Accept() Scalability on Linux

Stephen P Molloy, *University of Michigan*
smolloy@engin.umich.edu

Chuck Lever, *Sun-Netscape Alliance*
chuckl@netscape.com

Linux Scalability Project
Center for Information Technology Integration
University of Michigan, Ann Arbor

linux-scalability@citi.umich.edu
http://www.citi.umich.edu/projects/linux-scalability

Abstract

This report explores the possible effects of a "thundering herd" problem associated with the Linux implementation of the POSIX accept() system call. We discuss the nature of the problem and how it may affect the scalability of the Linux kernel. In addition, we identify candidate solutions and considerations to keep in mind. Finally, we present a solution and benchmark it, giving a description of the benchmark methodology and the results of the benchmark.

1. Introduction

Offered loads on network servers that use TCP/IP to communicate with their clients are rapidly increasing. A service may elect to create multiple threads or processes to wait for increasing numbers of concurrent incoming connections. By pre-creating these multiple threads, a network server can handle new connections and requests at a faster rate than with a single thread.

In recent years, the term scalability has been used to describe a number of different characteristics, so it may be useful to present our use now. Traditionally, scalability has meant that system performance changes in direct proportion to system resources. For this to be the case, all operations would have to be executed in constant time. Of course, it's impossible to have a system which achieves perfect scalability, but we can certainly try. For our purposes, we will use this interpretation of scalability. We feel that regardless of how many threads are waiting on a socket's wait queue, an accept() system call should execute in near-constant time.

This document was written as part of the Linux Scalability Project. The work described in this paper was supported via generous grants from the Sun-Netscape Alliance, Intel, Dell, and IBM.

In Linux, when multiple threads call accept() on the same TCP socket to wait for incoming TCP connections, they are placed into a structure called a wait queue. Wait queues are a linked list of threads that wait for some event. In the Linux 2.2 series kernel, when an incoming TCP connection is accepted, the `wake_up_interruptible()` function is invoked to awaken waiting threads. This function walks the socket's wait queue and awakens everybody. All but one of the threads, however, will put themselves back on the wait queue to wait for the next connection. This unnecessary awakening is commonly referred to as a "thundering herd" problem and creates scalability problems for network server applications.

This report explores the effects of the "thundering herd" problem associated with the `accept()` system call as implemented in the Linux kernel. In the rest of this paper, we discuss the nature of the problem and how it affects the scalability of network server applications running on Linux. We will investigate how other operating systems have dealt with the problem and finally, we will benchmark our solutions. All benchmarks and patches are against the Linux 2.2.14 kernel.

2. Background

This section is intended to give a detailed view of the current implementation of `accept()` and its problems. It will describe what we found in our initial research and explain the implications of each discovery. For the sake of comparison we will do the same for another widely used operating system, OpenBSD. At the end of the section we will layout the guidelines we used when formulating solutions to the problem.

2.1 Investigation

When a thread wants to listen for an incoming TCP connection, it creates a TCP socket and invokes the `accept()` system call. The system call uses the protocol specific `tcp_accept()` function to do all the work. The relevant sections of this procedure are the manipulations of the thread's state and socket's wait queue. In Linux, a thread's context is represented by a structure (`struct task_struct`) which maintains several variables pertaining to memory allocation and runtime statistics. One of these variables is named `state`. The `state` variable is used as a bitmask to indicate whether a thread is running, sleeping, waiting for an interrupt or yielding to an interrupt. Currently, when a thread calls `accept()` on a TCP socket the thread's state is changed from `TASK_RUNNING` to `TASK_INTERRUPTIBLE` and the thread is placed at the end of the wait queue associated with the socket. At this point, the thread puts itself to sleep and the system resumes normal operation. Every thread accepting on a socket follows this procedure, thus lengthening the wait queue whenever multiple threads `accept()` on the same socket.

The second part of this routine occurs each time another process (local or remote) initiates a TCP connection with the accepting socket. When the connection comes in, the network interface pulls the packet into kernel memory and passes it to the function `tcp_v4_rcv()`. This function parses the TCP packet header and identifies it as an attempt to connect with a listening socket. The TCP stack then calls `wake_up_interruptible()` on the corresponding socket's wait queue to wake and signal a thread to handle the new connection.

To completely understand how the Linux TCP stack awakens threads on a socket's wait queue requires a bit more detail. The socket structure in Linux contains a virtual operations vector that lists six methods (referred to as call-backs in some kernel comments). These methods initially point to a set of generic functions for all sockets when each socket is created. Each socket protocol family (e.g., TCP) has the option to override these default functions and point the method to a function specific to the protocol family. TCP overrides just one of these methods for TCP sockets. The four most commonly-used socket methods for TCP sockets are:

`sock->state_change`
 (pointer to sock_def_wakeup)
`sock->data_ready`
 (pointer to sock_def_readable)
`sock->write_space`
 (pointer to tcp_write_space)
`sock->error_report`
 (pointer to sock_def_error_report)

The code for each one of these methods invokes the `wake_up_interruptible()` function. This means that every time one of these methods is called, tasks could be unnecessarily awakened. In fact, in the `accept()` routine alone, Linux invokes three of these methods, essentially tripling impact of the "thundering herd" problem. The three methods invoked to wake tasks on a socket's wait queue are `tcp_write_space()`, `sock_def_readable()` and `sock_def_wakeup()`, in that order.

Because the most frequently used socket methods all call `wake_up_interruptible()`, the thundering herd problem potentially extends beyond the `accept()` system call and into the rest of the TCP code. In fact, it is rarely necessary for these methods to wake up the entire wait queue. Thus, almost any TCP socket operation could unnecessarily awaken tasks and return them to sleep. This inefficient practice robs valuable CPU cycles from server applications.

2.2 Comparison

In investigating the characteristics of thundering herd issues in Linux, we thought it might be a good idea to see how other systems deal with the issue. In particular, we examined the OpenBSD system to see how it behaves in the `accept()` system call. In OpenBSD 2.6, when a thread calls `accept()` on a socket, the thread puts itself to sleep with a socket specific identifier. When a connection is made to a socket, the kernel wakes up all threads sleeping on that socket's identifier. So it would appear that OpenBSD has the same thundering herd issues as Linux, but this is not the case. The OpenBSD kernel serializes all calls to `accept()`, so only one thread is waiting for a particular socket at any time. Although this approach prevents the thundering herd condition, it also limits performance, as we will see in section 5.

2.3 Guidelines

When developing solutions to any problem, it its important to establish a few rules to warrant acceptability and quality. While investigating the Linux TCP code, we set forth this particular set of guidelines to ensure the correctness and quality of our solution:

• *Don't break any existing system calls* - If the changes affect the behavior of any other system calls in an unexpected way, then the solution is unacceptable.

• *Preserve "wake everybody" behavior for calls that rely on it* - Some calls may rely on the "wake everybody" behavior of wake_up_interruptible(). Without this behavior, they may not conform to POSIX specifications.

• *Make solution as simple as possible* – The more complicated the solution, the more likely it is to break something or have bugs. Also, we want to try to keep the changes as local to the TCP code as possible so other parts of the kernel don't have to worry about tripping over the changed behavior.

• *Try not to change any familiar/expected interfaces unless absolutely necessary* - It would not be a good idea to require an extra flag to an existing function call. Not only would every use of that function have to be changed, but programmers who are used to its interface would have to learn to supply extra arguments.

• *Make the solution general, so it can be used by the entire kernel* - If any other parts of the kernel are experiencing a similar "thundering herd" problem, it may be easily fixed with this same solution instead of having to create a custom solution in other sections of the kernel.

3. Implementation

The fundamental idea behind solving the "thundering herd" problem is to somehow prevent all sleeping threads from waking up. This section will outline the implementation of a couple proposed solutions, including one that was incorporated into the 2.3 development series of the Linux kernel.

3.1 Task Exclusive

One proposed solution to this problem was suggested by the Linux community and incorporated into the 2.3 development kernel series. The idea is to add a flag to the threads state variable, change the handling of wait queues in `wake_up_interruptible()` and implement a new wait queue maintenance method called `add_wait_queue_exclusive()`. To use this solution, the soon to be sleeping thread would set the new `TASK_EXCLUSIVE` flag in the thread structure's state variable, then add itself to the wait queue using `add_wait_queue_exclusive()`. In the case of `accept()`, the protocol specific accept function (`tcp_accept()`) would be responsible for doing this work.

In handling the wait queue, `__wake_up()` (called by `wake_up_interruptible()`) will traverse the wait queue, waking threads as it goes until it runs into its first thread with the `TASK_EXCLUSIVE` flag set. It will wake this thread and then exit, leaving the rest of the queue waiting. To ensure that all threads that are not marked exclusive were awakened, `add_wait_queue()` will add threads to the front of a wait queue, while `add_wait_queue_exclusive()` will add exclusive threads to the end of a wait queue, after all non-exclusive waiters. Programmers are responsible for making sure that all exclusive threads are added to the wait queue with `add_wait_queue_exclusive()`. Special handling is required to wake all exclusive waiters in abnormal situations (like listening sockets being closed unexpectedly).

3.2 Wake One

Another solution, stemming from the idea that the decision point for waking one or many threads should not be made until wake time, was developed here at CITI. Processes or interrupts that awaken threads on a wait queue are generally better able to determine whether they want to awaken one thread or many. This solution does not use a flag in the task structure* and doesn't use any special handling in `add_wait_queue()` or `add_wait_queue_exclusive()`. With respect to the guidelines above, we felt that the easiest way to implement a solution is to add new calls to complement `wake_up()` and `wake_up_interruptible()`. These new calls are `wake_one()` and `wake_one_interruptible()`. They are #defined macros, just like `wake_up()` and `wake_up_interruptible()` and take exactly the same arguments. The only difference is that an extra flag is sent to `__wake_up()` by these macros, telling the system to wake only one thread instead of all of them. This way it's up to the waking thread whether it wants to wake one (e.g., to accept a connection) or wake all (e.g., to tell everyone the socket is closed).

For this "wake one" solution we examined the four most commonly used TCP socket methods and decided which should call `wake_up_interruptible()` and which should call `wake_one_interruptible()`. Where we elected to use `wake_one_interruptible()`, and the method was the default method for all socket protocols, we created a duplicate function just for TCP to be used instead of the default. We did this so the changes would affect only the TCP code, and not affect any other working socket protocols. If at some point later it is decided that `wake_one_interruptible()` should be the generic socket default, then the new TCP specific methods can be eliminated. Based on our interpretation of how each socket method is used, here's what we came up with:

`sock->state_change` - (tcp_wakeup)
wake_one_interruptible()

`sock->data_ready` - (tcp_data_ready)
wake_one_interruptible()

`sock->write_space` - (tcp_write_space)
wake_one_interruptible()

`sock->error_report` (sock_def_error_report)
wake_one_interruptible()

Notice that all three of the methods used in `accept()` call `wake_one_interruptible()` instead of `wake_up_interruptible()` when this solution is applied. The main obstacle with this approach is that system calls like `select()` depend on being awoken every time, even if there are threads ahead of them on the wait queue.

3.3 Always Wake

A third solution, which has not yet been implemented, combines the most desirable characteristics of the two previous solutions. The decision to wake one or many threads would still be deferred until the time of awakening by using `wake_one()` and `wake_one_interruptible()`. However, for the rare case where a thread would always need to wake up (like `select()`), a bit in the threads state could be set to indicate this. These threads would reside at the front of the wait queue and always be awoken on calls to `__wake_up()`. This solution is still easy for programmers to use, and only requires special care for the special cases. It gives the power to decide between awakening one or many threads to the more informed waking thread, while still providing a mechanism for the sleeper to make the decision if it knows better.

4. Performance Evaluation

Our focus is on improving system throughput. In this case, we hope to accomplish our goal by eliminating unnecessary kernel state CPU activity. To measure the performance of each solution we consider two questions. First, how long does it take for all threads to return to the wait queue after a TCP connection is initiated? Second, how does a network service perform under high load/stress situations with the new solutions? We took two different approaches to benchmarking the performance impact of the "wake one" and "task

* Although, there is a set of flags passed to __wake_up() that resemble the state variable in the task structure, i.e., the flags are set with the same bit masks as those used for the task structure. TASK_EXCLUSIVE is still #defined and passed as a bit to __wake_up() even though it is not used in the task structure.

Threads	Stock	TaskEx	WakeOne
100	4708	649	945
200	11283	630	1138
300	21185	891	813
400	41210	776	1126
500	52144	567	1275
600	75787	1044	599
700	96134	1235	707
800	118339	1368	784
900	149998	1567	1181
1000	177274	1775	843

Table I: The results of the microbenchmark (in usecs) are very rough estimates. But even at such a level of granularity, they still show significant improvement in settle time for the patched kernels over the stock kernel

exclusive" patches. The first is a simple micro-benchmark that is easy to set up and quick to run. We ran this to

get a rough idea of what sort of improvement we can expect with each patch. The other is a large-scale macro-benchmark on the patched kernels, to see if the patch improves performance under high loads as well.

4.1 Small Scale Performance

To measure how much time it takes for all unused threads to return to the wait queue after a connection is made, we wrote a small server program that spins **X** number of threads and has each of them accept on the same port. We also wrote a small client program that creates a socket and connects to the port on the server **Y** (in this case 1) times. We issue a `printk()` from the kernel every time a task is put on or removed from the wait queue. After the client has "tapped" the server, we examine the output of the `printk()`'s and identify the points where the connection was first acknowledged (in terms of wait queue activity) and where all threads have returned to the wait queue.

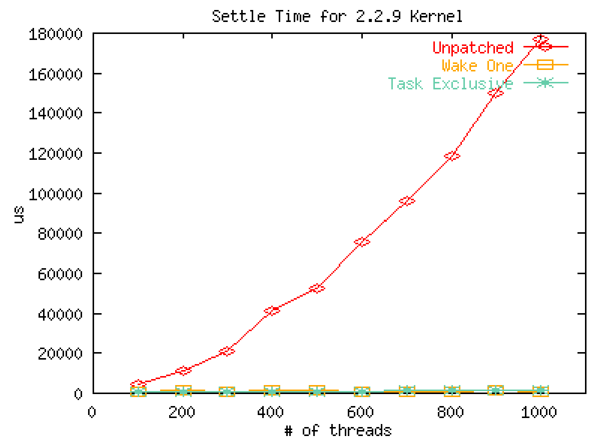

Figure I: This graph shows the difference in the time complexity between the stock kernel and the ones patched with thundering herd solutions.

The results are reported as an estimated elapsed time for the wait queue to settle down after an `accept()` call is processed. The measurements are not exact, as we were using `printk()`s and only ran the tests once. These two points can result in a slight skew of the results in three ways. First, `printk()`'s are not free operations and add to the execution time each time they are used. Second, to provide less room for statistical error, many samples should be taken, but these tests were only run once and could produce slightly different results on subsequent runs. However, even with these degrees of inaccuracy, this micro-benchmark is still able to give us a rough estimation of the time complexity involved with each scenario. Table I gives the settling time for stock and patched kernels with various numbers of threads on the wait queue. The server was running Linux 2.2.9 on a Dell PowerEdge 6300 with four 450 MHz Pentium II Xeon processors, a 100 Mbps Ethernet card and 512M of RAM (lent to the Linux Scalability Project by Intel).

The key observation to be made when looking at these rough estimates is the difference in time complexity. While the stock kernel settles in **O(n)** time, both of the patched kernels settle in nearly constant time. Figure I illustrates these differences.

4.2 Large Scale Performance

To set up the test harness for this benchmark, the Linux Scalability Project purchased new machines for use as clients against a web server. Four client machines are equipped with AMD K6-2's running at 400 MHz and 100 Mbps Ethernet cards. The server is a four processor Dell PowerEdge 6300 running with 400 MHz Pentium II Xeon processors, 512M of RAM and a 100 Mbps Ethernet card. The clients are all connected to the server through a 100 Mbps Ethernet switch. All machines used in the test are running a 2.2.14 Linux kernel. The server runs Red Hat Linux 6.0 with a stock 2.2.14 kernel as well as the "task exclusive" and "wake one" patched 2.2.14 kernels.

We elected to use the Apache web server as our network service because it's a widely used application and is easily modified to make this test more useful. Stock Apache 1.3.6 uses a locking system on Linux to prevent multiple `httpd` processes from calling `accept()` on the same port at the same time, which is intended to reduce errors and improve performance in production web servers. For our purposes, we want to see how the web serving machine will react when multiple httpd processes all call `accept()` at once. We modified Apache so that it doesn't wait to obtain a lock before calling `accept()`. This non-locking behavior is the default on systems where multiple `accept()`s are safe. The patch for this modification can be found on our web page at:
`www.citi.umich.edu/projects/linux-scalability`

To stress-test our web server, we used a pre-release version of SPEC's SpecWeb99 benchmark, courtesy of Netscape's web server development team. Because the benchmark is pre-release, SPEC rules constrain us from publishing detailed throughput results. However, we can still make general quantitative statements about the performance improvements.

Running the benchmark maintains between 300 and 1000 simultaneous connections to the web server from the client machines and measures throughput by requesting as many web pages as possible. Each connection requests a web page and then dies off while a new connection is generated to take its place. The Apache web server is configured to use 200 `httpd` daemons and does not support keep-alive connections (so idle connections do not linger). All `httpd` daemons accept on the same port. The throughput is measured by SpecWeb99 in terms of how many requests per second each of the 300 to 1000 simultaneous connections can make.

The results of the SpecWeb99 runs are very encouraging. While running with moderate to sizable loads of 300 to 1000 simultaneous connections to the web server, the number of requests serviced per second increased dramatically with both the "wake one" and "task exclusive" patches. While the performance impact is not as powerful as that evidenced by our microbenchmark, a considerable gain is evident in the testing. The performance increase due to either patch remains steady at just over 50% for all connection rates. There is no discernable difference between the "wake one" and "task exclusive" patches.

5. Application

Up to this point, the evaluation of the elimination of thundering herd problems seems overwhelmingly positive. However, there is one issue that seems unresolved. In the performance testing, SpecWeb99 was run against a modified Apache web server. Why did we put forth the effort to modify our web server and why would anybody want to do so in practice? To answer these questions, we performed a short evaluation of the stock Apache 1.3.9 web server and our patched version.

The stock Apache web server uses various locking schemes to prevent the servers threads from all calling `accept()` at the same time. This is done to prevent internal errors when the server receives connections on many different IP addresses or ports. When running an Apache web server on one IP address and one port, locking around `accept()` is not necessary.

If Apache server threads were all allowed to call `accept()` at the same time, then each thread could process a good portion of the `accept()` system call before a connection is even received. This in turn would reduce the effective overhead of accepting each incoming connection, since half the work is already done. To test this idea, we set up another test against a uniprocessor machine which would show the usefulness of these thundering-herd solutions on more common hardware.

This evaluation used a single processor AMD K6-2 machine running at 400 MHz equipped with a 100 Mbps ethernet card and the same four processor machine described in the macro-benchmark section. The quad-processor was used as a client machine running `httperf` to ensure that the web serving host (and not the client) would be under a significant load. The client was tested using two different configurations: a stock 2.2.14 Linux kernel with a stock locking Apache 1.3.9

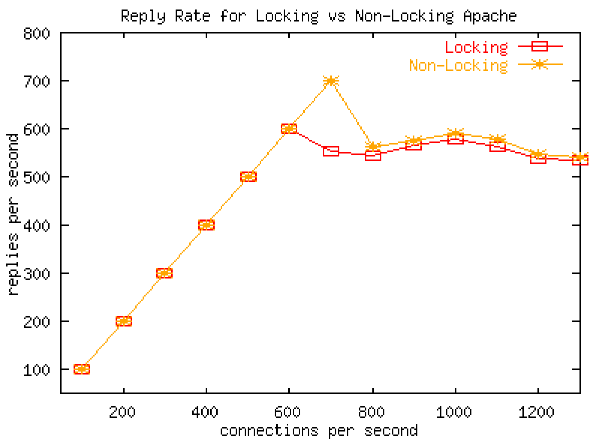

Figure II: This graph shows the rate at which the web server replied (y axis) for each level of client request-rates (x axis). Notice the point at which each server started to lose performance.

web server and the same kernel with the modified non-locking Apache 1.3.9 web server. The Apache web servers were configured to run a modest 20 serving threads (`httpd`'s) and to not support keep-alive connections.

The results of this test are plotted in Figure II. This graph demonstrates how Apache can increase the threshold rate at which it begins to fail by having all 20 `httpd's` accept at the same time, rather than deferring the accept overhead until later. You can imagine that if more `httpd`'s are started the difference in thresholds would decline, because on a stock 2.2.14 Linux kernel the system would begin to feel the effects of the thundering herd problem. It is not uncommon though, for medium to high traffic sites run more than 100 `httpd` processes.

6. Conclusion

By thoroughly studying this "thundering herd" problem, we have shown that it is indeed a bottleneck in high-load server performance, and that fixing it significantly improves the performance of a high-load server regardless of the method used. This performance increase is due to the fact that less time is spent in the kernel needlessly scheduling tasks which are not yet ready to run. All solutions presented resolve the issue by awakening as few tasks as necessary, thus reducing kernel overhead.

At first look, the "task exclusive" solution appears to be fairly complex. Upon closer examination though, it seems to fit in well with the new structure of Linux wait queues (doubly linked in 2.3 to make end-of-queue additions fast). Extra demands are placed on the programmer to get this solution to work, but the fix is extensible to all parts of the kernel and appears not to break any existing system calls. The "wake one" solution, on the other hand, is cleaner, easier for programmers to implement and is also extensible to all parts of the kernel. This fix is easily used by programmers since it requires just one line of code.

As previously mentioned, the process that awakens tasks is usually better able to determine if it wants to awaken one or more tasks. However, in the case of `select()`, the selecting process will want to be awakened regardless of whether or not it will continue on to handle the connection (perhaps it is monitoring the socket and collecting some statistics). For this case, the "task exclusive" model is a better fit. Conversely, if an application error occurs, a program may like to inform all of its associated tasks which are waiting on a socket. For this case, the "wake one" model is the better fit. Perhaps the most sound and elegant solution is the "always wake" hybrid of these two solutions which was presented in section 3.3.

6.1 Availability

All work and patches presented and used in this paper were written and performed at CITI and are available on the Linux Scalability Project's home page at http://www.citi.umich.edu/projects/linux-scalabilty/

6.2 Acknowledgements

Many Linux developers have contributed directly and indirectly to this effort. The authors are particularly grateful for input and contributions from Linus Torvalds and Andrea Arcangeli. Special thanks go to Dr. Charles Antonelli and Professor Gary Tyson for providing hardware used in the test harness for this report. The authors would also like to thank Peter Honeyman and Stephen Tweedie for their guidance, as well as the USENIX reviewers for their comments.

7. References

[1] M Beck, H Bohme, M Dziadzka, U Kunitz, R Magnus, D Verworner, Linux Kernel Internals, 2nd Ed., Addison-Wesley, 1998

[2] Samuel J Leffler, Marshall K McKusick, Micheal J Karels, The Design and Implementation of the 4.3BSD UNIX Operating System, Addison-Wesley, 1989

[3] Stevens, W Richard, UNIX Network Programming, Volume 1: Networking APIs: Sockets and XTI, 2nd Ed., Prentice-Hall, Inc., 1998

[4] The Single UNIX Specification, Version 2, `www.opengroup.org/onlinepubs/7908799`

[5] Apache Server, The Apache Software Foundation. `www.apache.org`

[6] D. Mosberger and T. Jin, “httperf – A Tool for Measuring Web Server Performance,” *SIGMETRICS Workshop on Internet Server Performance, June 1998*.

[7] SPECWeb99, Standard Performance Evaluation Corporation. `www.spec.org`

[8] Apache Performance Tuning, The Apache Software Foundation. `www.apache.org/docs/misc/perf-tuning.html`