MEMORY MANAGEMENT

Main Memory

CS2006: OS - Spring 2023

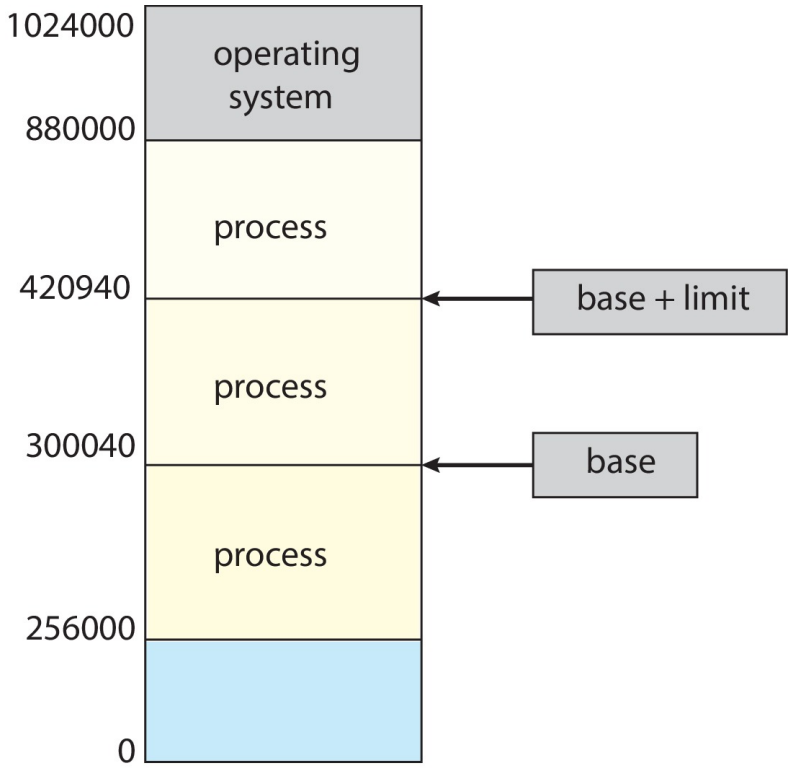
Course Supervisor: Anaum Hamid

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
- Register access is done 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

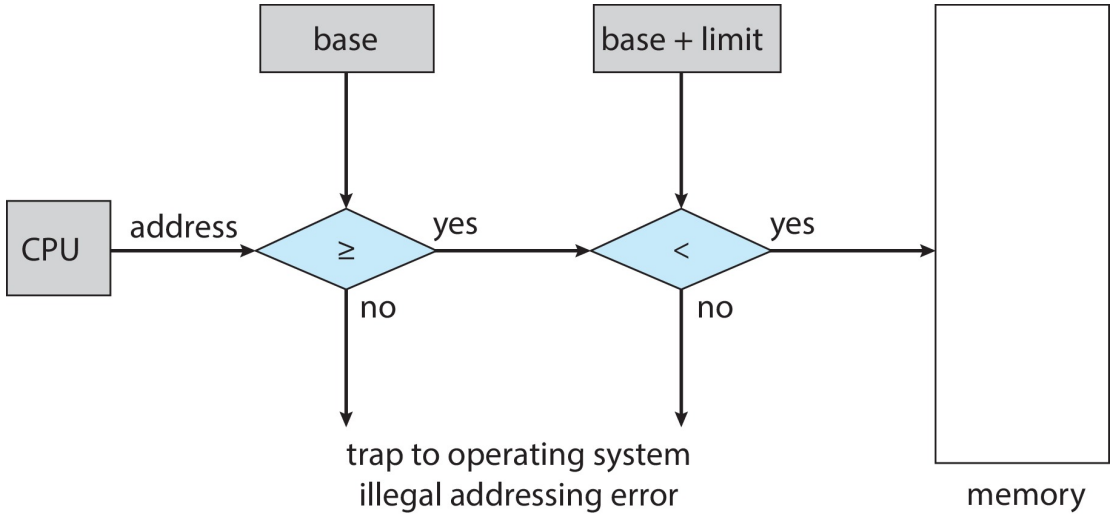
Base and Limit Registers

- 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.
- Limit register = 120900



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 instructions to loading the base and limit registers are privileged

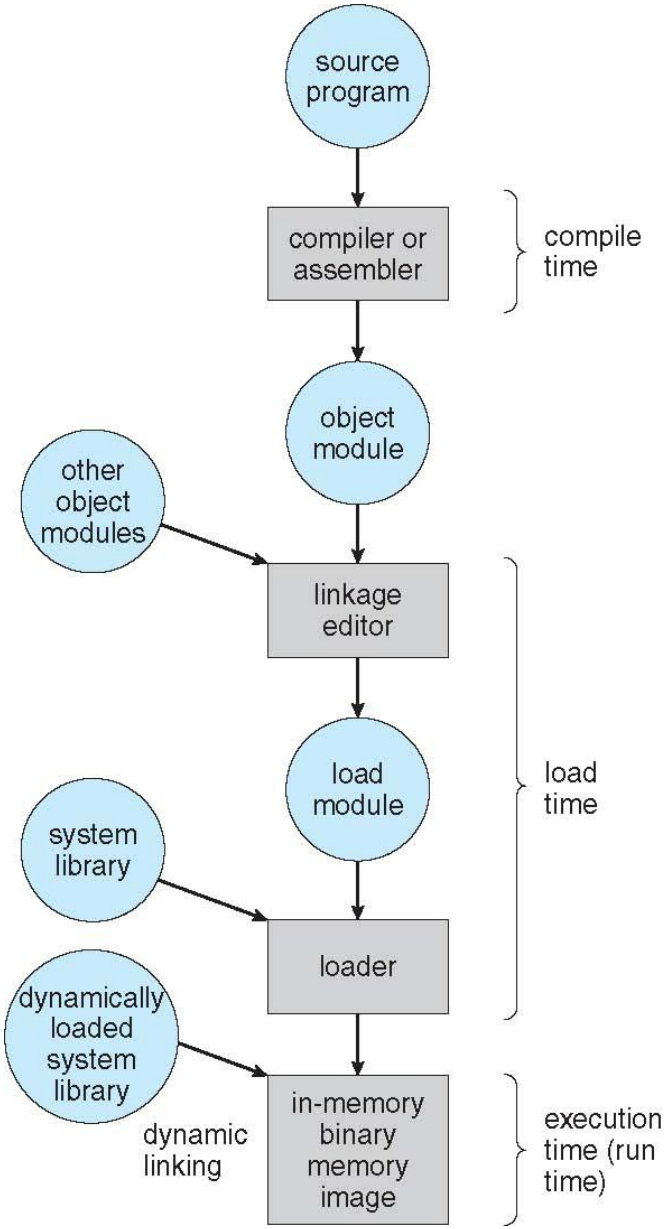
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- 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.

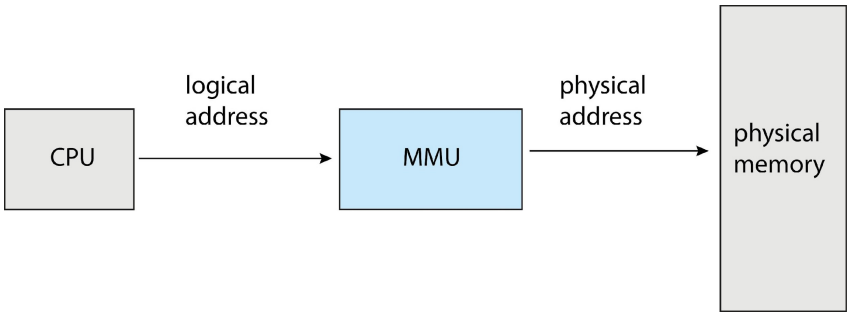
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme.

- **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)

- MMU is a hardware device that at run time maps virtual address to physical address

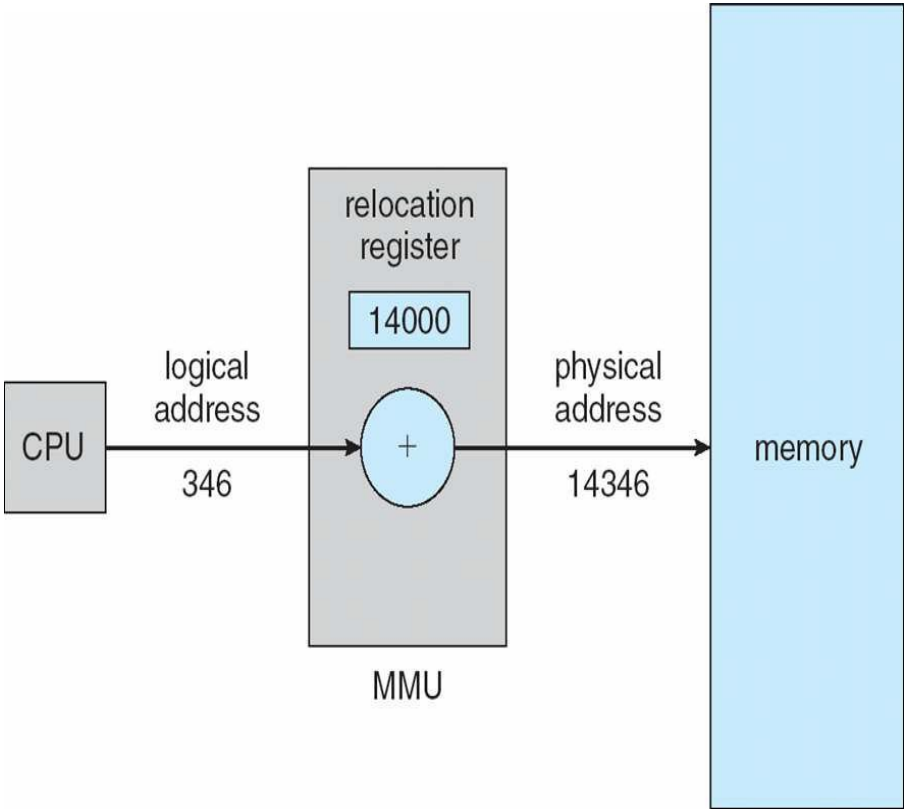


- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
  - Base register now called **relocation register**
  - MS-DOS on Intel 80x86 used 4 relocation registers

- 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

Dynamic relocation using a relocation register

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



STATIC & 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
- Dynamic linking is particularly useful for libraries

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.

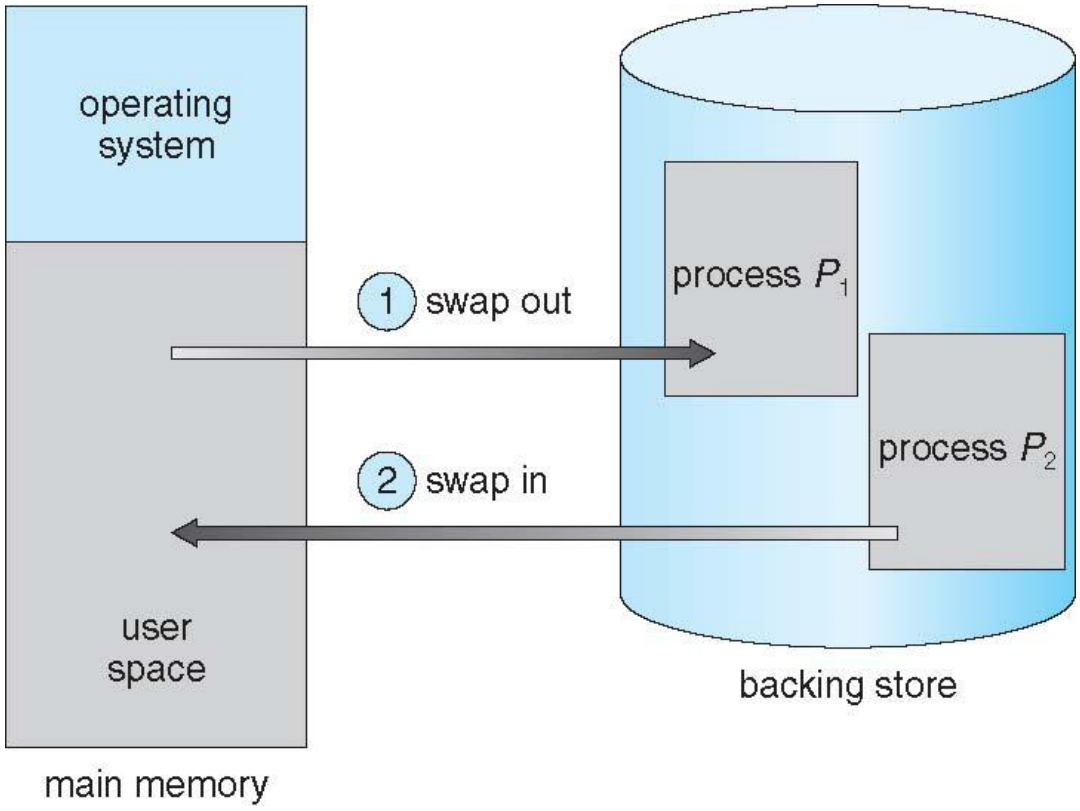
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

- 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

Problem:

100MB process swapping to hard disk with transfer rate of 50MB/sec from main memory.

- Swap Time (sec) = Size (mb)/ transfer rate (mb/sec)
  - Swap Time = 100/50 = 2sec = 2000ms
  - Swap[out] time of 2000 ms
  - Plus, swap in of same sized process
  - Total context switch swapping component time = 4000ms (4 seconds)

Context Switch Time including Swapping (Cont.)

Constraints of Swapping:

- Swap time can be reduced if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request memory()` and `release memory()`
- Other constraints are:
  - Pending I/O – can't swap out as I/O would occur to wrong process
  - 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
  - Swap only when free memory extremely low

Swapping on Mobile Systems

- Not typically supported
  - Flash memory based
    - Small amount of space
    - 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

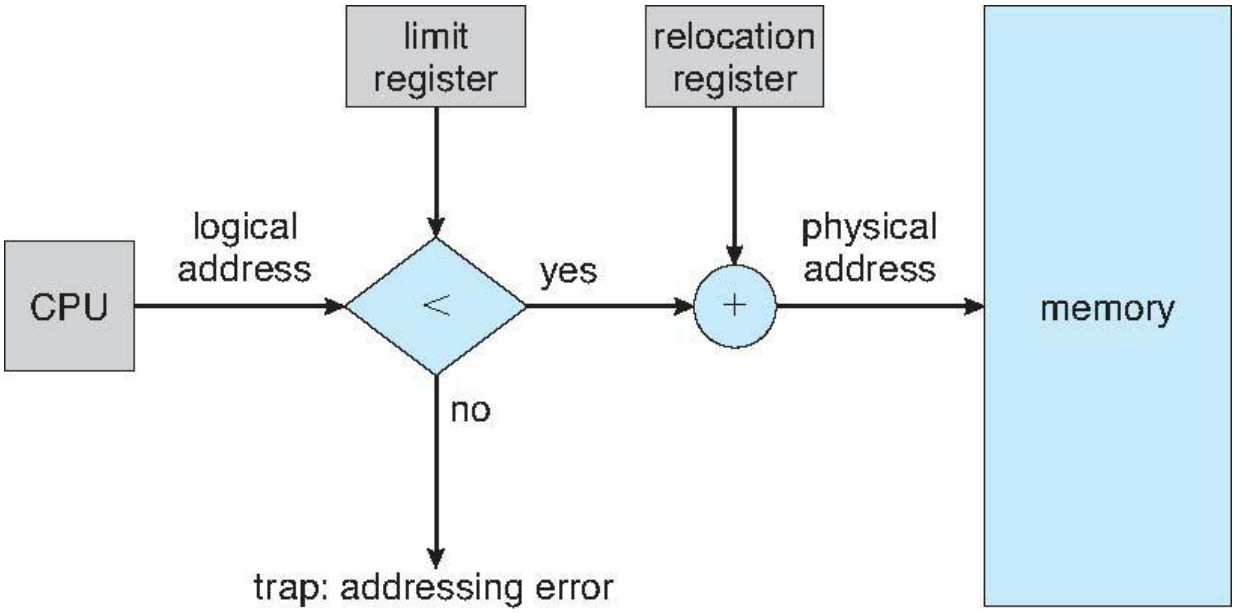
CONTIGUOUS ALLOCATION

- 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.)

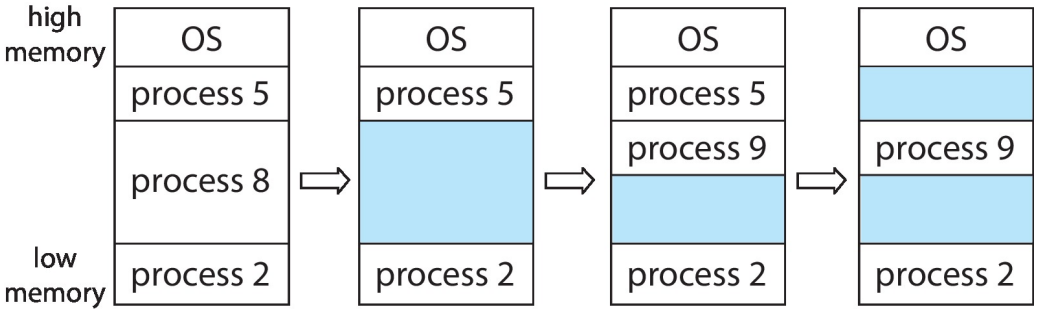
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  -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

Hardware Support for Relocation and Limit Registers



Variable Partition

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **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 (hole)



Dynamic Storage-Allocation Problem

- **First-fit**: First Fit fits data into memory by scanning from the beginning of available memory to the end, until the first free space which is at least big enough to accept the data is found. This space is then allocated to the data. Any left over becomes a smaller, separate free space.
- **Best-fit**: The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the smallest hole that is adequate. It then tries to find a hole which is close to actual process size needed.
- **Worst-fit**: The algorithm searches for free-space in memory in which it can store the desired information. The algorithm selects the largest possible free space that the information can be stored on (i.e. that is bigger than the information needing to be stored) and stores it there.

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

Dynamic Storage-Allocation Problem

Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K = 500K - 212K)
426K must wait

Best-fit:
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Dynamic Storage-Allocation Problem

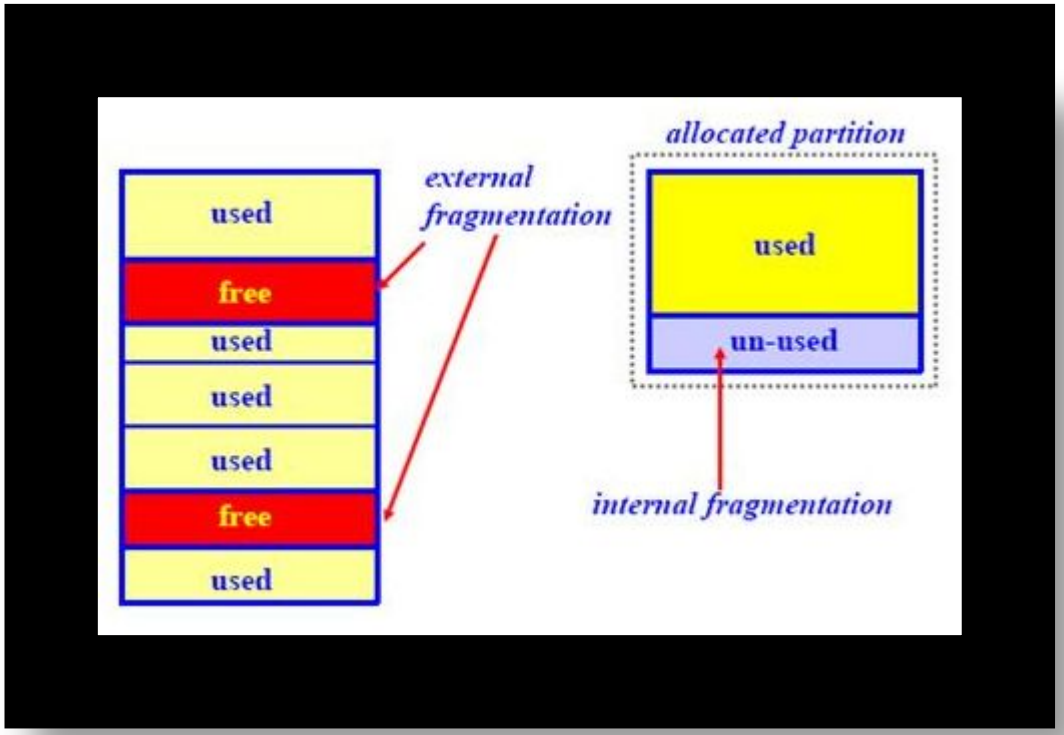
Worst-fit:
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait

In this example, best-fit turns out to be the best

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

Fragmentation



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

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

PAGING (Cont.)

- Calculating internal fragmentation

  Given:

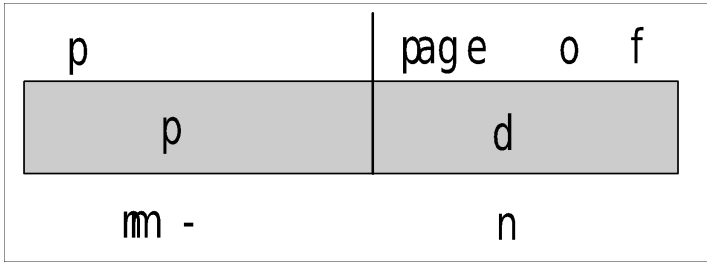
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes

  Calculate:

  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- Process view and physical memory now very different
- By implementation process can only access its own memory

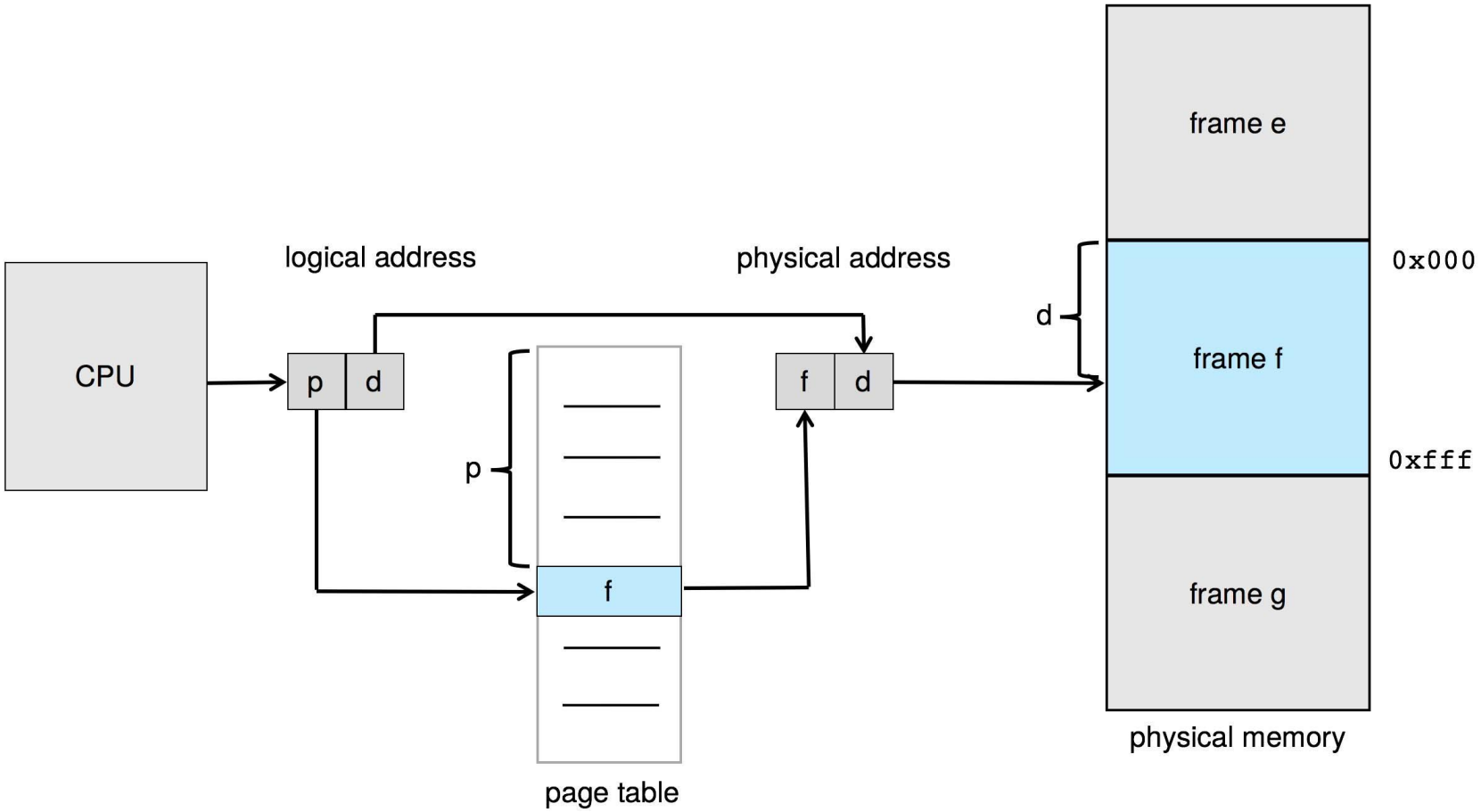
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number** (***p***) – used as an index into a **page table** which contains base address of each page in physical memory
  - **Page offset** (***d***) – combined with base address to define the physical memory address that is sent to the memory unit

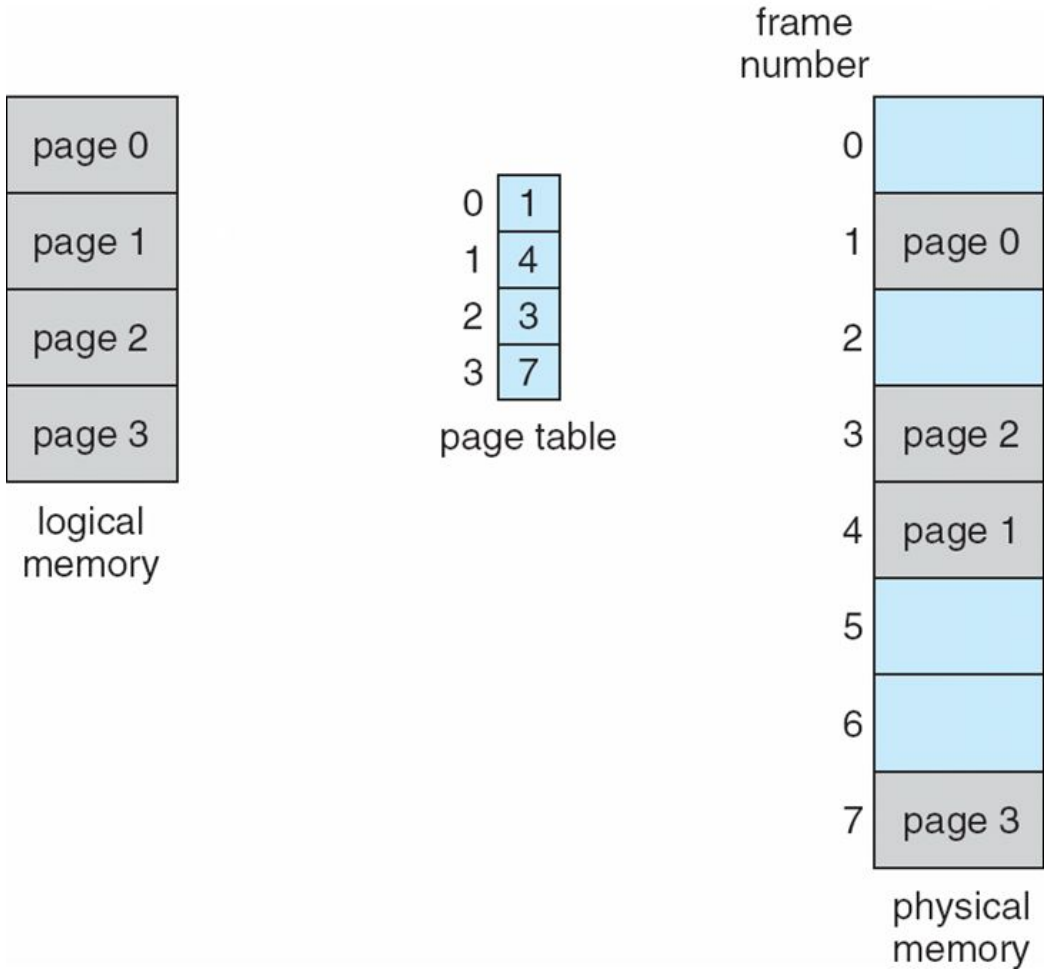


  - For given logical address space 2^m and page size 2^n

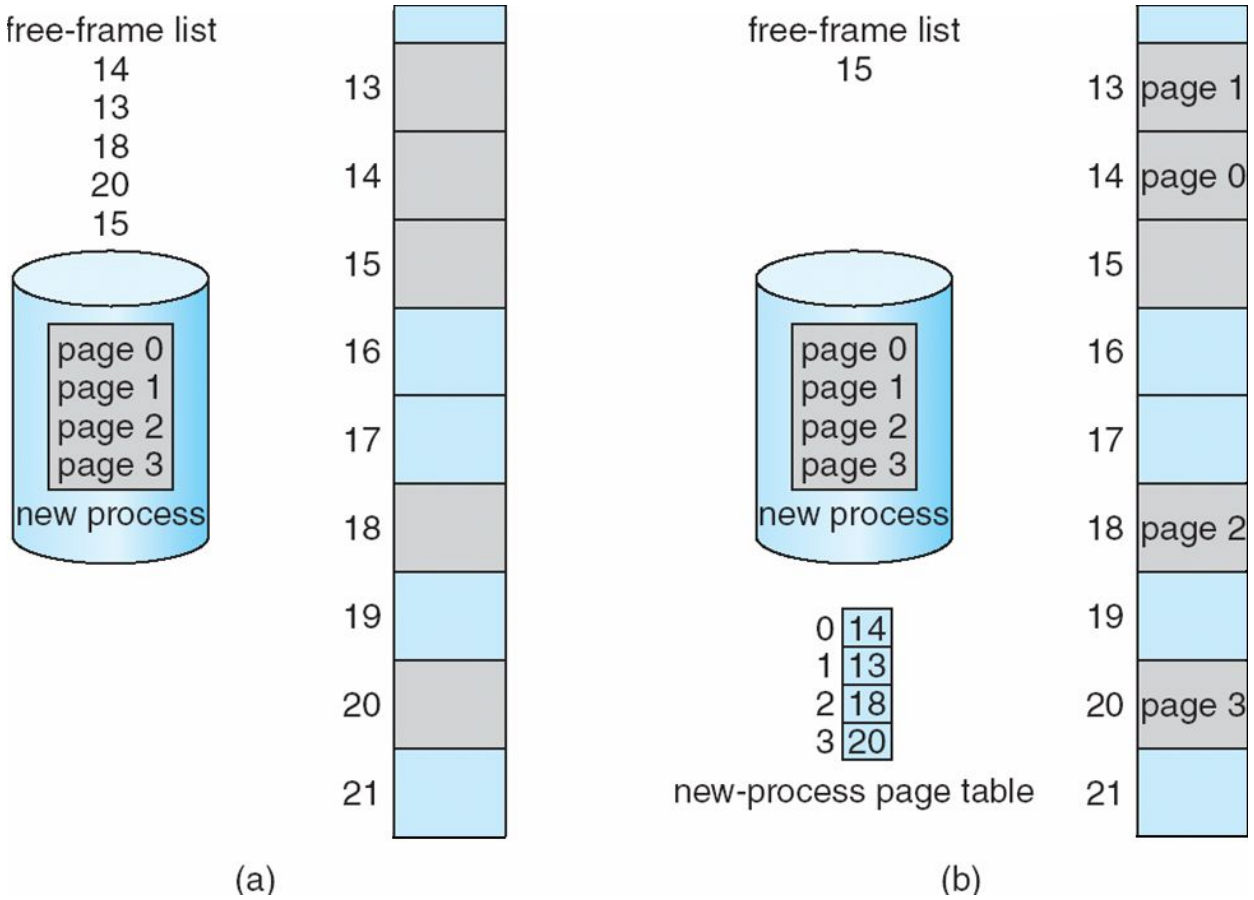
PAGING HARDWARE



Paging Model of Logical and Physical Memory



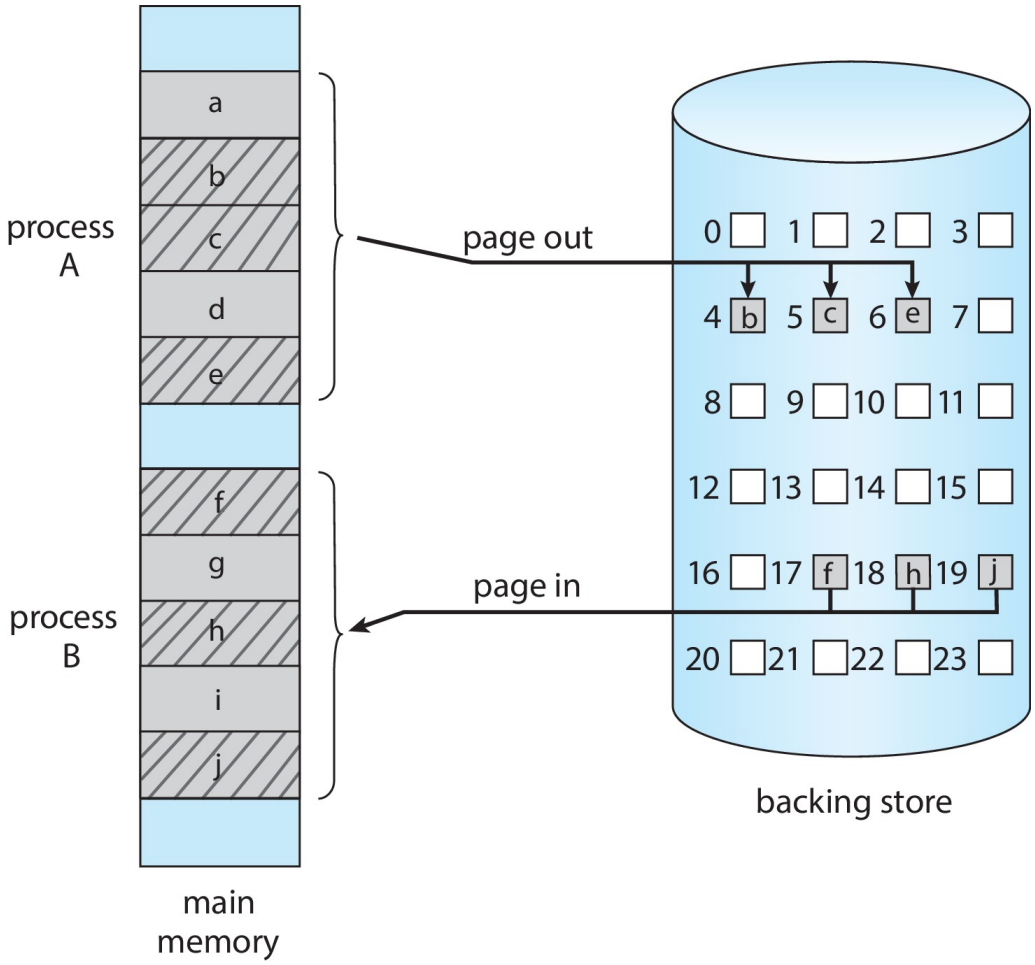
FREE FRAMES



Before allocation

After allocation

SWAPPING WITH PAGING



Paging Numerical-1

- Assuming a 1-KB page size, what are the page numbers and offsets for the following address references (provided as decimal numbers):

a. 3085

b. 42095

c. 215201

d. 650000

e. 2000001

Page Numerical-1 (Solution2)

3085/1024= 3.0126953125
3 = pages
0.126953125 * 1024 = 13

Page size = 1 KB = 1024B

Page number	Offset
3085/1024 = 3	3085 mod 1024 = 13
42095/1024 = 41	42095 mod 1024 = 111
215201/1024 = 210	215201 mod 1024 =161
650000/1024 =634	650000 mod 1024 =784
2000001/1024 = 1953	2000001 mod 1024 = 129

Paging Numerical-2

- Consider a logical address space (LAS) of 32 pages with 1024 words per page; mapped onto a physical memory of 16 frames.

1. How many bits are required in the logical address?
2. How many bits are required in the physical address?

Answer:

1. 2^5 * 2^10 = 2^15 = 15 bits.
2. 2^4 * 2^10 = 2^14 = 14 bits.

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 using a special fast-lookup hardware cache called **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

Translation Lookaside Buffer

- 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

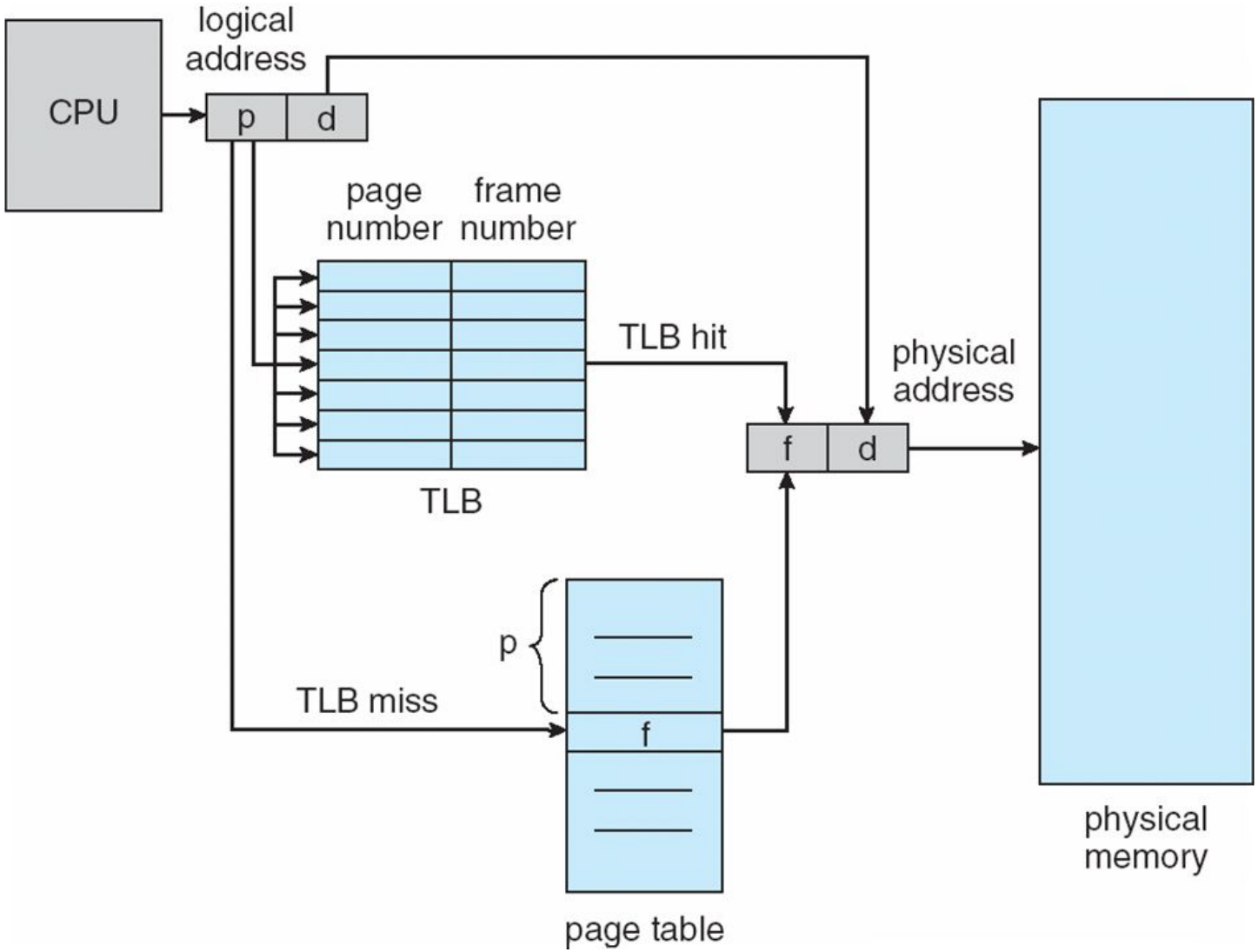
ASSOCIATIVE MEMORY

- 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



Hit ratio

- **Hit ratio = α**
  - Hit ratio – percentage of times that a page number is found in the TLB

- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.

- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise, we need two memory access, so it is 20 ns

Effective Access Time

- **Effective Access Time** (**EAT**)

EAT= α x memory access time (hit) + fail x memory access time (failed)

Effective Access Time (Cont.)

- Consider α = 80%, 100ns for memory access
  - EAT = 0.80 x 100 + 0.20 x (100+100) = 120ns

- Consider more realistic hit ratio -> α = 99% , 100ns for memory access
  - EAT = 0.99 x 100 + 0.01 x (100+100) = 101ns

EAT Numerical

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

1. If a memory reference takes 200 nanoseconds, how long does a paged memory reference take?
2. If we add associative registers, and 75 percent of all page-table references are found in the associative registers, what is the effective memory reference time? (Assume that finding a page-table entry in the associative registers takes zero time if the entry is there.)

EAT Numerical (Sol)

Answer:

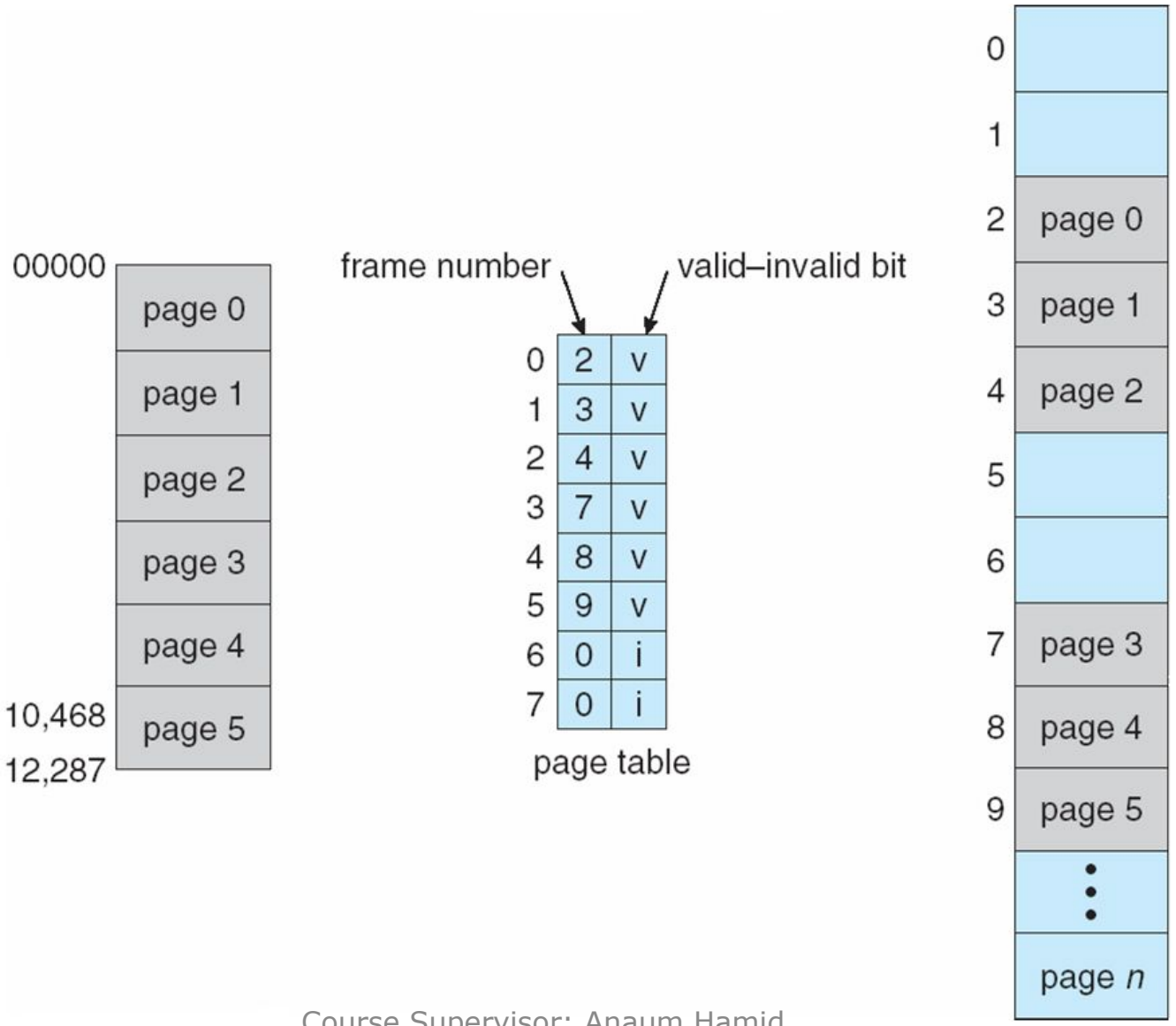
1. 400 nanoseconds: 200 nanoseconds to access the page table and 200 nanoseconds to access the word in memory.
2. Effective access time = 0.75 × (200 nanoseconds) + 0.25 × (400 nanoseconds) = 250 nanoseconds.

Memory Protection

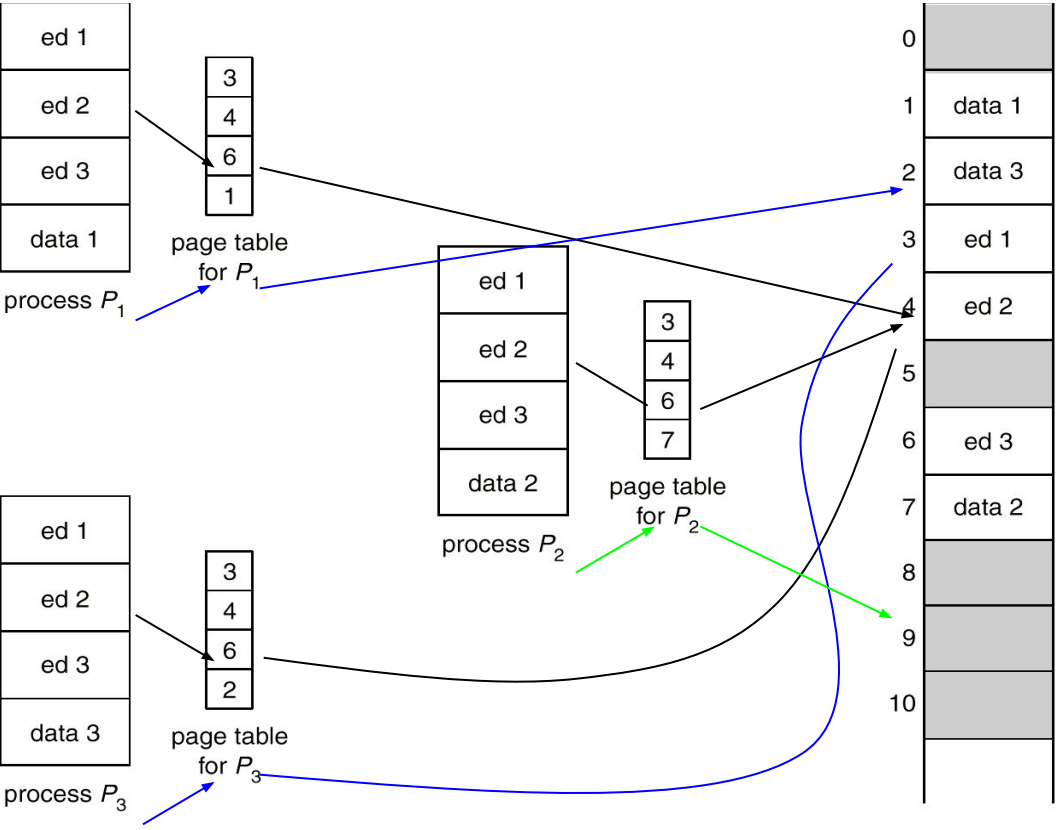
- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on

- **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**)

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



SHARED PAGES



- Shared code
  - Read-only (reentrant) code shared among processes
  - Shared code appeared in same location in the physical address space
- Private code and data
  - Each process keeps a separate copy of the code and data, (e.g., stack).
  - Private page can appear anywhere in the physical address space.
- Copy on write
  - Pages may be initially shared upon a fork
  - They will be duplicated upon a write

Structure of the Page Table

- Memory structures for paging can get huge using straightforward 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

Structure of the Page Table

1. Hierarchical Paging

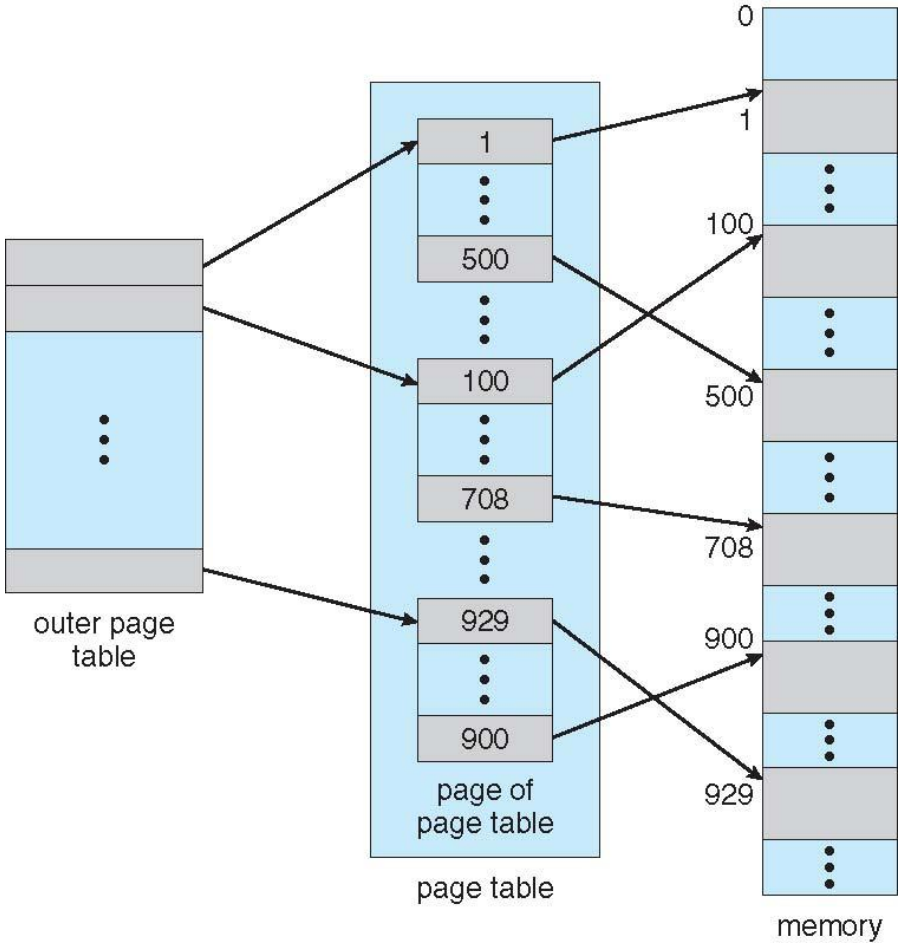
1. Hashed Page Tables

1. Inverted Page Tables

Hierarchical Page Tables

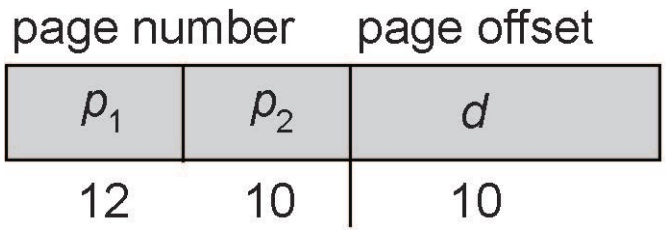
- Break up the logical address space into multiple page tables
- A simple technique is a two-level page table
- We then page the page table

Two-Level Page-Table Scheme



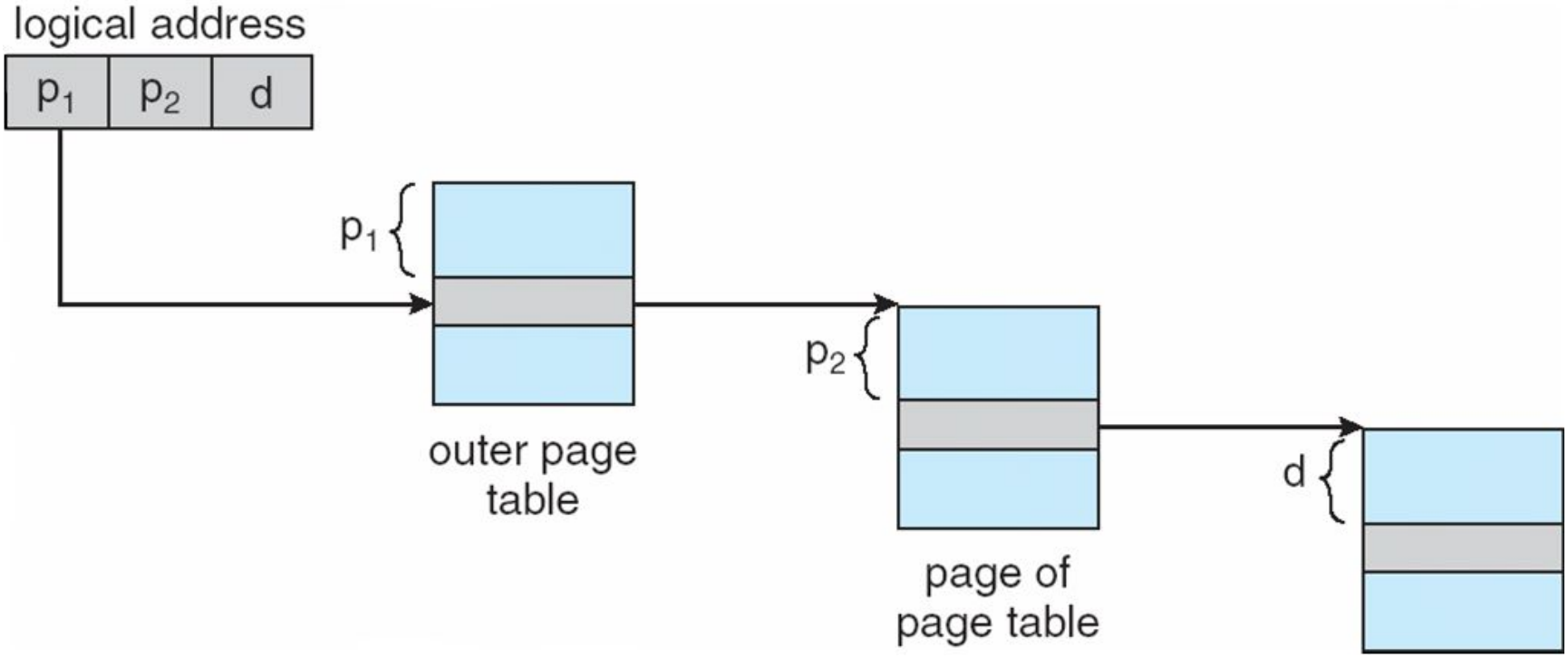
Two-Level Paging Example

- 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

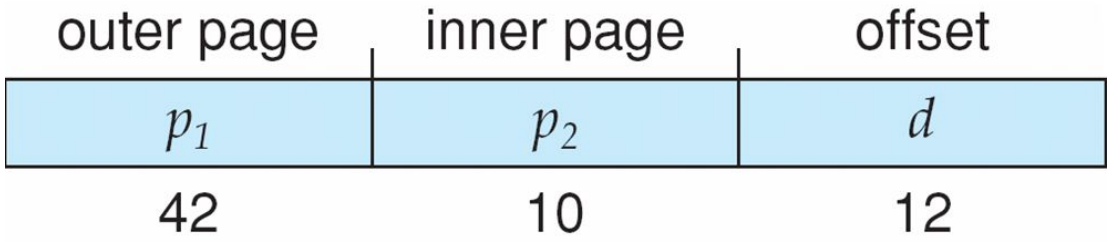


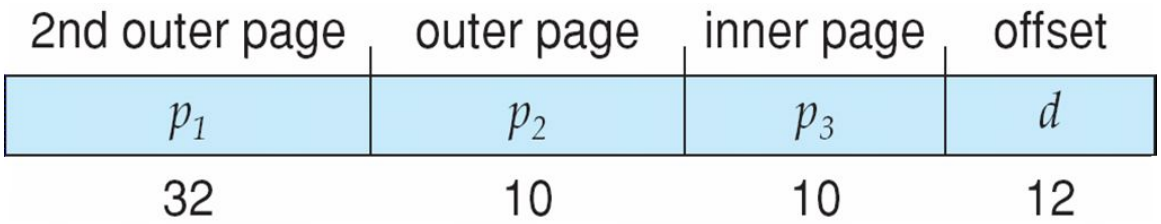
64-bit Logical Address Space

- Even two-level paging scheme not sufficient.

- If page size is 4 KB (2^{12})
  1. Then page table has 2^{52} entries
  2. If two level scheme, inner page tables could be 2^{10} 4-byte entries
  3. One solution is to add a 2^{nd} outer page table
  4. 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

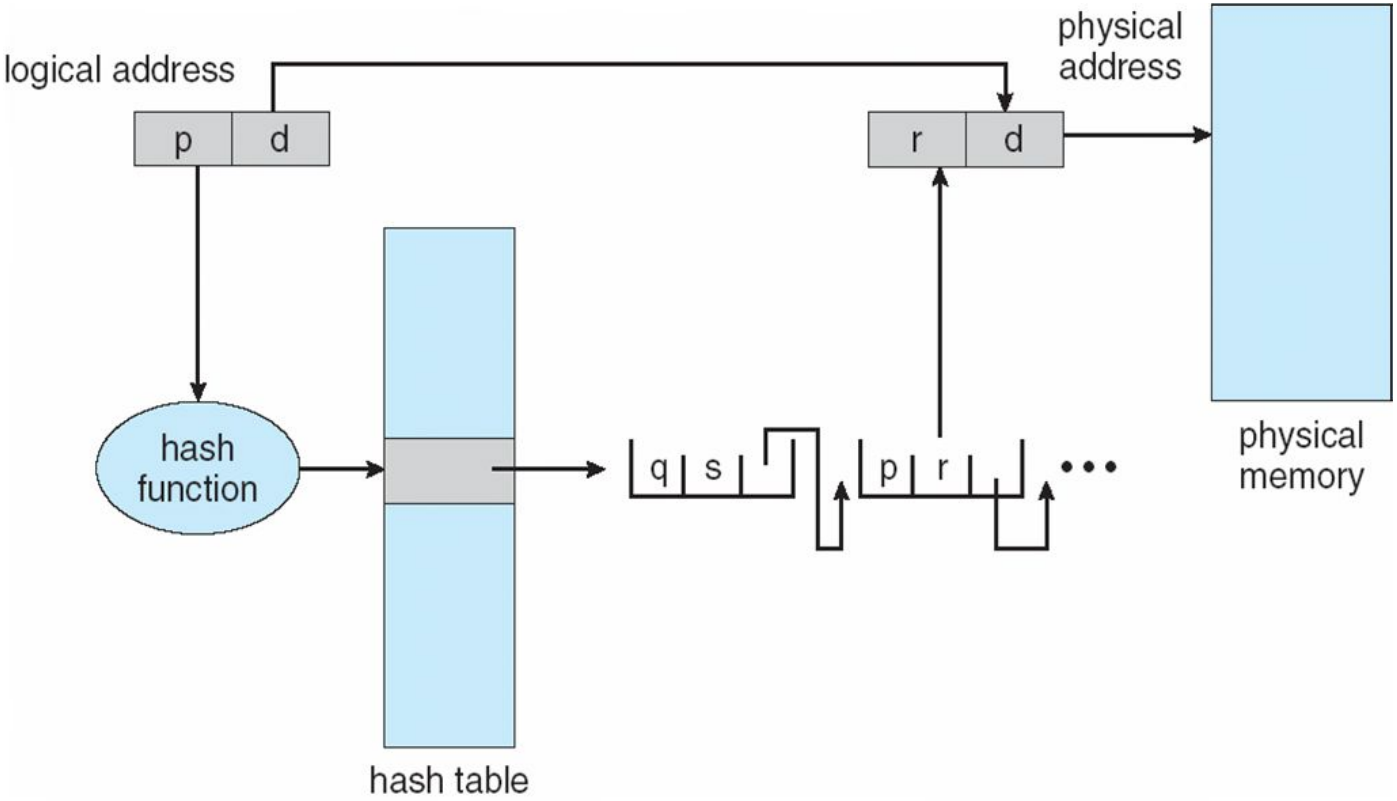




Hashed Page Tables

- 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)

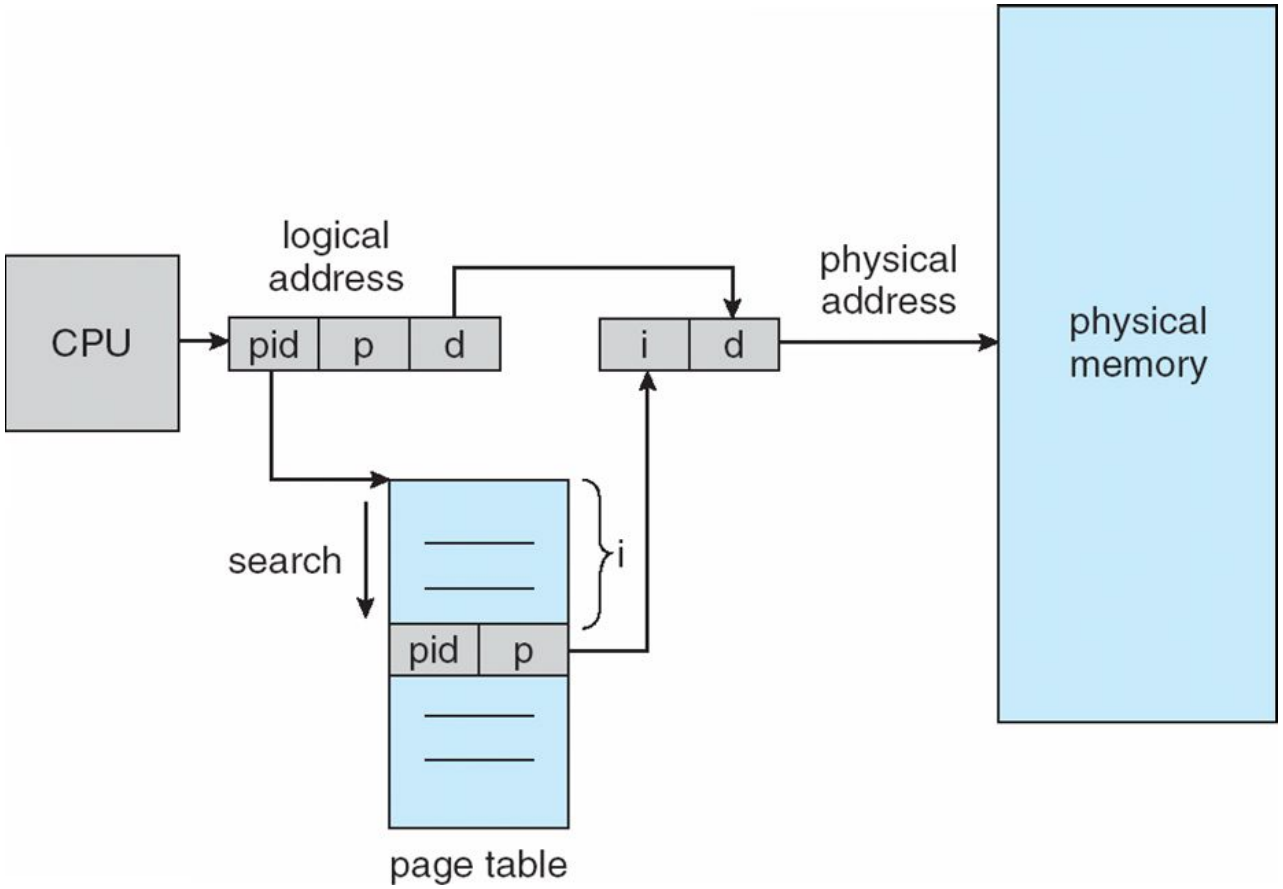
Hashed Page Table



Inverted Page Table

- Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- 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

Inverted Page Table Architecture



THANK YOU!