# Reduced Circular Slice Buffer

**Antoni Paavola**

2022 Laurea

**Laurea University of Applied Sciences**

# Reduced Circular Slice Buffer

Antoni Paavola

Bachelor's Degree in Business
Information Technology
Thesis
4, 2022

**Laurea University of Applied Sciences** **Abstract**
Degree Programme in Business Information Technology
Bachelor

Antoni Paavola

**Reduced Circular Slice Buffer**

Year 2022 Number of pages 31

This paper is a study describing the combining of a circular buffer and a slice to minimize the use of instance variables and to further increase reliance on compile-time information to reduce memory accesses. The paper undertakes an analysis on a theoretical as well as on a practical basis.

The study highlights the researcher's former experiment as a reference implementation which describes a performant method for designing circular buffer solutions for both construction and usage. It involves a modified technique for using the virtual memory system to guarantee buffer position optimizations in usage.

A concurrent circular buffer based on the guidance of the reference implementation was created, benchmarked, and compared with other concurrent buffers within an open-source community.

The results show that a circular buffer identical to a slice, which uses optimized custom techniques for circular buffering, can be achieved all while retaining the performance benefits of memory access based on the virtual memory system.

Keywords: Slice, Circular buffer, Ring buffer, Queue, Benchmark

Contents

# 1 Introduction

The usage of arrays, collections, queues, lists or buffers are essential in computer science. The idea of storing singular elements or types together carries a multitude of different names but are used with the same purpose. Considering how widespread these terms are it is likely that at least a few are familiar. With more complex requirements, it becomes ever more essential to provide greater efficiency for processing data and providing greater throughput from the hardware already present.

Slices are acquiring a greater importance as languages seek to enable lower-level handling of data and rely less on abstractions. To enable this change, the paper seeks to introduce a method of integrating buffers and slices to potentially combine their advantages.

This research takes a refreshed look on a specific variant of buffers, namely circular buffers, or ring buffers which have been created to have data loop around to optimize data processing. The paper intends to discuss and explain buffers in a wider sense rather than delve into implementation details. Each implementation should be unique which is why it is better to receive a more abstract idea rather than a copy of an existing one.

## 1.1 Purpose of the study

The research intends to create a modern version of a circular buffer, **a circular slice**. This research allows for any slice to potentially be a circular buffer. It can be made to be indistinguishable from a regular slice with language specific methods such as operator overloading or become a built-in compiler optimization.

The circular slice has to be a buffer in itself without need for storing the position or length of the underlying buffer in addition to it. Hence it should be a **reduced instance size circular buffer**.

Documentation (Chapter **Error! Reference source not found.**) of clear guidelines acquired from experience gained from the experimental implementation is necessary and hopefully serves as inspiration or instruction to any future research on improving the circular slice further.

Yet, the most important goal has always been to improve the throughput and overall performance of a circular buffer. By conforming as closely as possible with a slice, it has also been a goal to be as efficient as a slice.

Another purpose is to stray away from any older standard of implementing a buffer as much as possible to create something current and usable. The hope has been to remain intuitive and performant at the same time.

### 1.2 Research questions

The main research question the paper seeks to answer is whether it is possible to improve on current circular buffers and if so, using what means? Furthermore, how can a concurrent system be designed so that information can pass between the buffer reader and writer while avoiding delaying their respective tasks?

Secondly, is it possible to integrate a circular buffer with a slice and how? Is it possible to acquire the same form factor and similarly low-level access efficiency as a slice? What are the advantages and disadvantages of an implementation of a circular buffer that seeks to create a circular slice?

### 1.3 Research structure

Firstly, background information on circular buffers will be discussed. In the chapter emphasis is on aspects such as what are circular buffers and what are some existing ways of improving circular buffers. The chapter also gives some background to the choices taken in the implementation of the slice based circular buffer.

The next chapter can be thought as a guide to the reference implementation of a circular slice. In addition to the construction and usage of the reference implementation, optimizations and implementation options are discussed.

Finally, a benchmark comparing different implementations to the circular slice will be displayed in addition to the conclusions on the reference implementation.

## 2 Background

To get an overall sense of the topic itself, it is necessary to first understand the meaning of a circular buffer and in general what buffers are used for. Buffers are used in various areas of computer science. Buffers are generally used for network and general data related processing. For example, Han, Wald, Usher & al. (2020) presented a virtual frame buffer for walls of displays to render videos or Pirkle (2013) who devised a circular buffer to use with audio effects and filters.

A circular buffer is a buffer which can be addressed circularly which is unlike regular array access for example in C++. When creating an array as a buffer, its addressing is linear

meaning that offsetting a pointer pointing to the buffer's memory will always move linearly to the required position. (Pirkle 2013: 207-210.)

Linux kernel documentation (Circular Buffers) explains circular buffers to be of fixed and finite size. Access outside the bounds of a linear addressing buffer usually results in a crash, but with a circular buffer the accesses are wrapped to remain within the buffer bounds. A circular buffer's management of pointers also reduces the need to move data around inside the buffer as items in the buffer can be acquired through the pointer wrapping system. (Pirkle 2013: 207-210.)

There are multiple implementations of circular buffers in existence, but usually they consist of two pointers that point to the tail and head of data (Circular Buffers) as seen in Figure 1.

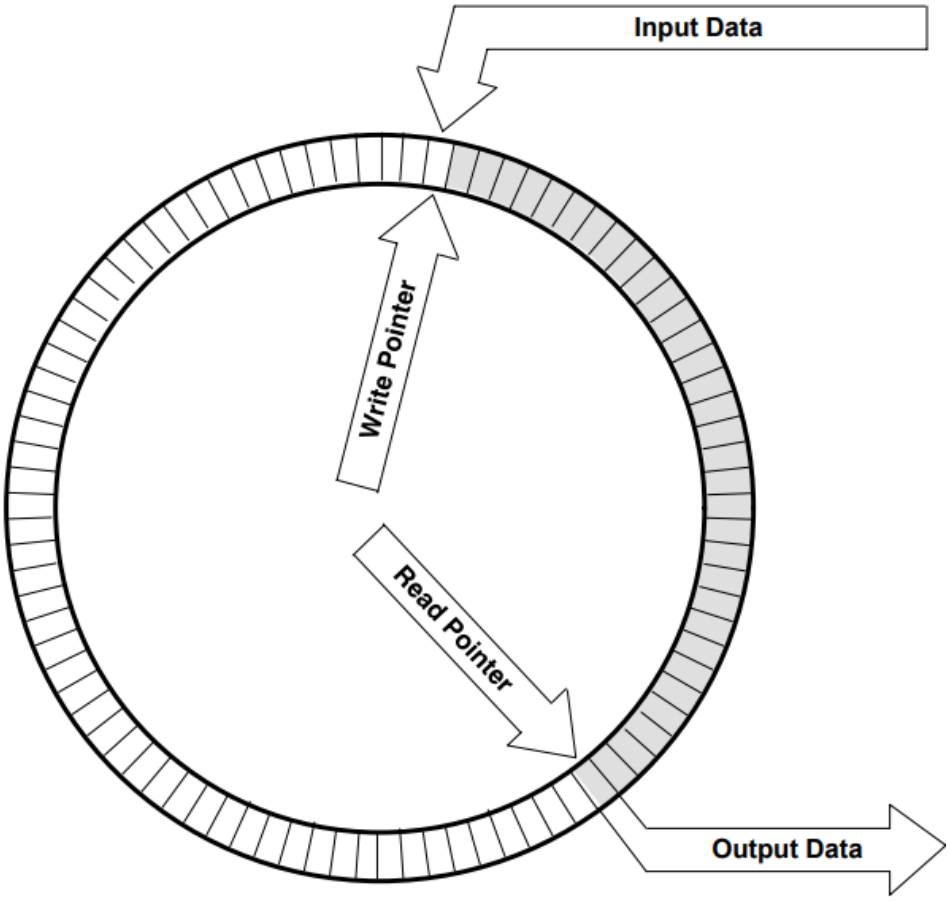


Figure 1: Circular Buffer (FIFO Architecture, Functions, and Applications 1999)

FIFO, First in First Out, indicates that the first item written is the first to be read (FIFO Architecture, Functions and Applications 1999). Circular buffers provide an efficient method

of implementing a FIFO queue (Ash 2012, Part I). The paper will henceforth continue to demonstrate circular buffers from a FIFO standpoint.

The buffer tail can be thought of as pointing to the start of the buffer where the read pointer is located, and the buffer head is pointing at the end, at the write pointer (FIFO Architecture, Functions, and Applications 1999). These can be referenced from Figure 1.

Once the buffer must be enlarged, the pointer pointing at the end is moved toward the positive direction and once items need to be removed the start pointer is also moved toward the positive direction. (Circular Buffers; linux/include/linux/circ_buf.h.)

There are other explanations in the kernel documentation in regard to the usual implementation of a circular buffer. For example, when the buffer is empty the end and start pointers are equal or when it is full, the end pointer is one less than the start pointer (see Circular Buffers; Ash 2012, Part II). However, there are other methods which are especially suitable for mirrored circular buffers which will be discussed in the “Mirroring” chapter.

## 2.1 Buffer management

Management of data locations is necessary to guarantee coherency. Buffer coherency may become an issue if the first portion of data resides at the end of the buffer and the second portion at the start of the buffer. If data is to be read in order, the reading must occur circularly by reading the start after the end.

To address memory circularly, it is possible to use a modulus operation. However, a common optimization in circular buffers is limiting the buffer size to a power of two to avoid using modulus division operations. This enables to use bitwise operations instead to calculate pointer distances when pointers wrap over the buffer border to wrap around to the start. (Circular Buffers; linux/include/linux/circ_buf.h.)

Circular buffers function well when writing is continuous and reading occurs soon after. They function well as a tool for communicating real-time information between threads. (Ash 2012, Part I.) In other words, it is best for short-term data.

When not using a circular addressing mode to access memory, for an example if using an ordinary array, it becomes necessary to copy data continuously. Copying is undesired as it can cause unnecessary overhead. This situation occurs when data is at the end of the buffer and needs to be moved to the start of the buffer for more data to be written. Circular buffers solve this issue as manual buffer management is no longer necessary. (Allen, Zucknick & Evans 2006.) See Figure 2.

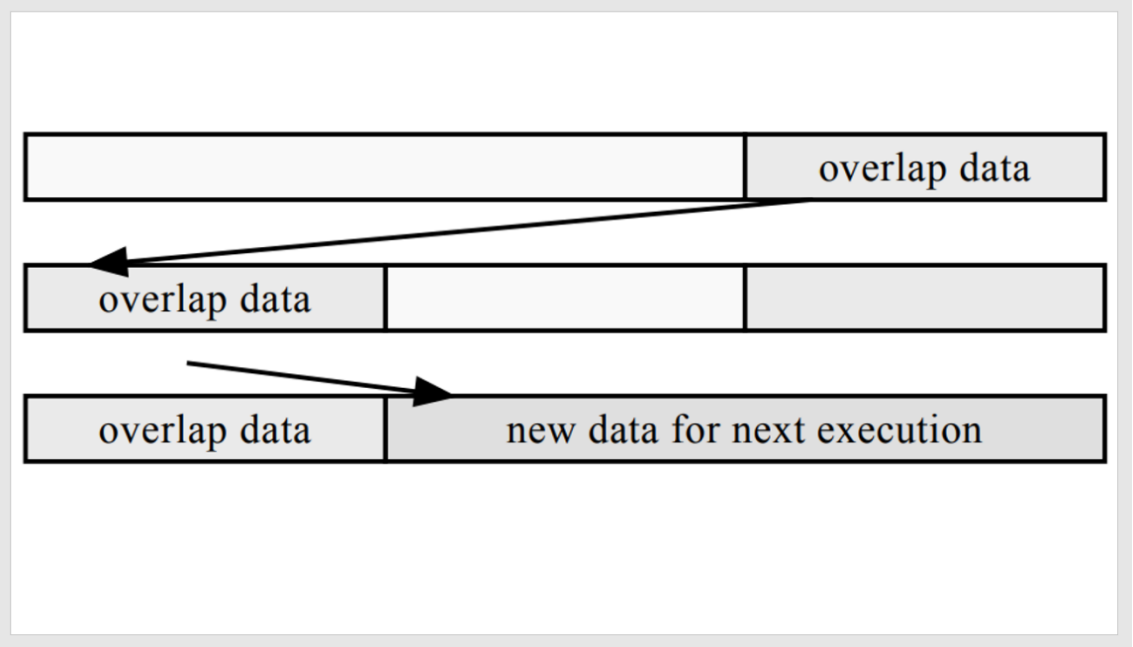


Figure 2: Data copying (Allen, Zucknick & Evans 2006)

## 2.2 Mirroring

In earlier chapters, circular addressing using a modulus operation was mentioned. It is, however, possible to achieve the same effect as pure modulo addressing using virtual memory management. It is a way to emulate circular buffers (Allen, Zucknick and Evans 2006).

In such a system, two contiguous sections of memory that contain identical data are created. The identicality of these sections are guaranteed by the CPU's virtual-to-physical address translation hardware by mapping both sections memory accesses to the same target, either a separate memory section or file, so that they are identical. (See for example Figure 3 from Allen, Zucknick & Evans 2006.)

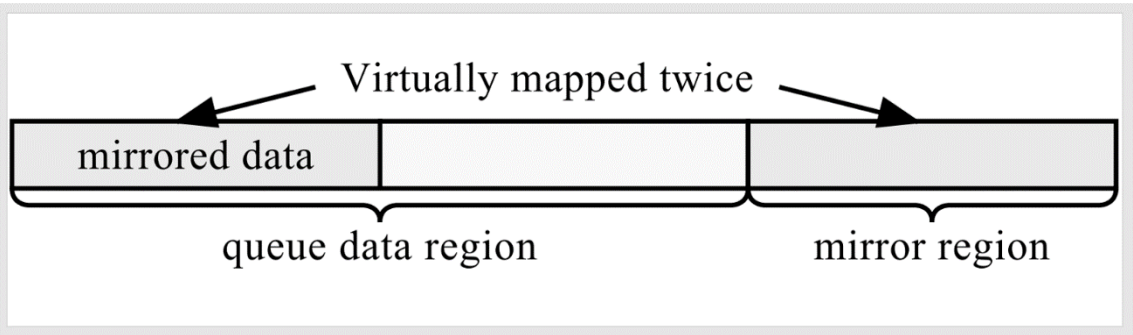


Figure 3: Mirroring (Allen, Zucknick and Evans 2006)

In a mirroring implementation, out of bounds access must be avoided using a modulus operation if needed after a pointer increment occurs. Using virtual memory mapping can

cause overhead, but it allows for copying to be avoided. Additionally, the ability to operate directly on memory can further reduce copying (Allen, Zucknick & Evans 2006).

The benefit of a mirrored circular buffer compared to non-mirrored is that it does not require wrapping of data. The data in a non-mirrored circular buffer may not be adjacent which means that passing pointers to the data can become difficult. This requires either combining data to a separate buffer by copying or separately processing non-adjacent memory. However, in a mirrored circular buffer non-adjacent memory is adjoined with a second copy of the memory and can there forth be used to pass data with just one pointer to previously non-adjacent memory when a single chunk of memory is required. (Ash 2012, Part I.)

Allen, Zucknick & Evans (2006) found in their tests on signal and image processing that using virtual memory management instead of manual memory copying had a meaningful and measurable difference. They also warned that their excessive use may result in less meaningful results due to overflow recovery measures. Several dozens, however, can be used without great negative impact.

An aspect of a virtual memory managed mirror buffer is that the buffer is page aligned (Allen, Zucknick & Evans 2006). A page of memory, henceforth referred to as page size, is the smallest amount of memory that can be used with a virtual memory system and is typically 4096 bytes. While the size of a memory page can be assumed, it is better to request the size of a single page from the system. (Ash 2012, Part I.)

Earlier it was said that typically a circular buffer is full when the end pointer is one less than the start pointer. However, this method becomes ill-favored when using mirrored circular buffers as the full potential of the buffer is not taken advantage of (Ash 2012, Part II).

Alternatively, to the typical method, fullness can also be expressed by the read pointer being exactly a buffer's maximum size less from the write pointer (Ash 2012, Part II). However, in such an implementation, either subtraction or addition is required to determine if the buffer is full.

Another way is to use a count instead of a write pointer, but then the write pointer would need to be derived using computation between the read pointer and count. In a concurrent setting, the read pointer and count can be inconsistent with each other from an outside perspective, leading to the resulting computation between them to be incorrect. (Ash 2012, Part II.)

According to Ash (2012), lockless thread safety becomes greatly more difficult or even completely impossible if using a read pointer and a count instead of both read and write pointers. However, this was achieved as seen in Chapter 3.2.1.

## 2.3 Concurrency

To optimize the use of the buffer further, it is necessary to read and write to it at the same time. This is where the Producer/Consumer or Bounded Buffer problem surfaces. It is necessary to design a system where multiple actors or threads can work together to manage reading and writing to the buffer. (Arpaci-Dusseau & Arpaci-Dusseau 2018.)

A consumer can be thought of as the one that reads the start of the buffer and removes the read or unnecessary data. A producer on the other hand will write more data to the buffer so that the consumer may acquire more data later when it needs some. (Arpaci-Dusseau & Arpaci-Dusseau 2018.)

There are multiple solutions to the consumer-producer problem. Concurrency can be achieved using locks, which block concurrent modifications allowing only one to modify at a time, but it is not able to utilize many processors efficiently. To minimize locking, it is possible to use message passing which can scale much better. (Wilhelmsson 2005.) However, message passing overhead can negatively affect the performance of the application (Morandi, Nanz & Meyer 2014: 99-114). The most typical way to achieve synchronization is using shared data structures (Wilhelmsson 2005).

Deciding which parts of memory to share and which to keep thread local can have implications on performance (ibid.). Some implementations avoid sharing memory between threads completely, which ends up negatively affecting performance due to communication overhead between threads (Schill, Nanz & Meyer 2013). According to Morandi, Nanz & Meyer (2014) typically about half of the work in producer-consumer programs consists of synchronous read accesses.

To achieve efficient concurrency on buffers, it is possible to use slices. Slices are portions of the original buffer but can also point at the whole buffer. Slices can be implemented as a pointer to memory and length. Slices can be set as read-only, or both read & modify. To modify a read-only slice, disconnecting it from the original array must be done by copying the content to a separate location in memory. (Schill, Nanz & Meyer 2013.)

Slices are often considerably smaller than the buffer itself, making it ideal for minimizing message passing overhead. Simply using a pointer in the consumer, will cause inefficiency in looping through the buffer as it is more efficient if the length is known. To get a length from a pointer, iteration of the buffer to search for a zero-byte string end marker is needed. (Fog 2021, 134 and 144.)

Efficient parallel array algorithm implementations require two types of access. Namely, parallel disjoint access and parallel read. In other words, a thread should be allowed to

mutate a part of the original array designated to it and multiple threads should be allowed to view a read-only portion of the original array in parallel. (Schill, Nanz & Meyer 2013: 37-48.)

Slices allow threads to work concurrently on their own sections of the original array. Slices can also be merged if the slices are sections of memory directly next to each other. In a concurrent setting, transferring of data is the most optimal when the underlying memory area is shared with multiple threads. (Ibid.)

## 2.4 Research scope

When looking at the source for the Linux kernel, it contains a circular buffer data structure. The structure contains a pointer to the location where the buffer's memory begins, and two integer offsets called head and tail (linux/include/linux/circ_buf.h). Assuming that a size of a pointer is 8 bytes, and an integer is 4 bytes, the total size of the structure would be 16 bytes.

As discussed previously, a slice is simply a pointer and a length (Schill, Nanz & Meyer 2013). A slice with a 4-byte unsigned integer as length is sufficient to contain 2^32 (4 294 967 296) elements or bytes if the elements type size is one byte. Using the previous assumptions on type sizes, the size of a slice would be 12 bytes with 4 additional bytes left over for other use if needed.

A circular buffer length is fixed, and finite as mentioned before, which means that 4-bytes can be sufficient for use as the length when it can be determined to be less ahead of time. Contrary to this if discussing normal slices designed for array use purposes, it is preferred to be using size_t as the slice length type, which can be an 8-byte unsigned integer on a 64-bit system, because the risk of overflow is non-existent due to the maximum length being 2^64 elements. (Fog 2021: 29-31.)

If a circular buffer were to be contained in a slice of 12 bytes in size, the remaining 4 bytes could be used for assisting for an example in concurrency as a mailbox for message passing. However, such a structure may have difficulties passing larger data such as 8-byte pointers in 64-bit systems but could be an interesting future topic for another paper.

While the research aims to minimize the amount of memory needed to manage a buffer, by removing the need to store the buffer's memory location, the paper only discusses the opportunity to reduce the length in theory. Meaning that there are still opportunities for future work on further reduction, as it is outside the scope of this research.

## 2.5 Problem specification

The buffer position cannot be saved separate to the read pointer and length, should the buffer be conformant to a slice. Meaning that there has to be a method of computing the position of the whole buffer from any position within the buffer.

It was stated that it is difficult to use a slice's count as a replacement for the write pointer in a concurrent setting. A method of computing the write pointer needs to be devised when the read pointer and count are inconsistent with each other.

A system to guarantee safe concurrency must be found in a way that would not require synchronization primitives with regular slice operations. Therefore, it should be possible to modify data within slices or the slice itself while having more data be written simultaneously without changing how a slice functions internally.

## 2.6 Methodology

The methodological design of this study is inspired by Pomberger et al. (1991) for the methods and tools necessary for supporting a prototyping-oriented approach. According to Pomberger et al. (1991, 59) experimentation is valuable for exploratory programming. Validation after each phase of development is necessary for the development phase's completion. Validation by itself is inadequate without experimentation mimicking its actual use environment. (Ibid., 1-2.)

As the objective of the study is to create a type containing only the variables needed for implementing a slice, a pointer and length, it is necessary to use an experimental approach to solve if and how saving the buffer structure can be made redundant. Knowing the buffer's overall structure can assist purposes such as ensuring access to the buffer is within bounds or releasing the buffer's memory back to the operating system.

The study relies not only on testing each development phase and designed component, but also the overall system success in its entirety using an open-source experimentation framework.

Figure 4 presents the different phases of software design according to Pomberger et al. (1991, 44). It illustrates correspondence with the paper's own phases of development and serves as a reference for the design and implementation cycle of this study. The reference implementation from Chapter **Error! Reference source not found.** can be seen to contain both System and Component Design. However, as mentioned in the introduction, the final implementation phase will not be documented in detail.

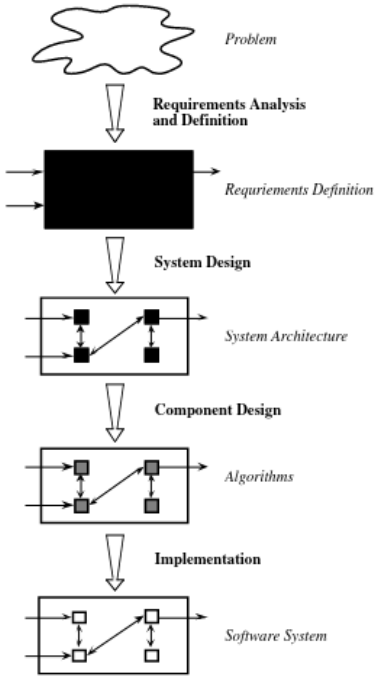


Figure 4: Stepwise refinement in the life cycle paradigm (Pomberger et al. 1991, 44)

Boar's 1983 definition of prototyping (cited in Pomberger et al. 1991) is that it is a strategy for accomplishing requirements definitions where user needs are extracted, presented, and refined by creating an implementation of the ultimate system quickly and in its working environment.

# 3 Reduced Circular Slice Buffer

This chapter portrays the reference implementation where different methods for implementation are discussed and occasional benefits to different options are presented. The actual implementation in Chapter 4 will be based on this reference implementation. It will show the performance and validify the reference implementation.

## 3.1 Single-threaded Buffer

The construction process of the reference implementation is remarkably similar to the regular virtual memory management system mentioned above in 2.2. However, with the difference of having guarantees of the position where the memory exists during the buffer's lifetime.

The buffer was constructed by creating an anonymous non-persistent file or memory object and mapping it into shared memory twice. While it may be called a file, it is not persistent, meaning it does not reside on disk, but instead in memory. The memory object mapping

consists of creating two contiguous sections of memory that each are set to contain the content of the object.

The size of the memory sections is a power of two and a multiple of page size. As mentioned in 2.2, page size depends on underlying system but is most commonly 4096 bytes. On Windows allocation granularity has to be used as a multiple instead of page size as it is required by the system.

The circular slice buffer works on the assumption that the positions of the two pages are partially known. When looking at the page size of 4096 (Binary 0001 0000 0000 0000), we notice that in an implementation of a circular slice buffer, the same bit in the start of the first page position is always 0 and in the second page it is always 1. See Figure 5.

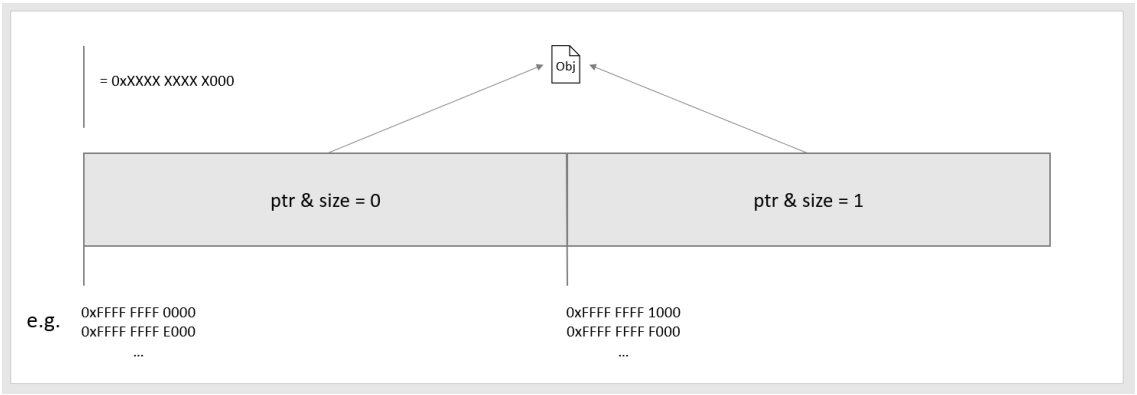


Figure 5: Construction

Implementations can reverse the starting bit as well, so that the first page size bit is 1 and second is 0. There are slight differences between these two implementation methods which show in both construction and usage.

In the construction process, three sections of memory are needed. In a system with a page size of 4096 the amount of memory requested would be 3 x 4096 x M. M being the multiplier in case more than 4096 of usable bytes are needed in the buffer which will be 1 for the sake of simplicity.

First reserving the memory is required. Once the amount of memory is reserved, it can be divided into three sections. When looking at the positions of data within each section, it is

noticeable that the bit corresponding to the section's size 4096 (0001 0000 0000 0000) differs depending on which section the data is located.

The reserving will return a group of three sections, with their section size bits being either A) 0, 1, 0 or B) 1, 0, 1. In an implementation where the first section should have the size bit unset, the last page is unreserved from result A or first from result B. Correspondingly, if the first section's size bit should be set, the first page should be unreserved from A and last from B. See Figure 6.

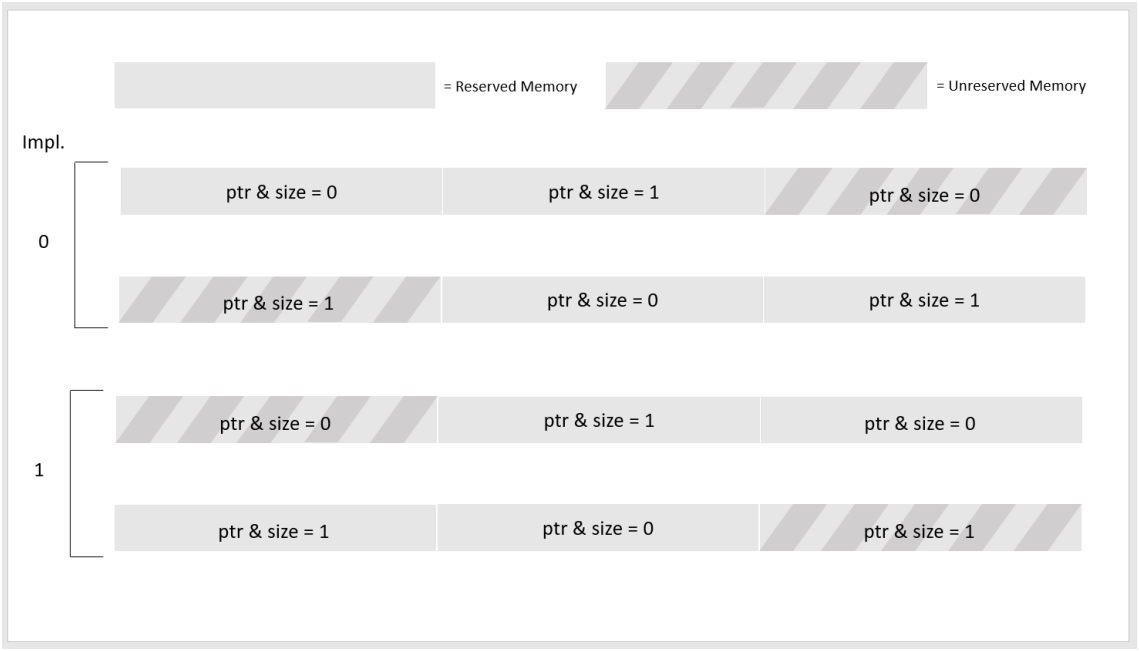


Figure 6: Reserve

This method guarantees the bit in question to be as required, however, it may take longer to construct the buffer as more memory is required. After the two sections are reserved, they are then mapped to the memory object so that the content of both sections is identical.

While the advantage of a predictable memory position is not apparent at first, a closer inspection shows that there is no longer a need to maintain a reference to the start of the buffer as long as a slice or pointer to any position within it is available. The buffer is identical to a slice and may be implemented the same way as a slice would be implemented. This means that it consists of a pointer and a length.

In the construction phase of the buffer, the memory object was created, but to avoid maintaining a reference to it, it may be closed after the two sections of memory are mapped.

This will automatically invalidate the object itself only after all references to it are closed. The remaining references are the two mapped memory sections, meaning that the memory object will be invalidated once both of the memory sections are closed.

At the end of the construction the only variables needed are a pointer and a length. Length being the count of how many items of data are stored within the buffer. A single item being a value of the type that the buffer is meant to hold. The maximum count that the buffer can hold can be calculated by dividing a single section's size with the byte size of the type that is ought to be held in the buffer.

### 3.1.1 Usage

The complex construction of the buffer allows for a simplified usage implementation. In this study, by usage is meant the managing of the data and variables inside the buffer.

The buffer differs from a slice only in expanding and destruction. Expanding the slice is also often called concatenation, adding, or joining to the slice. Expansion is much simpler with the partial knowledge of the position of the first page. In this chapter, a closer look is given to the two differences of a slice in its usage.

After the construction, it is apparent why the buffer could be described as a slice with an improved concatenation as in the construction phase a buffer that has the same internal state variables as a slice was created, meaning just a pointer and a length.

Similar to a slice, expanding the buffer means incrementing its length state when adding items to the end of its view. When expanding it can be assumed that data has been popped, meaning removing items from the buffer start can be thought to have occurred. This indicates that the buffer slice's read pointer may have moved on to the second memory section, unless otherwise informed by the developer, more on optimization potential in the next Chapter 3.1.2.

The buffer should be stopped from accessing out of bounds memory, meaning that the view of the buffer should always be within bounds. It is to be ensured that the expanded additional length will not expand the buffer's view outside the memory owned by the buffer.

A way to ensure this is by moving the read pointer to the first memory section on each expansion and limiting the amount that can be expanded to the maximum that a single memory section may hold. Popping would then be guaranteed to not cause the read pointer to point at out of bounds memory.

In a 4096-byte memory section buffer with 8-byte elements, limiting expansion to the maximum would limit it to 512 items. Expanding any more than this will also overwrite old

data starting from the start of the buffer or it may access memory outside the control of the buffer.

When destroying the buffer, meaning freeing memory, the start of the first memory section position will need to be known. It then has to be given to the operating system for the buffer itself to be deallocated, which is why usage includes destruction. To find the first memory section with just a pointer to any position within the buffer, see Chapter 3.1.3.

### 3.1.2 Optimization

In slices, it is important that popping remains simple as it can be used in loops where each loop repetition increments the slice pointer. The goal of the study had been to remain as efficient in popping elements as a regular slice, meaning corrections to the buffer should only be done when absolutely necessary and never when popping.

Popping is assumed to be a quick operation which is why it is ill advised to check if a move of the buffer pointer back to the first memory section is necessary after each pop. The buffer pointer reset check should be done only when required and preferably only when adding more items to the buffer to lessen the number of times it is done.

Sometimes a developer using the buffer may have a much greater understanding of when it is necessary to move the buffer pointer to the correct memory section. For example, if items are removed from the end of the buffer, it is possible to continuously fill the buffer to its maximum capacity without resetting the pointer because the buffer pointer would not have moved as removing an item decrements the length of the buffer, not the pointer.

While removing from the back is not in accordance with FIFO, it would still be possible in a circular slice. Should the buffer be restricted to only FIFO, then the previous example would not be relevant. In that case a better example would be if the developer can guarantee that the total popped memory in bytes is always less than buffer's memory section size. This means that the buffer pointer is guaranteed to be within bounds.

### 3.1.3 Implementations

This section is the documentation for the bitwise algorithms for finding the start of the buffer and deterring access out of bounds with circular addressing. These algorithms are important for adding data and deconstruction.

As a reminder, the zero-bit implementation, as mentioned in the construction phase, is the one where the first memory section's size bit equals to zero. It is the implementation that has been used in the paper's practical prototypes and benchmarks.

Moving the current read pointer to the first memory section is possible with a simple AND operation on the pointer using complement of the size. In other words, if the size is 4096, its bits are inverted resulting in -4097, which is then used as the AND mask on the pointer. To access the start of the buffer or in other words the start of the allocated memory sections, it is again, a bitwise AND operation using a mask. To acquire the mask, the size must be negated before doing a bitwise AND using an inverted size. See Figure 7.

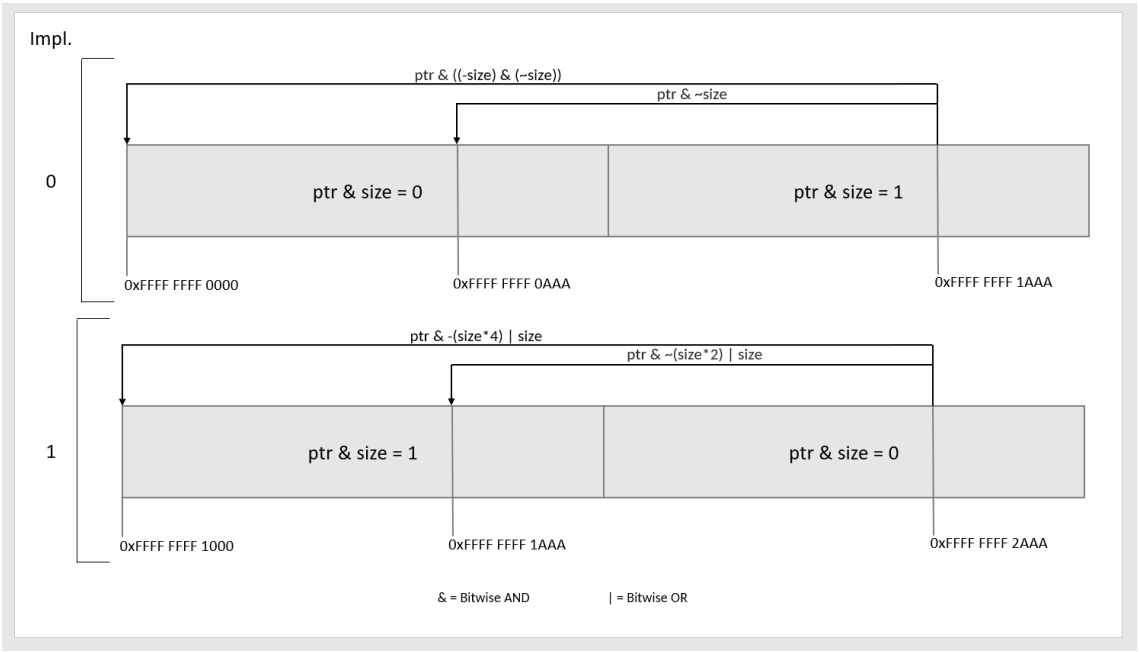


Figure 7: Pointer

The one-bit implementation ensures that the first page's size bit is one. In the paper this type of implementation is described as theoretical as no prototype has been implemented, but it is viable non the less. In the one-bit implementation, it can be noted that more complicated compute is required as it can be said to require two masks. Due to this the zero-bit variant is preferred over one-bit.

Moving the pointer to the first memory section in the one-bit implementation requires a bitwise AND operation using a mask and a bitwise OR using the size of the memory section. The mask is a complement of the size times two.

To acquire the position of the start of the memory sections, it is the same as in the previous paragraph except that the mask is an inverse rather than a complement operation and the amount inverted is times four of the size rather than two.

In a buffer with a read and write pointer, both pointers would need to be moved but with a length replacing the write pointer, only the read pointer needs to be moved to the first memory section.

It is possible to use a conditional for checking if the read pointer has entered the second section and then decrementing the pointer. However, the need for it can be completely removed by using the previously mentioned bitwise operations as the pointer is moved only when it is necessary even without a conditional.

## 3.2 Multi-threaded Buffer

The multi-threaded buffer follows the exact same construction pattern as the single-threaded version, except that after the buffer is constructed, a separate thread and mailbox will be created to manage writing to the free buffer space.

Should the writer thread's orders need to change dynamically, the mailbox should be sufficiently large to contain a pointer. Otherwise, as stated in Chapter 2.4, additional space may be available if taken from the slice count to serve as a mailbox. Details on mailbox usage in both cases can be found in the next Chapter 3.2.1.

In the threaded version of the buffer an implementation may choose to keep the third memory section instead of unreserving it and instead allocating the mailbox in it for message passing. This would allow the buffer to keep the same internal state variables as a regular slice even in the threaded buffer as is the case with the non-threaded version and thus allow it to be directly convertible to a slice again. In addition, it would also allow for dynamic changing of orders.

By using the third memory section as a mailbox, it would allow for the same slice pointer to be used to find the message box without requiring an additional reference and thus once again not lose information should the buffer need to be converted to a slice.

If the mailbox is at the start of the third memory section, an example of finding the mailbox without a reference would be zeroing the slice/buffer pointer bits starting from the size bit, to find the start of the first memory section, then adding the size of two sections to arrive at the position of the third section of memory. Details on zeroing pointers to find the start of the buffer are discussed in earlier Chapter 3.1.3.

A simple concurrency scheme would be to have just a single producer and consumer. As is known, the consumer thread has a read pointer ("ptr") and a count. However, the producer would only be passed a pointer ("ptr") to the start of the buffer when it is created and holds a function pointer source variable. Both should have access to a common shared mailbox. It is also possible to pre-emptively assign a specific source to the producer thread.

In essence, both the consumer and producer have a pointer to memory that they are allowed to access and modify without race conditions within the two contiguous shared memory sections. See Figure 8.

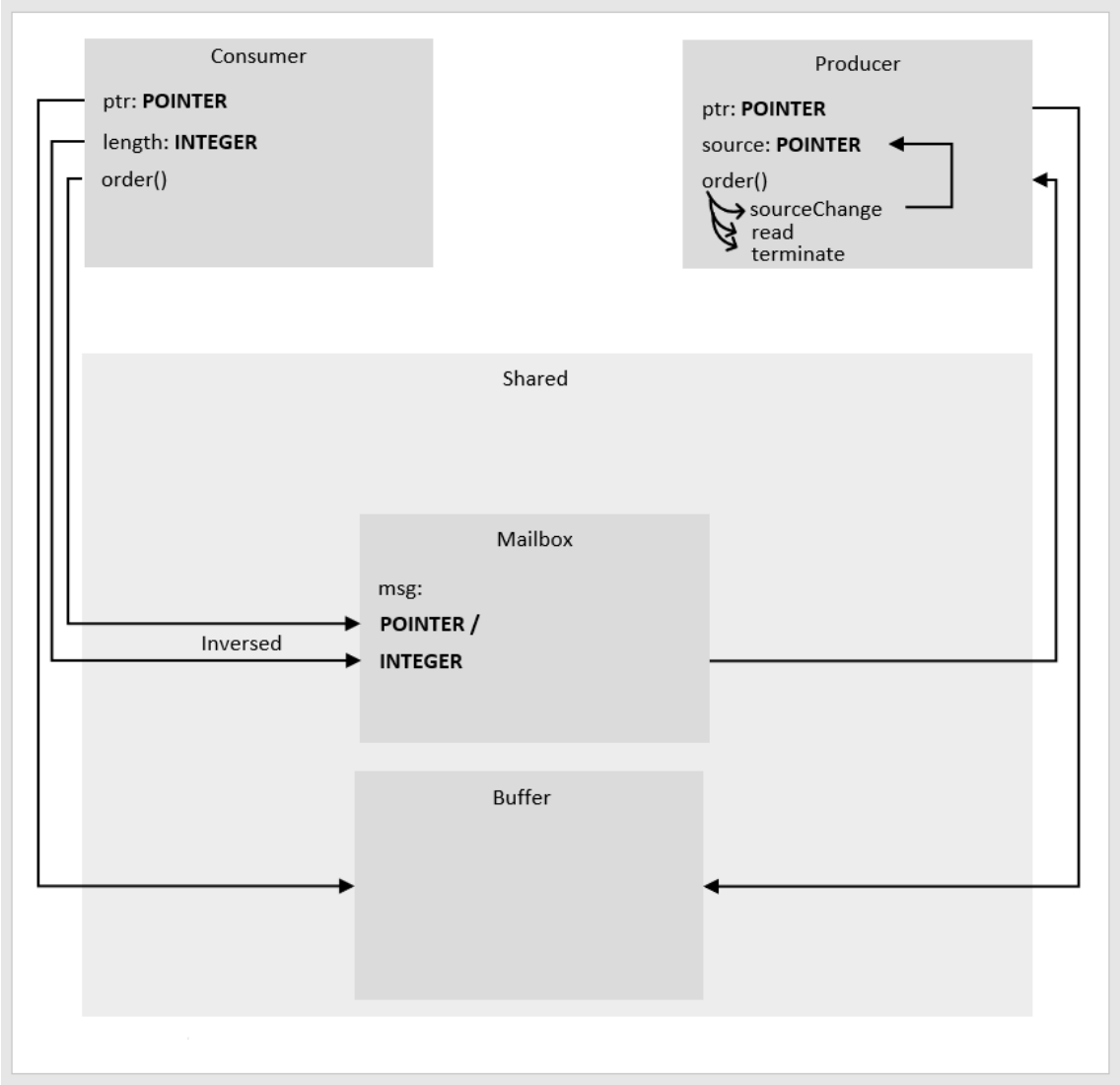


Figure 8: Concurrency

The consumer has a whole slice, meaning both the pointer and length, however the producer in this implementation is reliant on the consumers length to determine its own slice that it may access. It determines this through a message passed from the consumer. Additional information is given in the chapter below.

### 3.2.1 Usage

For information regarding the layout of the concurrent buffer, there is more in the multi-threaded buffer chapter. However here the usage of the concurrent buffer will be explained in further detail and how the mailbox is used for passing information between threads in a single consumer-producer multi-threaded buffer.

As mentioned above in 2.2 according to Ash (2012, Part II), the pointer and count in a buffer may be inconsistent with each other from an outside perspective when read. A solution to this would be to use message passing to have the consumer pass both the pointer and length to the producer through the mailbox as this would ensure that the passed data would be consistent with each other at the time of passing the value.

However, there can be additional complexity when doing this, which is why an alternative solution of passing only the length and tracing the current read pointer position was chosen instead. More information on the drawbacks of this in Chapter 3.2.2.

When the consumer is seeking more data, it will look into the mailbox to find if there is already data to expand its own slice with from the producer. If there is, then the mailbox value is positive, and it is added onto the length of the consumer's slice. After each seek of more data by the consumer, it will also request more data by setting its slice's inverted length to the mailbox. See Figure 9.

The producer only has a partial slice from a pointer, so it will wait until the consumer has set the mailbox value to less than or equal to zero and then add the maximum size of the buffer with the inverted length to acquire the amount of data that can be written by it. In other words, the consumer's inverted length gives the amount of data in the buffer that the consumer will no longer access.

This allows for the producer to begin writing beginning from its own write pointer. The producer's pointer is incremented when it writes data, meaning that in a way it can keep track of the approximate position of the consumer's pointer.

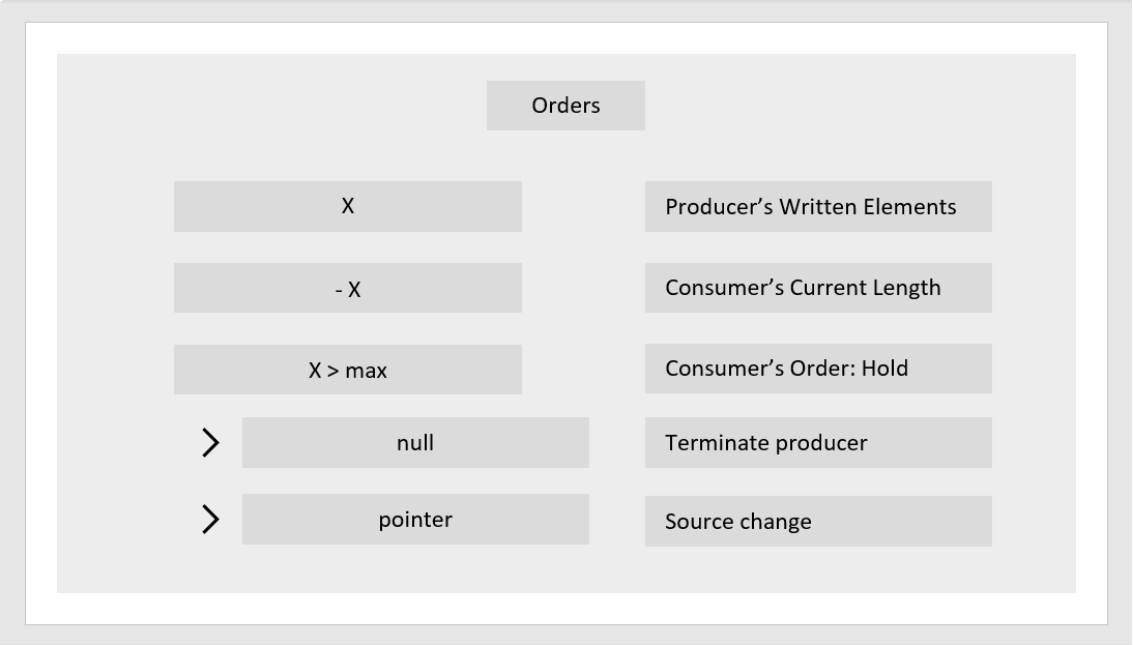


Figure 9: Orders

With a pointer and length, the producer has a full slice, and it can start writing new data to its slice without locking by means of parallel disjoint access. The producer's writing instructions are fetched from a function pointer. The function pointed to should accept a slice of the free space that the producer has access to write to. The function pointer, called ***source***, returns the amount of data written to the slice and the producer will set the mailbox to the amount of data written and increment its own slice's pointer by the same amount.

In addition to reading and writing data to the buffer, regular communication between the consumer and producer can occur. When the consumer wishes to hold the producer for passing orders to it, the consumer may set the mailbox message to be more than the maximum size of the buffer. In a 4096-byte memory section buffer where each element is 4 bytes, this would mean that the message should be set to an integer more than 1024. Holding the producer will cause it to reply with a predetermined value that is guaranteed to be different from the value given to it as a sign of a successful hold.

After a successful hold, the consumer is allowed to pass orders to the producer. The consumer may set the mailbox message to null, indicating that the producer should terminate. Any other non-null pointer value would then mean that the consumer wishes to change the data source of the producer to be the one set.

### 3.2.2 Implementations

In an implementation where the mailbox is not sufficiently large to hold a pointer, an alternative method of changing the source function, meaning writing orders to the producer, can be developed. It is possible, for example, to pass predetermined source functions when the producer has been created, any non-null message can then signify a specific source that was given previously ahead of time.

The consumer not passing its length to the producer can also be thought as a type of hold order without giving explicit order to hold. If the consumer wishes to pause work, it can simply wait for the producer to finish filling its own slice by not giving its new length or indicating that its length is the maximum size of the buffer, after the producer has finished its task.

It is important to note that the concurrency system where only the length is taken from the consumer does not allow the consumer to remove data from the end of its slice. This is not a major issue as doing so would be breaking the FIFO nature of circular buffers regardless but removing from the back of a slice is still inherently intuitive to slices. The point is that the consumer's length would change without its pointer incrementing, but the producer would not be informed of this and increment its own pointer regardless.

It is still possible to pass a copy of the buffer's slice where the data from the end is removed with a decrement to the copy's length, and therefore removing items from just the view of the copy. If complete similarity with slices is preferred even in the concurrent version of the buffer, then both the pointer and length should be passed to the mailbox as mentioned above.

# 4 Comparison

In this chapter the overall success of the implementation based on the reference implementation above is evaluated using an open-source benchmark. It is important to note that the buffer in question that was used in the benchmark is the zero-bit variant with inverted length message passing.

## 4.1 Data collection

The data was collected through an open-source community project which compared different alternative buffer implementations to the C-language Liblfds bounded single producer single consumer queue (bss) and bounded many producer many consumer queues (bmm) (liblfdsd/comparison/).

The project ran the example implementations provided by Liblfds and calculated how many messages per microsecond the two queues bss and bmm would be able to add and remove and ended up with 38314 & 21588 messages per microsecond respectively (ibid.).

These values became the bar to evaluate the success of the comparable implementations and more Implementations were gradually added by the open-source project maintainer and developers of different implementations to create a collection of different implementations attempting to accomplish the same task. All benchmark data in the chapter is collected from a fork (replica) of the project.

| | |
|---|---|
| Rwqueue | A single page-size memory section buffer with mathematical loop around with a binary AND mask. Uses atomic operations for concurrency and threads share internal states such as read & write pointers. Internal state also contains the buffer start pointer, making the internal states identical to the Linux buffer aside from using size_t instead of int. (FIFO Architecture, Functions, and Applications1999: lock-free/src/lock_free/rwqueue.d.) |
| Dlist | A linked list of nodes containing payloads. Requires iteration for acquiring length. Uses atomic operations for concurrency and threads share internal states such as head & tail pointers. (lock-free/src/lock_free/rwqueue.d.) |
| Javamp | Java blocking buffer from java.util.concurrent.ArrayBlockingQueue with a size of 4000 bytes consisting of 1000 integers (jdk7u-jdk/src/share/classes/java/util/concurrent/ArrayBlockingQueue.java. |
| Mp | Implementation similar to javamp, but uses the phobos D -language library thread message queue as a buffer (phobos/std/concurrency.d). |
| go_d | Implementation similar to rwqueue ring buffer, but using modulus instead of bitwise AND (go.d/source/jin/go/queue.d). |
| Elembuf | An implementation of the paper's described zero-page circular buffer with one memory section size being page-sized. (elembuf/source/elembuf.d). |

Table 1: Buffers

## 4.2 Results

In the benchmark, implementations that used atomic synchronization primitives as a means of concurrency such as Rwqueue, Go_d, Elembuf and Dlist fared exceptionally well compared to implementations with locks.

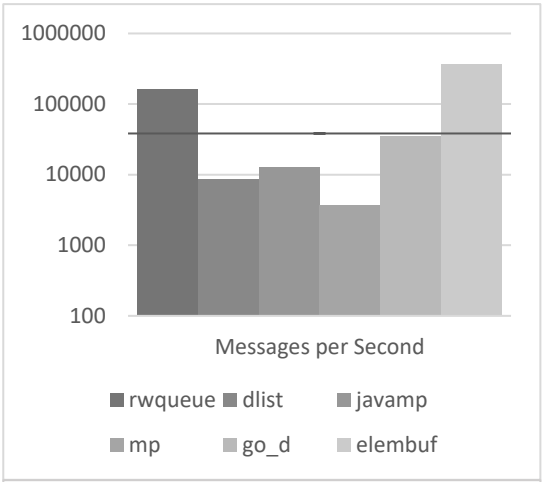


Figure 10: Performance

The performance of the linked list Dlist was similar to blocking buffers but was still higher than expected for being the only implementation in the benchmark not restricted to a fixed maximum size.

Rwqueue uses a read and a write pointer which are shared with all threads. In this implementation, acquiring the length requires fetching the pointers atomically and subtracting them, instead of having the length be a thread local state.

Go_d is similar to Rwqueue, however, it acquired lower results. Probable causes to this may have been that acquiring of its length is not a simple subtraction between the read and write pointer and it uses modulus instead of bitwise AND for circular addressing.

Reliance on memory mirroring, message passing and delayed circular addressing of memory has likely contributed favorably towards the results of Elembuf. It should also be noted that Elembuf does not require the use of circular addressing when removing items from the front, which may compare favorably in the benchmark results when compared to Rwqueue, Go_d and javamp. As each message must be popped individually, the emphasis on popping efficiency has a higher emphasis.

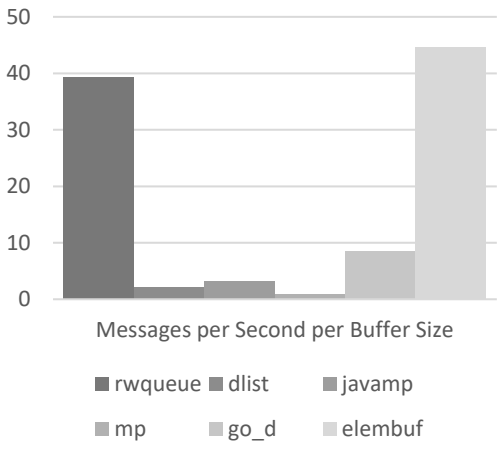


Figure 11: Performance per Buffer Size

Performance per buffer size is arguably the most interesting. Here the previous results were divided by the underlying buffer size, excluding the instance variables. Taking the buffer underlying buffer size into account, the difference becomes smaller, but is significant regardless. Elembuf is still more performant in spite of its small instance size and the fact that only half of its memory is directly usable.

## 4.3 Reliability and Validity

For consistent results, it is recommended that at least a million messages pass before an average result is taken as in the benchmark. Otherwise, there can be potentially inaccuracies if the selection group from which the average was taken is too small.

Two implementations were not included in the research paper as they were either not comparable or were not properly implemented. The first implementation was of a plain array, which was ensured to be large enough to not require any removing of elements to make room for more data. Therefore, it did not have the similar capabilities or comparable size to the other sub-page size buffers.

Another not included implementation was a custom variant of Elembuf converted to use an alternative concurrency model of avoiding message passing altogether. Unfortunately, the implementation was not implemented correctly to be included in the paper. It was using the concurrent message passing version with an additional redundant concurrency scheme on top of it.

The two MP implementations, namely Javamp & MP, had a size of 4000 instead of 4096, meaning that they had 96 bytes less than the other implementations. While this is not a substantial difference, it is important to account for when inspecting the comparison.

Finally, the most important aspect to keep in mind is that the comparison lacks other buffers that use the virtual memory system. While it is likely that other buffers using the virtual memory system can benefit from the optimizations described in the paper, such a comparison could be a task for future studies.

# 5 Conclusions

The research paper began by viewing previous work on ways of improving efficiency. Details on what is a slice and how they are implemented was also discussed. Using previous experience, a new slice compatible circular buffer was designed, which simplified the common functions of a buffer.

In the construction portion of the research, it was apparent that the creation of the buffer temporarily requires three times as much memory for storing data and permanently two times more after construction and may therefore take longer to acquire than other buffers. This was identified as a potential disadvantage for using the reference implementation that is based on the virtual memory system.

However, in the usage section it was apparent that the usage was simplified through not requiring any additional steps for removing data from the buffer. Removing elements became identical to how a slice removes data.

With length replacing the write pointer, only the read pointer needed to be kept within bounds. Additional benefit was acquired in that it became possible to identify the position of the whole buffer from any pointer position within the buffer.

Through the improved usage, the slower construction time of the buffer can be made insignificant through the reuse of the same buffer or construction can occur at the start of a longstanding program rather than when required.

In the comparison different buffering solutions were compared with the circular buffer relying on the virtual memory management system taking the lead in the ability to pass the most messages per second. Overall, a slice circular buffer using custom circular buffering techniques retained the performance benefits of a virtual memory system.

References

Printed

Arpaci-Dusseau, Remzi & Arpaci-Dusseau, Andrea.2018. Operating Systems: Three Easy Pieces. University of Wisconsin-Madison.

Han, Mengijiao; Wald, Ingo; Usher, Will; Morrical, Nate; Knoll, Aaron; Pascucci, Valerio & Johnson, Chris R. 2020. A Virtual Frame Buffer Abstraction for Parallel Rendering of Large Tiled Display Walls. IEEE Visualization Conference.

Morandi, Benjamin, Nanz, Sebastian & Meyer, Bertrand. Safe and Efficient Data Sharing for Message-Passing Concurrency. In Kühn E., Pugliese R. (Eds.) Coordination Models and Languages. COORDINATION 2014. Lecture Notes in Computer Science, vol 8459. Springer, Berlin, Heidelberg. Zurich: Department of Computer Science.

Pirkle, Will. 2013. Designing Audio Effect Plug-Ins in C++ With Digital Audio Signal Processing Theory. New York and London: Focal Press.

Schill, Mischael; Nanz, Sebastian & Meyer, Bertrand. 2013. Handling Parallelism in a Concurrency Model. In Lourenço J.M., Farchi E. (Eds.) Multicore Software Engineering, Performance, and Tools. MUSEPAT 2013. Lecture Notes in Computer Science, vol 8063. Springer, Berlin, Heidelberg.

Wilhelmsson, Jesper. 2005. Efficient Memory Management for Message-Passing Concurrency Part I: Single-threaded execution. Uppsala: Uppsala University.

Electronic

Ash, Mike. 2012. Ring Buffers and Mirrored Memory: Part I. Available at: <https://mikeash.com/pyblog/friday-qa-2012-02-03-ring-buffers-and-mirrored-memory-part-i.html>. Accessed: 2 Apr 2022.

Ash, Mike. 2012. Ring Buffers and Mirrored Memory: Part II. Available at: <https://www.mikeash.com/pyblog/friday-qa-2012-02-17-ring-buffers-and-mirrored-memory-part-ii.html>. Accessed: 2 Apr 2022.

Circular Buffers. Available at: <https://www.kernel.org/doc/html/v5.4/core-api/circular-buffers.html>. Accessed: 25 Jan 2022.

elembuf/source/elembuf.d. Available at <https://github.com/Cyroxin/Elembuf/blob/master/source/elembuf.d>. Accessed 25 Jan 2022.

FIFO Architecture, Functions, and Applications.1999. Available at: <https://www.ti.com/lit/an/scaa042a/scaa042a.pdf>. Accessed: 24.1.2022.

Fog, Agner. 2021. Optimizing software in C++ An optimization guide for Windows, Linux, and Mac platforms. Available at: <https://www.agner.org/optimize/optimizing_cpp.pdf>. Updated 8.11.2021. Accessed 24 Jan 2022.

go.d/source/jin/go/queue.d. Available at: <https://github.com/nin-jin/go.d/blob/master/source/jin/go/queue.d>. Accessed 2 March 2022.

Gregory E. Allen, Paul E. Zucknick, and Brian L. Evans. 2006. Zero-copy Queues for Native Signal Processing Using the Virtual Memory System. Available at:

<https://users.ece.utexas.edu/~bevans/papers/2006/zeroCopyQueues/ZeroCopyQueuesAsilConf2006Paper.pdf>. Accessed: 4 March 2021.

jdk7u-jdk/src/share/classes/java/util/concurrent/ArrayBlockingQueue.java. Available at: <https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/concurrent/ArrayBlockingQueue.java>. Accessed: 2 Apr 2022.

liblfdsd/comparison/. Available at: <https://github.com/mw66/liblfdsd/tree/7c7420a759e886022cbc8f1c1821a707ee7a550c/comparison>. Modified: 23 June 2020. Accessed: 22 Jan 2022.

linux/include/linux/circ_buf.h. Available at: <https://github.com/torvalds/linux/blob/2f47a9a4dfa3674fad19a49b40c5103a9a8e1589/include/linux/circ_buf.h>. Updated 8 May 2018. Accessed: 24 Jan 2022.

lock-free/src/lock_free/rwqueue.d. Available at: <https://github.com/MartinNowak/lock-free/blob/master/src/lock_free/rwqueue.d>. Accessed: 1 March 2022.

phobos/std/concurrency.d. Available at: <https://github.com/dlang/phobos/blob/master/std/concurrency.d>. Accessed: 2 March 2022.

Pomberger, Gustav; Bischofberger, Walter; Kolb, Dieter; Pree, Wolfgang & Schlemm, Holger. 1991. Prototyping-Oriented Software Development — Concepts and Tools. *Structured Programming* 12: 43-60. Springer-Verlag New York Inc. Available at: <https://www.softwareresearch.net/fileadmin/src/docs/publications/J001.pdf>. Accessed 2 April 2022.