Studia i Materiały Informatyki Stosowanej, Tom 17, Nr 3, 2025
str. 17-20

SEMAFORY I MUTEXY – PORÓWNANIE ZE WZGLĘDU NA CZAS POTRZEBNY NA WYKONANIE PRZYKŁADOWEGO ZADANIA

Julia Cieślicka, Aleksandra Mreła

Uniwersytet Kazimierza Wielkiego
Wydział Informatyki
ul..Kopernika 1, 85-064 Bydgoszcz
e-mail: a.mrela@ukw.edu.pl

Streszczenie: *W ramach paradygmatu programowania równoległego opracowano kilka metod synchronizacji przepływu danych w celu ochrony sekcji krytycznej programu przed warunkami wyścigu powodującymi błędne lub nieokreślone działanie programu. W artykule zostanie porównane semafory i mutexy pod względem czasu potrzebnego na wykonanie przykładowego zadania. Chociaż semafory ogólne (zliczające)mają możliwość udostępnienia sekcji krytycznej dla kilku wątków równocześnie, to semafory binarne i mutexy działają zdecydowanie szybciej.*

Słowa kluczowe: *Programowanie równoległe, semafor, mutex, sekcja krytyczna, czas działania programu*

SEMAPHORES AND MUTEXES – COMPARISON IN TERMS OF TIME REQUIRED TO COMPLETE THE SAMPLE TASK

Abstract*: Within the parallel programming paradigm, several data-flow synchronization methods have been developed to protect a program's critical section from race conditions that cause erroneous or unpredictable execution. This article will compare semaphores and mutexes in terms of the time required to complete a sample task. Although general (counting) semaphores can share the critical section with multiple threads simultaneously, binary semaphores and mutexes perform significantly faster.*

Keywords: *Parallel programming, C++, semaphore, mutex, critical region, program runtime*

1. INTRODUCTION

Multithreaded programs must address race conditions when multiple threads attempt to modify concurrently the same variable, leading to nondeterministic or erratic program behavior. Processes or threads may request access to a shared variable to read or modify it, but the program's outcome depends on the order in which the actions are performed.

When analyzing an algorithm, it is important to identify critical sections of the program, as they can degrade or slow down application performance or cause deadlocks. Therefore, methods for identifying critical sections and examining their impact on overall program performance are essential [1].

A race condition occurs in multithreaded programs when two or more threads simultaneously access the same variable (resource) and at least one attempts to modify it [2,3].

To prevent race conditions, synchronization methods are used, which allow only one thread or a specified number of threads to access shared variables.

2. SEMAPHORES AND MUTEXES

In parallel programming, one way to synchronize threads is a semaphore, which controls access to a shared variable (resource) by multiple processes or threads. Edsger Dijkstra

first described the concept of semaphores [4]. Semaphores are implemented as integer variables.

Semaphores are binary or counting. Binary semaphores hold 0 or 1; counting semaphores accept values above 1 [4].

In the case of a binary semaphore, only one thread can access the critical section at a time. A counting semaphore allows multiple threads to access a critical section; therefore, it controls access to critical resources by a given number of threads [5].

Another method for synchronizing access to a critical section is to use a mutual exclusion lock (mutex). A mutex allows only one thread to access critical resources at a time by acquiring the mutex lock. A thread can read the value of a variable in the critical section only if it acquires the lock and blocks access for other threads. A thread that has completed work in the critical section must release the lock, allowing other threads to access [3,6].

A mutex, like a binary semaphore, allows only one thread to access the critical section at a time. However, there is a significant difference between the two in terms of lock ownership: once a thread acquires a mutex, it becomes the owner and can release it. In contrast, a binary semaphore does not track ownership—any thread can signal or release it. Furthermore, the purpose of using mutexes in parallel programming is to protect access to the critical section, and semaphores are used for signaling between different threads or to control access to variables in the critical section to a specified number of threads [7].

3. IMPLEMENTATION OF THE APPLICATION IN C++

To select a thread synchronization method, you can compare the execution time of a program containing different methods for controlling thread access to critical resources. The algorithm for calculating and displaying array element values was written in C++ and implemented in two online programming environments: OnlineGDB [8] and Programiz C++ Online Compiler [9].

The algorithm contains four threads, each of which defines and displays the values of a portion of the K[100][4] matrix. The operations of calculating and displaying the array element values are defined as the critical section of the algorithm. The program is implemented in three versions. The first version defines a counting semaphore that allows a variable number of threads to access the critical section. The second version defines a mutex that locks access to the critical section using the `lock()` function and releases it with the `unlock()` instruction. The third version creates a lock using the `lock_guard<mutex>guard()`.

In the first version of the program, we define a counting semaphore of type `sem_t` as a global variable. Then, in the main() function, we initialize the semaphore using the instruction `sem_init(&semaphore, 0, k)`, where the parameter `k` denotes the number of threads that have access to the critical section. The last instruction of the program, `sem_destroy(&semaphore)`, destroys the semaphore. The thread launcher function contains the instruction `sem_wait(&semaphore)` to lock the semaphore and the instruction `sem_post (&semaphore)` to unlock the semaphore.

In the second version, we define a mutex as a global variable. In the thread launcher function, we lock the critical section using the instruction `m.lock()` and then unlock it using the instruction `m.unlock()`. In the third version of the program, we use only one instruction to lock and unlock the mutex: `lock_guard<mutex>guard()`.

To measure time, variables startTime and endTime of type int were created to store the number of cycles that had passed, using the `clock()` function. Then, the difference was converted to seconds using the CLOCKS_PER_SEC constant.

4. RESULTS

Using the OnlineGDB online programming environment, 10 program runtime measurements were taken, each time varying the number of threads accessing the critical section in the semaphore initialization instruction from 4 to 1. The results are presented in Table 1.

Table 1. Program runtime measurement using a semaphore [8]

k = 4	k = 3	k = 2	k = 1
0.004379	0.003921	0.004199	0.001416
0.00292	0.004238	0.003594	0.002575
0.003739	0.002692	0.002301	0.002491
0.003766	0.004421	0.003744	0.002148
0.003185	0.004663	0.004857	0.001049
0.004007	0.004232	0.003495	0.0012
0.005843	0.003112	0.003533	0.001725
0.003057	0.002639	0.004192	0.001492
0.003646	0.004405	0.004299	0.001673
0.00371	0.003329	0.003752	0.001293

The results of the algorithm runtime measurements with a mutex are presented in Table 2.

Table 2. Program runtime measurement when mutexes are used [8]

Mutex with lock() and unlock()	Mutex with lock_guard<mutex>
0.00281	0.001567
0.00317	0.001431
0.001633	0.001883
0.001747	0.002203
0.001521	0.002848
0.001747	0.002691
0.003078	0.001591
0.00182	0.001513
0.002434	0.001206
0.002424	0.001889

Next, measurements were performed using the Programiz C++ Online Compiler environment. The measurement results for the semaphore are presented in Table 3.

Table 3. Program runtime measurement using the semaphore [9]

k = 4	k = 3	k = 2	k = 1
0.001956	0.002971	0.003014	0.000914
0.007309	0.003435	0.003603	0.001502
0.004294	0.004532	0.002528	0.000705
0.004293	0.00341	0.003352	0.001295
0.0059	0.003614	0.003265	0.000718
0.004171	0.002982	0.003355	0.000781
0.004771	0.003154	0.002657	0.001254
0.004538	0.004622	0.002947	0.000363
0.003769	0.001655	0.003824	0.000636
0.005462	0.002394	0.001925	0.001857

The results of the algorithm runtime measurements with mutexes are presented in Table 4.

Table 4 Program runtime measurement when mutexes are used [9]

Mutex with lock() and unlock()	Mutex with lock_guard<mutex>
0.001175	0.000822
0.001181	0.001291
0.000918	0.001407
0.001618	0.002135
0.001067	0.001041
0.001034	0.001361
0.001125	0.000892
0.000969	0.001437
0.00094	0.002281
0.001209	0.001663

Based on the data presented in Tables 1 – 4, average measurement times were determined and illustrated in Figure 1.

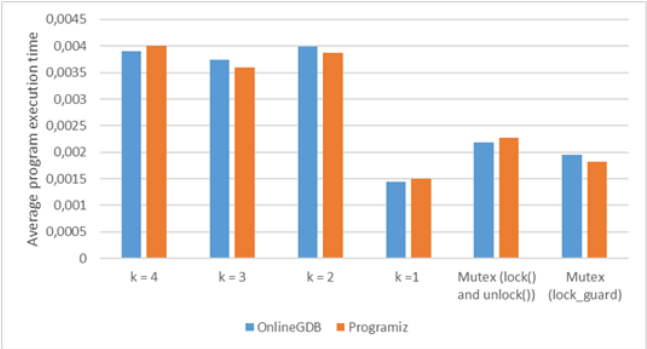

Fig. 1. Average task execution times depending on the synchronization method and programming environment used

Analyzing program execution times, we noticed that, regardless of the programming environment, using binary semaphores and mutexes results in significantly faster application execution.

5. SUMMARY AND CONCLUSION

This article compares the execution times of a computer program that defines four threads and uses semaphores and mutexes to block access to the critical section. Mutexes and binary semaphores provide faster program execution than counting semaphores. Counting semaphores, which allow two, three, or four threads to access the critical section, complete the task in a similar time.

Bibliography

1. Chen G., Stenstrom P., “Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications”, SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 2012, pp. 1-11, DOI: 10.1109/SC.2012.40.
2. Matuszek M., „Zjawisko wyścigu w programowaniu współbieżnym." Wydawnictwo Politechniki Gdańskiej, 2021, str. 261-270.
3. Silberschatz A., Galvin P. B., Gagne G. “Operating system concepts essentials”, Wiley Publishing, 2013.
4. Czech Z., „Wprowadzenie do obliczeń równoległych”, Wyd. 2. Warszawa: Wydawnictwo Naukowe PWN, 2013. ISBN 987-83-01-17290-9.
5. Williams A., “C++ concurrency in action”, Wyd. Simon and Schuster, 2019.
6. Tanenbaum A.S., Bos H., “Modern operating systems”, Pearson Education, Inc., 2015.
7. Ben-Ari M., “Principles of concurrent and distributed programming”, Pearson Education, 2006.
8. OnlineGDB**,** https://www.onlinegdb.com/online_c++_compiler (Accessed on 12 December 2025)
9. Programiz, C++ Online Compiler 0 https://www.programiz.com/cpp-programming/online-compile (Accessed on 12 December 2025)