9. Memory Management

Sungyoung Lee

College of Engineering

KyungHee University

Contents

- *Background*
- *Swapping*
- *Contiguous Allocation*
- *Paging*
- *Segmentation*
- *Segmentation with Paging*

Memory Management

- Goals
  - To provide a convenient abstraction for programming
  - To allocate scarce memory resources among competing processes to maximize performance with minimal overhead
  - To provide isolation between processes

Memory Management (Cont'd)

- Batch programming
  - Programs use physical addresses directly
  - OS loads job, runs it, unloads it

- Multiprogramming
  - Need multiple processes in memory at once
    - Overlap I/O and CPU of multiple jobs
  - Can do it a number of ways
    - Fixed and variable partitioning, paging, segmentation, etc.
  - Requirements
    - Protection: restrict which addresses processes can use
    - Fast translation: memory lookups must be fast, in spite of protection scheme
    - Fast context switching: updating memory hardware (for protection and translation) should be quick

Memory Management (Cont'd)

- Issues
  - Support for multiple processes
    - Each process should have a logically contiguous space
    - The size of each space is variable
  - Enable a process to be larger than the amount of memory allocated to it
    - Not all the memory spaces are used simultaneously
    - Memory references exhibit spatial and temporal locality
  - Protection
  - Sharing
  - Support for multiple regions per process (segments)
  - Performance
    - Memory reference overhead
    - Context switching overhead

Memory Management (Cont'd)

- Solution: Virtual Memory (VM)
  - VM enables programs to execute without requiring their entire address space to be resident in physical memory
    - Program can also execute on machines with less RAM than it needs
  - Many programs don't need all of their code or data at once (or ever)
    - e.g., branches they never take, or data they never R/W
    - No need to allocate memory for it, OS should adjust amount allocated based on its run-time behavior
  - VM isolates processes from each other
    - One process cannot name addresses visible to others
    - Each process has its own isolated address space

Background

- Program must be brought into memory and placed within a process for it to be run

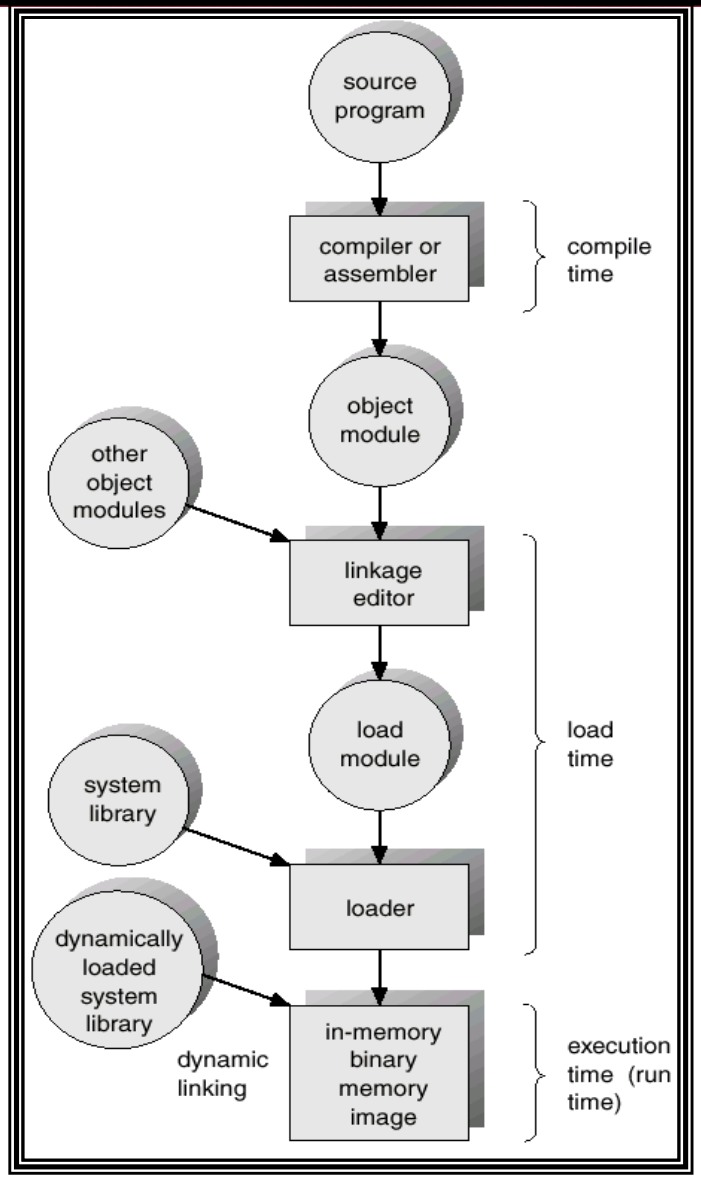
- *Input queue*
  - Collection of processes on the disk that are waiting to be brought into memory to run the program

- User programs go through several steps before being run

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 (Virtual) Address Space

- Example

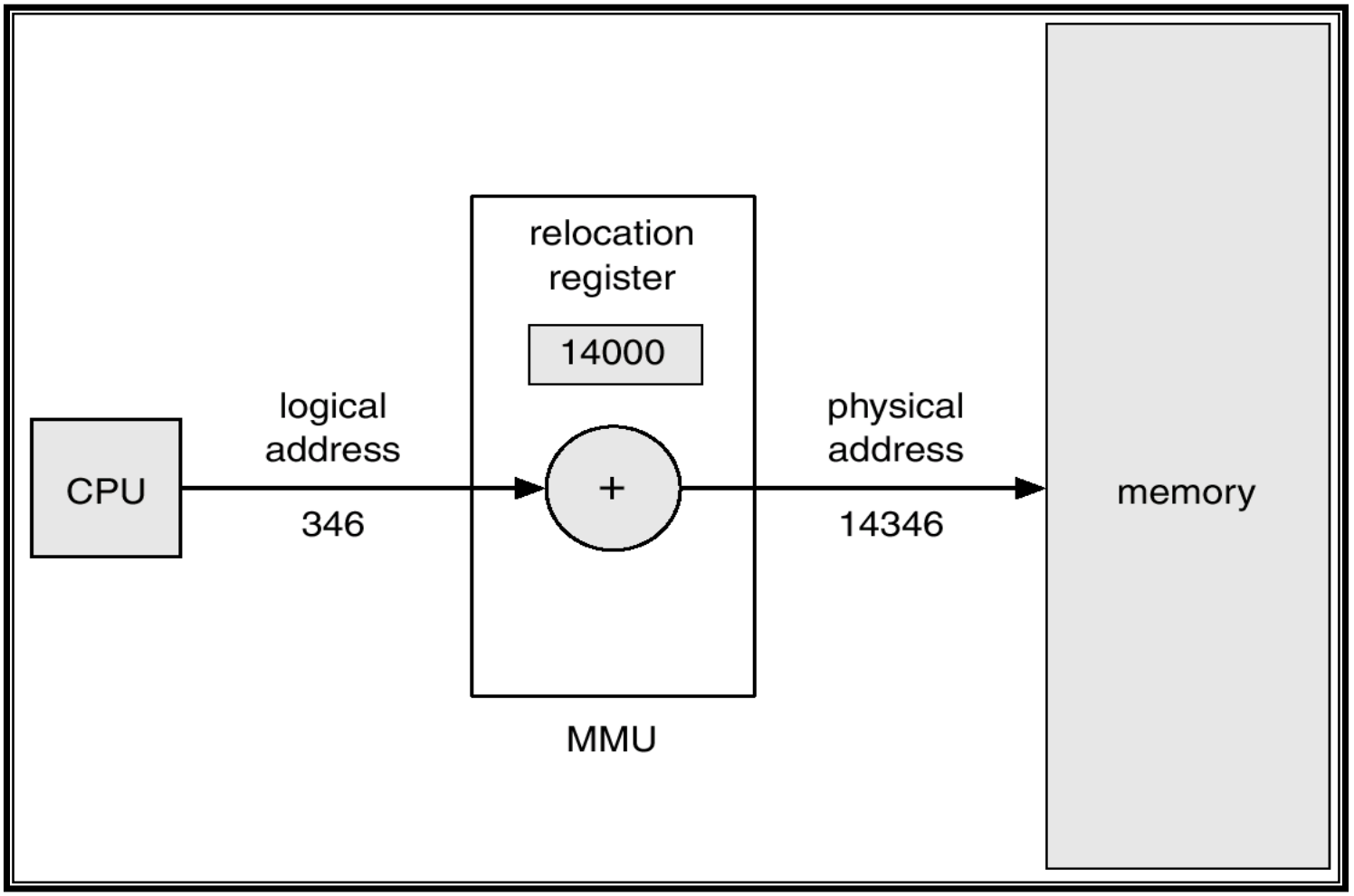
```
#include <stdio.h>

int n = 0;

int main ()
{
    printf ("&n = 0x%08x\n", &n);
}

% ./a.out
&n = 0x08049508
% ./a.out
&n = 0x08049508
```

- What happens if two users simultaneously run this application?

Memory-Management Unit (MMU)

- Hardware device that maps logical (virtual) to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses
  - It never sees the *real* physical addresses

Dynamic relocation using a relocation register



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

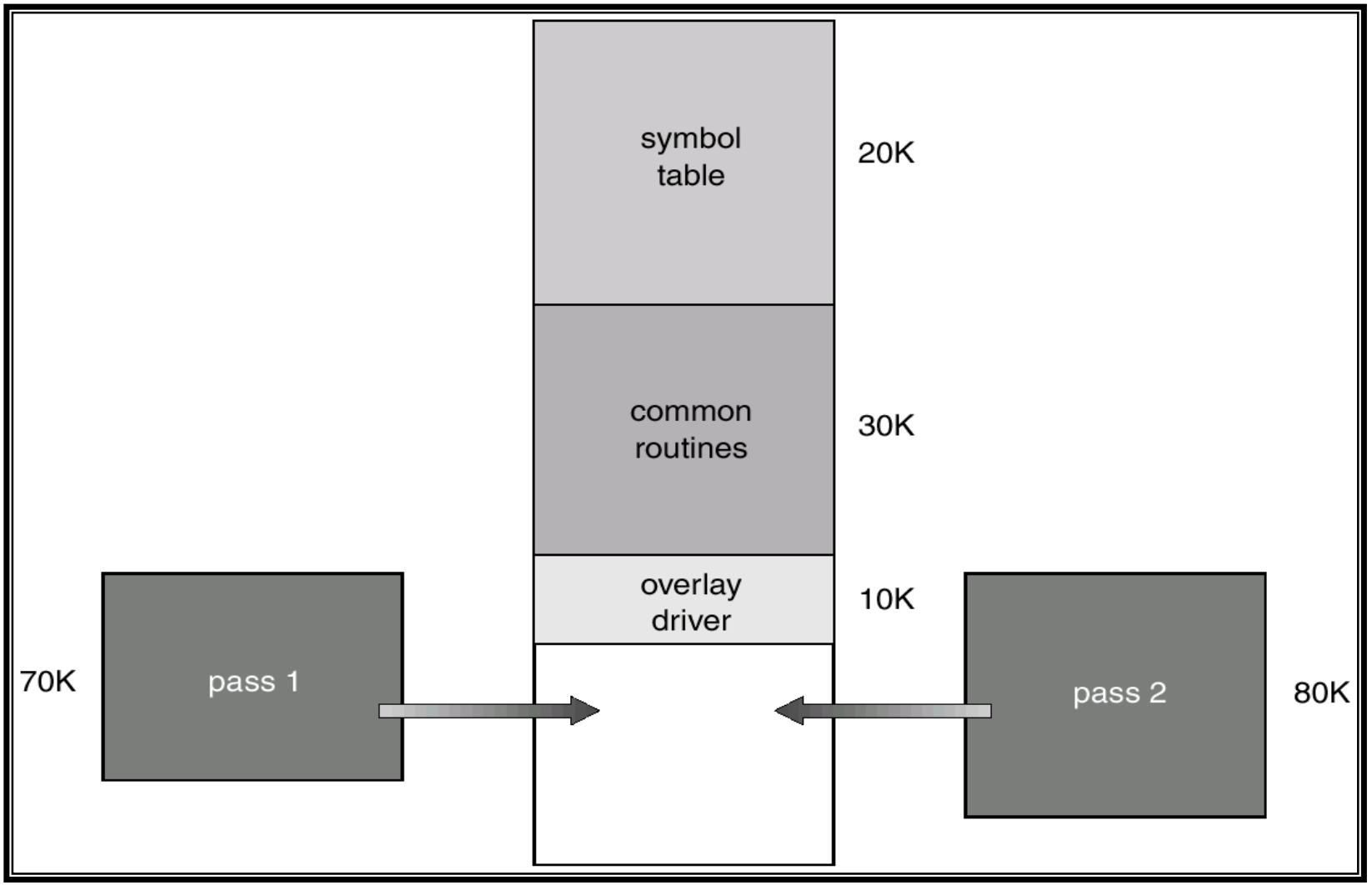
Dynamic Linking

- 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 is particularly useful for libraries

Overlays

- Keep in memory only those instructions and data that are needed at any given time
- Needed when process is larger than amount of memory allocated to it
- Implemented by user, no special support needed from operating system, programming design of overlay structure is complex

Overlays for a Two-Pass Assembler



Swapping

- A process can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution

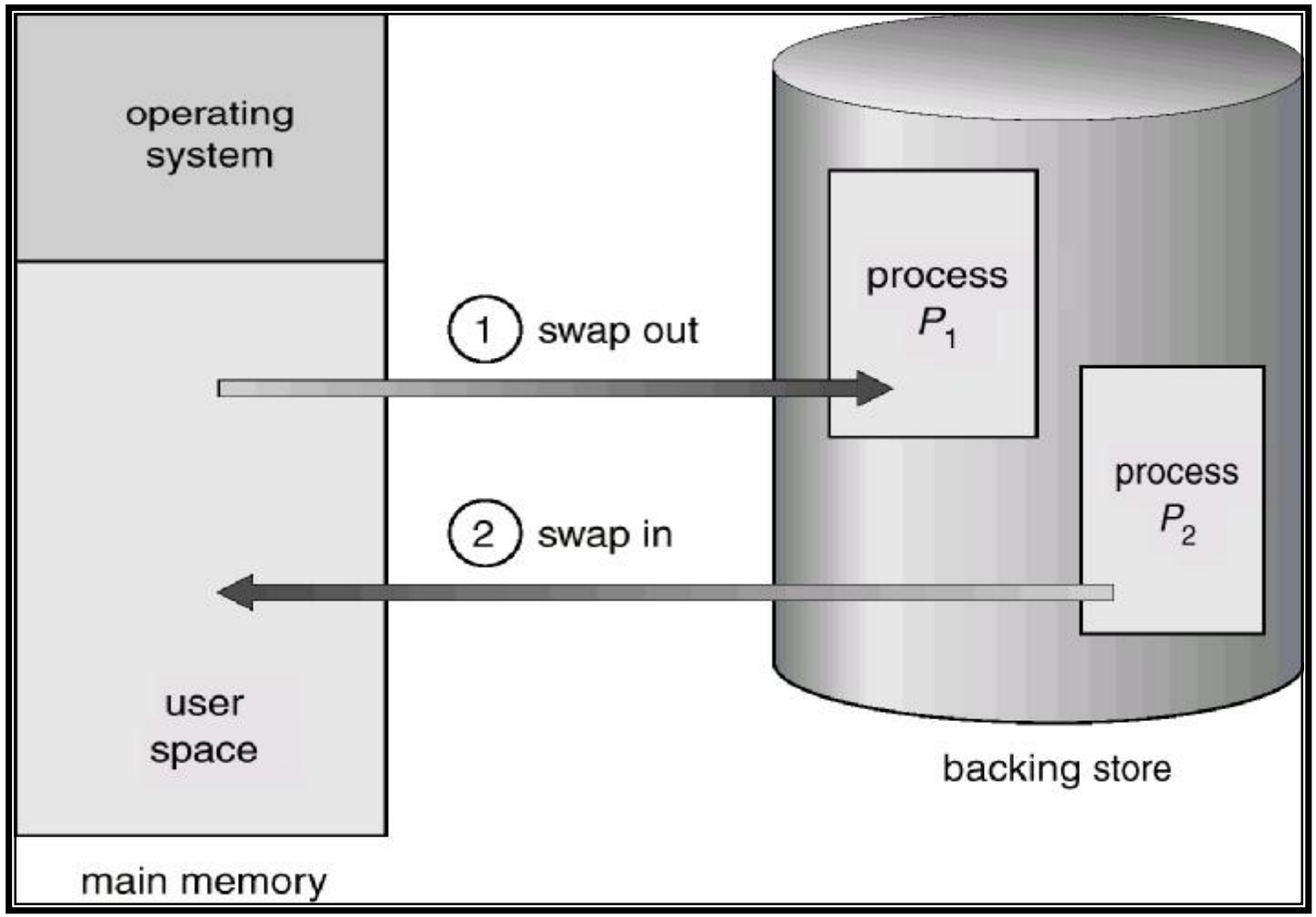
- Backing store
  - Fast disk large enough to accommodate copies of all memory images for all users
  - Must provide direct access to these memory images

- *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

- Modified versions of swapping are found on many systems
  - i.e., UNIX, Linux, and Windows

Schematic View of Swapping



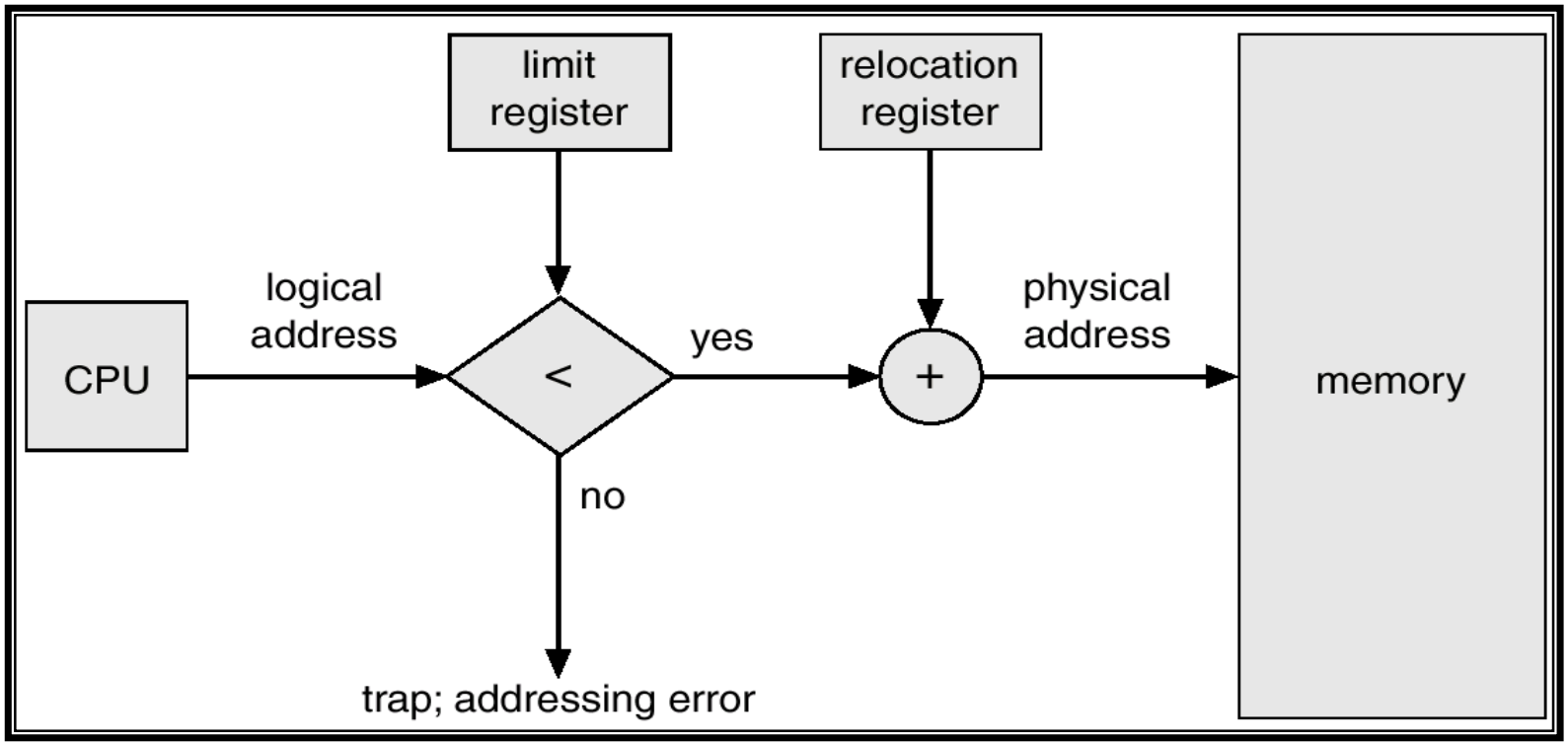
Contiguous Allocation

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

- Single-partition allocation
  - Relocation-register scheme used to protect user processes from each other, and from changing operating-system code and data
  - Relocation register contains value of smallest physical address
  - Limit register contains range of logical addresses
  - Each logical address must be less than the limit register

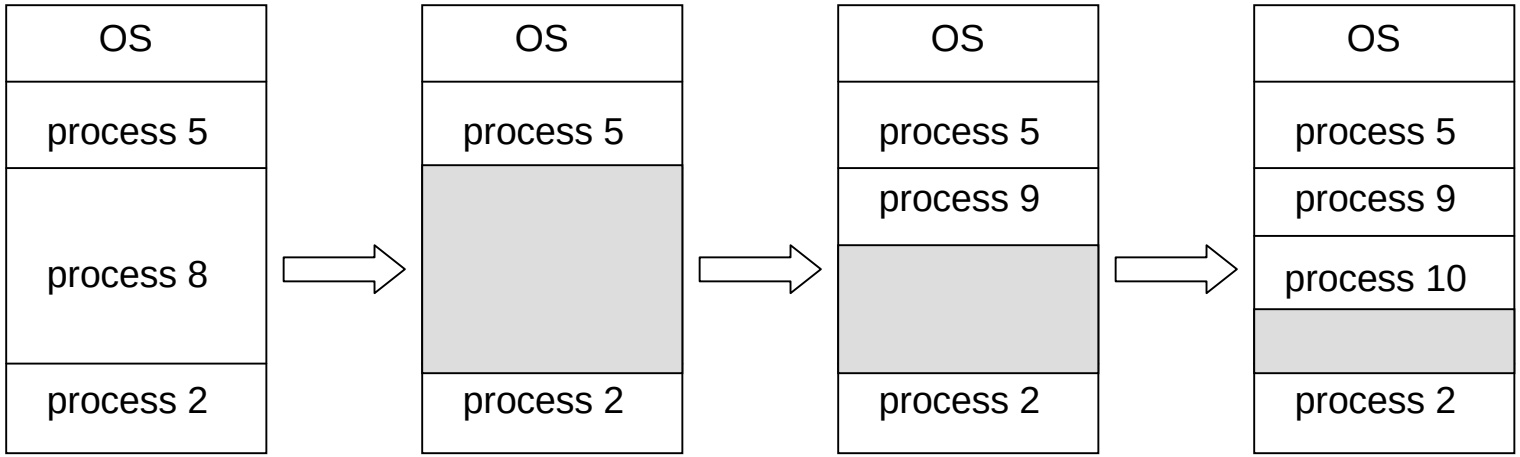
- *Cf) Physically contiguous vs. Logically contiguous allocation*

Hardware Support for Relocation and Limit Registers



Contiguous Allocation (Cont'd)

- 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

- 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

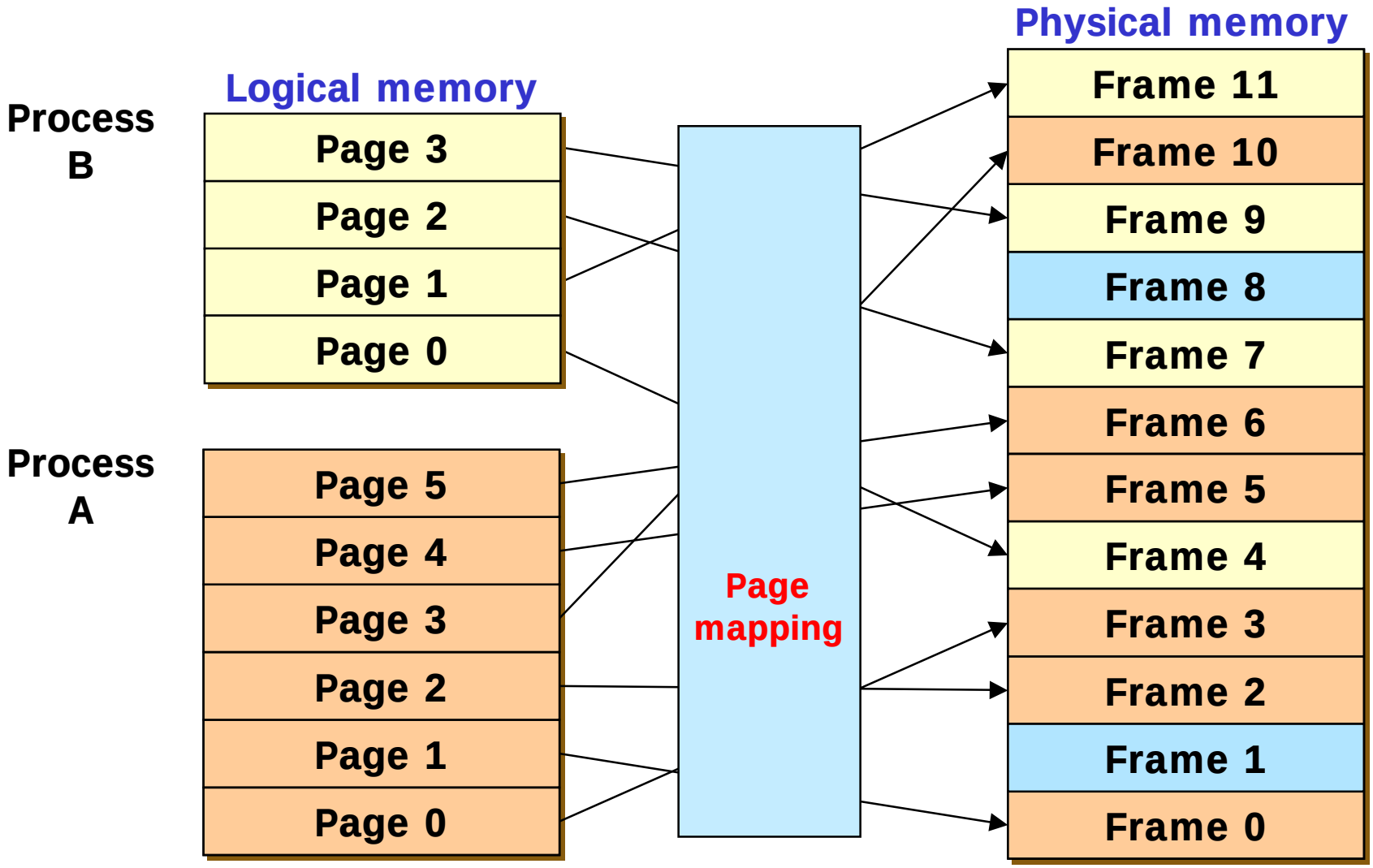
Fragmentation

- 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
- 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

Paging

- Physical 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**
- Divide logical memory into blocks of same size called **pages**
  - Frame (or Page) size is power of 2 (typically, 512 bytes ~ 8192 bytes)
- 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
- Internal fragmentation

Paging (Cont'd)



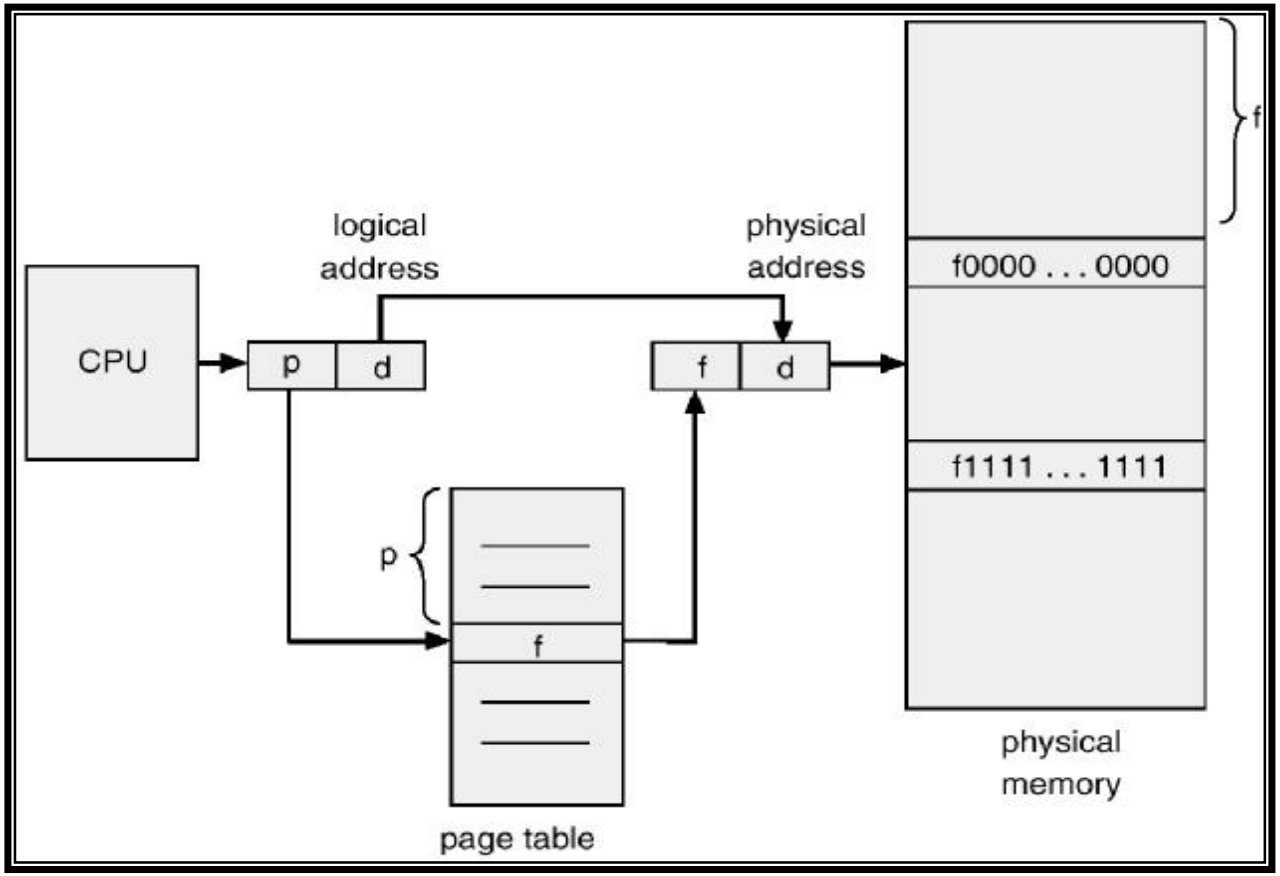
Paging (Cont'd)

- User's perspective
  - Users (and processes) view memory as one contiguous address space from 0 through N
    - Logical (or virtual) address space
  - In reality, pages are scattered throughout the physical memory
    - Logical-to-physical mapping
    - This mapping is invisible to the program
  - Protection is provided because a program cannot reference memory outside of its logical address space
    - The logical address 0xdeadcafe maps to different physical addresses for different processes.

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

Address Translation Architecture



Address Translation

- Translating addresses
  - A logical address has two parts:
    <page number::offset>
  - Page number is an index into a page table
  - Page table determines frame number
  - Physical address is <frame number::offset>

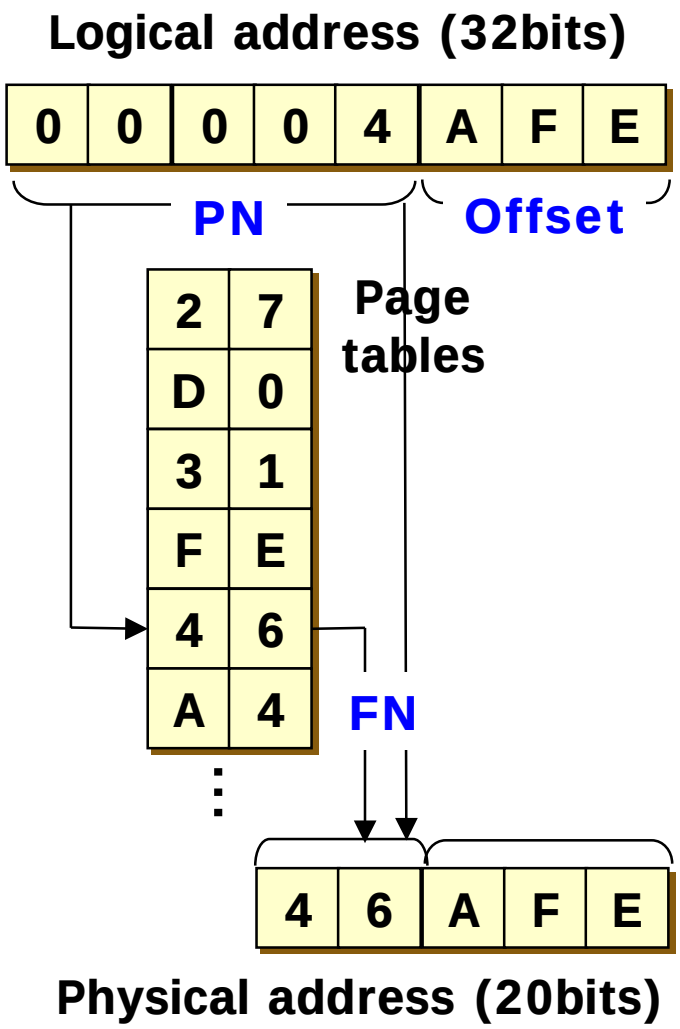
- Page tables
  - Managed by OS
  - Map page number to frame number
    - Page number is the index into the page table that determines frame number
  - One page table entry per page in virtual address space

- *Cf) Frame table*

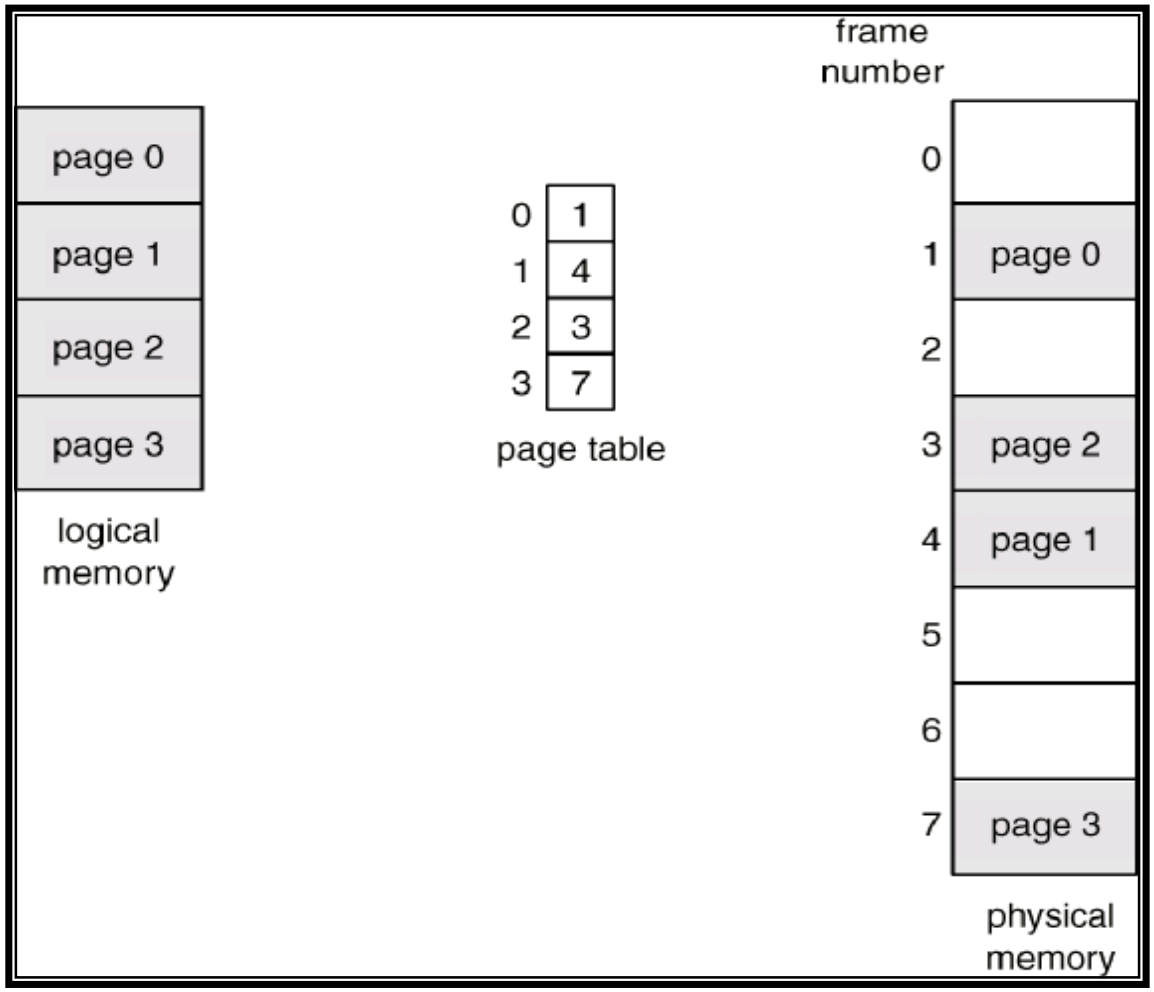
Paging Example

- Logical address: 32 bits
- Physical address: 20 bits
- Page size: 4KB

- Offset: 12 bits
- Page Number: 20 bits
- Page table entries: 2^{20}

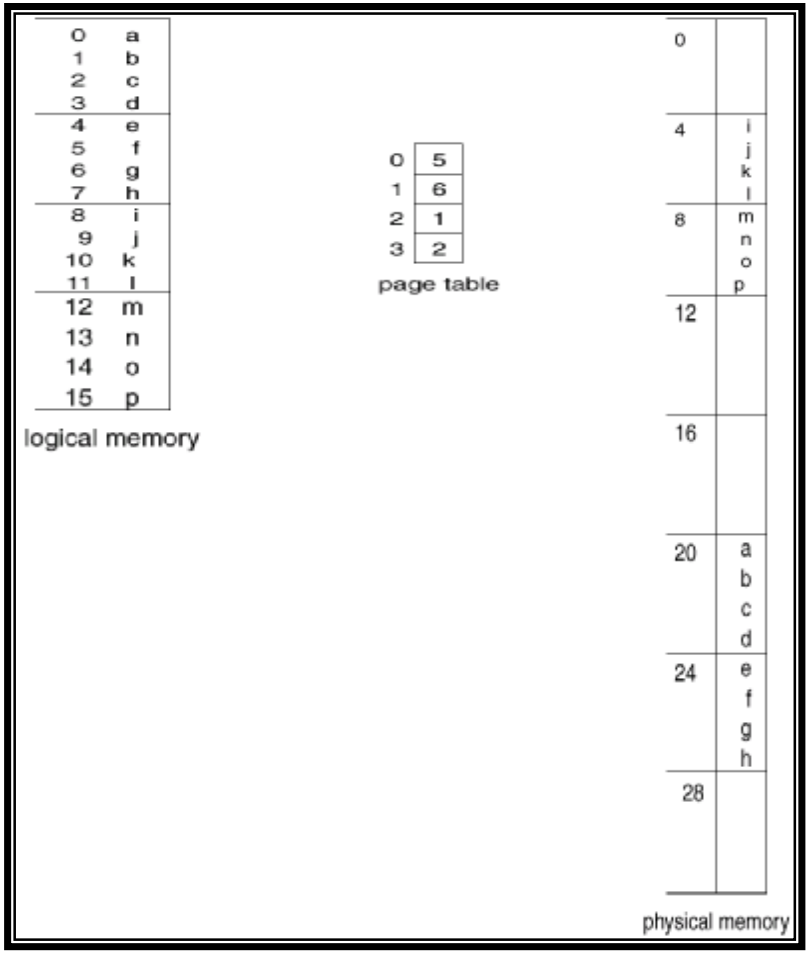


Paging Example

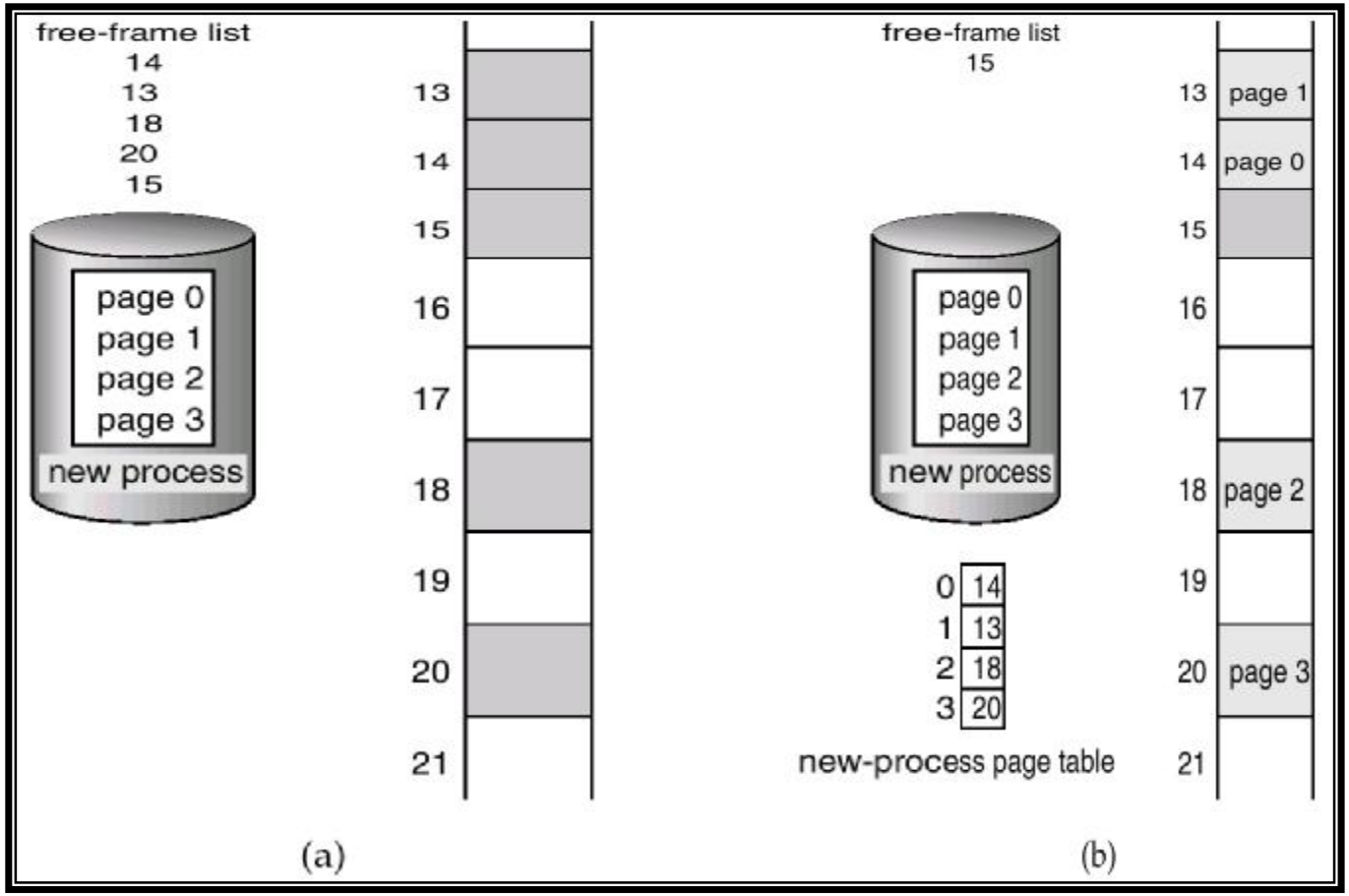


Paging Example (Cont'd)

- 32-byte memory with 4-byte pages



Free Frames



Before allocation

After allocation

Implementation of Page Table

- Page table is kept in main memory
- *Page-table base register (*PTBR) points to the page table
- *Page-table length register* (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)*

Associative Memory

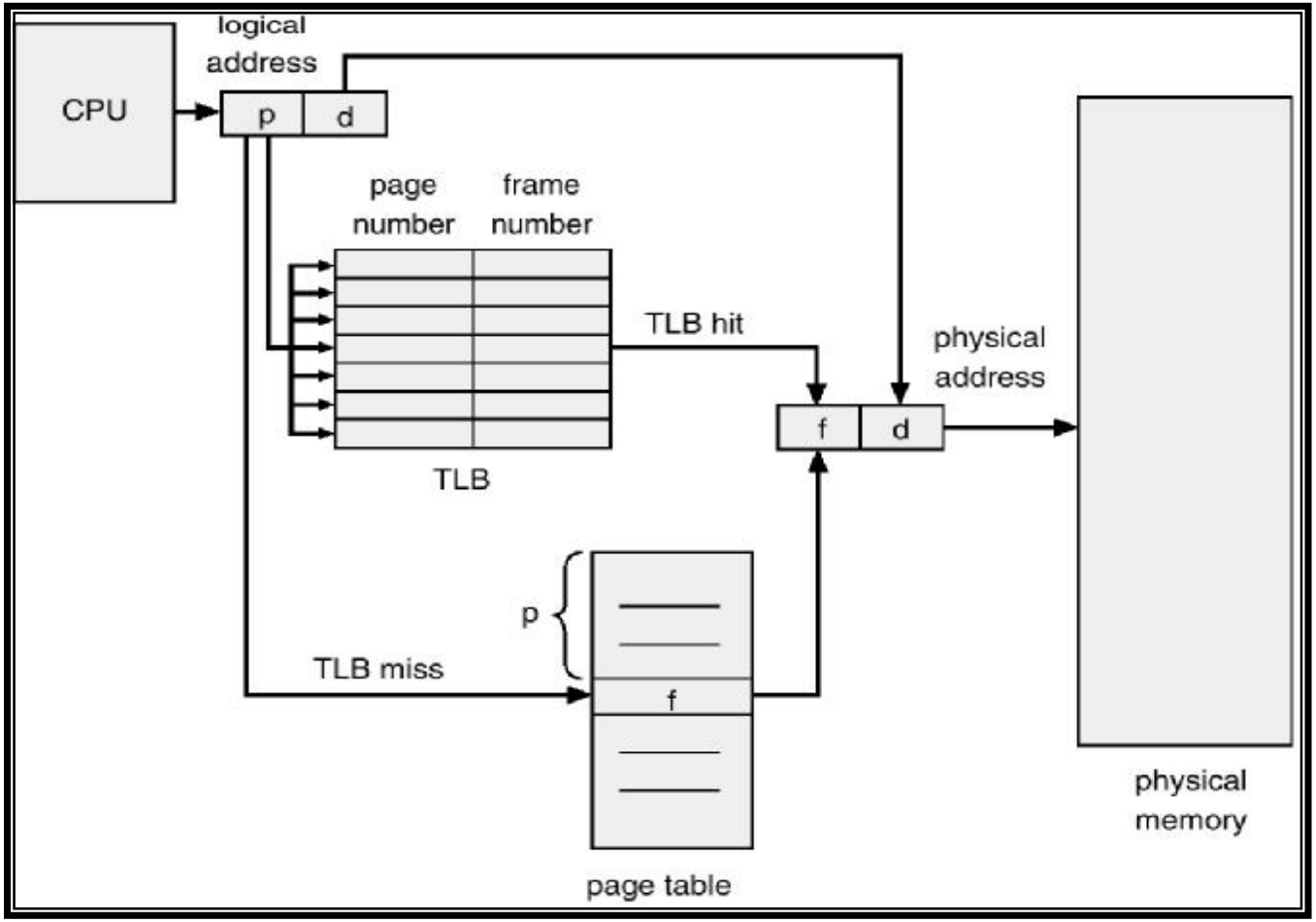
- Associative memory – parallel search

Page #	Frame #

- Address translation (A´, A´´)
  - If A´ is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

- *Cf) CAM (Content Addressable Memory)*

Paging Hardware With TLB



TLB

- Making address translation efficient
  - Original page table scheme doubled the cost of memory lookups
    - One lookup into the page table, another to fetch the data
  - Two-level page tables triple the cost!
    - Two lookups into the page tables, a third to fetch the data
    - And this assumes the page table is in memory
  - How can we make this more efficient?
    - Goal: make fetching from a virtual address about as efficient as fetching from a physical address
    - Solutions:
      - Cache the virtual-to-physical translation in hardware
      - Translation Look-aside Buffer (TLB)
      - TLB managed by the Memory Management Unit (MMU)

TLB (Cont'd)

- Translation Look-aside Buffers
  - Translate logical(virtual) page #s into page table entries (PTEs) (not physical address)
  - Can be done in a single machine cycle

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

TLB (Cont'd)

- TLB is implemented in hardware
  - Fully associative cache (all entries looked up in parallel)
  - Cache tags are logical(virtual) page numbers
  - Cache values are PTEs (entries from page tables)
  - With PTE+offset, MMU can directly calculate the physical address

- TLBs exploit locality
  - Processes only use a handful of pages at a time
    - 16~48 entries in TLB is typical (64~192KB)
    - Can hold the "hot set" or "working set" of process
  - Hit rates are therefore really important

- Locality
  - Temporal locality vs. Spatial locality

TLB (Cont'd)

- Handling TLB misses
  - Address translations are mostly handled by the TLB
    - > 99% of translations, but there are TLB misses occasionally
    - In case of a miss, who places translations into the TLB?
  - Hardware (MMU): Intel x86
    - Knows where page tables are in memory
    - OS maintains tables, HW access them directly
    - Page tables have to be in hardware-defined format
  - Software loaded TLB (OS)
    - TLB miss faults to OS, OS finds right PTE and loads TLB
    - Must be fast (but, 20-200 cycles typically)
    - CPU ISA has instructions for TLB manipulation
    - Page tables can be in any format convenient for OS (flexible)

TLB (Cont'd)

- Managing TLBs
  - OS ensures that TLB and page tables are consistent
    - When OS changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB
  - Reload TLB on a process context switch
    - Remember, each process typically has its own page tables
    - Need to invalidate all the entries in TLB (flush TLB)
    - In IA32, TLB is flushed automatically when the contents of CR3 (page directory base register) is changed
    - Cf) Alternatively, we can store the PID as part of the TLB entry, but this is expensive
  - When the TLB misses, and a new PTE is loaded, a cached PTE must be evicted
    - Choosing a victim PTE called the "TLB replacement policy"
    - Implemented in hardware, usually simple (e.g., LRU)

Effective Access Time

- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio
  - Percentage of times that a page number is found in the associative registers
  - Ratio related to number of associative registers
- Hit ratio = α
- Effective Access Time (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

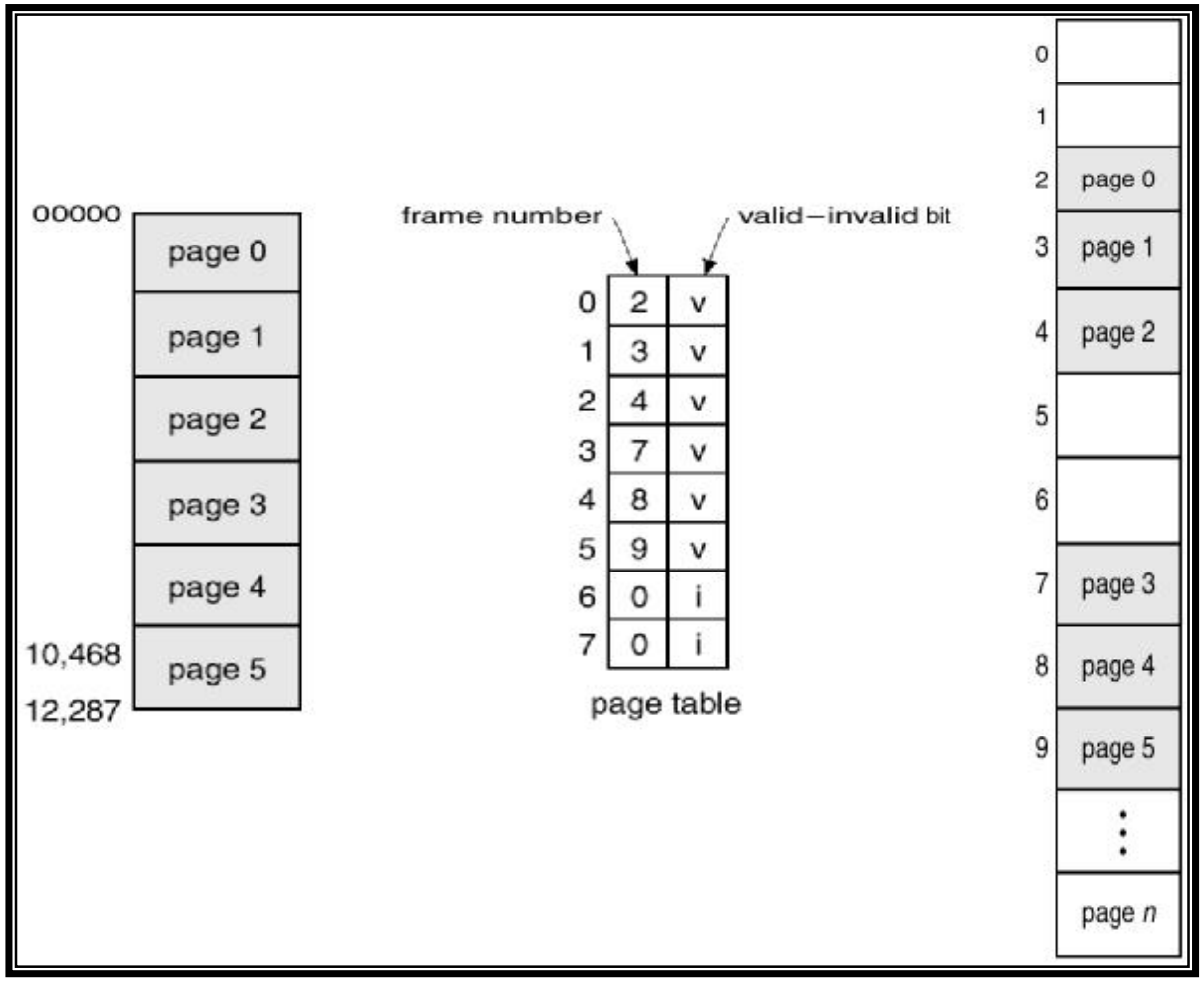
Memory Protection

- Memory protection implemented by associating protection bit with each frame

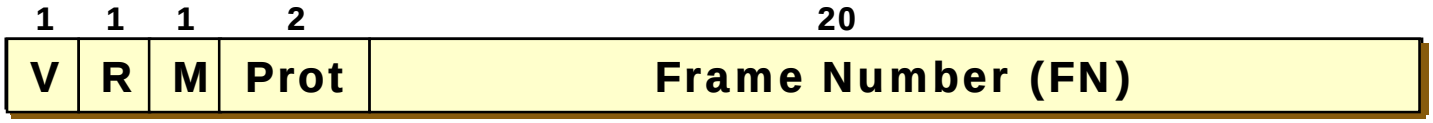
- *Valid-invalid* bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

- Finer level of protection is possible for valid pages
  - Provide read-only, read-write, or execute-only protection

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



Page Table Entries (PTEs)



- Valid bit (V) says whether or not the PTE can be used
  - It is checked each time a virtual address is used
- Reference bit (R) says whether the page has been accessed
  - It is set when a read or write to the page occurs
- Modify bit (M) says whether or not the page is dirty
  - It is set when a write to the page occurs
- Protection bits (Prot) control which operations are allowed on the page
  - Read, Write, Execute, etc.
- Frame number (FN) determines physical page

Page Table Structure

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Page Table Structure

- Managing page tables
  - Space overhead of page tables
    - The size of the page table for a 32-bit address space with 4KB pages = 4MB (per process)
  - How can we reduce this overhead?
    - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire address space)
  - How do we only map what is being used?
    - Make the page table structure dynamically extensible
    - Use another level of indirection:

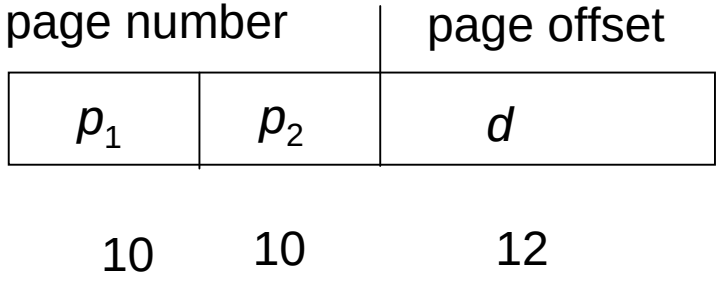
      - Two-level, hierarchical, hashed, etc.

Hierarchical Page Tables

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

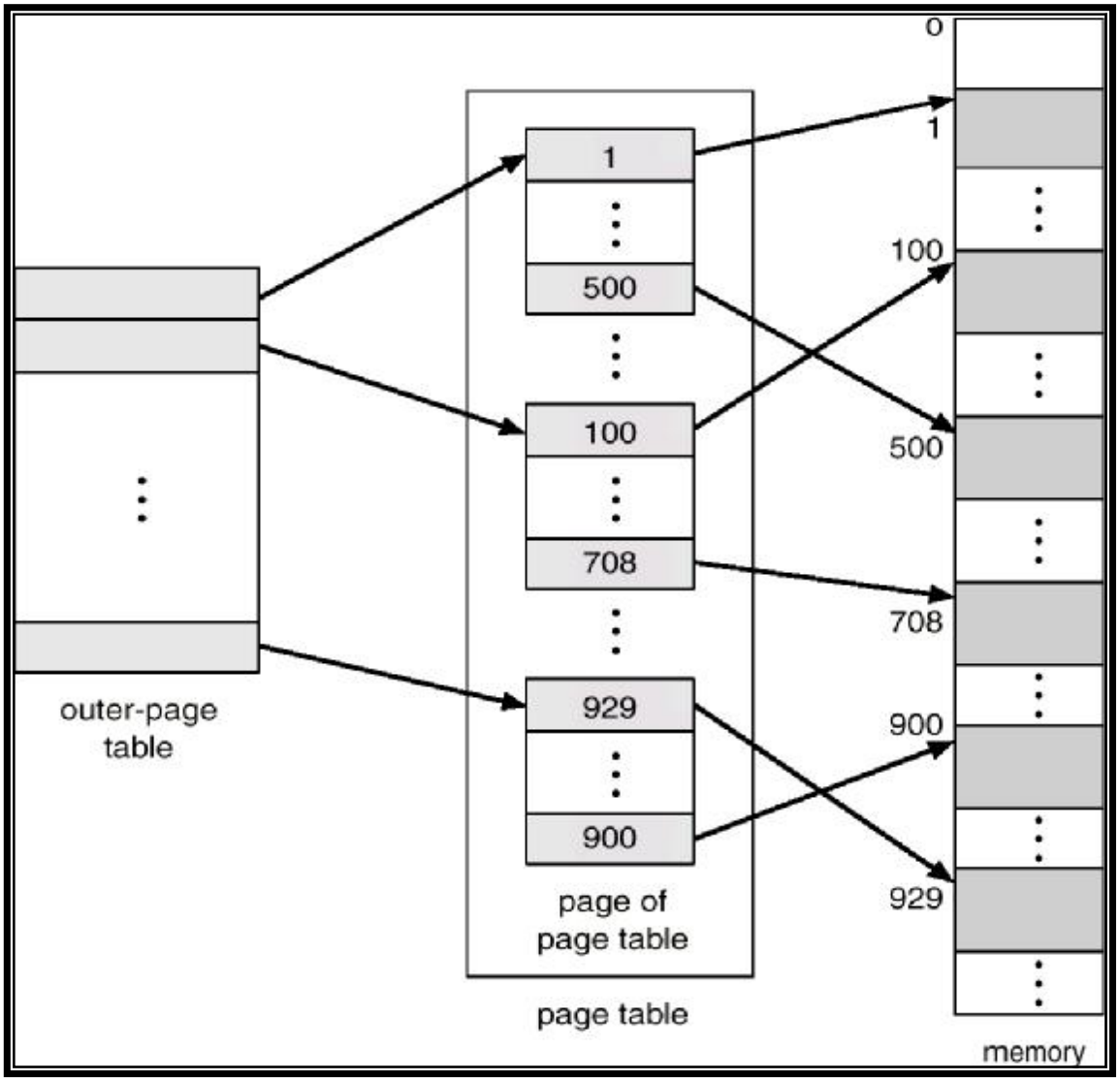
Two-Level Paging Example

- 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
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:



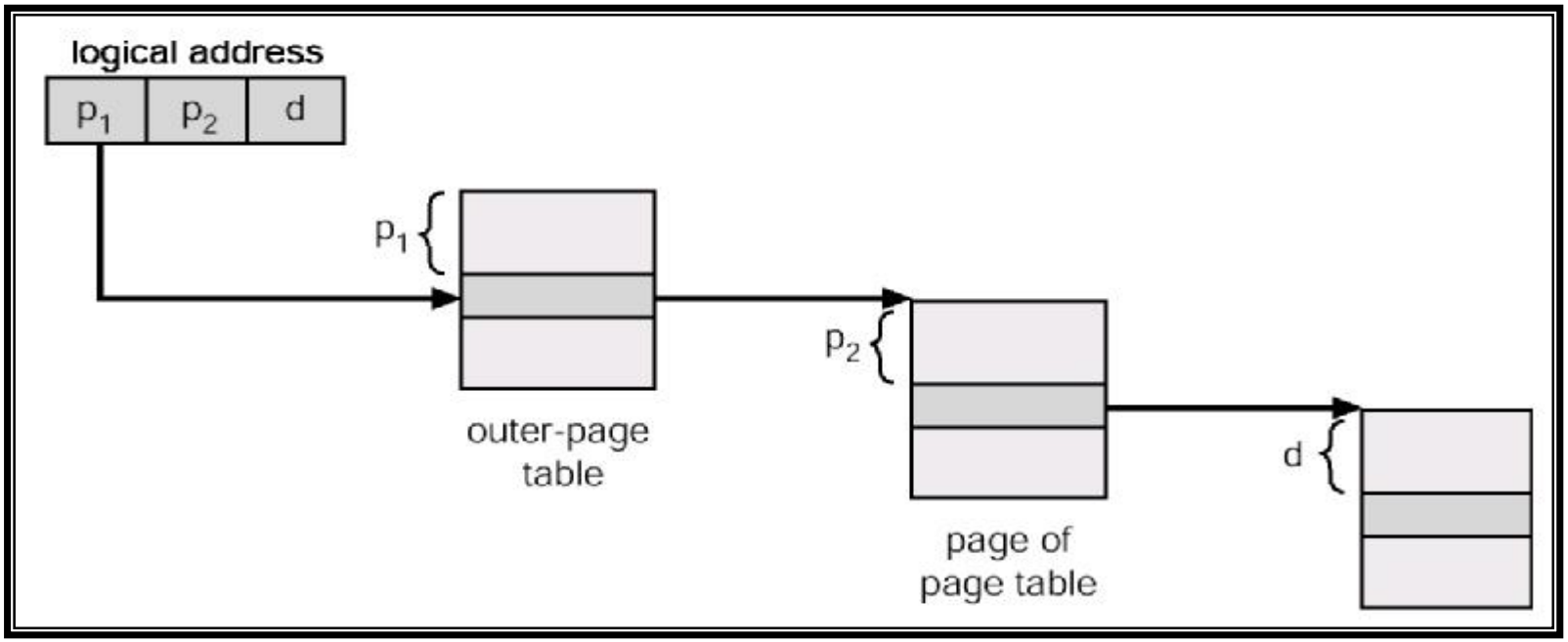
  - where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the outer page table

Two-Level Page-Table Scheme



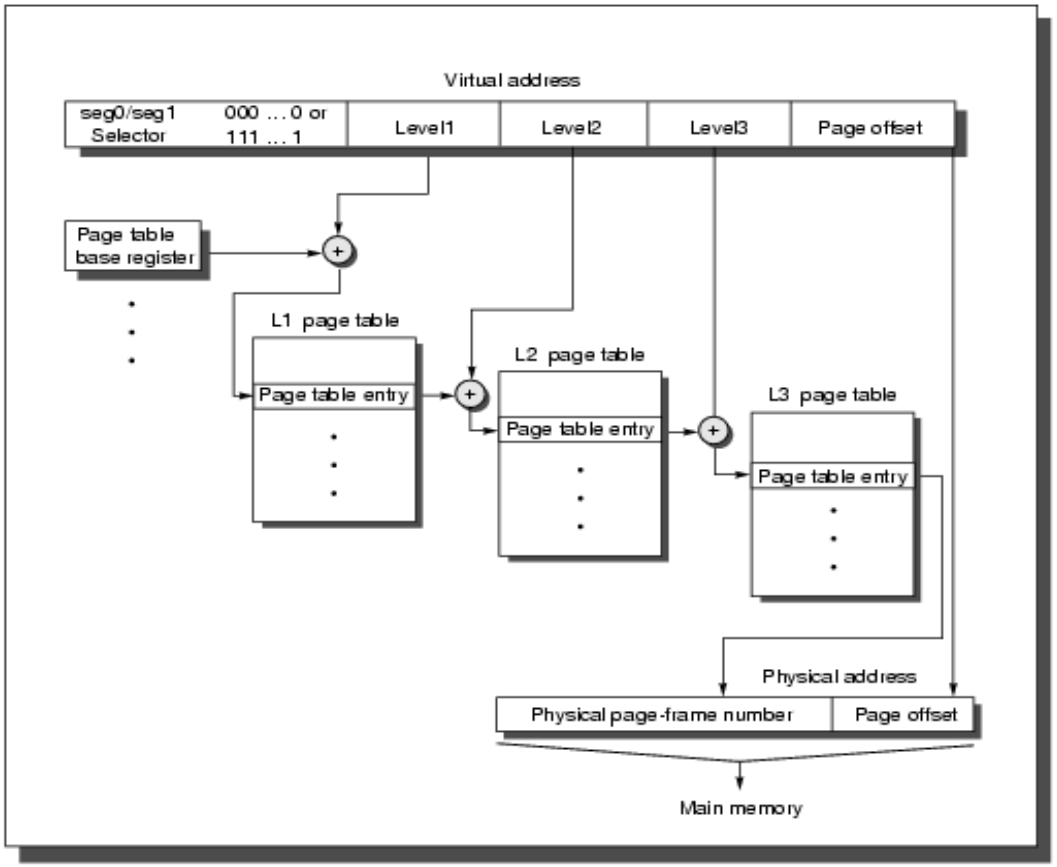
Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture



Multi-level Page Tables

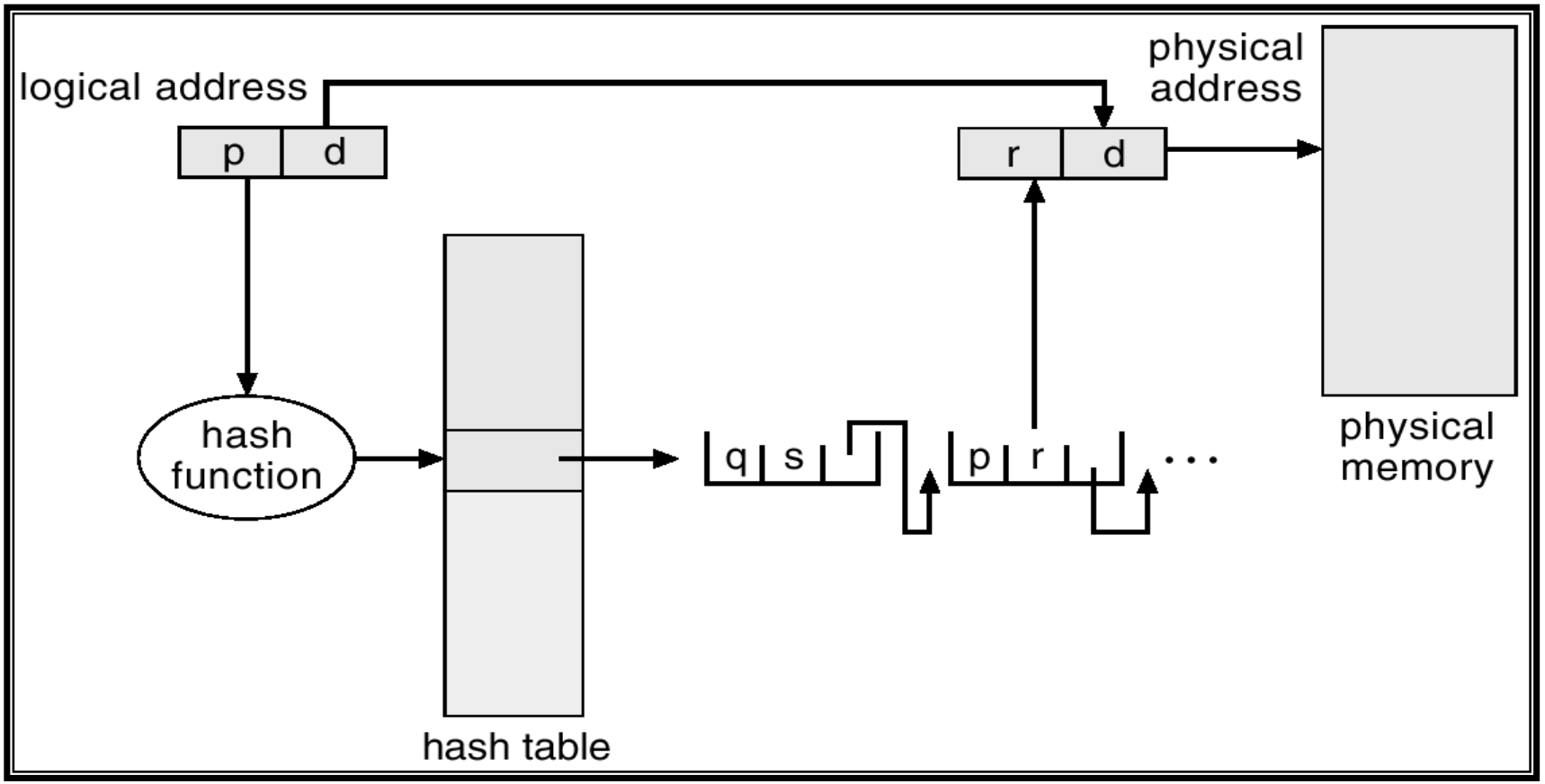
- Address translation in Alpha AXP Architecture
  - Three-level page tables
  - 64-bit address divided into 3 segments (coded in bits63/62)
    - seg0 (0x): user code
    - seg1 (11): user stack
    - kseg (10): kernel
  - Alpha 21064
    - Page size: 8KB
    - Virtual address: 43bits
    - Each page table is one page long



Hashed Page Tables

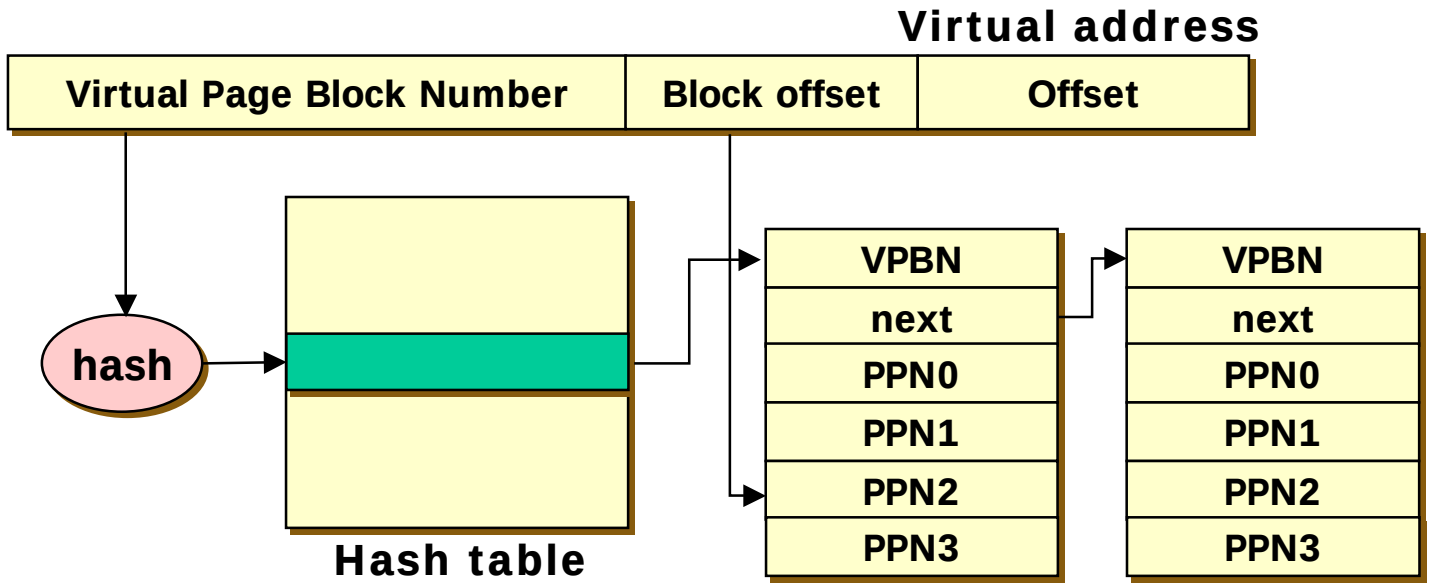
- Common in address spaces > 32 bits
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Virtual page numbers are compared in this chain searching for a match
  - If a match is found, the corresponding physical frame is extracted

Hashed Page Table



Hashed Page Tables

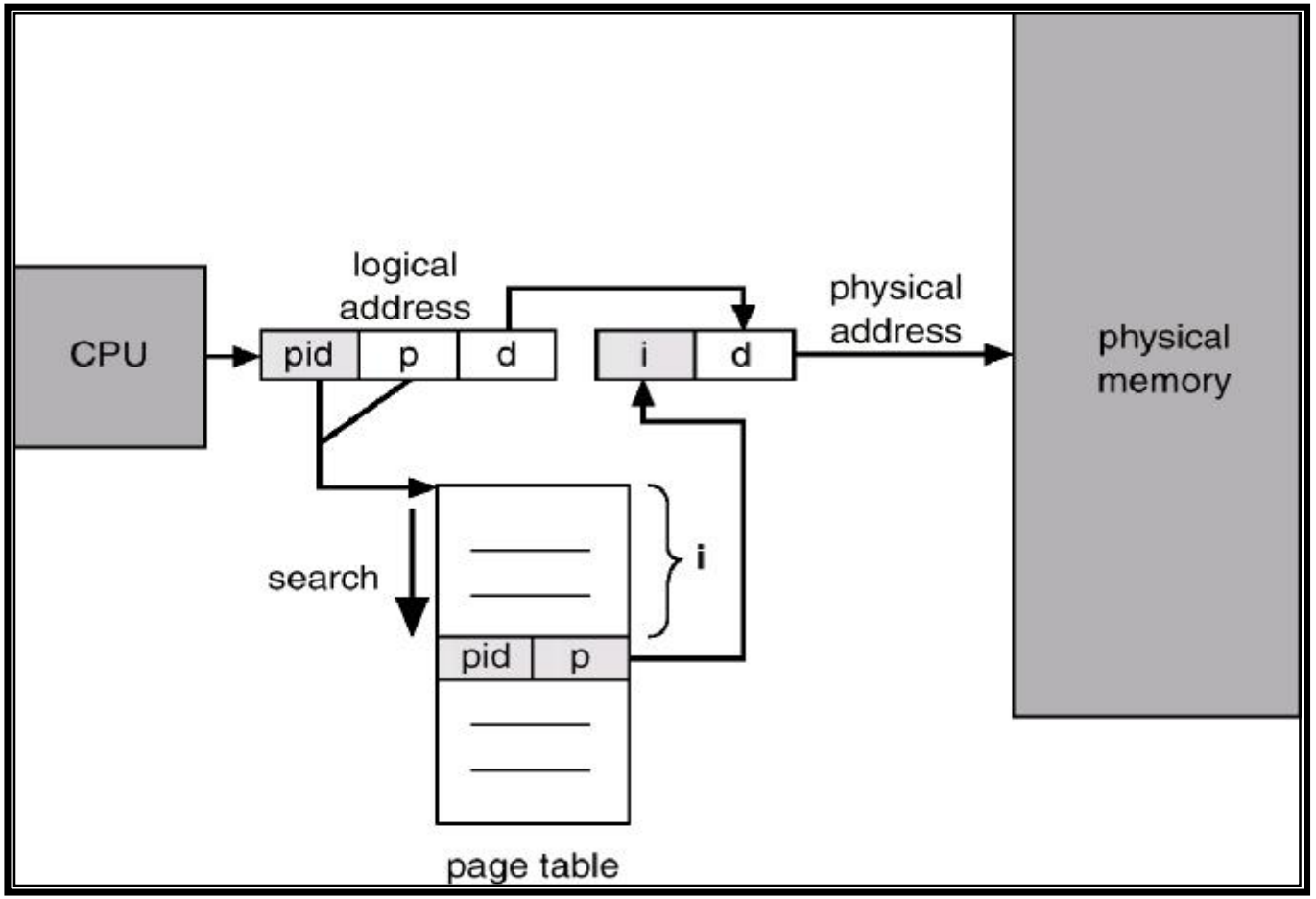
- Clustered page tables
  - A variant of hash page tables with the difference that each entry stores mapping information for a block of consecutive page tables



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

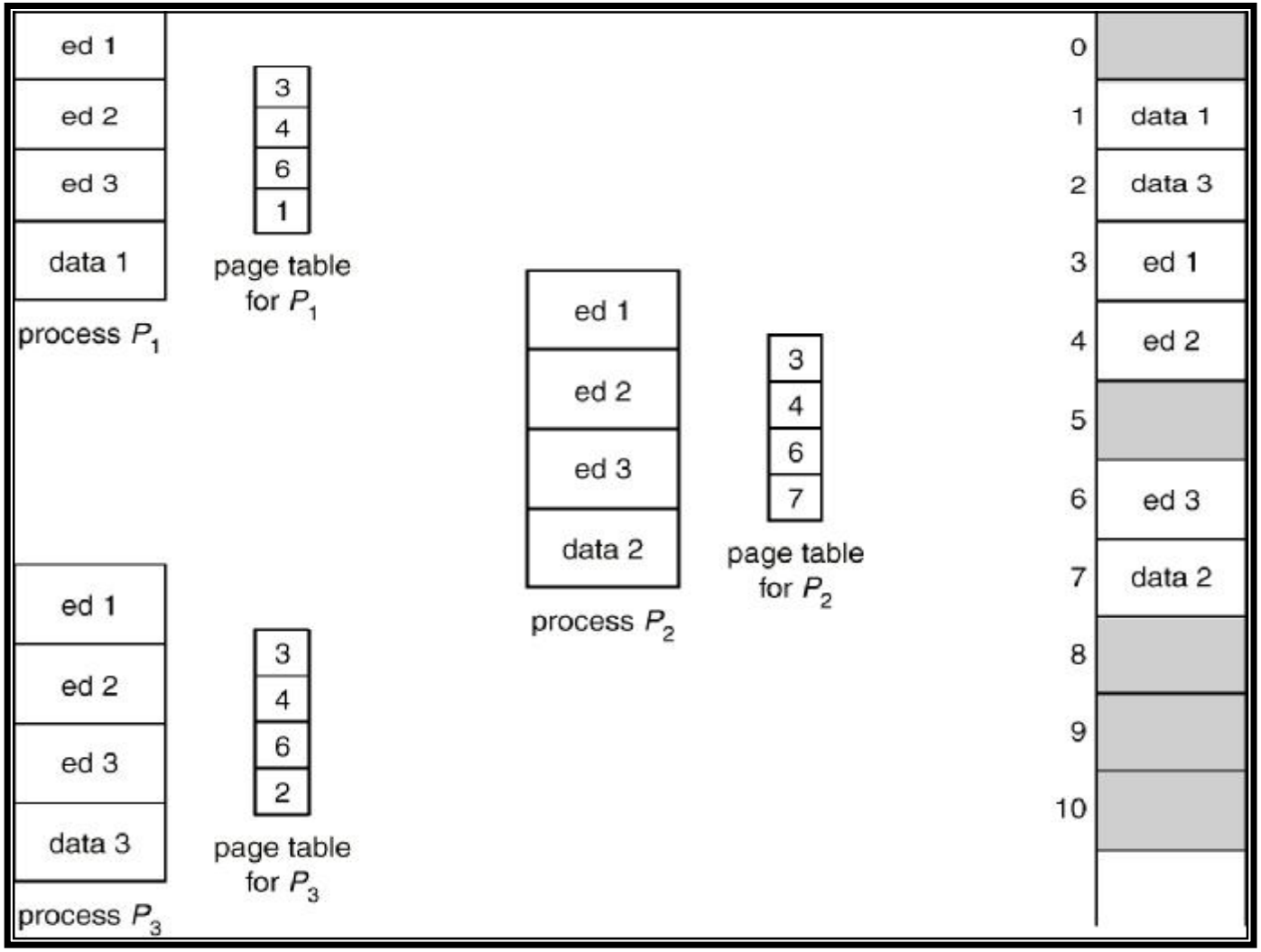


Shared Pages

- 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



Advantages of Paging

- Easy to allocate physical memory
  - Physical memory is allocated from free list of frames
  - To allocate a frame, just remove it from its free list
- No external fragmentation
- Easy to “page out” chunks of a program
  - All chunks are the same size (page size)
  - Use valid bit to detect reference to “paged-out” pages
  - Pages sizes are usually chosen to be convenient multiple of disk block sizes
- Easy to protect pages from illegal accesses
- Easy to share pages

Disadvantages of Paging

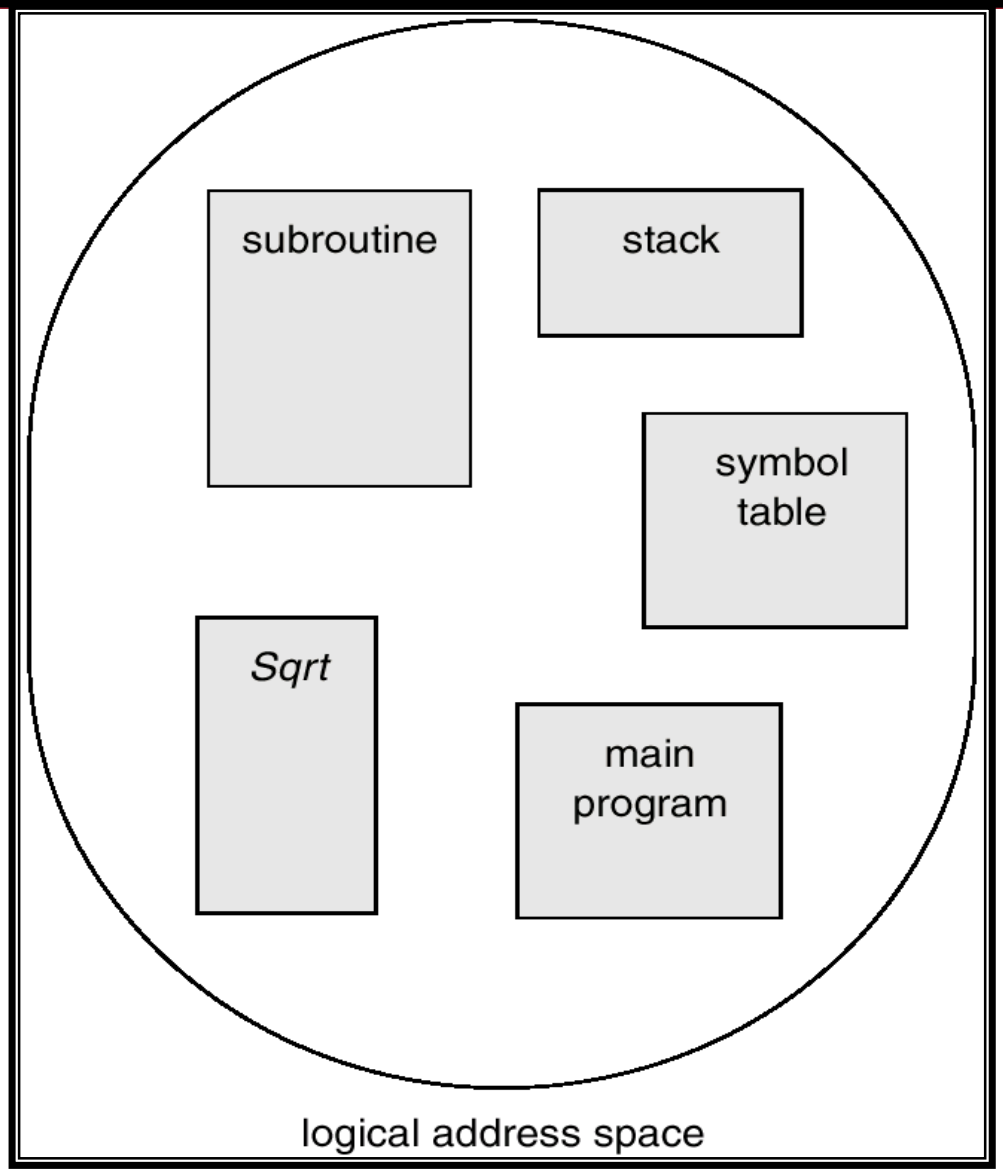
- Can still have internal fragmentation
  - Process may not use memory in exact multiple of pages
- Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - Solution
    - get a hardware support (TLB)
- Memory required to hold page tables can be large
  - Need one page table entry(PTE) per page in virtual address space
  - 32-bit address space with 4KB pages = 2^{20} PTEs
  - 4 bytes/PTE = 4MB per page table
  - OS's typically have separate page tables per process

    (25 processes = 100MB of page tables)
  - Solution
    - page the page tables, multi-level page tables, hashed page tables, inverted page tables, etc.

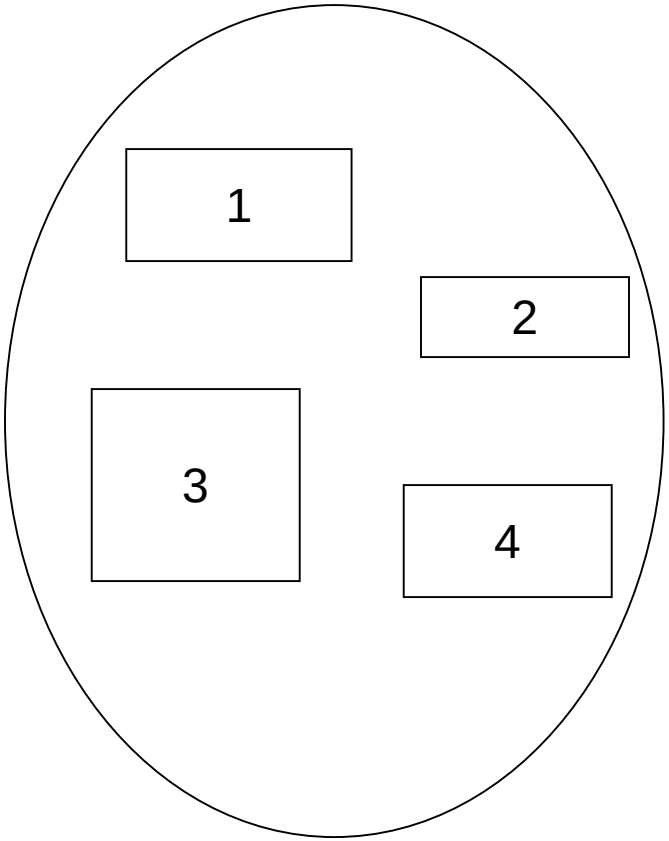
Segmentation

- 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

User's View of a Program



Logical View of Segmentation



user space

physical memory space

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'd)

- Relocation
  - dynamic
  - by segment table
- Sharing
  - shared segments
  - same segment number
- Allocation
  - first fit/best fit
  - external fragmentation

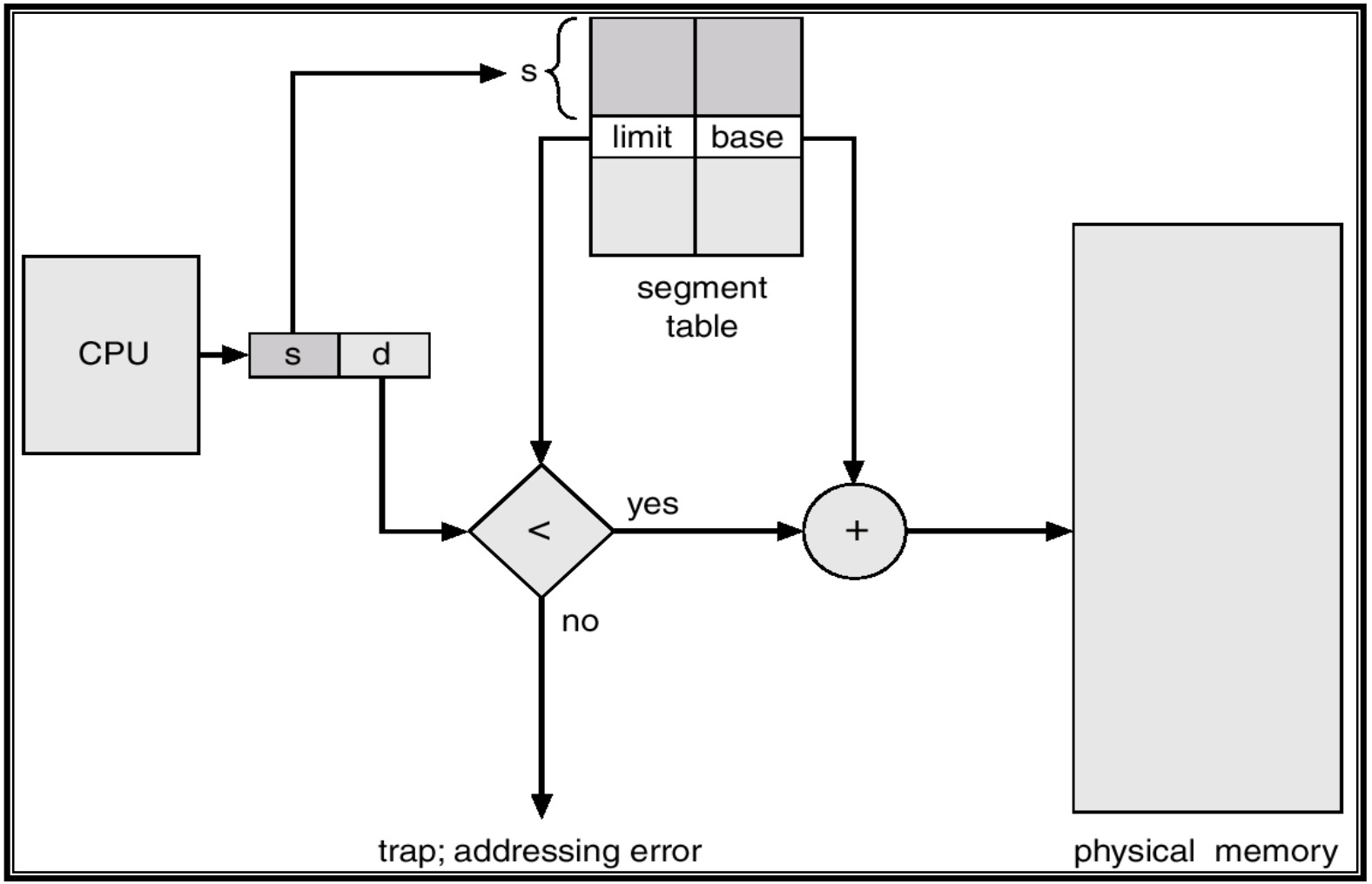
Segmentation Architecture (Cont'd)

- Protection
  - With each entry in segment table associate:
    - validation bit = 0 ⇒ illegal segment
    - read/write/execute privileges

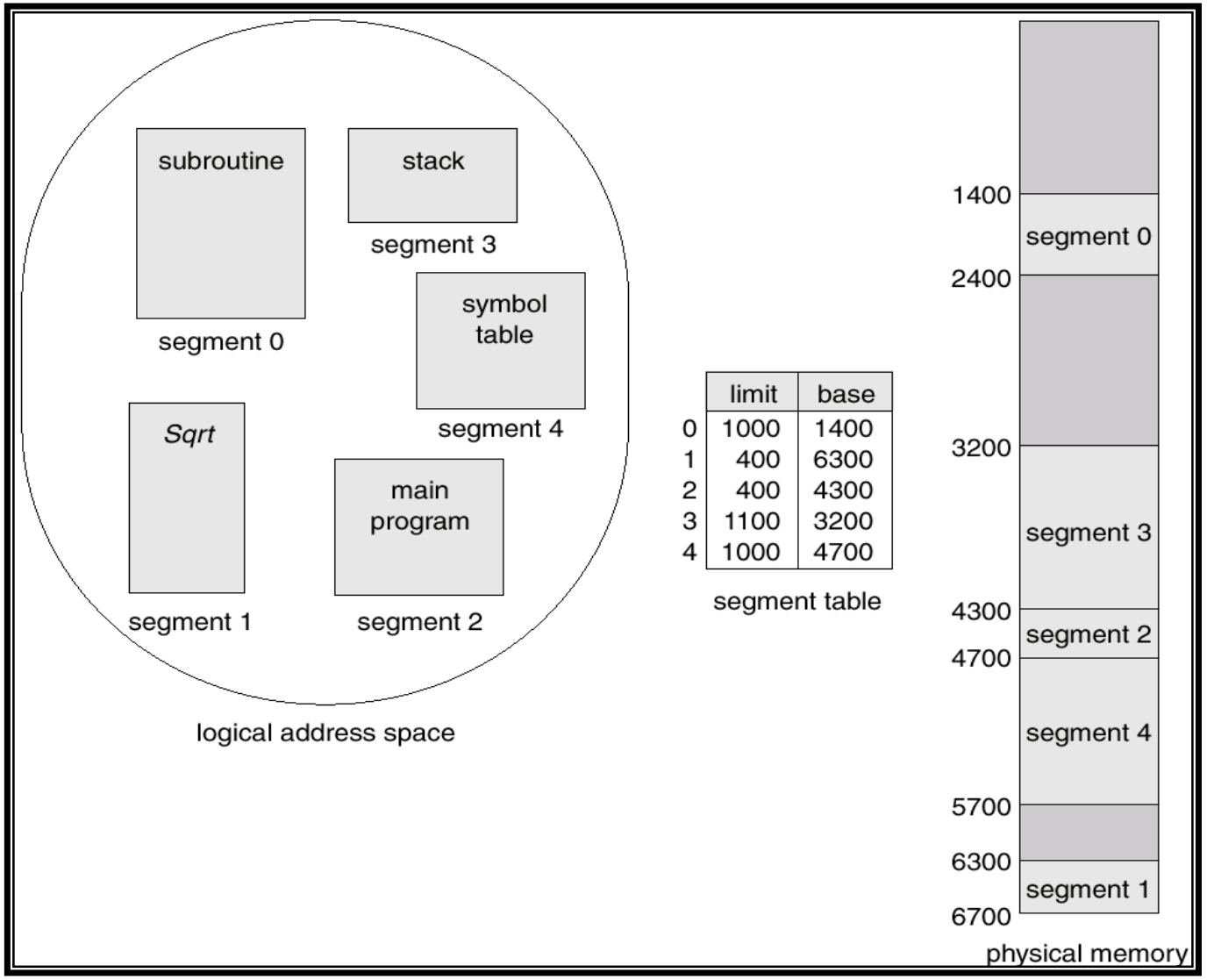
- Protection bits associated with segments
  - Code sharing occurs at segment level

- Since segments vary in length, memory allocation is a dynamic storage-allocation problem

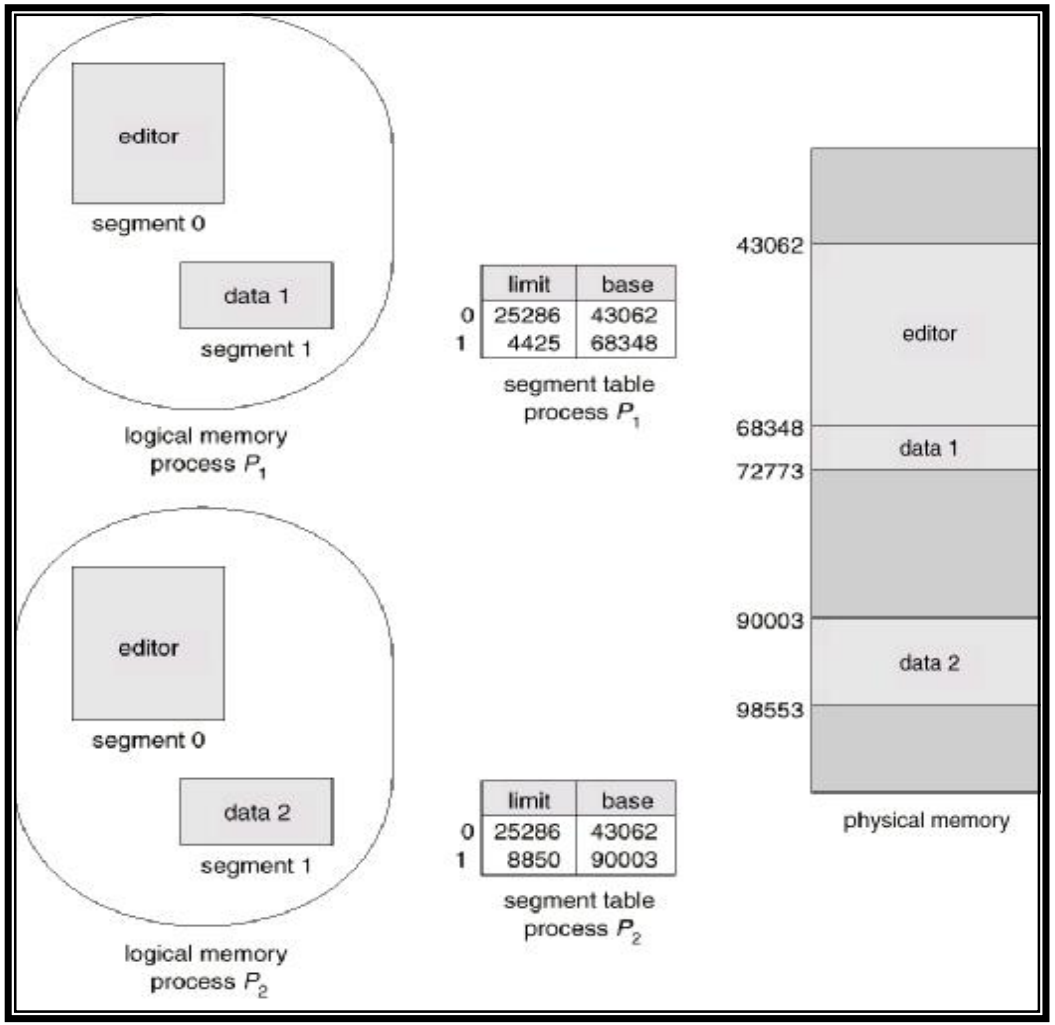
Segmentation Hardware



Example of Segmentation



Sharing of Segments



Advantages of Segmentation

- Simplifies the handling of data structures that are growing or shrinking

- Easy to protect segments
  - With each entry in segment table, associate a valid bit
  - Protection bits (read/write/execute) are also associated with each segment table entry

- Easy to share segments
  - Put same translation into base/limit pair
  - Code/data sharing occurs at segment level
  - e.g. shared libraries

- No internal fragmentation

Disadvantages of Segmentation

- Cross-segment addresses
  - Segments need to have same segment # for pointers to them to be shared among processes
  - Otherwise, use indirect addressing only
- Large segment tables
  - Keep in main memory, use hardware cache for speed
- External fragmentation
  - Since segments vary in length, memory allocation is a dynamic storage-allocation problem

Paging vs. Segmentation

	Paging	Segmentation
Block size	Fixed (4KB to 64KB)	Variable
Linear address space	1	Many
Memory addressing	One word (page number + offset)	Two words (segment & offset)
Replacement	Easy (all same size)	Difficult (find where segment fits)
Fragmentation	Internal	External
Disk traffic	Efficient (optimized for page size)	Inefficient (may have small or large transfers)
Transparent to the programmers?	Yes	No

Paging vs. Segmentation (Cont'd)

	Paging	Segmentation
Can the total address space exceed the size of physical memory?	Yes	Yes
Can codes and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of codes between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Paging vs. Segmentation (Cont'd)

- Hybrid approaches
  - Paged segments
    - Segmentation with Paging
    - Segments are a multiple of a page size
  - Multiple page sizes
    - 4KB, 2MB, and 4MB page sizes are supported in IA32
    - 8KB, 16KB, 32KB or 64KB in Alpha AXP Architecture (43, 47, 51, or 55 bits virtual address)

Segmentation with Paging

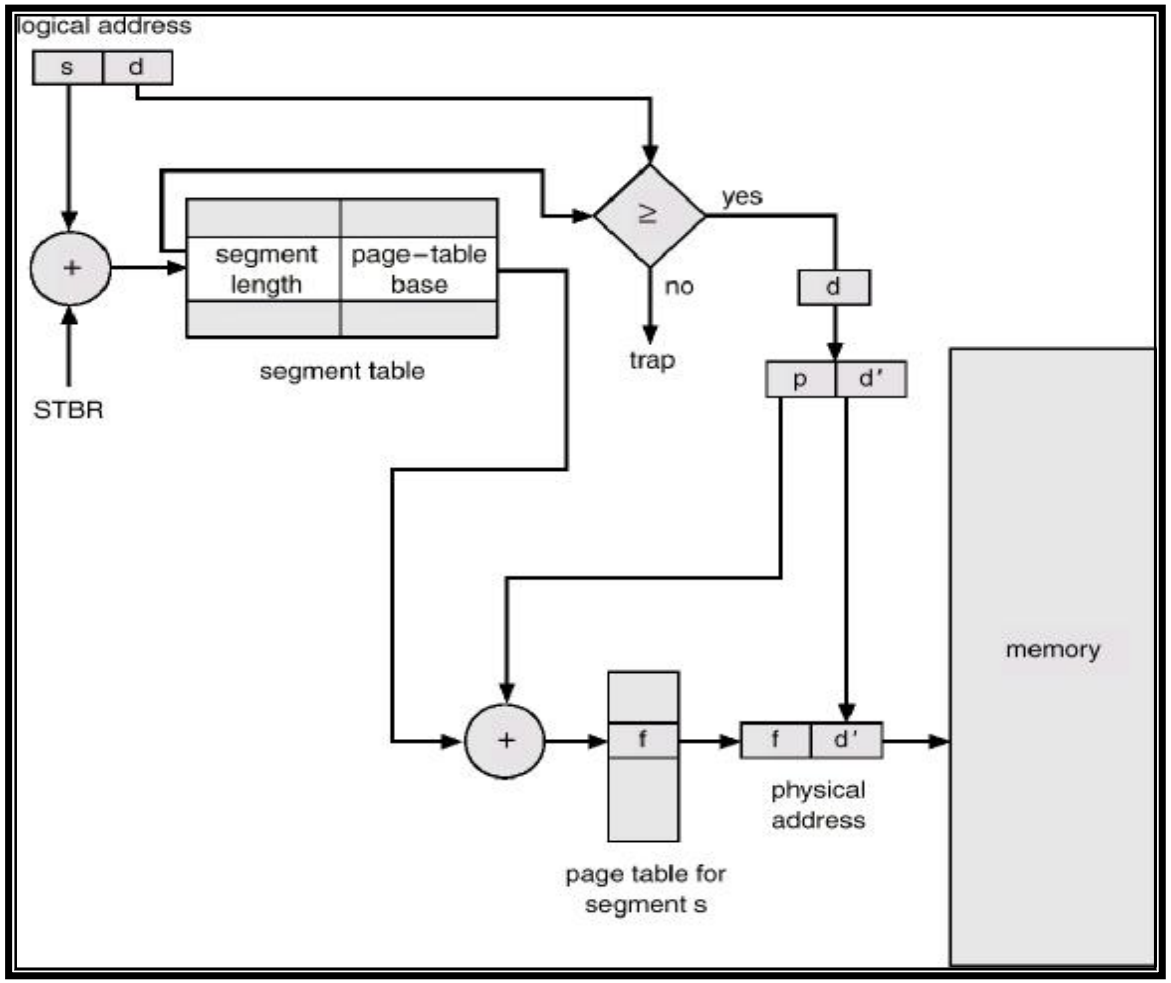
- Combine segmentation and paging
  - Use segments to manage logically related units
    - Code, data, heap, etc.
    - Segments vary in size, but usually large (multiple pages)
  - Use pages to partition segments into fixed size chunks
    - Makes segments easier to manage with in physical memory
    - Segments become “pageable” – rather than moving segments into and out of memory, just move page portions of segments
    - Need to allocate page table entries only for those pieces of the segments that have themselves been allocated
    - No external fragmentation
  - The IA32 supports segments and paging

Segmentation with Paging – MULTICS

- The MULTICS system solved problems of external fragmentation and lengthy search times by paging the segments

- Solution differs from pure segmentation in that the segment-table entry contains not the base address of the segment, but rather the base address of a *page table* for this segment

MULTICS Address Translation Scheme



Segmentation with Paging – Intel 386

- The Intel 386 uses segmentation with paging for memory management with a two-level paging scheme