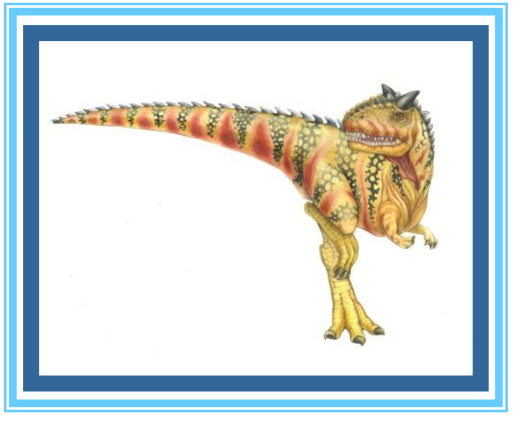
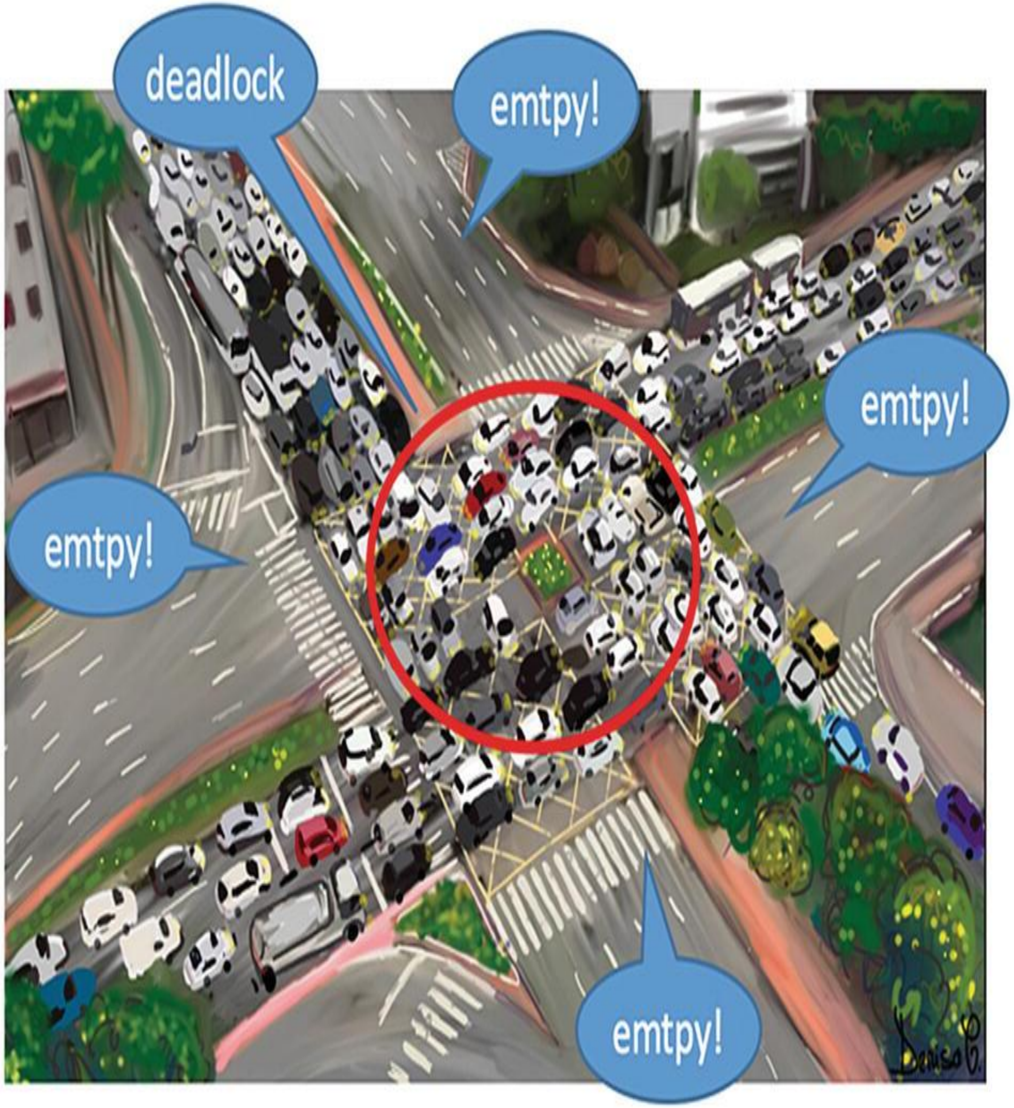
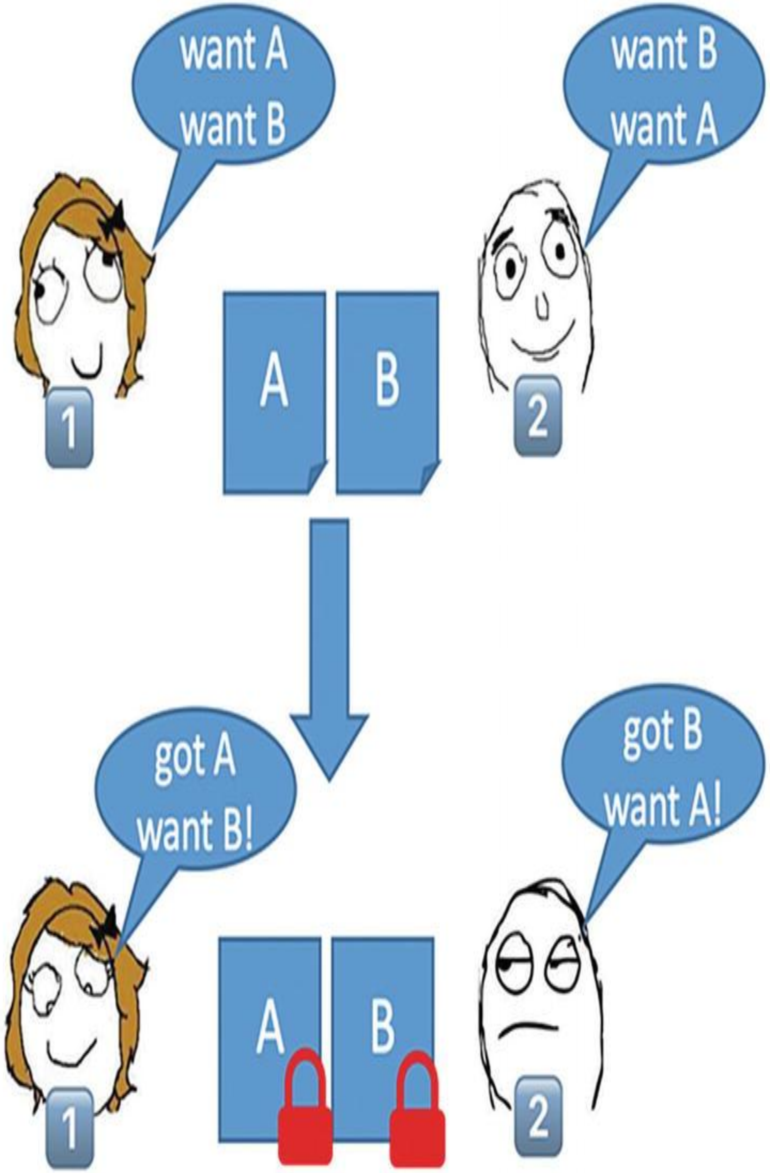
Chapter 7: Deadlocks





(a) Deadlock in real life

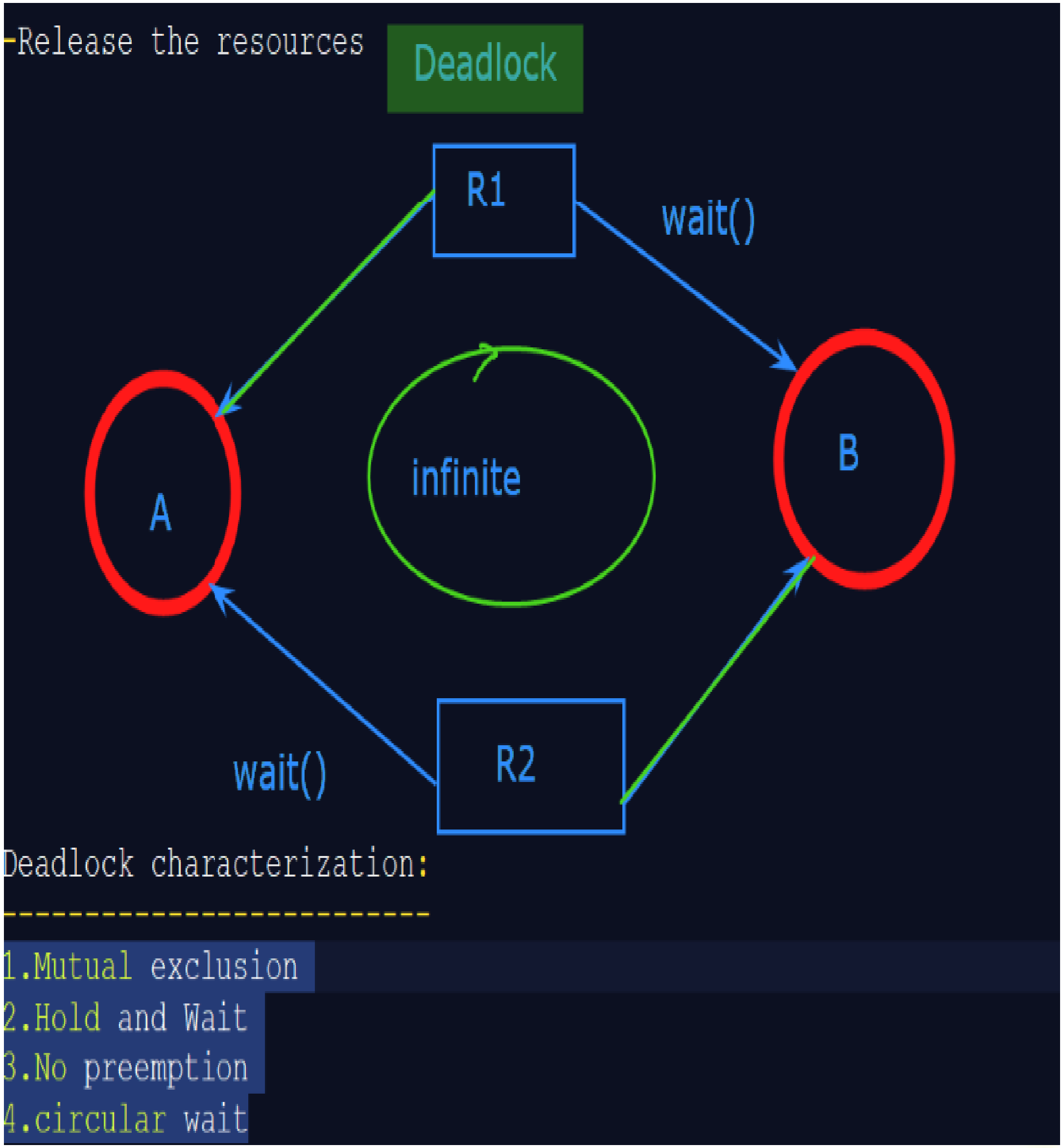


(b) Deadlock in virtual life

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**



System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion:** only one process at a time can use a resource
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Deadlock:

DEadlock is a situation where a set of process are blocked because each process is holding a resource and waiting for another resource acquired by some other process

-common problem: multiprocessing system

-System model:

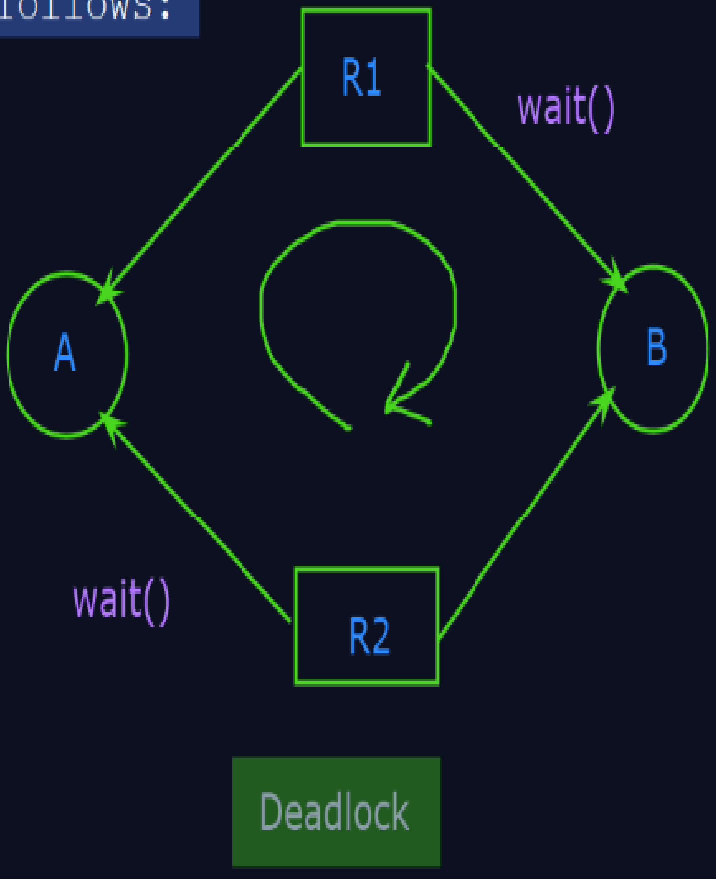
-Resources: R1, R2, R3...Rn

-Each resource utilize a resource as follows:

-1. Request a resource

-2. Use a resource

-3. Release that resource



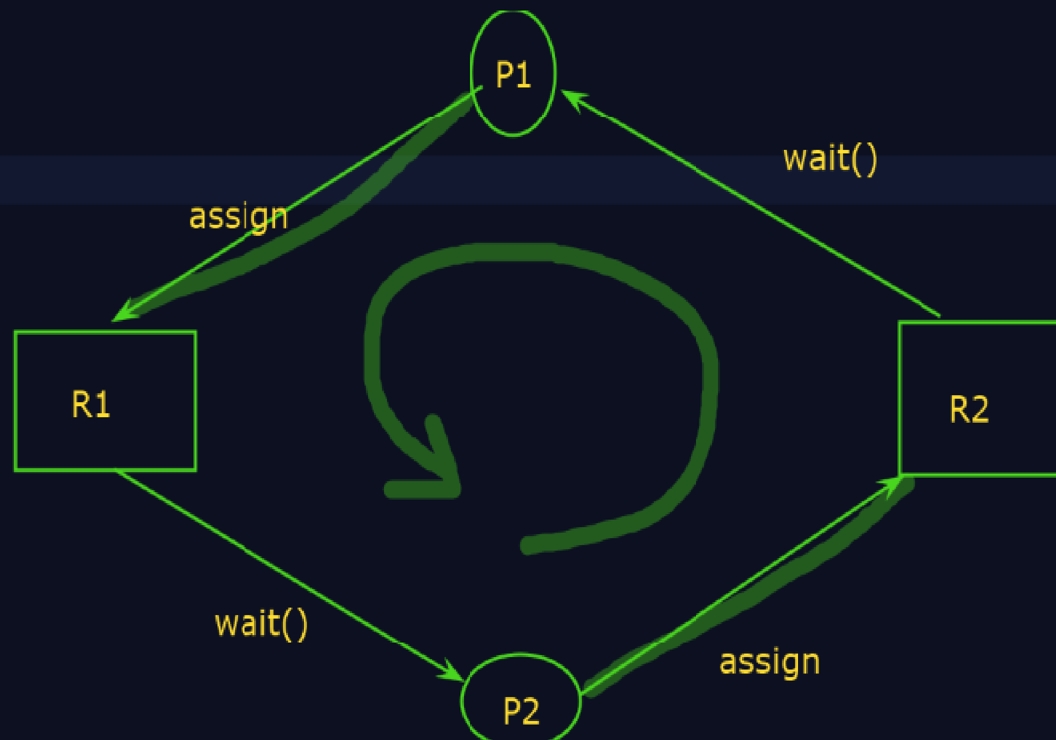
DEadlock characterization:

1. Mutual Exclusion
2. Hold and Waiting
3. No Preemption
4. Circular wait

Mutual Exclusion

DEadlock characterization:

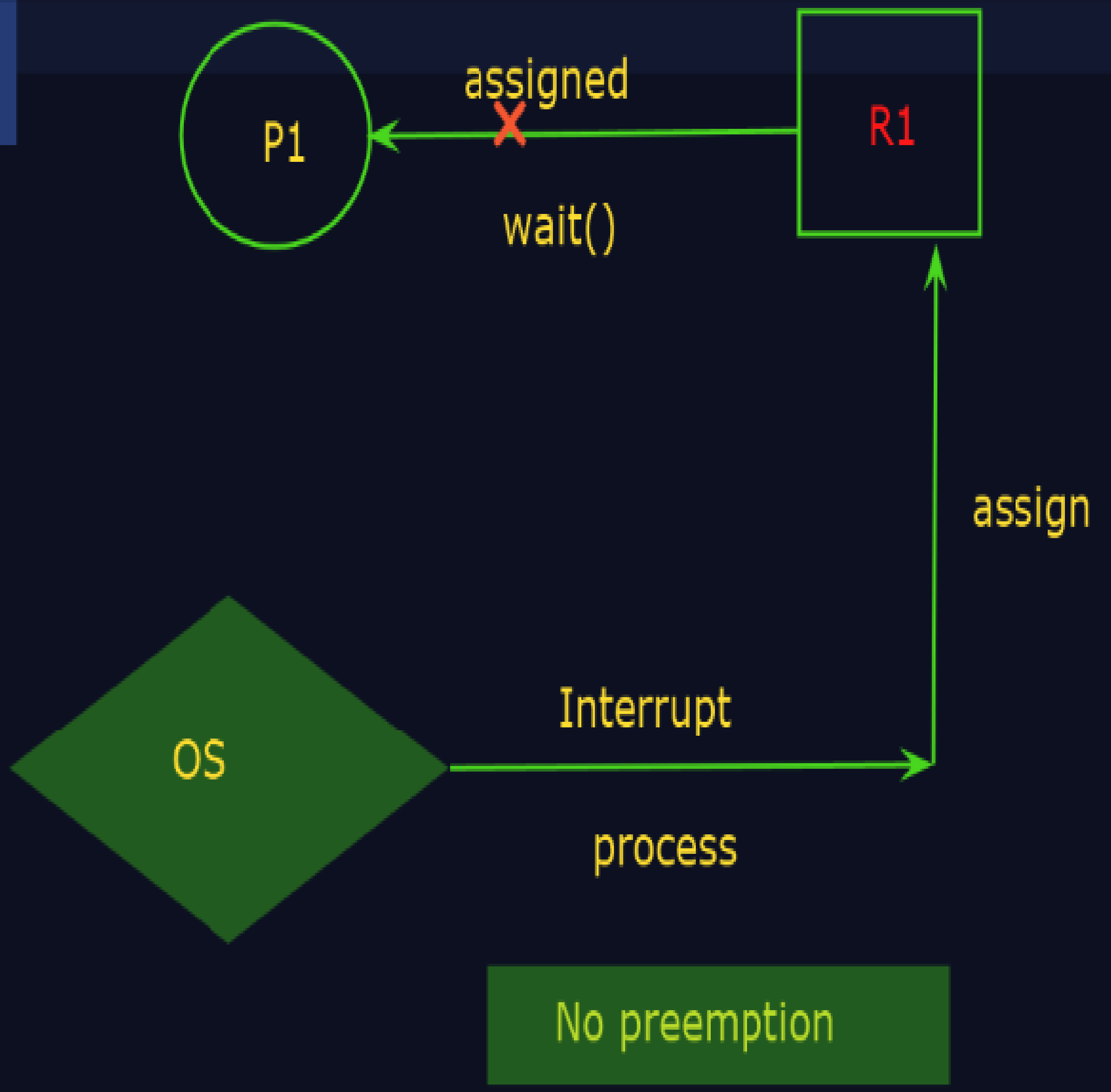
1. Mutual Exclusion
2. Hold and Waiting
3. No Preemption
4. Circular wait



Circular Wait

DEadlock characterization:

1. Mutual Exclusion
2. Hold and Waiting
3. No Preemption
4. Circular wait



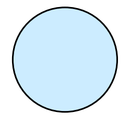
Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

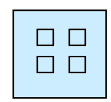
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

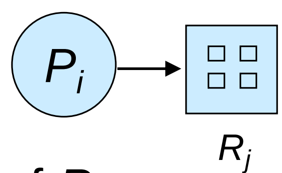
- Process



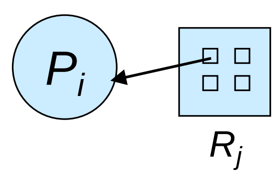
- Resource Type with 4 instances

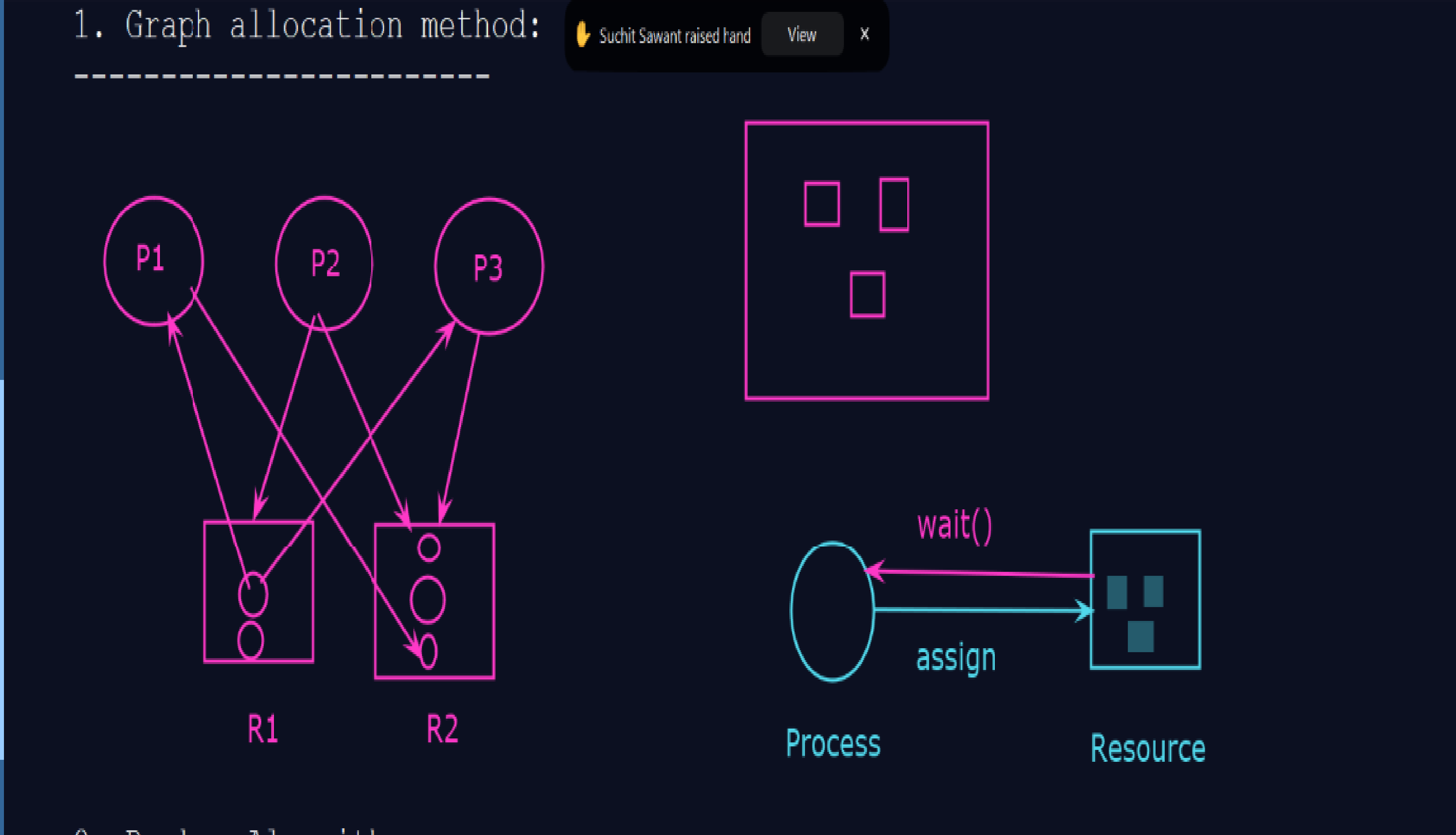


- P_i requests instance of R_j

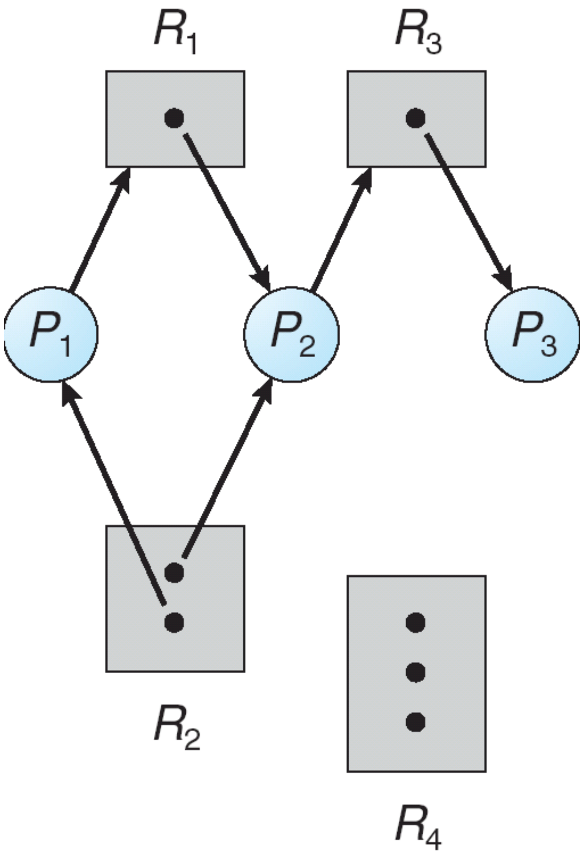


- P_i is holding an instance of R_j

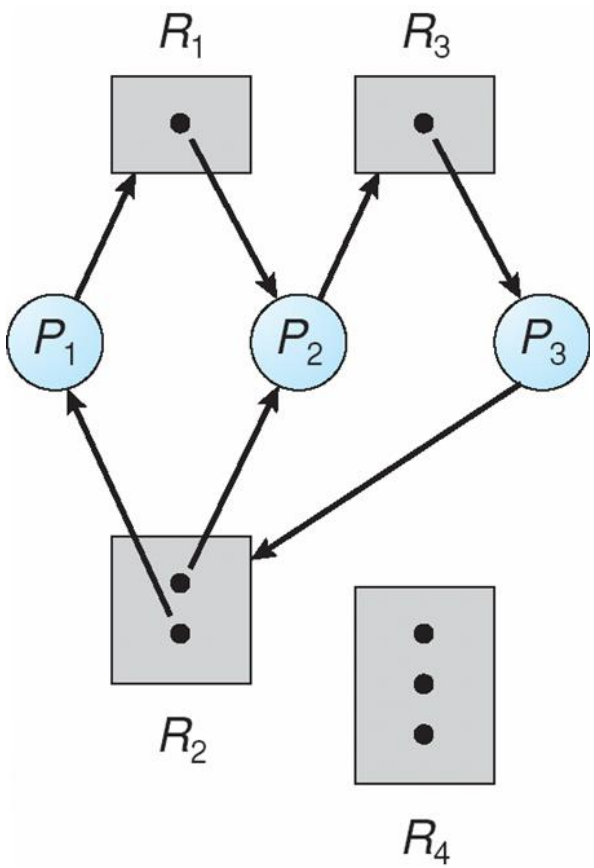



1. Graph allocation method:
Suchit Sawant raised hand
View
P1
P2
P3
R1
R2
wait()
assign
Process
Resource

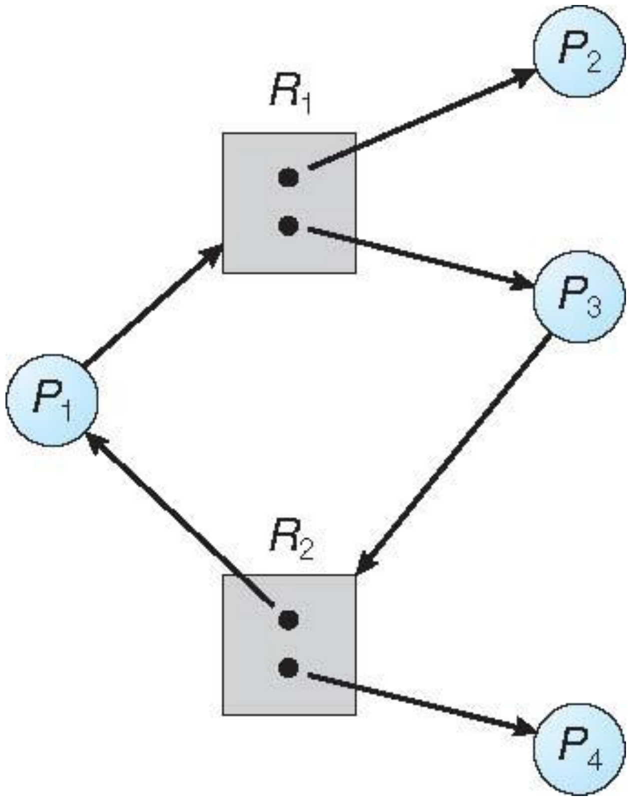
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidence
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

Requires that the system has some additional ***a priori*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

Basic Facts

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

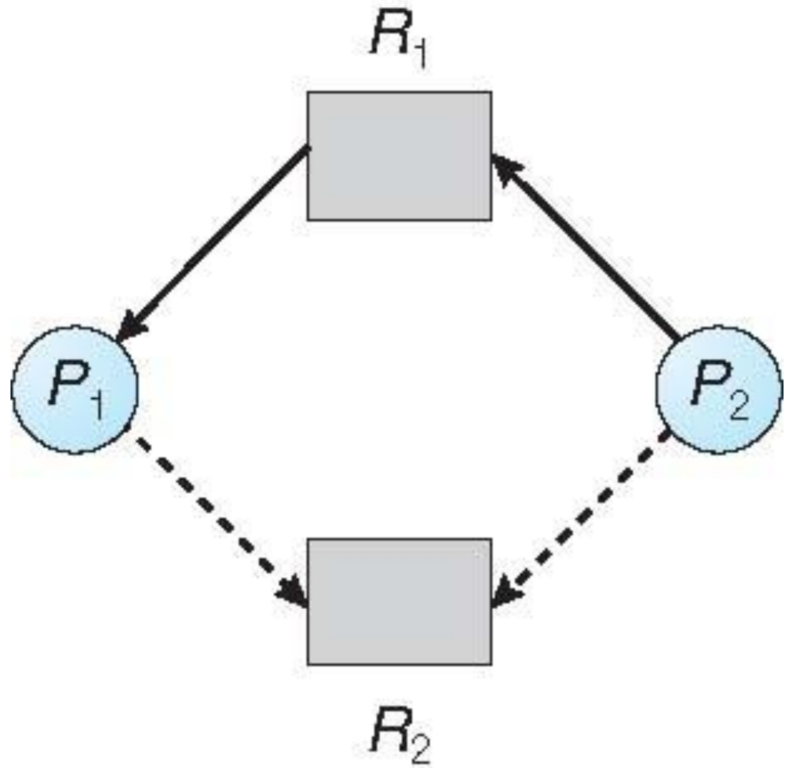
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

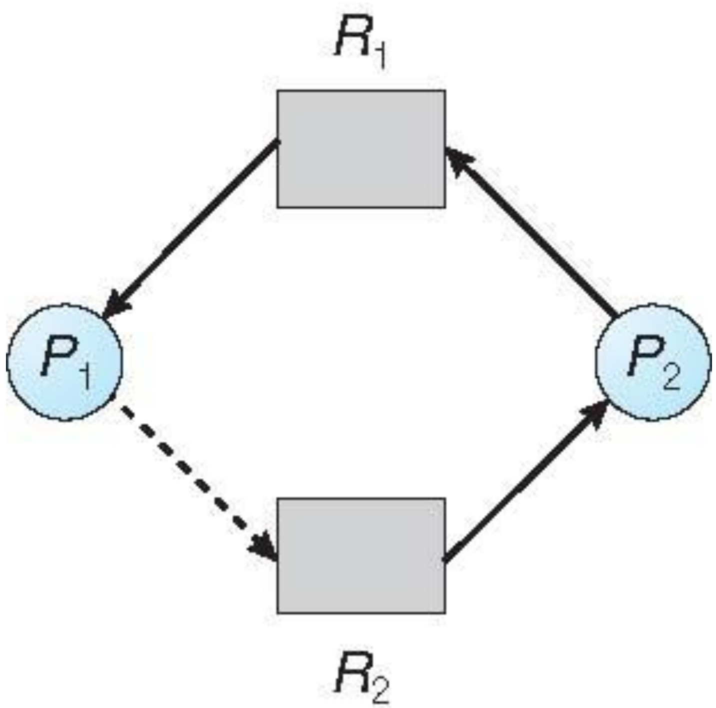
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Unsafe State In Resource-Allocation Graph



Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker’s Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available***:* Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max***:* n x m matrix. If $Max[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation***:* n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need***:* n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), *B* (5instances), and *C* (7 instances)

- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	<u>*Need*</u>
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_2, P_0> satisfies safety criteria

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

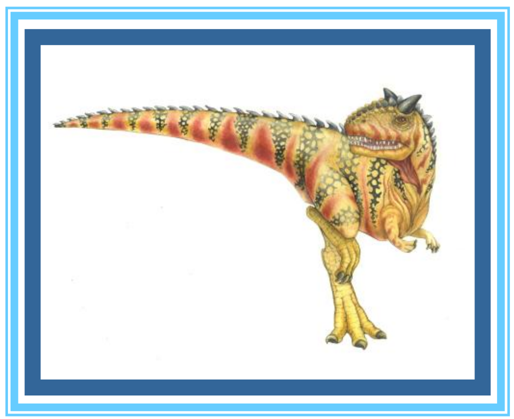
	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < $\boldsymbol{P_1}$, $\boldsymbol{P_3}$, $\boldsymbol{P_4}$, $\boldsymbol{P_0}$, $\boldsymbol{P_2}$> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

Deadlock Detection

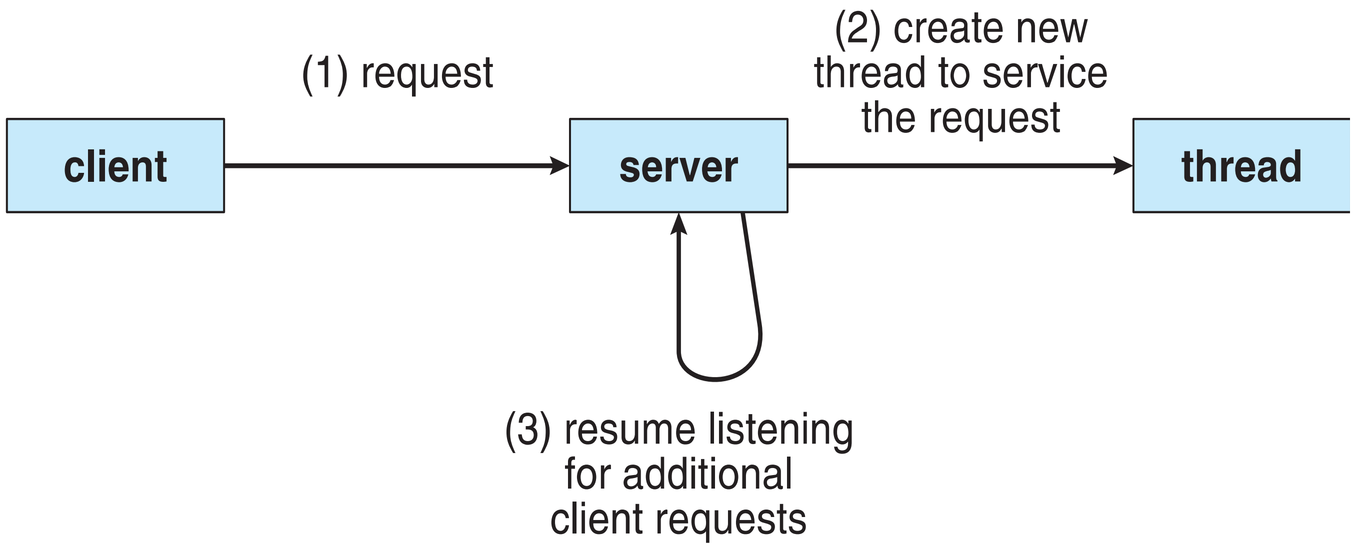
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

End of Chapter 7



Threads

Multithreaded Server Architecture



Benefits

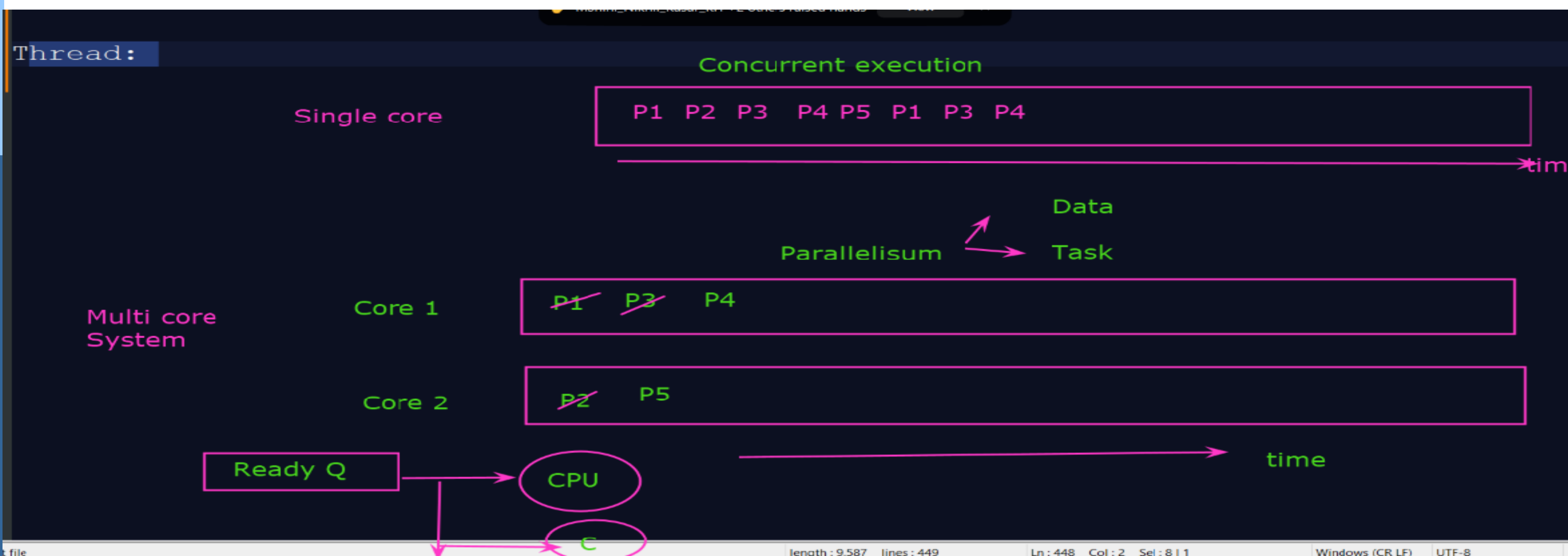
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

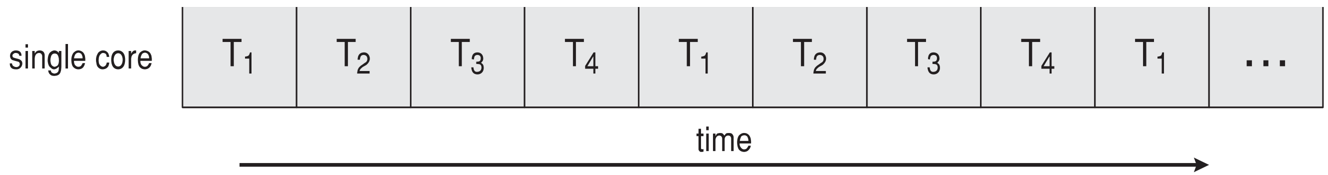
Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

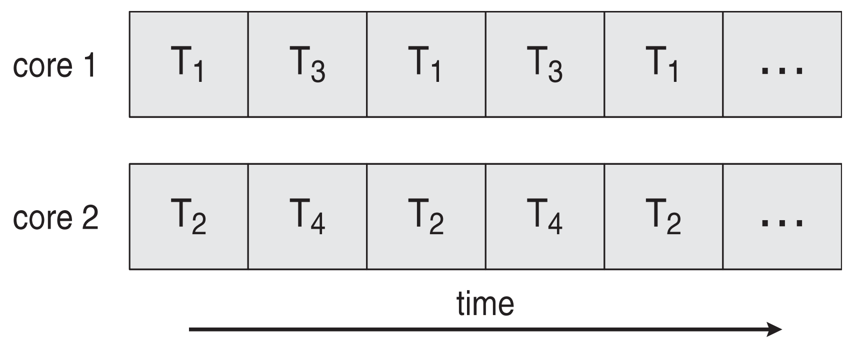


Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



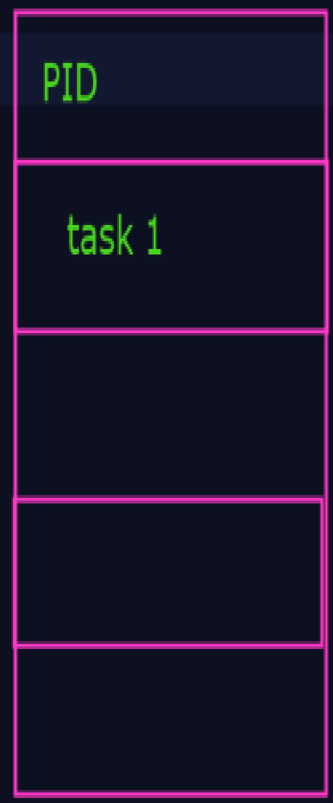
Process	Thread
A Process simply means any program in execution.	Thread simply means a segment of a process.
The process consumes more resources	Thread consumes fewer resources.
The process requires more time for creation.	Thread requires comparatively less time for creation than process.
The process is a heavyweight process	Thread is known as a lightweight process
The process takes more time to terminate	The thread takes less time to terminate.
Processes have independent data and code segments	A thread mainly shares the data segment, code segment, files, etc. with its peer threads.
The process takes more time for context switching.	The thread takes less time for context switching.
Communication between processes needs more time as compared to thread.	Communication between threads needs less time as compared to processes.
For some reason, if a process gets blocked then the remaining processes can continue their execution	In case if a user-level thread gets blocked, all of its peer threads also get blocked.

Thread:

A thread is a basic unit of CPU utilization.
It consist of:

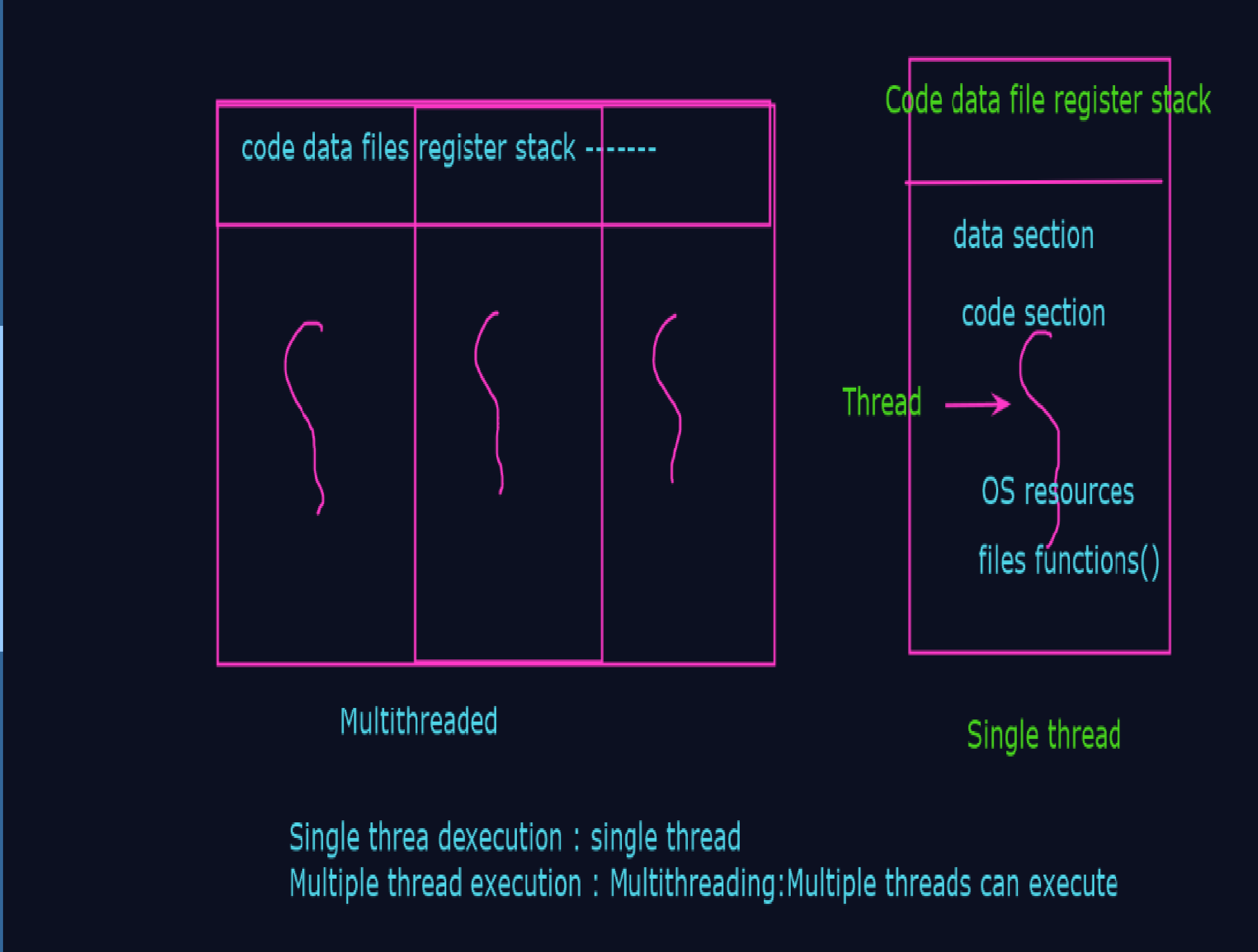
- -Thread ID
- -Program counter
- -Registers
- -Stack

Program --> Process ---> thread



Single thread

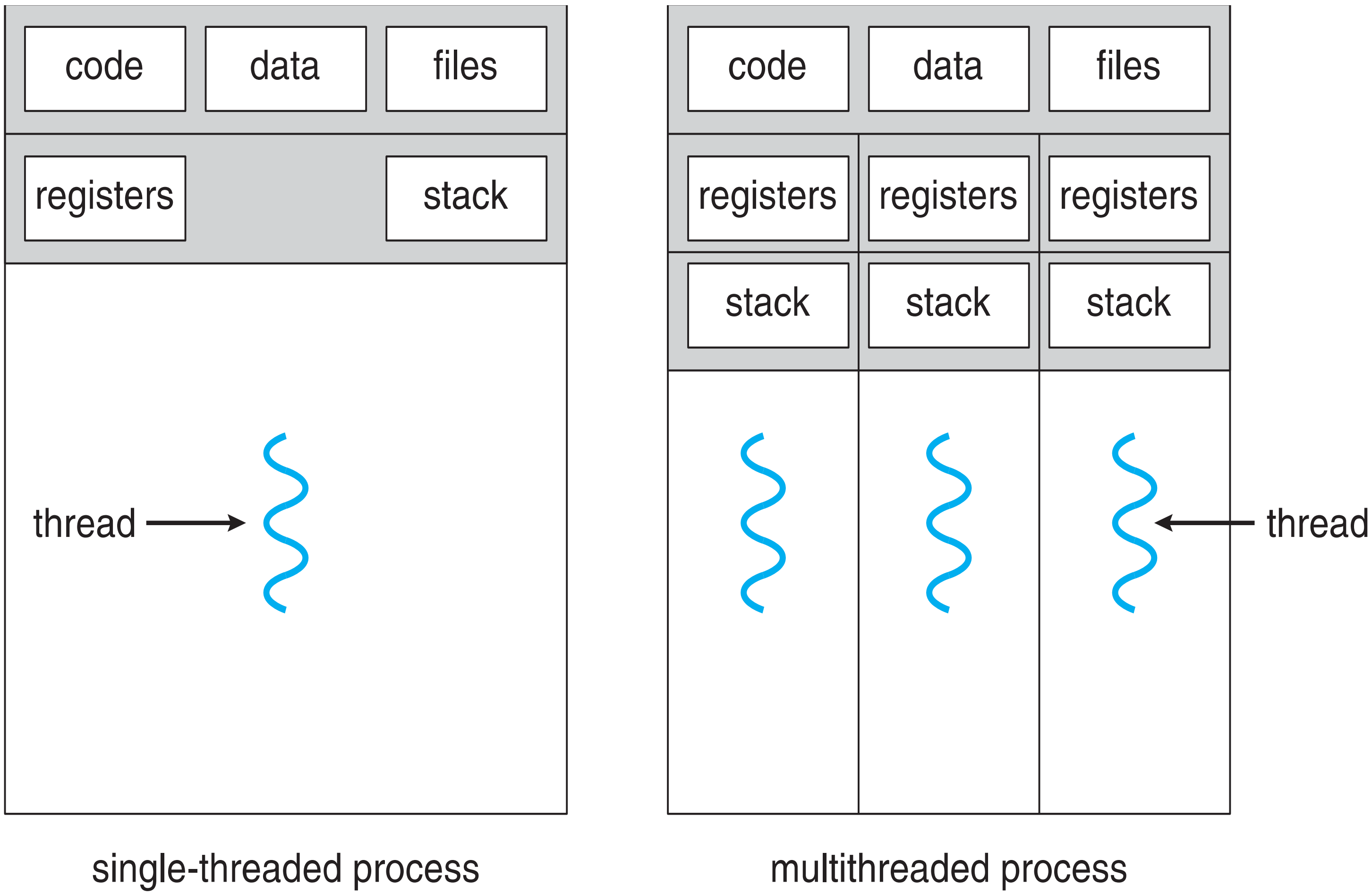
Single threa dexecution : single thread
Multiple thread execution : Multithreading:Multiple threads can execute


code data files register stack --------
Multithreaded
Code data file register stack
data section
code section
Thread
OS resources
files functions()
Single thread
Single threa dexecution : single thread
Multiple thread execution : Multithreading:Multiple threads can execute

What is Thread?

- Thread is an **execution unit** that consists of its own program counter, a stack, and a set of registers where the program **counter mainly keeps track of which instruction to execute next**, a set of registers mainly hold its current working variables, and a stack mainly contains the history of execution.
- Threads are also known as **Lightweight processes.**
- Threads are a popular **way to improve the performance of an application** through parallelism.
- Threads are **mainly used to represent a software approach** in order to improve the performance of an operating system just by reducing the overhead thread that is mainly equivalent to a classical process.

Single and Multithreaded Processes



Advantages of Thread

- Some advantages of thread are given below:

1. Responsiveness
2. Resource sharing, hence allowing **better utilization of resources.**
3. Economy. Creating and managing threads becomes easier.
4. Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.
5. Context Switching is smooth. Context switching refers to the procedure followed by the CPU to change from one task to another.
6. Enhanced Throughput of the system. Let us take an example for this: suppose a process is divided into multiple threads, and the function of each thread is considered as one job, then the number of jobs completed per unit of time increases which then leads to an increase in the throughput of the system.

Types of Thread

- There are two types of threads:
  - User Threads
  - Kernel Threads
- User threads are above the kernel and without kernel support. These are the **threads that application programmers use in their programs**.
- Kernel threads are **supported within the kernel of the OS itself**. All modern OSs support kernel-level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- But does the law take into account contemporary multicore systems?

User Threads and Kernel Threads

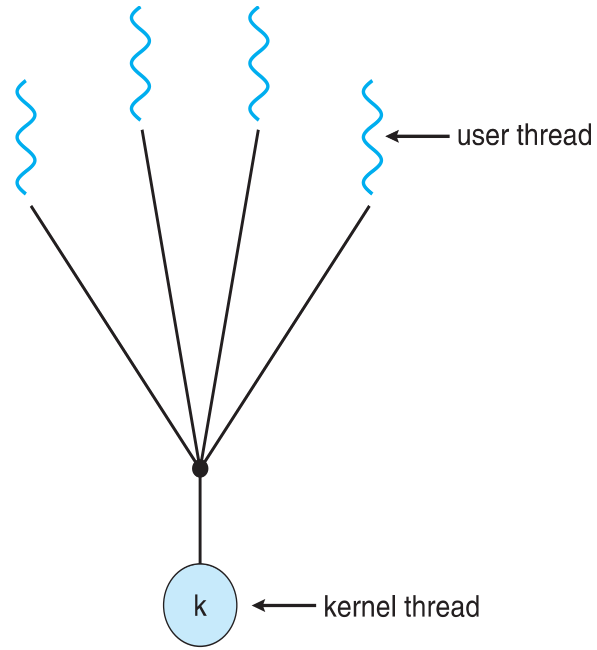
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

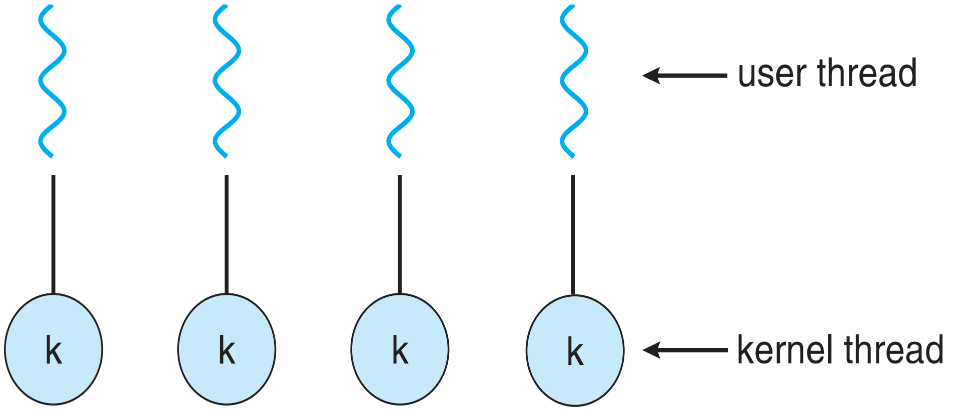
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



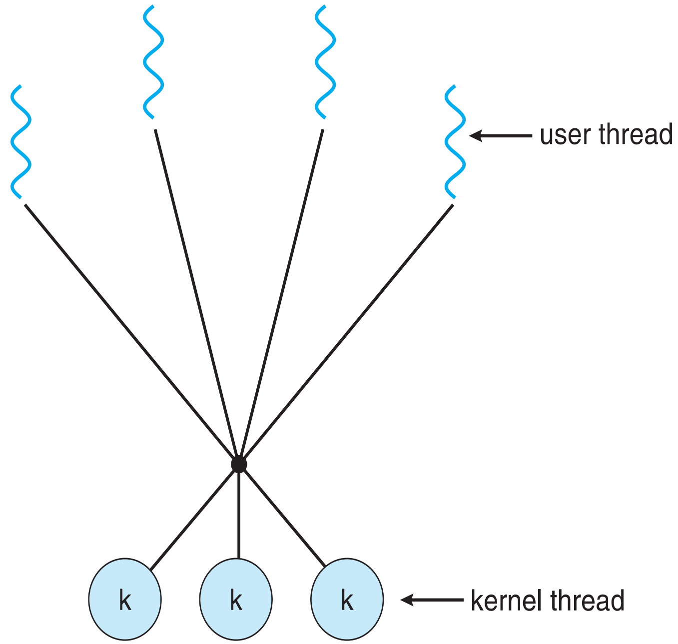
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



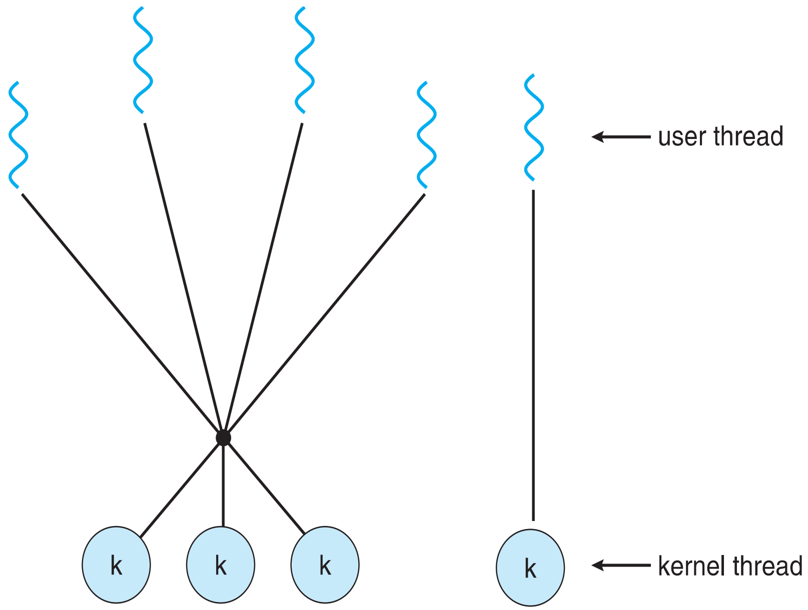
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



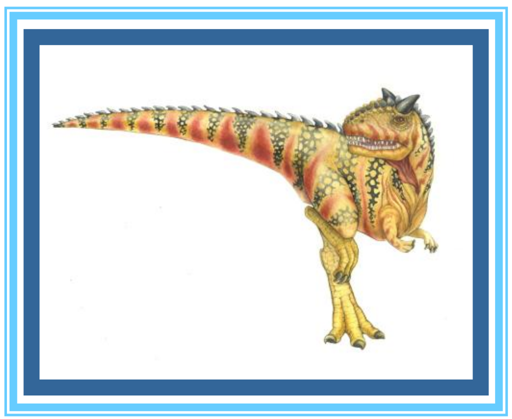
Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

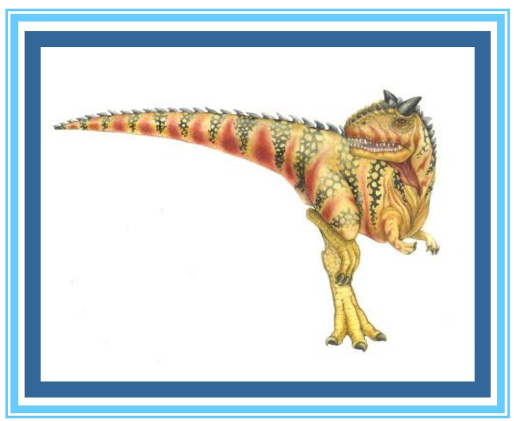
- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

End of Chapter 4



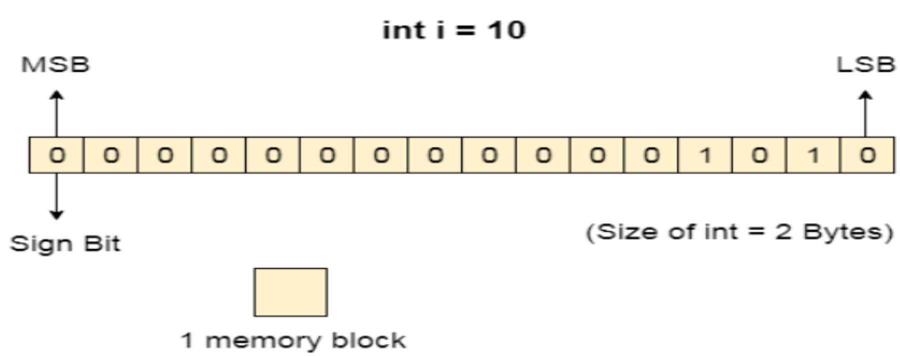
Memory Management
Day5: Sep 2021

Kiran Waghmare



What is Memory?

- Computer memory can be defined as a **collection of some data represented in the binary format.**
- On the basis of various functions, memory can be classified into various categories.
- Machine **understands only binary language that is 0 or 1**.
- Computer converts every data into binary language first and then stores it into the memory.

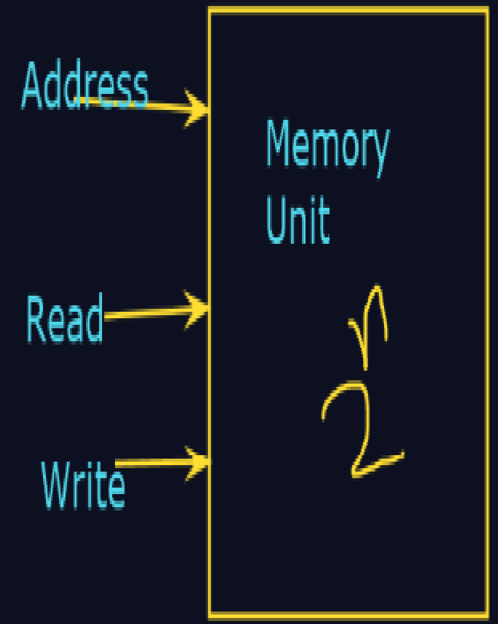


Memory Management:

Memory : collection of some amount of data, which is represented in the binary format.

-Bit : 0 / 1

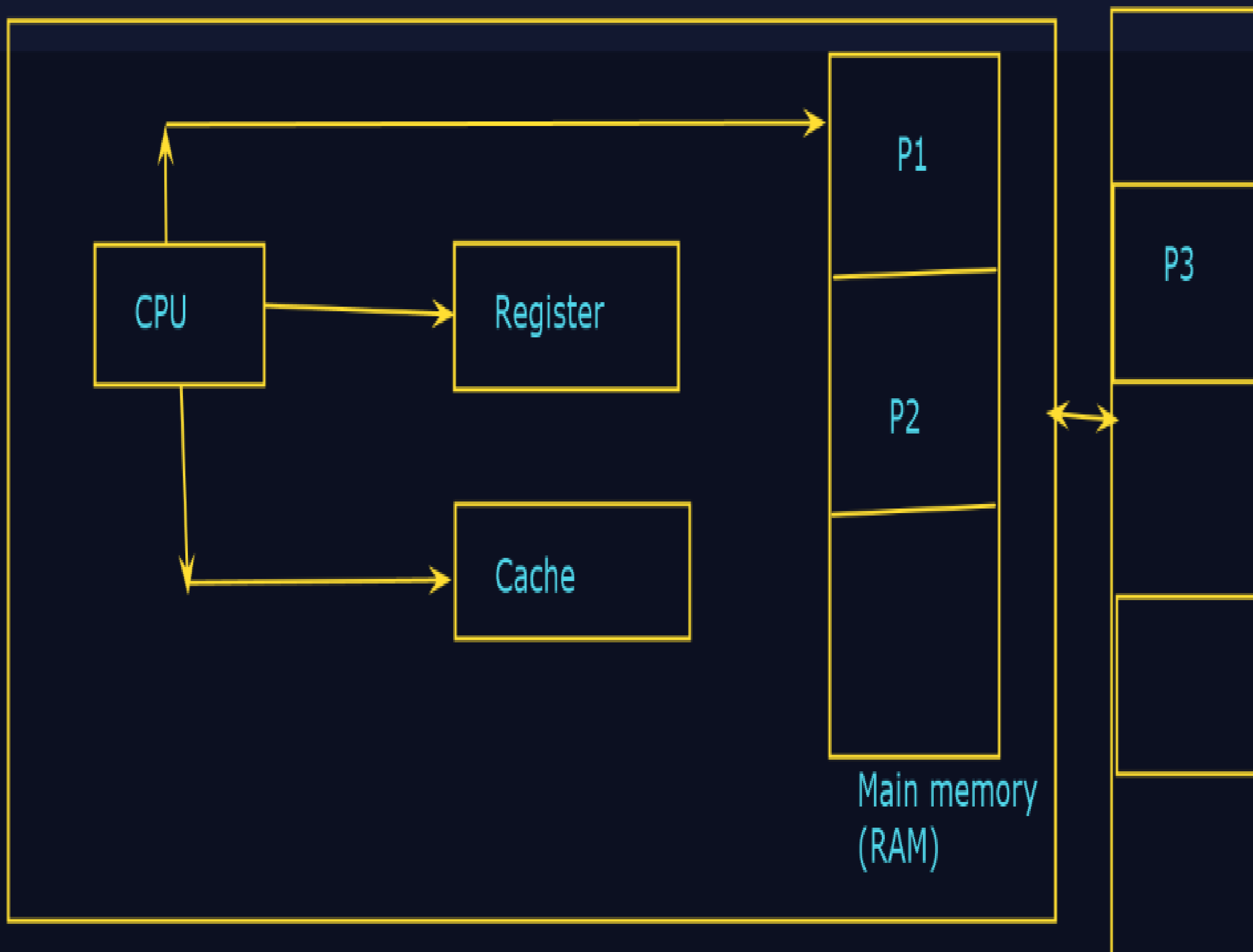
0 10 0 0 1 1 1 1 0 1 0 1 1



Memory Management:

Memory : collection of some amount of data, which is represented in the binary format.

-Bit : 0 / 1

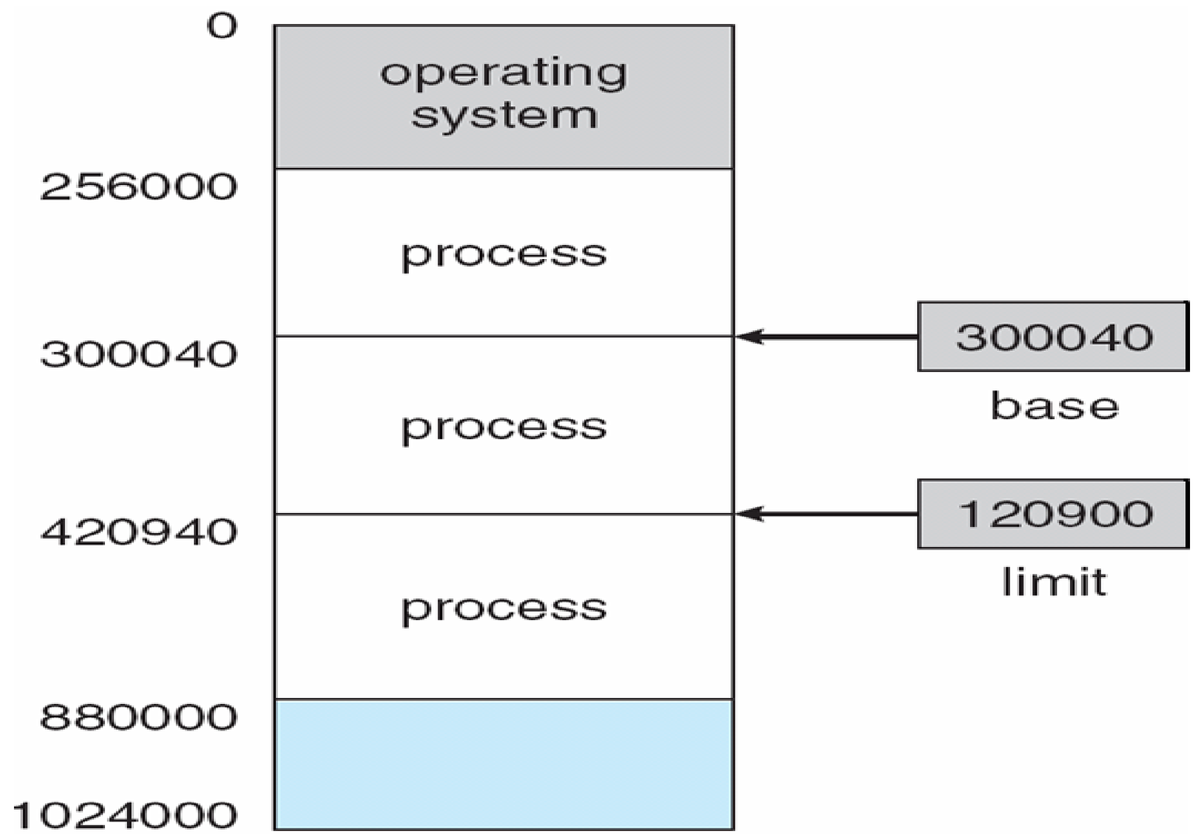


What is Main Memory:

- The main memory is **central to the operation of a modern computer**.
- Main Memory is a **large array of words or bytes, ranging in size from hundreds of thousands to billions.**
- Main memory is a **repository of rapidly available information** shared by the CPU and I/O devices.
- Main memory is the place where **programs and information are kept** when the processor is effectively utilizing them.
- Main memory is **associated with the processor, so moving instructions and information i**nto and out of the processor is extremely fast.
- Main memory is also known **as RAM(Random Access Memory).**
- This memory is a **volatile memory**.
- RAM lost its data when a power interruption occurs.

Base and Limit Registers

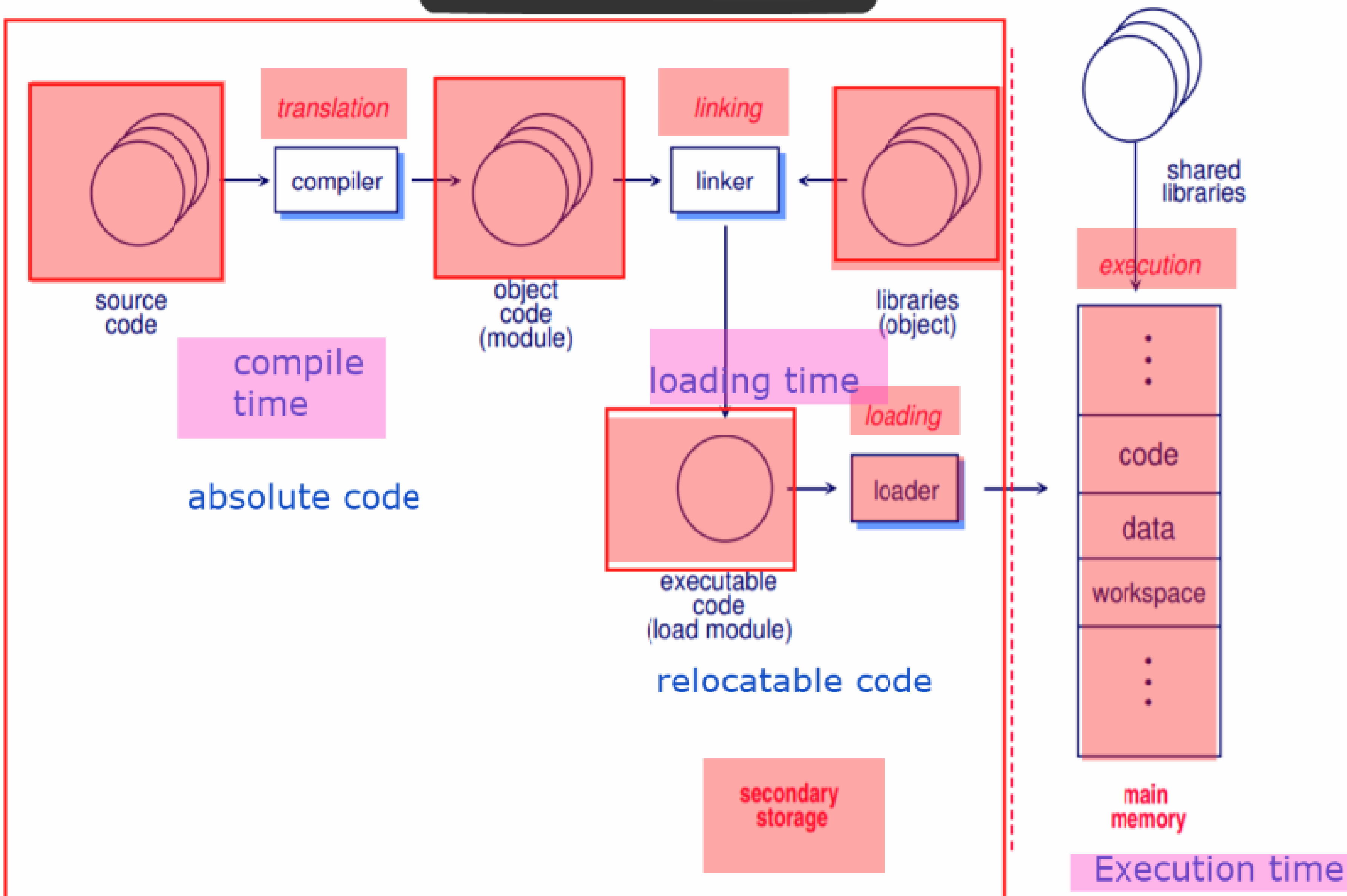
- A pair of **base** and **limit** registers define the logical address space



Why Memory Management is required:

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

From *source* to *executable* code



Binding of Instructions and Data to Memory

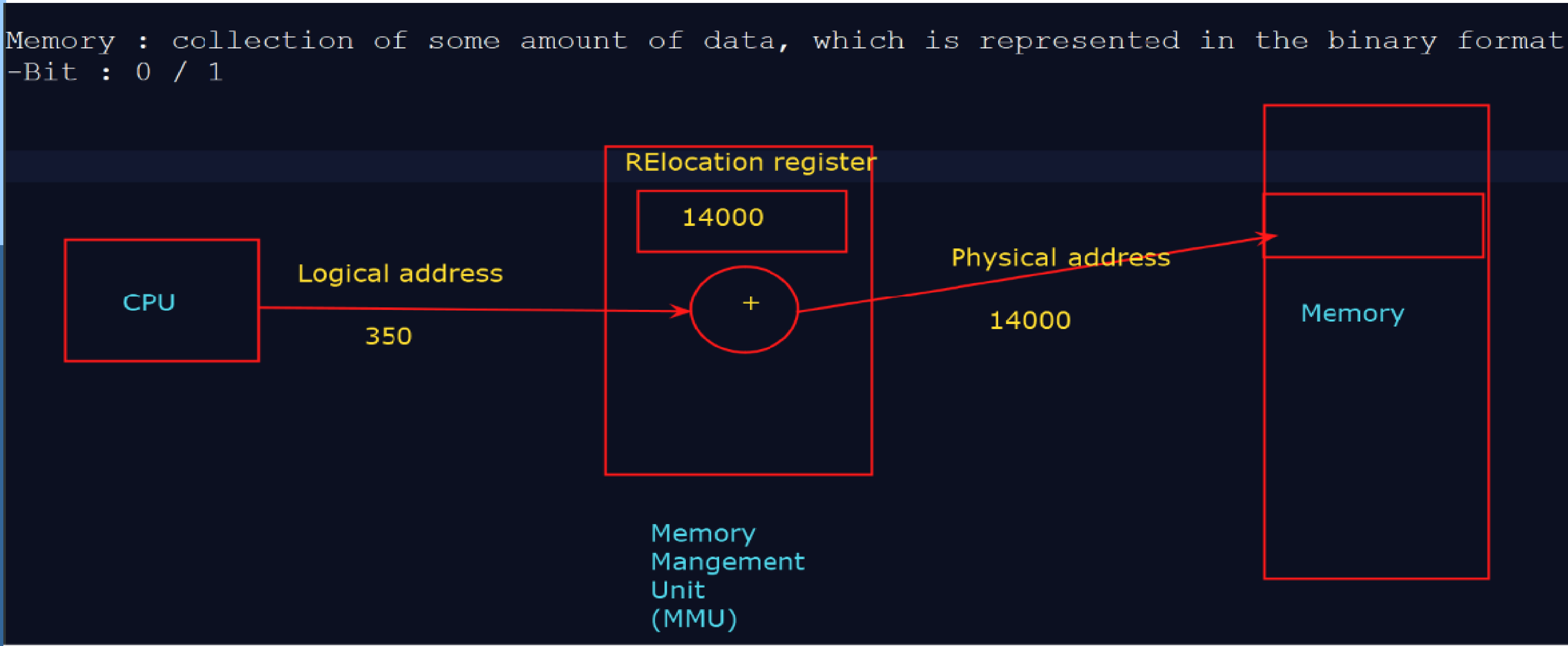
- Address binding of instructions and data to memory addresses can happen at three different stages
  - **Compile time**: If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
  - **Load time**: Must generate **relocatable code** if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Logical and Physical Address Space:

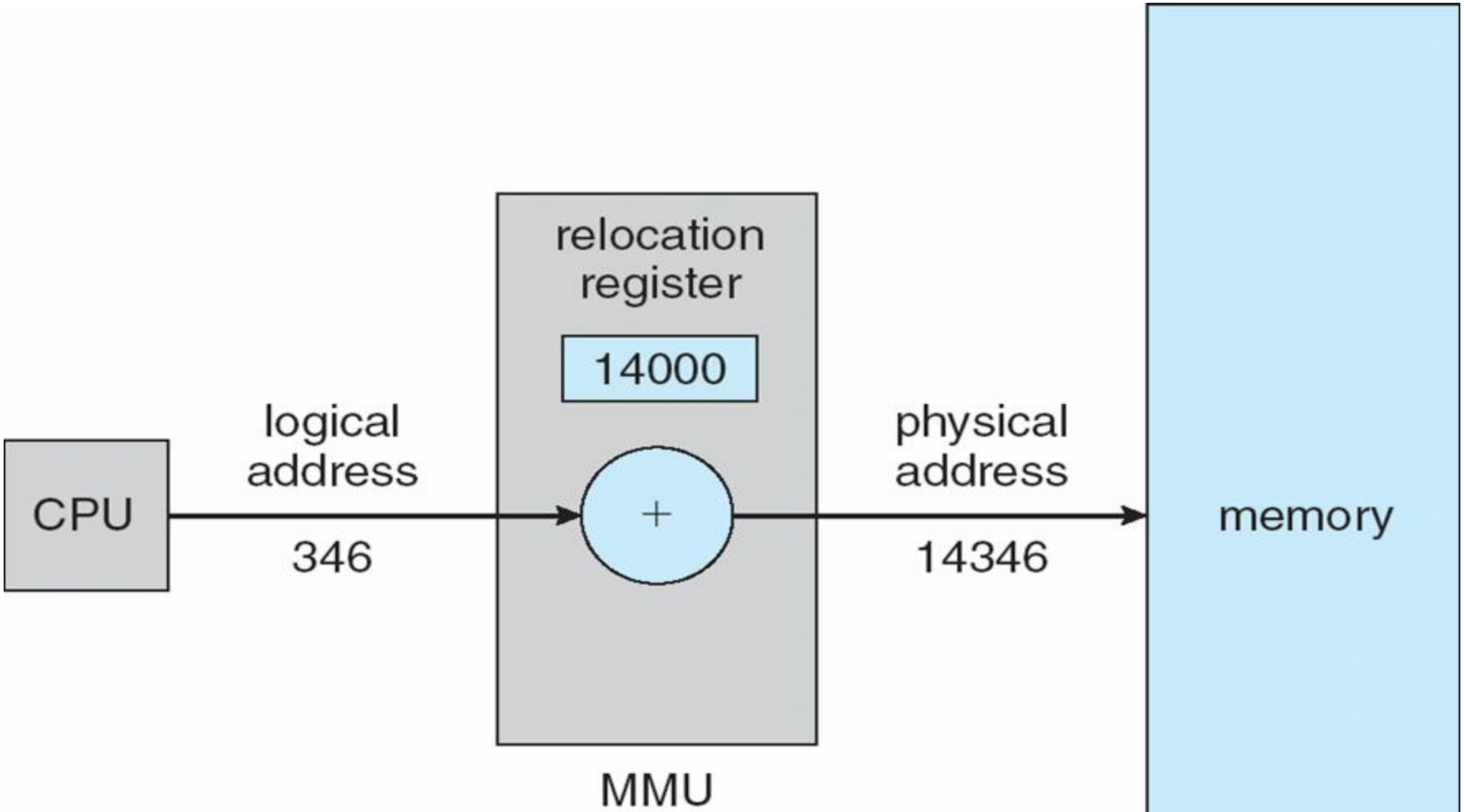
- **Logical Address space:** An address generated by the CPU is known as "Logical Address".
- It is also known as a **Virtual address**.
- Logical address space can be **defined as the size of the process**.
- A logical address can be changed.
- **Physical Address space:** An address seen by the memory unit (i.e the one loaded into the memory address register of the memory) is commonly known as a "Physical Address".
- A Physical address is also known as a **Real address.**
- The set of all physical addresses corresponding to these logical addresses is known as **Physical address space**.
- A physical address is **computed by MMU.**
- The run-time mapping from virtual to physical addresses is done by a **hardware device Memory Management Unit(MMU).**
- The physical address **always remains constant**.

Memory-Management Unit (MMU)

- Hardware device that **maps virtual to physical address**
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with ***logical*** **addresses;** it never sees the *real* physical addresses



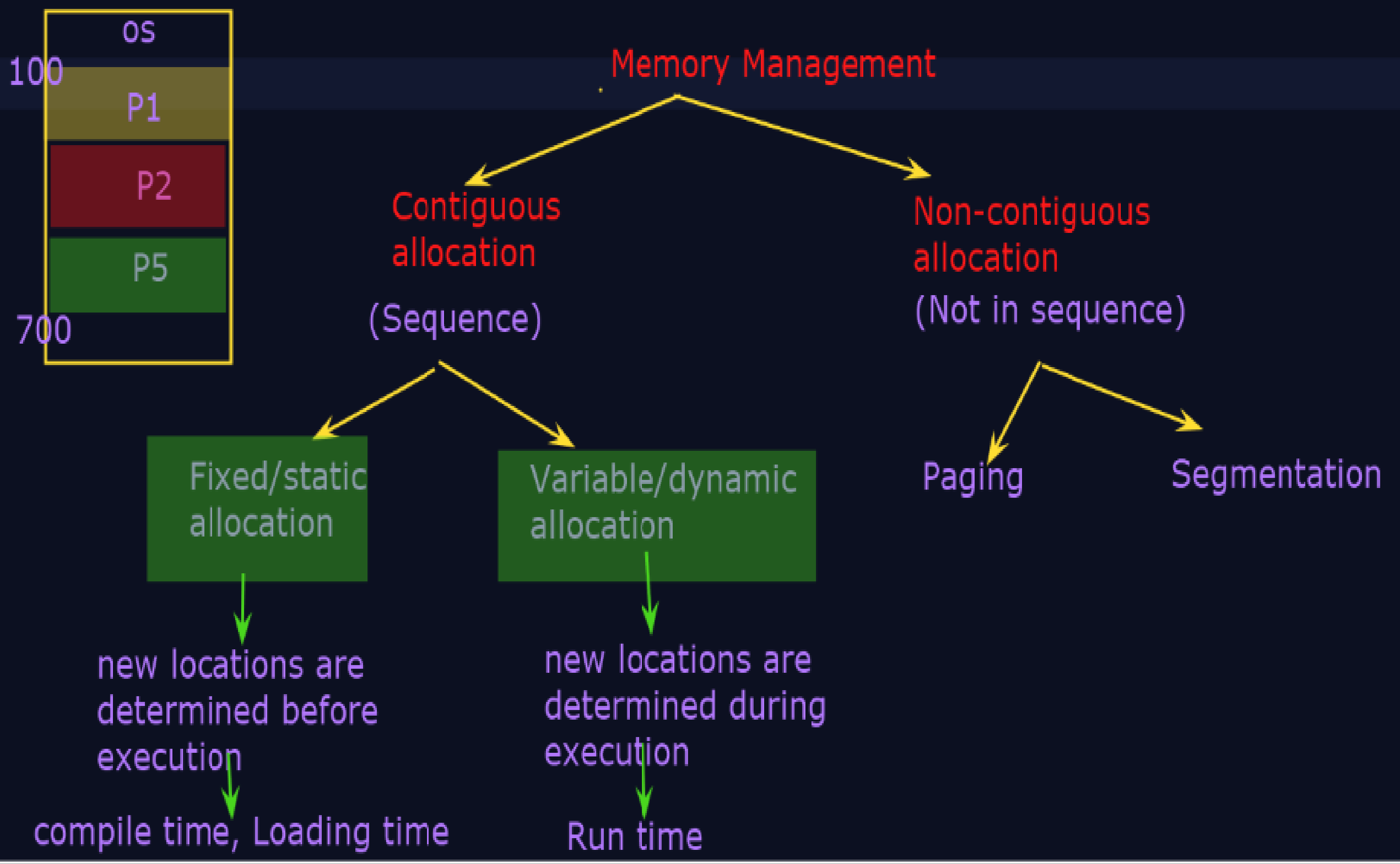
Dynamic relocation using a relocation register



Memory Management:

Memory : collection of some amount of data, which is represented in the

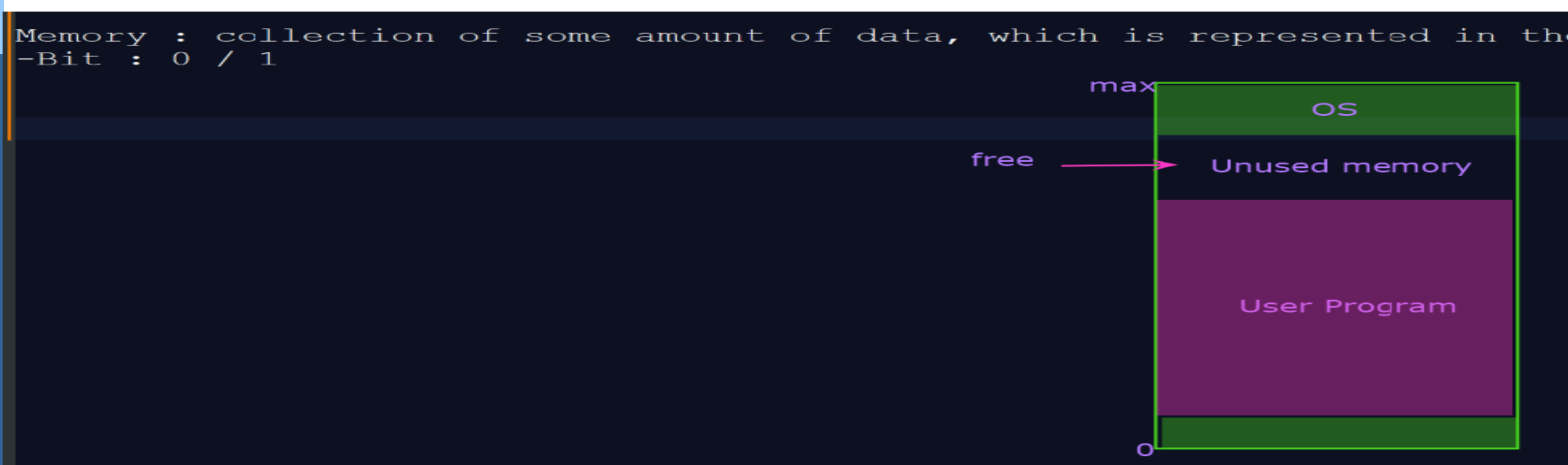
-Bit : 0 / 1



Contiguous Allocation

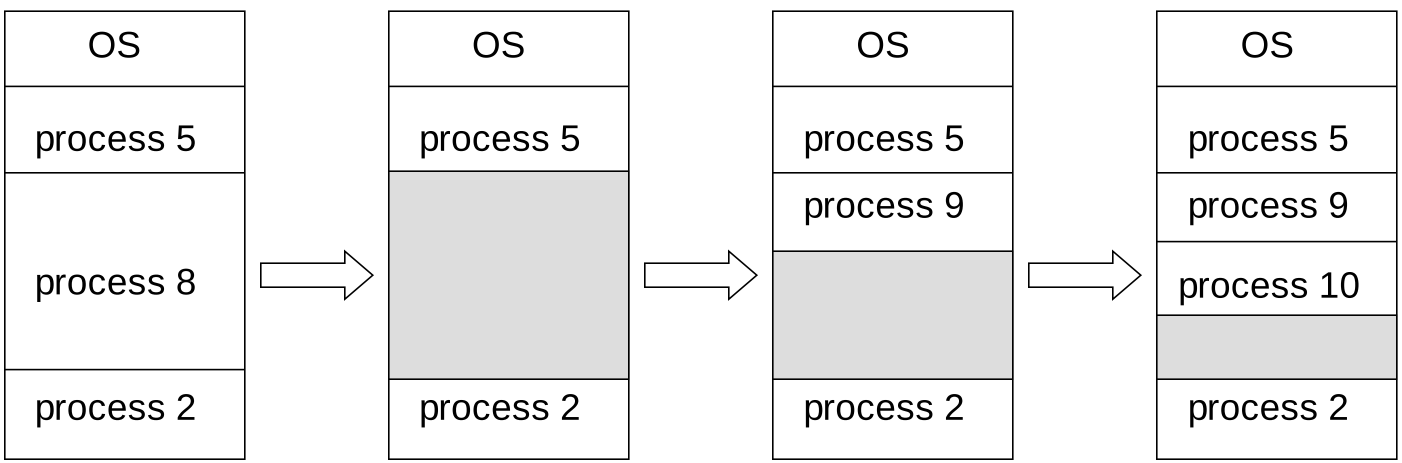
- Main memory usually into two partitions:
  - Resident operating system, usually held in low memory with interrupt vector
  - User processes then held in high memory

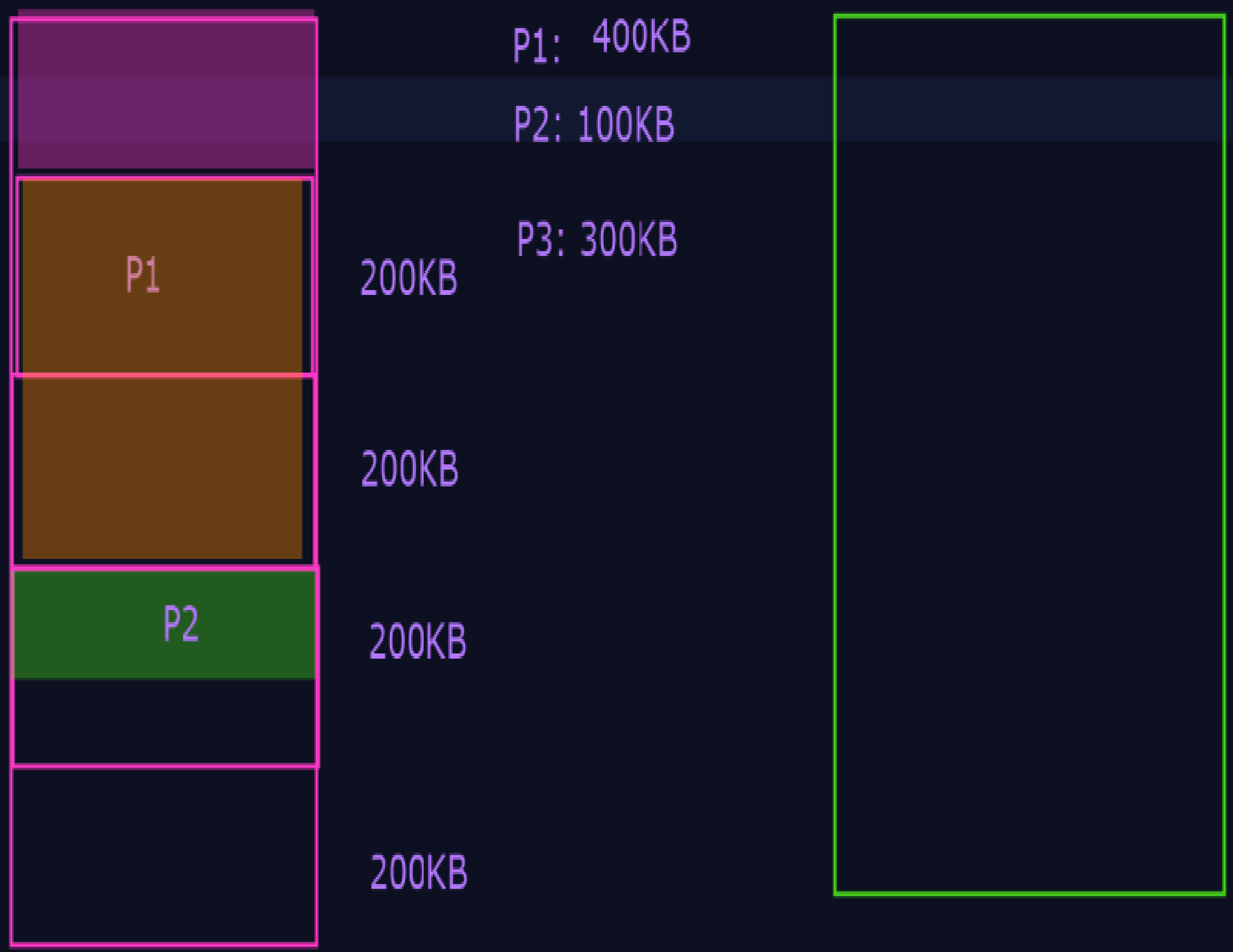
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - Base register contains value of smallest physical address
  - Limit register contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address *dynamically*



Contiguous Allocation (Cont)

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions   b) free partitions (hole)

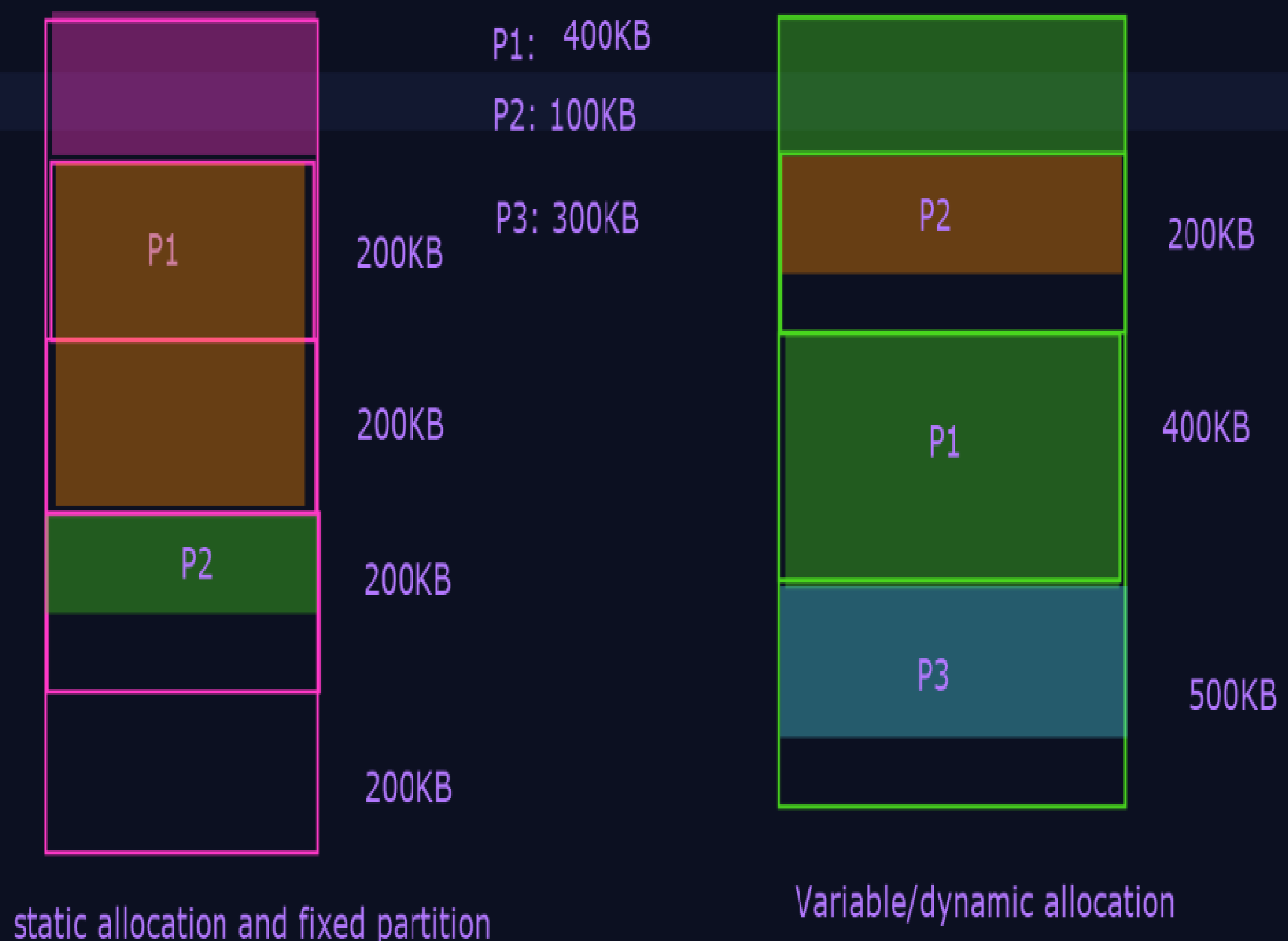




static allocation and fixed partition

Memory Management:

Memory : collection of some amount of data, which is represented in the binary format.
-Bit : 0 / 1



Dynamic Storage-Allocation Problem

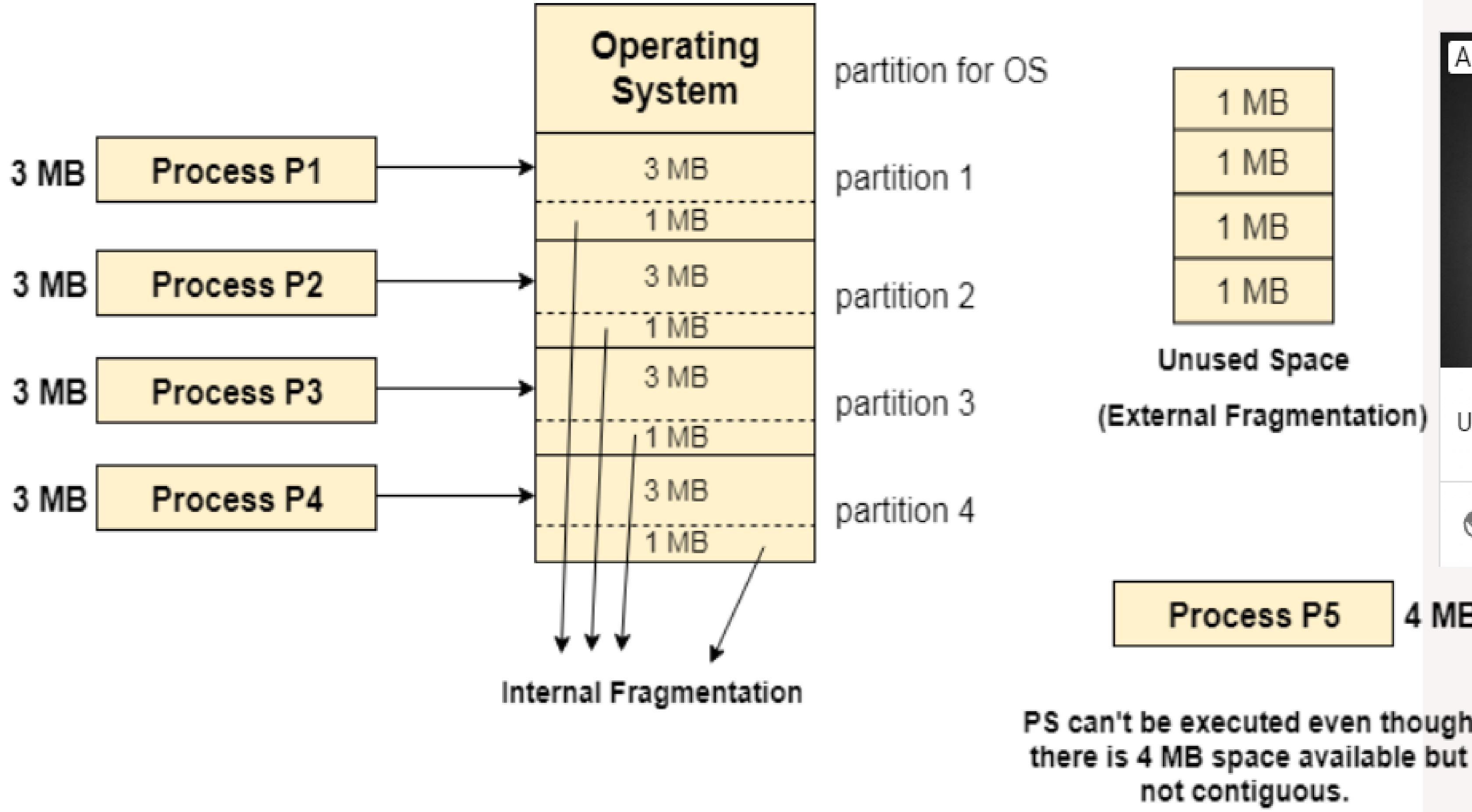
How to satisfy a request of size *n* from a list of free holes

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

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

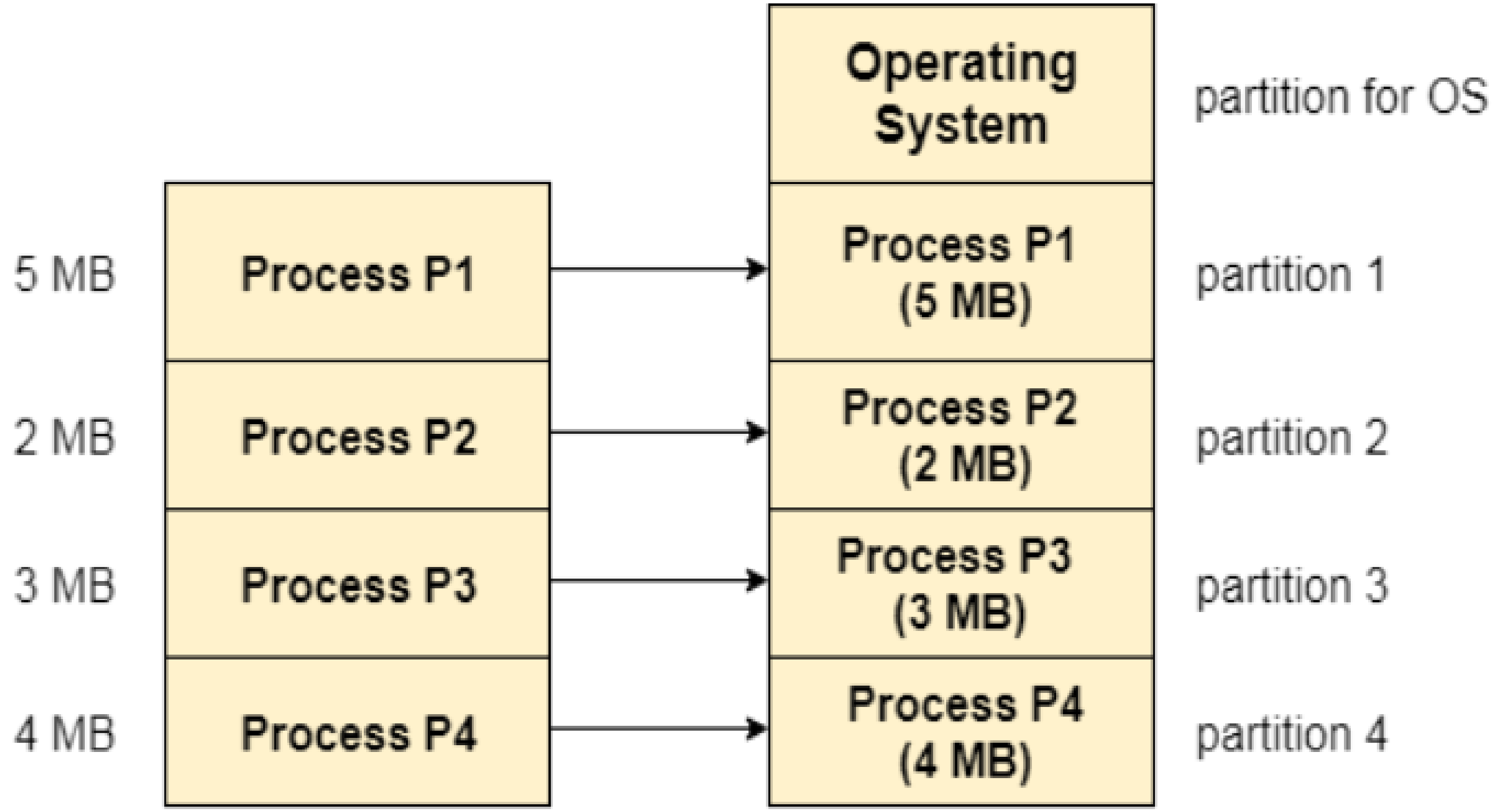
Contiguous Memory Allocation :

- The main memory should **oblige** both the operating system and the different client processes.
- Therefore, the **allocation of memory** becomes an important task in the operating system.
- The memory is **usually divided into two partitions**: one for the resident operating system and one for the user processes.
- We **normally need several user processes** to reside in memory simultaneously.
- Therefore, we need to consider how to allocate available memory to the processes



Fixed Partitioning

(Contiguous memory allocation)

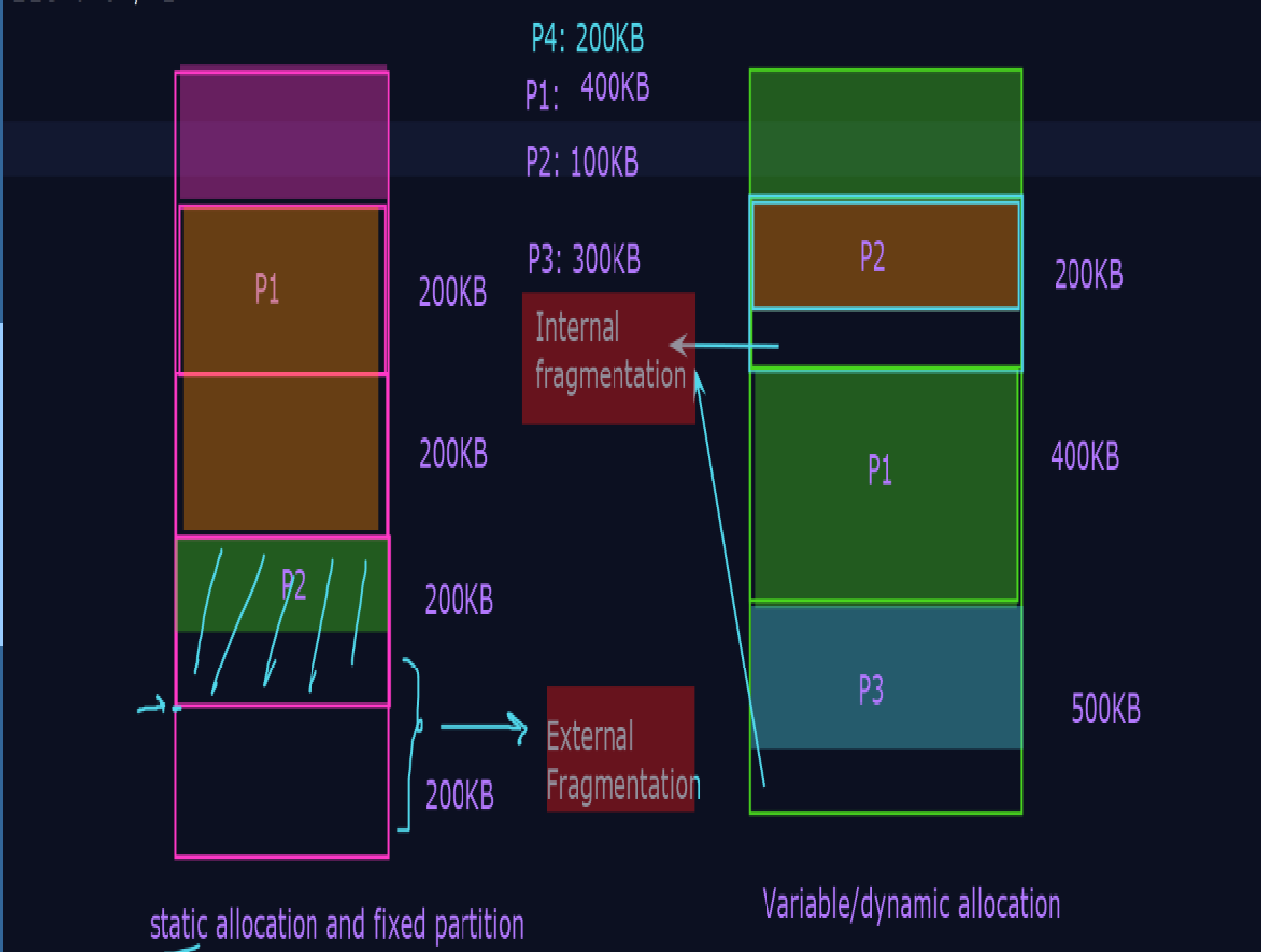


Dynamic Partitioning

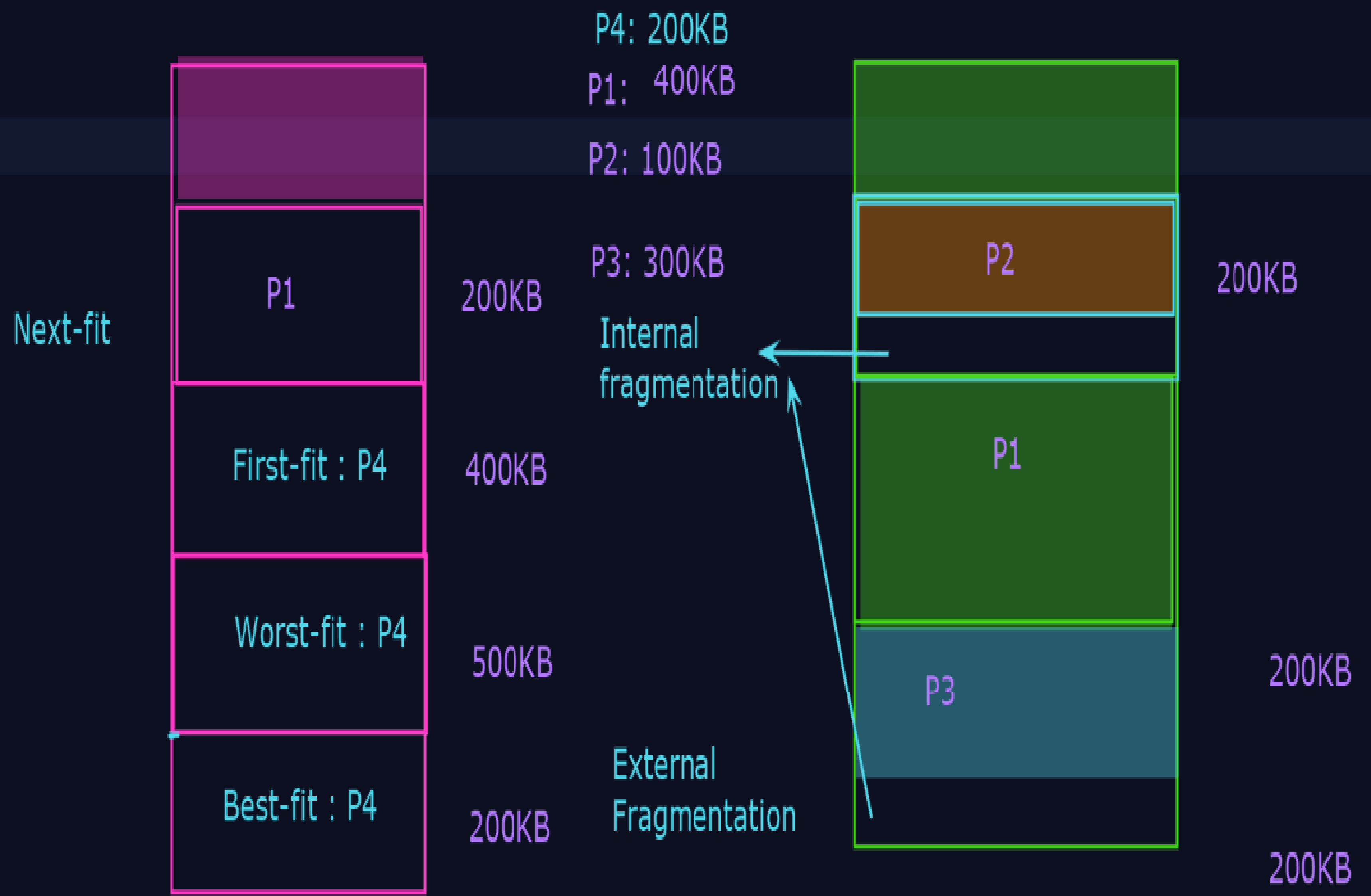
(Process Size = Partition Size)

Fragmentation

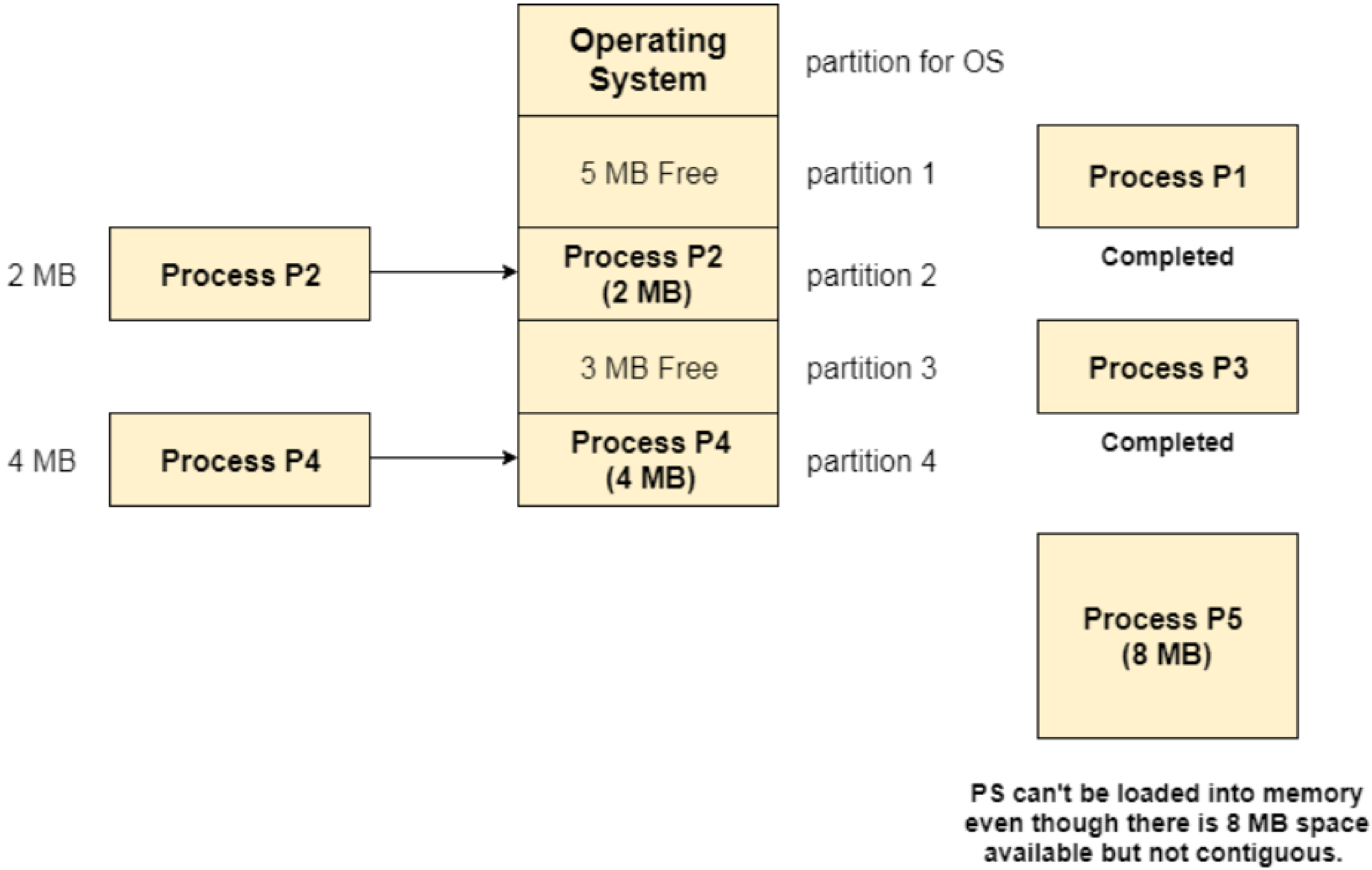
- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is **possible *only* if relocation is dynamic**, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers


P4: 200KB
P1: 400KB
P2: 100KB
P3: 300KB
P1
200KB
200KB
P2
200KB
200KB
Internal fragmentation
External Fragmentation
P2
200KB
P1
400KB
P3
500KB
static allocation and fixed partition
Variable/dynamic allocation

Memory : collection of some amount of data, which is represented in the binary format.
-Bit : 0 / 1



Memory allocation algorithm

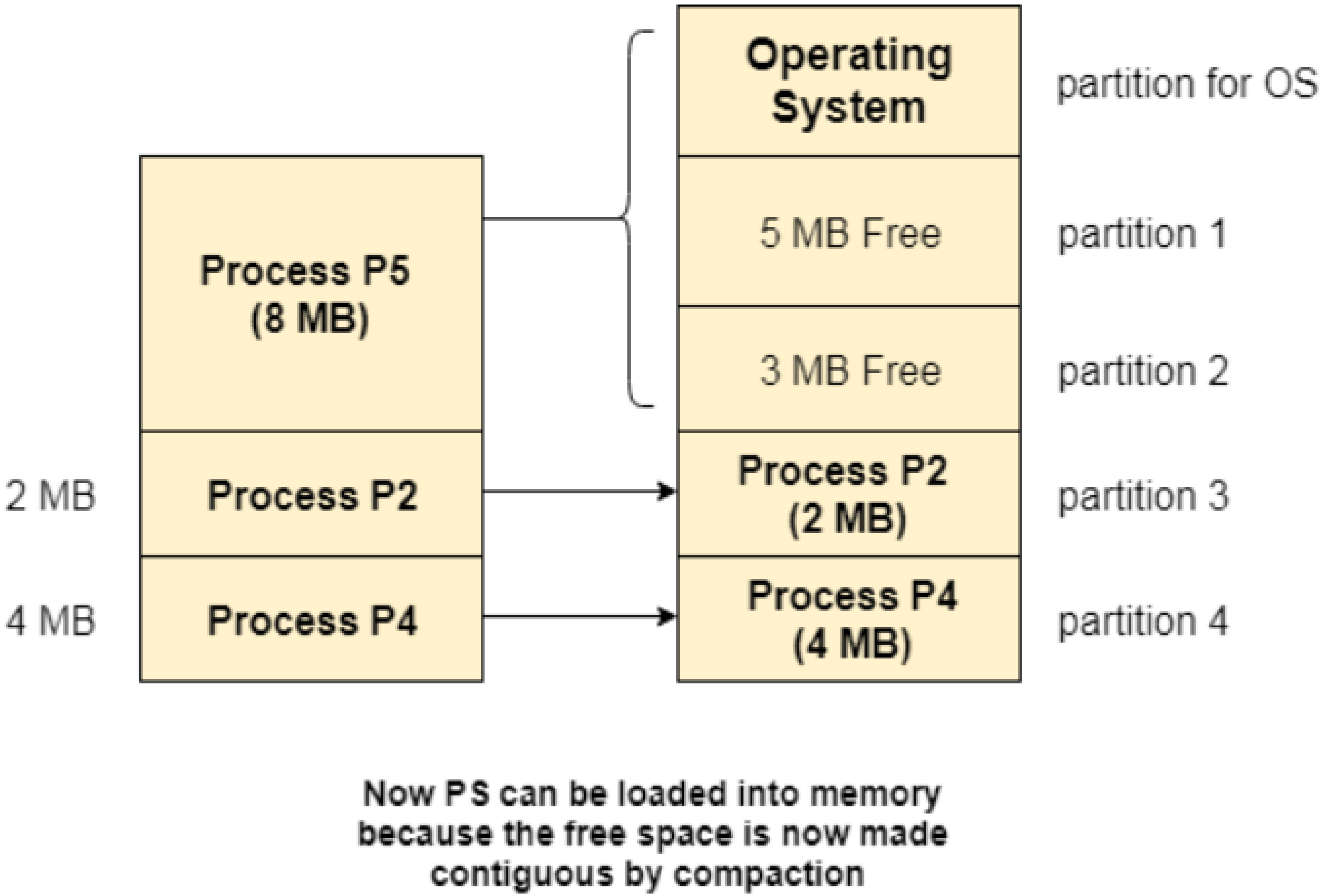


X

External Fragmentation in Dynamic Partitioning

Compaction

- We got to know that the **dynamic partitioning suffers from external fragmentation**. However, this can cause some serious problems.
- To avoid compaction, we need to change the rule which says that the process can't be stored in the different places in the memory.
- We can also **use compaction to minimize the probability of external fragmentation**. In compaction, all the free partitions are made contiguous and all the loaded partitions are brought together.
- By applying this technique, we can store the bigger processes in the memory.
- The free partitions are merged which can now be allocated according to the needs of new processes. This technique is also called defragmentation.


Operating System
partition for OS
5 MB Free
partition 1
Process P5 (8 MB)
3 MB Free
partition 2
2 MB
Process P2
Process P2 (2 MB)
partition 3
4 MB
Process P4
Process P4 (4 MB)
partition 4
Now PS can be loaded into memory because the free space is now made contiguous by compaction

Compaction

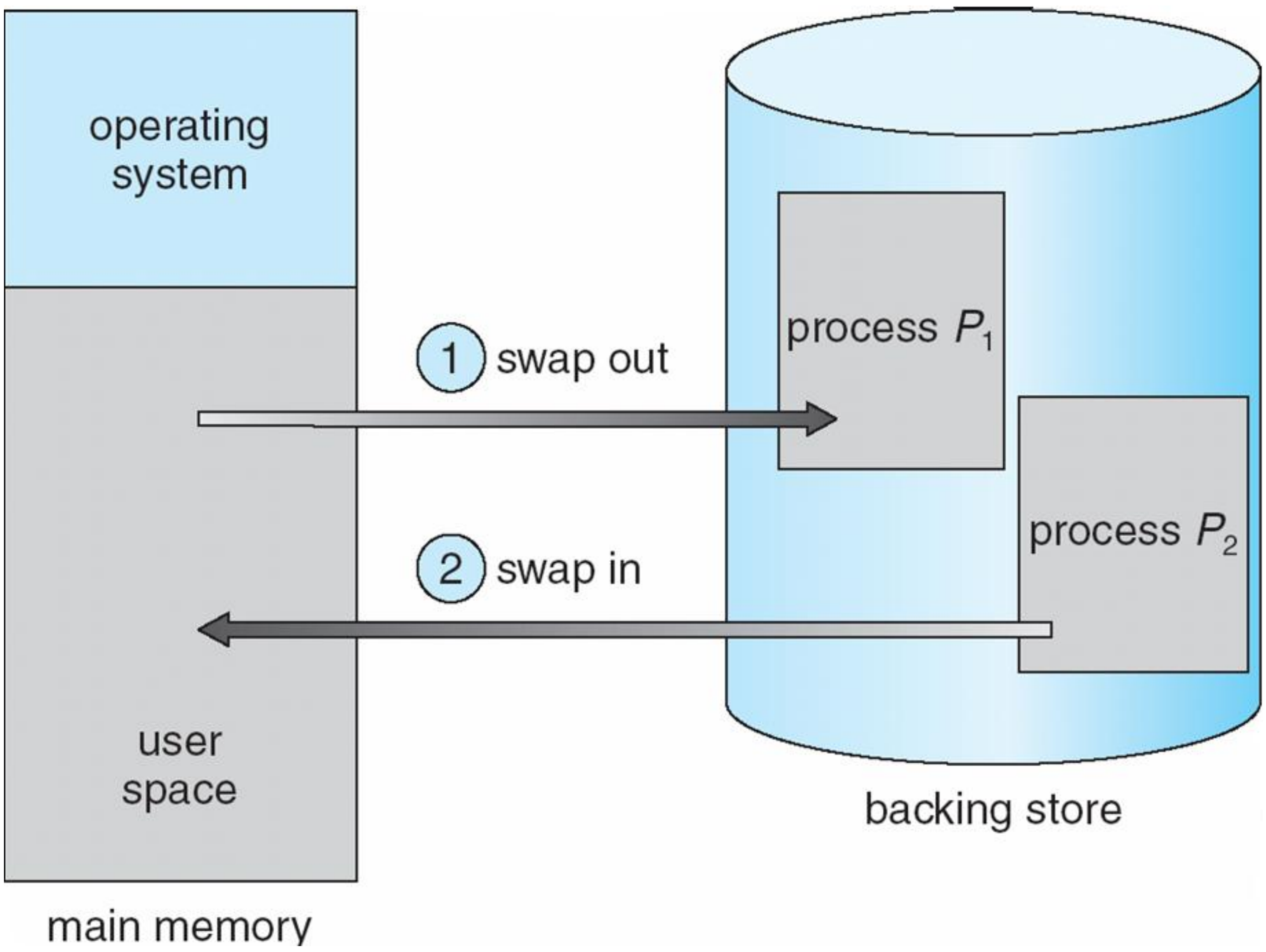
Bit Map for Dynamic Partitioning

- The Main concern for dynamic partitioning is keeping track of all the free and allocated partitions. However, the Operating system uses following data structures for this task.
  - Bit Map
  - Linked List

Swapping

- A process **can be swapped temporarily** out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

Schematic View of Swapping



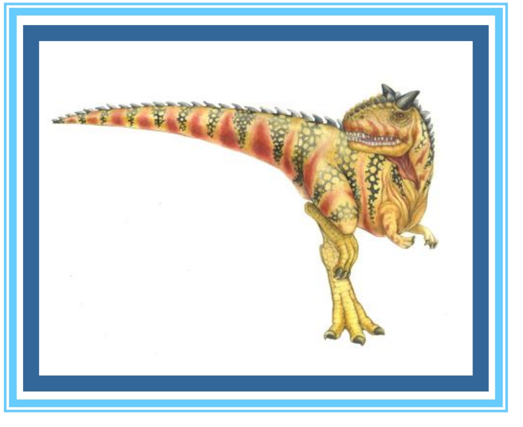
Swap In and Swap Out in OS

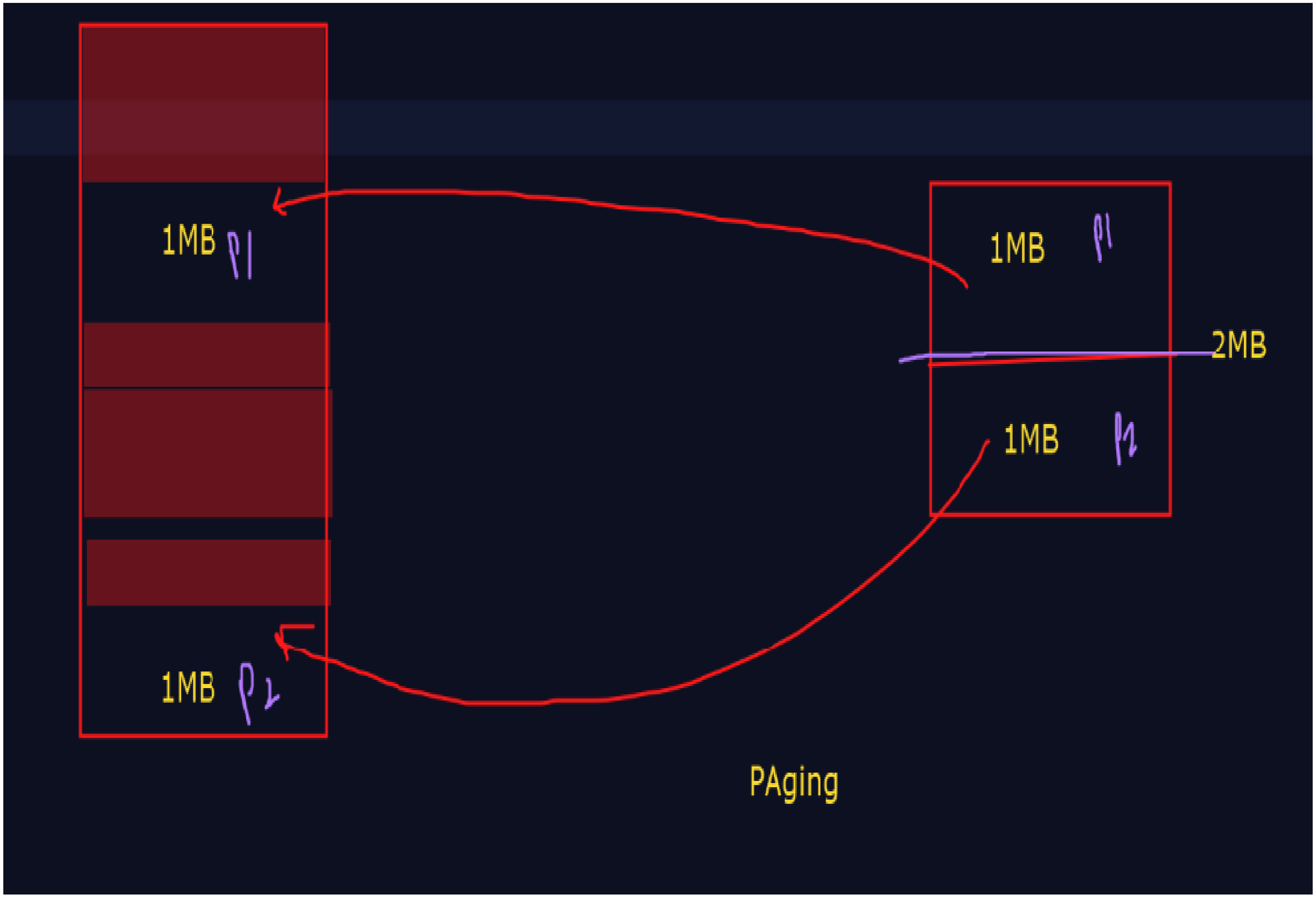
- The procedure by which any process gets removed from the hard disk and placed in the main memory or RAM commonly known as Swap In.
- On the other hand, Swap Out is the method of removing a process from the main memory or RAM and then adding it to the Hard Disk.
- **Advantages of Swapping**
- The swapping technique **mainly helps the CPU to manage multiple processes within a single main memory**.
- This technique **helps to create and use virtual memory**.
- With the help of this technique**, the CPU can perform several tasks** simultaneously. Thus, processes need not wait too long before their execution.
- This technique is **economical.**
- This technique can be easily applied **to priority-based scheduling** in order to improve its performance.

Disadvantages of Swapping

- There may occur inefficiency in the case if a resource or a variable is commonly used by those processes that are participating in the swapping process.
- If the algorithm used for swapping is not good then the overall method can increase the number of page faults and thus decline the overall performance of processing.
- If the computer system loses power at the time of high swapping activity then the user might lose all the information related to the program.

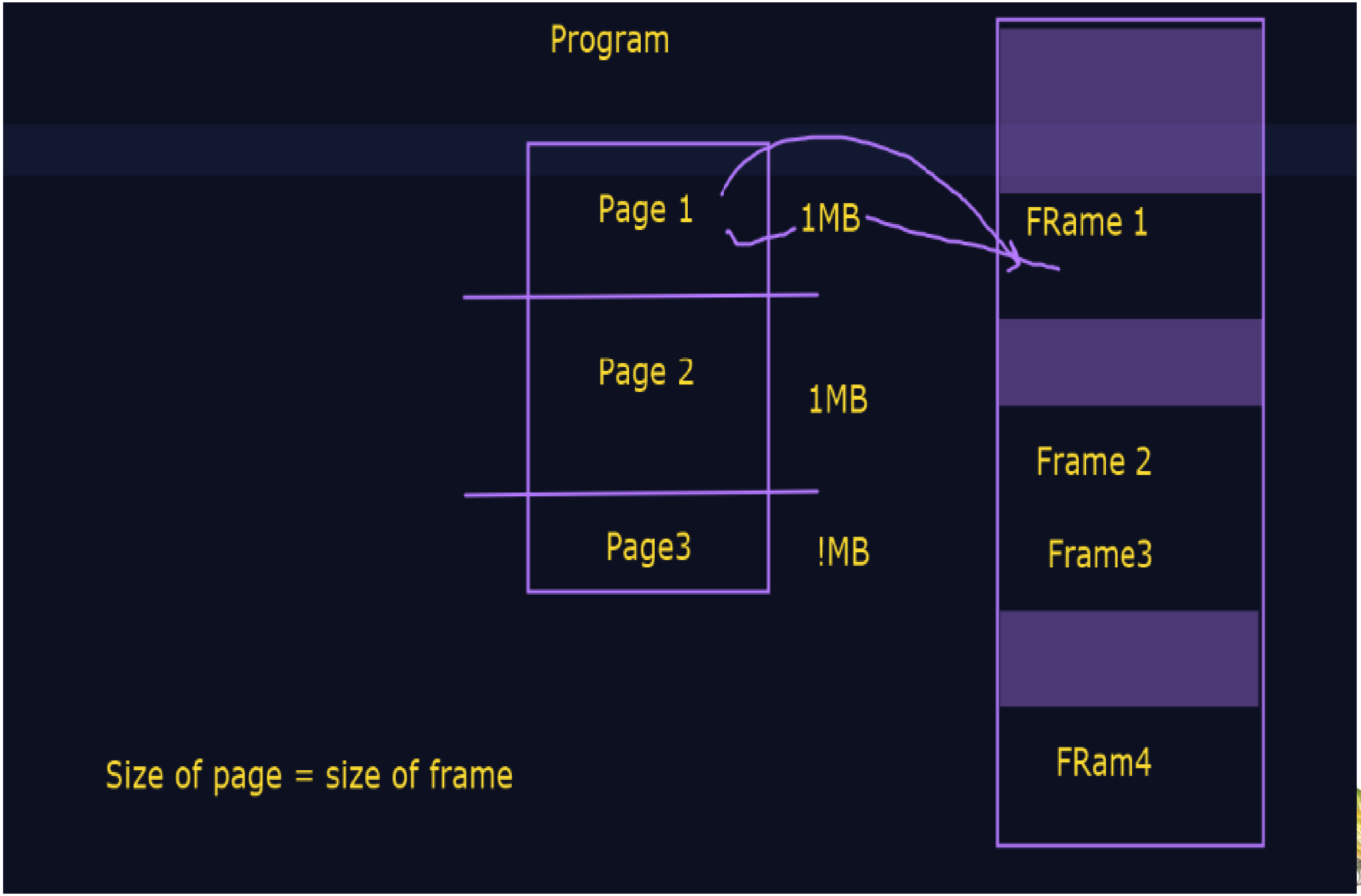
Paging and Segmentation




1MB p1
1MB p2
1MB p1
2MB
1MB p2
PAging

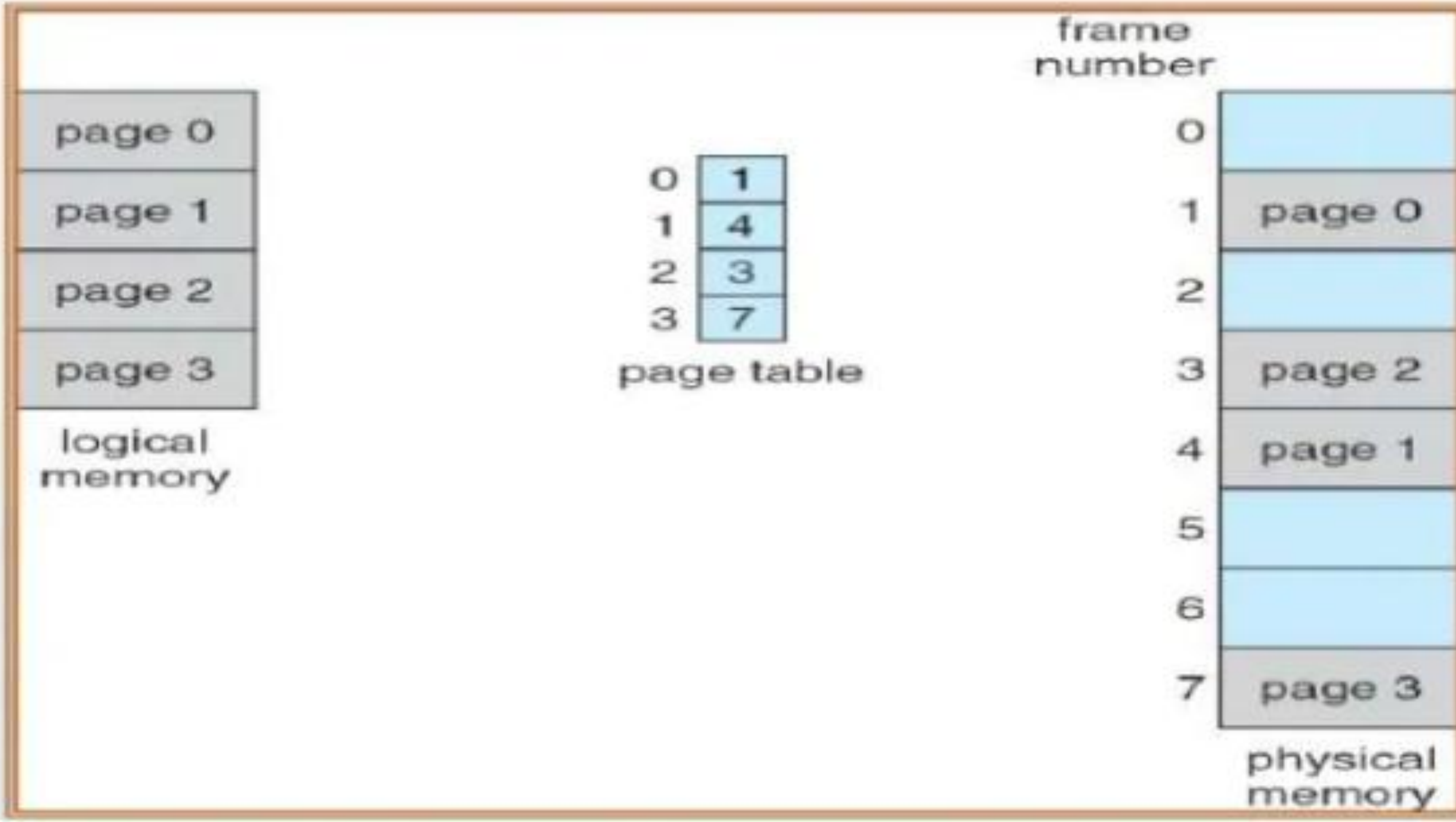
Need for Paging

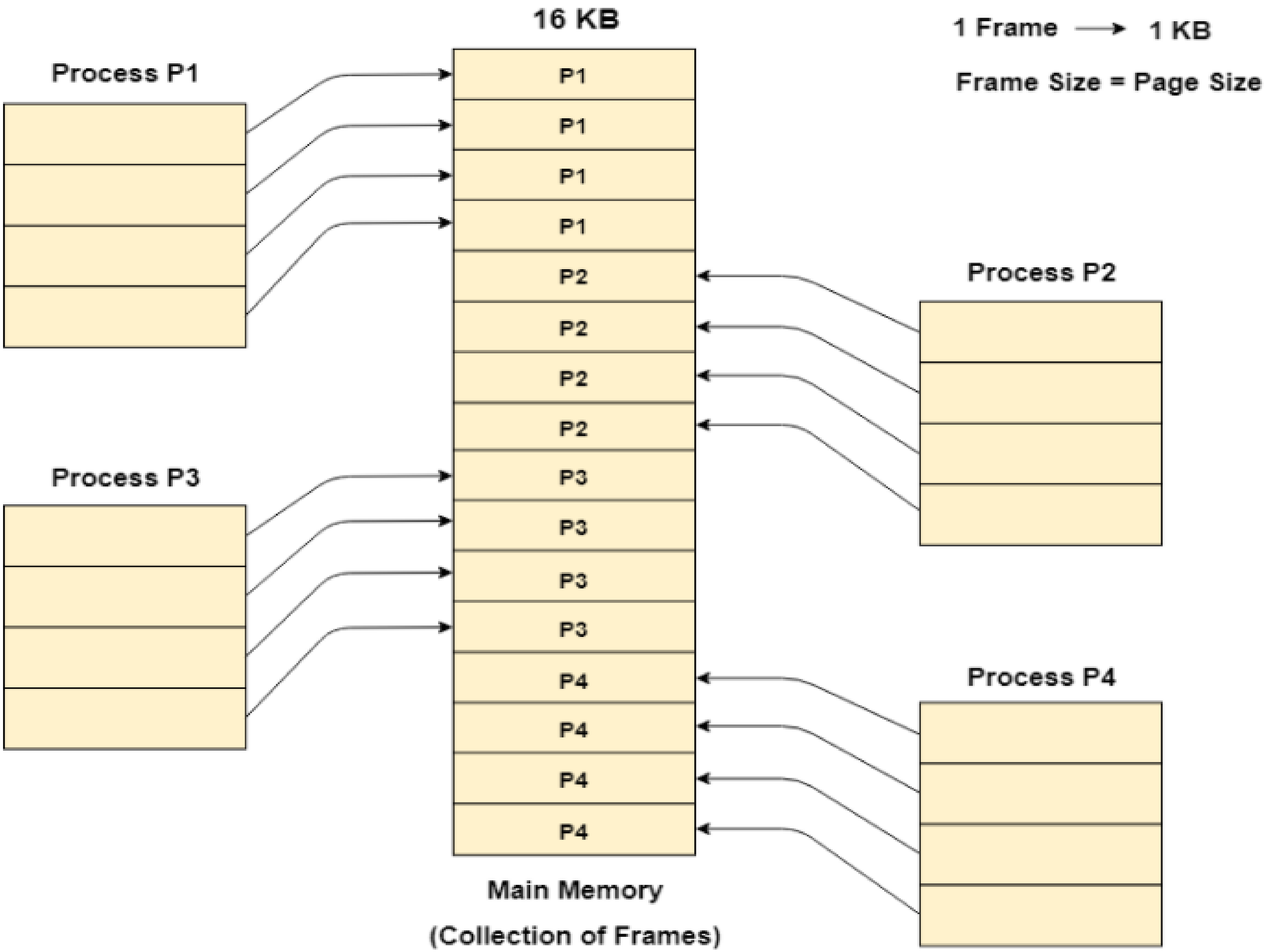
- Disadvantage of Dynamic Partitioning
  - The main disadvantage of Dynamic Partitioning is **External fragmentation.** Although, this can be **removed by Compaction** but as we have discussed earlier, the compaction makes the system inefficient.
  - We **need to find out a mechanism which can load the processes in the partitions** in a more optimal way. Let us discuss a dynamic and flexible mechanism **called paging.**
- **Need for Paging**
- 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.**


Program
Page 1
1MB
FRame 1
Page 2
1MB
Frame 2
Page3
!MB
Frame3
FRam4
Size of page = size of frame

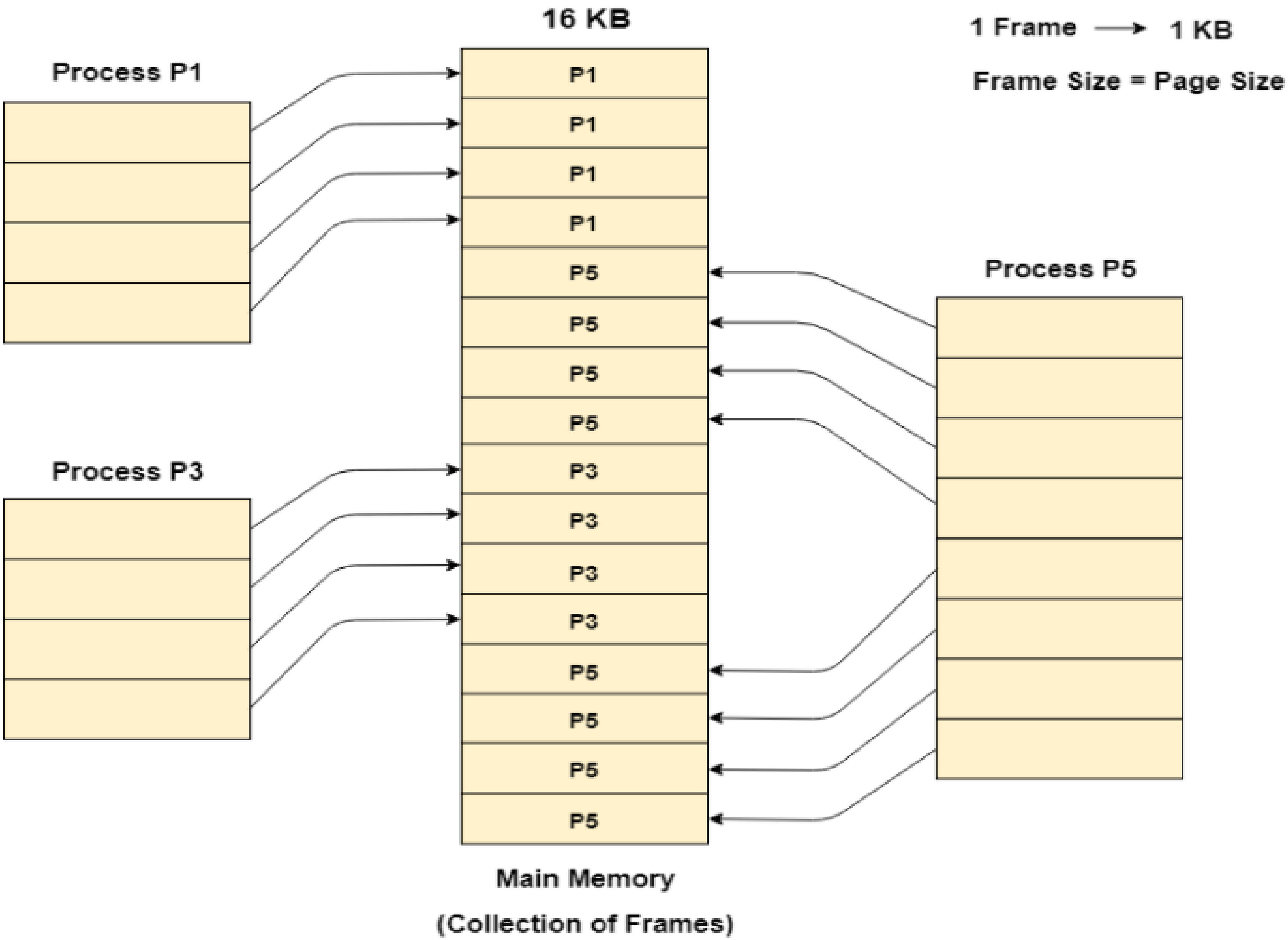
- The main **idea behind the paging is to divide each process** in the **form of pages**. The main memory will also be **divided in the form of frames**.
- **One page of the process is to be stored in one of the frames of the memory**. The pages can be stored at the different locations of the memory 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.

Paging Example




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)

Paging

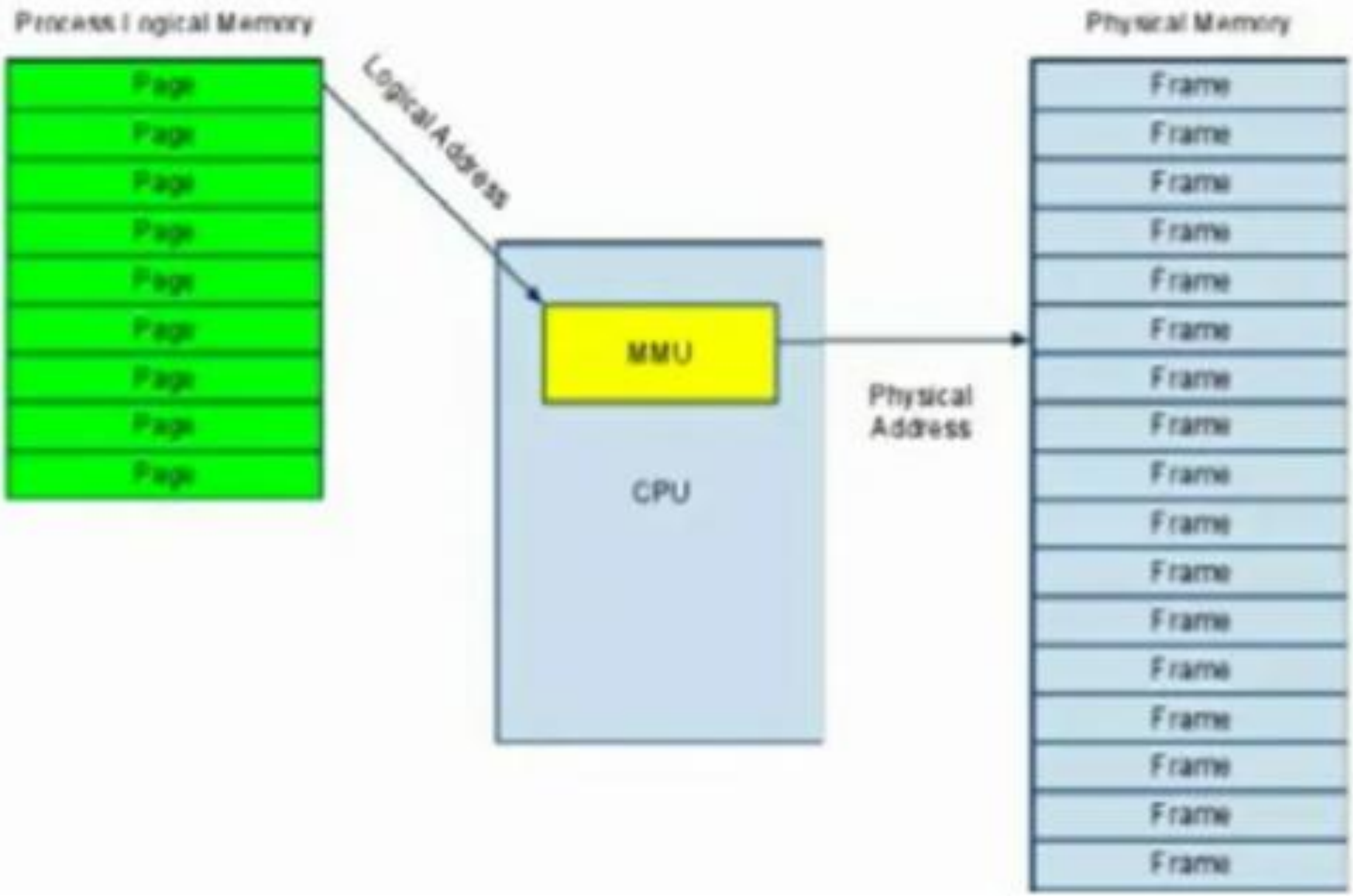

16 KB
1 Frame → 1 KB
Frame Size = Page Size
Process P1
Process P3
Process P5
P1
P1
P1
P1
P5
P5
P5
P5
P3
P3
P3
P3
P5
P5
P5
P5
Main Memory
(Collection of Frames)

Paging

Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)
- Divide logical memory into blocks of same size called **pages**
- Keep track of all free frames
- To run a program of size ***n*** pages, need to find *n* free frames and load program
- Set up a page table to translate logical to physical addresses
- Internal fragmentation

Page Translation

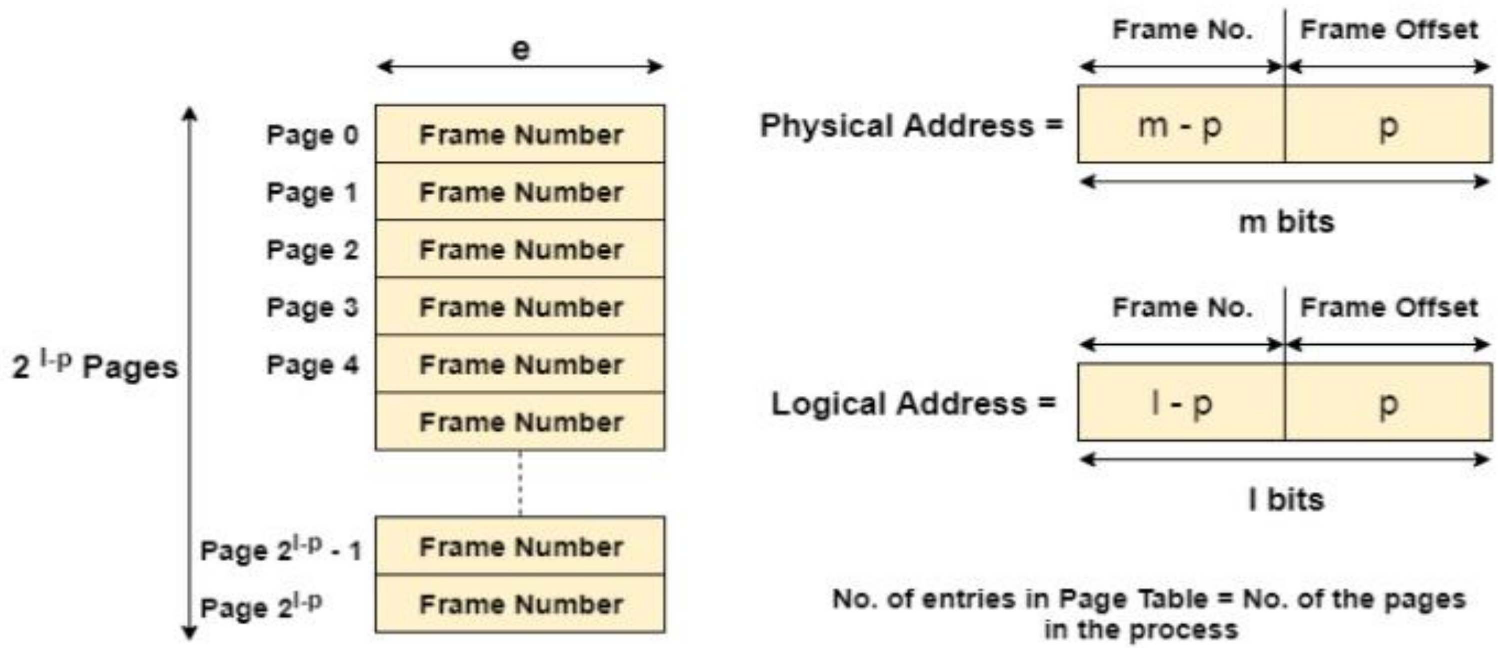


Memory Management Unit

- The purpose of Memory Management Unit (MMU) is **to convert the logical address into the physical address.**
- The logical address is the address generated by the CPU for every page while the physical address is the actual address of the frame where each page will be stored.
- When a page is to be accessed by the CPU by using the logical address, the operating system needs to obtain the physical address to access that page physically.
- The logical address has two parts.
  - Page Number
  - Offset
- Memory management unit of OS needs to convert the page number to the frame number.

Page Table

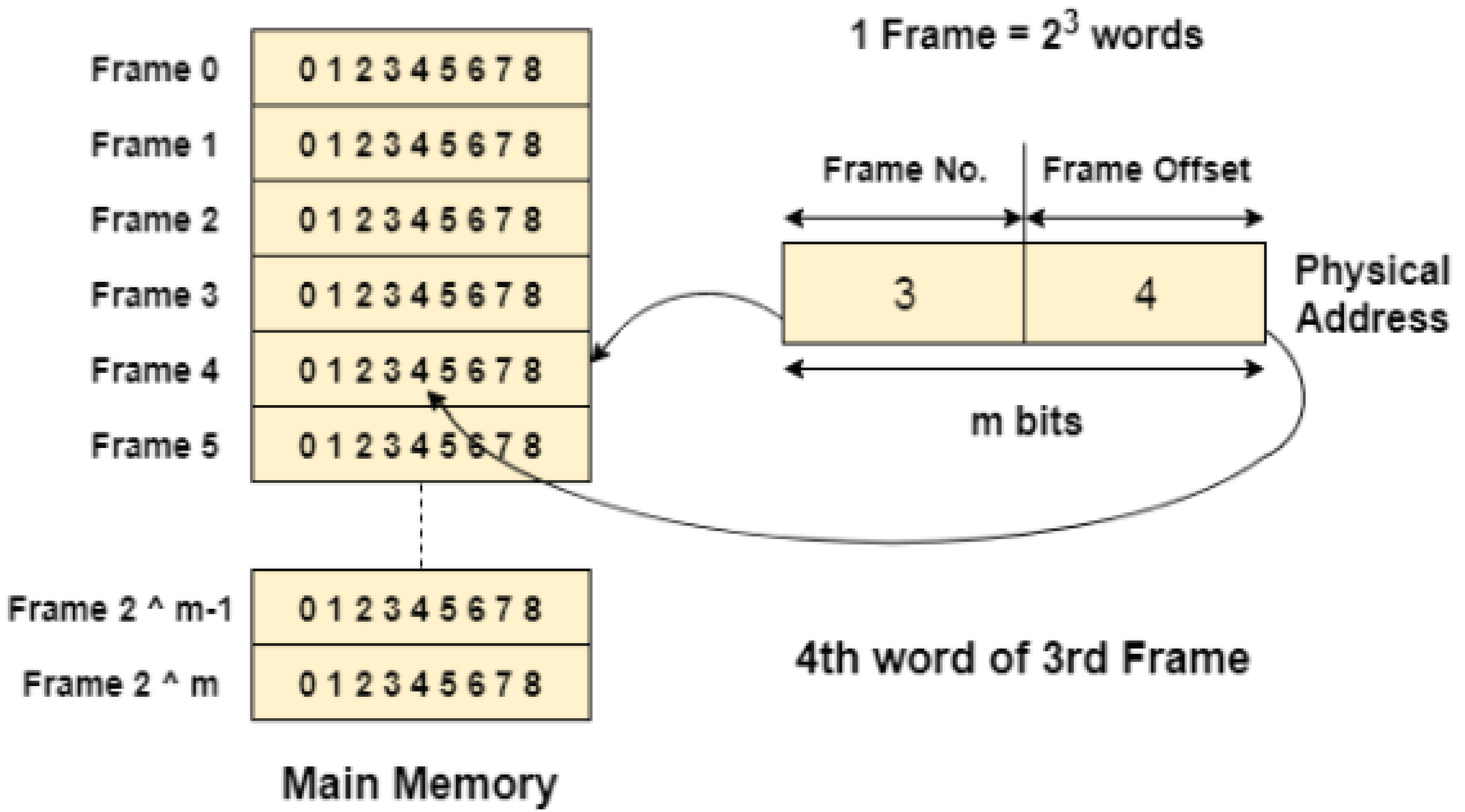
- Page Table is a data structure used by the virtual memory system to store the mapping between logical addresses and physical addresses.

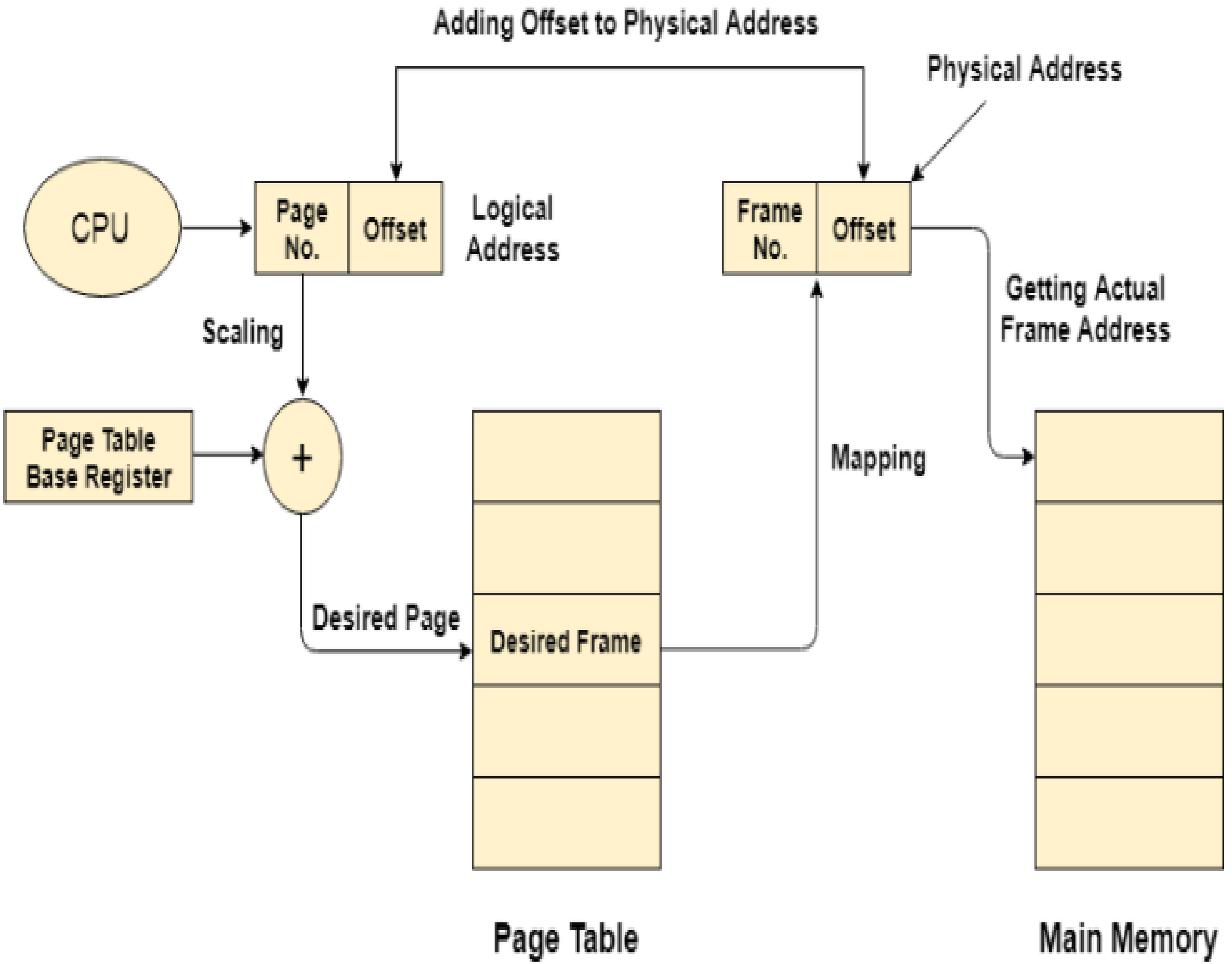


Page Table

Page Table Size = 2^{l-p} X e bytes

e = m-p (Frame Size) bits


1 Frame = 2^3 words
Frame 0
Frame 1
Frame 2
Frame 3
Frame 4
Frame 5
Frame 2 ^ m-1
Frame 2 ^ m
0 1 2 3 4 5 6 7 8
Frame No.
Frame Offset
3
4
Physical Address
m bits
Main Memory
4th word of 3rd Frame


Adding Offset to Physical Address
Physical Address
CPU
Page No.
Offset
Logical Address
Frame No.
Offset
Getting Actual Frame Address
Scaling
Page Table Base Register
+
Mapping
Desired Page
Desired Frame
Page Table
Main Memory

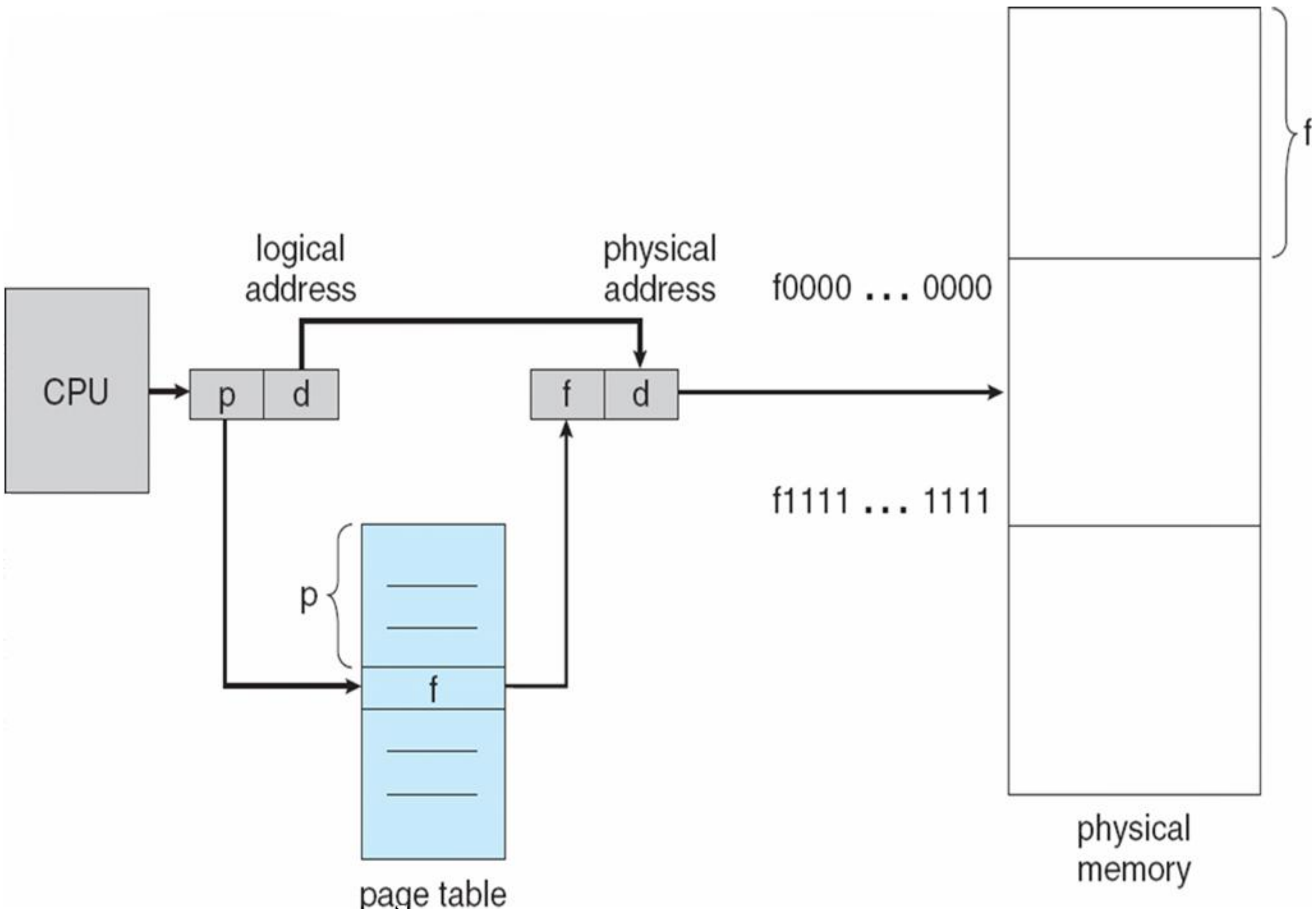
Address Translation Scheme

- Address generated by CPU is divided into:
  - **Page number (*p*)** – used as an index into a *page table* which contains base address of each page in physical memory
  - **Page offset (d)** – combined with base address to define the physical memory address that is sent to the memory unit

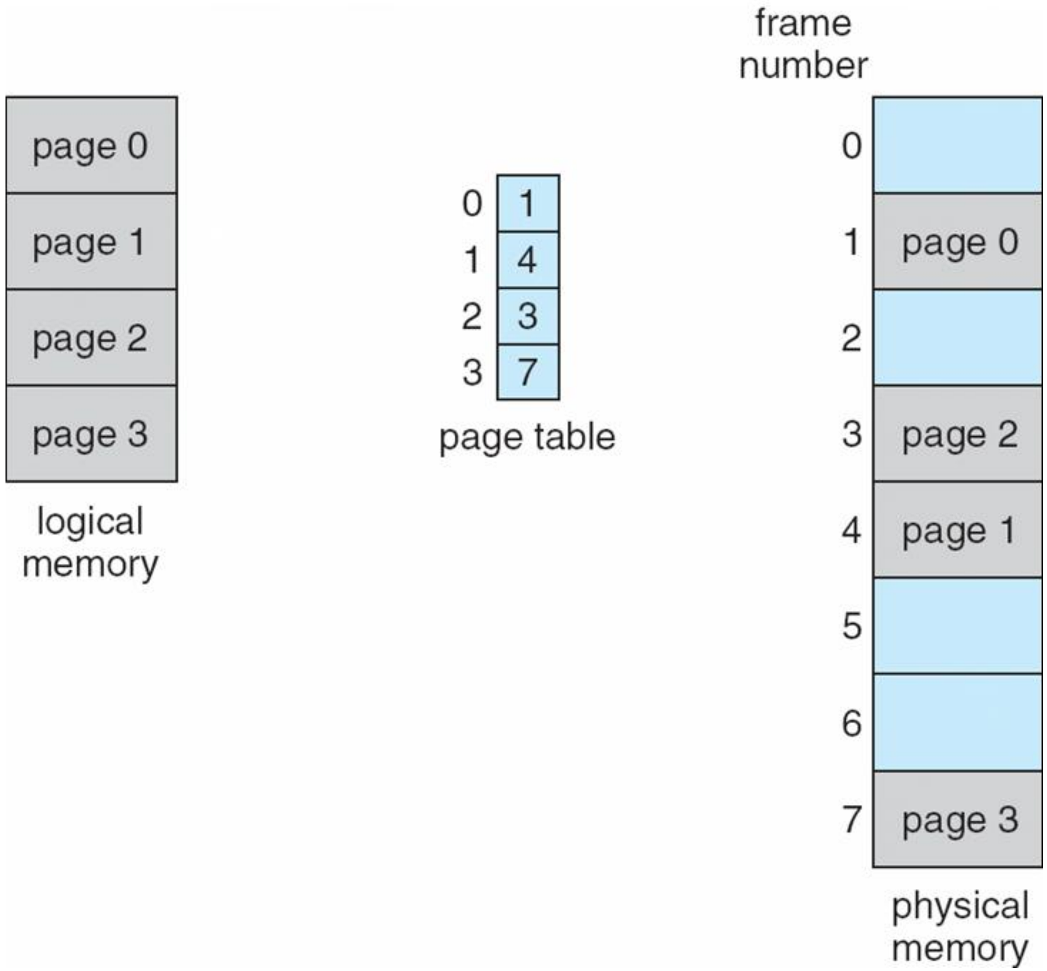
page number	page offset
p	*d*
m - n	*n*

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

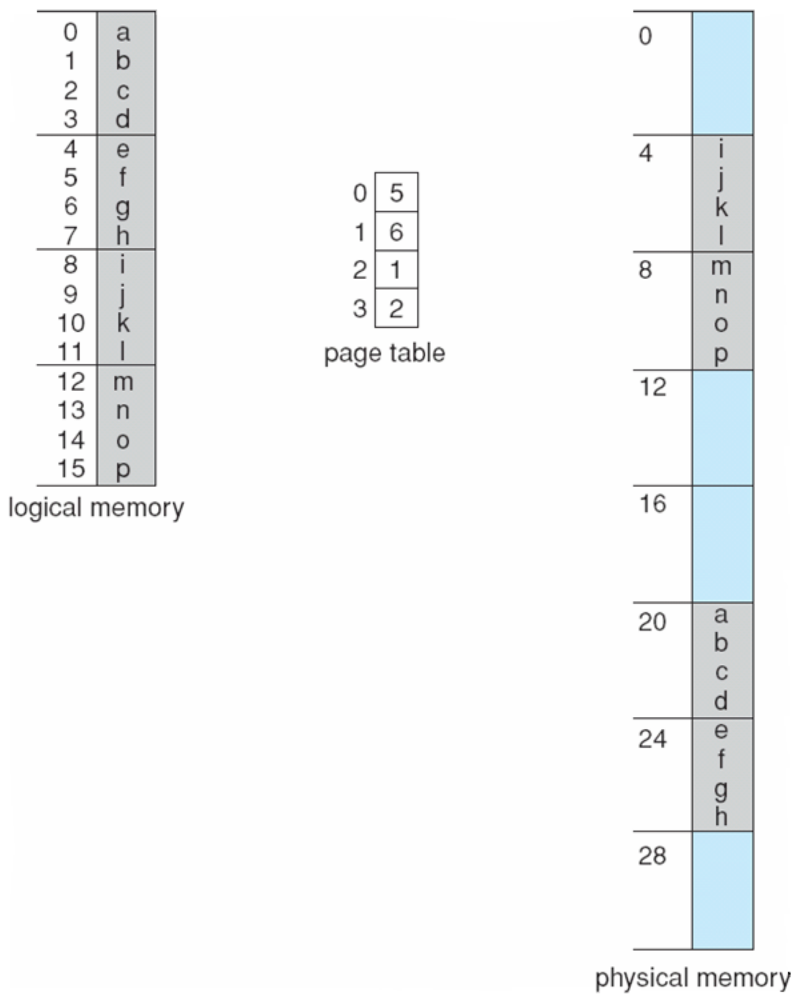
Paging Hardware



Paging Model of Logical and Physical Memory

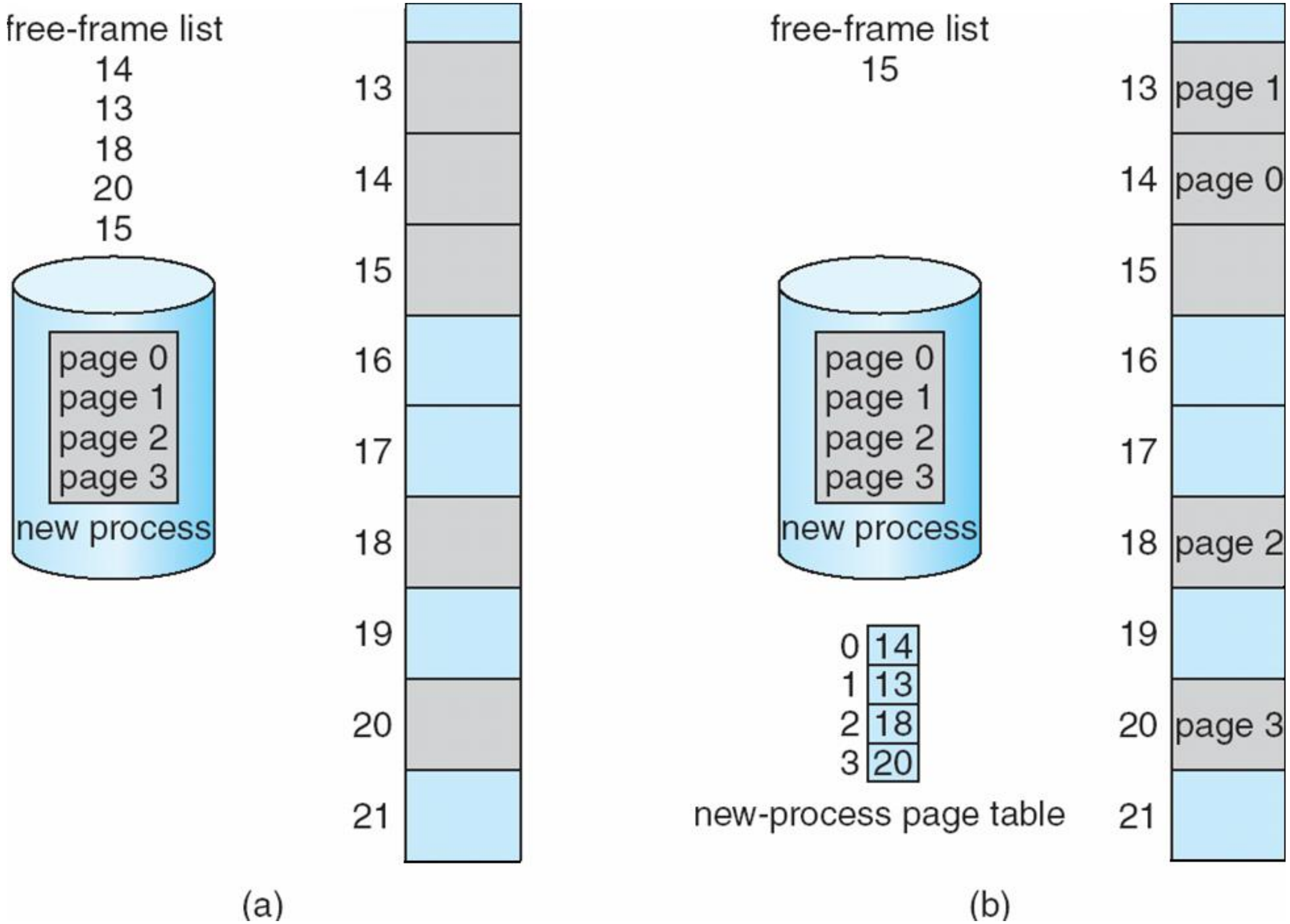


Paging Example



32-byte memory and 4-byte pages

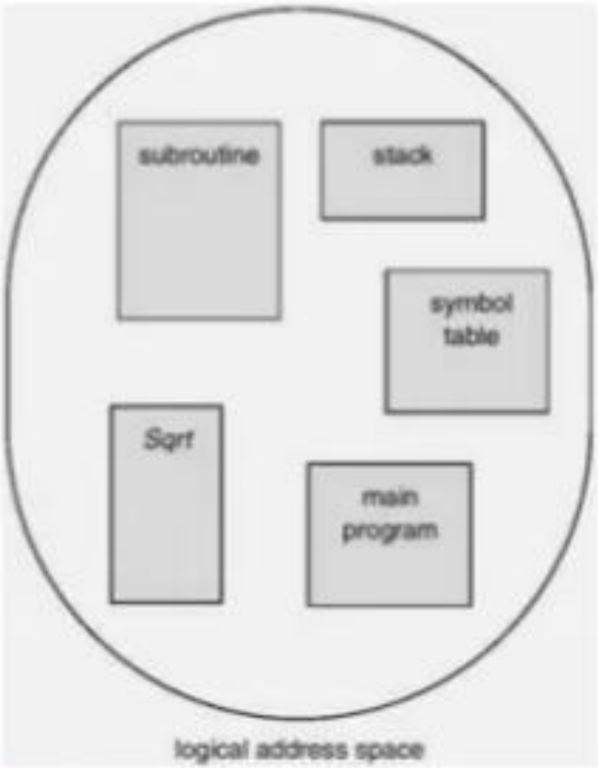
Free Frames



Before allocation

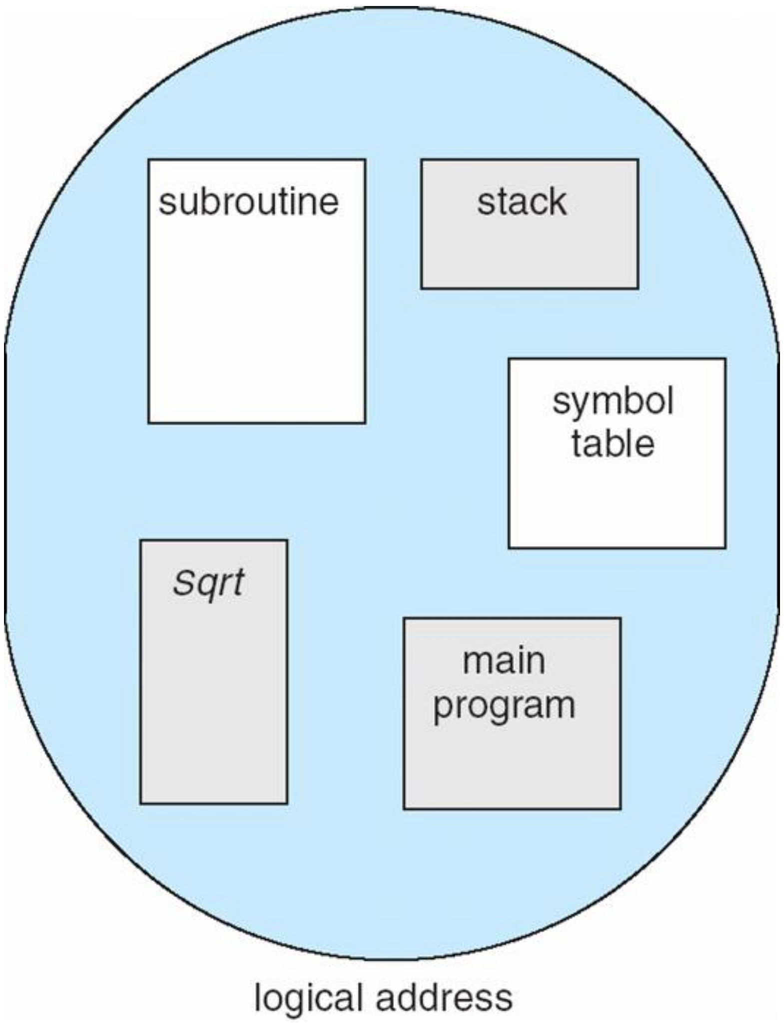
After allocation

Segmentation

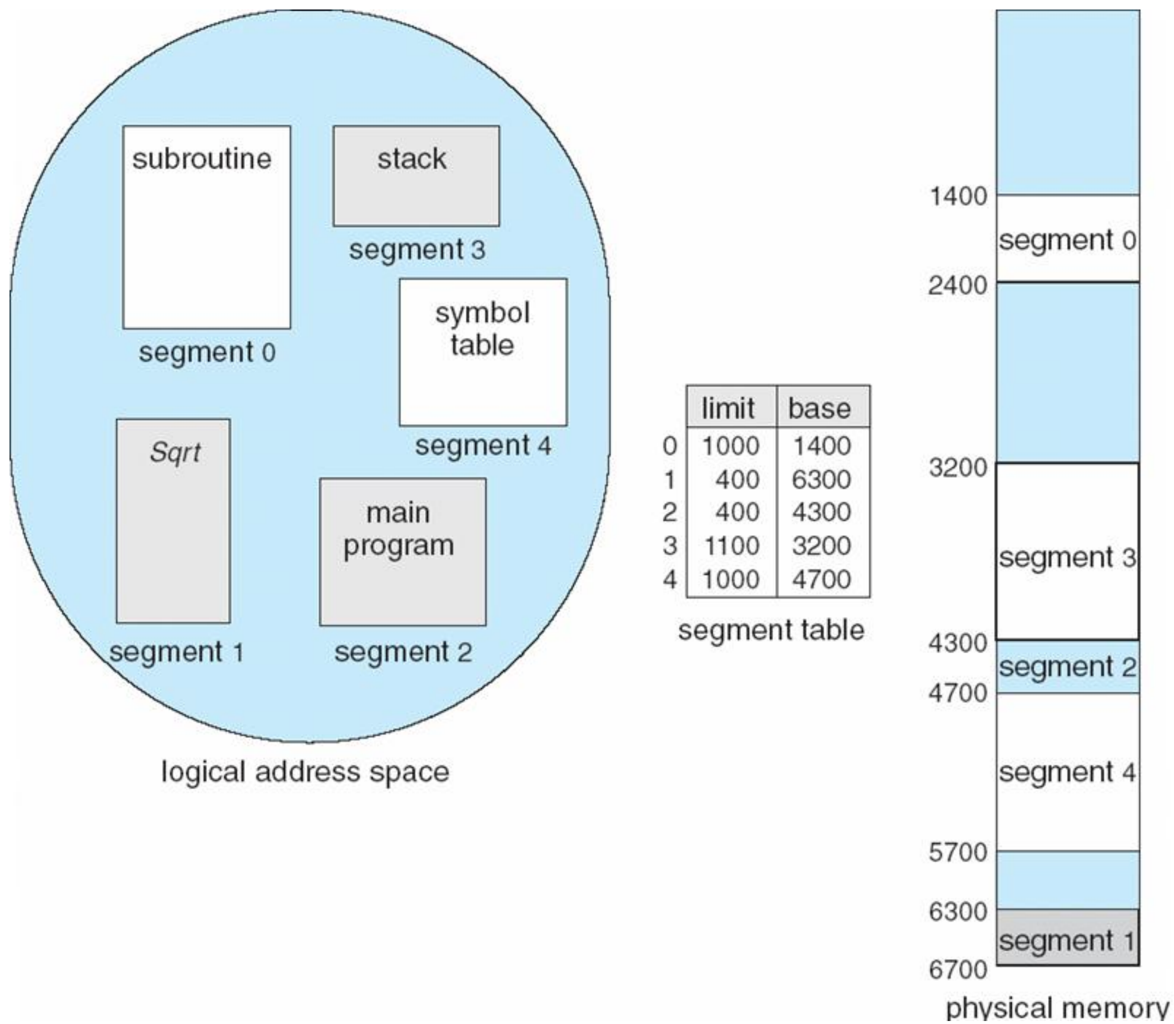


- User View of logical memory
  - Linear array of bytes
    - Reflected by the 'Paging' memory scheme
  - A collection of variable-sized entities
    - User thinks in terms of "subroutines", "stack", "symbol table", "main program" which are somehow located somewhere in memory.]
- Segmentation supports this user view. The logical address space is a collection of segments.

User's View of a Program



Example of Segmentation

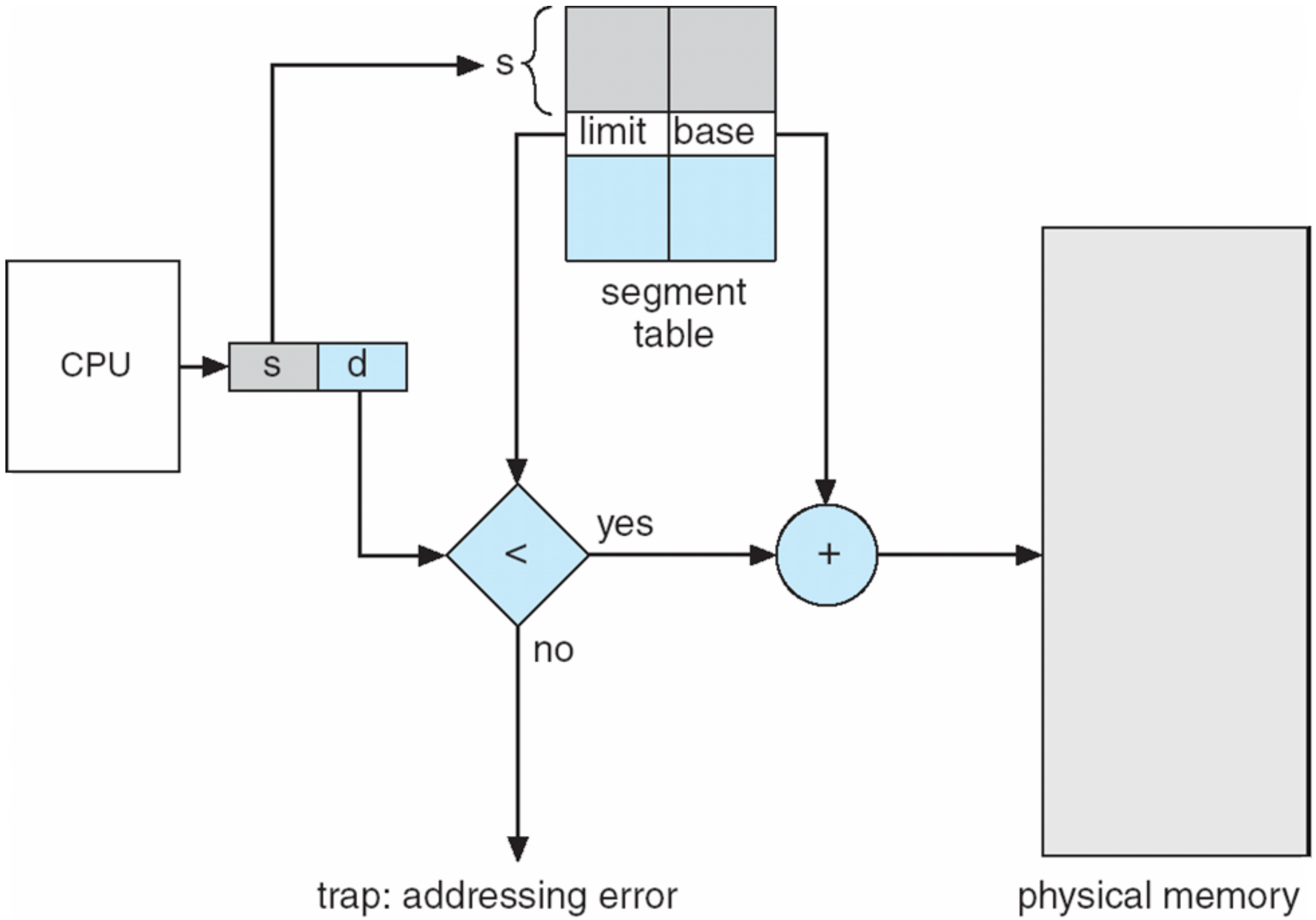


Segmentation

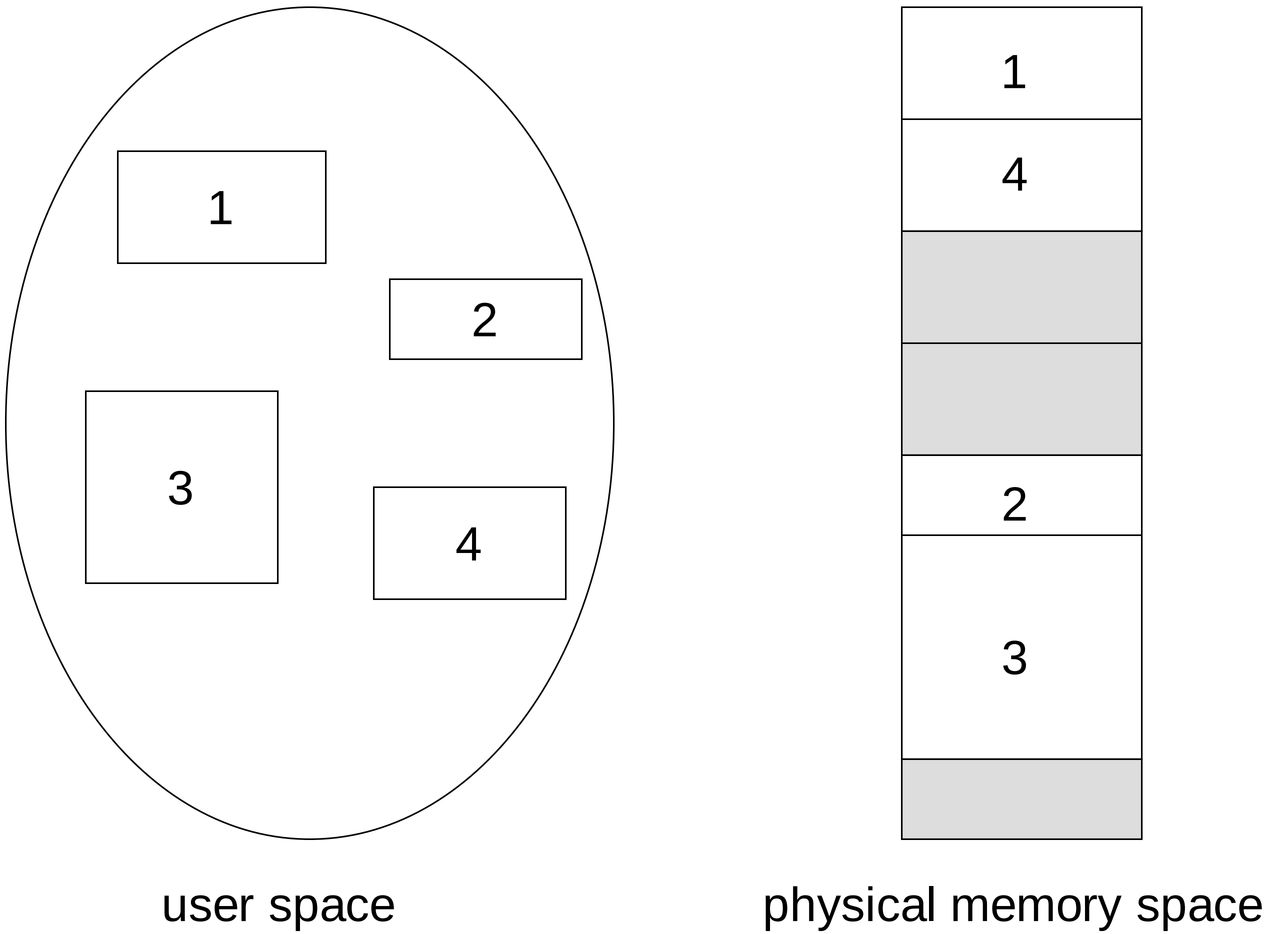
- Memory-management scheme that supports user view of memory
- A program is a collection of segments
  - A segment is a logical unit such as:

    main program
    procedure
    function
    method
    object
    local variables, global variables
    common block
    stack
    symbol table
    arrays

Segmentation Hardware



Logical View of Segmentation



Segmentation Architecture

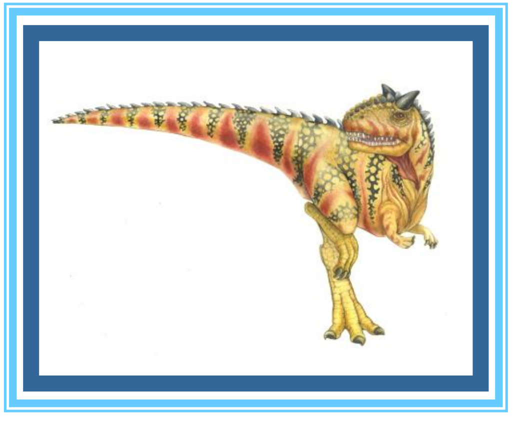
- Logical address consists of a two tuple:

  <segment-number, offset>,

- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - **base** – contains the starting physical address where the segments reside in memory
  - **limit** – specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;

  segment number ***s*** is legal if ***s*** < **STLR**

End of Chapter 8



Classical Problems of Synchronization

- In this tutorial we will discuss about various classical problem of synchronization.
- Semaphore can be used in other synchronization problems besides Mutual Exclusion.
- Below are some of the classical problem depicting flaws of process synchronaization in systems where cooperating processes are present.
- We will discuss the following three problems:
  - Bounded Buffer (Producer-Consumer) Problem
  - Dining Philosophers Problem
  - The Readers Writers Problem

Bounded Buffer Problem

- Because the buffer pool has a maximum size, this problem is often called the **Bounded buffer problem**.
- This problem is generalised in terms of the **Producer Consumer problem**, where a **finite** buffer pool is used to exchange messages between producer and consumer processes.
- Solution to this problem is, creating two counting semaphores "full" and "empty" to keep track of the current number of full and empty buffers respectively.
- In this Producers mainly produces a product and consumers consume the product, but both can use of one of the containers each time.
- The main complexity of this problem is that we must have to maintain the count for both empty and full containers that are available.

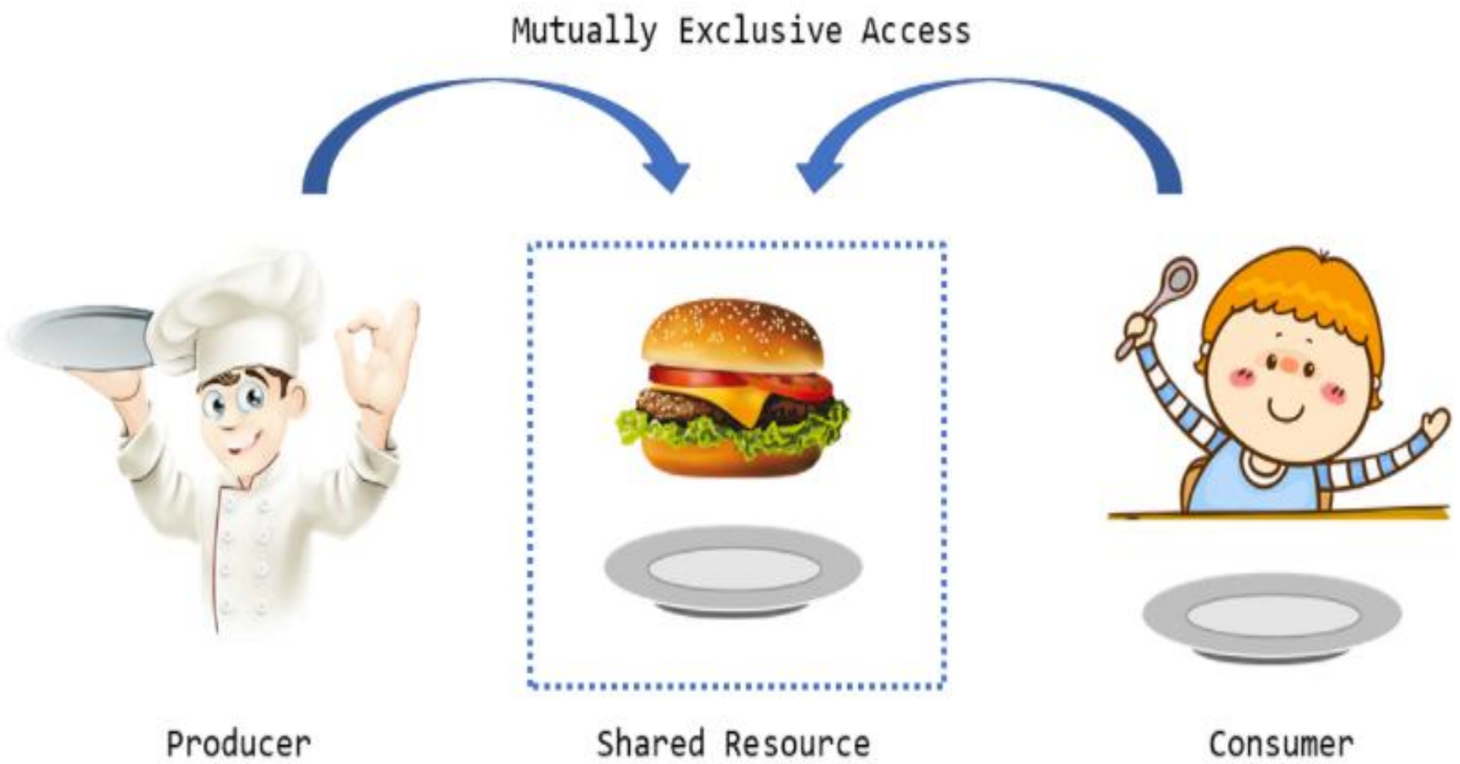
Dining Philosophers Problem

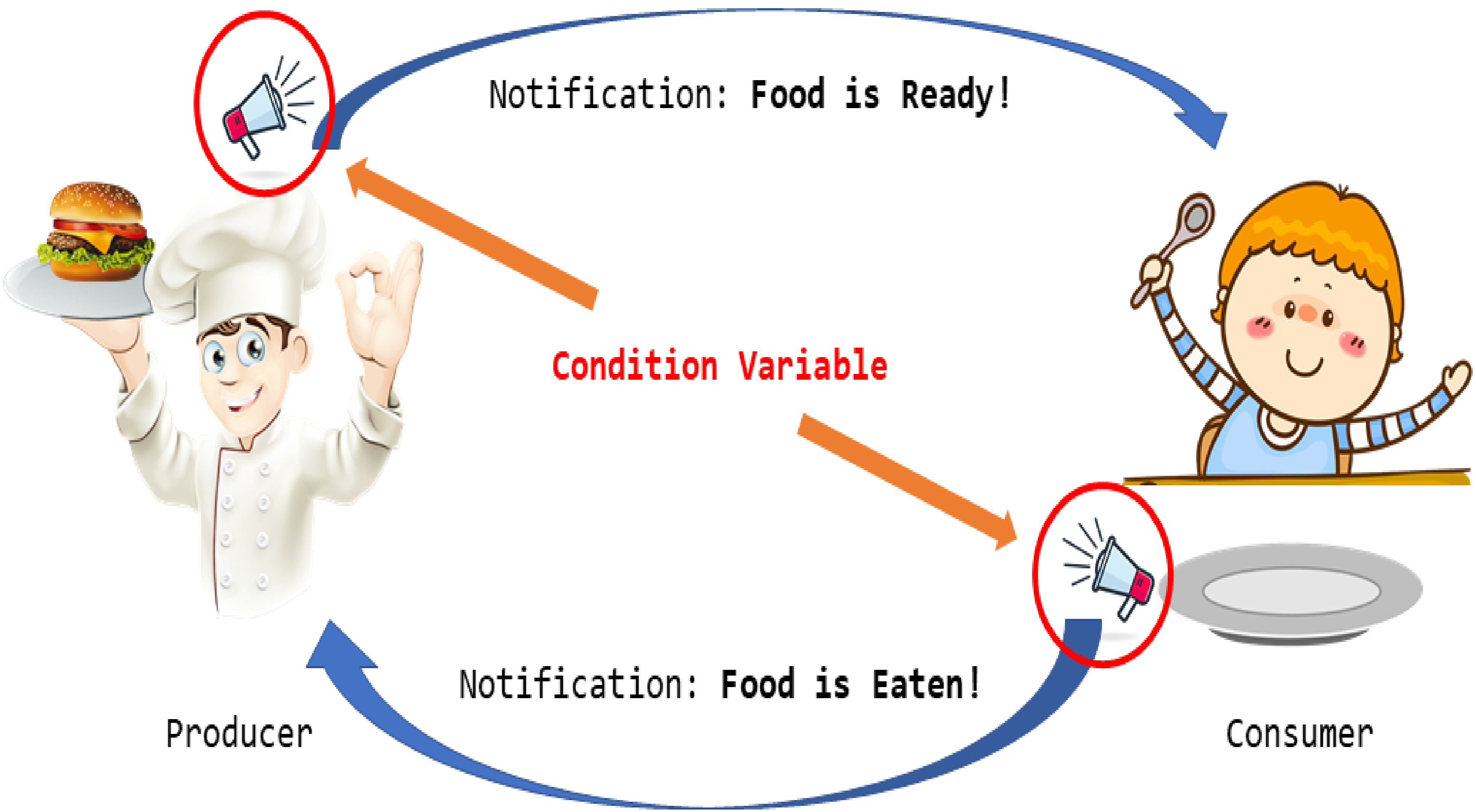
- The dining philosopher's problem involves the allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.

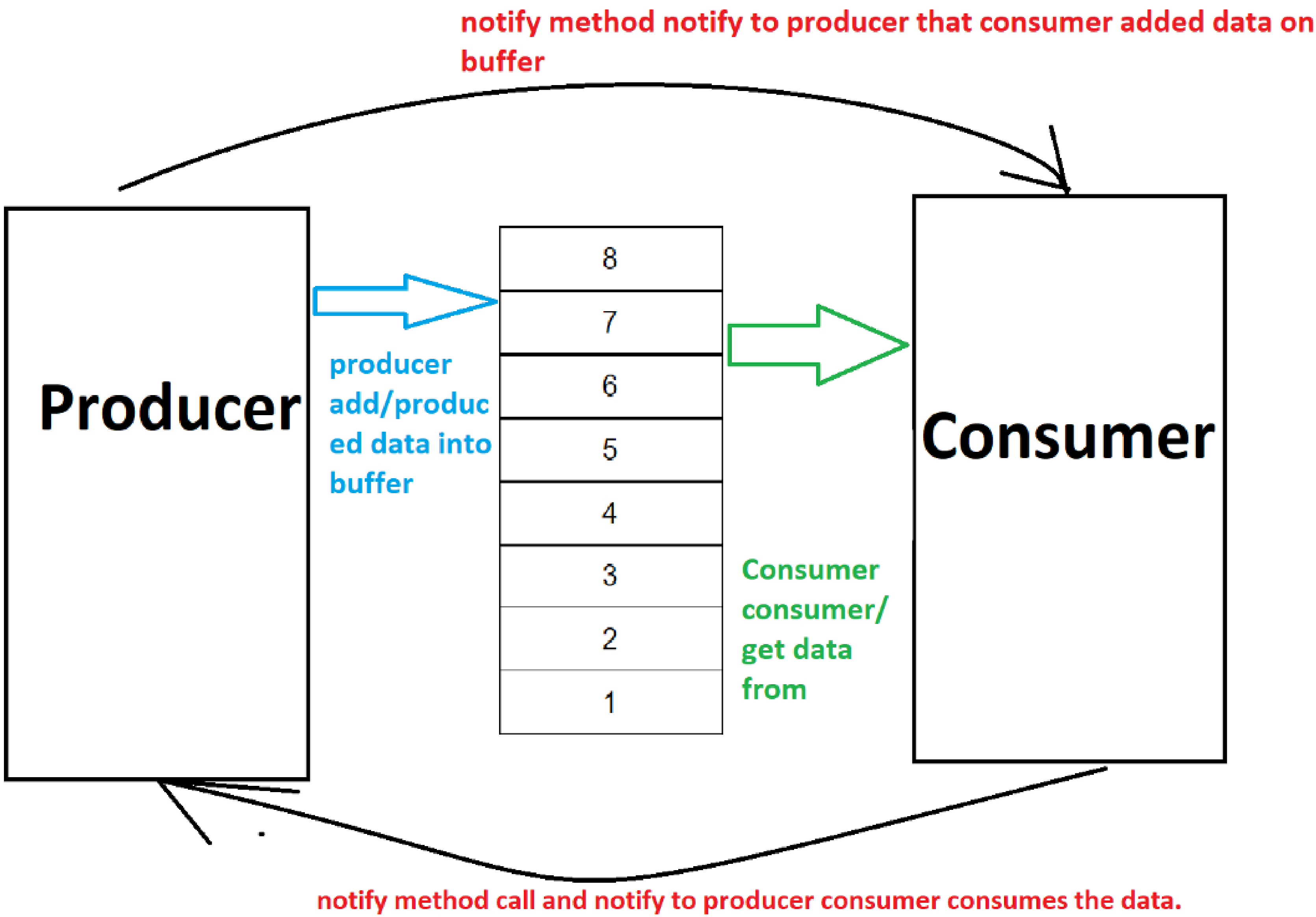
- There are five philosophers sitting around a table, in which there are five chopsticks/forks kept beside them and a bowl of rice in the centre, When a philosopher wants to eat, he uses two chopsticks - one from their left and one from their right. When a philosopher wants to think, he keeps down both chopsticks at their original place.

The Readers Writers Problem

- In this problem there are some processes(called **readers**) that only read the shared data, and never change it, and there are other processes(called **writers**) who may change the data in addition to reading, or instead of reading it.
- There are various type of readers-writers problem, most centred on relative priorities of readers and writers.
- The main complexity with this problem occurs from allowing more than one reader to access the data at the same time.


Mutually Exclusive Access
Producer
Shared Resource
Consumer


Notification: **Food is Ready!**
Condition Variable
Notification: **Food is Eaten!**
Producer
Consumer

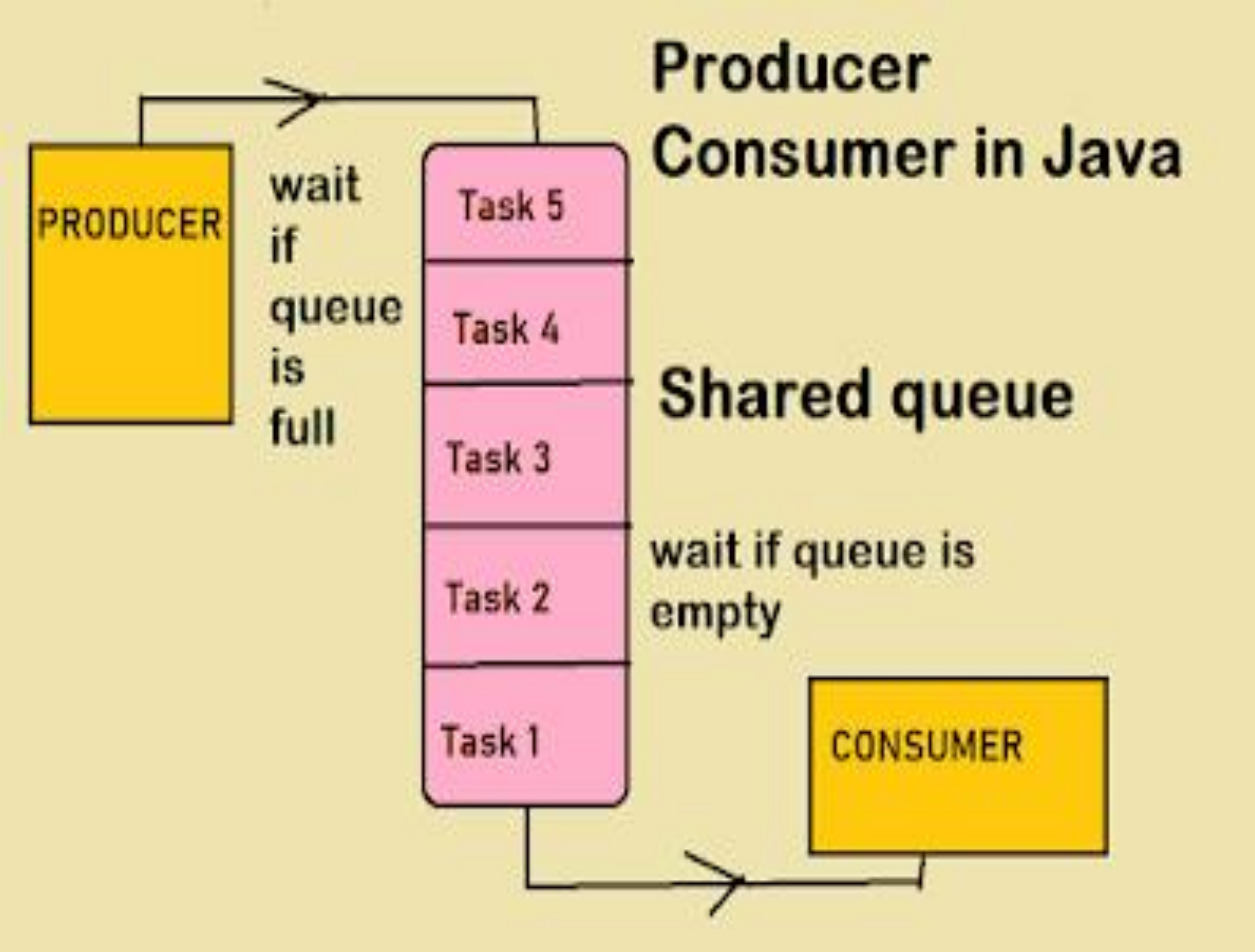

notify method notify to producer that consumer added data on buffer
Producer
producer add/produced data into buffer
8
7
6
5
4
3
2
1
Consumer consumer/ get data from
Consumer
notify method call and notify to producer consumer consumes the data.

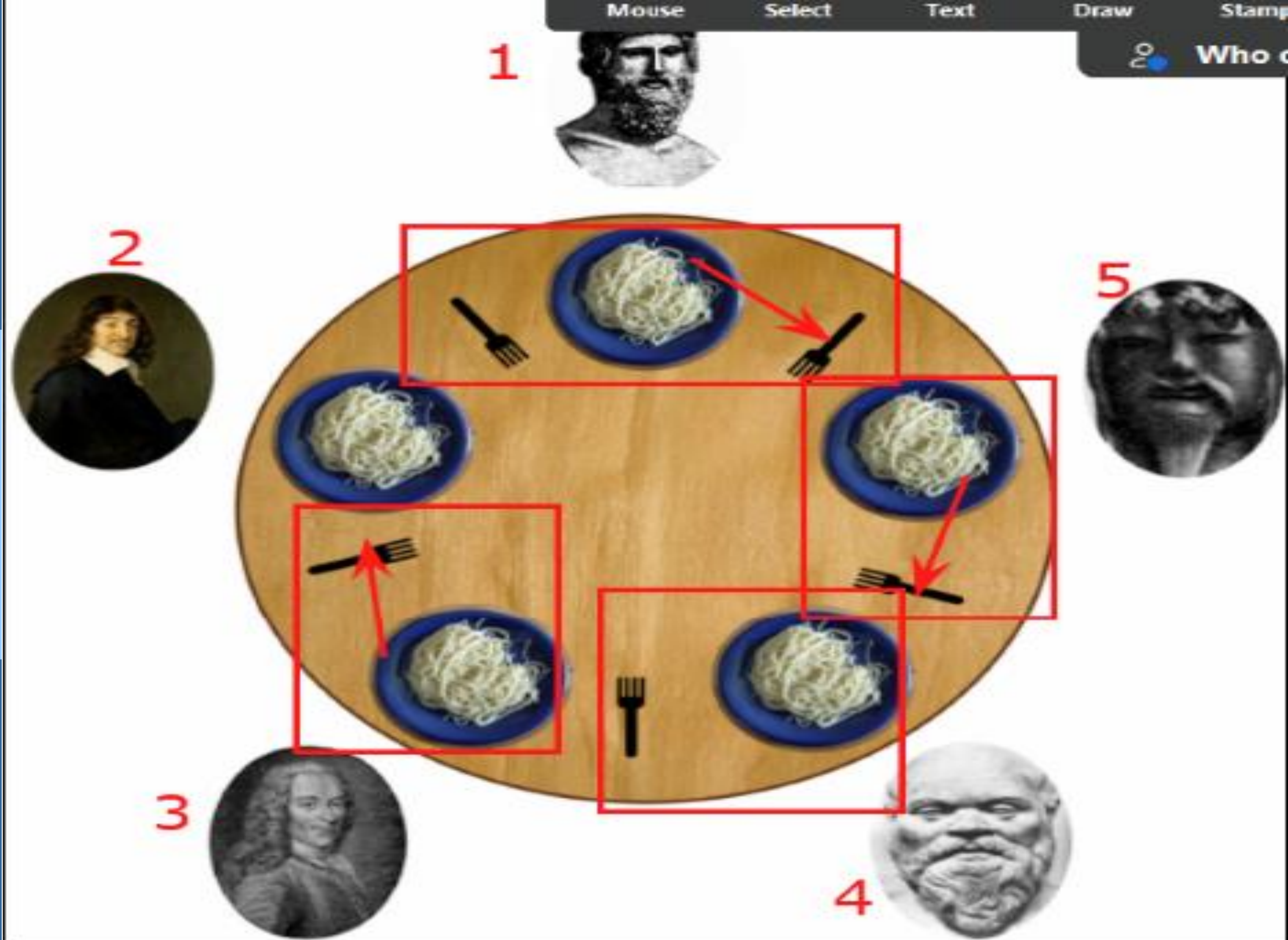
Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits


Producer
Consumer in Java
PRODUCER
wait
if
queue
is
full
Task 5
Task 4
Task 3
Task 2
Task 1
Shared queue
wait if queue is
empty
CONSUMER


Mouse
Select
Text
Draw
1
2
3
4
5