What you can get out of this course

- Knowledge that your OS provides you with features that you can use in your programs, both in general and specific features.
  - E.g., you probably can't reel off some memory-mapped I/O code, but you (should) know that it exists. You can look into it in more detail if you have a project that does file I/O and might benefit from it.
- Knowledge of how an OS is implemented
  - Know that different approaches will have different strengths & weaknesses; again, look up the details when you need them.
- Get 'cultural' knowledge that all Comp.Sci. people should know
  - For some older-timers, not knowing what a linker is will be embarrassing
- Hands-on experience getting up to speed on a 'large' software project and then successfully modifying it.

Memory

Main Memory: High Level Agenda

- Background
- Swapping
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Segmentation
- The x86 architecture

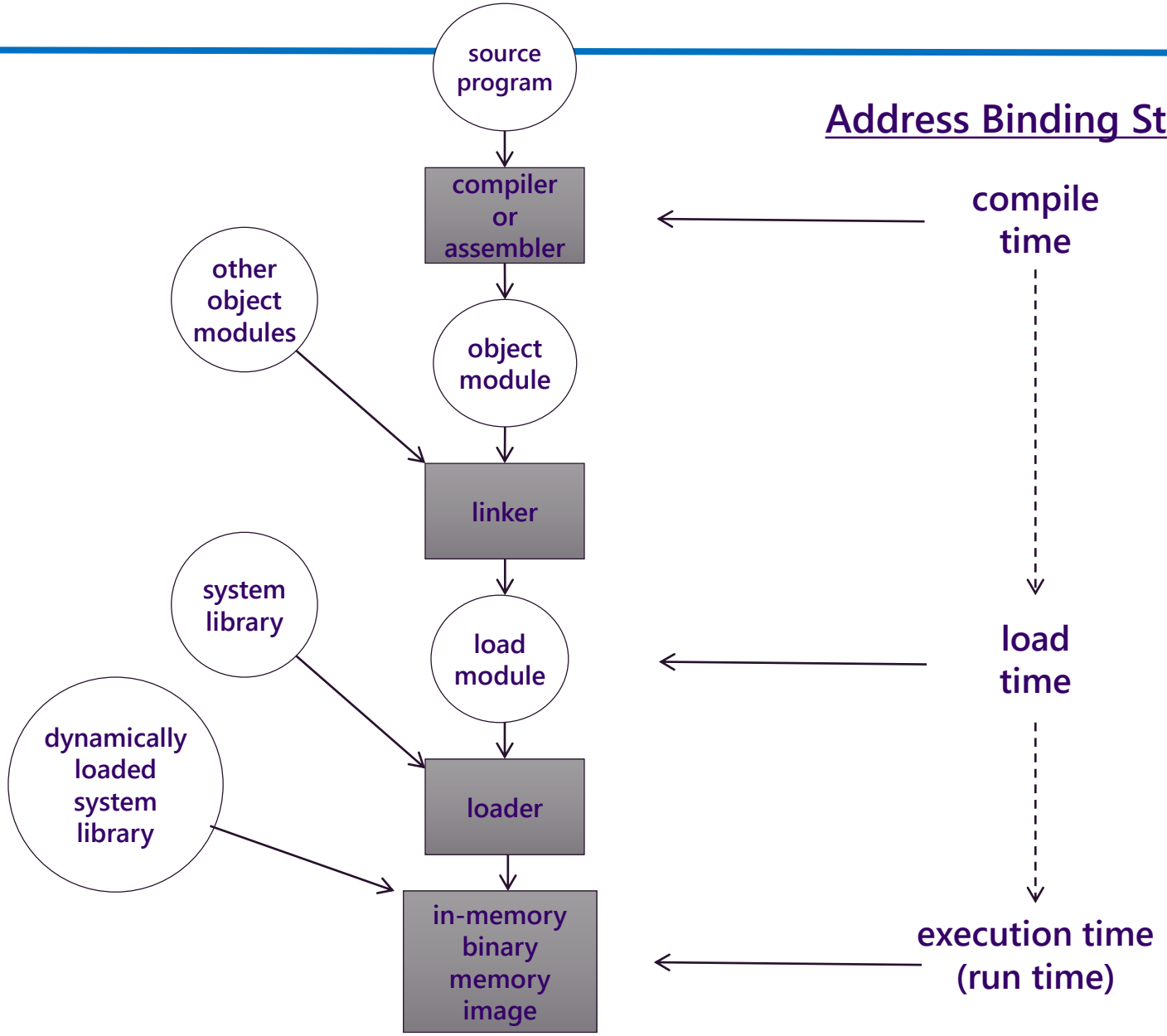
Key Background Points

- Programs must be read into memory (from disk) and placed within a process to be run
- Main memory and registers are the only storage a CPU can access directly
- Main memory access can take many cycles
- Register access is fast (one CPU clock)
- Cache memory sits between main memory and CPU
- Memory Protection is required to ensure correct operation

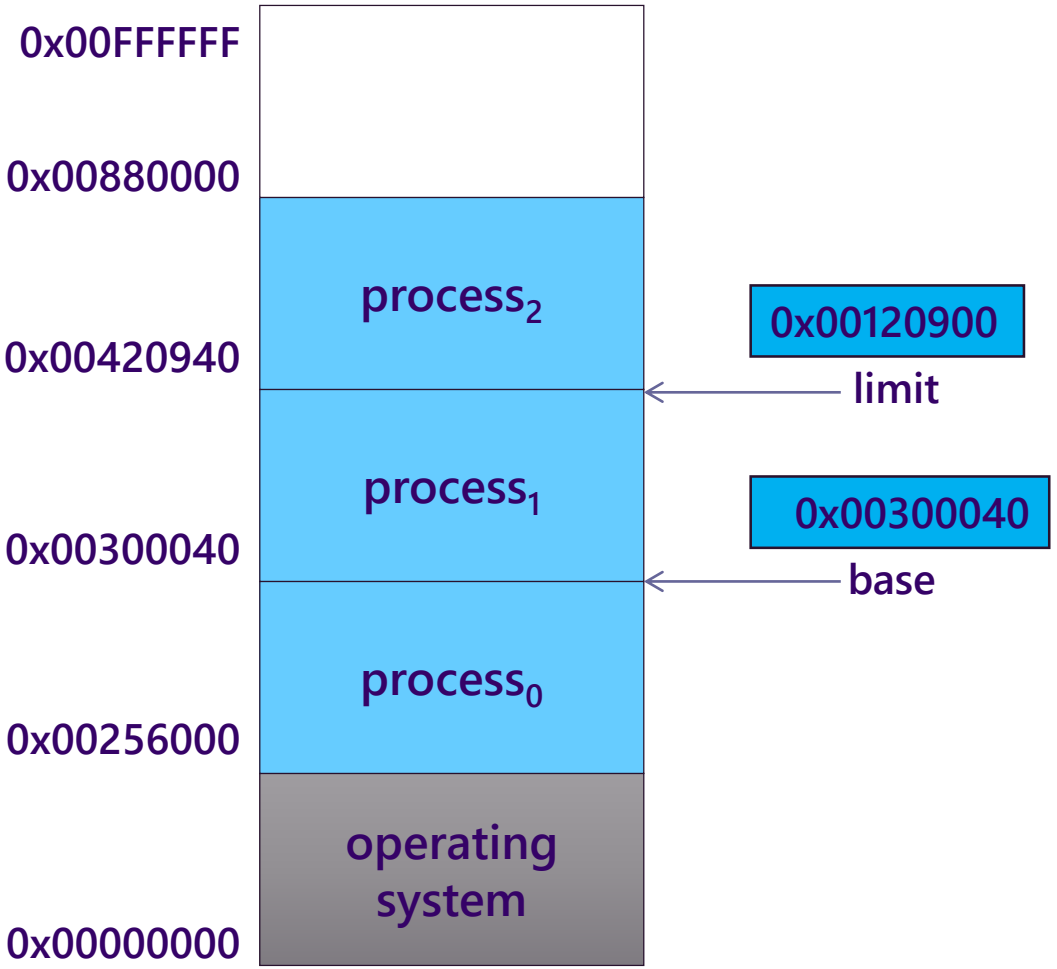
- Good reference for background material!

  http://www.cs.rutgers.edu/~pxk/416/notes/09-memory.html

Multistep Processing of a User Program



Base and Limit Registers: Logical Address Space

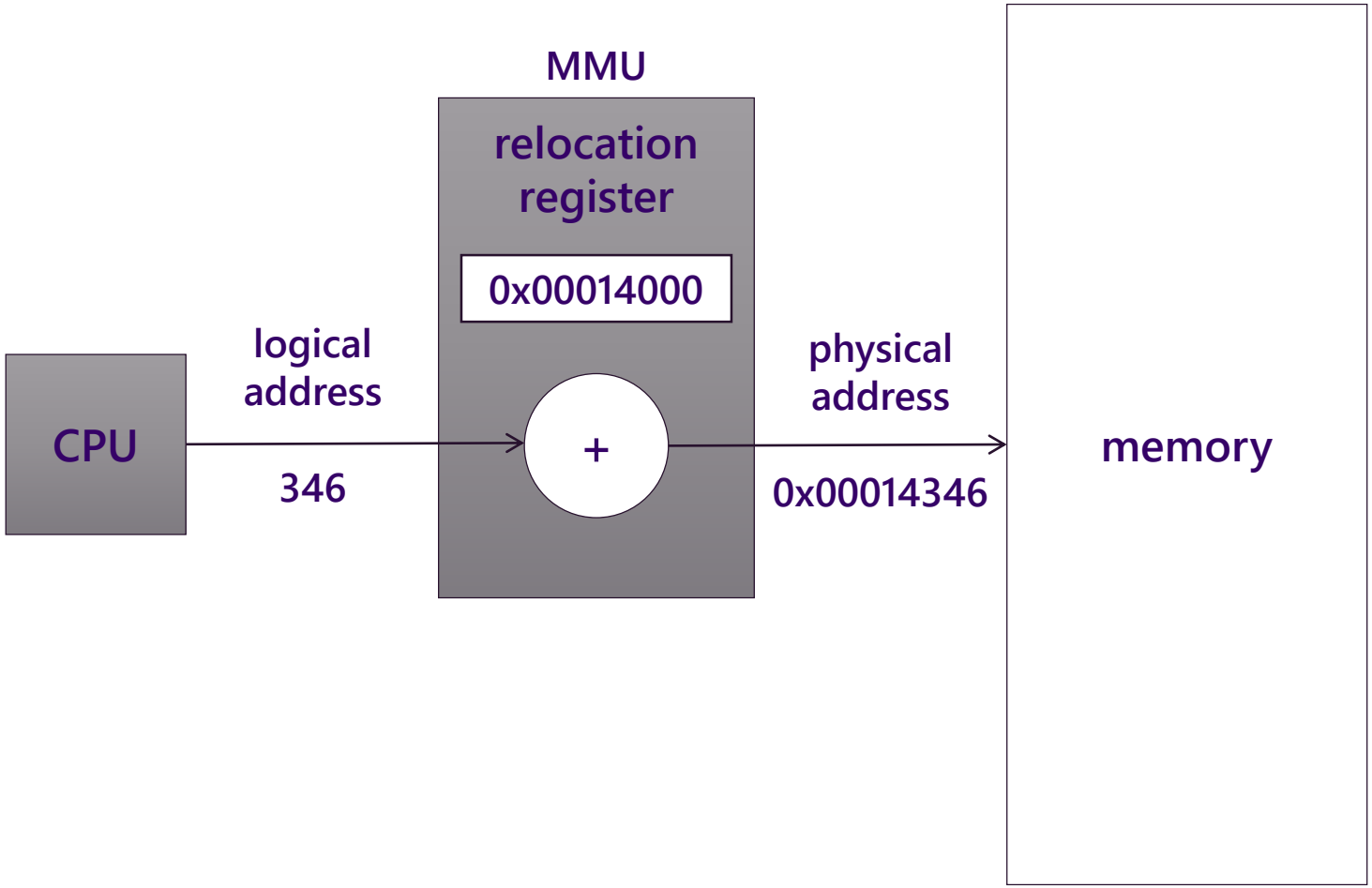


Logical vs. Physical Addresses

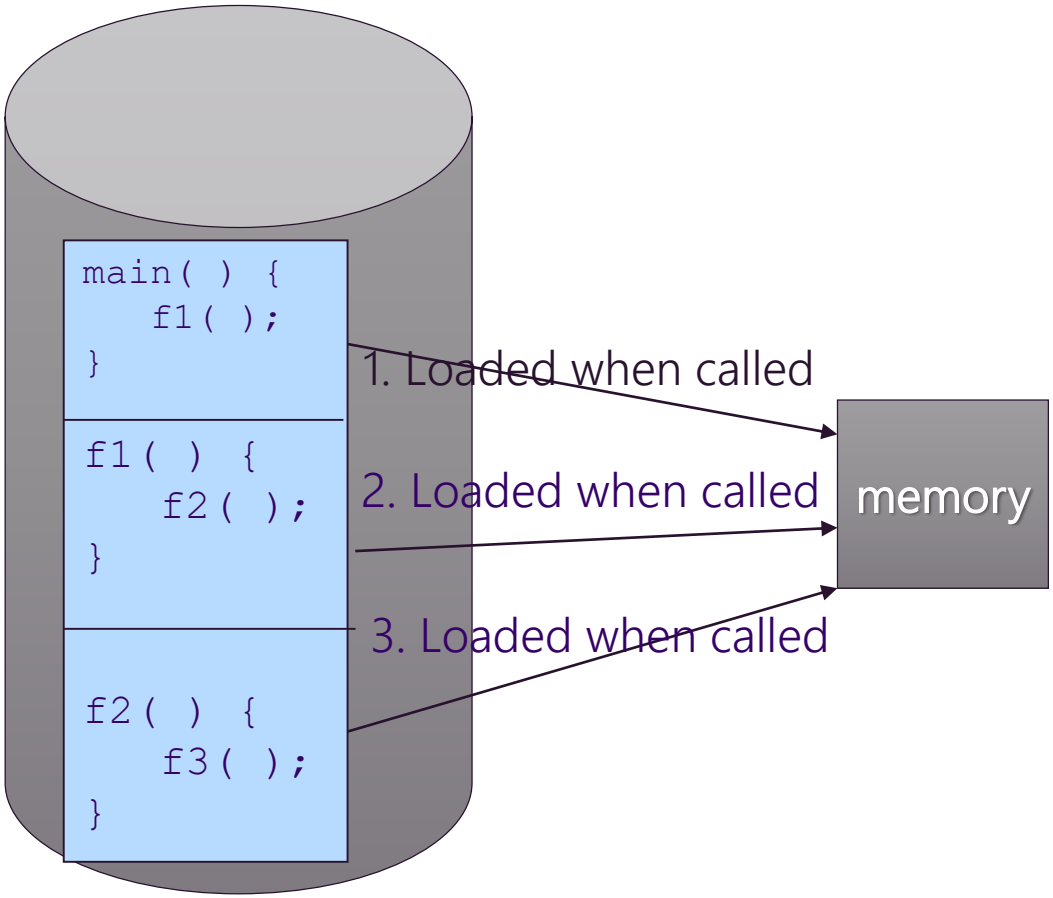
- Logical address – generated by the CPU
  - also called Virtual Address
  - Ranges from 0-max
  - CPU executes with logical addresses
- Physical address – actual hardware memory address
  - Ranges from R+0 to R+max
  - Mapped from the logical address by the relocation register
- Logical address and physical addresses differ at
  - execution-time address-binding stage

Compile-time	Load-time	Run-time
Logical=Physical	Logical=Physical	Logical Physical

Dynamic Relocation

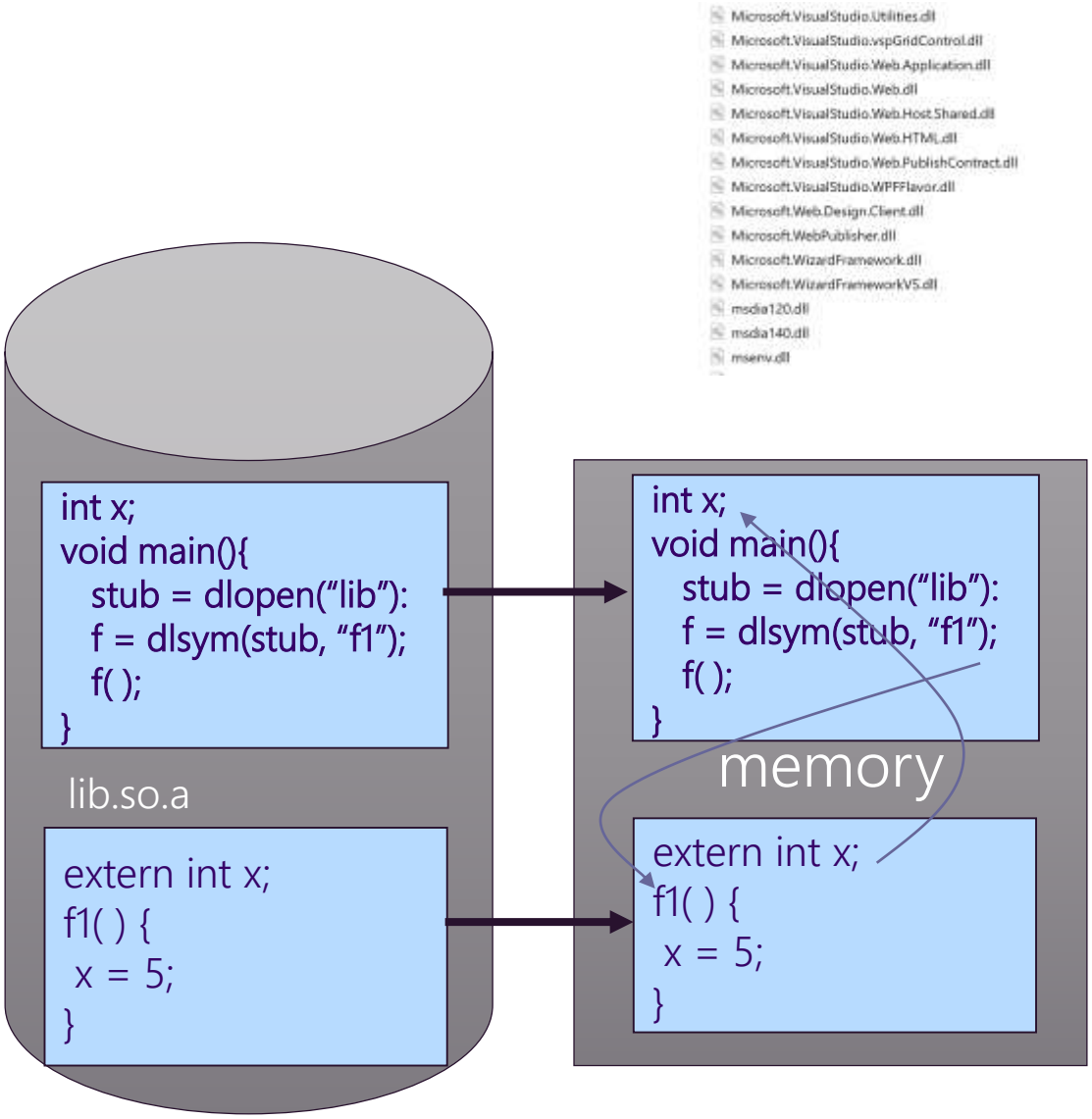


Dynamic Loading



- Unused routines are never loaded
- Useful when the code size is large
- Responsibility of the programmer to take advantage of dynamic loading
- OS may or may not provide support

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.
- OS may allow multiple processes to access the same shared memory addresses
- **Useful** for language libraries
- Also known as **shared libraries**

DISCUSSION

1. What is bigger, logical or physical memory?

2. Name three differences between logical and physical memory.

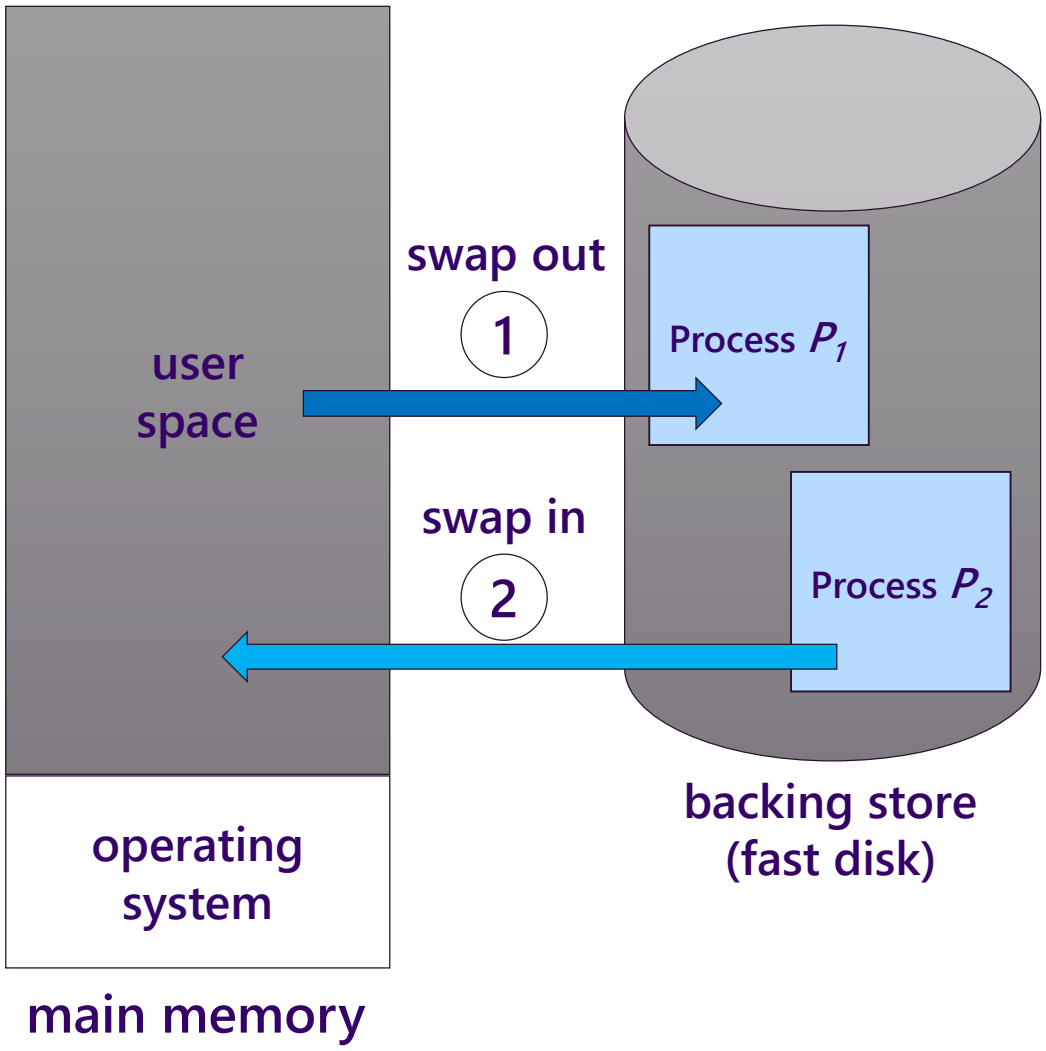
3. If a logical address is 0x567B0 and the relocation register has the program loaded with relocations register of 0x878000, what would the physical address be? What if the program were loaded at 0x980000?

Swapping

Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Not always an option. E.g., don't swap processes that are waiting for I/O
- 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
- Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- 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
- Today's main-line operating systems (Unix, Linux, Windows) only swap when overloaded

Swapping Diagram



1. When a process P_1 is blocked so long (for I/O), it is swapped out to the backing store, (swap area in Unix.)

2. When a process P_2 is (served by I/O and) back to a ready queue, it is swapped in the memory.

- Use the Unix `top` command to see which processes are swapped out.

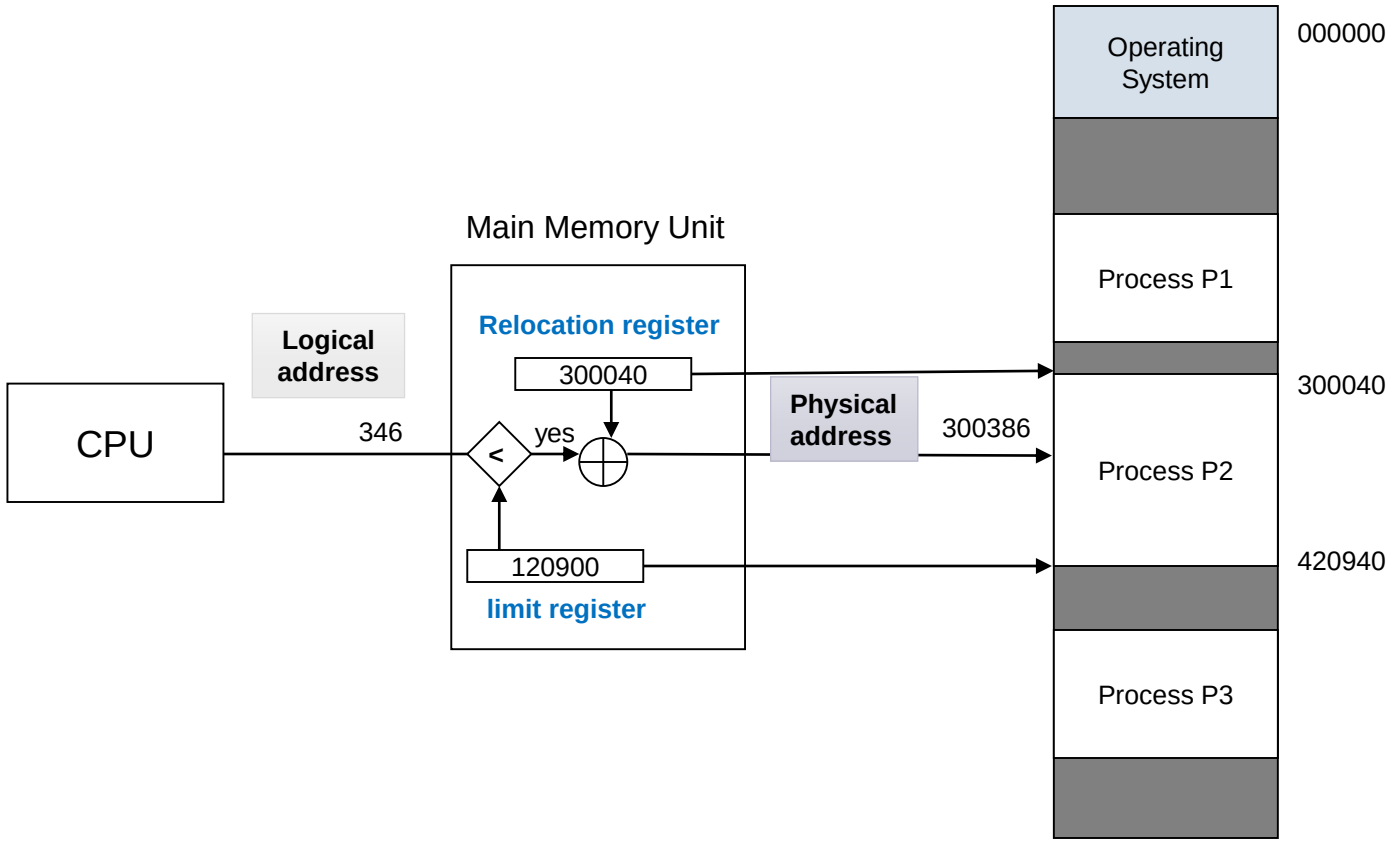
Contiguous Allocation

Contiguous Allocation

- Main memory usually split into two partitions:
  - Resident operating system, usually held in low memory
    - → with interrupt vector
  - User processes held in high memory

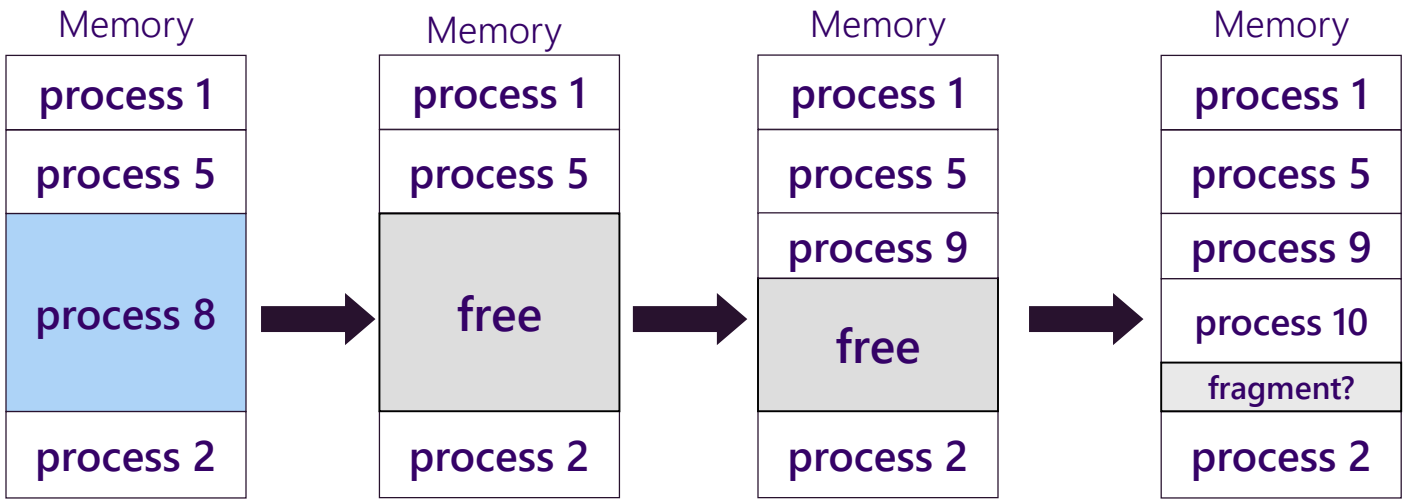
- For each process
  - Logical space is mapped to a contiguous portion of physical space by the MMU dynamically
  - A relocation and a limit registers are prepared
    - Relocation registers are used to protect user processes from each other, and from changing OS code and data
    - Relocation register contains value of smallest physical address
    - Limit register maximum possible logical address – each logical address must be less than the limit register
    - These registers are stored in the PCB and loaded as part of a context switch

Contiguous Allocation



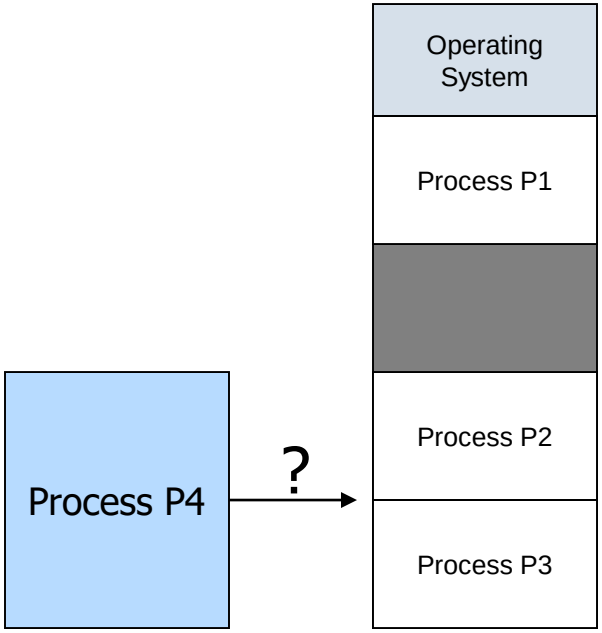
Contiguous Allocation (cont.)

- Multiple-partition allocation
  - 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
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



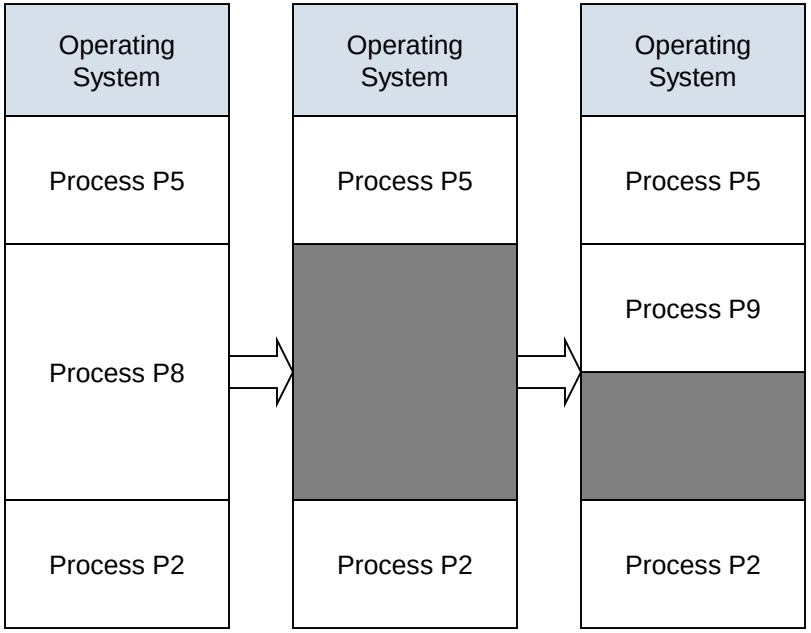
Fixed vs. Variable-Sized Partition

Fixed-Sized Partition



- Each partition is allocated to a process
- How about a bigger process?

Variable-Sized Partition



- Any size of processes, (up to the physical memory size) can be allocated.
- First, best, or worst fit?
- Holes (free spaces) still remain.

External Fragmentation

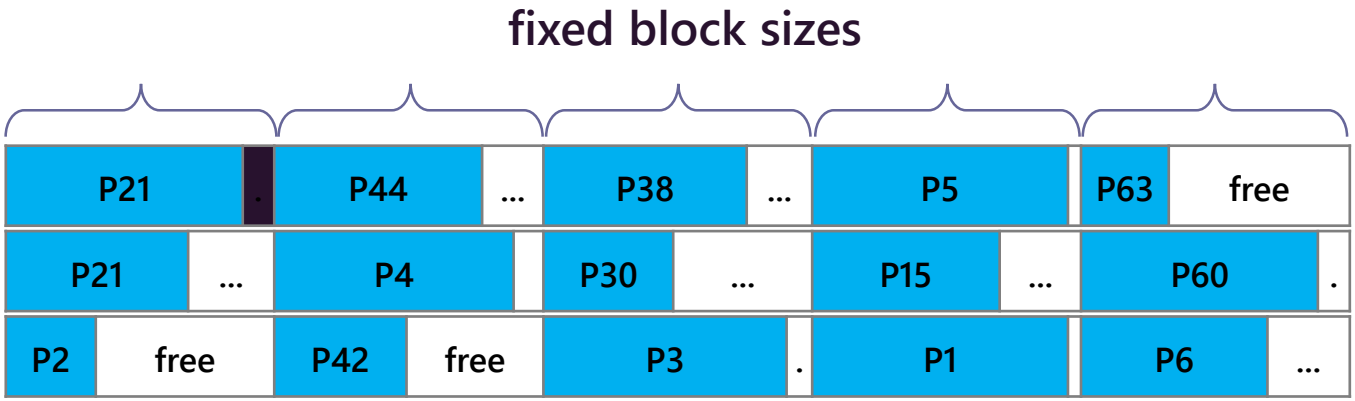
P21	P41	P44	P37	free	P38	P18	P15	free	
free		P47	P55			P58	free		
P57	P20	P51		P13	P77	P16	P6		
P7						P49	P38	P3	
P2	P53	P37	P56	P40	free		P37	P31	P4
free				P5	P63	P49	P14	P59	P21
P25		P58							
P24	P47	P9	P54	P62	P16	P61	P41	P11	P6
P61	free								P52
free	P43	P66	P48	P61	P67	free		P23	P68
P21	free	P8	free					P48	free
P55			free			P46	P7	P32	

Compaction

P21	P41	P44	P37	P38	P18	P15	P5	P63	P49
P47	P55			P58	P57	P20	P51		P13
P77	P16	P6			P49	P38	P3	P25	
P7						P5			P4
P2	P53	P37	P56	P40	P37	P31	P14	P59	P21
P58								P23	P68
P24	P47	P9	P54	P62	P16	P61	P41	P11	P6
P61	P43	P66	P48	P61	P67	P46	P7	P32	P52
P21	P8	P48							

free

Internal 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

Dynamic Storage Allocation

- First Fit
  - Allocate the first hole that is big for the new process
  - → No searching required
- Best Fit
  - Allocate the smallest hole from all available holes that is big enough to contain the new process
  - → Produces the smallest leftover hole. (requires search or sort)
- Worst Fit
  - Allocate the largest hole, from all available holes.
  - → Produces the largest leftover hole. (requires search or sort)

Contiguous Physical Allocation

- Summarizing…
  - Contiguous allocation of physical memory suffers from significant space utilization issues
  - Swapping out a full process is super-expensive
  - Logical space of a process can be quite large and may not fit in physical memory (more about this in Chapt. 9)
  - While used in early OS's today contiguous physical allocation is not used in most Operating Systems

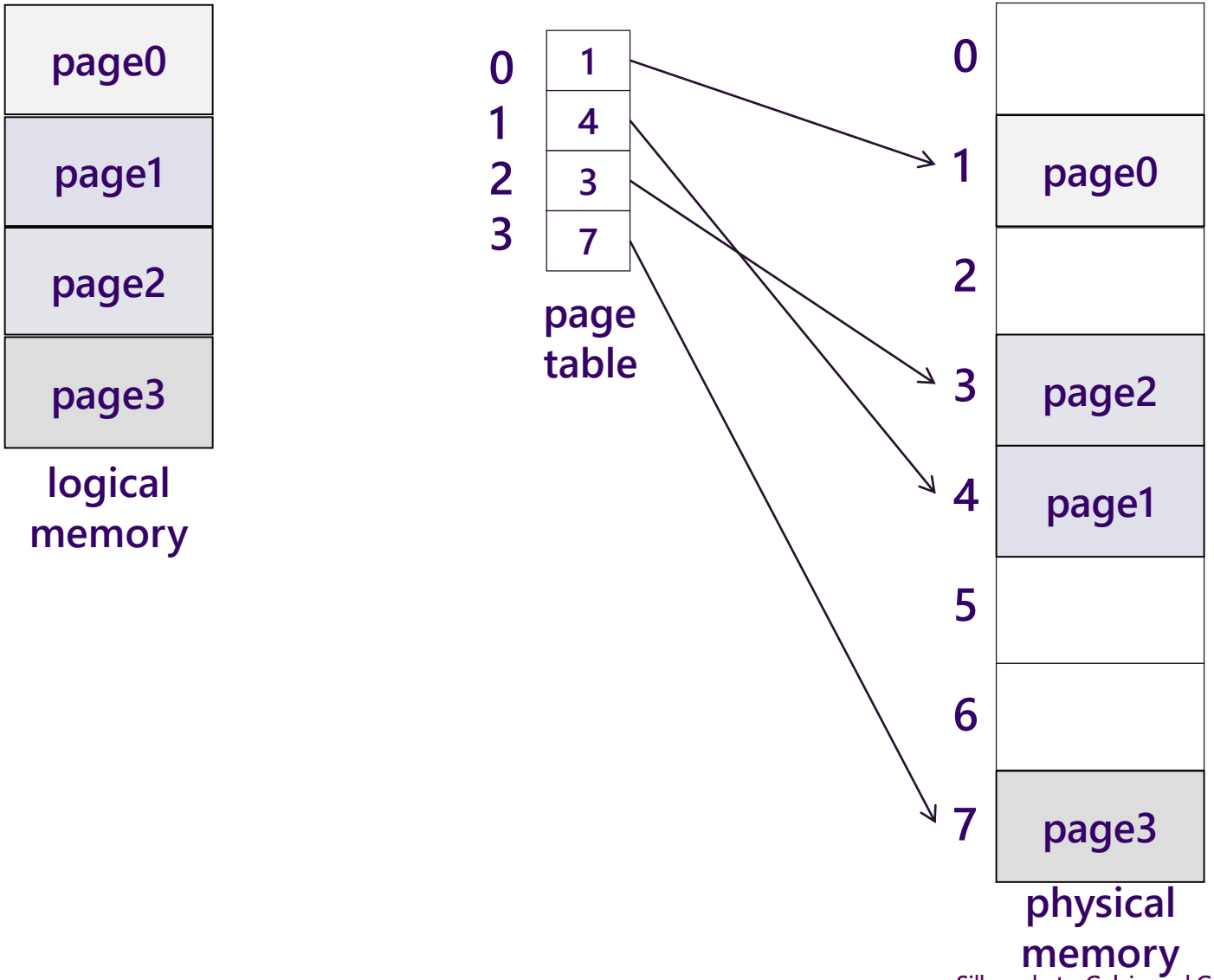
Paging

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- 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
- Can load pages on demand (Chapter 9)

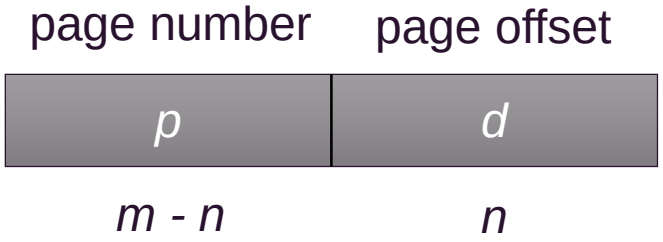
- Set up a page table to translate logical to physical addresses
- Still have Internal fragmentation, but only one partial page per process

Paging Model of Logical and Physical Memory



Address Translation Scheme

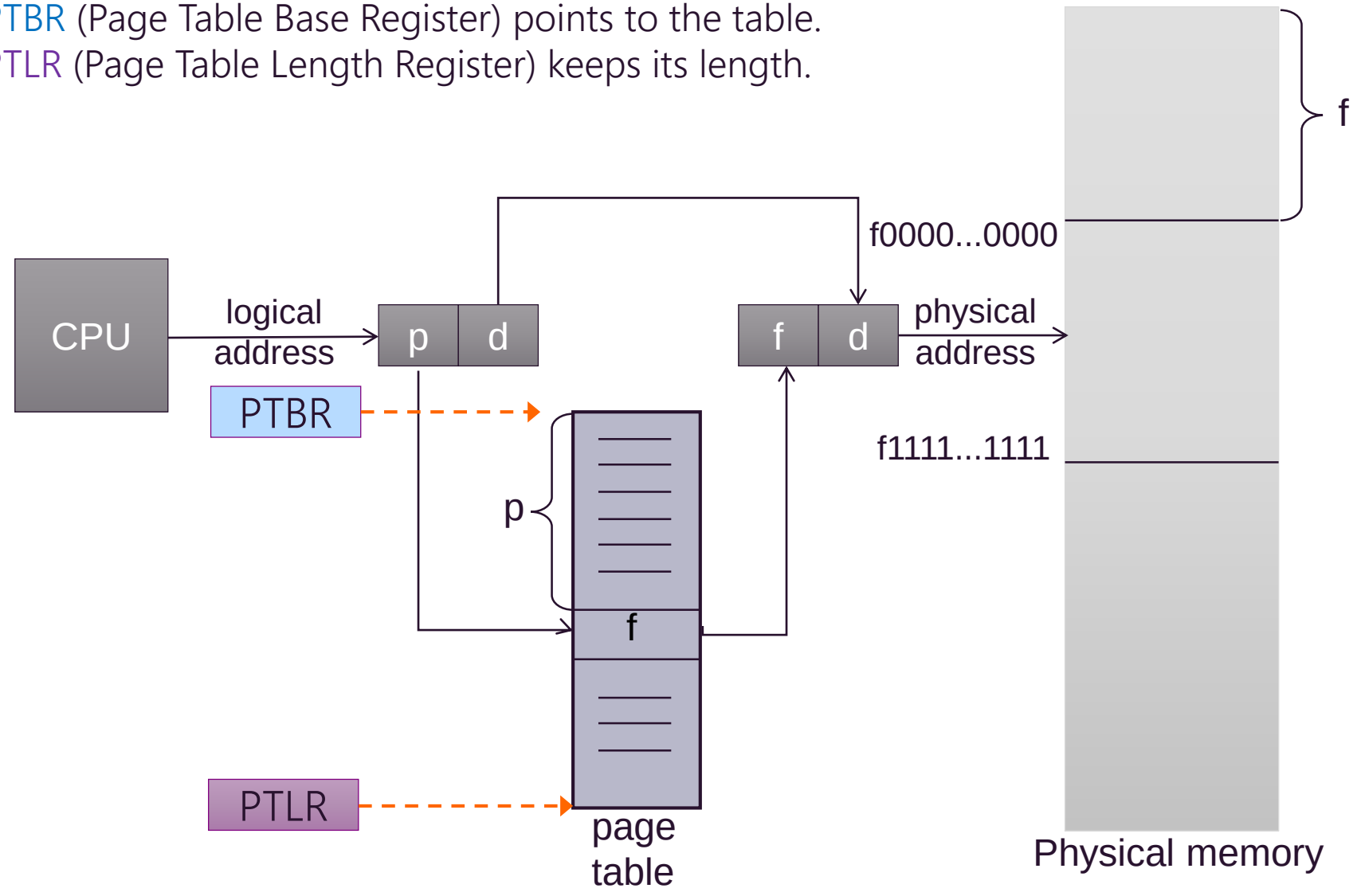
- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a ***page table*** which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit



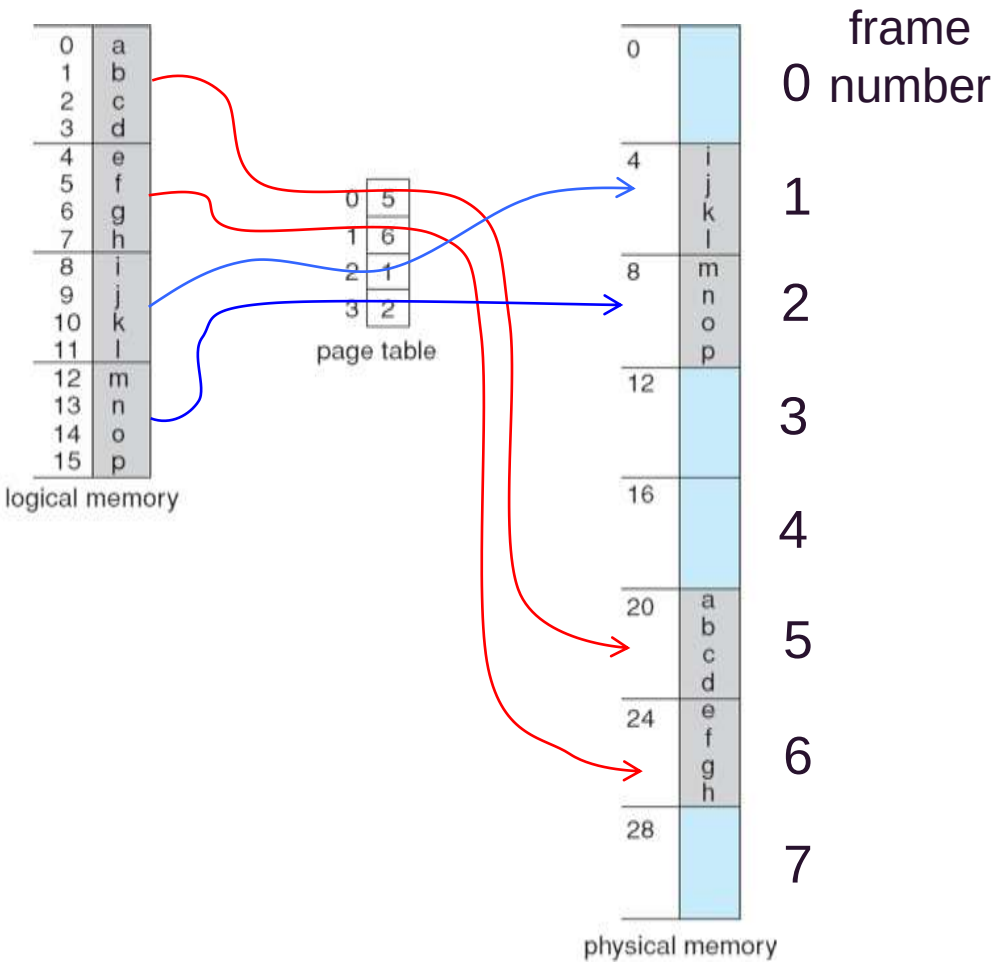
  - For given logical address space 2^m and page size 2^n
  - Example: with 4KB pages, n will be 12 (10^{12}=4096). With a 32 bit address space this will leave 20 bits for the page number (32-12)

Page Hardware

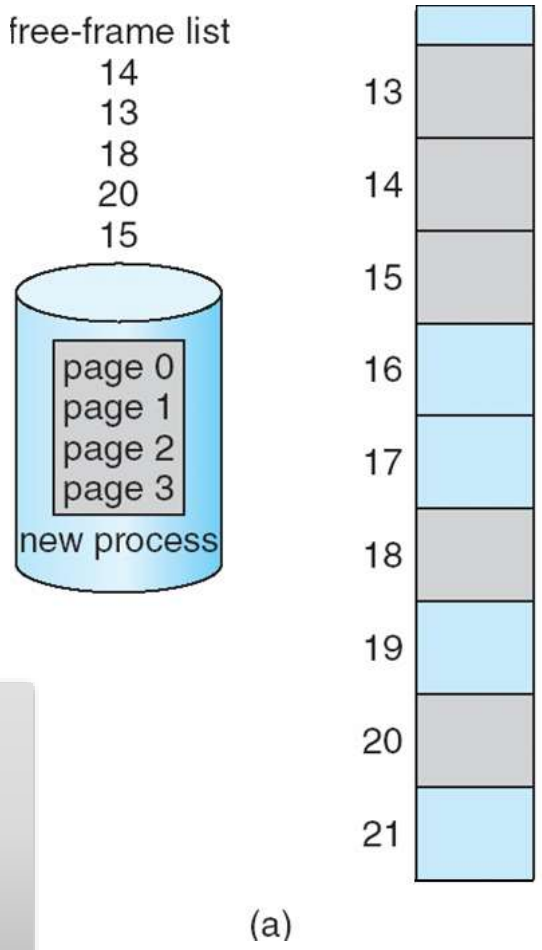
- PTBR (Page Table Base Register) points to the table.
- PTLR (Page Table Length Register) keeps its length.



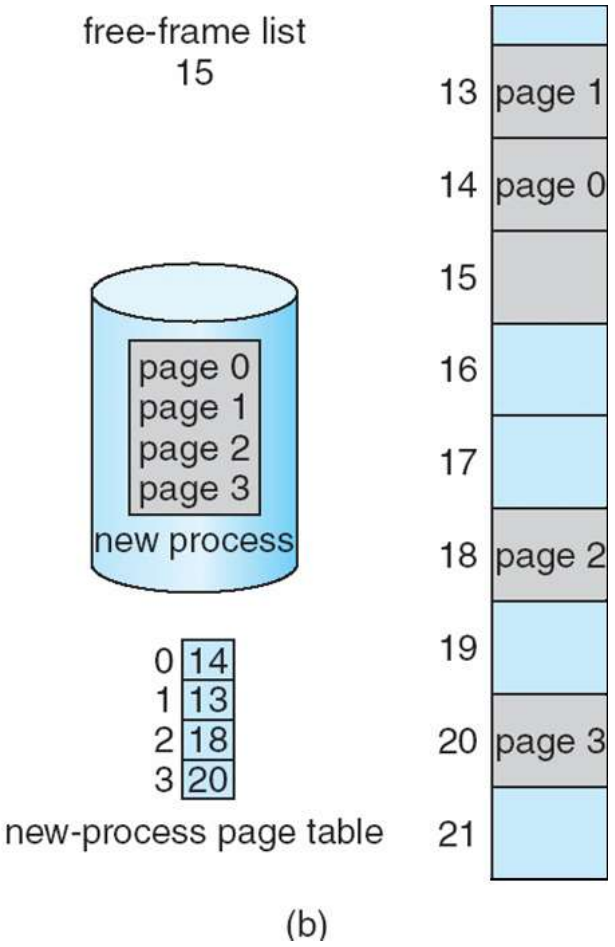
Paging Example



Free Frames



Before allocation



After allocation

Frame table stores allocation details of physical memory

For example: how many frames free/available)

Memory Allocation Strategies

Strategies	Key Idea	Pros	Cons
Contiguous	Allocate a logical space to a contiguous space of physical memory	■Semantically easy ■No memory fault	■External fragmentation ■No shared memory
Paging	Divide a logical space into pages and allocate them to physical space	■No more external fragmentation ■Shared pages available	■Internal fragmentation ■Page faults ■Two memory accesses ■No more semantically contiguous space
Segmentation	Give multiple contiguous logical spaces to each process	Semantically easy	External fragmentation if mapped like contiguous strategy

Modern Architecture: **Segmentation with Paging in TLB (Translation Look-aside Buffer) Support**

DISCUSSION

1. If you have a 16 MB logical address space and wish to have 8K byte pages for a logical to physical mapping, how many bits are allocated to index into the page table? register How many bits make up the page offset?
2. If you have a 32 bit logical address and an 4KB page, how many entries are there in a processes' page table?
3. How large must a page table be for 1K byte pages and a logical address space of 4GB?

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 (PRLR) 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)

- Some TLBs store address-space identifiers (ASIDs) in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative Memory
TLB (Translation Look-aside Buffer)

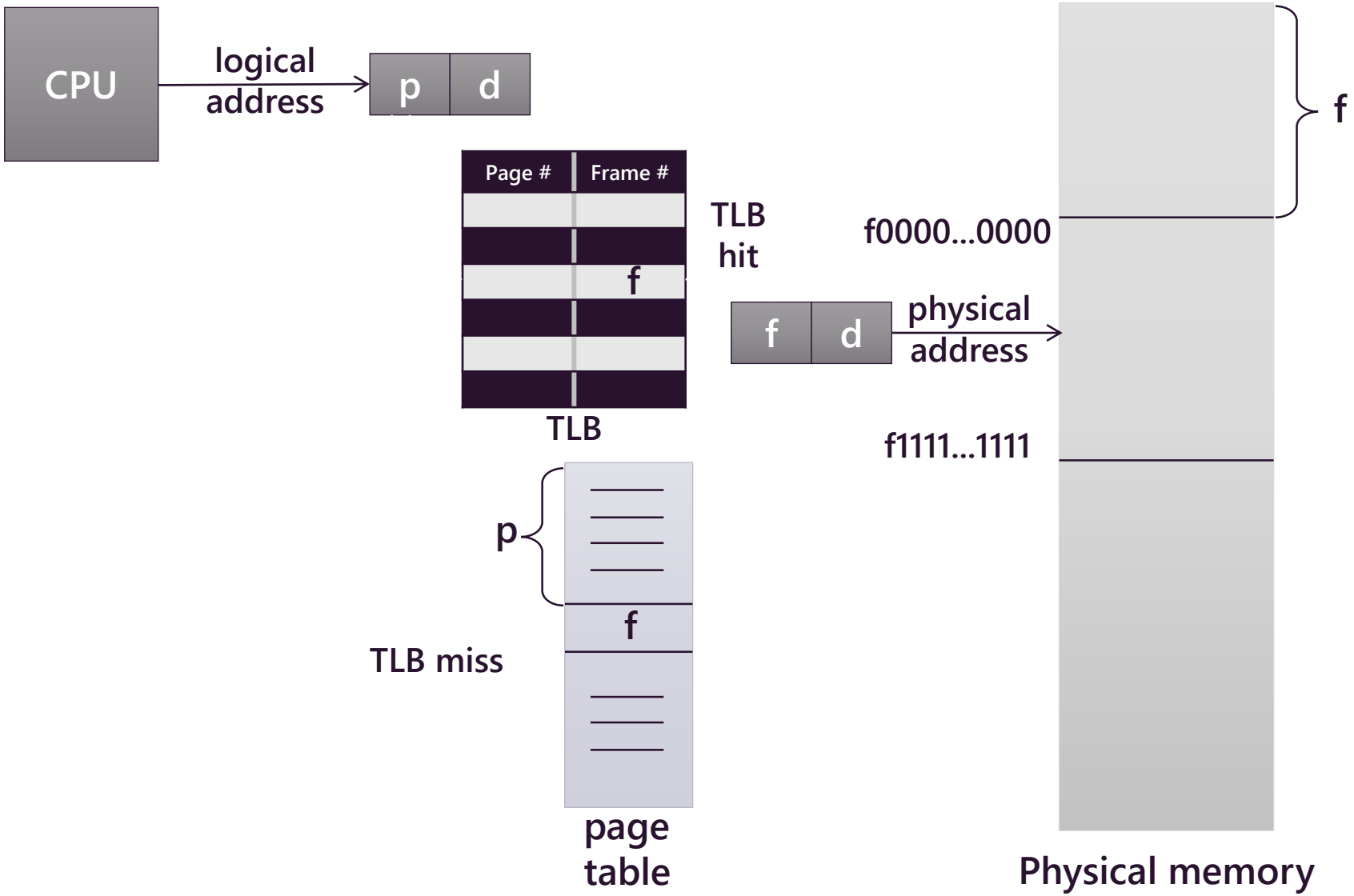
- Hardware support
- **Associative memory** – parallel search

Page #	Frame #

Address **translation** (p, d)

- If ***p*** is in associative register, get frame # (***f***)out
- Otherwise get **frame** # from page table in memory

Paging Hardware With TLB

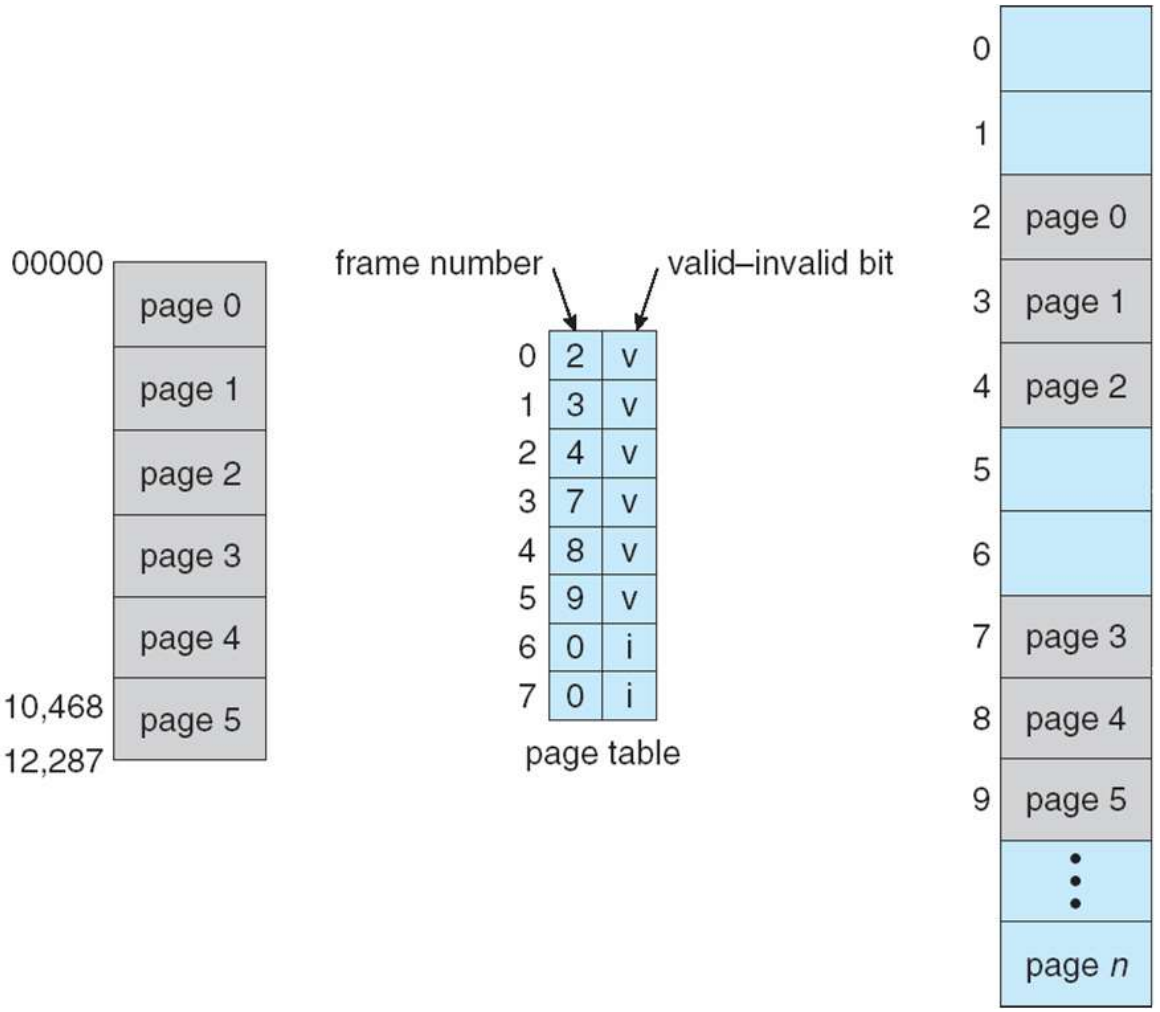


Memory Protection

- Memory protection implemented by associating protection bit with each frame

- 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

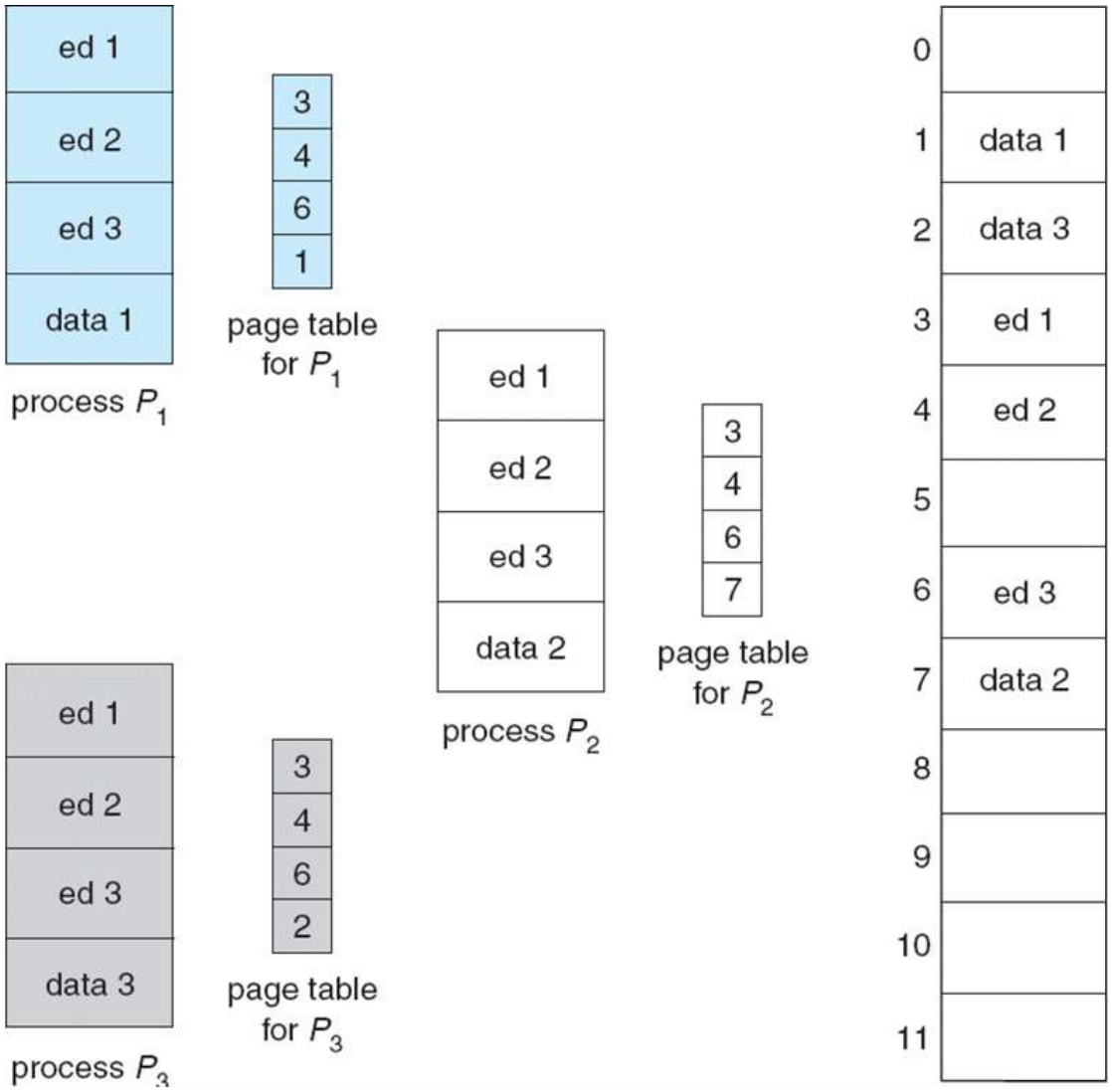
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in same location in the logical address space of all processes
- 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

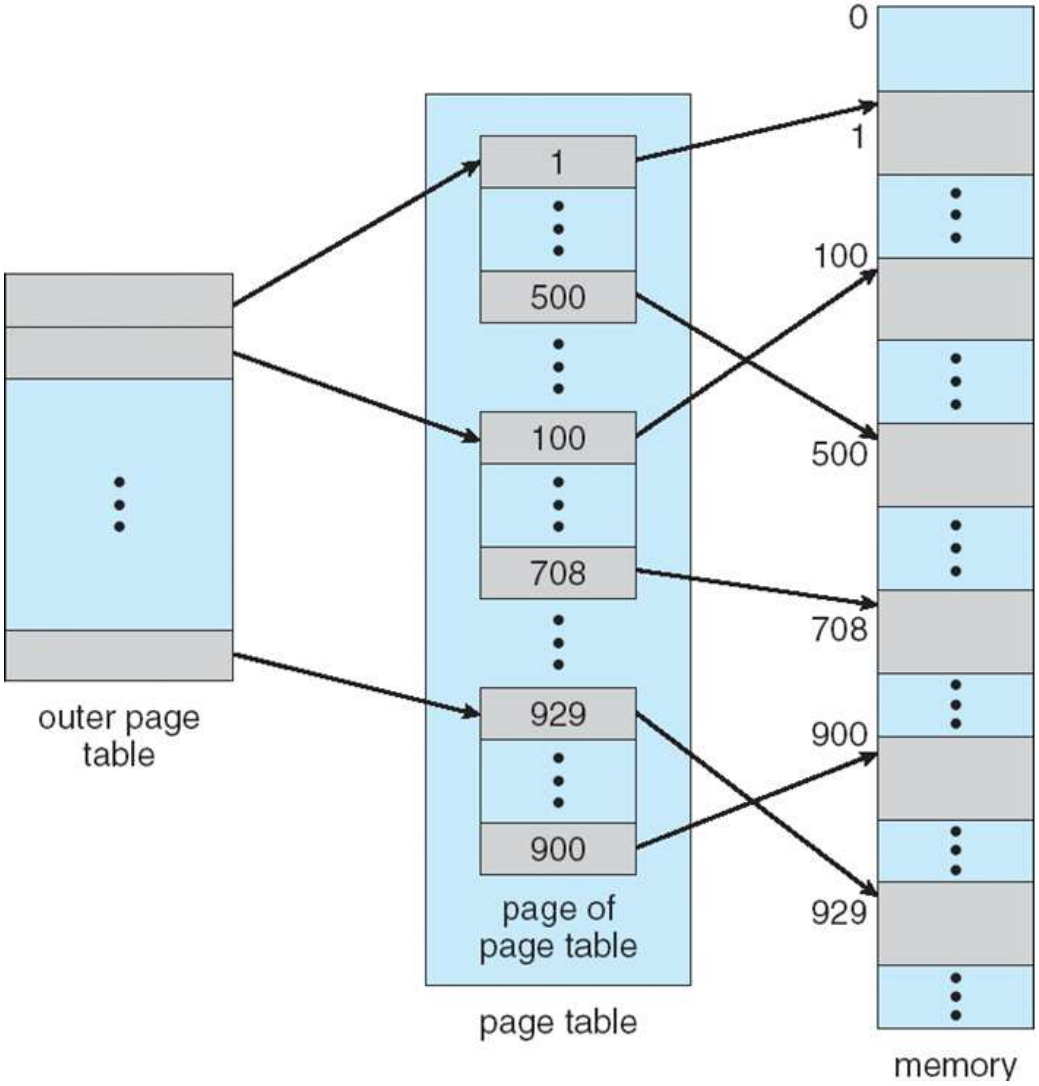
Shared Pages Example



Page Table Structures

- Large logical spaces require significant memory for page tables
  - Logical space usage may be sparse as well
  - Technique presented thus far unusable for large address spaces
    - 32 bit address space w/ 4 KB pages ➔ 4 MB page table
- Approaches include more efficient page table structures
  - Hierarchical Paging
  - Hashed Page tables
  - Inverted Page Tables

Hierarchical Page Tables
Two-level Page-Table Scheme



Two-Level Page-Table Scheme

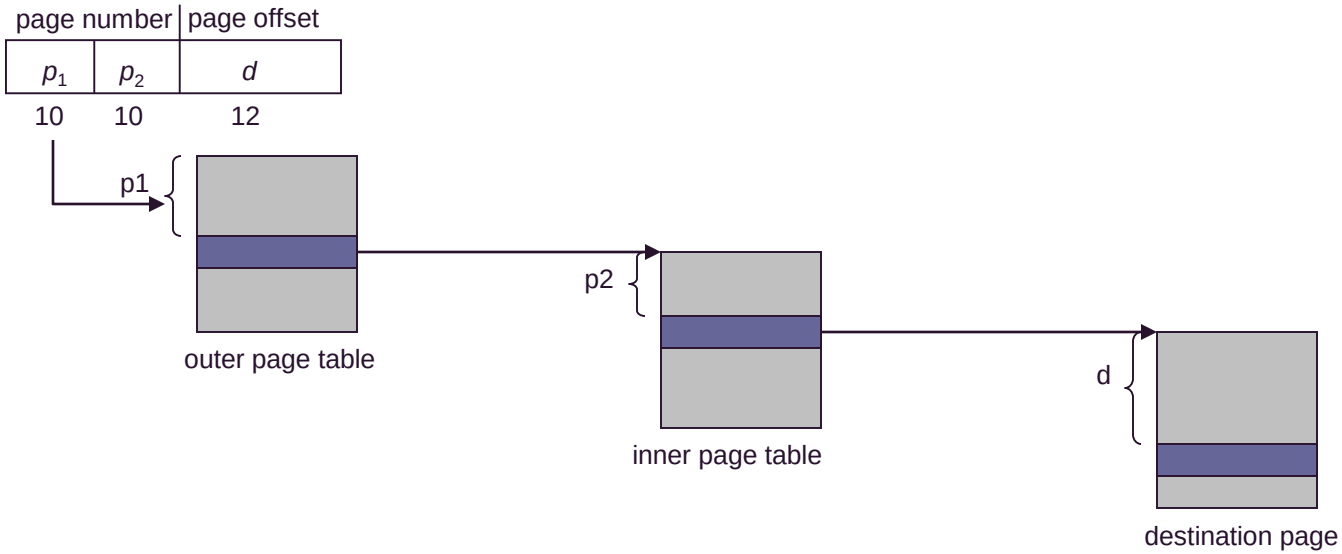
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
  - If each page entry requires 4 bytes, the page table itself is 4M bytes!
- Two-level page-table scheme
  - Place another outer-page table and let it page the inner page table
  - the page number is further divided into:
    - a 10-bit page number. (1K entries)
    - a 10-bit page offset. (1K entries)
- Thus, a logical address is as follows:

page number		page offset
p_1	p_2	d
10	10	12

P1: outer page index
P2: page index
D: offset in a page

Two-Level Page-Table Scheme

- Address-translation scheme for a two-level 32-bit paging architecture
  - Outer-page table with 1K entries: size = at least 4K
  - Inner page table with 1K entries: size = at least 4K
  - If a process needs only 1K page outer/inter page tables require only 8K, and thus the total process size = 9K.
- More multi-level paging:
  - Linux (three levels: level global, middle, and page tables)
  - Windows (two levels: page directory and page tables) etc.



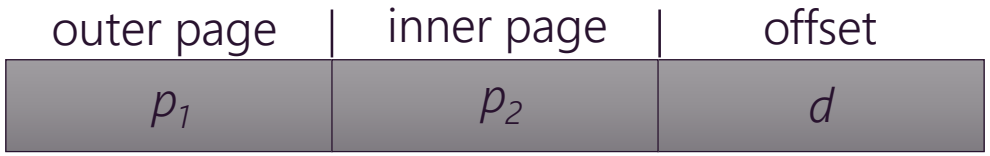
DISCUSSION / POP QUIZ

1. What is the difference between swapping and paging?
2. What are three contiguous-memory allocation algorithms discussed? Which is best?

3. If a 32bit system uses a two level page table with 4 KB pages and an outer page table of 1024 entries, how big is the page table?

Three-level Paging Scheme

- A logical address (on 64-bit machine with 4K page size) is divided into:
  - An outer page number consisting of **42** bits (2^{42} Pages of Page Tables!)
  - a page offset consisting of 12 bits



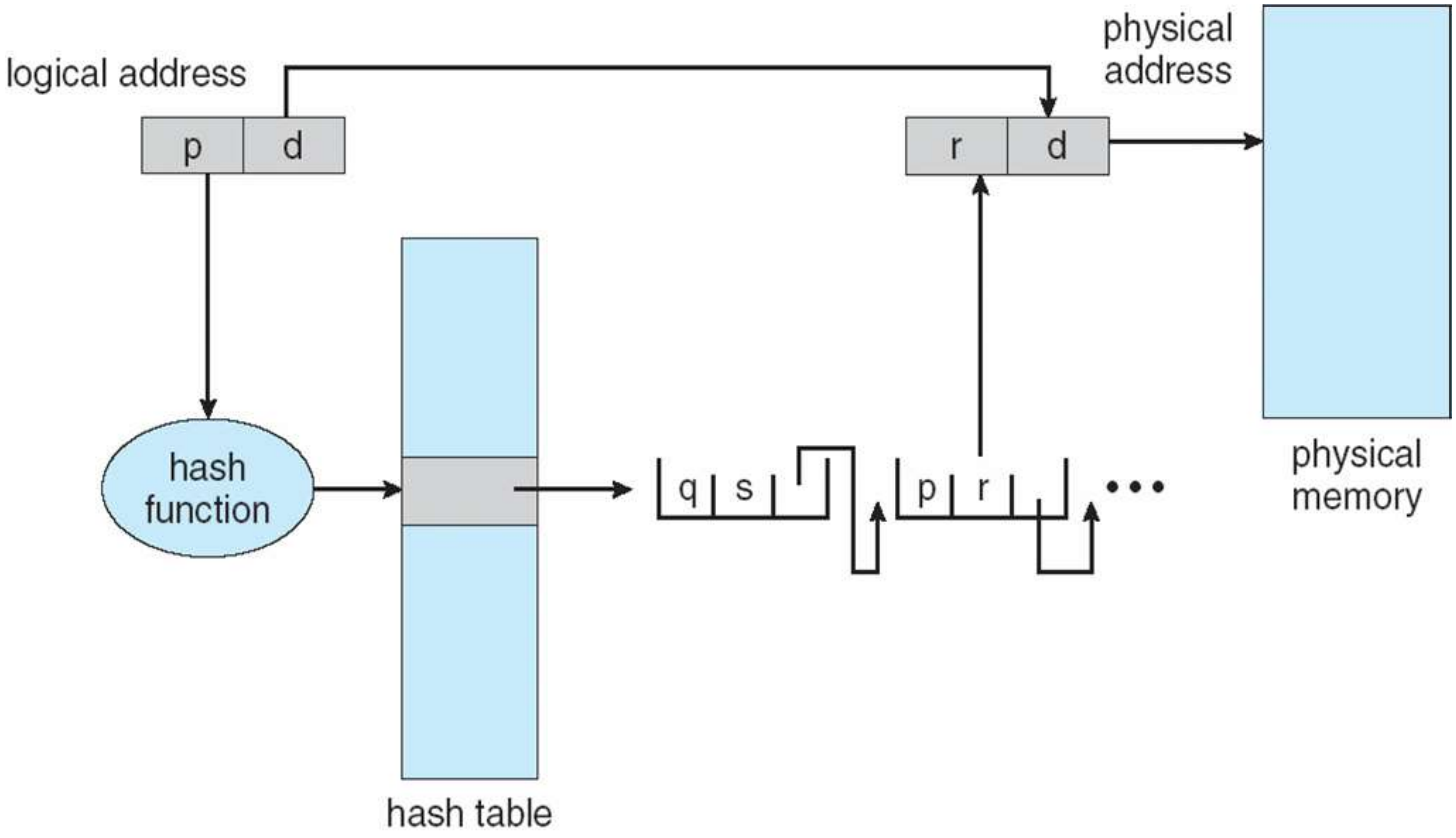
Add another level?

Still too large – need a better way!

Hashed Page Table

- Common in address spaces > `32 bits` (e.g. 64-bit architecture)
- A virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted (via lookup of the frame # in the hash table)
- Another option is cluster page tables using one virtual page number to link to several pages.

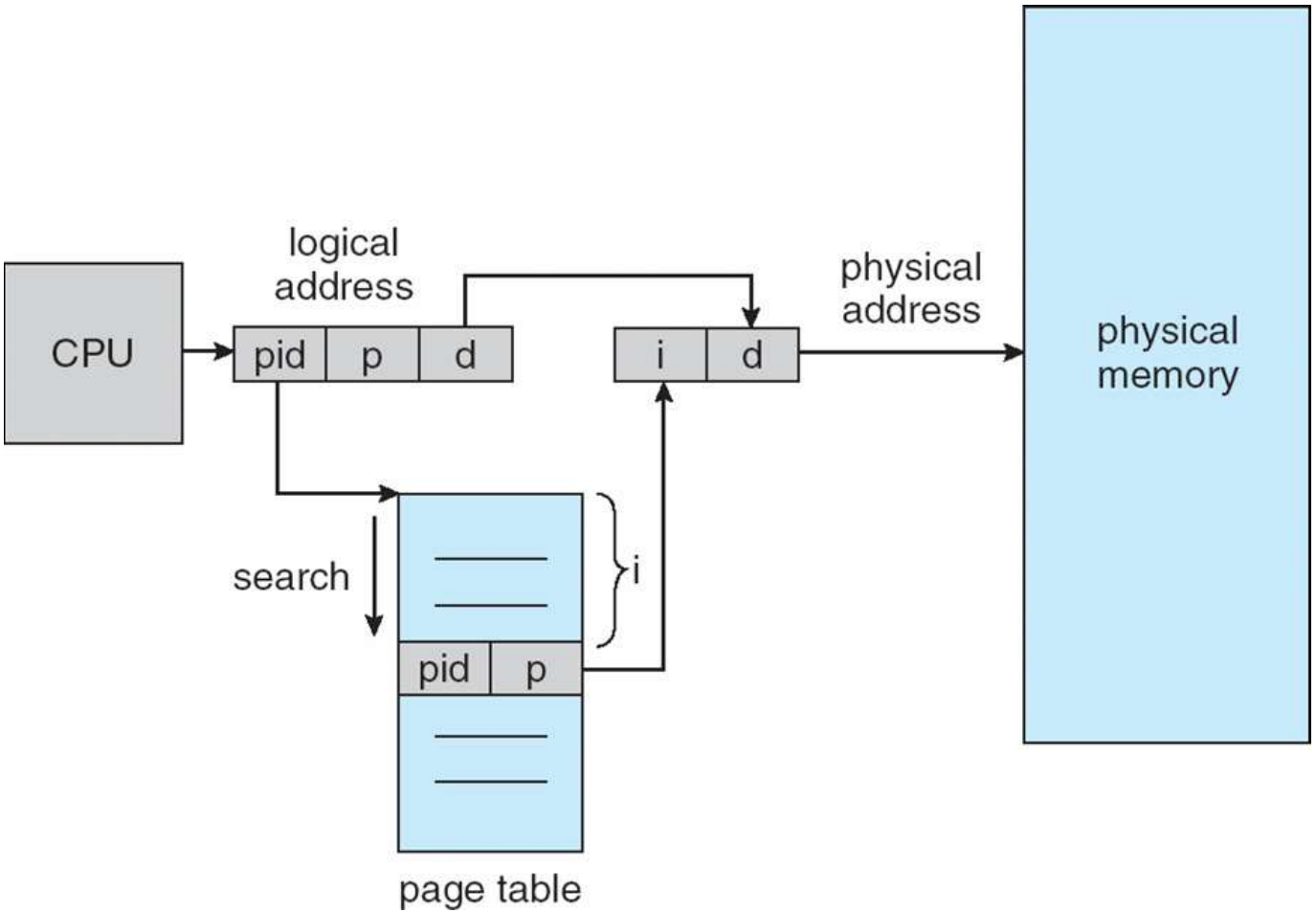
Hashed Page Table



Inverted Page Table

- 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

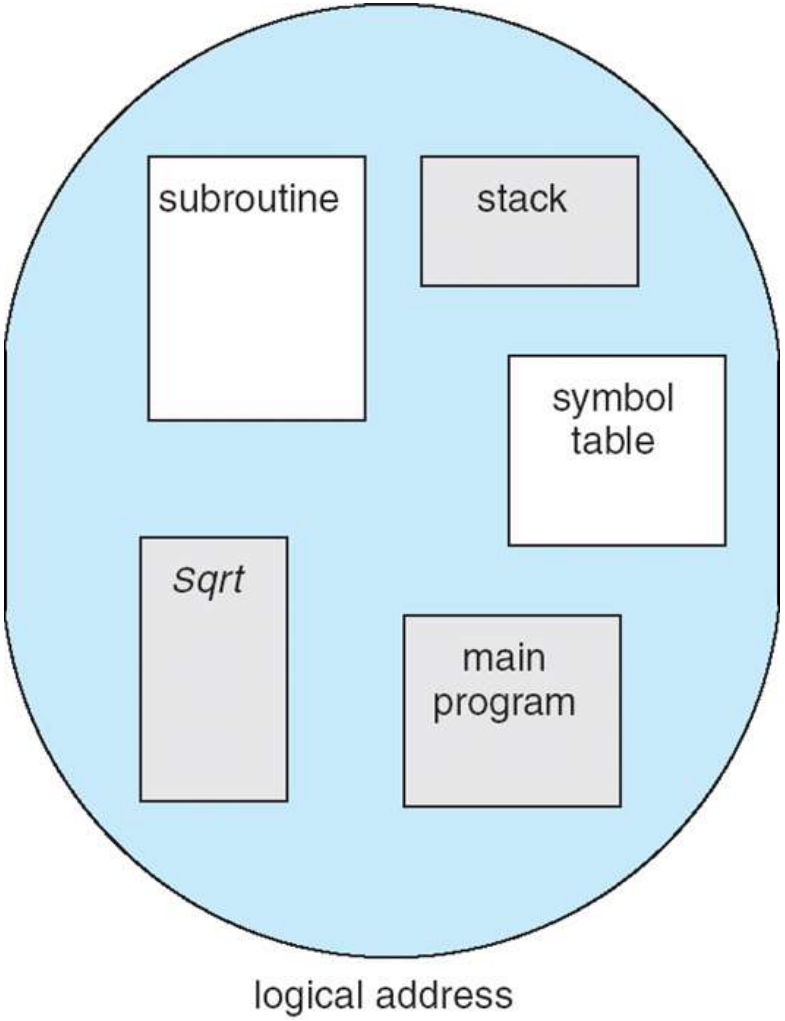
Inverted Page Table Architecture



Segmentation

Segmentation

- Programmers don't think of memory as a linear array of bytes.
- They think of it as multiple segments such as (examples):
  - The Code
  - Global Variables
  - The heap
  - Standard C library
  - Main function
- Segmentation memory-management supports programmers' view of memory



Segmentation

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

    main program

    procedure

    function

    method

    object

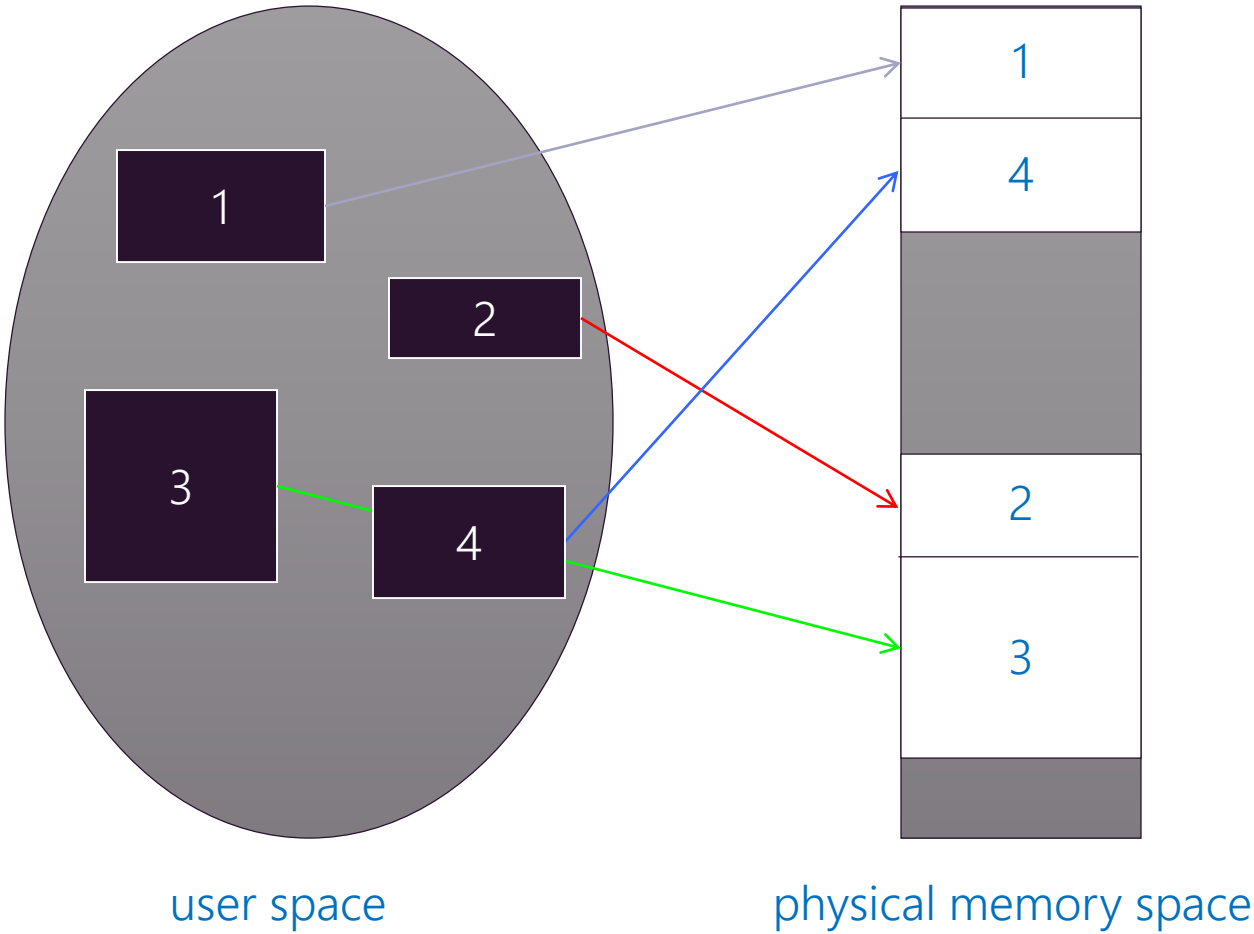
    local variables, global variables

    common block

    stack

    symbol table

    arrays



Segmentation Architecture

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

  `<segment-number, offset>`

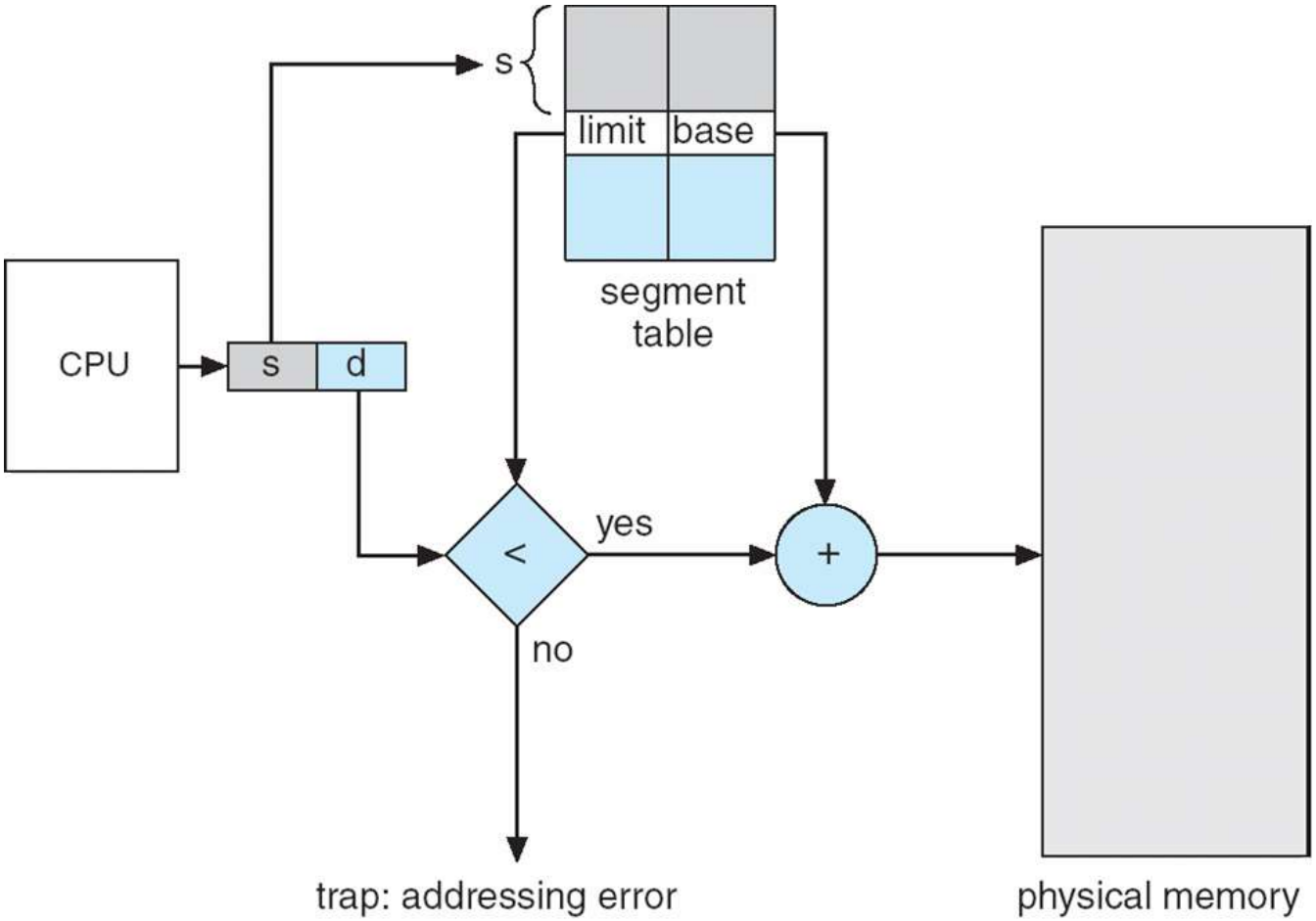
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

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

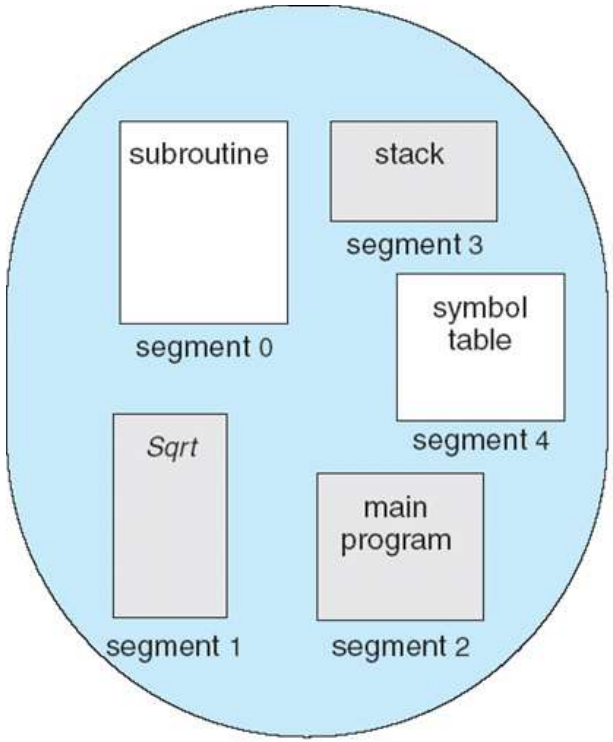
Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ 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

Segmentation Hardware

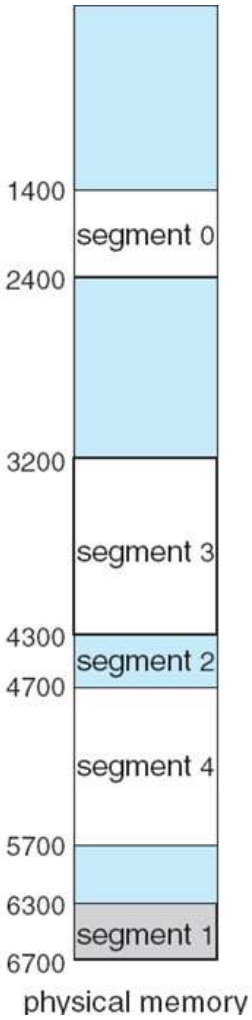


Example of Segmentation



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

segment table

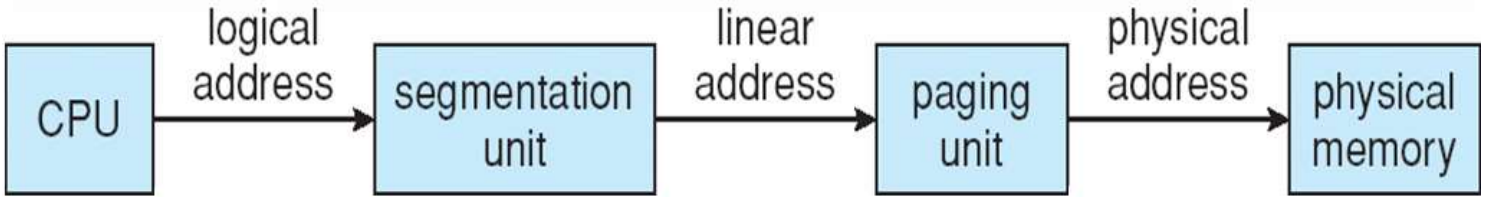


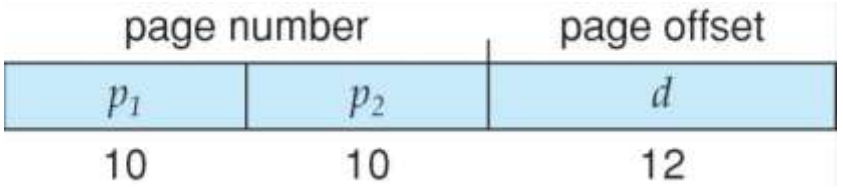
x86 Intel Architecture

Paging and Segmentation in ACTION!

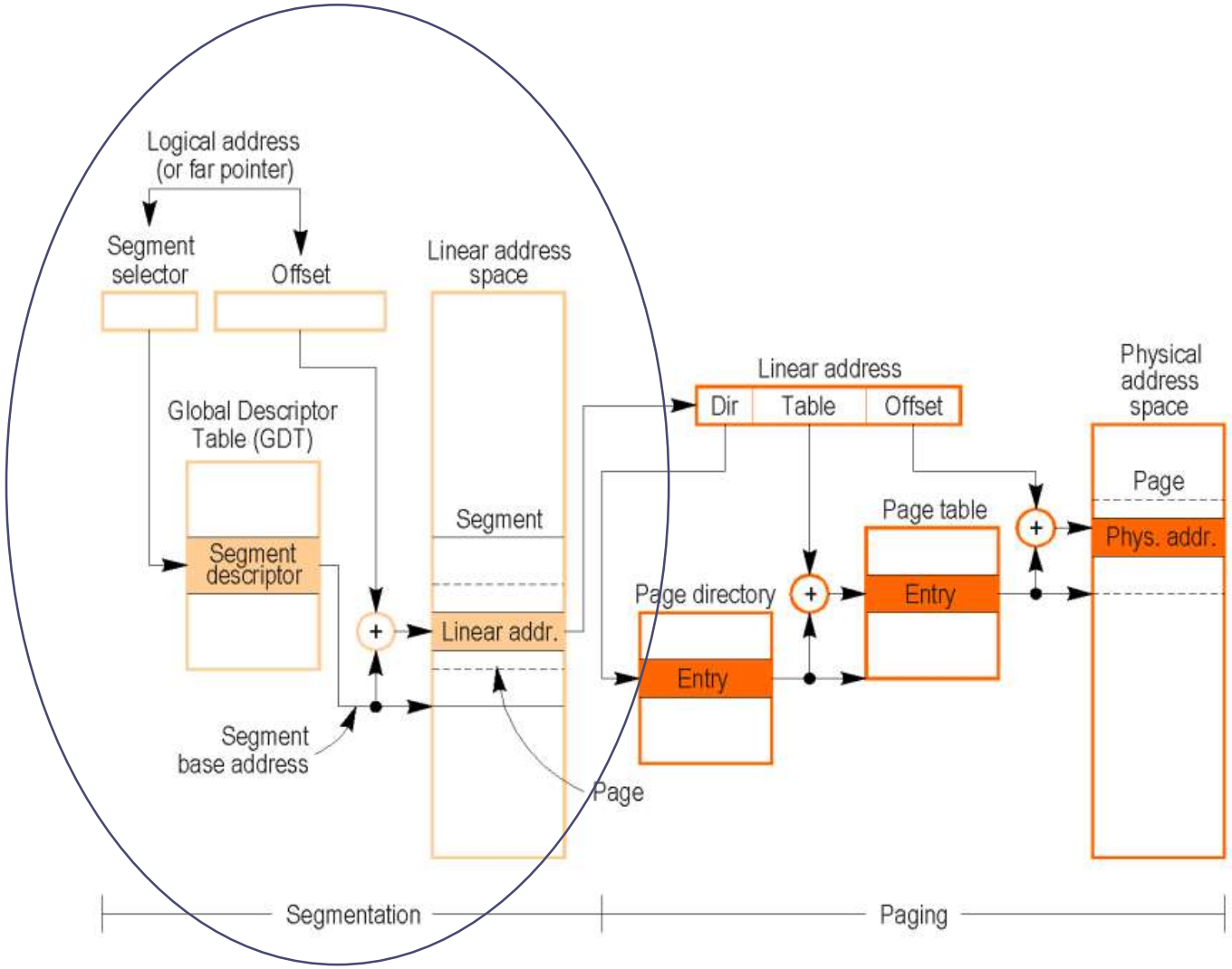
The Intel Pentium

- Supports both **segmentation** and ***segmentation with paging***
- CPU generates **logical address**
  - Given to segmentation unit, which produces ***linear addresses***
  - Linear address given to **paging unit**
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU

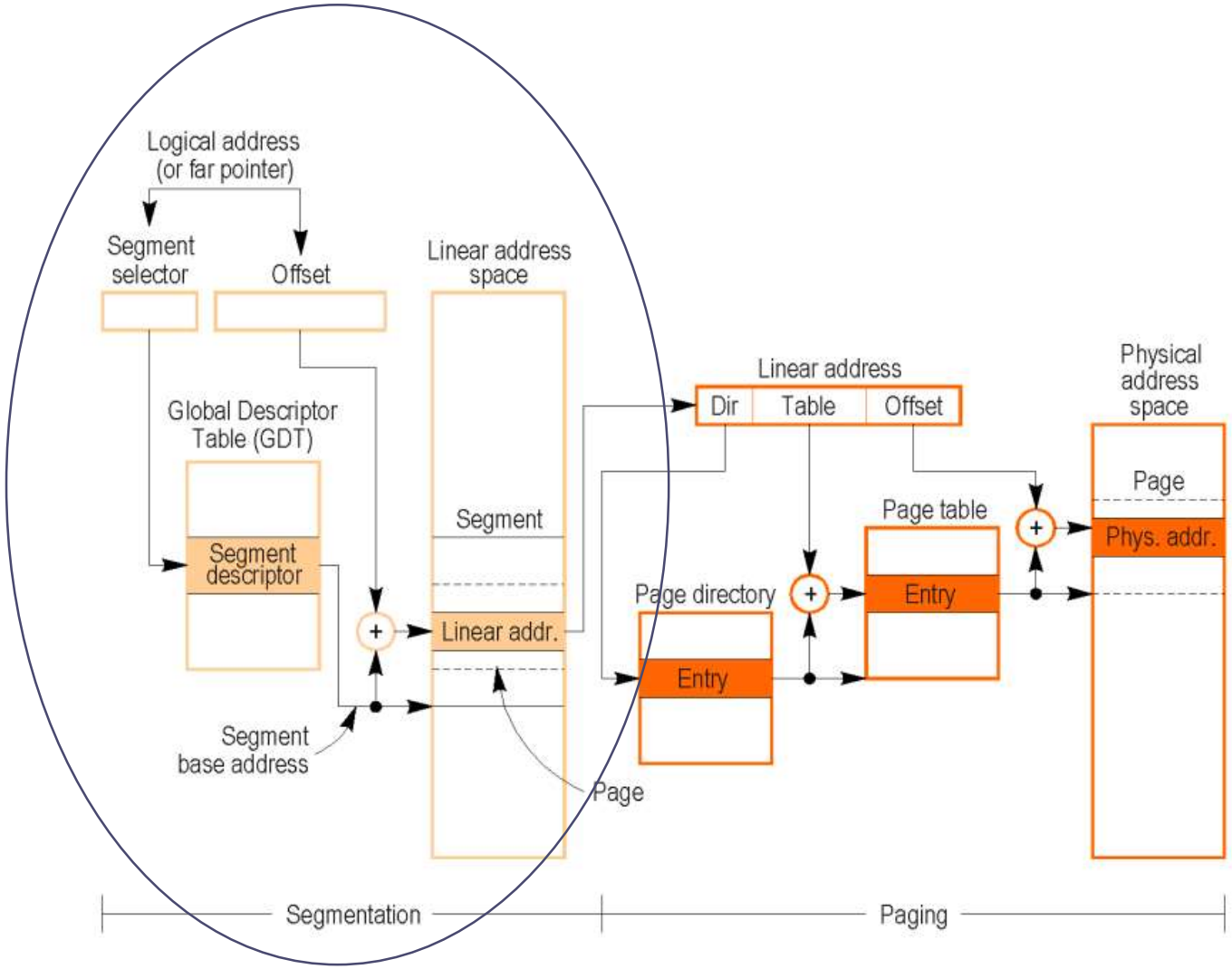




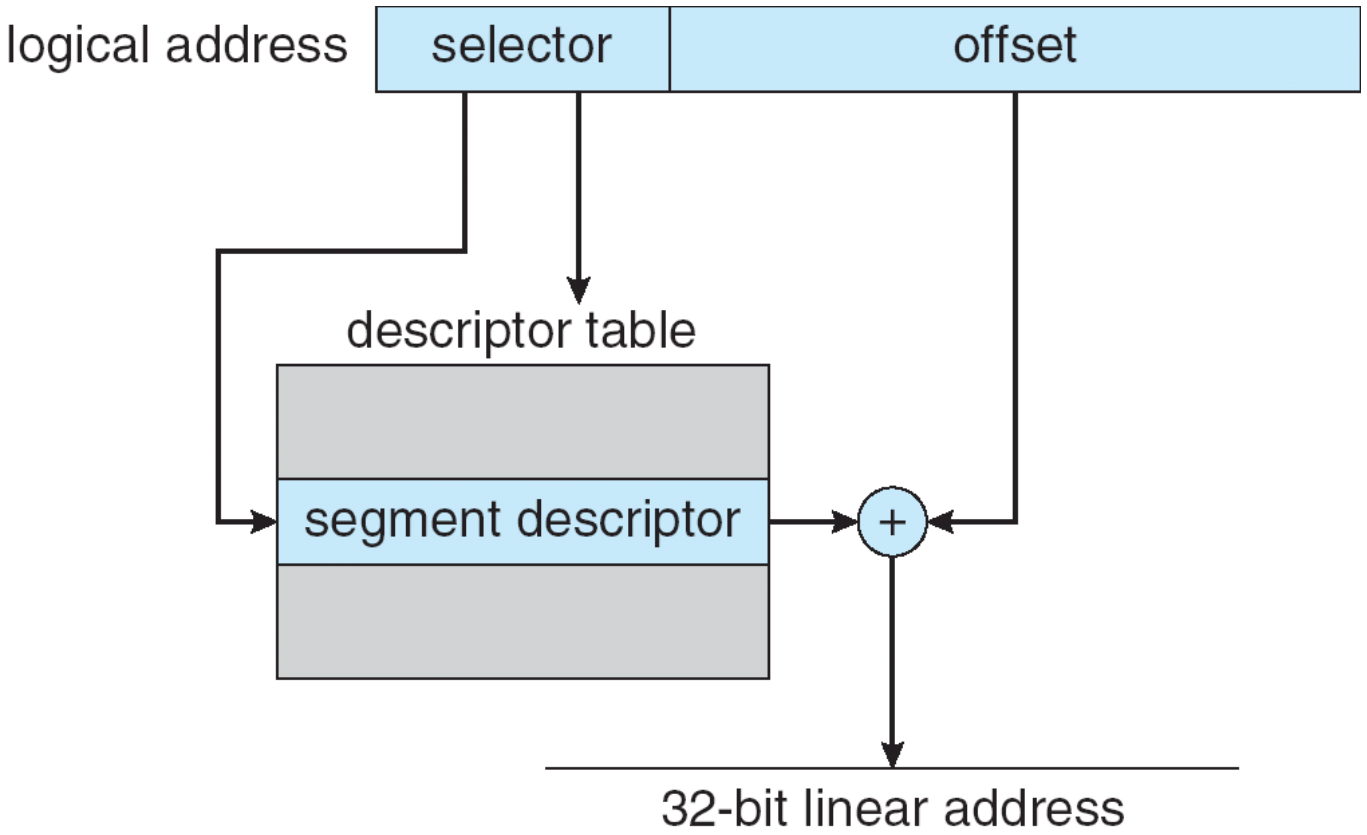
Segmentation + Paging
Intel Pentium



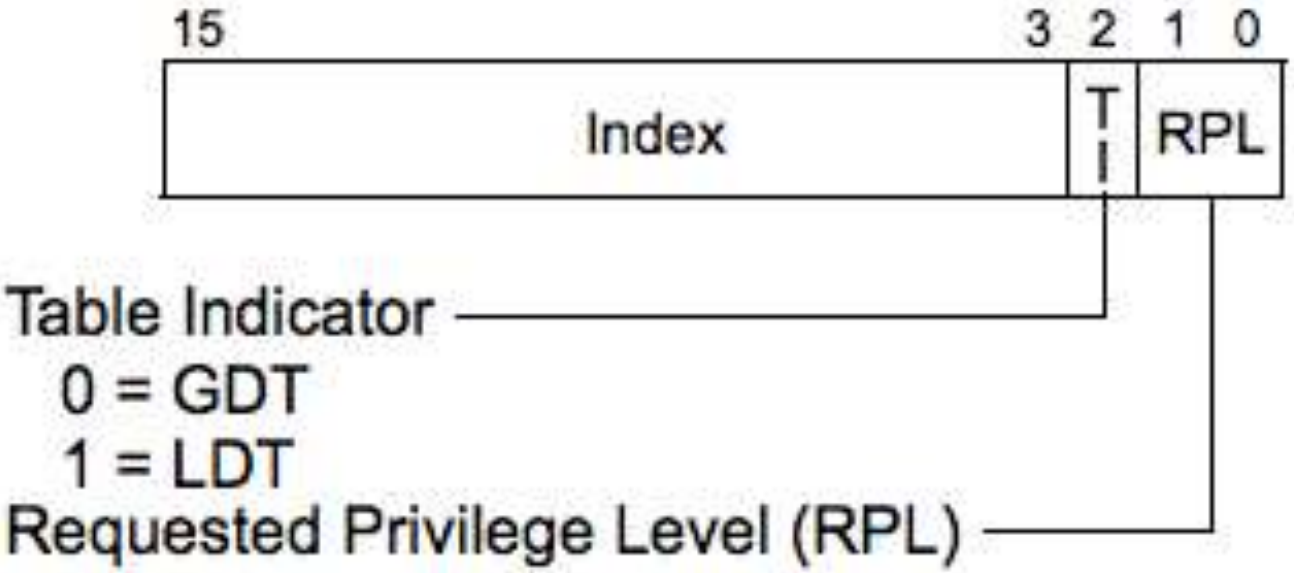
Segmentation + Paging
Intel Pentium



Intel Pentium Segmentation



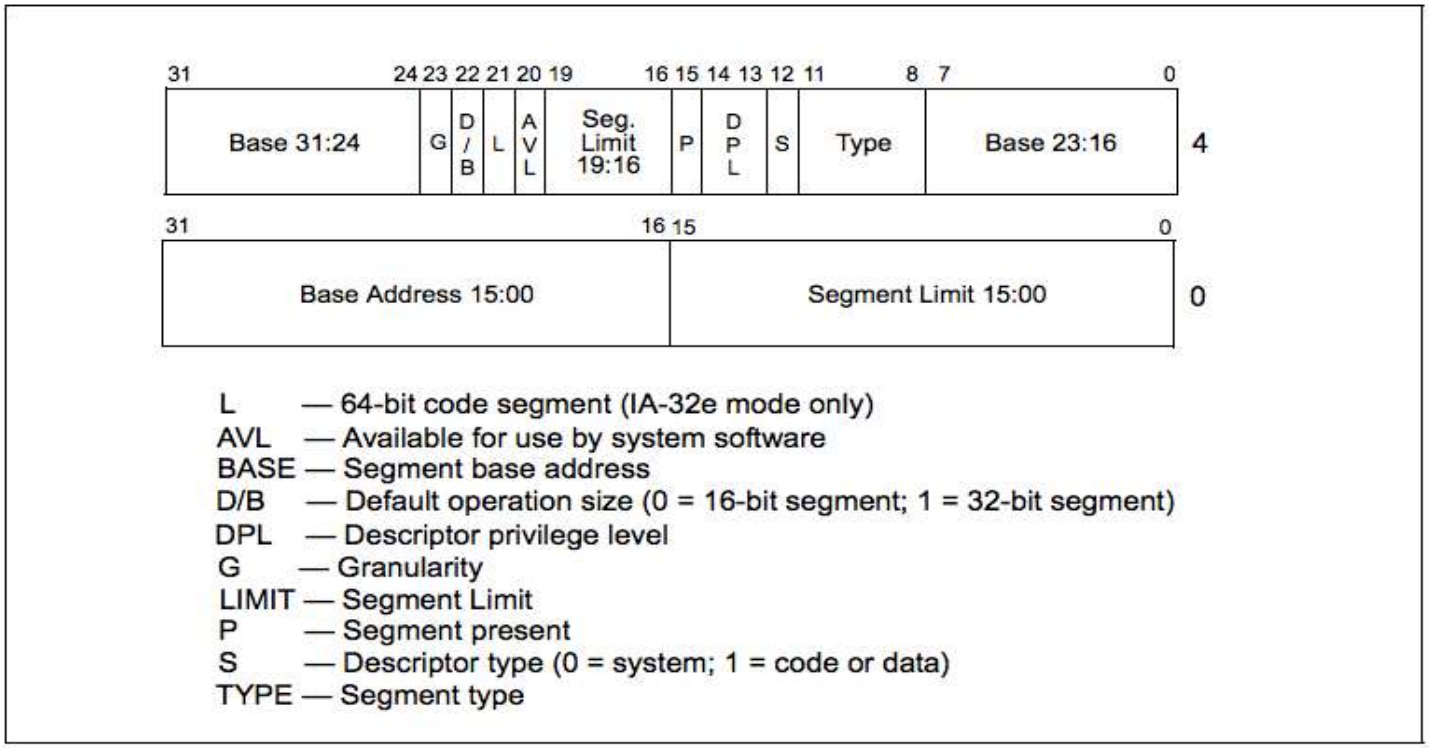
Segment Selector



GDT / LDT

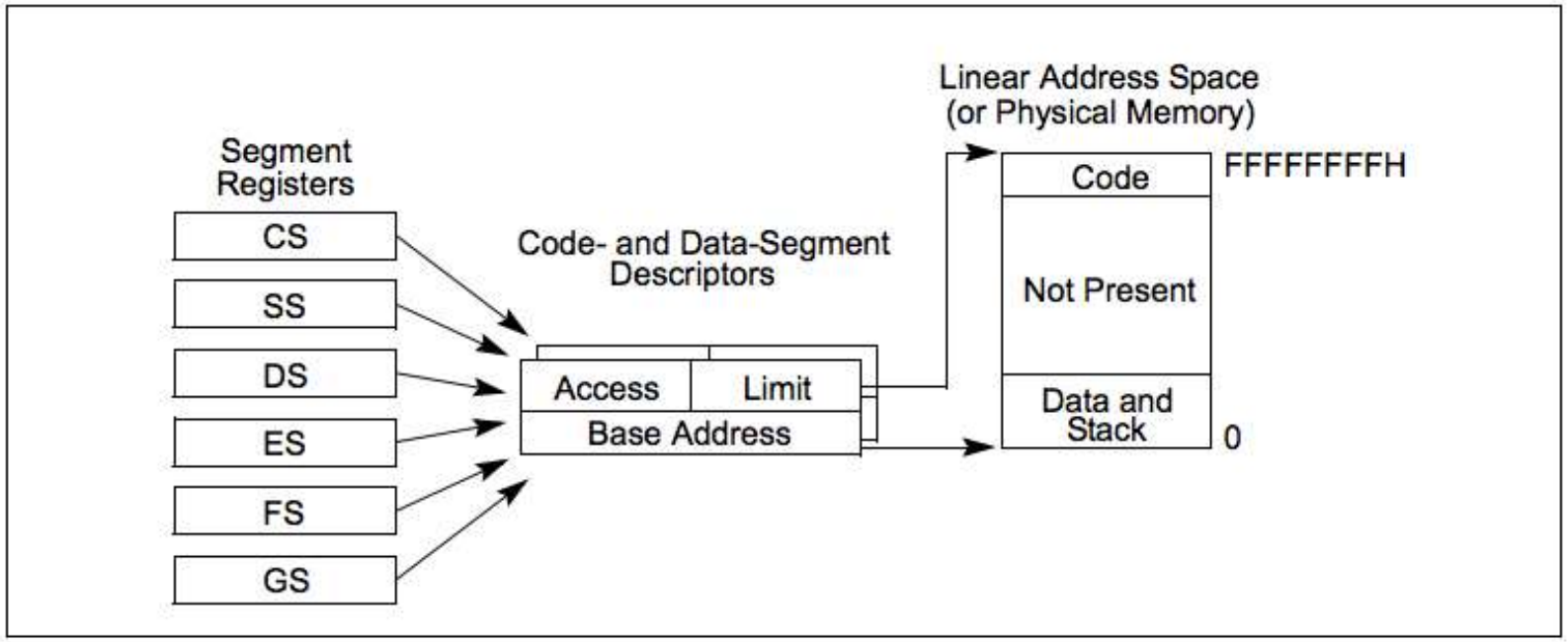
- Global Descriptor Table – System Wide
  - Each system must have one GDT defined, which may be used for all programs and tasks in the system.
  - 8 K segments that are shared
- Local Descriptor Table – Local User program
  - 8 K Segments that are private to the process

Segment Descriptor

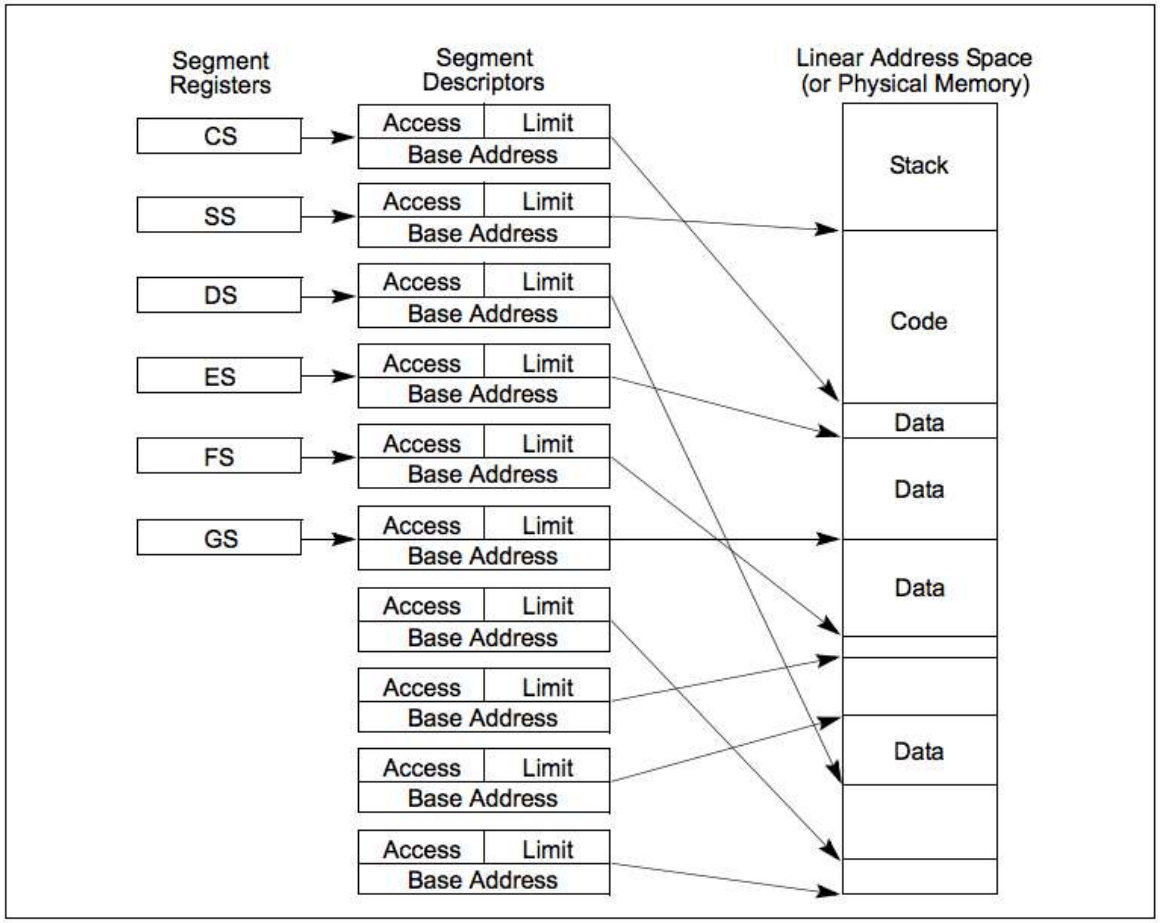


- Data structure in a GDT or LDT provides the processor with the size and location of a segment, as well as access control and status information.
- Segment descriptors are typically created by compilers, linkers, loaders, or the operating system or executive, but not application programs.

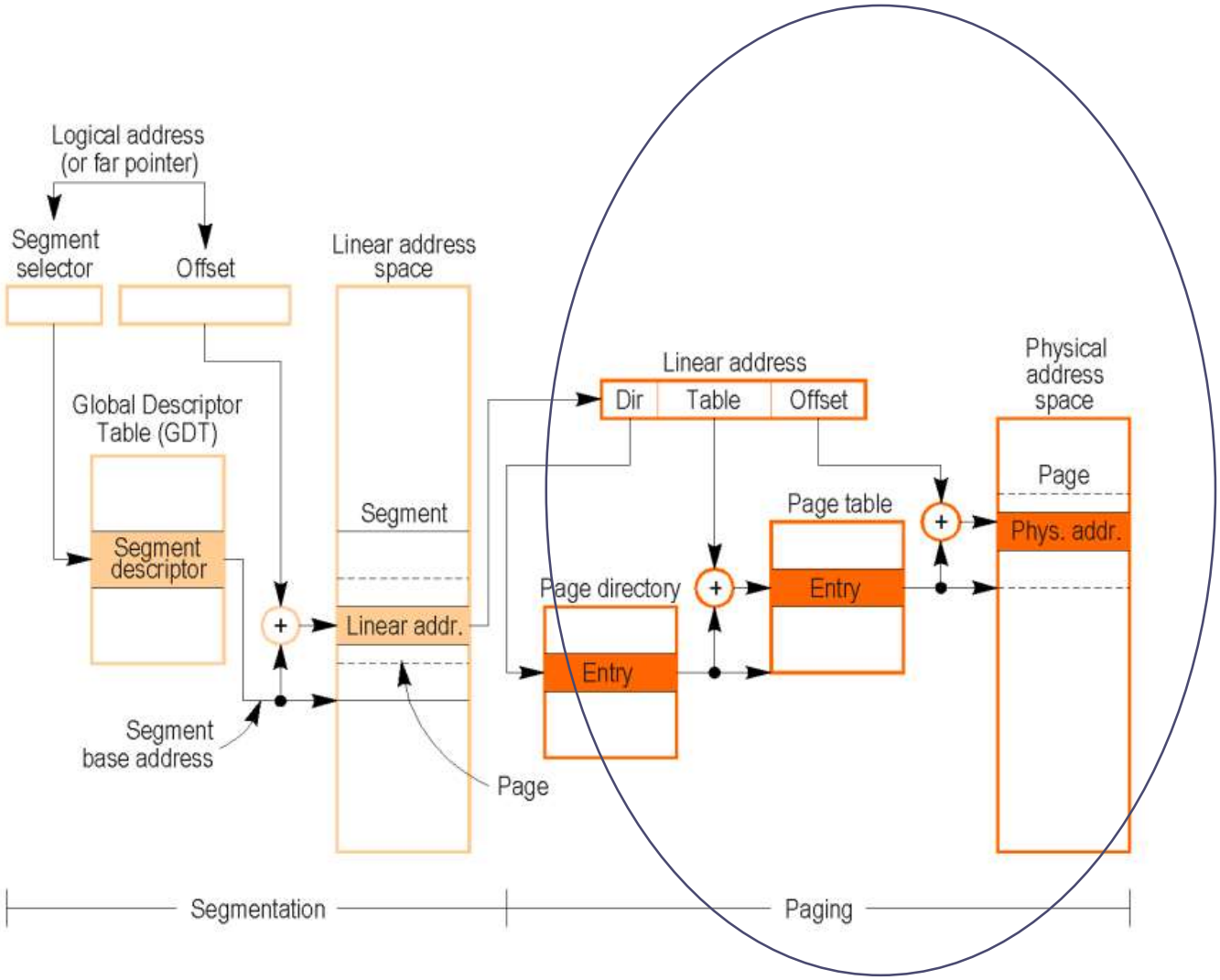
Basic Flat Model



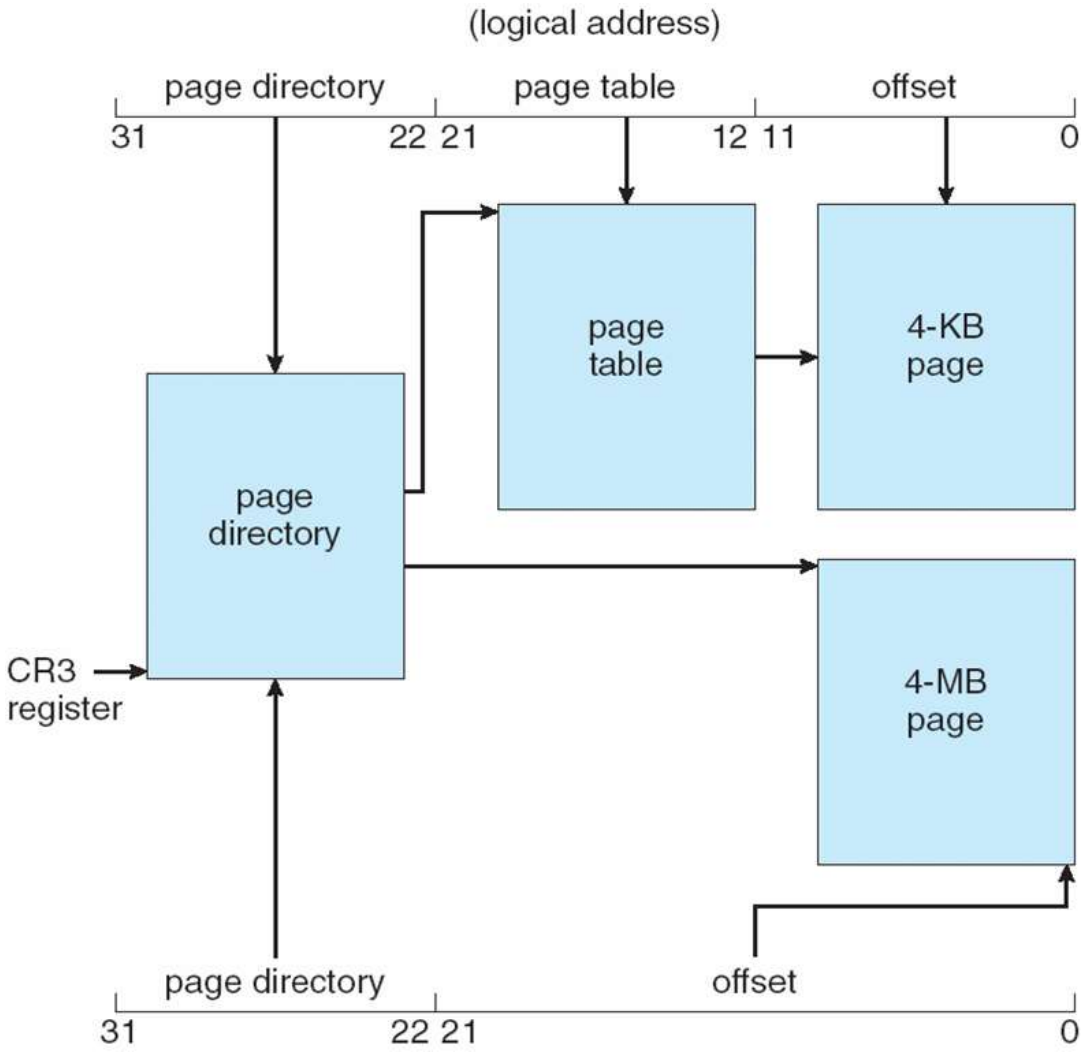
Multi-Segment Model



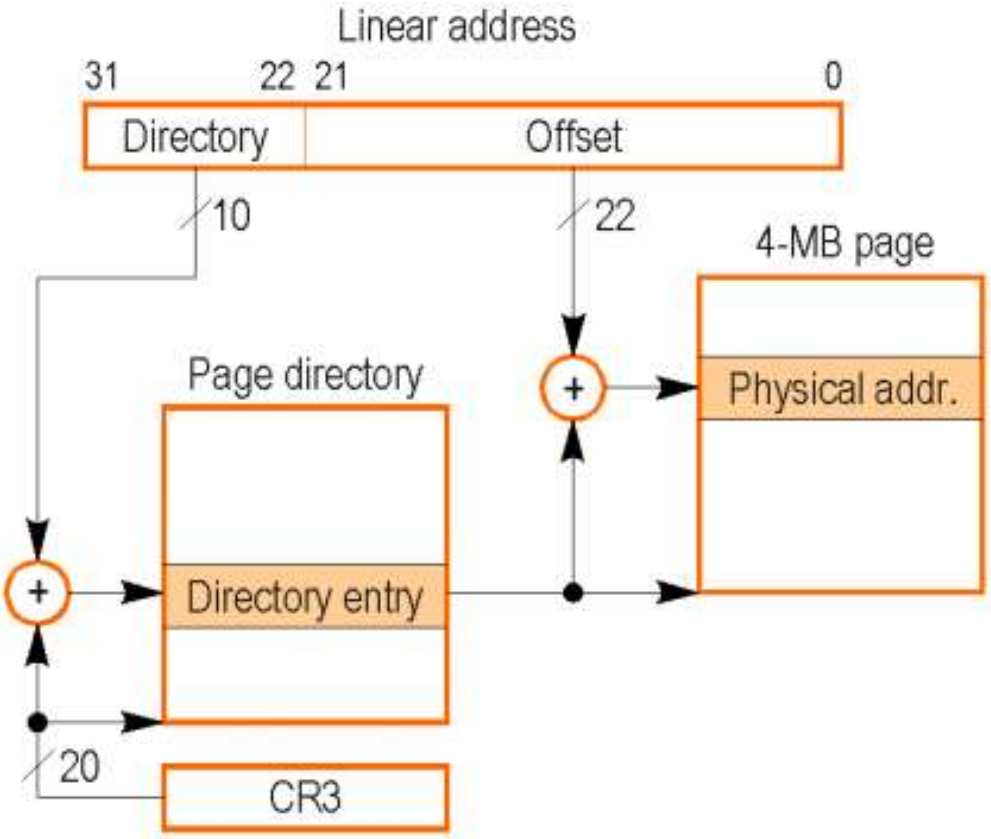
Segmentation + Paging
Intel Pentium



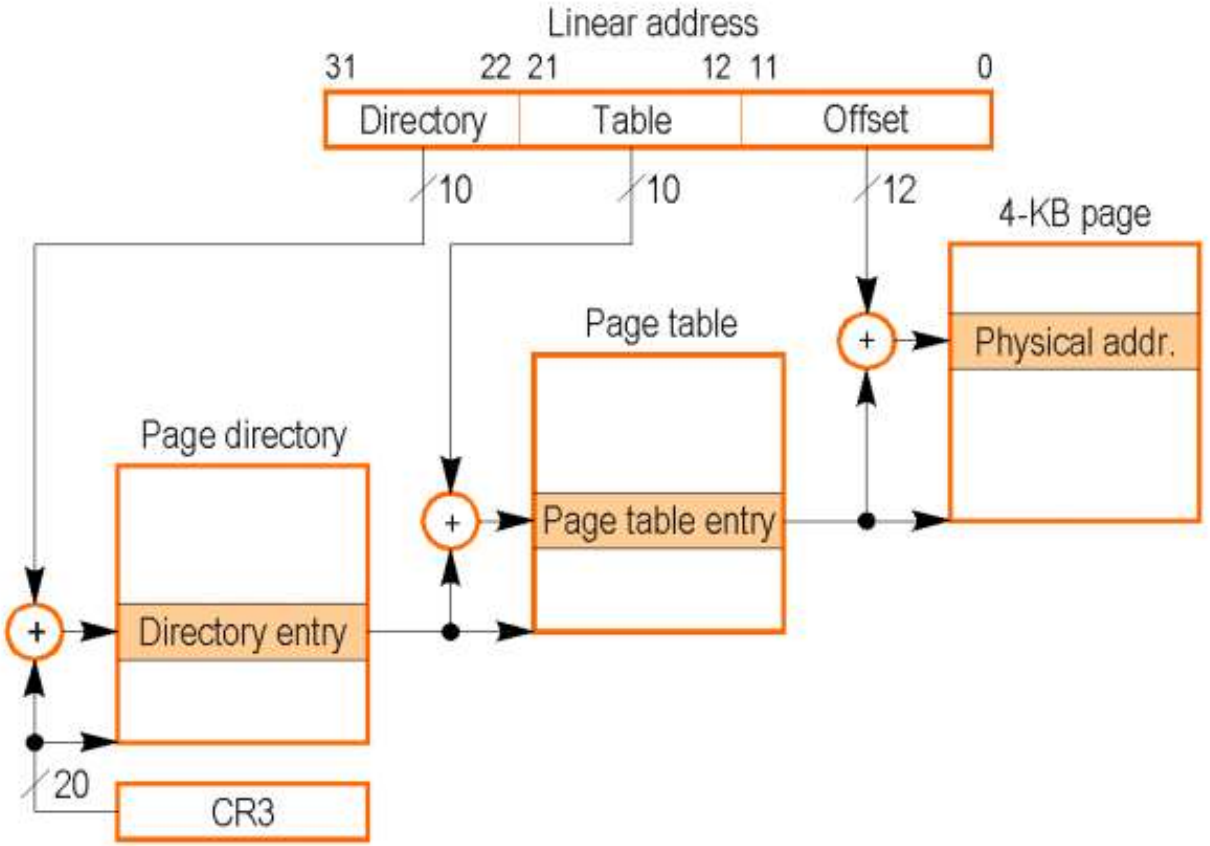
Pentium Paging Architecture



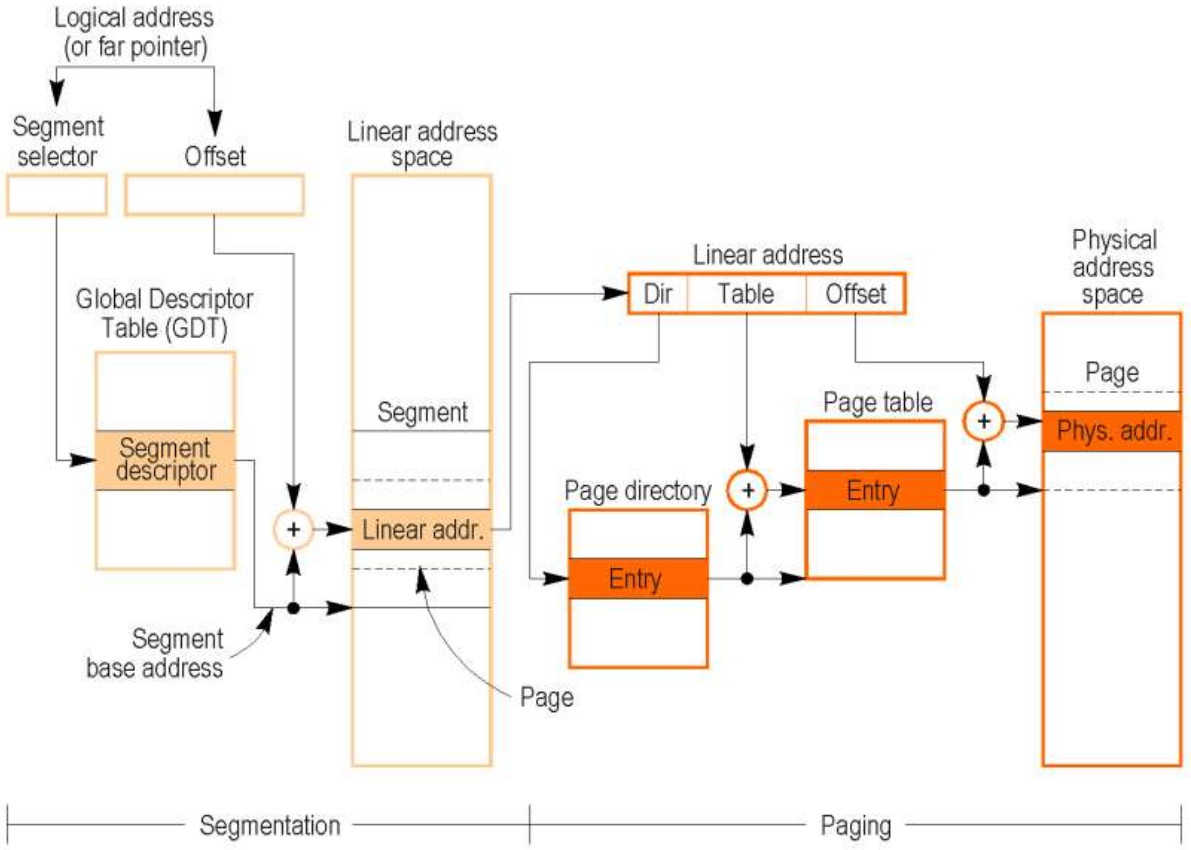
Paging (4MB Pages)



Paging (4kB pages)



Segmentation + Paging
Intel Pentium



Segmentation with Paging

- Segmentation
  - Is very similar to contiguous memory allocation
  - Causes external fragmentation

- Segmentation Example
  - Assuming
    - The segment size is 4G bytes, and thus the segment offset is 32 bits
    - a page is 4K bytes, and thus the page offset requires 12 bits.
  - Logical address = <segment#(13bits), segment_offset(32 bits)>
  - If segment# > STLR, cause a trap.
  - If offset > [STBR + segment#]'s limit, cause a segmentation fault.
  - Liner address = <[STBR + segment#]'s base | segment_offset>
  - Decompose liner address into <p1(10 bits), p2(10 bits), page_offset(12 bits)>
  - The first 10 bits are used in the outer page table to page the inner page table
  - The second 10 bits are used in the inner page table to page the final page

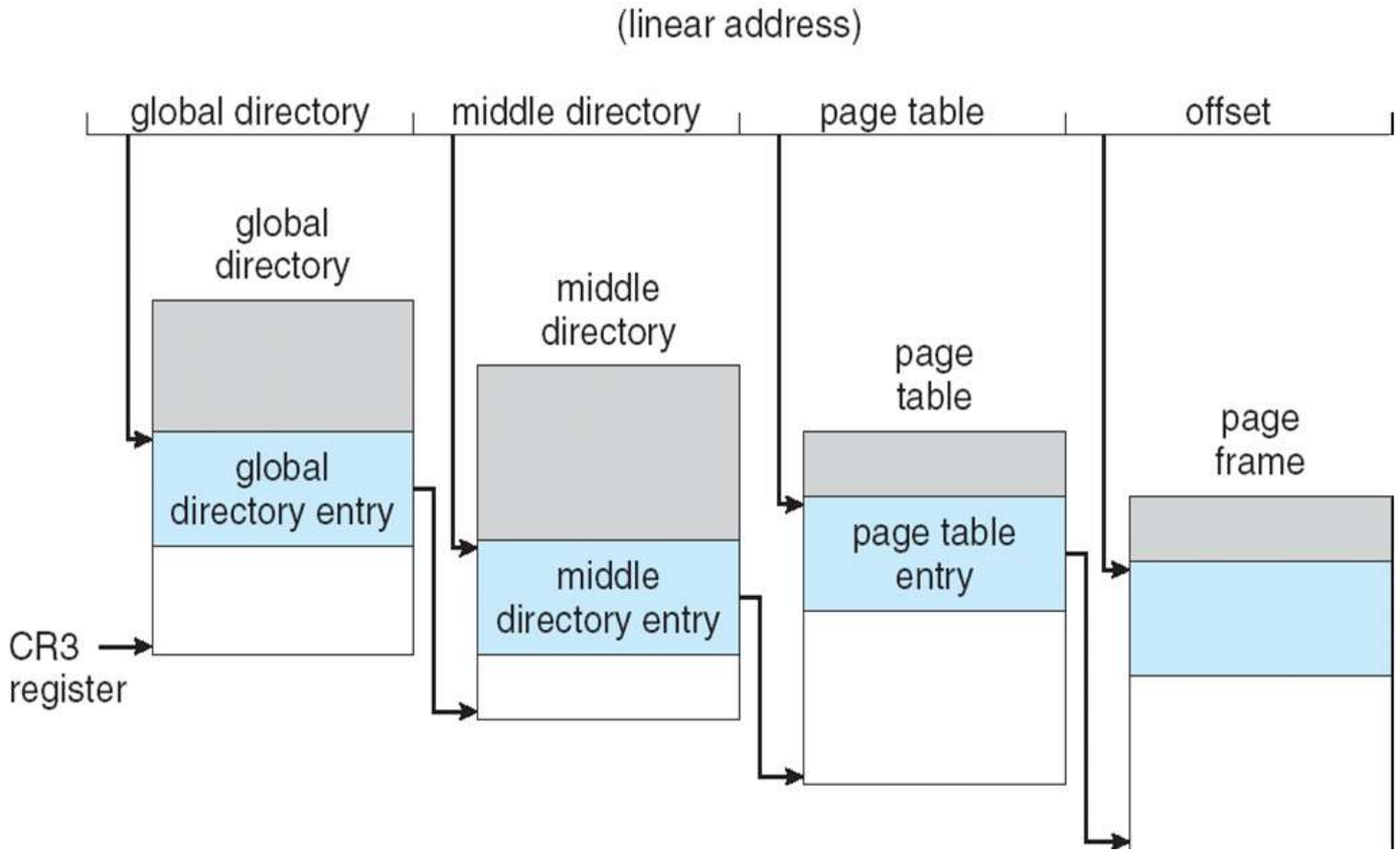
```
Physical address = <[PTBR + p2]'s frame# << 12 | page_offset>
```

Linear Address in Linux

Broken into four parts:

global directory	middle directory	page table	offset

Three-level Paging in Linux



Program 4

- Goal: Write a Disk cache to make disk operations more efficient
- How to start?
  - What is a cache?
  - Why do we need it? Think about memory hierarchy