Marmara University - Faulty of Engineering

Department of Computer Engineering

CSE3033 Operating Systems (Autumn 2023)

Submit Date: 04/01/2024.

Threads & Synchronization

Student Number (ID)	Name	Surname
150120998	Abdelrahman	Zahran
150120997	Mohamed	Nael Ayoubi
150120020	Mustafa	Said Çanak

Sections Of the Report: -

- Section (1): Introduction + Problem Definition.
- Section (2): Implementation Details.
- Section (3): Results.
- Section (4): Questions & Answers.

Section (1): Introduction + Problem Definition

Introduction:

The aim of this project is to develop a multithreaded program in C that efficiently calculates the sum of square roots within a user-specified range. The program offers three distinct methods of computation, each employing different thread synchronization strategies. Users will input parameters such as the range of start and end points, the number of threads, and the chosen synchronization method. The execution of the program will be measured using the Linux **time** command to analyze user, system, and total running times. This project leverages the PThread library and emphasizes proper synchronization techniques to ensure accurate and efficient results. The subsequent analysis includes comparing the performance of the three methods across various input configurations and addressing key questions regarding correctness, speed, scalability, and resource utilization.

Problem Definition:

The objective of this project is to develop a multithreaded program in C that calculates the sum of square roots of numbers within a given range. The program should be able to execute using three different methods, each involving a varying level of thread synchronization. The user will input the range (a-b), the number of threads (c), and the method (d) to be used.

Threads and synchronization are fundamental concepts in concurrent programming, where multiple tasks or processes run simultaneously. Understanding these concepts is crucial for developing efficient and reliable software that can make optimal use of modern multicore processors.

Threads: A thread is the smallest unit of execution within a process. Unlike traditional single-threaded programs, which execute sequentially, multithreaded programs can execute multiple threads concurrently, sharing the same resources within a process. Threads within a process share the same memory space, allowing them to communicate and exchange data more easily than separate processes.

Multithreading offers several advantages, such as improved responsiveness and better resource utilization. It enables concurrent execution of tasks, allowing a program to perform multiple operations simultaneously. However, it also introduces challenges related to data consistency and synchronization.

Synchronization: Synchronization is the coordination of multiple threads to ensure orderly and predictable execution. In a multithreaded environment, threads may access shared resources concurrently, leading to potential issues like data corruption, race conditions, and deadlocks.

Common synchronization mechanisms:

- **Locks/Mutexes (Mutual Exclusion):** Locks are used to ensure that only one thread can access a critical section of code at a time. This prevents multiple threads from interfering with each other and ensures data consistency.

Locks, also known as Mutexes (short for Mutual Exclusion), play a crucial role in concurrent programming by providing a mechanism to control access to shared resources. In a multithreaded environment, where multiple threads are executing simultaneously, it is essential to prevent interference and maintain data consistency. Locks serve this purpose by enforcing mutual exclusion, ensuring that only one thread can access a critical section of code at any given time.

Locks/Mutexes and their role in achieving mutual exclusion:

- Critical Sections:

A critical section is a portion of code that, if executed by multiple threads concurrently, could lead to data corruption or inconsistent results. It typically involves shared resources, such as variables, data structures, or files. The goal is to restrict access to the critical section so that only one thread can execute it at a time.

- Enforcing Mutual Exclusion:

Locks provide a mechanism to enforce mutual exclusion in critical sections. When a thread wants to enter a critical section, it must first acquire the lock associated with that section. If the lock is available, the thread obtains it and proceeds with the execution of the critical section. If the lock is already held by another thread, the requesting thread is blocked until the lock becomes available.

- Preventing Race Conditions:

Race conditions occur when multiple threads try to access shared resources simultaneously, leading to unpredictable and undesirable behavior. Locks prevent race conditions by ensuring that only one thread can execute the critical section at any given time. This guarantees the orderly execution of the code within the critical section, preventing conflicts and maintaining data integrity.

- Deadlocks:

While locks are essential for preventing race conditions, improper usage can lead to deadlocks. A deadlock occurs when two or more threads are blocked indefinitely because each is waiting for the other to release a lock. To avoid deadlocks, it's crucial to follow best practices, such as acquiring locks in a consistent order and releasing them in a timely manner.

In summary, locks are a fundamental tool in concurrent programming, providing a means to achieve mutual exclusion and prevent race conditions, ultimately contributing to the development of robust and thread-safe applications. Careful consideration of critical sections and proper usage of locks are essential for creating reliable and efficient multithreaded programs.

- **Semaphores:** Semaphores control access to a resource by limiting the number of threads that can access it simultaneously. They are useful for scenarios where a certain number of threads should be allowed to access a resource concurrently.

- **Condition Variables:** Condition variables are used to coordinate the execution of threads based on a particular condition. They are often used in producer-consumer scenarios.

Effective synchronization is essential for writing robust concurrent programs. It ensures that threads work together harmoniously, preventing data inconsistencies and preserving the integrity of shared resources. However, improper synchronization can lead to subtle and hard-to-debug issues, emphasizing the importance of careful design and testing in multithreaded applications.

Section (2): Implementation

In the implementation of this project, we used C programming language with Linux system. The program is evaluated using the Linux time command to obtain user, system, and total running times. The output will include the sum value found, user time, system time, and total time for each execution.

The methods delineate distinct approaches for concurrently updating a shared variable, global_sqrt_sum, within a multi-threaded environment.

Method 1 (d = 1):

In this method, each thread independently updates the global_sqrt_sum variable without utilizing any critical sections or mutex variables. The absence of mutexes implies that there are no explicit mechanisms in place to prevent race conditions, making it susceptible to concurrent access issues. This method might lead to unpredictable results as threads interfere with each other during simultaneous updates.

```c
// Worker function for Method 1
CodiumAI: Options | Test this function
void* method1_worker(void* arg) {
    // Extract thread-specific information from the ThreadInfo struct
    struct ThreadInfo* thread_info = (struct ThreadInfo*)arg;
    long long int start = thread_info->start;
    long long int end = thread_info->end;
    free(arg);

    // Calculate the square root sum for the specified range
    for (long long int x = start; x <= end; x++) {
        global_sqrt_sum += sqrt(x);
    }

    // Print information about the thread's range
    printf("  #=>Thread ID %lu calculating sqrt sum for range [%lld, %lld]\n", (unsigned long)pthread_self(), start, end);

    // Print local sum for the thread
    printf("###===>>>Thread ID %lu updated global_sqrt_sum without mutex. Final local_sqrt_sum = %e\n", (unsigned long)pthread_self(), global_sqrt_sum);

    pthread_exit(NULL);
}
```

Method 2 (d = 2):

Here, all threads still contribute to updating global_sqrt_sum, but this time they do so serially. A single shared mutex variable is introduced to safeguard the global_sqrt_sum updates. The mutex ensures that only one thread at a time can access and modify the shared variable. While this method mitigates the risk of race conditions, it introduces a trade-off by potentially limiting parallelism, as threads must now wait for the mutex, potentially leading to increased execution times.

```c
// Worker function for Method 2
CodiumAI: Options | Test this function
void* method2_worker(void* arg) {
    // Extract thread-specific information from the ThreadInfo struct
    struct ThreadInfo* thread_info = (struct ThreadInfo*)arg;
    long long int start = thread_info->start;
    long long int end = thread_info->end;
    free(arg);

    // Calculate and update global sum serially
    for (long long int x = start; x <= end; x++) {
        // Critical section (mutual exclusion) - only one thread can access this code at a time
        // Lock the mutex to ensure exclusive access to shared data
        pthread_mutex_lock(&mutex);
        // Update global_sqrt_sum value
        global_sqrt_sum += sqrt(x);
        // Unlock the mutex to release the lock
        pthread_mutex_unlock(&mutex);
    }

    // Print information about the thread's range
    printf("  #=>Thread ID %lu calculating sqrt sum for range [%lld, %lld]\n", (unsigned long)pthread_self(), start, end);

    // Print the final global_sqrt_sum after the update
    printf("###===>>>Thread ID %lu updated global_sqrt_sum with mutex. Final local_sqrt_sum = %e\n", (unsigned long)pthread_self(), global_sqrt_sum);

    pthread_exit(NULL);
}
```

Method 3 (d = 3):

In this method, each thread employs a local variable, local_sqrt_sum, of type double. Threads independently calculate the sum of square roots and store the result in their respective local variables. After completing the local computations, each thread uses a shared mutex variable to safeguard the update of global_sqrt_sum. This approach introduces a level of parallelism during the local calculations, reducing contention for the shared mutex. It combines the benefits of parallel computation with the necessary synchronization to ensure the accuracy of the global result.

```
// Worker function for Method 3
CodiumAI: Options | Test this function
void* method3_worker(void* arg) {
    // Extract thread-specific information from the ThreadInfo struct
    struct ThreadInfo* thread_info = (struct ThreadInfo*)arg;
    long long int start = thread_info->start;
    long long int end = thread_info->end;
    free(arg);

    // Local sum for the thread
    double local_sum = 0;

    // Calculate local sum
    for (long long int x = start; x <= end; x++) {
        local_sum += sqrt(x);
    }

    // Print information about the thread's range
    printf("  #=>Thread ID %lu calculating sqrt sum for range [%lld, %lld]\n", (unsigned long)pthread_self(), start, end);

    // Print local sum for the thread
    printf("###===>>>Thread ID %lu local_sqrt_sum = %e\n", (unsigned long)pthread_self(), local_sum);

    // Update global sum only once after local computations are complete
    pthread_mutex_lock(&mutex);
    global_sqrt_sum += local_sum;
    pthread_mutex_unlock(&mutex);

    // Return local sum
    pthread_exit(NULL);
}
```

Section (3): Results

The results of the 3 methods runs:

Method (1):

In this method, all threads concurrently update the global_sqrt_sum without any critical sections or mutex variables. This approach leads to a lack of synchronization among threads, resulting in a race condition.

Performance Analysis:

- As the number of threads increases, the sum value becomes inconsistent.
- Real time, user time, and system time increase with the number of threads.
- Lack of synchronization leads to incorrect results.

No. of Threads	Sum	Real Time	User Time	System Time
1	4.053457e+16	2m2.129s	2m1.809s	0m0.311s
2	2.363931e+16	3m12.345s	6m22.872s	0m0.791s
4	1.219876e+16	3m55.419s	15m36.440s	0m1.713s
8	7.236127e+15	4m0.726s	29m29.591s	0m48.498s

Method (2):

This method introduces a single shared mutex variable to protect the critical section where global_sqrt_sum is updated. Each thread acquires the mutex before updating the sum, ensuring mutual exclusion.

Performance Analysis:

- Real time, user time, and system time increase significantly with the number of threads.
- The use of mutex ensures correct results, but the overhead of synchronization becomes apparent.
- The performance improvement over Method 1 is limited due to the serialized access.

No. of Threads	Sum	Real Time	User Time	System Time
1	4.053457e+16	10m51.102s	10m51.092s	0m0.000s
2	4.053457e+16	52m20.006s	67m28.216s	35m58.594s
4	4.053457e+16	51m59.869s	67m11.020s	99m3.604s
8	4.053457e+16	68m37.965s	87m35.964s	284m56.042s
16	4.053457e+16	58m12.814s	71m18.633s	355m19.457s
32	4.053457e+16	62m53.580s	79m50.540s	364m5.106s

Method (3):

In this method, each thread calculates a local sum (local_sqrt_sum) using a local variable and updates the global sum after completing its computations. A single mutex variable ensures synchronization during global sum updates.

Performance Analysis:

- Real time, user time, and system time decrease significantly with the number of threads.
- The use of local variables reduces contention, resulting in better parallelism.
- The method demonstrates improved scalability compared to Method 2.

No. of Threads	Sum	Real Time	User Time	System Time
1	4.053457e+16	1m40.657s	1m40.656s	0m0.000s
2	4.053457e+16	1m3.215s	2m6.153s	0m0.061s
4	4.053457e+16	0m51.116s	3m23.274s	0m0.462s
8	4.053457e+16	0m33.623s	4m0.130s	0m3.231s
16	4.053457e+16	0m33.399s	4m11.107s	0m1.432s
32	4.053457e+16	0m33.686s	4m16.605s	0m2.277s

Overall Observations:

- Method 3 exhibits the best performance among the three methods.
- Method 2, despite ensuring correctness with a mutex, suffers from increased overhead and limited scalability.
- Method 1 lacks synchronization, leading to incorrect results and poor performance.

Section (4): Questions & Answers

1. **Which method(s) provide the correct result and why?**

All three methods - Method 1, Method 2, and Method 3 - are designed to provide the correct result, which is the sum of the square roots of numbers in the specified range. However, the correctness of the result depends on how well the program handles concurrent access to shared variables (global variable (**global_sqrt_sum**)). Method 1, which allows threads to update **global_sqrt_sum** concurrently without any critical sections or mutexes, introduces the risk of a race condition. A race condition occurs when multiple threads access and modify shared data simultaneously, leading to unpredictable and incorrect results. Therefore, while Method 1 may produce the correct result in some executions, it is not guaranteed to be correct due to the lack of proper synchronization mechanisms.

2. **Among the method(s) providing the correct result, which method is the fastest?**

Method 3, where each thread calculates a local sum and then updates **global_sqrt_sum** using a mutex after local computations, is expected to be more efficient and potentially faster. This efficiency arises from the reduction in contention for the global variable. In Method 3, each thread operates on a local variable (**local_sqrt_sum**) during its computation, minimizing the need for frequent access to the shared global variable. By using local computations and only acquiring the mutex for the final update to the global sum, Method 3 reduces the overhead of thread contention and synchronization. This approach allows threads to work more independently, potentially leading to better parallelization and improved overall performance.

3. **Among the method(s) providing the correct result, does increasing the number of threads always result in smaller total time? Discuss this considering the number of CPU cores available in your computer (in Linux, lscpu command provides the number of CPU cores available in your computer).**

No, increasing the number of threads does not always guarantee a reduction in total time. The impact of additional threads on total time depends on various factors, including the nature of the workload, the efficiency of parallelization, and the system's characteristics. While adding more threads can enhance parallelism and speed up certain computations, it may also introduce overhead. Factors such as contention for shared resources, the cost of thread creation and joining, and the specific characteristics of the workload can influence the overall performance. In some cases, an excessive number of threads might lead to diminishing returns or even performance degradation.

The performance of parallelizing a task across multiple threads does not always guarantee a reduction in total execution time, and the number of CPU cores plays a crucial role in this.

Method 1:

- As the number of threads increases, the total time increases significantly.
- This is because each thread operates independently without any synchronization, leading to race conditions.
- The lack of synchronization results in multiple threads updating the global sum simultaneously, causing contention and slowing down the overall execution.

Method 2:

- The total time also increases as the number of threads increases.
- Although this method uses a mutex to synchronize access to the global sum, the contention for the mutex becomes a bottleneck.
- With more threads, the overhead of acquiring and releasing the mutex increases, leading to performance degradation.

Method 3:

- This method shows a reduction in total time as the number of threads increases.
- By calculating a local sum within each thread and updating the global sum in a critical section with a mutex, the contention for the mutex is reduced.
- As a result, the overhead of synchronization is minimized, leading to better parallel performance.

Considerations based on the system information:

- The system has 8 CPU cores (CPU(s): 8).
- Each core supports 2 threads (Thread(s) per core: 2).

General Observations:

- The optimal number of threads depends on the nature of the task, the level of parallelism it allows, and the available hardware resources.
- Increasing the number of threads beyond the number of physical CPU cores may lead to contention for resources and increased context-switching overhead, potentially degrading performance.

Method-Specific Observations:

- **Method 1 and Method 2:**
  - Show an increase in total time as the number of threads increases, indicating that the overhead of parallelization outweighs the benefits.
  - Contention for shared resources (lack of synchronization or mutex contention) is a significant factor.
- **Method 3:**
  - Demonstrates a reduction in total time as the number of threads increases.
  - The design of local computations followed by a single update to the global sum with a mutex minimizes contention and improves parallel efficiency.

In summary, the optimal number of threads depends on the characteristics of the task and the available hardware resources. Method 3, with its reduced synchronization overhead, tends to perform better in this scenario, but the effectiveness of parallelization may vary based on the specific characteristics of the workload and the underlying hardware architecture.

4. **Are there any differences in user time/system time ratio of the processes as the number of threads increases? What could be the cause of these differences?**

The user time/system time ratio might vary based on the synchronization method and the system's characteristics. Method 2 and Method 3, which involve the use of a mutex for synchronization, could exhibit differences in the user time/system time ratio. The ratio reflects the proportion of time spent in user mode (executing application code) to system mode (executing kernel or OS code). In scenarios where contention for the mutex is high, the user time might increase due to threads waiting for access to the critical section. The system time may also increase as the kernel manages thread synchronization. The specific impact on the user time/system time ratio would depend on the efficiency of the synchronization mechanism and the overall system workload.

Analyzing the user time/system time ratio for the three methods as the number of threads increases:

Method 1:

- The user time/system time ratio remains relatively stable across different numbers of threads.

- The ratio is influenced by the computation time spent in user mode (actual task execution) compared to system mode (kernel operations, including synchronization).

Method 2:

- The user time/system time ratio increases as the number of threads increases.
- This suggests that a larger proportion of time is spent in user mode relative to system mode as more threads are employed.
- The increased ratio may be due to the contention for the mutex, causing threads to spend more time in user mode waiting for access to the critical section.

Method 3:

- The user time/system time ratio decreases as the number of threads increases.
- This implies that a smaller proportion of time is spent in user mode relative to system mode with an increasing number of threads.
- The reduced ratio can be attributed to the efficient design of Method 3, where local computations minimize contention for the mutex, leading to less time spent in user mode waiting for synchronization.

Possible Causes of Differences:

1. **Contention for Resources:**
    - In Method 1, lack of synchronization leads to contention for shared resources, causing threads to wait for access to the global sum.
    - In Method 2, the use of a mutex introduces contention, especially with a higher number of threads, resulting in increased time spent waiting for the mutex in user mode.
2. **Efficiency of Synchronization:**
    - Method 3 is designed to minimize synchronization overhead by performing local computations before updating the global sum with a mutex. This leads to less contention and more efficient parallelization, reflected in a lower user time/system time ratio.
3. **Context Switching Overhead:**
    - The operating system performs context switching when switching between threads. As the number of threads increases, the overhead of context switching may impact the user time/system time ratio differently for each method.
4. **Task Granularity:**
    - The nature of the computation and the granularity of the task can influence the effectiveness of parallelization. Method 3's approach of local computations and less frequent synchronization may be more suitable for certain types of tasks.
5. **CPU Architecture:**
    - The characteristics of the CPU architecture, including cache behavior and inter-thread communication mechanisms, can impact the performance of parallelized tasks.

In summary, differences in the user time/system time ratio are observed, and these differences can be attributed to the efficiency of synchronization mechanisms, contention for resources, and the design choices made in each

method. Method 3, with its focus on minimizing synchronization overhead, demonstrates a more favorable ratio as the number of threads increases.

Rescources:

1. https://tldp.org/
2. https://www.linuxquestions.org/
3. https://linuxcommand.org/
