developerWorks®

Java theory and practice: Thread pools and work queues

Thread pools help achieve optimum resource utilization

Brian Goetz

July 01, 2002

One of the most common questions posted on our Multithreaded Java programming discussion forum is some version of "How do I create a thread pool?" In nearly every server application, the question of thread pools and work queues comes up. In this article, Brian Goetz explores the motivations for thread pools, some basic implementation and tuning techniques, and some common hazards to avoid.

View more content in this series

Why thread pools?

Many server applications, such as Web servers, database servers, file servers, or mail servers, are oriented around processing a large number of short tasks that arrive from some remote source. A request arrives at the server in some manner, which might be through a network protocol (such as HTTP, FTP, or POP), through a JMS queue, or perhaps by polling a database. Regardless of how the request arrives, it is often the case in server applications that the processing of each individual task is short-lived and the number of requests is large.

One simplistic model for building a server application would be to create a new thread each time a request arrives and service the request in the new thread. This approach actually works fine for prototyping, but has significant disadvantages that would become apparent if you tried to deploy a server application that worked this way. One of the disadvantages of the thread-per-request approach is that the overhead of creating a new thread for each request is significant; a server that created a new thread for each request would spend more time and consume more system resources creating and destroying threads than it would processing actual user requests.

In addition to the overhead of creating and destroying threads, active threads consume system resources. Creating too many threads in one JVM can cause the system to run out of memory or thrash due to excessive memory consumption. To prevent resource thrashing, server applications need some means of limiting how many requests are being processed at any given time.

A thread pool offers a solution to both the problem of thread life-cycle overhead and the problem of resource thrashing. By reusing threads for multiple tasks, the thread-creation overhead is spread over many tasks. As a bonus, because the thread already exists when a request arrives, the delay introduced by thread creation is eliminated. Thus, the request can be serviced immediately, rendering the application more responsive. Furthermore, by properly tuning the number of threads in the thread pool, you can prevent resource thrashing by forcing any requests in excess of a certain threshold to wait until a thread is available to process it.

Alternatives to thread pools

Thread pools are far from the only way to use multiple threads within a server application. As mentioned above, sometimes it is perfectly sensible to spawn a new thread for each new task. However, if the frequency of task creation is high and the mean task duration is low, spawning a new thread for each task will lead to performance problems.

Another common threading model is to have a single background thread and task queue for tasks of a certain type. AWT and Swing use this model, in which there is a GUI event thread, and all work that causes changes in the user interface must execute in that thread. However, because there is only one AWT thread, it is undesirable to execute tasks in the AWT thread that may take a perceptible amount of time to complete. As a result, Swing applications often require additional worker threads for long-running UI-related tasks.

Both the thread-per-task and the single-background-thread approaches can work perfectly well in certain situations. The thread-per-task approach works quite well with a small number of long-running tasks. The single-background-thread approach works quite well so long as scheduling predictability is not important, as is the case with low-priority background tasks. However, most server applications are oriented around processing large numbers of short-lived tasks or subtasks, and it is desirable to have a mechanism for efficiently processing these tasks with low overhead, as well as some measure of resource management and timing predictability. Thread pools offer these advantages.

Work queues

In terms of how thread pools are actually implemented, the term "thread pool" is somewhat misleading, in that the "obvious" implementation of a thread pool doesn't exactly yield the result we want in most cases. The term "thread pool" predates the Java platform, and is probably an artifact from a less object-oriented approach. Still, the term continues to be widely used.

While we could easily implement a thread pool class in which a client class would wait for an available thread, pass the task to that thread for execution, and then return the thread to the pool when it is finished, this approach has several potentially undesirable effects. What happens, for instance, when the pool is empty? Any caller that attempted to pass a task to a pool thread would find the pool empty, and its thread would block while it waited for an available pool thread. Often, one of the reasons we would want to use background threads is to prevent the submitting thread from blocking. Pushing the blocking all the way to the caller, as is the case in the "obvious" implementation of a thread pool, can end up causing the same problem we were trying to solve.

What we usually want is a work queue combined with a fixed group of worker threads, which uses `wait()` and `notify()` to signal waiting threads that new work has arrived. The work queue is generally implemented as some sort of linked list with an associated monitor object. Listing 1 shows an example of a simple pooled work queue. This pattern, using a queue of `Runnable` objects, is a common convention for schedulers and work queues, although there is no particular need imposed by the Thread API to use the `Runnable` interface.

```
public class WorkQueue
{
    private final int nThreads;
    private final PoolWorker[] threads;
    private final LinkedList queue;

    public WorkQueue(int nThreads)
    {
        this.nThreads = nThreads;
        queue = new LinkedList();
        threads = new PoolWorker[nThreads];

        for (int i=0; i<nThreads; i++) {
            threads[i] = new PoolWorker();
            threads[i].start();
        }
    }

    public void execute(Runnable r) {
        synchronized(queue) {
            queue.addLast(r);
            queue.notify();
        }
    }

    private class PoolWorker extends Thread {
        public void run() {
            Runnable r;

            while (true) {
                synchronized(queue) {
                    while (queue.isEmpty()) {
                        try
                        {
                            queue.wait();
                        }
                        catch (InterruptedException ignored)
                        {
                        }
                    }

                    r = (Runnable) queue.removeFirst();
                }

                // If we don't catch RuntimeException,
                // the pool could leak threads
                try {
                    r.run();
                }
                catch (RuntimeException e) {
                    // You might want to log something here
                }
            }
        }
    }
}
```

You may have noticed that the implementation in Listing 1 uses `notify()` instead of `notifyAll()`. Most experts advise the use of `notifyAll()` instead of `notify()`, and with good reason: there are subtle risks associated with using `notify()`, and it is only appropriate to use it under certain specific conditions. On the other hand, when used properly, `notify()` has more desirable performance characteristics than `notifyAll()`; in particular, `notify()` causes many fewer context switches, which is important in a server application.

The example work queue in Listing 1 meets the requirements for safely using `notify()`. So go ahead and use it in your program, but exercise great care when using `notify()` in other situations.

Risks of using thread pools

While the thread pool is a powerful mechanism for structuring multithreaded applications, it is not without risk. Applications built with thread pools are subject to all the same concurrency risks as any other multithreaded application, such as synchronization errors and deadlock, and a few other risks specific to thread pools as well, such as pool-related deadlock, resource thrashing, and thread leakage.

Deadlock

With any multithreaded application, there is a risk of deadlock. A set of processes or threads is said to be *deadlocked* when each is waiting for an event that only another process in the set can cause. The simplest case of deadlock is where thread A holds an exclusive lock on object X and is waiting for a lock on object Y, while thread B holds an exclusive lock on object Y and is waiting for the lock on object X. Unless there is some way to break out of waiting for the lock (which Java locking doesn't support), the deadlocked threads will wait forever.

While deadlock is a risk in any multithreaded program, thread pools introduce another opportunity for deadlock, where all pool threads are executing tasks that are blocked waiting for the results of another task on the queue, but the other task cannot run because there is no unoccupied thread available. This can happen when thread pools are used to implement simulations involving many interacting objects, and the simulated objects can send queries to one another that then execute as queued tasks, and the querying object waits synchronously for the response.

Resource thrashing

One benefit of thread pools is that they generally perform well relative to the alternative scheduling mechanisms, some of which we've already discussed. But this is only true if the thread pool size is tuned properly. Threads consume numerous resources, including memory and other system resources. Besides the memory required for the `Thread` object, each thread requires two execution call stacks, which can be large. In addition, the JVM will likely create a native thread for each Java thread, which will consume additional system resources. Finally, while the scheduling overhead of switching between threads is small, with many threads context switching can become a significant drag on your program's performance.

If a thread pool is too large, the resources consumed by those threads could have a significant impact on system performance. Time will be wasted switching between threads, and having more threads than you need may cause resource starvation problems, because the pool threads are consuming resources that could be more effectively used by other tasks. In addition to the resources used by the threads themselves, the work done servicing requests may require additional resources, such as JDBC connections, sockets, or files. These are limited resources as well, and having too many concurrent requests may cause failures, such as failure to allocate a JDBC connection.

Concurrency errors

Thread pools and other queuing mechanisms rely on the use of `wait()` and `notify()` methods, which can be tricky. If coded incorrectly, it is possible for notifications to be lost, resulting in threads remaining in an idle state even though there is work in the queue to be processed. Great care must be taken when using these facilities; even experts make mistakes with them. Better yet, use an existing implementation that is known to work, such as the `util.concurrent` package discussed below in No need to write your own.

Thread leakage

A significant risk in all kinds of thread pools is thread leakage, which occurs when a thread is removed from the pool to perform a task, but is not returned to the pool when the task completes. One way this happens is when the task throws a `RuntimeException` or an `Error`. If the pool class does not catch these, then the thread will simply exit and the size of the thread pool will be permanently reduced by one. When this happens enough times, the thread pool will eventually be empty, and the system will stall because no threads are available to process tasks.

Tasks that permanently stall, such as those that potentially wait forever for resources that are not guaranteed to become available or for input from users who may have gone home, can also cause the equivalent of thread leakage. If a thread is permanently consumed with such a task, it has effectively been removed from the pool. Such tasks should either be given their own thread or wait only for a limited time.

Request overload

It is possible for a server to simply be overwhelmed with requests. In this case, we may not want to queue every incoming request to our work queue, because the tasks queued for execution may consume too many system resources and cause resource starvation. It is up to you to decide what to do in this case; in some situations, you may be able to simply throw the request away, relying on higher-level protocols to retry the request later, or you may want to refuse the request with a response indicating that the server is temporarily busy.

Guidelines for effective use of thread pools

Thread pools can be an extremely effective way to structure a server application, as long as you follow a few simple guidelines:

- Don't queue tasks that wait synchronously for results from other tasks. This can cause a deadlock of the form described above, where all the threads are occupied with tasks that are in turn waiting for results from queued tasks that can't execute because all the threads are busy.
- Be careful when using pooled threads for potentially long-lived operations. If the program must wait for a resource, such as an I/O completion, specify a maximum wait time, and then fail or requeue the task for execution at a later time. This guarantees that eventually *some* progress will be made by freeing the thread for a task that might complete successfully.
- Understand your tasks. To tune the thread pool size effectively, you need to understand the tasks that are being queued and what they are doing. Are they CPU-bound? Are they I/O-bound? Your answers will affect how you tune your application. If you have different classes of tasks with radically different characteristics, it may make sense to have multiple work queues for different types of tasks, so each pool can be tuned accordingly.

Tuning the pool size

Tuning the size of a thread pool is largely a matter of avoiding two mistakes: having too few threads or too many threads. Fortunately, for most applications the middle ground between too few and too many is fairly wide.

Recall that there are two primary advantages to using threading in applications: allowing processing to continue while waiting for slow operations such as I/O, and exploiting the availability of multiple processors. In a compute-bound application running on an N-processor machine, adding additional threads may improve throughput as the number of threads approaches N, but adding additional threads beyond N will do no good. Indeed, too many threads will even degrade performance because of the additional context switching overhead.

The optimum size of a thread pool depends on the number of processors available and the nature of the tasks on the work queue. On an N-processor system for a work queue that will hold entirely compute-bound tasks, you will generally achieve maximum CPU utilization with a thread pool of N or N+1 threads.

For tasks that may wait for I/O to complete -- for example, a task that reads an HTTP request from a socket -- you will want to increase the pool size beyond the number of available processors, because not all threads will be working at all times. Using profiling, you can estimate the ratio of waiting time (WT) to service time (ST) for a typical request. If we call this ratio WT/ST, for an N-processor system, you'll want to have approximately N*(1+WT/ST) threads to keep the processors fully utilized.

Processor utilization is not the only consideration in tuning the thread pool size. As the thread pool grows, you may encounter the limitations of the scheduler, available memory, or other system resources, such the number of sockets, open file handles, or database connections.

No need to write your own

Doug Lea has written an excellent open-source library of concurrency utilities, `util.concurrent`, which includes mutexes, semaphores, collection classes like queues and hash tables that perform well under concurrent access, and several work queue implementations. The `PooledExecutor` class from this package is an efficient, widely used, correct implementation of a thread pool based on a work queue. Rather than try and write your own, which is easy to get wrong, you might consider using some of the utilities in `util.concurrent`.

The `util.concurrent` library also serves as the inspiration for JSR 166, a Java Community Process (JCP) working group that will be producing a set of concurrency utilities for inclusion in the Java class library under the `java.util.concurrent` package, and which should be ready for the Java Development Kit 1.5 release.

Conclusion

The thread pool is a useful tool for organizing server applications. It is quite straightforward in concept, but there are several issues to watch for when implementing and using one, such as deadlock, resource thrashing, and the complexities of `wait()` and `notify()`. If you find yourself in need of a thread pool for your application, consider using one of the `Executor` classes from `util.concurrent`, such as `PooledExecutor`, rather than writing one from scratch. If you find yourself creating threads to handle short-lived tasks, you should definitely consider using a thread pool instead.

Related topics

- See IBM Bluemix in action
- Concurrent Programming in Java
- Overview of package util.concurrent Release 1.3.4
- JSR 166: Concurrency Utilities
- Writing multithreaded Java applications