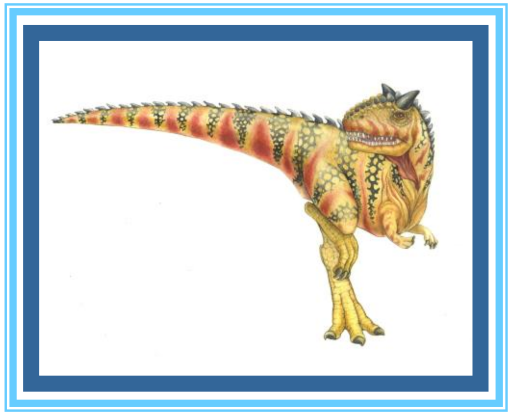
Chapter 8: Main Memory



Chapter 8: 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
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

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 that **CPU can access directly**
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Thus, Cache** sits between main memory and CPU registers to improve access speed.
- Not only speed but protection of memory is required to ensure correct operation,
- For proper system operation we must protect the **operating system** from access by **user processes and also processes must be protected from each other**
- This protection must be provided by the hardware

Background

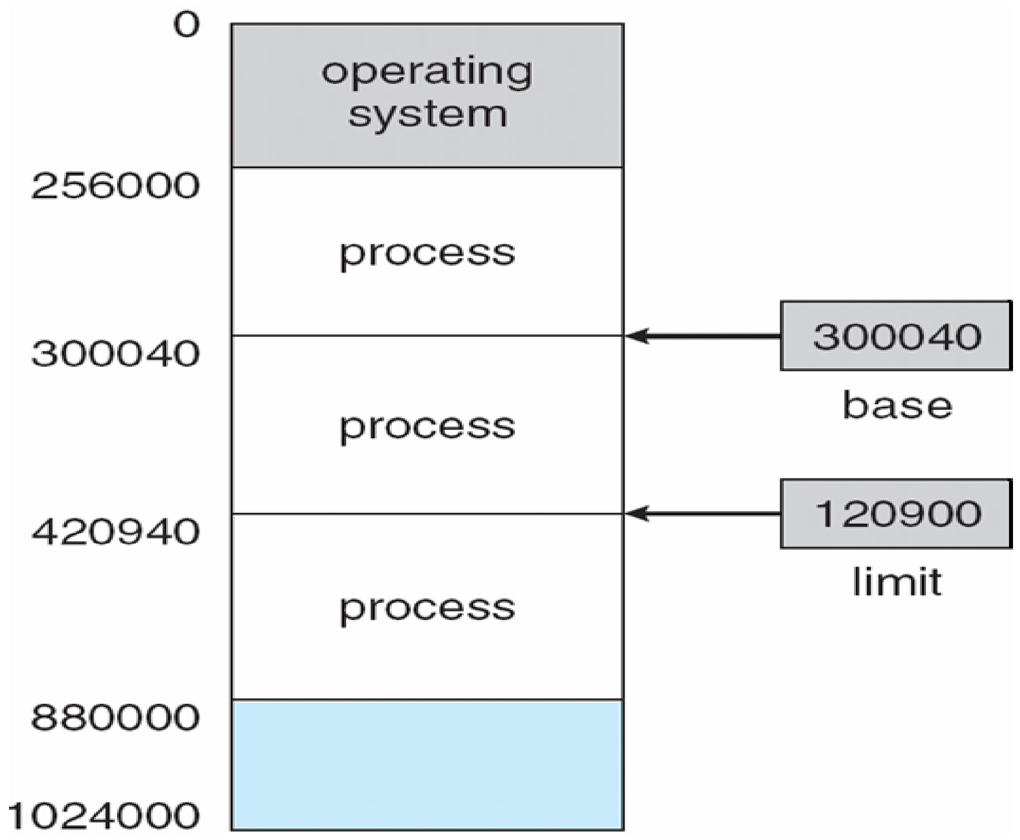
One way of implementing protection is:

- Each process has a **separate memory space**.
- Separate per-process memory space protects the processes from each other and **multiple processes can be** loaded in memory for concurrent execution
- Provide **two registers**, called **base and a limit** to determine the range of **legal addresses** that the process may access
- The **base register** holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

- **For example,** if **the base register holds 300040** and the **limit register** is **120900,** then the program can legally access all addresses from **300040 through 420939 (inclusive).**

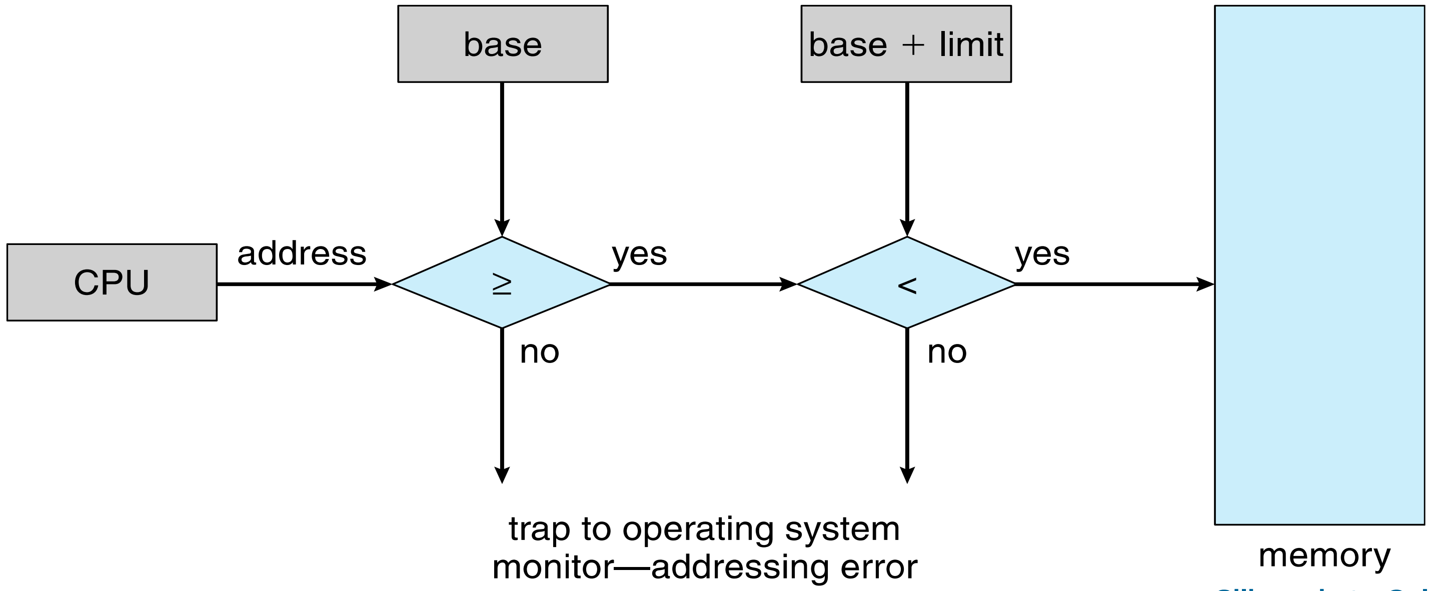
Background: Base and Limit Registers

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



Background: Hardware Address Protection

- CPU must check every memory access generated in **user mode** to be sure it is between **base and limit** for that user.
- The **base and limit** registers can be loaded only by the **operating system**, which uses a special privileged instruction.
- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system,



Address Binding

- Programs stored in secondary memory must be brought into main memory for execution.
- Most systems allow a user process to reside in any part of the physical memory
- Programs on disk, ready to be brought into memory to execute form an **input queue**
- User programs go through **many steps before execution**
- The first step done by OS in terms of memory management is **address binding**
- **Binding an address to data or instruction of a process is called address binding**

Address Binding

- CPU knows only the **actual address** of secondary memory
- Main memory address must be calculated for the process to execute.
- Thus it is needed to bind this main memory address with the instruction of the program for execution using address binding.
- Address binding can be done at three point of time

1) Compile time
2) Load time
3) Run time

Binding of Instructions and Data to Memory

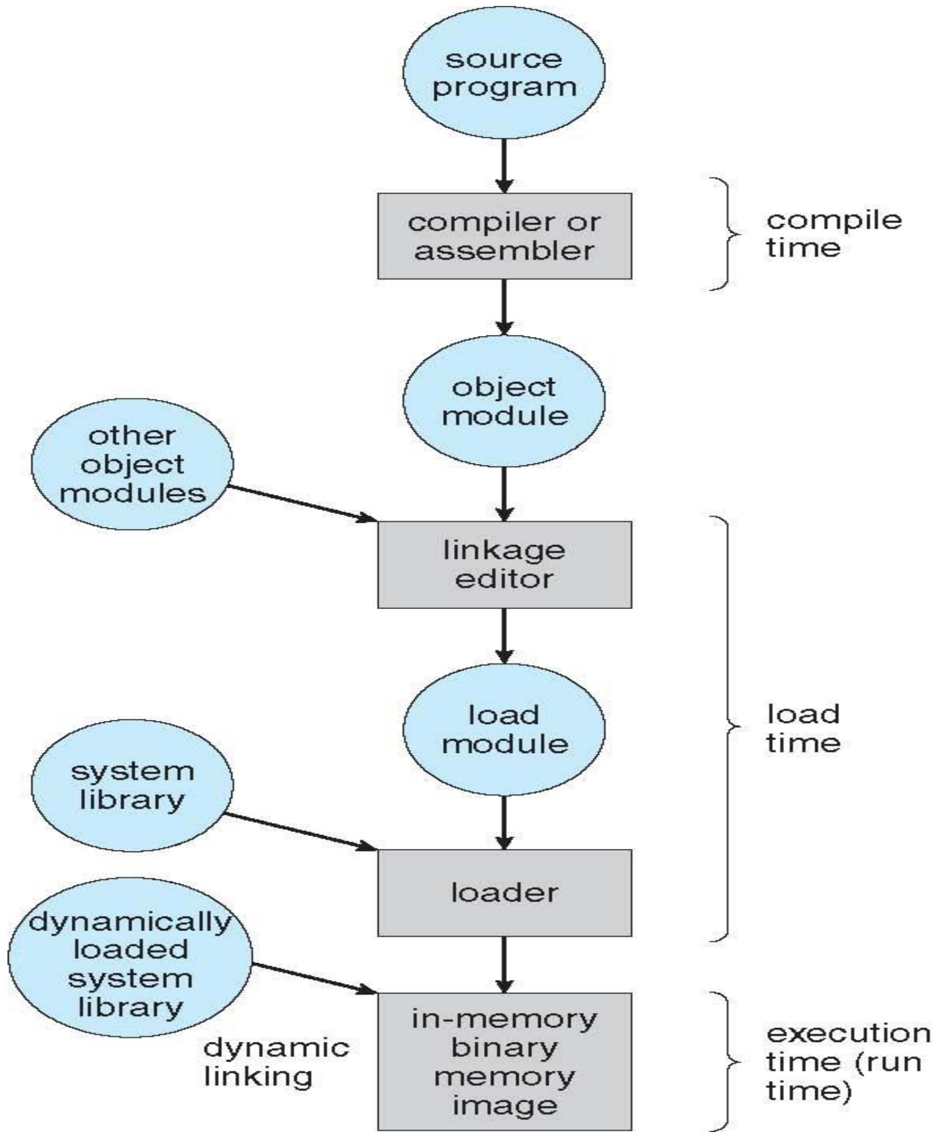
- Address binding of instructions and data to memory addresses can happen at **three different stages or times**

  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Compiler must generate **relocatable code** if memory location is not known at compile time. Here binding is done as **load 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)

Address Binding

- Further, addresses can be represented in different ways at different stages of a program's life
  - Source code addresses are usually **symbolic(variables)**
  - Compiler **binds symbolic address** 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

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

The process of **address binding** requires **2 types of address:**

- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit

- **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

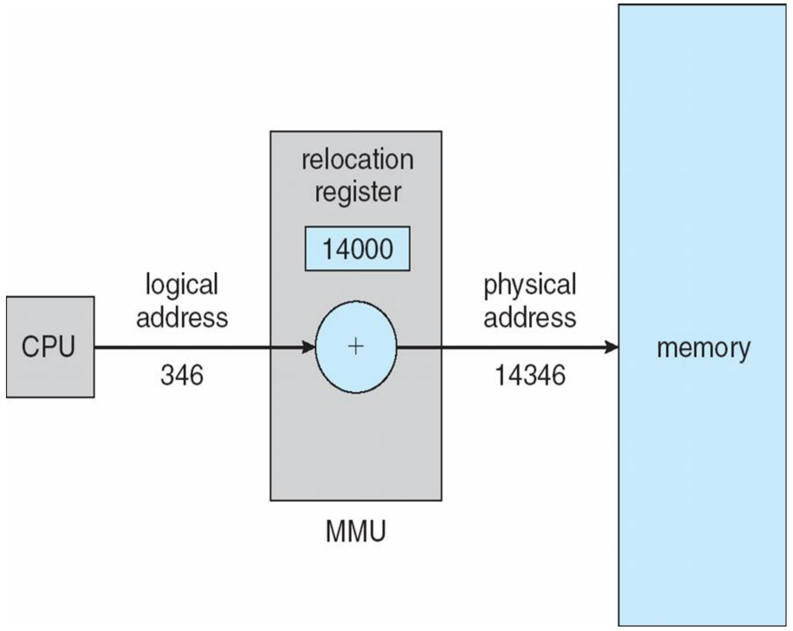
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes;
- Logical and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

- The translation from **logical addresses** to **physical addresses** is managed by the operating system's **memory management system.**
- The value in the **relocation register(base register)** is added to every address generated by a user process at the time it is sent to memory
- The user program deals with ***logical* addresses**; it never sees the *real* physical addresses

Dynamic Loading

- Routine is not loaded until it is called
- Better **memory-space utiliza**tion; 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



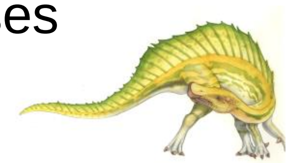
Dynamic relocation using a relocation register

Dynamic Linking

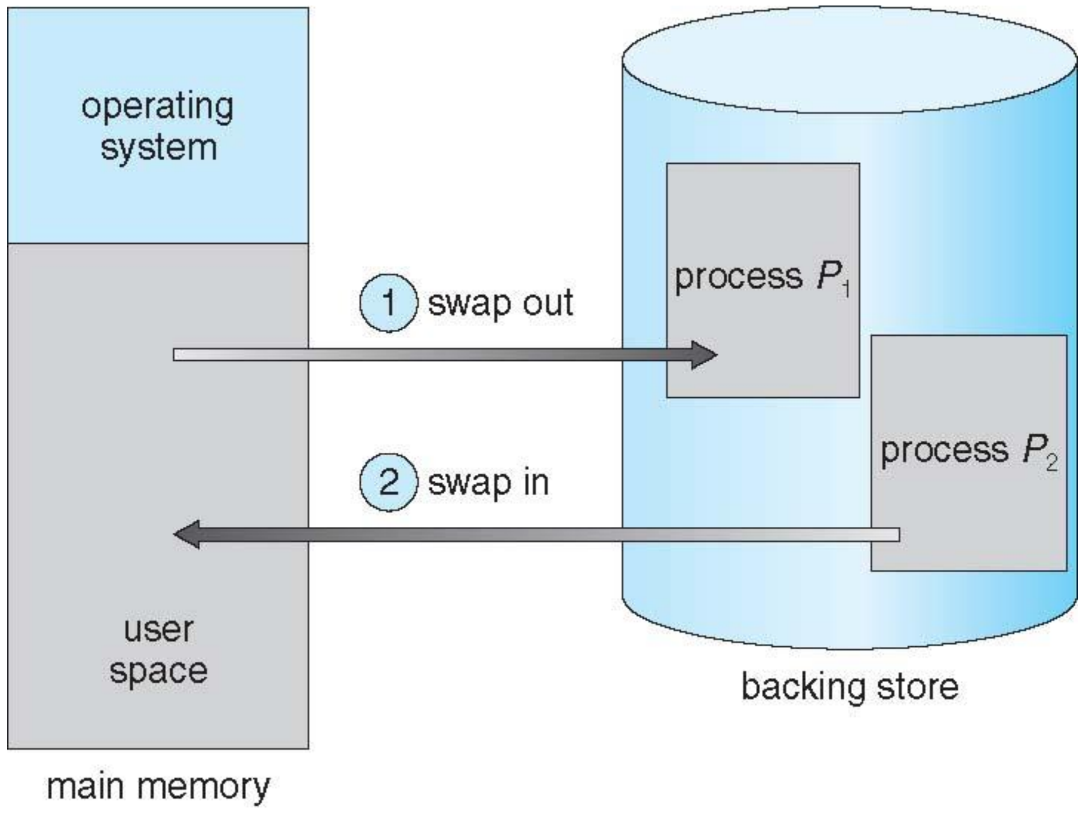
- **Static linking** – system libraries and program code combined by the loader into the binary program image
- Dynamic linking –linking postponed until execution time
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address 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 also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

Swapping

- A process can be **swapped** temporarily out of memory to a backing store (disk), and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed real physical memory
  - Swapping increases **degree of multiprogramming**
- **Dispatcher** is used to select a new process whenever CPU decides to execute a new process.
- If next processes is not in memory, dispatcher swap out a process in memory and swap in desired process.
- **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
- System maintains a **ready queue** of **ready-to-run** processes which have memory images on disk



Schematic View of Swapping

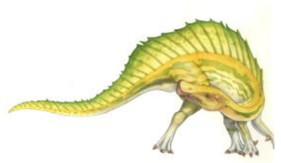


Context Switch Time including Swapping

- Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped
- **Context switch time** can then be very high in swapping system
- Consider a 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap to or from main memory takes 100 MB/50 MB/sec= 2000 ms (2 Sec) of swap time
  - Plus swap in of same sized process
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if we reduce size of memory swapped – by knowing how much memory really **being used** rather than it **might be using**
  - System calls to inform OS of memory use can be used via `request_memory()` and `release_memory()`

Context Switch Time and Swapping (Cont.)

- Other constraints on swapping are
  - Process must be **completely idle** for swapping
  - Process with pending I/O can't swap – can't swap out as I/O would occur to wrong process
  - Execute I/O always to kernel (OS) space, then to I/O device
    - Then transfer between OS buffer and process memory occurs only when process is swapped in.
    - Known as **double buffering**, adds overhead
- Standard swapping not used in modern operating systems
  - But modified version common
    - Swap only when free memory extremely low



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
  - But started if more than threshold amount of memory allocated
  - Disabled again once memory demand reduced below threshold
  - Swapping only portion of process and not entire process to decrease swap time.

Swapping on Mobile Systems

- Not typically supported due to following constraints
  - **Flash memory** is used
    - Small amount of space
    - Limited number of write supported by flash memeory
    - 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 next

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**:
  - One for resident **operating system,** usually held in low memory with interrupt vector
  - Next **user processes** then held in high memory
  - Each process contained in **single contiguous** section of memory

Contiguous Allocation (Cont.)

Memory Protection

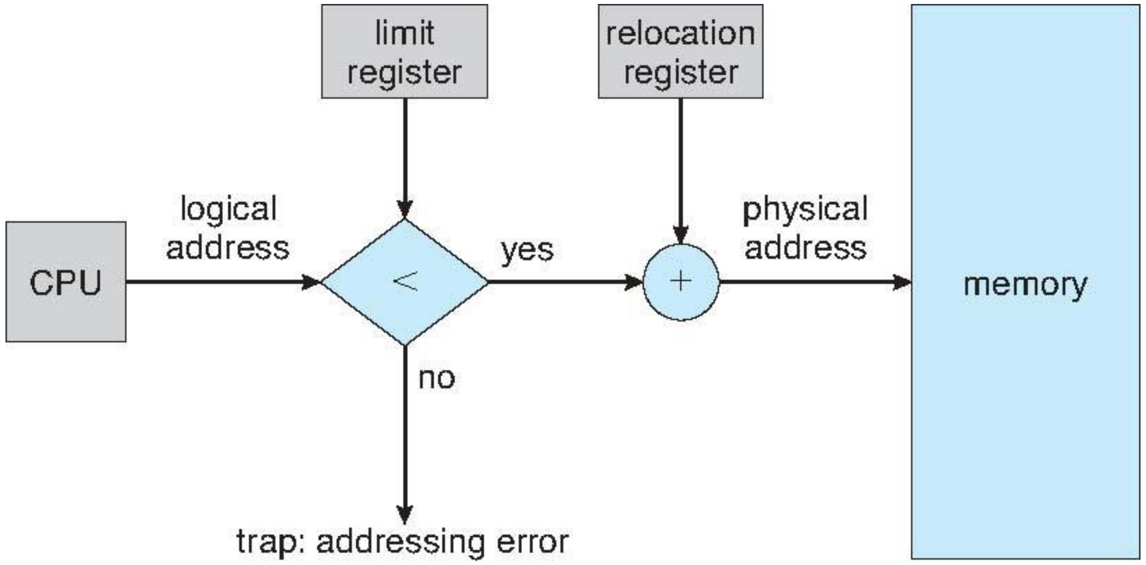
- **Relocation registers** used to protect user processes from each other, and from changing operating-system code and data
  - **Base register** contains value of smallest physical address
  - **Limit register** contains range of logical addresses – each logical address must be less than the limit register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

Hardware Support for Relocation and Limit Register

Contiguous Allocation (Cont.)

Memory Protection

- When the CPU scheduler selects a process for execution, the **dispatcher loads the relocation and limit registers** with the correct values
- Every address generated by a CPU is checked against these registers,
- Thus, protect both the operating system and the other users' programs and data from being modified by this running process



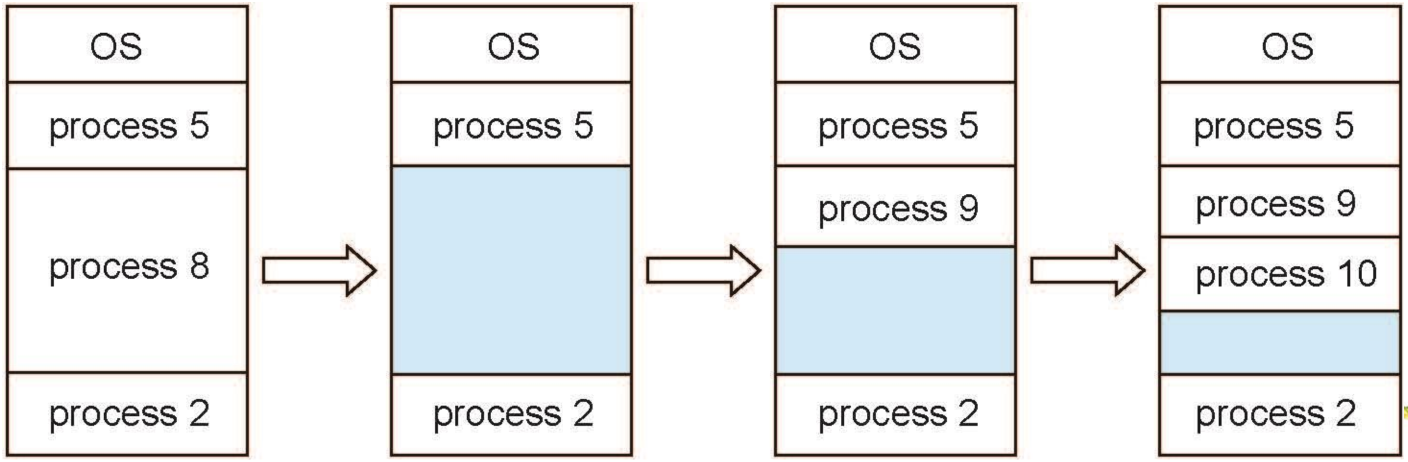
Hardware Support for Relocation and Limit Register

Memory allocation

- **Multiple-partition allocation**
  - One of the simplest method for allocating memory is to divide memory into several **fixed-sized partitions**.
  - Each partition may contain exactly one process.
  - Degree of **multiprogramming** limited by **number of partitions**
  - 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.
  - This method is no longer used.

Memory allocation

- **Variable-partition allocation**
  - **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 table for information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

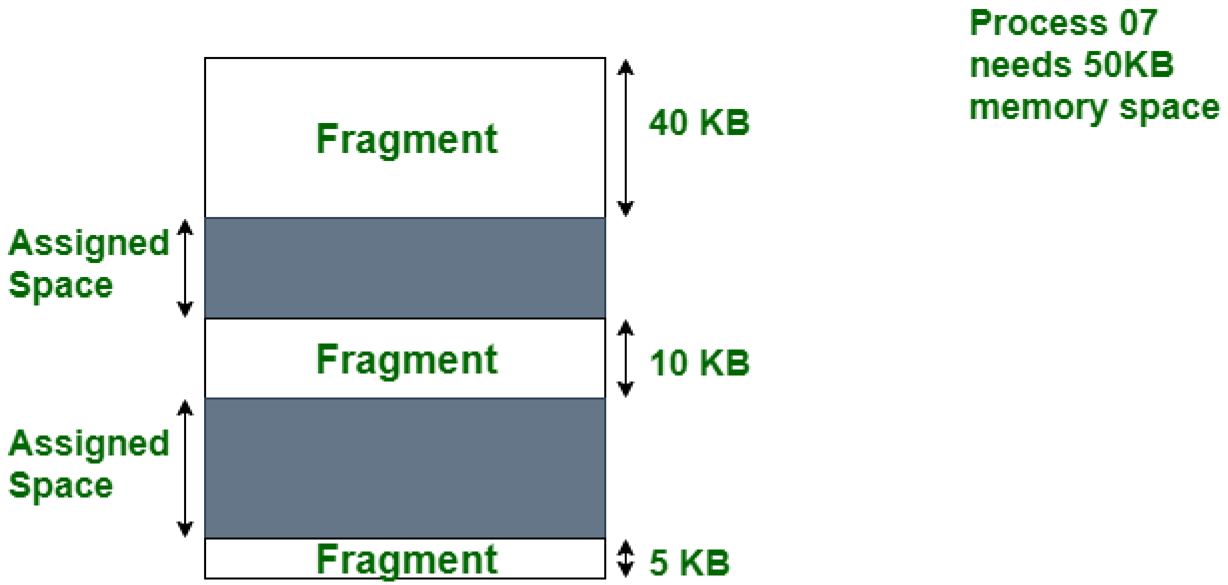
How to satisfy a request of size ***n*** from a list of free holes?

- **First-fit**: Allocate the ***first*** hole that is **big enough**. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large 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.
- Whether you apply a first-fit or best-fit memory allocation strategy it'll cause external fragmentation
- In the diagram, we can see that, there is enough space (55 KB) to run a process-07 (required 50 KB) but the memory (fragment) is not contiguous.



Fragmentation (Cont.)

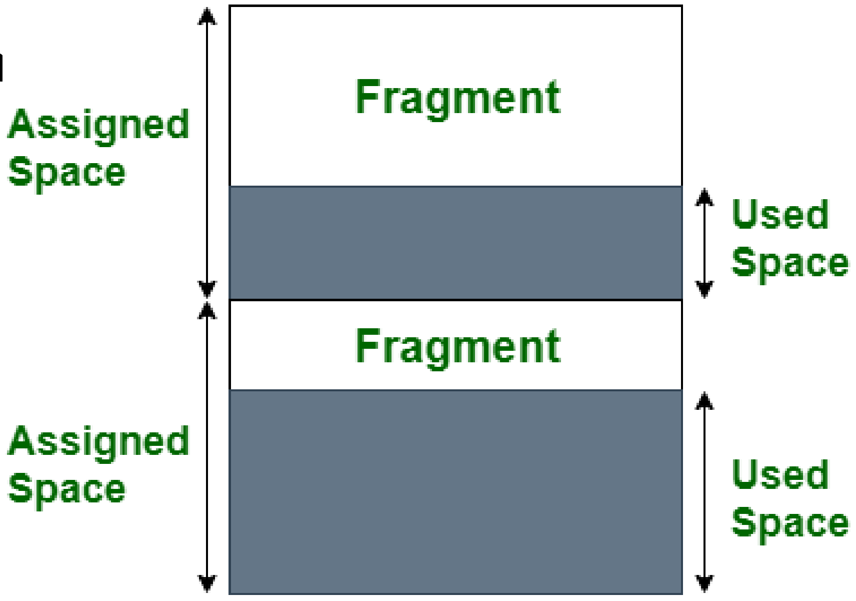
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time

- 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.

✓ **Segmentation and Paging** is used

Fragmentation

- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory;
- Fragmentation because the difference between memory allocated and required space or memory is called **internal fragmentation.**
- The size difference is memory internal to a partition, but not being used
- Internal fragmentation happens wh mounted-sized blocks.



Internal Fragmentation

Segmentation

- Is a memory-management scheme that supports **user view of memory**
- A **logical address space** is a collection of segments
- Each segment has **a name and a length**.
- The logical addresses specify both the **segment name** and the **offset** within the segment
- A program is a collection of segment : is a logical unit such as:

  main program

  procedure

  function

  method

  object

  local variables, global variables

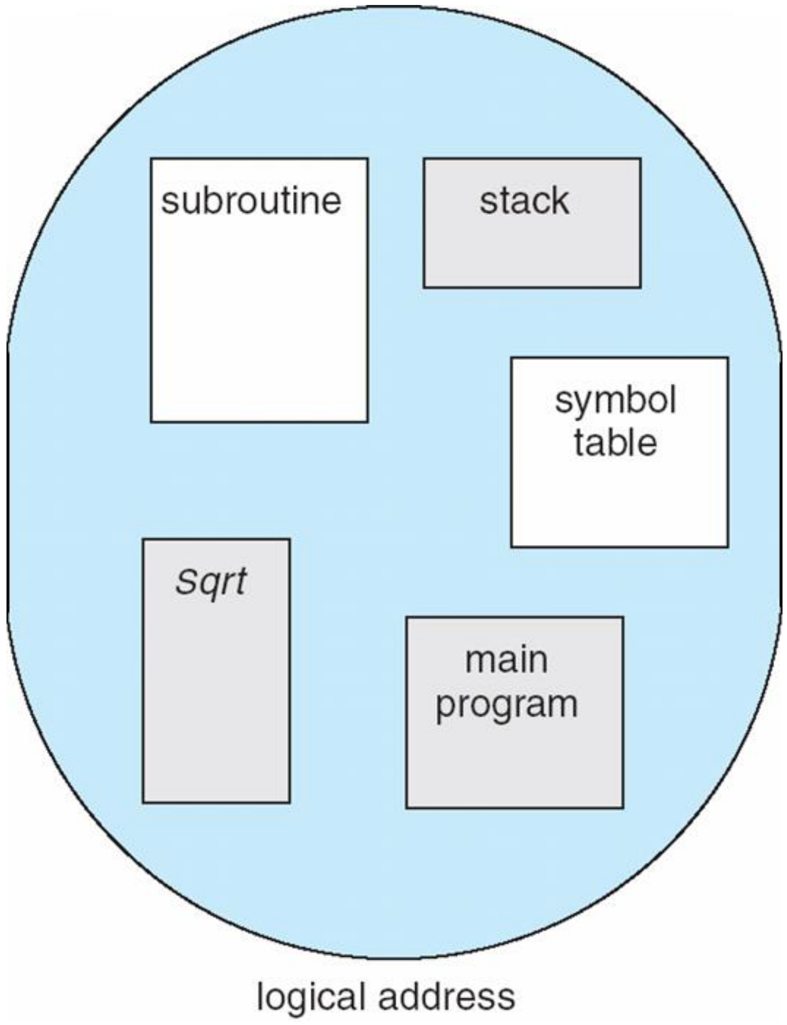
  common block

  stack

  symbol table

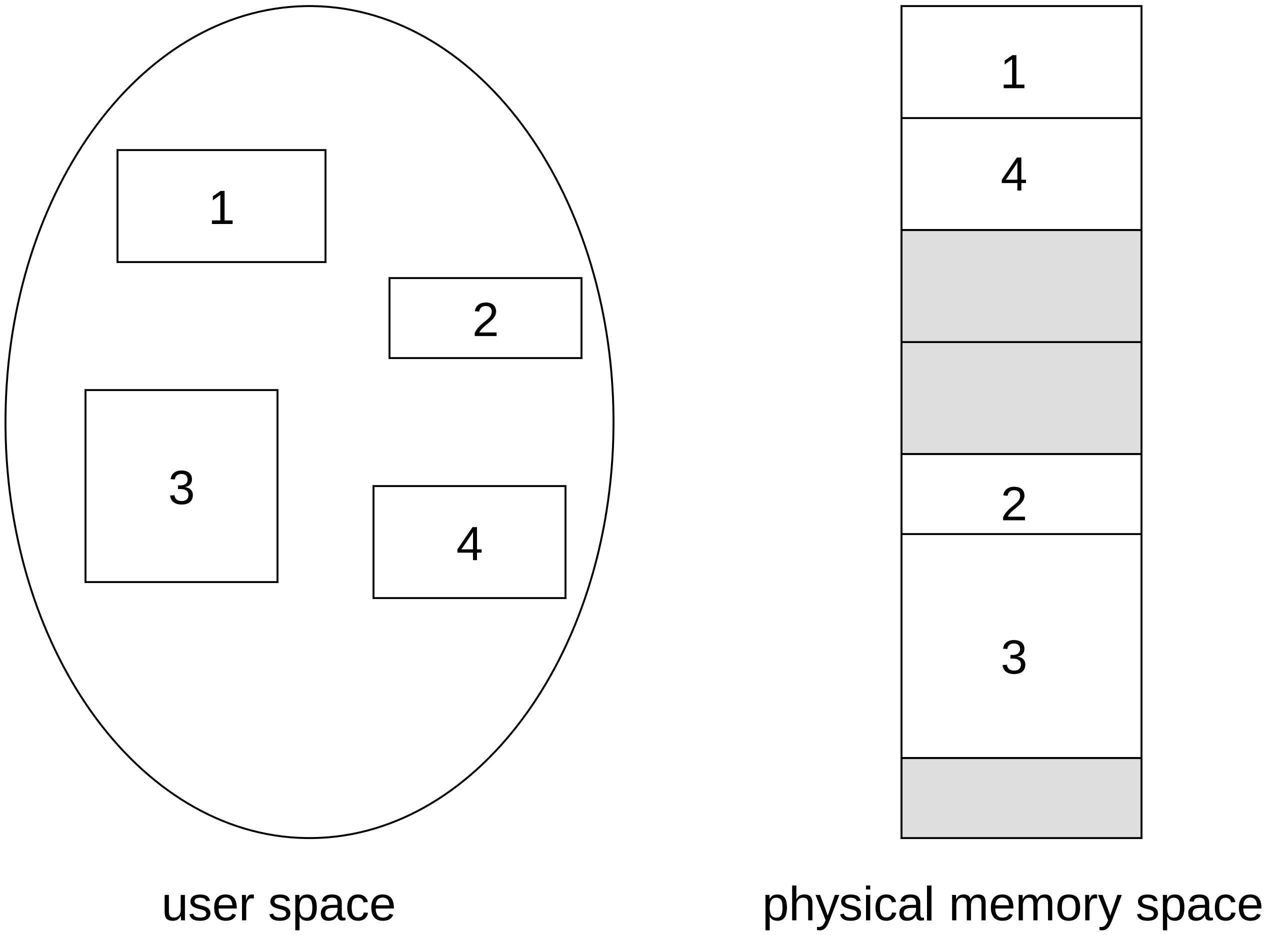
  arrays

User's View of a Program



- Normally, 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.

Logical View of Segmentation



Segmentation Architecture

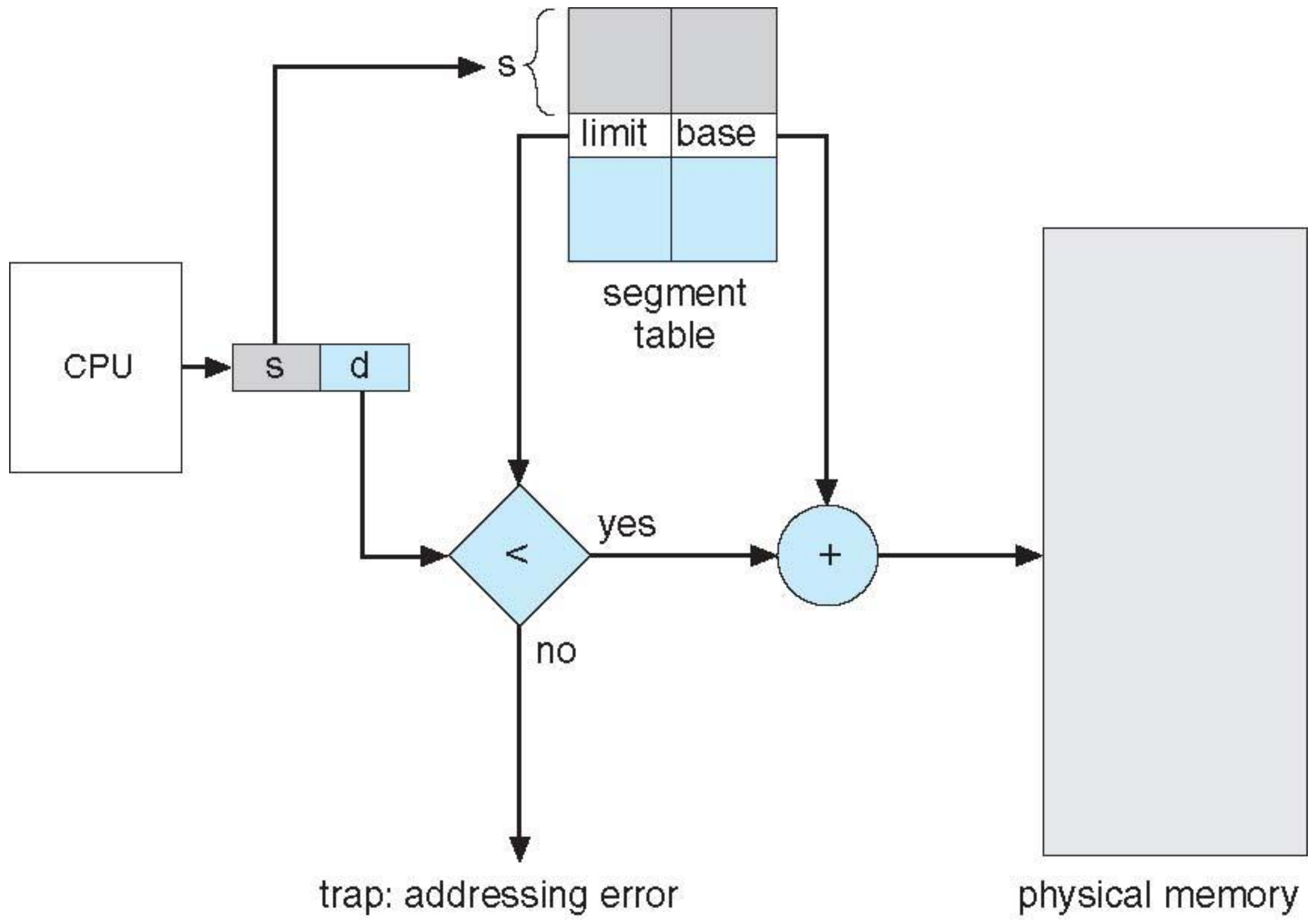
- Logical address consists of a **two tuple:**

  <segment-number, offset>,

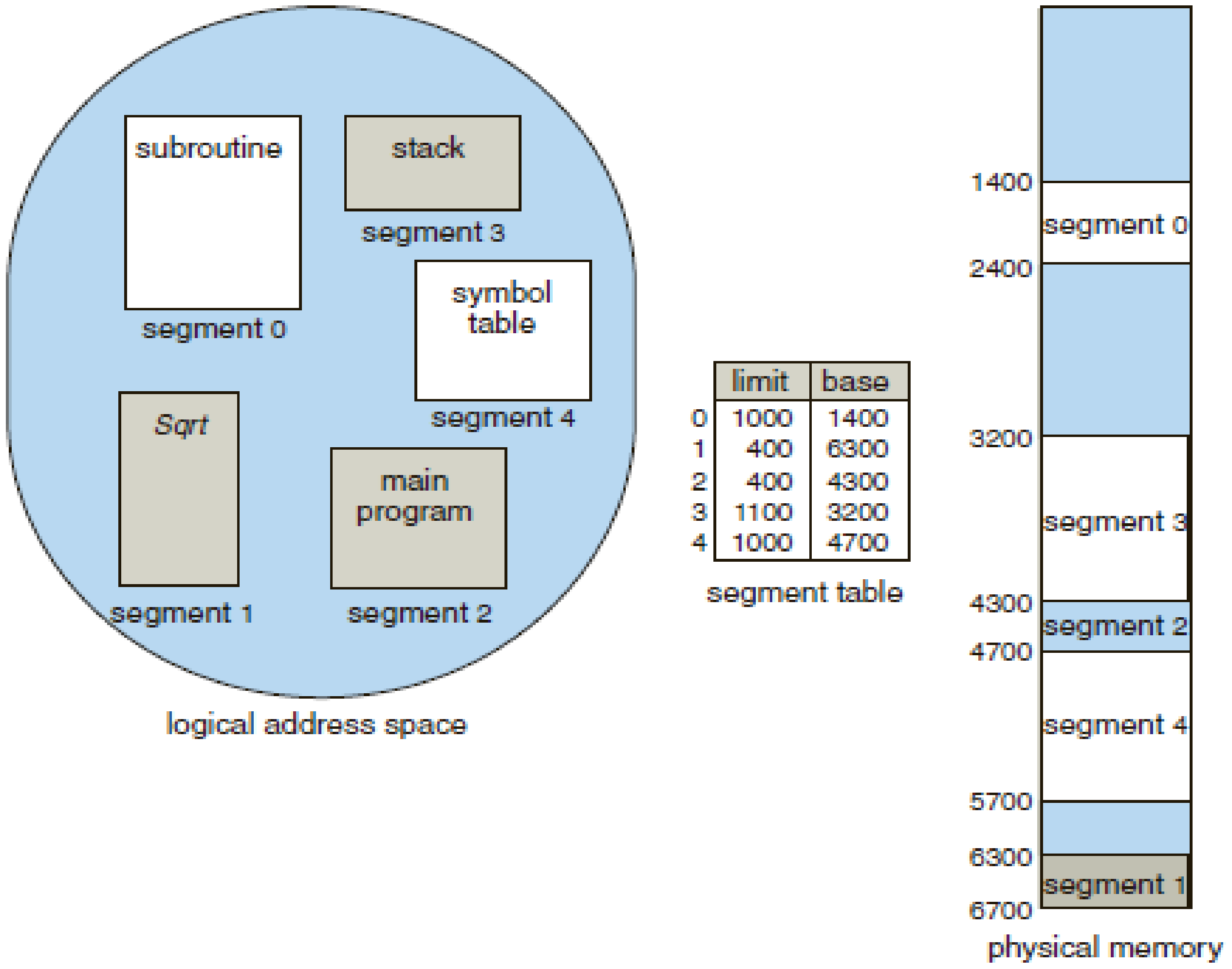
- **Segment table** – maps two-dimensional physical addresses to one-dimension physical address; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** 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 Hardware



Segmentation Example



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

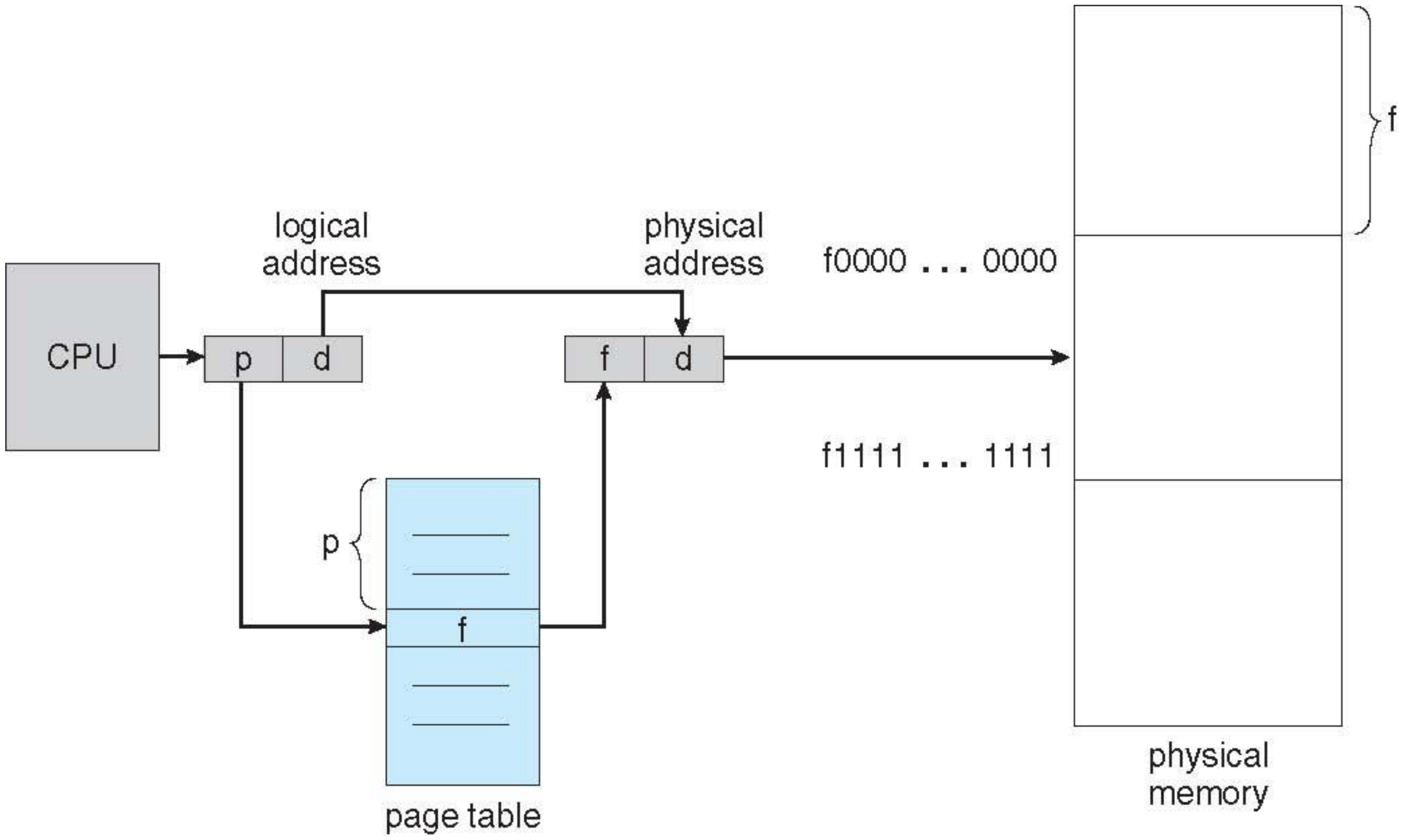
Paging

- Paging is another m**emory management** scheme that eliminates the need for a **contiguous allocation** of physical memory.
- Avoids external fragmentation

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, size is defined by hardware**
- Keep track of all free frames
- When a process is to be executed, its pages are loaded into any available memory frames
- The backing store is divided **into fixed-sized blocks** that are the same size as the **memory frames** or clusters of multiple frames

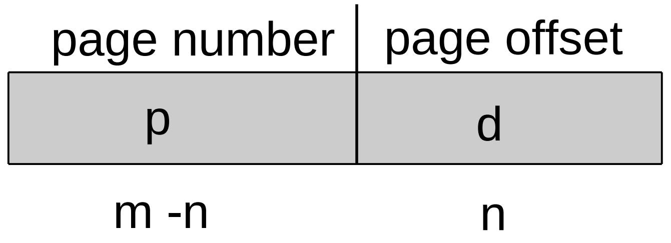
Paging Hardware



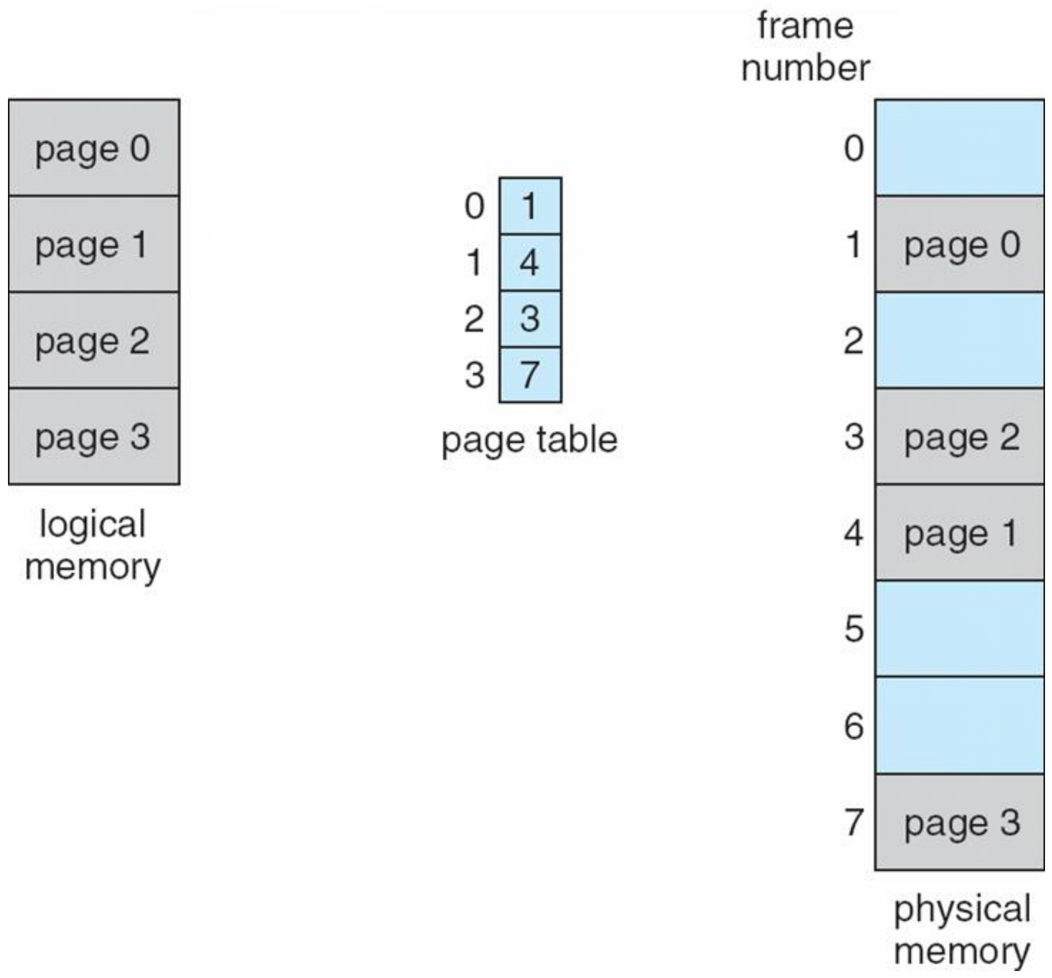
Address Translation Scheme

- Address generated by CPU is divided into 2 parts:
  - **Page number** (***p***) – used as an index into a **page table.** The **page table** 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,
- The high-order ***m−n* bits** of a logical address designate the **page**
- **Number(p),** and the ***n* low-order** bits designate the **page offset(d)**.
- Thus, the logical address is as follows:

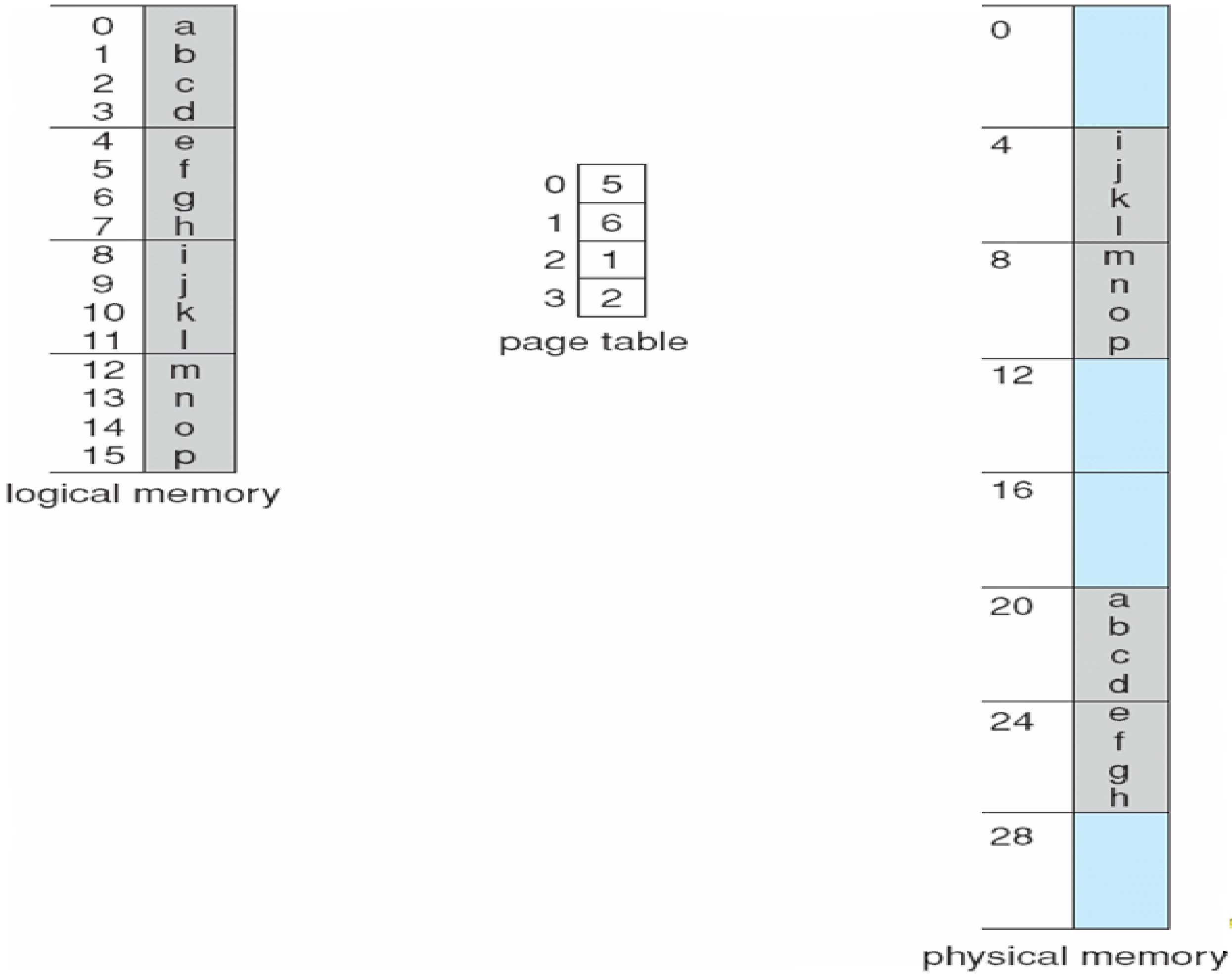


Paging Model of Logical and Physical Memory



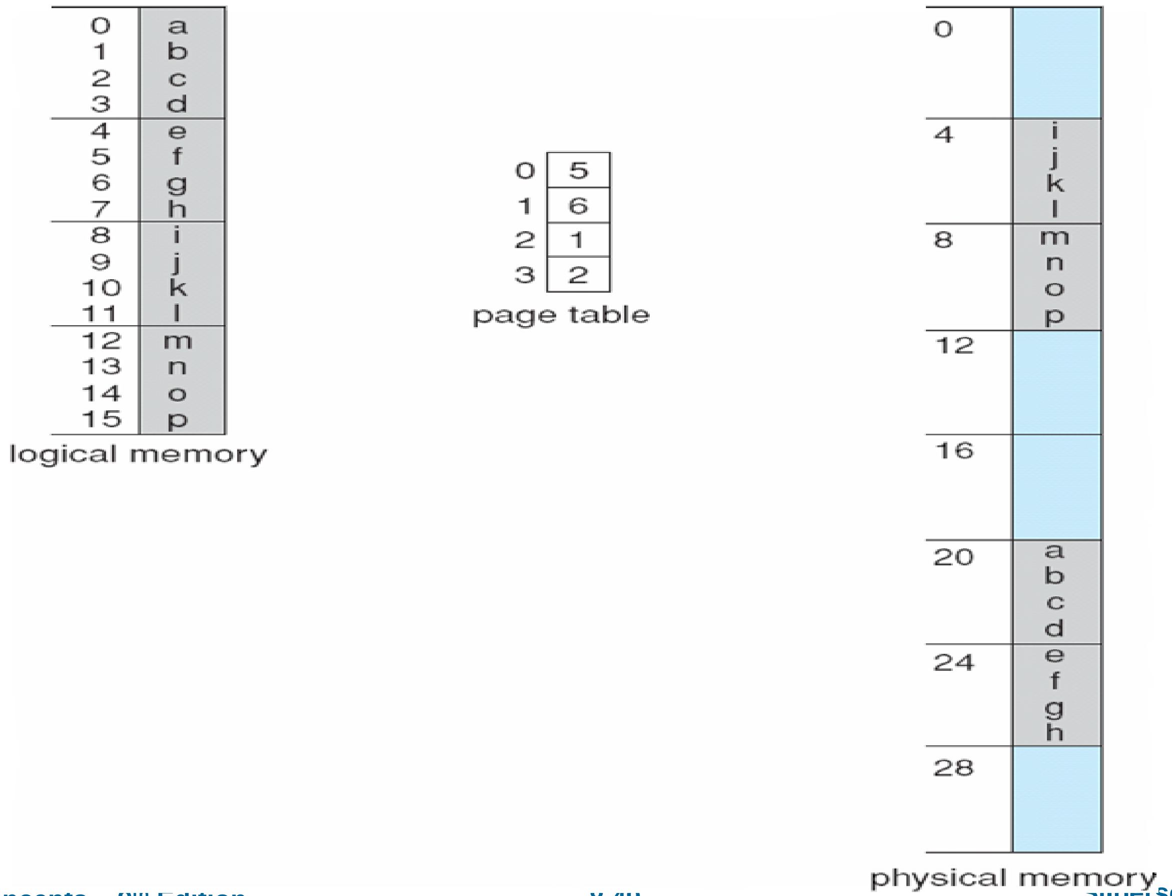
Paging Example

Consider the example with logical address, ***n*=2 and *m*=4**. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages),



Paging Example

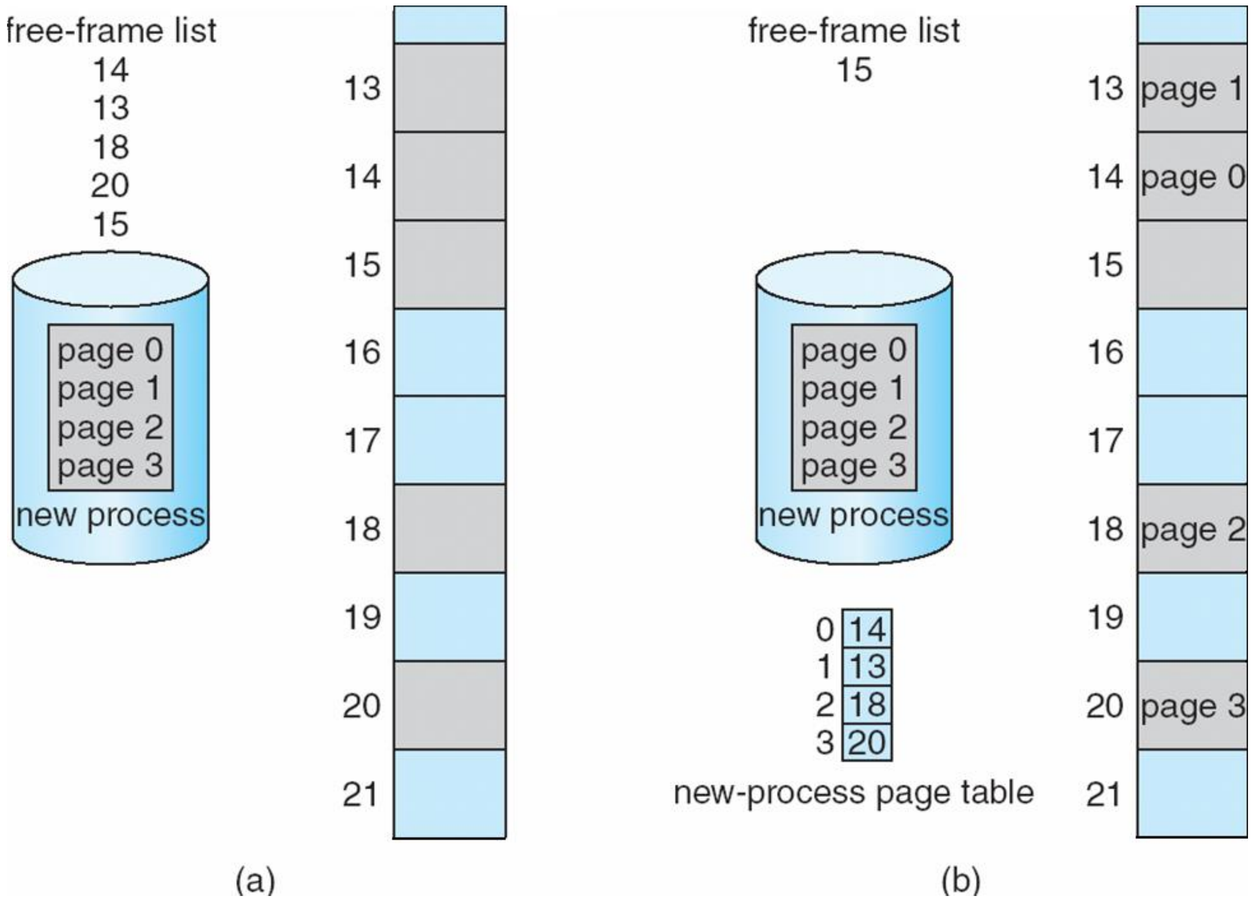
- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5.
- Thus, logical address 0 maps to physical address 20 [= (5 × 4) +0]



Paging (Cont.)

- Paging may have some **internal fragmentation**. If the memory requirements of a process do not happen to coincide with page boundaries, the last frame allocated may not be completely full, leading to internal fragmentation.
- Consider
  - Page size = 2,048 bytes, Process size = 72,766 bytes
  - The process needs 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
- In the worst case, a process would need *n* pages plus 1 byte, resulting in internal fragmentation of almost an entire frame.
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But overhead is involved in each page-table entry
  - Page sizes growing over time
    - Solaris supports two page sizes – 8 KB and 4 MB

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

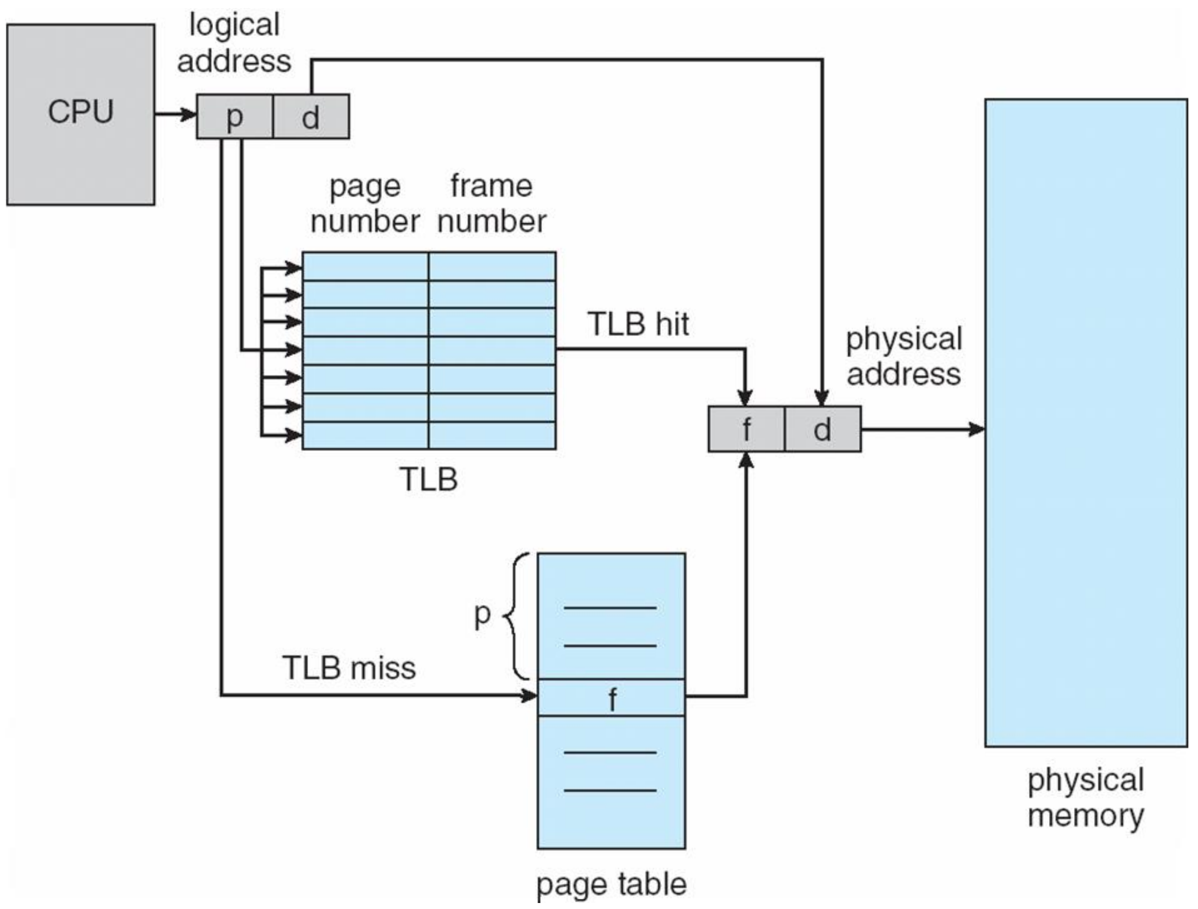
Associative Memory

- 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

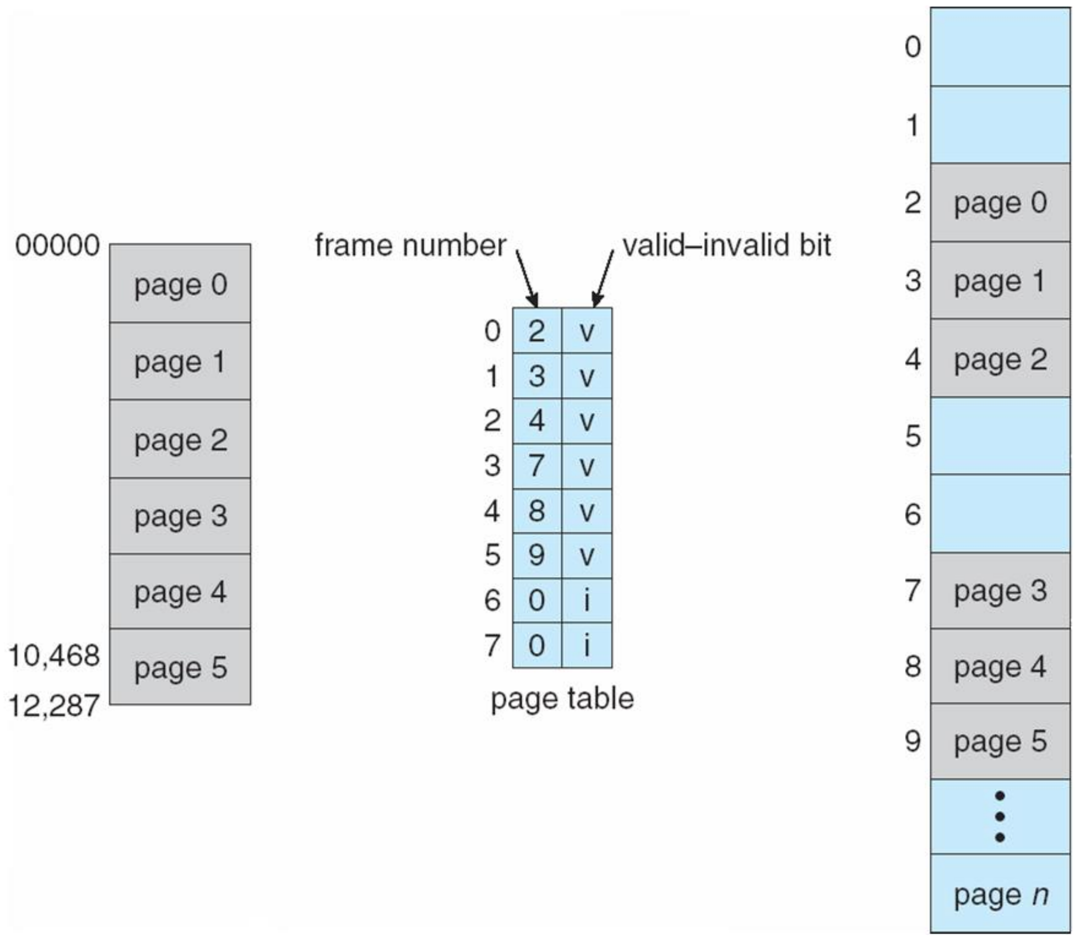
- Associative Lookup = ε time unit
  - Can be < 10% of memory access time
- Hit ratio = α
  - Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Consider α = 80%, ε = 20ns for TLB search, 100ns for memory access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

  EAT = 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**)
- Any violations result in a trap to the kernel

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

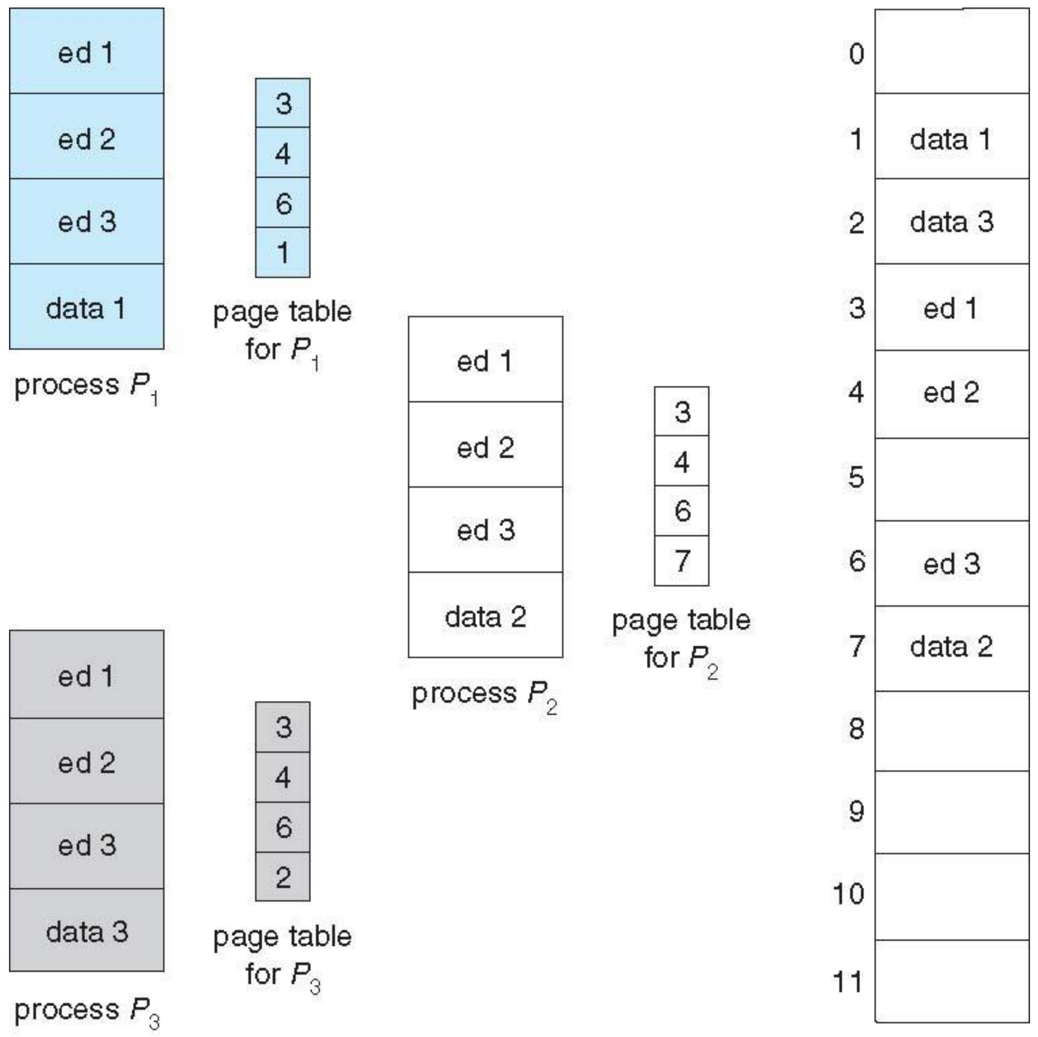


Shared Pages

Another advantage of paging is possibility of **sharing**

- **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



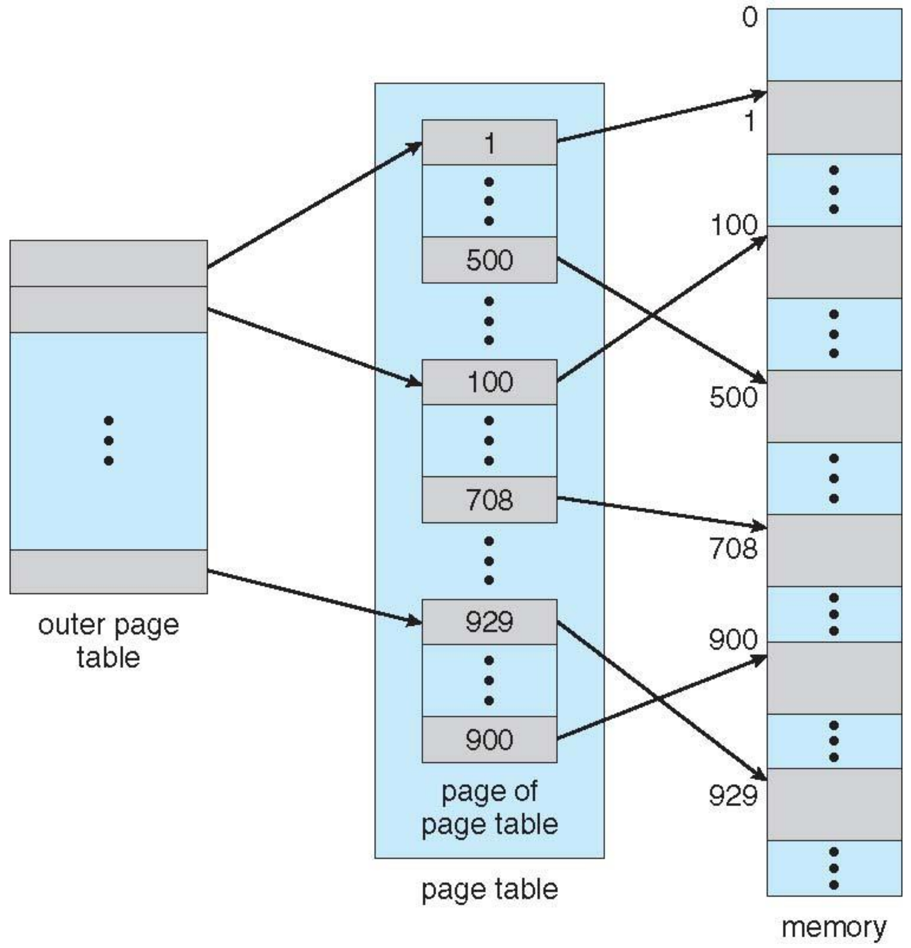
Structure of the Page Table

- 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

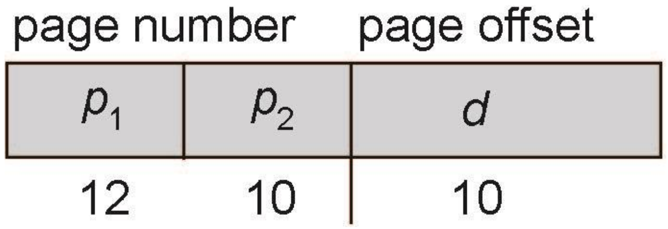
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



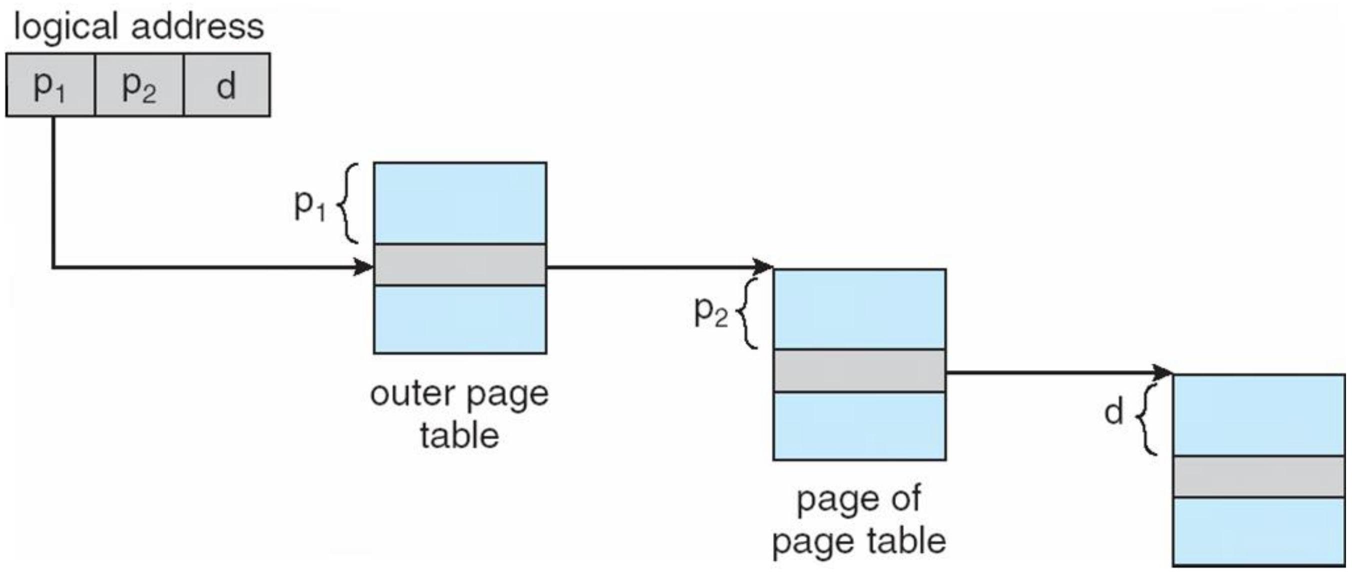
Two-Level Paging Example

- 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:



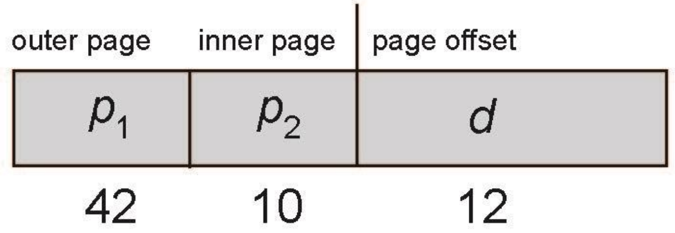
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



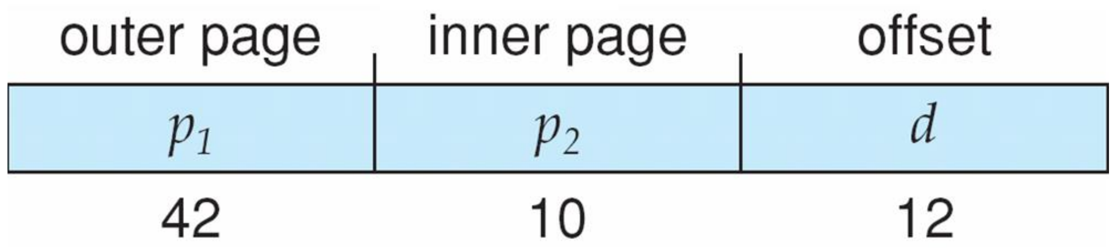
64-bit Logical Address Space

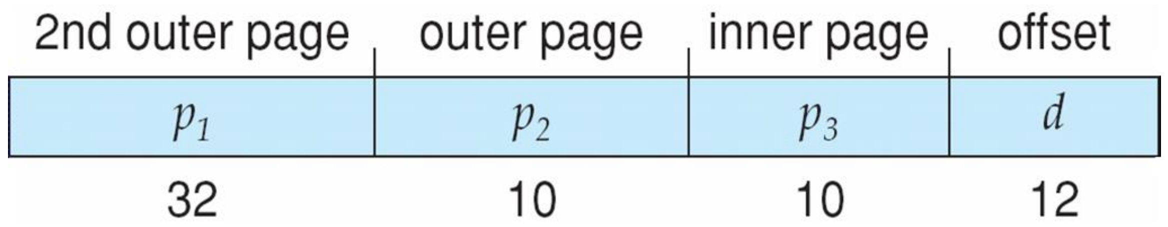
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd 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

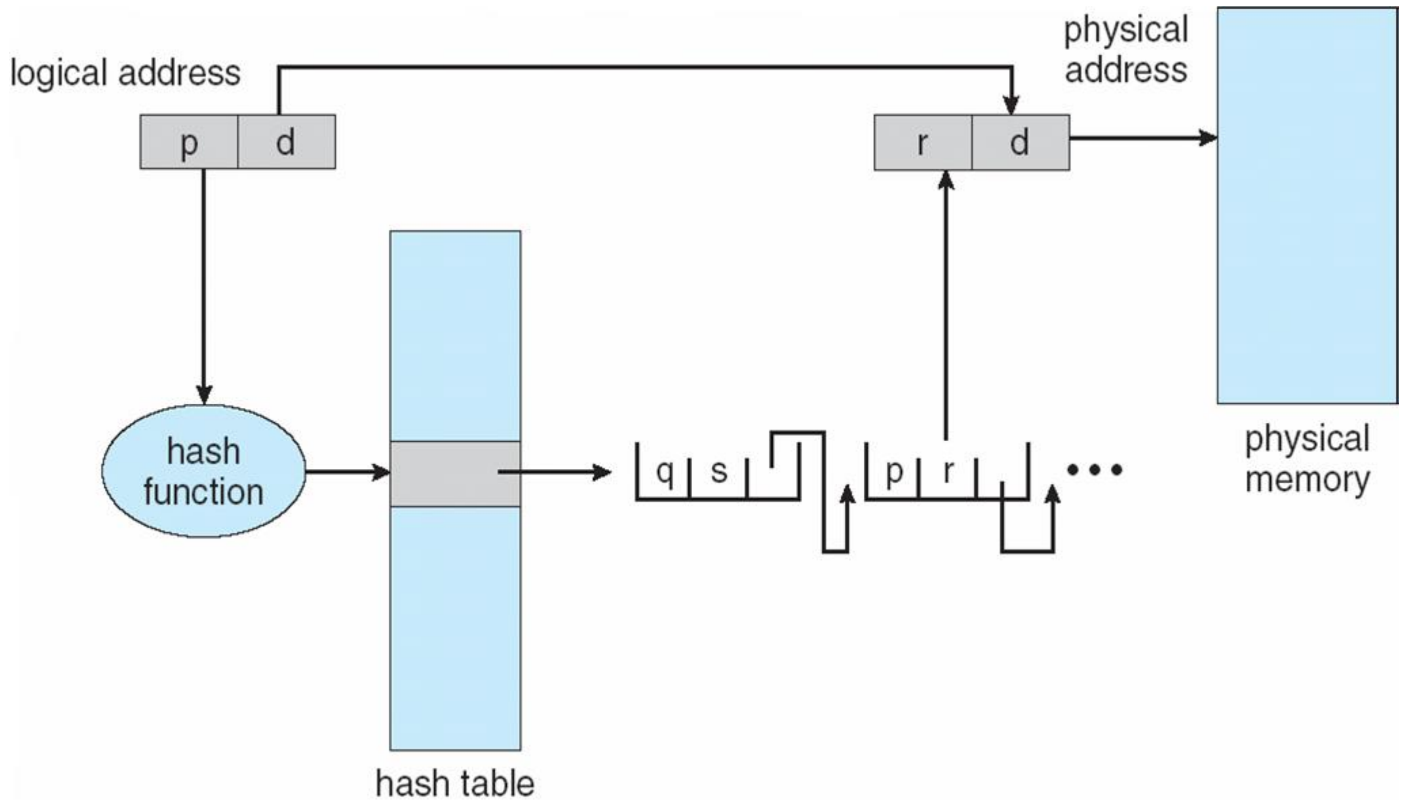




Hashed Page Tables

- 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)

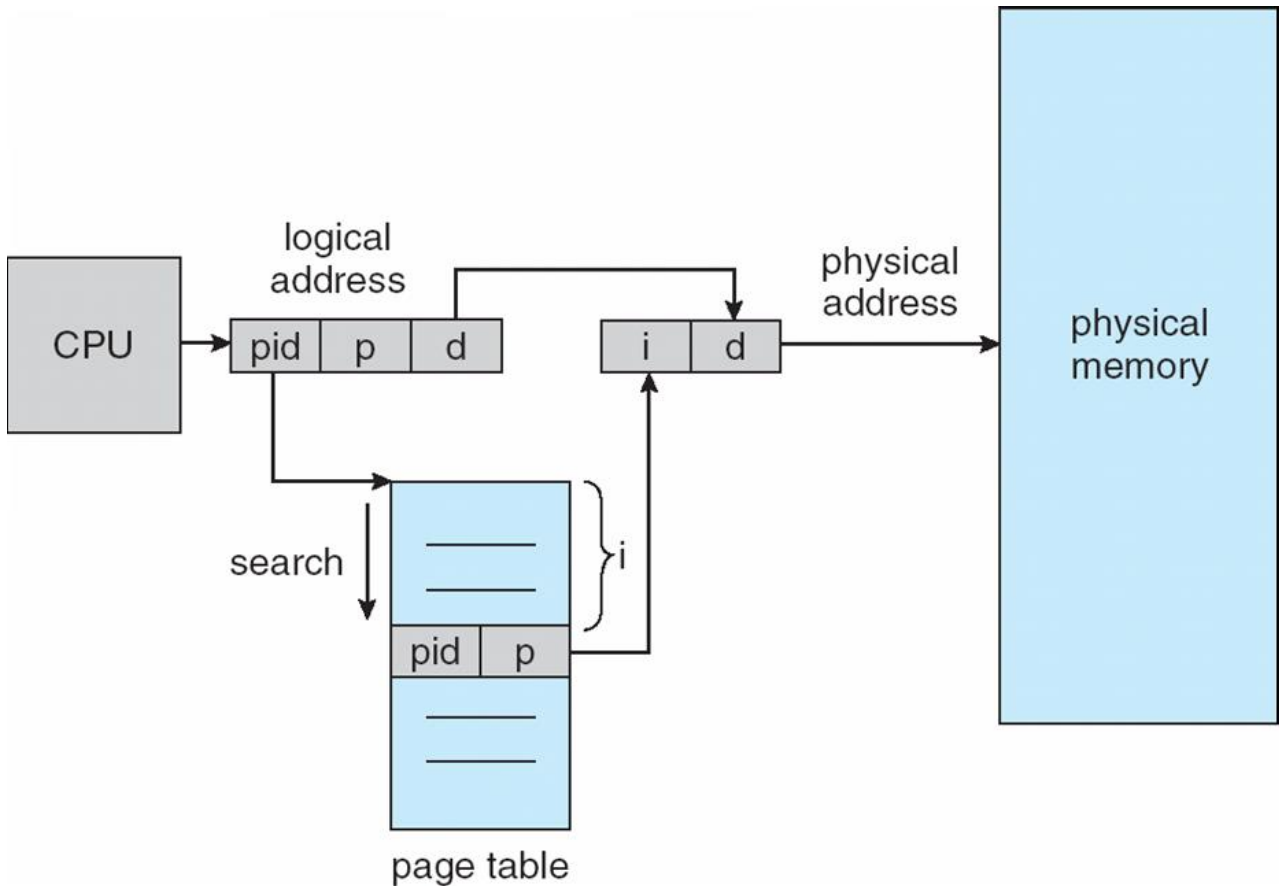
Hashed Page Table



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



End of Chapter 8