Title

Memory access scheduler to optimize memory bandwidth utilization in C

Objective

The goal of this project is to develop a memory access scheduler in C to optimize memory bandwidth utilization in multithreaded environments. The scheduler handles concurrent memory requests by dynamically dividing and scheduling memory blocks for multiple threads. The objective is to minimize memory access contention and maximize efficient data transfer between CPU and memory, thereby improving overall program performance in data-intensive tasks.

Theory

In modern computer systems, efficient memory access is crucial to maintaining high performance. This project leverages multithreading to parallelize memory access tasks. The memory access scheduler implements user-defined thread memory allocation, controlling how different memory blocks are accessed concurrently.

Memory bandwidth refers to the rate at which data can be read from or written to memory by the CPU. Optimizing memory bandwidth usage involves minimizing memory contention and organizing data access patterns to reduce cache misses and increase prefetching efficiency. In this project, the theory is applied through:

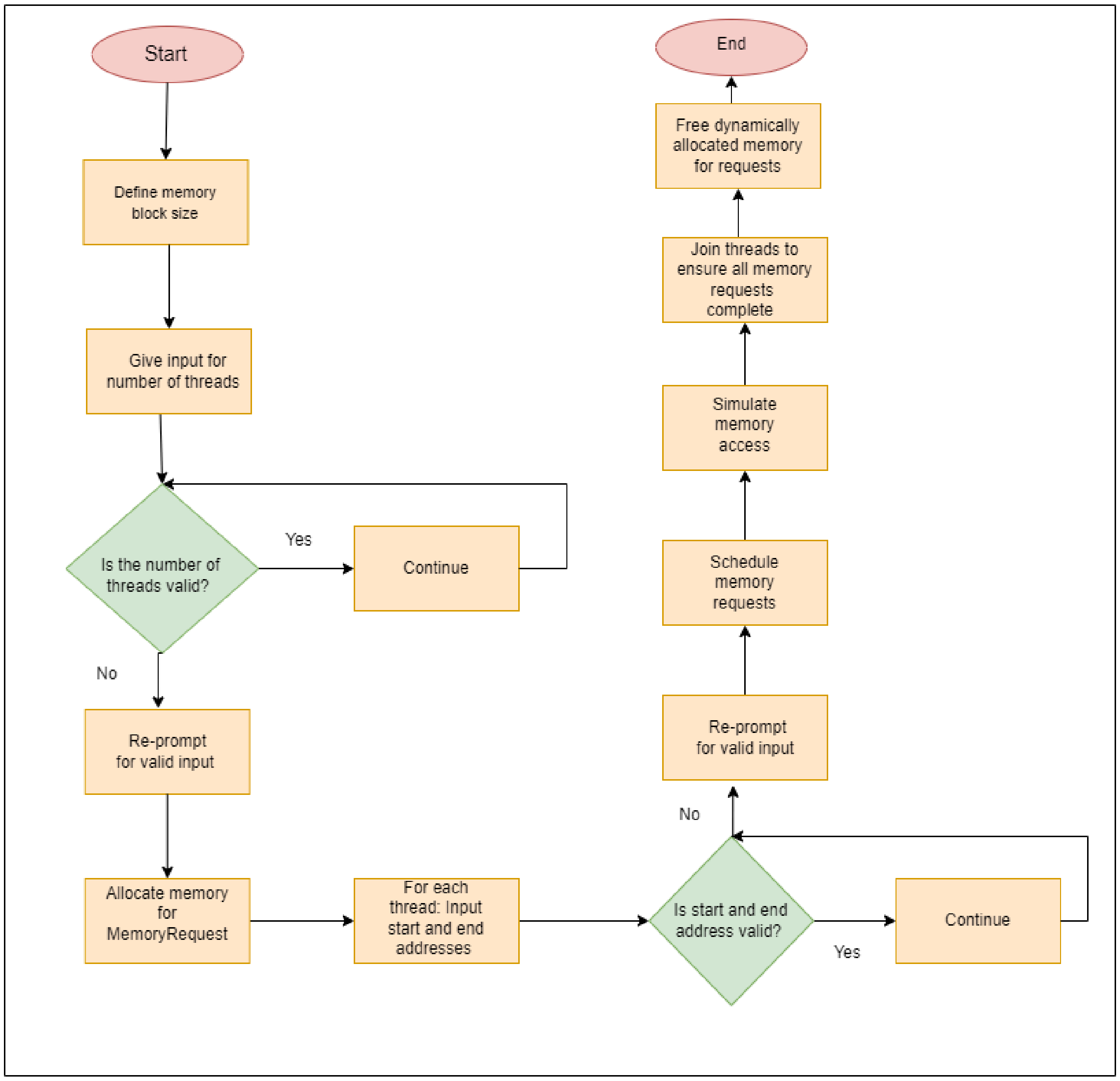
1. **Thread synchronization** using mutex locks to avoid race conditions.
2. **Efficient memory scheduling** by dividing memory blocks between threads.
3. **Round-robin memory access simulation** to balance memory usage between threads.

Design

The design of the scheduler involves several steps:

1. **Thread Creation**: User specifies the number of threads (dynamic input).
2. **Memory Block Assignment**: Each thread is assigned a memory block by the user. The program validates the start and end addresses for each block, ensuring there is no memory overflow.
3. **Mutex-based Synchronization**: To prevent race conditions, memory access is synchronized using mutex locks.
4. **Memory Access Simulation**: Each thread accesses its assigned memory range sequentially, simulating reading and writing operations.
5. **Testing and Debugging**: After the threads complete their tasks, they join the main process, ensuring all tasks finish without conflicts.

Flowchart



Start
Define memory block size
Give input for number of threads
Is the number of threads valid?
Yes
Continue
No
Re-prompt for valid input
Allocate memory for MemoryRequest
For each thread: Input start and end addresses
Is start and end address valid?
Yes
Continue
No
Re-prompt for valid input
Schedule memory requests
Simulate memory access
Join threads to ensure all memory requests complete
Free dynamically allocated memory for requests
End

Algorithm

Algorithm: Memory Access Scheduler Simulation

1. Initialize Program

Step 1: Define the constant MEMORY_BLOCK_SIZE to represent the size of the simulated memory (1024 bytes).

Step 2: Define the structure MemoryRequest with the fields:

request_id: Unique ID for the memory request.

start_address: Starting address of the memory block.

end_address: Ending address of the memory block.

Step 3: Initialize the simulated memory array memory[MEMORY_BLOCK_SIZE].

Step 4: Initialize a mutex memory_mutex for thread synchronization.

2. Get User Input for Threads

Step 5: Ask the user for the number of threads they want to use.

Step 6: Validate the input:

If the number of threads is not between 1 and MEMORY_BLOCK_SIZE, prompt the user again until valid input is given.

3. Allocate Memory for Requests

Step 7: Allocate memory dynamically for the MemoryRequest array to store thread requests based on the number of threads.

4. Get Memory Access Ranges for Each Thread

Step 8: For each thread:

Step 9: Ask the user for the start and end addresses for the memory access request.

Step 10: Validate the memory range:

If the start address or end address is outside the valid range or if the end address is less than the start address, prompt the user again until valid input is given.

5. Schedule and Handle Memory Access

Step 11: Schedule the memory requests by creating one thread for each memory access request.

Step 12: Each thread performs the following steps inside the memory_access function:

Step 12.1: Lock the memory using the mutex (pthread_mutex_lock).

Step 12.2: Write to the memory block within the range defined by the start and end addresses.

Step 12.3: Unlock the memory (pthread_mutex_unlock) after the memory access is complete.

Step 12.4: Simulate a small delay to represent memory latency (1 millisecond).

6. Join Threads

Step 13: After scheduling all threads, wait for each thread to finish its execution using pthread_join.

7. Complete Execution

Step 14: Once all threads have completed their memory access, print a success message indicating that all memory requests have been completed.

Step 15: Free the dynamically allocated memory for the memory requests array.

Step 16: End the program

Implementation

The memory access scheduler was implemented using **C** with the **pthread** library for multithreading and **mutexes** for synchronization. The following is a summary of the main components:

1. **User Input**: Users input the number of threads and their respective memory ranges (start and end addresses).
2. **Memory Simulation**: A simulated memory block of size MEMORY_BLOCK_SIZE (1024 bytes) is initialized.
3. **Thread Execution**: Each thread is responsible for accessing and modifying its assigned block of memory, with the memory access synchronized using mutexes.
4. **Thread Joining**: After the threads complete, they join the main process to ensure that all memory tasks are completed.

The following libraries and system components were used:

- **C Programming Language**.
- **pthread library** for thread management.
- **Mutexes** for thread synchronization.
- **POSIX System Functions** to simulate thread yielding and delays.

Debugging-Test-run

To ensure the program functions as expected, the following test cases were considered:

1. **Sequential Memory Access**: Tested with non-overlapping memory ranges to ensure threads do not interfere with each other.
2. **Overlapping Memory Access**: Tested with overlapping ranges to ensure the mutex lock mechanism prevents race conditions.
3. **Boundary Testing**: Ensured that the program correctly handles boundary conditions (e.g., start address = 0, end address = 1023).
4. **Multiple Threads**: Tested with different numbers of threads, including the maximum number of threads that the memory block could support (up to 1024).
5. **User Input Validation**: All user inputs (number of threads, start and end addresses) were validated to ensure valid ranges and prevent invalid memory accesses.

Results Analysis

The results of the program can be evaluated based on:

1. **Average and Worst-Case Time Complexity**: The program operates in **O(n)** time complexity where n is the size of the memory block for each thread. Each thread accesses memory sequentially, making it linear in complexity.
2. **Thread Synchronization Overhead**: The use of mutexes introduces some overhead, but it ensures safe and predictable access to shared memory.
3. **Scalability**: The program scales well as the number of threads increases, with each thread being able to access its assigned memory block efficiently, though care must be taken to avoid excessive contention over memory.

Conclusion

The memory access scheduler successfully manages memory access for multiple threads, allowing concurrent and synchronized access to different memory blocks. The project demonstrates the importance of synchronization in multithreading to avoid data races and maintain consistent program execution.

Future Improvements

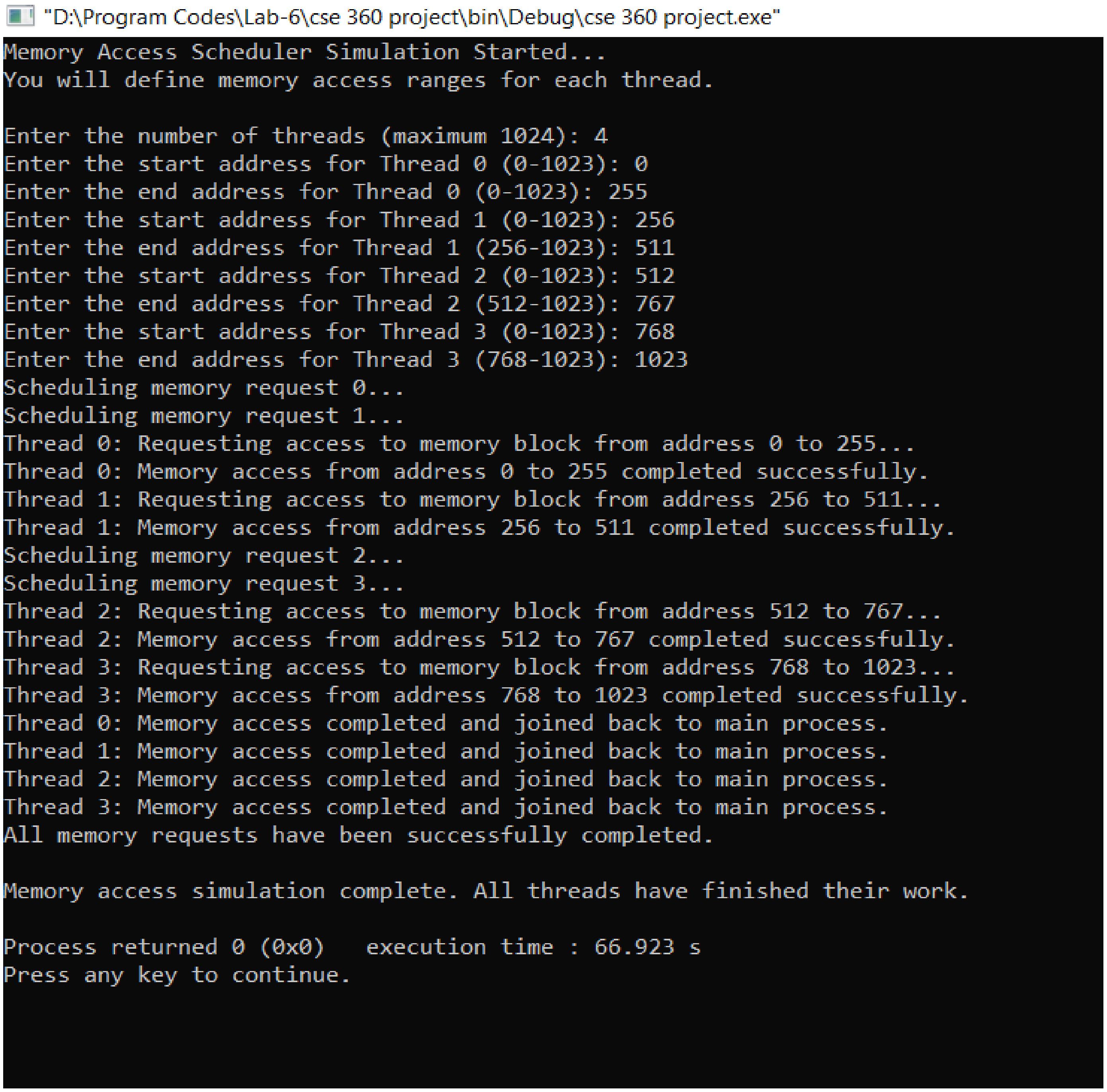
1. **Optimization for Cache Performance**: Introducing data prefetching and better memory access patterns could further improve the program's efficiency in systems with a multi-level cache hierarchy.
2. **Dynamic Load Balancing**: Implementing a dynamic scheduler that can adjust memory allocation based on current thread performance would optimize resource usage.
3. **Advanced Scheduling Algorithms**: Experimenting with other scheduling techniques (e.g., priority-based or time-sliced scheduling) could provide more efficient thread management in more complex systems.

CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define MEMORY_BLOCK_SIZE 1024    // Simulated memory block size

// Structure to represent a memory request
typedef struct {
    int request_id;       // Unique ID of the request
    int start_address;    // Start address of the memory block to be accessed
    int end_address;      // End address of the memory block to be accessed
} MemoryRequest;

// Simulated memory
char memory[MEMORY_BLOCK_SIZE];

// Mutex for controlling access to memory (for thread synchronization)
pthread_mutex_t memory_mutex = PTHREAD_MUTEX_INITIALIZER;

// Function to simulate memory access by each thread
void *memory_access(void *arg) {
    // Cast argument to MemoryRequest type
    MemoryRequest *request = (MemoryRequest *)arg;

    // Lock the memory to prevent race conditions
    pthread_mutex_lock(&memory_mutex);

    // Explain the memory access action for the current thread
    printf("Thread %d: Requesting access to memory block from address %d to %d...\n",
           request->request_id, request->start_address, request->end_address);

    // Simulate memory access by writing data
    for (int i = request->start_address; i <= request->end_address; i++) {
        memory[i] = request->request_id;  // Simulate writing to the memory block
    }

    // Inform that the memory access is complete
    printf("Thread %d: Memory access from address %d to %d completed successfully.\n",
           request->request_id, request->start_address, request->end_address);
```

```c
    // Unlock the memory after access is done
    pthread_mutex_unlock(&memory_mutex);

    // Simulate a small delay to represent memory latency
    usleep(1000);  // Sleep for 1 millisecond (simulating memory access time)
    pthread_exit(NULL);
}

// Function to schedule memory requests optimally
void schedule_memory_requests(MemoryRequest *requests, int num_requests) {
    pthread_t threads[num_requests];

    // Create and schedule memory access requests using multiple threads
    for (int i = 0; i < num_requests; i++) {
        // Inform about the thread creation for memory request
        printf("Scheduling memory request %d...\n", requests[i].request_id);

        // Create threads for each memory request
        pthread_create(&threads[i], NULL, memory_access, (void *)&requests[i]);
    }

    // Join threads to ensure all memory requests finish
    for (int i = 0; i < num_requests; i++) {
        pthread_join(threads[i], NULL);
        printf("Thread %d: Memory access completed and joined back to main process.\n",
               requests[i].request_id);
    }

    // Inform that all memory accesses have been completed
    printf("All memory requests have been successfully completed.\n");
}

int main() {
    int num_threads;

    // Explain the program's purpose to the user
    printf("Memory Access Scheduler Simulation Started...\n");
    printf("You will define memory access ranges for each thread.\n\n");

    // Get the number of threads from the user
    printf("Enter the number of threads (maximum %d): ", MEMORY_BLOCK_SIZE);
    scanf("%d", &num_threads);

    // Validate the number of threads
    while (num_threads <= 0 || num_threads > MEMORY_BLOCK_SIZE) {
                printf("Invalid number of threads. Please enter a value between 1 and %d: ",
MEMORY_BLOCK_SIZE);
        scanf("%d", &num_threads);
    }

    // Dynamically allocate memory for requests based on the number of threads
    MemoryRequest *requests = (MemoryRequest *)malloc(num_threads * sizeof(MemoryRequest));

    // Get user input for memory request ranges for each thread
    for (int i = 0; i < num_threads; i++) {
        requests[i].request_id = i; // Assign the thread ID

        // Get the start and end address from the user
        printf("Enter the start address for Thread %d (0-%d): ", i, MEMORY_BLOCK_SIZE - 1);
        scanf("%d", &requests[i].start_address);

        // Validate start address input
        while (requests[i].start_address < 0 || requests[i].start_address >= MEMORY_BLOCK_SIZE) {
            printf("Invalid start address. Please enter a value between 0 and %d: ", MEMORY_BLOCK_SIZE
- 1);
            scanf("%d", &requests[i].start_address);
        }

                printf("Enter the end address for Thread %d (%d-%d): ", i, requests[i].start_address,
MEMORY_BLOCK_SIZE - 1);
        scanf("%d", &requests[i].end_address);

        // Validate end address input
                while (requests[i].end_address < requests[i].start_address || requests[i].end_address >=
MEMORY_BLOCK_SIZE) {
            printf("Invalid end address. Please enter a value between %d and %d: ", requests[i].start_address,
MEMORY_BLOCK_SIZE - 1);
            scanf("%d", &requests[i].end_address);
        }
    }

    // Schedule the memory requests for optimal bandwidth utilization
    schedule_memory_requests(requests, num_threads);

    // Free dynamically allocated memory
    free(requests);

    // Simulation complete
    printf("\nMemory access simulation complete. All threads have finished their work.\n");

    return 0;
}
```

Output



```
"D:\Program Codes\Lab-6\cse 360 project\bin\Debug\cse 360 project.exe"
Memory Access Scheduler Simulation Started...
You will define memory access ranges for each thread.

Enter the number of threads (maximum 1024): 4
Enter the start address for Thread 0 (0-1023): 0
Enter the end address for Thread 0 (0-1023): 255
Enter the start address for Thread 1 (0-1023): 256
Enter the end address for Thread 1 (256-1023): 511
Enter the start address for Thread 2 (0-1023): 512
Enter the end address for Thread 2 (512-1023): 767
Enter the start address for Thread 3 (0-1023): 768
Enter the end address for Thread 3 (768-1023): 1023
Scheduling memory request 0...
Scheduling memory request 1...
Thread 0: Requesting access to memory block from address 0 to 255...
Thread 0: Memory access from address 0 to 255 completed successfully.
Thread 1: Requesting access to memory block from address 256 to 511...
Thread 1: Memory access from address 256 to 511 completed successfully.
Scheduling memory request 2...
Scheduling memory request 3...
Thread 2: Requesting access to memory block from address 512 to 767...
Thread 2: Memory access from address 512 to 767 completed successfully.
Thread 3: Requesting access to memory block from address 768 to 1023...
Thread 3: Memory access from address 768 to 1023 completed successfully.
Thread 0: Memory access completed and joined back to main process.
Thread 1: Memory access completed and joined back to main process.
Thread 2: Memory access completed and joined back to main process.
Thread 3: Memory access completed and joined back to main process.
All memory requests have been successfully completed.

Memory access simulation complete. All threads have finished their work.

Process returned 0 (0x0)   execution time : 66.923 s
Press any key to continue.
```

Bibliography

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). **Operating System Concepts**. Wiley.
2. Tanenbaum, A. S., & Bos, H. (2014). **Modern Operating Systems**. Pearson.
3. Pthread library documentation. Retrieved from https://man7.org/linux/man-pages/man7/pthreads.7.html
4. Stallings, W. (2012). **Operating Systems: Internals and Design Principles**. Pearson.