Dr. Dobb's Portal
The World of Software Development

Spin Buffers

If you are writing high-performance applications, you should consider using Spin Buffers that eliminate the need for synchronization.

By Prashanth Hirematada, Dr. Dobb's Journal
Jun 08, 2007
URL:http://www.ddj.com/dept/architect/199902669

Prashanth is the chief architect for Gamantra. He can be contacted at prash@gamantra.com.

Given any software, there is always a need for sharing resources or passing objects between two or more modules. When two threads are exchanging data as in Figure 1, the "producer" thread puts the resource into a common, shared area (a "shared FIFO buffer"), and the "consumer" thread takes the resource out of the shared area.

Figure 1: Two threads are exchanging data.

However, producer-consumer patterns are often performance bottlenecks. Why? Because one thread has to lock the other one out when both are accessing the shared resource area. This causes the operating system to put one thread on the wait list, while the other accesses the shared area. Of course, when this code becomes an application's hot spot (a common situation), there are alternatives. One approach is to minimize synchronization as much as possible. But no matter how small you make the synchronization functionality, the problem remains and is amplified when the data/resource exchange happens a thousand times a second. Consequently, you should consider using Spin buffers if you are writing high-performance applications because they eliminate the need for synchronization. They don't even need to employ low-level atomic instructions (such as Compare & Swap) found in advanced processors.

Ring Buffers

The most common way to implement the shared buffer is a Ring buffer (Listing One) with a certain size (*SIZE*). With Ring buffers, you maintain two pointers—one to read (*m_rptr*) and one to write (*m_wptr*). The producer inserts a new item into the buffer at *m_wptr*, then increments it by 1; for instance, *(m_wptr+1)%size*. Similarly, the

consumer inserts a read from *m_rptr*, then increments it by 1. If the special condition *m_rptr == m_wptr* is True, the buffer is empty.

One thing you should avoid is letting the read pointer get ahead of the write pointer. To ensure this doesn't happen, you could maintain the current size (*m_size*) so that the read pointer is not incremented. In this case, *m_size* is 0 (buffer is empty, nothing to read) and the write pointer should not be incremented if *m_size* is equal to the buffer size (buffer is full, write fails).

Let's package all this into a class and provide access to methods *put* and *get*. Note that I try to have the smallest amount of synchronized code; namely, the method *updateSize()*. In most cases, this is okay because it is straightforward. But if you are producing a high-performance application, you need to take a closer look at the producer-consumer patterns in the applications. For example, I work on game server engines that require more than 100,000 messages be processed every second, then broadcast back to the clients.

Spin Buffers

Spin buffers contain three internal "ordered" buffers (see Figure 2), but expose a simple API similar to Ring buffers. The buffers are ordered as 0, 1, and 2. The buffer of 0 is 1, then the buffer of 1 is 2, and finally, the buffer of 2 is 0. Two pointers are maintained—one that points to a buffer that a reader reads from, and another that a writer writes to. Referring to Figure 2, at any time there can be an assigned read buffer (*R*), an assigned write buffer (*w*), and free one (*f*). The buffers can be implemented as fixed-sized arrays.

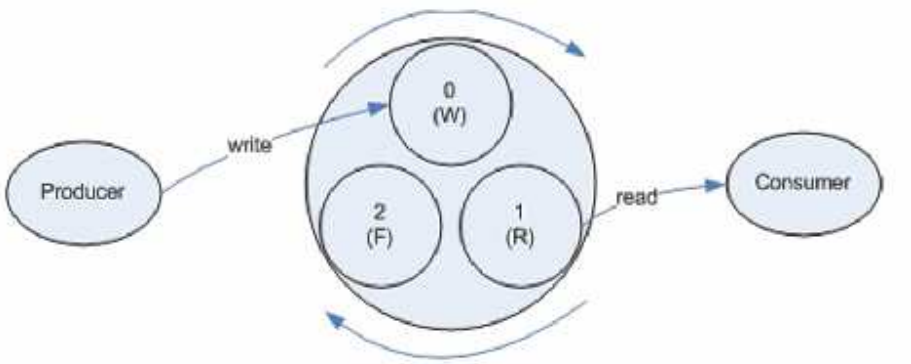

Figure 2: Spin buffers contain three internal "ordered" buffers.

The *put* method is as follows:

1. Put the item into the write buffer.
2. If the next buffer is free, make the free buffer become the write buffer, and the current write buffer becomes free.

The *get* method is as follows:

1. Read the item from the read buffer.
2. If the current read buffer is empty and the next buffer is free, make the next buffer the read buffer, and the current read buffer becomes free.

Listing Two is the Spin buffer implementation in Java (it should be straightforward to write it in other programming languages).

Caveats

There are things to watch out for when implementing Spin buffers. For instance, the reader and writer on the Spin buffer must be called all the time; otherwise, you could run into a situation where exiting items in the buffer cannot be retrieved by the reader. This happens when writers have nothing to write; if *put* is not called, the items in the current write buffer can never be read by readers. One way around this is to call *put* with null, which forces the write buffer to advance to the next buffer, thereby letting the reader thread access the items in the buffer.

Spin buffers are also sensitive to impedance mismatch, which means slow readers slow down the writer and visa versa. The performance peaks when reads/writes are called at matching frequency.

The current implementation works when there is only one reader and one writer. If multiple readers and writers are needed, the *put* method must be synchronized; similarly, if there are multiple writer threads, the *put* method must be synchronized. However, the code for reading/writing need not be synchronized.

Performance Analysis

I did the comparative performance analysis on these three implementations:

- Ring Buffer.
- Java *ConcurrentLinkedQueue* (CLQ) (based on "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms," by Maged M. Michael and Michael L. Scott; www .research.ibm.com/people/m/michael/podc-1996.pdf).
- Spin Buffer.

I conducted the performance tests on a hyperthreaded 3.0-GHz Pentium machine with 1 MB of RAM. I ran the tests multiple times for a minimum period of one hour on each of the different algorithm implementations. Table 1 presents the results. (In Table 1, I use the term "bandwidth" to describe how fast a buffer enables the data transfer from producer to consumer. The higher the bandwidth of the shared buffer implementation, the higher the number of items that can be transferred through—and the better the overall performance of the applications.) The performance test code is available www.ddj.com/code/.

Implementation	Ring buffer	CLQ	Spin buffer
Bandwidth (millions/sec)	1.17	1.77	5.05

Table 1: Test results.

Conclusion

Spin buffers are simple to implement, expose a straightforward API, and don't require special hardware or advanced processors. In spite of their shortcomings, the performance gains that Spin buffers deliver make an ideal choice for a wide range of applications such as game server engines, graphics, networking, and memory managers, among other high-performance applications.