VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SWE3001-Operating Systems

Prepared By
Dr. L. Mary Shamala
Assistant Professor
SCOPE/VIT

Module 5: Memory Management

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

Objectives

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

Background

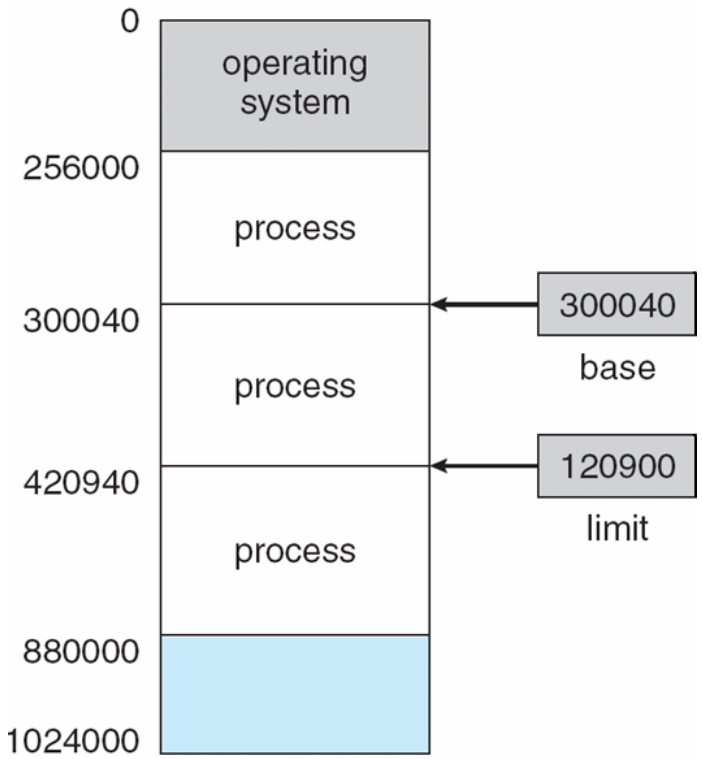
- To increase CPU utilization, keep several processes in memory
- The memory-management algorithms vary from a primitive bare-machine approach to paging and segmentation strategies.
- Several issues that are pertinent to managing memory
  - Basic hardware
  - Binding of symbolic memory addresses to actual physical addresses
  - The distinction between logical and physical addresses
  - Dynamic linking and shared libraries

Basic Hardware

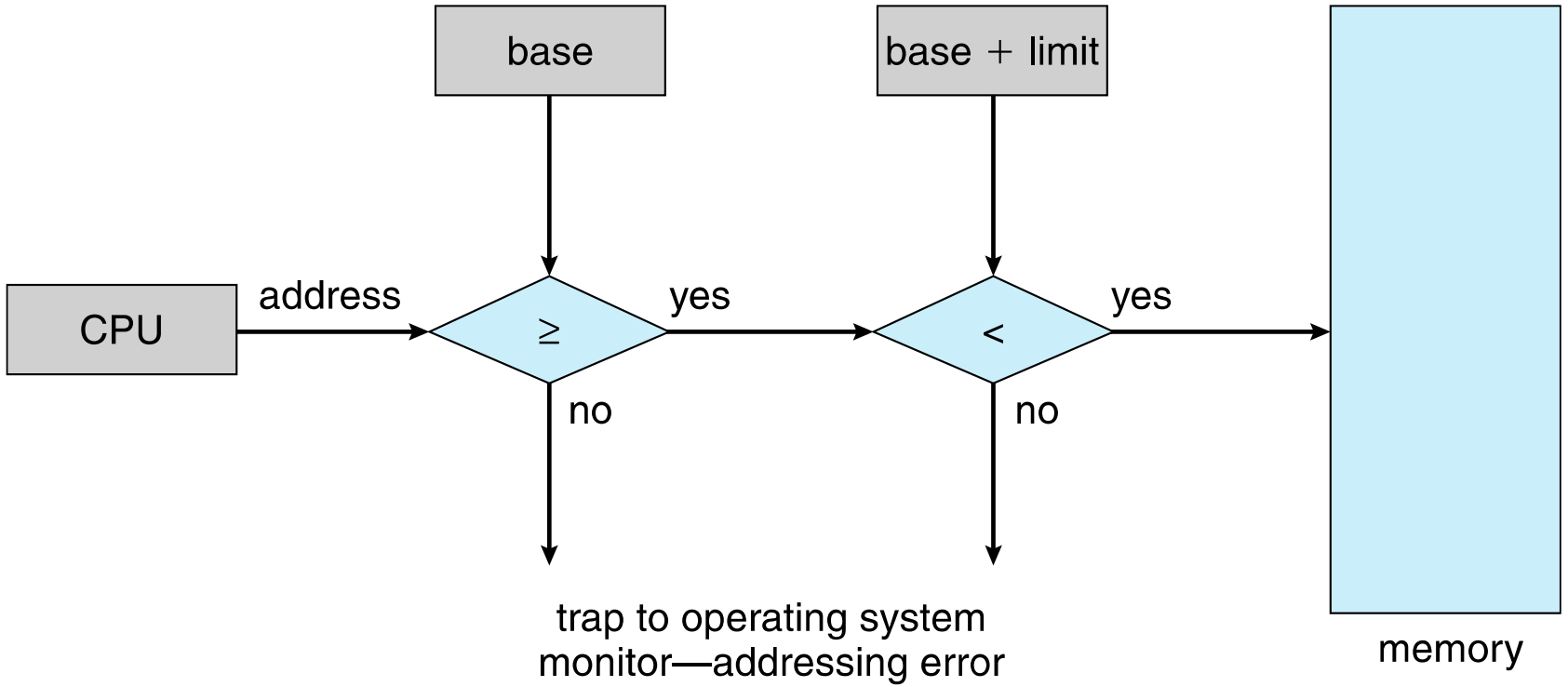
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or 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

- 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



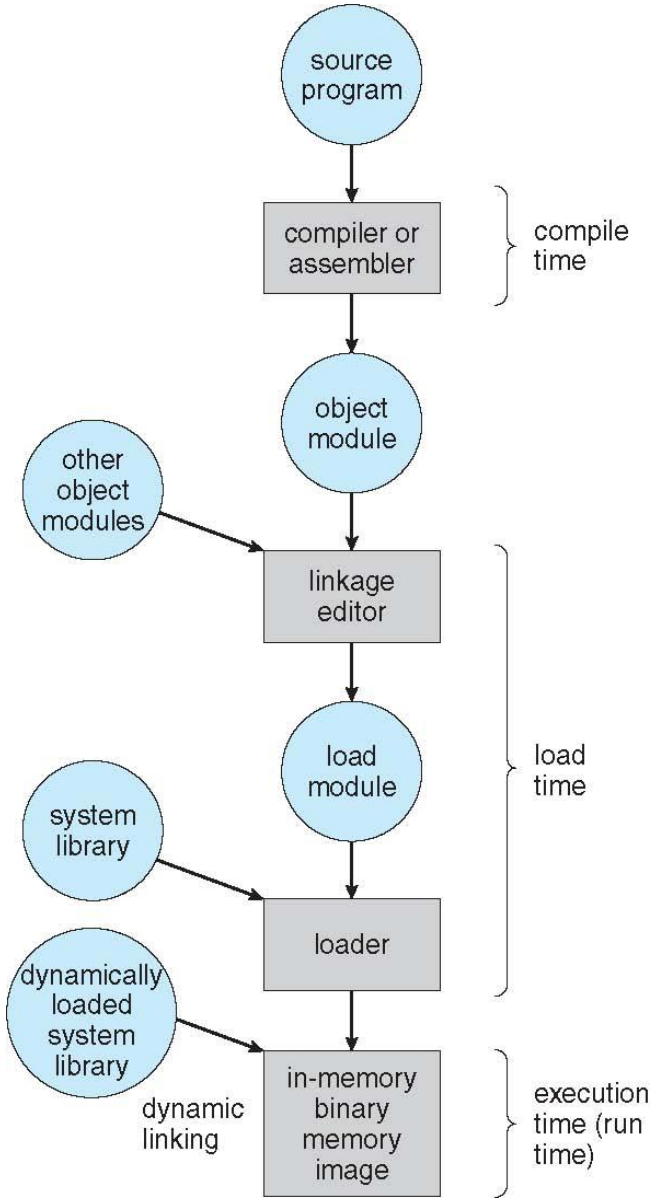
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
- 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
- Linker or loader will bind relocatable addresses to absolute addresses
- 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



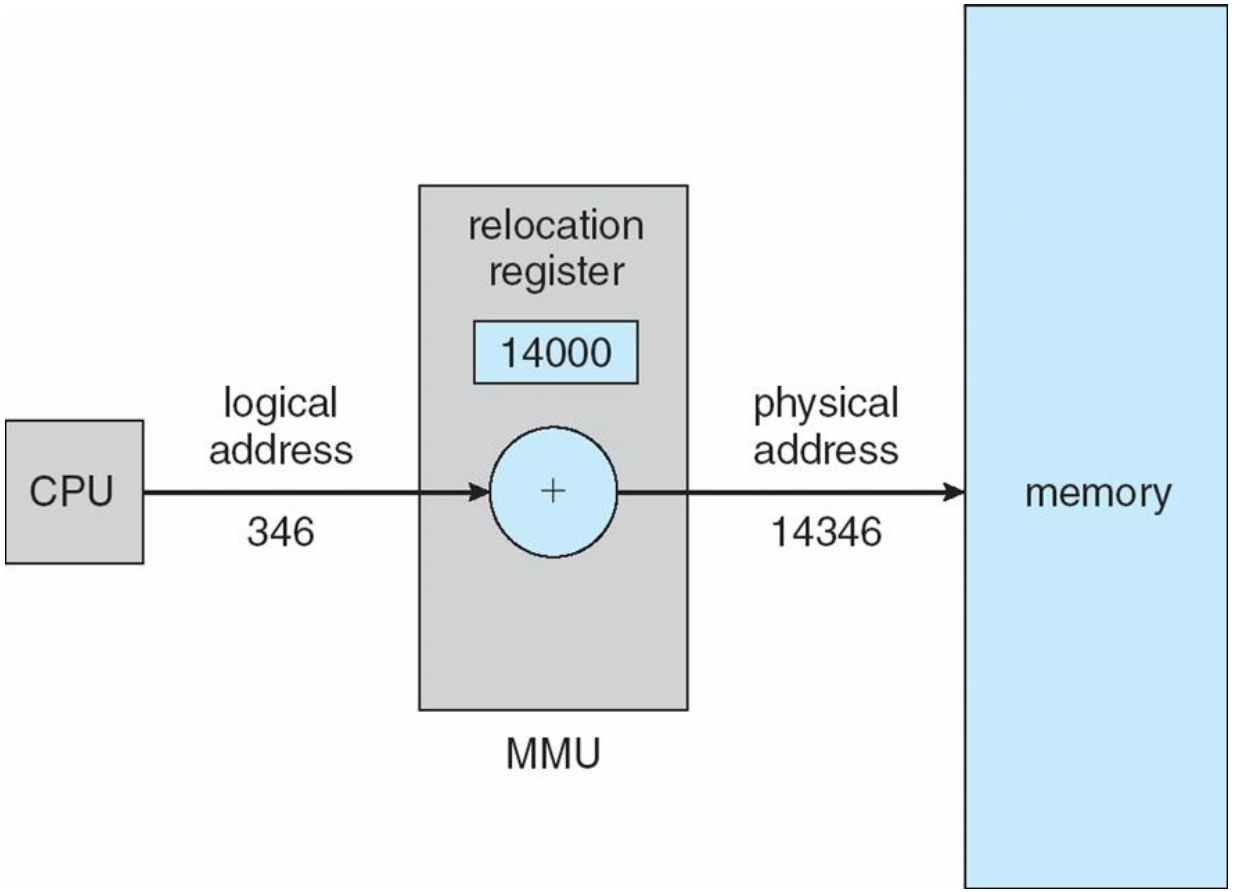
Logical vs. Physical Address Space

- 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
- In a simple 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
  - 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 location in memory
  - Logical address bound to physical addresses

Memory-Management Unit (MMU)…



Dynamic Loading

- 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

Dynamic Linking and Shared Libraries

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time.
- **Dynamically linked libraries** are system libraries that are linked to user programs when the programs are run
- 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
- 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 is also known as **shared libraries**
- Dynamic linking and shared libraries generally require help from the operating system.

Static linking

- Static linking is the process of incorporating the code of a program or library into a program so that it is linked at compile time.
- **Advantages**
  - It generally leads to faster program startup times since the library's code is copied into the executable file.
  - It makes it easier to debug programs since all symbols are resolved at compile time.
- **Disadvantages**
  - It can make programs harder to update since all of the code for a given library is copied into each program.
  - More computation cost as executable file size ewill be big.
  - This can lead to increased disk usage and memory usage.
  - It can lead to symbol clashes, where two different libraries define the same symbol.

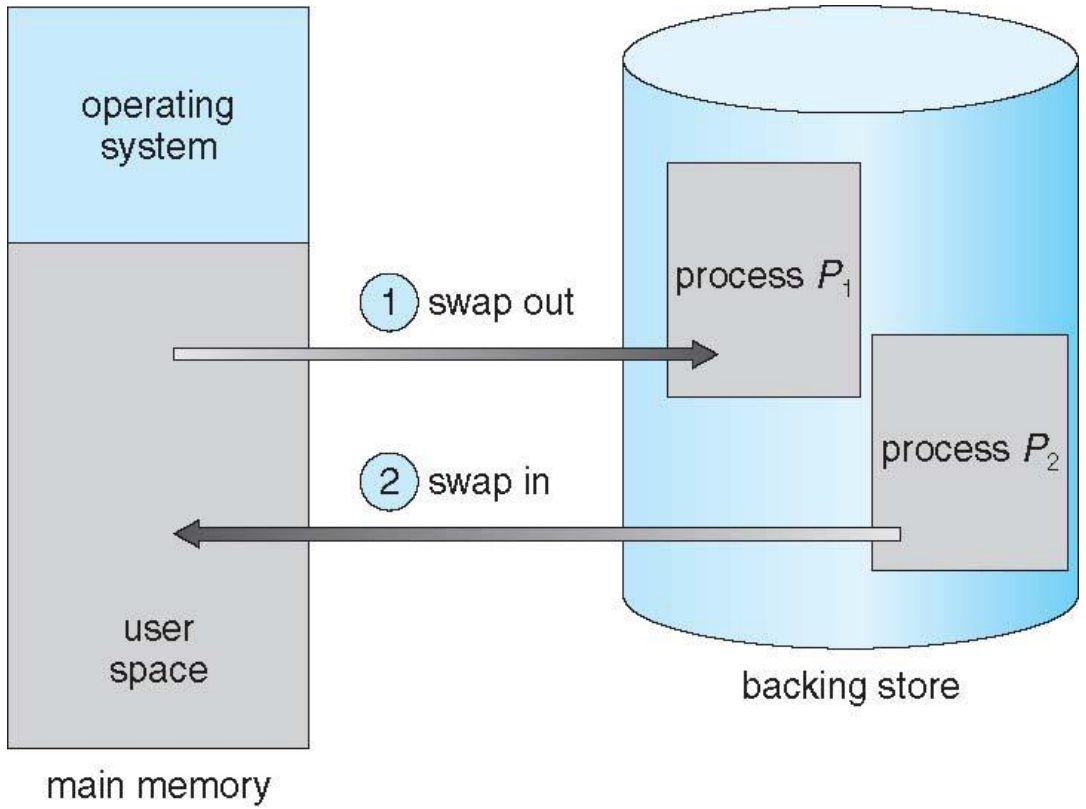
Dynamic linking

- In this, the library is not copied into the program's code. Instead, the program has a reference to where the library is located. When the program is run, it will load the library from that location.
- **Advantages**
  - Saves memory space.
  - Low maintenance cost.
  - Shared files are used.
- **Disadvantages**
  - Page fault can occur when the shared code is not found, then the program loads modules in the memory.

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
- It is further divided into two types:
  - **Swap-in:** Swap-in means removing a program from the hard disk and putting it back in the RAM.
  - **Swap-out:** Swap-out means removing a program from the RAM and putting it into the hard disk.
- **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

Schematic View of Swapping



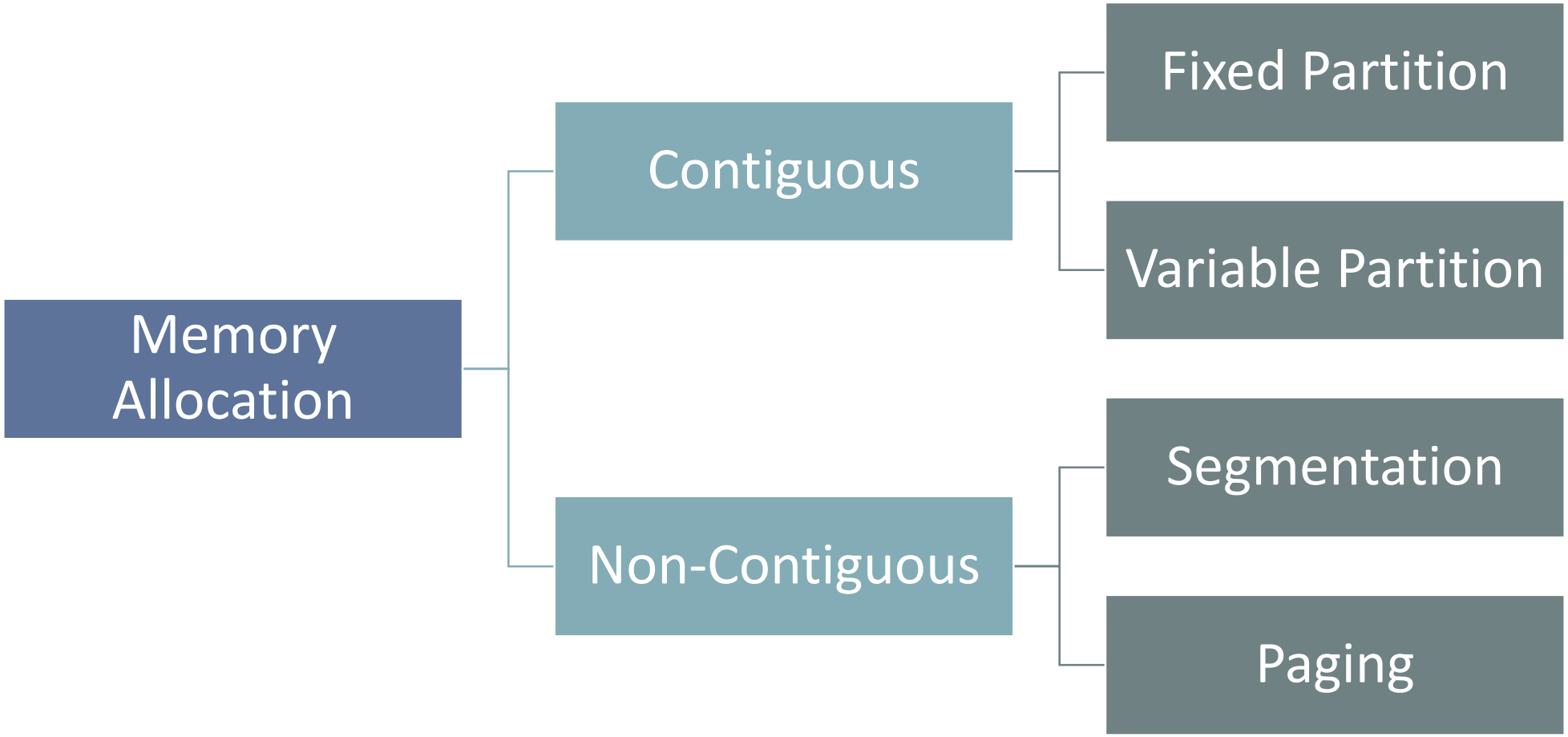
Context Switch Time and Swapping

- 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
- Swapping is constrained by other factors as well
  - 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 modified version is common-Swap only when free memory extremely low
- Swapping portions of processes—rather than entire processes—to decrease swap time

Memory Allocation

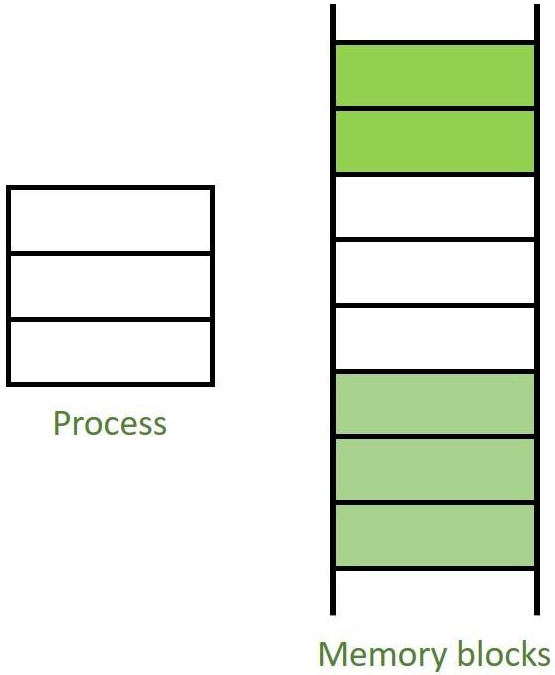
- Main memory must support both OS and user processes
- Limited resources, must allocate efficiently
- Main memory is usually into two **partitions**:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
- For several processes,
  - Allocate the same memory or partitions of different sizes
  - Place in contiguous block or any available slots in memory

Memory Allocation Techniques



Contiguous Allocation

- Contiguous allocation is one early method
- Each process is contained in single contiguous section of memory

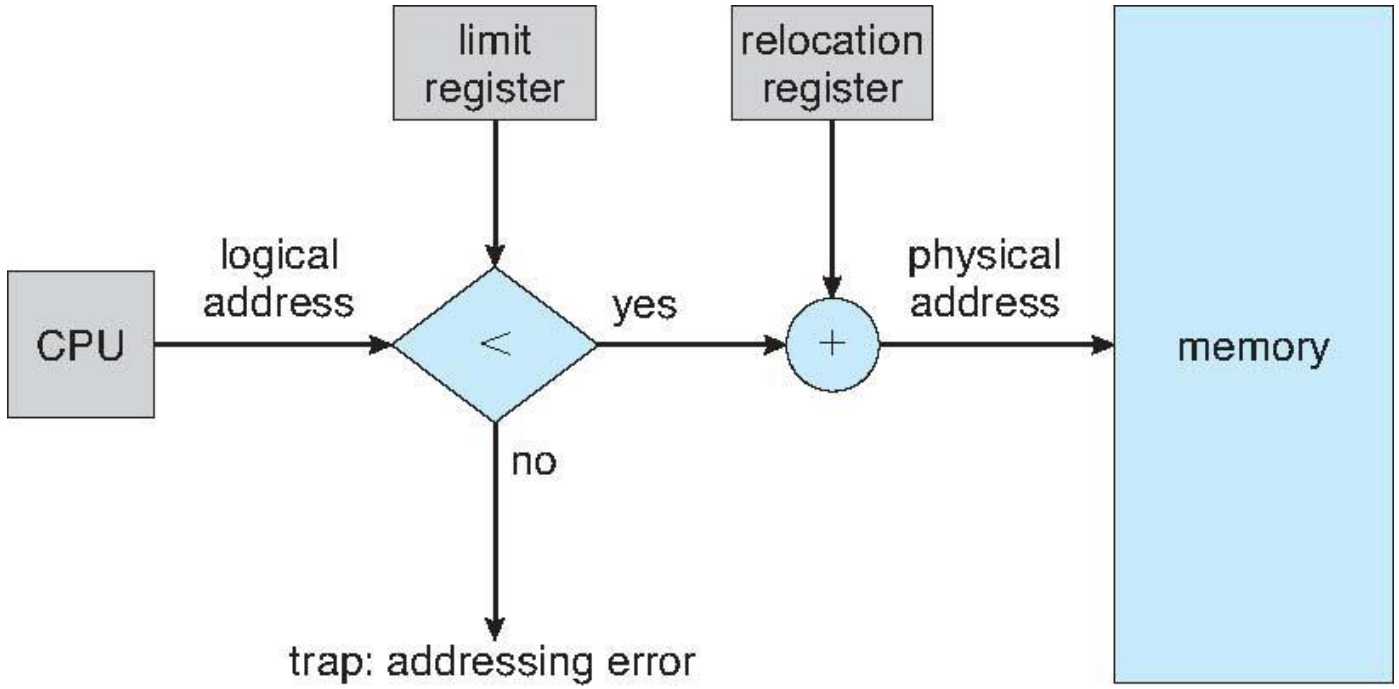


Contiguous Memory Allocation

Memory Protection

- Relocation registers are used to protect user processes from each other, and from changing operating-system code and data
- Relocation register contains the 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* by adding the value in relocation register
- The relocation register scheme provides an effective way to allow the operating system's size to change dynamically.
- Can then allow actions such as kernel code being **transient** and kernel changing size

Hardware Support for Relocation and Limit Registers



Memory Allocation- Fixed Partition

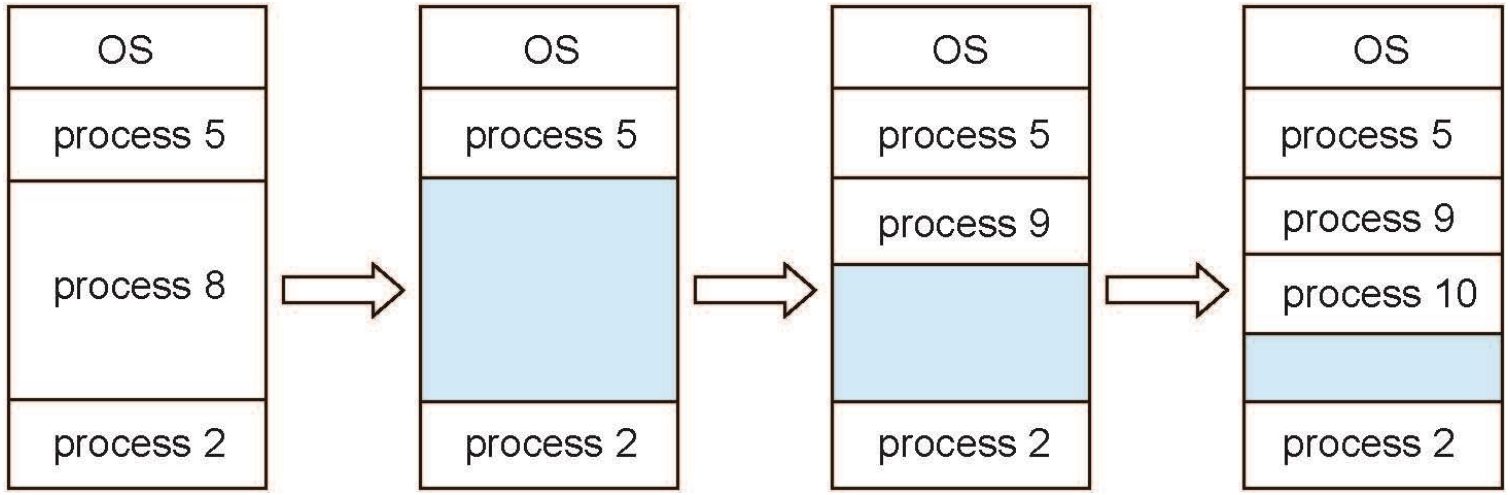
- Multiple-partition method
- Divide memory into several fixed-sized partitions.
- Each partition may contain exactly one process.
- When a partition is free, a process is selected from the input queue and is loaded into the free partition.
- When the process terminates, the partition becomes available for another process.
- Degree of multiprogramming limited by number of partitions

Memory Allocation- Variable Partition

- **Variable-partition** sizes for efficiency (sized to a given process' needs)
- OS keeps a table indicating which parts of memory are available and which are occupied.
- Initially, all memory is available for user processes and is considered one large block of available memory, a **hole**.
- The memory blocks available comprise a set of holes of various sizes scattered throughout memory.
- When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process.
- If the hole is too large, it is split into two parts.
  - One part is allocated to the arriving process;
  - The other is returned to the set of holes

Memory Allocation- Variable Partition…

- When a process terminates, it releases its block of memory, which is then placed back in the set of holes.
- If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole→ **Coalescing holes**
- Operating system maintains information about:
  a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

- How to satisfy a request of size ***n*** from a list of free holes?
- **First-fit**: Allocate the ***first*** hole that is big enough
- **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless the list is 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

Practice

- Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory

Fragmentation

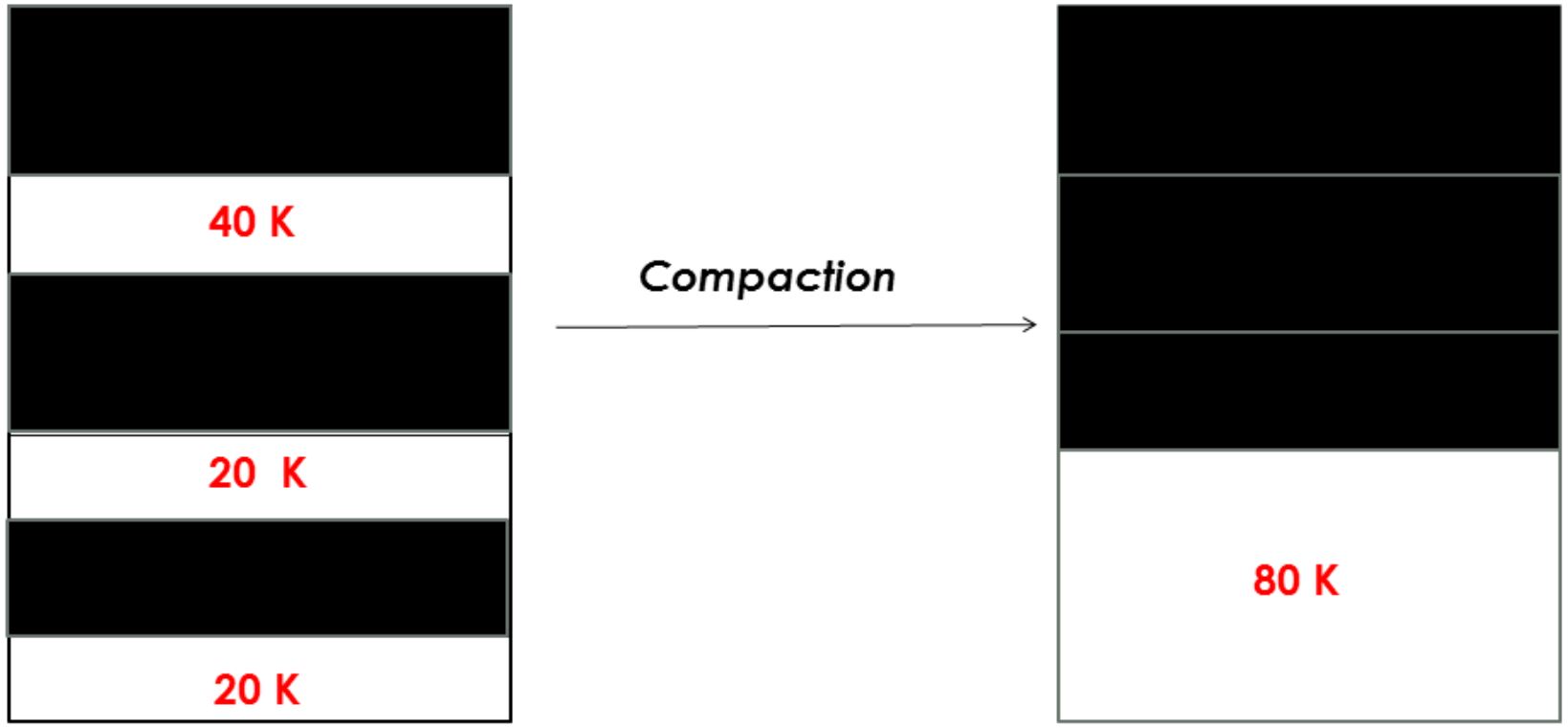
- **Fragmentation**- Having holes between adjacent memory partitions
- **Inability to use the available memory perfectly**
- Fragmentation occurs when most free blocks are too small/large to satisfy any request perfectly.
- There are two types of fragmentation.
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous; memory is fragmented into a large number of small holes
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation (Cont.)

- Reduce external fragmentation by storage **compaction**
- Shuffle memory contents to place all free memory together in one large block
- Sometimes referred to as **burping the storage** or **garbage collection**
- Compaction is possible *only* if relocation is dynamic, and is done at execution time
- The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory.
  - Expensive technique
- Another solution is to permit the logical address space of the processes to be noncontiguous
  - Segmentation
  - Paging
  - Combined

Compaction

- The free space of a running system is compacted, to reduce fragmentation problem and improve memory allocation efficiency.



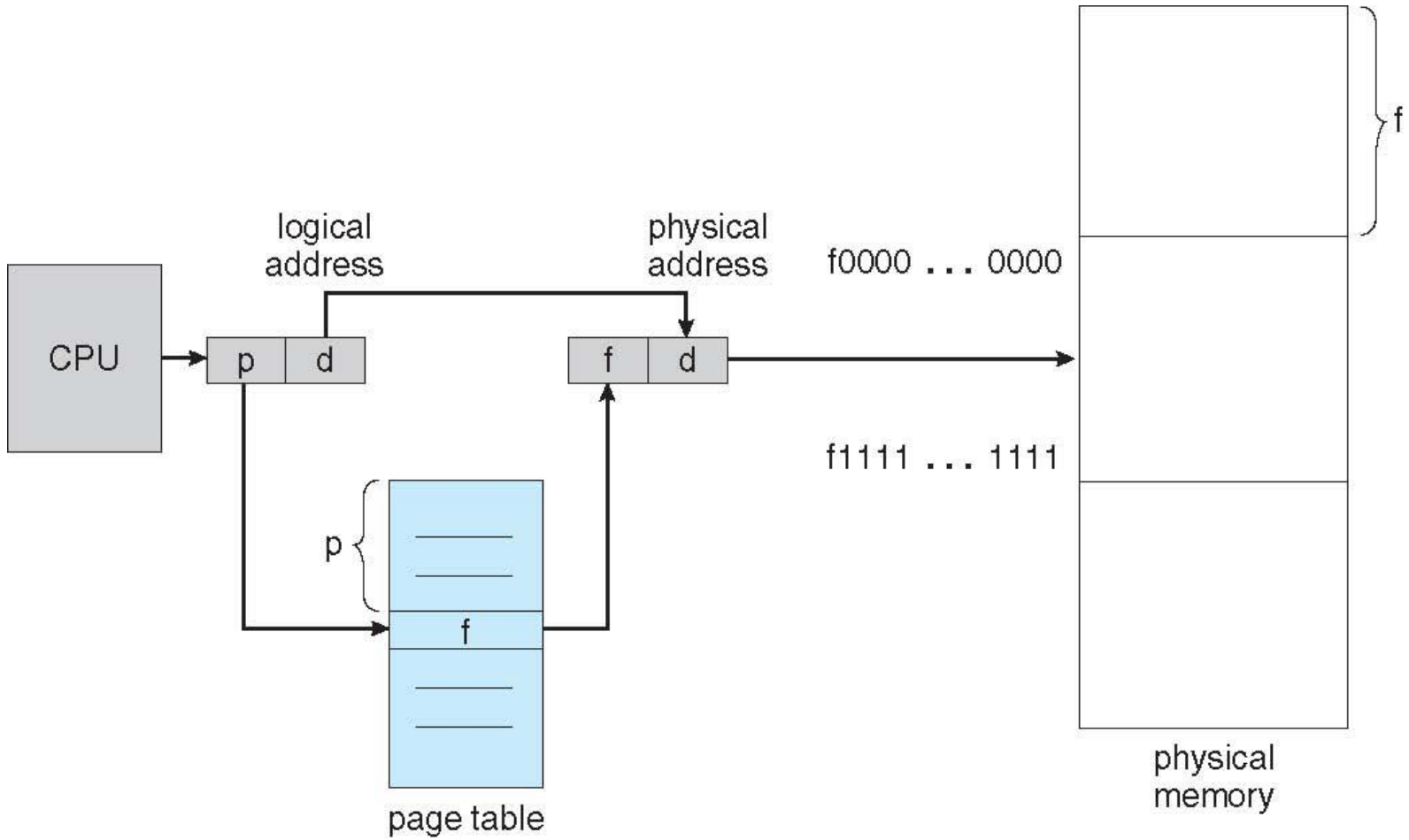
Paging

- The physical address space of a process can be noncontiguous; process is allocated physical memory whenever is available
- Avoids external fragmentation and the need for compaction
- Avoids the problem of fitting varying-sized memory chunks onto backing store
- Paging in its various forms is used in most operating systems, from mainframes to smartphones.
- Paging is implemented through cooperation between the operating system and the computer hardware

Basic Method

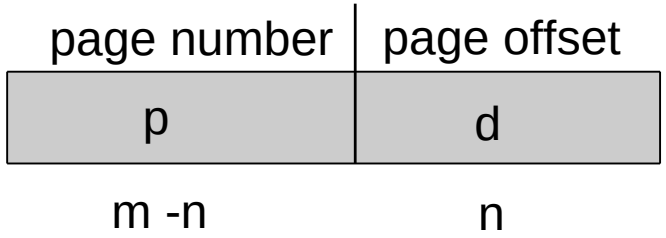
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**
- When a process is to be executed, its pages are loaded into any available memory frames
- Keep track of all free 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

Paging Hardware



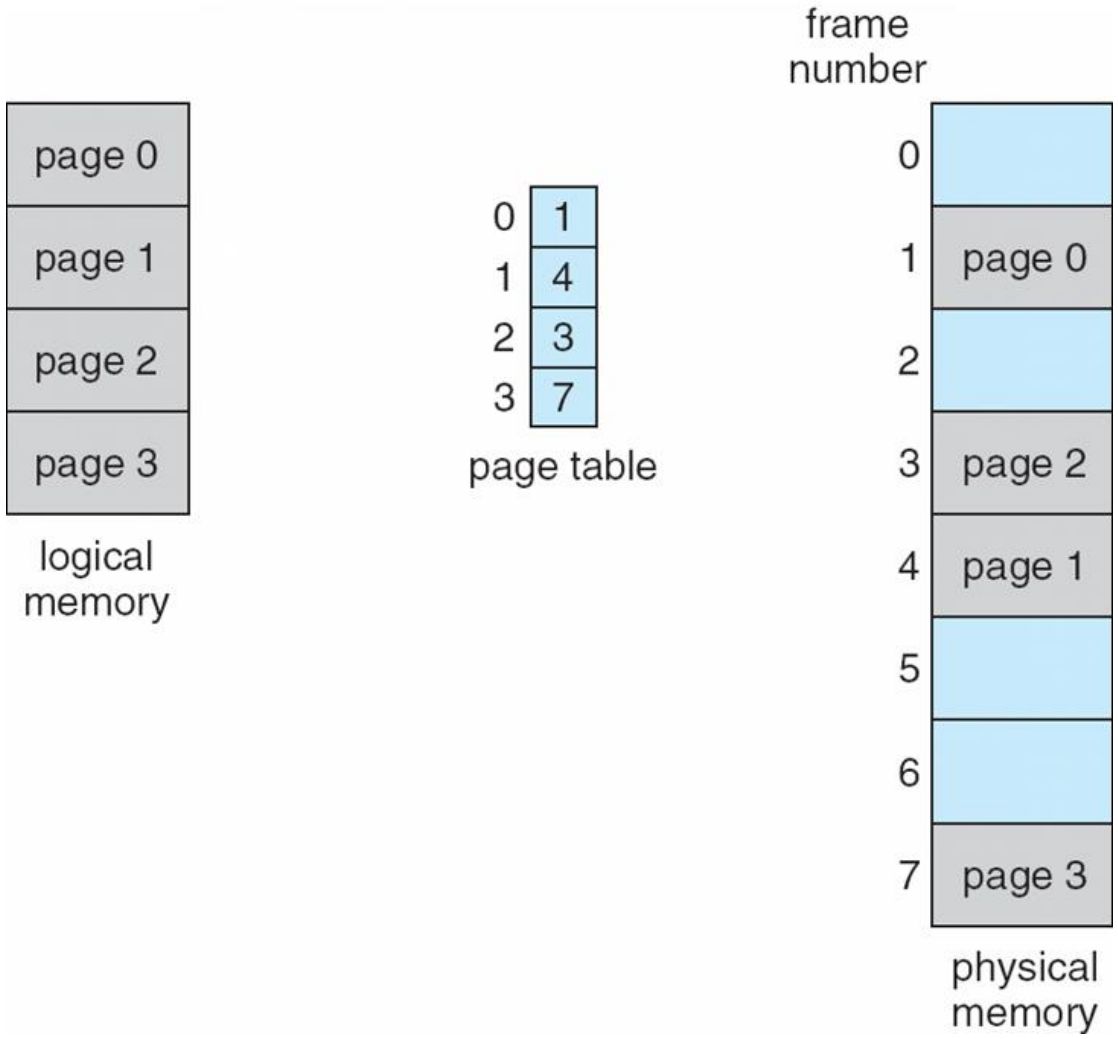
Address Translation Scheme

- 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 bytes and page size 2^n bytes, the logical address is defined as:

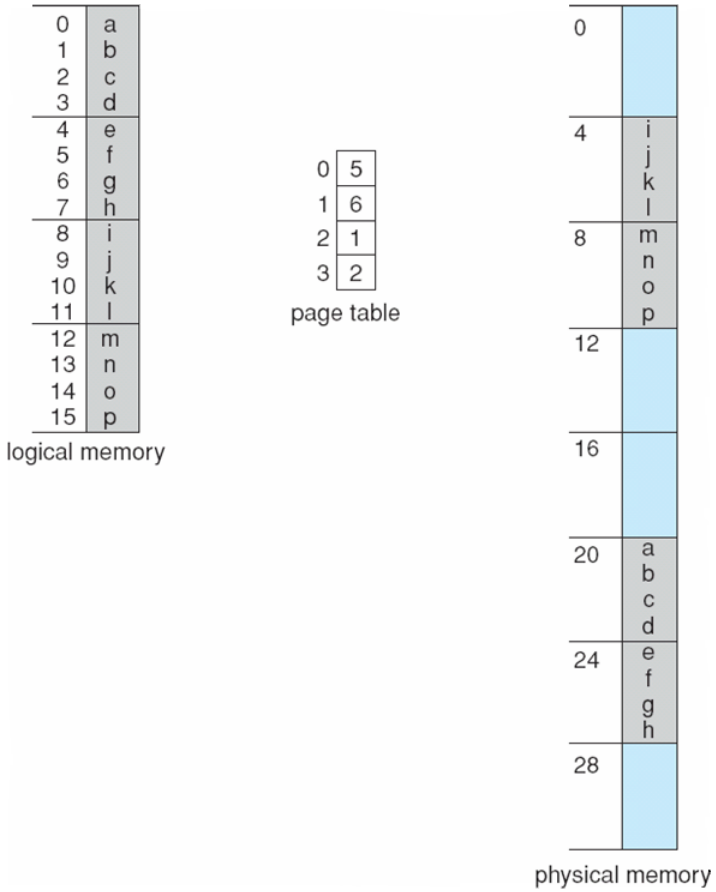


- The page size (like the frame size) is defined by the hardware.
- The size of a page is a power of 2, varying between 512 bytes and 1 GB per page

Paging Model of Logical and Physical Memory



Paging Example

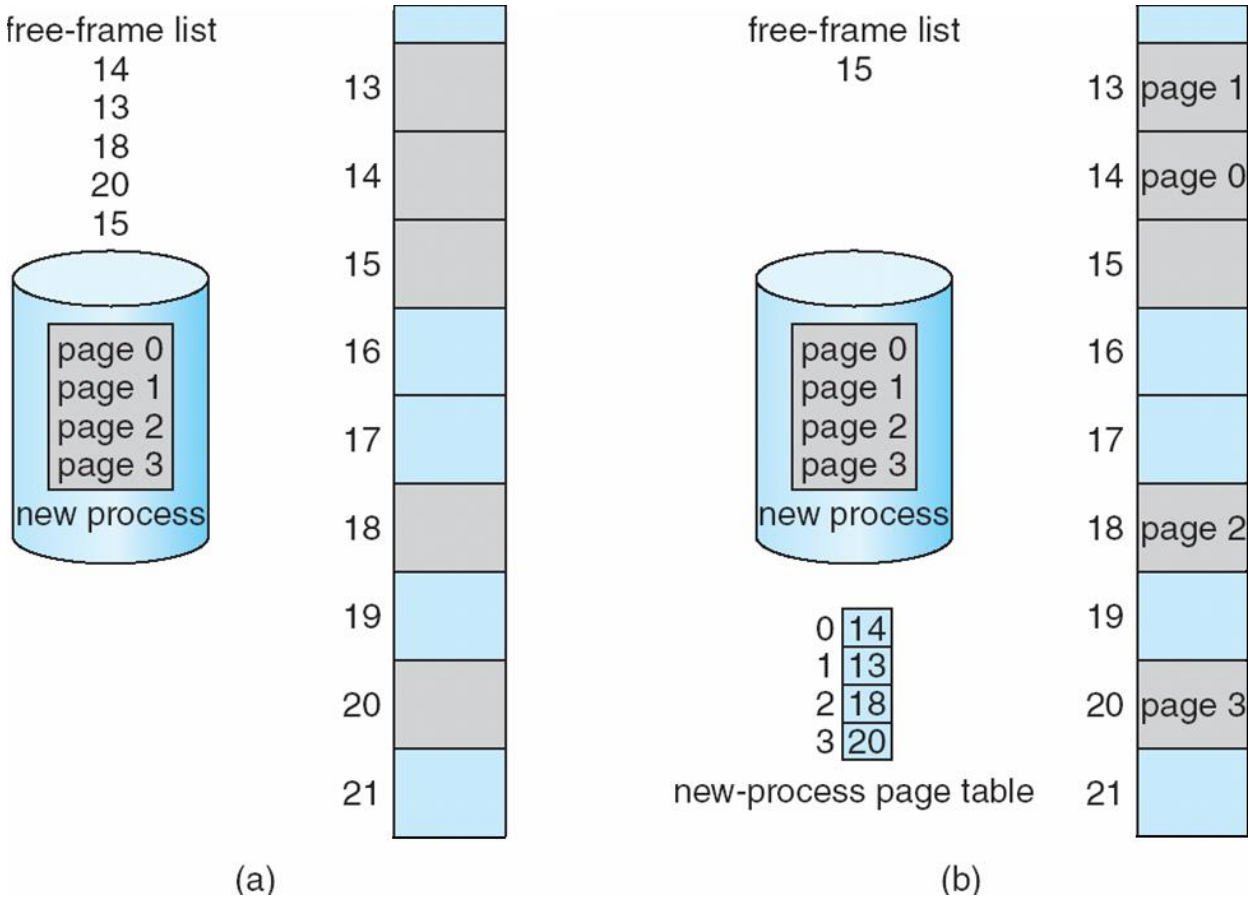


n=2 and *m*=4 32-byte Physical memory and 4-byte pages

Paging (Cont.)

- No external fragmentation, but have 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
- Page sizes growing over time- 4 KB to 8 MB
- An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory.
  - The programmer views memory as one single space, containing only this one program.
  - In fact, the user program is scattered throughout physical memory, which also holds other programs.
- By implementation process can only access its own memory

Free Frames



Before allocation

After allocation

Frame Table

- A data structure maintains the allocation details of physical memory
  - which frames are allocated
  - which frames are available
  - how many total frames there are, and so on.
- Contains one entry for each physical page frame
  - free or allocated
  - if it is allocated, to which page of which process or processes.

Page Table

- OS maintains a copy of the page table for each process
- This copy is used to translate logical addresses to physical addresses
- It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU
- The hardware implementation of the page table can be done in several ways
- Page table is implemented as a set of dedicated registers
  - These registers should be built with very high-speed logic to make the paging-address translation efficient
  - Suitable if the page table is reasonably small

Hardware Support

- 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
- 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.)

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
- If the item is found, the corresponding value field is returned.
- The search is fast; expensive hardware
- 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

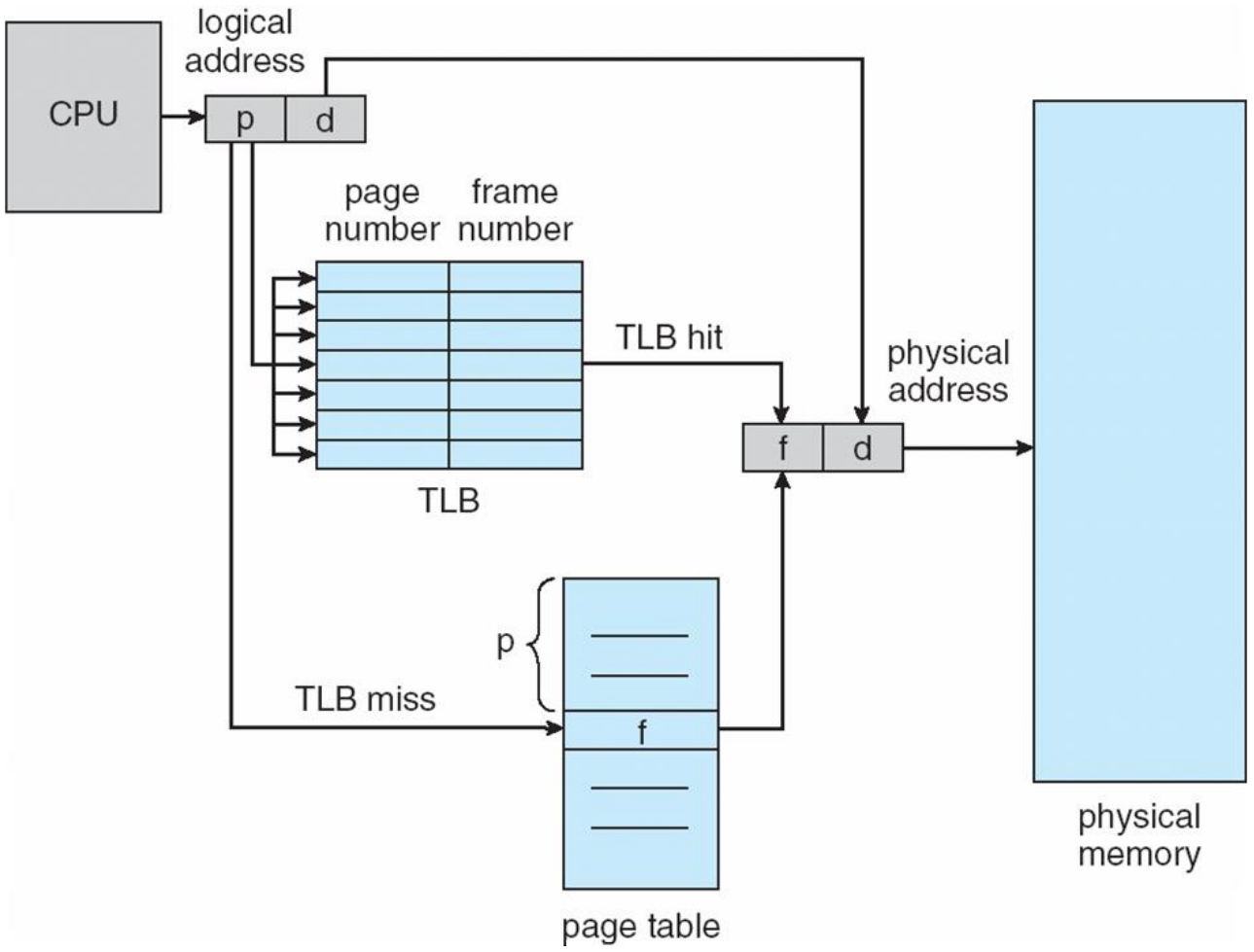
Implementation of Page Table (Cont.)

- 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
- 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



Implementation of Page Table (Cont.)

- The percentage of times that the page number of interest is found in the TLB is called the **hit ratio(**α).
- The fraction of memory references that can be satisfied from the associative memory is called the hit ratio.
- Hit ratio is related to number of associative registers
- The higher the hit ratio, better the performance

Effective Access Time

- **Effective Access Time (EAT)**

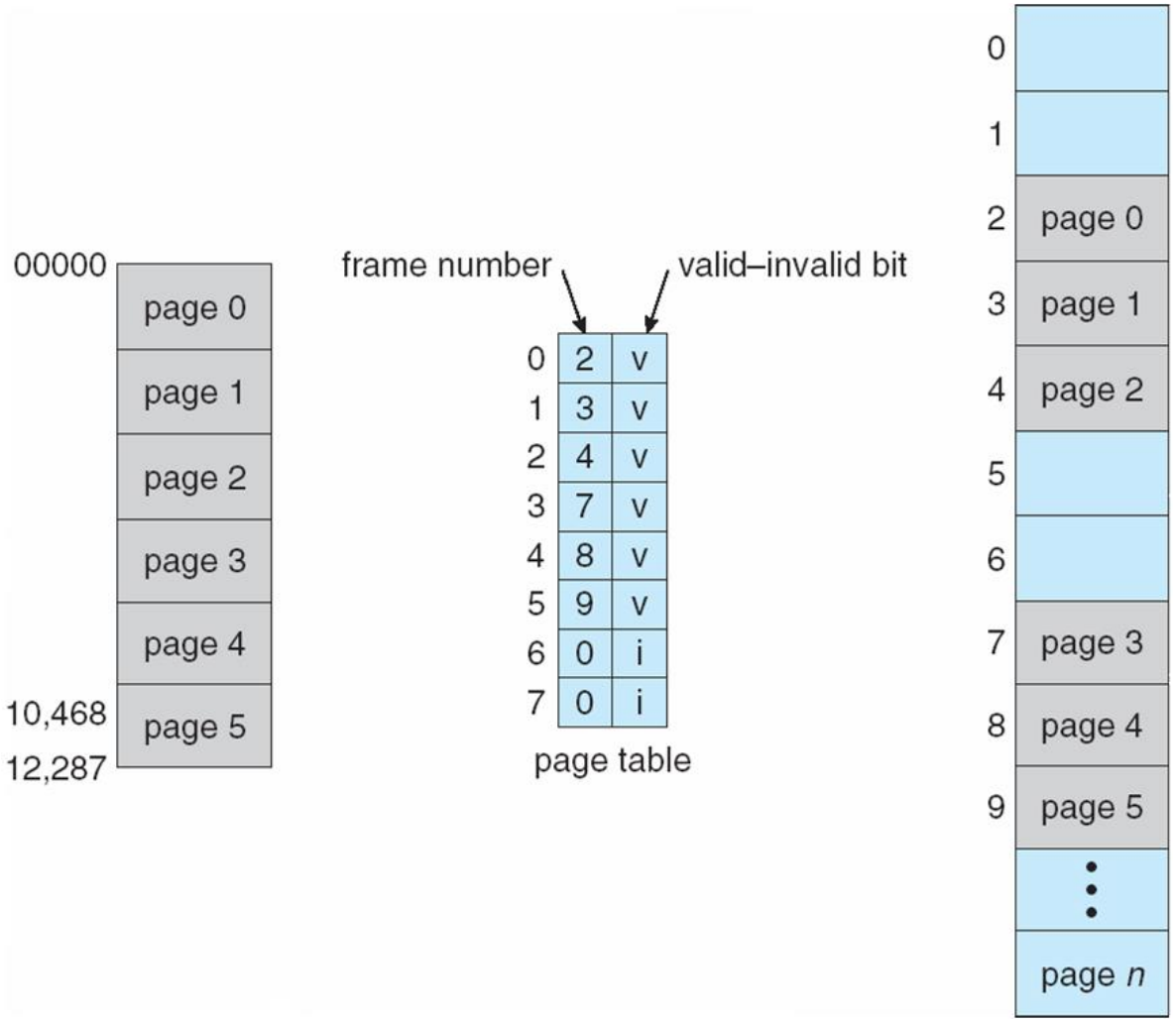
$$EAT = 2 * \varepsilon - \alpha$$

  - Associative Lookup = ε time unit
  - Hit ratio = α
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Consider more realistic hit ratio -> α = 99%, ε = 20ns for TLB search, 100ns for memory access
  - EAT = 0.99 x 120 + 0.01 x 220 = 121ns

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**) -indicates size of the page table
- Any violations result in a trap to the kernel

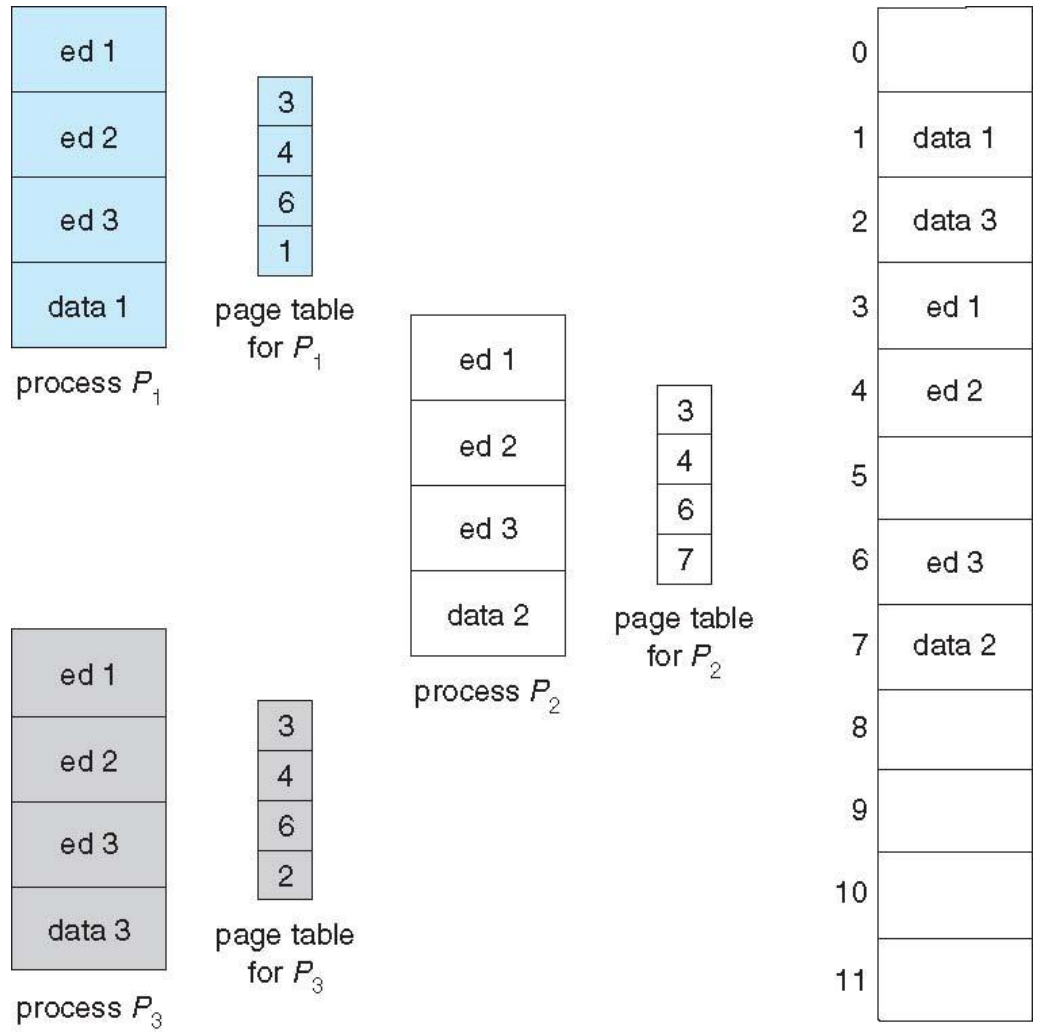
Valid (v) or Invalid (i) Bit In A Page Table



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)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **Reentrant code**
  - Reentrant code is non-self-modifying code: it never changes during execution.
  - Thus, two or more processes can execute the same code at the same time

Shared Pages Example



Practice

- Consider a paging system with the page table stored in memory.

a) If a memory reference takes 50 nanoseconds, how long does a paged memory reference take?

b) If we add TLBs, and 75 percent of all page-table references are found in the TLBs, what is the effective memory reference time? (Assume that finding a page-table entry in the TLBs takes 2 nanoseconds, if the entry is present.)

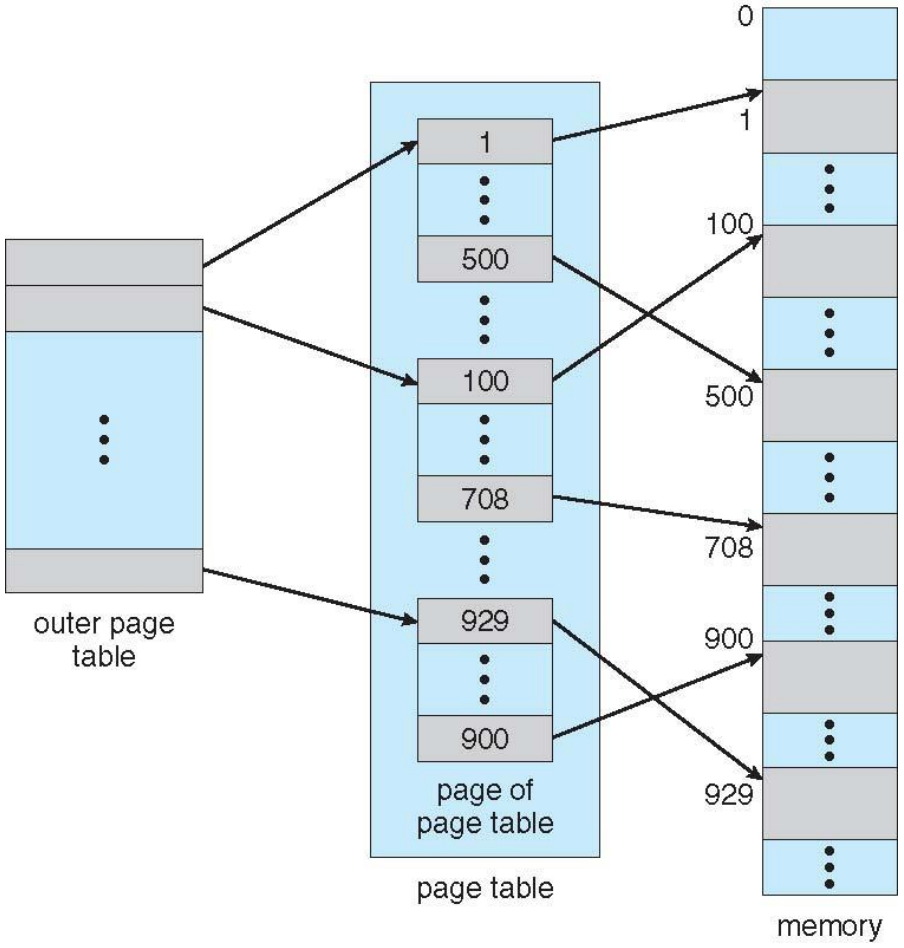
Structure of the Page Table

- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes -> 4 MB of physical address space / memory for page table alone
    - That amount of memory used to cost a lot
    - Don’t want to allocate that contiguously in main memory
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

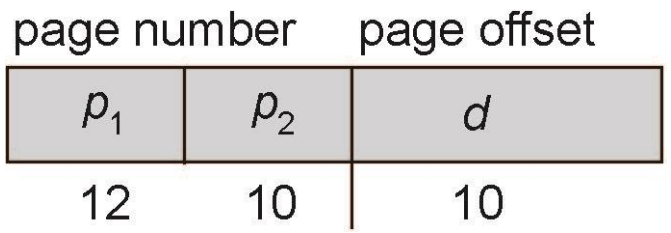
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



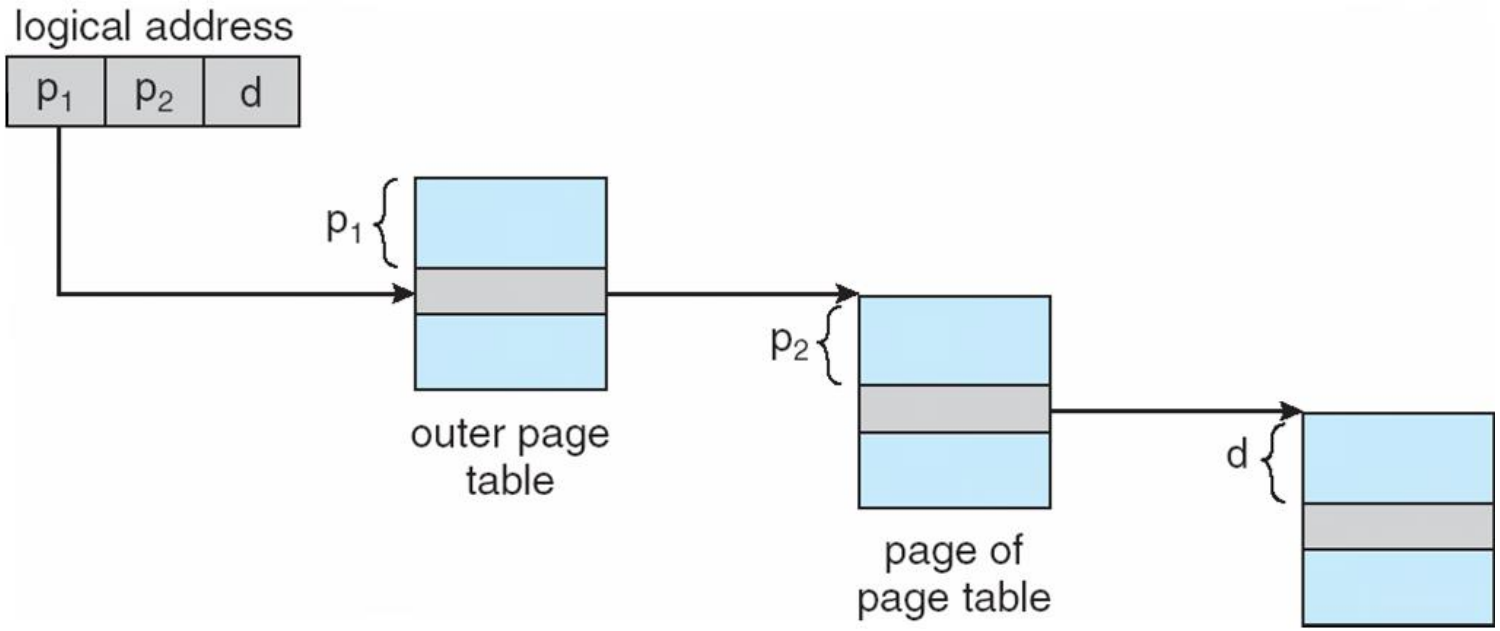
Two-Level Paging Example

- 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:



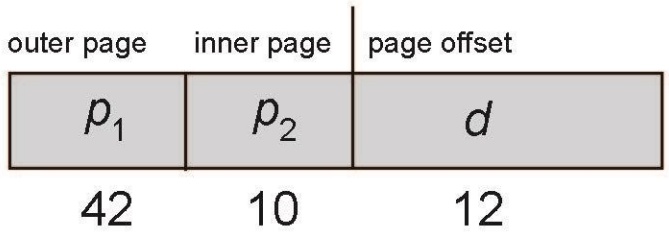
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation for a two-level 32-bit paging architecture



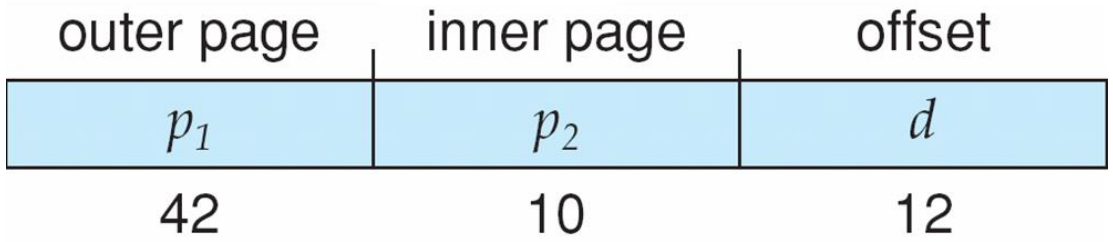
64-bit Logical Address Space

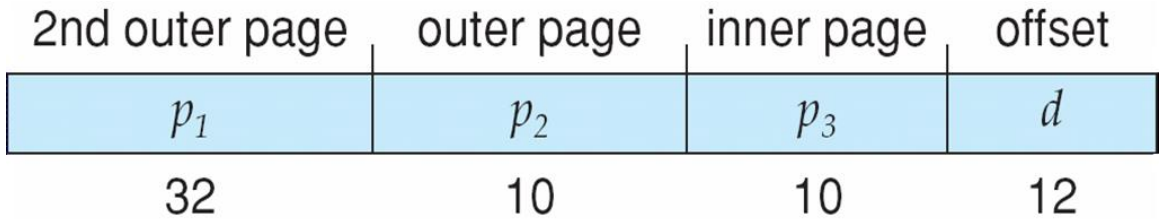
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look l''



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

Three-level Paging Scheme

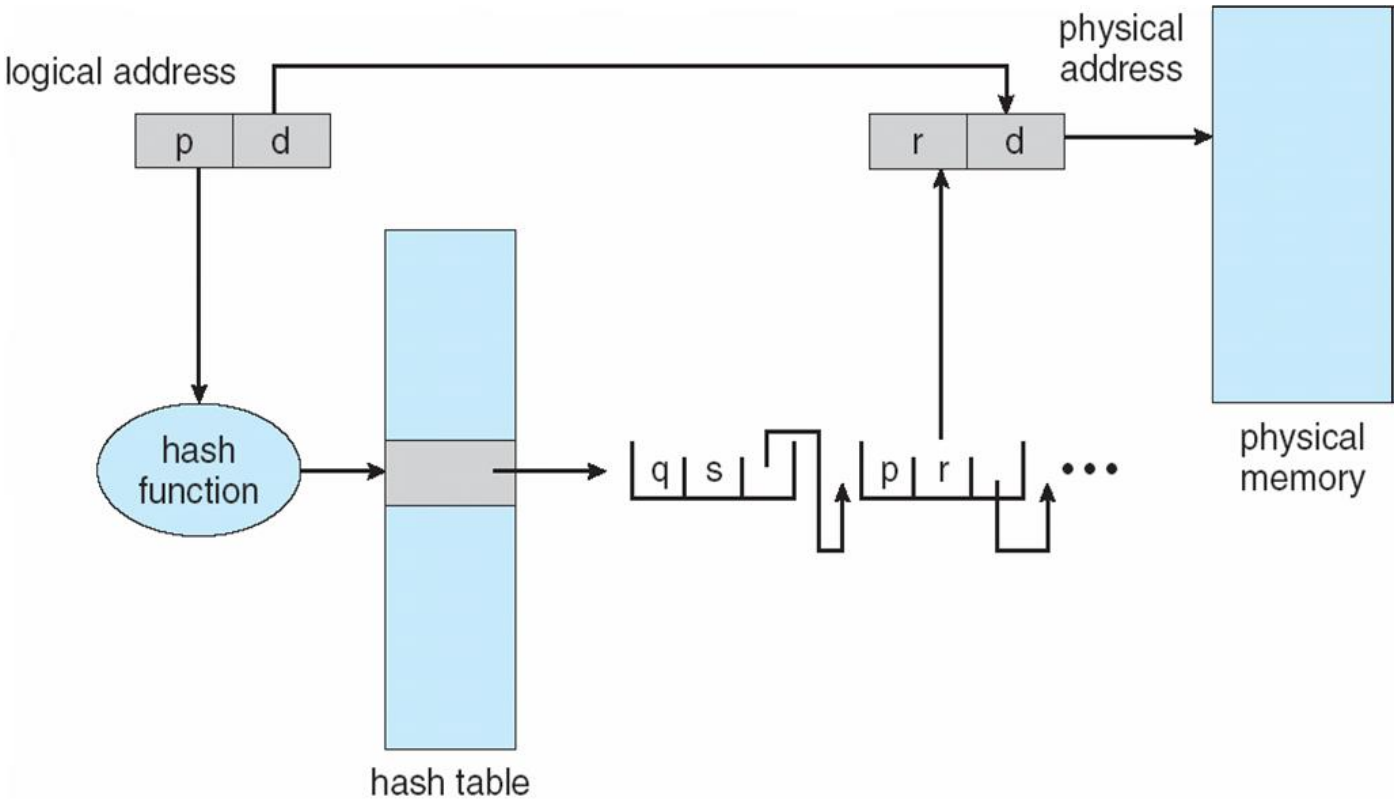




Hashed Page Tables

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

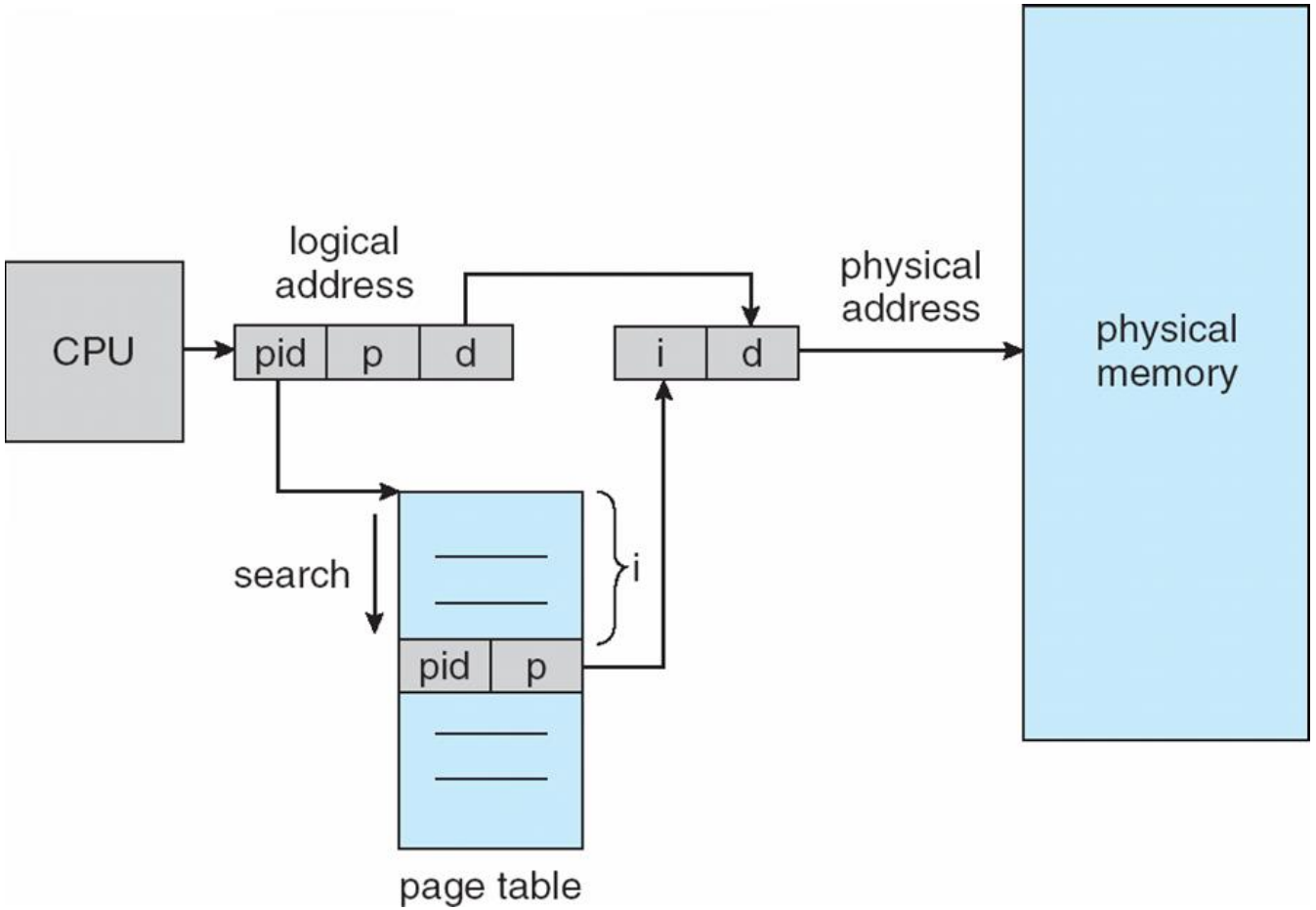
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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
- Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement shared memory?
  - One mapping of a virtual address to the shared physical address

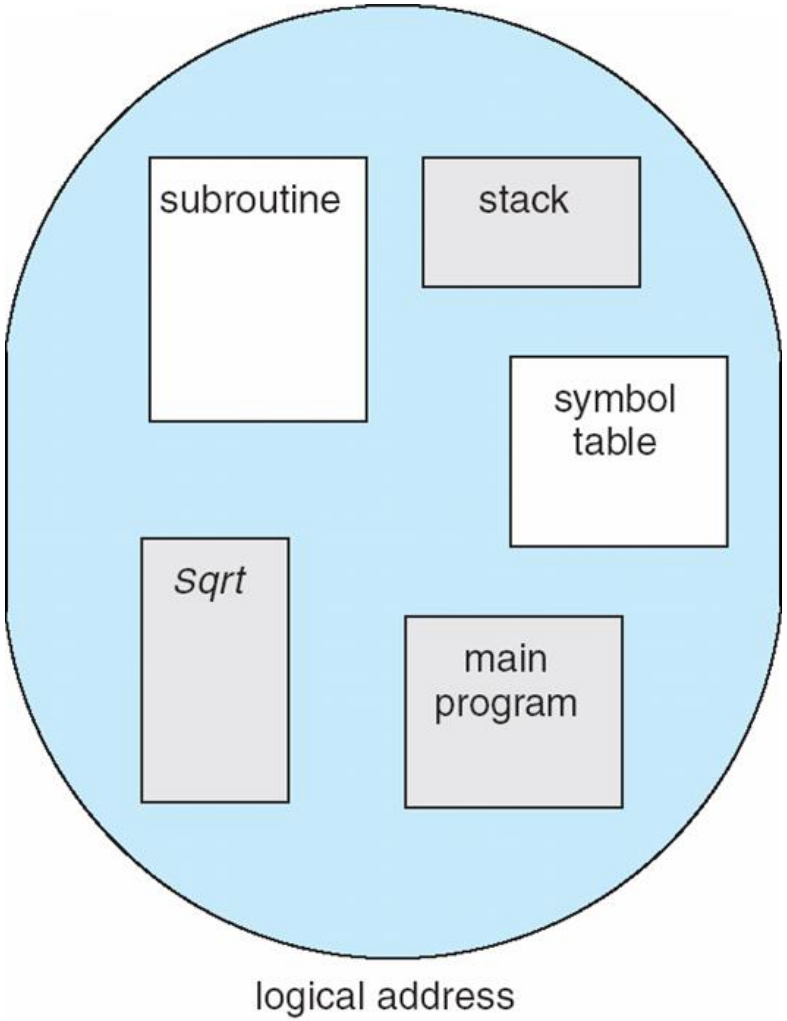
Inverted Page Table Architecture



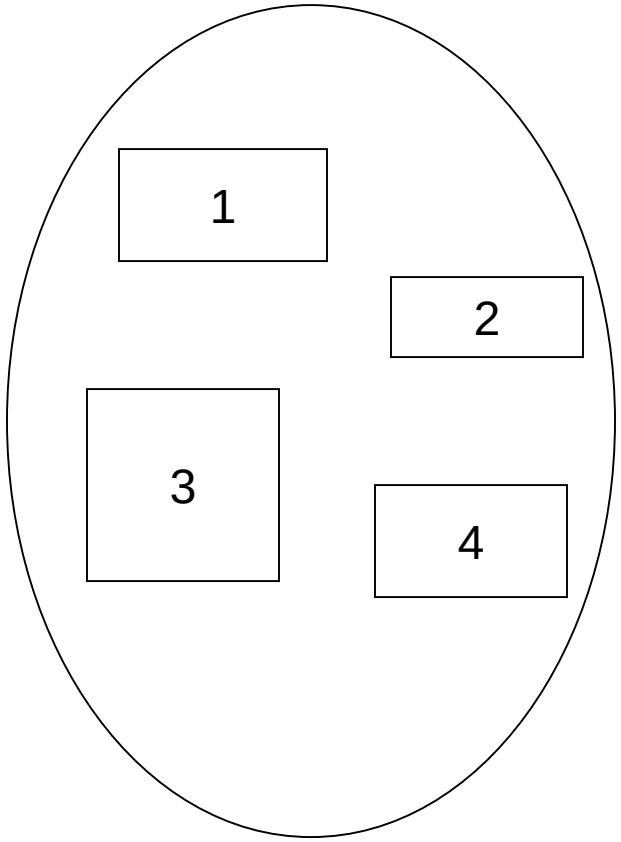
Segmentation

- In general, users view memory as a collection of variable-sized segments, with no necessary ordering among segments
- Segmentation is a memory management scheme that supports user's view of memory
  - Memory is viewed as a collection of variable-sized blocks called **segments**
- Each segment has a name and length
- A segment is a logical unit such as main program, Procedure/Function/Method, Object, Variables, Stack, Arrays, …
- Segmentation permits the physical address space of a process to be non-contiguous.

User's View of a Program



Logical View of Segmentation



user space

physical memory space

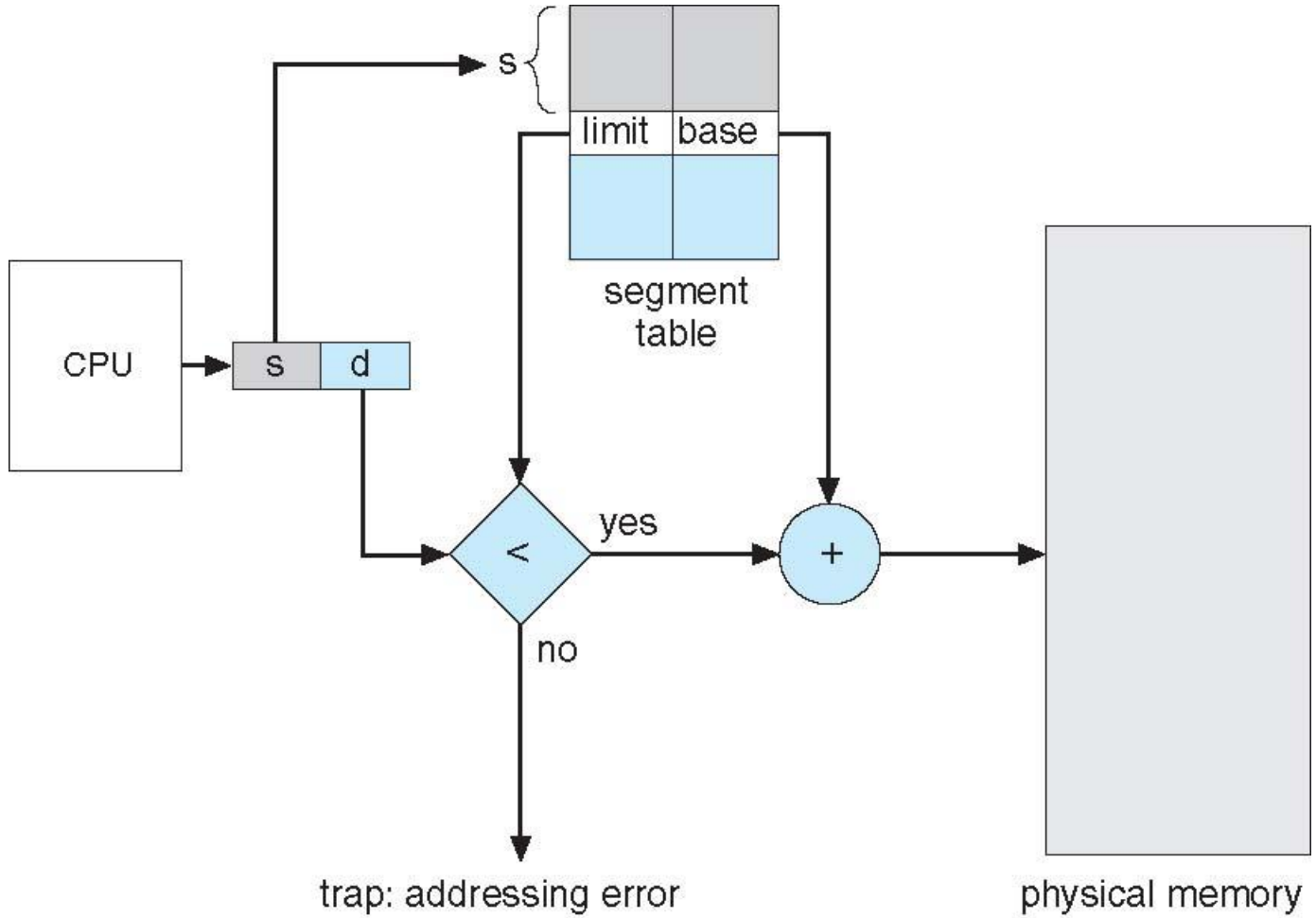
Basic Method

- A logical address space is a collection of segments
- The addresses specify both the segment name and the offset within the segment.
- The programmer specifies each address by two quantities: a segment name and an offset.
- Segments are numbered and are referred to by a segment number, rather than by a segment name.
- Logical address consists of a two tuple: **<segment-number, offset>**
- When a program is compiled, the compiler automatically constructs segments reflecting the input program
- Libraries that are linked in during compile time might be assigned separate segments.
- The loader would take all these segments and assign them segment numbers

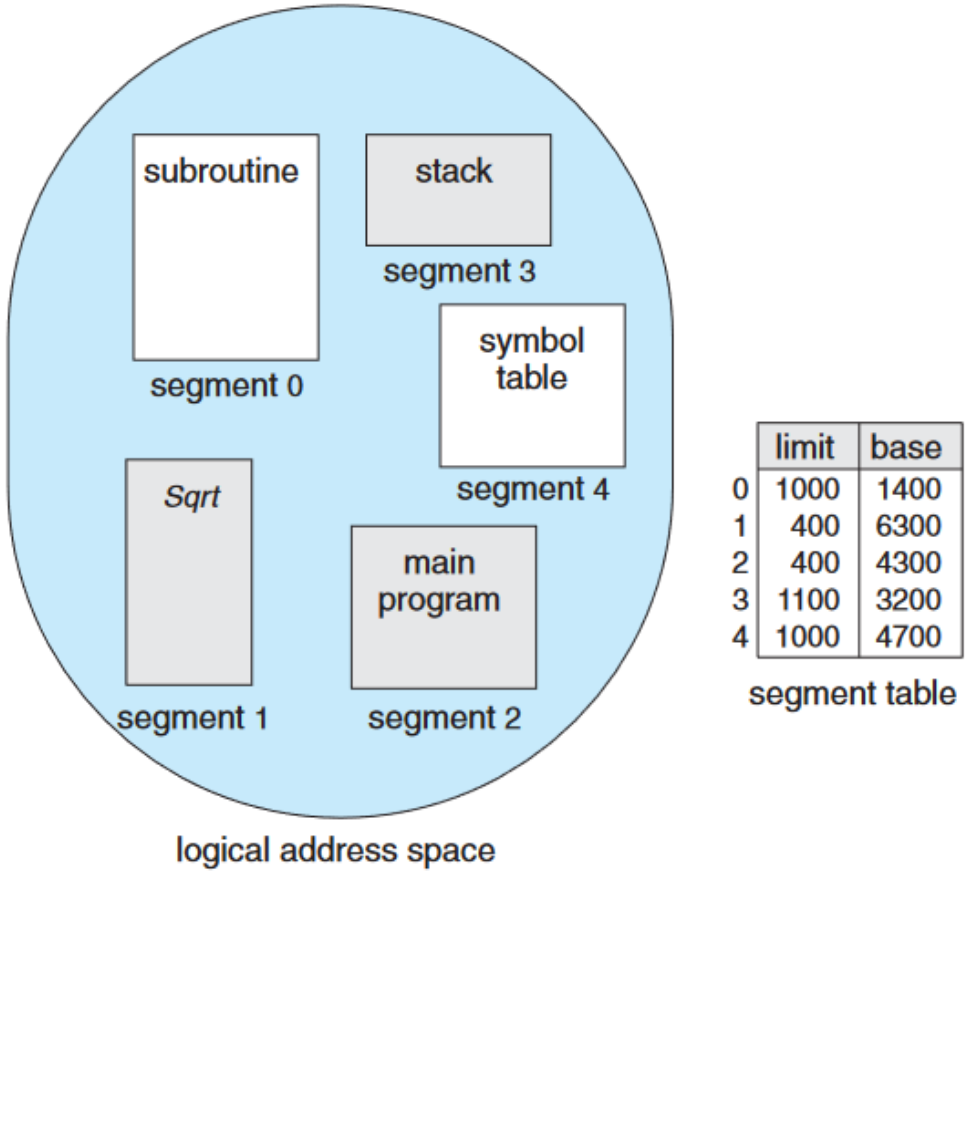
Segmentation Hardware

- The actual physical memory is a one- dimensional sequence of bytes.
- **Segment table** – maps two-dimensional user-defined addresses into one-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

Segmentation Hardware



Example of Segmentation



Segment Tables

- Segment table can be put either in fast registers or in memory
- Segment table kept in registers can be referenced quickly
- When a program consists of large number of segments, segment table is placed in memory
  - **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;
  - For a logical address check segment number ***s*** is legal if ***s*** < **STLR**→ add s to the STBR

Protection and Sharing

- 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

Practice

1. Consider a logical address space of 256 pages with a 4-KB page size, mapped onto a physical memory of 64 frames.
   a) How many bits are required in the logical address?
   b) How many bits are required in the physical address?
2. Consider a computer system with a 32-bit logical address and 4-KB page size. The system supports up to 512 MB of physical memory. How many entries are there in each of the following?
   a) A conventional, single-level page table
   b) An inverted page table

Practice…

- Consider the following segment table:

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

- What are the physical addresses for the following logical addresses?
  a) 0,430
  b) 1,10
  c) 2,500
  d) 3,400
  e) 4,112

Summary

- The various memory-management algorithms are contiguous allocation, paging, segmentation, and combinations of paging and segmentation.
- Selection of a memory-management scheme for a system depends on many factors, especially on the hardware design of the system.
- Systems with fixed-sized allocation units, such as the single-partition scheme and paging, suffer from internal fragmentation.
- Systems with variable-sized allocation units, such as the multiple-partition scheme and segmentation, suffer from external fragmentation.
- Another means of increasing the multiprogramming level is to share code and data among different processes.