Performance Testing Threaded Library Routines

Linda Hamm
SAS Institute Inc.
Cary, NC
linda.hamm@sas.com
(919) 531-3582

Introduction and Background

I am a member of a white box testing team whose current assignment is to validate a set of threaded library routines that are portable across many platforms. These routines provide such services as:

- Memory management.
- I/O management.
- Thread management.

Many routines in the library are thin layers over operating system (OS) functions. Application developers use these routines to perform common operations such as:

- Create a thread.
- Open a file.
- Read/write data.
- Allocate memory.

Midway through functional testing of the previous release, we determined a need for performance testing on the suite of routines. We had no stated performance requirements, so we began by asking questions. What does performance testing mean for us? How do we determine scalability? What are we supposed to measure? What metrics are we expected to deliver?

A team of testing professionals with over 60 years of combined testing experience agreed to begin the process by researching performance testing. The team found information on load testing and tools, performance testing of whole systems, and testing to meet stated performance testing requirements. Because our performance testing needed to deal with library routines, these concepts and tools wouldn't work for our product area.

After research and discussions, the team determined that we would need to define and implement our own Performance Test strategy. This paper discusses what the team did to accomplish the goal as well as successes and failures we met along the way.

Implementation

Because the library routines were so close to the OS, we started by contacting OS vendor representatives to find out how their OS teams conducted performance testing. We learned about some great benchmarks of system performance and how the OS scaled. Vendors used the OS-specific set of tools for monitoring. Although this was great marketing material, it was difficult to determine the origin of the benchmarks, or how we could use the multiple sets of OS tools for our project. We also learned that most OS groups had huge testing teams dedicated to performance testing. We had a small team whose testing expertise was more with functional testing than performance testing. In addition, our group was expected to continue its primary job of functional testing.

So, with no performance requirements, no usage profiles, and no benchmarks, the team decided to benchmark certain routines and, in doing so, defined the following goals:

- To create a set of benchmarks that can be used at various points in the development cycle to make sure no performance degradations get introduced.
- To make sure the implementation for each host is efficient (i.e. if locks perform reasonable on one target host and are 3 times slower on another, is it because of the architecture or is it an implementation bug?).

We also decided that the secondary goals of performance test would be to try and answer the following questions:

- Does the implementation truly scale?
- Are there any parameters that could be tuned to make it faster for certain cases?
- When does performance degrade?

We started by developing a list of measurements we wanted to capture. The planned measurements were then analyzed and prioritized with development. These measurements included such things as:

- Time it takes to create and destroy a thread.
- Time it takes to read X bytes of data.
- Time it takes to allocate and free memory.

The team developed a test harness that provided the ability to run each test with a variety of parameters. In combination with the test harness, we used in-house tools for automation to capture and provide analysis of the following performance data for each routine being tested:

- Wall clock time – time from start to finish of the process based on actual time of day.
- Thread user time – time spent for a thread in developed code.
- Thread system time – time spent for a thread in OS calls.
- Total process user time – time spent for the process in developed code.
- Total process system time – time spent for the process in OS calls.

With such a small time interval, unanticipated or spontaneous CPU usage could heavily skew the performance measurement. For this reason, we ran each test multiple times and averaged the performance measurement. To minimize thread overhead influencing the work unit performance measurement, we performed each work unit *n* times (where *n* can vary for each run).

Successes

Although we did not completely accomplish our primary goals, we did answer the questions from our secondary goals. Several performance improvements were implemented based on the data that we collected and the problems that we uncovered. Also, by developing the testing harness, we created a tool to test future releases, to compare different hardware configurations, and to use for targeted performance evaluations. A big success was the knowledge that we gained about performance testing. With this knowledge, everyone on the team, including development, has a better understanding of what can be done in the future.

Failures

Because we did not begin with usage profiles, we could not test scenarios. So, when an application developer noted that I/O was slow, we had to figure out how they were using it and analyze their usage before we could make suggestions. If we had begun with usage profiles, we could have offered direction on how to get the best performance from the routines.

Functional testing took a back seat to performance testing, creating a lack of functional coverage. Also, it was difficult to provide trustworthy, benchmark data on libraries that were not completely tested and that were constantly changing.

As we became more knowledgeable in the area of performance testing, we got caught up in the philosophy of performance testing. Some people wanted us to simulate scenarios and performance/load/stress/scalability test those scenarios. Others thought our goal should be to narrow it down to routines to find tunable areas (i.e. - if you are getting large chunks of memory, using these parameters will improve performance). We decided to benchmark the routines, and management agreed that this was a great first step. When we presented our plan to the developers in the prioritization meeting, it seemed that they accepted this direction. However, in their minds, we still were not doing what they thought we needed to do. Some developers wanted real-world scenarios and thought that our test driver was too fabricated, so that the data was invalid. Some developers considered the routines too close to the OS, so that there was not enough room for change. Some developers considered our data useless and walked away from the discussion. There was very little trust in the data we provided.

We also debated (and never concluded) how to measure tasks. Do you measure operations per second or time to do a task? Should the measurement be in microseconds, milliseconds? What granularity should we use? Looking back these questions seem silly, but we spent precious resources trying to make everyone happy with our results presentation.

Some of the underlying goals had no direction on how to achieve them. We generated mounds of data, but there was no analysis planning. For example, time to read a page is meaningless unless compared against something. Instead of stating just that we were going to measure time it takes to read a page at various page sizes, it would have been better to say we would measure the time it takes to read a page at various page sizes **to determine** optimal minimum and maximum page size.

Neither primary goal was entirely accomplished:

- Because of previously stated failures, benchmark data was not reliable or trusted.
- Because of time limitations, we did not complete all priority one measurements on all hosts.

Lessons Learned

Keep developers in the loop from the planning process throughout testing to create an environment of open communication and understanding. Buy-in from all participants is critical to establishing trust in the conclusions reached.

Provide definite goals and metrics to be collected. Make hypotheses, and prove or disprove those hypotheses based on the data collected.

Because we now have consumers of the routines, we can develop usage profiles. We are using coverage analysis to determine what parts of the libraries consumers are using and with what parameters those libraries are used.