Main Memory

Chapter 9

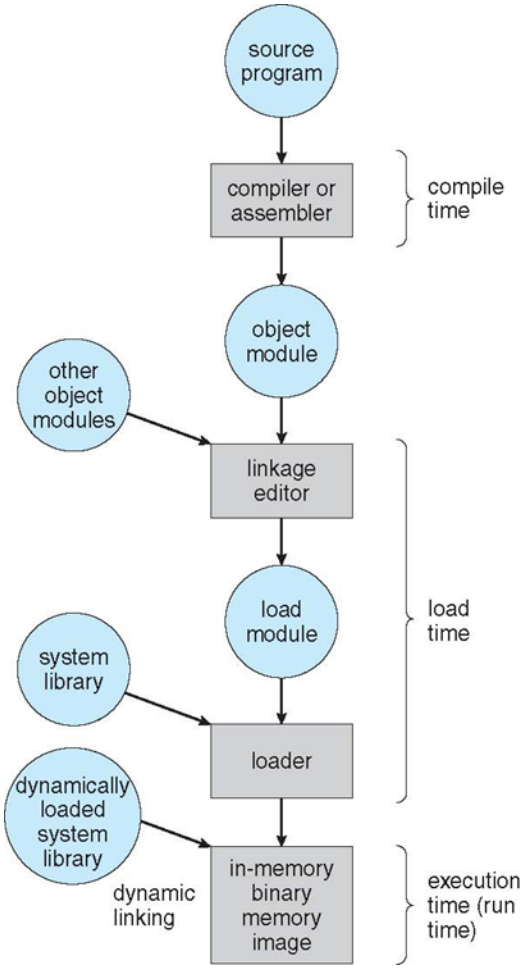
Background

- 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
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Address Binding

- Programs must be brought into memory and placed within a process for it to be run.
- The process resides in physical memory defined by its address
  - Source code addresses usually symbolic
  - Address binding of instructions and data to memory addresses can happen at three different stages
    - Compile time
    - Load time
    - Execution (run) time

Multi-step Processing of a User Program



Which physical address binding mode is most commonly used in modern computers?

A. Compile time
B. Load time
C. Run time

PollEv.com/xiaoyuzhang680

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**: compiler must generate **relocatable code** if memory location is not known at compile time.
  - 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
  - **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)

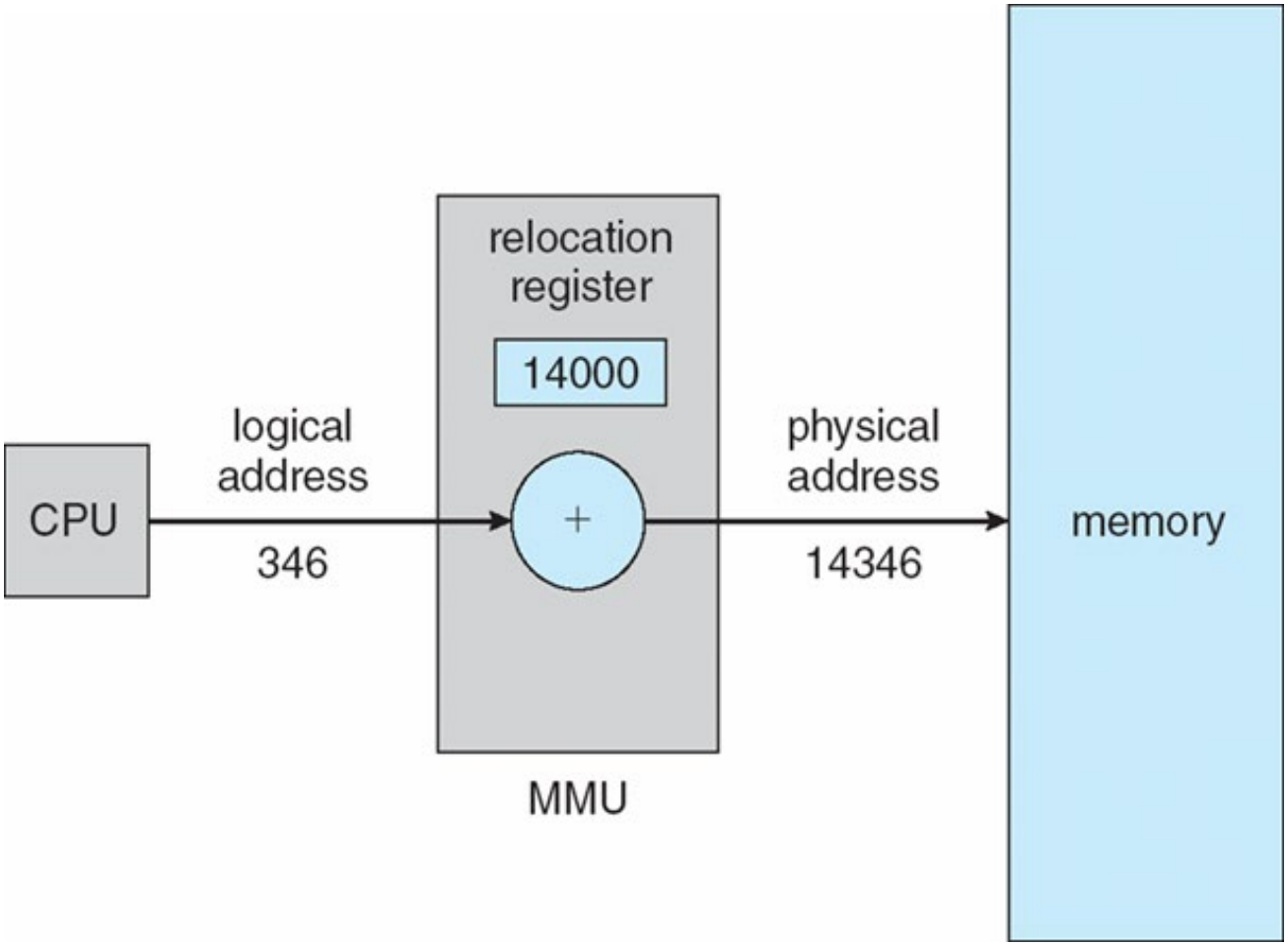
Logical vs. Physical Address Space

- 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

Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- Many methods possible, covered in the rest of this chapter
- To start, consider simple scheme where the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register

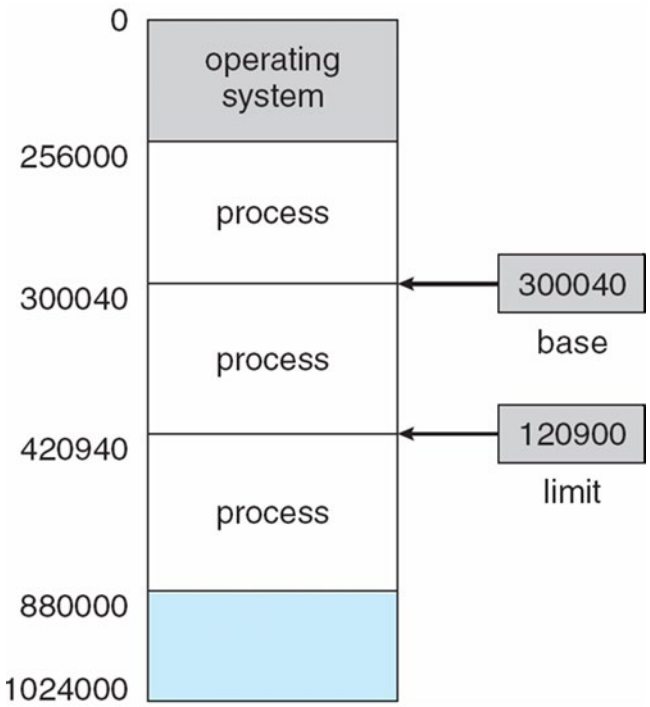


Contiguous Allocation

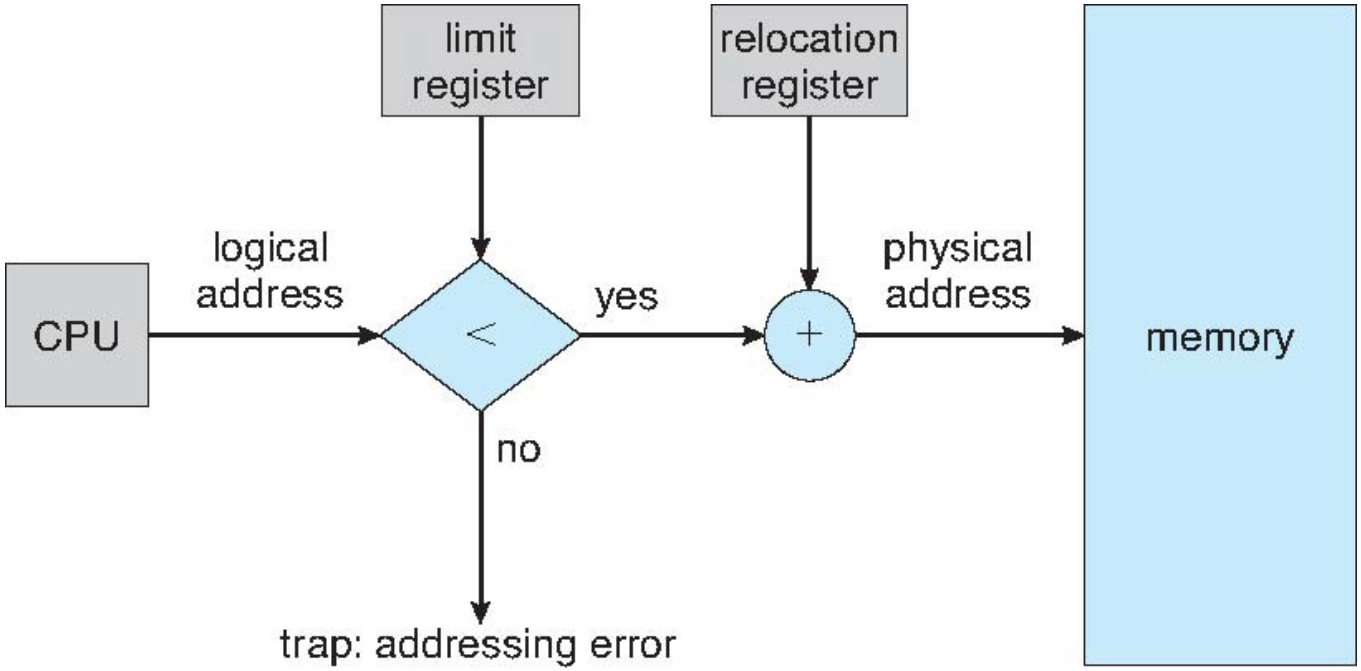
- Contiguous allocation is one early method of main memory management
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory
  - Each process contained in single contiguous section of memory
- 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*

Base and Limit Registers

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



HW address protection with base and limit registers



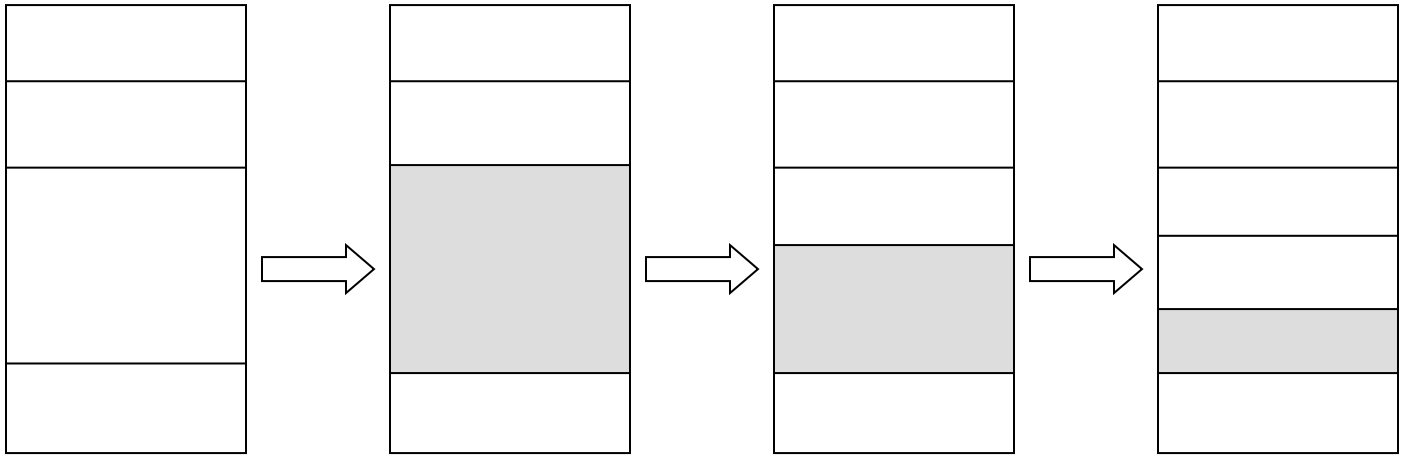
Given what we currently know about memory, what must we do during a context switch?

A. Allocate memory to the switching process
B. Load the base and limit registers
C. Convert logical to physical memory addresses
D. None of the above

PollEv.com/xiaoyuzhang680

Contiguous Allocation (Cont.)

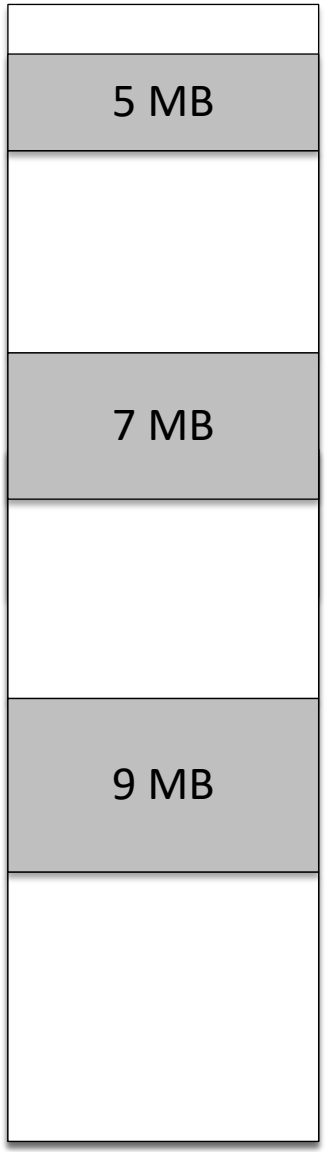
- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (hole)



Dynamic Storage-Allocation Problem

- When a process arrives, search for a hole big enough for it. If none available, the process must wait.
- When a process terminates, memory is freed, creating a hole.
- This new hole may join with other contiguous holes to create a bigger hole.
- How to satisfy a request of size *n* from a list of free holes?
  - **First-fit**: Allocate the *first* hole that is big enough
  - **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the *largest* hole; must also search entire list
    - Produces the largest leftover hole
  - First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Where would worst-fit place this memory chunk?



External Fragmentation

- As memory is allocated to processes and freed as processes terminate, the free memory space is broken into small pieces.
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous.
- Statistical analysis reveals that with the first fit memory allocation rule, given N allocated blocks, another 0.5N blocks may be lost due to fragmentation (50 percent rule).
- This means 1/3 of memory may be unusable!

Compaction

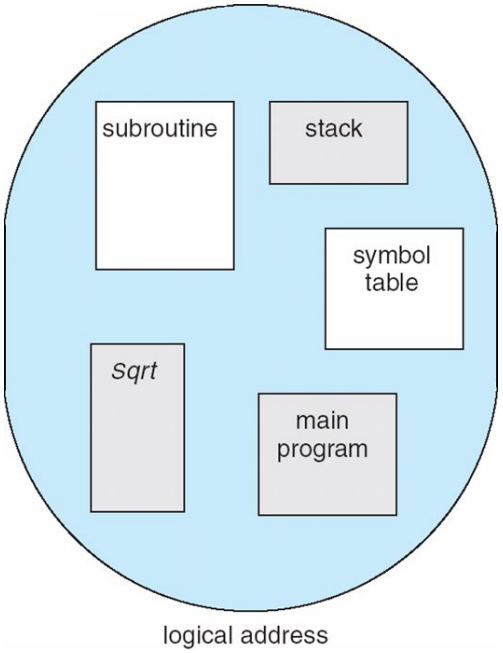
- One can reduce external fragmentation by **compaction.**
  - Shuffle memory contents to place all free memory together in one large block.
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time.
  - I/O problem
    - Latch job in memory while it is involved in I/O.
    - Do I/O only into OS buffers.

User's view of memory space

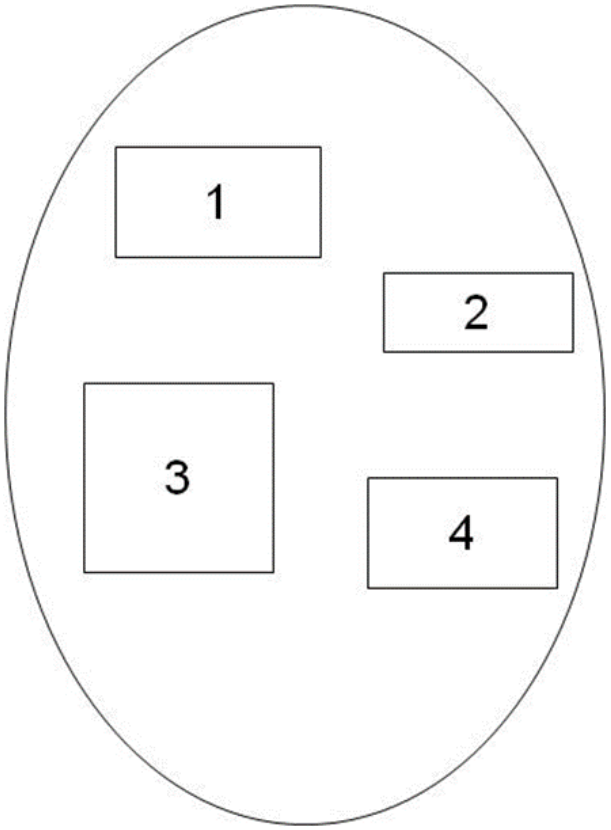
- Physical memory is a linear array of memory addresses.
- Users don't usually think of memory in terms of a linear array.
- Users generally think in terms of a collection of items (segments) that are stored in memory with no particular order among segments.
- Example: A C++ program may consist of:
  - A main function
  - A set of functions or classes
  - data structures
  - etc.
- We refer to each of these by name.
- We don't care where each piece is stored relative to another.

Segmentation

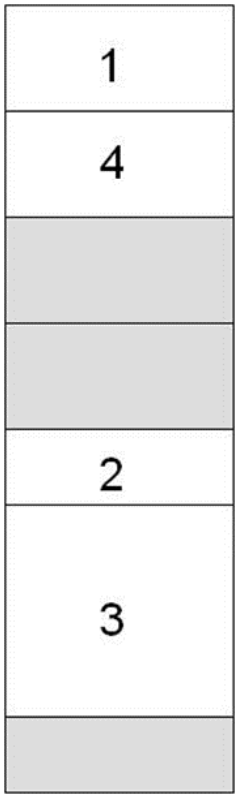
- Memory-management scheme that supports user view of memory
- A program is a collection of segments. A segment is a logical unit such as:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack,
  - symbol table, arrays



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

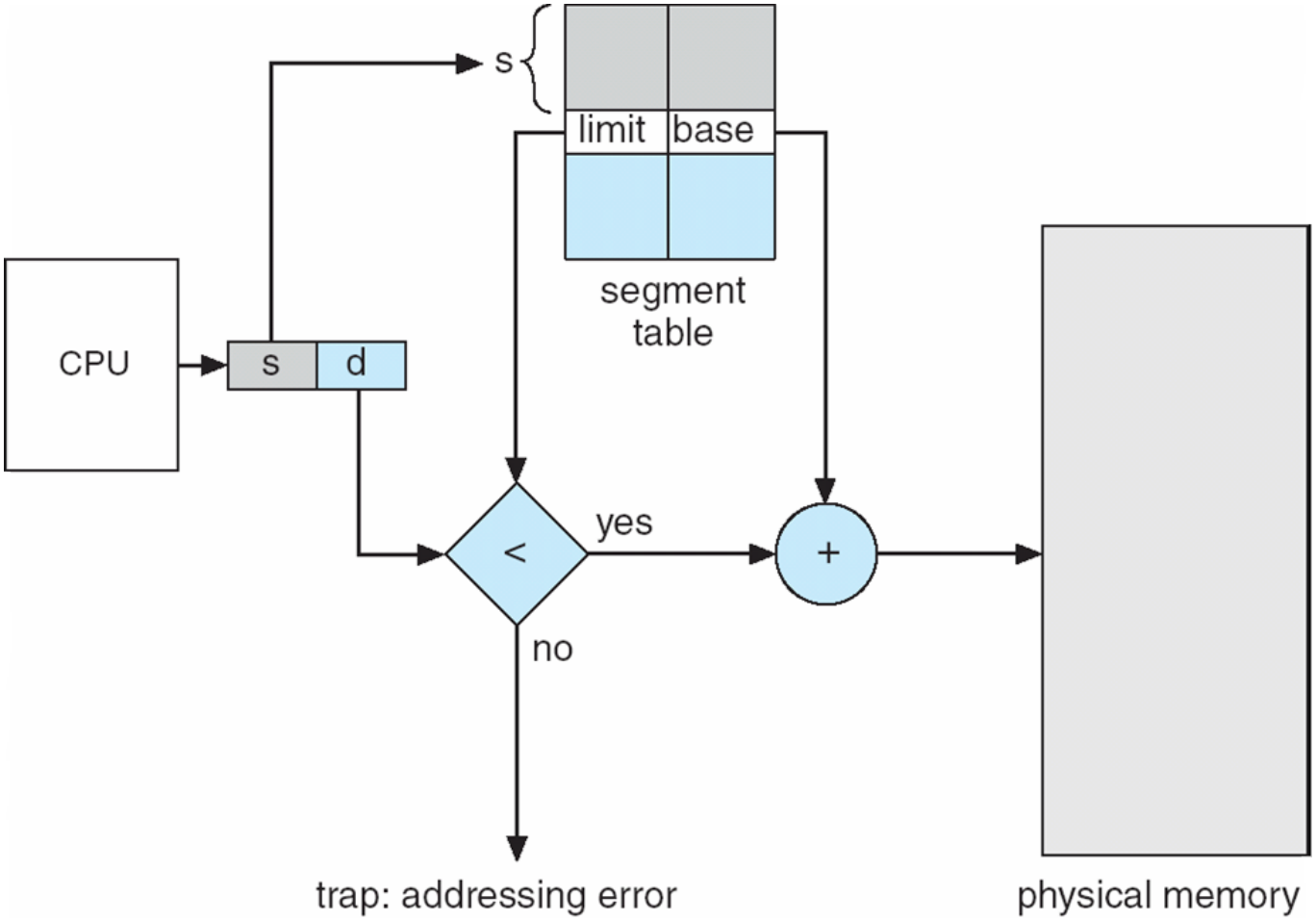
- Segments are numbered and referred to by a segment number, thus a logical address consists of a tuple:

  <segment-number, offset>,

- **Segment table** – maps logical addresses to 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 Hardware

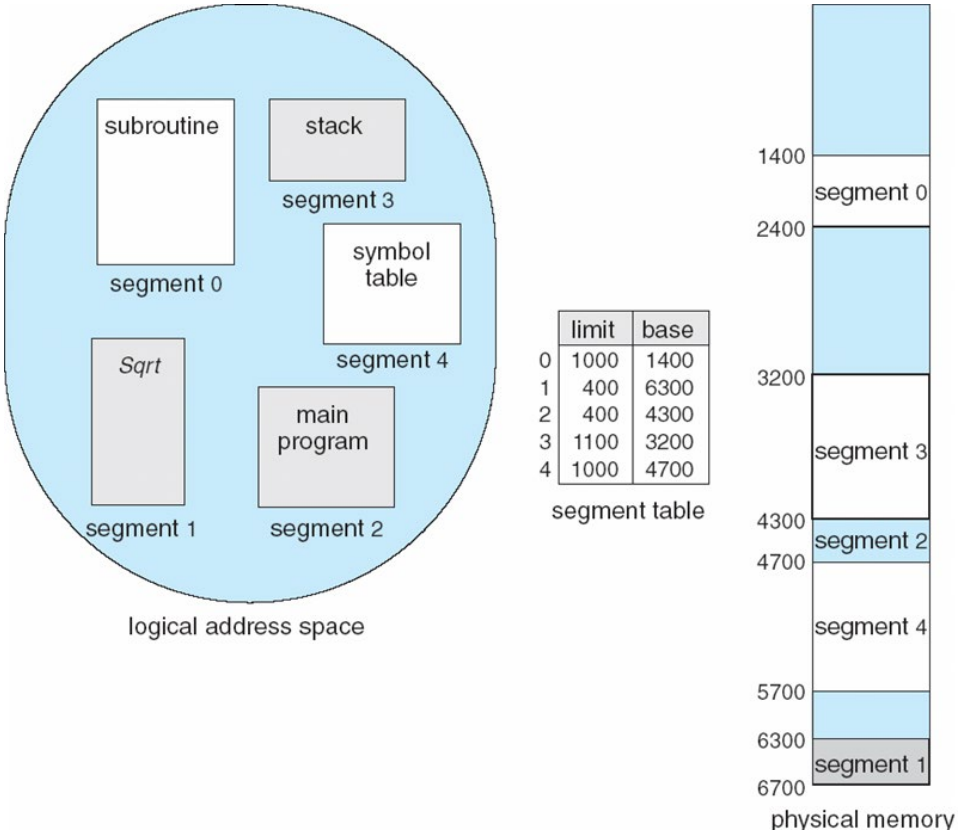


Example

- Suppose a program is divided into 5 segments:
  - Segment 0: Subroutine
  - Segment 1: sqrt function
  - Segment 2: main program
  - Segment 3: stack
  - Segment 4: symbol table
- The segment table is as follows:

Segment	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Example of Segmentation



Question: Where does segment 1, offset 43 map in physical memory? Where does segment 2, offset 567 map?

Segmentation Architecture (Cont.)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 $\Rightarrow$ illegal segment / segment not in physical memory
    - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Segment Table				
segment	Valid	Base	limit	Protect
1	0	120	600	r
2	1	8000	1000	rw
3	1	9000	512	rx

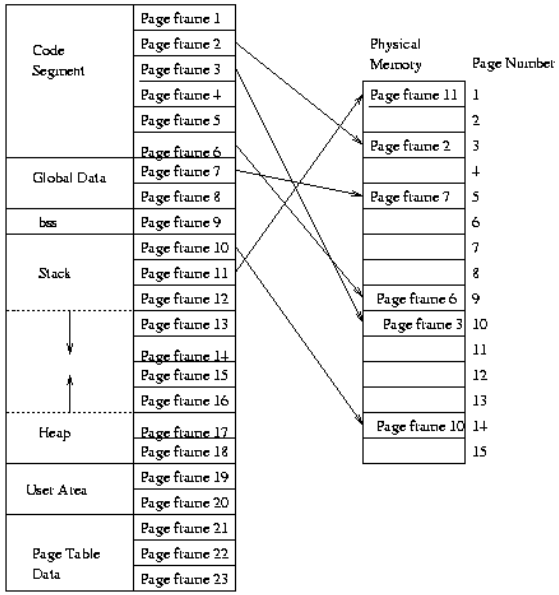
Fragmentation

- Segments are of variable length.
- Memory allocation is a dynamic process that uses a best fit or first fit algorithm.
- Is there external fragmentation in a segmentation system?

Paging and Internal Fragmentation

- Another method of allocating memory is to break memory into fixed sized blocks.
- Memory is allocated in units of block sizes - paging.
- The memory allocated may be slightly larger than the size needed by the process. This is **Internal Fragmentation.**
- With **Internal Fragmentation,** the size difference between the process size and the allocated memory is memory internal to a partition, but not being used.
- Example: Block size 1024 bytes. Process size: 4048 bytes.

  What is the internal fragmentation size?

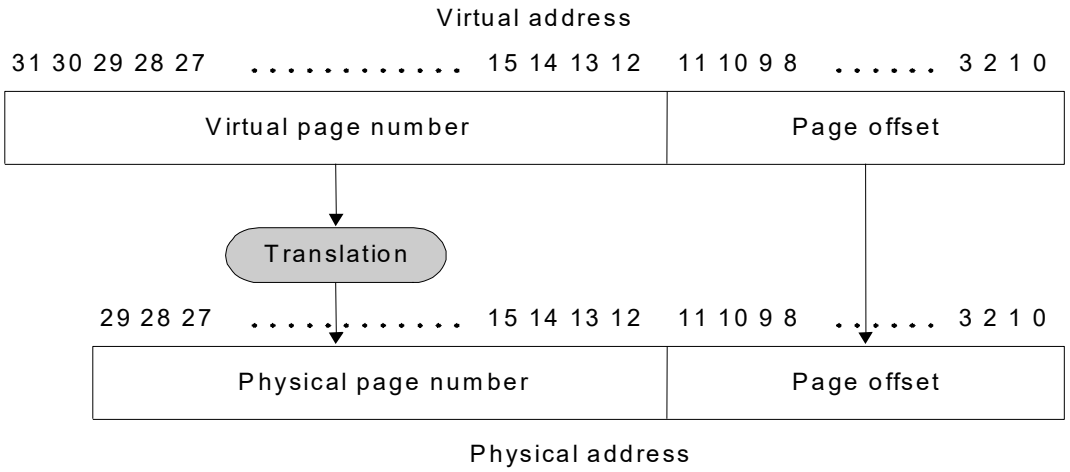


Paging

- Paging allows the physical address space of a process to be non-contiguous. process is allocated physical memory whenever the latter is available. Paging is commonly used in most operating systems.
  - 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
- Set up a page table to translate logical to physical addresses

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



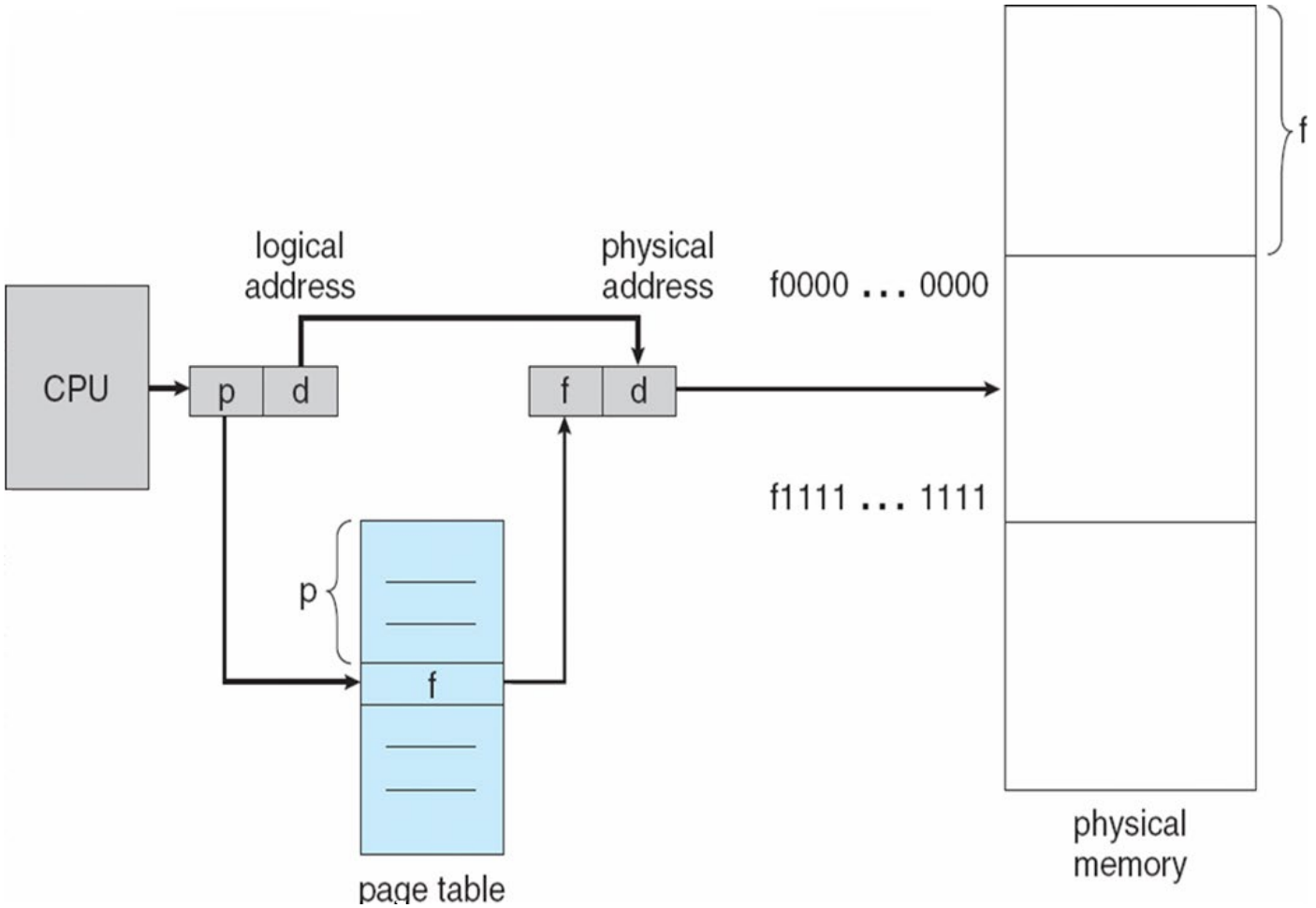
Example

Given the logical address 0xF4BCAEF9 (in hexadecimal) with a page size of 4KB (2^12 bytes), what are the page number and offset respectively?

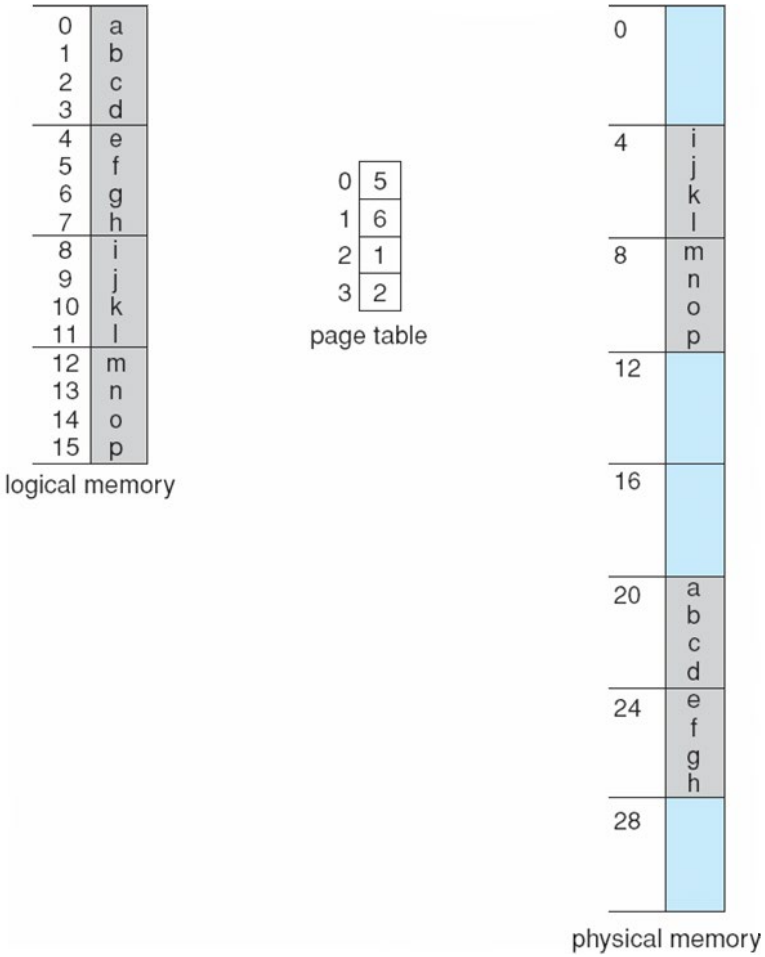
A. Page number = 0xF4BCA, page offset = 0xEF9
B. Page number = 0xF4BC, page offset = 0xAEF9
C. Page number = 0xF4B, page offset = 0xBCAEF9
D. Page number = 0xF4BCAE, page offset = 0xF9
E. none of the above.

PollEv.com/xiaoyuzhang680

Paging Hardware



Paging Example



- Page size = 4 bytes
- Physical memory = 32 bytes
- Number of frames = ?
- Where do the following map?
- Logical 0: page 0, offset 0:
- Logical 3: page 0, offset 3:
- Logical 4:
- Logical 13:

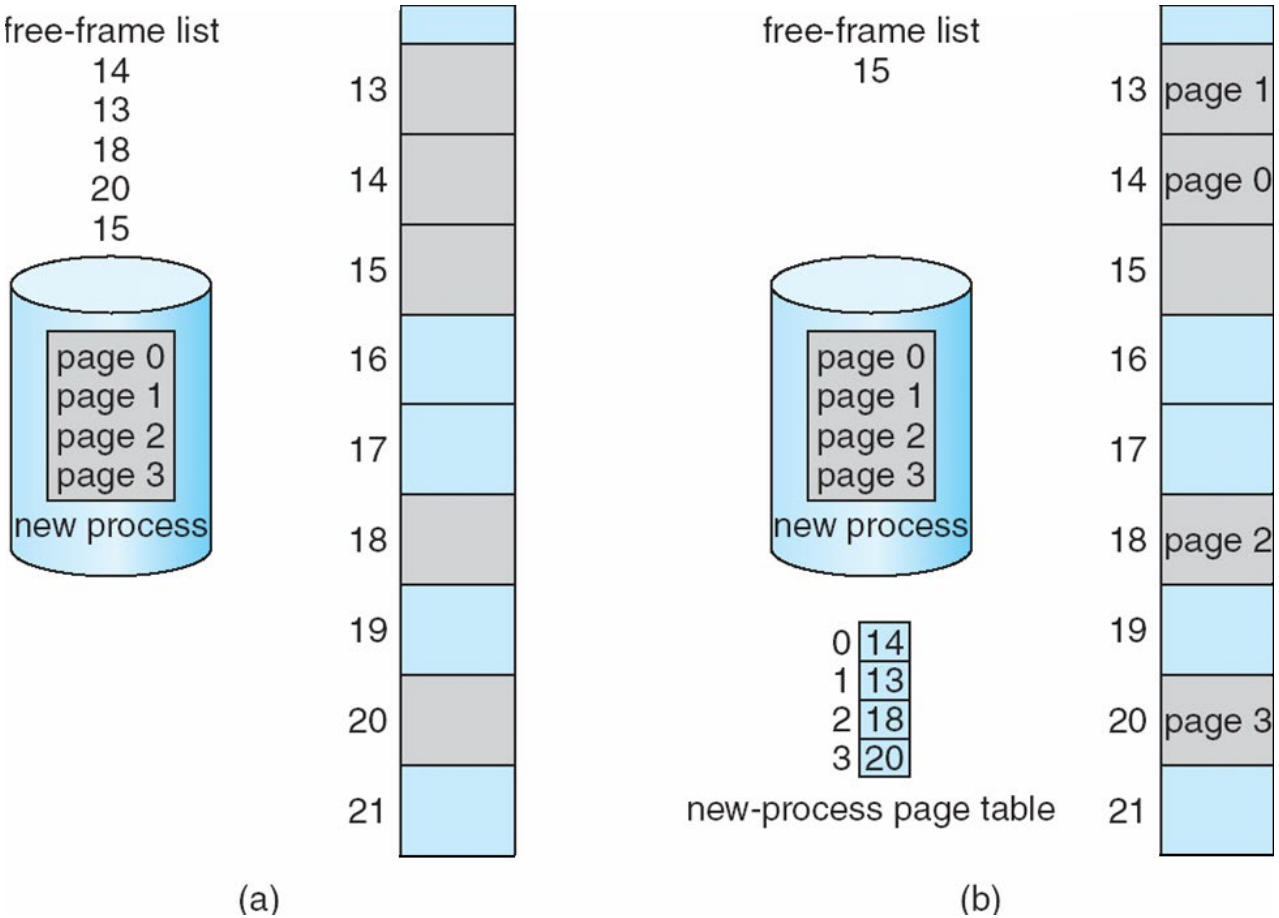
Paging and Fragmentation

- Don't have external fragmentation with paging.
  - Any free frame can be allocated to a process that needs it.
- May have internal fragmentation with paging.
  - Frames are allocated as units. Last frame may not be completely full.
  - Not as severe as external fragmentation.

Allocation of Frames

- The O.S. keeps track of which frames are allocated and which are free in a **frame table.**
- If a process requests *n* frames, there must be *n* frames available to satisfy the request. If so, they are allocated to the process.
- As each frame is allocated to each page, the frame number is put in the page table for that process.
- Note: The user views memory as contiguous space. The program is actually scattered throughout physical memory.

Free Frames


free-frame list
14
13
18
20
15
page 0
page 1
page 2
page 3
new process
13
14
15
16
17
18
19
20
21
(a)
free-frame list
15
page 0
page 1
page 2
page 3
new process
0 14
1 13
2 18
3 20
new-process page table
13 page 1
14 page 0
15
16
17
18 page 2
19
20 page 3
21
(b)

Implementation of Page Table

- Simplest implementation: Page table stored in dedicated registers in the CPU.
- The registers use high speed logic.
- The CPU dispatcher re-loads these registers when switching processes. (Each process has its own page table).
- Example: DEC PDP-11
  - Address size = 16 bitsNumber of bytes in memory = ?
  - Page size = 8 KB (8192) = 2^{13} Number of pages = ?
  - # of entries in page table = number of pages
- This method is only useful when the page table is small (< 256 entries).
- Most contemporary computers have much larger page tables (> 10^6 entries) so this method will not work well.

Storing of Addresses

- Suppose the size of logical address space = 2^m
- Suppose the page size = 2^n
- For example m = 32, n = 12
  - Allocate m bits to specify logical addresses.
  - First m - n bits of the address specify the page number.
  - The n lower order bits indicate the offset.
- What is the total number of pages?

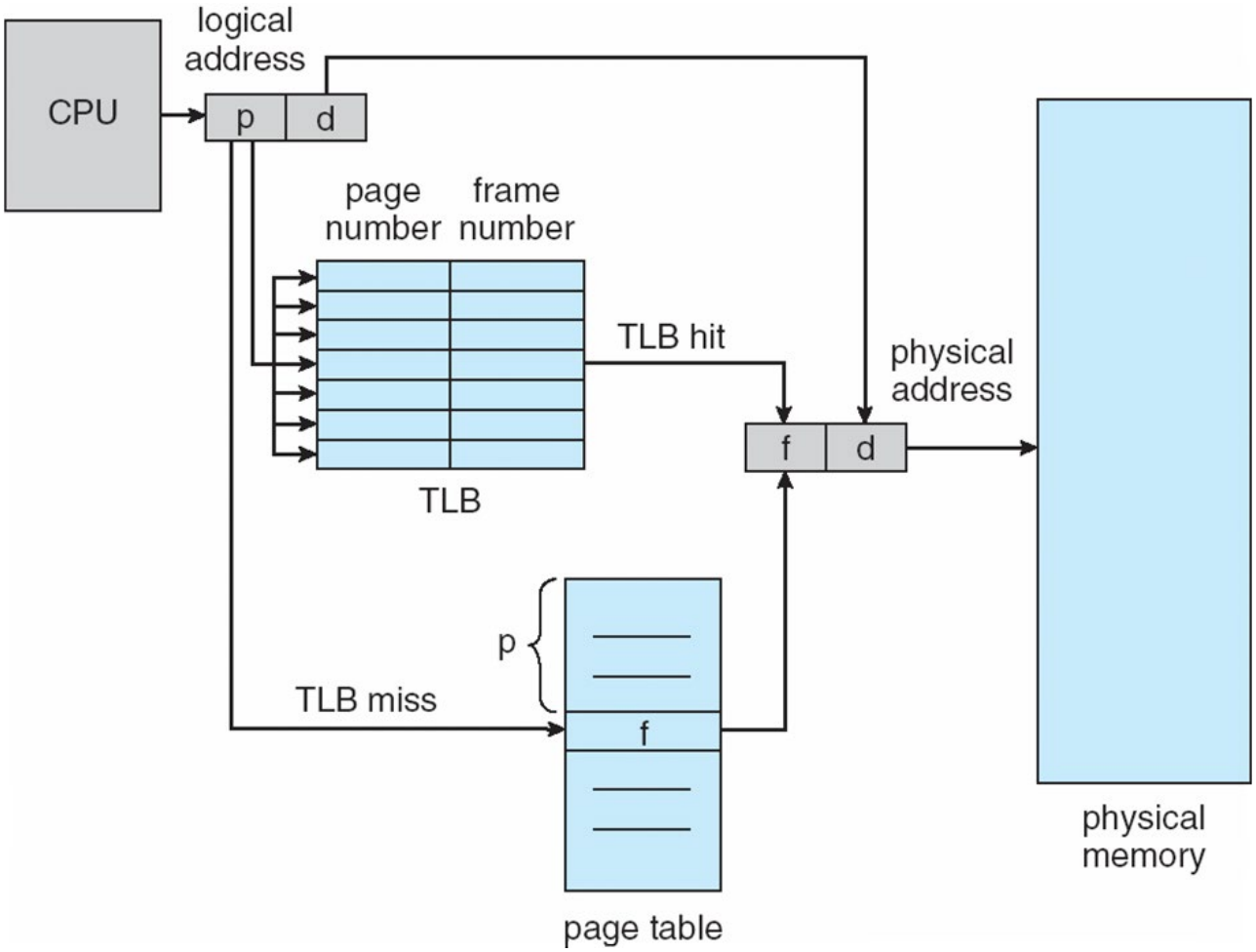
Storing Page Table in Main Memory

- For large page tables, the 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.
  - When changing processes, only need to change 1 register to change page tables. This reduces the context switch time.
- Drawback: In this scheme every data/instruction access requires two memory accesses.
  - One for the page table and one for the data/instruction.

Translation Look-Aside Buffer (TLB)

- 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)*
  - generally small (64 - 1024 entries).
- In each memory access, the TLB is searched first to locate the page number.
  - If the page is found (a hit) the associated frame is used to access the data in memory.
  - If the page is not found (a miss), the page number is looked up in the page table in main memory. The page number and associated frame is added to the TLB.
- Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Paging Hardware With TLB



Effective Access Time (EAT)

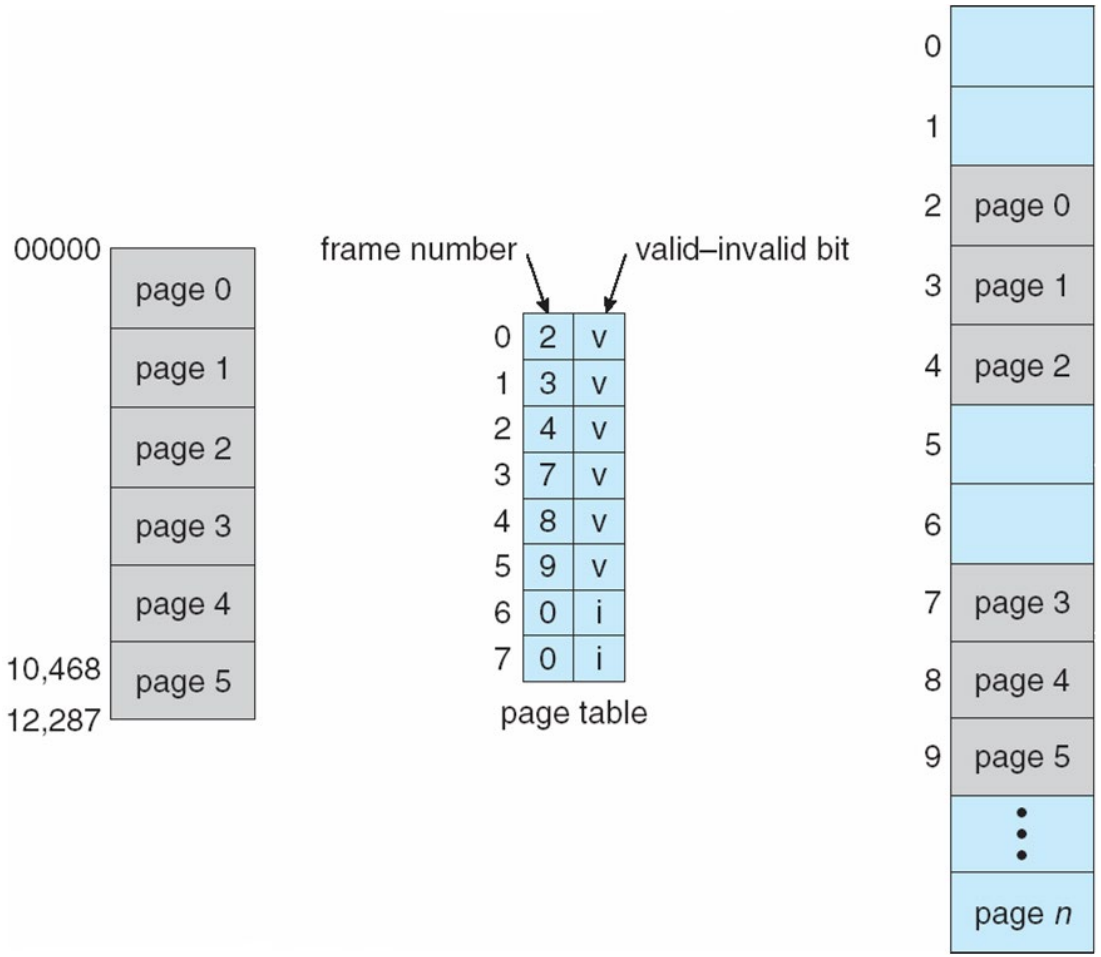
- **Effective access time (EAT)** is the average time needed to access memory.
- **Hit Ratio:** The percentage of times that a particular page number is found in the TLB.
- Effective access time can be calculated based on:
  - The time it takes to access main memory
  - The time it takes to access the TLB
  - The hit ratio for the TLB
- Hit ratio = α, Main Memory access time = m, TLB access = ε
- EAT = $(m + \varepsilon)\ \alpha + (2m + \varepsilon)\ (1 - \alpha)$

  $= 2m - m\alpha + \varepsilon$
- Example: m = 10 ns, $\alpha = 0.8$ or 0.99,

Memory Protection

- Memory protection implemented by associating protection bit with each frame in the page table.
  - Bits signal if a frame is read only, read-write, execute only or a combination.
- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical and physical address space, and is thus a legal page.
  - "invalid" indicates that the page is not in the process' logical address space or that page is not in the physical memory.
- Example:
  - System has 14 bit address space (0 - 16383)
  - Program uses addresses 0 - 10468
  - Page size = 2 KB (2048 = 2^{11})
  - Total # of pages = ?
  - 6 pages needed by program (5 pages = 5*2048 = 10240 Bytes)
  - Pages 0 - 5 have valid/invalid bit set to valid
  - Other pages have bit set to invalid.

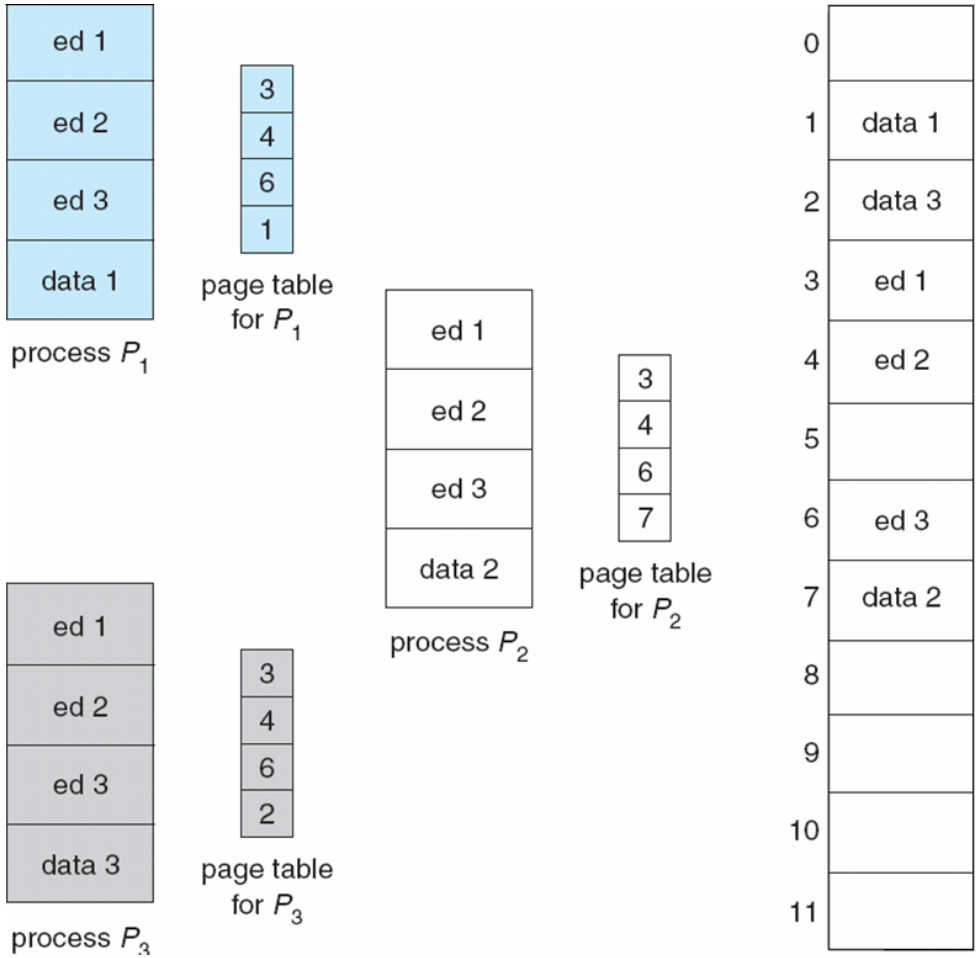
Valid (v) or Invalid (i) Bit In A Page Table



Shared Pages

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



Dynamic Loading

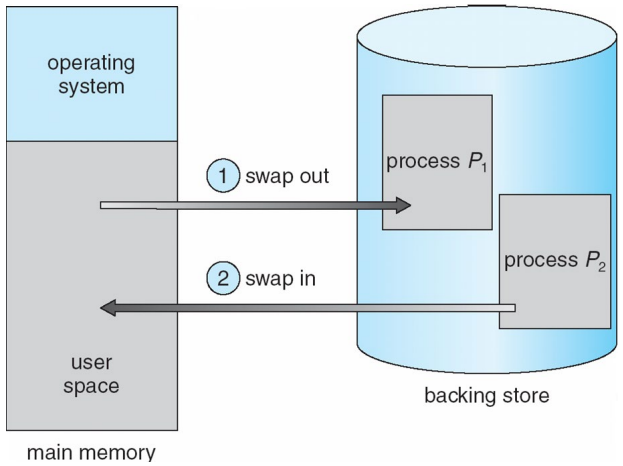
- If the entire program and data must be placed in physical memory for execution, then the size of the process is limited by the physical memory size.
- With **dynamic loading**, Routine is not loaded until it is called
  - Better memory-space utilization; unused routine is never loaded
  - Useful when large amounts of code are needed to handle infrequently occurring cases
  - No special support from the operating system is required. Implemented through program design.

Dynamic Linking

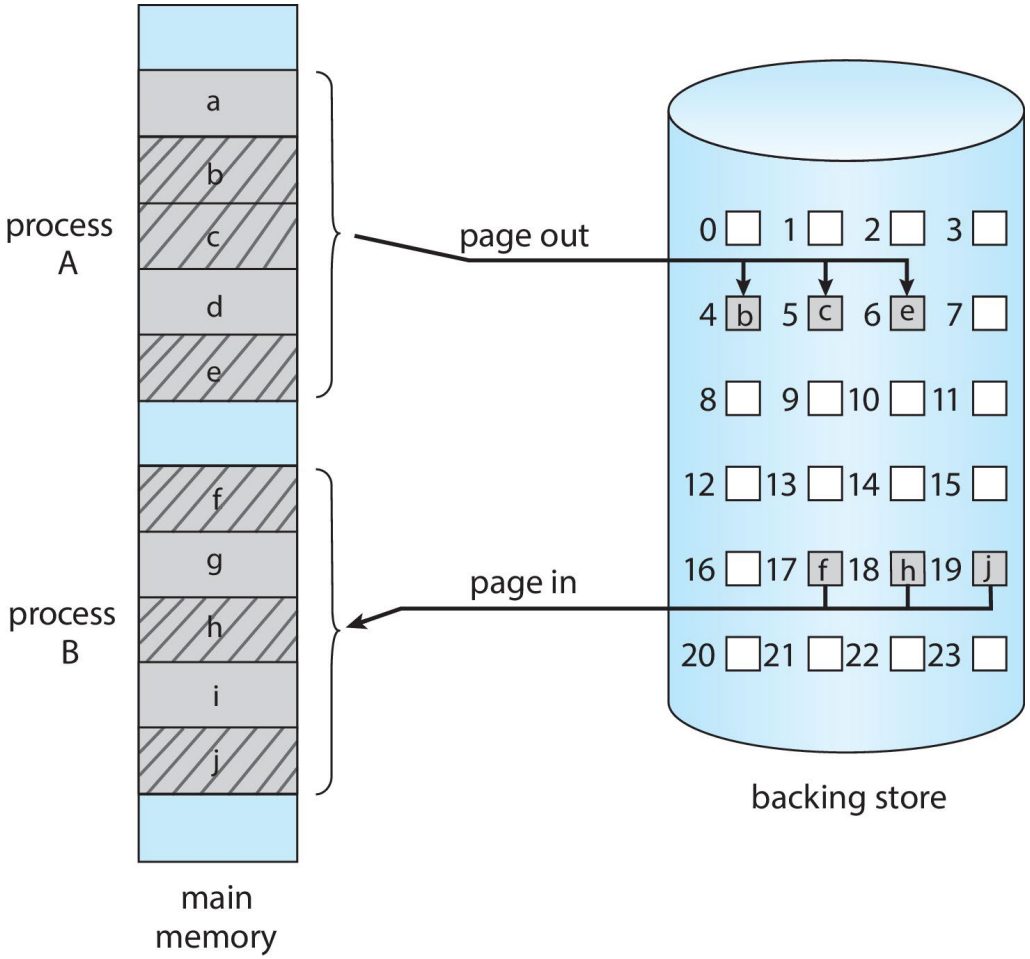
- In **static linking** the system libraries are combined by the loader into the binary program image.
- This can be wasteful of space if many programs use the same libraries.
- With **dynamic linking,** Linking postponed until execution time
  - Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
  - Stub replaces itself with the address of the routine, and executes the routine
  - Operating system needed to check if routine is in processes' memory address
  - Dynamic linking libraries also known as **shared libraries** in Unix
- Dynamic linking requires help from the operating system. Operating system needed to check if routine is in another processes' memory address, and to allow multiple processes to share memory space.

Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - **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, which is directly proportional to the amount of memory swapped
- Does the swapped out process need to swap back in to same physical addresses?
- 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



Swapping with Paging



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
  - Both OSes support paging as discussed below

Problem with large page tables

- Modern computers have large logical address spaces: 2^{32} or 2^{64} bytes.
- This can make page tables excessively large.
- Example: 32 bit logical address space

  4 KB Page size

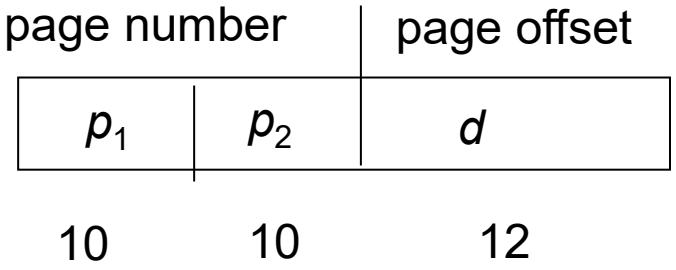
- How many entries in the page table?
- If each entry is 4 bytes, what is size of each table?
- With large page tables, it is good not to store them continuously in memory.

Structure of the Page Table

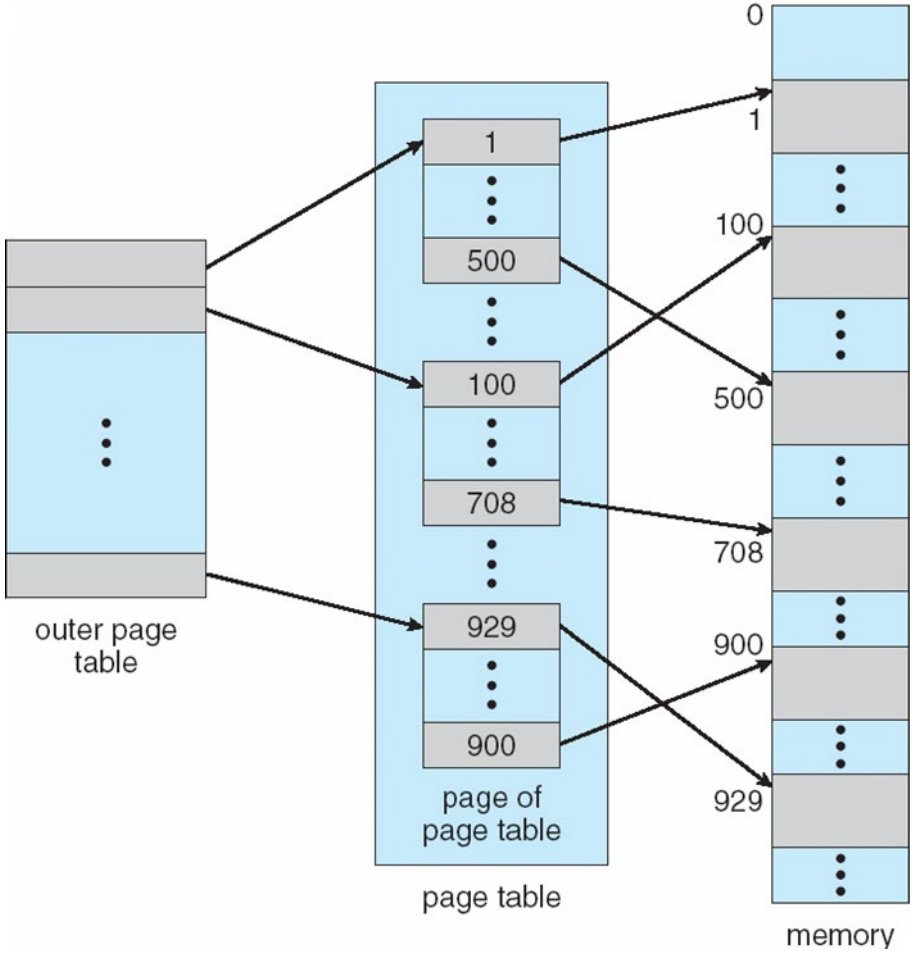
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Page Tables

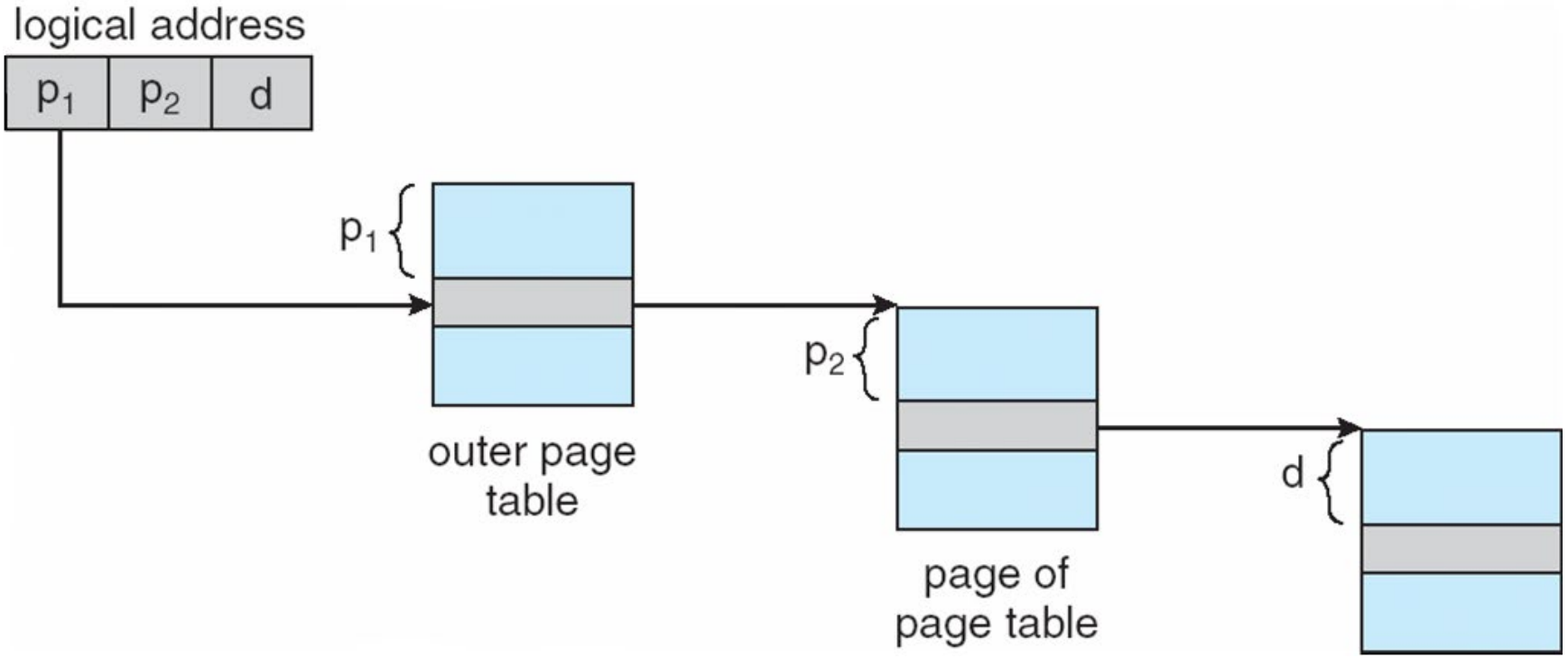
- In hierarchical page tables, the logical address space is broken up into multiple page tables.
- A two-level example: A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits.
  - a page offset consisting of 12 bits.
- We page the page table, e.g. the page number is further divided into:
  - $p_{1:}$ a 10-bit page number.
  - $p_{2:}$ 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.



Two-Level Page-Table Scheme

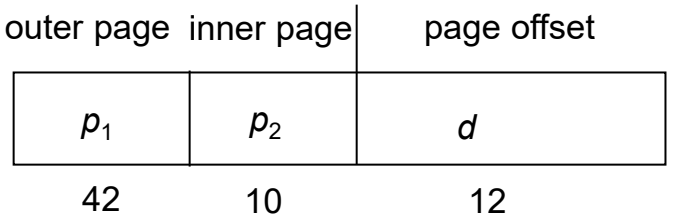


Address-Translation Scheme



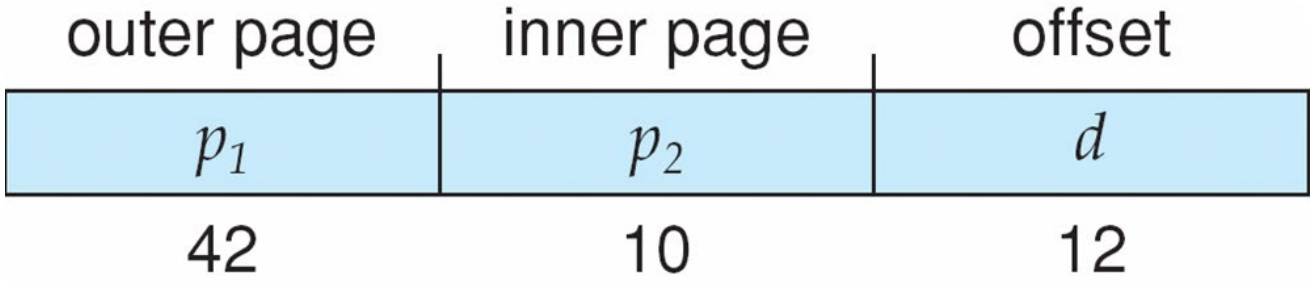
64-bit Logical Address Space

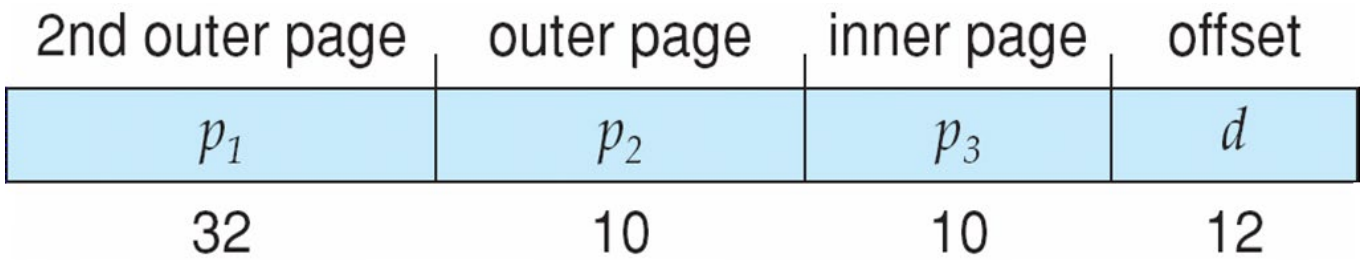
- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



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

Three-level Paging Scheme

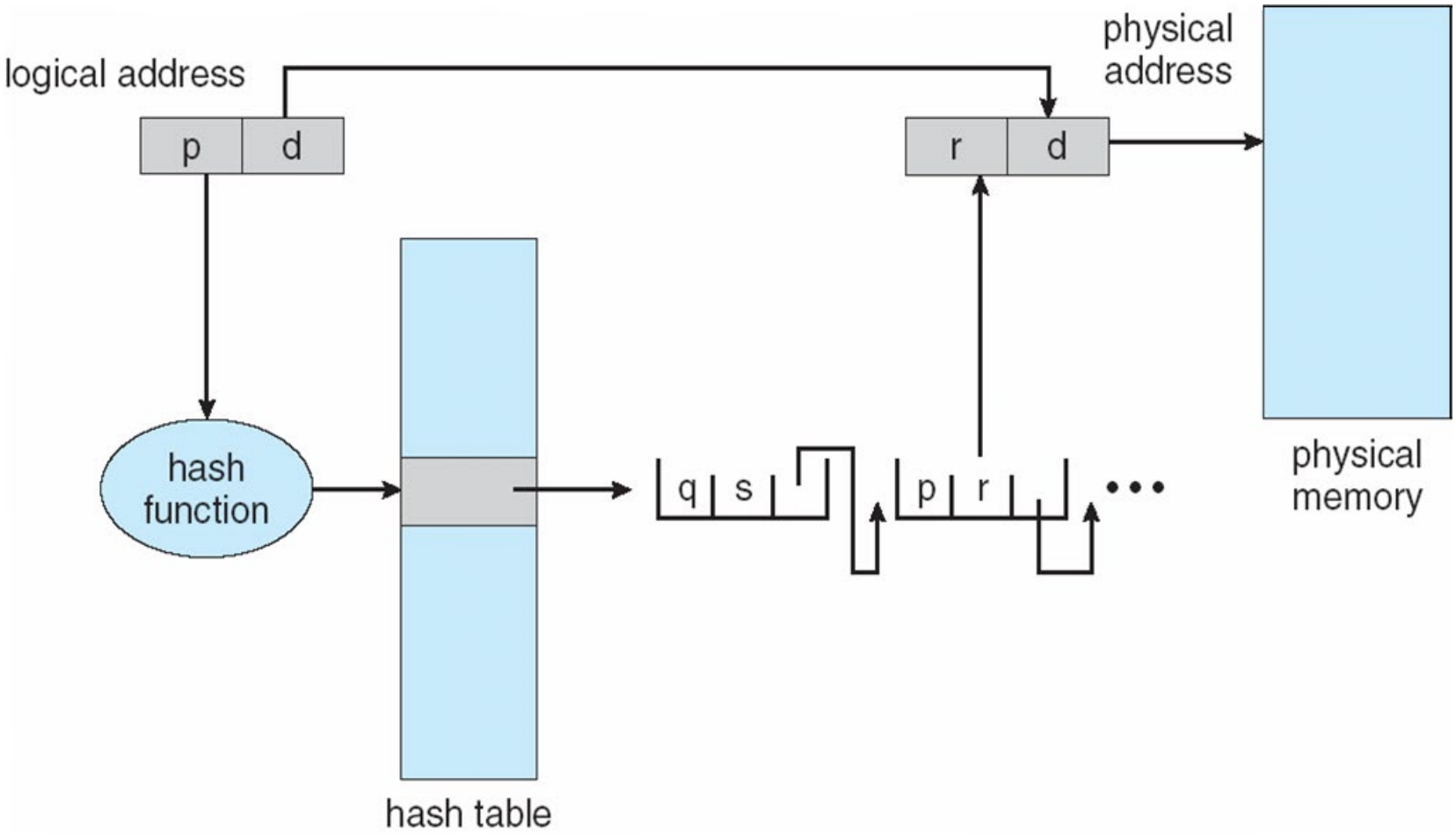




Hashed Page Tables

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

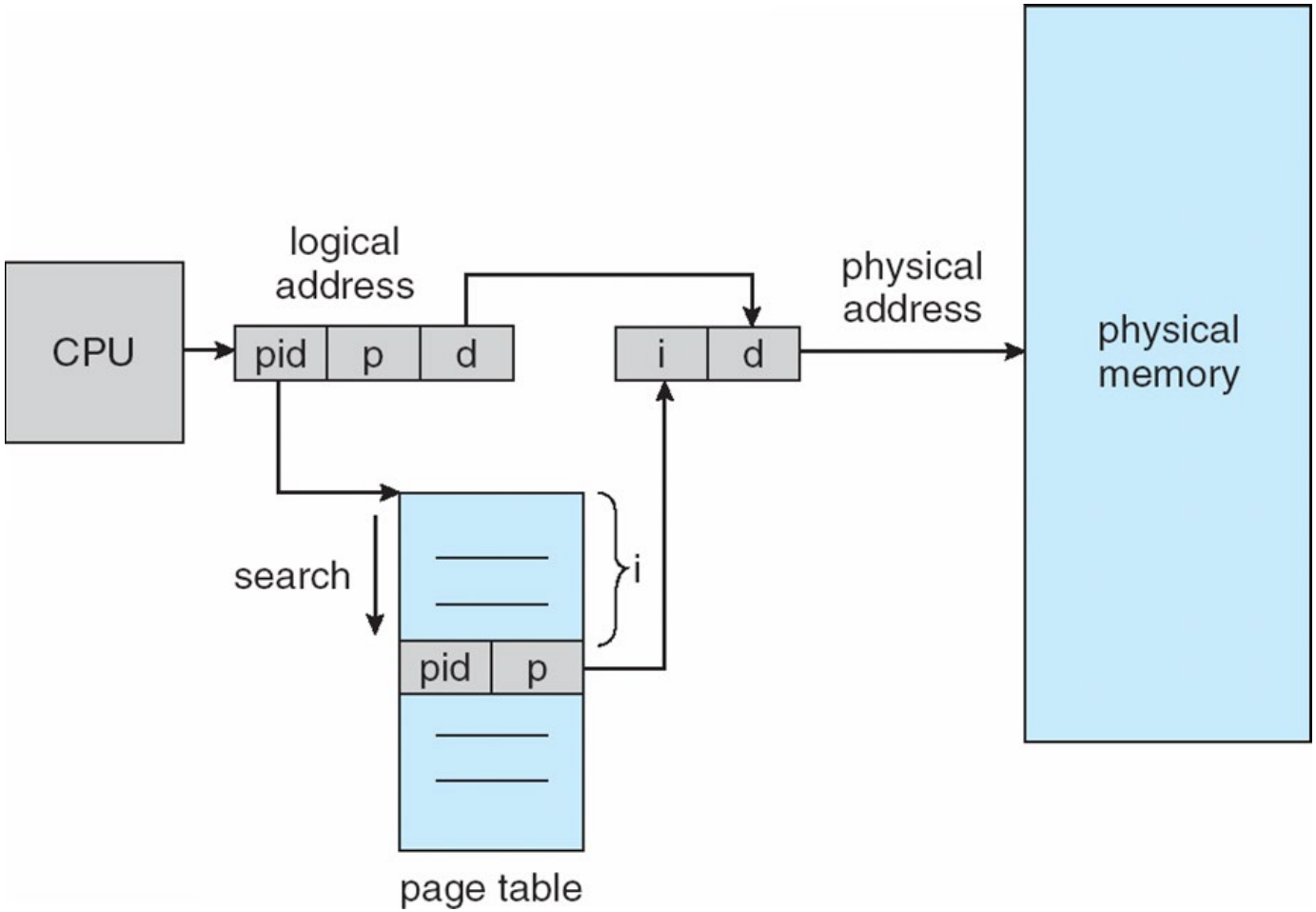
Hashed Page Table



Inverted Page Table

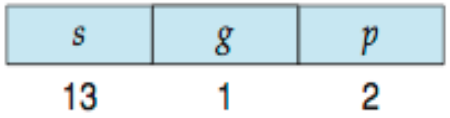
- One entry for each physical page (frame) of memory
- Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page
- Decreases memory needed to store each page table, but increases time needed to search the table when a 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 share pages between processes?

Inverted Page Table Architecture



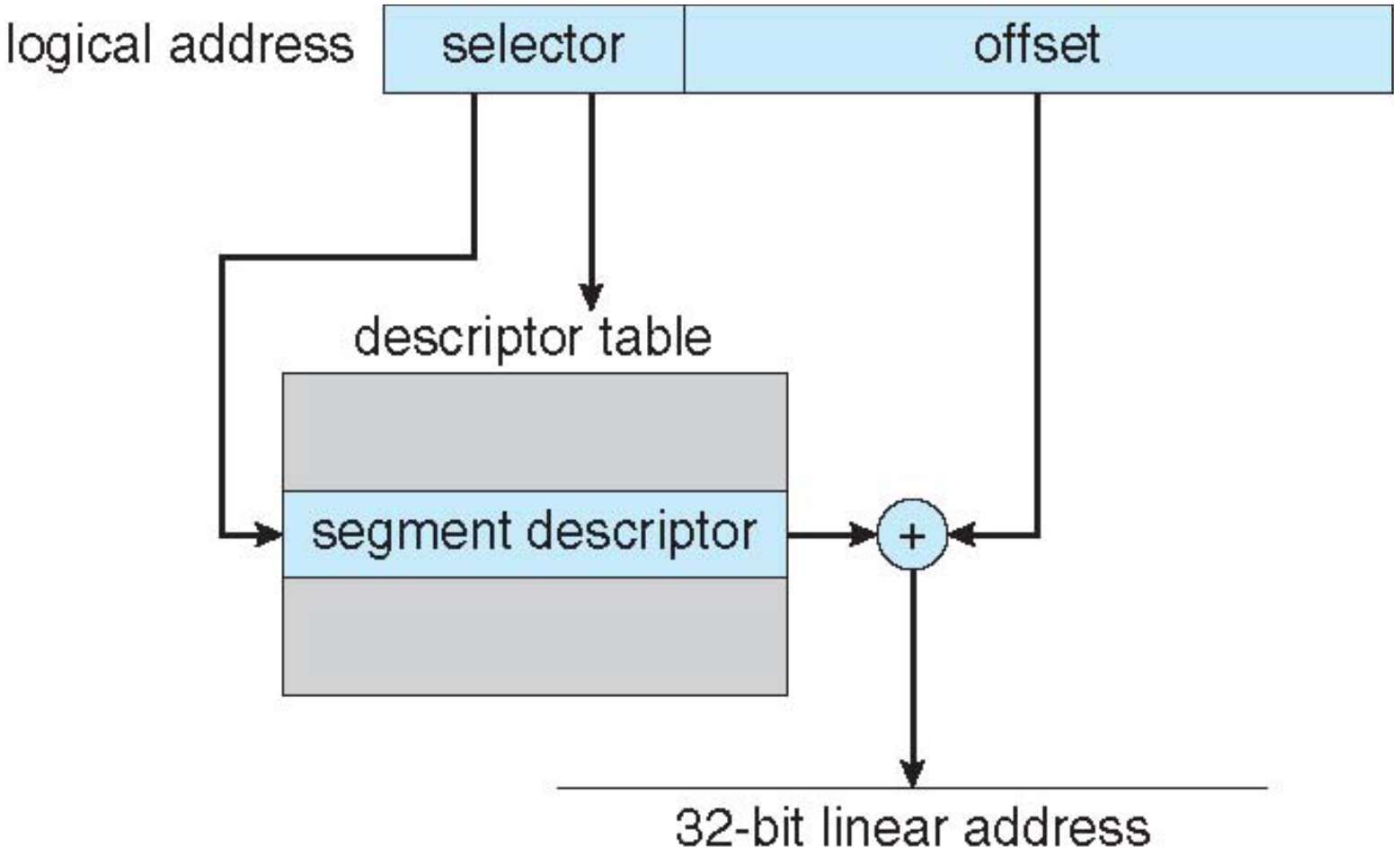
Example: The Intel IA-32 Architecture

- Supports both segmentation and segmentation with paging
  - Each segment can be 4 GB
  - Up to 16 K segments per process
  - Divided into two partitions
    - First partition of up to 8 K segments are private to process (kept in **local descriptor table** (**LDT**))
    - Second partition of up to 8K segments shared among all processes (kept in **global descriptor table** (**GDT**))
- CPU generates logical address as a pair (selector, offset)
  - Selector given to segmentation unit
    - Which produces linear addresses

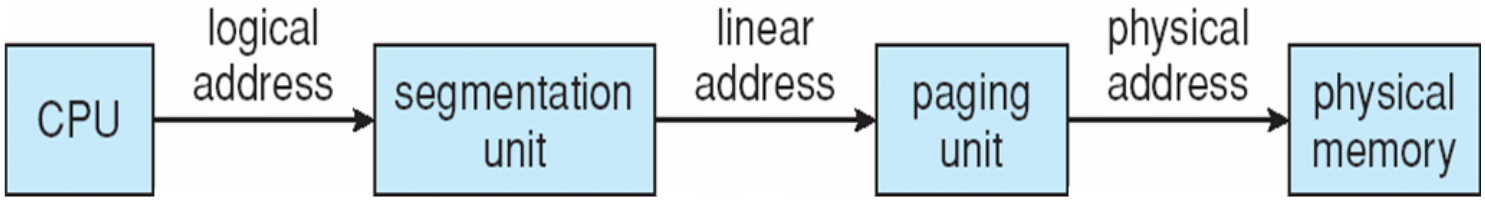


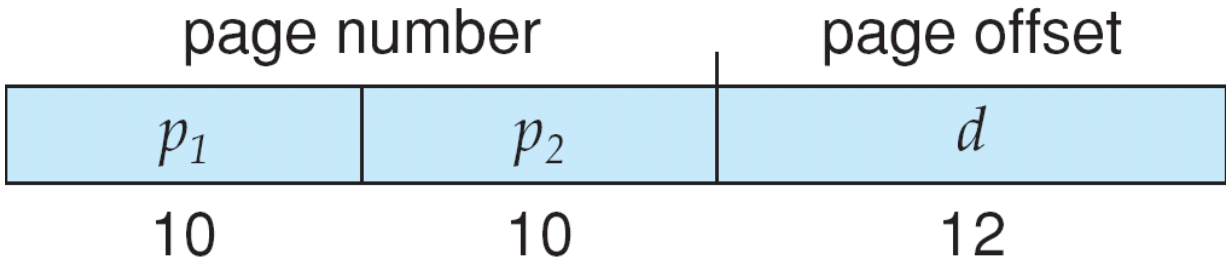
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

Intel IA-32 Segmentation

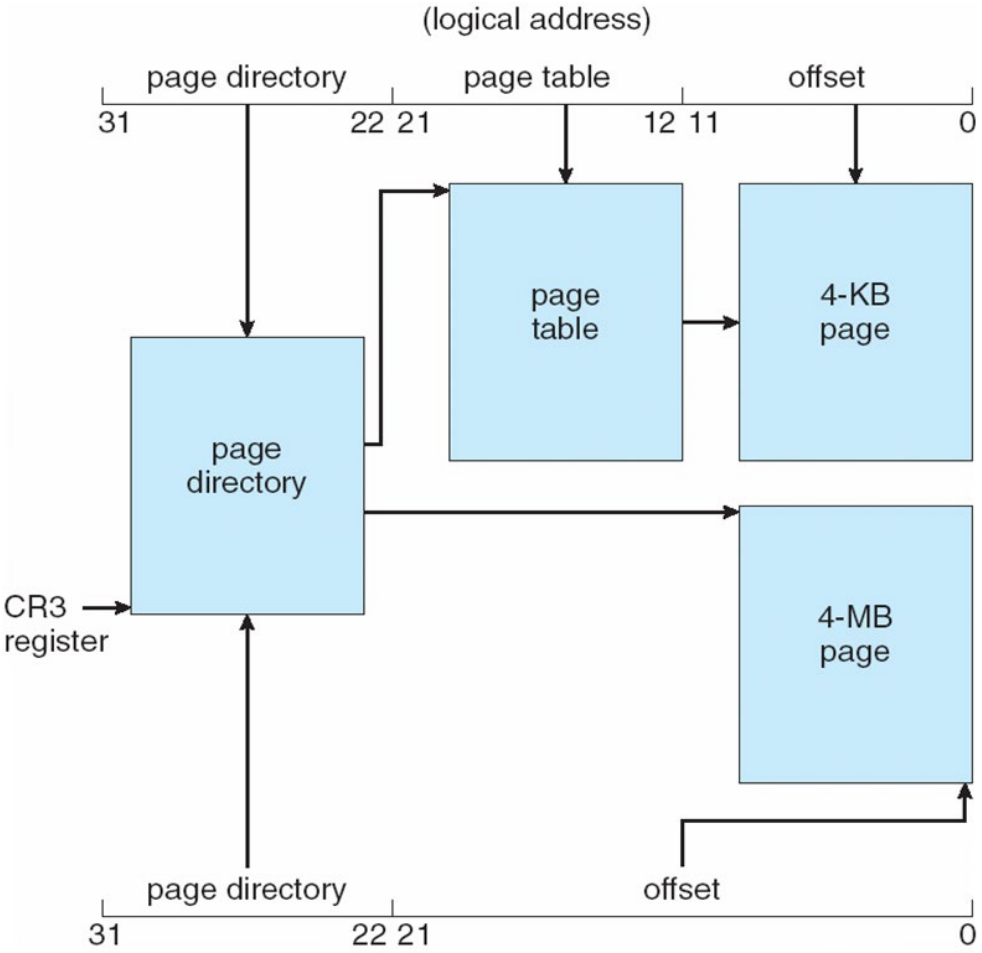


Logical to Physical Address Translation in IA-32



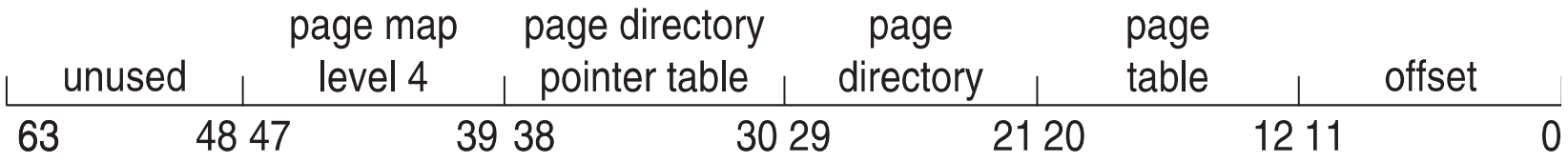


Intel IA-32 Paging Architecture



Intel x86-64

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy



Example: ARMv8 Architecture

- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 64-bit CPU (ARMv8)
  - 4-level of paging
  - 4 KB pages
  - 2 MB and 1GB pages (termed regions)