ADVANCED OPERATING SYSTEMS

TERM PROJECT WEEKLY REPORT

Under,

An-I Andy Wang

Shuanglong Zhang

By,

Akshara Dendi ([ad13r@my.fsu.edu](mailto:ad13r@my.fsu.edu))

Lehari Saggam ([ls13d@my.fsu.edu](mailto:ls13d@my.fsu.edu))

Sinduri Shyamala ([ss13w@my.fsu.edu](mailto:ss13w@my.fsu.edu))

**OPTION CHOSEN:** SURVEY PROJECT

**PROJECT TITLE:** A STUDY ON LINUX LOCK SYSTEMS.

**GOAL:** To do a detailed analysis on the Linux lock systems.

**TEAM MEMBERS:** Akshara Dendi ([ad13r@my.fsu.edu](mailto:ad13r@my.fsu.edu))

Lehari Saggam ([ls13d@my.fsu.edu](mailto:ls13d@my.fsu.edu))

Sinduri Shyamala ([ss13w@my.fsu.edu](mailto:ss13w@my.fsu.edu))

**SINDURI SHYAMALA’S CONTRIBUTION:**

Moving on from the evolution of the linux file systems to the various types of locks, we are trying to find out the differences between the locks of every version. Right now, we have read about different types of locks and we will continue reading about the locks for the next week as well. The Spinlock is one such lock, which was introduced, and this lock was very useful with the interrupt handlers. They have better performance and hence they were considered worth using. Spinlocks are either locked or unlocked. When two versions of linux are compared to know the differences, we usually compare the codes of the respective versions and try to understand the patch code and here we can know if the lock is in “locked” or “unlocked” state. This means that if a lock is set then the state is “locked” or if it is not set then it is “unlocked”. Sometimes, others can use the code that is in the patch and a loop will be created because of this. This loop is generally referred to as the “Spin”.

The spinlocks are better than the SMP’s because of their better performance. But the spinlocks must be carefully dealt because they are complex. At any given point of time, care should be taken that only one particular process will be analyzing or using the lock. This ultimately prevents any deadlocks in the patch. The spinlock is very useful on multiprocessor systems. It can be used on uniprocessor systems too but they have to be preemptive. If they are non-preemptive, there is a very high chance that the code might run into an infinite spin or loop and can never come out of it, no matter what.

The time for each spinlock should be small or limited. This is because, when different processes are fighting for one particular lock, time should be allocated for all the processes or threads equally failing which, some of the important processes might not be able to access the lock. There are several functions that can be used for spinlocks. Spin\_lock and spin\_unlock are the very basic ones that are used. Extensions to such functions are available and there are many such functions, which can be used to implement spinlocks.

Spinlocks however cannot be used for locking the data for very long durations. They are mostly useful for data with smaller durations of time because of the problems of spin or loop that can be encountered. We usually use locking to provide concurrency and synchronization between processes. Such locks can be used to bring about concurrency between various threads and a lot of development has been observed in the types of locks used in Linux. Semaphores, mutex etc., are some other locks that were introduced. With the introduction of new versions of Linux, different locks were created in order to enhance the performance and speed of each version. In the next week, we will try to analyze more about the different locks and also the differences in every version. Our final aim is to analyze the patch codes to know the differences.

References

[1] Linux Journal, August 01,2002 by Robert Love

[2] Linux Device Drivers, 3rd Edition By Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini, February 2005.

**AKSHARA DENDI’S CONTRIBUTION:**

This week my study was on semaphores (sleeping locks) and opportunistic locks and its functionalities. Using these locks the tasks are made to sleep and then waking up depending on situations created. We can synchronize our context using semaphores and they are mostly the structures presented. It consists of two main components: the waiting queue and the counter (usage count). The waiting queue contents processes that are to be blocked and usage count contents number of access. The operations to manipulate semaphores are down and up which decrements and increments the usage count respectively. Down\_interruptible() function call is used to acquire the semaphore, if the value is less than zero then the process is added to the waiting queue else the process gains the semaphore. The wait stores the address of the wait queue and sleepers indicate the flags. The init\_MUTEX() are used for exclusive access where 0 indicates free and 1 busy resources. The declare\_MUTEX() are the macros to gain the access. The read/write semaphores are similar to spin locks except that the waiting process are suspended insteaded of spinning. The read/write semaphores are strictly fifo and it increases concurrency and overall performance of kernel.

They are four types of opportunistic locks: level 1, level 2, batch and filter. In level 1, the client can cache read and write data in file as long as client alone has access. If some other client tries to gain access the server rejects the request. In level 2, the client can read the file locally by sending the request to the server along with other clients concurrently without modifying. In batch, the locks run for each line of the file using batch file. In filter, the file cannot be used for write or deletion and allows sharing.

References:

[1] Understanding the Linux Kernel, By Daniel P. Bovet, Marco Cesati.

[2] <http://www.superbase.com/service_tech_support_oplocks.html>

[3] Unreliable Guide To Locking, Paul Rusty Russell.

[4] Locking in Linux kernel, Galois.

[5] Locking in OS Kernels for SMP Systems,TU Berlin, March 2006.

[6] Opportunistic Locking and Read Caching on Microsoft Windows Networks A Data Access Worldwide White Paper by Dennis Piccioni.

**LEHARI SAGGAM’S CONTRIBUTION:**

**PAPERS READ:** Transactional Mutex Locks

**INFORMATION GATHERED FROM PAPERS:**

The paper “Transactional Mutex Locks” shows a transaction-like locking mechanism which tries to give the generality and strength of mutex locks without giving up the scalability and other features. Transactional mutex locks have avoided the major instrumentation issue which generally occurred in software transactional memory. When this was compared for reader/writer locks, it avoided the necessity for static data for those operations which are read-only and in the case of RCU and sequence locks, this avoided restrictions on programming idiom.

Transactional mutex locks have lower latency than the software transactional memory but this enables it to work potentially with the mutexes. The survey shows us the evaluation of the performance basing on some micro-benchmarks like x86, SPARC, and POWER architectures. Transactional mutex locks are built over software transactional memory by having the instrumental overheads and keeping the least storage. Contrary to the before software transactional memory, the paper shows the technique for limiting both programming mechanisms and potential scalability. By this, transactional mutex locks can be as short as one word of global metadata, two words of metadata and makes use of low per-access instrumentation.

We learnt about the creation of a simple STM algorithm of listing single word of global metadata which provides concurrency control. When this shows odd then this indicates that the writer is active and when it indicates even, this means that there are zero or more readers being active. Now, two words of metadata are stored per transaction i.e.., a local duplicate of the global orec and a count of the nesting depth. The instrumentation proves that an individual read/writes are performed significantly. When combined with an exception mechanism for rollback, it also proves exact behavior on contraries. Although, in transactions having a series of reads and writes, there is still effective redundancy, and this can be eliminated by making use simple static analysis.

The elimination of Thread-Local Storage is done by specifically storing per-thread metadata on the heap. Where every transactional access needs the address of the thread’s calling metadata, instead of adding an additional framework to every function call to give a reference to this metadata as most of these software transactional memories rely on thread-local storage. Having the OS support, Transactional Local Storage is almost available. Else, a costly pthread API call has TLS. As TML needs just two words of metadata, then depending on TLS is not required even when it is not costly. Rather, the compiler can position per-transaction metadata on the stack, and then work on functions accordingly.

**REFERENCES:**

[1] “Transactional Mutex Locks” by Luke Dalessandro, Dave Dice, Michael Scott, Nir Shavit, and Michael Spear.

URL: <https://anon.cs.rochester.edu/u/scott/papers/2010_EuroPar_TML.pdf>

[2] “Transactional Mutex Locks” by Michael F. Spear, Arrvindh Shriraman, Luke Dalessandro, and Michael L. Scott, Department of Computer Science, University of Rochester

<http://transact09.cs.washington.edu/12_paper.pdf>