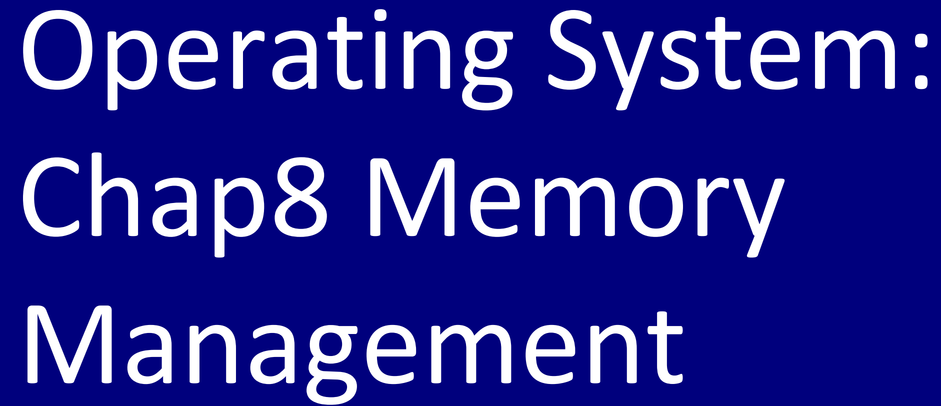


National Tsing-Hua University

2016, Fall Semester

Overview

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

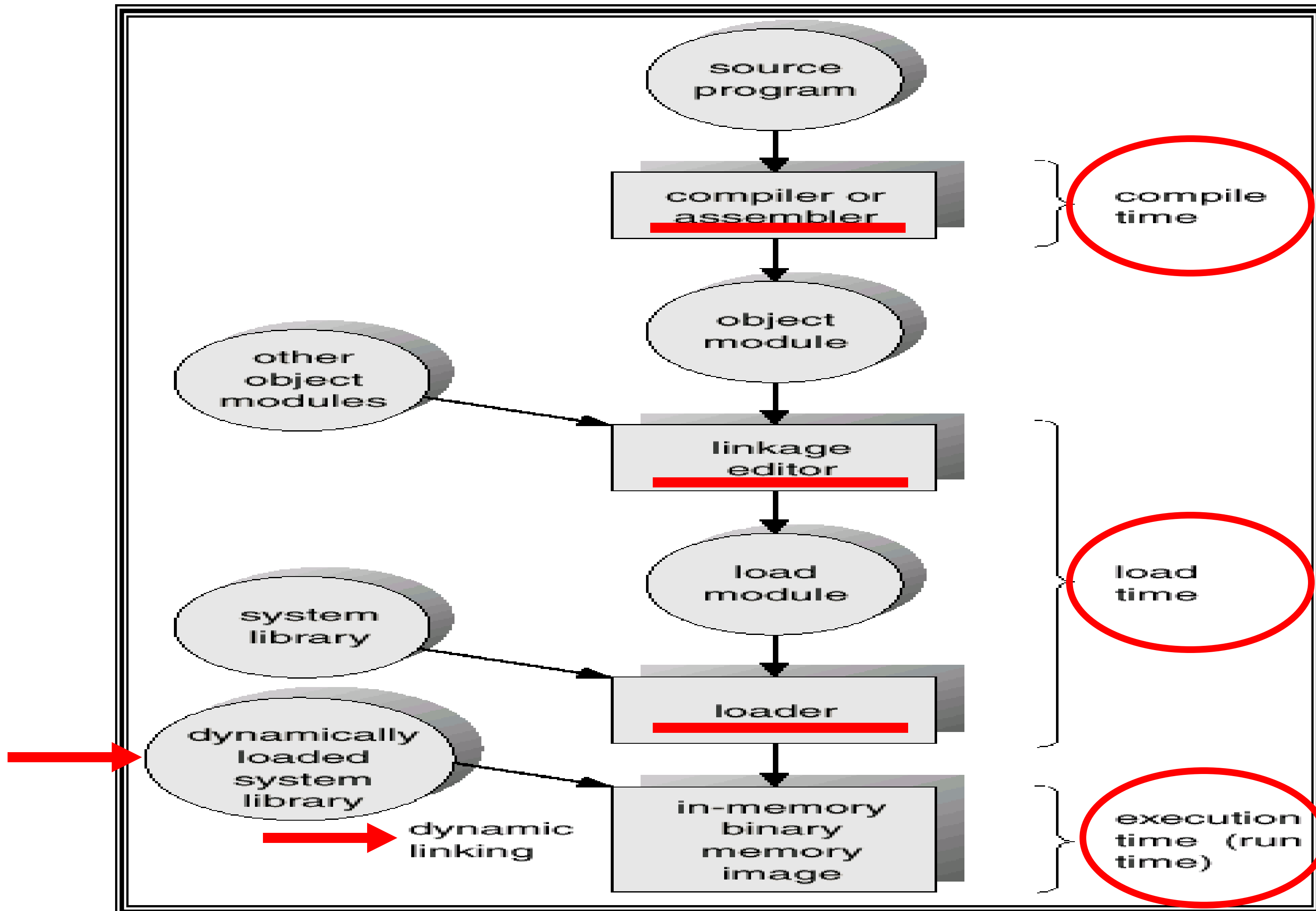
Background

- Main memory and registers are the only storage CPU can access directly
- Collection of processes are waiting on disk to be brought into memory and be executed
- Multiple programs are brought into memory to improve resource utilization and response time to users
- A process may be moved between disk and memory during its execution

Outline

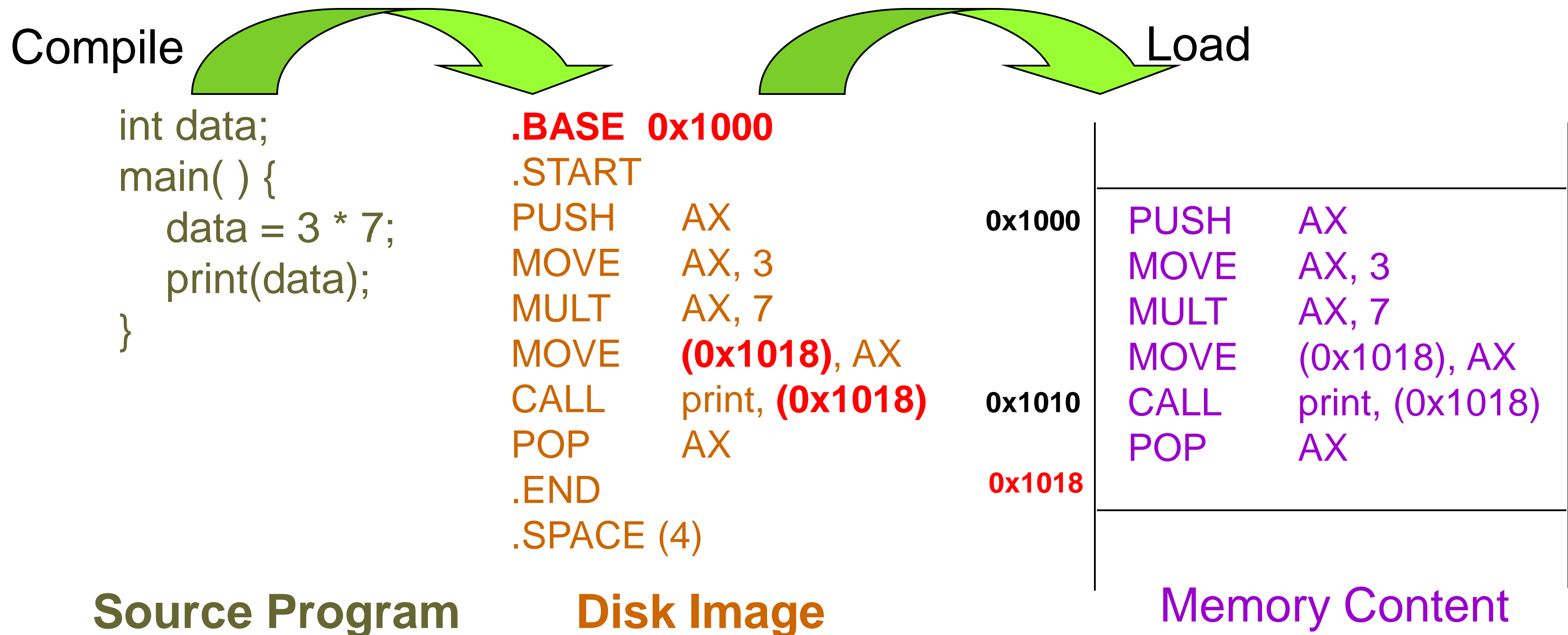
- How to refer memory in a program?
  - address binding
- How to load a program into memory ?
  - static/dynamic loading and linking
- How to move a program between mem. & disk?
  - swap
- How to allocate memory?
  - paging, segment

Multistep Processing of a User Program



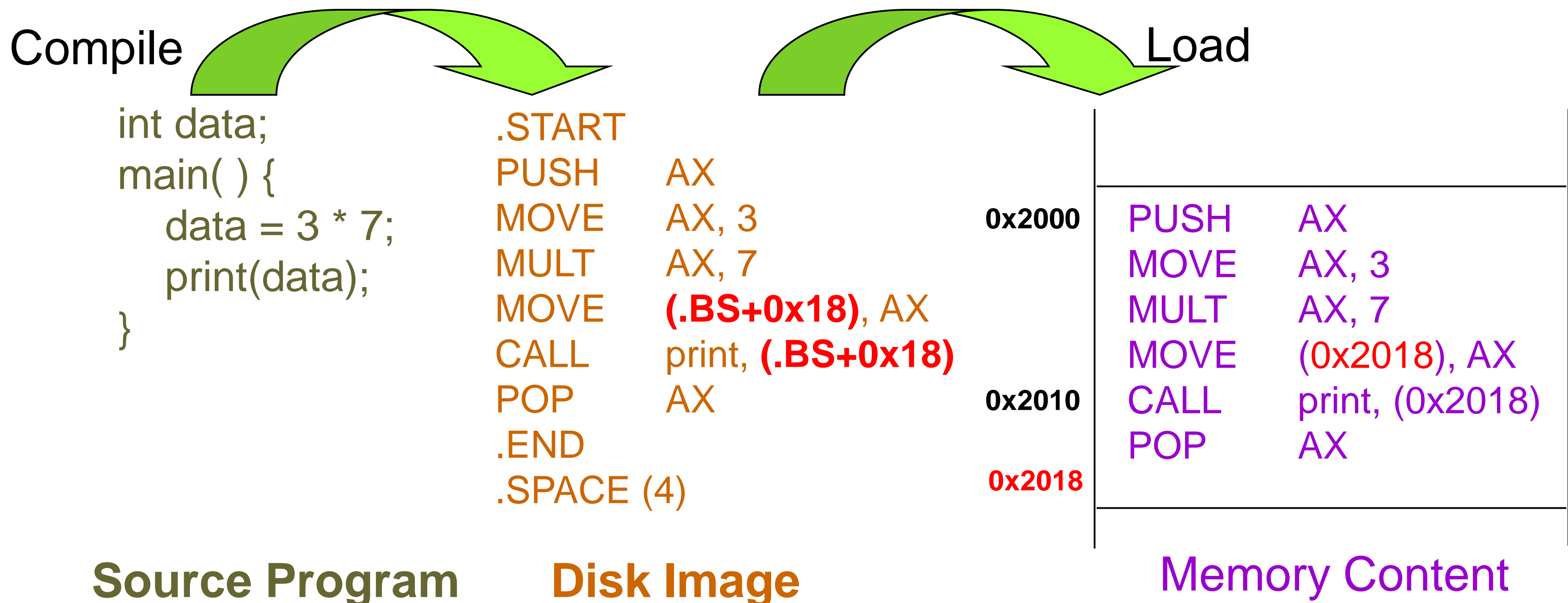
Address Binding – Compile Time

- Program is written as symbolic code
- Compiler translates symbolic code into *absolute code*
- If starting location changes → recompile
- Example: MS-DOS .COM format binary



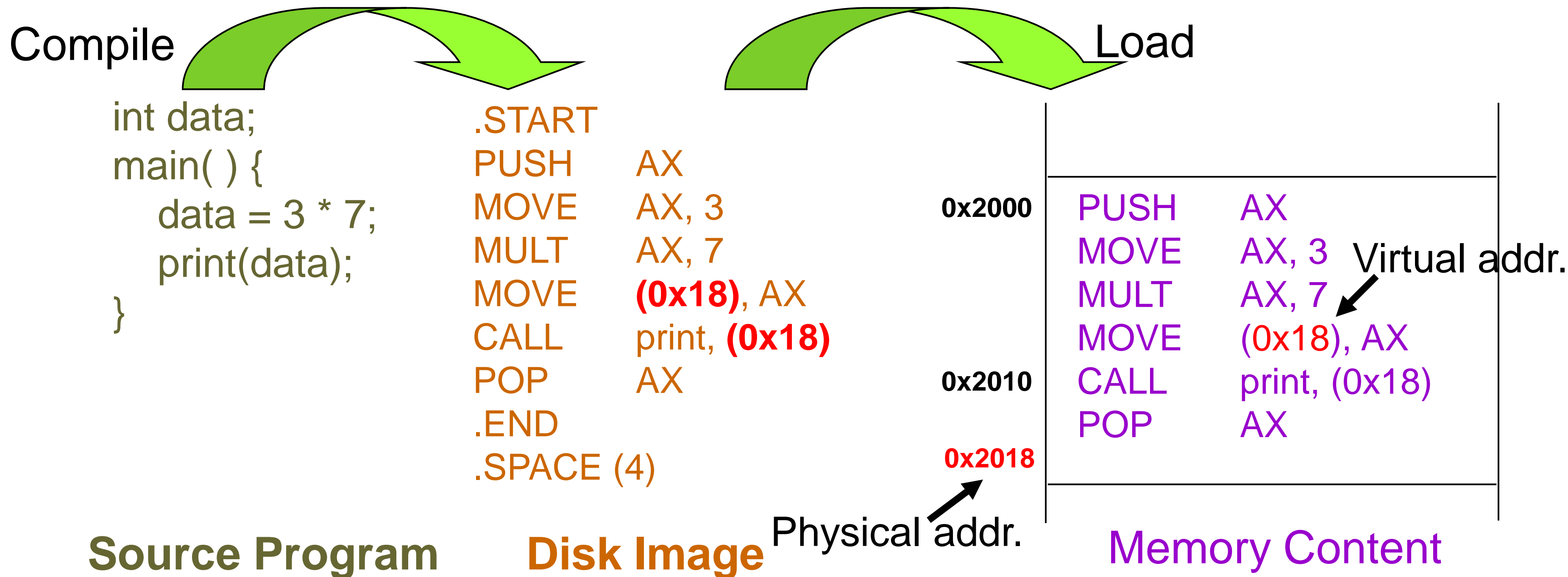
Address Binding – Load Time

- Compiler translates symbolic code into *relocatable code*
- *Relocatable code:*
  - *Machine language that can be run from any memory location*
- If starting location changes → reload the code



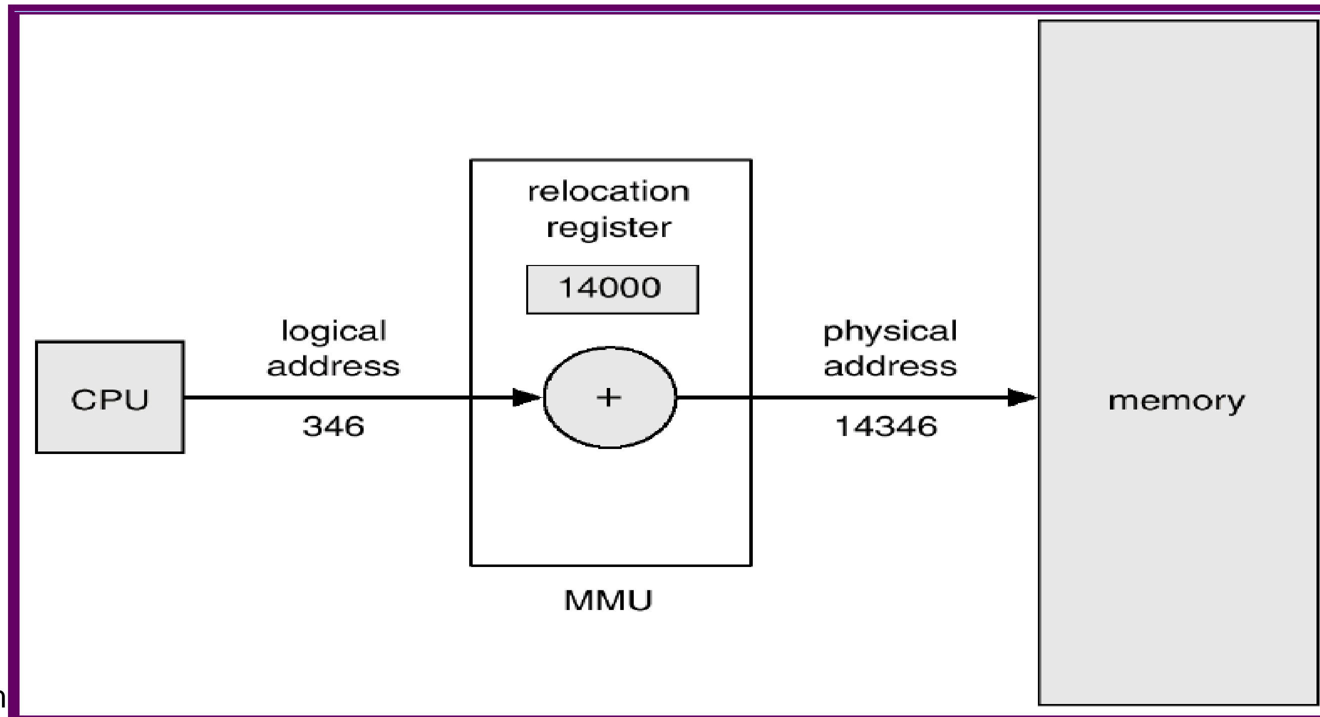
Address Binding – Execution Time

- Compiler translates symbolic code into *logical-address (i.e. virtual-address) code*
- Special **hardware (i.e. MMU)** is needed for this scheme
- Most general-purpose OS use this method



Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



Logical vs. Physical Address

- Logical address – generated by CPU
  - a.k.a. virtual address
- Physical address – seen by the memory module
- compile-time & load-time address binding
  - logical addr = physical addr
- Execution-time address binding
  - logical addr ≠ physical addr
- The user program deals with logical addresses; it never sees the real physical addresses

Outline

- How to refer memory in a program?
  - address binding
- How to load a program into memory ?
  - static/dynamic loading and linking
- How to move a program between mem. & disk?
  - swap
- How to allocate memory?
  - paging, segment

Dynamic Loading

- The entire program must be in memory for it to execute?
- No, we can use dynamic-loading
  - A routine is loaded into memory when it is called
- Better memory-space utilization
  - unused routine is never loaded
  - Particularly useful when large amounts of code are infrequently used (e.g., error handling code)
- No special support from OS is required implemented through program (library, API calls)

Dynamic Loading Example in C

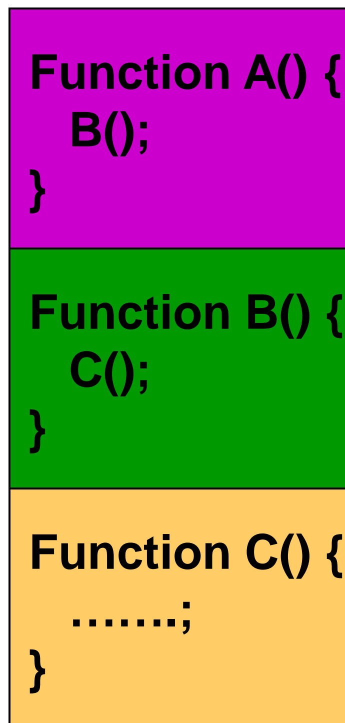
- dlopen(): opens a library and prepares it for use
- desym(): looks up the value of a symbol in a given (opened) library.
- dlclose(): closes a DL library

```
#include <dlfcn.h>
int main() {
  double (*cosine)(double);
  void* handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
  cosine = dlsym(handle, "cos");
  printf ("%f\n", (*cosine)(2.0));
  dlclose(handle);
}
```

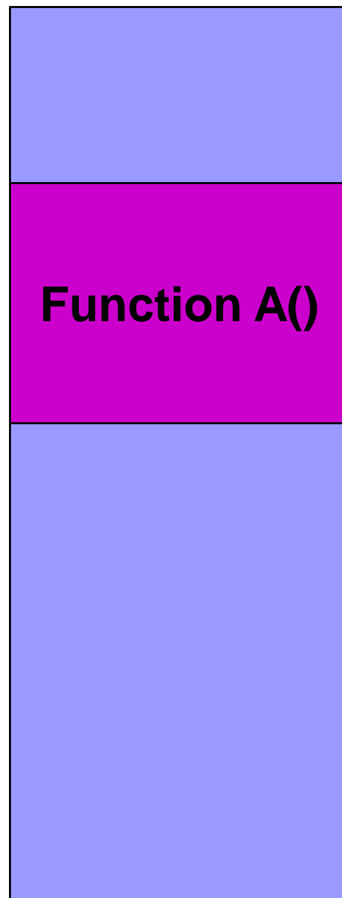
Dynamic Loading

Disk image

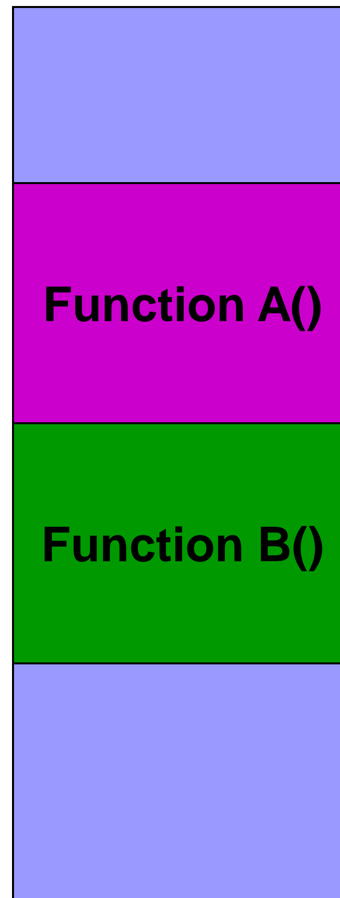
Memory content



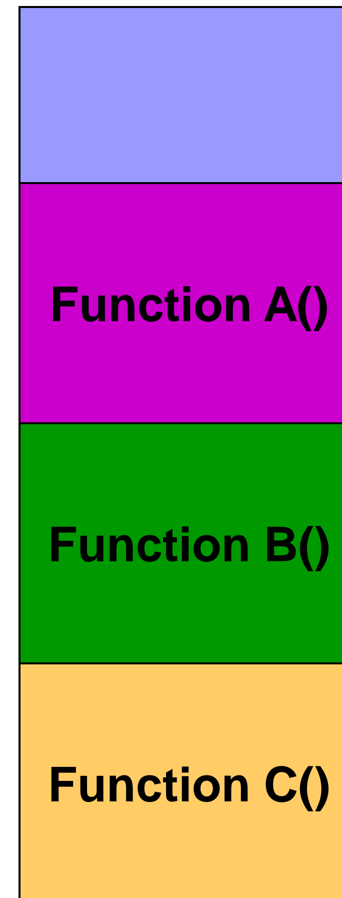
Init



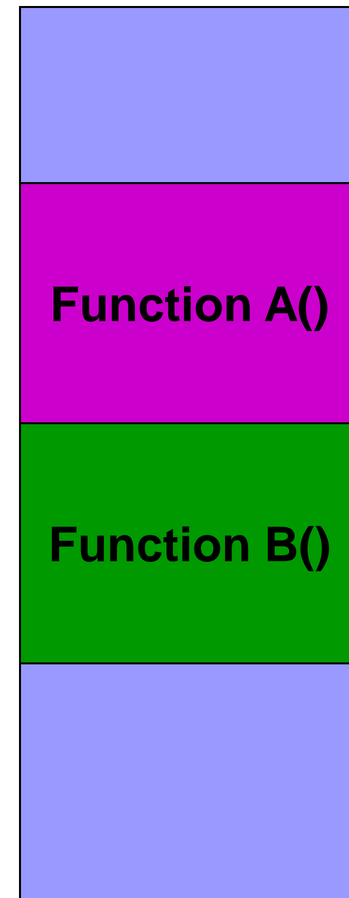
After B() called



After C() called



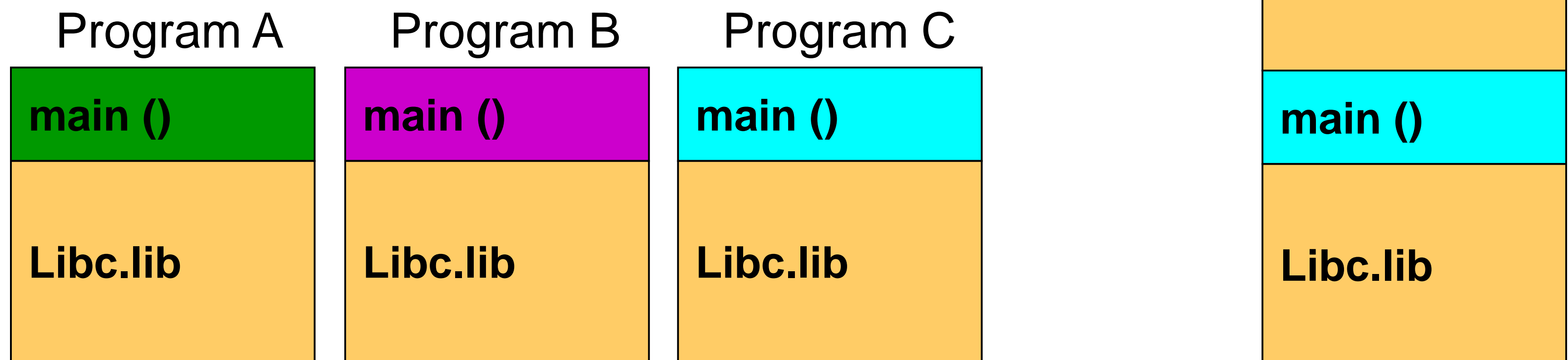
After C() ends



Static Linking

- *Static linking*: libraries are combined by the loader into the program in-memory image
  - Waste memory: duplicated code
  - Faster during execution time

*Static linking + Dynamic loading
  - Still can't prevent duplicated code

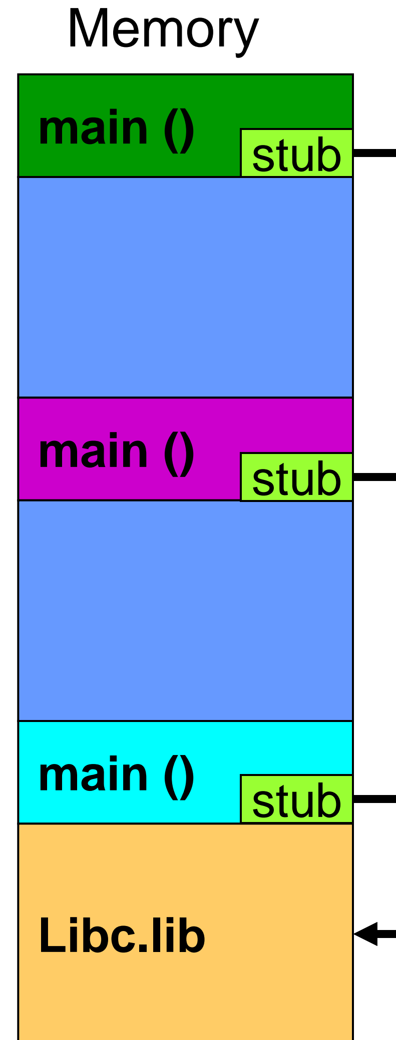


Memory

Memory
main ()
Libc.lib
main ()
Libc.lib
main ()
Libc.lib

Dynamic Linking

- *Dynamic linking*: Linking postponed until execution time
  - Only one code copy in memory and shared by everyone
  - A stub is included in the program in-memory image for each lib reference
  - Stub call → check if the referred lib is in memory → if not, load the lib → execute the lib
  - DLL (Dynamic link library) on Windows



Review Slides (1)

- 3 types of address binding?
  - compile-time
  - load-time
  - execution-time
- logical address? physical address?
  - virtual → physical mapping?
- dynamic loading? static loading?
- dynamic linking? static linking?

Outline

- How to refer memory in a program?
  - address binding
- How to load a program into memory ?
  - static/dynamic loading and linking
- How to move a program between mem. & disk?
  - swap
- How to allocate memory?
  - paging, segment

Swapping

Swapping

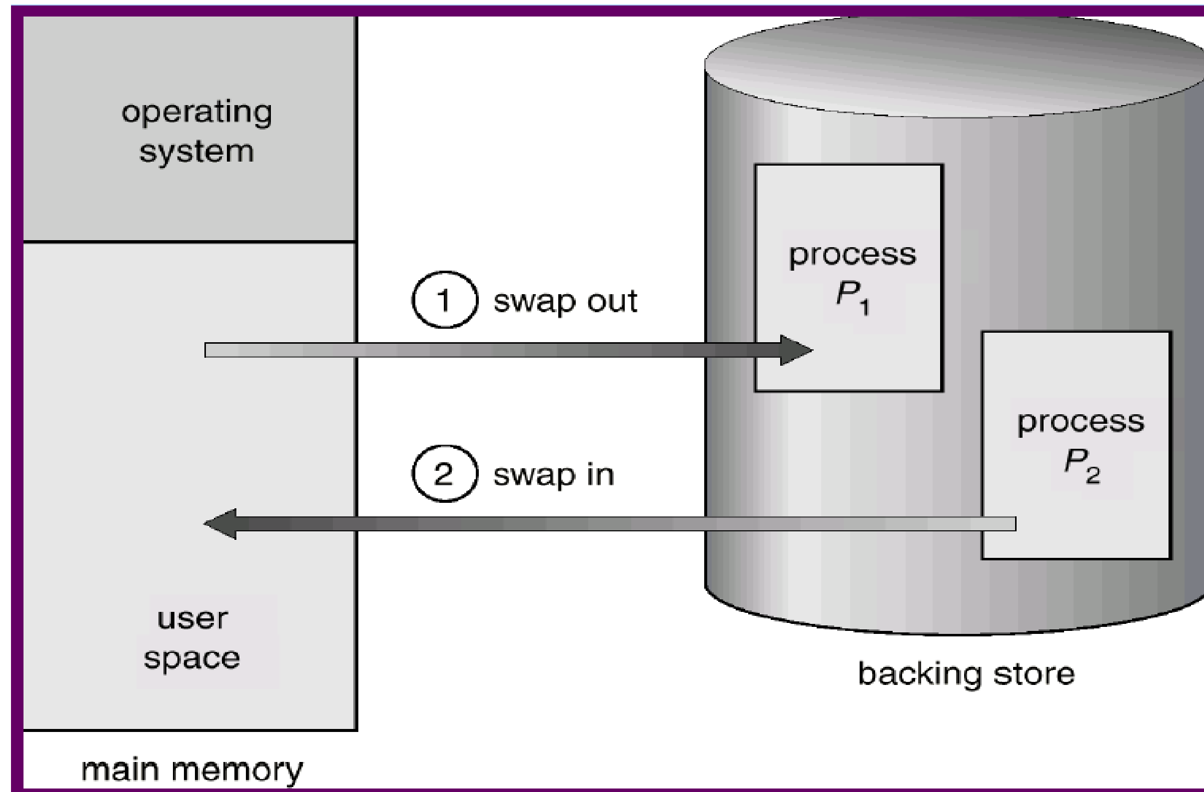
- A process can be swapped out of memory to a *backing store*, and later brought back into memory for continuous execution
  - Also used by midterm scheduling, different from context switch
- Backing store – a chunk of **disk**, *separated from file system*, to provide direct access to these memory images
- Why Swap a process:
  - Free up memory
  - Roll out, roll in: swap lower-priority process with a higher one

Swapping (cont'd)

- Swap back memory location
  - If binding is done at compile/load time
    - ➔ swap back memory address must be the same
  - If binding is done at execution time
    - ➔ swap back memory address can be different
- A process to be swapped == must be idle
  - Imagine a process that is waiting for I/O is swapped
  - Solutions:
    - Never swap a process with pending I/O
    - I/O operations are done through **OS buffers** (i.e. a memory space not belongs to any user processes)

Process Swapping to Backing Store

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



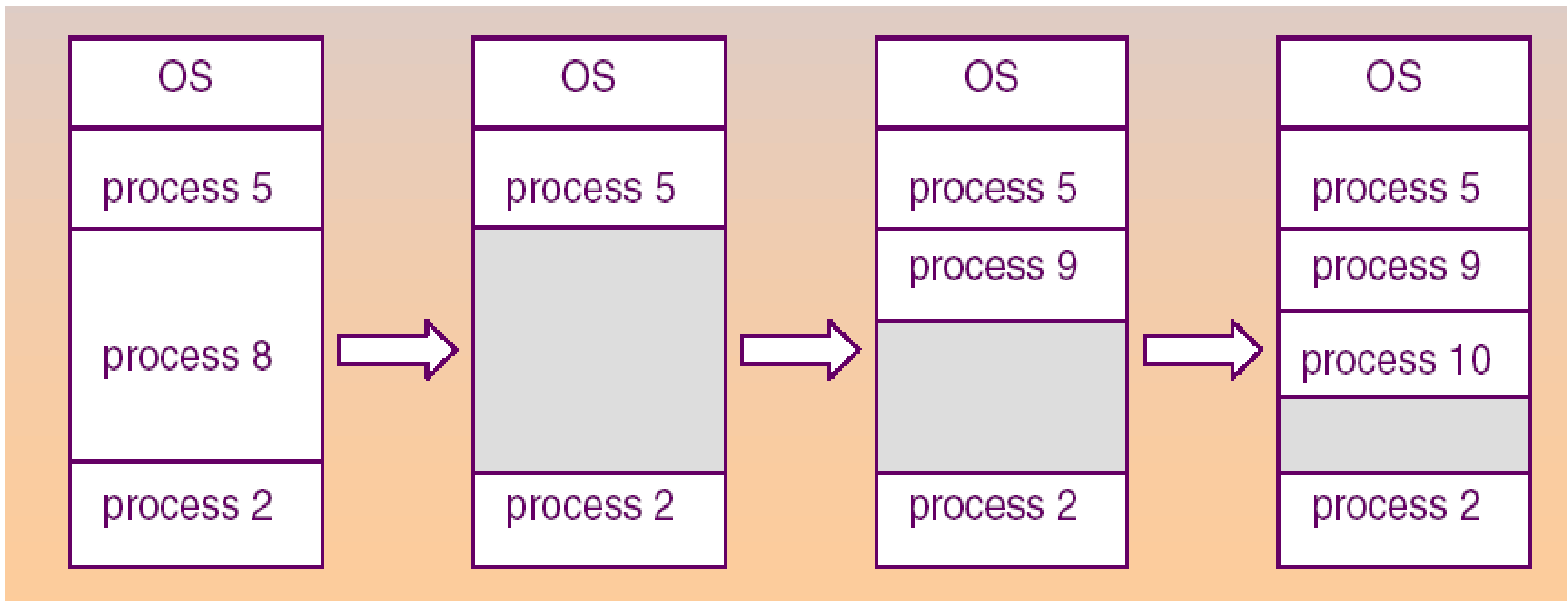
Contiguous Memory
Allocation

Memory Allocation

- Fixed-partition allocation:
  - Each process loads into one partition of fixed-size
  - Degree of multi-programming is bounded by the number of partitions
- Variable-size partition
  - Hole: block of contiguous free memory
  - Holes of various size are scattered in memory

Multiple Partition (Variable-Size) Method

- When a process arrives, it is allocated a hole large enough to accommodate it
- The OS maintains info. on each in-use and free hole
- A freed hole can be merged with another hole to form a larger hole

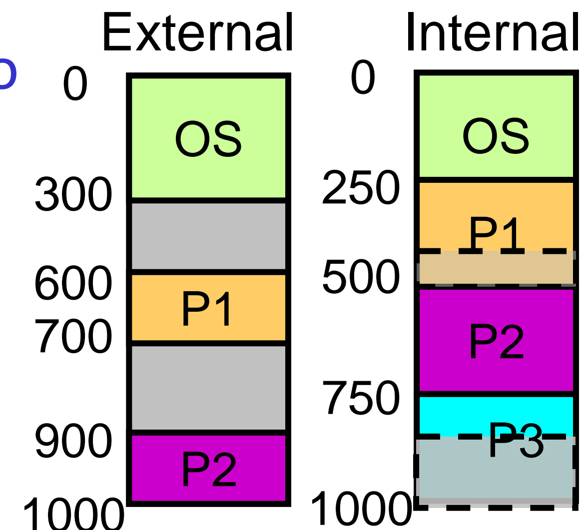


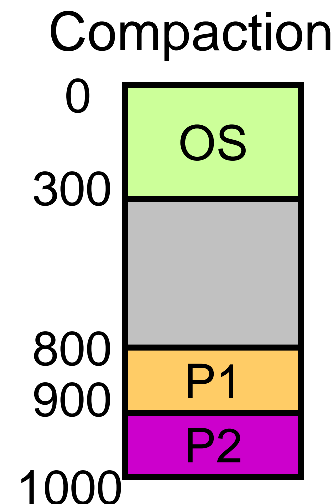
Dynamic Storage Allocation Problem

- How to satisfy a request of size *n* from a list of free holes
- *First-fit* – allocate the 1st hole that fits
- *Best-fit* – allocate the smallest hole that fits
  - **Must search through the whole list**
- *Worst-fit* – allocate the largest hole
  - **Must also search through the whole list**
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- *External fragmentation*
  - Total free memory space is big enough to satisfy a request, but is not contiguous
  - Occur in variable-size allocation
- *Internal fragmentation*
  - Memory that is internal to a partition but is not being used
  - Occur in fixed-partition allocation
- Solution: *compaction*
  - Shuffle the memory contents to place all free memory together in one large block at execution time
  - Only if binding is done at execution time





Review Slides (2)

- Swapping?
- Contiguous memory allocation?
  - fixed-size memory allocation?
  - variable-size memory allocation?
    - first-fit, best-fit, worst-fit?
- External & internal fragmentation?
  - compaction?

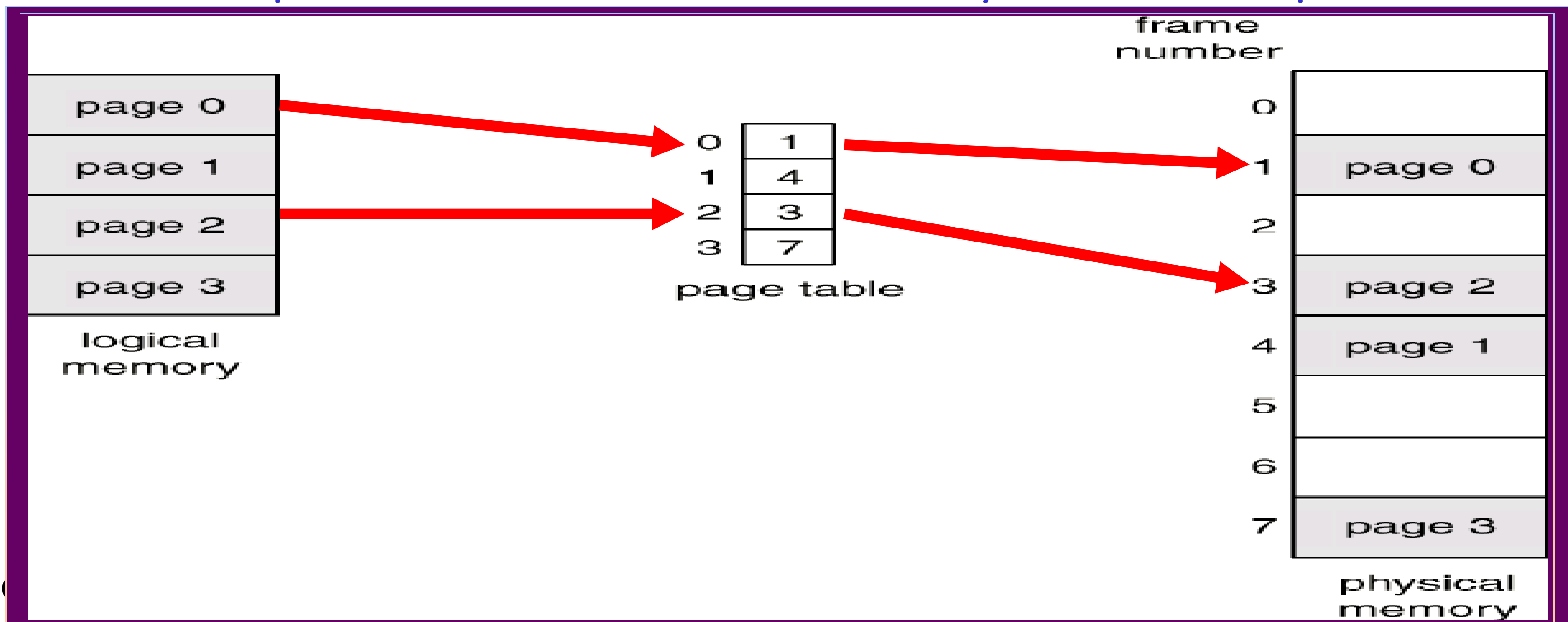
Non-Contiguous Memory Allocation — Paging

Paging Concept

- Method:
  - Divide physical memory into fixed-sized blocks called ***frames***
  - Divide logical address space into blocks of the same size called ***pages***
  - To run a program of *n* pages, need to find *n* free frames and load the program
  - keep track of free frames
  - Set up a **page table** to translate logical to physical addresses
- Benefit:
  - Allow the physical-address space of a process to be noncontiguous
  - Avoid external fragmentation
  - Limited internal fragmentation
  - Provide shared memory/pages

Paging Example

- Page table:
  - Each entry maps to the base address of a page in physical memory
  - A structure maintained by OS *for* ***each process***
    - Page table includes only pages owned by a process
    - A process cannot access memory outside its space

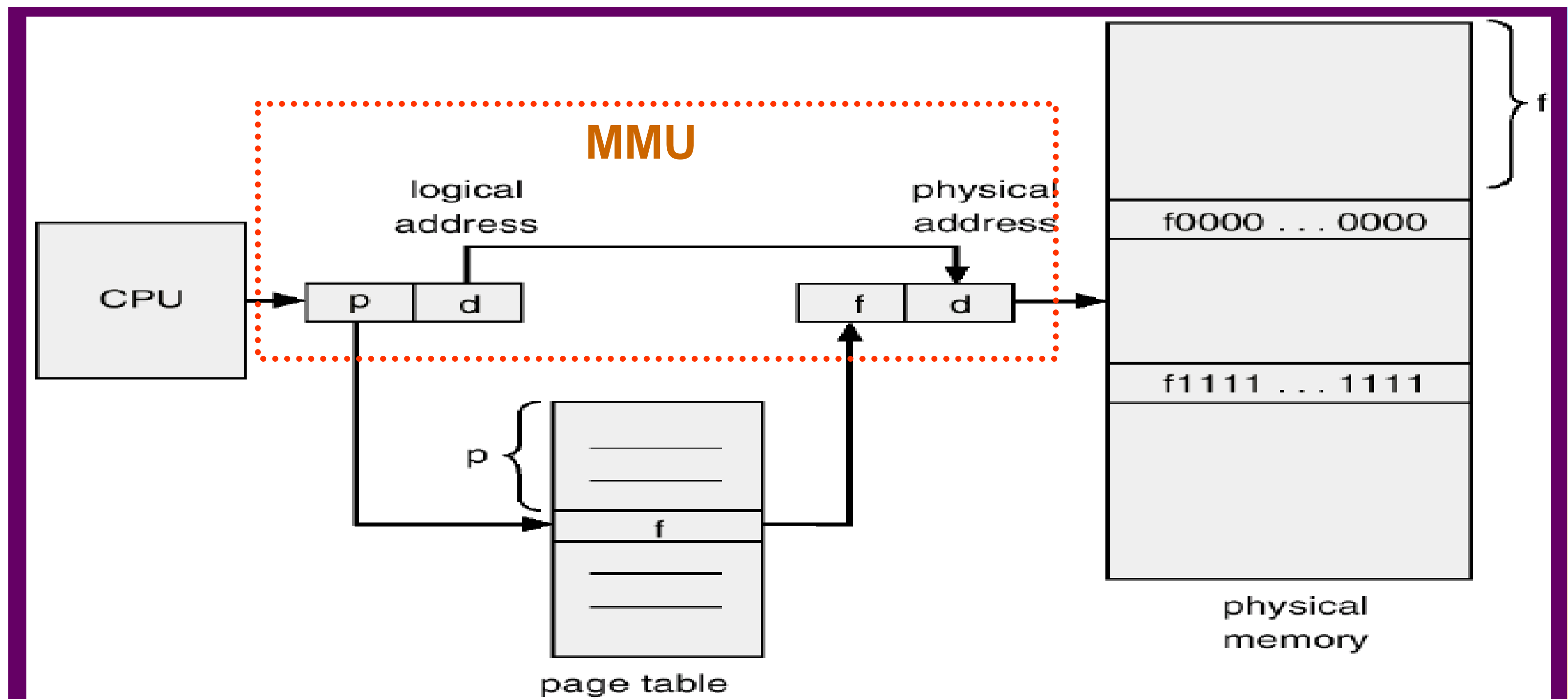


Address Translation Scheme

- Logical address is divided into two parts:
  - Page number (p)
    - used as an index into a page table which contains base address of each page in physical memory
    - N bits means a process can allocate at most 2^N pages
      → 2^N x page size memory size
  - Page offset (d)
    - combined with base address to define the physical memory address that is sent to the memory unit
    - N bits means the page size is 2^N
- Physical addr = page base addr + page offset

Address Translation Architecture

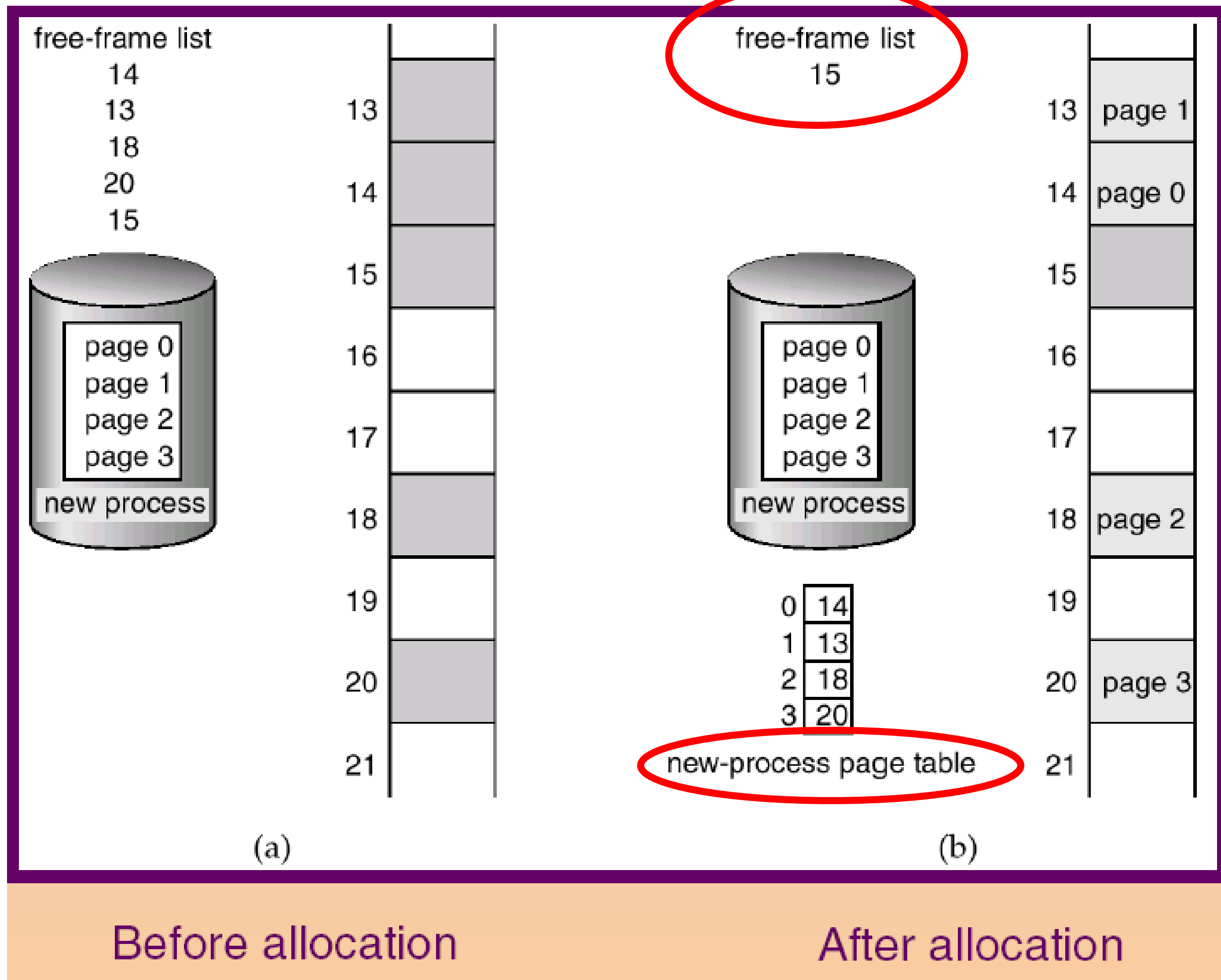
- If Page size is 1KB(2^10) & Page 2 maps to frame 5
- Given 13 bits logical address: (p=2,d=20), what is physical addr.?
  - 5*(1KB)+20 =1,010,000,000,000+0,000,010,100
    =1,010,000,010,100



Address Translation

- Total number of pages does not need to be the same as the total number of frames
  - Total # pages determines the logical memory size of a process
  - Total # frames depending on the size of physical memory
- E.g.: Given 32 bits logical address, 36 bits physical address and 4KB page size, what does it mean?
  - Page table size: $2^{32} / 2^{12} = 2^{20}$ entries
  - Max program memory: 2^{32} = 4GB
  - Total physical memory size: 2^{36} = 64GB
  - Number of bits for page number: 2^{20} pages → 20bits
  - Number of bits for frame number: 2^{24} frames → 24bits
  - Number of bits for page offset: 4KB page size = 2^{12} bytes → 12

Free Frames



Page / Frame Size

- The page (frame) size is defined by hardware
  - Typically a power of 2
  - Ranging from 512 bytes to 16MB / page
  - 4KB / 8KB page is commonly used
- Internal fragmentation?
  - Larger page size → More space waste
- But page sizes have grown over time
  - memory, process, data sets have become larger
  - better I/O performance (during page fault)
  - page table is smaller

Paging Summary

- Paging helps separate user's view of memory and the actual physical memory
- User view's memory: one single contiguous space
  - Actually, user's memory is scatter out in physical memory
- OS maintains a copy of the page table for each process
- OS maintains a frame table for managing physical memory
  - One entry for each physical frame
  - Indicate whether a frame is free or allocated
  - If allocated, to which page of which process or processes

Implementation of Page Table

- Page table is kept in memory
- Page-table base register (**PTBR**)
  - The physical memory address of the page table
  - The PTBR value is stored in PCB (Process Control Block)
  - Changing the value of PTBR during Context-switch
- With PTBR, each memory reference results in 2 memory reads
  - One for the page table and one for the real address
- The 2-access problem can be solved by
  - Translation Look-aside Buffers (**TLB**) (HW) which is implemented by **Associative memory** (HW)

Associative Memory

- All memory entries can be accessed at the same time
  - Each entry corresponds to an associative register
- But number of entries are limited
  - Typical number of entries: 64 ~ 1024

- Associative memory – parallel search

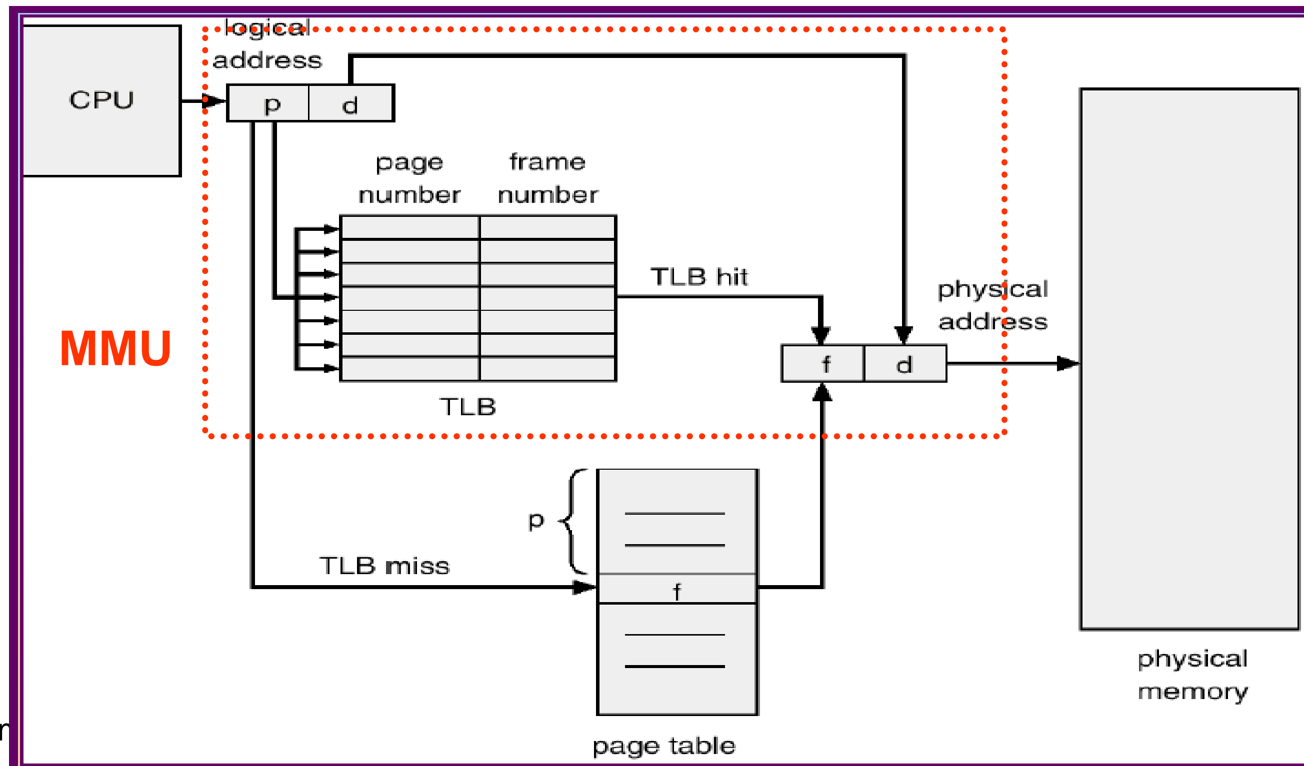
Page #	Frame #

Address translation (A´, A´´)

- If A´ is in associative register, get frame # out.
- Otherwise get frame # from page table in memory

Translation Look-aside Buffer (TLB)

- A cache for page table shared by all processes
- TLB must be flushed after a context switch
  - Otherwise, TLB entry must has a PID field (address-space identifiers (ASIDs))



Effective Memory-Access Time

- 20 ns for TLB search
- 100 ns for memory access
- Effective Memory-Access Time (EMAT)
  - 70% TLB hit-ratio:

    EMAT = 0.70 x (20 + 100) + (1-0.70) * (20+100+100) = 150 ns
  - 98% TLB hit-ratio

    EMAT = 0.98 x 120 + 0.02 x 220 = 122 ns

Review Slides (3)

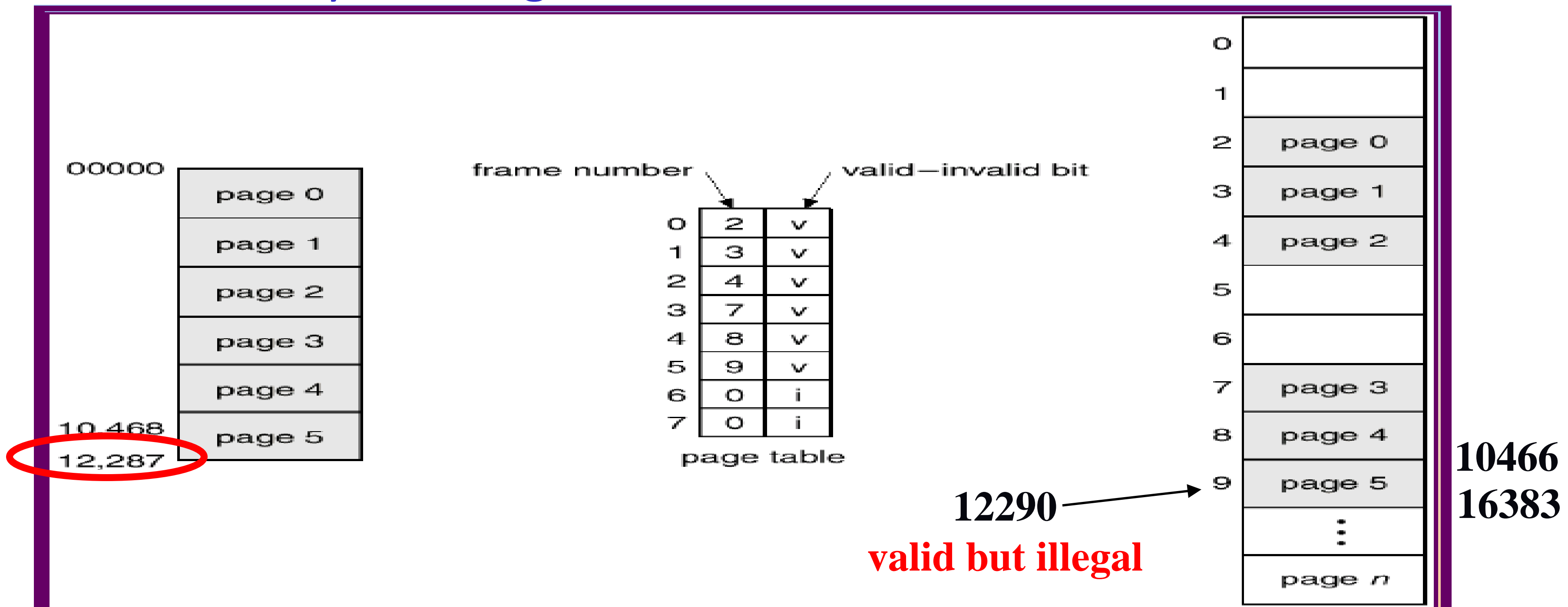
- memory frame? page? typical page size?
- page table? virtual → physical translation?
- What is PTBR register? When to update it?
- Memory reads # for each reference?
- HW support for paging speed?
  - associative memory
  - TLB

Memory Protection

- Each page is associated with a set of protection bit in the page table
  - E.g., a bit to define read/write/execution permission
- Common use: valid-invalid bit
  - Valid: the page/frame is in the process' logical address space, and is thus a legal page
  - Invalid: the page/frame is not in the process' logical address space

Valid-Invalid Bit Example

- Potential issues:
  - Un-used page entry cause memory waste ➔ use page table length register (PTLR)
  - Process memory may NOT be on the boundary of a page ➔ memory limit register is still needed

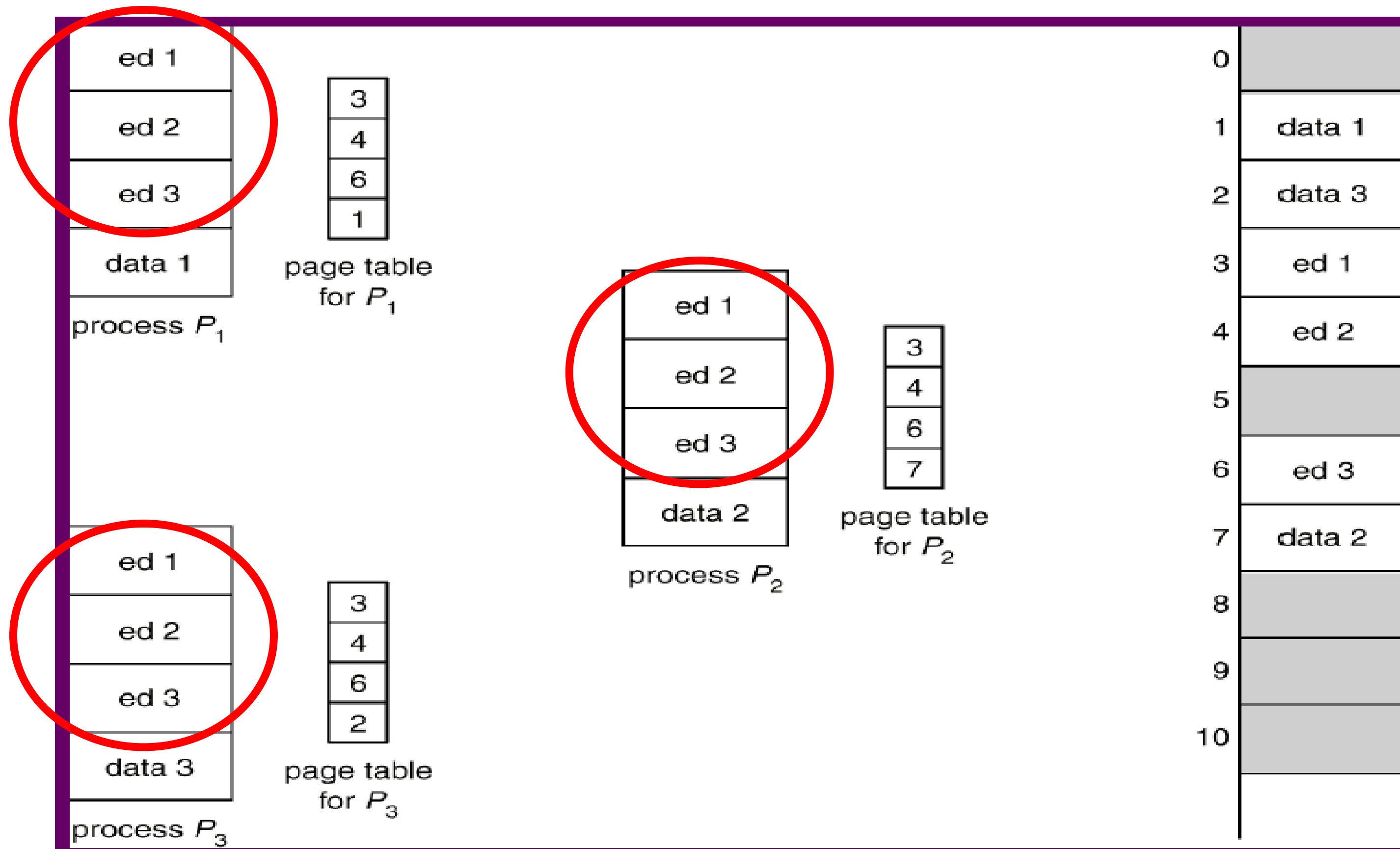


Shared Pages

- Paging allows processes share common code, which must be reentrant
- Reentrant code (pure code)
  - It never change during execution
  - text editors, compilers, web servers, etc
- Only one copy of the shared code needs to be kept in physical memory
- Two (several) virtual addresses are mapped to one physical address
- Process keeps a copy of its own private data and code

Shared Pages by Page Table

- Shared code must appear in the same location in the logical address space of all processes

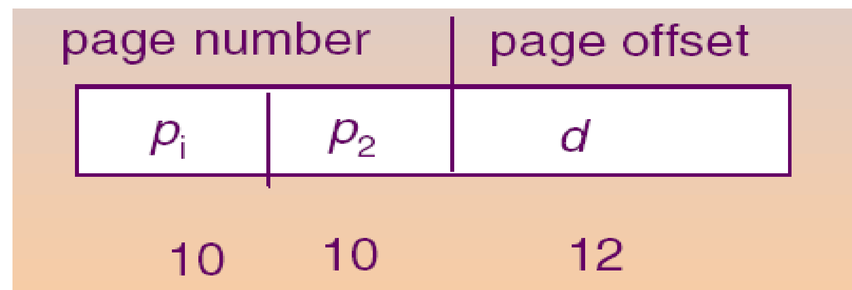


Page Table Memory Structure

- Page table could be huge and difficult to be loaded
  - 4GB (2^{32}) logical address space with 4KB (2^{12}) page
    - ➔1 million (2^{20}) page table entry
  - Assume each entry need 4 bytes (32bits)
    - ➔ Total size=4MB
  - Need to break it into several smaller page tables, better within a single page size (i.e. 4KB)
  - Or reduce the total size of page table
- Solutions:
  - Hierarchical Paging
  - Hash Page Tables
  - Inverted Page Table

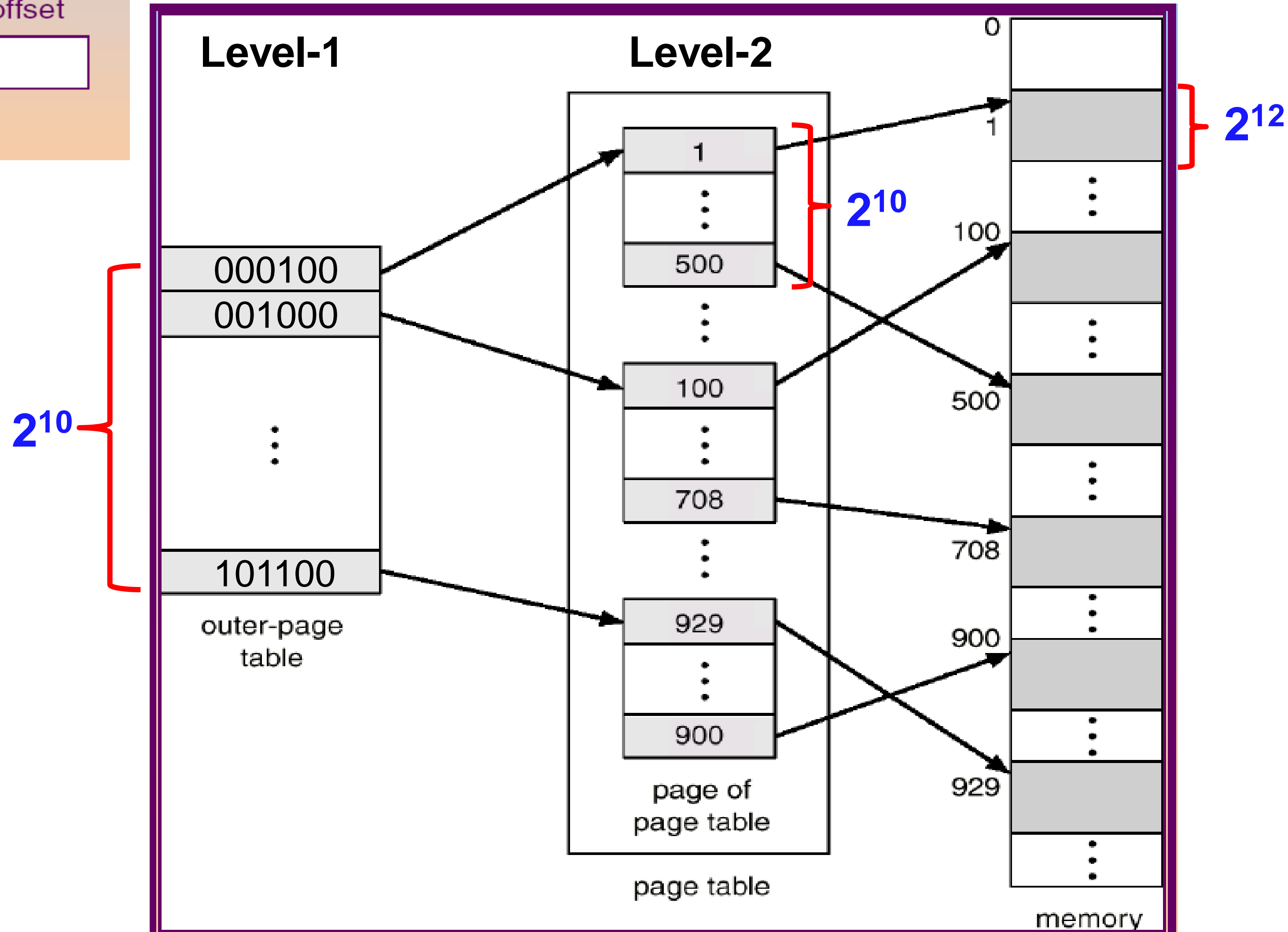
Hierarchical Paging

- Break up the logical address space into multiple page tables
  - Paged the page table
  - i.e. *n*-level page table
- Two-level paging (32-bit address with 4KB (2^{12}) page size)
  - 12-bit offset (d) → 4KB (2^{12}) page size
  - 10-bit outer page number → 1K (2^{10}) page table entries
  - 10-bit inner page number → 1K (2^{10}) page table entries
  - 3 memory accesses

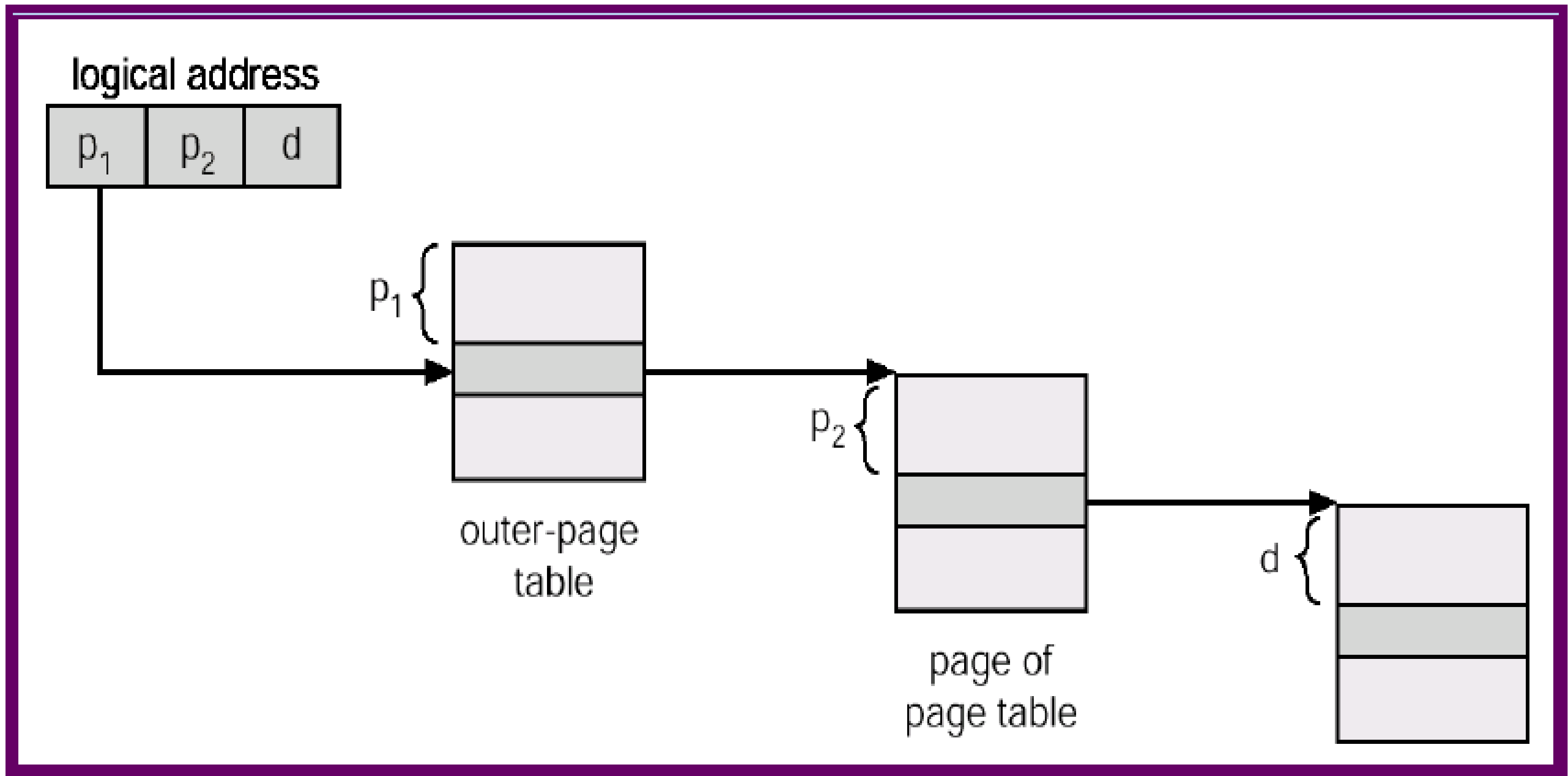


Two-Level Page Table Example

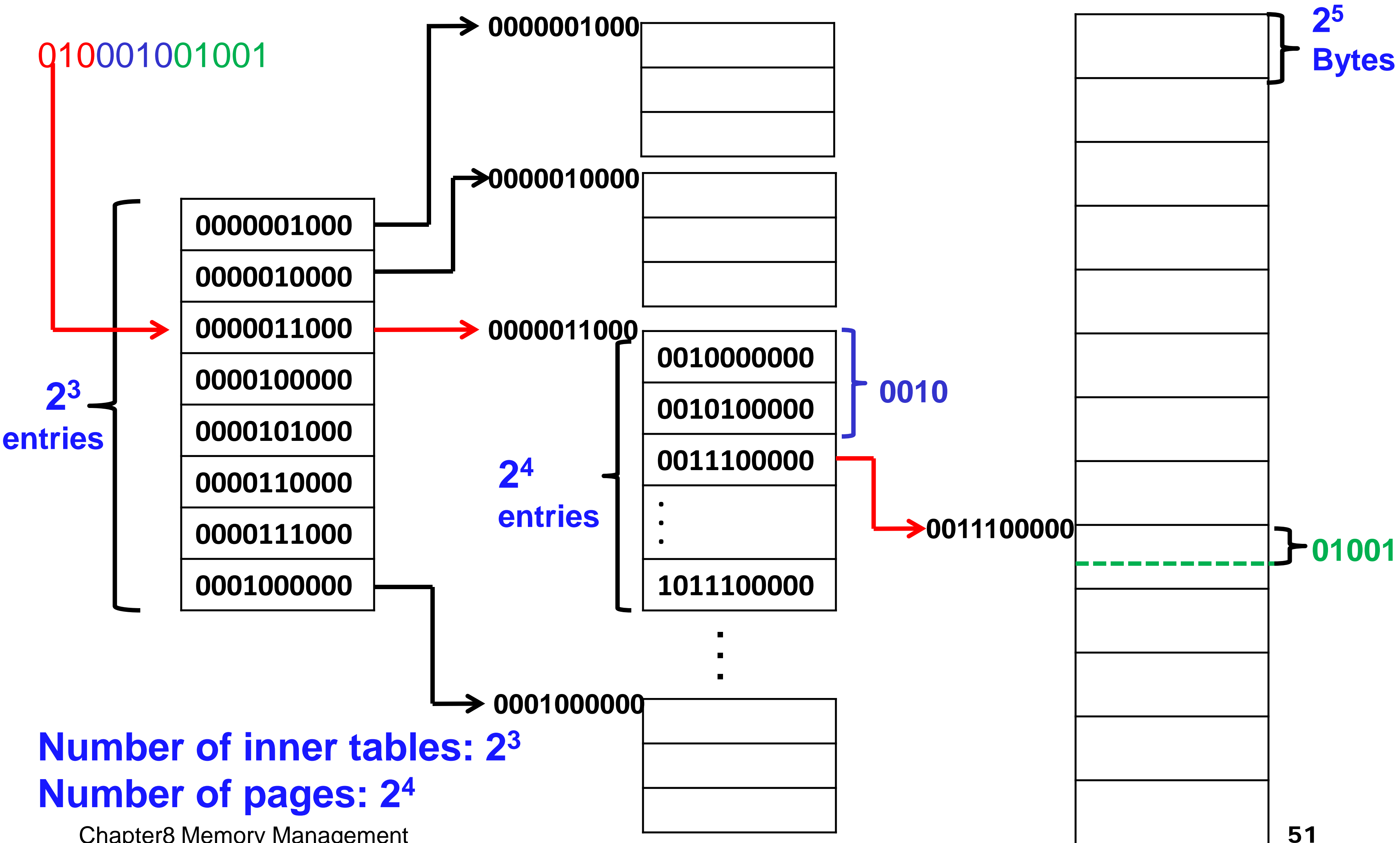
page number		page offset
p_i	p_2	d
10	10	12



Two-Level Address Translation



Two-Level Page Table Translation Example



64-bit Address

- How about 64-bit address? (assume each entry needs 4Bytes)
  - 42 (p1) + 10 (p2) + 12 (offset)
    - → outer table requires 2^{42} x 4B = 16TB contiguous memory!!!
  - 12 (p1)+10 (p2)+10 (p3)+10 (p4)+10 (p5)+12 (offset)
    - → outer table requires 2^{12} x 4B = 16KB contiguous memory
    - → 6 memory accesses!!!
- Examples:
  - SPARC (32-bit) and Linux use 3-level paging
  - Motorola 68030 (32-bit) use 4-level paging

Hashed Page Table

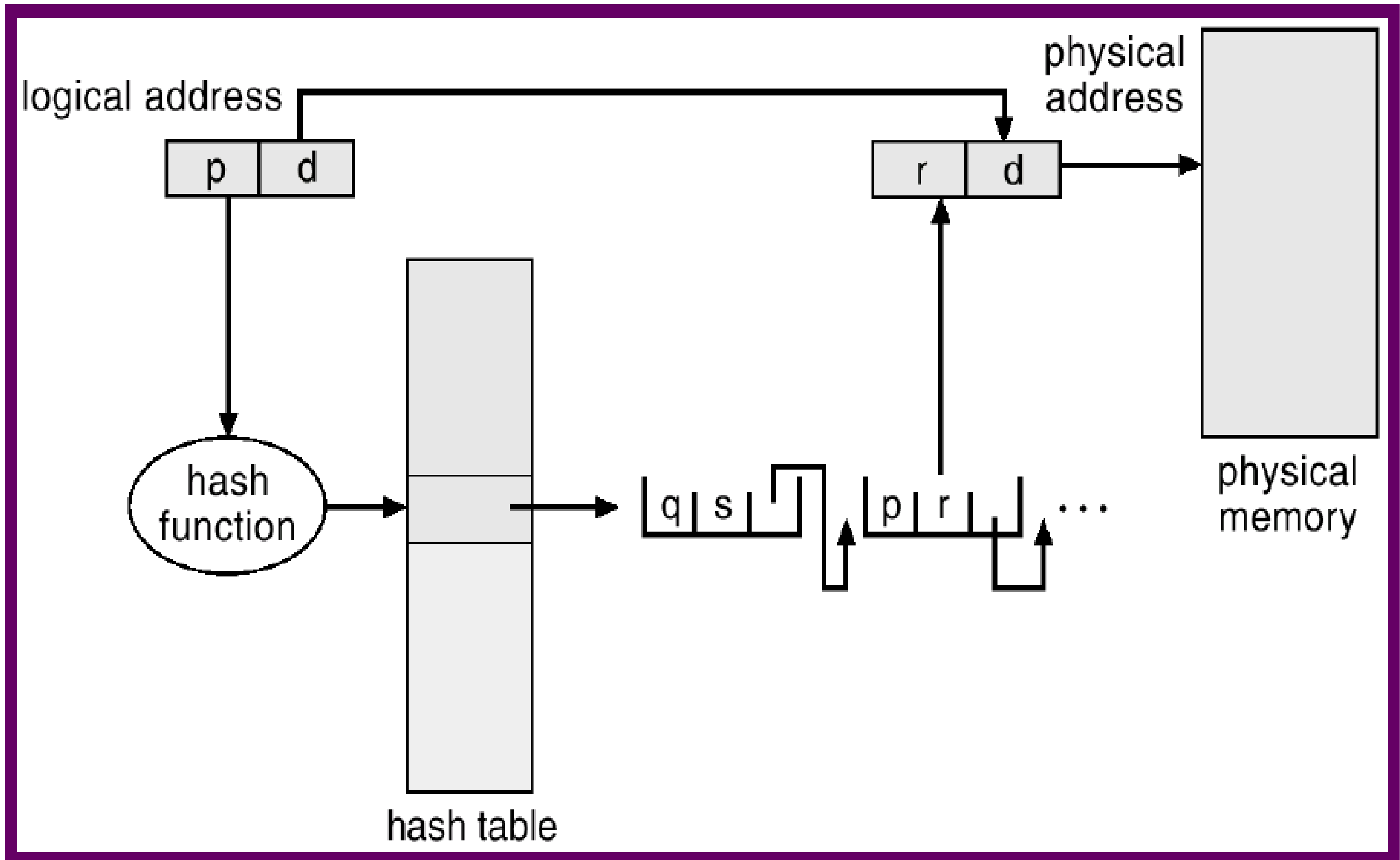
- Commonly-used for address > 32 bits
- Virtual page number is hashed into a hash table
- The size of the hash table varies
  - Larger hash table → smaller chains in each entry
- Each entry in the hashed table contains
  - (Virtual Page Number, Frame Number, Next Pointer)
  - Pointers waste memory
  - Traverse linked list waste time & cause additional memory references

Buckets:

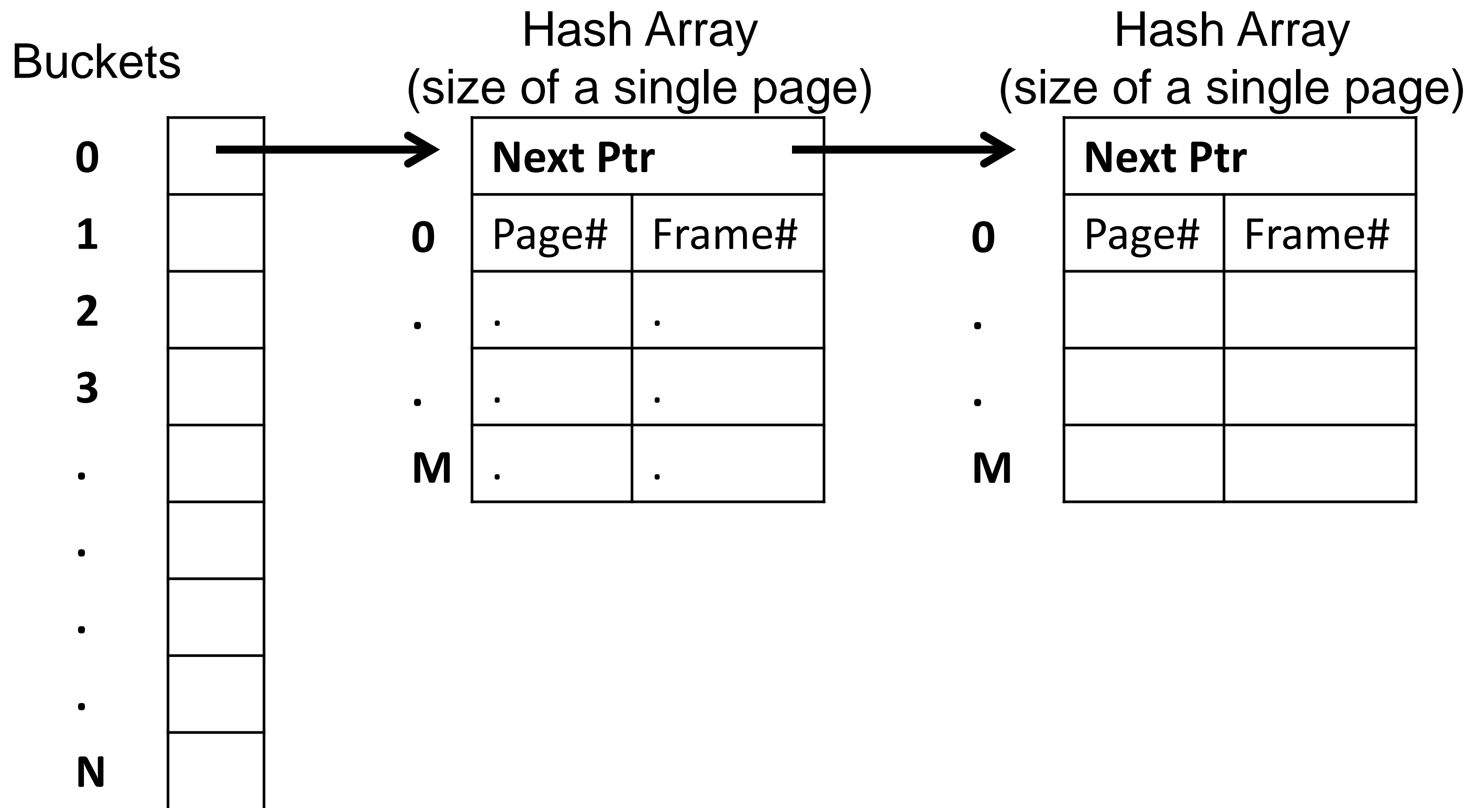
Bucket	Chain	
0		
1	1001	
2	7	
3		
4	14	59

Hash function
$f(p) = p\%5$

Hashed Page Table Address Translation



Improved Hashed Page Table Implementation

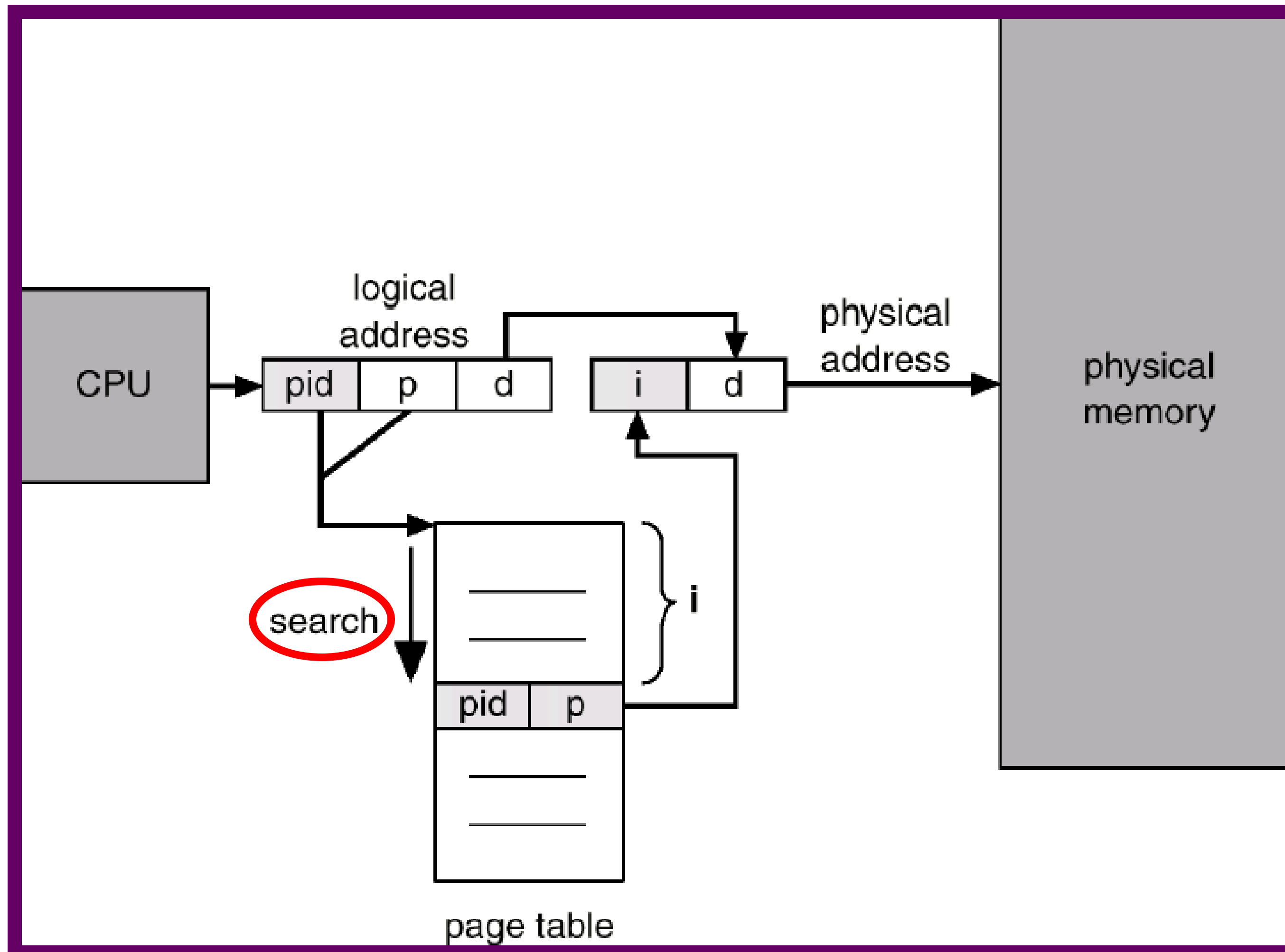


Source: M. Talluri, M. D. Hill, and Y. A. Khalidi. 1995. A new page table for 64-bit address spaces. *SIGOPS Oper. Syst. Rev.* 29, 5 (December 1995), 184-200. http://pages.cs.wisc.edu/~markhill/papers/sosp95_pagetables.pdf

Inverted Page Table

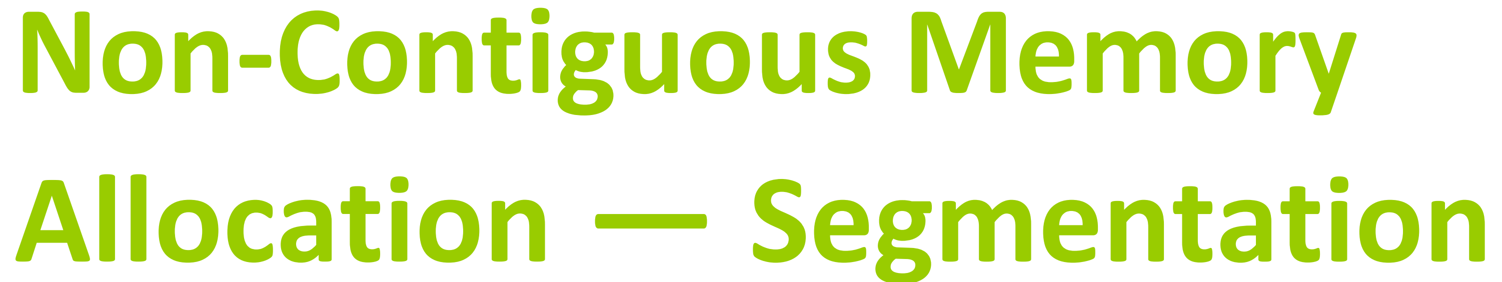
- Maintains **NO** page table for each process
- Maintains a *frame table* for the whole memory
  - One entry for each real frame of memory
- Each entry in the frame table has
  - (PID, Page Number)
- Eliminate the memory needed for page tables but increase memory access time
  - Each access needs to search the whole frame table
  - Solution: use hashing for the frame table
- Hard to support shared page/memory

Inverted Page Table Addr Translation



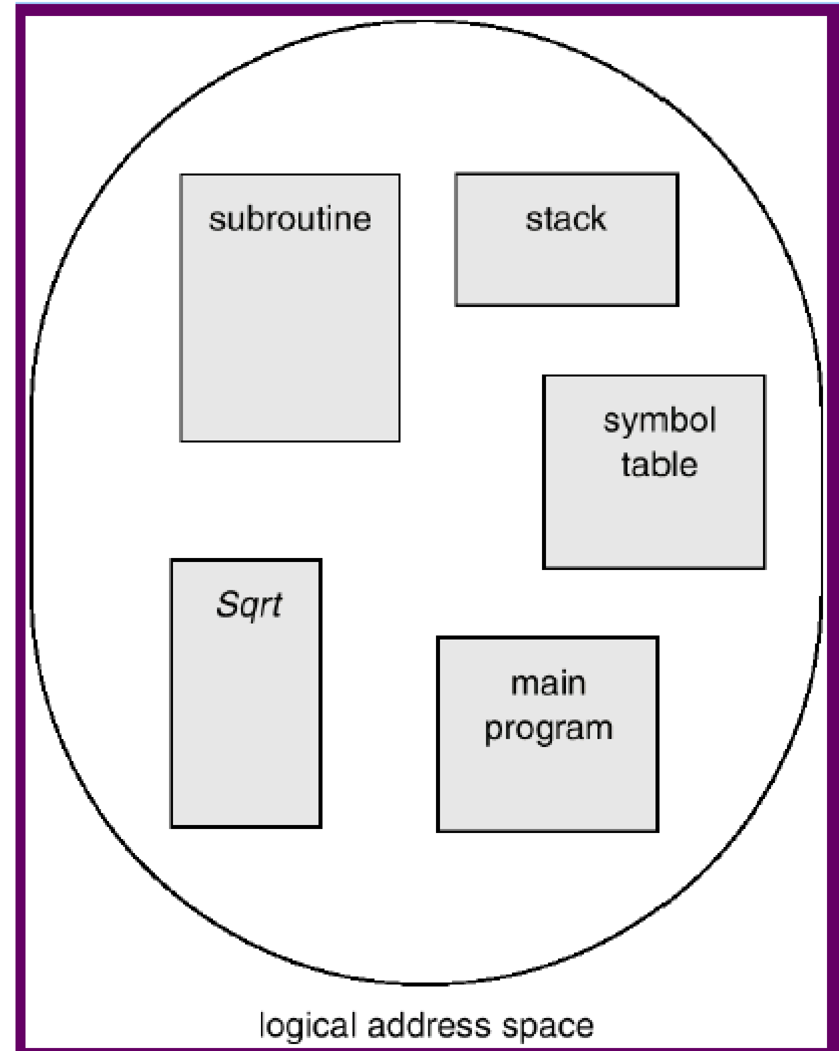
Review Slides (4)

- memory protection by page table?
  - valid, invalid bits?
- page table memory structure?
  - hierarchical → 2-level, 3-level, etc
  - hash table → linked list
  - inverted page table
- How are pages shared by different processes?

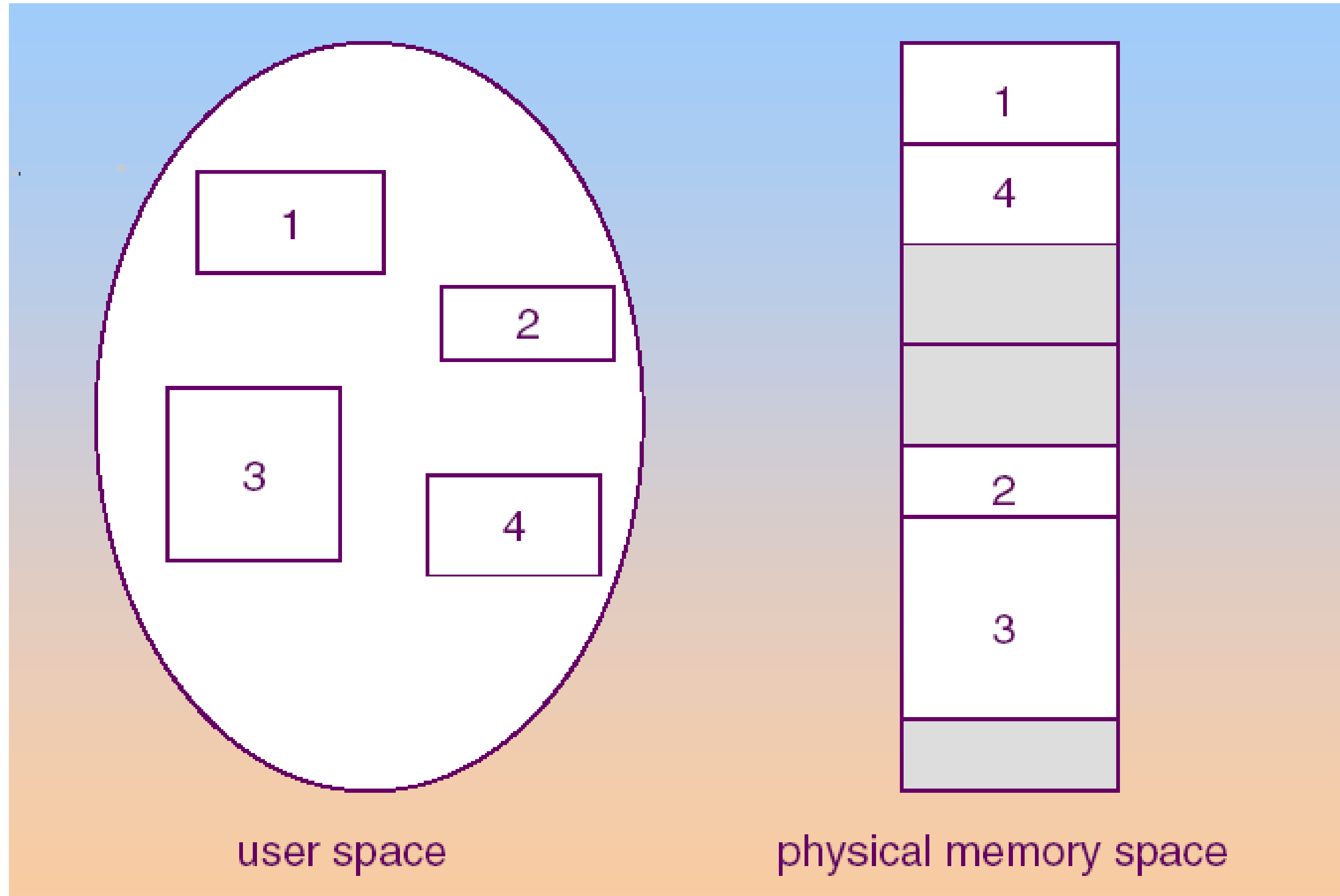

Non-Contiguous Memory
Allocation — Segmentation

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
  - function, object
  - local/global variables,
  - stack, symbol table,
  - arrays, etc…



Logical View of Segmentation

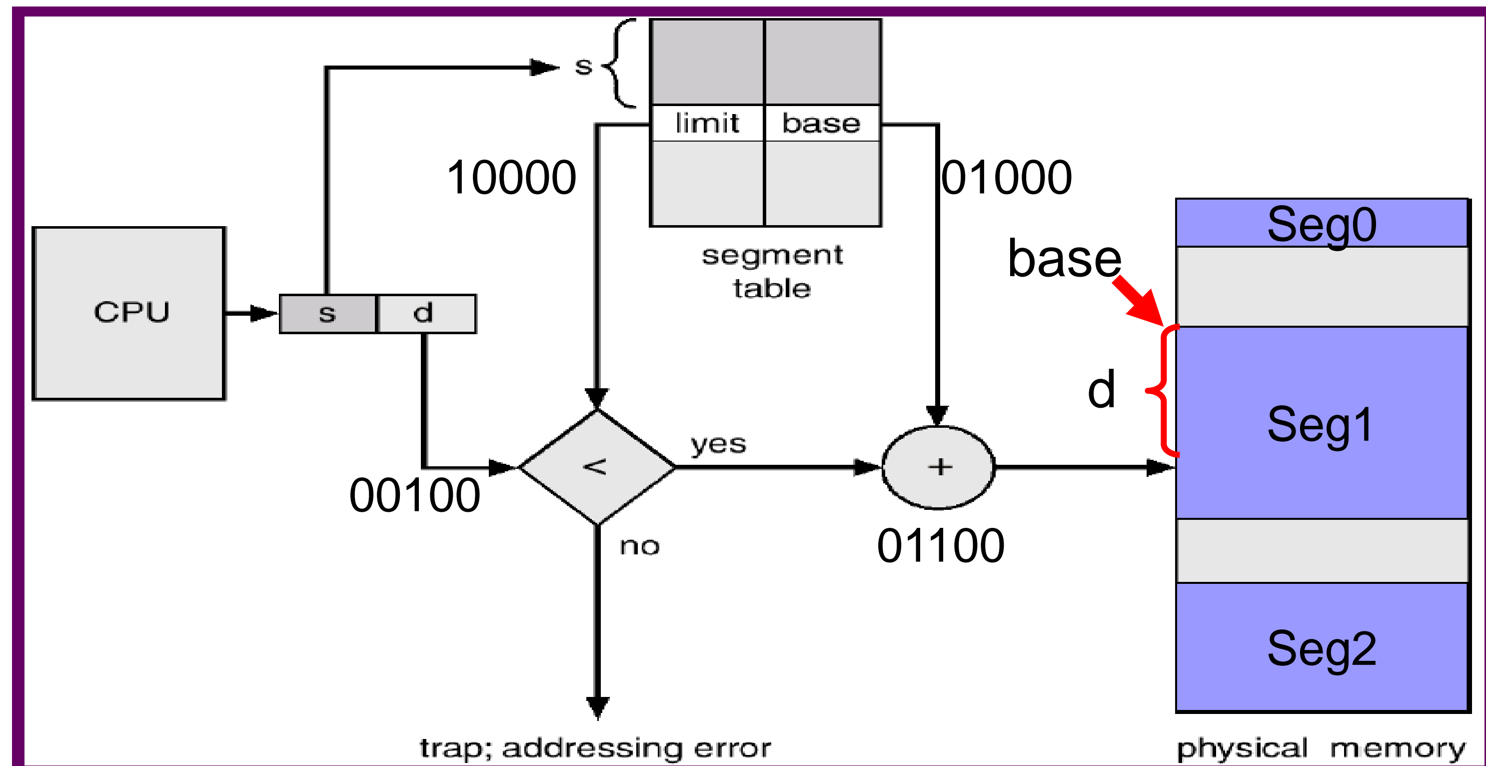


Segmentation Table

- Logical address: (seg#, offset)
  - Offset has the SAME length as physical addr.
- Segmentation table – maps two-dimensional physical addresses; each table entry has:
  - Base (4 bytes): the start physical addr
  - **Limit** (4 bytes): the length of the segment
- Segment-table base register (STBR):
  - the physical addr of the segmentation table
- Segment-table length register (STLR):
  - the # of segments

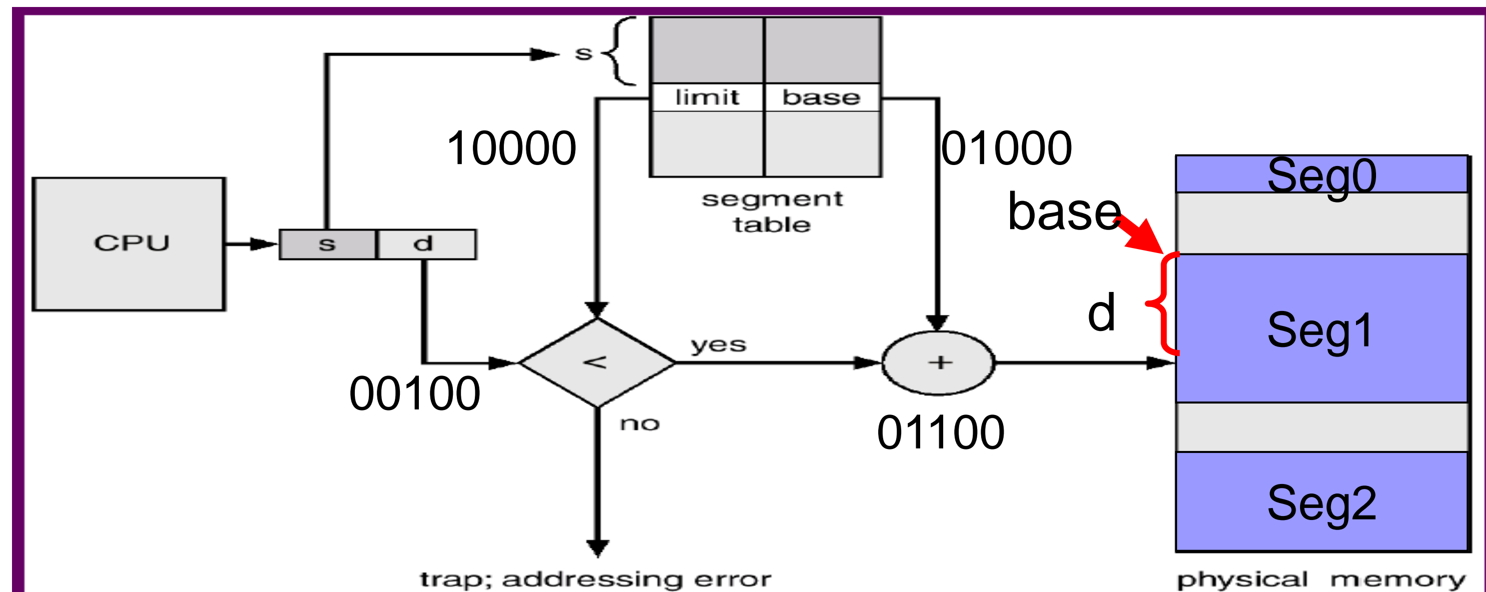
Segmentation Hardware

- Limit register is used to check offset length
- MMU allocate memory by assigning an appropriate base address for each segment
  - Physical address cannot overlap between segments

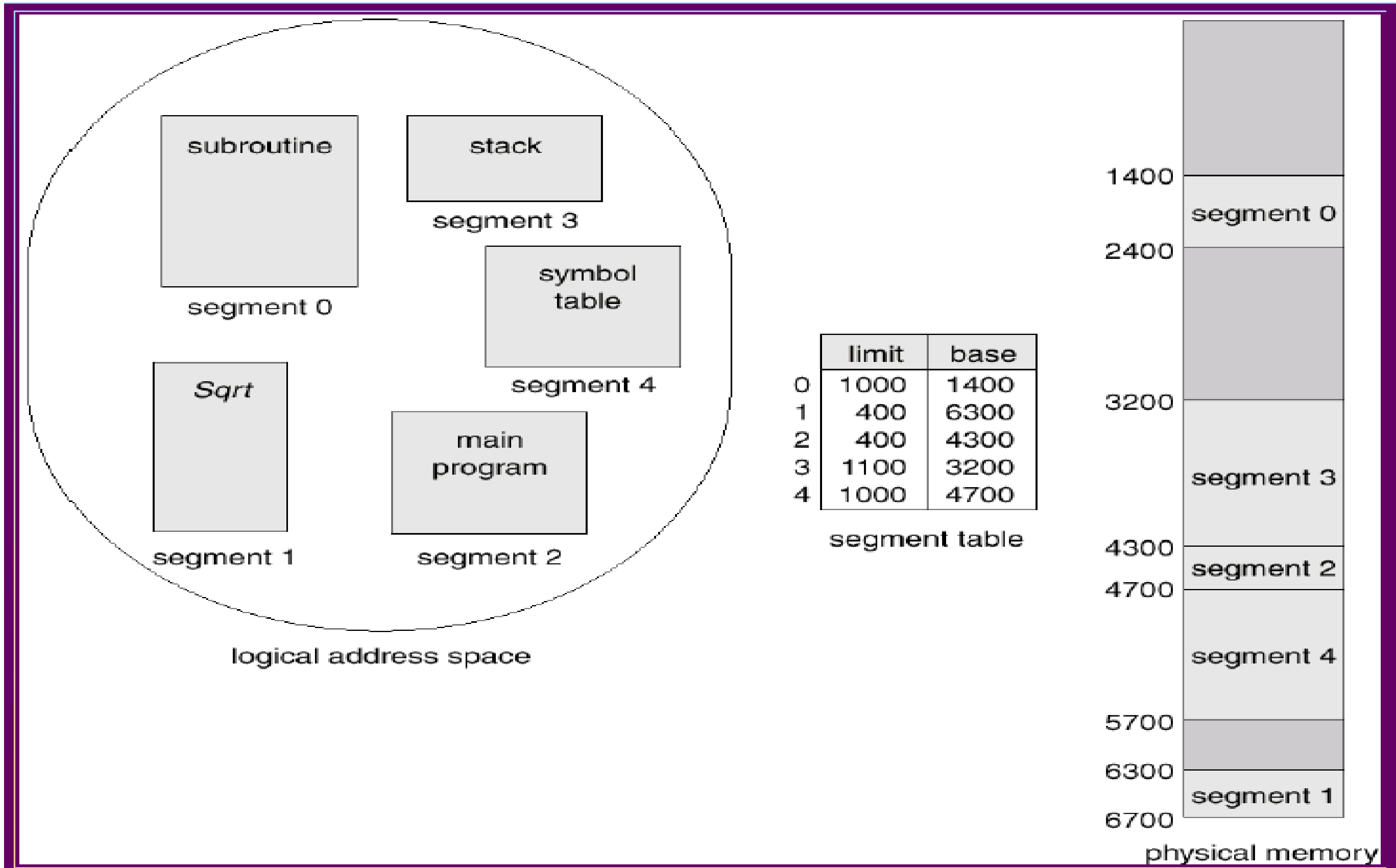


Address Translation Comparison

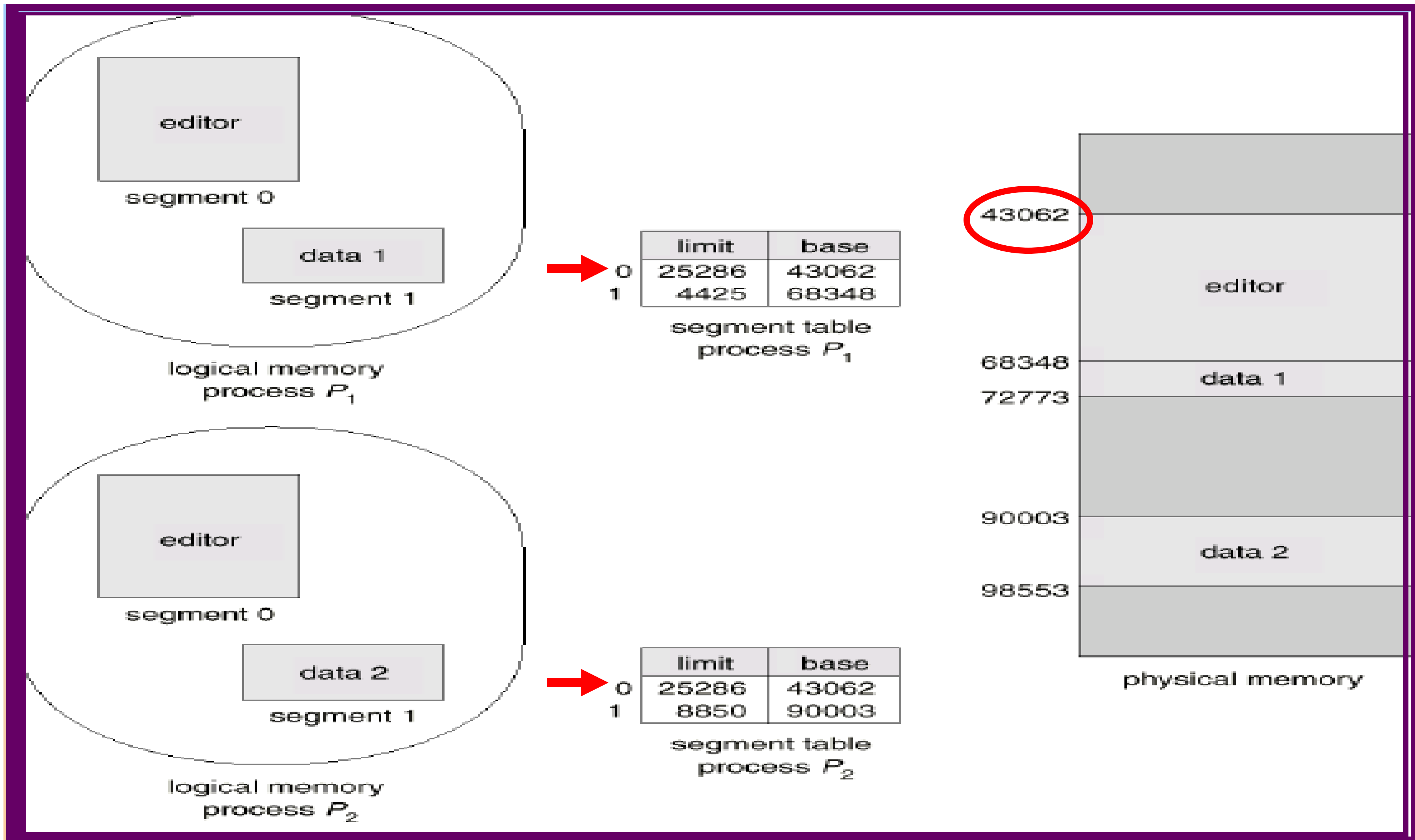
- Segment
  - Table entry: (segment base addr. , **limit**)
  - Segment base addr. can be **arbitrary**
  - The length of “offset” is the same as the **physical memory size**
- Page:
  - Table entry: (frame base addr.)
  - Frame base addr. = frame number * page size
  - The length of “offset” is the same as **page size**



Example of Segmentation



Sharing of Segments



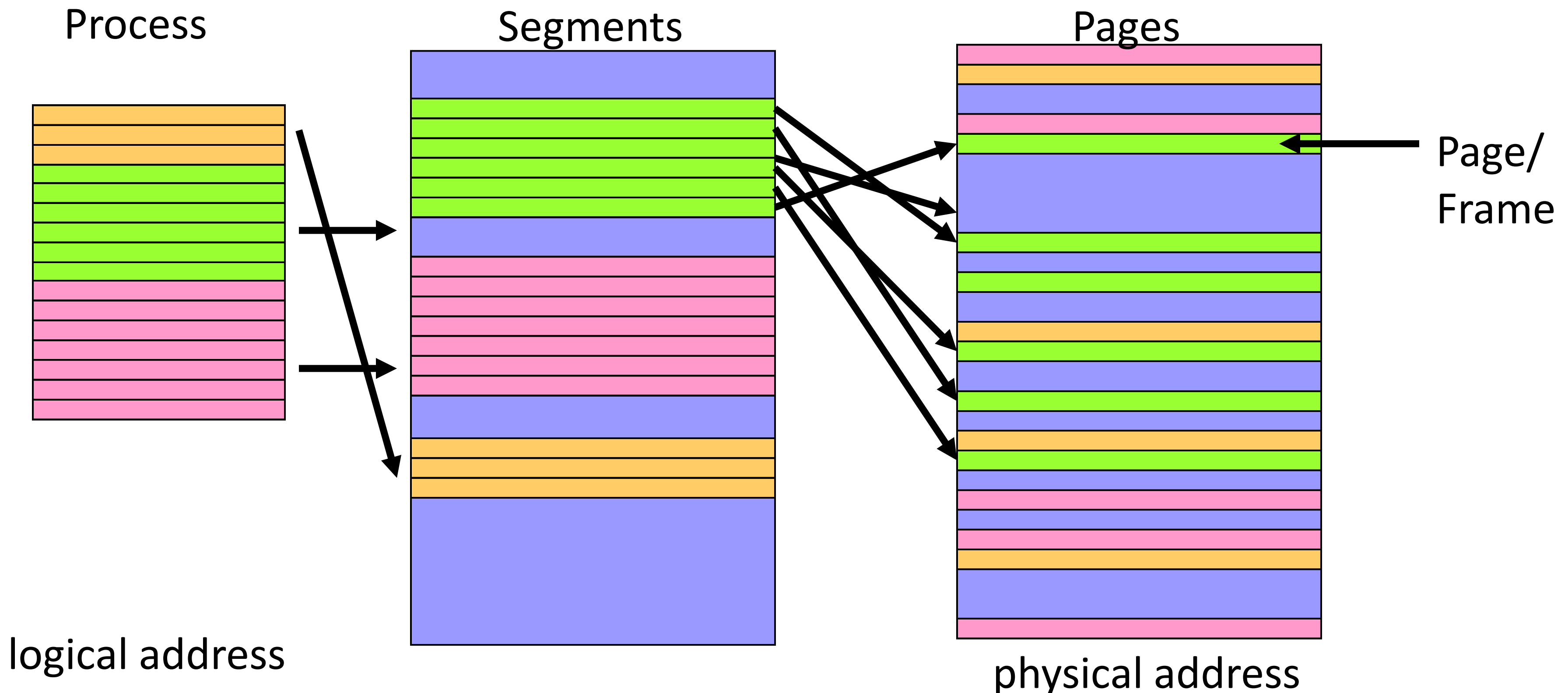
Protection & Sharing

- Protection bits associated with segments
  - Read-only segment (code)
  - Read-write segments (data, heap, stack)
- Code sharing occurs at segment level
  - Shared memory communication
  - Shared library
- Share segment by having same base in two segment tables

Segmentation with
Paging

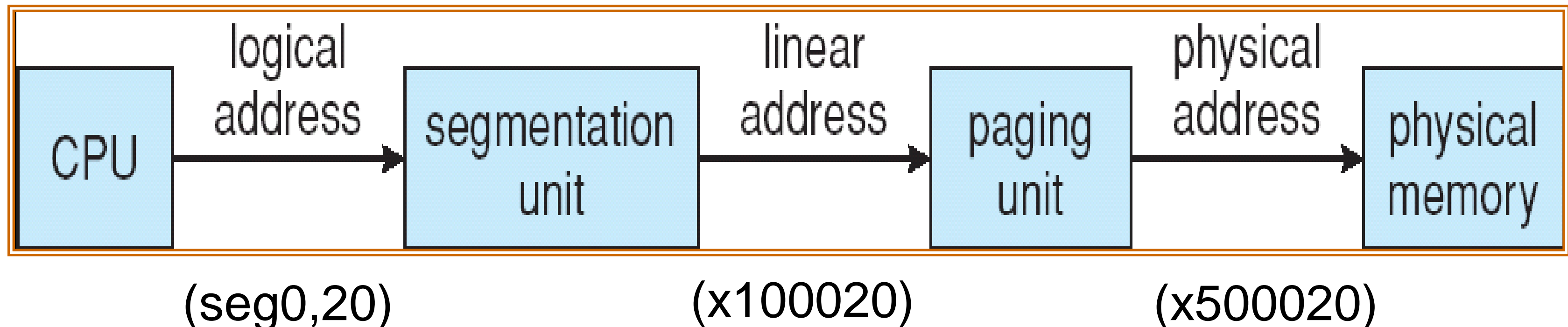
Basic Concept

- Apply segmentation in logical address space
- Apply paging in physical address space



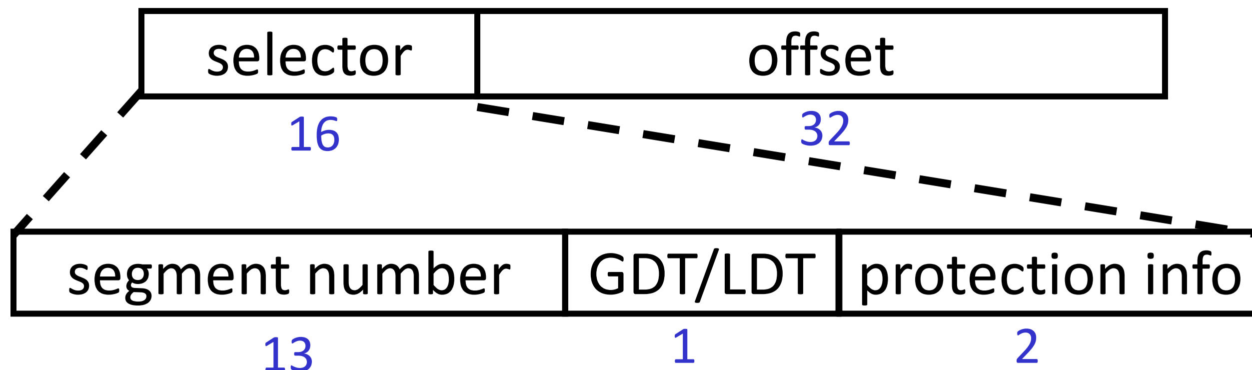
Address Translation

- CPU generates logical address
  - Given to segmentation unit
    - ➔produces linear addresses
  - Linear address given to paging unit
    - ➔generates physical address in main memory
- Segmentation and paging units form equivalent of MMU



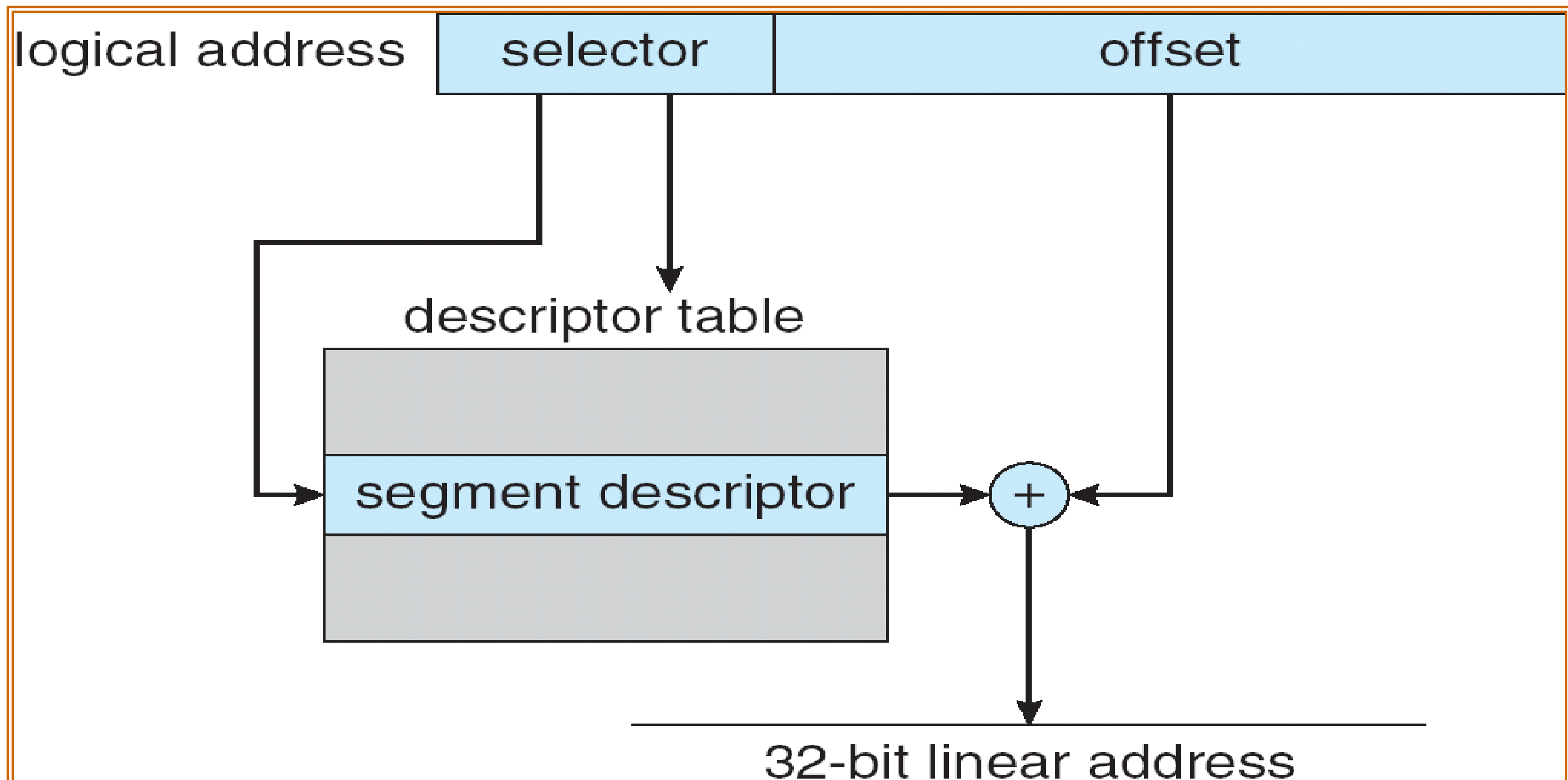
Example: The Intel Pentium

- Logical-address space is divided into 2 partitions:
  - 1st: 8K(2^{13}) segments (private), local descriptor table (LDT)
  - 2nd: 8K(2^{13}) segments (shared), global descriptor table (GDT)
- Logical address:
  - max # of segments per process = 2^{14} = 16K
  - size of a segment $\leq 2^{32}$ = 4GB



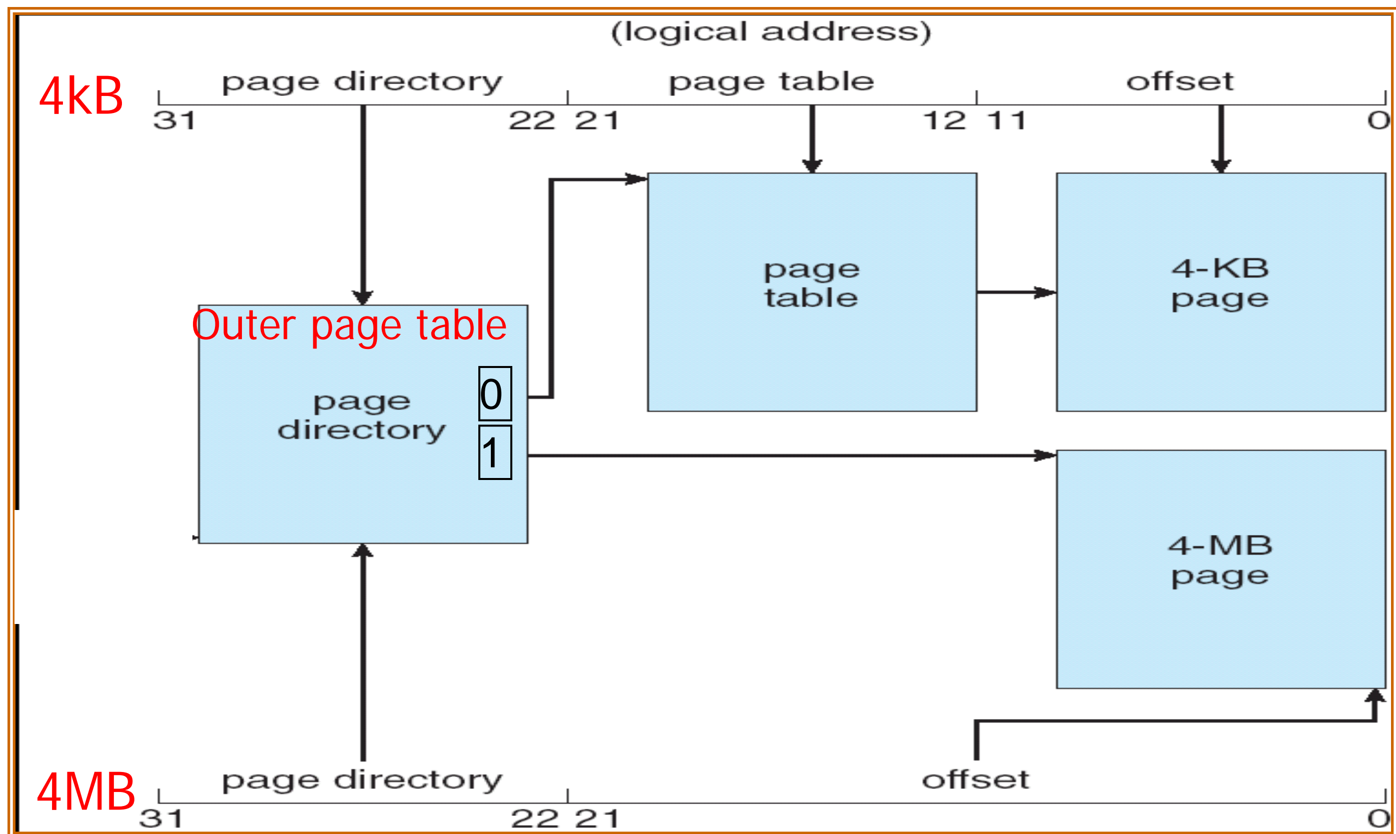
Intel Pentium Segmentation

- Segment descriptor
  - Segment base address and length
  - Access right and privileged level



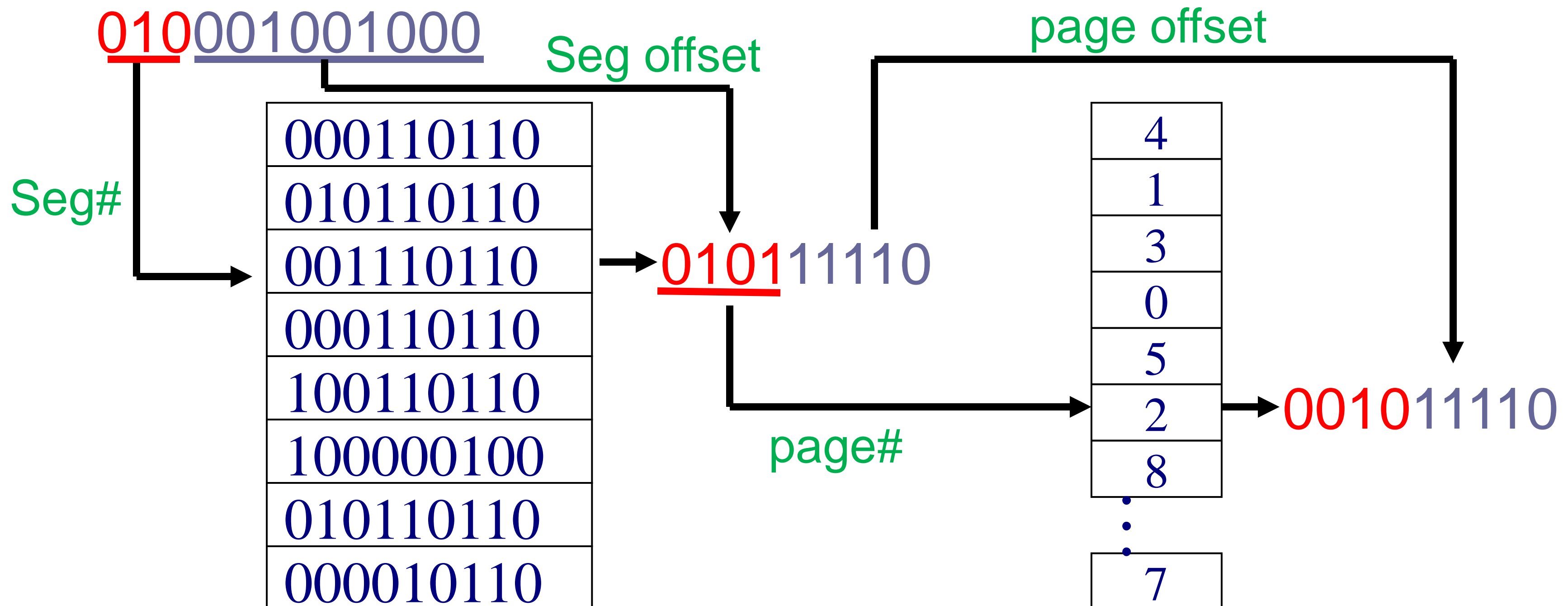
Intel Pentium Paging (Two-Level)

- Page size can be either 4KB or 4MB
  - Each page directory entry has a flag for indication



Example Question

- Let the physical mem size is 512B, the page size is 32B and the logical address of a program can have 8 segments. Given a 12 bits **hexadecimal** logical address "448", translate the addr. With blow page and segment tables.
- linear addr:010111110, phy addr:001011110



Review Slides (5)

- Segmentation vs. Paging?

	Paging	segmentation
Length	Fixed	Varied
Fragmentation	Internal	External
Table entry	Page number → frame number	Seg ID → (base addr, limit length)
View	Physical memory	User program

- Paged segmentation?

Reading Material & HW

- Chap 8
- Problem Set:
  - 8.1, 8.3, 8.4, 8.5, 8.12, 8.15, 8.16, 8.20, 8.23
- Interesting Reading:
  - M. Talluri, M. D. Hill, and Y. A. Khalidi. 1995. A new page table for 64-bit address spaces. *SIGOPS Oper. Syst. Rev.* 29, 5 (December 1995), 184-200.
  - http://pages.cs.wisc.edu/~markhill/papers/sosp95_pagetables.pdf