By Bharadwaj Rachakonda – mailto:rbharadwaj022@gmail.com LinkedIn

perplexity

Operating Systems

Introduction

An **Operating System (OS)** is system software that provides an environment for programs to run.

Functions of an Operating System

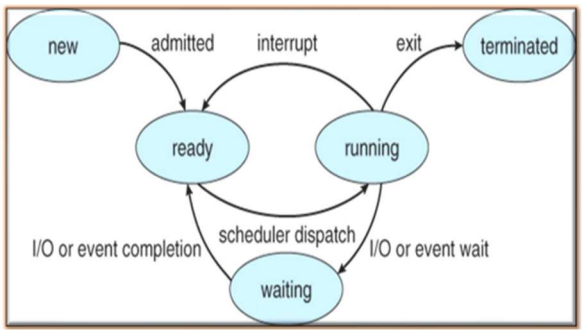
- Memory Management
- Process Management
- File Management
- Device Management
- Security
- Communication
- Error Detection
- Resource Allocation

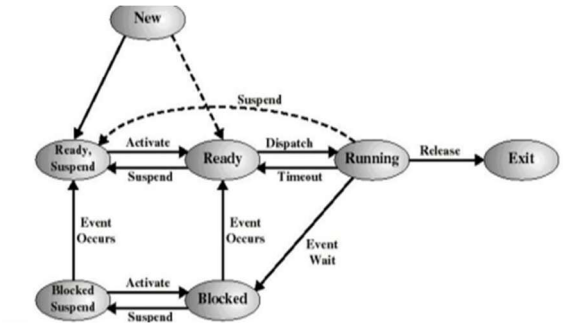
Process Management

Types of Processes

- **CPU-bound Process:** Requires more CPU time; spends more time in running state.
- **I/O-bound Process:** Requires more I/O time; spends more time in waiting state.

Process States





1. New
2. Ready
3. Running
4. Waiting
5. Terminated
6. Suspended Ready
7. Suspended Block

Process Control Block (PCB)

Each process is represented by a PCB containing:

- **Process State:** Current state (new, ready, running, etc.)
- **Program Counter:** Address of next instruction
- **CPU Registers:** general-purpose registers, etc.
- **CPU Scheduling Info:** Priority, scheduling queues, parameters
- **Memory Management Info:** page/segment tables
- **Accounting Info:** CPU usage, process numbers
- **I/O Status Info:** List of allocated I/O devices and open files

Threads

- **Process:** Single thread of execution.
- **Thread:** Lightweight process, requires fewer resources than a full process.

Comparison Basis	Process	Thread
Definition	A process is a program under execution i.e an active program.	A thread is a lightweight process that can be managed independently by a scheduler.
Context switching time	Processes require more time for context switching as they are more heavy.	Threads require less time for context switching as they are lighter than processes.
Memory Sharing	Processes are totally independent and don't share memory.	A thread may share some memory with its peer threads.
Communication	Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes .
Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation.	Threads require less time for creation.
Time for termination	Processes require more time for termination.	Threads require less time for termination.

Scheduling

Schedulers

- **Short-term Scheduler (CPU Scheduler):** Selects processes from ready memory for execution.
- **Long-term Scheduler (Job Scheduler):** When more processes are submitted than that can be executed immediately, they are spooled to a mass-storage device and are kept there for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution.

- **Scheduling Queues**
  - **Job Queue:** All processes in the system
  - **Ready Queue:** Processes in memory, ready to execute
  - **Device Queue:** Processes waiting for a particular I/O device

Context Switch

Switching the CPU between processes by saving and restoring states.

By Bharadwaj Rachakonda – mailto:rbharadwaj022@gmail.com LinkedIn

Operations on Processes

Process Creation

- Processes may create multiple child processes.
- Identified by a unique Process ID (PID).
- Parent process options:
  - Wait for child to terminate (`wait()`)
  - Run concurrently (background tasks)
  - Wait for child later (parallel processing)

Process Termination

- Occurs via the `exit()` system call.
- Parent may terminate child if:
  a. Resource usage exceeded
  b. Task no longer required
  c. Parent exits and children cannot continue

Special Cases

- **Zombie Process:** Terminated process whose parent hasn't called `wait()`.
- **Orphan Process:** Child remaining after parent termination; UNIX/Linux reassigns to `init` process.

Interprocess Communication (IPC)

- **Independent Process:** Cannot affect/affected by others.
- **Cooperating Process:** Can affect/be affected by others.

Purposes

- Information sharing
- Computational speedup
- Modularity

- Convenience

IPC Mechanisms

- **Shared Memory:** Processes exchange information via shared regions.
- **Message Passing:** Processes communicate via messages, no shared address space.

Scheduling Concepts

- **Preemptive Scheduling:** A process is forcibly removed from CPU (context switch)
- **Non-Preemptive Scheduling:** Process voluntarily relinquishes CPU
- **Dispatcher:** Gives control of CPU to selected process – assigns process to a CPU
- **Dispatch Latency:** Time to switch processes

Key Terms

- **Throughput:** Number of processes completed per unit time
- **Turnaround Time:** Total time from submission to completion
- **Waiting Time:** Total time spent in ready queue
- **Response Time:** Time from request submission to first response

Scheduling Algorithms

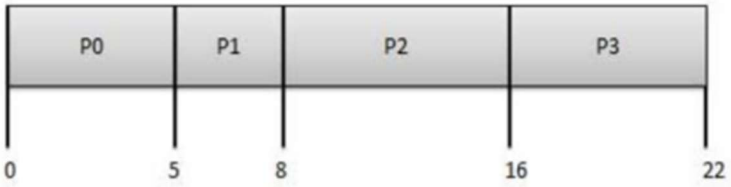
First-Come, First-Served (FCFS)

- Non-preemptive
- Susceptible to **starvation** & **convoy effect** (CPU-bound process delays I/O-bound processes)

Consider the snapshot of a system given here

Process	Arrival Time	Execute Time
P0	0	5
P1	1	3
P2	2	8
P3	3	6

A Gantt chart is a horizontal bar chart developed as a production control tool in 1917 by Henry L. Gantt, an American engineer and social scientist.



Wait time of each process is as follows –

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest-Job-First (SJF)

- Preemptive and non-preemptive versions
- Ideal but impractical since future process lengths can't be predicted
- **Starvation** possible for longer jobs

Priority Scheduling

- Both preemptive and non-preemptive
- **Starvation** can be reduced by aging (increasing priority of older processes)

Round Robin (RR)

- Assigns fixed time quantum for each process
- Frequent context switches, leading to better response time

Multilevel Queue Scheduling

- Processes assigned to fixed queues based on category
- Each queue has its own scheduling algorithm
- No transfer between queues

Multilevel Feedback Queue Scheduling

- Processes can move between queues based on characteristics or aging
- Helps balance CPU and I/O-bound jobs; promotes fairness

Processor Affinity

- **Soft Affinity:** The system prefers, but does not guarantee, to keep a process on the same processor.
- **Hard Affinity:** A process specifies it must not be moved between processors (supported by Linux and some OSes).

Process Synchronization and Critical Section Problems

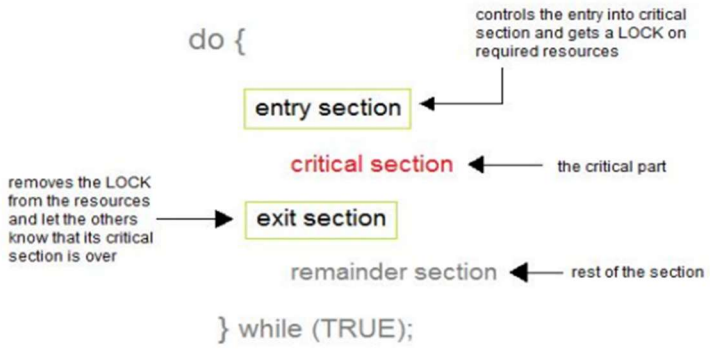
Race Condition

A **race condition** occurs when several processes access and manipulate shared data concurrently, and the result depends on the order of execution. To prevent race conditions, it is essential to synchronize processes and ensure only one process manipulates shared data at a time.

Critical Section Problem

Pieces of code that must be accessed in a mutually exclusive atomic manner by the contending threads are referred to as critical sections.

Components of a Critical Section Environment



- **Entry Section:** Code for requesting entry to the critical section.
- **Critical Section:** Code where only one process can execute at a time.
- **Exit Section:** Releases the critical section for others.

- **Remainder Section:** Code after leaving the critical section.

Requirements for a Solution

1. **Mutual Exclusion:** Only one process in the critical section at a time.
2. **Progress:** If no process is in its critical section, those wanting in must decide who enters next within a finite time.
3. **Bounded Wait:** There is a limit on the number of times other processes may enter their critical sections before a requesting process gains access.

Synchronization Methods

Peterson's Solution

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```


- A software-only solution for two-process synchronization.
- Allows other process to enter critical section before it by setting turn to other process.
- Uses two shared variables:
    - **int turn:** Indicates whose turn it is.
    - **boolean flag:** Indicates if a process wants to enter.
- Not guaranteed to be reliable on modern hardware.

Mutex Locks

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

- Software API providing locking (mutual exclusion) through **acquire** and **release** operations.

Acquire:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

Release:

```
release() {
   available = true;
}
```


- One problem with the implementation shown here is the busy loop used to block processes in the acquire phase. These types of locks are referred to as spinlocks, because the CPU just sits and spins while blocking the process

Semaphores

```
P(Semaphore s){
   while(S == 0);   /* wait until s=0 */
  s=s-1;
}

V(Semaphore s){
        s=s+1;
}
```

Note that there is Semicolon after while. The code gets stuck Here while s is 0.

- More robust than mutexes; use integer variables with only two atomic operations:
  - **wait** (P operation)
  - **signal** (V operation)

Better Implementation:

Semaphore Structure:

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Wait Operation:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            block();
        }
}
```

Signal Operation:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

- **Binary Semaphore:** Takes values 0 or 1 (functions as a mutex).
- **Counting Semaphore:** Can take any non-negative integer; tracks the number of available resources.

Scheduling Anomaly: Priority Inversion

Priority inversion occurs when a high-priority process is blocked by a lower-priority process holding a needed resource.

Classic Synchronization Problems

The Bounded Buffer Problem

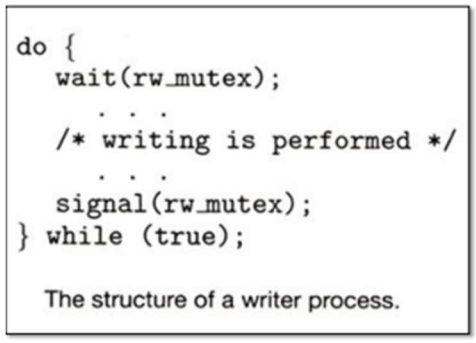
```
semaphore empty = BUFFER_SIZE;
semaphore full = 0;
semaphore mutex = 1;

void producer() {
    while (true) {
        produce_item();
        wait(empty);
        wait(mutex);
        add_item_to_buffer();
        signal(mutex);
        signal(full);
    }
}

void consumer() {
    while (true) {
        wait(full);
        wait(mutex);
        remove_item_from_buffer();
        signal(mutex);
        signal(empty);
        consume_item();
    }
}
```


- Extension of producer-consumer problem.
- Producers add items to a limited buffer; consumers remove items.
- Synchronization controls access to prevent buffer overflows and underflows.

The Readers-Writers Problem

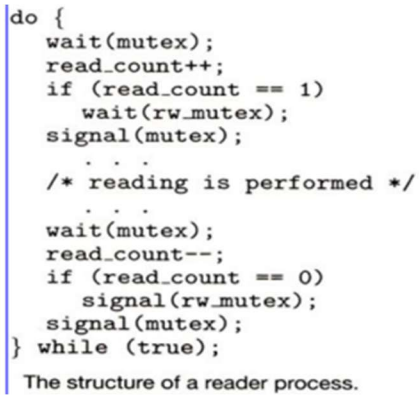
```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

- Processes either read (readers) or write (writers) shared data.
- Many readers can access data simultaneously, but writers need exclusive access.



```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

The structure of a writer process.



```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

The structure of a reader process.

Variations:

- **First Readers-Writers Problem:** Gives priority to readers; writers may face starvation.
- **Second Readers-Writers Problem:** Gives priority to writers; waiting readers are blocked when a writer arrives.

Semaphores Used:

- mutex and rw_mutex initialized to 1.
- read_count initialized to 0.
- rw_mutex is for both readers and writers; mutex protects updates to read_count.

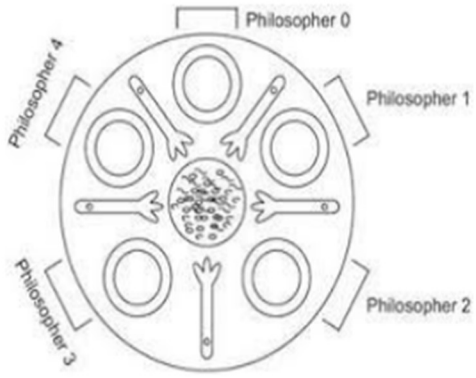
The semaphores mutex and rw_mutex are initialized to 1; read count is initialized to 0. The semaphore rw_mutex is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated.

Liveness

Liveness ensures that processes eventually make progress. A process waiting forever is a **liveness failure**.

Dining Philosophers Problem

A classic problem illustrating resource allocation and deadlock:



Scenario

- Five philosophers sit around a table with five chopsticks and a bowl of rice.
- Each alternates between **eating** and **thinking**.
- To eat, each philosopher needs two chopsticks (left and right).

Semaphore Solution

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```


- Represent chopsticks as semaphores: chopstick, each initialized to 1.
- Each hungry philosopher first wait on their left chopstick *(chopsticks[i])*, and then wait on their right chopstick *(chopsticks[(i + 1) % 5])*.
- A philosopher waits on their left, then right chopstick semaphore to eat, and signals (releases) both after eating.

Deadlock Example

If all philosophers simultaneously pick up their left chopstick, none can acquire the right one, causing a deadlock.

Potential Solutions

- Limit to four concurrent philosophers at the table.
- Allow philosophers to pick up chopsticks only when both are available.
- Implement asymmetric protocol: odd philosophers pick left up first, even pick right up first.

Deadlocks in Operating Systems

Introduction

In a multiprogramming environment, multiple threads may compete for limited resources. A **deadlock** occurs when threads wait indefinitely for resources held by each other, preventing any from progressing.

Resource Utilization Sequence

Threads use resources in the following sequence:

1. **Request**
2. **Use**
3. **Release**

Livelock

Livelock is a form of liveness failure where threads continuously change state in response to each other but do not progress.

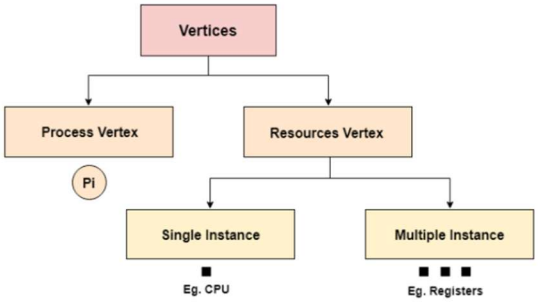
Deadlock Characteristics

Deadlock can occur only if all four conditions hold simultaneously:

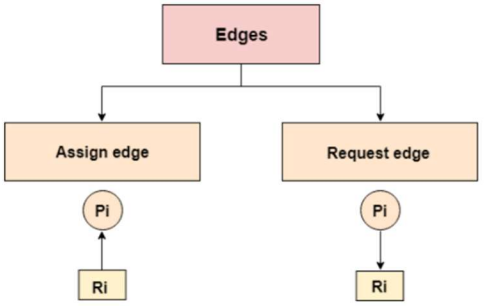
1. **Mutual Exclusion:** Resources are shared exclusively; only one process can use a resource at a time.
2. **Hold and Wait:** Processes hold at least one resource and wait for additional resources.
3. **No Preemption:** Resources cannot be forcibly taken away from a process once allocated.

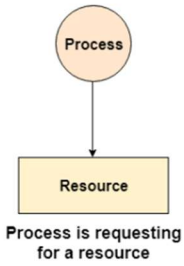
4. **Circular Wait:** A closed chain exists where each process holds a resource needed by the next process in the chain.

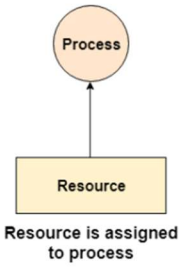
Resource Allocation Graphs



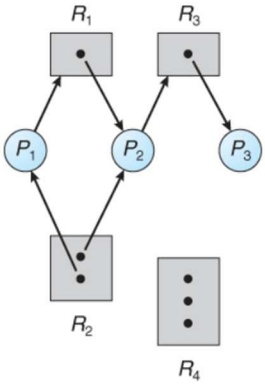
- **Components:**
  - **Process (Circle):** Represents a thread or process.
  - **Resource (Rectangle):** Represents resources; multiple instances shown as dots inside the rectangle.







- **Edges:**
  - **Allocation Edge (Resource → Process):** Resource is assigned to a process.
  - **Request Edge (Process → Resource):** Process is requesting a resource.



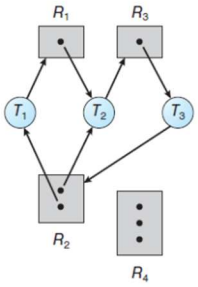


Figure 8.5 Resource-allocation graph with a deadlock.

Cycle Analysis:

- **No cycle:** System is not deadlocked.
- **Cycle present:** System may be in deadlock.

Deadlock Handling Methods

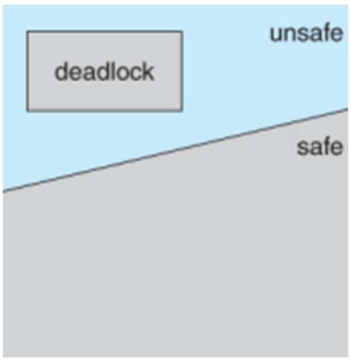
1. Prevention

Prevent at least one of the four necessary conditions for deadlock.

2. Avoidance

Grant resource requests only if the system remains in a **safe state**.

Safe & Unsafe States



- **Safe State:** Resources can be allocated in some order with no deadlock.
- **Unsafe State:** No guarantee that deadlock can be avoided.

Resource Allocation Graph with Claim Edges

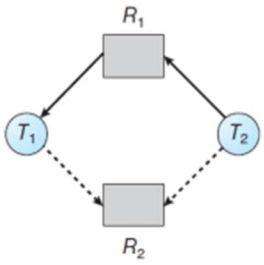


Figure 8.9 Resource-allocation graph for deadlock avoidance.

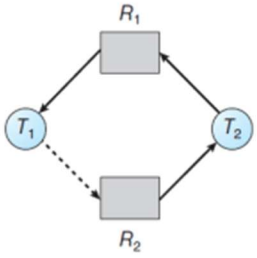


Figure 8.10 An unsafe state in a resource-allocation graph.

- **Claim Edge (Dashed Line):** Indicates possible future requests.
- If adding a request forms a cycle, deny allocation to avoid unsafe state.

3. Banker's Algorithm

Variables

- **n:** Number of processes
- **m:** Number of resource types
- **Available [m]:** Initial available resources
- **Allocation [n][m]:** Current allocation per process
- **Need [n][m]:** Remaining resource need per process
- **Work [m]:** Work vector (dynamic)
- **Finish [n]:** Process completion status

Steps

1. Initialize: Work = Available; all Finish[i] = false
2. Find an i such that:
    - Finish[i] = false
    - Need[i] <= Work
3. If found:

- Work = Work + Allocation[i]
- Finish[i] = true
- Repeat step 2

4. If all Finish[i] = true, system is in a safe state.

4. Detection and Recovery

Detection

1. **Single Instance Resources:**

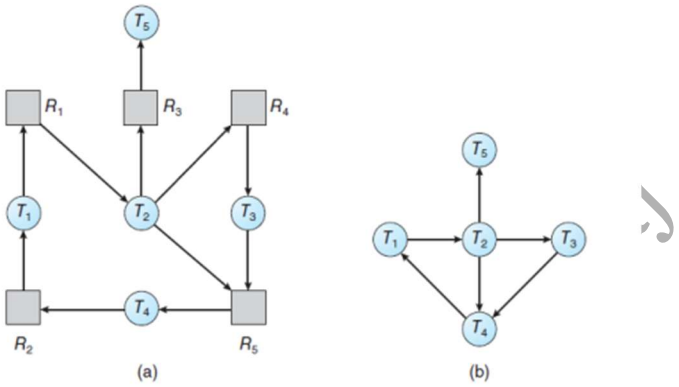


Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

- Use a **wait-for graph** (simplified resource allocation graph).
- A cycle in this graph indicates deadlock.

2. **Multiple Instances:**
   - Use a detection algorithm similar to the Banker's algorithm.
   - Replaces Need with a request array.

Recovery

- **Abort Processes:** Terminate all or some deadlocked processes until deadlock is broken.
- **Resource Preemption:** Preempt resources from some processes and assign them to others to break the deadlock cycle.

Memory Allocation and Management

Address Spaces

- **Logical Address Space:**
  An address generated by the CPU; also called a virtual address.
- **Physical Address Space:**
  The set of actual addresses used by the memory hardware.

Program Loading

- **Static Loading:**
  The entire program is loaded into a fixed memory address before execution; uses more memory.
- **Dynamic Loading:**
  Only necessary routines are loaded into memory as needed; conserves memory space.

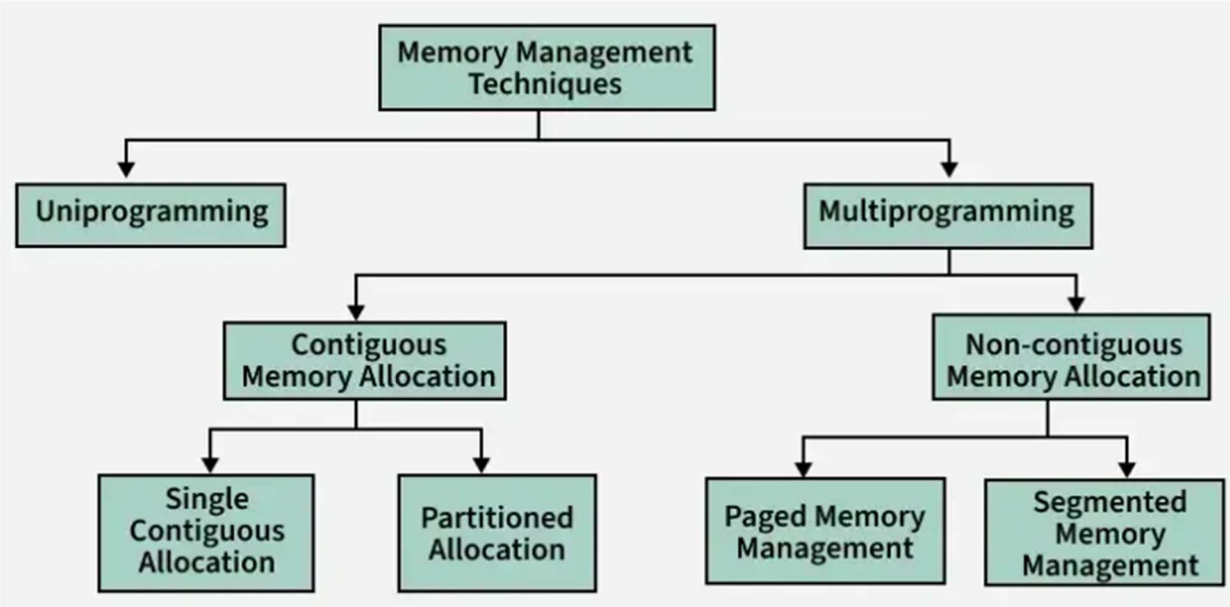
Memory Management Unit (MMU)

A hardware device that, at runtime, maps virtual (logical) addresses to physical addresses.

Address Components

- **Page Number:**
  Bits needed to represent pages in the logical address space.
- **Page Offset:**
  Bits representing the location within a page.
- **Frame Number:**
  Bits needed to indicate physical memory frames.
- **Frame Offset:**
  Bits representing the location within a frame.

Memory Allocation Techniques



Contiguous Memory Allocation

Fixed Partitioning

- Memory is divided into fixed-size partitions.
- Each partition is assigned to a process.
- Simple, but can waste memory if a process does not fit perfectly.

Dynamic Partitioning

- Memory is divided into variable-sized partitions.
- Each partition is tailored to fit the process, reducing waste.
- Requires overhead to track free memory.

Allocation Strategies

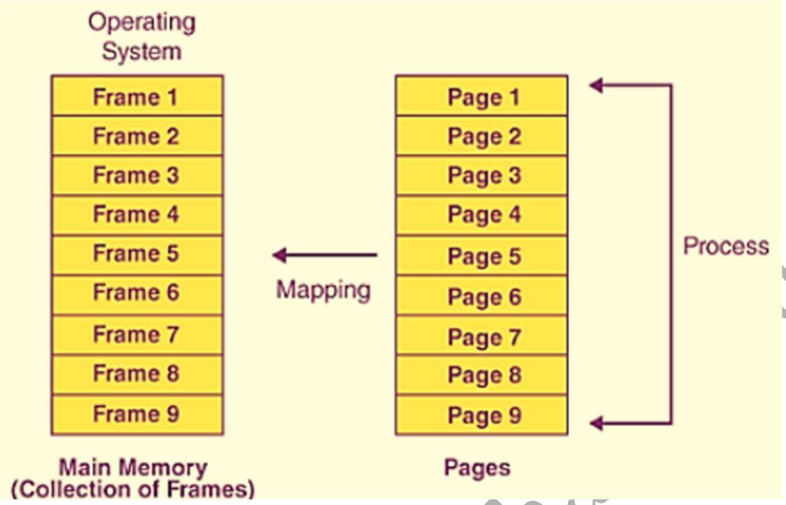
- **First-Fit:** Allocates the first available memory hole that is big enough.
- **Best-Fit:** Allocates the smallest hole that fits; requires searching the entire list.
- **Worst-Fit:** Allocates the largest available hole.

Fragmentation

- **Internal Fragmentation:** occurs when memory is allocated in fixed-sized blocks or partitions, and the process assigned to the block does not require the full amount of allocated memory. The unused space within the allocated block cannot be used by other processes, resulting in memory wastage.
- **External Fragmentation:** Total free memory is enough for a new process but not contiguous.

Non-Contiguous Allocation

Paging



- Physical memory divided into frames of fixed size.
- Process divided into equally sized pages.
- Pages are loaded into any available memory frame.
- Page size always matches frame size.

Translation Look-Aside Buffer (TLB)

- Hardware cache that stores key – value pairs of recent page-to-frame mappings.
- Checks TLB before the page table for faster translation.
- Speeds up memory access for repeated page requests.

Types of Page Tables

- **Hierarchical**

- **Hashed**
- **Inverted**

Segmentation

- Similar to paging but divides memory into segments of variable size.
- Uses a segment table; each segment can differ in length.
- Each segment mapped based on its base and limit.

Virtual Memory

- Only required pages/segments are loaded into main memory, making virtual memory appear larger than physical memory.
- Enables multiple processes to simultaneously use main memory.

Demand Paging

- Pages are loaded into memory only when needed (on demand).

Page Replacement Algorithms

When main memory is full, a page must be replaced to bring in a new one. Common strategies include:

- **FIFO (First-In First-Out)**
- **LRU (Least Recently Used)**
- **MRU (Most Recently Used)**
- **Optimal (Bélády's Algorithm)**

Note:

- ***Paging*** *suffers from* ***internal fragmentation*** *because fixed-size pages may not be entirely filled, leaving unused space within memory frames.*
- ***Segmentation*** *suffers from* ***external fragmentation*** *because variable-sized segments can leave scattered free areas in memory after allocation and deallocation, making it hard to find contiguous space for new segments.*