Operating System

Ch09: Main memory

BeomSeok Kim
Department of Computer Engineering
KyungHee University
passion0822@khu.ac.kr

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

- 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

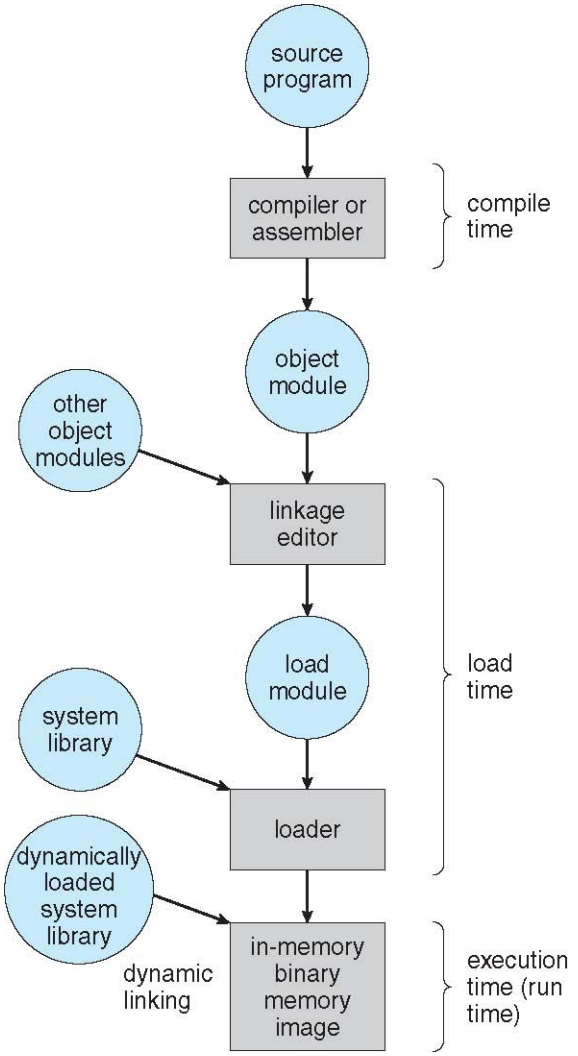
- Issues
  - ✓ Support for multiple processes
    - Each process should have a logically contiguous space
    - The size of each space is variable
  - ✓ Enable a process to be larger than the amount of memory allocated to it
    - Not all the memory spaces are used simultaneously
    - Memory references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

Memory Management

- 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
  - ✓ Chap. 9) mapping from virtual (logical) address to physical address
  - ✓ Chap. 10) virtual memory larger than physical memory

Binding of Instructions and Data to Memory

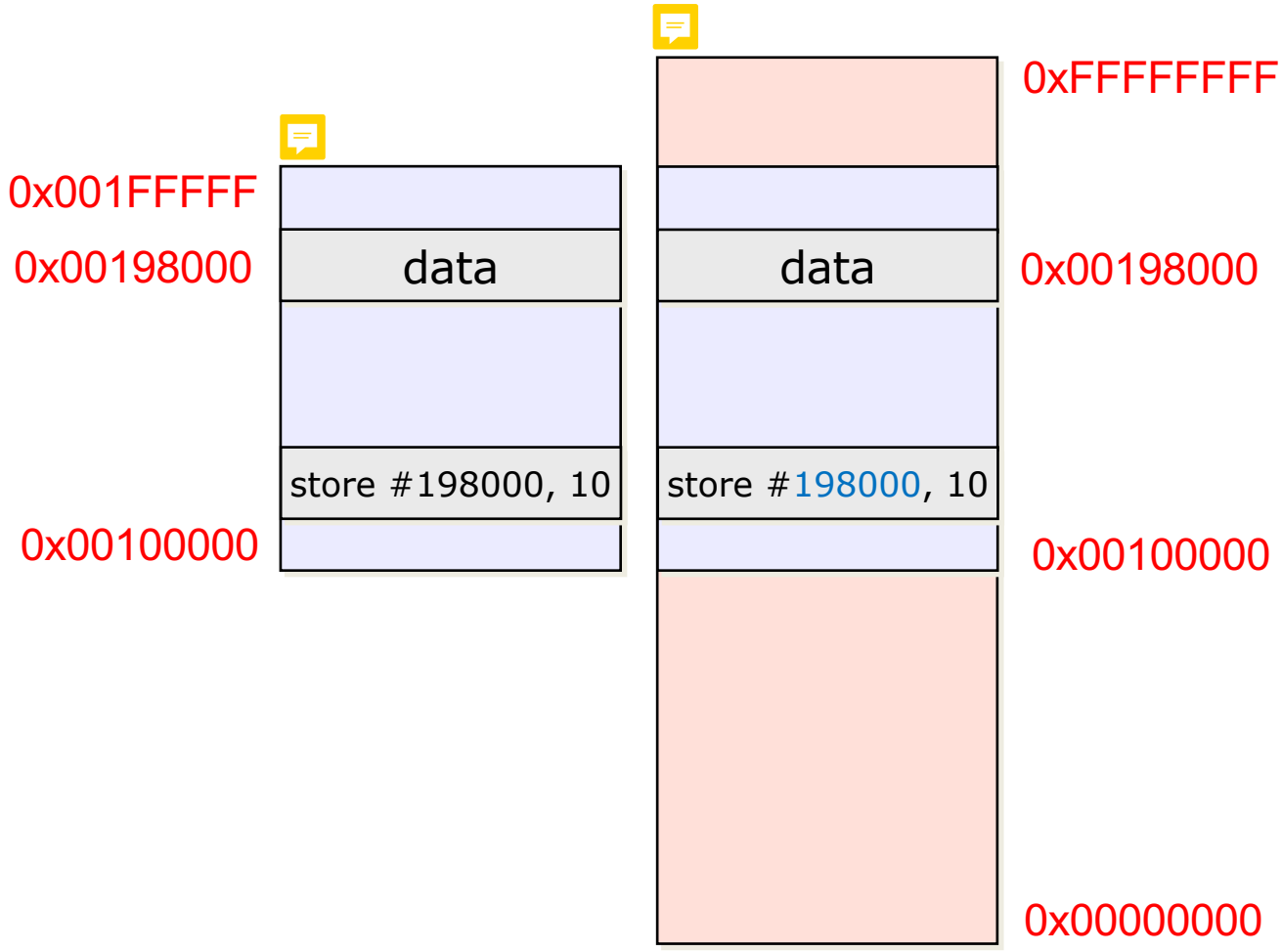
- Address binding of instructions and data to memory addresses can happen at three different stages
  - ✓ Compile time
  - ✓ Load time
  - ✓ Execution time



Binding of Memory Address

- Compile time

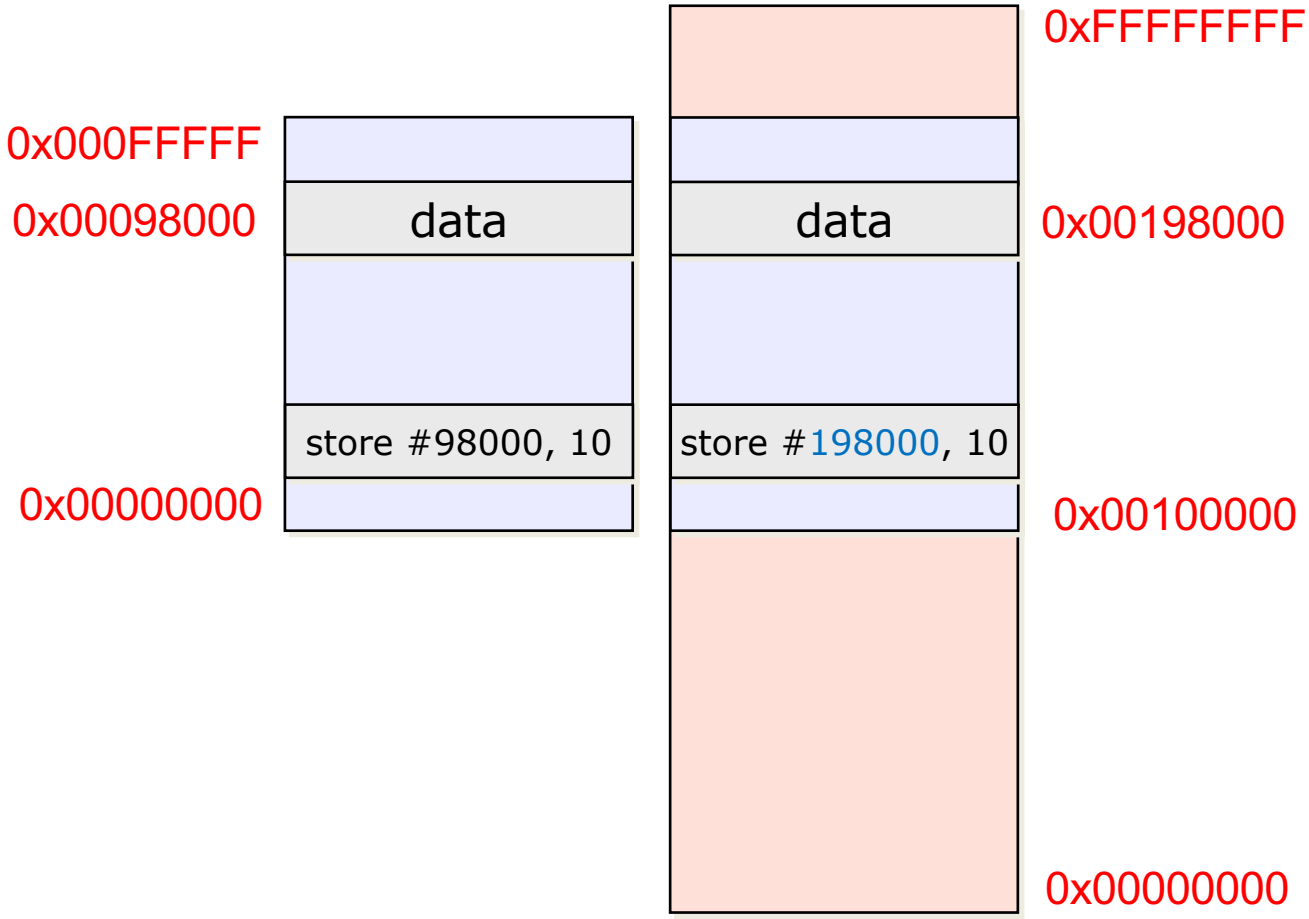
```
void func( )
{
   int data;

   …
   data = 10;
   …
}
```



Binding of Memory Address

- Load time

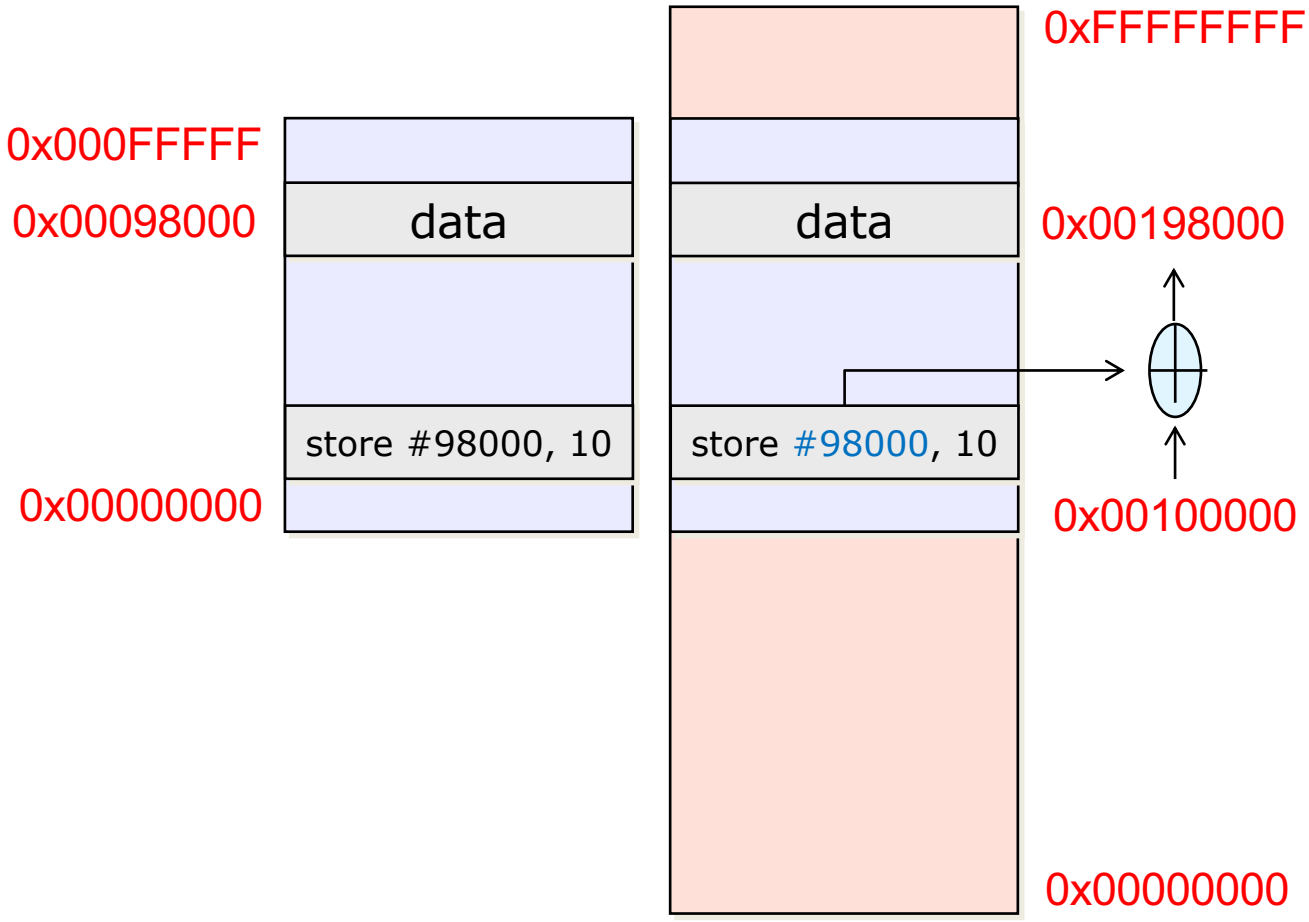
```
void func( )
{
   int data;

   …
   data = 10;
   …
}
```



Binding of Memory Address

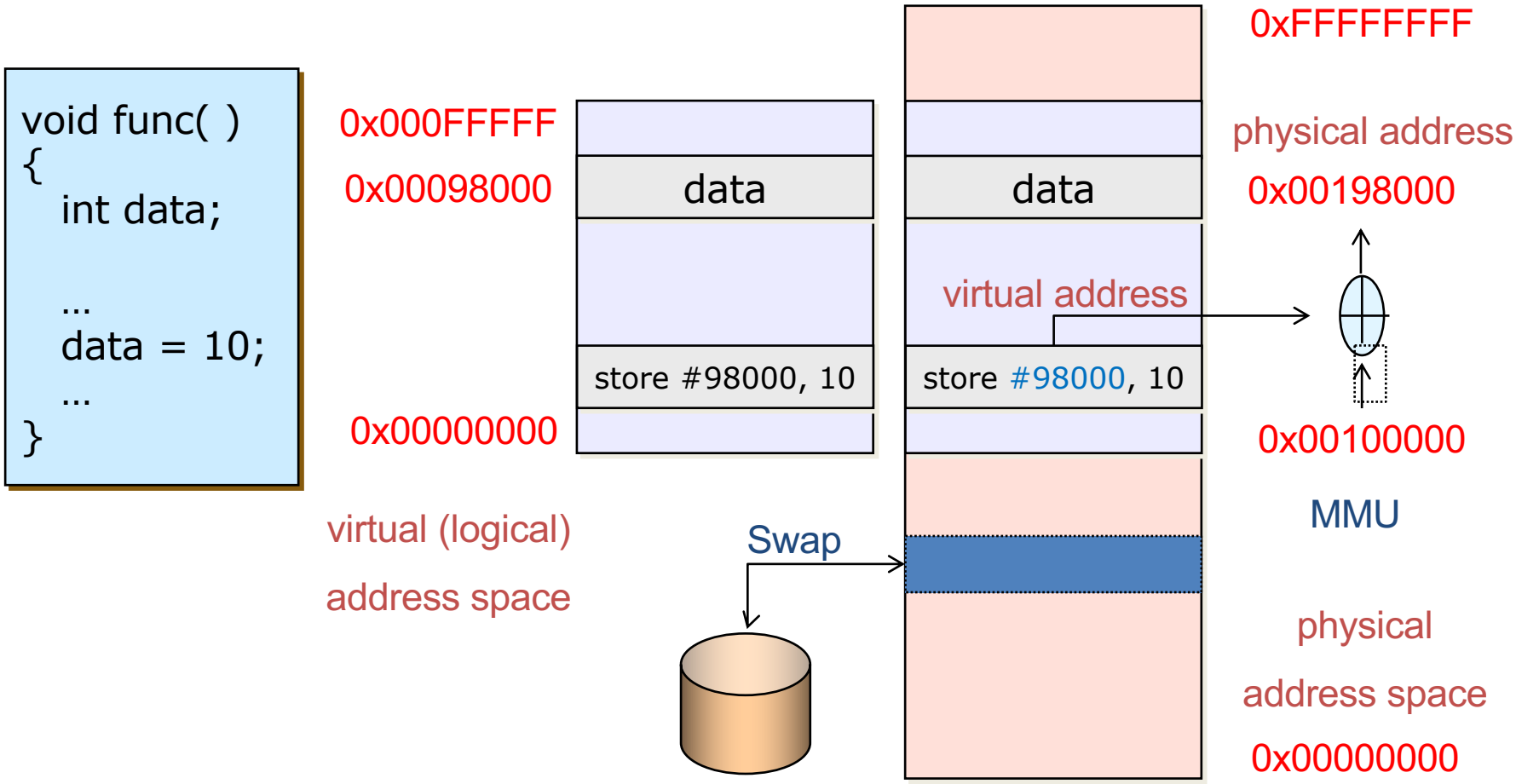
- Execution time

```
void func( )
{
   int data;

   …
   data = 10;
   …
}
```



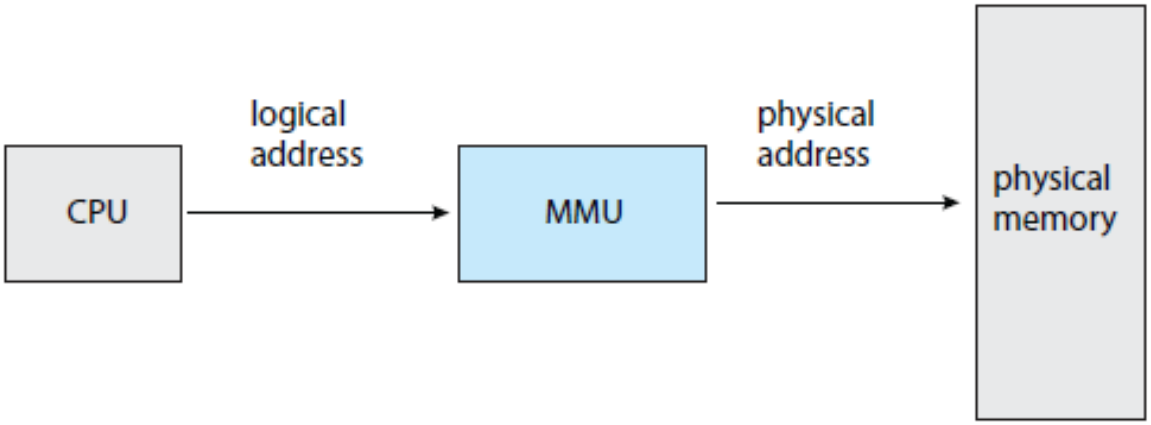
Virtual Memory Management

- Address mapping



Memory-Management Unit (MMU)

- Hardware device that maps logical (virtual) to physical address
  - ✓ in CPU
- The user program deals with *logical* addresses
  - ✓ It never sees the *real* physical addresses



Logical (Virtual) Address Space

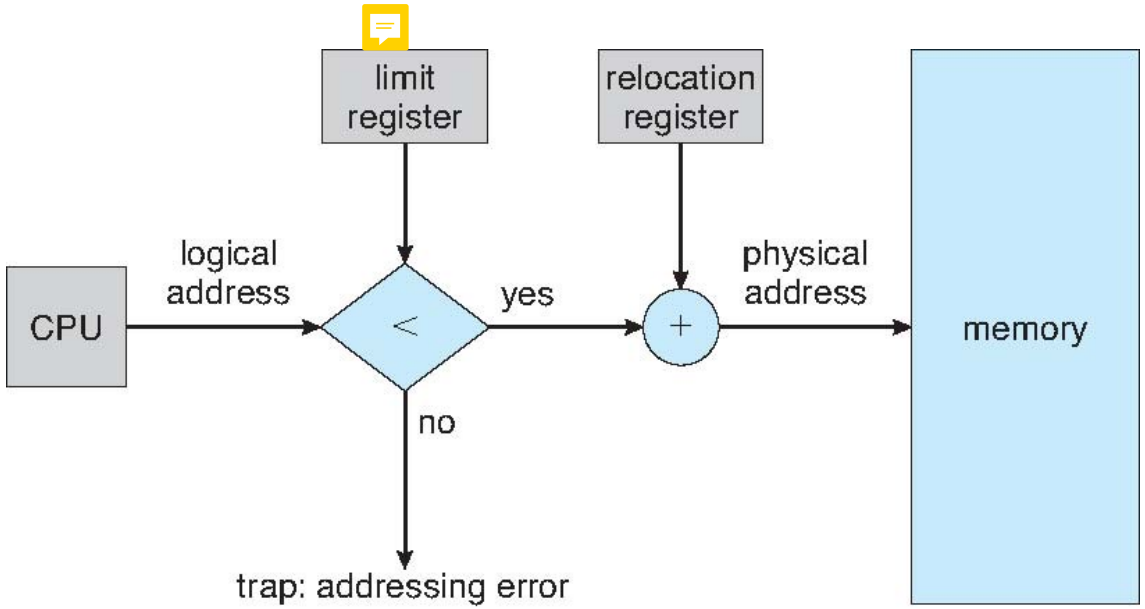
- Example

```
#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?

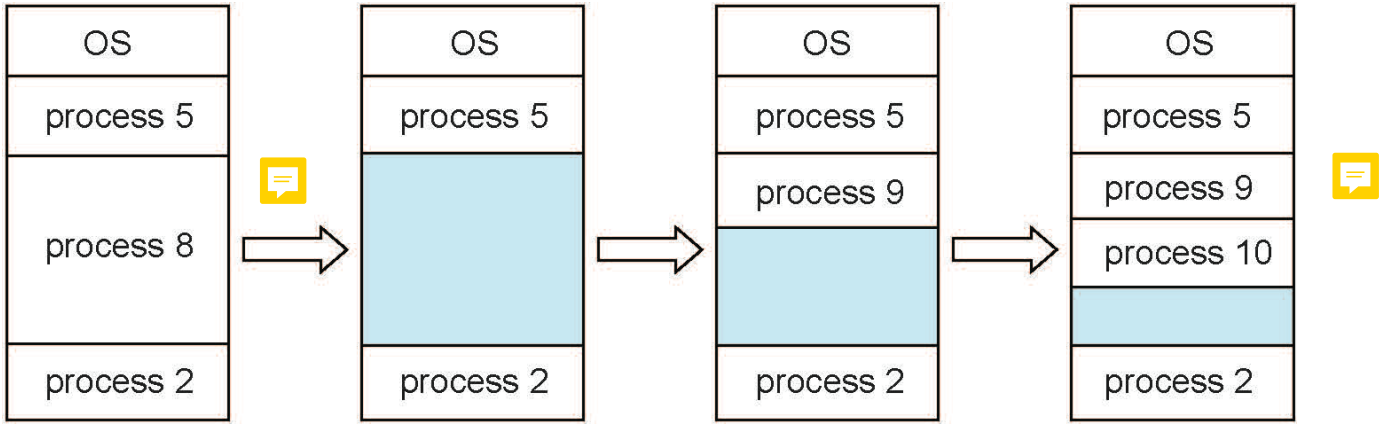
Contiguous Allocation

- Logically contiguous and physically contiguous
- Address translation
  - ✓ Physical address = logical address + relocation register
- Memory protection
  - ✓ Limit register



Contiguous Allocation

- 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



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

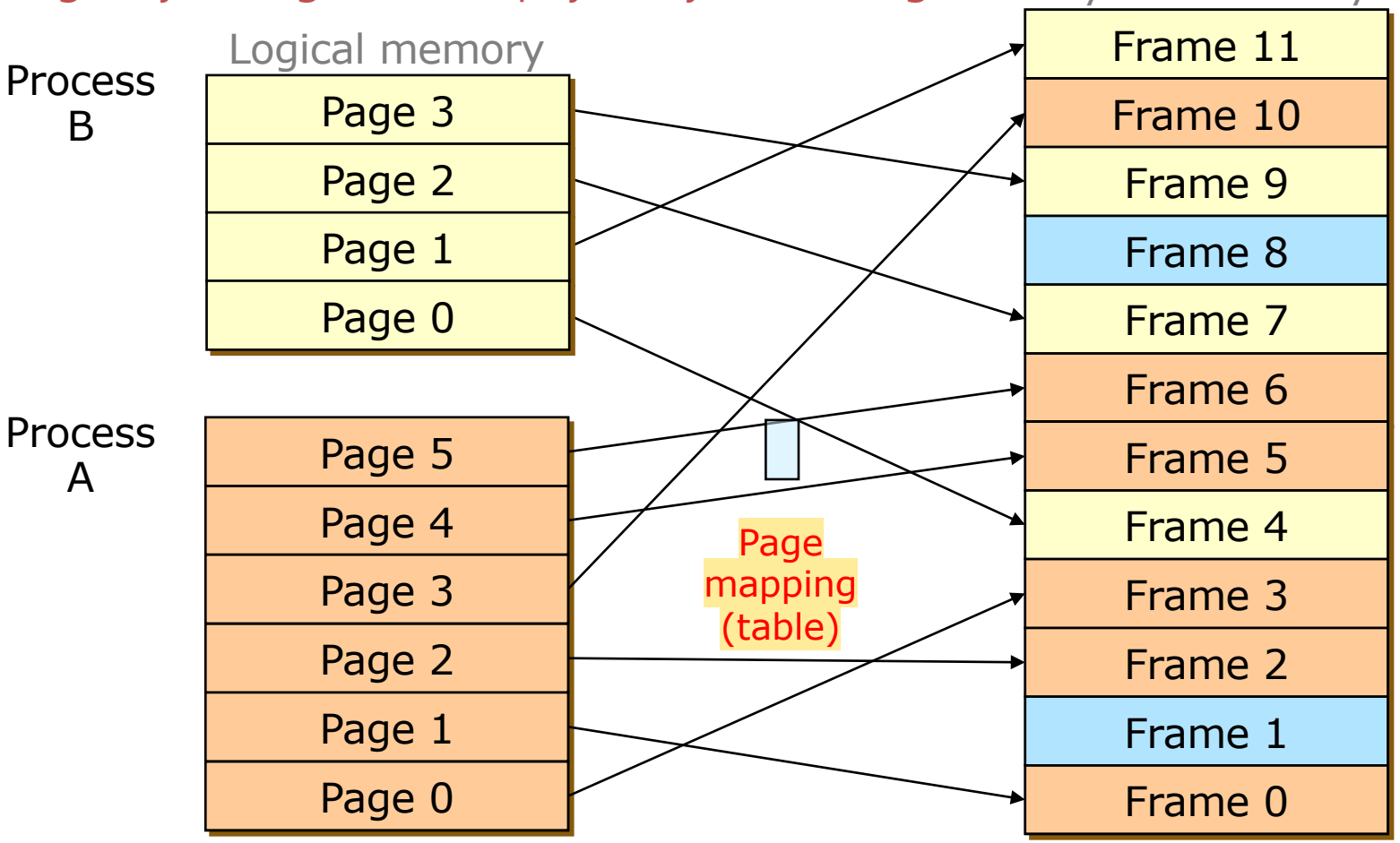
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

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

- Logically contiguous but physically not contiguous



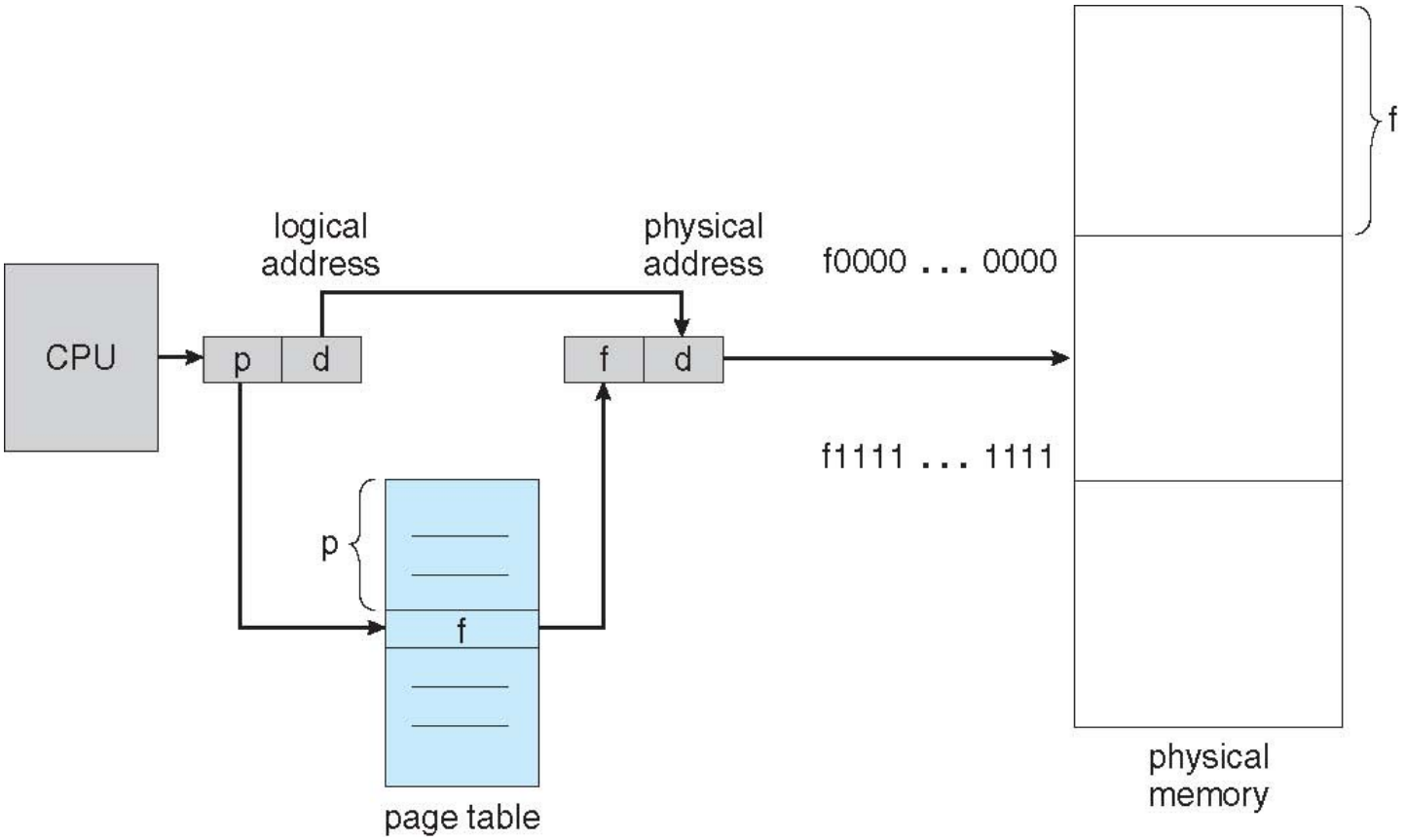
Address Translation

- Translating addresses
  - ✓ A logical address has two parts:
    <page number::offset> i.e., (p, d)
  - ✓ Page number is an index into a page table
  - ✓ Page table determines frame number
  - ✓ Physical address is <frame number::offset> i.e., (f, d)

- 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

- No external fragmentation but internal fragmentation

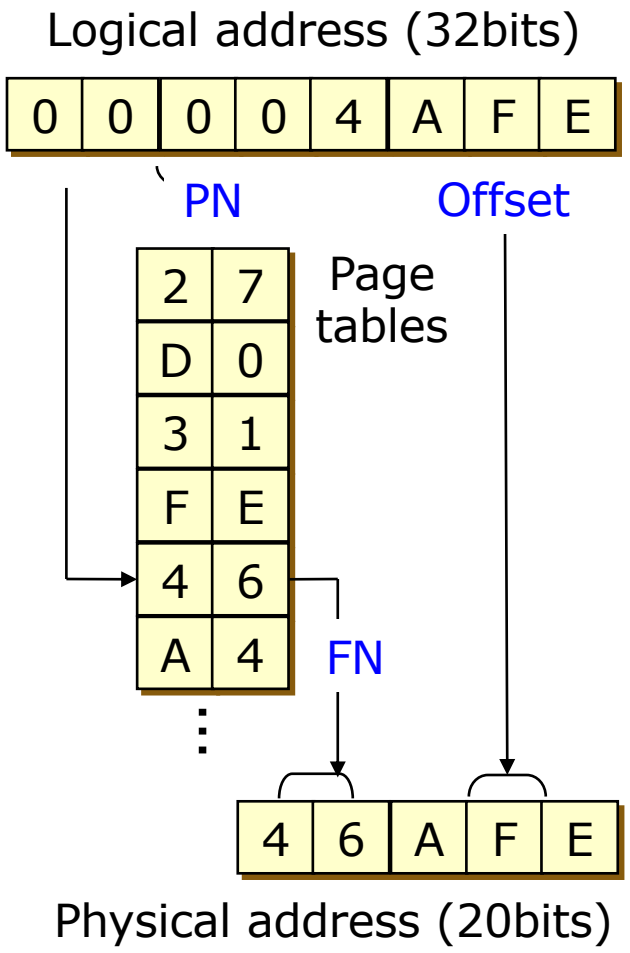
Address Translation Architecture



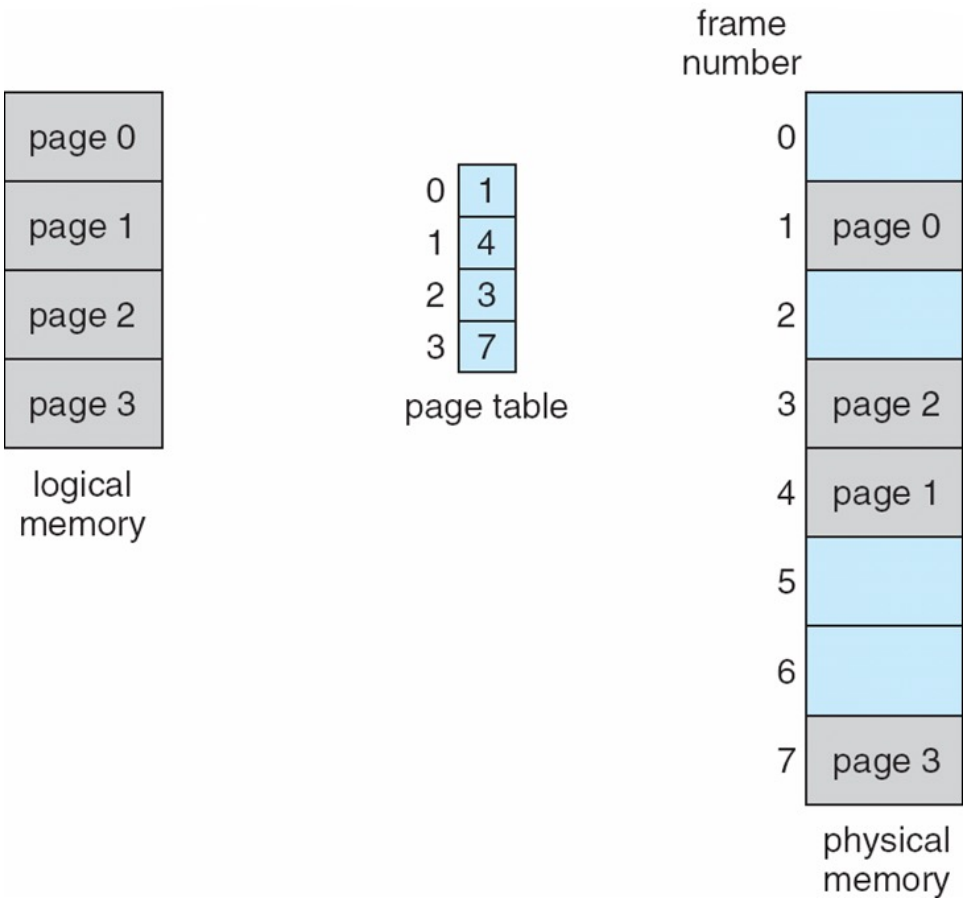
Paging Example

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

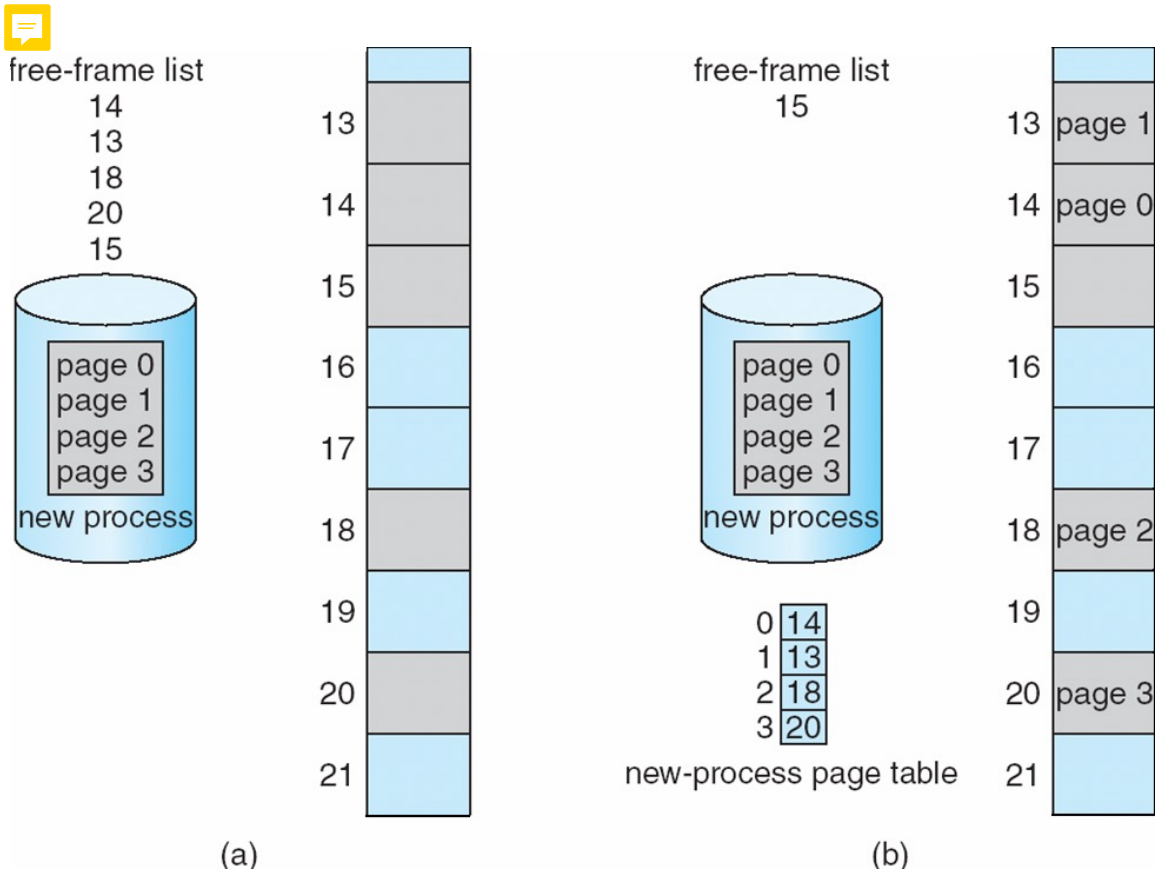
- Offset: 12 bits
- Page Number: 20 bits
- Page table entries: 2^{20}



Paging Example



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)

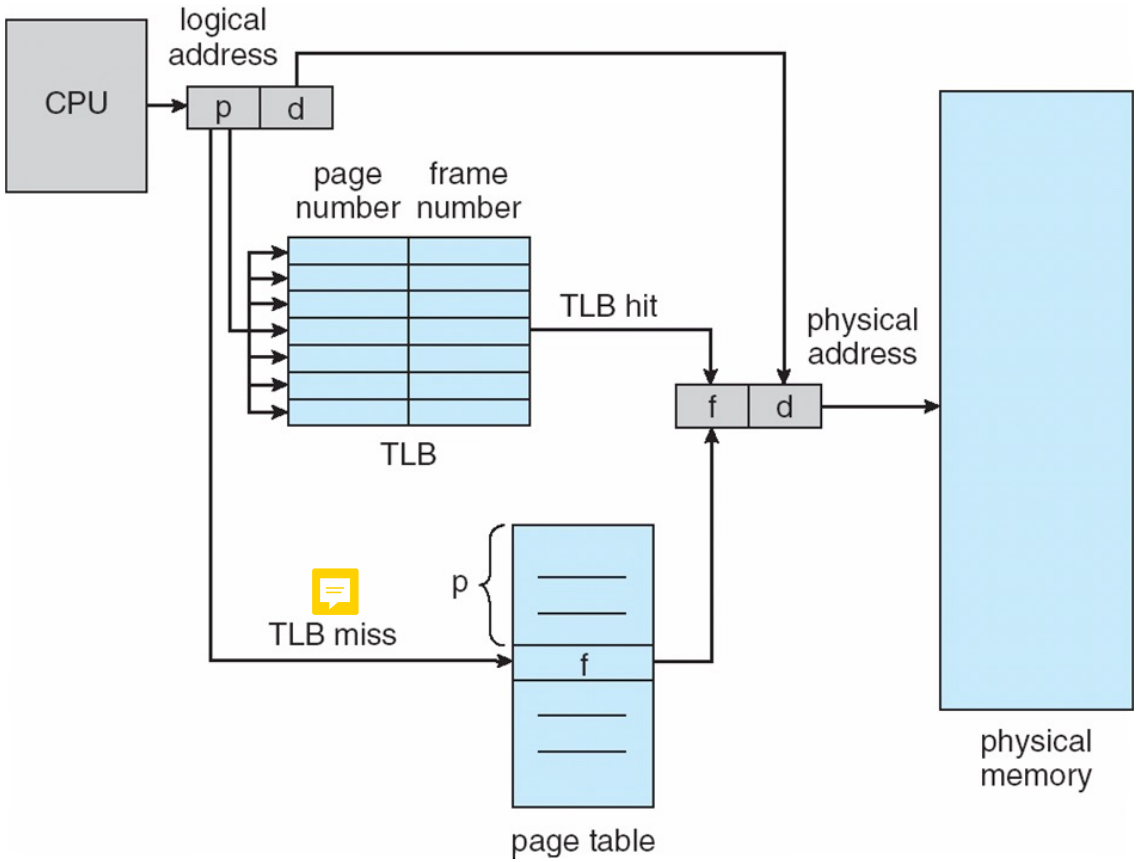
Implementation of Page Table

- Page table is kept in main memory
  - ✓ *Page-table base register (*PTBR) points to the page table
  - ✓ *Page-table length register* (PTLR) indicates size of the page table

- In this scheme every data/instruction access requires two memory accesses
  - ✓ One for the page table and one for the data/instruction

- 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)

Paging Hardware with TLB



Associative Memory

- Associative memory – parallel search

Page #	Frame #

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

- *Cf) CAM (Content Addressable Memory)*

TLB

- 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

- 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

- 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)

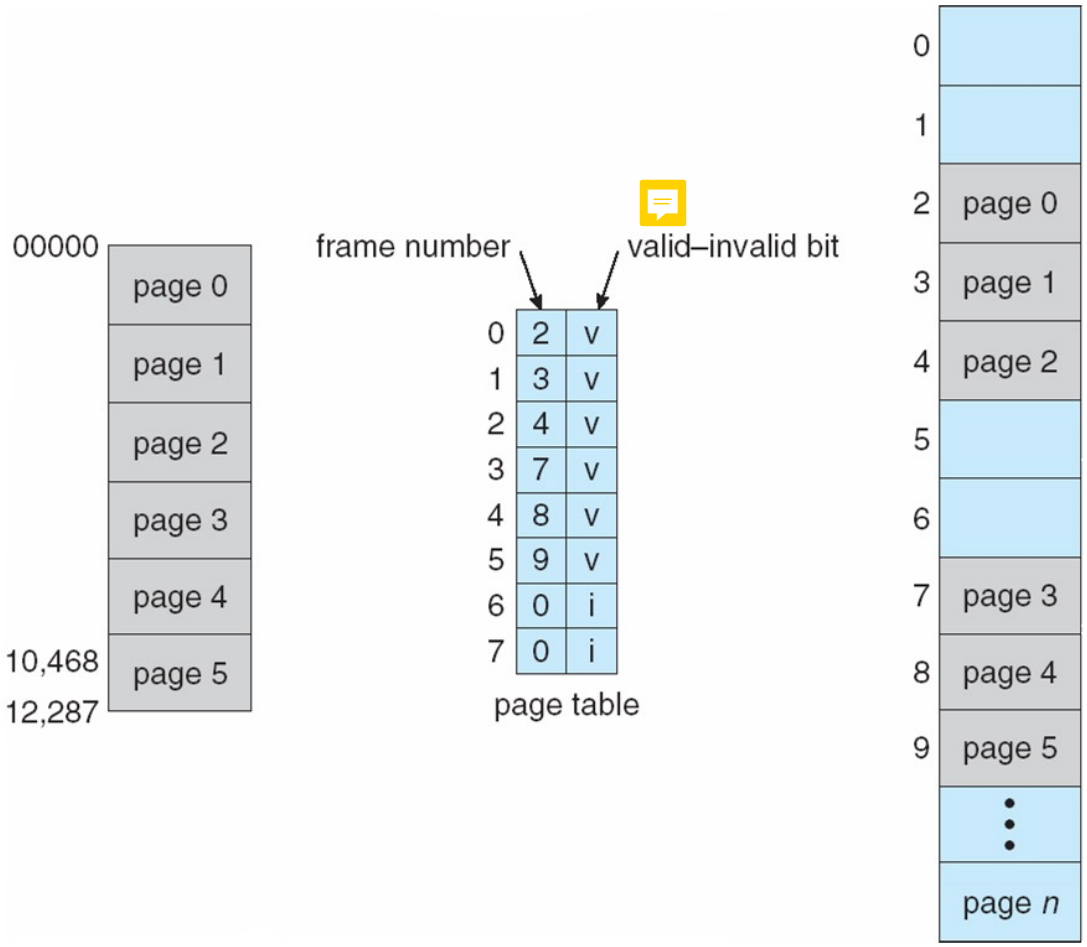
Effective Access Time

- Effective Access Time (EAT)

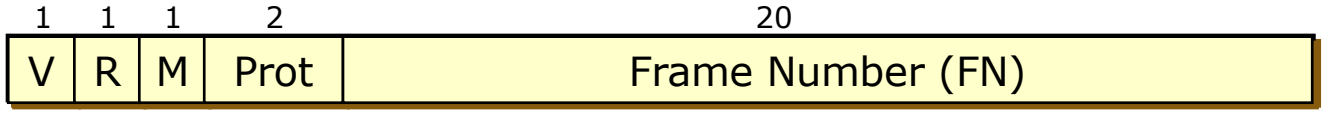
$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha$$

  - ✓ Associative Lookup: ε time unit
  - ✓ Hit ratio: α
  - ✓ Consider realistic values: α = 99%, ε = 20ns, 100ns for memory access
    - ➢ EAT = (100+20)x0.99 + (200+20)x0.01 = 121ns

Main Memory Protection



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

- 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

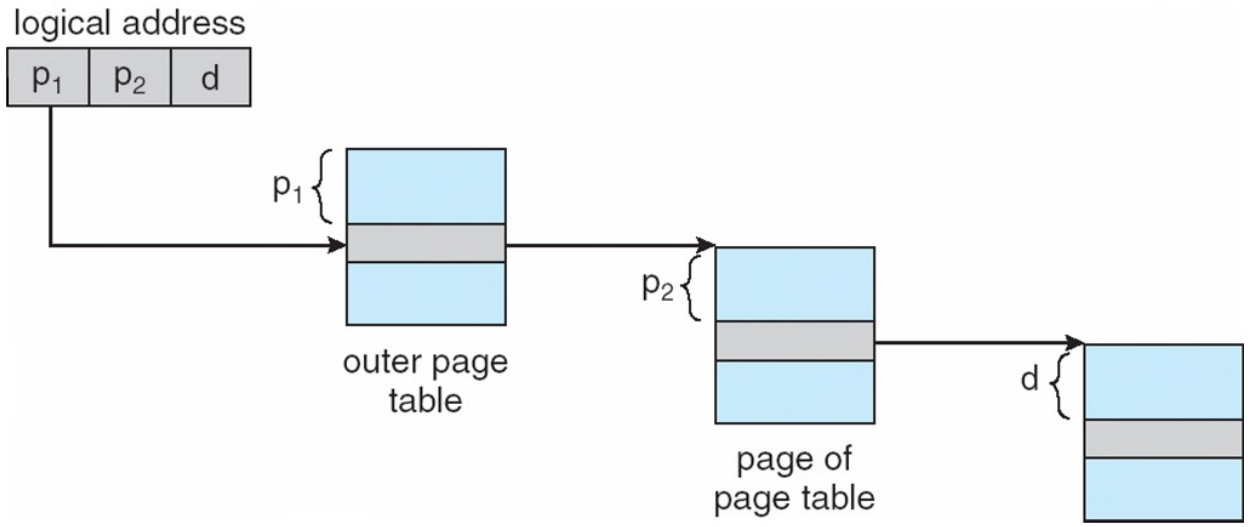
  - ✓ Hierarchical paging
  - ✓ Hashed page table
  - ✓ Inverted page table

Hierarchical Page Tables

- Break up the logical address space into multiple page tables
  - ✓ Two-level paging example
    - ➢ Logical Address (4KB page)

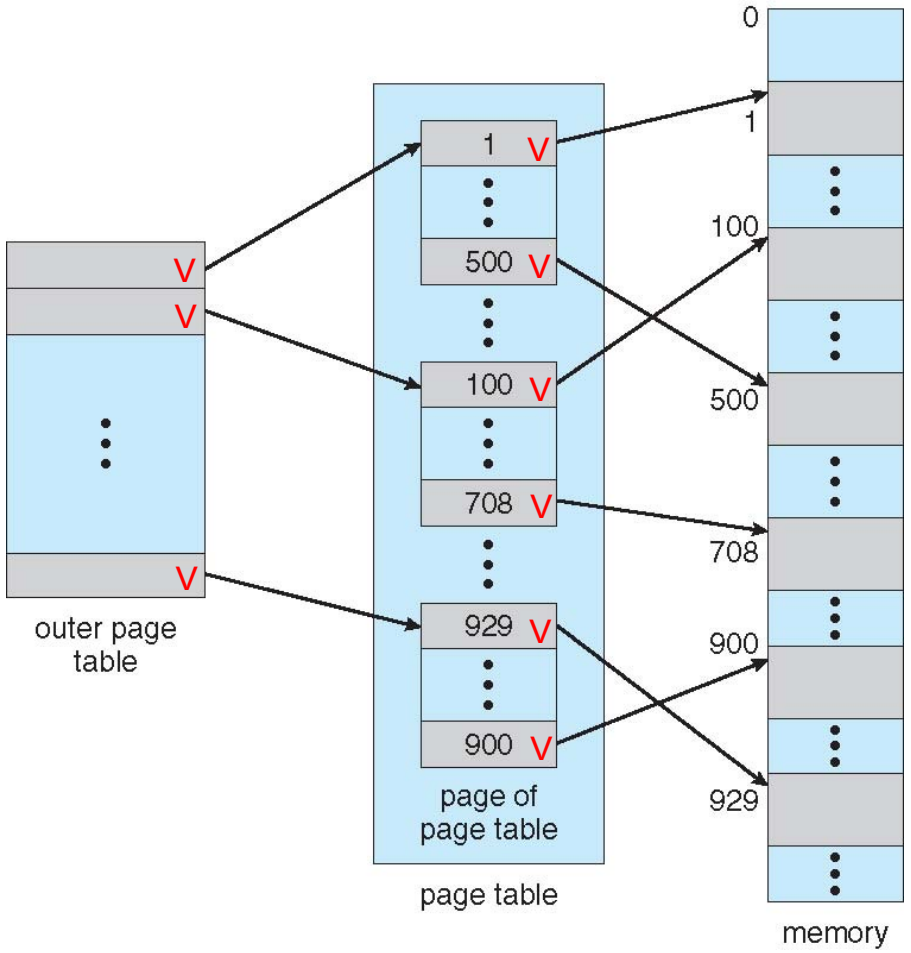
page number		page offset
p_1	p_2	d
1	1	1
0	0	2

    - ➢ Address translation



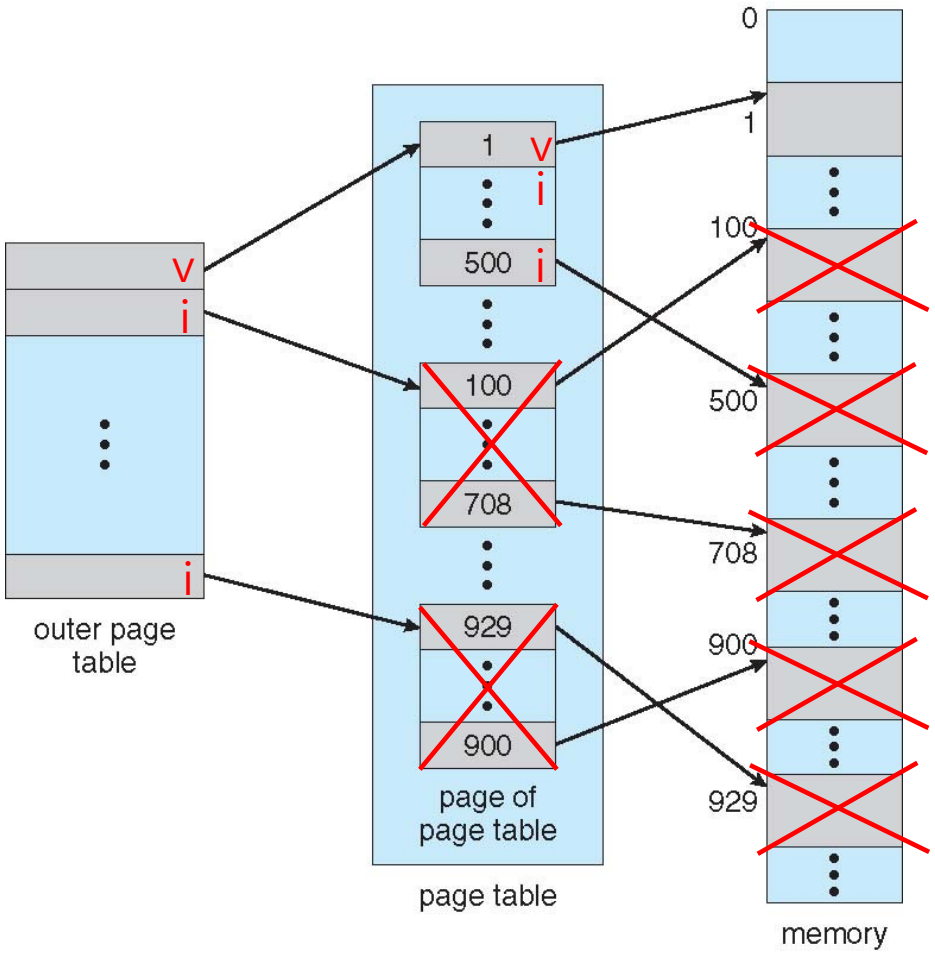
Hierarchical Page Tables

- Two-level paging example
  - ✓ Case 1) all the pages are used
  - ✓ Outer page table
    - ➢ 2^{10} entries
    - ➢ 4B x 2^{10} = 4KB
  - ✓ Page table
    - ➢ 2^{10} entries x 2^{10}
    - ➢ 4B x 2^{10} x 2^{10} = 4MB
- Single-level paging
  - ✓ (p, d): p=20bits
  - ✓ Page table
    - ➢ 2^{20} entries x 1
    - ➢ 4B x 2^{20} = 4MB

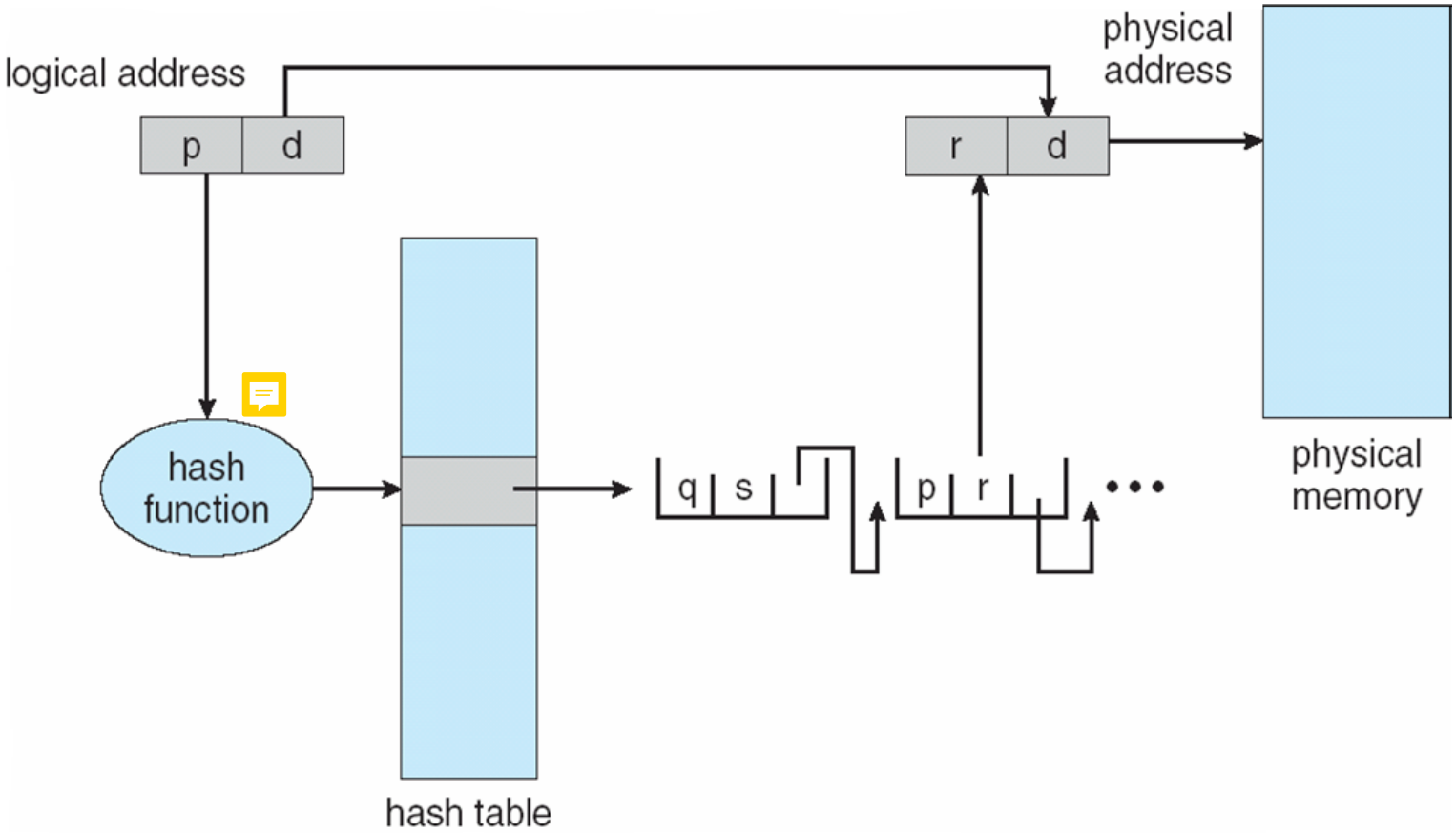


Hierarchical Page Tables

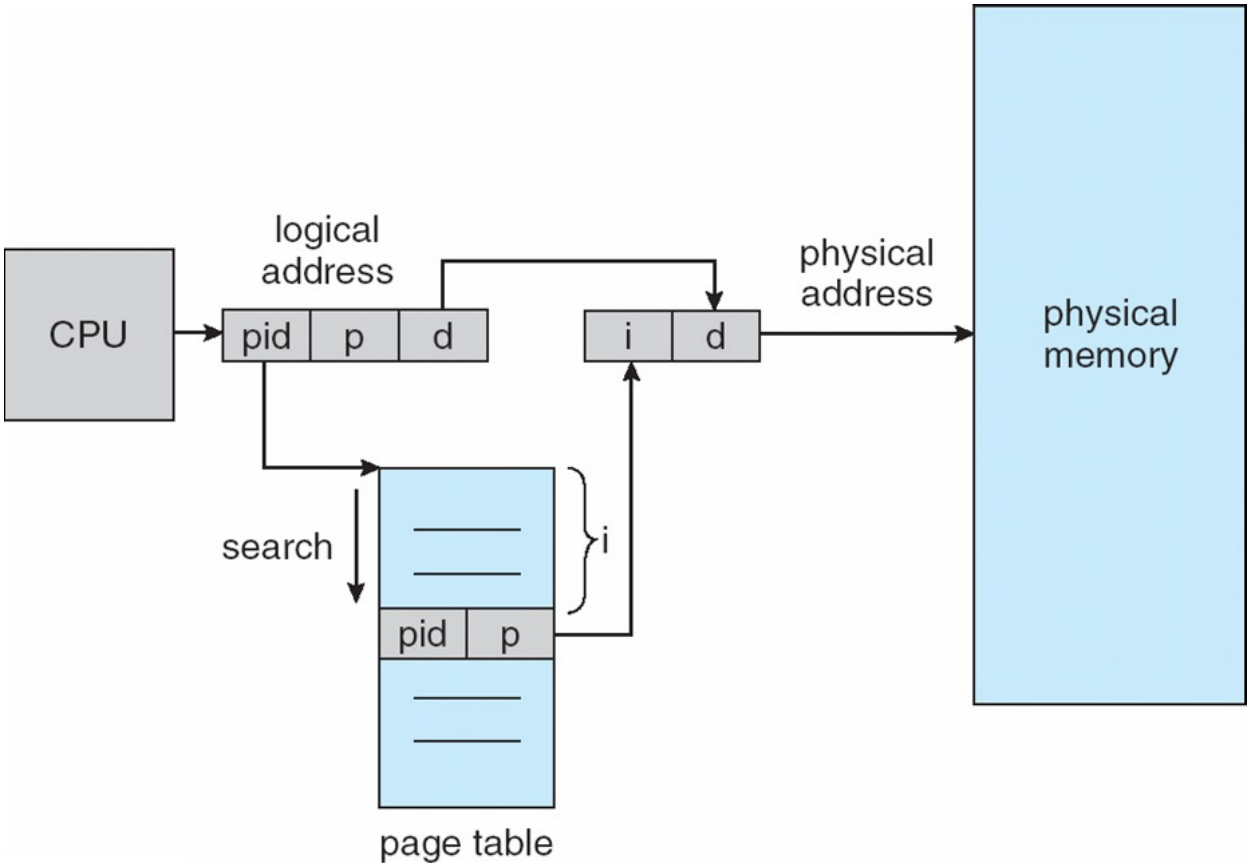
- Two-level paging example
  - ✓ Case 2) Only one page is used
  - ✓ Outer page table
    - 2^{10} entries
    - 4B x 2^{10} = 4KB
  - ✓ Page table
    - 2^{10} entries x 1
    - 4B x 2^{10} x 1 = 4KB



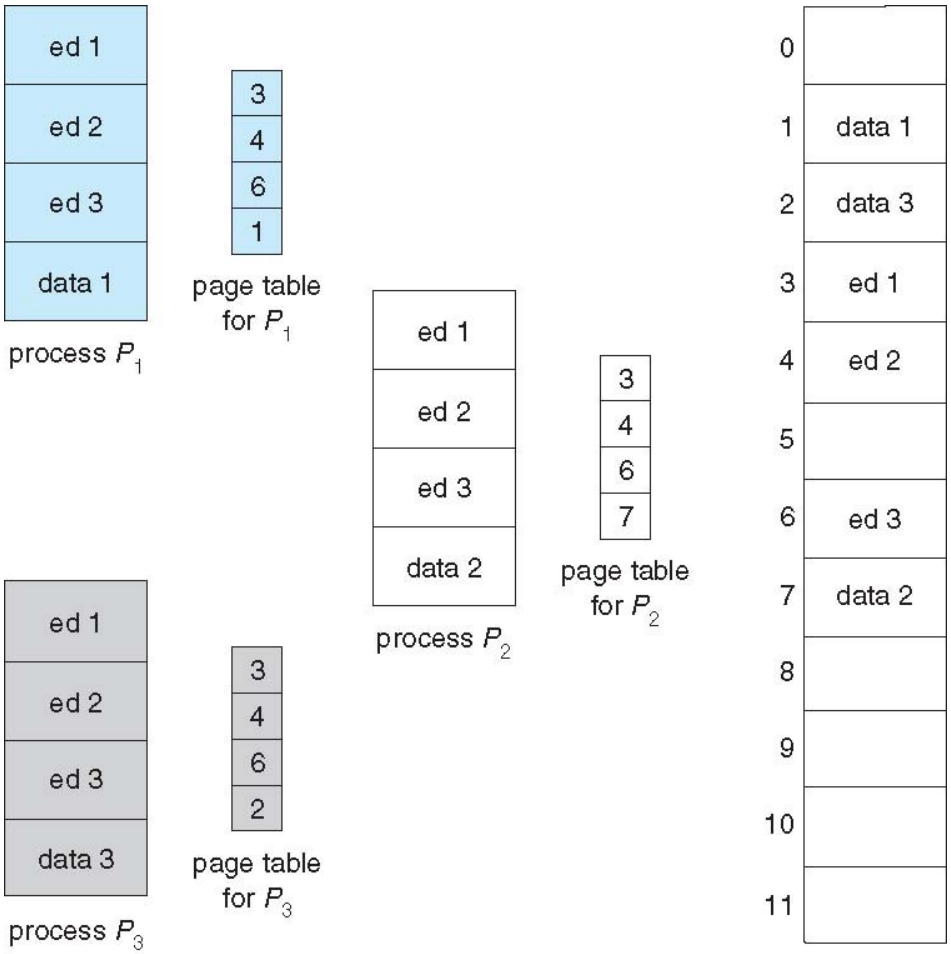
Hashed Page Tables



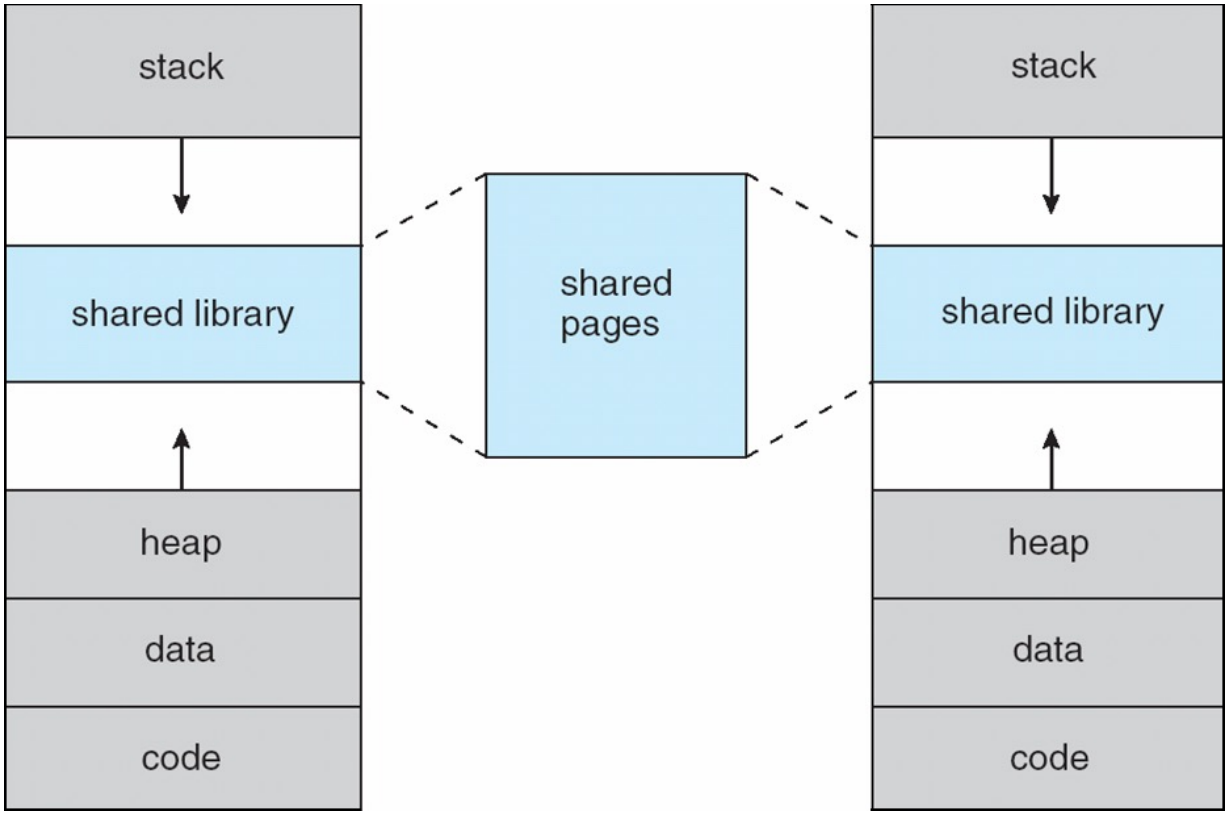
Invented Page Table



Shared Pages Example



Shared Pages Example



Advantages of Paging

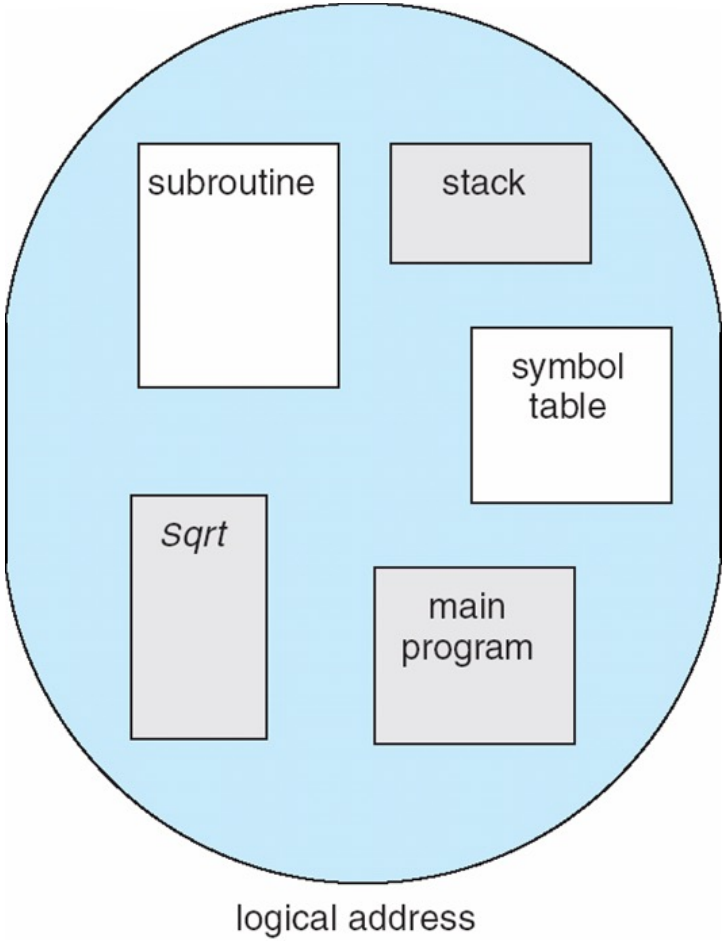
- Easy to allocate physical memory
  - ✓ Physical memory is allocated from free list of frames
  - ✓ To allocate a frame, just remove it from its free list
- No external fragmentation
- Easy to “page out” chunks of a program
  - ✓ All chunks are the same size (page size)
  - ✓ Use valid bit to detect 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
    - ➢ Hierarchical page tables, hashed page table, inverted page table, etc.

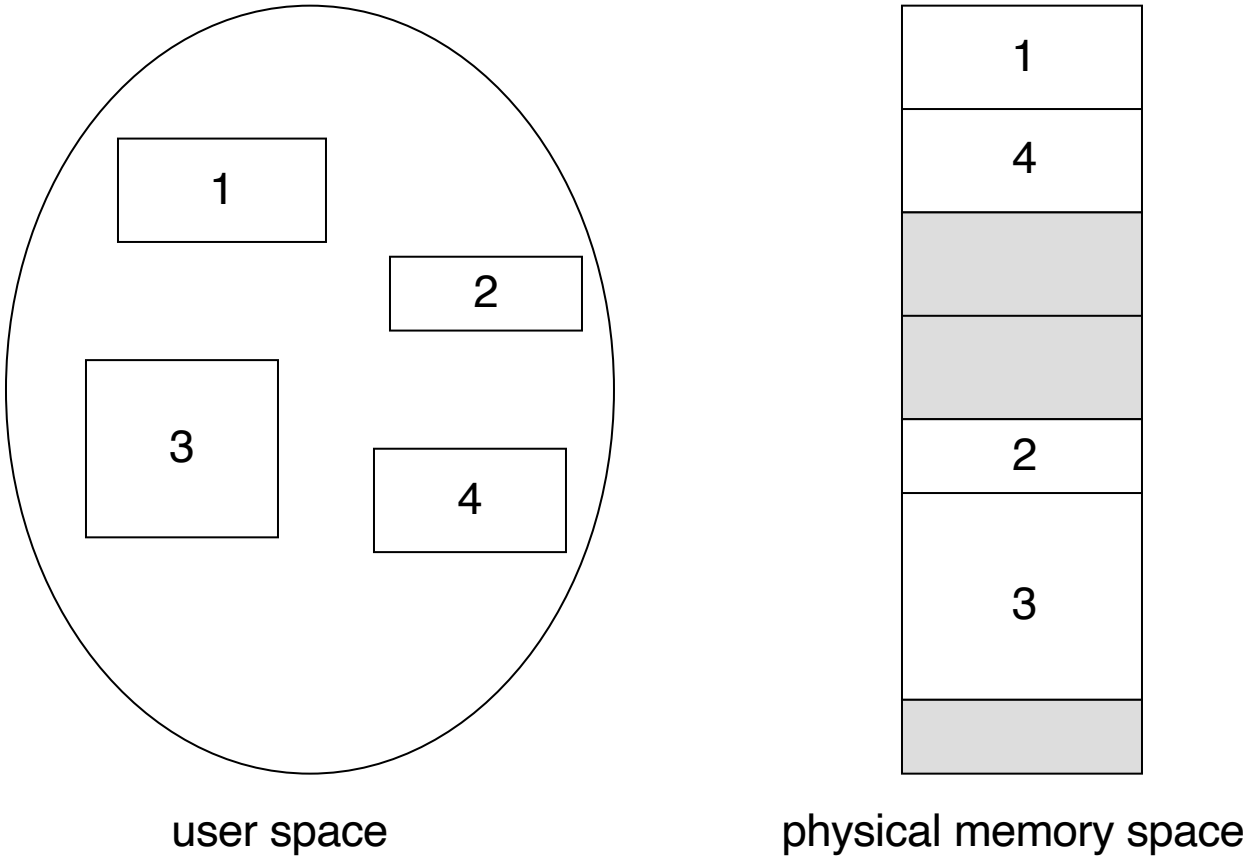
Segmentation

- User's view of a program
- A program is a collection of segments

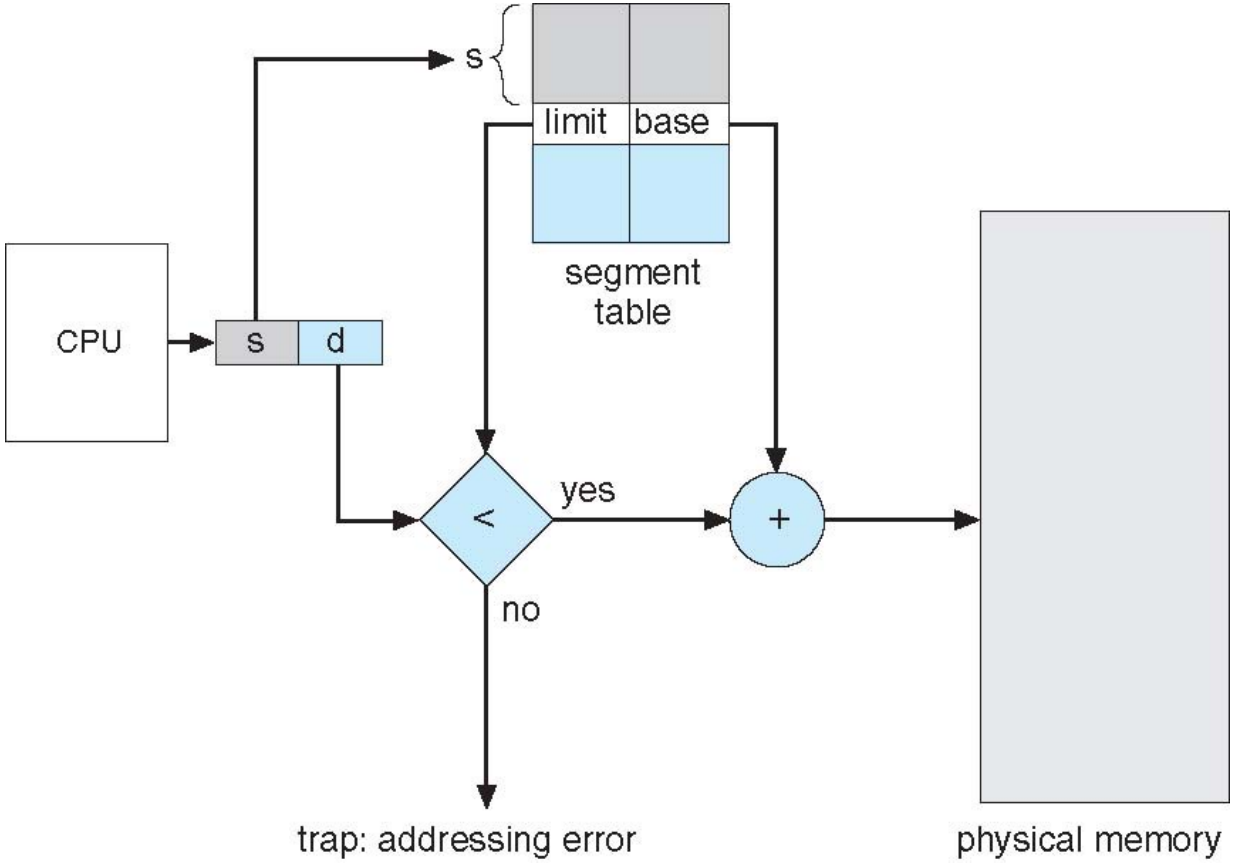


Segmentation

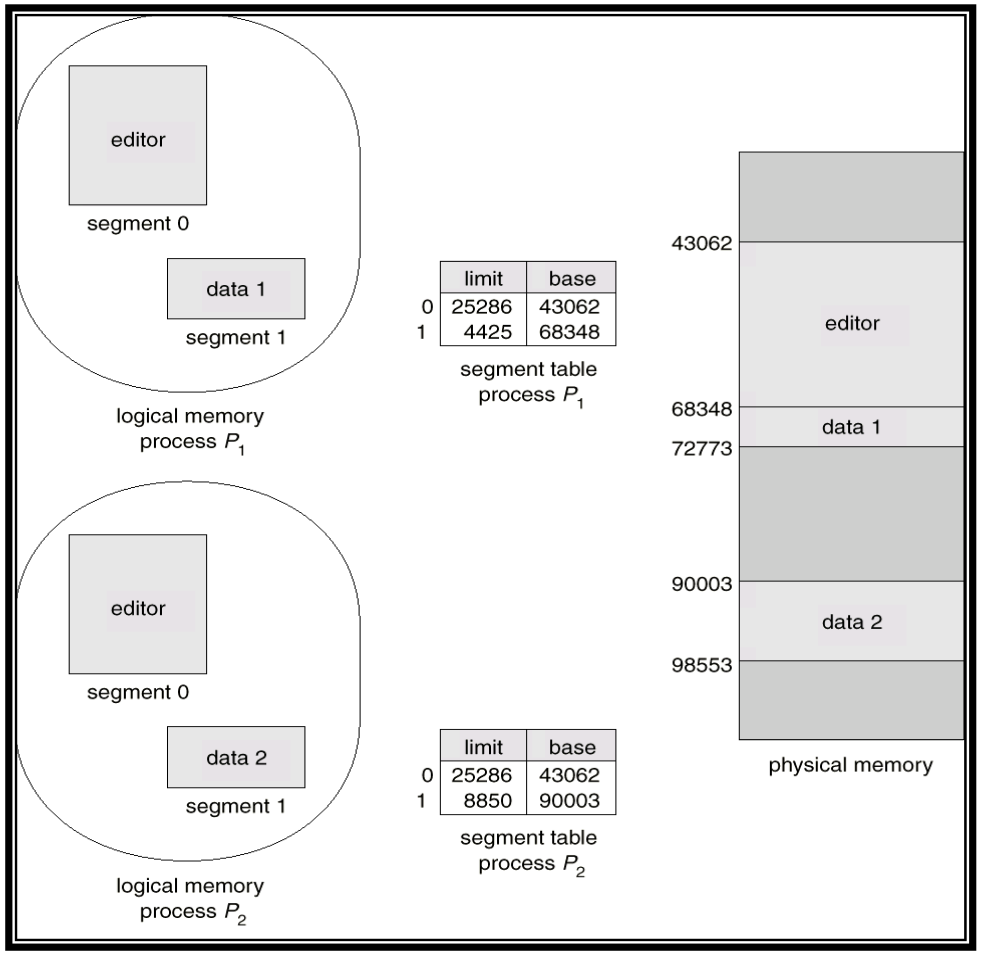
- Logical view of segmentation



Segmentation Hardware



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

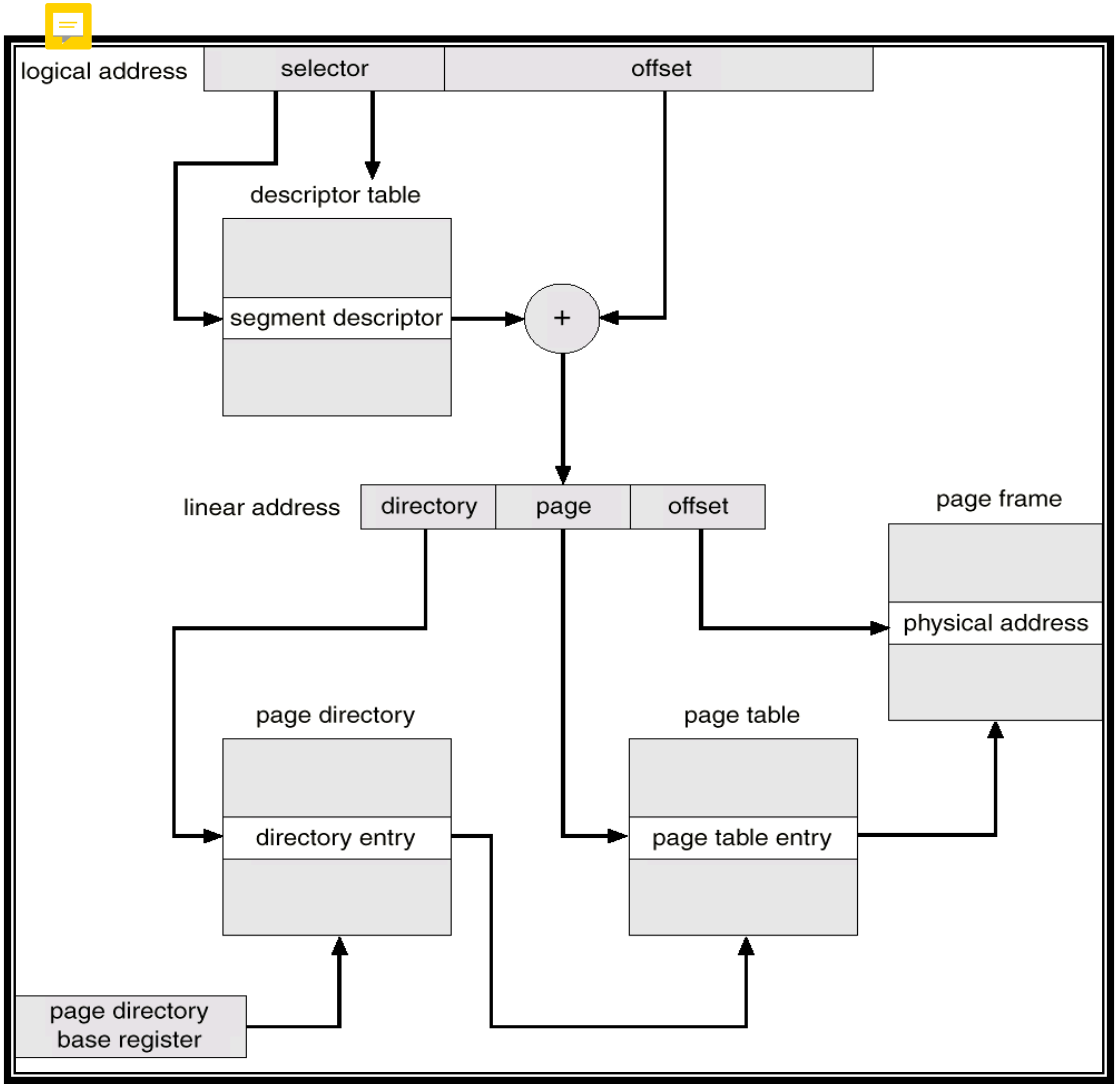
Paging vs. Segmentation

- 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

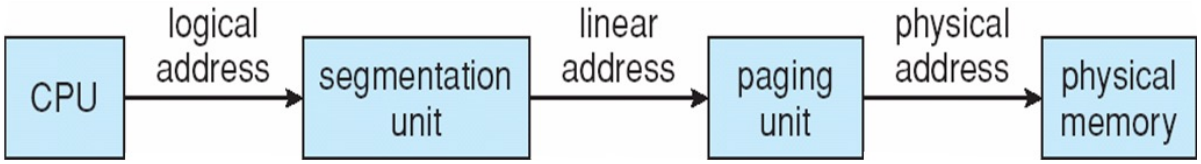
- Combine segmentation and paging
  - ✓ Use segments to manage logically related units
    - Code, data, heap, etc.
    - Segments vary in size, but usually large (multiple pages)
  - ✓ Use pages to partition segments into fixed size chunks
    - Makes segments easier to manage with in physical memory
    - 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 in Pentium

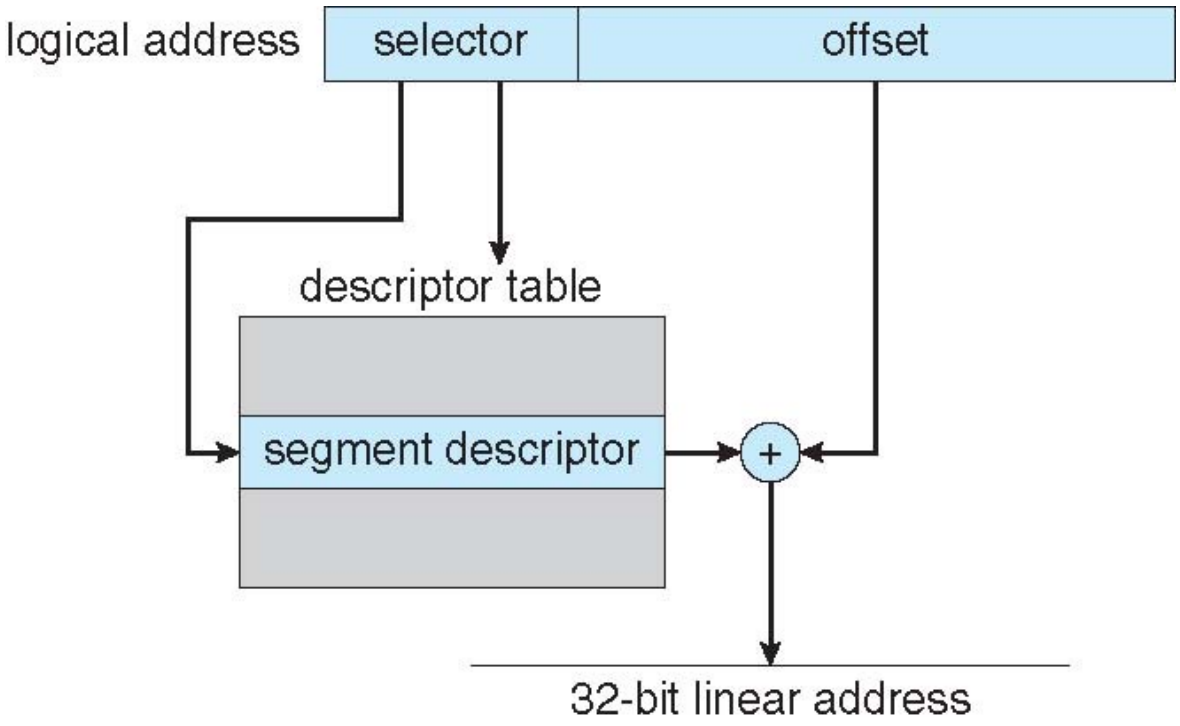


Address Translation in IA-32

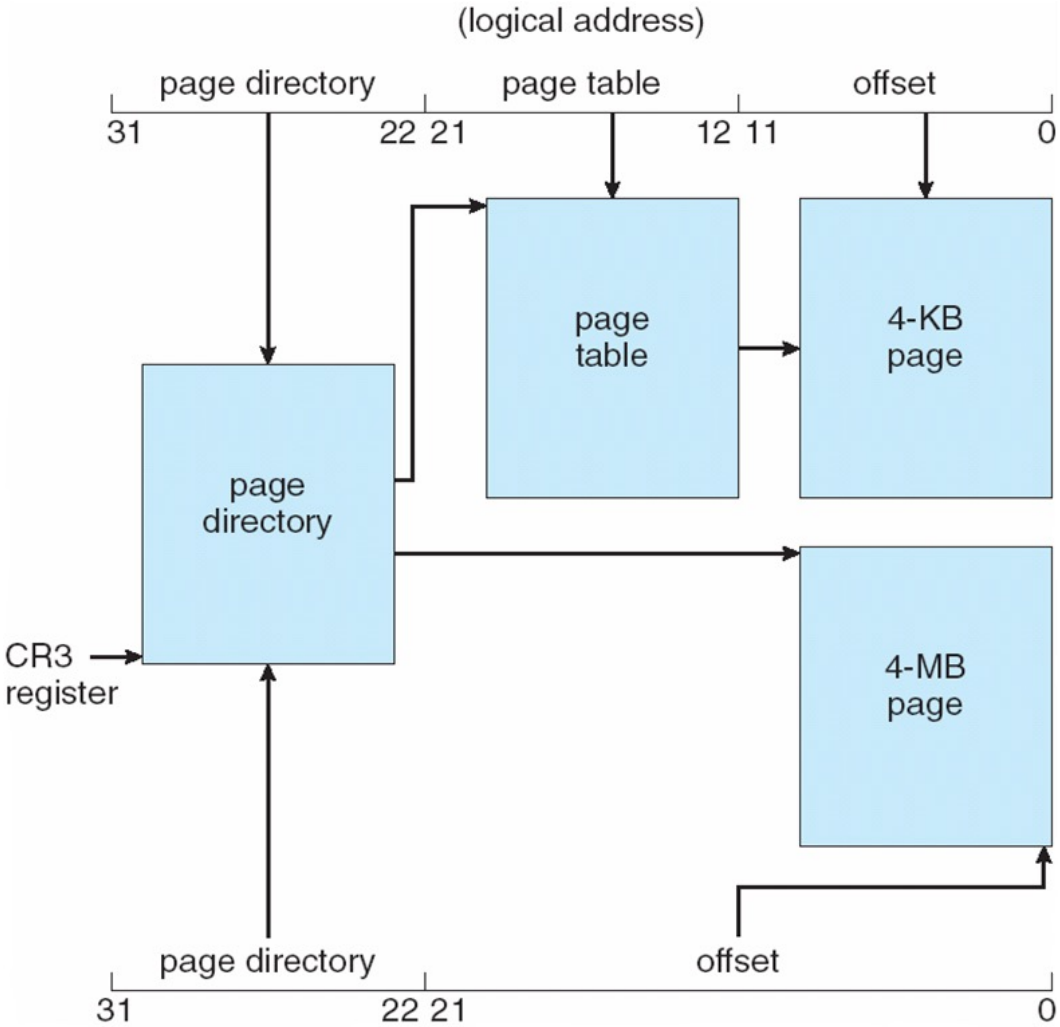
- IA-32 segmentation
- IA-32 paging
  - ✓ Two-level paging
  - ✓ Multiple page size
- IA-32 PAE (Page Address Extension)



IA-32 Segmentation



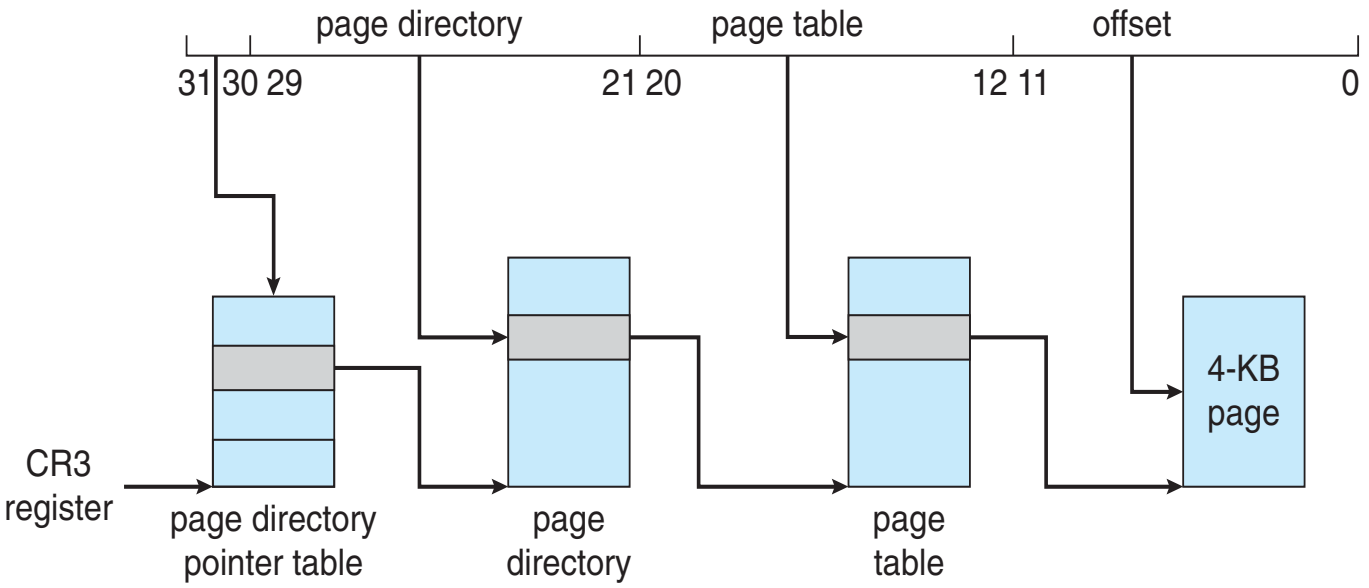
IA-32 Paging



Multiple page sizes

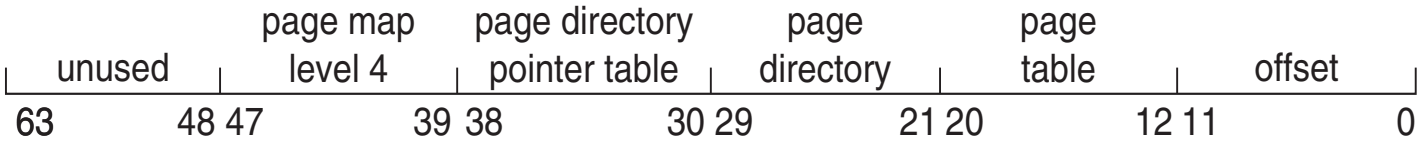
IA-32 PAE (Page Address Extensions)

- Support for access to more than 4GB of memory space
  - ✓ 3-level scheme
  - ✓ Top two bits refer to a page directory pointer table
  - ✓ Page-directory and page-table entries moved to 64-bits in size
  - ✓ Increasing address space to 36 bits (64GB of physical memory)



x86-64

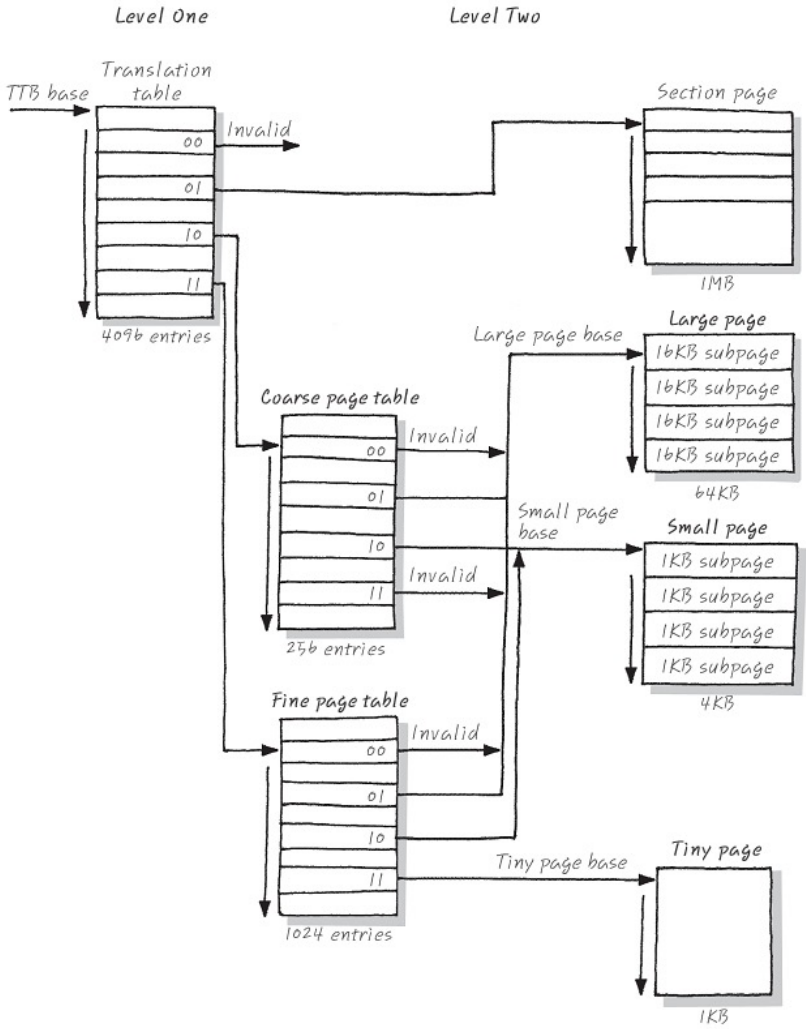
- AMD64 and IA-64 (Itanium)
  - ✓ 48-bits addressing
  - ✓ Multiple page sizes of 4KB, 2MB, 1GB
  - ✓ Four levels of paging hierarchy
  - ✓ Can also use PAE (up to 52-bits physical address)



Two Level Paging in ARM

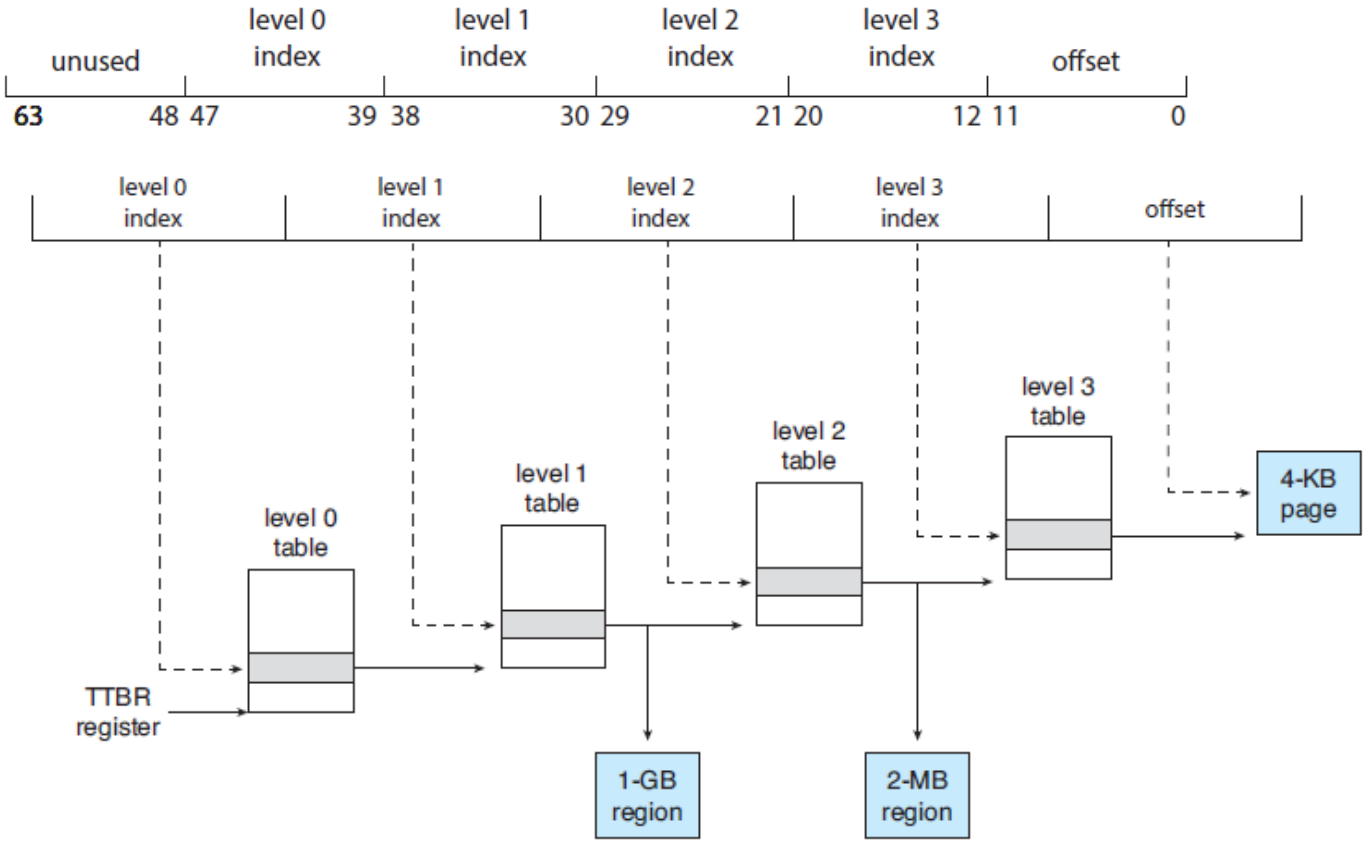
- Linear address space in AArch32
  - ✓ Translation table: Page table
  - ✓ TTB(Translation Table Base)
  - ✓ Section page (1MB)
  - ✓ Large page (64KB)
  - ✓ Small page (4KB)
  - ✓ Tiny page (1KB)

- http://recipes.egloos.com



ARMv8 (AArch64)

- Support for access to more than 4GB of memory space
  - ✓ four-level hierarchical paging



Thank You!
Q&A