




Single Producer – Multiple Consumers Ring Buffer Data Distribution System with Memory Management

Mariusz Orlikowski^(✉) 

Department of Microelectronics and Computer Science,
Lodz University of Technology, Lodz, Poland
mariuszo@dmcs.pl

Abstract. The paper presents the parallel data processing system for data acquisition systems. A solution utilizes on ring buffer data storage place and dedicated memory management class, to be able to provide efficiently a requested memory area and release it. An operation is based on independently working single producer, a number of consumers and release tasks. They are intended to allocate memory buffer and acquire the data, process them and finally release the buffer. To organize such a sequence of the operations a new synchronization object is proposed. All the components allows the system operation with zero copy and lock-free data operation except the tasks synchronization to save CPU processing power while waiting for previous operation completion. The system was tested and some performance results are presented. The solution is currently intended to work in multithreading applications, however the design on robust interprocess operation is ongoing. The presented work includes solutions suitable for the future extension.

Keywords: Ring buffer · Memory management · Lock-free · Zero copy · Synchronization object

1 Introduction

In the world of data acquisition systems the designers often face the situation that they need to handle high throughput data streams. The data first need to be acquired from the source to CPU, then they need to be processed and finally the results usually are sent away. For contemporary systems it is not so hard to saturate the system with available devices (e.g. ADCs, video cameras). They can generate extensive data stream for single processing unit. The multiple data sources connected to single CPU are not so rare as contemporary computers offering increasing performance are able to process high throughput data. In such a case the data stream can be close to the processing limits of the single system. In such a case a proper data handling and distribution among processing tasks is an issue.

The designed solution is intended to be used in high performance diagnostics systems where different processing algorithms need to be applied to the acquired data. Some processing parts, data archiving or pushing the data out may be enabled or

disabled when needed at any time. Additionally, the ring buffer approach is used to store required amount of data to work smoothly without data loss even when the tasks needs occasionally more time for the processing. The software implements single producer to many consumers data distribution system with zero copy approach implementing dedicated lock-free objects.

The existing solutions, e.g. [1–6], implement ring buffer approach, but in most cases they are intended to work with fixed size data, also called messages. The solutions have also other restrictions e.g. number of ring buffer elements must be a power of 2 [5, 6]. It simplifies the data management part and makes possible to optimize the latency and number of possible operations per second. Additionally to achieve the best results they often use active monitoring sacrificing CPU processing time. The features may not fit to many data acquisition and processing application requirement where the system can work with variable size data buffers and balanced performance vs. CPU utilization. The presented implementation is intended to work as multithreading application, but great part of the code was designed to make possible extending it for multiprocess operation in the future. The implementation includes some programming techniques to optimize data management time and minimize latency. The code was prepared for ongoing large scale high energy physics projects as ITER tokamak built in France and its diagnostics systems thus the implementation is done in C++ 11 for Linux 64-bit platforms to operate within CODAC based systems [7].

2 One-to-Many Data Distribution System Proposal

The proposed system [8] was designed to work with single writer (producer) feeding data into the system and many readers (consumers) which can operate onto the same data in parallel threads. The readers can attach to the system and detach at any time without interrupting the system operation.

2.1 System Operation Overview

The idea of the operation is derived from the classical ring buffer approach holding the data coming from the source. The acquisition task first reserves following regions of the buffer, which is filled with the data. When the data region is no longer needed, it is released and may be reused again. Memory management data holding information about reserved regions are placed in another structure which also can also be considered as a ring buffer (or a vector indexed by a modulo counter). The information about any specific memory region can be identified by a single number, the vector index.

A separate synchronization object is used to organize the task communication. It is also designed as ring buffer where each element holds an individual synchronization object assigned to memory region when data acquisition is completed. The synchronization objects are used to control the specific task access to given memory area for write, read or release.

The data management and synchronization objects with their data dependency and operation idea are presented in Fig. 1.

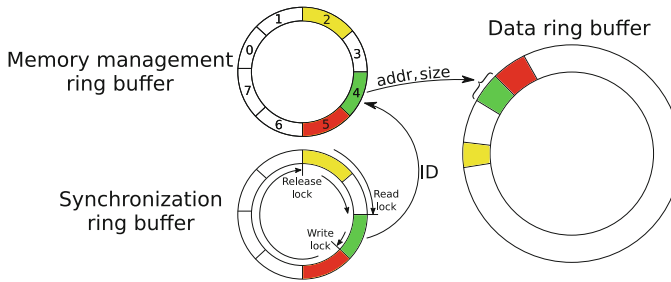


Fig. 1. The system operation concept.

The solution has always two tasks running: the acquisition thread as a part of user application and release thread run in background. Additionally, during the runtime, reader tasks can be started or stopped at any time to access the data. The synchronization objects are responsible for organizing proper sequence of operations on the data, i.e. the new data region first must be written, then may be read if any reader exists, and finally must be released, when no reads are in progress.

For the acquisition and reading task the system interface provides the calls for memory region allocation and committing written data (used in acquisition loop), and the calls for getting the following data regions for reading and committing when reading is completed (used in readers loop). The release task is an internal one and follows the readers and writer freeing memory regions when they commit. It is started on system initialization and operates till system stops.

To work with the ring buffers two classes **Producer** and **Consumer** have been designed to be used in the application. The provided methods create new or attach to the existing ring buffers and performs all necessary operations related to memory reservation, committing and receiving data and graceful cleaning up on exit.

Memory Management. A memory manager is responsible for providing the ring buffer memory areas for the producer and releasing them when they are no longer used. It was designed for (but not limited to) the data acquisition use case.

A standard dynamic memory allocation process is not optimal in terms of its usage in high performance data acquisition applications. The issues as built-in locks for thread safety, included metadata and aligned allocation, memory fragmentation, extra lazy allocation algorithms etc. may affect the memory usage, data processing performance and smoothness of operation. The problems can be avoided by using no dynamic allocation at data acquisition and processing time. All the data structures and buffers need to be allocated before any acquisition starts, and they are used sequentially to acquire and process data. A sequential operation allows designing a specialized memory management algorithm with free and allocated regions non-fragmented.

A system goes beyond the most common approach with fixed size buffers. They are easier to implement and faster to manage, but not flexible. The designed memory management allows varying size memory region reservation with byte size resolution.

The presented solution implements also zero copy and lock free ideas. It allows improving overall system performance. The producer and consumers use the same

memory areas for acquiring and processing the data. The locks were replaced with atomic operations without affecting the thread-safety of the operations.

The memory manager operates on two shared memory regions. The first one called later *memory management data* keeps the information about the reserved data regions. The data are managed using an operation on a set of atomic objects identifying current memory layout. The second region is a ring buffer memory, used directly to store the and exchange data.

The acquired data storage space is organized as a ring buffer structure. It means that acquired data regions are written one after another. When the writes reach the buffer tail, the writing starts from its beginning again. When a ring buffer memory is allocated in standard way (single memory area), we can face the situation that remaining tail of the memory is too small for current request. It needs to be left unused or the memory data must be provided as two separate regions. To avoid this problem a special memory allocation routine has been used. It uses a feature of MMU which allows to map given physical memory segment in virtual user space more than once. From application perspective the available memory area has doubled size, but writes beyond the first part have the corresponding updates in the first one. The described memory layout greatly simplifies memory management routines, never generates unused tail and allows lock-free memory management implementation.

The memory management routines identify ring buffer memory regions using abstract memory addressing in range $[0, \text{ring buffer size} - 1]$ and its size. To get the actual address of the specific memory area, the abstract address need to be added to a process specific address of the ring buffer. The approach makes the memory representation process-independent.

The information about reserved memory areas is maintained in memory management data. They are organized as a vector of structures (called later *data slots*) identifying each memory region in use. Each element keeps the information about abstract address, size and a slot index also used as a status flag. A count of data slots is a parameter of the initialization call and it limits the number of concurrently allocated areas. It needs to be estimated for specific data acquisition use case using the information how many data sets need to be buffered. The vector structure also simplifies identification of the memory region by index of the data slot in the table.

To manage the ring buffer, i.e. allocate and release data regions, a separate class has been designed. The allocation and release calls operate on dedicated object representing the memory region. The object provides methods to get ring buffer memory region address, the size, and provides the error in case of allocation failure.

To make memory management code suitable for other applications the algorithm can also operate in non-ordered release sequence with full thread-safety for allocation and release operations. Anyway, the best performance is possible where the allocation and release sequences match.

The memory management is done using atomic operations on the data representing the free data region with the first empty data slot index and the first to release data block information. Additionally, a semaphore is used to control the access to limited number of data slots with. Allocation and release operations alter the counter accordingly additionally providing synchronization for data slot objects.

The `alloc()` call (Fig. 2) tries first to reserve single data slot trying to decrement the data slots access semaphore. Next the free memory size is decreased if sufficient. The

last step is atomic increase of data slot index and first free abstract address to adjust for next allocation and returning the object representing currently allocated memory region. Managing the address together with the slot index makes sure that the data slot ring buffer holds always data regions ordered.

A free() call (Fig. 3) first marks the provided data block as it is requested to release. The flag allows release completion in following calls of free() if it cannot be done in current one. Then the first to release data block information is altered, but only if they match current data to release. A mismatch indicates that the current release is performed in not-ordered sequence or other free operation is in progress so the call exits and release is deferred. Next the data slot flag is set empty, free memory size is increased and the slots access semaphore is incremented. To prevent race with other concurrent free() calls which may increase the semaphore before current data slot is marked empty, a special flag is set and cleared so only one thread can complete the operation. The procedure repeats to complete all postponed releases.

```

DataBlock alloc(size, timeoutns){
    if (sem_wait(slotsSem, timeoutns)!=0)
        return status_timeout;
    free = freeSize;
    do {
        if (free < size){
            sem_post(slotsSem);
            return status_outofmemory;
        }
    } while (!cmp_xchg(freeSize, free, free-size));
    data = allocData;
    do {
        newData.ID = (data.ID + 1) % slotsNum;
        newData.addr = (data.addr + size) % rbSize;
    } while (!cmp_xchg(allocData, data, newData));
    dataSlots[data.ID].addr = data.addr;
    dataSlots[data.ID].size = size;
    return DataBlock(dataSlots[data.ID], rbBaseAddr);
}

```

Fig. 2. Memory allocation pseudocode (atomic operations marked bold).

```

free(ID) {
    dataSlots[ID].stat = torelease;
    do {
        reqData.ID = ID;
        reqData.addr = dataSlots[ID].addr
        newData.ID = (ID + 1) % slotsNum;
        newData.addr = (dataSlots[ID].addr + dataSlots[ID].size) % rbSize;
        newData.protect = true;
        if (!cmp_xchg(freeData, reqData, newdata)) break;
        dataSlots[ID].stat = empty;
        freeData.protect = false;
        freeSize += dataSlot.size;
        sem_post(slotsSem);
    } while (dataSlots[ID = newData.ID].stat == torelease);
}

```

Fig. 3. Memory release pseudocode (atomic operations marked bold).

Synchronization. To handle thread synchronization for the system a new lock object has been designed (called *wrflock*). The lock operation consists of set 3 steps for each type of the operation to be performed (write, read or release/free):

- *wacquire()*, *racquire()*, *facquire()* – the calls initiates the given type of operation and the lock will be able to transit to the specific state. The *racquire()* operation is blocking when the lock has acquired been for release and write to prevent starvation problem.
- *wwait()*, *rwait()*, *fwait()* – waits when given type of operation can be done. Read and write can block waiting for its turn. A *fwait()* uses wait yield approach, to prevent the slowest consumer *futex wake()* calls, which additionally degrade its speed.
- *wrelease()*, *rrelease()*, *frelease()* – the calls declares the end of the given operation and allows to unblock the next allowable operation in sequence.

The *wrflock* is based internally on the a 64-bit integer value containing a set of flags and counters to trace number of writer, reader and release tasks operating on the lock. For its internal operations *futex* and its wait/wake operations are used. The integer bitwise flags are used to keep the information about current state of the lock and its next allowed state. Counters for write and free operations are 1-bit thus only one type of each lock can be successfully hold. For read operation the counter is 16-bit wide thus up to 65535 concurrent readers are able to lock the object. A pseudocode for a set of operations is presented in Fig. 4.

Considering the above information *wrflock* operations can be also considered as a state machine. The possible states and transitions are shown in Fig. 5. The transitions are made in acquire and release (free) calls, where the related read/write/free counters are incremented and decremented accordingly.

```

facquire (lock) {
    data = lock.data;
    do {
        newdata = data | FR_NUM;
        if (data & NEXT_RDFR)
            newdata = newdata ^ (NEXT_RDFR | CURR_FR);
    } while (!cmp_xchg(lock.data, data, newdata));
}

fwait (lock) {
    while (!lock.data & CURR_FR)
        yield();
}

frelease (lock) {
    data = lock.data;
    do {
        newdata = data & ~(FR_NUM | CURR_FR);
        if (data & WR_NUM)
            newdata |= CURR_WR;
        else
            newdata |= NEXT_WR;
    } while (!cmp_xchg(lock.data, data, newdata));
    if (newdata & CURR_WR)
        futex_wake(lock.data);
}

```

Fig. 4. A pseudocode for *facquire*, *fwait* and *frelease* calls (atomic operations marked bold).

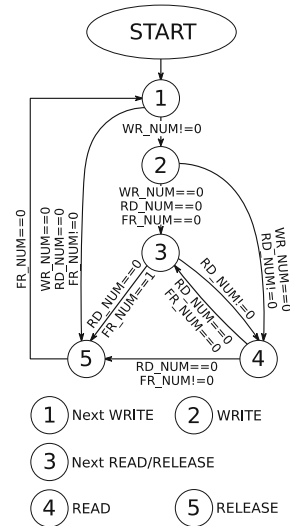


Fig. 5. State machine of *wrflock*

User Interface. A user of the system needs to use 3 designed classes:

- **Producer** – provides a set of methods to create data structures, reserve memory region and commit data when the writing operation is completed. It runs release thread dedicated to free the allocated data regions when they are no longer used.
- **Consumer** – provides a set of methods to attach to existing producer data structures, get the memory region for reading and commit when the work is completed.
- **DataBlock** – a class used to access specific data region.

A producer and consumer operation example has been presented in Fig. 6.

<pre> Producer p("test"); DataBlock b; p.create(1000000, 100); p.init(); while (!stop) { while (!p.allocateDataBlock(b, 123)); // do data acquisition // of 123 bytes // to address *b p.commit(b); } p.done(); </pre>	<pre> Consumer c("test"); DataBlock b; c.attach(); c.init(); do (!detach){ c.getDataBlock(b); if (b.isStop()) // producer break; // exited // read b.size() data at *b c.commit(b); } c.done(); </pre>
--	--

Fig. 6. Example code of the producer task (left) and consumer task (right).

3 Performance Tests

The system performance has been estimated on Nehalem dual Quad Core 2.5 GHz CPU under Linux RedHat 7.4 using <chrono> library high resolution clock. The tests shows minimal and average time of operations and the time where 99% of operation are completed. The statistics are done from 10000000 loop iterations.

3.1 WRF Lock Performance

A similar locking object available in GNUC has been compared to WRF lock assuming that no locking takes place, i.e. the lock does not block at all. The tests show the overhead of the internal code needed to check, verify and set the lock. It gives an information how complex, thus how efficient is the locking and unlocking algorithm. The results (Table 1) shows how much time specific lock calls require to complete.

The tests show that the performance of the designed locking object faster than remaining ones. In Linux implementation it takes relatively small amount of the memory, so it can be used to create large ring buffer queues. It proves its usability in the system in place of other ones.

A performance measures including latency when the lock waits blocked has not been shown. As the objects use internally the same futex calls, results are similar.

Table 1. Performance comparison of selected locking objects (non-blocking case).

Lock type	Size [B]	Operation	Lowest time [ns]	Average time [ns]	99% Less then [ns]
mutex_t	40	lock	6	12	82
		unlock	7	11	13
sem_t	32	post	9	13	81
		wait	10	10	10
rwlock_t	56	rlock/wlock	13	20	22
		unlock	14	18	19
wrflock_t (w/r/f)	8	acquire	8/19/9	11/21/12	8/20/11
		wait	1/1/1	3/4/3	2/4/2
		release	9/9/9	13/11/14	12/11/12

3.2 Memory Manager Performance

The memory manager performance is compared to standard malloc()/free(). A single test reserves 100 bytes of memory block including allocation success check and release it. For GNUC allocation calls additional tests were performed to test memory locking performance which may be needed to prevent lazy allocation and memory swapping. A memory fragmentation influence was not tested (Table 2).

Table 2. Performance comparison of memory allocation calls.

		Lowest time [ns]	Average time [ns]	99% Less then [ns]
GNUC	alloc	34	58	72
	free	<1	5	5
	mlock	3000	3200	3600
	munlock	730	1100	1100
Memory manager (semi MT)	alloc	31	35	36
	free	32	37	100
Memory manager (fully MT)	alloc	38	41	41
	free	45	49	114

The GNUC allocation algorithm works about 2 times faster than implemented memory manager if we measure overall alloc/free cycle. A big overhead of standard calls should be expected if memory locking or its initialization is additionally done. As profiler test shown, the bigger call time for the ring buffer memory manager comes from atomic operations. Slightly better results are possible for the proposed solution if some of atomic objects are eliminated keeping only independent operation of alloc() and free() from single separate threads.

Note that fully multithreaded memory management solution was designed to be used in future extension of the system as well as other applications, where the memory can be allocated and released from many threads and processes concurrently.

3.3 Overall System Performance

Each tests procedure has 3 steps: integrity check (to verify if sent and received data match), latency measurements (using high resolution clock) and operations per second estimation (with no data operations). Each step executes data allocation (100 bytes buffer) and buffer commit with various number of consumers running. A results with default scheduling policy are provided in Table 3. The same test has been executed with FIFO scheduling algorithm. The results are provided in Table 4.

Table 3. Overall system performance for selected number of consumers (default scheduler).

Number of consumers	1	2	3	4	5
Write operation performance [10^6 op/s]	2.3	2.1	1.8	1.5	1.6
Lowest producer-consumer latency time [ns]	130	128	133	132	144
Average producer-consumer latency time [ns]	3400	2000	5500	4600	4700
99% Producer-consumer latency time less than [ns]	2500	1800	3100	2500	2700

Table 4. Overall system performance for selected number of consumers (FIFO scheduler).

Number of consumers	1	2	3	4	5
Write operation performance [10^6 op/s]	3.4	3.1	2.8	2.5	2.3
Lowest producer-consumer latency time [ns]	159	126	131	154	202
Average producer-consumer latency time [ns]	3700	2000	2100	2500	2600
99% producer-consumer latency time less than [ns]	1100	2200	1200	1100	1400

A similar tests has been performed for port of LMAX-Disruptor to C++ [6] using “OneToThreeWorkerPoolThroughputTest” performance test. The test was modified to use similar synchronization mechanism (BlockingWaitStrategy) and it was run for the same number of consumers 1 to 5. The second set of tests includes additional realloc (100) call in producer code to emulate behavior when dynamic allocation takes place. The results are shown in Table 5.

Table 5. Port of LMAX-Disruptor to C++ performance.

Number of consumers	1	2	3	4	5
Write operation performance [10^6 op/s]	3.3	6.5*	7.2	3.5	4.3
Write operation performance with realloc() [10^6 op/s]	2.3	1.6	2.0	2.5**	2.2***

Significantly varying results: * from 1.9 to 11.2, ** from 0.7 to 4.3, *** from 0.9 to 3.8

Comparing a performance of presented system to other ring buffer approaches published in [2, 5], the results show that it is 8–10 times slower. However the solutions are message based (fixed size items), do not consider the memory management problem and uses non-blocking synchronization mechanisms.

The results for C++ implementation of LMAX-Disruptor [6] show that its performance for basic approach with no per-message data allocation the presented work with FIFO scheduler is up to three times slower. However if the allocation routines are included, the results are in most cases gives faster operation. The tests also show unpredictable behavior of the Disruptor as performance tests significantly vary from run to run in some configurations. As the Disruptor does not allow direct per-message memory allocation in producer which can be freed with last consumer reading, the same algorithm cannot be tested.

4 Conclusions and Future Work

The presented solution is applicable for data acquisition and processing systems. The performance of the data management and control makes it suitable in high performance systems. The proposed data management solution for ring buffer is suitable for using in other systems. A designed synchronization object is intended to support the data ring structures for single producer-many consumers approach with separate releasing task. Nevertheless simple structure allows its easy adaptation for other cases.

The tests show that the performance may be tuned to get better results. Thus the software need to be investigated against using other scheduling algorithms, binding threads to specific CPUs to optimize cache memory usage etc. It can also be considered to implement similar to Disruptor synchronization mechanisms (SpinWait or BusyWait strategies) in the wrflock.

A current implementation includes partial support for system wide data sharing. A next major step is extending the system to operate in interprocess environment. It is also planned to add support for more than one producer.

The system should also to compared to other distributed shared memory solutions, e.g. [9]. It may lead to creation of portable, platform-independent design in the future.

References

1. Feldman, S., Dechev, D.: A wait-free multi-producer multi-consumer ring buffer. *SIGAPP Appl. Comput. Rev.* **15**(3), 59–71 (2015)
2. Krizhanovsky, A.: Lock-free multi-producer multi-consumer queue on ring buffer. *Linux J.* **2013**(228), 4 (2013)
3. Inglés, R., Orlikowski, M., Napieralski, M.: A C++ shared-memory ring-buffer framework for large-scale data acquisition systems. In: 24th International Conference on Mixed Design of Integrated Circuits and Systems, Bydgoszcz, pp. 161–166 (2017)
4. Inglés, R., Orlikowski, M., Napieralski, A.: A C++ shared-memory IPC framework for high-throughput data acquisition systems. *Int. J. Microelectron. Comput. Sci.* **8**(2), 43–49 (2017)
5. LMAX-Exchange/disruptor Wiki. <https://github.com/LMAX-Exchange/disruptor/wiki>. Accessed 23 Oct 2019

6. Port of LMAX-Disruptor to C++. <https://github.com/Abc-Arbitrage/Disruptor-cpp>. Accessed 27 Oct 2019
7. CODAC. <https://www.iter.org/mach/codac>. Accessed 30 Dec 2019
8. Sources of the Single Producer – Multiple Consumers Ring Buffer Data Distribution System. https://gitlab.dmcs.pl/mariuszo/ringbuffer_daq.git. Accessed 29 Nov 2019
9. Geva, M., Wiseman, Y.: Distributed shared memory integration. In: 2007 IEEE International Conference on Information Reuse and Integration, Las Vegas, pp. 146–151 (2007)