Synchronization Possibilities and Features in Java

Beqir Hamidi[1]*

Lindita Hamidi[1]

[1]ILIRIA College Bregu i Diellit, st. Gazmend Zajmi, University of Pristina, Faculty of Mechanical Engineering. Bregu, Kosovo

*Email: beqirhamidi@hotmail.com

DOI: 10.2478/ejfe-2023-0019

Abstract

In this paper we have discussed one of the greatest features of the general-purpose computer programming language –Java. This paper represents concepts of Synchronization possibilities and features in Java. Today's operating systems support concept of "Multitasking". Multitasking achieved by executing more than one task at a same time. Tasks runs on threads. Multitasking runs more than one task at a same time. Multitasking which means doing many things at the same time is one of the most fundamental concepts in computer engineering and computer science because the processor execute given tasks in parallel so it makes me think that are executing simultaneously. Multitasking is related to other fundamental concepts like processes and threads. A process is a computer program that is executing in a processor, while a thread is a part of a process that has a way of execution: it is a thread of execution. Every process has at least one thread of execution. There are two types of multitasking: process – based and thread – based. Process-based multitasking, means that on a given computer there can be more than one program or process that is executing, while thread-based multitasking, which is also known as multithreading, means that within a process, there can be more than one thread of execution, each of them doing a job and so accomplishing the job of their process. When there are many processes or many threads within processes, they may have to cooperate with each other or concurrently try to get access to some shared computer resources like: processor, memory and input/output devices. They may have to, for example: print a file in a printer or write and/or read to the same file. We need a way of setting an order, where processes and/or threads could do their jobs (user jobs) without any problem, we need to synchronize them. Java has built-in support for process and thread synchronization, there are some constructs that we can use when we need to do synchronization.This paper, a first phase discussed the concept of Parall Programming, threads, how to create a thread, using a thread, working with more than one thread. Second phase is about synchronization, what is in general and in the end we disscused the synchronization possibilities and feautures in Java.

Keywords: Thread, Process, Multithreading, Synchronization, Java language.

Intoduction

Many applications from scientific computing can benefit from object-oriented programming techniques, since they allow a flexible and modular program development. Often, these applications are computation-intensive, so the sequential execution time is quite large. Therefore it is profitable to use a parallel machine for the execution. But up to now, no parallel object-oriented programming language has been established as a standard.

Java is a popular language and has support for a parallel execution integrated into the language. Hence, it is interesting to investigate the usefulness of Java for executing scientific programs in parallel.

In the following sections we have covered in more detail the problem starting threads, how to create a thread, using a thread, working with more than one thread, synchronization , what is in general and in the end we disscused the synchronization possibilities and feautures in Java.

Processes and Threads

“In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads”. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. “Processing time for a single core is shared among processes and threads through an OS feature called time slicing. It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores “.

Thread and Process are two closely related term in multi-threading and main difference between Thread and Process in Java is that *Threads are part of process*, while one process can spawn multiple Threads.

To understand multithreading, the concepts *process* and *thread* must be understood. A *process* is a program in execution. A process may be divided into a number of independent units known as *threads*. A *thread* is a dispatchable unit of work. Threads are *light-weight* processes within a process . A process is a collection of one or more threads and associated system resources. The difference between a process and a thread is shown in Figure 1. A process may have a number of threads in it. A thread may be assumed as a subset of a process [1].

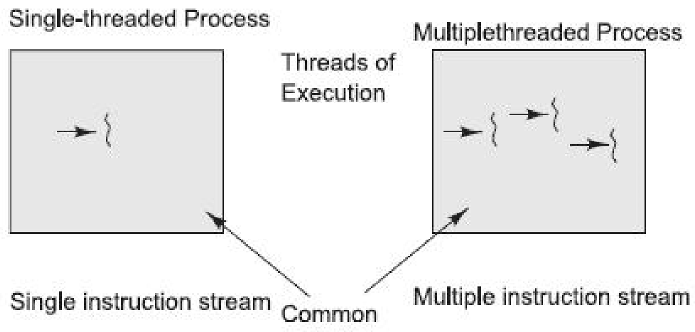

Figure 43 A process containing single and multiple threads.

Multitasking of two or more processes is known as *process-based multitasking*. Multitasking of two or more threads is known as *thread-based multitasking*. The concept of multithreading in a programming language refers to thread-based multitasking. Process-based multitasking is totally controlled by the operating system. But thread-based multitasking can be controlled by the programmer to some extent in a program

Thread creation: In contrast to most other programming languages where the operating system and a specific thread library like Pthreads [2] or C-Threads are responsible for the thread management, Java has a direct support for multithreading integrated in the language, see, e.g., [3]. The java.lang package contains a thread API consisting of the class Thread and the interface Runnable. There are two basic methods to create threads in Java.

Threads can be generated by specifying a new class which inherits from Thread and by overriding the run() method in the new class with the code that should be executed by the new thread. A new thread is then created by generating an object of the new class and calling its start() method.

An alternative way to generate threads is by using the interface Runnable which contains only the abstract method run(). The Thread class actually implements the Runnable interface and, thus, a class inheriting from Thread also implements the Runnable interface. The creation of a thread without inheriting from the Thread class consists of two steps: At first, a new class is specified which implements the Runnable interface and overrides the run() method with the code that should be executed by the thread. After that, an object of the new class is generated and is passed as an argument to the constructor method of the Thread class. The new thread is then started by calling the start() method of the Thread object. A thread is terminated if the last statement of the run() method has been executed. An alternative way is to call the interrupt() method of the corresponding Thread object.

Multithreading in Java: "A thread is an independent context of execution with in a process. A process can have more than one thread. All threads of a process share the resources allocated to the process. An operating system that supports this behavior is a multi-threaded operating system and a programming language that gives constructs/API to create and manage threads is called a multi-threaded programming language " [4]. Java has in built support to the multi-threading. Java is a *multithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs. Multitasking is when multiple processes share common processing.

Thread synchronization: The threads of one program have a common address space. Thus, access to the same data structures have to be protected by a synchronization mechanism. Java supports synchronization by implicitly assigning a lock to each object. The easiest way to keep a method thread-safe is to declare it synchronized. A thread must obtain the lock of an object before it can execute any of its synchronized methods. If a thread has locked an object, no other thread can execute any other synchronized method of this object at the same time. An alternative synchronization mechanism is provided by the wait() and notify() methods of the Object class which also supports a list of waiting threads. Since every object in the Java system inherits directly or indirectly from the Object class, it is also an Object and hence supports this mechanism. When a thread calls the wait() method of an object, it is added to the list of waiting threads for that object and stops running. When another thread calls the notify() method of the same object, one of the waiting threads is woken up and is allowed to continue running. These basic synchronization primitives can be used to realize more complex synchronization mechanisms like barriers, condition variables, semaphores, event synchronization, and monitors, see, e.g., [3].

Synchronization processes

Using many threads to running tasks has some advantages and disadvantages. We need to take a control when more than one thread share a same resource or same piece of data. Controlling access of threads in same resource done by technique called "Synchronization".

To use synchronization, specific data structures should be included in a system, and they should be manipulated by a set of functions.

To provide this functionality the Java has summarize synchronization variables in each and every object. The manipulating of there synchronized variables is done by the help of a thread.join(), synchronized keyword and other methods. These are included in all the libraries and they provide the coordinating of the instructions of threads. Shared data can be protected by means of synchronization variables.

Atomic Actions: Atomic operation means an operation that completes in its entirety without interruption [5].

One common source of bugs in concurrent programs is the failure to implement atomic actions correctly. An atomic action acts on the state of a program. The program's state contains a value for each variable defined in the program and other implicit variables, such as the program counter. An atomic action transforms the state of the program, and the state transformation is indivisible [6].

An action is atomic if the processes performing it:

- Are not aware of the existence of any other active process, and no other active process is aware of the activity of the processes during the time the processes are performing the action.
- Don't communicate with other processes while the action is being performed.
- Can detect no state change except those performed by themselves and it they don't reveal their state changes until the action is complete.
- Can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

First, an interface can be defined for a three-way atomic action, which is represented in Listing 3.

```
public interface ThreeWayAtomicAction
 {
   publlic void role1();
   public void role2();
   public void role3();
```

Listing 1 A fragment code of three-way atomic action.

In following is represented structure for three way atomic action which is show in Figure 2.

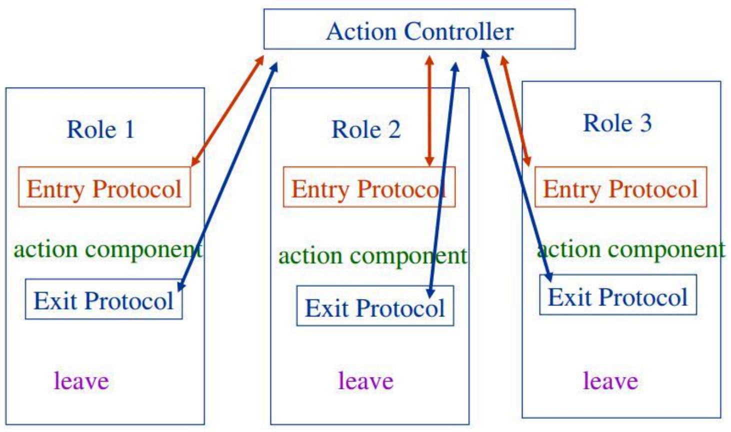

Figure 44 Structure of three role of action controller

The code listed below is based on the structure of which is shown in Figure 4, which declaration a declaration a public class Atomic Action Control.

```
public class AtomicActionControl implements
ThreeWayAtomicAction
 {
    protected Controller Control;
   public AtomicActionControl() // constructor
     {
      Control = new Controller();
     }
   class Controller
   {
      protected Bolean firstHere, secondHere, thirdHere;
      protected int allDone;
      protected int toExit;
      protected int
      numberOfParticipants;
     Controller()
     {
      firstHere = false;
      secondHere = false;
      thirdHere = false;
      allDone = 0;
      numberOfParticipants = 3;
      toExit = numberOfParticipants;
     }
    synchronized void first() throws InterruptedException
     {
       while(firstHere) wait();
       thirstHere = true;
     }
    synchronized void second() throws InterruptedException
     {
       while(secondHere) wait();
       secondHere = true;
     }
    synchronized void third() throws InterruptedException
     {
       while(thirdHere) wait();
       thirdHere = true;
    }
   }
  }
```

Listing 2 Declaration a public class for AtomicActionControl.

Fragment code in Listing 3 is the continuation of Listing 2, which synchronizes three action.

```
synchronized void first() throws InterruptedException

{
  while (firstHere)
  wait();
  firstHere = true;

}
synchronized void second() throws InterruptedException

{

  while(secondHere)
  wait();
  secondHere = true;

}
synchronized void third() throws InterruptedException

{
  while (thirdHere) = true;
  wait();
  thirdHere = true;

}
synchronized void finished() throws InterruptedException

{

  allDone++;

  if(allDone == numberOfParticipants)

    {
      notifyAll();

    }

  else while(allDone != numberOfParticipants)

    {

      wait();

    }
  to exit --;
  if (toExit == 0)
    {
      firstHere = false;
      secondHere = false;
      thirdHere = false;
      allDone = 0;
      toExit = numberOfParticipants;
      notifyAll ();

  // release processes waiting for the next action

    }
  }

}
```

Listing 3 A fragment of code that could be synchronized three actions.

Following fragment code in Listing 4 is the continuation of Listing 3, which declaration Entry Protocol and Exit Protocol in Role1.

```
public void role1()
   {
      bolean done = false;
      while (!done)
       {
          try                                          Entry Protocol
            {
               control.first();
               done = true;
            }

          catch (InterruptedException e) { // ignore
       }
         // .... perform action

        done = false;
        while(!done)
       {
         try                                           Exit Protocol
            {
               Control.finished();
            }

         catch (InterruptedException e) { // ignore }

       }


 };
```

Listing 4 Entry Protocol and Exit Protocol in Role1.

Critical Sections: “As an introduction to concurrent programming, a fundamental problem called the *critical section problem”* [7]. The problem is easy to understand and its solutions are small in terms of the number of statements they contain (usually, fewer than five). However, the critical section problem is not easy to solve, and it illustrates just how difficult it can be to write even small concurrent programs.

“A code segment that accesses shared variables and that has to be executed as an atomic action is referred to as a *critical section* [6]. The critical section problem involves a number of threads that are each executing the following code in Listing 5”:

```
while (true)

  {

    entry-section

    critical section // accesses shared variables
or other shareresources.

    exit-section

    noncritical section  // a thread may terminate its execution in this
section.



  }
```

Listing 5 A fragment of code that describe critical problem.

Listing 6 A fragment of code that could be synchronized three action.

The entry- and exit-sections that surround a critical section must satisfy the following correctness requirements [8SPG91]:

- ***Mutual exclusion***. When a thread is executing in its critical section, no other threads can be executing in their critical sections
- ***Progress***. If no thread is executing in its critical section and there are threads that wish to enter their critical sections, only the threads that are executing in their entry- or exit-sections can participate in the decision about which thread will enter its critical section next, and this decision cannot be postponed indefinitely.
- ***Bounded waiting.*** After a thread makes a request to enter its critical section, there is a bound on the number of times that other threads are allowed to enter their critical sections before this thread's request is granted.

Synchronization in Java

To use synchronization, specific data structures should be included in a system, and they should be manipulated by a set of functions.

To provide this functionality the Java has summarize synchronization variables in each and every object. The manipulating of there synchronized variables is done by the help of a thread.join(), synchronized keyword and other methods. These are included in all the libraries and they provide the coordinating of the instructions of threads. Shared data can be protected by means of synchronization variables.

Semaphores, condition variables, wait sets, join(), barriers etc are used to prevent threads from useless waste. Using the synchronized (this) block is so common in instance methods that Java provides a shortened notation for specifying that an instance method is synchronized against its invoking object. These are functionally equivalent definitions of instance method foo is represented in Listing 6 :

```
synchronized void foo()
 {
    void foo()
 }
   count +=2;
```

Listing 7 A fragment of code that describe Method foo.

Semaphores: "Semaphores are an abstract data type which is used to restrict the usage of common shared resources in concurrent programming. Semaphores are implemented mainly in restricting the number of threads that can access some resource"[9].

Semaphores are used for managing the threads that are waiting for something. This can performed by having a thread call sem_wait() in semaphore which have value zero, than the value of semaphore can increment by another thread [1]. This is shown in Listing 8.

```
Java (from Semaphore.java)
    s.semWait();
    s.semPost();
```

Listing 8 A fragment of code that initializing a semaphore.

In the Listing 9, we declare a semaphore s and initialize it to the value 1 by passing 1 in as the third argument. The second argument to sem_init() will be set to 0 in all of the examples we'll see; this indicates that the semaphore is shared between threads in the same process. See the man page for details on other usages of semaphores (namely, how they can be used to synchronize access across different processes), which require a different value for that second argument. After a semaphore is initialized, we can call one of two functions to interact with it, sem_wait() or sem_post(). The behavior of these two functions is seen in Listing 9. For now, we are not concerned with the implementation of these routines, which clearly requires some care; with multiple threads calling into sem_wait() and sem_post(), there is the obvious need for managing these critical sections. We will now focus on how to use these primitives; later we may discuss how they are built. We should discuss a few salient aspects of the interfaces here. First, we can see that sem_wait() will either return right away (because the value of the semaphore was one or higher when we called sem_wait(), or it will cause the caller to suspend execution waiting for a subsequent post. Of course, multiple calling threads may call into sem_wait(), and thus all be queued waiting to be woken. Second, we can see that sem_post() does not wait for some particular condition to hold like sem_wait(). Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up. Third, the value of the semaphore, when negative, is equal to the number of waiting threads [10].

Monitors: The most basic method of communication between threads in Java is synchronization, which is implemented using monitors. A unique monitor is assigned to each object in Java and any thread dealing with the objects reference can lock and unlock the monitor. A locked monitor ensures exclusive access and any other threads trying to lock the same monitor are blocked until the monitor is unlocked.

A monitor is a class used the content of concurrency. The instances of the class will thus be object simultaneously used by several processes. The all methods of the monitor are running with mutual exclusion. Thus, at the same time the methods of the monitor could be running from only one thread. The orders of the mutual exclusion do so easy usage the methods that include a monitor. The other property of the monitor makes the threads to wait for a condition, during this time, when threads are waiting, the thread provisionally take up its exclusive access and should re take it after completed the condition. After completion of the condition one or some threads can be signaled.

Conclusions

At the end of this paper, we have concluded that when it comes to process and thread synchronization, Java offers many possibilities. Its built in constructs for synchronization are not difficult to understand and use in many synchronization scenarios (although we have covered them in general), from those that require allowing just one thread of execution to use protected shared resources at a time to those that may allow a number of threads use protected shared collections of a computer resource.

References

[1] Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Operating System Concepts 8th edition, pages 234, July 29,2008

[2] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman, Inc., 1997.

[3] Dijkstra and Edsger W, Solution of a problem in concurrent programming control, *Communications of the ACM*, Vol. 8, No. 9 (September), page 569.

[4] Krishna Mohan Koyya. A quick at Java Threads. In *Threads and Multi-Threaded*

System, pages 11-12, Glarimy Technology Services, 2011.

[5] Richard H. Carver and Kuo-Chung Tai, Modern Multithreading, Implementing, testing and debugging Multithreaded Java and C++/Pthreads/Win32 Programs, Published by John Wiley & Sons, Inc., Hoboken, 2006.

[6] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, Inc., 1999, Multithreaded Programming.

[7] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, Inc., 1999.

[8] Silberschatz A, J. L. Peterson and P.Galvin. Operating System Concepts Reading, pages 121, 1991