Wound/Wait Deadlock-Proof Mutex Design

Please read mutex-design.rst first, as it applies to wait/wound mutexes too.

Motivation for WW-Mutexes

GPU's do operations that commonly involve many buffers. Those buffers can be shared across contexts/processes, exist in different memory domains (for example VRAM vs system memory), and so on. And with PRIME / dmabuf, they can even be shared across devices. So there are a handful of situations where the driver needs to wait for buffers to become ready. If you think about this in terms of waiting on a buffer mutex for it to become available, this presents a problem because there is no way to guarantee that buffers appear in a execbuf/batch in the same order in all contexts. That is directly under control of userspace, and a result of the sequence of GL calls that an application makes. Which results in the potential for deadlock. The problem gets more complex when you consider that the kernel may need to migrate the buffer(s) into VRAM before the GPU operates on the buffer(s), which may in turn require evicting some other buffers (and you don't want to evict other buffers which are already queued up to the GPU), but for a simplified understanding of the problem you can ignore this.

The algorithm that the TTM graphics subsystem came up with for dealing with this problem is quite simple. For each group of buffers (execbuf) that need to be locked, the caller would be assigned a unique reservation id/ticket, from a global counter. In case of deadlock while locking all the buffers associated with a execbuf, the one with the lowest reservation ticket (i.e. the oldest task) wins, and the one with the higher reservation id (i.e. the younger task) unlocks all of the buffers that it has already locked, and then tries again.

In the RDBMS literature, a reservation ticket is associated with a transaction. and the deadlock handling approach is called Wait-Die. The name is based on the actions of a locking thread when it encounters an already locked mutex. If the transaction holding the lock is younger, the locking transaction waits. If the transaction holding the lock is older, the locking transaction backs off and dies. Hence Wait-Die. There is also another algorithm called Wound-Wait: If the transaction holding the lock is younger, the locking transaction wounds the transaction holding the lock, requesting it to die. If the transaction holding the lock is older, it waits for the other transaction. Hence Wound-Wait. The two algorithms are both fair in that a transaction will eventually succeed. However, the Wound-Wait algorithm is typically stated to generate fewer backoffs compared to Wait-Die, but is, on the other hand, associated with more work than Wait-Die when recovering from a backoff. Wound-Wait is also a preemptive algorithm in that transactions are wounded by other transactions, and that requires a reliable way to pick up the wounded condition and preempt the running transaction. Note that this is not the same as process preemption. A Wound-Wait transaction is considered preempted when it dies (returning -EDEADLK) following a wound.

Concepts

Compared to normal mutexes two additional concepts/objects show up in the lock interface for w/w mutexes:

Acquire context: To ensure eventual forward progress it is important that a task trying to acquire locks doesn't grab a new reservation id, but keeps the one it acquired when starting the lock acquisition. This ticket is stored in the acquire context. Furthermore the acquire context keeps track of debugging state to catch w/w mutex interface abuse. An acquire context is representing a transaction.

W/w class: In contrast to normal mutexes the lock class needs to be explicit for w/w mutexes, since it is required to initialize the acquire context. The lock class also specifies what algorithm to use, Wound-Wait or Wait-Die.

Furthermore there are three different class of w/w lock acquire functions:

- Normal lock acquisition with a context, using ww_mutex_lock.
- Slowpath lock acquisition on the contending lock, used by the task that just killed its transaction after having dropped all already acquired locks. These functions have the _slow postfix.

  From a simple semantics point-of-view the _slow functions are not strictly required, since simply calling the normal ww_mutex_lock functions on the contending lock (after having dropped all other already acquired locks) will work correctly. After all if no other ww mutex has been acquired yet there's no deadlock potential and hence the ww_mutex_lock call will block and not prematurely return -EDEADLK. The advantage of the _slow functions is in interface safety:

  - ww_mutex_lock has a __must_check int return type, whereas ww_mutex_lock_slow has a void return type. Note that since ww mutex code needs loops/retries anyway the __must_check doesn't result in spurious warnings, even though the very first lock operation can never fail.
  - When full debugging is enabled ww_mutex_lock_slow checks that all acquired ww mutex have been released (preventing deadlocks) and makes sure that we block on the contending lock (preventing spinning through the -EDEADLK slowpath until the contended lock can be acquired).
- Functions to only acquire a single w/w mutex, which results in the exact same semantics as a normal mutex. This is done by calling ww_mutex_lock with a NULL context.

  Again this is not strictly required. But often you only want to acquire a single lock in which case it's pointless to set up an acquire context (and so better to avoid grabbing a deadlock avoidance ticket).

Of course, all the usual variants for handling wake-ups due to signals are also provided.

Usage

The algorithm (Wait-Die vs Wound-Wait) is chosen by using either DEFINE_WW_CLASS() (Wound-Wait) or DEFINE_WD_CLASS() (Wait-Die) As a rough rule of thumb, use Wound-Wait iff you expect the number of simultaneous competing transactions to be typically small, and you want to reduce the number of rollbacks.

Three different ways to acquire locks within the same w/w class. Common definitions for methods #1 and #2:

```
static DEFINE_WW_CLASS(ww_class);

struct obj {
      struct ww_mutex lock;
      /* obj data */
};

struct obj_entry {
      struct list_head head;
      struct obj *obj;
};
```

Method 1, using a list in execbuf->buffers that's not allowed to be reordered. This is useful if a list of required objects is already tracked somewhere. Furthermore the lock helper can use propagate the -EALREADY return code back to the caller as a signal that an object is twice on the list. This is useful if the list is constructed from userspace input and the ABI requires userspace to not have duplicate entries (e.g. for a gpu commandbuffer submission ioctl):

```
int lock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
      struct obj *res_obj = NULL;
      struct obj_entry *contended_entry = NULL;
      struct obj_entry *entry;

      ww_acquire_init(ctx, &ww_class);

retry:
      list_for_each_entry (entry, list, head) {
              if (entry->obj == res_obj) {
                      res_obj = NULL;
                      continue;
              }
              ret = ww_mutex_lock(&entry->obj->lock, ctx);
              if (ret < 0) {
                      contended_entry = entry;
                      goto err;
              }
      }

      ww_acquire_done(ctx);
      return 0;

err:
      list_for_each_entry_continue_reverse (entry, list, head)
              ww_mutex_unlock(&entry->obj->lock);

      if (res_obj)
              ww_mutex_unlock(&res_obj->lock);

      if (ret == -EDEADLK) {
              /* we lost out in a seqno race, lock and retry.. */
              ww_mutex_lock_slow(&contended_entry->obj->lock, ctx);
              res_obj = contended_entry->obj;
              goto retry;
      }
      ww_acquire_fini(ctx);

      return ret;
}
```

Method 2, using a list in execbuf->buffers that can be reordered. Same semantics of duplicate entry detection using -EALREADY as method 1 above. But the list-reordering allows for a bit more idiomatic code:

```
int lock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
      struct obj_entry *entry, *entry2;

      ww_acquire_init(ctx, &ww_class);

      list_for_each_entry (entry, list, head) {
```

```
                ret = ww_mutex_lock(&entry->obj->lock, ctx);
                if (ret < 0) {
                        entry2 = entry;

                        list_for_each_entry_continue_reverse (entry2, list, head)
                                ww_mutex_unlock(&entry2->obj->lock);

                        if (ret != -EDEADLK) {
                                ww_acquire_fini(ctx);
                                return ret;
                        }

                        /* we lost out in a seqno race, lock and retry.. */
                        ww_mutex_lock_slow(&entry->obj->lock, ctx);

                        /*
                         * Move buf to head of the list, this will point
                         * buf->next to the first unlocked entry,
                         * restarting the for loop.
                         */
                        list_del(&entry->head);
                        list_add(&entry->head, list);
                }
        }

        ww_acquire_done(ctx);
        return 0;
}
```

Unlocking works the same way for both methods #1 and #2:

```
void unlock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
        struct obj_entry *entry;

        list_for_each_entry (entry, list, head)
                ww_mutex_unlock(&entry->obj->lock);

        ww_acquire_fini(ctx);
}
```

Method 3 is useful if the list of objects is constructed ad-hoc and not upfront, e.g. when adjusting edges in a graph where each node has its own ww_mutex lock, and edges can only be changed when holding the locks of all involved nodes. w/w mutexes are a natural fit for such a case for two reasons:

- They can handle lock-acquisition in any order which allows us to start walking a graph from a starting point and then iteratively discovering new edges and locking down the nodes those edges connect to.
- Due to the -EALREADY return code signalling that a given objects is already held there's no need for additional book-keeping to break cycles in the graph or keep track off which looks are already held (when using more than one node as a starting point).

Note that this approach differs in two important ways from the above methods:

- Since the list of objects is dynamically constructed (and might very well be different when retrying due to hitting the -EDEADLK die condition) there's no need to keep any object on a persistent list when it's not locked. We can therefore move the list_head into the object itself.
- On the other hand the dynamic object list construction also means that the -EALREADY return code can't be propagated.

Note also that methods #1 and #2 and method #3 can be combined, e.g. to first lock a list of starting nodes (passed in from userspace) using one of the above methods. And then lock any additional objects affected by the operations using method #3 below. The backoff/retry procedure will be a bit more involved, since when the dynamic locking step hits -EDEADLK we also need to unlock all the objects acquired with the fixed list. But the w/w mutex debug checks will catch any interface misuse for these cases.

Also, method 3 can't fail the lock acquisition step since it doesn't return -EALREADY. Of course this would be different when using the _interruptible variants, but that's outside of the scope of these examples here:

```
struct obj {
        struct ww_mutex ww_mutex;
        struct list_head locked_list;
};

static DEFINE_WW_CLASS(ww_class);

void __unlock_objs(struct list_head *list)
{
        struct obj *entry, *temp;

        list_for_each_entry_safe (entry, temp, list, locked_list) {
                /* need to do that before unlocking, since only the current lock holder is
                allowed to use object */
```

```
                list_del(&entry->locked_list);
                ww_mutex_unlock(entry->ww_mutex)
        }
}

void lock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
        struct obj *obj;

        ww_acquire_init(ctx, &ww_class);

retry:
        /* re-init loop start state */
        loop {
                /* magic code which walks over a graph and decides which objects
                 * to lock */

                ret = ww_mutex_lock(obj->ww_mutex, ctx);
                if (ret == -EALREADY) {
                        /* we have that one already, get to the next object */
                        continue;
                }
                if (ret == -EDEADLK) {
                        __unlock_objs(list);

                        ww_mutex_lock_slow(obj, ctx);
                        list_add(&entry->locked_list, list);
                        goto retry;
                }

                /* locked a new object, add it to the list */
                list_add_tail(&entry->locked_list, list);
        }

        ww_acquire_done(ctx);
        return 0;
}

void unlock_objs(struct list_head *list, struct ww_acquire_ctx *ctx)
{
        __unlock_objs(list);
        ww_acquire_fini(ctx);
}
```

Method 4: Only lock one single objects. In that case deadlock detection and prevention is obviously overkill, since with grabbing just one lock you can't produce a deadlock within just one class. To simplify this case the w/w mutex api can be used with a NULL context.

Implementation Details

Design:

ww_mutex currently encapsulates a struct mutex, this means no extra overhead for normal mutex locks, which are far more common. As such there is only a small increase in code size if wait/wound mutexes are not used.

We maintain the following invariants for the wait list:

1. Waiters with an acquire context are sorted by stamp order; waiters without an acquire context are interspersed in FIFO order.
2. For Wait-Die, among waiters with contexts, only the first one can have other locks acquired already (ctx->acquired > 0). Note that this waiter may come after other waiters without contexts in the list.

The Wound-Wait preemption is implemented with a lazy-preemption scheme: The wounded status of the transaction is checked only when there is contention for a new lock and hence a true chance of deadlock. In that situation, if the transaction is wounded, it backs off, clears the wounded status and retries. A great benefit of implementing preemption in this way is that the wounded transaction can identify a contending lock to wait for before restarting the transaction. Just blindly restarting the transaction would likely make the transaction end up in a situation where it would have to back off again.

In general, not much contention is expected. The locks are typically used to serialize access to resources for devices, and optimization focus should therefore be directed towards the uncontended cases.

Lockdep:

Special care has been taken to warn for as many cases of api abuse as possible. Some common api abuses will be caught with CONFIG_DEBUG_MUTEXES, but CONFIG_PROVE_LOCKING is recommended.

Some of the errors which will be warned about:

- Forgetting to call ww_acquire_fini or ww_acquire_init.
- Attempting to lock more mutexes after ww_acquire_done.
- Attempting to lock the wrong mutex after -EDEADLK and unlocking all mutexes.
- Attempting to lock the right mutex after -EDEADLK, before unlocking all mutexes.
- Calling ww_mutex_lock_slow before -EDEADLK was returned.
- Unlocking mutexes with the wrong unlock function.
- Calling one of the ww_acquire_* twice on the same context.
- Using a different ww_class for the mutex than for the ww_acquire_ctx.
- Normal lockdep errors that can result in deadlocks.

Some of the lockdep errors that can result in deadlocks:

- Calling ww_acquire_init to initialize a second ww_acquire_ctx before having called ww_acquire_fini on the first.
- 'normal' deadlocks that can occur.

FIXME:

Update this section once we have the TASK_DEADLOCK task state flag magic implemented.