Operating System

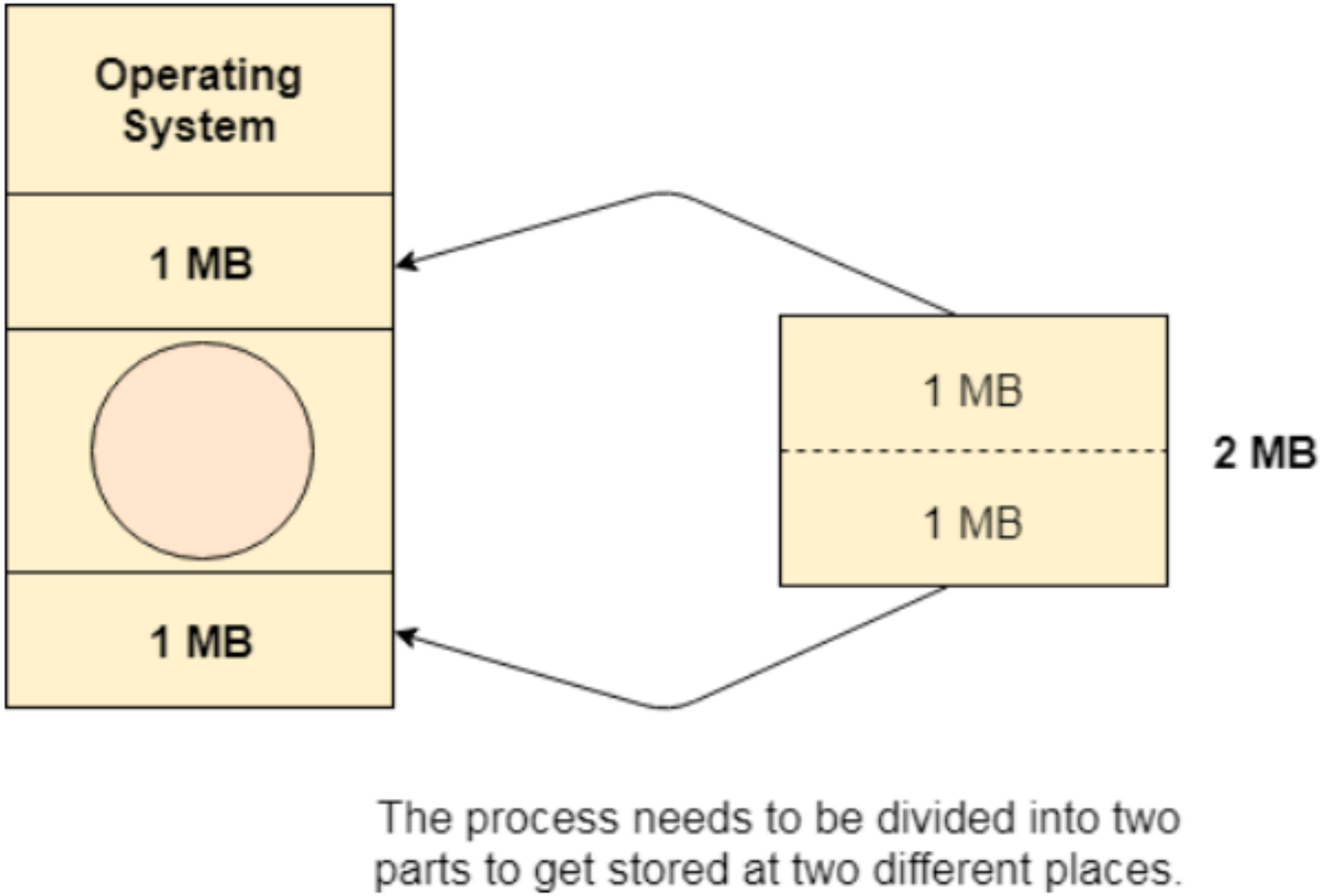
Unit – 5
(Part-B)
Memory Management

Mayank Mishra
School of Electronics Engineering
KIIT-Deemed to be University

Need of Paging

- The main disadvantage of Dynamic Partitioning is External fragmentation. Although, this can be removed by Compaction but as we have discussed earlier, the compaction makes the system inefficient.

- Example: Lets consider a process P1 of size 2 MB and the main memory which is divided into three partitions. Out of the three partitions, two partitions are holes of size 1 MB each.
  - P1 needs 2 MB space in the main memory to be loaded. We have two holes of 1 MB each but they are not contiguous.
  - Although, there is 2 MB space available in the main memory in the form of those holes but that remains useless until it become contiguous. This is a serious problem to address.
  - We need to have some kind of mechanism (The method is PAGING) which can store one process at different locations of the memory.



The process needs to be divided into two parts to get stored at two different places.

The Idea behind paging is to divide the process in pages so that, we can store them in the memory at different holes.

Need of Paging (Contd.)

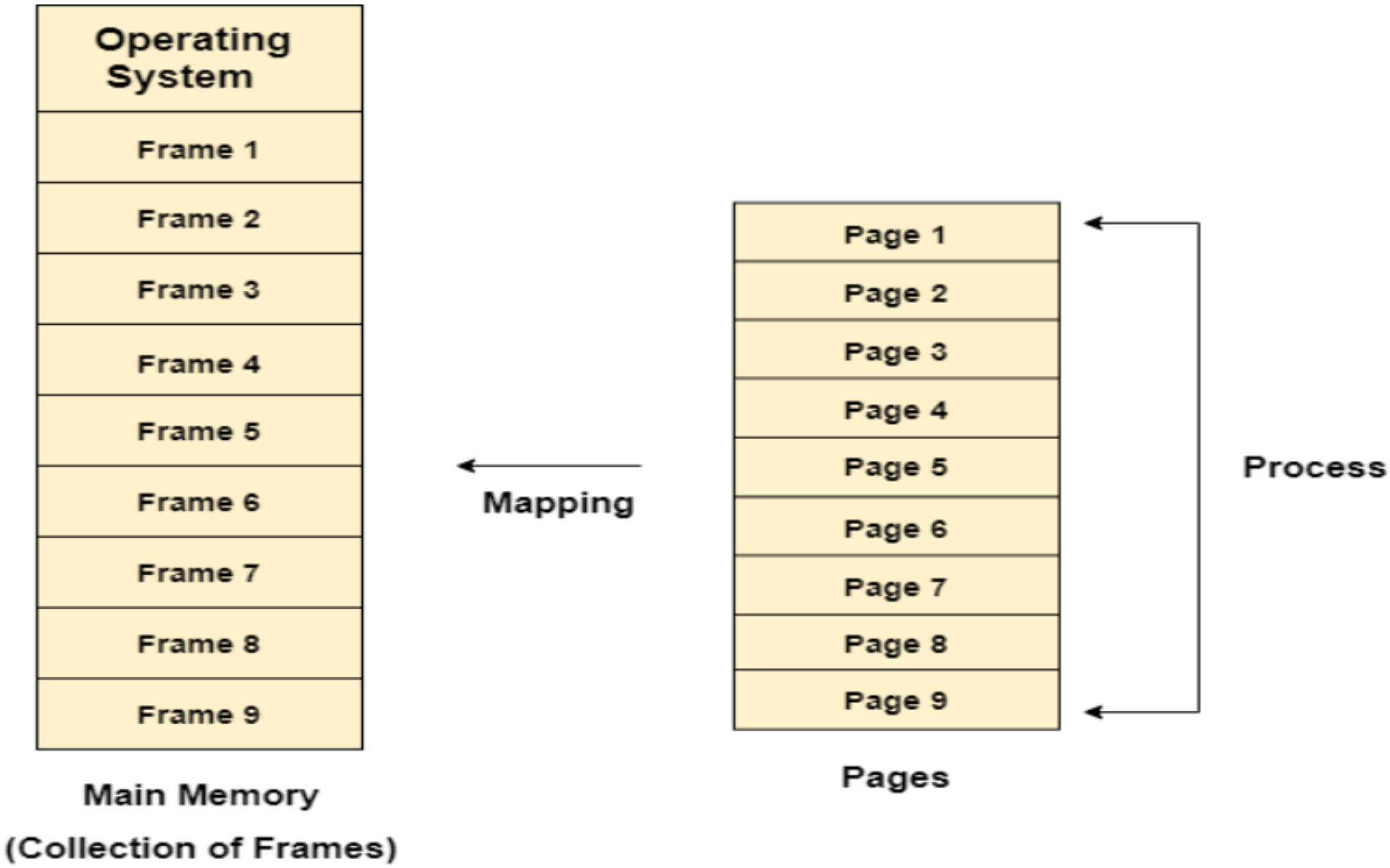
- In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages.

- The main idea behind the paging is to divide each process in the form of pages. The main memory will also be divided in the form of frames.

- One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes.

- Pages of the process are brought into the main memory only when they are required otherwise they reside in the secondary storage.

- Different operating system defines different frame sizes. The sizes of each frame must be equal. Considering the fact that the pages are mapped to the frames in Paging, **page size needs to be as same as frame size**.

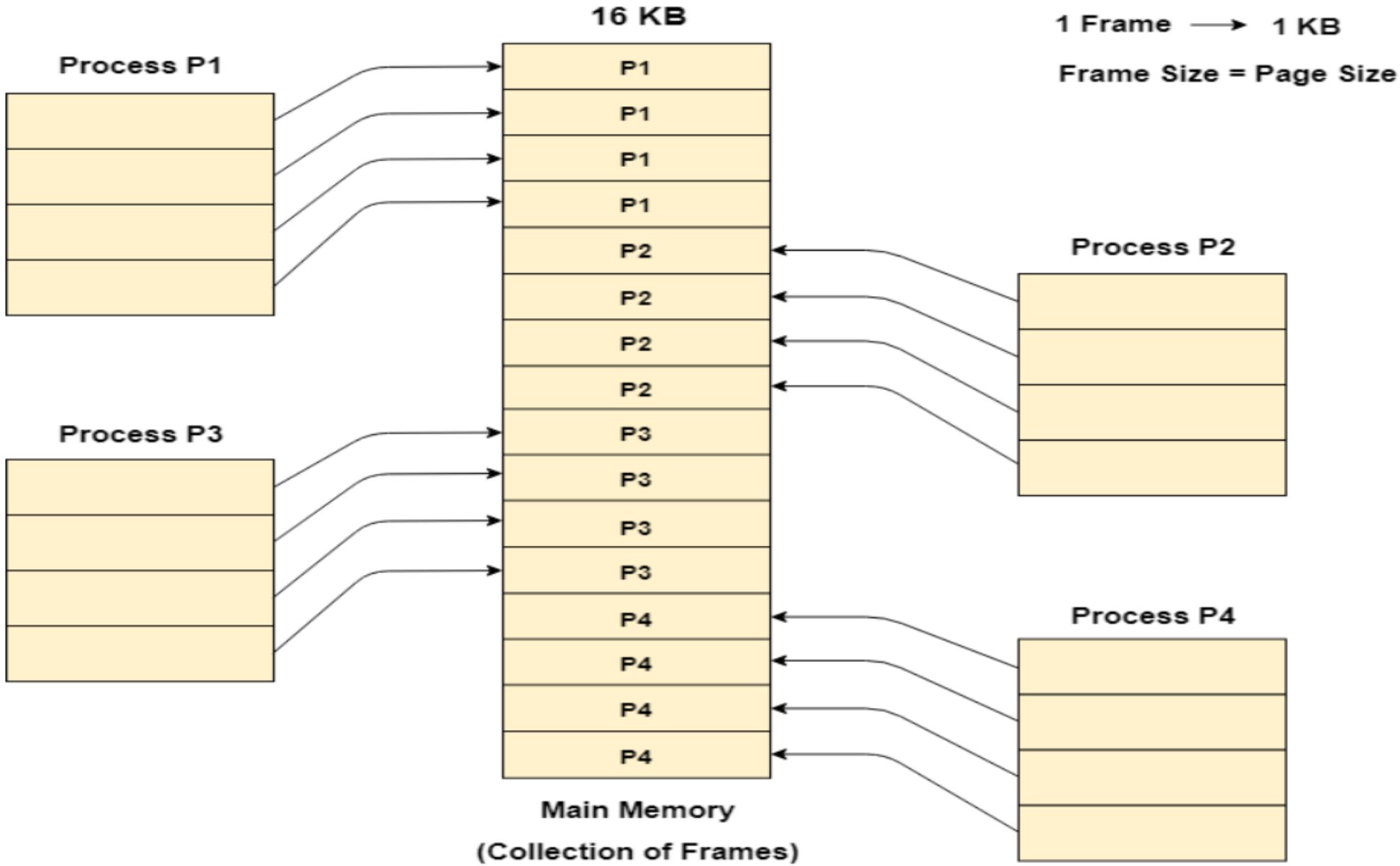

Operating System
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 6
Frame 7
Frame 8
Frame 9
Main Memory
(Collection of Frames)
Mapping
Page 1
Page 2
Page 3
Page 4
Page 5
Page 6
Page 7
Page 8
Page 9
Pages
Process

Example:

- Let us consider the main memory size 16 Kb and Frame size is 1 KB therefore the main memory will be divided into the collection of 16 frames of 1 KB each.

- There are 4 processes in the system that is P1, P2, P3 and P4 of 4 KB each. Each process is divided into pages of 1 KB each so that one page can be stored in one frame.

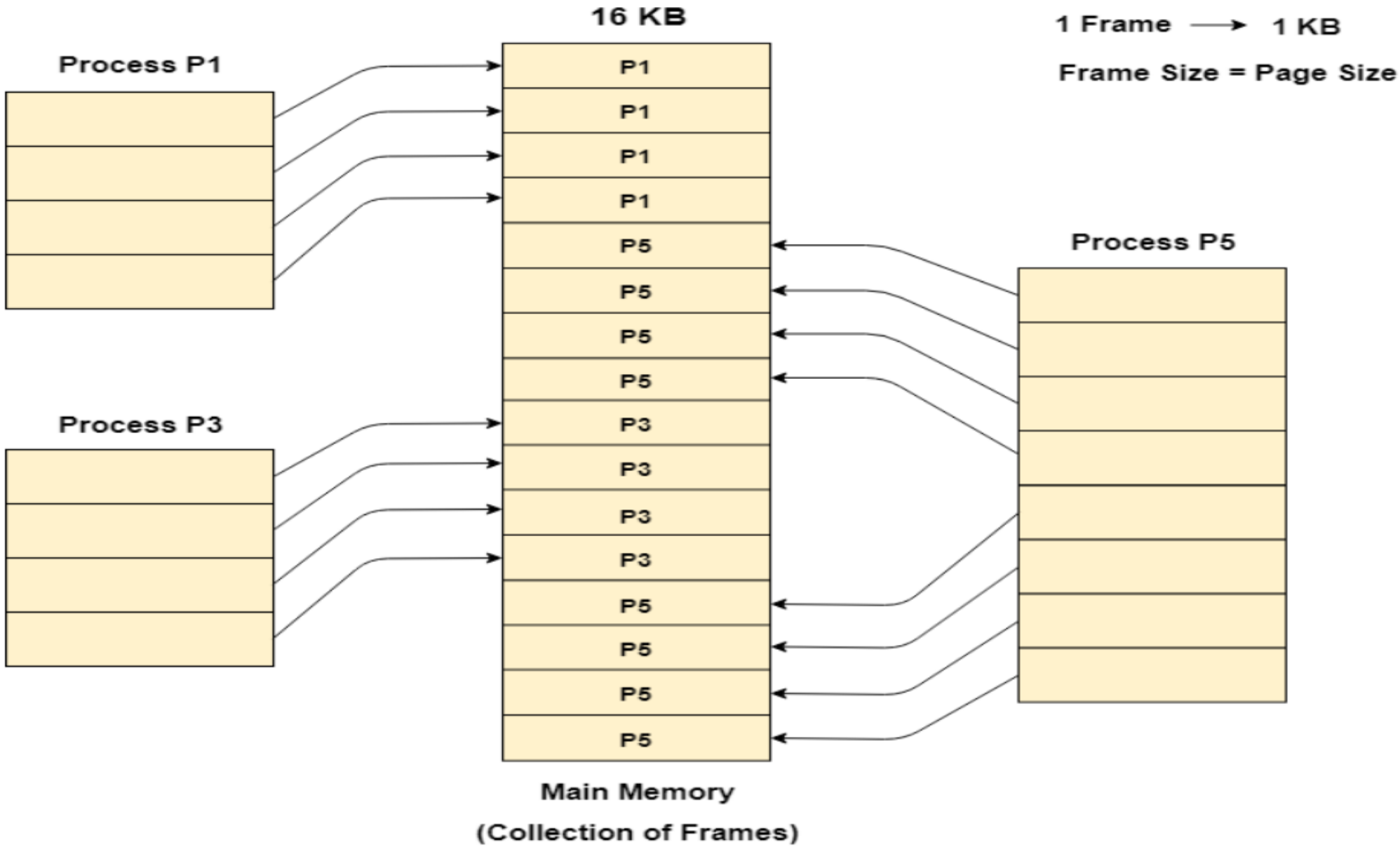
- Initially, all the frames are empty therefore pages of the processes will get stored in the contiguous way.


16 KB
1 Frame → 1 KB
Frame Size = Page Size
Process P1
Process P2
Process P3
Process P4
P1
P1
P1
P1
P2
P2
P2
P2
P3
P3
P3
P3
P4
P4
P4
P4
Main Memory
(Collection of Frames)

Example:

- Let us consider that, P2 and P4 are moved to waiting state after some time. Now, 8 frames become empty and therefore other pages can be loaded in that empty place. The process P5 of size 8 KB (8 pages) is waiting inside the ready queue.

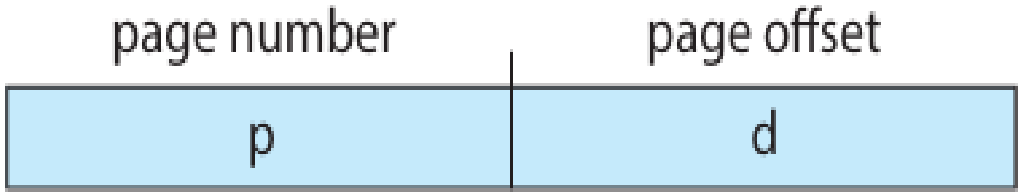
- Given the fact that, we have 8 non contiguous frames available in the memory and paging provides the flexibility of storing the process at the different places. Therefore, we can load the pages of process P5 in the place of P2 and P4.


16 KB
1 Frame → 1 KB
Frame Size = Page Size
Process P1
Process P3
Process P5
P1
P1
P1
P1
P5
P5
P5
P5
P3
P3
P3
P3
P5
P5
P5
P5
Main Memory
(Collection of Frames)

Paging

- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called **pages**.

Every address generated by the CPU is divided into two parts: a **page number** (p) and a **page offset** (d):



p - Used as index into a page tables

d - The displacement within the page

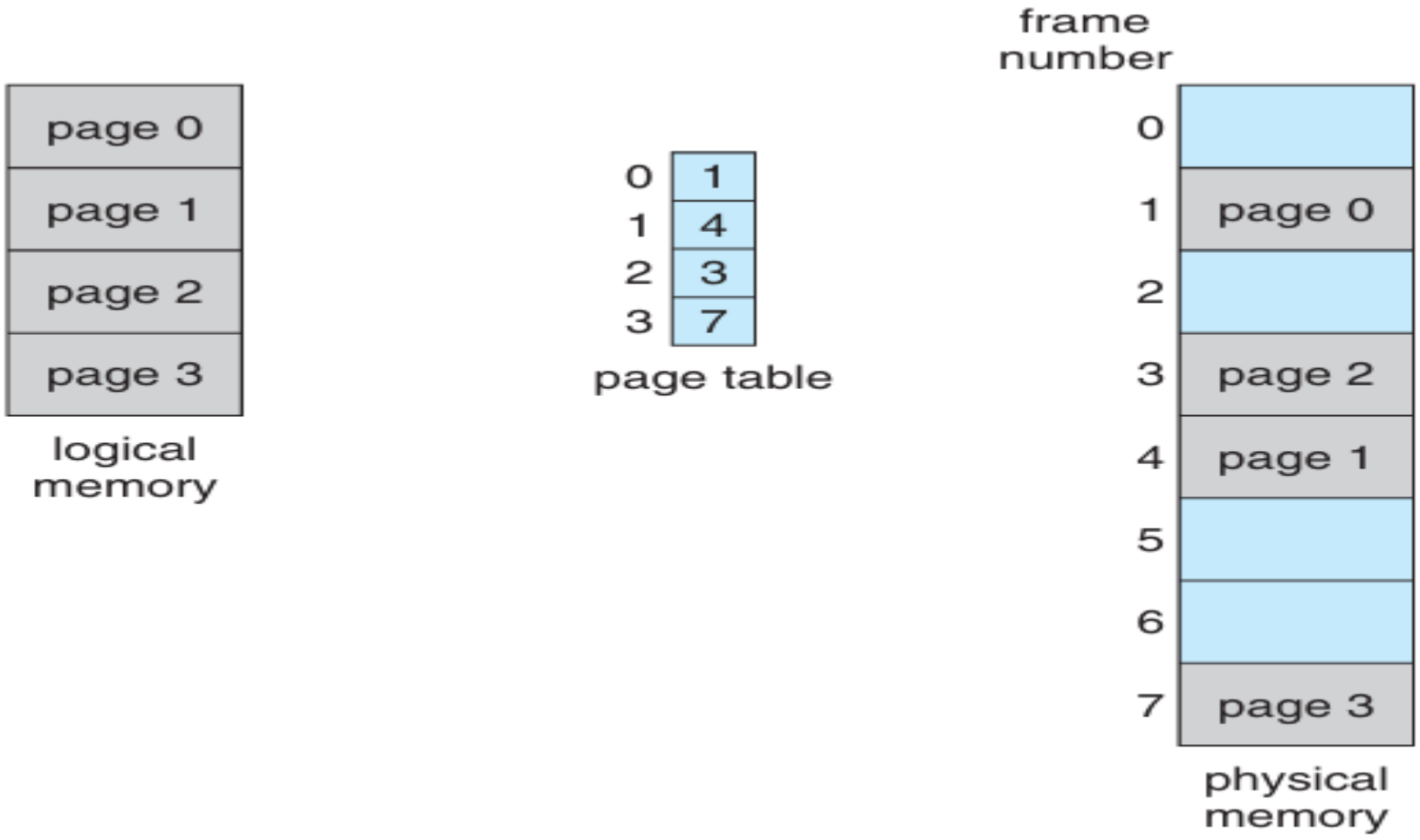
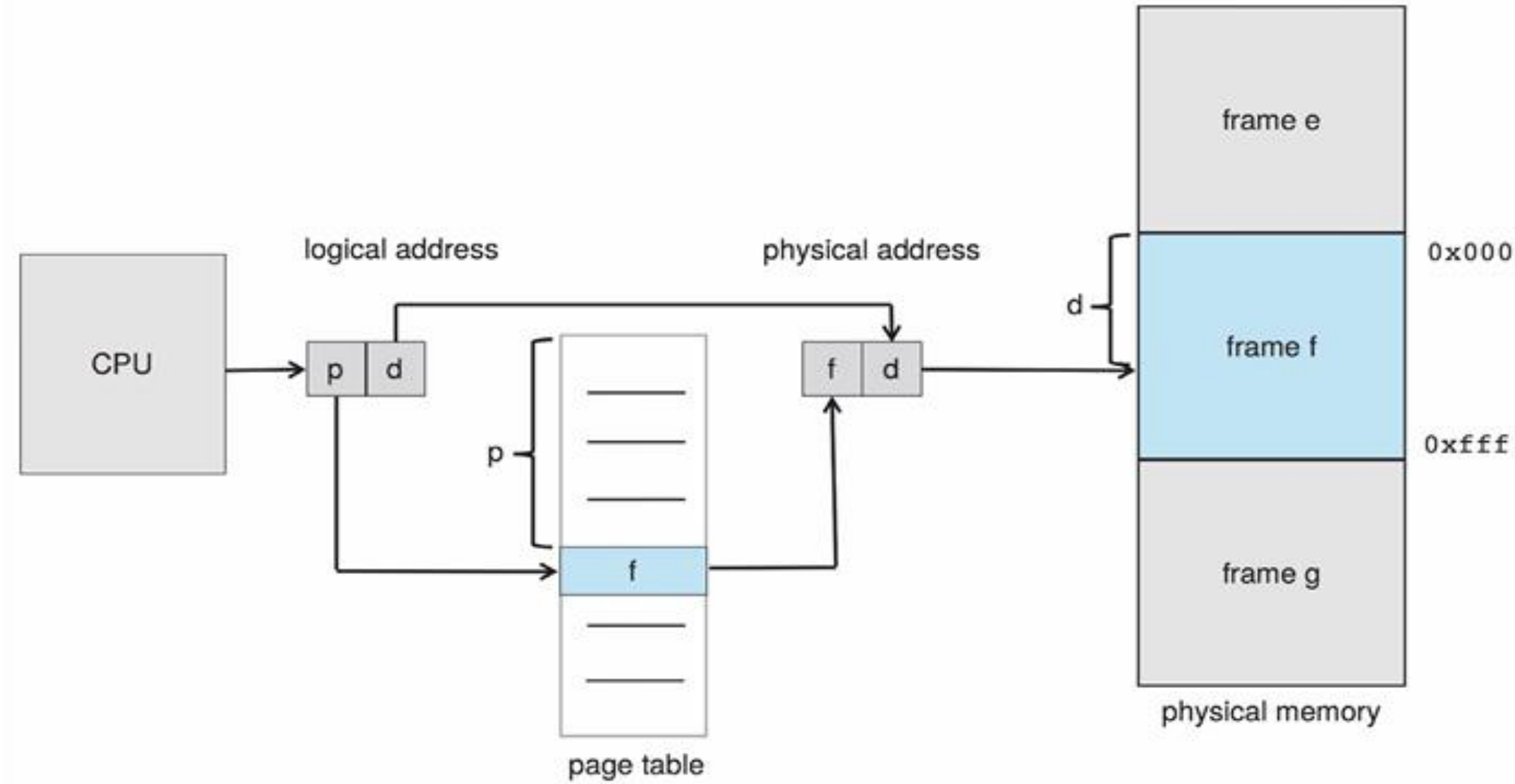


Fig. Paging model of logical and physical memory.

- The page table contains the base address of each frame in physical memory.
- The base address is combined with the page offset to define the physical memory address that is sent to the memory unit.


CPU
logical address
p
d
p
f
page table
physical address
f
d
d
frame e
frame f
frame g
0x000
0xfff
physical memory

How Paging Works:

- **Divide Logical Memory (Virtual Memory) into Pages**:
  - ✓ Logical memory is divided into equal-sized pages (e.g., 4 KB, 8 KB).

- **Divide Physical Memory into Frames**:
  - ✓ Physical memory is also divided into frames of the same size as the pages. For example, if a page is 4 KB, then each frame in the physical memory will also be 4 KB.

- **Page Table**:
  - ✓ The operating system maintains a ***page table*** for each process. The page table holds the mapping of virtual pages to physical frames.
  - ✓ For each page in virtual memory, the page table contains the frame number in physical memory where the page is stored.

- **Address Translation**:
  - ✓ When a process needs to access data, it provides a ***virtual address***. The operating system uses the page table to translate this virtual address into a ***physical address***.
    - ❖ The virtual address is divided into two parts: a ***page number*** and an ***offset*** within the page.
    - ❖ The page number is used to find the corresponding frame number in the page table, and the offset gives the exact location within that frame.

- **Loading Pages into Memory**:
  - ✓ If the required page is not in memory (a situation known as a ***page fault***), the operating system will load the page from disk into an available frame in physical memory.

- Modern CPUs have hardware support for paging. The hardware is typically responsible for translating virtual addresses to physical addresses.

- **Memory Management Unit (MMU)**: This component in the CPU performs address translation using the page table. When the CPU generates a virtual address, the MMU uses the page table to translate it into the corresponding physical address.

Translation of Logical Address to Physical Address using Page Table:

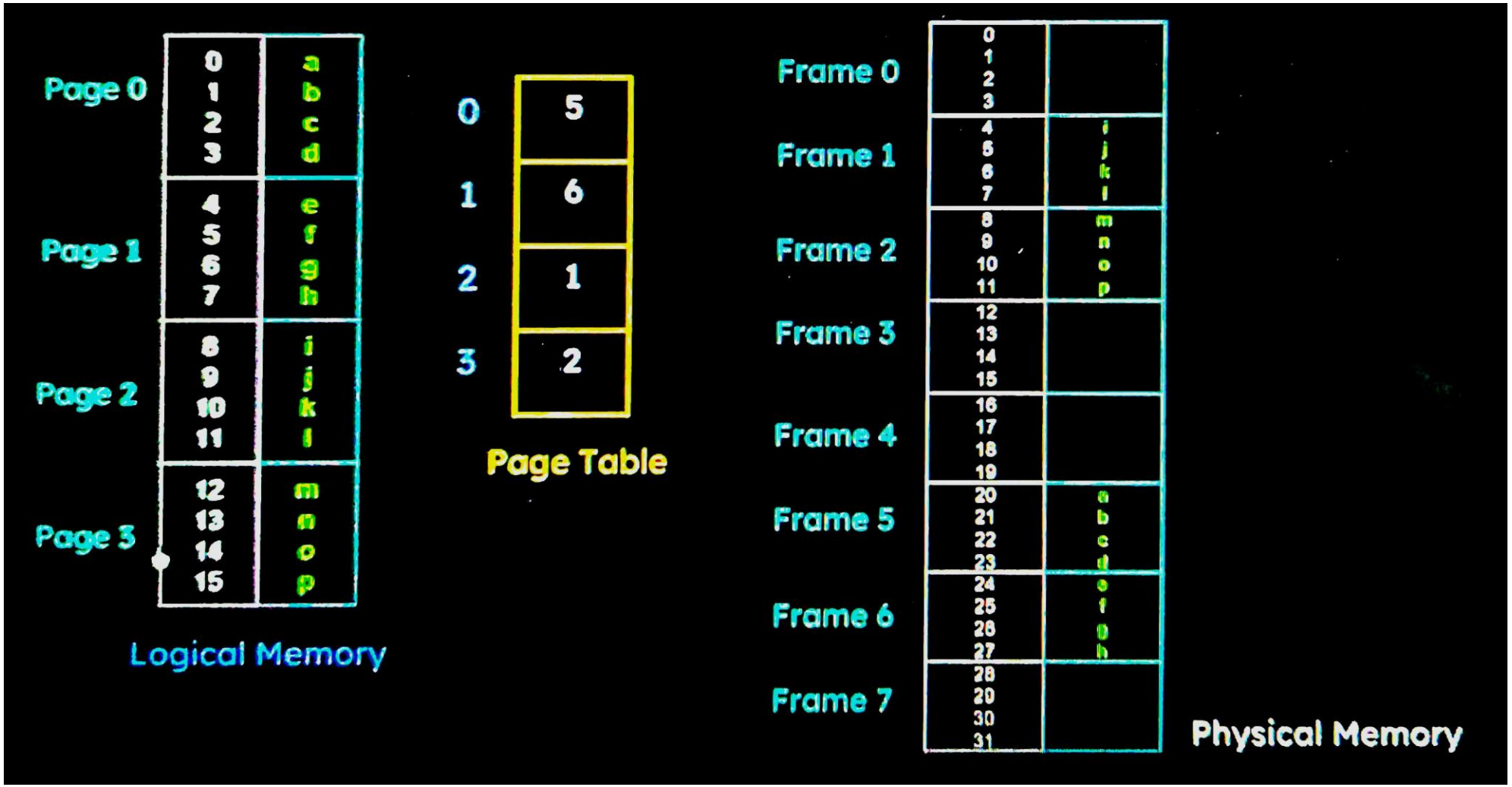
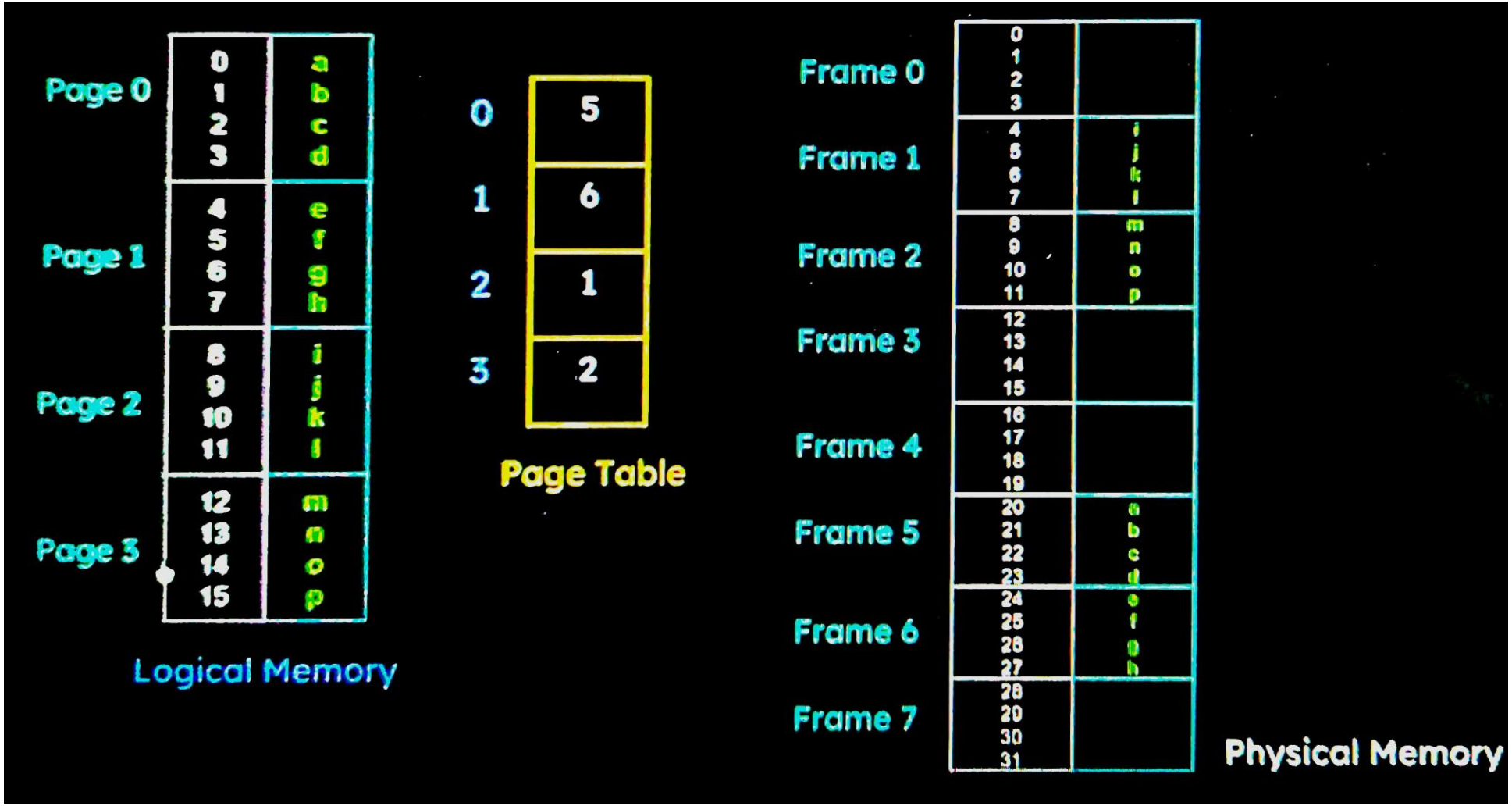
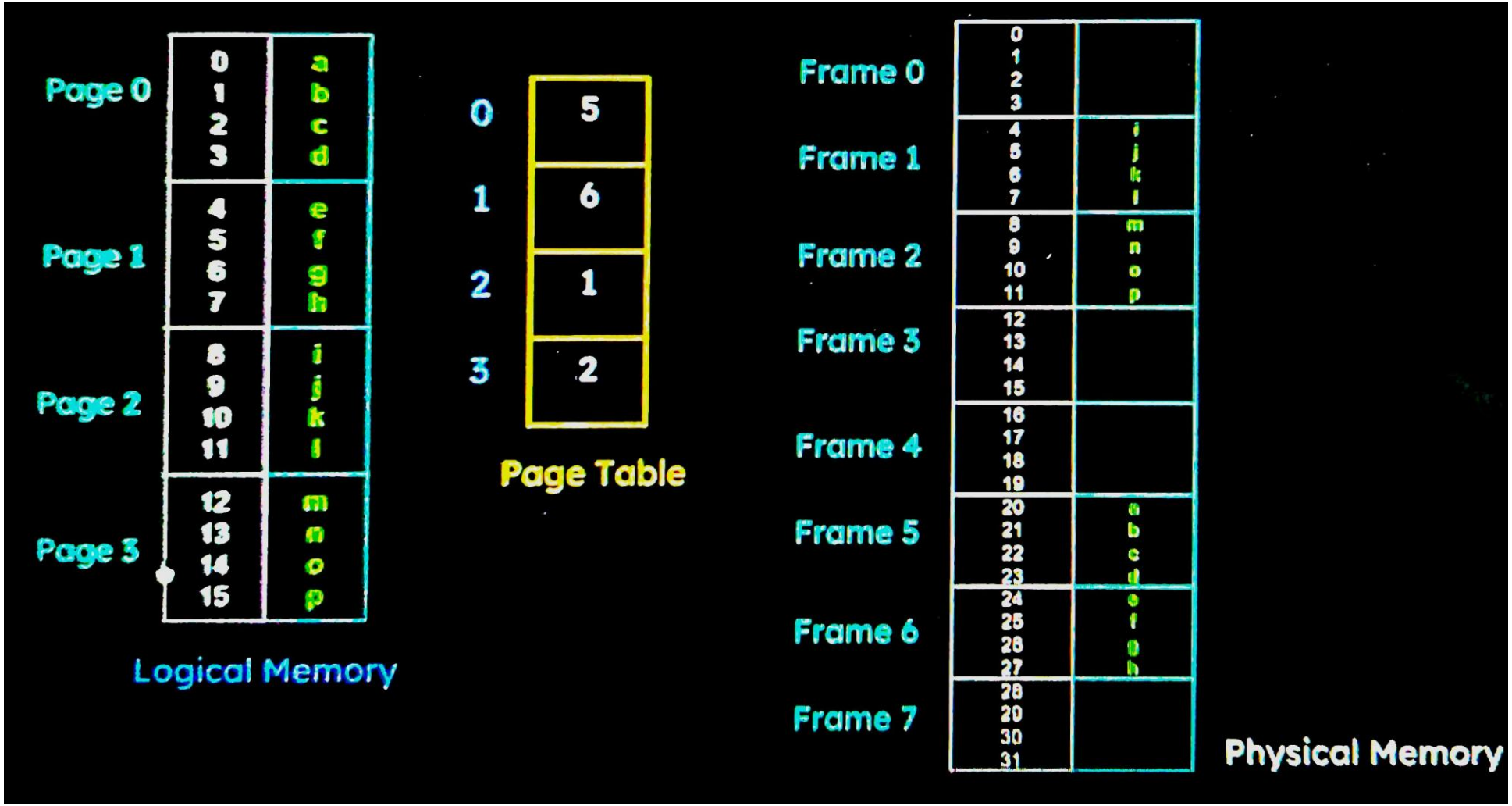


Fig. Paging example for a 32-byte memory with 4-byte pages.



Logical Address 0 maps to Physical Address 20

((Frame x Page size) + Offset) → **((5 X 4) + 0) = 20**



Logical Address 4 maps to Physical Address 24

((Frame x Page size) + Offset) → **((6 X 4) + 0) = 24**

Question: Consider a system which has:
Logical Address (LA) = 7 bits, Physical Address (PA) = 6 bits, Page Size = 8 words/bytes.
Calculate number of pages and number of frames

1. Number of Pages:

The logical address space is determined by the number of bits in the logical address. Since we have a 7-bit logical address, the total number of addressable words in the logical address space is:

$$\text{Total logical address space} = 2^{\text{LA}} = 2^7 = 128 \text{ words}$$

The page size is 8 words, so the number of pages is:

$$\text{Number of Pages} = \frac{\text{Total logical address space}}{\text{Page size}} = \frac{128 \text{ words}}{8 \text{ words/page}} = 16 \text{ pages}$$

2. Number of Frames:

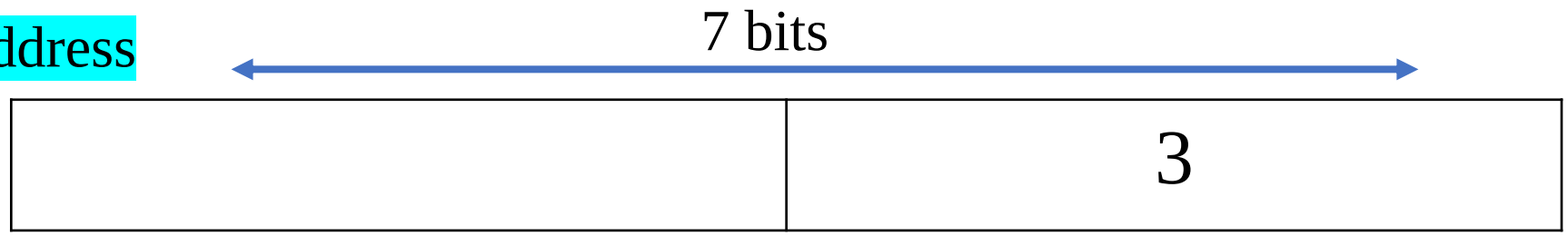
The physical address space is determined by the number of bits in the physical address. Since we have a 6-bit physical address, the total number of addressable words in the physical address space is:

$$\text{Total physical address space} = 2^{\text{PA}} = 2^6 = 64 \text{ words}$$

The frame size is the same as the page size (8 words), so the number of frames is:

$$\text{Number of Frames} = \frac{\text{Total physical address space}}{\text{Frame size}} = \frac{64 \text{ words}}{8 \text{ words/frame}} = 8 \text{ frames}$$

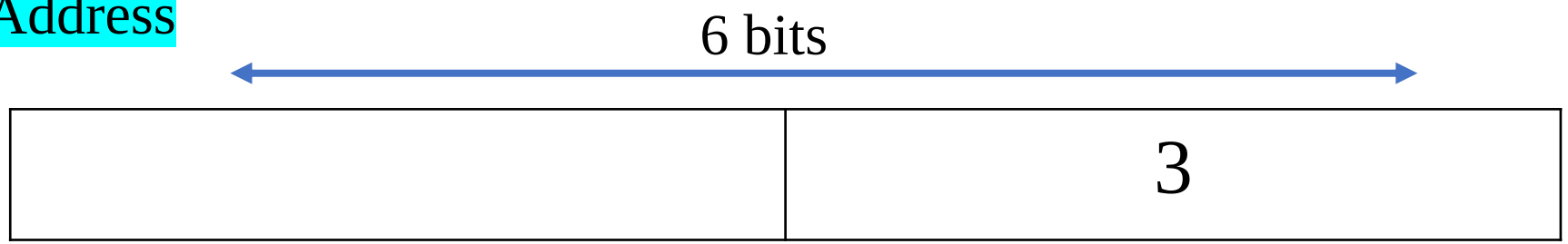
Logical Address



Page Number
(represented by 4 bits)

Page Offset/ Page size
(represented by 3 bits)

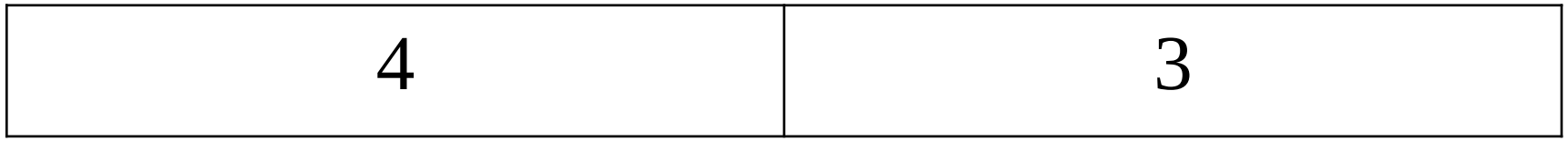
Physical Address



Frame Number
(represented by 4 bits)

Page Offset/ Page size
(represented by 3 bits)

Logical Address



Page Number
(represented by 4 bits)

Page Offset/ Page size
(represented by 3 bits)

Physical Address

Frame Number
(represented by 4 bits)

Frame Offset/ Frame size
(represented by 3 bits)

What will be the number of entries in the Page Table ?

What will be the number of entries in the Page Table ?

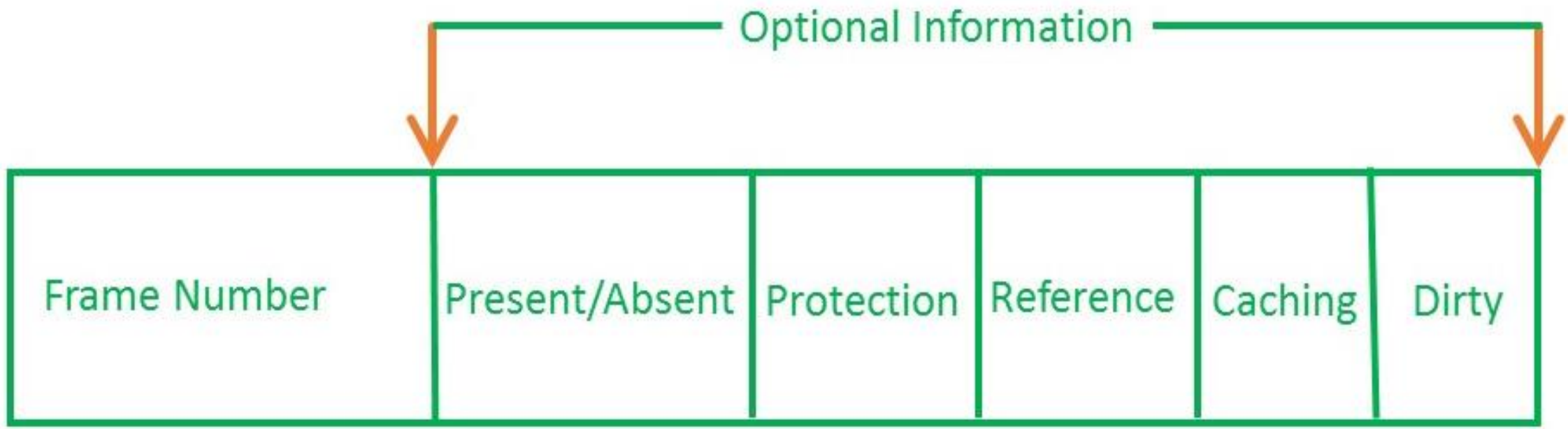
It will be equal to the number of pages in the process

Page Table Entries in Page Table

- A Page Table is a data structure used by the operating system to keep track of the mapping between virtual addresses used by a process and the corresponding physical addresses in the system's memory.

- A Page Table Entry (PTE) is an entry in the Page Table that stores information about a particular page of memory. Each PTE contains information such as the physical address of the page in memory, whether the page is present in memory or not, whether it is writable or not, and other access permissions.

- The size and format of a PTE can vary depending on the architecture of the system and the operating system used. In general, a PTE contains enough information to allow the operating system to manage memory efficiently and protect the system from malicious or accidental access to memory.



PAGE TABLE ENTRY

- **Frame Number –** It gives the frame number in which the current page you are looking for is present. The number of bits required depends on the number of frames. Frame bit is also known as address translation bit.

Number of bits for frame = Size of physical memory / Frame Size

- **Present/Absent Bit:** Present or absent bit says whether a particular page you are looking for is present or absent. In case it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control page faults by the operating system to support virtual memory. Sometimes this bit is also known as a **valid/invalid** bit.

- **Protection Bit:** The protection bit says what kind of protection you want on that page. So, these bits are for the protection of the page frame (read, write, etc).

- **Referenced Bit:** Referenced bit will say whether this page has been referred to in the last clock cycle or not. It is set to 1 by hardware when the page is accessed.

- **Caching Enabled/Disabled:**
  - Sometimes we need fresh data. Let us say the user is typing some information from the keyboard and your program should run according to the input given by the user. In that case, the information will come into the main memory. Therefore main memory contains the latest information which is typed by the user.
  - Now if you try to put that page in the cache, that cache will show the old information. So whenever freshness is required, we don't want to go for caching or many levels of memory. The information present in the closest level to the CPU and the information present in the closest level to the user might be different. So we want the information to be consistent, which means whatever information the user has given, the CPU should be able to see it as first as possible.
  - That is the reason we want to disable caching. So, this bit **enables or disables** caching of the page.

➢ Modified Bit/Dirty Bit:

- **Indicates Modification**: The dirty bit is set (usually to 1) when a page in memory is modified (i.e., a write operation occurs on that page).
- **Efficient Page Replacement**: If a page is replaced (e.g., when a page is swapped out to disk), the dirty bit helps the operating system determine whether the page needs to be written back to disk.
  - **If the dirty bit is 1**: It means the page has been modified, and it must be written back to disk when swapped out.
  - **If the dirty bit is 0**: The page has not been modified, so there is no need to write it back to disk when swapping it out.

This allows the operating system to optimize disk I/O, as it avoids writing unchanged pages to disk, which improves system performance

Hardware Implementation of Page Table

- In Operating System (Memory Management Technique: Paging), for each process page table will be created, which will contain a Page Table Entry (PTE).

- This PTE will contain information like frame number (The address of the main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit, etc). This page table entry (PTE) will tell where in the main memory the actual page is residing.

- Now the question is where to place the page table, such that overall access time (or reference time) will be less. The problem initially was to fast access the main memory content based on the address generated by the CPU (i.e. logical/virtual address). Initially, some people thought of using registers to store page tables, as they are high-speed memory so access time will be less.

Hardware Implementation of Page Table

- The idea used here is, to place the page table entries in registers, for each request generated from the CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides.

    - Everything seems right here, but the problem is registered size is small (in practice, it can accommodate a maximum of 0.5k to 1k page table entries) and the process size may be big hence the required page table will also be big (let's say this page table contains 1M entries), so registers may not hold all the PTE's of the Page table. So, this is not a practical approach.

- The entire page table was kept in the main memory to overcome this size issue. But the problem here is two main memory references are required:

    - To find the frame number
    - To go to the address specified by frame number

To overcome this problem a high-speed cache is set up for page table entries called a **Translation Lookaside Buffer (TLB).**

Translation Lookaside Buffer (TLB) in Paging

- Translation Lookaside Buffer (TLB) is a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used.

- Given a virtual address, the processor examines the TLB if a page table entry is present (**TLB hit**), the frame number is retrieved and the real address is formed.

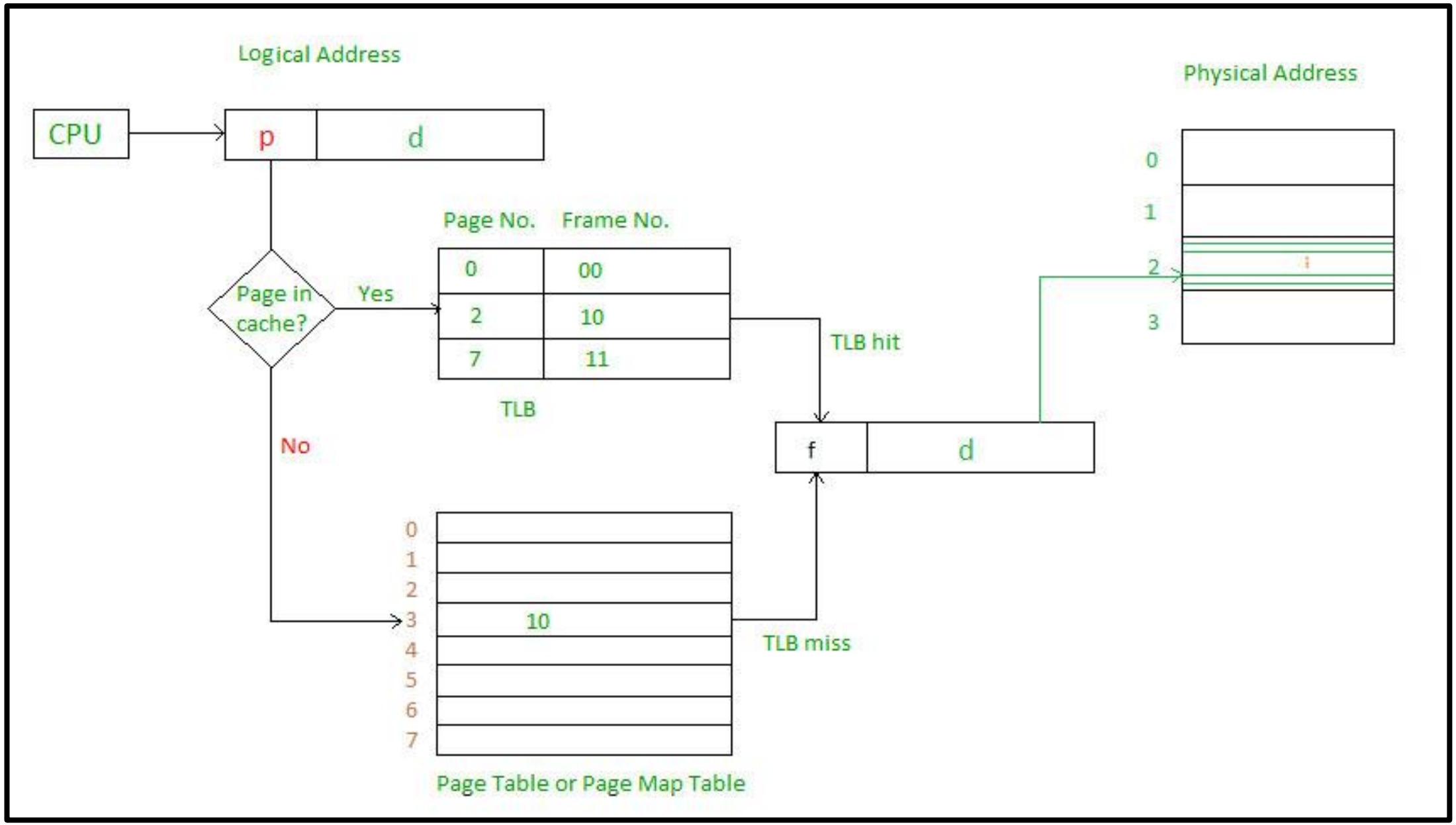
- TLB hit is a condition where the desired entry is found in translation look aside buffer. If this happens then the CPU simply access the actual location in the main memory.

- However, if the entry is not found in TLB (**TLB miss**) then CPU has to access page table in the main memory and then access the actual frame in the main memory.

Translation Lookaside Buffer (TLB) in Paging (Contd.)

- If the CPU cannot find the page table entry in the TLB (a miss), it checks the page table. If the page is **not in memory** at all (this is a page fault), the system has to retrieve the page from disk.
- **Page Fault Handling**: If the page table entry indicates that the page is not in memory (due to a page fault), the OS will invoke the page fault handler to bring the page into memory. This may require:
  - Swapping out another page.
  - Updating the page table with the new location.
  - Loading the page from secondary storage (disk) into a free frame in physical memory.

- **TLB Update After Page Fault**: Once the page is loaded into memory and the page table is updated, the TLB may be updated as well to avoid future page faults for this page. A subsequent access to the same page will likely result in a **TLB hit** if it was properly cached.


Logical Address
Physical Address
CPU
p
d
Page No.
Frame No.
Page in cache?
Yes
No
0
00
2
10
7
11
TLB
TLB hit
f
d
0
1
2
3
10
4
5
6
7
TLB miss
Page Table or Page Map Table
0
1
2
3

Steps in TLB hit

1.CPU generates a virtual (logical) address.

2.It is checked in TLB (present).

3.The corresponding frame number is retrieved, which now tells where the main memory page lies.

Translation Lookaside Buffer (TLB) in Paging (Contd.)

Steps in TLB miss

1.CPU generates a virtual (logical) address.

2.It is checked in TLB (not present).

3.Now the page number is matched to the page table residing in the main memory (assuming the page table contains all PTE).

4.The corresponding frame number is retrieved, which now tells where the main memory page lies.

5.The TLB is updated with new PTE (if space is not there, one of the replacement techniques comes into the picture i.e either FIFO, LRU or MFU etc)

Effective memory access time (EMAT)

= TLB hit (TLB time + Main memory time) + TLB miss (TLB time + 2 × Main memory time)

Note:

- In the above EMAT, we are assuming no page fault scenario.
- If page fault, occurs extra Page Fault Service Time has to be included.

Question:

Consider a paging hardware with a TLB. Assume that the entire page table and all the pages are in the physical memory. It takes 10 milliseconds to search the TLB and 80 milliseconds to access the physical memory. If the TLB hit ratio is 0.6, the effective memory access time (in milliseconds) is _____.

A. 120

B. 122

C. 124

D. 118

Given,

> TLB hit ratio = 0.6
>
> Therefore, TLB miss ratio = 0.4
>
> Time taken to access TLB (t) = 10 ms
>
> Time taken to access main memory (m) = 80 ms

Effective Access Time (EAT) = 0.6 (10 + 80) + 0.4 (10 + 80 + 80) = 90 X 0.6 + 0.4 X 170 = 122

Hence, the right answer is option B.

References

1. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, “Operating System Concepts,” Eleventh Edition (Willey).
2. Andrew S. Tanenbaum, “Modern Operating Systems”, Fourth Edition (Pearson Publications), 2014.
3. https://www.geeksforgeeks.org/
4. https://www.javatpoint.com/
5. https://www.tutorialspoint.com/
6. https://www.nesoacademy.org/
7. https://www.baeldung.com/
8. https://www.tpointtech.com/
9. https://www.educative.io/
10. https://prepinsta.com