Antoni Paavola

Reduced Circular Slice Buffer II - Pragmatic Multiplatform Implementation

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

8 May 2023

Abstract

Author:	Antoni Paavola
Title:	Reduced Circular Slice Buffer II – Pragmatic Multiplatform Implementation
Number of Pages:	42 pages
Date:	8 May 2023
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	Mobile Solutions
Supervisor:	Jarkko Vuori, Principal Lecturer

There is a growing interest in slices due to an ever-increasing trend to move toward a lower level within the field of programming. This paper pursues the integration of a slice and a circular buffer attempting to make an object that can be described using both terms, either separated or fused. The present paper strives to attain a more thorough practical approach by explaining the steps required to design a circular slice buffer.

This paper contrasts the envisioned circular slice buffer with an ordinary memory copying buffer. This is carried out by benchmarking the performance of different parts of both buffers within different environments. The study has a practice-based design approach to research.

This study is a follow-up to Paavola's paper (2022) which described a more abstract idea and outline of what a circular slice buffer could be and how one could implement concurrency within a slice-based buffer.

A circular slice buffer outperforms a memory copying buffer in its usage performance. However, it is to be noted that a circular slice buffer fails to outperform an ordinary memory copying buffer if it is not properly reused. The amount of reusage required depends on the system it is used in.

It would be worthwhile for future studies to investigate whether it is feasible to integrate circularity into existing ordinary slices and to what extent.

Keywords: circular buffer, slice, benchmark, pragmatic, implementation

Tiivistelmä

Tekijä:	Antoni Paavola
Otsikko:	Kevennetty ympyräviipalepuskuri II – Pragmaattinen monialustainen toteutus
Sivumäärä:	42 sivua
Aika:	8.5.2023
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Information Technology
Ammatillinen pääaine:	Mobile Solutions
Ohjaaja:	Yliopettaja Jarkko Vuori

Ohjelmointialalla kiinnostus nk. viipaleisiin (engl. slices) kasvaa jatkuvasti, sillä nykypäivän suuntaus on siirtyä jatkuvasti kohti matalamman tason ohjelmointia. Tutkimuksessa oli tarkoitus yhdistää alemman tason viipaleet ja ympyräpuskuri yhteen. Näin voitaisiin luoda objekti, jota voidaan kuvata molemmin termein joko erikseen tai yhdistettynä. Vaikka tätä aihetta on käyty läpi Paavolan (2022) aikaisemmassa I osan tutkimuksessa teorian pohjalta, nyt tehdyssä työssä pyrittiin luomaan pragmaattisempi ote esittämällä ympyräviipalepuskurin luomisen eri vaiheet.

Työssä vertailtiin ympyräviipalepuskuria ja tavanomaista muistia kopioivaa puskuria mittaamalla puskureiden eri ominaisuuksien suorituskykyä eri ympäristöissä. *Kevennetty* tutkimuksen nimessä tarkoittaa sitä, että puskuri käyttää vain kahta muuttujaa kuten viipalekin. Kevennyksen myötä oli tärkeää tietää, sisältääkö kevennettykin ympyräpuskuri tehokkuusetuja muistia kopioivaan puskuriin verrattuna. Tämä selvitettiin tekemällä tehokkuusvertailu. Työn toteuttamisessa käytettiin menetelmänä käytäntöön perustuvaa lähestymismallia (engl. practice-based design approach).

Työssä tehdyt mittaukset osoittivat, että ympyräpuskuri suoriutuu tavanomaista muistia kopioivaa puskuria paremmin käytön suhteen. Ympyräpuskuri vaatii kuitenkin uudelleenkäyttämistä, jotta se voisi olla nopeampi kuin tavanomainen muistia kopioiva puskuri, sillä sen luominen on tavallista puskuria hitaampaa. Uudelleenkäytön määrä riippuu siitä, missä ympäristössä puskuria käytetään.

Tulevissa tutkimuksissa olisi hyvä tutkia, onko mahdollista integroida tutkimuksen sisältöä olemassa olevien viipaleiden käyttötarkoituksiin ja kuinka laajasti tämä voitaisiin toteuttaa.

Avainsanat: ympyräpuskuri, viipale, suorituskykyvertailu, pragmaattinen, toteutus

Contents

List of Abbreviations

POSIX:	Portable Operating System Interface. A set of standards defining compatibility between multiple operating systems. Allows for abstraction of the operating system in software.
SLICE:	A type consisting of a pointer to memory and length. Used to signify a portion of data but can also be used to encompass all available data that is within memory.
RCSB:	Reduced Circular Slice Buffer. The combination of a circular buffer and a slice.
MCB:	Memory Copy Buffer. A linear memory buffer, where memory is copied to the start of the buffer when necessary.
LIFE CYCLE:	The lifetime of the buffer. Signifies a buffer's construction and destruction, excluding its usage.
USAGE:	All actions taken after construction and before deconstruction such as writing & removing items from a buffer.
CODE PATH:	The set of instructions chosen to be compiled. In the study, specific code is chosen based on which operating system the buffer is run in.

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 even more essential to provide greater efficiency for processing data and providing greater throughput from the hardware already present.

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

The research divides different sections of code into so-called code paths. These code paths are chosen at compile time depending on what system calls the target environment supports. In other words, the study concentrates on
code paths, their content and performance, hence taking a pragmatic approach which in turn simplifies validification.

This study is a follow-up to Paavola's paper (2022) which described a more abstract idea and outline of what a circular slice buffer could be and how one could implement concurrency within a slice-based buffer.

1.1 Purpose of the study

By contrast with the previous study which compared multiple different concurrent buffer implementations, the aim here is to concentrate on the Reduced Circular Slice Buffer itself, as a non-concurrent buffer, and to find out in practice how quickly it can be constructed and used compared to a memory copying buffer. Accordingly, the study bases itself on the former research but creates further insights into the Circular Buffer variant. This is done by opening broad abstract subjects mentioned in the previous study by Paavola (2022).

The 2022 study presented the results of an abstract buffer variant and left a possibility for further ideation without specifying or explaining how the buffer was created or used in practice. In a sense it can be thought that this research explains the previous study by unfolding topics and questions that may have been left unanswered.

The study narrowed itself to a certain sized buffer, whereas the current study takes into account the possibility of larger sizes than the minimum. Due to this, there are more complex definitions and formulae regarding the buffer. The former study also attempted to explain how to integrate a circular buffer with a slice in theory, while the current study seeks to present it in practice.

However, it should be remembered that there is not just a single way of implementing a circular slice buffer and this study does not attempt to depict an ideal variant of a circular slice buffer, but simply a possibility.

It is also important to remember the possible differences between both studies such as the target audience. The previous study conveyed innovation to researchers seeking to create new buffer variations while the current concentrates on instructing implementers.

1.2 Research questions

The main research question the paper seeks to answer is how the Reduced Circular Slice Buffer performs compared to a memory copying buffer. How do the lifecycle and buffer usage perform? What kind of results do we acquire from different implementation variations?

Secondly, to create a generally verifiable study, how can we pragmatically implement some of the topics discussed in the first study? Furthermore, in different operating systems and the Posix standard, what are the code paths like for both life cycle and usage?

1.3 Research structure

In the next chapter, called Background, the basic idea of a buffer is introduced, and it also delves slightly into the rationale in making a buffer circular. In this section, the study attempts to depict the starting point of literature and how the study has proceeded to develop further within the field.

In Chapter 3, Implementations, we will look into the Lifecycle and usage of both the study's own circular buffer and a generalized memory copying buffer. This will assist with understanding the differences between both variants and code paths of the same variant, allowing for greater comprehension of Chapter 4, Comparison.

In Chapter 4 the study will seek to compare the results of the implementations for both life cycle and usage. In Chapter 5, Conclusion, major findings are summarized, and information is further structured.

2 Background

To get an overall sense of the topic itself, it is necessary first to understand the meaning of a circular buffer and in general what buffers are used for. Buffers are used in various areas of computer science within the context of concurrency. 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.

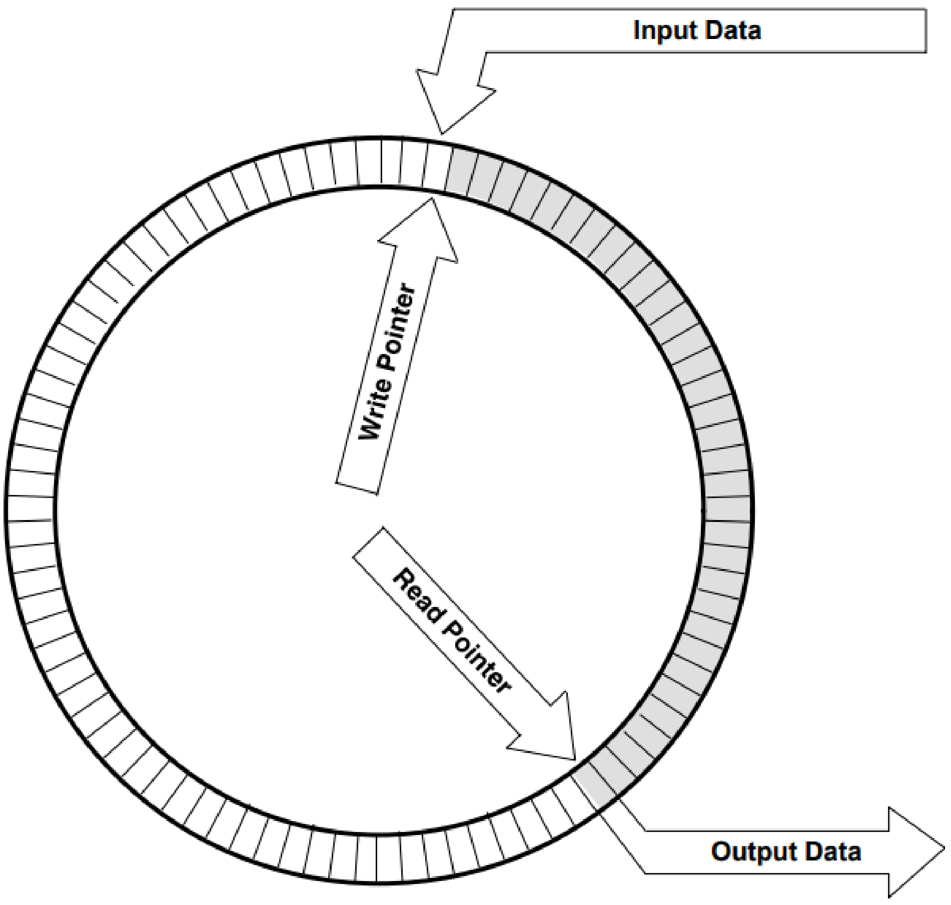


Figure 1: Circular Buffer (FIFO Architecture, Functions, and Applications 1999).

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 us 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 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.

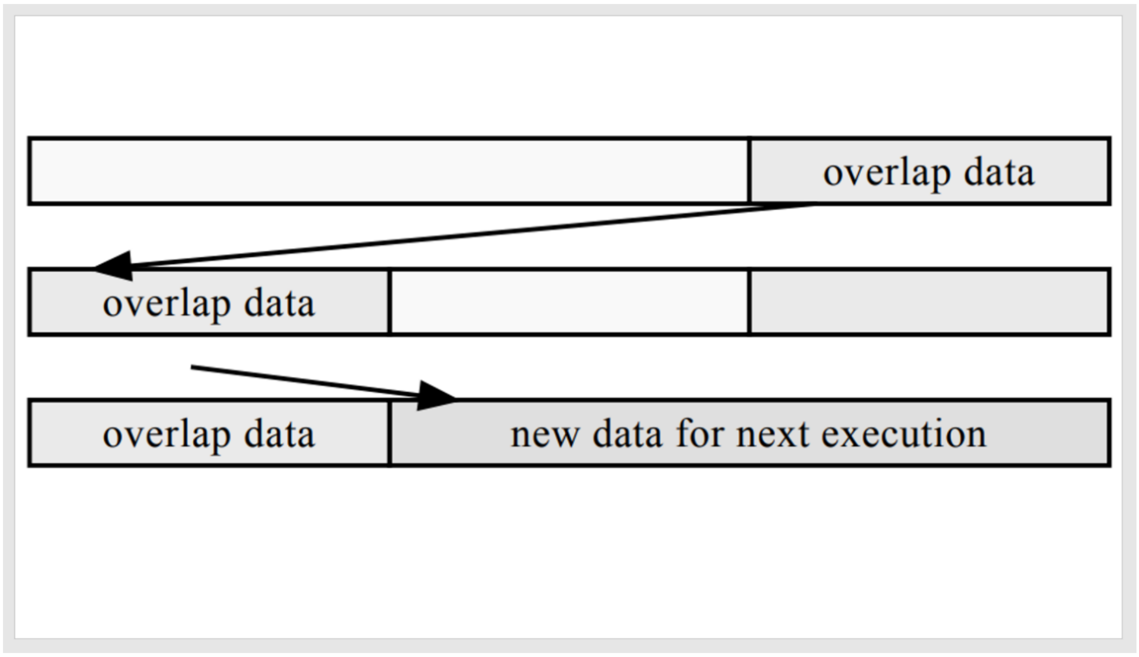


Figure 2: Data copying (Allen, Zucknick & Evans 2006).

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.)

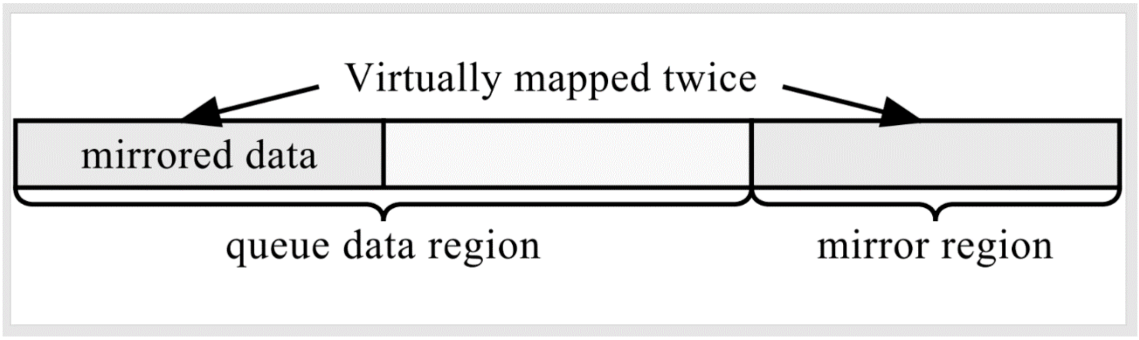


Figure 3: Mirroring (Allen, Zucknick and Evans 2006).

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 therefore 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.

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 to 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 also ideal due to their small size, since they fit well within a machine’s register. Registers are small pieces of memory within the processor. Variables within registers can be accessed quickly, but registers can only store a limited number of variables. (Fog 2021: 27, 36, 50, 69, 134 and 144.)

Using only a pointer in the consumer would cause inefficiency from looping through the buffer as it is more efficient if the length is known. This is why slices are preferred, as they contain the length as well. To get the length of data using just a pointer, iteration of the buffer to search for an end marker is needed. This is easily possible with a character array with a zero-byte character, but with other types a value would need to be designated as an end marker. (Fog 2021: 36, 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 Methodology

The methodological design of this study is inspired by practice-based design research (Horváth 2007). According to Horváth, the research approach, which intends to generalize and extend design knowledge based on strongly contextualized practical knowledge, has been called practice-based design research. See Figure 4 below.

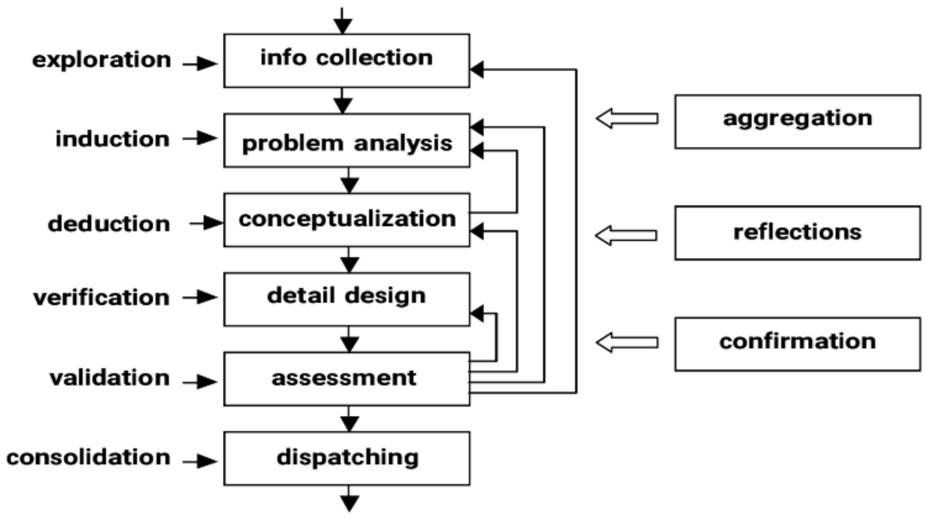


Figure 4: Methodology (Horváth 2007).

The information collected was acquired from the reference implementation in the previous study. This study found the lack of pragmatic solutions to be a problem with the previous study in addition to the unanswered question of life cycle performance, i.e. referring to the problem analysis. Taking advantage of existing research and aligning to Horvath`s conceptualization phases, a testing framework was built to verify the previous study (conceptualization & detail design). The testing framework was used to create a benchmark, it was then analyzed (assessment) and conclusions based on the analysis were drawn (dispatching).

According to Bose (2022) performance testing is a type of software testing for evaluating how a certain software performs under variant conditions. He lists numerous variables - stability, scalability, speed and responsiveness to which performance may refer. Yet, in this study performance solely refers to speed.

3 Implementations

The study developed two buffers for handling continuous data flow. One with circular slice characteristics, called the Reduced Circular Slice Buffer, or RCSB for short, and another that uses manual buffer management called Memory Copy Buffer or MCB.

The buffers’ ways of handling memory overlap are drastically different, but they do contain some shared terms, which are key to understanding the implementation details that are to be discussed later on.

As the buffers work with multiple different systems, the page size can potentially differ drastically, but for the purposes of the simplicity of this paper, it can be thought of being 4096 bits unless otherwise mentioned. As mentioned in Chapter 2.2, the page size can be assumed, but that is not recommended.

In a real-world environment, it is recommended that the page size is saved separately from the buffer internal variables so that multiple buffer instances may find and share a common immutable instance of the page-size, rather than a compile-time value as is the case in the study. Acquiring the page size from the operating system on every buffer construction should not be done, only when necessary.
The key term required for understanding the RCSB is **usable pages**. Memory mapped circular buffers contain multiple memory mapped buffer pages of equal size, but often this is limited to just two buffer pages. The RCSB requires two buffer pages to be mapped, the first one being identified as the **usable page**.

Buffer pages should not be mixed up with the operating system pages since a buffer page can contain multiple pages of system memory. However, a RCSB can choose to only support buffer page sizes of a single page size, making it inherently identical to one operating system page.

The **usable page's size** is a power of two and a multiple of the system page size. Sometimes **buffer memory** will be mentioned, in this case it can be assumed to refer to the total size of each buffer page in memory:

$$U_{\text{sable page size}} = P_{\text{age size}} \cdot 2^k, \qquad k \in W$$

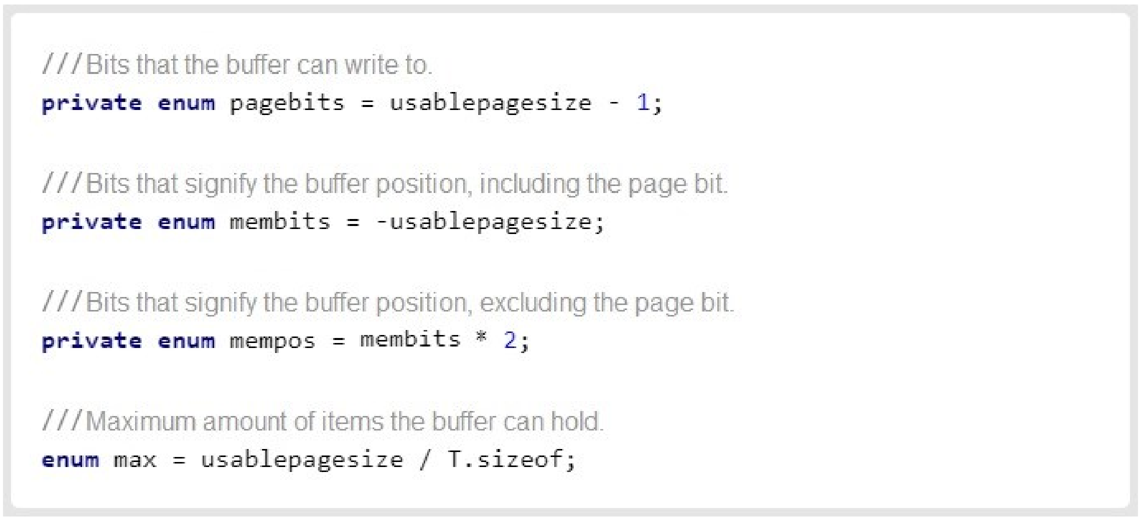
$$B_{\text{uffer memory}} = B_{\text{uffer pages}} \cdot U_{\text{sable page size}}.$$

Note that on some systems, such as Windows, **a platform specific alternative to page size can be used instead**, more information in the system specific documentation.

When discussing bits of the buffer pages, the study used three key masks that rely solely on the **size of the usable page**. Should the system page size be known or assumed at compile-time, the following variable values may also be acquired at compile-time.

The "**pagebits**" mask displays one for all bits that the buffer can write to and is acquired by subtracting one from the usable page size. The negation of **pagebits**, or alternatively inverse of the usable page size is called "**membits**", as it displays the buffer's memory position bits as ones, including the buffer page bit, when multiplying it by two, the buffer page bit can be excluded, resulting in "**mempos**". See Figure 5 below for more details.

The universal maximum that one may store in a buffer is the size of the usable page divided by the size of the type of a single element in the buffer. In the study however, the usable page size was was limited to one system page size, hence maximum is then the system page size divided by the buffer element's type size. More on this in Figure 5 below.



```
///Bits that the buffer can write to.
private enum pagebits = usablepagesize - 1;

///Bits that signify the buffer position, including the page bit.
private enum membits = -usablepagesize;

///Bits that signify the buffer position, excluding the page bit.
private enum mempos = membits * 2;

///Maximum amount of items the buffer can hold.
enum max = usablepagesize / T.sizeof;
```

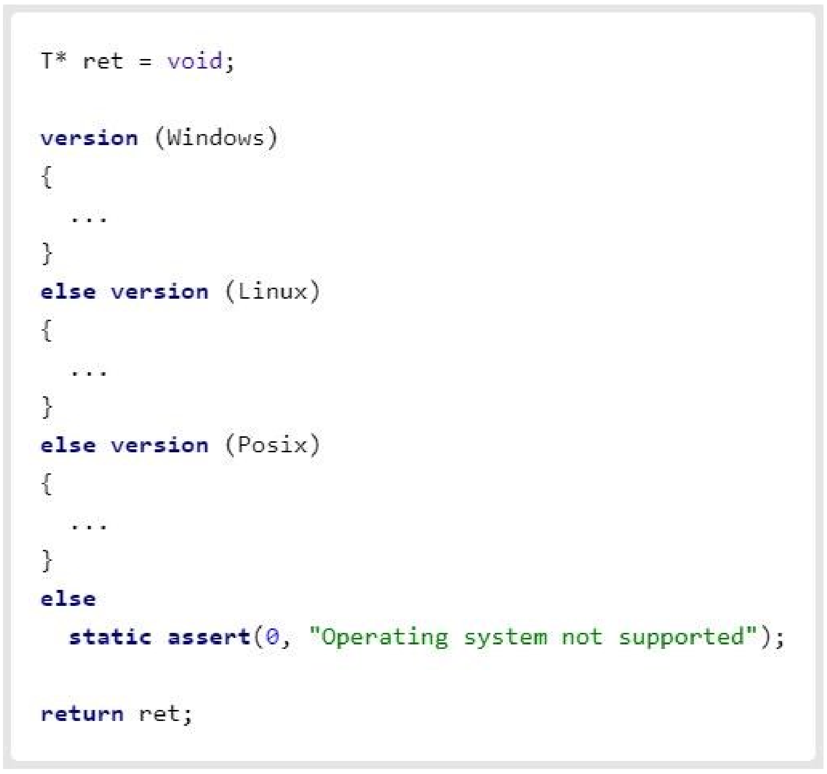
Figure 5: Constants.

3.1 RCSB – Reduced Circular Slice Buffer

The **Reduced Circular Slice Buffer** or RCSB for short, is a general-purpose buffer which varies slightly in implementation depending on the platform. The sub-chapters are divided into different platform lifecycle implementations and shared usage.

The lifecycle implementation details in each of the platforms, namely Linux, Posix and Windows, can be said to be very similar with only minor exceptions, understanding a single platform lifecycle is sufficient for continuing on with the usage general of the buffer.

In each lifecycle chapter, both construction and deconstruction will be discussed. In construction, the starting memory position of the constructed buffer is returned to the calling function. More details on platform specific chapters, see also Figure 6 below.



```
T* ret = void;

version (Windows)
{
  ...
}
else version (Linux)
{
  ...
}
else version (Posix)
{
  ...
}
else
  static assert(0, "Operating system not supported");

return ret;
```

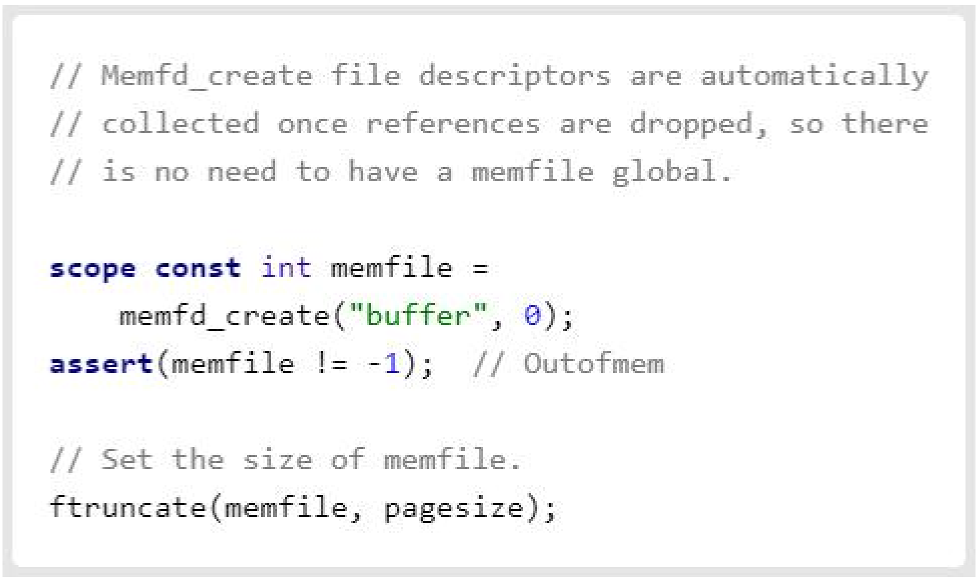
Figure 6: Code paths.

3.1.1 Linux Construction

RCSB’s construction under Linux can be considered a much more simplified version of the Posix -version. Although it is possible to run Posix code in a Linux environment, platform specific can be assumed to be more performant. More details on the performance difference in the chapter Comparison.

In construction, a memory file must be created, in Linux this is best done with **memfd_create** as to make the file anonymous. Although the file is anonymous, it is given a name in addition to flags as its arguments. A memory file descriptor is given or alternatively a value indicating an error. The name does not need to be unique, and the operating system can be thought to be fairly lenient on the correctness of the given name, as seen in Figure 7 below.

After the memory file is created, it is set to the required size with **ftruncate**, arguments being the previously created memory file handle and the required usable memory size. In this case, usable memory is defined to be a single page size. See Figure 7 below.



```
// Memfd_create file descriptors are automatically
// collected once references are dropped, so there
// is no need to have a memfile global.

scope const int memfile =
    memfd_create("buffer", 0);
assert(memfile != -1);  // Outofmem

// Set the size of memfile.
ftruncate(memfile, pagesize);
```

Figure 7: Linux Construction.

The steps after this will be identical to the Posix implementation. In general terms, three times the usable memory defined earlier needs to be requested with **mmap**. The function will return a pointer to memory.

As mentioned earlier at the start of the Implementation chapter, usable memory is a power of two, meaning it has only a single bit that is active at any time. Henceforth the position of this bit will be called the **usable memory bit**.

It is necessary to find the usable memory bit within the pointer returned by **mmap** and ascertain whether it is 0 or 1. The internal functions of the buffer rely on the compile-time assurance that the buffer's beginning position in memory is so that its usable memory bit is known. In the examples, the usable memory bit is chosen to always be zero, meaning not active. Implementations with this decision can be described as **zero-bit implementations**. The usable memory bit can also be chosen to be one, meaning active, but implementation will differ from the given examples. An implementation seeking the bit to be active can be labeled as a **one-bit implementation**.

Should an AND bitwise operation of the usable memory bit on the pointer result in zero, then the usable memory bit within the three requested sets of memory should correspond to 0, 1, 0. Otherwise it shall be 1,0,1.

In the first case a zero-bit implementation would seek to return the last set of memory to the operating system and a one-bit implementation would return the first set.

In the second case a zero-bit implementation would seek to return the first set and a one-bit the last set. Whichever case, a zero-bit implementation would now have usable memory bits of 0 and 1 and a one-bit 1 and 0. For more details on which sets of memory to choose according to the chosen implementation, see Figure 8 below.

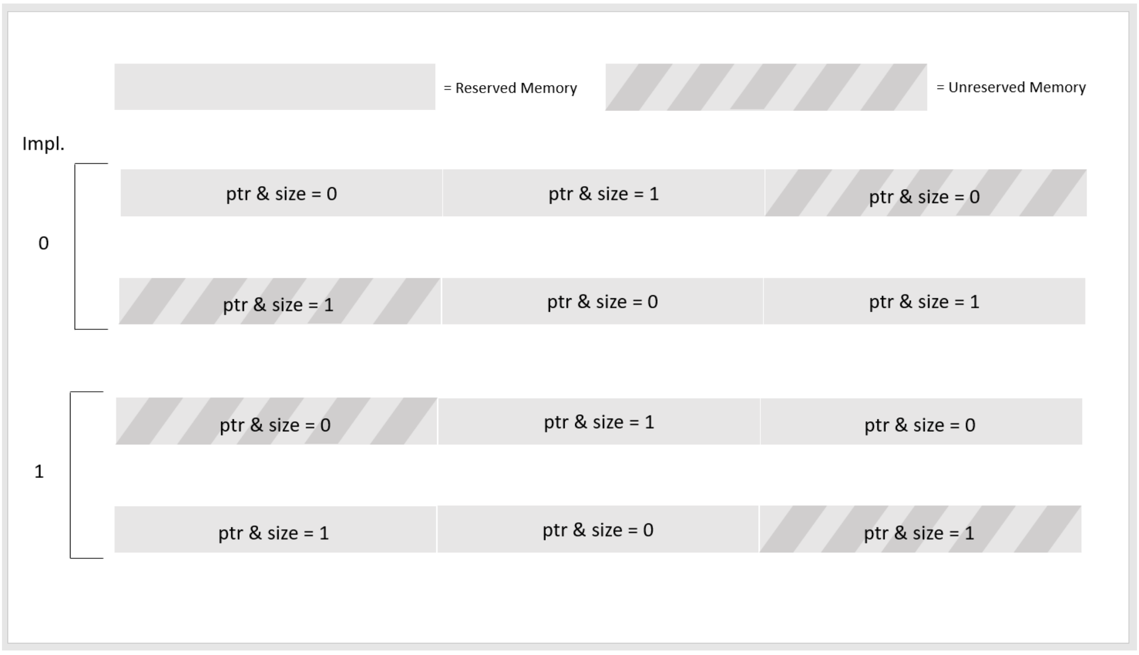
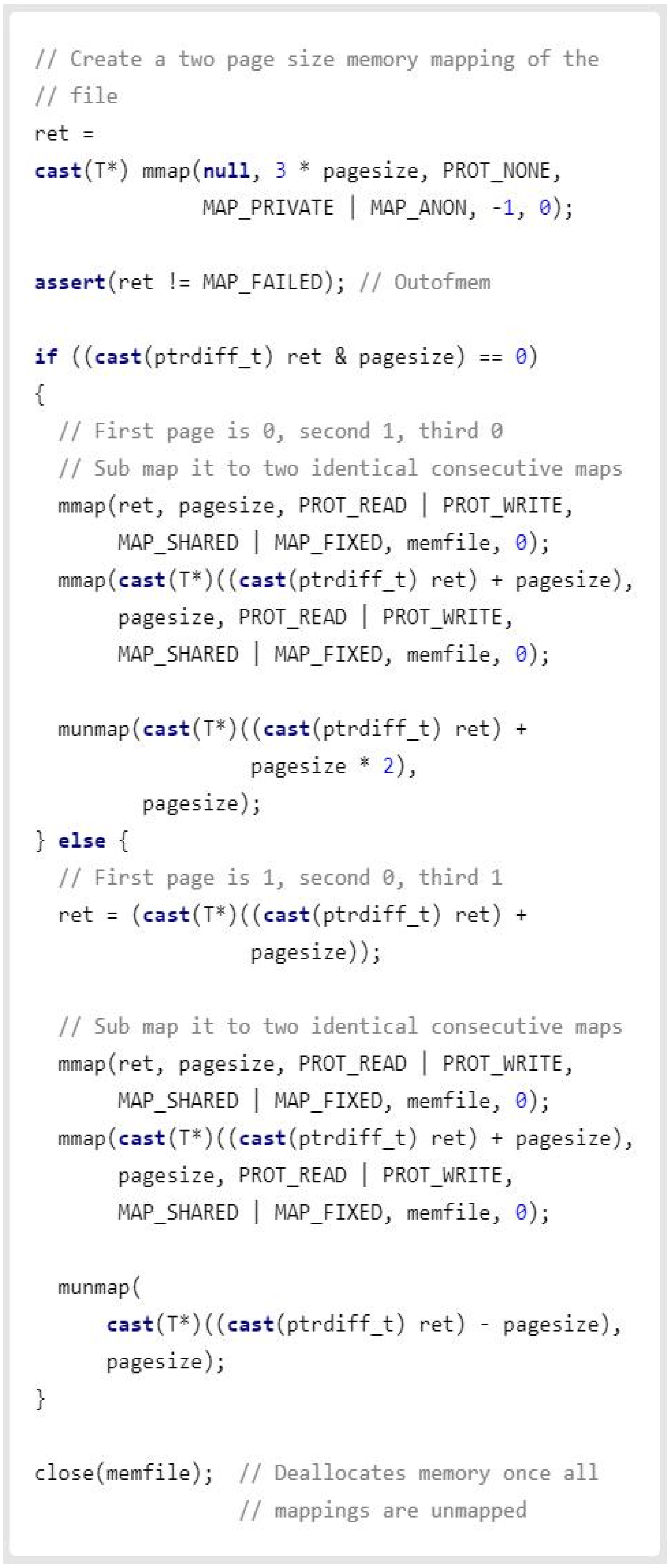


Figure 8: Memory Selection.

The remaining two memory sets must be mapped to the earlier created memory file so that both sets of memory contain identical data and always show the entirety of the data within the memory file.

Finally, the created memory file can be closed since the operating system cannot un-allocate the memory file since there are still two connections to it, namely the two maps linked to the data of the memory file. This is the base idea which makes storing the memory file in a variable redundant, since the memory file is now automatically collected by the operating system after un-mapping the memory from the memory file. An example of the construction working for both Linux and Posix systems can be found below in Figure 9 for 0-bit implementations.



```
// Create a two page size memory mapping of the
// file
ret =
cast(T*) mmap(null, 3 * pagesize, PROT_NONE,
              MAP_PRIVATE | MAP_ANON, -1, 0);

assert(ret != MAP_FAILED); // Outofmem

if ((cast(ptrdiff_t) ret & pagesize) == 0)
{
  // First page is 0, second 1, third 0
  // Sub map it to two identical consecutive maps
  mmap(ret, pagesize, PROT_READ | PROT_WRITE,
       MAP_SHARED | MAP_FIXED, memfile, 0);
  mmap(cast(T*)((cast(ptrdiff_t) ret) + pagesize),
       pagesize, PROT_READ | PROT_WRITE,
       MAP_SHARED | MAP_FIXED, memfile, 0);

  munmap(cast(T*)((cast(ptrdiff_t) ret) +
                  pagesize * 2),
         pagesize);
} else {
  // First page is 1, second 0, third 1
  ret = (cast(T*)((cast(ptrdiff_t) ret) +
                  pagesize));

  // Sub map it to two identical consecutive maps
  mmap(ret, pagesize, PROT_READ | PROT_WRITE,
       MAP_SHARED | MAP_FIXED, memfile, 0);
  mmap(cast(T*)((cast(ptrdiff_t) ret) + pagesize),
       pagesize, PROT_READ | PROT_WRITE,
       MAP_SHARED | MAP_FIXED, memfile, 0);

  munmap(
      cast(T*)((cast(ptrdiff_t) ret) - pagesize),
      pagesize);
}

close(memfile);  // Deallocates memory once all
                 // mappings are unmapped
```

Figure 9: Linux Construction 2.

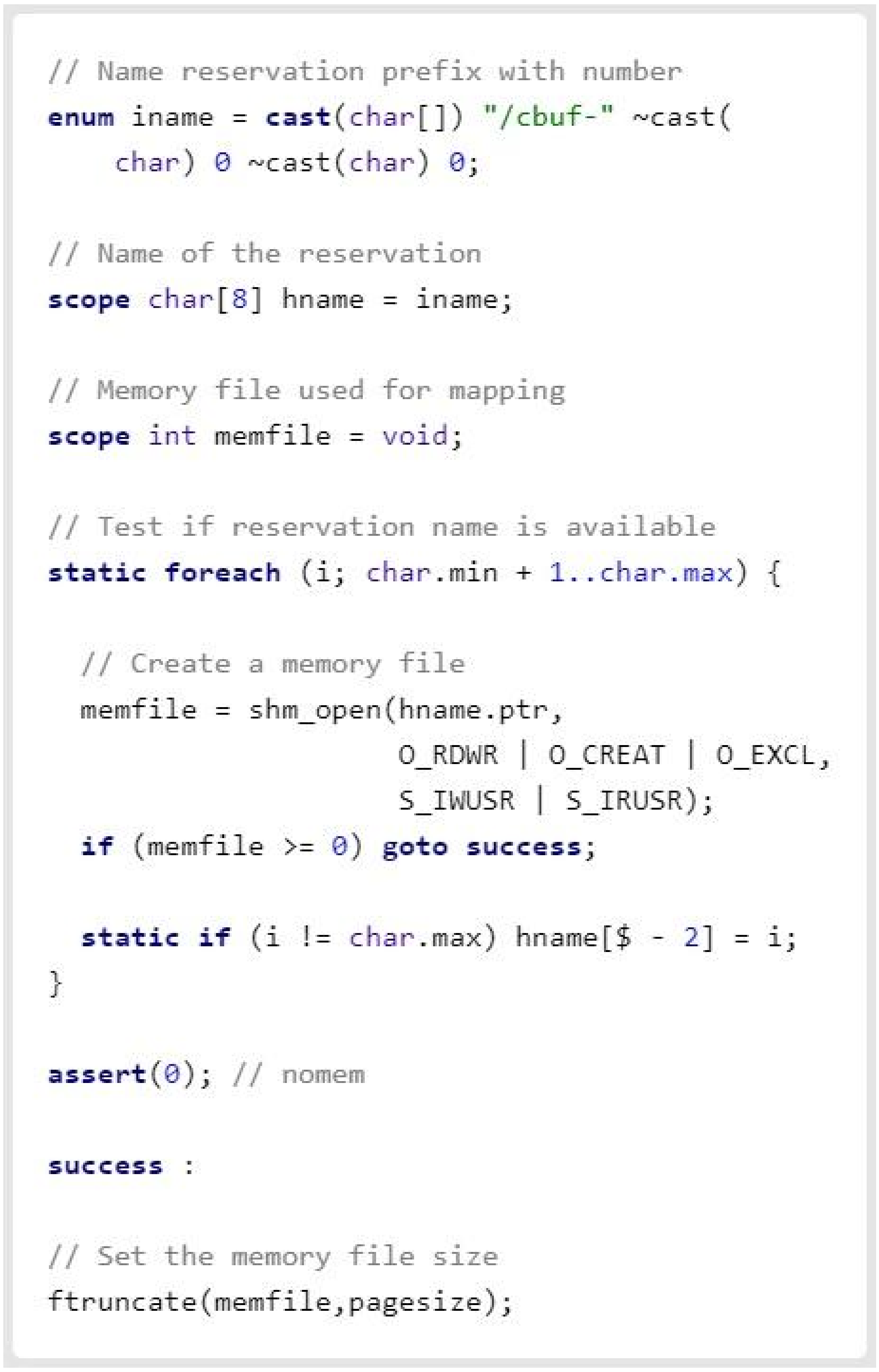
3.1.2 Posix Construction

The lifecycle of RCSB under Posix is identical except for the construction, more specifically creation of the memory file. In this environment, the memory file must have a unique name.

Instead of the previously mentioned memfd_create system call, that is not recognized under Posix, **shm_open** is to be used instead. Otherwise, its usage is very similar as it is given a name and flags. A memory file descriptor is returned by the system call similarly to memfd_create or alternatively a value indicating an error should an issue occur, such as the name not being unique.

As a general suggestion for successfully reserving available names, it is best to have a prefix and a buffer indicator, since it is possible that there are multiple instances of the same type of buffer. The name string must be null terminated.

Similarly, to Linux, the memory file size must be defined with **ftruncate.** An example of reserving a memory file can be seen from Figure 10 below.



```
// Name reservation prefix with number
enum iname = cast(char[]) "/cbuf-" ~cast(
    char) 0 ~cast(char) 0;

// Name of the reservation
scope char[8] hname = iname;

// Memory file used for mapping
scope int memfile = void;

// Test if reservation name is available
static foreach (i; char.min + 1..char.max) {

  // Create a memory file
  memfile = shm_open(hname.ptr,
                     O_RDWR | O_CREAT | O_EXCL,
                     S_IWUSR | S_IRUSR);
  if (memfile >= 0) goto success;

  static if (i != char.max) hname[$ - 2] = i;
}

assert(0); // nomem

success :

// Set the memory file size
ftruncate(memfile,pagesize);
```

Figure 10: Posix Construction.

The steps to map the memory file and deconstruct the buffer can be seen under the chapter where Linux lifecycle was discussed as it applies to Posix as well, however, it should be noted that **shm_unlink** should be used on the name pointer after mapping the memory file.

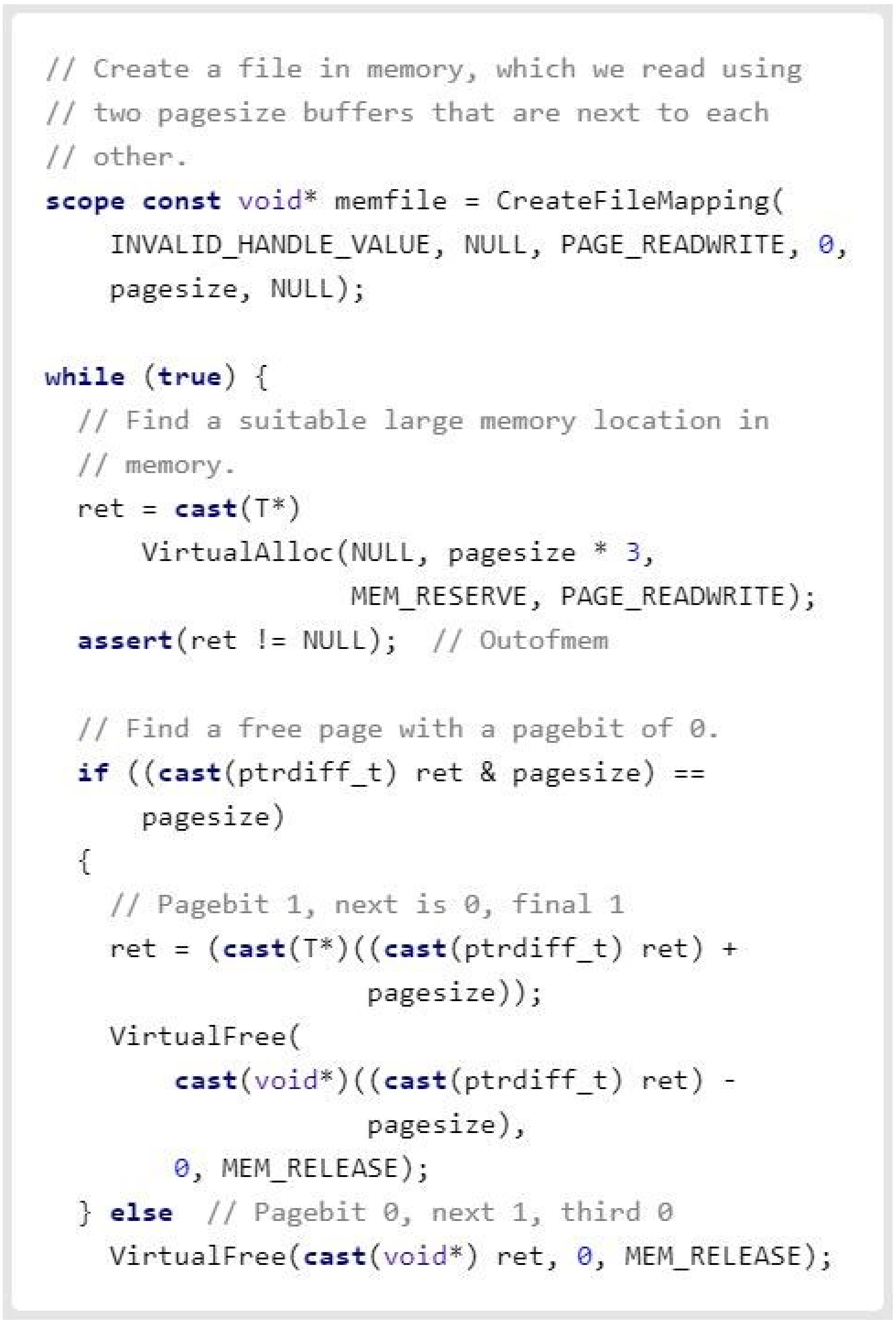
3.1.3 Windows Construction

In Windows, the general idea of the buffer construction remains the same, however, there are possible situations that require additional work. Additionally, it should be noted that the code examples within this chapter use **page size**, however, on Windows this keyword has been aliased to be **allocation granularity** instead of the usual system **page size,** to function properly.

As discussed earlier, the working of the buffer is based on a shared memory file that has its content mapped into two consecutive memory areas. The memory file creation on windows is performed with **CreateFileMapping**, its parameters can be referenced from its documentation.

To find a suitable area of memory, three buffer pages worth of memory must be reserved. Next the excess buffer page must be identified, according to the rules of a **zero-bit** or **one-bit** implementation, refer to the earlier Linux chapter for more in depth details.

The next steps differ greatly from the previous implementations. In Posix based systems, it is possible to directly map memory, however, on Windows the memory must be unreserved before mapping. This is why all the previously reserved memory must be unreserved after identifying the two out of three buffer pages that must be mapped. Details on this from Figure 11 below of a 0-bit implementation, where the buffer page is identical to page size.



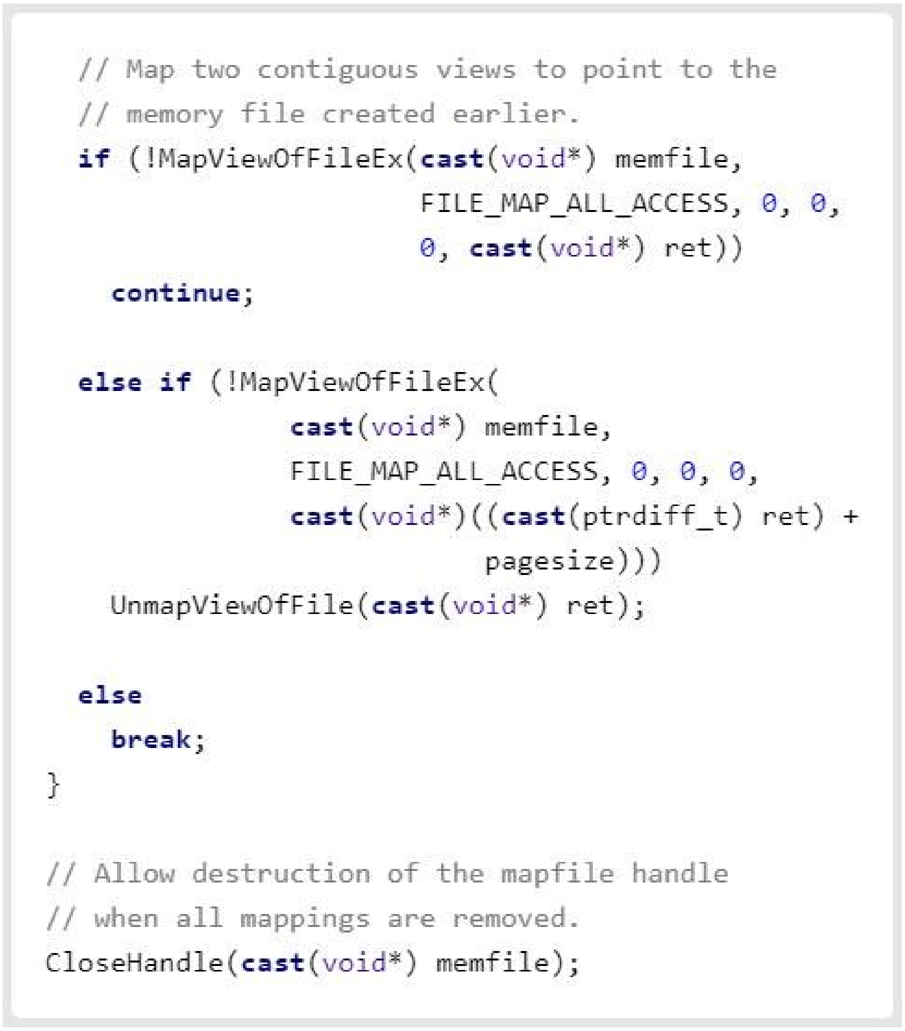
```
// Create a file in memory, which we read using
// two pagesize buffers that are next to each
// other.
scope const void* memfile = CreateFileMapping(
    INVALID_HANDLE_VALUE, NULL, PAGE_READWRITE, 0,
    pagesize, NULL);

while (true) {
  // Find a suitable large memory location in
  // memory.
  ret = cast(T*)
      VirtualAlloc(NULL, pagesize * 3,
                   MEM_RESERVE, PAGE_READWRITE);
  assert(ret != NULL);  // Outofmem

  // Find a free page with a pagebit of 0.
  if ((cast(ptrdiff_t) ret & pagesize) ==
      pagesize)
  {
    // Pagebit 1, next is 0, final 1
    ret = (cast(T*)((cast(ptrdiff_t) ret) +
                    pagesize));
    VirtualFree(
        cast(void*)((cast(ptrdiff_t) ret) -
                    pagesize),
        0, MEM_RELEASE);
  } else  // Pagebit 0, next 1, third 0
    VirtualFree(cast(void*) ret, 0, MEM_RELEASE);
```

Figure 11: Windows Construction.

Reserving, un-reserving and mapping memory must be done in a loop since after un-reserving, a race condition may occur. This means that either one of the buffer pages contains memory that came to use after un-reserving the found consecutive memory area. Mapping can be done with **MapViewOfFile**, but since the memory area to map must be chosen, **MapViewOfFileEx**, must be used instead.

After un-reserving, should any of the mapping fail, a race condition can be assumed to have occurred. If a race condition occurred when mapping the first, **usable,** buffer page to the memory file, then the program must find another three consecutive buffer pages of memory and retry mapping with the new memory.

If the first mapping was successful, but the second buffer page mapping was not, then the first mapping must be unmapped with **UnmapViewOfFile** and a similar re-attempt must occur as failing of the first map attempt. More details from Figure 12 below.



```
    // Map two contiguous views to point to the
    // memory file created earlier.
    if (!MapViewOfFileEx(cast(void*) memfile,
                         FILE_MAP_ALL_ACCESS, 0, 0,
                         0, cast(void*) ret))
      continue;

    else if (!MapViewOfFileEx(
                 cast(void*) memfile,
                 FILE_MAP_ALL_ACCESS, 0, 0, 0,
                 cast(void*)((cast(ptrdiff_t) ret) +
                             pagesize)))
      UnmapViewOfFile(cast(void*) ret);

    else
      break;
  }

  // Allow destruction of the mapfile handle
  // when all mappings are removed.
  CloseHandle(cast(void*) memfile);
```

Figure 12: Windows Construction 2.

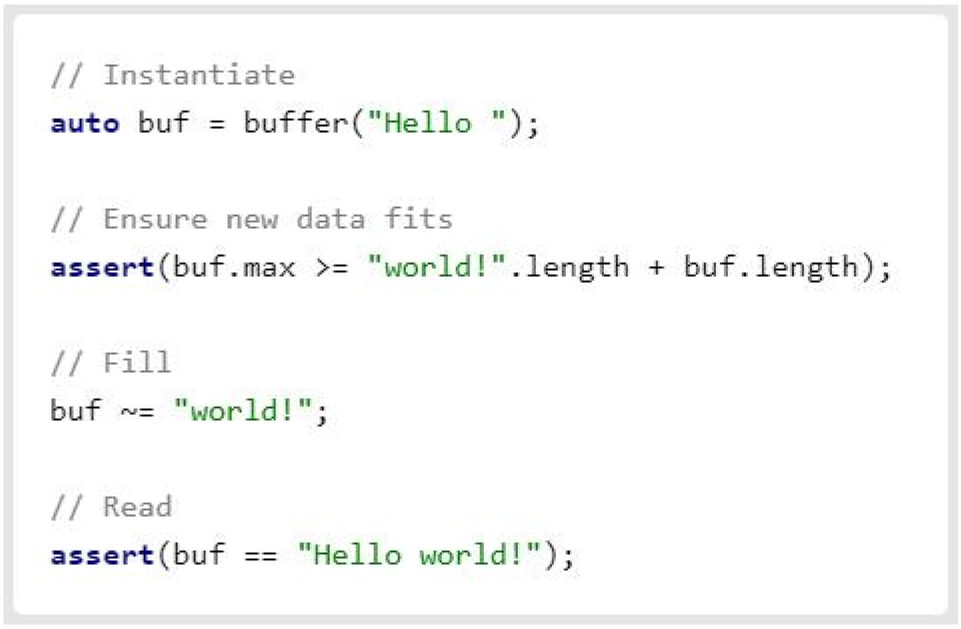
Once both mappings are successful, as usual, the memory file must be set to automatically free itself once the mappings are removed by closing the memory file with **CloseHandle**.

3.1.4 Usage

Regardless of the different methods of construction, the usage of the buffer remains the same across all platforms. The usage was designed to remain intuitive and closely match how an array would operate.

An example of intuitive design is accepting all types of data by means of compile-time analysis. As discussed before, the size of a single element of the buffer plays a key role in determining the maximum the buffer may hold, which makes compile-time computation essential.

The usage also attempts to closely match how a standard array slice would operate while hiding the circular addressing details behind the scenes. As the maximum size of a circular buffer is fixed, the user of the buffer must ensure any additional data does not exceed the buffer limit. An example of the standard usage of a RCSB, or Reduced Circular Slice Buffer, can be seen in the below example Figure 13.



```
// Instantiate
auto buf = buffer("Hello ");

// Ensure new data fits
assert(buf.max >= "world!".length + buf.length);

// Fill
buf ~= "world!";

// Read
assert(buf == "Hello world!");
```

Figure 13: General Usage Example.

How the **RCSB** operates internally in its usage is very much like how an ordinary slice would operate. As a reminder, a slice is simply a pointer and length to data and similarly so is the reduced circular slice buffer.

Slices can view portions or whole pieces of data. If a slice had no information regarding the total size of the underlying data that it is viewing a portion of (or if the underlying data was **immutable**), concatenating a string would not possible. Instead, the portion viewed by the slice would need to be copied and combined with the additional data to be concatenated.

Concatenating (in other words adding more data) into a slice where the underlying memory size is unknown, would need a copy of the underlying data to be made in order to avoid writing into unallocated memory. However, as the maximum size of the circular slice buffer is known, the scope or view of the slice buffer can be extended to see more of the underlying data.

To add more data, the length of the buffer slice must be increased by the size of the new data and the newly increased memory area must be written to contain the data.

Additional complexity comes from circular addressing, since the end of the buffer slice cannot be extended forever even if the total data after concatenation is below the buffer maximum hold capacity, since the buffer end may eventually be pointing to memory not owned by the buffer.

To be able to continue reading and extending the buffer slice, there must be a way to ensure the buffer never accesses memory not owned by it by moving the view of the slice to memory owned by the buffer when needed.

To do this, it is possible to force the slice view start to always remain within the first buffer page, hence referred to as the **usable page** as it is the only page which may be pointed to by the slice pointer. Although the length of the slice may, should and will encompass data within the second page.

It is possible to do a conditional to see if the buffer slice pointer has crossed to the next page and after which move it if needed. However, a more efficient method that the RCSB uses is binary operations on the pointer itself.

Due to the guarantees made during construction that the buffer page bit is always known to be whichever the implementation decides, either 0 or 1, it is possible to reset the buffer page bit of the buffer slice pointer regularly to ensure the pointer is always pointing at the first page. This can be achieved with basic binary operations.

In this way, a conditional is not required since the buffer does not need to be aware of which page the buffer slice pointer is on, it only needs to reset the buffer page bit when requested even if it already is on the first page. Resetting the buffer pointer bit is inexpensive due to using binary operations.

During development of the RCSB, the study initially decided to reset the pointer during each pop, in other words during the removal of elements from the start of the buffer slice. However, it was identified that it was unnecessarily expensive when small amounts of memory needed to be popped at a time and caused the loss of any performance gains.

It became apparent that the pointer is best reset during concatenation, since it was done less frequently and is an action necessary for enabling any popping, or removal of data from the start of the buffer, since it is not possible to pop if there is no data. This is the method the RCSB uses for infinite popping, a visual example is available in the below Figure 14.

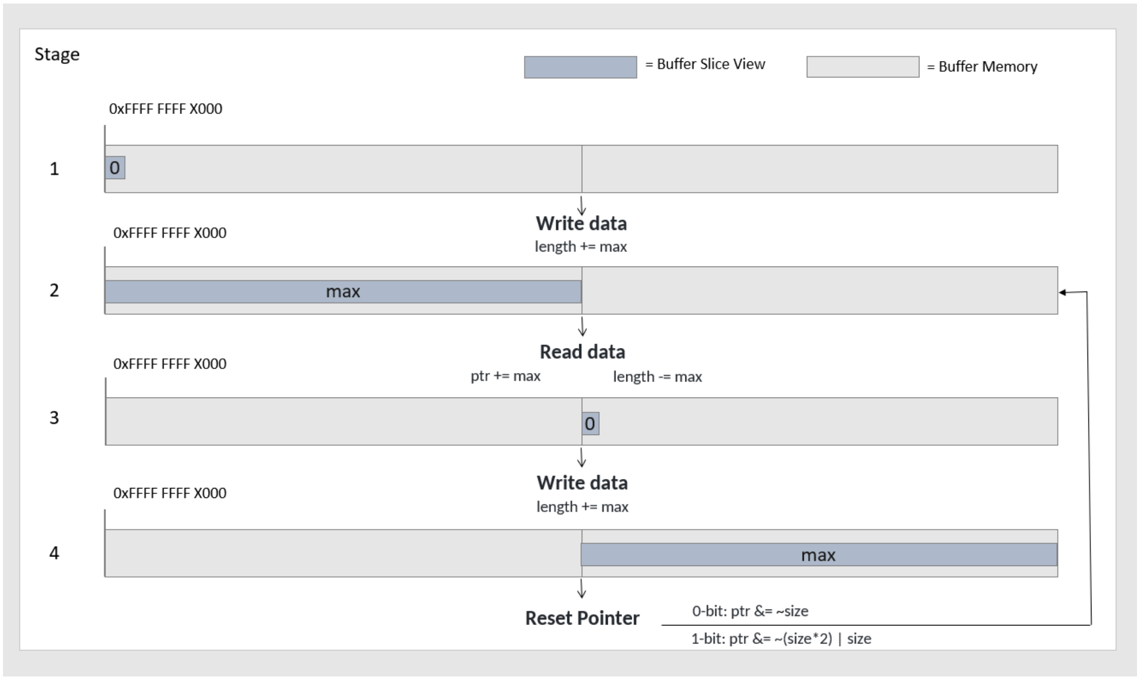


Figure 14: Pointer Reset.

In practice, concatenation in the RCSB is very simple. The buffer slice is set to have its pointer reset and its length increased by the amount that it should be extended with. The newly extended memory area must also be written to contain the new data. Example of a 0-bit implementation in Figure 15 below.

```
// Reset buffer pointer to the first page & add more length to buffer slice.
buf = (cast(T*)(cast(ptrdiff_t) buf.ptr & ~pagesize))[0 .. buf.length +
rhs.length];

// Write new data to the added memory area.
buf[$ - rhs.length .. $] = rhs[];
```

Figure 15: Pointer Reset in Practice.

3.1.5 Destruction

Deconstruction, or in other words freeing memory back to the operating system, is also made simple alike to usage since previously in buffer construction the memory file handle was already pre-closed.

Only two memory maps remain with links to the pre-closed memory file. As previously discussed, un-mapping the memory mapped to the memory file will allow for the memory to be freed. Do note that depending on the language, the buffer slice may also need to be manually freed.

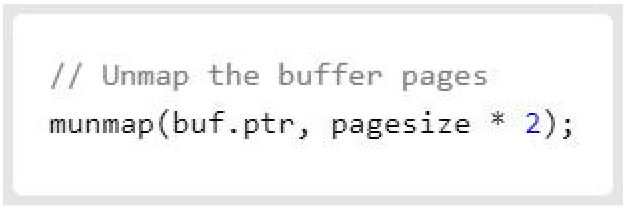
To free the memory maps and therefore allow for the automatic freeing of the memory file, it is required to give a pointer to indicate which memory page or pages need to be freed, this of course is highly dependent on the operating system.

In systems that require the start of the memory map for freeing the maps, it is possible to reset the buffer slice pointer to the start of the first page with simple binary operations, also called **usable page,** see Figure 16 below for a 0-bit example of this.

```
// Buffer first page start
auto buf = (cast(T*)(cast(ptrdiff_t) buf.ptr & (membits & (~pagesize))))
[0..buf.length];
```

Figure 16: Acquiring the Buffer Beginning Position.

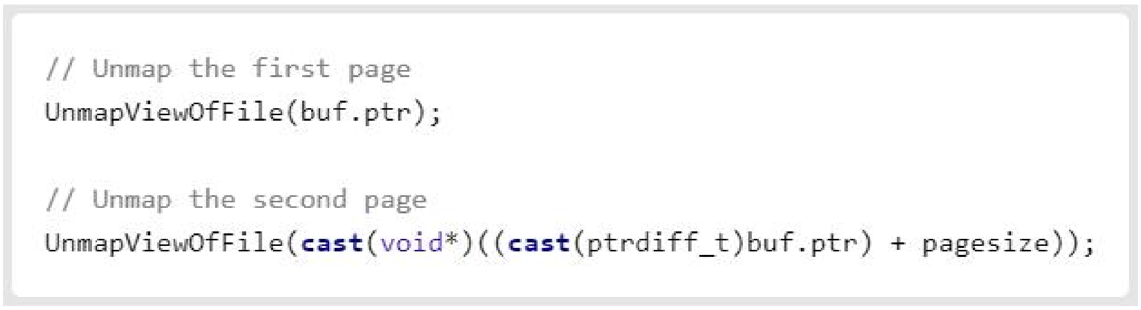
On systems compatible with Posix, unmapping is as simple as calling **munmap**, passing a pointer to the memory map start and the amount of memory mapped including both buffer pages. Example of this from Figure 17 below.



```
// Unmap the buffer pages
munmap(buf.ptr, pagesize * 2);
```

Figure 17: Munmap.

On Windows, the maps can be separately unmapped with **UnmapViewOfFile** passing the locations of both buffer pages separately. Example from Figure 18 below.



```
// Unmap the first page
UnmapViewOfFile(buf.ptr);

// Unmap the second page
UnmapViewOfFile(cast(void*)((cast(ptrdiff_t)buf.ptr) + pagesize));
```

Figure 18: UnmapViewOfFile.

After this the buffer slice can be freed as well or simply returned to a program managed memory pool. There should no longer be any references remaining to the memory file, allowing for it to be automatically collected by the operating system.

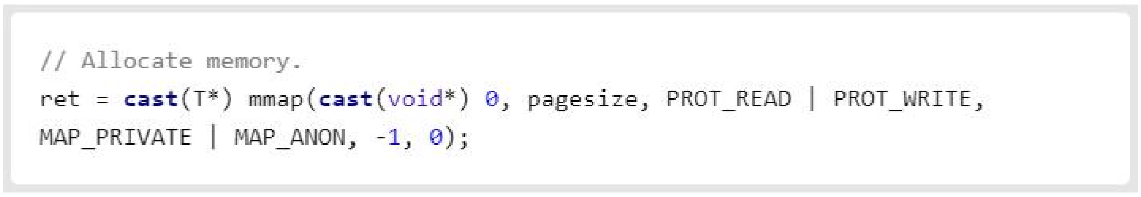
3.2 MCB – Memory Copy Buffer

The Memory Copy Buffer, or MCB shorthand, is designed as a comparable implementation for evaluating a linear buffer which moves data to allow for concatenation (see Allen, Zucknick & Evans 2006). This implementation attempts to replicate the buffer management of a copying buffer as described above in the Chapter called Buffer Management.

3.2.1 Lifecycle

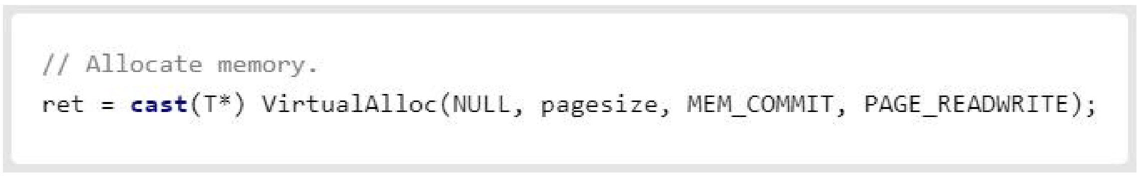
Since a linear buffer does not require mapping, the size of a linear memory buffer does not hold any constraints aside from being a positive integer and could be built on top of a language built-in array reserved from a program memory pool.

However, to ensure a version of a copying buffer that is comparable, its usable memory size was ensured to be equal to the **RCSB**'s. Additionally, memory was requested directly from the operating system, instead of using an internal memory pool. Pointer to the acquired memory is then set as the buffer pointer, example of doing this in both Windows and Posix systems in below Figures 19 and 20.



```
// Allocate memory.
ret = cast(T*) mmap(cast(void*) 0, pagesize, PROT_READ | PROT_WRITE,
MAP_PRIVATE | MAP_ANON, -1, 0);
```

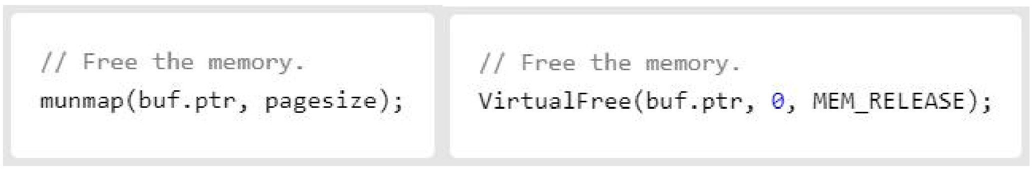
Figure 19: Requesting Memory Posix.



```
// Allocate memory.
ret = cast(T*) VirtualAlloc(NULL, pagesize, MEM_COMMIT, PAGE_READWRITE);
```

Figure 20: Requesting Memory Windows.

Deconstructing is as simple as passing the same pointer acquired initially to the respective system call depending on the operating environment, as discussed in previous sections. See Figure 21 below.



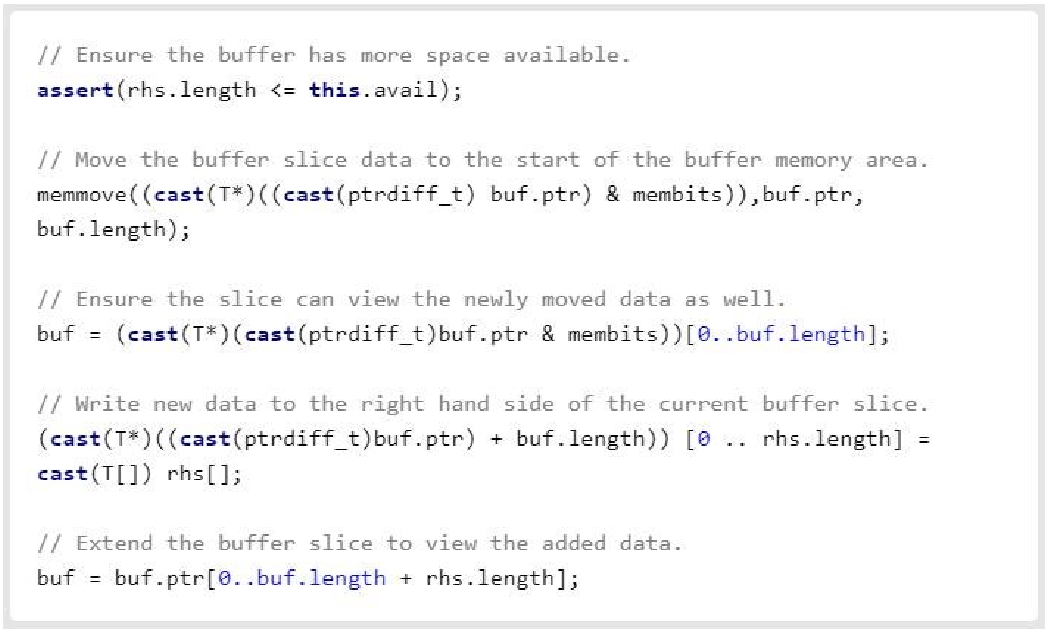
```
// Free the memory.
munmap(buf.ptr, pagesize);
```

```
// Free the memory.
VirtualFree(buf.ptr, 0, MEM_RELEASE);
```

Figure 21: Freeing Memory.

3.2.2 Usage

When concatenation is called on the buffer, there is additional work required to ensure the operation of a linear memory buffer. As memory has potentially been popped, the existing memory must be moved to the beginning of the memory allocated for the buffer. The buffer slice pointer must then also be moved to the start of the page. Concatenated data must be written to the memory subsequent to the buffer slice's view and the buffer slice's length increased by the amount of data that needs to be concatenated. See Figure 22 below for example.



```
// Ensure the buffer has more space available.
assert(rhs.length <= this.avail);

// Move the buffer slice data to the start of the buffer memory area.
memmove((cast(T*)((cast(ptrdiff_t) buf.ptr) & membits)),buf.ptr,
buf.length);

// Ensure the slice can view the newly moved data as well.
buf = (cast(T*)(cast(ptrdiff_t)buf.ptr & membits))[0..buf.length];

// Write new data to the right hand side of the current buffer slice.
(cast(T*)((cast(ptrdiff_t)buf.ptr) + buf.length)) [0 .. rhs.length] =
cast(T[]) rhs[];

// Extend the buffer slice to view the added data.
buf = buf.ptr[0..buf.length + rhs.length];
```

Figure 22: Memory Copying Buffer Usage.

As seen from the example, an additional move is now required in addition to writing new data and updating the buffer pointer**.** It should be noted that assertions are debug only and are not included in release builds.

The issue with this implementation of a Memory Copy Buffer is that since it uses binary operations and there is no second buffer page, the buffer pointer will cross to another page not owned by the buffer should all the buffer items be popped after the buffer is fully filled. To solve this issue, the Memory Copy Buffer must limit its maximum to be at least one less than what it is truly capable of holding.

4 Comparison

Both the Reduced Circular Slice Buffer and the Memory Copy Buffer both have their unique advantages and disadvantages, so to illustrate that, the study developed an internal benchmark for comparing both the construction and usage of both buffers.

In the benchmark lifecycle part, both buffers will be constructed and deallocated 100 000 times with the resulting time taken counted. In the usage part of the benchmark, both buffers will be filled fully with example data and then half of the filled data is popped, this as well is repeated the same number of times as in the lifecycle benchmark.

In the usage benchmark, the example data itself that is filled into the buffers is determined in a way that alternates between a less than sign ‘<’ and greater than sign ‘>’ so that the first requested item to be filled will always be ‘a less than’ sign and the next one a ‘greater than’ sign.

It was decided that both buffers should use the same buffer size of one usable page, but with two items removed from the maximum that the buffer may be filled with. This is due to the Memory Copy Buffer limiting its maximum to at least one less than that of the usable page and because the benchmark requiring half of the buffer content to be popped, creating the need for a non-odd sized buffer maximum.

Buffer data was set to be characters sized one byte, meaning that in a buffer with a 4096-byte usable page size, the buffers would be able to be filled with 4094 characters, since as mentioned earlier, it is one less than maximum and a non-odd number.

4.1 Data collection

The data was collected on a 64-bit system with AMD A8-6410 and 4GB DDR3L ram on Windows 2004 & MX-18.3. The D-language compiler was instructed to run in release with -nobounds optimization. The windows implementation was run on the LDC compiler while for Posix and Linux the DMD compiler was used.

4.2 Results

The first benchmark handles the construction and deconstruction of the buffers. Times are taken in microseconds. See Figure 23 below.

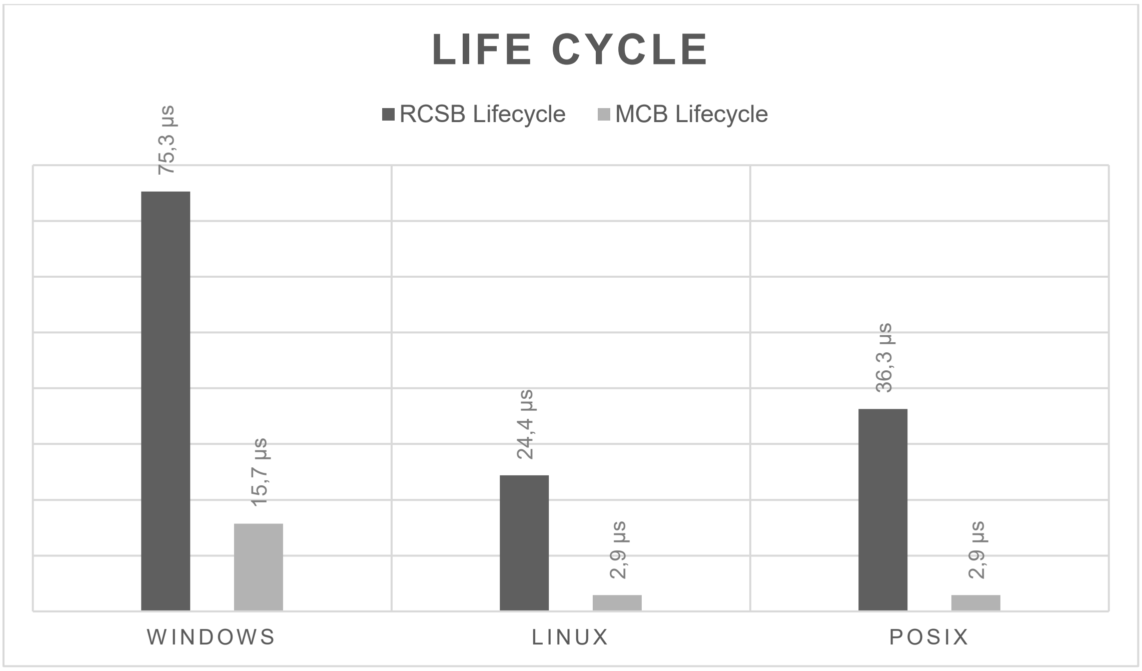


Figure 23: Benchmark Lifecycle in Microseconds.

In the results, it is clear that the MCB is much lighter to construct than the RCSB. On Windows, the difference was almost five times as much. While the Windows results cannot be compared with the other lifecycle benchmarks, they also show similar results. On Linux the difference was over 8 times while on Posix it was well over 12 times.

Both the Linux and Posix benchmark were run on the same operating system and compiler, but by using only Posix compatible system calls, the construction performance lessened by almost a third from 36,3 to 24,4 microseconds.

The usage benchmark consists of writing and removing items from the buffers. See Figure 24 below.

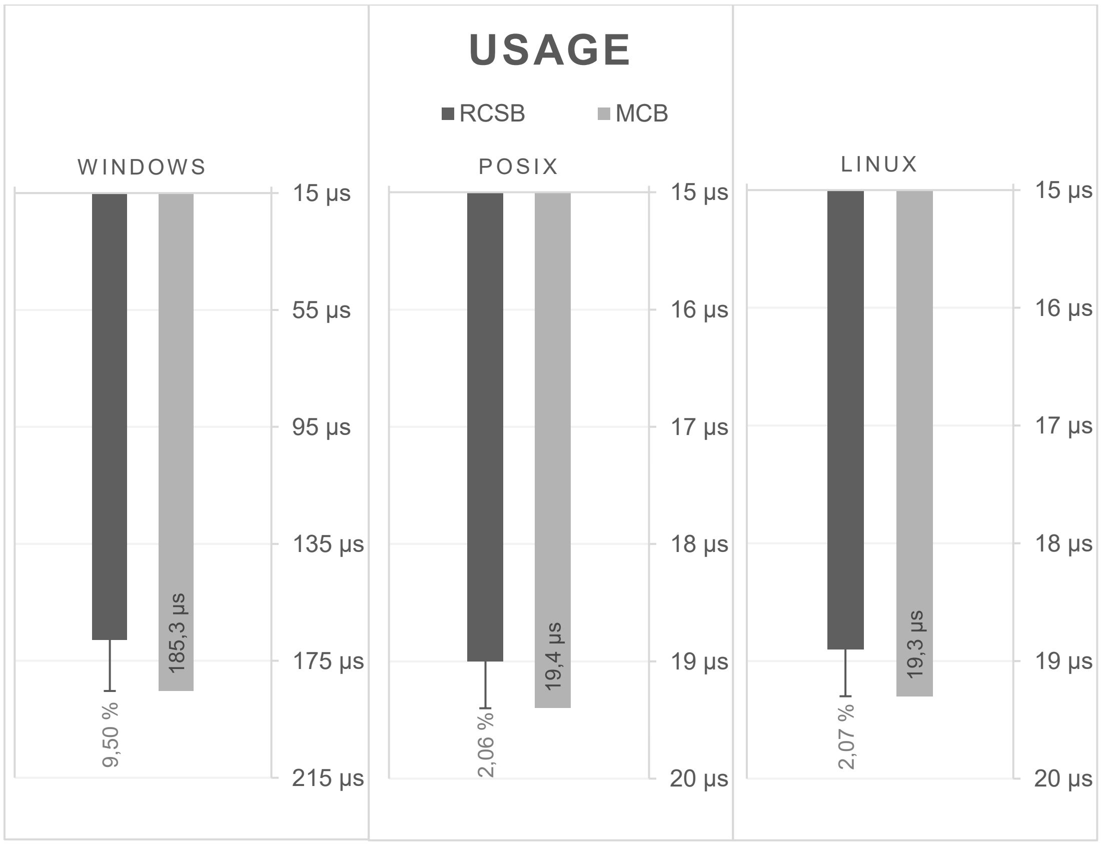


Figure 24: Benchmark Usage in Microseconds.

While construction of the RCSB is considerably slower, it makes up for it with performance of its usage. On Windows the difference in usage performance was 9.5% while on others it was a little over 2%.

Since the buffer usage is identical regardless of whether or not Posix compatible system calls are used in construction within a Linux system, the performance differences are likely statistical errors.

It is quite common for construction performance to be disregarded since a buffer can be created when the program starts, and it can be used for the program's whole lifetime. In these cases, the performance benefit of reusing the buffer can outweigh the additional time used for constructing the buffer.

4.3 Reliability and Validity

As discussed in earlier sections, the Windows benchmarks should not be compared with the other benchmarks since they use a different compiler and are not built to be comparable. The Windows implementation requires a larger buffer size which will also affect the comparability of buffers.

In addition, different operating systems have different performance characteristics. While in the benchmarks the term “Posix” was used, it is indicated to compare to the same operating system as used in the Linux benchmark, but with Posix compatibility. Running Posix compatible code on another operating system that supports Posix system calls will likely result in a different result.

There may also be better, more optimized ways of implementing the buffers indicated on the paper, such as using compiler -specific optimizations. In these cases, the results may differ considerably.

5 Conclusions

The study described pragmatic examples of how both the circular buffer and memory copying buffer could be implemented, including code path differences for the code on different systems.

In the study it was found that while a memory copying buffer has a more simply managed life cycle, it fails to perform at the same level as a circular slice buffer in its usage. Meaning that a properly reused circular buffer would outperform a memory copying buffer.

It was also noted that the circular slice buffer benefits from optimizing its usage instead of its life cycle. This is because life cycle code needs to be run only once while usage code multiple times within the buffer`s lifetime.

The circular slice buffer is time-consuming to construct compared to copying buffers. To further mitigate this, the construction of the buffer can be done in preparation at the start of a long running program.

It would be interesting to see whether circularity is feasible, and to what extent, within software that are currently employing slices and how can buffering be further improved.

References

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

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

Bose, Shreya. 2022. Performance Testing: A Detailed Guide. [online] Available at: https://www.browserstack.com/guide/performance-testing. Accessed 30 Sept 2022.

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

FIFO Architecture, Functions, and Applications.1999. [online] Available at: https://www.ti.com/lit/an/scaa042a/scaa042a.pdf. Accessed 24 Jan 2022.

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

Gregory E. Allen, Paul E. Zucknick, and Brian L. Evans. 2006. Zero-copy Queues for Native Signal Processing Using the Virtual Memory System. /online/ Available at: https://users.ece.utexas.edu/~bevans/papers/2006/zeroCopyQueues/ZeroCopyQueuesAsilConf2006Paper.pdf. Accessed 4 March 2021.

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

Horvàth, Imre. 2007. Comparison of three methodological approaches of design research. [online] Available at: https://www.designsociety.org/download-publication/25512/comparison_of_three_methodological_approaches_of_design.Accessed 2 Sept 2022.

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

Paavola, Antoni. 2022. Reduced Circular Slice Buffer. Laurea BsC. [online] Available at: https://urn.fi/URN:NBN:fi:amk-202205057475. Accessed 20 Aug 2022.

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.