IJIAET International Journal of Innovative Analyses and Emerging Technology

| e-ISSN: 2792-4025 | http://openaccessjournals.eu | Volume: 2 Issue: 3

Parallel Programming in Java for Mobile App Development

Xamidov Elnur Khamidovich, Xodjimatov Jahongir Murodovich

Assistants of Fergana branch of Tashkent University of Information Technologies

Abstract: This article discusses the importance of using parallel programming for mobile applications and creating a view of the processes in it. There is a brief introduction to the use of parallel programming in programs created using the Java programming language. Information is provided on the quality and improvement of the software being developed.

Keywords: synchronization, interaction, Count and Print, Atomic Boolean and Blocking Queue, Thread Local, My User Context.

The use of parallelism is done at the level of subprograms (functions). The solution of small tasks can be formalized with separate functions, the execution of which is distributed among the threads. When multiple processors are available, the execution of a series of threads can continue in parallel. This level requires careful design of synchronization and interaction, which, in turn, is determined by the architecture of the computing system.

The same level of parallelism can be imagined asynchronous execution of SIMD programs, in which the same program code is run on different compute nodes in a distributed memory environment that supports message exchange.

Further expansion of the level of parallelism - objects and applications - is specific to distributed applications that are not covered in our tutorial.

Planning connections or connections between parts of an application is the next stage of design. The connection methods are, as a rule, determined by the architecture of the computer system (common variables, channels, message transmission, etc.) and their specific software is determined by the selected programming tool - language or library.

Since most application threads work within the same task, their performance must be coordinated, so synchronization is an important design point. It is necessary to anticipate which tasks can be performed in parallel, which one - in series and in what order; if some parts of the program have already finished work, then whether they should be engaged in other tasks. It is also important to consider how different small tasks “know” that a solution has been achieved.

If you have too many tasks to perform and all of these tasks do not depend on the results of the previous ones, your computer will be able to use all the processors to perform all of these tasks at the same time. You can use **Multithreading** for If you have large independent tasks, this can speed up the execution of your program.

```
class CountAndPrint implements Runnable {
   private final String name;
CountAndPrint(String name) {
      this.name = name;
   }
```

ISSN 2792-4025 (online), Published under Volume: 2 Issue: 3 in March-2022

```
    /** This is what a CountAndPrint will do */
    @Override
    public void run() {
        for (int i = 0; i< 10000; i++) {
System.out.println(this.name + ": " + i);
        }
    }
    public static void main(String[] args) {
        // Launching 4 parallel threads
        for (int i = 1; i<= 4; i++) {
            // `start` method will call the `run` method
            // of CountAndPrint in another thread
            new Thread(new CountAndPrint("Instance " + i)).start();
        }
        // Doing some others tasks in the main Thread
        for (int i = 0; i< 10000; i++) {
System.out.println("Main: " + i);
        }
    }
}
```

For various instances, the start method code Count And Print is executed in an unpredictable order. An excerpt from the implementation example can look like this:

Instance 4: 1

Instance 2: 1

Instance 4: 2

Instance 1: 1

Instance 1: 2

Main: 1

Instance 4: 3

Main: 2

Instance 3: 1

Instance 4: 4

...

ISSN 2792-4025 (online), Published under Volume: 2 Issue: 3 in March-2022

A simple example of solving a producer and consumer problem. Note that JDK classes (AtomicBooleanandBlockingQueue) are used for synchronization, which reduces the chances of creating an invalid solution. Consult Javadoc for different types of blockingQueue; selecting another program can drastically change the behavior of this example (e.g.,DelayQueue or Priority Queue).

```
public class Producer implements Runnable {
   private final BlockingQueue<ProducedData> queue;
   public Producer(BlockingQueue<ProducedData> queue) {
this.queue = queue;
   }
   public void run() {
      int producedCount = 0;
      try {
         while (true) {
producedCount++;
            //put throws an InterruptedException when the thread is interrupted
queue.put(new ProducedData());
         }
      } catch (InterruptedException e) {
         // the thread has been interrupted: cleanup and exit
producedCount--;
         //re-interrupt the thread in case the interrupt flag is needeed higher up
Thread.currentThread().interrupt();
      }
System.out.println("Produced " + producedCount + " objects");
   }
}
public class Consumer implements Runnable {
   private final BlockingQueue<ProducedData> queue;
   public Consumer(BlockingQueue<ProducedData> queue) {
this.queue = queue;
   }
   public void run() {
      int consumedCount = 0;
      try {
         while (true) {
```

ISSN 2792-4025 (online), Published under Volume: 2 Issue: 3 in March-2022

```
    //put throws an InterruptedException when the thread is interrupted
ProducedData data = queue.poll(10, TimeUnit.MILLISECONDS);
                // process data
consumedCount++;
            }
        } catch (InterruptedException e) {
            // the thread has been interrupted: cleanup and exit
consumedCount--;
    //re-interrupt the thread in case the interrupt flag is needeed higher up
Thread.currentThread().interrupt();
        }
System.out.println("Consumed " + consumedCount + " objects");
    }
}
public class ProducerConsumerExample {
    static class ProducedData {
        // empty data object
    }
    public static void main(String[] args) throws InterruptedException {
BlockingQueue<ProducedData> queue = new ArrayBlockingQueue<ProducedData>(1000);
// choice of queue determines the actual behavior: see various BlockingQueue implementations
        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));
producer. start();
consumer.start();
Thread.sleep(1000);
producer.interrupt();
Thread.sleep(10);
consumer.interrupt();
    }
}
```

A useful tool in Java Concurrency is ThreadLocal, which allows you to have a variable specific to a given thread. Thus, if the same code is running on different threads, these executions do not share a value, but instead each thread has its own variable, i.e. thread-local.

ISSN 2792-4025 (online), Published under Volume: 2 Issue: 3 in March-2022

For example, this is often used to set context (e.g., authorization information) to process a query on a servlet. Here's what you can do:

```
private static final ThreadLocal<MyUserContext> contexts = new ThreadLocal<>();
public static MyUserContextgetContext() {
   return contexts.get(); // get returns the variable unique to this thread
}
public void doGet(...) {
MyUserContext context = magicGetContextFromRequest(request);
contexts.put(context); // save that context to our thread-local - other threads
                       // making this call don't overwrite ours
   try {
      // business logic
   } finally {
contexts.remove(); // 'ensure' removal of thread-local variable
   }
}
```

Now, instead of going to MyUserContext each separate method, you can use it wherever you want MyServlet.getContext (). Now, of course, this provides a variable that needs to be documented, but it is safe for workpieces, which eliminates many of the disadvantages of high-level use of such a variable.

The main advantage here is that each thread has its own local variable in the container contexts. If you use it from a specific access point (e.g., each servlet requires you to save its own context, or perhaps add a servlet filter), you can rely on that context as needed.

Java has a built-in locking mechanism at the language level: a block that can use any Java object as a synchronized set lock (i.e., each Java object can have a monitor connected to it) .

Internal locks provide atomicity for statement groups. To understand what this means for us, let’s look at an example where this synchronized is useful:

```
private static int t = 0;
private static Object mutex = new Object();
public static void main(String[] args) {
ExecutorServiceexecutorService = Executors.newFixedThreadPool(400); // The high thread count is for demonstration purposes.
   for (int i = 0; i< 100; i++) {
executorService.execute(() -> {
         synchronized (mutex) {
            t++;
```

ISSN 2792-4025 (online), Published under Volume: 2 Issue: 3 in March-2022

```
System.out.println(MessageFormat.format("t: {0}", t));
            }
        });
    }
executorService.shutdown();
}
```

In this case, if it weren't for the *synchronized* block, there would be a lot of concurrency issues. The first of these would be with the post increment operator (it's not an atom itself), and the second would be that we would observe the value of t after an arbitrary number of other threads had had a chance to modify it. However, since we have purchased a built-in lock, there will be no race conditions here, and the output will contain the numbers 1 to 100 in their normal order.

There are many parts of Java that have not yet been explored and fully exploited, and it will continue to evolve. Along with solving problems in the programming language, the process of improvement is constantly ongoing. Its potential is widely used and developed in all areas of IT. Parallel programming is also one of the great features of this programming language. In Java, parallel mobile applications have been created and this is natural to continue.

References

1. Воеводин В. В., Воеводин Вл.В. Параллельные вычисления. СПб: BHV-Петербург, 2002.
2. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования; [пер. с англ.]. М.: Вильямс, 2003.
3. Ходжиматов, Ж. М. Параллельное программирование в Java / Ж. М. Ходжиматов // Молодой ученый. – 2021. – № 22(364). – С. 30-34.
4. Lindgren, M. (2021). Evaluation of Battery Usage and Scalability when Performing Parallel Applications on Mobile Devices.
5. Chowdary Ravela, Srikar. 2010. Comparison of Shared memory based parallel programming models. Blekinge: Blekinge Institute of Technology
6. Pacheco, Peter. 2011. An Introduction to Parallel Programming. Burlington, MA: Morgan Kaufmann publishers.
7. Okhunov, M., &Minamatov, Y. (2021). Application of Innovative Projects in Information Systems. European Journal of Life Safety and Stability (2660-9630), 11, 167-168.
8. Минаматов, Ю. (2021). УМНЫЕ УСТРОЙСТВА И ПРОЦЕССЫ В ИХ ПРАКТИЧЕСКОЙ ЭКСПЛУАТАЦИИ. Eurasian Journal of Academic Research, 1(9), 875-879.

ISSN 2792-4025 (online), Published under Volume: 2 Issue: 3 in March-2022