BOOKS IN THE LIBRARY PROJECT

Hüsamettin Demirtaş 190315074

Books in the Library Problem

A group of students need to use the same books to perform their term project. There are 40 students in the class (they have consecutive school numbers from 1 to 40) and all need the same 6 books from the library. The students arrive at the library in random order, and they compete to talk one of the 3 librarians (Students may try to catch one of the librarians randomly).

When a student catches one of the librarians, he/she asks the book he/she needs. The book needed is generated randomly. The librarian should check if the book was borrowed by any other student. If not, that book is assigned to that student for a random time period in milliseconds. If the book is borrowed by someone else the student should leave, wait for a random duration and try to talk to one of the librarians once more.

Each librarian should be able to see the states of the books even if they were borrowed by another librarian.

Each student needs to read all of the 6 books to complete his/her term project.

The project will be implemented either with C or Java but using threads is a must.

Problem Analysis

As it is understood, in the program, including librarian, every actor should act independently from each other and do their acts as excepted from them continuously and simultaneously. Since we had to run multiple tasks at the same time in the same program, we had to use multithreading in our program to perform such an output.

To simulate the given problem, I create different threads for each student and another for the librarian to create a case that students ask for book from the librarians and read the books and librarians fulfill the students' wishes as the students ask. Since these threads had to share same resources(variables) and had to perform similar actions that effects the variables, these variables and actions become critical sections for our program and because of that we faced with race conditions in our code.

To overcome these troubles, we used different methods and classes like **Thread**.**sleep()**, **volatile variables and lock mechanism,** and etc.

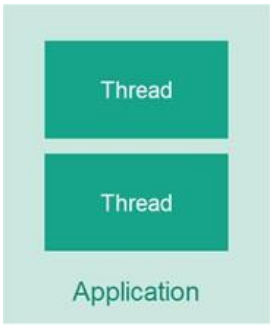
We shared an introduction to the main reasons of problems that we faced and structures that we used to solve these problems in our program.

A Brief Introduction to Multithreading in Java

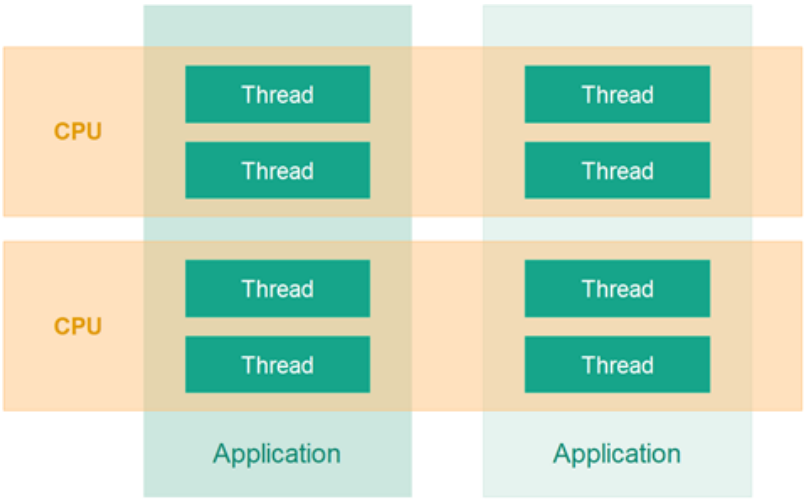
Java *Concurrency* is a term that covers multithreading, concurrency and parallelism on the Java platform. That includes the Java concurrency tools, problems and solutions.

What is Multithreading?

Multithreading means that you have multiple *threads of execution* inside the same application. A thread is like a separate CPU executing your application. Thus, a multithreaded application is like an application that has multiple CPUs executing different parts of the code at the same time.

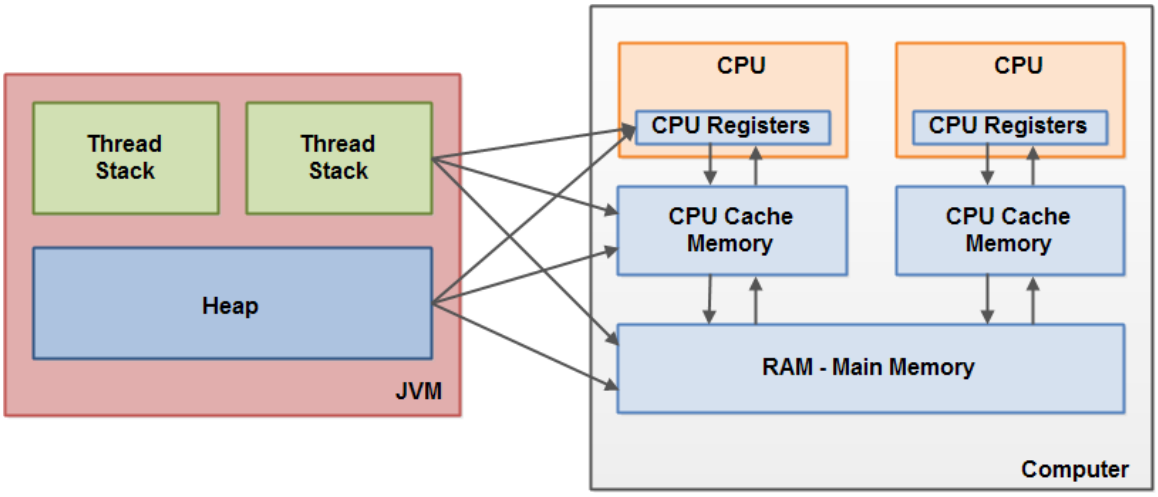


A thread is not equal to a CPU though. Usually a single CPU will share its execution time among multiple threads, switching between executing each of the threads for a given amount of time. It is also possible to have the threads of an application be executed by different CPUs.



Bridging The Gap Between The Java Memory Model And The Hardware Memory Architecture

As already mentioned, the Java memory model and the hardware memory architecture are different. The hardware memory architecture does not distinguish between thread stacks and heap. On the hardware, both the thread stack and the heap are located in main memory. Parts of the thread stacks and heap may sometimes be present in CPU caches and in internal CPU registers. This is illustrated in this diagram:



When objects and variables can be stored in various different memory areas in the computer, certain problems may occur. The two main problems are:

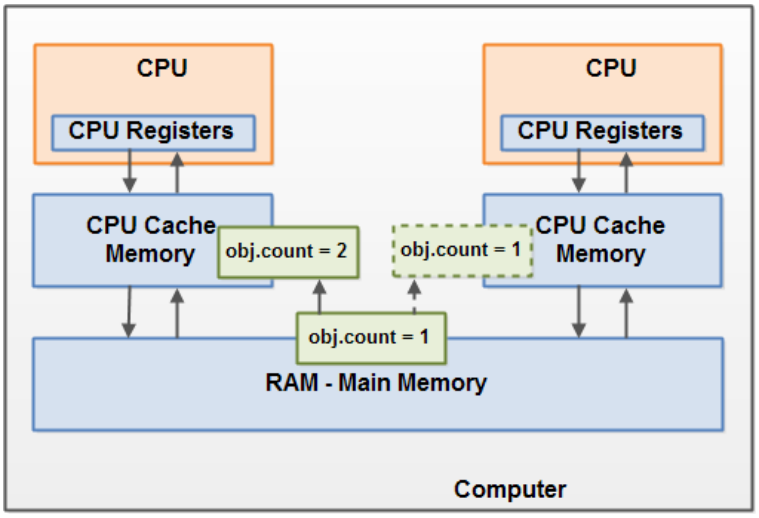
- Visibility of thread updates (writes) to shared variables.
- Race conditions when reading, checking and writing shared variables.

Visibility of Shared Objects

If two or more threads are sharing an object, without the proper use of either volatile declarations or synchronization, updates to the shared object made by one thread may not be visible to other threads.

Imagine that the shared object is initially stored in main memory. A thread running on CPU one then reads the shared object into its CPU cache. There it makes a change to the shared object. As long as the CPU cache has not been flushed back to main memory, the changed version of the shared object is not visible to threads running on other CPUs. This way each thread may end up with its own copy of the shared object, each copy sitting in a different CPU cache.

The following diagram illustrates the sketched situation. One thread running on the left CPU copies the shared object into its CPU cache, and changes its count variable to 2. This change is not visible to other threads running on the right CPU, because the update to count has not been flushed back to main memory yet.



To solve this problem you can use Java's volatile keyword. The volatile keyword can make sure that a given variable is read directly from main memory, and always written back to main memory when updated.

Race Conditions

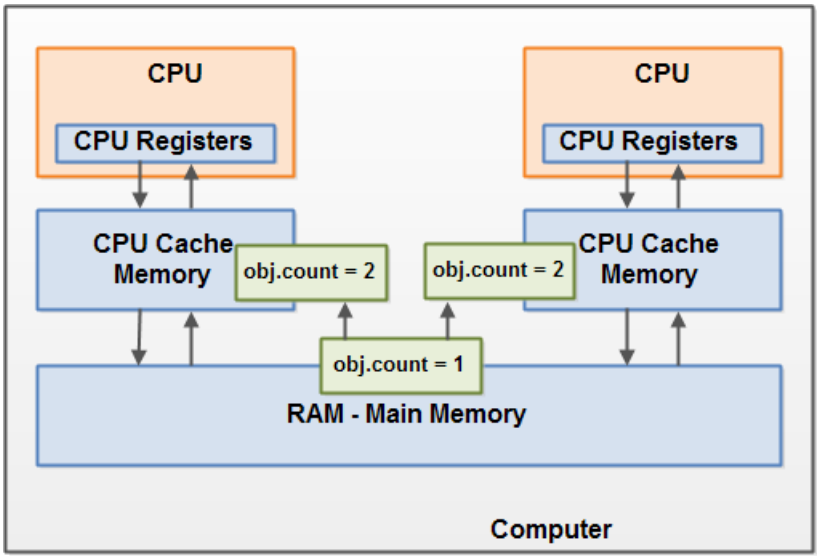
If two or more threads share an object, and more than one thread updates variables in that shared object, race conditions may occur.

Imagine if thread A reads the variable count of a shared object into its CPU cache. Imagine too, that thread B does the same, but into a different CPU cache. Now thread A adds one to count, and thread B does the same. Now var1 has been incremented two times, once in each CPU cache.

If these increments had been carried out sequentially, the variable count would be been incremented twice and had the original value + 2 written back to main memory.

However, the two increments have been carried out concurrently without proper synchronization. Regardless of which of thread A and B that writes its updated version of count back to main memory, the updated value will only be 1 higher than the original value, despite the two increments.

This diagram illustrates an occurrence of the problem with race conditions as described above:



To solve this problem you can use a Java synchronized block. A synchronized block guarantees that only one thread can enter a given critical section of the code at any given time. Synchronized blocks also guarantee that all variables accessed inside the synchronized block will be read in from main memory, and when the thread exits the synchronized block, all updated variables will be flushed back to main memory again, regardless of whether the variable is declared volatile or not.

Race Conditions and Critical Sections

A *race condition* is a special condition that may occur inside a critical section. A *critical section* is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

Critical Sections

Running more than one thread inside the same application does not by itself cause problems. The problems arise when multiple threads access the same resources. For instance, the same memory (variables, arrays, or objects), systems (databases, web services etc.) or files.

In fact, problems only arise if one or more of the threads write to these resources. It is safe to let multiple threads read the same resources, as long as the resources do not change.

Here is a critical section Java code example that may fail if executed by multiple threads simultaneously:

```
public class Counter {

  protected long count = 0;

  public void add(long value){
    this.count = this.count + value;
  }
}
```

Imagine if two threads, A and B, are executing the add method on the same instance of the Counter class. There is no way to know when the operating system switches between the two threads. The code in the add() method is not executed as a single atomic instruction by the Java virtual machine. Rather it is executed as a set of smaller instructions, similar to this:

1. Read this.count from memory into register.
2. Add value to register.
3. Write register to memory.

Observe what happens with the following mixed execution of threads A and B:

```
    this.count = 0;

A: Reads this.count into a register (0)
B: Reads this.count into a register (0)
B: Adds value 2 to register
B: Writes register value (2) back to memory. this.count now equals 2
A: Adds value 3 to register
A: Writes register value (3) back to memory. this.count now equals 3
```

The two threads wanted to add the values 2 and 3 to the counter. Thus, the value should have been 5 after the two threads complete execution. However, since the execution of the two threads is interleaved, the result ends up being different.

In the execution sequence example listed above, both threads read the value 0 from memory. Then they add their individual values, 2 and 3, to the value, and write the result back to memory. Instead of 5, the value left in this.count will be the value written by the last thread to write its value. In the above case it is thread A, but it could as well have been thread B.

Java Volatile Keyword

The Java volatile keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the computer's main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

The Java volatile Visibility Guarantee

The Java volatile keyword is intended to address variable visibility problems. By declaring the counter variable volatile all writes to the counter variable will be written back to main memory immediately. Also, all reads of the counter variable will be read directly from main memory.

Here is how the volatile declaration of the counter variable looks:

```
public class SharedObject {

    public volatile int counter = 0;

}
```

Declaring a variable volatile thus *guarantees the visibility* for other threads of writes to that variable.

The Java synchronized Keyword

Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

The synchronized keyword can be used to mark four different types of blocks:

1. Instance methods
2. Static methods
3. Code blocks inside instance methods
4. Code blocks inside static methods

These blocks are synchronized on different objects. Which type of synchronized block you need depends on the concrete situation. Each of these synchronized blocks will be explained in more detail below.

Synchronized Instance Methods

Here is a synchronized instance method:

```
public class MyCounter {

  private int count = 0;

  public synchronized void add(int value){
      this.count += value;
  }
}
```

Notice the use of the synchronized keyword in the add() method declaration. This tells Java that the method is synchronized.

A synchronized instance method in Java is synchronized on the instance (object) owning the method. Thus, each instance has its synchronized methods synchronized on a different object: the owning instance.

Only one thread per instance can execute inside a synchronized instance method. If more than one instance exist, then one thread at a time can execute inside a synchronized instance method per instance. One thread per instance.

This is true across all synchronized instance methods for the same object (instance). Thus, in the following example, only one thread can execute inside either of of the two synchronized methods. One thread in total per instance:

```
public class MyCounter {

  private int count = 0;

  public synchronized void add(int value){
     this.count += value;
  }
  public synchronized void subtract(int value){
     this.count -= value;
  }

}
```

Synchronized Blocks in Instance Methods

You do not have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

Here is a synchronized block of Java code inside an unsynchronized Java method:

```
  public void add(int value){

    synchronized(this){
       this.count += value;
    }
  }
```

This example uses the Java synchronized block construct to mark a block of code as synchronized. This code will now execute as if it was a synchronized method.

Notice how the Java synchronized block construct takes an object in parentheses. In the example "this" is used, which is the instance the add method is called on. The object taken in the parentheses by the synchronized construct is called a monitor object. The code is said to be synchronized on the monitor object. A synchronized instance method uses the object it belongs to as monitor object.

Only one thread can execute inside a Java code block synchronized on the same monitor object.

MY PROJECT

- **Librarian Class:**

This class has attributes such as, Book object array, Librarian Number to differentiate them and volatile Boolean named isBusy to check whether a librarian is busy or not. Its constructor takes librarian number and book object array as parameter and assign them to the attributes.

It has a method named takeBook(), which takes a book number, and which returns a boolean variable. In this method it firstly switches the isBusy to true which means the librarian object is now will be busy with something. It checks the given book from the books array properties and checks the specific book's availability, if this book is available to pick. The librarian picks it and gives it to the student, return true to notify student that can pick the book and changes the busy status to not busy. If it is not available, it changes busy status to not busy and return false and print to the console which says it tried to pick but it was not in place, must have been picked by another student.

It also has a method named putDown(), which takes the book number as a parameter. This method firstly change the object to the busy mode to prevent concurrent calls, puts the book to the shelf it belongs and switch back to the not busy mode.

It also has some getters such as getNumbe(), which returns the librarian's number and getIsBusy(), which returns the busy status.

- **Book Class:**

This is the class I use as a book object. It has a book number and volatile boolean named available which checks the book's availability.

This class has a method named getAccessibility to check if the book can get picked or cannot get picked. It simply returns the volatile boolean that I mentioned.

The take() method, which librarians use, changes boolean to false, because if the book has been given to someone, it should not be given to

anyone else until it is retrieved.

The putBack() method, which is also used by librarians, changes the boolean to true, which simply mean, that book is now available to pick.

- **Student Class:**

This class is the class which runs on different threads and use the same resources. This class has only two attributes such as studentID and booksToRead array. StudentID for the differentiation between students and the booksToRead is to create partly unique read order for each student.

In the constructor it takes a parameter for student ID and it assign that parameter to appropriate attribute. After that, constructor fills the booksToRead array with consecutive numbers and shuffle it with a method which has been imported by java.util.Collections.

The rest of the methods are very interconnected so I will talk all of them at once.

I implemented the runnable interface and override the run method. When the thread starts it first gets in a while loop which will continue while the booksToRead array is not empty. It will get empty because the student also gets in a for loop which will always take the first element of the booksToRead empty, read it, and delete it from the array. And do this until the student reads all the books to complete his/her project.

This class also has a method named findAvailableLibrarian(), which will run until it can return an available librarian which is not busy at the time. If a student can’t find an available librarian, student will wait random time and come back and try until he/she can find an available one.

Student has a method named askForTheBook(). This method grabs an available librarian ask the needed book from him/her. If the needed book is not available student wait random time and come back and do all those processes again until he/she can get the needed book to read.

When the student gets the book, he/she need, he/she read the book by readBook(), method and to simulate reading process, it wait random time before give back the book.

Student also has a method named giveBackTheBook(). This method takes a book and the librarian a parameter and refund the book by the given librarian. In this method I didn't do any control mechanism because all of those are getting done in overridden run method. When a student wants to refund a book, it first tries to catch an available librarian to refund the book. When he/she catches, he/she refund it. No need to check if is it not there because the main reason why we do this project is to work with multiple threads with same resources. This mean to refund a book it must be taken first. So, the it is for sure that the book is not in place.

The last method of Student is sleep. This sleep function simply sleeps the thread by random milliseconds between 5 and 15. this method is very useful, so I have used it everywhere.

- **Library Class:**

This class is the where all the things happen. This class has the main function and the hub for all other things. This class has attributes such as books', students' and librarians' arrays. Those attributes are getting initialized when the main method starts. Firstly, the books array is initialized. Books objects must be created first because, to make the librarians use the same books as resource, I need to firstly create books and assign them to librarians' parameter. So that way, librarians use the same books. 6 Books are initialized. After that 3 librarians, and Students are initialized. When all the objects are created, students' threads are run one by one.

There is a static method which needs to be mentioned. This static method's main aim to select a random librarian. This static method is used in the student class findAvailableLibrarian method repeatedly until the student can get a available librarian.

OUTPUT EXAMPLE:

```
Librarian 2 has put back the book0 to the shelf
Student#25 has asked for the book4 from the librarian2
Librarian 2 has checked the book4 to take
Librarian 2 has taken the book4 to give to the student who wants it
Student#25 has grabbed the book4 from the librarian2
Student#25 is now reading the book4
Librarian 1 has checked the book5 to take
Librarian 1 has taken the book5 to give to the student who wants it
Student#37 has grabbed the book5 from the librarian1
Student#37 is now reading the book5
======================Student#26 has completed the research======================
======================Student#22 has completed the research======================
Student#25 has given back the book4 to librarian3
Librarian 3 has put back the book4 to the shelf
Student#25 has asked for the book5 from the librarian2
Librarian 2 has checked the book5 to take
Librarian2 cannot take the book5 because it is not in place, it was borrowed to another student
The Book5 that student25 asked is already grabbed by another student, student25 should try again
Student#37 has given back the book5 to librarian2
Librarian 2 has put back the book5 to the shelf
Student#37 has asked for the book1 from the librarian2
Librarian 2 has checked the book1 to take
Librarian 2 has taken the book1 to give to the student who wants it
Student#37 has grabbed the book1 from the librarian2
Student#37 is now reading the book1
Student#37 has given back the book1 to librarian1
Librarian 1 has put back the book1 to the shelf
Librarian 2 has checked the book5 to take
Librarian 2 has taken the book5 to give to the student who wants it
Student#25 has grabbed the book5 from the librarian2
Student#25 is now reading the book5
Student#25 has given back the book5 to librarian2
Librarian 2 has put back the book5 to the shelf
======================Student#37 has completed the research======================
======================Student#25 has completed the research======================

Process finished with exit code 0
```