

Main Memory

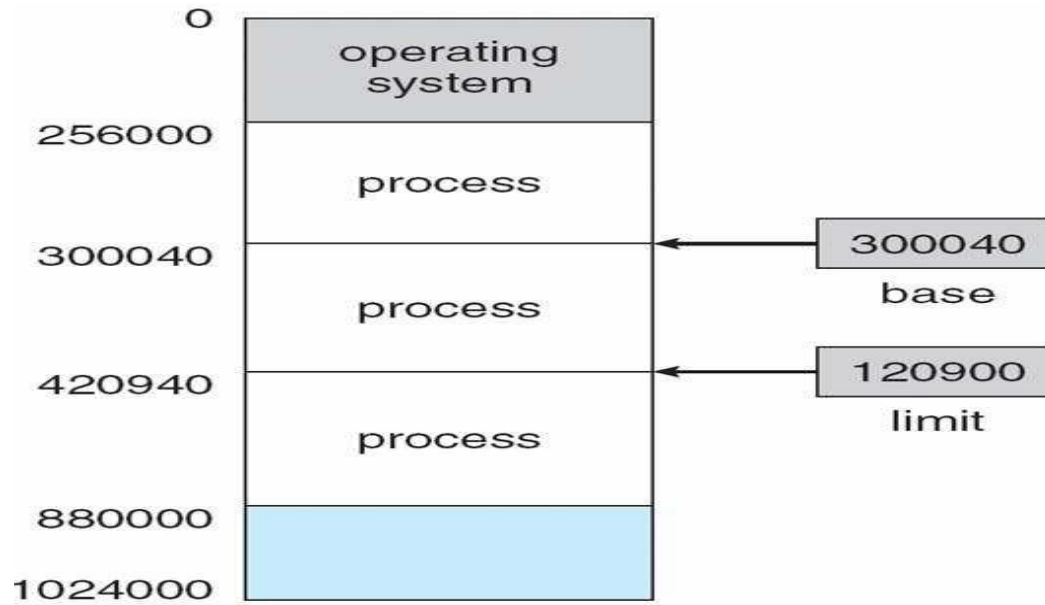
Course Instructor: Safia Baloch

Background

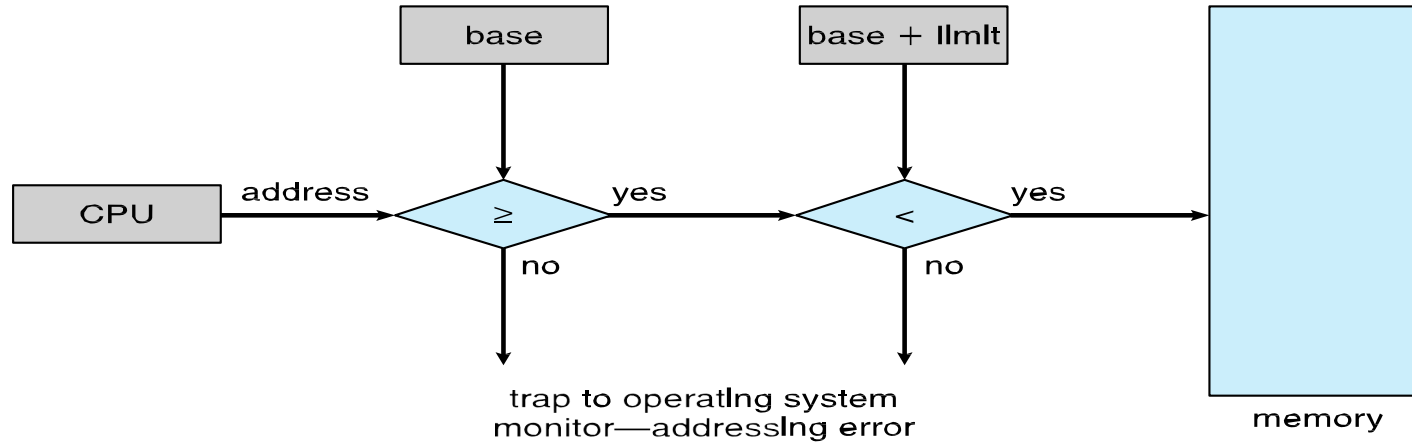
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage, CPU can access directly
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- Register access in one CPU clock (or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

- A pair of **base** and **limit registers** define the logical address space for a process
- 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 with Base and Limit Registers



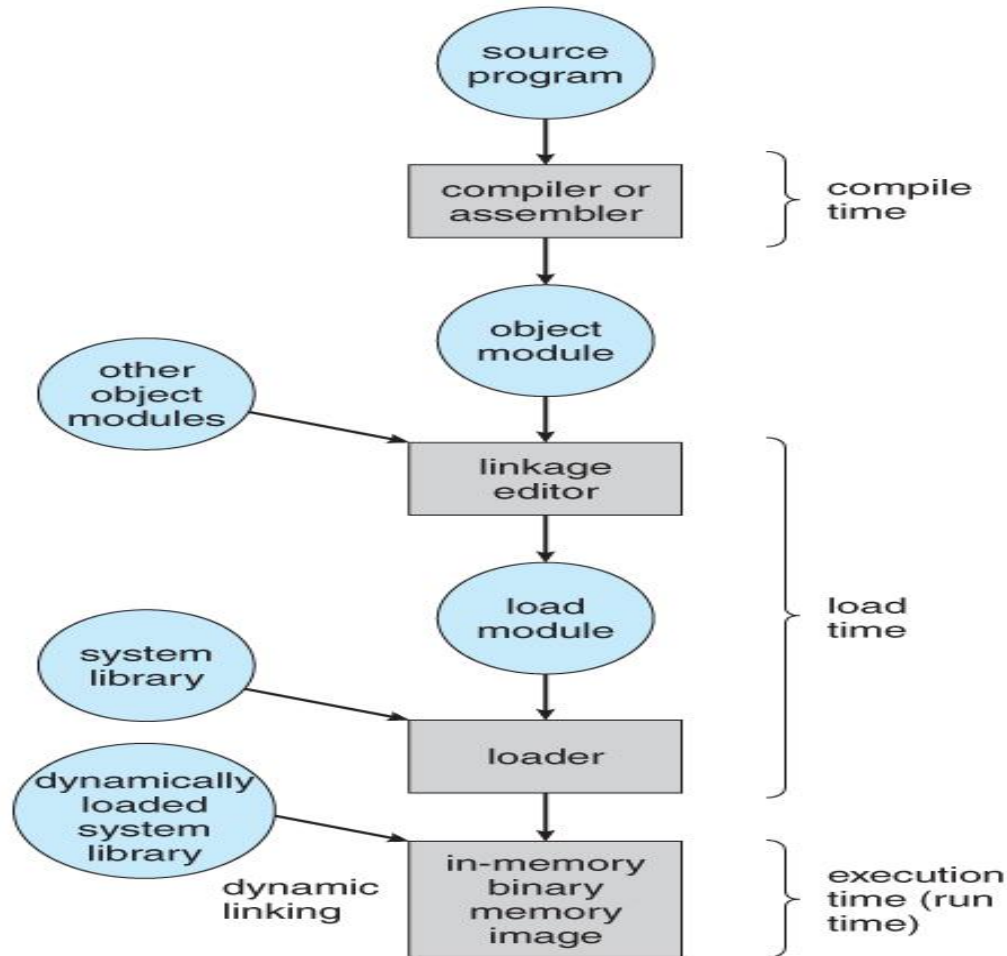
Address Binding

- 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
 - Because most of the time, system processes are allocated in this range.
- Further, addresses represented in different ways at different stages of a program's life
 - Source code addresses usually symbolic: variables
 - Compiled code addresses **bind** to relocatable addresses
 - Linker or loader will bind relocatable addresses to absolute addresses
 - 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 and here
 - Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program



Logical vs. Physical Address Space

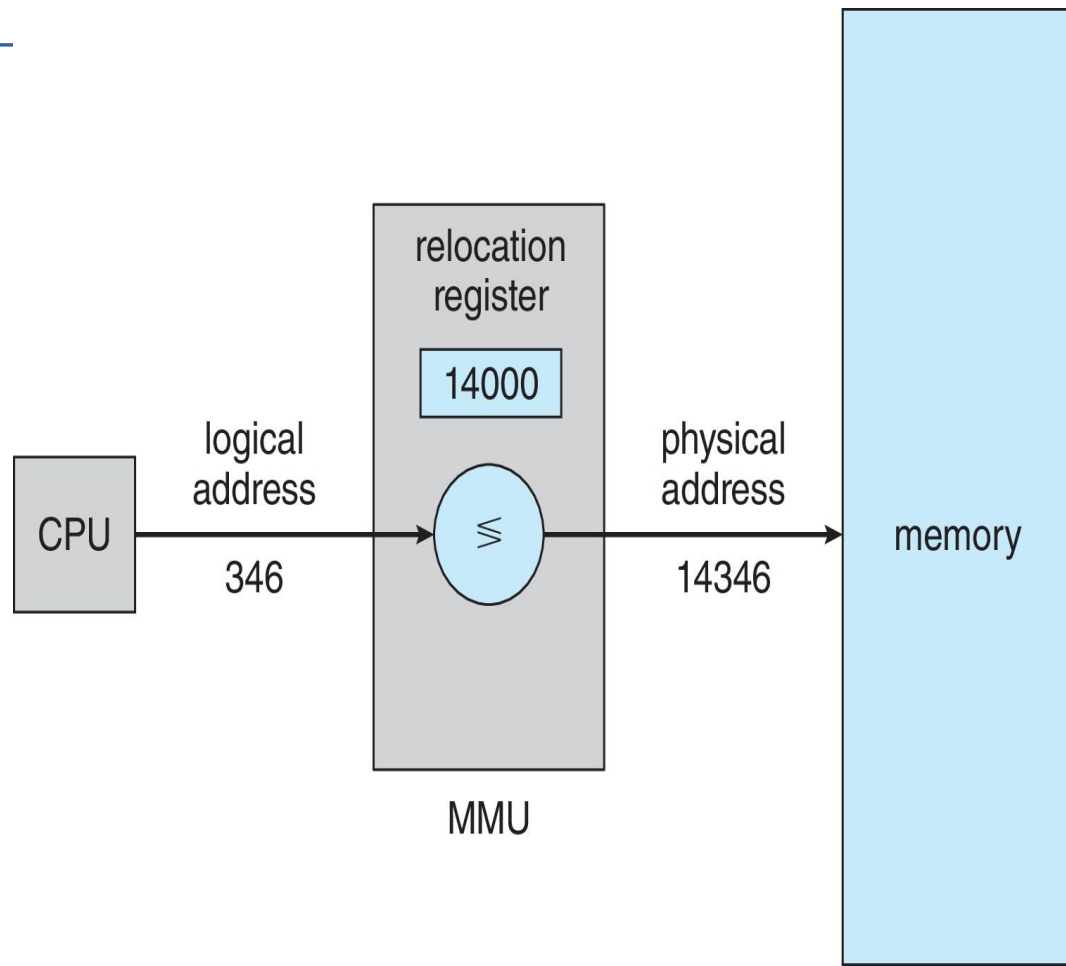
- The concept of a logical address space that is bound to a separate
- 1) **Logical address** – generated by the CPU; also referred to as **virtual address**
2) **Physical address** – address seen by the memory unit
- **Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme**
- **Logical address space** is the set of all logical addresses generated by a program
- **Physical address space** is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)

- MMU is a hardware device that at run time maps virtual address to physical address
- 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**
 - MS-DOS on Intel 80x86 used 4 relocation registers
- 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

Dynamic relocation using a relocation register

- 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



Static & 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
- Dynamic linking is particularly useful for libraries

Swapping [1/2]

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

Backing store – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

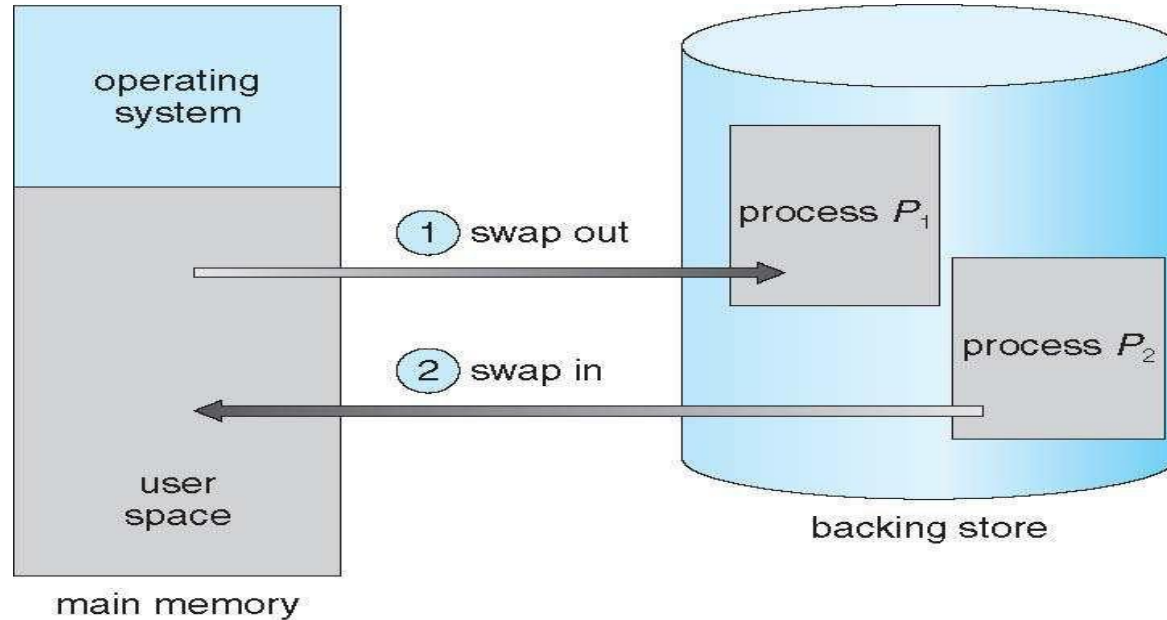
Roll out, roll in – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

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

Swapping [2/2]

- System maintains a **ready queue** of ready-to-run processes which have memory images on disk
- Does the swapped out process need to swap back in to same physical addresses?
- Depends on address binding method
 - Plus consider pending I/O to / from process memory space
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
 - Swapping normally disabled
 - Started if more than threshold amount of memory allocated
 - Disabled again once memory demand reduced below threshold

Schematic View of Standard Swapping



Context Switch Time including Swapping [1/2]

- 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
- $\text{Swap in Time} = \text{size of (process)} / \text{transfer rate}$
- 100MB process swapping to hard disk with transfer rate of 50MB/sec = 2 sec or 2000msec.
 - Swap out time = 2000 ms
 - Plus swap in of same sized process
 - Total context switch swapping time (Swap In +out) =4000ms (4 seconds)

Contiguous Allocation(not contagious;it was only to wake you up)

- 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

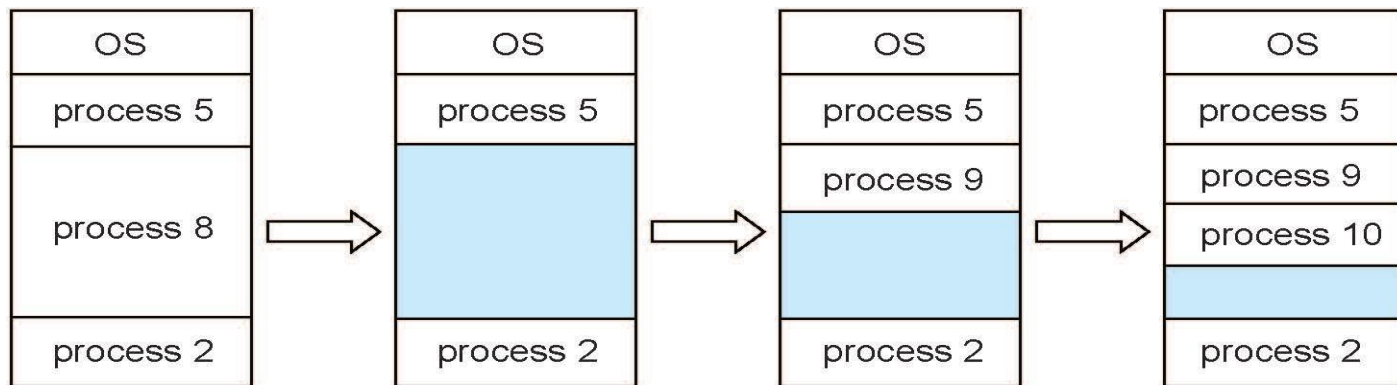
Contiguous Allocation (Cont.)

- 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*
 - Can then allow actions such as kernel code being **transient** and kernel changing size

Multiple-partition allocation

- 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)



Dynamic Storage-Allocation Problem

How to satisfy a request of size n from a list of free holes?

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
 - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
 - Produces the largest leftover hole

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

Fragmentation

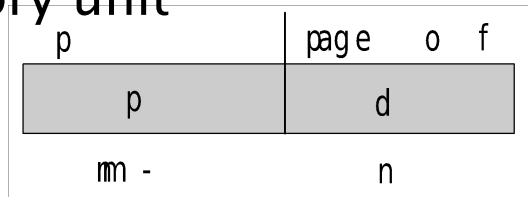
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals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

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 16 Mbytes
- 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: How?

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

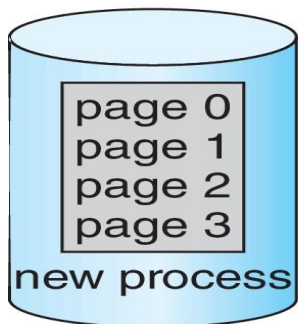


- For given logical address space 2^m and page size 2^n

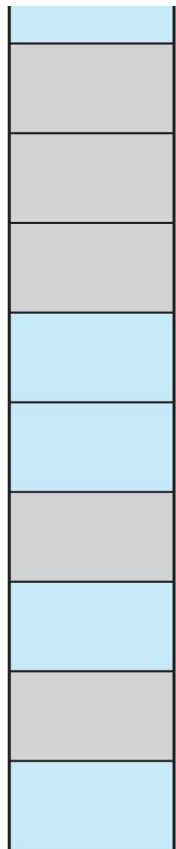
Paging Model of Logical and Physical Memory

free-frame list

14
13
18
20
15



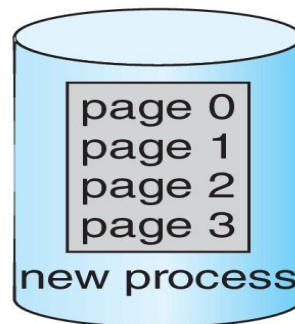
13
14
15
16
17
18
19
20
21



(a)

free-frame list

15



0	14
1	13
2	18
3	20

new-process page table

13
14
15
16
17
18
19
20
21



(b)

Paging Example

- The page size is $2^2 = 4$,
- the size of the logical memory is 2^4 bytes = 16 bytes, and
- the size of the physical memory is 2^5 bytes = 32 bytes.
- The page table maps logical page 0 to physical frame 5, and so on.

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

logical memory

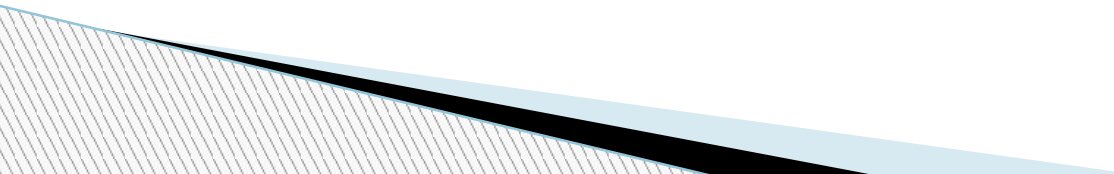
0	5
1	6
2	1
3	2

page table

0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

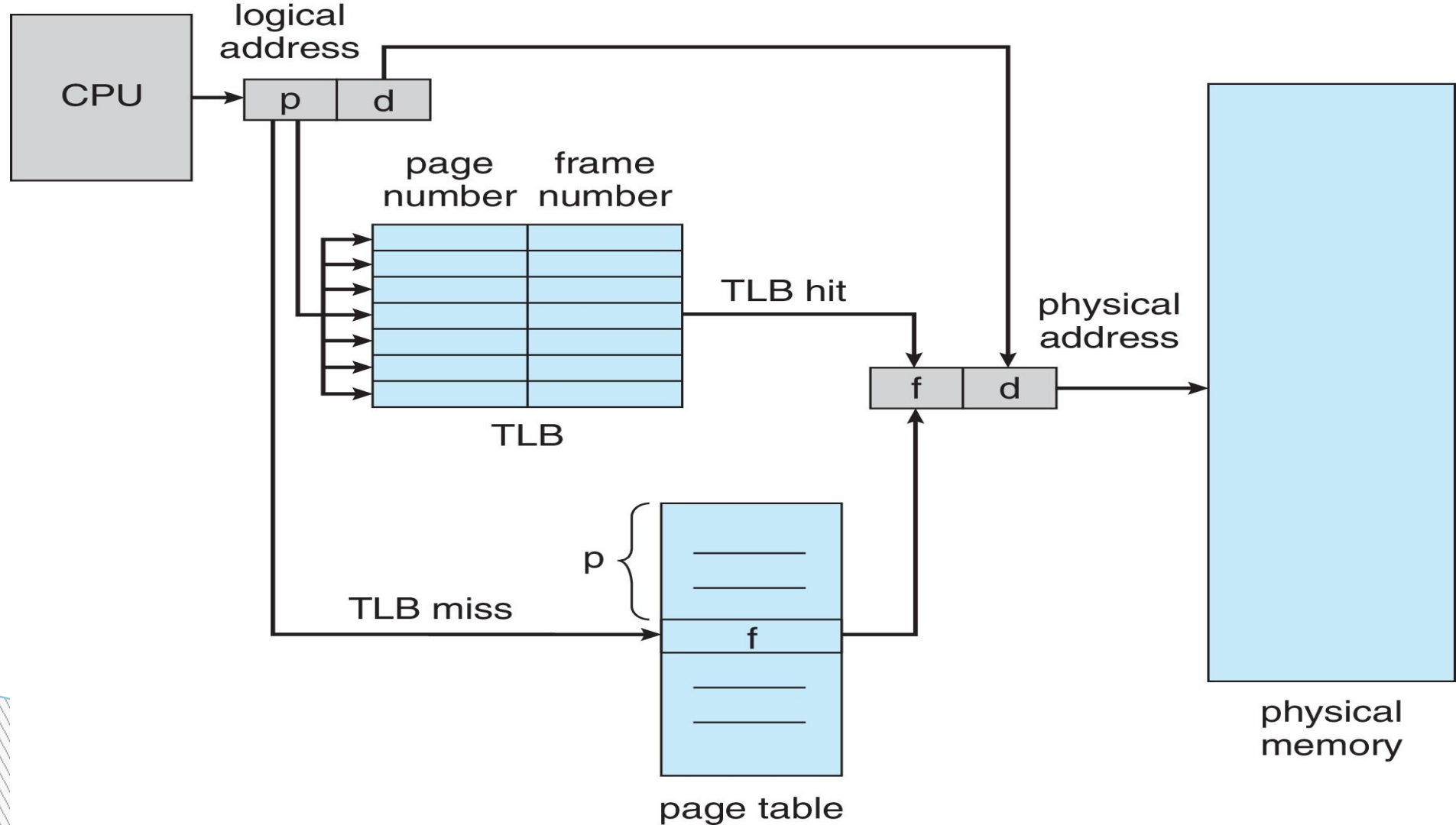
physical memory

Brain Buster

1. Internal fragmentation can be avoided (not completely) with small size of page or large size of page?
 2. Advantage of large size of page?
 3. Frames can be non contiguous but page must be ,WHY?
 4. For logical/physical address in bits we do $m+n$, for bits in a page we do?
- 

Implementation of Page Table

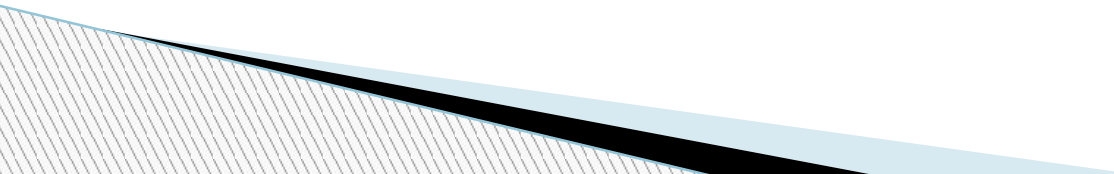
- Page table is kept in main memory
- **Page-table base register (PTBR)** points to the page table
- **Page-table length register (PTLR)** indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses
 - One for the page table and one for the data / instruction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



Effective memory Access Time:EAT

Effective memory access time (EAT) is a function of the hit ratio, memory access time, and TLB search time.

For example, if the hit ratio is 90% and the memory access time is 12 nanoseconds, then, according to our simple model, the EAT would be calculated as $(0.9)(12) + (0.1)(24) = 13.2$ ns.



Paging: Hardware support

Storing the page table in main memory result in slower memory access time. Solution TLB

TLB: each entry in consist of 1) Key, 2) value

Search item with all keys, if found then returned value of it.

TLB Hit,

TLB Miss

TLB full of entries?



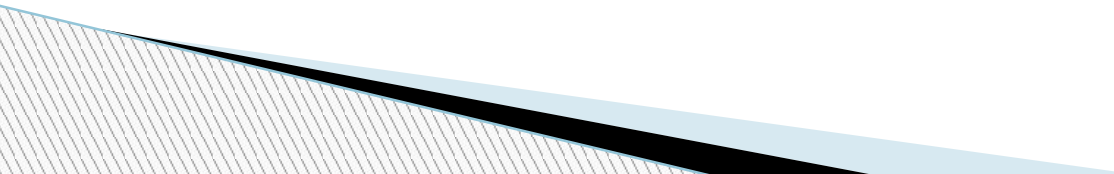
Replacement policies

In case of TLB full with entries, existing pages are replaced with current one with help of different algorithms. LRU, Round Robin , second chance etc.

Wired Down entries in TLB: Kernel entries

ASIDs: address space identifier to uniquely identify each process.

In-case of virtual page number, it helps to resolve, and help TLB to have several different process entries simultaneously.



Paging: Hardware support

Protection bit: Associate a bit to protect from illegal attempts of physical address (frames)

Valid_invalid bit: if page is in logical address space then bit is valid else invalid.

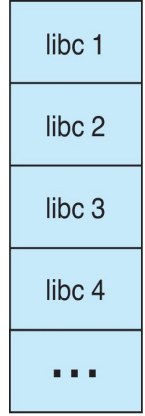
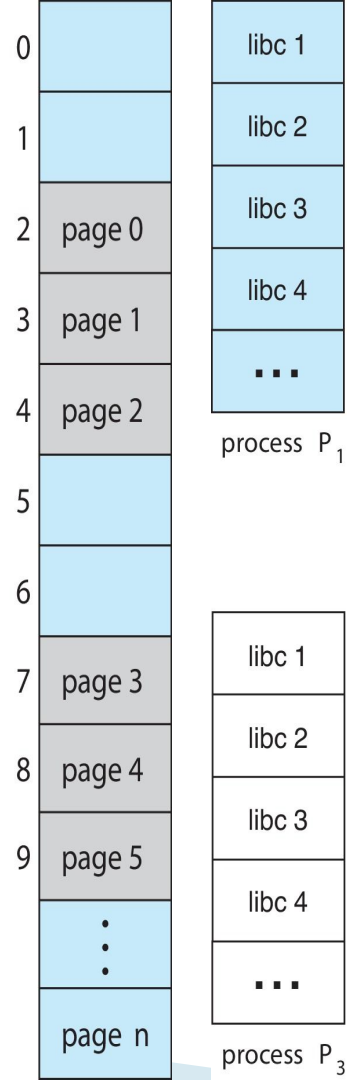
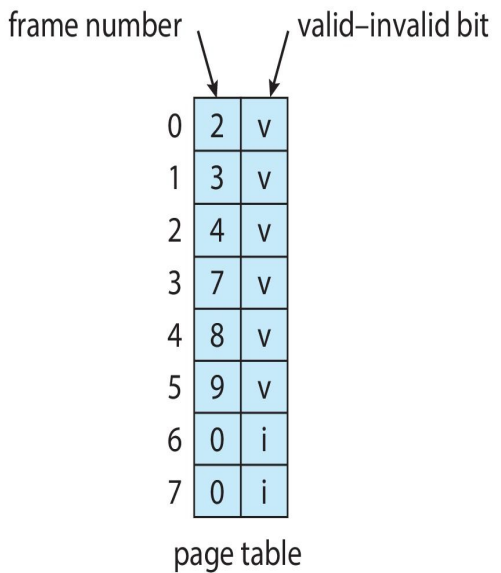
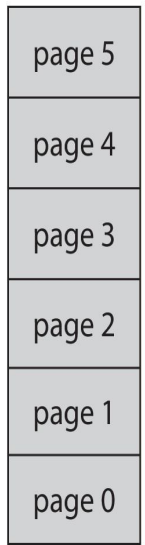
Shared Pages: process sharing common code: c library etc

40 user, 2 MB of libc



12,287
10,468

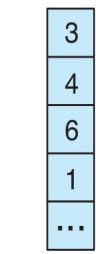
00000



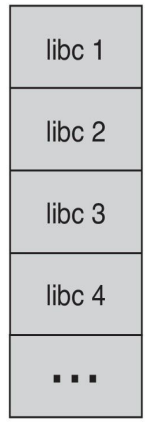
process P₁



process P₃



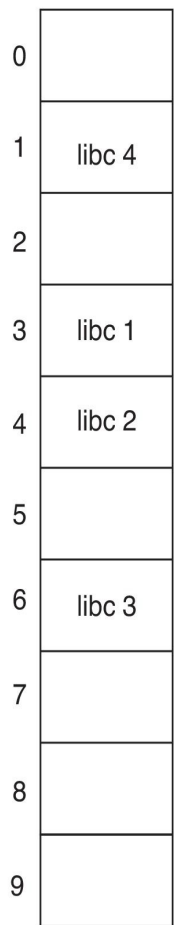
page table for P₁



process P₂



page table for P₂

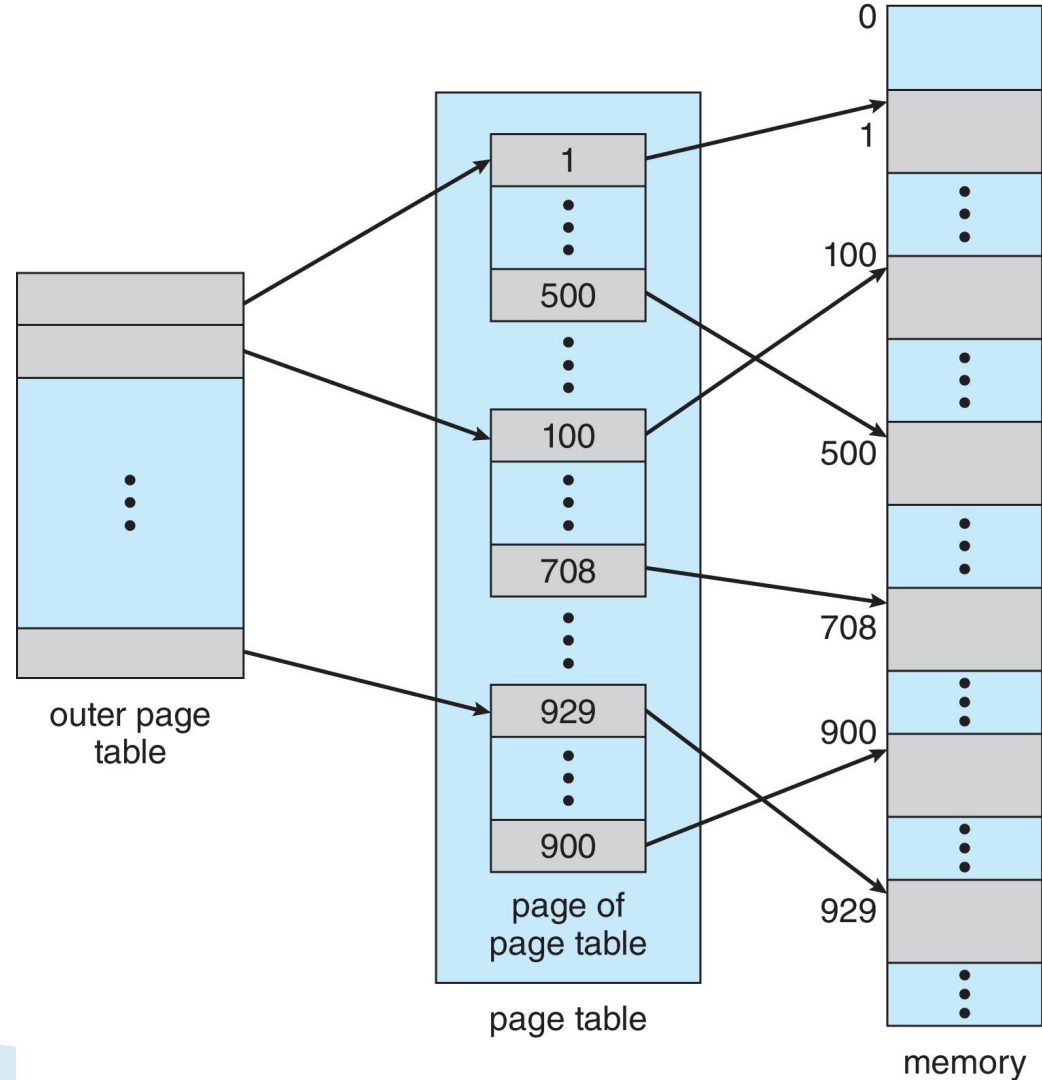


physical memory

Structure of Page Table

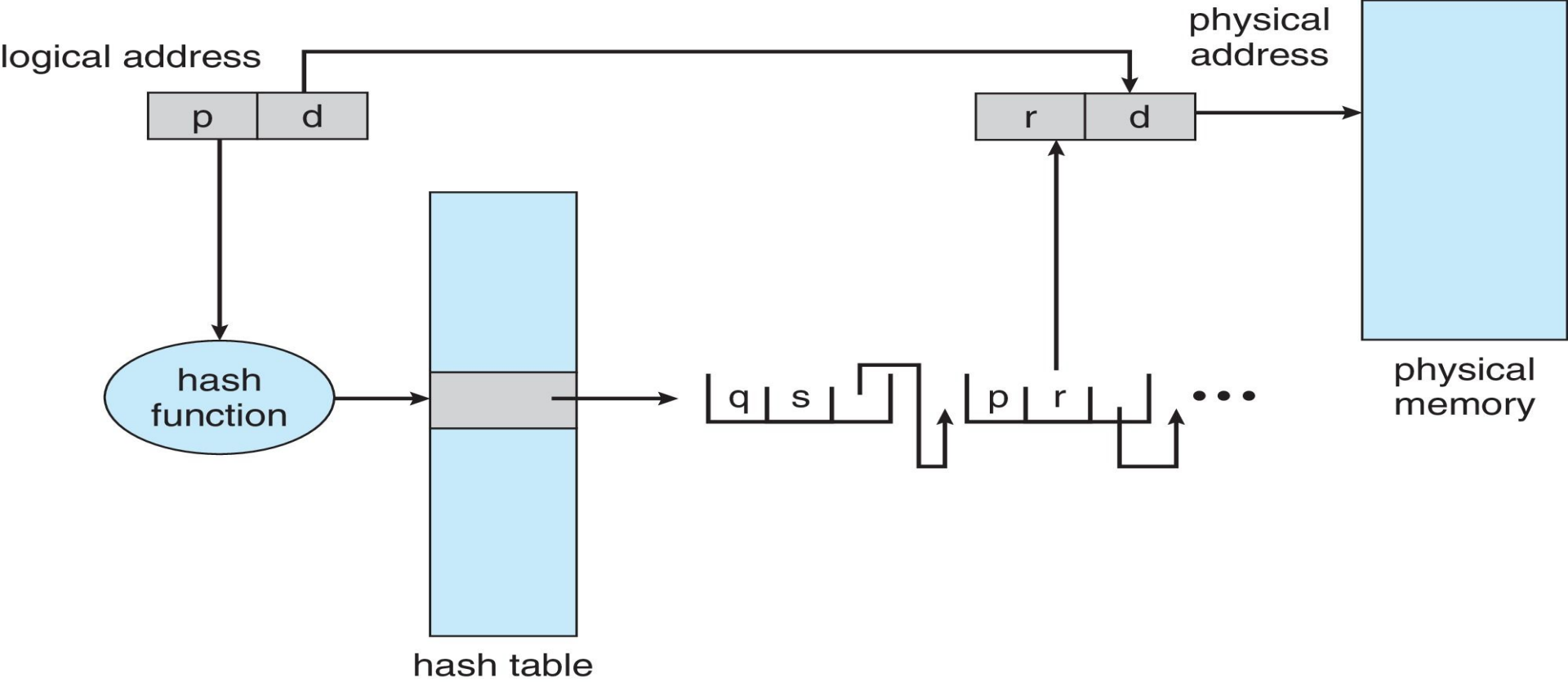
Hierarchical Paging:

Page the page table:



Structure of Page Table

Hashed Page Table:



Structure of Page Types

Inverted Page table: (Home Task)

- ❑ Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages
- ❑ 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
 - TLB can accelerate access

Attendance : Section A

0265

1362

0297

0471

0241

1747

1032

0334

0186

