#
Copyright (c) 2006 Steven Rostedt
Licensed under the GNU Free Documentation License, Version 1.2
#

RT-mutex implementation design

This document tries to describe the design of the rtmutex.c implementation. It doesn't describe the reasons why rtmutex.c exists. For that please see Documentation/rt-mutex.txt. Although this document does explain problems that happen without this code, but that is in the concept to understand what the code actually is doing.

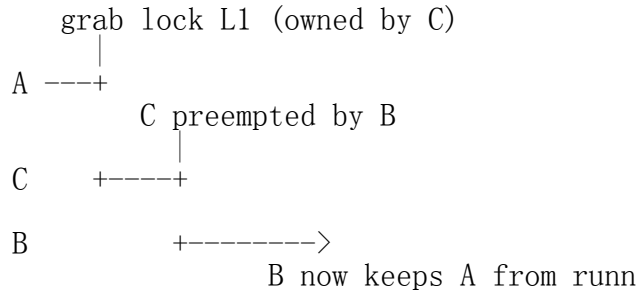
The goal of this document is to help others understand the priority inheritance (PI) algorithm that is used, as well as reasons for the decisions that were made to implement PI in the manner that was done.

Unbounded Priority Inversion

Priority inversion is when a lower priority process executes while a higher priority process wants to run. This happens for several reasons, and most of the time it can't be helped. Anytime a high priority process wants to use a resource that a lower priority process has (a mutex for example), the high priority process must wait until the lower priority process is done with the resource. This is a priority inversion. What we want to prevent is something called unbounded priority inversion. That is when the high priority process is prevented from running by a lower priority process for an undetermined amount of time.

The classic example of unbounded priority inversion is were you have three processes, let's call them processes A, B, and C, where A is the highest priority process, C is the lowest, and B is in between. A tries to grab a lock that C owns and must wait and lets C run to release the lock. But in the meantime, B executes, and since B is of a higher priority than C, it preempts C, but by doing so, it is in fact preempting A which is a higher priority process. Now there's no way of knowing how long A will be sleeping waiting for C to release the lock, because for all we know, B is a CPU hog and will never give C a chance to release the lock. This is called unbounded priority inversion.

Here's a little ASCII art to show the problem.



Priority Inheritance (PI)

There are several ways to solve this issue, but other ways are out of scope for this document. Here we only discuss PI.

PI is where a process inherits the priority of another process if the other process blocks on a lock owned by the current process. To make this easier to understand, let's use the previous example, with processes A, B, and C again.

This time, when A blocks on the lock owned by C, C would inherit the priority of A. So now if B becomes runnable, it would not preempt C, since C now has the high priority of A. As soon as C releases the lock, it loses its inherited priority, and A then can continue with the resource that C had.

Terminology

Here I explain some terminology that is used in this document to help describe the design that is used to implement PI.

PI chain - The PI chain is an ordered series of locks and processes that cause processes to inherit priorities from a previous process that is blocked on one of its locks. This is described in more detail later in this document.

mutex - In this document, to differentiate from locks that implement PI and spin locks that are used in the PI code, from now on the PI locks will be called a mutex.

lock - In this document from now on, I will use the term lock when referring to spin locks that are used to protect parts of the PI algorithm. These locks disable preemption for UP (when CONFIG_PREEMPT is enabled) and on SMP prevents multiple CPUs from entering critical sections simultaneously.

spin lock - Same as lock above.

waiter - A waiter is a struct that is stored on the stack of a blocked process. Since the scope of the waiter is within the code for a process being blocked on the mutex, it is fine to allocate the waiter on the process's stack (local variable). This structure holds a pointer to the task, as well as the mutex that the task is blocked on. It also has the plist node structures to place the task in the waiter_list of a mutex as well as the pi_list of a mutex owner task (described below).

waiter is sometimes used in reference to the task that is waiting on a mutex. This is the same as waiter->task.

waiters - A list of processes that are blocked on a mutex.

top waiter - The highest priority process waiting on a specific mutex.

top pi waiter - The highest priority process waiting on one of the mutexes that a specific process owns.

Note: task and process are used interchangeably in this document, mostly to differentiate between two processes that are being described together.

PI chain

The PI chain is a list of processes and mutexes that may cause priority inheritance to take place. Multiple chains may converge, but a chain would never diverge, since a process can't be blocked on more than one mutex at a time.

Example:

```
   Process:  A, B, C, D, E
   Mutexes:  L1, L2, L3, L4

   A owns: L1
           B blocked on L1
           B owns L2
                  C blocked on L2
                  C owns L3
                         D blocked on L3
                         D owns L4
                                E blocked on L4
```

The chain would be:

```
   E->L4->D->L3->C->L2->B->L1->A
```

To show where two chains merge, we could add another process F and another mutex L5 where B owns L5 and F is blocked on mutex L5.

The chain for F would be:

```
   F->L5->B->L1->A
```

Since a process may own more than one mutex, but never be blocked on more than one, the chains merge.

Here we show both chains:

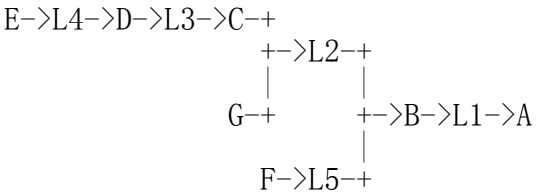
```
   E->L4->D->L3->C->L2-+
                       |
                       +->B->L1->A
                       |
                  F->L5-+
```

For PI to work, the processes at the right end of these chains (or we may also call it the Top of the chain) must be equal to or higher in priority than the processes to the left or below in the chain.

Also since a mutex may have more than one process blocked on it, we can have multiple chains merge at mutexes. If we add another process G that is blocked on mutex L2:

G->L2->B->L1->A

And once again, to show how this can grow I will show the merging chains again.



Plist

Before I go further and talk about how the PI chain is stored through lists on both mutexes and processes, I'll explain the plist. This is similar to the struct list_head functionality that is already in the kernel. The implementation of plist is out of scope for this document, but it is very important to understand what it does.

There are a few differences between plist and list, the most important one being that plist is a priority sorted linked list. This means that the priorities of the plist are sorted, such that it takes O(1) to retrieve the highest priority item in the list. Obviously this is useful to store processes based on their priorities.

Another difference, which is important for implementation, is that, unlike list, the head of the list is a different element than the nodes of a list. So the head of the list is declared as struct plist_head and nodes that will be added to the list are declared as struct plist_node.

Mutex Waiter List

Every mutex keeps track of all the waiters that are blocked on itself. The mutex has a plist to store these waiters by priority. This list is protected by a spin lock that is located in the struct of the mutex. This lock is called wait_lock. Since the modification of the waiter list is never done in interrupt context, the wait_lock can be taken without disabling interrupts.

Task PI List

To keep track of the PI chains, each process has its own PI list. This is a list of all top waiters of the mutexes that are owned by the process. Note that this list only holds the top waiters and not all waiters that are blocked on mutexes owned by the process.

The top of the task's PI list is always the highest priority task that is waiting on a mutex that is owned by the task. So if the task has inherited a priority, it will always be the priority of the task that is

at the top of this list.

This list is stored in the task structure of a process as a plist called pi_list. This list is protected by a spin lock also in the task structure, called pi_lock. This lock may also be taken in interrupt context, so when locking the pi_lock, interrupts must be disabled.

Depth of the PI Chain

The maximum depth of the PI chain is not dynamic, and could actually be defined. But is very complex to figure it out, since it depends on all the nesting of mutexes. Let's look at the example where we have 3 mutexes, L1, L2, and L3, and four separate functions func1, func2, func3 and func4. The following shows a locking order of L1->L2->L3, but may not actually be directly nested that way.

```
void func1(void)
{
        mutex_lock(L1);

        /* do anything */

        mutex_unlock(L1);
}

void func2(void)
{
        mutex_lock(L1);
        mutex_lock(L2);

        /* do something */

        mutex_unlock(L2);
        mutex_unlock(L1);
}

void func3(void)
{
        mutex_lock(L2);
        mutex_lock(L3);

        /* do something else */

        mutex_unlock(L3);
        mutex_unlock(L2);
}

void func4(void)
{
        mutex_lock(L3);

        /* do something again */

        mutex_unlock(L3);
```

```
}
```

Now we add 4 processes that run each of these functions separately.
Processes A, B, C, and D which run functions func1, func2, func3 and func4
respectively, and such that D runs first and A last. With D being preempted
in func4 in the "do something again" area, we have a locking that follows:

```
D owns L3
       C blocked on L3
       C owns L2
              B blocked on L2
              B owns L1
                     A blocked on L1
```

And thus we have the chain A->L1->B->L2->C->L3->D.

This gives us a PI depth of 4 (four processes), but looking at any of the
functions individually, it seems as though they only have at most a locking
depth of two. So, although the locking depth is defined at compile time,
it still is very difficult to find the possibilities of that depth.

Now since mutexes can be defined by user-land applications, we don't want a DOS
type of application that nests large amounts of mutexes to create a large
PI chain, and have the code holding spin locks while looking at a large
amount of data. So to prevent this, the implementation not only implements
a maximum lock depth, but also only holds at most two different locks at a
time, as it walks the PI chain. More about this below.

Mutex owner and flags

The mutex structure contains a pointer to the owner of the mutex. If the
mutex is not owned, this owner is set to NULL. Since all architectures
have the task structure on at least a four byte alignment (and if this is
not true, the rtmutex.c code will be broken!), this allows for the two
least significant bits to be used as flags. This part is also described
in Documentation/rt-mutex.txt, but will also be briefly described here.

Bit 0 is used as the "Pending Owner" flag. This is described later.
Bit 1 is used as the "Has Waiters" flags. This is also described later
  in more detail, but is set whenever there are waiters on a mutex.

cmpxchg Tricks

Some architectures implement an atomic cmpxchg (Compare and Exchange). This
is used (when applicable) to keep the fast path of grabbing and releasing
mutexes short.

cmpxchg is basically the following function performed atomically:

```
unsigned long _cmpxchg(unsigned long *A, unsigned long *B, unsigned long *C)
{
        unsigned long T = *A;
```

```
        if (*A == *B) {
                *A = *C;
        }
        return T;
}
#define cmpxchg(a,b,c) _cmpxchg(&a,&b,&c)
```

This is really nice to have, since it allows you to only update a variable if the variable is what you expect it to be. You know if it succeeded if the return value (the old value of A) is equal to B.

The macro rt_mutex_cmpxchg is used to try to lock and unlock mutexes. If the architecture does not support CMPXCHG, then this macro is simply set to fail every time. But if CMPXCHG is supported, then this will help out extremely to keep the fast path short.

The use of rt_mutex_cmpxchg with the flags in the owner field help optimize the system for architectures that support it. This will also be explained later in this document.

Priority adjustments

The implementation of the PI code in rtmutex.c has several places that a process must adjust its priority. With the help of the pi_list of a process this is rather easy to know what needs to be adjusted.

The functions implementing the task adjustments are rt_mutex_adjust_prio, __rt_mutex_adjust_prio (same as the former, but expects the task pi_lock to already be taken), rt_mutex_get_prio, and rt_mutex_setprio.

rt_mutex_getprio and rt_mutex_setprio are only used in __rt_mutex_adjust_prio.

rt_mutex_getprio returns the priority that the task should have. Either the task's own normal priority, or if a process of a higher priority is waiting on a mutex owned by the task, then that higher priority should be returned. Since the pi_list of a task holds an order by priority list of all the top waiters of all the mutexes that the task owns, rt_mutex_getprio simply needs to compare the top pi waiter to its own normal priority, and return the higher priority back.

(Note: if looking at the code, you will notice that the lower number of prio is returned. This is because the prio field in the task structure is an inverse order of the actual priority. So a "prio" of 5 is of higher priority than a "prio" of 10.)

__rt_mutex_adjust_prio examines the result of rt_mutex_getprio, and if the result does not equal the task's current priority, then rt_mutex_setprio is called to adjust the priority of the task to the new priority. Note that rt_mutex_setprio is defined in kernel/sched.c to implement the actual change in priority.

It is interesting to note that __rt_mutex_adjust_prio can either increase or decrease the priority of the task. In the case that a higher priority process has just blocked on a mutex owned by the task, __rt_mutex_adjust_prio

would increase/boost the task's priority. But if a higher priority task were for some reason to leave the mutex (timeout or signal), this same function would decrease/unboost the priority of the task. That is because the pi_list always contains the highest priority task that is waiting on a mutex owned by the task, so we only need to compare the priority of that top pi waiter to the normal priority of the given task.

High level overview of the PI chain walk

The PI chain walk is implemented by the function rt_mutex_adjust_prio_chain.

The implementation has gone through several iterations, and has ended up with what we believe is the best. It walks the PI chain by only grabbing at most two locks at a time, and is very efficient.

The rt_mutex_adjust_prio_chain can be used either to boost or lower process priorities.

rt_mutex_adjust_prio_chain is called with a task to be checked for PI (de)boosting (the owner of a mutex that a process is blocking on), a flag to check for deadlocking, the mutex that the task owns, and a pointer to a waiter that is the process's waiter struct that is blocked on the mutex (although this parameter may be NULL for deboosting).

For this explanation, I will not mention deadlock detection. This explanation will try to stay at a high level.

When this function is called, there are no locks held. That also means that the state of the owner and lock can change when entered into this function.

Before this function is called, the task has already had rt_mutex_adjust_prio performed on it. This means that the task is set to the priority that it should be at, but the plist nodes of the task's waiter have not been updated with the new priorities, and that this task may not be in the proper locations in the pi_lists and wait_lists that the task is blocked on. This function solves all that.

A loop is entered, where task is the owner to be checked for PI changes that was passed by parameter (for the first iteration). The pi_lock of this task is taken to prevent any more changes to the pi_list of the task. This also prevents new tasks from completing the blocking on a mutex that is owned by this task.

If the task is not blocked on a mutex then the loop is exited. We are at the top of the PI chain.

A check is now done to see if the original waiter (the process that is blocked on the current mutex) is the top pi waiter of the task. That is, is this waiter on the top of the task's pi_list. If it is not, it either means that there is another process higher in priority that is blocked on one of the mutexes that the task owns, or that the waiter has just woken up via a signal or timeout and has left the PI chain. In either case, the loop is exited, since we don't need to do any more changes to the priority of the current task, or any task that owns a mutex that this current task is waiting on. A priority chain

walk is only needed when a new top pi waiter is made to a task.

The next check sees if the task's waiter plist node has the priority equal to the priority the task is set at. If they are equal, then we are done with the loop. Remember that the function started with the priority of the task adjusted, but the plist nodes that hold the task in other processes pi_lists have not been adjusted.

Next, we look at the mutex that the task is blocked on. The mutex's wait_lock is taken. This is done by a spin_trylock, because the locking order of the pi_lock and wait_lock goes in the opposite direction. If we fail to grab the lock, the pi_lock is released, and we restart the loop.

Now that we have both the pi_lock of the task as well as the wait_lock of the mutex the task is blocked on, we update the task's waiter's plist node that is located on the mutex's wait_list.

Now we release the pi_lock of the task.

Next the owner of the mutex has its pi_lock taken, so we can update the task's entry in the owner's pi_list. If the task is the highest priority process on the mutex's wait_list, then we remove the previous top waiter from the owner's pi_list, and replace it with the task.

Note: It is possible that the task was the current top waiter on the mutex, in which case the task is not yet on the pi_list of the waiter. This is OK, since plist_del does nothing if the plist node is not on any list.

If the task was not the top waiter of the mutex, but it was before we did the priority updates, that means we are deboosting/lowering the task. In this case, the task is removed from the pi_list of the owner, and the new top waiter is added.

Lastly, we unlock both the pi_lock of the task, as well as the mutex's wait_lock, and continue the loop again. On the next iteration of the loop, the previous owner of the mutex will be the task that will be processed.

Note: One might think that the owner of this mutex might have changed since we just grab the mutex's wait_lock. And one could be right. The important thing to remember is that the owner could not have become the task that is being processed in the PI chain, since we have taken that task's pi_lock at the beginning of the loop. So as long as there is an owner of this mutex that is not the same process as the tasked being worked on, we are OK.

Looking closely at the code, one might be confused. The check for the end of the PI chain is when the task isn't blocked on anything or the task's waiter structure "task" element is NULL. This check is protected only by the task's pi_lock. But the code to unlock the mutex sets the task's waiter structure "task" element to NULL with only the protection of the mutex's wait_lock, which was not taken yet. Isn't this a race condition if the task becomes the new owner?

The answer is No! The trick is the spin_trylock of the mutex's

wait_lock. If we fail that lock, we release the pi_lock of the task and continue the loop, doing the end of PI chain check again.

In the code to release the lock, the wait_lock of the mutex is held the entire time, and it is not let go when we grab the pi_lock of the new owner of the mutex. So if the switch of a new owner were to happen after the check for end of the PI chain and the grabbing of the wait_lock, the unlocking code would spin on the new owner's pi_lock but never give up the wait_lock. So the PI chain loop is guaranteed to fail the spin_trylock on the wait_lock, release the pi_lock, and try again.

If you don't quite understand the above, that's OK. You don't have to, unless you really want to make a proof out of it ;)

Pending Owners and Lock stealing

One of the flags in the owner field of the mutex structure is "Pending Owner". What this means is that an owner was chosen by the process releasing the mutex, but that owner has yet to wake up and actually take the mutex.

Why is this important? Why can't we just give the mutex to another process and be done with it?

The PI code is to help with real-time processes, and to let the highest priority process run as long as possible with little latencies and delays. If a high priority process owns a mutex that a lower priority process is blocked on, when the mutex is released it would be given to the lower priority process. What if the higher priority process wants to take that mutex again. The high priority process would fail to take that mutex that it just gave up and it would need to boost the lower priority process to run with full latency of that critical section (since the low priority process just entered it).

There's no reason a high priority process that gives up a mutex should be penalized if it tries to take that mutex again. If the new owner of the mutex has not woken up yet, there's no reason that the higher priority process could not take that mutex away.

To solve this, we introduced Pending Ownership and Lock Stealing. When a new process is given a mutex that it was blocked on, it is only given pending ownership. This means that it's the new owner, unless a higher priority process comes in and tries to grab that mutex. If a higher priority process does come along and wants that mutex, we let the higher priority process "steal" the mutex from the pending owner (only if it is still pending) and continue with the mutex.

Taking of a mutex (The walk through)

OK, now let's take a look at the detailed walk through of what happens when taking a mutex.

The first thing that is tried is the fast taking of the mutex. This is done when we have CMPXCHG enabled (otherwise the fast taking automatically fails). Only when the owner field of the mutex is NULL can the lock be taken with the CMPXCHG and nothing else needs to be done.

If there is contention on the lock, whether it is owned or pending owner we go about the slow path (rt_mutex_slowlock).

The slow path function is where the task's waiter structure is created on the stack. This is because the waiter structure is only needed for the scope of this function. The waiter structure holds the nodes to store the task on the wait_list of the mutex, and if need be, the pi_list of the owner.

The wait_lock of the mutex is taken since the slow path of unlocking the mutex also takes this lock.

We then call try_to_take_rt_mutex. This is where the architecture that does not implement CMPXCHG would always grab the lock (if there's no contention).

try_to_take_rt_mutex is used every time the task tries to grab a mutex in the slow path. The first thing that is done here is an atomic setting of the "Has Waiters" flag of the mutex's owner field. Yes, this could really be false, because if the mutex has no owner, there are no waiters and the current task also won't have any waiters. But we don't have the lock yet, so we assume we are going to be a waiter. The reason for this is to play nice for those architectures that do have CMPXCHG. By setting this flag now, the owner of the mutex can't release the mutex without going into the slow unlock path, and it would then need to grab the wait_lock, which this code currently holds. So setting the "Has Waiters" flag forces the owner to synchronize with this code.

Now that we know that we can't have any races with the owner releasing the mutex, we check to see if we can take the ownership. This is done if the mutex doesn't have a owner, or if we can steal the mutex from a pending owner. Let's look at the situations we have here.

1) Has owner that is pending

The mutex has a owner, but it hasn't woken up and the mutex flag "Pending Owner" is set. The first check is to see if the owner isn't the current task. This is because this function is also used for the pending owner to grab the mutex. When a pending owner wakes up, it checks to see if it can take the mutex, and this is done if the owner is already set to itself. If so, we succeed and leave the function, clearing the "Pending Owner" bit.

If the pending owner is not current, we check to see if the current priority is
higher than the pending owner. If not, we fail the function and return.

There's also something special about a pending owner. That is a pending owner is never blocked on a mutex. So there is no PI chain to worry about. It also means that if the mutex doesn't have any waiters, there's no accounting needed

to update the pending owner's pi_list, since we only worry about processes blocked on the current mutex.

If there are waiters on this mutex, and we just stole the ownership, we need to take the top waiter, remove it from the pi_list of the pending owner, and add it to the current pi_list. Note that at this moment, the pending owner is no longer on the list of waiters. This is fine, since the pending owner would add itself back when it realizes that it had the ownership stolen from itself. When the pending owner tries to grab the mutex, it will fail in try_to_take_rt_mutex if the owner field points to another process.

2) No owner

If there is no owner (or we successfully stole the lock), we set the owner of the mutex to current, and set the flag of "Has Waiters" if the current mutex actually has waiters, or we clear the flag if it doesn't. See, it was OK that we set that flag early, since now it is cleared.

3) Failed to grab ownership

The most interesting case is when we fail to take ownership. This means that there exists an owner, or there's a pending owner with equal or higher priority than the current task.

We'll continue on the failed case.

If the mutex has a timeout, we set up a timer to go off to break us out of this mutex if we failed to get it after a specified amount of time.

Now we enter a loop that will continue to try to take ownership of the mutex, or fail from a timeout or signal.

Once again we try to take the mutex. This will usually fail the first time in the loop, since it had just failed to get the mutex. But the second time in the loop, this would likely succeed, since the task would likely be the pending owner.

If the mutex is TASK_INTERRUPTIBLE a check for signals and timeout is done here.

The waiter structure has a "task" field that points to the task that is blocked on the mutex. This field can be NULL the first time it goes through the loop or if the task is a pending owner and had its mutex stolen. If the "task" field is NULL then we need to set up the accounting for it.

Task blocks on mutex

The accounting of a mutex and process is done with the waiter structure of the process. The "task" field is set to the process, and the "lock" field to the mutex. The plist nodes are initialized to the processes current priority.

Since the wait_lock was taken at the entry of the slow lock, we can safely

add the waiter to the wait_list. If the current process is the highest priority process currently waiting on this mutex, then we remove the previous top waiter process (if it exists) from the pi_list of the owner, and add the current process to that list. Since the pi_list of the owner has changed, we call rt_mutex_adjust_prio on the owner to see if the owner should adjust its priority accordingly.

If the owner is also blocked on a lock, and had its pi_list changed (or deadlock checking is on), we unlock the wait_lock of the mutex and go ahead and run rt_mutex_adjust_prio_chain on the owner, as described earlier.

Now all locks are released, and if the current process is still blocked on a mutex (waiter "task" field is not NULL), then we go to sleep (call schedule).

Waking up in the loop

The schedule can then wake up for a few reasons.
1) we were given pending ownership of the mutex.
2) we received a signal and was TASK_INTERRUPTIBLE
3) we had a timeout and was TASK_INTERRUPTIBLE

In any of these cases, we continue the loop and once again try to grab the ownership of the mutex. If we succeed, we exit the loop, otherwise we continue and on signal and timeout, will exit the loop, or if we had the mutex stolen we just simply add ourselves back on the lists and go back to sleep.

Note: For various reasons, because of timeout and signals, the steal mutex algorithm needs to be careful. This is because the current process is still on the wait_list. And because of dynamic changing of priorities, especially on SCHED_OTHER tasks, the current process can be the highest priority task on the wait_list.

Failed to get mutex on Timeout or Signal

If a timeout or signal occurred, the waiter's "task" field would not be NULL and the task needs to be taken off the wait_list of the mutex and perhaps pi_list of the owner. If this process was a high priority process, then the rt_mutex_adjust_prio_chain needs to be executed again on the owner, but this time it will be lowering the priorities.

Unlocking the Mutex

The unlocking of a mutex also has a fast path for those architectures with CMPXCHG. Since the taking of a mutex on contention always sets the "Has Waiters" flag of the mutex's owner, we use this to know if we need to take the slow path when unlocking the mutex. If the mutex doesn't have any waiters, the owner field of the mutex would equal the current process and the mutex can be unlocked by just replacing the owner field with NULL.

If the owner field has the "Has Waiters" bit set (or CMPXCHG is not available), the slow unlock path is taken.

The first thing done in the slow unlock path is to take the wait_lock of the mutex. This synchronizes the locking and unlocking of the mutex.

A check is made to see if the mutex has waiters or not. On architectures that do not have CMPXCHG, this is the location that the owner of the mutex will determine if a waiter needs to be awoken or not. On architectures that do have CMPXCHG, that check is done in the fast path, but it is still needed in the slow path too. If a waiter of a mutex woke up because of a signal or timeout between the time the owner failed the fast path CMPXCHG check and the grabbing of the wait_lock, the mutex may not have any waiters, thus the owner still needs to make this check. If there are no waiters then the mutex owner field is set to NULL, the wait_lock is released and nothing more is needed.

If there are waiters, then we need to wake one up and give that waiter pending ownership.

On the wake up code, the pi_lock of the current owner is taken. The top waiter of the lock is found and removed from the wait_list of the mutex as well as the pi_list of the current owner. The task field of the new pending owner's waiter structure is set to NULL, and the owner field of the mutex is set to the new owner with the "Pending Owner" bit set, as well as the "Has Waiters" bit if there still are other processes blocked on the mutex.

The pi_lock of the previous owner is released, and the new pending owner's pi_lock is taken. Remember that this is the trick to prevent the race condition in rt_mutex_adjust_prio_chain from adding itself as a waiter on the mutex.

We now clear the "pi_blocked_on" field of the new pending owner, and if the mutex still has waiters pending, we add the new top waiter to the pi_list of the pending owner.

Finally we unlock the pi_lock of the pending owner and wake it up.

Contact

For updates on this document, please email Steven Rostedt <rostedt@goodmis.org>

Credits

Author: Steven Rostedt <rostedt@goodmis.org>

Reviewers: Ingo Molnar, Thomas Gleixner, Thomas Duetsch, and Randy Dunlap

Updates

This document was originally written for 2.6.17-rc3-mm1