

# Logic

1. This program is a **Linux system call simulator** using C++. It allows the user to run commands like:

- fork()
- wait()
- exec()
- cp (copy file)
- grep (search for string in file)

Each of these is implemented using **system calls** (fork, exec, waitpid, getpid, etc.).

---

## Main Loop (main)

- Shows a menu with 6 options.
  - Based on the user's choice, it calls the respective function.
  - Runs in an infinite loop until you choose option 6 (Exit).
- 

## Logic of Each Function:

### simulate\_fork()

- Creates a new process using fork().
- If fork() returns 0 → it's the **child process**.

- If fork() returns a number > 0 → it's the **parent**, and that number is the child's PID.
- Displays getpid() and getppid().

#### **simulate\_wait()**

- Forks a child.
- Child sleeps for 2 seconds, simulating "some task", then exits.
- Parent waits using waitpid() for the child to finish.
- Shows child's exit status.

#### **simulate\_exec()**

- Forks a child.
- Child executes the Linux command ls -l using execlp().
- If exec fails, it prints an error.
- Parent waits for the child and prints the status.

#### **simulate\_cp()**

- Forks a child.
- Child runs cp src.txt dest.txt using execlp() to copy a file.
- Parent waits and checks if the copy was successful.

#### **simulate\_grep()**

- Takes input string from user.
- Forks a child.
- Child runs grep -c <string> src.txt to search for the string.
- Parent waits and prints whether string was found or not.

Option 6 simply exits the program.

---

### **Important System Calls Used:**

#### **Function Purpose**

`fork()` Creates a new process

`getpid()` Gets process ID

`getppid()` Gets parent process ID

`waitpid()` Waits for a child to finish

`execlp()` Replaces process with a new program

`exit()` Terminates the current process

## 2. // 1. FCFS Scheduling

```
void FCFS(vector<Task>& tasks, int n) { /* Already implemented */ }
```

## // 2. SJF (Non-Preemptive) Scheduling

```
void SJF(vector<Task>& tasks, int n) { /* Already implemented */ }
```

## // 3. SRTF (Preemptive SJF) Scheduling

```
void SRTF(vector<Task>& tasks, int n) { /* Already implemented */ }
```

## // 4. Round Robin (Preemptive) Scheduling

```
void roundRobin(vector<Task>& tasks, int n, int quantum) { /*  
Already implemented */ }
```

```
// 5. Priority (Non-Preemptive) Scheduling  
void priorityScheduling(vector<Task>& tasks, int n) { /* Already  
implemented */ }  
  
// 6. Priority (Preemptive) Scheduling  
void preemptivePriorityScheduling(vector<Task>& tasks, int n) { /*  
Already implemented */ }  
  
// 7. LRU Page Replacement Algorithm  
void LRU(vector<int>& pages, int capacity) {  
    list<int> cache;  
    unordered_map<int, list<int>::iterator> pageMap;  
  
    for (int page : pages) {  
        if (pageMap.find(page) != pageMap.end()) {  
            cache.erase(pageMap[page]);  
        } else if (cache.size() == capacity) {  
            int lru = cache.back();  
            cache.pop_back();  
            pageMap.erase(lru);  
        }  
        cache.push_front(page);  
        pageMap[page] = cache.begin();  
    }  
}
```

```
}

for (int p : cache) cout << p << " ";

cout << endl;

}

// 8. FCFS Page Replacement Algorithm

void FCFS_PageReplacement(vector<int>& pages, int capacity) {

queue<int> q;
unordered_set<int> s;

for (int page : pages) {

if (s.find(page) == s.end()) {

if (q.size() == capacity) {

s.erase(q.front());
q.pop();

}

q.push(page);
s.insert(page);

}

}

while (!q.empty()) {

cout << q.front() << " ";

q.pop();

}

}
```

```
cout << endl;  
}  
  
// 9. Optimal Page Replacement Algorithm  
void Optimal(vector<int>& pages, int capacity) {  
    vector<int> memory;
```

```
    for (int i = 0; i < pages.size(); ++i) {  
        int page = pages[i];  
  
        auto it = find(memory.begin(), memory.end(), page);  
        if (it != memory.end()) continue;  
  
        if (memory.size() < capacity) {  
            memory.push_back(page);  
        } else {  
            int farthest = i + 1, index = -1;  
            for (int j = 0; j < memory.size(); ++j) {  
                int k;  
                for (k = i + 1; k < pages.size(); ++k) {  
                    if (pages[k] == memory[j]) break;  
                }  
                if (k > farthest) {  
                    farthest = k;
```

```

        index = j;

    }

    if (k == pages.size()) {

        index = j;

        break;

    }

    memory[index] = page;

}

}

for (int p : memory) cout << p << " ";

cout << endl;
}

```

### **Logic :-1. FCFS (First Come First Serve)**

- Sorts by **arrival time**
- Runs each process in the order they arrive
- No preemption; one process finishes before the next starts
- Updates Gantt chart, waiting, turnaround, completion times

### **2. SJF (Shortest Job First, Non-Preemptive)**

- Sorts by **burst time**
- Runs shortest job next (like FCFS on sorted burst times)
- Doesn't consider newly arriving tasks during execution

### **3. SRTF (Shortest Remaining Time First, Preemptive)**

- At each time unit, picks task with **shortest remaining time**
- Preempts current process if a shorter one arrives
- Runs until all are completed

#### **4. Round Robin (Preemptive)**

- Each task runs for a fixed **time quantum**
- If not finished, goes to back of queue
- Processes are added to the queue as they arrive
- Implements a **circular queue** style scheduling

#### **5. Priority Scheduling (Non-Preemptive)**

- Sorts by **priority**
- Lower priority value = higher priority
- Once a process starts, it runs until completion

#### **6. Preemptive Priority Scheduling**

- At each time unit, chooses task with highest priority (lowest number)
  - Preempts current task if higher priority arrives
  - Like SRTF, but based on priority not burst
- 

#### **Helper Functions**

##### **calculateTimes()**

- Calculates waiting time = turnaround - burst
- Turnaround = completion - arrival
- Displays stats for each process
- Prints average waiting and turnaround time

## **printGanttChart()**

- Visual representation of execution timeline
- Example: | P1 | P2 | P3 |
- Below it, prints time intervals

## **displayReadyQueue()**

- Displays up to 3 processes currently in ready queue
  - Helps visualize context switching
- 

### **Highlights**

- Uses **Gantt Chart** to visualize scheduling
- Supports **preemptive and non-preemptive** methods
- Tracks and shows **ready queue**
- Stores and prints all critical timing information

### **3. inter processor Program Overview:**

- The program uses **pipes** and **file redirection** to process the contents of file1.txt.
  - It reads from the file, sorts the numbers, and then removes duplicates using two different processes (via fork()). The two processes are connected using a pipe for communication.
- 

### **Steps Involved:**

## 1. File Existence Check:

- The program first checks if the file file1.txt exists using an ifstream. If the file doesn't exist, it prints an error and terminates.

## 2. Redirecting stdin to file1.txt:

- It opens file1.txt in **read-only** mode using the open() system call.
- The dup2() function is used to redirect the **standard input (stdin)** to the file's file descriptor. This means that when the program reads from stdin, it will get data from file1.txt instead of the keyboard.

## 3. Creating a Pipe:

- A **pipe** is created using the pipe() function. This pipe consists of two file descriptors: one for reading (fd[0]) and one for writing (fd[1]). The pipe will be used for communication between the two processes.

## 4. Forking the Process:

- The fork() system call is used to create a **child process**. After the fork, both the parent and child process execute the following code:
- **Child Process (pid == 0):**
  - The child process needs to **sort** the input from the file.
  - It redirects its **standard output (stdout)** to the pipe's write end using dup2(fd[1], STDOUT\_FILENO). This means anything the child writes will go to the pipe.
  - The child process then runs the sort command using execl(), sorting the data it gets from the file.

- **Parent Process (pid > 0):**

- The parent process is responsible for reading the sorted output from the child process.
- It redirects its **standard input (stdin)** to the pipe's read end using dup2(fd[0], STDIN\_FILENO). This means the parent will read the sorted data that the child wrote to the pipe.
- The parent process runs the uniq command using execl(), which removes duplicates from the sorted data it receives.

## 5. Closing Unused Pipe Ends:

- Both the parent and child processes close the unused ends of the pipe (fd[0] and fd[1] respectively) after they have set up the redirections.

## 6. Error Handling:

- The program includes error handling for key operations:
  - If the file fails to open, it exits with an error message.
  - If pipe() or fork() fails, it prints an error message and exits.
  - If execl() fails, the program prints an error message and exits (although execl() should generally not fail unless the command is missing or invalid).

---

## Summary of IPC Mechanism:

- **Pipes** are used to pass data between the parent and child processes.

- **Redirection** is used to change the standard input and output streams.
  - The **child process** sorts the input data and writes it to the pipe.
  - The **parent process** reads from the pipe and removes duplicates.
- 

### Key Concepts:

- **Forking:** The creation of a child process.
- **Pipes:** A unidirectional communication channel between processes.
- **Redirection:** Changing where the program reads from or writes to (e.g., from files or pipes).
- **exec()**: Executes a new program, replacing the current process with a new one.

## 4. Syncornization

### Key Concepts and Logic:

#### 1. Reader-Writer Problem:

- **Reader Threads:** These threads only read the shared resource (in this case, availableSeats), so multiple readers can access the resource simultaneously.
- **Writer Threads:** These threads modify the shared resource (availableSeats). Only one writer can access the resource at a time, and no readers should be reading while a writer is modifying the data.

#### 2. Synchronization Mechanisms Used:

- **Reader-Writer Lock (pthread\_rwlock\_t):**

- The `pthread_rwlock_rdlock` and `pthread_rwlock_wrlock` functions are used for read-write locking. This ensures that multiple reader threads can read the shared resource concurrently, but a writer thread has exclusive access to the resource (i.e., no readers or other writers can access it when writing).
- The `pthread_rwlock_unlock` function releases the lock after reading or writing.

- **Mutex Lock (`pthread_mutex_t`):**

- A mutex (`writeMutex`) is used to ensure that only one writer enters the critical section at a time. This prevents race conditions when multiple writers are attempting to modify the shared resource simultaneously.

## **Breakdown of the Code:**

- **Global Variables:**

- `availableSeats`: Shared resource representing the number of seats.
- `useSynchronization`: Flag to determine whether synchronization should be used.
- `rwlock` and `writeMutex`: Synchronization mechanisms for managing access to `availableSeats`.

- **Reader Thread:**

- The reader thread attempts to read the `availableSeats`. If synchronization is enabled, it acquires a **read lock** using `pthread_rwlock_rdlock` and prints the number of available seats. If synchronization is not used, it reads and prints the seats without locking, simulating a race condition.

- **Writer Thread:**

- The writer thread tries to decrement the number of availableSeats (i.e., reserve a seat). If synchronization is enabled, it acquires both a **write lock** (via `pthread_rwlock_wrlock`) and a **mutex lock** (via `pthread_mutex_lock`) to ensure that only one writer modifies the resource at a time.
- If synchronization is not enabled, it directly updates the availableSeats, potentially leading to inconsistent state due to lack of proper synchronization.

- **User Input:**

- The user is prompted to enter the number of available seats, the number of reader and writer threads, and the choice of synchronization mode.
- Based on the synchronization choice, the program either uses synchronization (with `pthread_rwlock_t` and `pthread_mutex_t`) or simulates a race condition by not using locks.

## Possible Improvements or Notes:

### 1. Memory Management:

- You're allocating memory for each thread ID (`id = (int*)malloc(sizeof(int))`). This is correct, but you might want to ensure that all allocated memory is properly freed, especially when using dynamic memory allocation for thread IDs.

### 2. Race Condition Without Synchronization:

- In the case of no synchronization (`useSynchronization = 0`), the reader threads might print inconsistent results (because of preemption), and the writer threads might

reserve seats incorrectly (due to concurrent read-modify-write operations). This simulates the problem, but in a real-world scenario, proper synchronization would be essential.

### 3. Edge Cases:

- If there are no available seats (`availableSeats <= 0`), the writer thread should handle this gracefully (which you've already done). Ensure that when the number of available seats becomes zero, no further seat reservations are attempted.

### 4. Performance:

- The random delays (`usleep(rand() % 50000)`) simulate preemption and the effect of concurrent thread execution. This is fine for a demonstration, but in production code, you'd typically control thread timing with more structured synchronization or explicit timeouts

## 5. Banker algo

1. **Resources:** The system has multiple types of resources (e.g., A, B, C). The number of available resources is provided by the user.
2. **Processes:** Each process in the system needs resources. Each process has:
  - **Allocated Resources:** The resources it currently holds.
  - **Maximum Resources:** The maximum resources it could ever need.
3. **Need Matrix:** The difference between **Maximum Resources** and **Allocated Resources** for each process represents how many more resources it needs to complete its task.

### Basic Idea:

The program tries to determine whether there is a **safe sequence** for the processes to execute. A safe sequence is a sequence of processes that can run to completion without causing a deadlock.

**Steps:**

**1. Input Data:**

- The user provides the number of resource types (like A, B, C).
- The user provides the total available resources for each type.
- The user provides the **allocated** and **maximum** resources for each process.

**2. Available Resources Calculation:**

- The available resources are calculated by subtracting the **allocated resources** from the **total resources**.

**3. Need Matrix Calculation:**

- The **need** matrix for each process is calculated by subtracting **allocated resources** from **maximum resources** for each process. This tells how much more of each resource the process needs.

**4. Safe Sequence Calculation (Main Logic):**

- The program tries to find a **safe sequence** where the processes can complete without causing a deadlock.
- The program uses **backtracking**:
  - It checks if a process can execute by comparing its need with the available resources.
  - If a process can execute, it temporarily adds its allocated resources back to the available resources and marks the process as "visited" (indicating it's completed).
  - It then recursively checks the next processes.

- If a process cannot be executed (because it needs more resources than available), it backtracks and tries another process.

## 5. Safe Sequence Output:

- If a safe sequence is found, it prints the order of processes that can execute safely (i.e., without causing a deadlock).

### Detailed Explanation of Classes:

#### 1. Process Class:

- Represents a process.
- Each process has two arrays:
  - **allocatedResources**: Resources the process is currently holding.
  - **max**: Maximum resources the process may need.

#### 2. BankersAlgorithm Class:

- The core logic of the Banker's algorithm is implemented here.
- **takelInput()**: Collects data for the number of processes, resources, allocated resources, and maximum resources.
- **findSafeSequences()**: This is a **recursive function** that tries to find the safe sequence of processes.
- **algo()**: This is the function that calculates the available resources, the need matrix, and calls the `findSafeSequences()` function to find the safe sequence.

### Process:

#### 1. User Input:

- The user provides the total number of resources (tr) and the details of each process' allocated and maximum resources.

#### 2. Calculate Available Resources:

- The available resources are computed by subtracting the allocated resources from the total resources.

#### 3. Calculate Need Matrix:

- The need[i][j] matrix is filled by subtracting the allocated resources of process i from the maximum required resources. This tells how many more resources each process needs for completion.

#### **4. Safe Sequence Search:**

- The findSafeSequences() function recursively checks if each process can execute.
- If a process can execute (its needed resources are less than or equal to the available resources), it is added to the safe sequence, and the available resources are updated by adding the process's allocated resources.
- The process is marked as completed (visited), and the function continues to find more processes that can execute.
- If a process cannot execute, the function backtracks and tries a different combination.

#### **5. Output:**

- Once all processes have been successfully executed (i.e., a safe sequence is found), the program prints the sequence of processes that can safely execute without causing a deadlock.

#### **Example Walkthrough:**

- Suppose there are 3 types of resources (A, B, C), and 3 processes.
- After the user inputs the data (e.g., total resources, allocated, and max resources for each process), the program calculates how many resources are available and how much more each process needs.
- The program will try to find a sequence in which each process can execute safely by ensuring that there are enough resources available for each process to complete its task.

#### **In Summary:**

The Banker's Algorithm ensures that the system remains in a safe state by calculating if there's an order of process execution where all processes can eventually finish without leading to deadlock. The program implements this by backtracking and trying different combinations of process execution orders. If a safe sequence is found, it is displayed; otherwise, the system is considered unsafe.

Let me know if you need further clarification!

**You said:**

I love the example along with it

**ChatGPT said:**

Sure! Let's go through an example of the **Banker's Algorithm** with all the steps.

**Example:**

Suppose we have 3 resource types (A, B, C), and 3 processes.

Here is the data for this example:

**Total Resources:**

- A = 10
- B = 5
- C = 7

**Process Information:**

Process	Allocated Resources (A, B, C)	Maximum Resources (A, B, C)
P1	3, 2, 2	7, 5, 3
P2	2, 3, 3	3, 3, 3
P3	3, 1, 2	9, 3, 2

**Step-by-Step Execution:**

1. **Total Resources:**

- A = 10
- B = 5
- C = 7

## 2. Allocated Resources:

- o Process P1 has allocated (A=3, B=2, C=2)
- o Process P2 has allocated (A=2, B=3, C=3)
- o Process P3 has allocated (A=3, B=1, C=2)

## 3. Available Resources: To calculate available resources:

- o Available = Total Resources - Sum of Allocated Resources
- o Available A =  $10 - (3 + 2 + 3) = 10 - 8 = 2$
- o Available B =  $5 - (2 + 3 + 1) = 5 - 6 = -1$  (Not enough resources for B)
- o Available C =  $7 - (2 + 3 + 2) = 7 - 7 = 0$

But we can immediately see that the system is not in a safe state because the available B is negative. This means that we can't run any process without causing a deadlock.

## 4. Need Matrix: The **Need** matrix represents how many resources each process still needs to finish. It's calculated by:

- o Need = Maximum Resources - Allocated Resources

Process	Need (A, B, C)
P1	$7-3=4, 5-2=3, 3-2=1 \rightarrow (4, 3, 1)$
P2	$3-2=1, 3-3=0, 3-3=0 \rightarrow (1, 0, 0)$
P3	$9-3=6, 3-1=2, 2-2=0 \rightarrow (6, 2, 0)$

Now, the **Need** matrix looks like this:

Process	Need (A, B, C)
P1	(4, 3, 1)
P2	(1, 0, 0)
P3	(6, 2, 0)

## 5. Safe Sequence Calculation:

We will try to find a safe sequence, which is a sequence where all processes can complete without causing deadlock.

- **Initial Available Resources:**  $(2, -1, 0) \rightarrow$  Not enough resources to even start the first process, so the system is in an unsafe state.

**Final Result:**

- The system is **unsafe** because it's not possible to find a safe sequence due to insufficient resources.
- 

**Now, Let's consider a safe example:**

Let's change the **Total Resources** a bit to ensure that the system is safe.

**New Total Resources:**

- $A = 10$
- $B = 5$
- $C = 7$

**Process Information:**

Process	Allocated Resources (A, B, C)	Maximum Resources (A, B, C)
P1	3, 2, 2	7, 5, 3
P2	2, 3, 3	3, 3, 3
P3	3, 1, 2	9, 3, 2

**Step-by-Step Execution:**

**1. Total Resources:**

- $A = 10$
- $B = 5$
- $C = 7$

**2. Allocated Resources:**

- Process P1:  $(A=3, B=2, C=2)$
- Process P2:  $(A=2, B=3, C=3)$
- Process P3:  $(A=3, B=1, C=2)$

**3. Available Resources:**

- Available  $A = 10 - (3 + 2 + 3) = 10 - 8 = 2$

- Available B =  $5 - (2 + 3 + 1) = 5 - 6 = -1$  (Not enough resources for B)
- Available C =  $7 - (2 + 3 + 2) = 7 - 7 = 0$

**If we change the total resources to ensure safety:**

**New Total Resources:**

**Total Resources:**

- A = 12
- B = 7
- C = 10

**Step-by-Step Execution:**

**1. Total Resources:**

- A = 12
- B = 7
- C = 10

**2. Allocated Resources:**

- Process P1: (A=3, B=2, C=2)
- Process P2: (A=2, B=3, C=3)
- Process P3: (A=3, B=1, C=2)

**3. Available Resources:**

- Available A =  $12 - (3 + 2 + 3) = 12 - 8 = 4$
- Available B =  $7 - (2 + 3 + 1) = 7 - 6 = 1$
- Available C =  $10 - (2 + 3 + 2) = 10 - 7 = 3$

So, the **Available Resources** are now (4, 1, 3).

**4. Need Matrix:**

- P1's Need: (7-3, 5-2, 3-2) = (4, 3, 1)
- P2's Need: (3-2, 3-3, 3-3) = (1, 0, 0)
- P3's Need: (9-3, 3-1, 2-2) = (6, 2, 0)

**5. Finding Safe Sequence:**

- **Step 1:** P2's need (1, 0, 0) is less than or equal to available resources (4, 1, 3), so P2 can finish.
- After P2 finishes, it releases resources (2, 3, 3), and the available resources become:
  - Available =  $(4 + 2, 1 + 3, 3 + 3) = (6, 4, 6)$

- **Step 2:** Now P1's need (4, 3, 1) is less than or equal to the available resources (6, 4, 6), so P1 can finish.
- After P1 finishes, it releases resources (3, 2, 2), and the available resources become:
  - Available =  $(6 + 3, 4 + 2, 6 + 2) = (9, 6, 8)$
- **Step 3:** Finally, P3's need (6, 2, 0) is less than or equal to the available resources (9, 6, 8), so P3 can finish.
- After P3 finishes, it releases resources (3, 1, 2), and the available resources become:
  - Available =  $(9 + 3, 6 + 1, 8 + 2) = (12, 7, 10)$

Now, all processes have finished, and the sequence of execution is:

**Safe Sequence:** P2 → P1 → P3

## Numerical-:

### 1.FCFS

**Numerical Example:**

Let's say there are 3 processes with the following burst times:

Process	Burst Time
P1	6
P2	8
P3	7

**Step-by-Step Calculation:**

#### 1. Waiting Time:

- P1: 0 (since it is the first process)
- P2: 6 (Waiting time of P1)
- P3:  $6 + 8 = 14$  (Waiting time of P1 + P2)

#### 2. Turnaround Time:

- P1: 6 (Burst time of P1)
- P2:  $8 + 6 = 14$  (Burst time of P2 + Waiting time of P2)

- P3:  $7 + 14 = 21$  (Burst time of P3 + Waiting time of P3)
- Average Waiting Time =  $(0 + 6 + 14) / 3 = 6.67$
- Average Turnaround Time =  $(6 + 14 + 21) / 3 = 13.67$

### 3. (Shortest Job First) - Non-Preemptive

Numerical Example:

Let's use the same burst times as before:

Process	Burst Time
P1	6
P2	8
P3	7

Step-by-Step Calculation:

- After sorting by burst time:

P1 (6), P3 (7), P2 (8)

#### 1. Waiting Time:

- P1: 0 (since it is the first process)
- P3: 6 (Waiting time of P1)
- P2:  $6 + 7 = 13$  (Waiting time of P1 + P3)

#### 2. Turnaround Time:

- P1: 6 (Burst time of P1)
- P3:  $7 + 6 = 13$  (Burst time of P3 + Waiting time of P3)
- P2:  $8 + 13 = 21$  (Burst time of P2 + Waiting time of P2)

Average Waiting Time =  $(0 + 6 + 13) / 3 = 6.33$

Average Turnaround Time =  $(6 + 13 + 21) / 3 = 13.33$

### 3. Round Robin (Preemptive)

Let's assume 3 processes with burst times as follows:

Process	Burst Time
P1	5
P2	9
P3	6

Step-by-Step Calculation (assuming quantum = 4):

1. P1 runs for 4 units, remaining time =  $5 - 4 = 1$ .
2. P2 runs for 4 units, remaining time =  $9 - 4 = 5$ .
3. P3 runs for 4 units, remaining time =  $6 - 4 = 2$ .
4. P1 runs for 1 unit, remaining time = 0 (P1 completes).
5. P2 runs for 4 units, remaining time =  $5 - 4 = 1$ .
6. P3 runs for 2 units, remaining time = 0 (P3 completes).
7. P2 runs for 1 unit, remaining time = 0 (P2 completes).

The completion times for each process will be:

- P1: 9
- P3: 12
- P2: 15

## 7.FCFS with Priority (Non-Preemptive)

**Explanation:**

- This is a modified FCFS, where each process has a priority. The CPU executes the process with the highest priority first, and within the same priority, it executes based on FCFS.

**Numerical Example:**

Process	Arrival Time	Burst Time	Priority
P1	0	5	3
P2	1	3	2
P3	2	8	1

1. Order of Execution based on Priority (highest first): P3 (priority 1), P2 (priority 2), P1 (priority 3)

### 2. Waiting Time Calculation:

- P3: Waiting time = 0.
- P2: P2 waits for P3 to complete, so waiting time = 8.
- P1: P1 waits for both P3 and P2 to complete, so waiting time =  $8 + 3 = 11$ .

### 3. Turnaround Time Calculation:

- P3: Turnaround time = burst time + waiting time =  $8 + 0 = 8$ .
- P2: Turnaround time = burst time + waiting time =  $3 + 8 = 11$ .
- P1: Turnaround time = burst time + waiting time =  $5 + 11 = 16$ .

$$\text{Average Waiting Time} = (0 + 8 + 11) / 3 = 6.33$$

$$\text{Average Turnaround Time} = (8 + 11 + 16) / 3 = 11.67$$

---

## 5. SJF (Preemptive)

**Explanation:**

- Preemptive SJF, also known as Shortest Remaining Time First (SRTF), preempts the currently running process if a new process arrives with a shorter burst time.

**Numerical Example:**

Process	Arrival Time	Burst Time
P1	0	6
P2	1	8
P3	2	7

**Step-by-step:**

1. At time 0: P1 starts executing.
2. At time 1: P2 arrives, but P1 is still running (remaining time 5).
3. At time 2: P3 arrives. Since P3 has a shorter burst (7), it preempts P1.
4. P3 runs till time 5. P2 now runs from time 5 till 12.
5. P1 resumes at 12, finishes at 18.

**Completion Times:**

- P1 completes at time 18
- P2 completes at time 12
- P3 completes at time 5

---

## 6. Priority Scheduling (Preemptive)

**Explanation:**

- In preemptive priority scheduling, the process with the highest priority gets the CPU. If a process with a higher priority arrives while a lower priority process is executing, it preempts the lower priority process.

**Numerical Example:**

Process	Arrival Time	Burst Time	Priority
P1	0	5	1
P2	1	3	3

Process	Arrival Time	Burst Time	Priority
P3	2	8	2

**Step-by-step:**

1. P1 starts executing at time 0.
  2. At time 1: P2 arrives with a higher priority, preempting P1.
  3. P2 runs till time 4.
  4. P1 resumes at time 4 and completes at time 9.
  5. P3 executes next, completing at time 17.
- 

## 9. Optimal Paging Replacement Algorithm

**Explanation:**

- The **Optimal** paging replacement algorithm replaces the page that will not be used for the longest period of time in the future. This is the most efficient but requires knowing the future references.

**Numerical Example:**

Assume we have 3 frames in memory and the reference string is:

**Reference String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3**

1. **Page Reference 7:** Page 7 is not in memory. It is loaded into the first available frame.

**Frames: [7, -, -]**

**Page Faults: 1**

2. **Page Reference 0:** Page 0 is not in memory. It is loaded into the next available frame.

**Frames: [7, 0, -]**

**Page Faults: 2**

3. **Page Reference 1:** Page 1 is not in memory. It is loaded into the third available frame.

**Frames: [7, 0, 1]**

**Page Faults: 3**

4. **Page Reference 2:** Page 2 is not in memory. We check which page will be used farthest in the future:

- Page 7 will be used at position 4,
- Page 0 will be used at position 5,
- Page 1 will be used at position 6.

Therefore, **Page 7** is replaced.

**Frames:** [2, 0, 1]

**Page Faults:** 4

5. **Page Reference 0:** Page 0 is already in memory.

**Frames:** [2, 0, 1]

**Page Faults:** 4 (no change)

6. **Page Reference 3:** Page 3 is not in memory. We check the future references:

- Page 2 will be used at position 6,
- Page 0 will be used at position 5,
- Page 1 will not be used.

Therefore, **Page 1** is replaced.

**Frames:** [2, 0, 3]

**Page Faults:** 5

7. **Page Reference 0:** Page 0 is already in memory.

**Frames:** [2, 0, 3]

**Page Faults:** 5 (no change)

8. **Page Reference 4:** Page 4 is not in memory. We check the future references:

- Page 2 will be used at position 6,
- Page 0 will be used at position 5,
- Page 3 will not be used.

Therefore, **Page 3** is replaced.

**Frames:** [2, 0, 4]

**Page Faults:** 6

9. **Page Reference 2:** Page 2 is already in memory.

**Frames:** [2, 0, 4]

**Page Faults:** 6 (no change)

10. **Page Reference 3:** Page 3 is not in memory. We check the future references:

- o Page 2 will be used at position 6,
- o Page 0 will be used at position 5,
- o Page 4 will not be used.

Therefore, **Page 4** is replaced.

**Frames:** [2, 0, 3]

**Page Faults:** 7

**Total Page Faults:** 7

---

## 10. Least Recently Used (LRU) Paging Replacement Algorithm

**Explanation:**

- **LRU** replaces the page that has not been used for the longest time. It keeps track of the most recent accesses.

**Numerical Example:**

Reference String	Frames
7, 0, 1, 2, 0, 3, 0, 4, 2, 3	3

1. **Page Reference 7:** Page 7 is not in memory. It is loaded into the first available frame.

**Frames:** [7, -, -]

**Page Faults:** 1

2. **Page Reference 0:** Page 0 is not in memory. It is loaded into the next available frame.

**Frames:** [7, 0, -]

**Page Faults:** 2

3. **Page Reference 1:** Page 1 is not in memory. It is loaded into the third available frame.

**Frames:** [7, 0, 1]

**Page Faults:** 3

4. **Page Reference 2:** Page 2 is not in memory. The least recently used page is **7**. So, replace page 7.

**Frames:** [2, 0, 1]

**Page Faults:** 4

5. **Page Reference 0:** Page 0 is already in memory.

**Frames:** [2, 0, 1]

**Page Faults:** 4 (no change)

6. **Page Reference 3:** Page 3 is not in memory. The least recently used page is **1**. So, replace page 1.

**Frames:** [2, 0, 3]

**Page Faults:** 5

7. **Page Reference 0:** Page 0 is already in memory.

**Frames:** [2, 0, 3]

**Page Faults:** 5 (no change)

8. **Page Reference 4:** Page 4 is not in memory. The least recently used page is **2**. So, replace page 2.

**Frames:** [4, 0, 3]

**Page Faults:** 6

9. **Page Reference 2:** Page 2 is not in memory. The least recently used page is **3**. So, replace page 3.

**Frames:** [4, 0, 2]

**Page Faults:** 7

10. **Page Reference 3:** Page 3 is not in memory. The least recently used page is **0**. So, replace page 0.

**Frames:** [4, 3, 2]

**Page Faults:** 8

**Total Page Faults:** 8

## 10. best fit, next fit

**Key Structures:**

1. **Block:** Represents a memory block with:

- **size:** The size of the block.
- **allocated:** A flag that indicates whether the block is allocated or not.

2. **Process:** Represents a process that needs memory:

- size: The size of the process.
- allocatedBlock: The index of the block allocated to this process.
- fragmentation: The unused space in the block after allocation.

### **Memory Allocation Strategies:**

The program includes the following memory allocation strategies:

#### **1. First Fit Allocation**

- **Logic:** The program allocates the first block that is large enough to accommodate the process.
- **Steps:**
  1. Start from the first block.
  2. If the block is free and its size is greater than or equal to the process size, allocate it to the process.
  3. Mark the block as allocated and calculate the fragmentation (block size - process size).
  4. Move to the next process.

#### **Example:**

- Blocks: [10, 20, 30]
- Processes: [5, 10, 25]

#### **First Fit Steps:**

1. Process 1 (size 5) → Allocated to Block 1 (size 10).  
Fragmentation =  $10 - 5 = 5$ .
2. Process 2 (size 10) → Allocated to Block 2 (size 20).  
Fragmentation =  $20 - 10 = 10$ .
3. Process 3 (size 25) → Allocated to Block 3 (size 30).  
Fragmentation =  $30 - 25 = 5$ .

#### **Allocation Result:**

- Process 1: Block 1, Fragmentation 5
- Process 2: Block 2, Fragmentation 10
- Process 3: Block 3, Fragmentation 5

## **2. Best Fit Allocation**

- **Logic:** The program allocates the block that has the smallest available space that can fit the process (minimizing wasted space).
- **Steps:**
  1. For each process, search through all the blocks.
  2. If the block is free and large enough, calculate the fragmentation (block size - process size).
  3. Choose the block with the minimum fragmentation and allocate the process to it.

### **Example:**

- Blocks: [10, 20, 30]
- Processes: [5, 10, 25]

### **Best Fit Steps:**

1. Process 1 (size 5) → Block 1 (size 10) → Fragmentation =  $10 - 5 = 5$ .
2. Process 2 (size 10) → Block 2 (size 20) → Fragmentation =  $20 - 10 = 10$ .
3. Process 3 (size 25) → Block 3 (size 30) → Fragmentation =  $30 - 25 = 5$ .

### **Allocation Result:**

- Process 1: Block 1, Fragmentation 5
- Process 2: Block 2, Fragmentation 10
- Process 3: Block 3, Fragmentation 5

## **3. Next Fit Allocation**

- **Logic:** The program allocates the block starting from the last allocated block and proceeds to the next block. Once it reaches the end, it starts from the first block again.
- **Steps:**
  1. Start from the last allocated block and search through the blocks in a circular fashion.
  2. If the block is free and large enough, allocate it to the process.

3. Mark the block as allocated and calculate the fragmentation.

**Example:**

- Blocks: [10, 20, 30]
- Processes: [5, 10, 25]

**Next Fit Steps:**

1. Process 1 (size 5) → Block 1 (size 10). Fragmentation =  $10 - 5 = 5$ .
2. Process 2 (size 10) → Block 2 (size 20). Fragmentation =  $20 - 10 = 10$ .
3. Process 3 (size 25) → Block 3 (size 30). Fragmentation =  $30 - 25 = 5$ .

**Allocation Result:**

- Process 1: Block 1, Fragmentation 5
- Process 2: Block 2, Fragmentation 10
- Process 3: Block 3, Fragmentation 5

**4. Worst Fit Allocation**

- **Logic:** The program allocates the block that has the largest available space (maximizing unused space).
- **Steps:**
  1. For each process, search through all the blocks.
  2. If the block is free and large enough, calculate the fragmentation (block size - process size).
  3. Choose the block with the maximum fragmentation and allocate the process to it.

**Example:**

- Blocks: [10, 20, 30]
- Processes: [5, 10, 25]

**Worst Fit Steps:**

1. Process 1 (size 5) → Block 3 (size 30). Fragmentation =  $30 - 5 = 25$ .
2. Process 2 (size 10) → Block 2 (size 20). Fragmentation =  $20 - 10 = 10$ .

3. Process 3 (size 25) → Block 3 (size 25). Fragmentation = 25 - 25 = 0.

**Allocation Result:**

- Process 1: Block 3, Fragmentation 25
- Process 2: Block 2, Fragmentation 10
- Process 3: Block 3, Fragmentation 0

**Analysis of Algorithms:**

After applying all allocation strategies, the program calculates the total fragmentation for each method and identifies the one with the least fragmentation. The result of each allocation strategy is printed, showing:

- Which block each process was allocated to.
- The fragmentation (unused space) left in each allocated block.

**Final Analysis Example:**

After applying all strategies to the same set of blocks and processes, the program will show the total fragmentation for each allocation strategy. The strategy with the minimum total fragmentation is identified as the best allocation strategy.

11.cscan

## **1. FIFO (First-In-First-Out)**

**Approach:** Process the requests in the order they arrive.

- **Steps:**
  - Start from the initial head position (50).
  - Move to the first request (55), then to the next, and so on, until all requests are processed.

**Execution:**

- Start at position **50**.

- Move to **55**: Seek =  $|55 - 50| = 5$ .
- Move to **60**: Seek =  $|60 - 55| = 5$ .
- Move to **98**: Seek =  $|98 - 60| = 38$ .
- Move to **10**: Seek =  $|10 - 98| = 88$ .
- Move to **35**: Seek =  $|35 - 10| = 25$ .

### **Total Seek Operations (FIFO):**

$$5 + 5 + 38 + 88 + 25 = 161$$

### **Order of head movement:**

$$50 \rightarrow 55 \rightarrow 60 \rightarrow 98 \rightarrow 10 \rightarrow 35$$

---

## **2. SSTF (Shortest Seek Time First)**

**Approach:** Always move to the request closest to the current head position.

### **Execution:**

- Start at position **50**.
- The closest request to **50** is **55**: Seek =  $|55 - 50| = 5$ .
- The closest request to **55** is **60**: Seek =  $|60 - 55| = 5$ .
- The closest request to **60** is **35**: Seek =  $|35 - 60| = 25$ .
- The closest request to **35** is **10**: Seek =  $|10 - 35| = 25$ .
- The last remaining request is **98**: Seek =  $|98 - 10| = 88$ .

### **Total Seek Operations (SSTF):**

$$5 + 5 + 25 + 25 + 88 = 148$$

### **Order of head movement:**

$$50 \rightarrow 55 \rightarrow 60 \rightarrow 35 \rightarrow 10 \rightarrow 98$$

---

### **3. SCAN (Elevator Algorithm)**

**Approach:** Move the disk arm towards one end, process all requests in that direction, and then reverse direction after reaching the end.

#### **Execution:**

- Start at position **50**.
- The disk size is 200, so we assume the arm moves towards the end first (direction towards 199).
- Sort the requests: [10, 35, 55, 60, 98], and include the head (50) and the disk boundaries (0 and 199).
- The new sorted requests: [0, 10, 35, 50, 55, 60, 98, 199].

#### **Move towards 199:**

- Start at **50**, move to **55**: Seek =  $|55 - 50| = 5$ .
- Move to **60**: Seek =  $|60 - 55| = 5$ .
- Move to **98**: Seek =  $|98 - 60| = 38$ .
- Move to **199**: Seek =  $|199 - 98| = 101$ .

#### **Reverse direction and move towards 0:**

- Move to **0**: Seek =  $|0 - 199| = 199$ .
- Move to **10**: Seek =  $|10 - 0| = 10$ .
- Move to **35**: Seek =  $|35 - 10| = 25$ .

#### **Total Seek Operations (SCAN):**

$$5 + 5 + 38 + 101 + 199 + 10 + 25 = 383$$

#### **Order of head movement:**

50 → 55 → 60 → 98 → 199 → 0 → 10 → 35

---

### **4. C-SCAN (Circular SCAN)**

**Approach:** Similar to SCAN, but when the head reaches one end, it jumps to the opposite end and continues moving in the same direction.

### **Execution:**

- Start at position **50**.
- The disk size is 200, so the arm moves towards the end first (direction towards 199).
- Sort the requests: [10, 35, 55, 60, 98], and include the head (50) and the disk boundaries (0 and 199).
- The new sorted requests: [0, 10, 35, 50, 55, 60, 98, 199].

### **Move towards 199:**

- Start at **50**, move to **55**: Seek =  $|55 - 50| = 5$ .
- Move to **60**: Seek =  $|60 - 55| = 5$ .
- Move to **98**: Seek =  $|98 - 60| = 38$ .
- Move to **199**: Seek =  $|199 - 98| = 101$ .

### **Jump to 0 and continue:**

- Move to **0**: Seek =  $|0 - 199| = 199$ .
- Move to **10**: Seek =  $|10 - 0| = 10$ .
- Move to **35**: Seek =  $|35 - 10| = 25$ .
- Then move to **50** (back to initial position): Seek =  $|50 - 35| = 15$ .

### **Total Seek Operations (C-SCAN):**

$$5 + 5 + 38 + 101 + 199 + 10 + 25 + 15 = 398$$

### **Order of head movement:**

50 → 55 → 60 → 98 → 199 → 0 → 10 → 35 → 50

---

## **Summary of Total Seek Operations:**

1. **FIFO**: 161 seek operations.
2. **SSTF**: 148 seek operations.
3. **SCAN**: 383 seek operations.
4. **C-SCAN**: 398 seek operations.

**Conclusion:** In this example, **SSTF** minimizes the total seek time, followed by **FIFO**. **SCAN** and **C-SCAN** have higher seek counts because of the way they handle movement to the end of the disk and back. However, SCAN and C-SCAN have more predictable and systematic behavior, whereas SSTF can lead to starvation in certain cases (requests far from the head could be delayed indefinitely).