NATIONAL ECONOMICS UNIVERSITY

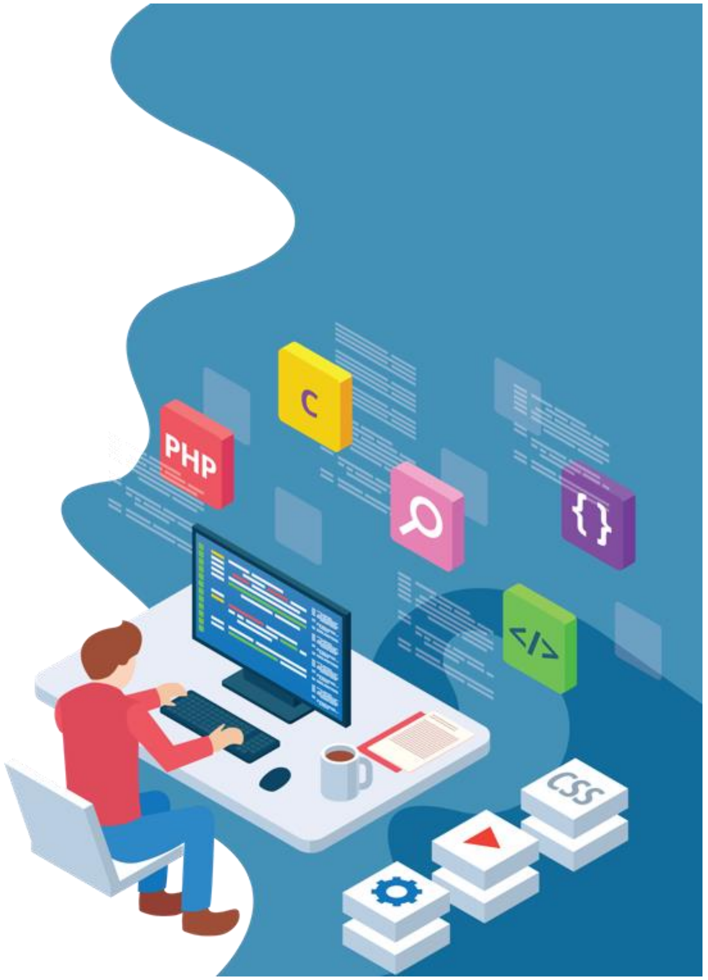
SCHOOL OF INFORMATION TECHNOLOGY AND DIGITAL ECONOMICS

CHAPTER 3

MAIN MEMORY

OUTLINE

- Background
- **Contiguous** Memory Allocation
- **Non-contiguous** Memory Allocation
  - Segmentation
  - Paging
- Virtual Memory



OBJECTIVES

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging
- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames

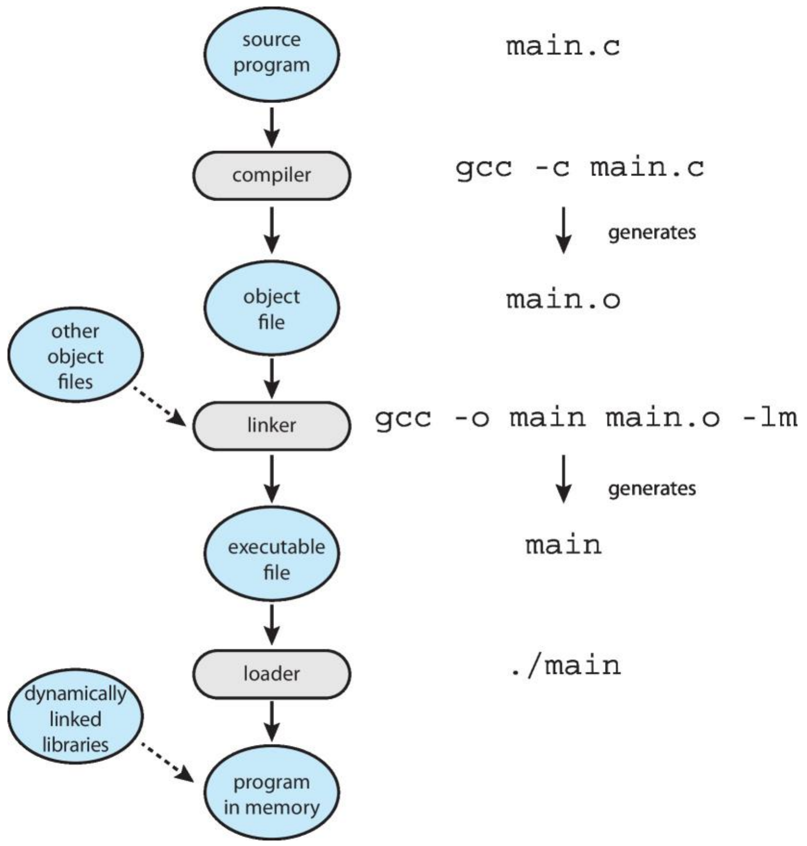
BACKGROUND

- 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
- If the data are not in memory, they must be moved there before the CPU can operate on them.
- Register access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

LINKERS AND LOADERS

- Source code compiled into object files designed to be loaded into any physical memory location – **relocatable object file**
- **Linker** combines these into single binary **executable** file
  - Also brings in libraries
- Program resides on secondary storage as binary executable
- Must be brought into memory by **loader** to be executed
  - **Relocation** assigns final addresses to program parts and adjusts code and data in program to match those addresses
- Modern general purpose systems don't link libraries into executables
  - Rather, **dynamically linked libraries** (in Windows, **DLLs**) are loaded as needed, shared by all that use the same version of that same library (loaded once)
- Object, executable files have standard formats, so operating system knows how to load and start them

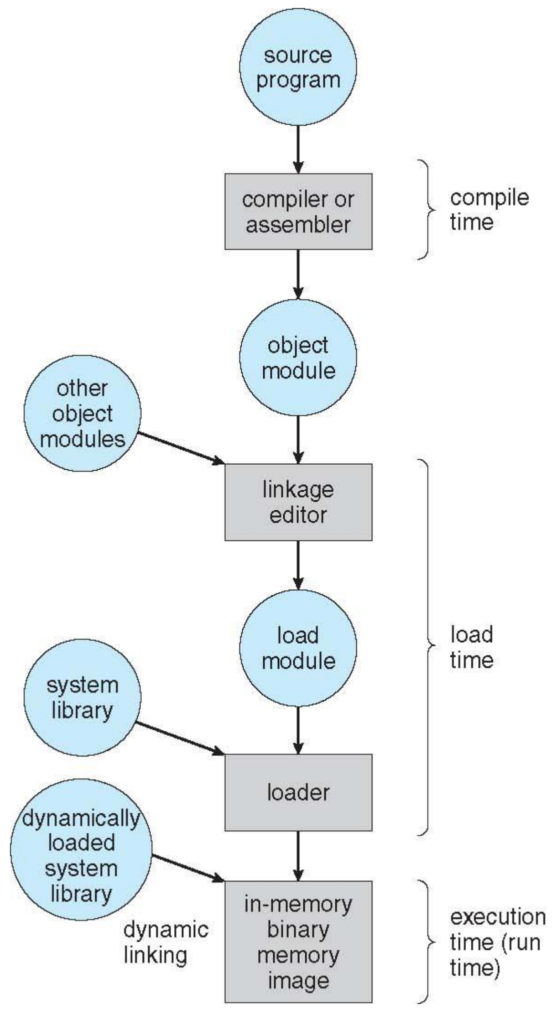
THE ROLE OF THE LINKER AND LOADER



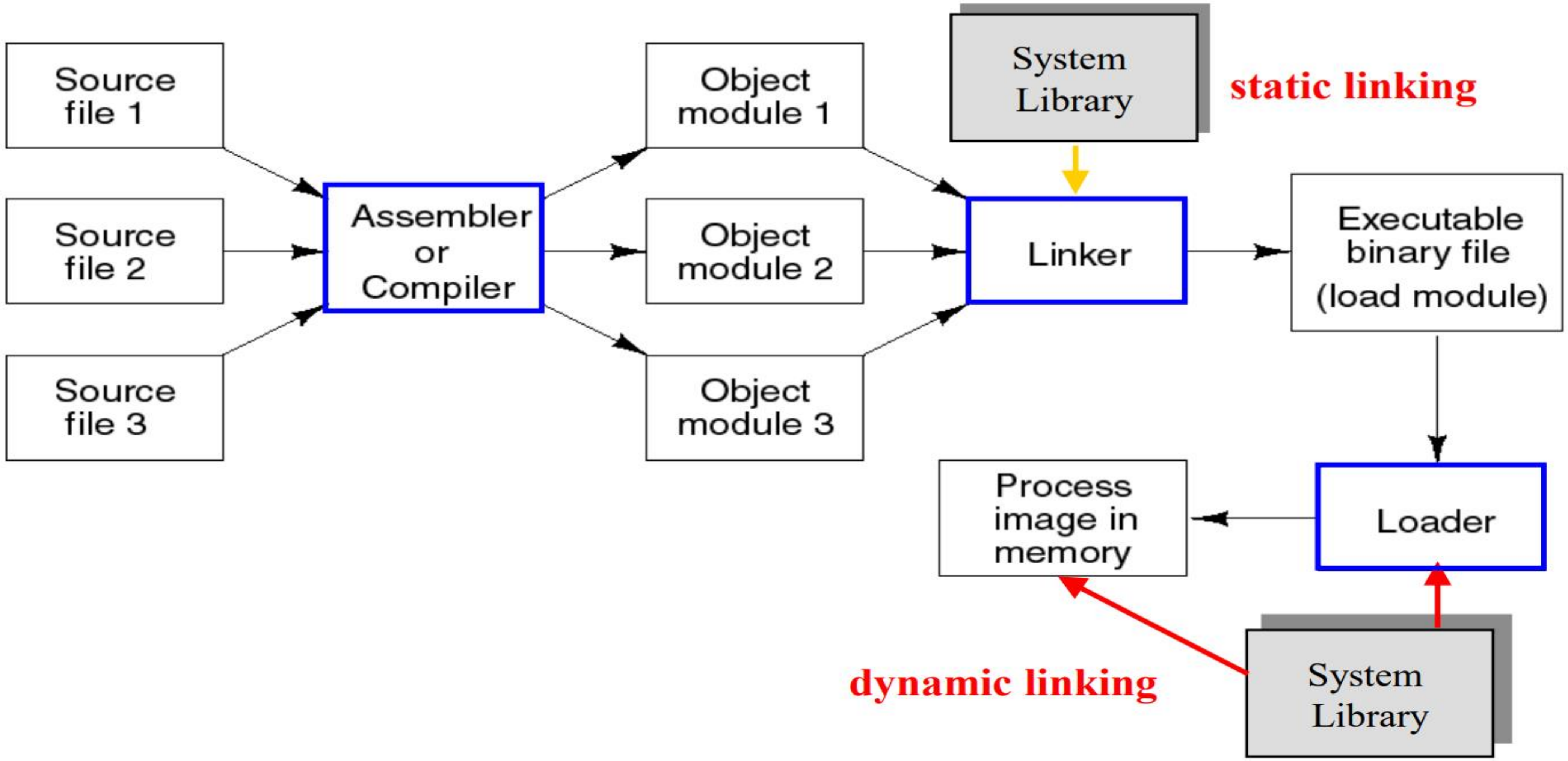
WHY APPLICATIONS ARE OPERATING SYSTEM SPECIFIC

- Apps compiled on one system usually not executable on other operating systems
- Each operating system provides its own unique system calls
  - Own file formats, etc.
- Apps can be multi-operating system
  - Written in interpreted language like Python, Ruby, and interpreter available on multiple operating systems
  - App written in language that includes a VM containing the running app (like Java)
  - Use standard language (like C), compile separately on each operating system to run on each
- **Application Binary Interface** (**ABI**) is architecture equivalent of API, defines how different components of binary code can interface for a given operating system on a given architecture, CPU, etc.

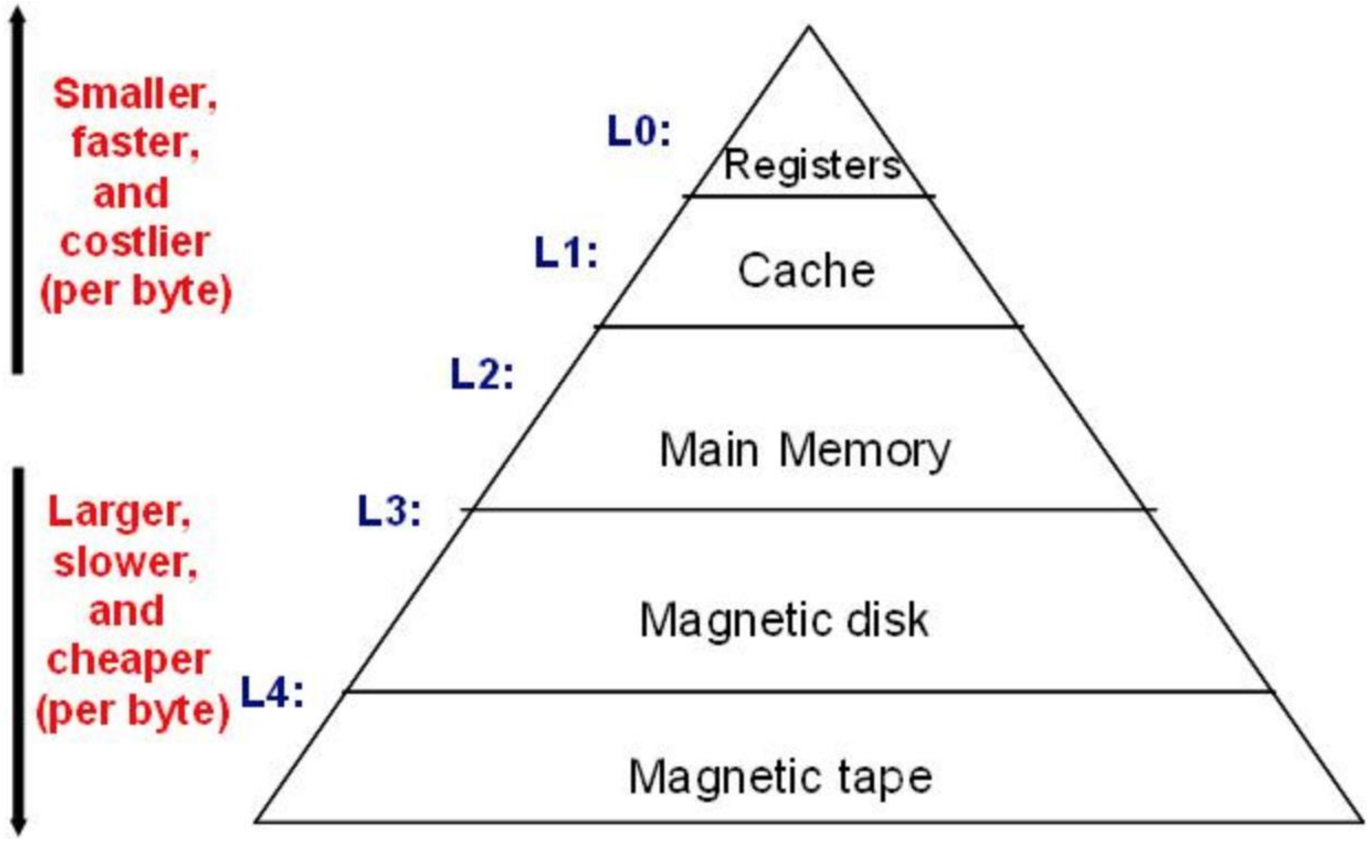
MULTISTEP PROCESSING OF A USER PROGRAM



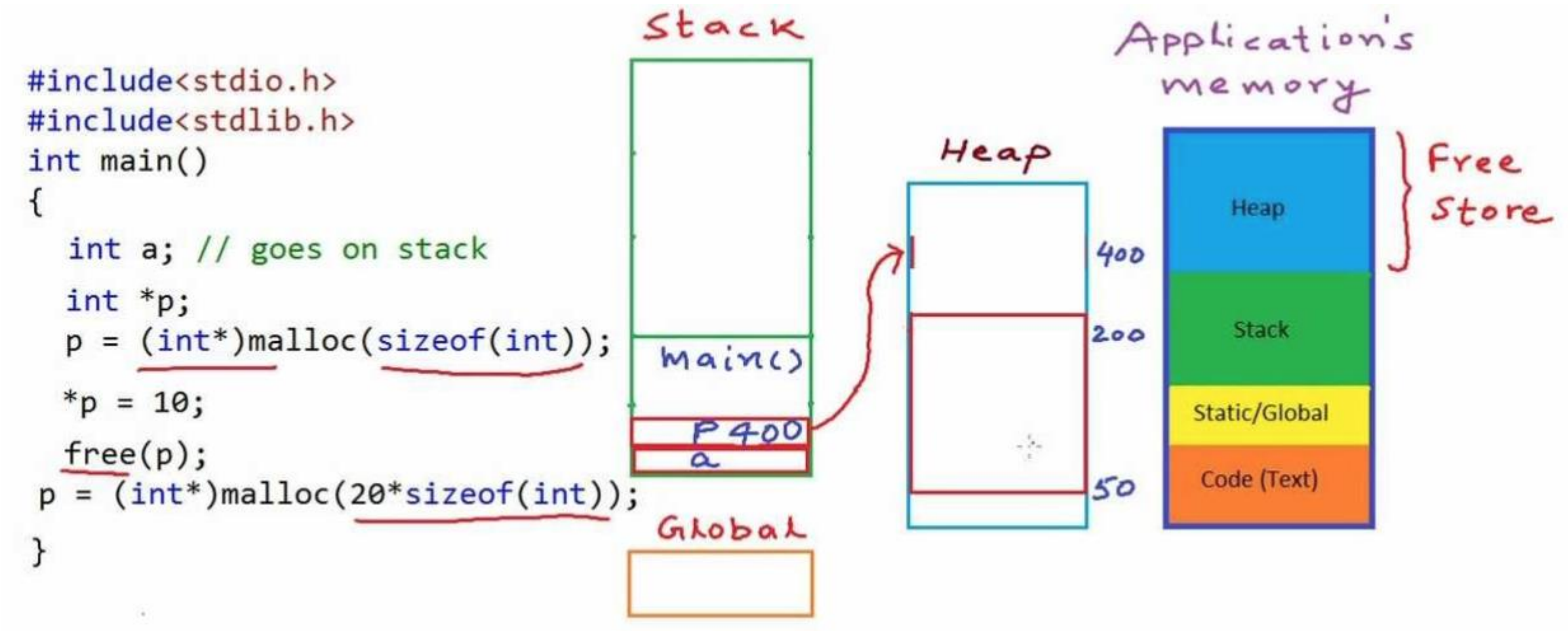
MULTISTEP PROCESSING OF A USER PROGRAM



MEMORY HIERARCHY


Smaller, faster, and costlier (per byte)
Larger, slower, and cheaper (per byte)
L0:
Registers
L1:
Cache
L2:
Main Memory
L3:
Magnetic disk
L4:
Magnetic tape

APPLICATION MEMORY



```
#include<stdio.h>
#include<stdlib.h>
int main()
{
   int a; // goes on stack
   int *p;
   p = (int*)malloc(sizeof(int));
   *p = 10;
   free(p);
  p = (int*)malloc(20*sizeof(int));
}
```

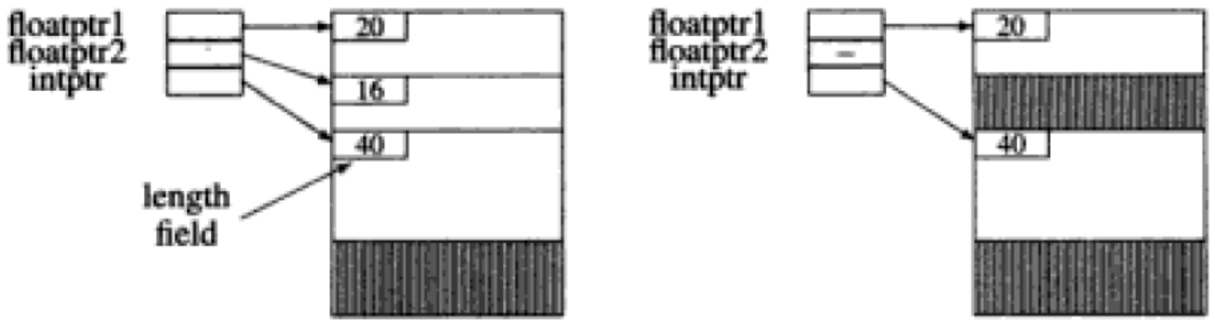
MEMORY ALLOCATION TO A PROCESS

- Stacks:
  - Allocations and deallocations are performed in a LIFO manner.
  - Only the last entry of the stack is accessible at any time
  - A contiguous area of memory is reserved for the stack

MEMORY ALLOCATION TO A PROCESS

- Heap permits allocation and deallocation of memory in a random order

```
float *floatptr1, *floatptr2;
int *intptr;
floatptr1 = (float *) calloc (5, sizeof (float));
floatptr2 = (float *) calloc (4, sizeof (float));
intptr = (int *) calloc (10, sizeof (int));
free (floatptr2);
```



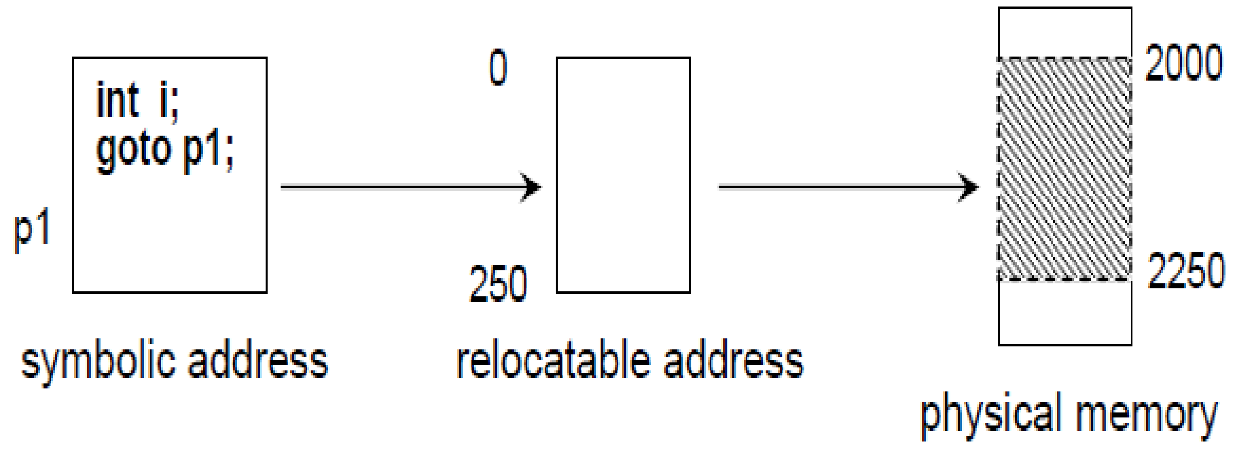
ADDRESS BINDING

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- Inconvenient to have first user process physical address always at 0000
  - How can it not be?
- Further, addresses represented in different ways at different stages of a program's life
  - Source code addresses usually symbolic
  - 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
  - Each binding maps one address space to another

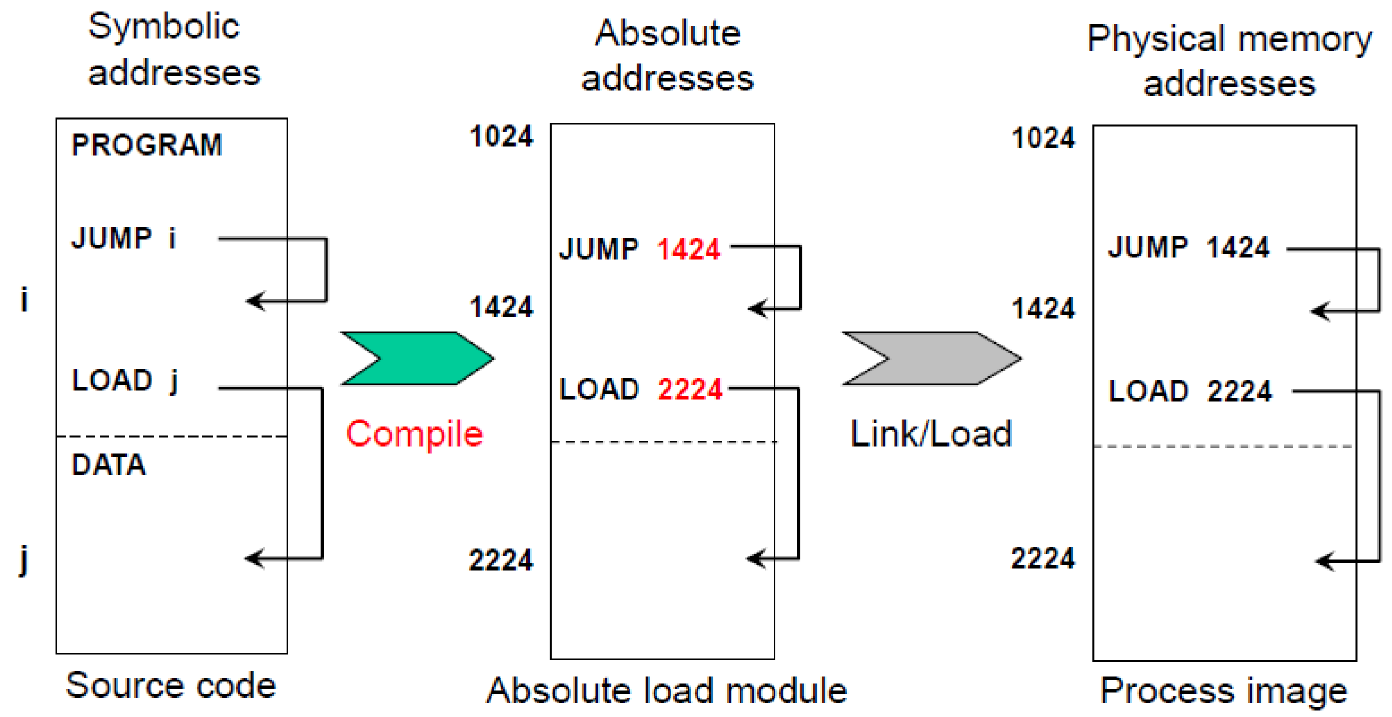
BINDING OF INSTRUCTIONS AND DATA TO MEMORY

- 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)

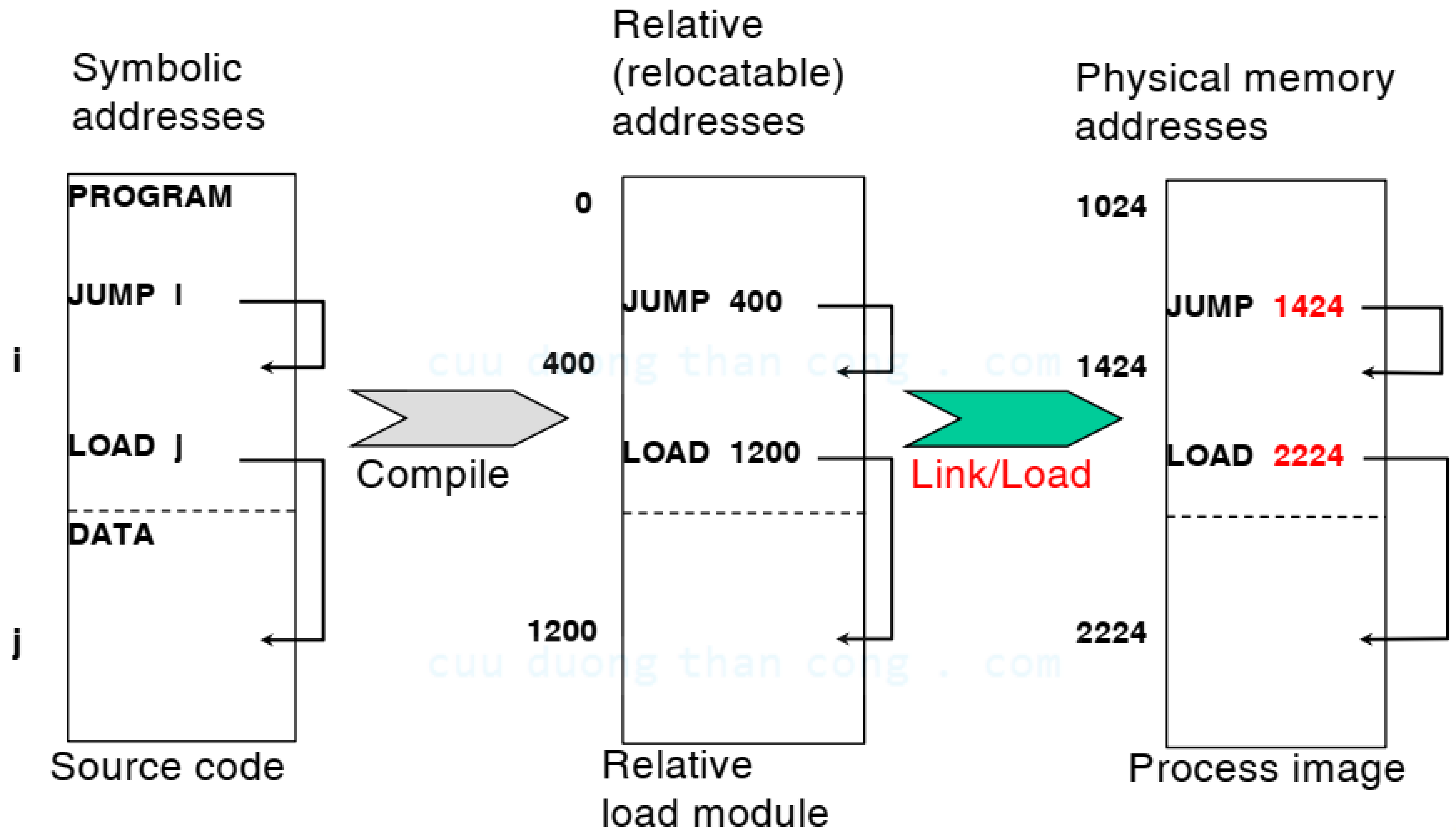
ADDRESS BINDING EXAMPLE



ADDRESS BINDING EXAMPLE – COMPILE TIME



ADDRESS BINDING EXAMPLE – LOAD TIME



LOGICAL VS. PHYSICAL ADDRESS SPACE

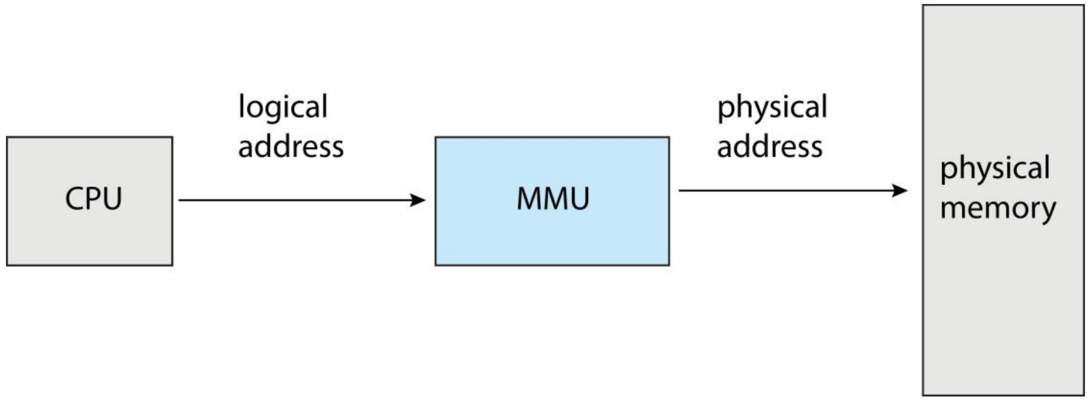
- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as **virtual address**
  - **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by CPU
- **Physical address space** is the set of all physical addresses generated by a program

MEMORY-MANAGEMENT UNIT (MMU)

- Hardware device that at run time 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
  - Base register now called **relocation register**
- The user program deals with *logical* addresses it never sees the *real* physical addresses
  - Execution-time binding occurs when reference is made to location in memory
  - Logical addresses must be mapped to physical addresses before they are used

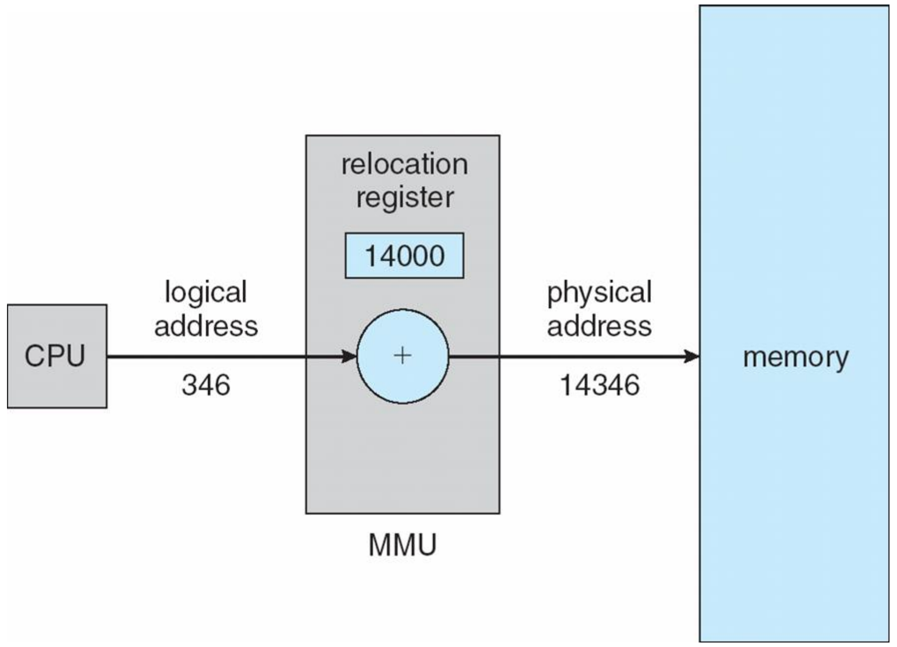
MEMORY-MANAGEMENT UNIT (MMU)

- Hardware device that at run time maps virtual to physical address



MEMORY-MANAGEMENT UNIT (CONT.)

- Consider simple scheme. which is a generalization of the base-register scheme.
- The base register now called relocation register
- The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory



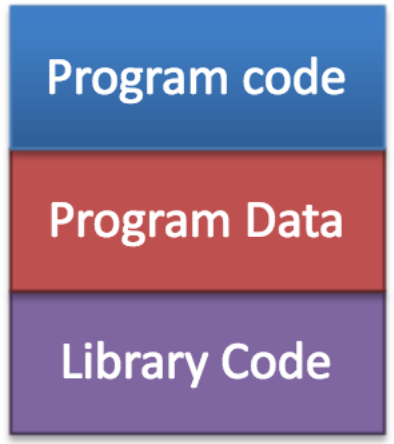
DYNAMIC LOADING

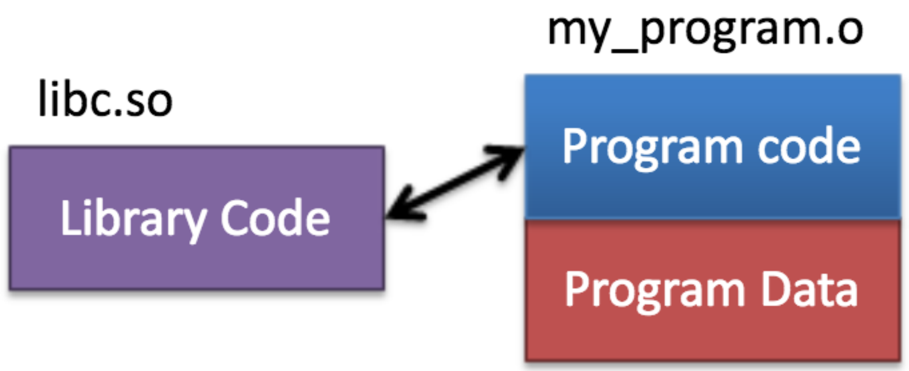
- The entire program does need to be in memory to execute
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required
  - Implemented through program design
  - OS can help by providing libraries to implement dynamic loading

DYNAMIC LINKING

- **Static linking** – system libraries and program code combined by the loader into the binary program image
- **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 checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**
- Consider applicability to patching system libraries
  - Versioning may be needed

STATIC LINKING VS DYNAMIC LINKING



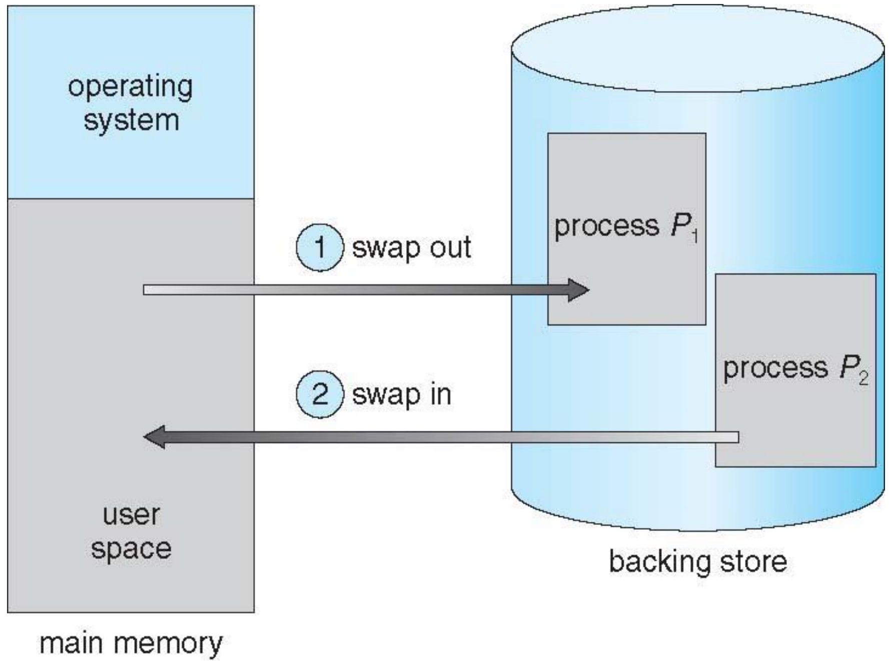


SWAPPING

- A process can be **swapped** temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Swap out, Swap in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- The system maintains a **ready queue** consisting of all processes whose memory images are on the backing store or in memory
- Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped

SWAPPING

- Swapping of two processes using a disk as a backing store.



CONTEXT SWITCH TIME INCLUDING SWAPPING

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - Swap out time of 2 seconds
  - Plus swap in of same sized process => 4 seconds
  - Total context switch swapping component time of 4000ms (4 seconds)
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - System calls to inform OS of memory use via `request_memory()` and `release_memory()`

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. Data that have been modified (such as the stack) are never removed
    - Failure to free can result in termination
  - Android terminates apps if low free memory, but first writes **application state** to flash for fast restart

FRAGMENTATION

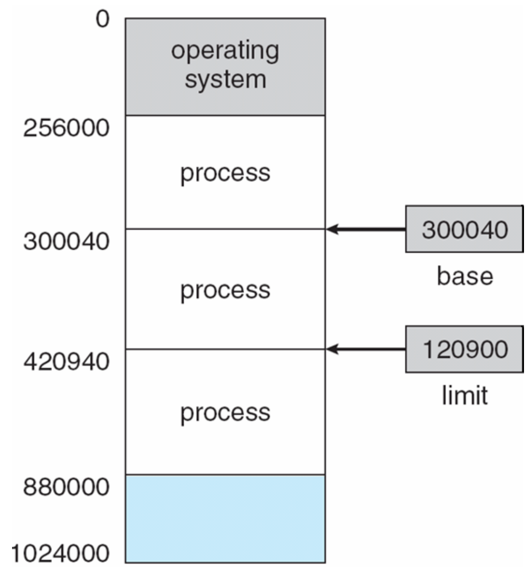
- External fragmentation: Total sufficient quantity of area within the memory to satisfy the memory request of a method. however the process's memory request cannot be fulfilled because the memory offered is during a non-contiguous manner
- Internal fragmentation: when the memory is split into mounted sized blocks. The mounted sized block is allotted to the method. The memory allotted to the method is somewhat larger than the memory requested, then the distinction between allotted and requested memory is that the Internal fragmentation.

REQUIREMENTS FOR MEMORY MANAGEMENT

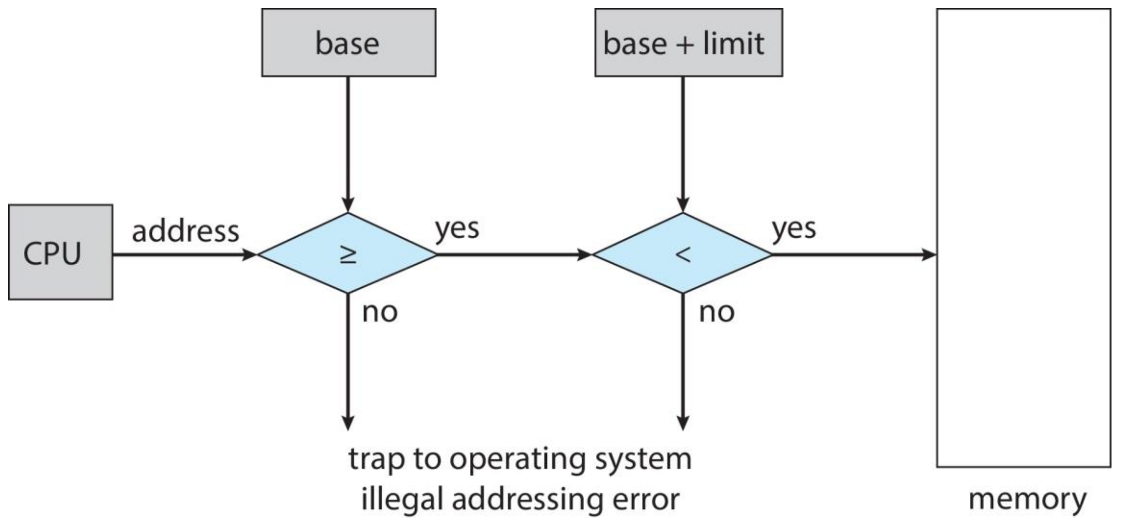
- Relocation
- Protection
- Sharing
- Logical organization
- Physical organization

PROTECTION (BASE & LIMIT)

- A pair of **base** and **limit registers** define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user

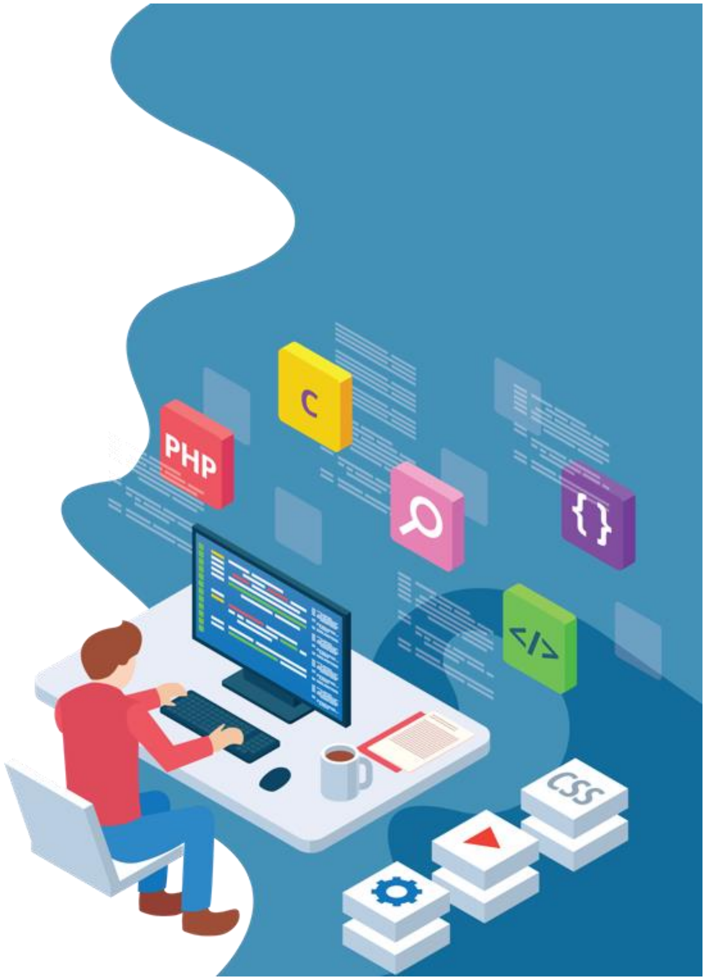


HARDWARE ADDRESS PROTECTION



OUTLINE

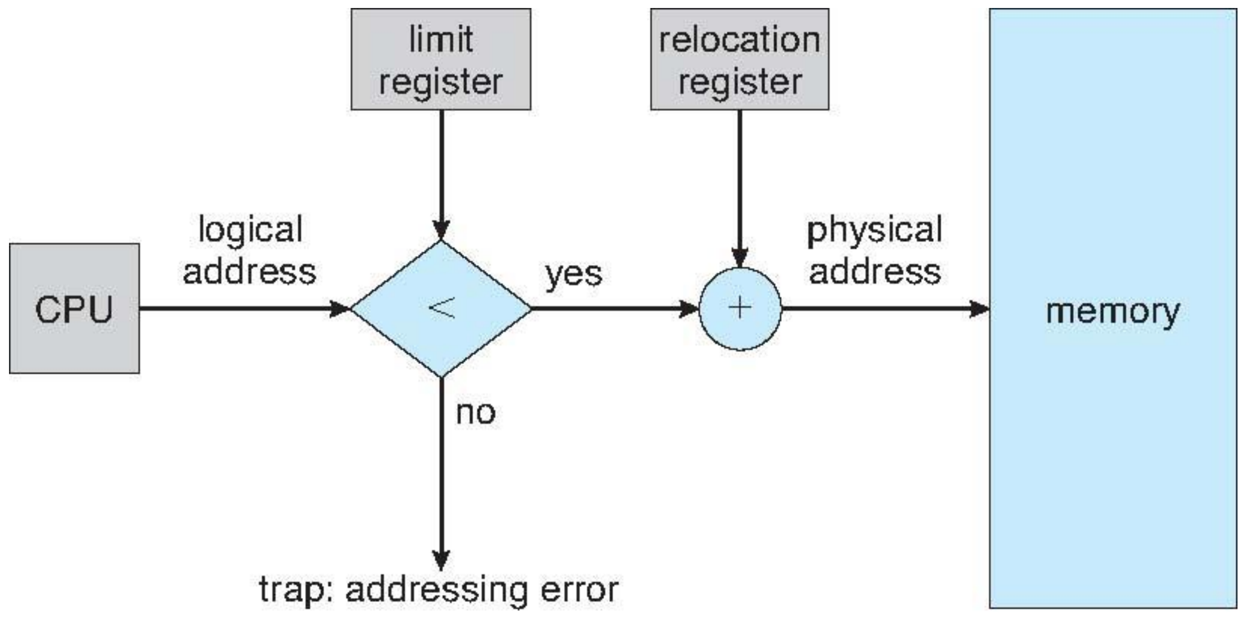
- Background
- **Contiguous** Memory Allocation
- **Non-contiguous** Memory Allocation
  - Segmentation
  - Paging
- Virtual Memory



CONTIGUOUS ALLOCATION

- Main memory must support both **OS** and **user processes**
- Limited resource, must allocate **efficiently**
- **Contiguous allocation** is one early method
- 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 register** contains the **smallest physical address** value; **limit register** contains **range of logical addresses** - each logical address must be less than **limit register**

HARDWARE SUPPORT FOR RELOCATION AND LIMIT REGISTERS

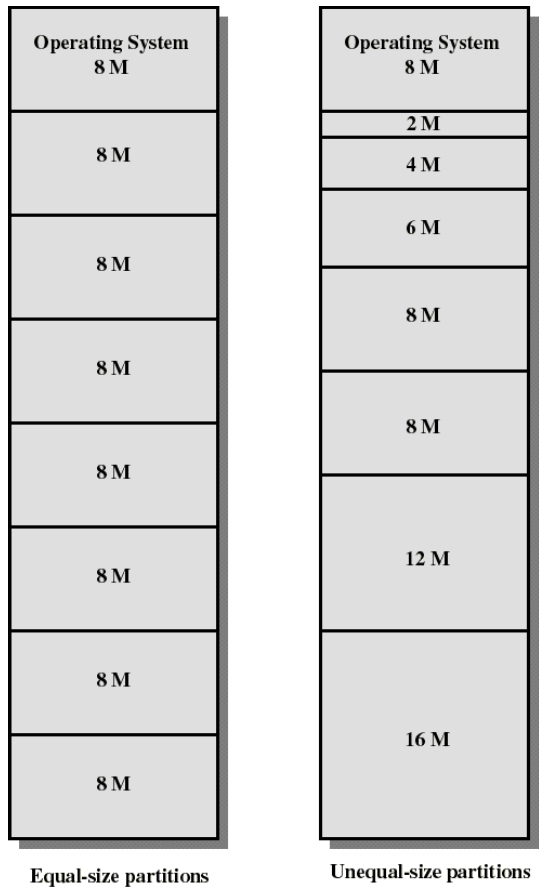


CONTIGUOUS ALLOCATION (CONT.)

- 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*
  - If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code;

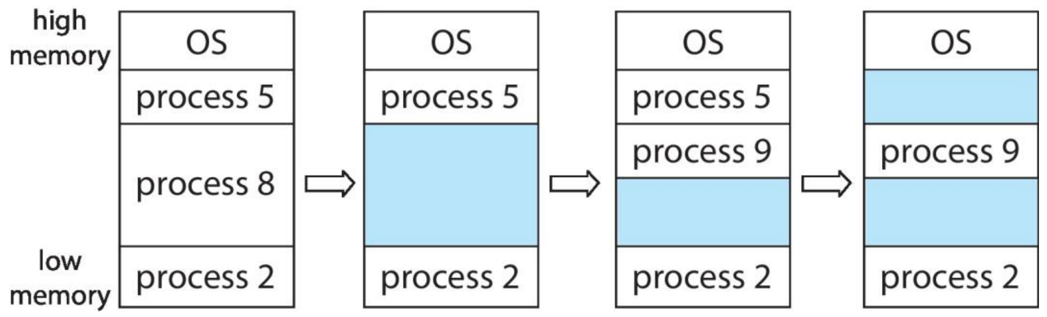
MEMORY ALLOCATION

- One of the simplest methods for allocating memory is to divide memory into several **fixed-sized partitions**
  - Degree of multiprogramming limited by number of partitions
  - When a partition is **free**, a process is selected from the input queue and is loaded into the free partition.
  - When the process **terminates**, the partition becomes available for another process.
  - This method was originally used by the IBM OS/360 operating system (called MFT) but is no longer in use.

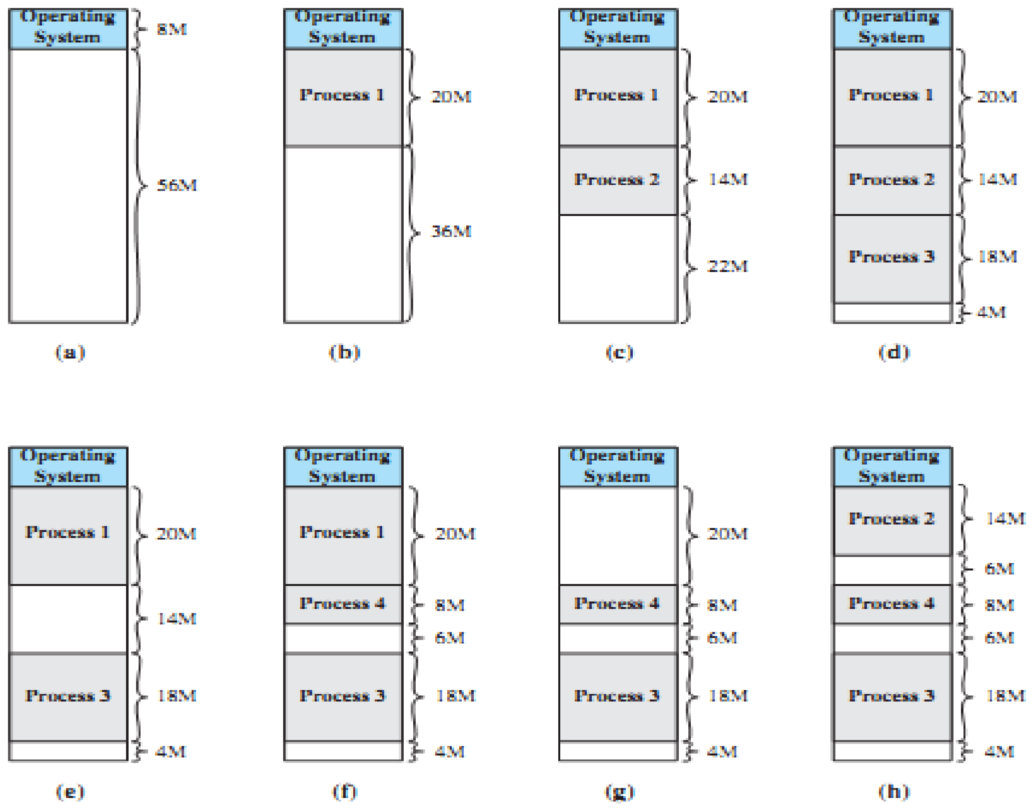


MEMORY ALLOCATION (VARIABLE PARTITION)

- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **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
  - Process exiting frees its partition, adjacent free partitions combined
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)

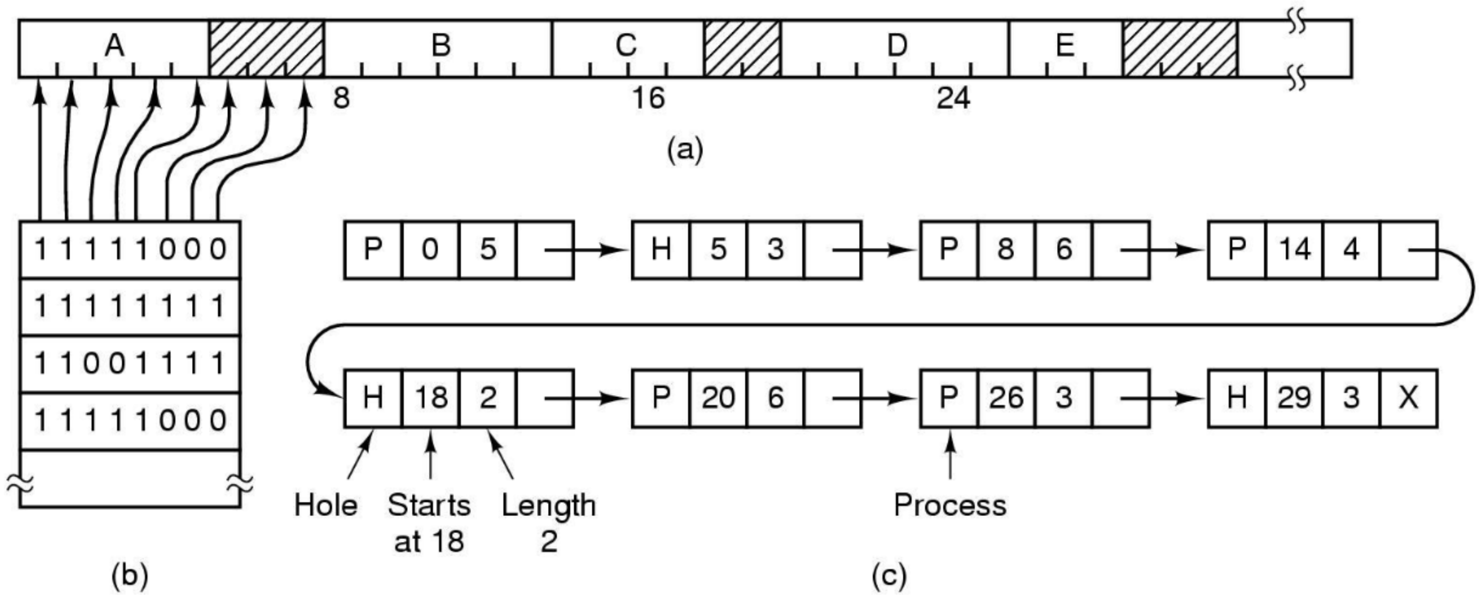


MULTIPLE-PARTITION ALLOCATION

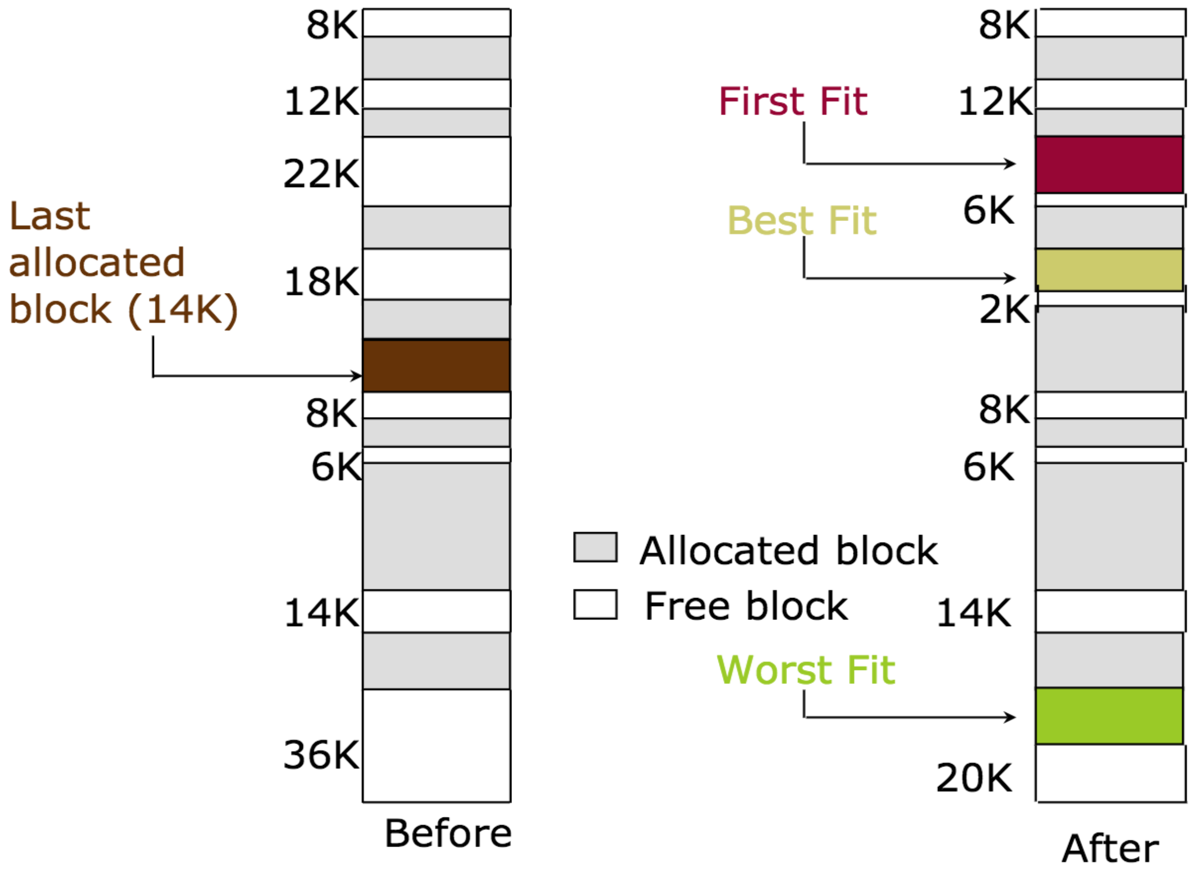


TWO WAYS TO TRACK MEMORY USAGE

- Bit map vs Linked list



DYNAMIC STORAGE-ALLOCATION



Example Memory Configuration
Allocation of 16 KB Block

MEMORY ALLOCATION -DYNAMIC STORAGE

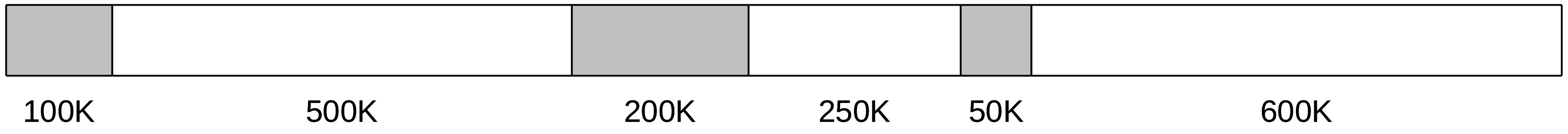
- How to satisfy a request of size ***n*** from a list of free holes?
  - **First-fit**: Allocate the ***first*** hole that is big enough
  - **Best-fit**: Allocate the ***smallest*** hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - **Worst-fit**: Allocate the ***largest*** hole; must also search entire list
    - Produces the largest leftover hole

EXAMPLE 1

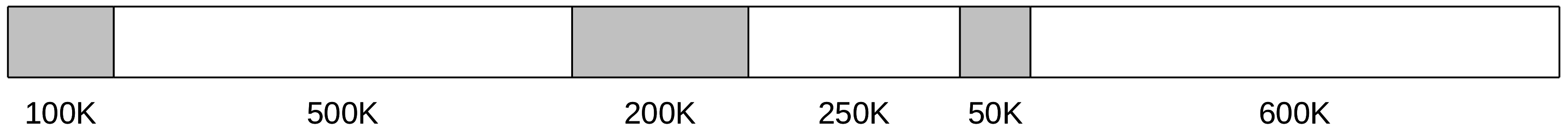
- Assume main memory is divided into partitions of size size is 600K, 500K, 200K, 300K (in order), progress with sizes 212K, 417K, 112K and 426K (in that order) will. How is memory allocated, if using :
  - **First-fit**
  - **Best-fit**
  - **Worst-fit**

EXAMPLE 2

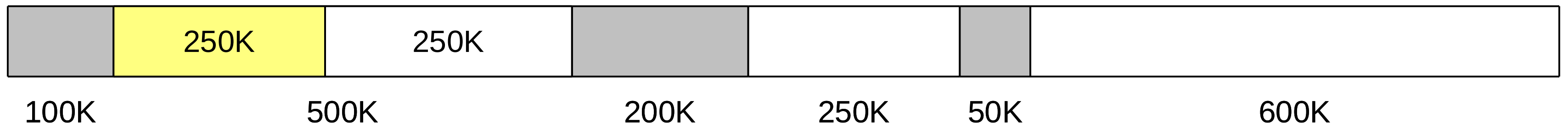
- A dynamic partitioning scheme is being used, and the following is the memory configuration at a given point in time. The shaded areas are allocated blocks and the white areas are free blocks. Assume next four memory request are for **250K, 419K**, **205K** and **330K**. Indicate the starting address (position on the given memory configuration) for each of the four blocks using the following placement algorithms.
  - First-fit
  - Best-fit
  - Worst-fit



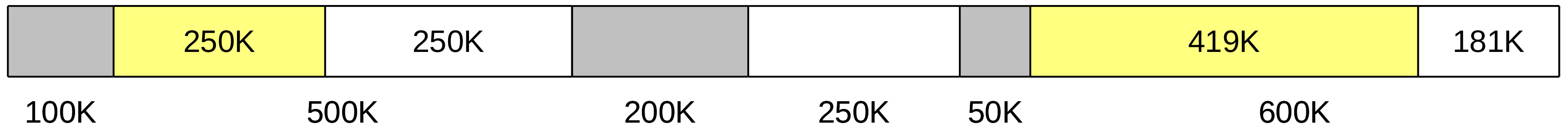
FIRST-FIT



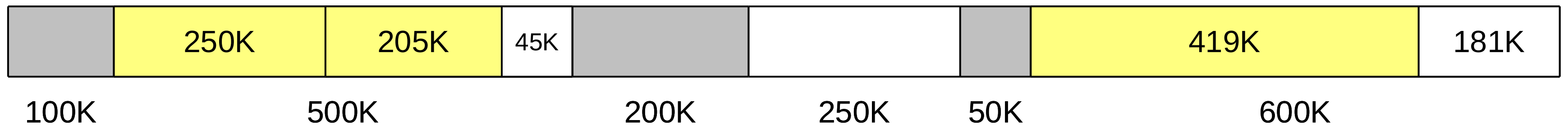
Allocation of 250K



Allocation of 419K

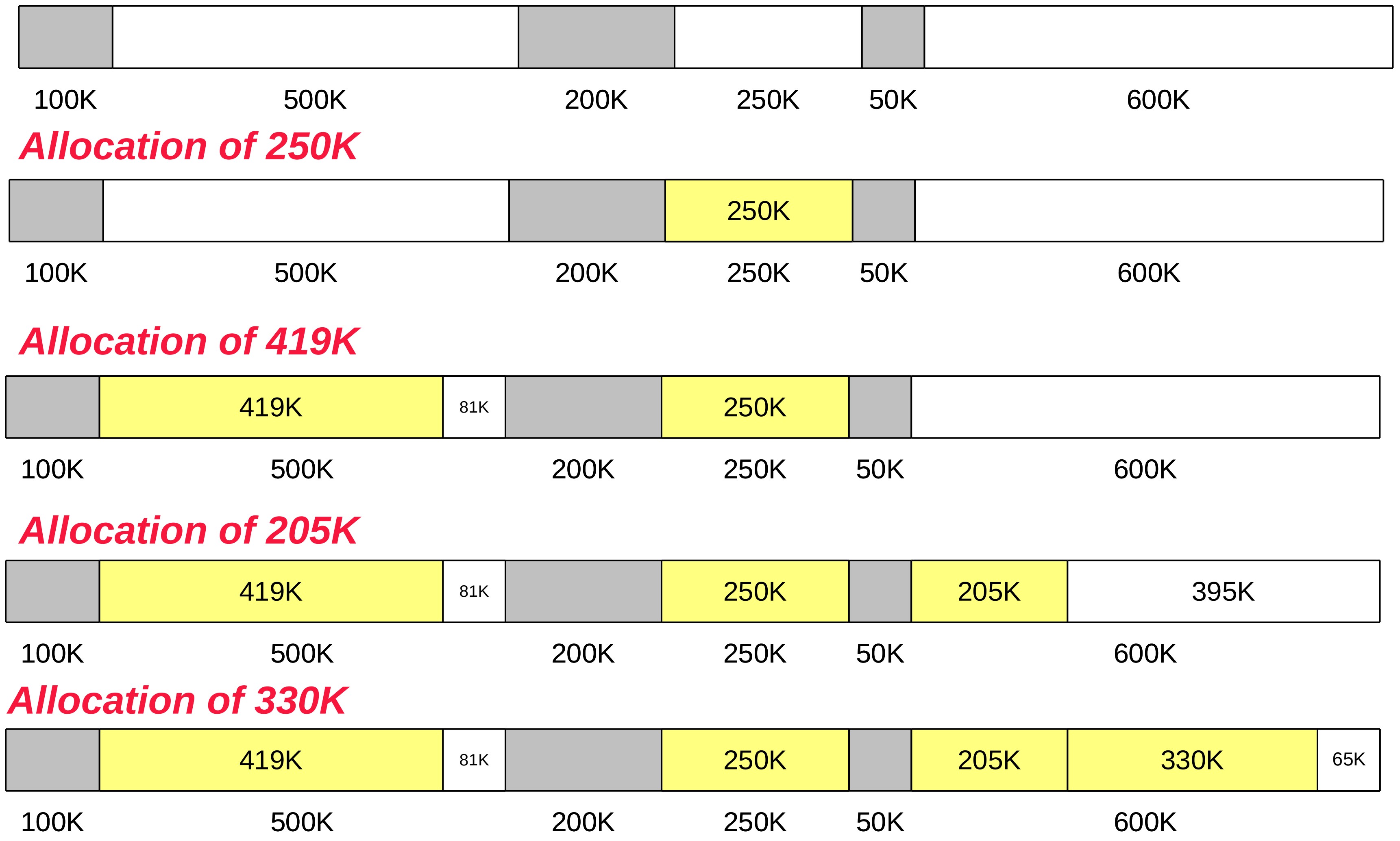


Allocation of 205K



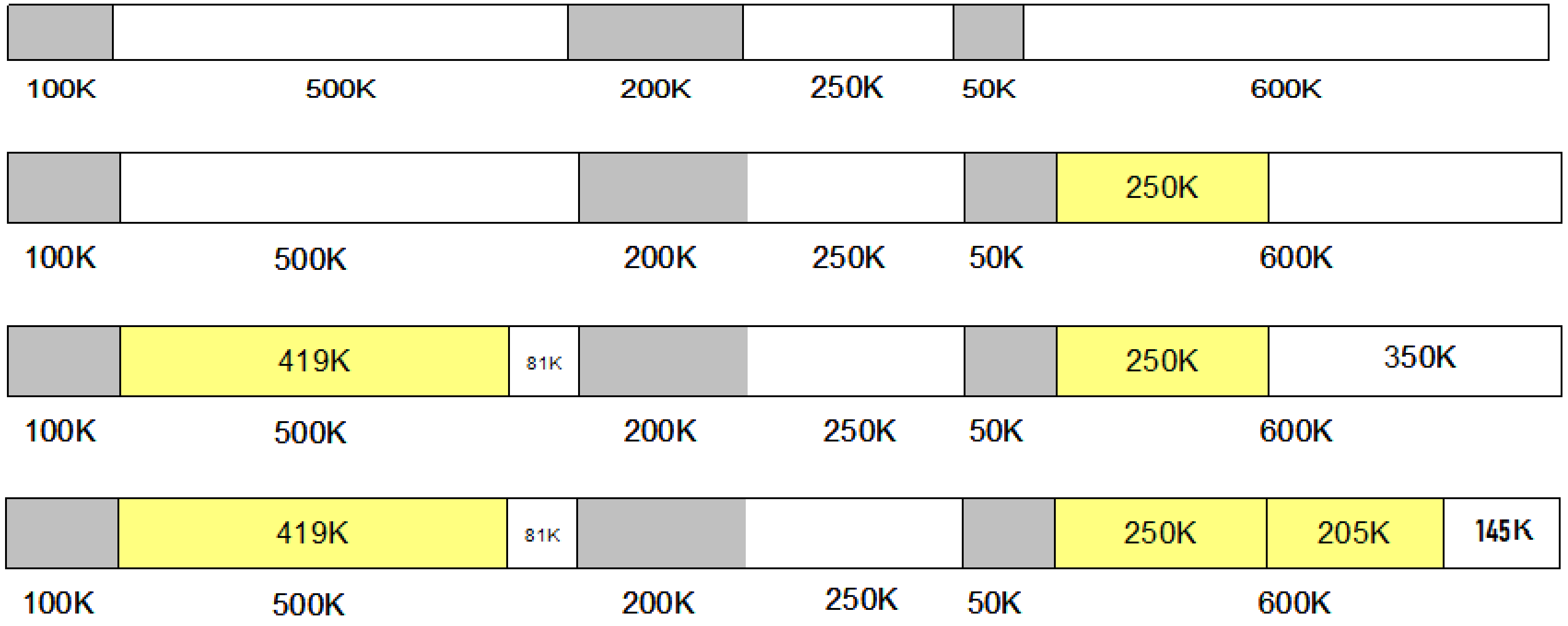
Last Allocation for 330K fails!!!

BEST-FIT



100K
500K
200K
250K
50K
600K
Allocation of 250K
250K
100K
500K
200K
250K
50K
600K
Allocation of 419K
419K
81K
250K
100K
500K
200K
250K
50K
600K
Allocation of 205K
419K
81K
250K
205K
395K
100K
500K
200K
250K
50K
600K
Allocation of 330K
419K
81K
250K
205K
330K
65K
100K
500K
200K
250K
50K
600K

WORST-FIT

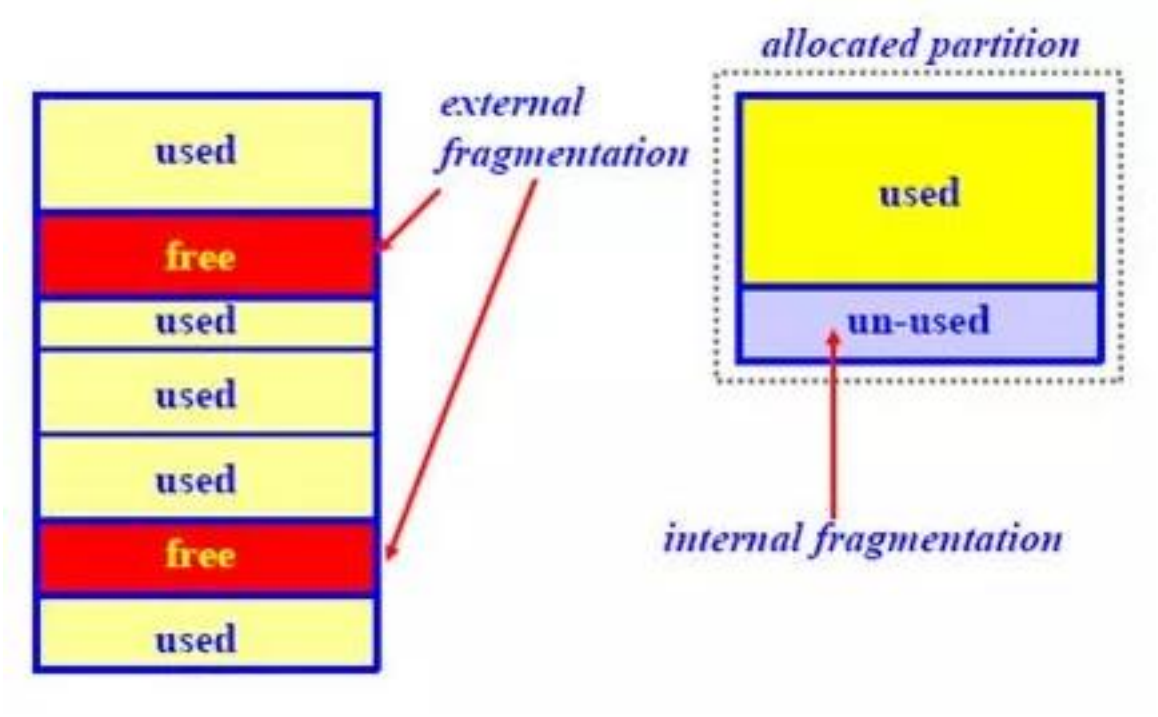


Last Allocation for 330K fails!!!

FRAGMENTATION

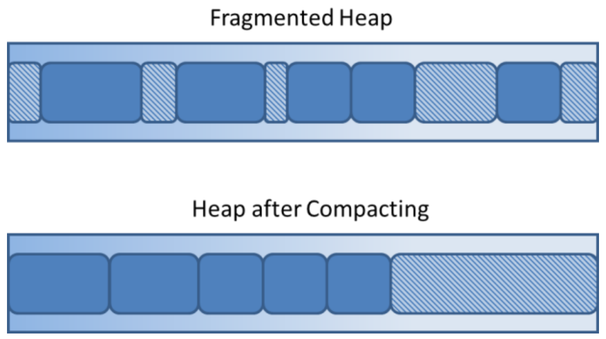
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - 1/3 may be unusable -> **50-percent rule**

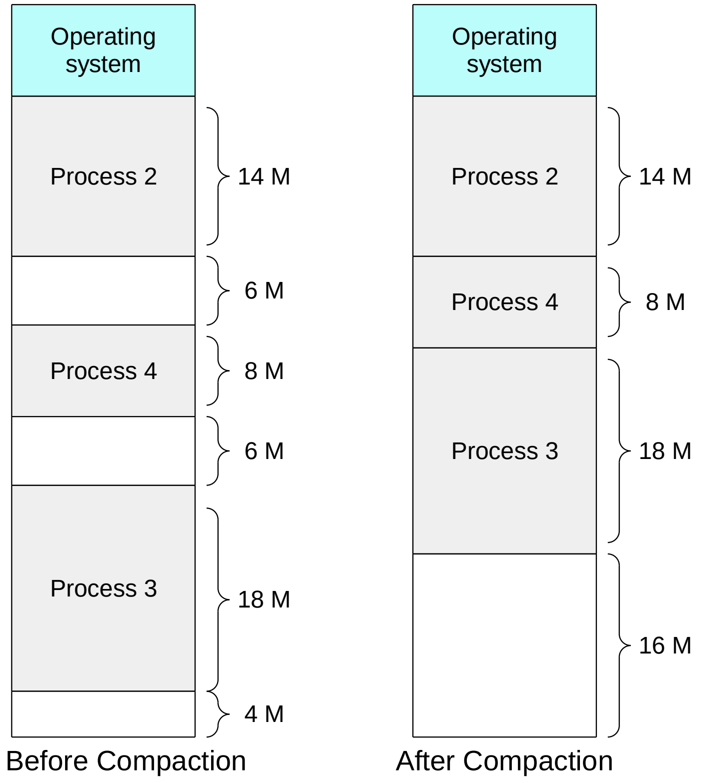
FRAGMENTATION



COMPACTION

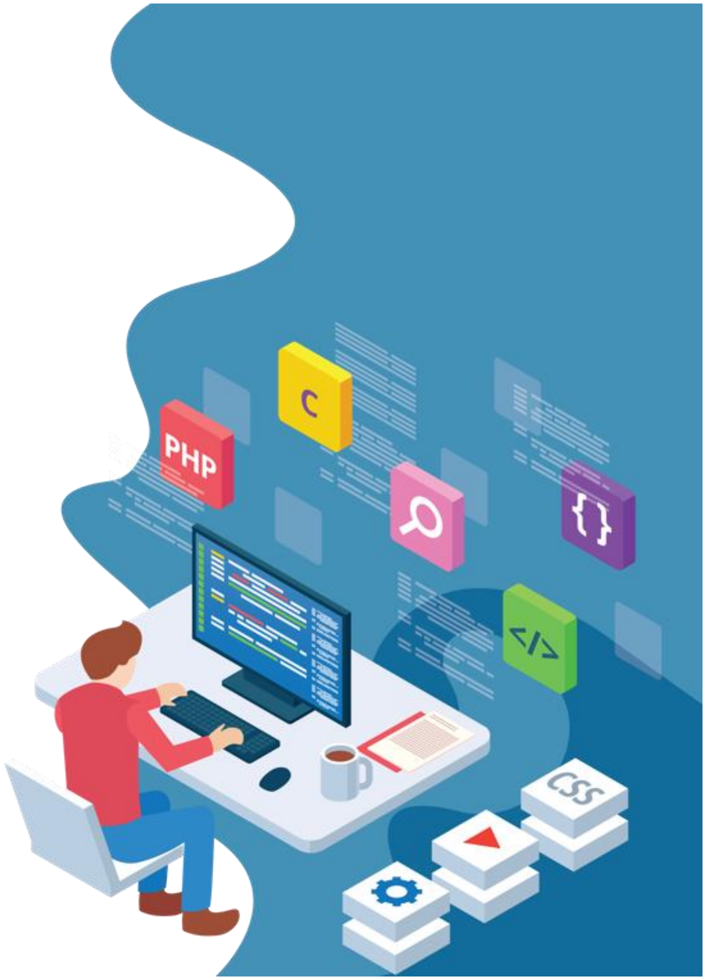
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic
  - I/O problem:
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers





OUTLINE

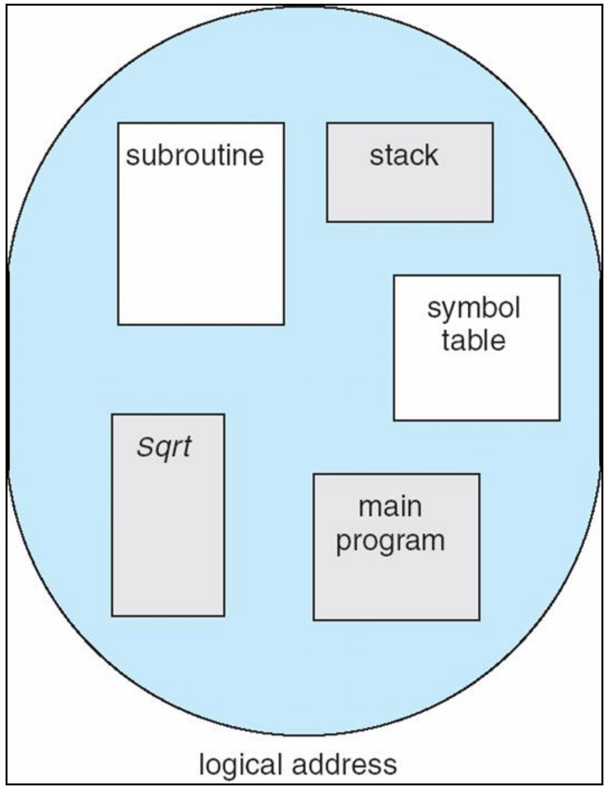
- Background
- **Contiguous** Memory Allocation
- **Non-contiguous** Memory Allocation
  - ➡ Segmentation
  - Paging
- Virtual Memory



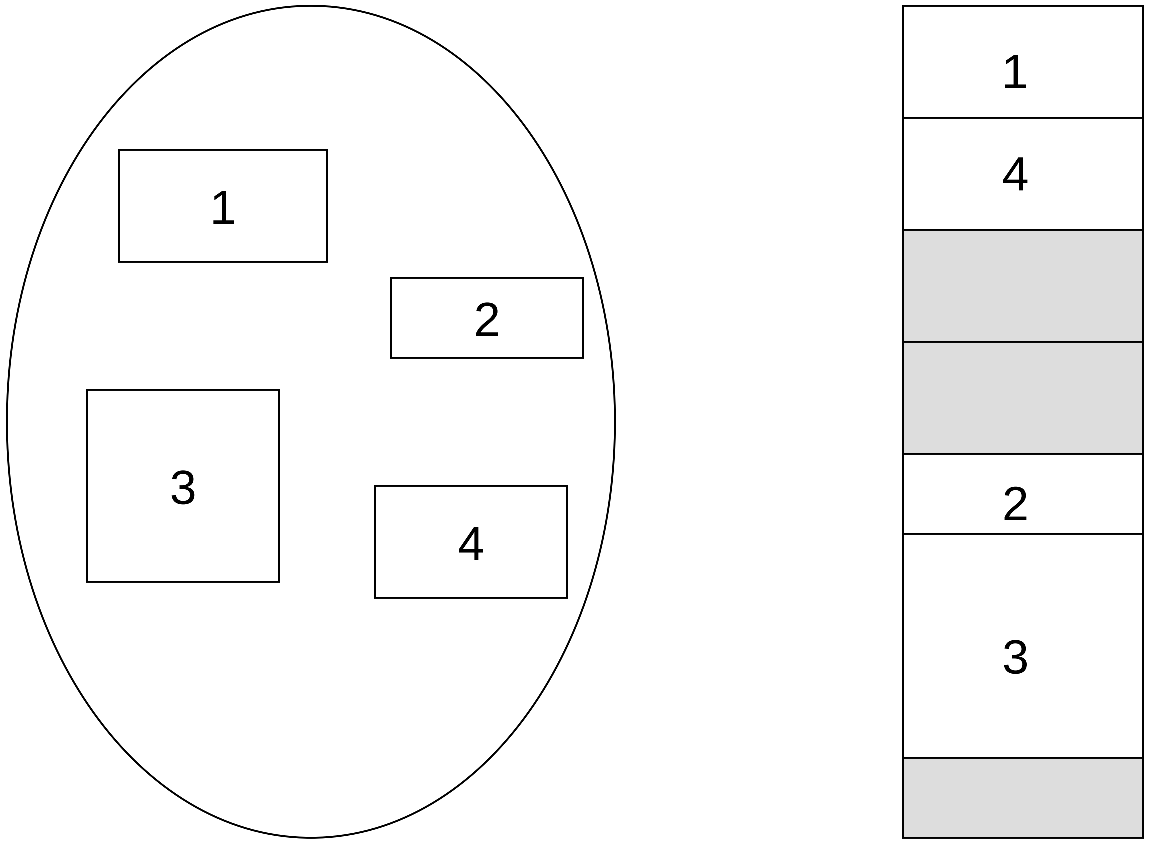
SEGMENTATION

- A program is a collection of segments. A segment is a logical unit such as:
  - main program
  - procedure
  - function
  - method
  - object
  - variables
  - common block
  - stack
  - arrays
  - …

USER'S VIEW OF A PROGRAM



LOGICAL VIEW OF SEGMENTATION



user space

physical memory space

SEGMENTATION ARCHITECTURE

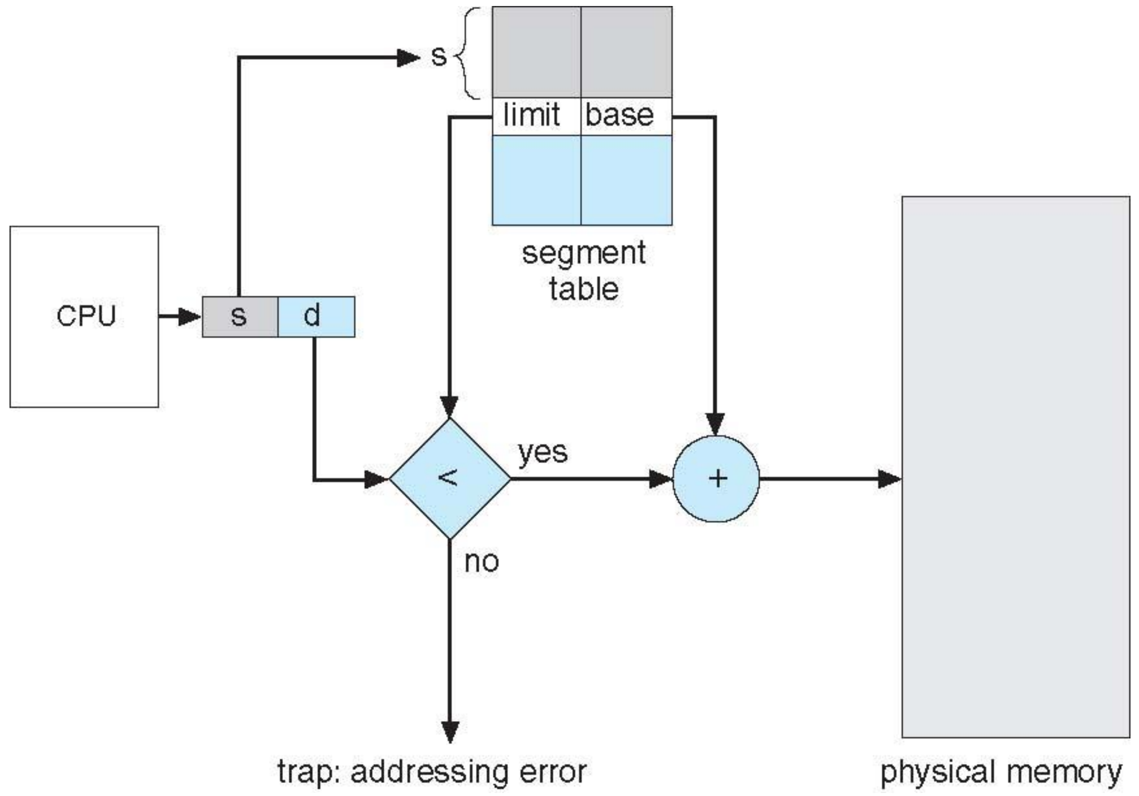
- Logical address consists of a two tuple:

<segment-number, offset>,

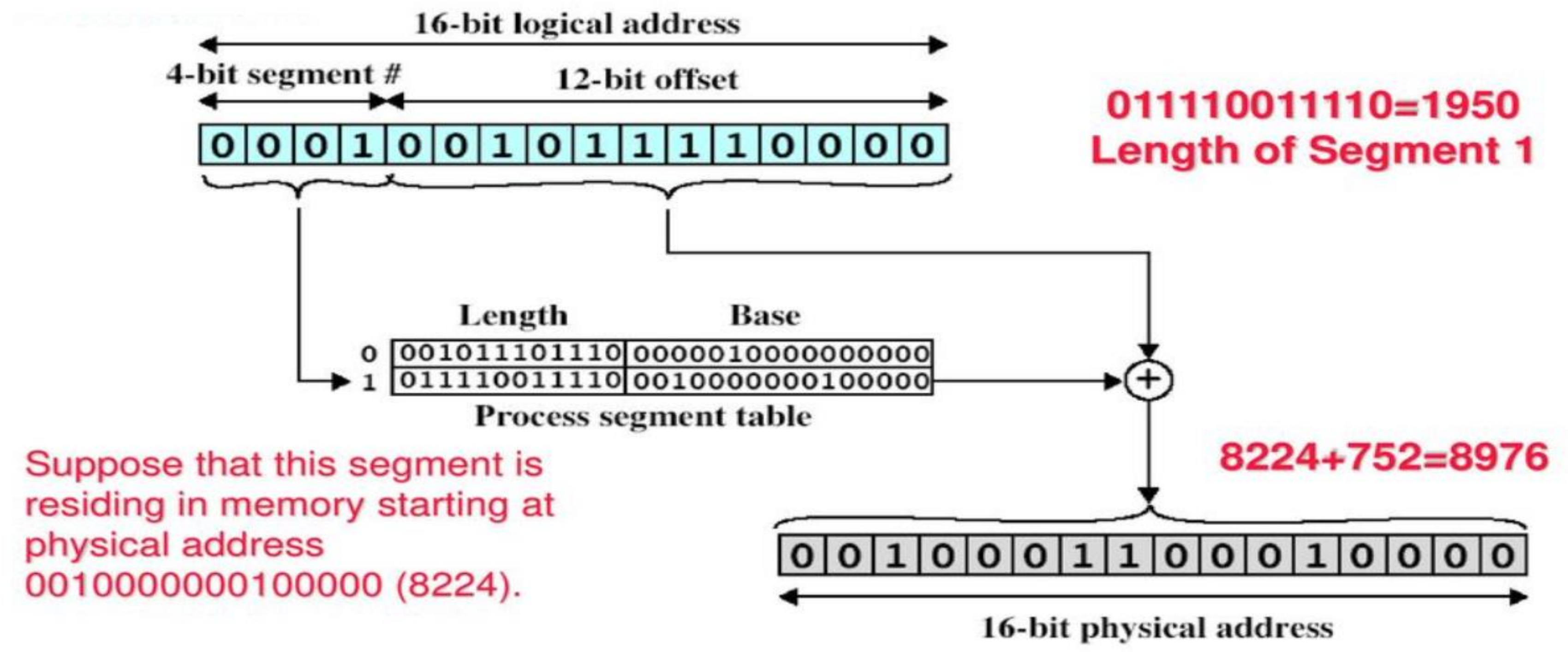
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location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

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

SEGMENTATION HARDWARE



LOGICAL-PHYSICAL ADDRESS TRANSLATION IN SEGMENTATION



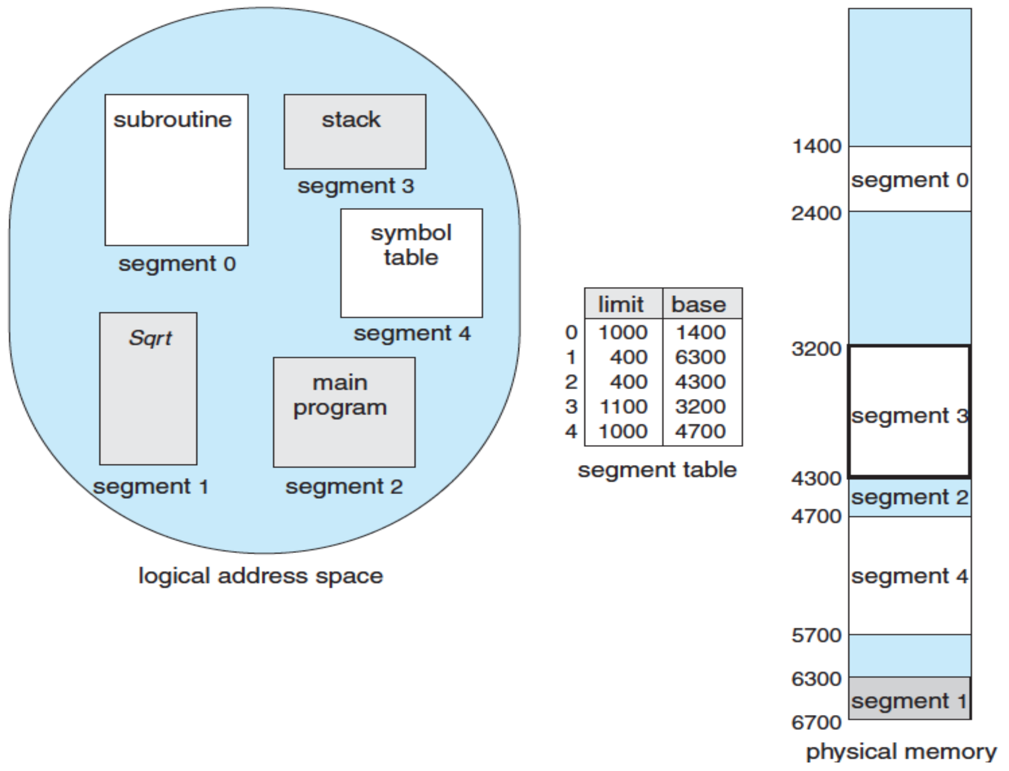
Logical address → 0001001011110000

Segment number 1 (0001) and offset 752 (001011110000).

Physical address → 0010000000100000+001011110000 =0010001100010000

(8224+752=8976)

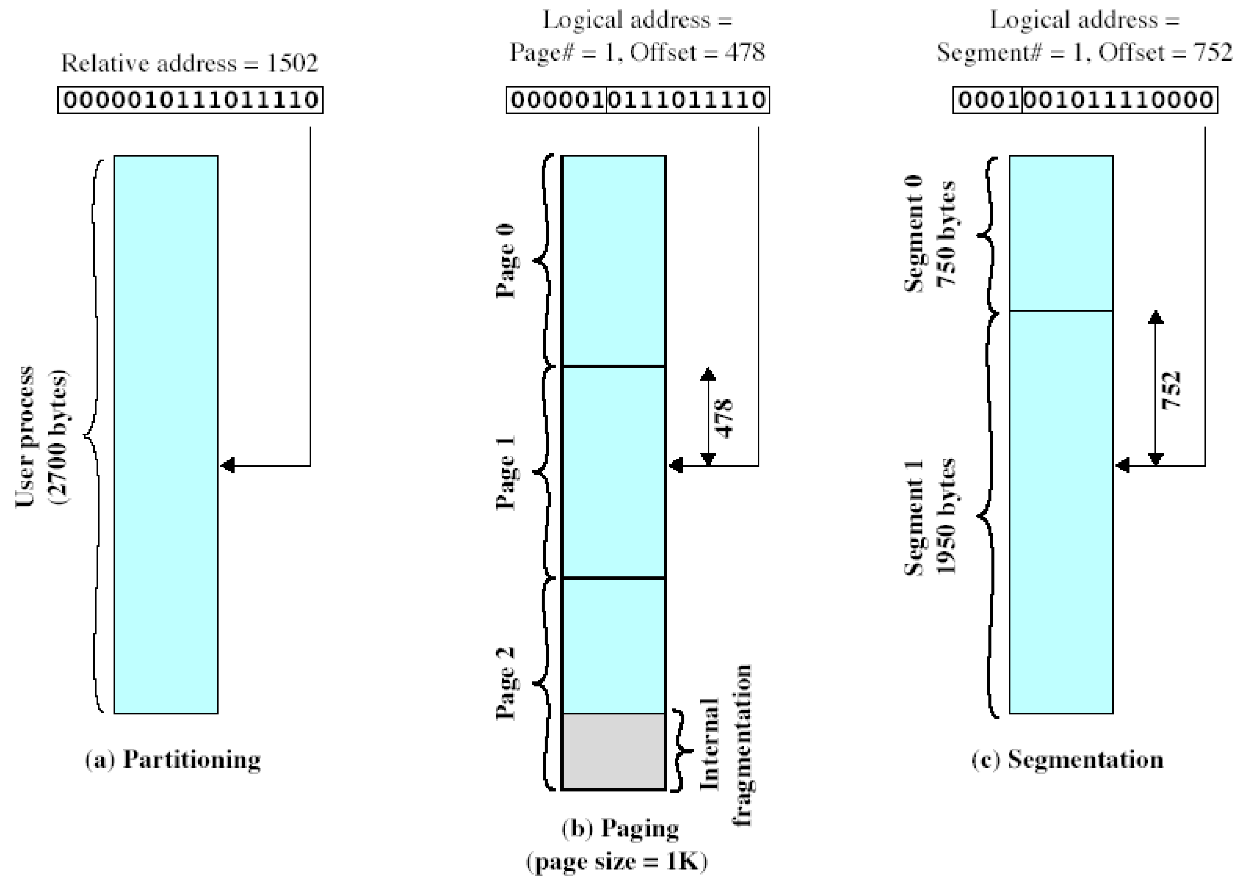
EXAMPLE OF SEGMENTATION.



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

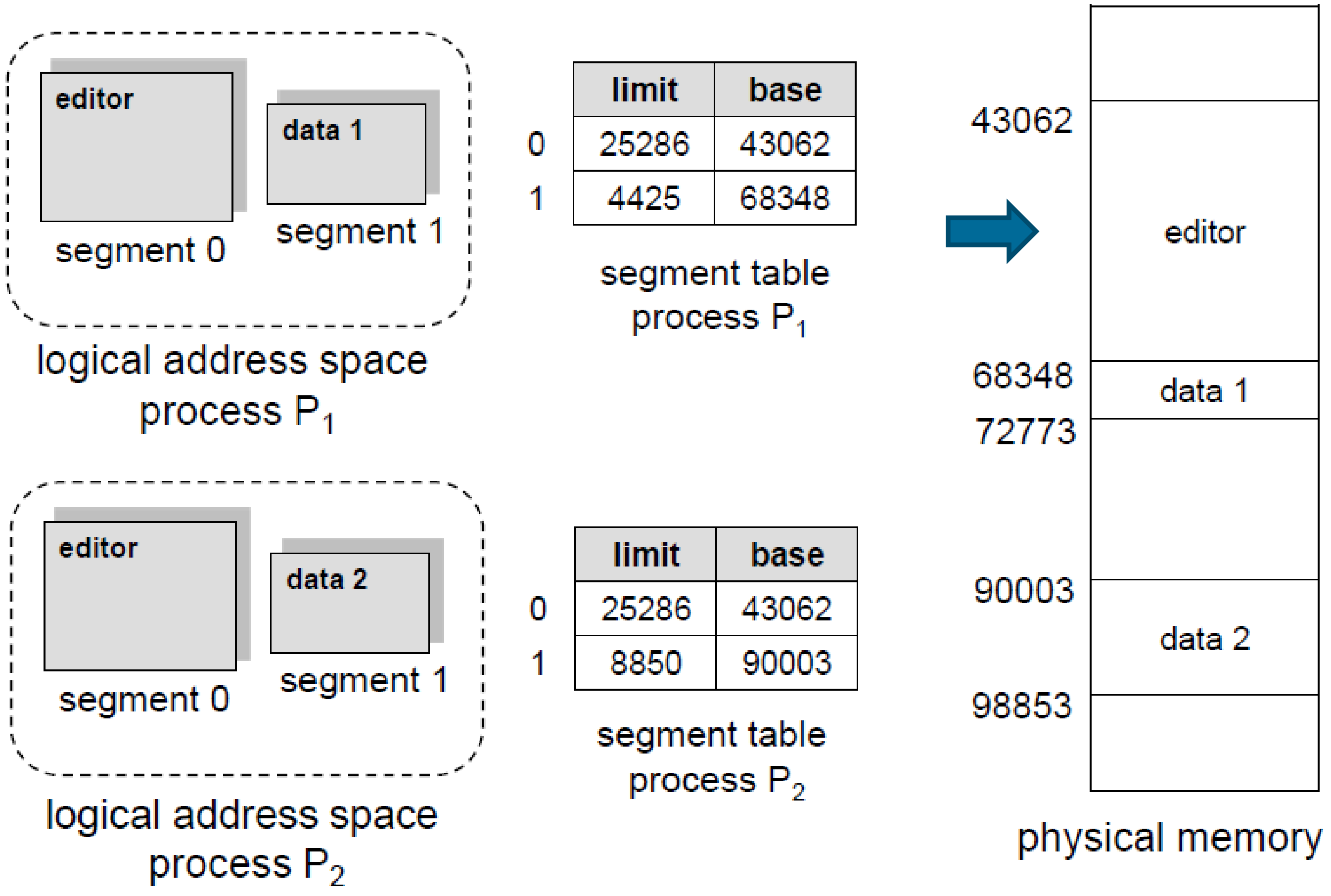
segment table

EXAMPLE OF SEGMENTATION (CONT.)



4 bits for Segment no
12 bits for Offset

SHARING OF SEGMENTS



editor
segment 0
data 1
segment 1
logical address space process P_1
editor
segment 0
data 2
segment 1
logical address space process P_2
segment table process P_1
segment table process P_2
43062
editor
68348
data 1
72773
90003
data 2
98853
physical memory

Segment table process P_1:

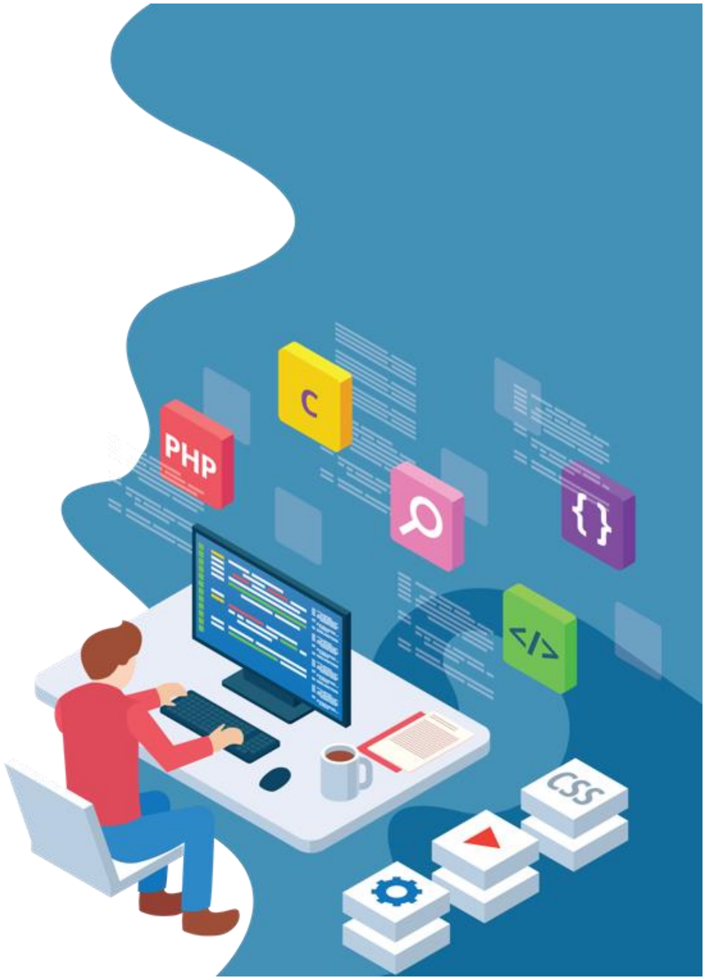
	limit	base
0	25286	43062
1	4425	68348

Segment table process P_2:

	limit	base
0	25286	43062
1	8850	90003

OUTLINE

- Background
- **Contiguous** Memory Allocation
- **Non-contiguous** Memory Allocation
  - Segmentation
  - ➡ Paging
- Virtual Memory

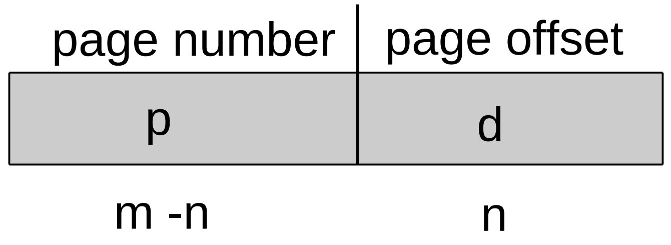


PAGING

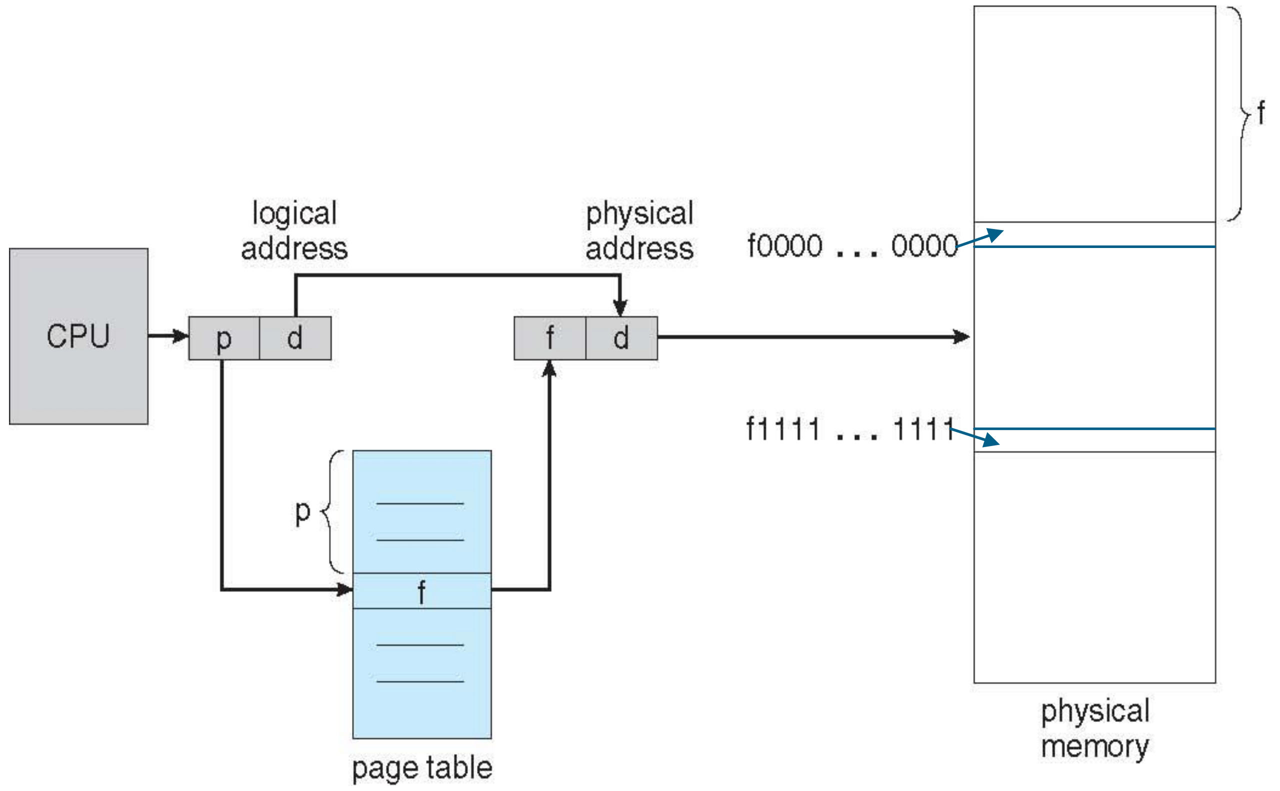
- Physical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
  - Avoids external fragmentation
  - Avoids problem of varying sized memory chunks
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 1Gb
- 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
- Backing store likewise split into pages
- Still have Internal fragmentation

ADDRESS TRANSLATION SCHEME

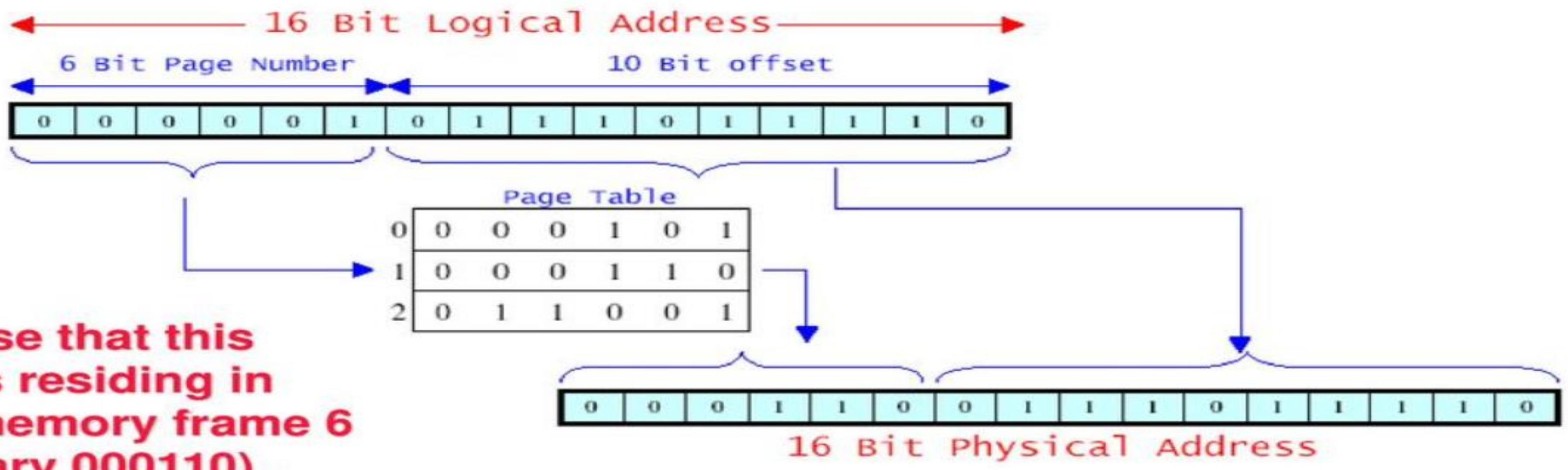
- Address generated by CPU is divided into:
  - **Page number(p)** – used as an index into a page table which contains base address of each page in physical memory
  - **Page offset(d)** – combined with base address to define the physical memory address that is sent to the memory unit



- For given logical address space 2^m and page size 2^n



LOGICAL-PHYSICAL ADDRESS TRANSLATION IN PAGING



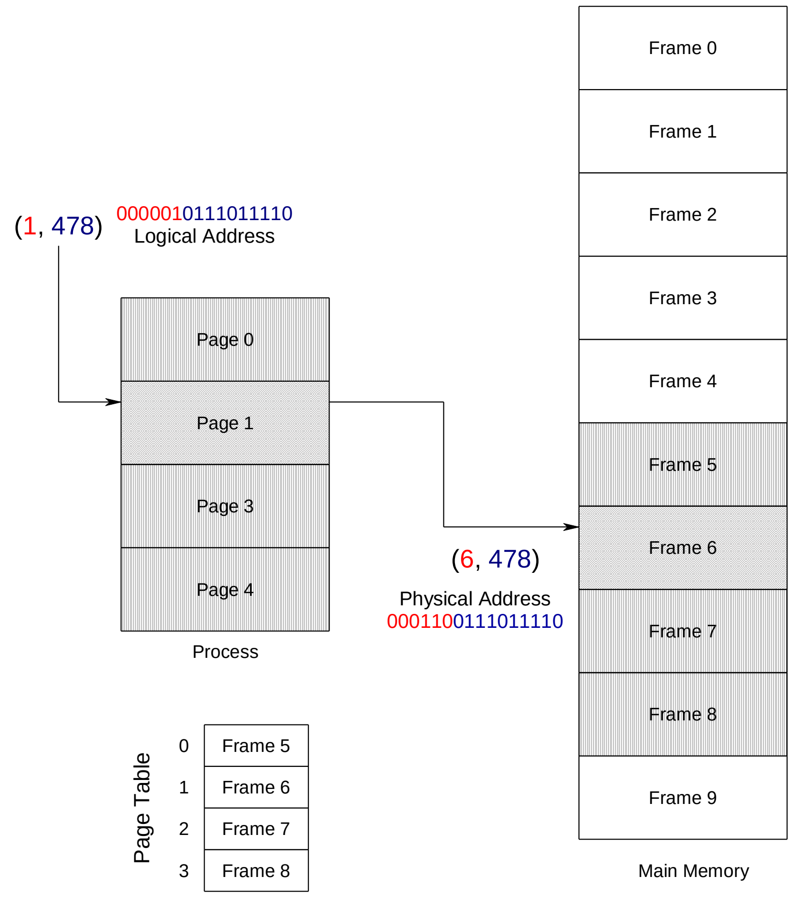
Suppose that this page is residing in main memory frame 6 (in binary 000110).

Logical address → 0000010111011110

Page number 1 (000001) and offset 478 (0111011110).

Physical address frame number 6 (000110) , offset 478 (0111011110)

LOGICAL-PHYSICAL ADDRESS TRANSLATION IN PAGING



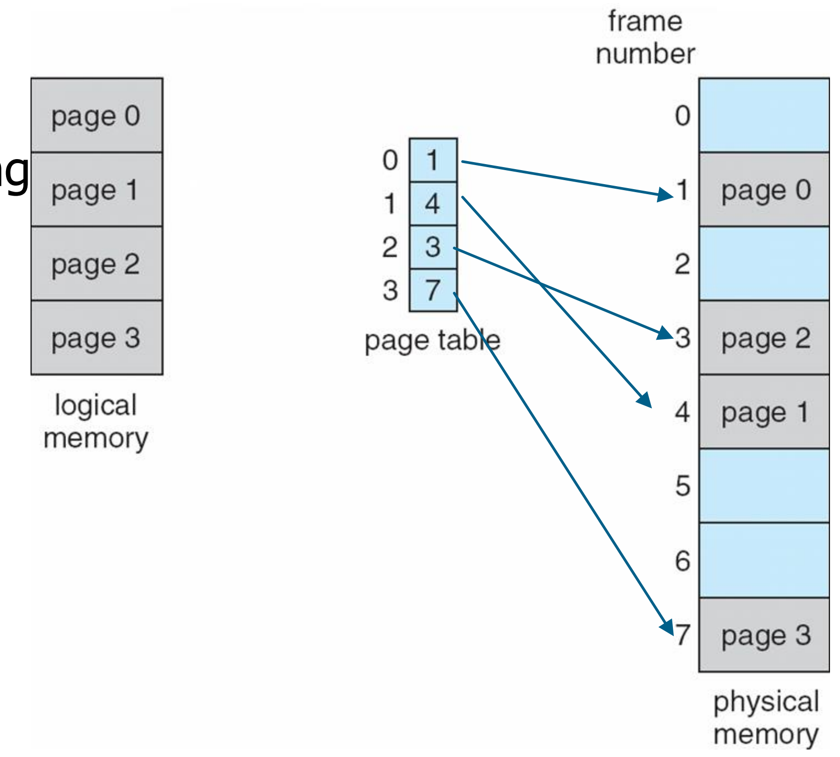
PAGING MODEL OF LOGICAL AND PHYSICAL MEMORY

- In the logical address, n=2 and m=4,

  => page size: $2^2 = 4$ bytes, logical address space $= 2^4 = 16$ bytes

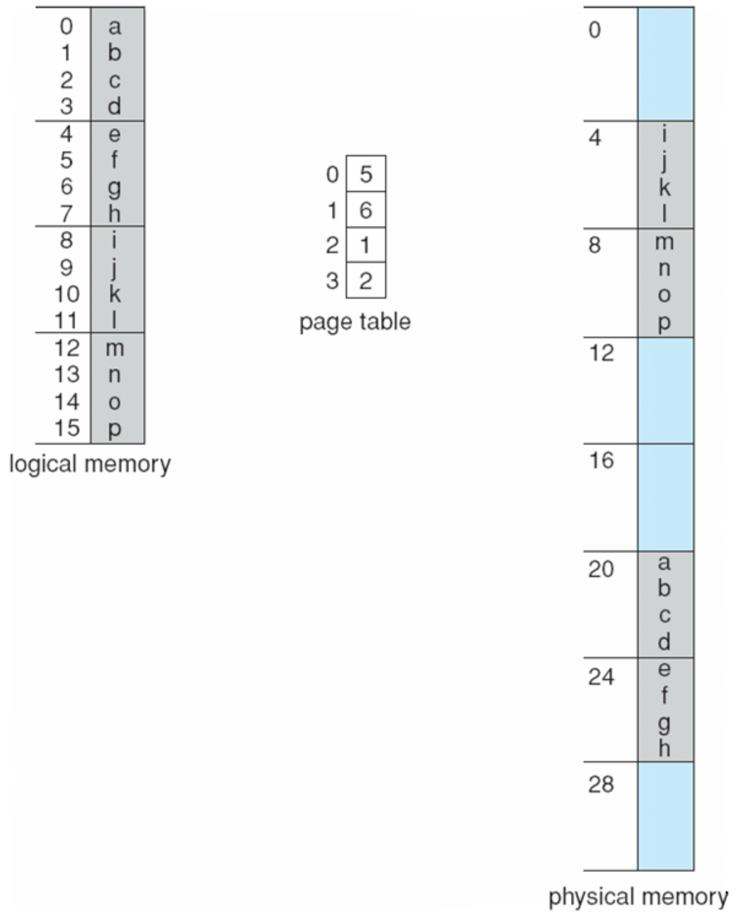
Bấm để thêm nội dung

- Physical memory of 32 bytes (8 page)



PAGING EXAMPLE

- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 [= (5 x 4) + 0].
- Logical address 3 is page 0 offset 3. Maps to physical address 23 [= (5 x 4) + 3].
- Logical address 4 is page 1, offset 0; according to the page table, page 1 is mapped to frame 6.



PAGING

- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - ➔ Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
  - So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
- Process view and physical memory now very different
- By implementation process can only access its own memory

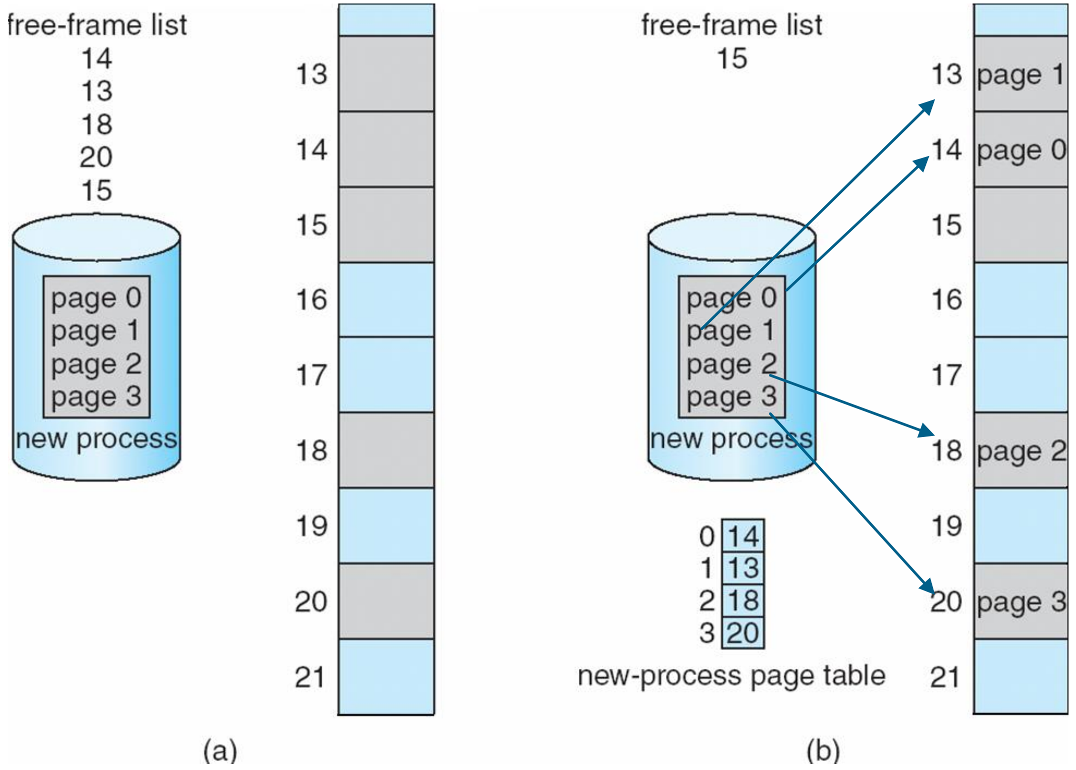
OBTAINING THE PAGE SIZE ON UNIX SYSTEMS

- The page size varies according to architecture, and there are several ways of obtaining the page size. One approach is to use the **getpagesize()** system call.
- Another strategy is to enter the following command on the command line:

```
getconf PAGESIZE
```

- Each of these techniques returns the page size as a number of bytes.

FREE FRAMES



Before allocation After allocation

IMPLEMENTATION OF PAGE TABLE

- Page tables are too large to be kept on the chip (Registers). Page table is kept in main memory.
- **Page-table base register** (**PTBR**) :points to the beginning of the page table for this process
- **Page-table length register** (**PTLR**) : size of of 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**)

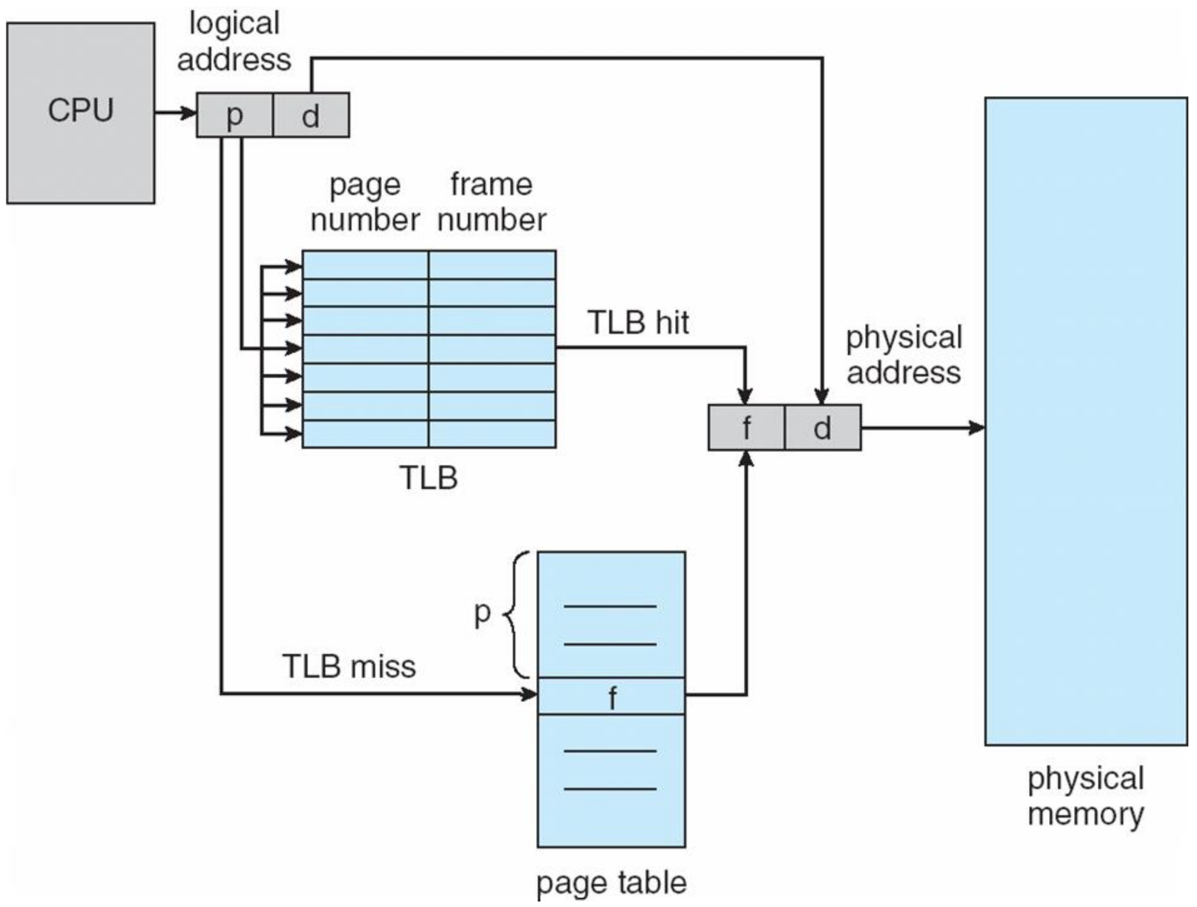
IMPLEMENTATION OF PAGE TABLE (CONT.)

- TLBs typically small (64 to 1,024 entries)
- On a TLB miss, value is loaded into the TLB for faster access next time
  - Replacement policies must be considered. Ex: Least recently used (LRU),
  - Some entries can be **wired down** for permanent fast access

Page #	Frame #

PAGING HARDWARE WITH TLB

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



EFFECTIVE ACCESS TIME

- Hit ratio – percentage of times that a page number is found in the TLB.
- An 80% hit ratio means that we find the desired page number in the TLB 80% of the time.
- Suppose that 10 nanoseconds to access memory.
  - If we find the desired page in TLB then a mapped-memory access take 10 ns
  - Otherwise we need two memory access so it is 20 ns
- **Effective Access Time** (**EAT**) : P x hit memory time + (1-P) x miss memory time.

  EAT = 0.80 x 10 + 0.20 x 20 = 12 nanoseconds

  implying 20% slowdown in access time
- Consider amore realistic hit ratio of 99%,

  EAT = 0.99 x 10 + 0.01 x 20 = 10.1ns

  implying only 1% slowdown in access time.

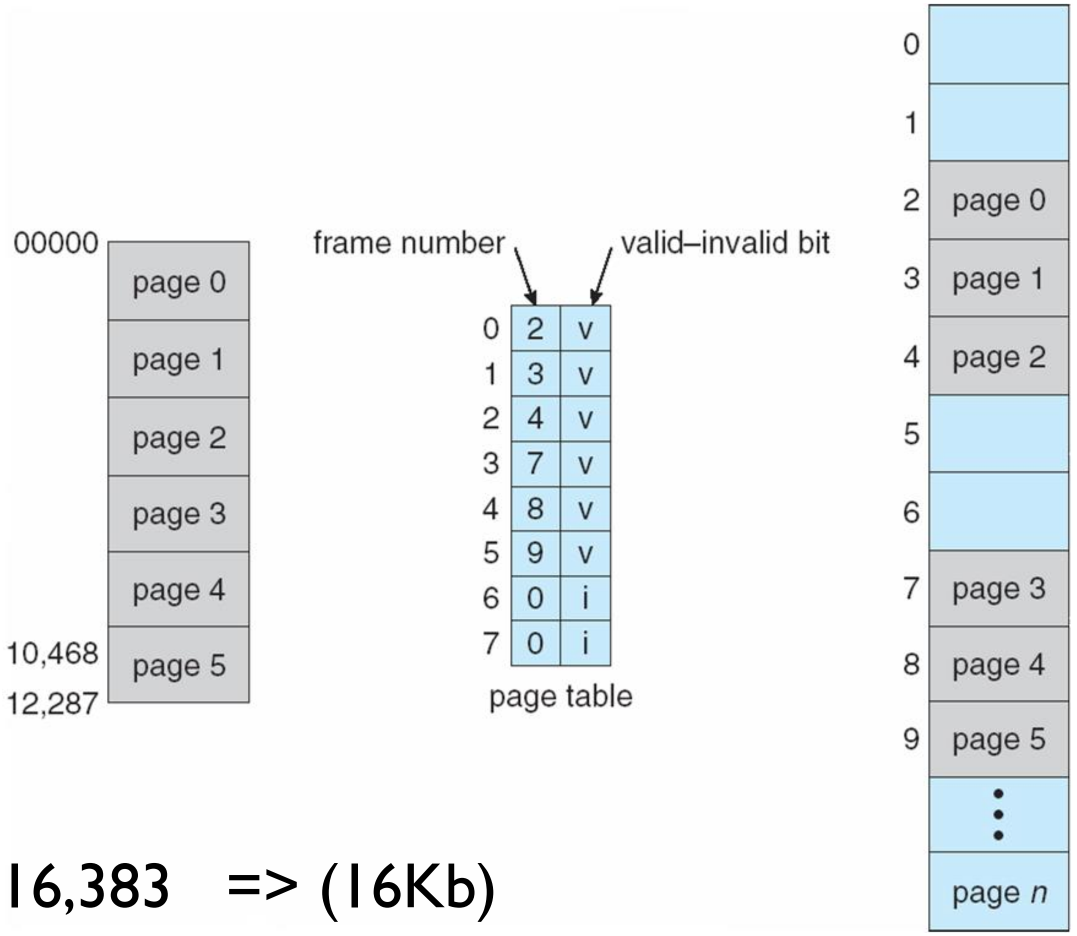
MULTIPLE LEVELS OF TLBS

- CPUs today may provide multiple levels of TLBs. Calculating memory access times in modern CPUs is therefore much more complicated than shown in the example above.
- For instance, the Intel Core i7 CPU:
  - Has a (128-entry L1 instruction TLB) and a 64-entry L1 data TLB. In the case of a miss at L1, it takes the CPU **six cycles** to check for the entry in the L2 (512-entry TLB).
  - A miss in L2 means that the CPU must either walk through the page-table entries in memory to find the associated frame address, which can take **hundreds of cycles**

MEMORY PROTECTION

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
  - Can also add more bits to indicate page execute-only, and so on
- **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
  - Or use **page-table length register** (**PTLR**)
- Any violations result in a trap to the kernel

VALID (V) OR INVALID (I) BIT IN A PAGE TABLE

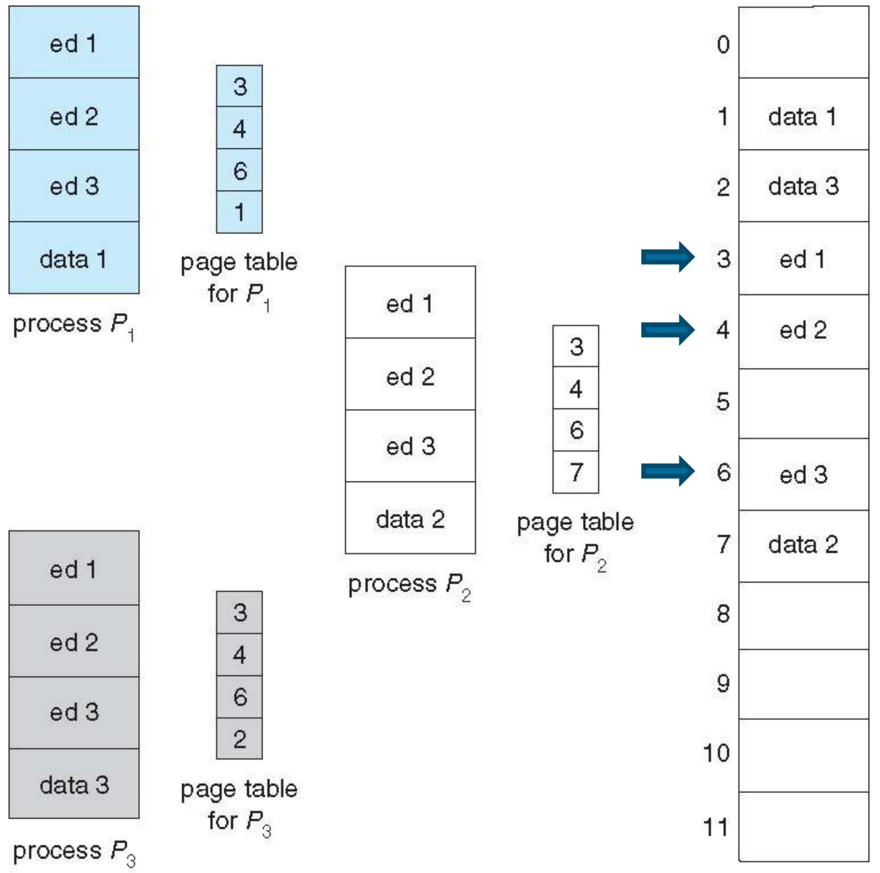


16,383 => (16Kb)

SHARED PAGES

- **Shared code**
  - One copy of read-only (**reentrant**) code shared among processes (i.e., text editors, compilers, window systems)
  - Similar to multiple threads sharing the same process space
  - Also useful for interprocess communication if sharing of read-write pages is allowed
- **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



STRUCTURE OF THE PAGE TABLE

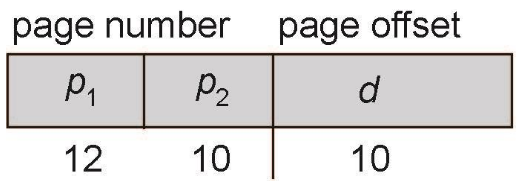
- Memory structures for paging can get huge using straight-forward methods
  - Consider a 32-bit logical address space as on modern computers with 4K page size:
    - Page size of 4 KB (2^{12}) $\Rightarrow$ a **page offset** consisting of 12 bits.
    - Page table would have 1 million entries (2^{32} / 2^{12})= 2^{20} $\Rightarrow$ a **page number** consisting of 20 bits.
  - If each entry is 4 bytes, each process may need up tó 4 MB of physical address space / memory for page table alone
- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

HIERARCHICAL PAGE TABLES

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

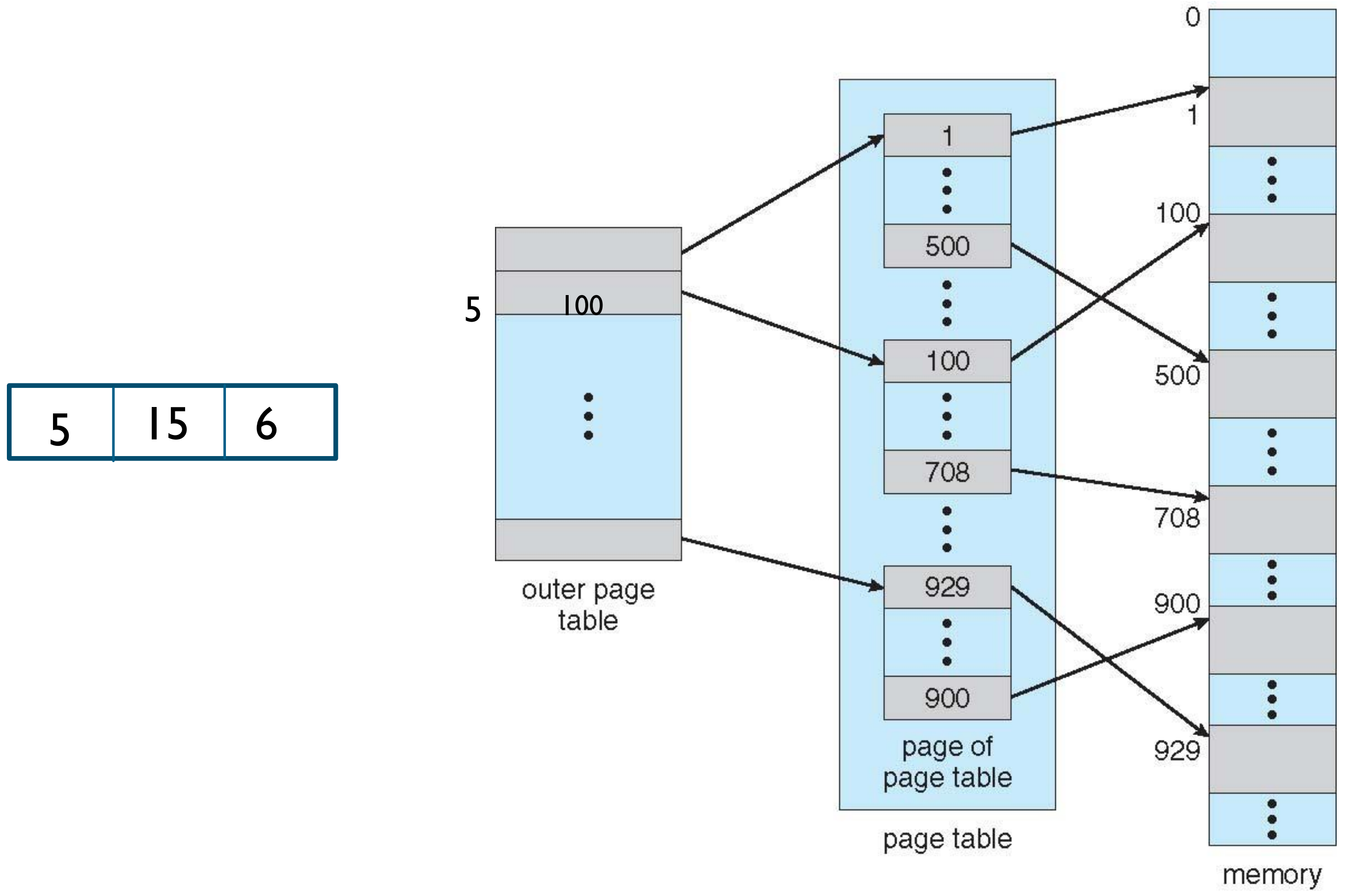
TWO-LEVEL PAGING EXAMPLE

- A logical address (on 32-bit machine with 1K page size) is divided into:
  - a page number consisting of 22 bits
  - a page offset consisting of 10 bits
- Since the page table is paged, the page number is further divided into:
  - a 12-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

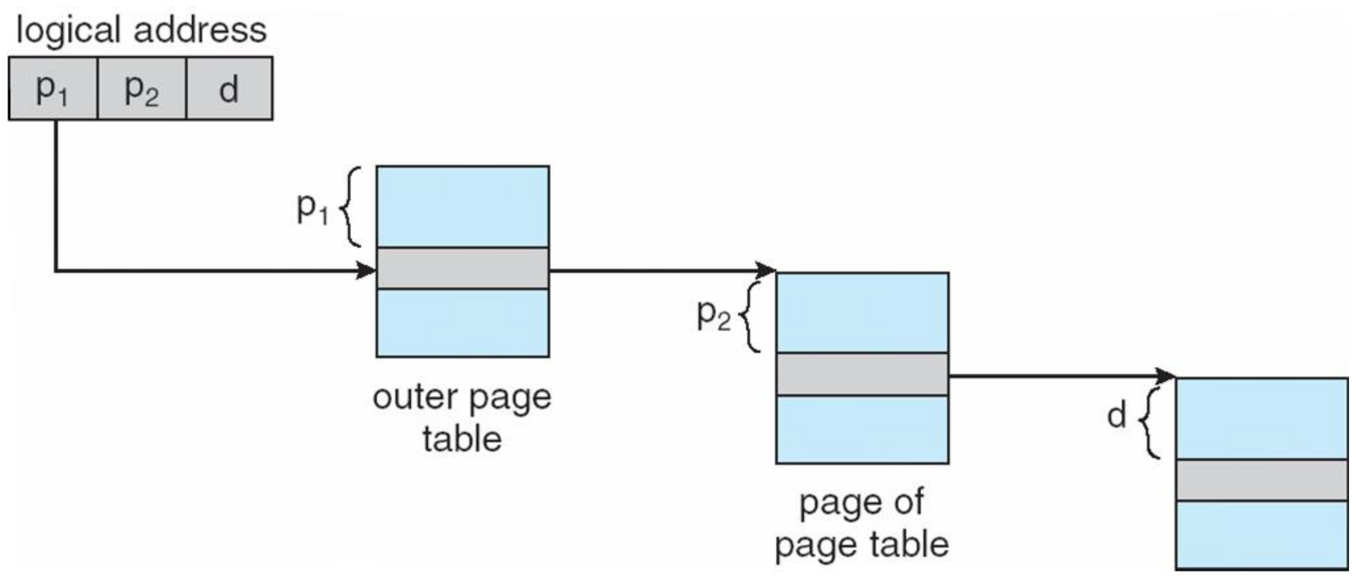


- Where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

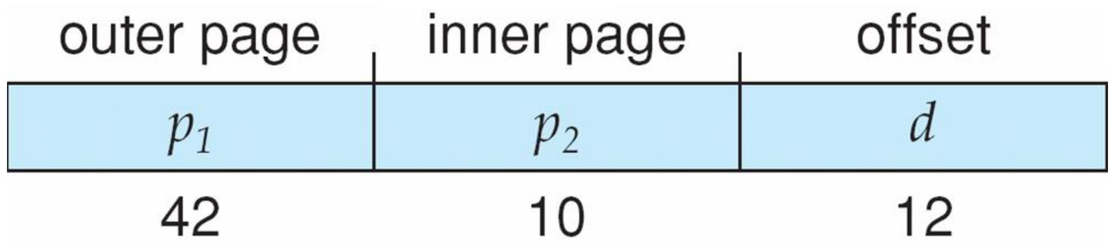
TWO-LEVEL PAGE-TABLE SCHEME

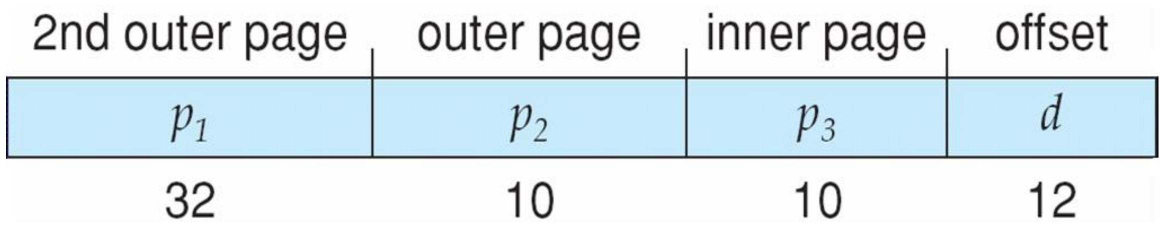


ADDRESS-TRANSLATION SCHEME



THREE-LEVEL PAGING SCHEME





EXAMPLE: THE INTEL 32 AND 64-BIT ARCHITECTURES

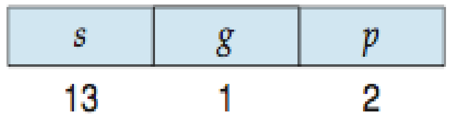
- Dominant industry chips
- Pentium CPUs are 32-bit and called IA-32 architecture
- Current Intel CPUs are 64-bit and called IA-64 architecture
- Many variations in the chips, cover the main ideas here

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**))

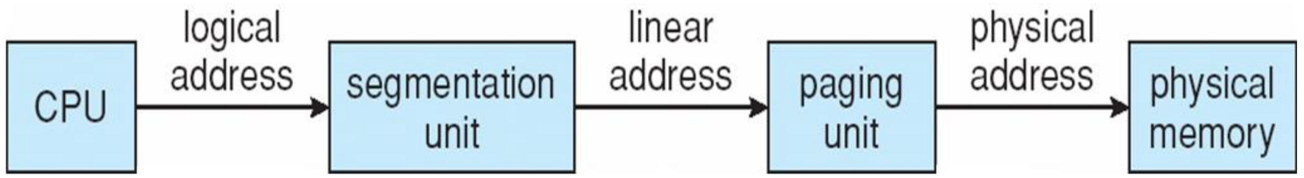
EXAMPLE: THE INTEL IA-32 ARCHITECTURE (CONT.)

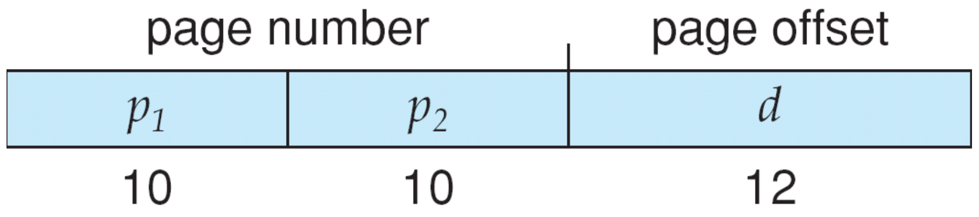
- CPU generates logical address
  - Selector given to segmentation unit
    - Which produces linear addresses



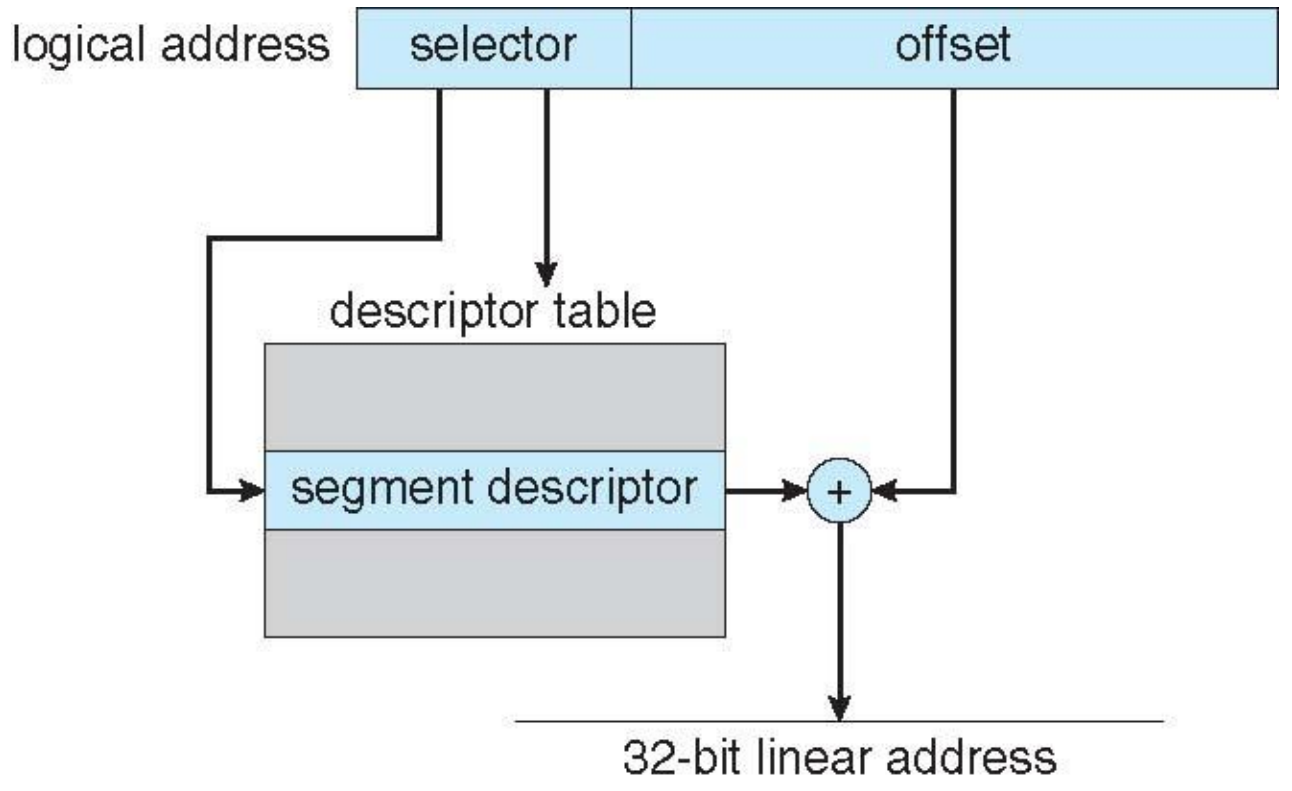
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
    - Pages sizes can be 4 KB or 4 MB

LOGICAL TO PHYSICAL ADDRESS TRANSLATION IN IA-32

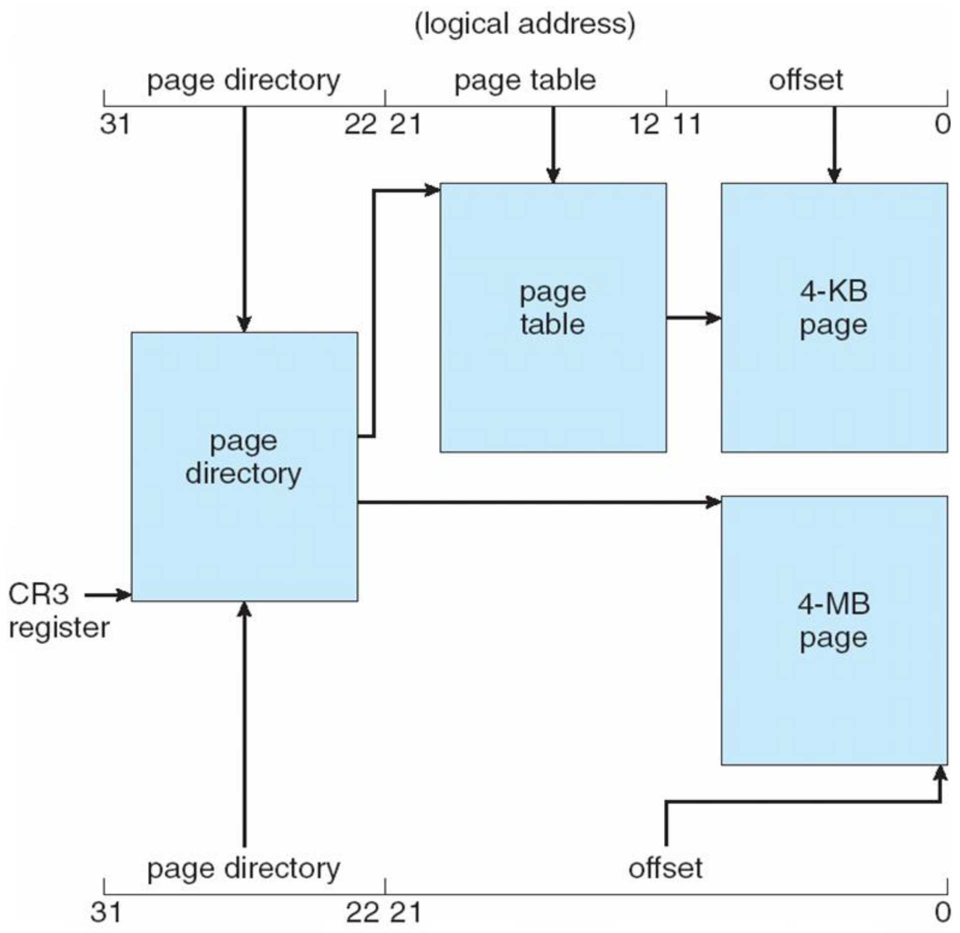




INTEL IA-32 SEGMENTATION

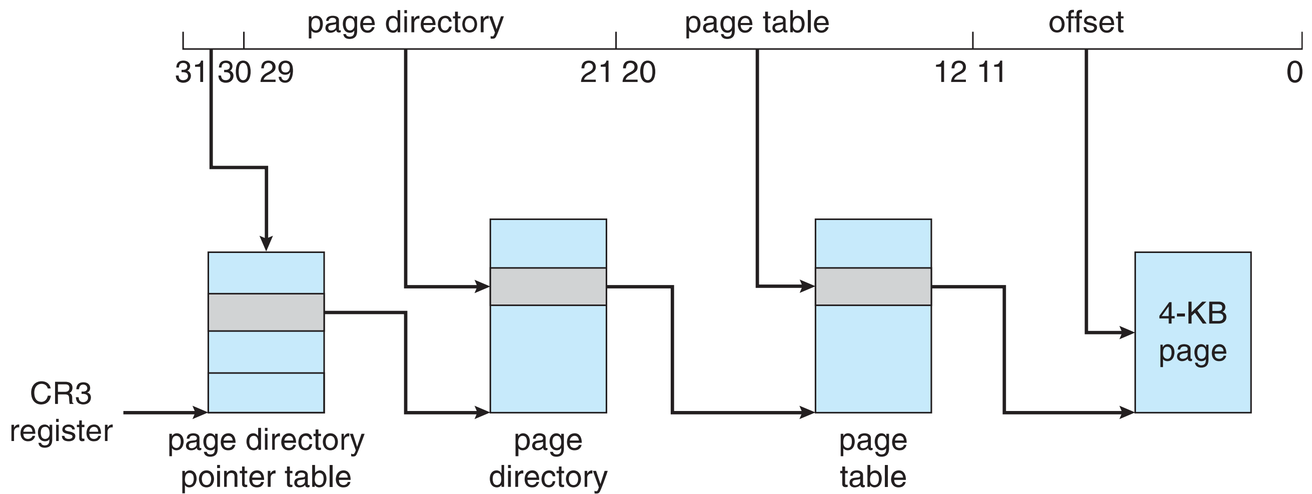


INTEL IA-32 PAGING ARCHITECTURE



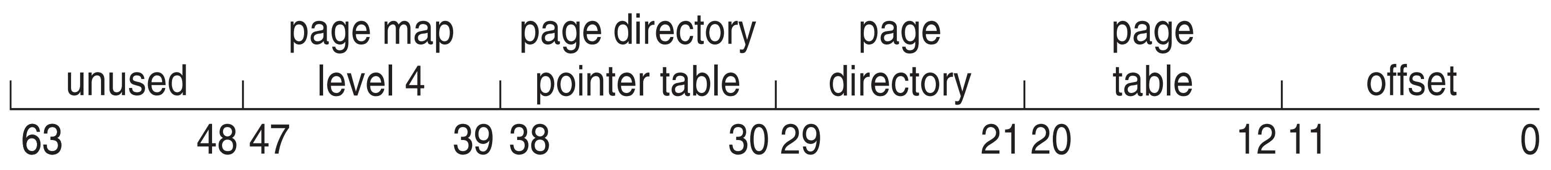
INTEL IA-32 PAGE ADDRESS EXTENSIONS

- 32-bit address limits led Intel to create **page address extension** (**PAE**), allowing 32-bit apps access to more than 4GB of memory space
  - Paging went to a 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
  - Net effect is increasing address space to 36 bits – 64GB of physical memory



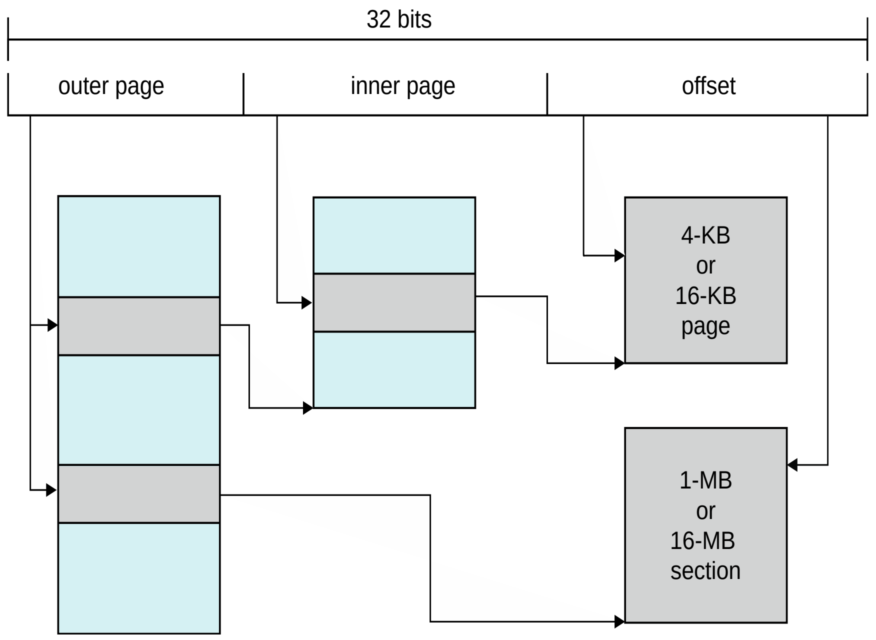
INTEL X86-64

- 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
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits



EXAMPLE: ARM ARCHITECTURE

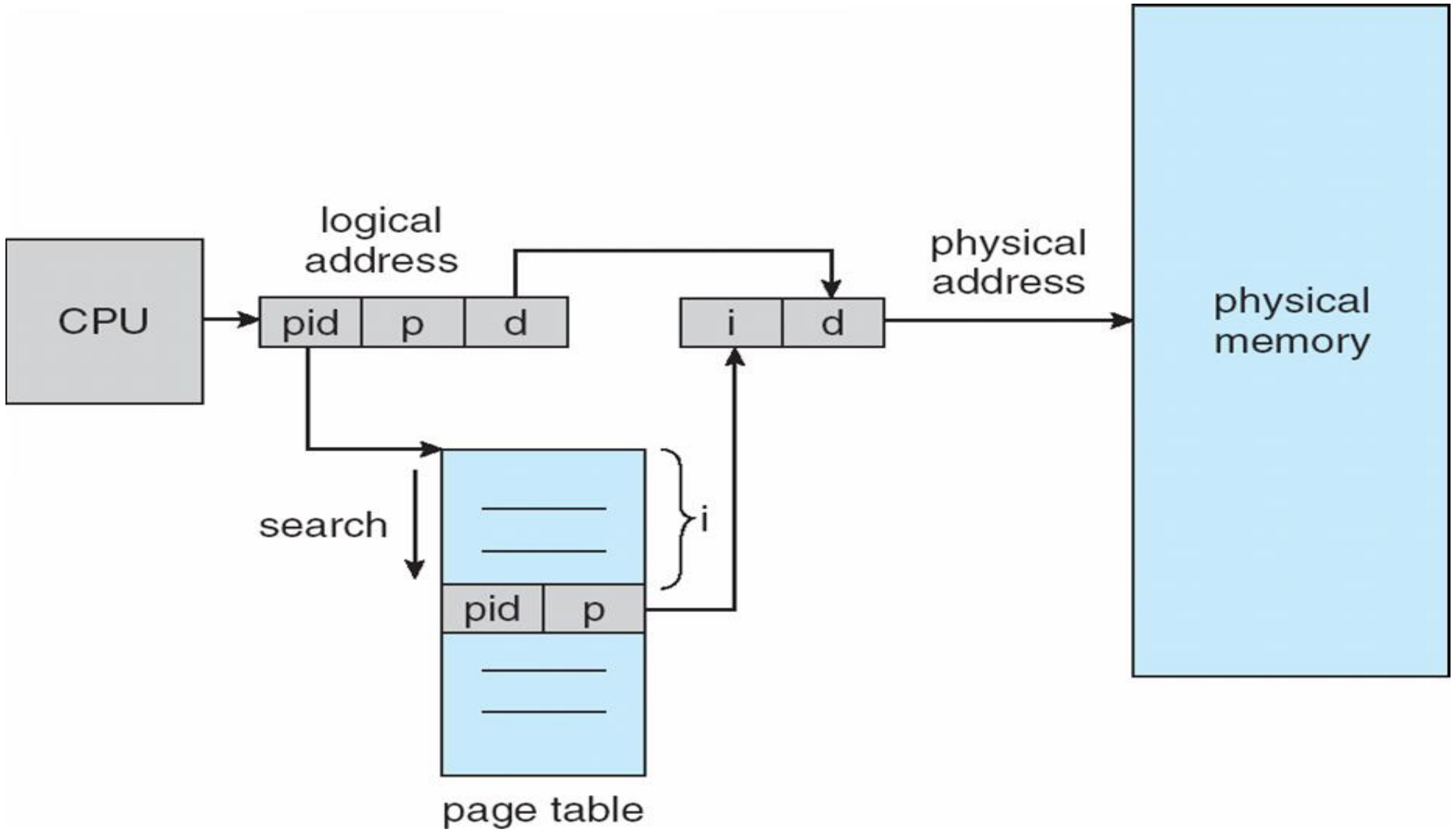
- Dominant mobile platform chip (Apple iOS and Google Android devices for example)
- Modern, energy efficient, 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed **sections**)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU



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

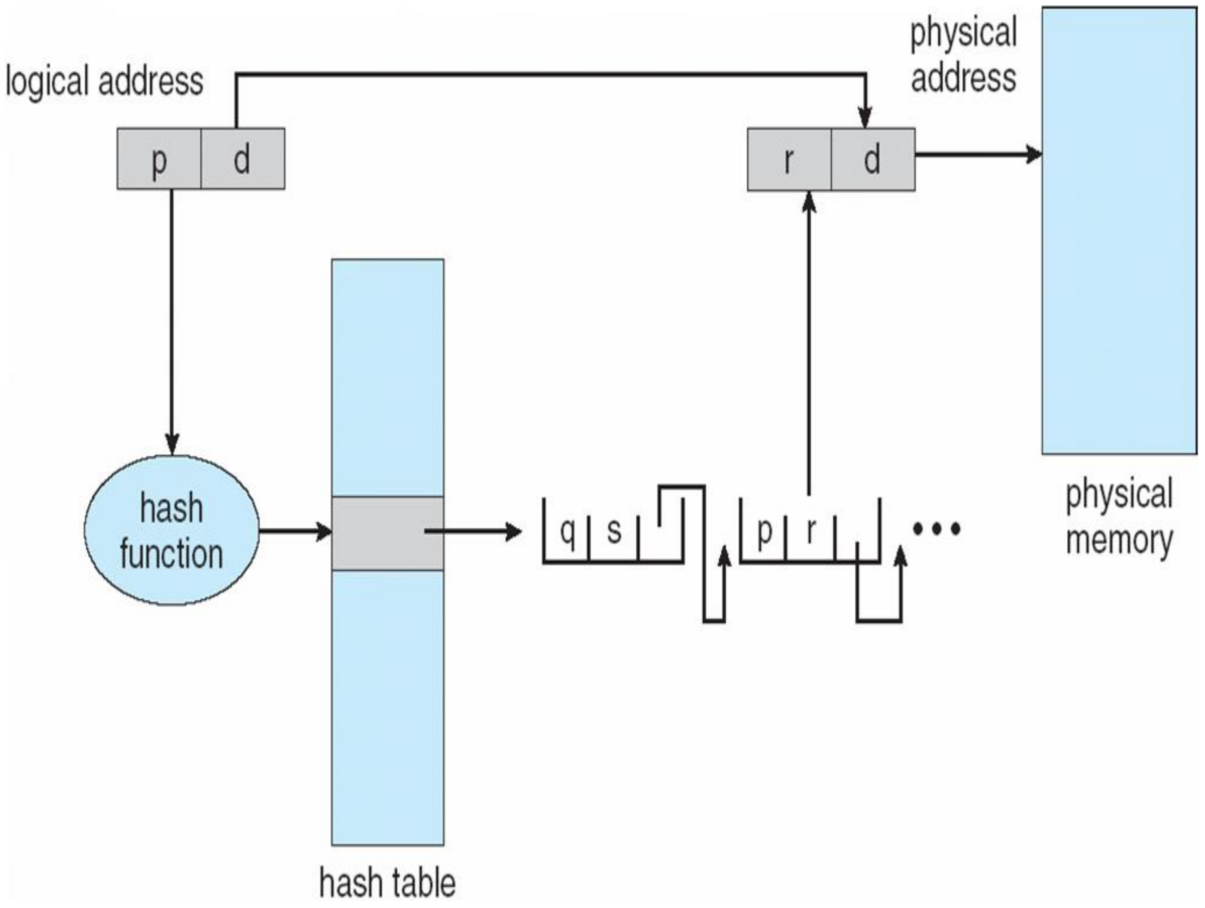
INVERTED PAGE TABLE ARCHITECTURE

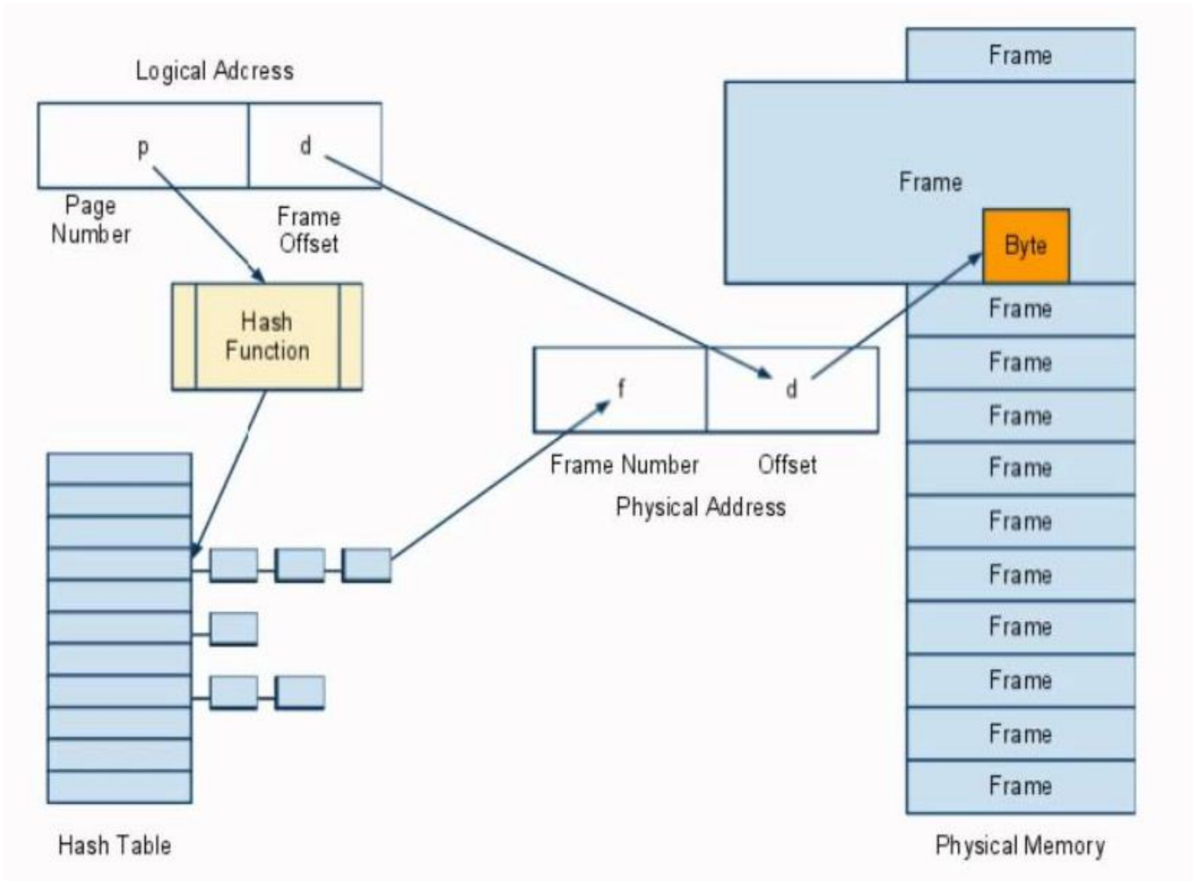


HASHED PAGE TABLE

- 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

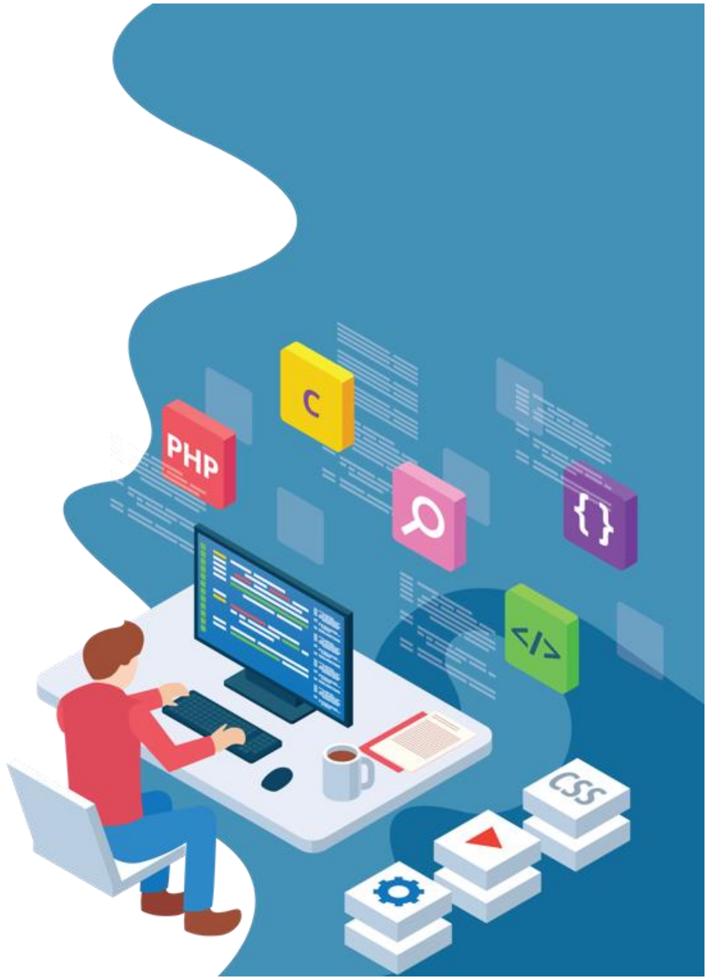
HASHED PAGE TABLE





OUTLINE

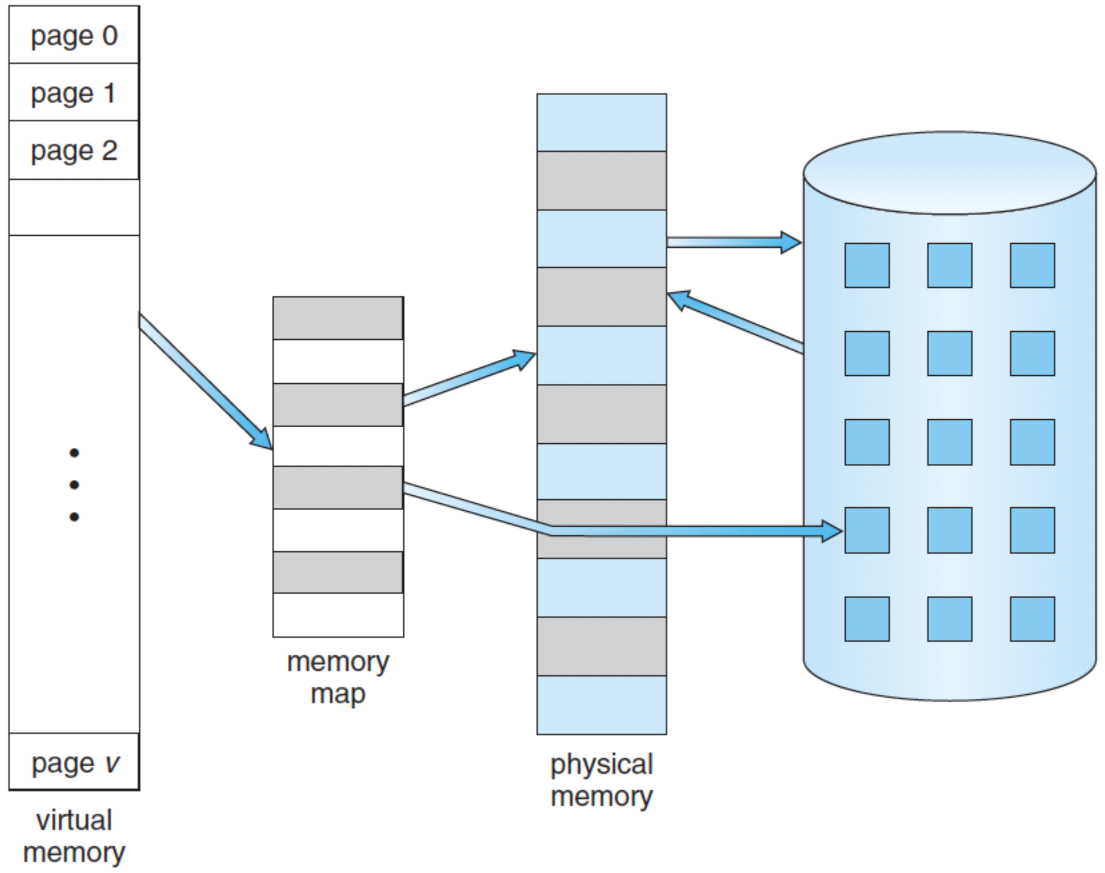
- Background
- **Contiguous** Memory Allocation
- **Non-contiguous** Memory Allocation
  - Segmentation
  - Paging
- ➡ Virtual Memory



VIRTUAL MEMORY

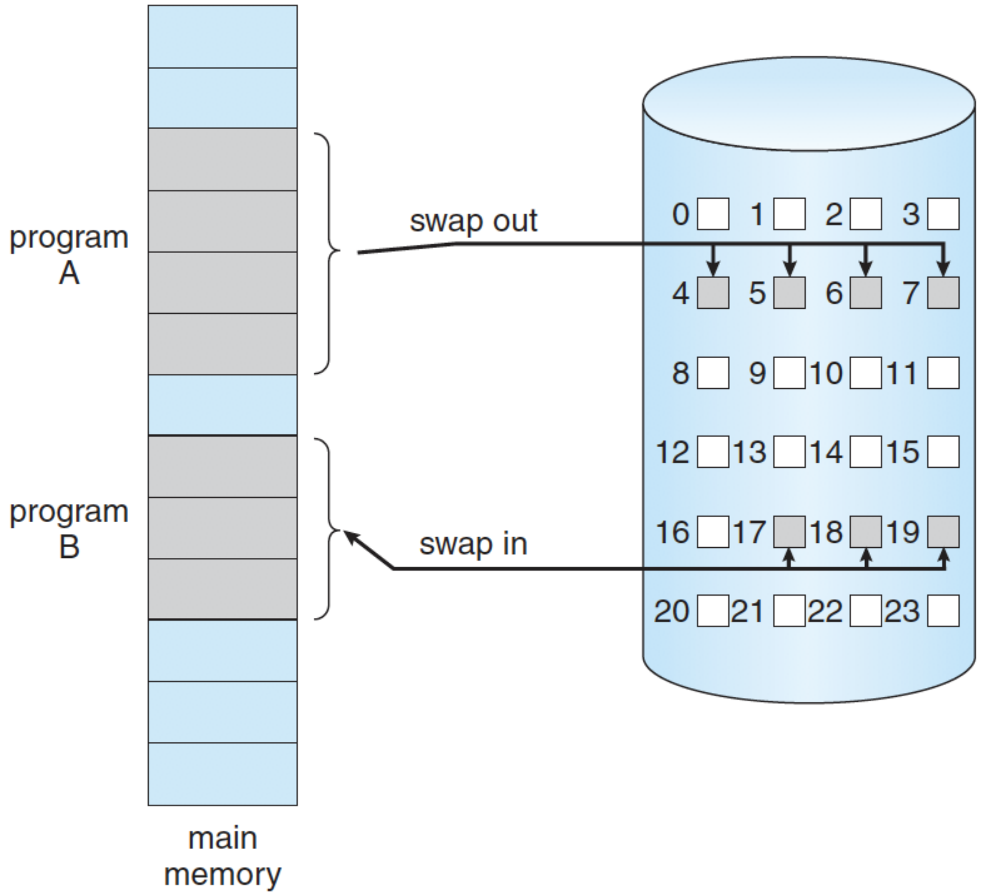
- Virtual memory is a technique that allows the execution of processes that are not completely in memory
- One major advantage of this scheme is that programs can be larger than physical memory.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory

BACKGROUND


page 0
page 1
page 2
page *v*
virtual memory
memory map
physical memory

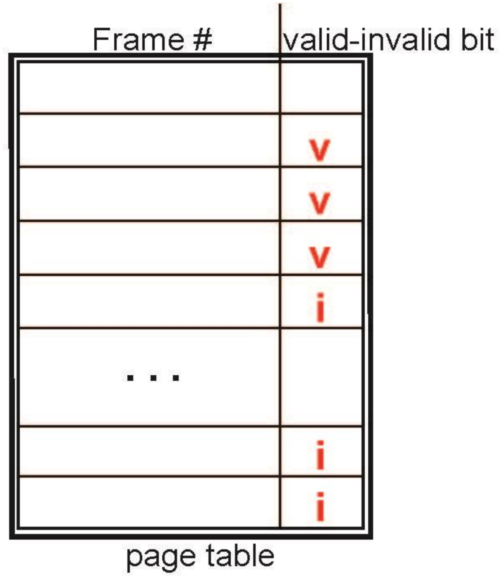
DEMAND PAGING

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping (diagram on right)



VALID-INVALID BIT

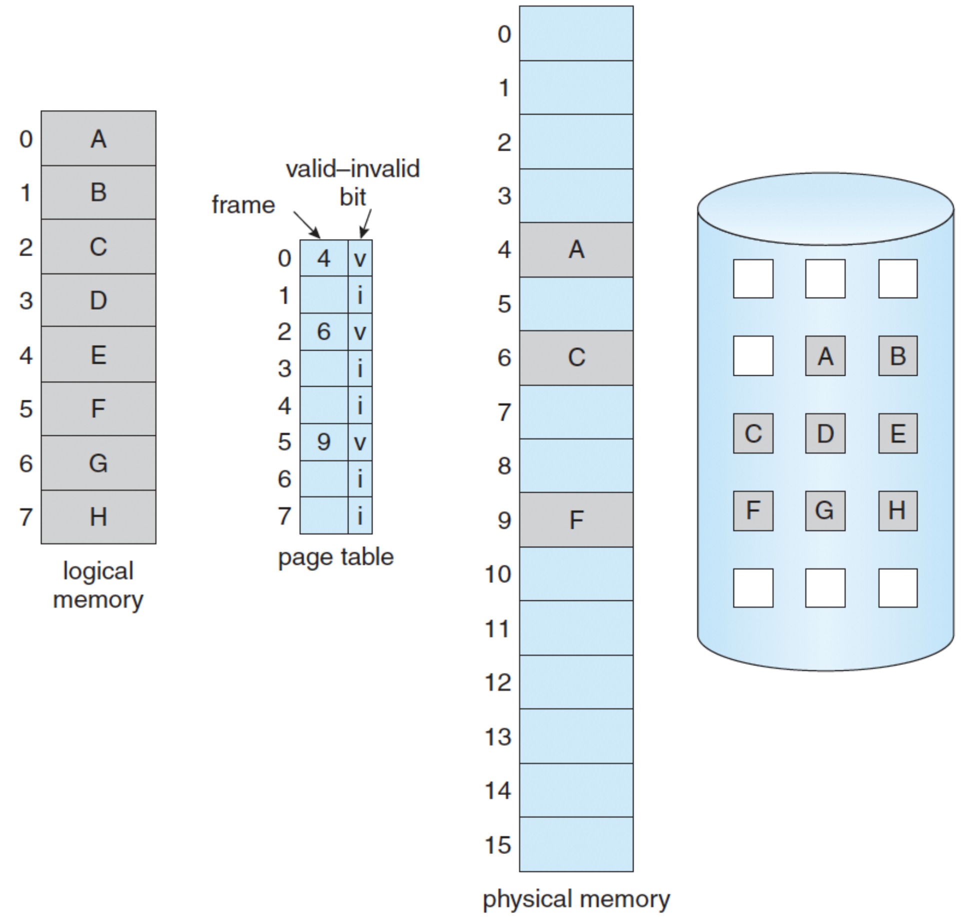
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

BASIC CONCEPTS

- When a process is to be swapped in, the pager **guesses** which pages will be used before the process is swapped out again.
- Instead of swapping in a whole process, the pager brings only those pages into memory.
- It avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

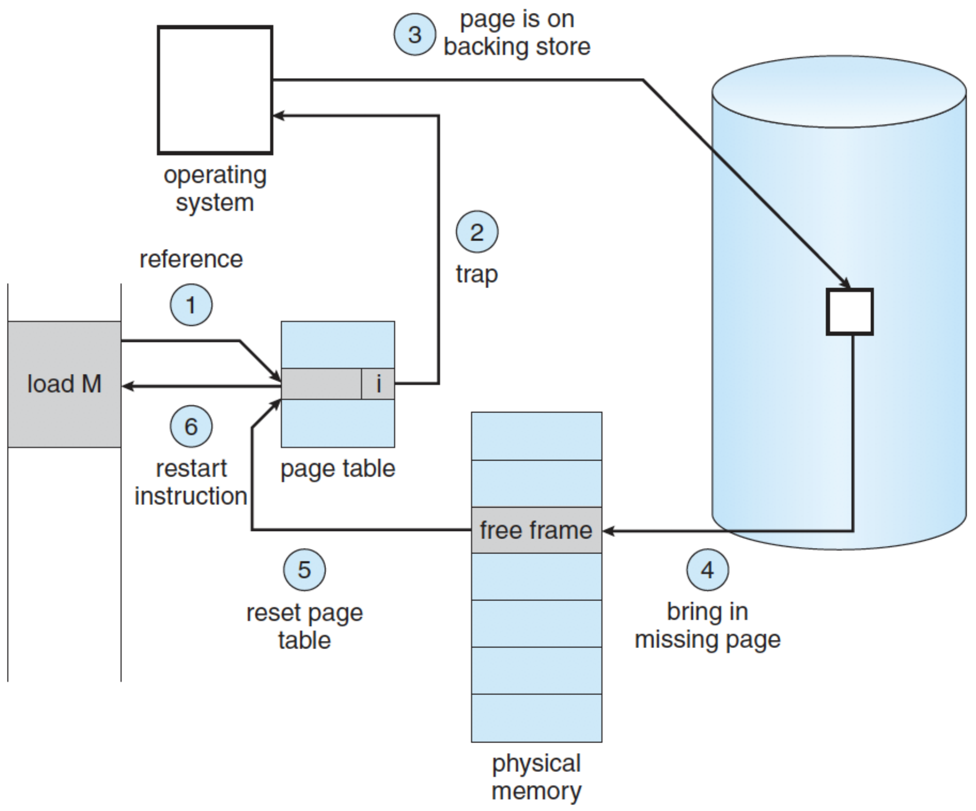


STEPS IN HANDLING PAGE FAULT

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

STEPS IN HANDLING A **PAGE FAULT**

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault



PERFORMANCE OF DEMAND PAGING

- **The effective access time**:

$$\text{effective access time (EAT)} = (1 - p) * ma + p * \text{page fault time.}$$

- *ma* ranges from 10 to 200 nanoseconds.
- p: page fault ratio $0 \le p \le 1$
  - p=0 no page fault (never happened)
  - p=1 all reference is fault
- *page fault time* takes ~ 8 milliseconds (A typical hard disk has an average latency of 3 milliseconds, a seek of 5 milliseconds, and a transfer time of 0.05 milliseconds)

$$\text{effective access time} = (1 - p) * (200) + p * (8 \text{ milliseconds})$$
$$= (1 - p) * 200 + p * 8{,}000{,}000$$
$$= 200 + 7{,}999{,}800 * p \text{ (ns)}$$

PAGE REPLACEMENT

1. Find the location of the desired page on the disk

2. Find a free frame:

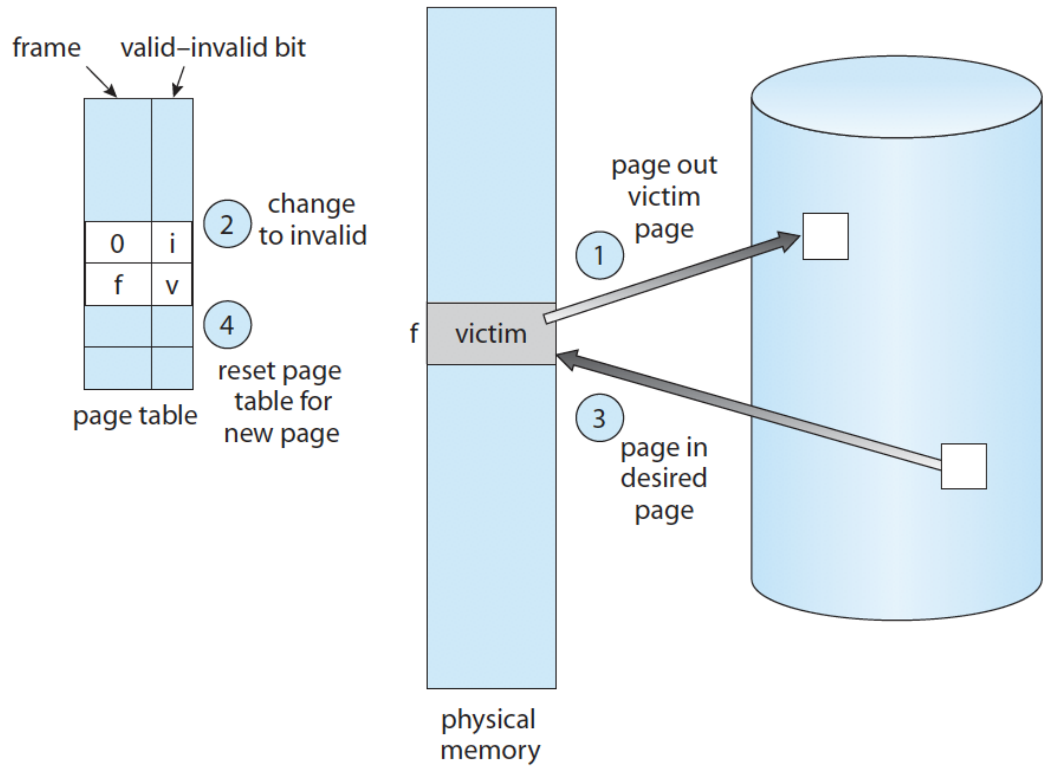
   a. If there is a free frame, use it

   b. If there is no free frame, use a page-replacement algorithm to select a victim frame

   c. Write the victim frame to the disk; change the page and frame tables accordingly

3. Read the desired page into the newly freed frame; change the page and frame tables

4. Continue the user process from where the page fault occurred

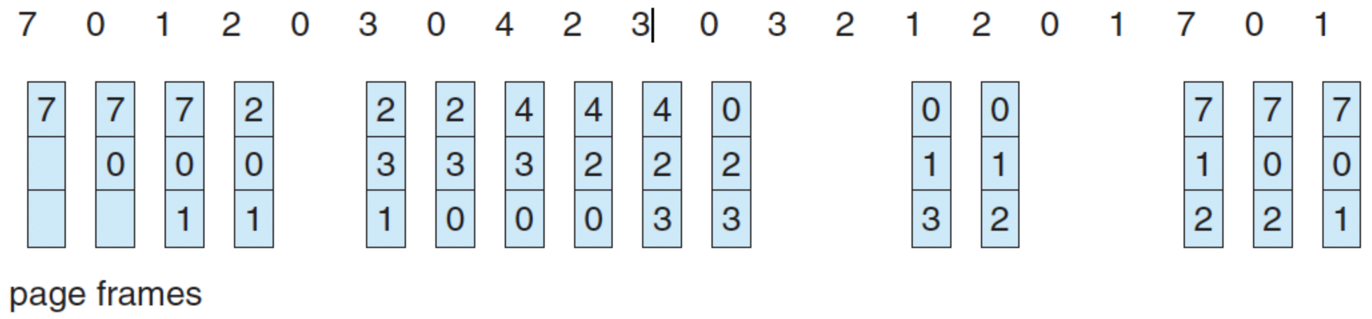


PAGE AND FRAME REPLACEMENT ALGORITHMS

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- FIFO Page Replacement
- OPT Page Replacement
- LRU Page Replacement

FIFO PAGE REPLACEMENT

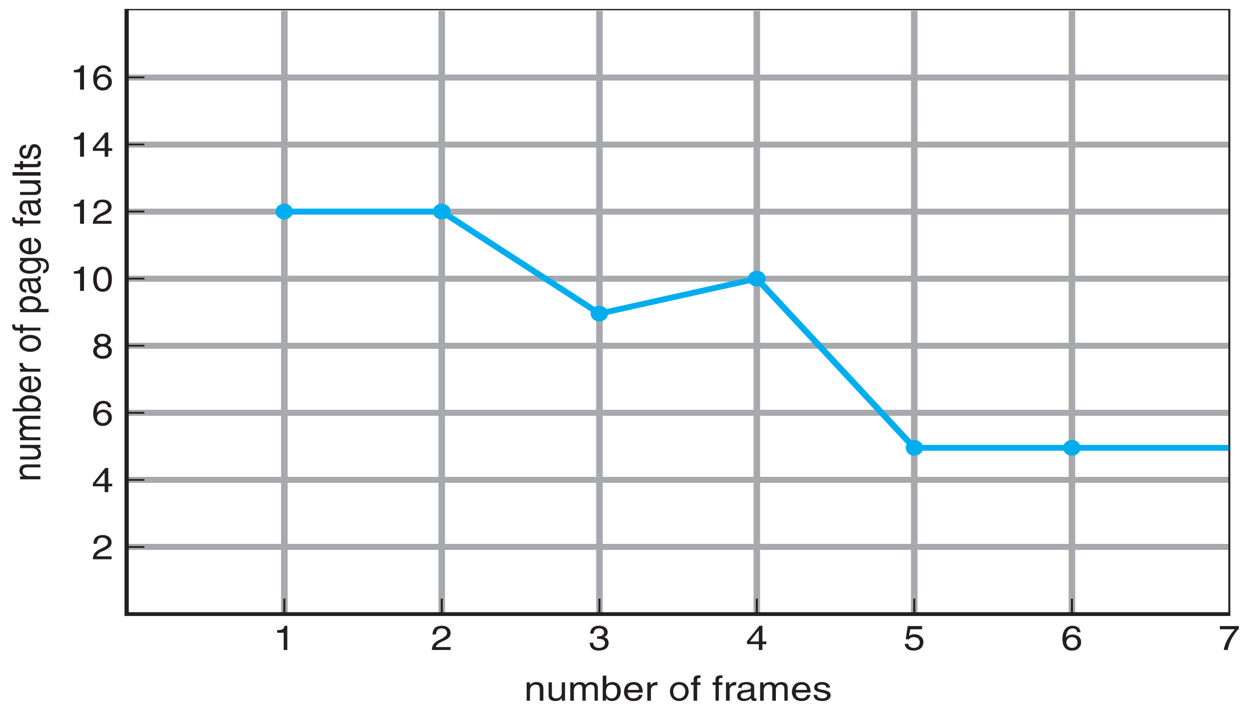
- A FIFO replacement algorithm associates with each page the time when that page was brought into memory.
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



Page fault=?

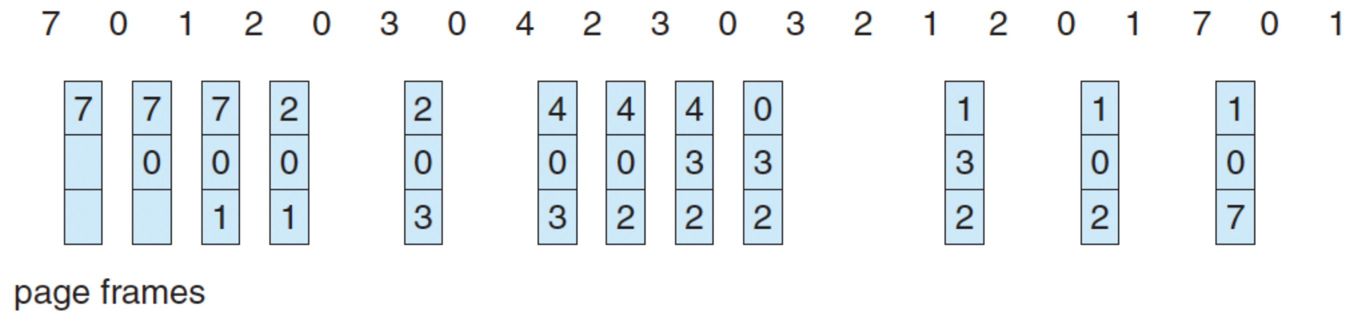
- Can vary by reference string: consider 1,2,3,2,4,1,2,3,0,3,1,5,7,1,3,2,1,2,4,5

FIFO ILLUSTRATING BELADY'S ANOMALY



LRU PAGE REPLACEMENT

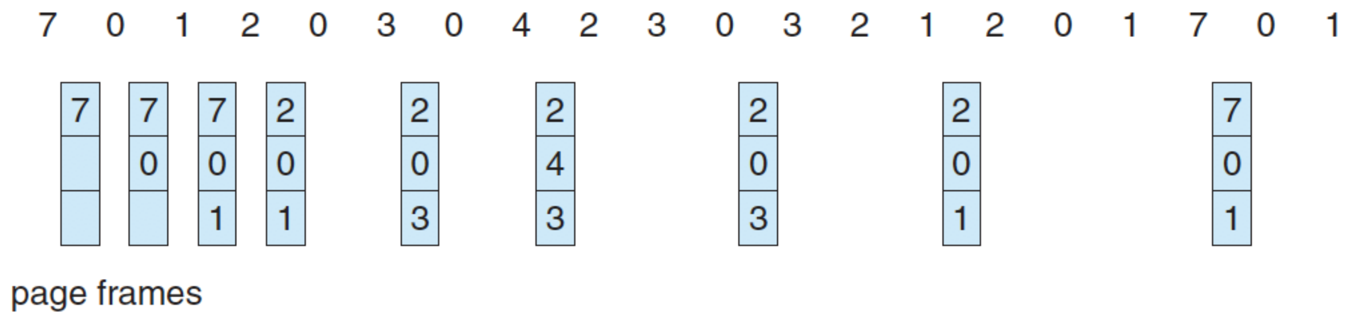
- Least Recently Used (LRU) algorithm.
- When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.



- Most Frequency Used (MFU): The page with the smallest count was probably just brought in and has yet to be used.

OPT(OPTIMAL) PAGE REPLACEMENT

- Replace the page that will not be used for the longest period of time.
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



EXERCICES

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- Reference string: 1, 3, 1, 4, 2, 1, 5, 6, 2, 1, 2, 3,6, 7, 3, 3, 2, 1, 2, 3, 6.
- Using:
  - FIFO Page Replacement
  - OPT Page Replacement
  - LRU Page Replacement

SOLUTIONS (FIFO)

- FIFO Page fault=10

1	2	3	4	2	1	5	6	2	1	2	3	6	7	3	2	1	2	3	6
1	1	1	1			5	5	5	1		1		1						1
	2	2	2			2	6	6	6		6		7						7
		3	3			3	3	2	2		2		2						6
			4			4	4	4	4		3		3						3
*	*		*			*	*	*	*		*		*						*

SOLUTIONS (RLU)

- RLU

Page fault=8

1	2	3	4	2	1	5	6	2	1	2	3	6	7	3	2	1	2	3	6
1	1	1	1			1	1				1		1						1
	2	2	2			2	2				2		2						2
		3	3			5	5				3		3						3
			4			4	6				6		7						6
*	*		*			*	*				*		*						*

SOLUTIONS

- OTP

Page fault=7

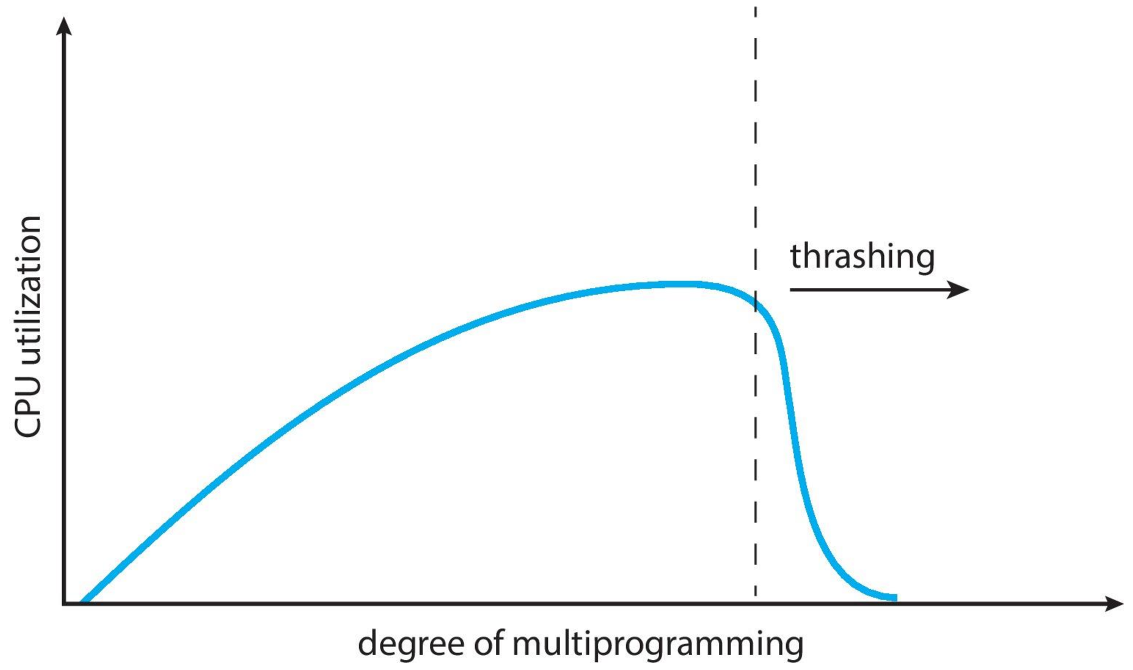
1	2	3	4	2	1	5	6	2	1	2	3	6	7	3	2	1	2	3	6
1	1	1	1			1	1						1						6
	2	2	2			2	2						2						2
		3	3			3	3						3						3
			4			5	6						7						7
*	*		*			*	*						*						*

THRASHING

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

THRASHING (CONT.)

- **Thrashing**. A process is busy swapping pages in and out


CPU utilization
thrashing
degree of multiprogramming