Process and Thread Management

Mai Duc An[1], Tran Thuy Chau[2], Do Hong Hai[3], Pham Quang Huy[4], and Nguyen Tieu Phuong[5]

[1] ICT 01 - K66, Ha Noi University of Science and Technology An.MD210008@sis.hust.edu.vn
[2] ICT 01 - K66, Ha Noi University of Science and Technology Chau.TT215182@sis.hust.edu.vn
[3] ICT 01 - K66, Ha Noi University of Science and Technology Hai.DH21599@sis.hust.edu.vn
[4] ICT 01 - K66, Ha Noi University of Science and Technology Huy.PQ215207@sis.hust.edu.vn
[5] ICT 01 - K66, Ha Noi University of Science and Technology Phuong.NT210692@sis.hust.edu.vn

1 Process and Thread

1.1 Definitions

Thread is a basic CPU using unit. A thread consists of ThreadID, Pgrogram Counter, Registers and Stack space. Inside a process, there can be many threads, and each individual thread can share the same code segment, data segment (for storing global objects and variables) as well as other resources allocated by the operating system, such as opening files, etc. Aside from that, each thread has its own stack space, registers set and program counter.

A thread is sometimes called a "lightweight process", and each process must have at least one thread.

1.2 Advantages of Threads

- **Responsiveness:** If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- **Effective utilization of multiprocessor system:** If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.
- **Resource sharing:** Resources like code, data, and files can be shared among all threads within a process.
- **Communication:** Communication between multiple threads is easier, as the threads share a common address space. while in the process we have to follow some specific communication techniques for communication between the two processes.
- **Enhanced throughput of the system:** If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.
- **Economy:** Create, switch, terminate threads is less expensive in comparison with processes.

1.3 Multi-threading

In single core system: The ability to provide concurrent execution paths for different sequences of instructions, which are managed as independent threads within a single process.

Benefits of multi-threading model:

- Multithreading can improve the performance and efficiency of a program by utilizing the available CPU resources more effectively. Executing multiple threads concurrently, it can take advantage of parallelism and reduce overall execution time.
- Multithreading can enhance responsiveness in applications that involve user interaction. By separating time-consuming tasks from the main thread, the user interface can remain responsive and not freeze or become unresponsive.
- Multithreading can enable better resource utilization. For example, in a server application, multiple threads can handle incoming client requests simultaneously, allowing the server to serve more clients concurrently.
- Multithreading can facilitate better code organization and modularity by dividing complex tasks into smaller, manageable units of execution. Each thread can handle a specific part of the task, making the code easier to understand and maintain.

Despite all the benefits of multi-threading, creating a lot of threads and execute all of them simultaneously can exert a huge load on the system, which may lead to poor overall execution, system deadlocks and so on. In order to overcome this setback, the concept of "Thread Pooling" was born, a strategy used to manage the execution of threads in a more controlled and efficient manner.

2 Thread Pool

2.1 Definition

A Thread Pool is a programming concept and often a part of concurrent programming libraries in various programming languages. It refers to a pool of worker threads that are pre-created and kept ready to perform tasks. The idea is to reuse these threads rather than creating a new thread for each task, which can be more efficient in terms of resource management.

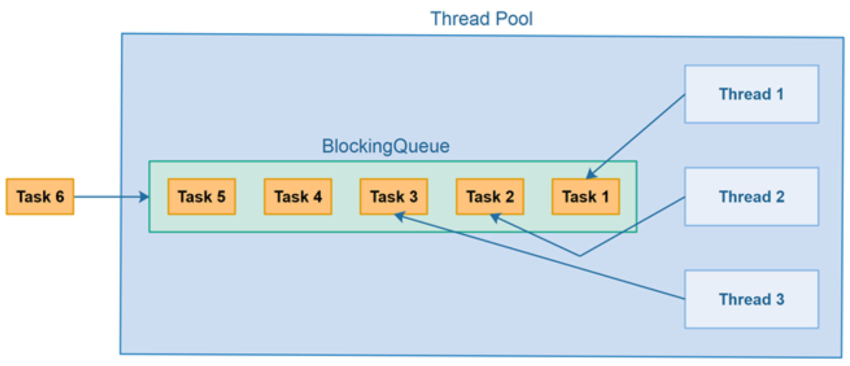

Fig. 1: How a Thread Pool works

2.2 Example

When we write a program to download files from the Internet, each file requires one thread to perform the download process. Assuming we need to download 10,000 audio files, we would need 10,000 threads operating simultaneously in the same program. This can easily lead to program overload, impacting memory and performance, and increasing the likelihood of a program crash due to difficulty in control.

Not every time creating multiple threads simultaneously will result in high performance, as each time a new thread is created and allocated memory, there can be memory and performance issues, potentially leading to program crashes.

The Thread Pool is introduced to limit the number of threads running inside our application at the same time.

3 Implementaion of thread pooling in Java

3.1 Runnable

- **Runnable** is an interface in Java that represents a task to be executed concurrently by a thread.
- It declares a single method, **run()**, which contains the code to be executed by the thread.
- To use **Runnable**, you typically create a class that implements the **Runnable** interface and overrides the **run()** method.

```
public class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed concurrently
    }
}
```

3.2 Executor

An **Executor** is an object responsible for managing threads and executing **Runnable** tasks. It abstracts the details of creating threads, scheduling, etc., allowing us to focus on developing the logic of tasks without worrying about thread management details. The Java Concurrency API defines three fundamental interfaces for Executors:

1. **Executor**: This is the parent interface of all Executors. It defines only one method, execute(Runnable).
2. **ExecutorService**: This is an Executor that allows monitoring the progress of tasks returning values (Callable) through Future objects, which represents the result of an asynchronous computation or task that is being executed by a Runnable or Callable, and managing the termination of threads. Its main methods include submit() and shutdown().
3. **ScheduledExecutorService**: This is an ExecutorService that can schedule tasks for execution after a certain period or at regular intervals. Its main methods are schedule(), scheduleAtFixedRate(), and scheduleWithFixedDelay().

Executors can be created using one of the methods provided by the utility class Executors, such as:

- **newSingleThreadExecutor()**: In this ThreadPool, there is only one thread, and tasks will be processed sequentially.

- **newCachedThreadPool()**: In this ThreadPool, there are multiple threads, and tasks will be processed concurrently. Unused threads will be reused for new tasks. By default, if a thread is not used within 60 seconds, it will be terminated.
- **newFixedThreadPool(int n)**: In this ThreadPool, a fixed number of threads are maintained. If a new task is submitted when all threads are busy, the task will be placed in a Blocking Queue. When a thread becomes available, the task will be taken from the queue and processed.
- **newScheduledThreadPool(int corePoolSize)**: Similar to newCachedThreadPool(), but with a delay between threads.
- **newSingleThreadScheduledExecutor()**: Similar to newSingleThreadExecutor(), but with a delay between threads.

```
    Executor executor = ...; // Implementation of Executor
    executor.execute(new MyRunnable());
```

3.3 InterruptedException

InterruptedException is an exception class in Java that is thrown when a thread is interrupted while it is in a blocked, waiting, or sleeping state. Interruption is a way for one thread to request the interruption (termination or stoppage) of another thread.

3.4 Code demo

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExample {

    // Constants for the number of tasks and the size of the thread pool
    private static final int TASK_COUNT = 20;
    private static final int THREAD_POOL_SIZE = 4;

    public static void main(String[] args) {
        // Conventional approach
        long startConventional = System.currentTimeMillis();
        runConventionalTasks();
        long endConventional = System.currentTimeMillis();
        System.out.println("Conventional approach took: " + (endConventional -
            startConventional) + " ms");

        // Thread pool approach
        long startThreadPool = System.currentTimeMillis();
        runThreadPoolTasks();
        long endThreadPool = System.currentTimeMillis();
        System.out.println("Thread pool approach took: " + (endThreadPool - startThreadPool)
            + " ms");
    }

    // Conventional approach: Run tasks sequentially
    private static void runConventionalTasks() {
        for (int i = 0; i < TASK_COUNT; i++) {
            performTaskWithDelay(i);
        }
    }

    // Thread pool approach: Use ExecutorService for parallel execution
    private static void runThreadPoolTasks() {
        // Create a fixed-size thread pool
        ExecutorService executorService = Executors.newFixedThreadPool(THREAD_POOL_SIZE);

        // Submit tasks to the thread pool
        for (int i = 0; i < TASK_COUNT; i++) {
            final int taskNumber = i;
            executorService.submit(() -> performTaskWithDelay(taskNumber));
        }

        // Shut down the thread pool and wait for termination
        executorService.shutdown();
        try {
            // Waits for all threads in the ExecutorService to terminate
            // 'Long.MAX_VALUE' is used as the timeout, indicating an effectively infinite
                timeout
            // 'TimeUnit.NANOSECONDS' specifies the time unit for the timeout
            executorService.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
        } catch (InterruptedException e) {
            // The purpose is to handle the 'InterruptedException' by catching it and then
                immediately restoring the interrupted status of the current thread
            Thread.currentThread().interrupt();
        }
    }

    // Simulate a task with a delay
    private static void performTaskWithDelay(int taskNumber) {
        // Print information about the current thread before the delay
        System.out.println("Task " + taskNumber + " started by thread: " +
            Thread.currentThread().getName());

        // Simulate a network call or I/O operation with a 100ms delay
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // The purpose is to handle the 'InterruptedException' by catching it and then
                immediately restoring the interrupted status of the current thread
            Thread.currentThread().interrupt();
        }

        // Print information about the current thread after the delay
        System.out.println("Task " + taskNumber + " completed by thread: " +
            Thread.currentThread().getName());
    }
}
```