ISSN 0361-7688, Programming and Computer Software, 2011, Vol. 37, No. 4, pp. 223–227. © Pleiades Publishing, Ltd., 2011.
Original Russian Text © A.V. Fedorov, 2011, published in Programmirovanie, 2011, Vol. 37, No. 4.

Optimization of the NPTL Thread Library in Linux for Hard Real-Time Systems

A. V. Fedorov

Graduated from the Moscow Institute of Physics and Technology
e-mail: Alexander.V.Fedorov@mcst.ru, alexvf@bk.ru

Received January 20, 2011

Abstract—The Native POSIX Thread Library (NPTL) fully supports the Portable Operating System Interface for Unix (POSIX) threads, which is one of the most popular interfaces for multithreading applications. This library has been gradually improved to meet the requirements of real-time systems but currently fits only soft real-time systems. This paper explains why the library fails to meet the requirements of hard real-time systems and tries to eliminate the reasons of it.

DOI: 10.1134/S0361768811040025

1. INTRODUCTION

Currently, the department of operational systems of the MCST closed joint-stock company is engaged in a number of projects aimed at adapting the Linux operating system to be used in Elbrus computer systems operating in the hard real-time mode. In this context, an implementation of threads in the POSIX (Portable Operating System Interface for Unix) user interface adapted to real time is of great importance. All primitives of this interface can be divided into two groups: the first is responsible for creating and destroying the threads and the second is responsible for their synchronous execution. This paper considers only the second group, the efficiency of which is much more valuable in hard real-time systems, because all the initialization can be performed at the system start, whereas there is no way to avoid synchronization of threads in parallel computations.

In the course or this study, the author analyzed the NPTL (Native POSIX Thread Library) to check whether it is suitable for hard real-time systems. This analysis constitutes the main content of the paper and is of most interest for the reader. First, the sys_futex system call is reviewed; next, the implementation of synchronization objects is detailed; and, finally, the optimization efficiency is measured.

2. SPECIFIC FEATURES OF IMPLEMENTATION OF SYNCHRONIZATION OBJECTS IN NPTL

The NPTL was developed specially for Linux, which makes it possible to make library-supported changes in the kernel. Thus, the *sys_clone* system call, which can be found only in Linux, was modified so to be able to create and destroy threads, and a system call *sys_futex* supporting an object of the same name [1] was developed with the aim of efficient implementation of any synchronization objects (e.g., semaphores or condition variables).

Of course, the simplest solution would be to implement these objects fully in the address space of the kernel and make a system call that would simply redirect application requests to the kernel of the operating system. Even more, one could use the locks and semaphores that have already been available in the kernel: it would suffice merely to match each user object with a “real” object in the kernel. The drawback of such a solution is that each operation will use a system call, and this is an expensive pleasure (loss of several hundreds of cycles and erasing the contents of the processor cache).

The designers of NPTL have found a simple solution for this problem. The role of any synchronization object is to delay execution of one thread until some event occurs. However, if there is no need in delaying an execution at some moment, then there is no need for the system call as well. In practice, such optimization is implemented through storing the state of the object in a variable that is located in the user space and is its part; then, the state can be modified without system calls.

It is this variable that is called a futex. The application checks the state of the object by calculating the value of the futex and, if there is no need for blocking (for example, when closing an opened semaphore) or waking waiting threads (for example, when opening a semaphore that was closed once), simply writes the new state into the futex. To avoid race states, reading and writing is performed by a single atomic instruction. The system call *sys_futex* is called either for waking a waiting thread (in this case, the thread is deleted from the queue in the kernel and marked as ready for execution) or when blocking is needed (in this case, the thread is added to the queue and, immediately before adding, the futex value is checked again to avoid the case when it has managed to change and the blocking is not required any more). Futexes are used in the NPTL to implement mutexes and semaphores, based on which all other objects are implemented.

It is due to this optimization that the term “futex” (Fast Userspace MUTex) involves the word “Fast”.

3. UNIVERSALITY OF FUTEXES

Hard real-time systems should guarantee that operations are completed in certain time. Along with this, one should improve the following main parameters:

(1) worst-case time of operation execution;

(2) average time of operation execution;

(3) delays between an external event and response to it (for the operating system, this delay directly depends on the length of critical sections protected by spinlocks).

Hereafter, by operations, we will mean functions working with synchronization objects: mutexes, barriers, and condition variables. The standard defines also read–write locks; however, in real-time systems, they are almost not used, since it is extremely hard to deal with the priority inversion (when a thread with a priority higher than that of the owner but lower than that of the waiter occupies the processor and prevents the owner from freeing the lock, thus blocking the waiter too [2]).

In the library, POSIX threads were fully implemented and have been rewritten over many years to allow for the requirements of real-time systems. However, these changes have always been hindered by frictions between the users who do not need real-time support and the users who do. Since the first group is larger, the incorporation of changes faces obstacles. For this reason, the library has bottlenecks; however, even if they are removed, a more global problem—universality of futex, which underlies all synchronization—will still remain.

But how does this universality turn out to be a shortcoming? It is very difficult to implement barriers and condition variables only on futexes; therefore, they were implemented on the basis of mutexes, which are easier to be implemented through futexes (after all, it is not without reason that the term “futex” means “fast mutex”). As a result, all objects turn out to be based only on futexes, but this hierarchy (contrary to the direct implementation on top of kernel spinlocks) leads to certain problems in hard real-time systems.

(1) The internal mutexes protecting the object state must follow the priority-inversion protocol, which is not done yet.

(2) In some cases, delays and overheads are increased. For example, while the “direct” implementation requires a single system call, in the current version of condition variables, the thread, in the worst case, will make system calls at the beginning and, the end of two critical sections plus one call for the operation execution itself, i.e., altogether, five calls; moreover, even if the internal mutex of an object turns out to be free, its acquisition and freeing are not free of charge, which adds four extra atomic instructions.

(3) The problem mentioned in the previous item is complicated by nuances of the historical evolution of futexes and the Linux kernel. The internal objects of the kernel have always been protected by spinlocks, which cannot be used in real-time systems: to get deterministic delays with spinlocks, the length of critical sections protected by them must be limited because the process cannot be preempted within these sections. The kernel developers not always took care about this, and, in many cases, the length of these sections is clearly unlimited. Therefore, Ingo Molnar’s patch for real-time systems has replaced many spinlocks with mutexes [3]. As a result, the barriers and condition variables in the NPTL were implemented on top of user mutexes based on futexes, which were implemented on top of kernel locks, which, in turn, were implemented on top of kernel spinlocks. Clearly, this lengthy hierarchy has a negative effect on both the average and worst-case time of operation execution.

(4) Some optimizations become impossible (the details can be found in the description of individual objects).

In addition, although the call sys_futex was designed as a versatile tool, it is impossible to implement a part of its functionality needed for some objects (for example, the robust attribute or the priority inheritance protocol) only at the user level. As a result, the use of futexes has become more and more complicated with time due to an additional code designed for solving specific problems. The concept of futex has gone much away from the original idea of a simple synchronization object described by a few rules, and the call sys_futex currently looks much more complicated than when it first appeared.

Finally, we arrive at the following two problems: implementation of all objects through futexes and the fact that the library was not oriented to real-time systems from the outset. The most substantial aspects of the second problem are considered in the next section.

4. HOW NPTL MEETS THE REQUIREMENTS OF REAL-TIME SYSTEMS

Initialization of Objects

To each synchronization object, a queue of waiting threads in the kernel is assigned. The NPTL allocates and frees memory for this queue automatically on as-needed basis; although a small cache is used, this provides no guarantee that there will be no memory allocations, which can take indefinitely long time. This approach simplifies the use of futexes: they do no need to be initialized, and the risk of memory leak is eliminated. This even makes it possible to completely avoid the use of destructors for objects.

However, due to uncertainty of time required for memory allocation, the initialization in hard real-time systems should be performed beforehand (i.e., first, to create all the threads and objects and run the system only when everything is ready). At the same time, the dynamic allocation and freeing of memory for a queue contradicts this approach and affects negatively the worst-case and average time of operation execution.

Another reason for using static initialization of queues is the speed-up of the operation of objects shared between processes. When the program is executed, one should be able to differentiate which queue in the kernel of the operating system corresponds to certain synchronization object in the user program. To this end, it is necessary to associate each object with a unique number, which will serve as an index of the array of queues or as a key in the hash table of queues. The static initialization used in real-time systems makes it possible to calculate that number once upon the initialization and store it within an object in the user program. However, during the NPTL operation, different queues can be used at different times for one and the same object; therefore, one has to calculate the key each time when the thread is blocked on the object, which increases the worst time of execution (the calculation of the key requires that the semaphore in the kernel be closed for reading).

Mutexes

Normally, real-time systems use mutexes with the priority inheritance protocol, according to which, the thread waiting for unlocking must increase the priority of the owner up to its priority. Due to this, it becomes impossible for the system to have running threads with a priority higher than that of the owner and lower than that of the waiting thread. A specific feature of these mutexes is the necessity to know which thread owns the mutex in order to increase its priority. Since one of the basic ways to speed up the operation of mutexes is to minimize the number of expensive (compared to atomic operations) system calls, it is required to find a way for locating the thread identifier without going into the kernel. This method is called Thread Local Storage, and the NPTL uses it to implement such mutexes. The inheritance algorithm used is the same as that for kernel mutexes; it is derived automatically due to the fact that the user mutexes with priority inheritance, *sys_futex* serves merely as an additional layer between them and kernel mutexes. However, because the kernel mutexes do not support the full functionality of user mutexes (for example, the priority protection protocol and the robust attribute), the use of their code instead of a special-purpose one will not allow the user mutexes to be implemented optimally.

Along with the priority inheritance protocol, the hard real-time systems include also the priority protection protocol. The latter yields more predictable execution times, because the priority inheritance may lead to the emergence of chains of threads waiting for unlocking of mutexes belonging to threads that also wait for unlocking of some other mutexes and so on. These chains may be very lengthy (or, even generate a loop). With the priority protection, to each mutex, a peak priority is assigned, which must not be less than the peak priority of the threads that use this mutex. In each acquiring, the thread temporarily increases its priority up to a given value; in freeing, its value decreases back (it can be seen from here that both the acquiring and freeing of such mutexes is always implemented by a system call, because the priority can be changed only by the kernel). In the NPTL, this was performed by two additional system calls increasing the priority during acquisition and decreasing it upon freeing. It is clear that this has an adverse effect on the performance and that it is much more efficient to support the priority protection in the kernel.

Another key aspect is the robust attribute. The list of mutexes owned by the thread that is needed for the implementation of this attribute can be allocated in either the kernel or the user space. The problem is what should be done when the mutex owner dies. If there is a list of acquired mutexes, it is possible to iterate through the list and free all mutexes, thus making impossible the appearance of "bad" queues that refer to the non-existing owner. This cannot be done in the NPTL, because the list is located in the user space [4] and, consequently, is unreliable. Therefore, each queue has a usage counter, which is increased by one when the corresponding mutex is acquired and decreased when it is freed (the same counter is also necessary for the dynamic creation and removal of queues: when it reaches zero, the queue can be removed). These two atomic additions are not the fastest instructions (almost a hundred cycles each on x86); therefore, it makes sense to put the list to the kernel to avoid this overhead. Then, it will not be possible to implement the robust attribute for mutexes without protection from priority inversion; however, this is still beneficial for hard real-time systems: such mutexes are not crucial for them.

Barriers

The barriers are synchronization objects that block the execution of threads calling function pthread_barrier_wait() until a preassigned number of threads arrives.

As already mentioned, the barriers in the NPTL are implemented on top of user mutexes that are implemented on top of futexes, and so on. Given that the direct implementation through kernel spinlocks is simple and, at the same time, efficient, it is not quite clear why the NPTL developers wanted to turn futexes into something with a finger in every pie and build all synchronization objects exclusively on them. The only exception is the straight-made priority inheritance mutexes, and this is merely because it is impossible to implement inheritance at the application level without a profound assistance from the operating system.

The only small problem arising in real-time systems with a direct implementation on top of spinlocks is caused by the fact that the number of threads in a queue is unlimited, and, consequently, the bypass of the entire queue can last endlessly. Since a thread within a critical section cannot be preempted, one has to limit the length of critical sections to reduce the delays. Therefore, it is required to wake the threads waiting on a barrier by small groups rather than all at once and, in the intervals between waking these groups, to free the spinlock and open the interrupts. To avoid the appearance of new threads at that time in the queue (which has already been filled once), the queue is detached from the barrier, which gets a new empty queue.

This may result in a situation when the group with a high-priority thread has not yet been woken, while the waking thread has a low priority and was preempted (i.e., we have an inversion of priorities). To avoid this situation, it will suffice to increase the priority of the waking thread in the interval between waking two groups up to the level of the first thread in the queue.

It remains only to notice that the current implementation of the barriers is subjected to inversion, which can be eliminated by inheritance of priorities in the user mutex that was used to implement the barrier. However, in the nearest future, this feature hardly will be added to the NPTL because only a small number of users need it.

Condition Variables

All that has been said about the barriers is fully related to condition variables: their queues are also not limited in size, their processing in parts leads to priority inversion, and the internal mutex does not currently utilize inheritance. Yet, for them, at least there is a suitable patch allowing for inheritance (published by Darren Hart as early as in May [5]).

The specific feature of condition variables is that only one thread can acquire the mutex as a result of the execution of function pthread_cond_broadcast, and there is no need even to wake the remaining ones. Nevertheless, the original implementation simply woke all threads independent of their quantity, and a small “scrap” between them yields a single lock-capturing thread for execution (this is often referred to as a “thundering herd problem”). Then, operation futex_requeue was added, which woke only one thread and shifted the others. Yet, there still remains a place for optimization here: if the mutex is occupied at the time when pthread_cond_broadcast is called, no thread needs to be woken. The optimization was implemented for mutexes that use priority inheritance but was not implemented for the others, which is explained by the universality of futexes.

It should not be expected that the situation with an acquired mutex is unlikely to happen. The following code, which involves no acquisition, is quite allowable:

```
pthread_mutex_lock(mx);
condition=true;
pthread_mutex_unlock(mx);
pthread_cond_broadcast(cv).
```

However, quite often, the following implementation also can be met:

```
pthread_mutex_lock(mx);
condition=true;
pthread_cond_broadcast(cv);
pthread_mutex_unlock(mx).
```

Here, the lock belongs to the thread calling pthread_cond_broadcast, and, therefore, there is no need to wake any thread in the queue for all cases.

Another bottleneck is the use of optimization for objects shared between processes. Because, in the NPTL, the queue of an object is closely related to the futex, two futexes must be passed to operation *requeue* as parameters: one for a condition variable and one for a mutex. However, *pthread_cond_broadcast* gets only a condition variable as a parameter; i.e., we should find the corresponding mutex by ourselves. In the case of private objects, this is achieved by storing in the condition variable a pointer to the mutex currently associated with it. However, the shared objects may have different addresses in different processes, and, in the user space, there is no way to rapidly find a pointer to the associated mutex within function *pthread_cond_broadcast*. However, this can be done in the direct (without futexes) implementation.

5. EXPERIMENTAL ASSESSMENT

Based of the above discussion, let us list what namely can be optimized in the NPTL from the point of view of hard real-time systems:

(1) The use of static initialization makes it possible to speed up the shared objects, as well as improve the maximum time of execution.

(2) The condition variables and barriers should be implemented in the kernel rather than on top of user mutexes.

(3) Use a separate code for user mutexes.

(4) If item (3) is fulfilled, it is possible to optimize performance of condition variables and locks shared between processes.

(5) Place the list of acquired locks in the kernel (which, together with item (3) will make it possible to speed up priority inheritance mutexes).

In other words, it is required to almost completely rewrite the implementation of synchronization objects in the NPTL (both the user and kernel parts). The author carried out this in the library named *elpthread*, which made it possible to compare the performance of mutexes and condition variables in the old and new versions (the principles of implementation of barriers are similar to those for condition variables; therefore, the difference in the performance will be the same).

The calculations were conducted on a computer with a Core Quad Q8400 @ 3.06 GHz processor, Ubuntu 10.04 operating system, NPTL version 11.1, and kernel 2.6.33. In all tests, except *cond-perf*, the swapping was switched off with the help of *mlockall*, and all the interrupts except, the 0th and 2nd ones, were switched off in the kernels where the testing was performed.

```
ptsematest -a -t -p99 -i100 -d25 l1000000
```

Test ptsematest was taken from the package rt-tests. The minimum/average/maximum results constituted 1, 2, and 19 mcs, respectively, for NPTL and 1, 2, and 16 mcs for elpthread.

To test the performance, we run on different kernels two threads that acquired and freed one lock per cycle. The first thread had priority 98, and the second thread had priority 99 and measured the total time of acquiring and freeing (altogether, 10000000 measurements). The priority of the first thread was decreased to avoid the so-called lock stealing effect. The time constituted from 176 to 137224 processor cycles for NPTL and from 192 to 49816 cycles for elpthread.

To assess the effect of optimization of the condition variables, the following test was used. On one kernel, we ran two threads with priority 90, which executed in a loop functions *pthread_cond_broadcast* and *pthread_cond_wait* to measure the time from the entry into the function *broadcast* of the first thread to the exit from *wait* of the second function. The minimum/average/maximum times constituted 4672, 4965, and 30904 cycles, respectively, for NPTL and 4016, 4255, and 25504 cycles for elpthread.

To assess the overall performance, the *cond-perf* test (of the NPTL library) was used.

```
$ time ./cond-perf -r1000000
real 0m30.887s
user 0m5.383s
sys 1m19.538s
$ time ./cond-perf -r1000000 -k
real 0m20.649s
user 0m2.467s
sys 0m19.972s
$ export LD_PRELOAD=libelpthread.so.1
$ time ./cond-perf-r1000000
real 0m16.705s
user 0m1.847s
sys 0m15.076s
$ time ./cond-perf -r1000000 -k
real 0m16.150s
user 0m1.887s
sys 0m14.469s
```

It can be seen that the worst times were mainly improved. This could be expected because this was the aim of the optimizations performed. However, for the condition variables, considerable improvement was achieved in the average times too, mainly due to the use of a more specialized code in the Linux kernel.

REFERENCES

1. http://people.redhat.com/drepper/nptl-design.pdf.
2. http://lwn.net/Articles/146861.
3. http://lwn.net/Articles/345076.
4. http://www.mjmwired.net/kernel/Documentation/robust-futexes.txt.
5. http://www.cygwin.com/ml/libc-alpha/2010-05/msg00054.html.