A

Seminar report

On

Futex

Submitted in partial fulfillment of the requirement for the award of degree
Of CSE

SUBMITTED TO: SUBMITTED BY:

www.studymafia.org www.studymafia.org

Preface

I have made this report file on the topic **Futex**; I have tried my best to elucidate all the relevant detail to the topic to be included in the report. While in the beginning I have tried to give a general view about this topic.

My efforts and wholehearted co-corporation of each and everyone has ended on a successful note. I express my sincere gratitude to …………..who assisting me throughout the preparation of this topic. I thank him for providing me the reinforcement, confidence and most importantly the track for the topic whenever I needed it.

Acknowledgement

I would like to thank respected Mr…….. and Mr. ……..for giving me such a wonderful opportunity to expand my knowledge for my own branch and giving me guidelines to present a seminar report. It helped me a lot to realize of what we study for.

Secondly, I would like to thank my parents who patiently helped me as i went through my work and helped to modify and eliminate some of the irrelevant or un-necessary stuffs.

Thirdly, I would like to thank my friends who helped me to make my work more organized and well-stacked till the end.

Next, I would thank Microsoft for developing such a wonderful tool like MS Word. It helped my work a lot to remain error-free.

Last but clearly not the least, I would thank The Almighty for giving me strength to complete my report on time.

CONTENTS

INTRODUCTION

In recent years, that is in past 5 years Linux has seen significant growth as a server operating system and has been successfully deployed as an enterprise for Web, file and print servicing.

With the advent of Kernel Version 2.4, Linux has seen a tremendous boost in scalability and robustness which further makes it feasible to deploy even more demanding enterprise applications such as high end database, business intelligence software ,application servers, etc. As a result, whole enterprise business suites and middleware such as SAP, Websphere, Oracle, etc., are now available on Linux.

For these enterprise applications to run efficiently on Linux, or on any other operating system, the OS must provide the proper abstractions and services. Usually these enterprise applications and applications suites or software are increasingly built as multi process / multithreaded applications.

These application suites are often a collection of multiple independent subsystems. Despite functional variations between these applications often they require to communicate with each other and also sometimes they need to share a common state. Examples of this are database systems, which typically maintain shared I/O buffers in user space.

Access to such shared state must be properly synchronized. Allowing multiple processes to access the same resources in a time sliced manner or potentially consecutively in the case of multiprocessor systems can cause many problems. This is due to the need to maintain

data consistency, maintain true temporal dependencies and to ensure that each thread will properly release the resource as required when it has completed its action. Synchronization can be established through locks. There are mainly two types of locks: - Exclusive locks and shared locks.

Exclusive locks are those which allows only a single user to access the protected entity, while shared locks are those which implements the multiple reader – single writer semantics.

Synchronization implies a shared state, indicating that a particular resource is available or busy, and a means to wait for its availability. The latter one can either be accomplished through busy-waiting or through an explicit / implicit call to the scheduler.

CONCURRENCY IN LINUX OS

As different processes interact with each other they may often need to access and modify shared section of code, memory locations and data. The section of code belonging to a process or thread which manipulates a variable which is also being manipulated by another process or thread is commonly called **critical section**. Proper synchronization problems usually serialize the access over critical section. Processes operate within their own virtual address space and are protected by the operating system from interference by other processes. By default a user process cannot communicate with another process unless it makes use of secure, kernel managed mechanisms. There are many times when processes will need to share common resources or synchronize their actions. One possibility is to use threads, which by definition can share memory within a process. This option is not always possible (or wise) due to the many disadvantages which can be experienced with threads. Methods of passing messages or data between processes are therefore required. In traditional UNIX systems the basic mechanisms for synchronization were System V IPC (inter process communication) such as ***semaphores***, ***msgqueues, sockets*** and the file locking mechanisms such as *flock()* and *fcntl()* functions. Message queues (msgqueues) consist of a linked list within the kernel's addressing space. Messages are added to the queue sequentially and may be retrieved from the queue in several different ways. Semaphores are counters used to control access to shared resources by multiple processes. They are most often used as a locking mechanism to prevent processes from accessing a particular resource while another

process is performing operations on it. Semaphores are implemented as sets, though a set may have a single member. Shared memory is a mapping of an area of memory into the address space of more than one process. This is the fastest form of IPC as processes do not subsequently need access to kernel services in order to share data. *fcntl()* locking implements locking directly via the kernel. This should work without problems on a local machine, but via NFS this requires locking support in the NFS server. The (old) user space nfsd does not support locking, while the kernel nfs server (knfsd), which comes with Linux 2.2 (and newer) supports locking. *fcntl()* locking will fail, if your NFS server doesn't support locking, so choosing the optimal locking technique does not only depend on the client machine but also on the NFS server machine.

While checking out for semaphores, the Famous Physicist and Computer Scientist **Edsger Wybe Dijkstra** had proposed the concept of semaphores. Semaphores supports two basic operations *UP* and *DOWN*. The initial value of semaphore is taken 1.The codes that are given by **Dijkstra** on semaphores are as follows :-

```
UP()
{
 Semaphore++;
}

DOWN()
{
while(Semaphore==0);
        Semaphore=0;
}
```

Actual Linux implementation of the semaphores in the kernel involves *UP* and *DOWN* to be coded as system calls. The *UP* system call, after incrementing the value of semaphore sends a wakeup signal to all the processes which are waiting for the same semaphore value to be 1 so as they can acquire it. (Wake up implies that the process which does *UP* of semaphore also change the task_structure's field 'status' to *TASK_RUNNING*). *DOWN* has a similar behavior proposed by **Dijkstra**. It simply sees whether the semaphore value is 1 else it sleeps within the *while()* loop. **Mr. NUTT** has provided the pseudo code as

```
DOWN (s) : [while (s==0) {wait}; s=s-1; ]
```

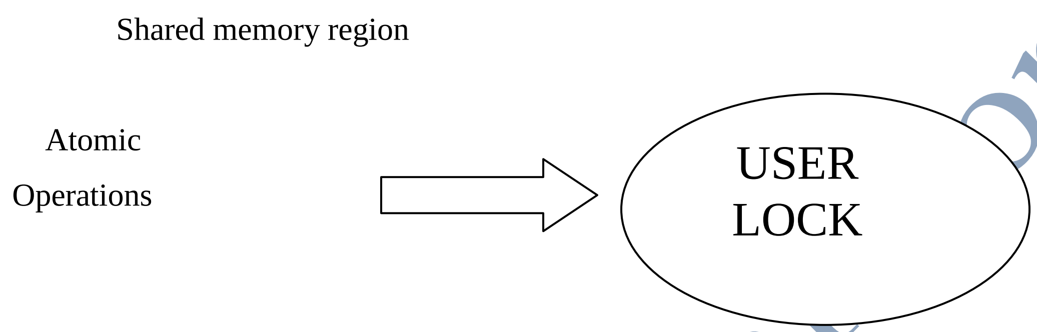
The code within square braces is indivisible and one in curly braces can be interrupted. Now to understand how these functions really achieve concurrency we choose an example where two processes A and B try to access the critical section. As the initial value of semaphore is one, when the two processes calls *DOWN()* concurrently, only one process will succeed in attaining the *DOWN()* operation. Say, suppose process A started executing *DOWN()* operation first. At this point, process B can’t

preempt A because *DOWN()* is a system call. Now process A sees that the value of semaphore is one and automatically decrements its value to zero and comes out of *DOWN()* without blocking. Now process B can preempt A because now A is off the system call. Now B starts executing *DOWN()* and checks the value of the semaphore which is zero at this moment and hence it goes to sleep and doesn’t come out of the system call. When process A has completed, it generates a wakeup call and thus B starts executing. That was the whole story of Linux semaphores (These IPC mechanisms are coded as C routines given in */usr/src/linux/ipc*. It is coded by an Indian – **Mr. Krishna Balasubramaniam**.)

Thus we see these mechanisms expose an opaque handle to a kernel object that naturally provides the shared state and atomic operations in the kernel. Services must be requested through system calls (eg :- *semop()*). The drawback of this approach is that every lock access requires a system call. When locks have low contention rates, the system call can constitute a significant overhead. A process may operate in one of two modes which are known as 'user' mode and 'system' mode (or kernel mode). A single process may switch between the two modes, i.e. they may be different phases of the same process. Processes defaulting to user mode include most application processes, these are executed within an isolated environment provided by the operating system such that multiple processes running on the same machine cannot interfere with each other's resources. A user processs switches to kernel mode when it makes a system call, generates an exception (fault) or when an interrupt occurs (e.g. system clock). At this point the kernel is executing on behalf

of the process. At any one time during its execution a process runs in the context of itself and the kernel runs in the context of the currently running process.

One solution to this problem is to deploy user level locking, which avoids some of the overhead associated with purely kernel-based locking mechanisms. It relies on a user level lock located in a shared memory region and modified through atomic operations to indicate the lock status. Only the contended case requires kernel intervention. The exact behavior and the obtainable performance are directly affected by how and when the kernel services are invoked.

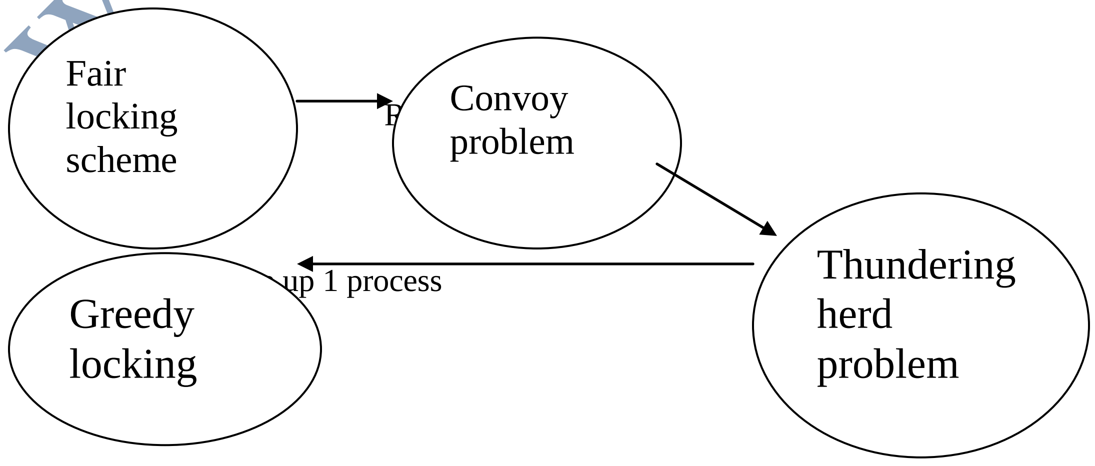


In this seminar, I am describing a particular fast user level locking mechanism called *futexes***,** that was developed in the context of the Linux operating system. It consists of two parts, the user library and a kernel service that has been integrated into the Linux kernel distribution version 2.5.7

PROBLEMS AND REQUIREMENTS IN IMPLEMENTATION

There are various behavioral requirements that need to be considered. Most center around the fairness of the locking scheme and the lock release policy. In a **fair** locking scheme the lock is granted in the order it was requested, i.e., it is handed over to the longest waiting task. This can have negative impact on throughput due to the increased number of context switches. At the same time it can lead to the so called **convoy problem**. Since, the locks are granted in the order of request arrival, they all proceed at the speed of the slowest process, slowing down all waiting processes. A common solution to the convoy problem has been to mark the lock available upon release, wake all waiting processes and have them recontend for the lock. This is referred to as **random fairness**, although higher priority tasks will usually have an advantage over lower priority ones. However, this also leads to the well known **thundering herd problem**. Despite this, it can work quite well on uni-processor systems if the first task to wake releases the lock before being preempted or scheduled, allowing the second herd member to obtain the lock, etc. It works less spectacularly on Symmetric Multi Processing Systems (SMP). To avoid this problem, one should only wake up one waiting task upon lock release. Marking the lock available as part of releasing it, gives the releasing task the opportunity to reacquire the lock immediately again, if so desired, and avoid unnecessary context switches and the convoy problem. Some refer to these as **greedy**, as the running task has the highest probability of reacquiring the lock if the lock

is hot. However, this can lead to starvation. Hence, the basic mechanisms must enable both fair locking, random locking and greedy or convoy avoidance locking (short ca-locking).



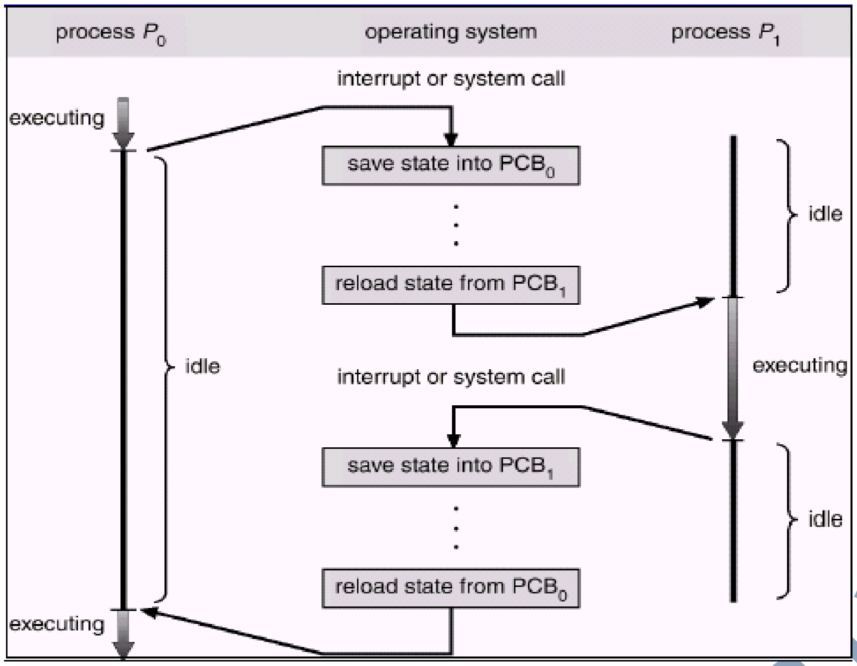
Another requirement is to enable spin locking, i.e., have an application spin for the availablilty of the lock for some user specified time (or until granted) before giving up and resolving to block in the kernel for its availability. Spin-locks are useful for short-critical sections. If you cannot avoid having a long critical section, you should not use spin-lock primitives in the first place, but use blocking synchronization primitives. Hence an application has the choice to either

A. Block waiting to be notified for the lock to be released, or
B. Yield the processor until the thread is rescheduled and then the lock is tried to be acquired again, or
C. Spin consuming CPU cycles until the lock is released.

Thus with respect to performance, there are basically two overriding goals:

- Avoid system calls if possible, as system calls typically consume several hundred instructions.
- Avoid unnecessary context switches: context switches lead to overhead associated with TLB invalidations. Each time a process is removed from access to the processor, sufficient information on its current operating state must be stored such that when it is again scheduled to run on the processor it can resume its operation from an identical position. This operational state data is known as its context and the act of removing the process's thread of execution from the processor (and replacing it with another) is known as a process switch or context switch.

A typical process switch or context switch involves the following:



Another requirement is that fast user level locking should be simple enough to provide the basic foundation to efficiently enable more complicated synchronization constructs, e.g. semaphores, rwlocks, blocking locks, or spin versions of these, pthread mutexes, DB latches..

It should also allow for a clean separation of the blocking requirements towards the kernel, so that the blocking only has to be implemented with a small set of different constructs. This allows for extending the use of the basic primitives without kernel modifications. Of interest is the implementation of mutex, semaphores and multiple reader/single writer locks.

Finally, a solution needs to be found that enables the recovery of “dead” locks. Deadlock is a permanent blocking of a set of processes that either compute for system resources or communicate with each other. Deadlock may be addressed by mutual exclusion or by deadlock avoidance. Mutual exclusion prevents two threads accessing the same resource simultaneously. Deadlock avoidance can include initiation denial or allocation denial, both of which serve to eliminate the state required for deadlock before it arises. We define unrecoverable locks as those that have been acquired by a process and the process terminates without releasing the lock. There are no convenient means for the kernel or for the other processes to determine which locks are currently held by a particular process, as lock acquisition can be achieved through user memory manipulation. Each process has some form of associated Process Identifier, (PID) through which it may be manipulated. The process also carries the User Identifier (UID) of the person who initiated

the process and will also have group identifier (GID). Registering the process's "pid" after lock acquisition is not enough as both operations are not atomic. If the process dies before it can register its pid or if it cleared its pid and before being able the release the lock, the lock is unrecoverable.

LINUX FAST USER LEVEL LOCKING:

HISTORY AND IMPLEMENTATION

Having stated the requirements in the previous section, we now proceed to describe the basic general implementation issues. As told before, futex is a fast user level locking mechanism and hence it is called as *F*ast *U*serspace mu*TEX*. Futexes are very basic and lend themselves well for building higher level locking abstractions such as POSIX mutexes. Most programmers will in fact not be using futexes directly but instead rely on system libraries built on them, such as the NPTL pthreads implementation. A futex is identified by a piece of memory which can be shared between different processes. In fast userlevel locking, there are mainly two cases upon which it has been implemented. They are: The uncontended case and the contended case.

The uncontended case should be efficient and should avoid system calls by all means. In the contended case we are willing to perform a system call to block in the kernel. Avoiding system calls in the uncontended case requires a shared state in user space accessible to all participating processes/task. This shared state, referred to as the ***user lock,*** indicates the status of the lock, i.e., whether the lock is held or not and whether there are waiting tasks or not. This is in contrast to the System V IPC mechanisms which merely exports a handle to the user, and performs all operations in the kernel. The user lock is located in a shared memory region that was create via *shmat()* or *mmap().*

As a result, it can be located at different virtual addresses in different address spaces. In the uncontended case, the application atomically changes the lock status word without entering into the kernel. Hence, atomic operations such as *atomic_inc(), atomic_dec(), cmpxchg(),* and *test_and_set()* are neccessary in user space.

In the contended case, the application needs to wait for the release of the lock or needs to wake up a waiting task in the case of an unlock operation. In order to wait in the kernel, a

kernel object is required, that has ***waiting queues*** associated with it. The waiting queues provide the queueing and scheduling interactions. Of course, the aforementioned IPC mechanisms can be used for this purpose. However, these objects still imply a heavy weight object that requires a priori allocation and often does not precisely provide the required functionality. Another alternative that is commonly deployed are ***spinlocks*** where the task spins on the availability of the user lock until granted. To avoid too many cpu cycles being wasted, the task yields the processor occasionally.

It is desirable to have the user lock be handlefree. In other words instead of handling an oqaque ***kernel handle,*** requiring initialization and cross process global handles, it is desirable to address locks directly through their virtual address. As a consequence, kernel objects can be allocated dynamically and on demand, rather than apriori. A lock, though addressed by a virtual address, can be identified conceptually through its ***global lock identity,*** which we define by the memory object backing the virtual address and the offset within that object. It is given by the tuple [B,O]. Since B represents the memory object backing the kernel object,it can be any of the three fundamental types. They are:

(a) Anonymous memory,
(b) Shared memory segment, and
(c) Memory mapped files.

While (b) and (c) can be used between multiple processes, (a) can only be used between threads of the same process. Utilizing the virtual address of the lock as the kernel handle also provides for an integrated access mechanism that ties the virtual address automatically with its kernel object. Despite the atomic manipulation of the user level lock word, race conditions can still exist as the sequence of lock word manipulation and system calls is not atomic. This has to be resolved properly within the kernel to avoid deadlock and improper functioning.

Next the user lock object in futexes are described. For the purpose of this discussion a general opaque datatype *ulock_t* is defined to represent the userlevel lock. At a minimum it requires a status word :-

```
typedef struct ulock_t {
long status;
} ulock_t;
```

Here, it has got a “long” field which indicates the status of the user lock object. We assume that a shared memory region has been allocated either through *shmat()* or through *mmap()* and that any user locks are allocated into this region. Again, the addresses of the same lock need not be the same across all participating address spaces.

The basic semaphore functions *UP()* and *DOWN()* can be implemented as follows.

```
static inline int
usema_down(ulock_t *ulock)
{
if (!__ulock_down(ulock))
return 0;
return sys_ulock_wait(ulock);
}
static inline int
usema_up(ulock_t *ulock)
{
if (!__ulock_up(ulock))
return 0;
return sys_ulock_wakeup(ulock);
}
```

The *ulock_down()* and *ulock_up()* provide the atomic increment and decrement operations on the lock status word. A non positive count (status) indicates that the lock is not available. In addition, a negative count could indicate the number of waiting tasks in the kernel. If a contention is detected, i.e.

(ulock->status <=0), the kernel is invoked through the *sys_** functions to either wait on the wait queue associated with ulock or release a blocking task from said waitqueue. All counting is performed on the lock word and race conditions resulting from the non atomicity of the lock word must be resolved in the kernel. Due to such race conditions, a lock can receive a wakeup before the waiting process had a chance to enqueue itself into the kernel wait queue. We describe below how various implementation resolved this race condition as part of the kernel service.

PREVIOUS IMPLEMENTATIONS

One early design suggested was the explicit allocation of a kernel object and the export of the kernel object address as the handle. The kernel object was comprised of a wait queue and a unique security signature. On every wait or wakeup call, the signature would be verified to ensure that the handle passed indeed was referring to a valid kernel object. The disadvantages of this approach have been mentioned in section 2, namely that a handle needs to be stored in *ulock_t* and that explicit allocation and deallocation of the kernel object are required. Furthermore, security is limited to the length of the key and hypothetically could be guessed.

Another prototype implementation, known as ***ulocks***, implements general user semaphores with both fair and convoy avoidance wakeup policy. Mutual exclusive locks are regarded as a subset of the user semaphores. The prototype also provides multiple reader/single writer locks (rwlocks). The user lock object *ulock_t* consists of a lock word and an integer indicating the required number of kernel wait queues.

User semaphores and exclusive locks are implemented with one kernel wait queue and multiple reader/single writer locks are implemented with two kernel wait queues. This implementation separates the lock word from the kernel wait queues and other kernel objects, i.e., the lock word is never accessed from the kernel on the time critical wait and wakeup code path. Hence the state of the lock and the number of waiting tasks in the kernel is all recorded in the lock word. For exclusive locks, standard counting as described in the general *ulock_t* discussion, is implemented. As with general semaphores, a positive number indicates the number of times the semaphore can be acquired, “0” and less indicates that the lock is busy, while the absolute of a negative number indicates the number of waiting tasks in the kernel.

The “premature” wakeup call is handled by implementing the kernel internal waitqueues using kernel semaphores *(struct semaphore)* which are initialized with a value 0. A premature wakeup call, i.e. no pending waiter yet, simply increases the kernel semaphore’s count to 1. Once the pending wait arrives it simply decrements the count back to 0 and exits the system call without waiting in the kernel. All the wait queues (kernel semaphores) associated with a user lock are encapsulated in a single kernel object. In the rwlocks case, the lock word is split into three fields: *write locked* (1 bit), *writes waiting* (15 bits), *readers* (16 bits). If write locked, the *readers* indicate the number of tasks waiting to read the lock, if not write locked, it indicates the numbers of tasks that have acquired read access to the lock. Writers are blocking on a first kernel wait queue, while readers are blocking on a

second kernel wait queue associated with a ulock. To wakeup multiple pending read requests, the number of task to be woken up is passed through the system call interface. To implement rwlocks and ca-locks, atomic compare and exchange support is required. Unfortunately on older 386 platforms that is not the case.

The kernel routines must identify the kernel object that is associated with the user lock. Since the lock can be placed at different virtual addresses in different processes, a lookup has to be performed. In the common fast lookup, the virtual address of the user lock and the address space are hashed to a kernel object. If no hash entry exists, the proper global identity [B;O] of the lock must be established. For this we first scan the calling process’s vma list for the vma containing the lock word and its offset. The global identity is then looked up in a second hash table that links global identities with their associated kernel object. If no kernel object exists for this global identity, one is allocated, initialized and added to the hash functions. The *close()* function associated with a shared region holding kernel objects is intercepted, so that kernel objects are deleted and the hash tables are cleaned up, once all attached processes have detached from the shared region. While this implementation provides for all the requirements, the kernel infrastructure of multiple hash tables and lookups was deemed too heavy. In addition, the requirement for compare and exchange is also seen to be restrictive.

FUTEXES

The Linux kernel provides futexes ('_F_ast _U_serspace mu_Texes_') as a building block for fast userspace locking and semaphores. Futexes were designed and worked on by Hubertus Franke IBM Thomas J. Watson Research Center, Matthew Kirkwood, Ingo Molnar (Red Hat) and Rusty Russell (IBM Linux Technology Center). Futexes are very basic and lend themselves well for building higher level locking abstractions such as POSIX mutexes. Initial futex support was merged in Linux 2.5.7 but with different semantics from those described below. Current semantics are available from Linux 2.5.40 onwards. A futex is identified by a piece of memory which can be shared between different processes. In these different processes, it need not have identical addresses. In its bare form, a futex has semaphore semantics; it is a counter that can be incremented and decremented atomically; processes can wait for the value to become positive. Futex operation is entirely userspace for the non-contended case. The kernel is only involved to arbitrate the contended case. As any sane design will strive for non-contension, futexes are also optimised for this situation. In its bare form, a futex is an aligned integer which is only touched by atomic assembler instructions. Processes can share this integer over *mmap()*, via shared segments or because they share memory space, in which case the application is commonly called multithreaded. There are three key points of the original futex implementation which was added to the 2.5.7 kernel:

1. We use a unique identifier for each futex (which can be shared across different address spaces, so may have different virtual addresses in each): this identifier is

the *"struct page"* pointer and the offset within that page. We increment the reference count on the page so it cannot be swapped out while the process is sleeping.

2. The structure indicating which futex the process is sleeping on is placed in a hash table, and is created upon entry to the futex syscalls on the process's kernel stack.
3. The compression of "F̲ast U̲serspace mu*T̲E̲X̲*" into *"futex"* gave a simple unique identifier to the section of code and the function names used.

THE 2.5.7 IMPLEMENTATION

The initial implementation which was judged a sufficient basis for kernel inclusion used a single two-argument system call, "*sys_futex(struct futex *, int op)*". The first argument was the address of the futex, and the second was the operation, used to further demultiplex the system call and insulate the implementation somewhat from the problems of system call number allocation. The latter is especially important as the system call is expand as new operations are required. The *sys_futex* system call provides a method for a program to wait for a value at a given address to change, and a method to wake up anyone waiting on a particular address while the addresses for the same memory in separate processes may not be equal, the kernel maps

them internally so the same memory mapped in different locations will correspond for *sys_futex* calls. The two valid op numbers for this implementation were *FUTEX_UP* and *FUTEX_DOWN*. The algorithm was simple, the file *linux/kernel/futex.c* containing 140 code lines, and 233 in total. The algorithm implemented is as given below:

1. The user address was checked for alignment and that it did not overlap a page boundary.

2. The page is pinned: this involves looking up the address in the process's address space to find the appropriate *"struct page *",* and incrementing its reference count so it cannot be swapped out.
3. The *"struct page *"* and offset within the page are added, and that result hashed using the recently introduced fast multiplicative hashing routines to give a hash bucket in the futex hash table.
4. The *"op"* argument is then examined. If it is *FUTEX_DOWN* then:
   - (a) The process is marked *INTERRUPTIBLE*, meaning it is ready to sleep.
   - (b) A *"struct futex_q"* is chained to the tail of the hash bucket determined in step 3: the tail is chosen to give FIFO ordering for wakeups. This structures contains a pointer to the process and the *"struct page *"* and offset which identify the futex uniquely.
   - (d) The page is mapped into low memory (if it is a high memory page), and an atomic decrement of the futex address is attempted,4 then unmapped again. If this does not decrement the counter to zero, we check for signals (setting the error to *EINTR* and going to the next step), schedule, and then repeat this step.
   - (e) Otherwise, we now have the futex, or have received a signal, so we mark this process *RUNNING*, unlink ourselves from the hash table, and wake the next waiter if there is one, and return 0 or -*EINTR*. We have to wake another process so that it decrements the futex to -1 to indicate that it is waiting (in the case where we have the futex), or to avoid the race where a signal came in just as we were woken up to get the futex (in the case where a signal was received).

5. If the op argument was *FUTEX_UP:*
   - (a) Map the page into low memory if it is in a high memory page
   - (b) Set the count of the futex to one ("available").
   - (c) Unmap the page if it was mapped from high memory
   - (d) Search the hash table for the first *"struct futex_q"* associated with this futex, and wake up that process.

6. Otherwise, if the op argument is anything else, set the error to *EINVAL*.

7. Unpin the page.

While there are several subtleties in this implementation, it gives a second major advantage over System V semaphores: there are no explicit limits on how many futexes you can create, nor can one futex user

“starve” other users of futexes. This is because the futex is merely a memory location like any other until the *sys_futex* syscall is entered, and each process can only do one *sys_futex* syscall at a time, so we are limited to pinning one page per process into memory, at worst.

READ-WRITE LOCKS

We considered an implementation of “*FUTEX_ READ_DOWN*” et. al, which would be similar to the simple mutual exclusion locks, but before adding these to the kernel, Paul Mackerras suggested a design for creating read/write lock in userspace by using two futexes and a count: f ast userspace read/write locks, or *furwocks*. This implementation provides the benchmark for any kernel-based implementation to beat to justify its inclusion as a first-class primitive, which can be done by adding new valid “op” values.

PROBLEMS WITH THE 2.5.7 IMPLEMENTATION

Once the first implementation entered the mainstream experimental kernel, it drew the attention of a much wider audience. In particular those concerned with implementing POSIX threads, and attention also returned to the fairness issue.

- There is no straightforward way to implement the pthread_cond_timedwait primitive: this operation requires a timeout, but using a timer is difficult as these must not interfere with their use by any other code.
- The pthread_cond_broadcast primitive requires every process sleeping to be woken up, which does not fit well with the 2.5.7 implementation, where a process only exits the kernel when the futex has been successfully obtained or a signal is received.

- For N:M threading, such as the Next Generation Posix Threads project [5] an asynchronous interface for finding out about the futex is required, since a single process (containing multiple threads) might be interested in more than one futex.
- Starvation occurs in the following situation: a single process which immediately drops and then immediately competes for the lock will regain it before any woken process will.

MODIFICATIONS OF THE 2.5.7 IMPLEMENTATION

With these limitations brought to light, we searched for another design which would be flexible enough to cater for these diverse needs. After various implemenation attempts and discussions we settled on a variation of *atomic_compare_and_swap* primitive, with the atomicity guaranteed by passing the expected value into the kernel for checking. With these modifications, futex has been implemented in the Linux kernel version 2.6.

To do this, two new "op" values namely, *FUTEX_WAIT* and *FUTEX_WAKE* replaced the operations above, and the system call was changed to two additional arguments, *"int val"* and *"struct timespec *reltime"*.

FUTEX_WAIT: This operation atomically verifies that the futex address still contains the value given, and sleeps awaiting *FUTEX_WAKE* on this futex address. If the timeout argument is non-NULL, its contents describe the maximum duration of the wait, which is infinite otherwise. It is similar to the previous ***FUTEX_ DOWN***, except that the looping and manipulation of the counter is left to userspace. This works as follows:

1. Set the process state to *INTERRUPTIBLE*, and place *"struct futex_q"* into the hash table as before.
2. Map the page into low memory (if in high memory).
3. Read the futex value.
4. Unmap the page (if mapped at step 2).

5. If the value read at step 3 is not equal to the "*val*" argument provided to the system call, set the return to *EWOULDBLOCK*.
6. Otherwise, sleep for the time indicated by the "*reltime*" argument, or indefinitely if that is *NULL*.
   (a) If we timed out, set the return value to *ETIMEDOUT*.
   (b) Otherwise, if there is a signal pending, set the return value to *EINTR*.
7. Try to remove our "*struct futex_q*" from the hash table: if we were already removed, return 0 (success) unconditionally, as this means we were woken up, otherwise return the error code specified above.

FUTEX_WAKE: This is similar to the previous *FUTEX_UP*, except that it does not alter the futex value, it simple wakes one (or more) processes. The number of processes to wake is controlled by the "*int val*" parameter, and the return value for the system call is the number of processes actually woken and removed from the hash table. It returns the number of processes woken up.

FUTEX_AWAIT: This is proposed as an asynchronous operation to notify the process via a SIGIO-style mechanism when the value changes. The exact method has not yet been settled (see future work in Section 5). This new primitive is only slightly slower than the previous one,6 in that the time between waking the process and that process attempting to claim the lock has increased (as the lock claim is done in userspace on return from the *FUTEX_WAKE* syscall), and if the process has to attempt the lock multiple times before success, each attempt will be accompanied by a syscall, rather than the syscall claiming the lock itself. On the other hand, the following can be implemented entirely in the userspace library:

1. All the POSIX style locks, including *pthread_cond_broadcast* (which requires the "wake all" operation) and *pthread_cond_timedwait* (which requires the timeout argument). One of the authors (Rusty) has implemented a "nonpthreads" demonstration library which does exactly this.
2. Read-write locks in a single word, on architectures which support *cmpxchg*-style primitives.
3. FIFO wakeup, where fairness is guaranteed to anyone waiting. Finally, it is worthwhile pointing out that the kernel implementation requires exactly the same number of lines as the previous implementation.

FUTEX_FD: It is used to support asynchronous wakeups. This operation associates a file descriptor with a futex. If another process executes a *FUTEX_WAKE*, the process will receive the signal number that was passed in *val*. The calling process must close the returned file descriptor after use. To prevent race conditions, the caller should test if the futex has been upped after *FUTEX_FD* returns. It returns the new file descriptor associated with the futex.

FUTURE DIRECTIONS

- A lot of time is being wasted between a process releasing the lock and another process acquiring the lock. Inorder to avoid that, an asynchronous wait extension will be added to consume less time.

- Currently, in the uncontended case, only the system calls are avoided. It would be much more good if the kernel interactions are also removed. For this purpose, they are working with the NGPT team to build Global POSIX mutexes over futex. NGPT supports a M : N threading model, i.e., M user level threads are executed over N tasks. Conceptually, the N tasks provide virtual processors on which the M user threads are executing. When a user level thread, executing on one of these N tasks, needs to block on a futex, it should not block the task, as this task provides the virtual processing. Instead only the user thread should be descheduled by the thread manager of the NGPT system.

Nevertheless, a waitobj must be attached to the waitqueue in the kernel, indicating that a user thread is waiting on a particular futex and that the task group needs a notification with respect to the continuation on that futex. Once the thread manager receives the notification it can reschedule the previously blocked user thread. For this we provide an additional operator *AFUTEX_WAIT* to the sys_futex system call. Its task is to append a waitobj to the futex's kernel waitqueue and continue. This waitobj cannot be allocated on the stack and must be allocated and de-allocated dynamically. Dynamic allocations have the disadvantage that the waitobjs must be freed even during an irregular program exit. It further poses a denial of service attack threat in that a malicious applications can continuously call *sys_futex (AFUTEX_WAIT).*

The general solutions seem to convert to the usage of a */dev/futex* device to control resource consumption. The first solution is to allocate a file descriptor *fd* from the */dev/futex* "device" for each outstanding asynchronous waitobj. Conveniently these descriptors should be "pooled" to avoid the constant opening and closing of the device. The private data of the file would simply be the waitobj. Upon completion a SIGIO is sent to the application. The advantage of this approach is that the denial of service attack is naturally limited to the file limits imposed on a process. Furthermore, on program death, all waitobjs still enqueued can be easily dequeued. The disadvantage is that this approach can significantly pollute the "*fd*' space. Another solution pro-posed has been to open only one *f*d, but allow multiple waitobj allocations for this *f*d. This approach removes the *fd* space pollution issue but requires an additional tuning parameter for how many outstanding waitobjs should be allowed per fd. It also requires proper resource management of the waitobjs in the kernel. At this point no definite decisions has been reached on which direction to proceed. The question of priorities in futexes has been raised: the current implementation is strictly FIFO order. The use of nice level is almost certainly too restrictive, so some other priority method would be required. Expanding the sys-tem call to add a priority argument is possible if there were demonstrated application advantage.

CONCLUSION

In this paper I've described a fast userlevel locking mechanism, called *futexes*, that were integrated into the Linux 2.5 development kernel and also in the kernel version 2.6 of Linux.

Also the various requirements for such a package,previous various solutions and the current futex package were outlined. In the performance futexes can provide significant performance advantages over standard System V IPC semaphores in all case studies.

REFERENCES

- www.google.com
- www.wikipedia.com
- www.studymafia.org