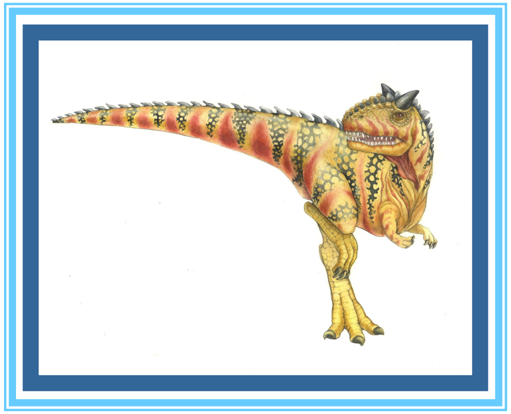
Chapter 9: Main Memory



Chapter 9: Memory Management

- Background
- Contiguous Memory Allocation
- Paging
- Structure of the Page Table
- Swapping
- Example: The Intel 32 and 64-bit Architectures
- Example: ARMv8 Architecture

Objectives

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques,
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging

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 is done 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

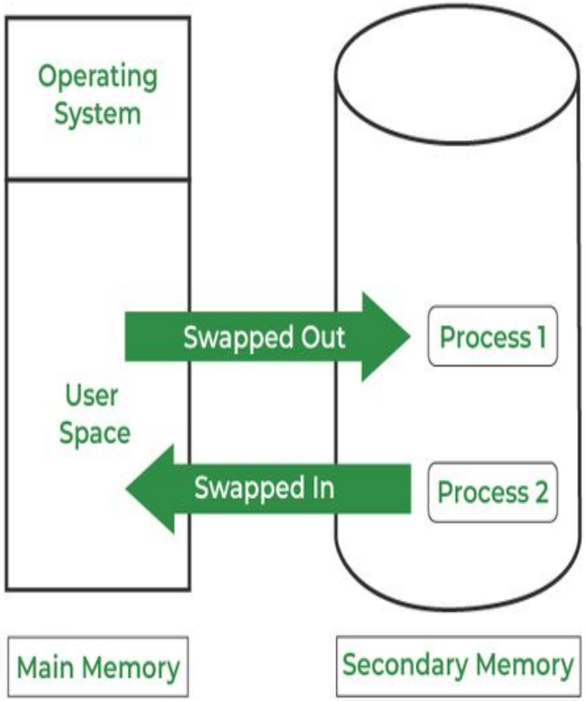
Memory Management

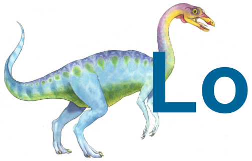
- The term memory can be defined as a collection of data in a specific format.
  - It is used to store instructions and process data.
  - The memory comprises a large array or group of words or bytes, each with its own location.
  - The primary purpose of a computer system is to execute programs.
  - These programs, along with the information they access, should be in the main memory during execution. The CPU fetches instructions from memory according to the value of the program counter.
- In a multiprogramming computer, the Operating System resides in a part of memory, and the rest is used by multiple processes. The task of subdividing the memory among different processes is called Memory Management. Memory management is a method in the operating system to manage operations between main memory and disk during process execution. The main aim of memory management is to achieve efficient utilization of memory.

Memory Management

Why Memory Management is Required?

- Allocate and de-allocate memory before and after process execution.
- To keep track of used memory space by processes.
- To minimize fragmentation issues.
- To proper utilization of main memory.
- To maintain data integrity while executing of process.





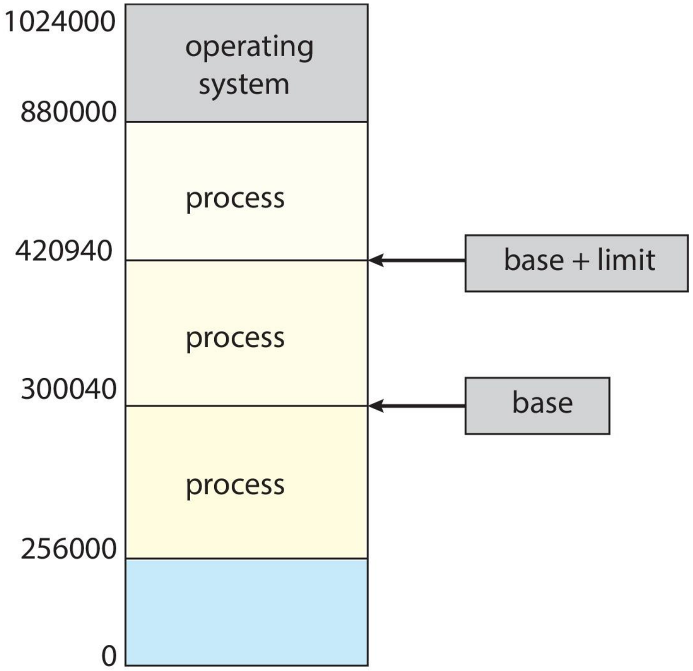
Logical and Physical Address Space

- **Logical Address Space:**
  - An address generated by the CPU is known as a “Logical Address”.
  - It is also known as a Virtual address.
  - Logical address space can be defined as the size of the process.
  - A logical address can be changed.

- **Physical Address Space:**
  - An address seen by the memory unit (i.e the one loaded into the memory address register of the memory) is commonly known as a “Physical Address”.
  - A Physical address is also known as a Real address.
  - The physical address always remains constant.

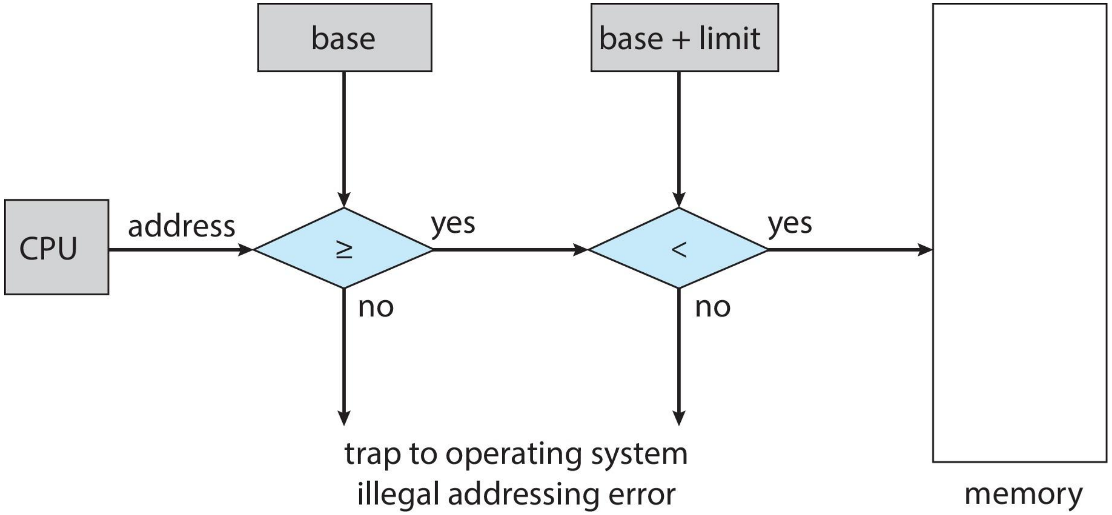
Protection

- Need to ensure that a process can access only those addresses in its address space.
- We can provide this protection by using a pair of **base** and **limit registers** define the logical address space of a process



Hardware Address Protection

- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user



- the instructions to loading the base and limit registers are privileged

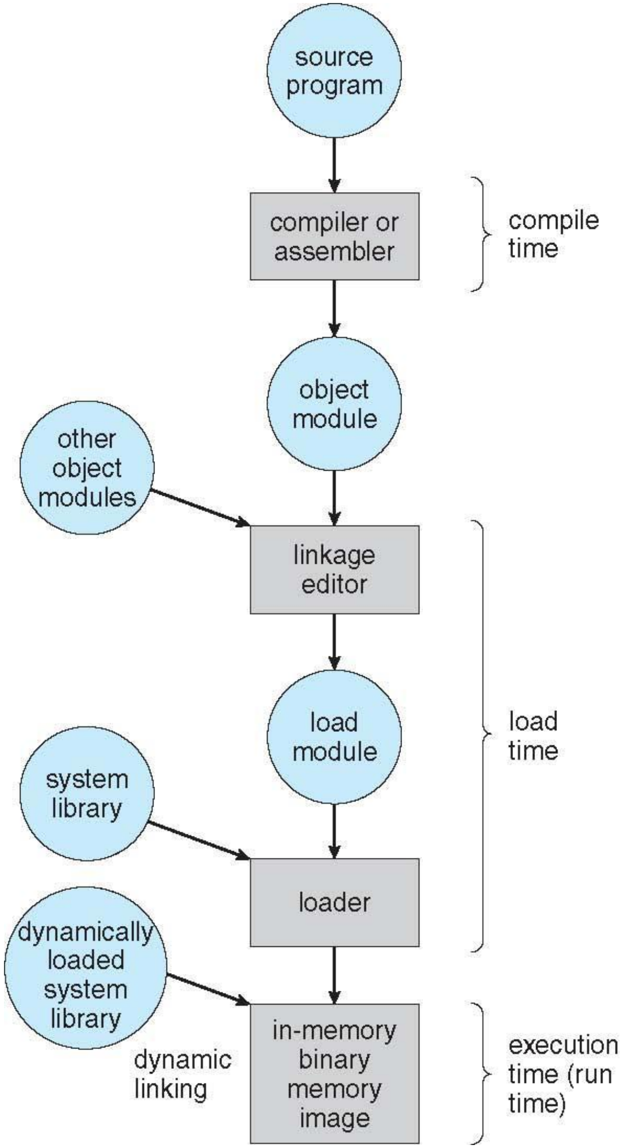
Address Binding

- Programs on disk, ready to be brought into memory to execute form an **input queue**
  - Without support, must be loaded into address 0000
- In brief, address binding means mapping computer instructions and data to locations in RAM.
- High-level programs are loaded into the memory for their execution in the CPU. The CPU transforms programs into processes and generates logical addresses to communicate with the main memory.
- 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
  - When the source code is compiled into machine code, the addresses are typically represented as relocatable addresses.
  - i.e., "14 bytes from beginning of this module"
  - **Linker or loader** Before execution, the compiled code needs to be linked and loaded into memory. The linker or loader binds these relocatable addresses to absolute addresses in memory 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(Absolute code refers to machine code or executable code where the memory addresses of instructions and data are fixed and known at compile time.)**can be generated; must recompile code if starting location changes.
  - **Load time**: Must generate **relocatable code(it includes information about the program's memory requirements and dependencies, allowing it to be loaded into different memory locations dynamically.)** 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)

Multistep Processing of a User Program

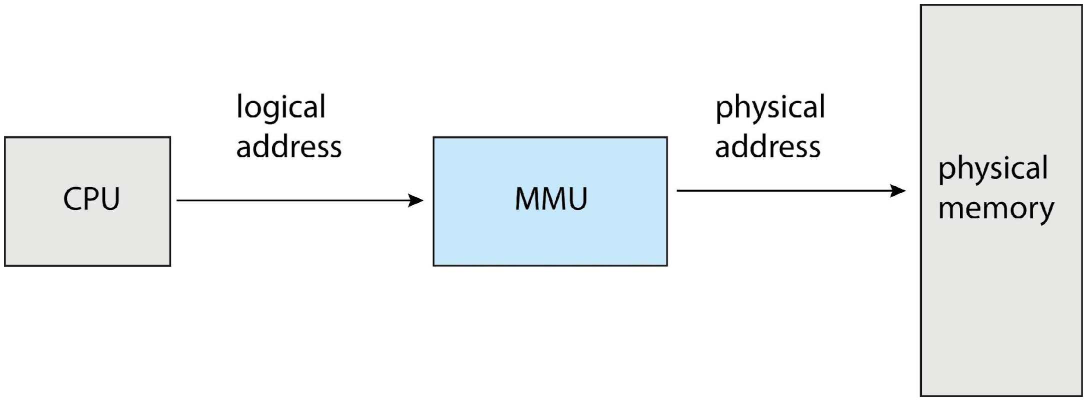


Logical vs. Physical Address Space

- 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 and physical addresses are **the same** in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses **differ i**n 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)

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



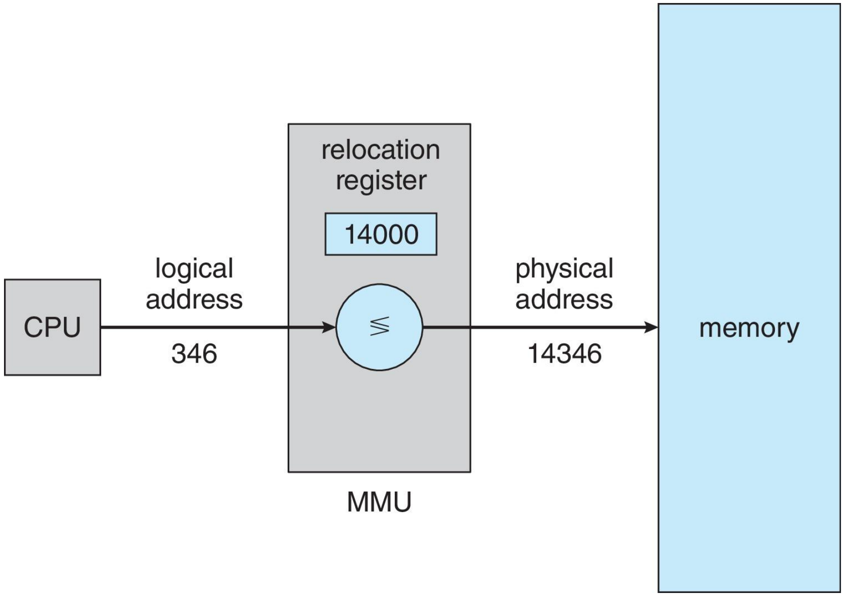
- Many methods possible, covered in the rest of this chapter

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 it is sent to memory
- 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 address bound to physical addresses

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 it is sent to memory



Static Loading: Static Loading is basically loading the entire program into a fixed address. It requires more memory space.

Dynamic Loading: The entire program and all data of a process must be in physical memory for the process to execute In dynamic loading, a process is not loaded until it is called. All processes are residing on disk . One of the advantages of dynamic loading is that the unused process is never loaded. This loading is useful when a large amount of code is needed to handle it efficiently.

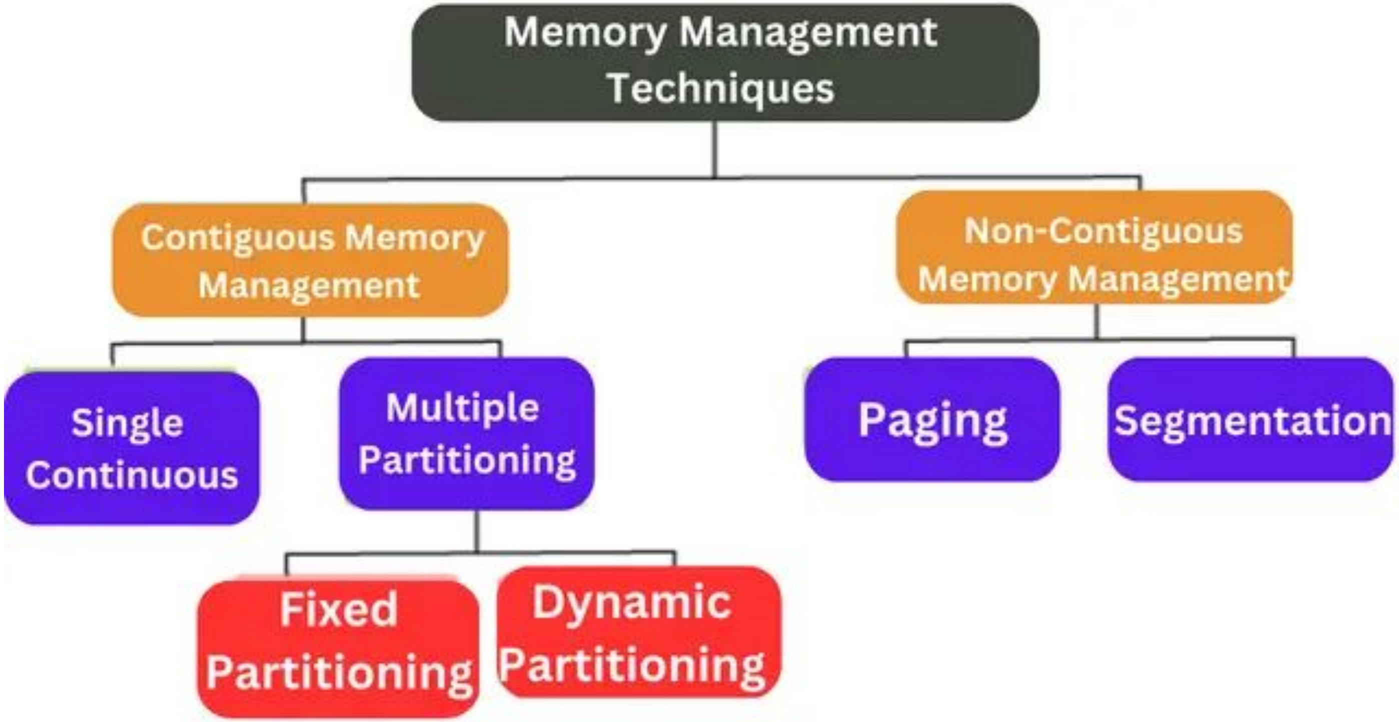
Dynamic Loading

- 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

Memory Management Technique



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**

Contiguous Allocation (Cont.)

- Contiguous memory allocation is a memory allocation strategy. As the name implies, we utilize this technique to assign contiguous blocks of memory to each task.

- whenever a process asks to access the main memory, we allocate a continuous segment from the empty region to the process based on its size. In this technique, memory is allotted in a continuous way to the processes.

- Contiguous Memory Management has two types:

  - Fixed(or Static) Partition
  - Variable(or Dynamic) Partitioning

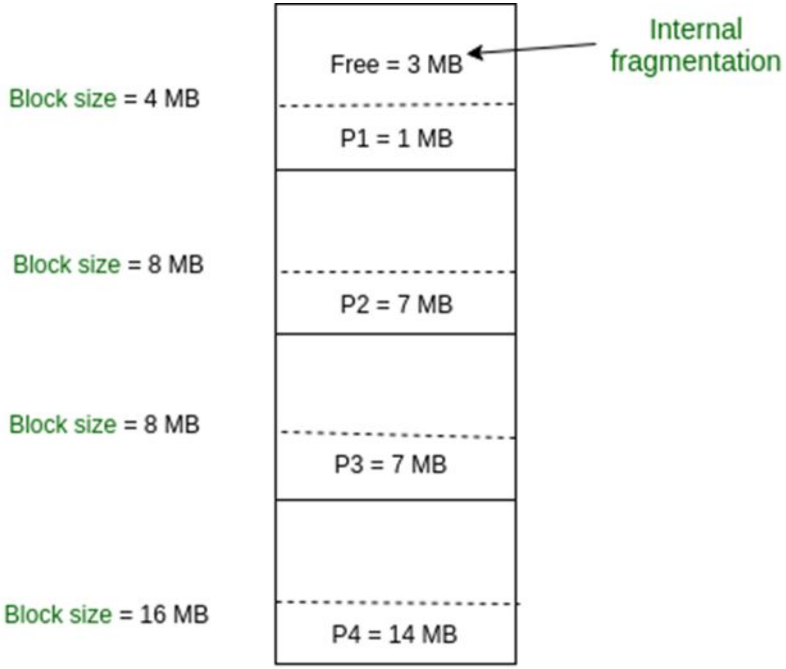
Fixed Partitioning

- This is the oldest and simplest technique used to put more than one process in the main memory.
- In this partitioning, the number of partitions (non-overlapping) in RAM is fixed but the size of each partition may or may not be the same.
- As it is a contiguous allocation, hence no spanning is allowed. Here partitions are made before execution or during system configure.
-
- **Disadvantages**
    - Internal Fragmentation:
    - External Fragmentation:
    - Limit process size:
    - Limitation on Degree of Multiprogramming:

Fixed Partitioning

- As illustrated in above figure, first process is only consuming 1MB out of 4MB in the main memory.
- Hence, Internal Fragmentation in first block is (4-1) = 3MB.
- Sum of Internal Fragmentation in every block = (4-1)+(8-7)+(8-7)+(16-14)= 3+1+1+2 = 7MB.

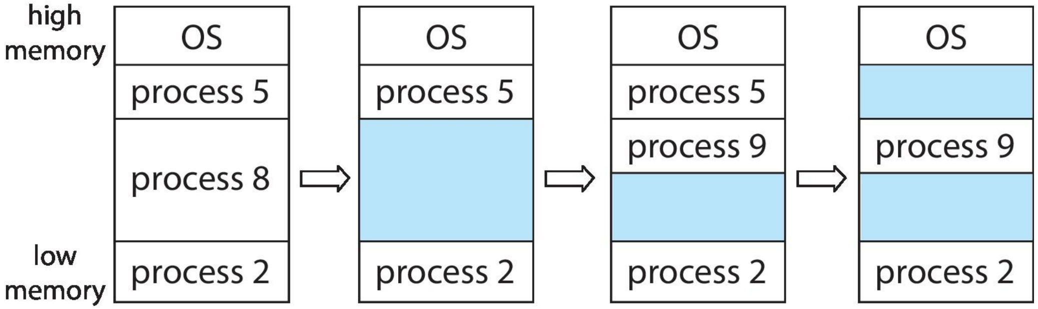
- Suppose process P5 of size 7MB comes. But this process cannot be accommodated in spite of available free space because of contiguous allocation (as spanning is not allowed). Hence, 7MB becomes part of External Fragmentation.



Fixed size partition

Variable Partition

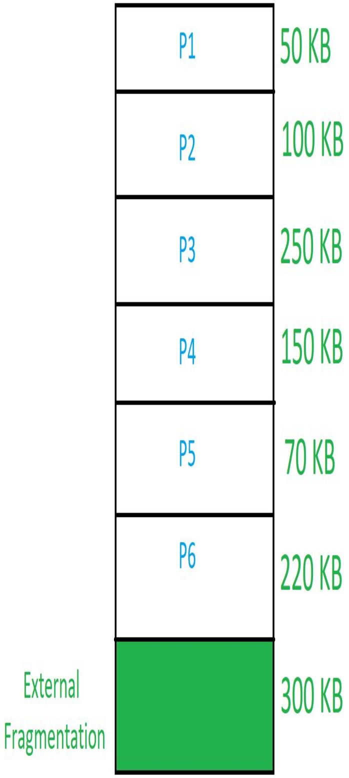
- Multiple-partition allocation
  - Degree of multiprogramming limited by number of partitions
  - **Variable-partition** sizes for efficiency (sized to a given process' needs)
  - **Hole** – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, 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)



Variable Partition

- In the variable partition scheme, initially memory will be single continuous free block. Whenever the request by the process arrives, accordingly partition will be made in the memory. If the smaller processes keep on coming then the larger partitions will be made into smaller partitions.

- In variable partition schema initially, the memory will be full contiguous free block
- Memory divided into partitions according to the process size where process size will vary.
- One partition is allocated to each active partition.



Variable Partition

Solution of External Fragmentation

1. Compaction: Moving all the processes toward the top or towards the bottom to make free available memory in a single continuous place is called compaction. Compaction is undesirable to implement because it interrupts all the running processes in the memory.

2. Non-contiguous memory allocation

Advantages

1. Portion size = process size
2. There is no internal fragmentation (which is the drawback of fixed partition schema).
3. Degree of multiprogramming varies and is directly proportional to a number of processes.

Disadvantage

4. External fragmentation is still there.

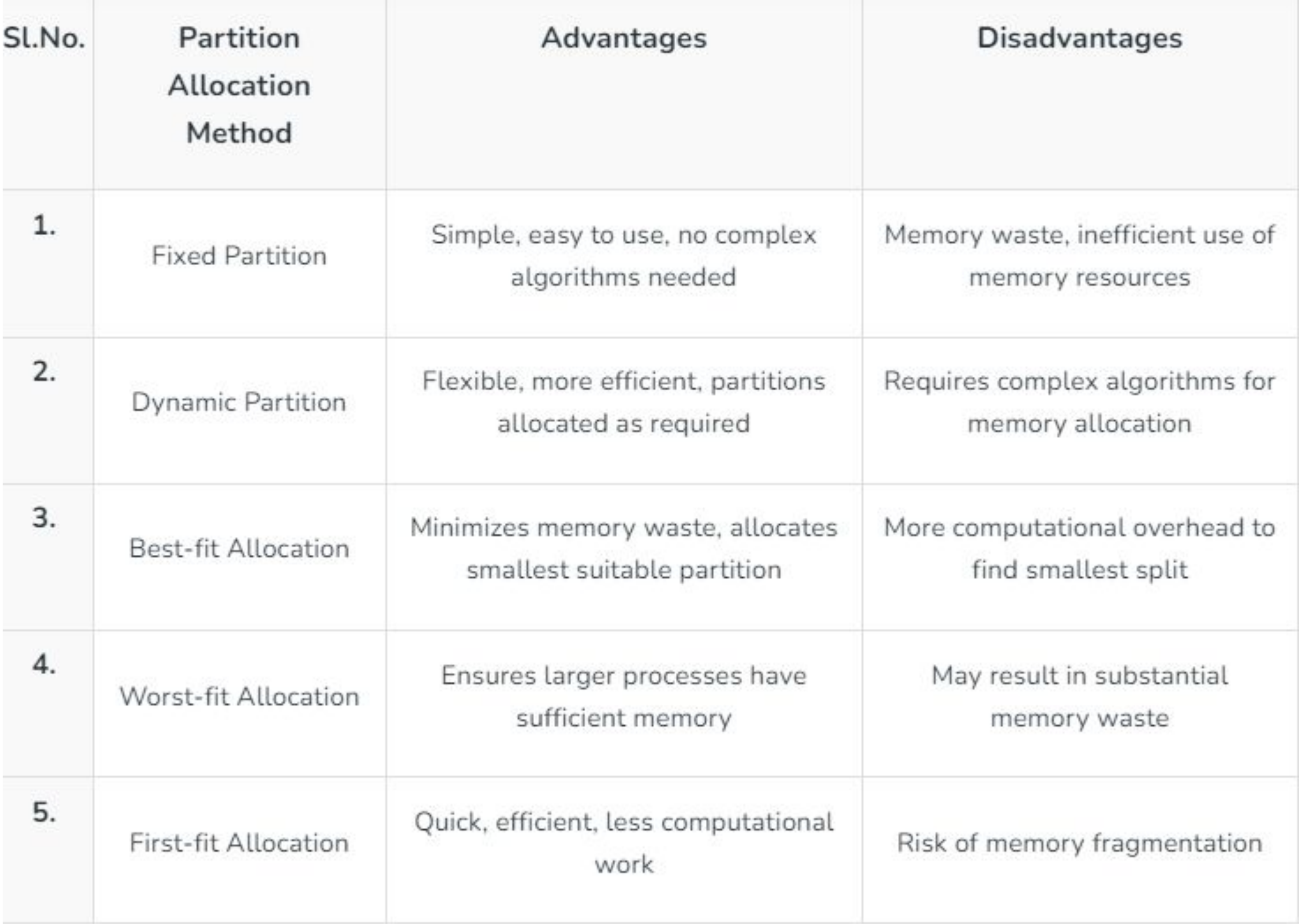
Dynamic Storage-Allocation Problem

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

- **First-fit**: Allocate the ***first*** hole that is big enough
- **Next-Fit**: Next fit is similar to the first fit but it will search for the first sufficient partition from the last allocation point.
- **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

Difference

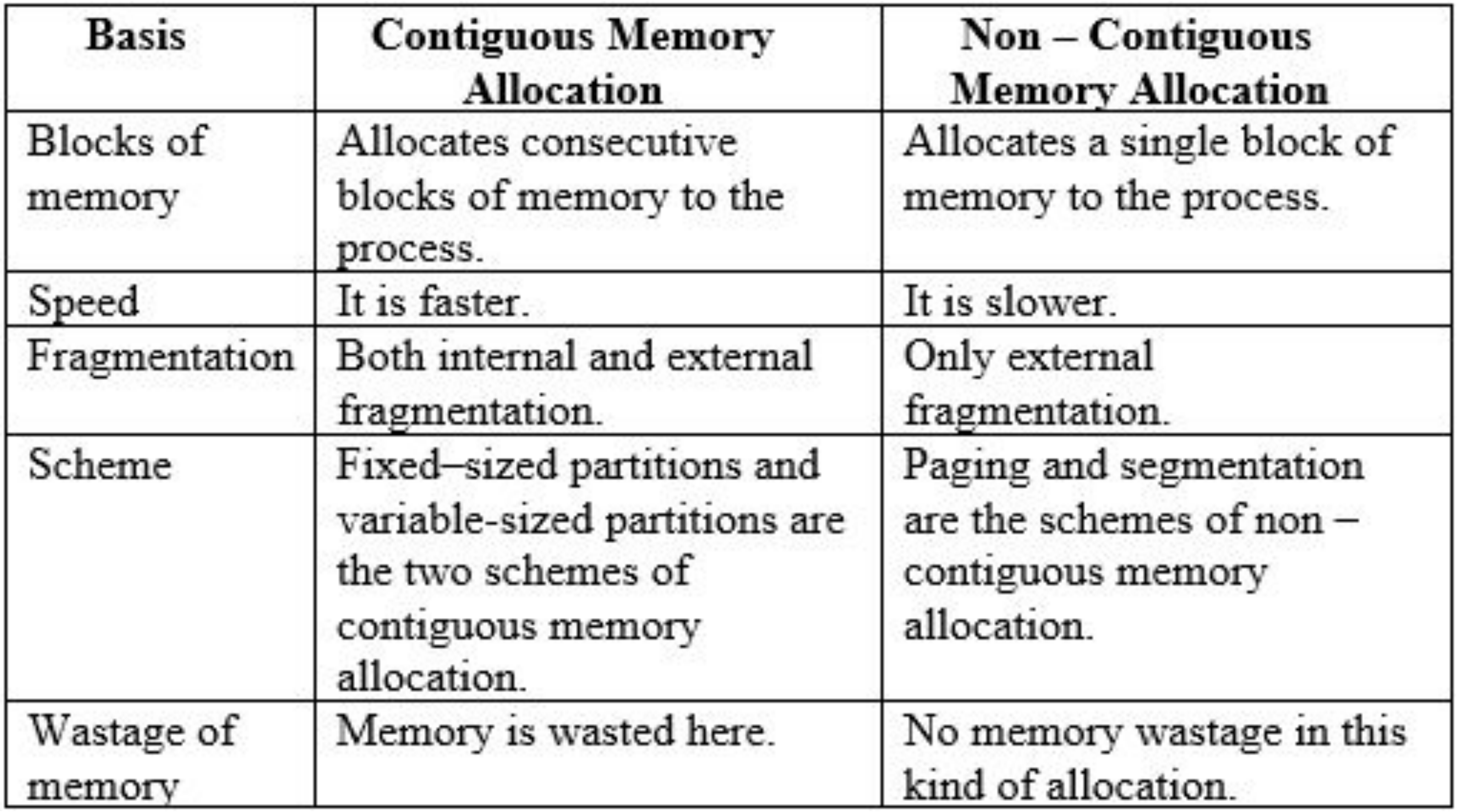


Sl.No.	Partition Allocation Method	Advantages	Disadvantages
1.	Fixed Partition	Simple, easy to use, no complex algorithms needed	Memory waste, inefficient use of memory resources
2.	Dynamic Partition	Flexible, more efficient, partitions allocated as required	Requires complex algorithms for memory allocation
3.	Best-fit Allocation	Minimizes memory waste, allocates smallest suitable partition	More computational overhead to find smallest split
4.	Worst-fit Allocation	Ensures larger processes have sufficient memory	May result in substantial memory waste
5.	First-fit Allocation	Quick, efficient, less computational work	Risk of memory fragmentation

Non-contiguous

- Non-contiguous allocation, also known as dynamic or linked allocation, is a memory allocation technique used in operating systems to allocate memory to processes that do not require a contiguous block of memory. In this technique, each process is allocated a series of non-contiguous blocks of memory that can be located anywhere in the physical memory.
- Fundamental approaches : There are two fundamental approaches to implementing noncontiguous memory allocation:
  - Paging
  - Segmentation

Difference



Basis	Contiguous Memory Allocation	Non – Contiguous Memory Allocation
Blocks of memory	Allocates consecutive blocks of memory to the process.	Allocates a single block of memory to the process.
Speed	It is faster.	It is slower.
Fragmentation	Both internal and external fragmentation.	Only external fragmentation.
Scheme	Fixed–sized partitions and variable-sized partitions are the two schemes of contiguous memory allocation.	Paging and segmentation are the schemes of non – contiguous memory allocation.
Wastage of memory	Memory is wasted here.	No memory wastage in this kind of allocation.

Fragmentation

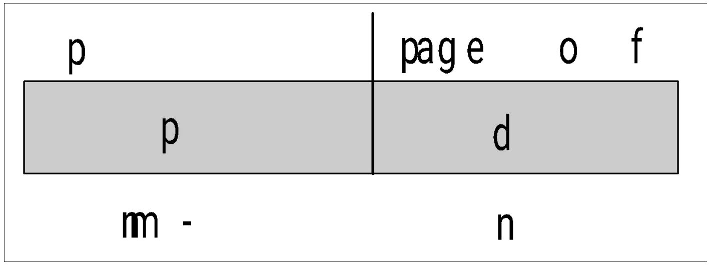
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals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

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

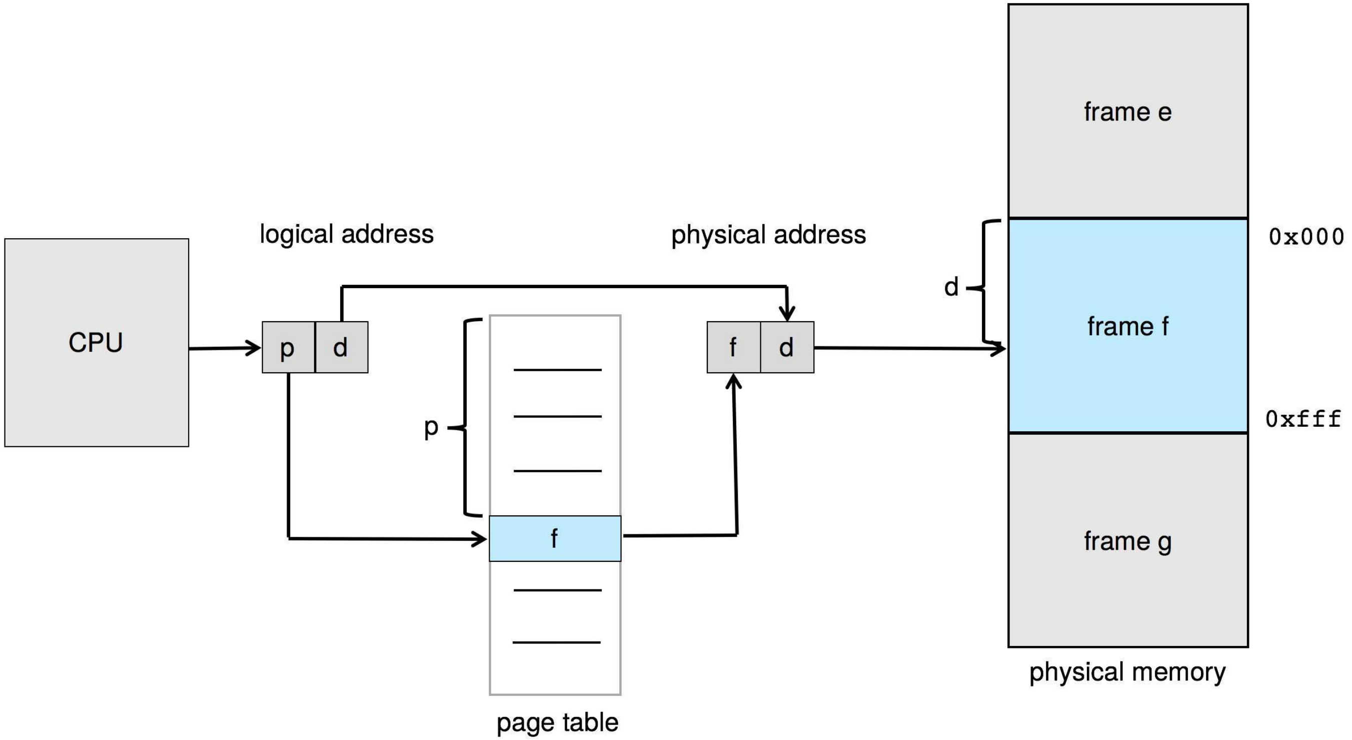
Address Translation Scheme

- 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

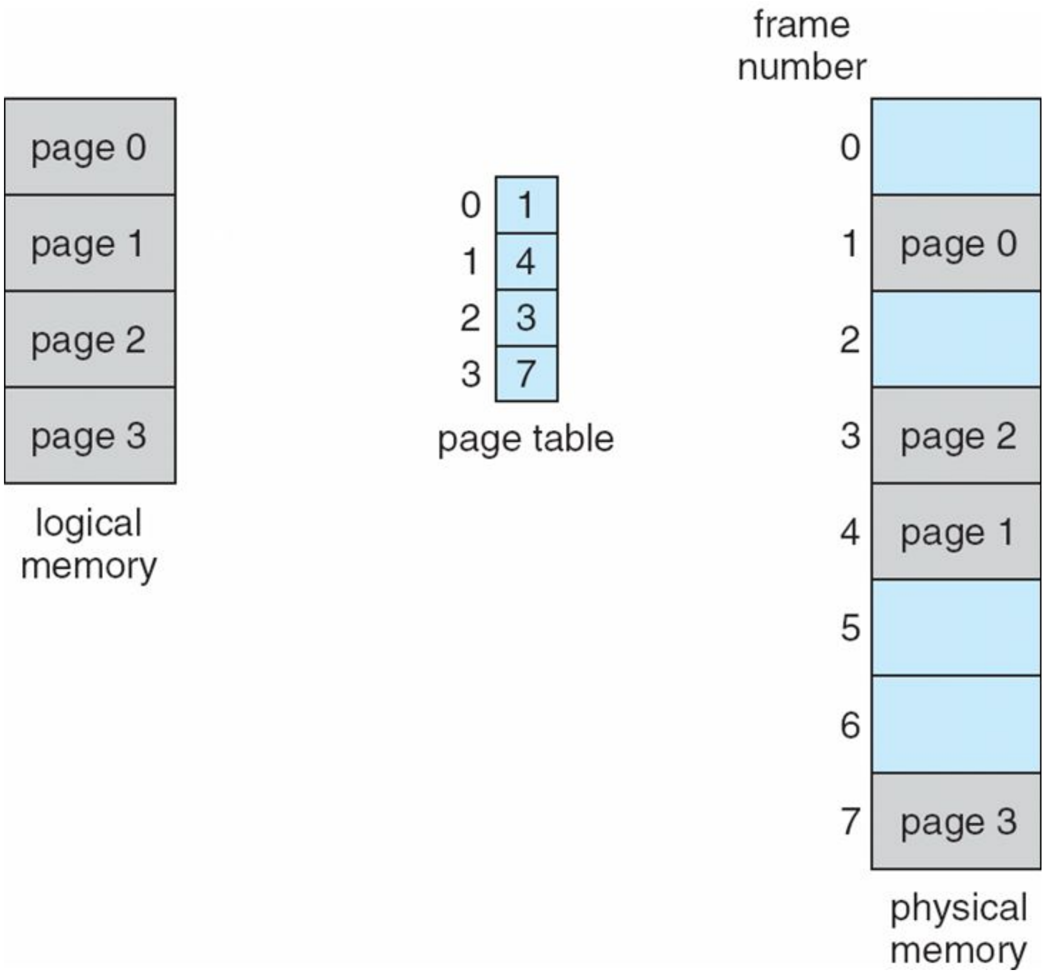


  - For given logical address space 2^m and page size 2^n

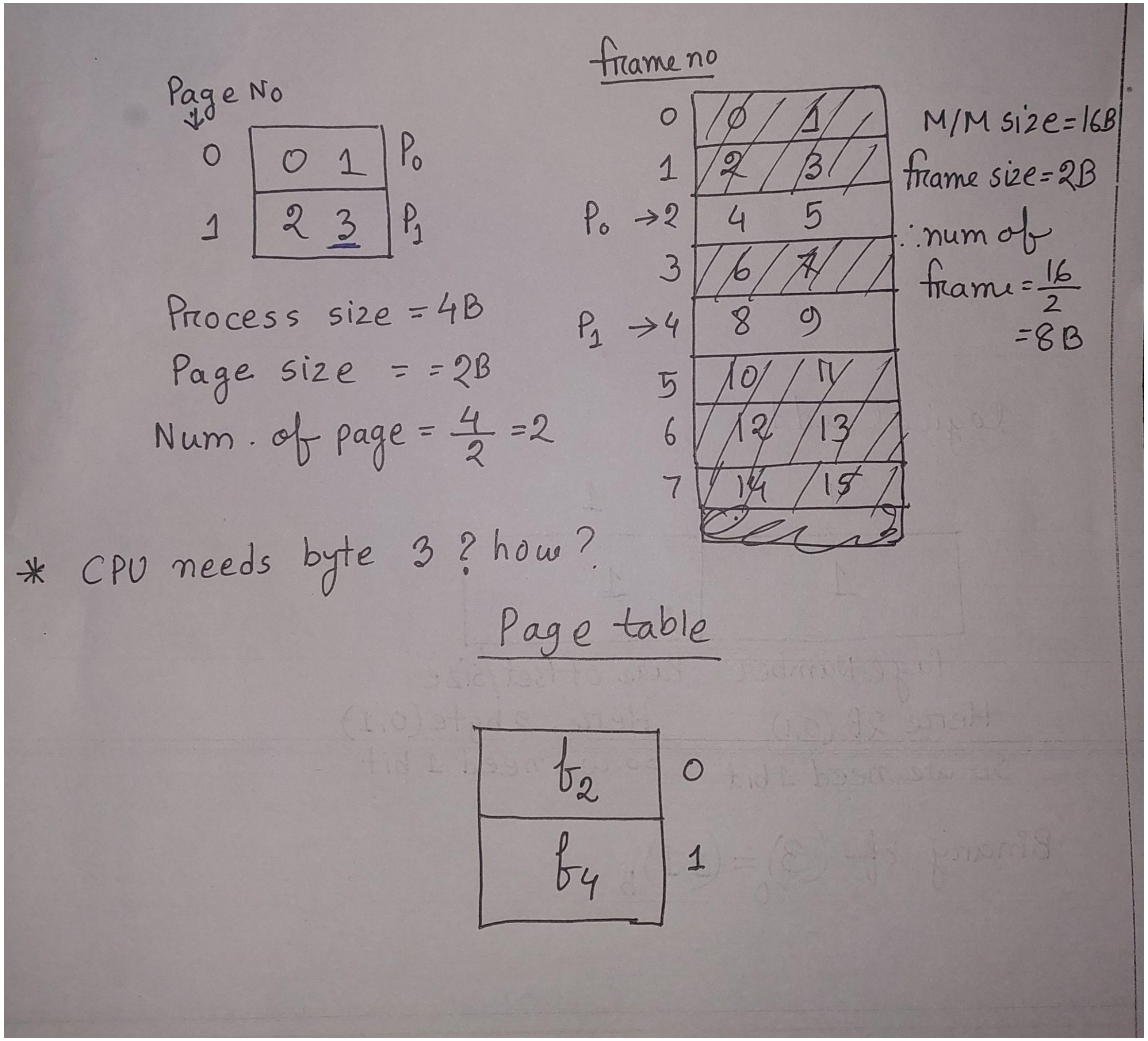
Paging Hardware



Paging Model of Logical and Physical Memory



Paging Example



Page No ↓

0	0	1	P_0
1	2	3	P_1

Process size = 4B

Page size = = 2B

Num. of page = $\frac{4}{2}$ = 2

frame no

	0	0	1
	1	2	3
P_0 →	2	4	5
	3	6	7
P_1 →	4	8	9
	5	10	11
	6	12	13
	7	14	15

M/M size = 16B

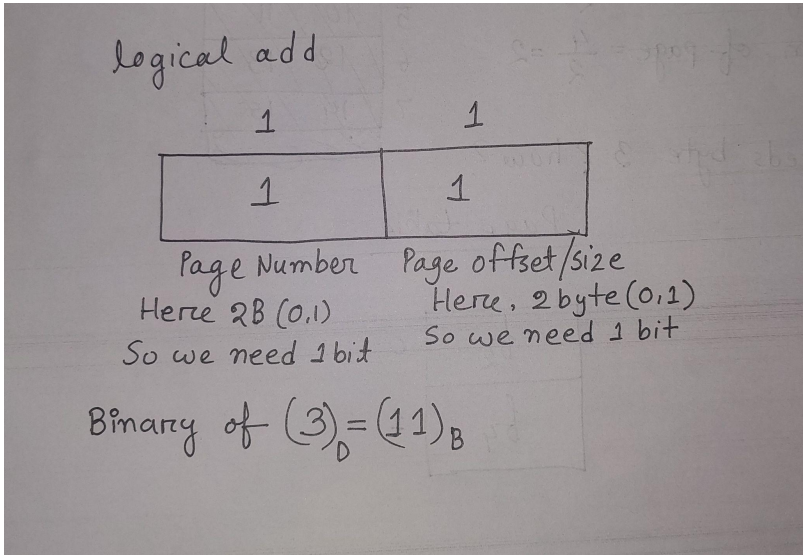
frame size = 2B

∴ num of frame = $\frac{16}{2}$ = 8B

* CPU needs byte 3 ? how ?

Page table

f_2	0
f_4	1



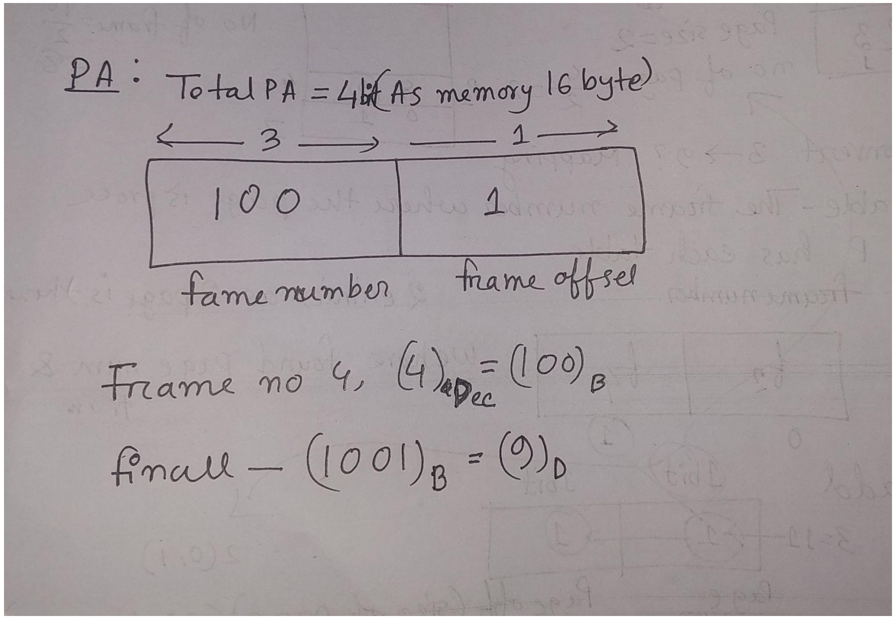
logical add

1	1
1	1
Page Number	Page offset/size

Here 2B (0,1)
So we need 1bit

Here, 2 byte(0,1)
So we need 1 bit

Binary of $(3)_D = (11)_B$



PA : Total PA = 4bit (As memory 16 byte)

← 3 →	— 1 →
100	1
fame number	frame offsel

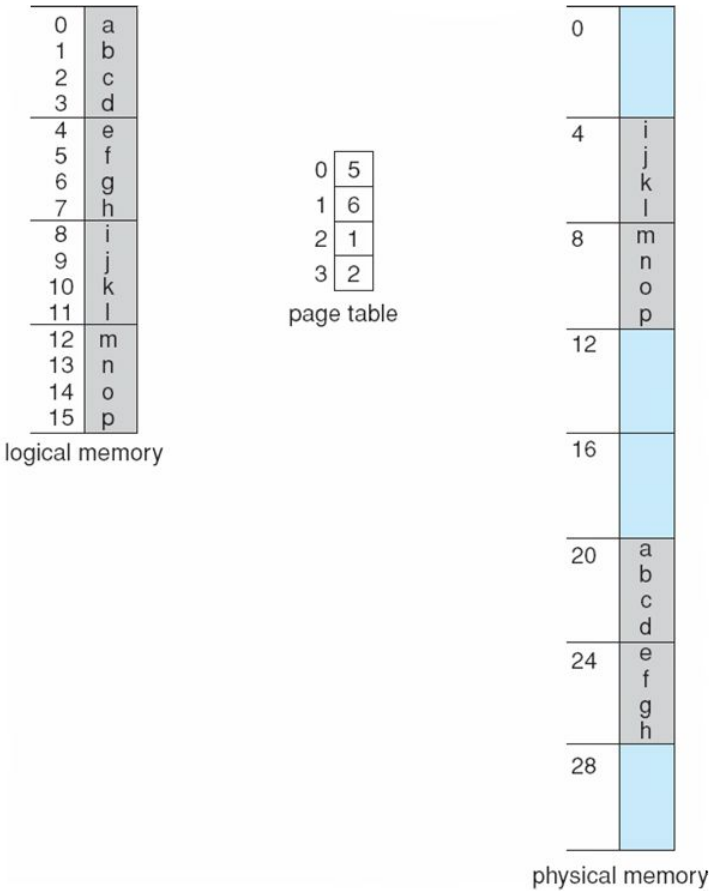
Frame no 4, $(4)_{Dec} = (100)_B$

finall — $(1001)_B = (9)_D$

https://youtu.be/6c-mOFZwP_8

Paging Example

- Logical address: n = 2 and m = 4. Using a page size of 4 bytes and a physical memory of 32 bytes (8 pages)



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

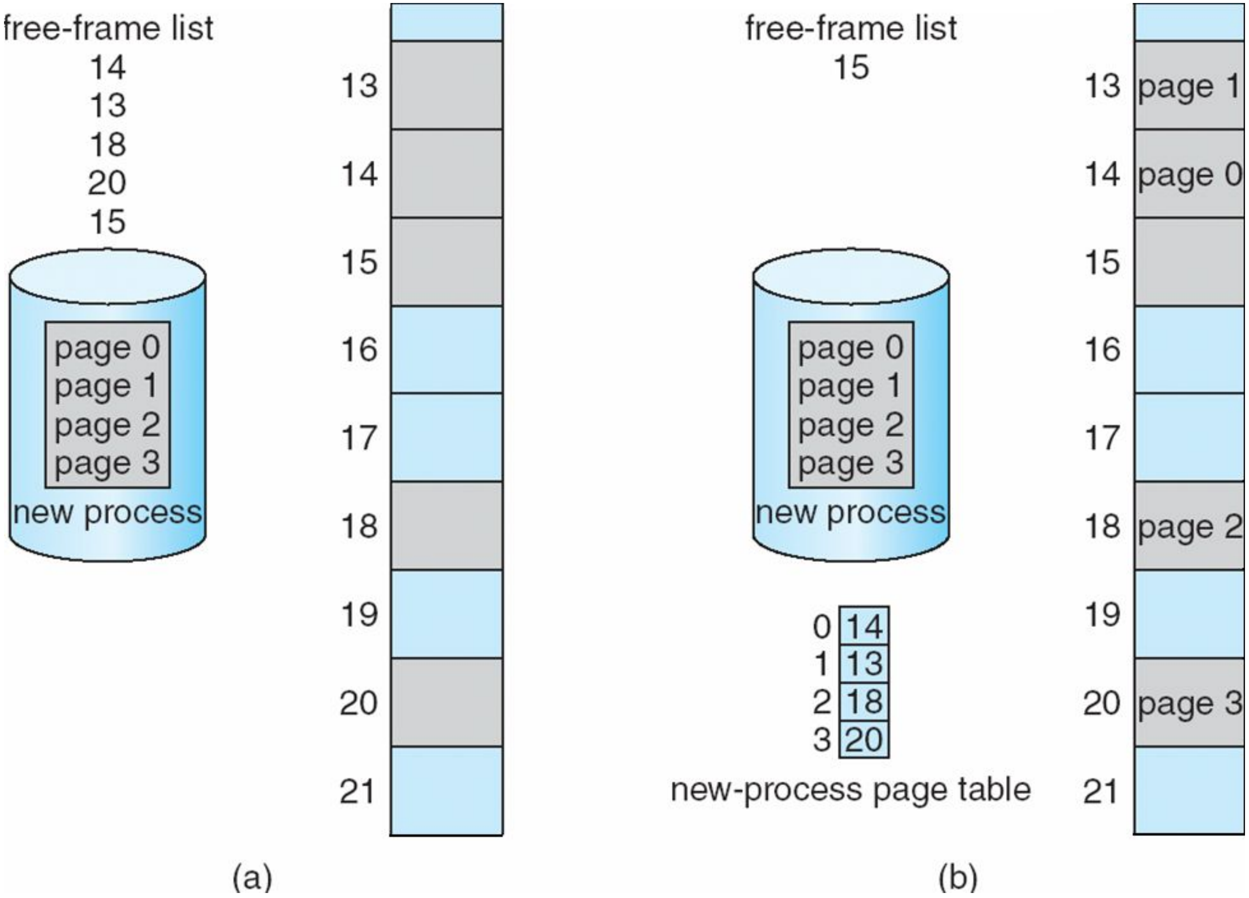
0	
4	i j k l
8	m n o p
12	
16	
20	a b c d
24	e f g h
28	

physical memory

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
  - Solaris supports two page sizes – 8 KB and 4 MB

Free Frames



Before allocation

After allocation

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 **translation look-aside buffers** (**TLBs**) (also called **associative memory**).

Translation Look-Aside Buffer

- Some TLBs store **address-space identifiers** (**ASIDs**) in each TLB entry – uniquely identifies each process to provide address-space protection for that process
  - Otherwise need to flush at every context switch
- 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
  - Some entries can be **wired down** for permanent fast access

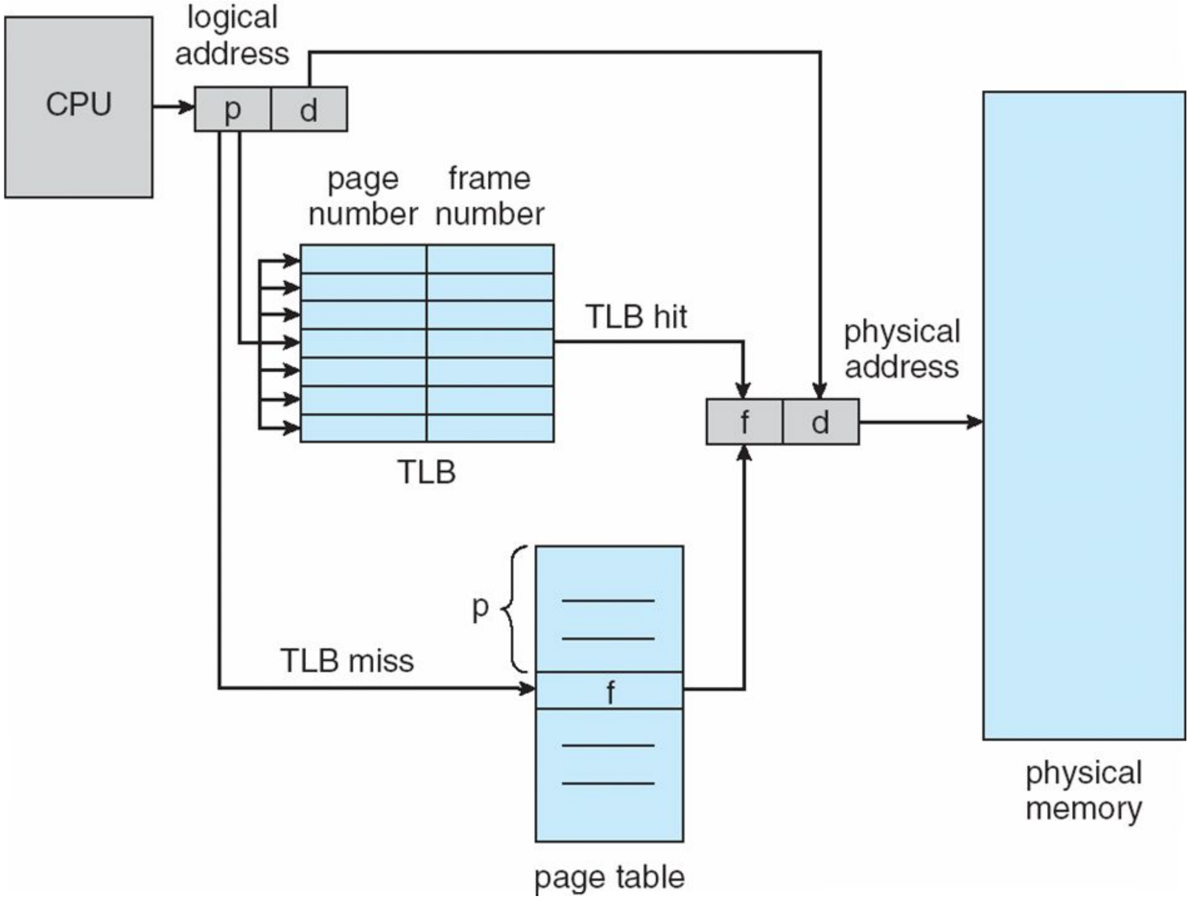
Hardware

- Associative memory – parallel search

Page #	Frame #

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

Paging Hardware With TLB



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**)

  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.

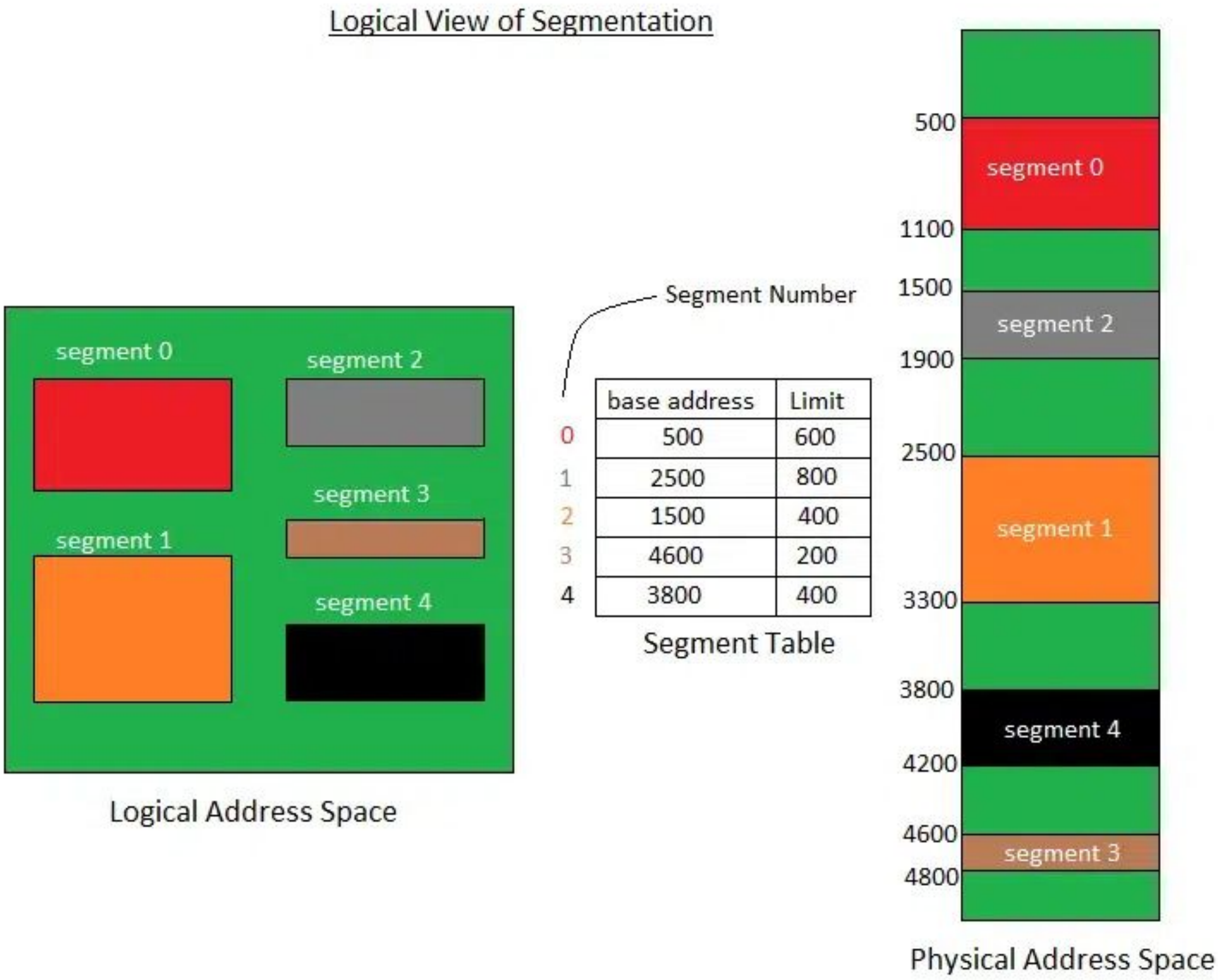
Segmentation

A process is divided into Segments. The chunks that a program is divided into which are not necessarily all of the exact sizes are called segments. Segmentation gives the user's view of the process which paging does not provide. Here the user's view is mapped to physical memory.

Base Address: It contains the starting physical address where the segments reside in memory.

Segment Limit: Also known as segment offset. It specifies the length of the segment.

Segmentation

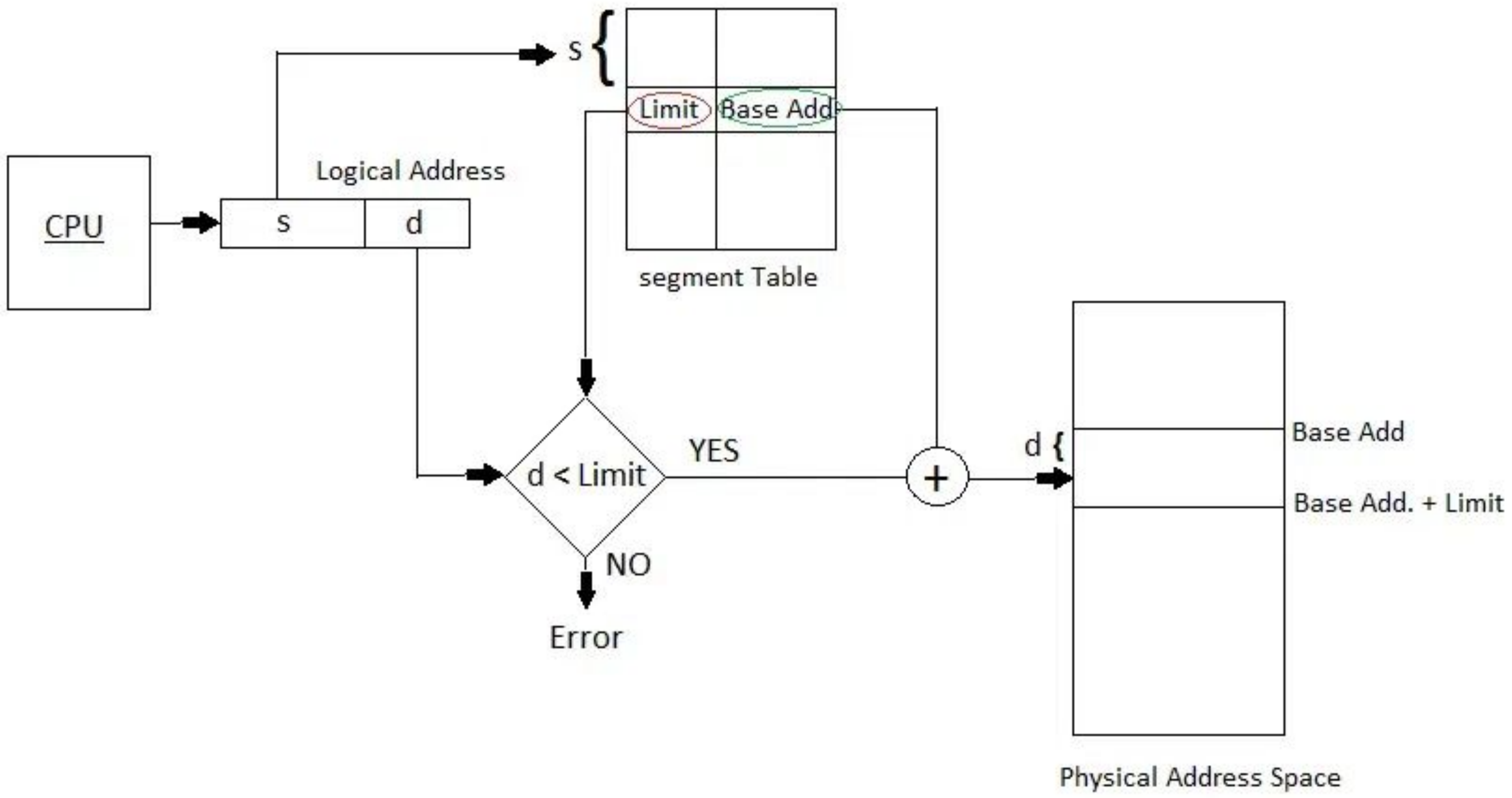


Logical View of Segmentation

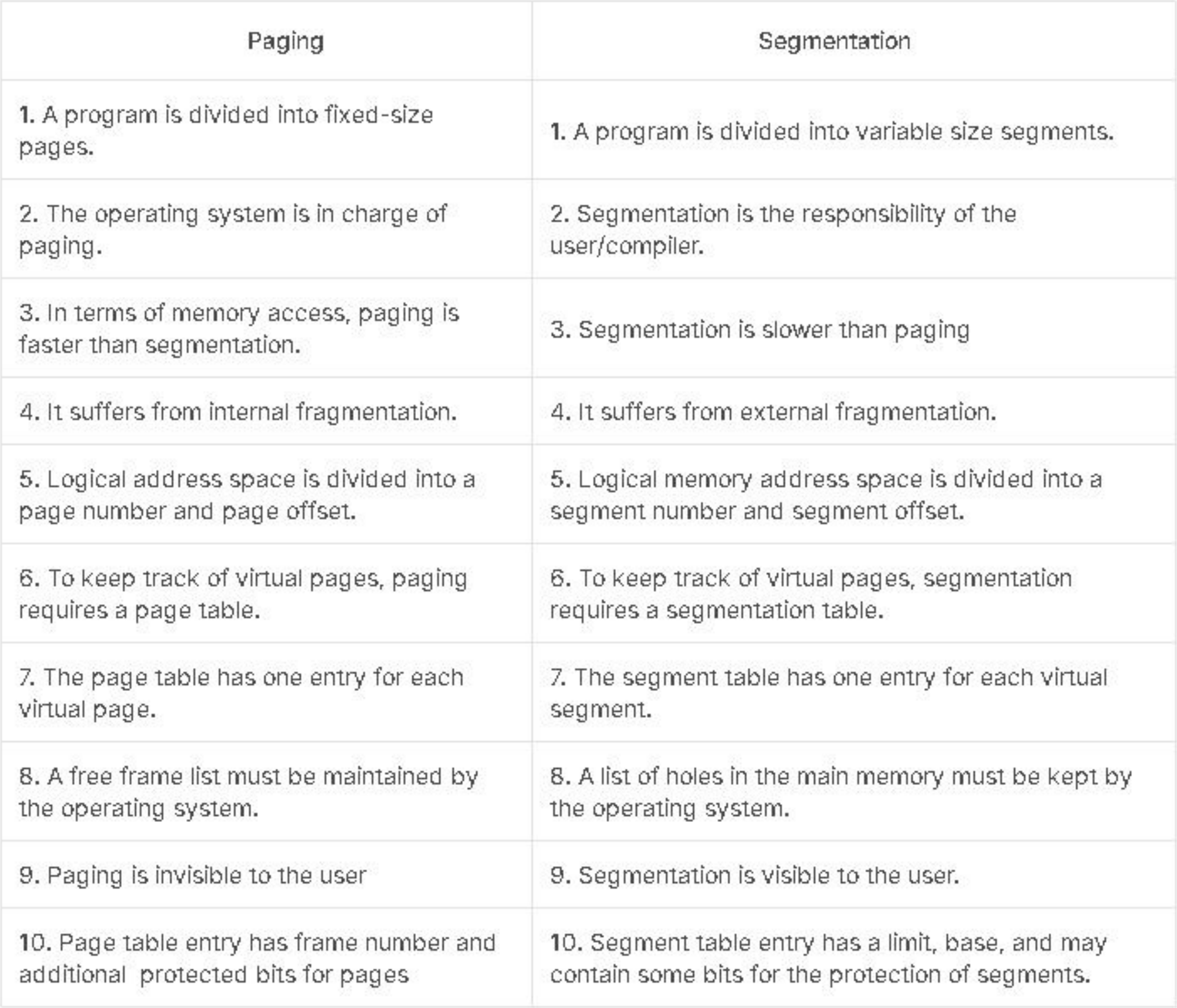
	base address	Limit
0	500	600
1	2500	800
2	1500	400
3	4600	200
4	3800	400

Segment Table

Segmentation



Difference between Paging and Segmentation



Paging	Segmentation
1. A program is divided into fixed-size pages.	1. A program is divided into variable size segments.
2. The operating system is in charge of paging.	2. Segmentation is the responsibility of the user/compiler.
3. In terms of memory access, paging is faster than segmentation.	3. Segmentation is slower than paging
4. It suffers from internal fragmentation.	4. It suffers from external fragmentation.
5. Logical address space is divided into a page number and page offset.	5. Logical memory address space is divided into a segment number and segment offset.
6. To keep track of virtual pages, paging requires a page table.	6. To keep track of virtual pages, segmentation requires a segmentation table.
7. The page table has one entry for each virtual page.	7. The segment table has one entry for each virtual segment.
8. A free frame list must be maintained by the operating system.	8. A list of holes in the main memory must be kept by the operating system.
9. Paging is invisible to the user	9. Segmentation is visible to the user.
10. Page table entry has frame number and additional protected bits for pages	10. Segment table entry has a limit, base, and may contain some bits for the protection of segments.

References

https://www.gatevidyalay.com/segmentation-in-os-practice-problems/

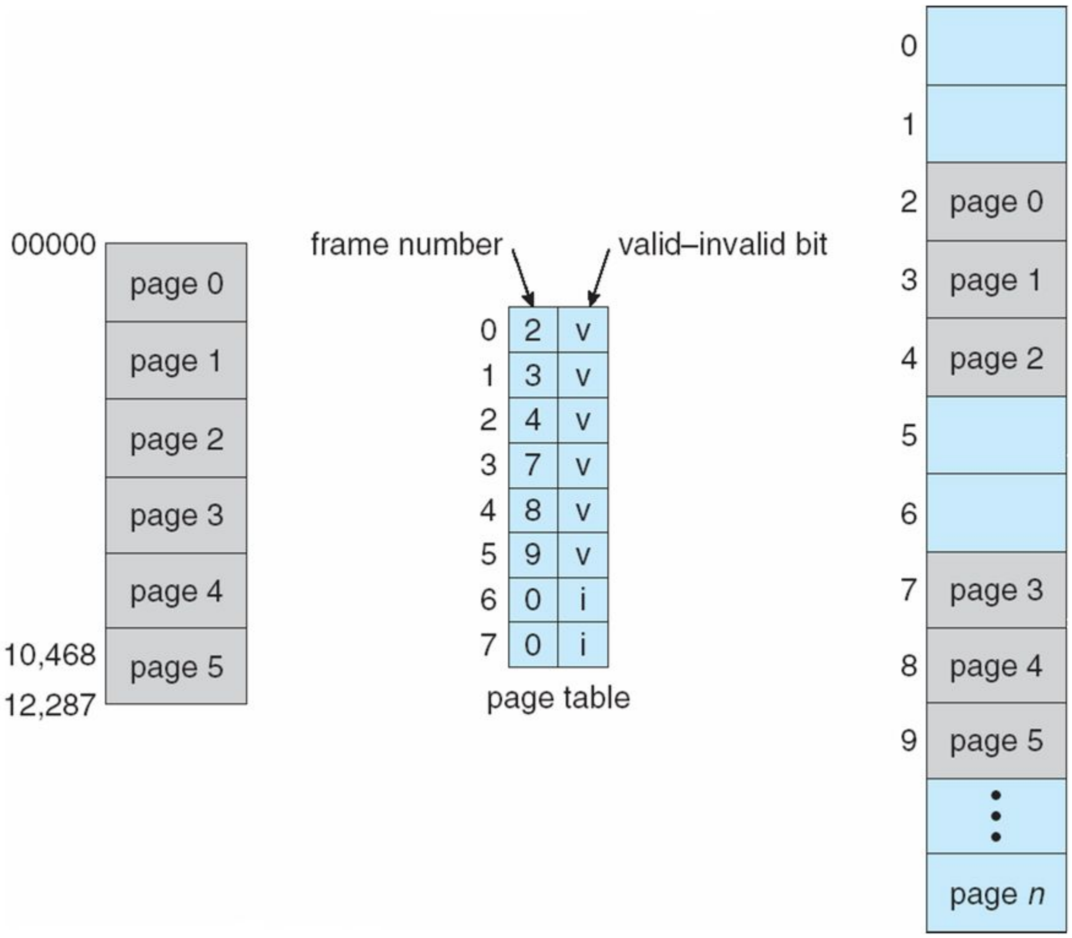
https://www.researchgate.net/profile/Qasim-Hussein/publication/279060822_Operating_system_questions_with_their_answers_Memory_management_Virtual_memory_Processes_synchronization_Part_two/links/5589610308ae273b2876b4c6/Operating-system-questions-with-their-answers-Memory-management-Virtual-memory-Processes-synchronization-Part-two.pdf

https://www.gatevidyalay.com/paging-formulas-practice-problems/

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

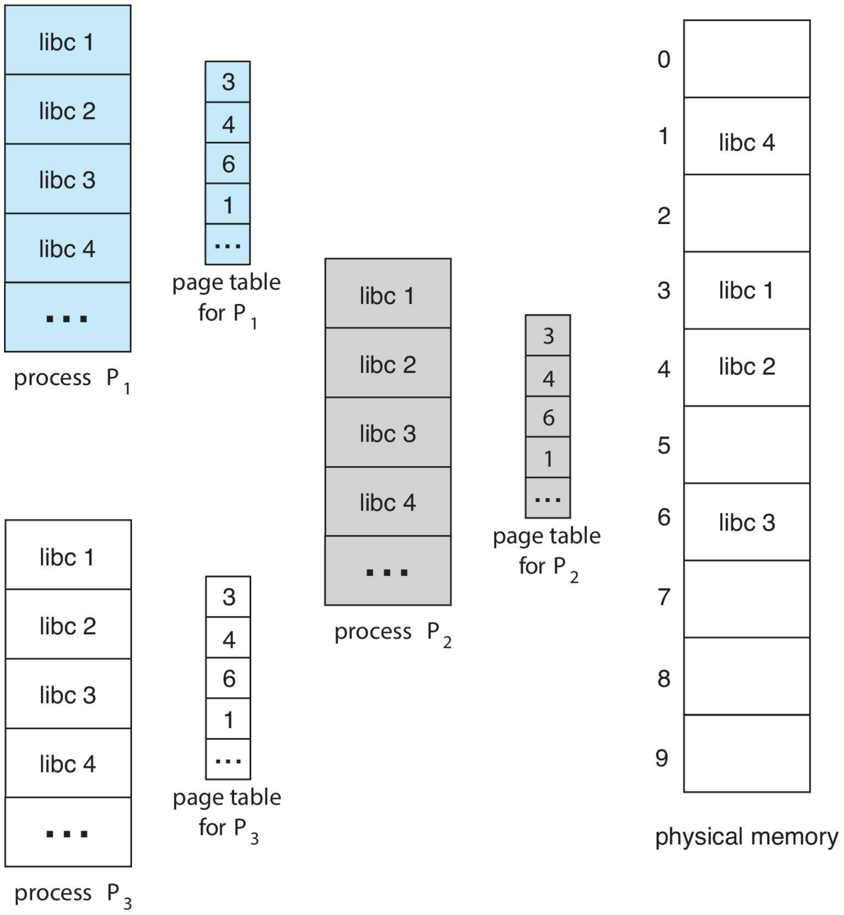


	frame number	valid–invalid bit
0	2	v
1	3	v
2	4	v
3	7	v
4	8	v
5	9	v
6	0	i
7	0	i

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

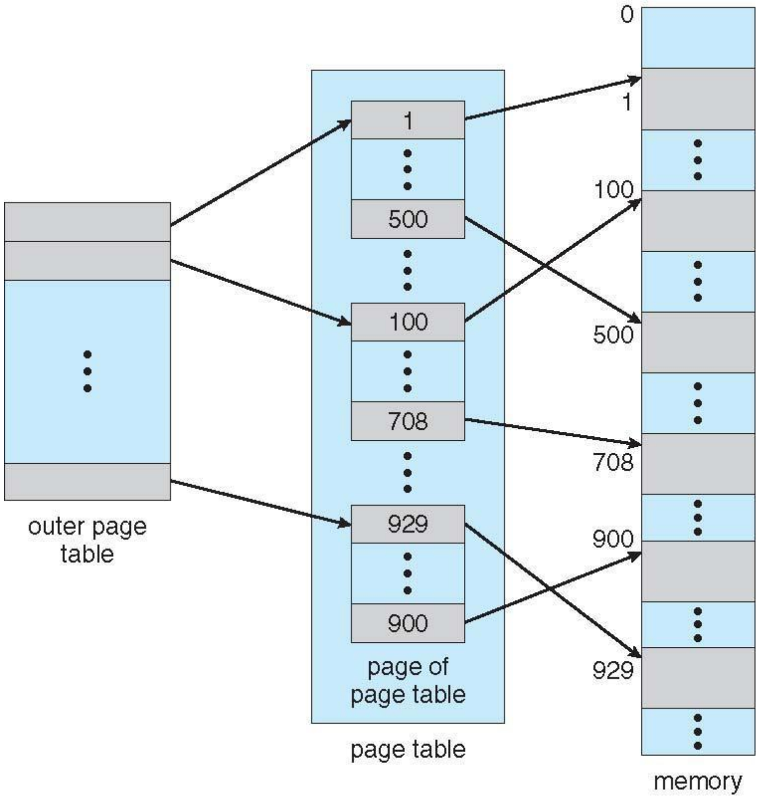


Structure of the Page Table

- 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 4 MB of physical address space for the page table alone
    - 4 Don’t want to allocate that contiguously in main memory
  - One simple solution is to divide the page table into smaller units
    - 4 Hierarchical Paging
    - 4 Hashed Page Tables
    - 4 Inverted Page Tables

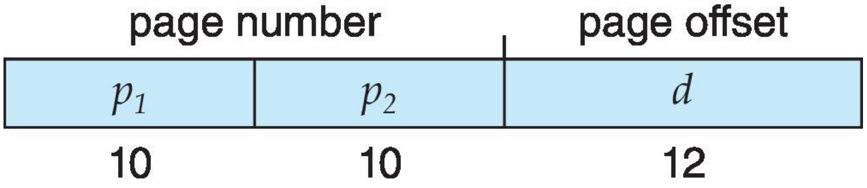
Hierarchical Page Tables

- 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



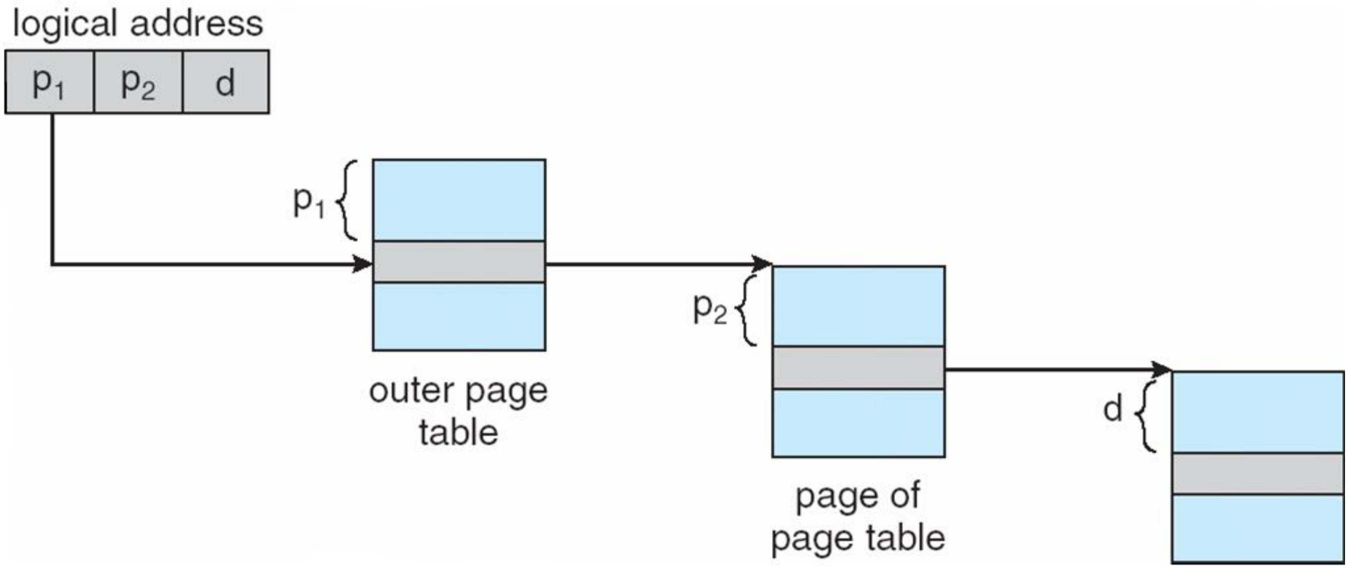
Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a l



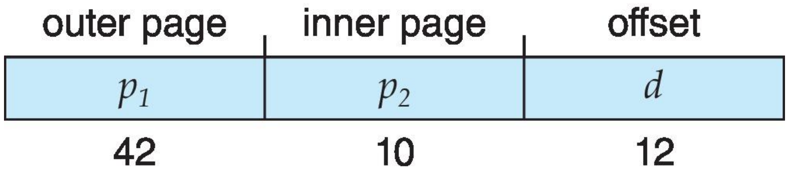
- where p_1 is an index into the outer page table, and p_2 is the displacement within the page of the inner page table
- Known as **forward-mapped page table**

Address-Translation Scheme



64-bit Logical Address Space

- Even two-level paging scheme not sufficient
- If page size is 4 KB (2^{12})
  - Then page table has 2^{52} entries
  - If two level scheme, inner page tables could be 2^{10} 4-byte entries
  - Address would look like



  - Outer page table has 2^{42} entries or 2^{44} bytes
  - One solution is to add a 2nd outer page table
  - But in the following example the 2nd outer page table is still 2^{34} bytes in size
    - 4 And possibly 4 memory access to get to one physical memory location

End of Chapter 9