Mutex mutandis: understanding mutex types and attributes

Mon, 07/06/2009 - 09:42 — csimmonds

The mutex is a simple little thing with one job to do: ensure mutual exclusion between threads. Yet Linux has up to 16 variations of the mutex, depending on which versions of the kernel and C library you have (see the table at the end). In this article I will try to explain why there are so many and how they affect your programs.

Basics

A mutex has two states: locked and unlocked. Once it has been locked by one thread any others are forced to wait until the first unlocks it again. Only the thread that locked the mutex may unlock it. With these three facts you can use a mutex to ensure exclusive access to a shared resource, such as a data structure. In the example below I have included the mutex in the structure being protected, which is good design practice

```
struct some_data {
pthread_mutex_t data_lock;
// other data items
} my_data;

...

pthread_mutex_lock (&my_data.data_lock);
// write some values
pthread_mutex_unlock (&my_data.data_lock);
```

Only one thread can hold data_lock at a time, ensuring that the data values are always consistent.

Complications

Mutexes are so important to the correct behavior of an application that small details about their implementation can make a big difference overall. Here are some things to consider

- what is most important: speed or correct behavior?
- what happens if you try to lock the same mutex twice?
- if several threads are waiting for a mutex to be unlocked, which one should get the mutex next?
- is it acceptable for a high priority thread to be blocked indefinitely by a lower priority thread (leading to priority inversion)?
- what happens if the thread that has locked the mutex terminates without unlocking?

Your response to these questions will determine the sort of mutex you need.

Types of mutex: fast, error checking, recursive and adaptive

Linux has four types of mutex. The code snippets below show how to declare and initialise the default type, which is **fast**. Sounds good, but what does that mean? It means that speed is preferred over correctness: there is no check that you are the owner in pthread_mutex_unlock() so any thread can unlock a fast mutex. Also it doesn't check if you have already locked the mutex, so you can deadlock yourself, and there are no checks anywhere that the mutex has been intialised correctly.

You declare a mutex of default (fast) type at run-time like this

```
pthread_mutex_t mutex;
...
```

```
pthread_mutex_init (&mutex, NULL);
```

or statically at compile-time like this

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

If you prefer correctness over speed, you need to set the type to be **error checking**. Error checking mutexes return EDEADLK if you try to lock the same one twice and EPERM if you unlock a mutex that isn't yours. To create such a mutex you need to initialise a *muter_attr* and pass it to pthread_mutex_init() like so:

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_settype (&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init (&mutex, &attr);
```

or, you can do it statically at compile-time in one line like this:

```
pthread_mutex_t mutex = PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

Note: the _NP suffix indicates a non-portable extension to the POSIX specification. In fact the latest specification, 1003.1-2008 [2], includes most of the Linux additions so you can leave the _NP ending off. I have chosen not to because some older versions of the header files only have the _NP variants.

Next is the **recursive** mutex, which does everything that the error checking mutex does except that you can lock the same mutex multiple times. It keeps a count of the number of times it has been locked and you must unlock it the same number of times before it becomes truly unlocked. As with the other types, you can declare and initialise one like this:

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_settype (&attr, PTHREAD_MUTEX_RECURSIVE_NP);
pthread_mutex_init (&mutex, &attr);
```

or like this:

```
pthread_mutex_t mutex = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
```

The last type is **adaptive** which is an extra fast mutex for multi processor systems. It combines a spinlock with an ordinary mutex: instead of blocking straight away on a locked mutex it spins for a short while re-trying the lock and then blocks in the normal way. On a single processor it doesn't spin and so is identical to a *fast* mutex.

Handling errors

There is not much point using error checking mutexes if you ignore the return value! Except that in reality the main reason that you will get errors from pthread_mutex_lock() and pthread_mutex_unlock() is because of *logic* errors in your code. I use this macro to detect errors during development and then compile it out for production.

```
#ifdef DEBUG
#define pthread_mutex_lock_check(mutex)         \
({                                              \
        int __ret = pthread_mutex_lock (mutex); \
        if (__ret != 0)                         \
                printf ("pthread_mutex_lock_check in %s line %u: error %d - %s\n", \
```

```
                        __FILE__, __LINE__, __ret, strerror (__ret)); \
       __ret;                           \
})
#else
#define pthread_mutex_lock_check pthread_mutex_lock
#endif
```

Wake-up order

when a mutex is unlocked and there are several threads blocked waiting for it the system has to decide which gets the mutex next. Until recently the choice was simply the thread that had been waiting longest, but with Linux kernels from 2.6.22 onwards the thread chosen will be the highest priority real-time thread. If there are no real-time threads then it will be the longest waiter as before.

Sharing a mutex between processes

Most often the shared resource being protected by a mutex is a global variable in a process address space and so the threads using the mutex are all local to that process. Sometimes you will have data in a shared memory segment, for example using the POSIX or SYSV IPC shared memory functions, and so the mutex needs to be locked and unlocked by threads from *different* processes. In such a case, the mutex must be initialsed with the shared attribute

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setpshared (&attr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init (&mutex, &attr);
```

Apart from being shared the behavior of the mutex is the same as local ones.

Problems with real-time threads: priority inversion

If (and only if) you have threads with real-time scheduling policies SCHED_FIFO or SCHED_RR you may experience *priority inversion*[3] which results in a high priority thread that is waiting to lock a mutex being blocked by a lower priority thread. One way to resolve the problem is to set the priority protocol of the mutex to *priority inheritance*, with the result that the thread holding the mutex inherits the priority of the highest priority thread waiting for the mutex and so it cannot be preempted by intermediate priority threads. Here is how to do it

```
#define __USE_UNIX98 /* Needed for PTHREAD_PRIO_INHERIT */
#include <pthread.h>

pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setprotocol (&attr, PTHREAD_PRIO_INHERIT);
pthread_mutex_init (&mutex, &attr);
```

Priority inheritance can be combined with any of the four types. However, it adds a large overhead to the implementation and so it does not make sense to combine it with the *fast* or *adaptive* types.

Unexpected termination: the robust mutex

Supposing a thread has locked a mutex and then terminates, what then? In the normal run of things the mutex will remain locked for ever (OK, until the next reboot), causing any threads trying to lock it to deadlock. This is particularly a problem if you are sharing a mutex between processes and one of them segfaults or is killed.

This is where the **robust** attribute comes in. The first stage is to initialise the mutex with the robust option. It can be combined with any of the four types and with the priority inheritance attribute. Here we go:

```
#define __USE_GNU /* Needed for PTHREAD_MUTEX_ROBUST_NP */
#include <pthread.h>

pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init (&attr);
pthread_mutexattr_setrobust_np (&attr, PTHREAD_MUTEX_ROBUST_NP);
pthread_mutex_init (&mutex, &attr);
```

Now, if the thread owning the mutex terminates with it locked, any other thread that is trying to lock it will unblock with error code EOWNERDEAD. In other words, this mutex no longer functions as a mutex. If you want to repair the situation you must validate the data that the mutex was protecting, maybe remove some inconsistent state, and then call pthread_mutex_consistent_np(). Then you must lock it, in the same thread that marked it as consistent. Now it is a fully functional mutex again.

Finally, if the mutex is unlocked without being made consistent, it will be in a permanently unusable state and all attempts to lockit will fail with the error ENOTRECOVERABLE. You have blown it: the only thing you can do with such a mutex is to destroy it.

All the above adds quite a lot of complexity to the implementation of a mutex, so robust mutexes are NOT going to be fast.

Summary

We have four types of mutex each of which may be robust and may have the priority inheritance protocol, which gives us 4 x 2 x 2 = 16 different possibilities. Here are my suggestions on which to use.

- During development, use error checking - the extra overhead is very small
- Use recursive mutexes if you have library code where you cannot be sure that a mutex has already been locked elsewhere
- In high performance production code, use fast or (if you have more than one CPU) adaptive types
- If you have real time threads, look carefully at the dependencies between the threads and use priority inheritance where necessary
- If you share mutexes between processes (or if your threads terminate in odd places) use robust mutexes

Features vs libc version

Some features are dependent on the version of the 'C' library you use. Here I am considering the two most often used in embedded devices: GNU libc (glibc) and the microcontroller C library, uClibc [4].

- Fast, error checking and recursive mutex types are present in all versions of glibc and uClibc
- The adaptive type and the robust and priority inheritance protocol are ONLY implemented in glibc 2.5 onwards
- Main-line uClibc DOES NOT implement adaptive, robust and priority inheritance (*)

(*) Because uClibc uses the older LinuxThreads library, not the Native POSIX Threads (nptl) library as glibc does. There is an on-going project to port nptl to uClibc but it is not yet functional on all architectures.

Features vs kernel version

If you are using glibc (or uClibc with nptl) you also need to keep a watch on the kernel version. Here are the minimum versions for the various features

Normal	Robust	Priority Inheritance	Priority wake-up queue
2.6.0	2.6.17	2.6.18	2.6.22

References

[1] The title *mutex mutandis* is a really bad pun on the phrase *mutatis mutandis*, meaning "the necessary changes having been made", see http://en.wikipedia.org/wiki/Mutatis_mutandis
[2] IEEE Std 1003.1-2008 System Interfaces http://www.opengroup.org/onlinepubs/9699919799/functions/contents.html
[3] For a description of priority inversion see http://en.wikipedia.org/wiki/Priority_inversion
[4] uClibc can be found at http://www.uclibc.org/

Chris Simmonds
chris@2net.co.uk

Comments

Comment viewing options

[redacted] [redacted] [redacted] Save settings

Select your preferred way to display the comments and click "Save settings" to activate your changes.

It is very useful, it seems

Tue, 12/18/2012 - 06:49 — Victor (not verified)

It is very useful, it seems the Linux does not support the robust for the read-write lock.

reply

Reply to comment | embedded-linux.co.uk

Thu, 07/11/2013 - 03:47 — ouka.mobi (not verified)

“Reply to comment | embedded-linux.co.uk” certainly got me personally simply addicted with your blog! I reallywill certainly be back again far more frequently. Thank you ,Chelsey

reply

Reply to comment | embedded-linux.co.uk

Thu, 08/01/2013 - 21:58 — Penni (not verified)

Where did u obtain the points to post ““Reply to comment | embedded-linux.

co.uk”? Thanks ,Patti

reply

Reply to comment | embedded-linux.co.uk

Wed, 07/31/2013 - 19:17 — Velda (not verified)

“Reply to comment | embedded-linux.co.uk” in reality got me simply addicted on ur blog! I personallydefinitely will wind up being returning far more frequently.
Many thanks -Zachary

reply

Reply to comment | embedded-linux.co.uk

Tue, 07/30/2013 - 22:49 — Twila (not verified)

Precisely how long did it acquire u to create “Reply to comment | embedded-linux.
co.uk”? It boasts a lot of wonderful material.
Thx -Shanon

reply

Reply to comment | embedded-linux.co.uk

Mon, 08/12/2013 - 21:20 — Jeanette (not verified)

I personally had to share this specific posting, “Reply to comment | embedded-linux.co.uk” with my best

good friends on facebook. I actuallybasically wished to disperse
your fantastic posting! Thx, Stephania

reply

Shared mutex : has to passed to users?

Thu, 08/26/2010 - 22:38 — IanVaughan

When creating a shared mutex via PTHREAD_PROCESS_SHARED
Does the "pthread_mutex_t mutex" instance/ID/handle returned from pthread_mutex_init have to sent to all the sharing processes?
i.e. I have two separate programs, when started they commutate via TCP messages, but they are in there own process space.
If I want both to lock/unlock on the same mutex, do I have to create the mutex as above in one program, and send the handle/ID to the other?

reply

Re: Shared mutex : has to be passed to users?

Sat, 09/04/2010 - 21:42 — csimmonds

Yes, all processes wishing to lock the mutex must have a reference to the pthread_mutex_t that defines it. In practice, the shared mutex is always defined in a shared memory segment along with the data structures it is protecting, so the mechanism for sharing the mutex descriptor is there all along.

reply

Reply to comment | embedded-linux.co.uk

Mon, 08/12/2013 - 21:51 — Lea (not verified)

“Reply to comment | embedded-linux.co.uk” was indeed a fantastic blog post and therefore I
really was pretty content to locate the article. Regards-Evelyn

reply

Reply to comment | embedded-linux.co.uk

Fri, 08/30/2013 - 04:22 — cheats for Hill Climb Racing (not verified)

Bianca constitutes a phone call to Professor Juniper to
see her that her Minccino learned Thunderbolt, but Ash interrupts her.
This is surely an unwritten rule that few individuals understand or
accept. Unfortunately, Angel didn't end up about the same part in the island when you, which means you set out to seek out her.

reply

Reply to comment | embedded-linux.co.uk

Thu, 08/08/2013 - 18:19 — Julianne (not verified)

How did you actually pick up the suggestions to post ““Reply to comment |
embedded-linux.co.uk”? Many thanks -Kathrin

reply

Reply to comment | embedded-linux.co.uk

Sat, 07/27/2013 - 20:21 — Gisele (not verified)

I personally blog likewise and I’m posting something comparable to
this specific post, “Reply to comment | embedded-linux.

co.uk”. Would you care in case I actuallyutilize a lot of
of your suggestions? I appreciate it -Arlie

reply

Reply to comment | embedded-linux.co.uk

Mon, 08/12/2013 - 20:56 — Nicki (not verified)

This unique blog, “Reply to comment | embedded-linux.
co.uk” ended up being superb. I’m printing out a copy to present my good
friends. Thanks for the post-Monica

reply

very useful

Mon, 06/28/2010 - 23:51 — maimutoi

It has saved a lot of my time. Thanks

reply

Great article

Wed, 09/09/2009 - 10:16 — feabhas

Great explanation of the Linux mutex, just what was needed. I've heard the term "futex" used to refer to the "fast mutex". Is this common term in the Embedded Linux community?
Can a fast mutex have priority inheritance? This seems counter intuitive if no check is made who is the mutex owner?

reply

Futex

Wed, 09/09/2009 - 21:06 — csimmonds

The futex (fast user-space mutex) is the low level primitive used to implement the types of mutex I describe in the article, but unless you are writing a replacement POSIX thread library you will never see a futex in the flesh. So, to answer the question directly: yes futexes can have priority inheritance and ownership, depending on the attributes of the upper level mutex.

reply

Priority ceiling

Wed, 09/02/2009 - 16:19 — wibble

You mention the priority inheritance protocol. Does Linux have the priority ceiling protocol as well?

reply

Reply to comment | embedded-linux.co.uk

Mon, 07/29/2013 - 17:22 — Rachele (not verified)

Precisely how long did it acquire u to write “Reply to comment | embedded-linux.
co.uk”? It possesses an awful lot of very good advice.
With thanks ,August

reply

Priority ceiling

Wed, 09/09/2009 - 21:54 — csimmonds

Yes, that is a little detail I missed out of the article. The function pthread_mutexattr_setprotocol can take three values
PTHREAD_PRIO_NONE - no priority protocol (the default)
PTHREAD_PRIO_INHERIT - priority inheritance, as described above
PTHREAD_PRIO_PROTECT - priority ceiling

After selecting PTHREAD_PRIO_PROTECT you need to give the mutex a priority in the range 1 to 99 with

pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);

Now, when a thread blocks on the mutex, its priority will be the higher of the thread's priority and the ceiling priority of the mutex. To be useful, the ceiling priority must be higher than or equal to the highest priority of any thread that may try to lock it, thus avoiding

priority inversion. As you can see, the priority ceiling protocol puts a bit more responsibility on the programmer than priority inheritance.

Note: including this option, we have 24 types of mutex!

reply

Post new comment

Your name: *

Visitor

E-mail: *

The content of this field is kept private and will not be shown publicly.

Homepage:

Subject:

Comment: *

Web page addresses and e-mail addresses turn into links automatically.

Allowed HTML tags: <a>   <cite> <code>   <dl> <dt> <dd>

Lines and paragraphs break automatically.

More information about formatting options

By submitting this form, you accept the Mollom privacy policy.

Preview