Java Concurrency

Chapter-3:
Sharing Objects

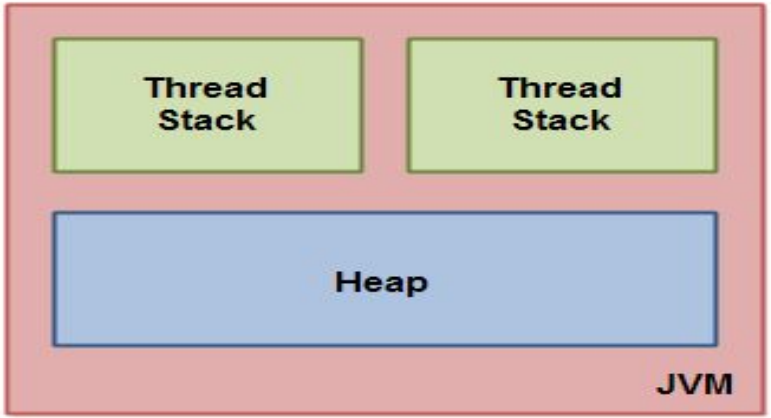
Upcode Software
Engineer Team

-2023-

CONTENT

1. Java Memory Model and CPU 1/1

- **The Java Memory Model (JMM)** describes the behavior of a program in a multithreaded environment.
- The **JVM** defines how the Java virtual machine works with computer memory (**RAM**) and explains the possible behavior of threads and what a programmer should rely on when developing an application.
- The Java memory model used internally in the JVM divides memory between **thread stacks** and the **heap**.

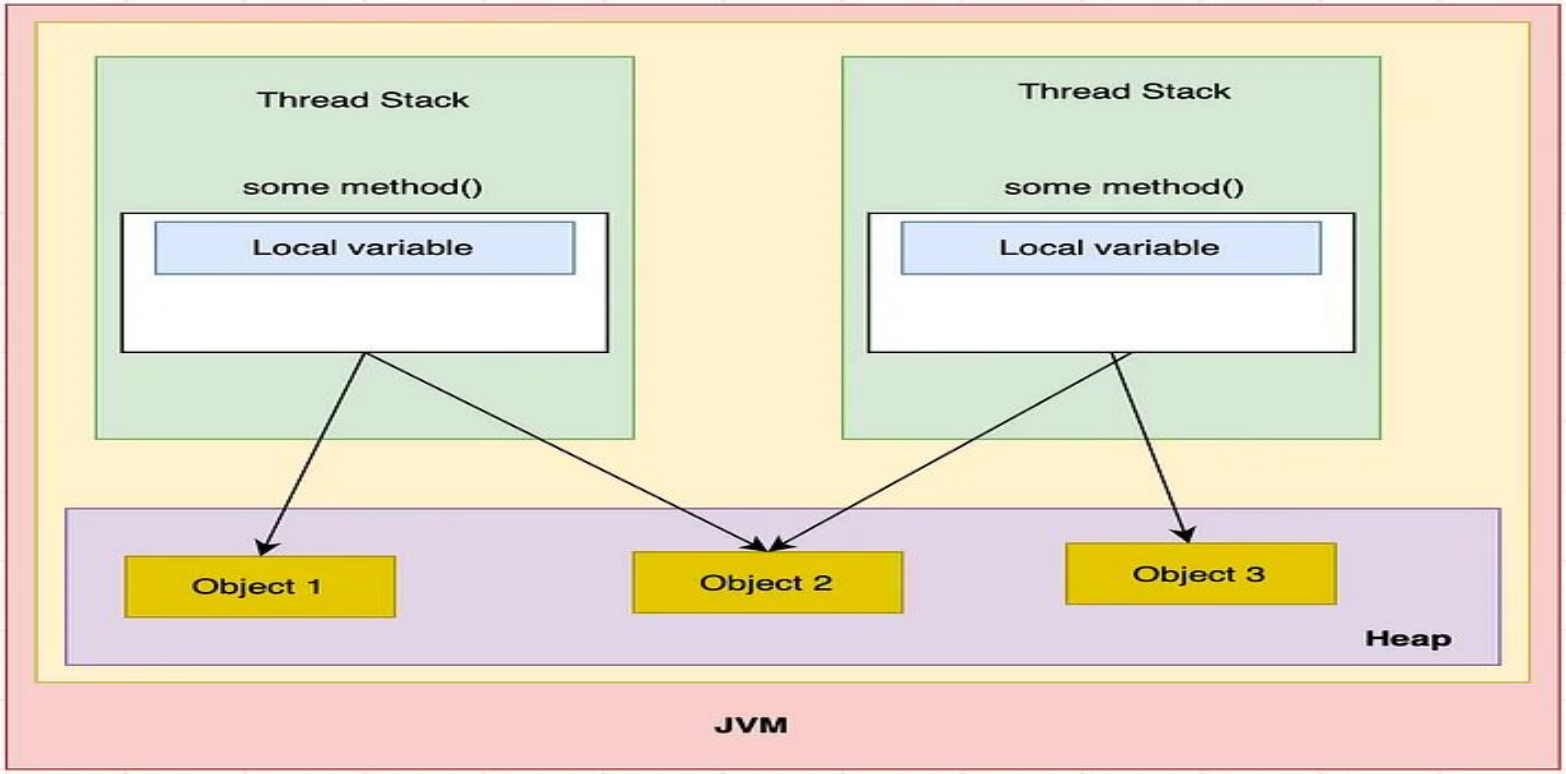


1. Java Memory Model and CPU 1/2

What is a thread stack?

- Each thread has its own stack.
- local variables, object references and information about which methods the thread has called are stored in stack memory.
- Local primitive variables on the stack are only visible to the thread that owns it.
- Even if two threads are executing the same code, they will still create local variables for that code on their own stacks.
- Each thread has its own version of each local variable
- Local reference variables, referencing objects on the heap, also only visible to the thread that owns it.

1. Java Memory Model and CPU 1/3



1. Java Memory Model and CPU 1/4

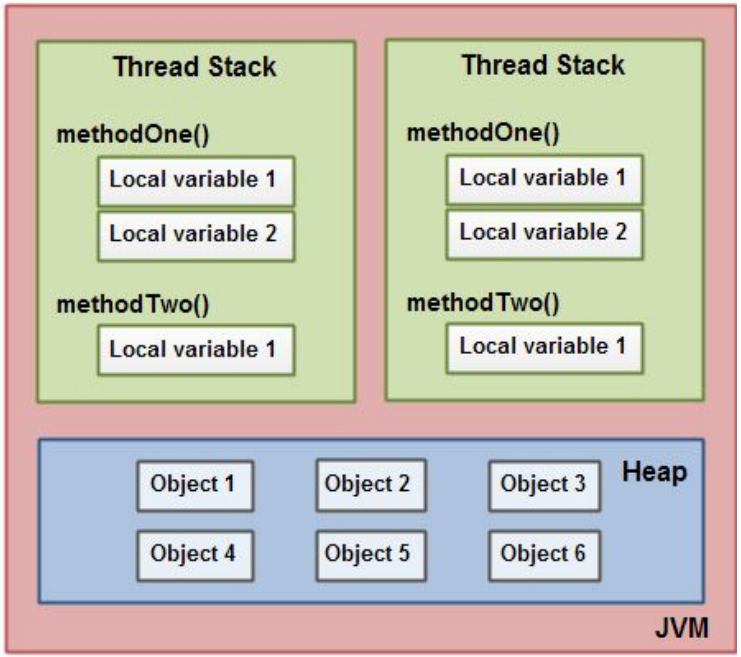
What is the heap?

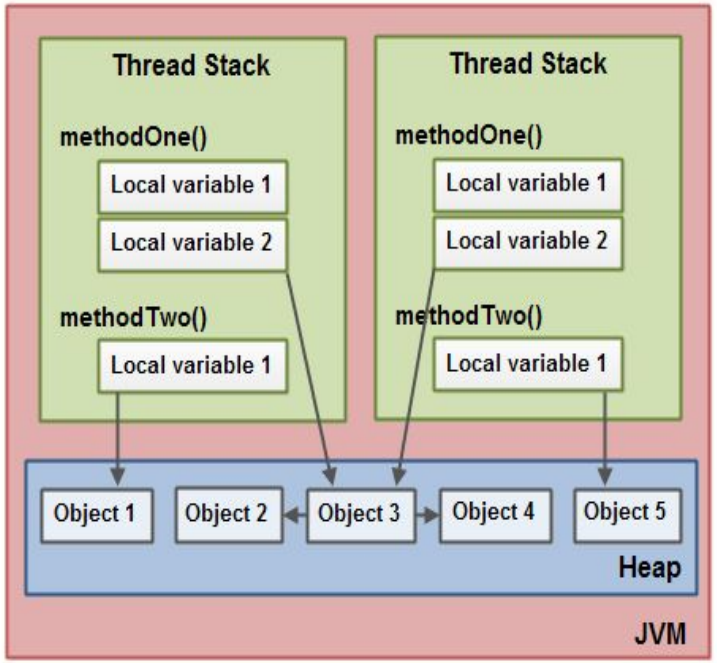
- JVM heap is an independent memory allocation.
- The heap contains all objects created by the application. Objects reside here regardless of which thread created them.
- Objects on the heap are visible to all threads.

What are the differences between stack and heap in terms of multithreading?

- Every thread has your own stack, stack keep locale variables, variables of methods and call stack.
- Variables in stack not visible for other threads.
- Heap is common part of memory. All objects is creating in heap.
- To improve performance, thread can caches values from the heap onto its stack.

1. Java Memory Model and CPU 1/5





1. Java Memory Model and CPU 1/6

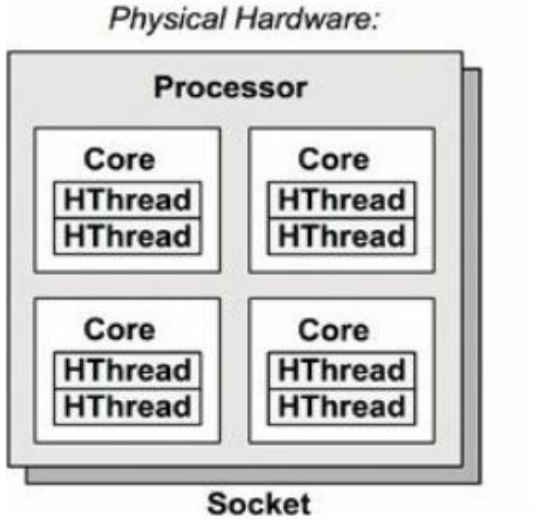
```
public class MyRunnable implements Runnable() {

    public void run() {
        methodOne();
    }

    public void methodOne() {
        int localVariable1 = 45;

        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;

        //... do more with local variables.

        methodTwo();
    }

    public void methodTwo() {
        Integer localVariable1 = new Integer(99);

        //... do more with local variable.
    }
}
```

```
public class MySharedObject {

    //static variable pointing to instance of MySharedObject

    public static final MySharedObject sharedInstance =
        new MySharedObject();


    //member variables pointing to two objects on the heap

    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);

    public long member1 = 12345;
    public long member2 = 67890;
}
```

1. Java Memory Model and CPU 1/7

Processor : is a physical chip that plugs in to socket of the system and it contains one or more CPUs which are implemented as Cores or Hardware threads.
Core : is an independent CPU instance on a multicore-processor. The use of cores in a way to scale the processor is called chip-level multi-processing.

Hardware Thread : in a CPU architecture that supports executing multiple threads simultaneously on each core, every thread is run as an independent CPU instance.
Physical CPU means the actual Physical Core that is present on a processor.
Physical CPU correlates to the actual Physical Cores present on a processor.
4 Physical Cores present on the Single Processor



1. Java Memory Model and CPU 1/8

- **Logical CPU** intern refers to the ability of each core doing 2 or more tasks simultaneously.
- **Each single physical core** can be divided in to multiple logical core by enabling hyperthreading on them.
- In the above diagram, although there are only 4 Physical cores, the system is tricked to look at it as 8 Logical CPUs, by enabling hyperthreading on each core.
- The above diagram correlates to below details : 1 Processor, 4 Physical cores, 8 Logical Cores. In other words - 1P4C - with 2 threads per core.

You can check these details on your system using the command : *lscpu*

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
CPU(s):                4
Thread(s) per core:    2
Core(s) per socket:    2
CPU socket(s):         1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 37
Stepping:              5
CPU MHz:               2667.000
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
NUMA node0 CPU(s):     0-3
```

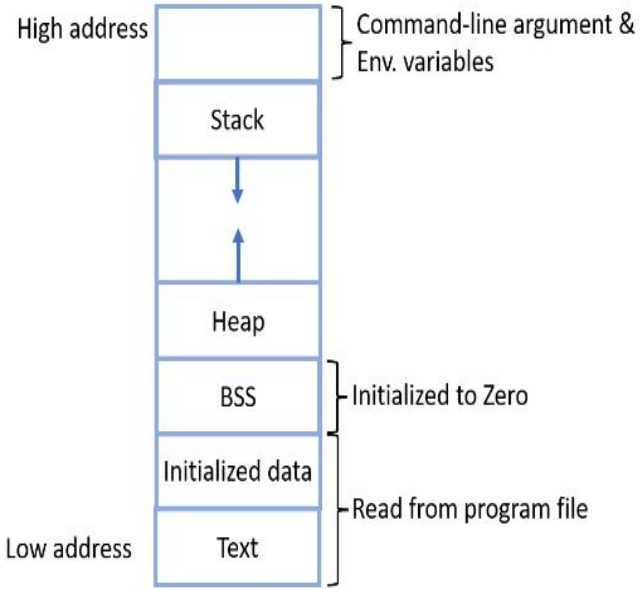
1. Java Memory Model and CPU 1/9

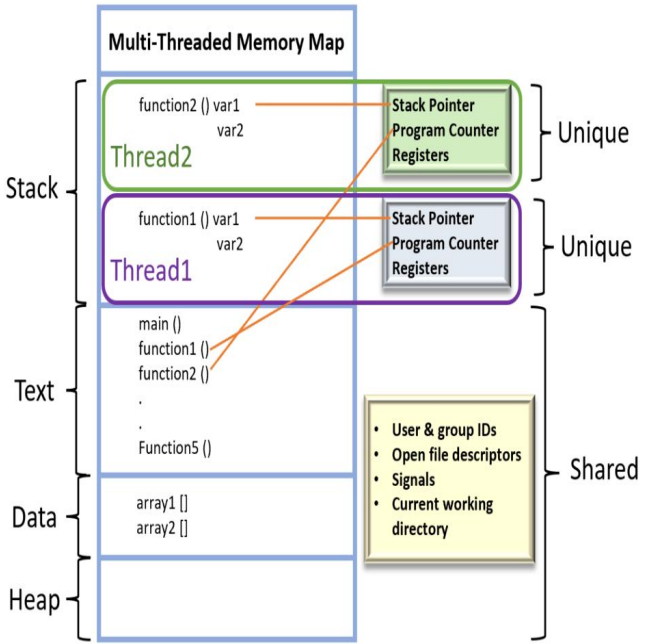
- Threads are usually described as lightweight processes.
- Each thread has its id, a set of registers, the stack pointer, the program counter, and the stack.
- threads share resources with one another within the process they belong to. In particular, they share the processor, memory, and file descriptors.

The memory is logically divided as follows:

- The stack contains local and temporary variables as well as return addresses.
- The heap contains dynamically allocated variables.
- **Text** (**Code**) – these are the instructions to execute.
- **Initialized data** – contains initialized variables.
- **Uninitialized data** – the **block starting symbol** (**BSS**) contains declared but uninitialized static variables.

1. Java Memory Model and CPU 1/10



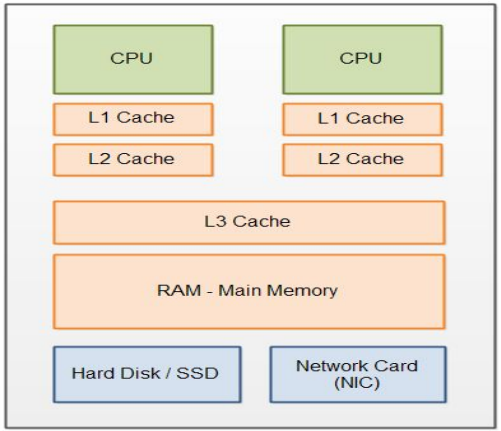


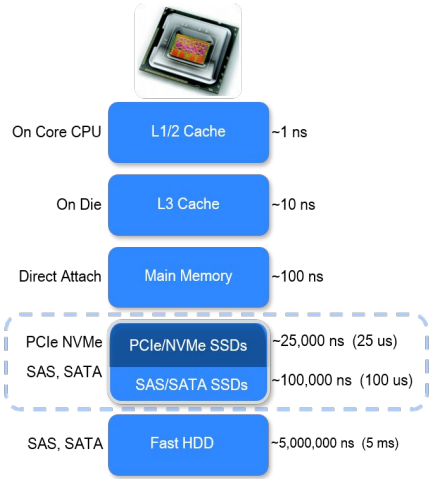
1. Java Memory Model and CPU 1/11

Private Resources	Shared Resources
Thread ID	Instructions (Text/Code)
Registers	Static and global data (Data)
Stack pointer	Uninitialized data (BSS)
Instruction pointer	Open file descriptors
Stack	Signals
Signal mask	Current working directory
Policy or priority (scheduling information)	User and group IDs

1. Java Memory Model and CPU 1/12

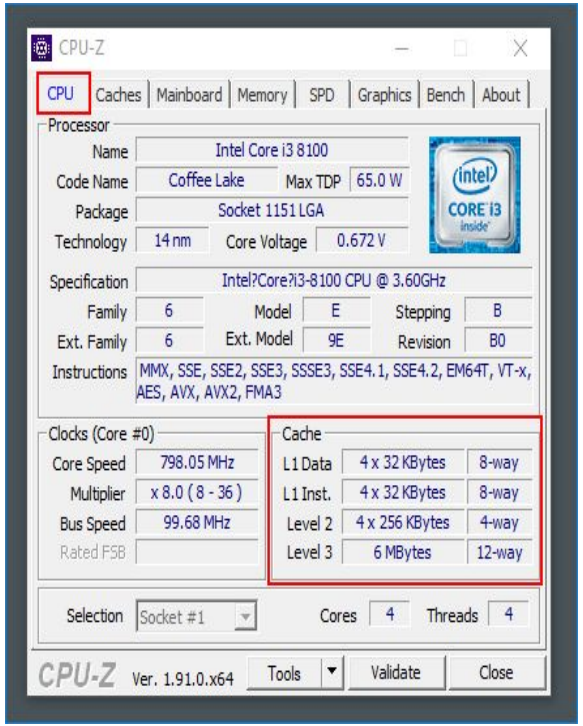
- Modern computers have several powerful **processors**.
- Each of the processors interacts with **computer memory**.
- The processor loads data from memory, processes it, and then writes that data to memory.
- **RAM** has a slower memory access speed than the processor can handle.

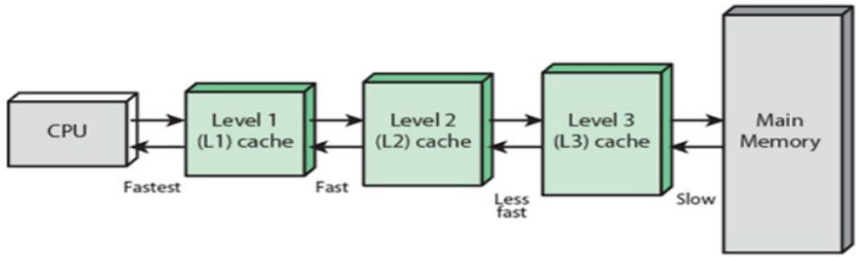


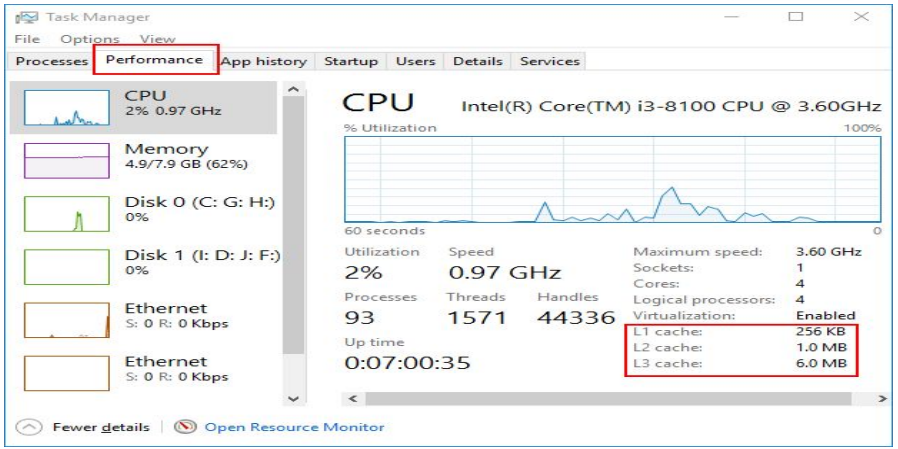


- L1 Cache = 64 KB (per core)
- L2 Cache = 256 KB (per core)
- L3 Cache = 2 - 6 MB (shared)

1. Java Memory Model and CPU 1/13

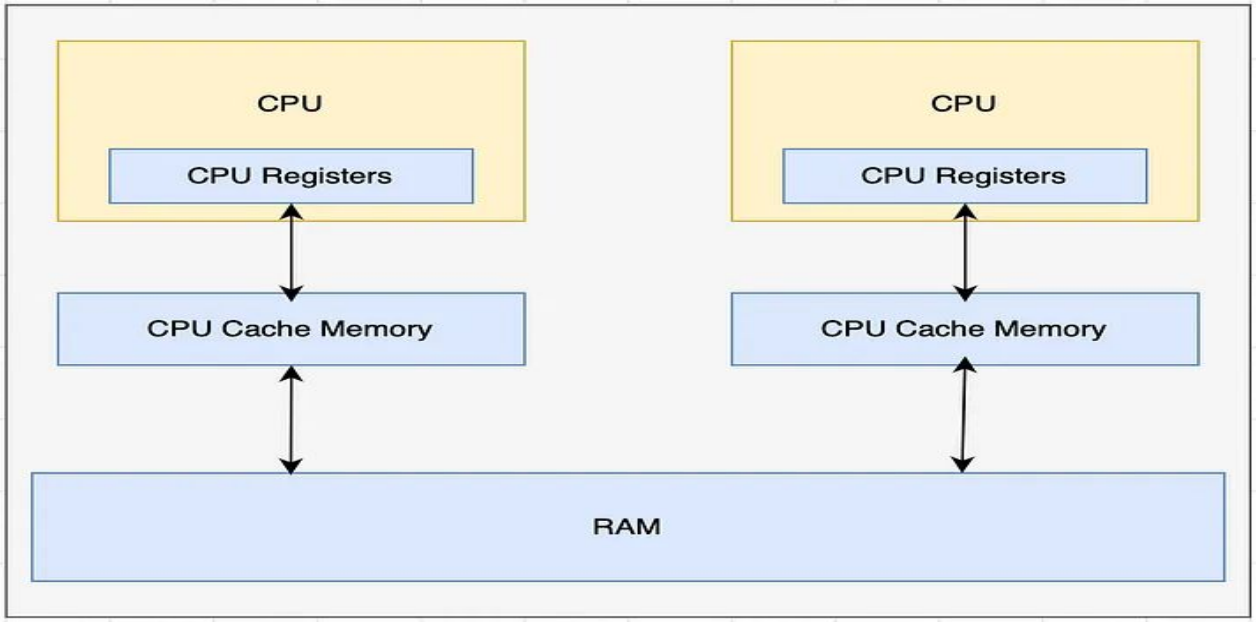






1. Java Memory Model and CPU 1/14

There is a situation in which the speed of the processor and the speed of memory access has a significant difference.



1. Java Memory Model and CPU 1/15

To solve the problem of the fact that the processor is in standby mode while it is accessing data in memory, the processors have added their own cache:

- The cache consists of allocated memory for each CPU and register.
- A cache is faster than accessing the main memory but slower than accessing its internal registers.
- Some CPUs have multiple levels of cache (Level 1, Level 2, Level 3).

1. Java Memory Model and CPU 1/16

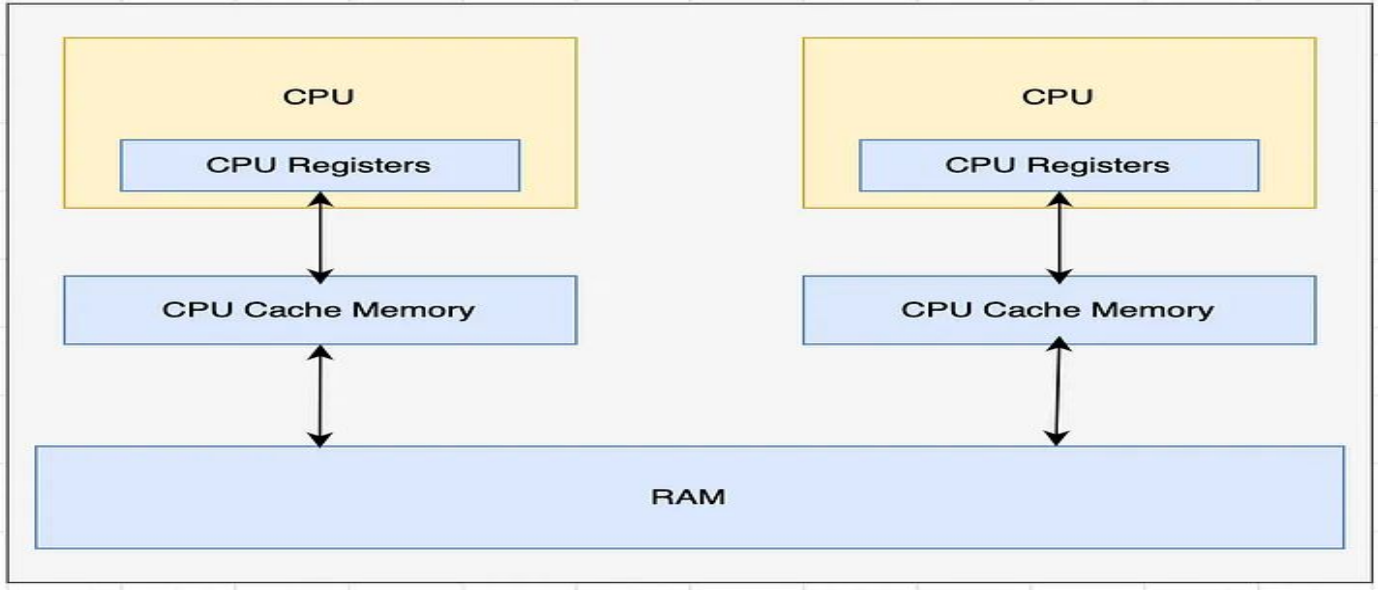
How does the CPU read and write data to main memory?

1. CPU reads data from main memory into it's cache;
2. CPU may read data into it's internal register from it's cache;
3. CPU performs operations on the data;
4. CPU writes data back to the cache if necessary;
5. CPU writes data from the cache to main memory as needed.

1. Java Memory Model and CPU 1/17

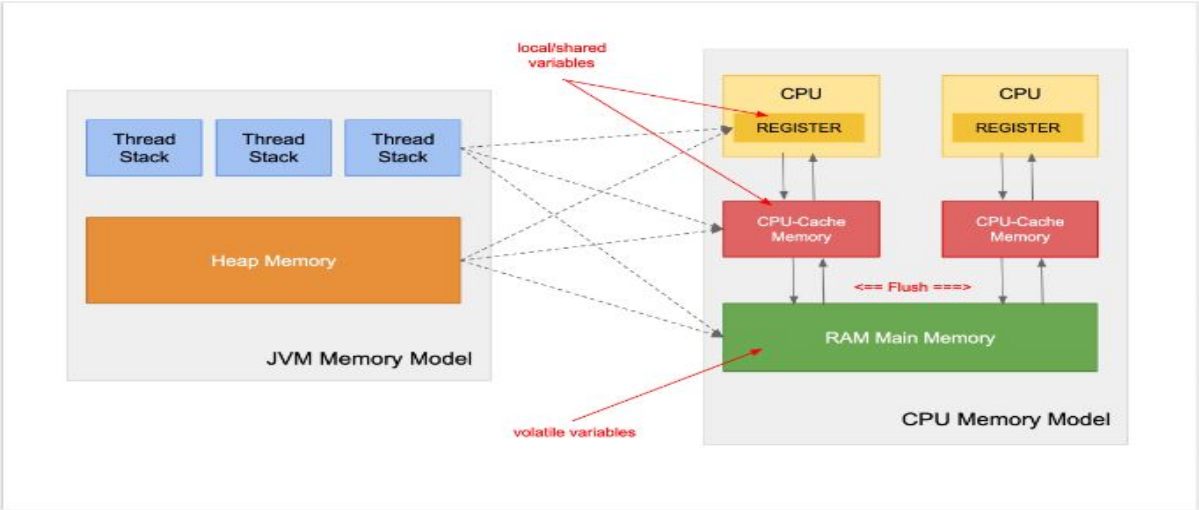
When does the CPU cache clear?

- This happens when the cache is full and the data stored in it is about to be overwritten by new data being loaded into the cache.



1. Java Memory Model and CPU 1/18

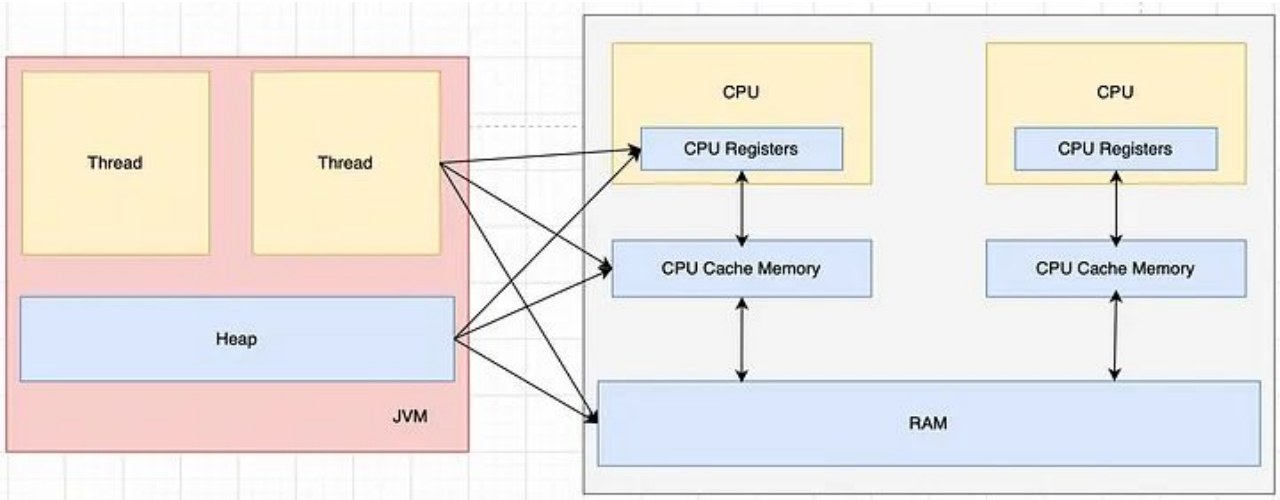
- A modern-day CPU consists of Registers, which act as the direct memory of the processor itself.
- **Cache Memory** — every processor has a cache layer to store data
- **RAM** or **main memory**, where application data is present



JVM vs CPU memory model

1. Java Memory Model and CPU 1/19

- Variables referenced by the stack of thread and heap can be cached in all levels of the computer's memory.



1. Java Memory Model and CPU 1/20

This behaviour of keeping data in memory can create problems:

- First, threads can lose visibility updates, what was made by other threads to shared objects.
- Second, it's race conditions. Threads can hold on to stale shared variable state and threads will be see unsynchronized data.

Data race *— known as* ***race condition*** *(or* ***thread interference****). When 2+ threads read/write same variable and as a result value of the variable gets messed up (recall first part where I spoke about cache).*

Synchronization actions *—* ***lock*** *acquisitions and release, reads and writes of* ***volatile*** *variables are totally ordered.*

1. Java Memory Model and CPU 1/21

What do synchronization actions stand for and what keywords? Look below:

- field scoped: **final, volatile**
- method-scoped: **synchronized, java.util.concurrent.***

Be aware that certain words like ***volatile*** *,* ***synchronized*** *slow down the performance.*

Be wise when applying those

1. Java Memory Model and CPU 1/22

- On the hardware or CPU, both the Thread Stack and Heap are located in main memory.
- Parts of the Thread Stack and Heap may sometimes be present in the **CPU Cache** and in internal **CPU Registers**.

The following are issues that can occur due to the above architecture:

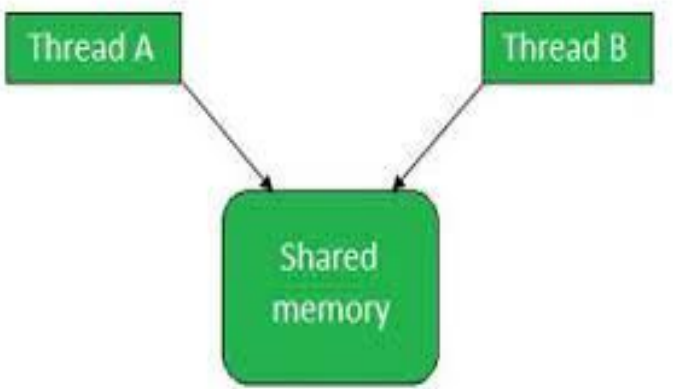
- **Visibility of thread** updates (writes) to shared variables are not immediately seen by all threads accessing the variables.
- **Race conditions** when reading, checking, and updating data of the shared variables.

2.Visibility in multiple threads 2/1

Visibility — determines when activities in one thread are made visible from another thread.

To ensure visibility we have two options:

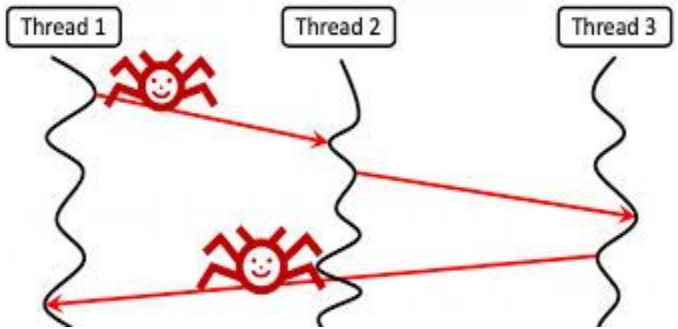
- **Locking**: Guarantees visibility and atomicity (as long as it uses the same lock).
- **Volatile field**: Guarantees visibility.




```
class Worker {
    boolean done = false;

    void work() {
        while (!done) {
            // do work
        }
    }

    void stopWork() {
        done = true;
    }
}
```

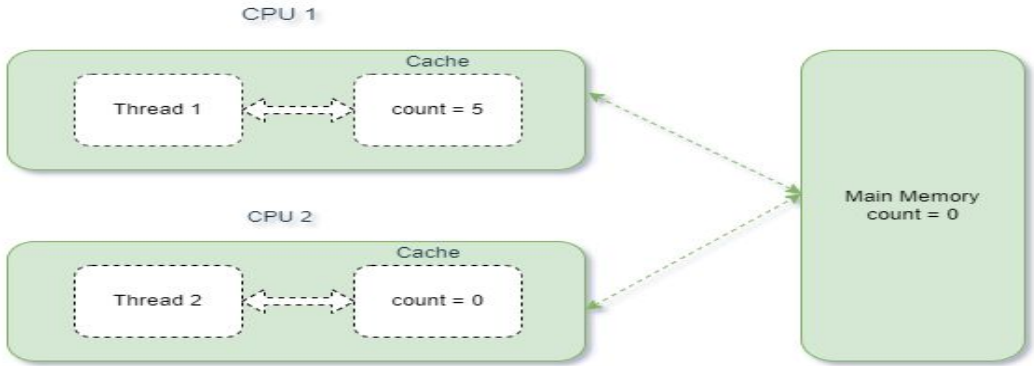
3. Stale data 3/1

- When the reader thread examines ready, it may see an out-of-date value: **stale data**
- Stale data can cause serious and confusing failures such as unexpected exceptions, corrupted data structures, inaccurate computations, and infinite loops



4. Volatile variables 4/1

- The CPU does not cache the value of a *volatile* variable.
- In the case of non-*volatile* variables, the JVM does not guarantee when the value will be written back to the main memory from the cache.
- On the other hand, if we declare *count* as *volatile*, each thread sees its latest updated value in the main memory without any delay.
- With the ***volatile*** keyword, we instruct the JVM and the compiler to store the ***count*** variable in the main memory.



4. Volatile variables 4/2

The ***volatile*** keyword is useful in two multi-threading scenarios:

- When only one thread writes to the ***volatile*** variable and other threads read its value. Thus, the reading threads see the latest value of the variable.
- When multiple threads are writing to a shared variable such that the operation is atomic. This means that the new value written does not depend on the previous value.

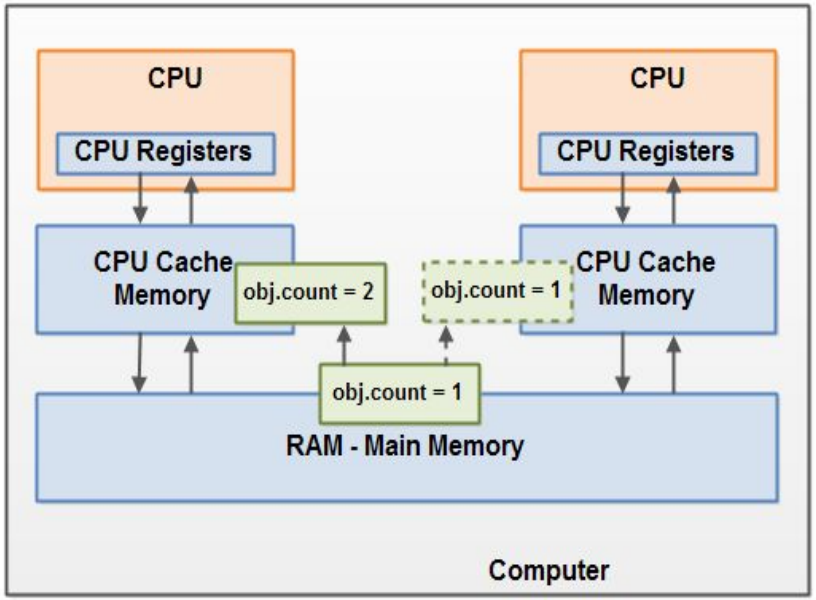
```
public class Counter {

    private volatile int counter;

    // standard constructors / getter

}
```

4. Volatile variables 4/3

- 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.
- 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.

4. Volatile variables 4/4

- 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.



4. Volatile variables 4/5

Volatile Keyword	Synchronization Keyword
Volatile keyword is a field modifier.	Synchronized keyword modifies code blocks and methods.
The thread cannot be blocked for waiting in case of volatile.	Threads can be blocked for waiting in case of synchronized.
It improves thread performance.	Synchronized methods degrade the thread performance.
It synchronizes the value of one variable at a time between thread memory and main memory.	It synchronizes the value of all variables between thread memory and main memory.
Volatile fields are not subject to compiler optimization.	Synchronize is subject to compiler optimization.

5. Thread confinement 5/1

Thread confinement is achieved by designing your program in a way that does not allow your state to be used by multiple threads and is, thus, enforced by the implementation.
There are three types of thread confinement:

- Ad-Hoc Thread Confinement
- Stack Confinement
- ThreadLocal

6. Immutability 6/1

Immutability means - a behavior of an object to not allow any modifications to its state, once the constructor for this object has finished execution.

making object immutable:

- private keyword
- final keyword
- do not use setter

```
public class MessageService {

    private final String message;

    public MessageService(String message) {
        this.message = message;
    }

    // standard getter

}
```

7. Final Fields 7/1

- **Final** keyword can be used with variable, method or class.
- If we make a method **final**, we can not override it in subclass.
- If we make the class **final** it can not be sub classed and making a class final also makes all of its method final.

```
1  final List<String> nameList = new ArrayList();
2  //valid
3  nameList.add("Code Pumpkin");
4  //valid
5  nameList.add("Code Cheif");
6  //Invalid as we can not change the collection reference
7  nameList = new LinkedList<>();
```

7. Final Fields 7/2

Final Method

```
1  class Bike {
2      final void start() {
3          System.out.println("kick start");
4      }
5  }
6
7  class Honda extends Bike {
8      void start() {
9          System.out.println("self start");
10     }
11
12     public static void main(String args[]) {
13         Honda honda = new Honda();
14         honda.start();
15     }
16 }
```

Here, we have declared `start` method in bike as final. so while extending the Honda class with bike we can not override the `start()` method. So, our program will give compilation error.

7. Final Fields 7/3

Final Class

Now lets modify above example and make the Bike class final.

```
1  final class Bike {
2      void start() {
3          System.out.println("kick start");
4      }
5  }
6
7  //Can not extend final class. Code will not compile
8  class Honda extends Bike {
9      void start() {
10         System.out.println("self start");
11     }
12
13     public static void main(String args[]) {
14         Honda honda = new Honda();
15         honda.start();
16     }
17 }
```

In above example, when we try to extend bike class the compiler will give error that we can not extend final class.

7. Final Fields 7/4

Benefits Of Final Keyword:

- Final keyword improves performance as JVM cache the final variables. Making a class, method or variable final in Java helps to improve performance because JVM gets an opportunity to make assumption and optimization.
- Immutable Class can be created with the help of final keyword.
- Final variables are safe to share in multi-threading environment without additional synchronization overhead.
- Final keyword allows JVM to optimized method, variable or class.

7. Final Fields 7/5

- Final member variable must be initialized at the time of declaration or inside constructor, failure to do so will result in compilation error.
- You can not reassign value to final variable in Java.
- Local final variable must be initializing during declaration.
- Only final variable is accessible inside anonymous class in Java.
- Final method can not be overridden in Java.
- All variable declared inside java interface are implicitly final.
- Making a java collection reference variable final means only reference can not be changed but we can add, remove or change object inside collection.

Reference

1. JAVA-Concurrency-in-Practice book
2. Java-multithreading-part-1-java-memory-model
3. Java-memory-model-practical-guide-java
4. Java-memory-model
5. Threads-sharing-resources
6. Thread-confinement-in-java
7. Volatile-keyword-in-java
8. Volatile
9. Java-happens-before-guarantee
10. Final Keyword In Java
11. Java-thread-safety

Thank you!

Presented by

Sanjar Suyunov

(sanjarsuyunov1304@gmail.com)