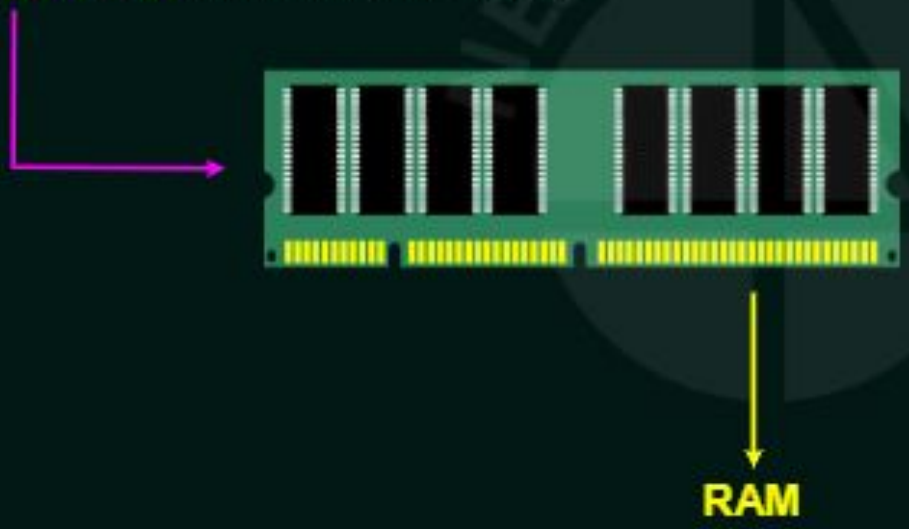
Memory management

Memory Management

- The main purpose of a computer system is to execute programs.
- During execution, these programs, together with the data they access, must be in main memory (at least partially).



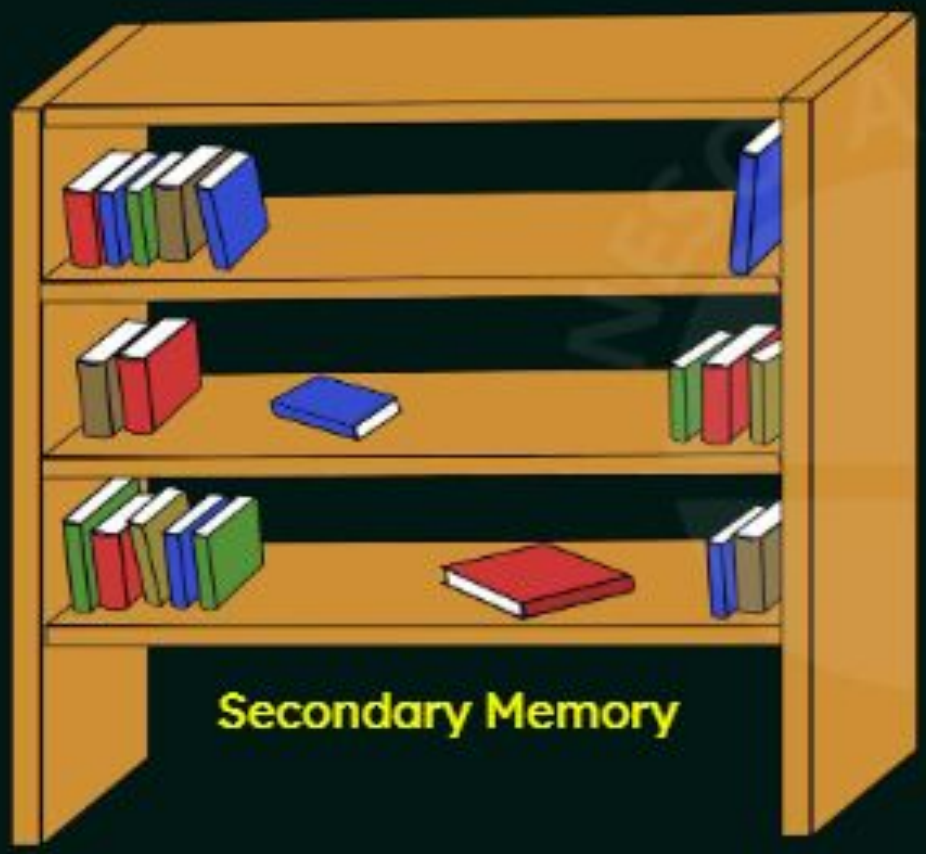
Main Memory

➔ In CPU scheduling we showed how the CPU can be shared by a set of processes.

as a result of this

CPU Utilization, as well as speed of the computer's response to its users could be improved.

To realize this increase in performance, however, we must keep several processes in memory- that is, we must share memory.


Secondary Memory


Main Memory

Topics to be discussed:

- ★ Various ways to manage memory.
- ★ Memory management algorithms:
  - Primitive bare-machine approach.
  - Paging strategies.
  - Segmentation strategies.

Main Memory

Basic Hardware

- Memory consists of a large array of words or bytes, each with its own address.
- The CPU fetches instructions from memory according to the value of the program counter.
- These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, first fetches an instruction from memory.

The instruction is then decoded and may cause operands to be fetched from memory.

After the instruction has been executed on the operands, results may be stored back in memory.

Basic Hardware

The CPU can directly access only
- Registers built into the processor
- Main Memory

- There are machine instructions that take memory addresses as arguments, but none that take disk addresses.
- So, any instructions in execution, and any data being used by the instructions, must be in one of these direct-access storage devices.
- If the data are not in memory, they must be moved there before the CPU can operate on them.

CPU Cycle Time for accessing memories

Registers - Accessible within one cycle of the CPU clock.
Most CPUs can decode instructions and perform simple operations on register contents at the rate of one or more operations per clock tick.

Main Memory - Access may take many cycles of the CPU clock to complete.
In this case, the processor normally needs to stall, since it does not have the data required to complete the instruction that it is executing.

This situation is intolerable because of the frequency of memory accesses.

Remedy - Add fast memory between the CPU and main memory.
A memory buffer used to accommodate a speed differential, called a cache.

Protection of OS from unauthorized access

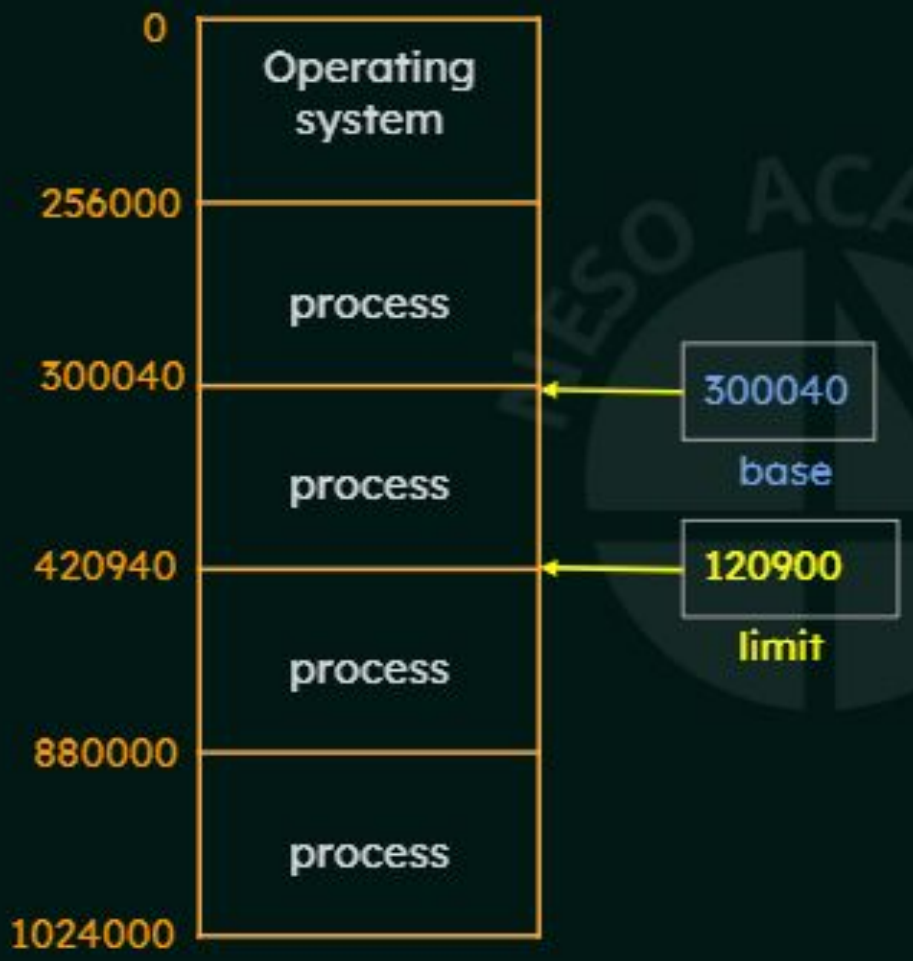
- The Operating System has to be protected from access by user processes.
- In addition, user processes must be protected from one another.
- This protection must be provided by the hardware.

How can this be done ?

Ensure that each process has a separate memory space.
To do this, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.

For this we use two registers - **BASE** and **LIMIT**

BASE and LIMIT

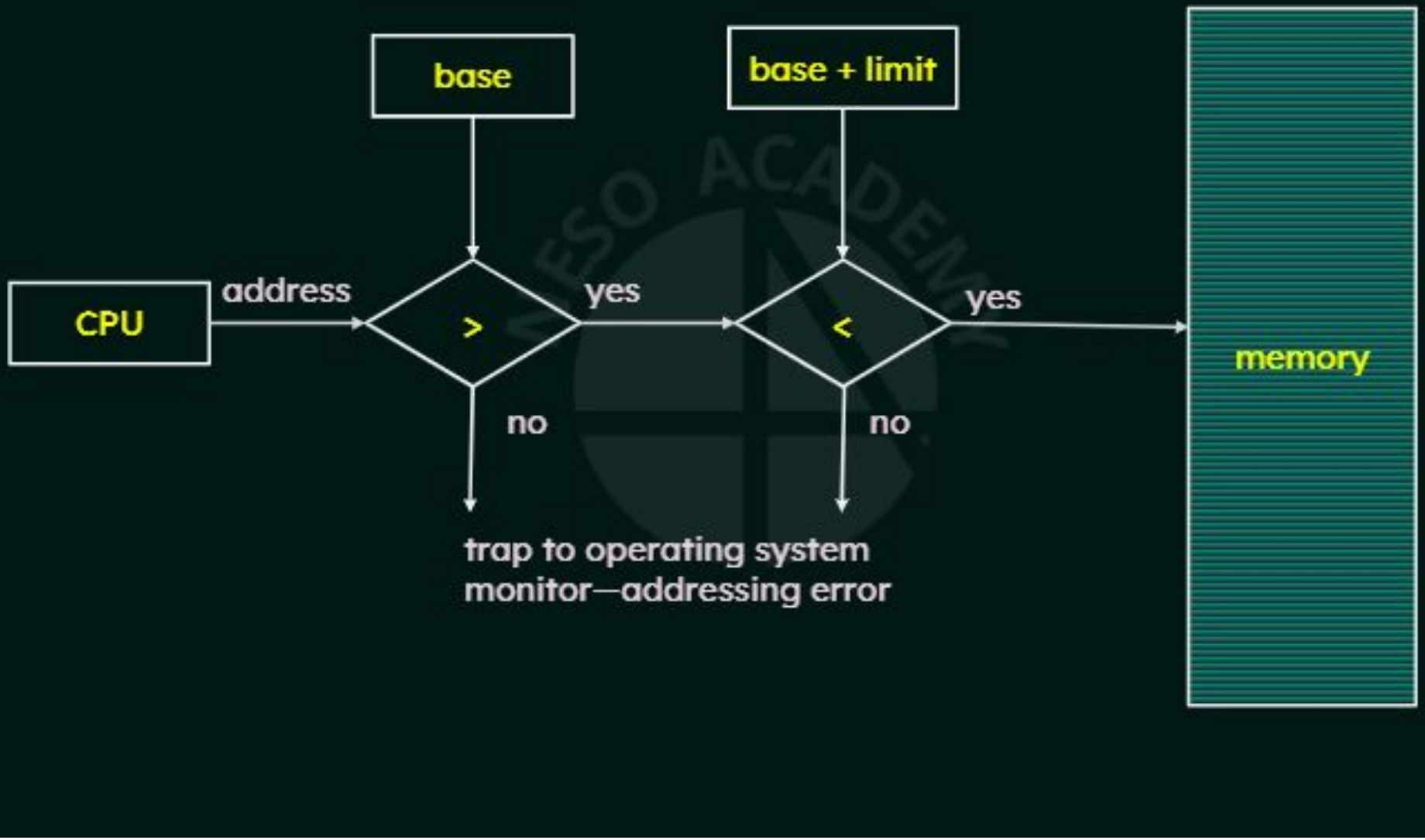


A base and a limit register define a logical address space.

The base register holds the smallest legal physical memory address.

The limit register specifies the size of the range.

For example, if the base register holds 300040 and limit register is 120900, then the program can legally access all addresses from 300040 through 420940 (inclusive).


base
base + limit
CPU
address
>
yes
<
yes
memory
no
no
trap to operating system
monitor—addressing error

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.

Since privileged instructions can be executed only in kernel mode, and since only the operating system executes in kernel mode, only the operating system can load the base and limit registers.

This scheme allows the operating system to change the value of the registers but prevents user programs from changing the registers' contents.

Address Binding

- Usually, a program resides on a disk as a binary executable file.
- To be executed, the program must be brought into memory and placed within a process.
- Depending on the memory management in use, the process may be moved between disk and memory during its execution.
- The processes on the disk that are waiting to be brought into memory for execution form the input queue.

Input Queue → Select a process → Loads it into memory → Executes and accesses instructions and data from memory → Process terminates → Memory space is declared available

Most systems allow a user process to reside in any part of the physical memory.

Though address space of the computer starts at 00000 the first address of the user process need not be 00000.

In most cases, a user program will go through several steps during
COMPILE TIME, LOAD TIME, EXECUTION TIME
before being executed.

Addresses may be represented in different ways during these steps.

Source Program - Addresses are generally symbolic (such as *count*).

Compiler - Typically binds these symbolic addresses to relocatable addresses (such as "*14 bytes from the beginning of this module*")

Linkage editor or Loader - Binds the relocatable addresses to absolute addresses (such as *74014*)

Each binding is a mapping from one address space to another.

Binding of instructions and data to memory addresses during

COMPILE TIME

If we know at compile time where the process will reside in memory, then absolute code can be generated.

For example:
If we know that a user process will reside starting at location R, then the generated compiler code will start at that location and extend up from there.

If, at some later time, the starting location changes, then it will be necessary to recompile this code.

Binding of instructions and data to memory addresses during

LOAD TIME

If it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code.

In this case, final binding is delayed until load time. If the starting address changes, we need only reload the user code to incorporate this changed value.

Binding of instructions and data to memory addresses during

EXECUTION TIME

If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

Logical Versus Physical Address Space

Logical Address - An address generated by the CPU.

Physical Address - An address seen by the memory unit—that is, the one loaded into the memory-address register of the memory.

Compile-time and Load-time Address-Binding Methods generate identical Logical and Physical addresses.

Execution-time Address Binding scheme results in differing Logical and Physical addresses.

In this case, we usually refer to the Logical address as a Virtual address.

Logical Address Space - The set of all logical addresses generated by a program.

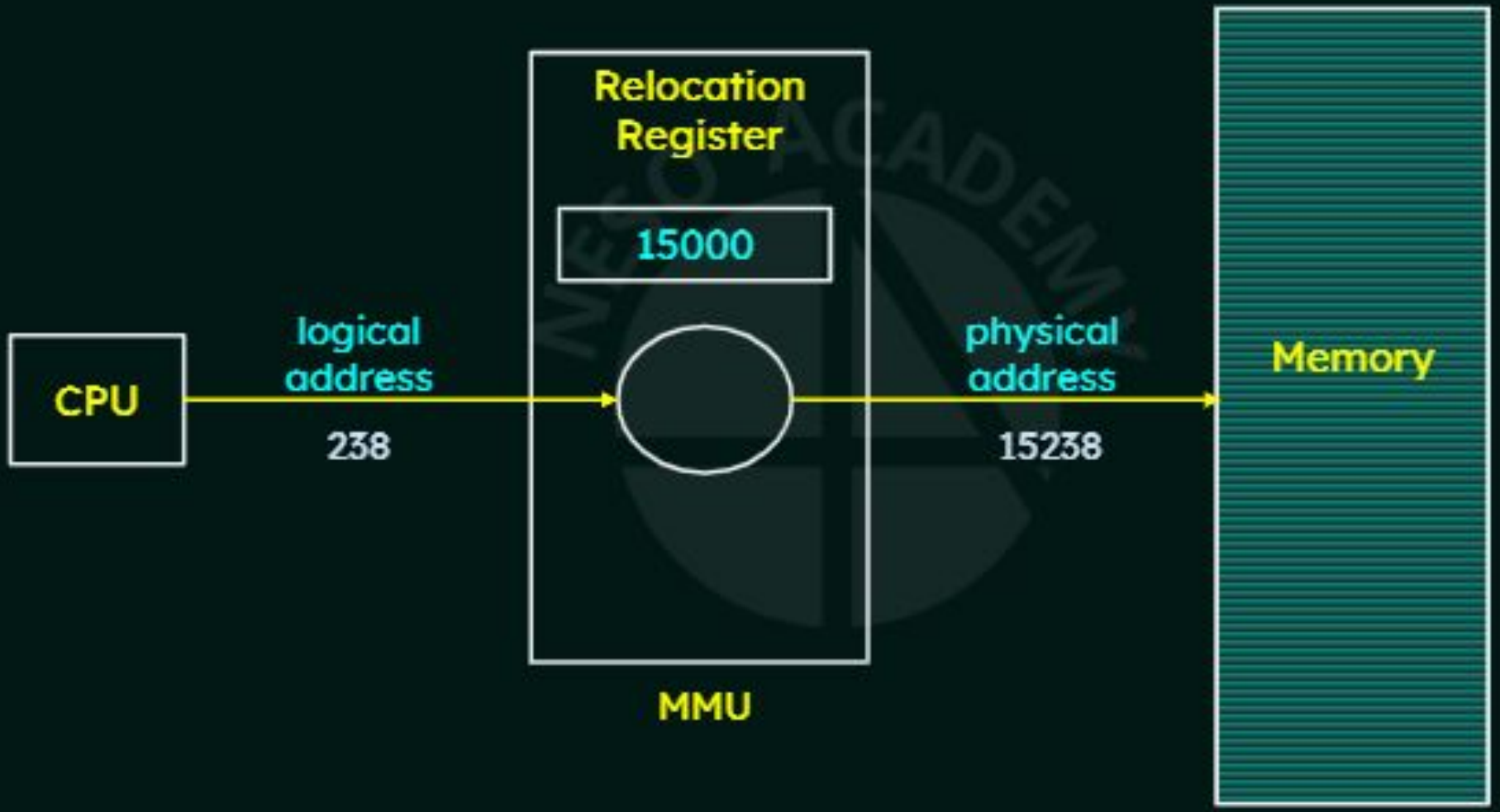
Physical Address Space - The set of all physical addresses corresponding to these logical addresses.

In the execution-time address-binding scheme, the logical and physical address spaces differ.

The run-time mapping from virtual to physical addresses is done by a hardware device called the

Memory-Management Unit (MMU).

Dynamic relocation using a relocation register



We now have two different types of addresses:

Logical addresses (in the range 0 to max) and

Physical addresses (in the range R + 0 to R + max for a base value R).

The user generates only logical addresses and thinks that the process runs in locations 0 to max.

The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used.

Dynamic Loading

- As we have discussed so far, the entire program and all data of a process must be in physical memory for the process to execute.
- The size of a process is thus limited to the size of physical memory.
- To obtain better memory-space utilization, we can use dynamic loading.

With dynamic loading, a routine is not loaded until it is called.

- All routines are kept on disk in a relocatable load format.
- The main program is loaded into memory and is executed.
- When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded.
- If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change.
- Then control is passed to the newly loaded routine.

Advantages:

- An unused routine is never loaded.
- This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases.
- Although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

Dynamic loading does not require special support from the operating system.

It is the responsibility of the users to design their programs to take advantage of such a method.

Dynamic Linking and Shared Libraries

When a programs runs, apart from its own modules, it also needs to use certains **System Libraries** as well.

Static Linking - System language libraries are treated like any other object module and are combined by the loader into the binary program image.

Disadvantage - Each program on a system must include a copy of its language library (or at least the routines referenced by the program) in the executable image. This requirement wastes both disk space and main memory.

Dynamic Linking - Checks to see whether the needed routine is already in memory. If not, the routine is loaded into into memory.

The concept of dynamic linking is similar to that of dynamic loading. But here, linking, rather than loading, is postponed until execution time.

Dynamically Linked Libraries

With dynamic linking, a **stub** is included in the image for each library routine reference.

A small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present.

- When the stub is executed, it checks to see whether the needed routine is already in memory.
- If not, the program loads the routine into memory.
- Either way, the stub replaces itself with the address of the routine and executes the routine.
- Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.

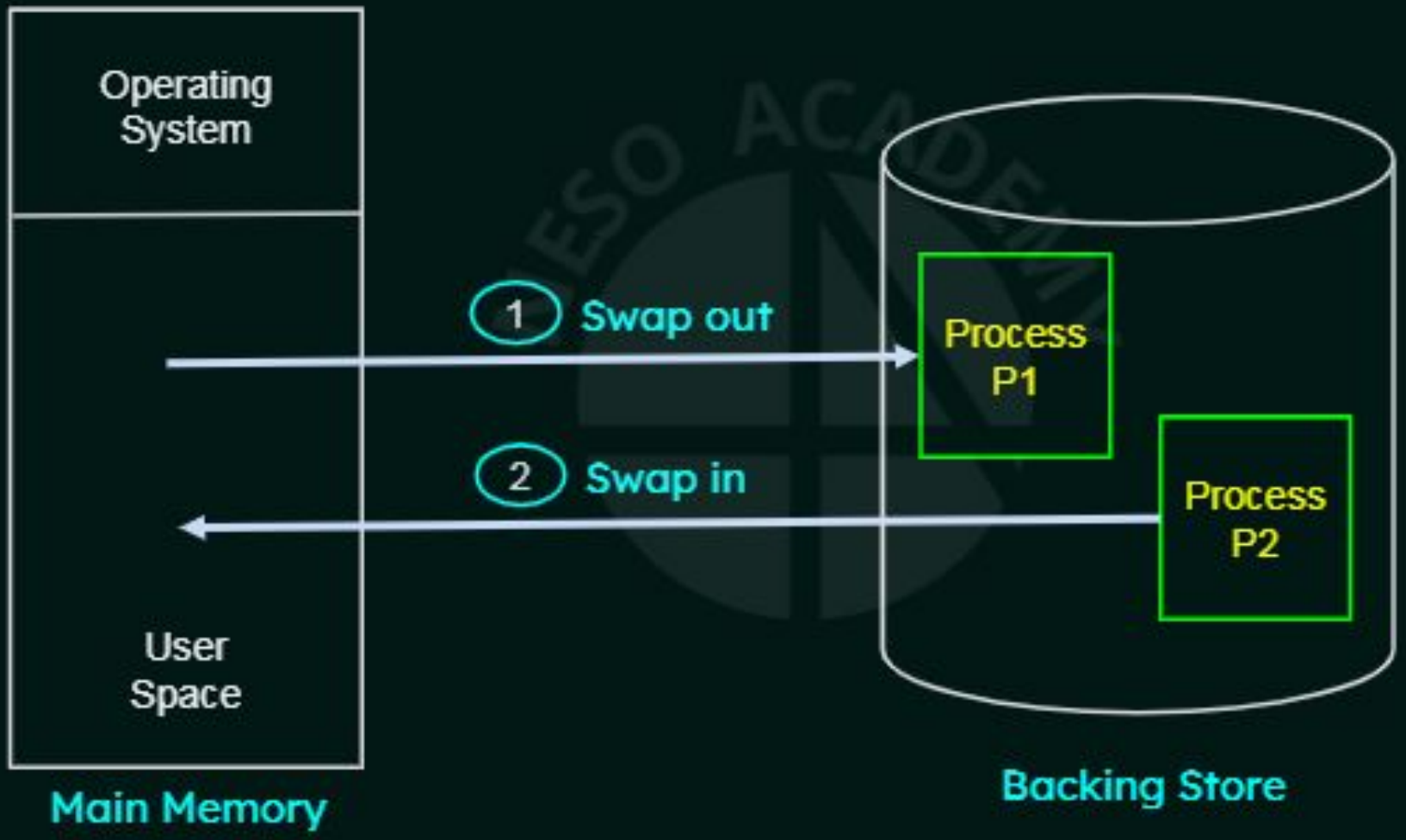
Swapping

- A process must be in memory to be executed.
- A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.

Example:
In a multiprogramming environment with a round-robin CPU-scheduling algorithm:

When a quantum expires, the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed.

Swapping of two processes using a disk as a backing store



Memory Space where Swapping happens

Normally:
A process that is swapped out will be swapped back into the same memory space it occupied previously.

But it depends on the address binding method.

If binding is done at assembly or load time: The process cannot be easily moved to a different location.

If binding is done at execution time: The process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Backing Store

Swapping requires a backing store.

→ Commonly a fast disk.

- It must be large enough to accommodate copies of all memory images for all users.
- It must provide direct access to these memory images.
- The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run.
- Whenever the CPU scheduler decides to execute a process, it calls the dispatcher.
- The dispatcher checks to see whether the next process in the queue is in memory.
- If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process.
- It then reloads registers and transfers control to the selected process.

Swap Time

The context-switch time in a swapping system is fairly high.

Example:
Let User Process Size = 10 MB
Transfer Rate of Backing Store = 40 MB per second

Actual transfer of the 10 MB process to or from main memory takes-

10000 KB/40000 KB per second
= 1/4 second
= 250 milliseconds

Assuming that no head seeks are necessary, and
assuming an average latency of 8 milliseconds, the swap time = 258 milliseconds.

Since we must both swap out and swap in, the total swap time is = 258 x 2
= 516 milliseconds.

Since we must both swap out and swap in, the total swap time is = 258 x 2
= 516 milliseconds.

- For efficient CPU utilization, we want the execution time for each process to be long relative to the swap time.
- Thus, in a round-robin CPU-scheduling algorithm, for example, the time quantum should be substantially larger than 0.516 seconds.
- The major part of the swap time is transfer time.
- The total transfer time is directly proportional to the amount of memory swapped.

Other factors that affect Swapping

- A process must be completely idle in order to be swapped.
- A process may be waiting for an I/O operation when we want to swap that process to free up memory.
- If the I/O is asynchronously accessing the user memory for I/O buffers, then the process cannot be swapped.

Memory Allocation

How to allocate memory ?

Divide memory into several fixed-sized partitions.

↓

Each partition may contain exactly one process.

↓

When a partition is free, a process is selected from the input queue and is loaded into the free partition.

↓

When the process terminates, the partition becomes available for another process.

This is one of the simplest methods for allocating memory

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

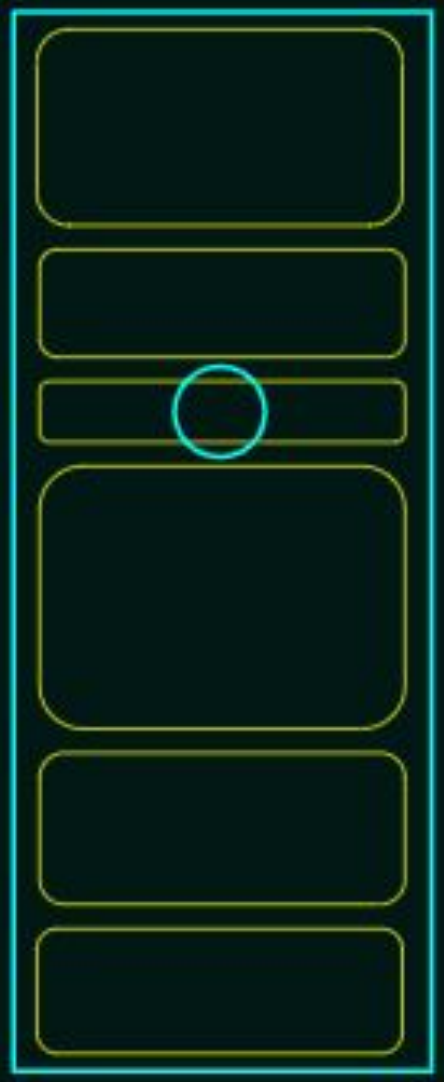
← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue



← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

← Main memory

← Memory divided into several fixed-sized partitions.

Processes waiting in ready queue

Dynamic Storage Allocation Problem

How to satisfy a request of size n from a list of free partitions/holes in main memory?

Solutions:

1. First Fit
2. Best Fit
3. Worst Fit

First Fit

- Allocate the first hole that is big enough.
- Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- We can stop searching as soon as we find a free hole that is large enough.

Processes waiting in ready queue

← Main memory

First Fit

- Allocate the first hole that is big enough.
- Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- We can stop searching as soon as we find a free hole that is large enough.

Processes waiting in ready queue

← Main memory

First Fit

- Allocate the first hole that is big enough.
- Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- We can stop searching as soon as we find a free hole that is large enough.

Processes waiting in ready queue

← Main memory

First Fit

- Allocate the first hole that is big enough.
- Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- We can stop searching as soon as we find a free hole that is large enough.

Processes waiting in ready queue

← Main memory

First Fit

- Allocate the first hole that is big enough.
- Searching can start either at the beginning of the set of holes or where the previous first-fit search ended.
- We can stop searching as soon as we find a free hole that is large enough.

Processes waiting in ready queue

← Main memory

Best Fit

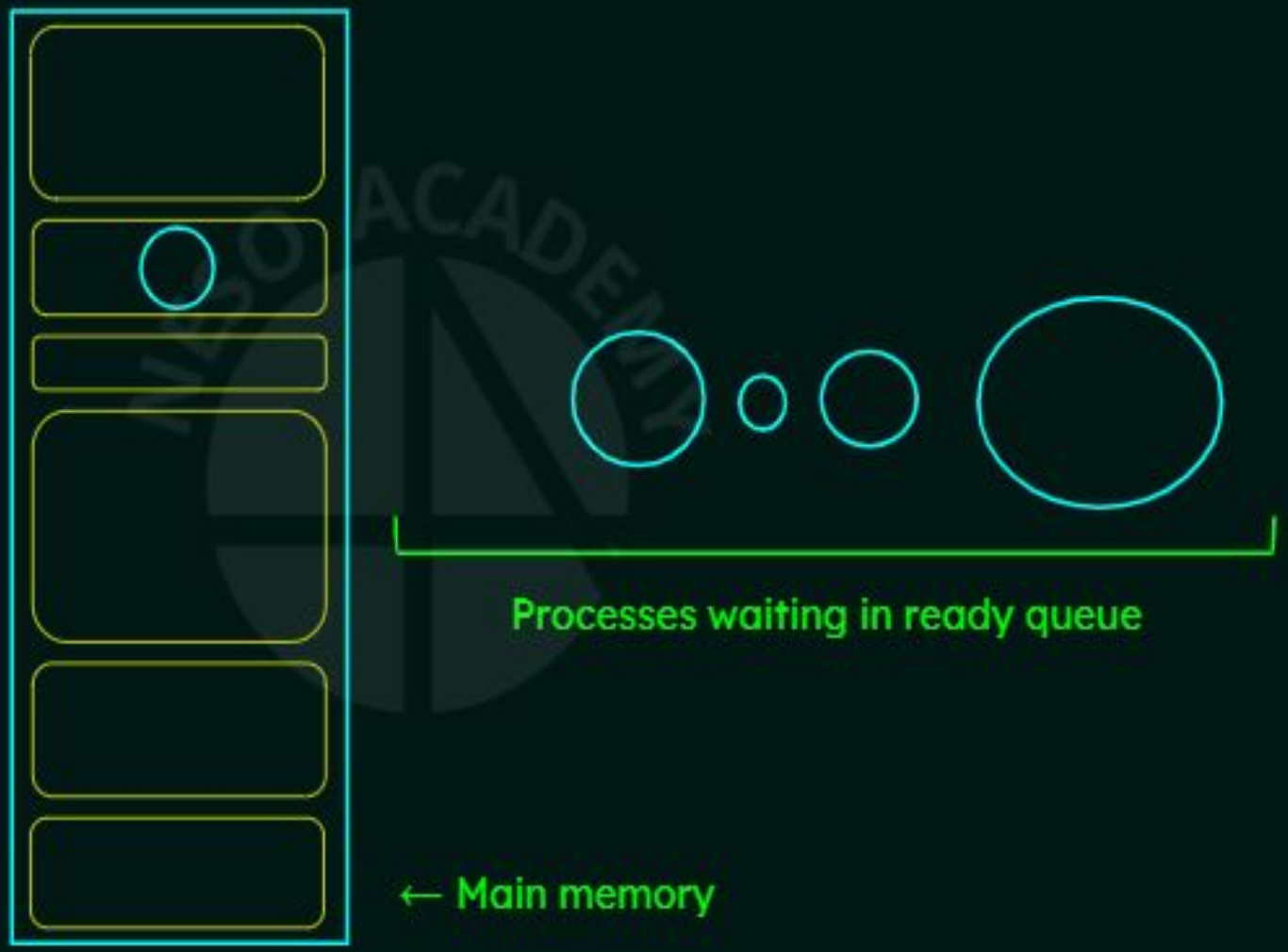
- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

Processes waiting in ready queue

← Main memory

Best Fit

- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.



Best Fit

- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

Processes waiting in ready queue

← Main memory

Best Fit

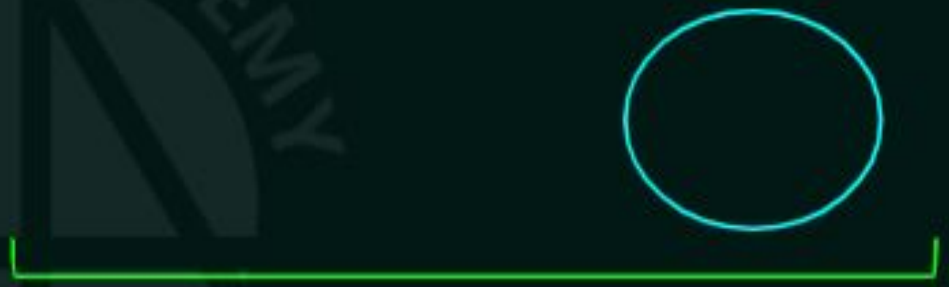
- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

Processes waiting in ready queue

← Main memory

Best Fit

- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.



Processes waiting in ready queue

← Main memory

Best Fit

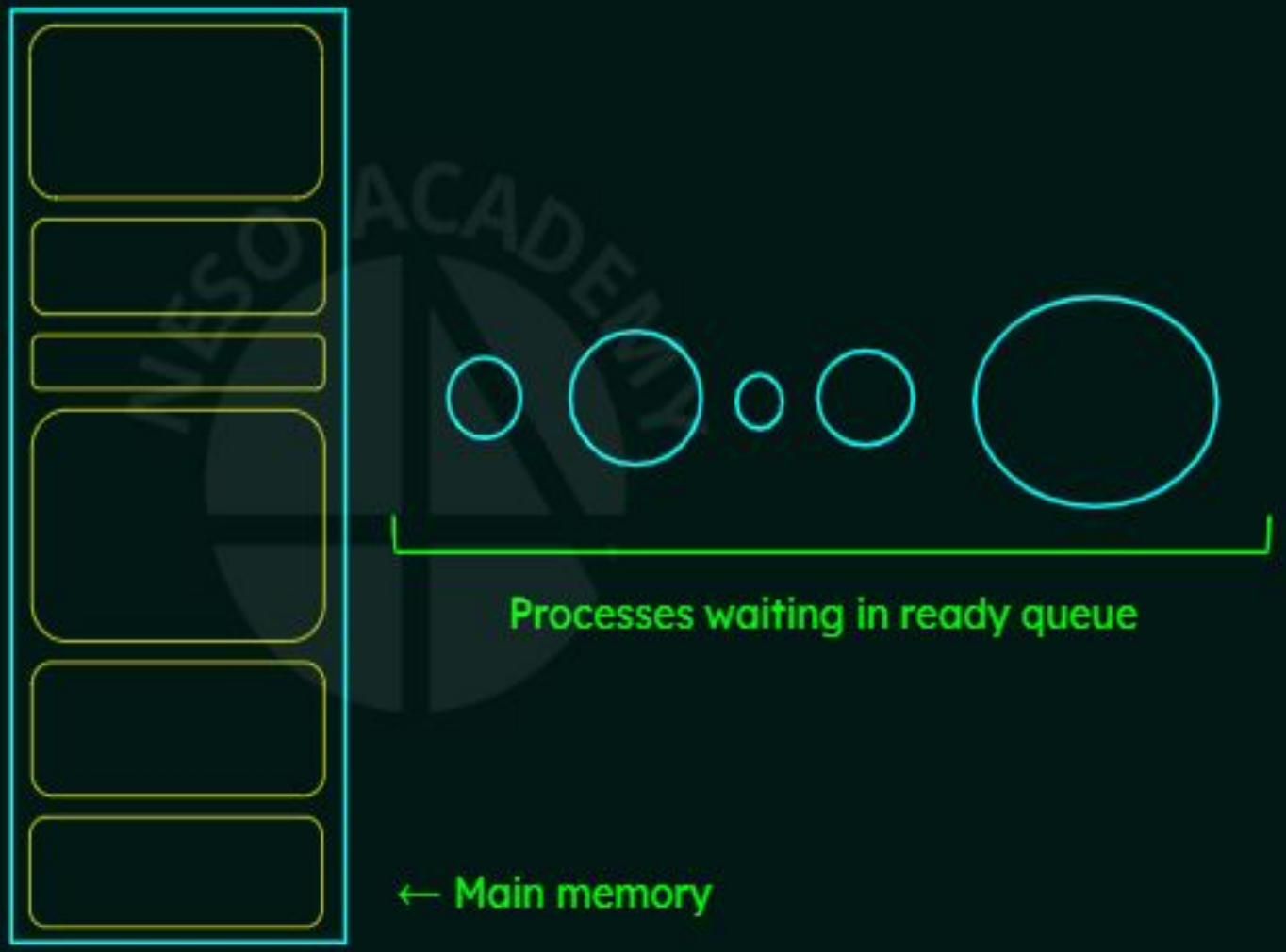
- Allocate the smallest hole that is big enough.
- We must search the entire list, unless the list is ordered by size.
- This strategy produces the smallest leftover hole.

Processes waiting in ready queue

← Main memory

Worst Fit

- Allocate the largest hole.
- We must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.



Worst Fit

- Allocate the largest hole.
- We must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.

Processes waiting in ready queue

← Main memory

Worst Fit

- Allocate the largest hole.
- We must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.

Processes waiting in ready queue

← Main memory

Worst Fit

- Allocate the largest hole.
- We must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.

Processes waiting in ready queue

← Main memory

Worst Fit

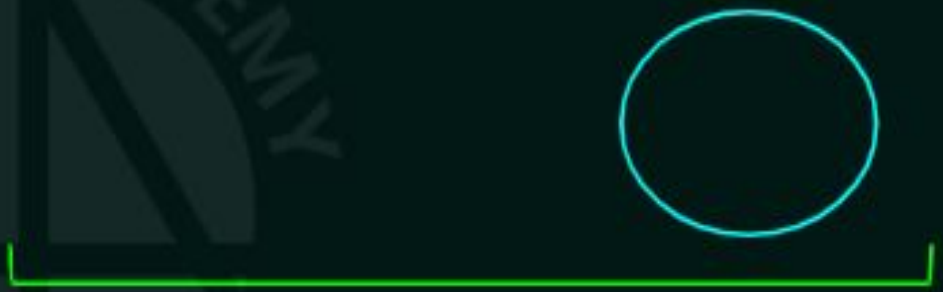
- Allocate the largest hole.
- We must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.

Processes waiting in ready queue

← Main memory

Worst Fit

- Allocate the largest hole.
- We must search the entire list, unless it is sorted by size.
- This strategy produces the largest leftover hole.



Processes waiting in ready queue

← Main memory

Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces which results in fragmentation.

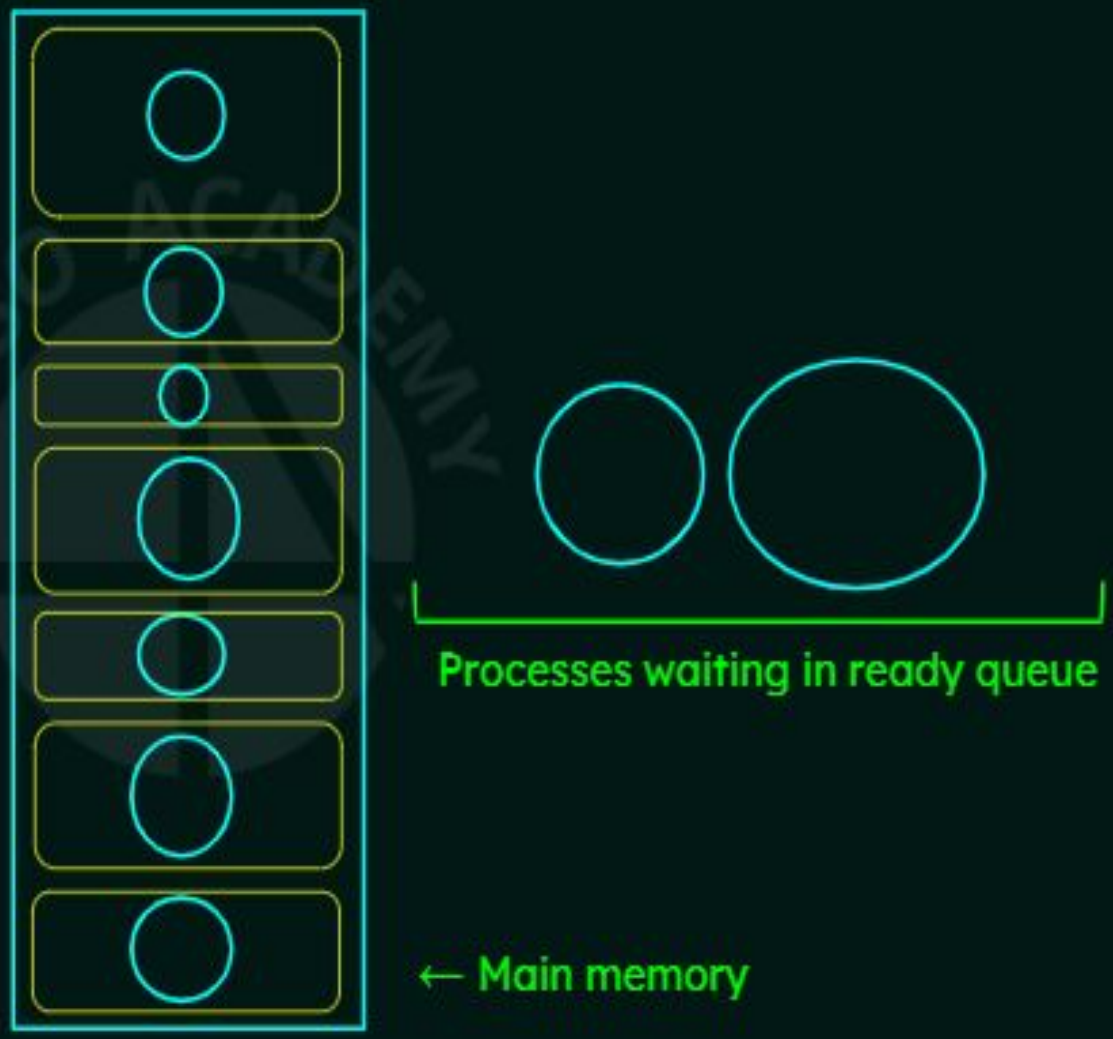
Types of Fragmentation:

1. External fragmentation

1. Internal fragmentation

External Fragmentation

- It exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.
- Storage is fragmented into a large number of small holes.
- This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.



External Fragmentation

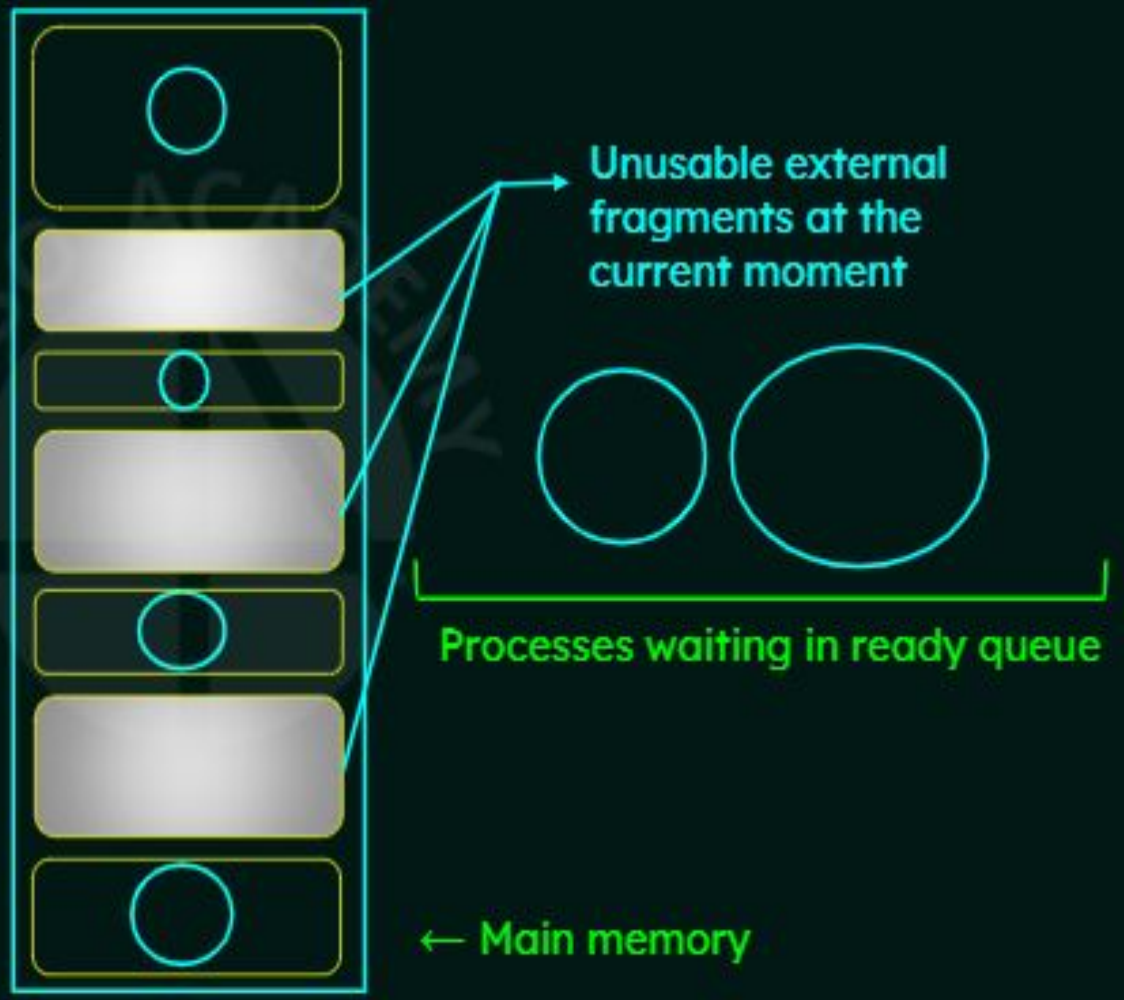
- It exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.
- Storage is fragmented into a large number of small holes.
- This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.

Processes waiting in ready queue

← Main memory

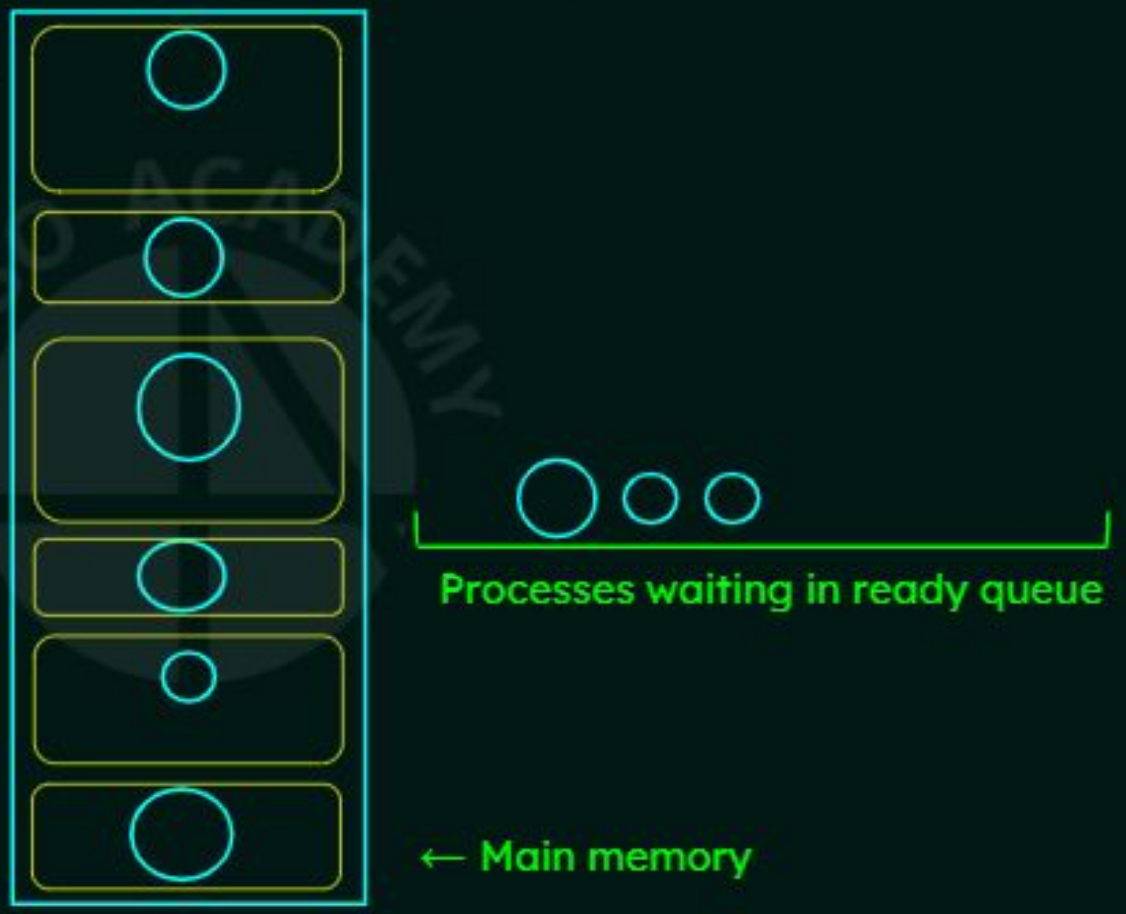
External Fragmentation

- It exists when there is enough total memory space to satisfy a request, but the available spaces are not contiguous.
- Storage is fragmented into a large number of small holes.
- This fragmentation problem can be severe. In the worst case, we could have a block of free (or wasted) memory between every two processes.
- If all these small pieces of memory were in one big free block instead, we might be able to run several more processes.



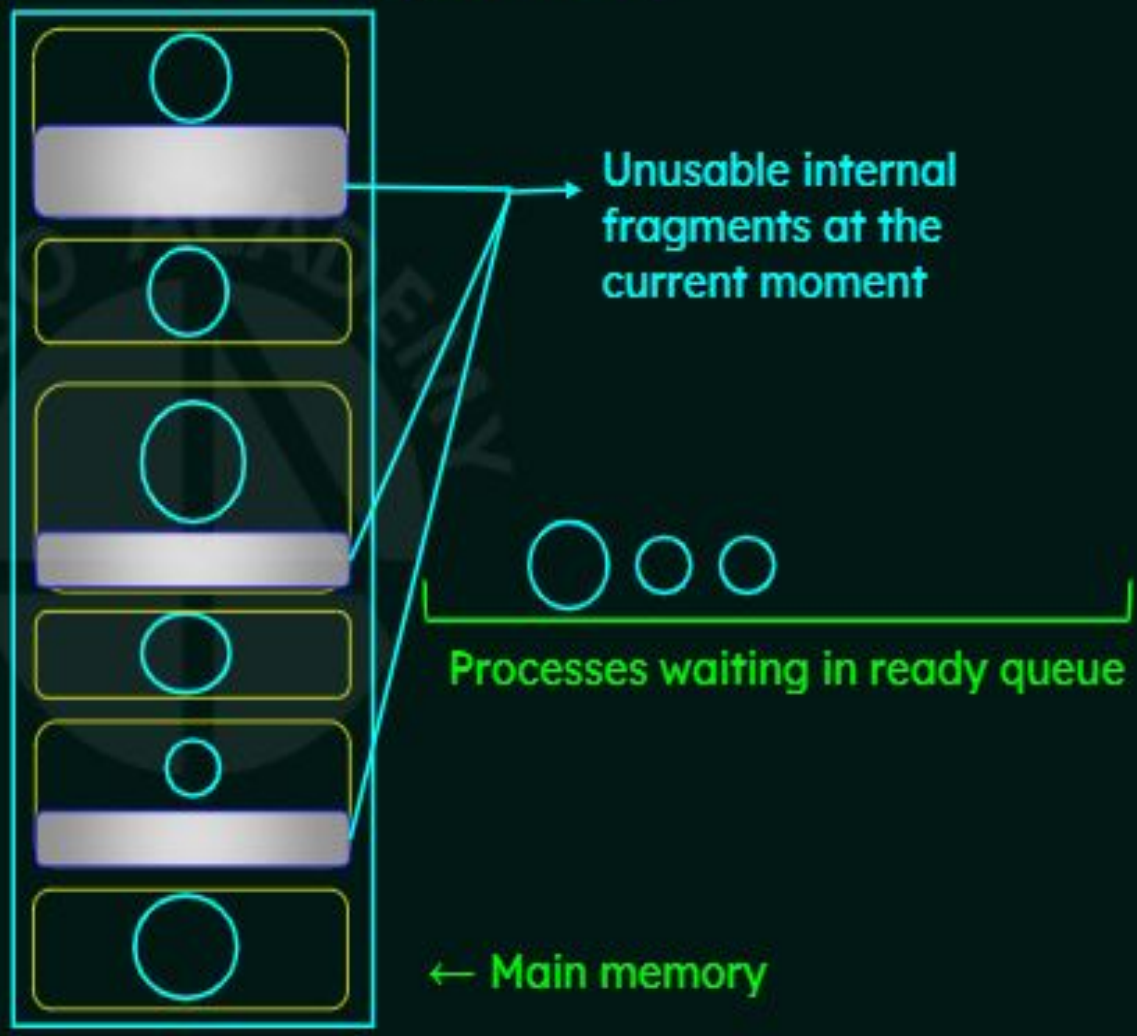
Internal Fragmentation

- It occurs when memory blocks assigned to processes are bigger than what the process actually needs.
- Some portion of memory is left unused, as it cannot be used by another process.



Internal Fragmentation

- It occurs when memory blocks assigned to processes are bigger than what the process actually needs.
- Some portion of memory is left unused, as it cannot be used by another process.

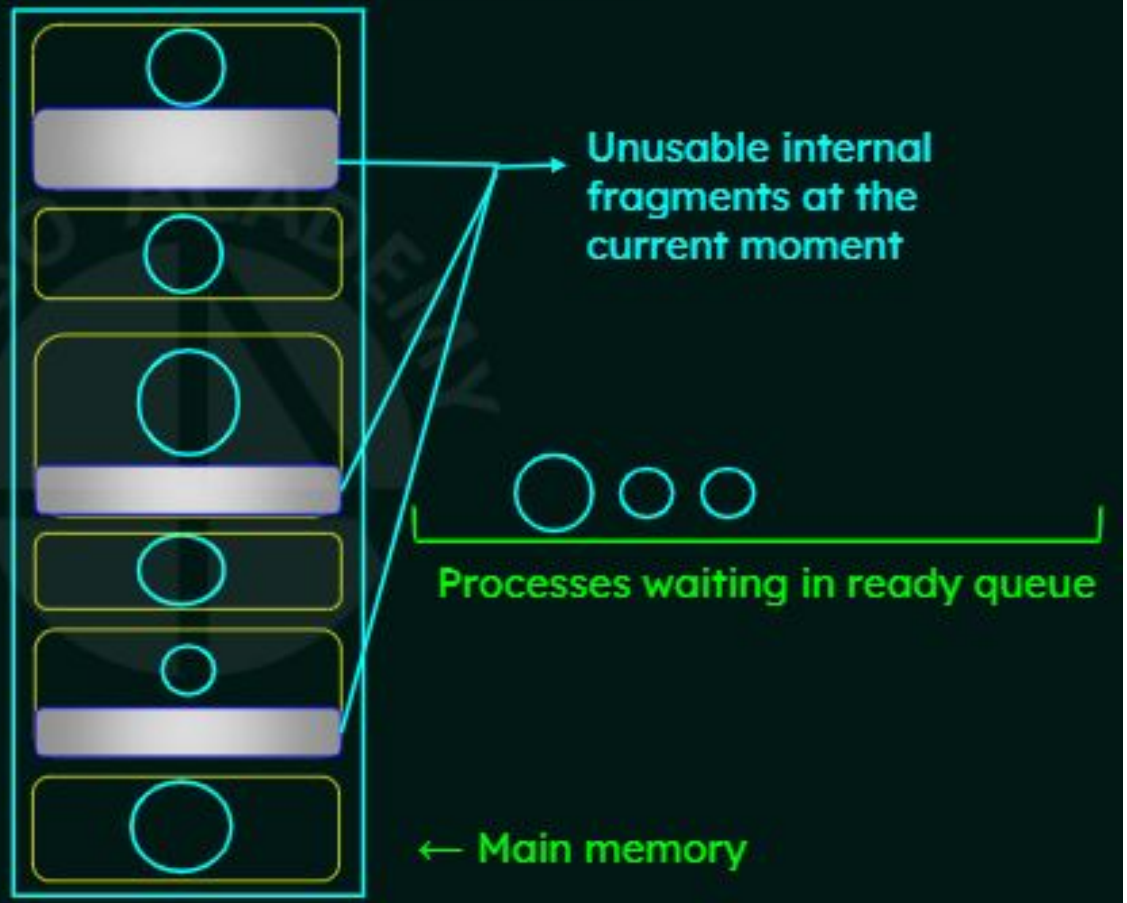


Internal Fragmentation

- It occurs when memory blocks assigned to processes are bigger than what the process actually needs.
- Some portion of memory is left unused, as it cannot be used by another process.

Solution:

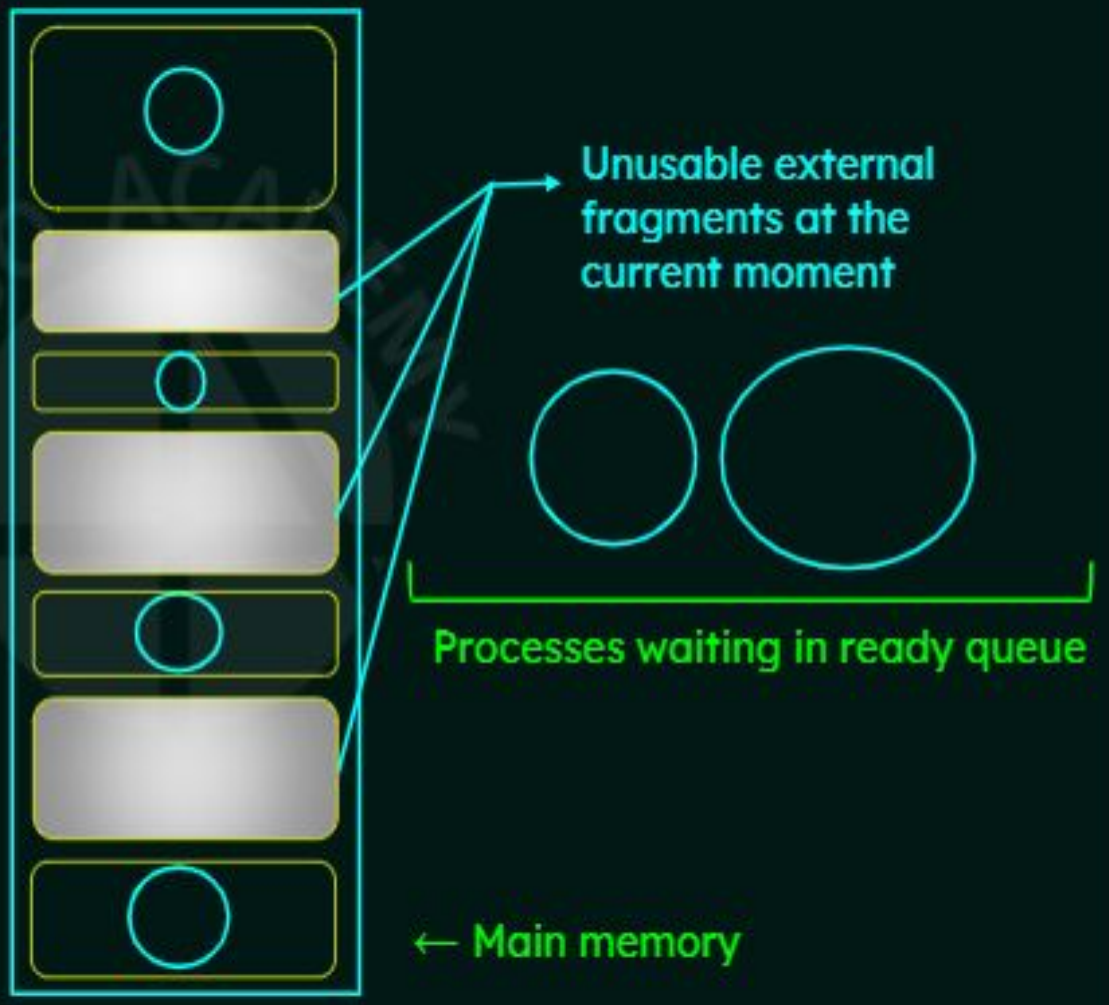
Make use of **Best-Fit approach** for allocating memory segments to processes



Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.



Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.

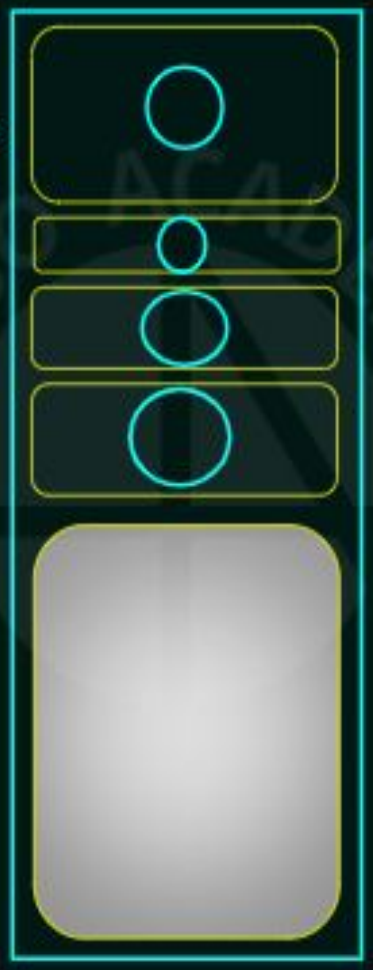
Processes waiting in ready queue

← Main memory

Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.



Processes waiting in ready queue

← Main memory

Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.

Processes waiting in ready queue

← Main memory

Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.

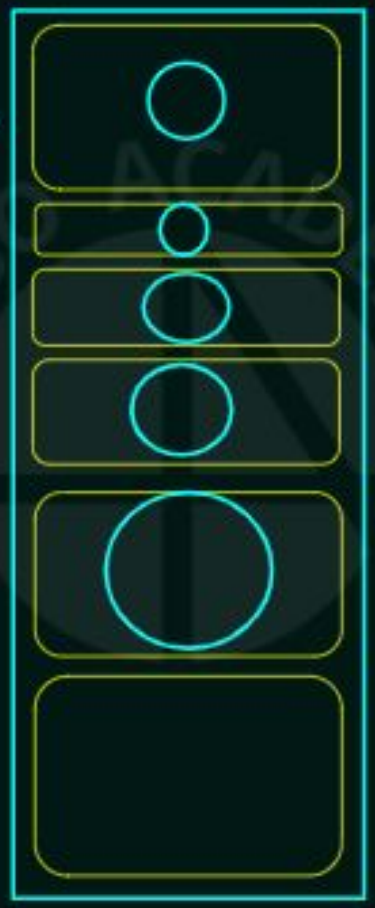
Processes waiting in ready queue

← Main memory

Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.



Processes waiting in ready queue

← Main memory

Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.

Processes waiting in ready queue

← Main memory

Solution to External Fragmentation

Compaction

Shuffle the memory contents so as to place all free memory together in one large block.

Problems:

- Compaction is not always possible.
- If relocation is static and is done at assembly or load time, compaction cannot be done.
- It is possible only if relocation is dynamic and is done at execution time.

Processes waiting in ready queue

← Main memory

Paging

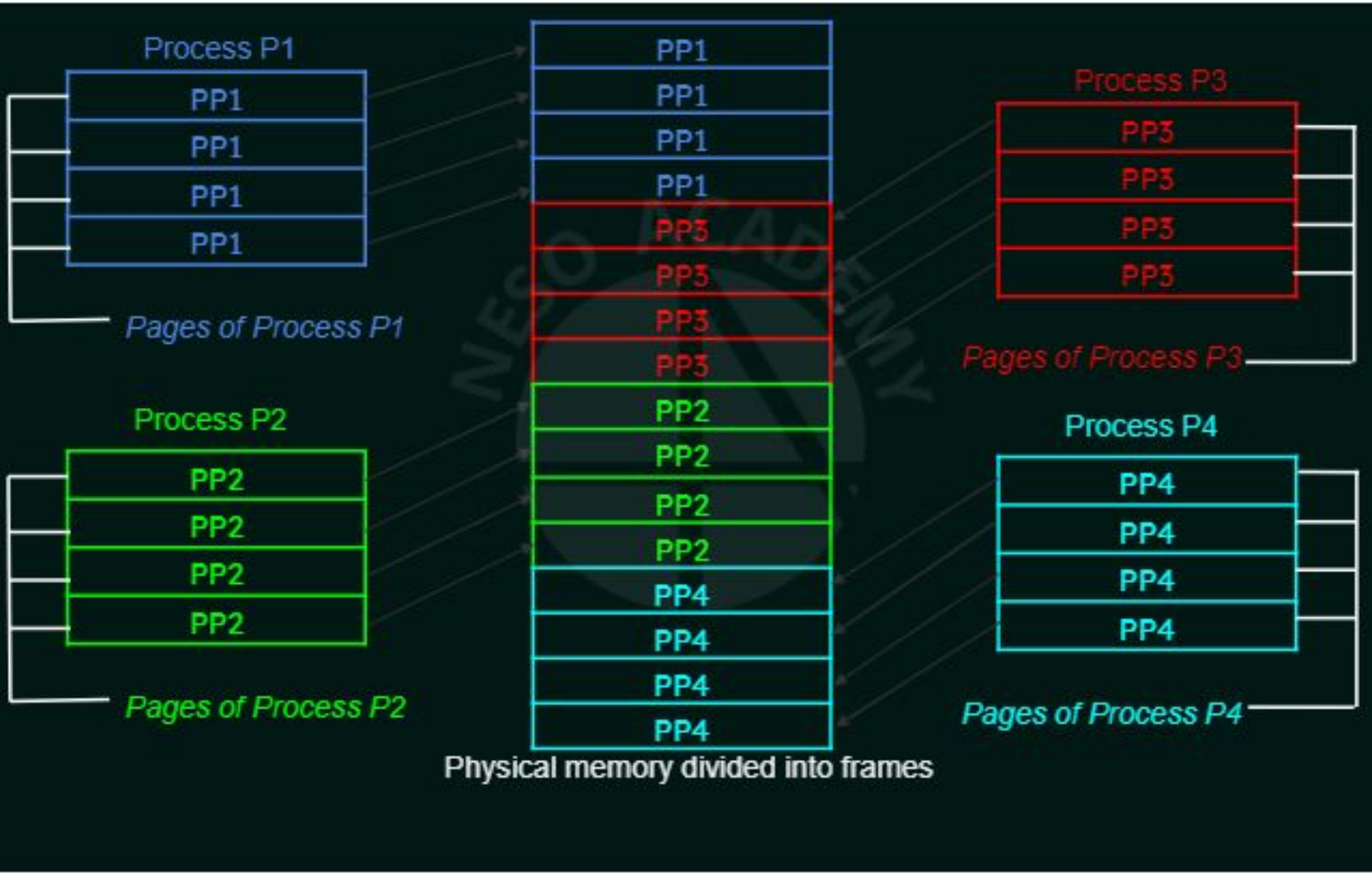
Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous.

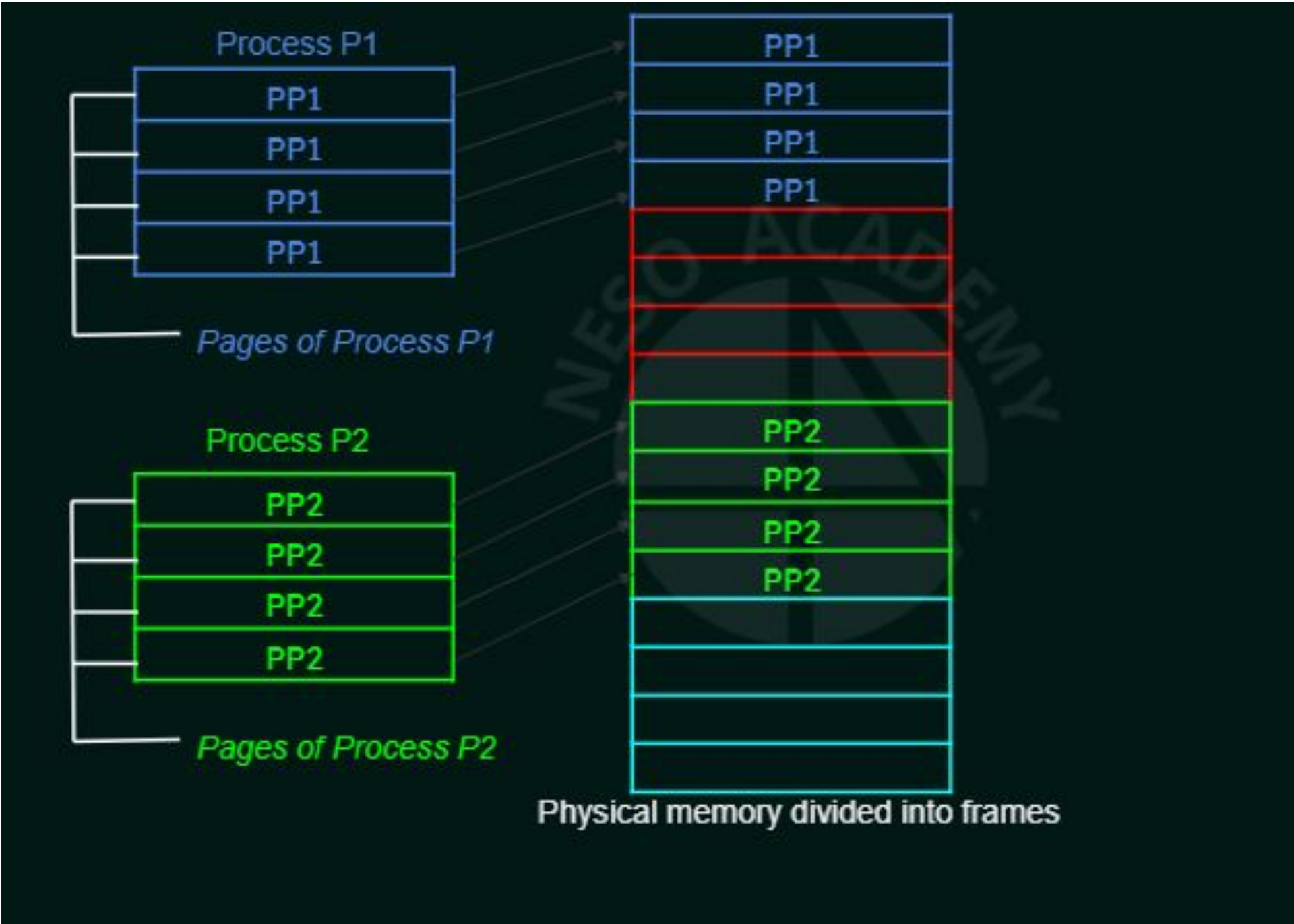
Paging avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store.

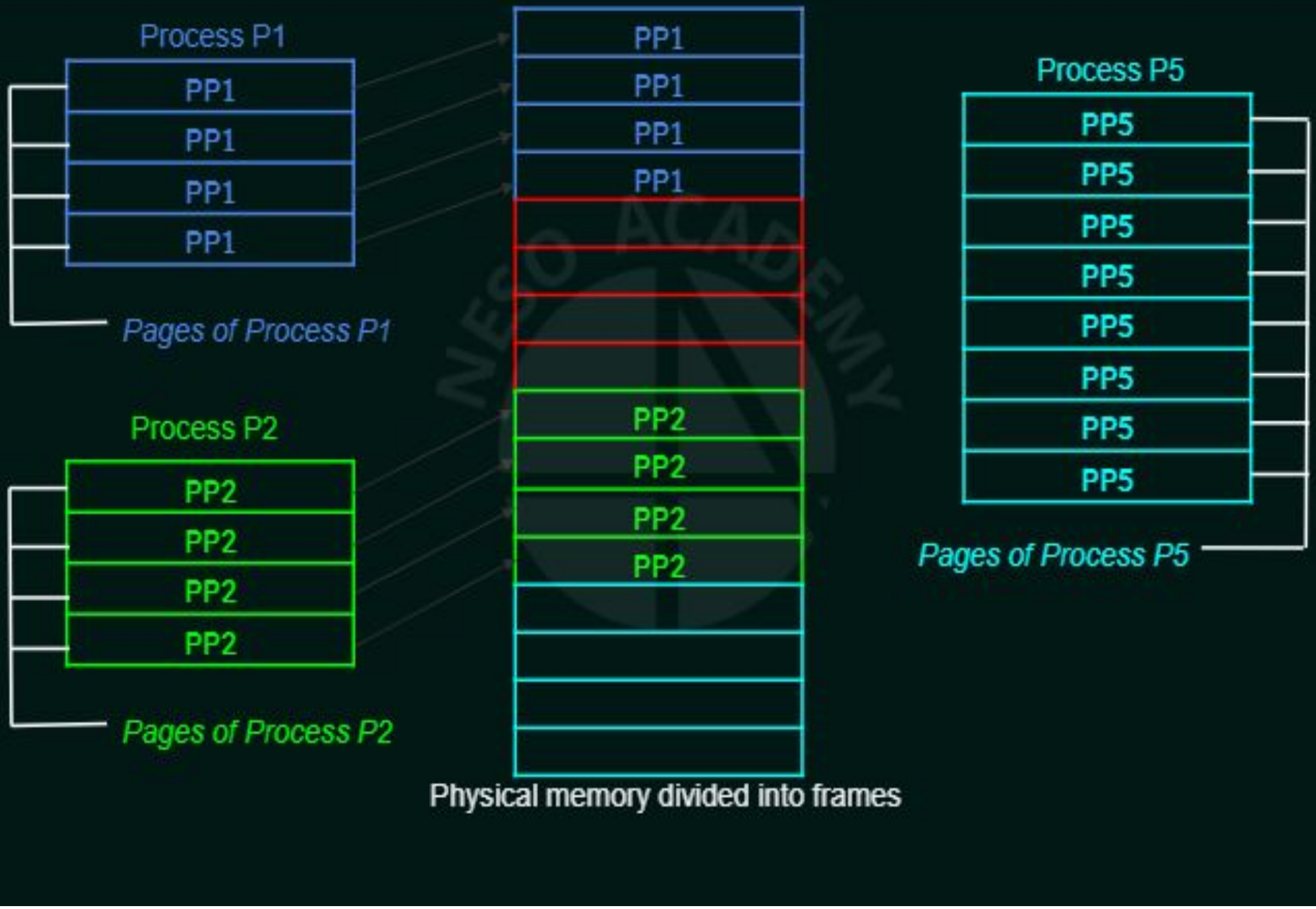
Most memory-management schemes that we discussed before suffered from this problem.

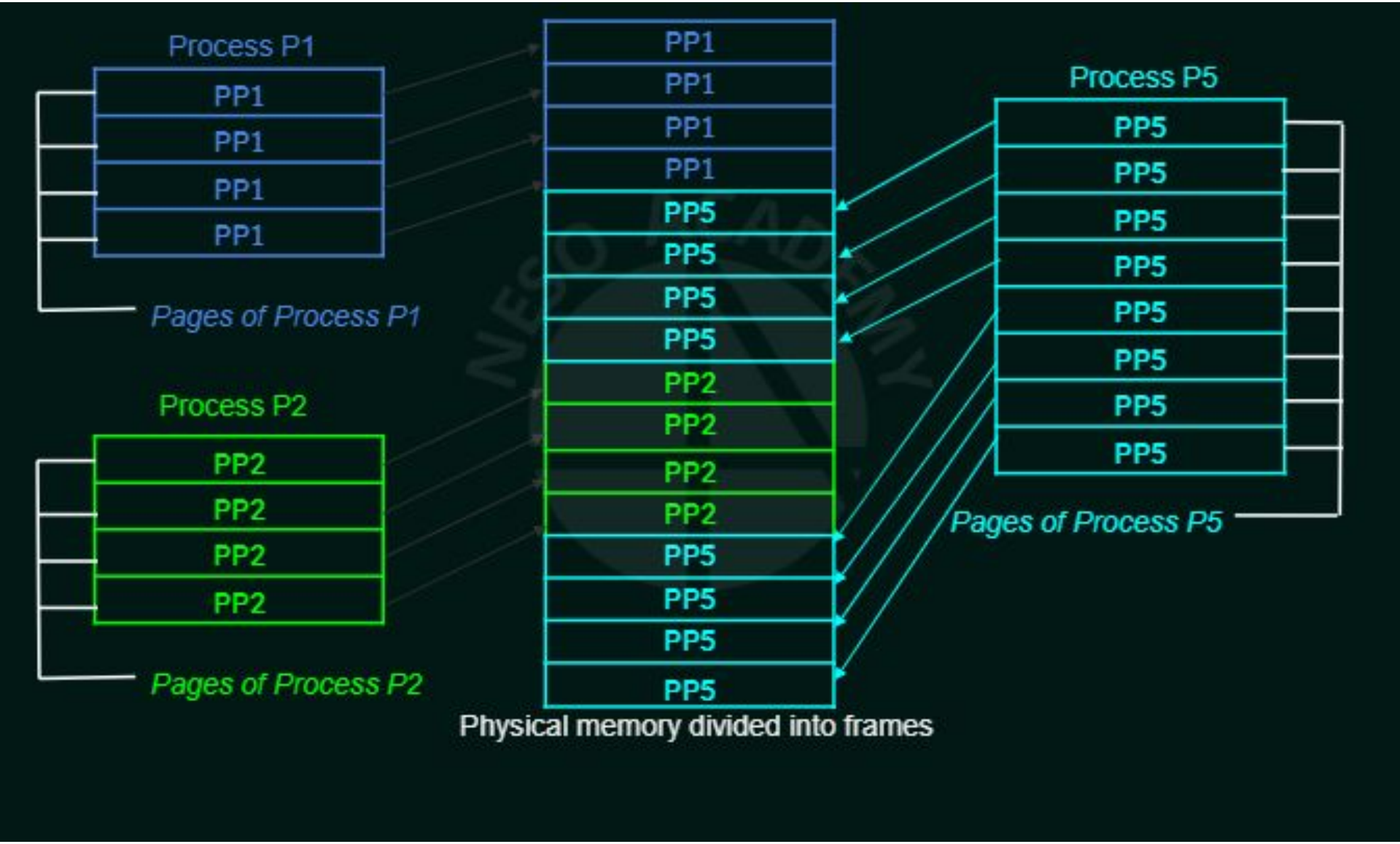
Basic method of Paging

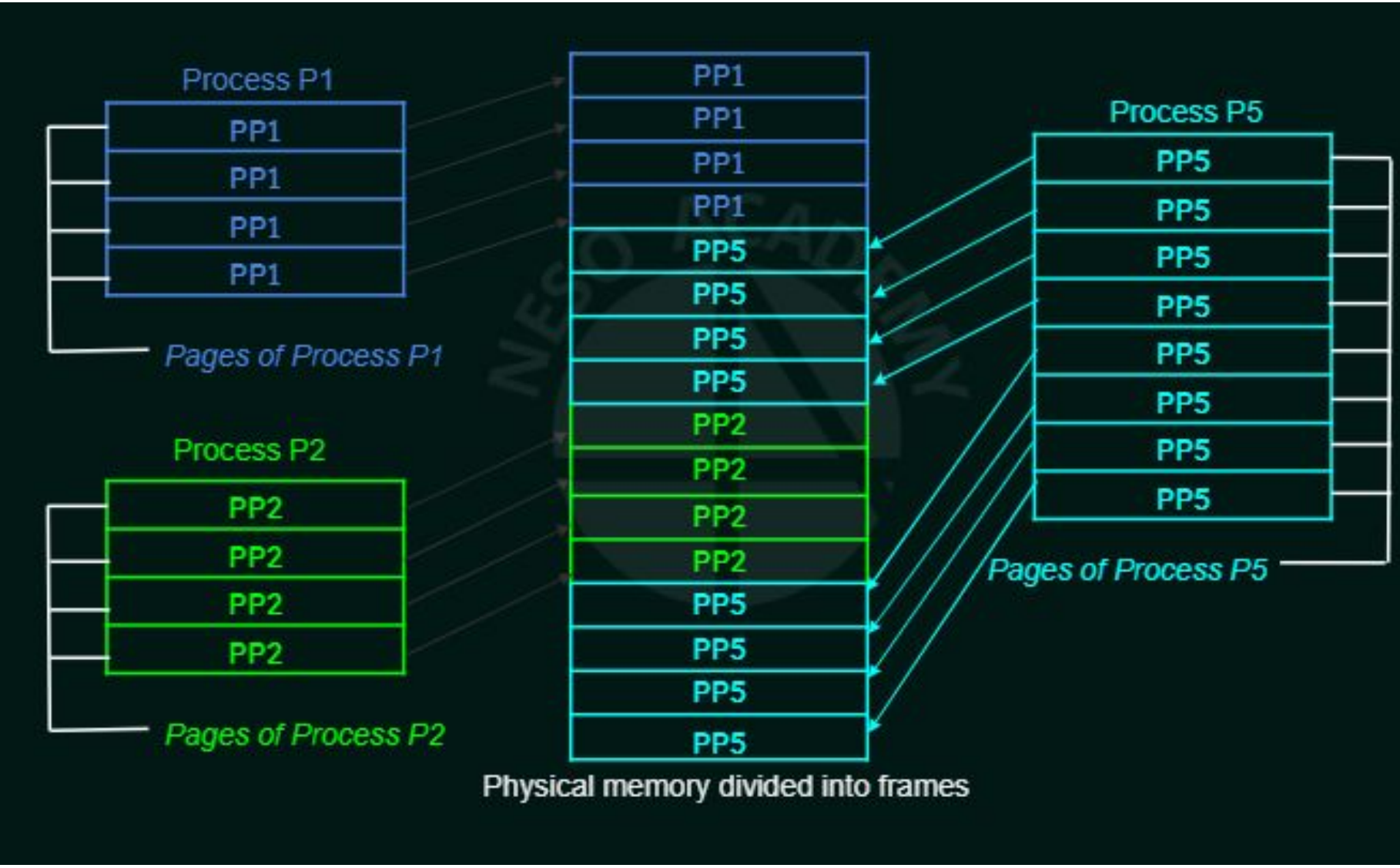
- Break physical memory into fixed-sized blocks called frames.
- Breaks logical memory into blocks of the same size called pages.
- When a process is to be executed, its pages are loaded into any available memory frames from the backing store.
- The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.


Process P1
PP1
PP1
PP1
PP1
Pages of Process P1
Process P2
PP2
PP2
PP2
PP2
Pages of Process P2
PP1
PP1
PP1
PP1
PP3
PP3
PP3
PP3
PP2
PP2
PP2
PP2
PP4
PP4
PP4
PP4
Physical memory divided into frames
Process P3
PP3
PP3
PP3
PP3
Pages of Process P3
Process P4
PP4
PP4
PP4
PP4
Pages of Process P4


Process P1
PP1
PP1
PP1
PP1
Pages of Process P1
Process P2
PP2
PP2
PP2
PP2
Pages of Process P2
PP1
PP1
PP1
PP1
PP2
PP2
PP2
PP2
Physical memory divided into frames


Process P1
PP1
PP1
PP1
PP1
Pages of Process P1
Process P2
PP2
PP2
PP2
PP2
Pages of Process P2
PP1
PP1
PP1
PP1
PP2
PP2
PP2
PP2
Physical memory divided into frames
Process P5
PP5
PP5
PP5
PP5
PP5
PP5
PP5
PP5
Pages of Process P5


Process P1
PP1
PP1
PP1
PP1
Pages of Process P1
Process P2
PP2
PP2
PP2
PP2
Pages of Process P2
PP1
PP1
PP1
PP1
PP5
PP5
PP5
PP5
PP2
PP2
PP2
PP2
PP5
PP5
PP5
PP5
Physical memory divided into frames
Process P5
PP5
PP5
PP5
PP5
PP5
PP5
PP5
PP5
Pages of Process P5


Process P1
PP1
PP1
PP1
PP1
Pages of Process P1
Process P2
PP2
PP2
PP2
PP2
Pages of Process P2
PP1
PP1
PP1
PP1
PP5
PP5
PP5
PP5
PP2
PP2
PP2
PP2
PP5
PP5
PP5
PP5
Physical memory divided into frames
Process P5
PP5
PP5
PP5
PP5
PP5
PP5
PP5
PP5
Pages of Process P5

Page Table

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

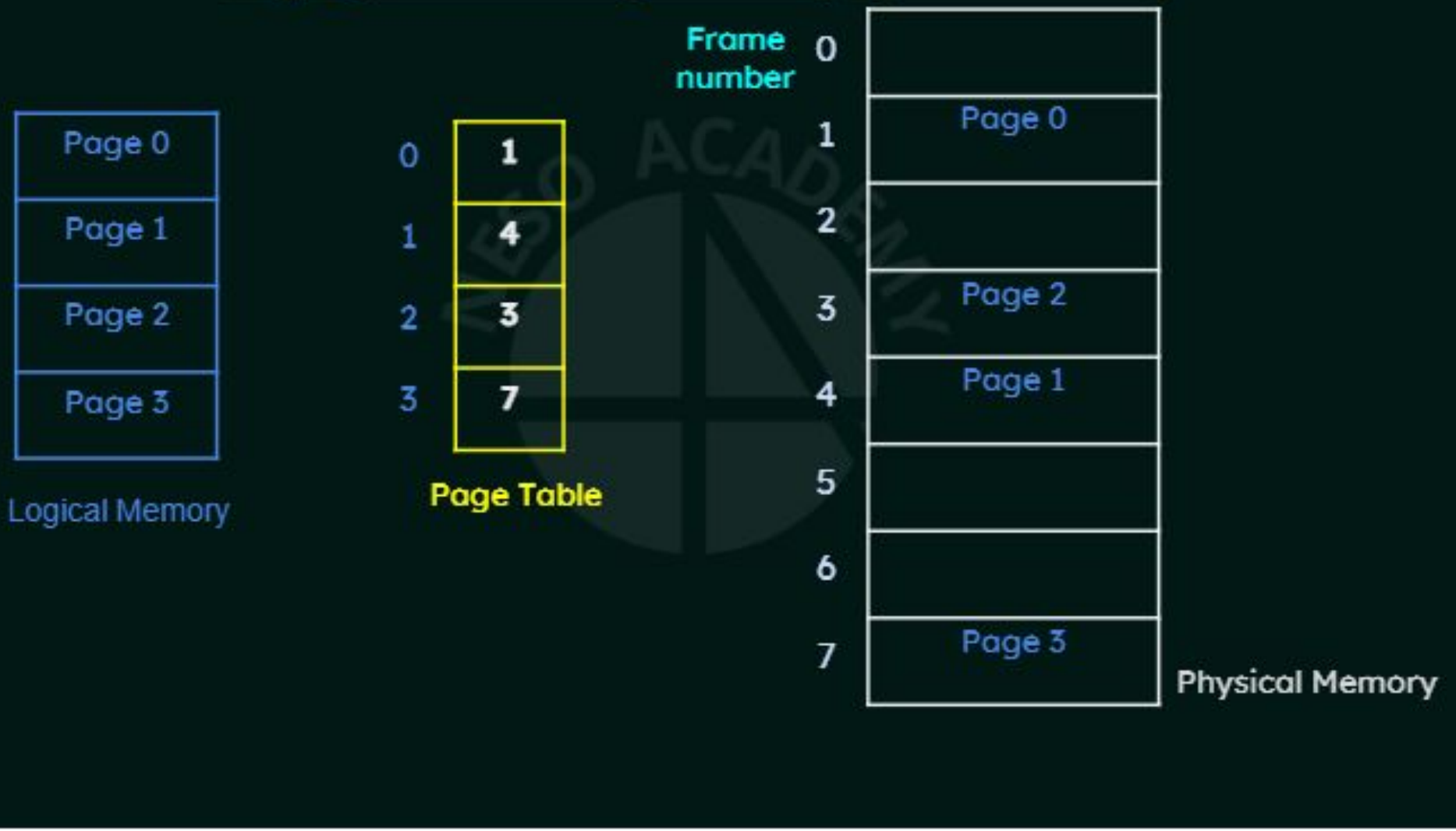
Page Number (p) - Used as an index into a page table.

Page Offset (d) - The displacement within the page

The page table contains the base address of each page in physical memory.

This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

Paging model of logical and physical memory



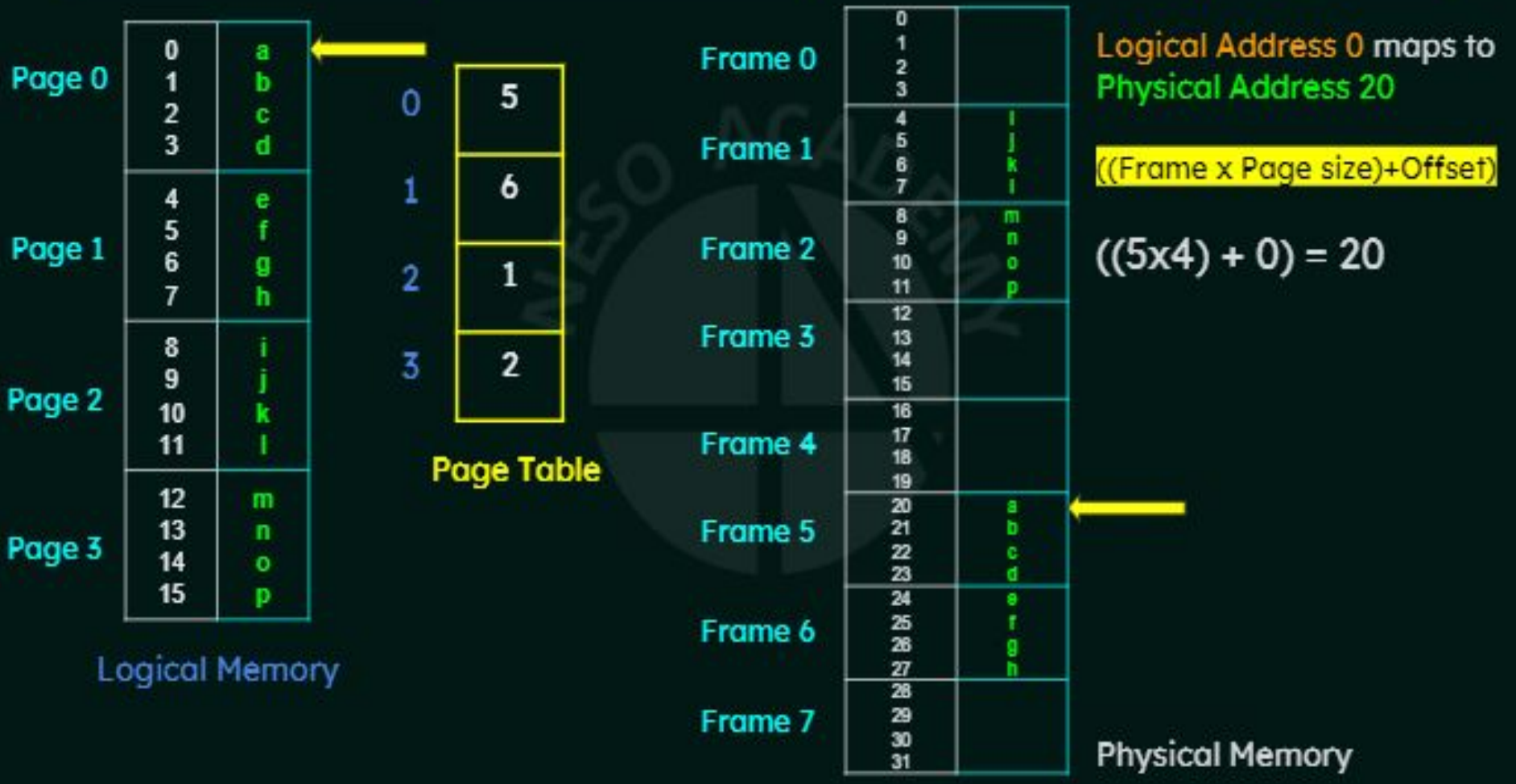
Translation of Logical Address to Physical Address using Page Table

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

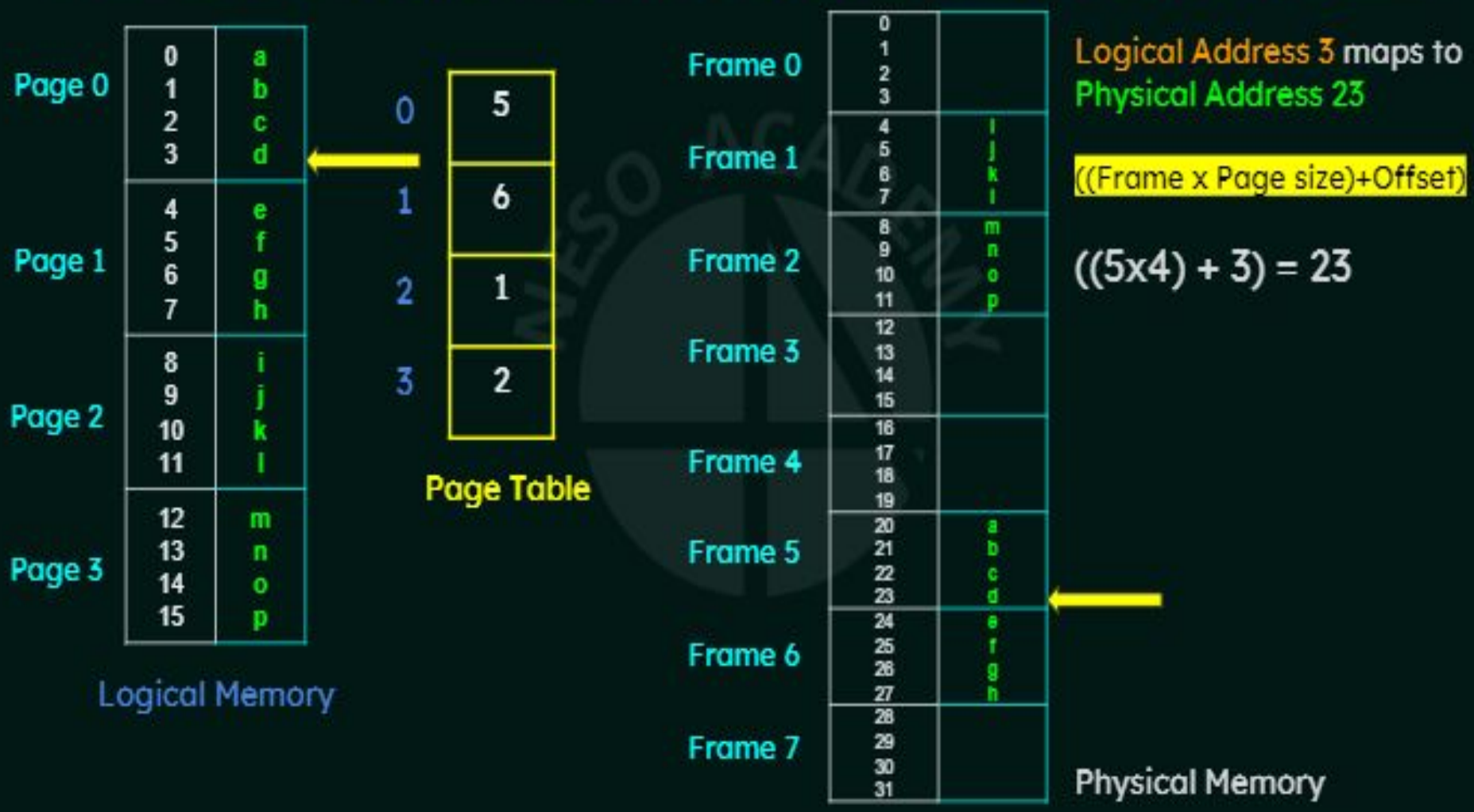
Page Number (p) - Used as an index into a page table.

Page Offset (d) - The displacement within the page

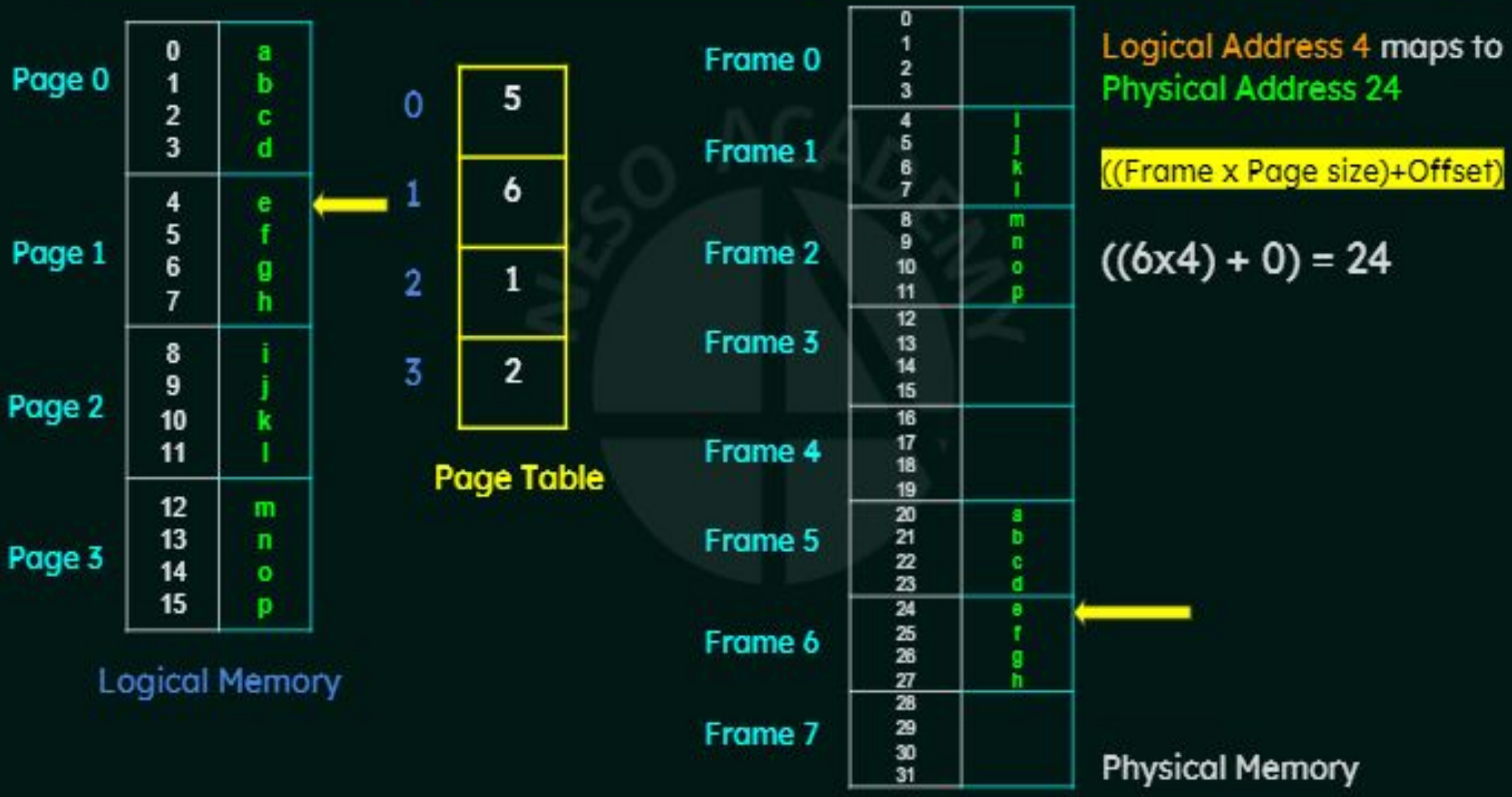
Translation of Logical Address to Physical Address using Page Table



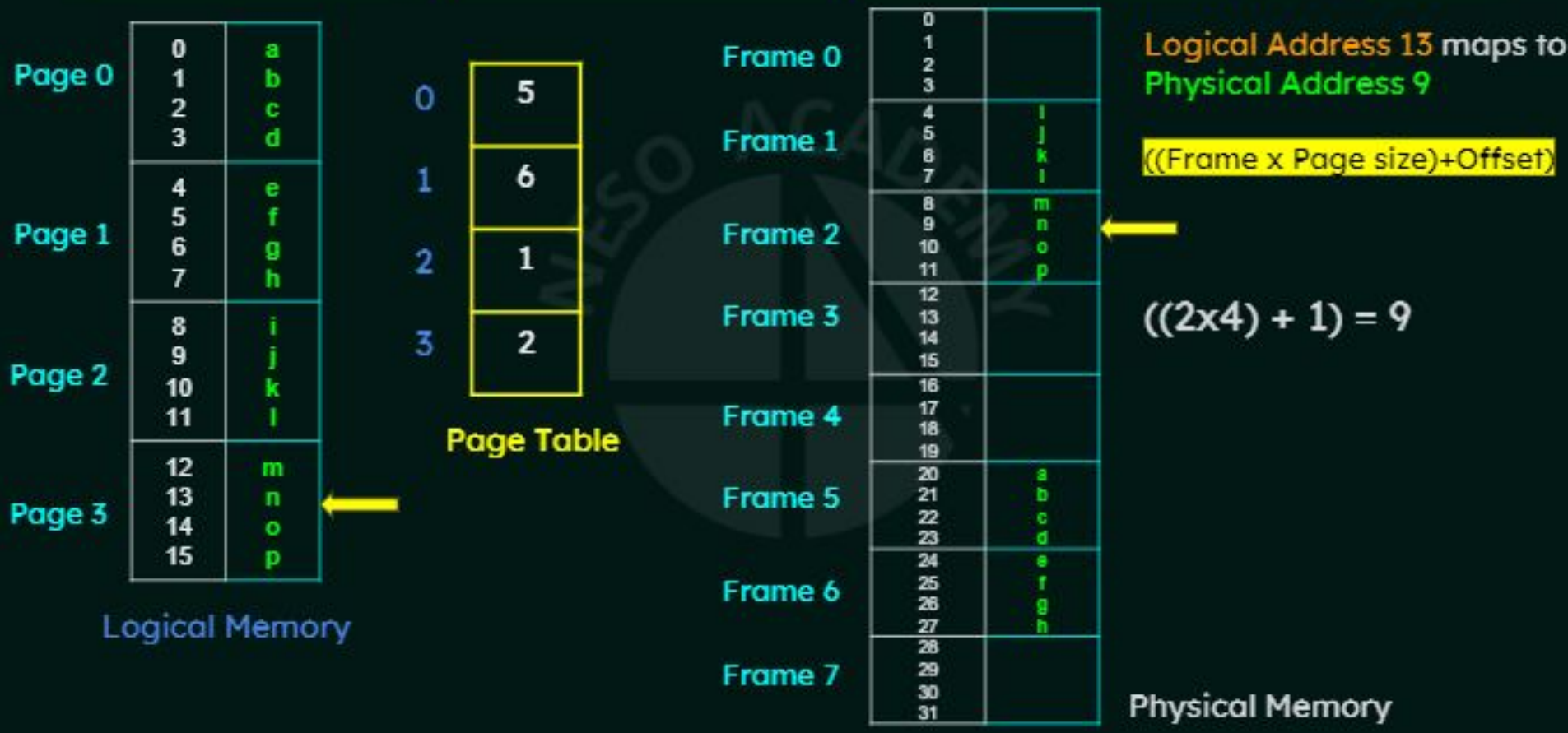
Translation of Logical Address to Physical Address using Page Table



Translation of Logical Address to Physical Address using Page Table



Translation of Logical Address to Physical Address using Page Table



Hardware implementation of Page Table

Case 1: Implement the page table as a set of dedicated registers.

Problem: This can be used only when page table is reasonably small.

Case 2: Keep the page table in main memory and a page-table base register (PTBR) points to the page table.

Problem: The time required to access a user memory location is increased as there are two memory accesses needed to access a byte (one for the page-table entry, one for the byte)

Solution: Make use of

Translation Lookaside Buffer (TLB)

The TLB is associative, high-speed memory.

It consists of two parts.

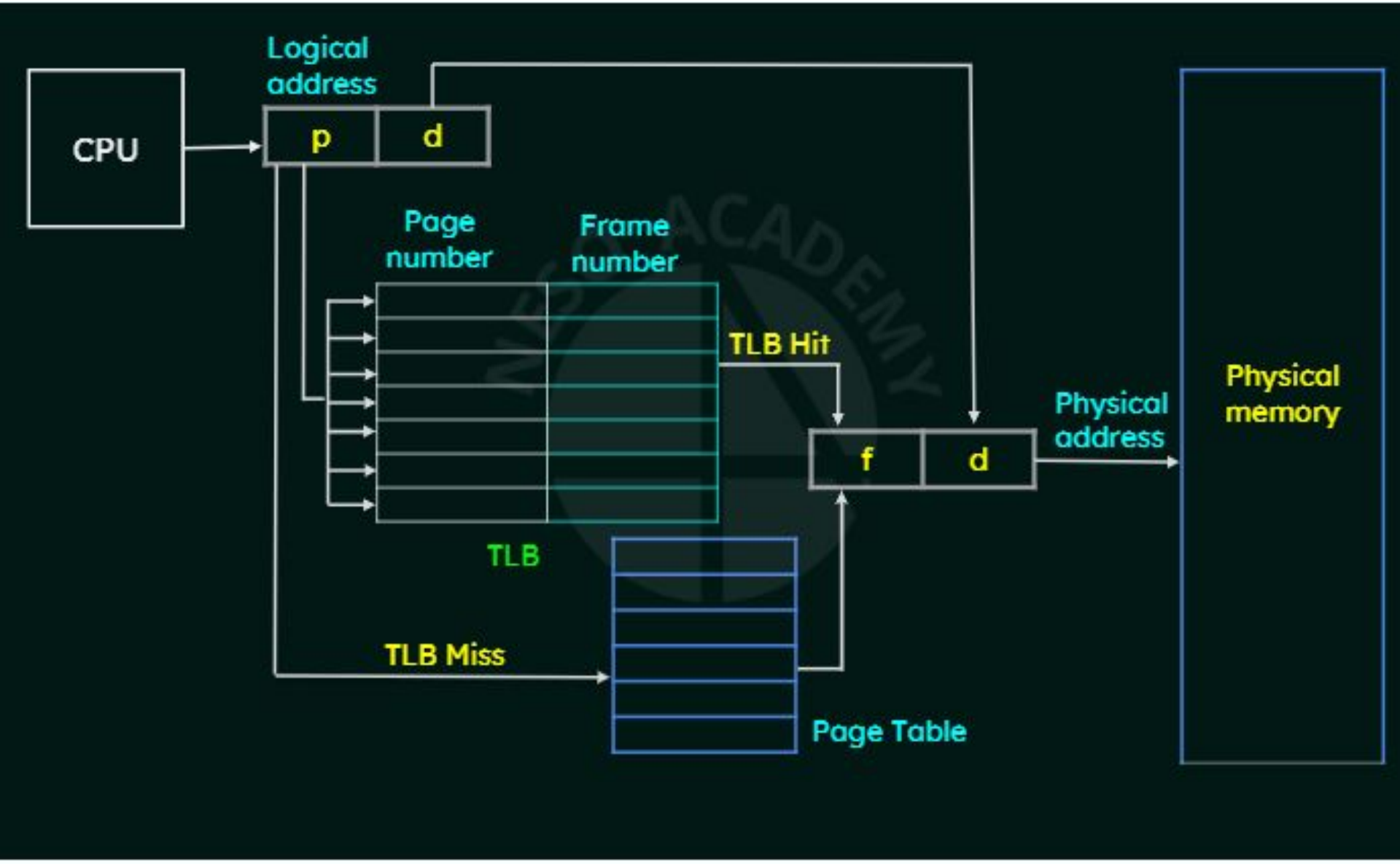
When the associative memory is presented with an item, the item is compared with all keys simultaneously.

If the item is found, the corresponding value field is returned.

The search is fast.

Usage of TLB with Page Table

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory. → **TLB Hit**
- If the page number is not in the TLB, a memory reference to the page table must be made. → **TBL Miss**
- When the frame number is obtained, we can use it to access memory.
- Also we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.

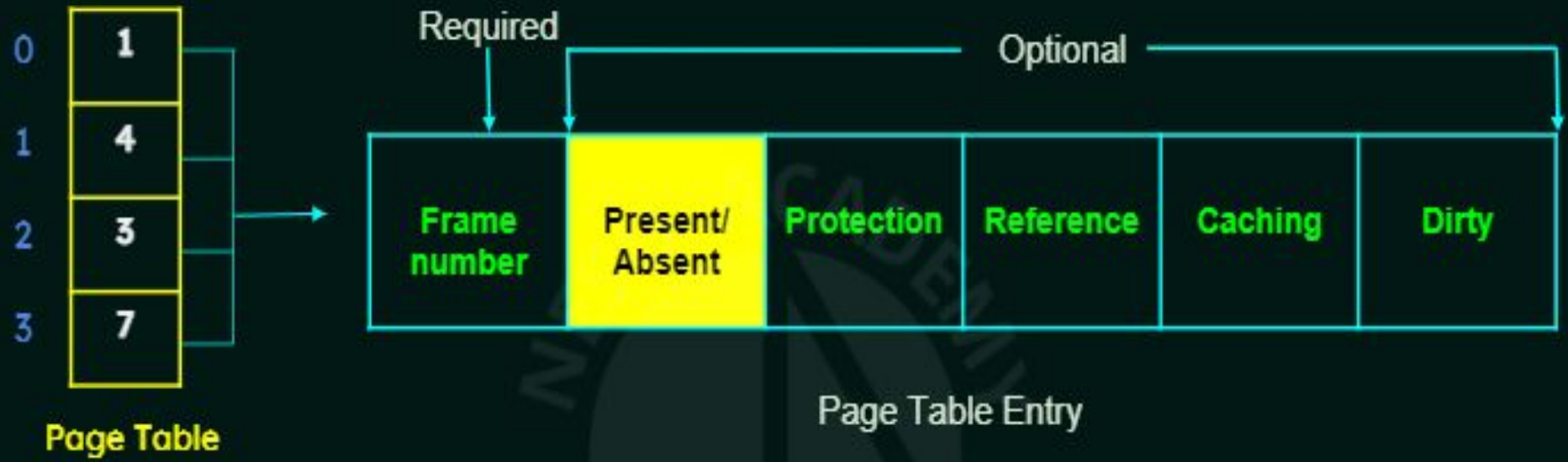

CPU
Logical address
p
d
Page number
Frame number
TLB Hit
Physical address
f
d
Physical memory
TLB
TLB Miss
Page Table

Page Table Entries

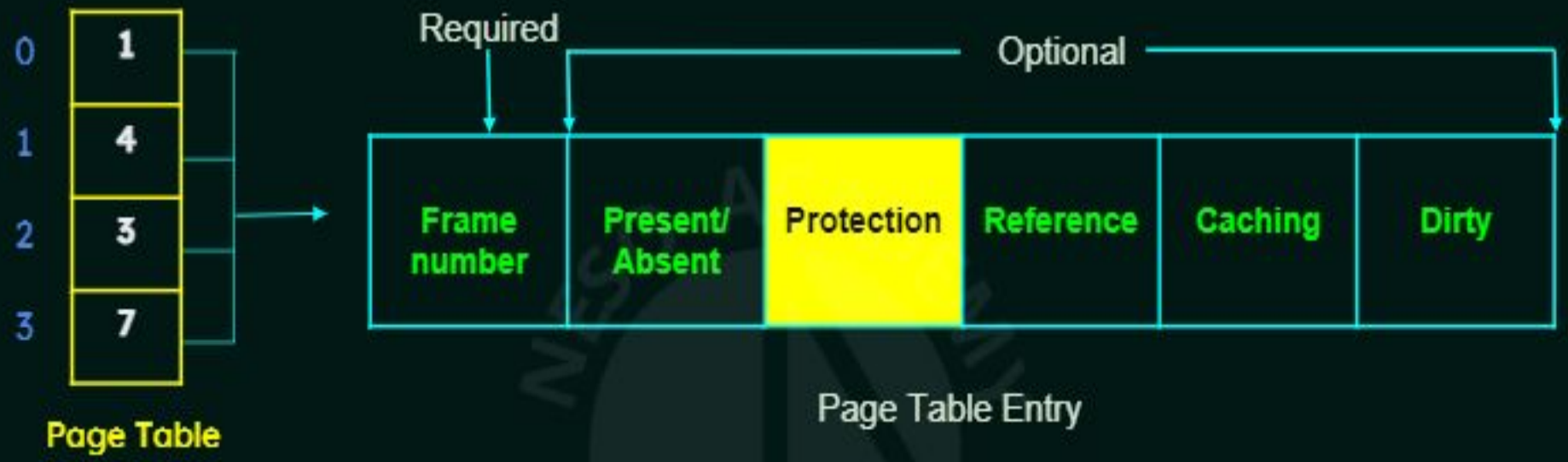
Page Table

- Page table entries contains several information about pages which vary from OS to OS.
- The most important information in a page table entry is the frame number.
- The remaining informations are optional.

- Frame number denotes the frame where the page is present in the main memory.
- The number of bits required for this depends on the number of frames in the main memory.



- The Present / Absent bit specifies whether the page is present in the main memory or not.
- It is also called Valid / Invalid bit.
- If the page we are looking for is not present in main memory, it is called PAGE FAULT.
- If the page we are looking for is not present in main memory, the Present/Absent bit is set to 0.



- The Protection bit also known as Read/Write bit is used for page protection.
- It specifies the permissions for read or write operations on the page.
- The bit is set to 0 if only read operation is allowed.
- The bit is set to 1 if both read and write operations are allowed.

- The Reference bit specifies whether the page has been referenced in the last clock cycle or not.
- It is set to 1 when the page is accessed.

- The caching bit is used for enabling or disabling caching of the page.
- When we need fresh data we have to disable caching so as to avoid fetching of old data from the cache.
- When caching has to be disabled, this bit is set to 1. Otherwise it is set to 0.

- The Dirty bit is also known as the Modified bit.
- It specifies whether the page has been modified or not.
- If the page has been modified, then this bit is set to 1 otherwise set to 0.
- This bit helps in avoiding unnecessary writes to the secondary memory when a page is being replaced by another page.

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- This is particularly important in a time-sharing environment

Consider a system that supports 40 users, each of whom executes a text editor.

If the text editor consists of 150 KB of code and 50 KB of data space, we need:

(150 x 40) + (50 x 40)

= 6000 + 2000

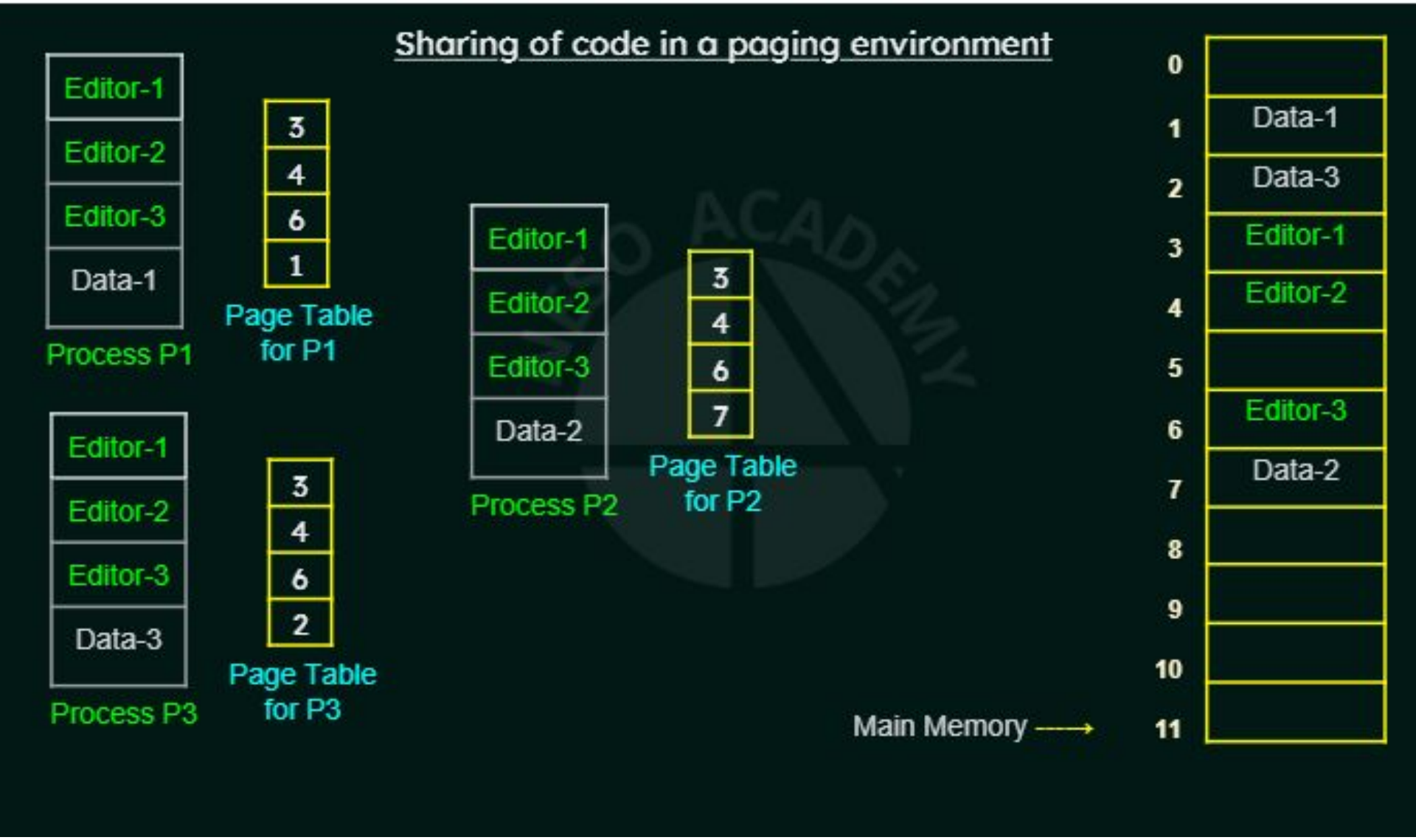
= 8000 KB to support the 40 users.

However, if the code is reentrant code (or pure code), it can be shared among different processes.

Code that is non-self-modifying code. It never changes during execution.

Thus, two or more processes can execute the same code at the same time.

- Each process has its own copy of registers and data storage to hold the data for the process's execution.
- The data for two different processes will, of course, be different.


Sharing of code in a paging environment
Editor-1
Editor-2
Editor-3
Data-1
Process P1
3
4
6
1
Page Table for P1
Editor-1
Editor-2
Editor-3
Data-3
Process P3
3
4
6
2
Page Table for P3
Editor-1
Editor-2
Editor-3
Data-2
Process P2
3
4
6
7
Page Table for P2
0
1
2
3
4
5
6
7
8
9
10
11
Data-1
Data-3
Editor-1
Editor-2
Editor-3
Data-2
Main Memory

Shared Pages

- An advantage of paging is the possibility of sharing common code.
- This is particularly important in a time-sharing environment

Consider a system that supports 40 users, each of whom executes a text editor.

If the text editor consists of 150 KB of code and 50 KB of data space, we need:

(150 x 40) + (50 x 40)

= 6000 + 2000

= 8000 KB to support the 40 users.

Now, if the code is shared, we need:

150 + (50 x 40)

= 150 + 2000

= 2150 KB to support 40 users

Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64}).

In such an environment, the page table itself becomes excessively large.

Example: Consider a system with:
32-bit logical address space.
If Page Size = 4 KB (2^{12})
Then a page table may consist upto (2^{32} / 2^{12}) = 1 million entries (approx.)

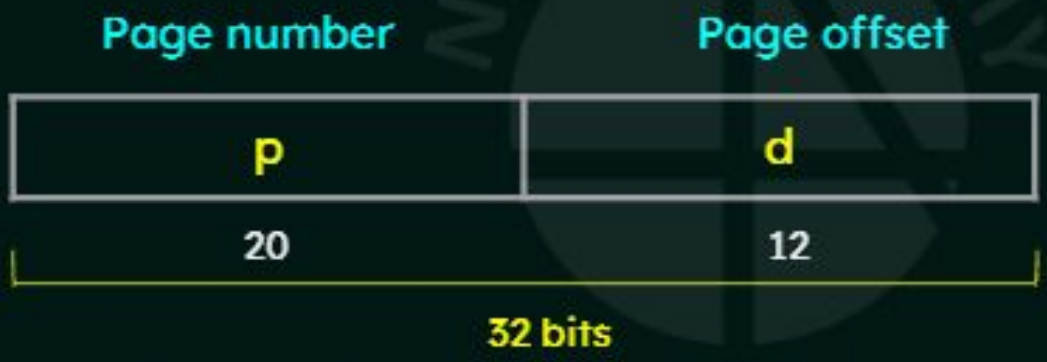
Consider each entry consists of 4 bytes
So, each process may need upto (4 x 1 million) bytes = 4 MB of physical address space.

Trying to allocate a Page Table of this size contiguously in main memory is clearly not a good idea.

Solution: Divide the page table into smaller sizes.

Multilevel Paging

Use a two-level paging algorithm, in which the page table itself is also paged.



Page Number (p) - Used as an index into a page table.

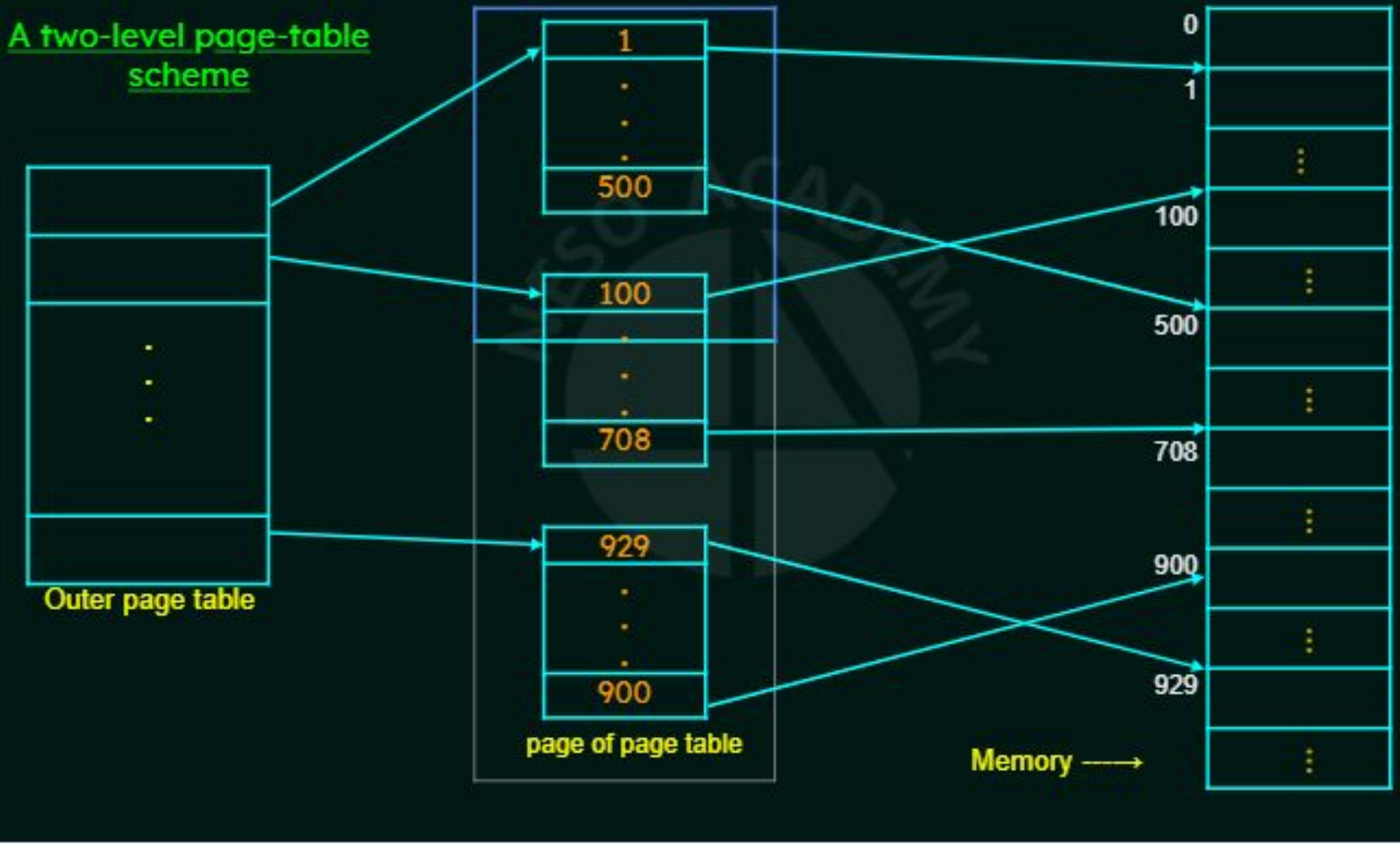
Page Offset (d) - The displacement within the page.

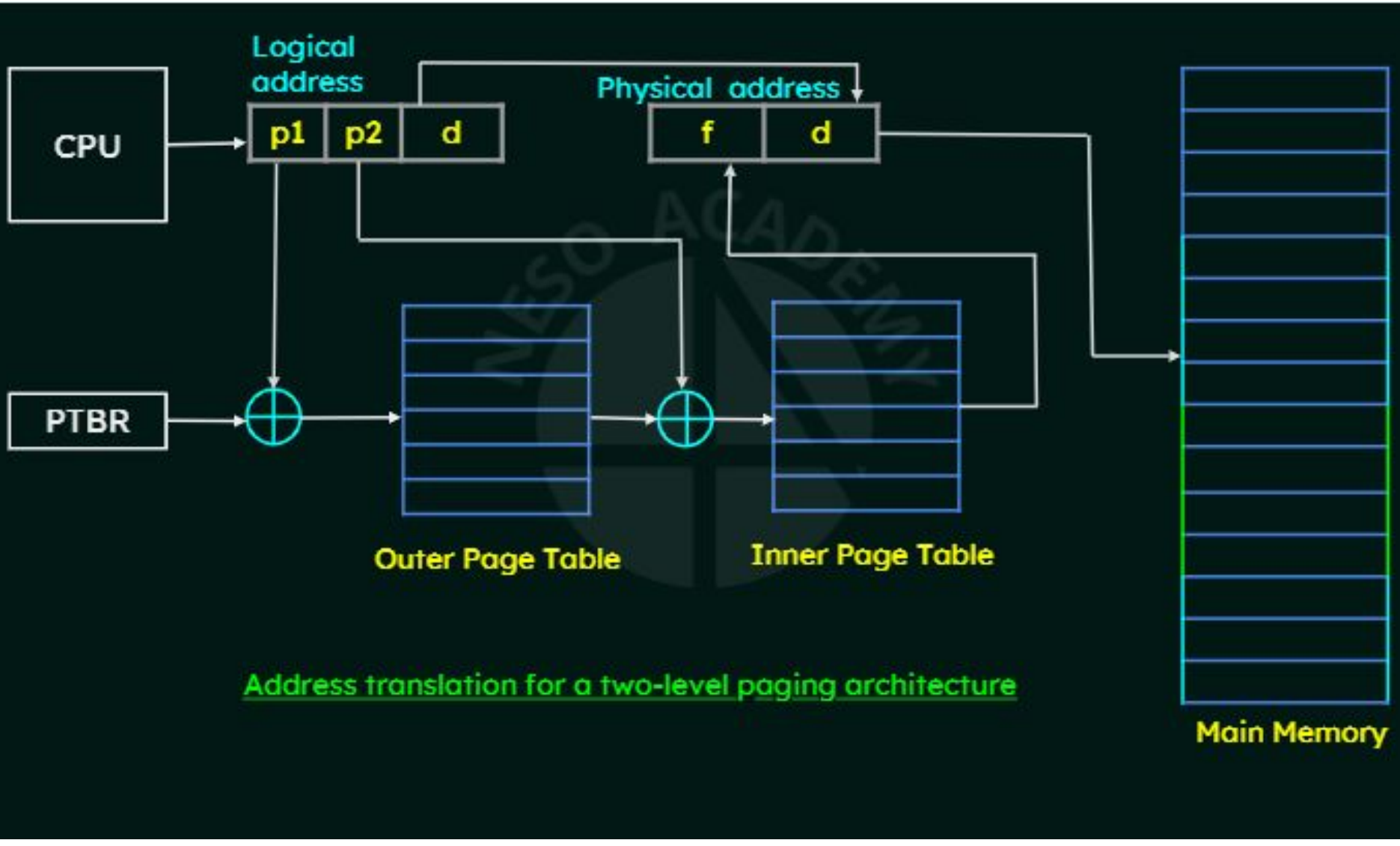
Solution: Divide the page table into smaller sizes.

Multilevel Paging

Use a two-level paging algorithm, in which the page table itself is also paged.

p1 - Index into the outer page table.
p2 - Displacement within the page of the outer page table.
Page Offset (d) - The displacement within the page.


A two-level page-table scheme
1
500
100
708
929
900
Outer page table
page of page table
0
1
100
500
708
900
929
Memory



Address translation for a two-level paging architecture

Hashed Page Tables

Used for handling address spaces larger than 32 bits (or large address spaces).

Here,

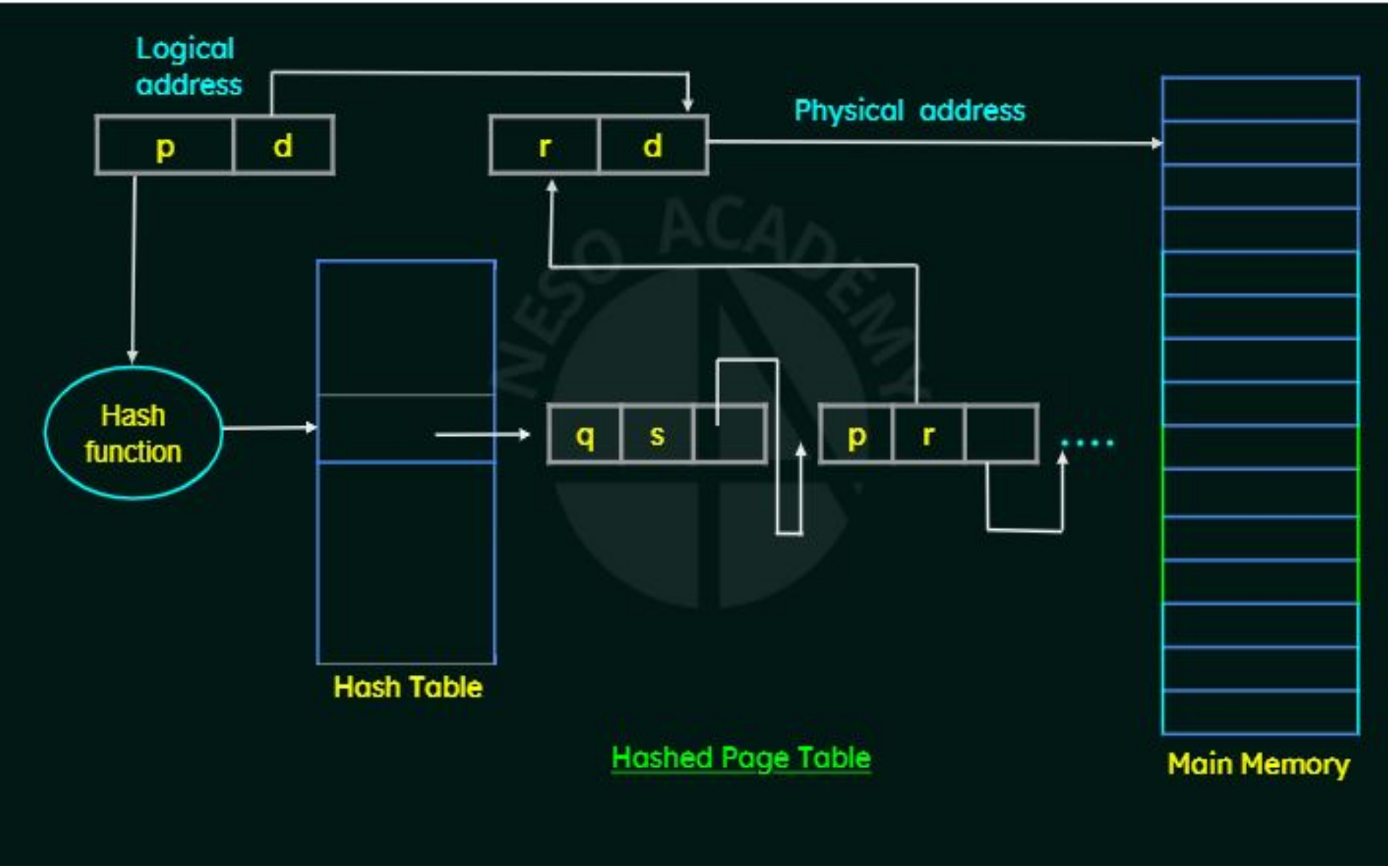
- The Hash value is the Virtual page number.
- Each entry in the hash table contains a linked list of elements that hash to the same location.

Each element consists of three fields:

Virtual page number	The value of the mapped page frame	A pointer to the next element in the linked list

The Algorithm:

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.


Logical address
p
d
r
d
Physical address
Hash function
q
s
p
r
....
Hash Table
Hashed Page Table
Main Memory

Clustered Page Tables

This a variation of the hashed page tables which is favorable for 64-bit address.

- It is similar to the hashed page tables except that each entry in the hash table refers to several pages rather than a single page.
- Therefore, a single page-table entry can store the mappings for multiple physical-page frames.
- Clustered page tables are particularly useful for sparse address spaces, where memory references are non-contiguous and scattered throughout the address space.

Inverted Page Tables

- In most Operating Systems, a separate page table is maintained for each processes.
- So, for 'n' number of processes, there will be 'n' number of page tables.
- For large processes, there would be many pages and for maintaining information about these pages, there would be too many entries in their page tables which itself would occupy a lot of the memory.
- Hence memory utilization is not efficient as a lot of memory is wasted in maintaining page tables itself.

Solution to this problem:
Use Inverted Page Tables

Inverted Page Tables

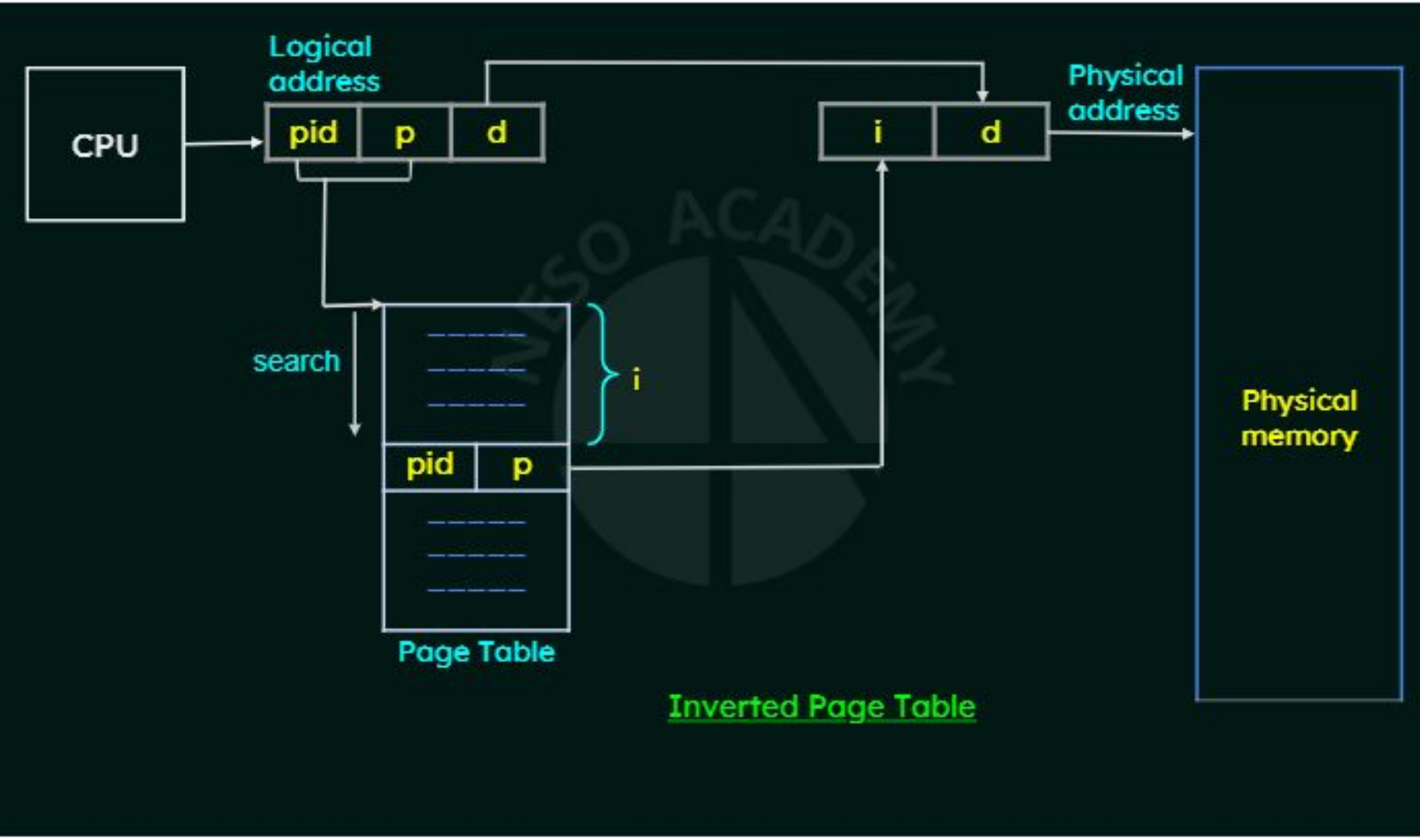
- An inverted page table has one entry for each real page (or frame) of memory.
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.
- Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

Each virtual address in the system consists of a triple

Each inverted page-table entry is a pair

The Working:

- When a memory reference occurs, part of the virtual address, consisting of <process-id, page number>, is presented to the memory subsystem.
- The inverted page table is then searched for a match.
- If a match is found—say at entry 'i', then the physical address <i, offset> is generated.
- If no match is found, then an illegal address access has been attempted.


Logical address
CPU
pid
p
d
i
d
Physical address
Physical memory
search
i
pid
p
Page Table
Inverted Page Table

Advantage of Inverted Page Tables:

- Reduces memory usage.

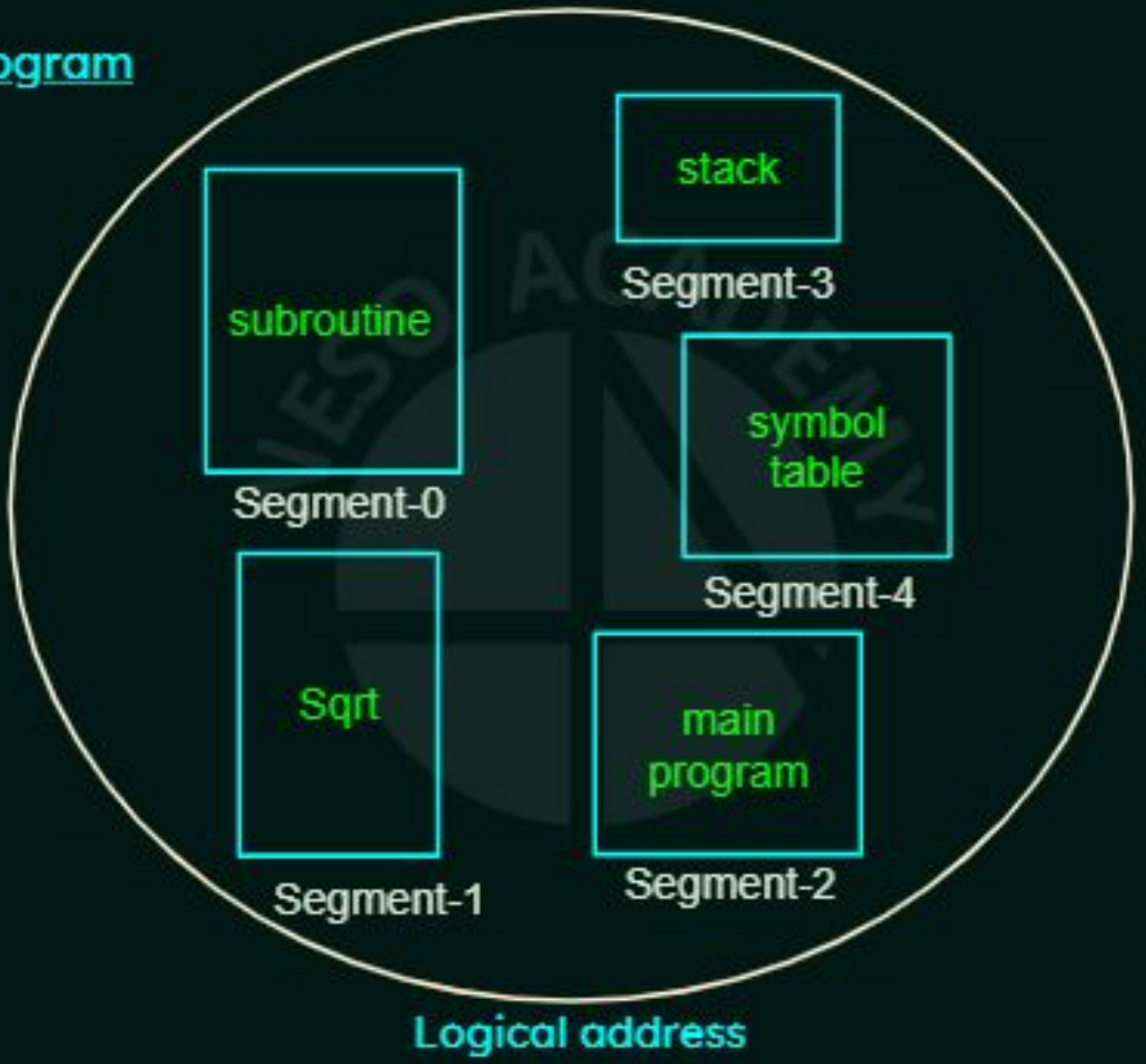
Disadvantage of Inverted Page Tables:

- Increased search time as inverted page table is sorted by physical address, but lookups occur on virtual addresses. The whole table might need to be searched for a match.
- Difficulty in implementing shared memory.

Segmentation

- Segmentation is another non-contiguous memory allocation technique like paging.
- Unlinke paging, in segmentation, the processes are not divided into fixed size pages.
- Instead, the processes are divided into several modules called segments which improves the visualization for the users.
- So, here, both secondary memory and main memory are divided into partitions of unequal sizes.

User's view of a program



- A logical address space is a collection of segments.
- Each Segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities: A Segment name and An Offset.

While in Paging - the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name.

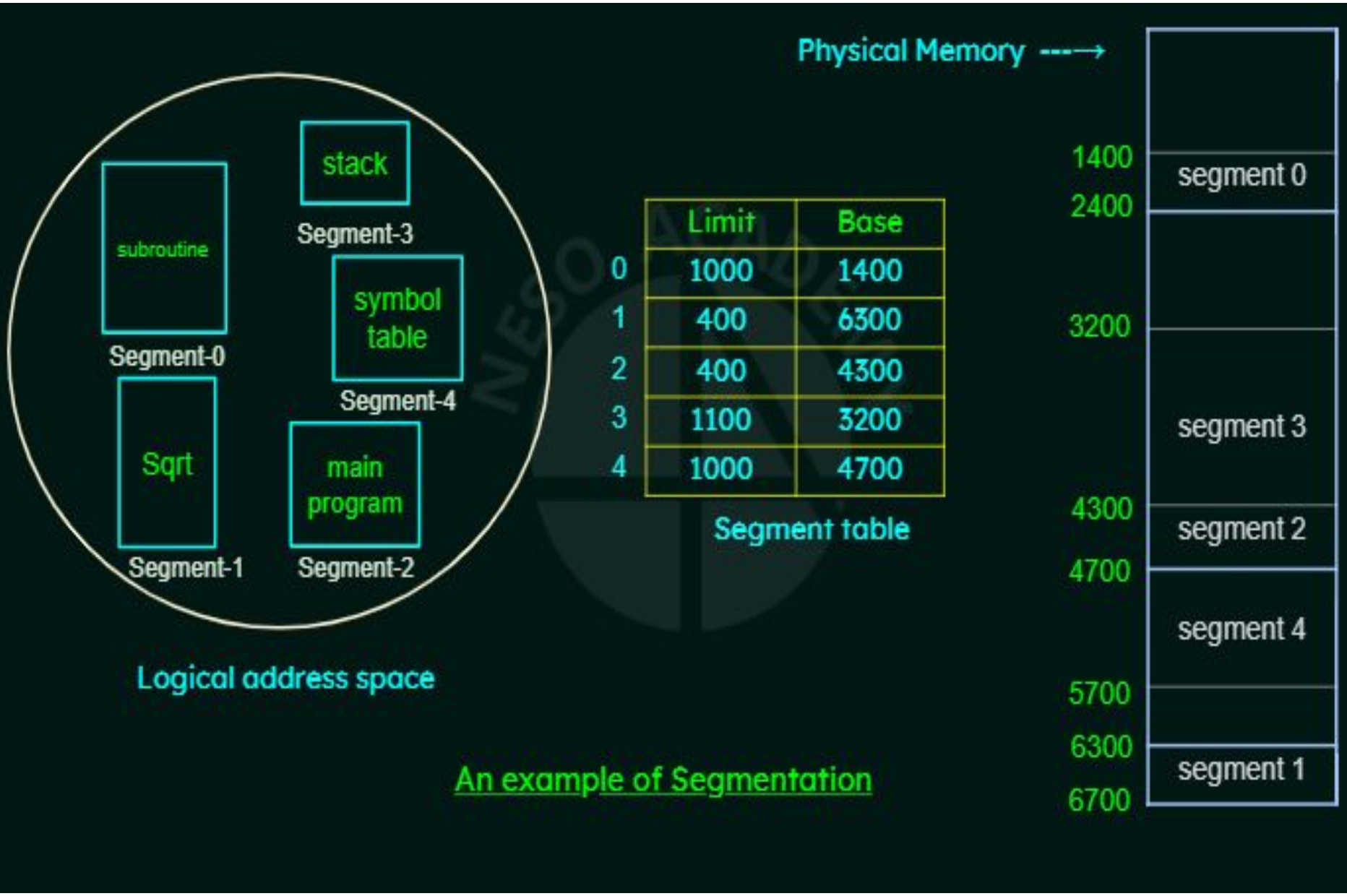
Thus, a logical address consists of a two tuple:

segment-number	offset

Segment Table

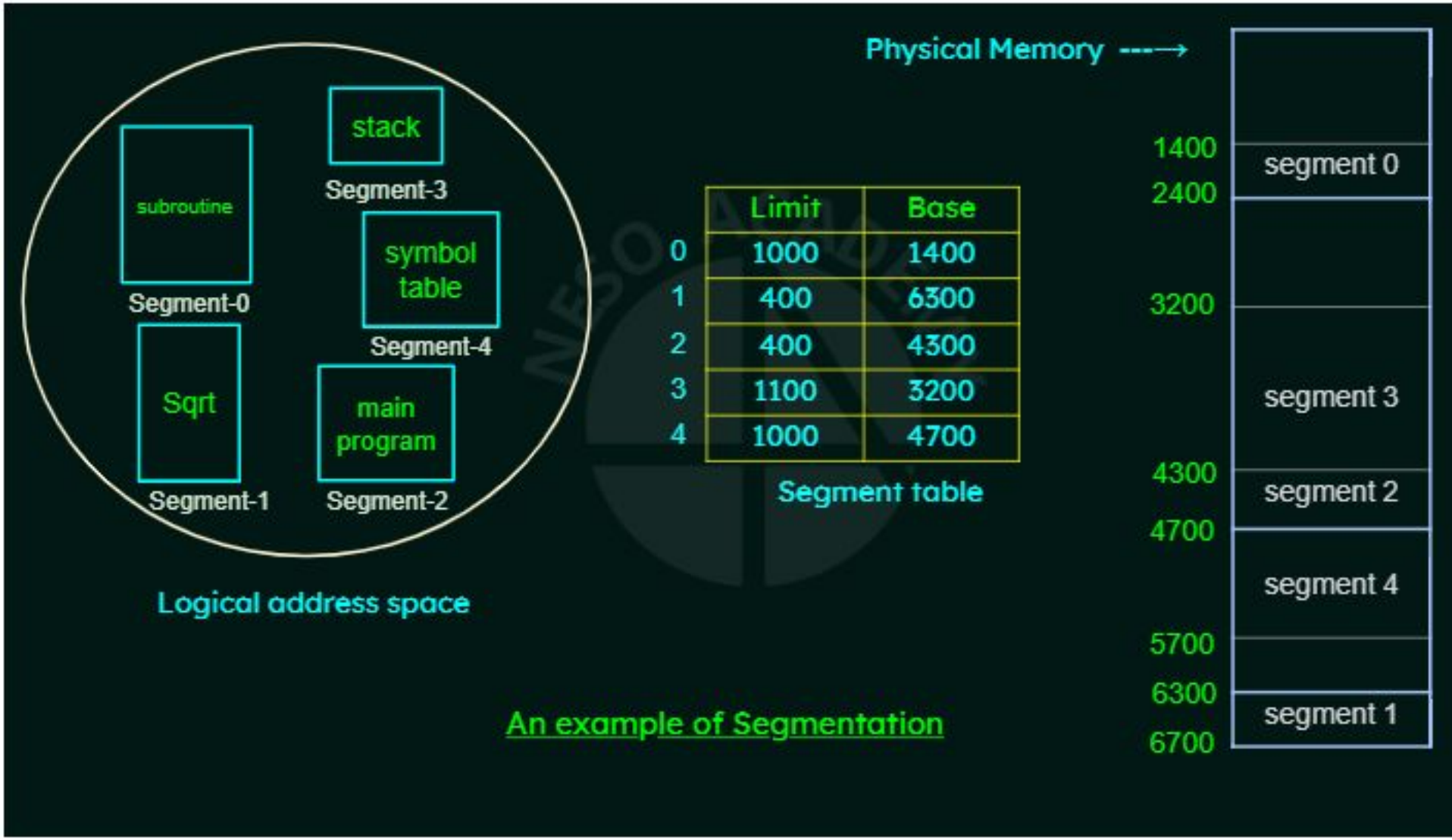
- We must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- This mapping is done by using a segment table.
- Each entry in the segment table has a segment base and a segment limit.
- The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

	Limit	Base
Segment-0	1000	1400
Segment-1	400	6300
Segment-2	400	4300
Segment-3	1100	3200
Segment-4	1000	4700

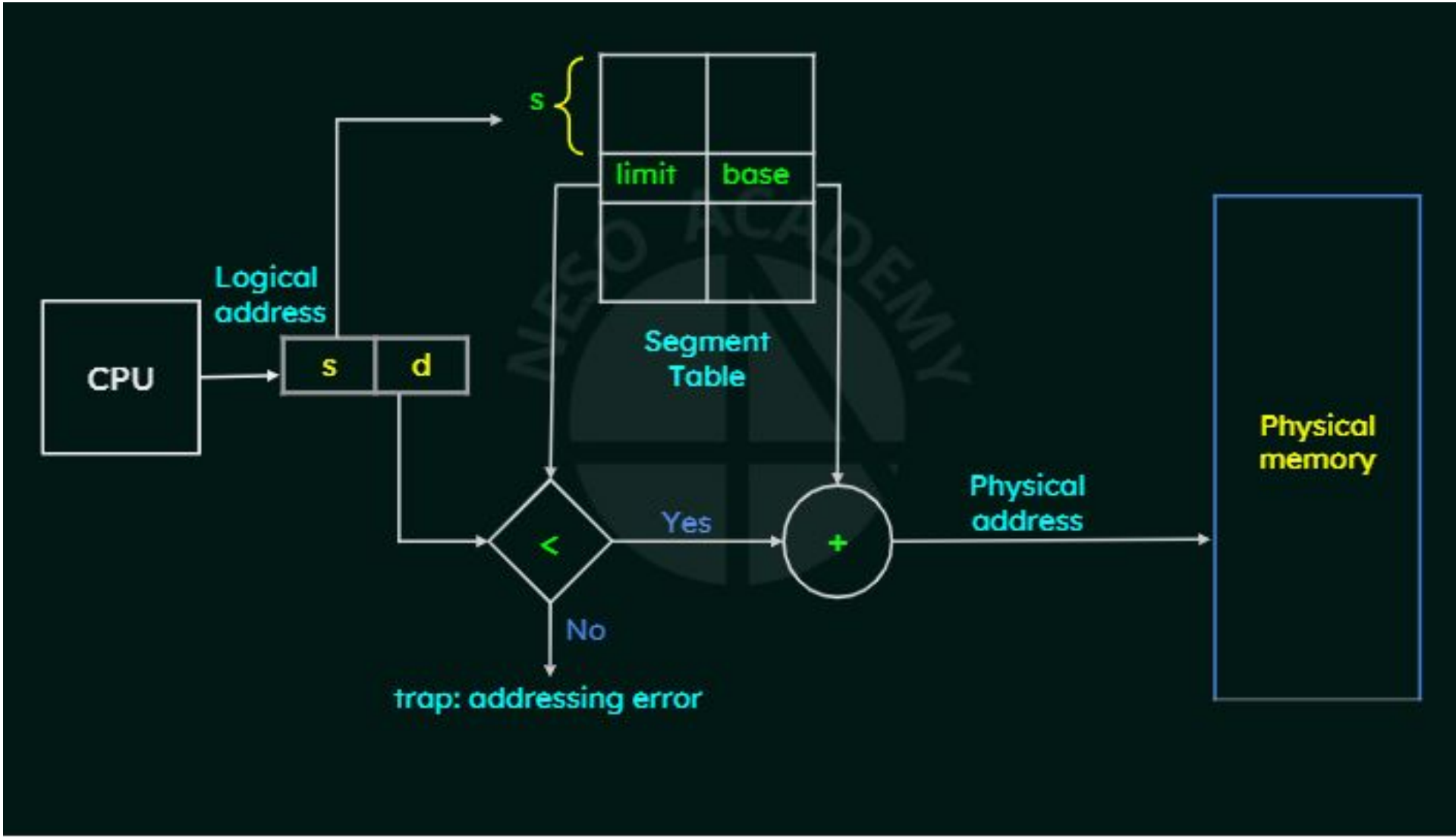

Physical Memory ---→
stack
Segment-3
subroutine
Segment-0
symbol table
Segment-4
Sqrt
Segment-1
main program
Segment-2
Logical address space
Limit
Base
0
1000
1400
1
400
6300
2
400
4300
3
1100
3200
4
1000
4700
Segment table
1400
2400
3200
4300
4700
5700
6300
6700
segment 0
segment 3
segment 2
segment 4
segment 1
An example of Segmentation

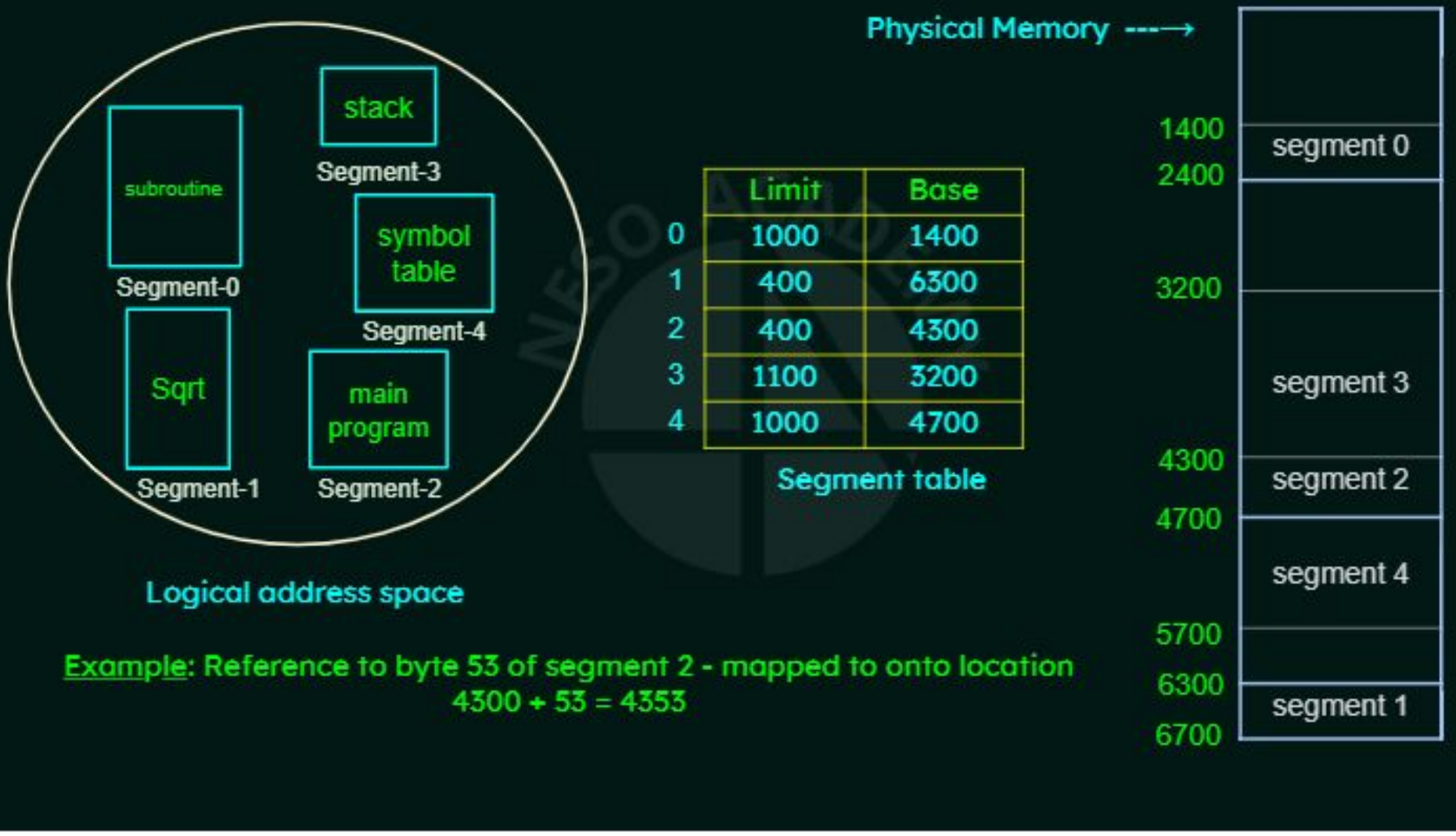
Segmentation Hardware

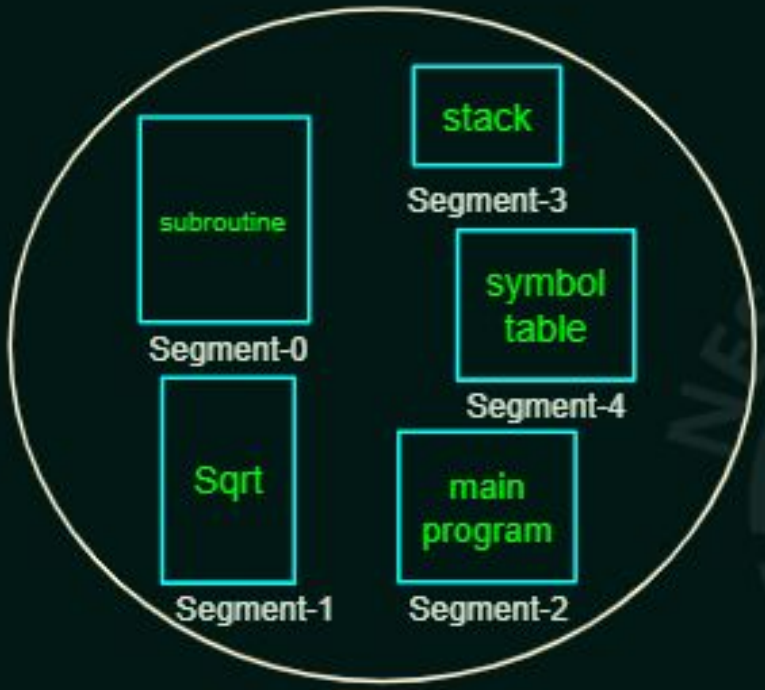
- Although in segmentation the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one-dimensional sequence of bytes.
- Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.
- This is done with the help of a segment table.


Physical Memory ---→
stack
Segment-3
subroutine
Segment-0
symbol table
Segment-4
Sqrt
Segment-1
main program
Segment-2
Logical address space
Limit
Base
0
1000
1400
1
400
6300
2
400
4300
3
1100
3200
4
1000
4700
Segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
An example of Segmentation

- A logical address consists of two parts:
  A segment number, 's', and an offset into that segment, 'd'.
- The segment number is used as an index to the segment table.
- The offset d of the logical address must be between 0 and the segment limit.
- If it is not, we trap to the operating system (logical addressing attempt beyond, end of segment).
- When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte.


s
limit
base
Segment Table
Logical address
CPU
s
d
<
Yes
+
Physical address
Physical memory
No
trap: addressing error

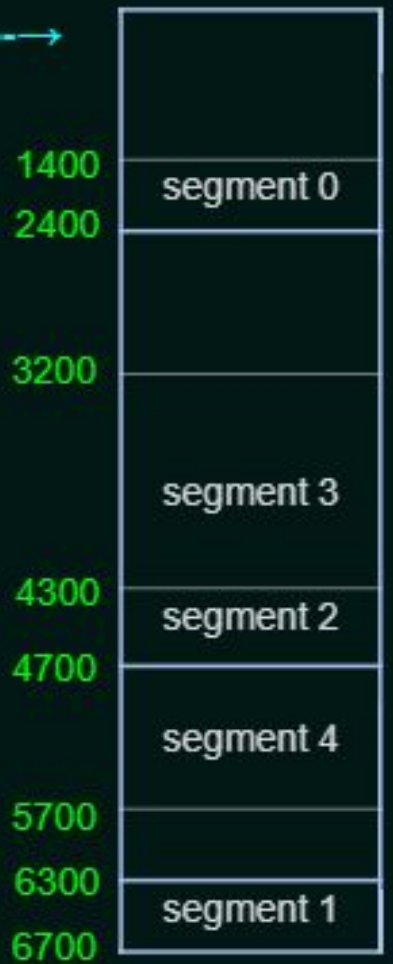

Physical Memory ---→
stack
Segment-3
subroutine
Segment-0
symbol table
Segment-4
Sqrt
Segment-1
main program
Segment-2
Logical address space
Limit
Base
0
1000
1400
1
400
6300
2
400
4300
3
1100
3200
4
1000
4700
Segment table
1400
segment 0
2400
3200
segment 3
4300
segment 2
4700
segment 4
5700
6300
segment 1
6700
Example: Reference to byte 53 of segment 2 - mapped to onto location
4300 + 53 = 4353



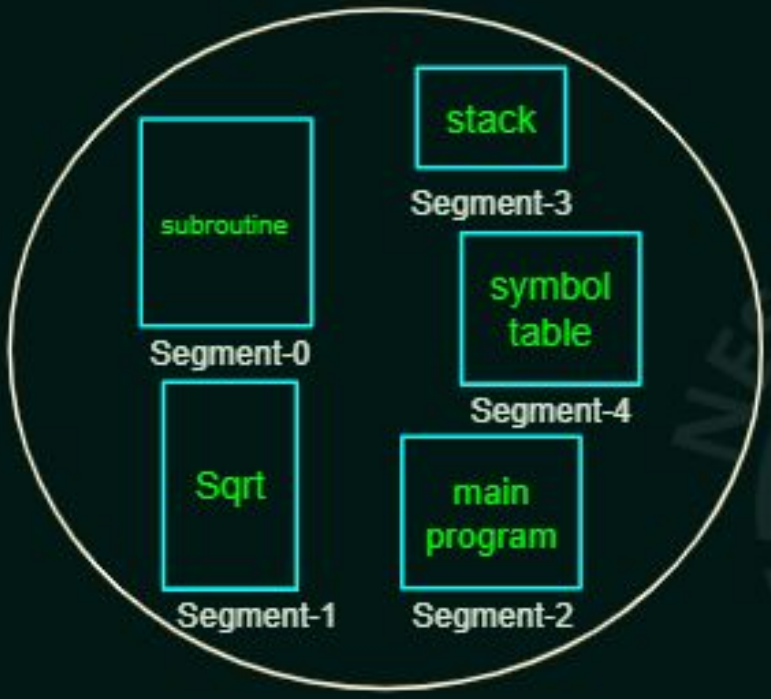
Logical address space

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table



Example: Reference to byte 852 of segment 3 - mapped to onto location 3200 + 852 = 4052

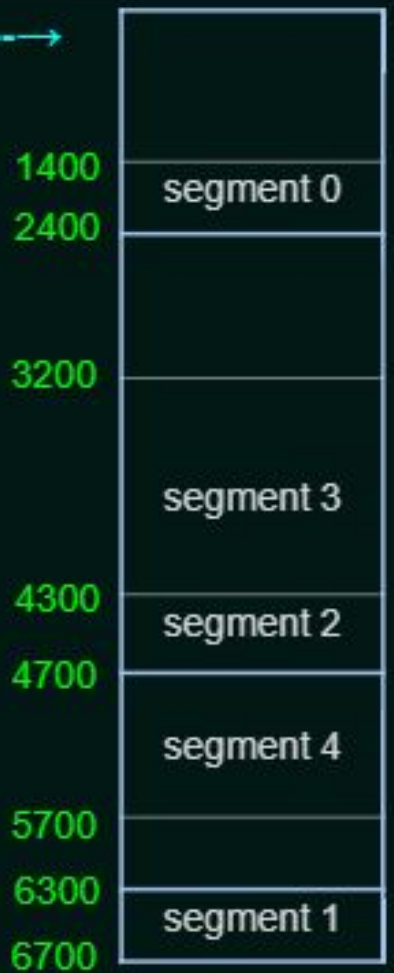


Logical address space

	Limit	Base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Segment table

Physical Memory ---→



Example: Reference to byte 1222 of segment 0 - would result in a trap to the operating system, as this segment is only 1,000 bytes long.

Main Memory

Solved Problem - 1

GATE 2013

A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE).
The PTE is 32 bits in size. The processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. The cache block size is 64 bytes.

What is the size of a page in KB in this computer?

(A) 2
(B) 4
(C) 8
(D) 16

A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE).
The PTE is 32 bits in size. The processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. The cache block size is 64 bytes.

Given,

- Logical address size = 2^{46} bytes
- 3 level multilevel paging is used with 3 page tables T1, T2, T3
- Page table entry size = 32 bits = 4 bytes [8 bits = 1 byte]

We need to find the Page Size in this system.
Note: Page size will be equal to size of third-level table T3.

Given,

- Logical address size = 2^{46} bytes
- 3 level multilevel paging is used with 3 page tables T1, T2, T3
- Page table entry (PTE) size = 32 bits = 4 bytes [8 bits = 1 byte]

T1

Let Page Size be of 'x' bits

Page Size = 2^x bytes

$$\text{Number of entries in T1} = \frac{\text{Logical address size}}{\text{Page size}} = \frac{2^{46} \text{ bytes}}{2^x \text{ bytes}} = 2^{46-x}$$

Size of T1 = No. of entries in page table x PTE size

$= (2^{46-x}) * (4 \text{ bytes}) = (2^{46-x}) * (2^2 \text{ bytes}) = 2^{48-x}$ bytes

Given,

- Logical address size = 2^{46} bytes
- 3 level multilevel paging is used with 3 page tables T1, T2, T3
- Page table entry (PTE) size = 32 bits = 4 bytes [8 bits = 1 byte]

T2

Page Size = 2^x bytes

Size of T1 = 2^{48-x} bytes

$$\text{Number of entries in T2} = \frac{\text{Size of T1}}{\text{Page size}} = \frac{2^{48-x} \text{ bytes}}{2^x \text{ bytes}} = 2^{48-2x}$$

Given,

- Logical address size = 2^{46} bytes
- 3 level multilevel paging is used with 3 page tables T1, T2, T3
- Page table entry (PTE) size = 32 bits = 4 bytes [8 bits = 1 byte]

T3

Page Size = 2^x bytes

Size of T1 = 2^{48-x} bytes

Size of T2 = 2^{50-2x} bytes

$$\text{Number of entries in T3} = \frac{\text{Size of T2}}{\text{Page size}} = \frac{2^{50-2x} \text{ bytes}}{2^x \text{ bytes}} = 2^{50-3x}$$

Size of T3 = No. of entries in page table x PTE size

= (2^{50-3x}) * (4 bytes) = (2^{50-3x}) * $(2^2$ bytes) = 2^{52-3x} bytes

Given,

- Logical address size = 2^{48} bytes
- 3 level multilevel paging is used with 3 page tables T1, T2, T3
- Page table entry (PTE) size = 32 bits = 4 bytes [8 bits = 1 byte]

Page Size = 2^{x} bytes

Size of T1 = 2^{48-x} bytes

Size of T2 = 2^{50-2x} bytes

Size of T3 = 2^{52-3x} bytes

Page size will be equal to size of third-level table T3

Size of T3 = Page size

2^{52-3x} bytes = 2^{x} bytes

$52-3x = x$

$4x = 52$

$x = \frac{52}{4} = 13$ bits. $\therefore$ Page size = 2^{13} = 8129 bytes = $\frac{8129}{1024}$ = 8 KB

Main Memory

Solved Problem - 1

GATE 2013

A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE).
The PTE is 32 bits in size. The processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. The cache block size is 64 bytes.

What is the size of a page in KB in this computer?

(A) 2
(B) 4
(C) 8
(D) 16

Main Memory

Solved Problem - 2

GATE 2013

A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE).
The PTE is 32 bits in size. The processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. The cache block size is 64 bytes.

What is the minimum number of page colors needed to guarantee that no two synonyms map to different sets in the processor cache of this computer?

(A) 2
(B) 4
(C) 8
(D) 16

A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE).
The PTE is 32 bits in size. The processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. The cache block size is 64 bytes.

Given,

- Cache size = 1 MB
- Uses 16 way set associative virtually indexed physically tagged cache
- Cache block size = 64 bytes
- Page size = 8 KB [from previous solution]

Given,

- Cache size = 1 MB
- Uses 16 way set associative virtually indexed physically tagged cache
- Cache block size = 64 bytes
- Page size = 8 KB [from previous solution]

$$\text{Number of pages in cache} = \frac{\text{Cache size}}{\text{Page size}} = \frac{1\text{ MB}}{8\text{ KB}} = \frac{2^{20}\ bytes}{2^{13}\ \text{bytes}} = 2^7 = 128$$

$$\text{Number of sets} = \frac{\text{Number of pages in cache}}{16} = \frac{128}{16} = 8$$

∴ Minimum number of page colors needed to guarantee that no two synonyms map to different sets in the processor cache = 8

Main Memory
Solved Problem - 2

GATE 2013

A computer uses 46-bit virtual address, 32-bit physical address, and a three-level paged page table organization. The page table base register stores the base address of the first-level table (T1), which occupies exactly one page. Each entry of T1 stores the base address of a page of the second-level table (T2). Each entry of T2 stores the base address of a page of the third-level table (T3). Each entry of T3 stores a page table entry (PTE).
The PTE is 32 bits in size. The processor used in the computer has a 1 MB 16 way set associative virtually indexed physically tagged cache. The cache block size is 64 bytes.

What is the minimum number of page colors needed to guarantee that no two synonyms map to different sets in the processor cache of this computer?

(A) 2
(B) 4
(C) 8
(D) 16

Main Memory

Solved Problem - 3

GATE 2015

A computer system implements a 40 bit virtual address, page size of 8 kilobytes, and a 128-entry translation look-aside buffer (TLB) organized into 32 sets each having four ways. Assume that the TLB tag does not store any process id.

The minimum length of the TLB tag in bits is _____.

(A) 20
(B) 10
(C) 11
(D) 22

A computer system implements a 40 bit virtual address, page size of 8 kilobytes, and a 128-entry translation look-aside buffer (TLB) organized into 32 sets each having four ways. Assume that the TLB tag does not store any process id.

Virtual address = 40 bits

Since TLB is organized into 32 sets, set offset = 5 $2^5 = 32$

Since page size = 8 KB, word offset = 13 8 KB = 8192 bytes = 2^{13}

Minimum length of the TLB tag = Total Logical address size – set offset – word offset
= 40 – 5 – 13 = 22 bits

Main Memory

Solved Problem - 3

GATE 2015

A computer system implements a 40 bit virtual address, page size of 8 kilobytes, and a 128-entry translation look-aside buffer (TLB) organized into 32 sets each having four ways. Assume that the TLB tag does not store any process id.

The minimum length of the TLB tag in bits is _____.

(A) 20
(B) 10
(C) 11
(D) 22

Main Memory

Solved Problem - 4

GATE 2015

Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each.

The size of the page table in the system in megabytes is_____.

(A) 2
(B) 4
(C) 8
(D) 16

Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each.

The size of the page table in the system in megabytes is_____.

Given,
Number of bits in logical address = 32 bits
Page size = 4KB
Page table entry size = 4 bytes

We have to find size of the page.

Given,

Number of bits in logical address = 32 bits

Page size = 4KB

Page table entry size = 4 bytes

We have to find size of the page.

Calculate the Process size:

Number of bits in logical address = 32 bits

So,

Process Size = 2^{32} bytes

= 4 GB

Given,
Number of bits in logical address = 32 bits
Page size = 4KB
Page table entry size = 4 bytes

We have to find size of the page.

We found:
Process Size = 2^{32} bytes = 4 GB
Number of Entries in Page Table = 2^{20} entries

Calculate the Number of Entries in Page Table:

$$\text{Number of Entries in Page Table} = \frac{\textit{Process size}}{\textit{Page size}}$$

$$= \frac{4\ GB}{4\ KB} = \frac{2^{32}\ bytes}{2^{12}\ bytes} = 2^{20} \text{ entries}$$

Given,
Number of bits in logical address = 32 bits
Page size = 4KB
Page table entry size = 4 bytes

We have to find size of the page.

We found:
Process Size = 2^{32} bytes = 4 GB
Number of Entries in Page Table = 2^{20} entries

Calculate the Page Table size:

Page table size = Number of entries in page table x Page table entry size

= 2^{20} * 4 bytes

= 2^{20} * 2^{2} bytes

= 2^{22} bytes = 4194304 bytes = $\frac{4194304}{1024 * 1024}$ = 4 MB

Main Memory

Solved Problem - 4

GATE 2015

Consider a system with byte-addressable memory, 32 bit logical addresses, 4 kilobyte page size and page table entries of 4 bytes each.

The size of the page table in the system in megabytes is_____.

(A) 2
(B) 4
(C) 8
(D) 16

Main Memory
Solved Problem - 5

GATE 2001

Consider a machine with 64 MB physical memory and a 32 bit virtual address space.

If the page size is 4 KB, what is the approximate size of the page table?

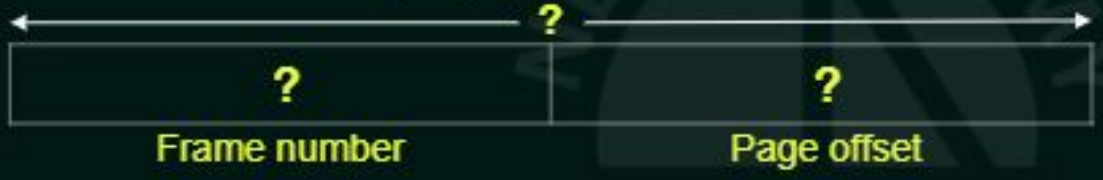
(A) 16 MB
(B) 8 MB
(C) 2 MB
(D) 24 MB

Consider a machine with 64 MB physical memory and a 32 bit virtual address space.

If the page size is 4 KB, what is the approximate size of the page table?

Approach:

Find the number of bits in physical address – frame number and page offset.



Calculate process size and with the help of that calculate the number of entries in page table.

Page Table Size = number of entries in page table x page table entry size (which is ≈ number of bits in frame number)

Consider a machine with 64 MB physical memory and a 32 bit virtual address space.

If the page size is 4 KB, what is the approximate size of the page table?

Given,
Size of Physical / Main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We have to find size of the page table.

Given,
Size of Physical / Main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We have to find size of the page table.

We found:
Number of bits in physical address = 26 bits

Calculate the number of bits in Frame Number:

$$\text{Number of frames in main memory} = \frac{\text{Size of main memory}}{\text{Frame size}} = \frac{64\ MB}{4\ KB} = \frac{2^{26}\ bytes}{2^{12}\ bytes}$$

$$= 2^{14}$$

So, Number of bits in frame number = 14 bits

Given,
Size of Physical / Main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We have to find size of the page table.

We found:
Number of bits in physical address = 26 bits
Number of bits in frame number = 14 bits

Calculate the number of bits in Page Offset :

Page Size = 4 KB = 2^{12} bytes

So, Number of bits in page offset = 12 bits

Given,
Size of Physical / Main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We have to find size of the page table.

We found:
Number of bits in physical address = 26 bits
Number of bits in frame number = 14 bits
Number of bits in page offset = 12 bits

Physical address:

Given,
Size of Physical / Main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We have to find size of the page table.

We found:
Number of bits in physical address = 26 bits
Number of bits in frame number = 14 bits
Number of bits in page offset = 12 bits

Calculate the Process Size:

Number of bits in virtual address space = 32 bits

So, Process Size = 2^{32} bytes
= 4 GB

Given,
Size of Physical / Main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We have to find size of the page table.

We found:
Number of bits in physical address = 26 bits
Number of bits in frame number = 14 bits
Number of bits in page offset = 12 bits
Process Size = 2^{32} bytes = 4 GB

Calculate the number of entries in Page Table:

$$\text{Number of entries in page table} = \frac{\textit{Process size}}{\textit{Page size}} = \frac{4\ GB}{4\ KB} = \frac{2^{32}\ bytes}{2^{12}\ bytes}$$

$$= 2^{20} \text{ pages}$$

Given,
Size of Physical / Main memory = 64 MB
Number of bits in virtual address space = 32 bits
Page size = 4 KB

We have to find size of the page table.

We found:
Number of bits in physical address = 26 bits
Number of bits in frame number = 14 bits
Number of bits in page offset = 12 bits
Process Size = 2^{32} bytes = 4 GB
Number of entries in page table = 2^{20} pages

Calculate the Size of Page Table:

Size of Page Table = Number of entries in page table x Page table entry size
= Number of entries in page table x Number of bits in frame number
= 2^{20} * 14 bits
= 2^{20} * 16 bits (approximating 14 bits ≈ 16 bits)
= 2^{20} * 2 bytes
= 2^{21} bytes = $\frac{2097152}{1024 * 1024}$ = 2 MB

Main Memory

Solved Problem - 5

GATE 2001

Consider a machine with 64 MB physical memory and a 32 bit virtual address space.

If the page size is 4 KB, what is the approximate size of the page table?

(A) 16 MB
(B) 8 MB
(C) 2 MB
(D) 24 MB

Main Memory

Solved Problem - 6

GATE 2006

For each of the four processes P1, P2, P3 and P4, the total size in kilobytes (KB) and the number of segments are given below.

The page size is 1 KB. The size of an entry in the page table is 4 bytes. The size of an entry in the segment table is 8 bytes. The maximum size of a segment is 256 KB. The paging method for memory management uses two-level paging, and its storage overhead is P. The storage overhead for the segmentation method is S. The storage overhead for the segmentation and paging method is T. What is the relation among the overheads for the different methods of memory management in the concurrent execution of the below four processes ?

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	8
P4	364	3

(A) $P < S < T$
(B) $S < P < T$
(C) $S < T < P$
(D) $T < S < P$

The page size is 1 KB. The size of an entry in the page table is 4 bytes. The size of an entry in the segment table is 8 bytes. The maximum size of a segment is 256 KB. The paging method for memory management uses two-level paging, and its storage overhead is P. The storage overhead for the segmentation method is S. The storage overhead for the segmentation and paging method is T. What is the relation among the overheads for the different methods of memory management in the concurrent execution of the below four processes ?

Given,
Page size = 1 KB
Page table entry size = 4 bytes
Segment table entry size = 8 bytes
Maximum Segment size = 256 KB

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	8
P4	364	3

Storage overhead for 2-level Paging = P
Storage overhead for Segmentation = S
Storage overhead for Segmentation & Paging = T

Find the relation between these

For 2-Level Paging: P

Given,

Page size = 1 KB

Page table entry size = 4 bytes

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	8
P4	364	3

Assuming that size of Page Table = Page Size = 1 KB = 1024 bytes

Number of entries in a Page table = $\frac{Size\ of\ Page\ table}{Page\ table\ entry\ size} = \frac{1024\ bytes}{4\ bytes} = 256$

Storage overhead for process P1 = 1 KB (for outer page table) + 1 KB for (inner page table) = 2 KB

Storage overhead for process P2 = 1 KB (for outer page table) + 1 KB for (inner page table) = 2 KB

Storage overhead for process P3 = 1 KB (for outer page table) + 1 KB for (inner page table) = 2 KB

Storage overhead for process P4 = 1 KB (for outer page table) + 2 KB for (2 inner page tables) = 3 KB

So, total overhead, P = (2 + 2 + 2 + 3) KB = 9 KB

For Segmentation: S

Given,

Segment table entry size = 8 bytes

Maximum Segment size = 256 KB

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	8
P4	364	3

Storage overhead = Segment table entry size x Number of segments

Storage overhead for process P1 = 8 bytes x 4 = 32 bytes

Storage overhead for process P2 = 8 bytes x 5 = 40 bytes

Storage overhead for process P3 = 8 bytes x 8 = 64 bytes

Storage overhead for process P4 = 8 bytes x 3 = 24 bytes

So, total overhead, **S** = (32 + 40 + 64 + 24) bytes = **160 bytes**

For Segmentation with Paging: T

Given,

Page size = 1 KB

Page table entry size = 4 bytes

Segment table entry size = 8 bytes

Maximum Segment size = 256 KB

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	8
P4	364	3

We have to apply segmentation & then paging.

Calculate Page table size: No. of entries in page table x page table entry size = 256 x 4 = 1024 bytes = 1 KB

Storage overhead = Storage overhead of segmentation + 1 KB overhead for Paging

Storage overhead for process P1 = (8 bytes x 4) + 1 KB = (32 + 1024) bytes = 1056 bytes

Storage overhead for process P2 = (8 bytes x 5) + 1 KB = (40 + 1024) bytes = 1064 bytes

Storage overhead for process P3 = (8 bytes x 8) + 1 KB = (64 + 1024) bytes = 1088 bytes

Storage overhead for process P4 = (8 bytes x 3) + 1 KB = (24 + 1024) bytes = 1048 bytes

So, total overhead, T = (1056 + 1064 + 1088 + 1048) bytes = 4256 bytes

The page size is 1 KB. The size of an entry in the page table is 4 bytes. The size of an entry in the segment table is 8 bytes. The maximum size of a segment is 256 KB. The paging method for memory management uses two-level paging, and its storage overhead is P. The storage overhead for the segmentation method is S. The storage overhead for the segmentation and paging method is T. What is the relation among the overheads for the different methods of memory management in the concurrent execution of the below four processes ?

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	8
P4	364	3

Storage overhead for 2-level Paging = P = 9 KB = 9216 bytes
Storage overhead for Segmentation = S = 160 bytes
Storage overhead for Segmentation & Paging = T = 4256 bytes

Find the relation between these

$$S < T < P$$

Main Memory

Solved Problem - 6

GATE 2006

For each of the four processes P1, P2, P3 and P4, the total size in kilobytes (KB) and the number of segments are given below.

The page size is 1 KB. The size of an entry in the page table is 4 bytes. The size of an entry in the segment table is 8 bytes. The maximum size of a segment is 256 KB. The paging method for memory management uses two-level paging, and its storage overhead is P. The storage overhead for the segmentation method is S. The storage overhead for the segmentation and paging method is T. What is the relation among the overheads for the different methods of memory management in the concurrent execution of the below four processes ?

Process	Total size (in KB)	Number of segments
P1	195	4
P2	254	5
P3	45	8
P4	364	3

(A) $P < S < T$
(B) $S < P < T$
(C) $S < T < P$
(D) $T < S < P$

Thank you