Operating Systems

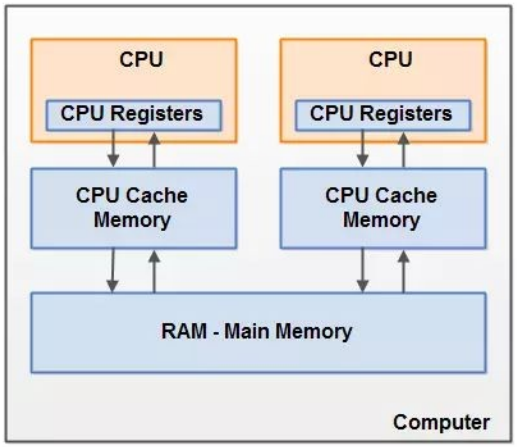
Isfahan University of Technology
Electrical and Computer Engineering Department

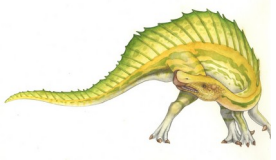
Zeinab Zali

Memory Management

Background

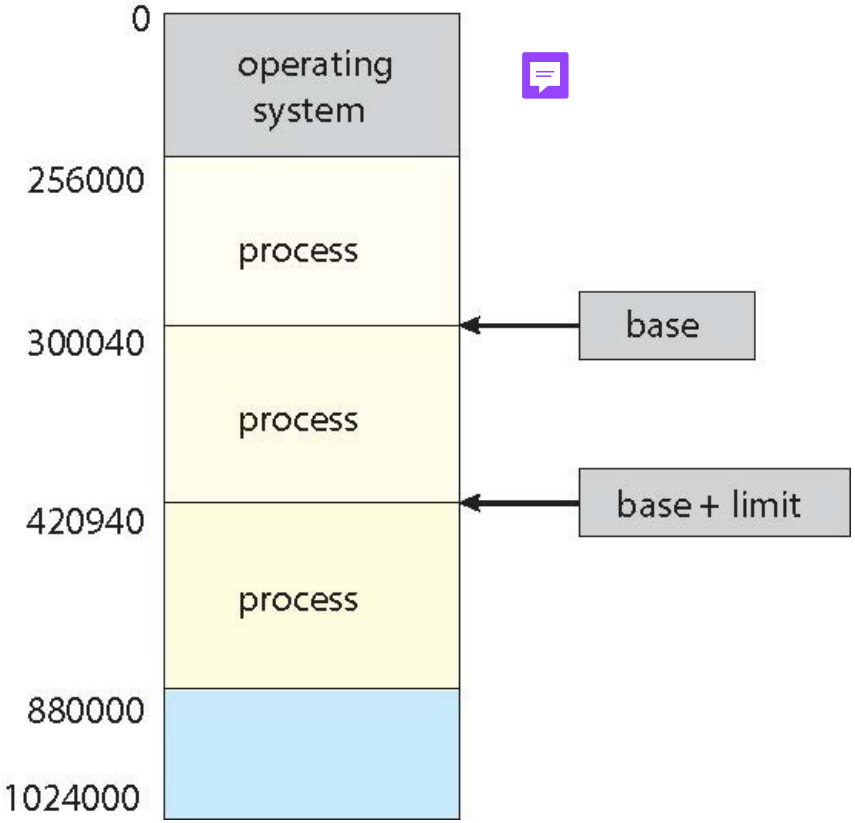
- Program must be brought (from disk) into memory and placed within a process for it to be run
- **Main memory** and **registers** are only storage CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or less)
- But 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

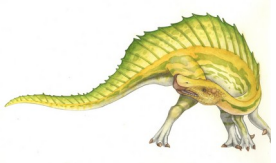




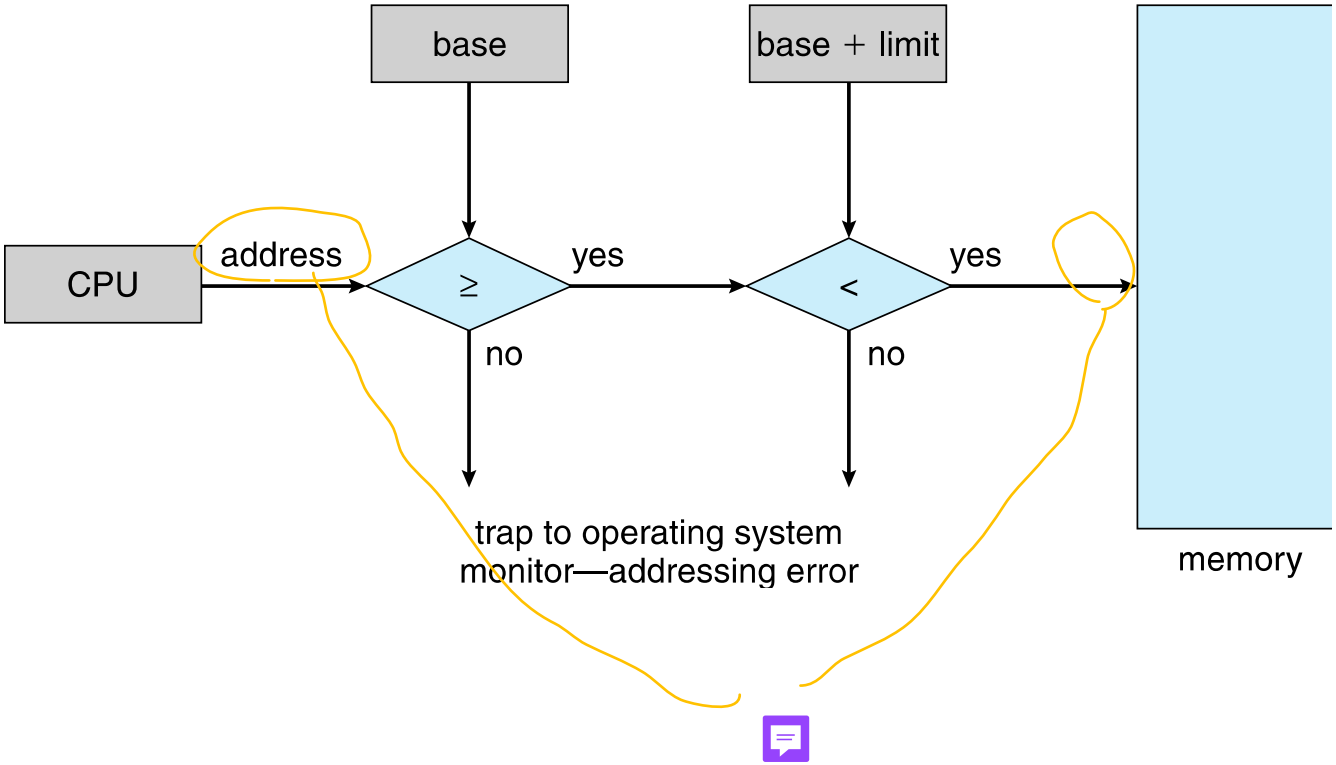
Base and Limit Registers

- 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

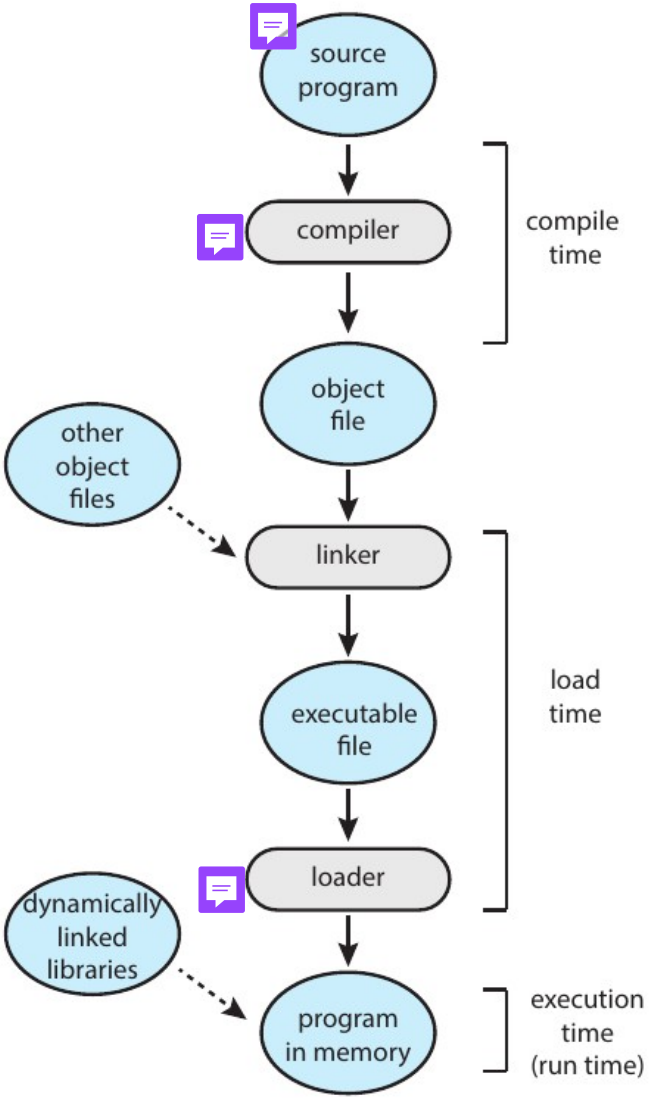




Hardware Address Protection



Multistep processing of a user program

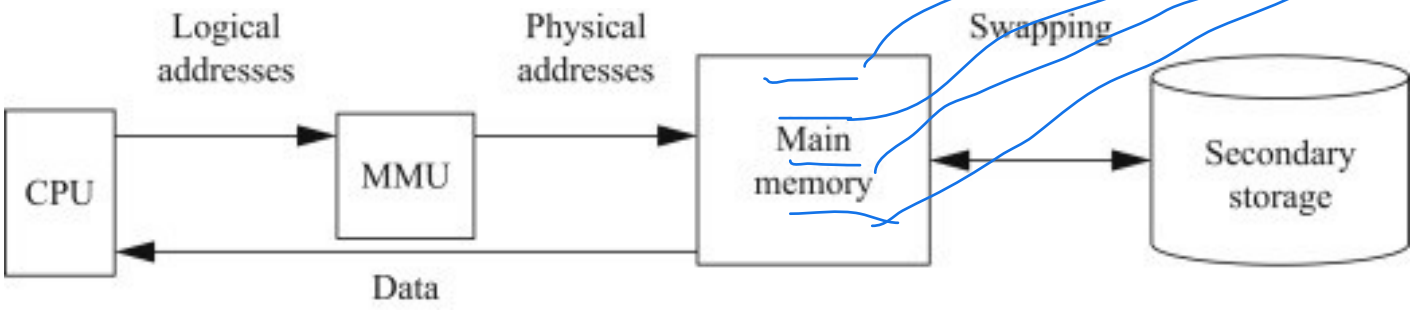


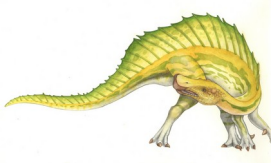
Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at different steps
  - **Compile time:** if at some later time the starting location changes, then recompiling is necessary
  - **Load time**: binding relocatable addresses (from compile time) to absolute addresses
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another
    - Need hardware support for address maps (e.g., base and limit registers)

Logical vs. Physical Address Space

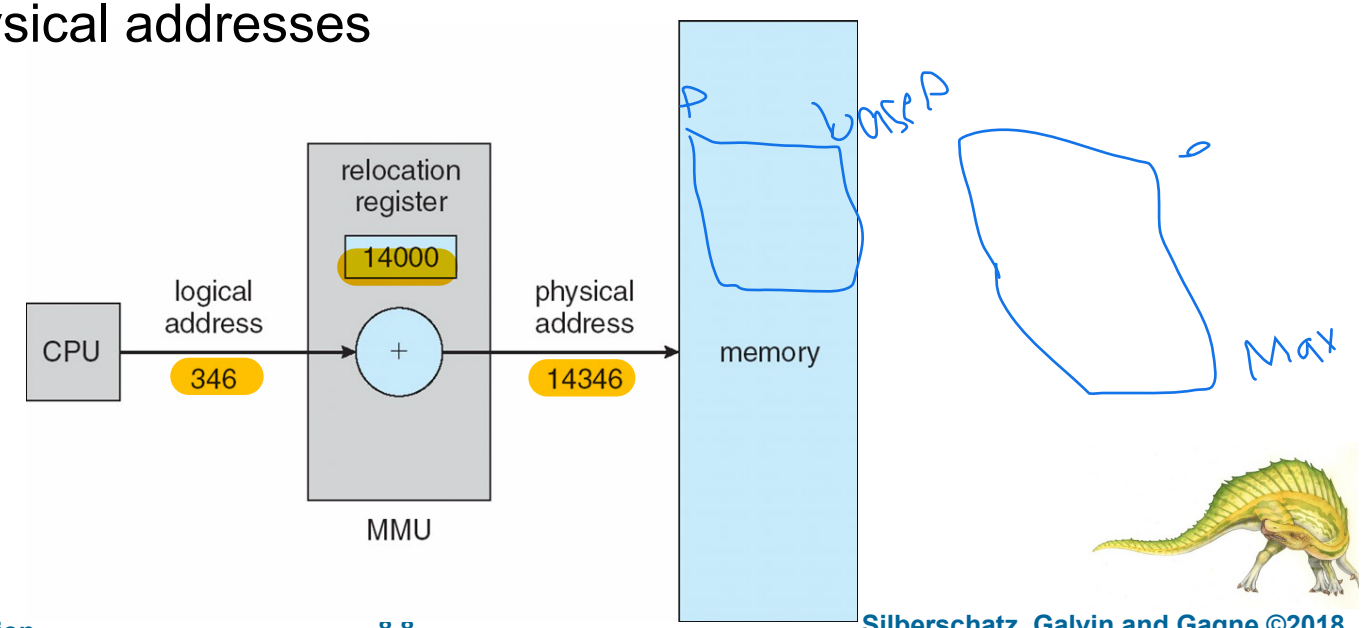
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses corresponding to the program logical addresses



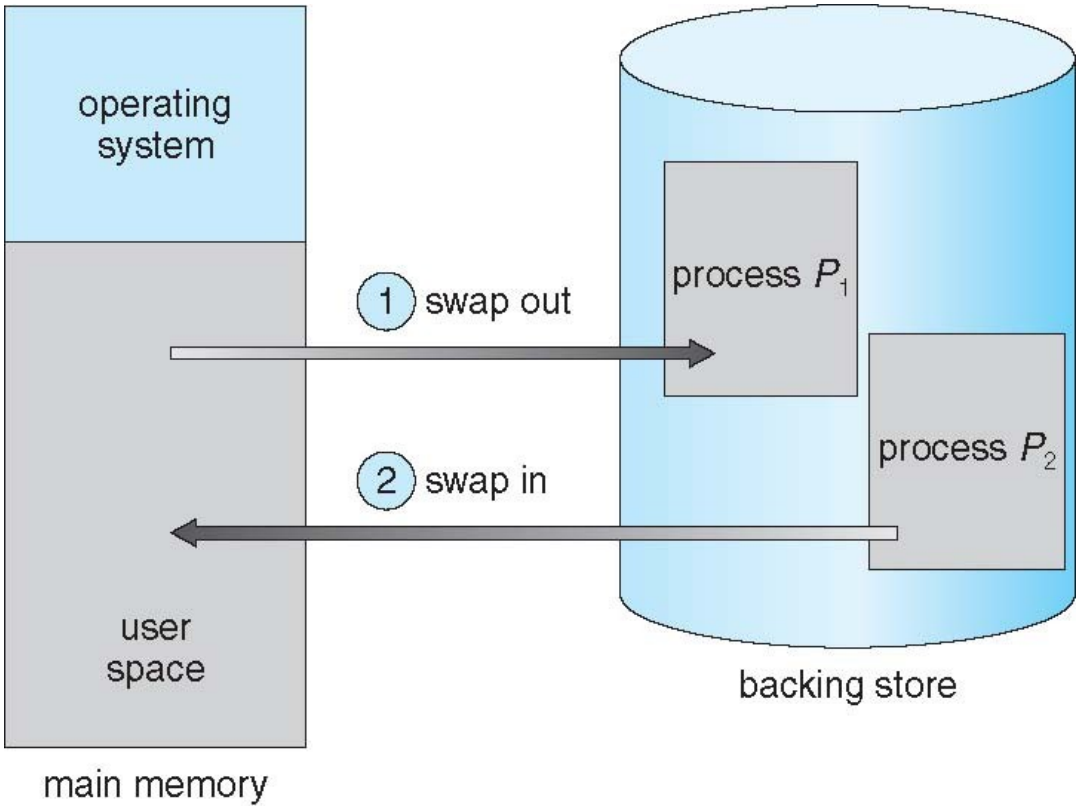


Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address
  - Usually a part of CPU
- 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



Schematic View of Swapping



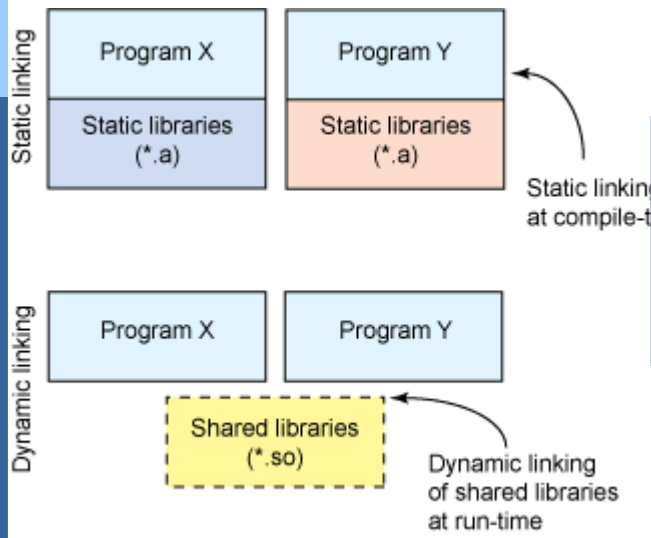
Dynamic Loading

- Until now we assumed that the entire program and data has to be in main memory to execute
- **Dynamic loading** allows a routine (module) to be loaded into memory only when it is called (used)
- Results in better memory-space utilization; an 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 (e.g., exception handling)
- No special support from the operating system is required
  - It is the responsibility of the users to design their programs to take advantage of such a method
  - OS can help by providing libraries to implement dynamic loading

```
Void * hndl dlopen("libname.so", RTLD_NOW);
Void * lib_func = dlsym(hndl, "func_name");
```

Dynamic Linking

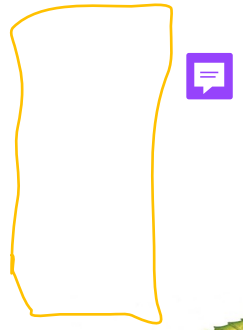
- **Dynamically linked libraries (DLL)** – system libraries that are linked to user programs when the programs are run.
  - Similar to dynamic loading. But, linking rather than loading is postponed until execution time
- Operating system checks if routine is in processes' memory address
  - If not in address space, add to address space
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

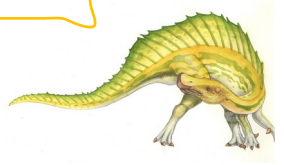



```
include library headers in your code.
In compile time, introduce the dynamic library
g++ sampleCode.c -ldynamicLibraryName
```

Memory allocation for process

- Contiguous Allocation
  - Fixed size partitions
    - Causes internal fragmentation
  - Variable size partitions
    - Causes external fragmentation
- **Segmentation**
  - Variable size segments
- **Paging**
  - Fixed size pages





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

Solution to Fragmentation problem

- 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

Segmentation

- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program

    procedure

    function

    method

    object

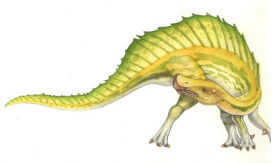
    local variables, global variables

    common block

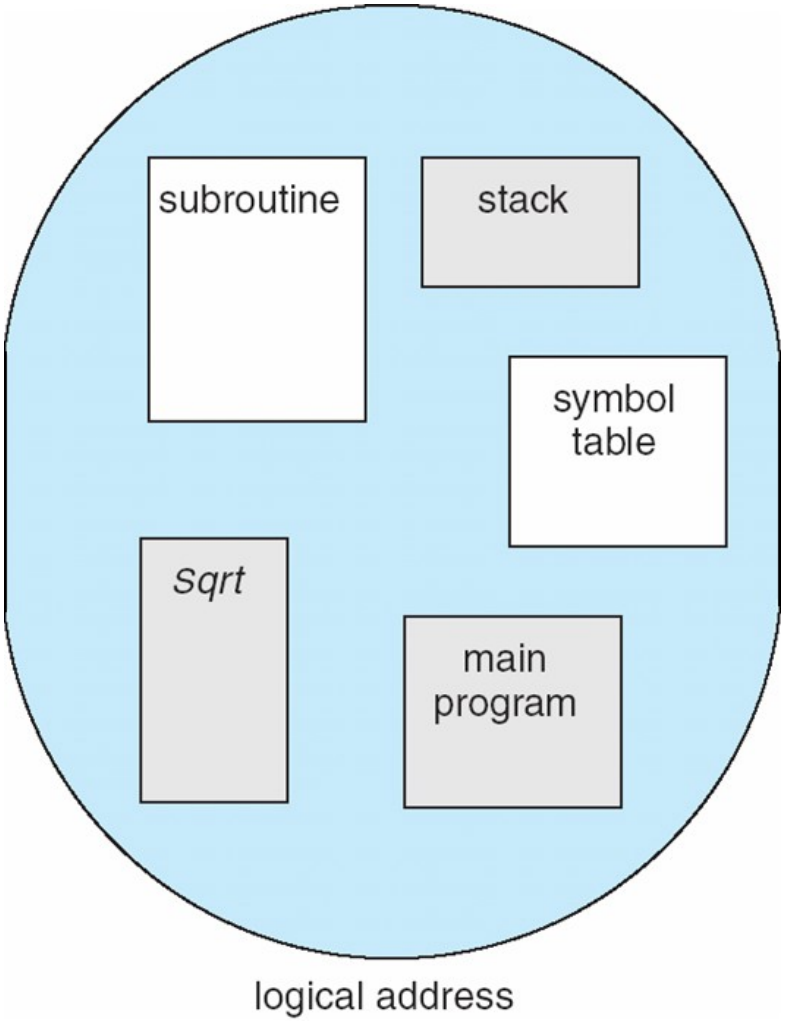
    stack

    symbol table

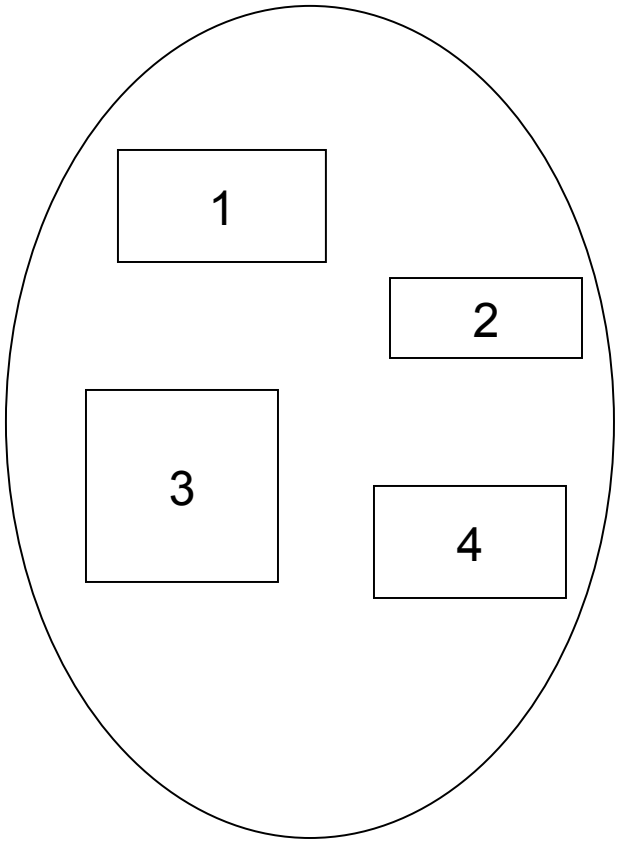
    arrays



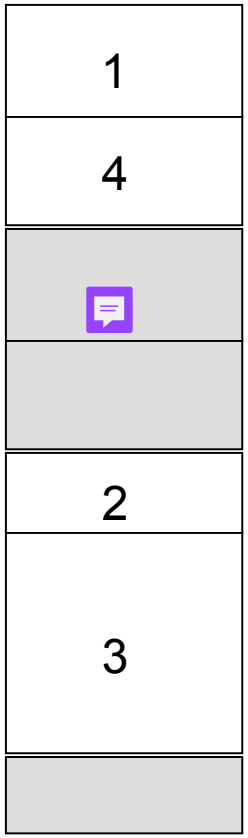
User's View of a Program



Logical View of Segmentation



user space



physical memory space

Segmentation Architecture

- Logical address consists of a two tuple:

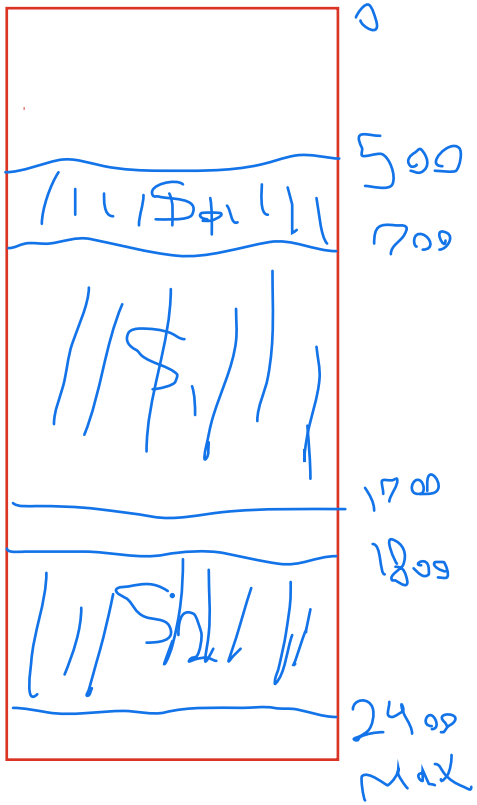
  <segment-number, offset>,

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

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

Segmentation: Example

- What is the physical addresses of below logical addresses according to the segment table?
  - 2,700
  - 0,150



Base	Length
500	200
700	1000
1800	600

0
1
2

Segmentation: Example

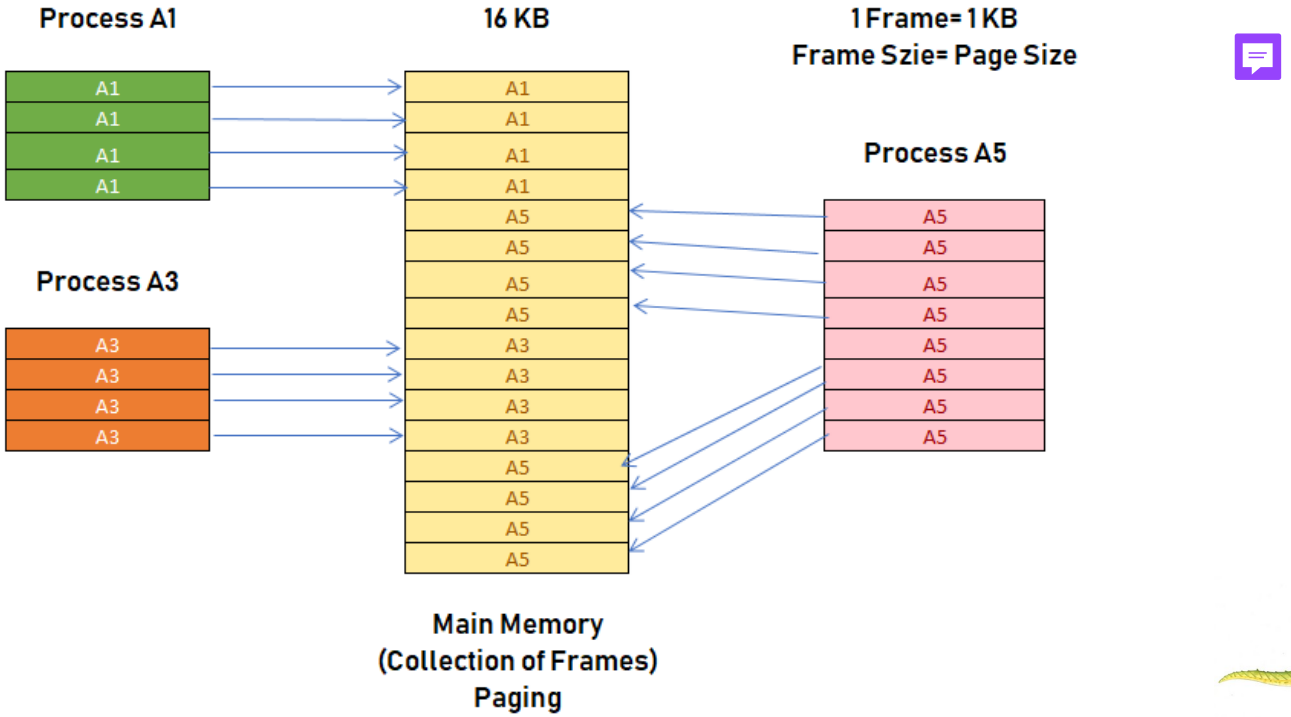
- What is the physical addresses of below logical addresses according to the segment table?
  - 2,700
  - 0,150

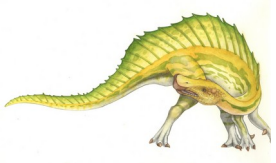
Base	Length
500	200
700	1000
1800	600

  - 2,700: Invalid
  - 0,150: 500+150=650

Paging

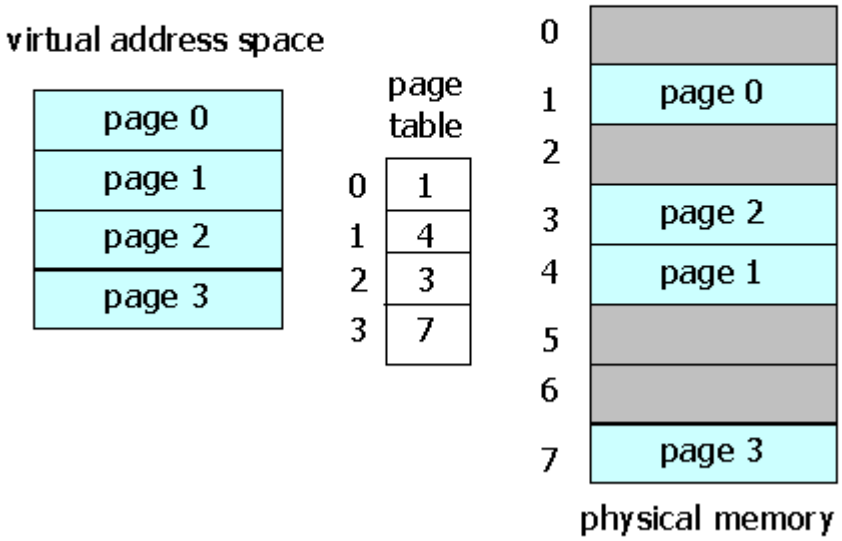
- Divide physical memory into fixed-sized blocks called **frames**
  - Size is power of 2, between 512 bytes and 16 Mbytes
- Divide logical memory into blocks of same size called **pages**

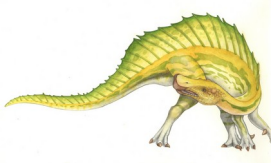




Paging

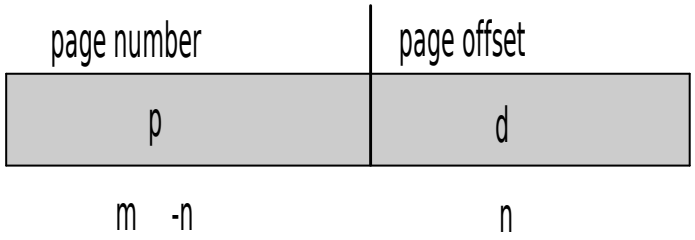
- Keep track of all free frames
- To run a program of size ***N*** pages, need to find ***N*** free frames and load program
- Set up a **page table** to translate logical to physical addresses
- **Still have Internal fragmentation**





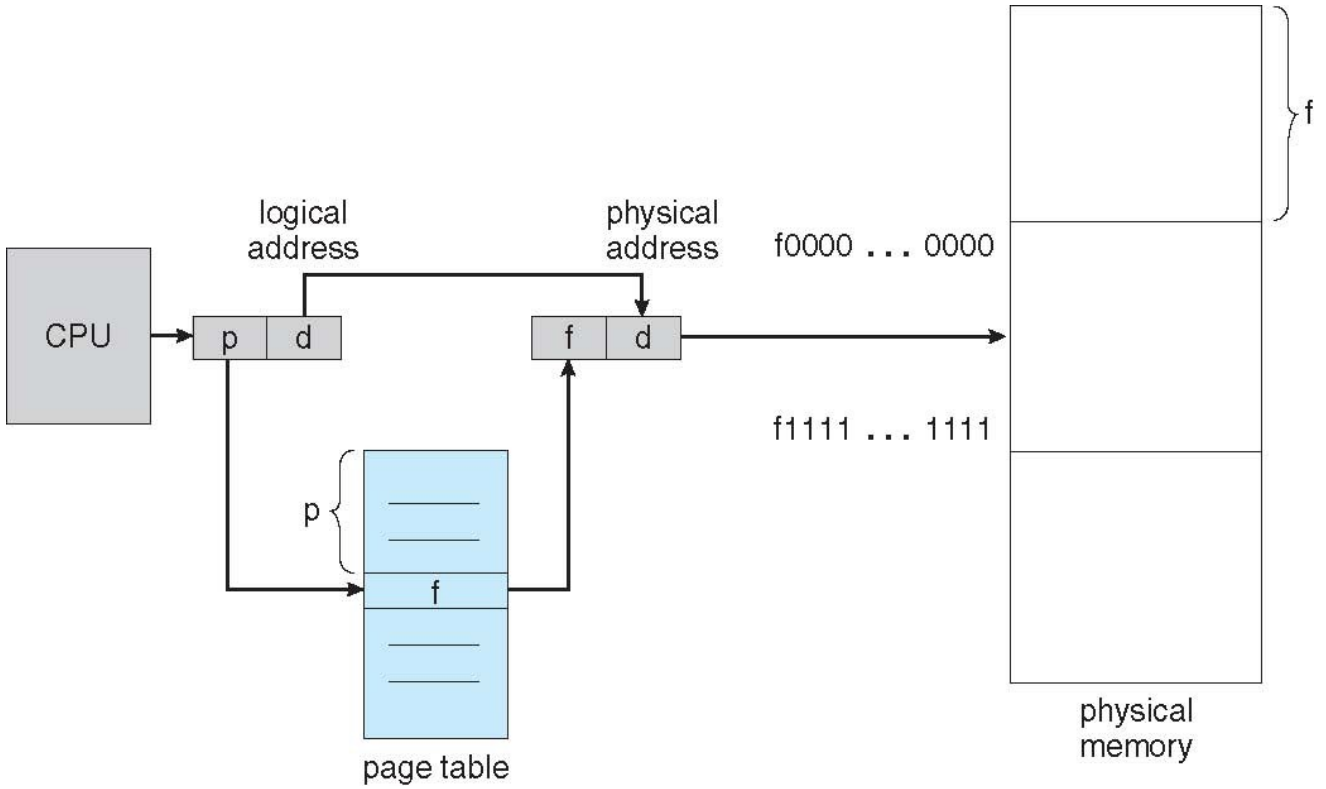
Address Translation Scheme

- 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

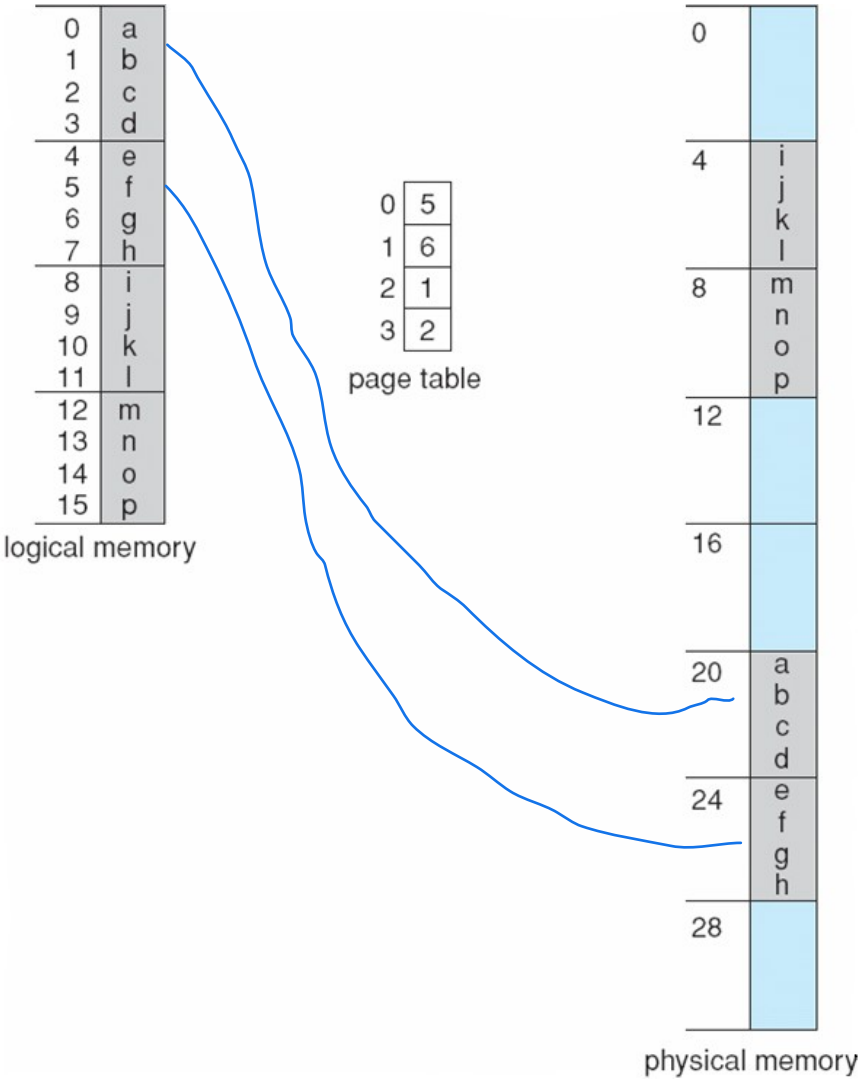


  - For given logical address space 2^m and page size 2^n

Paging Hardware



Paging Example



n=2 and *m*=4 32-byte memory and 4-byte pages

Paging: Example

- If the page size is equal to 1KB, in logical address 0000010111011110, how many pages are there in the logical memory space?

Paging: Example

- If the page size is equal to 1KB, in logical address 0000010111011110, how many bits are required for the page number?
  - 1 KB = 10 bits for page offset
  - 0000010111011110

16 bits

16-10=6 bits for the page #

So 2^6 pages in memory space

Paging: Example

- There are 4 pages in a logical address space and each page has 2 words. If these pages are assigned to a physical address space with 8 frames, how many bits are required for physical and logical addresses?

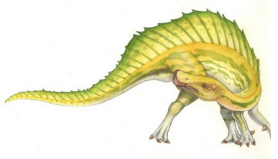
Paging: Example

- There are 4 pages in a logical address space and each page has 2 words. If these pages are assigned to a physical address space with 8 frames, how many bits are required for physical and logical addresses?
  - Logical address: 4 pages = 2bits, 2 words=1 bit => 3 bits
  - Physical address: 8 frames=3 bits, 2 words=1 bit => 4 bit

Paging (Cont.)

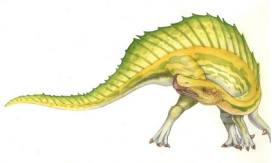
- Calculating internal fragmentation
  - Page size = 2,048 bytes
  - Process size = 72,766 bytes
  - 35 pages + 1,086 bytes
  - Internal fragmentation of 2,048 - 1,086 = 962 bytes
  - Worst case fragmentation = 1 frame – 1 byte
  - On average fragmentation = 1 / 2 frame size
- So small frame sizes desirable?
  - But each page table entry takes memory to track
  - Page sizes growing over time
    - Windows 10 supports page sizes of 4 KB and 2 MB .
    - Linux also supports two page sizes: a default page size (typically 4 KB) and an architecture-dependent larger page size called huge pages

getconf PAGESIZE



Paging (Cont.)

- Process view and physical memory now very different
  - The programmer views memory as one single space, containing only this one program
  - By implementation process can only access its own memory
- The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process (or processes).



Increasing or Decreasing page size

- What is the benefits of increasing page size?
- In what situation do you suggest to increase the page size?

Implementation of Page Table

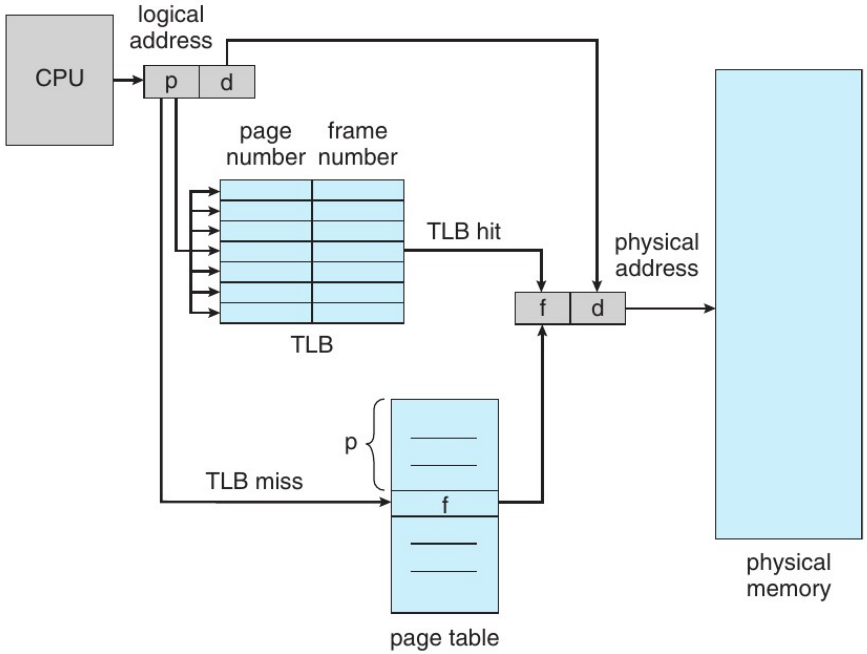
- Page table is kept in main memory
- **Page-table base register** (**PTBR**) points to the page table
- **Page-table length register** (**PTLR**) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
  - One for the page table and one for the data / instruction
- 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**)

TLB

- The TLB is associative, high-speed memory.
- Each entry in the TLB consists of two parts: a key (or tag) and a value.
- When the associative memory is presented with an item, the item is compared with all keys simultaneously.
  - If the item is found, the corresponding value field is returned. The search is fast;
- a TLB lookup in modern hardware is part of the instruction pipeline, essentially adding no performance penalty.
- To be able to execute the search within a pipeline step, however, the TLB must be kept small

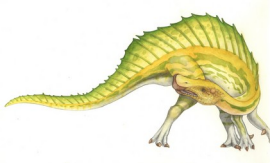
TLB

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
  - If the page number is found, its frame number is immediately available and is used
  - If the page number is not found, memory reference to the page table must be made.

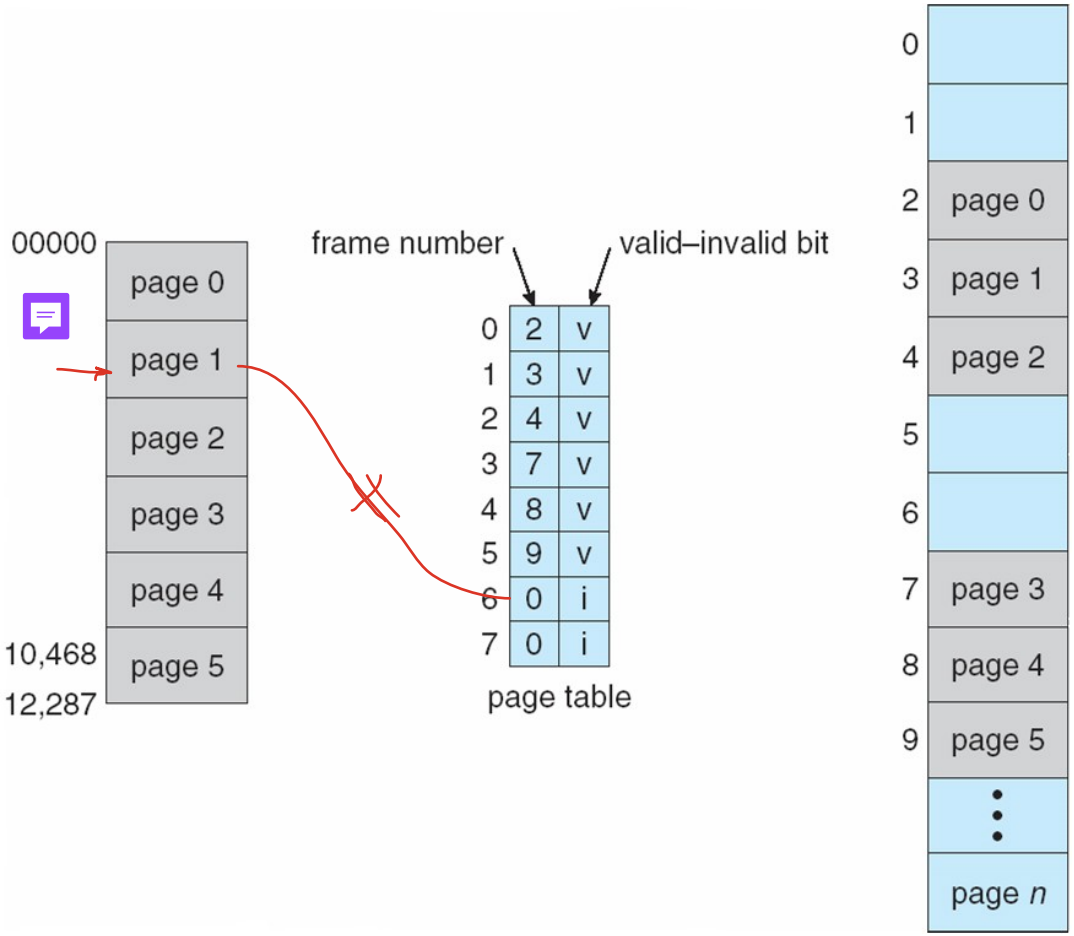


Memory Protection

- 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

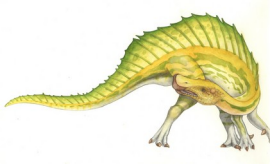


Valid (v) or Invalid (i) Bit In A Page Table

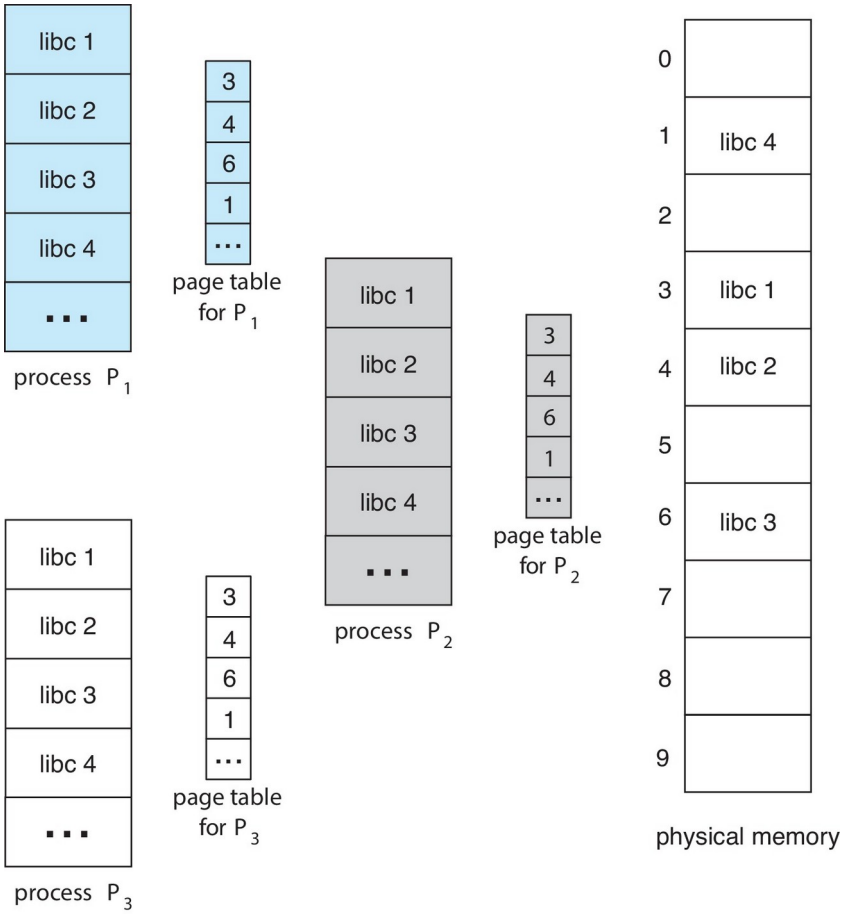


Shared Pages

- **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
  - Page size of 4 KB (2^{12})
  - Page table would have 1 million entries ($2^{32} / 2^{12}$)
  - If each entry is 4 bytes, each process needs 4 MB of physical address space for the page table alone
    - Don't want to allocate that contiguously in main memory

Structure of the Page Table

- One simple solution is to divide the page table into smaller units
  - Hierarchical Paging
  - Hashed Page Tables
  - Inverted Page Tables