Sequence counters and sequential locks

Introduction

Sequence counters are a reader-writer consistency mechanism with lockless readers (read-only retry loops), and no writer starvation. They are used for data that's rarely written to (e.g. system time), where the reader wants a consistent set of information and is willing to retry if that information changes.

A data set is consistent when the sequence count at the beginning of the read side critical section is even and the same sequence count value is read again at the end of the critical section. The data in the set must be copied out inside the read side critical section. If the sequence count has changed between the start and the end of the critical section, the reader must retry.

Writers increment the sequence count at the start and the end of their critical section. After starting the critical section the sequence count is odd and indicates to the readers that an update is in progress. At the end of the write side critical section the sequence count becomes even again which lets readers make progress.

A sequence counter write side critical section must never be preempted or interrupted by read side sections. Otherwise the reader will spin for the entire scheduler tick due to the odd sequence count value and the interrupted writer. If that reader belongs to a real-time scheduling class, it can spin forever and the kernel will livelock.

This mechanism cannot be used if the protected data contains pointers, as the writer can invalidate a pointer that the reader is following.

Sequence counters (`seqcount_t`)

This is the the raw counting mechanism, which does not protect against multiple writers. Write side critical sections must thus be serialized by an external lock.

If the write serialization primitive is not implicitly disabling preemption, preemption must be explicitly disabled before entering the write side section. If the read section can be invoked from hardirq or softirq contexts, interrupts or bottom halves must also be respectively disabled before entering the write section.

If it's desired to automatically handle the sequence counter requirements of writer serialization and non-preemptibility, use :ref:`seqlock_t` instead.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\locking\[linux-master][Documentation][locking]seqlock.rst`, **line 52**); *backlink*
>
> Unknown interpreted text role "ref".

Initialization:

```
/* dynamic */
seqcount_t foo_seqcount;
seqcount_init(&foo_seqcount);

/* static */
static seqcount_t foo_seqcount = SEQCNT_ZERO(foo_seqcount);

/* C99 struct init */
struct {
        .seq   = SEQCNT_ZERO(foo.seq),
} foo;
```

Write path:

```
/* Serialized context with disabled preemption */

write_seqcount_begin(&foo_seqcount);

/* ... [[write-side critical section]] ... */

write_seqcount_end(&foo_seqcount);
```

Read path:

```
do {
        seq = read_seqcount_begin(&foo_seqcount);

        /* ... [[read-side critical section]] ... */

} while (read_seqcount_retry(&foo_seqcount, seq));
```

Sequence counters with associated locks (`seqcount_LOCKNAME_t`)

As discussed at :ref:`seqcount_t`, sequence count write side critical sections must be serialized and non-preemptible. This variant of sequence counters associate the lock used for writer serialization at initialization time, which enables lockdep to validate that the write side critical sections are properly serialized.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\locking\[linux-master] [Documentation] [locking] seqlock.rst`, line 95); *backlink*
>
> Unknown interpreted text role "ref".

This lock association is a NOOP if lockdep is disabled and has neither storage nor runtime overhead. If lockdep is enabled, the lock pointer is stored in struct seqcount and lockdep's "lock is held" assertions are injected at the beginning of the write side critical section to validate that it is properly protected.

For lock types which do not implicitly disable preemption, preemption protection is enforced in the write side function.

The following sequence counters with associated locks are defined:

- `seqcount_spinlock_t`
- `seqcount_raw_spinlock_t`
- `seqcount_rwlock_t`
- `seqcount_mutex_t`
- `seqcount_ww_mutex_t`

The sequence counter read and write APIs can take either a plain seqcount_t or any of the seqcount_LOCKNAME_t variants above.

Initialization (replace "LOCKNAME" with one of the supported locks):

```
/* dynamic */
seqcount_LOCKNAME_t foo_seqcount;
seqcount_LOCKNAME_init(&foo_seqcount, &lock);

/* static */
static seqcount_LOCKNAME_t foo_seqcount =
        SEQCNT_LOCKNAME_ZERO(foo_seqcount, &lock);

/* C99 struct init */
struct {
        .seq   = SEQCNT_LOCKNAME_ZERO(foo.seq, &lock),
} foo;
```

Write path: same as in :ref:`seqcount_t`, while running from a context with the associated write serialization lock acquired.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\locking\[linux-master] [Documentation] [locking] seqlock.rst`, line 136); *backlink*
>
> Unknown interpreted text role "ref".

Read path: same as in :ref:`seqcount_t`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\locking\[linux-master] [Documentation] [locking] seqlock.rst`, line 139); *backlink*
>
> Unknown interpreted text role "ref".

Latch sequence counters (`seqcount_latch_t`)

Latch sequence counters are a multiversion concurrency control mechanism where the embedded seqcount_t counter even/odd value is used to switch between two copies of protected data. This allows the sequence counter read path to safely interrupt its own write side critical section.

Use seqcount_latch_t when the write side sections cannot be protected from interruption by readers. This is typically the case when the read side can be invoked from NMI handlers.

Check *raw_write_seqcount_latch()* for more information.

Sequential locks (`seqlock_t`)

This contains the :ref:`seqcount_t` mechanism earlier discussed, plus an embedded spinlock for writer serialization and non-preemptibility.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\locking\[linux-master][Documentation][locking]seqlock.rst`, **line 164); *backlink***
>
> Unknown interpreted text role "ref".

If the read side section can be invoked from hardirq or softirq context, use the write side function variants which disable interrupts or bottom halves respectively.

Initialization:

```
/* dynamic */
seqlock_t foo_seqlock;
seqlock_init(&foo_seqlock);

/* static */
static DEFINE_SEQLOCK(foo_seqlock);

/* C99 struct init */
struct {
        .seql   = __SEQLOCK_UNLOCKED(foo.seql)
} foo;
```

Write path:

```
write_seqlock(&foo_seqlock);

/* ... [[write-side critical section]] ... */

write_sequnlock(&foo_seqlock);
```

Read path, three categories:

1. Normal Sequence readers which never block a writer but they must retry if a writer is in progress by detecting change in the sequence number. Writers do not wait for a sequence reader:

   ```
   do {
           seq = read_seqbegin(&foo_seqlock);

           /* ... [[read-side critical section]] ... */

   } while (read_seqretry(&foo_seqlock, seq));
   ```

2. Locking readers which will wait if a writer or another locking reader is in progress. A locking reader in progress will also block a writer from entering its critical section. This read lock is exclusive. Unlike rwlock_t, only one locking reader can acquire it:

   ```
   read_seqlock_excl(&foo_seqlock);

   /* ... [[read-side critical section]] ... */

   read_sequnlock_excl(&foo_seqlock);
   ```

3. Conditional lockless reader (as in 1), or locking reader (as in 2), according to a passed marker. This is used to avoid lockless readers starvation (too much retry loops) in case of a sharp spike in write activity. First, a lockless read is tried (even marker passed). If that trial fails (odd sequence counter is returned, which is used as the next iteration marker), the lockless read is transformed to a full locking read and no retry loop is necessary:

   ```
   /* marker; even initialization */
   int seq = 0;
   do {
           read_seqbegin_or_lock(&foo_seqlock, &seq);

           /* ... [[read-side critical section]] ... */

   } while (need_seqretry(&foo_seqlock, seq));
   done_seqretry(&foo_seqlock, seq);
   ```

API documentation

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\locking\[linux-master][Documentation][locking]seqlock.rst`, **line 239)**

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/seqlock.h
```