Comparing two solutions to writing real-time audio applications in C++

UNIVERSITY *of* LIMERICK

OLLSCOIL LUIMNIGH

Harry van Haaren

09005201

Faculty of Science and Engineering

Department of Computer Science & Information Systems

University of Limerick

BSc in Music Media and Performance Technology

Submitted on the 18th of April, 2013.

Declaration

I herewith declare that I have produced this paper without the prohibited assistance of third parties and without making use of aids other than those specified; notions taken directly from other sources have been identified as such. This paper has not previously been presented in identical or similar form to any other Irish or foreign examination board.

This work was conducted under the supervision of Dr. Chris Exton at the University of Limerick.

Limerick, 2013

Acknowledgements

I would like to thank my family and housemates for keeping me in upbeat spirits, and motivating me to continuing working even when complications arose.

A special thanks to my supervisor Chris Exton, particularly for asking the tough questions regarding the structure and research topic of this report right at the start: I've learnt a lot about planning research from his guidance.

Contents

List of Figures

1

Introduction

Real-time software is software where the element of time is critical to its successful operation. An example is DJ software. If the audio output is not produced when it is needed to be played, it can be said the software has failed its purpose. It follows that real-time software must be written to be deterministic in the time it takes to produce its output. This fact has implications both for the architecture of the software as well as its implementation.

Overview

This project compares two C++ implementations of a real-time audio file player, in search for a better implementation to create live performance software. Different methods of storing data and inter-thread communication are analysed to determine which solution is more appropriate for real-time audio programs.

In C++ many function calls result in an unbounded execution time: the most common are the `malloc` and `free` memory functions. Thread synchronisation calls like locking mutexes can also cause unbounded execution time. These functions are known as non-real-time functions or blocking functions. Reliably producing audio output within a certain amount of time implies that no blocking functions can be used in the real-time thread.

Memory Management

In order reliably produce audio output it is necessary to remove all non-real-time function calls from the audio thread. As we cannot allocate memory in real-time it is necessary to pre-allocate the memory, or allocate it in another an offline thread and pass the resulting allocated memory into the real-time context. As live performance often involves improvisation it is not possible to rely on pre-allocate memory alone, so it is necessary to use a dynamic memory management model. The management of memory in real-time is one of the core components of this research.

Inter-thread Communication

As data cannot be read and written concurrently by two threads, we need a method of transferring data between threads. Ringbuffers can be used to safely write and read data from two seperate threads. Ringbuffers allow us to safely send move data between threads. It follows that we need to define a format for a message to send events between the threads. Defining what this message will consist of is the inter-thread communication component of this research.

Summary

This research focuses on two solutions to inter-thread communication and memory management in real-time, comparing them under a number of headings to indicate the better solution. The output of the research is which of the two methods of implementing memory management and inter-thread communication is superior.

2

Background

Real time programming is a form of programming where the time taken to process input and produce output is bounded. This is a distinction from general programming where the execution time is not an indication wether the system is working or failing. In the paper "Toward a Discipline of Real-time Programming", it is noted that real-time programming would be better described as "processing-time dependent" programming (Wirth 1977).

2.1 Definitions of Real-time

The term real-time is slightly ambiguous as three different types of real-time systems exist: hard, soft, and firm real-time. The two most common types of real-time system are hard real-time and soft real-time. The following sections describe in detail the requirements of hard and soft real-time.

2.1.1 Hard Real-time

The definition of a hard real-time system from the book "Real-time systems design and analysis: an engineer's handbook" by Phillip A. Laplante is as follows:

> A hard real-time system is one in which failure to meet a single deadline may lead to complete and catastrophic system failure.
>
> (Laplante 1993)

Hard real-time is when a system must be theoretically proven to always, regardless of state, act on an input signal within a certain amount of time. Examples of hard read-time systems include ABS brakes on cars and electronic ignition of a combustion engine.

The hard real-time requirements require documentation of the entire system with regards to timing and delay in reaction of each component. A function exists to determine if a system falls into the category of hard real-time systems in the paper "Real-time digital signal processing: implementations and applications" (Kuo et al. 2006). This is the function:

$$f_M \leq \frac{f_s}{2} < \frac{1}{2(t_p + t_o)}$$

The fastest signal that a real-time system can process sample-by-sample f_M is less than or equal to half of the sampling rate of the system f_s which is less than the time taken to process an input sample t_p and the time for the hardware to output the sample t_o.

The multiplication of $t_p + t_o$ by two is necessary due to aliasing taking place if frequencies higher than half of the sampling rate are present.

2.1.2 Soft Real-time

In Laplante's book the definition of soft real-time is as follows:

> A soft real-time system is one in which performance is degraded but not destroyed by failure to meet response time-constraints.
>
> (Laplante 1993)

Audio software applications that musicians use to perform live exhibit soft real-time constraints. A situation where a DJ is playing music for an audience is a good example: The DJ does not want his audio to fail due to a software error, but if it does fail there will be no deaths. Hence performance software is said to be soft real-time. There is no equation to determine if a soft real-time system adheres to the timing deadlines as there is for hard real-time systems, since missing deadlines is not a catastrophic result.

2.2 Audio Programs and Real-time

This research gears towards the use case where missing a deadline does not result in a catastrophy, but reliable operation of the program at a low latency is desired.

In the domain of software that is used for live audio performances, it is essential that the output is always available when needed. If the software cannot produce the output in due time, it can be said that the software failed its purpose of providing a constant stream of audio to the audience. Similarly if radio or television broadcast software glitches, it can be said to have failed. The real-time requirement of these use cases forces to the programmer of the software to write real-time safe code. It follows that the software written must be designed to suit real-time requirements.

2.3 Programming and Real-time Constraints

Any operation that could take an unbounded amount of time should not be performed in the real-time context. It follows that most system calls, most library calls and almost all thread synchronisation calls cannot be performed in the real-time context. Unless it is explicitly stated that a function is real-time safe, it is best practice to assume it is not real-time safe, and hence should not be called in a real-time thread.

Since using mutexes and allocating memory is a common part of non-real-time programming, different methods of achieving the same functionality must be used to write real-time safe code. There are many functions that cannot be called in real-time. The following is a non exhaustive list of common functions that are described to be non-real-time safe in Bencina's web article:

> malloc, free, printf, pthread_mutex_lock, sleep, wait, poll, select, pthread_join, pthread_cond_wait. (Bencina 2011)

For example memory can be allocated and freed in a different thread, moving the non-real-time operations into a non-real-time context, and passing the result of the operations into the real-time context.

Thread synchronisation is a harder problem to solve, as a real-time thread cannot block on a mutex. Ringbuffers can be employed to transfer data between threads, and this data can contain the information that would otherwise be protected by the mutex. This never synchronises the threads, but allows for data to be passed between them. This method using ringbuffers to communicate is explored further later in the report.

2.4 Real-time Programming Aids

There are a multitude of different frameworks and libraries that exist to aid programming real-time applications. The following frameworks all focus on a certain domain which involves real-time data processing. CLAM (Amatriain 2004) was created for audio analysis, Dempsy (Nokia 2012) is suited to stream-based distributed processing and FAROS (FAROS Consortium 2007) aims at performing machine automation. The Raul library (Robillard 2008) offers a set of utility classes that makes writing real-time code easier by providing various high-level structures like real-time safe arrays and lists.

Although these frameworks aid the developing of the software initially, they may hinder the developer later when they must verify the real-time capability of their software. As a general rule keeping the real-time critical code simple while using as few library calls as possible is advised by the author. It follows that using frameworks and libraries to aid real-time development is not usually effective unless the developers are extensively familiar with the framework or library in question.

3

Research Question

The question that this research answers is how to best implement real-time audio applications in C++. As presented in the introduction section, two primary issues arise when writing real-time code. The two issues are memory management and inter-thread communication.

The memory management issue arises as its not possible to call `malloc` or `free` from a real-time thread. By consequence inter-thread communication is necessary to provide or remove memory to or from the real-time thread.

This research investigates two types of memory management and two types of inter-thread communication. They are described in detail below.

Test Implementations

In order to compare the different methods of achieving memory management and inter-thread communication it was decided to create a showcase application. This showcase would provide functionality to play an audio file as a loop. Loading of audio files requires a non-real-time thread because it allocates memory. It follows that the resulting memory must be passed to the real-time thread, which would exercise the inter-thread communication system.

Two implementations are created: both with different methods of memory management and inter-thread communication. The details of which implementation uses which system is detailed below. Troughout the rest of the report, references will be made to implementation one and implementation two.

Memory Management

Two implementations of memory management are investigated. One relies on raw pointers, and manual management of these raw pointers. The other uses a smart pointer from the Boost library. This smart pointer, called a `shared_ptr` holds a reference count to the object, and deallocates it when it is no longer needed by the program.

Implementation one uses raw pointers and manual management, while implementation two uses the Boost `shared_ptr` method.

Inter-thread Communication

This research also analyzes two types of inter-thread communication. On method is to create a simple C++ `EventBase` class. Each event then inherits from EventBase, and represents a certain type of functionality within the program.

The second method uses the Boost C++ library for inter-thread communication. "Boost Function is a set of class templates that are function object wrappers" (Gregor 2001). It follows that a function call can be wrapped into this Boost Function object, and then executed in another thread. The function being called is written to handle the event in question, and hence this method of passing data through the ringbuffer provides a workable solution. Another way to achieve the same functionality is to pass an `Event` class through the ringbuffer.

Summary

This research compares memory management and inter-thread communication system of two implementations of a real-time audio program. The two test implementations are detailed below.

```
Implementation one:
 - Manual memory management
 - Event based inter-thread communication

Implementation two:
 - shared_ptr based memory management
 - Boost::Function based inter-thread communication
```

4

Materials

This chapter details the materials used for the research. It discusses the different tools and libraries used, and why those particular ones were chosen. Where possible open-source libraries and programs were used for this research. The choice to use open-source projects originates from the authors experience with the linux platform and open-source tools. The extensive configurability of linux as a platform affords tuning the system to the real-time requirements.

4.1 GNU/Linux

GNU/Linux is an operating system. The operating system functionality is built around the Linux kernel and many utility projects, of which GNU provides a significant portion (Stallman 1997). The following sections detail the different aspects of using Linux for real-time work.

4.1.1 Hardware

In order to achieve extremely low latencies it is necessary to use specialised hardware. The authors laptop has an integrated Intel HDA chipset based soundcard. This is a consumer grade soundcard, and is not able to handle latencies as low as is desirable for real-time audio performance programs.

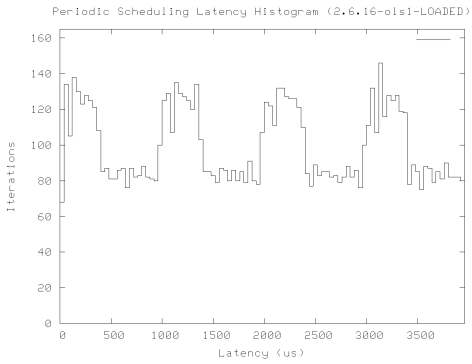
For this research a specialised soundcard is used as it allows lower latencies. It follows that testing the performance of the implementations can be done at

lower latencies, providing real-life results at genuinely low latencies. The actual devices used is an Echo Indigo DJ PCMCIA soundcard. Configuration of the system to prioritize the device is described later on in this section.

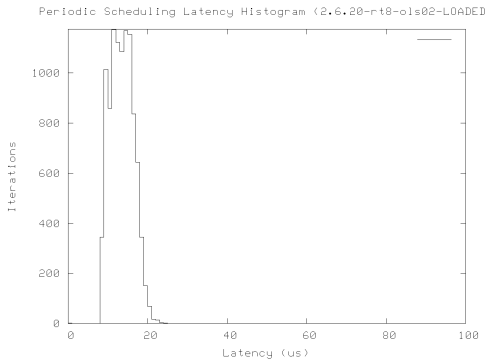
4.1.2 Kernel

The kernel is the core part of the operating system which schedules each program to run. Since about 2000, there have been efforts put towards making the kernel more preemptive. Clark Williams' presentation at the Linux Kernel Summit (2008) shows that the time between scheduling a program to run and it actually running is much more stable in a `-rt` (real-time) patched kernel than a vanilla kernel. Measurements of the performance of the 2.6 vanilla kernel versus the -rt kernel have been made, with their graphs of the acquired data shown below.

The histograms Fig. 4.1a and Fig. 4.1b show that the `-rt` kernel is considerably more consistent in scheduling tasks than a vanilla kernel.



(a) Scheduling latency of a vanilla kernel (Rostedt and Hart 2007a).



(b) Scheduling latency of a -rt kernel (Rostedt and Hart 2007b).

4.1.3 Interrupts

This section describes the setup stage where the interrupt requests for the audio hardware. When the audio hardware sends an interrupt to the operating system it signals that the start of the audio processing is to begin (Anderson 2009).

The signal indicates that the audio hardware has input data to pass to the operating system, and it wants data passed to it to play back. In order to achieve

reliable low latency, the interrupt signal from the audio hardware needs to be reacted to before similar signals from other hardware like printers and graphics cards.

Prioritising interrupt requests can be achieved manually or by using automated tools like Rui Nuno Capela's `rtirq` (2012).

Top and Bottom Halves

Interrupt requests are made up of two components in `-rt` kernels: top halves and bottom halves. Bottom halves are also known as tasklets. The IrqPriorities page on ffado.org describes how the functionality of the interrupts is divided:

> For efficiency reasons, interrupt handlers in Linux are divided into top halves, which acknowledge that an IRQ was received (so that the device that issued it is at peace), and bottom halves, where the actual work gets done. Bottom halves can be delayed if the kernel has more pressing things to do.
>
> (FFADO.org 2010)

These bottom halves can be scheduled with `rtprio` just like ordinary tasks. It follows that for an audio-tuned setup we must ensure that the audio hardware's interrupt request is being serviced before other interrupt requests.

Tasklet Priority

To set the bottom half (or tasklet) priority, we first identify which kernel modules are needed for the audio hardware. Since the soundcard used for this research uses PCMCIA, the PCMCIA kernel module `yenta` needs to be prioritised. The kernel module `snd_indigodj` is the ALSA audio driver for the soundcard, so that needs to be prioritised too. These two kernel modules are entered in `/etc/conf.d/rtirq`, the rtirq config file.

On boot rtirq automatically runs, and sets the interrupt priorities. When adding the soundcard when the system is already powered on, it is necessary to run rtirq manually to setup the tasklet priorities. The following command sets the tasklet priorites correctly using `rtirq`:

```
/etc/rc.d/rtirq start
```

4.1.4 Real-time Priorities

To achieve low latencies without glitches, the operating system must be tuned for optimal audio performance. There are many different parameters that affect audio performance, however it is essential to remember that reliability is key while performance is only desired. To schedule a task to run ahead of other tasks a property called `rtprio` is used. It stands for "real-time priority".

If task A has a `rtprio` of 89, while task B has an `rtprio` of 50, then task A will be run before task B. For example task A is the audio processing task at `rtprio` 89, and task B is an internet browser at `rtptio` of 50. In this situation the operating system will run the audio processing task before scheduling the internet browser. This is desired as it ensures that if CPU time is available, it will be dedicated to audio processing before redrawing HTML pages.

Unfortunately using `rtprio` also has disadvantages. The danger of using `rtprio` is that if the real-time task locks itself in an infinite loop, other programs cannot get CPU time. If the situation gets out of hand, it is even possible that the task can't be killed because the task with `rtprio` is continuously scheduled before the task that is trying to kill it. For this scenario, it is common to schedule a "watchdog" process, which gets an `rtprio` even higher than the real-time process. The watchdog does nothing except poll the real-time task, and if the real-time task enters an infinite loop locking the whole machine, the watchdog kills the RT task automatically.

4.1.5 Setup Issues

Although every effort has been made to ensure that the CPU will always schedule the audio processing tasks first, there are still issues. The author has noticed that scanning for WiFi networks often causes a glitch in audio processing. For this research turning off the WiFi hardware is a practical solution.

Finally the tool `realTimeConfigQuickScan` can be run, to perform a wide range of checks to analyse if the system is optimally tuned for audio processing work.

4.2 C++

C++ is a general-purpose, platform-neutral compiled programming language. (Kalev 2010) It is used for many audio processing programs including the proprietary Ableton Live (Henke 2004), the hard disk recorder Ardour (Davis 2004) and the sound file editor Audacity (Dannenberg and Mazzoni 2013).

It is considered a an intermediate level programming language, as it contains both high and low level elements. The low level components allow the programmer to write real-time safe code, using only functions that the programmer knows are real-time safe. The high level elements allow for the easier implementation of functionality and reusing libraries, which is ideal for creating graphical user interfaces and doing work that is not time critical. The combination of these elements and the ability to interact with any C library makes C++ a good candidate for real-time programming.

4.2.1 Manual Memory Management

The fact that C++ allows the programmer to manage memory allocation and deallocation manually lends itself to writing real-time code. Since memory allocation functions like `malloc` are system calls which doesn't have an upper bound for return time, it is not safe to allocate memory in any real-time context. This issue can be resolved due to the C++ affording the flexibility of allocating memory manually (allowing this operation to take place in a non-real-time context) and then passing the resulting memory into a real-time context.

4.3 JACK

JACK is an audio server. It takes control over the sound hardware in the computer, and provides an interface for other programs to playback audio data through JACK. It is designed to keep latency to a minimum and keep all streams in sync with sample accuracy. These are features that are essential for a professional audio environment. JACK can use different audio subsystems like ALSA (Kysela 2007) or FFADO (Palmers 2007), allowing it to use a wide range of audio hardware using either PCI, USB or Firewire interfaces.

An overview of its inter-operation with the previously mentioned backends can be found here. (Davis 2003) Each backend has its own impact on the lowest latency that JACK can run at, the author uses an RME Multiface II with PCMCIA adapter, and a PCMCIA Echo audio Indigo DJ interface.

4.3.1 Callback Design

The design of JACK is one based on a callbacks. Callbacks are a solution to "uncouple the caller from the callee" (Codeguru.com 2005). Its purpose is to allow the callee to idle, and the caller's thread does the work in the callee functions. This design is known as a "pull" model, and is widely used in low-latency audio APIs (Benton 2013).

One advantage of JACK calling the client's `process()` callback is that the JACK thread already has the appropriate real-time priorities. This removes the real-time thread scheduling code from each client, and instead JACK takes care of it once. This ensures that the audio processing thread is scheduled above other program threads. It also ensures when multiple programs are running within the JACK graph that each program runs at the same priority, avoiding situations where programs attempt to steal priority from each other.

4.3.2 JACK Ringbuffer

A ringbuffer is a structure which allows two threads to safely interact with it at the same time. A read thread can read available data, while a write thread can place more data into the structure. The cross thread real-time communication methods in this research are based around ringbuffer usage.

Ringbuffers operate by having a separate head pointer and tail pointer (Tyson 2011). There are many terms for the components of ringbuffers: head and write pointer refer to the same component. Similarly the tail or read pointer are also synonymous. The write pointer adds data to the ringbuffer, and then the read buffer consumes it.

Writing new data into a ringbuffer is done atomically: that implies that it is thread safe, and does not use any locking. It follows that read operations are also atomic, and hence reading data from a ringbuffer is also real-time safe.

Note that the JACK ringbuffer is safe only with single-reader, single-writer threads (Davis and Drape 2000). This means that the ringbuffer can only be used for one direction of communication between threads. It follows that two ringbuffers are often used, one for each direction of communication between the threads.

4.4 JACK-Interposer

Jack-interposer (Engelen 2007) is a shared object which hijacks library calls that are not real-time safe. By using the `LD_PRELOAD` environment variable we can force the use of the jack-interposer library during run time. JACK-interposer usually passes the hijacked function calls on to the actual library implementations. There is logic in place which allows JACK-interposer to detects if the JACK real-time thread is the currently executing thread, and if it is any of the hijacked library calls will cause sending of the `SIGABRT` signal.

The use case for this is that the implementations of jack-interposer's functions are all to halt the program. This allows a debugging program like `gdb` to halt execution, and provide a backtrace showing why that function was called in a real-time context.

Function calls that are not real-time safe are not intercepted when running in a non-real-time thread as that a thread local boolean value toggles when entering JACK's `process()` function, and that enables sending `SIGABRT`.

By allowing to programmer to test while JACK-interposer aborts the program on any real-time violation, is an immensely valuable tool to the real-time programmer. It is the best error checking tool that exists a programmer can use for verifying that a JACK client program is real-time capable.

4.5 Git

Git is a program for source code revision control (Hamano and Torvalds 2013). It was initially designed to handle the development of the Linux kernel, and has since seen widespread use. Due to its distributed architecture and automatic merging of branches of code, it is suited to allow working on the same code

from different computers. This afforded my intended workflow of doing some programming and testing a laptop while also working on the same codebase from a desktop computer.

As a method of synchronising the code on the separate computers a bitbucket repository was used to push the code to. The other computer could then pull the code from the repository again. This allowed working from two computers collaboratively without any synchronisation issues.

4.6 Make

Make is a build tool that makes the process of compiling programs easier. It was chosen for its configurability and widespread use. The authors familiarity with the tool also made it a logical choice.

4.7 GProf

Gprof is a code profiler. It can be used to determine where a program spends most of its processing time, and also which functions are called most often. To collect these statistics gprof inserts code into the program during compilation, which affects run time performance.

Gprof outputs a flat profile and a call graph after the program being profiled has exited. A flat profile details the amount of time spent per function, while the call graph informs us of which functions call each other (Linux Journal 1998).

5

Methodology

This section describes the methodology used for this research. Firstly the system setup stage is described. The two different real-time implementations are then explained. Thirdly the method of collecting results is detailed. Finally possible causes of error in collecting results are brought forward.

5.1 Setup of System

Upon starting the project the laptop intended for this project had an installation of Arch Linux. That installation was an ordinary install, without any specialised setup for real-time work. The following sections describe the setup phase of the project.

5.1.1 Real-time Kernel

In order to allow JACK to run reliably at very low latencies, a series of adjustments and changes needed to be made. The main goal of the mentioned changes is to make the kernel scheduler give priority to the audio processing hardware, the interrupts for that hardware, and the threads that handle those interrupts. Installing a real-time kernel in Arch Linux is a matter of typing a command:

```
pacman -S linux-rt
```

At the time of installing, the command installed a real-time kernel of version `linux-rt 3.6.11_rt25-1`. Following the installation of the kernel, it was set to be the default kernel to be booted when powering on the laptop. This can be achieved trough editing the `default` value in the configuration file `/boot/grub/menu.lst`.

5.1.2 Real-time Scheduling

As described in the materials section, there is a lot of setup to be done for an optimised real-time operating system. The rtirq script initialised the correct interrupt request priorities for the audio hardware. It has been set to run after the computer has finished booting. Manually re-running is possible by executing the command `/etc/rc.d/rtirq start`.

5.2 Implementation One

The first implementation uses a class as the object passing through the ringbuffer, and have data owned by the thread that can access it. This implies that in order for the user interface to show the data currently being played back, that data must be duplicated in memory.

5.2.1 Class Structure

The class structure of the program is shown in Figure 5.1. A Top class exists, and top has a `GUI` and `DSP`. There is no class to represent the audio data, as it is allocated on the heap and not owned by a class.

5.2.2 The Event Class

This implementation uses sub classes of the EventBase class for message passing. The concept of this idea was first introduced to the author on the linux audio development mailing list. (Beddingfield 2011) It involves writing a base class that provides pure virtual functions for retrieving the type and size of the event. This interface is used to read `size` bytes from the ringbuffer, and then the `type` variable is used to decode the event type. Once the type of the event is known,

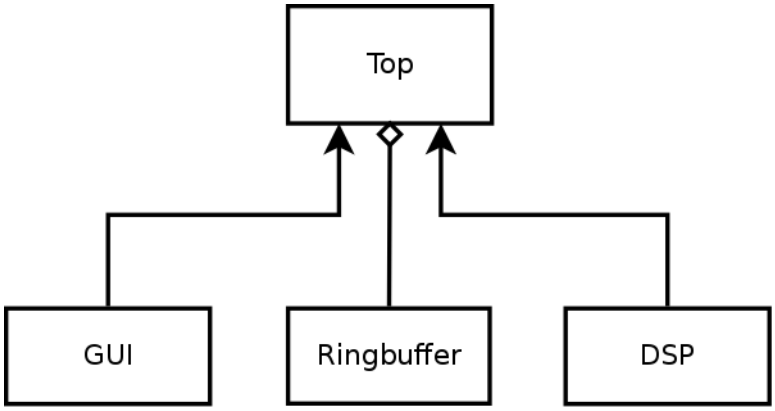


Figure 5.1: Class Structure of Implementation One - Top is the main point in the program, and the GUI and DSP classes write events to the ringbuffer trough a global `Top` pointer.

the EventBase data can be cast to the correct type, allowing the retrieval of information specific to that event type. Once the specific event data has been extracted, an ordinary function call can be made to achieve the desired result.

5.2.3 Memory Ownership

The memory management in this implementation is very primitive. Data is allocated on the heap in a non-real-time context. A raw pointer to this data is then passed through an Event to the `DSP` instance. The `DSP` instance is now said to have ownership of the data. While it owns the data it can purpose it until it is no longer needed. When that situation arises, the raw pointer is passed back to the `GUI` for deletion.

This method requires data to be duplicated, however there is no possibility of concurrent threading issues arising since both the `GUI` and the `DSP` have their own copy of the data. This may lead to issues of keeping the data in sync if processing is to be performed on the data. This is not the case for the test implementation in this research, as data is only played back.

5.2.4 Ringbuffer Issues

The issue at hand is that an event could be executed that is only partially written, hence interpreting un-initialised memory. This issue involves two threads reading and writing from the ringbuffer concurrently. The location of the `Event` currently being written must be in a specific place in the ringbuffer. This issue only arises when an exact order of execution takes place, and the above statements hold true. The paragraphs below examine the issue in detail.

The inner workings of the jack ringbuffer must be examined in order to understand this problem. The code in question of the ringbuffer implementation is included in Appendix C for convenience, as is part of the `event.hpp` header to understand the sizes `Event` class instances. The fundamental problem is that if the ringbuffer must "wrap around" to write the event, two separate `memcpy` calls are made.

The logic to check if a ringbuffer "wrap" write takes place is on line 283 of the implementation. The write pointer location is added to the size of the event to be written, and then compared with the size of the ringbuffer. If the size of the event to be written is greater than the end of the ringbuffer, it is necessary to use two `memcpy` calls.

In a hypothetical situation, we wish to pass a `EventBufferTransfer` event trough the ringbuffer. The writing thread runs: the ringbuffer logic must do a "wrap" write for the size of the event. The write thread makes the first `memcpy` call, writing `sizeof(EventBase)`. The remaining size of the `EventBufferTransfer` has to be written yet. Now the write thread gets interrupted, pausing its execution.

The reading thread is now run. It ensures the available read space is at least `sizeof(EventBase)`. This evaluates to true so read thread casts the read pointer memory to `EventBase*`. This is available at lines 106-110 of Appendix A, or the file src/imp1/top.cpp.

It seems logical to now cast the `EventBase*` to `eventBase->type()`, and then handle the event. This would cause the `EventBase*` to be cast to an `EventBufferTransfer*`, and handling it would lead the program to interpret the not-written-to memory location at the start of the ringbuffer. The effect of

this is a garbage pointer being inserted into the program, ultimately leading to a segmentation fault when that pointer is dereferenced.

The solution (as in Appendix A) is to do another size check. This time we compare the available read space against `eventBase->size()`. If this is also true it is guaranteed that the whole event is written, hence it is safe to handle the event. If, however, this check fails there is an else clause returning from the function (see lines 175 - 181 of `top.cpp`). This issue was discussed on the linux audio development mailing list, the thread is available publicly (Donalies 2013).

5.3 Implementation Two

This implementation uses Boost Function for communication between the threads, and the data is immutable and shared. Since the data can be shared between the user interface and the playback engine it need only be loaded once.

5.3.1 Class Structure

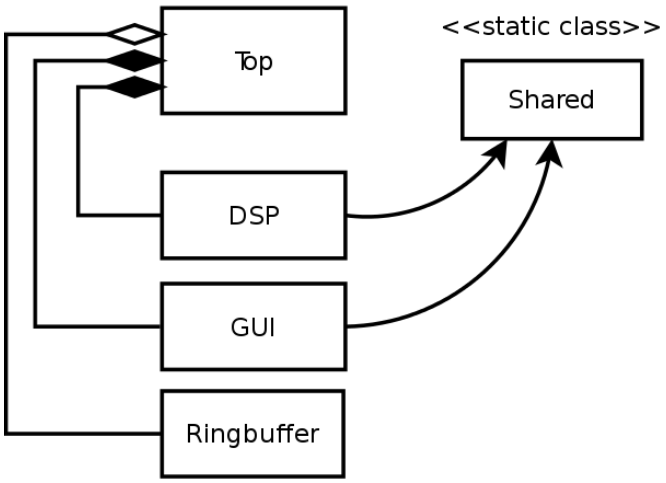


Figure 5.2: Class structure of Implementation Two - Top is the main point in the program, and the GUI and DSP classes write events to the ringbuffer trough a global `Top` pointer. `Shared` is a static class, which holds the `AudioBuffer` pointers. `GUI` and `DSP` read `Shared` using its static `::get()` method.

The class structure as shown in Figure 5.2 details how the class communication is achieved in this implementation. A `Top` class holds pointers to the other parts of the program, allowing the `GUI` to get a pointer to the `DSP` part of the program. This is essential for the use of Boost function objects, as they need to bind to the instance of a class. Two `Ringbuffer` classes are owned by `Top`, that allow message passing between the `DSP` and `GUI`. When used together with the Boost Function objects they provide the entire mechanism for communication between threads for this implementation.

5.3.2 Thread Shared Memory

The static class `Shared` is used as a storage class for the data that the `DSP` and `GUI` need access to. The fact that the data in `Shared` is immutable ensures that there are no thread safety issues with concurrent use of the data. Data is loaded into the `Shared` class by the `GUI`. Since the `GUI` thread does not have any real-time obligations accessing the hard disk and performing memory allocations is not an issue. Once these non-real-time operations are finished, a message is sent to the `DSP` instance informing of a new piece of data in the `Shared` class.

5.4 Collecting results

The following section details the steps taken to collect the data presented in the results section. In order to ensure that the same compiler flags and optimizations were applied to each of the programs scripts were written. These scripts also used conditional compilation macros in order to add the self-testing capability to the test implementations. Scripts were also written to automating repetitive work like generating graphs.

5.4.1 Self Testing Code

In order to collect statistics on the differences between the implementations, they must process the exact same input. If the input to the implementations would differ it would not be possible to draw meaningful conclusions. To make the implementations process the same input, they would callback a function that

would load a sample. This function is also called when the user clicks on screen, so the self-testing code tests the same functionality as would a user.

The advantages of having the code test itself is that the scheduling of function calls can be achieved. This in turn allowed me to schedule a new sample to be loaded every 2 seconds, and then interpret the resulting data with that knowledge. It follows that the memory usage graphs can be analysed and the actions of the program linked with the resulting changes in the graph.

In order to also allow the user to directly manipulate the program, it was necessary to compile versions of the program which do not self-test. In order to not have to manually change the code, `#ifdef` directives were used. These allow the inclusion of code based on if a certain compiler flag is provided. Passing the flag `-DSELF_TEST` to `make` would have the result in the inclusion of the self-testing code. To achieve this functionality using make, it was necessary to override the `CFLAGS` variable in the makefile. Overriding the `CFLAGS` variable is a trivial task: it involves adding `override` to the start of the `CFLAGS` list in the file, and using `+=` instead of `=`. The results are shown below:

`make` command:

```
make CFLAGS="-DSELF_TEST"
```

`CFLAGS` and `LDFLAGS` entries in the makefile:

```
override CFLAGS+=-c -Wall `pkg-config gtkmm-2.4 --cflags`
override LDFLAGS+=-ljack -lsndfile `pkg-config gtkmm-2.4 --libs`
```

5.4.2 Scripting

In order to ease the workflow of collecting data, various scripts were written. These scripts each serve a specific purpose, for example re-compiling the code with different profiling and debugging flags. Apart from ease of use, the scripts also afford a much less error-prone workflow, as forgetting a single compiler flag on a recompile could change the results drastically.

Compiling

The compile script recompiles the code of both implementations, and copies the resulting binaries to the testing directory. The script passes various `CFLAGS` and `LDFLAGS` to `make`. The results are six binaries, one of each implementation that is optimised, debuggable and self profiling.

The script can be found in the `src/testing` directory.

Running Tests

The `runTests.sh` script sets up the testing environment, and proceeds to profile the self-testing build of both implementations. In order to setup the test environment it turns off CPU-scaling, ensures that the soundcard interrupt priorities are in order and turns off the WiFi adapter. It then launches the showcase applications one by one, while polling their memory usage. The showcase applications themselves write profiling data to a file, which is then interpreted and plotted as a call graph. Finally, gnuplot is called on the memory data creating plots of memory usage of each implementation over time.

The script can be found in the `src/testing` directory. It generates four image files in the `src/testing/output` directory.

Gathering Data

The `gatherData.sh` script scrapes together data about the current real-time environment of the system. Various system files like `/proc/interrupts` are read, showing which interrupt requests are being prioritised. The `rtirq` tool is also consulted, providing a list of real-time interrupt request tasklets. Interpreting the order of the list shows if the audio hardware is prioritised, and hence if the current situation with regards to interrupt request priorities is appropriate for low-latency audio.

The script can be found in the `sysStats` directory. It outputs data to three text files in the same directory as it is situated.

5.4.3 Stress Testing

This section details how the system load is increased using a tool called cpuburn (Manpages 2013). The tool attempts to generate 100% load on the target processor core by looping work on the floating point unit (FPU) and algorithmic logic unit (ALU) in the processor.

The loops are coded in assembly with the goal of causing maximum CPU usage. Different versions of the cpuburn program exist, each is optimized for its own target processor architecture. The `burnP6` is best suited to the authors processor. Running the `burnP6` command will launch one instance of cpuburn, causing 100% load on the CPU core that it is being executed on. In order to test the stability of each implementation two instances were launched on the host machine as it has a dual core processor.

The `cpuburn` program is ideal for testing audio processing applications as the primary activity of audio-processing is adding and multiplying floating point values. As `burnP6` generates maximum load on the FPU, it stresses the processor, the scheduler and real-time program all at once. It follows that if the program doesn't glitch while the scheduler and the processor's FPU are being stress tested, the program holds up in the most demanding situation.

6

Results

This section presents the results of the research done. There are two main outputs: two real-time programs, and data comparing these programs. The graphical user interface is detailed, then the details of the implementations are brought forward.

6.1 Graphical User Interface

This section explains the different parts of the graphical user interface shown in Fig. 6.1. Firstly, on the left there is a list of files: these are audio samples that the program can play back. The top shows the waveform of a loaded audio sample, while below that is shown information about the current actions of the code.

Clicking on a file name on the left causes it to be loaded from disk into memory, and played back. Although this step seems trivial we will see in the next section that complex code executes and inter-thread messaging takes place in order to achieve the goal in a real-time safe way.

6.2 Explanation of Run Time

The run time interaction with the program allows showcasing that the inter-thread communication is working. The real-time thread runs the audio processing, which the user perceives as sound. Meanwhile the GUI responds to user input, and doing the non-real-time work like loading samples.

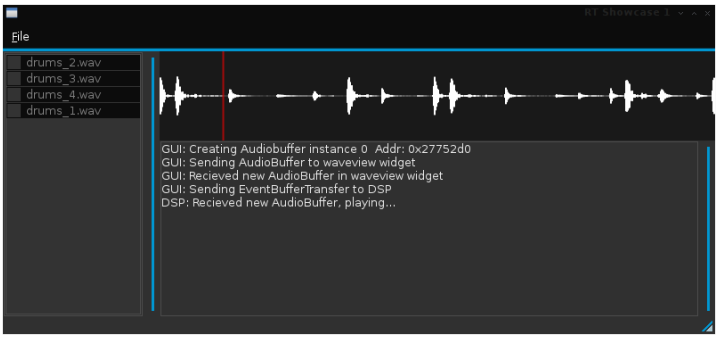


Figure 6.1: Screenshot of User Interface - Audio samples are shown on the left, a waveform is plotted and the current activity of the program is printed.

Although the user cannot see which thread runs where, the experience of glitch free audio and a responsive user interface demonstrate this fact. The graphical display of the waveform demonstrates that the audio sample data is available not only in the real-time audio thread, but also in the `GUI` thread.

6.2.1 Analysis of Audio Processing

This section analysis the actions that the program takes while performing the audio processing. The showcase programs send the user interface an update of the current playback position once every audio frame.

Implementation One

The graph shown in Fig. 6.2 represents the functions that are called while the program does one block of audio processing. The start of processing occurs when Jack calls the programs `process()` function. The `process()` function calls `Top::processDSP()`, a function which reads any available events from the ringbuffer and handles them.

Now the audio is processed and the output written, and a playback position message is sent to the GUI using `Top::guiToDspWrite( EventBase* e )`.

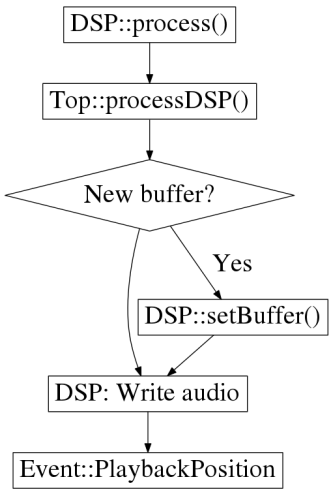


Figure 6.2: Process Callgraph of Implementation One

Implementation Two

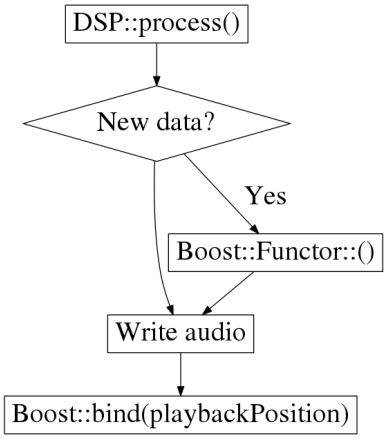


Figure 6.3: Process Callgraph of Implementation Two

The graph shown in Fig. 6.3 shows what function calls are made when processing a block of audio in implementation two. Firstly the ringbuffer is checked if readable data is available. If data is available the `Boost::Function` contained in that data is executed. The audio output is then computed, and finally an event is written to inform the `GUI` of the new playback position.

The implementation details of writing the new playback position are of high importance. In order to wrap the function in question, we call `boost::bind`. This function returns a `boost::function` object, which is transferred trough the ringbuffer to the `GUI`.

The `boost::bind` call takes the address of the function with regards to the instance it should be called on, a pointer to the instance, and any parameters that the function takes. The following is the declaration of the function to wrap:

```
void GUI::playbackPosition( float pos );
```

The function name we want to wrap is `playbackPosition`, and it takes one float argument. The `DSP` class can get a reference to the `GUI` instance by calling `top.getGUI()`. Using the address of operator `&` we can provide a pointer to the instance. Hence the final call to `boost::bind` is:

```
boost::bind( &GUI::playbackPosition, &top.getGUI(),position);
```

This function call returns a `boost::function` object, which is then passed to the `write()` method of the ringbuffer.

It seems like everything here should work in real-time as there are no calls to `malloc` or `free`, no mutexes or spinlocks, and no system calls. This assumption would lead to a broken real-time system as the fact is that when creating a `boost::function` using `boost::bind`, `malloc` is called behind the scenes.

The fact that `boost::bind` allocates memory when being written rules it out of being used to wrap functions in real-time. It is a necessary element of our real-time program to write events to the `GUI`, therefore the `Boost::Function` method of communication is fundamentally flawed.

6.2.2 Analysis of Loading a Sample

This section details the exact operations that take place when loading an audio sample for playback. As the loading of a sample involves allocating memory and accessing the hard disk, it is a non-real-time operation. Hence the `GUI` thread does the memory allocation, loads the sample data into the allocated memory and sends an event to `DSP` that it should update to playback the newly created `AudioBuffer`.

Implementation One

The sample loading as shown in Fig. 6.4 begins with a `buttonClick()`, which calls the `GUI::loadSample()` function. That function allocates the necessary objects like a new `AudioBuffer`, and does the disk access.

The `AudioBuffer` object is then cloned, and one instance is sent to the `GUI` to be displayed. The cloning of the `AudioBuffer` is necessary in this instance as there is no guarantee that the DSP owned `AudioBuffer` will not be deallocated. The data duplication here has the side effect of affording the modification of the data contained within the `AudioBuffer`.

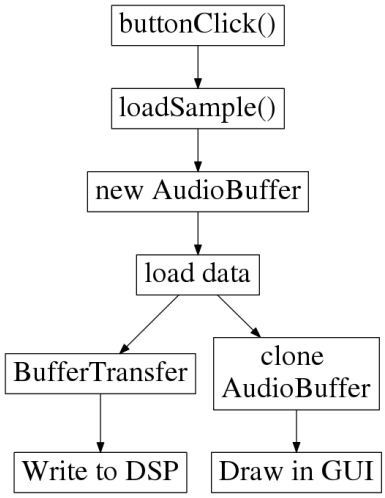


Figure 6.4: Loading an AudioBuffer in Implementation One

The address of the original `AudioBuffer` is then taken by an `EventBufferTransfer` message, which is written to the ringbuffer, and the `DSP` class will use the new `AudioBuffer` for playback when it starts processing next.

Implementation Two

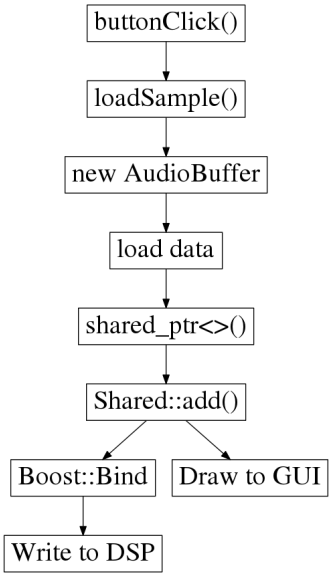


Figure 6.5: Loading an AudioBuffer in Implementation Two

This implementation uses the `Boost::Function` method of inter thread communication, and the `Shared` class for resource management. The call graph Fig. 6.5 shows the function call order when loading a sample. The start is similar to Fig. 6.4 where a `buttonClick()` event starts the chain, `GUI::loadSample()` is called next, and finally a new `AudioBuffer` is created.

The similarities stop there, because this implementation uses a static `Shared` class to keep track of the `AudioBuffer` instances. This class has methods which allow the adding of items like an `AudioBuffer`.

The tracking of items is implemented using smart pointers such as a `shared_ptr`. The `boost::shared_ptr` was used in this implementation. A `shared_ptr` is a template class, allowing it to operate on any type or class type. Internally an atomic integer is used to keep a reference count on how many pointers still exist to the item.

The `Shared` class iterates over all the items and calls `unique()` on them. If `unique()` returns true, the one reference that is left to that item is the one which `Shared` itself has. This implies that neither the GUI or the DSP are still using the `AudioBuffer` in question, and hence `Shared` can clean it up.

Although `Boost::Function` isn't real-time safe due to allocating memory, it is actually fine to use in this thread: this is not a real-time thread. Memory allocations here don't matter, so there is no problem when using `boost::bind` to write messages in the direction towards the `DSP` instance.

The `Shared` class is a method to avoid duplicate data that will not be modified. It tracks items lifetimes, removing the possibility of memory leaks.

6.2.3 Analysis of Removing a Sample

This section details how each implementation handles deallocating a sample. A sample gets deallocated in the test implementations when a new sample is sent to `DSP` to be played. In other words the event that triggers a sample to be deallocated is a new sample being played back.

Implementation One

In this implementation the `DSP` recieves an `EventBufferTransfer` message, containing a pointer to the new `AudioBuffer` to be played back. When this message is being handled, the current `AudioBuffer` pointer is compared with `NULL`. If it currently points to an `AudioBuffer` instance an `EventBufferTransfer` message must be written to move the pointer back to the `GUI`, since deallocating memory is not real-time safe. The code that performs the check is available in Appendix B.

The `AudioBuffer` pointer that is passed trough the `EventBufferTransfer` is deallocated when the GUI reads the `Event` from the ringbuffer. This deallocation is done using the normal C++ call `delete`.

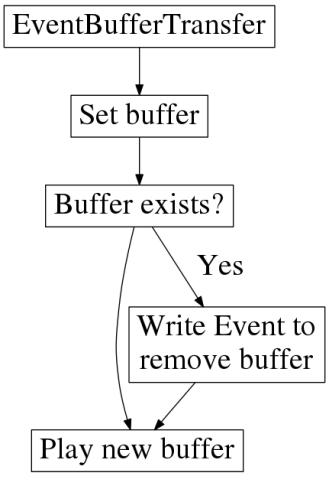


Figure 6.6: Removing an AudioBuffer in Implementation One

Implementation Two

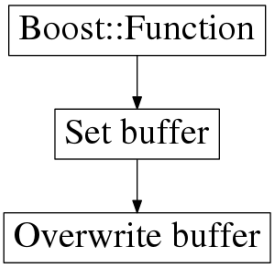


Figure 6.7: Overwriting a buffer pointer

Removing a buffer in implementation two happens much differently than in implementation one. The reason for this is that the `DSP` doesn't have to manually take care of pointers, as this responsibility is on the `Shared` class. Fig. 6.7 shows when a new buffer function call is executed (originating from a `boost::function`, bound in the GUI) it just overwrites the old buffer.

If raw pointers were used overwriting the pointer

would most likely result in a memory leak, however we are dealing with smart pointers here. When the `AudioBuffer` was created a `shared_ptr` to the instance was placed in the `Shared` class. Hence the reference count of the `AudioBuffer` instance remains at one, even when `DSP` overwrites its `shared_ptr` to that instance. It follows that in order to remove an `AudioBuffer` instance from `Shared` we must implement some logic to delete an `AudioBuffer` instance when the reference count is at one, not zero.

Deleting the `AudioBuffer` instances occurs in the `Shared::cleanup()` function. This function is called once every second by the `GUI` thread and iterates over the list of `shared_ptr` objects, calling `unique()` on them. If unique returns true there is only one reference to the object, and it is removed from the list using `std::list::erase()`. The reference count of the object is now at zero, and the smart pointer logic will deallocated the `AudioBuffer` instance.

Note that the deallocation of the object occurs in the thread where the last reference goes out of scope. Hence it is vital to ensure that the `Shared` class always owns a reference. If `DSP` was to overwrite a `shared_ptr` without `Shared` having a reference to the overwritten object, the destruction of the `AudioBuffer` instance would occur in the real-time thread, and thereby cause non-real-time operation.

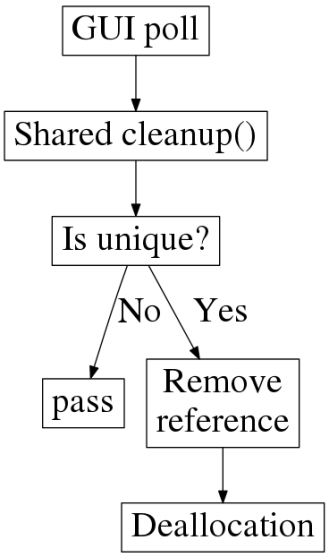


Figure 6.8: Removing an AudioBuffer from Shared

6.3 Data collection

Various different tools were used in the data collection process. First the "ps" linux command will be explained, followed by `massif`, a tool from the valgrind profiling suite.

6.3.1 Process Status

The data is collected by using the Linux tool `ps`, which is the shorthand name for "process status". There are a variety of options which can be passed to `ps` in order to change its behaviour. For this research the memory footprint of the showcase programs was collected using the following command:

```
ps -C showcase1 -o pid=,rss=,vsz=
```

The code snippet will print three values: the process ID of the program, the resident set size (RSS) and the virtual memory size (VSZ). It is difficult to obtain an accurate measurement of memory usage on Linux due to there being many factors of a process' memory consumption. The `ps` tool reports RSS size, which counts the shared memory pages as well as the programs own memory pages (Eqware.net 2009). Since the shared memory pages are shared, it is not really fair to "blame" them all on the analysed process.

The data gathered should not be interpreted to represent the programs physical memory usage. The RSS values presented by "ps" can be compared between the two implementations, providing insight into which uses more memory. Since the shared components of each implementation are the same size, any differences are caused by the programs own memory usage.

6.3.2 Valgrind Massif

A better tool to analyse the implementations memory usage is `massif`, a tool in the `Valgrind` suite of debugging and profiling tools. `Massif` is a heap profiling tool, taking snapshots of memory usage over time. Unfortunatly it runs programs about 20 times slower than normal (Valgrind.org 2012). A very nice feature of `massif` is that as its unit of time between snapshots it uses CPU instructions. It follows that the slowdown in execution speed does not affect the resulting graph, as it is not based on time but CPU instruction count.

Note that the memory usage that `massif` shows is only the heap usage, and not all the memory of the process. This value is of high importance to this research, as all samples loaded from disk interact with the heap using `new` and

`delete`. It follows that by profiling heap usage, a detailed insight of the programs allocation and de-allocation can be graphed.

The output of `massif` is a text file detailing the process's activities during the profiling run. To graph these profiling data files a scriptis used to reformat the data so that it is easier to plot with GNUplot (Sarine.nl 2012).

6.3.3 Profiling Execution

The `gprof` profiling tool outputs a text file called `gmon.out` after a profiling run. This text file is then interpreted to generate a graph showing the functions which consume the most time, and how much time was spent in each function. The command used to generate the callgraphs such as Fig. 6.15 is as follows:

```
gprof bin/selfProfiling/showcase1 bin/selfProfiling/showcase1_gmon.out \
| python2 ./gprof2dot.py -s | dot -Tsvg -o output/showcase1callgraph.svg
```

6.4 Graphs of Data

This section presents the data collected previously. Graphs of the data are generated using Gnuplot. This section presents the data as plotted, see the discussion section for an interpretation of the graphs.

Firstly four graphs of memory usage are presented. The first two graphs show implementation one and two's RSS and VSZ memory usage.

The next two graphs show heap memory usage over time versus CPU instruction time. Each implementation is graphed seperatly. The two after that compare heap memory usage of the implementations, first using CPU instructions as the time unit, the second using milli-seconds.

The final four graphs show and compare the performance of the implementations, drawing from both `gprof` profiling data and data gathered during CPU stress testing.

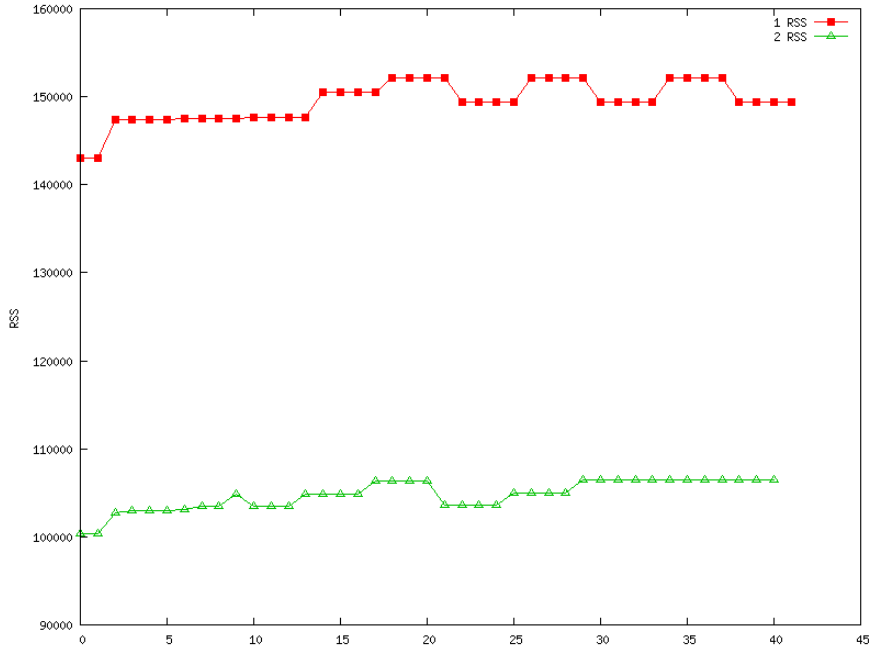


Figure 6.9: RSS Memory Usage Comparison - Compares the RSS memory usage of the two implementations.

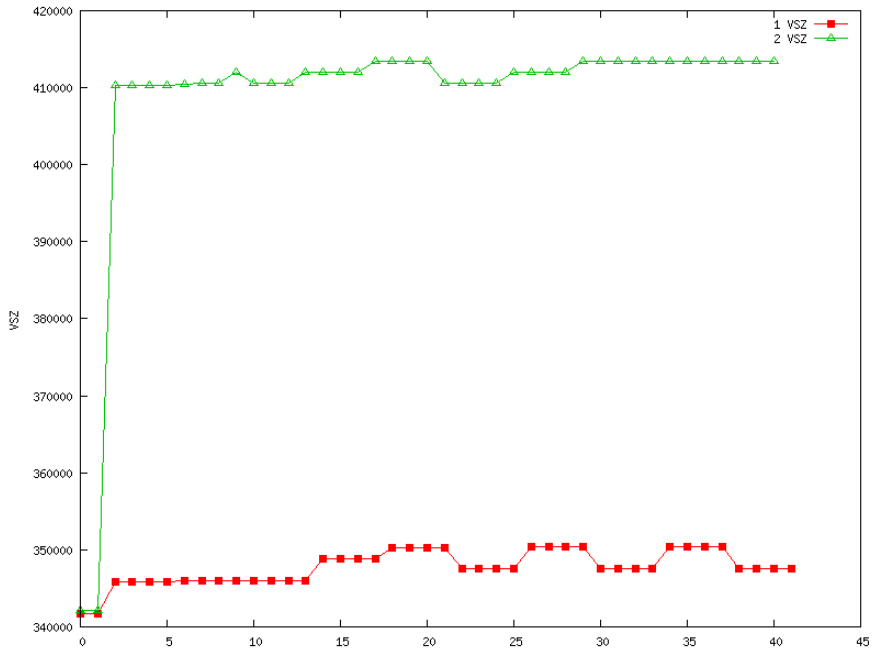


Figure 6.10: VSZ Memory Usage Comparison - Compares the VSZ memory usage of the two implementations.

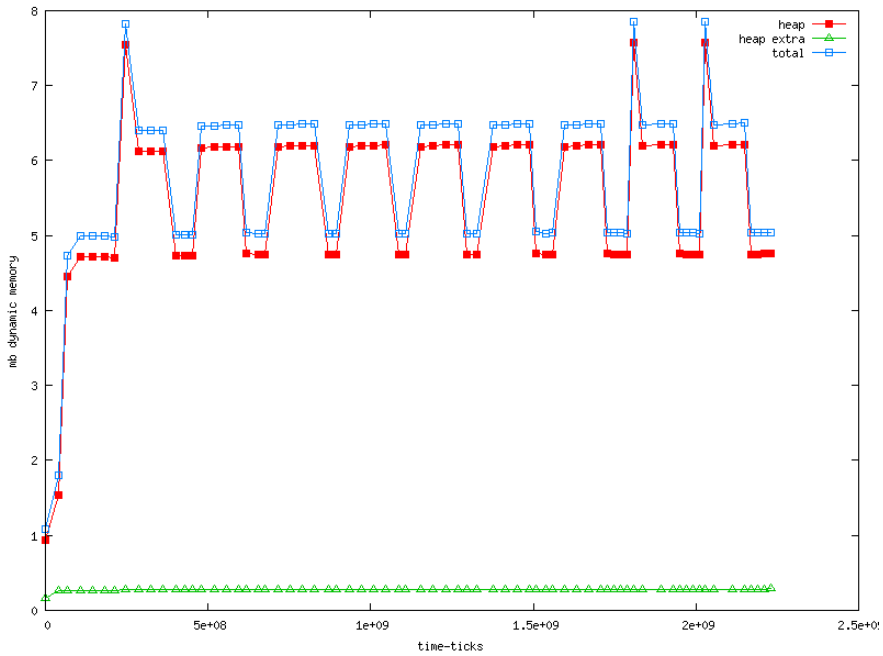


Figure 6.11: Heap Memory Usage of Implementation One - A repeating rise of memory consumption occurs when loading a new sample.

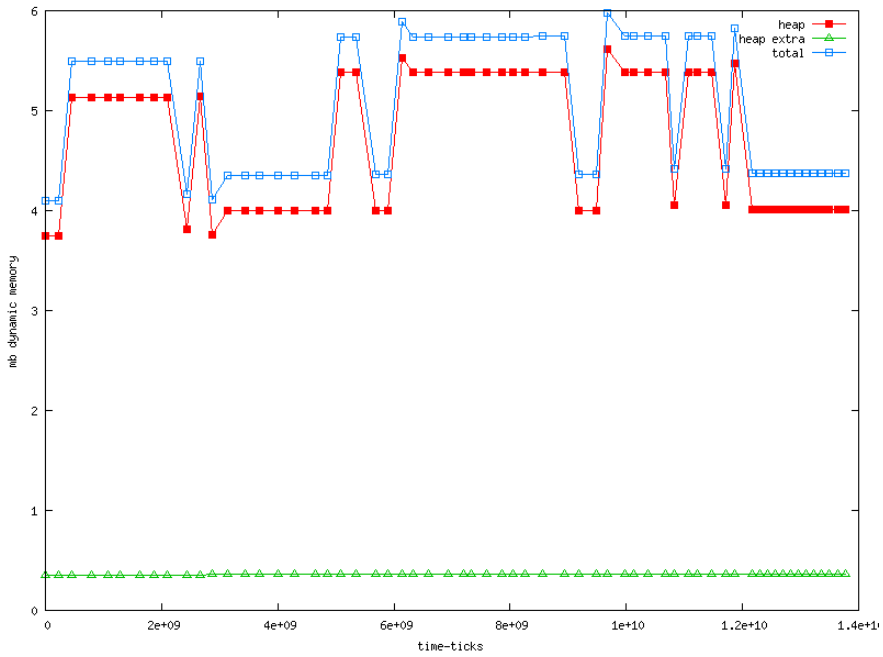


Figure 6.12: Heap Memory Usage of Implementation Two - Memory consumption of implementation two is not regular.

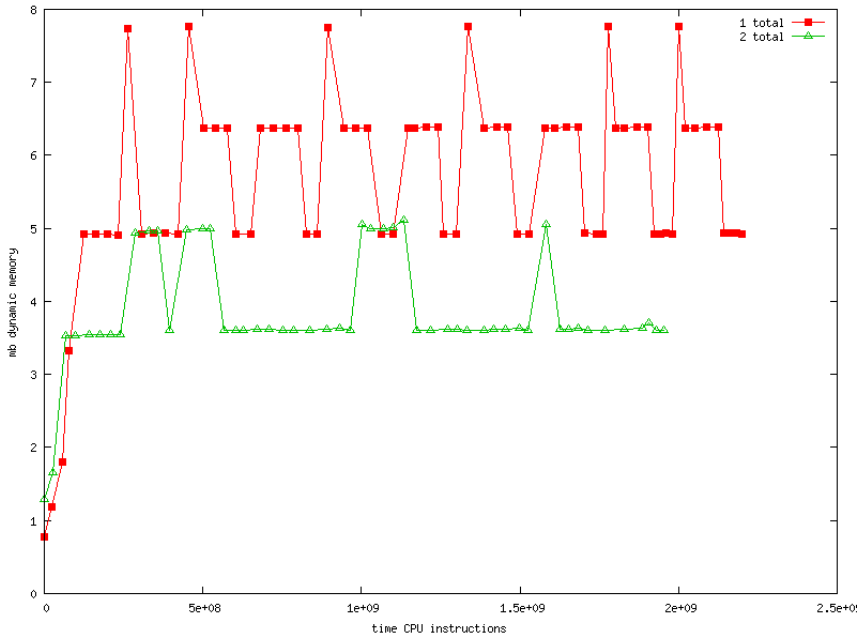


Figure 6.13: CPU Instruction Comparison of Heap Memory Usage - Graph comparing time in CPU instructions and total memory usage.

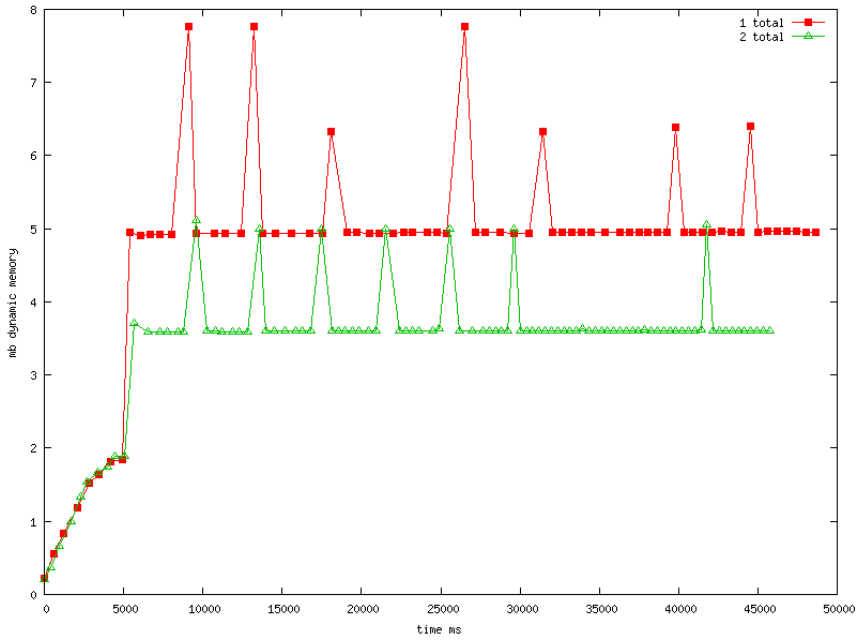


Figure 6.14: Time Comparison of Heap Memory Usage - Graph comparing the implementations' time in milliseconds and total memory usage.

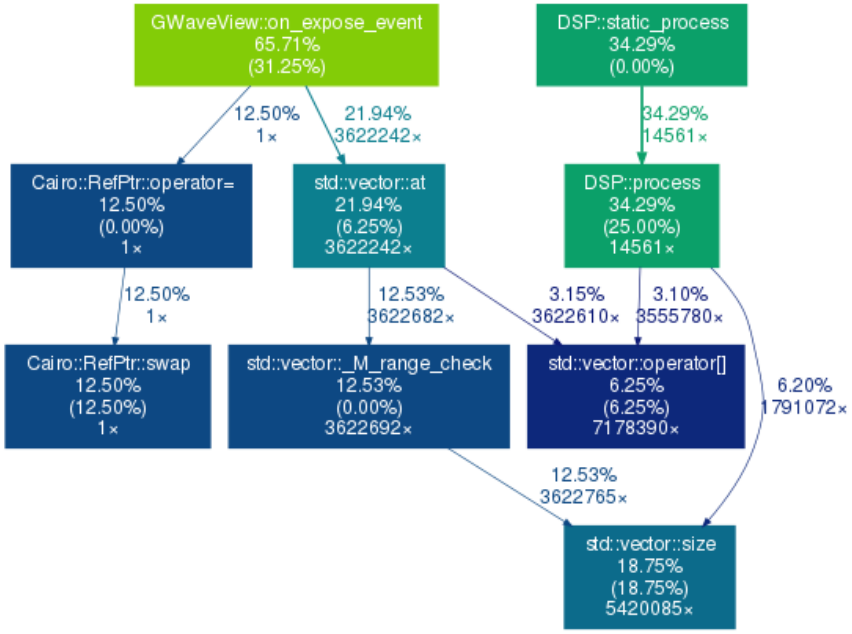


Figure 6.15: Callgraph of Implementation One - Graph showing percentage of time spent per function of implementation one.

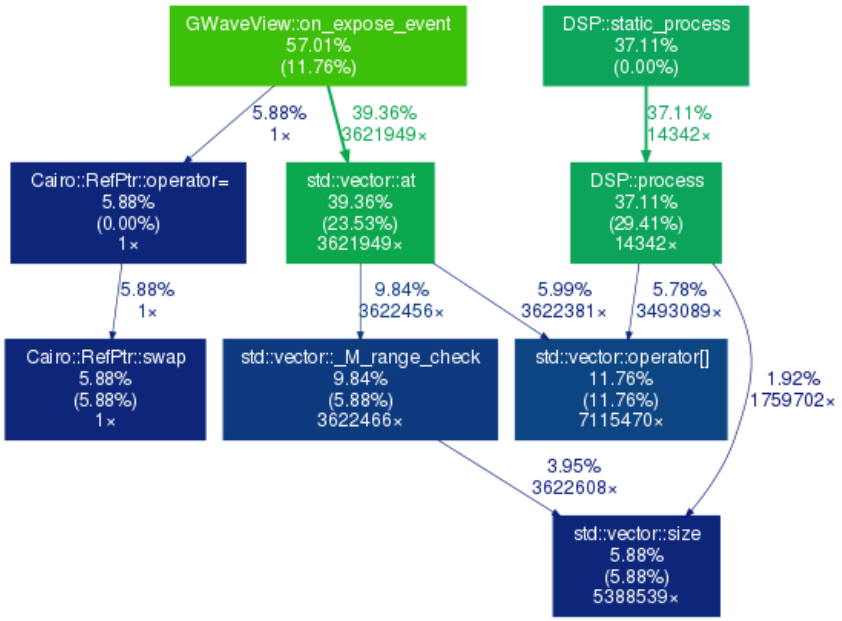


Figure 6.16: Callgraph of Implementation Two - Graph showing percentage of time spent per function of implementation two.

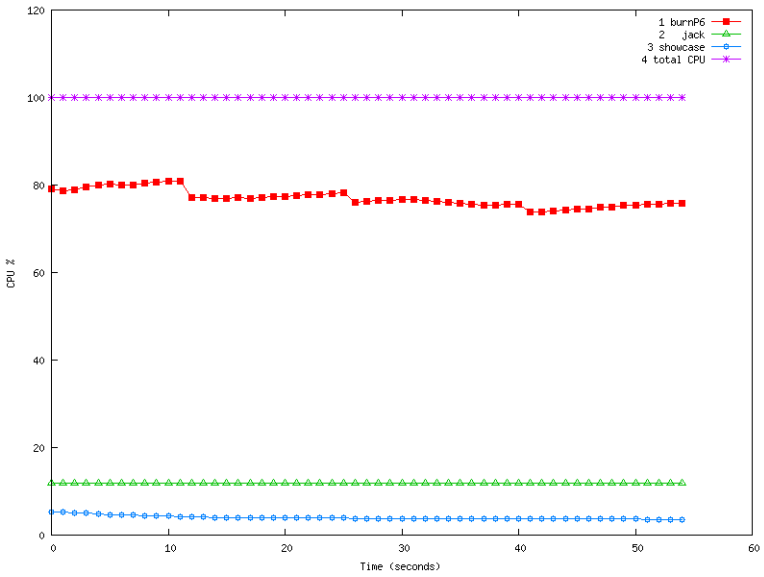


Figure 6.17: Processor Usage During Stress Testing - Graph showing CPU usage over time. No glitching occured during this test.

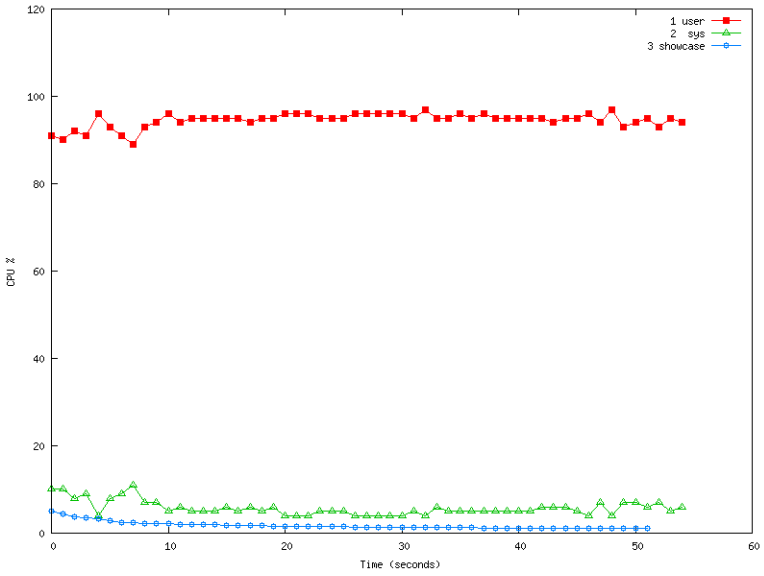


Figure 6.18: Comparison of Processor Usage by System and User - Graph comparing system and user usage. The user processor usage includes `burnP6` processes, hence it takes all available CPU.

7

Discussion

This section discusses the results presented in the previous section. The topics of inter-thread communication and memory managements are dealt with separately.

7.1 Inter-thread Communication

This section discusses the differences between inter-thread communication when using manual `Event` class instances and `Boost::Function`. The details of using each implementation are brought forward.

7.1.1 Event Messaging

The `Event` messaging system proved to be a workable solution. The events themselves are declared in a single file, providing easy reference for the programmer. Creating the events on the stack is certain to be real-time safe, and using `memcpy` to move them into the ringbuffer is also real-time safe. This method of communication uses fundamental C++ language features to achieve its goal, making it easy to check for real-time safety.

Reliably reading `Event` objects from the ringbuffer proved difficult due to their different sizes and the concurrent nature of real-time programming. The issue was solved by checking the size of the available data in the ringbuffer twice.

7.1.2 Boost::Function

The inter-thread communication method using Boost::Function works when binding the function in a non-real-time context, and executing it in a real-time context. The opposite is not real-time safe, and hence this method of communciation is not advised as the underlying communication method for real-time audio programs.

The author has investigated using custom allocators in order to circumvent Boost::Function calling `malloc`, and it was confirmed that Boost::Function accepts custom allocators (Semashev 2013).

The fact that custom allocates can be used with Boost function objects means it possible to use a pool of manually allocated memory to write the Boost::Function data. This would allow pre-allocation of the memory pool, and a real-time safe custom allocator would then be used in order to give Boost::Function the needed memory. This solution would prove to be quite complex, however it is expected to be a workable solution.

7.1.3 Summary

To finish the discussion on inter-thread communication it is important to note that any solution should scale to passing messages in both directions while not violating the real-time constraints. Since the Boost::Function method in its current state does not scale to sending messages from the real-time thread to another thread, the author does not concider it a workable solution. It follows that using the `Event` class method to pass data between the threads is advised by the author.

7.2 Memory management

This section discusses the different implementations of memory management. The discussion centers around the graphs of memory usage presented in the previous section. Firstly the manual memory management method is discussed, then the `Shared` class method is detailed.

7.2.1 Manual

Manual memory management has the advantage of each memory related event being controllable. This would allow for optimizations such as freeing `Audiobuffer` instances when CPU usage is dropping, or keeping caches of instances for usage later in the program's execution.

The downside to manual management is that if there is a bug somewhere in the code, memory may not always be freed. Leaked memory will over time burden the system, and ultimatly end in the program being terminated due to starving the system of resources.

Assuming a bug-free implementation, the author feels this is a workable solution. Extensive testing is advised to enusre there are no memory leaks. The Valgrind tool `massif` was extensivly used for ensuring there are no memory leaks in the test implementations that are part of this research.

7.2.2 Shared

The memory management scheme based on `Boost::shared_ptr` proved a workable solution. The `Shared` class ensures that even if a pointer goes out of scope everywhere else, it is de-allocated properly in a non-real-time thread.

Ensuring that `AudioBuffer` instances are added to the `Shared` instance is of vital importance in this implementation. If this is not achieved, the `AudioBuffer` instance could be de-allocated in the real-time thread, causing much a worse problem than a memory leak: the possiblity of degrading the quality of the real-time system.

In order to ensure the adding of the `AudioBuffer` instances to `Shared`, the common C++ pattern of using a factory applies. The factory class would automatically add the `AudioBuffer` instance to `Shared`, solving the runaway `shared_ptr` problem.

7.2.3 Summary

The graph shown in Fig. 6.9 compares the RSS memory usage of the implementations. Implementation one is the line labeled "1" in the ledgend. Fig. 6.10

compares the VSZ memory usage of the implementations. Again, implementation one is the line labeled "1" in the ledgend.

The graph in Fig. 6.11 shows implementation one's heap usage. The three lines show heap, heap extra and total usage. Fig. 6.12 shows implementation two's heap usage. Again the three lines show heap, heap extra and total usage.

The graph Fig. 6.13 compares the memory consumption of both implementations, showing that implementation two uses much less memory than implementation one.

7.3 Performance

This section analyses the performance difference between the implementations. The actual speed difference between the implementations is negligable. Data to support this hypothesis has been gathered from `gprof`.

Fig. 6.15 and Fig. 6.16 show the results obtained. The `on_expose_event()` function of the `GWaveView` class is almost identical in both implementations. It follows that the `DSP::process()` functions weight the graph, and in doing so influence the weight of `GWaveView::on_expose_event()`.

Since implementation one has a `DSP::process()` weight of 34%, while implementation two's `DSP::process()` has a weight of 37%, it can be concluded that implementation one is slightly more efficient.

Although these statistics are accurate, it is difficult to place the difference in speed on one aspect of the program. It could be the inter-thread communication system slowing down the program, or it could be the memory management. More profiling would allow a conclusion to be drawn.

The graph in Fig. 6.17 shows CPU usage over time with two instances of `burnP6` running, maxing out CPU usage. It is shown that the CPU usage of the showcase application is quite low: averaging on around 5% of total load, while JACK's CPU usage is approx. 11% troughout the profiling run.

Fig. 6.18 shows the CPU usage between the operating system, and the user tasks. We see that the line marked `sys` is in flux, and the `user` line is the opposite. This is caused by the system scheduling itself first, and the instances of `burnP6` consuming all remaining CPU time.

7.4 Possible Causes of Error

This section details possible causes of error. Memory usage and CPU usage are dependant factors of overall system load. Efforts were made to ensure that the system load was consistent while doing the profiling runs in order to keep statistics as accurate as possible.

7.4.1 Measuring Memory Usage

Measuring RSS and VSZ memory usage are not ideal solutions, as they include shared objects that may be needed for other programs also. It follows that the results should not be interpreted directly. All measurements were taken on one computer, running the same kernel version troughout the tests, and with the same amount of system memory.

When profiling the memory usage using `massif` comparison chart, it was necessary to use time in milliseconds as the X axis unit. This is due to the different implementations having different amounts of CPU instructions, and hence a different range on the X axis. As time is the dominating factor for reference, the background tasks of the system will influence the time taken to run the program. An attempt was made to ensure equal load was on the system, however some error may have crept into the results at this stage.

7.4.2 Profiling in Real-time

During a profiling run, JACK informs us that system calls interrupt processing by printing out errors similar to the following:

```
JackPosixSemaphore::TimedWait err = Interrupted system call
```

Although the collected statistics are not of the program running in real-time due the the profiler, the generated statistics of one implementation can be compared to statistics generated of the other implementation. Since the execution speed slowdown is the same on each profiling run, conclusions may still be drawn from the collected results.

7.4.3 Confounding Factors

The fact that the author has coded both of the implementations implies that there is no blind testing. As familiarity with the problem changes during implementation, the code written later in the project may be of higher quality. Also the author may favour certain designs and implementations due to personal preference. Unfortunately there is nothing the author can do about this, apart from note these confounding factors.

7.5 Summary

The two different methods of inter-thread communication are discussed in light of the results presented in the previous section. Both implementations' strong and weak points are brought forward, and it is decided that the method using the `Event` class is the better solution, as it is always real-time safe.

The two memory management methods are then discussed, drawing from the results section. Both solutions are workable, however the implementations using the `Shared` class is a cleaner and much less prone to memory leaks. For this reason the author advises its use over a system which relies on raw pointers.

8

Conclusion

This section first concludes the research question, which implementation of inter-thread communication is more suited to audio programs. It continues to conclude which memory managements system is superior. Finally a section is dedicated to concluding the authors experience while doing this research.

Inter-thread Communication

The inter-thread communication system based on the `Event` class is far superior to the `Boost::Function` based system, as the `Event` class is entirely real-time safe while the system based on `Boost::Function` is not. The solution using `Boost::Function` could be adapted to use a custom allocator allowing its use in a real-time thread, however the author feels that this is a false gain as the solution becomes very complex.

Memory Management

Both of the memory management solutions are workable solutions, and hence it is situation dependant which solution to choose. If working with immutable large objects, the `Shared` class solution is more appropriate as it requires less memory, and less chance of memory leaks. In a case where the data must be mutable a compromise between the `Shared` class and manual memory management may be used, providing additional flexibility which the `Shared` class alone doesn't afford.

A implementation of the showcase program using the `Event` class for messaging and the `Shared` class for memory management would seem to be the best solution out of those analysed in this research.

Experience During Research

The experience of developing a low-latency audio application as part of this research as convinced the author that the linux platform is very capable in the field of executing applications with real-time requirements. There are many different factors to achieve reliable real-time operation, ranging from hardware purchased to which kernel patches have been applied.

The linux environment is particularly suited to run real-time programs due to is configurable nature. Advanced features such as real-time priority scheduling and setting hardware interrupts allows the use to tune the system to its optimal performance. Latencies of less that one milli-second can be achieved while still operating reliably without glitches.

Final thoughts

The output of the research has informed the author on making implementation choices for real-time programs. The experience gained by doing this research will enable me to write real-time safe software in future: I hope it helps others in a similar way.

Appendix A

System Statistics

The following reports show the current state of the operating system with regards to real-time performance. Each section states the command used from a terminal to gather the displayed output.

Interrupt Requests

```
#:  cat /proc/interrupts
           CPU0       CPU1
  0:      83019      82078   IO-APIC-edge      timer
  1:          5          4   IO-APIC-edge      i8042
  8:          1          0   IO-APIC-edge      rtc0
  9:       1511       1527   IO-APIC-fasteoi   acpi
 12:         80         94   IO-APIC-edge      i8042
 14:        246        239   IO-APIC-edge      ata_piix
 15:          0          0   IO-APIC-edge      ata_piix
 17:      42167      41493   IO-APIC-fasteoi   uhci_hcd:usb6
 18:          0          0   IO-APIC-fasteoi   uhci_hcd:usb7
 20:         13          8   IO-APIC-fasteoi   ehci_hcd:usb1,
    uhci_hcd:usb3
 21:          0          0   IO-APIC-fasteoi   uhci_hcd:usb4
 22:     152704     154259   IO-APIC-fasteoi   mmc0, tifm_7xx1,
    yenta, snd_indigodj
 23:          1          1   IO-APIC-fasteoi   ehci_hcd:usb2,
    uhci_hcd:usb5
 44:      11059      11147   PCI-MSI-edge      ahci
 45:        104        105   PCI-MSI-edge      snd_hda_intel
 46:       2498       2457   PCI-MSI-edge      radeon
```

```
 47:       3802       3832   PCI-MSI-edge      iwl3945
 48:          2          0   PCI-MSI-edge      eth0
NMI:          0          0   Non-maskable interrupts
LOC:     198640     122405   Local timer interrupts
SPU:          0          0   Spurious interrupts
PMI:          0          0   Performance monitoring interrupts
IWI:          0          0   IRQ work interrupts
RTR:          0          0   APIC ICR read retries
RES:     359093     396627   Rescheduling interrupts
CAL:      10622       4642   Function call interrupts
TLB:          0          0   TLB shootdowns
TRM:          0          0   Thermal event interrupts
THR:          0          0   Threshold APIC interrupts
MCE:          0          0   Machine check exceptions
MCP:       1399       1399   Machine check polls
ERR:          0
MIS:          0
```

Real Time Tasks

```
#:  ps -eLo rtprio,cls,cmd | grep "FF" | sort -r
99  FF [posixcputmr/1]
99  FF [posixcputmr/0]
99  FF [migration/1]
99  FF [migration/0]
90  FF [irq/8-rtc0]
87  FF [irq/22-yenta]
84  FF /usr/bin/jackd -P84 -t2000 -dalsa -dhw:1 -r96000 -p32 -n2
   -Xseq
81  FF [irq/1-i8042]
80  FF [irq/12-i8042]
50  FF [irq/9-acpi]
50  FF [irq/48-eth0]
50  FF [irq/47-iwl3945]
50  FF [irq/46-radeon]
50  FF [irq/45-snd_hda_]
50  FF [irq/44-ahci]
50  FF [irq/23-uhci_hcd]
50  FF [irq/23-ehci_hcd]
50  FF [irq/22-tifm_7xx]
```

```
50  FF [irq/22-snd_indi]
50  FF [irq/22-mmc0]
50  FF [irq/21-uhci_hcd]
50  FF [irq/20-uhci_hcd]
50  FF [irq/20-ehci_hcd]
50  FF [irq/18-uhci_hcd]
50  FF [irq/17-uhci_hcd]
50  FF [irq/15-ata_piix]
50  FF [irq/14-ata_piix]
 1  FF [ksoftirqd/1]
 1  FF [ksoftirqd/0]
 -  TS grep FF
```

RTIRQ Status

```
#:  /etc/rc.d/rtirq status
```

```
  PID CLS RTPRIO  NI PRI %CPU STAT COMMAND
   34 FF      90   - 130  0.0 S    irq/8-rtc0
  260 FF      87   - 127  1.1 S    irq/22-yenta
   32 FF      81   - 121  0.0 S    irq/1-i8042
   31 FF      80   - 120  0.0 S    irq/12-i8042
   21 FF      50   -  90  0.0 S    irq/9-acpi
   52 FF      50   -  90  0.1 S    irq/44-ahci
   80 FF      50   -  90  0.0 S    irq/20-ehci_hcd
   96 FF      50   -  90  0.0 S    irq/14-ata_piix
   98 FF      50   -  90  0.0 S    irq/15-ata_piix
  101 FF      50   -  90  0.0 S    irq/23-ehci_hcd
  103 FF      50   -  90  0.4 S    irq/22-mmc0
  105 FF      50   -  90  0.3 S    irq/22-tifm_7xx
  107 FF      50   -  90  0.0 S    irq/20-uhci_hcd
  108 FF      50   -  90  0.0 S    irq/21-uhci_hcd
  109 FF      50   -  90  0.0 S    irq/23-uhci_hcd
  110 FF      50   -  90  0.9 S    irq/17-uhci_hcd
  111 FF      50   -  90  0.0 S    irq/18-uhci_hcd
  255 FF      50   -  90  0.0 S    irq/45-snd_hda_
  267 FF      50   -  90  0.0 S    irq/46-radeon
  268 FF      50   -  90  0.1 S    irq/47-iwl3945
  431 FF      50   -  90  0.0 S    irq/48-eth0
  700 FF      50   -  90  1.0 S    irq/22-snd_indi
```

```
    3 FF       1   -  41  0.5 S    ksoftirqd/0
   14 FF       1   -  41  0.2 S    ksoftirqd/1
```

Real Time Quick Config Scan

```
#:  realTimeConfigQuickScan
== GUI-enabled checks ==
Checking if you are root... yes - not good
You are running this script as root. Please run it as a regular
   user for the most reliable results.
Checking filesystem 'noatime' parameter...  - good

Checking CPU Governors... CPU 0: 'performance' CPU 1: '
   performance'  - good
Checking swappiness... 10 - good
Checking for resource-intensive background processes... none
   found - good
Checking checking sysctl inotify max_user_watches... >= 524288 -
   good
Checking access to the high precision event timer... readable -
   good
Checking access to the real-time clock... readable - good
Checking whether you're in the 'audio' group... no - not good
add yourself to the audio group with 'adduser $USER audio'

Checking for multiple 'audio' groups... no - good
Checking the ability to prioritize processes with chrt... yes -
   good
Checking kernel support for high resolution timers... found -
   good
Kernel with Real-Time Preemption... not found - not good
Kernel without real-time capabilities found
For more information, see http://wiki.linuxmusicians.com/doku.php
   ?id=system_configuration#installing_a_real-time_kernel
Checking if kernel system timer is set to 1000 hz... found - good
Checking kernel support for tickless timer... found - good
== Other checks ==
Checking filesystem types... ok.
ok.
** multiple devices found at the sound cards' IRQ
```

Appendix B

C++ code

src/imp1/event.hpp

```
31 // Transfers an AudioBuffer*
32 class EventBufferTransfer : public EventBase
33 {
34   public:
35     int type() { return int(EVENT_BUFFER_TRANSFER); }
36     uint32_t size() { return sizeof(EventBufferTransfer); }
37
38     AudioBuffer* audioBufferPtr;
39
40     EventBufferTransfer(){};
41     EventBufferTransfer(AudioBuffer* buf)
42     {
43       audioBufferPtr = buf;
44     }
45 };
```

src/imp1/dsp.cpp

```
70 int DSP::process(jack_nframes_t nframes)
71 {
72   // process events from ringbuffer
73   top->processDsp();
74
75
76   if ( buffer == 0 )
77   {
78     // no buffer loaded yet
79     return 0;
80   }
81
82
83   float* outLBuffer = (float*) jack_port_get_buffer( out_left, nframes );
84   float* outRBuffer = (float*) jack_port_get_buffer( out_right,nframes );
85
86   // playback the currently loaded sample
87   const std::vector<float>& buf = buffer->get();
88   size_t size = buf.size();
89
90   for ( int i = 0; i < nframes; i++)
91   {
92     if ( playbackIndex < size - 1 )
93     {
94       *outLBuffer++ = buf[playbackIndex];
95       *outRBuffer++ = buf[playbackIndex];
```

```
96        playbackIndex++;
97      }
98      else
99      {
100       playbackIndex = 0;
101     }
102   }
103
104   // update GUI of current playback position
105   float position = float(playbackIndex) / buf.size();
106   EventPlaybackPosition e(position);
107   top->dspToGuiWrite( &e );
108
109   return 0;
110 }
```

src/imp2/dsp.cpp

```
56  int DSP::process(jack_nframes_t nframes)
57  {
58    // process events from ringbuffer
59    while ( top.getGuiToDsp()->can_read() )
60    {
61      top.getGuiToDsp()->read() ();
62    }
63
64    if ( buffer == 0 )
65    {
66      // no buffer loaded yet
67      return 0;
68    }
69
70    float* outLBuffer = (float*) jack_port_get_buffer( out_left, nframes );
71    float* outRBuffer = (float*) jack_port_get_buffer( out_right,nframes );
72
73    // playback the currently loaded sample
74    const std::vector<float>& buf = buffer->get();
75    size_t size = buf.size();
76
77    for ( int i = 0; i < nframes; i++)
78    {
79      if ( playbackIndex < size - 1 )
80      {
81        *outLBuffer++ = buf[playbackIndex];
82        *outRBuffer++ = buf[playbackIndex];
83        playbackIndex++;
84      }
85      else
86      {
87        playbackIndex = 0;
88      }
89    }
90
91    // update GUI of current playback position
92    float position = float(playbackIndex) / buf.size();
93
94    if( top.getDspToGui()->can_write() )
95    {
96      // this is NON-RT! boost:function allocates memory!
97      top.getDspToGui()->write( boost::bind( &GUI::playbackPosition, &top.getGUI(), position ) );
98    }
99
100   return 0;
101 }
```

src/imp1/top.cpp

```
103 // GUI thread calls this function periodically to update itself
104 void Top::processGui()
105 {
106   // check if there's anything to read
107   int availableRead = jack_ringbuffer_read_space(dspToGui);
108
109   while ( availableRead >= sizeof(EventBase) )
110   {
111     jack_ringbuffer_peek( dspToGui, (char*)dspToGuiMem, sizeof(EventBase) );
112
113     EventBase* e = static_cast<EventBase*>( dspToGuiMem );
114
115     // again, have to check in case buffer is wrapping, and thread gets paused
116     // inbetween the two memcpy calls, leaving a possible non-full event
117     if ( availableRead >= e->size() )
118     {
119       switch ( e->type() )
120       {
121         case EventBase::EVENT_BUFFER_TRANSFER:
122         {
123           // create an empty of this event type
124           EventBufferTransfer ev( (AudioBuffer*)0 );
125
126           // read the event from the buffer
127           jack_ringbuffer_read( dspToGui, (char*)&ev, sizeof(ev) );
128
129           stringstream s;
130           s << "GUI: Deleting AudioBuffer instance " << ev.audioBufferPtr->getID() << "  Addr: "
131             << ev.audioBufferPtr;
132           gui->printTextView( true, s.str() );
133
134           AudioBuffer* tmp = ev.audioBufferPtr;
135           cout << "Top deleting DSP audioBuffer  Addr: " << tmp << endl;
136
137
138           // do the action: de-allocate buffer
139           delete tmp;
140
141           break;
142         }
143         case EventBase::EVENT_GUI_PRINT:
144         {
145           // create an empty of this event type
146           EventGuiPrint ev;
147
148           // read the event from the buffer
149           jack_ringbuffer_read( dspToGui, (char*)&ev, ev.size() );
150
151           // do the action: print it
152           gui->printTextView( false, ev.getMessage() );
153
154           break;
155         }
156         case EventBase::EVENT_PLAYBACK_POSITION:
157         {
158           // create an empty of this event type
159           EventPlaybackPosition ev;
160
161           // read the event from the buffer
162           jack_ringbuffer_read( dspToGui, (char*)&ev, sizeof(ev) );
163
164           // do the action: set the playback position
165           gui->getWaveview()->setPlaybackPosition( ev.position );
166
167           break;
168         }
169         default:
170         {
```

```
171           cout << "Top::processGUI() Recieved unknown event type " << e->type() << endl;
172         }
173       } // switch
174     } // if ( available >= e->size() )
175     else
176     {
177       // return if the EventBase was written but the whole event was not. This
178       // breaks out of the while loop, and the EventBase which is still in the
179       // ringbuffer will be read on the next call of this function
180       return;
181     }
182     // update available read, and see if there's more events
```

src/imp2/dsp.cpp

```
132 bool GUI::readRingbuffer()
133 {
134   // process events from ringbuffer
135   while ( top->getDspToGui()->can_read() )
136   {
137     // read event from ringbuffer, then call () on the functor object
138     top->getDspToGui()->read() ();
139   }
140
141   return true;
142 }
```

src/imp1/dsp.cpp

```
46  void DSP::setBuffer(AudioBuffer* newBuf)
47  {
48    if ( buffer )
49    {
50      // send old buffer to be de-allocated
51      { EventBufferTransfer e(buffer);
52        top->dspToGuiWrite( &e ); }
53
54      { EventGuiPrint e("DSP: Writing de-allocate event to GUI");
55        top->dspToGuiWrite( &e ); }
56    }
57
58    EventGuiPrint e("DSP: Recieved new AudioBuffer, playing...");
59    top->dspToGuiWrite( &e );
60
61    buffer = newBuf;
62  }
```

Appendix C

Jack Ringbuffer

ringbuffer.cpp (Davis and Drape 2000)

```
264 /* The copying data writer.  Copy at most 'cnt' bytes to 'rb' from
265    'src'.  Returns the actual number of bytes copied. */
266
267 LIB_EXPORT size_t
268 jack_ringbuffer_write (jack_ringbuffer_t * rb, const char *src, size_t cnt)
269 {
270         size_t free_cnt;
271         size_t cnt2;
272         size_t to_write;
273         size_t n1, n2;
274
275         if ((free_cnt = jack_ringbuffer_write_space (rb)) == 0) {
276                 return 0;
277         }
278
279         to_write = cnt > free_cnt ? free_cnt : cnt;
280
281         cnt2 = rb->write_ptr + to_write;
282
283         if (cnt2 > rb->size) {
284                 n1 = rb->size - rb->write_ptr;
285                 n2 = cnt2 & rb->size_mask;
286         } else {
287                 n1 = to_write;
288                 n2 = 0;
289         }
290
291         memcpy (&(rb->buf[rb->write_ptr]), src, n1);
292         rb->write_ptr = (rb->write_ptr + n1) & rb->size_mask;
293
294         if (n2) {
295                 memcpy (&(rb->buf[rb->write_ptr]), src + n1, n2);
296                 rb->write_ptr = (rb->write_ptr + n2) & rb->size_mask;
297         }
298
299         return to_write;
300 }
```

References

Alexandrescu, A. (2001), *Modern C++ Design*, Addison-Wesley.

Amatriain, X. (2004), *CLAM as an Audio Processing Framework* [online], available: http://xavier.amatriain.net/Thesis/html/node133.html [accessed: 10 April 2013].

Anderson, M. (2009), *What are interrupt threads and how do they work?* [online], available: http://elinux.org/images/e/ef/InterruptThreads-Slides_Anderson.pdf [accessed: 10 April 2013] .

Axford, T. (1989), *Concurrent Programming*, Wiley.

Beddingfield, G. (2011), *[LAD] Inter thread Communication: Design Approach* [online], 20 August, available: http://linuxaudio.org/mailarchive/lad/2011/8/20/184255 [accessed: 10 April 2013].

Bencina, R. (2011), *Real-time audio programming 101: time waits for nothing* [online], available: http://www.rossbencina.com/code/real-time-audio-programming-101-time-waits-for-nothing [accessed: 10 April 2013].

Benton, W. (2013), *Some experiences developing an Audio Unit effect* [online], available: http://willbenton.com/writings/audio-unit-effects.html [accessed: 10 April 2013].

Capela, R. N. (2012), *rtirq update - a 2012 edition* [online], available: http://www.rncbc.org/drupal/node/465 [accessed: 10 April 2013].

Codeguru.com (2005), *Callback Functions Tutorial* [online], available: http://www.codeguru.com/cpp/cpp/cpp_mfc/callbacks/article.php/c10557/Callback-Functions-Tutorial.htm [accessed: 10 April 2013].

Coulter, D. (2000), *Digial Audio Processing*, R&D Books.

Dannenberg, R. and Mazzoni, D. (2013), *Audacity: Get Involved* [online], available: http://audacity.sourceforge.net/community/ [accessed: 10 April 2013].

Davis, P. (2003), *The JACK Audio Connection Kit* [online], available: http://lac.linuxaudio.org/2003/zkm/slides/paul_davis-jack/audio.html [accessed: 10 April 2013].

Davis, P. (2004), *Ardour Development* [online], available: http://ardour.org/development.html [accessed: 10 April 2013].

Davis, P. and Drape, R. (2000), *JACK: ringbuffer.h File Reference* [online], available: http://jackaudio.org/files/docs/html/ringbuffer_8h.html [accessed: 10 April 2013].

Donalies, M. (2013), *[LAD] making sense of Jack MIDI; or, is this an appropriate use for Jack?* [online], 20 August, available: http://linuxaudio.org/mailarchive/lad/2013/2/15/197454 [accessed: 10 April 2013].

Engelen, A. (2007), *Jack Interposer* [online], available: https://github.com/raboof/jack_interposer [accessed: 10 April 2013].

Eqware.net (2009), *Capturing Process Memory Usage Under Linux* [online], available: http://www.eqware.net/Articles/CapturingProcessMemoryUsageUnderLinux/index.html [accessed: 10 April 2013].

FAROS Consortium (2007), *FAROS automation - Open-source real-time automation framework* [online], available: http://www.faros-automation.org [accessed: 10 April 2013].

FFADO.org (2010), *IRQ Priorities* [online], available: http://subversion.ffado.org/wiki/IrqPriorities?version=35 [accessed: 10 April 2013].

Gregor, D. (2001), *Boost.Function* [online], available: www.boost.org/libs/function [accessed: 10 April 2013].

Hamano, J. and Torvalds, L. (2013), *Git –fast-version-control* [online], available: http://git-scm.com/ [accessed: 10 April 2013].

Henke, R. (2004), *Ableton Forum : Ableton 6* [online], available: https://forum.ableton.com/viewtopic.php?p=280887 [accessed: 10 April 2013].

Kalev, D. (2010), *C++ Reference Guide More on Real-Time Programming InformIT* [online], available: http://www.informit.com/guides/content.aspx?g=cplusplus [accessed: 10 April 2013].

Kuo, S. M., Lee, B. H. and Tian, W. (2006), *Real-time digital signal processing: implementations and applications*, Wiley.

Kysela, J. (2007), *ALSA Project* [online], available: http://www.alsa-project.org/main/index.php/Main_Page [accessed: 10 April 2013].

Laplante, P. A. (1993), *Real-time systems design and analysis: an engineer's handbook*, Institute of Electrical and Electronics Engineers.

Linux Journal (1998), *Gprof, bprof and Time Profilers* [online], available: http://www.linuxjournal.com/article/2622 [accessed: 10 April 2013].

Manpages (2013), *Manpage for CPUburn* [online], available: http://man.cx/cpuburn [accessed: 10 April 2013].

Nokia (2012), *Dempsy* [online], available: http://dempsy.github.com/Dempsy [accessed: 10 April 2013].

Palmers, P. (2007), 'FireWire (Pro-)Audio for Linux', *In Proceedings of the Linux Audio Conference*, Berlin, March 22-25, 2007, Berlin, Germany, 113-120.

Robillard, D. (2008), *Raul* [online], available: http://drobilla.net/software/raul [accessed: 10 April 2013].

Rostedt, S. and Hart, D. (2007*a*), *'2.6.16 sched_latency histogram',* Internals of the RT patch, Linux Symposium, Ontario, Canada, 173, image.

Rostedt, S. and Hart, D. (2007*b*), *'2.6.20-rt8 sched_latency histogram',* Internals of the RT patch, Linux Symposium, Ontario, Canada, 173, image.

Sarine.nl (2012), *Gmpc 0.16.0 memory usage analyse* [online], available: http://blog.sarine.nl/2008/07/29/gmpc-0160-memory-usage-analyse/ [accessed: 10 April 2013].

Satir, G. and Brown, D. (1995), *C++ The Core Language*, O'Reilly.

Semashev, A. (2013), *[Function] : Memory allocation upon bind()* [online], 20 August, available: http://boost.2283326.n4.nabble.com/

Function-Memory-allocation-upon-bind-td4641417.html#none [accessed: 10 April 2013].

Stallman, R. (1997), *Linux and the GNU System* [online], available: http://www.gnu.org/gnu/linux-and-gnu.html [accessed: 10 April 2013].

Tyson, M. (2011), *A simple, fast circular buffer implementation for audio processing* [online], available: http://atastypixel.com/blog/a-simple-fast-circular-buffer-implementation-for-audio-processing/ [accessed: 10 April 2013].

Valgrind.org (2012), *Valgrind's Tool Suite* [online], available: http://valgrind.org/info/tools.html [accessed: 10 April 2013].

Williams, C. (2008), *Realtime Kernel at the Linux Summit 2008* [online], available: http://people.redhat.com/bche/presentations/realtime-linux-summit08.pdf [accessed: 10 April 2013].

Wirth, N. (1977), 'Toward a discipline of real-time programming', *Commun. ACM* 20(8), 577–583.