refcount_t API compared to atomic_t

Introduction

The goal of refcount_t API is to provide a minimal API for implementing an object's reference counters. While a generic architecture-independent implementation from lib/refcount.c uses atomic operations underneath, there are a number of differences between some of the `refcount_*()` and `atomic_*()` functions with regards to the memory ordering guarantees. This document outlines the differences and provides respective examples in order to help maintainers validate their code against the change in these memory ordering guarantees.

The terms used through this document try to follow the formal LKMM defined in tools/memory-model/Documentation/explanation.txt.

memory-barriers.txt and atomic_t.txt provide more background to the memory ordering in general and for atomic operations specifically.

Relevant types of memory ordering

> **Note**
>
> The following section only covers some of the memory ordering types that are relevant for the atomics and reference counters and used through this document. For a much broader picture please consult memory-barriers.txt document.

In the absence of any memory ordering guarantees (i.e. fully unordered) atomics & refcounters only provide atomicity and program order (po) relation (on the same CPU). It guarantees that each `atomic_*()` and `refcount_*()` operation is atomic and instructions are executed in program order on a single CPU. This is implemented using READ_ONCE()/WRITE_ONCE() and compare-and-swap primitives.

A strong (full) memory ordering guarantees that all prior loads and stores (all po-earlier instructions) on the same CPU are completed before any po-later instruction is executed on the same CPU. It also guarantees that all po-earlier stores on the same CPU and all propagated stores from other CPUs must propagate to all other CPUs before any po-later instruction is executed on the original CPU (A-cumulative property). This is implemented using smp_mb().

A RELEASE memory ordering guarantees that all prior loads and stores (all po-earlier instructions) on the same CPU are completed before the operation. It also guarantees that all po-earlier stores on the same CPU and all propagated stores from other CPUs must propagate to all other CPUs before the release operation (A-cumulative property). This is implemented using smp_store_release().

An ACQUIRE memory ordering guarantees that all post loads and stores (all po-later instructions) on the same CPU are completed after the acquire operation. It also guarantees that all po-later stores on the same CPU must propagate to all other CPUs after the acquire operation executes. This is implemented using smp_acquire__after_ctrl_dep().

A control dependency (on success) for refcounters guarantees that if a reference for an object was successfully obtained (reference counter increment or addition happened, function returned true), then further stores are ordered against this operation. Control dependency on stores are not implemented using any explicit barriers, but rely on CPU not to speculate on stores. This is only a single CPU relation and provides no guarantees for other CPUs.

Comparison of functions

case 1) - non-"Read/Modify/Write" (RMW) ops

Function changes:

- atomic_set() --> refcount_set()
- atomic_read() --> refcount_read()

Memory ordering guarantee changes:

- none (both fully unordered)

case 2) - increment-based ops that return no value

Function changes:

- atomic_inc() --> refcount_inc()
- atomic_add() --> refcount_add()

Memory ordering guarantee changes:

- none (both fully unordered)

case 3) - decrement-based RMW ops that return no value

Function changes:

- atomic_dec() --> refcount_dec()

Memory ordering guarantee changes:

- fully unordered --> RELEASE ordering

case 4) - increment-based RMW ops that return a value

Function changes:

- atomic_inc_not_zero() --> refcount_inc_not_zero()
- no atomic counterpart --> refcount_add_not_zero()

Memory ordering guarantees changes:

- fully ordered --> control dependency on success for stores

> **Note**
> We really assume here that necessary ordering is provided as a result of obtaining pointer to the object!

case 5) - generic dec/sub decrement-based RMW ops that return a value

Function changes:

- atomic_dec_and_test() --> refcount_dec_and_test()
- atomic_sub_and_test() --> refcount_sub_and_test()

Memory ordering guarantees changes:

- fully ordered --> RELEASE ordering + ACQUIRE ordering on success

case 6) other decrement-based RMW ops that return a value

Function changes:

- no atomic counterpart --> refcount_dec_if_one()
- `atomic_add_unless(&var, -1, 1)` --> `refcount_dec_not_one(&var)`

Memory ordering guarantees changes:

- fully ordered --> RELEASE ordering + control dependency

> **Note**
> atomic_add_unless() only provides full order on success.

case 7) - lock-based RMW

Function changes:

- atomic_dec_and_lock() --> refcount_dec_and_lock()

- atomic_dec_and_mutex_lock() --> refcount_dec_and_mutex_lock()

Memory ordering guarantees changes:

- fully ordered --> RELEASE ordering + control dependency + hold spin_lock() on success