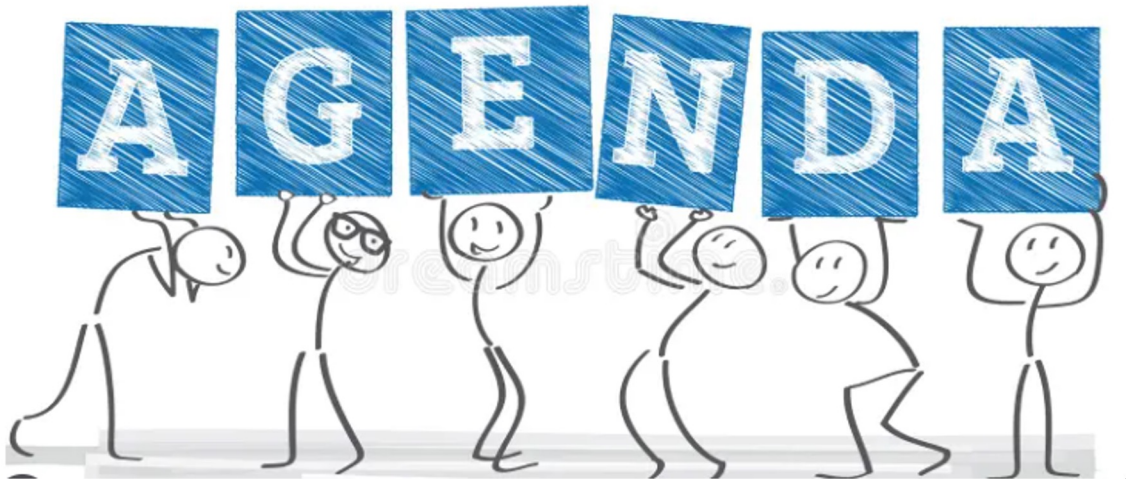
Operating Systems

Lecture 6.2

Non-Contiguous Memory Allocation Schemes - I

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda



- Overview of Non-Contiguous Memory Allocation
- Segmentation
  - What is Segmentation?
  - How segmentation work?
  - LA to PA translation in Segmentation
  - Sharing of Segments among Processes
  - Example Problems
- Paging
  - What is Paging
  - How paging work?
  - LA to PA translation in Paging
  - Sharing of Pages among Processes
  - Example Problems
- Where the Page Table is Kept?
  - CPU registers
  - Cache / TLB
  - Memory
  - LA to PA translation
  - Example Problems
- Structure of Page Table and Address translation in
  - Hierarchical PT
  - Inverted PT
  - Hashed PT
  - Example Problems

Overview of Non-contiguous Memory Allocation Techniques

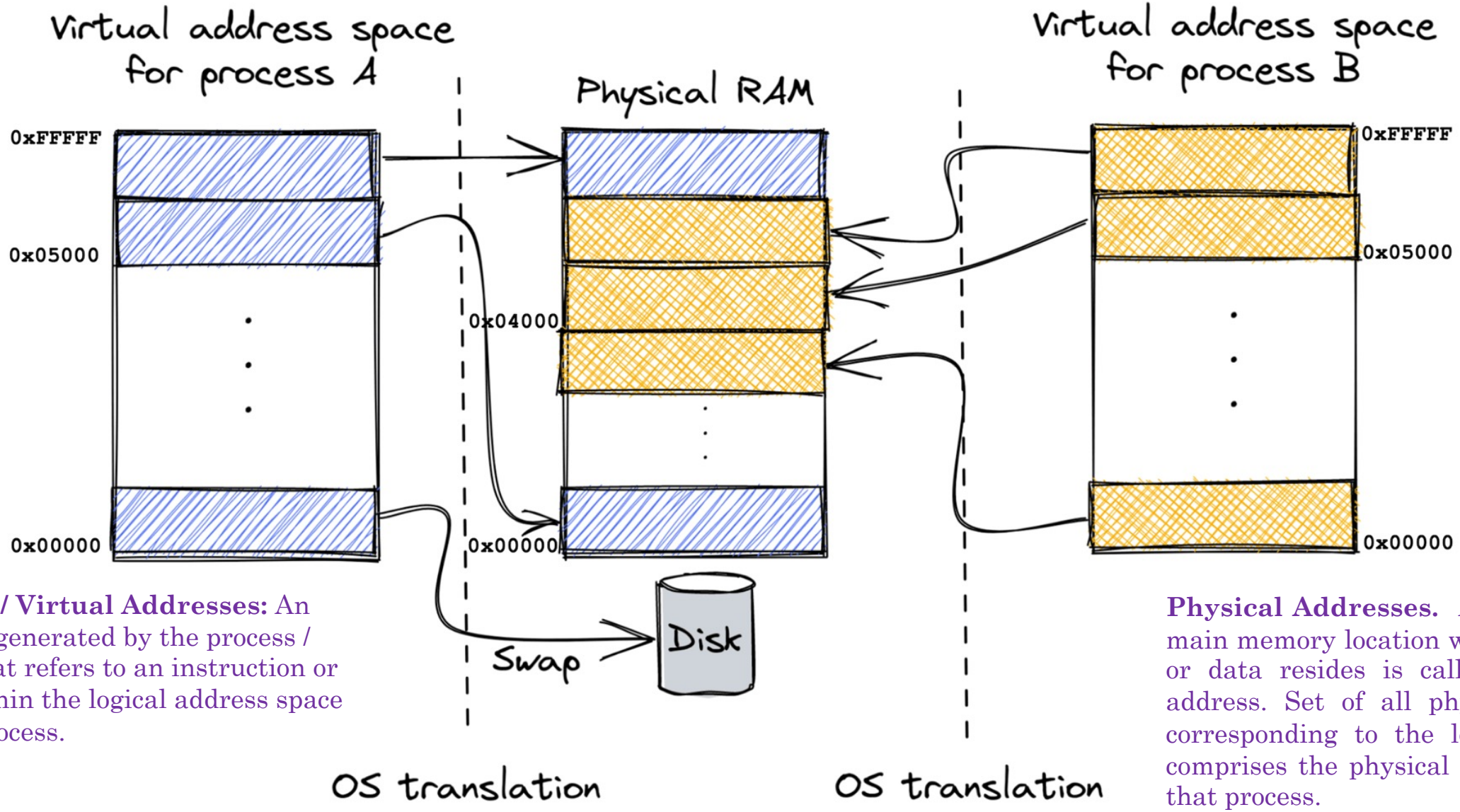
Non-Contiguous Memory Allocation

A memory management technique in which a process is divided into smaller parts that can be loaded into multiple non-adjacent blocks of physical memory. This approach allows better utilization of available space and eliminates external fragmentation, but it introduces additional overhead in address translation and may suffer from internal fragmentation depending on the scheme (e.g., paging, segmentation).

Memory Allocation Techniques

Allocation Techniques	Description	Advantages	Disadvantages
MFT	Memory divided into fixed-sized partitions, each assigned to a process.	Simple implementation, predictable partitioning.	Internal fragmentation (unused space within partitions), inflexible, multiprogramming limited.
MVT	Partitions dynamically sized to process needs; allocated at runtime.	More efficient memory use compared to fixed partitions	External fragmentation, complex allocation and compaction required.
Buddy System	Divides memory into power-of-two blocks; splits and coalesces "buddy" blocks as needed.	Reduces external fragmentation, fast allocation/deallocation, relatively easy to implement	Internal fragmentation due to rounding up size; still some memory waste; limited to power-of-two sizes.
Segmentation	Memory divided into variable-sized logical segments (e.g. code, data, stack).	Logical program organization, protection by segment, dynamic allocation	External fragmentation, complex address translation, harder to manage.
Paging	Divides both memory and processes into fixed-size pages and frames; uses page tables.	No external fragmentation, straightforward allocation, isolation between processes	Internal fragmentation (unused space in pages), page table overhead, address translation cost.
Paged Segmentation	A hybrid: segments (logical units) that are further divided into pages.	Logical grouping (like segmentation) + fragmentation reduction and protection (like paging)	High complexity, two-level translation overhead, management complexity.

LA to PA Address Translation



Logical / Virtual Addresses: An address generated by the process / CPU, that refers to an instruction or data within the logical address space of the process.

Physical Addresses. An address for a main memory location where instruction or data resides is called the physical address. Set of all physical addresses corresponding to the logical addresses comprises the physical address space of that process.

Segmentation

What is Segmentation?

Segmentation is a non-contiguous memory allocation technique in which a program is divided into variable-sized logical units called segments (such as code, data, and stack), each with its own base and limit. The operating system maps these segments independently into physical memory, enabling logical organization, protection, and sharing, though it can suffer from external fragmentation.

How Segmentation Work?

Key Features:

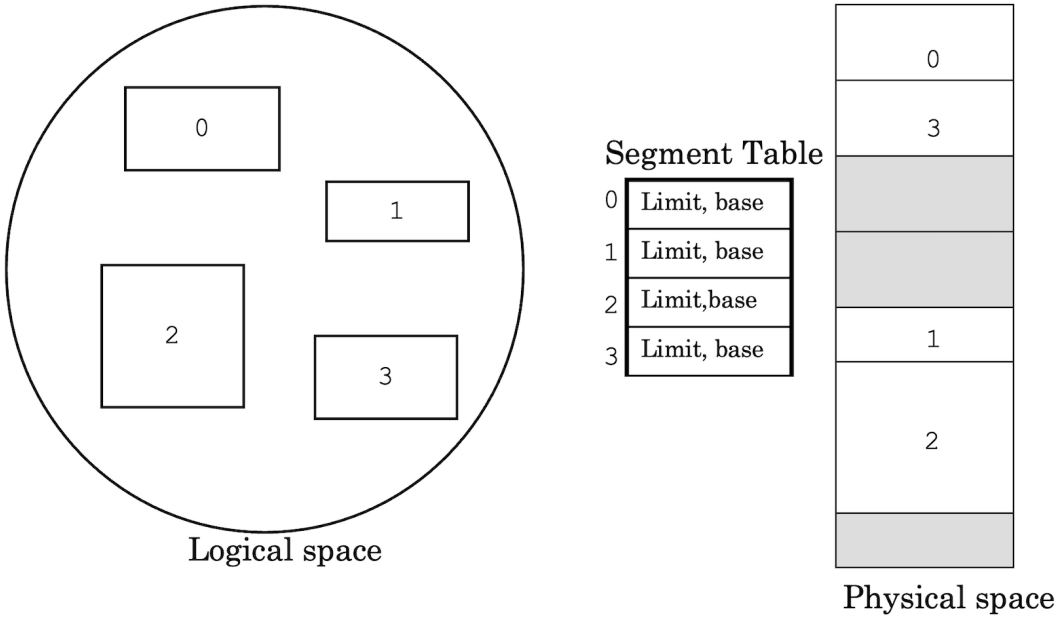
- Divides a program into variable-sized logical units (segments) like code, data, stack.
- Each segment has its own base (starting address) and limit (length).
- Provides logical view of memory aligned with how programs are structured.
- Allows independent protection and sharing at the segment level.

Advantages:

- Reflects programmer's logical program structure (modules, arrays, procedures).
- Enables fine-grained protection and controlled sharing of specific segments.
- Supports dynamic segment growth/shrinkage (e.g., stack expan
- Facilitates modular programming and separate compilation.

Disadvantages:

- Leads to external fragmentation, since segments are variable-sized.
- Complex address translation (needs segment table lookup).
- Segment tables can be large, requiring extra memory and management overhead.
- Slower access compared to simple contiguous schemes due to table lookup.



Segmentation: LA to PA Translation

Logical address = <segment#, offset>

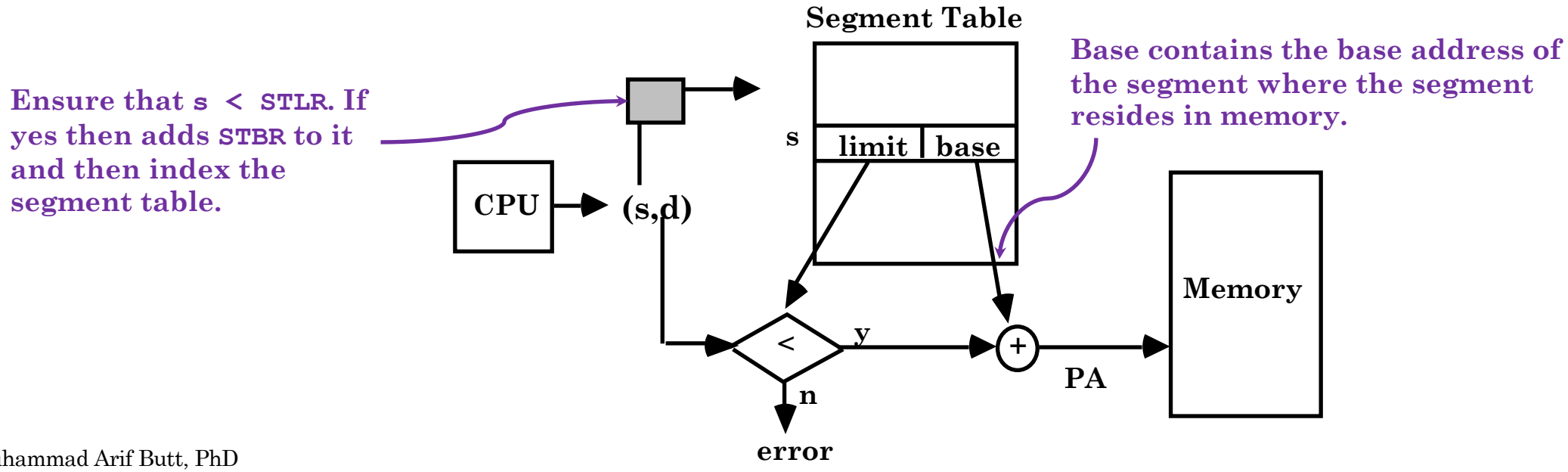
Each segment table entry has two values:

- **Base** contains the starting physical address where that segment resides in memory.
- **Limit** specifies the length of that specific segment.

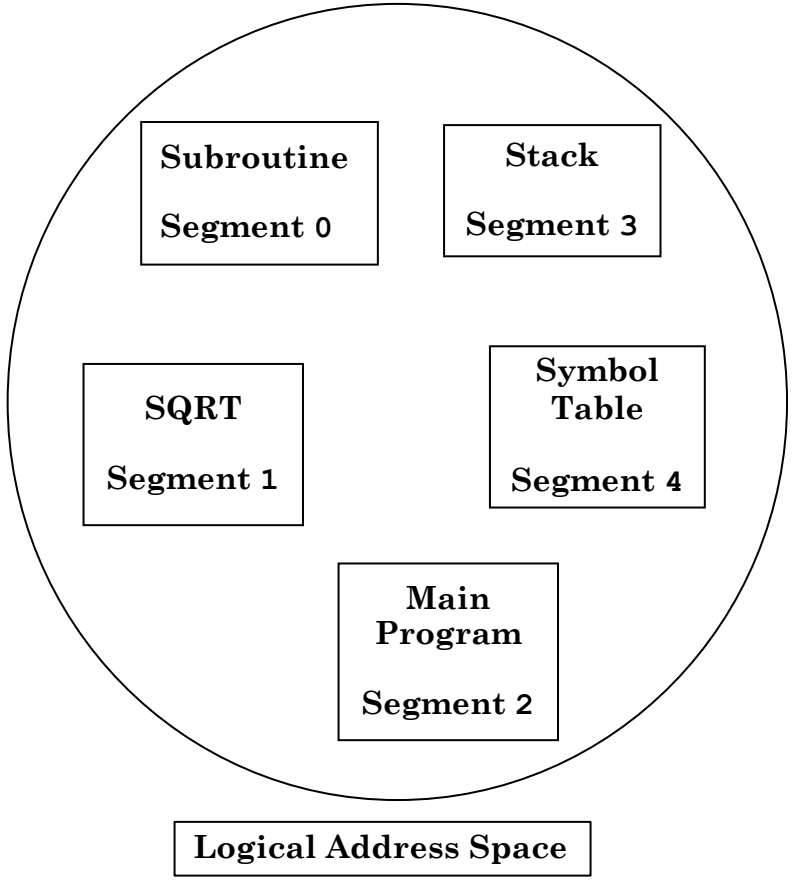
Segment Table Base Register (STBR): Points to the starting address of segment table in memory

Segment Table Length Register (STLR): Holds no. segments of program

A logical address is valid if `s< STLR` and `offset < limit`

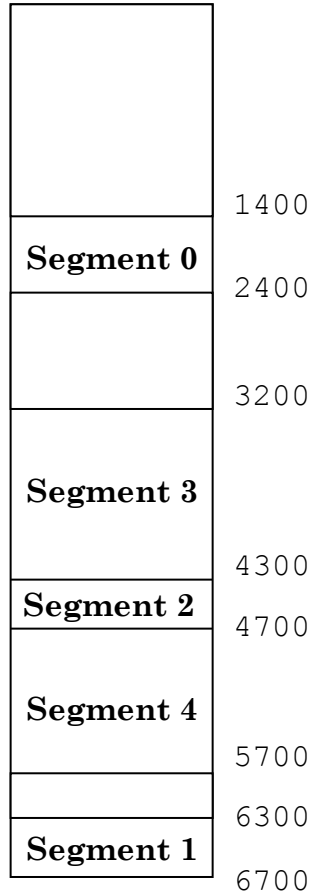


Segmentation: Example



Segment Table

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



Sample Problems

Problem 1 Consider the given segment table, What are the physical addresses for the following logical addresses?

- (2,399)
- (4,0)
- (4,1000)
- (3,1300)
- (6,297)

Segment Table

Segment #	Length	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Problem 2 Consider the given segment table, What are the physical addresses for the following logical addresses?

- (0,430)
- (1,10)
- (2,500)
- (3,400)
- (4,112)

Segment Table

Segment #	Length	Base
0	600	219
1	14	2300
2	100	90
3	580	1327
4	96	1952

Sample Problems (Cont.)

Problem 3 Consider the given segment table, What are the physical addresses for the following logical addresses:

- (0, 0)
- (2, 120)
- (6, 10)
- (5, 3399)
- (4, 1200)
- (0, 99)

Segment #	Length	Base
0	100	12000
1	1200	12100
2	190	13300
3	444	15500
4	19308	18008
5	3400	5000

Problem 4 Consider the above segment table, if segment table base register (STBR) contains 36500 and segment table entry size (STES) is 64 bits then what will be size of segment table? Also compute the address of the last entry?

Sample Problems (Cont.)

Problem 5 Compare between Segmentation and Base/Bound Address Translation Techniques

- Segmentation divides a process into multiple variable-sized logical units, while Base/Bound treats the process as a single continuous block.
- Segmentation uses a segment table (multiple base/limit pairs), while Base/Bound uses only one base and one limit register per process.
- Segmentation supports logical separation of code, data, and stack, whereas Base/Bound provides only relocation and protection for entire process
- Segmentation allows fine-grained protection and sharing at the segment level, while Base/Bound allows protection only for the whole process.
- Segmentation can suffer from external fragmentation, while Base/Bound faces the same limitation but with less flexibility.

External Fragmentation in Segmentation

External Fragmentation

- Occurs when free memory is split into many small non-contiguous holes, making it difficult to allocate large segments even if total free memory is sufficient.

Compaction as a Solution

- Involves shuffling/moving existing segments in memory so that scattered free spaces are merged into one large contiguous block.
- Helps in accommodating larger memory requests that otherwise wouldn't fit into fragmented holes.

Requirement of Dynamic Relocation

- Compaction requires dynamic relocation at execution time, because the physical location of segments changes during compaction.

Run-time Address Binding

- Needed so that the operating system can correctly map logical addresses to new physical locations after compaction.
- The segment table must be updated with the new base addresses of segments whenever they are moved.

Protection in Segmentation

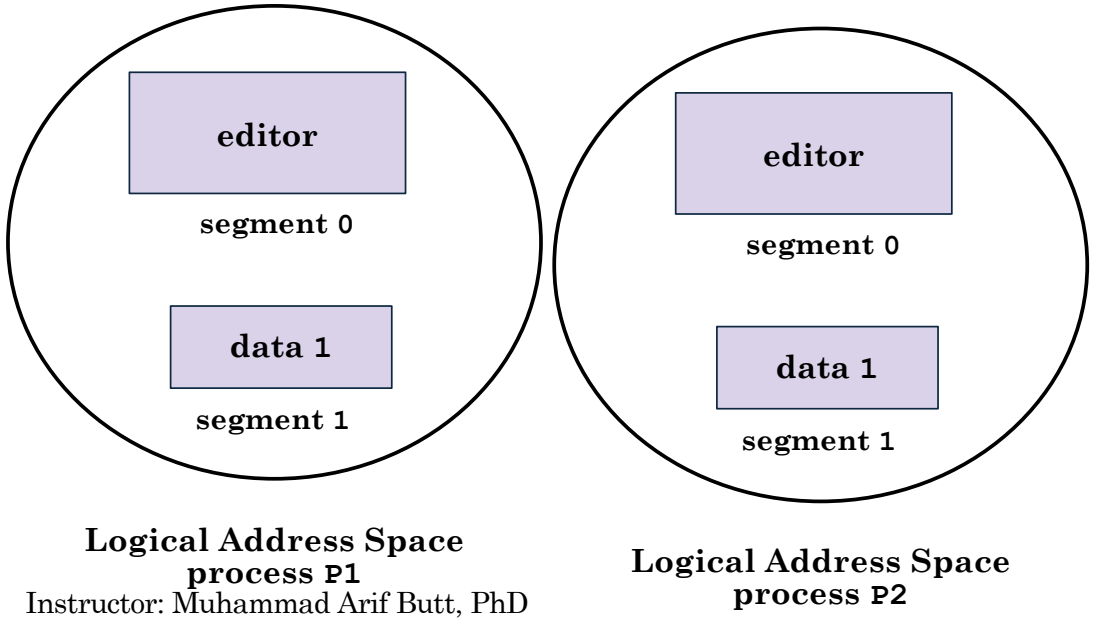
- Each entry in the segment table includes protection bits, as well specifying the read/write/execute permissions.
- These protection details may be stored in the segment table or in the segment registers managed by the memory management hardware.
- A segment descriptor typically contains the following fields:
  - **Base Address (selector)** – Starting physical address of the segment.
  - **Limit (Length)** – Maximum size of the segment.
  - **Protection/Access Rights** – Control bits (valid, read, write, execute).

Sharing of Segments among Processes

Sharing can be implemented at the segment level, i.e., the segment table entries of multiple processes can point to the same physical segment. Suppose Two processes P1 and P2 both use an editor program.

- The editor code segment is loaded only once in memory.
- Both P1 and P2 share this code segment → saves memory.
- Each process still has its own private data segments (e.g., buffers, user input).

The shared code segment is marked read-only/executable (so processes cannot overwrite it).

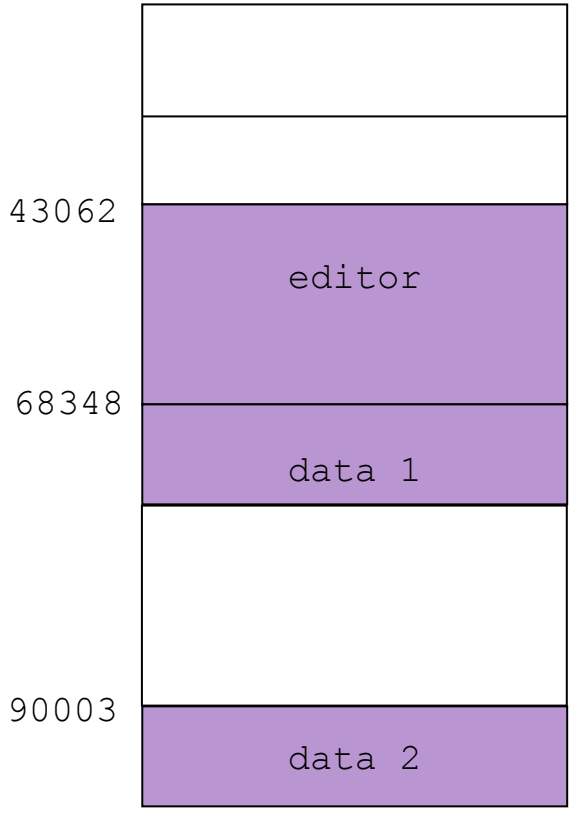


	limit	base
0	25286	43062
1	4425	68348

segment table process P1

	limit	base
0	25286	43062
1	8850	90003

segment table process P2



physical memory


Paging

What is Paging?

Paging is a non-contiguous memory allocation technique in which both logical and physical memory are divided into fixed-size blocks called pages (in logical memory) and frames (in physical memory). A page table maintained by the operating system maps each page to a frame, eliminating external fragmentation but introducing page table overhead and possible internal fragmentation within frames.

How Paging work?

Key Features:

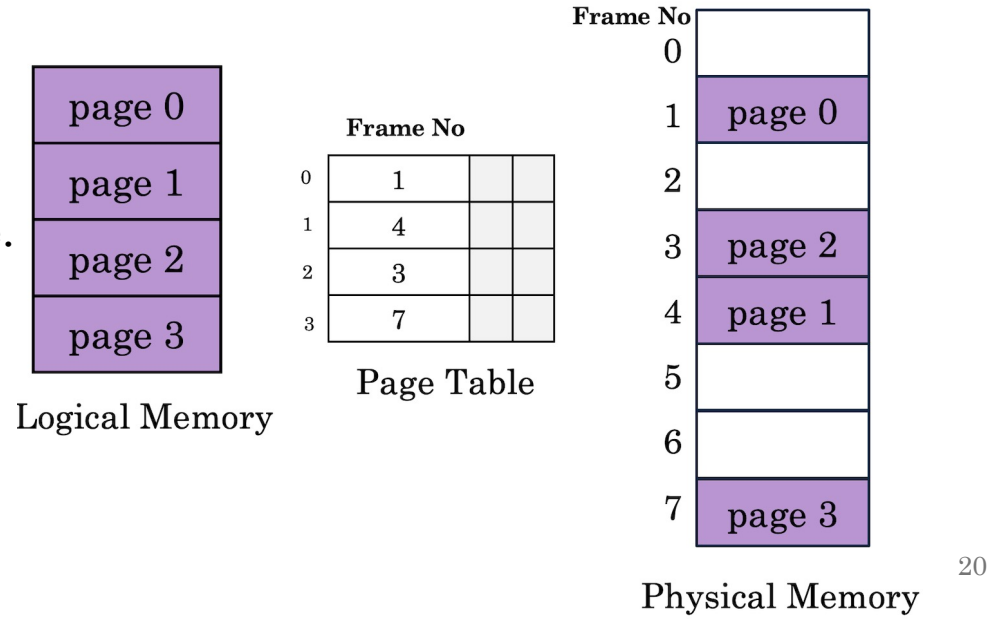
- Logical space is divided into pages while physical space is divided into frames (of same/fixed sizes).
- A process is loaded into memory by loading its pages into frames, which need not to be contiguous.
- Which page goes to which frame, this mapping information is kept in a structure calles page table.
- Provides a uniform abstraction of memory, hiding physical fragmentation.
- Eliminates external fragmentation by allocating memory in equal-sized chunks.

Advantages:

- No external fragmentation.
- Simplifies memory allocation and management due to fixed-size blocks.
- Enables efficient process swapping and virtual memory implementation.
- Provides strong isolation between processes.

Disadvantages:

- Can cause internal fragmentation (last page may waste space).
- Requires additional memory for page tables, which can be large.
- Address translation involves page table lookup, adding overhead.
- May increase memory access time without hardware support like TLB (Translation Lookaside Buffer).



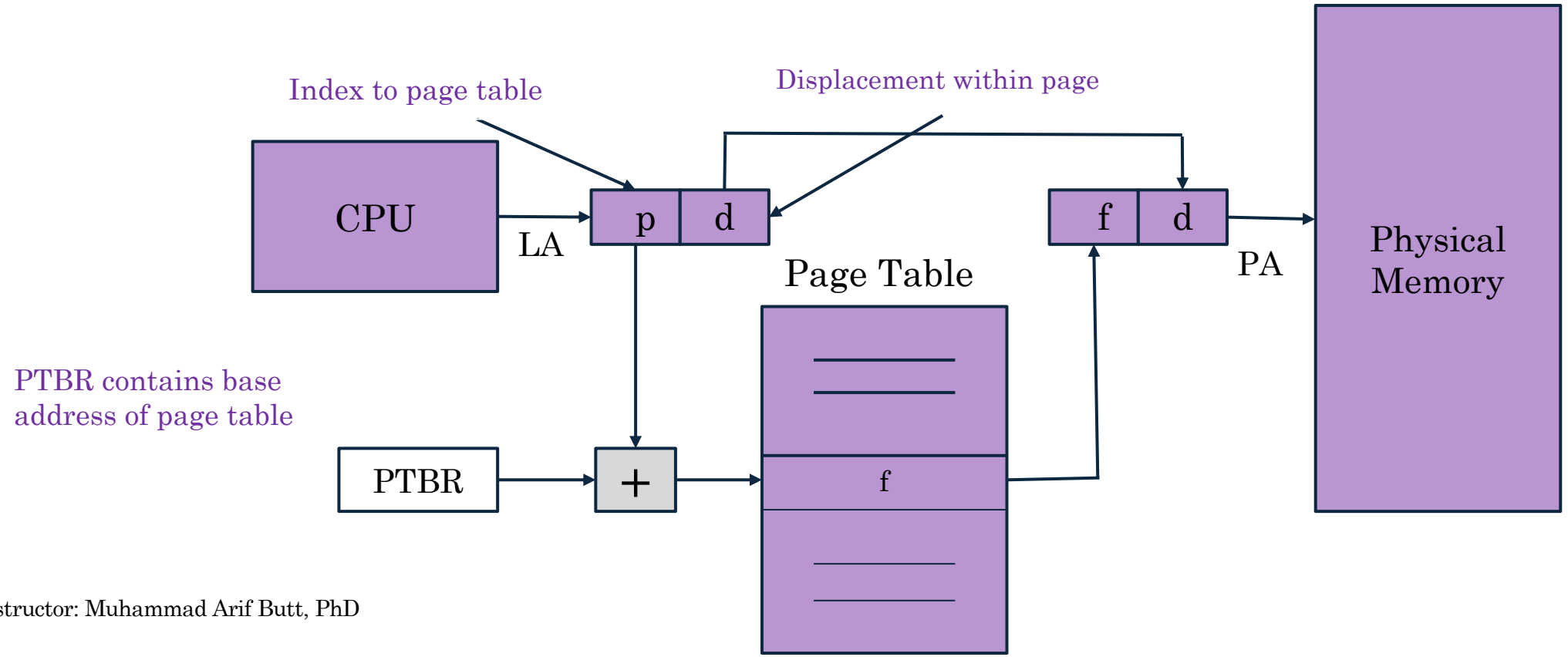
Paging: LA to PA Translation

Logical Address is divided into:

- **Page number (p)** used to index page table containing base address of that page in physical memory.
- **Page offset (d)** concatenated with base address of frame to define the physical address.

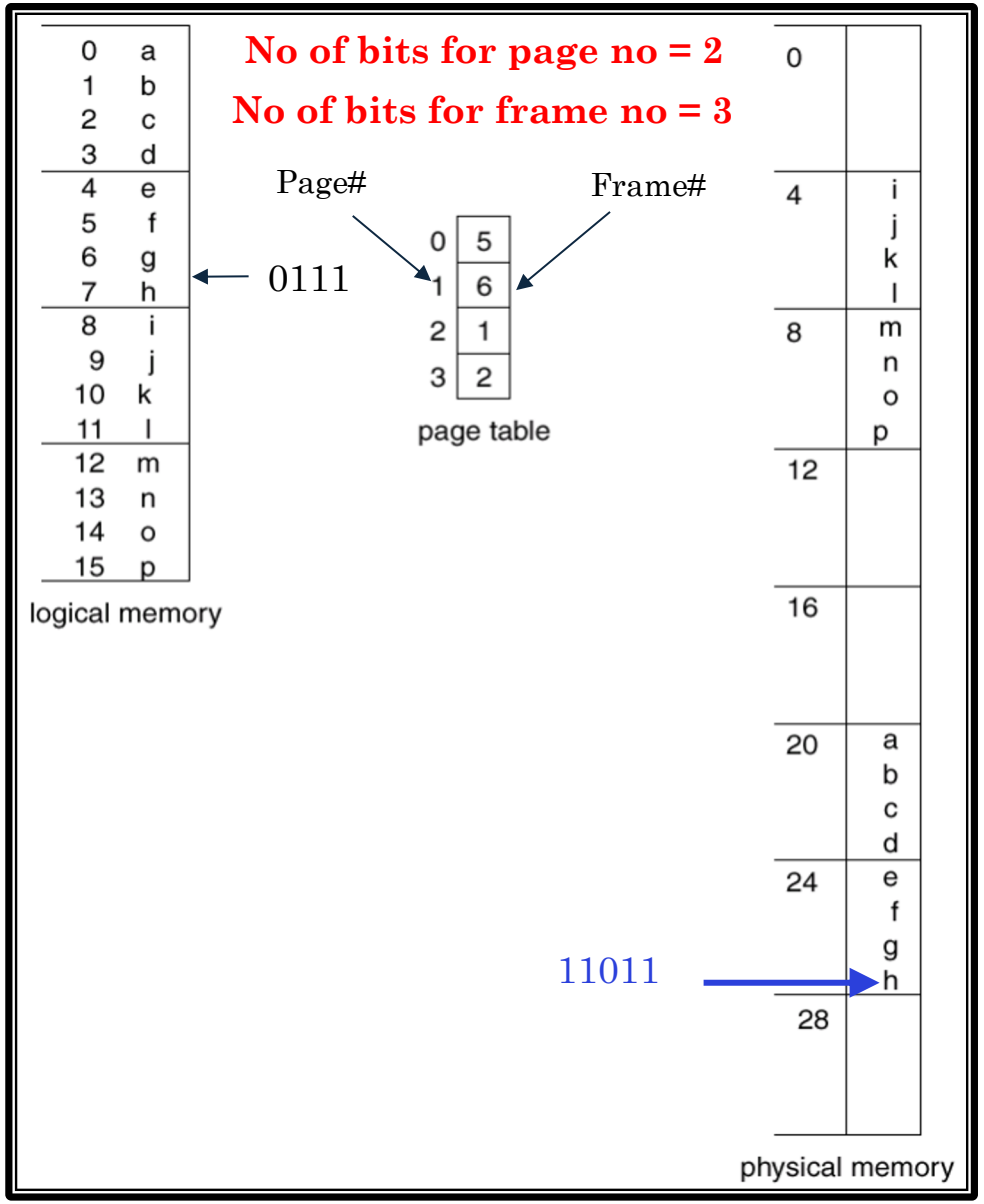
Physical Address is divided into:

- **Frame number (f)** kept in the page table against the page number.
- **Frame offset (d)** is the same as the page offset



Address Translation Example

- Page size = 4 bytes
- Process address space = 4 pages
- Physical address space = 8 frames
- Logical address: (p,d) (1,3) = 0111
- Physical address: (f,d) (6,3) = 11011

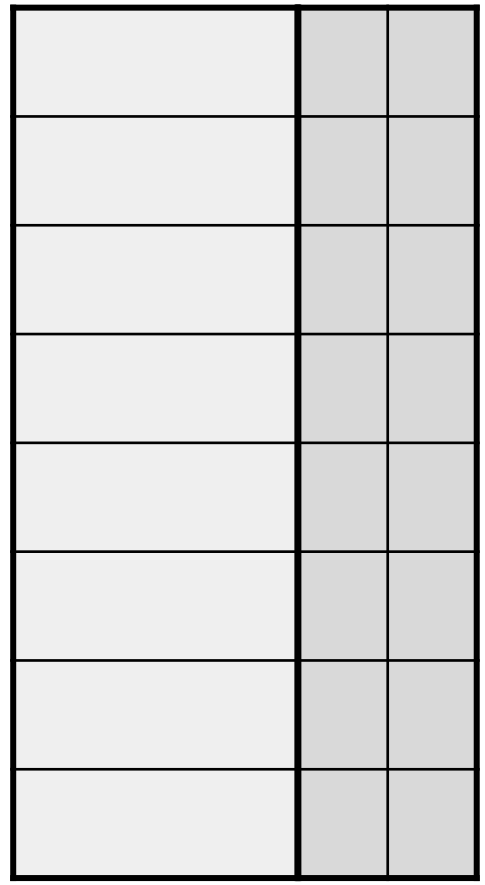


Page Tables

Page Tables are used to keep track of how logical addresses map to physical addresses. Page table entries generally contain the frame number where the particular page is loaded. In addition, a PTE may also contain:

- **Modify/Dirty bit:** Indicates whether a page is modified since loaded in memory.
- **Resident bit:** Indicates whether the page is resident in memory or not (may be on disk). It's not an error for a program to access a non-resident page. A page fault occurs and the page is brought into memory from disk.
- **Valid bit:** Page is legal for the program to access, e.g., is the page a process has requested is part of its address space. (If not Abort)
- **Protection bits:** Specify if a page is readable, writable or executable.

Page table size depends on the maximum number of pages of a process that a CPU support. For example, if a system support a process of maximum 8 pages then the page table will contain 8 rows (length is debatable). So the address of page table will consist of 3 bits.



Sample Problems

Problem 6 Given a logical address space consisting of 16 pages, each containing 1024 words (2 bytes per word), mapped into a physical memory of 32 frames:

- Specify the format of logical and physical addresses.
- Determine the total size of logical and physical address spaces.
- Calculate the required page table size.

Problem 7 In a system with a 48-bit logical address and 64 GB of main memory, using a page size of 4096 bytes:

- Compute the total number of pages and frames.
- Specify the format of logical and physical addresses.

Problem 8 Consider a system with a 32-bit logical address, a page size of 4 KB, and 512 MB of main memory:

- Determine the total process address space.
- Calculate the maximum number of pages per process.
- Specify the format of logical and physical addresses.
- Compute the page table size.

Sample Problems (Cont.)

Problem 9 Given a logical address space of 8 pages, each containing 1024 words, mapped into a physical memory of 32 frames:

- Determine the number of bits required for the logical address.
- Determine the number of bits required for the physical address.

Problem 10 A system features a logical address space comprising 64 pages, each of 512 bytes, mapped into a physical memory containing 1024 frames.

- Determine the bit lengths of the page number (p), offset (d), and frame number (f).
- Specify the formats of logical and physical addresses.

Problem 11 In a system with a 48-bit logical address, a 32-bit physical address space, and a page size of 4 KB:

- Calculate the bit lengths of p, d, and f.
- Specify the formats of logical and physical addresses.
- Determine the maximum number of pages per process and the maximum number of frames in the system.
- Compute the page table entry size (PTES) and the total size of the page table.

Sample Problems (Cont.)

Problem 12 A system supports a maximum of 2 million pages per process, with a page size of 2 KB. Determine the bit length of the logical address.

Problem 13 In a system with a 24-bit physical address space and a frame size of 512 bytes:

- Calculate the page table entry size (PTES).
- Determine the bit length of the physical address.

Problem 14 Given the following logical addresses in decimal: 20000, 32768, and 60000:

- Compute the corresponding page number and offset within the page for a page size of 4 KB.
- Repeat the computation for a page size of 8 KB.

Problem 15 A system has a 32-bit address space and uses 8 KB pages. The page table is fully implemented in hardware, with each entry occupying one 32-bit word.

- When a process starts, the page table is loaded from memory into hardware at a rate of one word per 100 nanoseconds.
- If each process runs for 100 milliseconds, including the time to load the page table, determine the fraction of CPU time spent on loading the page table.

Sample Problems (Cont.)

Problem 16 In a system with 48-bit virtual addresses and 32-bit physical addresses, using a page size of 8 KB. Calculate the number of entries required in the page table.

Problem 17 Given a logical address of 14000 and a page size of 1 KB, determine the page number in which this address resides.

Problem 18 In a system with a 34-bit logical address, a 32-bit physical address, and a 16 KB page size. Calculate the number of entries required in the page table.

Problem 19 Given a virtual address of 40808 and a page size of 4 KB. Compute the virtual page number and the offset within the page.

Sample Problems (Cont.)

Problem 20 What is the main difference between paging of memory and segmentation of memory?

Main Difference

•Paging: Divides memory into fixed-size blocks (pages/frames). Logical address = page number + page offset.

•Segmentation: Divides memory into variable-sized segments based on logical divisions of a program (code, stack, data, etc.). Logical address = segment number + segment offset.

Problem 21 List two ways in which this difference affects the address translation hardware required in each scheme.

1.Table Structure

- Paging: Needs a page table mapping each page to a frame → entries store frame number.
- Segmentation: Needs a segment table → entries store base address (start of segment) and limit (segment length).

2.Address Checking

- Paging: Hardware checks only the page table entry and **concatenate** offset to frame base address. No length check is needed because pages are fixed-size.
- Segmentation: Hardware must check that the offset < limit for protection, then **add** it to the base to form the physical address.

Paging Architectures

Paging Parameters in Intel P4

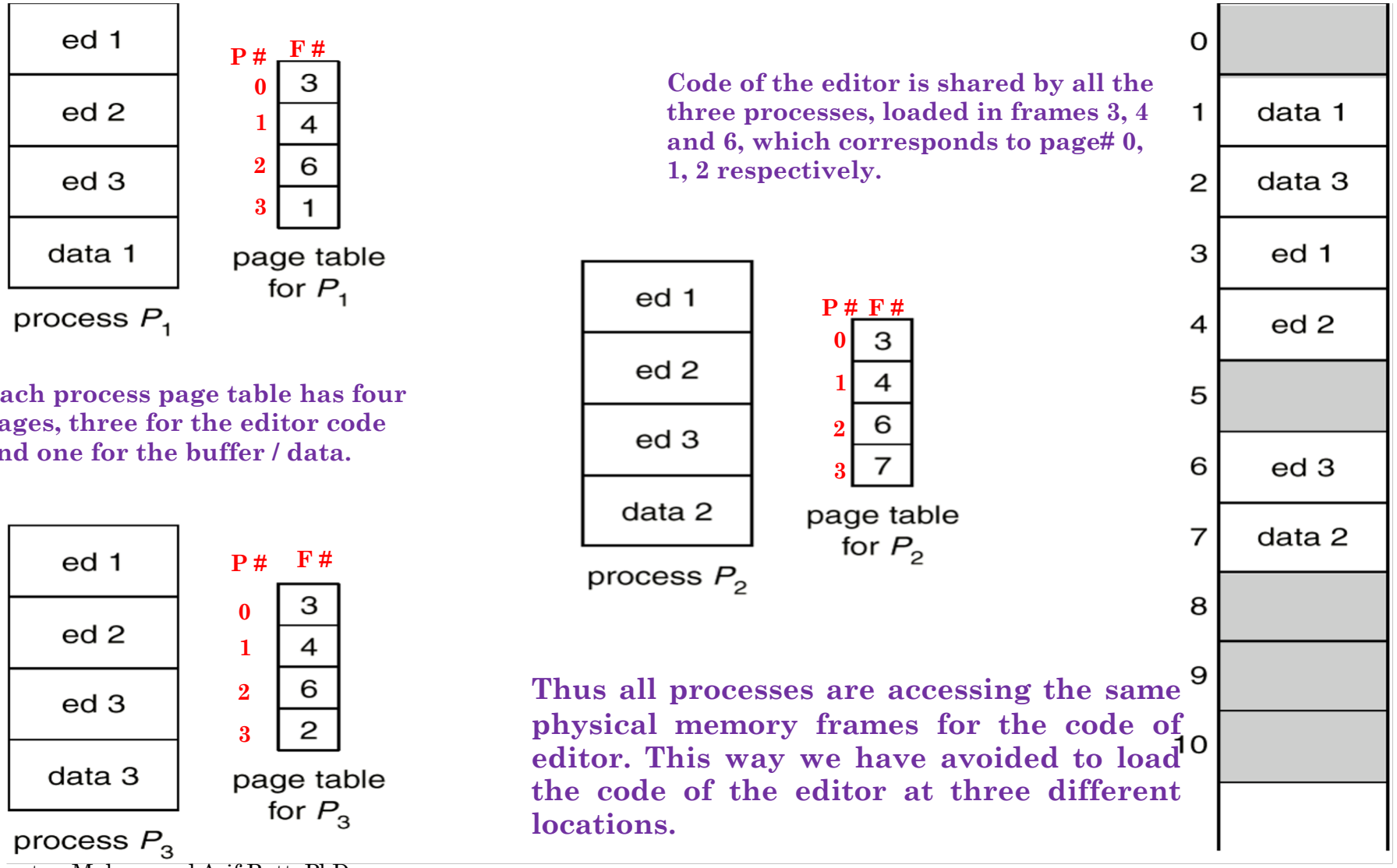
- 32 bit linear address. (Intel used the term linear instead of logical)
- 4K page size
- Maximum pages in a process address space = $2^{32}/2^{12} = 1M$
- No of bits for d = 12
- No of bits for p = 32 – 12 = 20
- What about the physical address format? / What about no of bits for f?

Paging Parameters in PDP 11

- 16 bit Logical address.
- 8K page size.
- Maximum pages in a process address space = $2^{16} / 2^{13} = 8$
- No of bits for d = 13
- No of bits for p = 16 – 13 = 3
- What about the physical address format? What about no of bits for f ?

Sharing of Pages among Processes

Consider three processes P1, P2, P3, each correspond to an editor, e.g. vim



Where the Page Table is Kept?

Where the Page Table is Kept?

IN MAIN MEMORY

- Page table is kept in main memory.
- Page-table base register (`PTBR`) points to the starting address of page table.
- Page-table length register (`PRLR`) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the page table (that resides inside the main memory) and one for the data / instruction.

$$T_{EFFECTIVE} = 2 \times T_{MEM}$$

Where the Page Table is Kept? (cont...)


If PT is kept inside the memory, system has to access memory twice, once to translate the LA to PA and second time to access the data?

IN CPU REGISTERS

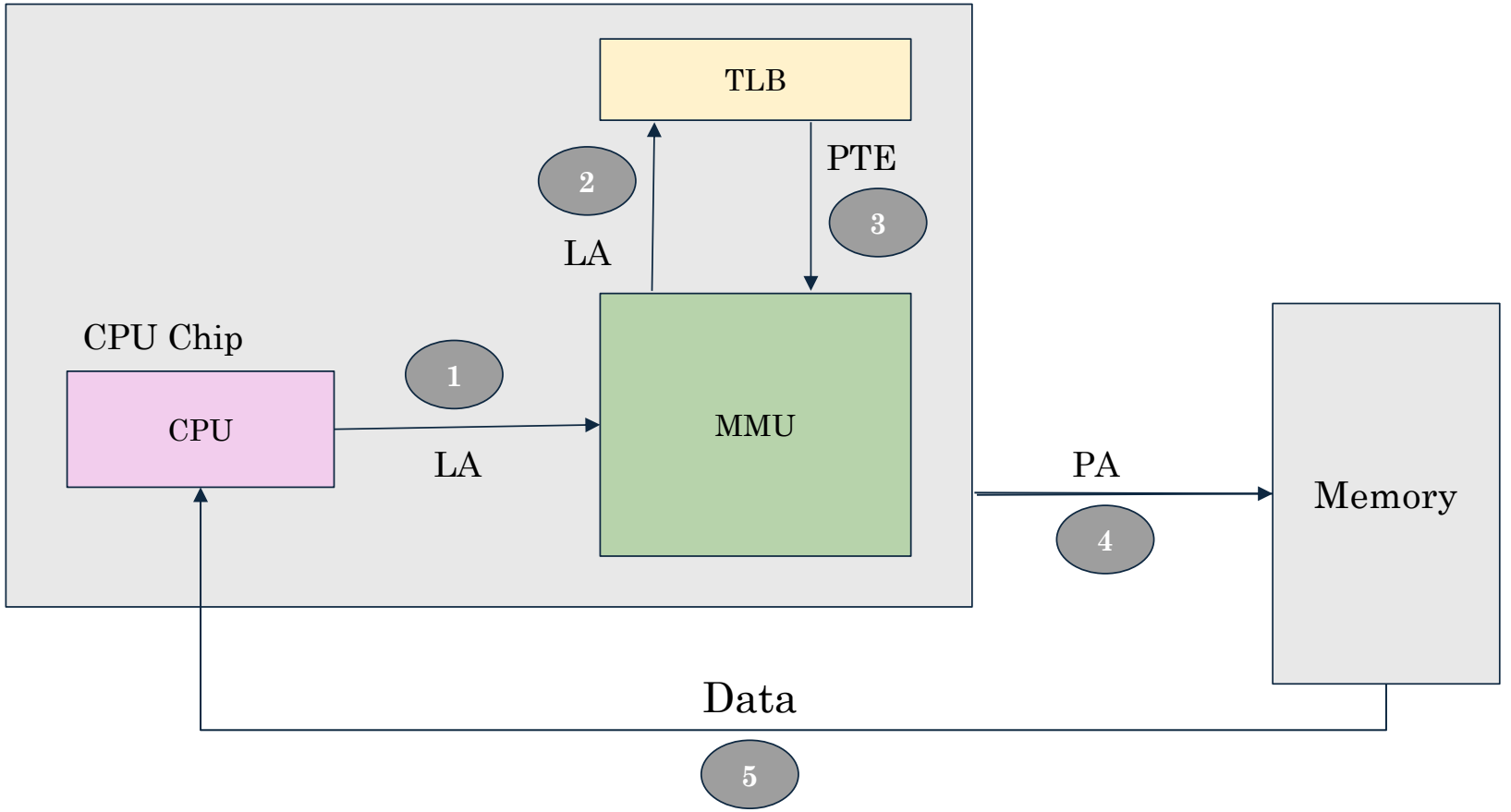
- Design CPU in such a way that page table can be kept / maintained within the CPU, using its registers.
- Feasible for small process address space with less number of pages which may be of large size.
- Effective Memory Access Time (time to convert `L.A` to `P.A`) is almost the same as the Physical memory access time.
- Example is `PDP-11`, which has eight pages each of size `8 KB`

IN CPU CACHE

Option 1: PTEs are cached in cache like any other memory word. However, PTEs may be evicted by any other data references

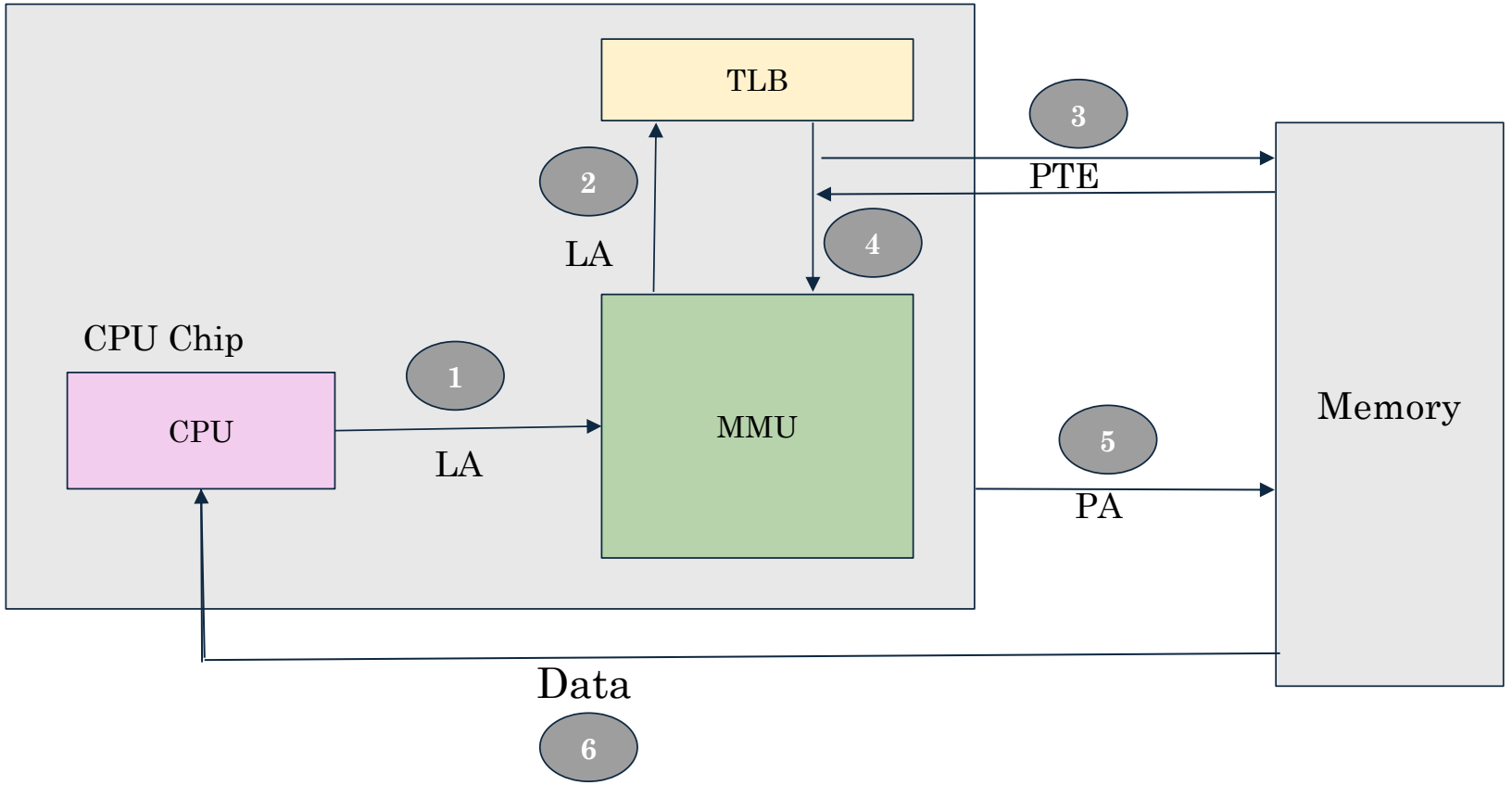
Option 2: Use a small, dedicated, superfast hardware cache of PTEs in MMU called Translation Look-aside Buffer (TLB). Place references of some of the recently used pages in TLB, i.e. a portion of page table resides in TLB and the rest in main memory. On a context switch, the TLB is flushed and loaded with values for the scheduled processes. (Normally TLB contains the page numbers of the currently running process). If mapping is found we say **Page hit / TLB hit** otherwise we say **Page fault / TLB miss**.

Address Translation - TLB Hit



Effective Memory Access Time (Page hit) = Time to access TLB + Time to access memory

Address Translation - TLB Miss



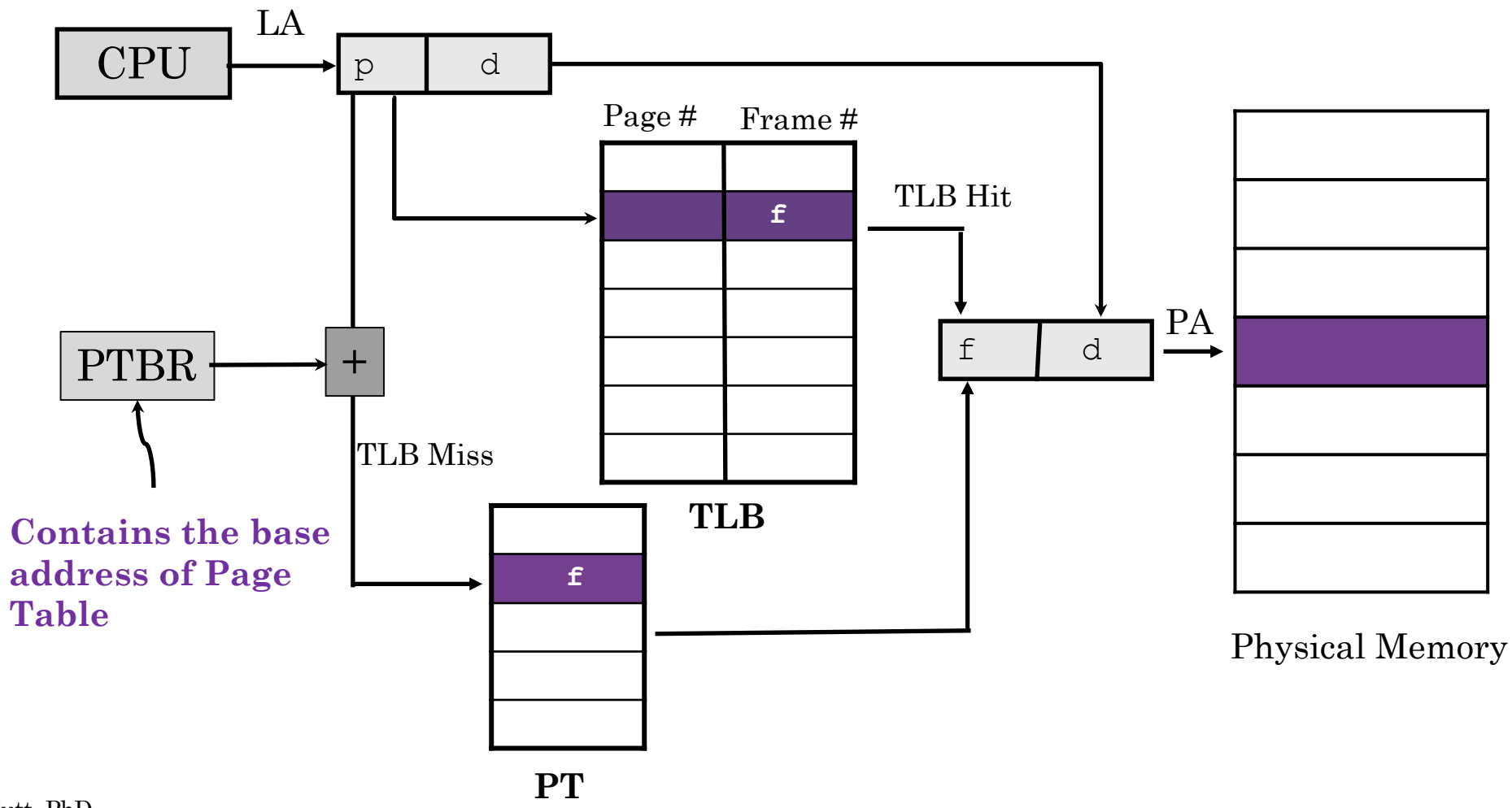
Effective Memory Access Time (Page miss) = Time to access TLB + 2 x Time to access memory

(Fortunately, TLB misses are rare)

Paging with TLB: LA to PA Translation

If **HR** is the hit ratio and **MR** is the miss ratio then:

$$T_{EFFECTIVE} = HR\ (T_{TLB} + T_{MEM}) + MR\ (T_{TLB} + 2\ T_{MEM})$$



Sample Problems

Problem 22 A system has a memory access time of 100 nanoseconds and uses associative memory to implement the page table. The TLB access time is 20 nanoseconds, with a hit ratio of 80%.

- Compute the effective memory access time.
- Recalculate the effective access time assuming no TLB, i.e., the page table is entirely stored in memory.

Problem 23 Repeat the calculations from Problem 26 using a TLB hit ratio of 95%.

- Compare the results to evaluate the impact of increased hit ratio on effective memory access time.

Problem 24 In a paging system where the page table is stored in memory, a single memory reference takes 200 nanoseconds.

- Determine the time required for a paged memory reference.
- If associative registers are added and 75% of page-table references are resolved through them (with zero access time when successful), compute the effective memory reference time.

Problem 25 A system uses a Translation Lookaside Buffer (TLB) with the following parameters:

- TLB hit ratio: 80%
- TLB access time: 15 nanoseconds
- Main memory access time: 150 nanoseconds

Calculate the effective memory access time in nanoseconds.

Sample Problems (Cont.)

Problem 26 A system employs both register-level caching and a TLB. The parameters are:

- Register hit ratio: 30%
- TLB hit ratio: 50%
- Register access time: 1 nanosecond
- TLB access time: 10 nanoseconds
- Main memory access time: 150 nanoseconds

Determine the effective memory access time in nanoseconds.

Problem 27 In a system with associative registers and main memory, the following parameters are given:

- Associative register hit ratio: 80%
- Associative register access time: 50 nanoseconds
- Main memory access time: 750 nanoseconds

Compute the access time when the page number is found in associative memory.

Compute the access time when the page number is not found in associative memory.

Calculate the effective memory access time.

Structure Of Page Table

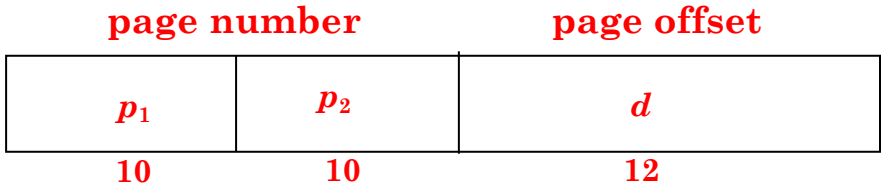
Structure Of Page Table

Consider a logical address space of 64 bits and page size of 4 KB. Also consider the 8 byte PTE. How big the page table need to be?

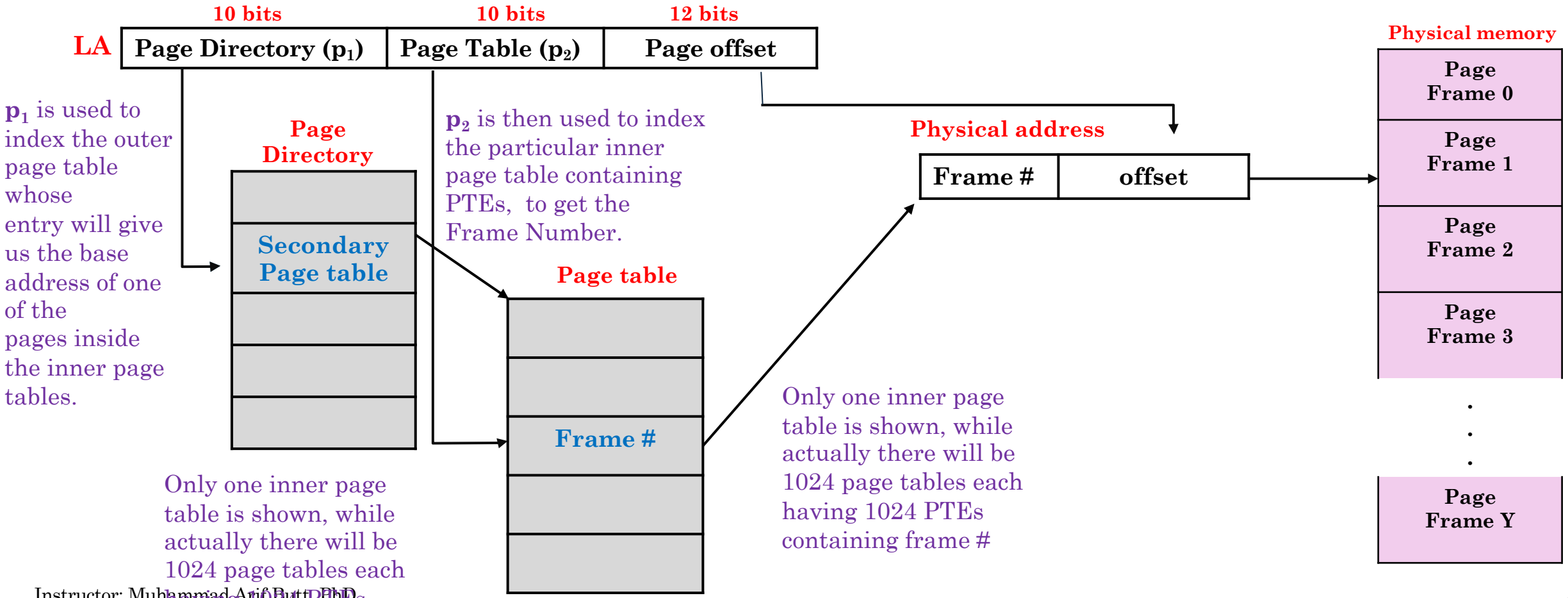
- Logical Address space increases day by day due to the large size of processes, thus increasing the size of the Page Table.
- Thus there is a dire need to structure Page Table in a better way especially in situation where the Page Table becomes larger in size than a single page size, i.e. a page table cannot be contained by a single page.
- To address this challenge, more efficient page table structures are required. Common techniques include:
  - **Multi-level (Hierarchical) Page Tables**
  - **Hashed Page Tables**
  - **Inverted Page Tables**

Multi level / Hierarchical Page Tables

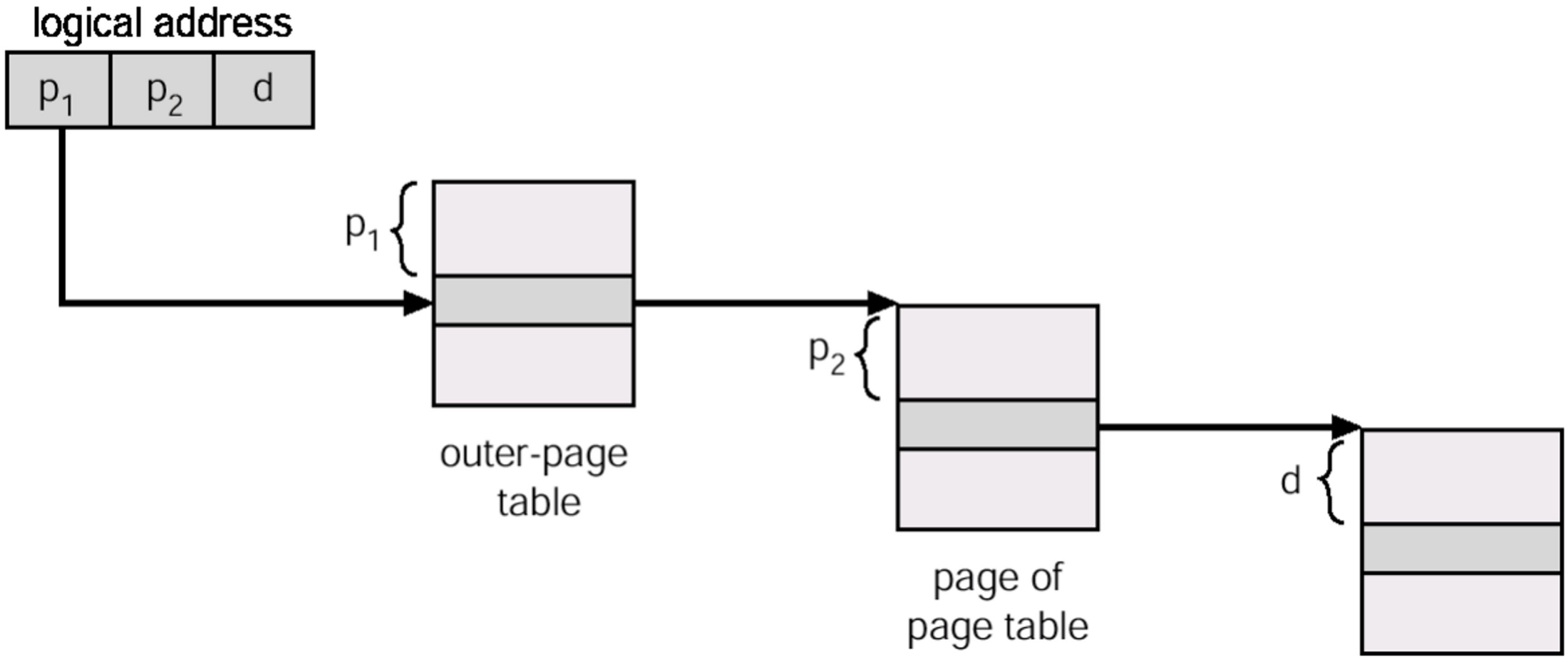
- Consider a system with a logical address of 32 bits, and page size of 4KiB and each page entry of 4 Bytes
- Maximum pages in a process address space = 1MiB
- Page Table size = 4 MiB (because each entry in page table is of 4 Bytes). Since total no of pages are 2^{20} i.e. 1 MiB, so there will be 2^{20} rows/ tuples /entries, each tuple of 4 Bytes. So page table size will be 4 MiB
- Since the system has a page size of 4KiB, therefore, the page table of 4MiB can't be accommodated in a single page of 4KiB. Thus we have to make pages of the page table
- We keep two page tables:
  - **Outer page table / page directory** (which keep track of the pages of the inner page table)
  - **Inner Page Table** which actually maps the frames
- No of pages in the outer page table = 4M / 4K = 1K
- So size of outer page table = 1K entries of 4 bytes each = 4KB
- This outer page table will now fit in one page



Hierarchical PT: LA to PA Translation



Two-Level Page Table Scheme



Example Processors having Multi-Level Page Tables

- Some example processors that use multi-level paging:
  - 32 bit Sun SPARC support 3 Level Paging
  - 32 bit Motorola 68030 support 4 Level Paging
  - 64 bit Sun Ultra SPARC support 7 Level Paging
- Since each level is stored as a separate table in memory, converting a logical address to a physical one will take multiple memory accesses.
- As the no of levels increases, too many memory references needed for address translation.
- But at times this is required to support large process address space, so that larger processes can be executed in larger applications.

Inverted Page Tables

An **Inverted Page Table** is a memory management structure that maintains one entry per **frame** in physical memory, rather than per **page** in a process's logical address space. This approach significantly reduces the size of the page table, especially in systems with large virtual address spaces.

Key Characteristics:

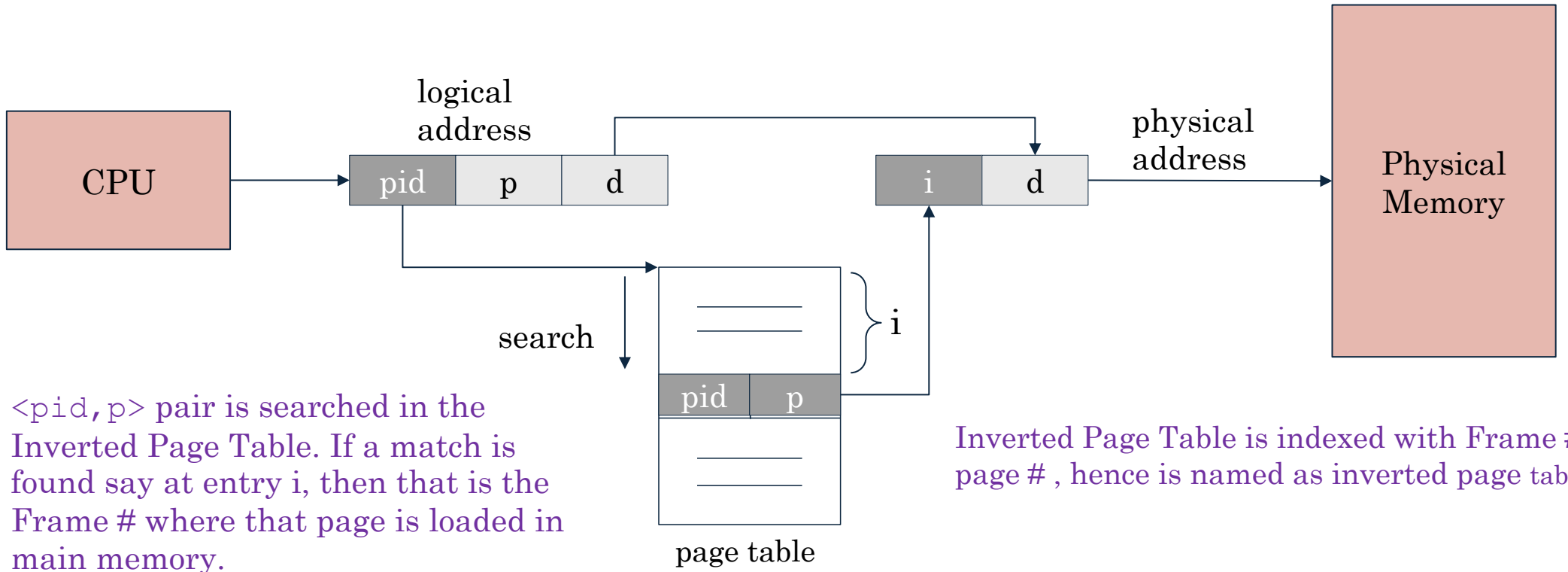
- One entry per physical frame.
- Indexed by frame number (f).
- Each entry stores the page number and process ID (PID) to identify ownership.

Advantages:

- **Space Efficiency**:
  - Table size depends on the number of physical frames, not the size of the logical address space.
  - Only one global page table is maintained for all processes.
- **Process Isolation**:
  - Inclusion of **PID** ensures correct mapping even when multiple processes use the same page number.

Inverted PT: LA to PA Translation

This scheme decreases the amount of memory needed to store each page table but increases the time needed to search the table. It is because inverted page table is sorted by Physical address, but looks up occur on Virtual address. So the whole table might need to be searched for a match.



<pid,p> pair is searched in the Inverted Page Table. If a match is found say at entry i, then that is the Frame # where that page is loaded in main memory.

Inverted Page Table is indexed with Frame # i, and not with page # , hence is named as inverted page table

Inverted Page Table has entries equal to the number of frames in Main memory. Each entry contain the pid and its page number.

Hashed Page Tables

The **Hashed Page Table** is an efficient memory management technique designed to handle **large virtual address spaces**, especially in systems with **64-bit architectures**. It uses a **hashing mechanism** to locate page table entries, making it suitable for sparse address spaces.

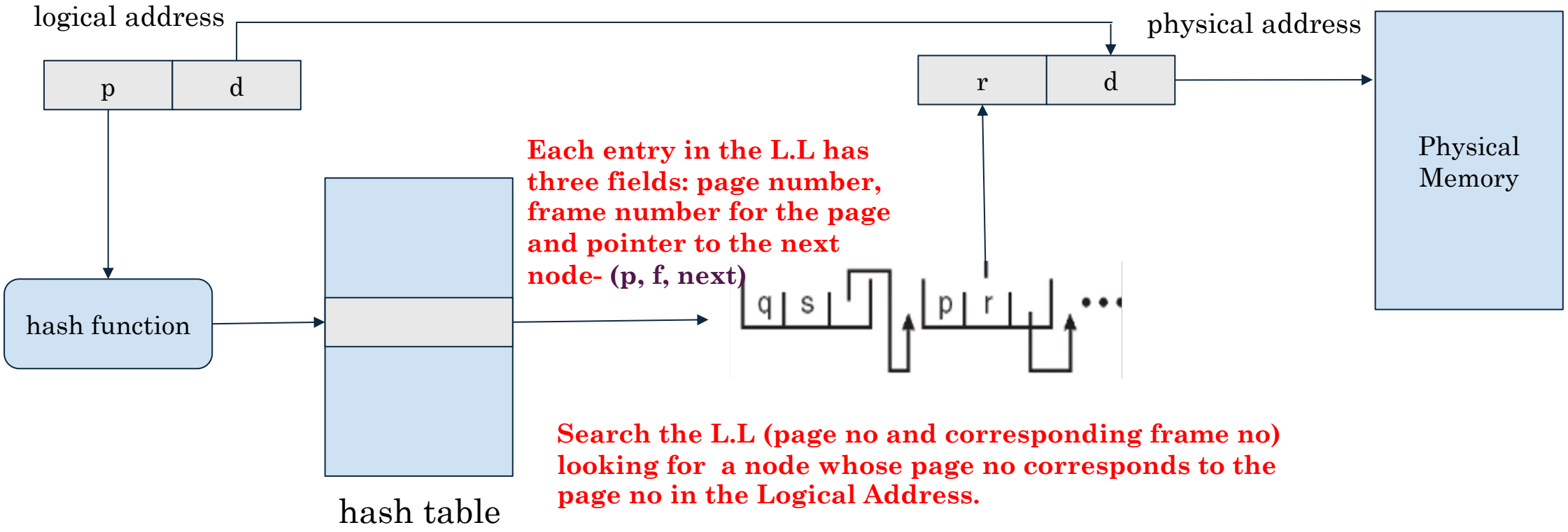
Key Concepts

- **Hashing**
  - A technique where a **key** (typically the virtual page number) is passed through a **hash function**
  - The result is a **hash value**, which indexes into a **hash table**
  - The indexed slot contains either the desired entry or a **linked list** of entries that share the same hash value (to resolve collisions)
- **Hash Function**
  - A deterministic function that maps a key to a valid index within the hash table
  - Ensures uniform distribution of entries and minimizes collisions
  - Must be fast and produce values within the bounds of the table size

Advantages

- **Scalable for 64-bit systems:** Efficiently handles large and sparse virtual address spaces
- **Space-saving:** Avoids allocating large contiguous page tables
- **Fast lookup:** Hashing provides near-constant time access in ideal conditions

Hash PT: LA to PA Translation



- Hashing can be combined with Inverted Page Tables, to overcome its limitation of complex / time consuming search.
- For 64 bit address space **Clustered Page Tables** are used.

Sample Problems


Problem 28 A system has a logical address space of 28 bits and a page size of 4 KiB. Each page table entry (PTE) requires 4 bytes. A memory pointer is 4 bytes. Answer the following:

- How many pages are there in the logical address space?
- How many entries fit in one page of a page table?
- What is the minimum number of paging levels required?
- Give the logical address format for the two-level scheme.

Problem 29 A system uses a 40-bit logical address space with a page size of 2 KiB. Each page table entry = 8 bytes. Each memory frame = 2 KiB. Answer the following:

- How many pages are in the logical address space?
- How many entries fit in one page of the page table?
- What is the minimum number of paging levels required?
- Give the logical address format for a three-level scheme.

Problem 30 Consider a logical address space of 36 bits with a page size of 8 KiB. Each PTE = 4 bytes. A page table must fit inside a page. Answer the following:

- How many entries can a single page table hold?
- What is the maximum number of levels required?
- Show the logical address format for minimum paging levels.

Sample Problems

Problem 31 A system has a 32-bit logical address space, page size = 2 KiB, and PTE size = 2 bytes.

- How many entries are needed for a single page table?
- How many entries fit in one page?
- Calculate the internal fragmentation in the last page table page.

Problem 32 Consider a system with a 32-bit logical address and a page size of 4KiB. Assume that the memory address take 20 bits.

- Can it be implemented in single level paging?
- If not what is the minimum level of paging it requires?
- employs a two-level page table. The virtual address is divided into three fields:
    - 9-bit top-level page table index
    - 11-bit second-level page table index
    - Offset field comprising the remaining bits

Problem 33 Consider a logical address space of 32 bits and page size of 1 KiB. A memory pointer takes two bytes. Each entry in the outermost page table also takes two bytes. Give answer to following questions:

- How many minimum levels of paging is required for the system? If not what is the minimum level of paging it requires?
- Give the logical address format that takes minimum internal fragmentation in the page tables
- Give the logical address format that takes maximum internal fragmentation in the page tables
- Calculate the amount of internal fragmentation for the best case in the first level only

Sample Problems (Cont.)

Problem 34 A system with a 36-bit logical address space, 4 KiB page size, and 64 GiB physical memory supports 32-bit process identifiers (PIDs). The task is to determine the following parameters:

- Number of bits for page number (p), offset (d), and frame number (f)
- Number of page table entries (PTES) in the inverted page table
- Size of the inverted page table
- Format of logical and physical addresses

Problem 35 A system uses 32-bit process identifiers and employs an inverted page table for address translation. The 34-bit logical address is divided into a 22-bit page number (p) and 12-bit offset (d). The physical address is 32 bits wide. Determine the following:

- Page size
- Number of page table entries (PTES)
- Total number of pages
- Size of the inverted page table

Sample Problems (Cont.)

Problem 36 In a system with 2048 MiB of RAM and a 4 KiB page size, calculate the total number of entries required in the inverted page table.

Problem 37 On a 64-bit architecture with 256 MiB of physical memory and a 4 KiB page size, determine the number of entries in the inverted page table.

Problem 38 A system has a logical address of 32 bits, physical memory size of 256 MiB, and Page size of 4 KiB. Each Inverted Page Table entry stores the Process ID (16 bits) and Virtual Page Number (20 bits) plus some control bits (assume 4 bits). Answer the following questions:

- How many frames are in physical memory?
- How many entries will the inverted page table contain?
- What is the total size of the inverted page table in bytes?

Problem 39 A system has a logical address of 30 bits, physical memory size of 64 MiB, and Page size of 1 KiB. Each Inverted Page Table entry stores the PID of 12 bits, VPN of 20 bits, plus 4 control bits. Given a logical address `0x12345A` for process PID = 25, answer following:

- Extract the Virtual Page Number (VPN) and page offset.
- Show how the inverted page table lookup works to find the frame number.
- Compute the physical address if the Inverted Page Table (IPT) lookup returns Frame Number = 1000.

Sample Problems (Cont.)

Problem 40 A system has a Logical address of 32 bits, physical memory of 512 MiB, and Page size of 4 KiB and uses hash page table. Assume each hash table bucket has one entry of size 8 bytes (VPN + PFN + next pointer).

Answer the following questions:

- How many pages are there in the logical address space?
- How many frames are in physical memory?
- What is the minimum size of the hash table in bytes if the load factor is 1 (i.e., one entry per bucket on average)?

Problem 41 A system uses a hashed page table with a Logical address of 28 bits, Page size = 1 KiB. The Hash function: `h(VPN) = VPN mod 256`. Assume hash table has 256 buckets and each entry stores (VPN, PFN). Moreover, following information is also known:

- VPN = `0x1F3` maps to PFN = `0x2A`.
- VPN = `0x123` maps to PFN = `0x15`.
- Logical address = `0x1F3_01C` (underscore separates VPN and offset).

Answer the following questions:

- Extract the VPN and page offset.
- Show how the hash function maps this VPN to a bucket.
- Find the physical address.

To Do

- Carefully review all concepts discussed in class and go through the slides to build a clear understanding of non-contiguous memory allocation schemes.
- Revise the key differences between paging, segmentation, and inverted/hashed page tables.
- Practice all numerical problems given in the slides and attempt similar ones from recommended books.
- Draw and analyze address translation diagrams (paging, segmentation, hierarchical paging, IPT) to strengthen visualization.
- Summarize the advantages, disadvantages, and real-world usage of each memory allocation scheme in your own words.
- Prepare a list of possible exam-style short/long questions from today's content and try answering them without notes.
- Form small groups to discuss and cross-check answers for numerical and conceptual questions.

Coming to office hours does NOT mean that you are academically weak!