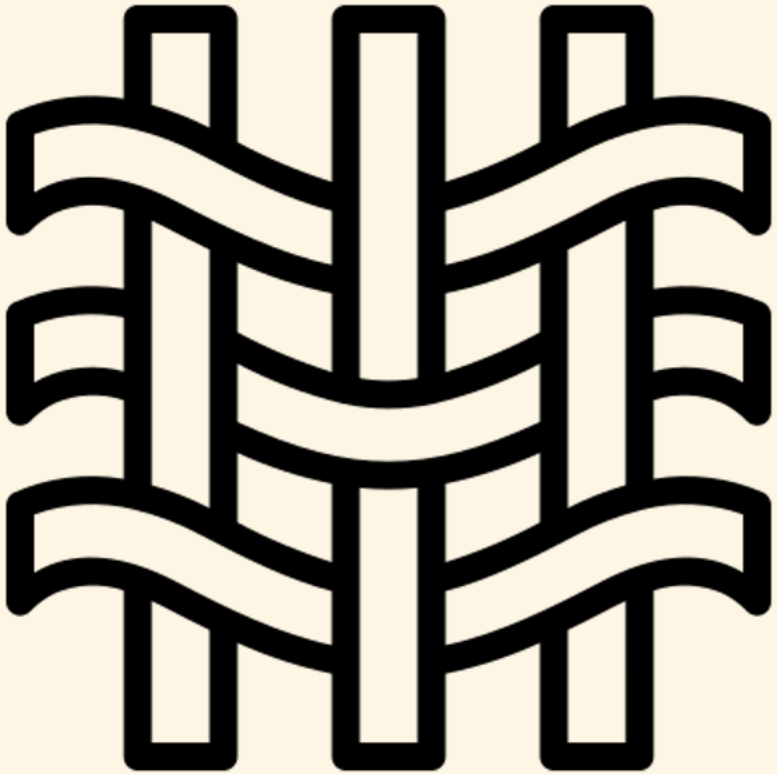
MultiThreading

eazy bytes

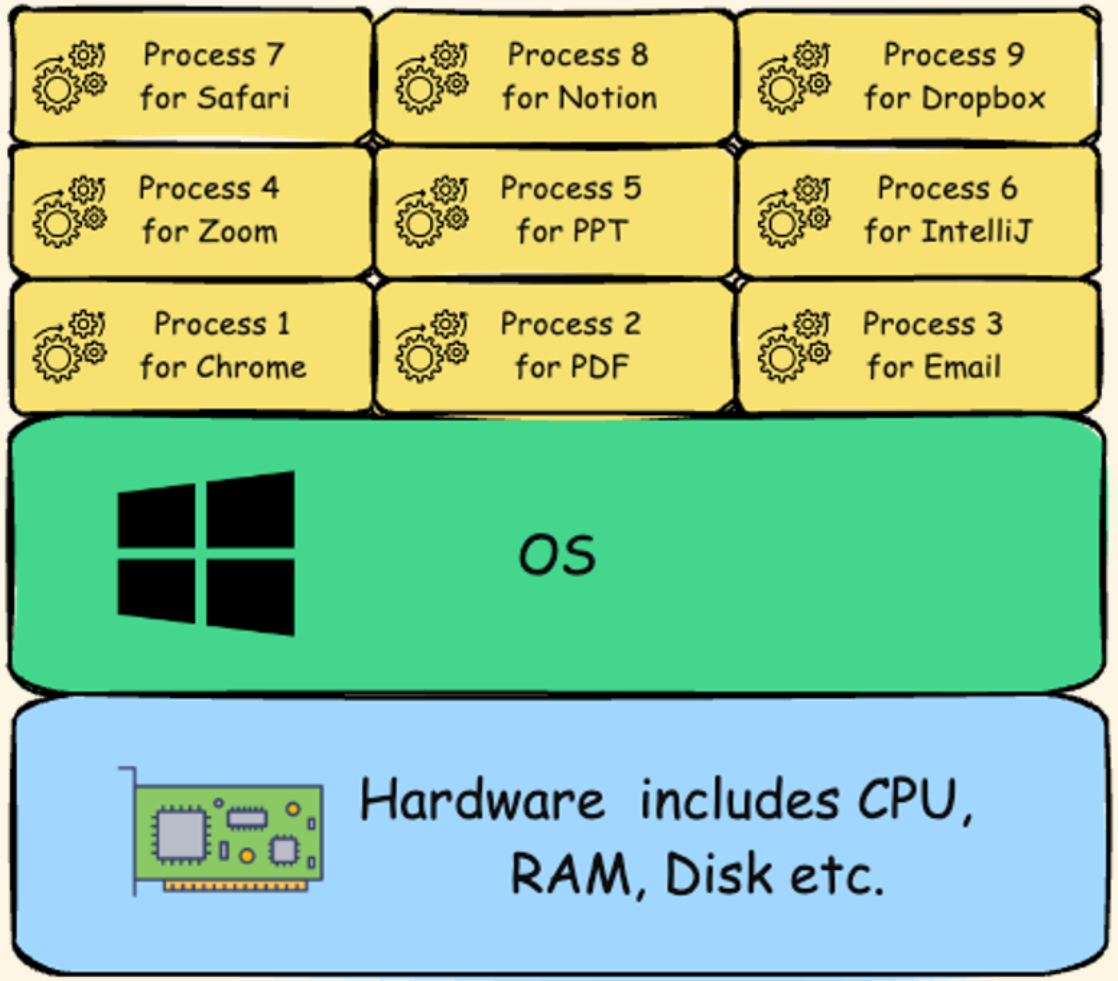


Multithreading in Java refers to the concurrent execution of multiple threads within a Java program. A thread is a lightweight sub-process, and multithreading allows you to perform multiple tasks simultaneously, improving the overall efficiency of your program.

Before we deep dive into Threads, first let's try to learn some basics & concepts around MultiThreading

How a program or a software executes inside a computer ?

eazy bytes



The relationship between hardware, the operating system (OS), processes, and threads is crucial for the execution of programs on a computer. Here's an overview:

Hardware:

The hardware consists of the physical components of a computer, such as the CPU, memory, storage, input/output devices, etc. The CPU is the key component responsible for executing instructions.

Operating System (OS):

The OS is system software that manages hardware resources and provides services for computer programs. It abstracts the hardware, providing a consistent interface for applications to run on different types of hardware.

Processes:

A process is a program in execution. It is an independent unit that runs in its own memory space. The OS manages processes, allocates resources, and provides isolation between them. Each process has its own address space, file descriptors, and system resources.

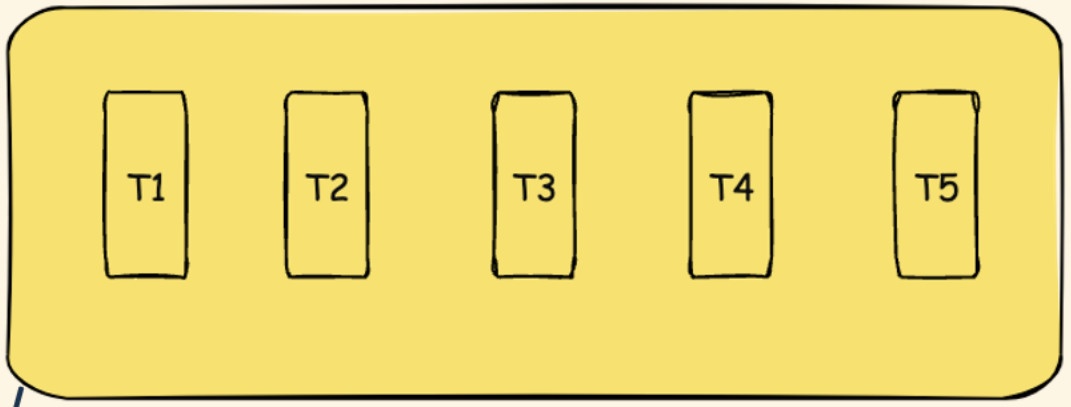
Threads:

A thread is the smallest unit of execution within a process. Multiple threads can exist within a single process. Threads within a process share the same memory space, allowing for efficient communication.Threads can run concurrently, and the OS schedules their execution on the CPU.

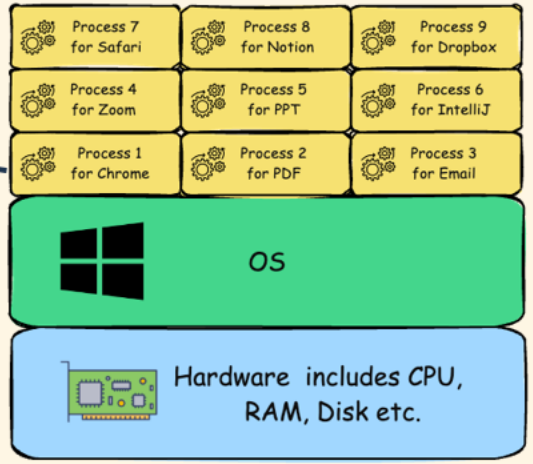
How a program or a software executes inside a computer ?

eazy bytes

Process of Chrome



Various threads inside a chrome process handling various tasks like showing a web page, Downloading a file, playing a video on Youtube, whatsapp chatting etc.



A thread refers to a sequence of instructions within a process that can be executed independently by a CPU core.

Within a process, multiple threads are created, enabling concurrent execution of various tasks and enhancing CPU resource utilization, ultimately boosting throughput.

Take the example of launching Google Chrome: the operating system generates a process for Chrome, and within that process, tasks like showing a web page, Downloading a file, playing a video on Youtube, whatsapp chatting etc. occur simultaneously due to separate threads.

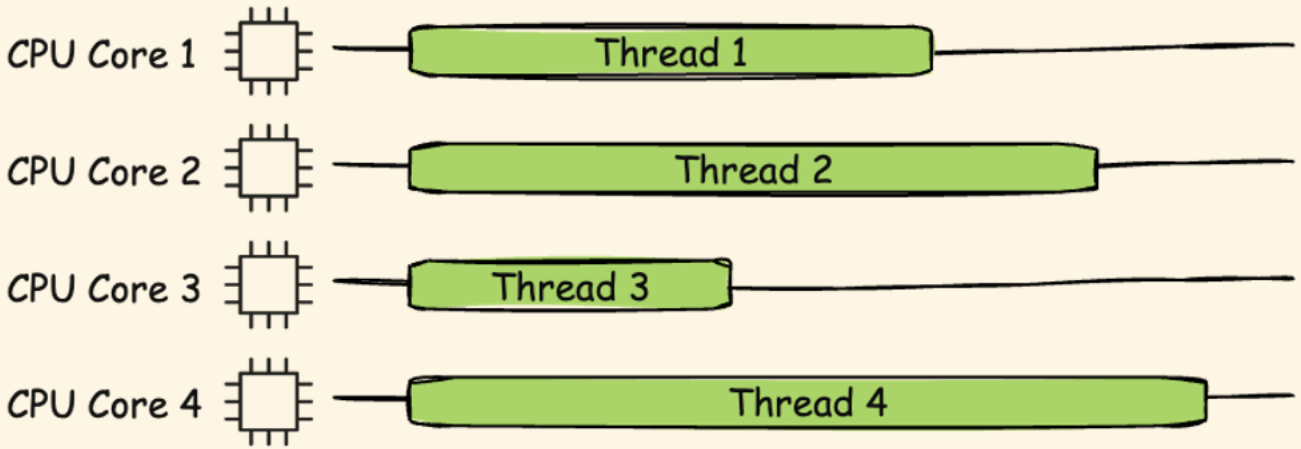
Threads are often termed as Lightweight processes since they exist within a process and share the same address space.

Parallel vs Concurrent Execution

eazy bytes

Parallel Execution

In the realm of computing, parallel execution involves a computer simultaneously performing multiple tasks. Consider a scenario with a 4-core CPU where each core is engaged in running a distinct task, allowing for genuine simultaneous execution of all four tasks.



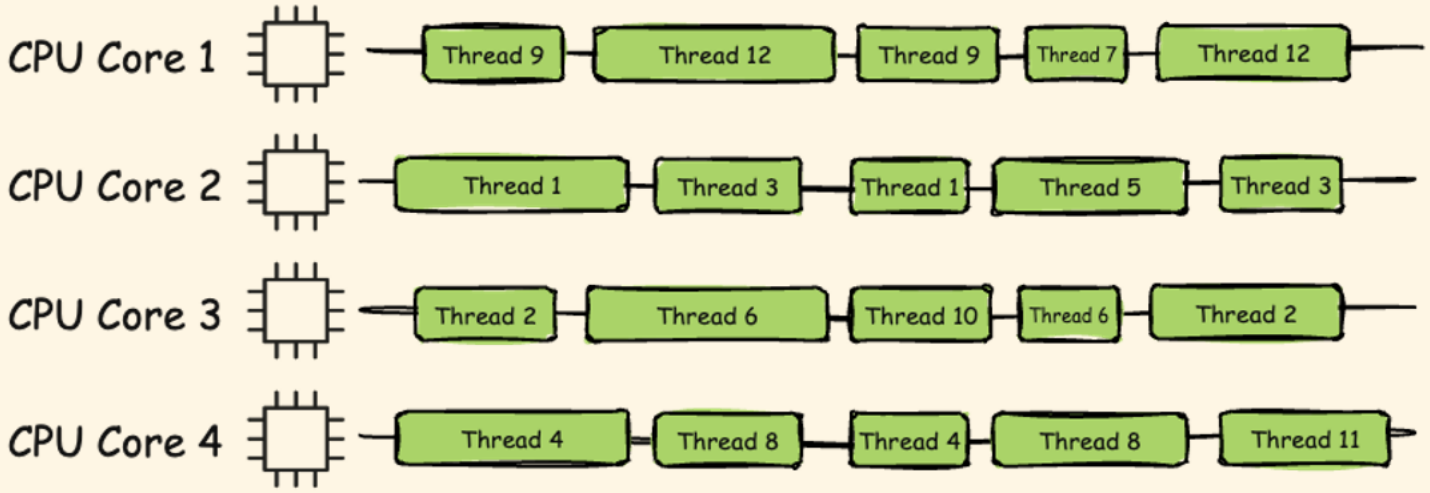
Analogy: Imagine a well-organized kitchen with four chefs, each working on a separate dish. Each chef independently prepares their dish at the same time. As a result, all four dishes are ready parallely.

Parallel vs Concurrent Execution

eazy bytes

Concurrent Execution

On the other hand, concurrent execution creates the appearance of simultaneous task execution when the number of tasks exceeds the available CPU cores. For instance, with a 4-core CPU, attempting to execute 12 different tasks triggers the operating system to engage in context-switching. This process gives the illusion of concurrently handling all eight tasks, even though, in reality, only four instructions can execute at any given time due to the limitation of having four cores.



Analogy: Now, envision the same kitchen, but with only four cooking stations (cores) available. However, there are 12 dishes to be prepared. In this case, the chefs (tasks) take turns using the available cooking stations. While it may seem like all 12 dishes are being prepared at once, only four chefs can actively cook at any given time due to the limited number of cooking stations.

How multiple threads improve performance ?

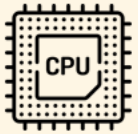
eazy bytes

With a CPU having just 4 cores, it implies that at most 4 parallel executions can occur simultaneously. Each thread must be assigned to a CPU core for execution, given the 4-core limitation, resulting in the capability to run only 4 threads in parallel.

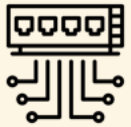
However, a question arises: why do applications often incorporate hundreds of threads? What purpose does this serve? Why not limit the number of threads to match the CPU core count?

Let's explore the reasons behind this practice below.

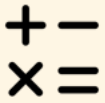
Tasks can be categorized into two main types: CPU-bound and IO-bound.



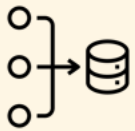
CPU-Bound tasks are those whose execution heavily relies on the CPU, involving operations like arithmetic, logical processing, and relational calculations.



On the other hand, IO-Bound tasks are highly dependent on Input/Output operations, such as communication with a network or performing read/write operations on a file within a file system.



Consider a scenario that exclusively involves CPU tasks, such as a mathematical computation algorithm. For instance, a program might perform intricate mathematical calculations, analyze large datasets, and generate complex statistical models—all without engaging in any input/output operations. In this context, the tasks are purely CPU-bound, emphasizing the computational capabilities of the system.

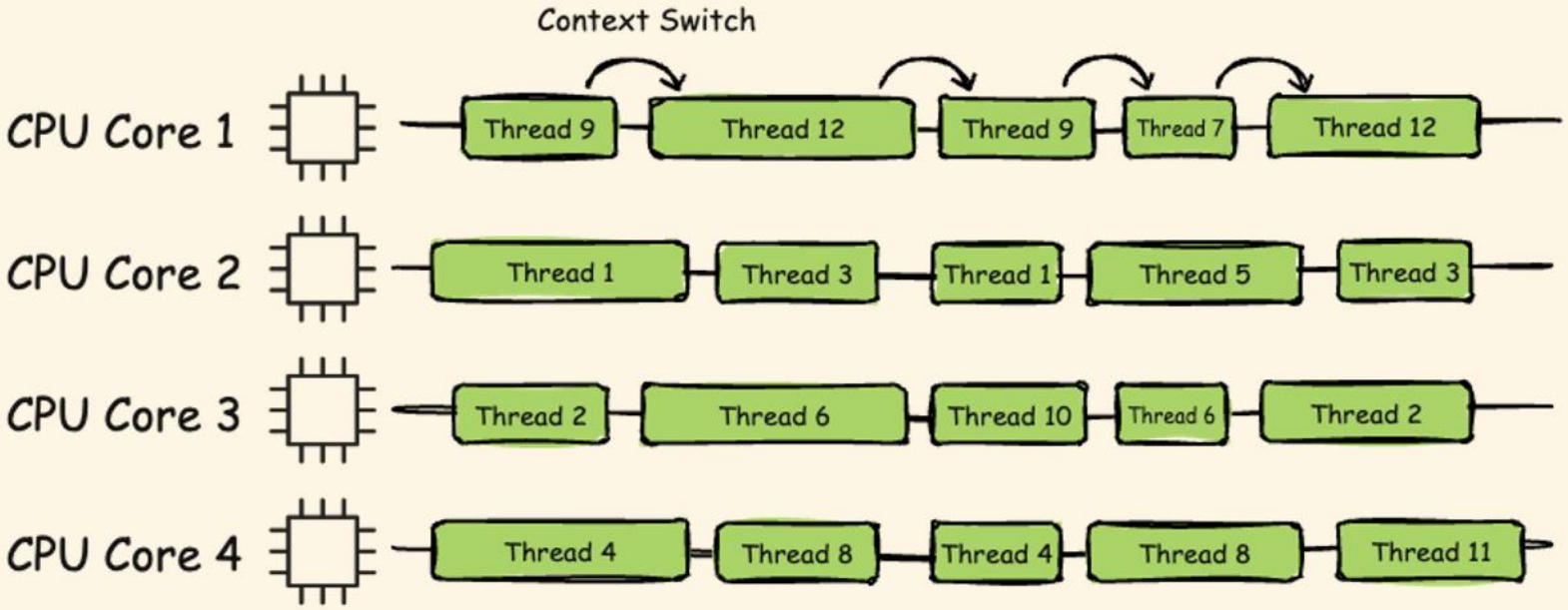


In practice, tasks often involve a combination of both CPU and IO operations. Take, for example, a task where data is fetched from a database (IO), complex calculations are performed on the retrieved data (CPU), and the results are then sent to a remote server (IO). Here, the database interaction and server communication involve IO operations, while the computational processing falls under CPU-bound activity. This illustrates the typical scenario of tasks combining various types of operations.

How multiple threads improve performance ?

eazy bytes

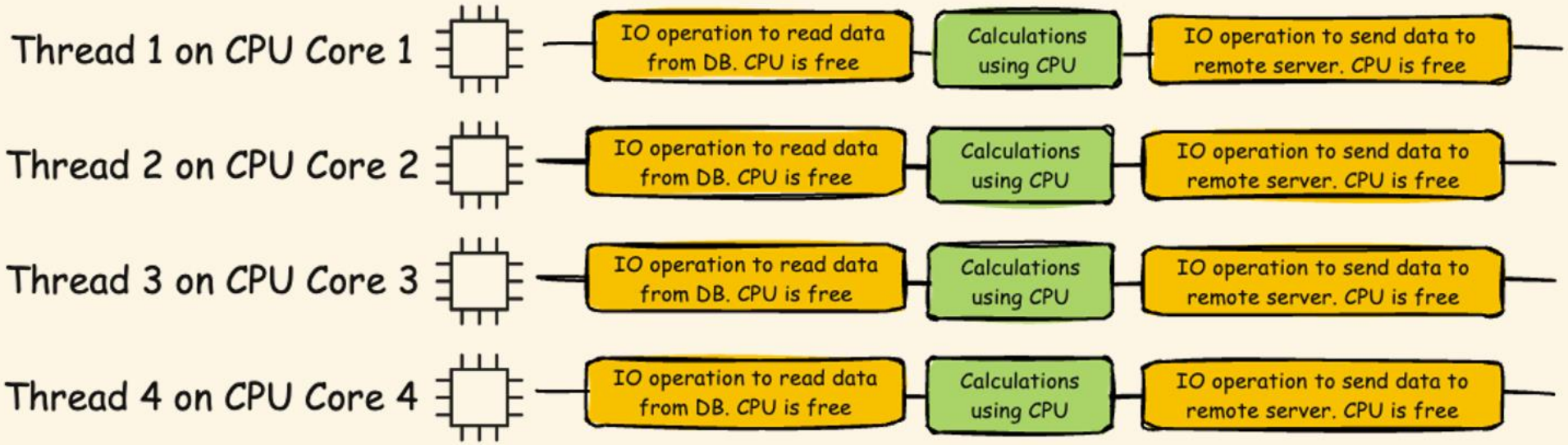
Examining our CPU-intensive arithmetic task executed across 12 threads reveals that, despite the potential for concurrent execution, the actual benefit is questionable. The CPU, with its 4 cores, engages in frequent context-switching between these threads before any of them completes. In this context, employing 12 threads is counter productive, as it leads to inefficient utilization of CPU resources. For tasks primarily bound by CPU processing, it is more beneficial to align the number of threads closely with the available cores to optimize efficiency.



How multiple threads improve performance ?

eazy bytes

Now let's take the example of the task where data is fetched from a database (IO), complex calculations are performed on the retrieved data (CPU), and the results are then sent to a remote server (IO). First let's try running on only 4 threads since 4 cores are available.

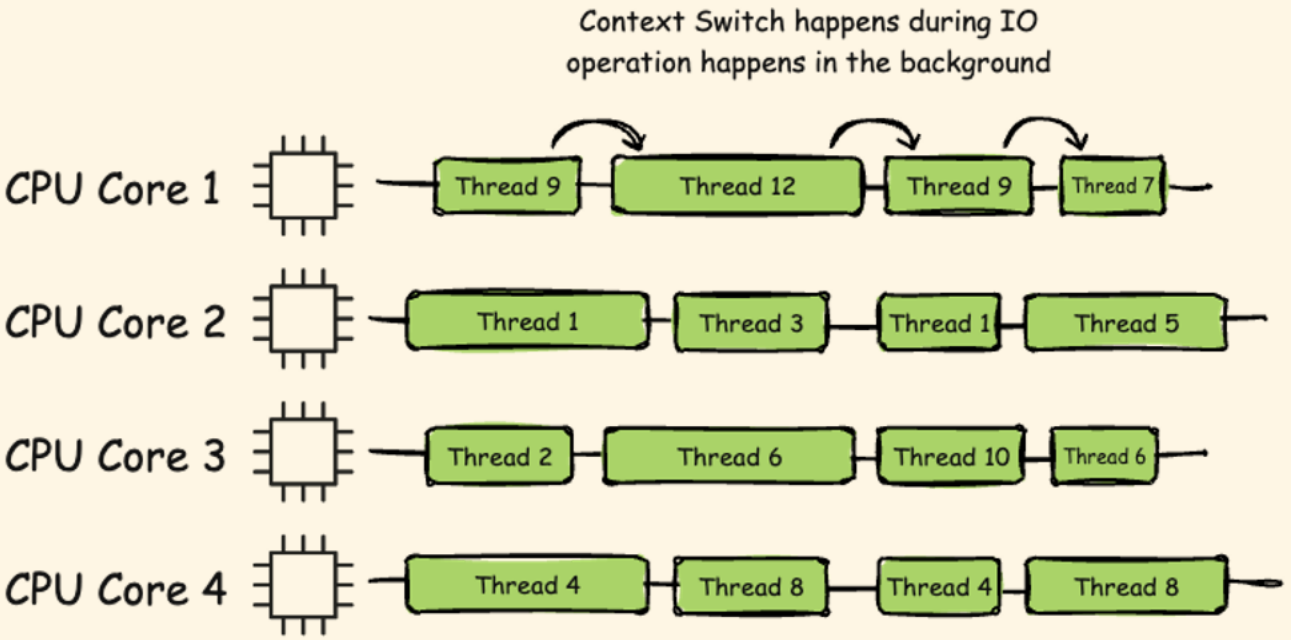


When we have IO operations involved and choose exactly same number of Threads as CPU cores, we will end up wasting lot of CPU power as CPU sits idle during the IO operations like shown above.

How multiple threads improve performance ?

eazy bytes

Let's see what happens if we run the same task on 12 threads.



Focusing on the first core, it is clear that as Thread 9 executes, it may encounter an I/O operation causing the CPU to go idle. However, with multiple threads at hand, instead of remaining idle, the CPU swiftly switches to another thread and continues executing it until Thread 9 completes the I/O operation. This way, the full capacity of the CPU can be effectively utilized, optimizing performance with multiple threads.

In our initial arithmetic calculation only task, using more than 4 threads proved counterproductive due to unnecessary context switching, given its highly CPU-bound nature. However, in the current scenario with 12 threads, efficiency is improved. This underscores the importance of judiciously choosing the number of threads based on the frequency and duration of IO operations. The optimal number of threads is directly linked to the quantity of IO operations in the system. Higher IO demands warrant a greater number of threads to enhance efficiency.

Creating Threads in Java

eazy bytes

The Java API simplifies the process of working with threads by allowing you to represent a thread as an object. In Java, a thread is represented by an object of the `java.lang.Thread` class. The creation and utilization of a thread are straightforward, involving the creation of a Thread class object and its subsequent use in a program. Let's delve into the fundamental steps of creating a thread in Java, which typically encompass at least two essential steps.

```
public class ThreadDemo {

    public static void main(String[] args) {
        // Creates a thread object
        Thread simpleThread = new Thread();
        // Starts the thread
        simplestThread.start();
    }

}
```

Upon running the `ThreadDemo` class, no visible output is observed as the program starts and concludes silently. Despite the lack of observable output, several actions take place within the Java Virtual Machine (JVM) when the statements in the `main()` method are executed.

- Upon executing the statement `simpleThread.start()`, the JVM schedules the thread for execution. Subsequently, when the thread acquires CPU time, it commences execution. In Java, a thread always initiates its execution in a `run()` method. In this context, the `run()` method of the `Thread` class is invoked when an object of the `Thread` class is created using its no-args constructor, as in `new Thread()`.

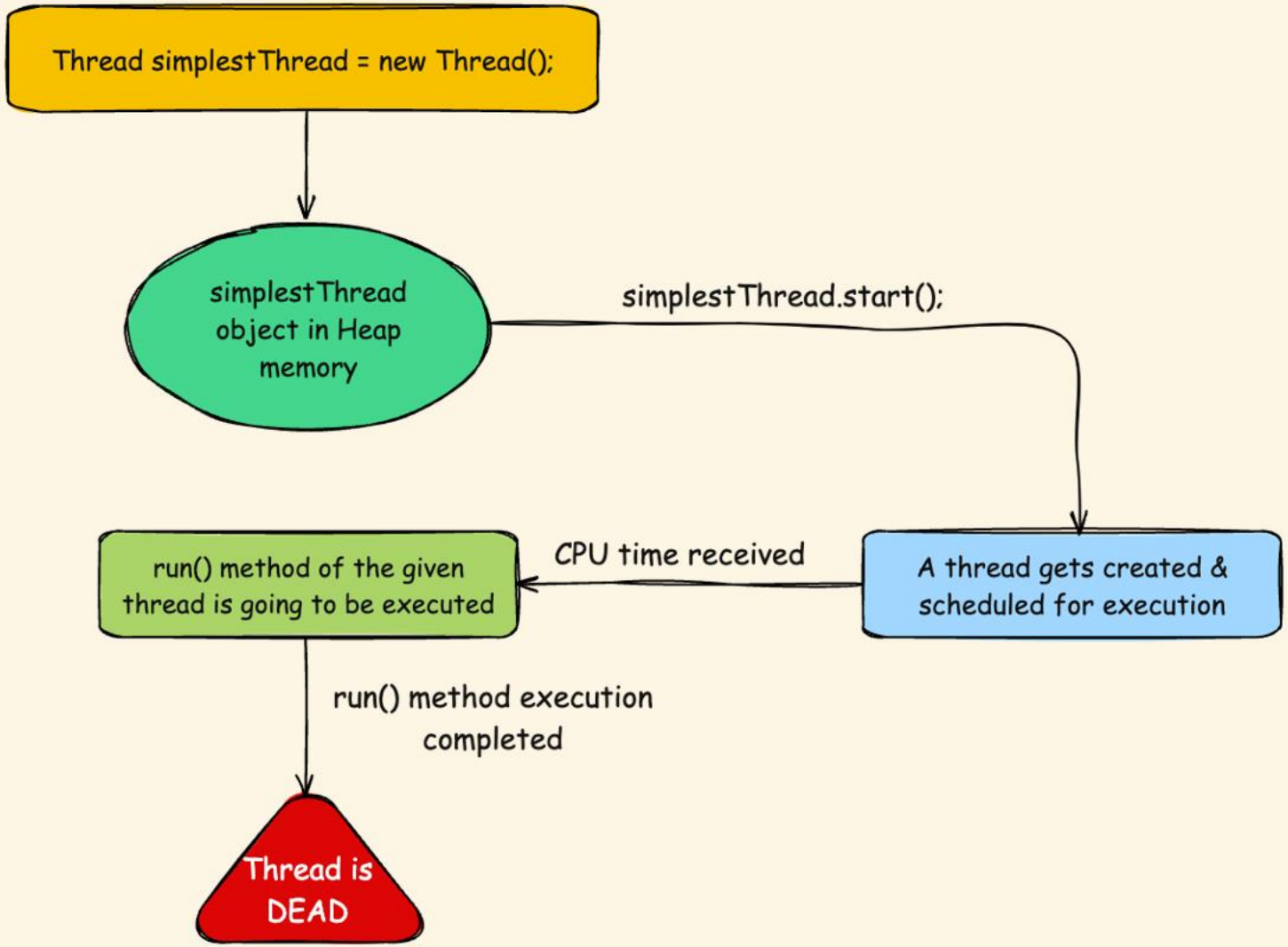
- It is important to note that the `run()` method of the `Thread` class, when created with the no-args constructor, performs no meaningful operations and promptly returns. Consequently, when the thread obtains CPU time in your program, it calls the `run()` method of the `Thread` class, which, in turn, does not execute any substantive code and concludes.

- Once the CPU completes the execution of the `run()` method, the thread is considered dead, indicating that it will not receive CPU time again.

Creating Threads in Java

eazy bytes

The flow chart of Thread execution,



Creating Threads in Java

eazy bytes

There are three approaches to specify code for execution by a thread:

1. By deriving your class from the Thread class

```
public class MyHelloThread extends Thread {

    @Override
    public void run() {
        System.out.println("Hello from Java threads!");
    }

}
```

If your class already inherits from another class, inheriting from the Thread class may not be feasible. In such cases, the second method, involving the implementation of the Runnable interface, becomes necessary. However, it's worth noting that the steps to create a thread object and initiate the thread remain consistent across both methods.

Creating Threads in Java

eazy bytes

2

By implementing the Runnable interface in your class. You have the option to create a class that implements the `java.lang.Runnable` interface, which is a functional interface declared in the `java.lang` package.

```
@FunctionalInterface
public interface Runnable {
    void run();
}

public class MyHelloThread implements Runnable {

    @Override
    public void run() {
        System.out.println("Hello from Java threads!");
    }

}
```

Alternatively, you can utilize a lambda expression to instantiate the `Runnable` interface.

```
Runnable runnableObject = () ->
    System.out.println("Hello from Java threads!");

Thread myThread = new Thread(runnableObject);
```

Creating Threads in Java

eazy bytes

3 For added conciseness, we can employ a method reference, either static or instance, that takes no parameters and returns void as the code to be executed by a thread. The subsequent code introduces a Hello class featuring an sayHello() method, encapsulating the code intended for execution within a thread.

```
public class Hello {
    public static void sayHello() {
        System.out.println("Hello from Java threads!");
    }
}
```

Below code uses the method reference of the sayHello() method of the Hello class to create a Runnable object:

```
Thread methRefThread = new Thread(Hello::sayHello);
methRefThread.start();
```

Sample multi threading program

eazy bytes

Below is an example to perform the addition of all int numbers using 2 threads,

```java
public class SumThread extends Thread{
    private int startIndex;
    private int endIndex;
    private long result;

    public SumThread(int startIndex, int endIndex) {
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }

    @Override
    public void run() {
        for (long i = startIndex; i <= endIndex; i++) {
            result += i;
        }
    }

    public long getResult() {
        return result;
    }
}
```

```java
public class ThreadDemo {

    public static void main(String[] args) {

        // Create and start two threads
        SumThread thread1 = new SumThread(0, Integer.MAX_VALUE / 2);
        SumThread thread2 = new SumThread((Integer.MAX_VALUE / 2)+1,
                                          Integer.MAX_VALUE);

        thread1.run();
        thread2.run();

        // Wait for both threads to finish
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        totalSum = thread1.getResult() + thread2.getResult();
        System.out.println(totalSum);
    }

}
```

Thread methods

eazy bytes

Thread class in Java provides several useful methods, and I will briefly introduce those that commonly appear in programs.

Thread.getId/getName, threadId and Thread.setName

The JVM assigns a number (identifier) to each thread for identification purposes. Additionally, the JVM automatically names threads, or you can assign names manually. To retrieve the thread identifier and name, you can use the `Thread.getId()` and `Thread.getName()` methods. getId() method is deprecated from Java 19, instead use the threadId() method. These methods are valuable for understanding and identifying the threads running in your program.

```
Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = new Thread();
t3.setName("MyThread");

long t1Id = t1.getId();
long t2Id = t2.getId();
long t3Id = t3.getId();
String t1Name = t1.getName();
String t2Name = t2.getName();
String t3Name = t3.getName();

System.out.println("Thread 1 ID is " + t1Id + ", name is " + t1Name);
System.out.println("Thread 2 ID is " + t2Id + ", name is " + t2Name);
System.out.println("Thread 3 ID is " + t3Id + ", name is " + t3Name);
```

Output

```
Thread 1 ID is 23, name is Thread-0
Thread 2 ID is 24, name is Thread-1
Thread 3 ID is 25, name is MyThread
```

Thread methods

eazy bytes

Thread.currentThread

As demonstrated earlier, when you inherit from the Thread class, you can directly call getId() or getName(). However, for Runnables and methods invoked from threads, determining the currently running thread isn't straightforward. This is because Thread.getId() or Thread.getName() cannot be directly called without obtaining the current Thread instance. In such cases, it is necessary to fetch the instance of the currently running Thread first, and then use getId() or getName() to identify the thread.

```
Thread currentThread = Thread.currentThread();
long id = currentThread.getId();
String name = currentThread.getName();

System.out.println("Thread ID is " + id + ", name is " + name);
```

Output

```
Thread ID is 1, name is main
```

Thread methods

eazy bytes

Thread.sleep

When you need to temporarily halt the execution of a thread to wait for a certain condition, you can use the `Thread.sleep` method. Invoking `Thread.sleep` causes the currently running thread to pause for the specified duration, measured in milliseconds. The method also offers an overridden version, `Thread.sleep(long millis, int nanos)`, which allows specifying the duration in nanoseconds.

It's essential to note that only the thread executing the `Thread.sleep` method is paused, while other threads continue their execution unaffected. Additionally, it's not possible to directly halt the execution of one thread from another thread; each thread independently manages its own execution flow.

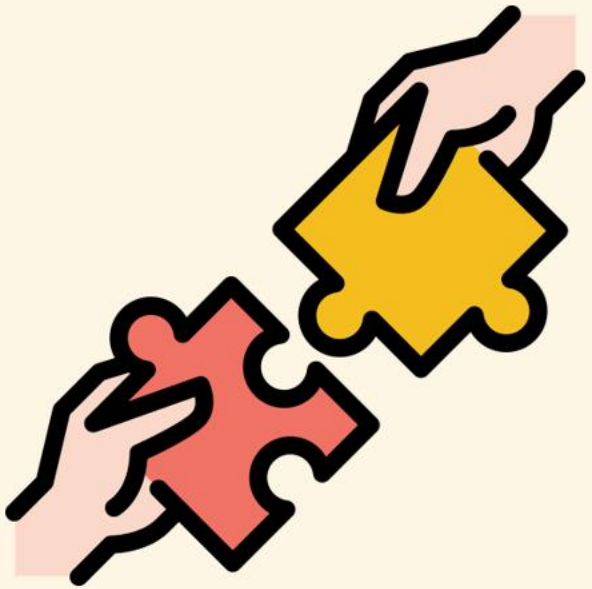
```java
public class SleepExample {
    public static void main(String[] args) {
        System.out.println("Program started...");

        for (int i = 1; i <= 5; i++) {
            try {
                // Pause the main thread for 1 second
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            System.out.println("Message " + i);
        }

        System.out.println("Program ended.");
    }
}
```

Thread methods

eazy bytes

Thread.join

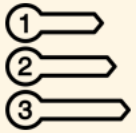
To synchronize the execution of threads and wait for a specific thread to complete its processing, you can use the Thread.join method. In the following example, when t.join() is invoked, the main thread becomes blocked, allowing it to resume only after the thread t has finished its processing.



```java
public class JoinExample {
    public static void main(String[] args) {
        System.out.println("Main thread started...");

        // Create and start threadA
        Thread threadA = new Thread(() -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Thread A - Count: " + i);
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

        threadA.start();

        try {
            // Wait for threadA to finish
            threadA.join();

        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread ended.");
    }
}
```

Thread methods

eazy bytes



Thread.setPriority, Thread.getPriority

In the Java programming language, a thread's priority is represented by an integer within the range of MIN_PRIORITY (1) to MAX_PRIORITY (10). The larger the integer, the higher the priority. The thread scheduler utilizes these priority values to determine the order in which threads are allowed to execute. Generally, the highest priority thread is given preference for execution. However, to prevent starvation, the scheduler might occasionally choose lower-priority threads. In cases where two threads share the same priority, the Java Virtual Machine (JVM) executes them in a First-In-First-Out (FIFO) order.

The `setPriority()` method of the Thread class is employed to modify a thread's priority. By default, a new thread is assigned the priority of NORM_PRIORITY (5). This method allows developers to adjust the priority based on the specific requirements of their application.

Additionally, the `getPriority()` method of the Thread class can be used to retrieve the current priority of a thread. This method provides a means to dynamically query the priority setting, facilitating runtime adjustments or monitoring of thread priority levels.

```
class PriorityExample implements Runnable {
    private final String name;

    public PriorityExample(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(name + " - Count: " + i + ", Priority: " +
                Thread.currentThread().getPriority());
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
public class ThreadPriorityDemo {
    public static void main(String[] args) {
        // Create three threads with different priorities
        Thread thread1 = new Thread(new PriorityExample("Thread A"));
        Thread thread2 = new Thread(new PriorityExample("Thread B"));
        Thread thread3 = new Thread(new PriorityExample("Thread C"));

        // Set priorities for the threads
        thread1.setPriority(Thread.MIN_PRIORITY); // Priority 1
        thread2.setPriority(Thread.NORM_PRIORITY); // Priority 5 (default)
        thread3.setPriority(Thread.MAX_PRIORITY); // Priority 10

        // Start the threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

Thread methods

eazy bytes

Thread.wait(), Thread.notify(), Thread.notifyAll()

In Java, the wait() and notify() methods are part of the inter-thread communication mechanism provided by the Object class. They are used to coordinate the execution of threads and ensure proper synchronization.

wait() Method:

The wait() method causes the current thread to release the lock on the object it is called on and enters a waiting state. It is often used in conjunction with a loop and a condition to wait until a specific condition is met. A thread must own the object's monitor (lock) to invoke wait(). Otherwise, an IllegalMonitorStateException is thrown.

notify() Method:

The notify() method wakes up one of the threads that are currently in the waiting state for the object's monitor. It does not release the lock immediately; the notified thread will compete for the lock before it can proceed. Like wait(), notify() must be called from within a synchronized block, and the calling thread must own the object's monitor.

notifyAll() Method:

The notifyAll() method wakes up all threads that are currently in the waiting state for the object's monitor. This is often used to ensure that all waiting threads have a chance to proceed.

Thread methods

eazy bytes

Consider a scenario where you have a shared resource (e.g., a buffer) between two threads, a producer and a consumer. The wait() and notify() methods can be used to ensure that the consumer waits when the buffer is empty and the producer notifies the consumer when it adds an item to the buffer.

```java
public class SharedResource {

    private boolean isEmpty = true;
    private int data;

    // Producer method
    synchronized void produce(int value) {
        while (!isEmpty) {
            try {
                // Buffer is not empty, wait for the consumer to consume
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Produce an item
        data = value;
        isEmpty = false;
        System.out.println("Produced: " + value);

        // Notify the waiting consumer
        notify();
    }

    // Consumer method
    synchronized int consume() {
        while (isEmpty) {
            try {
                // Buffer is empty, wait for the producer to produce
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        // Consume the item
        int consumedData = data;
        isEmpty = true;
        System.out.println("Consumed: " + consumedData);

        // Notify the waiting producer
        notify();

        return consumedData;
    }
}
```

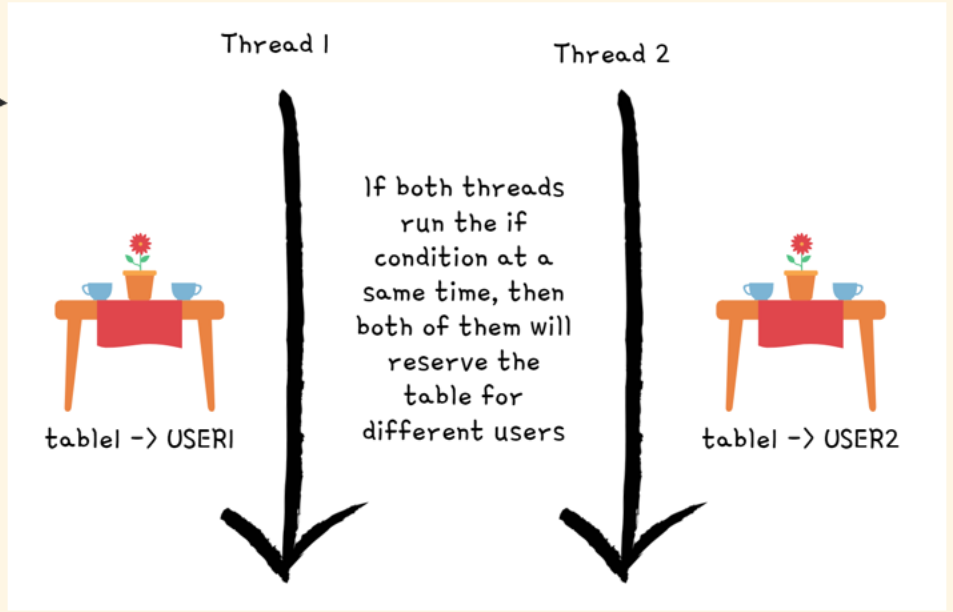
```java
public class ProducerConsumerExample {

    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource();

        Thread producerThread = new Thread(
                () -> {
                    for(int i = 1; i<=10; i++){
                        sharedResource.produce(i);
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                }
        );
        producerThread.start();

        Thread consumerThread = new Thread(
                () -> {
                    for(int i = 1; i<=10; i++){
                        sharedResource.consume();
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            throw new RuntimeException(e);
                        }
                    }
                }
        );
        consumerThread.start();
    }

}
```

Race condition

eazy bytes

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable. The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

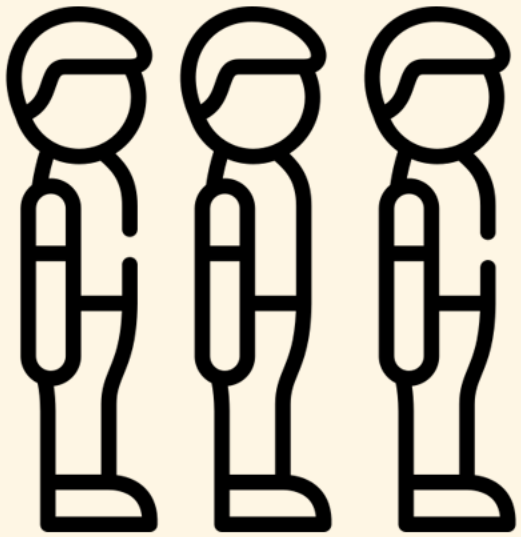
```
// Shared Value inside a object
Map<String, String> reservedTables = new HashMap<>();
// thread -1
if (!reservedTables.containsKey("table1")){
    reservedTables.put("table1", "USER1");
}
// thread - 2
if (!reservedTables.containsKey("table1")){
    reservedTables.put("table1", "USER2");
}
```



Synchronized keyword

eazy bytes

In Java, the synchronized keyword is used to control the access of multiple threads to shared resources, ensuring that only one thread can execute a synchronized block or method at a time. This is crucial for preventing race conditions and maintaining data consistency in multithreaded programs.



Synchronized Block:

```
synchronized (object) {
        // Code that needs to be synchronized
}
```

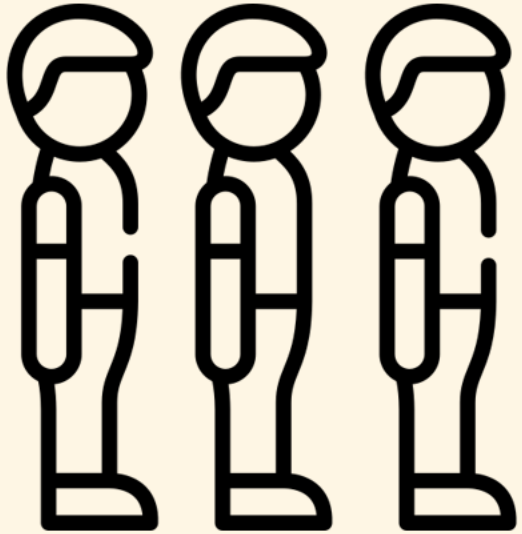
The synchronized block ensures that only one thread at a time can execute the code within the block.

object is the monitor (lock) on which synchronization is applied. It can be any object, often the shared resource itself.

Synchronized keyword

eazy bytes

The synchronized keyword can also be applied to entire methods



Synchronized Method:

```
public synchronized void synchronizedMethod() {
   // Code that needs to be synchronized
}
```

When a method is declared as synchronized, it implicitly uses the object instance (for non-static methods) or the class object (for static methods) as the monitor.

You can declare both an instance method and a static method as synchronized. A constructor cannot be declared as synchronized. A constructor is called only once by only one thread, which is creating the object. So it makes no sense to synchronize access to a constructor.

Counter example with Unsynchronized method

eazy bytes

Consider a simple Counter class where multiple threads increment a shared counter. Without synchronization, race conditions may occur and lead to incorrect results.

```java
public class Counter {

    private int count = 0;

    // Unsynchronized method
    public void incrementUnsynchronized() {
        count++;
    }

    public int getCount() {
        return count;
    }

}
```

Multiple threads increment the counter using unsynchronized methods. Without synchronization, the unsynchronized count may produce incorrect results due to race conditions.

```java
public class UnSynchronizationExample {

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Unsynchronized increment
        Runnable unsynchronizedTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.incrementUnsynchronized();
            }
        };

        // Create multiple threads for each task
        Thread unsynchronizedThread1 = new Thread(unsynchronizedTask);
        Thread unsynchronizedThread2 = new Thread(unsynchronizedTask);

        // Start the threads
        unsynchronizedThread1.start();
        unsynchronizedThread2.start();

        // Wait for threads to complete
        unsynchronizedThread1.join();
        unsynchronizedThread2.join();

        // Print final counts
        System.out.println("Unsynchronized Count: " + counter.getCount());

    }
}
```

Counter example with synchronized method

eazy bytes

Consider a simple Counter class where multiple threads increment a shared counter. With synchronization, race conditions will not occur and always lead to correct results.

```java
public class Counter {

    private int count = 0;

    // Synchronized method
    public synchronized void incrementSynchronized() {
        count++;
    }

    public int getCount() {
        return count;
    }

}
```

Synchronization is crucial when dealing with shared resources to avoid data corruption or inconsistencies caused by multiple threads accessing the resource concurrently. However, excessive use of synchronization can lead to performance issues, so it should be applied judiciously based on the specific requirements of the application.

```java
public class SynchronizationExample {

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        // Synchronized increment
        Runnable synchronizedTask = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.incrementSynchronized();
            }
        };

        // Create multiple threads for each task
        Thread synchronizedThread1 = new Thread(synchronizedTask);
        Thread synchronizedThread2 = new Thread(synchronizedTask);

        // Start the threads
        synchronizedThread1.start();
        synchronizedThread2.start();

        // Wait for threads to complete
        synchronizedThread1.join();
        synchronizedThread2.join();

        // Print final counts
        System.out.println("Synchronized Count: " + counter.getCount());
    }
}
```

volatile keyword

If you have a variable, such as a boolean flag or counter, is being operated on by a thread, there's a possibility that the thread maintains a local copy of the variable in the CPU cache and performs manipulations on it without immediately updating the main memory. The decision of when to synchronize the local copy with the main memory is left to the JVM. This approach may result in other threads reading stale values of the variable.

Characteristics of volatile:

Visibility: When a variable is declared as volatile, any write to that variable is immediately visible to other threads.

No Caching: Threads do not cache volatile variables. Each thread reads the value directly from the main memory.

volatile keyword

eazy bytes

Consider a scenario where multiple threads are involved in checking and updating a shared flag variable. Using volatile ensures that changes to the flag are immediately visible to all threads.

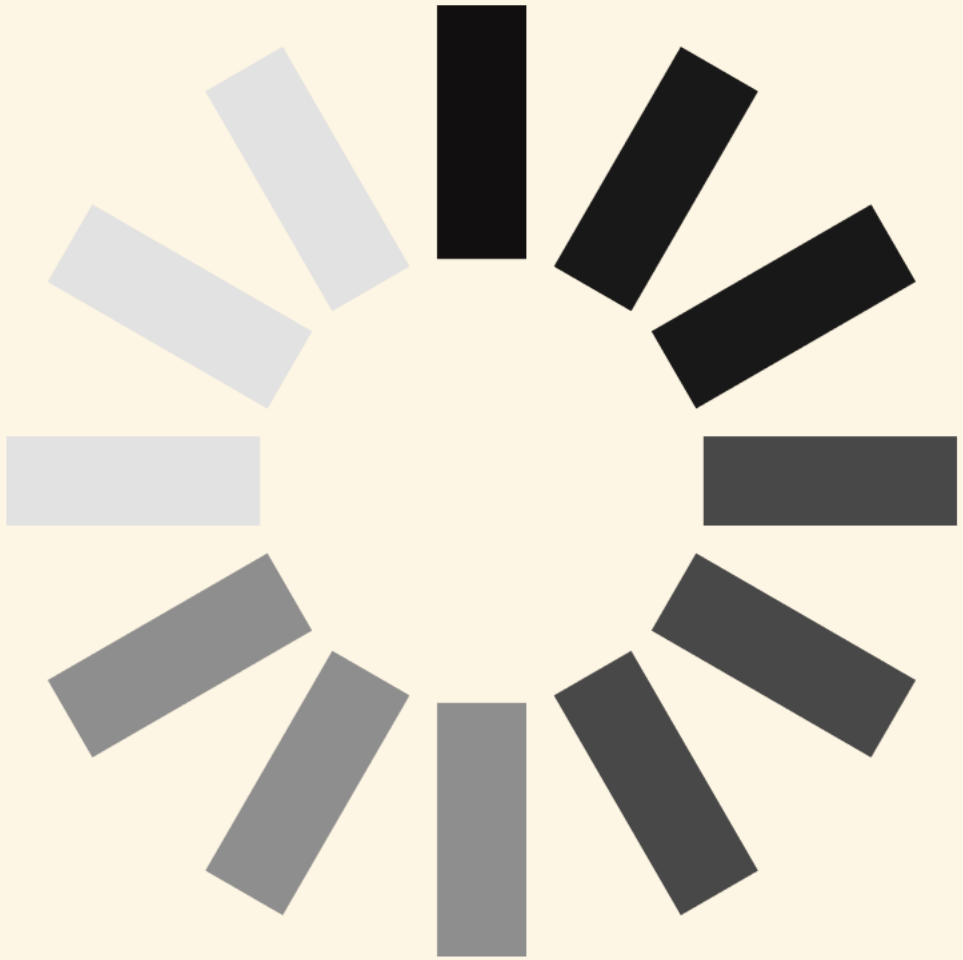
```java
public class SharedBooleanResource {

    private volatile boolean stopFlag = false;

    public void setStopFlag () {
        stopFlag = true;
    }

    public void doWork () {
        while(!stopFlag){
            System.out.println("Working...");
        }
        System.out.println("Work stopped");
    }
}
```

```java
public class VolatileExample {

    public static void main(String[] args) throws InterruptedException {
        SharedBooleanResource sharedBooleanResource = new SharedBooleanResource();
        Thread workerThread = new Thread(() -> sharedBooleanResource.doWork());

        Thread stopperThread = new Thread(() -> {
            try {
                Thread.sleep(3000);
                sharedBooleanResource.setStopFlag();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }) ;
        workerThread.start();
        stopperThread.start();
        workerThread.join();
        stopperThread.join();
    }

}
```

Without the volatile keyword, the stopFlag might not be visible across threads, and the worker thread might not see the updated value, leading to a situation where the work does not stop as expected. Using volatile ensures proper visibility and coordination between threads in this scenario.

Deadlock

eazy bytes

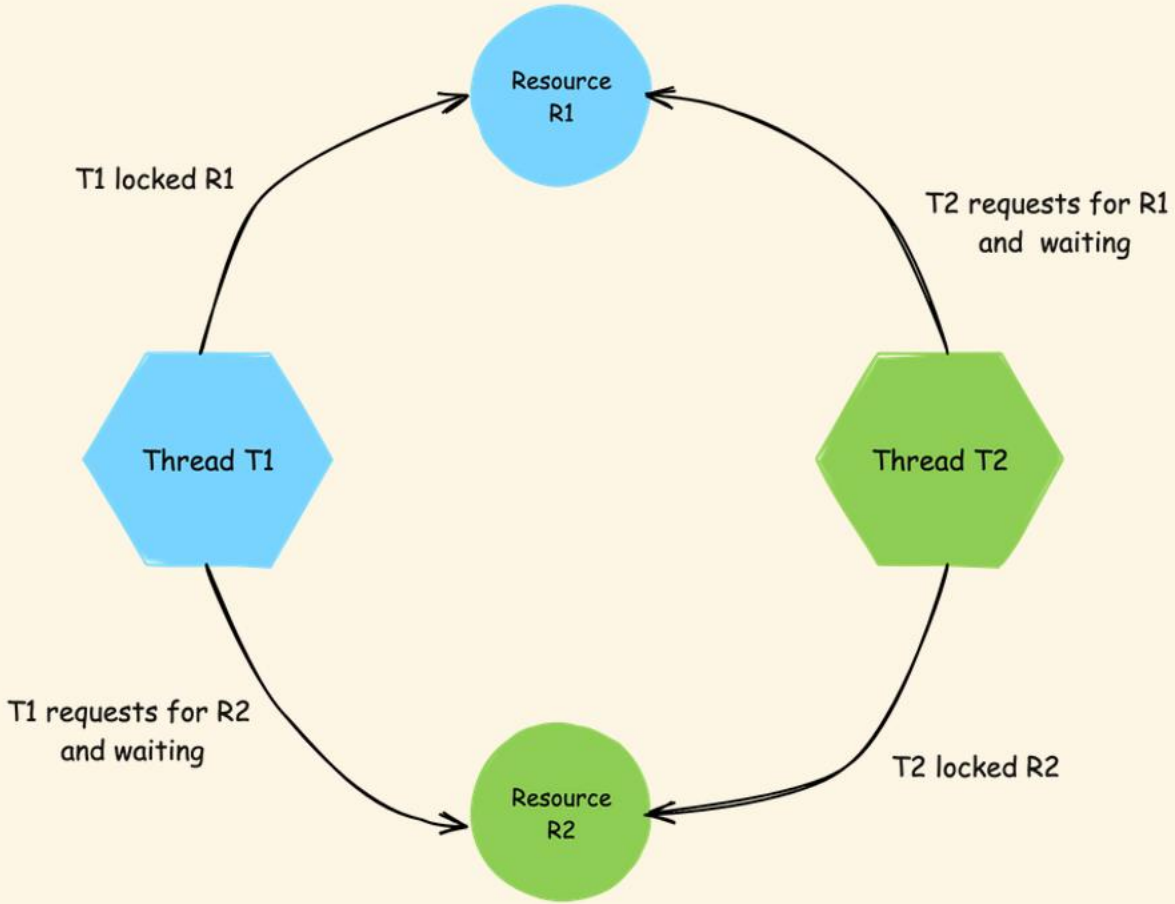


Deadlocks primarily occur in multithreaded programming scenarios. For instance, a deadlock may arise when one thread is waiting to acquire the lock of an object currently held by another thread, and simultaneously, the second thread is waiting to obtain the lock of a different object held by the first thread. Consequently, both threads find themselves in a state of mutual waiting, each expecting the other to release the lock, leading to a deadlock situation.

Deadlock

eazy bytes

Imagine two threads, T1 and T2, each currently holding a specific resource – T1 has R1, and T2 has R2. Now, T1 is in the process of requesting resource R2, while simultaneously, T2 is requesting resource R1. Both threads are unwilling to release their existing resources, R1 and R2, until they complete their execution. This situation sets the stage for a deadlock, as illustrated in the image below.



Deadlock example

eazy bytes

```java
public class DeadLockDemo {

    public static final String R1 = "Hello";
    public static final String R2 = "Hi";

    public static void main(String[] args) {
        // Thread 1
        Thread thread1 = new Thread(() -> {
            synchronized (R1) { // R1 locked by T1
                System.out.println("Thread T1 locked : Resource R1");
                synchronized (R2) { // R2 locked by T1
                    System.out.println("Thread T1 locked : Resource R2");
                }}});

        // Thread 2
        Thread thread2 = new Thread(() -> {
            synchronized (R2) { // R2 locked by T2
                System.out.println("Thread T2 locked : Resource R2");
                synchronized (R1) { // R1 locked by T2
                    System.out.println("Thread T2 locked : Resource R1");
                }}});

        // starting both the threads
        thread1.start();
        thread2.start();
    }

}
```

In the provided code snippet, we initially declare two strings, R_1 and R_2, as resources. Within the main method, we proceed to create two threads, T_1 and T_2. In the run method of thread T_1, we establish a synchronization block to lock resource R_1, and similarly, in the run method of thread T_2, we lock resource R_2.

Within the synchronization block of thread T_1, there is a request for resource R_2, which is concurrently locked by thread T_2. Conversely, in the synchronization block of thread T_2, there is a request for resource R_1, held by thread T_1. Upon starting both threads, it becomes evident that neither thread can complete its execution entirely. Consequently, the program enters an infinite loop, unable to progress further.

We can fix the Deadlock issue by making changes inside Thread 2 such a way that, T_2 will try to acquire R_1 first followed by R_2. Thus we can always reduce the circular dependency between T_1 and T_2.

How to avoid Deadlock

eazy bytes

Avoiding deadlocks in Java involves careful design and implementation of concurrent programs. Here are some strategies to prevent deadlocks:

Avoid Nested Locks

One of the most important reasons for deadlock in Java is nested locks. So if a thread already has a lock, try not to acquire another nested lock.

Avoid Unnecessary Locks

Only apply locks to the members that truly require them. Unnecessarily using locks on non-critical sections can also contribute to the likelihood of encountering a deadlock.

Lock Hierarchy:

Establish a hierarchy for acquiring locks and ensure that all threads follow this hierarchy. This approach helps prevent circular waiting scenarios.

How to avoid Deadlock

eazy bytes

Use java.util.concurrent Utilities

Java provides high-level concurrency utilities in the java.util.concurrent package that can help avoid deadlocks. For example, you can use ReentrantLock to manage threads and locks more efficiently.

Lock Timeout

Use tryLock() method present in ReentrantLock, with a timeout instead of synchronized blocks to acquire locks. This allows a thread to give up on acquiring a lock if it cannot be obtained within a specified time, reducing the chance of deadlocks.

```
ReentrantLock lock1 = new ReentrantLock();

if (lock1.tryLock(100, TimeUnit.MILLISECONDS)) {
    try {
        // Code block
    } finally {
        lock1.unlock();
    }
}
```

Virtual Threads

eazy bytes

From Java 21

Virtual threads in Java mark a noteworthy advancement in the realm of concurrency and multithreading for the language. A pivotal component of Project Loom, Oracle's initiative aimed at tackling the complexities associated with developing, maintaining, and monitoring high-throughput concurrent applications, virtual threads stand out for their lightweight nature. They are specifically crafted to simplify the intricacies of concurrency for developers.

Types of Threads in OS

eazy bytes

Prior to delving into Virtual Threads, understanding the various types of threads and their internal functioning is crucial.

Modern operating systems typically feature two types of threads: Kernel-level Threads and User-level Threads.

Kernel Threads

This is alternatively referred to as an OS Thread. Kernel Threads are overseen and scheduled by the operating system kernel. These threads are relatively heavyweight in terms of resources and require system calls for their creation, scheduling, and synchronization.

User Threads

User-level threads are handled and scheduled by user-level thread libraries like Java without the need for intervention from the operating system kernel. The kernel remains unaware of User threads. These threads are lightweight, enabling faster creation and destruction compared to OS threads.

In simpler terms, when a process initiates, it generates a default thread to run the application's entry point (main method in Java scenario). Following this, the process has the flexibility to generate extra threads as required. The execution of user threads is indirect; they must be mapped to a Kernel thread. Consequently, it is the Kernel thread that ultimately carries out every instruction in the computer system.

Thread Models

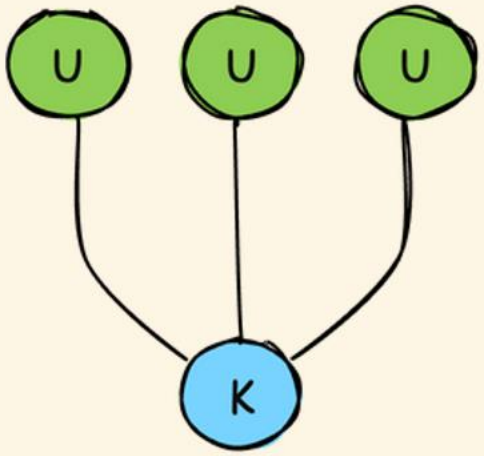
eazy bytes

The association between User threads and Kernel threads can take one of three forms:

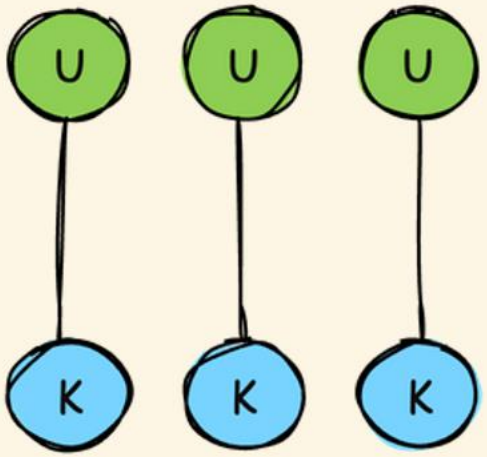
M:1 model: All user threads are linked to a single kernel thread.

1:1 model: Each user thread is associated with one kernel thread.

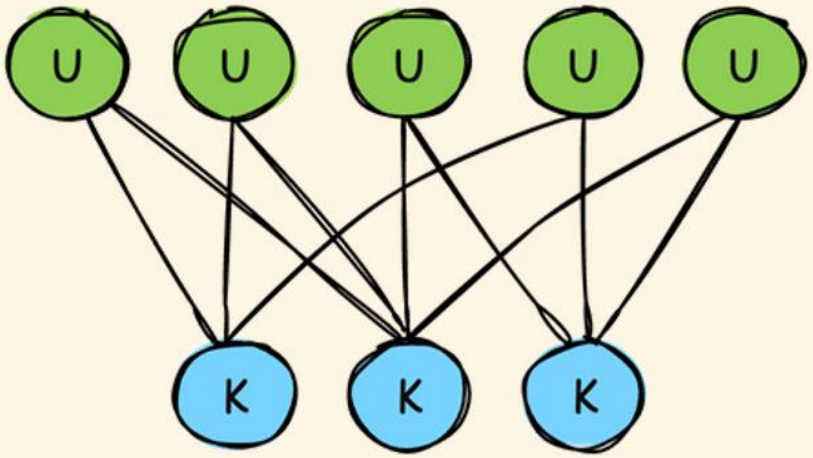
M:N model: All user threads are mapped to a pool of kernel threads.



Many-to-One Thread Model



One-to-One Thread Model

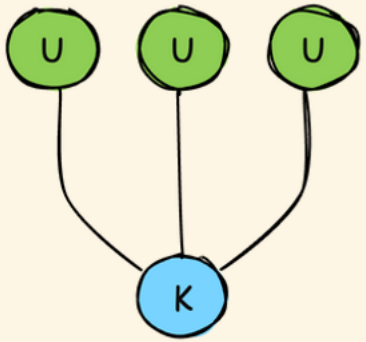


Many-to-Many Thread Model

Green Thread

eazy bytes

< Java 1.2



Many-to-One Thread Model

In its initial versions, Java adopted the Green Thread threading model, where threads were directly overseen and scheduled by the Java Virtual Machine (JVM).

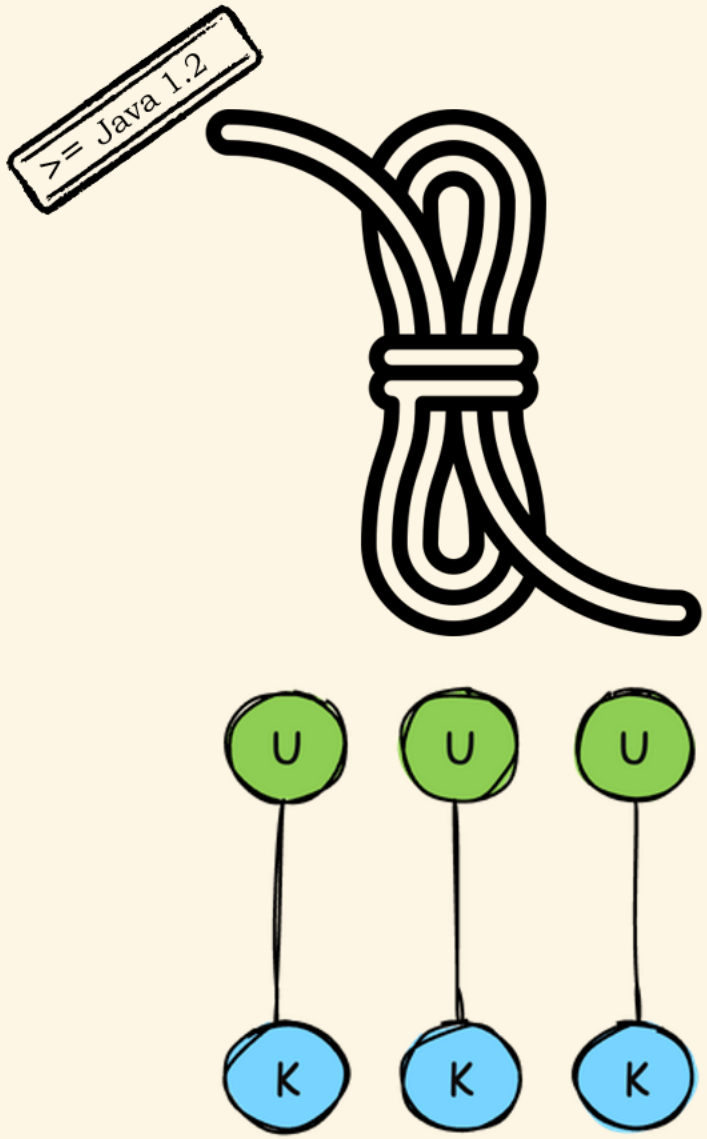
In this approach, the green thread model utilized M:1 thread mapping, meaning multiple user threads were managed by a single kernel thread. Green threads proved to be notably faster than native threads, but a drawback emerged – Java couldn't effectively scale across multiple processors, limiting its ability to leverage multiple cores.

One significant challenge with green threads was their difficulty in implementation within libraries due to the need for very low-level support to ensure optimal performance.

As a response, Java eventually transitioned away from green threading and embraced native threading, a move that, however, resulted in Java Threads becoming somewhat slower compared to green threads.

Native/Platform Thread

eazy bytes



One-to-One Thread Model

Since version 1.2, Java discontinued support for the Green thread model and transitioned to the Native thread model. In this approach, Native threads are managed by the Java Virtual Machine (JVM) with the assistance of the underlying operating system.

Native threads exhibit high efficiency during execution but incur a notable cost when starting and stopping. To mitigate this, contemporary practices often involve thread-pooling.

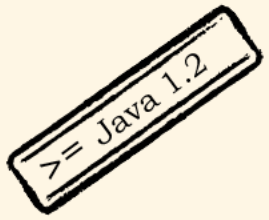
Following a 1:1 thread mapping, each Java thread corresponds to a distinct Kernel thread. When a Java thread is instantiated, the operating system creates a corresponding native thread responsible for executing the thread's code.

The adoption of the native thread model has persisted in Java since version 1.2, maintaining its relevance and effectiveness in concurrent programming.

Java followed the native thread model ever since 1.2 and continues to do so till Java 21. Let's try to understand why Virtual Threads came into picture to replace Native Threads

What's wrong with Native Threads ?

eazy bytes



In the previous discussion, we learned that Java uses the Native-thread model till Java 21. Now, let's explore some drawbacks of this model.

- The Java thread library was initially created in the early versions of Java, serving as a thin wrapper around platform threads, also known as Native Threads.
- However, Native threads come with certain issues. They are expensive to create and maintain, requiring a substantial amount of memory to store their call stack (typically between 1MB to 20MB, depending on the JVM and platform).
- Considering that a system with 8GB of RAM can only accommodate around 800 threads, each consuming 10MB of RAM, it becomes apparent that the memory constraints can limit the number of threads that can be created. Moreover, launching a new Native thread takes approximately 1 millisecond, and context switching in Native threads is also expensive, involving a system call to the kernel.
- These limitations impact the performance and scalability of applications. Due to the significant memory footprint and context-switching costs, creating numerous threads becomes impractical. This hinders the ability to scale applications by adding more threads, ultimately affecting performance.

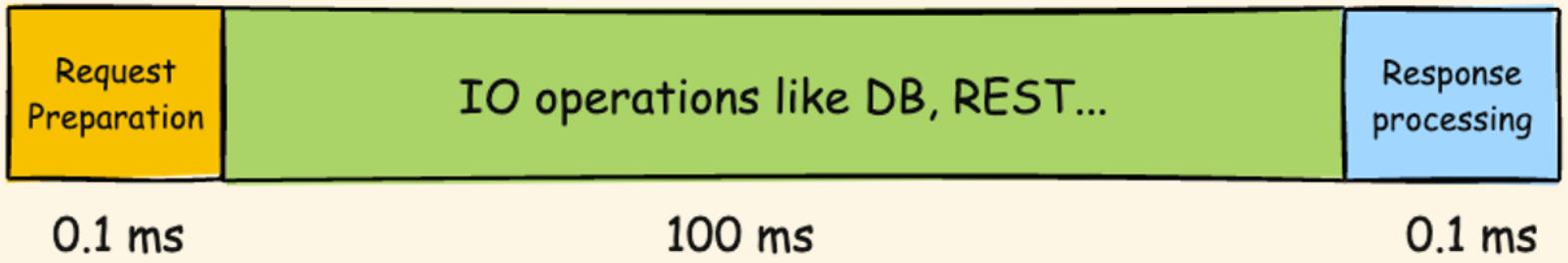
What's wrong with Native Threads ?

eazy bytes

Imagine a web server deployed on a machine boasting 16GB of RAM. Following the common thread-per-request approach typical for web servers, where each user request is handled by a separate thread, the system can potentially support up to 1600 threads given the available 16GB of RAM, with each thread requiring 10MB.

In contemporary backend applications, tasks often involve operations like interacting with databases or making API calls via REST. Consequently, these systems are predominantly IO-bound rather than CPU-bound.

Let's consider a scenario where, for a single request, IO operations take 100 milliseconds, request processing consumes 0.1 milliseconds, and response processing also takes 0.1 milliseconds.



What's wrong with Native Threads ?

eazy bytes

Below is the calculation of the total CPU time for a single request,

```
Total CPU Time = Request preparation time + Response preparation time
               = 0.1 ms + 0.1 ms
               = 0.2 ms
```

If a single request takes 0.2 milliseconds of CPU time. What will be the total CPU time for 1600 requests?

```
Total CPU time for 1600 requests = 1600 * 0.2 milliseconds
                                 = 320 milliseconds
```

Based on the server RAM capacity, in a second our server can handle only 1600 requests because we can create only a maximum of 1600 threads. With a fact of 1 second = 1000 milliseconds, let's calculate the CPU utilization of 1 second:

```
CPU Utilization in a second = 320 milliseconds / 1000 milliseconds
                            = 32%
```

Only 32% of the CPU was utilized in one second, indicating that the CPU is not being optimally utilized and is underused with native or platform threads.

Virtual Threads

eazy bytes

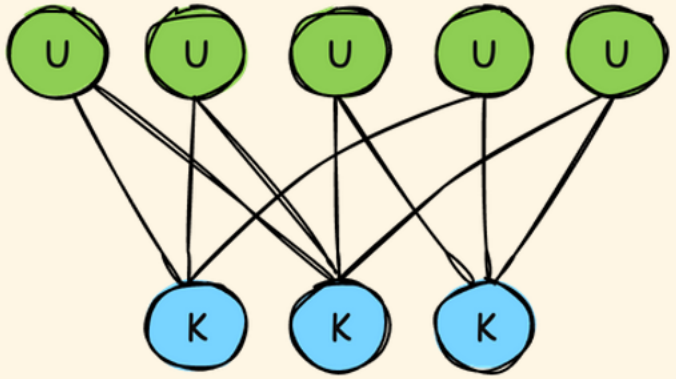
- Virtual threads are notably lightweight, boasting a smaller memory footprint compared to platform threads; each virtual thread consumes only a few bytes of memory. This characteristic makes them particularly well-suited for tasks involving a high volume of client connections or processing operations with significant input/output demands.
- One of the distinct advantages of virtual threads is their scalability, as there is no predefined limit on the number of virtual threads that can be created. In fact, you can generate millions of virtual threads without the need for system calls to the kernel.
- Creating and destroying virtual threads is a swift process, requiring minimal time and resources.

How does the Virtual thread work?

eazy bytes

From Java 21

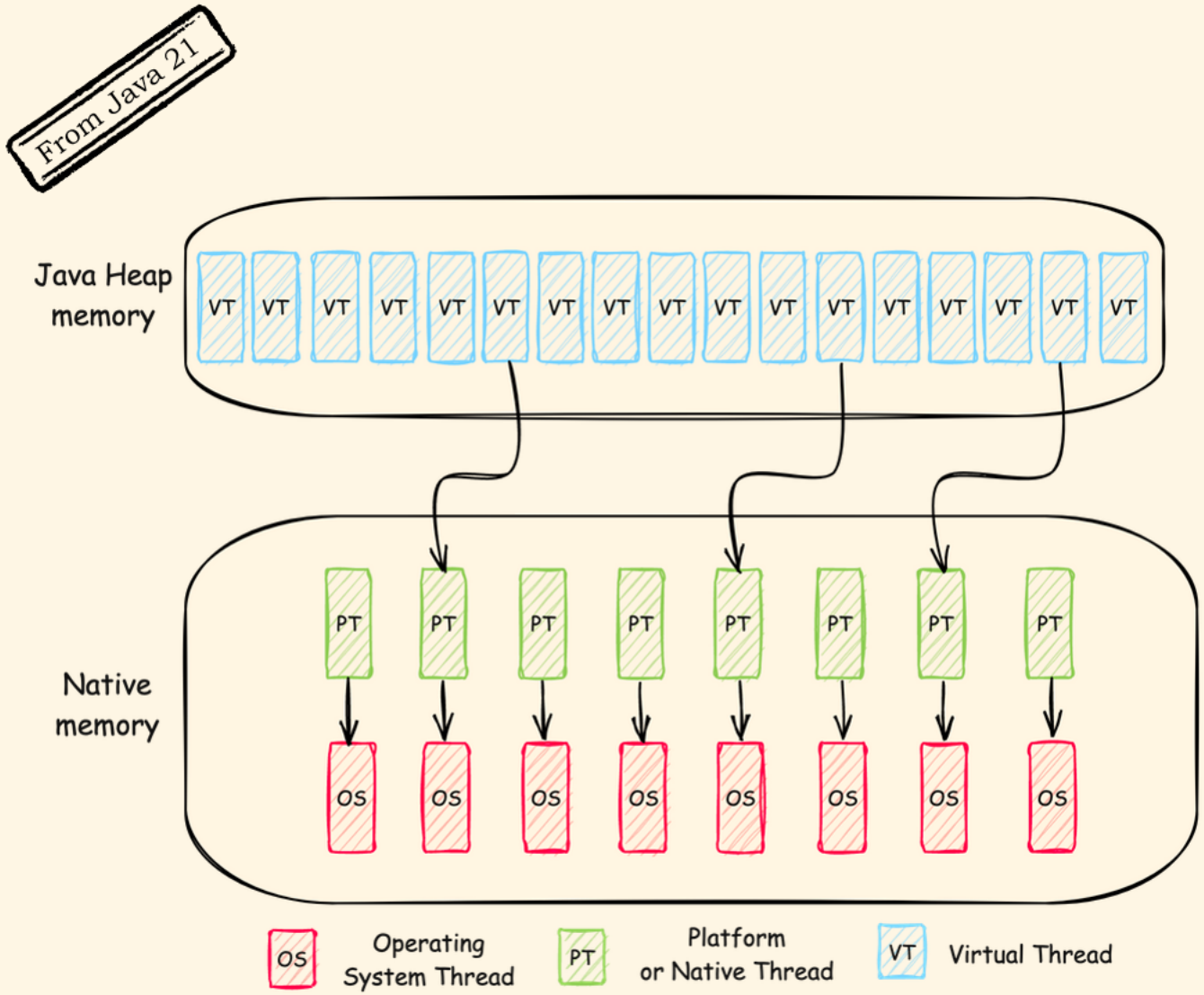
- Virtual threads employ M:N scheduling, where a large number (M) of virtual threads is scheduled to run on a smaller number (N) of OS threads.



Many-to-Many Thread Model

How does the Virtual thread work?

eazy bytes



Virtual threads contribute to enhanced CPU utilization. In conventional thread models, considerable CPU time is expended in the management and context-switching between numerous threads. The introduction of virtual threads mitigates this issue, minimizing the overhead associated with context switching and optimizing the execution of concurrent tasks.

A Virtual Thread while waiting for an IO operation to complete will detach from the Platform Thread. So that the platform threads can be used by other virtual threads which need CPU time.

How to create Virtual Threads ?

eazy bytes

Using Thread.startVirtualThread(Runnable r)

The most basic way to use a virtual thread is with Thread.startVirtualThread(Runnable r). This is a replacement for instantiating a thread and calling thread.start().

```
Runnable runnable = () -> System.out.println("Hello");
Thread.startVirtualThread(runnable);
```

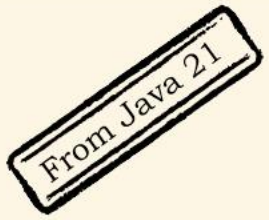
Using Thread.ofVirtual() method

```
Thread thread = Thread.ofVirtual().start(() -> System.out.println("Hello"));
```

Thread.isVirtual() method is also added in Java 21, to identify if a given thread is a Virtual Thread or not.

Sample demo of Virtual Threads

eazy bytes




```java
public class VirtualThreadDemo {

    public static void main(String[] args) throws InterruptedException {
        long startTime = System.currentTimeMillis();
        Random random = new Random();
        Runnable runnable = () -> {
            double result = random.nextDouble(1000) * random.nextDouble(1000);
             System.out.println(result);
        };
        for (int i =0; i<500000; i++) {
                    Thread.ofVirtual().start(runnable).join(); //8828
                    // Thread.startVirtualThread(runnable).join(); // 8844
                  // Thread thread = new Thread(runnable); // 30563
                  // thread.start();
                  // thread.join();
        }
        long endTime = System.currentTimeMillis();
        System.out.println("Total time : " + (endTime-startTime));
    }

}
```