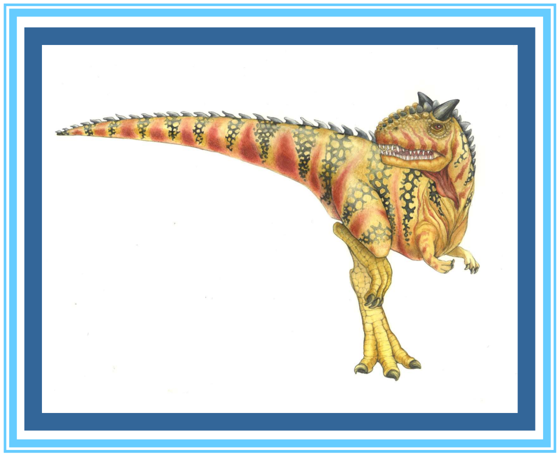
Chapter 8: Main Memory



Chapter 8: Memory Management

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging

Objectives

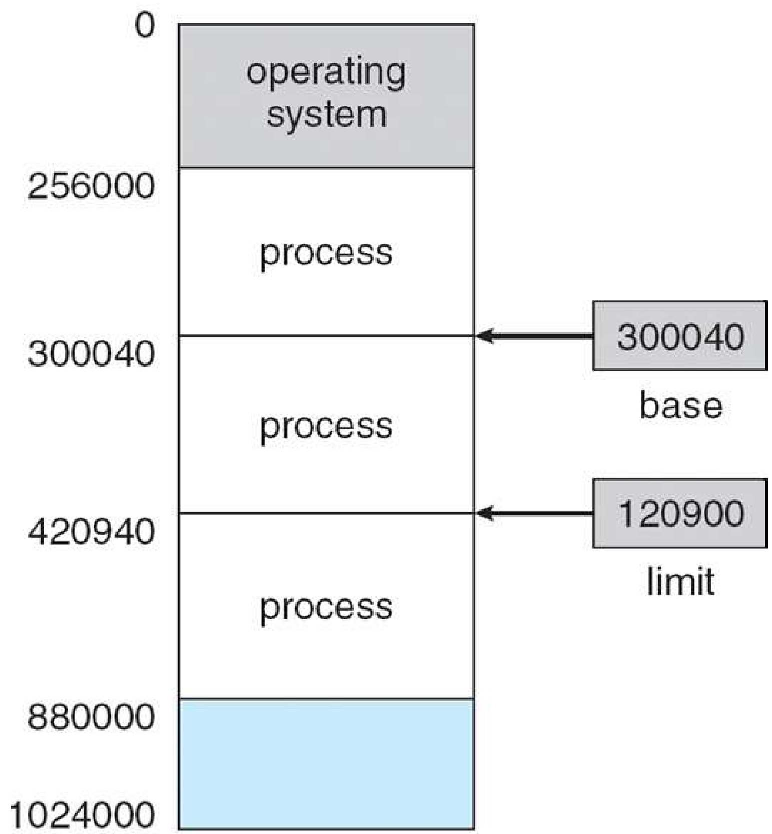
- To provide a detailed description of various ways of **organizing memory hardware**
- To discuss various memory-management techniques, **including paging and segmentation**

Background

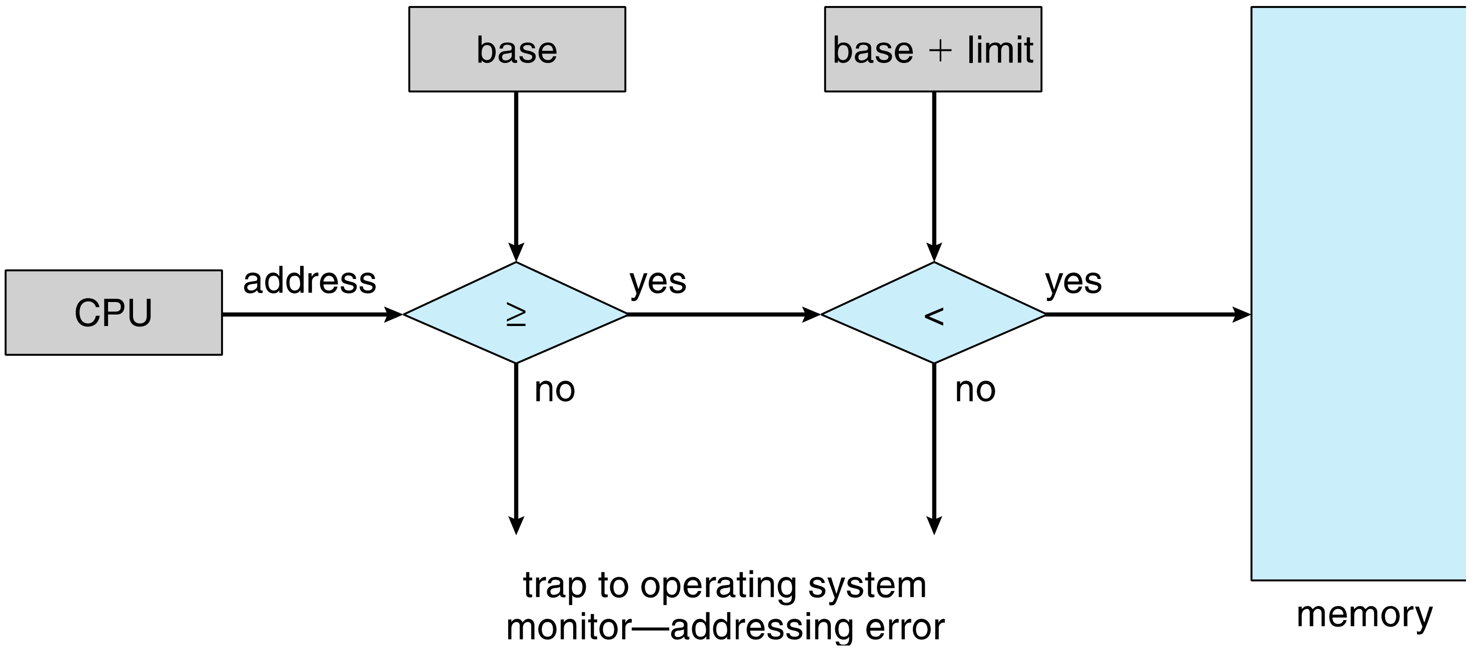
- **Program must be brought (from disk) into memory** and placed within a process for it to be run
- **Main memory** and **registers** are only storage **CPU can access directly**
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- i.e memory unit does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data).
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall → a cache is needed**
- **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 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



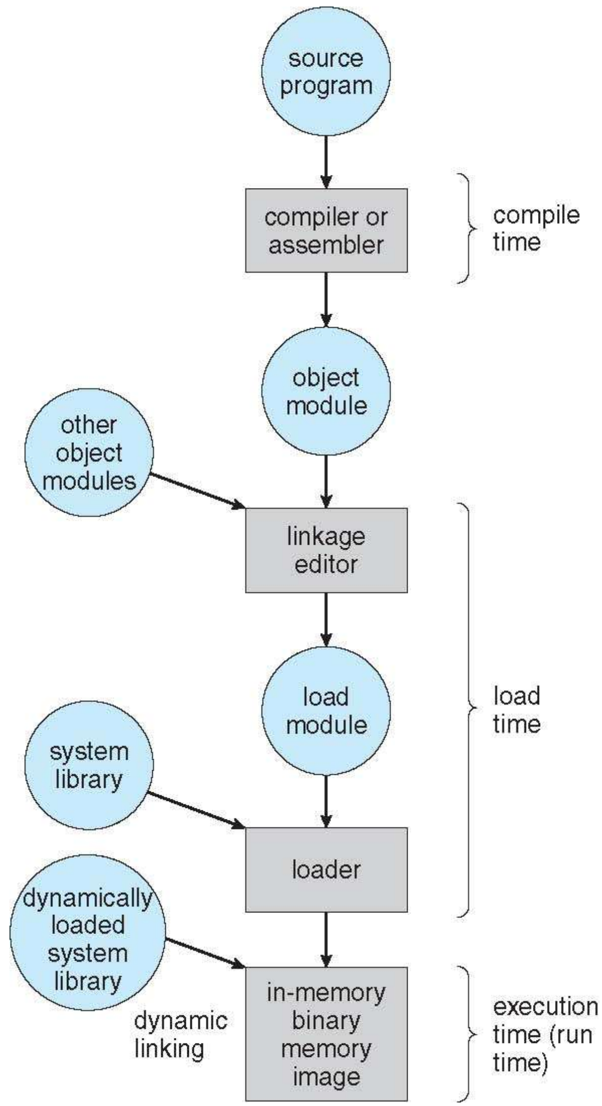
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue (contains the processes on the disk waiting to be executed)**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, 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 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**: 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 for address maps (e.g., base and limit registers)

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
  - **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
- **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

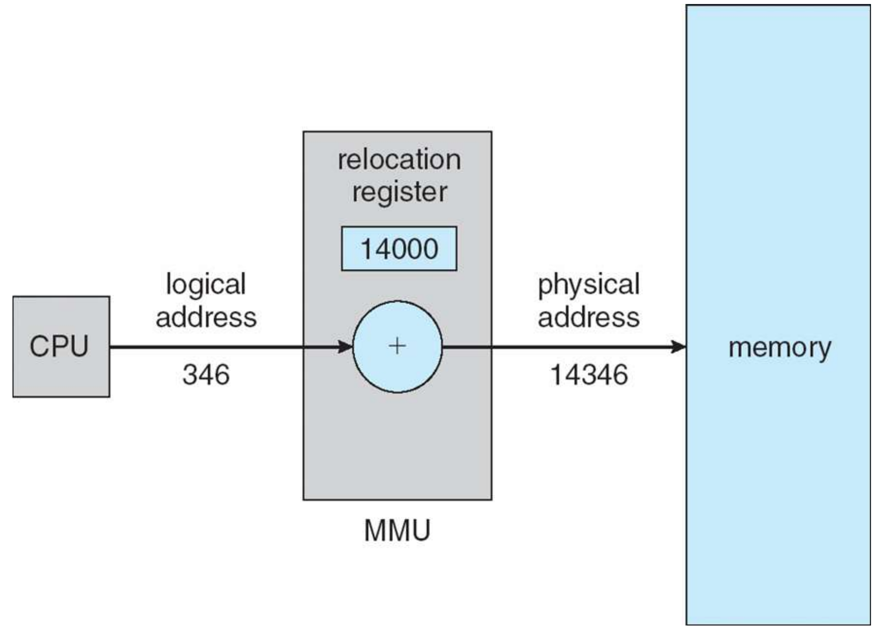
Memory-Management Unit (MMU)

- Hardware device that at run time **maps virtual to physical address**
- Many methods possible, covered in the rest of this chapter
- To start, consider 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
- For example, if the base is at14000, then an attempt by the user to address location 0 is dynamically relocated to location14000; an access to location 346 is mapped to location 14346.

- 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

Dynamic relocation using a relocation register

- Routine is not loaded until it is called (dynamic loading)
- 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



Dynamic Linking

- Dynamically linked libraries are **system libraries (such as language subroutine libraries)** that are linked to user programs when the programs are run
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine or to load the library if the routine is not present
- Stub replaces itself with the address of the routine, and executes the routine
- Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking
- Under this scheme, all processes that use a language library execute only one copy of the library code **(shared libraries)**.

- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- A library may be **replaced by a new version**, and all programs that reference the library will automatically use the new version.
- Consider applicability to patching system libraries
  - Versioning may be needed

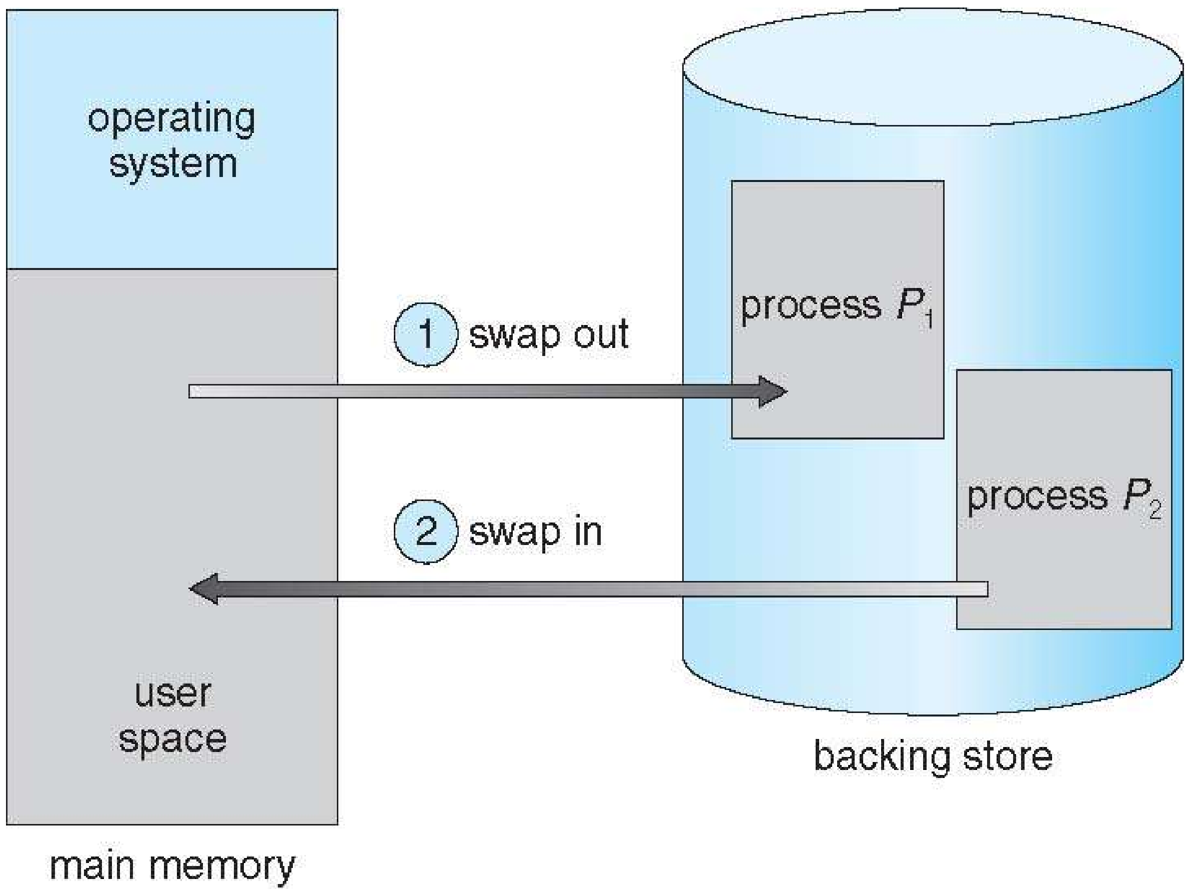
Swapping

- 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** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; 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

Swapping (Cont.)

- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
  - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
  - Swapping normally disabled
  - Started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold

Schematic View of Swapping



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
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2000 ms
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- If we have a computer system with 4 GB of main memory and a resident operating system taking 1 GB, the maximum size of the user process is 3 GB.
- However, many user processes may be much smaller than this—say, 100 MB.
- A 100-MB process could be swapped out in 2 seconds, compared with the 60 seconds required for swapping 3 GB.

- Clearly, it would be useful to know exactly how much memory a user process is using, not simply how much it might be using.
- Then we would need to swap only what is actually used, reducing swap time. For this method to be effective, the user must keep the system informed of any changes in memory requirements.
- Thus, a process with dynamic memory requirements will need to issue system calls (request memory() and release memory()) to inform the operating system of its changing memory needs.

Context Switch Time and Swapping (Cont.)

- Other constraints as well on swapping
- If we want to swap a process, we must be sure that it is completely idle
- A process may be waiting for an I/O operation when we want to swap that process to free up memory. However, if the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped. Assume that the I/O operation is queued because the device is busy. If we were to swap out process P1 and swap in process P2, the I/O operation might then attempt to use memory that now belongs to process P2.
- Two solutions
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - Or always transfer I/O to kernel space, then to I/O device
    - Known as **double buffering**, adds overhead

- Standard swapping not used in modern operating systems
  - But there is a modified version that is common
    - **Swap only when free memory extremely low**

Swapping on Mobile Systems

- Not typically supported (mobile systems typically do not support swapping in any form)
  - Flash memory based
    - Small amount of space (not large enough for swaping)
    - Limited number of write cycles
    - Poor throughput between flash memory and CPU on mobile platform
- Instead use other methods to free memory if low
  - iOS ***asks*** apps to voluntarily relinquish allocated memory
    - Read-only data thrown out and reloaded from flash if needed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart
  - Both OSes support paging as discussed below

Contiguous Allocation

- Main memory must support both OS and user processes
- Limited resource, must allocate efficiently
- Contiguous allocation is one early method
- Main memory usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory

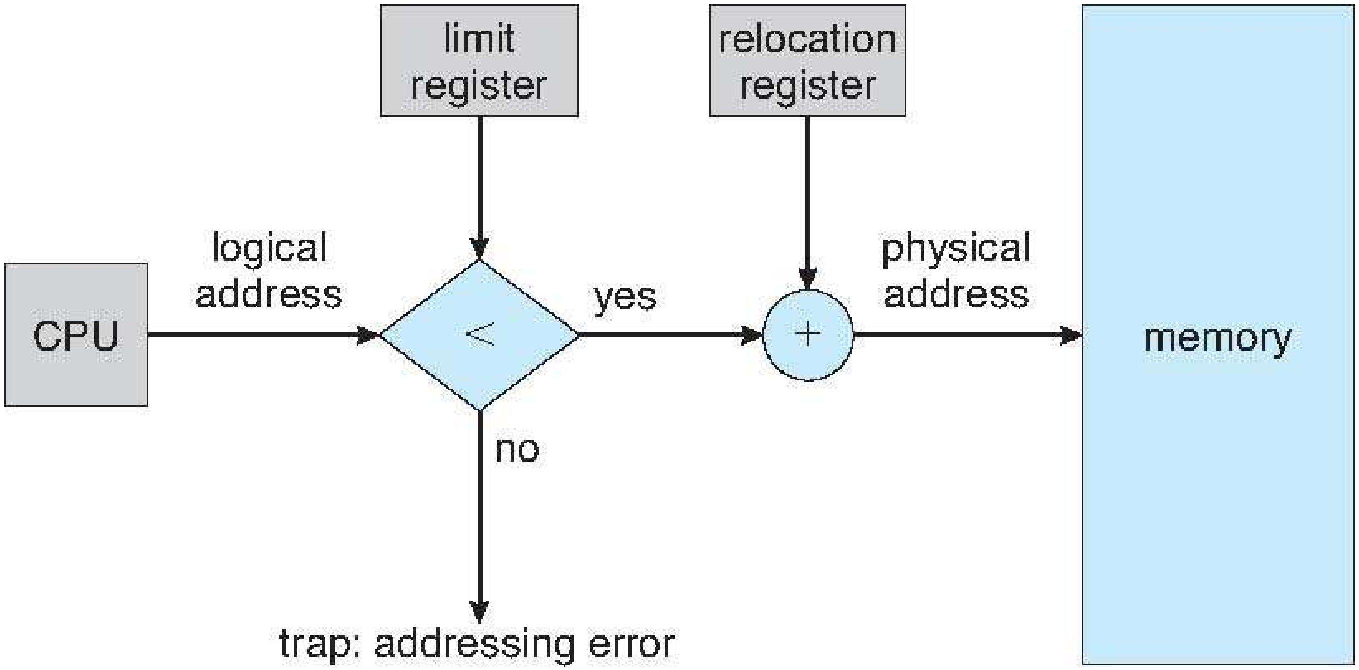
Contiguous Allocation (Cont.)

- 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*

  - Can then allow actions such as kernel code being **transient** and kernel changing size

- This allows the operating system's size to change dynamically.
- For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we donot want to keep the code and data in memory,as we might be able to use that space for other purposes.
- Such code is sometimes called **transient operating-system code**;
- it comes and goes as needed

Hardware Support for Relocation and Limit Registers

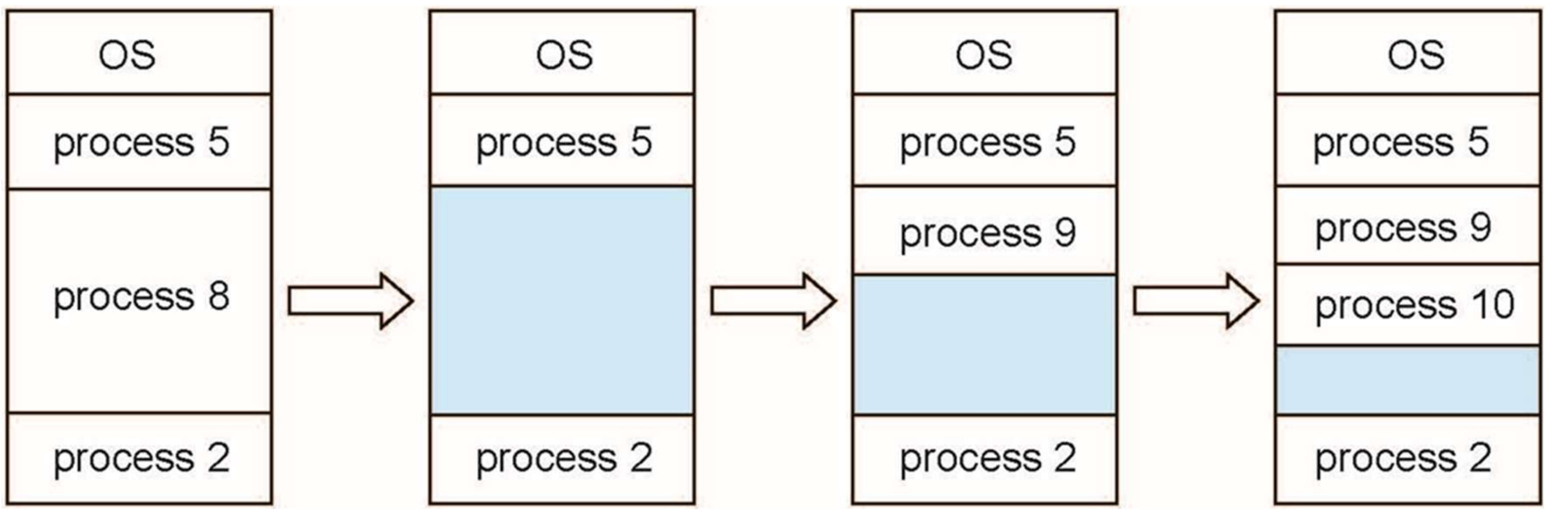


Memory Allocation

- One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**.
- Each partition may contain exactly one process.
- Thus, the **degree of multiprogramming is bound by** the number of partitions.
- In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- no longer in use

Multiple-partition allocation

- **Variable-partition**
  - Initially, all memory is available for user processes and is considered **one large block of available memory**, a hole.
  - Eventually, as you will see, memory contains a set of holes of various sizes.
  - **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**
  - Process exiting frees its partition, **adjacent free partitions combined**
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (holes)


OS
process 5
process 8
process 2
OS
process 5
process 2
OS
process 5
process 9
process 2
OS
process 5
process 9
process 10
process 2

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

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous **(broken into little pieces)**
- **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 0.5 *N* blocks lost to fragmentation
  - i.e. 1/3 may be unusable -> this is known as **50-percent rule**

Fragmentation (Cont.)

- 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**
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

- Another possible solution to the external-fragmentation problem is to **permit the logical address space of the processes to be noncontiguous**,
- thus allowing a process to be **allocated physical memory wherever such memory is available**.
- Two complementary techniques achieve this solution: **segmentation and paging**.

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

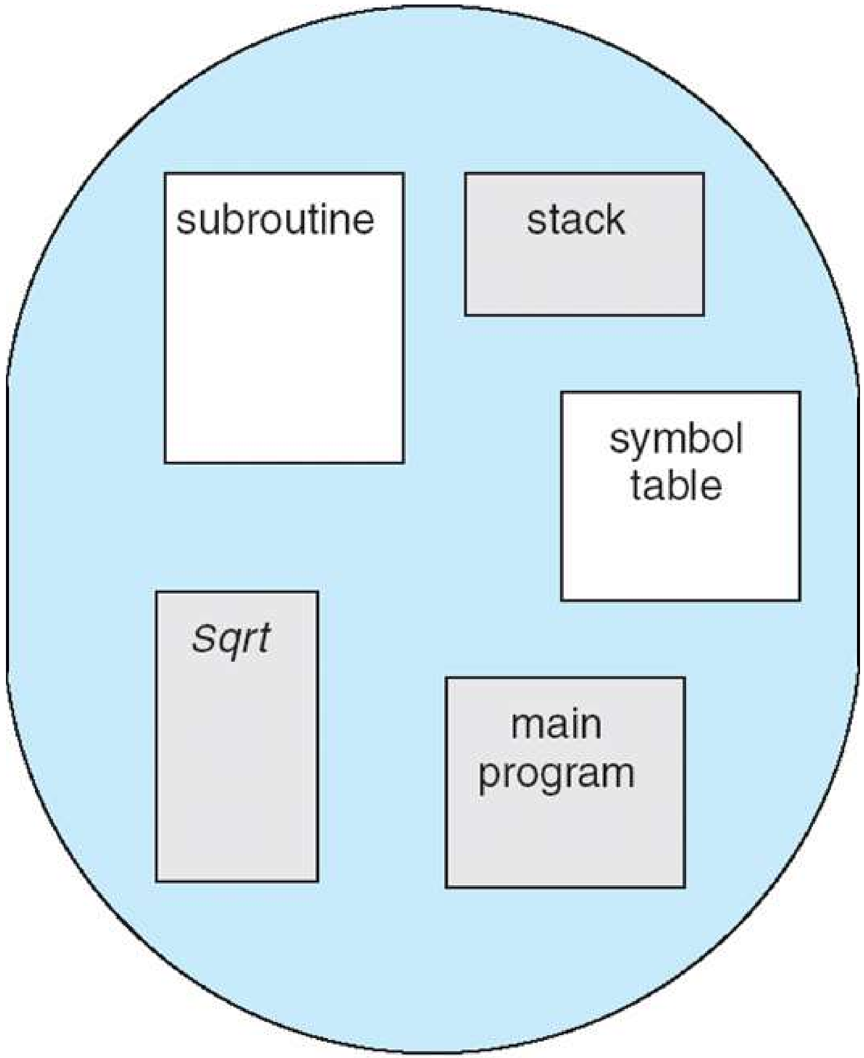
    common block

    stack

    symbol table

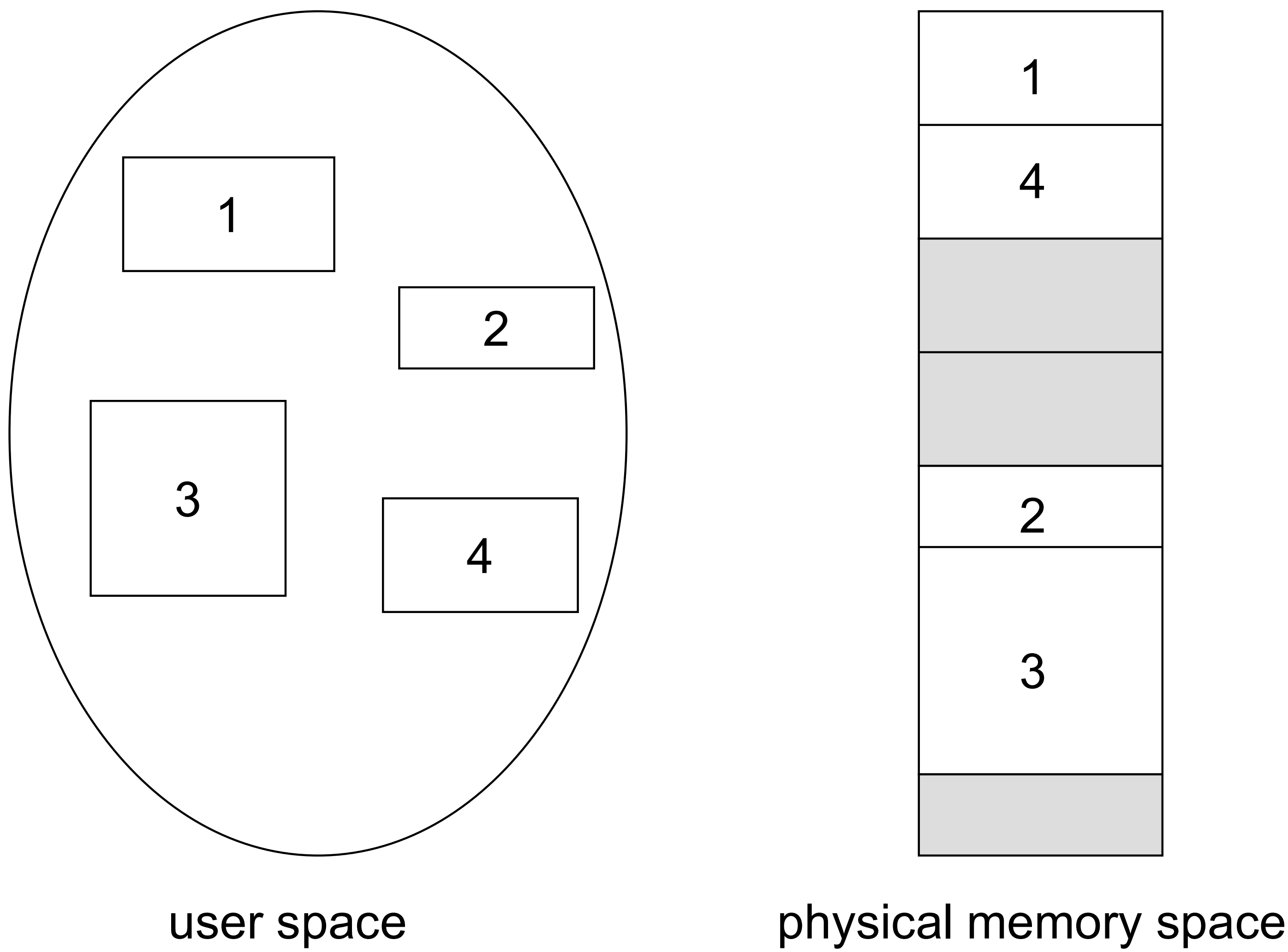
    arrays

User's View of a Program



logical address

Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:

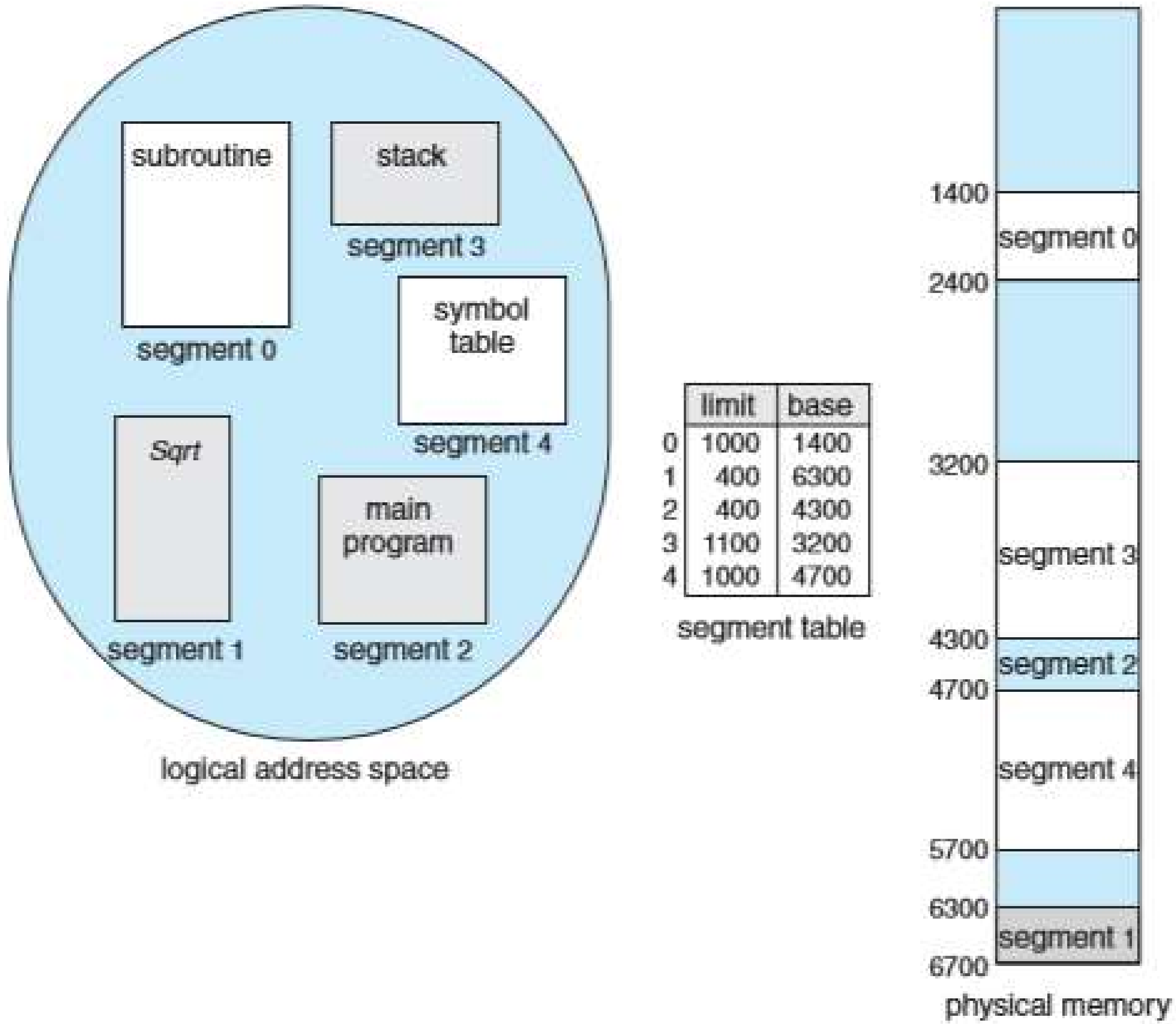
  <segment-number, offset>,

- **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 number of segments used by a program;

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

Segmentation Architecture (Cont.)

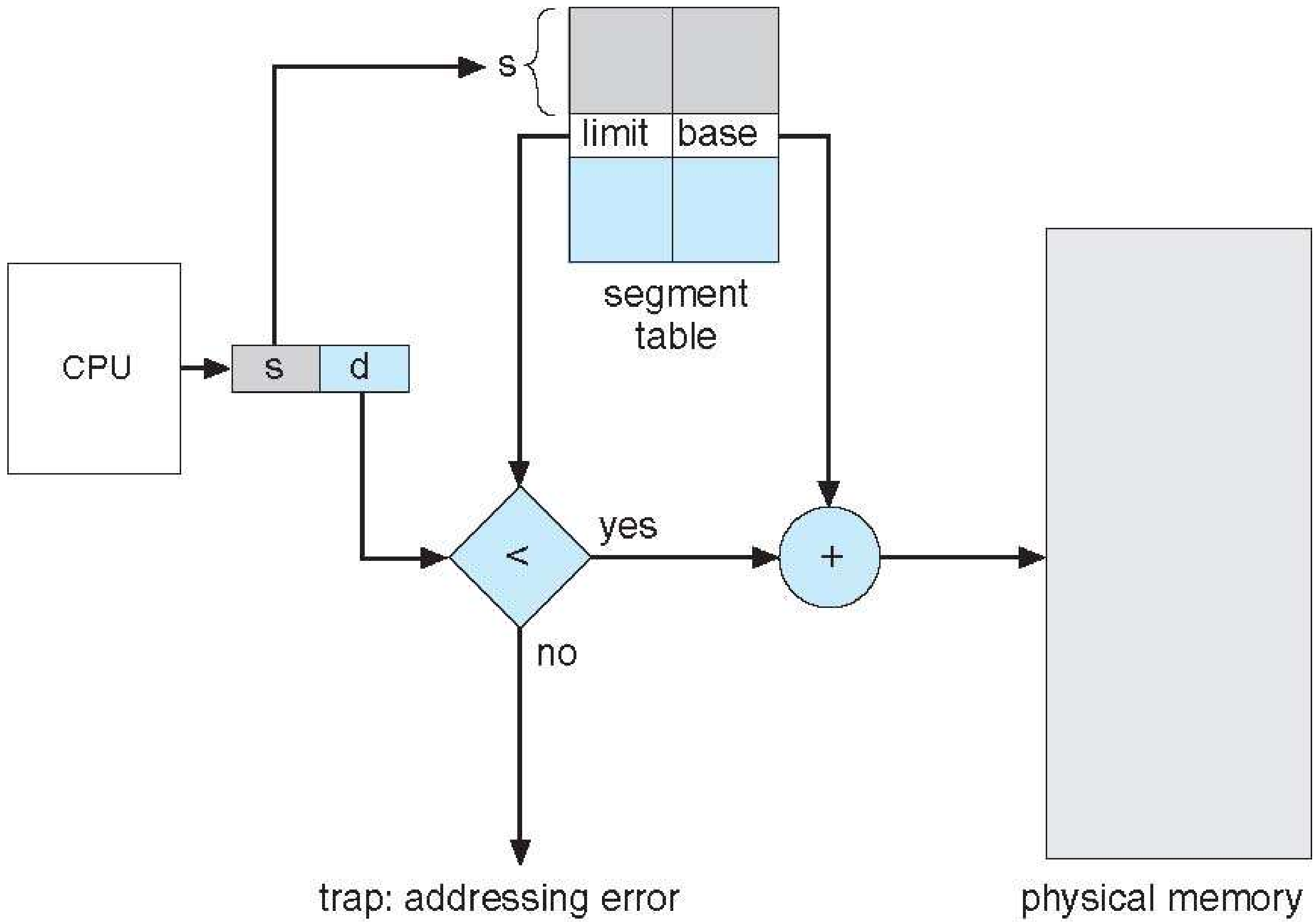
- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Segmentation Hardware



- Segmentation does not avoid external fragmentation and the need for compaction

Paging

- Physical address space of a process can be noncontiguous; 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 power of 2, between 512 bytes and 16 Mbytes
- **Divide logical memory into blocks of 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 program
- Set up a **page table** to **translate logical to physical addresses**
- **Backing store likewise split into pages**
- **Still have Internal fragmentation**

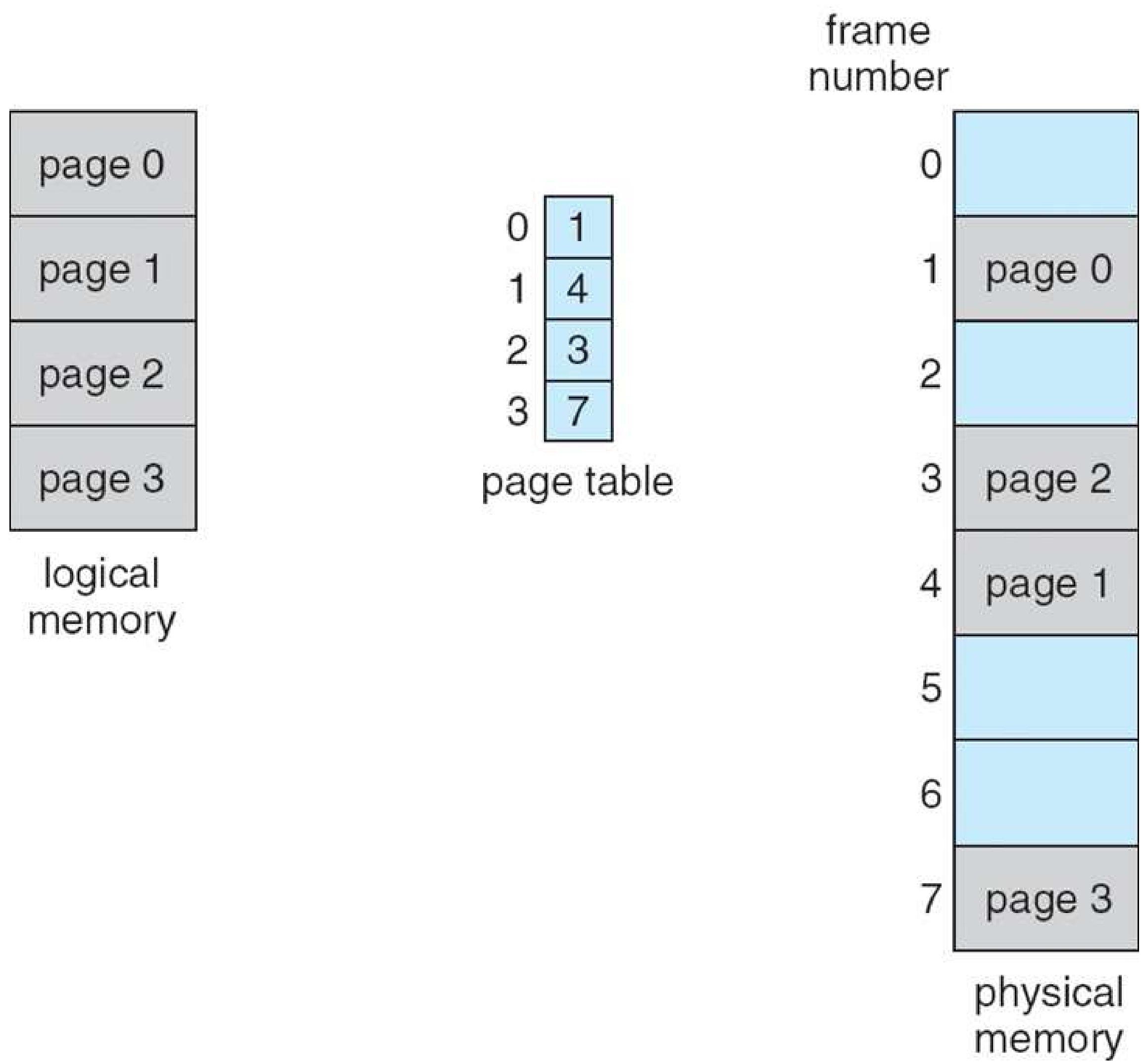
Address Translation Scheme

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

page number	page offset
p	d
m -n	n

  - For given logical address space 2^m and page size 2^n
  - where p is an index into the page table and d is the displacement within the page.
  - That is why we use a page size of power of 2.

Using a page size of 4 bytes and a physical memory of 32 bytes (8 frames),

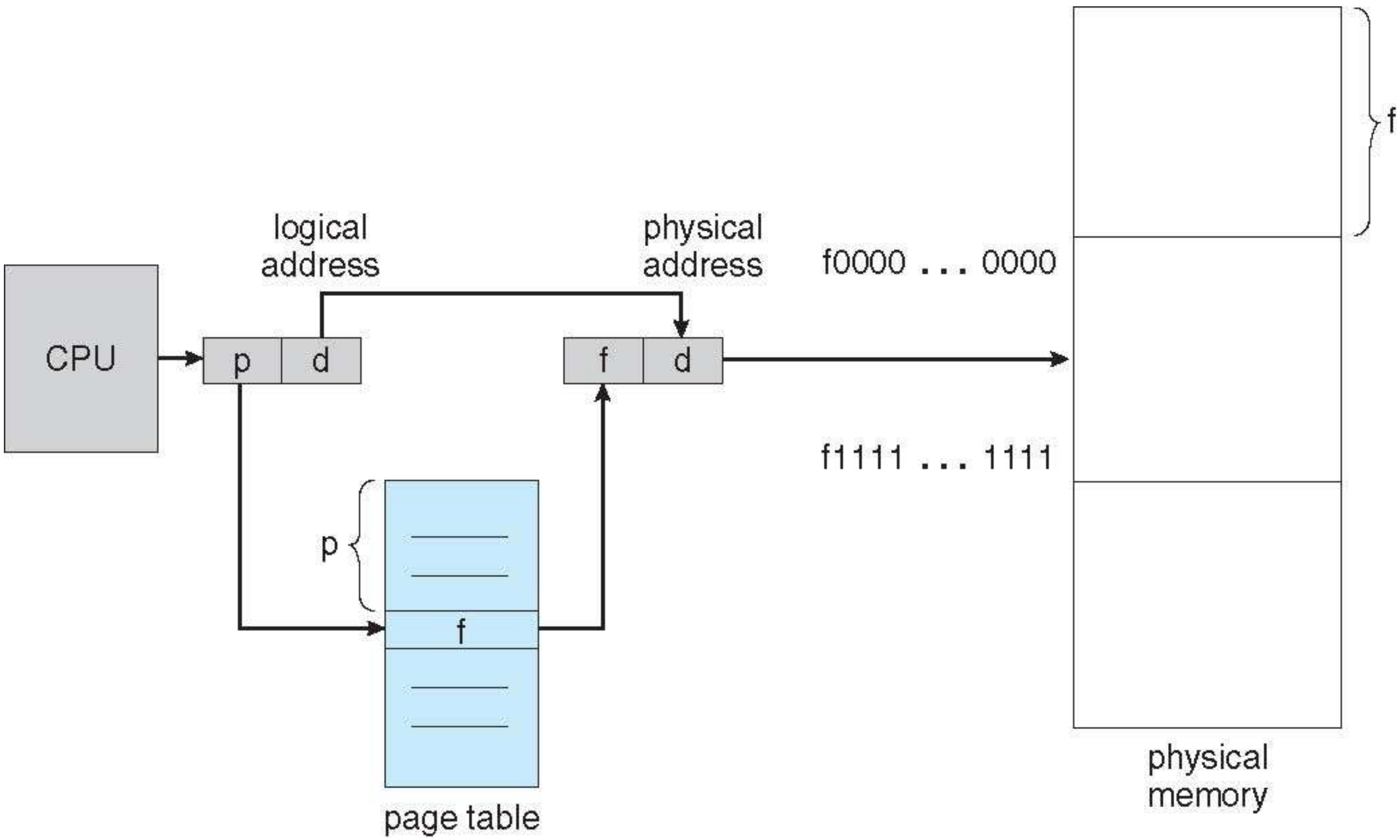


- Logical address 0 is page 0, offset 0.
- Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 [= $(5 \times 4) + 0$].
- Logical address3 (page 0, offset3) maps to physical address 23

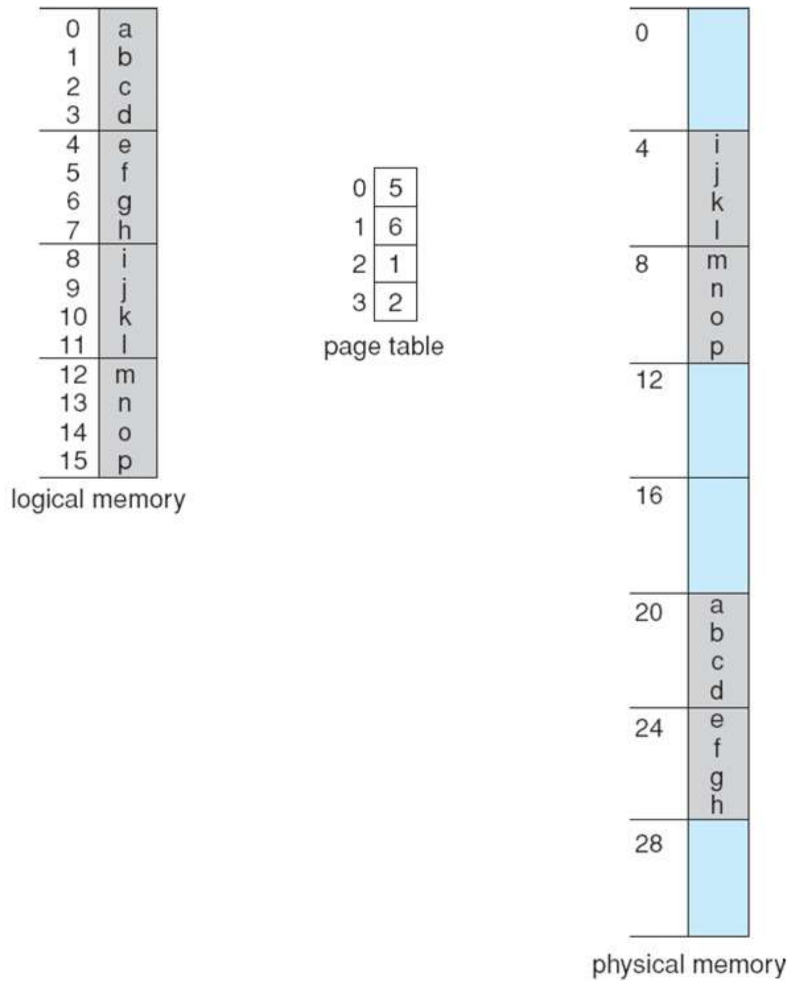
  [= $(5 \times 4) + 3$].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.
- Thus, logical address 4 maps to physical address 24 [= $(6 \times 4) + 0$].

- When we use a paging scheme,
- we have no external fragmentation: any free frame can be allocated to a process that needs it.
- However, we may have some internal fragmentation.

Paging Hardware



Paging Example

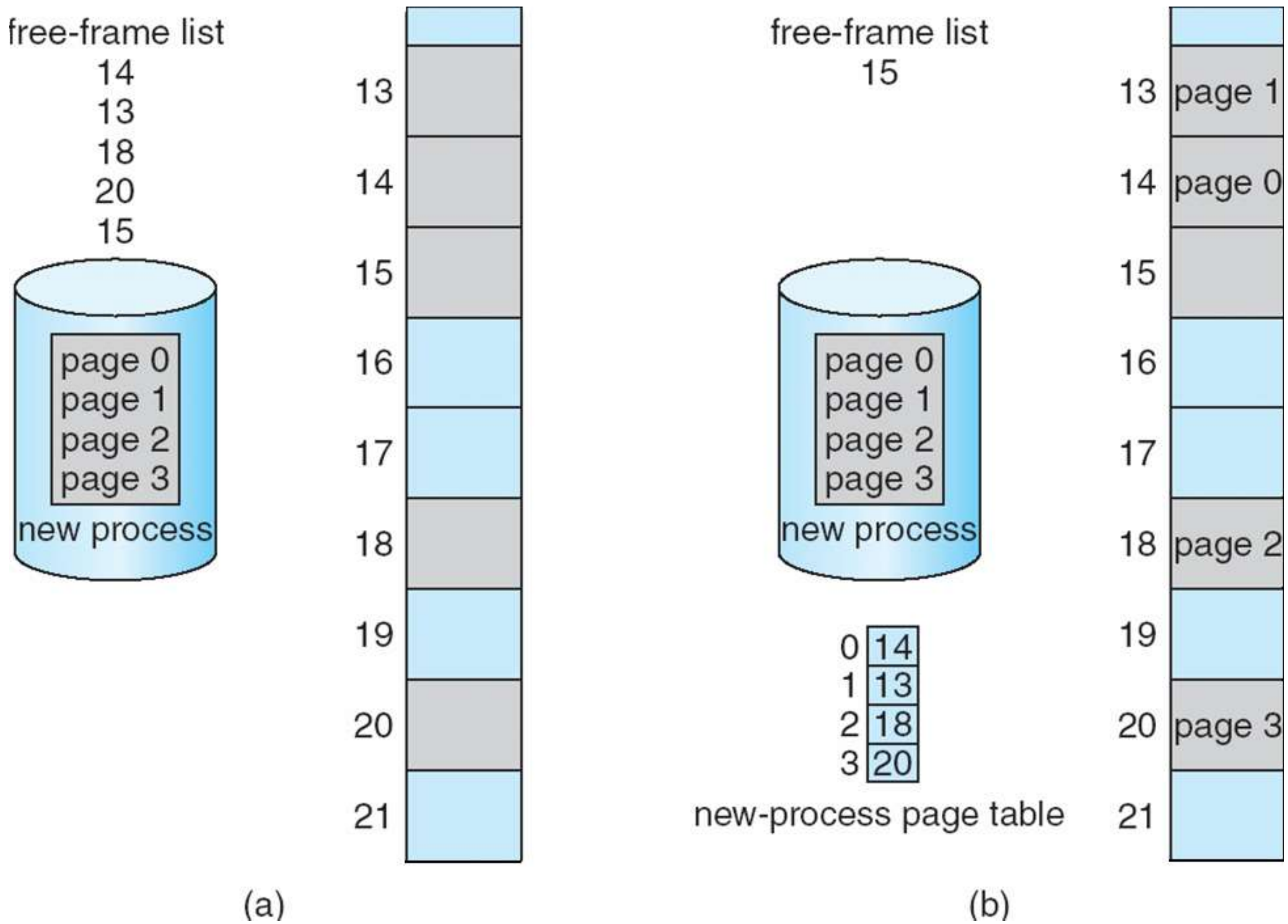


n=2 and *m*=4   32-byte memory and 4-byte pages

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
  - 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
- Process view and physical memory now very different
- By implementation process can only access its own memory

Free Frames



Before allocation

After 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
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- 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**)

Implementation of Page Table (Cont.)

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, 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

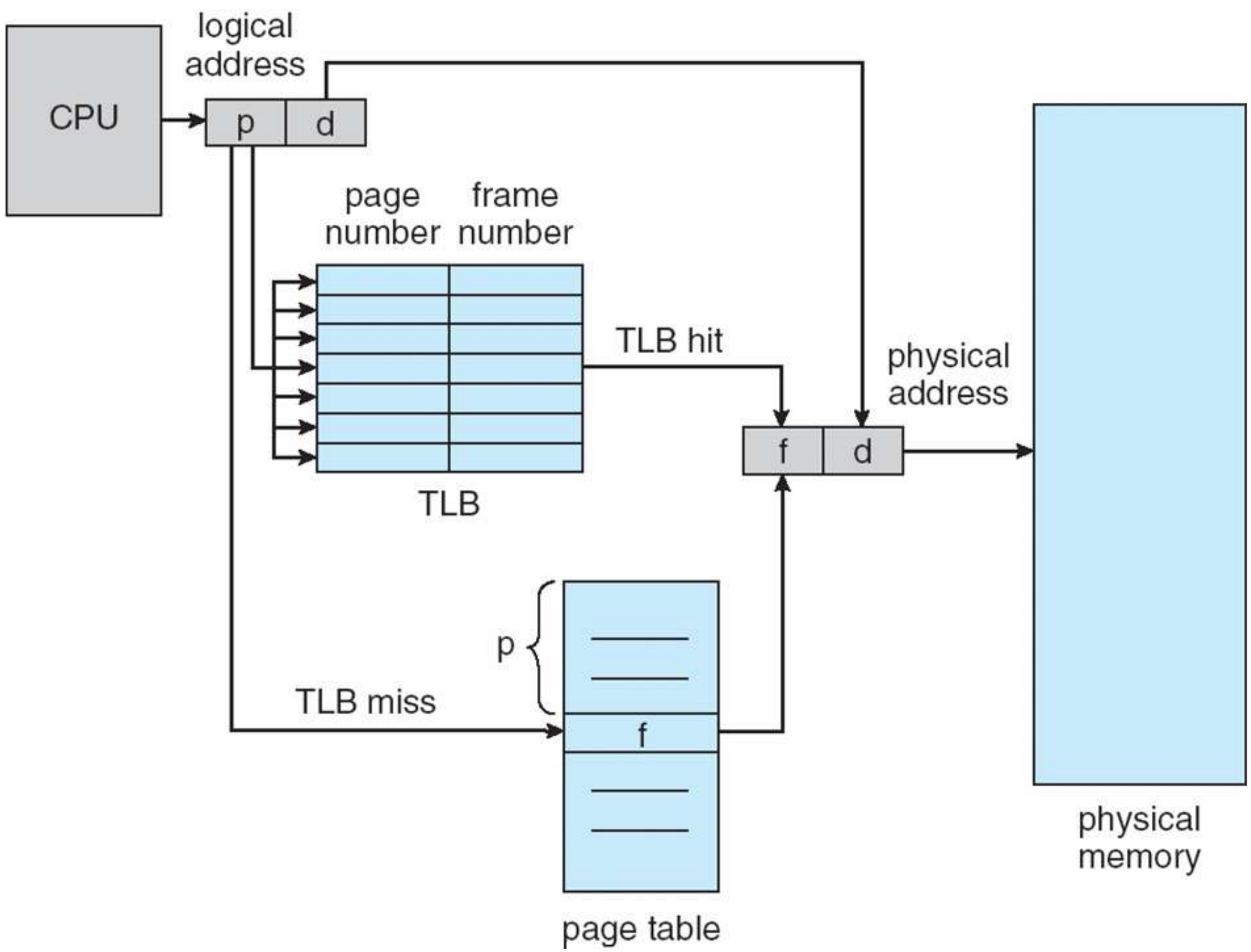
Associative Memory

- 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

- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
- **Effective Access Time** (**EAT**)

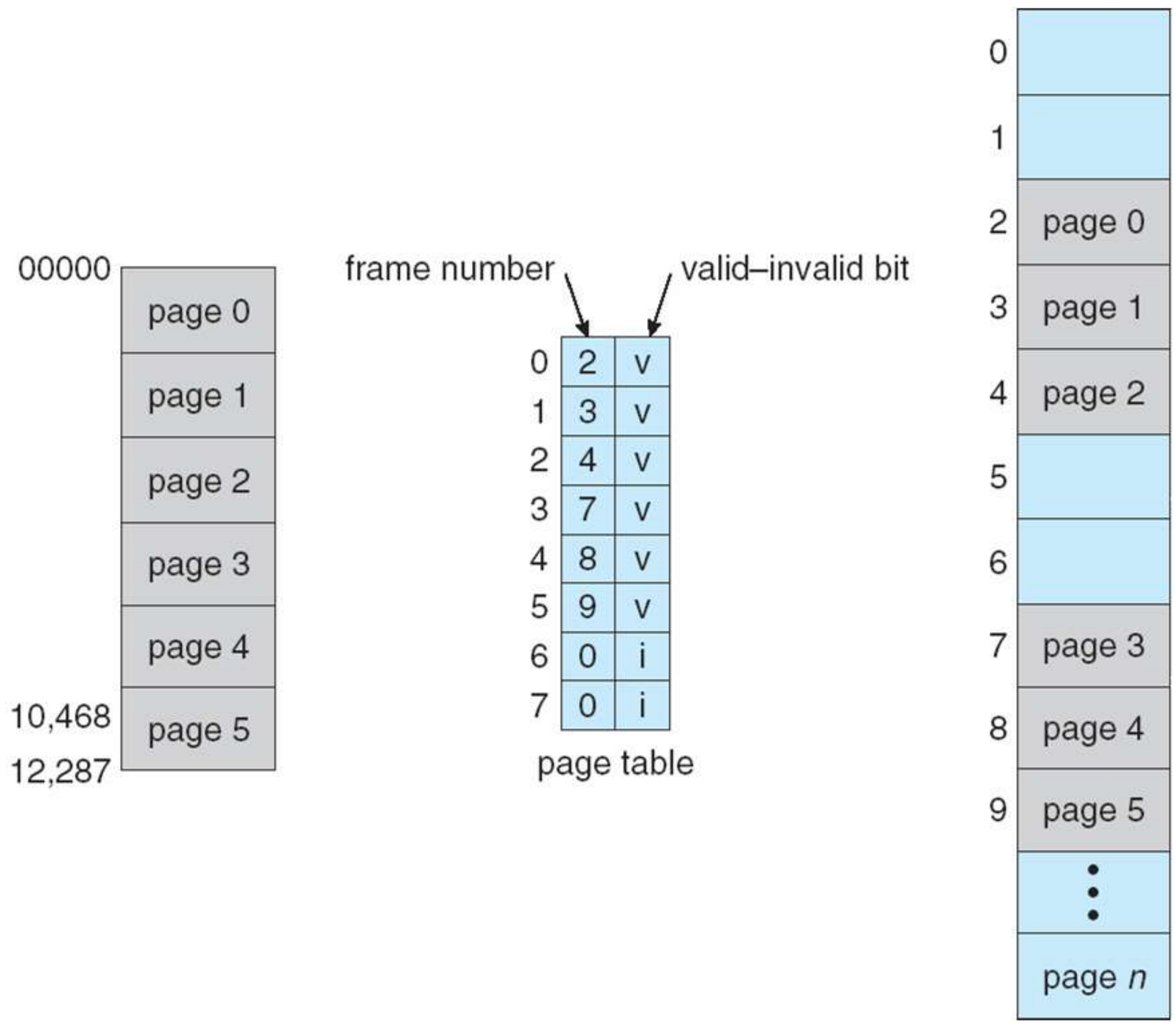
$$EAT = (1xMA + \varepsilon)\ \alpha + (2xMA + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha \qquad \text{(MA: Memory Access)}$$

- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = (1x100+20) x 0.8 + (2x100+20) x 0.20 = 96 + 44 = 140 ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = (1x100+20) x 0.99 + (2x100+20) x 0.01 = 118.8 + 2.20 = 121 ns

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" 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
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

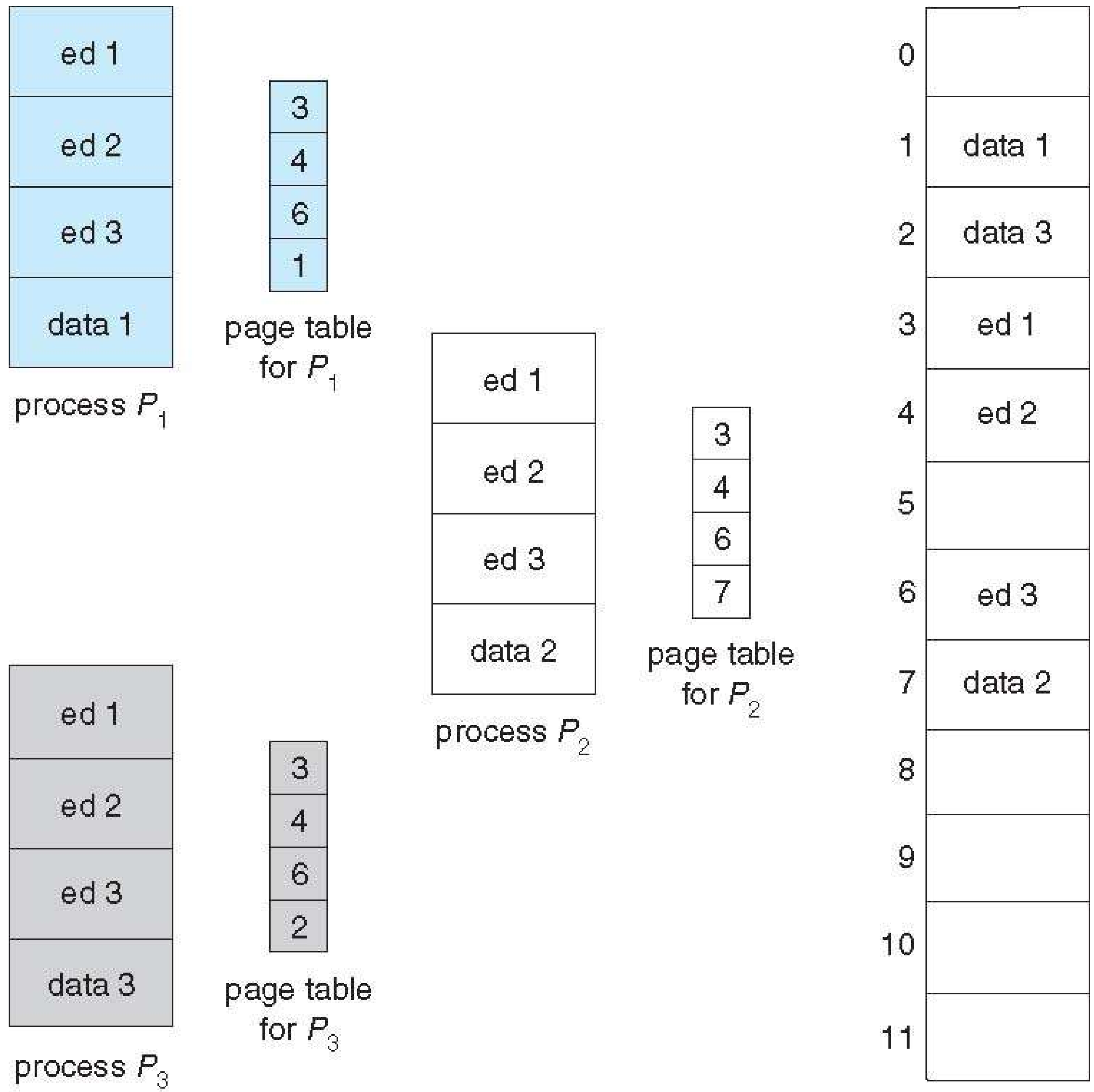
Valid (v) or Invalid (i) Bit In A Page Table



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 interprocess 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



End of Chapter 8