Thread Management in Java: Monitoring and Adjusting Priorities of Threads Using ThreadMXBean

Nathan Miller, Jae Woo Lee, Henning Schulzrinne
Columbia University

ABSTRACT

Java provides ThreadMXBean to monitor threads. The purpose of this project was to harness this ability and create a rudimentary algorithm to demonstrate fair thread scheduling using this interface, in environments where malicious threads can change their own priorities in order to overcome normal thread scheduling.

1. MOTIVATION

A virtual machine can be created in which resources of the host computer are unknown and inaccessible to any processes in the VM; this is in fact common practice. In doing so, the processes running on the VM are incapable of monopolizing resources of other VMs or external processes running on the machine unless the VM is given full measure of the resources. However, in a scenario where multiple processes occupy the same VM, measures must be undertaken to prevent these processes from depriving each other of resources.

In working on the NetServ project, my research led me to the question of whether Java could natively monitor and control threads, readjusting their priorities when the threads use too much or too little cpu time (indicative of a deprivation scenario). My goal was to use ThreadMXBean to monitor these threads and a very rudimentary algorithm to ensure that no thread can monopolize cpu time. Since a NetServ container consists of multiple threads, this would be useful if possible in NetServ.

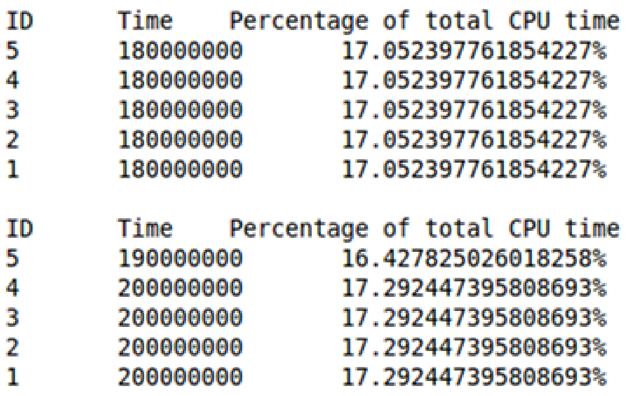
I also explored the usage of Java's SecurityManager to prevent threads from becoming abusive, and examined Java's thread hierarchy to treat child threads as part of their parent. However, neither of these two additional goals were successful for reasons that shall be made clear.

2. DESIGNING A SOLUTION

ThreadMXBean is a Java interface that handles management of threads inside a JVM. It handles, based on Thread ID, monitoring the cpu usage of each thread as well as other useful thread monitoring duties.

To design a solution in which ThreadMXBean could be used to detect resource choking, I created two programs and attempted to create two more to address all the issues mentioned above. Each of these programs was uncomplicated, and the final products are by no means scalable, but do provide a good understanding of what can and cannot be done using Java.

3. PART ONE: THREADMXBEAN

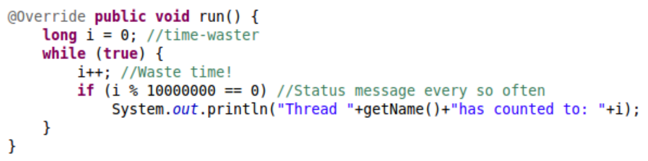


```
ID      Time      Percentage of total CPU time
5       180000000        17.052397761854227%
4       180000000        17.052397761854227%
3       180000000        17.052397761854227%
2       180000000        17.052397761854227%
1       180000000        17.052397761854227%

ID      Time      Percentage of total CPU time
5       190000000        16.427825026018258%
4       200000000        17.292447395808693%
3       200000000        17.292447395808693%
2       200000000        17.292447395808693%
1       200000000        17.292447395808693%
```

Figure 1: Example Output of Part One

The first program I created was very simple, with the goal of ensuring that ThreadMXBean would be able to provide the monitoring information I needed. Essentially, the program works as follows: the main thread spawns five friendly thread of class Worker, and assigns each of them MIN_PRIORITY, or Java Priority 1 [1]. Worker threads do nothing more than count in an infinite loop, occasionally reporting how high they've counted. See Figure 2.



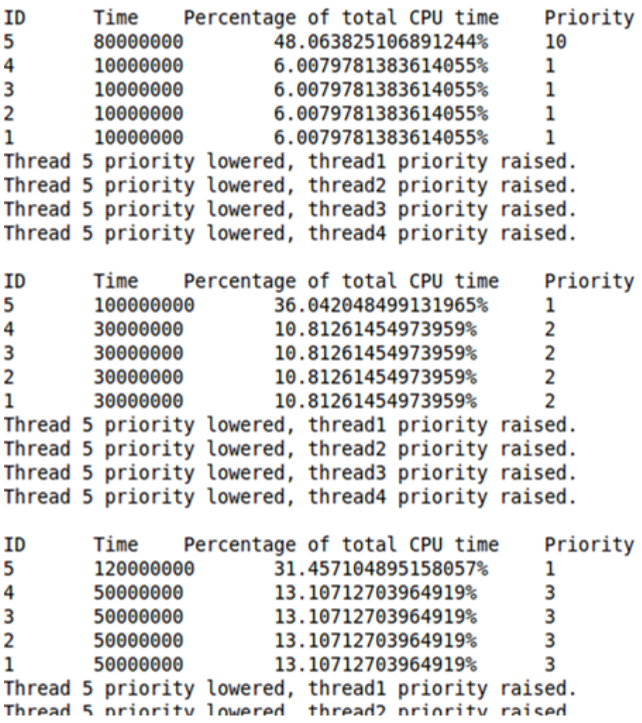
```
@Override public void run() {
    long i = 0; //time-waster
    while (true) {
        i++; //Waste time!
        if (i % 10000000 == 0) //Status message every so often
            System.out.println("Thread "+getName()+"has counted to: "+i);
    }
}
```

Figure 2: Worker.java Code Listing

The main thread reports, using a ThreadMXBean object, the cpu usage of all five Worker threads and the percentage of the total cpu usage time used. It then sleeps for another 100ms. Since all five threads act the same and have the same priority, their numbers stay about the same over the lifetime of the program. Figure 1 shows an example output of the first program.

At this point, it should be noted that ThreadMXBean keeps track of all running threads, and therefore at this point tracks nine threads rather than just the five Workers. The other four threads are: Signal Dispatcher, Finalizer, Reference Handler, and main.

4. PART TWO: DETECTING AND REPRIORITIZING ABUSIVE THREADS



```
ID      Time      Percentage of total CPU time     Priority
5       80000000         48.063825106891244%       10
4       10000000         6.0079781383614055%       1
3       10000000         6.0079781383614055%       1
2       10000000         6.0079781383614055%       1
1       10000000         6.0079781383614055%       1
Thread 5 priority lowered, thread1 priority raised.
Thread 5 priority lowered, thread2 priority raised.
Thread 5 priority lowered, thread3 priority raised.
Thread 5 priority lowered, thread4 priority raised.

ID      Time      Percentage of total CPU time     Priority
5       100000000        36.042048499131965%       1
4       30000000         10.81261454973959%        2
3       30000000         10.81261454973959%        2
2       30000000         10.81261454973959%        2
1       30000000         10.81261454973959%        2
Thread 5 priority lowered, thread1 priority raised.
Thread 5 priority lowered, thread2 priority raised.
Thread 5 priority lowered, thread3 priority raised.
Thread 5 priority lowered, thread4 priority raised.

ID      Time      Percentage of total CPU time     Priority
5       120000000        31.457104895158057%       1
4       50000000         13.10712703964919%        3
3       50000000         13.10712703964919%        3
2       50000000         13.10712703964919%        3
1       50000000         13.10712703964919%        3
Thread 5 priority lowered, thread1 priority raised.
Thread 5 priority lowered, thread2 priority raised.
```

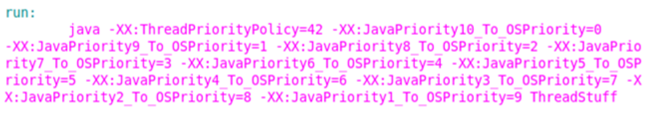
Figure 3: Example Output of Part Two

The second program deals with the first problem: a rogue thread that adjusts its own priority in order to monopolize cpu time. This thread, EvilWorker, starts out by adjusting its own priority to MAX_PRIORITY, or Java Priority 10 [1] and then acting as Worker. The main thread, when it polls threads, notes if threads have used too much cpu time in comparison to other threads, defined as a 150% difference. If so, it lowers the priorities of the offending threads or thread to MIN_PRIORITY and raises the priorities of the victimized threads or thread by one, so as to enable it to catch up in terms of cpu usage. This behavior repeats until usage is balanced.

In practice, this works out so that all threads eventually stabilize at Java priority 1 or 2. Figure 3 shows initial (pre-stabilization) output of this program. This works correctly *even* in the case that EvilWorker continues to reset its priority, although stabilizing takes much longer and all threads stabilize to MAX_PRIORITY rather than 1 or 2.

5. A NOTE ON LINUX, JVM, AND PRIORITIES

Interestingly, Linux ignores Java's priorities and assigns all threads the same priority, regardless of their priority inside the JVM. However, this can be disabled [2] as shown in Figure 4.



```
run:
        java -XX:ThreadPriorityPolicy=42 -XX:JavaPriority10_To_OSPriority=0
-XX:JavaPriority9_To_OSPriority=1 -XX:JavaPriority8_To_OSPriority=2 -XX:JavaPrio
rity7_To_OSPriority=3 -XX:JavaPriority6_To_OSPriority=4 -XX:JavaPriority5_To_OSP
riority=5 -XX:JavaPriority4_To_OSPriority=6 -XX:JavaPriority3_To_OSPriority=7 -X
X:JavaPriority2_To_OSPriority=8 -XX:JavaPriority1_To_OSPriority=9 ThreadStuff
```

Figure 4: Overriding Linux's ignoring of Java Priorities

This override not only forces Linux to recognize the Java priorities but also maps the Java priorities to Linux Nice priorities. Java priority 10 maps to Unix Nice level 0 and Java priority 1 maps to Unix Nice level 9.

Without this override, the issue to be solved actually does not exist on Linux, since all threads will have the same priority inside Linux. However, this is not the case for non-Linux VMs (Windows, for instance, does not ignore the Java priority system), nor will it solve the issue of a thread that spawns children to take more resources.

No matter what, no thread is completely deprived. Even in situations where its priority is lower, it simply receives a smaller cpu timeslice, rather than being cut off entirely. This has to do with Linux scheduling as well.

6. PART THREE: INTEGRATING SECURITYMANAGER

Of course, if a thread is incapable of setting its own priority, then EvilWorker will be incapable of its exploitation, and the monitoring main thread will have much less work to do (or be unnecessary altogether). To successfully do this, however, requires implementing a SecurityManager.

Unfortunately, there is no easy way of doing this. There are several problems that exist in attempting to do this. First, SecurityManager works with Permission objects, but Permissions are assigned per signing authority or per codebase. In my testing, I am unable to create separate signing authorities, nor codebases. Unfortunately, it is not feasible to assign Permissions on a per thread basis, so if the main thread has the permission to act in a certain way, the other threads will as well: this includes setting priorities.

Second, it is possible to create a subclass of SecurityManager in an attempt to create a customized checkPermission() method, however, overriding checkPermission seems to be a rather involved issue and in essence impossible; Java will not run if checkPermission is overridden. It is unclear if this a bug or by design (the latter would make sense) but in any case it is problematic.

It is, however, possible to override checkAccess, which is what is actually called when a thread attempts to setPriority. However, in practice, this is unreasonable: checkAccess calls checkPermission, and though it would be conceivable that an overridden checkAccess could itself throw a SecurityException based on the calling thread, this turns out to be very convoluted, as threads can spoof many of their "unique" properties, and others that cannot be spoofed are different from run to run. The result is the same as not overriding: all threads ostensibly have the same permissions.

There is a well-documented workaround for the problem above which allows one to override the deeper methods that checkPermission calls, in Java's AccessController. However, this is extremely tedious, and after a short attempt to do so, it became apparent that the time commitment would be unreasonable for a small-scale project.

7. PART FOUR: THREAD HIERARCHY

Another way that a thread could be malicious, even in an environment where priorities are controlled (either by the monitoring thread or by an OS policy) would be to spawn multiple threads. Since each thread will get the same time slice, the malicious thread and its children would still starve other threads.

In order to prevent this, it would be useful for the monitoring thread to group these children together and apply the same priority adjustment algorithm to the group as a whole, rather than the individual thread. However, this turns out to be difficult for two main reasons:

- First, there is an issue if the threads constantly readjust their priorities to MAX_PRIORITY. Although the algorithm will stabilize, each thread will gain an equal timeslice, not each grouping of threads.

- Second, and more importantly, Java does not treat the thread hierarchy in the manner that one would expect. A thread is identified by its threadgroup, not by its parent. Unless a parent voluntarily creates a new threadgroup (and malicious threads would obviously not voluntarily make things easier), then the threadgroup of a newly spawned child thread is the same as that of its parent. In other words, the thread hierarchy is quite flat.

Of course, rewriting the security architecture to solve the problems in part three could also be used to solve this problem: in theory, threads would not be allowed to spawn children, but rather a factory could be provided with the ability to create threads, and then group them according to the requesting "parent" thread.

8. FURTHER LIMITATIONS

Note that all these examples work only with a one-core VM. Using more than one core allows the VM to split the threads among cores according to various scheduling methods depending on the OS and JVM used. In these cases, a thread may starve only one thread rather than all of the other threads, or, if the thread scheduler is particularly good, it may not particularly starve any threads if, for instance, malicious threads are given their own cores.

In addition, ThreadMXBean reporting is not 100% accurate all times. There are times where adding up the cpu time used will yield a larger number than the actual time that has passed (the percentages will equal more than 100).

Also, because ThreadMXBean is a lightweight interface and deals with thread IDs and not the threads themselves, the programs I implemented are not easily scalable. The main thread is aware of all the threads in the system because it created them and kept track of them. Thus it is able to adjust their priorities; doing so through ThreadMXBean alone is impossible.

9. CONCLUSION

Using ThreadMXBean, it is possible to gain information about cpu usage of threads, rogue and benign. Coupling this with an algorithm to contain these rogue threads is successful, but not necessarily easily implementable in non-test environments. In addition, malicious threads have too much leeway to accomplish other malicious tasks that are not easily prevented in small-scale situations such as this.

Nonetheless, I do think ThreadMXBean provides a very useful tool, if not only to provide a monitor to check that an external scheduling or security mechanism is working as it should. In the future, I hope to further explore this topic, as well as expanding on the issues in parts three and four and attempting to solve them wherever possible.

10. REFERENCES

[1] Oracle. Java Constant Field Values. `http://download.oracle.com/javase/6/docs/api/constant-values.html#java.lang`.

[2] E. Stølsvik. Linux Java Thread Priorities workaround. `http://tech.stolsvik.com/2010/01/linux-java-thread-priorities-workaround.html`, 2010.