Circular Buffers

Author: David Howells <dhowells@redhat.com>
Author: Paul E. McKenney <paulmck@linux.ibm.com>

Linux provides a number of features that can be used to implement circular buffering. There are two sets of such features:

1. Convenience functions for determining information about power-of-2 sized buffers.
2. Memory barriers for when the producer and the consumer of objects in the buffer don't want to share a lock.

To use these facilities, as discussed below, there needs to be just one producer and just one consumer. It is possible to handle multiple producers by serialising them, and to handle multiple consumers by serialising them.

What is a circular buffer?

First of all, what is a circular buffer? A circular buffer is a buffer of fixed, finite size into which there are two indices:

1. A 'head' index - the point at which the producer inserts items into the buffer.
2. A 'tail' index - the point at which the consumer finds the next item in the buffer.

Typically when the tail pointer is equal to the head pointer, the buffer is empty; and the buffer is full when the head pointer is one less than the tail pointer.

The head index is incremented when items are added, and the tail index when items are removed. The tail index should never jump the head index, and both indices should be wrapped to 0 when they reach the end of the buffer, thus allowing an infinite amount of data to flow through the buffer.

Typically, items will all be of the same unit size, but this isn't strictly required to use the techniques below. The indices can be increased by more than 1 if multiple items or variable-sized items are to be included in the buffer, provided that neither index overtakes the other. The implementer must be careful, however, as a region more than one unit in size may wrap the end of the buffer and be broken into two segments.

Measuring power-of-2 buffers

Calculation of the occupancy or the remaining capacity of an arbitrarily sized circular buffer would normally be a slow operation, requiring the use of a modulus (divide) instruction. However, if the buffer is of a power-of-2 size, then a much quicker bitwise-AND instruction can be used instead.

Linux provides a set of macros for handling power-of-2 circular buffers. These can be made use of by:

```
#include <linux/circ_buf.h>
```

The macros are:

1. Measure the remaining capacity of a buffer:

   ```
   CIRC_SPACE(head_index, tail_index, buffer_size);
   ```

   This returns the amount of space left in the buffer[1] into which items can be inserted.

2. Measure the maximum consecutive immediate space in a buffer:

   ```
   CIRC_SPACE_TO_END(head_index, tail_index, buffer_size);
   ```

   This returns the amount of consecutive space left in the buffer[1] into which items can be immediately inserted without having to wrap back to the beginning of the buffer.

3. Measure the occupancy of a buffer:

   ```
   CIRC_CNT(head_index, tail_index, buffer_size);
   ```

   This returns the number of items currently occupying a buffer[2].

4. Measure the non-wrapping occupancy of a buffer:

   ```
   CIRC_CNT_TO_END(head_index, tail_index, buffer_size);
   ```

   This returns the number of consecutive items[2] that can be extracted from the buffer without having to wrap back to the beginning of the buffer.

Each of these macros will nominally return a value between 0 and buffer_size-1, however:

1. CIRC_SPACE*() are intended to be used in the producer. To the producer they will return a lower bound as the

producer controls the head index, but the consumer may still be depleting the buffer on another CPU and moving the tail index.

To the consumer it will show an upper bound as the producer may be busy depleting the space.

2. CIRC_CNT*() are intended to be used in the consumer. To the consumer they will return a lower bound as the consumer controls the tail index, but the producer may still be filling the buffer on another CPU and moving the head index.

   To the producer it will show an upper bound as the consumer may be busy emptying the buffer.

3. To a third party, the order in which the writes to the indices by the producer and consumer become visible cannot be guaranteed as they are independent and may be made on different CPUs - so the result in such a situation will merely be a guess, and may even be negative.

Using memory barriers with circular buffers

By using memory barriers in conjunction with circular buffers, you can avoid the need to:

1. use a single lock to govern access to both ends of the buffer, thus allowing the buffer to be filled and emptied at the same time; and
2. use atomic counter operations.

There are two sides to this: the producer that fills the buffer, and the consumer that empties it. Only one thing should be filling a buffer at any one time, and only one thing should be emptying a buffer at any one time, but the two sides can operate simultaneously.

The producer

The producer will look something like this:

```
spin_lock(&producer_lock);

unsigned long head = buffer->head;
/* The spin_unlock() and next spin_lock() provide needed ordering. */
unsigned long tail = READ_ONCE(buffer->tail);

if (CIRC_SPACE(head, tail, buffer->size) >= 1) {
        /* insert one item into the buffer */
        struct item *item = buffer[head];

        produce_item(item);

        smp_store_release(buffer->head,
                          (head + 1) & (buffer->size - 1));

        /* wake_up() will make sure that the head is committed before
         * waking anyone up */
        wake_up(consumer);
}

spin_unlock(&producer_lock);
```

This will instruct the CPU that the contents of the new item must be written before the head index makes it available to the consumer and then instructs the CPU that the revised head index must be written before the consumer is woken.

Note that wake_up() does not guarantee any sort of barrier unless something is actually awakened. We therefore cannot rely on it for ordering. However, there is always one element of the array left empty. Therefore, the producer must produce two elements before it could possibly corrupt the element currently being read by the consumer. Therefore, the unlock-lock pair between consecutive invocations of the consumer provides the necessary ordering between the read of the index indicating that the consumer has vacated a given element and the write by the producer to that same element.

The Consumer

The consumer will look something like this:

```
spin_lock(&consumer_lock);

/* Read index before reading contents at that index. */
unsigned long head = smp_load_acquire(buffer->head);
unsigned long tail = buffer->tail;

if (CIRC_CNT(head, tail, buffer->size) >= 1) {

        /* extract one item from the buffer */
        struct item *item = buffer[tail];

        consume_item(item);
```

```
		/* Finish reading descriptor before incrementing tail. */
		smp_store_release(buffer->tail,
				  (tail + 1) & (buffer->size - 1));
	}

	spin_unlock(&consumer_lock);
```

This will instruct the CPU to make sure the index is up to date before reading the new item, and then it shall make sure the CPU has finished reading the item before it writes the new tail pointer, which will erase the item.

Note the use of READ_ONCE() and smp_load_acquire() to read the opposition index. This prevents the compiler from discarding and reloading its cached value. This isn't strictly needed if you can be sure that the opposition index will _only_ be used the once. The smp_load_acquire() additionally forces the CPU to order against subsequent memory references. Similarly, smp_store_release() is used in both algorithms to write the thread's index. This documents the fact that we are writing to something that can be read concurrently, prevents the compiler from tearing the store, and enforces ordering against previous accesses.

Further reading

See also Documentation/memory-barriers.txt for a description of Linux's memory barrier facilities.