«Системні технології» 5 (148) 2023 «System technologies»
DOI 10.34185/1562-9945-5-148-2023-01
UDC 004.4’6

O.O. Zhulkovskyi, I.I. Zhulkovska, V.V. Kostenko, O.F. Bulhakova

RESEARCH ON SYNCHRONIZATION AND DATA PROTECTION PROBLEMS IN IMPLEMENTING MULTITHREADED PROGRAMS

Abstract. The issue of shared data usage by threads is especially relevant in modern multi core and multiprocessor systems. The main problems of implementing multithreaded programs are race conditions, deadlocks, and thread starvation. The aim of the work is to solve the problem of thread racing in multithreaded calculations of resource intensive tasks with parallel access to shared data using appropriate synchronization mechanisms, such as mutexes. A multithreaded algorithm for implementing a typical task of processing large data arrays with protection of the critical area in concurrent programs running on multiprocessor and multi core systems has been developed and researched.
Keywords: multithreaded computing, parallel programming, thread racing, synchronization mechanisms, mutexes.

Introduction. Urgent problems of modern development of processes and technologies require constant improvement of computer equipment, efficient use of its resources, processing of large volumes of data, and support for the growing requirements of modern information systems.

One of the approaches to address the mentioned problems is the use of multiprocessor and multi-core computing systems, where multiple physical or logical cores can be efficiently utilized, ensuring faster and more productive parallel solutions to specific resource-intensive tasks [1]. Software for computers is adapted and refined to utilize such systems. Parallel techniques, such as parallel algorithms, parallel databases, and parallel programming, are becoming increasingly important to ensure optimal system performance. Big data processing, training, and implementation of complex artificial intelligence algorithms, especially deep learning, have also become integral to many industries. Moreover, multithreaded computing remains a key technology in the powerful video game programming industry with high-quality graphics, the Internet of Things (IoT), and helps solve various tasks in many sectors.

Thus, researching synchronization methods and resource management in multithreaded environments, developing new methods and tools for parallel program-

ming with the creation of efficient multithreaded cross-platform programs, and optimizing algorithms and architectures for scalable computations are relevant tasks.

Literature Review. Multithreading is a property of a platform (e.g., an operating system, virtual machine, etc.) that allows a process created by the operating system to consist of several threads that are executed "in parallel", i.e., without a fixed order in time. During the execution of some tasks, such a distribution can achieve more efficient use of the computing machine's resources [2].

Full parallel execution of tasks is possible only in a multiprocessor (or multicore) system. In the case of a single-processor multitasking system, processes are actually executed sequentially - here pseudo-parallel execution is used, creating the appearance of parallel work of several processes [2].

From the user's perspective, a process is an instance of a program during execution, and threads are branches of the program that are executed "in parallel". When one program performs many tasks, supporting multiple threads within one process allows distributing responsibility for different tasks among different threads, as well as increasing performance. Furthermore, tasks often need to exchange data, use shared data, or the results of other tasks. Threads within a process provide this capability, as they use the address space of the process to which they belong [2].

A thread can be in one of three states: executing, runnable, and waiting or blocked.

A multithreaded system can be implemented with scalability. For example, in parallel processing, the number of threads created can adapt to the number of processor cores in the system, allowing the program to accelerate within certain limits, fundamentally without changing its code.

When threads need to interact with each other or work with shared data, multithreading problems can arise, most of which are illustrated by the following classic tasks: about dining philosophers, about a sleeping barber, about smokers, about readers-writers, and others [2, 3].

The main problems of implementing multithreaded programs related to thread synchronization are [4–6]:

– race conditions, when two or more threads try to simultaneously access the same data or system resources without proper control, which can lead to unpredictable results;

– deadlocks, when two or more threads mutually block each other, waiting for mutual unlocking, as a result of which the program seems to freeze;

– thread starvation, when one thread is given excessive access to resources,

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

and other threads suffer from insufficient access to these resources.

Research Objective. The aim of this study is to solve the problem of thread racing in multithreaded calculations of resource-intensive tasks with parallel access to shared data using appropriate synchronization mechanisms, such as mutexes.

To achieve this goal, the following tasks are formulated: development of a multithreaded algorithm for implementing a typical task of processing large data arrays with protection of the critical area using synchronization primitives - mutexes; research of the performance of the developed algorithm with a significant amount of processed data and a variable number of computing threads; development of a concept for further application of effective approaches to data protection in concurrent programs implemented on multiprocessor and multi-core systems.

Research methodology and results. In this work, the problem of race condition is explored - an error in designing a multithreaded system where the program's operation depends on the order of code execution. Since this error is a «heisenbug», it can manifest randomly at different times, making it unpredictable and challenging to analyses, detect, and correct due to its variable behaviour.

The issue of shared data usage by threads is especially relevant in modern multi-core and multiprocessor systems, where many threads compete for access to shared resources, such as memory or files.

Research in this area aims to develop synchronization algorithms that allow threads to interact with shared data without conflicts and ensure the program's correct execution. It's also essential to optimize access to shared resources to ensure efficiency and speed of program execution. This issue is crucial for software developers as it allows efficient use of modern computing systems' capabilities and creates high-performance and reliable applications that can operate under intense resource competition.

When different threads have access to shared data, especially modifying them, synchronization of such access is required.

The thread race problem can be solved using synchronization and appropriate synchronization mechanisms, such as mutexes, conditional variables, atomic operations, and semaphores [2, 3].

Mutexes allow blocking access to shared data to one thread at a particular time. The thread that first locks the mutex gets access to the data, and other threads wait until the mutex is released.

When designing a program to reduce thread conflict, immutable design can avoid the race condition. That is, it's desirable to avoid shared access to mutable da-

ta from multiple threads or use immutable objects and data.

The choice of a specific method depends on the task and context.

Consider the thread race problem by solving a typical task of finding the sum of elements of a massive array in multithreaded mode.

The algorithm for solving this problem is implemented in C++ in the Microsoft Visual Studio 2022 IDE using the std::vector class - a dynamic array from the standard STL (Standard Template Library), which provides a convenient and safe way to manage memory and array size.

The program generates a large array of integers and uses a user-specified number of threads for parallel calculation of the sum of array elements. The std::thread class is used to create and manage threads in the program [7]. Each thread is responsible for a specific segment of the array, for which it calculates the sum of elements in a specially developed function. After the calculations are completed, the program displays the total sum and execution time.

To avoid thread races when several threads simultaneously try to access a shared variable storing the sum of array elements, the std::mutex class is used. The mutex ensures the correct calculation of the sum of the given array elements. In this program, to simulate a more significant mutex capture load, the mutex was captured on each iteration of the sum accumulation loop. Of course, in practice, using such an approach is justified only for experimental purposes.

For the experiments, the following PC infrastructure was used: Intel Core i7-12700H (14 cores, 2.3 GHz / 4.7 GHz); Goodram DDR4 (16 GB)×2 = 32 GB; Microsoft Windows 10; IDE Microsoft Visual Studio C++ 2022.

The size of the output array varied in the range from 100 million to 1 billion elements.

Fig. 1 presents the results of the implementation of the described program without blocking the critical area with a mutex, i.e., with the thread race problem and a deliberately incorrect result of the sum calculation.

As can be seen, with an increase in the size of the output array, the computation time significantly increases, while splitting the program into separate computational threads contributes to a noticeable increase in its performance. For example, with an increase in the array size in the range of 10^8–10^9, the program execution time increases by 6–7 times in multithreaded (2–14 threads) and ten times in single-threaded implementation. The use of a multithreaded algorithm instead of a traditional single-threaded one provides a 1.5–2 times increase in program execution speed, depending on the array size. It should be noted that the results mentioned are

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

merely informative, as they were obtained without using mechanisms to avoid thread races, and the sum calculation here is most likely incorrect.

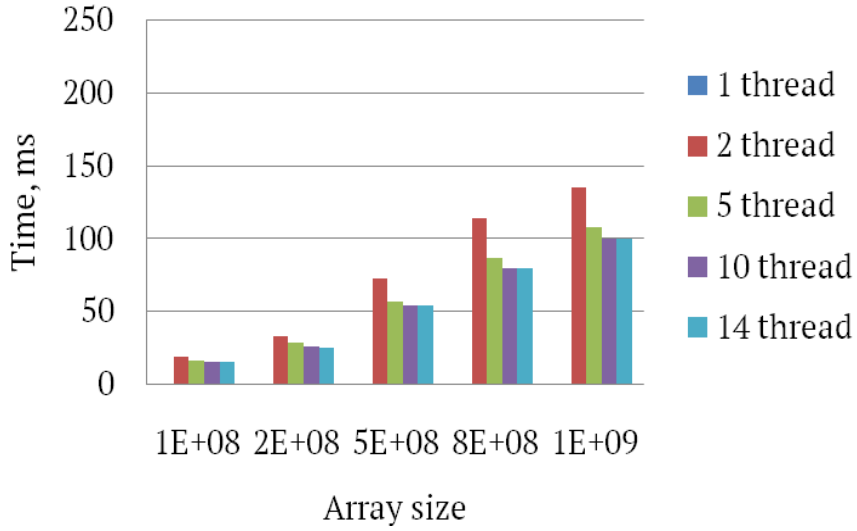

Figure 1 – Dependence of the computation time of the multithreaded program on the size of the output array (without blocking the critical area)

In Fig. 2, the results of the implementation of the described program are presented, addressing the thread race problem by blocking the critical area with a mutex, i.e., obtaining correct calculation results.

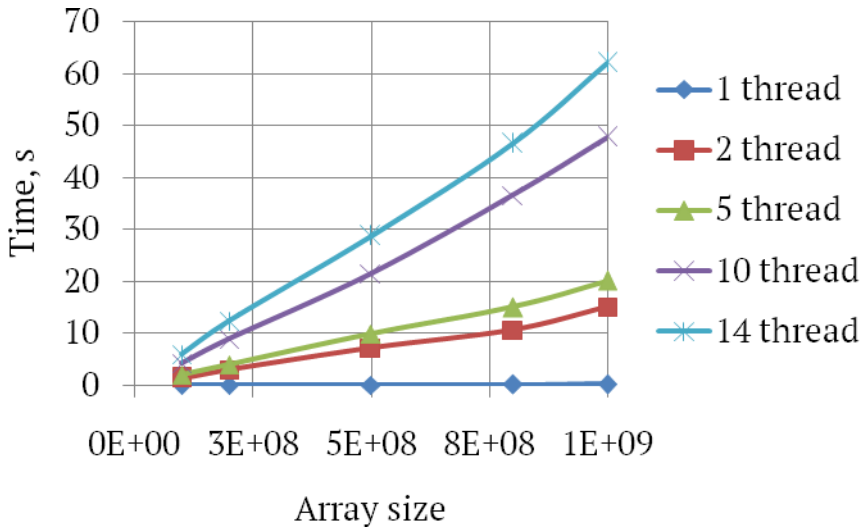

Figure 2 – Dependence of the computation time of the multithreaded program on the size of the output array (with blocking the critical area)

As can be observed, when solving the thread race problem in the considered task, with an increase in the array size in the range of 10^8–10^9, the program execution time increases approximately tenfold. Increasing the number of used threads for parallel calculations with mutex capture on each iteration of the sum accumulation

loop also leads to a noticeable slowdown in program execution. For example, when increasing the number of threads from 2 to 14, the program execution time increases approximately four times, regardless of the output array size.

Each time a thread captures a mutex, other competing threads must wait until the current «owner» of the mutex releases it. The more threads there are, the more situations can arise where threads wait for access to the mutex, affecting the overall program execution time. The more cores in the processor, the more intense the competition for access to the mutex becomes. Additionally, when threads compete for processor time, scheduling operations can cause delays due to context switching between threads.

To improve performance with many threads, it is desirable to consider alternative methods in the future, such as using atomic operations, etc.

Also, promising directions are considered to be the use of special tools, for example, ROMP [8], DataRaceBench [9], and new approaches to detecting thread races based on deep neural network models [10].

Conclusions. As a result of the work, a multithreaded algorithm was developed to implement a typical task of processing extremely large data arrays with protection of the critical area to prevent the thread race problem using mutexes; the performance of the developed algorithm was investigated with a significant amount of processed data (10^8–10^9 elements) and a variable (1–14) number of computing threads; a concept was developed for the further application of effective approaches to data protection in concurrent programs implemented on multiprocessor and multi-core systems.

It was established that solving the thread race problem in the considered task on a modern PC with an Intel Core i7-12700H processor, with an increase in the array size in the range mentioned above, slows down the program execution approximately tenfold. Increasing the number of used threads from 2 to 14 during this slows down the application implementation approximately four times, regardless of the amount of processed data.

REFERENCES

1. Almeida S. An Introduction to High Performance Computing / S. Almeida // International Journal of Modern Physics A. – 2013. – vol. 28, no. 22n23, 1340021, p. 1–9. https://doi: 10.1142/s0217751x13400216

2. Tanenbaum A. S. Modern Operating Systems / A. S. Tanenbaum, H. Bos. – New Jersey : Prentice Hall Press, 2014. – 1136 p. ISBN 978-0-13-359162-0

3. Williams A. C++ Concurrency in Action: Practical Multithreading / A.°Williams. –

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Island : Manning Shelter, 2019. – 592 p. ISBN 9781617294693
4. Kim S. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework / S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, T. Kim // Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP). – 2019. – p. 147–161. https://doi.org/10.1145/3341301.3359662
5. Xu W. Fuzzing File Systems Via Two-dimensional Input Space Exploration / W. Xu, H. Moon, S. Kashyap, P-N. Tseng, T. Kim // 2019 IEEE Symposium on Security and Privacy (SP). – 2019. – p. 818–834. https://doi.org/10.1109/SP.2019.00035
6. Xu M. Krace: Data Race Fuzzing for Kernel File Systems / M. Xu, S. Kashyap, H. Zhao; T. Kim. // 2020 IEEE Symposium on Security and Privacy (SP). – 2020. – p. 1643–1660. https://doi.org/10.1109/SP40000.2020.00078
7. Zhulkovskyi O. O. Evaluation of the Efficiency of the Implementation of Parallel Computational Algorithms Using the <thread> Library in C++ / O.°O.°Zhulkovskyi, I. I. Zhulkovska, V. V. Shevchenko, H. Ya. Vokhmianin // Computer Systems and Information Technologies. – 2022. – № 3. – p. 49–55. https://doi.org/10.31891/csit-2022-3-6
8. Gu Y. Dynamic Data Race Detection for OpenMP Programs / Y. Gu, J. Mellor-Crummey // SC18: International Conference for High Performance Computing, Networking, Storage and Analysis. – 2018. – p. 767–778.
https://doi.org/10.1109/SC.2018.00064
9. Verma G. Enhancing DataRaceBench for Evaluating Data Race Detection Tools / G. Verma, Y. Shi, C. Liao, B. Chapman, Y. Yan // 2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness). – 2020. – p. 20–30.
https://doi.org/10.1109/Correctness51934.2020.00008
10. TehraniJamsaz A. DeepRace: A Learning-based Data Race Detector / A. TehraniJamsaz, M. Khaleel, R. Akbari, A. Jannesari // 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). – 2021. – p. 226–233. https://doi.org/ 10.1109/ICSTW52544.2021.00046

Received 01.10.2023.
Accepted 10.10.2023.

Дослідження проблем синхронізації та захисту даних при реалізації багатопоточних програм

Нагальні проблеми сучасного розвитку процесів та технологій потребують постійного підвищення продуктивності комп'ютерної техніки, ефективного використання її ресурсів, обробки великих обсягів даних та підтримки зростаючих вимог сучасних інформаційних систем. Одним із напрямків, що забезпечують ви-

рішення вказаних проблем, є використання багатопроцесорних та обчислювальних систем із багатоядерними процесорами, що забезпечують швидке та продуктивніше паралельне вирішення окремих ресурсномістких завдань.

У зв'язку з цим актуальними задачами є дослідження методів синхронізації та управління ресурсами в багатопоточних середовищах, розроблення нових методів та інструментів для паралельного програмування зі створенням ефективних багатопоточних крос-платформних програм, оптимізація алгоритмів та архітектур для масштабованих обчислень.

Метою даної роботи є вирішення майже важливішої серед проблем багатопоточного програмування – проблеми гонки потоків при багатопоточних обчисленнях ресурсномістких задач із паралельним доступом до спільних даних за допомогою використання м'ютексів.

В роботі вирішені наступні завдання: розроблено багатопоточний алгоритм реалізації типової задачі обробки надвеликих масивів даних із захистом критичної області за допомогою примітивів синхронізації – м'ютексів; досліджено продуктивність виконання розробленого алгоритму при значній (10^8–10^9 елементів) кількості оброблюваних даних і змінному (1–14) числі обчислювальних потоків (ядер); вироблено концепцію для подальшого застосування ефективних підходів до захисту даних у конкурентних програмах, що реалізуються на багатопроцесорних та багатоядерних системах.

Встановлено, що вирішення проблеми гонки потоків у розглянутій задачі на сучасному PC із процесором Intel Core i7-12700H (14 cores, 2.3 GHz / 4.7 GHz) при збільшенні розміру масиву у досліджуваному діапазоні уповільнює виконання програми приблизно в 10 разів. Збільшення при цьому числа використаних потоків від 2 до 14 уповільнює реалізацію застосунку приблизно в 4 рази, незалежно від кількості оброблюваних даних.

Жульковський Олег Олександрович – доцент кафедри програмного забезпечення систем, Дніпровський державний технічний університет.

Жульковська Інна Іванівна – доцент кафедри кібербезпеки та інформаційних технологій, Університет митної справи та фінансів.

Костенко Вікторія Вікторівна – старший викладач кафедри комп'ютерних наук та інженерії програмного забезпечення, Університет митної справи та фінансів.

Булгакова Ольга Федорівна – ст. викладач кафедри комп'ютерних наук та інженерії програмного забезпечення, Університет митної справи та фінансів.

ISSN 1562-9945 (Print)
ISSN 2707-7977 (Online)

Zhulkovskyi Oleg – Associate Professor of Department of Software Systems, Dniprovsky State Technical University.

Zhulkovska Inna – Associate Professor of Department of Cybersecurity and Information Technologies, University of Customs and Finance.

Kostenko Victoria – Senior Lecturer of Department of Computer Science and Software Engineering, University of Customs and Finance.

Bulhakova Olha – Senior Lecturer of Department of Computer Science and Software Engineering, University of Customs and Finance.