Bài 14

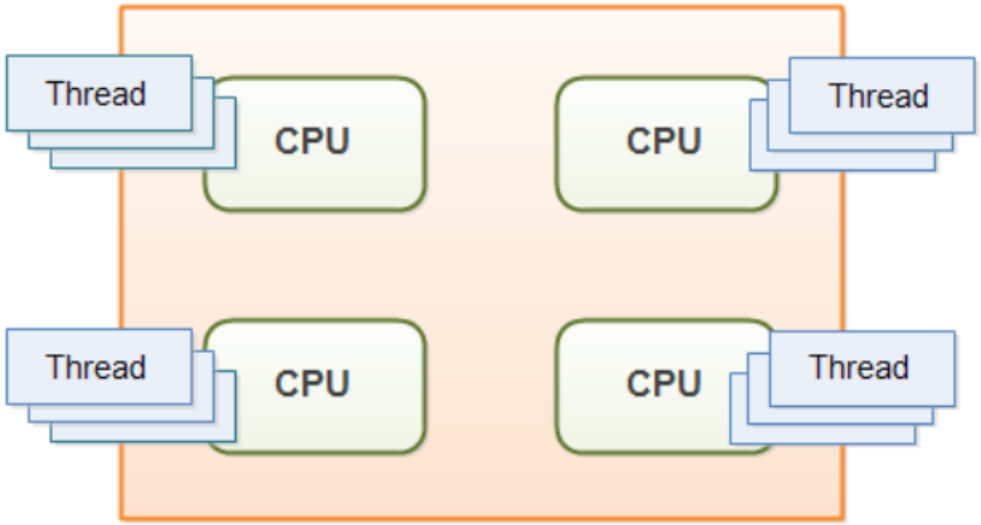
Thread

Thread

Java is a *multi-threaded programming language* which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition, multitasking is when multiple processes share common processing resources such as a CPU. Multi-threading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

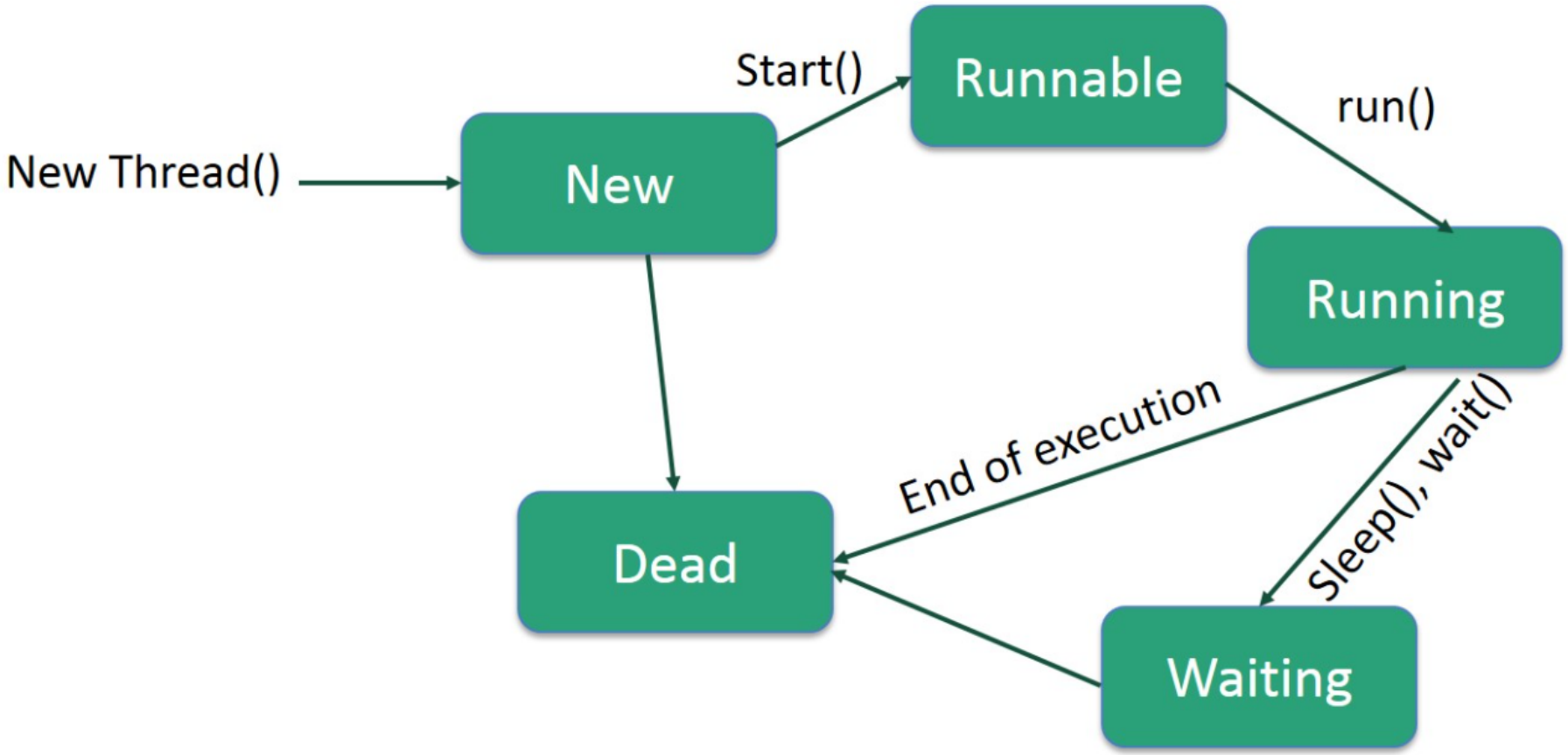
Multi-threading enables you to write in a way where multiple activities can proceed concurrently in the same program.



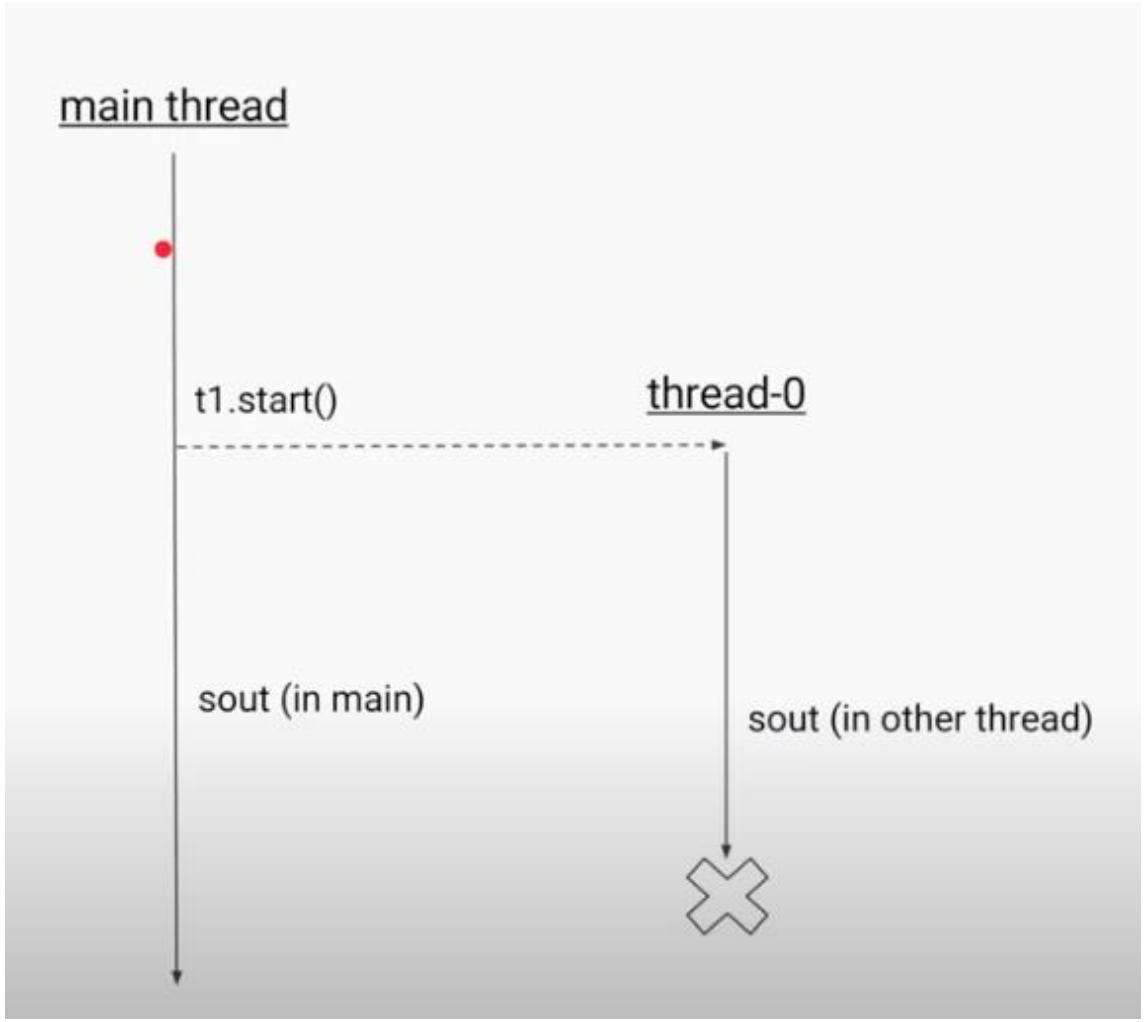
Thread

Life Cycle of a Thread

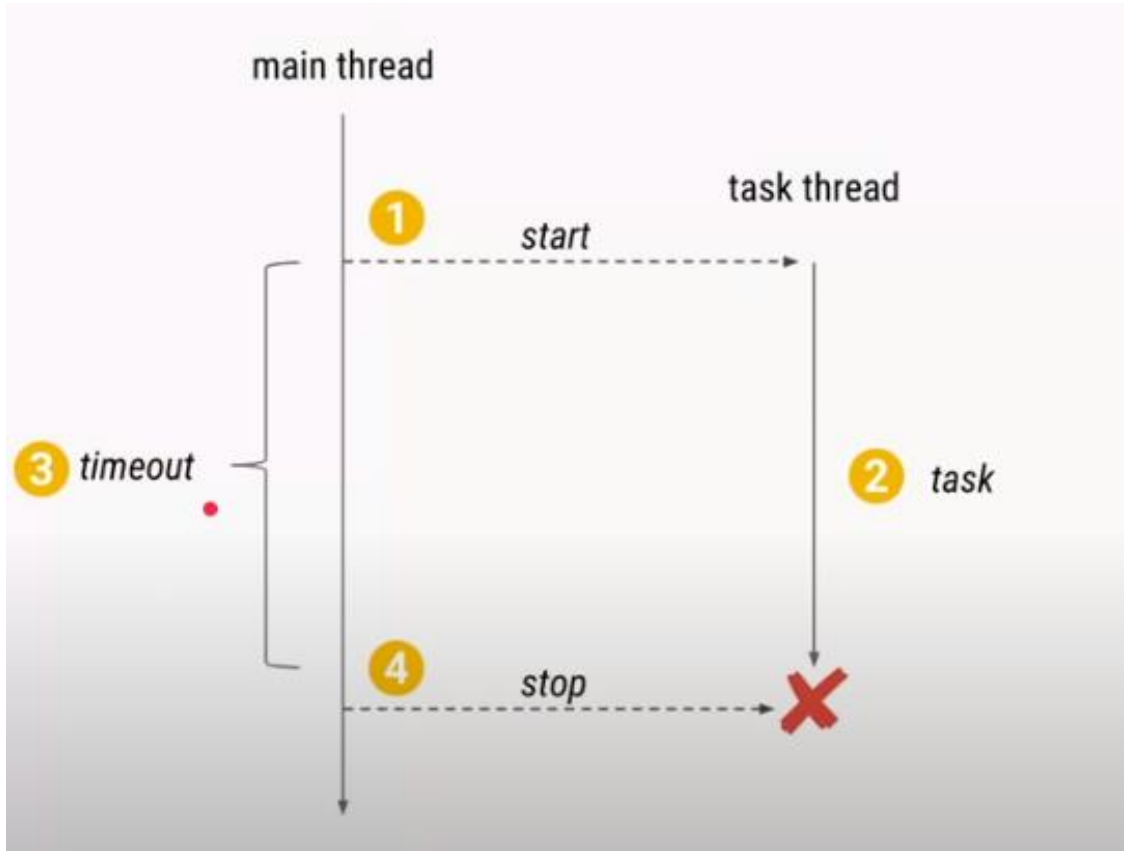
A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



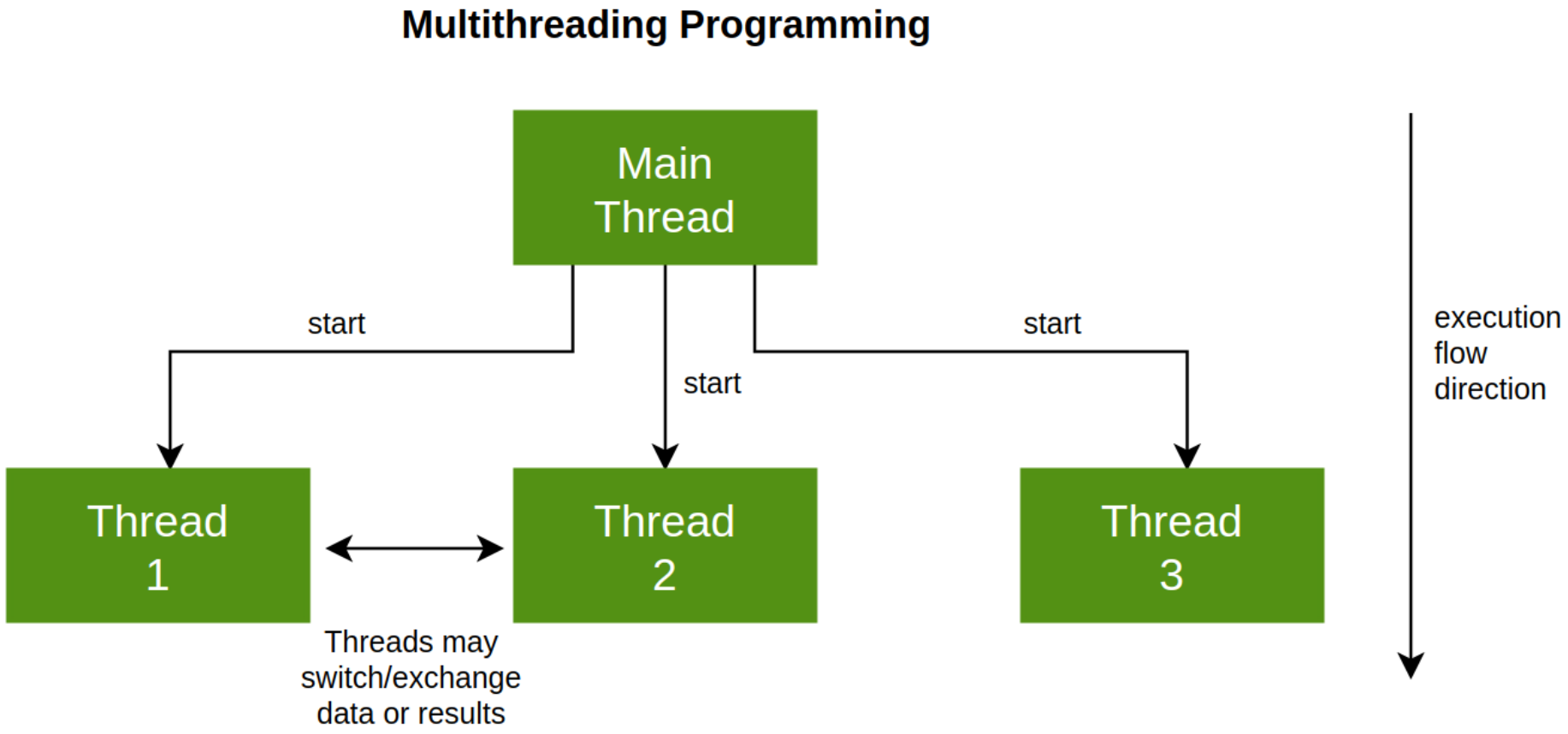
Thread


main thread
t1.start()
thread-0
sout (in main)
sout (in other thread)

Thread



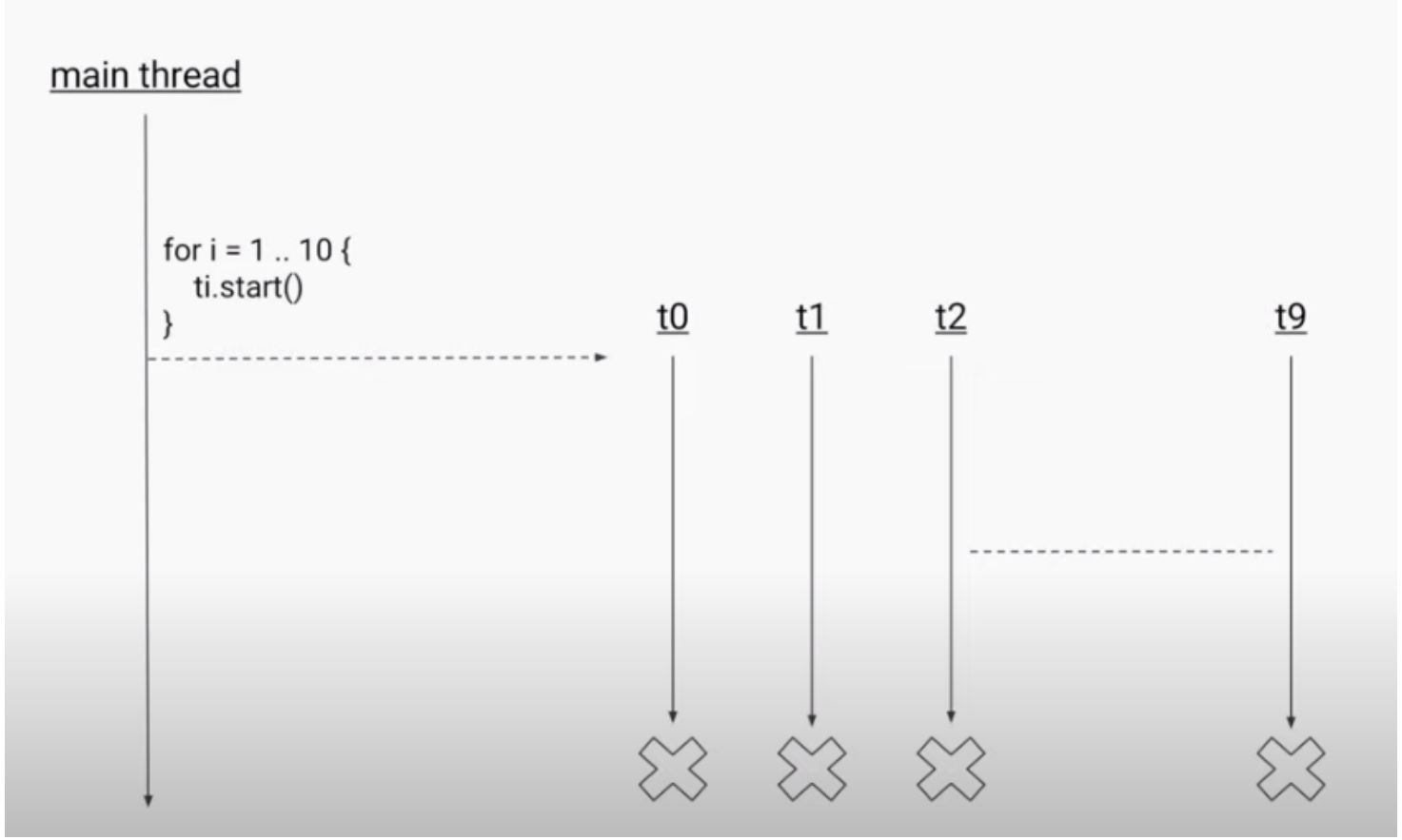
Thread


Multithreading Programming
Main Thread
start
start
start
Thread 1
Thread 2
Thread 3
Threads may switch/exchange data or results
execution flow direction

Thread

```
public static void main(String[] args) {

    Thread thread1 = new Thread(new Task());
    thread1.start();
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
```

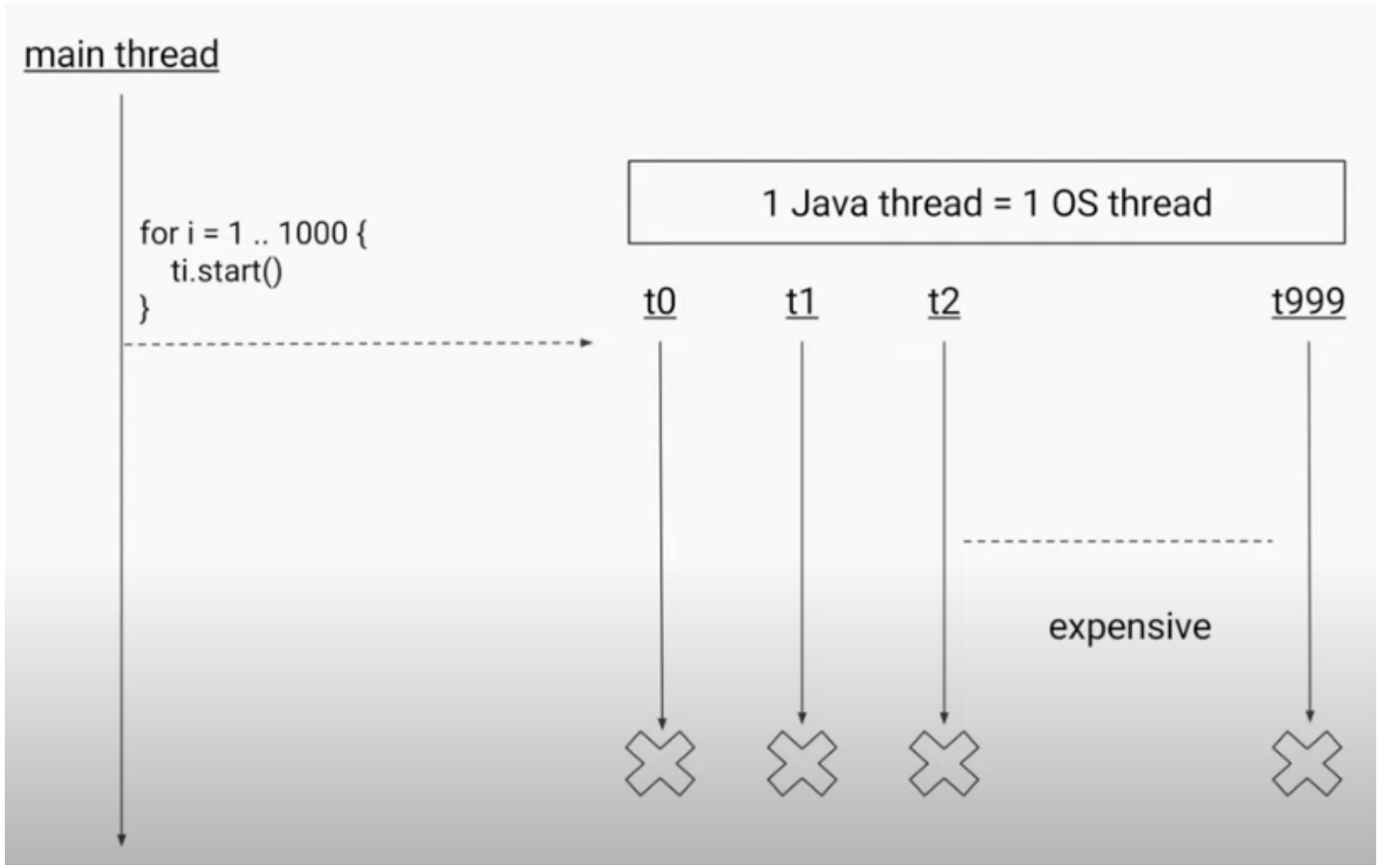
Thread



Thread

```
public static void main(String[] args) {

    for (int i = 0; i < 10; i++) {
        Thread thread = new Thread(new Task());
        thread.start();
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
```

Running 1000 thread asynchronously



Running restrictions

Although this brings several advantages, primarily regarding the performance of a program, the multithreaded programming can also have disadvantages – such as increased complexity of the code, concurrency issues, unexpected results and adding the overhead of thread creation.

Thread Pool

Why Use a Thread Pool?

Creating and starting a thread can be an expensive process. By repeating this process every time we need to execute a task, we're incurring a significant performance cost – which is exactly what we were attempting to improve by using threads.

For a better understanding of the cost of creating and starting a thread, let's see what the JVM actually does behind the scenes:

- it allocates memory for a thread stack that holds a frame for every thread method invocation
- each frame consists of a local variable array, return value, operand stack and constant pool
- some JVMs that support native methods also allocate a native stack
- each thread gets a program counter that tells it what the current instruction executed by the processor is
- the system creates a native thread corresponding to the Java thread
- descriptors relating to the thread are added to the JVM internal data structures
- the threads share the heap and method area

Of course, the details of all this will depend on the JMV and the operating system.

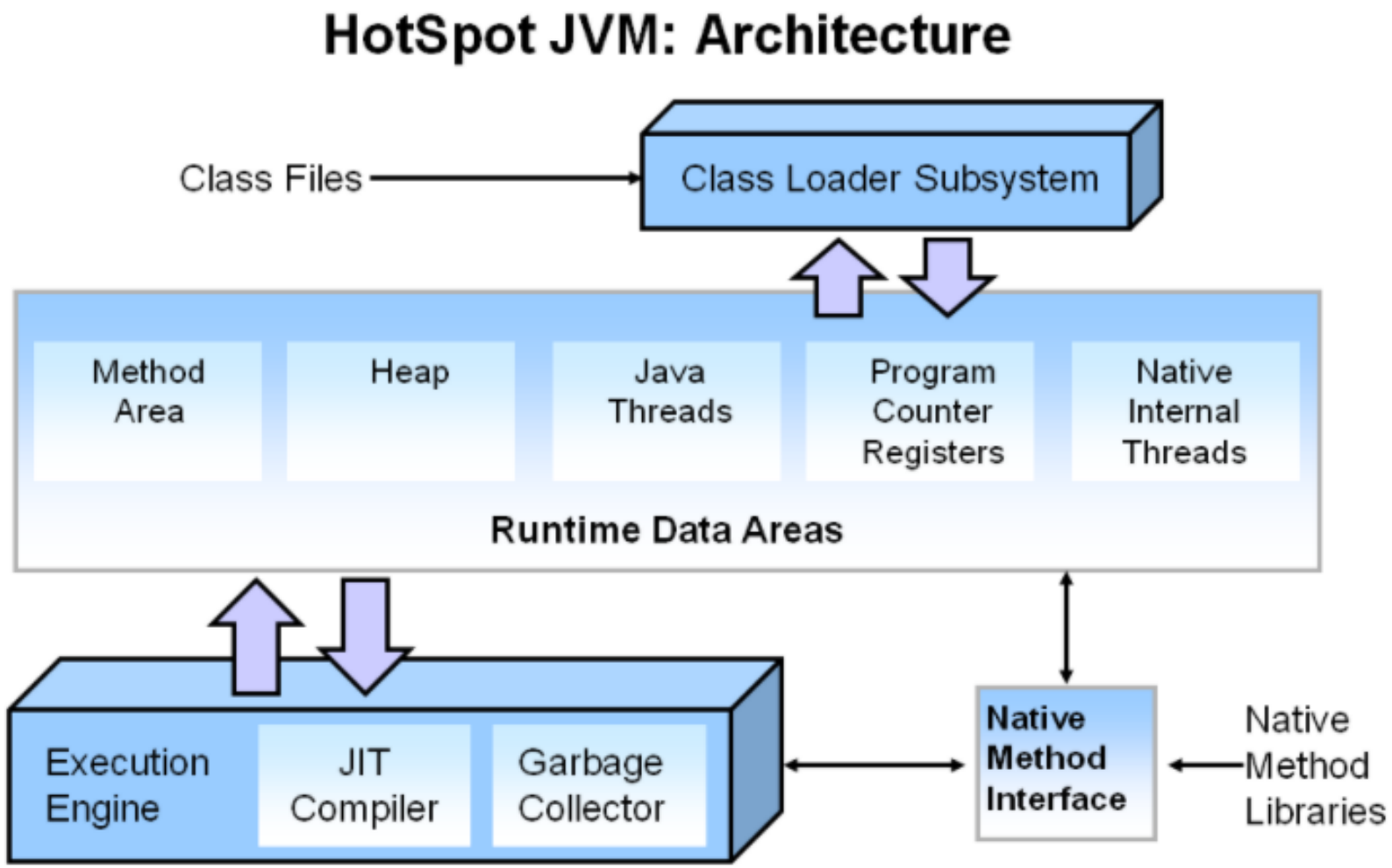
In addition, more threads mean more work for the system scheduler to decide which thread gets access to resources next.

Thread Pool

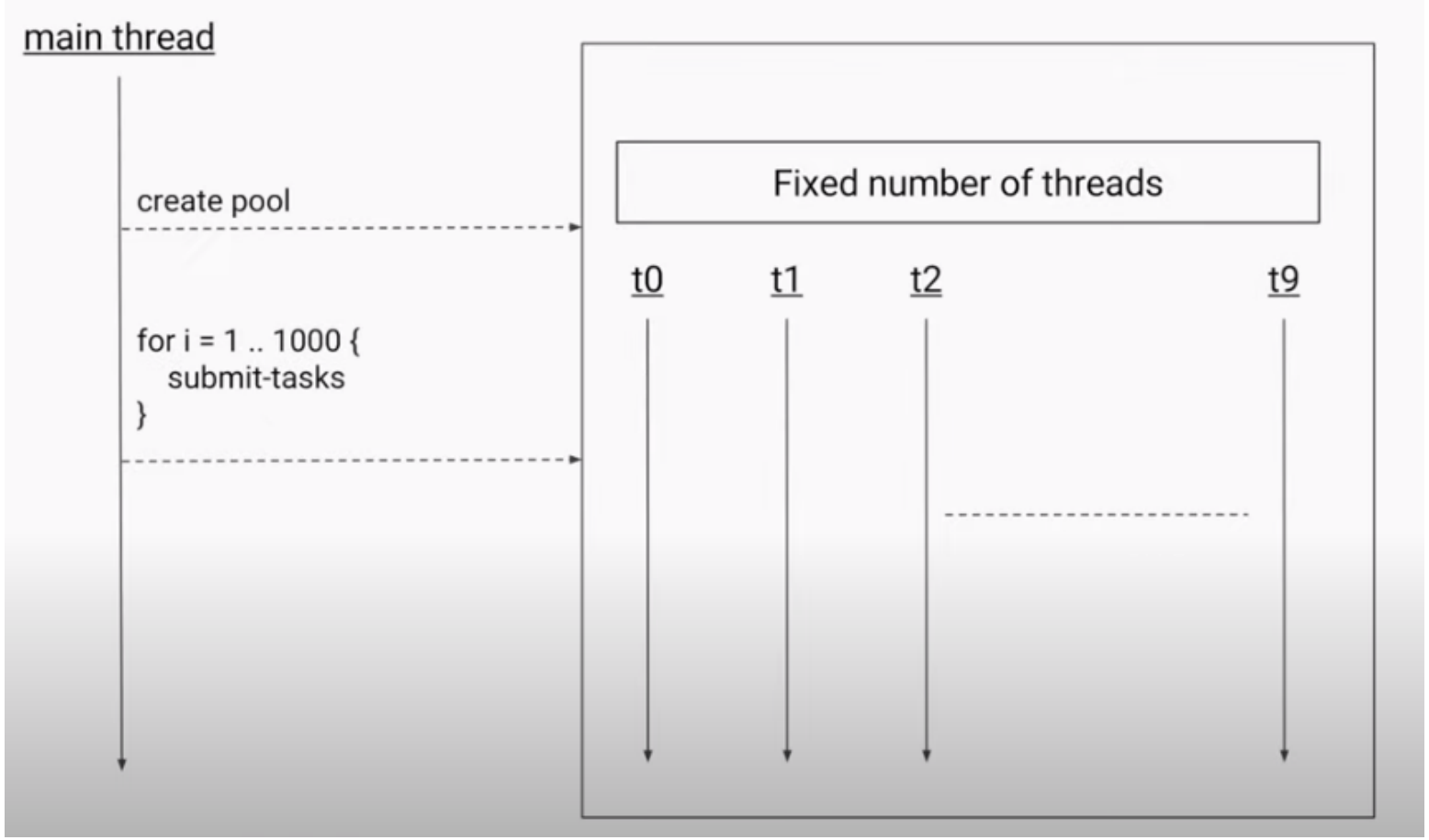
A thread pool helps mitigate the issue of performance by reducing the number of threads needed and managing their lifecycle.

Essentially, threads are kept in the thread pool until they're needed, after which they execute the task and return the pool to be reused later. This mechanism is especially helpful in systems that execute a large number of small tasks.

Thread Pool


HotSpot JVM: Architecture
Class Files
Class Loader Subsystem
Method Area
Heap
Java Threads
Program Counter Registers
Native Internal Threads
Runtime Data Areas
Execution Engine
JIT Compiler
Garbage Collector
Native Method Interface
Native Method Libraries

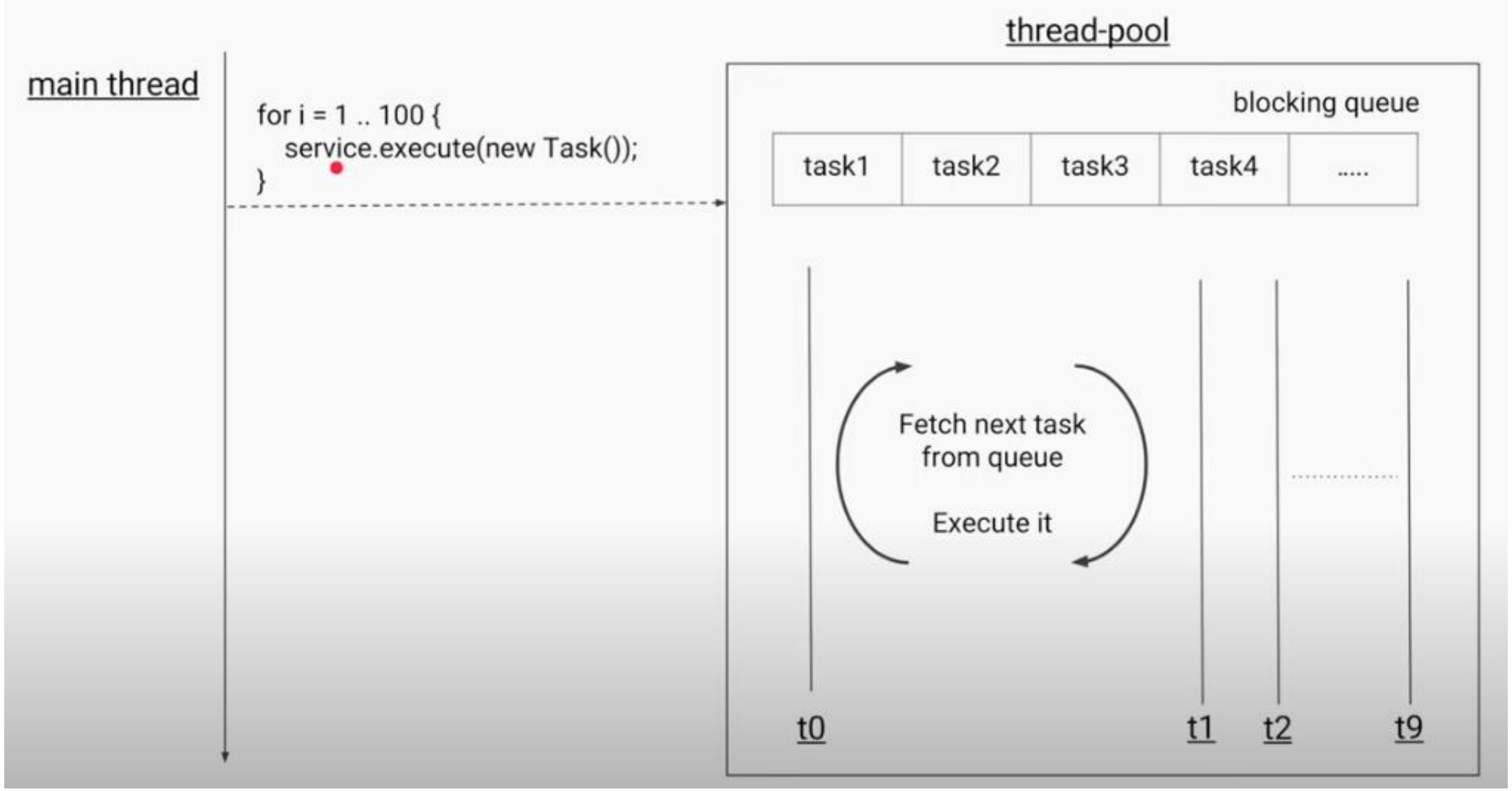
Thread Pool


main thread
create pool
for i = 1 .. 1000 {
submit-tasks
}
Fixed number of threads
t0
t1
t2
t9

Thread Pool

```
public static void main(String[] args) {

    // create the pool
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new Task());
    }
    System.out.println("Thread Name: " + Thread.currentThread().getName());
}

static class Task implements Runnable {
    public void run() {
        System.out.println("Thread Name: " + Thread.currentThread().getName());
    }
}
```

Thread Pool

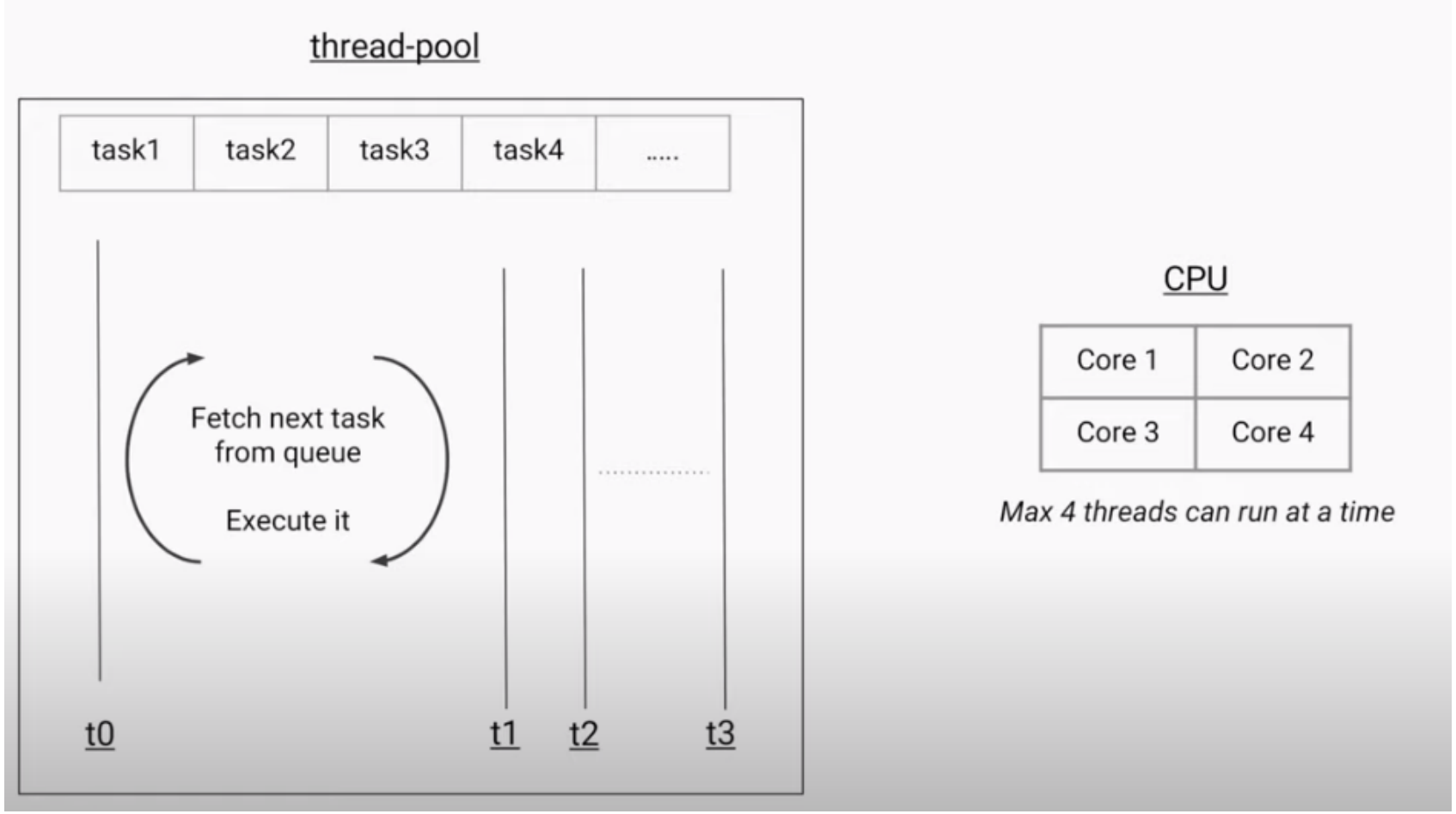

thread-pool
main thread
for i = 1 .. 100 {
service.execute(new Task());
}
blocking queue
task1
task2
task3
task4
.....
Fetch next task
from queue
Execute it
t0
t1
t2
t9

Thread Pool

Generally, a Java thread pool is composed of:

- the pool of worker threads, responsible for managing the threads
- a thread factory that is responsible for creating new threads
- a queue of tasks waiting to be executed

Idea pool size

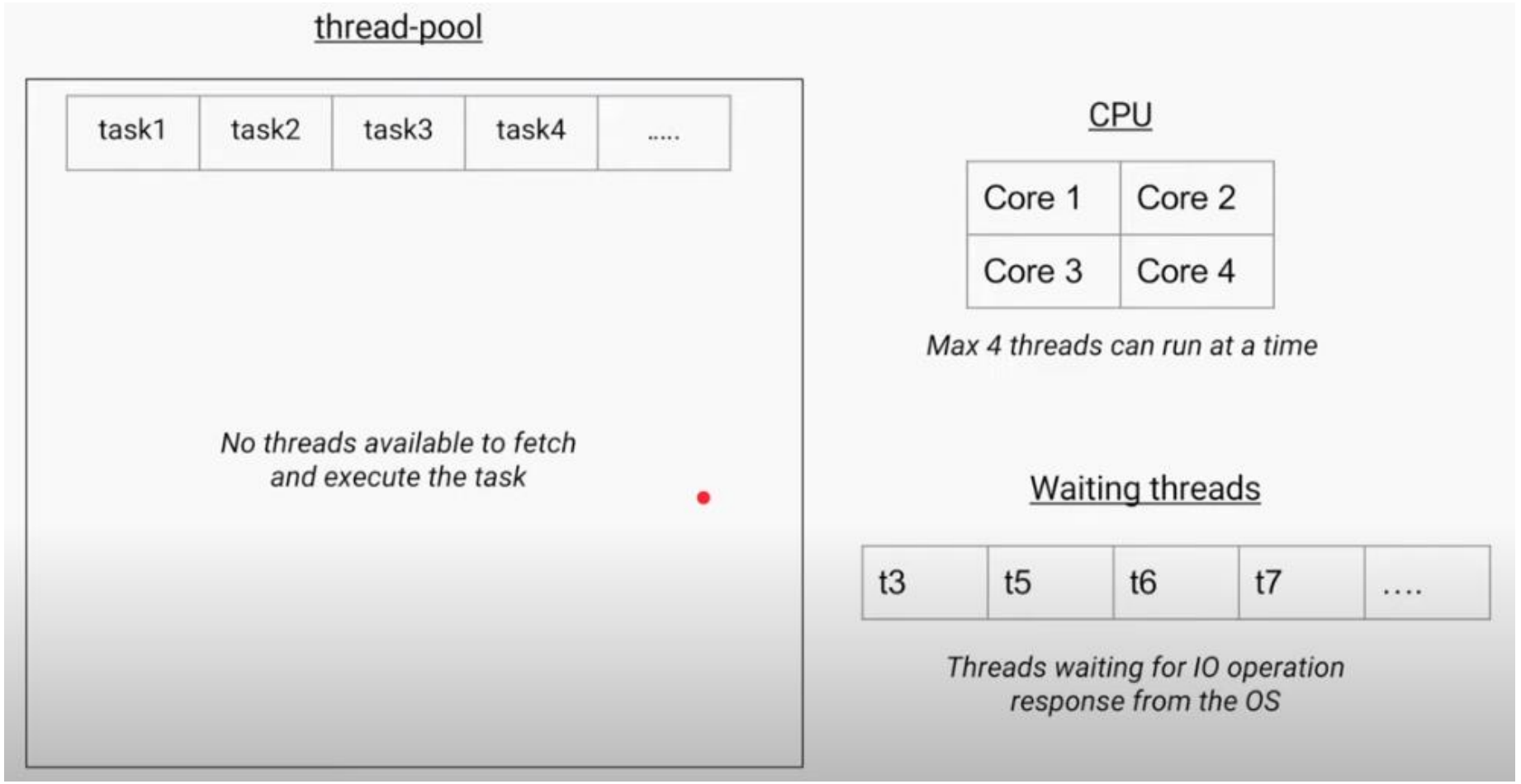


Idea pool size

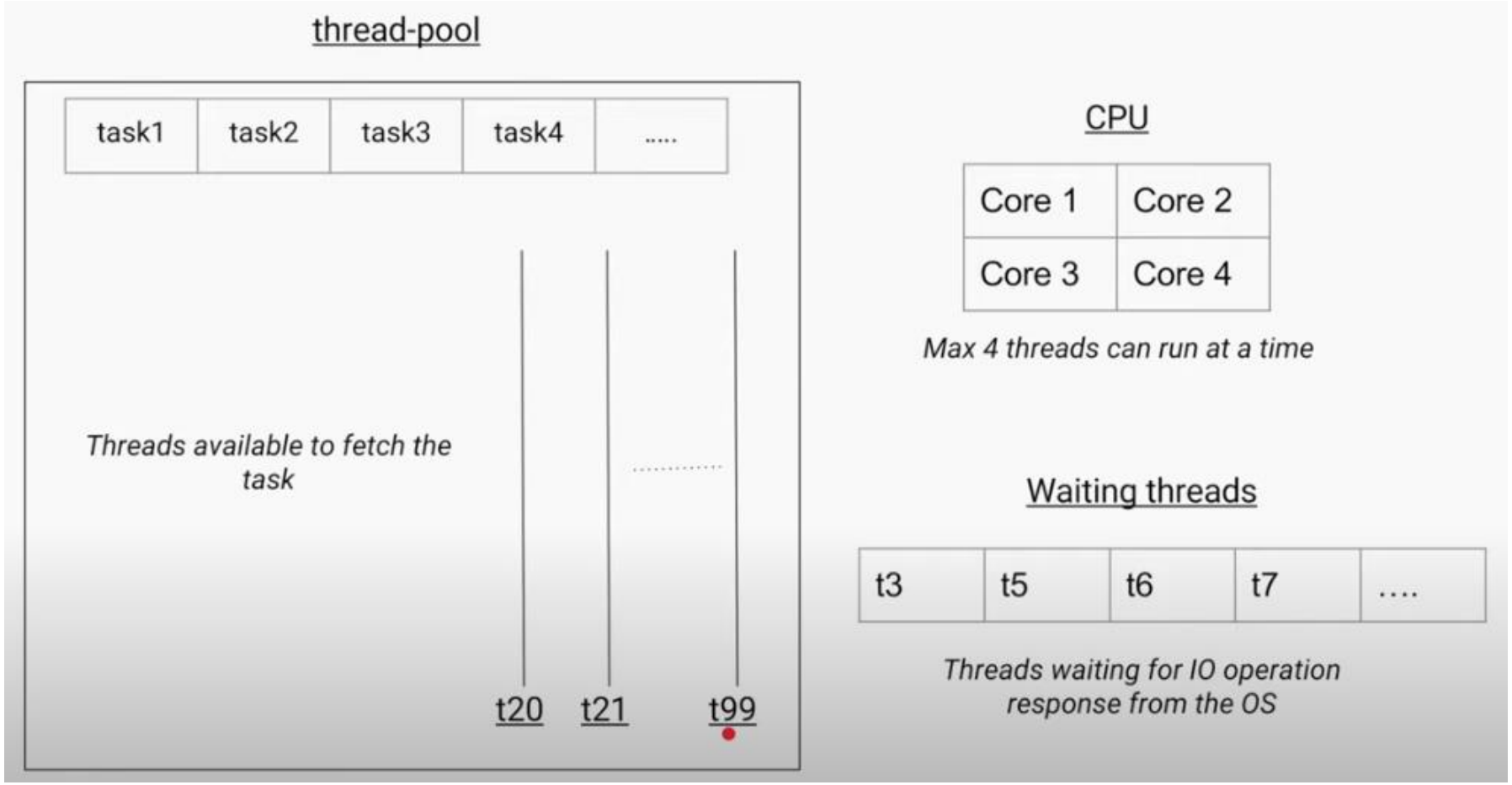
```
public static void main(String[] args) {

    // get count of available cores
    int coreCount = Runtime.getRuntime().availableProcessors();
    ExecutorService service = Executors.newFixedThreadPool(coreCount);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new CpuIntensiveTask());
    }
}
```

```
static class CpuIntensiveTask implements Runnable {
    public void run() {
        // some CPU intensive operations
    }
}
```

IO operation



IO operation

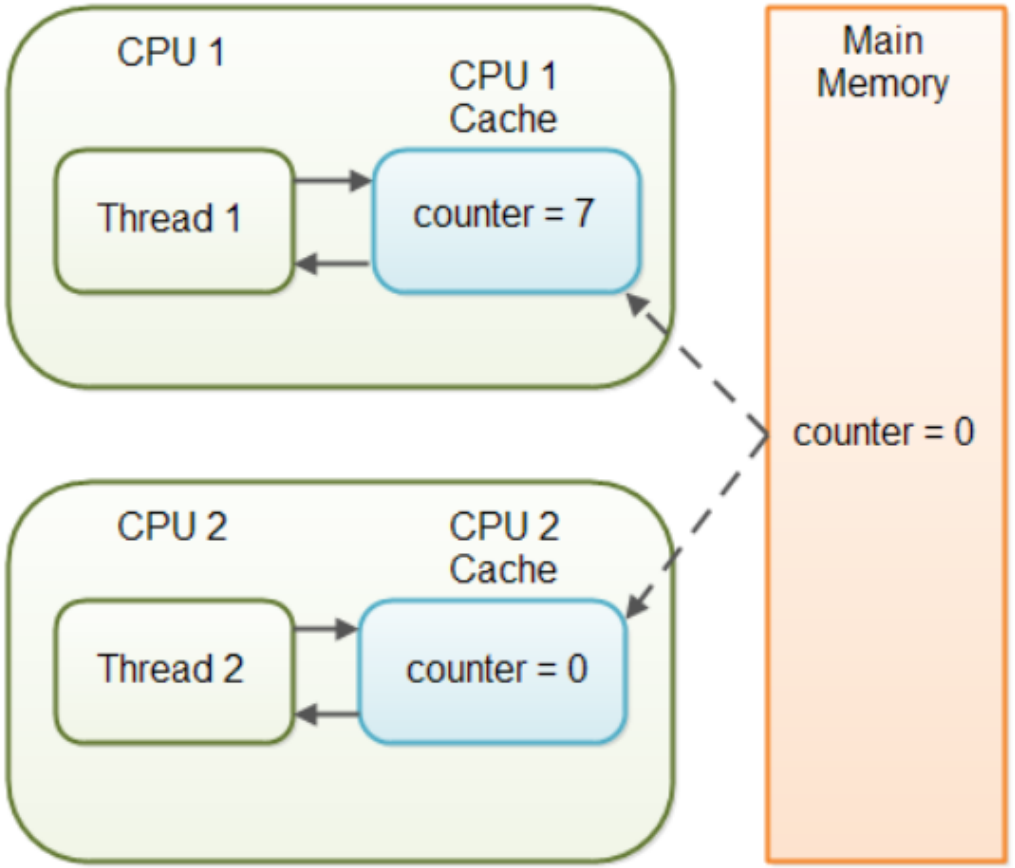


IO operation

```
public static void main(String[] args) {

    // much higher count for IO tasks
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 100);

    // submit the tasks for execution
    for (int i = 0; i < 100; i++) {
        service.execute(new IOTask());
    }
}

static class IOTask implements Runnable {
    public void run() {
        // some IO operations which will cause thread to block/wait
    }
}
```

Volatile

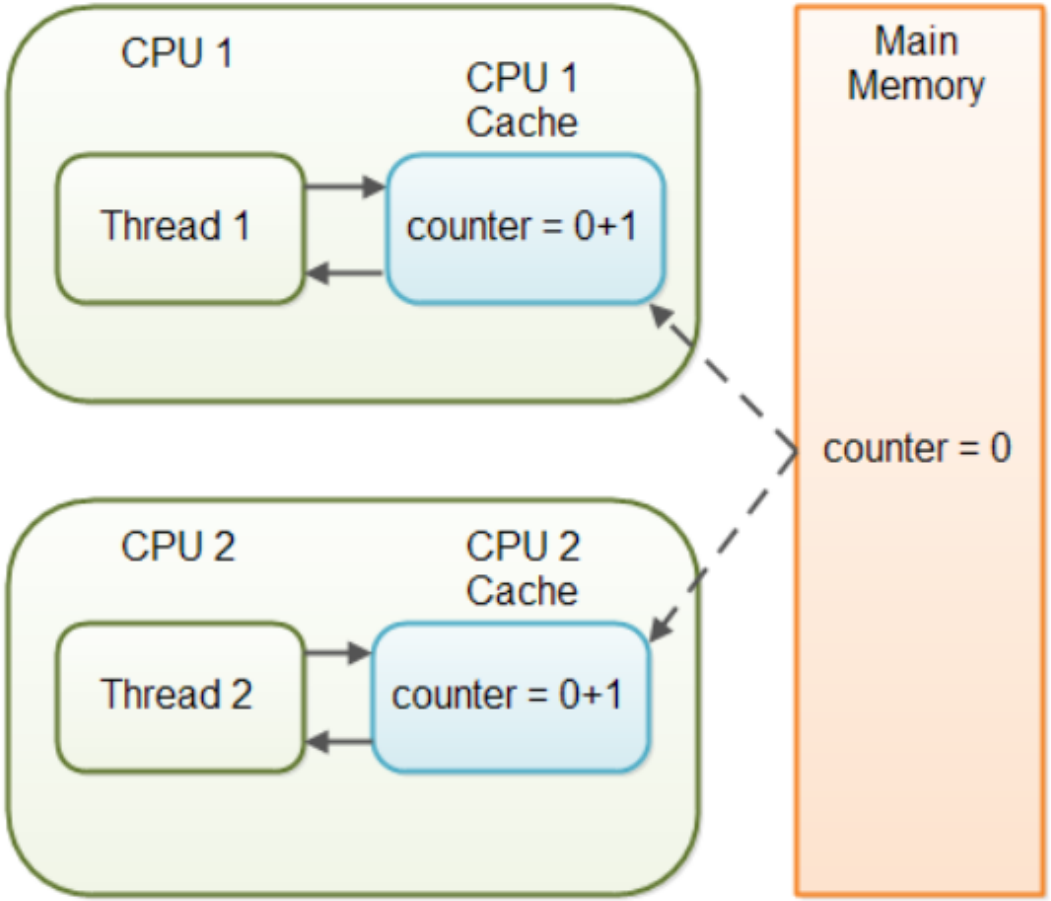
In a multithreaded application where the threads operate on non-volatile variables, each thread may copy variables from main memory into a CPU cache while working on them, for performance reasons. If your computer contains more than one CPU, each thread may run on a different CPU. That means, that each thread may copy the variables into the CPU cache of different CPUs



String Builder
String Buffer

Volatile

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.



Synchronized

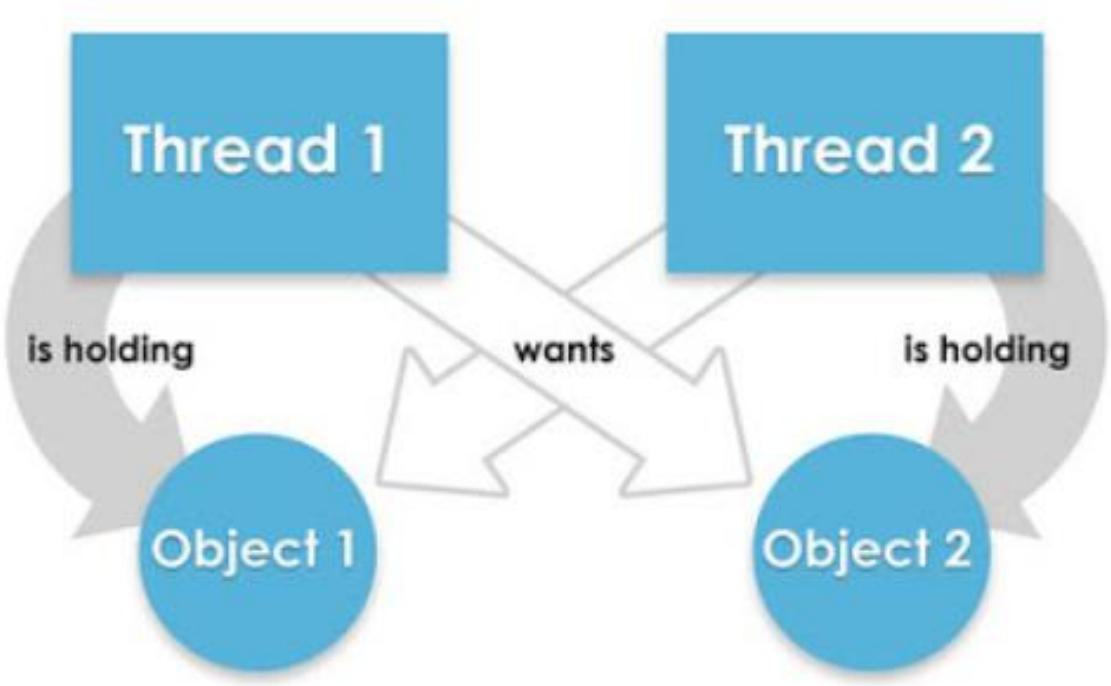
Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. 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 a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

```
synchronized(sync_object)
{
   // Access shared variables and other
   // shared resources
}
```

Deadlock


Thread 1
Thread 2
is holding
wants
is holding
Object 1
Object 2

Deadlock

Suspend() method is **deadlock prone**. If the target thread holds a lock on object when it is suspended, no thread can lock this object until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, it results in **deadlock formation**.
These **deadlocks** are generally called **Frozen processes**.

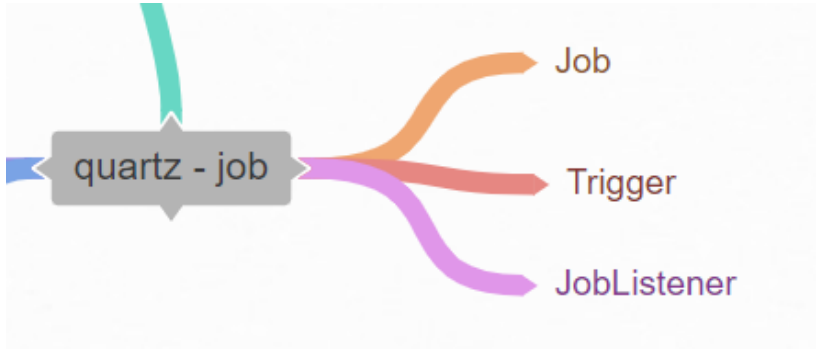
Deadlock

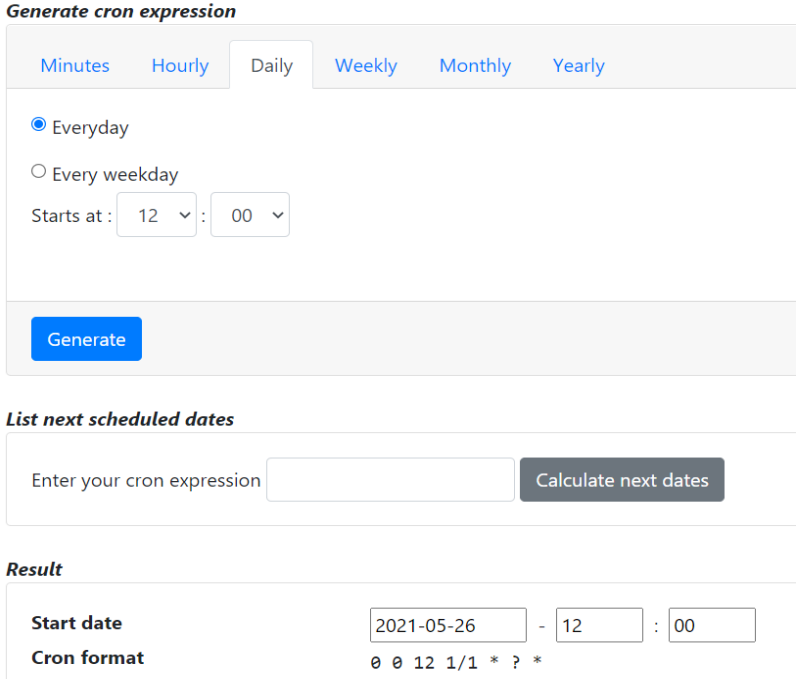
Suspend() method puts thread from running to waiting state. And thread can go **from waiting to runnable** state **only when resume() method is called** on thread. It is deprecated method.

Resume() method is **only used with suspend()** method that's why it's also deprecated method.

Quartz – Cron Job

- https://mvnrepository.com/artifact/org.quartz-scheduler/quartz
- https://mvnrepository.com/artifact/org.quartz-scheduler/quartz-jobs



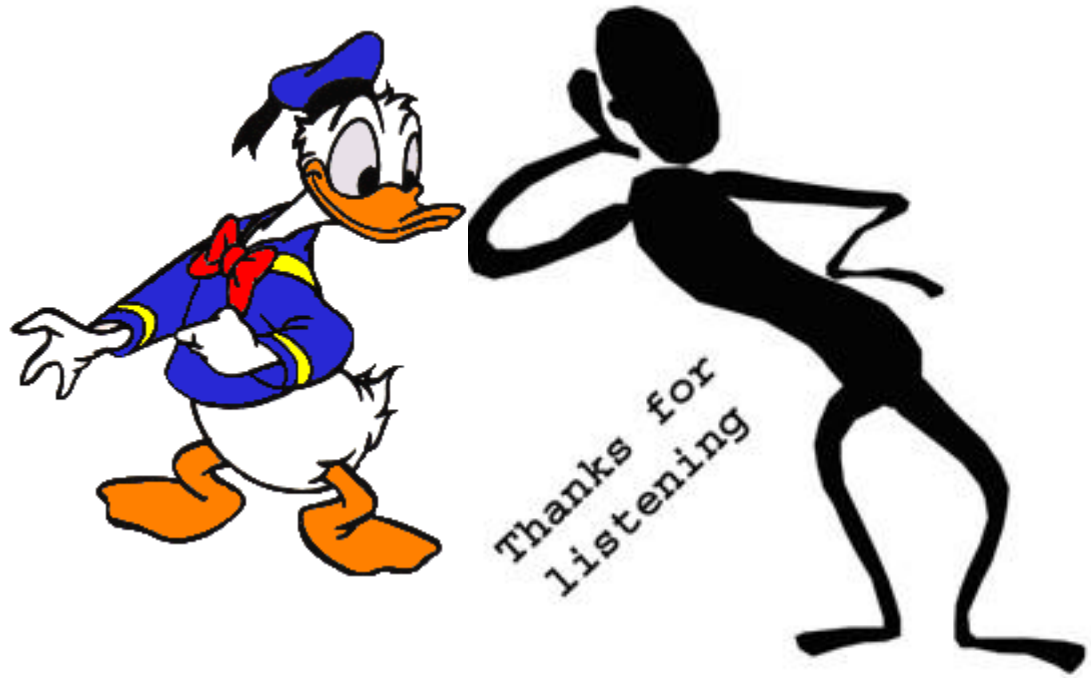



```java
final String group = "group";

// Job
final JobKey jobKey = new JobKey(HelloJob.class.getSimpleName(), group);
final JobDetail jobDetail = JobBuilder.newJob(HelloJob.class).withIdentity(jobKey).build();

// Trigger
final Trigger trigger = TriggerBuilder.newTrigger()
        .withIdentity(HelloJob.class.getSimpleName(), group)
        .withSchedule(CronScheduleBuilder.cronSchedule(expression))
        .build();

// Scheduler
final Scheduler scheduler = new StdSchedulerFactory().getScheduler();

// JobListener
scheduler.getListenerManager().addJobListener(new HelloJobListener());

scheduler.start();
scheduler.scheduleJob(jobDetail, trigger);
```

Quartz – Cron Job

- https://mvnrepository.com/artifact/org.quartz-scheduler/quartz
- https://mvnrepository.com/artifact/org.quartz-scheduler/quartz-jobs

Expression	**Meaning**
`0 0 12 * * ?`	Fire at 12pm (noon) every day
`0 15 10 ? * *`	Fire at 10:15am every day
`0 15 10 * * ?`	Fire at 10:15am every day
`0 15 10 * * ? *`	Fire at 10:15am every day
`0 15 10 * * ? 2005`	Fire at 10:15am every day during the year 2005
`0 * 14 * * ?`	Fire every minute starting at 2pm and ending at 2:59pm, every day
`0 0/5 14 * * ?`	Fire every 5 minutes starting at 2pm and ending at 2:55pm, every day
`0 0/5 14,18 * * ?`	Fire every 5 minutes starting at 2pm and ending at 2:55pm, AND fire every 5 minutes starting at 6pm and ending at 6:55pm, every day
`0 0-5 14 * * ?`	Fire every minute starting at 2pm and ending at 2:05pm, every day
`0 10,44 14 ? 3 WED`	Fire at 2:10pm and at 2:44pm every Wednesday in the month of March.
`0 15 10 ? * MON-FRI`	Fire at 10:15am every Monday, Tuesday, Wednesday, Thursday and Friday
`0 15 10 15 * ?`	Fire at 10:15am on the 15th day of every month
`0 15 10 L * ?`	Fire at 10:15am on the last day of every month
`0 15 10 L-2 * ?`	Fire at 10:15am on the 2nd-to-last last day of every month
`0 15 10 ? * 6L`	Fire at 10:15am on the last Friday of every month
`0 15 10 ? * 6L`	Fire at 10:15am on the last Friday of every month
`0 15 10 ? * 6L 2002-2005`	Fire at 10:15am on every last friday of every month during the years 2002, 2003, 2004 and 2005
`0 15 10 ? * 6#3`	Fire at 10:15am on the third Friday of every month
`0 0 12 1/5 * ?`	Fire at 12pm (noon) every 5 days every month, starting on the first day of the month.
`0 11 11 11 11 ?`	Fire every November 11th at 11:11am.



END