Thread Concepts and Thread Pools

Introduction to Threads

Threads are fundamental units of CPU utilization, allowing programs to perform multiple operations concurrently within a single process. They represent the smallest sequence of programmed instructions that can be managed independently by a scheduler [1].

1. Creating a New *Thread*

In Java, we can create a *Thread* in following ways:

- By extending *Thread* class
- By implementing *Runnable* interface
- Using Lambda expressions

1.1. By Extending *Thread* Class

To create a new thread, extend the class with *Thread* and override the `run()` method.

```
class SubTask extends Thread {
  public void run() {
    System.out.println("SubTask started...");
  }

}
```

1.2. By Implementing *Runnable* Interface

Implementing the *Runnable* interface is considered a better approach because, in this way, the thread class can extend any other class. Remember, in Java, a class can extend only one class but implement multiple interfaces.

```
class SubTaskWithRunnable implements Runnable {
  public void run() {
    System.out.println("SubTaskWithRunnable started...");
  }

}
```

1.3. Using Lambda Expressions

Lambda expressions help create inline instances of functional interfaces and can help reduce the boilerplate code.

```
Runnable subTaskWithLambda = () ->
{
  System.out.println("SubTaskWithLambda started...");

};
```

Key Characteristics of Threads:

- Share the same memory space within a process
- Have their own stack and register set
- Can be created and destroyed independently of other threads
- Enable parallel execution on multi-core processors

Threads are particularly useful for improving application performance, responsiveness, and resource utilization. They're extensively used in modern operating systems and programming languages to handle concurrent operations efficiently [2].

2. Thread Life Cycle States

Throughout their existence, threads transition through various states, forming what's known as the thread lifecycle:

2.1. New:

Thread is created but not yet started. *Thread* remains in *New* state until the program starts the thread using its *start()* method. At this point, the thread is not alive.

2.2 Runnable:

Thread is ready to run and waiting for CPU time

2.3 Running:

Thread is currently executing

2.4. Blocked/Waiting:

Thread is temporarily inactive (e.g., waiting for I/O or synchronization).A RUNNABLE thread can transition to the TIMED WAITING state if it provides an optional wait interval when it’s waiting

for another thread to perform a task. You can put a java thread in TIMED WAITING state by calling using following methods:

- *thread.sleep(long millis)*
- *wait(int timeout)* or *wait(int timeout, int nanos)*
- *thread.join(long* millis*)*

Such a thread returns to the RUNNABLE state when it is notified by another thread or when the timed interval expires—whichever comes first. Timed waiting threads and waiting threads cannot use a processor, even if one is available.

2.5. Terminated: A thread enters the `TERMINATED` state (sometimes called the dead state) when it successfully completes its task or otherwise terminated due to any error or even it was forcefully killed. Understanding these states is crucial for effective thread management and debugging concurrent applications [3].

3. Thread Pools

Thread pools are a design pattern for managing and reusing a collection of worker threads efficiently. They address the overhead associated with thread creation and destruction, especially in scenarios where tasks are short-lived and numerous.

Benefits of Thread Pools:

- Improved performance by reducing thread creation overhead
- Better resource management and control over system load
- Enhanced stability by limiting the number of concurrent threads
- Simplified task submission and execution

Common Thread Pool Types:

1. Fixed Thread Pool: Maintains a constant number of threads
2. Cached Thread Pool: Creates new threads as needed, reuses idle threads
3. Scheduled Thread Pool: Allows tasks to be executed after a delay or periodically
4. Work-Stealing Pool: Employs work-stealing algorithm for load balancing (Java's ForkJoinPool)

Thread pools are widely used in server applications, task scheduling systems, and parallel computing frameworks to manage concurrent workloads efficiently [6].

4. Starting a New Thread

We can start a new thread in Java in multiple ways, let us learn about them.

4.1. Using *Thread.start()*

Thread's `start()` method is considered the heart of multithreading. Without executing this method, we cannot start a new *Thread*. The other methods also internally use this method to start a thread, except *virtual threads*.

The *start()* method is responsible for registering the thread with the platform thread scheduler and doing all the other mandatory activities such as resource allocations.

Let us learn with an example how to create and start a *Thread* using *start()* method.

```
class SubTask extends Thread {
  public void run() {
    System.out.println("SubTask started...");
  }
}
```

```
Thread subTask = new SubTask();

subTask.start();
```

We can use the *start()* method to execute threads created using *Runnable* interface.

```
class SubTaskWithRunnable implements Runnable {
  public void run() {
    System.out.println("SubTaskWithRunnable started...");
  }
}

Thread subTaskWithRunnable = new Thread(new SubTaskWithRunnable());

subTaskWithRunnable.start();
```

Using the lambda expression technique is no different from *Runnable* interface method.

```
Runnable subTaskWithLambda = () -> {
  System.out.println("SubTaskWithLambda started...");
};

Thread subTask = new Thread(subTaskWithLambda);

subTask.start();
```

Eventually, *Thread* class `start()` method is the only way to start a new Thread in Java.

The other ways (except virtaul threads) internally uses *start()* method.

4.2. Using *ExecutorService*

Creating a new *Thread* is resource intensive. So, creating a new Thread, for every subtask, decreases the performance of the application. To overcome these problems, we should use *thread pools*.

A *thread pool* is a pool of already created *Threads* ready to do our task. In Java, *ExecutorService* is the backbone of creating and managing thread pools. We can submit either a *Runnable* or a *Callable* task in the *ExecutorService,* and it executes the submitted task using the one of the threads in the pool.

```
Runnable subTaskWithLambda = () -> {
  System.out.println("SubTaskWithLambda started...");
};

ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(subTaskWithLambda);
```

If we create the thread using lambda expression then the whole syntax becomes so easy.

```
ExecutorService executorService = Executors.newFixedThreadPool(10);

executorService.execute(() ->
{
  System.out.println("SubTaskWithLambda started...");

});
```

4.3. Delayed Execution with *ScheduledExecutorService*

When we submit a task to the executor, it executes as soon as possible. But suppose we want to execute a task after a certain amount of time or to run the task periodically then we can use *ScheduledExecutorService.*

The following example is scheduling a task to be executed after 1o seconds delay, and it will return the result "*completed*" when it has completed its execution.

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(2);


ScheduledFuture<String> result = executor.schedule(() -> {
 System.out.println("Thread is executing the job");
 return "completed";
}, 10, TimeUnit.SECONDS);



System.out.println(result.get());
```

4.4. Using *CompletableFuture*

CompletableFuture is an extension to *Future* API introduced in *Java 8*. It implements *Future* and *CompletionStage* interfaces. It provides methods for creating, chaining and combining multiple *Futures*.

In the following example, *CompletableFuture* will start a new *Thread* and executes the provided task either by using `runAsync()` or `supplyAsync()` methods on that thread.

```
// Running RunnableJob using runAsync() method of CompletableFuture
CompletableFuture<Void> future = CompletableFuture.runAsync(new RunnableJob());
```

```
// Running a task using supplyAsync() method of CompletableFuture and return the
 result
CompletableFuture<String> result = CompletableFuture.supplyAsync(() -> "Thread is
 executing");

// Getting result from CompletableFuture

System.out.println(result.get());
```

4.5. Executing as a Virtual Thread

Since Java 19, virtual threads have been added as JVM-managed lightweight threads that will help in writing high throughput concurrent applications.

We can run a task as a virtual thread using the newly added APIs:

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

//or

Thread.startVirtualThread(() -> {
 //Code to execute in virtual thread
 System.out.println("Inside Runnable");
});

//or

Thread.Builder builder = Thread.ofVirtual().name("Virtual-Thread");

builder.start(runnable);
```

5. Best Practices and Considerations

When working with threads and thread pools, consider the following best practices:

- Use thread pools instead of creating threads manually for better performance and resource management
- Avoid excessive synchronization to prevent deadlocks and improve concurrency
- Utilize higher-level concurrency utilities (e.g., [java.util.concurrent package](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html)) when possible
- Ensure thread safety in shared data structures through proper synchronization or use of thread-safe collections
- Consider using immutable objects to simplify concurrent programming
- Be mindful of thread contention and potential bottlenecks in heavily multi-threaded applications
- Use profiling tools to identify and resolve concurrency issues
- Implement proper error handling and logging in multi-threaded environments

6. Conclusion

Understanding thread concepts and thread pools is essential for developing efficient, scalable, and responsive applications in modern computing environments. Proper use of threads can significantly enhance application performance and resource utilization, while thread pools provide a managed approach to concurrent execution. As concurrent programming continues to evolve, staying updated with the latest best practices and concurrency patterns is crucial for software developers working on multi-threaded applications.

References

[1] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts. Wiley.

[2] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java Concurrency in Practice. Addison-Wesley Professional.

[3] Oracle. (n.d.). Thread States. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.State.html

[4] Herlihy, M., & Shavit, N. (2012). The Art of Multiprocessor Programming. Morgan Kaufmann.

[5] Oracle. (n.d.). Guarded Blocks. Retrieved from https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html

[6] Oracle. (n.d.). Thread Pools. Retrieved from https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html