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))

**DATE:** 3/2/2014

**LEHARI SAGGAM’S CONTRIBUTION:**

This week I have done my research on various kinds of locking primitives in the Linux kernel, why we need them and how kernel developers can use and implement them to write a safe code. In this research, I understood why kernel code requires locking, how it provides general rules for proper kernel locking semantics and then the various locking primitives in the Linux kernel. Linus has an excellent design decision of keeping SMP and UP kernels distinct. This allows a specific lock not to exist at all in a UP kernel. Various combinations of CONFIG\_SMP and CONFIG\_PREEMPT are compiled in varying lock support. It does not effect, although, to the developer which says lock everything appropriately and all situations will be covered.

**Spinlocks:** The most common and varying locking primitive in the kernel is the spinlock, defined in include/asm/spinlock.h and include/linux/spinlock.h. The spinlock is a simple single-holder lock. If a process tries to attempt and acquire a spinlock and if it is unavailable, the process will be keep trying (spinning) until it acquires the lock. This simplicity creates a small and fast lock. The use of spin\_lock\_irqsave() will try to disable and interrupts locally and provides spinlock on SMP. So, this covers both interrupt and SMP concurrency issues. With a call to spin\_unlock\_irqrestore(), interrupts are being restored to the state when the lock was actually acquired. With a UP kernel, the code compiles to the same.

### Semaphores: Semaphores in Linux are sleeping locks as they cause a task to sleep on contention, rather spin, they are implemented in situations where the lock-held time might be long. Conventionally, as they have the overhead of keeping a task to sleep and effectively waking it up, they must not be used where the lock-held time is short. Because they sleep, although, they can be used to synchronize user contexts whereas spinlocks cannot. In other words, it is safe and easy to block while holding a semaphore.Semaphores are represented by a structure i.e.., struct semaphore, which is defined in include/asm/semaphore.h. Semaphores are changed via two methods: down and up. The former tries to acquire the semaphore and blocks if it gives up. The later releases the semaphore, trying to wake up any tasks blocked through the way.

**Reader/Writer Locks:** Linux kernel has reader/writer variants that separate lock usage into two parts: reading and writing. As it is typically safe for many threads to read data concurrently, so long as nothing changes the data, reader/writer locks give various concurrent readers although only a single writer. If your data access naturally gets divided into clear reading and writing sequences, essentially with a large amount of reading than writing then reader/writer locks are often preferred. The reader/writer spinlock is known as rwlock and is used same as the standard spinlock, with an exception of separate reader/writer locking.

### The Big Kernel Lock: Linux has a global kernel lock, kernel\_flag that was initially introduced in kernel 2.0 which is the only SMP lock. During 2.2 and 2.4, much work was put into removing the global lock from the kernel and replacing it with finer-grained localized locks. But now, the global lock's use is very less. It still exists, although, and developers need to be careful when using it. The global kernel lock is known as the big kernel lock or BKL. This is a spinning lock that is recursive; i.e.., two consecutive requests for it will not deadlock the process. Furthermore, a process can sleep and can also enter the scheduler while handling the BKL. When the process holding the BKL enters the scheduler, the lock is made free so other processes can obtain it. These functions of the BKL helped simple during the introduction of SMP of 2.0 kernel series. However, they should provide plenty of reasons not to use the lock.

**AKSHARA DENDI’S CONTRIBUTION:**

This week my studied was on spin lock initialization, its operations and big kernel locks. spin\_lock\_init(x) and spin\_lock\_unlocked is used for initialization of lock. Spin\_lock\_string is used to acquire the available lock and do atomic operations and decrement the lock byte based on conditions. Spin\_unlock\_string is used to release the locks. Spin locks allow many users for one project and locks must be gained in order to access or for read/write using rwlock. To gain access to read we use read\_lock and in case some other process is also have the access or write lock then request loops or waits for the writer to complete first and calls to read lock fails. Read\_unlock releases the acquired read lock on objects. Write\_lock gives the access to write on objects and in case some other process is already having read/write access to objects then request waits or loops until they are finished and call to write lock returns fail. Write\_unlock releases the acquired write lock on objects and has the value of rw\_lock\_bias.

Big kernel lock is used in kernel for memory allocation, kernel initialization and configuration, creation and deletion of processes, query creation, adding and re4moving spaces etc. In kernel mode only single CPU can work but is has low SMP performance. Some of the operations of big kernel lock are void lock\_kernel(), void unlock\_kernel(), int kernel\_locked() for acquiring, releasing and checking the status respectively.

**REFERENCES:**

[1][http://www.linuxjournal.com/article/5833](http://www.linuxjournal.com/article/5833" \t "_blank)

[2]<http://www.linuxjournal.com/files/linuxjournal.com/linuxjournal/articles/058/5833/5833s1.html>

[3]Linux Core Kernel commentary 2nd Edition

**SINDURI SHYAMALA’S CONTRIBUTION:**

**PATCH MANAGEMENT:**

Managing the patches is a difficult process. Ever patch a different approach and no two patches will have the same approach for fixing bugs etc. Apart from fixing bugs, patches have many other features. They can be used to fix bugs, to develop different new functionalities, hardware’s etc. They can also be used to improve the performance of the kernels. Based on the type of features, the patch packages are installed and used. This can be done only for the updated version of the operating system. Any changes that are made in the code are displayed in the patches. There are 4 strategies for patch management. They are

**(a)Proactive patch management**

In most of the circumstances, the problem is already mentioned and patches are generated according to that. The important step in such situations is to identify the patches that have already been generated and to use them in a proper way which will help the user. Some problems such as memory and data corruptions, latency etc., can be observed and hence proactive patches will help in solving such problems. This type of patch management can be used to reduce the downtime, avoiding some issues related to the patches and this can also be used in situations where planning ahead is required. Solaris tools can be used for proactive patch management.

**(b)Relative patch management**

This type of patching is required in situations where immediate steps have to be taken to reduce the problems that occur while running the code. The latest patch can be applied in such situations and this will help reduce the errors that might occur. But there are issues with this approach. When the problem appears to be solved, we will never know if the patch has actually fixed it. Sometimes the patches might just change the system to avoid the problem at that time. Different patch tools from Solaris can be used to fix such issues.

**(c)Security patch management**

This type of patch management needs a whole new strategy to work with to deal with all the security issues. The security patch has to be installed even before the next task is performed. The tools used by proactive patch management and relative patch management can be used for security patch management.

**(d)Installing a new system**

Patching should be done when the system is installed. By doing so, system booting will not lose the patches and problems can be avoided. Patch automation tools are also available for patching. Most of them are from Solaris. Some of the tools are, Sun Connection 1.1.1 Satellite Edition, Patch check advanced, smpatch etc.

In the coming weeks, we will compare the different patches in the Linux kernels and show our work in the form of graphs.

**References**

[1] Patch Management Best Practices, Enda O’Connor, April 2008

[2] Creating a Patch and Vulnerability Management Program, Recommendations of the National Institute of Standards and Technology, Peter Mell, Tiffany Bergeron, David Henning, November 2005.