fit as it searches the smallest free partition first available.
    - Disadvantage: it is slower and may even tend to fill up memory with tiny useless holes.
- **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
  - Produces the largest leftover hole (that could be used effectively later.)
    - Advantage: reduces the rate of production of small gaps.
    - Disadvantage: if a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

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

Dynamic Storage-Allocation Problem

let us assume the jobs and the memory requirements as the following:

- Job 1 90k
- Job 2 20k
- Job 3 50k
- Job 4 200k

Let the free-pace memory allocation blocks be:

- Block 1 50k
- Block 2 100k
- Block 3 90k
- Block 4 200k
- Block 5 50k

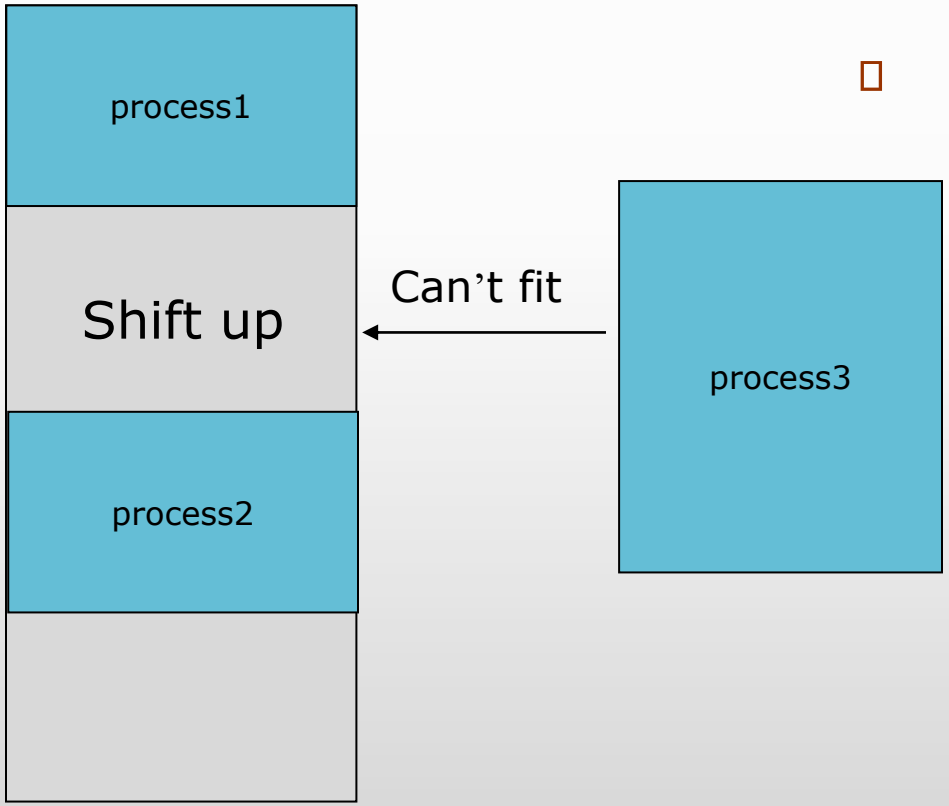
- 50k 100k 90k 200k 50k

50	Job2=30
100	Job1=10
90	Job3=40
200	Job4=0
50	

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
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation

External Fragmentation



- Problem
  - 50-percent rule (for first fit): total memory space exists to satisfy a request, but it is not contiguous.
- Solution
  - **Compaction**: shuffle the memory contents to place all free memory together in one large block
    - Relocatable code
    - Expensive
  - **Paging**: Allow non-contiguous logical-to-phyiscal space mapping.

Internal Fragmentation

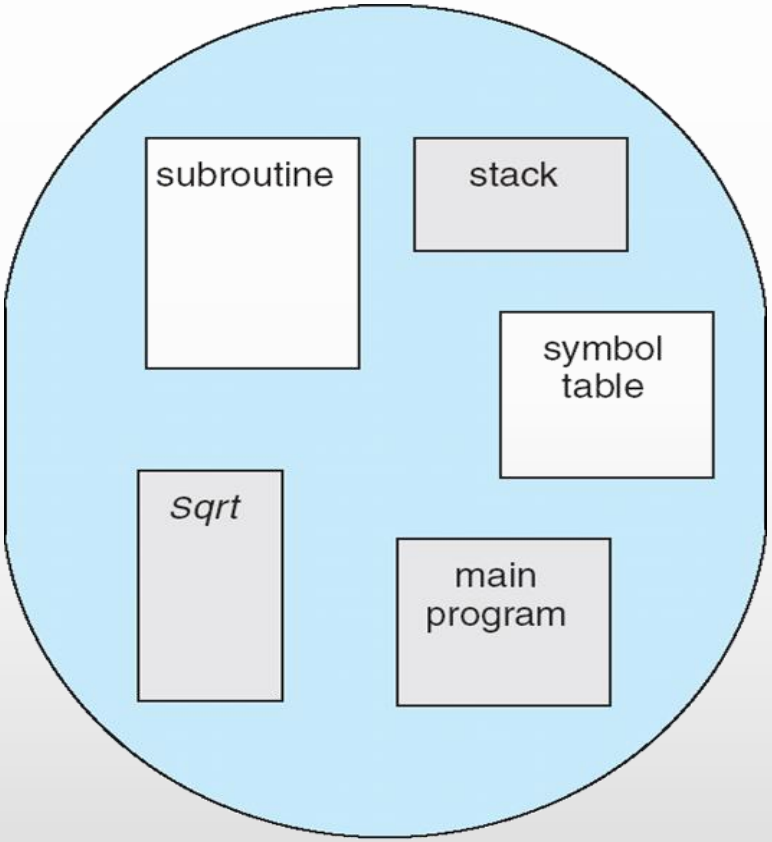
- With the scheme of breaking the physical memory into fixed-sized blocks, and allocating memory in units of block size, results from **internal fragmentation**.

- With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is internal fragmentation.

Segmentation

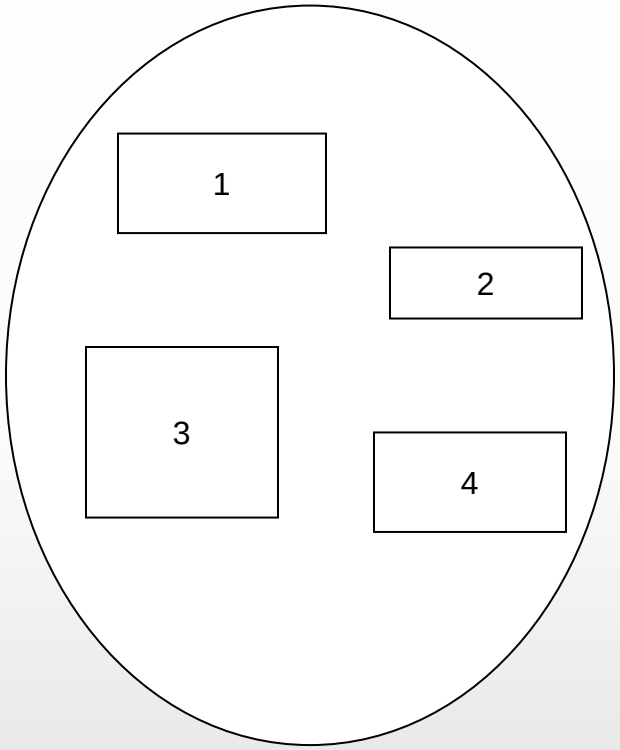
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

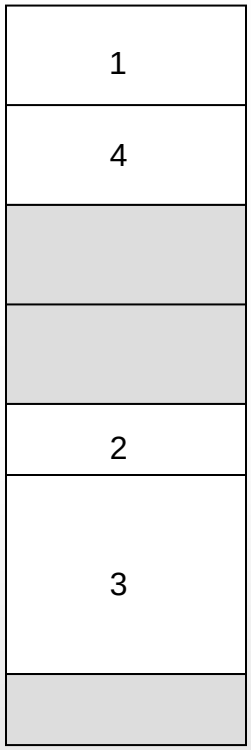


User's View of a Program

Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

- The logical address consists of two tuples:

  < segment-number, offset >

- **Segment table** – maps two-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;

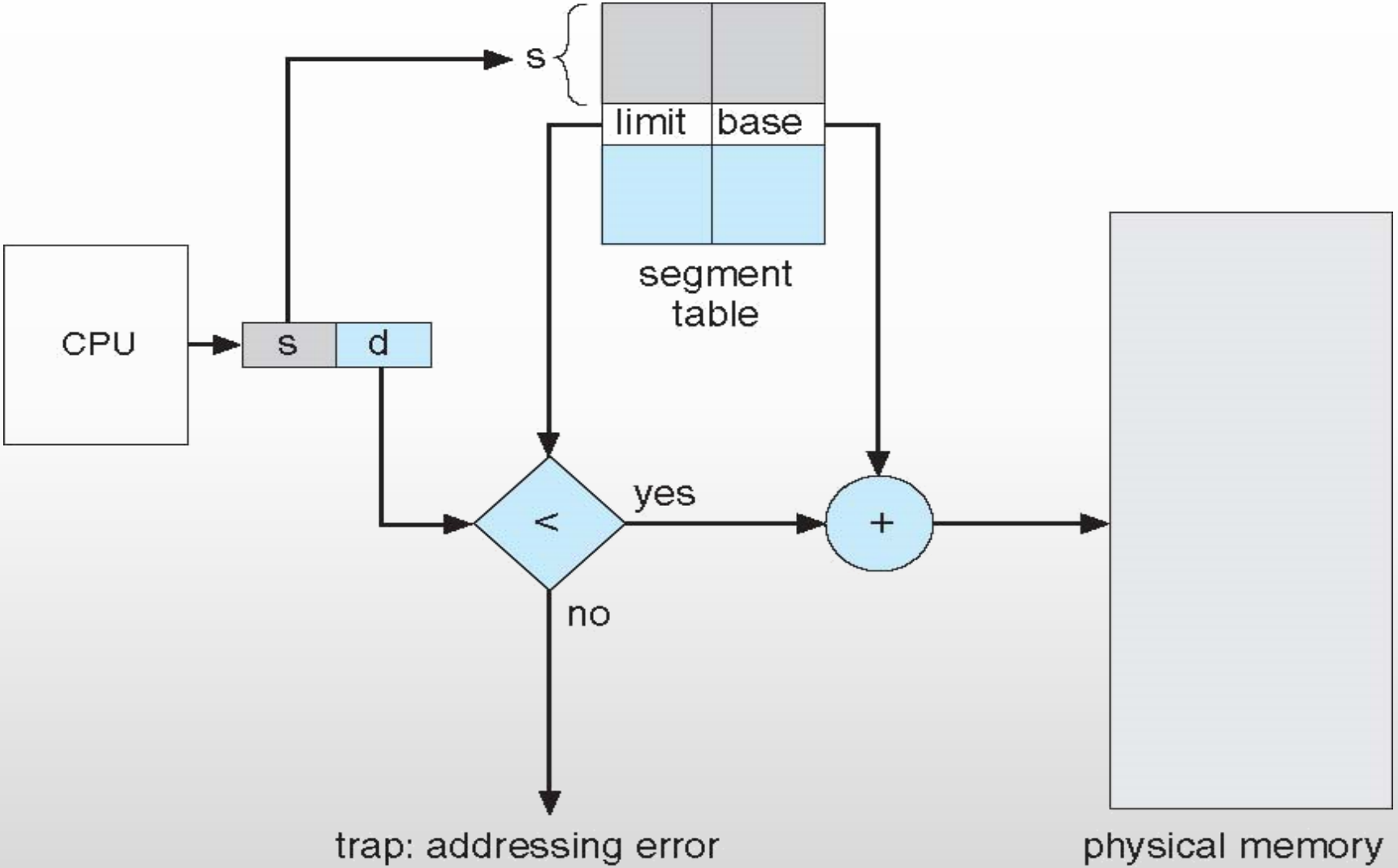
  segment number ***s*** is legal if ***s*** < **STLR**

Segmentation Architecture (Cont.)

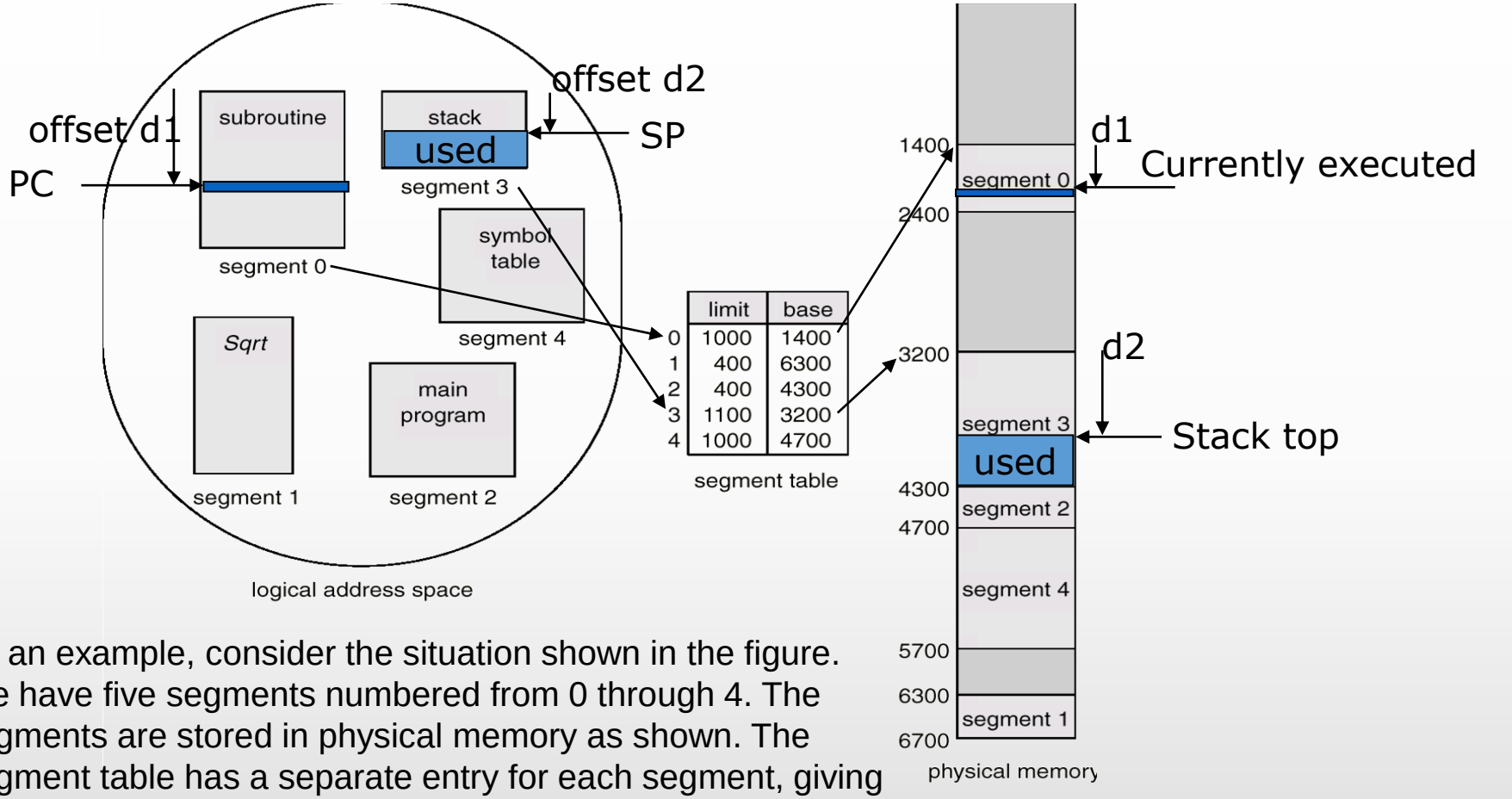
- Protection
  - With each entry in the segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges

- Protection bits associated with segments;
- Code sharing occurs at the segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segmentation Hardware



Segmentation Example



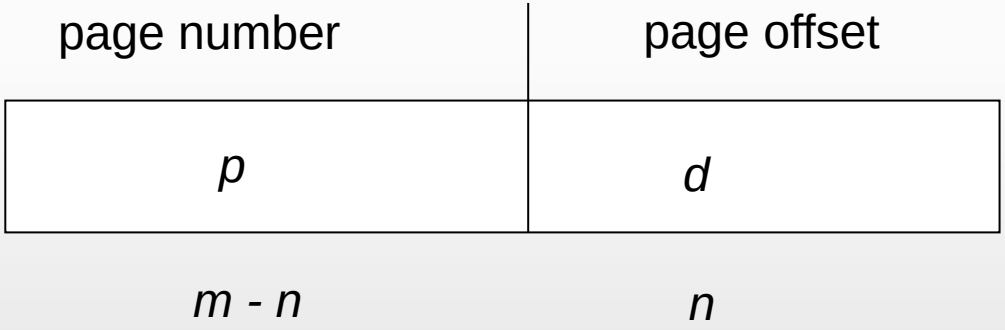
As an example, consider the situation shown in the figure. We have five segments numbered from 0 through 4. The segments are stored in physical memory as shown. The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 + 53 = 4353. A reference to segment 3, byte 852, is mapped to 3200 (the base of segment 3) + 852 = 4052. A reference to byte 1222 of segment 0 would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Paging

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

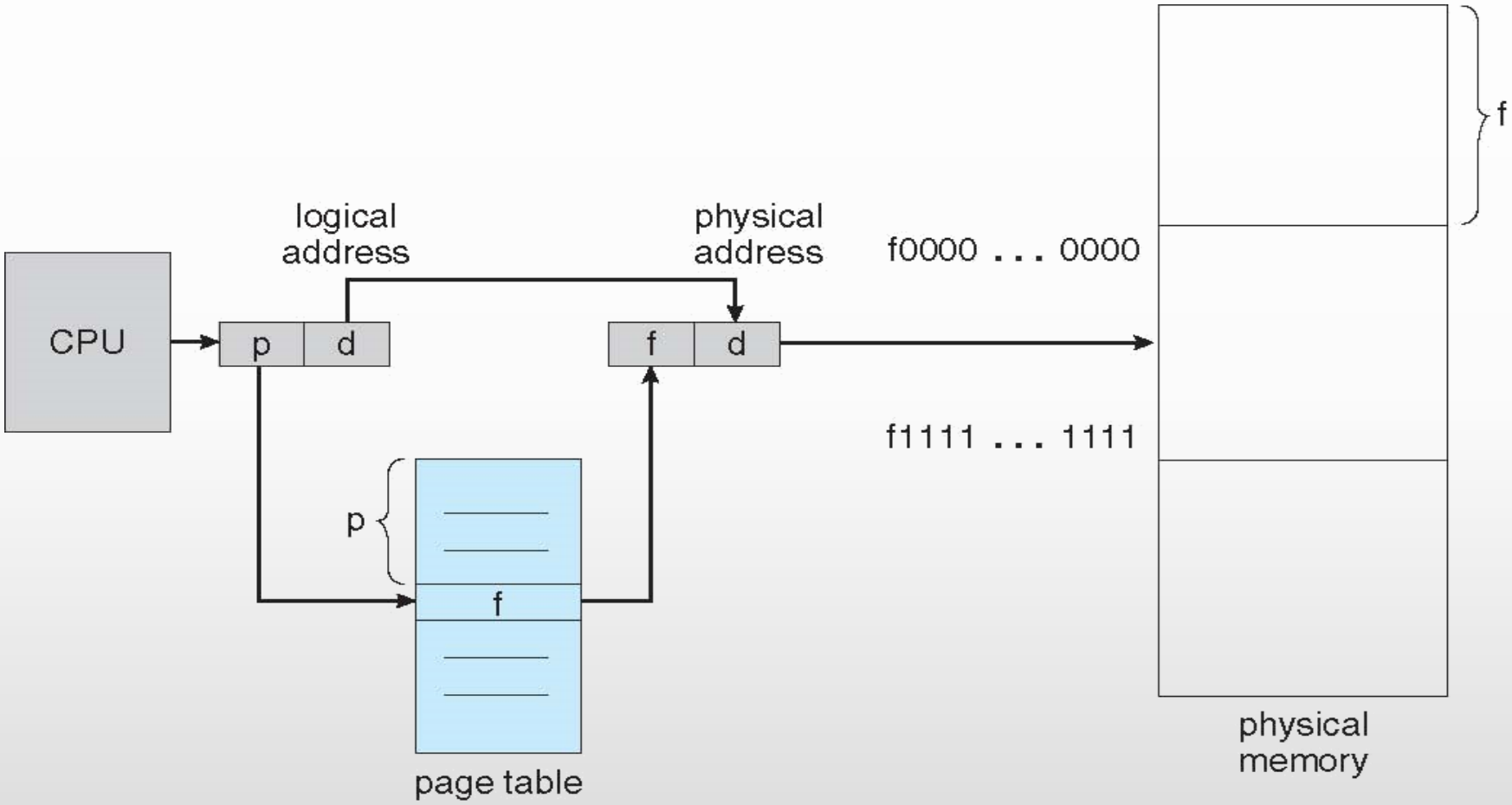
Address Translation Scheme

- 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 a base address to define the physical memory address that is sent to the memory unit

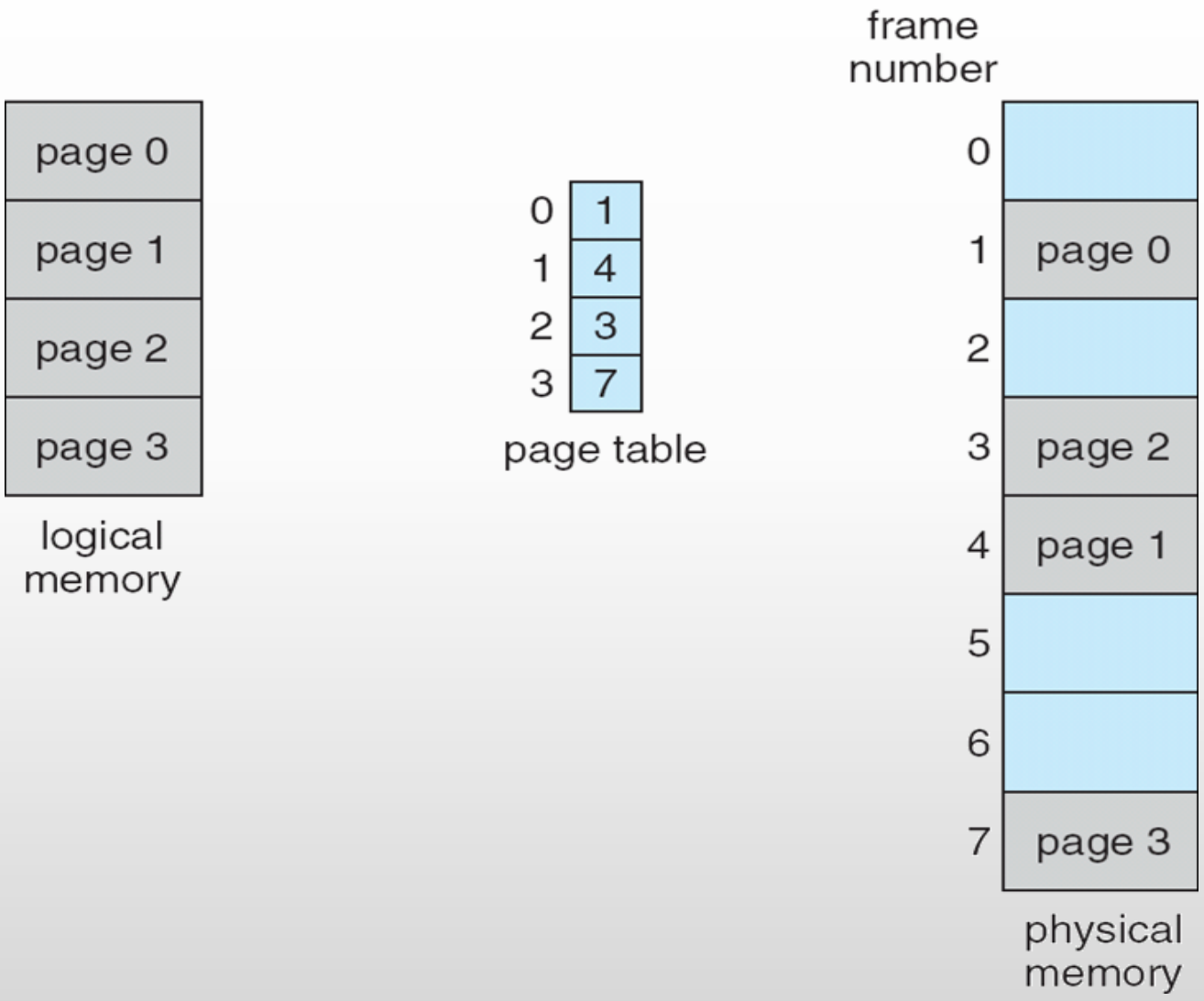


- For given logical address space 2^m and page size 2^n

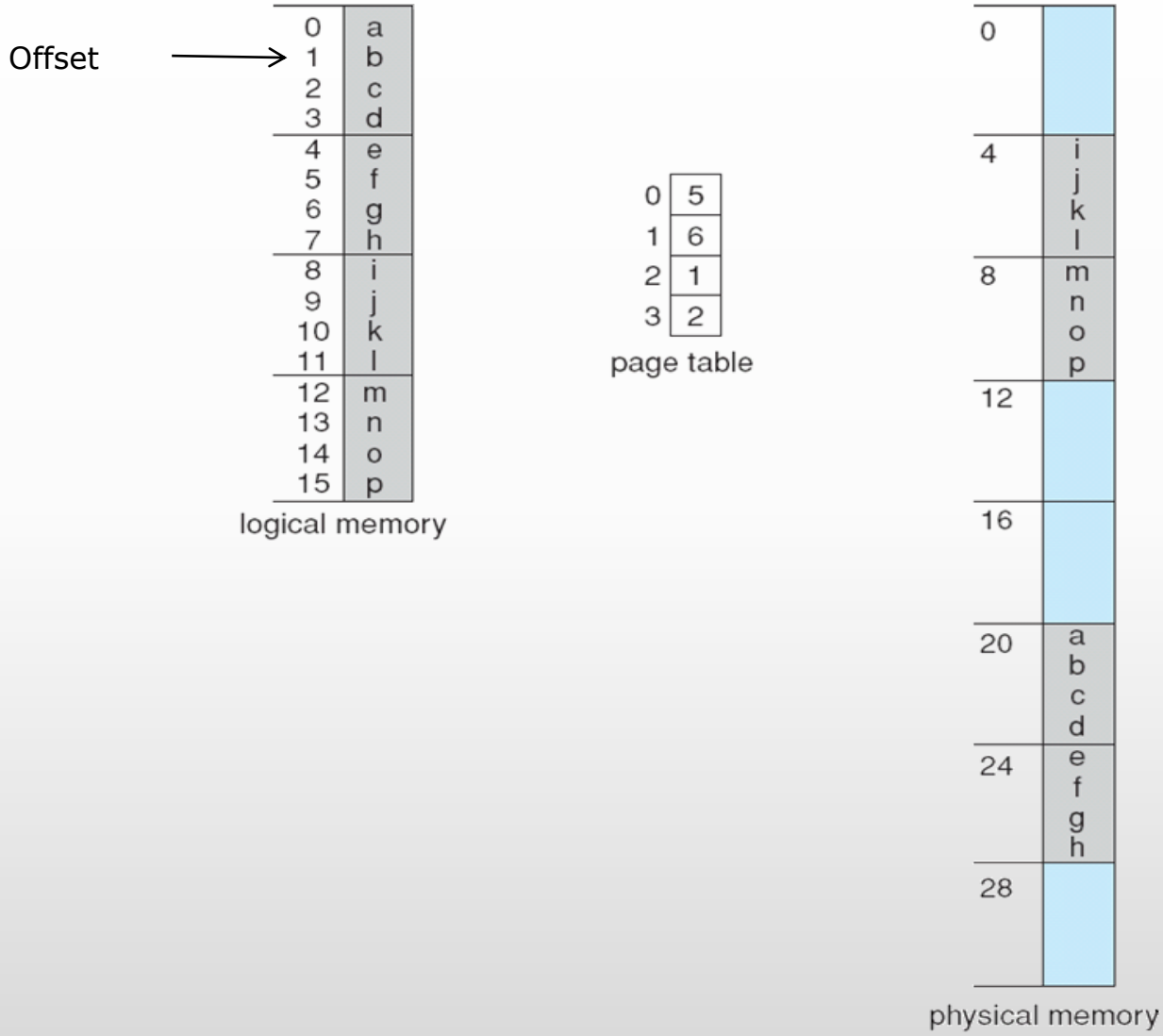
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



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

(Page 330)

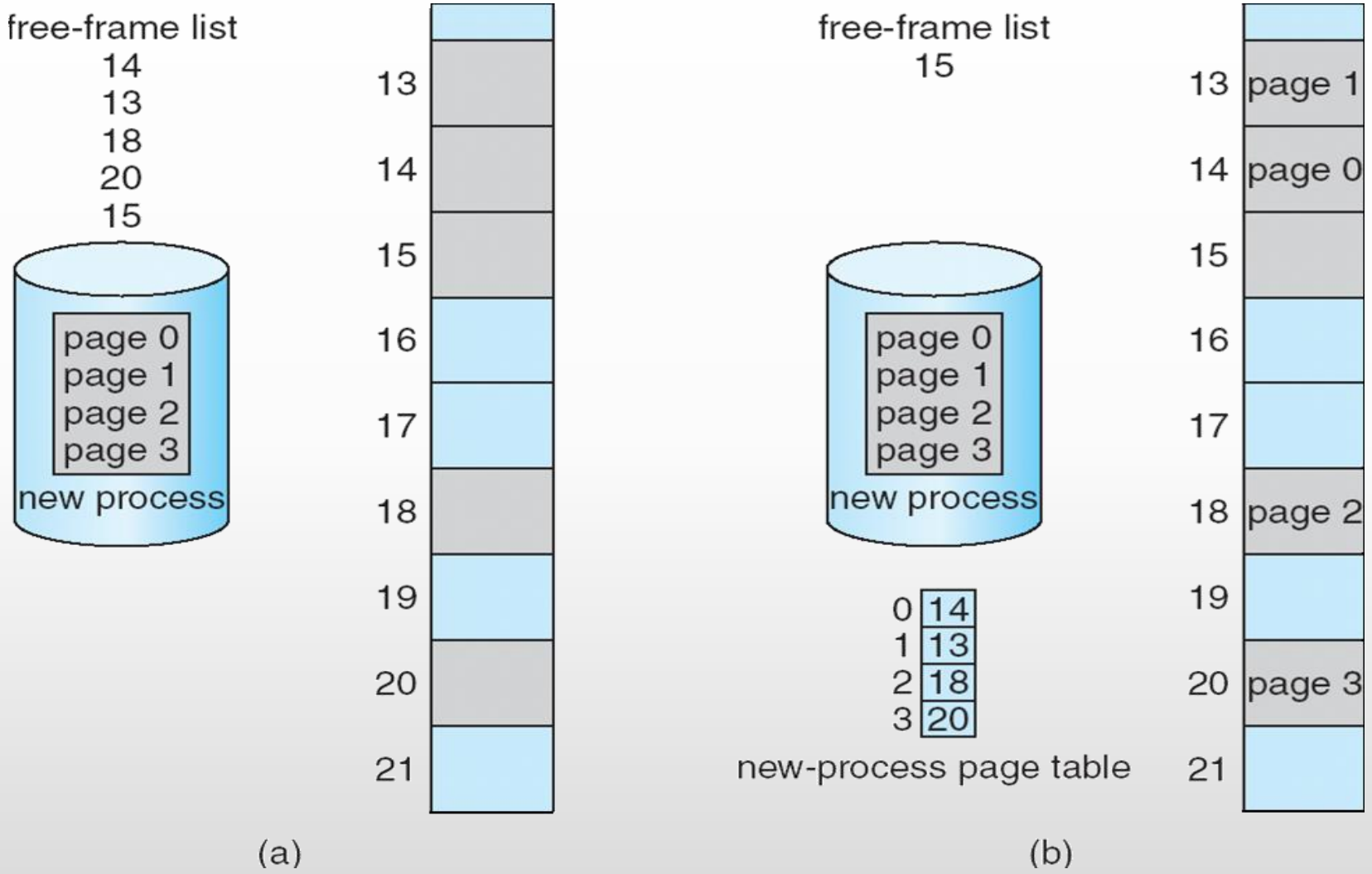
Paging (Cont.)

- 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
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Solaris supports two-page sizes – 8 KB and 4 MB

Free Frames

- l When a process arrives in the system to be executed, its size, expressed in pages, is examined.
- Each page of the process needs one frame.
- If the process requires n pages, at least n frames must be available.
- A list of free frames is maintained.
- When the first process is loaded into a frame its frame number is put into the page table.

Free Frames



Before allocation

After allocation

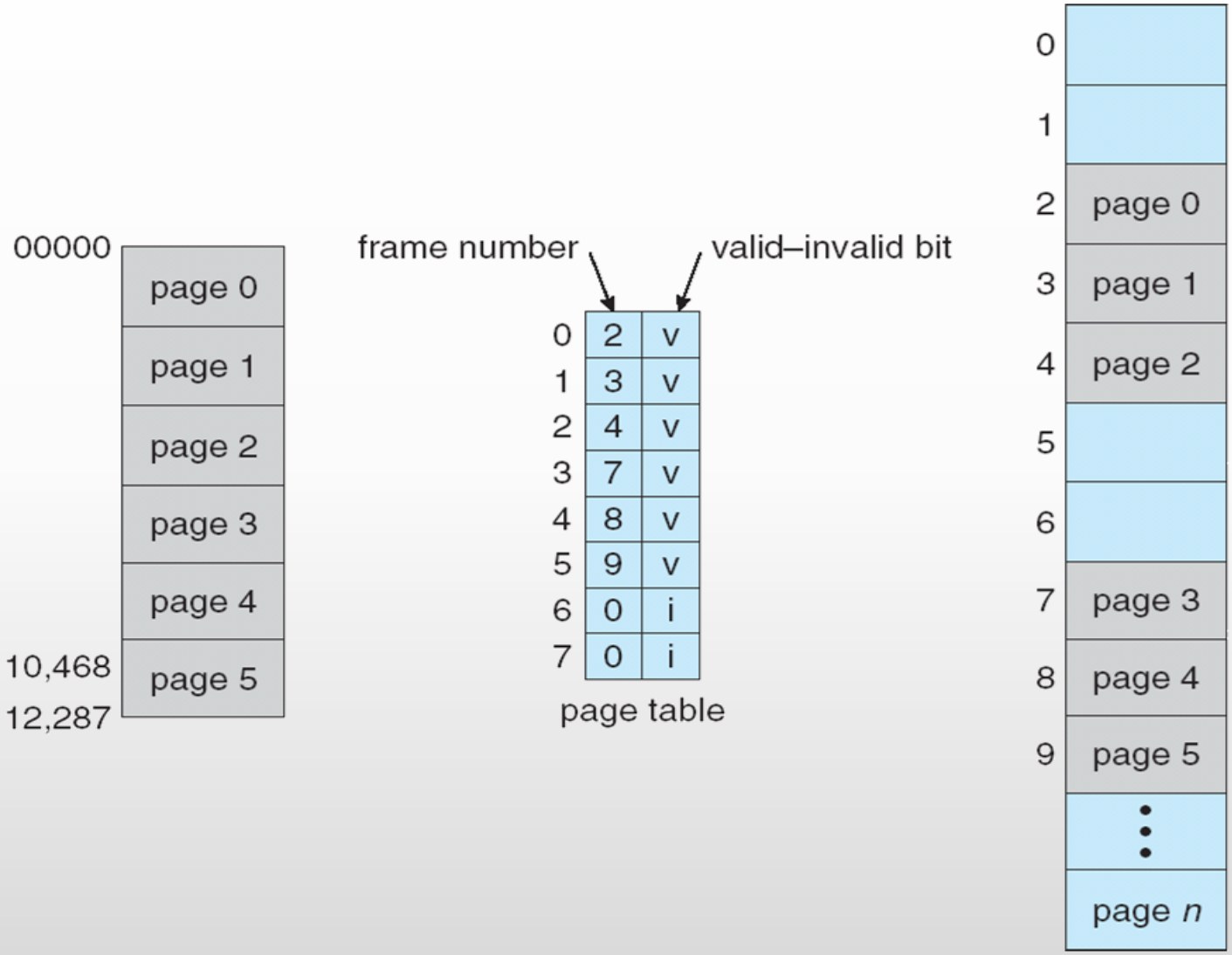
Memory Protection

- Memory protection is implemented by associating the 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” indicates that the associated page is in the process of logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process of logical address space
  - Or use **page-table length register** (**PTLR**)

- Any violations result in a trap to the kernel

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

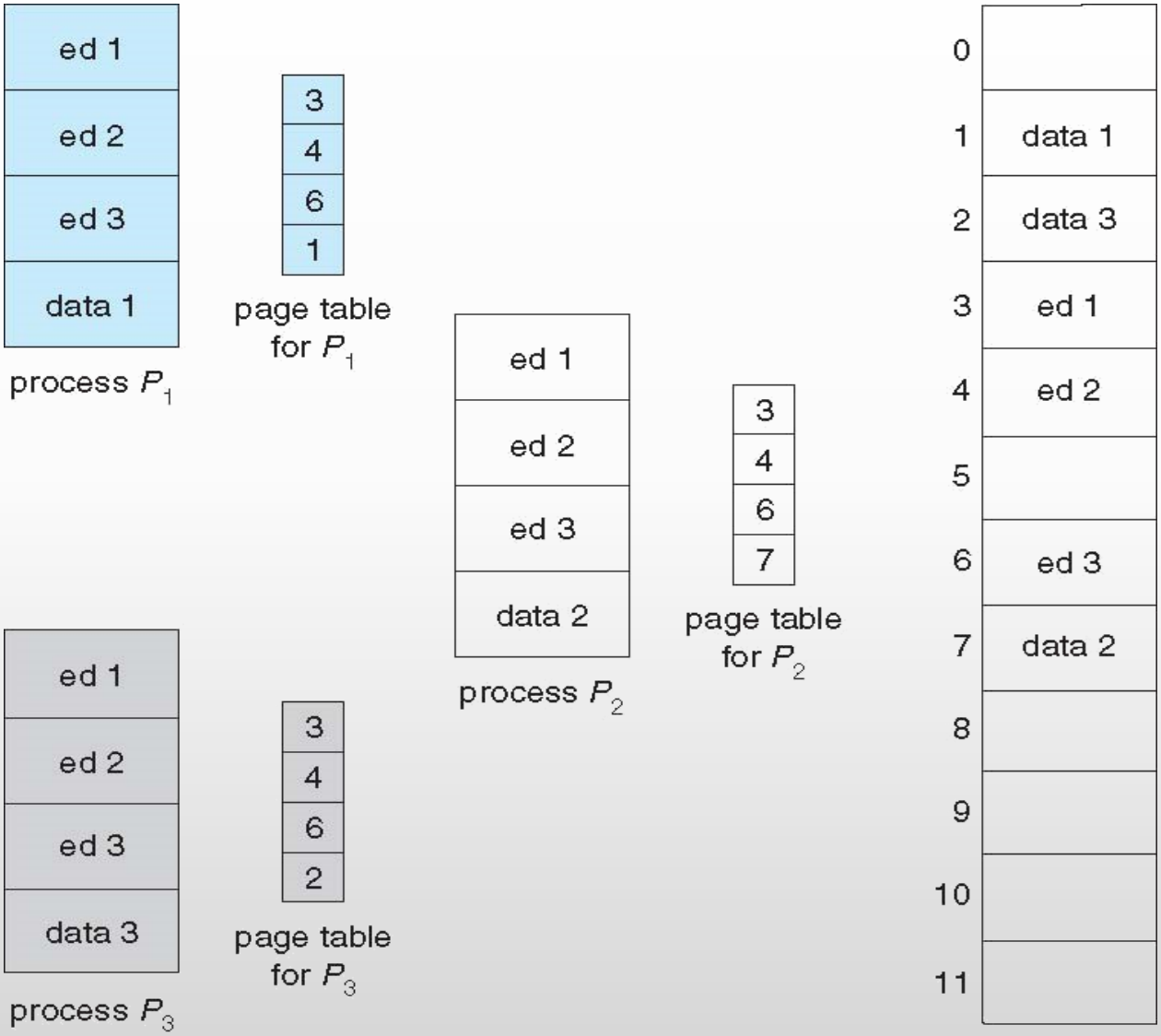


Shared Pages

- .**Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for inter-process communication if sharing of read-write pages is allowed

- **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



Question & Discussion

Task Assign

Thank You

kohinoor_cse@lus.ac.bd