**MINI OS PROJECT PROPOSAL**

**Step 1: Set Up Your Workspace**

1. Open **Ubuntu** in VMware and launch the **Terminal** (Ctrl+Alt+T).
2. Create a directory for your project:
3. mkdir MINI\_OS
4. cd MINI\_OS

This is where all your project files will go.

**Step 2: Install Required Tools**

1. **Update Ubuntu**:
2. sudo apt update && sudo apt upgrade
3. **Install C Compiler (GCC)**:
4. sudo apt install build-essential
5. **Install Text Editor (Nano)**:
6. sudo apt install nano

**Step 3: Create and Test the Basic Structure**

1. Create a file for your project’s main entry point:
2. nano main.c
3. Add this basic code:
4. #include <stdio.h>
5. int main() {
6. printf("Welcome to OPTIMA OS!\n");
7. return 0;
8. }
9. Save and exit (Ctrl+O, Enter, Ctrl+X).
10. Compile and run:
11. gcc main.c -o optima
12. ./optima

You should see: Welcome to OPTIMA OS!  
This confirms that your setup is working.

**Step 4: Implement Features**

We'll implement each feature in a modular way by creating separate files and combining them later.

**Feature 1: Process Management (First Come, First Serve - FCFS)**

1. Create a new file:
2. nano process\_management.c
3. Add FCFS scheduling code:
4. #include <stdio.h>
5. void fcfs(int processes[], int n, int burstTime[]) {
6. int waitTime[n], turnAroundTime[n];
7. int totalWaitTime = 0, totalTurnAroundTime = 0;
8. // Calculate waiting time
9. waitTime[0] = 0;
10. for (int i = 1; i < n; i++) {
11. waitTime[i] = waitTime[i - 1] + burstTime[i - 1];
12. }
13. // Calculate turnaround time
14. for (int i = 0; i < n; i++) {
15. turnAroundTime[i] = waitTime[i] + burstTime[i];
16. totalWaitTime += waitTime[i];
17. totalTurnAroundTime += turnAroundTime[i];
18. }
19. // Print results
20. printf("Process\tBurst Time\tWaiting Time\tTurnaround Time\n");
21. for (int i = 0; i < n; i++) {
22. printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], burstTime[i], waitTime[i], turnAroundTime[i]);
23. }
24. printf("\nAverage Waiting Time: %.2f\n", (float)totalWaitTime / n);
25. printf("Average Turnaround Time: %.2f\n", (float)totalTurnAroundTime / n);
26. }
27. int main() {
28. int processes[] = {1, 2, 3, 4};
29. int burstTime[] = {5, 3, 8, 6};
30. int n = sizeof(processes) / sizeof(processes[0]);
31. printf("FCFS Process Scheduling:\n");
32. fcfs(processes, n, burstTime);
33. return 0;
34. }
35. Save and exit.
36. Compile and run:
37. gcc process\_management.c -o process\_management
38. ./process\_management

You'll see a table of process scheduling with waiting and turnaround times.

**Feature 2: Memory Management (malloc/free)**

1. Create a new file:
2. nano memory\_management.c
3. Add this code:
4. #include <stdio.h>
5. #include <stdlib.h>
6. void memory\_demo() {
7. int \*arr, n;
8. printf("Enter the number of elements: ");
9. scanf("%d", &n);
10. arr = (int \*)malloc(n \* sizeof(int));
11. if (arr == NULL) {
12. printf("Memory allocation failed!\n");
13. return;
14. }
15. printf("Enter %d elements:\n", n);
16. for (int i = 0; i < n; i++) {
17. scanf("%d", &arr[i]);
18. }
19. printf("You entered: ");
20. for (int i = 0; i < n; i++) {
21. printf("%d ", arr[i]);
22. }
23. free(arr);
24. printf("\nMemory freed successfully.\n");
25. }
26. int main() {
27. memory\_demo();
28. return 0;
29. }
30. Save and exit.
31. Compile and run:
32. gcc memory\_management.c -o memory\_management
33. ./memory\_management

You'll test dynamic memory allocation and deallocation.

**Feature 3: Inter-Process Communication (IPC)**

1. Create a new file:
2. nano ipc\_shared\_memory.c
3. Add shared memory code:
4. #include <stdio.h>
5. #include <sys/ipc.h>
6. #include <sys/shm.h>
7. #include <string.h>
8. int main() {
9. key\_t key = 1234;
10. int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);
11. char \*str = (char \*)shmat(shmid, (void \*)0, 0);
12. printf("Write Data: ");
13. fgets(str, 1024, stdin);
14. printf("Data written in shared memory: %s\n", str);
15. shmdt(str);
16. return 0;
17. }
18. Save, compile, and test it:
19. gcc ipc\_shared\_memory.c -o ipc
20. ./ipc

**Feature 4: Process Synchronization with Mutex Locks**

Process synchronization is critical to ensure multiple threads or processes don't corrupt shared resources. We'll use **POSIX Threads (pthreads)** for this feature.

**Step 2: Write Mutex Synchronization Code**

1. Create a new file for this feature:
2. nano synchronization\_mutex.c
3. Add the following code:
4. #include <stdio.h>
5. #include <pthread.h>
6. #include <stdlib.h>
7. int counter = 0; // Shared resource
8. pthread\_mutex\_t lock; // Mutex lock
9. void\* increment\_counter(void\* arg) {
10. pthread\_mutex\_lock(&lock); // Locking
11. counter++;
12. printf("Thread %d incremented counter to %d\n", \*(int\*)arg, counter);
13. pthread\_mutex\_unlock(&lock); // Unlocking
14. return NULL;
15. }
16. int main() {
17. pthread\_t threads[5];
18. int thread\_ids[5];
19. // Initialize the mutex lock
20. if (pthread\_mutex\_init(&lock, NULL) != 0) {
21. printf("Mutex initialization failed\n");
22. return 1;
23. }
24. // Create 5 threads
25. for (int i = 0; i < 5; i++) {
26. thread\_ids[i] = i + 1;
27. pthread\_create(&threads[i], NULL, increment\_counter, &thread\_ids[i]);
28. }
29. // Wait for all threads to complete
30. for (int i = 0; i < 5; i++) {
31. pthread\_join(threads[i], NULL);
32. }
33. // Destroy the mutex lock
34. pthread\_mutex\_destroy(&lock);
35. printf("Final counter value: %d\n", counter);
36. return 0;
37. }
38. Save and exit (Ctrl+O, Enter, Ctrl+X).

**Step 3: Compile and Run**

1. Compile using the -pthread flag:
2. gcc synchronization\_mutex.c -o synchronization -pthread
3. Run the program:
4. ./synchronization

**Expected Output**

* Each thread will increment the shared counter one by one.
* You'll see output like:
* Thread 1 incremented counter to 1
* Thread 2 incremented counter to 2
* Thread 3 incremented counter to 3
* Thread 4 incremented counter to 4
* Thread 5 incremented counter to 5
* Final counter value: 5

This demonstrates how mutex locks prevent race conditions.

**Feature 5: I/O Handling using System Calls**

We’ll create a program to read from and write to a file using Linux system calls.

**Step 1: Write I/O Handling Code**

1. Create a new file:
2. nano io\_handling.c
3. Add the following code:
4. #include <stdio.h>
5. #include <fcntl.h>
6. #include <unistd.h>
7. int main() {
8. int file;
9. char buffer[100];
10. // Create and write to a file
11. file = open("output.txt", O\_WRONLY | O\_CREAT, 0644);
12. if (file < 0) {
13. printf("Error creating file\n");
14. return 1;
15. }
16. write(file, "Hello, OPTIMA OS!\n", 19);
17. close(file);
18. // Read from the file
19. file = open("output.txt", O\_RDONLY);
20. if (file < 0) {
21. printf("Error opening file\n");
22. return 1;
23. }
24. read(file, buffer, sizeof(buffer));
25. printf("File content: %s\n", buffer);
26. close(file);
27. return 0;
28. }
29. Save and exit.

**Step 2: Compile and Run**

1. Compile the program:
2. gcc io\_handling.c -o io\_handling
3. Run it:
4. ./io\_handling

**Expected Output**

* The program creates a file named output.txt and writes "Hello, OPTIMA OS!" to it.
* It then reads the content of the file and displays:
* File content: Hello, OPTIMA OS!

**Next Steps**

* Integrate all features into one **main program**.
* Add a **menu-driven interface** for user interaction.
* Test and debug the entire system.

Should we start integrating everything, or do you want to refine individual features further?

Awesome! Now that we've covered **Process Synchronization with Mutex Locks** and **I/O Handling using System Calls**, let's move on to the next step: **Integrating Everything into the Main Program**.

**Step 1: Create the Main Program**

Now that we have individual features, we’ll integrate all the functionality into a **main menu-driven program**. This way, users can choose the feature they want to test (such as Process Management, Memory Management, Synchronization, and I/O Handling).

**Step 1.1: Set Up the Project Folder**

To keep things organized:

mkdir optima\_os

cd optima\_os

**Step 1.2: Create a New File for the Main Program**

nano main.c

**Step 1.3: Integrating the Features**

We’ll combine the Process Management (FCFS), Memory Management (malloc/free), Synchronization (Mutex), and I/O Handling into the main program. Here’s an example integration for you to start with:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <fcntl.h>

#include <unistd.h>

// Global variables for shared resources

int counter = 0;

pthread\_mutex\_t lock;

// Function for Process Management (FCFS)

void process\_management() {

printf("\nProcess Management (FCFS) Example:\n");

// Simulate FCFS with simple print statements

printf("Process 1: Task 1\n");

printf("Process 2: Task 2\n");

printf("Process 3: Task 3\n");

}

// Function for Memory Management (malloc/free)

void memory\_management() {

printf("\nMemory Management Example:\n");

int \*arr = (int \*)malloc(5 \* sizeof(int));

if (arr == NULL) {

printf("Memory allocation failed!\n");

return;

}

for (int i = 0; i < 5; i++) {

arr[i] = i \* 10;

printf("Memory[%d] = %d\n", i, arr[i]);

}

free(arr);

printf("Memory has been freed!\n");

}

// Function for Process Synchronization with Mutex

void\* increment\_counter(void\* arg) {

pthread\_mutex\_lock(&lock);

counter++;

printf("Thread %d incremented counter to %d\n", \*(int\*)arg, counter);

pthread\_mutex\_unlock(&lock);

return NULL;

}

void process\_synchronization() {

printf("\nProcess Synchronization Example:\n");

pthread\_t threads[5];

int thread\_ids[5];

// Initialize the mutex lock

if (pthread\_mutex\_init(&lock, NULL) != 0) {

printf("Mutex initialization failed\n");

return;

}

for (int i = 0; i < 5; i++) {

thread\_ids[i] = i + 1;

pthread\_create(&threads[i], NULL, increment\_counter, &thread\_ids[i]);

}

for (int i = 0; i < 5; i++) {

pthread\_join(threads[i], NULL);

}

pthread\_mutex\_destroy(&lock);

printf("Final counter value: %d\n", counter);

}

// Function for I/O Handling using System Calls

void io\_handling() {

printf("\nI/O Handling Example:\n");

int file;

char buffer[100];

file = open("output.txt", O\_WRONLY | O\_CREAT, 0644);

if (file < 0) {

printf("Error creating file\n");

return;

}

write(file, "Hello, OPTIMA OS!\n", 19);

close(file);

file = open("output.txt", O\_RDONLY);

if (file < 0) {

printf("Error opening file\n");

return;

}

read(file, buffer, sizeof(buffer));

printf("File content: %s\n", buffer);

close(file);

}

// Main Menu

void display\_menu() {

printf("\nOPTIMA OS - Main Menu\n");

printf("1. Process Management (FCFS)\n");

printf("2. Memory Management (malloc/free)\n");

printf("3. Process Synchronization (Mutex)\n");

printf("4. I/O Handling (System Calls)\n");

printf("5. Exit\n");

}

int main() {

int choice;

while (1) {

display\_menu();

printf("Enter your choice: ");

scanf("%d", &choice);

switch (choice) {

case 1:

process\_management();

break;

case 2:

memory\_management();

break;

case 3:

process\_synchronization();

break;

case 4:

io\_handling();

break;

case 5:

printf("Exiting OPTIMA OS. Goodbye!\n");

exit(0);

default:

printf("Invalid choice! Please try again.\n");

}

}

return 0;

}

**Step 2: Compile and Run the Integrated Program**

1. Save and close the file.
2. Now, compile all the code:
3. gcc main.c -o optima\_os -pthread
4. Run the program:
5. ./optima\_os

**Step 3: Test the Features**

* When you run the program, you’ll be prompted with a menu to select different options (Process Management, Memory Management, Synchronization, and I/O Handling).
* Each feature will execute based on the user’s selection, and the results will be displayed on the terminal.

**Step 4: Additional Enhancements (Optional)**

* **File System Handling**: You can also enhance the I/O section to perform more advanced file operations like reading multiple files, creating directories, or handling errors more robustly.
* **Advanced Process Scheduling**: If you want to go beyond FCFS, you can implement **Round Robin** or **Priority Scheduling** algorithms.
* **UI Enhancements**: While we are using a CLI interface, you can explore creating a graphical user interface (GUI) using libraries like GTK or Qt.

**Next Steps**

* After testing the program, ensure everything works correctly.
* Refine the features based on feedback or additional requirements.
* Add comments in the code for better readability.
* **Documentation**: Write a brief report on how each feature works and any challenges you faced.

Would you like to proceed with testing or enhancing any of the features? Let me know!

Great! Now that you have the basic integration working, the next step is to **debug, test, and optimize** your project. Let's break it down further into steps:

### ****Step 5: Debugging and Testing****

Now, let's test the system thoroughly to make sure everything is working as expected.

#### ****5.1: Check Process Management (FCFS)****

* Test the FCFS scheduling feature by simulating a few more processes. For instance, add more print statements or create additional tasks to see if they’re handled in the correct order.
* If the output is as expected (tasks executed in the order they were received), your process management is working fine.

#### ****5.2: Check Memory Management****

* Ensure that the memory allocated via malloc() is properly freed with free() after use.
* Run the program multiple times to check if memory leaks happen or if any malloc() fails.
* Add error handling for malloc() to check if memory allocation was successful:
* if (arr == NULL) {
* printf("Memory allocation failed\n");
* exit(1);
* }

#### ****5.3: Check Process Synchronization****

* Since the synchronization uses mutex locks, test it by adding additional threads or by trying to access the counter variable in different sections of the program concurrently.
* You can even add print statements before and after each lock to verify that threads are waiting for the lock and that no race conditions occur.

#### ****5.4: Check I/O Handling****

* Ensure that the file is created properly, data is written correctly, and it's read back as expected. If there are issues, check the file permissions, or whether the file path is correct.
* Make sure the file content is printed correctly and that the open()/close() system calls are handled without error.