Juha Erkkilä

Real-Time Audio Servers on BSD Unix Derivatives

Master's Thesis
in Information Technology
June 17, 2005

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Juha Erkkilä
Contact information: erkkila@cc.jyu.fi
Title: Real-Time Audio Servers on BSD Unix Derivatives
Työn nimi: Reaaliaikaiset äänipalvelinsovellukset BSD Unix -johdannaisjärjestelmissä
Project: Master's Thesis in Information Technology
Page count: 146
Abstract: This paper covers real-time and interprocess communication features of 4.4BSD Unix derived operating systems, and especially their applicability for real-time audio servers. The research ground of bringing real-time properties to traditional Unix operating systems (such as 4.4BSD) is covered. Included are some design ideas used in BSD-variants, such as using multithreaded kernels, and schedulers that can provide real-time guarantees to processes. Factors affecting the design of real-time audio servers are considered, especially the suitability of various interprocess communication facilities as mechanisms to pass audio data between applications. To test these mechanisms on a real operating system, an audio server and a client utilizing these techniques is written and tested on an OpenBSD operating system. The performance of the audio server and OpenBSD is analyzed, with attempts to identify some bottlenecks of real-time operation in the OpenBSD system.
Suomenkielinen tiivistelmä: Tämä tutkielma kattaa reaaliaikaisuus- ja prosessien väliset kommunikaatio-ominaisuudet, keskittyen 4.4BSD Unix -johdannaisiin käyttöjärjestelmiin, ja erityisesti siihen kuinka hyvin nämä soveltuvat reaaliaikaisille äänipalvelinsovelluksille. Tutkimusalueeseen sisältyy reaaliaikaisuusominaisuuksien tuominen perinteisiin Unix-käyttöjärjestelmiin (kuten 4.4BSD:hen). Mukana on suunnitteluideoita, joita on käytetty joissakin BSD-varianteissa, kuten säikeistetyt kernelit, ja skedulerit, jotka voivat tarjota reaaliaikaisuustakeita prosesseille. Huomioon otettavina seikkoina ovat myös reaaliaikaisten äänipalvelimien suunnitteluun vaikuttavat tekijät, erityisesti erilaisten prosessien välisten kommunikaatiomekanismien soveltuvuus sovellusten väliseen äänidatan välitykseen. Näiden mekanismien testaamista varten tutkielmaa varten on kirjoitettu yksi palvelinsovellus sekä asiakassovellus, joita testataan OpenBSD-käyttöjärjestelmän päällä. Analyysin kohteena ovat testipalvelinsovelluksen ja OpenBSD:n suorituskyky, tarkoituksena on erityisesti löytää reaaliaikaisen suorituskyvyn pullonkauloja OpenBSD-käyttöjärjestelmästä.
Keywords: real-time distributed BSD unix kernel audio scheduling
Avainsanat: reaaliaikainen hajautettu BSD unix kernel ääni skedulointi

Preface

You might already know this, but in case you're reading some treeware edition, you might want to learn that this work can also be found on a server that speaks that famous "http"-protocol too. At address *http://www.cc.jyu.fi/~erkkila/thesis/* you should be able to find the LaTeX sources, the test software code that has been typeset into the appendices, test scripts, and some other related things. If you ever find them useful, that's great, as that's more than I could ever expect ;-)

As is pretty much obvious, this work is heavily inspired by the work that the whole BSD community is doing. While the main reasons for that are actually quite political, it is also because of the way that the community handles politics: it's not in what they say, but in what they do. Thus, if this work inspires anyone to hack on something, anything, it would be interesting to know about that, as that would be the best reward I can ever get.

Glossary

API Application Programming Interface. Whatever rules a programmer must abide in order to use some other software component from another software component

Asynchronous I/O With asynchronous I/O, I/O operations proceed independently from the execution of the process that initiated them

BSD Unix Berkeley Software Distribution Unix, the branch of Unix that was developed in University of Berkeley, California. OpenBSD is a derivative of BSD Unix, like other systems such as NetBSD, FreeBSD, and Macintosh OS X

Exokernel A kernel of an operating system model, where the kernel is reduced to merely controlling access to hardware resources. All higher level functionality (such as device drivers and the abstractions they offer) are provided by application libraries

Fork The act of a software project splitting off from another project. Also the project that does the forking. The term comes from the Unix *fork()* system call that creates a new process from the current one

GUI Graphical User Interface

IPC Interprocess communication. Means exchange of data between processes. Facilities for IPC in practically all Unix system include signals, (anonymous and named) pipes, files, sockets, and shared memory

IPL Interrupt Priority Level. Hardware and software interrupts are associated with a level, and kernel can delay interrupt handling ("mask" interrupts) until it is ready to deal with interrupts that have happened on some specific level. This way access to critical data in the kernel can be protected

Kernel The core of an operating system. A kernel handles accessing hardware, and provides a uniform interface to hardware for application software

Kqueue A generic kernel event notification mechanism, common in BSD variants. It provides an extensible framework for various kernel events, and a notification interface for applications interested in those events

JACK Jack Audio Connection Kit. A low-latency audio server project for Unix systems

Preemptive scheduling Preemptive scheduling means a kind of multitasking, where scheduler may interrupt the execution of a process in order to switch to another process. Preemptive kernel means that a kernel can allow this to happen also when the process is executing in kernel mode

QNX A fault-tolerant, real-time, distributed operating system that is based on a microkernel architecture

Scheduler The part of the operating system that apportions CPU time to processes

SMP Symmetric Multi-Processing. Means that operating system and its processes run on more than one CPU, where CPUs have symmetrical relation to each other, that is, no CPU is treated special

SMT Simultaneous multithreading. A performance improvement technique in hardware, which means that a CPU can have several threads executing in parallel. Does not usually provide any benefit unless the operating system knows how to utilize this feature. The system could see deal with it as a variation of SMP

SUSv3 Single Unix Specification version 3, the most prevalent standard for the interfaces of Unix systems. Current version combines SUSv2 and the POSIX standard (the older Unix standard published by IEEE), and also in many ways supersedes POSIX

System V The version of Unix created by AT&T, first released in 1983. Historically the two major flavors of Unix were considered to be System V and BSD, from which most of the Unix flavors were derived from

Thread A thread is a kind of process that shares some execution context with other threads. A process can be viewed as a thread that has its execution context totally isolated from other processes. What this means in practice may be partially system-dependent

Unix Originally a system developed at Bell Labs at AT&T. In this paper the term is used to mean a family of operating systems. Unix is a trademarked term, so its use is controlled and for correctness, all occurrences of it in this paper should possibly be replaced with “Unix-like”. This, however, is ridiculous

X Window System The mostly commonly implemented graphics system in most Unix operating systems. It has client-server architecture, where server accesses the underlying graphics hardware and clients send requests to the server to draw things to the screen. Often referred to simply as "X"

Contents

1 Introduction

One of the primary features of Unix has always been that it should be easy to connect applications together, for the purpose of creating larger applications. When applications find unexpected uses as parts of some larger system, that is a sign of synergy at work, and that something has been done right in their design.

In most software engineering, the deconstructionist model of reducing the functionality of software components to their core tasks, and then tying them together to create larger software systems, has been considered a practical virtue. One of the distinguishing characteristics of Unix culture has been (in all honesty, only to some extent and not totally lacking in others), that this principle is not considered valid only within applications, but also between applications. "Design programs to be connected with other programs." [32, Basics of Unix Philosophy] While it is sometimes necessary to create big applications that do not communicate a lot with their environment, it is the deconstructionist approach applied to the whole system that is the main source of flexibility and robustness in Unix systems.

In this thesis, we are concerned about distributed real-time audio systems, where each system is made of independent applications, which can be combined by a user in arbitrary configurations. The applications need to co-exist with other applications, some of the parts of the audio system may be distributed over a network, while the audio system as a whole still has to meet real-time deadlines. And the primary task of the operating system should be to not only schedule the applications correctly, but also to provide the means of communication between independent parts of the system.

However, we are not interested in these systems on the context of embedded real-time operating systems, but on systems that are not designed for any particular purpose. And for this thesis, those systems mean Unix, concentrating on the BSD branch of the Unix system family. Yet in operating system design, some compromises always have to be made, because of conflicting goals. For example, in system design maximizing system throughput is often a contradictory goal with minimizing system latency. In Unix, the former has traditionally been favored at the expense of the latter. But audio systems are such that, while good real-time performance is a very important requirement, the common usage context is such that dedicated real-time systems are not interesting for our purposes.

Real-Time applications have a characteristic that they can not be considered totally separately from the underlying operating system. If an operating system is not real-time, the application can not be, and it is critical for the application to know how to use the operating system properly, so that its real-time requirements can be met. Thus, if we chose to use some dedicated real-time system, our problem would be a lot easier from a purely real-time perspective. But as we are interested in real-time audio systems, we need also a system that has strong interprocess communication and networking capabilities. It would also be very beneficial if the system could be used for a wide range of typical "desktop" activities, such as browsing the web, because audio systems should be especially useful when used with common application software.

The BSD family of Unix operating systems fulfill these requirements, with the exception of real-time performance. Now that is not quite true, as some of the current variants of BSD have fairly good real-time properties, and all of them can be quite adequate for many soft real-time applications. But historically Unix was not designed for real-time in mind, and the design decisions made still affect many current systems. It should also possibly be so, as the effects of the design are still be beneficial in many usage contexts. Traditional Unix design had the objective of achieving good "best effort" performance, which translates to, for example, high throughput for networking and fast execution time of scientific computations. This should happen while also providing good response times for interactive applications. Real-Time audio systems are a demanding application area, because they prefer systems with good networking built-in, are often used with interactive applications, and also require good real-time guarantees.

1.1 Real-Time audio servers

While the phrase "distributed real-time audio system" has already been thrown here to scare some people off, perhaps a more detailed explanation of what is being meant by it is in place. First of all, the word "distributed" will be used quite loosely throughout this thesis, meaning only that audio components execute independently. They *may* be running in geographically separate locations, or may not — for most cases, it is not essential for us to consider if it is so. In addition, because the systems that we consider will all adhere to client-server -model, it is probably a better idea to use the term *real-time audio servers* instead. The term "server", after all, implies a distributed system model, as the purpose of a server is to coordinate other audio applications (clients) and handle communication between them.

But even then, the term "real-time audio servers" possibly deserves some more detailed explanation, so we will start with what is *not* meant by it. We are not considering the kind of streaming audio client-server -systems, where the point is only to keep downloading the audio from a server to a client, while the user is listening to the transmitted audio. These systems typically use audio buffering to achieve an interrupt-free audio stream, and this buffering adds latency. The latency may typically be somewhere around five seconds, meaning that clients plays the audio five seconds after it has been sent by the server. These systems have a rather narrow application range, they are not real-time in the sense we are interested in, and thus we are not interested in them.

What we also do not mean are the so called "teleconferencing" related software packages, such as software using VoIP (Voice over Internet Protocol) to transmit audio data through the network. These do have some similar problems to solve, as they should function over the network while achieving a fairly low latency for transmission. Too long delays in transmitting voice in phone calls, for example, is rather an awkward effect and is disruptive to normal flow of conversations. However, the problem domain for real-time audio servers is more complex than simply "how to pass audio data through the network as fast as possible". In fact the networking perspective is not of particular interest in this thesis. The point of this thesis is not the problem of building some specific type of audio software, but how servers can be used to glue together the audio I/O of a wide range of different applications.

This means that audio servers should be useful even when used only in one computer. To make this more clear, consider some of the problems that are prevalent in contemporary Unix audio programming. First of all, many different Unix variants have different kernel level audio APIs. Open Sound System API might possibly be the most widespread, used by FreeBSD, Linux and others, but systems such as NetBSD, OpenBSD and Solaris have a different API, and even Linux is in transition phase to a new API. To program portably one has to write the audio interface separately for different systems.

Other problem with most kernel level audio APIs is that applications have exclusive access to audio device. A simple task of mixing the audio streams of two or more applications into the audio device, real-time, is not typically possible, unless the system works around this with "virtual audio devices" (as in FreeBSD), or some other way. Neither is it easy to capture audio output of an application, to process it before it goes to the device or a file, or route it as input for another application. In Unix, one can not associate a special file with a device of another computer. Thus, it is not possible to run an audio application on one computer and listen to its output

on another. One other possibly problematic issue is proper synchronization with the primary graphics system on Unix systems, the X Window System.

Audio servers can provide a solution to the above problems. By using a server that is portable across many platforms, one can write applications to use only that one server-provided API, and have it work on all of those platforms. If a server has an exclusive access to an audio device, it can serve as a proxy for client applications, and also enables flexible processing and routing of audio streams. Furthermore, if a client-server system uses sockets or some similar technique to pass audio data, doing audio I/O becomes a network transparent operation for clients, and server and clients can be distributed over a network.

X Window System makes an interesting comparison point, because it solves the same or very similar issues for graphics in Unix systems. It includes an X server, and X libraries that clients utilize to connect to the server. The server has an exclusive access to a video device, that clients can access by communicating with the server using the X protocol, that consists mostly of drawing requests for the server. Programs can be run in different computers than where their actual display is, and X implementations are mostly compatible with each other, with some running on many different computing platforms. Almost all software that have GUIs and are written for Unix systems utilize X as their underlying graphics engine.

X Window System, however, is not concerned about audio. X Consortium had a project called X Audio System[42] in the early nineties to complement it the windowing system, but it was halted in August 1996. Among the current projects, the Media Application Server -project at address *http://www.mediaapplicationserver.net/* might be closest to the idea of the X Audio System, working in conjunction with the windowing system. There are other audio server projects as well, which will be briefly introduced in chapter 7. Their main purpose is to provide a network transparent system for mixing together audio streams.

However, most servers have not primarily been designed for real-time, in the most strict sense of the term. Real-Time audio servers have at least two additional requirements, that can be quite difficult to achieve. The server and its clients should be able to operate in low latency. In addition, the system should ensure that the audio streams of all clients are kept synchronized, preferably with an accuracy of a single audio sample. There is a project known as JACK (Jack Audio Connection Kit), that tries to achieve these goals. JACK, however, uses shared memory to achieve them, and thus trades off the ability to distribute the system over a network, which is probably a fairly reasonable design decision when real-time is of especial importance.

An important part of client-server systems is the protocol used to exchange data between processes. Sound can be fairly straightforwardly represented as samples of variations in air pressure, with a small amount of information about the used encoding, such as mono/44.1kHz/16bit/linear/little-endian. Because even simple protocols can be perfectly adequate for these systems, it is not an objective of this thesis to consider protocol design issues except as they relate to the core problems of this thesis. Internally real-time audio servers can be fairly simple, but it turns out that they depend on operating system in their essential characteristics. This means that building them — if we want to consider the task as a whole — becomes also an issue of operating system design.

1.2 4.4BSD Unix and its derivatives

The operating system family that we are primarily interested in is 4.4BSD Unix and its derivative systems. BSD Unix was an operating system developed in University of Berkeley, California, in United States. It has been quite influencial in the history of Unix systems, for example the widespread use of the Internet protocols can probably be attributed to the fact that their reference implementation code was written into BSD, which has mostly been rather liberally licensed. The liberal licensing policy has also meant that the last "real" BSD system, the 4.4 release, has served as a basis for fully open source variants of the system. The current major BSD variants are OpenBSD, NetBSD, FreeBSD and DragonFlyBSD. The open source -nature of the these systems, coupled with a history of an academic research system, should make them as ideal systems for practical academic systems' research. Though rooted in 4.4BSD, they have taken somewhat different development paths, with designs that have some interesting consequences to real-time applications.

As previously mentioned, the original BSD was not designed for real-time applications. Among the current variants, the OpenBSD system is possibly the most faithful follower of the historical design. On the other hand, NetBSD shares a lot of code and similarities with OpenBSD, which is only partially due to the fact that OpenBSD was forked from NetBSD in 1995. Both systems still lack features that are commonly associated with real-time systems. It could be mentioned that the systems have other goals, such as portability for NetBSD and security for OpenBSD, which might be considered more important than real-time by their developer groups. Bringing real-time to systems with traditional Unix design often involves some serious redesign of the system kernel, which can be a problematic change for some Unix systems. But then, this does not necessarily mean that developing real-time features

to a system would lead to compromising other possible goals of the system.

OpenBSD is of particular importance to this paper, because it is used as a testbed for a test server and client applications, written for this thesis. The purpose of these programs is to gain some insight into the suitability of various IPC mechanisms for passing audio and related control data between applications. Because the test applications have purposefully been kept fairly simple, this means that the test results should tell more about the suitability of OpenBSD for real-time, than about the used test software. The reasons for choosing OpenBSD for this work are not of any special significance, it should be considered merely as a "good, somewhat typical Unix system".

The other two open source variants of BSD, the FreeBSD and DragonFlyBSD systems, have paid more attention to real-time performance issues. DragonFlyBSD is actually a fairly recent fork of FreeBSD, so it shares most of the code with FreeBSD, but is developed into a rather different direction. These systems have more of the features that are typically expected from real-time systems, and probably offer considerably better real-time performance than NetBSD and OpenBSD. It would be interesting to do testing with the test servers on FreeBSD or DragonFlyBSD as well. This would, however, involve at least some rewriting of the used test software to work on a different kernel audio API. Furthermore, if software used the real-time interfaces provided by these systems, the software would be less easy to port to systems such as OpenBSD.

BSD has also had some influence on other systems, and even spawned some commercial derivatives. SunOS by Sun Microsystems was based on BSD, until the advent of Solaris 2 that adopted System V as its base. Early Solaris systems took real-time issues into consideration in its design, which means that bringing real-time to BSD was a treaded ground even ten years ago. The design steps taken in contemporary FreeBSD resemble the ones taken in Solaris. Contemporary Macintosh OS, the Mac OS X, is also based on BSD and especially FreeBSD, even though the base system has code from Mach and NextStep as well. Nevertheless, Mac OS X has some ways to better its real-time performance, which points out another set of possible solutions for current BSDs. Unfortunately, this paper is sorely lacking in presenting these solutions.

Because this thesis examines real-time audio servers from operating system perspective, an introduction to BSD system design, as it is relevant for audio servers, is given in the next chapter. It is followed by a chapter of explanation of issues related to real-time audio servers. The next three chapters focus on the main factors in operating system design that have high relevance to real-time audio servers. The

first one mainly discusses process dispatch latencies and how they relate to kernel design, but also includes some other concepts that are often useful for real-time applications. The second one of these chapters is about process scheduling algorithms, and the third focuses on interprocess communication. These chapters are followed by a chapter on real audio servers and testing, followed by conclusions.

2 System design in BSD Unix derivatives

The 4.4BSD system follows the traditional design decisions in Unix-like systems — it could also be said that in many cases it was the system that set those designs. In 4.4BSD, the operating system is split between a kernel and a userland. The kernel runs while processor is in a privileged mode, able to access memory directly, and userland programs run in unprivileged mode. The kernel controls access to various resources needed by user programs, that can access these resources by calling the kernel with system calls. These mean requests, where a process execution suspends for the time the kernel executes some code on behalf of the process.

4.4BSD kernel is a monolithic kernel, meaning that various subsystems, such as filesystems, device drivers and networking code, that could technically be written as userland programs, are part of the kernel. Thus, they run in kernel mode, and share the same address space, having access to the whole of physical memory. Parts of the kernel form the "bottom half of kernel", which means code that gets executed only as hardware devices demands it, through hardware level interrupts. Some services or background work that the kernel must handle are written as processes that spend their whole execution time (typically the whole time a system is up) in kernel mode. Most parts of the kernel form the "top half of kernel", and run when processes execute system calls. 4.4BSD has a preemptive scheduler, which means that the execution of any user process may be suspended as it is executing in user mode, after which processor resources can be given to other processes.

The above are common design decisions in most non-Unix operating systems as well, and are shared by all the current open source 4.4BSD derivatives. What follows is a closer look at processes, interrupts, and some basic principles of the 4.4BSD scheduler, I/O model and interprocess communication. It is pointed out if some things differ significantly in current BSD variants, and some more complex concepts are deferred for closer discussion in later chapters.

2.1 Processes

Processes are executions of images in memory, where an image means a computer execution environment that "includes a memory image, general register values, status of open files, current directory and the like."[33, p. 7] Processes are isolated

both from the system and each other by means of a virtual address space, provided by the kernel. Thus critical system data is protected from direct access by programs, and programs also can not directly manipulate each others' execution.

Processes are typically created through the use of *fork(2)*[3] system call. The first program loaded into memory by kernel is always `init`, and all the other processes are descendants of this process. When a process forks, it is split into two independent processes, that have their own copies of the original memory image. However, the virtual memory subsystem in BSDs implement *fork()* by using copy-on-write -semantics, meaning that data does not actually get copied except when either the child or parent process tries to modify it. The two processes also have their own process IDs and can control their actions depending on whether they are a child or a parent process. This is the basis for multitasking in all Unix systems.

4.4BSD provided another system call for process creation, the *vfork(2)*[3] call. It is an optimization of *fork()*, with the difference that it will not create a separate address space for a new process, yet also mandates that the child process executes a new program immediately after the fork. It was historically more useful when *fork()* was less efficient, yet all current BSD derivatives include it. The contemporary BSDs, with the exception of NetBSD (that has a similar *clone()* call), also include *rfork(2)*[26] system call, that allows more control over which resources are shared between parent and child processes. It is, for example, possible for them to share an address space.

Even though *rfork()* can be used as a simple way to create processes that have some shared memory segments, a more typical way to achieve similar effect is to use threads. A process may contain one (implicit) thread or many threads, which mean separate executions of a process, and they share the process address space. This terminology is not, however, totally consistent among Unix systems. 4.4BSD did not have support for threads, except for some preliminary work towards them: "The 4.4BSD distribution did not have kernel-thread support enabled, primarily because the C library had not been rewritten to be able to handle multiple threads"[19, p. 81] The current BSD derivatives include some kind of a threading library, implementing the *pthreads(3)*[26] commonly used in Unix. On the other hand, it is also possible to base a threading system on top of *rfork()*.

The implementations differ on whether the operating system kernel is aware of any separate threads in process address space, or whether it is not. It is also possible for separate processes to share only a part of their address space. This can happen via the *mmap(2)*[3] call, or through some implementation of System V IPC interface. The former facility was already a part of the 4.4BSD system, the System V interface can only be found in current BSD variants. The use of threads and shared

memory typically demands more complex programming techniques, sometimes also meaning less robust programs.

2.2 Interrupts

Interrupts are events that change the normal flow of execution of a running process. There are essentially three kinds of interrupts that can occur in 4.4BSD. Hardware traps are a kind of interrupt that occurs when a process does something that the system should react on, such as trying to divide by zero. System calls are implemented as hardware traps on most system architectures. Device interrupts occur as hardware devices demand some action from the system, typically related to I/O between the device and the kernel. Another class of interrupts are software interrupts, which mean interrupts not generated by hardware events, but which the system still uses in order to arrange task priorities.

For every interrupt the kernel has associated an interrupt handler. Interrupts are prioritized, so that a high priority interrupt can preempt or delay the execution of low priority interrupts. Each interrupt has an interrupt priority level (IPL) determining the priority of the interrupt. The system execution always happens on some priority level, normally zero. Processes executing in kernel can modify this level. Interrupts of some IPL are blocked (or, deferred for later handling) until system priority level drops below that IPL. Interrupt blocking is necessary to ensure that interrupt handler code can not access or modify data which may be used by some process:

> Therefore, whenever the base-level code is about to update a data structure that is shared with an interrupt handler, it first disables interrupts, executes the critical section, and then reenables interrupts. The act of disabling and reenabling interrupts implements mutual exclusion. [36, p. 163]

As blocking interrupts too frequently and too a long time may disrupt proper functioning of the system, the number of critical sections should be low and their length short. However, proper use of IPLs and interrupt blocking ensure that interrupts will be served in a controlled manner: important tasks will be made on time, and system structures are properly protected from interrupt handling code.

Clock interrupts are specific kinds of interrupts that are very central to system operation. Interrupts are called *ticks*, and a typical frequency for clock interrupts is 100Hz. This interrupt frequency also sets the accuracy that the system can provide for processes that depend on proper timing for correct execution, even though

modern BSDs provide a *nanosleep(2)*[26] system call to get around this limitation. Clock interrupt handling drives many important system tasks, such as accurant timekeeping, process scheduling and statistics gathering. It also drives the processing of periodic events, among them posting a real-time timer signal to processes requesting it.

Hardware interrupts have a parallel on process level, implemented by the system, and these are called signals: "The system defines a set of *signals* that may be delivered to a process. Signals in 4.4BSD are modeled after hardware interrupts."[19, p. 27] Signals are thus like virtual interrupts to a process "virtual machine", having a similar relation as virtual memory to a process memory space. Normally, when a process receives a signal, its normal execution is halted until the signal is handled by the process, or some default action is taken. Signals can also be blocked just like hardware interrupts.

2.3 Scheduler

Scheduler is the part of the kernel that apportions CPU time to processes. The scheduling algorithm in 4.4BSD is very much the same as the one known as traditional Unix scheduling algorithm. It is designed for timesharing systems, where each process should have a fair share of CPU, while favouring interactive processes over computing intensive processes. The scheduler uses several run queues, each containing runnable processes that have the same priority. The processes are chosen to be run from the highest priority queue that contains processes, with a time quantum of 0.1 seconds for each in its turn. Processes are, however, moved between queues depending on their resource usage. Processes that have to wait for I/O operations to complete typically are moved to higher priority queues, whereas processes that consume lots of CPU time are moved to lower priority queues. This means that applications with a typical interactive usage pattern will be given higher priorities, and thus will have a better interactive response.

Scheduler uses two values of the process structure to calculate process priorities: *p_estcpu* and *p_nice*. *p_estcpu* is an estimation of the process CPU usage, and *p_nice* is an adjustable factor that affects priority calculation. New user-mode process priorities (*p_usrpri*) are calculated using the following equation:

$$p\_usrpri = PUSER + \frac{p\_estcpu}{4} + 2 * p\_nice \quad [19, p.94] \tag{2.1}$$

In case the resulting *p_usrpri* value is not between PUSER and 127, it is then raised up to PUSER or lowered to 127. This priority calculation occurs every four

clock ticks (that means every 40ms with normal clock rate). Higher values mean lower priorities, meaning also that low *p_nice* values will result in higher process priorities. User-mode priorities, however, are not used when the process is in kernel mode. As a process enters a kernel, it will execute until it returns to user-mode, or, if it needs to wait for some activity to occur, its priority value is temporarily lowered below its user-priority. This kernel level priority depends on the activity that the process is involved in. Because low priority values correspond to higher priorities, processes blocked in kernel mode will have priority over other processes, and this way the time the process reserves kernel resources can be kept to minimum.

While the scheduler treats the *p_nice* value is a constant that can only be modified by the user, the scheduler does adjust the *p_estcpu* value, by incrementing it at each clock tick if the process is the one currently executing on the CPU. In addition, the *p_estcpu* is adjusted each second by

$$p\_estcpu = \frac{2 * load}{2 * load + 1} p\_estcpu + p\_nice \quad [19, p.94] \tag{2.2}$$

Here, *load* is the average number of processes in all the scheduler run queues. The point of this adjustment is that the CPU usage of a process is gradually forgotten, at a rate depending on system load. The above equation translates to forgetting ninety percent of the usage by the time of $5 * load$ seconds. In case the process has slept over one second, a slightly modified equation is used to calculate *p_estcpu* that takes note of the time the process has spent sleeping.

Modern BSD versions still follow this traditional algorithm. FreeBSD has added real-time scheduling policies in their scheduler, and has "idle" priorities for processes that should not run except when the system is otherwise idle. FreeBSD is also currently in the process of replacing this traditional algorithm with a new one, that should be more suitable for SMP and SMT systems, and offer better scalability.

2.4 I/O model and IPC

A very central concept in Unix has always been the filesystem, and the idea of representing many different entities of the system as files. 4.4BSD adheres to this idea to a reasonable degree. File descriptors may refer to objects such as ordinary files, pipes, sockets, physical devices such as hard drives, or virtual devices that provide access to kernel level subsystems. Depending on its function, each file descriptor is associated with some part of the kernel — possibly a filesystem implementation, a device driver, or the networking layer.

The subsystems implement methods that correspond to various I/O system calls, such as *open(2)*[3], *close(2)*[3], *read(2)*[3], *write(2)*[3] and *ioctl(2)*[3]. Programs that want to operate on files (or anything represented as files) in the system, obtain a file descriptor with the *open()* system call, and use it for subsequent operations. *ioctl()* is the system call that provides a device specific way of controlling device behaviour, and it is typically used when general I/O calls such as *read()* and *write()* are not adequate in performing some special operations.

A closely related concept is interprocess communication, or IPC, which means exchanging messages between processes through some communication facilities. It is possible to simply use the filesystem for this purpose, but 4.4BSD provided facilities such as pipes and sockets to do it more efficiently. When comparing pipes and sockets, sockets have the advantage of enabling communication between unrelated processes, nhat may also run on different computers on a network. One other way of doing IPC in BSD systems is using shared memory to pass data between processes. Processes may agree on common memory segments by obtaining file descriptors (by using *mmap()* in BSD) referring to the same memory resident file or by using a token constructed from a filesystem pathname (as in System V IPC).

2.5 Audio subsystems in BSD kernels

The original 4.4BSD system lacked an audio subsystem. Since then, the contemporary BSD systems have built audio systems into their kernels. NetBSD chose to implement the audio API used in SunOS, with a few differences. The FreeBSD Project wrote their audio subsystem to implement the audio API used in Open Sound System,[29] a commercial audio driver architecture for various Unix systems. The NetBSD audio subsystem has been inherited by OpenBSD, and the FreeBSD one by DragonFlyBSD. Both NetBSD and OpenBSD also contain an *ossaudio(3)*[26] library to emulate the OSS API on top of the native kernel API.

Because the graphics subsystem in Unix is customarily implemented in userland, as is the case with The X Window System, this raises an interesting question why audio subsystems are commonly implemented in OS kernels. There are at least two reasons why this can be a reasonable design decision. The complexity level demanded by a useful audio subsystem is considerably less than is the case with a graphics subsystem, and thus it can be well fitted into a kernel. The other reason relates to the fact that disruptions in audio streams typically result in awkward silences (or twitching sound loops with most audio hardware) that are commonly more disconcerting than sluggish motion in video. Thus, it can be desirable that

audio processing code is run at a somewhat higher priority than is the case with other code. Because the code that runs in a kernel is running at a higher priority than most application code (most, because some Unix systems make an exception with real-time processes), writing audio processing into the kernel alleviates this problem to some extent.

On the other hand, it should be noted that while writing an audio subsystem into the kernel may help with some real-time performance issues, such a design rationale is probably rather weak in any system that support real-time process scheduling, and possibly in others as well. In addition, the phrase "useful audio system" used above does not mean "adequate for all purposes". The kernel level audio systems in BSDs mostly provide a way for applications to use the essential features of audio hardware. They may compensate for the lack of audio encodings in some hardware devices by converting audio from one sample rate and/or encoding to another, but generally do little more than that regarding real audio processing. Some hardware devices provide features to process audio, such as reverberation effects, but to access these on BSDs one usually has to bypass the audio API.

These shortcomings are probably not too serious. More problematic is the fact that access to audio devices is exclusive. Following the traditional Unix I/O model as explained in the previous section, audio devices are represented as files in the filesystem. Files such as `/dev/sound0`, `/dev/sound1` and `/dev/sound2` each represents an *audio(4)*[26] device. To play audio, an application opens an audio device file, and writes some audio data to it. To record audio, an application reads from the same file. In case the system would allow two or more applications to open the device for playback, normal filesystem semantics imply that applications have to contest for the device. The device would then play audio from the application that simply "got there first" — likely resulting in sheer cacophony.

This problem could be solved by device cloning, where each application opening the audio device would gain access to some "virtual instance" of the device instead. To make this possible, the audio system would have to at least mix the audio streams into one and then write the mixed stream into the real hardware device.

However, modifying the audio system to do this might prove to be somewhat difficult to do cleanly if compatibility with the current audio API is necessary. This is because the current API has many features that specifically affect the behaviour of actual hardware devices. This concept of virtual audio devices have been implemented in FreeBSD slightly differently: the filesystem contains special files for each virtual audio device. But, even though virtual devices are a solution to the exclusive property of audio devices, they alone do not provide the kind of flexibility from the

audio system that some applications may demand.

Using audio devices is not limited to reading and writing. An interesting feature provided by some audio hardware and drivers is mapping the device to process virtual memory. After a mapping is done via *mmap()* system call, a process can read from and write to an audio device simply by accessing a part of its own memory space. As a hardware device is using the same physical memory space to do its I/O, doing this requires some care. Consequently a process can request information from the audio system about how a device is using a mapped memory segment, for example about the memory offset at which the device is currently doing I/O.

Applications can also control audio devices with *ioctl()* to change used audio encoding, set half or full duplex operation and other device behaviours. Kernel level audio systems also implement devices such as `/dev/mixer0` and, in NetBSD and OpenBSD, `/dev/audioctl0`. These devices are not exclusive, but can be opened and used by several processes concurrently. `/dev/audioctl*` devices are like `/dev/audio*`, but only accept *ioctl()* calls.

The mixer device is used to control channel volumes with *ioctl()*. Related to audio subsystem is also the MIDI subsystem, as MIDI devices are commonly found in audio hardware. MIDI can be used to control the internal synthesizers on some audio hardware, or to drive external instruments or other audio devices supporting the MIDI protocol. MIDI is supported by all BSDs to some extent. However, as mixers and MIDI are not particularly relevant to the main topic of this thesis, they are not discussed further.

2.5.1 NetBSD/OpenBSD kernel audio internals

In NetBSD and OpenBSD kernels, the *audio(9)*[25] subsystem is split into two layers: the hardware independent upper layer, and the hardware dependent lower layer. The upper layer is a general audio framework that handles buffering, and provides the necessary glue from system calls to specific audio drivers. The lower layer contains drivers for audio hardware. During system startup, drivers in lower layer for each found audio hardware device will attach to the upper layer. It does this by hooking (that is, setting C function pointers to appropriate memory addresses) a required set of its functions to the audio interface structure provided by the upper layer.

The audio subsystem code gets executed in both of the following situations: a process makes a system call that should be handled by the audio subsystem, or a real audio device makes a hardware interrupt. Interrupts are handled as they

happen, except when the effective IPL is higher than `IPL_AUDIO`, in which case the handling is deferred until the IPL is lowered below `IPL_AUDIO`. Interrupt handling may be interrupted only by interrupts that have higher priority than audio interrupts, but these include basically only clock interrupts and anything that manipulates scheduler run queues. [28, `/sys/arch/i386/include/intrdefs.h`]

System calls are also handled as they happen, except that they are subject to interruption from all hardware interrupts, and in case they block, they need to wait until required resources are free and higher priority kernel tasks are completed. Thus, the most urgent activities should occur at interrupt handling. Everything else should be done at ordinary system level, since those sections should still be kept as short as possible in order to not interfere with normal kernel operation.

Audio interrupts occur as audio hardware copies data from kernel audio buffers when playing, or as it needs to copy data to buffers when recording. Hardware devices use DMA (Direct Memory Access) to access physical memory. As DMA I/O has completed moving a block of audio data, an interrupt is generated and handled by the audio subsystem. It might seem preferable to copy each sample from audio hardware immediately as it is recorded or it is needed for playing, but this would mean that audio interrupts should happen at a sample rate, typically 44100 times per second. However, this per-sample copying would be quite ineffective and infeasible. This means some buffering is necessary. Kernel maintains a buffer for both playback and recorded audio, used as "rings", so that when audio is read from (written to) the end, the reader (writer) has to jump back to the beginning. These buffers are used as audio data is copied between kernel and hardware, by using DMA.

Buffering, however, introduces latency. As audio is moved in blocks, the used blocksize sets the lower limit of average input and output latency. After a block has been read/written by audio hardware, an interrupt is generated. For the time the interrupt is handled, and especially because handling may be delayed by other interrupts, the hardware needs to be able to read more audio: it will be reading another audio block. For this reason kernel audio buffers have to be of the size of at least two audio blocks, one to be currently read by hardware, and the other for audio that follows. For example, if a process writes data to an audio device, it will write into a buffer segment that is currently not read by hardware using DMA, and latency depends on how close to the block the DMA copying occurs. There might be ways to get around this limitation in kernel level, or using *mmap()*, but that is problematic.

Working around such latencies introduced by blocksizes is also unnecessary. The default blocksize in OpenBSD is set to correspond to 50 milliseconds of audio, which

means approximately 8800 bytes of data in CD-quality audio. Nevertheless, the system allows to set it to as low as 32 bytes, corresponding to audio of about 180 microseconds. [28, `/sys/dev/audiovar.h`] That should be perfectly acceptable for even the most demanding audio applications. On the other hand, using blocksizes as low as this may not be possible in practice: for example, various audio hardware may have some limitations, and audio interrupts occur so frequently that associated inefficiencies will outweight the latency benefits.

Besides the blocksize used, another factor that affects latency is how kernel audio buffers are used. There are separate buffers for playback and recorded audio data. Their size is patchable, but have to be at least 512 bytes, and buffer size also sets the upper limit for blocksize, because at least two audio blocks need to fit into buffer. Normally, when a process writes to an audio device, the buffer takes in as much data as it can take. This means that each byte written will be read by audio hardware after an average number of $buffersize - \frac{blocksize}{2}$ bytes are read first. Because audio data size corresponds to playing time, it is possible to calculate the audio latency introduced by buffering:

$$\frac{buffersize - blocksize/2}{audio\ data\ bytes\ each\ second} \tag{2.3}$$

The number of audio data bytes per second depends on audio encoding. For example, a second of CD-quality audio data has 44100 samples, where each sample is represented by sixteen bits, or two bytes. Because CD-quality audio has two channels, this means $2 * 2 * 44100 = 176400$ bytes of audio each second. As the default playback buffer size in OpenBSD is 65536 bytes, while the default blocksize is 8800 bytes, we can calculate that the audio latency introduced by the kernel level audio system is approximately

$$\frac{65536 - 8800/2}{176400} \approx 0.347 \tag{2.4}$$

seconds. Using large buffers helps ensuring interrupt free audio stream, because an audio application can now take a longer time between writes to an audio device. This is good for applications such as media players, but highly undesirable for interactive applications such as games, software synthesizers, or internet telephony.

For this reason, it is possible to control buffer use by setting watermarks. The high watermark sets the upper bound to buffer use. The low watermark controls the amount of data that is considered to be enough — writes to an audio device will block until data in the buffer will drop below this level. Watermarks are expressed as integers, and they are multiplied by the used blocksize to get the number of bytes

they correspond to. Obviously, a low watermark value has to be at least one smaller than a high watermark value. Setting it less than that has the benefit that less writes are needed to write to an audio device, but this also increases the risk of buffer underruns (resulting interrupted audio).

Therefore, to ensure good latency in kernel level audio system, the high watermark may to be set to a low value, so that for each audio sample the time spent in the buffer can be minimized. This should be assisted by setting a low watermark to the highest possible value to decrease the likelihood of buffer underruns. Watermarks control only playback behaviour, recorded data can be returned to processes as soon as the audio hardware DMA write operation has completed. On the other hand, processes do not require such restrictions on buffer use if they itself enforce low latency. For example, if a process reads from audio device, processes it and then writes back to audio device, the output buffer should not get filled past two blocks of audio, because the process is not producing audio quickly enough to fill it totally. Here, the audio input functions as a timer controlling process audio output flow, but this timer could just as well be a system clock or some other external synchronizer.

While watermarks can affect latencies and setting them properly is useful, it may be more important to adjust the audio blocksize appropriately, as it determines the minimum for latency introduced by the kernel level audio subsystem. Real-Time processes should set the blocksize to a relatively low value. Audio hardware may adjust that value to better fit what it can support, however. It can be useful for audio applications to query the used blocksize and use that (or possibly its multiples) when writing audio to kernel, as this will minimize the number of needed system calls.

Considering the above points, it seems that audio dropouts should happen only on three occasions:

- IPL is above `IPL_AUDIO` for a longer time than it takes from an audio device to read a block of audio data
- an audio output system call blocks, for example when trying to write to a full buffer and some kernel activity occurs above the audio handling priority level, and this activity will take such a long time that the audio buffers will be drained before the system call execution may resume again
- the application writing to kernel audio buffers is not getting enough CPU time to output audio data

The corresponding situations may also occur with audio recording, for example a buffer overrun may occur if an application can not read from a recording buffer in

time. The first one of these is highly unlikely, and should only result because of a kernel bug in interrupt handling. The second occasion should also be rare, because buffers are generally sized well enough to ensure they are not drained before audio related system calls may resume.

The third occasion, however, is most surely the most probably reason for audio dropouts. Two major factors contribute to this. First of all, the process scheduling policies may not be giving enough CPU time to the reader/writer process, or processes that affect its execution. The other reason is that the process dispatch latency provided by the kernel to the process may be suboptimal. Aiming at low latency will highly increase the risk of audio dropouts, which is why ensuring good real-time reliability necessarily involves proper scheduler and kernel designs. For these reasons, this thesis will focus on kernel designs as they relate to process dispatch latencies in chapter 4, and on various scheduling algorithms in chapter 5. However, before these issues are tackled, some consideration of the problems involved in real-time audio servers might be in place. This is the focus of the next chapter.

3 Real-Time audio servers

The introduction already covered some of the reasons why real-time audio servers might be a good idea. A server can help in providing a unified audio API across platforms, it can give flexibility for audio processing that can be difficult to achieve in kernel subsystems, and it can be used to provide network transparency for applications. Such servers should be useful as a part of any end-user desktop environment, but application areas that can benefit from them may include anything from teleconferencing to virtual music studios.

Since the concept of real-time system is very central to this thesis, it is probably worth to consider what we mean by it and how it relates to audio systems in general.

3.1 Real-Time systems

Real-Time systems differ from non-real-time systems in that "the correctness of a real-time system depends not only on the logical result of the computation, but also on the time at which the results are produced." [4, p. 2] Real-Time systems may be categorized whether or not the real-time requirements are absolute or merely important but not absolutely essential. Systems which are at fault if task deadlines are missed are called hard real-time systems. Systems where the system can still function correctly even if some deadlines are missed are known as soft real-time systems. Soft real-time systems are not the same as interactive ones, in that in interactive tasks there are no explicit deadlines, but merely a wish of fast response times.

Audio systems fit well into the category of soft real-time systems. The system has deadlines to produce audio output within some given time, and if those deadlines are not met, the practical effect is typically a sudden silence in perceived sound. However, the system can not be characterized as hard real-time, because breaks in sound are merely annoying but not critical faults, and the system can still produce correct audio output even after missing an occasional deadline. This also means that for purposes of real-time audio it is enough if the system (meaning the operating system and the audio system that runs on it) can guarantee low latency with some reasonable probability. Reasonable, though, for most uses means something very high, such as about 99.99% of all processing time.

Software audio systems typically process the digitized sound data in some way, accepting input and producing processed output. The lag from input time to output time, or in other words, the audio latency, should be kept fairly small. On a programming level this involves a question of choosing buffer sizes for audio data processing, where large buffer sizes cause higher latency, which is of course less satisfactory from real-time perspective. But higher latencies associated with larger buffer sizes also mean longer deadlines, which reduces the risk of sudden silences in system output. Typically simple audio playback operations use high buffer sizes, because there are no input events and thus latency is irrelevant — low latency can be traded off for achieving a non-interrupted audio output stream.

Many techniques like Rate-Monotonic Analysis, that are used in implementing hard real-time systems, are not very well applicable when improving some generic operating system's soft real-time performance. Hard real-time systems can and have to assume, that tasks' periods, frequencies, resource usage patterns, deadlines and other such attributes are known, and can be used when considering job scheduling and task priorities. However, in many soft real-time systems such as servers and workstation audio systems, resource usage can not be predicted beforehand. Users can request the system to handle possible and impossible loads, and the system should be made to handle them as efficiently as possible, and degrade gracefully when necessary. To some extent it can also be left to user's responsibility to ask the system to perform only tasks that are feasible, and possibly also advise the system on tasks priorities. Some of this responsibility might be handled by the applications using system resources.

3.2 Are audio servers necessary?

It is a reasonable question to ask why bother with servers at all. Technically, an audio server is merely glue that connects together other applications and a kernel level audio system. But such functionality may be provided by other means as well. One alternative way of doing it is to make applications into plugins that can be dynamically loaded into a host application. The host would then handle the actual passing of audio data among application components. This design might reasonably be called "the monolithic host application"-style. It has the benefit of efficiency, as passing data does not need to be done or assisted by the operating system. An application can also exercise greater freedom in deciding how the audio data is passed among components. Furthermore, problems of scheduling applications correctly should be less problematic, as the host application can have

full control over shared object application scheduling.

This kind of shared object based approach is indeed the dominant paradigm in performance sensitive audio work. Most audio interconnection technologies used in pro audio applications in Windows and Macintosh platforms use this paradigm.

The downside of this architecture is that applications — that are shared objects — share an address space, meaning that a single misbehaving application may disrupt the proper functioning of others. Moreover, separate applications can not be distributed over a network and put to communicate together, because they need to be loaded dynamically into a single host application. Some other additional flexibility, that could be achieved with better modularity and with ability to interfere with data flows from outside the application, is possibly also traded off. Conceptually this designs is quite alien to Unix, which has had a culture of small tools that can be connected in novel ways using operating system facilities.

The basic difference between the plugin approach and using a server is their relation to operating system IPC mechanisms. The plugin approach is fairly operating system agnostic, as it does not depend on the system for passing audio data, and the operating system is not aware of what happens inside applications. In contrast, the server approach depends on the system to communicate with clients.

Drawing a line between these approaches, however, may not be always be clear, because a server may have a role of only assisting the communication. If shared memory is used, the operating system may have only a secondary function in communication, as it is mainly needed to arrange the shared memory segments for applications. Nevertheless, in this paper we are primarily interested in real-time audio servers from an operating system perspective. Unix has historically had a stronger background of integrating IPC facilities into the system than Macintosh or Windows has had, and there is no reason that its audio interconnection systems should mirror those of other systems.

3.3 Server requirements

Ideally, a properly designed RT audio server should fulfill the following requirements:

- run as a user process
- have exclusive access to audio device
- provide a common API for audio programming

- provide a way to control its operation
- be able to multiplex audio streams from clients
- be able to route audio between clients
- be network transparent
- be extensible, for example to offer some audio processing features

Some of the points presented above are debatable. It might be reasonable to provide these features from a kernel level subsystem. However, the complexity level demanded by these features is such that providing them from an application running in user space seems clearly a saner way than doing it in kernel. This is especially true if extensibility through some shared object interface is called for.

Nevertheless, implementing the equivalent software in kernel is more difficult to code, difficult to debug and risks system stability. Some of the features such as communicating with clients over a network are more naturally done in userspace. The overall development trend in operating systems, albeit it being slow, has long been towards providing services from userland processes, instead of them being in kernel.

For these reasons, it is assumed from this point on, that audio servers should be written to run as userspace processes, and that kernel level functionality should not be extended except when it is necessary to help the proper functioning of a userspace server. Moreover, a feature such as network transparency should be useful, but not necessarily an important requirement. Audio servers can be useful in a local setting only, and designing for network transparency may have a detrimental effect on latency.

In addition to the requirement list above, audio servers that have real-time correctness requirements should also be able to

- operate with low latency
- synchronize its clients
- synchronize with other servers, if any
- keep sync with graphics system, if any

The ability to synchronize clients may not be a relevant feature for a common desktop audio system, but it is necessary in case a server is used to mix and route audio streams that have interrelated timing requirements. This is the case with music,

where each audio stream may contain a separate instrument. The capability of running several servers concurrently is probably not nearly as useful, but can be, for example in cases where each server accesses a separate physical audio device, and servers need to co-operate.

It is also useful that server has some kind of synchronization mechanism with graphics system, commonly The X Window System. This is necessary, as, because of audio buffering, the audio sent to kernel audio system may not reach the actual physical device except after a noticeable delay. It is possible to get good audio/video sync by using only a little buffering for audio data, because that results in low latency.

However, because buffering is useful for interrupt-free audio, an application should possibly know about which part of the audio stream is actually being played at any moment, instead of merely being sent to the kernel. The application could then use that data to co-ordinate its actions with graphics server. The kernel level audio subsystem can provide such information, but if the audio is sent via the server, the server should pass the information to the audio client. It may also be argued that, instead of handling the synchronization problem in the application, the audio and the graphics server should co-ordinate to provide an adequate audio/video sync, transparently to the application.

While the other requirements are more inherent to the actual server design and implementation, the latency requirements are much more dependent on the real-time features that the underlying operating system can provide. A central question here is: how much latency is acceptable?

3.4 Acceptable latencies

As sound travels through some substance, be it air, electrical wires, or computer memory, some delay is always introduced. Inside a computer, the main source of latency is buffering, but using small buffers yields a greater risk of interruption. How short latencies are achievable depends on the characteristics of operating system and the audio application software. But how good should operating system real-time performance be in order to be useful for various tasks?

When considering simple playback of audio streams, possibly combined with video, the acceptable maximum for latency is mostly defined by the interactivity demands of the media player. That is, when a user presses “play” or does some equivalent action, latency means the response time to start the actual playback of the audio stream. This delay should probably not be more than half a second, so that the interaction does feel smooth enough to the user. Unless a system is under a

heavy load, no commonly used operating system is incapable of accomplishing this level of latency.

Another interesting task where latency is relevant is internet telephony. In telephone conversations, if the delay introduced by audio transmission gets higher than about 0.2-0.3 seconds, the delay starts interfering with normal flow of conversation, while participants also take notice of it. The latency in internet telephony occurs, because audio has to pass through many computers which all add some delay to the stream. However, latency may also be added by computers at both end points, and should ideally be kept fairly low.

Another class of applications is the wide range of interactive applications, that resemble the media player situation above, but where audio should respond to user actions more or less immediately. These include games, desktop audio effects, and synthesizers where user has direct control over what can be heard from speakers. For these applications, 0.2 second latency is often too much, and desirable response time may be well below 50 milliseconds. An article on Sound On Sound suggests, that, while some musicians may be comfortable with longer latencies, "musicians will be most comfortable with a figure of 10mS or less — the equivalent latency you get between pressing a key on a MIDI keyboard and hearing the sound"[38]

There are even more critical cases, for example when a computer is used for real-time audio processing and the output audio stream gets mixed together with the original audio stream. In these cases, the soundwaves may cancel each other in a special way, creating a comb-filter effect, that is perceived as if the sound is somehow weirdly distorted. Paul Davis suggests that "when the latency or the delay between action and effect gets above about three milliseconds you start to make it hard for musicians and audio engineers to do the work".[6, 13:52-13:59] Furthermore, if an audio track is shifted relative to other tracks, even shifts of a millisecond can have a perceptual effect on listeners. Nevertheless, three milliseconds should be a good guideline, because that is about the extent of the latency that the best current Unix systems can provide, when considering that some time needs to be spent also for doing the actual audio work. And for many applications, longer latencies can be totally acceptable as well.

3.5 The context switching problem

One characteristic of the client-server model is that clients are independent applications running in their own virtual address spaces. While this has its benefits, it means that each time a client should run, the system has to perform a context switch.

This means changing the processor state so that it contains and can access only data that is part of the execution environment of the client process. If the process tries to read or write some other data, a page fault is triggered, and the kernel is called to handle the situation. This offers protection to other processes running on the same machine.

Let us assume that we have a goal of designing and implementing a real-time audio interconnection system using this model. Each client processes audio data that corresponds to what we are actually hearing at the same time. This means that, within the same time interval, each client should have a little bit of CPU time to do its audio processing. This, in turn, results in at least one context switch per client within each interval.

If system calls are involved, as is true in most cases where data needs to be passed between processes, there needs to be more context switching. This is necessary, because in order to handle system calls, execution has to step into the kernel, which involves a context switch, and out, which involves another. The monolithic host application model does not have this problem, because there is no need to do context switching so that clients can execute or pass data between themselves — because they share their address space.

Now, assume that we also want to achieve a latency of about three milliseconds. This means that all this context switching should occur in the timeframe of three milliseconds, and in between the clients should also have some time to do their real work. So, depending on what the context switching times are, we can see that this might become a real problem if we want to achieve very low latencies. But what is the cost of context switching then?

Context switching involves changing a CPU's state, which means changing register values to fit the new execution environment (a program thread). Switching between threads in user space involve mostly only this cost, and it is relatively cheap. System calls involve switching between user and kernel level, which means also changing the CPU privilege level. According to Coulouris et al., in this case "the cost is therefore greater but, if the kernel is mapped into the process's address space, it is still relatively low." [5, p. 224] This is commonly the case with operating systems, most probably including the BSDs. But Coulouris et al. continue:

> When switching between threads belonging to different execution environments, however, there are greater overheads. Longer-term costs of having to acquire hardware cache entries and main memory pages are more liable to apply when such a domain transition occurs.[5, p. 224]

This means that the cost of the context switch is not only the time it takes to swap CPU register values. Modern CPUs use caches to speed up their performance, and at context switch time some cached data has to be invalidated. This is because processes run in their virtual address spaces, and when the CPU switches to run another process, the process should not be able to use cached data that belongs to another process. Thus, context switching between processes is potentially much slower than between threads within a same process, but this actually depends on several factors. But in most cases this may not matter, as Vahalia affirms: "the difference in performance between the two types of context switches may be insignificant, depending on the hardware and applications involved"[43, p. 139]

One factor is the cache implementation, that depends mostly on hardware. However, some cache designs may need management by the operating system at each context switch. In case of virtual caches, where cache contains virtual addresses, the same addresses may be used by different processes, and thus the system needs to flush the cache at each context switch. Curt Schimmel explains:

> If context switches happen too frequent, as can happen with highly interactive UNIX applications, then the benefit of having the cache in the system will be reduced because of the low hit ratio and the operating system overhead required to flush the cache. Large virtual caches are best suited for compute-bound, batch-oriented application environments where context switching is infrequent.[36, p. 68]

In an alternative cache architecture, virtual caches are associated with keys. These keys are associated with entries in CPUs Translation Lookaside Buffer, which contains cross-references between real and virtual addresses. Each key may correspond to a different process, meaning that cache can contain data from many processes at the same time. Schimmel explains, that this cache usage pattern can eliminate cache flushing at context switch times. Whether this is possible is very much processor architecture dependent: at least the early processors of the i386-architecture (standard PCs) had the previous cache usage pattern, where flushing is necessary. In contrast, on processor architectures such as MIPS, the cache architecture may contain a fair number of keys. For example, MIPS R4000 can contain 48 keys tagging cache addresses, and caches need to be flushed only when address translations change, and not during a typical context switch.

Another example of using keys with virtual caches are most Sparc processors commonly used with Solaris operating system. In Sparc, the Translation Lookaside Buffer (TLB) can contain tags that allow identifying different contexts, which means

that they be simultaneously active in TLB. In a book on Solaris internals, the authors Jim Mauro and Richard McDougall explain this:

> This behavior offers a major performance benefit because traditionally we need to flush all the TTEs for an address space when we context-switch. Having multiple TTE contexts in the TLB dramatically decreases the time taken for a context switch because translations do not need to be reloaded into the TLB each time we context-switch."[21, p. 195-196]

A third cache architecture is using different cache pages for unrelated processes. This architecture also has the benefit of not requiring cache flushing at context switch time, but it can lead to less optimal cache usage, as processes can use only a subset of the whole available cache.

Thus, a critical factor that affects context switch times is cache usage patterns. In practice, this means that context switch times may considerably increase depending on how much memory do processes touch before a context switch. If a process uses a lot of memory, and thus a lot of cache, updating TLB and invalidating cache pages takes a longer time, resulting in a higher context switch latency. Moreover, the number of processes involved in context switches is a relevant factor, because the number of possible contexts that can be simultaneously active in cache varies among CPUs.

None of this should matter if the context switch latency performance was adequate to run several isolated clients, scheduling them rapidly to achieve distributed (on a process level), low latency audio processing system. Some look into benchmarks is therefore necessary to gain insight into this issue. Lmbench is a widely used benchmark to test the performance of computers running various versions of Unix. A paper[22] explaining the benchmark methodology shows also some context switch performance results. A curve showing the context switch performance of Intel Pentium Pro system (running Linux) shows that context switching times remain below ten microseconds even with a fair number of processes, if processes do not use memory almost at all between the switches. Ten microseconds can be considered to be low enough to cause no problems for real-time audio. However, if the memory area used by processes between switches is at the level of 64Kb, then, as the number of processes increase, a single context switch can take as much as 350 microseconds.

This means that if we want to run several isolated clients, and these need a fair amount of memory to do their work, achieving a latency of about three milliseconds can become difficult. Three milliseconds of CD-quality audio data take about 265 bytes per channel, so in practice the memory area touched by audio applications

does not necessarily need to be high. In practice, it depends on the number of channels involved, on how complicated processing the audio applications perform, and on other possible tasks they might have. Nevertheless it is totally possible that a rapidly context switching client-server system might spend a considerable amount of its time on those context switches, seriously affecting the real-time performance of the system.

From the same Lmbench-paper it can also be seen, that memory usage does not affect the context switch performance of other CPUs as badly as it does in the case of Pentium Pro. In fact all tested CPUs do worse than Pentium Pro when the number of processes and memory usage remain low, but in cases where these attributes are higher, they perform considerably better. Because the benchmark results (published in 1996) may not reflect the currently used generation of CPUs, a more detailed analysis is probably pointless — nevertheless, the architectural differences between CPU families are still likely to exist.

The problem can also partially be attributed to the common computing model, where a large application instruction set is used to process a small data-set. Most CPUs in desktop computers are generally designed to optimize for this, and depend on data caches (capable of storing the used data set) to perform efficiently. In most media processing, however, the requirements are backwards, as a small instruction set is used to process large amounts of data. This model demands a different architecture, where fast memory bandwidth is preferred over data caches. This is the architecture model that current game consoles have been moving to.[41]

It is probably necessary to do real benchmarks before it can be said how seriously the context switch times may affect real-time performance. Unfortunately, the benchmarks used in this paper do not measure the context switch times, and as such are not adequate to gain insight on this issue.

3.5.1 Working around the context switches

Since context switching between processes that run in their own virtual memory spaces might be a problem, it can be a good idea to see if the issue could be worked around. One way of doing this would be to adopt the "monolithic host application"-model. Here, when an application is executed, it would only contact the server to instruct it to dynamically load the application image into itself. The host would create a new thread for the application, and load it with *dlopen(3)*[26] or some equivalent function.

It appears that the issue when doing this is that an application may disrupt the

execution of another application, for example causing it to crash. In this case it is also pointless to run the applications with differing user credentials, which might also provide some security benefits in some rare cases. This procedure also can not be used if applications reside in different machines. Nevertheless, this technique does not affect the actual IPC methods chosen, so applications may still communicate with sockets or any other techniques they choose to use.

Unfortunately it appears that, at least in some threading implementations, applications need to be especially designed for this technique to work. This does not mean simply how the actual startup and dynamic loading is handled. It appears that a process cannot simply share an address space with another (in other words become a thread of another process) in order to save on context switch times. An example of this is given by GUI toolkits for X Window System, pointed out by Paul Davis.[6, 18:30–18:35] It appears that a process can not use two different GUI toolkits, with one thread using one and another thread using some other toolkit. This is probably because in most implementations, threads of a process share signals, PIDs and possibly some other attributes that GUI toolkits need in order to function properly. On the other hand, some other factors may also affect their concurrent use.

In OpenBSD and FreeBSD it is, however, possible to use *rfork()* to fork a process that shares an address space with its parent, yet has its own process structure. NetBSD has a similar *clone()* system call, whose use is discouraged. The GUI toolkit and some other similar problems may not occur in this case. It should also be possible to do a context switching between these processes without cache management overhead. Whether or not processes can share address spaces this way without any side effects is unclear. On the other hand, it does seem reasonable that separate applications should be run in different address spaces, and making applications share address spaces only because of some efficiency reasons does not look like good design. Working around bad CPU architectures with software hacks is not a good long-term solution, and ideally the efficiency problem should be solved where it exists (if it exists): in the CPU.

3.6 Audio libraries

By libraries we do not mean collections of audio samples, but shared objects that can application can link to and use. In typical Unix audio programming the audio device is accessed directly and controlled through *ioctl()* system calls, which can be adequate for many purposes. Yet building audio libraries is still a useful approach when creating features into an audio system.

One such benefit can be portability across platforms: if applications were written to use an audio API offered by a library, porting that library to different platforms would enable using the same API in many platforms. Audio libraries can also contain some useful audio processing functions, whose implementation in kernel might not be reasonable. Another benefit they can provide is API emulation. For example, OpenBSD and NetBSD have a library, that translates the audio API of Open Sound System (OSS) to their native API. This can be tricky. In the OSS emulation case, *ioctl()* calls used by OSS need to be redefined to use the library instead of the system call interface, which can actually cause some slight problems.

Audio libraries are relevant to audio servers, because they can be used to conveniently handle the communication between applications. This is what the X Window System does: it includes an X library that provides a convenient API for applications to use, and handles the actual communication with the X server. This technique should be useful for audio servers as well. Real-Time requirements do not change this situation significantly, but would benefit if the library has a low overhead, and ideally be thread-safe, since many real-time applications prefer a threaded programming model.

3.6.1 Audio API emulation

Audio servers also have a practical problem due to the fact that applications typically access the kernel audio devices directly. The proper long-term solution would probably be modifying these applications to use the audio server instead. However, this would mean that running an audio server would be mandatory. By using an application library, it could be made possible to modify applications to use the library instead, and the library could handle whether the kernel should be accessed directly or the audio server is used. Application library could also provide emulation of the kernel level audio interface, as is the case with OSS emulation library in both NetBSD and OpenBSD, and in ALSA (Advanced Linux Sound Architecture).

This technique is actually used by many current audio servers, except in a slightly different way. In systems supporting dynamically loadable libraries, which includes all BSDs on most CPU architectures, it is possible to achieve this by affecting the library loading mechanism. Applications that link to shared libraries, can be made to link to a modified version so that some of the functions are replaced with ones that redirect the audio data to the server instead of the kernel. The main problems with this technique are that it can not be easily made transparent for users, it does not work with statically linked binaries, and is difficult to use with application plugins

that want to access audio device.

A better solution would be that the actual device file be directly handled by a server process. For this, BSDs provide a special portal file system, where a directory is mounted so that accesses to it can be completely handled by some process. Marshall Kirk McKusick et al. explain the operation of this filesystem:

> When a pathname that traverses the location of the portal is used, the remainder of the path is passed to the process mounted at that point. The process interprets the path in whatever way it sees fit, then returns a descriptor to the calling process. This descriptor may be for a socket connected to the portal process. [19, p. 237]

It should be possible to write a server that implements files in this directory that behave exactly or almost exactly like the kernel audio device. This actually resembles the designs used in Plan9 operating system, and seems like a reasonable path to take in some areas of BSDs as well. The downsides to using the portal file system is that it is not generally used by any applications at all, and as such it is uncertain how practical this scheme really is.

Another way to solve this problem is to slightly extend kernel functionality so that processes could implement parts of a kernel, such as device drivers. One example of this used with Linux is FUSD[7] — a Linux Framework for User-Space Devices. As an audio device is accessed, kernel can simply pass the access information to the responsible process. This technique actually allows going even further and implementing an audio subsystem entirely in userspace. It can provide some of the benefits of running code in userspace, such as flexibility, and better system stability. On the other hand, because user space drivers need some extra privileges to the kernel, strict error checking of their operations at kernel boundary is necessary.

Similar developments have been going on in DragonFlyBSD, that has implemented in-kernel message passing system that should be usable by userspace programs. This should allow implementing whole filesystems and other kernel subsystems, which would also allow implementing a unified audio subsystem/server in userspace. In this case, a server could implement the kernel level API with no problems, and applications need not necessarily concern themselves whether they are accessing the kernel or some audio server.

3.7 Working around real-time requirements

It is obvious that a system that is not real-time can not emulate a real-time system, but in some cases it is useful to try to work around the system limitations. One simple case is the use of buffering, which is used to guarantee interrupt free audio stream, that would otherwise need a real-time system to support itself. In addition, features of audio hardware can sometimes help. For example, most audio hardware allows mixing of audio input to audio output, possibly with some processing, while audio recording can still be done by the system. Later the recorded audio can be played back while being processed at the same time, and this playback can use buffering. This essentially relies on bypassing the operating system for real-time critical work, which can sometimes be acceptable.

For audio processing it is in fact useful to separate two classes of real-time, based on whether the occurring real-time activities are deterministic or non-deterministic. In "determistic real-time", the audio stream can be precalculated based on some data that is already accessible. However, it is necessary that the audio stream is still played back at a correct point of time. This class might be called "pseudo real-time". In "non-deterministic real-time" the audio stream can not be precalculated, but instead some data from the outside world (such as audio input or a press of a button) affects the stream. In this case, the audio stream needs to be calculated within some delay period, depending on the latency requirements of the application.

This separation is relevant to real-time servers, because it can be necessary to distinguish between these stream classes. A server may be connected to clients that have different expectations about when their audio streams are played. Some clients may not care when playback occurs — such could be the case with music players. Some others may expect that playback occurs after certain delay period, so that audio is properly synchronized with video or some other audio. Yet another group of clients may expect as low delay as possible. It is also good to note that classes should not be client-specific, because we can imagine a client that has several audio I/O streams belonging to different stream classes.

A simple solution to this problem might be simply ignoring the non-deterministic class at server level. Clients can pre-calculate audio as much as they like, if they can, but give it out only at the moment the audio needs to be played out. The problem with this is that this needs a real-time operating system to work properly, with real-time scheduling priorities for all clients. Because most clients need to do work not related to audio, audio processing should ideally be isolated to separate threads that have real-time priorities. In addition, this also means wasting resources in real-time

systems as well, because using buffering is a more efficient solution when real-time is not an objective. In case audio bandwidth is a constant, optimizing for latency for no reason leads to unnecessary inefficiency.

Thus, it seems reasonable that a server should be able to deal with clients that have different latency requirements. This will actually bring about an interesting problem, regarding how a server should interface with the kernel level audio subsystem. A server commonly sends and receives audio streams, while routing, mixing and otherwise processing audio data, and its goal is to calculate a block of audio data that it can write out to the kernel level subsystem. But sometimes a server may not be able to calculate the audio block in time it should be sent to the kernel — it might be delayed by some client or for some other reason. In this case a server may simply wait until it has the audio block and then send it even though it arrives late — causing an interrupt in heard audio stream.

This is not an optimal solution, because in many real-time tasks it is actually better to simply throw away data instead of using it late. It should be a better idea to prioritize audio calculation so that there is always at least some data to be sent to the kernel. Therefore, a server should schedule audio streams according to their latency requirements. As clients can request some delay, or will simply accept some default delay, latency requirements for streams are known.

A consequence of this is that the Earliest Deadline First (EDF) scheduling algorithm should be adequate for the purpose of scheduling client streams. With EDF, the stream with the nearest deadline should always be chosen to be processed before others. This means that the processing of audio data related to low latency clients has the highest priority. However, when data processing related to low latency clients is finished, processing related to high latency clients can occur. The data from high latency clients can be buffered, ready to be sent to the kernel later at a correct time. This allows for graceful degradation of audio processing, as, in case streams with low latency requirements can not be processed in time, the streams that accept high latencies can still benefit from buffering and will not be interrupted.

To illustrate this, consider the case where a media player and an interactive game are running concurrently, using the same audio server. A media player can accept relatively high latencies, whereas an interactive game demands low latencies. It is the server's task to mix their outputs together and send the mixed stream to the kernel. If the streams are prioritized adequately, the stream from media player can benefit from audio buffering, and remain uninterrupted even if the audio stream from the game is not received in time, or otherwise can not be properly processed.

Because some audio buffering is used in the server, a question remains whether

any buffering should happen in the kernel. Depending on what clients demand, some kernel level buffering may be used, but in case any client requires low latency then kernel buffering should be kept as low as possible. This means a higher risk of dropouts, which should be unnecessary because a server has some audio data buffered, only it should be accessible by kernel level audio system. One solution to this problem is to run a server with real-time priority, which actually only drops the dropout risk, and works only if a system has support for real-time scheduling. However, the problem can be properly solved in non-real-time systems as well, by making the server and the kernel audio system to share the output audio buffer. This can be done by using *mmap()* system call to map the audio buffer to the server address space.

3.8 Audio protocols

It is not the purpose of this thesis to consider audio protocol design, and in fact most of the concepts presented here are relevant for any kind of distributed real-time system. Indeed, an audio server does not need to concern itself whether or not it is passing audio data or something totally different. As long as a client-server system implements low latency message passing, where message processing can use synchronization to some virtual clock, the system is theoretically useful for real-time audio. This idea is somewhat related to end-to-end -argument in distributed system design, which says that "functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level."[35]

Thus, when designing a client-server audio system, it can be useful to design the system so that it could potentially be useful for non-audio work. Alternatively, we could imagine an audio system implemented on top of some general system, by adding features that are inseparably audio related. And this is in fact what we do if we implement audio systems that use operating system features for their basic operations.

Nevertheless, it is still relevant to consider some points in protocol design. First of all, audio data representation is a fairly simple issue. For many purposes of audio servers, complex encodings such as *ogg vorbis* data format or *mp3* are not necessary. They can be useful only when bandwidth issues are a concern, which does not happen in settings where audio server serves only local clients.

When passing data through a network, such data compression encodings often become useful, but these situations not commonly associated with low latency re-

quirements. These requirements does not, however, mean that these formats can not be used: this mostly depends on the minimum size of data chunks that is needed by the encoding and decoding processes. Encoding audio data in different audio formats does not introduce latency in itself, but encodings create a limit for obtainable latency, depending on the size limits of data chunks. Commonly used formats probably do not have serious restraints on blocksize, meaning that they can be adequate for low latency audio.

Simple encodings such as using samples that correspond to fluctuations in air pressure are therefore adequate for most purposes. In these cases, audio protocol involves exchange of information about sample rates, bitdepth, channel count (if audio data is interleaved) and byte order, and then using this negotiated data representation. Questions about data representation are not, however, all that there is to procotol design. Some applications may require sending control information between a server and its clients: the format and content of this information should be decided in the protocol. An ideal protocol should also include a method for clients to inform a server about their latency requirements regarding the audio streams they produce.

A more important question in protocol design is, however, how should exceptional situations be handled by the server and clients. For example, what should a client do if the server needs to be restarted, or in general, what if the server asks for a client to use another server (running on a another computer) instead? Being able to do this can be a useful feature for a client-server audio system.

A more commonly occurring problem is that clients can fall out of synchronization, if the system can not schedule them correctly or for some other reason. Audio data sent between the server and clients may miss its deadline, and this may occur independently of the progress of either. The server should take a note of this happening, and inform clients about it. A properly written client should then take steps that ensure that it is in sync with other clients, if required, or otherwise make necessary changes to its behaviour.

For example, if the system can not ensure some level of latency for a client's streams, this means missing audio deadlines and manifests itself as frequent audio dropouts. For each dropout, a client may want to skip some audio processing and not send the corresponding data to server, so that it remains properly synchronized. The perceived result is that the client only appears silent for the duration of dropouts, instead of dropping behind the audio flow. Furthermore, if dropouts occur frequently, the client might be satisfied with higher latency, if that can mean less dropouts.

For this to work, the protocol should contain a method for a client to inform the server about its revised latency requirements. Thus, to ensure a robust client-server audio system, the audio protocol needs to cover some exceptional situations, and contain a way to pass control information between server and clients. This can mean such protocol complexity that implementing some parts of client-server protocols should possibly be done in a shared library that clients can link to.

3.9 Threaded programming

Writing real-time systems commonly demands a programming style that uses different threads within an application. If a program has two or more tasks, where some task is time critical and some other task may occasionally take a longer time to finish, then using threads may be necessary to ensure that the time critical task does not miss its deadline. Therefore, we should look whether threaded programming is necessary to do some real-time audio work, or if it can sometimes be avoided. Generally this is an application design issue, which this paper does not want to be overly concerned with. However, the design behind an operating system threading implementation can affect the suitability of threaded programming for real-time applications. In addition, an operating system may have features that make threaded programming an unnecessary venture on some cases.

In Unix programming, using threads have generally been avoided, and forking separate processes to do concurrent tasks has been a preferred technique, when this has been reasonable. The design of representing various objects as file descriptors allows for a more event-based programming style, where a program waits for an event to occur on a file descriptor (representing a file, socket, device etc.), and then can react to that event. The addition of kernel event subsystem and related system calls allows using this efficient programming model with a wider range of system events.

The event-driven programming model is generally used to create high-performance servers, including the most common implementations of the X Window System, XFree86 and its fork, X.org. The technique is efficient, because a server is always either serving a request or is waiting for a request. It also means that if a request is being served, another request has to wait in a queue until the server is ready to take a look at it. This is a problem for real-time work, because previous request may need work that takes a long time to finish, and new request may have a short and a strict deadline. Threading can solve this problem, but on the other hand may not be necessary, depending on tasks that a server is required to handle.

Threaded programming can thus be useful for servers, but it is probably more important model for clients. This surely depends on what clients do. But most audio clients probably have other tasks in addition to audio processing, such as handling GUI interaction with user, loading and saving files and so on. The audio task often has some real-time requirements, and therefore should have a higher priority than others. To solve this issue, threading can be used to ensure, that other tasks will not interfere with audio tasks. Threads are obviously not strictly necessary, because it is possible to use forking processes communicating through socketpairs, but for many programs it can still be a more practical programming model.

4 Real-Time Unix design

When considering real-time client-server systems, we need to keep in mind that the response times of clients and the server can never be determined only by application behaviour. In fact, the operating system usually plays a greater part in response times. If applications rely on the system for some of their operations, such as passing data between them, the role of the operating system becomes even more pronounced.

There are some features that are typically expected from a real-time operating system. Some way to affect scheduling priorities is necessary, typically by requesting static, real-time priorities for processes that require them. But scheduling issues can be considerably more complex than this, which is why the topic is discussed thoroughly in the next chapter. Another feature often associated with real-time Unix systems is real-time signals, which is covered in chapter 6. In addition, memory locking and especially timers are useful for many real-time applications, which is why these are briefly considered at the end of this chapter.

However, none of these features matter if the kernel can not guarantee short dispatch latencies to processes. This means the time it takes to continue the execution of a process having the highest execution priority in the system, after an event has occurred that demands that execution. Typically this event is an interrupt that should trigger the execution of a process. Process dispatch latency consists of the time it takes to handle the interrupt, the time it takes for the kernel to rearrange its state so that the process execution can proceed, and performing a context switch to that process context. These latency delays are unimportant for non-real-time processes, but if they are unbounded or otherwise high, the system can not guarantee good real-time performance for applications that need it.

Process dispatch latencies are a tricky issue, because they are intimately related to kernel design. It has been said that Unix can not be made to do hard real-time tasks without re-designing it from the ground up. While this may not be entirely true, the claim deserves serious consideration, and hopefully the following sections explain why.

4.1 Traditional Unix kernel design

Process scheduling in Unix has always been preemptive, meaning that the scheduler will always choose to run the highest priority process, interrupting the currently running one. But the traditional Unix kernel was designed so that switching to another process can not always occur immediately. This design was adopted by the BSD strain of Unix, and was also present in 4.4BSD. Marshall Kirk McKusick et al. explain it:

> Because 4.4BSD does not preempt processes executing in kernel mode, the worst-case real-time response to events is defined by the longest path through the top half of the kernel. Since the system guarantees no upper bounds on the duration of a system call, 4.4BSD is decidedly not a real-time system.[19, p. 97]

What this means is that processes may make a system call, which will all take some non-determinate time to execute. The execution can not switch to userspace processes, until the system call has finished, or it voluntarily sleeps, waiting for some event to occur. It does not matter if the quantum for process execution has expired — the system call can not be interrupted. Thus, a kind of priority inversion occurs: kernel is performing a service to a process irrespective of process priority and its quantum, thus delaying the execution of higher priority processes, or any other process waiting for its turn.

This reason for this model is that it offers a simple way to ensure that kernel data structures are always in a stable state, when a switch to other process context occurs. If preemption would occur when the kernel is manipulating some data structure, then some other process might want to manipulate the very same data, and chaos would ensue. This makes kernel programming easier, but it has serious consequences to real-time, because a real-time task needs to wait for some less urgent kernel processing to finish. Thus, in this design model, process dispatch latencies may be quite high, depending on the kind of processing that occurs in kernel.

It is interesting that if a machine had only one process using the kernel, this system model could mostly guarantee real-time execution to this process. However, if any other process was making kernel calls, some priority inversion would likely occur. In typical situations, however, there are always several processes blocking in kernel at any time, potentially interrupting the execution of some real-time processes. If the system supported assigning a high static priority to some process, this process would have real-time guarantees only if it totally monopolized the CPU(s). Giving

CPU time to any other processes would mean that these processes could execute system calls, leading to a risk of priority inversion.

The above system behaviour is slightly complicated by the fact, that system calls will actually be preempted by hardware interrupts, but once interrupts are handled, system call execution will (and must) continue. System calls may temporarily delay interrupt handling by using *spl(9)*[25] kernel functions, and sometimes need this to ensure that interrupt handlers can not access some kernel data that is being manipulated by the system call. Interrupt handlers, on the other hand, can be interrupted only by higher level interrupts. This means there may be several interrupt handlers active at the same time, and depending on the tasks they need to do, finishing them all may delay all other kernel level processing.

In practice, interrupt handlers are not a significant source of latency, because they need to finish quickly in order to not interfere with other interrupts and kernel tasks. They are likely to become an issue only when the system needs to achieve latencies below a millisecond or so. However, the non-preemptive nature of the kernel is a serious source of high process dispatch latency, meaning latencies on the range of about a few hundred milliseconds.

Of the current 4.4BSD derivatives, the NetBSD and OpenBSD systems still follow this model, meaning that they can not be considered real-time systems. This does not mean that it is not possible to achieve a totally satisfactory performance for some soft real-time tasks, such as watching videos or playing games. But because this real-time is "fragile", the users must be careful to not disrupt the real-time tasks by running other tasks. This can be rather difficult in applications where many processes are necessarily involved, such as in the case of several audio clients connected to an audio server.

4.2 Semi-preemptibility

What could be changed in traditional Unix design to achieve a guarantee of a short process dispatch latency? Interrupting a system call at any time will not work, unless there is some other level of protection for the data structures that kernel is manipulating during system call execution. In traditional design, some locks are used to protect things such as file I/O queues, but the vast majority of kernel data is not protected by any kind of explicit locking mechanism. The non-interruptible nature of system calls functions as a simple and implicit locking mechanism for all kernel data.

In a kernel design without explicit locks, one method of improving process

dispatch latency is to separately deal with each case where high delays related to kernel processing may occur. This might be called as "semi-preemptible" kernel design, and it was used in System V Release 4 kernel:

> The SVR4 kernel defines several *preemption points*. These are places in the kernel code where all data structures are in a stable state, and the kernel is about to embark on a lengthy computation. When such a preemption point is reached, the kernel checks a flag called `kprunrun`. If set, it indicates that a real-time process is ready to run, and the kernel preempts the current process.[43, p. 123]

In SVR4, system calls are interrupted only if a real-time process needs to be run, and execution switches to that process. Because system calls are interrupted only when kernel structures are consistent, problems of data corruption do not occur. After reconsidering some implementation details, this technique should not limited to switching to real-time processes only, but be adequate for interrupting processes whose time quantum has expired.

Semi-preempting a kernel can also provide a very good process dispatch latency, because basically all the kernel sections that may cause high latencies can be dealt separately, and changed so that their execution can be safely interrupted. The process of improving kernel latency then becomes a question of identifying the problematic sections of kernel code, and modifying them so that they use conditional scheduling. In-between performing the task that any particular section of the code is doing, at some intervals the code should check if some other process should be run instead, and then make a switch to that.

The problem with this approach is that it is a rather messy solution to the latency problem. In this model, real-time is not an inherent characteristic of the kernel, but rather a feature that has been "bolted in", by re-writing problematic sections. This also means that finding out and fixing the problematic sections is a time-consuming task, and will never be properly finished, as it needs to track all other developments in the kernel.

System V is not the only Unix system where this concept has been tried out. There have been some "low-latency patches" for Linux versions 2.2 and 2.4 written by Ingo Molnar and Andrew Morton, that are based on this concept. The effectiveness of these patches were tested by a Red Hat Linux employee Clark Williams, and he concludes that: "the low-latency patches really make a difference in the latency behaviour of the kernel. The maximum observed latency value is 1.3ms, the mean latency value is 54.2μs, and 96.66% of the samples occurred below 100μs." [45] This

was in contrast to maximum latency of 232.6*ms* with vanilla 2.4.17 Linux kernel.

This model has not been tried out with any of the major BSD kernels, and possibly never will. It has no apparent benefit for server systems, and instead unnecessary complicates many sections of kernel. The low-latency patches for Linux were never merged into the official version — instead, Linux has chosen another design path to deal with latency issues, that relates to their symmetric multiprocessor design.

4.3 Preemption in kernels with fine-grained locking

The traditional kernel design has another serious drawback: that the implicit locking model works is based on an assumption that the system will be run on uniprocessor machines only. If the system were run on a multiprocessor machine, then, because of the lack of an explicit lock, more than one CPU could be simultaneously running kernel code. The result is the same as when preempting a system call, because nothing can ensure the consistency of kernel data.

The simplest way to adjust the traditional design to multiprocessor systems is to put an explicit lock that controls access to the whole kernel. One way to do this is to adopt the master-slave SMP model:

> A simple technique for doing this on an MP system is to require that all kernel execution occur on one physical processor, referred to as the *master*. All other processors in the system, called *slaves*, may execute user code only. A process executing in user mode may execute on any processor in the system.[36, p. 176]

A variant of this model is a "giant lock" model, where any CPU can execute kernel code, but only if it first grabs a lock that functions as an access control mechanism to the whole kernel. Many historical Unix systems adopted this model for SMP, and among the current BSD variants it is used by NetBSD and OpenBSD systems.

In both models an operating system becomes multiprocessor ready, and in such a way that it demands no changes to most kernel subsystems such as the audio system. On the other hand, it is clear that the design can waste some CPU time, and the unefficiency becomes more drastic with a higher number of CPUs. If one process is running in kernel mode, and other processes need to access the kernel, they can not run concurrently on other CPUs. Instead the CPUs have to sit idle until the process that reserves the kernel has finished its system call.

Considering multiprocessor machines is interesting for our purposes, because of the way they relate to real-time kernel design. But it should possibly be made clear

that they do not, by itself, offer a simple way to get a good real-time performance. At first sight, it might seem that the system could achieve latency guarantees simply by dedicating one or more processors for real-time work. Because nothing else would be run on those processors, the real-time processes could be run immediately, and the process dispatch latency would thus be effectively zero. The emergence of dual-core CPUs (containing two processors in one chip), and especially SMT (hardware simulation of several processors within one processor) would make this scheme quite lucrative.

Unfortunately this view is too simplistic. SMP and SMT will not significantly help real-time performance, because the issue of high process dispatch latency is caused by the fact that the kernel data structures are inconsistent. These structures exist in the same physical memory, and are thus shared by the processors. This means that a process executing in one CPU can delay the execution of a process executing in another CPU, in case they both need to access the kernel.

While multiple processors can not help with high process dispatch latency, these ideas are still linked together. The main cause for process dispatch latency is the non-preemptive kernel design. With SMP, however, it becomes possible to make the kernel preemptive. Schimmel explains this:

> Since short-term mutual exclusion has now been made explicit, the old UP approach of not preempting processes executing in kernel mode can be relaxed. Anytime a process is not holding a spin lock and has not blocked any interrupts, then it is not executing in a critical region and can be preempted. When a kernel process is preempted under these circumstances, it is guaranteed that it is only accessing private data. This observation is useful on real-time systems, for example, as it can reduce the dispatch latency when a higher priority process becomes runnable.[36, p. 209]

To put more simply, with SMP locks the system calls can be interrupted and execution can switch to another process context, because the locks protect access to the kernel. Processes can not enter the kernel, until the lock is released by the process that holds it. Thus, when some process tries to execute a system call, it will block, and execution has to switch back to the process holding the lock. This process then needs to finish the system call, and then the other process can continue, by grabbing the giant kernel lock, and starting its execution in kernel mode.

Technically, we can see that by creating an explicit kernel lock, it becomes possible to make the kernel preemptive, and low process dispatch latency can be achieved. However, whether or not this would really be useful for real-time process is debat-

able. The real-time process may get to run almost as soon as it needs to, but in case it needs to access the kernel, it will have to wait for the process holding the giant lock anyway. This does not mean that it can not have any effect outside the machine. For example, reading and writing to an audio device normally is done by using system calls, but in case the kernel audio buffers are mapped to process virtual memory, the real-time process can affect audio output without accessing the kernel.

For this reason, the Unix kernels that use the giant lock model for SMP, do not typically enable kernel preemption. Implementing kernel preemption in most kernels is probably not very difficult, but the benefit is also limited. But the situation would be different if, instead of using a one giant lock protecting the kernel, there were several ones protecting smaller subsystems of the kernel.

This is in fact the design path that has been taken by most commercial Unix implementations, such as Solaris, as well as Linux and FreeBSD. In these kernels, some kind of giant lock may exists for legacy kernel code, but generally system calls do not need to grab it if the kernel code only needs to access subsystems protected by other locks.

To accomplish this, FreeBSD has introduced some lock types, or *synchronization primitives*, that were unnecessary in 4.4BSD kernel. *Spin mutexes* and *sleep mutexes* provide short term exclusion for data objects. The difference between these is that requesting a sleep mutex will cause a context switch if the requesting process can not get the mutex immediately, whereas requesting a spin mutex will cause the CPU to "spin" on requesting the lock until the request can be satisfied. Whether or not to use either a spin or a sleep mutex depends mostly on the kernel context. The choices made may have some direct effects on real-time performance, as noted by John Baldwin: "spin mutexes block interrupts while they are held. Thus, holding spin mutexes can increase interrupt latency and should be avoided when possible."[1]

FreeBSD has also introduced condition variables, which provide a mechanism to block a thread waiting for some condition to occur. Shared/exclusive locks can be used to let threads share some resource, but allow only one thread to have exclusive access to it. Basically data locked using a shared lock should only be read, and if data needs to be written, a thread needs to request an exclusive lock.

The differences between particular synchronization primitives and their usage patterns are not very interesting for our purposes. But it is interesting to note that these synchronization primitives can be useful for uniprocessor machines as well, if good real-time performance was a serious goal for operating systems targeting these computers. It seems that many Unix variants have only introduced them, because not using them results in serious shortcomings in the throughput of multiprocessor

machines.

While commonly optimizing for both throughput and latency are somewhat contradictory goals, in this case it is curiously so that the introduction of these primitives also has beneficial side effects to system's real-time performance. While the introduction of this kind of "fine-grained kernel locking" lessens the system throughput for uniprocessor machines, it will provide a better process dispatch latency, while the throughput for multiprocessor systems still increases.

On the other hand, the tradeoff between throughput and latency still exists, it is only different with different classes of machines. The efficiency of fine-grained locking design depends on locking granularity. A kernel with a small number of locks (protecting large subsystems) will run inefficiently on machines with many CPUs. Increasing locking granularity by creating more locks (that each protects smaller amounts of kernel data) will give better throughput for systems with a higher number of CPUS, and better latency for every class of machines. But this can occur only up to a certain point, as Uresh Vahalia explains:

> Clearly, that is not the ideal solution either. The locks would consume a large amount of memory, performance would suffer due to the overhead of constantly acquiring and releasing locks, and the chances of deadlock would increase because it is difficult to enforce a locking order for such a large number of objects"[43, p. 212]

Thus, given any machine with a particular number of CPUs, both its throughput and latency can be increased by up to a certain point by more fine-grained locking. After that, the throughput will suffer and process dispatch latencies will be better. This turning point will depend mostly on the number of CPUs.

We can see that a desire to scale Unix to multiprocessor systems can provide latency benefits for uniprocessor systems. But this can happen only if the kernel is made preemptive. Once all kernel data structures are protected by some locks, preemption needs only fairly small changes to the kernel. These changes typically mean testing if process rescheduling is needed, in situations such as releasing spin mutexes and returning from interrupt handlers, because these occur frequently and may cause sleeping processes to become runnable.

Interestingly, experiments done by Clark Williams with Linux kernel suggest that preemption does not offer as good latency behaviour as semi-preemptible strategy of Linux low latency patches: "while both patches reduce latency in the kernel, the low-latency patches are the biggest single win." [45] On the other hand, the difference is mostly negligible for the purposes of real-time audio, because Linux

preemption patches could achieve latencies below one millisecond for 99.99% of time. The maintainability in kernel preemption is also considerably better than in the semi-preemptible approach.

Would there be some way to analyze the connections between some particular locking pattern and process dispatch latency? In a general case, it can be said that a real-time process can always proceed unless there is another process runnable with a higher priority, and the kernel subsystems it needs are not locked by any other process. With smaller sized locks, it is less likely that a process has to stop waiting for some other process to release the resource it needs. We could consider audio servers. These need access to audio subsystem, as well as some parts of the IPC and networking layers, and possibly the filesystems. For these, the ideal locking granularity should be moderate or high, but other subsystems hardly matter, as long as they do not use the giant lock.

Nevertheless, situations where a high thread requests a lock that is held by a lower priority thread may still occur, irrespective of the locking granularity. What should the kernel do in these situations? Unless this situation is specifically handled by the kernel, the wrong thing happens: the high priority thread needs to wait for the low priority thread until it releases the lock. What makes this worse is that

> These periods of time are potentially very long — in fact, they are unbounded, since the amount of time a high priority thread must wait for a *mutex* to become unlocked may depend not only on the duration of some critical sections, but on the duration of the complete execution of some threads. [16]

This problem is known as *priority inversion*. There are various solutions for this problem, such as *priority ceiling protocol* and *priority inheritance*, that are commonly used in real-time operating systems. At least priority inheritance is possible to implement in Unix kernel without very serious difficulties: Solaris kernel had it as a part of their multiprocessor designs over ten years ago.

Even though the use of fine-grained locking can lead to better real-time performance, and this design is proven to work fine in many Unix variants, not everyone is totally content with it. The main problem is that locking brings some additional complexity to the kernel, which increases risk of race conditions and deadlocks. In addition, the concurrency model is probably not as easy to reason about as would be the case with with co-operating threads. Thus, we want to take a look at an alternative model in making Unix multiprocessor ready, and consider the effect that it could have on real-time performance.

4.4 Message passing kernels with LWKTs

A necessary prerequisite for kernel preemption is that all kernel data structures must be protected by locks, so that interrupting kernel processing can be made safely. A typical development path for Unix kernels has been to increase the number of those locks, covering smaller parts of kernel subsystems. This leads to better performance, but typically only worsens kernel stability and security, which is why systems such as NetBSD and OpenBSD have not adopted the fine-grained locking model.

But the issue can be considered from another point of view as well. Maybe the real problem is not the lack of kernel locks, but perhaps there are too many global kernel data structures that are directly accessible by kernel subsystems. Instead of writing more locks to protect data structures, those data structures could be hidden behind some abstraction that should be used to get access to them. In this scenario, the multiprocessor and latency performance problems could be addressed by reducing the *need* for explicit kernel locks, instead of by writing *more* of them.

It should be feasible to use this idea in kernels with fine-grained locking, but it is possibly simpler to go directly from giant kernel lock to this SMP model. Among BSD variants, this is in fact one of the main ideas behind DragonFlyBSD. Similar philosophy can be observed in microkernel designs, where the kernel only performs some very basic operations, that include message passing between other operating system components. There is no need for a huge number of locks, because subsystems can run concurrently in their own address spaces, and can coordinate their actions by message passing provided by the kernel.

The goal of the DragonFlyBSD is not to become a microkernel, but it clearly borrows some ideas from them. In contrast to microkernel concepts, the kernel subsystems in DragonFlyBSD should in most cases share an address space, and use their own Light Weight Kernel Thread (LWKT) infrastructure instead of moving functionality into userspace. This can have clear performance benefits, because there is no need to switch between execution contexts when switching between kernel threads.

The structure of LWKTs differs from commonly used kernel level threads in that they have no process context involved, and thus creating and switching between them should be faster. The communication with LWKTs happens through a message passing API associated with them. There is also a plan export the API to userspace, so that parts of the kernel could optionally be moved to userspace, similarly as in microkernels:

> We want to go further along these lines and decouple some of the depen-

> dencies in the I/O subsystem on the current address space, generalizing them to work with virtual memory objects instead. The end goal is to be able to run device drivers and large subsystems such as the VFS layer in userland.[15]

Thus, the DragonFlyBSD project plans to partition the FreeBSD kernel into subsystems that run on LWKTs, and are mostly isolated from each other. This makes it possible to run several instances of each subsystem on its own thread, these threads being mostly bound to some particular CPU. This way, DragonFlyBSD can achieve the SMP goal of running kernel subsystems concurrently in different CPUs, and makes this more explicit than is the case in fine-grained locking model. To make this work, subsystems should be mostly decoupled from the current kernel address space, and instead put to using message passing to co-operate with other kernel subsystems. Furthermore, because the communication between subsystems happens through the message passing API, it is possible to export parts of the kernel functionality to userspace. This carries with it the benefits of better reliability and better debuggability of system components.

The DragonFlyBSD model seems to require somewhat more radical changes to kernel than is the case in FreeBSD model of fine-grained locking. On the other hand, it does seem like a clean and coherent design, that may prove itself superior to the convential Unix SMP model of threading the kernel with locks. But the real interesting question for our purposes is: how does this model affect the real-time performance of the system?

The answer has to be that most of what is true with kernel preemption in the fine-grained locking model is true in the DragonFlyBSD model as well. Consider the case where a programs makes a system call. A system call executes some kernel code, and it needs to use some kernel subsystem (or possibly many). To do this, it sends a message to the subsystem, subsystem does some work on its behalf, it receives an answer, and returns to userspace. The messaging may occur asynchronously so that system call may do some other things before receiving the answer, but this complication is not relevant for our purposes.

The condition for preemption is that a system call should not do anything where it can not be cleanly preempted. When it sending and receiving messages with a subsystem running on some LWKT, preemption is possible. This is because the subsystem has its own thread, and message passing provides an access control mechanism to that subsystem, so that kernel data structures can be kept stable. And when a system call is not communicating with a subsystem on LWKT, it should not change any kernel data that has no access controls. If that area of kernel data

needs to be frequently modified, that should be candidate to be hidden behind some LWKT that handles the task of modifying that data. Alternatively, for structures that change only infrequently, a giant kernel lock could be used.

Doing this is in fact necessary if DragonFlyBSD model wants to reach both SMP safety and good performance. This makes kernel preemption a more useful feature to implement, as the ability to get good real-time responses through preemption is simply a natural byproduct of these goals. A part of the real-time response time is determined by process dispatch latency, which should always be very short with kernel preemption. The other part of response time is determined by the time it takes to execute the real-time task. The kernel usually plays a part in this time, and in fine-grained locking model this depends mostly on locking granularity and handling of priority inversion. In most cases, with kernels using fine-grained locking, the latency added by kernel is usually very low, on the order of two milliseconds. As preemption itself does not tell much about real-time performance, it should be interesting to consider if DragonFlyBSD model can also be as good as fine-grained locking in this front as well.

In DragonFlyBSD, the concept that mostly corresponds to locking granularity is subsystem size. The benefits of a higher locking granularity are that it is less likely for a real-time process to request a lock that is already reserved, and as that happens, the time it takes for the reserving process to release that lock will likewise be shorter. Hence the system can deliver a better real-time response. Similarly, if the subsystems running on LWKTs are smaller (they are more fine-grained, covering a smaller set of kernel functionality), the system should perform better regarding real-time. This is because as there will be more LWKTs running, they will be doing less work, and thus should be faster to pick up any new tasks they need to accomplish.

Priority inversion can be a problem with LWKTs as well. The message passing system works so that processes will queue their requests to LWKTs. If a LWKT would simply process these requests in the order that they came, the requests of low priority processes might be handled before the requests of high priority processes. Thus, priority inversion would occur. In fact it could occur if a LWKT were in the middle of processing a request, and a higher priority request arrived on its message queue. The proper way to deal with this would be to prioritize the message queues according to the priorities of the requesting processes. It seems that in principle this is not very difficult to accomplish, and thus the DragonFlyBSD model can also provide a solution to the priority inversion problem.

It is difficult to say how these two models would compare in real world situations. Benchmarking is not yet possible, because current DragonFlyBSD does not yet

implement kernel preemption. At least in SMP efficiency the models are not likely to be equivalent, because subsystems using LWKTs are more strongly localized to specific processors. With fine-grained locking, there is a tradeoff between locking overhead and SMP efficiency, whereas in DragonFlyBSD model the overhead should always the same independent of the number of CPUs. At least the DragonFlyBSD developers believe their model can provide better scalability:

> Through judicious application of partitioning and replication along with lock-free synchronization techniques, we believe we can achieve greater scalability as the number of processors increase than a system that has locking overhead and contention for shared resources. [15]

It should probably be noted that the model behind DragonFlyBSD, and microkernel systems in a more wider sense, is not in any way in contradiction with real-time goals. Considering that microkernels are commonly associated with inefficiency, this might seem somewhat surprising, but regarding real-time, possibly the complete opposite is true. There has been research in real-time microkernel systems for a long time now, for example a paper[31] by Seong Rak Rim and Yoo Kun Cho outline a design for a simple real-time microkernel system. In their paper, they came to conclusion that most real-time critical things in microkernel designs are kernel preemption and efficient mechanisms for message passing between processes.

This question of efficiency is in fact possibly the most commonly raised issue regarding microkernel architectures. However, for our purposes, possible inefficiencies do not tell much about real-time performance, because with real-time, the major consideration is not bandwidth, but response time. Furthermore, some of the perceived inefficiency of microkernels may be due to designs that are no longer current and relevant. Even though microkernel systems have not seen as widespread distribution and use as was predicted a decade or two ago, real-time microkernel systems have actually been proven to work well in real-world situations. The most relevant example is QNX, which is based on a microkernel architecture and has seen widespread use in almost every field where fault-tolerant real-time systems are essential. Another good example is BeOS, which is also microkernel based and known to provide good response time for real-time applications.

An interesting feature of message passing system model is that it allows an easy extension of kernel functionality by userspace applications. We have already considered the problem relevant to audio systems: how should the audio functionality be split between kernel and a userspace application (an audio server)? But if the audio system were converted to use the kernel message passing system, it could be

run either as a specially-privileged application or in kernelspace. If it was run as a process, this would make it more reasonable to turn it into a more flexible subsystem, with mixing, processing and inter-application routing. Bugs in audio system would not lead to a system crash, but it could simply be restarted by the kernel.

This idea is not in fact limited to message passing kernels. Monolithic systems could also move part of the audio functionality to userspace by using a server that implements parts of the kernel audio system, as the kernel would contain some simple API that would merely provide a narrow, unrestricted access to audio hardware. Applications would be forced to connect to the server, instead of using the kernel directly. However, in DragonFlyBSD plans, the message passing interface could provide a generic interface to kernel subsystems from userspace, solving the problem of split functionality for every system service that might face it.

What is also interesting in this is that exporting message passing to userspace can allow easier kernel customization for application requirements. At least the DragonFlyBSD developers believe, that there are "huge performance benefits associated with closer integration between user-land and OS facilities"[15], which is a claim that can be supported by various research that has been done on exokernels. The enabling technology in DragonFlyBSD for this integration is their message passing model. On the other hand, it is rather difficult to say whether or not this could provide any meaningful performance benefits for audio software, and if yes, what specific audio system designs would enable this. Nevertheless, it is an interesting question, and some experimentation might be necessary to gain a better understanding of this issue.

One benefit that message passing monolithic kernels can have over microkernel models is efficiency, as microkernels have more context switching overhead associated with them. On the other hand, quantitative and qualitative developments in CPUs can possibly mean that this overhead will not be meaningful in the long term. Microkernels can also provide an interesting way to bring real-time performance to Unix with only small changes to monolithic kernel itself.

4.5 Monolithic kernels on microkernels

One possible technique to turn monolithic non-real-time Unix kernels into real-time systems is to run them as an application on top of a real-time microkernel. This model can provide real-time guarantees to applications, by using the microkernel to interrupt monolithic kernel, and by scheduling real-time processes from the microkernel. This model has been successfully used with Linux, FreeBSD and NetBSD

kernels by a company named FSMLabs, in its RTLinux and RTCoreBSD offerings. They can provide hard real-time guarantees that are commonly needed in industrial settings, and can be a good fit in cases where real-time control of some devices needs to be used in conjunction with some other applications.

The basic problem with this model is that real-time processes can not commonly access services provided by the monolithic kernel, since it does not provide real-time guarantees. Instead, the applications need to use a special purpose API for communication with the RT microkernel. FSMLabs' software also provide interfaces that allow real-time processes to communicate with non-real-time processes, and let them even be run as part of the address spaces of non-real-time processes.

The systems offered by FSMLabs do not, in fact, utilize a pure microkernel architecture. Instead,

> unlike the famously slow microkernel or, even worse, virtual machine, RTCore allows the platform operating system to go directly to hardware for non-real-time devices, to manage its own memory, and to schedule its own processes free from interference or costly layers of emulation.[9]

Providing this direct access should thus sidestep the inefficiency that has commonly been associated with microkernel architectures. It might be interesting to know how serious this inefficiency could be. Jochen Liedtke et al. have written a paper addressing this issue, and tested the performance of a combination of Linux kernel and an L4 microkernel. Linux was modified to be run as a single application on L4, that implements its own IPC and address space control mechanisms. Their results indicate that the performance penalty associated with L4 microkernel was about five to ten percent for most applications. They arrived to the following conclusion:

> The goal of this work has been to understand whether the L4 μ-kernel can provide a basis on which specialized applications, including those with real-time requirements, can be built such that they run along with normal operating systems and their applications on a single machine. The results described in this paper encourage us to pursue that line of development. [18]

Thus, at least with some microkernel designs, the inefficiency should not be a very serious issue. The main practical difficulty for audio systems is that audio subsystems should be written to use the microkernel, instead of the monolithic kernel services, if they want to gain the real-time guarantees. This also means rewriting

device drivers to run on the microkernel instead. Thus, it seems that this model does not provide an easy way to any real benefits for most audio work.

However, because microkernel systems can clearly provide fine real-time performance, reliability and flexibility, this model could be useful as a starting point for an evolution of a monolithic system to a real microkernel based one. This path of system evolution can be interesting to contrast with the one in DragonFlyBSD, where a monolithic system evolves gradually into a more microkernel-like system, instead of being put first to run on such one.

4.6 Real-Time features of operating systems

Kernel design is quite certainly the most critical aspect that affects a system's real-time behaviour. Yet real-time applications may need to expect more than simply quick process dispatch latency and wait-free execution of system calls from an operating system. These features commonly include memory locking and access to accurate timing information, so we will briefly introduce these here.

4.6.1 Memory locking

Memory locking is in fact a mechanism that is needed to guarantee good process dispatch latency for processes. It means locking (or "wiring") an application's virtual memory pages to physical memory, so that any memory pages will not be temporarily swapped to disk. This is generally needed by real-time applications, because otherwise, if some part of process address space needed to be fetched from disk before it could be accessed, process execution would be face unexpected delays. These delays would be determined by the access and read times of swapping media, which may be affected by other tasks.

By using the *mlockall(2)*[26] system call, these delays can be avoided by disallowing the swapping of a process memory space. The disadvantage of this is that wiring address space in this way needs to be an operation that is limited to superuser, because otherwise any user could consume most or all of physical memory. In addition, memory locking may not always be possible, due to a lack of free physical memory.

4.6.2 Clocks and timers

Real-Time implies that application execution needs to be synchronized with time — some actual temporal events, clocks or equivalent. This raises a practical problem: how should this connection be established?

A general interface to get timing information in Unix systems is the *gettimeofday(2)*[3] system call, that can be used to get the current time. This timing information is accurate to the system clock level, and corresponds to one system clock tick, which is normally ten milliseconds in BSD systems. Because continuously polling timing information is rather inefficient, Unix systems also provide an interval timer that can be accessed with *getitimer(2)*[3] and *setitimer(2)*[3] calls. Using this interface, an application can request a signal to be delivered to it after some time interval, which can be used to synchronize application with system clock. These times can measure real or process virtual time, where these correspond to `SIGARLM` and `SIGVTALRM` signals.

The problem of the inaccuracy of the system clock is present with interval timer interfaces as well, because short time intervals will still be rounded up to system clock tick resolution. It is possible to try to work around this limitation with *nanosleep(2)*[26] call, that allows an application to suspend its execution for a shorter time, but this can obviously be a problematic approach.

On the other hand, relying on these interfaces may not be necessary at all for many applications. If an application's I/O activity is in some way connected to interrupts of some hardware device, then, as those interrupts are synchronized with real-world time, they can also be used to synchronize process execution. For example, in the case of audio applications, if an application reads audio data from an audio device, the input data blocks will be received in some specific intervals. This way, the audio device can function as a clock controlling the execution of an audio application. This means that even in a distributed audio system, the processes may not need to resort to querying system time or using interval timers in order to function correctly.

5 Real-Time scheduling

As was explained in chapter 2, scheduling means the process of apportioning CPU resources to kernel tasks and user applications. The traditional Unix scheduling algorithm does not pay any special attention to tasks that may have real-time requirements. This is mostly because it is simpler and more fair to treat all processes equally, and the concept of a real-time task implies that some tasks have special requirements over others. In real-time scheduling, however, there must be mechanisms that can be used to guarantee some amount of CPU resources to processes that need them.

5.1 Traditional Unix scheduling and real-time applications

In traditional Unix scheduling, the scheduler is mostly oblivious to different process resource requirements. The algorithm mostly tries to offer good latency to processes that do not demand much processor time, thus favoring most interactive applications. This has an important quality that the scheduler behaviour is self-tuning, adapting to the CPU usage patterns of applications. Typically, the user nor the applications do not need to inform the system how they should be treated, and the scheduler still provides satisfactory performance for most workloads.

On the other hand, in the traditional scheduling algorithm it is possible to adjust the relative priorities of processes. Applications can do this via *setpriority(2)*[3] system call, that changes the *nice*-value of a process, affecting priority calculation. A user can adjust the same value with `nice` and `renice` utilities. Even though the main purpose of this mechanism is specifying the relative urgency of various tasks, it looks like this can be a useful mechanism for real-time tasks as well. This is because with most applications that have real-time requirements, it is necessary to inform the system scheduler about their special priorities anyway, and the nice-mechanism offers one simply way to do this.

Thus, it is interesting to consider how well the scheduling algorithm used in 4.4BSD could support real-time tasks, if process priorities were adjusted properly through the nice-mechanism. As explained in section 2.3, the scheduler uses mainly two equations to calculate user-level process priorities. The equation 2.1 uses nice-value and an estimation of CPU time to calculate process priority. The decay equation

(2.2 on page 12) is used to adjust the estimation of CPU time used by the process. The decay equation is used only once a second, the main one tunes process priorities in shorter cycles.

The algorithm could provide proper service to a real-time process, if using the highest possible priority adjustment would guarantee that a process would always be run preferentially to other processes not having a similar priority adjustment. The highest priority corresponds to the lowest nice-value, -20. Here, we ignore issues such as kernel design and priority inversion, that can interfere with process scheduling but are not actual faults in the scheduler itself.

A real-time process will have a monopoly on CPU time (on user-level) if the user-level priority of the process is always below or equal to PUSER, which means that in priority calculation it should always be true that

$$p\_usrpri = PUSER + \frac{p\_estcpu}{4} + 2 * p\_nice \leq PUSER \tag{5.1}$$

where PUSER has value 50 (the low limit of user-level priorities). The *p_estcpu* value depends on process execution characteristics, and the decay algorithm. If we assume that the process uses all the available CPU time, the *p_estcpu* will be increased by one at every clock tick. As the decay algorithm adjusts the *p_estcpu* value at every second, it will be incremented one hundred times before decay equation will adjust it to some lower value. Thus, to find the range of possible *p_estcpu* values that a constantly running process may have, we calculate the minimum for *p_estcpu* with

$$min(p\_estcpu) = \frac{2 * load}{2 * load + 1} max(p\_estcpu) + p\_nice \tag{5.2}$$

Here, $max(p\_estcpu)$ must be equal to $min(p\_espcpu) + 100$. After solving the equation, we get

$$min(p\_estcpu) = 200 * load + (2 * load + 1) * p\_nice \tag{5.3}$$

Since the *p_nice* value is under user control, we can adjust it to -20, thus giving the highest priority to our real-time process. *load* means load average, that approximates the number of processes in scheduler run queue. For purposes of this proof, we can basically choose any value we like for it, so we give it a value of one, corresponding to the constantly running real-time process. From this we obtain that

$$min(p\_estcpu) = 200 * 1 + (2 * 1 + 1) * (-20) = 140 \tag{5.4}$$

which means that in this case *p_estcpu* value will be between 140 and 240. Since the value 240 represents the worst case, where the user-level priority of a process gets

its maximum value (and thus lowest priority), we use that to check if it satisfies our criteria for a real-time process:

$$p\_usrpri = PUSER + \frac{240}{4} + 2 * (-20) = PUSER + 20 \leq PUSER \tag{5.5}$$

As this is obviously false, this means that a process with a nice-value of -20, in a system with load average of one, can not constantly run and hold the highest user-level priority. This does not mean that a process running only in short bursts can not keep the highest possible priority. Nevertheless, it is clear that the nice-mechanism is not adequate to elevate a priority of even a single process over other ones — thus, the scheduler is strictly time-sharing.

It, however, turns out that none of the current BSD variants use the original scheduler as it is, but have made a few small modifications to it.[27] First of all, the algorithm used in both NetBSD and OpenBSD have left out the nice adjustment when decaying CPU usage estimates. Furthermore, the equation for user-level priority calculation has been modified to this:

$$p\_usrpri = PUSER + p\_estcpu + 2 * p\_nice \tag{5.6}$$

These changes alone would not make any considerable difference to real-time operation. But with these changes, the value for *p_estcpu* was also limited to the maximum of 36.[28, `/sys/sys/sched.h`] This has two interesting effects. It means, that processes with nice-priority of +20 will always get a lower priority than processes with normal priority. The other interesting effect is that processes that are given nice-priority of -20 will get their user-level priority below or equal to PUSER. Thus,

$$p\_usrpri = PUSER + 36 + 2 * (-20) = PUSER - 4 \leq PUSER \tag{5.7}$$

and these processes have an effective real-time priority, while the actual scheduler algorithm used is primarily time-sharing.

However, this situation is further complicated by the fact that kernel level tasks always have higher priorities than user processes in traditional kernel design. Hardware interrupts can trigger system activity that overrides the needs of a real-time process. Furthermore, if there are some processes blocking in kernel, these may get to execute preferentially, even if their user-level priorities were lower than those of other processes. Thus, a more complete solution to real-time scheduling should involve separating the real-time priorities into their own class, or treating user priorities in some different way.

It should also be noted that if this mechanism is used to elevate priorities of several processes, those will be run in round-robin fashion — it can not possibly to

specify that the priority of some process should be higher than some other process at all times. Thus, while the nice-value mechanism is not a perfect solution, it may be good enough to differentiate real-time audio processes and threads in desktop applications, so that they will be scheduled correctly. In any case, many Unix systems have included an ability to assign static priorities to processes, as mandated by the SUS and POSIX standards.

5.2 SUS scheduling policies

Currently, the most important specification determining the Unix system API is probably the Single Unix Specification version 3 (SUSv3)[24], that is mostly based on POSIX, an older standard. In addition to the basic priority control interfaces for time-sharing scheduling, SUSv3 defines some scheduling policies that real-time applications might expect to find in an operating system. The main policy is `SCHED_OTHER`, whose behaviour is implementation defined and commonly corresponds to time-sharing. Applications can use it to say that they do not need any special scheduling policy.

In addition to `SCHED_OTHER`, SUSv3 defines three new policies, known as `SCHED_FIFO`, `SCHED_RR` and `SCHED_SPORADIC`. Processes that are run under the first two of these policies will have static priority assignment, so that using strict priority hierarchies becomes possible: it can be assumed that some processes will always run preferentially over others.

`SCHED_FIFO` means that for the processes that request this policy, the scheduler has a queue that contains the processes in the order they are going to be scheduled. The queue is ordered so that the processes that have spent the longest time without being executed are at the head of the queue. Processes run until they either make a blocking system call, or yield their turn to the next one in the queue. Processes can have at least 32 different scheduling priorities, and thus there may be several scheduling queues for this policy. Because processes have priority over time-sharing processes, this policy is well suitable for real-time applications.

`SCHED_RR` policy is identical to `SCHED_FIFO`, with the exception that processes are moved to the tail of the scheduling queue after some specific time period has elapsed. The point of `SCHED_RR` is to guarantee that all processes at the same priority can receive some processor time. Like `SCHED_FIFO`, the policy should be useful for applications that may need real-time scheduling.

SUSv3 also contains a definition for `SCHED_SPORADIC` scheduling policy, that is suitable for a sporadic server and similar to `SCHED_FIFO`, except that process priorities

may be dynamically adjusted. Priorities are chosen between two different priorities, depending on two parameters, *replenishment period* and *execution capacity*. The policy is designed for applications that need to be activated periodically, for example to handle some events. It should be a good fit for real-time tasks with hard deadlines, where these tasks should be known so well that *replenishment period* and *execution capacity*-parameters can be calculated or estimated. This policy is not as commonly implemented as the previous two, and does not seem particularly useful for tasks such as real-time audio processing.

Both `SCHED_FIFO` and `SCHED_RR` seem to be quite widely implemented in Unix systems, but they are still currently missing from OpenBSD and NetBSD. Adding these policies to the time-sharing scheduler should not be very difficult. On the other hand, considering these systems' kernel design and inability to guarantee good process dispatch latencies, these scheduling policies might not be very useful anyway.

It is an interesting question what additional benefits these policies could bring into a distributed audio system. That is, in the case where audio processing occurs in separate processes, but which are all run in the same computer. Compared to using the nice-mechanism, using `SCHED_FIFO` or `SCHED_RR` policies offer better control of the scheduling preference of the server and clients.

One interesting application of these policies is fitting the scheduling order of applications with the tasks that they are accomplishing. Due to the fact that the system scheduler can not (and should not) know how some process's activities may affect the schedulability of other processes, the scheduling is likely to make some unoptimal scheduling decisions, leading to unnecessary context switching. By adjusting process priorities appropriately, the scheduler can be made to do the right thing, even when it is not aware of the basis of the decisions behind the priority hierarchy.

For example, if within a single processing cycle the server should be run only after all clients are run first (as would be the case in pure parallel topology, explained further on at subsection 5.3.5), the server priority could be set to a lower value than all clients. Thus, if any client has some work to do, it will be run before the server can continue its execution. To make this work, this means that the server needs to ask clients to request the kernel to set their scheduling priorities to some proper values. Nevertheless, these scheduling policies can be used to gain finer control over application execution order.

5.3 Scheduler implementations

As discussed above, the problem of real-time processing can partially be solved with small changes to the traditional Unix scheduling algorithm, or by somewhat larger changes that add handling of additional policies. But there are still other problems in these scheduling algorithms, such as inability to guarantee a portion of CPU resources to some specific process or process group, as well as bad scalability to a very large number of processes. We will now look at some of the scheduler implementations and see how their design might answer these problems, and how that could affect their suitability for real-time tasks.

5.3.1 SVR4 scheduler

Coming up with scheduling algorithms that could support real-time applications, and implementing those algorithms, was part of the research that went into many commercial Unix-variants in the late eighties and early nineties. As a joint effort by AT&T and Sun, Unix System V Release 4 (SVR4) was released in 1989, and it included a complete scheduler redesign, using a modular scheduling framework. The framework was designed to separate scheduling policies from the mechanisms implementing them, thus following a common philosophy in Unix design. Similar design can be seen in many other areas in Unix systems, like in audio driver frameworks, and in the virtual file system interface, where file systems attach themselves to a high-level layer that provides common functions to access files in the file system. This meant that it was possible to extend the scheduler by writing separate scheduling classes (implementing policies) using functions provided by a generic, high-level scheduling layer:

> The scheduler provides a set of class-independent routines that implement common services such as context switching, run queue manipulation, and preemption. It also defines a procedural interface for class-dependent functions such as priority computation and inheritance.[43, p. 122]

SVR4 scheduler has 160 different priorities, with time-sharing, system and real-time priorities. Scheduler always runs the processes with highest priority, unless it is involved with non-preemptible kernel processing, but changing priorities is left to classes that implement specific scheduling policies. Two scheduling classes were included by default in SVR4: the time-sharing class using time-sharing priorities, and real-time class using real-time priorities.

The time-sharing class is similar to traditional Unix scheduling, using round-robin scheduling and priority decay. However, it does not decay process priorities periodically. Instead it uses *event-driven scheduling*, where, "instead of recomputing the priorities of all processes every second, SVR4 changes the priority of a process in response to specific events related to that process."[43, p. 126] The scheduler behaviour can be adjusted by defining how various events should change process priorities. This use of events for priority decay has the benefit that the scheduler is fast and can scale well to a very large number of processes, because the decay does not have to be computed for all processes every second. Processes in the real-time class have fixed priorities, so decay does not apply to them, and their priorities can be changed only through *priocntl()* system call, that is restricted to superuser.

For real-time applications, the SVR4 scheduler design is a clear improvement over traditional scheduling, because it can support the real-time policies in SUSv3. What makes this even more useful is that real-time processes can have priorities that are higher than system priorities. In previous chapter the issues of kernel preemption was considered, but it was still slightly uncertain when the kernel should preempt and run some process, instead of continuing kernel processing. In SVR4 design the decision is clear: kernel should preempt only if processes with real-time priority become runnable. To achieve this, SVR4 kernel uses the semi-preemptible strategy:

> SVR4 uses preemption points to divide lengthy kernel algorithms into smaller, bounded units of work. When a real-time process becomes runnable, the *rt_wakeup()* routine that handles the class-dependent wakeup processing sets the kernel flag `kprunrun`. When the current process (presumably executing kernel code) reaches a preemption point, it checks this flag and initiates a context switch to the waiting real-time process. [43, p. 128]

While SVR4 scheduler provided useful flexibility and features to scheduling, event-driven scheduling has a side-effect that interactive processes might not be as responsive as with the traditional scheduler if they do not generate enough events. It has also been observed, that tuning the system for a diverse application set can be rather difficult, as Vahalia explains:

> A major problem with the SVR4 scheduler is that it is extremely difficult to tune the system properly for a mixed set of applications. [Nieh 93] describes an experiment that ran three different programs — an interactive typing session, a batch job, and a video film display — concurrently. This

> became a difficult proposition since both the typing and the film display required interaction with the X-server. [43, p. 130]

This in fact suggests a problem in scheduling of client-server systems, which will be discussed later in this chapter. But it is also true that a strict separation of real-time and time-sharing priorities may not always be useful in real-world situations. Possibly a better scheduling framework should be able to make finer distinctions than simply "yes or no" regarding real-time requirements, and be readily useable without any specific priority tuning.

5.3.2 Schedulers in BSD systems

As previously covered, current NetBSD and OpenBSD systems include a scheduler almost the same as was in 4.4BSD, with a few real-time friendly modifications. The FreeBSD project has, however, experimented with a scheduler called ULE[34], that has an event-driven algorithm similar to the one in SVR4 scheduler. Since its design does not seem to bring anything new to real-time performance (while it does have support for real-time priorities), it is not covered in more depth here.

The current version of DragonFlyBSD still contains a scheduler mostly using the traditional algorithm with SUSv3 scheduling policies. However, they appear to have some interesting plans for their scheduling framework. In other current BSD variants, kernel level threads still have process context, and scheduling them is a task that belongs to the same scheduler that apportions time also to normal userland processes. However, since the DragonFlyBSD design heavily relies on Light Weight Kernel Threads lacking process context, they have their own scheduler in the kernel. This means and allows the separation of userland process scheduler (or many of them) from the kernel thread scheduler:

> The LWKT thread scheduler is MP-safe and utilizes a fast per-cpu fixed-priority round-robin scheme with a well-defined API for communicating with other processors' schedulers. The traditional BSD multilevel scheduler is implemented on top of the threads scheduler.[15]

This should make it easier to experiment with different scheduling algorithms for userland applications, or to run several schedulers concurrently. Some possible implications of this are covered in the following sections.

5.3.3 Some problems in scheduler implementations

Scheduler design is intimately related to kernel preemption, because it is the scheduler that needs to decide when to preempt the currently running process (possibly running in-kernel), and run some other process instead. An essential question about kernel preemption is when should it occur.

A good answer could be that it should be done whenever the user-level priority of the currently running process is lower than that of some other process. This would lead to better responsiveness in the whole user environment. But then the meaning of system level priorities becomes unclear, since they are considered only with kernel level threads (that lack user-level priorities). Kernel preemption could also be done only to preempt to processes with real-time priorities, as was done in SVR4 scheduler. In any case, real-time priorities should be higher than system level priorities.

5.3.4 SMP and SMT

Another issue that scheduler implementations have to face is how well they handle multiprocessor systems (MP) and also processors that are capable of doing simultaneous multithreading (SMT), that is, able to execute several program threads in parallel. Taking MP systems into special consideration is necessary, because if processes were blindly assigned to any available CPU, this would mean loss of efficiency in CPU cache utilization.

Thus, scheduling in MP systems should use some scheme to ensure CPU affinity, meaning that a process should usually be scheduled on the same CPU where it ran before. This is often implemented by having local scheduling run queues for each processor, and only occasionally moving processes from one run queue to some other one, to ensure proper CPU load balancing. However, CPU affinity is not a good idea for real-time processes: "fixed-priority threads are scheduled from a global run queue, since they should run as soon as possible."[43, p. 143] Separating run queues for time-sharing and real-time processes is also natural if there is a clear separation between the two classes.

SMT processors complicate this matter, because to applications they are essentially CPUs that masquerade as many CPUs. The real benefit that CPUs with SMT can provide is slightly higher CPU bandwidth, because operations can be parallelized, but, as was considered in previous chapter, they do not offer any clear latency benefit.

In fact, SMT considerably complicates the scheduling algorithms, because algo-

rithms need to take into account the existence of "virtual CPUs" to take advantage of symbiosis, which means that "the execution time of a job depends on which other jobs are co-scheduled with it".[14] One consequence of SMT is that the CPU affinity policy presented above is not adequate for handling SMT processors. One algorithm for implementing real-time scheduling on SMT processors has been proposed by Jain et al., where the algorithm "uses global scheduling, exploits symbiosis, gives priority to tasks with high utilization, and uses dynamic resource sharing."[14]

5.3.5 Scheduling of client-server systems

Distributed systems, where individual parts are in the same scheduling domain (that is, under the control of the same scheduler) also raise additional difficulties. The main problem is that the schedulers do not and can not know enough about the dynamics of the distributed system. In such systems, scheduling an application can and will affect the schedulability and scheduling requirements of other applications. If processes work in concert to accomplish some task, an optimal scheduling algorithm should choose to run processes in order, where the task can be completed as quickly as possible. This means it should know the dependencies between subtasks that processes handle. The scheduler should choose to run processes so that when context switches are done, most or all the pre-requisite tasks each process depends on have been finished, and thus context switching between processes can be minimized.

For example, consider two basic client-server topologies, where clients are used for audio stream processing. In one with a "serial" topology, the server passes data through each client in its turn, and no parallel audio processing takes place. In "parallel" topology, the server sends the same data to all clients, letting them process it concurrently, and then receives and combines (mixes) the processed stream together.

Figure 5.1 illustrates both these cases. The picture on the left shows the parallel case, where the audio streams flow concurrently to all clients, and the server receives them to mix them into one output stream. The solid lines in the picture on right side show the audio flows from client to client, corresponding to serial audio processing. Yet, as it simpler to implement such topology by letting the server have full control of the audio flows in the system, the dotted lines show how the server can act as a router for audio flows between clients. The test applications, which will be introduced in chapter 7, have been implemented to use these patterns.

In a serial topology, there is always only one process that can and should be scheduled next. Among these processes, the optimal scheduling should obviously

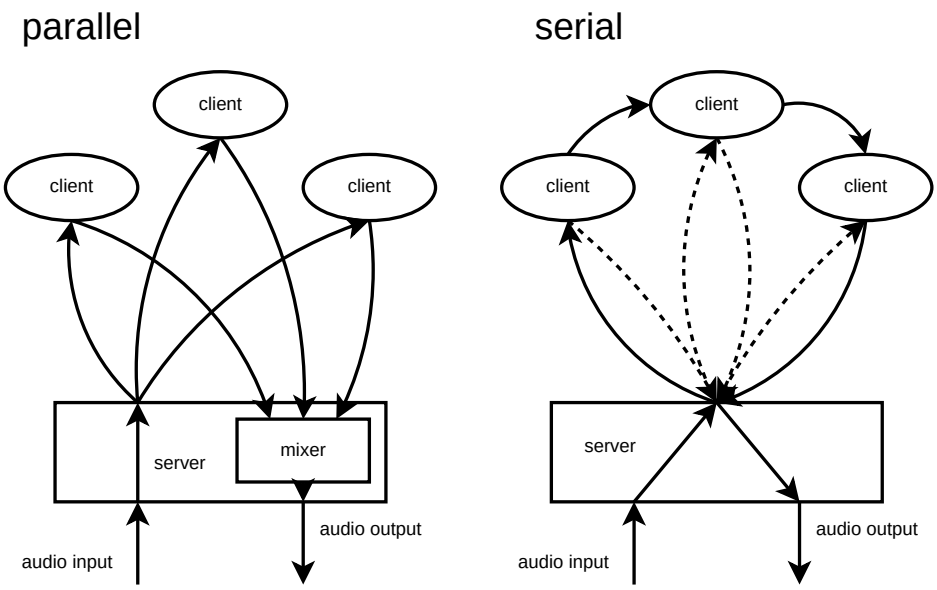


Figure 5.1: Two audio processing topologies

always schedule that process. This may look like a difficult task, but in fact this is what will happen even if the scheduler does not know about this, if only processes do not have any other tasks and each of them block waiting on data.

With parallel topology, the problem becomes more real. Ideally, the server should run until it has sent the data to each client, and run again only after clients have finished. Otherwise the execution can ping-pong between server and clients, which is not necessary because the server can wait for all clients to complete before it needs to continue. The scheduler can not know this without any extra information about processes. Depending on the complexity of the system, the extra price paid by too frequent context switching might vary considerably — nevertheless, it is possibly not insignificant, as was considered in section 3.5.

Another problem in distributed systems is scheduling that is internal to the processes of the system. Some processes commonly provide services to other processes, for example, a client may request some action from a server. Thus, because a server may need to be handle many requests at any time, it raises the question on how should those requests be prioritized. This form of scheduling of clients is outside the system scheduler, but needs to be built into application itself.

The problem is actually relevant at least in the case of X server, that can benefit from applying some fairness to servicing clients. Otherwise any one client could tie up the server by repeatedly sending requests that trigger high workloads. One solution to this can be a scheduling algorithm proposed by Keith Packard, where

> Each client is assigned a base priority when it connects to the server. Requests are executed from the client with highest priority. ... Clients with the same priority are served in round-robin fashion, this keeps even many busy clients all making progress.[30]

On the other hand, X server might be a special case, because the work that requests create can vary greatly. If each request (or some other kind of atom of communication) had the same size and caused a similar workload, then simple FIFO scheduling is possibly totally adequate. In addition, one other (possibly naive) solution to this problem would be to use a separate thread for each client. Here, the scheduling decision would be deferred to the threading implementation. However, at least in the case of X this has been problematic, because most requests need access to display hardware, which thus needs locking and reduces the possibility for parallelism.

While servers should possibly apply fairness to client requests, what if some clients should be considered more important than others? A server may service clients with different system scheduling priorities, yet normally these priorities are not considered when servicing those clients. Because in distributed systems a process's progress may be dependent on other processes, a real-time process may need to wait for a server to handle its request before it can continue, or before the task it needs to accomplish can truly be finished.

This was in fact the primary cause of the difficulty in tuning the SVR4 scheduler to properly handle mixed workloads, that was referred to by Vahalia in the quotation at page 62. If a real-time client needs to use a non-real-time (graphics, audio, or some other kind of) server, there is no guarantee that the output will meet its deadlines. Changing the server priority to real-time may help in fixing the problem, but is actually not the right thing because the server will still treat all clients as equal.

Thus, a server will be servicing even non-real-time clients with real-time priority. Because scheduling of clients (by system) and client requests (by server) occur in different domains, this will likely lead to poor scheduling decisions in both domains. This difficulty is made worse by a rigid distinction between real-time and non-real-time priorities, because it means that scheduling can not be as adaptive regarding changing workloads as could be possible.

One solution could simply be modifying the server to prioritize client requests according to their system scheduling priorities. This has been tried with X server and Linux by David Ingram. When experimenting with some video players, he tried boosting a single player performance over others using two techniques: by setting it to a real-time priority and by setting the request scheduling priority in X server.

The results were that:

> It was found that for this test POSIX RT priority made no difference at all. However, the X priority made a significant difference. [13]

It is quite probable that this problem is not as clear with audio as it can be with video display. This is mostly because in graphics servers such as X server the generation of actual visuals is more heavily the task of the server itself, and the applications may have a much higher range of priorities regarding graphics than with audio. With audio, most servers (at least of the current ones) are typically a lot more simple than X that has comparatively a lot more complex protocol. Moreover, all audio data streams have stable and clear deadlines that needs to be met, otherwise the output audio flow will disrupt — this also means a narrower set of priorities.

In any case, if the bottlenecks of real-time performance in distributed systems occur in some of the processes in it (typically in servers), trying to solve those problems elsewhere may not offer any significant benefit. A real solution involves modifying the applications. But could there be better solutions? One could be limiting server functionality so that clients could have direct access to resources and accomplish what they need to without any server help. But for most applications, this does not seem a reasonable design decision, and has serious access control related difficulties. Another choice could be threading a server so that one thread would be created for each client, and have the system somehow associate these so that they are scheduled thus as if they belong together. Here, the problems are in the words "somehow" and "belong", because to actually make these possible may require some serious changes to the scheduler.

5.3.6 Security and scheduler partitioning

Real-Time scheduling policies (as well as using negative nice-values in traditional scheduling) also leads to a problem of resource monopolization. Because requesting special scheduling can mean that applications can monopolize CPU time, some access control is thus necessary.

Thus, setting scheduling priorities is commonly restricted to superuser. It is possible to get around this by installing applications setuid root, which obviously opens up serious security concerns. These can be alleviated by making the application to drop privileges, but this is hardly a satisfactory general solution. A somewhat better solution could be using *systrace(4)*[26] framework or some other means of access control, that make possible to grant special privileges to any single application.

This way, the need to run the actual application with superuser privileges could be bypassed.

While these mechanisms are useful, it does appear that the real problem is the lack of partioning mechanisms in Unix scheduling. It could be very useful if the system could be set up so that, similarly to the "quota"-mechanism that can be used to control disk usage, there could be controls on CPU resource usage. In most Unix systems, there are indeed some limits put to CPU usage, but only by indirect means. Limits can be (and commonly will be, by default) placed on number of processes that can be run, the number of file descriptors that can be open, and other similar resources. Yet while this translates to some limits on CPU time that can be used, it can hardly function as a basis for resource partitioning.

Proper mechanism to partition scheduling would lead to two things. First of all, if some user or group could be guaranteed some absolute portion of CPU time, then all real-time processes run by that user would be confined to that CPU time portion. There would not be an issue of resource monopolization, since only some level of CPU usage would be explicitly allowed. Of course, this does not mean that users could not exhaust all the resources that they have available, which might happen either accidentally or purposefully.

The second point is that proper partitioning could function as a basis that allows the implementation of scheduling policies in userland. This would allow users having control over the scheduling of processes they run, inside their own portion of the system scheduling framework.

5.4 Research on scheduling algorithms

There has been a lot of research on scheduling algorithms that have not yet seen implementations in mainstream operating systems. This may be for various reasons, but one is probably that even if a scheduling algorithm may look good in theory, that does not mean it will be effective in practice. Schedulers need to scale to handle thousands of processes, often also considering the nature of multiprocessor environments, while ensuring correctness with little overhead. The effectiveness demands does put some limits on what algorithms can be considered practical.

One interesting concept that appears quite practical is implementing *handoff scheduling*, a concept that was implemented in Mach scheduler. In Mach, a process could yield the CPU to another process, thus bypassing the system scheduling queue. This looks like a very useful feature for a distributed system on a same scheduling domain, because it allows for effective scheduling decisions. But to

make this useful, applications need to know about how their progress depends on the progress of other applications, and then co-operate accordingly. There is also a problem of treating these processes' priorities and quantums properly in the system scheduler. Therefore, compared to possible costs, the benefits of *handoff scheduling* may be too small.

One interesting approach would be to start with a scheduling concept that is more natural to hard real-time systems and then make it more flexible to be suitable for a wider range of tasks. This could mean *deadline-driven scheduling*, where processes set deadlines for the tasks they must accomplish, and then "the basic principle is to vary priorities dynamically, increasing the priority as the deadline draws closer."[43, p. 144] Deadlines could have different strictness levels, where soft real-time deadlines should be met with some probability, and other tasks may have no deadlines or deadlines of many hours.

This kind of scheduling should be useful in a system where the required response times for processes are known, as is the case with real-time audio processing. On the other hand, the concept does not generalize well to all other kinds of processes, which suggests a need for some partitioning of CPU resources. Deadline-driven scheduling could also be combined with some form of admission control, where applications must request for all resources they need, including CPU time, before they can begin. However, the main problem with admission control schemes is that predicting the resource usage of any sufficiently complex application can be very difficult.

The reason why deadline-driven scheduling can be a good answer to a particular problem such as real-time audio, is that it considers the latency requirements of an application as its primary concern. This actually highlights a problem with most scheduling algorithms, that are mostly concerned with CPU resource usage, while latency is a secondary (and related) concern. The traditional Unix scheduling algorithm is based on an idea that low latency requirements should have an inverse relation to CPU resource requiments. Thus, processes that do not use a lot of CPU time will get better latency. Real-Time scheduling policies simply bypass this setting.

However, while this model works well for many workloads, it is problematic for real-time audio processing that typically needs both low process dispatch latency and high CPU resource usage. While Unix schedulers typically treat these two factors as intertwined, possibly for efficiency reasons, there is no particular reason why it should be so.

Scott A. Brandt et al. have proposed a scheduling algorithm based on this idea, calling it *Rate-Based Earliest Deadline (RBED) scheduler*[2]. In RBED, these two factors

are kept separate, and also made dynamically adjustable. The algorithm works by dispatching processing in Earliest Deadline First order, but interrupting them if they exceed their CPU time. This means that real-time processes must inform scheduler about their needed period and worst-case exexution time.

The scheduler can also guarantee hard real-time by not accepting to run them if not enough resources are available. It determines suitable periods for rate-based tasks according to their processing requirements, while best-effort tasks are given semi-arbitrary periods. This way the EDF algorithm can be used, yet it is still possible to dynamically adjust the deadlines of soft-real-time and best-effort processes. Their algorithm has been tested as a modification to Linux kernel, and their conclusion was that

> Its management of best-effort processes closely mirrors that of Linux, its management of soft real-time processes is better than that of Linux, and it provides guaranteed hard real-time performance.[2]

5.4.1 Fair-Share scheduling algorithms

An important positive quality of traditional unix scheduling is that all processes will get at least some CPU time and thus can make progress. The scheduler tries to be protective about system resources, by not allowing any group of processes to monopolize resources. Yet most common approaches to provide real-time performance are based on an exact opposite idea: the system allows real-time processes to easily consume all CPU time and other resources they demand.

It might be interesting to look at how could real-time guarantees be achieved by basing them on stricter protective measures instead of simply allowing some privileged processes to bypass them. *Fair-share scheduling* means scheduling based on algorithms, that allow allocating a fixed share of CPU time to some group of processes. Superuser can choose how to share CPU time between groups, where each group will have a guaranteed portion of CPU time. The scheduler enforces that any group will not exceed its portion, and in the case that not all CPU time is used by some group, that unused portion can be divided among other processes that request it.

What is interesting in this scheme is that it is based on a protective idea of system resources, allowing accurate control of CPU resource sharing between users, but that the mechanism also "could be used to guarantee resources to critical applications in a time-sharing system."[43, p. 144]

Consider, for example, that process groups (that each get some guaranteed fixed

share of CPU time) were divided according to users, that is, the processes in each process group were all the processes owned by some specific user. Then a user "critical" was created, and it could be guaranteed that that user could receive ninety-five percent of all CPU resources it asks for. This would mean that all processes run by user "critical" could nearly monopolize the CPU for their needs (they would need to compete mostly only which each other). By setting some applications setuid "critical", other users could also run those applications with special CPU time guarantees.

However, could the "critical" above be also be translated to "real-time"? If an application could receive ninety-five percent of all CPU time, is that not almost like real-time? The answer is that it depends: it depends on the time period where that ninety-five percent can be guaranteed. If a process could get that much CPU time during a period of every fifty milliseconds, that would mean that a scheduler could have a maximum of five millisecond delay between a process requesting to be run and its actual scheduling. This can be quite acceptable for many real-time tasks.

This illustrates the problem that, even though interrelated, the CPU time and latency guarantees are really two distinct issues. The practical problem with the approach presented above is getting accurate measures of CPU usage within such a short period as fifty milliseconds. Unfortunately, this is not possible by using the resource usage monitoring facilities present in most Unix kernels. In BSDs, the system timing accuracy is ten milliseconds, and the estimates of processes' CPU usage are based on a lot longer periods than that.

Nevertheless, these system limits can actually be overcome by implementing a scheduler that will not rely on common system accounting methods, but instead provides support for running processes in fractions of a normal process time quantum. If these fractions correspond to the proportions that each process group can get of all CPU time, then the scheduling delays will also be limited within the quantum. Then, setting the quantum to smaller values will also mean better real-time response for applications that need it.

On the other hand, setting the quantum to some low value also means higher process scheduling overheads. In an ideal algorithm, it should be possible to have different quantums for different processes or process groups. It should also be possible to dynamically adjust these quanta, the proportions that are used by processes, and which processes belong to which proportion groups.

This is in fact the kind of research done by Stoica et al., presented in their paper A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. The main claim was that

> ... our proportional share algorithm (1) provides bounded lag for all clients, and that (2) this bound is optimal in the sense that it is not possible to develop an algorithm that affords better bounds. Together, these properties indicate that our algorithm will provide real-time response guarantees to clients and that with respect to the class of proportional share algorithms, these guarantees are the best possible. [40]

Even though the proof for this was somewhat complex, the actual algorithm was not too difficult to implement as a prototype in a version of FreeBSD. Their conclusion was that the overhead of the implementation did not cause it to miss the theoretical bounds of the algorithm.

One of the main benefits of fair-share scheduling approach is that it can provide a seamless integration of the scheduling of both time-sharing and real-time processes. This can address some of the difficulties in scheduling soft real-time applications, because the CPU requirements of many applications of this class have significant variations in time. Most imporantly it can sometimes be desirable to temporarily degrade the performance of some real-time applications, and also to protect the system from a single process monopolizing all system resources. These concerns are significant on general purpose workstations, as opposed to embedded, industrial hard real-time applications, for which a more rigid notion of real-time scheduling is necessary.

5.4.2 Approaches based on partitioning

Fair-share schedulers can provide a good basis for partitioning CPU resources among various process groups, but they still do not address the question of partitioning the scheduling process itself. That is, a possibly more ideal solution might be separating scheduling activities into several autonomous groups, that have full control over how they are scheduled within their own domain. Furthermore, the scheduling system could be built as a hierarchy, so that any scheduling group (with some number of processes) could further be split into subgroups, having control over their scheduling subdomains.

Consider, for example, a case where a desktop user has some applications running and some of these applications have threads that make up a real-time audio system, by communicating with each other. A hierarchical scheduling framework would allow the user to have some amount of CPU resources (also covering process dispatch latency), which is needed to guarantee that other users (running system daemons and such) will not interfere with user processes. In addition, the framework would

enable the user to reserve some portion of available CPU resources for scheduling real-time audio thread, even to the level that those threads could be scheduled by some special thread of some application.

The ideas behind hierarchical scheduling seem enticing, mostly because it allows the scheduling framework to easily adapt to various scheduling needs. After all, it is unlikely that any single scheduling algorithm is likely to fit to the needs of all applications. While any single application can have control over the scheduling of all its own threads, independently from the system scheduler, a need for these kinds of schemes becomes more clear once we are considering systems which consists of many independent processes.

Thus, it is interesting to see that the DragonFlyBSD project has some aspirations to try walking on this path. The separation of the LWKT thread scheduler from the userland process schedulers, which could possibly be exported to userland and run as user processes, clearly allows for more experimentation regarding fair-share scheduling policies, as well as with hierarchical scheduling frameworks.

The main problems with hierarchical schedulers are their relative complexity, while they are also more likely to contain a higher scheduling overhead than more traditional schedulers. In addition, in order for them to be useful in problems that they can provide solutions for, they require an interface that is quite different from the convential *getpriority()/setpriority()* used in Unix systems. However, research by Pawan Goyal et al. has shown that algorithms such as Start-time Fair Queueing (SFQ) can be used as a basis for hierarchical schedulers, and can provide soft real-time guarantees for applications. Furthermore, this algorithm was implemented as a modification to Solaris operating system, and the researchers' conclusion was that the scheduling framework "does not impose higher overhead than conventional time-sharing schedulers."[11]

Some related work has been done by Bryan Ford and Sai Susarla, that suggest *CPU inheritance scheduling* as a basis for hierarchical scheduling. The main idea is that instead of using a single kernel-level scheduler, threads may donate CPU time to other threads, and thus threads can be scheduled by other threads. This idea could be useful in a real-time audio system, where the audio server could be given a share of CPU time that should be adequate for real-time tasks. Then, the server could control the scheduling of its clients (or rather, do the scheduling itself) by donating them CPU time as necessary. The clients could be scheduled optimally, because the scheduler (the server) can know the correct scheduling patterns.

However, in CPU inheritance scheduling the real scheduling overhead may become a considerable problem:

> our framework introduces two additional source of overhead: first, the overhead caused by the [scheduler thread] dispatcher itself while computing the thread to switch to after a given event, and second, the overhead resulting from additional context switches to and from scheduler threads.[8]

Thus, if the main point of moving scheduling decisions to application-level was to avoid the problem of extra context switches between server and clients, we can see that CPU inheritance scheduling is a no-solution. On the other hand, when it comes to finding optimal process scheduling, the problem may actually be very negligible. The strengths of hierarchical scheduling frameworks do not lay in minor optimizations, but in making possible to have a much more fine grained control over scheduling decisions than is possible through simple priority control interfaces. Some of this power should be useful in providing a smooth desktop experience, that also includes a distributed, real-time audio system.

This has probably been enough about scheduling to get to some conclusions on various algorithms. One of the main problems in scheduling algorithms is finding a balance between ideal scheduling decisions and implementation overhead. The overheads can a serious concern, because scheduling is such an integral and frequent activity of the system. When considering the suitability of various scheduling algorithms for real-time audio, it is clear that the time-sharing scheduling algorithm in 4.4BSD is inadequate. Nevertheless, slight changes that use a more steep interpretation of nice-priorities can make it suitable for some real-time work, including real-time audio.

Real-Time scheduling policies such as the ones in SUSv3 can provide a better control over scheduling, but possibly provides real benefits only in somewhat complex real-time application settings. These two scheduling policies are likely the ideal ones when the goal is to reach very low latency levels, on the range of one to ten milliseconds. The use of hierarchical scheduling frameworks do not significantly change this situation, because they can be used in conjunction with static priority process scheduling.

The downside of real-time priorites (and extreme nice-values) is the lack of safety: the system may get out of control if any real-time process starts behaving unexpectedly, and starts consuming all free CPU time. Fair-share scheduling algorithms can offer protection against this, but as a consequence the algorithm makes it difficult to reach the very lowest levels of latency that the system might otherwise guarantee.

6 Interprocess communication

To enable communication between processes running on the same computer or between computers connected by a network, operating systems provide various *interprocess communication* (IPC) facilities. IPC has many different purposes. In addition to simple data transfer, they can be used for event notification, sharing of resources, and process tracing (which is used when debugging the execution of programs).

In Unix, the filesystem offers possibly the most basic form of IPC, as it is an environment that is shared by most processes running on the same computer. Practically all Unix variants also support signals, pipes and process tracing, and most current systems also provide BSD sockets, and some mechanism for sharing memory. Many of these techniques are associated with filesystem and its operations by design.

This chapter will introduce the most common IPC mechanisms used in Unix and especially BSD systems. Their efficiency and special characteristics are considered, as these relate to building real-time distributed audio systems.

6.1 Signals

Signals are a very simple mechanism for notifying processes of various events, typically related to system operation. In most cases, the events have no relation to activity that a process is currently performing, while they also need to be handled immediately at their arrival. This is done mostly because signals serve an important purpose as error handling mechanisms, which is what they were originally designed for. Most signals have predefined meaning, yet the signals `SIGUSR1` and `SIGUSR2` can be used with application specific meanings.

Signals can be used as a synchronization mechanism, if processes call *sigpause(2)*[3] to wait for a signal to arrive before they will continue execution. Processes can also reject some arriving signals, by setting their *signal mask* accordingly. A process receives signals only after the kernel has checked if the process has pending signals, which happens when a process blocks to wait for or wakes up from an event:

> The receiving process becomes aware of the signal when the kernel calls the *issig()* function on its behalf to check for pending signals. The kernel

calls *issig()* only at the following times:[43, p. 85]

- before returning to user mode from a system call or interrupt
- just before blocking on an interruptible event
- immediately after waking up from an interruptible event

Thus, while there is a delay between a process sending a signal and another one receiving it, a process can not run normally before it has handled the signals that has been sent to it. Then, as a process has received a signal, its execution will switch to a *signal handler* that will be run in its own context.

While signals look like an ideal method for synchronizing applications running on a same computer, it is questionable whether or not they are efficient enough for the purposes of real-time applications. The main point of signals is that they are supposed to be used with *exceptional* events, such as errors, and not as a frequently used scheduling mechanism or something equivalent. Thus, it is possible there are some efficiency problems:

> Signals are expensive. The sender must make a system call; the kernel must interrupt the receiver and extensively manipulate its stack, so as to invoke the handler and later resume the interrupted code. [43, p. 151]

Furthermore, Paul Davis experimented with signals as a synchronization mechanism for Jack Audio Connection Kit (introduced in next chapter), a real-time audio connection system, and regarding this he explains:

> I think early on in the implementation of Jack I did try using signals which seem like they would be sort of custom made for this, but they're actually way slow, incredibly slow.[6, 24:25–24:35]

To actually know how slow they were, I did some experiments with some very small programs. A program was written that continuously sent a signal to itself, where a signal handler was set to count how many times it got invoked. In the test computer (running OpenBSD, for exact hardware configuration see appendix C.3) the program would receive 170000 signals in a second, corresponding to an average signal handling time of about six microseconds.

In another case, a program was written to run as two separate processes, where one would continuously send signals to the other process, and other process's signal handler would count the number of signals received. In this case, the signal handler was invoked over sixty thousand times each second, meaning each signal

took about twenty microseconds to send, receive and handle. Sending signals between two processes takes a longer time because of context switching and scheduler overheads. These experiments suggest that signals are so efficient that there should be no problems in using them as a synchronization mechanism for most real-time applications, contrary to what Paul Davis said about his experiences with Jack.

On the other hand, it is quite probable that signals will behave differently in a practical setting. To handle signals, some process stack manipulations need to be performed to ensure that signals will not interfere with normal process execution. The test programs did nothing but signal handling, and it can be that these stack manipulations can be considerably more expensive in real programs. It can also be that signal handling may be delayed for some reasons that are not related to processes involved.

Unfortunately, the test servers which will be introduced in next chapter do not use signals for synchronization purposes, so a further insight on this issue remains to be seen. One likely answer is that it depends: different Unix systems may very well exhibit different behaviour regarding signals, as they were mainly meant for rarely occurring events, and thus their handling could be delayed without harm.

6.1.1 Real-Time signals

Some Unix systems also have support for *real-time signals*, as defined by SUSv3[24]. These are somewhat like normal signals, except that instances of each signal type may be queued, and the signal delivery will follow a priority hierarchy, so that handling of some signals can be made more urgent than others. Signal-generating functions can specify a *sigevent*-structure that can contain additional information for the signaled process.

Real-Time signal mechanism can be seen as a step up from simple signal delivery, yet there are still some problems associated with them. Signals may become "stale" — they may be in signal queues even when the events that triggered them no longer hold true. And, according to Jonathan Lemon,

> a further drawback to the signal queue approach is that the use of signals as a notification mechanism precludes having multiple event handlers, making it unsuitable for use in library code.[17]

This problem is actually present in ordinary signals as well, and very relevant for audio applications, because hiding synchronization issues into audio libraries should be a good design principle.

Real-Time signals could be seen as a precursor of *kqueues* (kernel event queues) as a generic event notification mechanism. Because none of the current BSD variants support real-time signals, they are not considered further in this thesis.

6.2 Pipes and sockets

Anonymous pipes, named pipes (FIFOs) and sockets are the main IPC facilities provided by Unix systems for sending arbitrary data from one process to another. They all have an interface similar to general filesystem I/O, yet individual sockets and pipes may not have any names on a filesystem. The main difference between pipes and sockets is that socket connections can be set between processes that have no relation to each other, whereas with pipes, processes need a common parent to set up the pipe for them. Thus, sockets are needed for communication between processes over a network, and sometimes in local communication when pipes are simply inadequate for the required task.

Even though sockets are a more natural fit for building distributed systems, pipes can still be useful if all processes are run in a single computer. Thus, we will first consider what benefits they might offer, and what problems need to be solved when using them.

6.2.1 Anonymous and named pipes

Pipes are the simplest communication channel in Unix that can be used to send any data between processes. Standards dictate they are uni-directional, but in fact some systems such as all BSDs have implemented them so that it is possible to send data to both directions. It is of course possible to create two pipes instead, and use both only unidirectionally, if communication between processes needs to happen to both directions. Another option is *socketpair(2)*[3] system call, that creates a pair of socket file descriptors connected to each other, which properly allows bi-directional communication.

A process creates a pipe with a *pipe(2)*[3] system call, which creates a channel from one communication end to another. As one process writes data to the write end, another process can read the same data from the read end. 4.4BSD system implemented pipes internally as sockets, but at least FreeBSD and OpenBSD have changed this implementation to a different one to gain better efficiency:

> For performance reasons, FreeBSD no longer uses sockets to implement pipes and fifos. Rather, it uses a separate implementation optimized for

local communication.[20, p. 34]

This should mean that an audio server system that uses pipes for transfering data with clients could possibly have a higher bandwidth than one that uses sockets. However, the problem with pipes was that processes need to have a common ancestor process to set up the endpoints for communicating processes. So how could this be done by an audio server that wants to communicate with a client through a pipe?

One way to do this is to always start clients so that they will always be started by the server. This would make it impossible to connect to a server that has started later than a client. But in any case, a server could create a pipe, then fork itself, and the child process could execute a client that inherits the other end of the pipe from the server. This way it is possible to set up a pipe between two mostly unrelated processes.

In fact, Unix systems support two kinds of pipes: anonymous pipes and named pipes, also known as FIFOs. FIFOs allow to create a name to a filesystem that functions like a pipe. Thus, the main limitation of pipes actually reduces to the fact that processes that communicate over pipes need to run on the same computer. This means that for high bandwidth demands (as is the case with audio data) it is better to not use sockets except as a mechanism for control data, for passing pipe file descriptors, and for other data where efficiency is not a concern.

Many Unix systems also support a mechanism called *message queues*, which is a System V invention similar to pipes. They should have some advantages over pipes, such as ability to transmit discrete messages, and associate priorities with them. However, they were not designed for exchanging large amounts of data: "message queues are effective for transferring small amounts of data, but become very expensive for large transfers." [43, p. 161] As System V message queues are mostly a legacy feature in modern Unix systems, and do not follow the design style commonly used in BSD systems, they are not considered in any better detail here.

6.2.2 Sockets

The main benefit of sockets over pipes is the fact that they can be used for communication between unrelated processes that can also reside on different computers on a network. But to do this, processes need some common namespace which they can use to agree on when creating the connection. These namespaces are known as "domains". The commonly used ones are the internet domain and the Unix (or local) domain. The internet domain is used for communication between independent machines, and the Unix domain, that uses filesystem for rendezvous between

processes, is confined to local processes only.

Unix domain sockets have some benefits over internet sockets, taking advantage of the fact that sockets can rely on existing in the same computer. At least in BSD kernels, the networking layer is kept separate from the interprocess communication layer. This means that data that passes through Unix domain sockets does not need to pass through the networking layer, which means less IPC overhead. At least the X Window System takes advantage of this fact:

> On a Unix System, when the X client and X server are on the same host, the Unix domain protocols are normally used instead of TCP, because there is less protocol processing than if TCP were used.[39, p. 486]

Thus, sockets over Unix domain should be more efficient compared to sockets using internet domain, if both are used in a same machine.

Unix domain sockets can also be used to pass file descriptors between processes on the same computer, which is not possible in internet domain. This can be very useful when setting up communication channels between applications. For example, to set up a channel between a client and a server, a server can create a socketpair, or a pipe, and send the other endpoint to the client. Or, to set up a channel between two clients, a server could send both endpoints of the socket to different clients.

In Unix domain it is also possible to determine the effective user and group IDs of connecting processes with *getpeereid(2)*[26] system call. This can be used as an access control mechanism in an audio server that runs on a multi-user system.

When comparing Unix domain sockets with pipes for local communication, we can see some features that pipes are lacking, that can be very useful for some applications that need local communication. It is possible to *send(2)*[3] *out-of-band data* through a socket, that arrives "outside" the normal data stream, and can contain information about exceptional conditions or whatever an application protocol decides to use it for. The arrival of out-of-band data can also be used to trigger a `SIGURG` signal at the moment the data arrives, which can be used to prioritize the handling of some data and situations over normal data processing.

Another useful feature of sockets is that applications can exercise control over watermarks and I/O buffer sizes used in socket I/O. Normally, all input and output is buffered in kernel, and output is sent only when data size exceeds a low watermark. Because this introduces latency, an application can obtain a better latency response from socket communication by setting the low watermark to a suitable value. The *setsockopt(2)*[3] manual page for 4.4BSD states that "the default value for SO_SNDLOWAT is set to a convenient size for network efficiency, often 1024,"

a value that is also left unchanged in OpenBSD. On the other hand, because the default low watermark is as low as 1024, the latency added by buffering is only $1024/2/2/44100 \approx 0.006$ seconds with CD-quality stereo audio. Nevertheless, it is useful that an option for tuning this exists.

As with all file I/O, pipes and sockets can be used in both synchronous and asynchronous (non-blocking) modes. We will take a look at what these can mean for the real-time systems we are primarily considering.

6.2.3 Synchronous and asynchronous I/O

The basic difference between synchronous and asynchronous I/O is that when synchronous I/O is used, a process will block waiting for the I/O request to complete before its execution will continue. But this creates a problem for real-time applications: a process may stop waiting for an indefinite time until the I/O request can be completed. If a process has some other real-time critical tasks that needed to be done, those tasks will be delayed and can miss their deadlines, even if they did not depend on the I/O request made by the process. Hence the need for asynchronous I/O.

One way that asynchronous I/O is supported by all BSD systems is that I/O requests can be made on *non-blocking mode*, that is, the operation is not carried out if it can not be performed immediately. This way, a real-time process can keep running and periodically try out the I/O operation, until it can be done without blocking. However, this leads to a complex programming model, because an application needs to keep track of the I/O operations that had to be deferred for later requests.

Handling this complexity can be made easier by using signal-driven I/O. We already considered the effectiveness of using signals for real-time work, so all that was true there is also true here. In any case, it is possible to use *fcntl(2)*[3] system call to ask the kernel to send a `SIGIO` signal to any process requesting it, each time it becomes possible to do I/O operations on some chosen file descriptors. This way, all asynchronous I/O can be handled by a signal handler, which seems like a good fit, because signal handlers are triggered asynchronously anyway.

Asynchronous I/O operations will be prioritized over normal tasks, because signals will interrupt normal process execution flow. But the main problem in this model is that it is unsafe to write complex signals handlers. This is because of the risk of race conditions — in Unix systems, typically only a small subset of C library functions are written to be signal thread safe.

One other way to handle this is to use a threading library to hide underlying asyn-

chronous operations behind a synchronous interface. Applications can be threaded so that there can be separate threads for handling I/O operations. Threading libraries commonly convert all their I/O operations to non-blocking mode, so that the library can schedule some other threads instead of blocking. Then, the library can periodically test if the I/O thread can finish its request.

There are a few problems in this scheme. One is that not all non-blocking I/O requests can actually be made non-blocking. Consider a case where a process wants to read some data from a filesystem. If that data does not happen to be cached in memory, there is always a small delay caused by hard drive seeks and reads. Thus, if a process does a non-blocking read, it can never complete immediately.

Basically, the system should take note of the first non-blocking read, then begin reading that data into disk cache, and let the non-blocking request complete only after the data can be read directly from cache. However, this is not how the current BSD systems function, but instead they simply convert non-blocking file I/O calls to blocking calls:

> This facility is not implemented for local filesystems in FreeBSD, because local-filesystem I/O is always expected to complete within a few milliseconds.[20, p. 229]

This means that if an operating system implements threading using only a user-level library, and thus the kernel has no knowledge about application threads, then the progress of a real-time process can momentarily stop when it makes filesystem operations. This is the case with the current version of OpenBSD. However, the problem does not exist in threading subsystems where application threads can be scheduled by the kernel, because blocking calls block only the thread that calls them and not the whole process. The current versions of FreeBSD and NetBSD have been implemented this way.

On the other hand, the problem can actually be solved in systems such as OpenBSD without changing the system. An application can fork a child process, set up a pipe between the child and the parent, let the child handle file I/O and caching, and let the parent read data from the pipe. It may be somewhat cumbersome, but it works, because reads and writes to pipes (and sockets) can always be made properly non-blocking.

All this means that if a real-time audio server or clients need to do filesystem I/O, using some kind of asynchronous I/O mechanism is necessary to ensure real-time performance. Because hard drives do not give very low latency guarantees, applications need to anticipate the use of data and buffer it beforehand. This is

especially relevant in audio systems such as multitrack recorders, that process large amounts of audio data, which can not be made fit into memory. The asynchronous I/O mechanisms can become useful for socket connections and other I/O as well.

While asynchronous I/O model can be usefully implemented with non-blocking system calls, and signals or threads, asynchronous I/O operations could also be made a part of the system itself. This way, an application could simply request the system to do an asynchronous I/O operation on its behalf, and then later on make a query about whether the operation has completed.

If a system can make a distinction between asynchronous and synchronous operations on a kernel level, asynchronous requests could be queued properly and handled in some priority order. Moreover, it becomes possible to do filesystem operations in the background, so that an application can request to read some data from a file and later get a notification that the data is available for an immediate read.

In fact, SUSv3 contains a definition for an asynchronous I/O interface that can be used to achieve these goals. The interface uses separate system calls for asynchronous I/O, and its primary purpose is to solve the problems that real-time applications can face when dealing with I/O operations. Kernel level asynchronous I/O is typically implemented by kernel threads:

> Kernel threads are useful for performing operations such as asynchronous I/O. Instead of providing special mechanisms to handle this, the kernel can simply create a new thread to handle each such request. The request is handled synchronously by the thread, but appears asynchronous to the rest of the kernel. [43, p. 53]

This kind of asynchronous I/O interface has been implemented on kernel level in FreeBSD and DragonFlyBSD. On the other hand, it is lacking in NetBSD and OpenBSD, that need to resort to non-blocking calls, threads and I/O signal driven I/O. For real-time applications, it is not necessary as such, but can provide a simpler programming interface and also better performance, as some of the I/O handling work can be moved to kernel level.

6.2.4 Kernel event queues

One problem in Unix design is the multiplicity of various event handling mechanisms. There are signals, a separate asynchronous I/O interface that needs to be used for queries about I/O operation completions, *select()* and *poll()* system calls for multi-

plexing file and socket I/O, while many events do not have any elegant mechanism at all.

Kqueues (or, *kernel event queues*) aim to solve this problem, by providing a generic interface for monitoring kernel related events. Its design makes it easier to add support for new kinds of kernel events. Applications can register to be informed about selected events, such as activity on some file descriptor, and wake up once any such activity occurs.

One outcome of the design of kqueue-mechanism is that polling for events can be made considerably more efficient. A common design for servers is to set up connections with sockets and use *select()* or *poll()* system calls to listen for activity on those sockets. These system calls are fairly effective when the number of connections on any server remains within a few thousand, but have scalability problems with a higher number of concurrent connections:

> These problems stem from the fact that poll() and select() are stateless by design; that is, the kernel does not keep any record of what the application is interested in between system calls and must recalculate it every time. This design decision not to keep any state in the kernel leads to main inefficiency in the current implementation.[17]

Kqueues solve this problem by keeping track of which events applications are interested in. Thus they eliminate some overhead that occurs when both the kernel and some application have to search through all the open file descriptors to find the ones where activity occurs.

Because real-time servers need to query for incoming data more frequently than non-real-time servers, it seems that kqueues can provide an efficiency benefit. However, it has to be noted that, assuming current hardware technology, real-time audio servers are not going to have many thousands of connections open at any time. To satisfy real-time requirements, staying within latency deadlines with thousands of connections is very difficult in any practical settings, and if connections are passive, they can be dropped altogether, because the data should arrive with some latency guarantee. Because the number of connections has to be much lower, the bottleneck with stateless *select()* and *poll()* is probably not a serious issue. Thus, while it is difficult to come to any conclusion without doing tests, it seems unlikely that kqueues could offer some significant benefits for real-time audio data connections.

Kqueues can also be used to solve some possible efficiency problems with signals. They can be used to receive notification about received signals that are blocked by a process signal mask. Signals will not be handled normally by a signal handler,

but instead a process can make queries about their arrival whenever they want to. Thus, expensive stack manipulations associated with normal signal handling can be avoided, and signals can be used as "just another IPC mechanism". Because this provides a means for decoupling signals as an IPC facility from the conventional signal handling interface, there should be less obstacles for using signals in real-time tasks.

One other interesting property of kqueues is the possibility for a kqueue to monitor another kqueue, to find out when it receives any events. This allows the creation of a priority hierarchy for event handling.

6.3 Shared memory

One problem with sockets and pipes — as they are commonly implemented in Unix systems — is that they are not as an efficient mechanism for passing data between processes as is possible. Each pair of send/receive -operations involve a data copy from the address space of a sending process to kernel memory, and from kernel memory to the memory area of a receiving process. But these data copies can be bypassed altogether by letting processes share some segments of their physical address space. Once shared memory segments are set up by using system IPC facilities, processes can then communicate without a need to do any (relatively expensive) system calls, and thus reaching a higher data bandwidth should be possible.

On the other hand, shared memory model does have some problems. First of all, with shared memory, it is not possible to exchange data with processes running on different computers. The exceptions to this are some operating systems that support the distribution of their virtual memory address space over a network, but these do not include any mainstream systems, or BSDs, for that matter. Another problem is that shared memory is probably a better fit for processes, that have a real need to co-operate using some common data structures, as opposed to using it as a super-fast method for passing data.

One significant problem with shared memory is also that using it can sometimes lead to a somewhat complex programming model, because access to shared memory segments needs to be controlled in some way. In client-server audio systems, it should be the server's task to create and manage shared memory segments so that clients have access to appropriate memory segments. The segments should also be arranged so that the number of needed data copy operations can be minimized, because otherwise the main benefit of shared memory would be lost. Yet even

when two independent agents have basic permissions to the same shared memory segments, they need to coordinate their actions on those segments, so that data corruption can not occur.

Shared memory has been used successfully at least in some X Window System implementations, that use it for communication between X server and X clients, in case they are running on the same computer. But it should be noted that data representing graphics behaves fundamentally differently in time than audio data. Graphics data is not necessarily stream-oriented, but can be rather static, mostly consisting of still pictures. The portions of any video frame that change in time can be rather small compared to the portions that stay the same.

It makes a lot of sense to use shared memory for such applications, because each new video frame can share a lot of data with a previous frame, and only small segments of graphics data may need to be modified. When contrasting this with audio data, it can be seen that audio data is in a constantly changing state. While a previous audio data block typically holds some similarity with a current data block, still none of it can be reused, because the whole stream is continuously changing.

Thus, it appears that the semantics of stream oriented IPC methods such as sockets and pipes are clearly a more natural fit for passing audio data. It is also questionable whether the efficiency benefit of shared memory is worth the limitations inherent in it. In fact, it is not totally clear whether any such benefit exists at all. This is because the access control mechanisms needed with shared memory have additional overheads associated with them, which do not exist with socket IPC.

Most current Unix systems support two shared memory interfaces, one developed as a part of System V IPC, and BSD-style *mmap()* for mapping files to process memory space. The main difference between the two interfaces is that System V IPC has its own namespace that is not associated with filesystem, whereas *mmap()* relies on a filesystem to provide the shared memory segment. The main problem with a separate namespace is that applications may accidentally abandon some shared memory segments (by crashing or in some other way), that will then consume memory resources for no reason. When a filesystem is used as a basis for shared memory, in a similar situation the consumed resource will be disk space. That space will be associated with a name in a filesystem, and thus easily spotted and removed, or cleaned up at boot time.

On the other hand, shared memory mechanisms that rely on filesystems have their own problems. When using *mmap()*, the system may flush the contents of the memory segment to disk, which is a completely pointless and wasteful operation if the *mmap()* is merely used to provide the means for passing data between processes.

One solution to this is use a memory resilient filesystem to hold shared memory files, but setting this up not trivial.

Because the use of shared memory segments commonly requires some access control mechanism, System V and *mmap()* APIs can be contrasted on this basis as well. For this purpose, System V IPC provides kernel-based semaphores, while *mmap()* actually has none. On the other hand, some filesystem operations can serve as a substitute for them, and might be a natural fit with *mmap()*. Vahalia writes about the conventions before semaphores were supported by Unix kernels:

> On early UNIX systems that did not support semaphores, applications requiring synchronization sought and used other atomic operations in UNIX. One alternative is the *link()* system call, which fails if the new link already exists. If two processes try the same *link()* operation at the same time, only one of them will succeed. It is, however, expensive and senseless to use file system operations such as *link()* merely for interprocess synchronization, and semaphores fill a major need of application programmers. [43, p. 159]

On the other hand, nothing prohibits from using System V semaphores in conjunction with *mmap()*.

The major efficiency problem with file system operations is that they require system calls. But in fact this problem is present with System V semaphores as well, because they are kernel-based. Marshall Kirk McKusick et al. point this out:

> The overhead of using such semaphores is comparable to that of using the traditional interprocess-communication methods. Unfortunately, these semaphores have all the complexity of shared memory, yet confer little of its speed advantage. [19, p. 138]

This means that the possible efficiency advantage that shared memory could provide can become more difficult to reach in practice. Each time a shared memory segment is accessed, a process needs to request a semaphore, which involves a system call. Once the process has done modifying data on the segment, it needs to perform another call to release the semaphore. Doing this is not very different from calling *read()* and *write()* to pass data through sockets.

Shared memory could provide better efficiency, if only the semaphores could be stored in the shared segments themselves. If processes can only be assumed to be cooperative (which is a reasonable assumption), they could get and set semaphores directly in the memory segments. There are two major technical problems that need solving in this scheme.

One problem is atomic access to memory: applications need to use a test-and-set CPU instruction or some equivalent one when handling semaphores, in order to avoid race conditions. An application library can be used to hide this operation behind some abstraction. A more difficult problem is that co-operation from kernel is still needed, because processes need to be blocked, while they wait for semaphores that they can not immediately get. As this happens only in situations where a context switch is needed anyway, this means that user-level semaphores have no overheads that are not also associated with kernel-based semaphores. Furthermore, most operations lack the overheads involved with system calls, that occur frequently in kernel-based semaphore solutions.

The *mmap()* facility was in fact designed so that using semaphores on it could be possible some day. This is exemplified by the fact the *mmap(2)*[3] interface includes a modifier parameter `MAP_HASSEMAPHORE`. But no BSD variant actually provides a semaphore facility that could be used to take advantage of this possibility. BSD systems actually do provide support for a semaphore interface defined in SUSv3, that is different from the System V semaphore API, and uses functions *sem_init()* and some others. But these can actually be used only between application threads, and not between processes, as explained in *sem_init(3)*[26] manual page: "this implementation does not support shared semaphores, and reports this fact by setting errno to `EPERM`." This is due to the fact that the SUSv3 semaphore interface is completely implemented by a user-level threading library.

While none of the current BSDs have implemented user-space semaphores, there has been some work on Linux to provide a similar facility. It has been happening under the name of *futexes*, or *Fast User Space Mutexes*. With *futexes*, the locking overhead associated with kernel level semaphores can be avoided. Consequently, the shared memory based IPC mechanisms may become more useful for passing audio data between processes. How much efficient this might be, however, is unclear.

It should also be noted that any real-world efficiencies will also depend on the tasks that applications need to accomplish. With shared memory, it is possible to deliver one block of data to several other processes, with roughly the same expense as it is possible to deliver it to one. Local socket based connections can not derive such benefit from multicasting, because copying data many times to process address spaces is still necessary. Yet, while there is no universal IPC facility that can always the best one for each situation, we will still take a hopeful peek at Mach IPC.

6.3.1 Techniques in Mach IPC

The main promise of efficiency that shared memory can give is based on the fact that to pass data between processes, it is not necessary to copy data. With user-level semaphores, some escapes to kernel mode may also be avoided. But there is actually no reason why read/write -operations with sockets could not be made zero-copy also. There is nothing in the interface of reads and writes to sockets and pipes that dictates that memory should always be copied between process address spaces.

This is an idea that Mach operating system has heavily relied upon. In Mach, the IPC facilities are not only something that the system provides for applications, but actually the system depends on them in its design. The system is microkernel based, and uses message passing for communication between subsystems. But the main difference between some other microkernel designs and Mach is that message passing is tied to memory subsystem.

Mach has implemented the concept of out-of-line memory to make message passing happen as efficiently as possible. In the commonly used message passing model, or "in-line memory", data is copied twice: first from sender to kernel, and then from the kernel to receiver. But message passing in Mach uses copy-on-write (COW) semantics, which in Unix culture are commonly used with the *fork()* system call. The sent data is copied only in the case that either the sending or the receiving process modifies it.

This message passing technique benefits from the fact that often the sent data does not need to be modified by neither the sending process nor the receiver. In this case, making an actual copy is not necessary. But the technique is more efficient even when either process wants to modify the data:

> Even if the pages are modified, this approach saves a copy operation. In-line memory is copied twice — once from sender to kernel, then again from kernel to receiver. Out-of-line memory is copied at most once, the first time either task tries to modify it.[43, p. 176]

If sockets were implemented using this idea of out-of-line memory, they could offer performance at least as fine as shared memory can ever offer. That is, at least in theory. One issue is that its implementation binds socket implementation tightly with virtual memory subsystem. This results with a higher implementation overhead, as the VM subsystem will have more bookkeeping to do. One other issue is that typical applications will modify the sent data segment immediately after it has been sent — thus forcing at least one copy operation. Thus, while out-of-line memory is a great idea, it may not work as well in practice.

It might be useful to note that, while DragonFlyBSD is moving to a messaging model for their kernel, they do not implement COW semantics as a part of their message passing API. It seems that this concept has not seen much use outside of Mach.

6.3.2 Solaris doors

Mach IPC merges in an interesting way some aspects of message passing and shared memory, but one problem with it is that implementing it on a typical Unix system is rather difficult. The *doors* facility in Solaris has another interesting approach to IPC, similarly stepping outside the message passing and shared memory -distinctions. With Solaris *doors*, it is possible to let processes invoke procedure calls of another process, that has properly set itself up as a "door server". These calls can do anything the door server has set them up to do, and can return data objects to the calling "door client" process.

There is one crucial aspect in the implementation of doors that sets them apart from other common methods of remote procedure calls. It is that once a door clients makes a call to a door server, the scheduler run queue is bypassed and context switches from the client to the door server and back are made immediately:

> In the case of a `door_call()`, control can be transferred directly from the caller (or client in this case), to a thread in the door server pool, which executes the door function on behalf of the caller. When the door function has completed, control is transferred directly back to the client (caller), all using the kernel shuttle interfaces to set thread state and to enter the dispatcher at the appropriate places. This direct transfer of processor control contributes significantly to the IPC performance attainable with doors.[21, p. 463]

Let us consider how the doors facility could be used with audio servers. One idea could be to make the audio server function as a door server that audio clients make calls into. However, this does not appear to have much significant benefit, because clients' audio threads still need to be scheduled by the system, and need two door calls to pass audio data in and out. Instead, one could make the clients function as door servers, and let the audio server be the door client. This way, the server could be used to invoke the audio processing functions in the audio clients, and each client could be handled with a single door call at each audio processing loop.

It appears that at least in theory that the doors facility can be adequate for passing large amounts of real-time critical data between processes. Doors can be

seen providing an elegant way for the server to schedule some specific parts of its clients' code, that its proper functioning depends on. Thus, doors might be an ideal IPC mechanism for distributed real-time systems whose parts are run on the same scheduling domain.

6.4 Data on a network

In this thesis we have mainly been concerned with audio servers running on a same machine, and thus on a local scheduling domain. Nevertheless, a small introduction to some of the most basic issues that arise when processes are distributed over a network might be in place. One thing to keep in mind is that the choice of IPC mechanisms governs whether it is possible to do this at all. Basically, audio servers that may be used with clients running on different computers, should rely almost exclusively on socket protocols, and specifically on internet domain sockets. But once this requirement is fulfilled, the existence of a network creates some issues that may need to be dealt in the audio server design.

6.4.1 Stream and datagram sockets

Actually, even the choice of an appropriate socket type is an open question, that exists on local domain as well, but becomes more pronounced in connections over a network. The main choice is between stream and datagram sockets, that correspond to TCP and UDP protocols that use the IP protocol. Stream sockets provide a reliable, connection-oriented data stream, in contrast to datagram sockets, that provide connectionless and unreliable message passing. Datagram sockets commonly have a lower overhead (and better raw performance), because with stream sockets there is some additional connection-oriented protocol data that must be handled by the system.

On the question of whether to use stream or datagram sockets, we could quote Stuart Sechrest from the 4.4BSD Interprocess Communication Tutorial: "The difference in performance between the two styles of communication is generally less important than the difference in semantics."[37] Because audio streams very naturally map to stream-oriented socket connections, on this basis we could decide that stream stream sockets are a better choice for audio data.

On the other hand, it might be that real-time applications are a special case where the difference in performance does matter. But this performance benefit is not likely in the latency behaviour of the two socket types. Even though stream sockets can be

slower initially, once the connection has been set up, there is no inherent reason why stream sockets would induce an unacceptably high latency. This might of course be true in some particular TCP/IP implementation, but nothing in the TCP protocol itself forces high latency once a connection has been set up.

However, one benefit of datagram sockets is that, because no guarantees of reliable delivery are given, datagram packets can and will be dropped if delivery does not happen in some timeframe. This is what should happen with real-time data: there is no use in transmitting stale packets that are can no longer be useful, but will only be rejected by the receiver. Possibly for this reason, datagram sockets are often used with real-time data, and the data packets are ordered by a higher level protocol such as RTP or by the application directly.

However, there is one subtle issue with latency in stream sockets and TCP protocol, that may sometimes arise and needs to be taken into consideration. To avoid network congestion, the sending of small data segments may be delayed by TCP protocols under certain conditions. Stevens explains this concept known as the *Nagle algorithm*:

> This algorithm says that a TCP connection can have only one outstanding small segment that has not yet been acknowledged. No additional small segments can be sent until the acknowledgment is received. Instead, small amounts of data are collected by TCP and sent in a single segment when the acknowledgment arrives. [39, p. 267]

While this algorithm induces delay, the algorithm only kicks in in situations where small amounts of data is sent in separate segments. It does not seem likely that the issue arises with audio data, because audio usually takes enough bandwidth that the algorithm will not be used, as IP packets will be larger. On the other hand, if latency requirements are very strict, then the data has to be sent in smaller blocks of data, which means that the Nagle algorithm will be used.

In fact, the whole problem can be completely avoided by disabling the algorithm, which can be done with the `TCP_NODELAY` socket option. This is what the X Window System does, and some other interactive network applications as well, to guarantee fast delivery of small messages such as mouse movements. Though it may not be necessary except when reaching for extremely low latencies, it is probably a safe choice to use this option with all real-time audio as well.

6.4.2 The network layer

Once data packets are copied from memory to network devices, to travel through a physical network, some factors can cause delays that are mostly out of operating system's control. One crucial factor that affects network latencies is whether the underlying network technology has any *Quality of Service (QoS)* -guarantees. The lack of any QoS means that any data packets on a network may be delayed by the handling of some other data packets. This situation is similar to ones that happen in process scheduling with time-sharing scheduling algorithms. Processes (packets) that need real-time execution (delivery) may get delayed by any random process (packets) that are run (delivered) instead.

Thus, the QoS guarantees are associated with network traffic scheduling, except that on a network level, no traffic scheduling may be possible. This depends on network technology used, and with contemporary technologies, the main division on this issue appears to be between Ethernet and ATM technologies:

> Ethernet technology lacks the latency and bandwidth guarantees needed by many multimedia applications. ATM networks were developed to fill this gap, but their cost has inhibited their adoption in local area applications. Instead, high-speed Ethernets have been deployed in a switched mode that overcomes these drawbacks to a significant degree, though not as effectively as ATM. [5, p. 70]

The QoS features need to be supported to some degree by the internet protocols, and in fact IPv6 includes features that make it possible to separate and handle real-time packets differently from normal traffic. Furthermore, QoS requires the use of a protocol such as Resource Reservation Protocol (RSVP) for proper allocation of network bandwidth for real-time streams.

It is interesting that network traffic priorities bring us yet another scheduling level, in addition to process scheduling on system level and client scheduling on server level. There should be some correspondance between audio stream priorities on server level and network traffic scheduling done by the kernel, because real-time audio data can not be treated as bulk data. But actually establishing this connection is not very difficult in many modern Unix systems (such as BSDs), if they only support traffic shaping in their kernel. For example, the *pf.conf(5)*[26] manual page in OpenBSD explains how the system's packet filter supports Class Based Queueing, Priority Queueing and Hierarchical Fair Service Curve scheduling policies for network traffic. As a simple case, all traffic from the audio server port could be associated with the highest priority queue using Priority Queueing

discipline. This should guarantee good network latency for audio streams, but only if the underlying physical network can enable it.

It should be noted that on the networking level, the real-time performance related problems are more or less non-existent. This is because in most Unix systems, the networking has been built into the kernel, and thus all network data handling happens with relatively high kernel level priorities. Thus, there are no practical issues in using most Unix (or BSD) systems for handling real-time network data, or apply real-time traffic shaping for it, as long as the data does not need to pass through user-level applications.

One important question that needs an answer is about whether distributing real-time applications over a network is actually feasible at all, because transmitting data packets over a network always adds some latency. The answer will obviously depend on the network. Such latency is commonly measured by round-trip time, that means the time it takes to send a short message over a network and receive a reply. It turns out that on a local network this time can be very short: "The time to transmit a short request message and receive a short reply in a lightly loaded local network between standard PCs or UNIX systems is typically under a millisecond."[5, p. 68]

This means that accessing data from a computer over a network can even be considerably faster than reading it from a local hard drive, if only the other computer has the data loaded in its memory. Latencies are so low that it is totally feasible to reach good real-time performance even when audio clients are run in computers connected by a network. But this becomes increasingly difficult if the computers are geographically dispersed, and data has to pass through several computers and other network devices.

It might be difficult to consider many actual applications for this, outside internet telephony that does not need very strict latency guarantees. But one might imagine the challenge that a symphony orchestra might face when performing a concert, where the players are geographically distributed and connected only by the internet. Providing technology for handling such extreme cases might be an interesting challenge for software and hardware engineers as well.

7 Servers and tests

As we have so far seen, the design of distributed real-time audio systems needs to be concerned with many levels of system design. We have so far covered some of the problems that may arise when building such systems, paying special attention to operating system level issues, such as kernel architecture, scheduling, and IPC facilities. Yet unfortunately the complexity level of most general purpose operating systems is such that it is very difficult to model the behaviour of such systems by computational methods, and calculate some meaningful results indicating how such systems will behave on different conditions. In fact, it is much easier to build such systems, and then gather some empirical data on their behaviour.

For this purpose, one server and client application was written, that were designed to provide a testing mechanism for various IPC mechanisms, as well as operating system real-time performance. These applications and their results with OpenBSD 3.7 are introduced and analyzed in this chapter. But before this, we will take a small look at some audio server systems that have been built and are used in actual real-world settings, paying special attention to Jack, that has had an excellent real-time performance among its primary goals.

7.1 Existing audio servers

Several audio servers have been implemented for Unix systems, even though none of them has achieved such success that their associated APIs could be considered standard. None of the commonly known servers implement most that would be required from distributed, synchronized real-time systems.

One of the earliest projects was X Audio System[42] that drew its inspiration from the X Window System. Goals were network transparency and an API that would allow good platform portability. Among possible applications that could benefit from the server were web audio and teleconferencing, but real-time processing was not a primary concern. The project, however, stopped at early stage. It seems that Media Application Server (*http://www.mediaapplicationserver.net/*) project associated with X.org tries to fulfill a somewhat similar purpose, supplementing the X Window System. Currently it is still in development stages, and not widely used.

Other audio server implementations include Network Audio System (NAS),

Enlightened Sound Daemon (ESD), ARTSD (Analog RealTime Synthesizer Daemon), and JACK. NAS is a fairly old system. A paper by Jim Fulton and Greg Renda outline it as providing "a powerful, yet simple, mechanism for transferring audio data between application and desktop X terminals, PCs, and workstations".[10] It is not stream oriented, however, and part of its design is ability to store single sounds in the server, with possibility to use them multiple times. It also lacks support for synchronization, and thus it is not very similar to real-time audio servers at all. Today NAS seems to see only very marginal use.

Enlightened Sound Daemon is one of the most used audio servers in unix, but its development has stopped some years ago. It does not offer low latency or synchronization. ARTSD is a somewhat more sophiscated server, supporting inter-application routing and low latency. However, possibly the most interesting current real-time audio server server project is Jack Audio Connection Kit.

7.1.1 JACK — Jack Audio Connection Kit

Jack is a kind of departure from existing Unix audio server designs, but it is also not the kind of monolithic shared object system commonly used in Windows/Mac professional audio. In fact it is somewhat of a hybrid of these two models, drawing inspiration from the shared object model, but not completely adopting it. It is also not designed to be a network transparent system. A study of Jack's design should be insightful, because it is fairly widely used by Linux audio applications, and does meet most of the goals it has set for itself, offering synchronous, low latency operation for audio clients.

Like X Window System, Jack contains more than just a server, including a client library that Jack clients can use to connect to server. Paul Davis, one of the main developers of Jack, describes it as a system that should be able to move "large quantities of data with very low latency and very importantly to have the exchange be sample synchronized."[6, 2:29–2:35] Jack uses a normalized 32-bit floating point representation for audio, where audio data should not be interleaved, but instead each client can send and receive multiple data streams. To ensure platform portability, Jack adheres to POSIX standards, and its API is specified in ANSI C. Jack servers can be run concurrently, but each server process is independent and not meant to cooperate with other Jack servers. Jack clients should be free to connect and disconnect with the server whenever they want to.

One of the main concerns of Jack developers is that the traditional Unix I/O model is possibly inadequate for real-time audio work. One problem with it is that it does

itself not maintain synchronous operation, which needs to be ensured by some other means. In addition, their design documentation refers to a problem of blocking system calls:

> There are numerous audio servers and application frameworks available, but nearly all them have serious problems when it comes to latency and bandwidth considerations. The basic problem is that all common mechanisms for process-to-process communication (IPC) involve possibly blocking system calls. This is especially true for network operations.[44]

In addition to IPC costs, running clients as separate processes causes context switching costs. These are made worse by unoptimal scheduling of client-server -systems, that may occur when the system scheduler is not aware of the correct scheduling patterns demanded by the audio system. Another problem affecting Jack's design is the GUI toolkit problem pointed out in section 3.5.1, which makes it impossible to run several X toolkits within a single process.

To solve these issues, Jack clients can be run as shared objects that are loaded into the server, or as separate processes. In order for this to work correctly, Jack mandates threaded programming model for audio clients. Quoting Paul Davis, each client "has at least one thread that is created and managed for it by Jack", and "that thread executes callbacks that are registered by the client, saying when a certain condition happens this is the code i want you to run."[6, 30:51–31:13] Callbacks are simply functions that client has associated with various events. The thread handled by Jack (meaning it can run as a part of Jack process, or it is "driven" by Jack by other means) runs with real-time priority, which also means it needs to be real-time safe. As is written in Jack Design Documentation,

> [the RT thread code] must be deterministic and not involve functions that might block for a long time. These functions are for example malloc, free, printf, pthread_mutex_lock, sleep, wait, poll, select, pthread_join, and pthread_cond_wait.[44]

In a typical client-server system, if a client makes a blocking system call (which may be invoked by a library function such as any of the listed above), the risk of the client missing its deadline increases. This does not necessarily mean that other clients would be affected. Jack's designers have taken a stance, that this should never happen, and proper functioning of the whole system relies on this. Jack also takes advantage of this requirement in the fact that, by taking the scheduling of

clients' audio threads into its own hands, it can ensure even more deterministic audio processing behaviour. Paul Davis has called this "userspace co-operative scheduling" [6, 28:19] and it works as follows.

Audio processing happens in a processing cycle, which repeats in very short intervals, depending on latency requirements. Thus, to achieve audio latency of three milliseconds, the processing loop should always take less than this time so that dropouts will not occur. During each cycle Jack executes all the audio threads of internal clients, meaning clients that have been loaded to Jack server's address space as shared objects.

In order to run external clients, Jack server sets up named pipes (FIFOs), that the server and clients use to send messages with each other. External clients use the Jack client library executing in the audio thread, and library code waits for a message from a FIFO. Once it gets it, a callback function to process audio is executed by the client library, and once it has finished processing it will send a wakeup message through another FIFO to another client or the server. Paul Davis explained this scheme in his presentation using the following words:

> [Jack server] writes a single byte into this FIFO which wakes up client N [and] because it is sleeping in poll on this FIFO, it calls this process function and it writes a single byte to the next FIFO and goes back to sleep. The write to this FIFO wakes up client N+1 which does exactly the same thing, which then writes to the FIFO, which wakes up [Jack server].[6, 28:41–29:04]

This means that Jack system has perfect control of the execution of the client audio threads. Jack can then serialize their scheduling so that it fits the paths that audio streams take from client to client. For example, if client X uses the output of client Y as its input, the Jack system knows this, and can schedule clients correctly (by letting Y execute before X). Furthermore, because the server and client audio threads run with real-time priority, applications running with normal priorities can not interfere with audio processing in Jack system. Assuming that all clients are implemented correctly, the benefit of this scheme is robust, low latency audio.

This description still does not cover the methods of actually passing audio data between server and clients. For internal clients this is not a problem: because the address space is shared, the server and client only need to negotiate the memory segments used for passing audio data. But to handle communication with external clients, Jack uses shared memory. The decision to use shared memory comes from the goal of achieving high efficiency, and using other mechanisms involve using the

kernel, which needs to copy data from userspace to kernel memory, and back.

7.1.2 Some remarks on Jack

It should probably be noted here that the author has never actually used Jack, and thus can only consider it on the basis of the few documents and a presentation by Paul Davis. The current implementation of Jack probably does very well what it intends to do. It may also be true that some details above and points presented here are wrong, especially because Jack is still evolving.

Nevertheless, some points can still be made. One consequence of Jack's design is that any applications whose jack-thread is incorrectly programmed, can inappropriately disrupt the whole Jack system. Thus, it does not allow any "graceful degradation" of audio quality (for example, by dropping a misbehaving client), but either works correctly or not. A more serious shortcoming is that it can not be distributed over a network: all clients need to be run locally. This reflects the primary purpose of Jack, which is mostly to connect together different music making applications, whereas network audio can be problematic when reaching for low latency.

Even though the theory behind Jack looks solid, it could be questioned whether all the design rationales presented in JACK Design Documentation[44] totally make sense. One thing claimed was that commonly used IPC mechanisms are problematic, because the system calls involved can block process execution, which is "especially true for network operations".

However, Jack itself uses blocking reads and writes with FIFOs to control scheduling, and it is difficult to see why passing audio data in this way would cause serious latency related problems. Jack uses shared memory to gain high efficiency, but that is a bandwidth related issue which should not drastically affect latency. System calls can also be used with non-blocking option set, so that applications can actually test if a call would actually block or not, and behave accordingly. Furthermore, even in the case that a system can not guarantee low latency for socket operations, there remains an option of fixing the system in this respect. It does not seem as if a socket-based programming model is inherently inadequate for the work that Jack does.

Jack's design is clearly a departure from traditional Unix server programming model, and finds inspiration from plugin-architecture used in some other audio systems. The technique to run external clients emulates the shared object model as far as is possible when clients run outside server address space. Choosing between these options has a tradeoff between efficiency and safety. The efficiency gains of

shared object model can be marginal depending on CPU architectures. Safety gains of the external client option — as it has been implemented in Jack — is likely to be marginal as well, because a single misbehaving client can disrupt the proper functioning of Jack system anyway.

7.2 Test servers and clients

Comparing the real-time performance of Jack and some other audio server applications would probably be an interesting experiment. Nevertheless, because application design issues have been a secondary concern in this thesis, and because operating system behaviour can drastically affect the performance of any such system, the primary focus has been given to the real-time performance characteristics of the operating system. It was considered that measuring this performance under various conditions, with different workloads and IPC methods, should be easier if test applications were written for this primary purpose.

Both the test server and the client were written to do some marginally useful real audio work. They create a simple audio stream loop, where audio recorded by some hardware device is first read by the server [see server source code at appendix A, line 352]. The server then routes it through each client by sending and then receiving the processed audio data, and then sends audio stream back to the device [line 113]. Test clients do some fake audio processing, thus simulating real workloads, but this does not in fact alter the audio stream in any meaningful way [see client source code at appendix B, line 172].

A crucial feature of this test system is that each client must perform synchronized with every other client, that is, the clients must progress in a single lockstep so that each process handles audio data that correspond to the same point in time. While the system should provide as low audio processing latency as is possible, that is not very useful if any client gets left behind. Other than this, there is no other kind of timing control: there is no mechanism for pausing clients, nor rewinding them, nor any other such a sophisticated control facility. In fact the applications have no notion of time at all, and instead rely on an implicit and simple mechanism for timing, by reading audio data from the audio hardware device as soon as it becomes available.

Both the server and the client have been written so that there is flexibility in the choice of audio blocksize, IPC facilities used for passing audio data, and routing topology [lines 73-90]. These parameters can be passed as arguments to the server process as it is executed. Clients will adapt to these options as necessary, after receiving information about them as a part of the connection protocol [server source,

line 304, and client source, lines 80-95]. The chosen audio blocksize will be used to set both the kernel level audio blocksize [server source, line 141] and the data blocksize used for communication between server and clients. The server and the client application can both use either sockets or *mmap()*-based shared memory for communication.

As far as routing topology is concerned, there are two choices. These are the same that were depicted in figure 5.1, covering common audio processing patterns. The server can be used in a parallel mode, where all clients are treated as equivalent and the recorded audio stream is sent to all clients, after which the received audio streams will be mixed together. Or a serial mode can be chosen instead, where data will pass each client on its turn, thus creating a kind of an audio processing pipeline. However, because no connections are created between clients, audio data will still be passed through the server at each phase of the processing loop.

The differences between socket- and shared memory -based communication were purposely made as small as possible, so that the benchmarks would measure their differences in efficiency instead of the quality of their implementation. In the server, each client has its own input and output buffers. For each client, the server always writes the audio input into the client's input buffer, and reads data from its output buffer [see "preprocessing operations" at lines 368, 392 and 397 in server code]. These could be optimized away in some cases, but are still done to ensure a fairer comparison between both IPC mechanisms. The difference between the IPC methods can then be reduced to a single call, which is either a socket related system call or *flock()* [see the implementations of *send_audio()* and *receive_audio()* at lines 409 and 421].

In the socket case, data is actually transmitted through a socketpair, where a file descriptor of the other endpoint is sent to each client through the socket connection [line 260]. We will now take a look at some of the decisions made in the shared memory implementation.

7.2.1 Shared memory issues

Because stream sockets (and socketpairs) have communication semantics that make a good fit for sending and receiving audio data, there is not much to argue about how they should be used. However, shared memory offers some more room on how it can be used for passing data between processes.

Effectively, the problem is about simulating stream-oriented communication with shared memory, but this is actually fairly easy with suitable shared memory inter-

faces and locking patterns. Test applications were written to use the *mmap()*-interface for shared memory instead of using the System V -interface. This choice is mostly due to the fact that *mmap()* is a more natural, or "native" interface for BSD systems.

A somewhat more controversial choice was that System V semaphores were also rejected, and instead also the means of synchronization to shared memory blocks was provided by filesystem operations. This means using dedicated files and advisory file locks with *flock(2)*[3] as semaphores.

This is actually something that a quote from Uresh Vahalia at page 88 warned against. On the other hand, there are no apparent reasons why a dedicated semaphore system call should be more efficient than an equivalent filesystem related call. It may very well be, but benchmarks might be necessary to come to that conclusion. Simply because a shared memory interface is filesystem based does not make it inefficient, and the same can be true with semaphores as well. After all, *flock()* does not induce any visible changes to the filesystem. Nevertheless, it should be kept in mind that that the locking technique may be badly chosen, and therefore the results may not be totally fair regarding shared memory,

While semaphores are required for controlling access to shared memory segments, it is not immediately clear how they should be used in our particular case. In the test applications, each client has its own input and output segments that the server has access to. These segments should be used so that first the server writes to the input segment, after which the client can read the input and write its output to the output segment, which can then be read by the server. Thus, the server and each client need to take turns in accessing shared memory segments. A decision was made that this turn-taking should rely on semaphores only.

The need for additional information about the state of progression, that is, which application should be the next one to proceed, means that it is not possible to have one-to-one mapping between semaphores and shared memory segments. A process can stop another process from progressing by holding a semaphore that the other process has requested. It can then allow it to proceed by releasing that semaphore. This is a good basis for establishing turn-taking, but unless a process is always holding at least one semaphore, it can not stop the progression of another process. Moreover, the next semaphore that a process tries to grab can not be the same it has previously released, because there are no guarantees that the other application has made any progress between these operations. This means that at least three semaphores are needed for establishing turn-taking.

This is what the test applications do in the shared memory case. For each client, the server creates three files that serve no other purpose than providing locks. At

every point in time, either the server or the client holds two locks, meaning that it can progress and modify data on the shared memory segment. The three locks are grabbed and released in a cycle, facilitating proper semaphore-based turn-taking [see lock cycling in server source, lines 376 and 399, and client source, line 146].

Each processing loop requires two system calls: one for an unlock operation after sent data has been written, and another for a lock operation after which received data be read. This is the equivalent of *write()* and *read()* system calls with sockets. Thus, the use of shared memory does not provide any efficiency benefit in the system call count, but the system call length should be shorter, because less work needs to be done. In an implementation such as this, shared memory has successfully been used to optimize away the data copy operations inherent in socket-based IPC.

7.3 Testing system real-time behaviour

The applications were tested on an OpenBSD 3.7 system, running on a computer whose exact hardware configuration can be found in appendix C.3. Without much difficulty, the test machine could support about 10 test clients, each passing through one CD-quality stereo audio stream. However, what is interesting is not how suitable some particular machine is for audio processing, but the patterns of system behaviour that various workloads reveal.

The primary concern with real-time audio work is not audio data bandwidth that any system can offer, but the latencies that it can achieve. The latency figures are also related to reliability levels, so that each particular audio processing latency can be guaranteed only with some particular probability. Reaching very low latencies occasionally can be easy, but sustaining such latencies over a long period of time without audio dropouts can be much more difficult, depending on other tasks that the system may need to complete.

Thus, the benchmarks done with test applications measure the audio latency as it travels from the audio device to the server, is routed through the clients, and then passed back to the hardware device. This is done by filling the kernel audio buffers before anything is read from the device, and then measuring the buffer fill levels as audio data is passed through the system. The output buffer levels will drop if the system can not maintain a constant stream of audio data flowing into the buffer, and their levels can be used to calculate the latency caused by audio processing. The maximum measurable latency is determined by the size of the audio buffer and used audio encoding, which is about $65536/2/2/44100 = 0.372$ seconds in our test cases.

Two small changes were made to the vanilla OpenBSD kernel, to reliably collect

data about audio playback buffer levels. Each time the audio device has played another block of audio, an interrupt handling routine is invoked, which is modified to print data about the playback buffer fill level. This data is read by `syslogd`, which writes it to the standard system messages log, wherefrom it will later be read and analyzed. As the process generates a lot of latency data at a fast pace, `syslogd` is not normally able to read data as soon as it is written to kernel message buffers. To ensure that no data gets lost in the collection process, the size of kernel message buffers is also raised from default 16 kilobytes to 4 megabytes. In fact, this means `syslogd` can perform its work after each test has finished. See appendix C.2 for the kernel patch that was applied to make these changes.

After some testing it turned out that there were also some other reliability related problems in this technique of measuring audio latency. One was that during heavy load, the audio server might not be able to read data from kernel audio recording buffers in time, leading to buffer overflows and thus recording errors. If these rare situations would not be taken into account, the amount of recorded audio data would be less than the amount needed for the playback data buffers, leading to permanently dropped playback levels and misread latency results. The server deals with this problem by creating an extra block of silence for every dropped block of audio data [server source, line 342].

However, the problem of permanently dropping playback levels would mysteriously persist even after this change. It is still uncertain what the real cause of this discrepancy is, but one possible explanation is that with some audio hardware the recording and playback sample rates may not match exactly. Note that the hardware used for tests was cheap consumer-grade equipment. Some errors during playback might also explain this phenomenon. In any case, this problem was "solved" in test server by generating an extra block of silence for each 512th block read [server code, line 324]. This should guarantee that there is always enough input data to keep the playback buffer full at all times. Most real applications should deal with problems such as this in some other way.

During the tests, both server and client processes were run with the closest equivalent of real-time priority in OpenBSD, that is, the nice level of -20, except in the tests that specifically test the effect of this priority adjustment. The server sets up its connection socket and files such as shared memory files into `/tmp/audiod`-directory. Because shared memory relies on *mmap()*, this directory is mounted as a memory file system, to eliminate any penalization that could occur by possible audio data writes to some disk device.

All tests are also run with some background stress, generated by the tool `Stress`,

that can be obtained from *http://weather.ou.edu/~apw/projects/stress/* . `Stress` can create a configurable amount of system load, by spawning processes to work on CPU, doing VM (virtual memory) related operations, writing data to hard drive (HDD), or forcing disk I/O through *sync(2)*[3] system call. The default background stress consists of four CPU workers, plus three workers, where each of these is dedicated to performing either VM, HDD, or I/O operations. This level of background stress is quite harsh and probably unrealistic in most use cases, but should be useful to find out distinctions between test cases, and the characteristics of system behaviour. The generated stress should also be similar to usual media processing workloads.

Five major tests were done, where the point of each test was to find data about how some particular parameters affect the latency behaviour of the system. To achieve this, each test consisted of four subtests, where the first one had default parameters, and the latter three had some test parameter values changed. Thus, the first subtest was the same one in each test, and their results should be very similar if the tests can be considered at all reliable. The default parameters were the background stress levels explained above, with real-time priorities, using parallel audio routing and sockets for passing audio between processes. Five clients were connected to the server, and a blocksize of 512 bytes was used to pass audio data between processes and kernel audio subsystem.

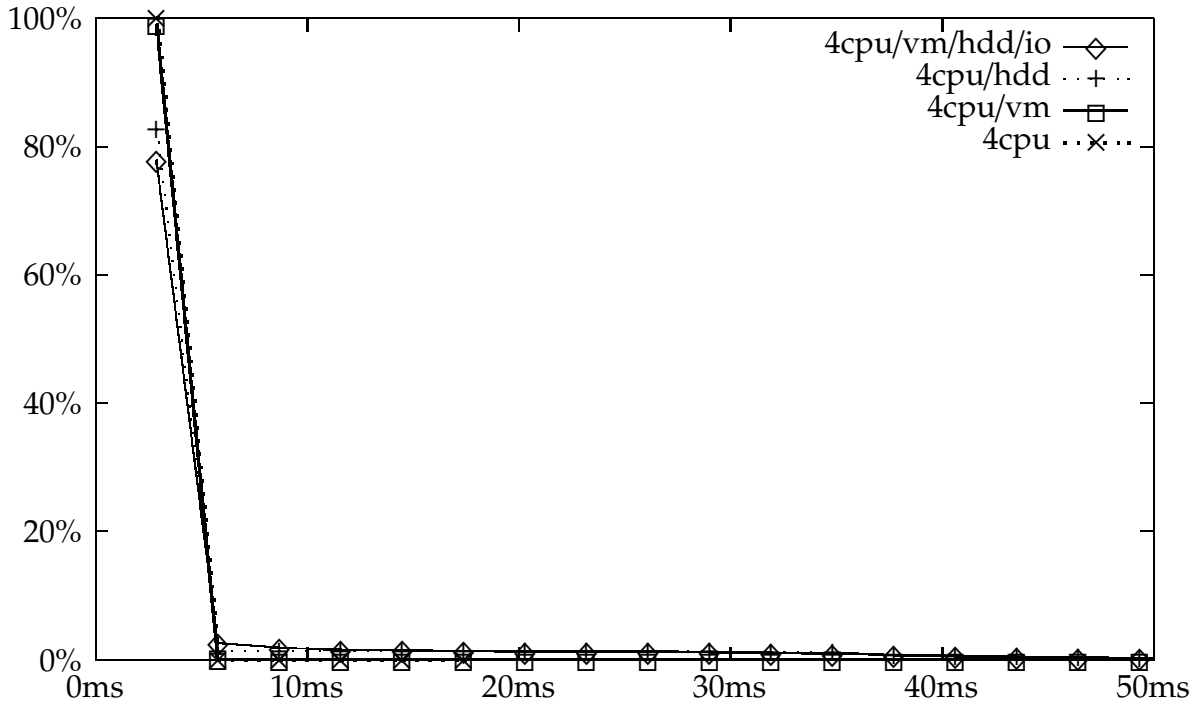
On each subtest, the test applications were run for five minutes. A small amount of the first and last samples were dropped out of the results, so that any anomalies that might happen during application startup and exit are not taken into the results. The results from each test are captured into a figure that shows a histogram of the latency behaviour of the client-server audio system over the test time. The X-axis of each figure represents the delay in milliseconds, that audio data blocks will see as they travel through the system. The Y-axis represents the probability of each discrete delay time.

Ideally, of course, the delay should be very low, with high probability, meaning that low latency could be guaranteed with good reliability. Note that the figures are not cumulative, that is, the probability of any delay time does not include the probabilities of all shorter delays. But now, let us see the figures, numbers and what is behind them.

7.3.1 The results

The first test done was about the effects of various background stress to the real-time performance of the client-server audio system. Ideally, the kernel should schedule

Figure 7.1: The effects of background stress



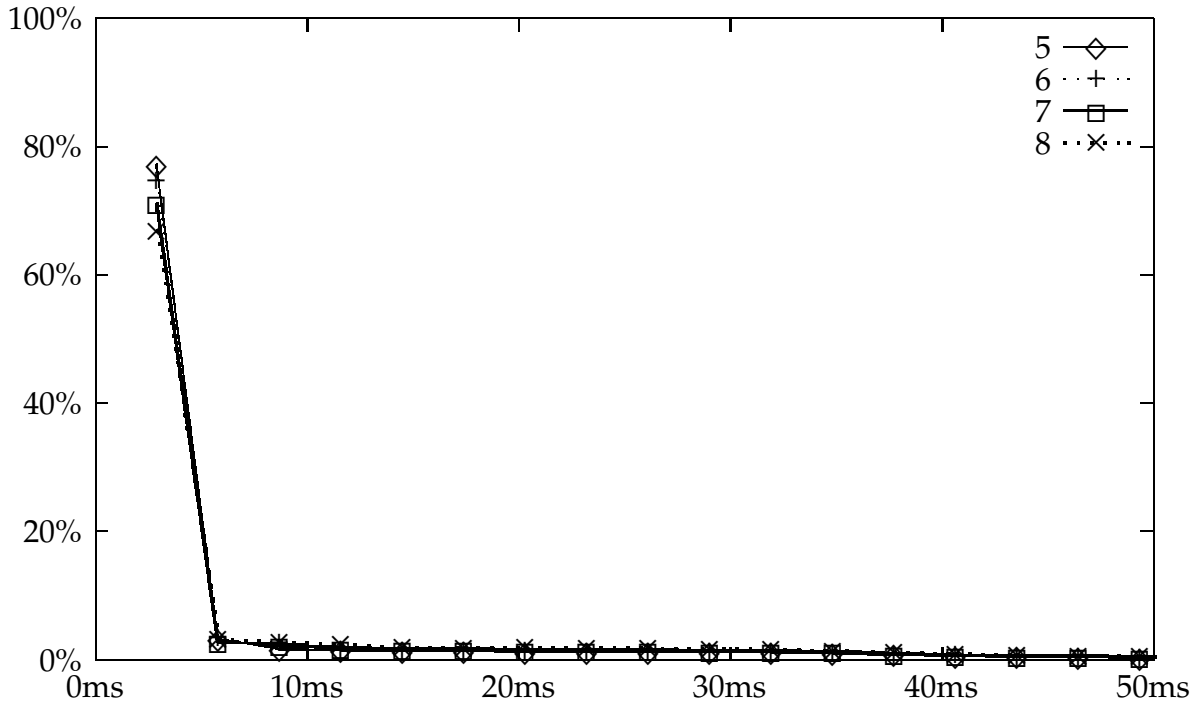
the real-time processes whenever they need to be run. However, in OpenBSD, the "real-time" priority still is a lower priority than any kernel-level priority. In case other processes make system calls, the processes can execute in-kernel with a higher priority, and real-time processes have to wait until calls complete.

Four subtests where made. One was with default parameters, and three others dropped certain types of stress load. In one case, only CPU and HDD stress was left, and in another, CPU stress co-existed with stress on virtual memory. On the four subtest, only four CPU workers were left as background stress. See figure 7.1.

The differences in the system's latency behaviour in these tests should mostly be attributed to the differences in the system call behaviour of background processes, and how lengthy calculations these calls can trigger. We can see that in the test where there are only four CPU stressing processes running, the latency behaviour is very close to ideal. This shows that a system such as OpenBSD, that has a non-preemptive kernel, but with some kind of support for real-time scheduling, can offer a very good real-time performance that is suitable for some real-time audio work. Running some CPU stressing work in the background does not significantly interfere with real-time audio work.

However, this is only the case if the background processes do not need to perform

Figure 7.2: Different number of clients



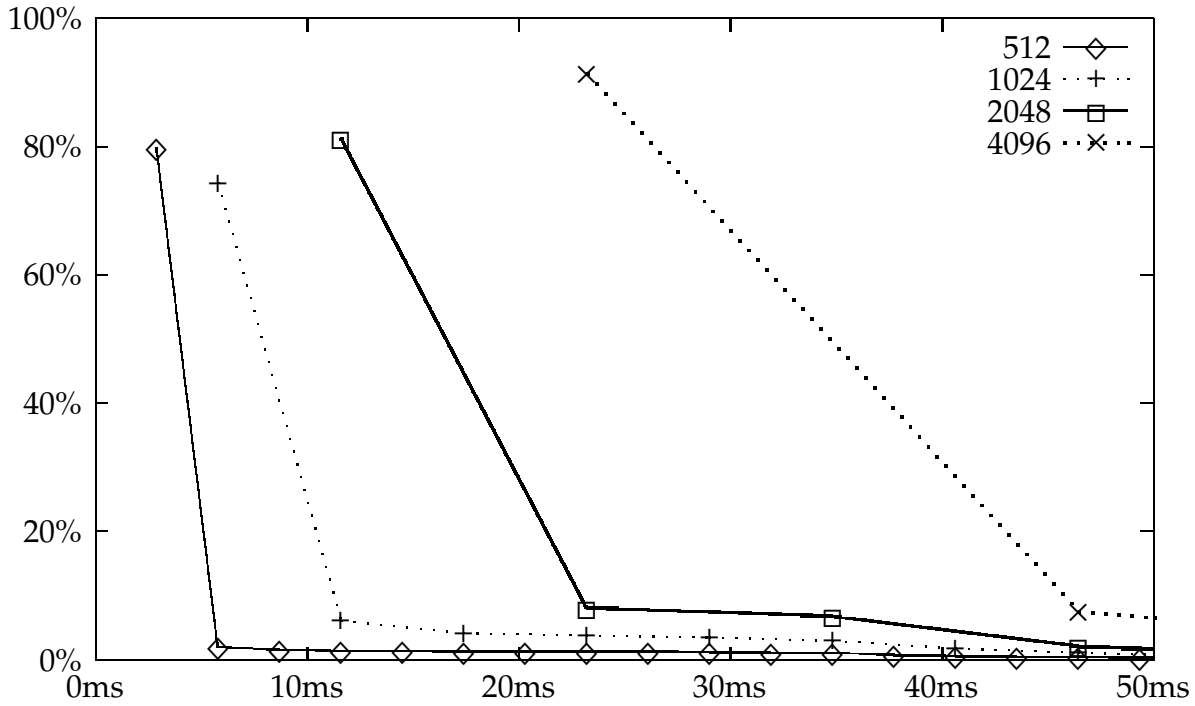
any system calls. If we introduce a single process that is constantly doing hard drive I/O, the achieved latencies do not look very ideal anymore. On the other hand, not all kernel activity has such serious effects, because activities on the kernel VM subsystem do not seem to affect real-time performance very much.

See figure 7.2 for the results of the second test. Here, the first subtest has the default value of five clients, but for the other subtests, the number of clients were raised to six, seven and eight. The figure mostly tells the expected: as you increase the number of clients, the audio processing load becomes greater, which makes it more difficult to sustain low latencies.

On the other hand, this does not necessarily occur. There is one thing about this that the figure does not tell. It is the fact that the latency might drop only once some particular number of clients is reached, because at that point the system simply can not handle the load it is supposed to carry. Then, when it drops, it does drop dramatically. However, in these tests the number of clients is kept so modest that in all cases the system can basically maintain its load, even though the latency guarantees drop as the number of clients increases.

On the third test, the audio data blocksize was changed for each subtest. When audio is passed in bigger chunks between the server and the kernel audio subsystem,

Figure 7.3: Audio data blocksizes

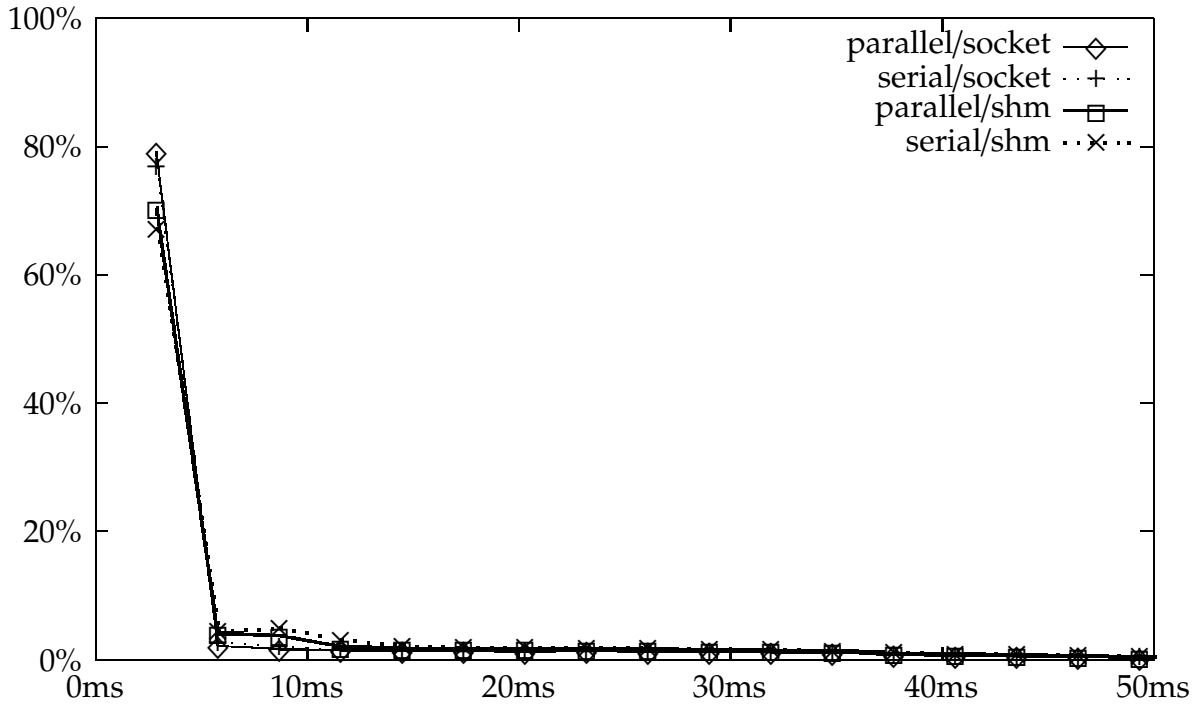


the audio latency should get higher as the blocksize increases. Figure 7.3 shows that this is what really happens: for example, when a blocksize of 2048 bytes is used, the audio processing latency can not drop below eleven milliseconds. Reaching a latency of about three milliseconds is only possible with blocksize of about 512 bytes or lower (assuming stereo, CD-quality audio).

The figure shows another interesting effect. Even though the test done with blocksize of 4096 bytes can not reach very low latencies, it appears that it can reach the lowest possible latency with a greater reliability than the tests with lower blocksizes. This is because the system can function more efficiently, as there is less system call and context switching overhead, thus reaching a higher data bandwidth, but at the expense of higher latency.

The case of blocksize of 1024 bytes should also be noted, as it performs surprisingly badly. Most likely a system using this blocksize gets interference from some system activity initiated by background stress processes. This suggests that there can not be such a thing as an optimal blocksize, because what is optimal depends on the type of background stress occurring in the system. An algorithm that periodically recalculates the used blocksize, depending on audio dropouts and required reliability levels, might offer a better solution than what any constant blocksize can

Figure 7.4: Sockets, shared memory, and audio routing topologies



provide.

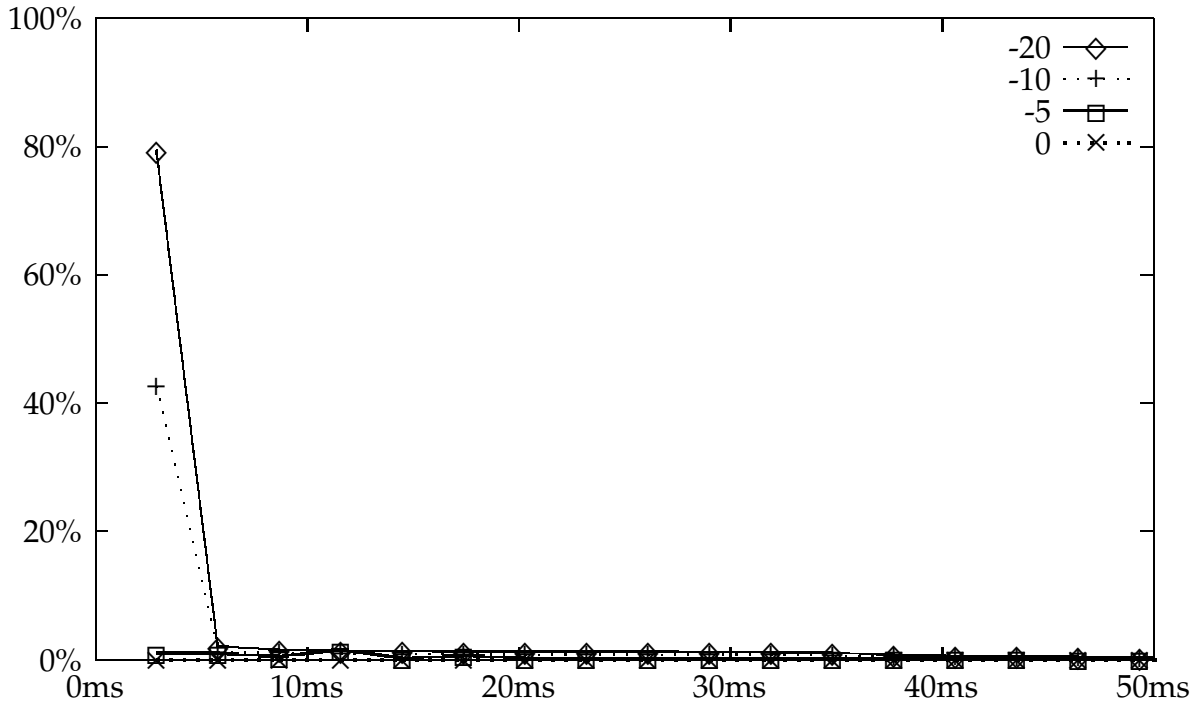
On the next test, the default case of socket-based IPC with parallel topology was contrasted with the use of shared memory and serial topologies. The results are in figure 7.4.

The difference between parallel and serial audio routing is that in parallel routing, the execution needs to step into server only twice during each audio processing loop: as audio is read from kernel and sent to all clients, and then when processed audio is received from clients, mixed and sent back to kernel. However, the system scheduler may possibly not schedule the applications in this optimal way, but instead will schedule the server more frequently during each loop.

However, with serial topology, a lot more context switching to server must occur, because it needs to pass audio from each client to another client. Implemented this way, it is not surprising that tests using serial routing will not achieve as low latencies than in parallel case.

Another notable issue is that when applications use shared memory, they appear to not perform as well as when they rely on socket-based IPC. This can only mean that that the overhead of file locking operations must be higher than passing small amount of data through a socketpair between processes. Possibly if System V

Figure 7.5: unniced



semaphores were used instead, the results would be different. On the other hand, it is encouraging to see from the overall results that socket-based IPC can also maintain very low latencies, and their overhead may be rather negligible.

Note that what these tests measure is not the data bandwidths that some particular IPC facilities might offer. In fact, it still might be true that shared memory with *flock()* provides a higher data bandwidth. Yet when it comes to latency (which is more essential for real-time work), sockets appear to be a more ideal solution.

The purpose of the last test was to find out the effect of nice level of -20 to the real-time performance of the client-server audio system. In three subtests, the niceness levels of the audio system were dropped to -10, -5, and 0. Again, see figure 7.5 for the results.

The latter two subtests almost do not show up in the figure at all, because the applications are not receiving enough CPU time to pass the audio through them at all. Note that this is only because the background stress processes run on same priority as the applications — if there was no background stress at all, the performance should be very close or equal to the real-time case. Thus, it is clear that running applications with elevated priority levels can give a very significant boost to the real-time performance. This boost appears tunable: in the subtests where a nice-

value of -10 is used, the performance is better than if no adjustment is done, but still considerable worse than with the extreme setting of -20.

While looking at the latency figures can be instructive, from them it is difficult to estimate the reliability with which various latencies can be sustained. Moreover, audio applications typically demand reliability levels of 99,9%, and even higher, so that audio dropouts do not occur as frequently as to be disturbing.

Table 7.1 shows some latencies that can be sustained with a probability of 90%, 99% or 99,9% at each test case. The latencies with 99,9% are probably not very reliable, because of a low number of samples collected. This can be seen from the fact that the first subtest of each case should give very similar numbers, but this is can not be said about the values in the last table column.

Nevertheless, the table shows clearly how OpenBSD is not a real-time system, because reliably sustainable latencies can be above three hundred milliseconds when there is some particular kinds of background stress. On the other hand, it is interesting to see that audio latency of three milliseconds could be sustained with a fairly good reliability, in the case where the background stress did not invoke any system activity.

From the table we can also find out some other information that could not be deciphered from the figures. One is that, while the virtual memory activity in kernel did not seem to affect real-time performance very much, it turns out that it can occasionally jump into lengthy computations and cause in some serious audio dropouts over a longer period of time.

From these results, the logical next step would probably involve collecting data about kernel activities that are the main culprits of bad system real-time operation. This could be done by connecting bad latencies with kernel code through the use of kernel profiling. Doing such experiments, however, is left for more adventurous folks.

Table 7.1: Audio latencies and their probabilistic guarantees

test	parameters	latency guaranteed with probability of		
		90%	99%	99,9%
stressed	4cpu/vm/hdd/io	26.12ms	98.68ms	322.18ms
	4cpu/hdd	20.32ms	43.54ms	49.34ms
	4cpu/vm	2.90ms	2.90ms	238.00ms
	4cpu	2.90ms	2.90ms	2.90ms
clients	5	26.12ms	78.37ms	333.79ms
	6	29.02ms	78.37ms	171.25ms
	7	34.83ms	235.10ms	368.62ms
	8	37.73ms	-	-
blocksize	512 bytes	26.12ms	69.66ms	104.49ms
	1024 bytes	29.02ms	63.85ms	92.88ms
	2048 bytes	34.83ms	69.66ms	92.88ms
	4096 bytes	23.22ms	69.66ms	116.10ms
ipc/topology	parallel/socket	26.12ms	66.76ms	92.88ms
	serial/socket	26.12ms	72.56ms	133.51ms
	parallel/shm	29.02ms	69.66ms	95.78ms
	serial/shm	29.02ms	69.66ms	95.78ms
unniced	-20	26.12ms	69.66ms	95.78ms
	-10	-	-	-
	-5	-	-	-
	0	-	-	-

8 Conclusion

I suppose the purpose of a good conclusion is to do two things. First, after spending pages and pages of explaining the mess that the whole thing really is, one should show that it was not a mess after all, and how everything really does make sense. And second, to give good guidelines to all practical system implementors, based on fundamental theoretical insights.

Yet, during the time I have been writing this thesis, I feel I have been coming back to one of the first things I wrote into the source files, although commented out:

> So instead of doing what they really want to do, which is to design beautiful software, hackers in universities and research labs feel they ought to be writing research papers. In the best case, the papers are just a formality. Hackers write cool software, and then write a paper about it, and the paper becomes a proxy for the achievement represented by the software. But often this mismatch causes problems. It's easy to drift away from building beautiful things toward building ugly things that make more suitable subjects for research papers.[12]

And that is how things really are. Not just for me, but between computer science and software engineering in general. Because, with many problems, one can gain a much better understanding of the problem by actually trying to solve it, instead of trying to analyze and theoretize it. In some fields, there may be no difference between these two. Yet with software, we are also creating things that, to a large extent, live their lives outside of what our theories can say about them.

This is why I felt it was very important to actually try writing a tiny distributed audio system. Not simply because it can be used for some fancy benchmarks, but because I suspected that it could teach me a lot more about real-time audio servers than any research papers might do. My experience confirms that this was very much true. Thus, at least with small things like audio servers, the fastest way to enlightenment appears to be building them using several different approaches, and then seeing how they compare.

My fear is that the same is true with larger things also, such as operating systems. One just have to build many to see how they compare. The only problem with larger things is not that it takes enormous efforts to actually build them. The problem is

also that, to a large extent, the value of software technologies is not determined only by their inherent qualities, but by their relationship to other technologies, and the extent we place our dependance on them.

For these reasons, when dealing with large things such as operating systems, it can be very comforting to believe that design decisions made once have to be good, because they have been *proven* to work. That may be, but it is also that proving some other designs to be better may be next to impossible, because of the extent of efforts needed to implement large systems. Simply witness the extent of problems faced when redesigning Unix to support real-time applications, which was considered in chapter 4.

Thus, it appears that we can not completely trust neither science nor engineering on these matters. Rather, it must be a process of letting science correct engineering when it can, while keeping track of the reasons for the decisions made in building software systems. And this must sometimes mean challenging even ideas that almost everyone have accepted as truth — accepted, because it may be too daunting to acknowledge they are false:

> When communication with other people becomes the primary source of new ideas, there is a danger that we receive "destilled perception", where it is no longer possible to know what was rejected or retained to arrive at it, nor why. Given that we are moving into an Information Age where communication of ideas takes an ever bigger share of our new ideas, losing track of the rejected data is a major problem. I do not believe humans are prepared to handle the lack of rejected data. That is, the validity of our ideas rests on knowledge of the information that argues for *and against* it. If no such knowledge or information exists, our ideas are *not* valid. In particular, the lack of verifiable, rejected data, leaves us no means to examine the conditions on which it was rejected. [23]

And that was the conclusion. Discomforting, isn't it?

9 References

[1] John Baldwin: Locking in the Multithreaded FreeBSD kernel. BSDCon 2002, at *http://www.usenix.org/publications/library/proceedings/bsdcon02/baldwin.html*

[2] Scott A. Brandt, Scott Banachowski, Caixue Lin, Timothy Bisson: Dynamic Integrated Scheduling of Hard Real-Time, Soft Real-Time and Non-Real-Time Processes. 24th IEEE International Real-Time Systems Symposium, not available at *http://csdl.computer.org/comp/proceedings/rtss/2003/2044/00/20440396abs.htm*

[3] BSD Programmer's Manual, for 4.4 release, available at *http://www.freebsd.org/cgi/man.cgi?sektion=2&manpath=4.4BSD+Lite2*

[4] Alan Burns, Andy Wellings: Real-time systems and their programming languages

[5] George Coulouris, Jean Dollimore, Tim Kindberg: Distributed Systems — Concepts and Design

[6] Paul Davis: JACK's design, a presentation at Karlsruhe in the spring of 2003. *http://www.linuxdj.com/audio/lad/contrib/zkm_meeting_2003/recordings/paul_davis-jack.ogg*

[7] Jeremy Elson: FUSD — a Linux Framework for User-Space Devices. Available at *http://www.circlemud.org/~jelson/software/fusd/*

[8] Bryan Ford and Sai Susarla: CPU Inheritance Scheduling. USENIX 2nd Symposium on OS Design and Implementation, available at *http://www.usenix.org/publications/library/proceedings/osdi96/ford.html*

[9] FSMLabs: RTCoreBSD — Technology. See *http://www.fsmlabs.com/rtcorebsd.html*

[10] Jim Fulton, Greg Renda: The Network Audio System — Make Your Applications Sing (As Well As Dance)! Available at *http://radscan.com/nas/docs/xcon94paper.ps*

[11] Pawan Goyal, Xingang Guo and Harrick M. Vin: A Hierarchical CPU Scheduler for Multimedia Operating Systems. USENIX 2nd Symposium on OS Design and Implementation, available at *http://www.usenix.org/publications/library/proceedings/osdi96/vin.html*

[12] Paul Graham: Hackers and Painters. *http://www.paulgraham.com/hp.html*

[13] David Ingram: Soft Real Time Scheduling for General Purpose Client-Server Systems. Available at *http://www.srcf.ucam.org/~dmi1000/papers/hotos.pdf*

[14] Rohit Jain, Christopher J. Hughes, Sarita V. Adve: Soft Real-Time Scheduling on Simultaneous Multithreaded Processors. Proceedings of the 23rd IEEE International Real-Time Systems Symposium, available at *http://www.cs.uiuc.edu/rsim/Pubs/jain-rtss02.ps.gz*

[15] Jeffrey M. Hsu: The DragonFlyBSD Operating System. AsiaBSDCon 2004, available at *http://www.dragonflybsd.org/docs/pdfs/dragonflybsd.asiabsdcon04.pdf*

[16] S. Khanna, M. Sebree, J. Zolnowsky: Realtime Scheduling in SunOS 5.0. USENIX Association Conference Proceedings, January 1992, also available at *http://citeseer.ist.psu.edu/khanna92realtime.html*

[17] Jonathan Lemon: Kqueue: A generic and scalable event notification facility. BSDCon 2000, available at *http://people.freebsd.org/~jlemon/papers/kqueue.ps*

[18] Jochen Liedtke, Hermann Härtig, Michael Hohmuth, Sebastian Schönberg, Jean Wolter: The Performance of μ-Kernel-Based Systems. Proceedings of the 16th ACM Symposium on Operating System Principles, available at *http://l4ka.org/publications/paper.php?docid=650*

[19] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, John S. Quarterman: The Design and Implementation of the 4.4BSD Operating System

[20] Marshall Kirk McKusick, George V. Neville-Neil: The Design and Implementation of the FreeBSD Operating System

[21] Jim Mauro, Richard McDougall: Solaris Internals — Core Kernel Architecture

[22] Larry McVoy, Carl Staelin: lmbench: Portable tools for performance analysis. Available at *http://www.bitmover.com/lmbench/lmbench-usenix.ps.gz*

[23] Erik Naggum's Ideas and Principles — Perception, Abstraction and Communication. Available at *http://naggum.no/erik/perception.html*

[24] The Open Group: The Single UNIX Specification, Version 3. Available at *http://www.unix.org/version3/*

[25] OpenBSD Kernel Manual, for 3.7 release, available at *http://www.openbsd.org/cgi-bin/man.cgi?sektion=9&manpath=OpenBSD+3.7*

[26] OpenBSD Programmer's Manual, for 3.7 release, available at *http://www.openbsd.org/cgi-bin/man.cgi?sektion=2&manpath=OpenBSD+3.7*

[27] OpenBSD scheduler modification patch, applied to source tree in 15.8.1999 (and taken from NetBSD) to file `src/sys/kern/kern_synch.c`. Available at *http://www.openbsd.org/cgi-bin/cvsweb/src/sys/kern/kern_synch.c.diff?r1=1.15&r2=1.16*

[28] OpenBSD source code, for release 3.7, available at *http://www.openbsd.org/cgi-bin/cvsweb/?only_with_tag=OPENBSD_3_7_BASE*

[29] Open Sound System, 4Front Technologies. Available at *http://www.opensound.com/oss.html*

[30] Keith Packard: Efficiently Scheduling X Clients. USENIX Annual Conference, Freenix Session — June 2000. Available at *http://www.usenix.org/publications/library/proceedings/usenix2000/freenix/packardefficient.html*

[31] Seong Rak Rim, Yoo Kun Cho: Message-based Microkernel for Real-time System. Available at *http://intl.ieeexplore.ieee.org/xpl/abs_free.jsp?arNumber=217498*

[32] Eric Steven Raymond: The Art of Unix Programming. Available at *http://www.catb.org/~esr/writings/taoup/html/*

[33] D. M. Ritchie and K. Thompson: The UNIX Time-Sharing System, Unix Programmer's Manual, Seventh Edition, Volume 2A. Available at *http://plan9.bell-labs.com/7thEdMan/v7vol2a.pdf*

[34] Jeff Roberson: ULE: A Modern Scheduler for FreeBSD. BSDCon '03, available at *http://www.usenix.org/publications/library/proceedings/bsdcon03/tech/roberson.html*

[35] J.H. Saltzer, D.P. Reed and D.D. Clark: End-to-End Arguments in System Design. Available at *http://www.reed.com/Papers/EndtoEnd.html*

[36] Curt Schimmel: UNIX Systems for Modern Architectures

[37] Stuart Sechrest: An Introductory 4.4BSD Interprocess Communication Tutorial. Computer Science Research Group, University of California, Berkeley. Available at *http://docs.freebsd.org/44doc/psd/20.ipctut/paper.ps.gz*

[38] Sound On Sound: Mind the Gap — Dealing With Computer Audio Latency. Available at *http://www.soundonsound.com/sos/apr99/articles/letency.htm*

[39] Stevens, W. Richard: TCP/IP Illustrated, Volume 1 — The Protocols

[40] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, C. Greg Plaxton: A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems. IEEE Real-Time Systems Symposium, 1996. Available at *http://www.cs.unc.edu/~jeffay/papers/RTSS-96a.pdf*

[41] Jon "Hannibal" Stokes: The PlayStation2 vs. the PC: a system-level comparison of two 3D platforms. Available at *http://arstechnica.com/articles/paedia/cpu/ps2vspc.ars/*

[42] Ray Tice, Mark Welch: The Broadway Audio System. Available at *ftp://ftp.x.org/contrib/audio/Xaudio/xtech96/xtech96.html*

[43] Uresh Vahalia: Unix Internals — The New Frontiers

[44] Kai Vehmanen, Andy Wingo, Paul Davis: JACK Design Documentation. Available at *http://jackit.sourceforge.net/docs/design/*

[45] Clark Williams, Red Hat, Inc.: Linux Scheduler Latency. Available at *http://www.linuxdevices.com/files/article027/rh-rtpaper.pdf*

A Server source code

```
#include <sys/queue.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/audioio.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <sys/uio.h>
#include <stdlib.h>                                                    10
#include <unistd.h>
#include <limits.h>
#include <err.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>
#include <util.h>

#include "common.h"                                                    20

#define AUDEVICE "/dev/sound"
#define MIXDIVIDE 8

struct client {
        enum ipc_used ipcstyle;
        char dir[PATH_MAX];
        FILE *control, *audio;
        int mmapfds[MMAPFDCOUNT], semfds[SEMFDCOUNT];
        short *in, *out;                                               30
        LIST_ENTRY(client) link;
};
LIST_HEAD(clients, client);

FILE *open_audevice(size_t *, int *);
int create_listen_sock(void);
int accept_clients(int, size_t, struct clients *, enum ipc_used);
struct client * newclient(int, size_t, enum ipc_used);
void read_audioblock(int *, short *, size_t, FILE *);
void parallel_connections(struct clients *, short *, short *, size_t);  40
void serial_connections(struct clients *, short *, short *, size_t);
void preprocess_audio(short *, short *, size_t);
void send_audio(struct client *, int, size_t);
void receive_audio(struct client *, int, size_t);
void mix_clientoutputs(short *, struct clients *, size_t);
void send_fds(int, int *, int);
```

```
int
main(int argc, char **argv)
{
        FILE *audev;
        struct clients clist;
        enum ipc_used ipcstyle;
        enum { PARALLEL, SERIAL } flowstyle;
        short *in, *out;
        int errorblocks, l_sock, ch;
        size_t blksize;
        extern char *malloc_options;
        const char *errstr;
        void (*handle_connections)(struct clients *, short *, short *, size_t);

        if (mlockall(MCL_CURRENT|MCL_FUTURE) < 0)
                err(1, "locking process pages to physical memory failed");

        malloc_options = "X";
        LIST_INIT(&clist);

        /* set default values */
        ipcstyle = SOCKET;
        flowstyle = PARALLEL;
        blksize = AUBLKSIZE;

        while ((ch = getopt(argc, argv, "b:ms")) != -1) {
                switch (ch) {
                case 'b':
                        blksize = strtonum(optarg, 1, BLKSIZE_MAX, &errstr);
                        if (errstr)
                                errx(1, "blocksize %s: %s", optarg, errstr);
                        break;
                case 'm':
                        ipcstyle = SHAREDMEMORY;
                        break;
                case 's':
                        flowstyle = SERIAL;
                        break;
                case '?':
                default:
                        errx(1, "incorrect arguments");
                }
        }

        handle_connections = flowstyle == PARALLEL ?
                parallel_connections : serial_connections;

        l_sock = create_listen_sock();
        audev = open_audevice(&blksize, &errorblocks);
        in = malloc(blksize);
        out = malloc(blksize);
```

```
	for (;;) {
		int delayed;
		delayed = accept_clients(l_sock, blksize, &clist, ipcstyle);
		if (delayed) {
			/* XXX workaround ''a delayed read'' bug in audio(4),
			 * XXX simply flushing won't work.
			 * XXX blksize might even change */
			fclose(audev);
			audev = open_audevice(&blksize, &errorblocks);
		}
		read_audioblock(&errorblocks, in, blksize, audev);
		bzero(out, blksize);
		handle_connections(&clist, in, out, blksize);
		if (fwrite(out, blksize, 1, audev) <= 0)
			err(1, "writing to audio device");
	}

	/* NOTREACHED */
}

FILE *
open_audevice(size_t *blksize, int *errorblocks)
{
	FILE *audev;
	struct audio_info auinfo;
	int audevfd, i;
	void *tmpbuf;

	if ((audevfd = open(AUDEVICE, O_RDWR, 0)) < 0)
		err(1, "open %s", AUDEVICE);

	if (ioctl(audevfd, AUDIO_GETINFO, &auinfo) < 0)
		err(1, "ioctl: info from %s", AUDEVICE);

	AUDIO_INITINFO(&auinfo);
	auinfo.mode = AUMODE_PLAY|AUMODE_RECORD;
	auinfo.play.sample_rate = auinfo.record.sample_rate = 44100;
	auinfo.play.channels = auinfo.record.channels = 2;
	auinfo.play.precision = auinfo.record.precision = 16;
	auinfo.play.encoding = auinfo.record.encoding = AUDIO_ENCODING_SLINEAR;

	auinfo.blocksize = *blksize;
	auinfo.hiwat = auinfo.play.buffer_size / auinfo.blocksize;
	auinfo.lowat = auinfo.hiwat - 1;	/* should be typical for RT */

	if (ioctl(audevfd, AUDIO_SETINFO, &auinfo) < 0)
		err(1, "ioctl: setting %s parameters", AUDEVICE);
	i = 1;
	if (ioctl(audevfd, AUDIO_SETFD, &i) < 0)
		err(1, "ioctl: setting %s full duplex", AUDEVICE);
	if (ioctl(audevfd, AUDIO_FLUSH) < 0)
		err(1, "ioctl: flushing %s", AUDEVICE);
```

```
        /* check out if hardware prefers some other blocksize, and return it */
        if (ioctl(audevfd, AUDIO_GETINFO, &auinfo) < 0)
                err(1, "ioctl: info from %s", AUDEVICE);

        *blksize = auinfo.blocksize;

        if ((audev = fdopen(audevfd, "r+")) < 0)
                err(1, "opening %s stream", AUDEVICE);                          160
        if (setvbuf(audev, NULL, _IOFBF, *blksize) != 0)
                errx(1, "could not set audio device blocksize");

        /* fill the playback buffer, so we get better latency estimates */

        tmpbuf = malloc(auinfo.play.buffer_size);
        if (fread(tmpbuf, auinfo.play.buffer_size, 1, audev) <= 0
            || fwrite(tmpbuf, auinfo.play.buffer_size, 1, audev) <= 0)
                err(1, "could not fill playback audio buffer");
        free(tmpbuf);                                                           170

        *errorblocks = 0;

        return audev;
}

int
create_listen_sock(void)
{
        struct sockaddr_un server;                                              180
        int l_sock;

        l_sock = socket(AF_UNIX, SOCK_STREAM, 0);
        if (l_sock < 0)
                err(1, "socket");

        server.sun_family = AF_UNIX;
        strlcpy(server.sun_path, ACCEPTSOCKET, sizeof(server.sun_path));

        unlink(ACCEPTSOCKET);                                                   190
        mkdir(AUDIODDIR, 0700);
        if (bind(l_sock, (struct sockaddr *) &server,
            sizeof(struct sockaddr_un)) < 0)
                err(1, "bind");

        if (listen(l_sock, 1) < 0)
                err(1, "listen");

        return l_sock;
}                                                                               200

int
accept_clients(int l_sock, size_t blksize, struct clients *clist,
               enum ipc_used ipcstyle)
{
```

```
        struct client *ac;
        struct timeval tv, *to;
        fd_set read_fds;
        int delayed, s;
                                                                        210
        FD_ZERO(&read_fds);
        FD_SET(l_sock, &read_fds);

        tv.tv_sec = tv.tv_usec = 0;
        if (LIST_EMPTY(clist)) {
                to = NULL;              /* block if no clients */
                delayed = 1;
        } else {
                to = &tv;
                delayed = 0;                                            220
        }

        while (s = select(FD_SETSIZE, &read_fds, NULL, NULL, to)) {
                if (s < 0)
                        err(1, "select for new clients");
                ac = newclient(l_sock, blksize, ipcstyle);
                if (ac) {
                        LIST_INSERT_HEAD(clist, ac, link);
                        to = &tv;       /* we have a client, only poll now */
                }                                                       230
        }

        return delayed;
}

struct client *
newclient(int l_sock, size_t blksize, enum ipc_used ipcstyle)
{
        struct client *ac;
        int audiosckts[2], ctrlsckt;                                    240
        short *tmpbuf;
        char dir[PATH_MAX], *line;
        int i;

        ac = malloc(sizeof(struct client));
        ac->ipcstyle = ipcstyle;

        /* create client directory for shared memory and semaphore files */

        strlcpy(ac->dir, (AUDIODDIR "/client.XXXXXXXXXX"), sizeof(ac->dir));    250
        if (mkdtemp(ac->dir) == NULL)
                err(1, "mkdtemp");

        /* create control and audio sockets, and corresponding streams */

        if ((ctrlsckt = accept(l_sock, NULL, NULL)) < 0)
                err(1, "accept");
        if (socketpair(AF_UNIX, SOCK_STREAM, 0, audiosckts) < 0)
```

```
        err(1, "socketpair");
send_fds(ctrlsckt, &audiosckts[1], 1);                                  260
close(audiosckts[1]);
if ((ac->control = fdopen(ctrlsckt, "r+")) == NULL)
        err(1, "fdopen control stream");
if (setvbuf(ac->control, NULL, _IOLBF, 0) != 0)
        errx(1, "could not set control stream to linebuffering mode");
if ((ac->audio = fdopen(audiosckts[0], "r+")) == NULL)
        err(1, "fdopen audio stream");
if (setvbuf(ac->audio, NULL, _IOFBF, blksize) != 0)
        errx(1, "could not set audio stream blocksize");
                                                                        270
/* set up shared memory files */

tmpbuf = calloc(blksize, 1);
for (i = 0; i < MMAPFDCOUNT; i++) {
        snprintf(dir, sizeof(dir), "%s/shm.%d", ac->dir, i);
        if ((ac->mmapfds[i] =
            open(dir, O_CREAT|O_EXCL|O_RDWR, 0644)) < 0) /* XXX */
                err(1, "open a shared memory file");
        if (write(ac->mmapfds[i], tmpbuf, blksize) < 0)
                err(1, "could not initialize a shared memory file");    280
}
ac->in = mmap(0, blksize, PROT_READ|PROT_WRITE, MAP_SHARED,
        ac->mmapfds[MMAPINPUT], 0);
ac->out = mmap(0, blksize, PROT_READ|PROT_WRITE, MAP_SHARED,
        ac->mmapfds[MMAPOUTPUT], 0);
if (ac->in == MAP_FAILED || ac->out == MAP_FAILED)
        err(1, "mmap");
free(tmpbuf);

/* set up semaphore files */                                            290

for (i = SEMONE; i < SEMFDCOUNT; i++) {
        snprintf(dir, sizeof(dir), "%s/sem.%d", ac->dir, i);
        if ((ac->semfds[i] =
            open(dir, O_CREAT|O_EXCL|O_RDWR, 0644)) < 0)
                err(1, "open a semaphore file");
        if (i == SEMONE || i == SEMTHREE)
                if (flock(ac->semfds[i], LOCK_EX) < 0)
                        err(1, "flock");
}                                                                       300

/* send protocol information */

fprintf(ac->control, "%s\n%d\n%s\n",
        ipcstyle == SOCKET ? PROT_SOCKET : PROT_SHM, blksize, ac->dir);

/* wait for confirmation that client is ready */

if ((line = fparseln(ac->control, NULL, NULL, NULL, 0)) == NULL)
        err(1, "fparseln");    /* XXX should not wait indefinitely */   310
if (strcmp(line, READYMSG) != 0)
```

```
                errx(1, "problem connecting a client");
        free(line);

        return ac;
}

void
read_audioblock(int *errorblocks, short *in, size_t blksize, FILE *audev)
{                                                                          320
        static int blockcount = 0;
        int recdrops, i;

        if (++blockcount % 512 == 0) {
                /*
                 * XXX for some mysterious reason, we appear to get less
                 * XXX recorded data than we need for playback.
                 * XXX one possible explanation is that some audio devices
                 * XXX may have slight mismatches between playback
                 * XXX and recording sampling rates. in any case,          330
                 * XXX as this is really bad thing for us, we create
                 * XXX an extra audio block for every 512 audio blocks.
                 * XXX nasty, and leads to extra recording errors,
                 * XXX but must be done for now
                 */
                bzero(in, blksize);
                return;
        }

        if (ioctl(fileno(audev), AUDIO_RERROR, &recdrops) < 0)             340
                err(1, "determining recording drops");
        if (recdrops / blksize > *errorblocks) {
                /*
                 * if some recording errors have occurred (samples
                 * are not read from buffer in time), we don't read
                 * from audio device but instead make a block of zero bytes
                 */
                (*errorblocks)++;
                bzero(in, blksize);
        } else {                                                           350
                /* situation normal */
                if (fread(in, blksize, 1, audev) <= 0)
                        err(1, "reading from audio device");
        }
}

void
parallel_connections(struct clients *clist, short *in, short *out,
                     size_t blksize)
{                                                                          360
        struct client *ac;
        static int holdlock = SEMONE, lock = SEMTWO, unlock = SEMTHREE, tmp;

        /* XXX we assume the clients work correctly.
```

```
         * XXX there should also be a mechanism for dropping them */

        LIST_FOREACH(ac, clist, link) {
                preprocess_audio(ac->in, in, blksize);
                send_audio(ac, unlock, blksize);
        }
        LIST_FOREACH(ac, clist, link) {
                receive_audio(ac, lock, blksize);
        }
        mix_clientoutputs(out, clist, blksize);

        tmp = holdlock; holdlock = lock; lock = unlock; unlock = tmp;
}

void
serial_connections(struct clients *clist, short *in, short *out,
                   size_t blksize)
{
        struct client *ac;
        static int holdlock = SEMONE, lock = SEMTWO, unlock = SEMTHREE, tmp;
        short *pbuf;

        /* XXX we assume the clients work correctly.
         * XXX there should also be a mechanism for dropping them */

        pbuf = in;
        LIST_FOREACH(ac, clist, link) {
                preprocess_audio(ac->in, pbuf, blksize);
                send_audio(ac, unlock, blksize);
                receive_audio(ac, lock, blksize);
                pbuf = ac->out;
        }
        preprocess_audio(out, pbuf, blksize);

        tmp = holdlock; holdlock = lock; lock = unlock; unlock = tmp;
}

void
preprocess_audio(short *out, short *in, size_t blksize)
{
        memcpy(out, in, blksize);
}

void
send_audio(struct client *ac, int unlock, size_t blksize)
{
        if (ac->ipcstyle == SOCKET) {
                if (fwrite(ac->in, blksize, 1, ac->audio) <= 0)
                        errx(1, "write to client");
        } else {
                if (flock(ac->semfds[unlock], LOCK_UN) < 0)
                        err(1, "flock (LOCK_UN)");
        }
```

```
}

void
receive_audio(struct client *ac, int lock, size_t blksize)
{
        if (ac->ipcstyle == SOCKET) {
                if (fread(ac->out, blksize, 1, ac->audio) <= 0)
                        errx(1, "read from client");
        } else {
                if (flock(ac->semfds[lock], LOCK_EX) < 0)
                        err(1, "flock (LOCK_EX)");
        }
}

void
mix_clientoutputs(short *out, struct clients *clist, size_t blksize)
{
        struct client *ac;
        int i;

        LIST_FOREACH(ac, clist, link) {
                for (i = 0; i < blksize / sizeof(short); i++)
                        out[i] += ac->out[i] / MIXDIVIDE;
        }
}

void
send_fds(int sckt, int *fds, int fdcount)
{
        struct iovec empty;
        struct msghdr msg;
        struct cmsghdr *fdmsg;
        struct {
                struct cmsghdr hdr;
                int fd[FDSENDMAX];
        } fddata;
        int i;

        if (fdcount > FDSENDMAX)
                errx(1, "too many file descriptors for send_fds");

        empty.iov_base = &empty;
        empty.iov_len = 1;

        msg.msg_name = NULL;
        msg.msg_namelen = 0;
        msg.msg_iov = &empty;
        msg.msg_iovlen = 1;
        msg.msg_control = &fddata;
        msg.msg_controllen = sizeof(struct cmsghdr) + fdcount * sizeof(int);
        msg.msg_flags = 0;

        fdmsg = CMSG_FIRSTHDR(&msg);
```

```
        fdmsg->cmsg_len = msg.msg_controllen;
        fdmsg->cmsg_level = SOL_SOCKET;
        fdmsg->cmsg_type = SCM_RIGHTS;

        for (i = 0; i < fdcount; i++)
                ((int *) CMSG_DATA(fdmsg))[i] = fds[i];

        if (sendmsg(sckt, &msg, 0) < 0)
                err(1, "sending access rights");
}                                                                    480
```

B Client source code

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <sys/uio.h>
#include <sys/mman.h>
#include <unistd.h>
#include <stdlib.h>
#include <limits.h>
#include <stdio.h>
#include <util.h>
#include <fcntl.h>
#include <math.h>

#include "common.h"

struct client {
        enum ipc_used ipcstyle;
        FILE *audio, *control;
        int mmapfds[MMAPFDCOUNT], semfds[SEMFDCOUNT];
        short *in, *out;
        size_t blksize;
};

int contact_server(void);
struct client * connect_to_server(void);
void handle_connection(struct client *);
void receive_audio(struct client *, int);
void process_audio(struct client *);
void send_audio(struct client *, int);
void recv_fds(int, int *, int);

int
main(int argc, char **argv)
{
        struct client *ac;
        int ctrlsckt;
        extern char *malloc_options;

        if (mlockall(MCL_CURRENT|MCL_FUTURE) < 0)
                warn("locking process pages to physical memory failed");

        malloc_options = "X";

        ac = connect_to_server();

        handle_connection(ac);
```

```
        return 0;
}
                                                                        50
struct client *
connect_to_server()
{
        struct sockaddr_un server;
        struct client *ac;
        int ctrlsckt, audiosckt, i;
        char file[PATH_MAX], *line;

        ac = malloc(sizeof(struct client));
                                                                        60
        /* establish a control socket connected to server,
         * and receive an audio data socket */

        if ((ctrlsckt = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
                err(1, "socket");
        server.sun_family = AF_UNIX;
        strlcpy(server.sun_path, ACCEPTSOCKET, sizeof(server.sun_path));
        if (connect(ctrlsckt, (struct sockaddr *) &server,
            sizeof(struct sockaddr_un)) < 0)
                err(1, "connect");                                      70
        recv_fds(ctrlsckt, &audiosckt, 1);

        /* create control stream */

        if ((ac->control = fdopen(ctrlsckt, "r+")) == NULL)
                err(1, "fdopen ctrlsckt");
        if (setvbuf(ac->control, NULL, _IOLBF, 0) != 0)
                errx(1, "could not set control stream to linebuffering mode");

        /* receive protocol information (ipcstyle and blocksize) */     80

        if ((line = fparseln(ac->control, NULL, NULL, NULL, 0)) == NULL)
                err(1, "fparseln");
        if (strcmp(line, PROT_SOCKET) == 0)
                ac->ipcstyle = SOCKET;
        else if (strcmp(line, PROT_SHM) == 0)
                ac->ipcstyle = SHAREDMEMORY;
        else
                errx(1, "received protocol is strange");
        free(line);                                                     90
        if ((line = fparseln(ac->control, NULL, NULL, NULL, 0)) == NULL)
                err(1, "fparseln");
        if ((ac->blksize = strtonum(line, 1, BLKSIZE_MAX, NULL)) == 0)
                errx(1, "received blocksize is strange");
        free(line);

        /* now that we have the blocksize, we create the audio stream */

        if ((ac->audio = fdopen(audiosckt, "r+")) == NULL)
```

```
                err(1, "fdopen audiosckt");
        if (setvbuf(ac->audio, NULL, _IOFBF, ac->blksize) != 0)
                errx(1, "could not set audio stream blocksize");

        /* receive client directory information,
         * and hook up shared memory and semaphore files to client */

        if ((line = fparseln(ac->control, NULL, NULL, NULL, 0)) == NULL)
                err(1, "fparseln");
        for (i = 0; i < MMAPFDCOUNT; i++) {
                snprintf(file, sizeof(file), "%s/shm.%d", line, i);
                if ((ac->mmapfds[i] = open(file, O_RDWR, 0644)) < 0) /* XXX */
                        err(1, "open a shared memory file");
        }
        for (i = SEMONE; i < SEMFDCOUNT; i++) {
                snprintf(file, sizeof(file), "%s/sem.%d", line, i);
                if ((ac->semfds[i] = open(file, O_RDWR, 0644)) < 0)
                        err(1, "open a semaphore file");
        }
        if (flock(ac->semfds[SEMTWO], LOCK_EX) < 0)
                err(1, "flock");
        ac->in = mmap(0, ac->blksize, PROT_READ|PROT_WRITE,
                MAP_SHARED, ac->mmapfds[MMAPINPUT], 0);
        ac->out = mmap(0, ac->blksize, PROT_READ|PROT_WRITE,
                MAP_SHARED, ac->mmapfds[MMAPOUTPUT], 0);
        free(line);

        /* tell server we are ready */
        fprintf(ac->control, "%s\n", READYMSG);

        return ac;
}

void
handle_connection(struct client *ac)
{
        int nolock, unlock, lock, tmp;

        nolock = SEMONE; unlock = SEMTWO; lock = SEMTHREE;

        /* XXX we assume the server works correctly */

        for (;;) {
                receive_audio(ac, lock);
                process_audio(ac);              /* we have two locks */
                send_audio(ac, unlock);
                tmp = nolock; nolock = unlock; unlock = lock; lock = tmp;
        }

        /* NOTREACHED */
}

void
```

```
receive_audio(struct client *ac, int lock)
{
        if (ac->ipcstyle == SOCKET) {
                if (fread(ac->in, ac->blksize, 1, ac->audio) <= 0)
                        err(1, "read from server");
        } else {
                if (flock(ac->semfds[lock], LOCK_EX) < 0)
                        err(1, "flock");
        }
}

void
process_audio(struct client *ac)
{
        int i;

        for (i = 0; i < ac->blksize / sizeof(short); i++)
                /* ''process'' only every fourth sample, gives a nice load */
                ac->out[i] = i % 4 == 0 ?
                                powf(cbrtf(ac->in[i]), 3) :
                                ac->in[i];
}

void
send_audio(struct client *ac, int unlock)
{
        if (ac->ipcstyle == SOCKET) {
                if (fwrite(ac->out, ac->blksize, 1, ac->audio) <= 0)
                        err(1, "write to server");
        } else {
                if (flock(ac->semfds[unlock], LOCK_UN) < 0)
                        err(1, "flock (LOCK_UN)");
        }
}

void
recv_fds(int sckt, int *fds, int fdcount)
{
        struct iovec empty;
        struct msghdr msg;
        struct cmsghdr *fdmsg;
        struct {
                struct cmsghdr hdr;
                int fd[FDSENDMAX];
        } fddata;
        int i;

        if (fdcount > FDSENDMAX)
                errx(1, "too many file descriptors for recv_fds");

        empty.iov_base = &empty;
        empty.iov_len = 1;
```

```
        msg.msg_name = NULL;
        msg.msg_namelen = 0;
        msg.msg_iov = &empty;
        msg.msg_iovlen = 1;
        msg.msg_control = &fddata;                                          210
        msg.msg_controllen = sizeof(struct cmsghdr) + fdcount * sizeof(int);
        msg.msg_flags = 0;

        fdmsg = CMSG_FIRSTHDR(&msg);
        fdmsg->cmsg_len = msg.msg_controllen;
        fdmsg->cmsg_level = SOL_SOCKET;
        fdmsg->cmsg_type = SCM_RIGHTS;

        for (i = 0; i < fdcount; i++)
                ((int *) CMSG_DATA(fdmsg))[i] = -1;                         220

        if (recvmsg(sckt, &msg, MSG_WAITALL) < 0)
                err(1, "receiving file descriptors");

        for (i = 0; i < fdcount; i++)
                fds[i] = fddata.fd[i];
}
```

C Other files

C.1 Common definitions for server and client

```
#define ACCEPTSOCKET (AUDIODDIR "/accept")
#define AUBLKSIZE 1024
#define AUDIODDIR "/tmp/audiod"
#define BLKSIZE_MAX (1 << 15)

#define FDSENDMAX 64

#define PROT_SOCKET "socket"
#define PROT_SHM "shared memory"
#define READYMSG "client ready"

enum mmapfds { MMAPINPUT, MMAPOUTPUT, MMAPFDCOUNT };
enum semfds { SEMONE, SEMTWO, SEMTHREE, SEMFDCOUNT };
enum ipc_used { SOCKET, SHAREDMEMORY };
```

C.2 Patch to OpenBSD kernel

```
--- sys/arch/i386/include/param.h   Sat Aug 7 05:21:04 2004
+++ sys/arch/i386/include/param.h   Mon May 30 20:15:16 2005
@@ -100,7 +100,7 @@
 #define        USPACE_ALIGN (0)          /* u-area alignment 0-none */

 #ifndef MSGBUFSIZE
-#define MSGBUFSIZE 4*NBPG      /* default message buffer size */
+#define MSGBUFSIZE 1024*NBPG /* default message buffer size */
 #endif

 /*
--- sys/dev/audio.c  Sat Jul 10 20:29:22 2004
+++ sys/dev/audio.c  Sun May 29 10:29:50 2005
@@ -2020,6 +2020,7 @@ audio_pint(v)
        int cc, ccr;
        int blksize;
        int error;
+       static int blockid = 0;

        if (!sc->sc_open)
                return;         /* ignore interrupt if not open */
@@ -2068,6 +2069,7 @@ audio_pint(v)
 #endif
```

```
        cb->used -= blksize;
+       printf("ALD used=%d id=%d\n", cb->used, blockid++);
        if (cb->used < blksize) {
                /* we don't have a full block to use */
                if (cb->copying) {
```

C.3 Test hardware configuration (dmesg)

```
OpenBSD 3.7 (AUDIOLATENCYDEBUG) #0: Mon May 30 21:19:06 EEST 2005
    je@zero.purplesea.net:/home/je/share/university/thesis/sys/arch/i386/compile/AUDIOLATENCYDEBUG
cpu0: Intel Celeron ("GenuineIntel" 686-class, 128KB L2 cache) 434 MHz
cpu0: FPU,V86,DE,PSE,TSC,MSR,PAE,MCE,CX8,SEP,MTRR,PGE,MCA,CMOV,PAT,PSE36,MMX,FXSR
real mem = 133787648 (130652K)
avail mem = 111476736 (108864K)
using 1658 buffers containing 6791168 bytes (6632K) of memory
mainbus0 (root)
bios0 at mainbus0: AT/286+(75) BIOS, date 08/18/99, BIOS32 rev. 0 @ 0xf0530
apm0 at bios0: Power Management spec V1.2 (BIOS mgmt disabled)
apm0: APM power management enable: unrecognized device ID (9)
apm0: APM engage (device 1): power management disabled (1)
apm0: AC on, battery charge unknown
pcibios0 at bios0: rev 2.1 @ 0xf0000/0xbe2
pcibios0: PCI IRQ Routing Table rev 1.0 @ 0xf0b60/128 (6 entries)
pcibios0: PCI Interrupt Router at 000:04:0 ("VIA VT82C586 ISA" rev 0x00)
pcibios0: PCI bus #2 is the last bus
bios0: ROM list: 0xc0000/0x8000
cpu0 at mainbus0
pci0 at mainbus0 bus 0: configuration mode 1 (no bios)
pchb0 at pci0 dev 0 function 0 "VIA VT82C691 PCI" rev 0x22
ppb0 at pci0 dev 1 function 0 "VIA VT82C598 AGP" rev 0x00
pci1 at ppb0 bus 1
vga1 at pci1 dev 0 function 0 "ATI Rage Pro" rev 0x5c
wsdisplay0 at vga1: console (80x25, vt100 emulation)
wsdisplay0: screen 1-5 added (80x25, vt100 emulation)
pcib0 at pci0 dev 4 function 0 "VIA VT82C596A ISA" rev 0x09
pciide0 at pci0 dev 4 function 1 "VIA VT82C571 IDE" rev 0x06: ATA33, channel 0 configured to compatibility,
wd0 at pciide0 channel 0 drive 0: <Maxtor 91021U2>
wd0: 16-sector PIO, LBA, 9770MB, 20010816 sectors
wd1 at pciide0 channel 0 drive 1: <ST33232A>
wd1: 16-sector PIO, LBA, 3077MB, 6303024 sectors
wd0(pciide0:0:0): using PIO mode 4, Ultra-DMA mode 2
wd1(pciide0:0:1): using PIO mode 4, Ultra-DMA mode 2
atapiscsi0 at pciide0 channel 1 drive 0
scsibus0 at atapiscsi0: 2 targets
cd0 at scsibus0 targ 0 lun 0: <PLEXTOR, CD-R PX-320A, 1.03> SCSI0 5/cdrom removable
atapiscsi1 at pciide0 channel 1 drive 1
scsibus1 at atapiscsi1: 2 targets
st0 at scsibus1 targ 0 lun 0: <HP, COLORADO 8GB, 2.06> SCSI2 1/sequential removable
```

```
st0: drive empty or not ready
cd0(pciide0:1:0): using PIO mode 4, Ultra−DMA mode 2
st0(pciide0:1:1): using PIO mode 4, DMA mode 2
uhci0 at pci0 dev 4 function 2 "VIA VT83C572 USB" rev 0x02: irq 5
usb0 at uhci0: USB revision 1.0
uhub0 at usb0
uhub0: VIA UHCI root hub, class 9/0, rev 1.00/1.00, addr 1
uhub0: 2 ports with 2 removable, self powered
ppb1 at pci0 dev 4 function 3 "VIA VT82C596 Power Mgmt" rev 0x00
ppb1: not configured by system firmware                                    50
eap0 at pci0 dev 9 function 0 "Ensoniq AudioPCI97" rev 0x06: irq 5
ac97: codec id 0x43525913 (Cirrus Logic CS4297A rev 3)
ac97: codec features headphone, 20 bit DAC, 18 bit ADC, Crystal Semi 3D
audio0 at eap0
midi0 at eap0: <AudioPCI MIDI UART>
xl0 at pci0 dev 11 function 0 "3Com 3c900 10Mbps−Combo" rev 0x00: irq 10, address 00:60:97:58:c0:03
isa0 at pcib0
isadma0 at isa0
pckbc0 at isa0 port 0x60/5
pckbd0 at pckbc0 (kbd slot)                                                60
pckbc0: using irq 1 for kbd slot
wskbd0 at pckbd0 (mux 1 ignored for console): console keyboard, using wsdisplay0
pms0 at pckbc0 (aux slot)
pckbc0: using irq 12 for aux slot
wsmouse0 at pms0 mux 0
pcppi0 at isa0 port 0x61
midi1 at pcppi0: <PC speaker>
sysbeep0 at pcppi0
lpt0 at isa0 port 0x378/4 irq 7
lm0 at isa0 port 0x290/8: W83781D                                          70
npx0 at isa0 port 0xf0/16: using exception 16
pccom0 at isa0 port 0x3f8/8 irq 4: ns16550a, 16 byte fifo
pccom1 at isa0 port 0x2f8/8 irq 3: ns16550a, 16 byte fifo
fdc0 at isa0 port 0x3f0/6 irq 6 drq 2
fd0 at fdc0 drive 0: 1.44MB 80 cyl, 2 head, 18 sec
biomask eb65 netmask ef65 ttymask ffe7
pctr: 686−class user−level performance counters enabled
mtrr: Pentium Pro MTRR support
dkcsum: wd0 matched BIOS disk 80
dkcsum: wd1 matched BIOS disk 81                                           80
root on wd0a
rootdev=0x0 rrootdev=0x300 rawdev=0x302
```