Threads and Thread Pool

Threads

A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads. But threads can be effective only if the CPU is more than 1 otherwise two threads have to context switch for that single CPU.

Why Do We Need Thread?

- Threads run in parallel improving the application performance. Each such thread has its own CPU state and stack, but they share the address space of the process and the environment.
- Threads can share common data so they do not need to use inter-process communication. Like the processes, threads also have states like ready, executing, blocked, etc.
- Priority can be assigned to the threads just like the process, and the highest priority thread is scheduled first.
- Each thread has its own Thread Control Block (TCB). Like the process, a context switch occurs for the thread, and register contents are saved in (TCB). As threads share the same address space and resources, synchronization is also required for the various activities of the thread.

Thread Pool

A thread pool is a software design pattern for achieving concurrency of execution in a computer program. Often also called a replicated workers or worker-crew model,[1] a thread pool maintains multiple threads waiting for tasks to be allocated for concurrent execution by the supervising program. By maintaining a pool of threads, the model increases performance and avoids latency in execution due to frequent creation and destruction of threads for short-lived tasks.[2] The number of available threads is tuned to the computing resources available to the program, such as a parallel task queue after completion of execution.

Performance

The size of a thread pool is the number of threads kept in reserve for executing tasks. It is usually a tunable parameter of the application, adjusted to optimize program performance.[3] Deciding the optimal thread pool size is crucial to optimize performance.

One benefit of a thread pool over creating a new thread for each task is that thread creation and destruction overhead is restricted to the initial creation of the pool, which may result in better performance and better system stability. Creating and destroying a thread and its associated resources can be an expensive process in terms of time. An excessive number of threads in reserve, however, wastes memory, and context-switching between the runnable threads invokes performance penalties. A socket connection to another network host, which might take many CPU cycles to drop and re-establish, can be maintained more efficiently by associating it with a thread that lives over the course of more than one network transaction.

Fixed Thread Pool

In Fixed thread pool, the number of threads is set upon creation and remains constant throughout its lifespan. This configuration is beneficial when seeking predictability. The worker threads are always ready for use and the maximum number of concurrent tasks remains the same.

Cached Thread Pool

The Cached Thread Pool dynamically adjusts its size in response to the workload, prioritizing the reuse of idle threads before resorting to creating new ones. This configuration is useful when we want to adapt allocated resources to accommodate varying task loads. The cached pool doesn’t have an upper limit on the maximum number of threads it can spawn. Consequently, during a substantial influx of tasks, a large number of threads will be created, potentially causing undesired behavior.

Some noteworthy use cases for a Cached Thread Poolinclude:

- **Asynchronous processing**, particularly when dealing with a varying number of tasks.
- Handling **short-lived tasks** that sporadically spawn.
- When the requirement is for **each task to be executed concurrently**.

Single Thread Pool

Fixed Thread Pool with a single thread. Its primary use cases include:

- Executing **thread-unsafe** code not designed for concurrent environments
- Handling **background tasks** that lack time sensitivity
- Enforcing **sequential execution**

Scheduled Thread Pool

Scheduled Thread Pools offer the capability to execute tasks with a delay or at a fixed rate. Similar to the Fixed Thread Pool, it takes the number of threads as a parameter, ensuring that the pool maintains a consistent number of active threads at all times.

Custom Thread Pools

All types of thread pools can be implemented using the ThreadPoolExecutor class. It gives us control over every aspect of the pool, including:

- **Core Pool Size**: The number of threads that are kept alive at all times.
- **Max Pool Size**: The maximum number of threads the pool can hold.
- **Keep Alive Time**: The time after which idle threads will be terminated.
- **Queue Implementation**: By providing our own task queue, we can limit the maximum number of pending tasks or make it infinite. The most commonly used queue is LinkedBlockingQueue, which can be either infinite or have a specified max size. If the max size is reached, new tasks will be rejected.

ThreadPoolExecutor woking when taks a few

The six threads (core pool size) are kept alive at all times, regardless of the number of tasks in the queue. If the queue size never exceeds 6 the pool will never spawn extra threads

ThreadPoolExecutor woking under heavy load

In case of a task influx, the pool will start utilizing the extra space for threads. On top of the 6 threads it already has, an extra 10 are spawned. The total number of threads will never exceed the max pool size (16). The number of new threads spawned is determined by the queue size; in this case, we have more than 16 tasks in the queue, and the thread pool will ramp up to its maximum capacity. For example, if the queue size was 12, only 6 new threads would be spawned.

After tasks have been processed and the queue is again empty, the pool will scale down. It will wait for the specified **keep alive time**, before removing the surplus threads. The pool is back in its initial state.

Code-wise, the ThreadPoolExecutor is quite straightforward to initialize. After that, it can be used like any other thread pool discussed so far.

References

https://web.archive.org/web/20080207124322/http://today.java.net/pub/a/today/2008/01/31/query-by-slice-parallel-execute-join-thread-pool-pattern.html

https://usf-cs272-spring2022.github.io/files/Thread%20Pools%20and%20Work%20Queues.pdf

https://medium.com/@b.stoilov/everything-you-need-to-know-about-thread-pools-in-java-fe02e803d339

https://en.wikipedia.org/wiki/Thread_pool