Operating Systems Concepts

Review I

Final exam structure

I. Fundamentals

II. CPU scheduling, Memory management, Disk scheduling

III. Resource allocation

IV. Operating System in C Language

Types of resources managed by an OS

- CPUs (processes)
- main memory and virtual memory
- secondary storage
- I/O devices
- file system and user interface
- communications
- provides protection and security

PROCESSES

- **Process** – a **program in execution**; process execution must progress in sequential fashion

- The state of a process is defined in part by the **current activity of that process.**

    - **new**: The process is being created

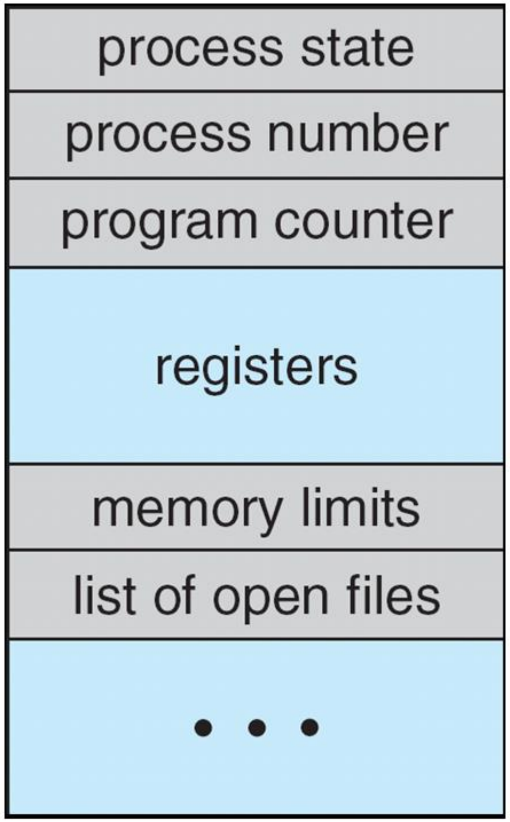
    - **running**: Instructions are being executed

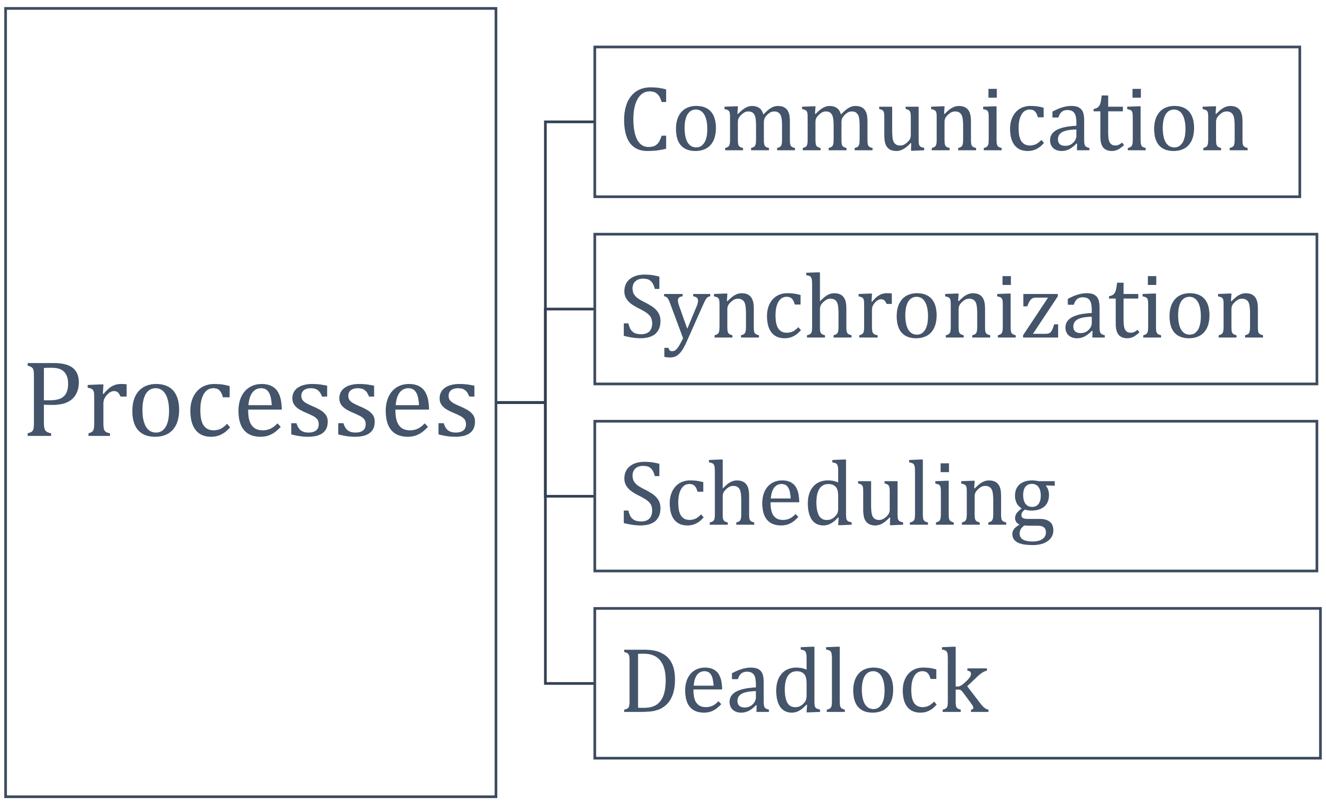
    - **waiting**: The process is waiting for some event to occur

    - **ready**: The process is waiting to be assigned to a processor

    - **terminated**: The process has finished execution

- **Process Control block** is a data structure used for storing the information about a process.
- Each & every process is identified by its own PCB.
- It is also called as **context of the process**.
- PCB of each process resides in the main memory.
- PCB of all the processes are present in a linked list.

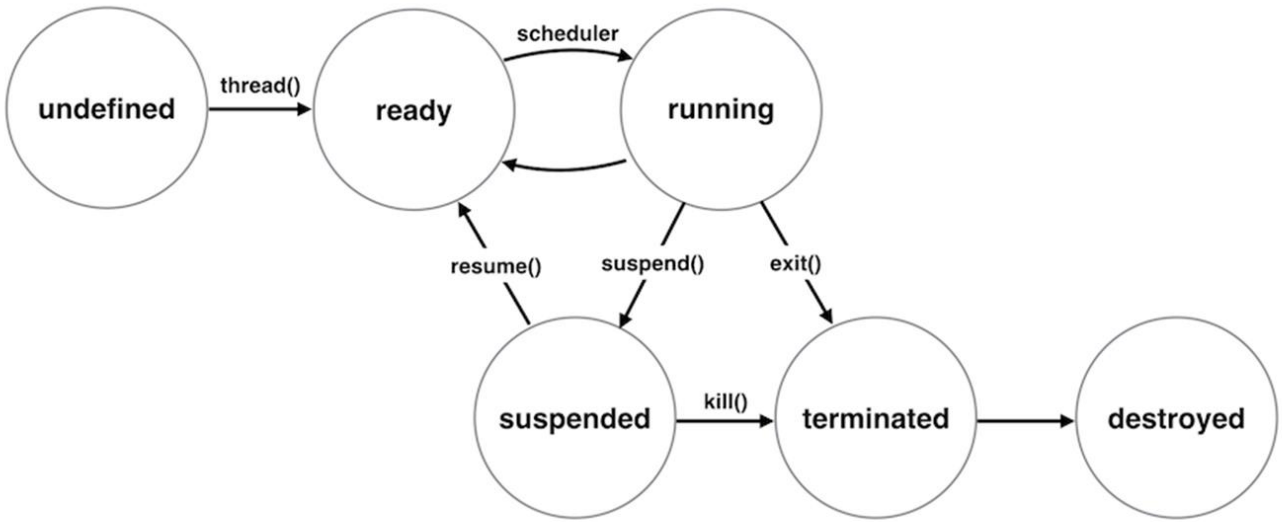



Processes
Communication
Synchronization
Scheduling
Deadlock

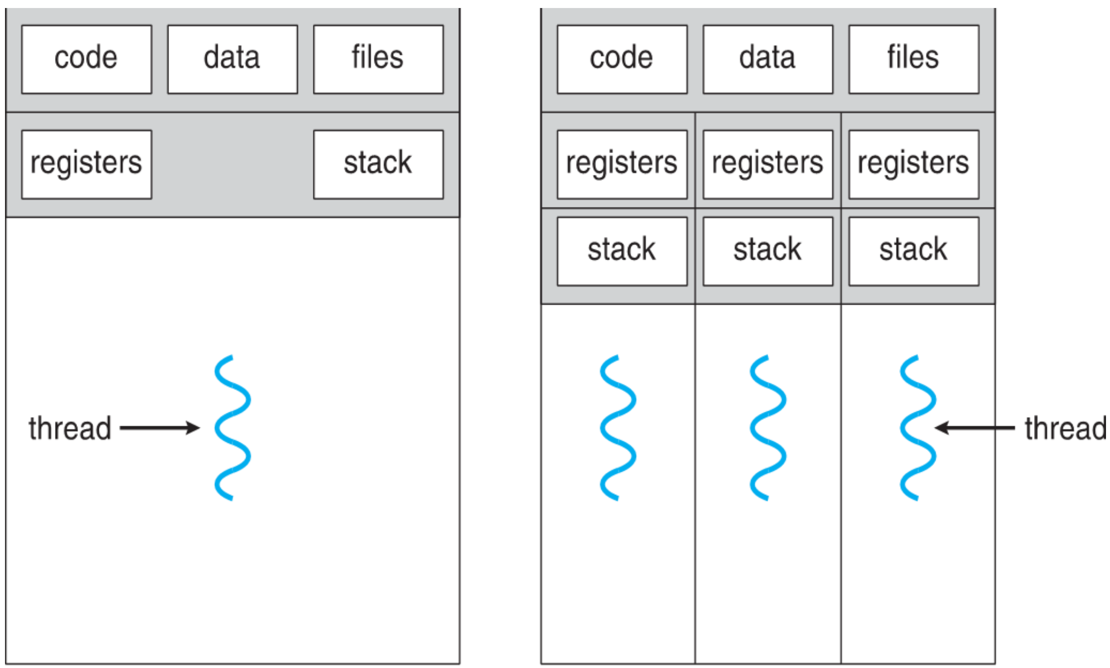
THREADS

- A fundamental unit of CPU utilization that forms the basis of multithreaded computer systems

- Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.

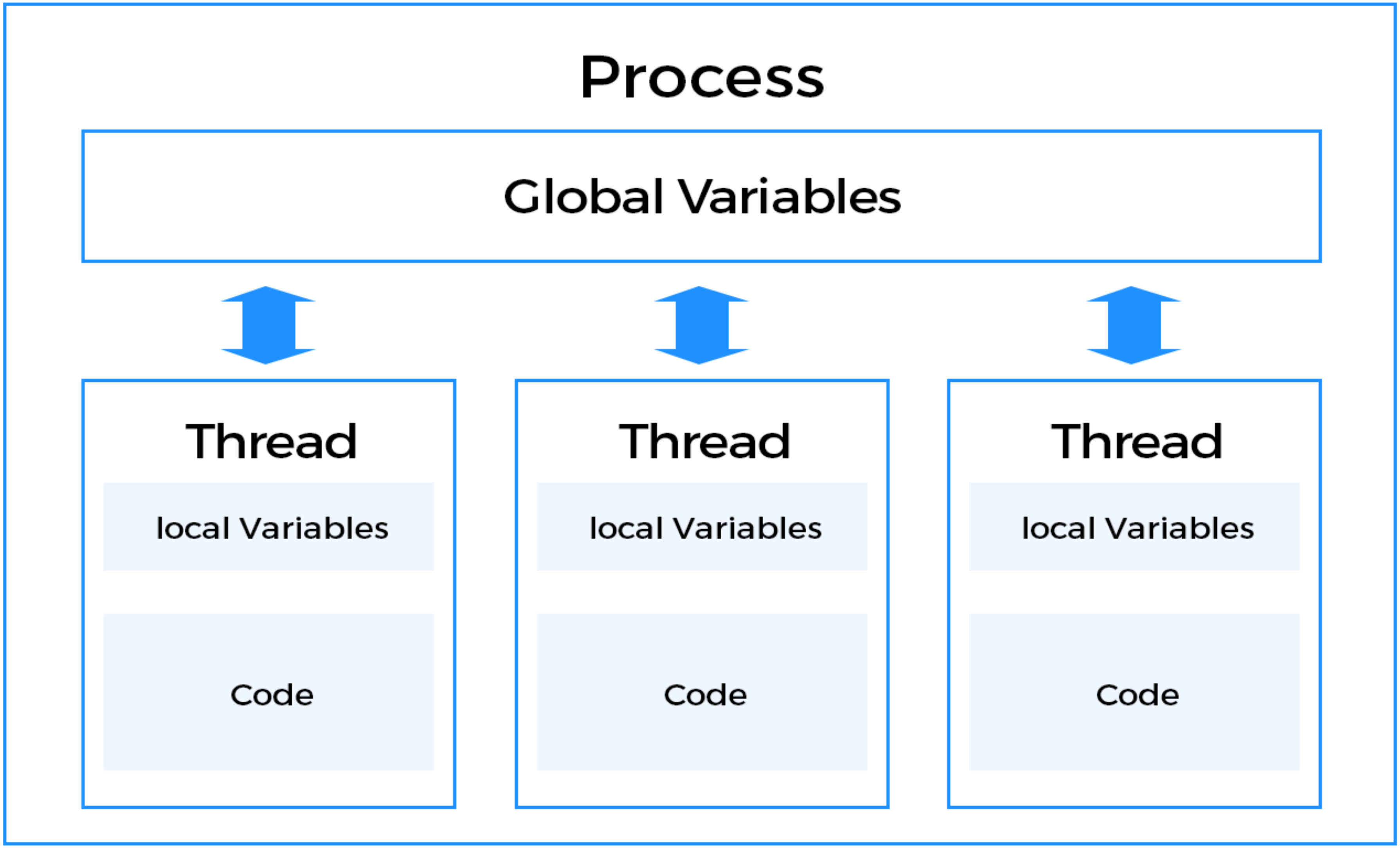


undefined
thread()
ready
scheduler
running
resume()
suspend()
exit()
suspended
kill()
terminated
destroyed



code
data
files
registers
stack
thread
code
data
files
registers
registers
registers
stack
stack
stack
thread
single-threaded process
multithreaded process

Process vs Thread



User Level & Kernel Level Thread

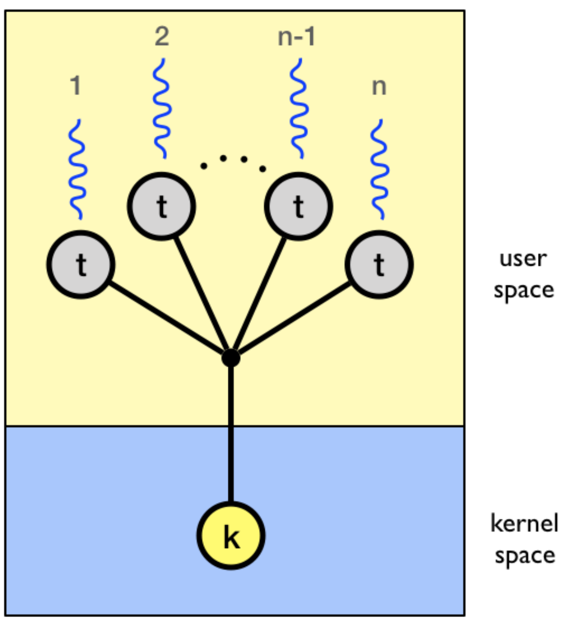
User Level Threads	Kernel Level Thread
User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads
User level thread is generic and can run on any operating system	. Kernel level thread is specific to the operating system.
Multi-threaded application cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

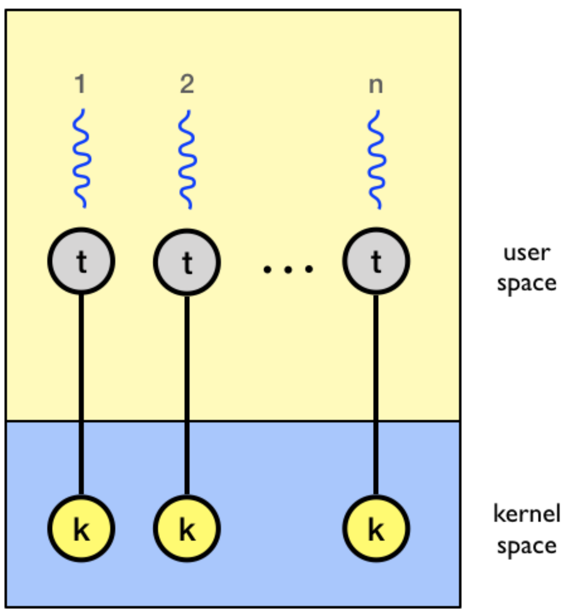
Multithreading models are three types

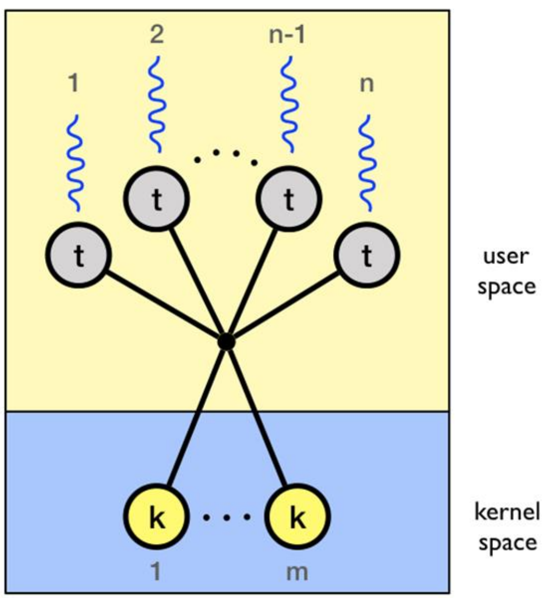
Many – to – One

One – to – One

Many – to - Many







EXPLICIT THREADING - the programmer creates and manages threads.

IMPLICIT THREADING - the compilers and **run-time libraries** create and manage threads.

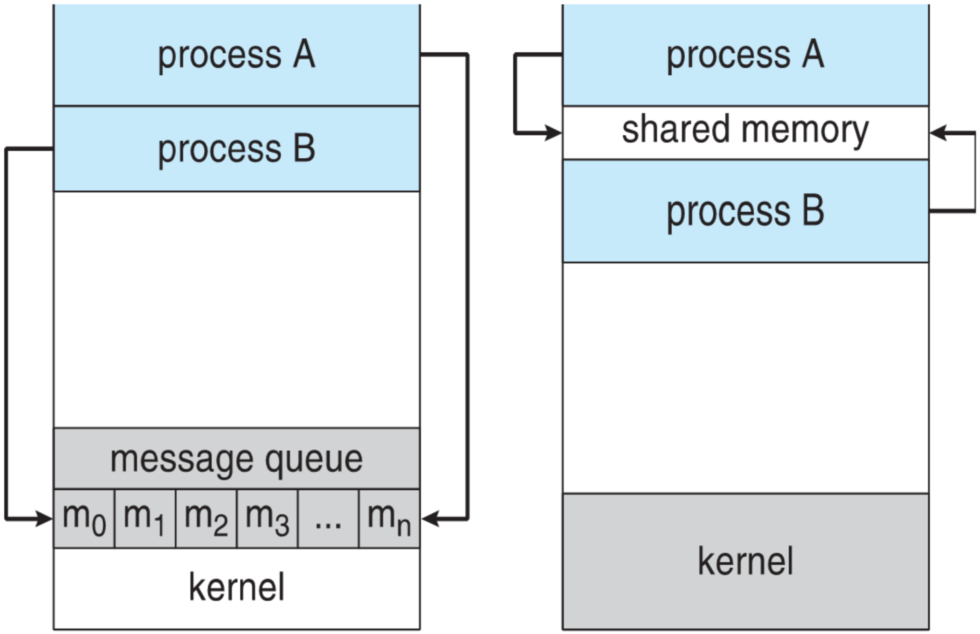
- Processes
  - **IP Communication**
  - Synchronization
  - Scheduling
  - Deadlock

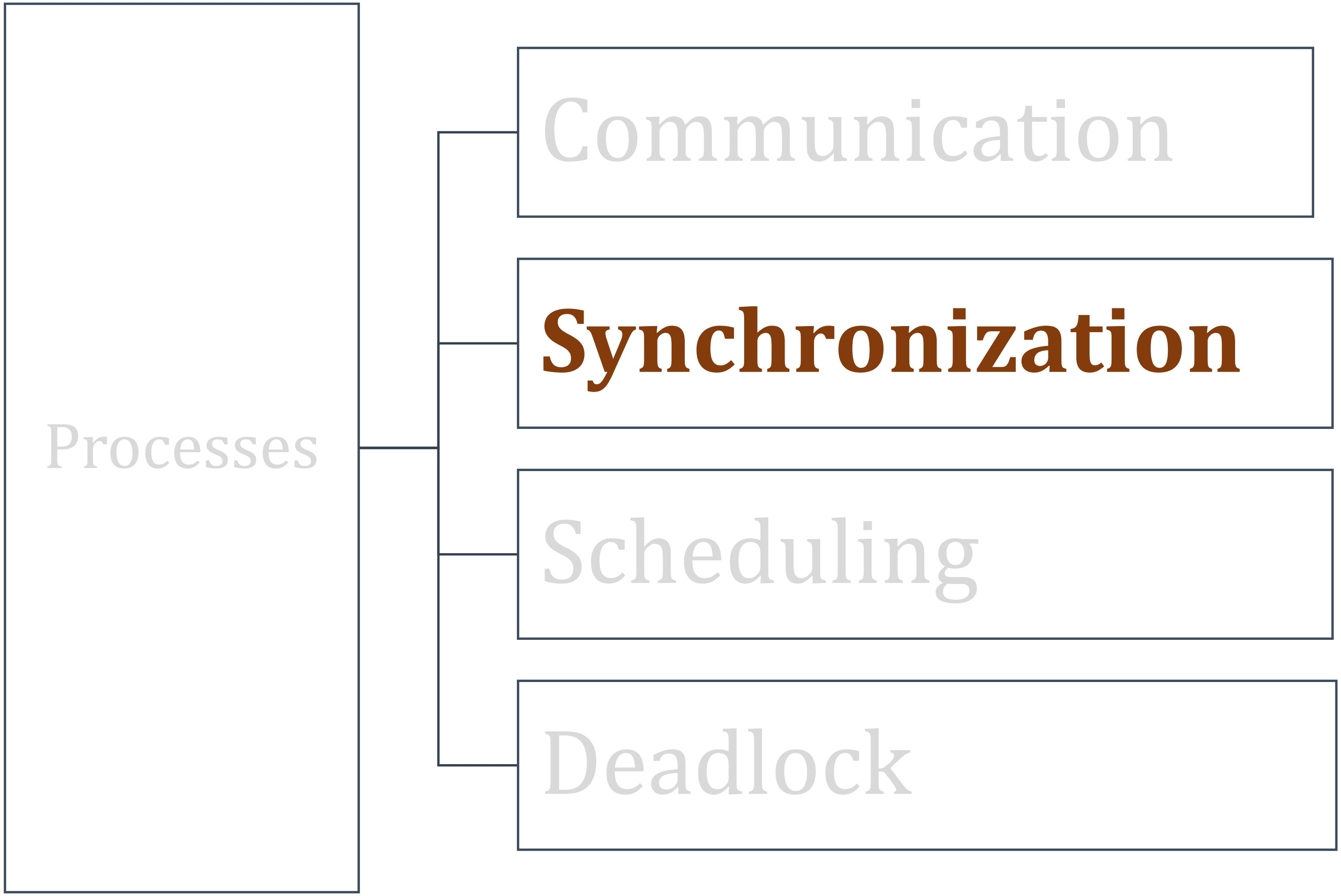
INTER-PROCESS COMMUNICATION

Independent Processes - neither affect other processes or be affected by other processes.

Cooperating Processes - can affect or be affected by other processes.

(a) Message passing. (b) Shared memory.





Processes
Communication
Synchronization
Scheduling
Deadlock

THE CRITICAL SECTION PROBLEM

code that tries to enter the CS

Some arbitrary code (such as initialization).

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Code that constitutes the CS.

Code that leaves the CS

can contain any code

```
int i = 7;              //global variable or shared resource
void increment() {
        i++;                // critical section of code
}
int main() {
        thread T1,T2;
        T1(increment);
        T2(increment);
}
```

What will be the final value of i after both thread have completed execution?

Case 1

Thread 1	Thread 2
get i (7)	—
increment i (7 -> 8)	—
write back i (8)	—
—	get i (8)
—	increment i (8 -> 9)
—	write back i (9)

Case 2

Thread 1	Thread 2
get i (7)	get i (7)
increment i (7 -> 8)	—
—	increment i (7 -> 8)
write back i (8)	—
—	write back i (8)

The entry- and exit-sections that surround a critical section must satisfy the following correctness requirements:

- **Mutual exclusion** - no two process/thread can be simultaneously present inside critical section at any point in time
- **Progress** - no process/thread running outside the critical section should block the other interesting process from entering into a critical section when in fact the critical section is free
- **Bounded waiting** - No process/thread should have to wait forever to enter into the critical section.

SOLUTIONS TO THE CRITICAL SECTION PROBLEM

- **Peterson's solution** - restricted to **two processes** that alternate execution between their critical sections and remainder sections.

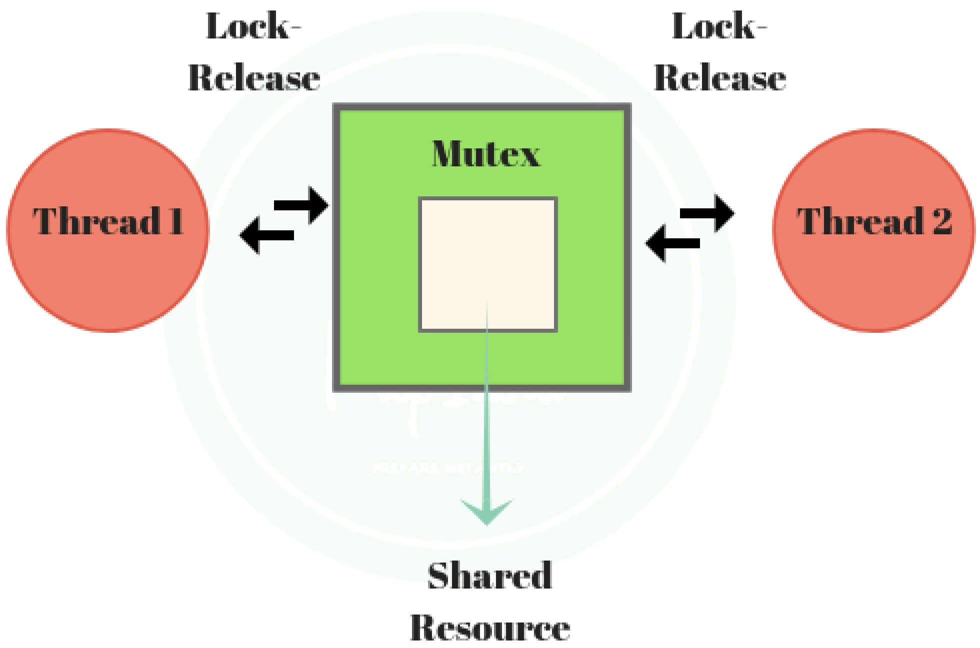
- **Hardware solution/Synchronization Hardware -** rely on some special machine instructions

- **Mutex lock -** software to protect critical regions and avoid race conditions.

- **Semaphores -** similar to the mutex lock, provide sophisticated ways for the process to synchronize their activities.

MUTEX LOCKS / MUTUAL EXCLUSION

- a mutex is locking mechanism used to synchronize access to a resource.



- Mutex object is locked or unlocked by the process/thread requesting or releasing the resource

SEMAPHORES

There are two main types of semaphores:

- **COUNTING SEMAPHORE** – allow an arbitrary resource count. Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances.

- **BINARY SEMAPHORE** – This is also known as **mutex lock**. It can have only two values – 0 and 1.

Its value is initialized to 1. It is used to implement the solution of critical section problem with multiple processes.

SEMAPHORES

A semaphore is a signaling mechanism.

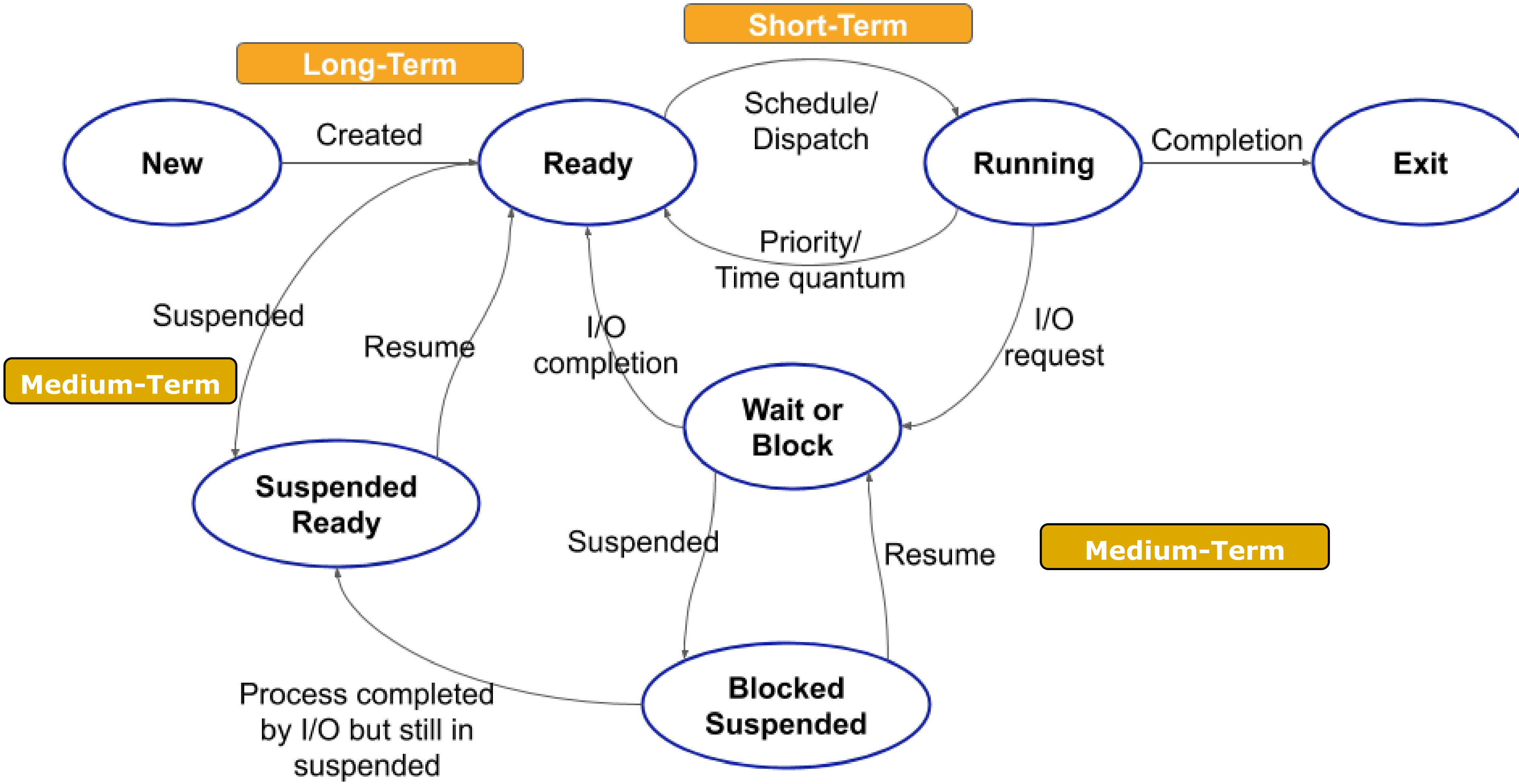
- The **wait** operation decrements the value of its argument S, if it is positive. If *S* is negative or zero, then no operation is performed.

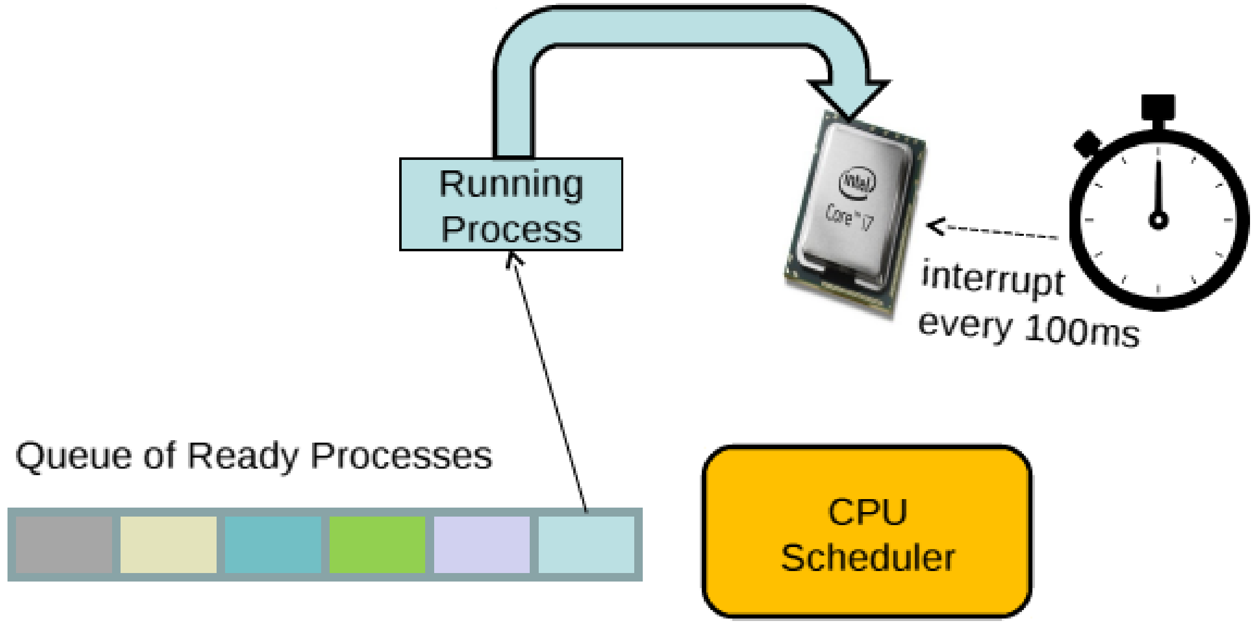
```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

- The **signal** operation increments the value of its argument *S*:

```
signal(S) {
    S++;
}
```

- Processes
  - Communication
  - Synchronization
  - **Scheduling**
  - Deadlock


Short-Term
Long-Term
New
Created
Ready
Schedule/
Dispatch
Running
Completion
Exit
Priority/
Time quantum
Suspended
Resume
I/O
completion
I/O
request
Medium-Term
Wait or
Block
Suspended
Ready
Suspended
Resume
Medium-Term
Process completed
by I/O but still in
suspended
Blocked
Suspended



The scheduling in which a running process **can be interrupted** if a high priority process enters the queue and is allocated to the CPU is called **PREEMPTIVE SCHEDULING.**

The scheduling in which a running process **cannot be interrupted** by any other process is called **NON-PREEMPTIVE SCHEDULING.**

Scheduling Algorithms

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-First (SJF) Scheduling
- Shortest Remaining Time First
- Priority Scheduling
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling
- Multilevel Feedback Queue Scheduling

Real-Time CPU Scheduling

Real-time systems are those in which **the time is crucial** to their performance.

- **Priority-Based Scheduling** - each process a priority based on its importance
- **Rate-Monotonic Scheduling** - static priority *(the shorter the period = the higher the priority)* with preemption.
  - Missed Deadlines with Rate Monotonic Scheduling
- **Earliest-Deadline-First Scheduling** – dynamic priority *(the earlier the deadline = the higher the priority)* with preemption.
- **Proportional Share Scheduling** - T shares are allocated among all processes in the system.

Processes

- Communication
- Synchronization
- Scheduling
- **Deadlock**

Conditions Of Deadlock

There are certain conditions that give rise to a deadlock:

- **Mutual exclusion**: Some resource types cannot allow multiple processes to access it at the same time $\Rightarrow$ only one process at a time can use a resource
- **Hold and wait**: When all the processes are holding some resources and waiting for other resources, a deadlock may occur
- **No preemption**: If a resource cannot be pre-empted, it may lead to a deadlock.
- **Circular wait**: The Resource Allocation Graph RAG has a cycle.

Ensure that the system will ***never*** enter a deadlock state:

- **Deadlock prevention**
- **Deadlock avoidance**
- **Deadlock Detection**

If a system is in safe state $\Rightarrow$ no deadlocks

If a system is in unsafe state $\Rightarrow$ possibility of deadlock

Prevention $\Rightarrow$ eliminating any of the four conditions

Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

How to check if system is in safe state

Total resources

R1	R2	R3
15	8	8

Available

R1	R2	R3
3	3	2

Process	*Max*			*Alloc*			*Need(Max – Alloc)*		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	5	6	3	2	1	0	3	5	3
P2	8	5	6	3	2	3	5	3	3
P3	4	8	2	3	0	2	1	9	0
P4	7	4	3	3	2	0	4	2	3
P5	4	3	3	1	0	1	3	3	2

Available = Available – Request[i];

Allocation[i]= Allocation[i] + Request[i];

Need[i] = Need[i] – Request[i];

MEMORY

Objectives of a Memory Management (MM) System

- **Protection**
- **Relocation**
- **Sharing**
- **Logical Organization of memory**
- **Physical Organization of memory**

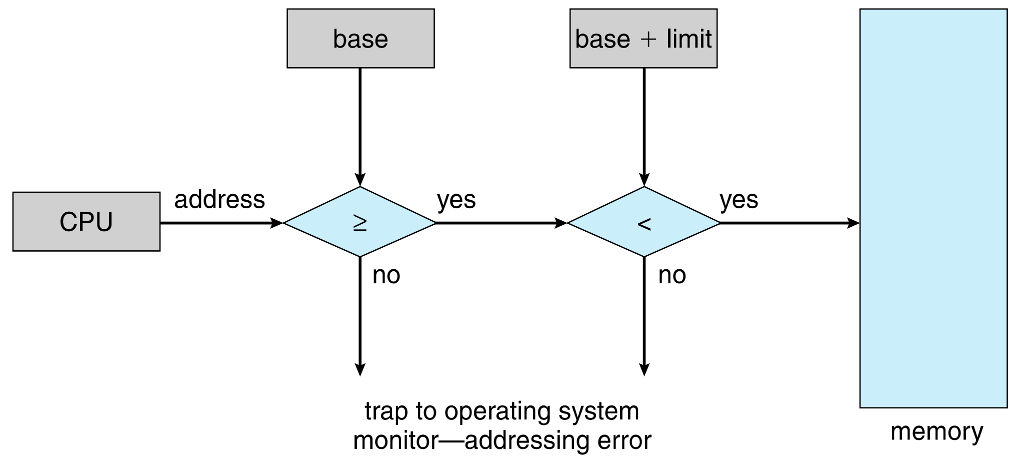
Logical address – generated by the CPU; also referred to as *virtual address.*

When the process starts executing, *relative or logical addresses* are generated by the CPU.

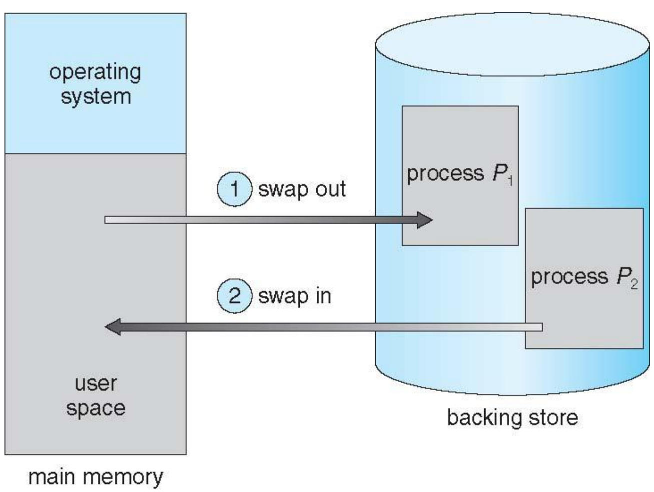
Physical address – address seen by the memory unit *(the absolute addresses)*

A pair of **base register / relocation register** and **limit registers** define the *logical address space*

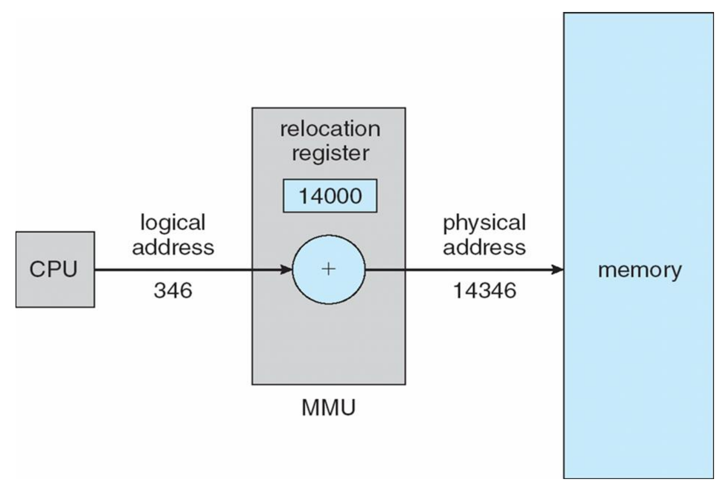
- To translate all the memory references of the process, there must be an origin or base address in the memory.
- All relative addresses generated are added in this base address to get the new location in the memory.



Swapping



Relocation



Logical Organization of Memory:

Allocation

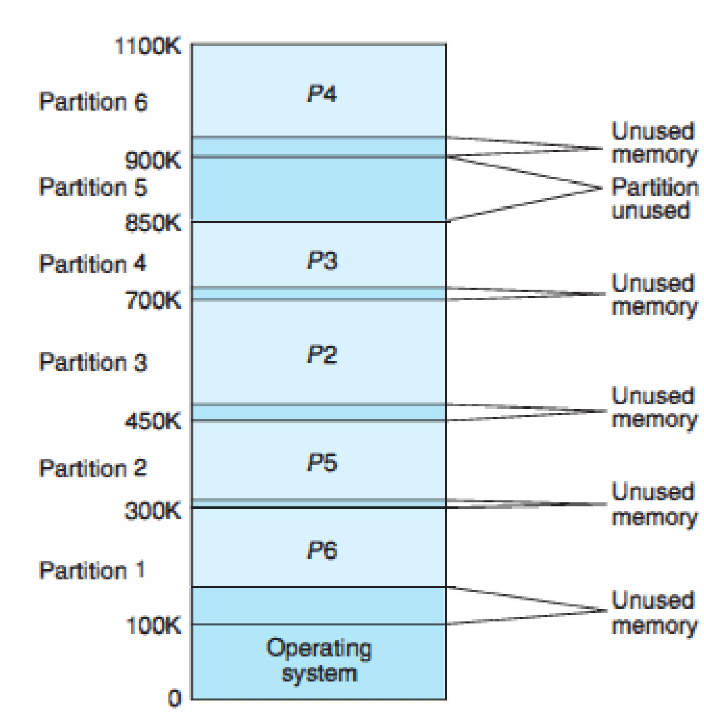
The memory allocation can be classified into two methods:

- **contiguous memory allocation** - assigns consecutive memory blocks to a process.
- **non-contiguous memory allocation** - assigns different blocks of memory in a nonconsecutive manner to a process.

Contiguous Allocation

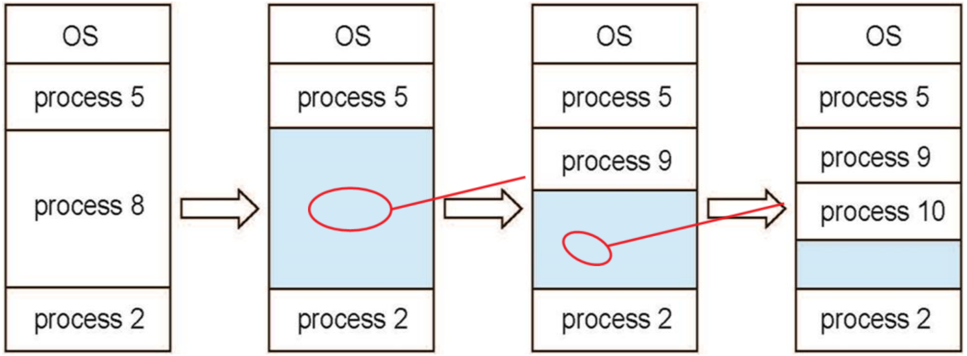
Fixed/Static Partitioning

- fixed partitions can be of *equal* or *unequal* sizes.



Variable/Dynamic Partitioning

- list of free memory blocks **(holes)** to find a hole of a suitable size whenever a process needs to be loaded into memory.



Placement Algorithm:

First-fit: Allocate the first hole that's big enough

Best-fit: Allocate the smallest hole that's big enough

Worst-fit: Allocate the largest hole

Fragmentation

Internal fragmentation

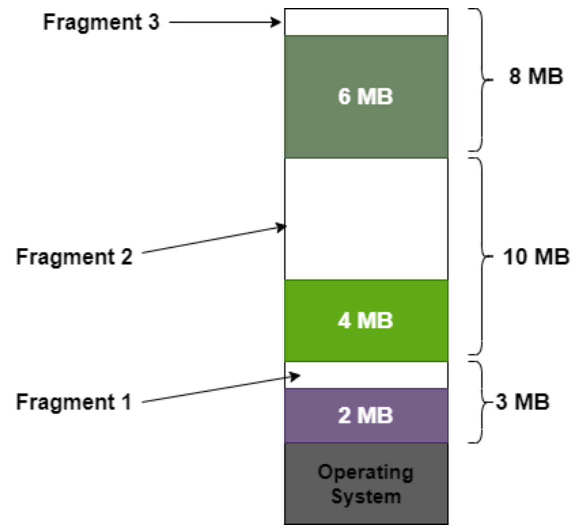
- occurs in fixed size blocks, because the last allocated process is not completely filled.

Solution:

- can be reduced by using variable sized memory blocks rather than fixed sized.

External fragmentation

- occurs with variable size segments, because some holes in memory will be too small to use.



Solutions:

Compaction - moving all occupied areas of storage to one end of memory. This leaves one big hole.

Non-contiguous memory allocation: **Segmentation** and **Paging**.

To Be Continued