**Detailed Report on OS Simulator Project**

**Table of Contents**

1. [Introduction](#introduction)
2. [Features Implemented](#features-implemented)
   * [Process Management](#process-management)
   * [Scheduling Algorithms](#scheduling-algorithms)
   * [Memory Management](#memory-management)
   * [Segmentation Management](#segmentation-management)
   * [File System](#file-system)
3. [Code Structure](#code-structure)
4. [How It Works](#how-it-works)
5. [Example Simulation](#example-simulation)
6. [Future Work](#future-work)

**Introduction**

The OS Simulator Project simulates key functionalities of an operating system (OS), including process scheduling, memory management, segmentation, and file system operations. The project is developed using C++ to emulate OS concepts like process lifecycle management, page-based memory allocation, and different scheduling strategies such as FCFS, SJF, Round Robin, and Priority scheduling.

The project is designed to offer a simplified, educational look at OS principles while keeping the complexity manageable for development and understanding.

**Features Implemented**

**1. Process Management**

The project implements **Process Management** through a ProcessControlBlock (PCB) structure. Each process is created with the following attributes:

* Process ID (PID)
* Process name
* Priority level
* Arrival time
* Burst time
* Remaining time
* Process state (READY, RUNNING, WAITING, TERMINATED)

The system supports multiple processes and allows basic operations like process creation, state changes, and termination.

**Key Functions:**

* createProcess(): Initializes a process.
* terminateProcess(): Terminates a process and marks it as TERMINATED.
* displayProcessInfo(): Displays process details like PID, name, priority, and state.

**2. Scheduling Algorithms**

The simulator includes five classic CPU scheduling algorithms:

**a. First Come, First Serve (FCFS)**

Processes are executed in the order they arrive, regardless of their priority or burst time.

**Key Logic:**

* The processes are sorted by arrival time.
* Each process runs from its arrival until completion without preemption.

**b. Shortest Job First (SJF)**

Processes with the shortest burst time are executed first, ensuring minimal waiting time.

**Key Logic:**

* Processes are sorted based on their burst time.
* Once a process starts running, it completes before the next process.

**c. Shortest Remaining Time First (SRTF)**

SRTF is a preemptive version of SJF where the process with the shortest remaining time runs first. If a new process arrives with a shorter remaining time, it preempts the current process.

**Key Logic:**

* A priority queue is used to manage the processes.
* The process with the shortest remaining time runs until preempted or terminated.

**d. Round Robin (RR)**

This algorithm provides time-sharing by allowing each process to run for a fixed time quantum. If a process isn’t finished within its quantum, it is moved to the end of the queue.

**Key Logic:**

* Each process runs for a specific time quantum.
* If incomplete, it goes to the back of the queue.

**e. Priority Scheduling**

Processes with higher priority run first. If two processes have the same priority, they are executed in order of arrival.

**Key Logic:**

* Processes are sorted by priority.
* Each process runs until it completes.

**3. Memory Management**

The memory management system in this simulator uses **paging** to allocate memory to processes. The MemoryManager class manages page allocation, deallocation, and displays memory status.

**Key Features:**

* Allocates memory in pages (each process gets a set number of frames).
* Implements a **FIFO page replacement algorithm**.
* Can deallocate pages as needed.

**Key Functions:**

* allocatePage(): Allocates a new page.
* deallocatePage(): Frees a page.
* displayMemoryStatus(): Shows the current state of memory.

**4. Segmentation Management**

Segmentation is implemented as a form of memory management in which memory is divided into segments of varying sizes.

**Key Features:**

* Allows creation of named memory segments (e.g., "Code", "Data").
* Supports allocation and deallocation of segments.
* Displays the state of all memory segments.

**Key Functions:**

* allocateSegment(): Allocates a segment of memory.
* deallocateSegment(): Frees a segment.
* displaySegments(): Displays current memory segments.

**5. File System**

The simulated **File System** supports basic file operations such as file creation, deletion, reading, and writing. The file system is managed using a hash map where file names are associated with their data.

**Key Features:**

* Creates files with specified names.
* Writes data to files.
* Reads data from files.
* Deletes files and displays current files in the system.

**Key Functions:**

* createFile(): Creates a new file.
* writeFile(): Writes data to an existing file.
* readFile(): Reads the content of a file.
* deleteFile(): Deletes a file from the system.

**Code Structure**

The project follows a modular structure where each component (process, scheduler, memory, segmentation, file system) is implemented in its own header and source file. This allows for easy extensibility and maintenance.

**Directory Structure**

makefile

Copy code

OS\_Simulator/

├── src/

│ ├── main.cpp # Main entry point of the simulator

│ ├── process/

│ │ ├── process.h # Process management header

│ │ ├── process.cpp # Process management implementation

│ ├── scheduler/

│ │ ├── scheduler.h # Scheduling algorithms header

│ │ ├── scheduler.cpp # Scheduling algorithms implementation

│ ├── memory/

│ │ ├── memory\_manager.h # Memory management header

│ │ ├── memory\_manager.cpp # Memory management implementation

│ ├── segmentation/

│ │ ├── segmentation.h # Segmentation management header

│ │ ├── segmentation.cpp # Segmentation management implementation

│ ├── file\_system/

│ │ ├── file\_system.h # File system header

│ │ ├── file\_system.cpp # File system implementation

├── tests/ # Unit tests

├── include/ # Shared headers

├── docs/ # Documentation

├── Makefile # Build script

└── README.md # Project overview and instructions

**How It Works**

1. **Process Creation**: The simulator initializes processes with specific properties such as PID, arrival time, burst time, etc.
2. **Scheduling**: The processes are then passed through various scheduling algorithms (e.g., FCFS, SJF, etc.).
3. **Memory Management**: Memory is allocated in frames, and processes occupy available memory. If memory is full, page replacement algorithms are triggered.
4. **Segmentation**: The system simulates memory segmentation, dividing memory into logical segments (Code, Data, etc.).
5. **File System**: The file system supports creation, deletion, reading, and writing operations for text-based files.

**Example Simulation**

**1. Process Scheduling**

The simulation initializes five processes and applies different scheduling algorithms:

* FCFS: Runs processes in the order of arrival.
* SJF: Processes with the shortest burst time run first.
* SRTF: Pre-empts longer processes in favour of shorter ones.
* Round Robin: Executes processes in time slices (quantum).
* Priority Scheduling: Runs high-priority processes first.

**2. Memory Management**

Memory is managed using paging with FIFO/LRU page replacement. For example:

* 5 frames are available.
* Pages are allocated to processes.
* FIFO replaces older pages when memory is full.
* LRU replace least recently when memory is full.

**3. Segmentation Management**

Segments of memory like Code and Data are allocated and displayed to showcase segmentation.

**4. File System**

The file system creates, reads, writes, and deletes files, showing the interaction with stored data.

**Future Work**

Several improvements and extensions can be made to enhance the OS Simulator:

**1. Multi-threading Support**

* Currently, processes are simulated without actual multi-threading. Future work could include multi-threading and synchronization primitives like semaphores and mutexes.

**2. Memory Management Enhancements**

* Implement additional page replacement algorithms (e.g., Optimal).
* Add support for demand paging and virtual memory simulation.

**3. Process Synchronization**

* Implement inter-process communication (IPC) and process synchronization mechanisms such as semaphores, monitors, and condition variables.

**4. I/O Operations**

* Introduce I/O-bound processes and simulate I/O device interactions, as well as handling I/O interrupts.

**5. File System Improvements**

* Implement directory structures and hierarchical file systems.
* Add support for disk scheduling algorithms like FCFS, SSTF, and SCAN.

**6. Graphical Interface**

* Create a graphical user interface (GUI) for better interaction, visualization of scheduling, memory, and file operations.

**7. Performance Evaluation**

* Add performance metrics such as CPU utilization, average waiting time, and turnaround time for processes.