FUTEX

Juhi Maheshwari

-121020

Index

1. Introduction

With significant growth in Linux as server operating system, all of the application suites are built as multi-process/multithreaded applications. These application suites are generally the collection of multiple independent parts. Though these parts are functionally independent, there must be some sort of communication among them to maintain mutual exclusion.

2. Mutex

Mutex is an object of a program that allows threads of multiple programs to share the same resource, but not simultaneously. Basically mutex implements mutual exclusion which ensures that no two concurrent processes are in their critical section at the same time. [Critical section refers to a period when the process accesses a shared resource.]

Mutex provides the wait queue which resides in user space. But access to that queue happens through system call to kernel. Thus every lock access requires a system call which becomes the over-head for the system.

The solution to the above problem is User level locking which removes some of the problems of pure kernel based locking mechanism.

3. User space locking

There are some traditional user space locking techniques available like spin locking, random fairness, etc. But due to some kind of problems there were not acceptable. And hence fast user space locking (FUTEX) is used nowadays.

4. Fast User space locking (FUTEX)

To understand Futex, we need to consider contended and uncontended case. The uncontended case should avoid system calls while in the contended case we are willing to perform system call.

Shared state in user space accessible to all requesting processes, is required to avoid system calls in uncontended case. This shared state is called user lock. User lock indicates the status of the lock which says about the resources whether it is free or acquired by other processor thread.

User lock is located in the shared memory region. In an uncontended case, any process or an application atomically changes the lock status word without entering the kernel. Hence this reduces overhead. While considering contended case application needs to wait for lock to get released and then acquire it. This again would turn very heavy.

Thus Futex provides the wait queue technique of mutex accessible in user space.

5. Implementation

Futex call takes six arguments. Futex can be implemented as follows

Static long sys_futex (void *addr1, int op, int val1, struct timespec *timeout, void *addr2, int val3)

- Addr1 – It tells about which wait-queue to use
- Op – op works as a multiplexer. There are about 13 operations which can be done by Futex. Op is used to selected a function
- Val1 – it is an argument to multiplexing function
- Timeout – a second int argument to the function.
- Addr2 – it is used to refer to second wait-queue whenever required.
- Val3 – the third argument to the function. This parameter is rarely used.

6. Various Operations

As mentioned above, 13 operations can be done using Futex. Some of them are as follows.

a. Futex_wait

This function tells the current thread to sleep in the wait queue. Now normally the thread sleeps only when it knows that it is going to be woken up. This is atomically checked by thread itself using val1 parameter. If the value pointed by addr1 and val1 matches then the thread will be put to sleep. Here the timeout parameter can be used to specify the time for which a thread can be put to sleep.

b. Futex_wake

This function is used to wake thread/ threads, sleeping in a wait-queue. It requires passing the value of addr1 and val1 as arguments to specify which thread to be woken up. If we require all threads to wake up, we simply need to pass INT_MAX function.

c. Futex_Requeue

This function allows us to move sleeping threads from one queue to another. Addr1 would be a value of source queue and addr2 will be value of destination queue. The val1 specifies how many thread to transfer.

d. Futex_Wait_Bitset and Futex_Wake_Bitset

These are the functions which are addition in Futex_wait and Futex_wake operations. The Futex_Wake_Bitset will only wake threads that have a specified bit set. Thus Futex_Wait_Bitset will allow the process waiting in the wait-queue to ignore wake up requests while Futex_Wake_Bitset will allow choosing which subset of waiters to wake.

e. Priority Inheritance

This function was added due to priority inversion problem.

7. Disadvantages

The Futex implementation doesn't allow choosing exactly one thread from pool of threads to wake up. We can choose set of threads or the process chosen would be random or based on some kind of algorithm.

In Futex implementation the question of priorities has remained since long. The current implementation is strictly FIFO based. We do have a kind of priority inversion operation but it is further possible to expand system call to use some kind of priority method.

8. Possible Future Extensions

Presently the size of Futex integer is 32 bits which can be incremented to 64 bits in near future for the expansion in Futex system call.

The current implementation of read-write locks in glibc uses a mutex to establish mutual exclusion between two operations. Two queues are assigned, one for readers and one for writers. The possible future extension is to remove the explicit outer lock and have some other kind of synchronization between them.

9. References

1. http://www.quora.com/How-different-is-a-futex-from-mutex-conceptually-and-also-implementation-wise
2. http://stackoverflow.com/questions/6364314/why-is-a-pthread-mutex-considered-slower-than-a-futex
3. http://man7.org/linux/man-pages/man7/futex.7.html
4. http://locklessinc.com/articles/mutex_cv_futex/
5. https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf
6. http://stackoverflow.com/questions/18559463/what-is-meant-by-fast-path-uncontended-synchronization
7. http://www.opensourceforu.com/2013/12/things-know-futexes/
8. http://locklessinc.com/articles/futex_cheat_sheet/
9. https://www.kernel.org/doc/Documentation/robust-futexes.txt