Scientific Journal of Riga Technical University
Computer Science. Applied Computer Systems
2011
Volume 47

Blocking and Non-Blocking Process Synchronization: Analysis of Implementation

Vladislav Nazaruk[1], Pavel Rusakov[2], [1-2]*Riga Technical University*

***Abstract* —In computer programs with multiple processes, inter-process communication is of high importance. One of its main aspects is process synchronization, which can be divided into two classes: blocking and non-blocking. Blocking synchronization is simpler and mostly used; however, non-blocking synchronization allows avoiding some negative effects. In this paper, there is discussed the logic of widespread process synchronization mechanisms and is analyzed the support of these mechanisms in different platforms.**

***Keywords* — concurrent computing, process synchronization, blocking and non-blocking algorithms**

I. Introduction

Nowadays the use of parallel and concurrent computing rapidly increases. As a stimulating factor, the development and expansion of computers with multi-core processors (as well as multi-processor systems) can be mentioned. In such computers, at any time several processes (threads) can be executed. Parallel computing implies that one computing task (or its part) is divided into several equal subtasks, which will be executed simultaneously by different processors or processor cores. Therefore, the goal of parallel computing is mainly to decrease the computation time. Concurrent computing also implies separation of a computing task into several threads, but with the aim of structuring a program in an appropriate way for the task, in order to dynamically reallocate processor time for other threads when one thread is idle.

Both parallel and concurrent computing are closely tied together; and they both can be used to speed up computations. In both parallel and concurrent computing there is a need for inter-process communication (IPC) — a way how different threads of one process can exchange data with one another. There exist different techniques for inter-process communication, for example message passing, synchronization, shared memory, remote procedure calls (RPC). Synchronization techniques by themselves form a large class of mechanisms; and at a first approximation they can be divided into two classes: data synchronization mechanisms and process synchronization mechanisms. The aim of *data synchronization* is to assure that all copies of specific data are coherent (up-to-date); the aim of *process synchronization* is to assure specific coherence of execution of actions between several processes (or threads).

This paper discusses algorithms used for synchronizing processes. The *goal* of this paper is to analyze exactly *process* synchronization mechanisms in terms of their logic of operation and their support in modern software and hardware platforms. The *tasks* of the research are the following:

- to analyze the logic of operation of widespread process synchronization mechanisms,
- to analyze the support of widely used process synchronization mechanisms in different modern object-oriented programming languages, software and hardware platforms,
- to compare blocking and non-blocking process synchronization.

II. Related works

Process synchronization is common to all operating systems. Therefore, many books dedicated to operating systems, for example [1], describe basic principles of process synchronization and synchronization mechanisms commonly used in operating systems. These works describe mostly blocking synchronization mechanisms. In [2], another class of process synchronization mechanisms — non-blocking synchronization, is described and studied.

Before trying to implement a specific synchronization mechanism on a new software or hardware platform, the platform should be examined to discover whether it in principle supports all instructions or primitives needed to implement the mechanism. It is true that all commonly used synchronization mechanisms nowadays can be implemented on a hardware platform which includes any modern central processing unit (CPU). However, when speaking about a graphic processing unit (GPU) as such hardware platform, as it is a relatively new platform, the possibility of implementing synchronization mechanisms should be still considered. The paper [3] shows that the most demanding class of synchronization mechanisms (wait-free non-blocking synchronization) could be implemented on GPUs even without the need of strong synchronization primitives in hardware. In contrast, in the corresponding chapter of this paper, possible support of such hardware primitives by GPUs is discussed.

III. Blocking process synchronization

Many different process synchronization mechanisms exist. Two base classes of these mechanisms are blocking *and* non-blocking synchronization mechanisms. (The differences between these two classes are described further in this paper.) *Blocking synchronization mechanisms* or *locks* are most commonly used; and most widespread blocking synchronization mechanisms are the following: semaphores, monitors, condition variables, events, barriers, rendezvous, etc. These synchronization mechanisms are considered next in this chapter.

A. Overview of blocking synchronization

The aim of *locks* is to control access to a specific shared resource (for example, to a shared memory area) in such an abstract way that when a process *holds* a lock, it can operate with the shared resource to such a degree that is allowed by a lock. For example, when a process holds a lock which grants full access to a shared resource, the process has a right to do anything with the resource. On the other hand, if a process holds a lock which grants read-only access to a shared resource, the process has a right only to read the resource. As to a process that does not hold a lock, it is forbidden for it to do any operations with the corresponding shared resource.

Sharing a resource (and thus synchronizing process access to it) with locks can cause several undesirable situations such as [4]:

1) Insufficient throughput in case of a heavily contended lock — when a lock is trying to be acquired by processes frequently while it is held by some other process. In this case much of processor resource is spent in vain.
2) If a process holding a lock is paused, no other process can take this lock, and therefore all other processes requiring the lock cannot make any progress. A situation when a higher-priority process is waiting for a lock held by a lower-priority process for a long period of time is called a *priority inversion.*
3) When there exists a set of processes where each process is waiting for a lock held by another process in this set, a *deadlock* occurs. In this situation execution of all processes from the set is indefinitely postponed — and without special actions, there will never be a progress of any process of the set.

B. Classification of locks

Locks can be classified in several ways, including the following:

1) by possible mode of access to a shared resource:
   a) *semaphores* (binary semaphores, mutexes, critical sections *and* counting semaphores) — it is supposed that all threads have an exclusive (read/write) access to a resource [1],
   b) *readers-writers locks* — threads are differentiated by their access level to a resource: read-only or read-write [1];
2) by necessity to call a lock explicitly when accessing a shared resource [5]:
   a) *advisory locks* (most common) — threads should call the lock before accessing a resource,
   b) *mandatory locks* — when thread tries to access a shared resource, the system itself checks a possibility of accessing the resource;
3) by a thread waiting technique:
   a) *sleep locks* (most common) — a process is blocked when it attempts to access a busy resource,
   b) *spin locks* — after an attempt to access a busy resource, a process is notified that the resource is busy, and is not blocked; then it usually attempts to access the resource over again [1].

The aim of *monitors*[1]is similar to the aim of locks; however, monitors are on a more abstract level than locks — monitor is an object which by itself guarantees that its methods are always executed with mutual exclusions (i. e., at the same time only one method of this object can be executed). Although for internal implementation of monitors, binary semaphores are usually used [1], when using monitors, there is no need for programmer to understand the inner working of them.

Monitors can be extended in order to allow threads to access a shared resource only when a specific condition (provided by a corresponding thread) is met — in this case, they are called *condition variables*.

The aim of *events* is to suspend (block) all threads which explicitly asked for that, until a specific condition is met (this happens when another thread notifies the corresponding event object).

Barriers are synchronization mechanisms for a group of threads, when it is needed to block execution of each thread from the group at a specific location in the executable code ("barrier") of the corresponding thread. Just after all threads from the group are blocked ("reach a barrier"), barrier automatically unblocks all these threads — allowing them to resume working together at the same time.

Rendezvous are process synchronization mechanisms based on a client-server model for interaction: one thread (a server) declares a set of services (methods with specific entries) being offered to other threads (clients). To request a rendezvous, a calling thread must make an entry call on an entry of another thread. The calling thread becomes blocked waiting for the rendezvous to take place — i. e., until the called thread accepts this entry call. When the accepting thread ends the rendezvous, both threads are freed to continue their execution [6].

C. Support of blocking synchronization in imperative programming languages

Schematic results of comparison of some widespread imperative object-oriented programming languages (and, where applicable, software frameworks) by support of process synchronization mechanisms described above are given in Table 1. This table was created using scientific, pedagogical and practical experience of the authors of this paper. Additional analysis of specifications of programming languages and software platforms has been made too. For the comparison, five commonly used object-oriented programming languages for concurrent application were chosen: C++, Java, C#, Python and Ada.

TABLE 1

NATIVESUPPORT OF DIFFERENT BLOCKING SYNCHRONIZATION MECHANISMS

	C++	**Java** (Java Platform, Standard Edition)	**C#** (Microsoft .NET Framework)	**Python**	**Ada**
critical sections / binary semaphores	no built-in support for synchronization mechanisms; however, several parallel or concurrent computing APIs (for example OpenMP, POSIX Threads, MPI) can be used	yes: keyword *synchronized* for a code block	yes: keyword *lock* for a code block	yes: object factory *threading.Lock()*	manual: e. g. by using rendezvous
counting semaphores		yes: class *java.util.concurrent.Semaphore*	yes: class *System.Threading.Semaphore*	yes: object factory *threading.Semaphore()*	manual: by using protected objects or rendezvous
monitors		yes: keyword *synchronized* for specific methods	yes: class *System.Threading.Monitor*	manual: by using conditional variables	yes: protected objects (keyword *protected type*)
condition variables		yes: by calling a method *newCondition()* in an instance if interface *java.util.concurrent.locks.Lock*	—	yes: object factory *threading.Condition()*	—
events		—	yes: classes *System.Threading.AutoResetEvent*, and *.ManualResetEvent*	yes: object factory *threading.Event()*	—
barriers		yes: class *java.util.concurrent.CyclicBarrier*	yes: class *System.Threading.Barrier*	—	—
rendezvous		yes: class *java.util.concurrent.Exchanger*	—	—	yes: keywords *entry/accept*

In the C++ programming language, there is no native support for synchronization mechanisms; however, to support them, several libraries for parallel or concurrent computing can be used.

In Java SE (Java Platform, Standard Edition), there is a standard package *java.util.concurrent* which has utility classes commonly used in concurrent programming. These classes include those to implement some common synchronization mechanisms, for example class *Semaphore* to implement counting semaphores, class *Lock* from package *locks* to implement more generalized locks, class *CyclicBarrier* to implement barriers, and class *Exchanger* which can be used to implement the mechanism similar to rendezvous. In the Java programming language, there is a special keyword *synchronized* which, when applied to a statement block or a method, provides a critical section for them.

Similarly to Java SE, the Microsoft .NET Framework has the *System.Threading* namespace, which provides classes and interfaces for multithreaded programming. This includes classes for synchronizing thread activities, for example *Semaphore*, *Monitor*, *Barrier* for synchronization mechanisms of the same name, and *AutoResetEvent* and *ManualResetEvent* for events. Similarly to the *synchronized* keyword in Java, the *lock* keyword in C# marks a statement block as a critical section.

In Python, the *threading* module provides high-level threading interfaces, including those for process synchronization, for example, object factories *Lock()*, *Semaphore()*, *Condition()*, *Event()*.

The Ada programming language natively supports only two types of synchronization mechanisms: rendezvous (by using keywords *entry* and *accept*) and protected objects or monitors (by using keyword *protected type*).

As one can see, every programming language compared has either built-in process synchronization mechanisms, or they can be added by using corresponding software libraries (for C++). Even if in the comparison table it is stated that a specific synchronization mechanism is not natively supported by a specific programming language, there may exist two possible solutions: support of the needed synchronization mechanism to the language could be added by using other software libraries, or the needed process synchronization could be implemented via other (supported) process synchronization mechanisms.

IV. NON-BLOCKING PROCESS SYNCHRONIZATION

The second base class of process synchronization mechanisms (along with blocking synchronization) is *non-blocking process synchronization*; they are discussed in this chapter.

As it was stated in Chapter III, use of blocking synchronization can cause some undesirable results — when using locks, a deadlock (a situation when all processes become blocked, waiting for one another) may appear. In other words, use of blocking process synchronization can cause the entire system to be blocked indefinitely. In contrast to such synchronization mechanisms, there is a class of non-blocking synchronization mechanisms. Their use ensures that a deadlock will never ap-

pear [7]: a system-wide progress is guaranteed at any time. Non-blocking synchronization algorithms are designed in such a way that they do not require a critical section; in their implementation, specific atomic operations are used (for example, "read-modify-write").

A. Overview of non-blocking synchronization

Two degrees of non-blocking synchronization algorithms exist [7]:

- *wait-free algorithms*, which guarantee progress for each thread, and therefore eliminate possibility of deadlocks,
- *lock-free algorithms*, which guarantee system-wide progress; however, for different threads there is a possibility of starvation (including even permanent starvation).

The third degree of non-blocking synchronization algorithms is sometimes considered: *obstruction-free* algorithms, which guarantee per-process progress if a process becomes isolated (i. e., in case all other processes in the system are suspended).

These three degrees of non-blocking algorithms form a hierarchy:

- every wait-free algorithm is a lock-free algorithm,
- and every lock-free algorithm is an obstruction-free algorithm.

Non-blocking synchronization algorithms have the following advantages over blocking synchronization algorithms [4]:

- deadlocks and priority inversions are impossible by design,
- contention for a shared resource is less expensive,
- lesser idle time, which can have positive influence on execution speed,
- coordination occurs at a finer level of granularity, enabling a higher degree of parallelism.

However, non-blocking synchronization algorithms are more difficult to implement than blocking synchronization algorithms [4]. Although non-blocking algorithms have been discovered for many common data structures, up to now, efficient wait-free algorithms (which are more valuable than lock-free algorithms) have not been developed [7]. However, there exist some working and acceptable lock-free synchronization algorithms.

Regardless of these difficulties, non-blocking synchronization is extensively used in operating systems and software frameworks and/or virtual machines (Java, Microsoft .NET Framework). It is used for tasks such as thread and process scheduling [4]; in JVM (Java Virtual Machine), it is used in garbage collection or to accelerate concurrent and parallel garbage collection [8].

B. Implementation of non-blocking synchronization

In order to implement most of non-blocking algorithms, there is a need for atomic read-modify-write primitives. They can be provided either by hardware, or by an underlying software framework (such as Java SE). There exist several such atomic primitives, including:

- compare-and-swap,
- test-and-set,
- fetch-and-add,
- load-link *and* store-conditional.

The main idea with such atomic primitives is that both actions should be completed together (exactly one after another; no other instruction may be executed between them).

The compare-and-swap (CAS) primitive is the most common one. In 1991, Maurice Herlihy proved [9] that with the use of only the CAS atomic primitive and other primitive operations, assuming a sufficient amount of memory, one can implement any lock-free or wait-free algorithm.

The CAS primitive is provided by most widespread software frameworks:

- In the Microsoft .NET Framework (languages C#, VB.NET, managed C++, F#), there is a class *Interlocked* in module *System.Threading*, which provides atomic operations for variables that are shared by multiple threads [10].
- In the Java SE platform, there is a package *java.util.concurrent.atomic*, which contains classes that support lock-free thread-safe programming on single variables. For example, there are classes *AtomicInteger* and *AtomicLong*, which provide the following atomic operations: add-and-get, compare-and-set, get-and-set, etc. [11].
- For the C++ programming language, there exist several libraries which provide lock-free algorithm implementation, including the CAS primitive (see [12]).

This means that any non-blocking algorithm (either lock-free or wait-free) can be implemented with the use of any widespread programming language or software framework.

For example, on the Java SE platform, non-blocking counter could be implemented in the following way on top of the variable *AtomicInteger*, which provides a needed atomic operation (*compareAndSet*) [8]:

```
public class NonblockingCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        }
         while (!value.compareAndSet(v, v + 1));
        return v + 1;
    }
}
```

C. Hardware support of non-blocking synchronization

Talking about hardware support for atomic read-modify-write instructions, it is important to determine hardware platforms that should be analyzed.

It should be noted that many modern computers have video cards that can also be used for arbitrary (even not related with

graphics) calculations. Such technique of using graphic processing units (GPUs) for general-purpose calculations is called *general-purpose computing on graphics processing units (GPGPU)*. GPU developers provide free GPU programming libraries (or SDKs), e. g. OpenCL (Open Computing Language), CUDA by Nvidia, Stream SDK by AMD.

Thus, nowadays (in contrast to a period of several years before) most processing systems belong to one of the following two classes:

— central processing units (CPUs),
— graphic processing units (GPUs).

Considering hardware support for atomic read-modify-write primitives in CPUs, such CPU instructions can be mentioned:

— CMPXCHG ("Compare and Exchange") instruction, as well as
— CMPXCHG8B /CMPXCHG16B instructions ("Compare and Exchange Bytes")

for the Intel 64 and IA-32 CPU architectures [13], and for the AMD64 architecture [14], which by themselves are the implementation of the CAS primitive.

Thus, any modern CPU supports implementation of non-blocking algorithms. However, for GPUs, the process synchronization has not been used widely yet.

Studying the Instruction Set Architecture book for AMD Evergreen Technology GPUs [15], an instruction GWS_SEMA_P ("Global Wavefront Sync Semaphore P") is found — it belongs to Control Flow (CF) instructions and performs an atomic semaphore "P" operation on the resource. The further description of this instruction states: "Execution of this instruction is stalled until the semaphore has a positive value. The semaphore value is then decremented by one before execution can continue". The instruction GWS_SEMA_Pcomes in pair with the instruction GWS_SEMA_V ("Global Wavefront Sync Semaphore V"), which performs an atomic semaphore "V" operation on the resource.

In practice this means that using these two instructions (GWS_SEMA_P and GWS_SEMA_V), it is easy to implement a critical section: in the beginning, the semaphore should be set to be equal 1, then, when entering a critical section, the operation "P" should be called, and after the critical section, the operation "V" should be called. If in the critical section there are several operations, for example compare and swap, then an atomicity of such operations will be achieved. Therefore, by using the GWS_SEMA_P and GWS_SEMA_V instructions, it is possible to implement any atomic primitive, including read-modify-write primitives on GPUs.

This means that non-blocking synchronization is possible not only in programs running on CPUs, but also in programs running on GPU platforms. Furthermore, GPUs have a SIMD architecture by Flynn's taxonomy [16], and are especially intended for parallel calculations. Here, non-blocking synchronization algorithms, which provide a high degree of parallelism, could be especially suitable.

It is also important to get to know how performance of non-blocking algorithms depends on certain factors. In [8], it is stated: "Under light to moderate contention, non-blocking algorithms tend to outperform blocking ones because most of the time the CAS succeeds on the first try, and the penalty for contention when it does occur does not involve thread suspension and context switching, just a few more iterations of the loop. An uncontended CAS is less expensive than an uncontended lock acquisition (this statement has to be true because an uncontended lock acquisition involves a CAS plus additional processing), and a contended CAS involves a shorter delay than a contended lock acquisition."

However, when the contention is high, blocking algorithms could be better than non-blocking. This is mainly because when a process wants to get a lock, and it is unavailable, the process becomes blocked, and in such a way it does not make an impact on high contention any more [8].

V. Conclusions and Further Work

Process synchronization is an integral part of concurrent computing — its aim is to assure a specific coherence of execution of actions between several threads (or processes).

There exist many different blocking process synchronization mechanisms. Most of such widely used mechanisms are supported in common programming languages. Also, many blocking process synchronization mechanisms can be implemented using other basic synchronization mechanisms.

In contrast to blocking mechanisms, the use of non-blocking mechanisms allows avoiding deadlocks in a system. In many cases, non-blocking mechanisms can provide an alternative to blocking mechanisms.

However, blocking mechanisms have their own characteristics that can limit their application (for example, due to some of their specific architectural and implementation properties) or, on the other hand, that make them more appropriate than blocking ones for some situations (for example, due to prevention of deadlocks, as well as their lesser idle time and good applicability in parallel computing). A deeper analysis of such characteristics of common non-blocking synchronization algorithms is one of the goals of further research.

Despite being the most demanding class of synchronization mechanisms from the point of view of their implementation, non-blocking synchronization can be supported not only on central processing units, but also on graphics processing units. Implementing non-blocking algorithms on GPUs could be another direction for further research.

Also, the current use of synchronization mechanisms, as well as possibilities of applying them in practice efficiently in different situations could be analyzed.

A thorough analysis of performance of blocking and non-blocking algorithms could also be considered as a further research topic. This may help develop some guidelines for choosing a better process synchronization algorithm for a specific situation. Development of methods for analyzing such performance for different situations can be considered as a subtask of this research direction.

Acknowledgements

This work has been supported by the European Social Fund within the project "Support for the implementation of doctoral studies at Riga Technical University".

REFERENCES

[1] A. Silberschatz, P. B. Galvin, G. Gagne, "Operating System Concepts", 7th ed.: John Wiley & Sons, Inc., 2005.
[2] H. Sundell. "Efficient and Practical Non-Blocking Data Structures". Thesis for the degree of Doctor of Philosophy. Chalmers University of Technology and G?teborg University, 2004, pp. 240.
[3] P. H. Ha, P. Tsigas, O. J. Anshus, "Wait-free Programming for General Purpose Computations on Graphics Processors". Proceedings of the 22th International Parallel and Distributed Symposium (IPDPS 2008), 14–18 April 2008, pp. 1–12.
[4] "Java theory and practice: Going atomic". [Online]. Available: https://www.ibm.com/developerworks/java/library/j-jtp11234/. [Accessed: Oct. 11, 2010].
[5] "Programming Interfaces Guide", Sun Microsystems, Inc. [Online]. Available: http://dlc.sun.com/pdf/817-4415/817-4415.pdf. [Accessed: Oct. 11, 2010].
[6] J. Miranda, "A Detailed Description of the GNU Ada Run Time". [Online]. Available: http://www.iuma.ulpgc.es/users/jmiranda/gnat-rts/main.htm [Accessed: Oct. 11, 2010].
[7] K. Chirls, "A Comparison of Blocking & Non-Blocking Synchronization". [Online]. Available: http://dspace.nitle.org/bitstream/handle/10090/801/s10csi2007chirls.pdf?sequence=1. [Accessed: Oct. 11, 2010].
[8] "Java theory and practice: Introduction to nonblocking algorithms". [Online]. Available: http://www.ibm.com/developerworks/java/library/j-jtp04186/index.html. [Accessed: Oct. 11, 2010].
[9] M. Herlihy. "Wait-Free Synchronization", Digital Equipment Corporation. [Online]. Available: http://www.cs.brown.edu/~mph/Herlihy91/p124-herlihy.pdf. [Accessed: Oct. 11, 2010].
[10] "Interlocked Class (System.Threading)". [Online]. Available: http://msdn.microsoft.com/en-us/library/5kczs5b5.aspx. [Accessed: Oct. 11, 2010].
[11] "java.util.concurrent.atomic (Java Platform SE 6)". [Online]. Available: http://download.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html. [Accessed: Oct. 11, 2010].
[12] "Computer Laboratory. Practical lock-free data structures". [Online]. Available: http://www.cl.cam.ac.uk/research/srg/netos/lock-free/. [Accessed: Oct. 11, 2010].
[13] "Intel® 64 and IA-32 Architectures. Software Developer's Manual. Volume 2A: Instruction Set Reference, A-M". [Online]. Available: http://www3.intel.com/Assets/PDF/manual/253666.pdf. [Accessed: Oct. 11, 2010].
[14] "AMD64 Technology. AMD64 Architecture. Programmer's Manual. Volume 3: General-Purpose and System Instructions". [Online]. Available: http://support.amd.com/us/Processor_TechDocs/24594.pdf. [Accessed: Oct. 11, 2010].
[15] "Evergreen Family Instruction Set Architecture. Instructions and Microcode. Reference Guide", Advanced Micro Devices, Inc., September 2010. [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/assets/AMD_Evergreen-Family_Instruction_Set_Architecture.pdf. [Accessed: Oct. 11, 2010].
[16] M. Flynn, "Some Computer Organizations and Their Effectiveness". IEEE Trans. Comput., Vol. C-21, 1972, pp. 948.

Vladislav Nazaruk, born in 1986. Gained a degree of Bachelor of Engineering Science (Bc. sc. ing.) in Computer Control and Computer Science in 2007, a degree of Master of Engineering science (Mg. sc. ing.) in Computer Systems in 2009, both at Riga Technical University (RTU), Latvia. Now is a doctoral student at Institute of Applied Computer Systems, RTU.

Works as an Information Systems Engineer at Department of Applied Computer Science, RTU.

Fields of research interests: computer science, mathematics, pedagogy. Special interests: algorithms, information theory, protection of information.

Email: Vladislavs.Nazaruks@rtu.lv. Address for correspondence: *Lietišķo datorzinātņu katedra, Rīgas Tehniskā universitāte, Meža iela 1 k-3 — 506, Rīga, LV-1007, Latvia*.

Pavel Rusakov was born in Riga, Latvia in 1972. Dr.sc.ing. (1998), Mg. sc. ing. (1995),Bc. sc. ing. (1993) — Riga Technical University (RTU).

Associated professor at RTU, Institute of Applied Computer Systems. Head of a laboratory, responsible for the Professional Bachelor and Professional Master studies at the Department of Applied Computer Science. Field of interests: computer science. Special interests: programming paradigms, object-oriented approach to systems development, parallel computing, Web technologies, distributed systems, computer graphics, and protection of information.

Diploma with distinction: Mg. sc. ing.

Email: Pavels.Rusakovs@cs.rtu.lv. Address for correspondence: *Lietišķo datorzinātņu katedra, Rīgas Tehniskā universitāte, Meža iela 1 k-3 — 506, Rīga, LV-1007, Latvia*.

Vladislavs Nazaruks, Pāvels Rusakovs. Bloķējošā un nebloķējošā procesu sinhronizācija: implementēšanas analīze
Mūsdienās laiksakritīgās skaitļošanas izmantošana strauji palielinās. Šāda skaitļošana paredz skaitļošanas uzdevumu sadalīšanu vairākos procesos (vai pavedienos). Datorprogrammās ar vairākiem procesiem starpprocesu komunikācijai ir liela nozīme. Viena no būtiskiem starpprocesu komunikācijas mehānismu klasēm ir procesu sinhronizācijas algoritmi. Piemēram, var runāt par semaforiem, nosacījuma mainīgajiem, satikšanām. Procesu sinhronizācijas algoritmi var tikt iedalīti divās klasēs: bloķējošie un nebloķējošie algoritmi; turklāt, katrai no šīm klasēm ir savas priekšrocības un savi trūkumi. Tā, atšķirībā no bloķējošās sinhronizācijas, nebloķējošās sinhronizācijas algoritmu lietošana garantē to, ka sistēmā nekad neparādīsies strupsaķere — sistēmas progress tiek garantēts katra laika momentā. Daudzās modernajās programmēšanas valodās un programmatūras platformās eksistē daži iebūvēti procesu sinhronizācijas mehānismi. Tādējādi, izvēlētā programmēšanas valoda vai platforma noteiktā mērā ietekmē sinhronizācijas mehānismu (kā arī citu starpprocesu komunikācijas mehānismu) izmantošanu. Viens no šī raksta uzdevumiem ir procesu sinhronizācijas mehānismu klašu analīze un salīdzināšana. Otrs uzdevums ir analizēt procesu sinhronizācijas algoritmu atbalstu dažādās modernajās objektorientētajās programmēšanas valodās (C#, Java, C++u. c.) un aparatūras platformās. Pēdējais attiecas uz nebloķējošās sinhronizācijas algoritmiem. Rakstā arī tiek norakstūrotas grafisko procesoru, kas atbalsta universālskaitļošanas tehnoloģiju GPGPU, iespējas procesu sinhronizācijas kontekstā.

Владислав Назарук, Павел Русаков. Блокирующая и неблокирующая синхронизация процессов: анализ реализации
В наши дни стремительно возрастает использование параллельных вычислений.Это подразумевает разделение вычислительного задания на несколько процессов (или потоков). Важную роль в компьютерных программах с несколькими процессами играет коммуникация между соответствующими структурными единицами. Один из основных классов механизмов межпроцессного взаимодействия — алгоритмы синхронизации процессов. Например, можно говорить о семафорах, условных переменных, рандеву. Алгоритмы синхронизации процессов могут подразделяться на два класса: блокирующие и неблокирующие алгоритмы;при этом каждый из этих классов имеет свои преимущества и недостатки. Так, в противоположность блокирующей синхронизации, использование алгоритмов неблокирующей синхронизации обеспечивает отсутствие тупиковых ситуаций — т. е., в любое время гарантирован прогресс на уровне системы. Во многих современных языках программирования и программных платформах существуют некоторые встроенные механизмы синхронизации процессов.Следовательно, выбранный язык программирования или платформа в некоторой степени влияет на использование механизмов синхронизации (также как и других механизмов межпроцессной коммуникации). Одной из задач данной статьи является анализ и сравнение классов механизмов синхронизации процессов. Другая задача заключается в анализе поддержки алгоритмов синхронизации процессов в различных современных объектно-ориентированных языках программирования (C#, Java, C++ и др.) и аппаратных платформах. Последнее относится к неблокирующим алгоритмам синхронизации. В статье также приводится характеристика возможностей графических процессоров, поддерживающих технологию произвольных вычислений GPGPU, в контексте синхронизации процессов.