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/locking/rt-mutex.rst. 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 where 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:

```
     grab lock L1 (owned by C)
       |
A ---+
         C preempted by B
           |
C    +-----+

B          +-------->
                B now keeps A from running.
```

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 rbtree node structures to place the task in the waiters rbtree of a mutex as well as the pi_waiters rbtree 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:

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

If process G has the highest priority in the chain, then all the tasks up the chain (A and B in this example), must have their priorities increased to that of G.

Mutex Waiters Tree

Every mutex keeps track of all the waiters that are blocked on itself. The mutex has a rbtree to store these waiters by priority. This tree is protected by a spin lock that is located in the struct of the mutex. This lock is called wait_lock.

Task PI Tree

To keep track of the PI chains, each process has its own PI rbtree. This is a tree of all top waiters of the mutexes that are owned by the process. Note that this tree 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 tree 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 tree.

This tree is stored in the task structure of a process as a rbtree called pi_waiters. It 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 two byte alignment (and if this is not true, the rtmutex.c code will be broken!), this allows for the least significant bit to be used as a flag. Bit 0 is used as the "Has Waiters" flag. It's set whenever there are waiters on a mutex.

See Documentation/locking/rt-mutex.rst for further details.

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_waiters 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 and rt_mutex_setprio. rt_mutex_setprio is only used in rt_mutex_adjust_prio.

rt_mutex_adjust_prio examines the priority of the task, and the highest priority process that is waiting any of mutexes owned by the task. Since the pi_waiters of a task holds an order by priority of all the top waiters of all the mutexes that the task owns, we simply need to compare the top pi waiter to its own normal/deadline priority and take the higher one. 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/core.c to implement the actual change in priority.

Note:

> For the "prio" field in task_struct, the lower the number, the higher the priority. A "prio" of 5 is of higher priority than a "prio" of 10.

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_waiters 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, 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), a pointer to the mutex on which the task is blocked, and a top_task as the top waiter of the mutex.

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 rbtree nodes of the task's waiter have not been updated with the new priorities, and this task may not be in the proper locations in the pi_waiters and waiters trees that the task is blocked on. This function solves all that.

The main operation of this function is summarized by Thomas Gleixner in rtmutex.c. See the 'Chain walk basics and protection scope' comment for further details.

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, 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 waiters tree of the mutex, and if need be, the pi_waiters tree 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. By setting this flag now, the current owner of the mutex being contended for 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 current owner to synchronize with this code.

The lock is taken if the following are true:

1. The lock has no owner
2. The current task is the highest priority against all other waiters of the lock

If the task succeeds to acquire the lock, then the task is set as the owner of the lock, and if the lock still has waiters, the top_waiter (highest priority task waiting on the lock) is added to this task's pi_waiters tree.

If the lock is not taken by try_to_take_rt_mutex(), then the task_blocks_on_rt_mutex() function is called. This will add the task to the lock's waiter tree and propagate the pi chain of the lock as well as the lock's owner's pi_waiters tree. This is described in the next section.

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 rbtree node of waiter 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 task waiter tree. 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_waiters of the owner, and add the current process to that tree. Since the pi_waiter 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_waiters 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).

W[illegible]

Waking up in the loop

The task can then wake up for a couple of reasons:

1. The previous lock owner released the lock, and the task now is top_waiter
2. we received a signal or timeout

In both cases, the task will try again to acquire the lock. If it does, then it will take itself off the waiters tree and set itself back to the TASK_RUNNING state.

In first case, if the lock was acquired by another task before this task could get the lock, then it will go back to sleep and wait to be woken again.

The second case is only applicable for tasks that are grabbing a mutex that can wake up before getting the lock, either due to a signal or a timeout (i.e. rt_mutex_timed_futex_lock()). When woken, it will try to take the lock again, if it succeeds, then the task will return with the lock held, otherwise it will return with -EINTR if the task was woken by a signal, or -ETIMEDOUT if it timed out.

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.

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 waiters tree of the mutex as well as the pi_waiters tree of the current owner. The "Has Waiters" bit is marked to prevent lower priority tasks from stealing the lock.

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>

Updated: Alex Shi <alex.shi@linaro.org> - 7/6/2017

Original Reviewers:

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

Update (7/6/2017) Reviewers: Steven Rostedt and Sebastian Siewior

Updates

This document was originally written for 2.6.17-rc3-mm1 was updated on 4.12