Parallel Sorting Farm

Operating Systems – Multithreaded Programming Project

Name/ Surname : İrem Büşra Sürüm

No : 5001230034

1. Introduction

The purpose of this project is to demonstrate the concepts of multithreading and synchronization that are taught in the Operating Systems course. In this project, a large dataset is divided into smaller parts and processed in parallel by multiple threads.

In the developed application, an array containing random numbers is sorted using multiple worker threads, and the execution time of the program is measured. In addition, possible race condition problems that may occur on shared data are prevented by using a mutex mechanism.

2. Project Design

2.1 General Structure

The project is implemented using the C programming language and the POSIX Threads (pthreads) library. The main workflow of the program is as follows:

- A global array is created and filled with random numbers.
- The array is divided into equal-sized parts according to the number of threads.
- Each thread sorts its assigned portion of the array.
- A shared counter is protected using a mutex.
- The total execution time of the program is measured and printed.

This structure allows tasks to be executed in parallel and demonstrates core operating system concepts in practice.

```
#define ARRAY_SIZE 1000
#define NUM_THREADS 4
```

The array size and the number of worker threads are defined as constants to ensure a clear and configurable program structure.

2.2 Data Structures Used

The following data structures are used in the project:

- arr[]: A global array that stores the numbers to be sorted
- total_swaps: A shared counter accessed by all threads
- pthread_t threads[]: Thread definitions
- pthread_mutex_t mutex: Mutex used for synchronization

The global array is defined as shared memory that can be accessed by all threads.

```c
int arr[ARRAY_SIZE];
int total_swaps = 0;
pthread_mutex_t mutex;
```

The global array and shared counter are accessed by all threads. A mutex is used to protect shared data and prevent race conditions.

3. Thread Structure and Parallel Execution

3.1 Thread Creation

The minimum requirement of using at least three threads is satisfied, and a total of four worker threads are used in the program. Threads are created using the pthread_create function. Each thread determines which part of the array it will process based on its thread identifier (thread_id).

```c
void* thread_sort(void* arg) {
    int thread_id = *(int*)arg;

    int chunk_size = ARRAY_SIZE / NUM_THREADS;
    int start = thread_id * chunk_size;
    int end = (thread_id == NUM_THREADS - 1)
                ? ARRAY_SIZE - 1
                : start + chunk_size - 1;

    bubble_sort(start, end);
    return NULL;
}
```

Each thread calculates the range of the array it is responsible for and sorts only that specific segment.

3.2 Workload Distribution (Load Balancing)

The array to be sorted is divided into equal-sized segments according to the number of threads. This ensures that each thread receives approximately the same amount of work, achieving load balancing.

Since each thread operates only on its assigned segment of the array, direct data conflicts between threads are avoided. This structure clearly demonstrates the concept of task parallelism.

```c
pthread_create(&threads[i], NULL, thread_sort, &thread_ids[i]);
```

Threads are created using the pthread_create function, allowing concurrent execution of sorting tasks.

4. Synchronization and Mutex Usage

4.1 Need for Synchronization

In multithreaded applications, race conditions may occur when multiple threads access shared data simultaneously. Such situations can cause incorrect or inconsistent program behavior.

In this project, the variable total_swaps is updated by all threads. Therefore, this variable represents a critical section that requires synchronization.

4.2 Mutex Mechanism

To prevent race condition problems, a mutex (mutual exclusion) mechanism is used. The mutex ensures that only one thread can access the critical section at a time.

Before incrementing the total_swaps variable, a thread locks the mutex. After the update is completed, the mutex is unlocked. This approach guarantees data consistency.

```
pthread_mutex_lock(&mutex);
total_swaps++;
pthread_mutex_unlock(&mutex);
```

This code represents the critical section where the shared variable is safely updated using a mutex to prevent race conditions.

4.3 Conscious Use of Synchronization

Although each thread processes a different portion of the array, the mutex is deliberately applied to a shared variable in order to demonstrate proper synchronization, as required by the assignment.

In this way, a clear critical section is created and the correct use of a mutex is shown within the project.

5. Performance Measurement

The execution time of the program is measured using the clock() function. Timing starts before the threads are created and stops after all threads complete their execution.

The measured time is used to observe the overall cost of parallel execution. This measurement is sufficient to evaluate the effect of multithreading on program performance.

```
clock_t start_time = clock();
/* threads execute */
clock_t end_time = clock();
```

The execution time is measured to observe the performance impact of parallel execution.

6. Conclusion

In this project, the concepts of multithreading, synchronization, and shared memory discussed in the Operating Systems course are successfully applied. A large dataset is processed in parallel using multiple threads, and race condition problems are prevented with the help of a mutex.

The developed application clearly demonstrates task parallelism and load balancing, and presents the fundamental principles of multithreaded programming in an understandable manner.