New Synchronizer

121001 Aditya Parikh

121007 Atman Jain

121012 Erali Shah

Contents

Idea....2
Synchronization....2
Synchronization in Linux....3
Memory barrier....3
Atomic operations....4
Synchronize with interrupts....4
Spin locks....4
Read-copy-update....5
Completion....5
Semaphore....5
Mutex....6
Wait queue....6
Evolution Stages....7
Final version....8
Conclusion....8
References....9

Idea

The idea was to develop a synchronization technique to achieve concurrency and atomicity between processes while using critical section. Also to link this synchronization module to a system call which will call this module when required.

Synchronization

Synchronization in terms of operating systems refers to the algorithms and methods that lead to smooth flow of processes and devise strategies that either prevent or avoid situations leading to deadlock starvation etc.

Semaphores, monitors are used to solve producer-consumer, readers-writers or dining philosopher's problem which in turn leads to process synchronization.

Mutual exclusion refers to process being independent of other process in its execution so that there is no problem with their resource allocation what so ever. Synchronization can be achieved this way.

Synchronization is needed under following conditions:

- In case of a race condition where process are competing among each other for the same resources
- To detect and protect the critical section from exception or interrupt handlers, and kernel threads
- In a multiprocessor environment

Synchronization in Linux

Kernel synchronization is extremely complex and subtle. On linux kernel synchronization is done at 2 levels

Hardware provides low-level atomic operations. These operations can be platform on which we can build high-level, synchronization primitives like semaphores and using this we can implement critical sections and build correct multi-threaded/multi-process programs thereby achieving synchronization

Low-level synchronization primitives in Linux

Memory barrier

Compiler is ordered from the synchronizer to complete all pending resource allocation access before allocating anymore.There are two sorts of barriers

Read memory barriers: prevent reordering of read instructions.

Write memory barriers: prevent reordering of write instructions.

Some of the Linux barrier operations are smp_mb which prevents read and write instruction reordering, smp_rmb for preventing read instruction reordering, smp_wmb for write, set_mb for assigning or allocating memory barrier (only for multiprocessors).

Atomic operations

Operations like atomic_read write or modify which have one aligned access to resources like memory provide a basic level of synchronization.

Basic atomic instructions in linux are atomic_read, atomic_write, atomic_add, atomic_sub, atomic_inc (incrementing), atomic_dec etc.

Synchronize with interrupts

Synchronization can be established by disabling all the interrupts when a process is in critical section.

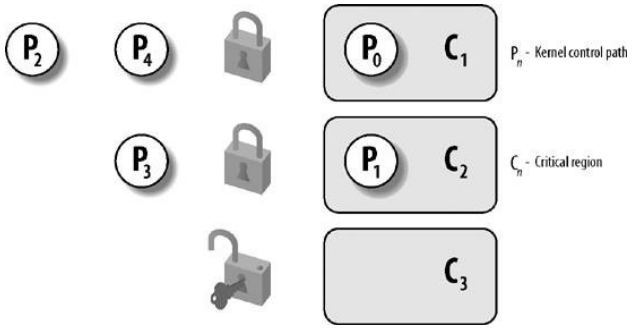
Irq_enable for enabling the interrupts, irq_disable for disabling, local_save_flags for saving the status before disabling and local_restore_flags to restore are some of the operations used widely.

Spin locks

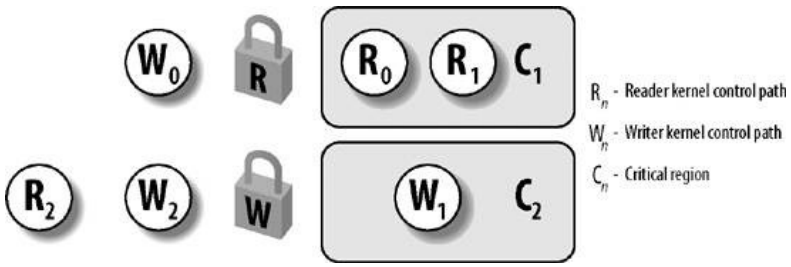
Spin locks are special types of locks implemented in Linux (more specifically for a multiprocessing environment). They lock the process with its resource thereby setting up mutual exclusion.

Spin_lock_init for initializing &spin_unlock for unlocking, spin_trylock&spin_lock for locking are mainly used to establish this.

There are multiple versions to spinlock: read/write spinlock where reader have priority. Seq lock (linux kernel 2.6) where writers have priority.



spinlock



Operating System
New Synchronizer

Read/write spinlock

Read-copy-update

Read-copy-update is a synchronization mechanism implemented in kernel 2.6. Another way of establishing mutual exclusion. Deals with multiple readers and writers. A key property of RCU is that readers can access a data structure even when it is in the process of being updated. No mechanism can force the reader to override their access. RCU ensures that reads are consistent by maintaining multiple versions of objects and ensuring that they are not freed up until all pre-existing read-side critical sections complete.

High-level synchronization primitives in Linux

Completion

Way to devise an execution order for two or more process which are in race condition to eachother. Works in somewhat of a similar fashion as in semaphores

Two operations or routine used to establish this complete () corresponding to up () which will tell the other process that it has finished execution and wait_for_completion() corresponding to down() which confirms or waits for the execution of complete from other process

Semaphore

It is essentially a locking mechanisms that allows other process waiting in queue for demanding resources in the critical section to sleep (by blocking) until the desired resources becomes free.

Linux offers two kinds of semaphores:

- Kernel semaphores, which are used in kernel mode
- System V IPC semaphores, which are used at application layer

Kernel semaphore is similar to the spinlocks except after locking when other process demands the same resource, it is blocked and put in a queue. i.e. the primary difference between a semaphore and spinlock is the process demanding the busy resource is spinned in spinlock and suspended in semaphore. At one place kernel processes poll and other works more like an interrupt.

Read/write semaphores are also present at kernel level and can be mapped to read/write spinlock.

up() is used to acquire a kernel semaphore and down() to release it

There are other set of functions that are specialized up and down in case of interrupts, timeout etc. implemented in newer versions of kernel.

Mutex

Mutex is short for mutual exclusion. While semaphore provide synchronization over processes, mutex provides synchronization of resource over multiple threads. (Essentially communicates with different threads) but note that these access over multiple threads cannot be simultaneous.

They really simple to implement and were implemented in kernel 2.6.16

90% of semaphores in kernel are used as mutexes, 9% of semaphores should be spin_locks quoted by Andrew Norton.

Functions or routine like mutex_unlock that release the mutex, mutex_lock in order to get the mutex (can block), mutex_trylock in order to try to get the mutex without blocking, mutex_is_locked for determining if mutex is locked etc. are used to implement mutexes at kernel level

Specialized functionalities like mutex_lock_interruptible are also implemented in the later versions of the kernel.

Wait queue

- Wait queue is used when a process is blocked due to unavailability of critical section. When a process is blocked it is stored into list.h file which is a doubly link list. And the process is swapped out once it acquires the semaphore.
- Functions used in semaphore of list.h: list_add_tail() which adds the process at the tail of the list. List_del() which will delete the process from the list.

Evolution Stages

- The final code was developed after many failed attempts and readings. We started with reading about the deadlock handling and synchronization achieving mechanisms in the Linux kernel.

- Initially we tried understanding few of the mutual exclusion file like semaphore.c, mutex.c, spinlock.c, semaphore.h, semaphore.h. As it was kernel level and synchronization in linux was associated with many of the files it was difficult to link the files and flow.

- In the latter ideas, we thought of implementing monitor structures with semaphore inside them. It turned out to be really complex while coding it.

- Then we tried to work upon functionalities that can be added into semaphores except those which are already

implemented. Addition to that we tried to implement semaphores using monitors. We thought of solving the reader-writers problem with writers having priority where in three semaphores were dealt with. One would be to maintain the atomicity of the writer, the second one being hindrance to other readers whilst the writer is writing and would ensure that a long queue is not allowed to build up on it, i.e. only one reader is allowed to queue on that semaphore and any additional readers must queue on a third semaphore. The need for this implementation is that the writers will have to jump the long queue if readers start queuing on second semaphore.

- Final version was to implement semaphore code of kernel in a binary fashion rather than a counter one using the same libraries of kernel.

Final version

- Semaphore in linux kernel is a counting semaphore. So we converted counting semaphore into binary semaphore. Important files associated are list.h, semaphore.h, sched.h.
- List.h has a doubly link list implementation which is used to store the blocked processes when the critical section is in used. When the semaphore is incremented, a process is taken from the queue from the list according to the scheduling algorithm declared in sched.h file.
- Once the process is blocked in to the queue, it continuously checks whether the semaphore is freed. Two functions: down() and up() which we have edited. Down() function will decremented the semaphore if value is greater than 1 else the

process is blocked into the queue and continuously keeps checking from the blocked queue. Similarly up() function will check whether the wait queue is emptied or not. If it is emptied then the semaphore value will increment to initial value else a process which was given critical access will be removed from the wait list and all other process which are in the wait list will be notified about the access.

- Every process that want to access critical section are assigned a state whether they are in in blocked state or runnable state or remove state. This process state is continuously checked and critical section is assigned accordingly.

Conclusion

In order to access mutual exclusion linux kernel implements different functions for different style of accessing critical section where clash between using the resource might occurred as resources are limited in operating system. We implemented one way of it by interchanging the functionalities of semaphore and mutex. Also to test whether it works or not we need to write printk statement and run a program which we can see it in dmseg file.

References

http://mvista.com/download/vision08/Linux-synchronization-mechanisms-in-driver-development-Vision2008.pdf

http://linux.linti.unlp.edu.ar/images/6/63/Ulk3-capitulo5.pdf

http://www.cs.columbia.edu/~junfeng/10sp-w4118/lectures/l11-synch-linux.pdf

http://www.makelinux.net/books/lkd2/ch08

https://lwn.net/Articles/262464/